

Corso completo di programmazione assembler in due dischi

Copyright (c) 2016 Fabio Ciucci, Alessandro Sperindé, Stefania Calcagno
 è garantito il permesso di copiare, distribuire e/o modificare
 questo documento seguendo i termini della Licenza per
 Documentazione Libera GNU, Versione 1.3 o ogni versione
 successiva pubblicata dalla Free Software Foundation; con le
 Sezioni Non Modificabili

* "Prefazione"

con i Testi Copertina

- * "Corso completo di programmazione assembler in due dischi"
- * "Per Amiga 500, 500+, 600 e 1200"
- * "di Fabio Ciucci aka Randy/RAMJAM"
- * "Proofreading, update dei testi e pagina della versione 2016 del corso
 a cura di Alessandro Sperindé aka Trantor/RamJam"
- * "Prefazione, assemblaggio e immagini delle lezioni, sito internet, testi
 2016 a cura di Stefania Calcagno aka Yuki/RamJam"
- * "RAMJAM"

e con il seguente Testo di Retro copertina:

- * "Corso completo di programmazione assembler in due dischi"
- * "di Fabio Ciucci aka Randy / Ram Jam"
- * "Proofreading, update dei testi e pagina della versione 2016 del corso a cura di Alessandro Sperindé aka Trantor / Ram Jam"
- * "Prefazione, assemblaggio e immagini delle lezioni, sito Internet, testi 2016 di Stefania Calcagno aka Yuki / Ram Jam"
- * "Questo libro è la riedizione moderna del 2016 su carta e in formato E-Book del famoso Corso Completo di Programmazione Assembler in 2 Dischi per Amiga distribuito su Floppy Disk direttamente dall'autore negli anni '90. I dischi originali con le lezioni assemblabili e la versione completa in E-Book con 1466 pagine di parte pratica, composta da sorgenti commentati, con immagini del risultato assemblato, è disponibile nel CD ROM allegato. Sono anche scaricabili gratuitamente dal sito: <http://corsodiassembler.ramjam.it> Dopo 22 anni dalla sua uscita il Corso Completo di Programmazione Assembler in 2 dischi per Amiga ha una nuova veste, e può essere consultato da tutti gratuitamente in versione E-Book.

Le motivazioni che ci hanno condotto a questo remake sono diverse e riassumibili in due principali argomentazioni:

- Permettere agli appassionati che vogliono cimentarsi nella realizzazione di una Demo o un gioco sulla meravigliosa piattaforma Amiga di usufruire di un corso completo di Assembler. Da "Hello World" alle complesse routines 3d.
- Mostrare alla nuove generazioni come si programmava nel tempo in cui il programmatore era più importante della CPU. Raccontare la passione che animava la "scena demo" e che ha contagiato un'intera generazione di sviluppatori.

Nella prefazione del 2016, Fabio Ciucci ci racconta di come ci si potesse sentire "sovranaturali" realizzando degli effetti strabilianti con pochissime risorse. Stefania Calcagno ci racconta di come nacque il gruppo Ram Jam, della scena demo e del testamento spirituale di Fabio per tutti gli utilizzatori di un Amiga: Massive Killing Capacity."

- * "RamJam - Taste the Difference!"

Una copia della licenza è acclusa nella sezione intitolata "Licenza per Documentazione Libera GNU".

INDICE

Indice	i
Prefazione	v
Fabio <i>Randy</i> Ciucci	v
Stefania <i>Yuki</i> Calcagno	vii
Massive Killing Capacity	viii
I Teoria	1
1 Lezione 1	3
2 Lezione 2	21
3 Lezione 3 - La prima CopperList	33
4 Lezione 4 - Immagini sullo schermo	41
5 Lezione 5 - Lo scrolling orizzontale e verticale	55
6 Lezione 6 - I Fonts e lo scrolling	63
7 Lezione 7 - Gli Sprites e i dispositivi di input	73
7.1 I colori degli sprite	76
7.2 La priorità video tra gli sprite	79
7.3 Sprite “attached”	79
7.4 Mouse e joystick	85
7.5 Riutilizzo degli sprite	87
7.6 Il dual playfield mode	90
7.7 Priorità tra sprite e playfield	92
7.8 Collisioni	93
7.9 Uso diretto dei registri degli sprite	97
7.10 Animazione sprite	100
8 Lezione 8 - Approfondimenti sul 68000	105
9 Lezione 9 - Il Blitter	135
9.1 IL BLITTER	135
9.2 Funzioni del blitter	135
9.3 Prima applicazioni del blitter	139

9.4	Blittate “a colori”	148
9.5	Maschere	155
9.6	Copia di zone di memoria sovrapposte	158
10	Lezione 10 - Blitter Avanzato	165
10.1	I MINTERMs	165
10.2	I BOBs	169
10.3	La velocità del Blitter (e non solo)	172
10.4	Il double buffering	175
10.5	Uso dei canali Blitter non attivati	176
10.6	Il flag Zero e le collisioni	177
10.7	Il SinusScroll	178
10.8	Animazione	180
10.9	I modi speciali del Blitter	181
10.10	Modo di Riempimento aree	184
11	Lezione 11 - Interrupt, CIAA/CIAB, DOSLib	189
11.1	Il registro VBR nei processori 68010 e superiori	193
11.2	Come si costruisce una routine di interrupt	195
11.3	Come si usano INTENA ed INTENAR	195
11.4	Come si usano INTREQ e INTREQR	197
11.5	Gli interrupt e il sistema operativo	199
11.6	Gli interrupt COPER chiamati da COPPERList	201
11.7	Informazioni avanzate sul COPPER - Uso del solo COLOR0 (\$180) - No BitPlanes	201
11.8	Uso della COPPER2 (COP2LC/COPJMP2)	202
11.9	Informazioni avanzate sul COPPER - Anche i BitPlanes abilitati	203
11.10	Come fare uno schermo in interlace (lungo 512 linee)	205
11.11	I due chip 8520, detti CIAA e CIAB	206
11.12	Routine di interrupt \$68 (livello 2) - gestione tastiera	213
11.13	I timers del CIAA e del CIAB	215
11.14	Il caricamento di files con la dos.library	215
11.15	AllocMem	219
11.16	FreeMem	220
12	Lezione 12 - La compatibilità del codice	221
12.1	Errori riguardanti COPPERList e Blitter	222
12.2	Errori riguardanti CIAA/CIAB, tastiera, timers, trackloaders	225
12.3	Errori riguardanti i processori 68010/20/30/40/60	226
13	Lezione 13 - Ottimizzazione del codice assembly	239
13.1	Ottimizzazioni di primo livello: lo “scambio” e la “scelta” di istruzioni	241
13.2	Ottimizzazioni di secondo livello: il “tabellamento”	253
13.3	Ottimizzazioni varie - gruppo misto	256
13.4	Ottimizzazioni del Blitter	268
14	Lezione 14 - Fondamenti di acustica e audio digitale	271
14.1	Le replay routines sofisticate (Autore: Fabio Ciucci)	280
15	Lezione 15 - Il chipset AGA	285
15.1	La nuova palette a 24 bit	288

15.2	I nuovi modi a 128 e 256 colori	289
15.3	FMode	291
15.4	HAM8	294
15.5	Sprite	295
15.6	Nuovo posizionamento orizzontale ad 1/4 di pixel	298
15.7	Il bit BRDRSPRT	298
15.8	Il modo attached	298
15.9	La palette degli sprite AGA	299
15.10	Nuovo scroll orizzontale superfluido (1/4 di pixel) per i bitplanes	300
15.11	Una nuova possibilità per ciclare la palette	301
15.12	Dual playfield AGA	303
15.13	VGA/Productivity 640x480 non interlacciata	303
15.14	Collisioni	306
15.15	Blitter ECS+	306
 II Pratica		 308
16 Dalla teoria alla pratica		311
16.1	Introduzione	311
 17 Licenza d'uso		 313
17.1	Introduzione	313
17.2	BSD 3-clause	314
 III Approfondimenti		 315
18 Texture Mapping		317
18.1	Premessa	317
18.2	Cenni sul formato dei numeri in virgola fissa	318
18.3	Cos'è il Texture Mapping	319
18.4	Come viene implementato il texture mapping in applicazioni real-time?	320
18.5	Il calcolo della scena da tracciare	324
18.6	Saliamo le scale!	328
18.7	Illuminiamo il nostro mondo	330
18.8	Texture mapping e Amiga	332
 19 Real-Time Computer Graphics 3D		 337
19.1	Prefazione	337
19.2	Parte 0: Cenni di grafica 2D relativa agli 80x86 e VGA	338
19.3	Parte 1: Geometry Engine	341
19.4	Appendici	345
19.5	Note finali	352
19.6	Rotazione	353
19.7	Ottimizzazione delle rotazioni	354
19.8	Wireframe	355
19.9	Hidden Face	356
19.10	Algoritmo del pittore	357
19.11	Filled vector e scan-line	358

19.12 Flat shading	361
19.13 Ottimizzazioni per il calcolo della sorgente di luce	363
19.14 Gouraud shading	364
19.15 Phong shading	365
19.16 Reflection mapping	366
19.17 Affine texture mapping	367
19.18 Ottimizzazione del fill	368
19.19 Clipping 2D	369
19.20 Z-Buffer	372
19.21 Bump mapping 2D	373
19.22 Notazione in virgola fissa	374
19.23 Coordinate polari	375
19.24 Gestione degli oggetti	376
IV Appendici	378
A Matematica	381
B Licenza per Documentazione Libera GNU	387

PREFAZIONE

Fabio Randy Ciucci

Inizio a scrivere una prefazione del corso asm dopo 22 anni dalla sua uscita, gli stessi giorni che il Javascript, un linguaggio nemmeno compilato, si è confermato il più popolare del 2015 sia in *StackExchange* che *GitHub*, non solo per web app ma per server con Node e mobile apps.

Nessuno fa caso che ogni progetto tipo *Angular* o altro produce all'incirca 50-100MB di cartella *node_modules* ad ogni *npm install*, anche per fare un bottone “hello world”. La prima volta che provai, esclamai: “Ma stiamo scherzando!?”. Non potevo credere che la tecnologia più moderna e in voga del momento, per 4 sorgenti di prova in 4 cartelle, genera 4 copie di librerie quasi uguali, ognuna da 100MB con migliaia di files opensource di dubbia coerenza e utilità, e che solo a me sembrava inefficiente.

L'aver scritto *4kb intros* da piccolo mi fa dimenticare che 400MB sul mio HD SSD turbo da 512GB occupano proporzionalmente meno di una 4kb intro in un dischetto da 700k. E che tutto va in diretta su 8 core a 3Ghz con 16GB RAM e Nvidia spaziale... e sto parlando di un portatile. Pure i telefoni sono così potenti che con *Cordova/Phonegap* gli stessi impasti di *Javascript* e *CSS* sono sempre più usati per fare app Android e iOS, così da non usare Java o Objective C. Tranne che per i giochi più complicati, per quelli se non altro su usa sempre il C++, anche se gli engine 2d e 3d sono scritti da altri, i quali a loro volta chiamano OpenGL e simili che chiamano le schede grafiche. Insomma altre migliaia di files di cui ognuno sa solo il proprio piccolo pezzo pertinente. In asm dai e togli letteralmente la corrente ai circuiti integrati che ti pare, per i millisecondi che ti pare, affinché arrivi corrente al tale punto nel monitor, al tale motore del disk drive, al tale altoparlante. Non chiami librerie di altri se non all'inizio per dire gentilmente al sistema operativo di togliersi di torno che tanto non serve. Il codice è solo il tuo, salvo se hai rippato, ovvero copia e incollato dei pezzi, ma comunque è tutto e solo nel tuo sorgente, questo controllo assoluto e conoscenza totale fa sentire *sovrannaturali*. Una sensazione che non si sente quando scrivi 100 linee che chiamano un altro milione di linee tra API e sistemi operativi di cui ignori i dettagli e che, diciamocelo, spesso sono piene di bug e le versioni vanno in conflitto tra loro e via dicendo.

D'altronde, il software non ha lo scopo di farti sentire sovrannaturale, ma invece deve risolvere problemi in tempi e costi ridotti, e oggi costa più il tempo del programmatore dello spazio hard disk o velocità CPU/GPU. Infatti, da molti anni scrivo le mie cose in Python quando possibile, perché (a differenza del Javascript) si fa veramente presto, e spesso si rimane compiaciuti della maggiore eleganza e brevità rispetto a C++ o Java, dei cui grovigli di parentesi graffe e cicli “for” si fa certo a meno.

Il più grande cambiamento non è quello che tutti notano ovvero l'hardware e i linguaggi, bensì internet e opensource: ai tempi in cui rilasciai il corso ASM, era l'unica fonte completa di informazioni sull'argomento in italiano. E comunque anche in inglese i libri erano pochi ed evasivi, e soprattutto nessuno rilasciava i codici sorgenti, spesso neppure ai membri dello stesso gruppo. Il codice era una cosa esoterica Elite da tenere segreta sia ai Lamer che alla

concorrenza, esterna e pure interna. L'unica era *decompilare*, una volta per un dramma in Ram Jam che alcuni non davano il sorgente agli altri mi è stato chiesto gentilmente di disassemblare un magazine per poi updatarlo ad AGA, un lavoro che sembra assurdo a pensarci oggi, ma era normale. Fece sia scandalo, perché era un *rippaggio* seppure semi autorizzato, sia ammirazione, perché l'operazione di *hackeraggio* e l'update AGA non erano una *lamerata* come chi copia incolla una scritta scorrevole. Infatti non nascondo che ho anche *crackato* e *trainerato* assai (tanto ormai è in prescrizione), e un decompilato, premesso che di solito non parte, non ha nomi label né commenti né la struttura o formattazione dei moderni opensource a API a cui tutti sono abituati, magari in IDE che mettono in sovraimpressione la documentazione di ogni funzione passandoci sopra col mouse. E comunque qualsiasi problema o domanda la googli, e trovi sempre qualcuno che l'ha risposta. Oggigiorno i programmatori sono degli artisti del copia incolla da StackExchange e simili, io stesso googlo tanto, proprio ricordandomi i tempi in cui non c'era documentazione e l'unica era fare dei loop che provavano tutti i valori in un registro e vedere cosa succedeva, come quando è uscito l'AGA ma la Commodore si è dimenticata di documentarlo.

Tutto è cambiato, ma sempre attuale è il successo di ex demo scener nel passare ai video giochi, essendo le competenze simili e trasferibili. L'esempio supremo, ovviamente in Finlandia, sono i Virtual Dreams (<http://demozoo.org/groups/609/>), (<http://www.pouet.net/groups.php?which=201>), la sezione demo dei Fairlight, che vincono l'Assembly party nel 1996 con la demo "Sumea", da cui nel 2001 la società "Sumea" per fare giochi per cellulari (come me negli stessi tempi, con ex scener meno nordici). Poi nel 2010 Supercell ([https://en.wikipedia.org/wiki/Supercell_\(video_game_company\)](https://en.wikipedia.org/wiki/Supercell_(video_game_company))), in 6 persone (ma quasi nessuno dei coder Amiga originari, persi per strada) investono 250mila euro, e fanno i miliardi. Letteralmente, 2 miliardi di euro le vendite del 2015, e 180 dipendenti, grazie a "Clash of Clans" con 100 milioni di giocatori. Anche Rovio, creatore finnico di Angry Birds, brulica di ex scener. In realtà contano molto banche e venture capital in queste cose, per questo in Italia non succede anche ai coder sovranaturali. In compenso un italiano emigrato in UK (non un coder) ha fondato King, la società di Candy Crush, venduta nel 2015 per 5 miliardi di dollari. Questi signori non avrebbero fatto i miliardi se fossero rimasti a fare Amiga asm, d'altronde alcuni di loro non avrebbero fatto i miliardi se non avessero precedentemente conosciuto l'asm.

Sono certo che non leggete il corso per fare soldi con l'asm, ma per provare a sentirvi con poteri retro-sovrannaturali, e chi sa, indirettamente la rara conoscenza di asm che opera in background, potrebbe essere la differenza che vi fa fare i miliardi con altre tecnologie contemporanee. Infatti spesso mi scrivono sconosciuti che fanno di tutto tranne l'asm Amiga, ringraziandomi che tot anni fa, iniziarono proprio con questo mio corso che, se chiedete a un accademico, vi dirà insegna gli "anti-pattern" ovvero il contrario di come bisognerebbe programmare nel 2016.

Stefania Yuki Calcagno

Nel 1989 in un uggioso pomeriggio invernale eravamo a casa di Paolo *Levin*, a Savona. A quel tempo avevamo entrambi nick differenti, ma in questa prefazione userò solo quelli nuovi, *Yuki* per me e *Levin* per Paolo. Insieme decidemmo di formare un nuovo gruppo Amiga, completamente nuovo per la scena Italiana. Con le nostre BBS, con i propri coders, grafici, swappers, musicisti. Io ero una ‘coder’, Levin un ‘sysop’. Entrambi già conosciuti e con un passato nel Commodore 64 e in altri gruppi Amiga.

L’idea era di fare un gruppo tutto Italiano che potesse competere con i più strutturati e famosi gruppi Nordici (Svezia, Norvegia, Danimarca, Finlandia e Germania erano il Top all’inizio della scena Amiga). In realtà la cosa ci sfuggì un poco di mano e diventammo uno dei più produttivi e famosi gruppi mondiali, con sezioni in USA, Francia, Germania, Svezia, Norvegia, Svizzera e membri addirittura in Turchia e Ungheria.

Quel pomeriggio io e Paolo *Levin* decidemmo di chiamare questo gruppo **Ram Jam**. Perché suonava bene “*Marmellata di Memoria ad Accesso Casuale*”, tradotto da un gioco di parole in Inglese. E anche perché la musica Indie e un po’ new wave dei Ram Jam, gruppo musicale Inglese degli anni 70-80, che cantavano la splendida Black Betty ci piaceva non poco!

Tra i primi membri, tutti liguri: eravamo amici e vicini all’inizio dell’avventura, ricordo Roberto *Bar*, Andrea *Executor* e Piero *Grizzly*, il mitico Sergio *Deus* unico non ligure della prima guardia, capostipite dell’avvento della mitica sezione Marchigiana – Abruzzese – Molisana, che ha contribuito a scrivere pezzi di storia dell’Amiga! Fabio *Randy*, l’autore di questo fantastico corso di Assembler, scritto con un editor di testi ASCII su un Amiga nei primi anni ’90, si unì poco dopo alla ciurma.

Fabio è stato sicuramente uno dei migliori coders della scena Amiga Italiana e non, e sicuramente il più conosciuto tra i RamJammers. Insieme ad Andrea *Executor*, Pietro *Darkman*, Danilo *M.a.s.e.*, Fabio *Maverick*, Piergiorgio *Zibri* e Massimo *Mr.Buck* (i nostri coders Italiani, questo è un libro Italiano, non me ne vorranno i nostri fratelli in giro per il mondo!), Fabio *Randy* ha contribuito in modo importante alle produzioni Ram Jam ed a rendere la scena Amiga, Italiana e no, un posto bellissimo!

Qualche tempo fa, ad Alessandro *Trantor* venne in mente una magnifica idea: riprendere il corso di Assembler degli anni 90 di Fabio e trasformarlo dall’originale ASCII Amiga in un formato impaginabile e stampabile. Sia per preservare questo lavoro magnifico, che per dare la possibilità a “nuovi adepti” di studiare l’assembler Amiga.

Nel 2016. Sì, nel 2016, proprio oggi tante persone vorrebbero cimentarsi a scrivere giochi o demo con questo vecchio codice macchina Motorola. Complesso, difficile, poco intuitivo ma, come dice Fabio, che da poteri sovrannaturali. Altri RamJammers hanno dato una mano a questo progetto, chi poco chi tanto: Nicola *HanTareX*, Giacomo *Jack Tory*, Dario *Rio*, Christian *DaGoN*. È grazie a iniziative come questa che la leggenda dell’Amiga continua a sopravvivere all’aumento di capacità di calcolo, di potenza grafica, di dimensione sia fisica (in piccolo) che logica (in grande) dei supporti di memoria.

Grazie a Fabio per averci raccontato negli anni 90 come sfruttare al massimo le capacità dei chip Amiga e per aver incarnato lo spirito hacker e di Ramjam “*knowledge is not a crime*”; e anche per aver dato il permesso di ripubblicare l’intera opera in versione moderna e renderla disponibile gratuitamente alla collettività.

Grazie ad Alessandro che ha fatto un immane lavoro per mettere insieme tutti i pezzi e renderli di nuovo fruibili da tutti.

Grazie a tutti quelli che hanno comunque collaborato, anche solo per esserci stati vicini!

Amiga will never die.

Stefania Yuki Calcagno

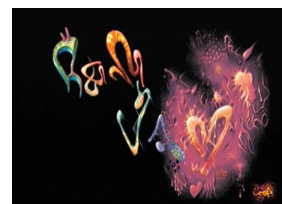
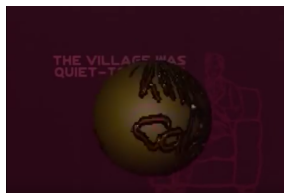
Massive Killing Capacity

Una delle più belle Demo della storia Italiana Amiga è stata sicuramente Massive Killing Capacity, per Amiga 1200 base, inespanso. L'ultima Demo di Fabio Randy per Amiga. Il suo testamento in Assembler. Lasciata ai posteri. Fondamentalmente è l'ultima e massima opera, nonché testamento spirituale di Fabio Ciucci.

In quest'ultima prefazione riportiamo alcuni screenshots di quella Demo, che può essere visualizzata tutta su YouTube cercando *Ram Jam Massive Killing Capacity* o semplicemente inserendo questo link: https://www.youtube.com/watch?v=vQqNDGJUu_8.



Resiste con una comunità enorme che continua a sviluppare tool, utility e giochi. Resiste con un numero incredibile di videogiocatori che adorano i titoli usciti per Amiga, la fluidità e la bellezza dei giochi stessi. Resiste, tant'è che oggi nel 2016, comprare un Amiga reale costa quasi di più di quando uscì sul mercato alla fine degli anni 80. Resiste con la scena che è sopravvissuta a se stessa e continua a fare i Demoscene party: ogni anno al **Revision**, in Germania c'è ancora una gara (*compo* in gergo, o *competition*) di Demo Amiga.



Per chi vuole è anche possibile scaricarla da qui: <http://www.ramjam.it/ramjam/down.htm> ed eseguirla direttamente sul proprio Amiga o su un emulatore come FS-UAE.

Probabilmente neanche alla fine del corso sarete in grado di fare effetti come questi, ma sicuramente avrete le basi per poter cercare di fare una Demo o anche un videogioco su Amiga. Che era ed è lo scopo del corso di Assembler di Fabio. Cercare di rendere possibile a tutti programmare questa splendida macchina che resiste ancora oggi a 20 anni dal fallimento di Commodore.



Anche Massive Killing Capacity fu presentata a un Demo Party, con relativa Competition. All'Assembly a fine 1996 In Finlandia. E successivamente in Italia al Gathering ad inizio 1997.

Buona lettura.

Stefania Yuki Calcagno e Fabio Randy Ciucci

Parte I

Teoria

CAPITOLO 1

LEZIONE 1

Per tutti coloro che hanno provato ad imparare a fare demo o giochi che sfruttino l'hardware di Amiga direttamente, ma non ci sono mai riusciti perché i libri erano scritti in maniera astratta e astrusa e i sorgenti di esempio, i listati cioè, erano poco commentati o troppo difficili, oppure per quelli che non ci hanno mai provato e si chiedono come si fa.

Devo ringraziare e salutare tutti coloro che hanno contribuito materialmente o moralmente alla realizzazione di questi due dischetti, in particolare:

- Luca Forlizzi (The Dark Coder)
- Andrea Fasce (Executor/RAM JAM)
- Sirio Zuelli (PROXIMA DESIGN)
- Alberto Longo (VIRTUAL DREAMS)

Nonché coloro che hanno testato le lezioni verificando se capivano o meno:

- Andrea Scarafoni
- Federico "GONZO" Stango, e altri.

Nella mia carriera di programmatore hobbista posso vantare la realizzazione di alcune demo/intro per delle BBS, ad esempio "AMILINK.EXE", per la banca dati AmigaLink, oppure per dei Club, come quella per il nuovo "Amiga Expert Team". Le mie "opere" maggiori sono la mia prima demo per il chipset AGA, il "World of Manga", che è stata pubblicata anche su alcune riviste, e il "Naos", che ho programmato per il gruppo Nova ACIES.

Devo precisare che sarebbe bene sapere almeno un poco di DOS prima di accingersi a leggere il mio corso, se non altro per sapere come salvare i listati! Dovreste aver trovato un manuale assieme all'Amiga... Comunque, in breve, nei dischi (sia Hard che Floppy) i dati sono immagazzinati in "file", ossia una serie di numerini uno dopo l'altro, che insieme possono formare file grafici, musicali, eseguibili, listati eccetera. Da notare che un dischetto vergine per poter essere

utilizzato deve essere *formattato*, altrimenti è impossibile scrivervi. Una volta formattato, ci si può salvare qualsiasi file, sia figure con programmi di grafica, che testi (come questo che state leggendo) che altro. Un file si può copiare da un disco ad un altro, si può cancellare, oppure gli si può cambiar nome, eccetera. In un disco ci possono molti file, fino a che non si riempiono gli 880Kb circa, magari con 2 file da 400Kb o con una trentina di file più piccoli. Da notare che all'interno di un disco, per fare un pò di ordine, si possono generare varie "subdirectory", ossia dei "cassetti" più piccoli, delle divisioni in cui mettere i file. Ad esempio si possono generare le subdir DISEGNI e TESTI, in cui copieremo o salveremo rispettivamente delle immagini e delle lettere per la fidanzata, in modo da non mettere disegni e testi insieme sciolti nella dir principale. È come se il disco fosse un armadio, e le subdir dei cassetti di questo armadio. Dato che si possono fare delle subdir dentro le subdir, ognuno di questi cassetti può contenere file sciolti o "scatole" con file o altre "scatoline" più piccole dentro. Dunque un sistema simile a quello dei mobili! Per eseguire le operazioni tra i file si può usare la *CLI/Shell*, in cui occorre scrivere dei comandi come:

Dir Elenca i file e le subdir contenute in un disco

Copy Copia i file

Delete Cancella un file (ATTENZIONE AD USARE QUESTO!!!)

Makedir Crea un "cassetto" (o subdir)

Oppure si può agire col mouse da WorkBench, dove i file sono "raffigurati" come icone e le subdir come cassetti. Da notare che il drive interno si chiama "df0:", quelli esterni "df1:", "df2:" eccetera. L'hardisk di solito si chiama "Dh0:" (o "Hd0:"). Un sistema più veloce è quello di usare utility come DiskMaster o DirOpus.

Dunque quando avrete scritto qualche listato, lo dovrete salvare in un disco formattato, o sull'Hard Disk in quale subdir. Altra cosa da sapere è come si fa a fare un disco "autoboot", ossia che parte automaticamente inserendolo nel drive all'accensione o dopo un reset. Supponiamo di aver salvato un nostro programma *eseguibile* in un dischetto, col nome "mioprogramma". Occorrerà fare una subdirectory "S", in cui salvare un file di testo col nome di startup-sequence, in cui sia scritto il nome del programma da caricare automaticamente:

mioprogramma

La *startup-sequence* si può scrivere (editare) anche con il programma con cui state leggendo, che fa anche da text-editor. Ultima cosa, occorre "installare" il disco in questione, digitando da cli/shell il comando:

Install df0:

Oppure `install df1:` se si inserisce il dischetto nel drive esterno. Detto questo, si può continuare con le note.

NOTA: Se volete installare il corso sull'hard disk, ricordatevi di copiare nella vostra directory S: il file "TRASH'M-ONE16.pref" che si trova nella directory S: del disco del corso.

NOTA2: Se volete stampare i listati, considerate che sono compressi col powerpacker, per cui vi serve il *PowerPacker Patcher*, quello usato in questo corso. (il file è quello chiamato PP nella directory C). Per installarlo, basta avere in LIBS: la `powerpacker.library` ed eseguire il comando PP. I listati saranno autoscompattati al caricamento.

In questo corso verranno trattati i vari argomenti della programmazione, come il *COPPER*, gli *SPRITE*, il *BLITTER*, nonché il nuovo chipset AGA e la programmazione della scheda video

PICASSO II. Nel disco 1 gli argomenti sono: 68000, copper, playfields e sprites. Il blitter, l'AGA e il resto sono nei dischi 2 e 3, non del tutto terminati.

Per quanto riguarda la distribuzione e la copia di questo corso, dovete sapere che è GiftWare/-Shareware e non propriamente di pubblico dominio. Con questo intendo che potete copiare ai vostri amici questo corso senza problemi, basta che non lo VENDIATE per soldi, dato che i diritti su questo corso sono dell'autore, cioè me, e non certo del primo furbacchione che vuole speculare sul lavoro altrui. D'altronde, se è vero che lo potete copiare e distribuire AL SOLO PREZZO DEI DISCHI VERGINI, dovete anche ricordarvi che se seguite con successo le varie lezioni, riuscendo a programmarvi qualche cosa, avete tratto giovamento dal mio lavoro, per cui **dovete** ringraziarmi in qualche modo, specialmente se diventate i programmatori più ricchi del mondo (beh, nell'eventualità. . . Questo ringraziamento è quantificabile a vostro piacere, preferisco biglietti da 10.000. L'eventuale afflusso di regalucci o, meglio, vile denaro, mi incoraggerebbe a proseguire l'hobby della programmazione Amiga, e anche a fare nuovi capitoli del corso.

Mi farete anche un grande favore se copierete a tutti i vostri amici questo disco 1 del corso, anche se a voi personalmente non interessasse, perché darete la possibilità a qualcun altro di averlo e di imparare a programmare. Ho deciso di scrivere un corso di ASM (assembler) perché 10000 persone me lo hanno chiesto, e considerato che lo faccio per divertimento l'ho scritto in maniera molto discutibile, ma, a mio avviso, risulterà più chiaro ai principianti i quali, una volta iniziato a capire, potranno continuare più approfonditamente. Chi è già esperto di ASM troverà divertenti le lezioni, magari ci troverà delle inesattezze, perciò gli consiglio di consultare direttamente i listati di esempio: questo corso è per chi parte da zero. Infatti, dalla mia esperienza personale e da quello che mi dicono gli aspiranti "CODER" (in gergo programmatori CATTIVI), il problema è proprio capire il tutto e fare i primi due o tre programmi, dopodiché si diviene in grado di continuare da soli. Mi propongo, dunque, di insegnare a far girare delle palline per lo schermo o a farci saltellare una scritta a chi non sa nemmeno cosa sia il 68000. Se poi costoro vorranno diventare programmatori di giochi ed entrare nel TEAM 17 basterà che continuino.

PER IMPARARE A PROGRAMMARE UN GIOCO TIPO GODS O PROJECT X O COMunque UN GIOCO CHE NON SIA UN SIMULATORE DI VOLO O UNO 3D, CHE INSOMMA NON COMPRENDA CUBETTI CHE RUOTANO, TUNNELL SINUSOIDALI O DISTORSIONI PROSPETTICHE, FRATTALI O TEXTURE MAPPING, GARANTISCO CHE BASTA AVERE LE COGNIZIONI DI MATEMATICA DI TERZA MEDIA.

Con questo voglio togliere dalla testa a tutti che la programmazione assembler dell'Amiga sia piena di matematica. *Io credo invece che non c'entri nulla.* Se si intende fare un programma di matematica, si deve conoscere la matematica, come se si vuol fare un gioco del calcio bisogna conoscere il calcio.

L'importante è conoscere come funziona l'Amiga, il suo processore (nel caso dell'Amiga un Motorola 68000) ed i suoi chip custom (ossia quelli dedicati a fare la grafica ed il suono). Personalmente ho fatto le superiori all'Istituto d'Arte della mia città, ed ho imparato a fare cosucce in ASM già quando ero alle medie, quindi basta usare bene il tempo che si tiene acceso l'Amiga, anziché giocarci: non serve frequentare la facoltà di informatica all'università, dove non insegnano certo a programmare giochi o demo sull'Amiga!!!

Ma perché imparare a programmare giochi o demo? E cosa sono le demo? Dunque, i giochi cosa sono lo sanno tutti, quindi si suppone che chi voglia imparare a programmarli si sia stancato di vedere giochi che non sono come vorrebbe, e si vuole fare il "SUO", come vuole lui, pixel per pixel. Per quanto riguarda le demo invece occorre fare una breve spiegazione. Demo sta per "demonstration", ossia dimostrazione grafica. Dimostrazione di cosa? Della potenza dell'Amiga e della bravura dei programmatori, naturalmente.

Comunque c'è qualcosa di più: *LA SCENA*. Non quella del teatro, ma l'*"AMIGA SCENE"* (in inglese, la lingua ufficiale della scena stessa). Immaginatevi la scena della musica: ci sono vari gruppi con cantanti, batteristi, eccetera. Per l'Amiga, invece, troviamo vari gruppi con *coder* (programmatori), *GFX artist* (grafici), *musicians* (musicisti), che invece di fare un "VIDEO" come quelli che fanno i gruppi della scena musicale come loro contributo, fanno una "DEMO", che si aggiunge alle altre fatte da altri gruppi in tempi e luoghi diversi. Ci sono poi gli *swapper* e i *trader* che sono rispettivamente coloro che scambiano e distribuiscono le demo via posta o via modem. . . costoro non producono niente, ma hanno una importanza nella scena, perché una cosa che non circola è come se non ci fosse. D'altronde, costoro aspirano a diventare *coder*, *grafici* o *musicisti*, per contribuire a fare una DEMO, anziché scambiare opere altrui.

Ci sono molti gruppi nell'*Amiga Scene*, che hanno membri in tutti il mondo, in particolare in Europa. I nomi di alcuni gruppi più famosi sono *ANDROMEDA*, *BALANCE*, *COMPLEX*, *ESSENCE*, *FAIRLIGHT*, *FREEZERS*, *MELON DEZIGN*, *POLKA BROTHERS*, *PYGMY PROJECTS*, *RAM JAM*, *SANITY*, *SPACEBALLS*. . . Da notare che ogni membro della scena si fa chiamare con uno pseudonimo, detto "handle". Insomma, un nome d'arte: per esempio due *coder* degli *ANDROMEDA* si fanno chiamare "Dr.Jekyll" e "Mr.Hyde", uno dei *FREEZERS* si fa chiamare "Sputnik", poi altri di vari gruppi sono: Hannibal, Dan, Paradroid, Dak, Wayne Mendoza, Performer, Bannasoft, Laxity, Vention, Psyonic, Slammer, Tron, Mr. Pet, Chaos, Lone Starr, Dr. Skull, Tsunami, Dweezil. . . Il nome completo si indica con l'handle seguito dal gruppo di appartenenza, ad esempio *CHAO-S/SANITY*, *DWEEZIL/STELLAR*, *DAK/MAD ELKS*, e così via. Io, per la scena, sono *RANDY/RAM JAM*, ma ovviamente Fabio Ciucci per chi rimarrebbe perplesso, non conoscendo l'argomento. La scena organizza dei *PARTY*, delle specie di feste-ritrovo, dove i gruppi portano la loro demo, e ci sono delle competizioni con votazioni e premi anche di milioni per i vincitori.

Alcuni *coder* di demo poi passano a fare i giochi, dato che l'argomento è sempre quello. Ad esempio il programmatore di Banshee è *HANNIBAL/LEMON.*, quello di Elfmania è *SAVIOUR/COMPLEX*, quelli di Stardust sono *DESTOP/CNCD* e *SCY/CNCD*, e la lista potrebbe continuare. . . Comunque nel disco 2 è presente una lezione solo sulla *SCENA*.

Ritornando alla programmazione assembler, sia che vogliate fare demo o giochi, vi sconsiglio di cominciare ad imparare studiando i listati di *routines 3d* (routine=parte di un listato o programma), perché sono le più complesse, che io stesso digerisco male, non per la programmazione in sé ma per le formulacce di matematica che contengono. Ma attenzione! Non dovete nemmeno pensare che se non serve la matematica servano conoscenze di elettronica o che sia necessario studiare gli schemi elettrici di Amiga!!! Quello va fatto solo se volete fare un programma per gestire una scheda grafica o un digitalizzatore video o simili.

Vi assicuro che potete, ad esempio, far apparire sullo schermo una figura o suonare una musica senza conoscere di dove passano i fili!!! Conosco persone che hanno imparato l'assembler a 12 anni e altre che lo hanno imparato a 30 o 40, senza conoscere bene la matematica e senza conoscere l'inglese. Quindi nemmeno l'età è una scusa accettabile per non provare! *Già! Perché dovete togliervi dalla testa anche che è indispensabile la conoscenza dell'inglese!*

Devo ammettere, però, che la conoscenza dell'inglese può rendere più facile il tutto, perché i comandi ASM sono abbreviazioni di parole inglesi, tipo *SUB* e *ADD* che significano *SOTTRAI* e *ADDIZIONE*. La conoscenza del *WorkBench* e dell'*AmigaDOS* non vi saranno utili per la programmazione in sé, in quanto il computer in realtà funziona molto diversamente. Io direi, in maniera più semplice, che queste "sovrastutture" sono il sistema operativo, localizzato nel chip del *KickStart*, senza il quale all'accensione non comparirebbe nemmeno la schermata che chiede di inserire il dischetto. Le finestre che vedete e spostate sono il frutto di migliaia di linee di codice ASM, contenute nel *KickStart*, infatti basta vedere la differenza delle finestre tra il *Kick 1.3* e il *Kick 2.0*, che non sono dovute alla differenza dei dischi inseriti, ma alle differenze nel *Kick* stesso.

Se volete fare programmi stile *DeLuxe Paint*, gestione casalinga, word processor, o comunque utility per WorkBench che aprano le loro finestrelle su cui selezionare i gadget e i menu a tendina, vi consiglio di imparare il linguaggio C anziché l'ASM, in quanto è più indicato e una volta imparato potete convertire i vostri listati facilmente all'ambiente MS-DOS e WINDOWS, nel caso che voleste abbandonare l'Amiga.

Se invece siete affascinati dalle demo grafiche con le palline rimbalzanti e le scritte metalizzate e sognate di programmare giochi tipo AGONY, LIONHEART, SHADOW OF THE BEAST, TURRICAN, APYDIA, PROJECT X, SUPERFROG, ZOOL, GODS, CHAOS ENGINE, XENON II, LOTUS ESPRIT, e mettiamoci anche SENSIBLE SOCCER, sia chiaro che si possono fare solo in ASSEMBLER PURO!!! E non richiedono particolari conoscenze di matematica: bastano le classiche addizioni, sottrazioni, moltiplicazioni e divisioni, e qualche tabella di SENI e COSENI per fare, ad esempio, le palline che cadono con una traiettoria a parabola, o comunque seguendo una curva: queste tabelle non sono altro che una serie di numeri in memoria tipo 1, 2, 3, 5, 8, 10, 13, 15, 18, 23 che sono, ad esempio, la progressione della posizione orizzontale e un'altra serie di numeri che sono la progressione della posizione verticale; queste serie di "tabelle" o *SINUSTAB*, cioè una serie di numeri che definiscono le coordinate di una curva, possono essere costruite con un apposito comando, il CS, presente nell'ASMONE, l'assemblatore, anche senza conoscere esattamente la trigonometria, può bastare sapere i parametri da passare e fare delle prove.

Di queste *SINUSTAB* o *TABELLE* ce ne sono molte nei giochi e nelle demo, in quanto molti movimenti ondegianti non sono calcolati del tutto sul posto. Se invece sognate di fare delle *adventure* tipo *Monkey Island*, o dei giochi manageriali, in cui appaiono cioè solo schermate grafiche ferme con qualche ometto che ci si muove dentro lentamente, in cui il gioco consiste nel selezionare con il mouse degli oggetti o delle scritte, allora si può usare anche il linguaggio C, perché il gioco potrebbe essere convertito facilmente su PC, dove si farebbero un bel po di soldoni. D'altronde il C del PC lo insegnano nelle scuole scientifiche, e molto bene nelle università informatiche, e i soldoni li faranno loro.

NOTA: conoscere l'assembler dell'Amiga può rivelarsi utile se si passa, in seguito, a programmare anche un altro tipo di computer con lo stesso microprocessore, ossia il Motorola 68000 che, per fare un esempio, è usato da computer quali Apple MacIntosh e Atari ST.

Questi computer hanno però diversi sistemi operativi (diversi dal KickStart Amiga) e diversi chip dedicati alla grafica ed al suono, dunque vi servirà la conoscenza delle istruzioni del 68000, ma non quella del sistema operativo Amiga e dei suoi chip grafici, dovreste imparare da capo; d'altronde anche con linguaggi come il C dovreste imparare il nuovo sistema operativo.

Se ad esempio usate il linguaggio C e fate un programma per WorkBench che apre le finestre e magari fa dei disegni tipo montagnine, nel caso che compraste un PC MSDOS e voleste rifarvelo su WINDOWS, le parti del vostro programma inerenti ai calcoli per fare le montagnine e la struttura generale la potreste riutilizzare, ma tutta la parte inerente all'apertura delle finestre WorkBench e dei suoi gadget di selezione la dovreste buttare e sostituire con le istruzioni per Windows, e vi assicuro che imparare un altro sistema operativo e convertire un programma costa dei mesi o degli anni. NOTA: un programma scritto in assembler 68000 funziona benissimo sugli altri processori più potenti, a patto che si siano tenute presenti alcune cose.

Se state leggendo ancora significa che siete imperterriti. Allora completo la lista delle utilità dell'assembler... (il linguaggio in sé si dice *ASSEMBLY*, il programma che lo compila si dice *ASSEMBLER*, ma è uso comune chiamare assembler anche il linguaggio). Innanzitutto l'assembler rimane il linguaggio più veloce, specialmente se lo sapete bene, e la stessa cosa fatta con un altro linguaggio sarà sempre più lenta di una fatta in assembler.

Poi rimane anche l'unico mezzo per creare effetti grafici speciali, mai visti prima: potete ottenere effetti speciali anche con un titolatore, ma potete fare SOLO quelli definiti dal programma. Infatti non è difficile scoprire con che programma è stata fatta una titolazione o un effetto spe-

ciale; lo stesso vale per i DEMO MAKER, di cui il migliore è il *TRSI DEMOMAKER*, che ha degli effetti interessanti, ma ormai anche i bambini riconoscono una cosa fatta col demomaker, perché c'è sempre la scritta dorata sopra e sotto e al centro o le palline o le stelline... E BASTA!!! non se ne può più! Imparando a programmare in assembler, invece, si possono inventare degli effetti mai visti prima, perché non si è limitati a dover scegliere tra una ventina di effetti pronti per l'uso che altre migliaia di persone hanno usato, riempiendo reti televisive private e dischi. Per darvi un'idea dell'infinita varietà di cose che potete inventare in assembler posso nominare la *SPACE-BALLS "state of the art" DEMO*, una delle più conosciute, che non è difficile da programmare e ha stupito per le figure stilizzate di donne che ballano in mezzo a degli effetti speciali.

Se un programmatore ha più pazienza può anche programmare un gioco, dapprima per giocarselo, per il gusto di farsi il gioco dei sogni, per sperimentare i veri limiti dell'Amiga, per vedere quanti ometti si riesce a muovere senza rallentare lo schermo, poi nulla vieta di provare a fare un gioco commerciale, che richiede anche la collaborazione di grafici e musicisti, nonché tutta la parte relativa alla commercializzazione che spesso premia più la pubblicità fatta al gioco che la sua effettiva validità, a parte i casi in cui la validità è tanta che il successo arriva comunque.

Perché non mettersi a fare un gioco per CD32?? Basta fare un gioco AGA che sfrutti i 600MB di capienza del CD: per esempio un gioco in cui lo sfondo sia un "film" caricato in tempo reale, su cui far girotondare un RAMBO ammazzatutti o un'astronave. La difficoltà non sta né nell'imparare il nuovo chipset AGA, né nell'adattamento per CD, infatti il chipset AGA è molto simile a quello normale, basta impararsi qualche registro nuovo, e il processore funziona allo stesso modo, mentre per quanto riguarda la gestione del CD è ancora più facile, perché basta studiarsi i 2 dischi del "CD 32 DEVELOPER KIT" che circola tra i programmatori. Dunque l'assembler alle soglie del 2000 può essere ancora all'avanguardia, ovviamente per certi compiti in particolare, e se la tecnologia del 2000 sarà tutta su CD, come auspicano coloro che si comprano il PC per giocare ai giochi del CD o spesso per vedercisi le donne nude, dato che i CD sul PC MS-DOS sono in maggioranza slideshow sexy, anche l'Amiga avrà il suo software su CD, che potrebbe essere sviluppato da qualche tizio che un giorno cominciò la sua avventura leggendo un certo corso di programmazione...

Conoscendo come funziona il tutto, si può anche capire come funzionano certi programmi o giochi e si possono modificare delle sue parti: ad esempio si può capire come mai un gioco o un programma non funziona sui nuovi modelli Amiga e si può modificare per farlo funzionare, si possono fare certe modifiche ai programmi, per fare un esempio ho modificato una utility in modo che usasse la memoria virtuale su disco sull'Amiga 4000, altre volte ho velocizzato dei programmini PD, di cui ho "rubato" e velocizzato le parti più importanti. Infine si possono fare i cosiddetti trainer, le vite infinite, si trovano cioè le parti di listato che sottraggono una vita al povero PLAYER 1 e si modifica il tutto, magari facendo aumentare le vite quando si è ammazzati... per vedere e capire come funziona un gioco o un programma però è necessario conoscere VERAMENTE bene l'ASM e disporre di un monitor L.M o meglio di una cartuccia tipo ACTION REPLAY (Il monitor L.M. è un'utilità che permette di disassemblare, cioè visualizzare le istruzioni presenti in una sezione di memoria, e se si trova in che punto della memoria sono le istruzioni che tolgono una vita, si può modificare il tutto).

L.M sta per Linguaggio Macchina, cioè linguaggio del microprocessore, che è quello prodotto dall'assemblatore). Queste operazioni comunque sono una cosa difficile, e cominciare tentando di far diventare verde un ometto blu in un gioco non è certo utile. Ho visto tanti ragazzi sciupare il loro tempo andando a caso con i monitor L.M. e le cartucce tentando invano di fare non si sa cosa, cambiando le scritte nei programmi o nei listati, senza capirli, dicendo che li avevano fatti loro o che ci avevano fatto non si sa quali importanti modifiche. Costoro tutt'oggi non sanno visualizzare un'immagine in assembler; in gergo questi ciarlatani sono detti *LAMER*.

Facciamo il punto della situazione: se siete uno di quei diciottenni classici bianchicci e gobbi,

senza donne, e state andando a caso con i monitor LM per la memoria del vostro povero Amiga, vantandovi di essere un grande hacker, allora vi consiglio di posare il monitor e seguirmi per la retta via. Anche io ho cominciato in quella maniera ridicola (a 8 anni però! non a 18!), ma poi mi sono ravveduto e ho cominciato a leggere i libri senza saltare le pagine. Ecco un libro che vi potrebbe servire: *IL MANUALE DELL'HARDWARE DELL'AMIGA*, della IHT. Questo manuale spiega come funzionano i CHIP CUSTOM, quelli che fanno la grafica ed il suono dell'amiga, nonché come pilotare il DISK DRIVE eccetera.

Questo è indispensabile, ma per visualizzare anche una sola immagine bisogna conoscere anche il 68000, essendo il 68000 che gestisce i chip grafici. Inoltre il tutto rimane una cosa astratta, una specie di sintetica serie di tabelle di riferimento, e non ci sono validi esempi. Molti esempi li potete comunque trovare nel mio corso!!!! Se sapete l'inglese cercate l'ultimo *HARDWARE REFERENCE MANUAL*, che è aggiornato sui nuovi chip ECS. Comunque potete farne a meno per la durata del mio corso in quanto le cose principali ci sono, anche sui chip AGA dell'Amiga 1200. Inoltre nell'ASMONE è incluso un comando, `<=C>`, che da una spiegazione di tutti i registri `$dffXXX`, sia in generale che in particolare, ad esempio:

=C 100

vi darà una spiegazione del registro BLTCON0, concernente la risoluzione grafica, allo stesso modo `<=C 040>` vi darà un sunto del BLTCON0, reg. del BLITTER (`$dff040`).

I libri tipo *ROM KERNEL MANUAL* e *PROGRAMMARE L'AMIGA volume 1 e 2* della IHT, non sono utili alla programmazione diretta all'hardware, quella che tenterò di insegnare in questo corso, ma sono utili a chi voglia fare programmi per il workbench o l'amigados, che usino il sistema operativo contenuto nel kickstart e nei disks del workbench... programmi con finestre intuition dunque, non schermate con palline ed equalizzatori o ometti saltellanti tra le fiammelle... dunque più utili ai programmatori in C. *Non ve li consiglio...* programmare così è noiosissimo.

NOTA: Se, invece di essere utilizzatori bianchicci di monitor LM a caso, siete degli avidi ricercatori di giochi nuovi da copiare e da finire, passate ore al telefono a chiedere delle ultime novità, e le ore rimanenti a copiare con XCOPY e a giocare, magari sempre col trainer tanto per finire più in fretta, allora è peggio che essere gobbi col monitor LM: o interrompete questo affanno della ricerca e della copia, o rimarrete degli ipertesi che non sanno assolutamente come mai gli si muovono gli ometti per lo schermo, ne saprete mai come farvelo da voi un trainer al gioco col menu e tutto, e vi assicuro che quando vi siete fatto un trainer da voi, poi non vi interessa più di finire il gioco, ma piuttosto di capire come funziona.

Questa è la differenza che c'è tra il giocatore ed il creatore del gioco, tra il popolo assoggettato e stolto e i capi del regime che lo comandano facendogli passare notti insonni a finire (col trainer o meno) una miriade di giochi, non importa quali, basta che siano tanti e nuovi, copiati con l'XCOPY (che tra l'altro è il peggior copiatore al mondo! usate il DCOPY piuttosto!).

P.S: a proposito di donne, NON HO MAI VISTO NULLA PROGRAMMATO IN ASM DA UNA DONNA!!!! Se sta leggendo un rappresentante del sesso femminile, credo che questo sia un motivo in più per essere la prima!!! Una ragazza che, invece di interessarsi a pettegolezzi su persone sconosciute, o alle vetrine dei negozi, si metta a programmare robe pazzesche in gonna, credo che metterebbe in crisi di identità un bel po' di bambinoni che si ritengono intelligenti facendo vedere, alle (poche) ragazze che conoscono, quanto muovono bene la freccia del mouse o le finestre del workbench, pensando che tanto non capiscono nulla e che gli possono inventare che sono dei geni e che stanno avendo dei collegamenti con la NASA, quando invece non sanno nemmeno formattare un dischetto.

Vi anticipo che la LEZIONE2.TXT, che leggerete con i suoi sorgenti esempio dopo questa LEZIONE1.TXT, è la più **difficile**, quella **fatidica**, cioè se riuscite a passarla il gioco è fatto, perché

già dalla lezione 3 si fanno i primi effetti speciali col copper e procederete veloci come delle fucilate fino in fondo. Dunque vi chiedo di avere la pazienza di superare con calma e impegno la LEZIONE2.TXT, senza saltare niente.

Ora facciamo un'analisi dei programmi usati per programmare in assembly:

ASSEMBLATORE è il programma che traduce il listato fatto di comandi in formato simbolico (move, add...) nel suo equivalente binario (cioè in bytes). Cioè traduce un testo, leggibile dal programmatore, nel formato reale delle istruzioni come le legge ed esegue il processore (una sequenza di numeri). Per esempio il comando RTS sarà trasformato in \$4e75, e così via. Questo rende umano programmare, perché immaginatevi che roba programmare sapendo a memoria i numeri corrispondenti a ogni istruzione!!!! Programmare per NUMERI vorrebbe dire programmare in vero LINGUAGGIO MACCHINA, ossia L.M, ma è inutile, si fa molto meglio in ASSEMBLY, cioè usando delle parole convenzionali, dette COMANDI, al posto dei numeri reali. Questo codice binario risultante è chiamato codice oggetto ed è direttamente eseguibile dal computer, infatti può essere salvato il file eseguibile, oppure si può collaudare il programma. Ricordatevi che in assembler è in uso anche la numerazione esadecimale! I numeri esadecimali sono quelli preceduti dal \$, e sono in base 16, come spiegheremo, e possono contenere anche le lettere ABCDEF, come in \$4e75. Si deve tenere presente che se il listato ha degli errori "grammaticali" ci viene comunicato dall'assemblatore, infatti ci sono delle precise regole a cui attenersi: ad esempio le LABEL (o ETICHETTE), devono cominciare dall'inizio della riga, ossia non devono essere precedute da spazi, e devono terminare con i due punti (:). Ad esempio una LABEL corretta è

```
1 PIPPO:
```

Infatti il nome si dà a piacere, e PIPPO va bene, perché non contiene simboli come = + - eccetera, non ha spazi che la precedono, e termina con i :. Le *label* sono nomi che si danno qua e là nel listato a delle cose, e servono per indicare quelle cose durante il programma, se per esempio si dà nome PIPPO: a una certa serie di istruzioni, quando nel programma diremo che si deve eseguire PIPPO verranno eseguite le istruzioni sotto PIPPO, allo stesso modo possiamo mettere una label ad una figura o a una musica; la label dunque rappresenta l'indirizzo di memoria dove si trova, come il nome dei luoghi rappresenta la posizione di quei luoghi! Se voglio andare in Australia, vedrò una bella label AUSTRALIA: sopra di essa. Ricordatevi però che le label servono a noi per orientarci, ma quando l'assemblatore trasforma tutto, nel codice oggetto non ci sono label, solo i numeri corrispondenti alle istruzioni.

Ci sono poi le istruzioni, che invece devono essere SEMPRE precedute da degli spazi, meglio se da un TAB (che fa 8 spazi in un colpo solo, è il tasto sopra CTRL), e seguite dagli operandi, ad esempio:

```
1 PIPPO:
2   move.l $10,$20
```

In questo caso MOVE.L è l'istruzione, mentre il primo operando è \$10 e il secondo è \$20. Alcune istruzioni necessitano di un solo operando e altre di nessun operando, ad esempio:

```
1   clr.l $10
```

Necessita di un solo operando. Istruzioni come RTS non necessitano di operandi. Infine ci può essere un commento, utile per ricordarci cosa si sta facendo con le istruzioni: il commento si può scrivere dopo un punto e virgola (;).

```

1 PIPPO:           ; LABEL, che rappresenta l'indirizzo di MOVE.L
2   move.l $10,$20 ; istruzione a 2 operandi
3   clr.l $10      ; istruzione ad 1 operando
4   rts           ; istruzione senza operandi

```

I commenti sono ignorati durante l'assemblaggio, quindi potete scrivere di tutto, basta che sia dopo i ;. Questa è la grammatica. Seguendo queste semplici regole il programma viene assemblato. Poi che faccia quello che deve fare o no dipende da voi!!!

EDITOR invece è un programma che serve per scrivere o modificare i testi, nel nostro caso per scrivere i listati, che non sono altro che dei testi, fatti di parole chiave (MOVE, ADD...) e commenti del programmatore (posti dopo i ;). I più potenti editor possono cercare, agganciare e sostituire caratteri. Solitamente ai listati assembly si dà un nome che finisce con .ASM o .S, io personalmente preferisco .S, infatti quelli del corso finiscono in .S, mentre i testi da leggere finiscono in .TXT, ma il nome del file ovviamente non ha importanza per l'assemblatore, che lo carica comunque.

MONITOR non è in questo caso da intendersi come quello schermo su cui vedete le immagini dell'Amiga, ma un altro programma che permette di far vedere i contenuti della memoria, per esempio che numero c'è all'indirizzo \$100, e così via. Solitamente i *monitor* hanno anche un *disassemblatore*, ossia il contrario dell'*assemblatore*, che ci permette di vedere la memoria come istruzioni, anziché come numeri, ossia traduce i numeri nei rispettivi comandi simbolici (MOVE, ADD...), in modo da rendere chiaro il funzionamento. Trasforma cioè il *linguaggio macchina* in *assembly*, ossia ricostruire le istruzioni assembly che ogni numero rappresenta, riportando il *codice oggetto* alla forma originaria che avete usato nel listato. Per riprendere l'esempio usato per l'assemblatore, trasforma \$4e75 in RTS.

DEBUGGER serve per collaudare il programma istruzione per istruzione, visualizzando gli effetti delle istruzioni ogni volta, e può indicare la causa del malfunzionamento del programma. Quindi consente di far eseguire il programma a pezzi, cioè definire fino a quando eseguirlo, per controllare la situazione, e poi riprendere l'esecuzione, per trovare ogni errore. Infatti *bug* significa *errore*, in gergo; in inglese significa *pulce pestifera*, infatti gli errori di solito sono difficili da trovare nel programma; con il debugger si può verificare in che punto si verifica l'irregolarità.

Alle volte il codice oggetto per poter funzionare effettivamente su un sistema operativo deve essere *linkato* con il *linker*, perché i file eseguibili non sono semplicemente il blocco di istruzioni che avete assemblato, ma hanno delle parti che permettono di farlo caricare in memoria dal sistema operativo. Questo vale per i file .EXE e .COM del PC MS-DOS e per i file eseguibili di qualsiasi altro sistema operativo, è per questo che un eseguibile per Amiga non viene caricato da un ATARI ST o da un MACINTOSH, che pure hanno un 68000, proprio perché il formato FILE è diverso. Gli Amiga in particolare hanno gli *HUNK*, e per trasformare il codice oggetto in un file con HUNK, che possa essere eseguito clickandoci col mouse o caricandolo dallo SHELL, bisogna linkarlo. Per fortuna molti assemblatori hanno il linker incorporato, per cui non occorre fare questo passaggio.

Ebbene, l'assemblatore incluso in questo corso, il **TRASH'M'ONE**, ha un EDITOR, un ASSEMBLATORE, un MONITOR/DEBUGGER e un LINKER!!! Ossia tutto in UNO!!!! è la versione modificata PD (Ossia liberamente copiabile?) dell'Asmone.

A proposito dell'editor, potete cercare un testo premendo contemporaneamente i tasti AMIGA <destro+SHIFT+S> oppure selezionando col mouse (<tasto destro>) l'opzione Search nel menù a tendina sotto la voce "Edit Funct."; a questo punto apparirà in alto a sinistra la scritta "Search for:", dove dovreste scrivere la parola (o le parole) da cercare. Può esservi utile intanto

per ritrovare il punto dove siete arrivati nella lettura: se per esempio volete smettere qua per oggi, potete segnarvi la linea dove siete arrivati, in questo caso la 549 (indicata in fondo a sinistra), oppure il potete anche ritrovare questo punto del testo cercando una sua parola, per esempio “Funct”, oppure “caso la 549”, oppure “cercando”, oppure quello che vi pare.

Normalmente, avremmo dovuto scrivere il nostro listato con un EDITOR, e salvare il listato (detto SORGENTE) con un nome a piacere. Poi avremmo dovuto caricare l’assemblatore, da cui caricare il listato, assemblare (cioè trasformare dal testo a suo equivalente in L.M) e salvare il codice oggetto. Per collaudare il programma, ovvero controllare se funziona, avremmo dovuto eseguirlo dall’assemblatore, oppure linkarlo, rendendolo eseguibile, e farlo partire dal DOS. Per tornare a modificarlo avremmo dovuto cercare l’editor, ricaricare il listato, modificarlo, salvarlo, e rifare tutto l’assemblaggio. Sul PC MSDOS questo è quello che si deve fare, infatti ho rinunciato a programmarci, mentre sull’Amiga col multitasking si possono caricare insieme l’EDITOR, l’ASSEMBLATORE ecc. Come se non bastasse, qualcuno ha inventato il mitico SEKA, simile all’attuale ASMONE, che aveva editor, assemblatore e monitor insieme. Con l’evoluzione siamo arrivati al MASTERSEKA, poi all’ASMONE e infine alle molte versioni modificate dell’ASMONE dai più svariati programmatori hobbisti. I due più accaniti modificatori (bravi!!) sono i TFA, che hanno fatto il TFA ASMONE, e DEFTRONIC, che ha fatto questo TRASH’M’ONE. Ho scelto quello di Deftronic perchè è quello che ha meno BUG, infatti essendo modificati alla meglio questi ASMONE spesso assemblano fischi per fiaschi o si bloccano improvvisamente, ma non ci si può certo lamentare con loro che si divertono ad aggiungere opzioni senza guadagnare un soldo!

Il risultato finale è che vi potete scrivere il listato, poi premendo ESC passate all’assemblatore/monitor, da cui potete assemblare (con <A>), oppure vedere i contenuti della memoria, sia come numeri che come istruzioni DISASSEMBLATE, potete verificare il funzionamento del programma, e infine salvare direttamente il FILE eseguibile con <#0>. Per non fare confusione considerate che una cosa è salvare il listato, ossia il SORGENTE, che è un TESTO, un’altra è salvare l’eseguibile, che è un PROGRAMMA fatto di istruzioni nel formato FILE ESEGUIBILE. Il sorgente può essere scritto anche con un altro editor, come il CED, e poi potete caricarlo dall’ASMONE. Allo stesso modo un testo fatto con l’Asmone può essere caricato da un editor. L’Editor dell’Asmone quindi non è che un normale EDITOR inserito in un assemblatore, con cui potete scrivere anche una lettera per la mamma, oppure modificare la STARTUP-SEQUENCE di un disco (Chi non sa cos’è, per favore si legga il manuale dell’AmigaDos!).

Orunque, procederò facendo chiarimenti e spiegando a modo mio come funziona il computer, per evitare fraintendimenti.

Quello che organizza tutto è il microprocessore 68000, la CPU, ovvero Central Processing Unit, insomma il Boss... Il processore esegue delle istruzioni, infatti ha un set di istruzioni ben precise che sa eseguire, e che esegue una dopo l’altra (di seguito), a meno che nel suo cammino non trovi l’istruzione di saltare ad eseguire più avanti o più indietro, o di fare un certo numero di loop (o cicli). Nomino per esempio alcune istruzioni: MOVE, che significa “copia un valore da un posto ad un altro”, ad esempio “move \$10,\$20” muove quello che è in \$10 nella locazione \$20, oppure CLR, che significa AZZERARE: “clr \$10” azzerare la locazione \$10... (per LOCAZIONE intendo un punto della memoria accessibile dal processore).

A proposito! Il processore opera sulla memoria! Facciamo una mappa. Quando le istruzioni operano con indirizzi minori di \$200000 si sta operando nella CHIP RAM, ossia: da \$000000 a \$80000 ci sono i primi 512k di CHIP, quelli dei vecchi a500 o a2000, mentre se la RAM continua fino a \$100000 significa che c’è 1 MB di chip RAM, come negli a500+, a600 o nei nuovi a2000, se la memoria CHIP invece è di 2MB, come negli a1200 o negli a500+ o a600 espansi, ad esempio, la chip va da \$000000 a \$200000. Insomma quando il processore lavora su indirizzi minori di \$200000 ci troviamo in CHIP RAM, ad esempio:

```
2  move.l $150000,$1a0000
```

Sono istruzioni che operano sulla CHIP RAM.

Quando invece operano su indirizzi da \$200000 in avanti, ci troviamo in FAST RAM, ad esempio un a500 vecchio con 1MB di memoria, divisa in 512k di CHIP e 512k di FAST ha la memoria divisa in 2 pezzi:

1. da \$000000 a \$80000 i primi 512k di CHIP RAM
2. da \$c00000 a \$c80000 512k di FAST RAM.

Potete verificare con utility come SYSINFO i blocchi di memoria che avete.

Poi ci sono delle zone di memoria speciali, come quelle della ROM Kickstart, ossia di solito da \$fc0000 per kick 1.2 e 1.3 o \$f80000 per kick 2.0 o 3.0. La ROM, a differenza della RAM, non può essere sovrascritta, si può solo leggere, e non si cancella quando si spegne il computer.

Un importantissimo indirizzo è \$dff000, in quanto quando le istruzioni operano su indirizzi che vanno da \$dff000 a \$dff1fe vengono azionati i CHIP CUSTOM della grafica e del suono, infatti per azionare la grafica bisogna mettere i valori giusti in questi indirizzi \$dffxxx, detti anche REGISTRI, proprio perché ognuno ha una funzione: provate a fare dalla linea di comandi (premendo <ESC> si scambia tra l'Editor e i comandi) il comando <=C>, e vedrete il riassunto di quei registri, con il numero, in cui 000 sta per \$dff000 e 100 sta per \$dff100, e il nome, ad esempio \$dff006 è VHP0SR, mentre \$dff100 è BPLCON0. Questi indirizzi o si possono solo leggere, o si possono solo scrivere, per esempio \$dff006 si può solo leggere, e \$dff100 si può solo scrivere. Noterete una W o una R tra il numero e il nome: quelli che hanno una W sono quelli che si possono solo scrivere, (WRITE in inglese), quelli con la R si possono solo leggere (READ). Alcuni sono S (strobe) o ER (EarlyRead), ne parleremo in seguito quando li useremo.

Altri indirizzi speciali si trovano nella zona \$bfexxx, ossia da \$bfe001 a \$bfef01: si tratta di indirizzi collegati al chip CIAA, che si occupa di varie cose come fare da timer, ossia da cronometro, e di controllare le porte come la parallela (quella della stampante). Analoghi compiti li svolge il CIAB, connesso agli indirizzi \$bfdxxx.

Quello che dovete ricordarvi in pratica è che quando vedete un indirizzo del tipo \$dffxxx o \$bfdxxx o \$bfexxx, stiamo operando su un registro CUSTOM, causando cose come il cambiamento dei colori dello schermo, o la verifica dei movimenti del joystick o del mouse, o altro ancora.

Per quanto riguarda la memoria RAM, sia CHIP che FAST, non vi interesserà sapere a che indirizzo si trova ogni istruzione, perché l'assemblatore, come sapete, ci permette di usare le LABEL, al posto degli indirizzi: le metteremo solo nei punti utili, ci penserà l'ASMONE poi a mettere gli indirizzi reali al posto delle label. Potremo vedere dopo a che indirizzo sono finite le nostre istruzioni, se ci interesserà.

Continuiamo con gli esempi delle istruzioni. Ci sono comandi come ADD e SUB, che significano ADDIZIONE E SOTTRAI, ad esempio SUB #10,ENERGIA sottrarrà 10 al valore dell'energia; ci sono le moltiplicazioni e le divisioni con MULS, MULU, DIVS e DIVU, e le operazioni logiche OR, AND, NOT ed altre. JMP significa JUMP, ovvero salta ad eseguire ad una certa locazione (esempio JMP \$40000), JSR invece significa esegui una routine ad una data locazione fino a che non trovi un RTS, ovvero "ritorna che è finita la routine", e l'esecuzione continuerà dopo il JSR; BRA fa la stessa cosa di JSR e BSR fa come JSR. TST significa TESTA rispetto a zero, ovvero controlla se una data locazione o registro è uguale a zero; questa istruzione o l'istruzione CMP, ovvero COMPARA qualcosa con qualcos'altro, è seguita di solito da un salto condizionato: BEQ e BNE ad esempio, che significano:

BEQ Salta a una certa locazione se è vera la condizione (*Branch if Equal*)

BNE Salta se non è vera (*Branch if Not Equal*). Si creano così delle diramazioni varie.

Facciamo un esempio stupido:

```

1 Principale:
2   bsr CAMPANE           ; BSR fa saltare sotto la label CAMPANE, dopodiché
3                           ; ritorna qua ad eseguire BSR aspettamouse
4   bsr aspettamouse      ; Aspetta che sia premuto il MOUSE
5   bsr PAVAROTTI
6   rts                  ; torna all'asmone o al workbench
7
8 aspettamouse:
9   controlla se il tasto del mouse è premuto
10  se non è premuto vai a aspettamouse, ossia fai il girotondo fino a che
11  non è premuto il mouse. (in questo caso si mette un "BNE aspettamouse")
12  rts                  ; fine subroutine aspettamouse, torna sotto il BSR
13
14 CAMPANE:
15   dindon                ; una routine che suona dindon
16   rts
17
18 PAVAROTTI:
19   AAAAHHHHHHHHH        ; una routine che fa cantare Pavarotti
20   rts
21
22   end                  ; Indica la fine del listato, si può anche non metterlo.
23
24 (quello che viene scritto sotto l'END non viene letto nè assemblato)

```

ordunque, eseguendo questo ipotetico programma, si può dire che “Principale” è la routine, appunto, principale, che richiama 3 routines (parti del programma a cui viene dato un nome, ad esempio PAVAROTTI) in sequenza: all’inizio il processore salterebbe sotto CAMPANE: e suonerebbe le campane, poi trova un RTS e torna sotto BSR CAMPANE, dove trova un altro BSR che lo porta sotto aspettamouse: che è una routine che fa un ciclo fino a che non è premuto il tasto del mouse... il processore controlla miliardi di volte il mouse e se non è premuto ritorna sempre a controllare senza sosta; quando il mouse viene calpestato (premuto) la situazione cambia, perché si esce dal ciclo infinito ASPETTAMOUSE, e si arriva al suo RTS, ossia all’uscita, che lo fa tornare a PRINCIPALE sotto il BSR aspettamouse che abbiamo superato (il processore esegue sempre l’istruzione seguente, ossia sotto, e anche quando torna da un BSR, ossia dall’esecuzione di certe istruzioni messe in altro luogo) e trova l’ennesimo BSR che lo porta a far cantare pavarotti. Infine tornato dal concerto di Pavarotti trova un RTS, che lo fa uscire da PRINCIPALE e quindi torna all’asmone o al workbench: IL PROGRAMMA è finito.

Ora spiegherò meglio come si sposta il processore tra le varie istruzioni. Nel caso BEQ label, si può parlare di diramazione, infatti a questo punto si possono prendere 2 vie: immaginatevi proprio un albero, di quelli secchi senza foglie, una quercia secolare, con il tronco nodoso, che a un certo punto si divide in 2 rami, poi ognuno di questi 2 rami si divide in 2, e così via. Quando arriviamo al BEQ è come se fossimo una formichina che è partita dall’inizio del programma, ossia dalla base dell’albero, in cui c’è il nostro formicaio START:, e siamo arrivati alla prima diramazione: a questo punto o scegliamo di proseguire sul ramo destro o su quello sinistro. Questa scelta il 68000 la fa in base al risultato della condizione, sia essa un CMP o un TST:

```

1 INIZIO:                ; formicaio nell'erbetta
2   tst.b LABEL30        ; Il byte della LABEL30 è= 0??? (condizione esempio)
3   beq RAMODESTRO       ; se sì, allora salta a RAMODESTRO
4   ...                  ; non è=0, allora eseguiamo il RAMOSINISTRO
5                           ; (significa che il byte è un numero da $01 a $FF)
6   (Istruzioni del RAMOSINISTRO)
7   rts                  ; Fine, usciamo: abbiamo percorso il RAMOSINISTRO e non
8                           ; quello DESTRO
9
10 RAMODESTRO:
11   ...                  ; (Istruzioni del RAMODESTRO)
12   ...

```


13 **rts** ; Fine, abbiamo percorso il RAMODESTRO e non quello
 14 ; SINISTRO

In questo caso una condizione di TST (confronta con 0) e di CMP (confronta il primo operando col secondo) seguita da un BEQ (se sì, salta a...) o BNE (se no, salta a...), serve a scegliere se eseguire una certa serie di istruzioni o un'altra, se prendere una via o un'altra. Abbiamo già usato il BNE per fare un ciclo (o loop) in cui invece un certo numero di istruzioni sono eseguite ripetutamente fino a che non è verificata la condizione, per esempio il loop che aspetta che sia premuto il mouse. Il Ciclo può essere paragonato, anziché ad una formichina che sale un albero, ad un ROBOT che ha la pazienza di fare anche un miliardo di volte la stessa cosa senza stancarsi o scioperare, ad esempio:

```
VAI IN CUCINA, CONTROLLA SE LA TORTA è COTTA, SE NON è COTTA TORNA IN SALOTTO
E TOGLI LE PULCI AL CANE PER 30 SECONDI, DOPODICHE'
VAI IN CUCINA, CONTROLLA SE LA TORTA è COTTA, SE NON è COTTA TORNA IN SALOTTO
E TOGLI LE PULCI AL CANE PER 30 SECONDI, DOPODICHE'
VAI IN CUCINA, CONTROLLA SE LA TORTA è COTTA, SE NON è COTTA TORNA IN SALOTTO
E TOGLI LE PULCI AL CANE PER 30 SECONDI, DOPODICHE'
VAI IN CUCINA, CONTROLLA SE LA TORTA è COTTA, SE NON è COTTA TORNA IN SALOTTO
E TOGLI LE PULCI AL CANE PER 30 SECONDI, DOPODICHE'
VAI IN CUCINA, CONTROLLA SE LA TORTA è COTTA, SE NON è COTTA TORNA IN SALOTTO
E TOGLI LE PULCI AL CANE PER 30 SECONDI, DOPODICHE'...
```

Come si vede un essere umano si ribellerebbe a fare per il tempo necessario alla cottura di una torta un vai e vieni simile in continuazione, ma il 68000 non batte ciglio. Quando finalmente la torta è cotta, si verifica il BEQ, e il ROBOT salta alla routine TOGLILADALFORNOSENZASCOTTAR-TIEMETTILAINTAVOLA:.

Potete intuire che con delle diramazioni qua e là, anche dentro loop più o meno grandi, si può fare una struttura complicata che soddisfa qualsiasi tipo di necessità, basti pensare alla complessità dei programmi che simulano lo sviluppo di una città, che a seconda di migliaia di situazioni simula il comportamento dei cittadini. Tutto questo è possibile tramite diramazioni alle volte connesse o cicliche tra di loro.

I rami, ossia i pezzi di istruzioni eseguiti quando sono verificati i BEQ o i BNE che li richiamano, o semplicemente perché sono trovate durante il cammino del 68000, sono chiamate *routine* o *subroutine*, cioè pezzi di programma fatti di un certo numero di istruzioni che eseguono un dato compito, nel caso del ciclo del ROBOT, le istruzioni che fanno togliere la torta dal forno potrebbero essere isolate in una unica routine, che potrebbe essere eseguita ogni volta che è necessario togliere la torta dal forno. Infatti l'utilità delle routine e specialmente delle subroutine isolate sta proprio nel non dover riscrivere ogni volta che si deve togliere la torta dal forno la stessa serie di istruzioni, ad esempio.

Queste istruzioni le possiamo isolare, e metterle da parte dandogli un nome, assegnandogli cioè una *label* all'inizio, e decidendone una fine con l'istruzione RTS. Diamo una definizione alla parola *subroutine*:

dicesi subroutine di un blocco di istruzioni alle quali è stato dato un nome, facendola iniziare con una label, ossia un nome a piacere seguito dai :, e finire con una speciale istruzione di ritorno, l'RTS (ReTurn from Subroutine), e solitamente si fa eseguire con l'istruzione BSR, seguita dal nome della subroutine; dopo aver eseguito il BSR, il processore tornerà ad eseguire le istruzioni sotto il BSR che hanno fatto eseguire la subroutine stessa.

Questo si può paragonare al comandante di un sommergibile, che in questo caso è il programma principale, il quale dando gli ordini esegue delle subroutine, ad esempio immaginatevi

che il comandante abbia visto al periscopio una nave nemica, a questo punto farà un BSR *ArmateSiluri*, ossia darà il comando di armare i siluri. Fino a che la subroutine che arma i siluri non sarà eseguita non potrà procedere. Una volta avisato che sono stati armati, il comandante, ossia il programma principale, continuerà la procedura: ossia dare BSR *DESTRA* e BSR *SINISTRA* al reparto macchinisti fino a che la nave non si trova sulla traiettoria dei siluri; questo lo potremmo paragonare ad un ciclo in cui c'è un CMP *NAVE*, *SILURI* seguito da un BNE *SPOSTASOTTOMARINO*, ossia: "la LABEL che contiene la posizione della nave è uguale al contenuto della LABEL che contiene la posizione che raggiungeranno i siluri?", se non ancora (BNE), allora spostati ancora, cioè torna alla routine che controllerà se siamo più a destra o più a sinistra, e di conseguenza esegui le subroutine *SINISTRA* e *DESTRA*.

Questo ciclo è simile a quello del *ROBOT* che aspettava che la torta fosse cotta, ma in questo caso invece di aspettare la cottura siamo noi che attivamente dobbiamo raggiungere la posizione esatta, come nel ciclo che aspetta il *MOUSE* siamo noi che lo dobbiamo premere per fermarlo. Eravamo rimasti al ciclo di allineamento: all'improvviso il comandante dà il comando di lanciare i siluri! (BSR *FUORIUNO*, BSR *FUORIDUE*). *BOOOOOOOOOOOM*... Ha funzionato... morti da tutte le parti, calzini galleggianti, vedove e orfani sparse per tutta la Germania (nei film muoiono sempre i tedeschi), un relitto in fondo al mare. *TRANQUILLI!* Era solo una simulazione al computer ben riuscita.

Se siete entrati nella logica del processore, il gioco è fatto. Tutto quello che vedete girare sul computer, sia un programma per le previsioni del tempo, una demo con cubi e palline, un gioco di azione, è fatto di pezzi di programma che sono eseguiti ciclicamente o sequenzialmente, a seconda dei responsi delle varie condizioni TST, BTST, CMP. Dunque ogni tipo di operazione e di decisione, di qualunque ordine di complessità, è fatto di un certo numero di condizioni semplici, considerando che ogni subroutine può essere fatta con altre subroutine più piccole, ad esempio *TOGLILATORTADALFORNO*:

```

1 TOGLILATORTADALFORNO:
2   bsr SpengillForno
3   bsr AprillForno
4   bsr PrendiLaTorta (tanto è un robot e non si scotta)
5   bsr PosaLaTortaSulTavolo
6   bsr RichiudiilForno
7   rts

```

A sua volta le subsubroutine possono essere fatte di altre subroutine:

```

1 SpengillForno:
2   bsr VaiSull'interruttore
3   bsr GiraloVersoSinistra
4   rts

```

La maggior utilità delle subroutine sta nel rendere più chiaro il programma, dividendolo in parti logiche, e nella possibilità di farsi una raccolta di routines che possono essere usate per altri programmi, ad esempio se avete una routine che legge la posizione del joystick la potete riutilizzare in tutti i giochi che farete, con lievi modifiche se necessario, allo stesso modo la routine che suona la musica, o quella che fa camminare un ometto sul video.

Questo è per dare un'idea del continuo eseguire e girovagare a seconda delle condizioni vere o false del povero microprocessore. Quando saltellando qua e là c'è un errore, ad esempio salta in una zona con dati caricati male da disk o dove il programmatore ha fatto cilecca, allora appare il mitico *GURU MEDITATION*, o *SOFTWARE FAILURE* nella sua inquietante finestra rossa lampeggiante.

La memoria riscrivibile (RAM) può essere modificata, e si divide in CHIP ram e FAST ram, come già detto. La differenza è che la GRAFICA e i SUONI devono essere in CHIP RAM, mentre le istruzioni del processore possono essere sia in CHIP che in FAST. Per esempio l'Amiga 500 vecchio 1.2 o 1.3 ha 512Kb di RAM, ovvero mezzo mega, e se si espande si arriva ad un mega di

RAM, ma gli altri 512k sono FAST, è per quello che ad esempio con il DeLuxe Paint si finisce la memoria prima con 1MB diviso in 512k CHIP e 512k FAST rispetto ad un a500+ che ha invece 1MB tutto di CHIP: la memoria nel vecchio 500 avanza, ma è di tipo FAST e non serve ad aprire un nuovo schermo, quindi dice che non c'è memoria. Quando si programma se si prova a visualizzare grafica messa in FAST succede il finimondo, di tutto tranne visualizzare quell'immagine. La memoria è fatta a blocchi di varie misure, ad esempio su un a500 vecchio i primi 512k di chip ram vanno dall'indirizzo \$00000 a \$80000 e i 512k di espansione da \$c00000 a \$c80000: il sistema operativo sa dove sta la memoria e carica i programmi automaticamente nelle zone vuote, ad esempio caricando un programma dal WorkBench o dal CLI o SHELL i dati dal dischetto saranno trasferiti (grazie al kickstart) in memoria, a seconda che sia richiesta memoria CHIP o FAST, dopodiché il processore salterà al punto in memoria dove ha caricato, (o meglio copiato dal dischetto) il programma. All'utente rimane oscuro in che punto della memoria sia stato messo il programma e dove il microprocessore stia lavorando. Ho detto che la memoria chip nel vecchio 500 va da \$00000 a \$80000, la memoria infatti è divisa in parti, come una strada con tante casine le quali abbiano il loro indirizzo: non a caso si chiamano indirizzi o locazioni di memoria (*address* in inglese): all'inizio della strada c'è la casa 0, che contiene un byte, la casa dopo ha l'indirizzo 1, che contiene un altro byte, e così via. è usato però il sistema di numerazione *esadecimale*, cioè in base 16. Questo non è un problema, perché da ASMONE si può convertire il numero in qualsiasi momento usando il comando <?>: facendo “?\$80000” si avrà un 524288 in decimale, che corrisponde a 1024*512, cioè mezzo Kb o “KAPPA RAM”, appunto 1024 bytes, moltiplicato per 512. \$100000 invece è il doppio, ossia un mega... provate ?\$80000*2 (“*” ovvero “MOLTIPLICATO”).

I numeri esadecimali sono preceduti dal dollaro, come hai visto, i numeri decimali non sono preceduti da nulla, quelli binari da un %. Queste cose sono basilari: come per le distanze esiste il metro, il decametro ed il chilometro, per la memoria esiste il **bit**, il **byte**, la **word** e la **longword**. Il *bit* è la parte più piccola di memoria; il *byte*, composto da 8 bit, è una unità che ha il suo indirizzo. Il processore cioè può dire: muovi (o meglio copia) il byte che è nella casina in “via della memoria n10” nella casina in “via della memoria n16”, in questo caso ha copiato gli otto bit che erano nel byte 10 (ovvero \$A in esadecimale o HEXadecimale) nel byte 16.

Per evitare confusioni, facciamo l'esempio inequivocabile: i bit possono essere a 0 o ad 1; nel byte 10 i bit erano: 00110110, nel byte 16 invece 11110010, dopo il MOVE.B 10, 16 il byte 10 rimane 00110110, il byte 16 diventa 00110110. Il .B al MOVE significa che viene mosso un byte, cioè la parte più piccola che si possa copiare. Si può anche fare un MOVE.W ed un MOVE.L, ossia muovere una *word* (.w) o una *longword* (.l), che non sono altro che: 1 word = 2 bytes, una longword = 4 bytes, ovvero 2 word. Allora se si fa un MOVE.W 10, 16, nel byte 16 verrà copiato il byte 10, nel byte 17 il byte 11, ossia viene spostato un blocco di 2 bytes. Nel caso di un MOVE.L vengono spostati 4 bytes, ossia: nel byte 16 il byte 10, nel 17 l'11, nel 18 il 12, nel 19 il 13. Facciamo uno schemino:

```
PRIMA DEL MOVE.L 10,16 ; 08/09/10/11/12/13/14/15/16/17/18/19/20
                        C A N E      G A T T O
```

```
DOPO IL MOVE.L 10,16 ; 08/09/10/11/12/13/14/15/16/17/18/19/20
                        C A N E      C A N E O
```

```
Se facciamo MOVE.B 20,14 ;08/09/10/11/12/13/14/15/16/17/18/19/20
                        C A N E O      C A N E O
```

Nella nostra supposizione le locazioni 08, 09, 14, 15 erano azzerate, mentre le 10-13 e le 16-20 avevano un valore, qua delle lettere per esempio. Concludiamo con un MOVE.W 8,10:

;08/09/10/11/12/13/14/15/16/17/18/19/20
N E O C A N E O

Con 3 istruzioni abbiamo trasformato CANE GATTO in NEO CANEO!!!! A parte gli scherzi, non proseguite a leggere fino a che non vi è rimasto impresso nella memoria cerebrale il funzionamento della memoria sintetica!!!! Fate un po' di giochetti coi MOVE.X, che vi fa bene! Provate ad esempio a trasformare ANTANI in TANTI NANI con vari MOVE, oppure SBLINDO in DOBLONI, oppure RENULOZ in ZUZZURELLONE, eccetera. Ricordatevi che le **istruzioni** del processore devono essere ad indirizzi pari, tipo 2, 4, 6... ossia allineati a *word*, oppure va tutto in GURU.

Per togliere dubbi, nella memoria ci sono una serie di valori uno dietro l'altro, che possono essere istruzioni del 68000, o dati come ad esempio le *sinustab* prima citate, figure, suoni, scritte da visualizzare... le istruzioni in memoria non sono nella forma MOVE.B 10,16, quella è una versione DISASSEMBLATA, in memoria ad esempio quella istruzione occupa 10 bytes, ed è: \$13,\$F9,\$00,\$00,\$00,\$0A,\$00,\$00,\$00,\$10, in cui \$13f9 significa in grandi linee MOVE.B, \$0000a è 10 in esadecimale e \$10 è 16 in esadecimale... allo stesso modo ogni istruzione ha i suoi bytes, ad esempio l'istruzione NOP, ossia no operation, che non fa nulla, in memoria è \$4e71. Anticipo che oltre che operare sulla memoria il processore ha a disposizione dei registri, denominati registri dati e registri indirizzi, che sono 16 e lunghi una longword ciascuno, chiamati a0, a1, a2, a3, a4, a5, a6, a7 gli Address reg, d0, d1, d2, d3, d4, d5, d6, d7 i data reg; sono dentro il processore e quindi sono molto più veloci le operazioni tra 2 registri di quelle tra 2 indirizzi di memoria, ad esempio MOVE.L d0, d2 sarà più veloce di MOVE.L \$100, \$200; si preferisce quindi fare operazioni mettendo i numeri nei registri piuttosto che nella memoria, se possibile.

La ROM come già detto non si può scrivere, cioè un MOVE che scrive nella ROM non ha effetto: un MOVE su \$FC0000 o su \$f80000 non serve a niente. Si possono solo eseguire le *routines* contenute nel ROM. Ma **essendo il Kick diverso in ogni versione, mai si deve saltare al Kick direttamente**. Il sistema operativo è fatto in modo che le *routines*, ossia i singoli programmi presenti nel kickstart, possano essere chiamate nello stesso modo qualunque sia il kick e ovunque sia collocato in memoria: questo viene fatto tramite dei JSR, ossia dei *jump to subroutine* (Salta ad un indirizzo, dopodiché ritorna e continua da sotto il JSR), che sono fissi partendo però dall'indirizzo presente nell'indirizzo 4, in cui è sempre presente l'indirizzo da cui regolarsi per fare i giusti JSR per eseguire le routines del kickstart.

I programmi per aprire le finestre del workbench o per stampare caratteri, per leggere o scrivere un file su disco devono chiamare la routine presente nel CHIP del kickstart ROM ogni volta, passandogli ad esempio il nome del file da caricare o le dimensioni della finestra da aprire; invece quando un gioco o una demo "salta" il sistema operativo non vengono fatte chiamate al kickstart: ad esempio il noto XCOPY apre un suo schermo, ed appare evidente che toglie di mezzo il multitasking e non ha le finestre ed i menù da tasto destro come i programmi da sistema operativo. Allo stesso modo un gioco come quelli che ho menzionato prima, come SENSIBLE SOCCER, funzionerebbe anche se dopo il boot (la partenza) si rimuovesse il chip del kickstart, in quanto non vengono chiamate routines per aprire finestre o caricare file: le cose che appaiono a video sono controllate una per una e i dati dal disco sono caricati non come files DOS, ma come tracce lette direttamente spostando le testine del DRIVE dando corrente o meno ai pin del cavo.

Appare chiara questa differenza? Tra i programmi o giochi che *usano* il sistema operativo, ossia richiamano continuamente routines nella ROM e mantengono il multitasking e le finestre, e gli altri prog. che non aprono finestre o le aprono in maniera diversa dal WorkBench, e non possono essere eseguiti insieme al Deluxe Paint, scambiando la finestra o spostandola in basso??

Insomma la ROM si preoccupa di dialogare con l'hardware per noi se glielo chiediamo, e fa un certo numero di cose prestabilite, mentre se decidiamo di dialogare NOI con l'hardware, possiamo fare tutto il possibile, sempre che ne siamo capaci!!!

Or dunque ci occuperemo di fare codice senza usare la ROM. Ma allora useremo solo il micro-processore? e come si fa a visualizzare un'immagine o suonare una musica? con dei MOVE???? Ora entrano in gioco i **chip custom**!!! Questi *chip* si chiamano *Paula*, *Agnus* e *Denise*, inoltre ci sono altri 2 *chip* detti *CIAA* e *CIAB*. Questi furboni sono quelli che fanno suonare l'amiga e che gli fanno visualizzare tutti quei colori sullo schermo. La maggior parte dei registri in questione si trovano alla locazione `$dff000` fino a `$dff1fe`, altri riguardanti le porte seriali, parallele, e dei disk drives si trovano in zona `$bfexxx` o `$bfdxxx`.

Una volta imparate tutte le istruzioni del 68000 si possono costruire programmi grossi come case, ma se si sposta memoria qua e la non si visualizza né si suona nulla! col processore bisogna pilotare questi chip; uno principale è il **Blitter**, che si occupa di disegnare le linee, copiare pezzi di memoria come scroll o ometti in giro per lo schermo, riempire aree (i solidi 3d sono disegnati e riempiti con il blitter; il processore si occupa di calcolare le coordinate delle linee che poi il Blitter disegna).

Quello che però visualizza il tutto e che determina i colori è il **Copper**: per fare un esempio il `$dff180` corrisponde al colore 0 ed il `$dff182` al colore 1, mentre nel `$dff006` c'è la linea dove il pennello elettronico è arrivato nel disegnare lo schermo, che viene disegnato 50 volte al secondo: questi registri infatti sono a SOLA lettura o a SOLA scrittura, ad esempio nel `$dff180` si può mettere un valore, ma non si può leggere che valore c'è, lo stesso vale per il `$dff006`, sul quale non si può scrivere; per cambiare la posizione al pennello elettronico esiste comunque un apposito registro, così per molti altri. Nei registri `$bfexxx` si può controllare il disk drive o le varie porte, tra cui quella del mouse: ad esempio al bit 6 dell'indirizzo `$bfe001` corrisponde lo stato del bottone sinistro del mouse, se questo è premuto o no, e si può controllare col processore ed aspettare che sia premuto prima di uscire. Ed è questo il primo esempio di programmazione che puoi analizzare caricando LEZIONE1a.s, il primo sorgente del corso, che comprende insieme un ciclo con il 68000, l'utilizzo di un registro `$dffxxx` e di uno `$bfexxx`. (caricatelo in un altro buffer di testo come spiegato sotto).

Un breve accenno su come usare l'assemblatore, in questo caso ASMONE. All'inizio si deve selezionare se allocare memoria CHIP o FAST, è bene selezionare quella chip per i sorgenti del corso, a seconda di quanta ne avete, selezionate il numero di Kb, almeno 250. Per selezionare una directory o un drive usate il comando `<v>`, per esempio per andare nella directory delle lezioni fate un "V df0:LEZIONI", per andare nella directory dei sorgenti fate un bel "V df0:SORGENTI", poi per leggere il sorgente o la lezione usate `<R>`, e selezionatelo con la finestrella. Si può scambiare con `<ESC>` tra la funzione di editor e la linea di comandi; cioè premete ESC e potete scorrere o modificare il testo, ripremete `<ESC>` e tornate alla linea di comandi dove potete ad esempio ASSEMBLARE il listato con `<A>`, dopodiché per farlo eseguire dovete premere `<J>` (JSR!!).

Potete anche caricare simultaneamente 10 testi, siano essi sorgenti o lezioni, perché, quando siete in modo EDIT, quando cioè potete scorrere il testo con il cursore e potete cambiarlo, se premete F2 scambierete listato, e andrete al secondo, che in questo caso sarà vuoto: se premete nuovamente F1 ritornate al testo caricato prima: in questo modo potete, ad esempio, tenere nel buffer 1 (ossia listato 1 richiamabile con F1) la Lezione1.TXT, mentre nel buffer 2, selezionabile con F2, potete caricare il listato inerente alla lezione1, ossia Lezione1a.s. In seguito potete mettere lezione1 nel buffer 1, lezione2 nel buffer2, nel buffer 3,4,5 dei listati della lezione2, eccetera, quindi potrete consultare la lezione, poi premendo un `<F4>` o un `<F5>` verificare subito l'esecuzione di un listato, o ritornare a vedere una cosa della lezione1 che non ricordate, eccetera.

NOTA: per scorrere di pagina in pagina usate i cursori (freccie) più lo `<SHIFT>`, ossia, per chi non aveva il C64, il tasto grande sopra `<ALT>` con la freccia. Vi spiego che succede quando fate `<A>`: il listato (o sorgente) è in formato testo normalissimo, ed è fatto di parole chiave che sono i comandi o altri simboli che conosce l'assemblatore... per segnare un gruppo di istruzioni o una

“variabile”, o l’inizio di una tabella, o comunque avere un riferimento di un preciso punto del listato, si fanno delle ETICHETTE o LABEL, che non devono avere spazi dall’inizio del bordo, e devono finire con i : (DUE PUNTI). Il nome della label è a scelta, ma non si deve dare un nome che sia uguale ad un comando 68000!!! esempio:

```

1 WAITMOUSE:      ; la label
2   btst #6,$bfe001 ; tasto sinistro premuto?
3   bne.s WAITMOUSE ; se no torna a WAITMOUSE (ripeti il btst)
4   rts           ; Esci

```

Vi ricordo che i comandi devono avere una spaziatura, in questo caso ho usato il <TAB> (Il tasto sopra <CTRL> e <CAPS LOCK>), che fa 8 spazi con un colpo solo... Notate che non vanno messi i : (due punti) finali al nome della label (o etichetta) quando viene richiamata, ma solo a se stessa. Dunque, una volta editato il sorgente, va ASSEMBLATO con <A>; questa operazione fa leggere all’ASMONE il testo, e lo trasforma in codice, cioè nei bytes che saranno letti dal 68000 ed eseguiti come istruzioni. Una volta assemblato, il suddetto codice è in un punto della memoria che si può vedere con <=R>, e con il comando <J> il processore salta a quel punto della memoria ed esegue il nostro programma. Se l’ASMONE trova un errore nel listato non assembla tutto fino a che non viene corretto l’errore. I sorgenti del corso funzionano anche con altri assembler come DEVPAC 3 e MASTERSEKA, con tutti i kickstart e con tutti gli Amiga, compresi quelli AGA come il 1200 o il 4000.

Se avete verificato il funzionamento di Lezione1a.s, caricate in un altro buffer di testo (quello <F3>, ad esempio) il file LEZIONE2.TXT con <R>.

Se mancasse memoria quando scambiate buffer, significa che avete selezionato troppa memoria all’inizio (al messaggio ALLOCATE), e non ne è rimasta per la RAM DISK. La prossima volta selezionatene meno.

CAPITOLO 2

LEZIONE 2

Avete capito esattamente come funziona il sorgente LEZIONE1a.s??? Se non lo avete capito allora siete da neuro, e dovete smettere il corso.

Ora andiamo ad approfondire il linguaggio del 68000. Ho voluto anticipare col primo sorgente il fatto che il processore serve grossomodo per organizzare tutte le cose, ma che da solo non fa che cambiare valori nella sua memoria; mettendo certi valori in aree particolari di memoria come `$dffxxx` o `$bfexxx` si dà corrente ai piedini dei chip della grafica, del suono e delle porte, e di conseguenza si può, come nell'esempio precedente, cambiare il colore dello schermo, o leggendo queste locazioni sapere a che linea il pennello elettronico sia arrivato o se il bottone del mouse è premuto. Per fare un gioco o una demo è necessario usare un gran numero di questi indirizzi, detti REGISTRI, e quindi è necessario conoscerli almeno quanto il linguaggio del 68000, (MOVE, JSR, ADD, SUB etc.) con cui si impostano. Per la programmazione di questo tipo come ho già detto non si usano le *librerie* del ROM kickstart 1.2/1.3/2.0/3.0 (Ovvero le sue routines, o sotto-programmi che consentono di aprire una finestra workbench o leggere un file, ad esempio) cioè si usano pochissimo: ad esempio per evitare di far andare in guru il workbench o per disabilitare il multitasking. Ritengo necessario quindi in questa lezione n.2 di approfondire l'uso del 68000, una volta che si è capito quale sia il suo ruolo.

La cosa più importante da imparare sono i modi di indirizzamento del processore, più che i comandi in se, infatti una volta imparato quello, ogni comando usa la stessa sintassi per l'indirizzamento e basta sapere che cosa fa il comando. Abbiamo già detto che il processore opera sulla memoria che è divisa in locazioni o indirizzi, la cui unità di misura è il byte, e solitamente l'indirizzo è in formato esadecimale, cioè in un formato numerico diverso da quello decimale, avendo infatti base 16. Questo non è assolutamente un problema: mentre con i numeri decimali una sequenza di 30 numeri ad esempio fa: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 etc. . . , in esadecimale fa 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1a, 1b, 1c, 1d, 1e etc. . . , cioè nei numeri esadecimali si trovano anche le prime 6 lettere dell'alfabeto come se a fosse un 10, b un 11 eccetera; per convertire un numero da esadecimale a decimale o viceversa basta usare il comando `<?>` dell'asmone: per esempio facendo `<?10000>` si otterrà `$2710`, il corrispondente valore in esadecimale (i numeri esadecimali cominciano col \$, quelli decimali non sono preceduti da niente e quelli binari da un %). I numeri esadecimali

sono usati perché sono più vicini al modo di pensare del computer che è ovviamente *binario*, cioè composto di soli 0 ed 1. Come esempio per iniziare a capire i vari modi di indirizzamento del 68000 useremo il comando CLR, che azzerla la locazione di memoria indicata:

```
1 CLR.B $40000 ; vi ricordate la differenza tra .B, .W e .L?
```

Questa istruzione “pulirà”, cioè azzererà il byte n. \$40000, ossia l’indirizzo \$40000. Questo è il caso più semplice, detto *assoluto*; cioè si indica direttamente in che indirizzo fare il CLR; nell’assemblatore sono in uso le LABEL, che servono ad identificare un punto del programma, in cui può esserci per esempio un byte da indicare: in questo caso invece di scrivere l’indirizzo si scriverà il nome della LABEL; l’assemblatore allora scriverà l’indirizzo effettivo del byte in questione: ad esempio, se modificassimo così il nostro primo listato:

```
1 Waitmouse:
2     move.w $dff006,$dff180 ; metti il valore di $dff106 in $dff180
3                               ; cioè il VHPOSr nel COLORE
4     btst #6,$bfe001 ; tasto sinistro del mouse premuto?
5     bne.s Waitmouse ; se no ritorna a waitmouse e ripeti
6                               ; (il .s è un equivalente del .b per questo
7                               ; tipo di comandi: bne.s = bne.b)
8     clr.b dato1 ; AZZERA DATO1
9     rts ; esci
10
11 dato1:
12     dc.b $30 ; dc.b significa METTI IN MEMORIA IL SEGUENTE BYTE
13             ; in questo caso viene messo un $30 sotto dato1:
```

Prima di uscire con l’RTS si azzererebbe il byte contrassegnato con la label dato1:, che sarebbe allocato nella fase di assemblaggio (o compilazione) a qualche indirizzo assoluto ben preciso, ad esempio se il programma fosse assemblato dall’ASMONE a partire da \$50000, si troverebbe in memoria dopo l’assemblaggio un CLR.B \$5001c, cioè l’indirizzo reale di dato1:, non certo CLR.B DATO1, essendo dato1: un nome dato dal programmatore per contrassegnare il dc.b \$30; di qui si intuisce anche l’utilità delle label, infatti se si dovesse scrivere il listato indicando l’indirizzo numerico tutte le volte, nel caso che si aggiungesse una routine nel mezzo del programma si dovrebbero riscrivere tutti gli indirizzi. Per vedere a quale indirizzo vengono assemblate le label, basta usare il comando <D> dell’ASMONE: ad esempio dopo aver assemblato LEZIONE1a.s facendo <D waitmouse> otterrete il disassemblato della memoria a partire da waitmouse, e nel listato non appariranno le label, ma gli indirizzi reali.

Nei sorgenti esempio del corso noterete che non vengono mai indicati indirizzi numerici, ma solo LABEL, a parte gli indirizzi speciali come \$dffxxx o \$bfexxx. Nell’ultimo esempio ho usato un dc.b, che è un comando dell’assemblatore che serve per inserire bytes definiti; per esempio per inserire un \$12345678 in un dato punto del programma, dovrò usare il comando DC, e lo posso usare nelle 3 forme .B (BYTE), .W (WORD) e .L (LONGWORD):

```
1     dc.b $12,$34,$56,$78 ; in bytes
2
3     dc.w $1234,$5678 ; in words
4
5     dc.l $12345678 ; in longwords
```

Questo comando si usa anche per mettere delle frasi in memoria, ad esempio per mettere nel listato il testo che dovrà essere stampato a video da una routine di PRINT che stampi ciò che è alla label TESTO:

```
1 TESTO:
2     dc.b "tanti saluti"
3
4 oppure:
5     dc.b 'tanti saluti'
6
7 Di solito si termina il testo con uno zero:
8
9     dc.b "tanti saluti",0
```


Bisogna ricordarsi di mettere il testo tra virgolette e di usare il `dc.b`, non il `dc.w` o il `dc.l`!! I caratteri sono lunghi un byte ciascuno, e corrispondono ad un certo byte: ad esempio provate a fare `<?"a">`, e vedrete che corrisponde a \$61, quindi scrivere `dc.b "a"` sarà equivalente a scrivere `dc.b $61`. Attenzione che le lettere grandi hanno valore diverso! un "A" per esempio è \$41. L'uso più comune del `dc.b` è quello di definire byte, word, o zone più grandi dove verranno tenuti dei dati, ad esempio se si volesse fare un programma che registri il numero di volte che si preme un certo tasto, bisognerà definire una label seguita per esempio da un byte azzerato, ed ogni volta si aggiungerà 1 con il comando `ADD` a quella label, ossia a quel byte sotto la label, e all'uscita basterà leggere il valore del byte:

```

1      ; se il tasto è premuto allora ADDQ.B #1,NUMPREMUTO, ossia
2      ; aggiungi uno al byte sotto la label numpremuto.
3
4 NUMPREMUTO:
5      dc.b 0

```

All'uscita del programma lo 0 iniziale sarà cambiato nel numero di volte che il tasto è stato premuto. Un esempio simile è in `LEZIONE2a.s`, che contiene anche un ampio commento. Vi consiglio di caricarlo in un altro buffer di testo: per selezionare uno dei 10 disponibili basta premere un tasto da `<F1>` a `<F10>`, se per esempio avete `LEZIONE2.TXT` nel buffer di `<F1>`, premete `<F2>`, e caricateci `LEZIONE2a.s` con il comando `<R>`. In seguito potete caricare `LEZIONE2b.s` e i seguenti nel buffer di `<F3>`, `<F4>`..., in modo da averli sempre a disposizione premendo un solo tasto; è meglio comunque se seguite la `LEZIONE.TXT`, e man mano che trovate l'indicazione del sorgente esempio, continuate caricandolo in un altro buffer, eseguendolo e verificandolo, dopodiché ritornate a leggere la `LEZIONE` da dove eravate: questo credo sia il miglior sistema per imparare, infatti si fa un po di teoria e si verifica subito.

Avete letto i commenti di `LEZIONE2a.s`?

Avrete visto l'importanza che hanno il byte, la word e la longword: per quanto riguarda il binario, per contare i bit, si comincia da destra e si va verso sinistra, al "contrario" insomma, e si parte da 0, non da 1, dunque un byte (che ha 8 bit) parte 0 e va fino al 7. Per esempio in questo numero:

```

1      %000100010000

```

Sono "accesi" i bit 4 e 8. Per aiutarvi a numerarli potete fare così:

```

1      ;5432109876543210      - un utilizzo intelligente del ;
2      move.w  #%0011000000100100,$dffxxx

```

In questo caso sono "accesi" i bit 2, 5, 12 e 13 della WORD. Ricordo che un byte ha 8 bit, una word ne ha 16 (da 0 a 15), una longword ne ha 32 (da 0 a 31). Nell'istruzione

```

1      BTST #6,$bfe001

```

Si controlla se il bit 6 del byte `$bfe001` è azzerato. Se fosse:

```

1      ;76543210
2      %01000000

```

Il bit 6 è invece ad 1, dunque il mouse non è premuto!!!

Per ricapitolare, un byte è fatto di 8 bit: per indicarli, il primo a destra è il bit 0, detto anche *bit meno significativo*. La numerazione procede da destra verso sinistra fino al 7, (ossia l'ottavo perché si parte da 0 anziché da 1: 01234567, ossia 8 bit); il bit 7 è detto *bit più significativo*. È più significativo perché conta di più allo stesso modo in cui conta di più, nel bigliettone da centomila, l'uno più a sinistra degli zeri più a destra. Un byte, al massimo può valere 255, ossia %11111111.

Una WORD invece è fatto di 16 bit, ovvero due byte, allo stesso modo si parte da destra col bit 0, sempre il meno significativo, fino al bit 15 ultimo a sinistra, il più significativo. Al massimo può contenere 65535.

Una LongWord è fatta di 32 bit, da 0 a 31, ossia 4 bytes, o 2 word, o se preferite una word e 2 bytes, insomma sempre 32 bit attaccati l'uno all'altro che al massimo possono contenere 4294967299 (4 miliardi!! come la lotteria!).

Ora procederemo con diversi modi di indirizzare: abbiamo visto che se facciamo ad esempio un CLR.W \$100, azzereremo le locazioni \$100 e \$101, ossia una word a partire da \$100 (essendo una word 2 bytes, e le locazioni divise in bytes, puliremo 2 bytes!!). Allo stesso modo un MOVE.B \$100,\$200 copierà il contenuto di \$100 in \$200. Questo si può indicare anche con le LABEL invece di specificare l'indirizzo, ad esempio MOVE.B LABEL1,LABEL2, ovvero copia il byte di LABEL1 in LABEL2. Ci sono però anche diversi modi di indirizzare, infatti posso fare un MOVE.L #\$50000,LABEL2, ossia mettere un valore *fisso* in LABEL2. Se ad esempio LABEL2 fosse all'indirizzo \$60000, muoveremmo il valore \$00050000 in \$60000, ovvero i bytes facendo un <M \$60000>: 00 05 00 00. Infatti quando c'è il simbolo del cancelletto (#) prima di un numero o di una label significa che si sta muovendo un valore stabilito, e non il valore contenuto nell'indirizzo indicato con quel valore, come avviene se non ci sono cancelletti prima del numero o della LABEL. Per esempio analizziamo questi 2 casi:

```

1 1)      MOVE.L  $50000,$60000    ; il valore contenuto negli indirizzi
2                                     ; di memoria $50000,$50001,$50002,$50003
3                                     ; vengono copiati in $60000,
4                                     ; $60001,$60002,$60003, ossia una longword
5                                     ; composta di 4 bytes viene copiata da un
6                                     ; indirizzo all'altro.
7
8 2)      MOVE.L  #$50000,$60000    ; Questa volta in $60000 viene messo il
9                                     ; numero indicato dopo il cancelletto,
10                                    ; ossia $50000. Da notare che questa
11                                    ; volta l'indirizzo $50000 non viene letto
12                                    ; e non c'entra assolutamente, viene
13                                    ; implicato solo il $60000.

```

Se si usano delle label, non ci sono cambiamenti:

```

1 1)      MOVE.L  CANE,GATTO        ; Il contenuto della longword cane, ossia
2                                     ; $00123456 viene copiato nella longword
3                                     ; GATTO (infatti $123456 è la prima cosa
4                                     ; sotto la label CANE)

```

prima dell'istruzione:

```

1 CANE:
2     dc.l      $123456
3
4 GATTO:
5     dc.l      0

```

dopo l'istruzione:

```

1 CANE:
2     dc.l      $123456
3
4 GATTO:
5     dc.l      $123456
6
7 2)      MOVE.L  #CANE,GATTO        ; Questa volta l'INDIRIZZO della label
8                                     ; CANE viene copiato nella label GATTO

```

prima dell'istruzione:

```

1                                     ; SUPPONIAMO CHE LA LABEL CANE: sia alla locazione
2                                     ; $34500, cioè facendo un M CANE dopo aver assemblato
3                                     ; compare un :
4                                     ; 00034500 00 12 34 56 00 00 00 00 .....
5                                     ;          (cane)      (gatto)
6
7 CANE:
8     dc.l      $123456
9

```

```

10 GATTO:
11     dc.l    0

```

dopo l'istruzione:

```

1 CANE:
2     dc.l    $123456
3
4 GATTO:
5     dc.l    $34500 ; ossia DOVE è LA LABEL CANE IN MEMORIA.

```

Da notare che se si faceva un `MOVE.W #CANE,GATTO` o un `MOVE.B #CANE,GATTO` l'assemblatore avrebbe dato un errore, in quanto un *indirizzo è lungo una longword*. In memoria un `MOVE.L #LABEL,LABEL` si trasforma in un'istruzione del tipo `MOVE.L #$12345,$12345`, ossia l'assemblatore scrive l'indirizzo reale al posto delle label. Questo lo potete verificare con LEZIONE2b.s.

Ora affronteremo gli altri indirizzamenti con i registri (che sono più difficili); come avevo già accennato, ci sono 8 registri dati e 8 registri indirizzi: ossia D0,D1,D2,D3,D4,D5,D6,D7 sono i registri *dati*, mentre a0,a1,a2,a3,a4,a5,a6,a7 sono i registri *indirizzi*. Premetto che il registro A7 è detto anche SP o *stack pointer*, ed è un registro particolare di cui parleremo dopo, quindi considerate di usare i registri indirizzi solo fino ad a6. questi indirizzi sono lunghi una longword ciascuno, ed sono in pratica una piccola memoria dentro il 68000, che di conseguenza è molto veloce. Tramite i registri si possono fare varie cose, infatti esiste una sintassi particolare per i registri. Innanzitutto non si può lavorare per byte con i registri INDIRIZZI: per esempio un `move.b LABEL,a0` dà un messaggio di errore. Con i registri indirizzi a0, a1, etc... si può dunque lavorare per longword o per word. Con i registri Dati D0, D1, etc..., invece si possono usare sia .b che .w che .l. I registri indirizzi sono dedicati a contenere indirizzi, ed hanno comandi dedicati, come il LEA, che significa LOAD ENTIRE ADDRESS, ovvero carica l'indirizzo interamente nel registro (infatti questo comando non può essere `lea.b`, `lea.w` o `lea.l`, ma solo LEA essendo sempre .L) Per esempio, per mettere un valore nei registri indirizzi si possono usare 2 metodi:

```

1 1)      MOVE.L  #$50000,a0      (oppure MOVE.L #LABEL,a0)
2
3 2)      LEA     $50000,a0      (oppure LEA LABEL,a0)

```

Mentre il primo metodo si può usare sia con gli indirizzi che con i registri (es: `move.l #$50000,d0` - `move.l #$50000,LABEL` - `MOVE.L #LABEL,LABEL...`)

P.S: scrivere `move.l #$50000,d0` o `MOVE.L #$50000,D0` è identico, si può scrivere anche `MoVe.L #$50000,d0`, il risultato a livello di programma è identico, solo che esteticamente potete creare delle situazioni simpatiche o orribili. Va fatto un discorso diverso per le LABEL: le label possono essere identificate anche se in un punto del listato le scrivete in minuscolo e in un altro in maiuscolo, questo però solo perchè è settata nelle preferenze del TRASH'M-ONE questa opzione, che è "UCase=UCase" nel menù "Assembler/Assemble..", che significa "Upper Case=Lower Case, ossia lettere grandi=lettere piccole". Se togliete questa opzione, nel riconoscimento delle label sarà tenuto presente anche il maiuscolo/minuscolo, per cui Cane: sarà diverso da CANE: o da cAne: o da caNe, eccetera eccetera.

il secondo metodo con il LEA si può usare solo con i registri indirizzi, di conseguenza si intuisce che questo modo è più veloce: ricordatevi quindi che se volete mettere un indirizzo in un registro a0, a1... dovete usare il LEA seguito dall'indirizzo SENZA CANCELLETTO e dal registro in questione. Fate attenzione a questi 2 esempi:

```

1 1)      MOVE.L  $50000,a0      ; metti in a0 il valore contenuto nella
2                                ; locazione $50000 (+$50001,$50002 e $50003
3                                ; in quanto 1 locazione è lunga 1 byte, ed
4                                ; il move.l copia 4 bytes = 4 locazioni a
5                                ; partire in questo caso da $50000
6
7 2)      LEA     $50000,a0      ; metti il numero $50000 in a0

```

State attenti quindi a maneggiare i MOVE con o senza il cancelletto ed i LEA, perché è facile nei primi tempi sbagliarsi e mettere l'indirizzo invece del valore di quell'indirizzo nel registro o viceversa. Come ulteriore commento di questa differenza consultate il programmino esempio LEZIONE2c.s

Con i registri indirizzi sono possibili vari tipi di indirizzamento: Per cominciare analizziamo queste 2 istruzioni:

```

1      move.l  a0,d0      ; Metti il numero contenuto in a0 nel registro d0
2      move.l  (a0),d0    ; Metti la longword contenuta dall'indirizzo in a0
3                          ; nel registro d0

```

L'indirizzamento tra parentesi si dice *indiretto*, perché anziché venir copiato *direttamente* il valore in a0 viene copiato il valore contenuto nell'indirizzo che è in a0. Un esempio pratico è in LEZIONE2d.s

Usando l'indirizzamento indiretto si può agire sugli indirizzi *indirettamente*, ad esempio mettendo l'indirizzo del tasto del mouse e del colore 1 nei registri si può riscrivere il listato della lezione 1. Così ho fatto in LEZIONE2e.s

Facciamo gli ultimi esempi per togliere gli eventuali dubbi sull'indirizzamento indiretto:

```

1      move.l  a0,d0      ; copia il valore di A0 nel reg. d0
2      move.b  (a0),d0    ; copia il byte contenuto nell'indirizzo
3                          ; in a0 nel reg. d0
4      move.w  (a0),(a1)   ; copia la word contenuta dall'indirizzo
5                          ; in a0 all'indirizzo contenuto in a1
6                          ; (e seguente, essendo una word fatta di
7                          ; 2 bytes, ossia 2 indirizzi!)
8      clr.w   (a3)        ; pulisce (azzerà) la word (2 bytes) "dentro"
9                          ; l'indirizzo in a3 – Più precisamente,
10                         ; viene azzerato il byte dell'indirizzo in
11                         ; a3 e l'indirizzo seguente.
12      clr.l   (a3)        ; Come sopra, ma sono azzerati 4 indirizzi
13                         ; (una long = 4 bytes = 4 indirizzi)
14      move.l  d0,(a5)     ; viene copiato il valore di d0 nell'indirizzo
15                         ; contenuto in a5 (più precisamente dovrei
16                         ; dire nell'indirizzo in a5, e nei 3 seguenti,
17                         ; in quanto una long occupa 4 indirizzi)
18      move.l  d0,a5       ; viene copiato il valore di d0 in a5

```

Mi raccomando! Toglietevi ogni dubbio sugli indirizzamenti fin qui studiati, consultando anche i sorgenti fino a LEZIONE2e.s, perché gli indirizzamenti di cui parlerò ora si basano su quelli indiretti normali.

Vi comunico che questa è la parte più astratta della lezione2, in quanto si devono imparare gli ultimi indirizzamenti del processore, ma vi assicuro che già dalla lezione 3 metterete in pratica il tutto e visualizzerete degli effetti video col copper!, quindi considerate che passata questa parte il resto del corso sarà tutto più PRATICO: a ogni spiegazione corrisponderà un nuovo effetto speciale o colore ultravivace, dunque fate lo sforzo di non annoiarvi e di non lasciar perdere ora, perché io stesso lasciai perdere all'incirca a questo punto la prima volta che tentai di imparare a programmare in ASM, proprio perché ero scoraggiato dal CASINO di comandi e parentesi aperte e chiuse che poi non riuscivo più a seguire. Vi assicuro però che una volta imparato a leggere i comandi, potete partire come una fucilata e imparare da voi leggendo listati qua e là, facendo passi sempre più grandi: è come imparare le regole di uno sport: uno che non conosce il set di istruzioni del 68000 è come uno che non conosce le regole, ad esempio, del calcio: guardando le partite (i listati) costui non capirà nulla di cosa stanno facendo quegli scalmanati in un campo a dare calci ad una palla, e si annoierà a morte, ma una volta capite le regole (indirizzamenti) potrà interpretare le fasi delle partite ed imparare sempre di più le tecniche di gioco (i trucchi della programmazione ed i registri della grafica).

Vediamo altri 2 modi di indirizzamento:

```

1      move.l  (a0)+,d0    ; Indiretto con post-incremento
2      move.l  -(a0),d0    ; Indiretto con pre-decremento

```

Analizziamo il primo indirizzamento ipotizzando questa situazione:

```

1      lea      NONNO,a0      ; mettiamo in a0 l'indirizzo di NONNO:
2      MOVE.L   (a0)+,d0      ; mettiamo in d0 il valore .L contenuto
3                               ; dall'indirizzo in a0, ossia $3231020
4                               ; (come un normale MOVE.L (a0),d0)
5                               ; dopodichè AGGIUNGIAMO 4 AL VALORE IN a0
6                               ; ovvero andiamo a PUNTARE alla long seguente
7                               ; con l'indirizzo in a0; se fosse stato un
8                               ; move.w (a0)+,d0 ad a0 DOPO (POST-INCREMENTO)
9                               ; sarebbe stato aggiunto 2 (una word=2),
10                              ; mentre nel caso di un MOVE.B (a0)+,d0
11                              ; sarebbe stato aggiunto 1, (un byte),
12                              ; ovvero sarebbe andato a puntare l'indirizzo
13                              ; seguente.
14      MOVE.L   (a0)+,d1      ; stessa cosa: copia in d1 il valore .L
15                              ; contenuto nell'indirizzo in a0, che ora
16                              ; contiene l'indirizzo di NONNO+una longword,
17                              ; ovvero NONNO+4, ossia $13478.
18      rts                               ; ESCE!
19
20 NONNO:
21      dc.l     $3231020,$13478
22
23      END

```

Possiamo tradurre questo tipo di indirizzamento con un 2 istruzioni:

```

1 1)      MOVE.L   (a0)+,LABEL

```

è equivalente a:

```

1 1b)     MOVE.L   (A0),LABEL      ; copia una long dall'indirizzo in a0
2                               ; nella label
3      ADDQ.W   #4,a0              ; Aggiungi 4 ad a0 (.L=4)
4                               ; NOTA: se si addiziona un numero minore di
5                               ; 9 si usa il comando ADDQ invece di ADD
6                               ; perchè è dedicato a tali numeri e veloce.
7                               ; Inoltre su registri INDIRIZZI se il numero
8                               ; che aggiungiamo o sottraiamo è minore di
9                               ; $FFFF, ossia una word, si può usare il .W
10                              ; anzichè il .L, e si agirà comunque su
11                              ; tutta la longword dell'indirizzo.

```

Allo stesso modo:

```

1 2)      MOVE.W   (a0)+,LABEL

```

è equivalente a:

```

1 2b)     MOVE.W   (A0),LABEL      ; copia una word dall'indirizzo in a0
2                               ; nella label
3      ADDQ.W   #2,a0              ; Aggiungi 2 ad a0 (.W=2)

```

Allo stesso modo:

```

1 3)      MOVE.B   (a0)+,LABEL

```

è equivalente a:

```

1 3b)     MOVE.B   (A0),LABEL      ; copia il byte contenuto nell'indirizzo
2                               ; in a0 nella label
3      ADDQ.W   #1,a0              ; Aggiungi 1 ad a0 (.B=1)

```

Dunque, riassumendo in altri termini, l'indirizzamento indiretto con post incremento si può paragonare ad un operaio di una catena di montaggio che *prima* esegue il suo MOVE o la sua istruzione sul pezzo che sta sul nastro trasportatore, e ogni volta che ha fatto il suo lavoro sul pezzo sposta *avanti* il nastro trasportatore (l'indirizzo in a0) con un pedale (il +). Un esempio di loop può risultare più chiaro:

```

1  Inizio:
2      lea    $60000,a0      ; inizio pulizia
3      lea    $62000,a1      ; fine pulizia
4  CLELOOP:
5      clr.l  (a0)+          ; azzera una long dall'indirizzo in A0 e aumenta a0
6                          ; di una long, ossia di 4 indirizzi, in altre
7                          ; parole pulisci una long e vai alla prossima
8      cmp.l  a0,a1          ; A0 è arrivato a $62000? Ossia, a0 è uguale ad a1?
9      bne.s  CLELOOP        ; se non ancora, continua con un altro ciclo CLELOOP
10     rts

```

Come si vede, questo programmino pulisce la memoria dall'indirizzo \$60000 a \$62000, utilizzando un `clr (a0)+` ripetuto fino a che non si è arrivati all'indirizzo desiderato. Un esempio simile lo potete trovare in `Lezione2f.s`

Ora impareremo l'indirizzamento indiretto con pre-decremento, ossia un indirizzamento opposto a quello appena descritto, infatti invece di aumentare l'indirizzo contenuto nel registro dopo aver eseguito l'operazione, con un `clr.l -(a0)`, per esempio, prima viene decrementato `a0`, poi viene eseguita l'istruzione sul nuovo indirizzo (in questo caso `a0-4`). Esempio:

```

1      lea    NONNO,a0      ; mettiamo in a0 l'indirizzo di NONNO:
2      MOVE.L -(a0),d0      ; a0 viene decrementato, in questo caso
3      rts                  ; essendo un'istruzione .L viene decrementato
4                          ; di 4, dopodichè viene copiato in d0
5                          ; il valore .L contenuto dall'indirizzo
6                          ; in a0, ossia $12345678, cioè NONNO-4
7                          ; (nel registro rimane il valore iniziale -4)
8      dc.l   $12345678     ; se fosse stato un
9  NONNO:    move.w -(a0),d0 ad a0 PRIMA (PRE-INCREMENTO)
10     dc.l   $ffff0f0f     ; sarebbe stato sottratto 2 (una word=2),
11     END      ; mentre nel caso di un MOVE.B -(a0),d0
12             ; sarebbe stato sottratto 1, (un byte),
13             ; ovvero sarebbe andato a puntare l'indirizzo
14             ; precedente.

```

Possiamo tradurre questo tipo di indirizzamento con un 2 istruzioni:

```

1  1)      MOVE.L  -(a0),LABEL

```

è equivalente a:

```

1  1b)     SUBQ.W  #4,a0      ; Sottrai 4 ad a0 (.L=4)
2                          ; NOTA: se si sottrae un numero minore di
3                          ; 9 si usa il comando SUBQ invece di SUB
4                          ; perchè è dedicato a tali numeri e veloce.
5
6      MOVE.L  (A0),LABEL    ; copia una long dall'indirizzo in a0
7                          ; nella label

```

Allo stesso modo:

```

1  2)      MOVE.W  -(a0),LABEL

```

è equivalente a:

```

1  2b)     SUBQ.W  #2,a0      ; Sottrai 2 ad a0 (.W=2)
2      MOVE.W  (A0),LABEL    ; copia una word dall'indirizzo in a0
3                          ; nella label

```

Allo stesso modo:

```

1  3)      MOVE.B  -(a0),LABEL

```

è equivalente a:

```

1  3b)     SUBQ.W  #1,a0      ; sottrai 1 ad a0 (.B=1)
2      MOVE.B  (A0),LABEL    ; copia il byte contenuto nell'indirizzo
3                          ; in a0 nella label

```

Riassumendo con l'operaio come prima, l'indirizzamento indiretto con pre decremento si può paragonare sempre ad un operaio di una catena di montaggio che *prima* sposta *indietro* il nastro

trasportatore (l'indirizzo in a0) con un pedale (il -), poi esegue il suo MOVE o la sua istruzione sul pezzo che sta sul nastro trasportatore. Un esempio di loop:

```

1 Inizio:
2   lea    $62000,a0      ; inizio pulizia
3   lea    $60000,a1      ; fine pulizia
4 CLELOOP:
5   clr.l  -(a0)          ; diminuisci a0 di una long e azzeri quella long
6                   ; in altre parole vai alla precedente long e puliscila
7   cmp.l  a0,a1          ; A0 è arrivato a $60000? Ossia, a0 è uguale ad a1?
8   bne.s  CLELOOP       ; se non ancora, continua con un altro ciclo CLELOOP
9   rts

```

Come si vede, questo programmino pulisce la memoria dall'indirizzo \$62000 a \$60000, utilizzando un `clr -(a0)` ripetuto fino a che non si è arrivati all'indirizzo desiderato (All'indietro però! mentre con `(a0)+` si parte da \$60000 e di 4 in 4 si arriva a \$62000, in questo caso si parte da \$62000 e si arriva a \$60000 indietreggiando di 4 in 4). Vedete Lezione2g.s e Lezione2h.s per verificare gli ultimi 2 indirizzamenti.

Ora impareremo come usare la distanza di indirizzamento: un `MOVE.L $100(a0),d0` copia in d0 la long contenuta dall'indirizzo in `a0+$100`, ossia: se per esempio in A0 avevamo l'indirizzo \$60200, in d0 ci andrà la longword contenuta dall'indirizzo \$60300. Allo stesso modo un `MOVE.L -$100(a0),d0` copierà in d0 la long a partire dall'indirizzo \$60100. Da notare che a0 non cambia di valore: semplicemente il processore calcola ogni volta a che indirizzo operare, facendo la somma tra il valore prima della parentesi e l'indirizzo nel registro tra parentesi. La massima distanza di indirizzamento è da -32768 a 32767 (-\$7FFF, \$8000) Un esempio su questo tipo di indirizzamento è Lezione2i.s

L'ultimo tipo di indirizzamento è questo:

```

1   MOVE.L 50(a0,d0),label

```

che ha sia una *distanza di indirizzamento* (il 50) che un *indice* (il d0): la distanza di indirizzamento e il contenuto di d0 sono tutti sommati per definire l'indirizzo da cui copiare il contenuto. In pratica è come la distanza di indirizzamento, ma in più viene aggiunto anche il contenuto dell'altro registro alla distanza di indirizzamento, che in questo caso però va da un minimo di -128 ad un massimo di +128. Non vi voglio annoiare con altri esempi su questo indirizzamento, potrete verificarlo quando lo troverete nei prossimi listati.

Per terminare la LEZIONE2, che se avete seguito bene vi rende in grado di seguire le operazioni di un qualsiasi programma in ASM, è indispensabile spiegare il ciclo DBRA, che è usato moltissimo: usando un registro dati si possono far eseguire delle istruzioni varie volte, basta mettere nel registro dati (sia esso d0, d1...) il numero di volte-1. Ad esempio la routine che pulisce la memoria fatta con il `CLR.L (a0)+` può essere modificata con un loop DBRA che faccia eseguire la pulizia il numero desiderato di volte:

```

1 Inizio:
2   lea    $60000,a0      ; Inizio
3   move.l #($2000/4)-1,d0 ; Metti in d0 il numero di cicli necessari
4                   ; per cancellare $2000 bytes: cioè
5                   ; $2000/4 (ovvero DIVISO 4, perchè ogni
6                   ; clr.l pulisce 4 bytes), il tutto -1,
7                   ; perchè il loop viene eseguito una volta
8                   ; in più.
9 CLEARLOOP:
10  CLR.L  (a0)+
11  DBRA   d0,CLEARLOOP
12  rts

```

Questa routine pulisce da \$60000 a \$62000 come l'esempio precedente in cui con il comando `CMP` si COMPARA a0 con a1, ossia si verifica se siamo arrivati a \$62000 che è in a1. In questo caso invece viene eseguito il `CLR 2047 volte`, provate infatti a fare `<?($2000/4)-1>` da ASNONE. Il DBRA funziona in questo modo: se per esempio la prima volta in d0 viene messo 2047, viene

eseguito il CLR, poi arrivati al DBRA d0 viene diminuito di 1 e il processore salta nuovamente al CLR, lo esegue eccetera, fino a che d0 è esaurito. Bisogna mettere il numero di cicli necessari meno uno perché la prima volta il ciclo viene eseguito senza decrementare d0.

Come ultimo esempio studiatevi `Lezione21.s`, che ha delle subroutines richiamate con il BSR e il ciclo DBRA in azione, utile per capire la struttura di un programma complesso.

Per terminare vorrei farvi notare la differenza tra un BSR ed un BEQ/BNE: nel caso del BSR `label`, il processore salta ad eseguire la routine sotto la label, fino a che non trova l'RTS, che lo fa tornare ad eseguire l'istruzione sotto il BSR `label`, dunque si può dire che ha eseguito una *sottoroutine*, cioè una routine eseguita in mezzo ad un'altra routine:

```

1 principale:
2     move.l  roba1,d0
3
4     move.l  roba2,d1
5
6     bsr.s   sottoposto
7
8     move.l  roba3,d2
9
10    move.l  roba4,d3
11
12    rts      ; FINE DELLA ROUTINE PRINCIPALE, TORNA ALL'ASMONE
13
14
15 sottoposto:
16     move.l  robaccia,d4
17
18     move.l  robaccia2,d5
19
20     rts      ; FINE DELLA SOTTOROUTINE, TORNA A "move.l roba3,d0", ossia
21             ; sotto il bsr.s sottoposto

```

Nel caso di una *diramazione* beq/bne invece si prende O una strada O l'altra:

```

1 principale:
2     move.l  roba1,d0
3
4     move.l  roba2,a0
5
6     cmp.b   d0,a0
7     bne.s   strada2
8
9     move.l  roba3,d1
10
11    cmp.b   d1,a0
12    beq.s   strada3
13
14    move.l  roba4,d0
15
16    rts      ; FINE DELLA ROUTINE PRINCIPALE, TORNA ALL'ASMONE
17
18
19 strada2:
20     move.l  robaccia,d5
21
22     move.l  robaccia2,d6
23
24     rts      ; FINE ROUTINE, TORNA ALL'ASMONE, non sotto il bne!!!
25             ; qua abbiamo scelto questa strada, e come si trova un RTS
26             ; si torna all'ASMONE!!!
27
28
29 strada3:
30     move.l  robaccia3,d1
31
32     move.l  robaccia4,d2
33
34     rts      ; FINE ROUTINE, TORNA ALL'ASMONE, non sotto il beq!!!
35             ; qua abbiamo scelto questa strada, e come si trova un RTS
36             ; si torna all'ASMONE!!!

```


Lo stesso vale per il `BRA label`, che significa SALTA A label, equivalente del `JMP`, per cui è come un treno che trova uno scambio ai binari, non torna allo scambio quando ha finito il binario!! Arriva alla fine del binario e basta, senza teletrasporti alla star trek all'indietro.

Per un'ultima precisazione sui registri indirizzi, vedetevi `Lezione2m.s`

Per caricare la `LEZIONE3.TXT` potete fare in due modi: o scrivete `<R>` e andate a capo, facendo aprire il requester dove potete selezionare col mouse quale testo caricare (in questo caso `<df0:SORGENTI/LEZIONE3.TXT>`), oppure dovete assicurarvi di trovarvi nella directory giusta con un `<V df0:LEZIONI>` e potete caricarla in seguito con un semplice `<R LEZIONE3.TXT>`

LEZIONE 3 - LA PRIMA COPPERLIST

Ora procederemo nella pratica, ma prima vi consiglio di caricarvi in un buffer di testo il file 68000.TXT che è un riassunto della lezione². Questo potrà esservi utile nel caso che non ricordaste un indirizzamento o una istruzione leggendo i listati di questa lezione, che presuppongono la familiarità con gli indirizzamenti e le istruzioni affrontate prima. Il quel testo sono spiegati tutti gli indirizzamenti, anche quelli che non sono quasi mai usati, dunque leggetelo ma non preoccupatevi se non capite gli indirizzamenti con *indice*, tanto nella lezione³ non saranno usati!

In questa lezione si comincia a visualizzare qualcosa sullo schermo: per fare questo dobbiamo scrivere una **CopperList**, cioè un programma per il chip *Copper* che si occupa della grafica, che abbiamo già usato per cambiare di colore lo schermo (\$dff180 è un registro del copper, che si chiama COLOR00). Per ora però abbiamo solo fatto delle modifiche col processore direttamente nei registri, e come avete potuto notare eseguendo i listati con <AD> un istruzione alla volta, quando mettiamo un valore nel COLOR00 (ossia \$dff180) avviene solo un brevissimo lampo, e subito torna il colore normale del sistema operativo, ossia dell'ASMONE. Solo facendo un ciclo in cui si immette continuamente un numero si può colorare tutto lo schermo, ma una volta usciti dal programmino il colore ritorna inesorabilmente quello normale. Questo avviene perché lo schermo che vediamo con finestre, scritte e tutto il resto è il frutto di una *CopperList*, e precisamente di una *CopperList di sistema*. La copperlist non è altro che una specie di:

```
1  MOVE.W  #$123,$dff180    ; COLOR00 — immetti il colore 0
2  MOVE.W  #$123,$dff182    ; COLOR01 — immetti il colore 1
3  eccetera ...
```

Che viene eseguito continuamente, quindi ecco spiegato perché se col processore cambiamo il colore subito torna quello di sistema: perché la copperlist ridefinisce ogni cinquantesimo di secondo tutti i colori!!!! Intuirete che per visualizzarsi delle figure in pace non è possibile continuare a fare loop tentando di combattere con la copperlist di sistema che ridefinisce simultaneamente tutto, ma dovremo togliere di mezzo la copperlist di sistema e farcene una tutta nostra. *Niente di più facile!* Come ho già premesso, la copperlist non è altro che una sfilza di MOVE che mettono dei valori nei registri del COPPER, ossia quelli \$dffxxx; comunque non sono dei move fatti col processore, ma fatti dal copper stesso, che, non a caso, esegue questa COPPERLIST

indipendentemente mentre col processore stiamo facendo altre cose... questo è uno dei motivi per cui su PC non hanno LIONHEARTH o PROJECT X dell'Amiga.

Dovremo quindi scrivergli proprio un *listato*, come facciamo per il 68000, dopodiché dovremo informare il COPPER dove si trova il nostro per farglielo leggere ed eseguire al posto di quello del WorkBench. Il copper ha **solo** 3 istruzioni, di cui in pratica ne vengono usate solo 2: le due usate sono il MOVE ed il WAIT; quella che non usa nessuno è lo SKIP, quindi di quella ne parleremo solo se la troveremo in un listato di esempio. Il MOVE è *facilissimo*! Avete presente un:

```
1  MOVE.W  #$123,$dff180    ; Immetti il colore RGB nel COLOR00
```

Si traduce in copperlist in:

```
1  dc.w    $180,$123        ; si mettono in memoria direttamente i
2                                     ; numeri col dc.w, tanto basta
3                                     ; impararci 2 istruzioni solamente!
```

Ossia: si deve mettere prima l'indirizzo di destinazione, senza il *\$dff* come abbiamo già visto quando mettiamo *\$dff000* in *a0*, basta fare *\$180(a0)*: allo stesso modo i progettisti hanno pensato bene di risparmiarci la fatica di fare *\$DFF* tutte le volte e così basta mettere il *\$180*, o il *\$182* o qualsiasi altro registro del COPPER, infatti SOLO i registri del Copper possono essere scritti dalla COPPERLIST, e si può accedere solo ai registri **pari**, come *\$180*, *\$182*...mai *\$181*, *\$183*!!!!, inoltre potete modificare solo una WORD alla volta. Come avete visto, la COPPERLIST non viene assemblata come i comandi del 68000 che vengono trasformati da *istruzioni* come RTS, MOVE... a *\$4e75*, etc., bensì si devono mettere i *bytes* come sono realmente in memoria e come sono letti dal coprocessore COPPER: per la COPPERLIST appunto dobbiamo usare il comando DC per metterla in memoria a forza di BYTES, ma è facilissimo. Per esempio per definire i primi 4 colori:

```
1  COPPERLIST:
2      dc.w    $180,$000      ; COLORE 0 = NERO
3      dc.w    $182,$f00      ; COLORE 1 = ROSSO
4      dc.w    $184,$0f0      ; COLORE 3 = VERDE
5      dc.w    $186,$00f      ; COLORE 4 = BLU
```

Vi ricordate come è il formato dei colori? RGB=RED, GREEN, BLU. Per avere un aiuto in ogni momento sul significato dei registri *\$dffXXX* fate *<=C 180>* oppure *<=C numero>* e avrete un breve riassunto (in inglese). Per esempio fate *<=c 006>* e vedrete il nome e la spiegazione del registro che avete usato per far lampeggiare il colore. Per vedere tutti i registri fate semplicemente un *<=C>*.

Il WAIT invece serve per aspettare una certa linea dello schermo, ad esempio se si vuol fare il colore di sfondo (*color0*) nero fino a metà, mentre nella metà inferiore si vuole blu, basta mettere un

```
1  dc.w    $180,0            ; colore 0 NERO
```

seguito da un WAIT che aspetta la metà dello schermo, dopodiché

```
1  dc.w    $180,$00f         ; colore 0 BLU
```

Con questo stratagemma si può cambiare l'intera palette (i colori) a qualsiasi linea del video, cosa che invece su PC in VGA nemmeno si sognano, infatti anche se i giochi Amiga solitamente hanno schermi di soli 32 colori, cambiando la tavolozza dei colori ogni tanto man mano che lo schermo scende si possono fare più tonalità di una VGA a 256 colori, specialmente se si considera che con un solo colore di sfondo si può fare una sfumatura cambiando il colore ad ogni linea, come faremo nel primo listato di questa lezione. Il comando WAIT si presenta in questa forma:

```
1  dc.w    $1007,$fffe      ; WAIT coordinata X= $10, Y= $07
```

Questo comando significa: ATTENDI LA LINEA ORIZZONTALE \$10, colonna 7 (cioè al settimo punto a partire da sinistra; i punti sono detti **pixel**). Il *\$fffe* significa WAIT, e va messo comunque

tutte le volte, mentre il primo byte è la linea orizzontale (x) da aspettare e il secondo è quella verticale (y).

Lo schermo infatti è fatto di molti punti disposti l'uno accanto all'altro, come un foglio a quadretti molto piccoli, ad esempio la carta millimetrata. Per indicare il punto (pixel) situato (come nella battaglia navale) alla posizione 16,7, ossia a 16 punti dal bordo superiore del foglio verso il basso e 7 dal bordo sinistro verso destra, indicherò \$1007. (\$10=16!). Di solito basta indicare la linea orizzontale al suo inizio, (la posizione è \$07 anziché \$01 perché tanto quest'ultima è fuori del monitor all'estrema sinistra). L'istruzione WAIT è usata anche per terminare la COPPERLIST: infatti per indicare la fine della COP va messo un

```
1 dc.w $FFFF,$FFFE ; Fine Copperlist
```

Che per convenzione il Copper considera la fine, anche perché indica di attendere una linea che non esiste! (la copperlist poi riparte da capo). Si è sparsa la voce tempo fa che sarebbero necessarie due istruzioni di fine copperlist anziché una sola per alcuni vecchi modelli di Amiga, ma sembra sia una psicosi di massa, dato che nessuno ne ha mai messe due e ha sempre funzionato tutto.

Un ultima cosa: per fare la nostra copperlist che per ora è priva di disegni, ha solo sfumature, bisogna disabilitare i BITPLANE, ovvero i PIANI di BIT che sovrapponendosi danno luogo alle figure. per fare questo basta aggiungere la linea DC.W \$100,\$200, ossia mettiamo il valore \$200 nel \$dff100, che è il registro di controllo dei bitplane.

Ora siamo in grado di fare completamente la copperlist che aspetta la metà del video e cambia il colore!

```
1 COPPERLIST:
2 dc.w $100,$200 ; BPLCON0 Nessuna figura, solo lo sfondo
3 dc.w $180,0 ; Color 0 NERO
4 dc.w $7f07,$FFFE ; WAIT - Aspetta la linea $7f (127)
5 dc.w $180,$00F ; Color 0 BLU
6 dc.w $FFFF,$FFFE ; FINE DELLA COPPERLIST
```

Considerando che per verificare il funzionamento delle vostre copperlist dovrete fare delle sfumature di colore, ecco una *tabella di riferimento per la scelta dei colori del copper*.

L'Amiga ha 32 registri colore per 32 colori diversi:

```
1 $dff180 ; color0 (sfondo)
2 $dff182 ; color1
3 $dff184 ; color2
4 $dff186 ; color3
5 ...
6 $dff1be ; color31
```

In ognuno di questi 32 registri colore si può selezionare uno dei 4096 colori visualizzabili, “mischiando” i 3 colori fondamentali ROSSO, VERDE, BLU. Ognuno di questi 3 colori può avere una intensità da 0 a 15, ossia 16 toni. Infatti il massimo numero di combinazioni è $16 \times 16 \times 16 = 4096$, ossia 16 ROSSI moltiplicato 16 VERDI moltiplicato 16 BLU. Il valore del colore si può mettere col processore o col COPPER:

```
1 move.w #$000,$dff180 ; colore NERO in color0
2
3 dc.w $180,$FFF ; colore BIANCO in color0
```

In questo esempio abbiamo visto i due estremi: \$FFF, ossia BIANCO, e \$000, ossia NERO. Per scegliere il colore infatti occorre tener presente che la WORD del colore è composta così:

```
1 dc.w $0RGB
2
3 dove il quarto zero è inutilizzato, mentre:
4
```

5	R	=	componente ROSSA (RED)
6	G	=	componente VERDE (GREEN)
7	B	=	componente BLU (BLU)

Infatti i bit dal 15 al 12 non sono utilizzati, i bit dall'11 all'8 sono il ROSSO, quelli dal 7 al 4 sono il VERDE, quelli dal 3 allo 0 sono il BLU.

Ogni colore RGB come già detto può avere un valore da 0 a 15, ossia da 0 a \$F in esadecimale, dunque è facile scegliere il colore:

\$FFF	=	Bianco
\$D00	=	Rosso mattone
\$F00	=	Rosso
\$F80	=	Rosso-Arancio
\$F90	=	Arancione
\$fb0	=	Giallo-oro
\$fd0	=	Giallo-Cadmio
\$FF0	=	Limone
\$8e0	=	Verde chiaro
\$0f0	=	Verde
\$2c0	=	Verde scuro
\$0b1	=	Verde albero
\$0db	=	Acqua
\$1fb	=	Acqua chiaro
\$6fe	=	Blu cielo
\$6ce	=	Blu chiaro
\$00f	=	Blu
\$61f	=	Blu brillante
\$06d	=	Blu scuro
\$c1f	=	Violetto
\$fac	=	Rosa
\$db9	=	Beige
\$c80	=	Marrone
\$a87	=	Marrone scuro
\$999	=	Grigio medio
\$000	=	nero

Ora il problema è solo come costringere il copper ad eseguire ordini dalla nostra COPPERLIST sviando la sua attenzione da quella del WorkBench; ma c'è anche un altro problema: se facciamo eseguire la nostra, come facciamo dopo essere usciti a fargli rileggere quella di sistema??? Risposta: Bisogna segnarsi su un foglietto dove era!!! Ovvero: lo segniamo in un apposita longword denominata OLD COP, ovvero VECCHIA COPPERLIST, quella di sistema. Ma a chi lo dobbiamo chiedere dove è la copperlist di sistema? al sistema operativo ovviamente!! Per chiederglielo dovremo eseguire delle routines che sono nel CHIP del kickstart!!! Per fare questo bisogna sempre prendere come riferimento l'indirizzo che si trova nell'indirizzo \$4, che viene scritto dal kickstart e contiene appunto l'indirizzo da cui si possono fare le distanze di indirizzamento prefissate, di cui parleremo in seguito. per raccogliere la long all'indirizzo \$4 basta fare un:

```
1 MOVE.L $4,a6 ; In a6 ora abbiamo l'ExecBase
```

O meglio

```
1 MOVE.L 4.w,a6 ; Infatti 4 è un numero piccolo e si può scrivere
2 ; 4.w, il che risparmia spazio. (scrive l'istruzione
3 ; con $0004 invece di scriverla con $00000004, in cui
4 ; i primi zeri non servono. VIENE COMUNQUE SPOSTATA
5 ; UNA LONGWORD! la long contenuta nei 4 bytes 4,5,6,7.
```

Messo l'indirizzo che era contenuto in \$4 in a6, possiamo eseguire le routines del kickstart facendo dei JSR con la distanza di indirizzamento giusta, infatti esistono delle distanze di indirizzamento precise che corrispondono a certe routines già pronte nel kickstart. Ora sappiamo

che se facciamo, ad esempio, un JSR `-$78(a6)` disabilitiamo il multitasking!!! Ovvero viene eseguito solo il nostro programma! facciamolo subito! Caricate LEZIONE3a.s in un buffer Fx ed eseguitelo.

Però la Exec non si occupa di tutto: il kickstart, lungo 256k se è la versione 1.2 o 1.3, oppure lungo 512k se è 2.0 o 3.0, è diviso in library, ossia delle “raccolte” di routine già pronte che possono essere chiamate, e siccome ogni kickstart è diverso proprio fisicamente, nel senso che ad esempio la routine della Exec che disabilita il sistema operativo nel kick 1.3 potrebbe essere a `$fc1000`, mentre nell’1.2 o nel 2.0 a diversi indirizzi ancora, i cari progettisti hanno avuto una delle loro clamorose idee: *“perché non mettiamo un indirizzo alla locazione \$4 da cui si possa sempre eseguire la stessa routine facendo un JSR allo stesso offset (ovvero distanza di indirizzamento)?”* (P.S. JSR è come BSR, solo che può eseguire routines in qualsiasi parte della memoria, mentre il bsr le può eseguire se sono entro 32768 bytes in avanti o indietro).

Ed è questo quello che hanno fatto! Per eseguire ad esempio il Disable, che disabilita il sistema operativo, su tutti i kickstart basta fare:

```

1      move.l 4.w,a6      ; Indirizzo della Exec in a6
2      jsr   -$78(a6)     ; Disable – blocca multitask
3      bsr.w mioprogramma
4      jsr   -$7e(a6)     ; Enable – riabilita multitask

```

In ogni kickstart la routine sarà ad un indirizzo diverso, ma facendo in questo modo siamo sempre sicuri di eseguire quella routine. Basta sapere tutte le distanze di indirizzamento delle varie routines del sistema operativo per eseguirle, ma a noi interessa soltanto di salvare l’indirizzo della copperlist di sistema, e per farlo dobbiamo rivolgerci ad una parte delle routines del kick che si chiama: `graphics.library`, ossia quella che si occupa della GRAFICA, sia chiaro solo a livello di sistema operativo, non a livello hardware. Per accedere alla libreria grafica va APERTA, ossia dobbiamo fare così:

```

1      move.l 4.w,a6      ; Execbase in a6
2      lea   GfxName,a1   ; Indirizzo del nome della lib da aprire in a1
3      jsr   -$198(a6)    ; OpenLibrary, routine della EXEC che apre
4                        ; le librerie, e da in uscita l'indirizzo
5                        ; di base di quella libreria da cui fare le
6                        ; distanze di indirizzamento (Offset)
7      move.l d0,GfxBase  ; salvo l'indirizzo base GFX in GfxBase
8      ....
9
10     GfxName:
11     dc.b   "graphics.library",0,0 ; NOTA: per mettere in memoria
12                        ; dei caratteri usare sempre il dc.b
13     GfxBase:
14     dc.l   0            ; e metterli tra "", oppure ''

```

In questo caso abbiamo usato la routine della Exec *OpenLibrary* che richiede che sia messo in A1 l’indirizzo del testo con il nome della libreria da aprire. Per esempio potevamo aprire altre librerie come `dos.library` per caricare dei file o simili, `intuition.library` per aprire finestre ecc. Una volta eseguita al ritorno da in d0 l’indirizzo della libreria in questione, per intenderci un indirizzo come *GfxBase* da cui fare dei JSR con degli offset a proposito della grafica. Oltre ai JSR, sappiamo anche che, per esempio, l’indirizzo della COPPERLIST di sistema attuale è situata a \$26 bytes dopo il *GfxBase*, quindi continueremo il nostro programma salvando quell’indirizzo in una label `OldCop`:

```

1      move.l 4.w,a6      ; Execbase in a6
2      lea   GfxName,a1   ; Indirizzo del nome della lib da aprire in a1
3      jsr   -$198(a6)    ; OpenLibrary, routine della EXEC che apre
4                        ; le librerie, e da in uscita l'indirizzo
5                        ; di base di quella libreria da cui fare le
6                        ; distanze di indirizzamento (Offset)
7      move.l d0,GfxBase  ; salvo l'indirizzo base GFX in GfxBase
8      move.l d0,a6
9      move.l $26(a6),OldCop ; salviamo l'indirizzo della copperlist

```

```

10      ....      ; di sistema
11
12 GfxName:
13     dc.b      "graphics.library",0,0 ; NOTA: per mettere in memoria
14                                     ; dei caratteri usare sempre il dc.b
15 GfxBase:
16     dc.l      0 ; e metterli tra "", oppure ''
17
18 OldCop:
19     dc.l      0

```

Ora possiamo puntare la nostra copperlist, possiamo mettere un MouseWait e dopo ristabilire la vecchia cop; per puntare intendo mettere l'indirizzo della nostra copperlist nel registro COP1LC, ossia \$dff080, che è il puntatore alla copperlist nel senso che il copper esegue la copperlist il cui indirizzo si trova in \$dff080: basterà dunque mettere l'indirizzo in \$dff080, poi per far partire la copperlist basta scrivere nel registro \$dff088 (COPJMP1) qualsiasi cosa, basta che ci si scriva o che si legga che fa partire la copperlist, è un registro detto **strobe**, come fosse un bottone che basta toccarlo (**non usate però CLR.W \$dff088, da dei problemi**).

Verrà così eseguita ripetutamente ogni fotogramma la nostra copperlist fino a che non ne sarà messa un'altra nel \$dff080 (COP1LC). Un problema è che il \$dff080 è a sola scrittura, infatti se provare a fare un <=c 080> noterete il W di WRITE. Per poter rimettere a posto la copperlist di sistema, quella che visualizza l'asmone o il workbench, non potendone leggere l'indirizzo dal \$dff080, dovremo chiedere al sistema operativo quale ci ha messo, e questo si può fare con delle routine del kickstart: una volta ottenuto l'indirizzo di quella copperlist lo salveremo in una LONGWORD del nostro programma, poi punteremo la nostra copperlist, e all'uscita del programma rimetteremo a posto quella vecchia.

```

1      move.l 4.w,a6 ; Execbase in a6
2      jsr -$78(a6) ; Disable - ferma il multitasking
3      lea GfxName,a1 ; Indirizzo del nome della lib da aprire in a1
4      jsr -$198(a6) ; OpenLibrary, routine della EXEC che apre
5                                     ; le librerie, e da in uscita l'indirizzo
6                                     ; di base di quella libreria da cui fare le
7                                     ; distanze di indirizzamento (Offset)
8      move.l d0,GfxBase ; salvo l'indirizzo base GFX in GfxBase
9      move.l d0,a6
10     move.l $26(a6),OldCop ; salviamo l'indirizzo della copperlist
11                                     ; di sistema
12     move.l #COPPERLIST,$dff080 ; COP1LC - Puntiamo la nostra COP
13     move.w d0,$dff088 ; COPJMP1 - Facciamo partire la COP
14 mouse:
15     btst #6,$bfe001
16     bne.s mouse
17
18     move.l OldCop(PC),$dff080 ; COP1LC - Puntiamo la cop di sistema
19     move.w d0,$dff088 ; COPJMP1 - facciamo partire la cop
20
21     move.l 4.w,a6
22     jsr -$7e(a6) ; Enable - riabilita il Multitasking
23     move.l gfxbase(PC),a1 ; Base della libreria da chiudere
24                                     ; (vanno aperte e chiuse le librerie!!!)
25     jsr -$19e(a6) ; Closelibrary - chiudo la graphics lib
26     rts
27
28 GfxName:
29     dc.b      "graphics.library",0,0 ; NOTA: per mettere in memoria
30                                     ; dei caratteri usare sempre il dc.b
31 GfxBase:
32     dc.l      0 ; e metterli tra "", oppure ''
33
34 OldCop:
35     dc.l      0
36
37 COPPERLIST:
38     dc.w      $100,$200 ; BPLCON0 - Nessuna figura, solo lo sfondo
39     dc.w      $180,0 ; Color 0 NERO
40     dc.w      $7f07,$FFFE ; WAIT - Aspetta la linea $7f (127)

```



```

41      dc.w    $180,$00F      ; Color 0 BLU
42      dc.w    $FFFF,$FFFE   ; FINE DELLA COPPERLIST

```

Troverete questo esempio con suggerimenti e modifiche in Lezione3b.s Caricatevelo nel buffer <F2> o un altro qualsiasi ed ammirate il primo programma del corso che “BATTE NEL METALLO” dei CHIP dell’Amiga.

Avete fatto i vostri esperimenti sulla copperlist? Bene, ora vediamo di fare qualche effetto in movimento. Per cominciare però devo informarvi che per fare un qualsiasi movimento bisogna sincronizzare le routines con il pennello elettronico che ridisegna lo schermo. Per chi non lo sapesse infatti lo schermo viene ridisegnato 50 volte al secondo, e i movimenti che ci appaiono fluidi, ad esempio quelli dei videogames meglio programmati, sono spostamenti che coincidono con il cinquantesimo di secondo. Abbiamo già usato il registro \$dff006, che come abbiamo visto cambia di valore continuamente, proprio perché c’è la posizione del pennello elettronico, il quale parte da zero, ossia dalla parte più alta dello schermo, e arriva in fondo 50 volte al secondo. Se facciamo una routine che fa dei movimenti sul video senza temporizzarla, andrà alla velocità effettiva del processore, dunque troppo veloce per vedere qualcosa. Per attendere una certa linea video basta leggere il primo byte del \$dff006, in cui troviamo la linea raggiunta, ossia la posizione verticale (uguale al WAIT del COPPER):

```

1 WaitLinea:
2      CMPI.B  #$f0,$dff006    ; VHPOSr — Siamo alla linea $f0? (240)
3      bne.s   WaitLinea      ; se no, ricontrolla
4      ...

```

questo ciclo aspetta la linea 240, dopodiché l’esecuzione continua con le istruzioni seguenti, come la routine del Mouse che aspetta la pressione del tasto, dopodiché continua l’esecuzione. Inseriamo anche il WaitMouse:

```

1 mouse:
2      cmpi.b  #$f0,$dff006    ; VHPOSr — Siamo alla linea 240?
3      bne.s   mouse          ; Se non ancora, non andare avanti
4
5      bsr.s   RoutineTemporizzata ; Questa routine viene eseguita 1
6                                   ; volta sola per ogni fotogramma
7
8      bsr.s   MuoviCopper      ; Il primo movimento sullo schermo!!!!
9      btst   #6,$bfe001       ; tasto sinistro del mouse premuto?
10     bne.s   mouse           ; se no, torna a mouse:
11     rts

```

A questo punto abbiamo una routine che esegue una routine 1 sola volta per ogni FRAME video, ossia per ogni fotogramma, ossia 1 volta ogni cinquantesimo di secondo, e più esattamente viene eseguita non appena siamo arrivati alla linea 240, dopodiché, una volta eseguita, non sarà eseguita nuovamente fino a che non saremo nuovamente alla linea 240, il fotogramma successivo.

NOTA: L’immagine viene disegnata con la tecnica **raster** tramite un pennello elettronico, che parte a disegnare dalla prima linea in alto a sinistra, prosegue verso destra fino alla fine della linea, poi riparte dall’estrema sinistra della linea 2, per andare verso destra ecc, analogamente al percorso che facciamo noi per leggere: ogni linea da sinistra verso destra, partendo dalla prima in alto fino all’ultima in basso, dopodiché il pennello elettronico riparte dalla prima linea, primo punto a sinistra, come se noi avendo finito di leggere una pagina di libro ricominciassimo a leggerla anziché leggere la pagina seguente. D’altronde il monitor è uno solo e deve scrivere su quello solamente, il pennello non scrive sul muro.

Caricatevi l’esempio LEZIONE3c.s in un altro buffer di testo e provatelo. Questo esempio fa muovere in basso una WAIT e quindi il colore seguente quando premete il tasto destro del mouse. Tasto sinistro per uscire.

Avete compreso Lezione3c.s? Allora complichiamo leggermente le cose! Caricate la Lezione3c2.s in un buffer e studiatelo, ho aggiunto un controllo della linea raggiunta per fermare lo scroll.

Tutto chiaro in Lezione3c2.s?? Bene, continuiamo con la pratica caricando la Lezione3c3.s, in cui viene spostata una barretta sfumata fatta con 10 wait anziché una sola linea WAIT. Sempre più difficile!!!

Siete sempre vivi dopo la Lezione3c3.s? Massacratevi il cervello con la lezione seguente, la Lezione3c4.s, in cui passiamo da 10 label BARRA ad una sola label eseguendo delle distanze di indirizzamento.

Beh, non era poi così difficile. Il difficile viene ora con Lezione3d.s, in cui la barra va su e giù, e cambieremo anche la velocità della barra.

Avete capito Lezione3d.s? Sì? Non ci credo! Vi sembra di aver capito, non può essere... io lo rivedrei un attimo prima di proseguire... lo avere rivisto? Beh... allora caricatevi una variazione sul tema, Lezione3d2.s

Ora siete pronti per affrontare la Lezione3e.s, in cui è spiegato come fare una RASTERBAR ossia un effetto di scorrimento ciclico dei colori.

Un altro caso particolare: Come si fa a raggiungere la zona PAL (dopo \$FF) con i wait del copper in Lezione3f.s.

Per completare la lezione3.txt, caricatevi la Lezione3g.s, e Lezione3h.s, concernenti uno scorrimento a destra e sinistra anziché in basso ed in alto, dopodiché sarete pronti per la LEZIONE4.TXT, in cui sarà trattata la gestione delle immagini colorate e dei possibili effetti su di esse!

NOTA: Gli Esempi4x.s della LEZIONE4.TXT si trovano nella directory SORGENTI2, dunque dovete fare un `<V DF0:SORGENTI2>` per rendere possibile il caricamento delle immagini da quella directory. Dopodiché caricate la LEZIONE4.TXT in questo o in un altro buffer di testo. (con `<r>`)

Complimenti per essere arrivati qua! Il grosso è fatto! Ora andando avanti capirete con facilità, essendo entrati nella logica della programmazione ASM!.

LEZIONE 4 - IMMAGINI SULLO SCHERMO

In questa lezione impareremo a visualizzare delle figure in varie risoluzioni tramite la copperlist. Fino ad ora abbiamo potuto cambiare solo il `color0`, ossia il `$dff180`, con cui abbiamo fatto delle sfumature, ma chiaramente le figure non si fanno a forza di WAIT!!! Per visualizzare una normale figura IFF, creata col Deluxe Paint, digitalizzata, scannerizzata o renderizzata con un programma di ray tracing quali Image o Real 3d non serve alcun WAIT!!!! Basta dire al copper che risoluzione grafica ha la figura (numero di colori, risoluzione hires o lowres, interlacciata o meno) tramite il registro `BPLCON0`, ossia il `$dff100`, che per ora abbiamo sempre mantenuto col valore `$200`, che indica: *solo il colore di sfondo senza immagini in "sovraimpressione"*. È per questo che se in una copperlist del genere cambiamo ad esempio il `color1`, ossia `$dff182`, non accade nulla: perché non è abilitato nessun *bitplane*, ossia "piano di bit", ma solo lo "sfondo", il cui colore può essere cambiato col `$dff180`. Dopo aver definito il numero di colori e la risoluzione grafica (ad esempio 320x200 pixel, dove per PIXEL si intende ognuno dei piccoli punti che compongono la figura), avendo indicato dove si trova la figura da visualizzare mettendo il suo indirizzo negli appositi *puntatori* (registri come il `$dff080` (`COP1LC`) dove invece viene messo l'indirizzo dei *bitplane*), bisogna definire i colori della figura, ossia la *palette* (cioè la "tavolozza" dei colori definita dal programma di disegno (es. DeLuxe Paint) per la figura in questione), oppure la figura apparirà con i colori sbagliati. In pratica dobbiamo mettere in copperlist i registri colore necessari, se la figura è a 4 colori dobbiamo definire i 4 colori:

1	<code>dc.w</code>	<code>\$180,\$xxx</code>	<code>; color 0</code>
2	<code>dc.w</code>	<code>\$182,\$xxx</code>	<code>; color 1</code>
3	<code>dc.w</code>	<code>\$184,\$xxx</code>	<code>; color 2</code>
4	<code>dc.w</code>	<code>\$186,\$xxx</code>	<code>; color 3</code>

Questo pezzo di copperlist comunque lo salva direttamente il KEFCON

Ci sono anche altri *registri* che regolano la dimensione della figura per farla di dimensioni "speciali", come l'*overscan* che la rende più grande, o si può fare una "finestrella" che occupi solo una parte dello schermo. Altri registri speciali sono i *moduli*, usati spesso per gli effetti di allungamento delle figure. Nei primi esempi comunque manterremo i registri speciali azzerati o comunque con i valori normali per visualizzare una figura. Innanzitutto deve essere chiaro che c'è differenza tra un file IFF, ossia una figura nel formato standard caricabile dal DeLuxe Paint, e la figura REALE (detta RAW o BITMAP) che sta in memoria e viene visualizzata dal copper. Nel

disco è presente un programmino capace appunto di convertire figure in formato IFF nel formato RAW indispensabile per visualizzare le figure direttamente con il COPPER. Le figure in realtà sono composte di molti 0 ed 1, come tutti i dati BINARI in memoria. Abbiamo già visto che ogni dato in memoria è composto di BIT, ossia di zeri e di 1, che sono gli unici 2 stati possibili della memoria (essendo possibile solo *passaggio di corrente* e *assenza di corrente*); per comodità usiamo il sistema decimale e esadecimale, ma la realtà è sempre quella dei BIT. Allora come è possibile visualizzare una figura a 32 colori quando i bit possono essere solo 0 ed 1??? Se mettessimo in memoria una specie di foglio di carta a quadretti con dei quadretti anneriti (ossia 1) ed altri bianchi (ossia zero) potremmo solamente lavorare con 2 colori, il nero e il bianco, come i vecchi computer con i monitor a fosfori verdi che potevano visualizzare solo il colore di sfondo (i bit a zero) e certi disegni o parole fatte di bit accesi (ad 1). Con il COPPER si può anche operare in questo modo, a 2 colori, basta *accendere un solo bitplane*. In questo caso dovremo mettere in memoria la figura RAW, composta similmente al foglio di carta millimetrata descritto prima, con dei punti “accesi” e dei punti “spenti”. Fino qui tutto dovrebbe essere chiaro: è come fare la battaglia navale! Una nave sarebbe per esempio fatta di un certo numero di PIXEL (punti), allo stesso modo si può fare qualsiasi cosa:

UN UOMO:		UN AEREO (ho risparmiato gli zeri!)
		11
000011100000	000001100000	1111
000001000000	000010010000	1111
000111111000	000010010000	11111111111111111111111111111111
000101101000	000111111000	11111111111111111111111111111111
000101101000	000100001000	1111
000011110000	000100001000	1111
000010010000		111111
000010010000	UNA "A"	11111111
000010010000		
000110011000		

Quando la figura è grande conviene chiaramente farsela col programma di disegno o scannerizzarla, poi convertirla in RAW (0000110101...) col programma in questo disco (KEFCON). Per definire il colore di sfondo (gli zeri) basterà mettere un dc.w \$180,\$000 (nero), per definire il colore degli 1 basterà mettere un dc.w \$182,\$0f0 (verde). Per le figure a più colori il trucco è questo: i vari *bitplanes*, ossia *piani di bit* (0001010 ecc.) vengono “SOVRAPPOSTI” in una specie di *trasparenza*, per cui dove vengono sovrapposti due “1” appare un certo colore, dove sono sovrapposti tre “1” appare un altro colore eccetera. Tutto questo non va calcolato!!! Basta caricarsi la figura con l’iffconverter e salvarla in RAW, poi mettere il numero dei colori e la risoluzione nel \$dff100 (bp1con0), dire al copper dove abbiamo messo la figura RAW, mettere i colori giusti (che salva direttamente l’iffconverter separatamente), e la figura appare senza problemi. L’importante è aver chiaro il procedimento, in pratica poi per convertire l’immagine IFF in RAW e modificare il sorgente ci vogliono due minuti.

Innanzitutto chiariamo cosa fa l’iffconverter (nel nostro caso usiamo quello chiamato KEFCON, che potete caricare spostando la finestra dell’asmon e scrivendo nel MENU del DOS il suo nome; esistono iffconverters più recenti e con più opzioni, alcune delle quali spesso inutili, ma per ragioni di spazio e di compatibilità con kickstart 1.3 ho deciso di mettere questo nel corso, poi c’è anche il fatto che è programmato usando i registri hardware anziché il sistema operativo, dunque è in linea col corso. Se volete usare altri iffconverters fate pure, ma prima imparate ad usare questo, che è stato usato per fare giochi e demo gloriosi). Una immagine nella realtà abbiamo visto che è un insieme di piani di bit, più piani per più colori, un piano solo per 2 colori. Abbiamo anche visto che per essere visualizzata servono i colori giusti (la palette) e la risoluzione giusta nel \$dff100 (bp1con0); i programmatori dell’Amiga decisero di creare un formato

particolare per memorizzare le figure e trasferirle da un programma all'altro: questo formato per l'Amiga è l'IFF ILBM, ed è in pratica composto da i bitplanes compattati con una certa routine per prendere meno spazio, con attaccata la palette e la risoluzione; un programma quando carica un'immagine IFF scompatta i PIANI compattati, mette la palette a posto nei registri dei colori (\$dff180, \$dff182, \$dff184 eccetera) e la risoluzione nel \$dff100, il BPLCON0 (Sommariamente). Allo stesso modo quando ha una immagine in memoria, per salvarla in IFF deve comprimere i BITPLANES nel formato IFF, attaccarci la palette e il resto. L'Iffconverter fa queste operazioni: può caricare un RAW, e salvarlo in IFF, a patto che gli si dia la PALETTE e la RISOLUZIONE giusta, oppure può caricare un'immagine IFF e salvarla in RAW, e può salvare la PALETTE già nel formato dc.b \$180,xxx,\$182,xxx, ossia la PALETTE già da inserire nella copperlist. In altri computer si utilizzano diversi formati per le figure, ad esempio il GIF, il PCX e il TIFF sono usati dai PC MSDOS; oltre ad essere diversamente COMPRESSE le figure RAW+PALETTE, in questi computer è diverso anche il sistema di visualizzazione, infatti hanno il sistema **chunky** anziché bitplane, un sistema utile quando si debbono gestire 256 colori, ma meno capace di quello Amiga per quanto riguarda gli *scroll* (scorrimenti) e senza possibilità di cambiare la PALETTE come fa il COPPER con i WAIT. Le possibili risoluzioni grafiche dell'Amiga normale (non AGA) sono queste:

- 320x256 PIXEL detta LOW RES (bassa risoluzione)
- 640x256 PIXEL detta HIGH RES (alta risoluzione)

L'immagine può essere anche più lunga (312 linee in overscan) oppure lunga il doppio (tramite l'interlace che però causa uno sfarfallamento). Anche la larghezza può essere leggermente aumentata con l'*overscan*.

Le immagini in LOW RES (larghe 320 pixel) possono avere fino ad un massimo di 32 colori, ci sono poi 2 modi speciali detti *EHB* (Extra Half Bright) e *HAM* (Hold And Modify) che visualizzano rispettivamente 64 e 4096 colori, ma hanno delle limitazioni particolari che vedremo. Le immagini in *high res* (alta risoluzione) possono avere un massimo di 16 colori e non dispongono di modi speciali. I giochi sono quasi tutti in *low res*, per sfruttare il maggior numero di colori disponibili, per il risparmio di memoria (che purtroppo FINISCE!), e per la maggiore velocità raggiungibile (infatti l'HIGH RES rallenta le operazioni maggiormente del LOW RES, inoltre bisognerebbe spostare pezzi più grandi di memoria essendo l'immagine larga il doppio!).

Analizziamo la tecnica di visualizzazione dei colori: abbiamo detto che il massimo dei colori in lowres è 32 (senza contare i modi speciali); è possibile scegliere una risoluzione video con 2,4,8,16 o 32 colori. Questo perché sono determinati dalla sovrapposizione di BIT, dunque ogni bitplane che "sovrapponiamo" aggiungiamo 2 bit ad ogni "pixel" che diventa *più profondo* di 2 bit: ora con 2 bit si può ottenere soltanto 0 ed 1, ossia 2 colori, allora la risoluzione 320x200 a 2 colori avrà un solo BITPLANE, come abbiamo già detto. Aggiungendo un altro bitplane i colori possibili diventano 4, infatti si possono verificare 4 situazioni di sovrapposizione per ogni PIXEL: 00, 01, 10, 11 ossia "entrambi i bit a zero", "primo bitplane a zero e secondo ad 1", "primo bitplane ad 1 e secondo a zero", "entrambi i bitplane ad 1". Aggiungendo un altro bitplane si possono verificare 8 situazioni diverse (corrispondenti a 8 colori diversi):

000,001,010,011,100,101,110,111 (3 bitplanes=3 bit per PIXEL=8 possibilità)

Aggiungendo un bitplane, il quarto, arriviamo ad 4 bit possibili per PIXEL, ossia 16 diverse possibilità per 16 colori:

0000,0001,0010,0011,0100,0101,0110,0111,1000,1001,1010,1011,1100,1101,1110,1111

Lo stesso vale per il quinto bitplane, che porta a 5 il numero di bit per pixel, ossia 32 colori possibili. Dunque ogni bitplane eleva alla seconda il numero dei colori:

- 0 bitplane = solo il colore di sfondo COLOR00 (\$dff180), 1 COLORE
- 1 bitplane = 2 COLORI
- 2 bitplane = 4 COLORI (2*2, ossia 2 alla seconda)
- 3 bitplane = 8 COLORI (2*2*2, ossia 2 alla terza)
- 4 bitplane = 16 COLORI (2*2*2*2, ossia 2 alla quarta)
- 5 bitplane = 32 COLORI (2*2*2*2*2, ossia 2 alla quinta)

L'Amiga ha 32 registri per i 32 colori possibili in LOWRES, che partono dal COLOR0 per arrivare fino al COLOR31 (la numerazione parte contando lo zero come per i bit). Il color0 è il \$dff180, seguito dagli altri:

- \$dff182 = COLOR1
- \$dff184 = COLOR2
- \$dff186 = COLOR3
- \$dff188 = COLOR4
- \$dff18a = COLOR5
- eccetera...

per esempio se un pixel di una immagine lowres a 16 colori è del colore dello "SFONDO", ossia del COLOR0, modificabile agendo sul \$dff180, vuol dire che tutti e 4 i bitplanes sono a zero: 0000, mentre un pixel che ha il colore 32, detto COLOR31, sarà il risultato di questa combinazione binaria: 1111. Gli altri colori sono il frutto delle altre combinazioni. L'amiga 1200 dispone di 8 bitplanes al massimo con il suo *chipset* AGA, infatti può generare 256 colori (2 all'ottava=256), infatti nei programmi grafici AGA si possono scegliere anche risoluzioni con 64 colori (6 planes), 128 colori (7 planes). Una schermata video è detta anche **playfield**.

Vediamo di calcolare quanta memoria occupa una figura a 2 colori in 320*256: ogni linea ha 320 pixel, essendo un byte composto di otto bit, in una linea ci sono 40 byte (infatti 8*40=320). Allora basterà moltiplicare 40, ossia il numero di byte per linea, per il numero delle linee, cioè 256: 40*256=10240. Dunque un bitplane in lowres occupa 10240 bytes. Allora possiamo calcolare anche una figura con 4 colori, ossia 2 bitplanes: 40*256*2=20480. Dunque per una figura in LOWRES standard basta moltiplicare 40*256*bitplanes. Stabilito che in LOWRES ci sono 320 bit per linea, ossia 40 bytes, in HIRES essendo largo il doppio ci saranno 80 bytes per linea: 80*256*bitplanes. In definitiva, la formula generale per calcolare la grandezza è:

Byte per linea * linee del playfield * numero bitplane

Analizziamo ora il BPLCON0, il registro dove va indicata la risoluzione e il numero dei colori: (Potete leggere un riassunto scrivendo `<=C 100>`)

\$dff100 - BPLCON0

Bit Plane Control Register 0 (1 word, ossia 16 bit, dallo 0 al 15)

NUMERO DEL BIT (nota: bit ad 1 = ON, bit a 0 = OFF)

15	-	HIRES	Modo hires (1=640x256 , 0=320x256)
14	-	BPU2	\
13	-	BPU1) 3 bit per scegliere il numero di bitplanes
12	-	BPU0	/
11	-	HOMOD	Hold And Modify mode (HAM 4096 colori)
10	-	DBLPF	Double playfield
09	-	COLOR	video composito (per il GENLOCK)
08	-	GAUD	Genlock audio
07	-	X	
06	-	X	
05	-	X	
04	-	X	
03	-	LPEN	Lightpen (Penna ottica)
02	-	LACE	Interlacciato (320x512 o 640x512)
01	-	ERSY	External resync (Per il GENLOCK)
00	-	X	

Questo registro è *bitmapped*, ossia ogni suo bit ha una funzione:

- Il bit 15 abilita il modo hires: questo modo grafico visualizza 640 pixel per linea orizzontale invece di 320. Ricordatevi di mettere DDFSTART/STOP a \$003c e \$00d4 anziché \$0038 e \$00d0, altrimenti non verranno visualizzate le prime linee del bordo sinistro!
- I bit 14-12 servono a stabilire il **numero** di plane da accendere, **non** quali plane; infatti i bit sono 3 ed i plane possibili sono 6. Qui bisogna scrivere **quanti** plane da accendere, proprio come un numero, **non** selezionando **quali**. Es.: '3', '0', '6'. In 3 bit, infatti, sono esprimibili 8 numeri, dallo 0 al 7. Ripeto: **lavorate con un numero vero e proprio in binario a 3 bit, non con singoli bit da accendere o spegnere, a differenza degli altri bit!** N.B.: Scrivendo '0' (=000) si spengono tutti i plane, scrivendo %101 se ne accendono 5; con 6 plane si attiva l'*Half-Bright* mode a 64 colori.
- Il bit 11 serve ad azionare il modo *[HAM]* (vanno accesi 6 plane) L'HAM consente agli amiga normali di visualizzare 4096 colori, l'HAM8 permette agli Amiga 1200/4000 di visualizzarne 262144.
- Il bit 10 consente di attivare il modo *Dual PlayField*, uno speciale modo a 2, 4 o 6 plane che permette di creare due schermi da 1, 2 o 3 plane ciascuna, chiamati PlayField1 e PlayField2, accavallati l'uno sull'altro in trasparenza reneendo trasparente il colore 0 del PlayField sovrastante. È possibile creare, quindi, un effetto parallattico simile a quello presente in molti giochi. Per esempio, si potrebbe utilizzare un PlayField a 3 plane (8 colori) per l'area di gioco ed un altro PlayField di sfondo, che magari scrolla (scorre) più lentamente per dare un maggiore senso di profondità, raffigurante pianure e montagne. Appena settato il bit i plane dispari (1, 3, 5) formano il *Playfield1*, mentre quelli pari (2, 4, 6) il secondo: l'hardware, appena attivato il bit DPF, raggruppa così i plane per poterli rendere indipendenti, dato che, come vedremo, esistono registri di scroll ed altri che distinguono parametri per plane pari e dispari, usati anche per controllare indipendentemente due interi playfield in dual mode! N.B.: Il modo DualPlayField consente di sovrapporre solo 2 schermi e, comunque, di analoga risoluzione grafica (es.: Hires+Hires, Lowres+Lowres, ecc...).
- Il bit 9 serve ad attivare anche l'uscita videocomposita degli Amiga posta accanto a quella RGB del monitor. Personalmente, la attivo sempre per consentire di vedere qualcosa sul monitor durante i miei prodotti anche a chi non possiede un monitor RGB standard. **Settatelo sempre a 1.**

- Il bit 8 attiva l'audio di un eventuale genlock collegato all'Amiga: non serve praticamente a niente, sorvoliamo.
- Il bit 7 è usato solo dai chipset evoluti dell'A1200, su amiga normale non ha funzioni. Ricordatevi comunque di lasciare sempre a zero questi bit inutilizzati, altrimenti su a1200 rischiate di compromettere il funzionamento del vostro demo/gioco/programma.
- Il bit 6 non ha funzioni su amiga normale, lasciatelo a zero.
- Il bit 5 lasciatelo a zero
- Il bit 4 lasciatelo a zero
- Il bit 3 serve a far ricevere le coordinate della penna ottica nei registri VHPOS (\$dff006) e VPOS (\$dff004) del pennello elettronico. La penna ottica su Amiga non è utilizzata quasi mai, non interessa questa opzione.
- Il bit 2 imposta il modo InterLace, che consente la visualizzazione di una videata con doppia risoluzione verticale, ma interlacciata. (512 linee) Vedremo in seguito come funziona questa modalità
- Il bit 1 serve a sincronizzare il movimento del pennello con la frequenza di qualche apparecchio esterno all'Amiga, dunque lasciatelo sempre a zero.
- Il bit 0 lasciatelo a zero

Detto questo facciamo degli esempi sull'uso del \$100 (BPLCON0) in copperlist:

```

1      ; 5432109876543210
2 dc.w $100,%0100001000000000 ; —> 4 plane in Lowres (320x256)
3 dc.w $100,%1011001000000100 ; —> 3 plane in Hires+Interlace (640x512)
4 dc.w $100,%0110001000000100 ; —> 6 plane in HALF-BRIGHT Lowres+Lace
5 dc.w $100,%0110101000000000 ; —> 6 plane in HAM lowres (4096 colori)
6 dc.w $100,%0110011000000000 ; —> DualPlayField 3+3 plane in Lowres
7 dc.w $100,%1100011000000100 ; —> DualPlayField 2+2 in Hires+interlace

```

Nella lezione 3 abbiamo usato il BPLCON0 nella copperlist dandogli valore \$200:

```

1      dc.w    $100,$200

```

Infatti abbiamo settato solo il bit 9, che serve per attivare il genlock:

```

1      ; 5432109876543210
2 dc.w $100,%0000001000000000

```

Il genlock è un apparecchio che serve per mettere in sovraimpressione titoli o grafica fatta con l'amiga su video televisivi, quindi chi non ha questo accessorio non vede cambiamenti tra una copperlist con il bit 9 attivato ed una con il bit disattivato, ma conviene tenerlo sempre ad 1, per chi volesse usare il genlock con le nostre copperlist, e perché il vecchio Amiga 1000 ha una uscita videocomposita per il monitor a colori. Dunque avremmo avuto lo stesso risultato in RGB anche con un dc.w \$100,0. Come vedete, i bitplane sono ZERO, dunque c'è solo il colore di sfondo senza figure in sovraimpressione. Per "accendere" dei bitplane basta mettere il numero di bitplanes che vogliamo, in binario, nei bit 12, 13, 14 del registro.

Per esempio, per fare uno schermo con 1 bitplane (2 colori): (320x256!)

```

1      ; 5432109876543210
2 dc.w $100,%0001001000000000 ; BPLCON0 — bit 12 acceso!! (1 = %001)

```

*

Per uno schermo a 2 bitplanes: (4 colori)


```

1      ; 5432109876543210
2  dc.w  $100,%0010001000000000 ; BPLCON0 — bit 13 acceso!! (2 = %010)

```

*

Per uno schermo a 3 bitplanes: (8 colori)

```

1      ; 5432109876543210
2  dc.w  $100,%0011001000000000 ; bits 13 e 12 accesi!! (3 = %011)

```

*

Per uno schermo a 4 bitplanes: (16 colori)

```

1      ; 5432109876543210
2  dc.w  $100,%0100001000000000 ; BPLCON0 — bit 14 acceso!! (4 = %100)

```

*

Per uno schermo a 5 bitplanes: (32 colori)

```

1      ; 5432109876543210
2  dc.w  $100,%0101001000000000 ; bits 14,12 accesi!! (5 = %101)

```

Per uno schermo a 6 bitplanes: (per modi speciali EHB e HAM 4096 colori)

```

1      ; 5432109876543210
2  dc.w  $100,%0110001000000000 ; bits 14,13 accesi!! (6 = %110)

```

(In questa modalità se non si mette ad 1 il bit dell'HAM (11) lo schermo è in Extra Half Bright, se il bit 11 invece è settato lo schermo è HAM.

Dunque basta mettere il numero di bitplanes richiesto nei 3 bit 12, 13, 14 del registro. Se si desidera uno schermo in hires, largo 640 pixel anziché 320, basterà mettere ad 1 il bit 15, il primo a sinistra, **ricordandosi** che il massimo numero di bitplanes in HIRES è 4 (16 colori), e che bisogna cambiare il DFFSTART (\$dff092) e il DFFSTOP (\$dff094) rispetto al LOWRES:

```

1  dc.w  $92,$003c ; DdfStart HIRES normale
2  dc.w  $94,$00d4 ; DdfStop HIRES normale

```

Lo stesso vale per l'interlacciato (lunghezza 512 linee anziché 256), basta mettere ad 1 il bit 2.

Una volta impostato correttamente il BPLCON0, bisogna dire dove sono i bitplane che abbiamo “attivato”. Per fare ciò basta mettere i loro indirizzi negli appositi registri, che sono:

```

$dff0e0 = BPLOPT (puntatore bitplane 1)
$dff0e4 = BPL1PT (puntatore bitplane 2)
$dff0e8 = BPL2PT (puntatore bitplane 3)
$dff0ec = BPL3PT (puntatore bitplane 4)
$dff0f0 = BPL4PT (puntatore bitplane 5)
$dff0f4 = BPL5PT (puntatore bitplane 6)

```

Anche qua si parte dallo zero, dunque si arriva al 5 per definire il sesto. Certe volte invece si può trovare indicato il \$dff0e0 con BPL1PT, e di conseguenza si arriva al 6 per definire il sesto. L'help dell'Asmone parte da BPLOPT, potete verificare digitando <=c 0e0>. Per visualizzare una figura dunque basta puntare una copperlist con il BPLCON0 giusto e i colori giusti, poi bisogna *puntare* anche i bitplanes, ad esempio così:

```

1  MOVE.L #BITPLANE0,$dff0e0 ; indirizzo di BITPLANE0 in BPLOPT
2  MOVE.L #BITPLANE1,$dff0e4 ; BPL1PT
3  MOVE.L #BITPLANE2,$dff0e8 ; BPL2PT
4  ...

```

E la figura apparirà magicamente. I bitplanes comunque sono puntati nella copperlist direttamente, in quanto i puntatori devono essere riscritti ogni fotogramma.

Non bisogna mai dimenticarsi di mettere nella copperlist i registri “speciali”, che per ora useremo azzerati o comunque col valore normale, altrimenti rimangono con valore della copperlist

del workbench e la visualizzazione può essere disturbata se questi registri non erano azzerati (per esempio il workbench del kickstart 1.3 ha i MODULI azzerati, mentre il kickstart 2.0 li ha con valori diversi: i giochi e le demo grafiche che vanno sul kick 1.3 e invece visualizzano male le figure sul kick 2.0 spesso sono state fatte con copperlist dove mancavano i registri dei moduli, il \$dff108 e \$dff10a, che su kick1.3 sono a zero, dunque funzionava al collaudo, ma su kick 2.0 sballano la visualizzazione. Per evitare questi problemi dunque in ogni copperlist è sempre bene definirsi tutti i registri, anche quelli che non usiamo; i registri in questione sono:

```
$dff08e - DIWSTRT, inizio finestra video - normalmente a $2c81
$dff090 - DIWSTOP, fine finestra video - normalmente a $2cc1
$dff092 - DDFSTRT, data fetch start - normalmente a $0038
$dff094 - DDFSTOP, data fetch stop - normalmente a $00d0
$dff102 - BPLCON1, Bitplane control 1 - normalmente a $0000
$dff104 - BPLCON2, Bitplane control 2 - normalmente a $0000
$dff108 - BPL1MOD, modulo bitplanes pari - normalmente a $0000
$dff10a - BPL2MOD, modulo bitplanes dispari - normalmente a $0000
```

Parleremo di questi registri quando li useremo per creare effetti speciali, per ora basta ricordarsi di mettere sempre all'inizio della copperlist questi registri con i valori standard:

```
1  COPPERLIST:
2      dc.w      $8e,$2c81      ; DiwStrt
3      dc.w      $90,$2cc1      ; DiwStop
4      dc.w      $92,$0038      ; DdfStart * NOTA: per HIREs 640x256 $003c
5      dc.w      $94,$00d0      ; DdfStop * NOTA: per HIREs 640x256 $00d4
6      dc.w      $102,0         ; BplCon1
7      dc.w      $104,0         ; BplCon2
8      dc.w      $108,0         ; Bpl1Mod
9      dc.w      $10a,0         ; Bpl2Mod
10
11     dc.w      $100,xxxx      ; Bplcon0 — Definiamo i colori e la risoluzione
12
13     ;      Qua possiamo mettere i colori della figura; questo pezzo di
14     ;      copperlist lo genera direttamente l'iffconverter KEFCON, la
15     ;      salva a parte con un nome a piacere, successivamente si può
16     ;      includere qua con l'opzione TAGLIA&INCOLLA dell'editor caricandolo
17     ;      in un altro buffer, ad esempio.
18
19     dc.w $0180,$0010,$0182,$0111,$0184,$0022,$0186,$0222
20     dc.w $0188,$0333,$018a,$0043,$018c,$0333,$018e,$0154
21     dc.w $0190,$0444,$0192,$0455,$0194,$0165,$0196,$0655
22     dc.w $0198,$0376,$019a,$0666,$019c,$0387,$019e,$0766
23     dc.w $01a0,$0777,$01a2,$0598,$01a4,$0498,$01a6,$0877
24     dc.w $01a8,$0888,$01aa,$05a9,$01ac,$0988,$01ae,$0999
25     dc.w $01b0,$06ba,$01b2,$0a9a,$01b4,$0baa,$01b6,$07cb
26     dc.w $01b8,$0bab,$01ba,$0cbc,$01bc,$0dcd,$01be,$0eef
27
28     ;      Come vedete qua sono definiti tutti i 32 registri colore dell'Amiga,
29     ;      infatti ho caricato col KEFCON una figura a 32 colori e questa è
30     ;      la sua PALETTE generata assieme alla figura in RAW.
31
32     ;      Qua si possono fare eventuali effetti video con i WAIT....
33
34     dc.w      $FFFF,$FFFE      ; Fine della copperlist
```

Questa copperlist è sufficiente per la visualizzazione di una figura. Procediamo dunque con il primo esempio di visualizzazione di un *playfield* con 3 bitplanes, ossia 8 colori. Nel primo esempio di questa lezione, visualizziamo una figura già convertita in formato RAW, ed è presente sul disco del corso: per “caricare” la figura in memoria esiste una direttiva dell'ASMONE, chiamata INCBIN, che permette appunto di caricare dal disco un certo file di dati e copiarlo nel punto del nostro programma dove sta l'incbin: se per esempio preparassimo una copperlist e la salvassimo come file, potremmo caricarla così:

```
1  COPPERLIST:
2      incbin    "copper1"
```

Il risultato è lo stesso che mettere tanti dc.w equivalenti alle word che stanno nel file copper1. Nel nostro caso carichiamo l'immagine sotto una label PIC:

```
1 PIC:
2     incbin "amiga.320*256*3"
```

La figura comunque non è in formato TESTO, ma proprio in byte che compongono i bitplane: provate a caricarla in un buffer di testo e noterete che non si tratta di un testo.

Come notate il nome della figura è dato secondo le caratteristiche della figura stessa; è bene scegliere nomi per le figure che rispecchino le sue caratteristiche per non rischiare di dimenticarsi di che dimensione e quanti bitplanes hanno le figure RAW che convertite. Dalla lunghezza di questa immagine raw però si può dedurre la risoluzione e il numero di bitplanes: è lunga 30720 bytes, ossia 40*256*3 (40 bytes per linea*256 linee, moltiplicato 3 bitplanes fa 30720). Basterà dunque dire al COPPER che la figura si trova alla label PIC, e il gioco è fatto.

Comunque per puntare i bitplanes senza rischi di errore bisogna mettere i puntatori nella copperlist. Infatti i puntatori dei bitplanes possono contenere una word ciascuno, ossia la metà di un indirizzo (infatti un indirizzo è lungo una longword! es \$00020000). Se usiamo il processore possiamo anche caricare 2 registri di una word con un solo move.l

```
1     MOVE.L #BITPLANE00,$dff0e0 ; BPLOPT
2     MOVE.L #BITPLANE01,$dff0e4 ; BPLIPT (2 word più avanti di $dff0e0)
```

Ma nella copperlist come noto ogni move può essere di una WORD solamente:

```
1     MOVE.W #$123,$dff180 diventa dc.w $180,$123
```

Nel caso dei puntatori ai bitplanes allora dovremmo “spezzare” ogni indirizzo lungo una longword in 2 words per poter fare così:

```
1     MOVE.W #BITPL,$dff0e0 ; BPLOPTH (H=word ALTA dell'indirizzo)
2     MOVE.W #ANE00,$dff0e2 ; BPLOPTL (L=word BASSA dell'indirizzo)
3     MOVE.W #BITPL,$dff0e4 ; BPL1PTH
4     MOVE.W #ANE01,$dff0e6 ; BPL1PTL
5
6     BPLxPTH      = BitPLane x PoinTer High word , puntatore word alta
7     BPLxPTL      = BitPLane x PoinTer Low word , puntatore word bassa
```

Abbiamo spezzato BITPLANE00 (lungo una longword) in 2 words BITPL e ANE00, e abbiamo ottenuto lo stesso risultato del MOVE.L con 2 MOVE.W, adatti alla copperlist, dove tradurremmo così:

```
1     dc.w $e0,BITPL ; BPLOPTH \primo bitplane
2     dc.w $e2,ANE00 ; BPLOPTL /
3
4     dc.w $e4,BITPL ; BPL1PTH \secondo bitplane
5     dc.w $e6,ANE01 ; BPL1PTL /
6
7     (infatti $dff0e0 si traduce in copperlist in $e0, ecc.)
```

Questa divisione si dice *divisione di una longword in word alta e word bassa*, dove la *word alta* è quella a sinistra, BITPL, quella *bassa* è quella destra, qua ANE00. Facciamo un esempio con indirizzi veri:

Il bitplane0 si trova a \$23400, il bitplane1 a \$25c00

```
1     dc.w $e0,$0002 \primo bitplane (word alta) \000023400
2     dc.w $e2,$3400 / (word bassa) /
3
4     dc.w $e4,$0002 \secondo bitplane (word alta) \000025c00
5     dc.w $e6,$5c00 / (word bassa) /
```

Vi starete già immaginando che per mettere gli indirizzi giusti nella copperlist sia necessario controllare a che indirizzo sta la figura e cambiare a mano le word in questione. Invece basta una piccola routine di una decina di istruzioni per risparmiarci il lavoro di “spezzare” gli indirizzi e

metterli nella copperlist al punto giusto. Questa routine si può usare per *puntare* qualsiasi figura di qualsiasi dimensione con il numero di bitplanes che vogliamo, basta cambiare i parametri!!! Il trucco sta nell'uso di una istruzione particolare del 68000, lo SWAP, che in inglese significa SCAMBIA, ed in effetti SCAMBIA le 2 words di una longwords facendo diventare BASSA quella ALTA e viceversa:

```

1      MOVE.L  #CANETOPO,d0      ; in d0 mettiamo la longword CANETOPO
2
3      SWAP    d0                  ; SCAMBIAMO LE WORDS: il risultato è che
4                                  ; in d0 abbiamo TOPOCANE!!!!

```

Questo comando *funziona solo sui registri DATI*. Allo stesso modo \$00023400 viene scambiato in \$34000002. Vediamo la routine:

```

1      MOVE.L  #PIC,d0            ; in d0 mettiamo l'indirizzo della PIC,
2                                  ; ossia dove inizia il primo bitplane
3
4      LEA     BPLPOINTERS,A1      ; in a1 mettiamo l'indirizzo dei
5                                  ; puntatori ai planes della COPPERLIST
6      MOVEQ   #2,D1               ; numero di bitplanes -1 (qua sono 3)
7                                  ; per eseguire il ciclo col DBRA
8  POINTBP:
9      move.w  d0,6(a1)            ; copia la word BASSA dell'indirizzo del plane
10                                  ; nella word giusta nella copperlist
11      swap    d0                  ; scambia le 2 word di d0 (es: 1234 > 3412)
12                                  ; mettendo la word ALTA al posto di quella
13                                  ; BASSA, permettendone la copia col move.w!!
14      move.w  d0,2(a1)            ; copia la word ALTA dell'indirizzo del plane
15                                  ; nella word giusta nella copperlist
16      swap    d0                  ; scambia le 2 word di d0 (es: 3412 > 1234)
17                                  ; rimettendo a posto l'indirizzo.
18      ADD.L   #40*256,d0          ; Aggiungiamo 10240 ad D0, facendolo puntare
19                                  ; al secondo bitplane (si trova dopo il primo)
20                                  ; (cioè aggiungiamo la lunghezza di un plane)
21                                  ; Nei cicli seguenti al primo faremo puntare
22                                  ; al terzo, al quarto bitplane eccetera.
23
24      addq.w  #8,a1               ; a1 ora contiene l'indirizzo dei prossimi
25                                  ; bplpointers nella copperlist da scrivere.
26      dbra    d1,POINTBP          ; Rifai D1 volte POINTBP (D1=num of bitplanes)

```

dove cambiamo questa parte di copperlist:

```

1  BPLPOINTERS:
2      dc.w  $e0,$0000,$e2,$0000      ;primo  bitplane (BPL0PT)
3      dc.w  $e4,$0000,$e6,$0000      ;secondo bitplane (BPL1PT)
4      dc.w  $e8,$0000,$ea,$0000      ;terzo  bitplane (BPL2PT)

```

La routine non fa altro che prendere l'indirizzo del bitplane, copiarne la word BASSA nella copperlist alla word dopo il \$e2, ossia il puntatore della word bassa dell'indirizzo (che si trova 6 bytes dopo BPLPOINTERS, infatti viene usato un move.w d0,6(a1), dove in a1 c'è l'indirizzo di BPLPOINTERS). Dopo aver sistemato la word bassa, con lo SWAP scambiamo word bassa con word alta, permettendo con un successivo move.w d0,2(a1) di copiare la word ALTA, anziché quella bassa, nella word dopo il \$e0, ossia il puntatore della word alta del primo bitplane, che si trova, appunto, 2 bytes dopo BPLPOINTERS. A questo punto abbiamo PUNTATO il primo bitplane: (es. a \$23400)

```

1  BPLPOINTERS:
2      dc.w  $00e0,$0002,$00e2,$3400      ; BPL0PT - primo  bitplane * PUNTATO *
3      dc.w  $00e4,$0000,$00e6,$0000      ; BPL1PT - secondo bitplane
4      dc.w  $00e8,$0000,$00ea,$0000      ; BPL2PT - terzo  bitplane
5
6      ^      ^
7      |      |
8      2(a1)   6(a1)      ; notare l'uso degli offset per
9                          ; inserire le word nel punto
10                         ; giusto.

```

NOTA: con il `move.w d0,x(a1)` copiamo la word bassa della long in d0 perché la copia avviene in questo modo:

```
1  move.w #CANETOPO,2(a1) ; in 2(a1) viene copiato TOPO
```

Successivamente rimettiamo a posto l'indirizzo con un altro SWAP, per poter passare al bitplane successivo con un `ADD.L #LUNGHEZZABITPLANE,d0`. Con un `addq.w #8,a1` passiamo a puntare il secondo bitplane, infatti se in a1 c'era l'indirizzo di BPLPOINTERS, aggiungendo 8 bytes (4 words) passiamo in questo punto:

```
1 BPLPOINTERS:
2   dc.w $00e0,$0002,$00e2,$3400 ; BPLOPT — primo bitplane * PUNTATO *
3 a1PUNTAQUA:
4   dc.w $00e4,$0000,$00e6,$0000 ; BPL1PT — secondo bitplane
5   dc.w $00e8,$0000,$00ea,$0000 ; BPL2PT — terzo bitplane
```

Ripetiamo quindi con un loop DBRA d1,label questa routine, in questo caso 3 volte, per puntare 3 bitplanes. (Come ricorderete al loop DBRA bisogna mettere nel registro che conta i cicli il numero richiesto meno 1, che è il primo che non viene contato: qua infatti in d1 viene messo il valore 2.

Questa routine ha la struttura classica delle routines che generano effetti col copper, dunque capirla esattamente è fondamentale. Una routine simile la avete già trovata in Lezione3h.s, in quel caso c'erano dei loop DBRA per cambiare 29 wait della copperlist.

Caricate Lezione4a.s per vedere in pratica l'esecuzione di questa routine PUNTABITPLANES con il DEBUG.

A questo punto mancano solo due “rifiniture” al nostro listato per evitare problemi nel visualizzare un'immagine: un paio di istruzioni per disabilitare il chipset AGA, che rende compatibile con l'Amiga 1200 e 4000 il nostro lavoro, e qualche altra linea nella copperlist per far sparire gli sprites, che altrimenti girovagherebbero a caso per il nostro disegno provocando disturbi intermittenti. Per disabilitare l'AGA bastano queste 2 linee:

```
1  move.w #0,$dff1fc ; FMODE — Disattiva l'AGA
2  move.w #$c00,$dff106 ; BPLCON3 — Disattiva l'AGA
```

E se volete proprio essere sicuri potete aggiungere questa: (palette sprite)

```
1  move.w #$11,$dff10c ; BPLCON4 — Resetta palette Sprite
```

Basta eseguirle dopo aver puntato la nuova copperlist, mentre per fermare gli sprites ubriachi basta puntare i loro puntatori a ZERO:

```
1  dc.w $120,$0000,$122,$0000,$124,$0000,$126,$0000,$128,$0000
2  dc.w $12a,$0000,$12c,$0000,$12e,$0000,$130,$0000,$132,$0000
3  dc.w $134,$0000,$136,$0000,$138,$0000,$13a,$0000,$13c,$0000
4  dc.w $13e,$0000
```

(Nota: I registri da \$dff120 a \$dff13e si chiamano SPR0PT, SPR1PT...SPR7PT.)

Parleremo degli sprites in seguito, per ora basta che li togliate di mezzo aggiungendo con un semplice `<COPY+INSERT>` (`<Amiga+b+c+i>`) questo pezzo di testo alla vostra copperlist. Gli sprites non apparivano con i bitplanes azzerati, mentre accendendo anche un solo bitplane gli sprite si fanno vivi.

Finalmente potete vedere in pratica come viene visualizzata una figura caricandovi Lezione4b.s in un buffer di testo Fx a piacere.

Avete provato ad aggiungere effetti copper alla figura??? Caricate l'esempio Lezione4c.s per una fusione di alcuni degli effetti già visti.

Avete capito ora l'importanza del WAIT in una copperlist con dei bitplane? Ci può far cambiare i colori (e non solo) anche ogni linea! Non ci resta che visualizzare un'immagine fatta da voi anziché quella nel corso. Per fare ciò dovete disporre di una figura in 320x256 a 8 colori, se non la avete fatevela anche sommariamente con un programma di disegno, oppure convertite

con ADPRO o altro programma una figura a più colori in quel formato. Disponendo di quella immagine in formato IFF (su un disco formattato) con il nome che preferite, immaginiamo che si chiami “FIGURA”, bisogna convertirla in formato RAW, ossia nel formato REALE dei bitplanes leggibili dal Copper, caricandola con l’iffconverter in questo disco, il “KEFCON”, che ha molte funzioni di cui però parleremo in seguito. Leggete queste istruzioni prima di caricarlo: Il converter è programmato in assembler hardware, dunque non supporta il multitasking e non si può “spostare” la sua finestra per leggere la lezione, dato che la sua finestra è una copperlist propria e non di sistema, che però è compatibile AGA e non crea problemi (I buoni vecchi programmatori!). Preparatevi prima la figura in un disco formattato, che poi inserirete dopo aver caricato il KEFCON nel df0: (disk drive interno) o nel df1: (disk drive esterno) se lo avete. Una volta caricato, vi apparirà un quadro comandi in alto, con delle opzioni, quelle che ci interessano sono: (vi faccio uno schema dei “PULSANTI”)

```

-----
| SAVE |      | IFF ILBM |
-----

-----
| LOAD |      | READ DIR |
-----

-----
| QUIT |
-----

-----
| QUA C'E UNA FINESTRELLA A STRISCIA | -Dove si deve scrivere il nome del file
-----
```

LOAD, SAVE e QUIT significano naturalmente CARICA, SALVA ed ESCI DAL PROGRAMMA. READ DIR serve a visualizzare nella finestra a destra la lista dei file che sono nel disco, ossia la sua directory. IFF ILBM è un pulsante che indica il tipo di file che si può salvare o caricare, in questo caso è giustamente ad IFF ILBM perchè dobbiamo caricare una figura iff; successivamente per salvarla in RAW basterà clickare su quel pulsante, il quale si cambierà in “RAW NORM”, e la figura sarà salvata come RAW; tra i possibili formati ci sono anche SPRITE e RAW BLIT, quelli li useremo in seguito. Per ora ci interessano solo “RAW NORM” e “COPPER” dove con COPPER si salva la PALETTE dei colori direttamente in un file di testo con i DC.W da inserire nella nostra copperlist!

Per fare la conversione, clickate sulla finestrella a striscia in basso a sinistra dove appare la scritta “allocated GFX-buffer” e si cambierà in df0: ossia il drive interno. Se avete inserito il disco con la figura iff in df0: lasciatelo, oppure indicate il nome del drive dove avete messo il disco (ad esempio df1:, oppure dh0: per l’hardisk). Per leggere la lista dei file premete READ DIR, e potrete selezionare la vostra figura clickando col mouse sopra il suo nome e premendo il pulsante “LOAD”. Vedrete a questo punto comparire la figura, che potete vedere per intero scorrendola in alto e basso con i tasti cursore. Notate che una volta caricata la figura appaiono le sue caratteristiche nella finestrella a striscia: “bitplane \$2800, total \$7800”. Infatti ogni singolo bitplane è lungo \$2800 (ossia 10240 decimale, 40*256), e in totale la figura RAW è lunga \$7800, ossia 30720 (40*256*3). Sopra sono indicate anche le altre caratteristiche:

```
WIDTH: 320 (larghezza), HEIGHT 256 (LUNGHEZZA), DEPTH 3 (NUMERO BITPLANES)
```

Ora clickate sul pulsante “IFF ILBM” facendolo cambiare in “RAW NORM”; per salvare la pic in RAW premete un’altra volta col tasto sinistro sulla finestrella a striscia e definite il nome, ad

esempio “df0:FIGURA.RAW” e premete il pulsante “SAVE”. Il RAW da includere con l’INCBIN è salvato! Ora manca solo la palette dei colori per la nostra COPPERLIST; per selezionare il modo ‘salvataggio palette clickate 5 volte sul gadget “RAW NORM”, ossia fino a che non si cambia in “COPPER”. Per salvarla ripetete l’operazione di clickaggio della finestrella a striscia, dategli un nome, es: “df0:FIGURA.S”, e premete il pulsante “SAVE”. A questo punto potete anche uscire, perché abbiamo salvato sia il .RAW che il testo della palette in dc.w da includere nella nostra copperlist.

Per visualizzarvi la figura, caricate l’esempio Lezione4b.s e fate queste 2 sostituzioni: cambiate il nome della figura da caricare mettendoci la vostra, e inserite la palette della vostra figura eliminando quella preesistente:

```
1 PIC:
2      incbin  "amiga.320*256*3"
```

lo potete cambiare in:

```
1 PIC:
2      incbin  "df0:figura.raw"
```

oppure, scrivendo <v df0:> dalla linea comandi basta un:

```
1 PIC:
2      incbin  "figura.raw"
```

Per la palette potete fare in 2 modi: o caricate FIGURA.S in un altro buffer e poi lo copiate con <Amiga+b+c+i>, oppure potete usare il comando dell’Asmone <I>, ossia INSERT, che inserisce un testo nel punto in cui vi trovate col cursore prima di premere ESC per passare alla linea comandi. Comunque facciate, togliete la vecchia palette con il CUT (TAGLIA) dell’editor: <Amiga+b> per selezionare il blocco, <Amiga+x> per eliminarlo.

Ha funzionato?? Spero di sì, altrimenti significa che avete sbagliato qualche passaggio, pena la ripetizione di tutta l’operazione.

Per continuare in bellezza ora proviamo a visualizzare una figura a 32 colori: basta che disponiate della consueta figura in 320x256, questa volta a 32 colori (se proprio non la avete caricate il DeLuxe Paint e “schizzate” qualche oscenità). Convertitela come avete fatto per la figura precedente, questa volta noterete che dopo il caricamento le caratteristiche cinceranno con le previsioni: “bitplane \$2800, total \$c800”, infatti ogni bitplane è sempre di \$2800 bytes, mentre il totale è \$c800, ossia 51200 (\$2800*5), essendo una figura a 32 colori composta di 5 bitplanes. Salvate il .RAW e la palette, ad esempio con nomi come FIGURA32.RAW e FIGURA32.s.

Per visualizzarla dovreste fare al sorgente i due cambiamenti di prima, ossia far caricare la nuova figura dall’incbin e sostituire la vecchia palette con quella nuova (che come noterete è più lunga coinvolgendo tutti i 32 registri colore). In più dovete cambiare il numero di bitplanes nella routine di puntamento e aggiungere i bitplanes mancanti alla copperlist:

```
1      MOVE.L  #PIC,d0      ; in d0 mettiamo l'indirizzo della PIC,
2      LEA     BPLPOINTERS,A1 ; in a1 i puntatori in COP
3      **->    MOVEQ  #4,D1    ; numero di bitplanes -1 (qua sono 5!!!!)
4                                     ; per eseguire il ciclo col DBRA
5 POINTBP:
6      ....
```

Cambiate il MOVEQ #2,d1 in MOVEQ #4,d1, ossia mentre prima eseguivamo 3 cicli DBRA per 3 bitplanes (3-1=2), ora eseguiamo 5 cicli (5-1=4) per 5 bitplanes. Ma allora bisogna aggiungere i puntatori di bitplanes mancanti nella copperlist:

```
1 BPLPOINTERS:
2      dc.w $e0,$0000,$e2,$0000      ;primo  bitplane - BPLOPT
3      dc.w $e4,$0000,$e6,$0000      ;secondo bitplane - BPL1PT
4      dc.w $e8,$0000,$ea,$0000      ;terzo  bitplane - BPL2PT
5      dc.w $ec,$0000,$ee,$0000      ;quarto bitplane (AGGIUNTO ORA!)
6      dc.w $f0,$0000,$f2,$0000      ;quinto bitplane (AGGIUNTO ORA!)
```

Ultima e più importante modifica bisogna “accendere” 5 bitplanes anziché 3, questo lo si può fare modificando il \$dff100 (bp1con0) nella copperlist:

```
1      ; 5432109876543210
2  dc.w $100,%0101001000000000 ; bits 14,12 accesi!! (5 = %101)
```

Assemblando il tutto dovrebbe apparire la figura a 32 colori.

Da questi due esempi potete intuire facilmente come visualizzare immagini a 2,4 e 16 colori! basta cambiare il numero di loop nella routine che punta i bitplanes e sistemare i bit giusti in \$dff100 (BPLCON0).

Vediamo ora di visualizzare una figura in EHB a 64 colori e una in HAM a 4096 colori, attivando i due modi grafici speciali.

Partiamo da quella in HAM: fatevi una figura in 320x256 in HAM, oppure cercate una delle tante figure ham che si trovano spesso nei dischi di riviste o nei dischi di immagini “SEXY”, che sono per lo più in HAM, dato che la fedeltà è importantissima per l’immagine di una donna nuda. Credo anzi che sia più piacevole visualizzare una donna nuda che un cesto di frutta. Caricate come da copione la figura in HAM 320x256 col KEFCON e salvate il RAW e la COPPERLIST. Purtroppo il KEFCON ha un errore di programmazione per cui quando vengono caricate su A4000 figure a 6 bitplanes, siano HAM o EHB, si provoca una specie di “spappolamento” dei numeri e dei segni di punteggiatura (.,:) del quadro di comando, (sull’Amiga500/2000/600 funziona invece!) per cui potete vedere correttamente solo le parole, ma non è certo un problema, in quanto basta clickare sulla finestrella a striscia ed aggiungere un .RAW, per esempio, al nome della PIC che è visibile, poi un .s per salvare la copperlist. Le modifiche da fare sono l’aggiunta degli ultimi puntatori al bitplane 6 in copperlist, la sostituzione della palette e il settaggio del numero di cicli della routine di puntamento a 6:

```
1  BPLPOINTERS:
2      dc.w $e0,$0000,$e2,$0000 ;primo bitplane - BPLOPT
3      dc.w $e4,$0000,$e6,$0000 ;secondo bitplane - BPL1PT
4      dc.w $e8,$0000,$ea,$0000 ;terzo bitplane - BPL2PT
5      dc.w $ec,$0000,$ee,$0000 ;quarto bitplane - BPL3PT
6      dc.w $f0,$0000,$f2,$0000 ;quinto bitplane - BPL4PT
7      dc.w $f4,$0000,$f6,$0000 ;sesto bitplane (AGGIUNTO ORA!)
8
9
10 **--> MOVEQ #5,D1 ; numero di bitplanes -1 (qua sono 6!!!!)
11 ; per eseguire il ciclo col DBRA
12 POINTBP:
13     ...
14
15 Nonchè il BPLCON0:
16
17     ; 5432109876543210
18  dc.w $100,%0110101000000000 ; --> 6 plane in HAM lowres (4096 colori)
19 ; BIT 11 settato = HAM!
```

Il funzionamento teorico dell’HAM sarà affrontato meglio in seguito.

Per visualizzare una immagine in Extra Half Bright, invece, convertitene una col KEFCON, fatela caricare dall’incbin, sostituite la palette, lasciate puntare 6 bitplanes alla routine e azzerate il bit 11 del bp1con0:

```
1      ; 5432109876543210
2  dc.w $100,%0110001000000000 ; --> 6 plane in EHB lowres (64 colori)
3 ; BIT 11 azzerato = Extra Half Bright
```

NOTA: Il modo EHB ha 64 colori, ma non sono tutti e 64 selezionabili liberamente, in quanto l’Amiga ha solo 32 registri colore; gli altri 32 colori sono come i primi 32 ma più scuri, a “mezza luce”, ossia *Half Bright*.

Ora che sappiamo visualizzare delle figure, vediamo che effetti si possono fare con i registri di scorrimento. Caricate la LEZIONE5.TXT con <r>

LEZIONE 5 - LO SCROLLING ORIZZONTALE E VERTICALE

In questa lezione tratteremo lo scorrimento orizzontale e verticale delle figure, nonché alcuni effetti speciali.

Cominciamo dallo scorrimento orizzontale: l'Amiga ha un registro speciale dedicato allo scorrimento, il BPLCON1 (\$dff102), che può far scorrere di un pixel alla volta verso destra i bitplane per un massimo di 15 pixel. Ciò è ottenuto dal copper ritardando il trasferimento dei dati dei bitplane, che arrivano “dopo” di uno o più pixel. Si possono inoltre scorrere separatamente i bitplanes pari e quelli dispari: i bitplanes dispari sono chiamati *PLAYFIELD 1* (1, 3, 5), mentre quelli pari *PLAYFIELD 2* (2, 4, 6). Il \$dff102, lungo una word, è diviso in 2 byte: quello alto, ossia quello a sinistra, (\$xx00), composto dai bit dal 15 all'8, non è utilizzato e bisogna lasciarlo a zero, mentre il byte basso (\$00xx) controlla lo scroll:

\$dff102, BPLCON1 - Bit Plane Control Register 1

BITS		NOME-FUNZIONE
15	-	X
14	-	X
13	-	X
12	-	X
11	-	X
10	-	X
09	-	X
08	-	X
07	-	PF2H3 \
06	-	PF2H2 \ 4 bit per scroll PLANES PARI (playfield 2)
05	-	PF2H1 /
04	-	PF2H0 /
03	-	PF1H3 \
02	-	PF1H2 \ 4 bit per scroll PLANES DISPARI (playfield 1)
01	-	PF1H1 /
00	-	PF1H0 /

In pratica si deve agire sulla word in maniera simile ai registri colore: mentre nei registri colore si agisce su 3 componenti RGB, che vanno da 0 a 15, ossia da 0 a \$F, qua agiamo su 2 sole


```

....++....
...+..+...
...++++...
...+..+...
...+..+...
.....

```

Abbiamo un ipotetico bitplane con 10 byte per linea, che può essere a zero (.) o ad 1 (+), in questo caso raffigura una “A”. Per spostare la “A” in alto, dobbiamo “puntare” una linea più in basso, ossia 10 bytes più in basso, e per puntare più in basso, occorre AGGIUNGERE 10 (add.l #10,puntatori)

```

1234567890
....++....
...+..+...
...++++...
...+..+...
...+..+...
.....
.....

```

Allo stesso modo, per farla “scendere”, dobbiamo puntare una linea più in alto, ossia 10 bytes più in alto (SUB.L #10,puntatori):

```

1234567890
.....
.....
....++....
...+..+...
...++++...
...+..+...
...+..+...

```

In pratica per fare questo dobbiamo ricordarci che i puntatori in copperlist hanno l’indirizzo dei plane (che noi cambieremo) divisi in 2 word. Il problema è facilmente risolvibile con una lieve modifica della routine di puntamento dei bitplane, infatti dobbiamo “prendere” l’indirizzo dei bitplanes dalla copperlist (operazione contraria), aggiungere o sottrarre 40 per lo scroll, e rimettere il nuovo indirizzo nella copperlist con la vecchia routine di puntamento. Vedetevi l’esempio Lezione5c.s che usa questo sistema.

Ora caricatevi l’esempio Lezione5d.s, in cui sono presenti le due routines di scroll orizzontale e verticale contemporaneamente.

In Lezione5d2.s troverete un’altra applicazione dello scroll orizzontale insieme al \$dff102 (bplcon1), ossia la distorsione in movimento.

Vedremo ora i registri più importanti per gli effetti speciali video Amiga, ossia i *moduli*: \$dff108 e \$dff10a (BPL1MOD e BPL2MOD). Ci sono due registri modulo perché si può cambiare il modulo separatamente per i bitplanes pari e per quelli dispari, come il BPLCON1 (\$dff102). Per operare sulla nostra figura a 3 bitplanes dovremo agire su entrambi i registri. Avrete notato che quando una immagine in LOW RES 320x256 viene visualizzata, il pennello va a capo ogni 40 bytes, mentre i dati sono tutti di seguito. Allo stesso modo, nel caso di una figura in HI-RES 640x256 il pennello va a capo ogni 80 bytes. Infatti il modulo viene automaticamente assegnato quando si setta il \$dff100 (BPLCON0): se viene selezionato il LOWRES il copper sa che una figura in lowres ha 40 bytes per linea, dunque partendo a visualizzare dall’inizio dello schermo (in alto a sinistra), si legge 40 bytes e scrive col pennello elettronico la prima linea, poi “va a capo” e i dati che seguono li scrive alla linea dopo, e così via. La figura in memoria però ha i dati tutti

consecutivi, non c'è una figura “quadrata” ! La memoria è una fila di byte consecutivi, per cui ogni bitplane è una linea consecutiva di dati: immaginate di dividere le 256 linee dello schermo, lunghe 40 bytes ciascuna, e di metterle l'una dopo l'altra per fare una sola linea di 40×256 bytes, ottenendone una lunga una settantina di metri: questa sarebbe la linea come è veramente in memoria. Mettendo il modulo a zero, come abbiamo fatto fino ad ora, lasciamo andare "a capo" come il LOWRES o HIGHRES comanda, ossia ogni 40 o 80 linee, e la visualizzazione è normale. Il valore che mettiamo al modulo viene *addizionato* ai puntatori ai bitplanes alla *fine* della linea, ossia una volta raggiunto il byte 40. In questo modo possiamo *saltare* dei bytes, che non vengono visualizzati. Per esempio se aggiungiamo 40 ad ogni termine di linea ne saltiamo una intera, per cui ne viene visualizzata una ogni due, infatti:

- IMMAGINE NORMALE -

```
.....          ; al termine di questa linea "salto" 40 bytes
.....+.....
.....+++.....   ; e visualizzo questa linea, poi "salto"...
.....+++++.....
.....+++++..... ; e visualizzo questa linea, poi "salto"...
.....+++++.....
.....+++.....   ; e visualizzo questa linea, poi "salto"...
.....+.....
.....          ; e visualizzo questa linea, poi "salto"...
```

Il risultato sarà che visualizziamo solo una linea ogni due:

- IMMAGINE MODULO 40 -

```
.....          ; al termine di questa linea "salto" 40 bytes
.....+++.....   ; e visualizzo questa linea, poi "salto"...
.....+++++..... ; e visualizzo questa linea, poi "salto"...
.....+++.....   ; e visualizzo questa linea, poi "salto"...
.....          ; e visualizzo questa linea, poi "salto"...
.....
.....
.....
.....
```

La figura apparirà schiacciata, lunga la metà, inoltre andremo a visualizzare anche byte “sotto” la nostra figura, dato che lo schermo finisce sempre alla linea 256: in pratica visualizziamo sempre 256 linee, ma in un raggio di 512 linee di cui visualizziamo solo una linea ogni due. Provate a ricaricare *Lezione5b.s* e modificate i moduli nella copperlist:

```
1      dc.w      $108,40      ; Bpl1Mod
2      dc.w      $10a,40      ; Bpl2Mod
```

Noterete che l'immagine è alta la metà come previsto e la parte inferiore dello schermo è riempita dai bitplane che “avanzano”, ossia dal secondo bitplane visualizzato sotto il primo, e dal terzo visualizzato sotto il secondo mentre dopo il terzo si vede la memoria dopo la figura, insomma vengono visualizzate 256 linee in un raggio di 512. Provate a saltare 2 linee, saltando 80 bytes ogni 40 visualizzati:

```
1      dc.w      $108,40*2    ; Bpl1Mod
2      dc.w      $10a,40*2    ; Bpl2Mod
```

La figura si dimezzerà ancora, e spunteranno in basso altri bytes. Verificherete un dimezzamento dell'altezza continuando con moduli di 40×3 , 40×4 , 40×5 eccetera, fino a rendere illeggibile il disegno. Se scegliete un modulo che non sia multiplo di 40 causerete lo “sfaldamento”

dell'immagine, infatti il copper visualizzerà le linee partendo non dal loro inizio ma da una parte sempre diversa.

Vedetevi `Lezione5e.s` per una veloce routine che aggiunge 40 al modulo per dimezzare la figura.

I moduli oltre che positivi possono essere anche negativi. In questo caso viene sottratto il numero negativo in questione alla fine di ogni linea visualizzata. In questo caso si possono creare effetti strani: immaginatevi di mettere il modulo come -40: in questo caso, il copper legge 40 bytes, li visualizza in una linea, poi torna indietro di 40 bytes, visualizza gli stessi dati nella linea successiva, poi torna indietro di 40 bytes, e così via. In pratica non avanza oltre i primi 40 bytes e ogni linea ricopia la prima linea: se per esempio abbiamo la prima linea tutta nera, le altre riprodurranno questa e lo schermo sarà tutto nero. Se ci fosse un solo punto nel mezzo della linea, questo sarebbe ridisegnato ogni linea e si produrrebbe una riga verticale:

```

.....+......      ; linea 1 (sempre ridisegnata: modulo -40!)
.....+......      ; linea 2
.....+......      ; linea 3
.....+......      ; linea 4
.....+......      ; linea 5
.....+......      ; linea 6
.....+......      ; linea 7
.....+......      ; linea 8
.....+......      ; linea 9
.....+......      ; linea 10

```

Allo stesso modo ogni colore provoca una specie di “colatura” fino alla fine dello schermo. Questo effetto è stato usato in giochi come *Full Contact*, nel *red-sector demomaker* e in moltissimi altri programmi.

Vediamo il suo funzionamento in pratica in `Lezione5f.s`.

Suggestivo e semplice da fare, o sbaglio? è detto anche effetto *flood*. Il modulo viene addizionato, ogni fine linea, ai puntatori dei bitplanes che “camminano” nella memoria per visualizzare tutta la figura. Quindi addizionando un numero negativo, sottraiamo. In questo caso specifico, i puntatori dopo aver trasferito ogni linea assumono il valore $X+40$, vengono quindi incrementati del valore del modulo ($=-40$: la lunghezza in byte di una singola riga di bitplane, al negativo): decrementati dunque di ‘40’ byte, assumono infine di nuovo il valore X di partenza.

```

+---->->->-----+
|                               |
|BPL POINTER=  X+ 0.....39   |
|                               |
|INIZIO RIGA  -+---xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx---+  ULTIMO BYTE ->
|      (X)      |      |                               |      |      (X+39)
|               +---+                               +---+
|               |
| RIGA DOPO  -+----xxxx[...]|
~               |
|               +-X+ 40  (il puntatore, dopo il trasferimento, ha camminato per
~               |      l'intera lunghezza della riga (40 byte), fermandosi
|               |      al 40esimo che altri non è se non il primo byte della
~               |      riga successiva)
|               +-> (Qui viene ADDIZIONATO al puntatore di ogni plane il
|               |      valore del modulo a lui assegnato: in questo caso '-40')
|               +-> X=X+(-40) => X=X-40 => X=0 >-+
|               |
+-----<-<-<-+-----<-<-<-+

```

Visto? Il puntatore, sul più bello, arrivato a $X+40$, viene sottratto di 40 e torna all'inizio della riga appena trasferita, visualizzando ancora la stessa riga in quella sotto, in quanto il pennello

elettronico cammina sempre verso il basso e disegna quanto gli viene “detto” al punto in cui si trova, in questo caso sempre la stessa riga, ripetuta.

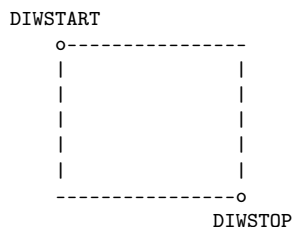
Abbiamo visto in Lezione5f.s anche l'effetto specchio, ossia il modulo -80. Vediamolo da solo nell'esempio Lezione5g.s.

Vediamo ora come utilizzare molti \$dff102 (BPLCON1) consecutivi in copperlist per creare un effetto di ondulazione: caricate Lezione5h.s

Vediamo un utilizzo particolare dello scroll con i bitplanes: Lezione5i.s è un cosiddetto GRAPHIC-SCANNER, un antenato dei GFX-RIPPERS, ossia i programmi che “RUBANO” figure dalla memoria. Questo breve programmino serve semplicemente a mostrare la memoria CHIP, con tutte le figure visibili in essa contenute.

Ancora un esempio inerente ai moduli in Lezione5l.s, questa volta per fare un “allungamento” della figura anziché un dimezzamento.

In Lezione5m.s vedremo un'altro metodo per spostare in basso e in alto la figura, questa volta modificando il DIWSTART (\$dff08e) I registri DIWSTART e DIWSTOP determinano l'inizio e la fine della “finestra video”, ossia la parte rettangolare di schermo dove vengono visualizzati i bitplanes. DIWSTART contiene le coordinate YYXX dell'angolo in alto a sinistra, dove inizia il “rettangolo video”, mentre DIWSTOP contiene le coordinate dell'angolo in basso a destra:



In questi registri però non si possono indicare tutte le possibili coordinate XX e YY, infatti sia la posizione XX che YY sono byte, e come sappiamo i byte possono raggiungere 256 diversi valori (\$00-\$ff). Vediamo in quali posizioni possiamo cominciare la finestra video col DiwStart e in quali possiamo terminarlo col DiwStop.

```

1      dc.w    $8e,$2c81      ; DiwStrt YY=$2c,      XX=$81
2      dc.w    $90,$2cc1      ; DiwStop YY=$2c(+ $ff), XX=$c1(+ $ff)

```

La finestra video normale ha questi valori di DIWSTRT e DIWSTOP; la posizione verticale, la YY, funziona esattamente come la posizione YY dei wait del copper: infatti se col copper aspettate una linea sopra \$2c e ci fate delle sfumature, non saranno visibili perché troppo in alto, o comunque risulterà sopra qualsiasi figura visibile; analogamente al wait dopo la linea \$FF la posizione riparte da \$00, che sarebbe \$FF+1. infatti lo schermo inizia dalla posizione verticale \$2c, e finisce al \$2c dopo la linea 256, ossia \$FF+\$2c, ossia \$12b, visualizzando un totale di 256 linee, come previsto. Per esempio per uno schermo alto 200 linee dovremo mettere questo DiwStop:

```

1      dc.w    $90,$f4c1      ; DiwStop YY=$2c(+ $ff), XX=$f4

```

Infatti \$f4-\$2c = 200. Se indichiamo \$00,\$01... aspetteremo dopo la linea \$ff. Le limitazioni sono queste: il DiwStart può posizionarsi verticalmente in una delle posizioni YY da \$00 a \$FF, ossia fino alla linea 200. La finestra video dunque non può comunicare dalla linea 201 o seguenti, sempre prima. Per il DIWSTOP i progettisti si sono serviti di uno stratagemma: se il valore YY è sotto \$80, ossia 128, allora aspetta le linee sotto \$FF, per cui il \$2c si riferisce a \$2c+\$FF, ossia la linea 256. Se il numero è superiore a \$80 allora lo prende così com'è, (dato che non esistono linee \$80+\$ff=383!!), e aspetta veramente le linee 129,130 eccetera. Dunque, se il DIWSTART può arrivare al massimo alla linea \$FF partendo dallo ZERO, il DIWSTOP può superare

la linea \$FF e arrivare ai limiti del video in basso, ma non può partire da linee inferiori alla \$80. Questo trucco è stato fatto considerando i numeri con il bit 7 a zero (quelli, appunto, prima \$80), come se avessero un ipotetico bit 8 impostato, il che aumenta tutto di \$FF. Quando invece il bit 7 viene impostato (i numeri dopo \$80 lo hanno impostato) allora il bit fantasma sparisce e i numeri sono presi per quello che sono. Per quanto riguarda la linea orizzontale il `diwstart` può partire da una qualsiasi `XX` da \$00 a \$FF, quindi fino alla posizione 256, (ricordatevi però che lo schermo parte dalla posizione \$81 e non da \$00, dunque è la posizione 126 dall'inizio dello schermo!). Il `DiwStop` invece con \$00 indica la linea 127, e proseguendo può raggiungere la fine del bordo destro dello schermo, infatti ha il bit 8 “fantasma” sempre ad 1, per cui viene sempre aggiunto \$FF al suo valore di `XX`. In definitiva il `DiwStart` può posizionarsi in una qualsiasi delle posizioni `XX` e `YY` da \$00 a \$FF, mentre il `DiwStop` può posizionarsi orizzontalmente dopo la linea \$FF, e verticalmente dalla linea \$80 alla linea \$FF, dopodiché i numeri da \$00 a \$7f sono, come nel `wait` dopo la linea \$FF, le linee 201,202 eccetera, per cui \$2c è \$2c+\$ff.

In `Lezione5m2.s`, `Lezione5m3.s` e `Lezione5m4.s` viene trattato questo argomento.

Come termine della LEZIONE5, caricatevi `Lezione5n.s`, che è un riassunto delle lezioni precedenti e in più è il primo listato che suona anche la musica.

Una volta capito questo esempio, non vi resta che caricare la `LEZIONE6.TXT`.

LEZIONE 6 - I FONTS E LO SCROLLING

In questa lezione vedremo come visualizzare dei testi sullo schermo, come far scorrere schermi più grandi della finestra video, e l'uso delle tabelle di valori predefiniti per simulare movimenti di rimbalzo e ondeggiamento.

Imparare a visualizzare delle scritte sullo schermo è importantissimo, non si può fare a meno di una routine di stampa caratteri in un gioco o in una demo grafica: se vogliamo scrivere il punteggio e il numero delle vite, o un messaggio tra un livello e l'altro, oppure il dialogo tra i personaggi, una scritta con i saluti agli amici, eccetera. È chiaro che non vengono visualizzate delle figure 320x256 con le scritte già fatte! Immaginatevi di voler visualizzare 5 pagine di testo per introdurre la storia del vostro gioco: “un cavaliere di un periodo storico imprecisato decise di andare alla ricerca del santo graal...” eccetera. Le soluzioni sono due: o vi disegnate col programma da disegno cinque figure col testo stampato, e in questo caso avremmo 5 figure da $40 \times 256 = 51200$ byte utilizzati, che vi rubano spazio su disco e memoria, oppure con 1k di FONT caratteri e pochi byte di routine che stampa quei caratteri fate lo stesso lavoro, risparmiando 50k. Avrete presenti i FONT di caratteri del sistema operativo: TOPAZ, DIAMOND eccetera, che potete scegliere? Ebbene a noi non interessano i FONT di sistema, perché ne usiamo di nostri. Si possono usare anche i font di sistema, ma sono limitati, mentre facendosi i font e la routine che stampa i caratteri di quel font si possono visualizzare scritte di qualsiasi dimensione, anche colorate, basta disegnare il font e farsi la routine giusta. Una volta capito il sistema di *print*, ossia di *stampa* dei caratteri si possono fare variazioni senza difficoltà. Per cominciare vediamo come stampare un font piccolo, largo 8 pixel e alto 8, ad un solo colore. Come prima cosa bisogna disporre di un *bitplane* dove stampare il testo e di un *font caratteri* dove sono disegnati tutti i caratteri da copiare. Per il bitplane non ci sono problemi, infatti basta crearsi nel listato un pezzo di memoria azzerato della dimensione di un bitplane, e “puntarlo”, ossia farlo visualizzare. Per fare uno spazio azzerato si può usare il comando `DCB.B 40*256,0` che, appunto, crea uno spazio azzerato della dimensione giusta; ma esiste una SECTION specifica per i *buffer azzerati*: la `section BSS`, in cui si può usare la sola direttiva `DS.B/DS.w/DS.l`, che stabilisce quanti bytes/word/longword azzerati creare. Il vantaggio sta nella lunghezza finale del FILE ESEGUIBILE: mentre creando lo spazio azzerato con un: `BITPLANE: dcb.b 40*256,0` i 10240 bytes sono aggiunti alla lunghezza totale del file, definendo una `Section BSS`:

! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G H I J K L M N O

ECCETERA ECCETERA.....

Il font 8x8 che usiamo nel corso non è altro che una figura del genere in RAW. In realtà questo tipo di font viene fatto normalmente con un apposito EDITOR, un programma dedicato al disegno di questi font 8x8 ad un colore. Per i font più grandi e colorati però conviene disegnare le lettere in una figura, normalmente 320x256, e usare una routine propria per prelevare i caratteri da stampare. Per cominciare però vediamo il font più semplice come viene stampato a video: innanzitutto bisogna preparare una stringa di testo con le parole da stampare, ad esempio:

1	dc.b	"Prima scritta!"	; nota: si possono usare '' oppure ""
2			
3	EVEN		; ossia allinea ad indirizzo PARI

La direttiva EVEN serve ad evitare gli indirizzi dispari per le istruzioni o i dati che si trovano sotto il dc.b. Le stringhe di testo sono composte di bytes e può succedere che siano un numero dispari, in tal caso la label sottostante sarà ad un indirizzo dispari, e questo può generare errori di assemblaggio: infatti, nel 68000, le istruzioni devono sempre essere ad indirizzi pari, e anche i dati dovrebbero essere ad indirizzi pari per evitare GURU MEDITATION in fase di esecuzione, infatti un MOVE.L o MOVE.W eseguito su un indirizzo dispari causa un bel Crash con GURU MEDITATION ed esplosioni. Ricordatevi dunque di mettere sempre un EVEN al termine di una stringa di testo, o di accertarvi che sia pari. Potete anche aggiungere uno zero in più al termine della stringa per pareggiare il conto, come ho fatto per GfxName:

```

1 GfxName:
2   dc.b    "graphics.library",0,0
3
4 Potete scrivere anche:
5
6 GfxName:
7   dc.b    "graphics.library",0
8   even

```

Infatti basta uno zero alla fine del testo, l'altro lo metterà EVEN. Dunque, una volta stabilita la stringa di testo da visualizzare, basta vedere come copiare i caratteri giusti al posto giusto. Vi propongo già la routine che stampa un carattere:

```

1 PRINT:
2   LEA     TESTO(PC),A0      ; Indirizzo del testo da stampare in a0
3   LEA     BITPLANE,A3      ; Indirizzo del bitplane destinazione in a3
4   MOVEQ   #0,D2            ; Pulisci d2
5   MOVE.B  (A0),D2          ; Prossimo carattere in d2
6   SUB.B   #$20,D2          ; TOGLI 32 AL VALORE ASCII DEL CARATTERE, IN
7                               ; MODO DA TRASFORMARE, AD ESEMPIO, QUELLO
8                               ; DELLO SPAZIO (che è $20), in $00, quello
9                               ; DELL'ASTERISCO ($21), in $01...
10  MULU.W   #8,D2            ; MOLTIPLICA PER 8 IL NUMERO PRECEDENTE,
11                               ; essendo i caratteri alti 8 pixel
12  MOVE.L   D2,A2            ;
13  ADD.L    #FONT,A2         ; TROVA IL CARATTERE DESIDERATO NEL FONT...
14
15                               ; STAMPIAMO IL CARATTERE LINEA PER LINEA
16  MOVE.B   (A2)+(A3)        ; stampa LA LINEA 1 del carattere
17  MOVE.B   (A2)+,40(A3)     ; stampa LA LINEA 2 " "
18  MOVE.B   (A2)+,40*2(A3)   ; stampa LA LINEA 3 " "
19  MOVE.B   (A2)+,40*3(A3)   ; stampa LA LINEA 4 " "
20  MOVE.B   (A2)+,40*4(A3)   ; stampa LA LINEA 5 " "
21  MOVE.B   (A2)+,40*5(A3)   ; stampa LA LINEA 6 " "
22  MOVE.B   (A2)+,40*6(A3)   ; stampa LA LINEA 7 " "
23  MOVE.B   (A2)+,40*7(A3)   ; stampa LA LINEA 8 " "
24
25  RTS

```

Avete già capito??? Analizziamola punto per punto:

```

1   LEA     TESTO(PC),A0      ; Indirizzo del testo da stampare in a0
2   LEA     BITPLANE,A3      ; Indirizzo del bitplane destinazione in a3
3   MOVEQ   #0,D2            ; Pulisci d2
4   MOVE.B  (A0),D2          ; Prossimo carattere in d2

```

Fino a qua non ci sono problemi, abbiamo in d2 il valore del carattere, se fosse una "A", allora abbiamo \$41 in d2

```

1   SUB.B   #$20,D2          ; TOGLI 32 AL VALORE ASCII DEL CARATTERE, IN
2                               ; MODO DA TRASFORMARE, AD ESEMPIO, QUELLO
3                               ; DELLO SPAZIO (che è $20), in $00, quello
4                               ; DELL'ASTERISCO ($21), in $01...

```

Anche qua cosa succede è chiaro, vediamo perché sottraiamo 32 (\$20):

```

1   MULU.W   #8,D2            ; MOLTIPLICA PER 8 IL NUMERO PRECEDENTE,
2                               ; essendo i caratteri alti 8 pixel
3   MOVE.L   D2,A2            ;
4   ADD.L    #FONT,A2         ; TROVA IL CARATTERE DESIDERATO NEL FONT...

```

Questa operazione porta ad avere in A2 l'indirizzo del carattere "A" presente nel font, ossia l'indirizzo da dove dobbiamo "prendere" il carattere per copiarlo nel bitplane che stiamo visualizzando. Vediamo cosa è successo: ricordate che i caratteri sono stati messi nel font nello stesso ordine dello standard ASCII? Per cui, disponendo del valore ASCII del carattere, in questo caso \$41 per la "A", possiamo individuare a che distanza dall'inizio del FONT si trova la "A" in RAW! Se ogni carattere è di 8x8 pixel, significa che è lungo 8 bit, ossia un byte a linea * 8 linee, in totale 8 bytes. Dunque lo spazio (il primo carattere nel FONT) si trova all'inizio del FONT stesso e finisce al byte 8, dove inizia "!" (il secondo), e così via. Avendo sottratto \$20 al valore ASCII, il valore dello spazio diventerà \$00, il carattere successivo "!" \$01, eccetera (la "A" risulterà \$21), dunque basta moltiplicare per 8 il numero ottenuto dopo la sottrazione per ricavare la distanza, dall'inizio del FONT, del carattere in questione!!! Rivediamo il passaggio:

1	SUB.B	#\$20 ,D2	;	TOGLI 32 AL VALORE ASCII DEL CARATTERE, IN
2			;	MODO DA TRASFORMARE, AD ESEMPIO, QUELLO
3			;	DELLO SPAZIO (che è \$20), in \$00, quello
4			;	DELL'ASTERISCO (\$21), in \$01...
5	MULU.W	#8,D2	;	MOLTIPLICA PER 8 IL NUMERO PRECEDENTE,
6			;	essendo i caratteri alti 8 pixel

Ora in D2 abbiamo la distanza (l'offset) dell'inizio del carattere dall'inizio del FONT! Ora per trovare l'indirizzo effettivo del carattere, aggiungiamo la "distanza dall'inizio" che abbiamo in D2 all'indirizzo del FONT:

```
1      MOVE.L D2,A2
2      ADD.L  #FONT,A2          ; TROVA IL CARATTERE DESIDERATO NEL FONT...
```

Ora abbiamo in a2 l'indirizzo dove si trova il nostro carattere da copiare, ad esempio la "A". Basterà ora copiarla da FONT allo schermo, cioè al BITPLANE 320x256, in cui ogni linea è lunga 40 bytes:

1			; STAMPIAMO IL CARATTERE LINEA PER LINEA
2	MOVE.B	(A2)+,(A3)	; stampa LA LINEA 1 del carattere
3	MOVE.B	(A2)+,40(A3)	; stampa LA LINEA 2 " "
4	MOVE.B	(A2)+,40*2(A3)	; stampa LA LINEA 3 " "
5	MOVE.B	(A2)+,40*3(A3)	; stampa LA LINEA 4 " "
6	MOVE.B	(A2)+,40*4(A3)	; stampa LA LINEA 5 " "
7	MOVE.B	(A2)+,40*5(A3)	; stampa LA LINEA 6 " "
8	MOVE.B	(A2)+,40*6(A3)	; stampa LA LINEA 7 " "
9	MOVE.B	(A2)+,40*7(A3)	; stampa LA LINEA 8 " "

La copia avviene per “linee”, infatti il carattere è alto 8 linee, ognuna delle quali è larga 8 bit (1 byte):

12345678

```

...####. linea 1 - 8 bit, 1 byte
..#...#. 2
..#...#. 3
...####. 4
..#...#. 5
..#...#. 6
..#...#. 7
..... 8

```

Dunque per copiarlo una linea alla volta occorre copiarne un byte alla volta. Ma lo schermo destinazione è largo 40 bytes per linea, e dobbiamo considerare che ogni linea deve essere allineata l'una sotto l'altra, se non saltiamo 40 bytes ogni volta copieremmo così il carattere:

...###...#...#...#...#...#####...#...#...#...#...#...#...

Invece dobbiamo copiare un byte, poi ANDARE A CAPO saltando 40 bytes, e copiare un altro byte:

```
1 MOVE.B (A2)+,(A3) ; stampa LA LINEA 1 del carattere
```

Sul monitor:

```
...###..
```

```
1 MOVE.B (A2)+,40(A3) ; stampa LA LINEA 2 (40 bytes dopo)
```

Sul monitor:

```
...###..
..#...#.
```

```
1 MOVE.B (A2)+,40*2(A3) ; stampa LA LINEA 3 (80 bytes dopo)
```

Sul monitor:

```
...###..
..#...#.
```

Eccetera. Per uno schermo largo 80 bytes (640x256 HIRES) basterebbe cambiare la routine così:

```
1 MOVE.B (A2)+,(A3) ; stampa LA LINEA 1 del carattere
2 MOVE.B (A2)+,80(A3) ; stampa LA LINEA 2 " "
3 MOVE.B (A2)+,80*2(A3) ; stampa LA LINEA 3 " "
4 MOVE.B (A2)+,80*3(A3) ; stampa LA LINEA 4 " "
5 MOVE.B (A2)+,80*4(A3) ; stampa LA LINEA 5 " "
6 MOVE.B (A2)+,80*5(A3) ; stampa LA LINEA 6 " "
7 MOVE.B (A2)+,80*6(A3) ; stampa LA LINEA 7 " "
8 MOVE.B (A2)+,80*7(A3) ; stampa LA LINEA 8 " "
```

Vediamo in pratica la stampa di questa "A" su un bitplane in Lezione6a.s.

Ora passeremo a stampare un'intera riga di testo con Lezione6b.s.

E infine stampiamo quante righe vogliamo in Lezione6c.s. Questa routine è quella *definitiva*, che potete usare quando volete scrivere qualcosa a video.

Perché non disegnarsi il proprio font di caratteri? In Lezione6c2.s il FONT è nel listato in dc.b come questo esempio:

```
1 ; "B"
2 dc.b %01111110
3 dc.b %01100011
4 dc.b %01100011
5 dc.b %01111110
6 dc.b %01100011
7 dc.b %01100011
8 dc.b %01111110
9 dc.b %00000000
```

I caratteri sono messi in memoria con dei dc.b % (binario). Potete cambiare ogni singolo carattere come volete. Se fate un vostro font, salvatelo su un disco formattato o sull'HARD DISK!

Ora abbiamo l'occasione di provare una cosa che non abbiamo mai fatto prima: nello stesso schermo proviamo a far convivere una figura in LOWRES a 8 colori e un bitplane in HIRES. L'Amiga infatti può visualizzare contemporaneamente diverse risoluzioni video, (cosa che non mi risulta possa fare il PC MSDOS), basta mettere un WAIT nella copperlist e ridefinire sotto di esso il ??, proprio come se definissimo i colori per fare una sfumatura! Per esempio potremmo visualizzare dalla prima linea alla linea \$50 una figura in HAM a 4096 colori in LOWRES, sotto di essa una in HIRES a 16 colori, sotto ancora una in LOWRES a 32 colori, e così via. In alcuni giochi per esempio la schermata dove si muovono i personaggi è in LOWRES, mentre il pannello

con il punteggio e simili è in HIRES (vedi AGONY). Visualizziamo subito la figura in LOWRES sopra una in HIRES in Lezione6d.s.

Vediamo ora un “trucchetto” che ci permette di ottenere un effetto di “rilievo” alle parole che stampiamo: in Lezione6e.s attiviamo 2 bitplane anziché 1 e sovrapponiamo il secondo al primo, ma il secondo spostato in basso di una linea. Cosa succede se mettiamo due immagini uguali trasparenti l’una sull’altra? L’immagine si sdoppia!!! E se scegliamo i colori giusti, facendo più chiaro lo sdoppiamento in “alto” e più scuro quello in “basso” cosa succede? Che abbiamo capito come funziona Lezione6e.s.

A proposito di sovrapposizioni, perché non attivare un bitplane “sopra” una figura per scriverci?? Vediamo in Lezione6f.s cosa succede.

In Lezione6g.s viene evidenziato l’effetto *trasparenza* muovendo la scritta sopra la figura.

In Lezione6h.s, invece, troverete un modo per stampare testi a 3 colori, sovrapponendo due testi in due bitplanes.

In Lezione6i.s viene fatto lampeggiare uno dei 3 colori del testo, usando una *tabella* di valori predefiniti. Abbiamo già parlato di tabelle nella LEZIONE1, ora vediamo in pratica il vantaggio che portano.

In Lezione6l.s viene usata una variazione della routine che legge da una tabella per variare un colore; la variazione consiste nel fatto che anziché leggere dall’inizio alla fine della tabella e ripartire da capo, rilegge la tabella all’indietro, cioè dalla fine all’inizio.

Le tabelle possono essere utili o indispensabili per molti usi, ad esempio per simulare movimenti di rimbalzi o di oscillazioni. Vediamo in pratica la superiorità dell’uso di una tabella rispetto a semplici ADD e SUB nel movimento di una figura in Lezione6m.s.

A proposito di movimento, per ora abbiamo visto lo scroll orizzontale tramite il BPLCON1 (\$dff102) che permette uno scorrimento massimo di 16 pixel. Ma allora come si fa a scorrere lo schermo a destra e a sinistra quanto vogliamo?? La risposta è abbastanza semplice: basta usare anche i puntatori ai bitplanes! Infatti, tramite i puntatori ai bitplanes abbiamo già visto che possiamo scorrere in alto e in basso, basta aggiungere o sottrarre la lunghezza di una linea (40 in lowres e 80 in HIRES). Ma possiamo scorrere anche in avanti e indietro, per la precisione a “scatti” di 8 pixel alla volta, basta sottrarre o aggiungere 1 al puntatore bitplane e abbiamo spostato a destra o a sinistra la figura di un byte, ossia 8 bit, ossia 8 pixel. Se possiamo scorrere di 8 pixel alla volta con i Bitplane Pointers e di 1 alla volta con il \$dff102 (BPLCON1), basterà scorrere 8 pixel uno alla volta col \$dff102, appunto, poi “scattare” 8 pixel più avanti con un:

```
1 subq.l #1,BITPLANEPOINTER
```

E azzerare contemporaneamente il BPLCON1 (\$dff102), andando al nono pixel, dopodiché scorrere di altri 8 pixel col \$dff102 di un pixel alla volta, giungendo al pixel 9+8= 11, poi scattare in avanti di 8 pixel col Bitplane Pointer eccetera. Negli esempi però, considerando che il \$dff102 può scorrere fino ad un massimo di \$FF, ossia da 0 a 15, e non solo da 0 a 7, ho adottato questa tecnica: per scorrere di 16 pixel alla volta basta aggiungere o sottrarre 2 ai puntatori bitplane (dato che con 1 spostavamo la PIC di 8 pixel) Dunque scorro un pixel alla volta col \$dff102 usando la sua possibilità massima, cioè da \$00 a \$FF, totale 16 posizioni, dopodiché “scatto” ai 16 pixel seguenti con un ADDQ o SUBQ #2,BITPLANEPOINTERS. Ecco una routine che scorre verso destra un bitplane di un pixel alla volta per quanti pixel vogliamo: considerate che MIOBPCON1 è il byte del \$dff102.

```
1 Destra :
2     CMP.B  #$ff,MIOBPCON1 ; siamo arrivati al massimo scorrimento? (15)
3     BNE.s  CONIADDA       ; se non ancora, scorri in avanti di 1
4                                     ; con il BPLCON1
5
6 ;     Legge l'indirizzo del bitplane
7
8     LEA    BPLPOINTERS,A1 ; Con queste 4 istruzioni preleviamo dalla
```

```

9      move.w 2(a1),d0      ; copperlist l'indirizzo dove sta puntando
10     swap  d0             ; attualmente il $dff0e0 e lo poiniamo in d0
11     move.w 6(a1),d0
12
13 ;      Scorre a destra di 16 pixel col puntatore bitplane
14
15     subq.l #2,d0          ; punta 16 bit più indietro ( la PIC scorre
16                             ; verso destra di 16 pixel)
17
18 ;      Fa ripartire da zero il BPLCON1
19
20     clr.b  MIOBPCON1      ; azzera lo scroll hardware BPLCON1 ($dff102)
21                             ; infatti abbiamo "saltato" 16 pixel con il
22                             ; bitplane pointer, ora dobbiamo ricominciare
23                             ; da zero con il $dff102 per scattare a
24                             ; destra di un pixel alla volta.
25
26     move.w d0,6(a1)       ; copia la word BASSA dell'indirizzo del plane
27     swap  d0              ; scambia le 2 word
28     move.w d0,2(a1)       ; copia la word ALTA dell'indirizzo del plane
29     rts                  ; esci dalla routine
30
31 CONIADDA:
32     add.b  #$11,MIOBPCON1 ; scorri a destra di 1 pixel la figura
33     rts

```

La routine aumenta di uno il BPLCON1 (\$dff102), facendolo passare per le 16 posizioni possibili: 00, 11, 22, 33, 44, 55, 66, 77, 88, 99, aa, bb, cc, dd, ee, ff dopodiché salta al pixel ff+1 facendo 2 operazioni:

1. Puntare 2 bytes (1 word, 16 bits) più indietro i puntatori bitplanes, facendo scorrere a destra di 16 pixel la figura (dunque 1 pixel dopo la posizione \$FF, ossia 15 raggiunta il fotogramma precedente dal \$dff102)
2. Azzerare il \$dff102, dato che abbiamo “saltato” 16 pixel, altrimenti si sommerebbero i 16 pixel aggiunti col Puntatore Bitplane e i 15 (\$FF) raggiunti col \$dff102 (BPLCON1). Invece azzerando il BPLCON1 ripartiamo da \$00+16= sedicesimo pixel, dopodiché andremo ai seguenti 15 con il BPLCON1, lasciando inalterato il puntatore bitplane.

Se non fosse ancora chiaro, seguite questo schemino, tenendo presente che # è la “figura” che spostiamo verso destra:

		VAL.	BPLCON1	—	BYTE	SOTTRATTI	AI	PUNT.	PLANE
1									
2									
3	#		\$00	—	0	—	tot.	pixel:	
4	#		\$11	—	0	—		1	
5	#		\$22	—	0	—		2	
6	#		\$33	—	0	—		3	
7	#		\$44	—	0	—		4	
8	#		\$55	—	0	—		5	
9	#		\$66	—	0	—		6	
10	#		\$77	—	0	—		7	
11	#		\$88	—	0	—		8	
12	#		\$99	—	0	—		9	
13	#		\$aa	—	0	—		10	
14	#		\$bb	—	0	—		11	
15	#		\$cc	—	0	—		12	
16	#		\$dd	—	0	—		13	
17	#		\$ee	—	0	—		14	
18	#		\$ff	—	0	—		15	
19	#		\$00	—	2	—		16	
20	#		\$11	—	2	—		17	
21	#		\$22	—	2	—		18	
22	#		\$33	—	2	—		19	
23	#		\$44	—	2	—		20	
24	#		\$55	—	2	—		21	
25	#		\$66	—	2	—		22	
26	#		\$77	—	2	—		23	

27
28 eccetera....

Questo schema parla da solo: per esempio se vogliamo scorrere verso destra di 22 pixel un bitplane basta sottrarre 2 al bitplane pointer e mettere \$66 al BPLCON1 (\$dff102).

Per scorrere a sinistra dovremo invece aggiungere 2 ai puntatori bitplanes ogni 16 pixel e procedere al contrario con il \$dff102: \$ff,\$ee,\$dd...

Vediamo in Lezione6n.s la routine in funzione. Noterete un imprevisto: sul lato sinistro avviene un disturbo a scatti; questo non è dovuto ad errori nella routine, ma ad una caratteristica dell'hardware di Amiga, per toglierlo basta un piccolo accorgimento già presente nelle modifiche consigliate del listato stesso.

Già che sappiamo scorrere quanto vogliamo anche in orizzontale, perché non scorrere un bitplane più grande della finestra video?? Esattamente facciamo scorrere uno schermo largo 640 pixel in uno largo 320 spostandolo a destra e sinistra, tutto questo in Lezione6o.s.

Abbiamo già visto per le tabelle l'utilizzo di una longword come puntatore ad un indirizzo:

1 PUNTATORE:
2 DC.L TABELLA

Nella longword "puntatore" viene assemblato l'indirizzo di tabella, per cui possiamo tenere "il conto" di dove siamo arrivati nella tabella aggiungendo o sottraendo la lunghezza di un elemento della tabella. Dobbiamo salvare l'indirizzo a cui siamo arrivati ogni volta perché la routine viene eseguita ogni fotogramma e non continuamente, dunque possono essere eseguite anche altre routine prima che quella routine sia eseguita nuovamente. Quando questa routine viene rieseguita, deve continuare a prelevare valori dalla tabella da dove era rimasta la volta prima, e lo può fare leggendo l'indirizzo in PUNTATORE con un semplice:

1 MOVE.L PUNTATORE(PC),d0 ; In d0 l'indirizzo dove siamo arrivati
2 ; l'ultima volta.

Prima di uscire dalla routine basterà salvare l'ultima posizione. Questo espediente può essere usato per molti scopi, per esempio per poter stampare un carattere solo ogni fotogramma, anziché stampare tutto il testo e poi vederlo. Per fare ciò basta modificare la routine PRINT: e farsi due puntatori: uno che punti all'ultimo carattere stampato, ed uno che punti all'ultimo indirizzo nel bitplane dove abbiamo stampato l'ultimo carattere. In questo modo è come se stampassimo un carattere, congelassimo la routine per tutto un fotogramma, la riattivassimo per stampare un carattere, poi la ricongelassimo eccetera. In realtà anziché congelarla la eseguiamo per stampare un solo carattere, poi salviamo il punto dove siamo arrivati, usciamo dalla routine, aspettiamo che passi il fotogramma, rieseguiamo la routine ripartendo dal punto dove siamo arrivati, risolviamo tutto, usciamo eccetera. Il listato che mette in pratica questa possibilità è Lezione6p.s.

In un bitplane oltre che stampare testo possiamo anche creare disegni con routine apposite, come scacchiere, trame e tessiture. Basta porre ad 1 i bit giusti!!! In Lezione6q.s ci sono delle routine esempio.

Siamo giunti alla fine della LEZIONE6, non ci resta che mettere insieme i listati e le "novità" di questa lezione nel consueto listatone finale di esempio con musica: Lezione6r.s.

Ora passeremo allo studio degli sprite. Quello che dovete fare è caricare la LEZIONE7.TXT, dopodiché dovete cambiare il path per caricare gli incbin dei suoi listati, con <V DFO:SORGENTI3>. I sorgenti, infatti, si trovano nella directory SORGENTI3 del disco 1.

LEZIONE 7 - GLI SPRITES E I DISPOSITIVI DI INPUT

In questa lezione parleremo degli sprites, del joystick e delle istruzioni 68000 riguardanti le operazioni sui bit come AND, OR, EOR, NOT, LSR, ROL...

Ricordatevi di scrivere `V df0:SORGENTI3` per poter caricare i file `.raw` dalla directory dove si trovano i listati di questa lezione.

Gli sprite sono oggetti grafici di una dimensione precisa, larghi al massimo 16 pixel, che si possono muovere per lo schermo indipendentemente dai bitplanes, per esempio il puntatore a freccia che muovete col mouse per selezionare dai menu o premere “pulsanti” è uno sprite gestito dal sistema operativo, che può muoversi dove vuole senza curarsi dei bitplanes che gli stanno “sotto”.

Gli sprite si potrebbero considerare come immagini “fantasma” che si aggirano “sopra” i bitplanes, ma non tutte le cose che si muovono sono sprites! Infatti ci possono essere solo 8 sprites al massimo, essendoci sono solo 8 puntatori in copperlist per gli sprites:

```
1  COPPERLIST:
2
3  SpritePointers:
4      dc.w    $120,0,$122,0    ; Puntatore per lo Sprite 0
5      dc.w    $124,0,$126,0    ; Puntatore per lo Sprite 1
6      dc.w    $128,0,$12a,0    ; " " " " " " 2
7      dc.w    $12c,0,$12e,0    ; " " " " " " 3
8      dc.w    $130,0,$132,0    ; " " " " " " 4
9      dc.w    $134,0,$136,0    ; " " " " " " 5
10     dc.w    $138,0,$13a,0    ; " " " " " " 6
11     dc.w    $13c,0,$13e,0    ; " " " " " " 7
```

I puntatori agli sprite si chiamano registri `SPRxPT` (al posto della `x` si mette il numero dello sprite: abbiamo dunque `SPR0PT`, `SPR1PT`, ... `SPR7PT` e quando parliamo di `SPRxPT` ci riferiamo in generale a tutti gli 8 puntatori).

Per ora li abbiamo messi nella copperlist azzerati solo per evitare che questi oggetti “fantasma” saltellino sulle nostre figure senza controllo. Gli sprite sono isolati dal resto dello schermo, come fossero in una “busta trasparente” applicata sopra il monitor, infatti la risoluzione degli sprite è sempre il lowres, 320x256, anche se i bitplanes sottostanti sono in hires o interlacciati.

Una verifica che gli sprite non fanno parte dei bitplane è che per muoverli non occorre cancellarli e ridisegnarli più avanti ogni volta, come avremmo invece dovuto fare per spostare un pezzo di grafica in un bitplane.

Per muovere uno sprite basta cambiare le sue coordinate agendo con poche e veloci istruzioni su appositi byte dedicati a questo compito che si trovano all'inizio della struttura dati dello sprite stesso.

Quando gli sprites non bastano a fare astronavi e ometti in un gioco viene usato il blitter per copiare blocchi di grafica (*bob*), che vedremo in seguito. Come già detto, la dimensione di uno sprite è di 16 pixel di larghezza, mentre l'altezza può essere scelta a piacere, anche tutto lo schermo, cioè 256 linee. Per fare un mostro di fine livello si potrebbero usare tutti e 8 gli sprites affiancati, raggiungendo la larghezza totale di $16 \times 8 = 128$ pixel.

Il problema è che tale mostro sarebbe poco colorato per i tempi che corrono, infatti uno sprite può avere 3 colori al massimo, dato che il "quarto" è la parte "trasparente", ossia la parte in cui traspare lo sfondo, ossia i bitplanes.

La caratteristica degli sprites è che sono semplici da fare e da animare. Infatti lo sprite si può disegnare con un programma da disegno, basta che sia largo non più di 16 pixel e che abbia 3 colori più lo sfondo, ossia 4, e può essere convertito in *SPRITE* dall'*IFFCONVERTER KEFCON*.

Oppure si può disegnare direttamente in binario, come abbiamo visto per il font 8x8:

```

1      - piano 1 -      - piano 2 -      ; la sovrapposizione di
2                                     ; questi 2 "piani" di
3      dc.w      %0111110000000000,%0111110000000000      ; bit determina il
4      dc.w      %1000001000000000,%1111111000000000      ; colore.
5      dc.w      %1111010000000000,%1000110000000000      ; Questa è la freccia
6      dc.w      %1111101000000000,%1000110000000000      ; puntatore di default
7      dc.w      %1111101000000000,%1001001100000000      ; del kickstart 1.3, la
8      dc.w      %1110111010000000,%1010100110000000      ; riconoscete??
9      dc.w      %0100011101000000,%0100010011000000
10     dc.w      %0000001110100000,%0000001001100000
11     dc.w      %0000000111100000,%0000000100100000
12     dc.w      %0000000011000000,%0000000011000000
13     dc.w      %0000000000000000,%0000000000000000
14
15     dc.w      0,0 ; Due word azzerate indicano la fine dello sprite.

```

In questo caso la larghezza è di 16 pixel e non di 8 come nel font 8x8, per cui lo disegniamo in una word (dc.w) e non in un byte. Inoltre ha 3 colori più il trasparente, ossia 4 possibilità come una figura a 2 bitplanes, dunque servono un paio di "piani" proprio come per i bitplanes, e la loro sovrapposizione determinerà il colore, che può essere:

	Piano 1 - Piano 2		
binario:	0	- 0	= COLORE 0 (TRASPARENTE)
binario:	1	- 0	= COLORE 1
binario:	0	- 1	= COLORE 2
binario:	1	- 1	= COLORE 3

Infatti, come abbiamo già visto, con 2 piani di bit si possono formare 4 combinazioni diverse: %00,%01,%10,%11

Per decidere la posizione dello sprite basta inserire le coordinate X ed Y nei primi byte dello sprite stesso. Infatti, prima dei dati del disegno, lo sprite è composto da 4 byte, ossia 2 word, dette *word di controllo*, ed in questi byte vanno scritte le coordinate sullo schermo dello sprite. Per essere più esatti, il primo byte, detto VSTART, contiene la posizione verticale di inizio dello sprite; il secondo byte invece contiene la posizione orizzontale (HSTART). Il terzo contiene la posizione della fine dello sprite in senso verticale: per determinarla basta aggiungere l'altezza dello sprite alla posizione inizio, e come risultato avremo la posizione verticale dove finisce lo sprite.

Il quarto byte contiene dei bit per funzioni speciali che vederemo. VSTART e HSTART (Vertical Start e Horizontal Start) dunque sono le coordinate dell'angolo in alto a sinistra dove inizia lo sprite:

```
#....
....
....
....
....
```

Mentre VSTOP è la posizione verticale dove termina lo sprite:

```
....
....
....
....
##### -> linea verticale indicata da VSTOP.
```

Per esempio, uno sprite visualizzato alla posizione XX=\$90 e YY=\$50, lungo 20 pixel, comincerebbe così:

```
1      ; IY IX FY      - IY=Inizio Y, IX=Inizio X, FY=Fine Y
2  SPRITE:
3      dc.w      $5090,$6400      ; Y=$50, X=$90, altezza= $50+20, cioè $64
4  ; da qua iniziano i dati dei 2 piani dello sprite
5      dc.w      %0000000000000000,%0000110000110000
6      dc.w      %0000000000000000,%0000011001100000
7      ...
8      dc.w      0,0      ; fine dello sprite
```

Infatti il primo byte, VSTART, è a \$50, il secondo, HSTART, è a \$90, mentre il terzo, la posizione verticale di fine sprite, è a \$64, ossia a \$50+20, la posizione inizio più la lunghezza dello sprite. Il quarto byte per ora lo lasciamo a zero, vedremo in seguito a cosa serve. Posso premettere che il byte HSTART, ossia quello che si occupa della posizione orizzontale, fa spostare lo sprite a “scatti” di 2 pixel alla volta, per cui muovendo uno sprite dalla posizione \$50 alla posizione \$51, ad esempio, scatterebbe a destra di 2 pixel, e non di uno: vedremo che usando un bit del quarto byte si può far scorrere lo sprite di un pixel alla volta orizzontalmente.

Per quanto riguarda la posizione verticale, invece, lo scorrimento avviene già con VSTART/-VSTOP a scatti di un pixel, ma la limitazione è la linea video \$FF, oltre la quale si può andare usando un altro dei bit del quarto byte. Per ragioni di semplicità nei primi esempi sposteremo gli sprite solamente agendo sui byte HSTART, VSTART e VSTOP, ossia con le limitazioni di uno scorrimento orizzontale a “scatti” di due pixel alla volta. Solo in un secondo momento vedremo come fare scorrimenti più fluidi. Ricordatevi dunque della particolarità che, ad esempio, con un:

```
1  ADDQ.B #1,HSTART
```

spostiamo lo sprite di 2 pixel e non di uno. Per agire sui 3 byte VSTART, HSTART, VSTOP si potrebbe fare così:

```
1  MOVE.B #$50,SPRITE      ; VSTART = $50
2  MOVE.B #$90,SPRITE+1    ; HSTART = $90
3  MOVE.B #$64,SPRITE+2    ; VSTOP = $64 ($50+20)
```

Oppure si può definire una label per ogni byte per renderlo più chiaro:

```
1  SPRITE:
2  VSTART:      ; posizione inizio VERTICALE
3      dc.b $50
4  HSTART:      ; posizione inizio ORIZZONTALE
5      dc.b $90
6  VSTOP:
7      dc.b $64      ; posizione fine VERTICALE
```

```

8      dc.b $00      ; byte per funzioni speciali azzerato
9
10     ; da qua iniziano i dati dei 2 piani dello sprite
11
12     dc.w           %0000000000000000,%0000110000110000
13     dc.w           %0000000000000000,%0000011001100000
14     ...
15     dc.w           0,0 ; fine dello sprite

```

In questo caso agiremmo sulle label VSTART, HSTART e VSTOP:

```

1      ADDQ.B #1,HSTART ; sposta lo sprite a destra di 2 pixel
2      ; (2 pixel e non 1 per le ragioni descritte)
3      SUBQ.B #1,HSTART ; sposta lo sprite a sinistra di 2 pixel

```

Per spostare in basso o in alto lo sprite dovremmo però ricordarci di modificare sia VSTART che VSTOP, perché è ovvio che se spostiamo in basso o in alto lo sprite si sposta sia il primo pixel a sinistra che l'ultimo:

```

1      ADDQ.B #1, VSTART ; \ sposta lo sprite in basso di 1 pixel
2      ADDQ.B #1, VSTOP  ; /
3
4      SUBQ.B #1, VSTART ; \ sposta lo sprite in alto di 1 pixel
5      SUBQ.B #1, VSTOP  ; /

```

Ricapitolando questa è la struttura dello sprite:

```

prima word di controllo,      seconda word di controllo
prima linea (.w) del piano 1, prima linea (.w) del piano 2
seconda linea (.w) del piano 1, seconda linea (.w) del piano 2
terza linea (.w) del piano 1, terza linea (.w) del piano 2
quarta linea (.w) del piano 1, quarta linea (.w) del piano 2
quinta linea (.w) del piano 1, quinta linea (.w) del piano 2
...
dc.w      0,0 ; l'ultima riga deve contenere due zeri

```

I dati dello sprite sono divisi in piano 1 e piano 2 solo per indicare che la loro sovrapposizione determina i 3 colori più il trasparente in maniera analoga ai bitplanes dello schermo, ma non vanno confusi con questi ultimi!

7.1 I colori degli sprite

Per definire i colori degli sprite bisogna usare gli stessi registri colore usati dai bitplanes, in quanto l'Amiga ha solo 32 registri colore. I progettisti hanno pensato di far assumere agli sprites i colori dal 16 al 31, per cui se le figure non sono a 32 colori, ossia a 5 bitplanes, gli sprites possono avere colori diversi dalle figure. Altrimenti gli sprites avranno 16 colori in comune con la figura a 32 colori sottostante. Per ora vediamo come definire i colori del primo sprite (gli sprite sono numerati dallo 0 al 7):

```

COLORE 0 dello sprite 0 = TRASPARENZA, non va definito
COLORE 1 dello sprite 0 = COLOR17 ($dff1a2)
COLORE 2 dello sprite 0 = COLOR18 ($dff1a4)
COLORE 3 dello sprite 0 = COLOR19 ($dff1a6)

```

il colore 0, ossia il quarto, è la trasparenza e non occorre definirlo.

Vediamo, prima di procedere, il primo esempio di visualizzazione di uno sprite in Lezione 7a. s. In questo esempio viene puntato il primo sprite, lasciando azzerati gli altri 7. Per puntare uno sprite bisogna fare come per i bitplanes, in quanto lo sprite ha i puntatori che funzionano allo stesso modo:

```

1  move.l #MIOSPRITE,d0      ; indirizzo dello sprite in d0
2  lea SpritePointers,a1     ; Puntatori in copperlist
3  move.w d0,6(a1)
4  swap d0
5  move.w d0,2(a1)

```

Va ricordato che per visualizzare gli sprite occorre aver “acceso” almeno un bitplane, con i bitplane disabilitati vengono disabilitati anche gli sprite. Allo stesso modo, uno sprite viene “tagliato” se va oltre la finestra video, definita col DIWSTART e DIWSTOP, essendo visualizzabile solo al suo interno. Da notare che per posizionare nello schermo 320x256 lo sprite, per esempio alla coordinata centrale (160,128) bisogna tener conto che la prima coordinata in alto a sinistra, dove inizia la finestra video, non è (0,0), ma \$40,\$2c per cui bisogna sommare \$40 alla coordinata X e \$2c alla coordinata Y. Infatti \$40+160, \$2c+128, corrispondono alla coordinata (160,128) di uno schermo 320x256 non overscan.

Non avendo ancora il controllo della posizione orizzontale a livello di 1 pixel, ma ogni 2 pixel, dobbiamo sommare non 160, ma 160/2, per individuare il centro dello schermo:

```

1  HSTART:
2  dc.b $40+(160/2)      ; posizionato al centro dello schermo
3  ...

```

Ecco uno schema dello schermo, in cui la parte visibile, ossia la finestra video, è bianca, mentre l'intero schermo, fuori dai bordi, che inizia con le coordinate 0,0 è fatto di ####. Si noti che la finestra video comincia dalle coordinate \$40 XX e \$2c YY.

```

(0,0)\
      +-----+
      |#####|
      |#####|
      |###+-----+###|
      || |###| $40,$2c |###| -- Bordi dello schermo
      || |###| ----- |###| / visibile (finestra video)
      || |###| /Sprite\ |###|/
      || |###| |++XX++| |###|/
      || |###| \|/\|/\| |###|/
      |###| |###| |###|
ASSE Y |###| |###| |###|
      |###| |###| |###|
      || |###| |###|
      || |###| |###|
      || |###| |###|
      || |###| |###|
      |###+-----+###|
      |#####|
      |#####|
      +-----+
      <----- ASSE X ----->

```

La posizione *orizzontale* dello sprite può andare da 0 a 447, ma è chiaro che per essere visibile su schermo largo 320 pixel deve andare da 64 a 383. La posizione *verticale* dello sprite invece può andare da 0 a 262, ma per essere visibile su schermo largo PAL (256 linee) deve andare da 44 (\$2c) alla fine dello schermo, 44+256= 300 (\$12c). Per ora abbiamo raggiunto solo la posizione \$FF, vedremo più avanti come andare fino alla \$12c.

In Lezione7b.s lo sprite viene fatto scorrere sullo schermo con degli ADD e SUB sulle due word di controllo.

In Lezione7c.s, lo sprite viene spostato in orizzontale sullo schermo con delle tabelle di valori predefiniti anziché con ADD e SUB. In Lezione7d.s viene fatto saltellare in verticale. In

Lezione7e.s le due coordinate XX ed YY vengono definite da due tabelle per creare movimenti circolari, ad ellisse eccetera. In questo esempio viene anche spiegato come crearsi proprie tabelle!

Prima di procedere nella lettura caricate ed eseguite in altri buffer di testo questi esempi, leggendone i commenti finali.

Per ora abbiamo visualizzato un solo sprite, vediamo cosa occorre sapere se si visualizzano tutti e 8 gli sprite. Innanzitutto ogni sprite ha posizione indipendente rispetto agli altri, ed ha un VSTART, HSTART e VSTOP proprio nelle prime 2 word. Per quanto riguarda invece i colori (e anche altre proprietà degli sprite che vedremo successivamente, come per. es. le collisioni) gli sprite non sono totalmente indipendenti ma sono accoppiati a due a due. Ci sono dunque 4 coppie di due sprite: Sprite0+Sprite1, Sprite2+Sprite3, Sprite4+Sprite5, ed infine Sprite6+Sprite7. In tutto il resto della lezione, quando parleremo di *coppia di sprite* non intenderemo 2 sprite qualunque, ma una di queste 4 coppie.

Per i colori, bisogna tenere conto del fatto che gli sprite di una coppia hanno i colori in comune, ossia ogni coppia di sprite ha la sua palette (tavolozza) diversa da quella delle altre coppie. Sappiamo che i 3 colori dello sprite 0 sono definibili coi registri COLOR17, COLOR18 e COLOR19. Questi 3 colori valgono anche per lo sprite “fratello”, ossia lo sprite 1. Ogni coppia ha una palette colori diversa perché sono disponibili i registri colore dal 16 al 31, ossia 16 registri. Considerando che ogni sprite ha 4 colori (di cui 1 trasparente), servirebbero $8 \times 4 = 32$ registri, quando ne sono rimasti solo 16. Dunque, avendo 8 sprites con 4 colori ciascuno ecco da quali registri le coppie di sprite prendono i colori:

	Sprite	Valore binario	Registro di colore:
	-----	-----	-----
Coppia 1:	0 o 1	00	Non Usato perché trasparente
		01	Color17 - \$dff1a2
		10	Color18 - \$dff1a4
		11	Color19 - \$dff1a6
Coppia 2:	2 o 3	00	Non Usato perché trasparente
		01	Color21 - \$dff1aa
		10	Color22 - \$dff1ac
		11	Color23 - \$dff1ae
Coppia 3:	4 o 5	00	Non Usato perché trasparente
		01	Color25 - \$dff1b2
		10	Color26 - \$dff1b4
		11	Color27 - \$dff1b6
Coppia 4:	6 o 7	00	Non Usato perché trasparente
		01	Color29 - \$dff1ba
		10	Color30 - \$dff1bc
		11	Color31 - \$dff1be

Facciamo un esempio pratico: nella copperlist per definire il colore degli 8 sprite è necessario fare questo:

```

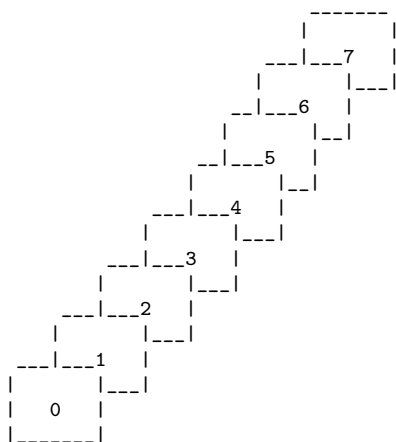
1  dc.w $1A2,$F00 ; color17, - COLOR1 degli sprite0/1 -ROSSO
2  dc.w $1A4,$0F0 ; color18, - COLOR2 degli sprite0/1 -VERDE
3  dc.w $1A6,$FF0 ; color19, - COLOR3 degli sprite0/1 -GIALLO
4
5  dc.w $1AA,$FFF ; color21, - COLOR1 degli sprite2/3 -BIANCO
6  dc.w $1AC,$0BD ; color22, - COLOR2 degli sprite2/3 -ACQUA
7  dc.w $1AE,$D50 ; color23, - COLOR3 degli sprite2/3 -ARANCIO
8
9  dc.w $1B2,$00F ; color25, - COLOR1 degli sprite4/5 -BLU
10 dc.w $1B4,$F0F ; color26, - COLOR2 degli sprite4/5 -VIOLA
11 dc.w $1B6,$BBB ; color27, - COLOR3 degli sprite4/5 -GRIGIO
12
13 dc.w $1BA,$8E0 ; color29, - COLOR1 degli sprite6/7 -VERDE CH.
14 dc.w $1BC,$a70 ; color30, - COLOR2 degli sprite6/7 -MARRONE
15 dc.w $1BE,$d00 ; color31, - COLOR3 degli sprite6/7 -ROSSO SC.

```


NOTA: Se impostate una figura a 2,4,8 o 16 colori come sottofondo, non ci sono problemi per la palette, ma se decidete di attivare uno schermo a 32 colori, ossia 5 bitplanes, la figura avrà in comune gli ultimi colori con gli sprite, per cui dovete fare in modo che i colori siano giusti sia per la figura che per lo sprite, che insomma il colore sia “multiuso”.

7.2 La priorità video tra gli sprite

Quando ci sono due o più sprite sullo schermo può avvenire che degli sprite si sovrappongano. In questo caso viene coperto lo sprite con la priorità minore. La priorità tra gli sprite è sempre uguale, lo sprite con numero minore ha sempre priorità su quelli con numero maggiore, i quali rimangono “dietro”. Di conseguenza lo sprite 0 può coprire tutti gli altri sprite, mentre lo sprite 7 può essere coperto da tutti gli altri. Ecco uno schemino:



Verifichiamo caricando ed eseguendo in un altro buffer di testo `Lezione7f.s`, il quale visualizza 8 sprite, e dopo la pressione del tasto sinistro del mouse li sovrappone per evidenziare le priorità. Tasto destro del mouse per uscire.

7.3 Sprite “attached”

Esiste anche una modalità di accoppiamento degli sprite a 2 a 2, l’uno sull’altro, che riduce il numero degli sprite disponibili alla metà, ossia a quattro, ma con 16 colori ciascuno anziché 4. (15 colori più il trasparente) Possono essere combinati solo in questo modo:

```

SPRITE0+SPRITE1    - Sprite ATTACCHED (attaccato) Numero 1
SPRITE2+SPRITE3    - Sprite ATTACCHED (attaccato) Numero 2
SPRITE4+SPRITE5    - Sprite ATTACCHED (attaccato) Numero 3
SPRITE6+SPRITE7    - Sprite ATTACCHED (attaccato) Numero 4

```

In pratica si attaccano gli sprite che in modalità normale fanno già coppia perché hanno la stessa palette. I 4 sprite “attaccati” condividono la stessa palette di 16 colori, dato che sono disponibili solo i registri colore dal Color16 al Color31. Gli sprite *attached* funzionano in questo modo: normalmente uno sprite ha al massimo 4 possibilità di sovrapposizione per i suoi piccoli “bitplanes”, ossia %00 per il trasparente e %01,%10,%11 per gli altri 3 colori. La modalità *attached* fa sovrapporre i piani di bit dei due sprites per formare 16 possibilità, infatti ponendo i due piani del primo sopra i 2 piani del secondo si possono ottenere %1111 possibilità anziché

%11, ossia 16 anziché 4. Nella tabella che segue, nella colonna “valore binario”, sono elencate le varie possibilità di sovrapposizione e il relativo colore che ne risulta.

Colore Sprite	Valore binario	Numero del registro colore
0	0000	Color16 - NON USATO, è IL TRASPARENTE
1	0001	Color17 - \$dff1a2
2	0010	Color18 - \$dff1a4
3	0011	Color19 - \$dff1a6
4	0100	Color20 - \$dff1a8
5	0101	Color21 - \$dff1aa
6	0110	Color22 - \$dff1ac
7	0111	Color23 - \$dff1ae
8	1000	Color24 - \$dff1b0
9	1001	Color25 - \$dff1b2
10	1010	Color26 - \$dff1b4
11	1011	Color27 - \$dff1b6
12	1100	Color28 - \$dff1b8
13	1101	Color29 - \$dff1ba
14	1110	Color30 - \$dff1bc
15	1111	Color31 - \$dff1be

Dunque in COPPERLIST bisogna definirli in questo modo:

```

1 dc.w $1A2,$F00 ; color17, COLORE 1 per gli sprite attaccati
2 dc.w $1A4,$0F0 ; color18, COLORE 2 per gli sprite attaccati
3 dc.w $1A6,$FF0 ; color19, COLORE 3 per gli sprite attaccati
4 dc.w $1A8,$FF0 ; color20, COLORE 4 per gli sprite attaccati
5 dc.w $1AA,$FFF ; color21, COLORE 5 per gli sprite attaccati
6 dc.w $1AC,$0BD ; color22, COLORE 6 per gli sprite attaccati
7 dc.w $1AE,$D50 ; color23, COLORE 7 per gli sprite attaccati
8 dc.w $1B0,$D50 ; color24, COLORE 7 per gli sprite attaccati
9 dc.w $1B2,$00F ; color25, COLORE 9 per gli sprite attaccati
10 dc.w $1B4,$F0F ; color26, COLORE 10 per gli sprite attaccati
11 dc.w $1B6,$BBB ; color27, COLORE 11 per gli sprite attaccati
12 dc.w $1B8,$BBB ; color28, COLORE 12 per gli sprite attaccati
13 dc.w $1BA,$8E0 ; color29, COLORE 13 per gli sprite attaccati
14 dc.w $1BC,$a70 ; color30, COLORE 14 per gli sprite attaccati
15 dc.w $1BE,$d00 ; color31, COLORE 15 per gli sprite attaccati

```

Per “attaccare” due sprite basta porre ad 1 il bit 7 della seconda word di controllo dello sprite dispari della coppia (ossia del famigerato quarto byte delle funzioni speciali). Per esempio per attaccare gli sprite 0 ed 1 basta settare tale bit allo sprite 1, per attaccare lo sprite 4 ed il 5 basta settarlo al 5. è ovvio che gli sprite attaccati devono avere le stesse coordinate, ossia essere l'uno sopra l'altro, per consentire la giusta sovrapposizione dei 4 piani. Facciamo un esempio: per attaccare gli sprite 0 ed 1 bisogna porre ad 1 il bit 7 del quarto byte dello sprite 1:

```

1 SPRITE0:
2 VSTART0: ; posizione inizio VERTICALE
3 dc.b $50
4 HSTART0: ; posizione inizio ORIZZONTALE
5 dc.b $90
6 VSTOP0:
7 dc.b $64 ; posizione fine VERTICALE
8 dc.b $00 ; non occorre settare il bit 7 agli sprite pari.
9 ; da qua iniziano i dati dei 2 piani dello sprite
10 dc.w %0000000000000000,%0000110000110000
11 dc.w %0000000000000000,%00001100110000
12 ...
13 dc.w 0,0 ; fine sprite0
14
15 SPRITE1:
16 VSTART1: ; posizione inizio VERTICALE
17 dc.b $50
18 HSTART: ; posizione inizio ORIZZONTALE

```

```

19      dc.b $90
20 VSTOP:
21      dc.b $64          ; posizione fine VERTICALE
22
23      ;76543210
24      dc.b %10000000 ; BIT 7 SETTATO! ATTACHED MODE per sprite 0/1
25
26 ; da qua iniziano i dati dei 2 piani dello sprite
27      dc.w %0000000000000000,%0000110000110000
28      dc.w %0000000000000000,%0000011001100000
29      ...
30      dc.w 0,0          ; fine sprite1

```

Dunque per far sì che tutti gli sprite siano in modo "attached" basta porre ad 1 i bit 7 del quarto byte degli sprite 1,3,5 e 7, cioè di quelli dispari.

Per farsi uno sprite a 16 colori è necessario disegnarlo con un programma da disegno e convertirlo in formato SPRITE con l'iffconverter KEFCON, infatti è difficile "calcolarsi" ad occhio i colori risultanti da 4 piani di bit, divisi in due sprite!

Caricatevi ed eseguitevi il listato Lezione7g.s, che visualizza uno sprite a 16 colori in modalità attached, in cui è anche descritto come convertirsi uno sprite con il KEFCON, sia a 4 colori che a 16.

È possibile visualizzare contemporaneamente sprite a 16 colori e sprite a 4 colori, per esempio gli sprite 0 ed 1 "attaccati" e gli altri no, o qualsiasi altra combinazione.

Nel listato esempio Lezione7h.s sono visualizzati i 4 sprite attached a 16 colori, ognuno con un movimento indipendente dagli altri.

A questo punto vi starete chiedendo come mai non è ancora stato eliminato l'inconveniente dello scorrimento orizzontale scattoso a passi di 2 pixel alla volta anziché uno. Ebbene, è giunta l'ora di risolvere il problema, ma per fare ciò è necessario imparare una nuova istruzione del 68000, la quale opera sui singoli bit di un numero: LSR.

Questa istruzione significa *Logic Shift Right*, cioè *scorrimento logico dei bit a destra*, in altre parole, se un numero binario in d0 è %00111, dopo un bel LSR #1, d0 il risultato è %00011, dopo un LSR #2, d0 è %00001. Allo stesso modo, un %00110010 dopo un LSR #1, d0 diventa %00011001, mentre dopo un LSR #5, d0 diventa %00000001. Dunque il numero, considerato nella sua forma binaria, viene spostato a destra come se i bit fossero su una tovaglia che tiriamo: tirando di #1 si sposta la tovaglia con tutti i BitPiatti sopra e il primo BitPiatto cade in terra. . . tirando troppo si può spostare tutto facendo cadere tutto in terra e azzerando il tavolo.

Ma cosa c'entra questa istruzione assembler con il byte HSTART??? Il problema sta in questi termini: come sapete le posizioni orizzontali possibili sono ben più di \$FF (255), per il solo fatto che lo schermo è largo 320 pixel. Per indicare un numero superiore a 255 (8 bit, dallo zero al sette), occorre aggiungere almeno un altro bit, il nono, detto bit 8, in tal modo anziché un massimo di %11111111 (\$ff) si può avere un massimo di %111111111, ossia 511, che per l'HSTART va benissimo. Ma dove mettere questo bit?? Quei mattacchioni dei progettisti hanno pensato bene di metterlo nel famigerato quarto byte di controllo, quello che abbiamo già visto per attaccare gli sprite (il bit 7 di tale byte infatti serve ad attaccare gli sprite per farli a 16 colori).

Avendo altri 6 bit liberi per usi vari, decisero di usare il bit 0 come bit *basso* della coordinata a 9 bit della coordinata orizzontale, spezzando il numero a 9 bit in questo modo:

```

; numero a 9 bit, che rappresenta la
; coordinata HSTART
;876543210
%111111111
 \-----/ \ /
  |         |
8 bit alti |
messi nel  |

```


controllo, infatti dopo \$ff viene \$100, \$101 eccetera, dunque il byte basso riparte da zero, ma col nono bit settato. Vediamo come fare una routine analoga a quella vista per la posizione orizzontale, ossia che parte dalla coordinata reale (è necessaria una word) e la “divide” in bit alto e byte basso. Da ricordare che in questo caso abbiamo da aggiornare anche VSTOP oltre a VSTART ogni volta!!! Teniamo presente che il bit alto di VSTOP è il bit 1 del quarto byte di controllo, mentre quello di VSTART è il bit 2:

```

1      MOVE.w (A0),d0      ; copia la word dalla tabella in d0
2      ADD.W  #$2c,d0      ; aggiungi l'offset dell'inizio dello schermo
3      MOVE.b d0,VSTART    ; copia il byte in VSTART
4      btst.l #8,d0        ; numero maggiore di $FF?
5      beq.s  NonVSTARTSET
6      bset.b #2,MIOSPRITE+3 ; Setta il bit 8 di VSTART (numero > $FF)
7      bra.s  ToVSTOP
8 NonVSTARTSET:
9      bclr.b #2,MIOSPRITE+3 ; Azzera il bit 8 di VSTART (numero < $FF)
10 ToVSTOP:
11      ADD.w  #13,D0       ; Aggiungi la lunghezza dello sprite per
12                        ; determinare la posizione finale (VSTOP)
13      move.b d0,VSTOP     ; Muovi il valore giusto in VSTOP
14      btst.l #8,d0
15      beq.s  NonVSTOPSET
16      bset.b #1,MIOSPRITE+3 ; Setta il bit 8 di VSTOP (numero > $FF)
17      bra.w  VstopFIN
18 NonVSTOPSET:
19      bclr.b #1,MIOSPRITE+3 ; Azzera il bit 8 di VSTOP (numero < $FF)
20 VstopFIN:
21      rts

```

Questa routine funziona in maniera analoga alla precedente per il settaggio del bit “staccato”, mentre si differenzia per il fatto che deve agire sia su VSTART che su VSTOP, e per l'assenza dell'LSR, qua inutile. Potete provarla in pratica caricando la Lezione71.s.

Ora che abbiamo il completo controllo sugli sprite, vediamo di ottimizzare le routine con le quali li controlliamo: innanzitutto, la prima cosa da fare è quella di fare una routine universale di controllo degli sprite, in modo da non dover riscrivere per ognuno degli 8 sprite la parte della sistemazione del bit “staccato”. Serve una routine parametrica, la quale richieda in entrata l'indirizzo dello sprite interessato e la coordinata X ed Y che deve assumere, in questo modo basterà eseguire un BSR Routine per ogni sprite anziché riscrivere tutto. Potremo così riutilizzare tale routine ogni volta che vogliamo programmare gli sprite, al massimo con piccole modifiche. Un esempio di routine del genere la troviamo nella lezione7m.s.

La routine universale si chiama UniMuoviSprite, e per funzionare è necessario che le vengano indicate oltre all'indirizzo dello sprite da muovere e alle nuove coordinate che deve assumere, anche l'altezza dello sprite, che serve alla routine per calcolare il valore del byte VSTOP. Questi valori vengono comunicati o meglio “passati” alla routine mettendoli in alcuni registri prima di eseguire la routine.

Più precisamente si deve mettere l'indirizzo dello sprite nel registro a1, la sua altezza nel registro d2, la coordinata Y nel registro d0 e la coordinata X nel registro d1. La coordinate dello sprite “passate” alla routine sono i valori nello schermo 320x256. Infatti la routine si occupa di “centrare” lo sprite sullo schermo sommando \$40 alla coordinata X e \$2c alla coordinata Y. Inoltre pensa a mettere a posto il bit basso di HSTART e i bit alti di VSTART e VSTOP.

Brevemente:

```

1 ;
2 ;      Parametri in entrata di UniMuoviSprite:
3 ;
4 ;      a1 = Indirizzo dello sprite
5 ;      d0 = posizione verticale Y dello sprite sullo schermo (0-255)
6 ;      d1 = posizione orizzontale X dello sprite sullo schermo (0-320)
7 ;      d2 = altezza dello sprite
8 ;

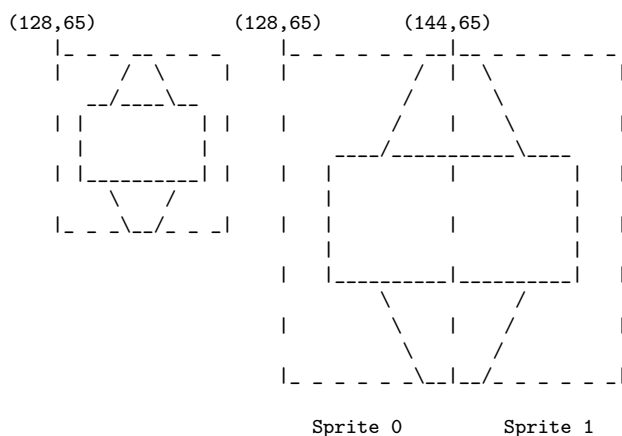
```

Avendo a disposizione questa routine che ci risolve una volta per tutte i problemi relativi al posizionamento degli sprite, possiamo divertirci a usarla per qualche applicazione che ci consentirà di fare un po' di esperienza con gli sprite. Prima di proseguire però caricate ed eseguite la `Lezione7m.s`, e guai a voi se continuate a leggere la `lezione7.txt` o a caricare listati prima di averla capita **completamente**. Dato che sarà usata in tutti gli altri esempi sugli sprite, sarebbe poco produttivo continuare senza aver capito una routine che trovate continuamente.

Nella `lezione7n.s` vediamo uno sprite che si muove sullo schermo seguendo traiettorie rettilinee. Le posizioni dello sprite non sono contenute in un tabella, ma vengono calcolate di volta in volta facendo muovere lo sprite con velocità costante. Caricatelo ed eseguitelo, vedremo anche come far rimbalzare uno sprite contro i bordi dello schermo.

Nella `lezione7o.s` vedremo invece due sprite che vengono entrambi mossi dalla routine universale. È un ottimo esempio di come grazie all'utilizzo dei parametri la nostra routine universale sia in grado di muovere senza nessuna modifica sprite che hanno forma e dimensioni diverse. Se non usassimo i parametri dovremmo scrivere una routine per ciascun sprite, sprecando tempo per farlo e memoria sul computer (con 8 sprite dovremmo scrivere 8 routine).

Nella `lezione7p.s` sempre usando la routine universale vediamo come si possano creare degli oggetti larghi più di 16 pixel utilizzando degli sprite affiancati. State *bene attenti* a non confondere gli sprite "attaccati" con quelli "affiancati": i primi sono 2 sprite della stessa coppia usati in modalità "attached", hanno le stesse coordinate (sono perfettamente sovrapposti) e lo sprite dispari ha il bit "attach" settato a 1; per sprite "affiancati" si intende invece un insieme di due o più sprite che vengono posizionati sullo schermo uno di fianco all'altro senza lasciare neppure una colonna di pixel tra l'uno e l'altro, in maniera da sembrare un unico oggetto largo più di 16 pixel, dato che sono mossi contemporaneamente. Non c'è nessun bit da settare per gli sprite affiancati, non si tratta di un "modo" speciale degli sprite, ma solo di una particolare disposizione sullo schermo di normalissimi sprite. Ecco uno schemino che mostra un'astronave fatta da un solo sprite, e un'altra fatta da due sprites:



Con una tecnica del genere si possono fare mostri di fine livello larghi fino a 128 pixel (16×8) se fatti con sprite a 3 colori, oppure larghi fino a 64 pixel (16×4) se fatti con sprite attached a 15 colori. Se il mostro in questione è più alto che largo, ad esempio di forma umana, si potrebbe sfruttare tutta la lunghezza dello schermo, dato che non ci sono limiti per l'altezza di uno sprite, e si potrebbe cambiare la palette verticalmente col copper per colorargli, ad esempio, le scarpe con un colore diverso dai jeans.

7.4 Mouse e joystick

Ora che abbiamo visto come far muovere gli sprite all'Amiga, perché non impariamo muoverli noi? Naturalmente con l'aiuto di un joystick o di un mouse!

Prima di vedere come si usano questi dispositivi è necessario imparare delle nuove istruzioni assembler, che riguardano la manipolazione dei bit di un registro e che si chiamano NOT, AND, OR, EOR. Queste istruzioni lavorano sui singoli bit di un registro (o di una locazione di memoria), sia per il registro sorgente che per quello destinazione. Ad esempio queste istruzioni considerano un byte non come un numero formato da 8 bit (cifre binarie) ma come un insieme di 8 bit indipendenti tra di loro. In pratica questo vuol dire che l'effetto che l'istruzione ha su un singolo bit del registro è indipendente da quello che succede agli altri bit del registro.

Per prima vediamo la NOT. Essa funziona su un solo operando, e il suo effetto è quello di rovesciare i bit dell'operando, cioè di scambiare 1 con 0 e 0 con 1. Se ad esempio nel registro d0 abbiamo il numero %01001100, se facciamo

```
1 NOT.B d0
```

il risultato sarà %10110011.

Le altre 3 istruzioni, invece lavorano con 2 operandi, uno sorgente e l'altro destinazione, fanno un'operazione tra i contenuti degli operandi e mettono il risultato nell'operando destinazione. Le operazioni (che sono ovviamente diverse per ogni istruzione) sono bit-a-bit, cioè avvengono tra ogni bit dell'operando sorgente e il corrispondente bit dell'operando destinazione, nel quale inoltre viene poi messo il risultato. Quindi fare D0 AND D1 in pratica significa fare:

```
(bit 0 di D0) AND (bit 0 di D1)
(bit 1 di D0) AND (bit 1 di D1)
(bit 2 di D0) AND (bit 2 di D1) e così via per tutti i bit di D0 e D1
```

Vediamo dunque come funziona l'AND tra 2 bit. Poiché un bit vale 0 o 1, ci sono 4 possibili casi:

```
0 AND 0 = 0
0 AND 1 = 0
1 AND 0 = 0
1 AND 1 = 1
```

AND dà come risultato 1 soltanto quando sia il bit del primo operando che quello del secondo operando sono ad 1. Infatti AND in inglese significa "e", quindi dà come risultato 1 se il primo e il secondo bit sono a 1. Si potrebbe tradurre con: *sono ad 1 sia il primo che il secondo bit? Se sì, rispondo con 1, se no invece rispondo con uno zero*. Un AND può essere utile ad azzerare certi bit di un numero:

```
1 AND.W #%1111111111111011,LABEL
```

Azzererà il bit 2 del numero in LABEL, perché è l'unico che è azzerato nell'operando, e l'unico che sarà cambiato nella destinazione, infatti tutti gli altri sono ad 1, quindi questi non cambiano la destinazione. Se il bit di destinazione è 0, facendo un 1 AND 0 il risultato rimane 0, allo stesso modo se è 1, facendo un 1 AND 1 il risultato rimane 1. Per quanto riguarda il bit che è a 0, invece, condanna la destinazione ad essere 0, infatti per dare un 1 di destinazione entrambi gli operandi devono essere 1, in questo caso essendo a 0 il primo, sia che il secondo sia 0 o 1 il risultato sarà 0. alcuni esempi:

```
1111001111 AND 0011001100 = 0011001100 - Nessuna modifica
1101011011 AND 0001110001 = 0001010001 - 1 bit azzerato
1111101101 AND 0011111111 = 0011101101 - 2 bit azzerati
```

Questa operazione di azzeramento si dice *mascheratura*:

```

1 AND #%11110000,LABEL  (%11110000 è la maschera, infatti è come se si
2                         mettesse una maschera di ZERI sopra il numero
3                         in LABEL, in questo caso è come se "tappassimo"
4                         i primi 4 bit come si "tappa" un neo una ragazza
5                         quando si mette il fondotinta. Il neo è un 1 che
6                         si trovava nella posizione della maschera dove
7                         c'erano degli 0, e il neo che viene "coperto"
8                         dal trucco, ossia viene azzerato).
```

L'OR invece si comporta in questo modo:

```

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

In questo caso basta che 1 dei 2 bit sia ad 1 per dare risultato 1. Dunque il risultato è sempre 1 tranne quando entrambi i bit sono a zero. Anche qui sapere che OR in inglese significa "o" ci aiuta a ricordare che il risultato è 1 se il primo O il secondo bit sono a 1. Si potrebbe tradurre in *o uno o l'altro bit devono essere ad 1 per dare 1*. Questo comando è utile, all'opposto dell'AND, per *settare* dei bit, per porli cioè ad 1: alcuni esempi:

```

0000000001 OR 1101011101 = 1101010001 - Nessun cambiamento
1000000000 OR 0010011000 = 1010011000 - 1 bit settato
0001111000 OR 1111100000 = 1111111000 - 2 bit settati
```

In questo caso, è come se la ragazza di prima, anziché mettersi il fondotinta rosaceo (gli 0) perappare i nei neri (ossia gli 1), si mettesse del nero per farsi dei nei falsi, come quello che aveva Marilyn Monroe sopra il labbro. Oppure come se fosse una ragazza di colore (ossia tutta ad 1) che si è truccata con il rosa per sembrare bianca (come Michael Jackson), ossia per essere tutta a zero, che si toglie il fondotinta dove il numero dell'OR è ad uno, scoprendo il nero.

Invece il comando EOR, ovvero OR esclusivo, setta il bit solo quando è ad 1 o il primo o il secondo bit, non quando sono ad 1 entrambi, come invece fa il comando OR:

```

0 EOR 0 = 0
0 EOR 1 = 1
1 EOR 0 = 1
1 EOR 1 = 0 ; Questa è la differenza con l'OR! infatti 1 OR 1 = 1.
```

Alcuni esempi:

```

0000000001 EOR 1101011101 = 1101010000 - 1 bit azzerato
1000000000 EOR 0010011000 = 1010011000 - 1 bit settato
```

Quest'ultima istruzione ci sarà utile per leggere il joystick. Come sapete l'Amiga ha 2 porte usate per collegare joystick o mouse. Ad ognuna di queste porte si può collegare indifferente-mente un joystick o un mouse. Per ogni porta esiste un registro hardware che si può leggere per sapere se e in che modo sono mossi joystick e mouse. La porta 0 (dove di solito è collegato il mouse) viene letta attraverso il registro JOY0DAT (\$dff00a) mentre la porta 1 attraverso JOY1DAT (\$dff00c).

Per prima cosa vediamo come leggere il joystick. Ci riferiremo al registro JOY1DAT che è quello usato di solito, ma JOY0DAT funziona esattamente allo stesso modo quando ci colleghiamo un joystick. Possiamo pensare ad un joystick come ad un insieme di 4 interruttori (uno per ogni direzione), ognuno dei quali può assumere 2 stati: chiuso (1) o aperto (0) a seconda che la leva del joystick sia premuta o meno nella direzione associata all'interruttore. Per sapere in quali direzioni è mosso il joystick dobbiamo conoscere gli stati degli interruttori. Per 2 di questi interruttori è molto semplice, in quanto il loro stato è riportato in un bit del registro JOY1DAT:

- il bit 1 di JOY1DAT è lo stato dell'interruttore "destra"
- il bit 9 di JOY1DAT è lo stato dell'interruttore "sinistra".

Se un bit vale 1 l'interruttore associato è chiuso, altrimenti è aperto. Per quanto riguarda le altre 2 direzioni lo stato non è mappato direttamente in un bit, ma deve essere ottenuto mediante il calcolo di un'operazione, precisamente dell'EOR che abbiamo spiegato poco fa, effettuata tra 2 bit del registro JOY1DAT:

- lo stato dell'interruttore "alto" è il risultato di un EOR tra il bit 8 e il bit 9
- lo stato dell'interruttore "basso" è il risultato di un EOR tra il bit 0 e il bit 1.

Anche in questo caso se un bit vale 1 l'interruttore associato è chiuso, altrimenti è aperto. Conoscendo gli stati dei 4 interruttori possiamo dunque usare il joystick per muovere uno sprite sullo schermo.

Caricate in un altro buffer di testo la lezione7q.s ed eseguirla. Veniamo ora al mouse. Quando colleghiamo un mouse ad una delle porte, il registro corrispondente si comporta in maniera diversa che nel caso del joystick. Infatti prendendo il registro JOYODAT (ma è lo stesso per l'1), troviamo che il byte alto è usato per rilevare gli spostamenti in direzione verticale e quello basso quelli in direzione orizzontale. Ogni byte rappresenta un numero (da 0 a 255) che varia secondo i movimenti del mouse.

- il byte alto diminuisce ogni volta che il mouse viene spostato verso l'alto e aumenta ogni volta che il mouse viene spostato verso il basso.
- il byte basso diminuisce ogni volta che il mouse viene spostato verso sinistra e aumenta ogni volta che il mouse viene spostato verso destra.

Vediamo come usare queste informazioni per muovere uno sprite con il mouse.

Il primo metodo che viene in mente è di usare i 2 byte di JOYODAT come coordinate per lo sprite, visto che anche le coordinate dello sprite diminuiscono se esso va in alto o a sinistra e aumentano se va in basso o a destra. Questo metodo ha l'inconveniente che in un byte possiamo raggiungere il valore 255, quindi i valori che possiamo leggere dal byte di JOYODAT dedicato alla direzione orizzontale possono arrivare al massimo a 255, mentre le coordinate orizzontali di uno sprite possono arrivare oltre 320. Caricate Lezione7r1.s e verificate questo metodo.

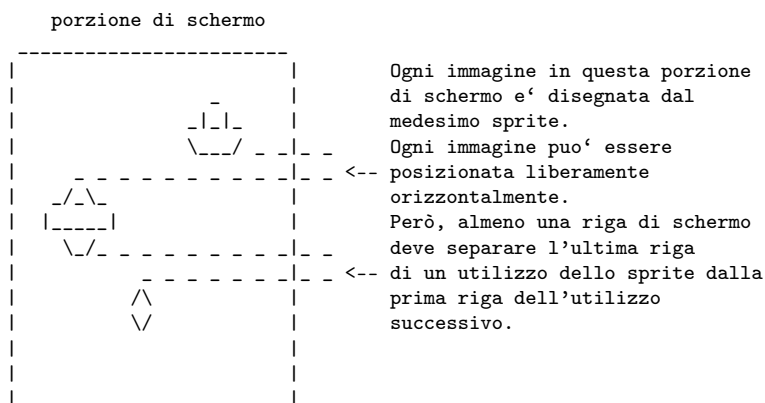
Un metodo un po' più complesso che però risolve il problema della limitazione in senso orizzontale a 255 pixel anziché 320 è presentato in Lezione7r2.s. Per una spiegazione del metodo leggetevi il commento alla fine del listato.

Sapendo come muovere una freccia sullo schermo, si può facilmente simulare il sistema intuition, ossia si può fare un pannello di controllo con dei bottoni disegnati da attivare spostandoci la freccia (lo sprite) sopra e premendo il pulsante, sia esso del joystick o del mouse. Basta controllare al momento della pressione del bottone in quale coordinata si trova la freccia, e se si trova sopra un bottone attivare l'opzione di quel bottone. Fare questo è piuttosto facile, provate da voi a farlo. Comunque in lezioni più avanzate del corso ci sarà un listato di questo tipo.

7.5 Riutilizzo degli sprite

Il riutilizzo degli sprite è una tecnica che ci consente di visualizzare più di 8 sprite contemporaneamente. In pratica uno stesso sprite viene usato per disegnare diversi oggetti situati a diverse altezze. Se ad esempio utilizziamo uno sprite per visualizzare un alieno nella parte alta dello

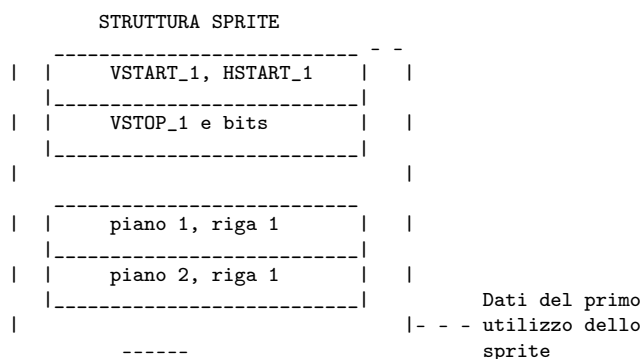
schermo, possiamo poi utilizzare di nuovo lo stesso sprite per disegnare l'astronave del giocatore nella parte bassa dello schermo. L'unica limitazione che si ha quando si riutilizzano gli sprite è che 2 oggetti disegnati da uno stesso sprite devono essere posizionati ad altezze differenti. Non è possibile visualizzare su una stessa riga dello schermo 2 righe che compongono 2 oggetti disegnati con il medesimo sprite. Per di più l'ultima riga della figura disegnata durante un utilizzo e la prima riga della figura disegnata con l'utilizzo successivo dello stesso sprite *devono* essere separate da almeno una riga nella quale lo sprite non è utilizzato. La figura seguente illustra meglio la situazione:

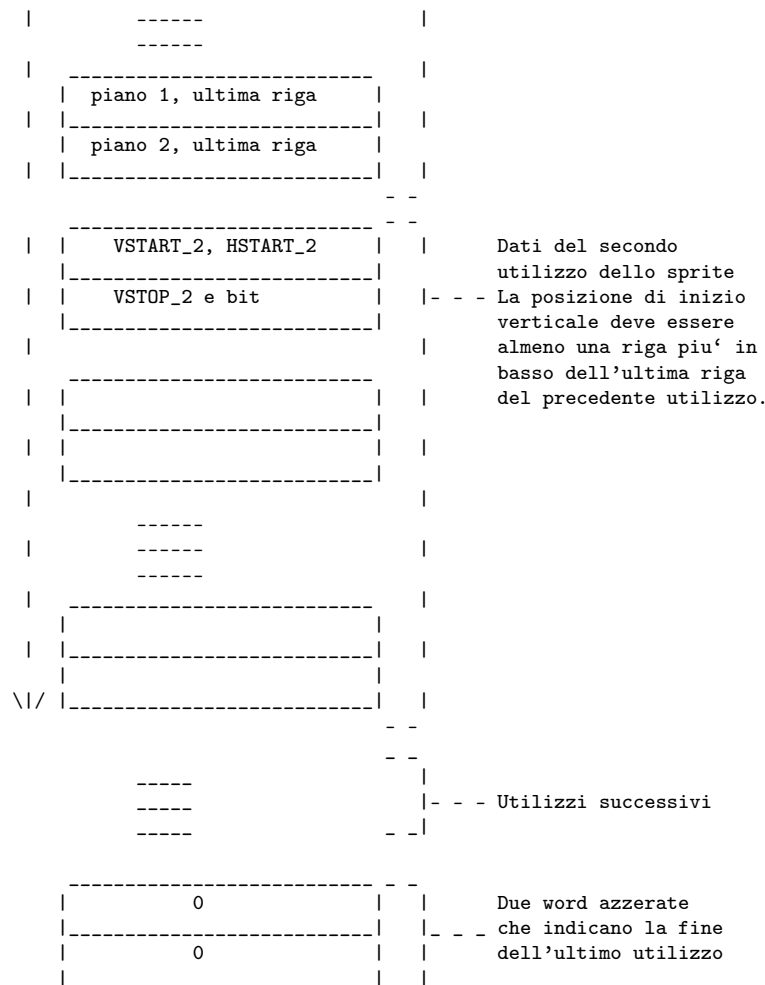


Non c'è nessuna limitazione invece per quanto riguarda le posizioni orizzontali, né per quanto riguarda figure disegnate mediante sprites diversi. Uno sprite può essere riutilizzato un numero qualunque di volte, ogni volta ad un'altezza diversa. Questa tecnica si può applicare ad ogni sprite, e in modo indipendente tra uno sprite e l'altro: per esempio si possono utilizzare 1 volta sola gli sprite 0,3 e 4, utilizzare 3 volte lo sprite 1, quattro volte lo sprite 2 e non utilizzare affatto gli sprite 5,6 e 7.

Applicare questa tecnica è molto semplice, in quanto richiede solo una modifica della struttura dati dello sprite.

Normalmente, alla fine della struttura dello sprite, dopo tutti i dati che descrivono la forma ci sono 2 word di valore 0 che appunto indicano la fine della struttura. Per riutilizzare uno sprite, al posto di queste 2 word ci mettiamo un'altra struttura sprite, che descrive un'altra figura da disegnare sullo schermo più in basso della prima. Se si vuole riutilizzare per una terza volta lo sprite, si mette una terza struttura sprite subito dopo la seconda, e lo stesso si fa per tutti i riutilizzi che si vuole. Dopo la struttura dati dell'ultimo utilizzo si mettono le 2 word di valore 0 che indicano la fine dell'ultimo utilizzo.





Da notare che i vari utilizzi verticali devono essere messi nella struttura in ordine da quello più in alto a quello più in basso. Per cui il byte VSTART di ogni utilizzo deve essere *maggiore* del byte VSTOP dell'utilizzo precedente dello sprite.

Vediamo un esempio pratico di struttura in cui uno sprite è riutilizzato 2 volte:

```

1 MIOSPRITE:
2 VSTART_1:
3     dc.b $50                                ; posizione primo utilizzo
4 HSTART_1:
5     dc.b $40+12
6 VSTOP_1:
7     dc.b $58
8     dc.b $00
9     dc.w %0000001111000000,%0111110000111110    ; dati "forma" del primo
10    dc.w %0000111111110000,%1111001110001111    ; utilizzo
11    dc.w %0011111111111100,%1100010001000011
12    dc.w %0111111111111110,%1000010001000001
13    dc.w %0111111111111110,%1000010001000001
14    dc.w %0011111111111100,%1100010001000011
15    dc.w %0000111111110000,%1111001110001111
16    dc.w %0000001111000000,%0111110000111110
17 VSTART_2:                                ; posizione utilizzo 2
18     dc.b $70                                ; NOTATE CHE VSTART_2 > VSTOP_1
19 HSTART_2:

```


Il numero massimo di bit-planes che ognuno dei 2 playfield può avere è 3 bit-plane in LOW-RES e 2 bit-plane in HI-RES. In pratica i 6 bit-planes dell'Amiga vengono ripartiti in due gruppi da 3, e ogni gruppo costituisce un playfield. Il playfield 1 è costituito dai bit-plane dispari, cioè i bit-plane 1, 3 e 5. Il playfield 2 è costituito dai bit-plane pari, cioè 2, 4 e 6.

Naturalmente non è sempre necessario usare tutti i bit-plane disponibili. Non possiamo però assegnare indipendentemente ai 2 playfield i bit-plane che vogliamo. Infatti il numero di bit-planes da usare si indica esattamente nello stesso modo che per i modi grafici "normali". Nei bit 14-12 del registro BPLCON0 (\$dff100), chiamati bit BPU2, BPU1 e BPU0 viene indicato il numero complessivo di bit-plane da attivare nei 2 playfield. In base al numero complessivo che noi indichiamo nei bit BPU, l'hardware assegna i bit plane secondo la seguente tabella:

Numero di bitplanes usati (bit BPU di BPLCON0)	Bit-planes al Playfield 1	Bit-planes al Playfield 2
0	nessuno	nessuno
1	plane 1	nessuno
2	plane 1	plane 2
3	plane 1,3	plane 2
4	plane 1,3	plane 2,4
5	plane 1,3,5	plane 2,4
6	plane 1,3,5	plane 2,4,6

Come potete vedere in pratica il playfield 1 ha sempre più planes del playfield 2, ed inoltre, il playfield 2 ha al massimo un plane in meno del playfield 1; non è possibile assegnare 3 plane al playfield 1 e un solo plane al playfield 2.

Analogamente ai modi grafici standard, la sovrapposizione dei bit-plane determina il colore usato per rappresentare ogni pixel sul video. Però la corrispondenza tra combinazioni dei bit-plane e registri colore è un po' differente, ed illustrata nelle 2 tabelle seguenti:

PLAYFIELD 1			
Valore plane 5	Valore plane 3	Valore plane 1	Colore selezionato
0	0	0	trasparente
0	0	1	COLOR01
0	1	0	COLOR02
0	1	1	COLOR03
1	0	0	COLOR04
1	0	1	COLOR05
1	1	0	COLOR06
1	1	1	COLOR07

PLAYFIELD 2			
Valore	Valore	Valore	Colore
plane 6	plane 4	plane 2	selezionato
0	0	0	trasparente
0	0	1	COLOR09
0	1	0	COLOR10
0	1	1	COLOR11
1	0	0	COLOR12
1	0	1	COLOR13
1	1	0	COLOR14
1	1	1	COLOR15

A questo punto sapete come funziona il Dual Playfield mode. C'è solo una cosa che non sapete... come si attiva il dual playfield !! È molto semplice basta settare a 1 il bit 10 del registro BPLCON0. Come abbiamo già detto è possibile scegliere quale dei due playfield appaia al di sopra dell'altro. Si dice che il playfield che appare sopra ha priorità maggiore. C'è un bit che determina la priorità, il bit 6 del registro BPLCON2 (\$dff104): se esso vale 0 il playfield 1 appare sopra al 2, se invece vale 1 è il playfield 2 che appare sopra all'1. Potete vedere un esempio di Dual Playfield in lezione7u.s

7.7 Priorità tra sprite e playfield

Abbiamo già visto le priorità relative dei vari sprite. Cioè se due sprite si sovrappongono, quello con il numero più basso apparirà al di sopra dell'altro. Inoltre abbiamo appena visto come stabilire la priorità tra i 2 playfield nel modo Dual Playfield. Non ci resta ora che vedere le priorità tra sprite e playfield. Innanzitutto notiamo che gli sprite appaiono sempre al di sopra del colore zero. Per gli altri colori la priorità è controllata dal registro BPLCON2. È possibile settare la priorità indipendentemente per i bit-planes pari e per i dispari. Ciò è molto utile nel modo Dual Playfield, perché ci consente di dare ad ogni playfield una diversa priorità rispetto agli sprite. Nel modo standard, invece è opportuno dare la stessa priorità rispetto agli sprite a planes pari e dispari. Il registro BPLCON2 possiede alcuni bit nei quali scrivere il livello di priorità desiderato per planes pari e dispari. I bit da 0 a 2 contengono il livello di priorità dei bit-planes dispari (che corrispondono al PLAYFIELD 1 nel modo Dual Playfield) mentre i bit da 3 a 5 contengono il livello di priorità dei bit-planes pari (PLAYFIELD 2 nel modo Dual Playfield).

Vediamo come è codificato il livello di priorità, riferendoci ad un generico playfield, visto che la codifica è identica nei 2 casi. Per quanto riguarda le priorità con i playfield gli sprite si considerano a coppie (0-1, 2-3, 4-5 e 6-7). Come sappiamo, la priorità tra gli sprite (e quindi tra le coppie) è fissa:

PRIORITÀ MASSIMA	COPPIA 1 (SPRITES 0 E 1)
	COPPIA 2 (SPRITES 2 E 3)
	COPPIA 3 (SPRITES 4 E 5)
PRIORITÀ MINIMA	COPPIA 4 (SPRITES 6 E 7)

Il livello di priorità ci permette di inserire in questa pila il nostro playfield: lo possiamo mettere al di sopra di tutte le coppie, al di sotto di tutte le coppie, o in mezzo a 2 coppie. Non

è quindi possibile far comparire il playfield al di sotto della coppia 4 e al di sopra della coppia 2, perché la coppia 2 si trova più in alto della coppia 4 nella pila. È invece possibile il contrario. Mostriamo ora una tabella con tutte le possibili priorità, a seconda del livello che settiamo nei bit di BPLCON2

CODICE	000	001	010	011	100	
PRI. MAX	PLAYFIELD	COPPIA 1	COPPIA 1	COPPIA 1	COPPIA 1	
	COPPIA 1	PLAYFIELD	COPPIA 2	COPPIA 2	COPPIA 2	
	COPPIA 2	COPPIA 2	PLAYFIELD	COPPIA 3	COPPIA 3	
	COPPIA 3	COPPIA 3	COPPIA 3	PLAYFIELD	COPPIA 4	
PRI. MIN	COPPIA 4	COPPIA 4	COPPIA 4	COPPIA 4	PLAYFIELD	

Per esempio, come si vede dalla tabella, se vogliamo che gli sprite 0,1,2,3 (cioè le coppie 1 e 2) appaiano al di sopra del playfield e gli altri sprite invece al di sotto, dobbiamo scegliere il codice %010. Questo codice andrà scritto nel registro BPLCON2, nei bit da 0 a 2 se ci si riferisce al playfield 1 in dual-playfield, nei bit da 3 a 5 se ci si riferisce al playfield 2 in dual-playfield, mentre se stiamo usando uno schermo normale, lo dovremo scrivere 2 volte, sia nei bit da 0 a 2 che nei bit da 3 a 5.

In lezione7v1.s trovate un esempio di come settare le priorità degli sprite con uno schermo “normale”. In lezione7v2.s invece, viene usato uno schermo Dual Playfield.

7.8 Collisioni

L'hardware di Amiga mette a disposizione del programmatore un sistema di rilevamento delle collisioni tra sprite e sprite, di quelle tra sprite e playfield e di quelle tra i 2 playfield. Tutti questi tipi di collisione vengono gestiti mediante 2 soli registri: CLXDAT (\$dff00e) che è un registro a sola lettura nel quale vengono segnalate le collisioni, e CLXCON (\$dff098), che è un registro di controllo mediante il quale si può modificare il modo in cui le collisioni vengono rilevate. Cominciamo illustrando la struttura di questi registri. I bit del registro CLXDAT si comportano come dei rilevatori di collisione. Ogni bit è dedicato ad un particolare tipo di collisione. Quando si verifica una collisione di un determinato tipo, il bit ad essa dedicato in CLXDAT assume il valore 1. Quando la collisione non si verifica più il bit ritorna al valore 0. Nella seguente tabella illustriamo il significato dei bit di CLXDAT:

USO DEI BIT DI CLXDAT

```

bit 15 non usato
bit 14 collisione tra coppia 3 e coppia 4
bit 13 collisione tra coppia 2 e coppia 4
bit 12 collisione tra coppia 2 e coppia 3
bit 11 collisione tra coppia 1 e coppia 4
bit 10 collisione tra coppia 1 e coppia 3
bit 9 collisione tra coppia 1 e coppia 2
bit 8 collisione tra playfield 2 e coppia 4
bit 7 collisione tra playfield 2 e coppia 3
bit 6 collisione tra playfield 2 e coppia 2
bit 5 collisione tra playfield 2 e coppia 1
bit 4 collisione tra playfield 1 e coppia 4
bit 3 collisione tra playfield 1 e coppia 3
bit 2 collisione tra playfield 1 e coppia 2
bit 1 collisione tra playfield 1 e coppia 1
bit 0 collisione tra playfield 1 e playfield 2

```

Il registro CLXCON ha la seguente struttura

```

USO BIT DI CLXCON
bit 15  abilita sprite 7
bit 14  abilita sprite 5
bit 13  abilita sprite 3
bit 12  abilita sprite 1
bit 11  abilita bit-plane 6
bit 10  abilita bit-plane 5
bit 9   abilita bit-plane 4
bit 8   abilita bit-plane 3
bit 7   abilita bit-plane 2
bit 6   abilita bit-plane 1
bit 5   valore-collisione bit-plane 6
bit 4   valore-collisione bit-plane 5
bit 3   valore-collisione bit-plane 4
bit 2   valore-collisione bit-plane 3
bit 1   valore-collisione bit-plane 2
bit 0   valore-collisione bit-plane 1

```

(nota: dove è scritto “abilita” si intende *abilita per il rilevamento collisioni*: se per esempio il bit 15 di CLXCON vale 0 *non* vuol dire che lo sprite 7 non può apparire sullo schermo, ma solo che le collisioni che riguardano lo sprite 7 non vengono rilevate)

Spiegheremo un po’ alla volta il significato di questi bit. Cominciamo a parlare della collisione tra sprite e sprite. Diciamo subito che anche per quanto riguarda le collisioni gli sprite sono considerati al livello di coppie. Infatti è possibile rilevare solo le collisioni tra sprite appartenenti a coppie diverse, e non fra sprite appartenenti alla stessa coppia. Per esempio non è possibile rilevare la collisione di sprite 0 con sprite 1. Invece vengono rilevate collisioni tra sprite appartenenti a coppie diverse. Per esempio se si verifica una collisione tra sprite 0 e sprite 2 il bit 9 di CLXDAT (collisione tra coppia 1 e coppia 2) assume il valore 1. Se si verifica una collisione tra sprite 1 e sprite 2, poiché anche lo sprite 1 appartiene (come lo 0) alla coppia 1, sarà sempre il bit 9 ad assumere il valore 1.

Questo però non accadrà sempre. Infatti le collisioni che riguardano gli sprite di numero pari (cioè gli sprite 0,2,4 e 6) vengono sempre rilevate, ma le collisioni che riguardano gli sprite dispari, vengono rilevate solo se noi vogliamo. Per abilitare uno sprite dispari al rilevamento collisioni, dobbiamo mettere a 1 il corrispondente bit di abilitazione nel registro CLXCON. Potete vedere quali sono i bit nella tabella che abbiamo riportato sopra. Gli sprite dispari possono essere abilitati indipendentemente l’uno dall’altro. Abilitare uno o più sprite dispari al rilevamento delle collisioni, comporta vantaggi e svantaggi.

Consideriamo per esempio solo le coppie 1 e 2, e supponiamo di non aver abilitato né lo sprite 1 né lo sprite 3. In questo caso, se si verifica una collisione tra sprite 0 e 2, il bit 9 (di CLXDAT) assume valore 1. Se invece la collisione si verifica tra sprite 1 e 2, oppure tra 0 e 3, oppure tra 1 e 3, non accade nulla, e noi non possiamo sapere che la collisione è avvenuta. Supponiamo invece di aver abilitato uno degli sprite dispari, per esempio sprite 1. In questo caso le collisioni tra sprite 0 e 2 e tra sprite 1 e 2 pongono a 1 il bit 9 di CLXDAT, mentre le collisioni tra sprite 0 e 3 e tra sprite 2 e 3 non provocano nessun effetto. In questa situazione c’è uno svantaggio rispetto al caso precedente, in cui lo sprite 1 non era abilitato. Infatti nel caso precedente, se il bit 9 assumeva il valore 1 eravamo sicuri che la collisione era avvenuta tra sprite 0 e sprite 2. Nel caso presente invece ci sono 2 possibilità: o c’è collisione tra sprite 0 e 2 oppure tra sprite 1 e 2. Non vi è modo di risolvere l’enigma leggendo il registro CLXDAT. Se lo sprite 1 è disabilitato, ma lo sprite 3 è abilitato, si ha una situazione analoga, in quanto vengono rilevate le collisioni tra sprite 0 e 2 e tra sprite 0 e 3, ma non si riesce a distinguere quale delle 2 si è verificata.

Infine nel caso in cui sono abilitati sia lo sprite 1 che il 3, vengono rilevate collisioni tra sprite 0 e 2, tra sprite 0 e 3, tra sprite 1 e 2 e tra sprite 1 e 3, e non c’è modo di distinguere.

Un esempio di collisione tra sprite, con gli sprite dispari disabilitati al rilevamento collisioni, è in lezione7w1.s. Caricatelo e verificate il funzionamento.

Un esempio di collisione tra sprite con uno sprite dispari abilitato è nella lezione7w2.s. Noterete che questo esempio così com'è non funziona; per farlo funzionare, dovreste seguire le modifiche indicate nel commento. In questo esempio, per distinguere se una collisione riguarda lo sprite dispari abilitato, o lo sprite pari ad esso accoppiato, viene impiegata una tecnica basata sul confronto delle posizioni, illustrata nel commento.

Veniamo ora alla collisione tra sprite e playfield. È possibile rilevare una collisione tra una coppia di sprite e uno o più colori del playfield. Anche in questo caso le collisioni sono rivelate considerando coppie di sprite e non i singoli membri della coppia. L'abilitazione degli sprite dispari tramite i bit del registro CLXCON ha effetto anche in questo caso.

Il rilevamento delle collisioni avviene in modo diverso se stiamo usando uno schermo normale o Dual Playfield. Con uno schermo normale, i bit da 1 a 4 di CLXDAT indicano una collisione tra una coppia di sprite e il colore (o i colori) che abbiamo scelto per la collisione. Il bit 1 indica collisione tra il playfield e la coppia 1, il bit 2 tra playfield e coppia 2, il bit 3 tra playfield e coppia 3, il bit 4 tra playfield e coppia 4. I bit da 5 a 8 invece non vanno usati.

Nel caso di modo dual playfield è possibile rilevare una collisione tra uno dei 2 playfield e una coppia di sprite, e i bit di CLXDAT si usano come indicato nella tabella del registro CLXDAT: i bit da 1 a 4 indicano le collisioni tra playfield 1 e le varie coppie di sprite, mentre i bit da 5 a 8 indicano le collisioni tra playfield 2 e le varie coppie di sprite. Per scegliere i colori con cui rilevare le collisioni si usa il registro CLXCON. Iniziamo con il caso di un solo colore. I bit da 6 a 11 di CLXCON indicano quali bit-planes sono attivi per le collisioni. Nel caso in cui vogliamo rilevare collisioni tra sprite e un solo colore dobbiamo abilitare per le collisioni tutti i bit-planes che vengono visualizzati. La scelta del colore con cui rilevare una collisione viene effettuata scrivendo il numero del registro in cui è contenuto il colore nei bit da 0 a 5 di CLXCON.

Per esempio supponiamo di avere uno schermo normale a 16 colori (4 bit-planes) e di non voler considerare le collisioni degli sprite dispari. Se vogliamo rilevare una collisione tra uno sprite e il colore 13 dobbiamo scrivere nel registro CLXCON il valore

```
111111
5432109876543210
$03cb=%0000001111001101
```

Guardiamo il significato dei bit. I bit da 12 a 15 disabilitano gli sprite dispari. Dei bit da 6 a 11 sono a 1 solo i bit 6,7,8,9. Questo indica che solo i bit-planes da 1 a 4 sono abilitati per le collisioni. Si tratta dei soli bit-planes attivi. I bit da 0 a 5 contengono il numero %001101=13 cioè il numero del registro che ci interessa. Nel caso Dual Playfield la situazione è la stessa, solo che attivando tutti i bit-planes utilizzati per le collisioni si abilitano le collisioni con 2 colori contemporaneamente: per esempio, se con 2 playfield da 8 colori ciascuno vogliamo abilitare il rilevamento delle collisioni per il colore 7 del playfield 1 e il colore 2 del playfield 2 dobbiamo scrivere in CLXCON il numero

```
111111
5432109876543210
$0fbb=%0000111111011101
```

Questa combinazione di bit indica che tutti i bit-planes sono abilitati al rilevamento collisioni (tutti i bit da 6 a 11 valgono 1). Inoltre il numero del colore usato per il playfield 1 è dato dai bit 0,2 e 4 che messi affiancati formano il numero %111=7, mentre il numero del colore usato per il playfield 2 è dato dai bit 1,3 e 5 che messi affiancati formano il numero %010=2.

Da notare che la collisione di uno sprite con un colore del playfield 1 provoca l'accensione di un bit di CLXDAT diverso dal caso di collisione dello stesso sprite con un colore del playfield

2. Per esempio, come potete verificare nella tabella del registro CLXDAT, la collisione sprite 0 - playfield 1 mette a 1 il bit 1 di CLXDAT, mentre la collisione sprite 0 - playfield 2 mette a 1 il bit 5 di CLXDAT.

È possibile anche rilevare collisioni di uno sprite con più di un colore contemporaneamente sebbene solo in alcune particolari condizioni. Per capire come ciò sia possibile occorre tenere presente la rappresentazione binaria dei numeri dei registri colore. Ci sono, come sapete 32 registri colore numerati da 0 a 31. La possibilità di rilevare collisioni con 2 colori contemporaneamente si basa sul fatto che le rappresentazioni di alcuni numeri binari sono simili.

Per esempio consideriamo i numeri 2 e 21. In binario si ha $2 = \%00010$ e $21 = \%10101$ (consideriamo 5 bit per poter scrivere numeri fino a 31). Come vedete le rappresentazioni binarie di questi 2 numeri sono completamente diverse. Non vi è modo dunque di rilevare collisioni con entrambi i colori contemporaneamente.

Consideriamo invece i numeri 22 e 23. Osserviamo ora che espressi in binario $22 = \%010110$ e $23 = \%010111$. Le rappresentazioni dei 2 numeri differiscono solo per un bit, il bit più basso. In questo caso è possibile rilevare collisioni con entrambi i colori. Infatti il valore del bit più basso (che in questo caso differenzia i colori) è dato dal bit-plane 1. Se noi *non* abilitiamo il bit-plane 1 al rilevamento collisioni, verranno presi in considerazione solo i valori dei bit-plane 2,3,4 e 5 (siamo su uno schermo a 32 colori, quindi in totale 5 bit-planes), e il valore assunto dal bit-plane 1 non avrà alcuna influenza. Scriviamo dunque in CLXCON il valore:

```
111111
5432109876543210
CLXCON= %0000011110010110
```

Questo vuol dire che la collisione verrà rilevata basandosi sui soli bit-plane abilitati (e cioè 2,3,4 e 5) e precisamente quando il nostro sprite si sovrapporrà ad un pixel che abbia:

```
bitplane 1=(0 o 1) perché non è abilitato
bitplane 2=1
bitplane 3=1
bitplane 4=0
bitplane 5=1
```

Come abbiamo visto, sia la rappresentazione binaria di $22 = \%010010$ che quella di $23 = \%010111$ hanno questa particolare configurazione di bit, pertanto entrambi i colori provocano una collisione al passaggio dello sprite. Notate che il bitplane che NON abbiamo abilitato (l'1) corrisponde proprio all'unico bit che differenzia le rappresentazioni binarie di 22 e di 23.

Questa tecnica è applicabile a una qualunque coppia di colori le cui rappresentazioni binarie differiscano di un solo bit. Per esempio anche i numeri $8 = \%001000$ e $9 = \%001001$ differiscono per il bit più basso, quindi anche per rilevare collisioni tra lo sprite e questi 2 colori si deve disabilitare il bit-plane 1. Se invece consideriamo i colori $10 = \%001010$ e $14 = \%001110$, notiamo che le 2 rappresentazioni binarie differiscono nel bit 2 (numeriamo i bit da destra a sinistra a partire da 0) che corrisponde al bit-plane 3. Per rilevare collisioni tra lo sprite e questi 2 colori si deve disabilitare il bit-plane 3, e pertanto assegnare a CLXCON il valore riportato sotto:

```
111111
5432109876543210
CLXCON= %0000011011001010 ; bit 8=0 indica bit-plane 3 NON abilitato
```

Se disabilitiamo 2 bit-plane possiamo rilevare collisioni tra 4 colori. Il principio è sempre lo stesso. Prendiamo per esempio i colori:

```

1=%00001
3=%00011
5=%00101
7=%00111

```

questi 4 colori hanno i bit 0, 3 e 4 uguali tra loro, mentre si differenziano in quanto ogni colore ha una diversa combinazione di valori nei bit 1 e 2. Per rilevare collisioni tra uno sprite e tutti e 4 questi colori basta disabilitare i bit-plane 2 e 3 che corrispondono appunto ai bit 1 e 2. Disabilitando 3 bit plane si rilevano collisioni con 8 colori contemporaneamente, disabilitandone 4 con 16 colori.

Anche operando in modo Dual Playfield è possibile, per ciascun playfield, disabilitare alcuni bit-planes per rilevare collisioni tra lo sprite e più di un colore per ogni playfield (ricordiamo che se dobbiamo rilevare la collisione tra lo sprite e 2 colori che però appartengono uno al playfield 1 e uno al playfield 2 ciò non è necessario, perché in CLXDAT abbiamo per ogni playfield un bit che ci consente di rilevare contemporaneamente la collisione con entrambi i playfield).

In *lezione7x1.s* vediamo un esempio di collisione tra sprite e playfield in modo “standard”. In *lezione7x2.s* invece c’è un esempio con il modo Dual Playfield. In entrambi i listati nel commento sono riportati diversi esempi di come rilevare collisioni con più di un colore per volta.

L’ultimo tipo di collisione è tra playfield 1 e playfield 2, ovviamente in modo Dual Playfield. È possibile rilevare una collisione tra uno o più colori del playfield 1 e uno o più colori del playfield 2, abilitando solo alcuni bit-planes esattamente con la stessa procedura adottata nel caso di collisione tra sprite e playfield. Quando viene rilevata una collisione tra i due playfield, bit 0 di CLXDAT assume il valore 1. Un esempio di questo tipo di collisione è in *lezione7x3.s*.

7.9 Uso diretto dei registri degli sprite

Vedremo ora un diverso metodo per utilizzare gli sprite. Finora abbiamo generato degli sprite utilizzando i registri SPRxPT, ovvero dei puntatori a delle strutture dati (dette strutture sprite) che contengono tutte le informazioni necessarie alla visualizzazione degli sprite. Esiste però un altro metodo per creare degli sprite che può essere usato in alternativa o anche in aggiunta a quello con i puntatori. Chiameremo questo nuovo metodo “uso diretto degli sprite”. L’uso diretto degli sprite non è conveniente nella maggior parte dei casi, ma a volte può risultare utile. Per comprendere bene di cosa si tratti dobbiamo approfondire il discorso sulla visualizzazione degli sprite.

Quando memorizziamo in un registro SPRxPT l’indirizzo di una struttura sprite (secondo le modalità della tecnica “standard” di utilizzo degli sprite), attiviamo una procedura automatica che permette di visualizzare effettivamente gli sprite. Infatti i dati sulla posizione e la forma che noi abbiamo memorizzato nella struttura sprite, vengono trasferiti automaticamente, attraverso un “meccanismo” hardware chiamato DMA, in appositi registri, diversi dai registri SPRxPT; è proprio la scrittura dei dati in questi registri che REALMENTE permette la visualizzazione degli sprite. Del DMA, che è uno strumento molto importante dell’Amiga, diremo di più in una prossima lezione. Per il momento ci basta conoscere il ruolo che esso svolge nella visualizzazione degli sprite. Si comporta in pratica come un postino. Immaginate che la struttura dati dello sprite che voi avete costruito in memoria sia un mucchio di lettere indirizzate a diversi destinatari (registri). Il DMA si occupa di portare queste lettere a destinazione, smistandole tra i vari destinatari.

L’uso diretto degli sprite consiste, appunto, nello scrivere direttamente i dati degli sprite negli appositi registri, cioè nel portare “di persona” le lettere ai vari destinatari, rubando il lavoro al postino DMA. Visto che il DMA svolge il suo lavoro gratis, vi potreste chiedere a cosa serva questa tecnica. In effetti come abbiamo già detto di solito essa non offre vantaggi; tuttavia in alcuni casi può rivelarsi utile. Vediamo dunque in cosa consiste questa tecnica. Come abbiamo

già detto i dati degli sprite vengono scritti direttamente in alcuni registri. Ci sono 4 registri per ogni sprite, chiamati SPRxPOS, SPRxCTL, SPRxDATA, SPRxDATB (al posto della x dovete mettere il numero dello sprite che volete usare). Gli indirizzi di questi registri dipendono dallo sprite a cui ci si riferisce. Li possiamo calcolare con delle semplici formule. Con “x” indichiamo il numero dello sprite, da 0 a 7.

```
indirizzo SPRxPOS = $dff140+(x*8)
indirizzo SPRxCTL = $dff142+(x*8)
indirizzo SPRxDATA = $dff144+(x*8)
indirizzo SPRxDATB = $dff146+(x*8)
```

Potete comunque cercarli con l’help dell’ASMONE <=C>.

Ora descriviamo l’uso di questi registri. La forma di uno sprite viene scritta nei registri SPRxDATA e SPRxDATB, che costituiscono i 2 piccoli bit-planes dello sprite (SPRxDATB è il piano 2). Questi registri hanno lo stesso ruolo delle coppie di word che definiscono la forma di una riga dello sprite nella struttura sprite. Notate che per ogni sprite ci sono 2 registri che contengono i dati relativi ad una sola riga dello sprite. La posizione orizzontale di uno sprite come sapete è costituita da 9 bit, chiamati H0, H1 ... H8. Questi 9 bit sono suddivisi in due registri: il bit H0, cioè il bit basso si trova nel bit 0 del registro SPRxCTL. Gli altri 8 invece nel byte basso del registro SPRxPOS. In poche parole questi 2 registri si comportano, per quanto riguarda la posizione orizzontale, esattamente come le 2 word di controllo della struttura sprite. La posizione verticale, invece, con questa tecnica non viene determinata, perché gli sprite si comportano in modo piuttosto strano.

Per essere visualizzato, uno sprite deve essere attivato. Ciò accade quando si scrive nel registro SPRxDATA. Una volta attivato, lo sprite viene visualizzato ad ogni riga nella posizione orizzontale indicata, come abbiamo appena visto, nei registri SPRxPOS e SPRxCTL. La forma dello sprite è per ogni riga quella contenuta nei registri SPRxDATA e SPRxDATB. Quindi se il contenuto di questi registri non viene modificato ad ogni riga, lo sprite avrà la stessa forma ad ogni riga. Lo sprite viene visualizzato, fino a che non lo si disattiva scrivendo nel registro SPRxCTL. Per visualizzare uno sprite che cambi forma ad ogni riga si dovrebbe dunque utilizzare una copperlist fatta in questa maniera (supponiamo di usare lo sprite 0 e che sia VSTART=\$40, VSTOP=\$60, HSTART=\$160):

```
1      dc.w    $4007,$fffe    ; WAIT - aspetta la linea VSTART
2      dc.w    $140,$0080    ; SPROPOS - posizione orizzontale
3      dc.w    $142,$0000    ; SPROCTL
4      dc.w    $146,$0e70    ; SPRODATAB - forma sprite riga 1, piano 2
5      dc.w    $144,$03c0    ; SPRODATA - forma sprite riga 1, piano 1
6                                ; inoltre attiva la visualizzazione, per
7                                ; questo va scritta per ultima.
8
9      dc.w    $4107,$fffe    ; WAIT - aspetta la linea VSTART+1
10     dc.w    $146,$0a70    ; SPRODATAB - forma sprite riga 2, piano 2
11     dc.w    $144,$0300    ; SPRODATA - forma sprite riga 2, piano 1
12
13     dc.w    $4107,$fffe    ; WAIT - aspetta la linea VSTART+2
14     dc.w    $146,$0a7f    ; SPRODATAB - forma sprite riga 3, piano 2
15     dc.w    $144,$030f    ; SPRODATA - forma sprite riga 3, piano 1
16
17 ; ripeti per ogni riga Y
18 ;      dc.w    $40+Y07,$fffe    ; WAIT - aspetta la linea VSTART+Y
19 ;      dc.w    $146,DAT0Y2    ; SPRODATAB - forma sprite riga Y, piano 2
20 ;      dc.w    $144,DAT0Y1    ; SPRODATA - forma sprite riga Y, piano 1
21 ; mettendo al posto di DAT0Y1 e DAT0Y2 i dati della forma degli sprite.
22
23     dc.w    $6007,$fffe    ; WAIT - aspetta la linea VSTOP
24     dc.w    $142,$0000    ; SPROCTL - disattiva lo sprite
```

Come vedete per sprite abbastanza alti è necessaria una copperlist molto lunga e complicata. In questo caso conviene decisamente usare il DMA. Supponiamo però di dover visualizzare uno

sprite che abbia la stessa forma ad ogni riga. Per esempio uno sprite che rappresenti una colonna. In questa situazione la nostra copperlist diventa semplicissima e cortissima (supponiamo di usare lo sprite 0 e che sia VSTART=\$40, VSTOP=\$60, HSTART=\$160):

```

1      dc.w    $4007,$fffe      ; WAIT – aspetta la linea VSTART
2      dc.w    $140,$0080      ; SPROPOS – posizione orizzontale
3      dc.w    $142,$0000      ; SPROCTL
4      dc.w    $146,$0e70      ; SPRODATB – forma sprite riga 1, piano 2
5      dc.w    $144,$03c0      ; SPRODATA – forma sprite riga 1, piano 1
6                                     ; inoltre attiva la visualizzazione, per
7                                     ; questo va scritta per ultima.
8
9      dc.w    $6007,$fffe      ; WAIT – aspetta la linea VSTOP
10     dc.w    $142,$0000      ; SPROCTL – disattiva lo sprite

```

Notate che la nostra copperlist, oltre a essere corta, non varia con l'altezza dello sprite. Al contrario, se volessimo usare il DMA per visualizzare questo sprite, saremmo costretti a memorizzare nella struttura dati le 2 word che rappresentano la forma tante volte quante sono le righe che costituiscono lo sprite. Pensate al caso in cui si deve visualizzare una colonna alta 100 righe. Se usassimo il DMA dovremmo memorizzare una struttura sprite che occupa molta memoria:

```

1  StrutturaSprite:
2      dc.b    VSTART,HSTART,VSTOP,0
3      dc.w    $ffff,$0ff0      ; riga 1
4      dc.w    $ffff,$0ff0      ; riga 2
5      dc.w    $ffff,$0ff0      ; riga 3
6      dc.w    $ffff,$0ff0      ; riga 4
7      dc.w    $ffff,$0ff0      ; riga 5
8      dc.w    $ffff,$0ff0      ; riga 6
9      dc.w    $ffff,$0ff0      ; riga 7
10     dc.w    $ffff,$0ff0      ; riga 8
11
12     .... e cosi' via, fino a:
13
14     dc.w    $ffff,$0ff0      ; riga 99
15     dc.w    $ffff,$0ff0      ; riga 100
16     dc.w    0,0              ; fine sprite

```

Con l'uso diretto degli sprite, invece basta una semplice copperlist:

```

1      dc.b    VSTART,7,$ff,$fe      ; WAIT – aspetta la linea VSTART
2      dc.w    $140
3      dc.b    $00,HSTART            ; SPROPOS – posizione orizzontale
4      dc.w    $142,$0000            ; SPROCTL
5      dc.w    $146,$ffff            ; SPRODATB – forma sprite riga 1, piano 2
6      dc.w    $144,$0ff0            ; SPRODATA – forma sprite riga 1, piano 1
7                                     ; inoltre attiva la visualizzazione, per
8                                     ; questo va scritta per ultima.
9
10     dc.b    VSTOP,7,$ff,$fe      ; aspetta la linea VSTOP
11     dc.w    $142,$0000            ; SPROCTL – disattiva lo sprite

```

Un semplice esempio di uso diretto degli sprite è riportato in lezione7y1.s. Nel programma lezione7y2.s, invece, usando degli sprite in accesso diretto realizziamo delle barre verticali analoghe a quelle che si fanno orizzontalmente con il copper.

Con la tecnica dell'uso diretto degli sprite, è possibile anche visualizzare uno stesso sprite più volte su una stessa riga. Il metodo viene spiegato e applicato in lezione7y3.s. Gli sprite generati più volte sulla stessa linea sono anche detti *multiplexed*, cioè “multiplexati”. Dunque eravamo partiti dicendo che ci sono 8 sprite solamente, ma abbiamo visto che l'assembler ci permette di moltiplicare gli sprite e anche di fargli assumere molti più colori di quelli standard, cambiando la palette più volte anche orizzontalmente. L'unico inconveniente è che ci vogliono delle copperlist molto lunghe, ma ne vale sicuramente la pena.

Uno sviluppo di questa idea ci porta a realizzare una schermata completamente grazie agli sprite, nell'esempio Lezione7y4.s.

Per compiere tale operazione però occorre scrivere una copperlist lunghissima, e per renderla più comprensibile sono stati usati dei *simboli* o *equates*, una direttiva del linguaggio assembler che permette di chiamare con un nome scelto a piacere un certo numero fisso, per cui scrivendo il nome viene assemblato il numero che gli corrisponde. Facciamo questo esempio: vogliamo fare in modo di accedere al registro COLORE0, che come sappiamo è \$dff180. Possiamo scrivere:

```
1      move.w    #$123, $dff180
```

Ma se volessimo potremmo anche scrivere così:

```
1  COLORE0      EQU      $dff180      ; Definizione di un simbolo
2      move.w    #$123, COLORE0
```

In pratica abbiamo definito che quando l'asmone trova scritto COLORE0 deve assemblare come se avesse trovato \$dff180. È come se definissimo una label, infatti bisogna inventarci un nome e scriverlo senza precederlo da spazi, ma non occorrono i : (in realtà si possono anche mettere i :, allo stesso modo le label potrebbero avere i : o non averle, l'ASMONE assembla comunque, ma certi assembler preferiscono che le LABEL siano seguite dai : e che i simboli (o equates) non li abbiano). EQU significa infatti EQUIVALE A. Quasi tutti gli assembler accettano anche il simbolo = al posto del simbolo EQU per la definizione. Facciamo un altro esempio:

```
1  NUMEROLOOP   =        10
2
3      MOVEQ     #NUMEROLOOP-1,d0
4  Loop:
5      clr.l     (a0)+
6      dbra      d0,NUMEROLOOP
7      rts
```

Con questo listatino azzeriamo 10 longword. L'utilità degli EQUATES è che possiamo metterli tutti all'inizio del listato, in modo che se vogliamo modificare certi valori, ad esempio quanti loop fare o quanti bitplanes puntare, basta modificare il valore del simbolo dopo l'= o l'EQU all'inizio del listato. Inoltre è possibile eseguire operazioni tra simboli. Un esempio pratico può essere il calcolo dello spazio da azzerare per un bitplane:

```
1  BytesPerRiga =        40
2  NumeroRighe  =       256
3  SpazioBitplane =      BytesPerRiga*NumeroRighe
4
5      ...
6
7      section plane, bss_C
8
9  Bitplane:
10     ds.b      SpazioBitplane
```

Nel listato SpazioBitplane vale 10240, ossia 40*256. In Lezione7y4.s vengono definiti dei simboli per la copperlist.

Infine nella lezione7y5.s faremo scorrere la schermata formata dagli sprite e ne approfitteremo per conoscere 2 nuove istruzioni del 68000, chiamate ROR e ROL. Le spiegheremo nel commento al listato.

7.10 Animazione sprite

Concludiamo questa lezione con una spiegazione sull'animazione degli sprite. Ritorniamo ora a considerare sprites "normali", cioè generati tramite i puntatori SPRxPT e il DMA. Per animare uno sprite è necessario cambiarne la forma ogni volta che esso viene ridisegnato. Ogni forma che viene assunta dallo sprite è detta "fotogramma dell'animazione". Di solito l'animazione è fatta in modo da avere una certa sequenza di fotogrammi che viene continuamente ripetuta. Pensate

per esempio ad un omino che cammina sullo schermo; noterete che tutti i passi sono uguali tra loro.

Per animare un omino che cammina sullo schermo si disegnano un certo numero di fotogrammi che visti in successione raffigurano un passo completo dell'omino. Quando l'omino ha completato il passo deve iniziarne uno nuovo: a questo punto si mostrano di nuovo gli stessi fotogrammi iniziando dal primo. Ripetendo per ogni passo sempre gli stessi fotogrammi possiamo mostrare l'omino camminare tutto il tempo che vogliamo, con un numero limitato di fotogrammi (è evidente che essendo i fotogrammi delle immagini, occupano memoria, quindi si deve cercare di usarne il meno possibile).

Fin qui il discorso è valido per un qualsiasi oggetto animato, e sarà bene che lo teniate presente anche quando tratteremo le animazioni realizzate tramite blitter. Ora invece ci occupiamo delle animazioni fatte tramite sprite. Questo vuol dire che abbiamo uno sprite che si muove sullo schermo e che ogni volta che viene ridisegnato assume una forma diversa. Di solito si procede in questo modo: per ogni fotogramma si realizza una struttura sprite, e ogni volta che lo sprite viene ridisegnato si fa puntare il registro SPRxPT ad un diverso fotogramma (ovvero ad una diversa struttura dati). La posizione dello sprite viene scritta ogni volta nella struttura del fotogramma a cui SPRxPT viene fatto puntare. Un esempio pratico è in *lezione7z.s*.

Questo esempio è anche il termine della lezione7 e del DISCO 1 del corso. Il disco 2 nel momento in cui scrivo (Maggio 1995) non è ancora del tutto terminato, comunque gli argomenti trattati sono:

- BLITTER (copymode, linemode e fill)
- Interrupt, CIAA/CIAB, caricamento da disco, Tastiera
- Audio
- Approfondimenti sul 68000, accenni al 68020
- Programmazione di videogiochi
- Routines matematiche (3d, frattali)
- Chipset AGA
- Compatibilità e ottimizzazioni
- Programmazione della scheda video PICASSO II!!!!

Non so se avrò tempo di terminare un lavoro così immane, già questo primo disco (in cui per la verità non ho trattato tanti argomenti) mi sembrava che fosse interminabile. Devo ringraziare Luca Forlizzi (The Dark Coder) per avermi aiutato a terminare la LEZIONE7, e anche i due volenterosi beta tester ANDREA SCARAFONI e FEDERICO STANGO che mi hanno indicato dove non ero stato chiaro, o addirittura dove avevo scritto frasi incomprensibili. Purtroppo non posso ringraziare coloro che hanno solo PROMESSO di aiutarmi, ma che poi sono spariti, come Alvis Spanò (AGA/LUSTRONES).

Per ricevere il disco 2, o almeno quello che avrò fatto in tempo a fare, potete scrivermi una lettera, o meglio sarebbe un disco con dei vostri programmini:

Fabio Ciucci
[omissis]

Non scrivetemi fino a che non avete veramente assimilato tutto quello che c'è in questo disco, non si diventa programmatori solamente avendo listati. Nel frattempo copiate a tutti questo disco, chiunque deve averlo, anche il Papa quando si affaccia dal balcone... chiunque abbia un Amiga in Italia deve avere questo disco. E non datelo solo a chi pensate che sia interessato, perché ho verificato che quelli che hanno seguito il corso e imparato di più sono quelli che meno mi aspettavo che lo facessi. Pensate che il più avanti qua a Lucca è un ragazzo, Michele, a cui avevo dato il disco perché lo copiasse ad un suo amico, che pareva fosse interessato. Non li ho più sentiti per un paio di mesi, poi mi si è presentato Michele a casa con la preview del suo gioco!!! Mentre il suo amico era rimasto impantanato! Quindi diffondete questo disco come fossero volantini per il vostro partito politico o opuscoli della vostra religione, scrivete annunci nelle bacheche o giornalotti delle vostre scuole o università per avvertire tutti che avete il disco per imparare a programmare l'Amiga, date delle copie del disco ai negozianti di Amiga della vostra zona chiedendogli di darli a chi chiede informazione sulla programmazione Amiga, insomma evangelizzate (assemblizzate) questo povero paese!

Quando poi sarà il momento di avere il disco 2, io spero di averlo finito, comunque vi manderò quello che c'è, dovrete scrivermi una lettera normale da 750 lire con una breve presentazione con età, segni particolari, computer posseduto, aspirazioni e quello che vi pare, con il vostro indirizzo, è ovvio. In concomitanza dovrete mandarmi un vaglia postale ad offerta libera, minimo 10.000 (per ora mi hanno mandato quasi tutti 10.000, qualcuno 20.000, poi sempre più rari i 30.000 e un mitico 60.000 da un Genovese, tanto per sfatare la leggenda che sono tirchi). Nella lettera dovete specificare quanto mi avete mandato di vaglia, perché lettera e vaglia di solito arrivano in tempi un pò diversi. Spero che non accadano ritardi galattici per le PT.

Altrimenti potete variare in questo modo: potete spedirmi un dischetto con la lettera in formato .txt, e magari qualche vostro listatino o altro per riempire il disco. Da notare che vi conviene mettere in disco in una busta da spedizione di quelle piccole, e di chiuderla solo con il fermacampioni o l'autoadesivo della busta stessa, senza mettere foglietti o lettere oltre al disco. Insomma non dovete megasigillare tutto con nastro adesivo o altro, questo per permettere l'ispezione postale, che tanto poi non la fanno mai, ma se il pacco è così si paga meno.

Dovete poi scrivere "pacchetto" sulla busta, e dire alla (spero) gentile impiegata dell'ufficio postale che si tratta di un pacchetto senza lettere dentro, e questo vi farà risparmiare rispetto ad un pacco chiuso con lettera, che conta come "LETTERA PESANTE"! Un pacchetto così con 1 solo dischetto dovrebbe costare 1200 lire. Se volete che arrivi prima, potete farlo espresso, aggiungendo 3000 lire, ma allora vi conviene calcolare il tempo che vi manca a finire il disco 1 e spedire per tempo... e che programmatori siete se non siete furbi? NOTA: Se volete che io vi mandi il disco 2 in espresso, potete fare i calcoli come prima, oppure partire da una base minima di 13000 lire e indicare nella lettera (cartacea o .txt) che volete il disco 2 espresso.

Un ultimo sistema potrebbe essere quello di mettere i soldi "sciolti" nella lettera o nel pacchetto, ma occorre metterli in modo che non si vedano in trasparenza, e il pacchetto allora dovrebbe essere fatto chiuso... decidete voi la strada, ma credo che lettera o pacchetto + vaglia sia meglio.

Eventuali donazioni o contatti incoraggeranno me e i miei "collaboratori" a continuare l'opera di stesura di questo corso. Saranno graditissime anche eventuali offerte di collaborazione da parte di programmatori esperti, in particolare di coloro che hanno programmato giochi (quelli che fanno le demo li conosco tutti qua in Italia, la scena è una specie di famiglia, con i litigi delle famiglie, però!).

Devo anche segnalarvi il miglior club di utenti Amiga, dove potreste trovare altri programmatori, grafici o musicisti per collaborazioni:

Mirko Lalli
[omissis]

LEZIONE 8 - APPROFONDIMENTI SUL 68000

In questa lezione verranno approfondite le conoscenze sul 68000 e saranno fatte delle precisazioni su vari argomenti già trattati. Una nota per chi installa il corso su HardDisk: vi consiglio di fare delle directory, con il nome del relativo disco del corso:

```
Assembler1  
Assembler2  
Assembler3  
...
```

Dove copiare gli interi dischetti. Poi aggiungete ad `s:startup-sequence`:

```
assign Assembler1: dh0:Assembler1  
assign Assembler2: dh0:Assembler2  
assign Assembler3: dh0:Assembler3  
...
```

(dh0: è solo un esempio... ci metterete il drive opportuno, naturalmente!). Poi vi consiglierai di scompattare tutti i sorgenti e i dati, che sono in formato powerpacker. Per fare ciò, copiate il file `c:PP` nell'HardDisk, e verificate se avete in LIBS: la `PowerPacker.library`, altrimenti copiatela da questo disco. Ora, eseguite dallo shell PP, in modo da abilitare la scompattazione automatica. Ora fatevi una directory "provvisoria", per esempio chiamatela "buffer". Se copiate TUTTI i file dalla directory `Assembler1` alla directory `Buffer`, tutti i file saranno scompattati, infatti si "allungheranno". Ora potete ricopiarli tutti in `Assembler1` (magari con un `MOVE` del `DiskMaster` o di `DirOpus`, che li ricancella anche da `buffer`). Allo stesso modo potete ricopiare tutto `Assembler2` in `buffer`, poi ricopiare in `Assembler2`. Per risparmiare questa pur veloce operazione, potete caricare il PP prima di copiare i file da dischetto alle directory su HD, in modo che i file in `AssemblerX` siano scompattati. I 3 comandi `assign` servono per fare in modo che anziché cercare il disco con il nome "`AssemblerX:`" si cerchi nella directory `dh0:AssemblerX`. In alcuni dei prossimi listati, infatti, si cerca `Assembler2:`, e così sarà anche per `Assembler3:`.

P.S: Ho intenzione di tradurre in Inglese l'intero corso. Però questo mi costringerebbe a non scrivere più nuove lezioni per MESI interi... Quindi, se trovassi qualcuno che abbia già letto il disco 1, che sappia l'inglese decentemente, e abbia voglia di tradurre almeno una lezione, io

sarei felicissimo. Chi mi aiutasse nell'opera di traduzione naturalmente avrebbe una percentuale molto alta sui profitti ricavati dall'estero (che ne dite del 30%? Forse è troppo...) Chi può aiutarmi, (sto parlando di un BEL lavoro di traduzione), mi contatti quanto prima.

P.S2: Mi raccomando di copiare a TUTTI i vostri amici (e non) il disco 1 del corso, di darlo ai negozianti della vostra città, di mettere annunci nelle bacheche o nei giornaletti per vedere se interessa a qualcuno, e trovare nuovi contatti per programmare. In particolare potreste diffondere la filosofia CyberAssemblica della scena, di cui trovate un sunto nel file SCENA.TXT. Anche il Papa quando si affaccia dal balcone deve avere il disco 1 del corso! (liberamente copiabile). Per quanto riguarda il disco 2, ossia questo, invece, non è liberamente copiabile, altrimenti io poi non prenderei nemmeno quei (non molti) soldi che mi mandano coloro che hanno avuto solo il disco 1. Immaginiamoci se costoro avessero subito entrambi i dischi! Comunque, quando (e se) farò dischi 3, 4, eccetera, probabilmente renderò anche il disco 2 liberamente copiabile (shareware, però), in modo che i nuovi possano avere subito i disk1+2, poi io mi rifaccio qualche spicciolo col disco 3, 4...

Continuiamo la lezione, sia che il file sia su HardDisk che su Floppy. Innanzitutto è necessario completare la lezione sul 68000, dato che per ora ne è stato fatto un uso semplificato. Già dalla lezione precedente avete potuto constatare che spessissimo è necessario operare sui singoli bit dei numeri o dei registri, ebbene più andrete avanti nella programmazione e più tenderete ad inserire istruzioni come AND, OR, NOT, ROL, ASL...eccetera, ossia le operazioni logiche booleane e di scorrimento sui bit. Ah! Nella directory LEZIONI è presente un testo che spiega cosa è la SCENA AMIGA. Ora che state diventando dei coder è opportuno che sappiate chi dovete ringraziare per la nascita della cultura della programmazione delle demo, in quel modo "illegale", che come avete visto però funziona, e anche molto bene. Il testo è SCENA.TXT, leggetevelo quando non avete voglia di farvi fumare il cervello con le lezioni di asm!

Prima di procedere nel corso, è necessario fare un listato di startup, ossia di salvataggio e ripristino della copper di sistema, più efficiente di quello usato fino ad adesso, inoltre tale startup dovrà essere inclusa in tutti i prossimi listati, dunque sarà certamente più utile caricarla tramite la direttiva INCLUDE già vista per includere la routine che suona la musica. "Costruiremo" questa startup nella lezione passo dopo passo, come risultato delle varie precisazioni. Analizziamo la procedura di startup usata nelle lezioni precedenti:

```

1  Inizio:
2      move.l 4.w,a6          ; Execbase
3      jsr   -$78(a6)         ; Disable
4      lea   GfxName(PC),a1   ; Nome lib
5      jsr   -$198(a6)        ; OpenLibrary
6      move.l d0,GfxBase
7      move.l d0,a6
8      move.l $26(a6),OldCop  ; salviamo la vecchia COP
9
10 ; Qua è puntata la nostra copperst e ci sono le routines
11
12      move.l OldCop(PC),$dff080 ; Puntiamo la cop di sistema
13      move.w d0,$dff088        ; facciamo partire la cop
14      move.l 4.w,a6
15      jsr   -$7e(a6)           ; Enable
16      move.l gfxbase(PC),a1
17      jsr   -$19e(a6)         ; Closelibrary
18      rts

```

In pratica arrestiamo il multitasking e gli interrupt di sistema tramite il Disable, poi apriamo la libreria grafica, tramite la quale possiamo trovare l'indirizzo della vecchia copperlist, sapendo che si trova \$26 bytes dopo l'indirizzo di GfxBase. Sapendo come rimettere a posto la vecchia copperlist e avendo immobilizzato il WorkBench, agiamo in modo diretto sui chip custom senza temere incompatibilità. Al termine delle routines sarà necessario eseguire l'Enable per riattivare il multitasking e puntare la vecchia copperlist per rivisualizzare le finestrelle del sistema

operativo. Queste operazioni sono il minimo indispensabile per poter lavorare, ma si potrebbero fare dei perfezionamenti al codice, per esempio si potrebbero eseguire delle routines della libreria grafica che resettano il modo video, in modo da resettare anche modi video per monitor VGA/Multisync/Multiscan o altri. Una funzione apposita esiste, e si chiama LoadView, vediamo:

```

1 ; Abbiamo il GfxBase nel registro A6
2
3     MOVE.L  $22(A6),WBVIEW ; Salva il WBView attuale di sistema
4     SUBA.L  A1,A1          ; View nullo per azzerare il modo video
5     JSR     -$DE(A6)       ; LoadView nullo - modo video azzerato

```

La funzione LoadView richiede che sia specificato l'indirizzo della struttura view in a1, ma in questo caso A1 è AZZERATO, dato che sommiamo a1 a se stesso, ricavando a1=0. Quando A1 è NULLO la funzione resetta il modo video riportandolo ad un LOWRES non interlacciato e senza frequenze speciali per i monitor. A questo punto siamo più sicuri di avere la situazione della copperlist sotto controllo, inoltre abbiamo salvato il vecchio puntatore alla struttura WBVIEW in una label, che ci permetterà alla fine del listato di ripristinarlo, e con esso le eventuali frequenze speciali per monitor:

```

1     MOVE.L  WBVIEW(PC),A1 ; Vecchio WBVIEW in A1
2     MOVE.L  GFXBASE(PC),A6 ; GFXBASE in A6
3     JSR     -$DE(A6)       ; loadview - rimetti il vecchio View

```

Per essere sicuri che anche il modo interlacciato sia resettato e ripristinato correttamente, si può attendere per due fotogrammi tramite l'esecuzione della routine asmWaitOF, sempre della graphics.library:

```

1     MOVE.L  WBVIEW(PC),A1 ; Vecchio WBVIEW in A1
2     MOVE.L  GFXBASE(PC),A6 ; GFXBASE in A6
3     JSR     -$DE(A6)       ; loadview - rimetti il vecchio View
4     JSR     -$10E(A6)      ; WaitOf ( Risistema l'eventuale interlace )
5     JSR     -$10E(A6)      ; WaitOf

```

Per avere l'animo in pace, mettiamo un paio di WaitOF anche dopo il primo loadview che resetta il modo video, e già che ci siamo controlliamo se il reset è veramente avvenuto testando se il WBVIEW è azzerato come previsto:

```

1 ; Abbiamo il GfxBase nel registro A6
2
3     MOVE.L  $22(A6),WBVIEW ; Salva il WBView attuale di sistema
4     SUBA.L  A1,A1          ; View nullo per azzerare il modo video
5     JSR     -$DE(A6)       ; LoadView nullo - modo video azzerato
6     JSR     -$10E(A6)      ; WaitOf ( Queste due chiamate a WaitOf )
7     JSR     -$10E(A6)      ; WaitOf ( servono a resettare l'interlace )

```

Avendo usato routines del sistema operativo, siamo certi che in macchine del futuro il modo video sarà comunque azzerato. Per “esagerare” nella compatibilità, possiamo richiamare alla fine del listato delle funzioni della intuition.library che “ridisegnano” gli schermi e le finestrelle:

```

1     move.l  4.w,a6          ; ExecBase in A6
2     LEA     IntuiName(PC),A1 ; Nome libreria da aprire (intuition)
3     JSR     -$198(A6)       ; OldOpenLibrary - apri la lib
4     TST.L  D0              ; Errore?
5     BEQ.s  EXIT            ; Se sì, esci senza eseguire il codice
6     MOVE.L  D0,A6          ; IntuiBase in a6
7     jsr     -$186(A6)       ; ReThinkDisplay - Riordina i connotati degli
8                               ; schermi...

```

Questa operazione è analoga a quella svolta col WBView. Per ora non abbiamo ancora usato il blitter, ma nelle prossime lezioni ci saranno molte blittate, e siccome useremo questa startup, sarà utile predisporla per tale scopo. Basta accertarsi che il blitter non sia usato dal sistema operativo mentre lo stiamo usando noi, ed esiste una funzione della GfxLib in grado di far cessare l'utilizzo del blitter da parte del WorkBench:

```

1      jsr      -$1c8(a6)      ; OwnBlitter, che ci da l'esclusiva sul blitter
2                                ; impedendone l'uso al sistema operativo.

```

Al termine del listato, basterà chiamare la funzione che fa l'opposto, ossia riabilita l'uso del blitter da parte della `graphics.library`:

```

1      jsr      -$1ce(a6)      ; DisOwnBlitter, il sistema operativo ora
2                                ; può nuovamente usare il blitter

```

Queste due funzioni sono simili al Disable e all'Enable, che come abbiamo visto fermano il multitasking e gli interrupts di sistema e li riabilitano. In realtà esiste anche una funzione meno drastica del Disable, cioè il Forbid, il quale disabilita il multitasking lasciando in funzione gli interrupt di sistema; nessuno vieta di usare insieme Forbid e Disable, forse rende l'arresto del sistema meno brusco, proviamoli insieme:

```

1      move.l   4.w,a6          ; ExecBase in A6
2      JSR      -$84(a6)        ; FORBID - Disabilita il Multitasking
3      JSR      -$78(A6)        ; DISABLE - Disabilita anche gli interrupt
4                                ; del sistema operativo
5      ; routines
6
7      MOVE.L   4.w,A6          ; ExecBase in a6
8      JSR      -$7E(A6)        ; ENABLE - Abilita System Interrupts
9      JSR      -$8A(A6)        ; PERMIT - Abilita il multitasking

```

Ora l'Amiga non può protestare con un Guru Meditation o un SoftWare Failure dicendo che non l'avevamo avvertita che stavamo programmando l'hardware!

Dato che salviamo lo status di tutto, perché non salvare i valori dei registri dati ed indirizzi? Esiste una istruzione che viene usata prevalentemente per questo scopo, ed è il MOVEM. I registri però sono salvati nello STACK, ossia il registro A7, detto anche SR, che per ora abbiamo evitato di usare. Vediamo cosa è lo stack: pensate che si tratta di un registro analogo ad un registro indirizzi, non per niente è il registro A7, dunque il valore che contiene è un indirizzo, ossia PUNTA ad un indirizzo. Il fatto è che se modifichiamo l'indirizzo contenuto in A7 (o SP) l'Amiga impazzisce totalmente. Ma chi mette quell'indirizzo nello Stack Pointer? Dato che modificandolo si verifica il Guru/Software Failure, potrete intuire che è il sistema operativo che decide tale numero ogni reset, ed è lui a modificarlo quando serve. Sapendo come usarlo, però può esserci molto utile. Abbiamo visto nel corso come sia possibile indicare una zona di memoria con l'indirizzamento indiretto, ad esempio scrivendo:

```

1      lea      bitplane,a0
2      move.l   #$123,(a0)+
3      move.l   #$456,(a0)+

```

Abbiamo inserito i valori \$123 e \$456 nel bitplane agendo sul registro a0, dato che abbiamo fatto PUNTARE a0 al bitplane. Da questo listatino vediamo anche come sia possibile, con l'indirizzamento indiretto con post-incremento, inserire dati consecutivamente, uno dopo l'altro, nella zona di memoria. Cosa succederebbe se, dopo quelle istruzioni, scrivessimo:

```

1      move.l   -(a0),d0
2      move.l   -(a0),d1

```

Succederebbe che in d0 sarebbe copiato l'ultimo valore inserito, ossia \$456, mentre in d1 il primo, \$123, e a0 punterebbe di nuovo a bitplane. In pratica siamo "tornati indietro". Ebbene, immaginate di fare l'operazione opposta a questa: nel caso che abbiamo visto, c'è una zona di memoria, che abbiamo chiamato BITPLANE, e scriviamo da quell'indirizzo in avanti con i `move.l #xxx,(a0)+`

```

Bitplane
o----->

```

Poi, dopo un certo numero di istruzioni, `a0` punta a `bitplane+x`, cioè molto più avanti in memoria. Possiamo “riprendere” i valori che abbiamo “seminato” in questo campo con dei `move.l -(a0),xxx` che ci fanno tornare indietro fino a raggiungere nuovamente l’indirizzo di partenza `BITPLANE`. Ma attenzione! abbiamo raccolto i dati nell’ordine inverso rispetto a quello di immisione, infatti l’ultimo inserito è il primo ripreso. Lo stack punta ad un indirizzo in memoria, che serve da “campo” in cui seminare, ossia da zona dove salvare e riprendere dati. Bisogna stare attenti, però, che viene usato “all’indietro”, al contrario dell’esempio del `bitplane`. L’esigenza dello stack nasce con i primi CPU, ed è organizzato in questo modo: la memoria di un computer di solito viene riempita dalle locazioni più basse fino alle più alte, per esempio se abbiamo un computer con 512k di memoria e dobbiamo caricarci un file lungo 256k, saranno riempiti i primi 256k e rimarranno liberi i Kb dal 257 al 512. Volendo riservare uno spazio `STACK` per salvare dei dati generici, si è pensato di far partire questo spazio dalla fine della memoria circa, e di salvare i dati “all’indietro” verso la prima locazione di memoria, in modo da utilizzare al meglio la memoria:

```
ZERO -----FINE MEMORIA
Programmi ----->>      <<-----STACK
```

In questo modo lo stack non viene sovrascritto a meno che la memoria non sia proprio finita, e comunque i programmi sotto sistema operativo evitano tale scontro! Noi dobbiamo fare demo o giochi eseguibili dal sistema operativo, dunque dobbiamo usare lo stack in modo standard per non creare conflitti o sovrascritture. Se facessimo un programma in autoboot e senza bisogno di uscita, potremmo definire un’area nostra per lo stack, ma questo può generare problemi di compatibilità e per ora vi consiglio di non farlo. Vediamo infine come immettere e prelevare dati dallo `STACK`, cominciando con un esempio semplicissimo: salvare il contenuto del registro `D0`, e successivamente ripristinarlo.

```
1      MOVE.L  d0,-(SP)      ; salviamo d0 nello stack. NOTA: se dobbiamo
2                          ; salvare un solo registro, usiamo MOVE e
3                          ; non MOVEM, usato per Multipli registri.
4
5  ;      Routines che modificano D0
6
7      MOVE.L  (SP)+,d0      ; ripristiniamo il vecchio valore di d0
8                          ; prendendolo dallo STACK
```

Da notare che scrivere `MOVE.L d0,-(SP)` o `MOVE.L d0,-(A7)` è equivalente, in memoria viene assemblata la stessa sequenza binaria. Notiamo che il contenuto di `d0` viene copiato nell’indirizzo cui punta `SP`, e `SP` stesso viene a puntare una long più indietro. Poi `d0` viene modificato da varie routines, e quando vogliamo ottenere il suo vecchio valore basta riprenderlo dall’indirizzo in `SP`, e notate bene che con `(SP)+` riportiamo `SP` a puntare l’indirizzo che puntava prima di aver salvato `d0`, cioè siamo andati indietro di una long, poi siamo ritornati avanti ripescando il valore. Proviamo ora a salvare il valore di più registri:

```
1      MOVE.L  d0,-(SP)      ; salviamo d0 nello stack
2      MOVE.L  d1,-(SP)      ; salviamo d1 nello stack
3      MOVE.L  d2,-(SP)      ; salviamo d2 nello stack
4      MOVE.L  d3,-(SP)      ; salviamo d3 nello stack
5
6  ;      Routines che modificano d0,d1,d2,d3
7
8      MOVE.L  (SP)+,d3      ; ripristiniamo il vecchio valore di d3
9      MOVE.L  (SP)+,d2      ; ripristiniamo il vecchio valore di d2
10     MOVE.L  (SP)+,d1      ; ripristiniamo il vecchio valore di d1
11     MOVE.L  (SP)+,d0      ; ripristiniamo il vecchio valore di d0
12                          ; prendendoli dallo STACK
```

Notate che l’ultimo valore salvato è il primo che può essere ripescato, proprio per il fatto che andiamo indietro e poi ritorniamo avanti, leggendo dall’ultimo valore immesso al contrario fino al primo:

Indirizzo di partenza STACK
 IMMISSIONE: -(SP) <-----o - indietro -

Indirizzo di partenza STACK
 LETTURA: (SP)+ -----> o - avanti -

è una struttura detta a “pila”, infatti può essere immaginata in questo modo: pensate di avere una collezione di fumetti, e di volerla ordinare dal numero uno al numero 50. Trovato il numero 1, lo mettete su un tavolo. Trovato il numero 2 lo mettete sopra il numero 1. Poi il 3 sul 2, e via via fate una “pila” di fumetti, fino a che non avrete messo il numero 50 in cima alla catasta. Ora, se voleste riprendere i fumetti, il primi che si presenta è il 50, poi sotto trovate il 49, il 48 eccetera, e come ultimo trovate l’1. Lo stack infatti è del tipo “first in, last out”, ossia “il primo messo dentro è l’ultimo tirato fuori”. Capirete che modificando impropriamente lo stack vengono presi valori in memoria a caso e considerati come valori salvati prima. Dunque state attenti, quando avete eseguito un:

```
1    MOVE.L    xxxx, -(SP)      ; salviamo xxxx nello stack
```

La prossima volta che leggete con (SP)+ dallo stack ricavate xxxx.

Nello stack potete salvare e riprendere qualsiasi dato, ma una delle più evidenti utilità è quella di salvare lo stato dei registri, e per fare questo si può usare il semplice MOVE.L, nel caso già visto del salvataggio di un solo registro, oppure il MOVEM (MOVE Multiple) per più registri. Vediamo come funziona il MOVEM: per salvare tutti i registri (escluso a7, ovviamente, che è l’SP, dunque d0, d1, d2, d3, d4, d5, d6, d7, a0, a1, a2, a3, a4, a5, a6), si deve eseguire questo solo MOVEM anziché 15 MOVE:

```
1    MOVEM.L   d0-d7/a0-a6, -(SP)      ; salva tutti i registri nello STACK
```

E per ripristinarli tutti basta un:

```
1    MOVEM.L   (SP)+, d0-d7/a0-a6      ; riprendi tutti i registri dallo STACK
```

Praticamente il MOVEM sposta una lista di registri nella destinazione, nel caso del MOVEM.L d0-d7/a0-a6, destinazione, oppure copia una sorgente in vari registri, nel caso di MOVEM.L sorgente, d0-d7/a0-a6. La sorgente e la destinazione sono in formato “standard”, per cui si può copiare da e verso LABEL/INDIRIZZI o indirizzamenti indiretti:

```
1    MOVEM.L   d0-d7/a0-a6, -(SP)
2    MOVEM.L   d0-d7/a0-a6, LABEL
3    MOVEM.L   d0-d7/a0-a6, $53000
4
5    MOVEM.L   $53000, d0-d7/a0-a6
6    MOVEM.L   LABEL(PC), d0-d7/a0-a6
7    MOVEM.L   (SP)+, d0-d7/a0-a6
```

La lista segue questo standard: si possono indicare i registri separatamente, separandoli con la barretta “/”, per cui si può dire che:

```
1    MOVEM.L   d0-d7/a0-a6, -(SP)
```

è equivalente a:

```
1    MOVEM.L   d0/d1/d2/d3/d4/d5/d6/d7/a0/a1/a2/a3/a4/a5/a6, -(SP)
```

Ma, poiché le serie consecutive di registri possono essere indicati ponendo il primo registro della serie e l’ultimo separati da un “-”, si dividono con la barretta solamente i registri dati da quelli indirizzi. In realtà l’asmone accetta anche:

```
1    MOVEM.L   d0-a6, -(SP)
```


Considerandolo come la lunghissima istruzione precedente, ma dato che non tutti gli assembleri accettano questa forma, è meglio mettere la “/” tra le serie di registri dati e quella dei registri indirizzi. Facciamo degli esempi: vogliamo salvare i registri d0,d1,d2,d5 e a3:

```
1 MOVEM.L d0-d2/d5/a3,-(SP)
```

Abbiamo semplificato d0/d1/d2 con d0-d2. Proviamo ora a salvare d2,d4,d5,d6,a3,a4,a5,a6:

```
1 MOVEM.L d2/d4-d6/a3-a6,-(SP)
```

Chiaramente per ripristinare questi registri si scriverà:

```
1 MOVEM.L (SP)+,d2/d4-d6/a3-a6
```

Credo che la sintassi del MOVEM sia chiara. Tramite questa istruzione è possibile gestire il multitasking, infatti vi siete mai chiesti come è possibile far girare due programmi insieme, che usano gli stessi registri dati e indirizzi, senza che interferiscano fra loro: la risposta è semplice! All'inizio di ogni routine c'è un MOVEM che salva lo stato dei registri, la routine viene eseguita, e all'uscita i registri tornano al loro stato originario come se tale routine non fosse mai stata eseguita. Molte routines infatti sono così strutturate:

```
1 Routine:
2 MOVEM.L d0-d7/a0-a6,-(SP)
3 ....
4 ....
5 MOVEM.L (SP)+,d0-d7/a0-a6
6 rts
```

In questo modo, un BSR.W ROUTINE non causa la modifica dei registri, per cui se in a5 c'era \$dfff000 e in a6 ExecBase, siamo sicuri che dopo aver eseguito la routine ci sono sempre questi valori. Nell'usare il MOVEM capita spesso, le prime volte, di perdere “il filo” dei movem già fatti per cui può succedere una cosa del genere:

```
1 Routine:
2 MOVEM.L d0-d7/a0-a6,-(SP)
3 ....
4 ....
5 MOVEM.L (SP)+,d0-d7/a0-a6
6 ....
7 ....
8 MOVEM.L (SP)+,d0-d7/a0-a6
9 rts
```

In questo caso c'è un **errore madornale**, perché come prima cosa lo stack è andato troppo avanti, per cui tutti i dati che verranno ripresi dallo stack in seguito saranno sbagliati, come seconda cosa già i registri avranno valori diversi da quelli in entrata. Per far tornare tutto si potrebbe fare così:

```
1 Routine:
2 MOVEM.L d0-d7/a0-a6,-(SP)
3 ....
4 ....
5 ....
6 MOVEM.L d0-d7/a0-a6,-(SP)
7 ....
8 ....
9 MOVEM.L (SP)+,d0-d7/a0-a6
10 ....
11 ....
12 MOVEM.L (SP)+,d0-d7/a0-a6
13 rts
```

In questo modo all'uscita della routine i registri hanno il valore di entrata e lo stack è tornato all'indirizzo di entrata. (entrata nella routine!)

A questo punto possiamo dotare la nostra startup di salvataggio iniziale dei registri e ripristino finale, analogamente a questo ultimo esempio. Ecco come si presenta la nostra startup in questo momento:

```

1  MAINCODE:
2      movem.l d0-d7/a0-a6,-(SP)      ; Salva i registri nello stack
3      move.l 4.w,a6                  ; ExecBase in a6
4      LEA GfxName(PC),A1             ; Nome libreria da aprire
5      JSR -$198(A6)                  ; OldOpenLibrary - apri la lib
6      MOVE.L d0,GFXBASE              ; Salva il GfxBase in una label
7      BEQ.w EXIT2                    ; Se si, esci senza eseguire il codice
8      LEA IntuiName(PC),A1           ; Intuition.lib
9      JSR -$198(A6)                  ; Openlib
10     MOVE.L D0,IntuiBase
11     BEQ.w EXIT1                     ; Se zero, esci! Errore!
12
13     MOVE.L IntuiBase(PC),A0
14     CMP.W #39,$14(A0)              ; versione 39 o maggiore? (kick3.0+)
15     BLT.s VecchiaIntui
16     BSR.w ResetSpritesV39
17 VecchiaIntui:
18
19     MOVE.L GfxBase(PC),A6
20     MOVE.L $22(A6),WBVIEW           ; Salva il WBView attuale di sistema
21
22     SUBA.L A1,A1                    ; View nullo per azzerare il modo video
23     JSR -$DE(A6)                    ; LoadView nullo - modo video azzerato
24     SUBA.L A1,A1                    ; View nullo
25     JSR -$DE(A6)                    ; LoadView (due volte per sicurezza...)
26     JSR -$10E(A6)                   ; WaitOf ( Queste due chiamate a WaitOf )
27     JSR -$10E(A6)                   ; WaitOf ( servono a resettare l'interlace )
28     JSR -$10E(A6)                   ; Altre due, vah!
29     JSR -$10E(A6)
30
31     MOVEA.L 4.w,A6
32     SUBA.L A1,A1                    ; NULL task - trova questo task
33     JSR -$126(A6)                   ; findtask (d0=task, FindTask(name) in a1)
34     MOVEA.L D0,A1                   ; Task in a1
35     MOVEQ #127,D0                   ; Priorità in d0 (-128, +127) - MASSIMA!
36     JSR -$12C(A6)                   ; _LVOSetTaskPri (d0=priorità, a1=task)
37
38     MOVE.L GfxBase(PC),A6
39     jsr -$1c8(a6)                   ; OwnBlitter, che ci da l'esclusiva sul blitter
40                                     ; impedendone l'uso al sistema operativo.
41     jsr -$E4(A6)                    ; WaitBlit - Attende la fine di ogni blittata
42     JSR -$E4(A6)                    ; WaitBlit
43
44     move.l 4.w,a6                   ; ExecBase in A6
45     JSR -$84(a6)                    ; FORBID - Disabilita il Multitasking
46     JSR -$78(A6)                    ; DISABLE - Disabilita anche gli interrupt
47                                     ; del sistema operativo
48 *****
49     bsr.w HEAVYINIT                 ; Ora puoi eseguire la parte che opera
50 *****                             ; sui registri hardware
51
52     move.l 4.w,a6                   ; ExecBase in A6
53     JSR -$7E(A6)                    ; ENABLE - Abilita System Interrupts
54     JSR -$8A(A6)                    ; PERMIT - Abilita il multitasking
55
56     SUBA.L A1,A1                    ; NULL task - trova questo task
57     JSR -$126(A6)                   ; findtask (d0=task, FindTask(name) in a1)
58     MOVEA.L D0,A1                   ; Task in a1
59     MOVEQ #0,D0                     ; Priorità in d0 (-128, +127) - NORMALE
60     JSR -$12C(A6)                   ; _LVOSetTaskPri (d0=priorità, a1=task)
61
62     MOVE.W #$8040,$DFF096           ; abilita blit
63     BTST.b #6,$dff002               ; WaitBlit...
64 Wblittez:
65     BTST.b #6,$dff002
66     BNE.S Wblittez
67
68     MOVE.L GFXBASE(PC),A6           ; GFXBASE in A6
69     jsr -$E4(A6)                    ; Aspetta la fine di eventuali blittate
70     JSR -$E4(A6)                    ; WaitBlit
71     jsr -$1ce(a6)                   ; DisOwnBlitter, il sistema operativo ora
72                                     ; può nuovamente usare il blitter
73     MOVE.L IntuiBase(PC),A0

```

```

74      CMP.W  #39,$14(A0)      ; V39+?
75      BLT.s  Vecchissima
76      BSR.w  RimettiSprites
77  Vecchissima:
78
79      MOVE.L  GFXBASE(PC),A6   ; GFXBASE in A6
80      MOVE.L  $26(a6),$dff080 ; COP1LC - Punta la vecchia copper1 di sistema
81      MOVE.L  $32(a6),$dff084 ; COP2LC - Punta la vecchia copper2 di sistema
82      JSR     -$10E(A6)        ; WaitOf ( Risistema l'eventuale interlace)
83      JSR     -$10E(A6)        ; WaitOf
84      MOVE.L  WBVIEW(PC),A1    ; Vecchio WBVIEW in A1
85      JSR     -$DE(A6)         ; loadview - rimetti il vecchio View
86      JSR     -$10E(A6)        ; WaitOf ( Risistema l'eventuale interlace)
87      JSR     -$10E(A6)        ; WaitOf
88      MOVE.W  #$11,$DFF10C     ; Questo non lo ripristina da solo..!
89      MOVE.L  $26(a6),$dff080 ; COP1LC - Punta la vecchia copper1 di sistema
90      MOVE.L  $32(a6),$dff084 ; COP2LC - Punta la vecchia copper2 di sistema
91      moveq   #100,d7
92  RipuntLoop:
93      MOVE.L  $26(a6),$dff080 ; COP1LC - Punta la vecchia copper1 di sistema
94      move.w  d0,$dff088
95      dbra   d7,RipuntLoop    ; Per sicurezza...
96
97      MOVEA.L IntuiBase(PC),A6
98      JSR     -$186(A6)        ; LVORethinkDisplay - Ridisegna tutto il
99                                ; display, comprese ViewPorts e eventuali
100                               ; modi interlace o multisync.
101      MOVE.L  a6,A1            ; IntuiBase in a1 per chiudere la libreria
102      move.l  4.w,a6           ; ExecBase in A6
103      jsr     -$19E(a6)        ; CloseLibrary - intuition.library CHIUSA
104  Exit1:
105      MOVE.L  GfxBase(PC),A1   ; GfxBase in a1 per chiudere la libreria
106      jsr     -$19E(a6)        ; CloseLibrary - graphics.library CHIUSA
107  Exit2:
108      movem.l (SP)+,d0-d7/a0-a6 ; Riprendi i vecchi valori dei registri
109      RTS                                ; Torna all'ASMONE o al Dos/WorkBench

```

Ci sono solo quattro particolari aggiunti: uno è quello del controllo dopo l'apertura della `Graphics.library`, infatti se per qualche motivo non si potesse aprire, in `d0` anziché l'indirizzo del `GfxBase` troveremmo ZERO. Non si fa altro che uno pseudo TST.L `D0` e un salto alla label `EXIT` in caso di non apertura. Vedrete, con lo studio dei Condition Codes, come mai basti fare un `beq` dopo un `move`, senza usare il `tst`, per sapere se `d0` è azzerato. Un altro particolare è la comparsa della `COPPER2` di sistema, (`GfxBase+$32`) che non è altro che il valore inserito in `$dff084`, `COP2LC`, dal sistema operativo; per ora non abbiamo mai usato la `copperlist 2`, ma in certe lezioni più avanti non mancheremo di illustrarne i casi di utilità. Altra “finezza” è quella di resettare gli `sprites`, ma solo se siamo sul `kickstart 3.0` o superiori, dato che la funzione di `reset` degli `sprites` è disponibile da questa versione in avanti. La SubRoutine che reseta gli `sprites` è un classico esempio di programmazione “legale”, con chiamate al sistema operativo... come potrete notare sbirciandola è più complicato usare il sistema operativo che programmare via hardware (non è vero??). Infine c'è il settaggio della priorità del task. Come sapete ogni programma che viene eseguito in multitasking ha una sua “priorità” rispetto agli altri. Ebbene, mettiamola al massimo! Ossia 127. In realtà non servirebbe, dato che dopo disabilitiamo del tutto il multitasking, ma vedremo in seguito che è utile settare la priorità al massimo e riabilitare il multitasking per caricare files dati da dischetto, harddisk o CDrom.

Con questa startup facciamo il possibile affinché il sistema operativo possa essere “scavalcato” senza problemi. Vediamo ora cosa possiamo fare per prendere in modo più completo il controllo dell'hardware di Amiga. Innanzitutto vanno introdotti i registri `DMACON`, `INTENA`, `ADKCON` ed `INTREQ`, che sono dedicati alla “chiusura” o “apertura” dei CANALI DMA, nonché alla abilitazione degli interrupt e di altre cose. Per ora nei listati abbiamo dato per scontato che il `COPPER`, i `BITPLANES` e gli `SPRITE` sono abilitati, infatti possiamo vedere sia i testi e i menù dell'ASMONE (`BITPLANE`) che la freccia puntatore del mouse (`SPRITE`). Questo significa che tali canali sono

abilitati. Comunque è meglio modificare di persona lo stato di questi registri per essere sicuri che i canali che ci interessano siano abilitati, e quelli che non ci interessano non lo siano. Come facciamo per le copperlist, basterà salvare lo stato di questi registri all'inizio, poi eseguire il nostro codice che abilita e disabilita a volontà, infine rimettere i registri nello stato di partenza, come nulla fosse accaduto. Ma innanzitutto vediamo cosa sono questi canali DMA. DMA significa "Direct Memory Access", ossia "accesso diretto alla memoria". Infatti nell'Amiga l'accesso alla memoria è molto complesso, dato che ci deve accedere non solo il processore, ma anche il copper per visualizzare immagini, il blitter per copiarle e spostarle, l'audio per suonare. Per evitare che succedano "incidenti" a tutti questi processori che vogliono mettere le mani sulla memoria (almeno quella CHIP) tutti contemporaneamente, è stato messo un sistema di "semafori" e di viadotti, si può parlare di urbanistica e viabilità. Infatti nel chip AGNUS esiste un gestore dei canali DMA, il quale coordina le operazioni, facendo accedere i chip custom e il 68000 alla memoria "a turno", quando il canale è libero. Tale accesso può essere sia di lettura che di scrittura (il copper LEGGE le copperlist, l'audio LEGGE le musiche, il blitter però SCRIVE anche le immagini, e così via). Esistono vari canali DMA, ognuno dedicato ad un particolare scopo, vediamoli:

DMA COPPER Attraverso questo canale il copper legge la COPPERLIST. Se viene disabilitato la copperlist non viene più letta, di conseguenza spariscono sia i bitplane, che gli sprite, che le eventuali sfumature fatte modificando il colore di sfondo varie volte mettendo dei WAIT in copperlist. In pratica lo schermo rimane di colore unito, del colore COLOR0. In questo caso si può cambiare il colore dello schermo solo col processore, con `MOVE.W #xxx,$dff180`.

DMA SPRITE Questo canale esegue il trasferimento delle strutture sprite, quelle puntate nei registri SPRxPT in copperlist. Abbiamo comunque già visto come sia possibile visualizzare sprites scrivendo direttamente nei registri SPRxDAT, facendo manualmente il lavoro del DMA. Disabilitando il solo canale del DMA SPRITE, spariscono gli sprite come se fossero puntati a zero, e rimangono sullo schermo i bitplane e le sfumature ottenute con i WAIT e MOVE della copperlist. Da notare che se viene disattivato il DMA BITPLANE, anche tenendo il DMA SPRITE attivo gli sprite spariscono.

DMA BITPLANE Disabilitando questo canale i bitplane puntati nei BPLxPT non vengono più visualizzati, in compenso, però, se è attivo il canale DMA del copper le eventuali sfumature fatte col COLOR0 vengono visualizzate. Spegnerne questo canale può equivalere a mettere ZERO bitplanes nel BPLCON0, ossia il `dc.w $100,$200` in copperlist. Da notare che se viene disabilitato il DMA BITPLANE, gli sprites spariscono assieme ai bitplane, anche se il canale DMA SPRITE è attivo. Questo succedeva anche quando mettevamo zero bitplanes nel BPLCON0.

DMA DISCO Serve per il trasferimento dei dati dal drive alla memoria CHIP in fase di lettura o scrittura.

DMA AUDIO1...DMA AUDIO4 Si tratta di 4 canali separati che controllano le 4 voci stereo dell'Amiga. Per esempio, per emettere un suono dalla voce 1, occorre aprire il canale DMA AUDIO1, e per rendere muta tale voce basta richiudere tale canale DMA. Ovviamente i 4 canali sono sempre chiusi quando l'Amiga tace, ad esempio quando si utilizza il WorkBench senza musiche di sottofondo.

DMA BLITTER Questo DMA si occupa degli accessi in lettura e scrittura del blitter. Analizzeremo i canali DMA del blitter nella lezione dedicata a questo processore.

Ma come viene spartito il tempo di accesso alla memoria tra il processore e i chip custom? Dipende molto dalla risoluzione video e da quali canali sono abilitati. In pratica meno canali sono accesi e più vanno veloci il 68000 e gli altri CHIP in funzione. Vediamo il rapporto tra risoluzione video e DMA: l'immagine video è fatta di linee raster, cioè di linee disegnate dal pennello elettronico, che si chiama, appunto, raster. Sappiamo già come attendere una data linea verticale leggendo il \$dff006 (VHP0SR), oppure con la copperlist tramite un WAIT. Ebbene, in ogni linea raster sono possibili 227,5 accessi alla memoria, e il DMA ne utilizza solo 225. Un ciclo di accesso alla memoria, se vi interessa, ha la durata di 0,00000028131 secondi in uno schermo 320x256 PAL a 50Hz. Dato che il 68000 non farebbe in tempo ad accedere alla memoria ogni ciclo di BUS, gli sono concessi accessi solo durante i cicli pari, dunque 113 volte per linea raster. Il problema è che anche il Blitter ed il Copper possono accedere nei cicli pari, rubando cicli al povero 68000. I Cicli dispari sono utilizzati invece dal gestore DMA per gli accessi AUDIO, DISCO, SPRITE.

Riassumendo, ci sono 227/228 cicli per linea raster, divisi in cicli pari e cicli dispari. Nei 113 cicli dispari possono accedere alla memoria CHIP solo l'AUDIO, il DISCO, e gli SPRITE, a turno. Nei 113 cicli pari possono accedere alla memoria IL BLITTER, il COPPER e il 68000, a turno, dove però il povero 68000 ha poca priorità.

Capirete che se il blitter DMA è disattivato, il 68000 potrà accedere alla memoria più spesso, avendo più cicli pari liberi. Considerate che il DMA copper ha priorità sul DMA BLITTER, il quale a sua volta ha priorità sul 68000, che farebbe meglio a lavorare in FAST RAM. Infatti se il codice che il 68000 sta eseguendo è in FAST RAM anziché in CHIP RAM, il processore non subisce il minimo rallentamento. È per questo che conviene mettere il codice in fast ram con le SECTION CODE. Facciamo un esempio: se il copper stesse occupando il bus, sia il blitter che il 68000 dovrebbero attendere il prossimo ciclo pari. Il problema è che, mentre con una risoluzione di 320x256 LOWRES a 6 bitplanes il 68000 deve concedere "solo" la metà dei cicli pari al copper per visualizzare i 6 bitplanes, totalizzandone 56 per linea, nel caso di un 640x256 HIRES a 16 colori, ossia 4 bitplanes, il copper "ruba" quasi tutti i cicli pari al 68000, di conseguenza il programma rallenta (se non c'è FAST RAM sul computer). Gli accessi DMA durante la linea raster seguono un preciso schema: abbiamo visto che i cicli pari sono spartiti tra il COPPER, il BLITTER e il 68000. Nel caso dei cicli dispari, gli accessi per DISCO, AUDIO, SPRITE e BITPLANE seguono questo ordine: dalla linea orizzontale \$7 alla \$c avvengono gli accessi al DMA DISCO, dalla linea \$D alla \$14 quelli AUDIO, dalla \$15 alla \$34 quelli SPRITE, infine dalla \$35 alla \$e0 quelli per i BITPLANE. Riassumiamo:

- MAPPA DEGLI ACCESSI DMA IN OGNI LINEA RASTER -

CICLI PARI: Sono 113, e sono spartiti tra Copper, Blitter e 68000, dove il copper ha la priorità maggiore, per cui se siamo in una risoluzione alta, es. 640x256 a 4 bitplanes, il 68000 non può accedere quasi mai alla memoria, causando un rallentamento molto evidente. L'unico rimedio è porre il codice in fast ram, per cui non ci sono rallentamenti al processore. Tra l'altro nei processori 68020 e superiori il codice in fast ram è sempre molto più veloce di quello in CHIP RAM.

CICLI DISPARI: Sono 113, e sono spartiti tra Audio, Disco e Sprite in questo ordine:

linea orizzontale:

\\$07 - \\$0C	Accesso al DMA DISCO
\\$0d - \\$14	Accesso ai 4 canali DMA AUDIO
\\$15 - \\$34	Accesso agli 8 canali DMA SPRITE
\\$35 - \\$e0	Accesso ai bitplane in memoria

In realtà ai fini della programmazione non serve sapere questi particolari tecnici, ma possono far capire come sia importante fare economia di canali DMA per raggiungere la massima velocità operativa. Se per esempio in una vostra produzione avete una schermata in HAM o in HIRES nella parte alta dello schermo, mentre sotto fate girare altre cose in bassa risoluzione, considerate che per il periodo dalla prima linea alla fine della schermata “impegnativa” come DMA (es. hires a 16 colori) sia il processore che il blitter subiscono rallentamenti, e potrebbero non farcela nel tempo rimasto sotto la figura. Per acquistare velocità potreste innanzitutto attivare i 16 colori solo dove effettivamente servono, esempio:

```
----- inizio schermata, BPLCON0 settato per 16 colori HIRES
\ spazio NERO
/
*** #####          ***  ##  ##  ##  ##  ***          *** * #####
*** #####          ***  ##  ##  ##  ##  ***          *** * ##### > FIGURA
*** #####          ***  ##  ##  ##  ##  ***          *** * #####
\ spazio NERO
/
----- bplcon0 settato per risoluzione minore

**
                                **      > PALLINE E CUBETTI 3d ROTANTI
                                **
----- spazio nero

----- fine schermata, dc.w $ffff,$fffe
```

In questo caso vedete la cronaca di una copperlist. Supponiamo che alla routine 3d sotto la figura manchi proprio poco per essere eseguita in tempo per il cinquantesimo. Basta cambiare leggermente la copperlist e la routine potrebbe filare ad un fotogramma al secondo, vediamo cosa fare:

```
1  COPPERLIST
2      dc.w  $100,$200      ; 0 bitplanes nella zona "NERA" iniziale
3      dc.w  $3507,$fffe   ; aspetta la linea dove comincia la figura
4      dc.w  $100,$c200    ; attiva hires 16 colori
5      dc.w  $a007,$ffe    ; aspetta linea dove finisce la figura
6      dc.w  $100,$200     ; 0 bitplanes nella zona NERA sotto la pic
7      dc.w  $b007,$fffe   ; aspetta la fine della zona nera
8      dc.w  $100,$3200    ; 3 bitplanes lowres per routine vettoriale
9      dc.w  $e007,$fffe   ; sotto questa linea non arrivano le figure
10     dc.w  $100,$200     ; quindi spegnamo per bene il dma BITPLANE
11     ;                  ; e magari facciamo una sfumatura col COLORE e i WAIT,
12     ;                  ; per riempire la parte inferiore del monitor senza impegnare
13     ;                  ; il DMA
14     dc.w  $ffff,$fffe
```

Per esagerare, potremmo anche restringere la finestra video dove le figure non riempiono in senso orizzontale tutto lo schermo. Facciamo questo caso: abbiamo un solido 3d che ruota al centro dello schermo, e abbiamo già chiuso il dma bitplane sopra e sotto di esso:

```
----- inizio schermo, dc.w $100,$200

-----
      /\      ---- inizio solido, dc.w $100,$3200
     /\  \
    /\   \
   /\    \
  /\     \
 /\      \
/\       \
\_____/

----- fine solido, dc.w $100,$200
```

```
----- fine schermo, dc.w $ffff,$fffe
```

Come vedete, il solido ruota al centro dello schermo, e non occupa mai le zone di estrema destra ed estrema sinistra dello schermo. A questo punto, potremmo anche agire sul DIWStrt e DIWStop per "chiudere" un poco lo schermo, rendendolo largo solo il necessario, poi potremmo "riallargarlo" quanto serve per eventuali disegni più larghi sopra o sotto di esso:

```
1      dc.w    $8E,$2c81      ; DiwStrt LARGO normale per figura larga
2      dc.w    $90,$2cc1      ; DiwStop LARGO normale
3
4      WAIT
5
6      dc.w    $8E,$2c91      ; DiwStrt ristretto nella zona del solido
7      dc.w    $90,$2cb1      ; DiwStop
```

Infatti restringendo la finestra video risparmiamo tempo DMA, dato che il trasferimento dei bitplane avviene solo nella zona interna alla finestra video definita.

Chiudiamo questa parentesi, e vediamo come aprire e chiudere questi canali. Nell' Amiga esiste un registro hardware (\$dff096), denominato DMACon (=DMA Controller), che gestisce l'accensione di ogni singolo canale DMA. Il DMAConW (\$dff096) serve solo per *scrivere* eventuali modifiche, mentre il DMAConR (\$dff002) serve solamente per *leggere* i vari bit. Ecco la mappa dei 2 registri \$dff096 e \$dff002: (uguali ma uno per lettura e uno per scrittura). Il registro è *bitmapped* come il \$dff100 (BPLCON0) per cui conta quali bit sono accesi o spenti, singolarmente:

(NOTA: i bits 13 e 14 sono a sola lettura (R), il 15 a sola scrittura (W))

DMACon (\$dff096/\$dff002)

bit- 15 DMA Set/Clear	(W)	(si può solo scrivere dal \$dff096)
14 BlitBusy (o BlitDone)	(R)	(si può solo leggere dal \$dff002)
13 Blit Zero	(R)	(si può solo leggere)
12 X		(non usato)
11 X		(non usato)
10 BlitterNasty (BlitPri)	(R/W)	(R/W = Sia leggibili che scrivibili)
9 Master (DmaEnable)	(R/W)	- "interruttore generale"
8 DMA BitPlane (RASTER)	(R/W)	- detto anche BPLEN
7 DMA del Copper	(R/W)	- detto anche COPEN
6 DMA del Blitter	(R/W)	- detto anche BLTEN
5 DMA degli Sprite	(R/W)	- detto anche SPREN
4 DMA dei Dischi	(R/W)	- detto anche DSKEN
3 DMA Audio3 (voce 4)	(R/W)	- ossia AUD3EN
2 DMA Audio2 (voce 3)	(R/W)	- ossia AUD2EN
1 DMA Audio1 (voce 2)	(R/W)	- ossia AUD1EN
0 DMA Audio0 (voce 1)	(R/W)	- ossia AUD0EN

*SET/CLR

-Il bit 15 è importantissimo: se esso è acceso allora i bit settati a 1 in scrittura nel \$96 servono ad accendere i relativi DMA, se il bit 15 è a 0, allora gli altri bit a 1 nel registro servono a spegnere i relativi canali. Mi spiego meglio: per accendere o spegnere uno o più canali è comunque necessario impostare ad 1 i relativi bit; quello che determina se quei canali devono venir spenti od accesi è il bit 15: se è ad 1 si accendono, mentre a 0 si spengono (sempre indipendentemente dal loro precedente stato). Diciamo che si sceglie su quali OPERARE, poi si decide se spegnere(0) od accendere(1) in base al bit 15.

Facciamo un esempio:

```
          ;5432109876543210
move.w   #1000000111000000,$dff096      ; sono ACCESI i bits 6,7 e 8
```


Il valore \$7fff è %0111111111111111, quindi vengono resettati tutti i DMA-bit. Poi vengono impostati i DMA del Copper, dei Plane e del blitter, più quello master, grazie al bit 15 impostato ad 1 !

Il funzionamento di questo importantissimo registro è analogo a quello dei registri INTENA ed INTREQ, dunque non proseguite fino a che non avete più dubbi sulla funzione del bit 15 come “accendi/spegni” bit.

Nei listati che abbiamo visto fino ad ora non sono stati mai usati i registri \$dff096 (DMACON) e \$dff002 (DMACONR), perché abbiamo dato per scontato che i canali DMA del copper, dei bitplanes e degli sprites fossero abilitati. In effetti, se al momento dell'esecuzione del programma si può vedere lo schermo dell'asmone, significa che sia il dma COPPER che quello BITPLANE è abilitato. La presenza della freccia puntatore indica che essa è visualizzata con il DMA SPRITE. Ma programmando a livello hardware non si possono fare compromessi, non si deve “sperare” che sia tutto come vogliamo. Abbiamo già visto come sia importante settare TUTTI i registri della copperlist come il BPL1MOD, il DIWSTART/STOP eccetera, per evitare di trovarseli con valori strani. Lo stesso faremo con i canali DMA: ne salveremo lo stato all'inizio della startup, poi li spegneremo tutti e accenderemo solo quelli desiderati, e alla fine rimetteremo i canali DMA nello stato iniziale, proprio come facciamo per la copperlist. Abbiamo detto che per leggere lo stato del DMACON occorre leggere dal DMACONR, cioè il \$dff002. Una routine di “salvataggio” potrebbe essere:

```
1  move.w  $dff002,OLDDMA ; DMACONR — salvo lo stato del DMA
```

Ora possiamo cambiarlo a nostro piacimento agendo sul \$dff096, il registro per la scrittura:

```
1  move.w  #$7fff,$dff096 ; DMACON — azzero tutti i canali
2
3          ; 5432109876543210
4  move.w  #%1000001110100000,$dff096 ; Abilito Copper, Bitplane e Sprite
```

Niente di più facile. Ora dobbiamo rimettere a posto il vecchio valore, prima dell'uscita. Però ATTENZIONE! Non possiamo mettere OLDDMA direttamente nel DMACON (\$dff096) così come lo abbiamo letto dal DMACONR (\$dff002), perché il bit 15, quello SET/CLR, è di sola scrittura e in lettura è sempre a zero, dunque rimettendo il valore col bit 15 azzerato, i bit settati anziché accendere i canali DMA il spegnerebbero eventualmente. Serve dunque prima di settare il bit 15 del valore salvato in OLDDMA, in questo modo i bit settati varrebbero come ACCENSIONE. Ma come fare per settare il bit 15 di una word?? Ci sono infiniti modi. Uno sarebbe quello di usare l'istruzione BSET, ad esempio:

```
1  move.w  $dff002,d0      ; Salvo il DMACONR in d0
2  bset.l  #15,d0          ; setto il bit 15 (SET/CLR)
3  move.w  d0,OLDDMA      ; e salvo il valore in OLDDMA
4  ...
5  bsr.w   routines
6  ...
7  move.w  #$7fff,$dff096 ; azzero tutti i canali
8  move.w  OLDDMA(PC),$dff096 ; riattivo solo quelli che erano
9  rts      ; attivi all'inizio.
```

Altrimenti si può usare l'istruzione OR. Ricordiamo il suo effetto sui bit:

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

L'esempio di prima diventerebbe:

```
1      or.w    #$8000,OLDDMA    ; $8000 = %1000000000000000, ossia bit 15 ad 1
```

Come vedete dalla tabella sopra, i bit azzerati lasciano invariata la destinazione, in questo caso i primi 14 bit sono azzerati, dunque i primi 14 bit di OLDDMA dopo tale OR rimangono invariati (0 OR 0=0, 0 OR 1=1). Essendo il bit 15 settato, abbiamo che 1 OR 0=1, dunque viene settato il bit 15 e rimangono invariati gli altri 14 bit. Lo stesso che col BTST #15,d0. Nella startup conviene usare l'OR, perché sono salvati anche altri registri oltre al DMACON. Si tratta di INTENA (\$dff09a in scrittura e \$dff01c in lettura), INTREQ (\$dff09c in scrittura e \$dff01e in lettura) e ADKCON (\$dff09e in scrittura e \$dff010 in lettura). Per ora posso solo anticiparvi che tali registri sono bitmapped come il DMACON, e funzionano analogamente con il bit 15 che serve da SET/CLR. L'INTENA e L'INTREQ servono per gli interrupt, mentre ADKCON per compiti vari per il *disk drive* e l'*audio*. Vedremo come usare questi registri quando saranno trattati gli interrupt e l'audio, per ora salviamone lo stato assieme al DMACON. Vediamo ora come salvare questi 4 registri:

```
1      LEA     $DFF000,A5        ; Base dei registri CUSTOM per Offsets
2      MOVE.W  $2(A5),OLDDMA     ; DMACONR — Salva lo status del DMA
3      MOVE.W  $1C(A5),OLDINTENA ; Salva il vecchio status di INTENA
4      MOVE.W  $10(A5),OLDADKCON ; Salva il vecchio status di ADKCON
5      MOVE.W  $1E(A5),OLDINTREQ ; Salva il vecchio status di INTREQ
```

Ora dobbiamo settare il bit 15 di tutte e 4 le word contrassegnate dalle label OLDDMA, OLDINTENA, OLDADKCON, OLDINTREQ, per poter ripristinare il valore all'uscita. Considerate che le 4 label sono messe consecutivamente:

```
1  OLDDMA:      ; Vecchio status DMACON
2      dc.w     0
3  OLDINTENA:   ; Vecchio status INTENA
4      dc.w     0
5  OLDADKCON:   ; Vecchio status ADKCON
6      dc.w     0
7  OLDINTREQ:   ; Vecchio status INTREQ
8      dc.w     0
```

Dunque entra in gioco l'OR. se per una word basta fare un OR.w #\$8000,dest possiamo sistemare con un solo OR 2 word, con un OR.L #\$80008000,dest!!! In questo caso per 4 word bastano un paio di questi OR:

```
1      MOVE.L  #$80008000,d0      ; Prepara la maschera dei bit alti
2                                  ; da settare nelle word dove sono
3                                  ; stati salvati i registri
4      OR.L    d0,OLDDMA          ; Setta il bit 15 di tutti i valori salvati
5      OR.L    d0,OLDADKCON       ; dei registri hardware, indispensabile per
6                                  ; rimettere tali valori nei registri.
```

Ecco che con poche istruzioni abbiamo salvato e “settato” tutti e 4 i registri che andremo ad azzerare subito dopo:

```
1      MOVE.L  #$7FFF7FFF,$9A(a5) ; DISABILITA GLI INTERRUPTS & INTREQS
2      MOVE.L  #0,$144(A5)         ; SPRODAT — ammazza il puntatore!
3      MOVE.W  #$7FFF,$96(a5)      ; DISABILITA I DMA
```

A questo punto possiamo abilitare i soli canali DMA che ci servono. All'uscita basterà azzerare tutti i registri e ripristinarli:

```
1      MOVE.W  #$7FFF,$96(A5)      ; DISABILITA TUTTI I DMA
2      MOVE.L  #$7FFF7FFF,$9A(A5) ; DISABILITA GLI INTERRUPTS & INTREQS
3      MOVE.W  #$7fff,$9E(a5)      ; Disabilita i bit di ADKCON
4      MOVE.W  OLDADKCON(PC), $9E(A5) ; ADKCON
5      MOVE.W  OLDDMA(PC), $96(A5) ; Rimetti il vecchio status DMA
6      MOVE.W  OLDINTENA(PC), $9A(A5) ; INTENA STATUS
7      MOVE.W  OLDINTREQ(PC), $9C(A5) ; INTREQ
```

Niente di più semplice! Ora abbiamo il completo controllo dei canali DMA, e siamo sicuri che possiamo attivarli e disattivarli come ci pare, tanto all'uscita vengono ripristinati.

Per finire la nostra startup, potremmo definire un EQUATE. Ricordate cosa sono gli EQUATES? le direttive assembler EQU o =, che definiscono delle uguaglianze tra delle parole inventate a piacere e dei numeri, es:

```

1 CANE EQU 10
2 GATTO EQU 20
3
4 MOVE.L #CANE,d0 ; viene assemblato come MOVE.L #10,d0
5 MOVE.L #GATTO,d1 ; assemblato come MOVE.L #20,d1
6 ADD.L d0,d1 ; RISULTATO = 30
7 rts

```

Le equates sono simili alle label, ma non terminano con i :. Al posto di EQU si può usare l'uguale (=):

```

1 CANE = 10

```

Potremmo definire un EQU per i canali DMA da settare:

```

1 ;5432109876543210
2 DMASET EQU %1000001110000000 ; copper e bitplane DMA abilitati
3 ; -----a-bcdefghij
4
5 ; a: Blitter Nasty (Per ora non ci interessa, lasciamolo a zero)
6 ; b: Bitplane DMA (Se non è settato, spariscono anche gli sprite)
7 ; c: Copper DMA (Azzerandolo non è eseguita nemmeno la copperlist)
8 ; d: Blitter DMA (Per ora non ci interessa, azzeriamolo)
9 ; e: Sprite DMA (Azzerandolo spariscono solo gli 8 sprite)
10 ; f: Disk DMA (Per ora non ci interessa, azzeriamolo)
11 ; g-j: Audio 3-0 DMA (Azzeriamo lasciando muto l'Amiga)

```

Come vedete i bit 15 e il 9 devono essere **sempre settati**, dato che uno è il SET/CLR e l'altro il Master, l'interruttore generale. Nel listato si può mettere:

```

1 MOVE.W #DMASET,$96(a5) ; DMACON - abilita bitplane e copper

```

In questo modo possiamo avere all'inizio del listato l'EQU da modificare con sotto un breve sommario di aiuto con il significato dei bit.

Ma vediamo la startup, caricate in un buffer di testo Lezione8a.s e studiatelo. Nel commento finale sono riportate alcune note su certe modifiche minori.

Ora che abbiamo la startup "universale", possiamo anche metterla a parte in un file ed includerla all'inizio dei prossimi listati tramite la direttiva INCLUDE, che abbiamo già usato per includere la routine musicale. Basterà iniziare ogni listato con un:

```

1 Section UsoLaStartup.CODE
2
3 *****
4 include "startup1.s" ; con questo include mi risparmio di
5 ; riscriverla ogni volta!
6 *****

```

Da notare che Startup1.s è la startup senza il SECTION, per cui dobbiamo mettere la direttiva SECTION nome, CODE o CODE_C ogni volta prima dell'include. La Startup fa un BSR.S START, per cui inizieremo il listato con:

```

1 START:
2 MOVE.W #DMASET,$96(a5) ; DMACON - abilita bitplane, copper
3 ; e sprites.
4
5 move.l #COPPERLIST,$80(a5) ; Puntiamo la nostra COP
6 move.w d0,$88(a5) ; Facciamo partire la COP
7 move.w #0,$1fc(a5) ; Disattiva l'AGA
8 move.w #$c00,$106(a5) ; Disattiva l'AGA
9 move.w #$11,$10c(a5) ; Disattiva l'AGA

```

Considerate che in a5 è presente \$dff000. In questo caso lo ho sfruttato.

Sembrerebbe una startup perfetta, ma gli manca ancora la ciliegina sopra. Questa ciliegina è la possibilità di lanciare il programma da icona del WorkBench senza problemi. Infatti, fino a che facciamo partire da cli/shell i nostri programmi basta questa startup, ma nel caso si vogliano disegnare delle icone per farli partire da WorkBench col doppio click del mouse occorre aggiungere qualche istruzione. È solo una formalità burocratica, ma se non si fa con programmi grandi, che allocano pure la memoria, può capitare che all'uscita non tutta la memoria sia liberata, o anche peggio. Ecco cosa occorre aggiungere all'inizio:

```

1  ICONSTARTUP:
2      MOVEM.L D0/A0-A1/A4/A6,-(SP)      ; salva i registri nello stack
3      SUBA.L  A1,A1
4      MOVEA.L 4.w,A6
5      JSR      -$126(A6)                  ; _LVOfindTask(a6)
6      MOVEA.L  D0,A4
7      TST.L    $AC(A4)                    ; pr_CLI(a4) stiamo eseguendo dal CLI?
8      BNE.S    FROMCLI                    ; se si, salta le formalità
9      LEA      $5C(A4),A0                  ; pr_MsgPort
10     MOVEA.L  4.W,A6                      ; Execbase in a6
11     JSR      -$180(A6)                   ; _LVOWaitPort
12     LEA      $5C(A4),A0                  ; pr_MsgPort
13     JSR      -$174(A6)                   ; _LVOMsg
14     LEA      RETURNMSG(PC),A0
15     MOVE.L   D0,(A0)
16  FROMCLI:
17     MOVEM.L  (SP)+,D0/A0-A1/A4/A6        ; ripristina i registri dallo stack
18     BSR.w    MAINCODE                    ; esegui il nostro programma
19     MOVEM.L  D0/A6,-(SP)
20     LEA      RETURNMSG(PC),A6
21     TST.L    (A6)                        ; Eravamo partiti dal CLI?
22     BEQ.S    ExitToDos                    ; se si, salta le formalità
23     MOVEA.L  4.w,A6
24     JSR      -$84(A6)                    ; _LVOForbid - nota! Non serve il permit
25     MOVEA.L  RETURNMSG(PC),A1
26     JSR      -$17A(A6)                   ; _LVOREplyMsg
27  ExitToDos:
28     MOVEM.L  (SP)+,D0/A6                  ; exit code
29     MOVEQ    #0,D0
30     RTS
31
32  RETURNMSG:
33      dc.l    0

```

Non sto a commentare approfonditamente le chiamate alle librerie del sistema operativo, vi basti sapere che sono queste le formalità di cui parlavo. Se eseguite da workbench un programma che non ha questo codice all'inizio, il problema più grosso è quello che all'uscita da tale programma la memoria che ha occupato non viene liberata!!! Come potete notare, all'inizio si controlla se il programma è stato eseguito dal CLI o dal WorkBench, controllando un apposito flag di sistema. Se il programma è stato lanciato dal CLI, vengono saltate le formalità da seguire in caso di esecuzione da WB. Altrimenti, tali formalità sono eseguite. Anziché unire questo pezzo all'altra startup, conviene tenerla a parte, in modo da scegliere se includerlo o meno, dato che alcuni assembleri, inclusa la versione di ASMONE modificato del corso, al momento dell'esecuzione causano un loop infinito, dato che "sembrerebbe" caricato da WorkBench, ma poi al momento dell'esecuzione delle "formalità" sembrerebbe il contrario. Altre versioni di Asmone o altri assembleri invece eseguono tranquillamente questo codice, ma per compatibilità con ogni assembler si preferisce metterlo a parte:

```

1  ; Include "DaWorkBench.s" ; togliere il ; prima di salvare con "WO"

```

In questo modo, durante gli assemblaggi e le prove con <J> non lo includiamo, mentre prima di salvare l'eseguibile finale con <WO> lo facciamo includere.

Caricate la Lezione8b.s, il primo listato che utilizza la startup universale inclusa con INCLUDE. Comprende l'utilizzo sia dei bitplane che degli sprite, per cui potete fare delle prove per verificare l'abilitazione o meno dei canali DMA.

Vi siete spaventati trovando la NUOVA routine che attende la linea verticale? Ebbene, non c'è nulla di mostruoso, invece è molto migliore. Analizziamo la vecchia "routine":

```
1      cmp.b    #$xx,$dff006      ; VHPOSR
```

Ebbene, non facciamo che controllare il byte *\$dff006*, che contiene la posizione verticale del pennello elettronico, i bit da 0 a 7, ossia da \$00 a \$ff. Ma come sapete dalla gestione dei WAIT in copperlist, il pennello elettronico supera la linea \$FF, che in realtà non è che la 200 in uno schermo normale. Per raggiungere le posizioni oltre il \$FF con i WAIT del COPPER abbiamo visto che dobbiamo aspettare la fine di tale zona:

```
1      dc.w     $FFDF,$FFFE      ; aspetta il limite della zona NTSC
```

Dopodiché il contatore riparte da \$00

```
1      dc.w     $0007,$FFFE      ; aspetta linea $100
2      dc.w     $0107,$FFFE      ; aspetta la linea $FF+$01=$101
```

Fino a \$38. Ebbene, anche il byte in *\$dff006* si comporta in questo modo: una volta raggiunta la posizione \$ff riparte da \$00, indicando però \$100, e continua fino a \$138 (con \$38), dopodiché riparte da \$00, lo ZERO vero, per poi arrivare nuovamente a \$ff, per fare gli altri \$38, eccetera. È per questo che nei listati si attende sempre la linea \$FF, o la \$80, perché attendere la linea \$00 o la linea \$20 col *\$dff006* avrebbe significato eseguire la routine 2 volte per fotogramma, in quanto \$00 si verifica alla linea \$00 e alla linea \$100. Ma allora, come fare per aspettare tranquillamente le prime 38 linee, e le linee dopo la \$ff? Insomma, serve una routine che aspetti senza errori una qualsiasi delle 312 linee della scansione. Non è difficile, dato che il bit alto, l'ottavo, si trova molto vicino al *\$dff006*, esattamente al *\$dff005*, il byte prima. Dobbiamo fare come abbiamo fatto con la posizione verticale degli sprite, infatti abbiamo il bit alto a parte. In questo caso, però, non si trova in giro per la memoria, ma proprio prima del byte in questione. Analizziamo la situazione:

```
$dff004 byte che ora non ci interessa, contiene il bit LOF per l'interlace
$dff005 ci interessa! il bit 0 è il V8, ossia il bit alto della pos. vertic.
$dff006 ormai lo conosciamo! i bits V7-V0, gli 8 bit bassi della pos. vertic.
$dff007 contiene la posizione orizzontale (H8-H1). La risoluzione è 1/160
        dello schermo. Ora non ci interessa proprio!!!
```

Il *\$dff004/\$dff005* è il registro VPOSR, mentre il *\$dff006/\$dff007* è il VHPOSR ogni registro infatti è lungo una WORD. Possiamo però accedere ad essi come singoli bytes, in certi casi. Per attendere la linea \$100, possiamo fare così:

```
WaitVbl:
        btst.b   #0,$dff005
        beq.s    WaitVbl
```

Questa routine aspetta che il bit alto, il V8, sia settato. Se è settato significa che siamo alla linea \$100, o comunque dopo di essa. Per fare una routine UNIVERSALE, possiamo fare così: (a5=\$dff000)

```
1  Waity1:
2      MOVE.L    4(A5),D0          ; $dff004 e $dff006, ossia VPOSR e VHPOSR
3      LSR.L     #8,D0             ; sposta i bit di 8 posizioni a destra
4      AND.W     #%11111111,D0     ; Seleziona solo i bit della pos. verticale
5      CMP.W     #300,D0           ; linea 300? ($12c)
6      bne.s     Waity1
```

In questo caso abbiamo copiato *\$dff004/5/6/7* in d0, poi spostiamo il tutto di 8 bit a destra, dato che i primi 8 bit di destra sono occupati dalla posizione orizzontale del *\$dff007* che non ci interessa, portando la posizione verticale all'estrema destra. A questo punto, con un AND,

Se il video viene settato a frequenza NTSC (azzerando \$dff1dc), il limite massimo è la linea \$106.

Ora che abbiamo visto l'utilità delle istruzioni AND/OR/LSR per salvare i DMA e per controllare meglio la linea del VBLANK, veniamo a nuovi utilizzi di tali istruzioni logiche. Senza dubbio è il caso di analizzare come fare una routine di fade, ossia di dissolvenza di una figura dal nero, sfumando via via verso il colore pieno e lucente (e viceversa). Innanzitutto vediamo dove dobbiamo operare:

```
1 CopColors:
2     dc.w $180,0,$182,0,$184,0,$186,0
3     dc.w $188,0,$18a,0,$18c,0,$18e,0
4     ....
```

In copperlist, dove sono i registri colore. Quello che dobbiamo fare è mettere al posto di quegli zeri i valori RGB (esattamente la word: \$0RGB) giusti, “aumentandoli” in modo che con vari passaggi, uno per fotogramma, si innalzino fino a raggiungere i colori della nostra figura:

```
1 CopColors:
2     dc.w $180,$000,$182,$fff,$184,$200,$186,$310
3     dc.w $188,$410,$18a,$620,$18c,$841,$18e,$a73
4     ...
```

Innanzitutto dobbiamo avere la lista dei colori della figura in una tabella da “consultare”, altrimenti non sapremmo quando siamo “arrivati”:

```
1 TabColorPic:
2     dc.w $fff,$200,$310,$410,$620,$841,$a73,...
3     ...
```

(NOTA: il colore 0, il \$dff180, in questa tabella non è stato messo, dato che è nero, \$000, rimane sempre nero e non lo comprendiamo nella routine, partiamo invece dal colore 1, \$dff182, che in questo caso è \$FFF).

Per fare questa tabella basta togliere “a mano” i \$180,\$182,\$184 dalla copperlist copiata con l'editor, e ovviamente rimangono i colori.

Ora che abbiamo la tabella con i colori “destinazione”, come fare una routine che “aumenti” i colori fino a quelli giusti, nella tabella? Sicuramente dobbiamo lavorare separatamente per ognuna delle 3 componenti RGB, e per separarle possiamo usare l'AND, che come abbiamo visto “seleziona” solo una parte di bits, azzerando gli altri. Avendo l'indirizzo della tabella coi colori in a0, vediamo, per esempio, come separare la sola componente blu:

```
1     MOVE.W (A0),D4 ; Metti il colore dalla tabella colori in d4
2     AND.W  #$00f,D4 ; Seleziona solo la componente blu ($RGB->$00B)
```

Ora in d4 abbiamo solo il valore del BLU...se il colore era \$0123, in d4 dopo l'AND.w #\$000000001111,d4 (selezionati solo i 4 bits, o nibble), il valore è \$0003, dunque siamo riusciti nell'impresa. Vediamo come selezionare la componente verde:

```
1     AND.W  #$0f0,D4 ; Selez. solo la componente verde ($RGB->$0G0)
```

E quella rossa:

```
1     AND.W  #$f00,D4 ; Selez. solo la componente rossa ($RGB->$R00)
```

Fin qui dovrebbe essere tutto chiaro. Ora, potremmo già fare la routine “FAKE”, quella “FALSA”, che funziona in questo modo: ogni volta si aggiunge #1 ad ogni singola componente, e si compara con il colore in tabella, se dobbiamo smettere di aggiungere a quella componente. Per esempio, se abbiamo il colore \$0235 da raggiungere, avremo questi passaggi ogni fotogramma:

```
1 1) $111 ; +$111, tutti e 3
2 2) $222 ; +$111, tutti e 3
3 3) $233 ; +$011, la componente RED va bene, aggiungo solo a Gr. e Blu
4 4) $234 ; +$001, le componenti RED e GREEN vanno bene, +1 solo Blu
5 5) $235 ; +$001, come sopra, +1 solo blu
```

Ogni volta dovremmo comparare con un CMP la componente ROSSA del colore in tabella con quello che stiamo “aumentando”, se non la abbiamo raggiunta aggiungere 1, se la abbiamo raggiunta non aggiungere, poi fare lo stesso con il VERDE e il BLU, infine unire le 3 componenti “risultanti” tramite uno o più istruzioni OR, ottenendo la word colore risultante da mettere nella copperlist. E questo per ognuno dei 16, 32 colori o quanto sono. La quantità non è un problema per il processore, con dei cicli DBRA si può fare tutto. L'unico particolare è che il sistema che ho descritto non è esattissimo, e specialmente in AGA si vede che i colori vanno per i fatti suoi. Dunque, la struttura della routine rimane la stessa, ma dobbiamo cambiare il modo di calcolo. Dobbiamo prima notare una cosa: quanti fotogrammi, dunque quante volte dovremmo richiamare la routine per eseguire un fade completo? Se abbiamo il colore \$0F3, un bel verde, per esempio, partendo da \$000 e aggiungendo 1 ogni volta, con la routine di prima ci vorrebbero 15 add.w #\$010 per la componente verde, dato che deve raggiungere il valore \$f (15). Dunque, consideriamo di fare una routine “parametrica”, la quale possa calcolare i colori ad una delle 16 possibili FASI del fade, dove la fase 0 è il NERO completo, e la fase 16 è il colore pieno. Supponiamo di tenere “il conto” della fase da fare in una label FaseDelFade. Ogni volta dovremo fare:

```
1      addq.w  #1,FaseDelFade ; sistema per la prossima volta la fase da fare
```

Dunque, al primo fotogramma faremo un BSR.s Fade con FaseDelFade ad 1, e i colori saranno messi scurissimi, il fotogramma successivo richiameremo la routine, ma con FaseDelFade a 2, e i colori si schiariranno (2/16 del colore pieno), infine quando la eseguiamo con FaseDelFade a 3, i colori saranno uguali a quelli della tabella. Prima riferendomi alla frazione, 2 sedicesimi del colore, stavo anticipando la tecnica da usare! Infatti mentre una routine Fake, uno degli orrori di routine fade che aggiungono semplicemente 1 ogni volta non sono precise frazionariamente, quella che faremo è accettabile. Veniamo al dunque: con la routine Fake avremmo questi passaggi per arrivare ad un \$084:

```
$011
$022
$033
$044
$054
$064
$074
$084
```

Ebbene, quando si arriva a metà abbiamo un GRIGIO! \$044!! anziché un verdino. In realtà, a metà, saremmo dovuti essere a \$042, ossia al verdino scuro, che guarda caso è proprio 1/2 di \$084. Ora, ecco che si affaccia la soluzione: disponendo del valore FaseDelFade, che possiamo chiamare MULTIPLIER, abbiamo che quando è a 0, dobbiamo calcolare 0/16 (zero sedicesimi) dei colori, ossia tutti ZERO. D'altronde, quando è ad 1, dobbiamo calcolare 1/16 dei colori. Così fino a 16/16, in cui il colore rimane uguale. Come implementare in istruzioni tale formula? Facile! Disponendo di una componente RGB isolata, ad esempio la BLU: (abbiamo in a0 il MULTIPLIER)

```
1      MOVE.W  (A0),D4      ; Metti il colore dalla tabella colori in d4
2      AND.W   #$00f,D4     ; Seleziona solo la componente blu ($RGB->$00B)
3      MULU.W  D0,D4        ; Moltiplica per la fase del fade (0-16)
4      ASR.W   #4,D4        ; shift 4 BITS a destra, ossia divisione per 16
5      AND.W   #$00f,D4     ; Seleziona solo la componente BLU
6      MOVE.W  D4,D5        ; Salva la componente BLU in d5
```

In pratica moltiplichiamo per il MULTIPLIER la componente, poi la dividiamo per 16, in questo caso dividere per 16 equivale ad un ASR.W #4,Dx, come abbiamo già visto per la routine di print dei caratteri 8x8 che MULU.W #8,Dx si può sostituire da un LSL.w #3,Dx. Consideratelo

come una `DIVU.w #16,D4`, e tutto torna. Ripetendo 3 volte questo procedimento per le 3 componenti RGB, abbiamo la routine di FADE dal NERO ai colori, e se partiamo col multiplier a 16, sottraendo ogni volta #1 fino allo zero, avremo il fade contrario, dal colore al nero. Quest'ultimo si chiama FADE OUT, mentre il primo è il FADE IN.

Possiamo vedere in pratica il funzionamento della routine descritta nei due listati `Lezione8c.s` e `Lezione8d.s`. La differenza tra questi due listati è solo nell'ordine in cui vengono eseguite le operazioni di divisione delle 3 componenti RGB, ma il principio di moltiplicazione per il multiplier e divisione per 16 è lo stesso. La più chiara forse è quella di `Lezione8d.s`.

Il disegno è un logo del gruppo *RAM JAM*, fatto da FLENDER, che è italiano. Ho utilizzato questo disegno perché sono entrato a far parte di questo gruppo proprio mentre stavo scrivendo questa lezione. Dunque il corso da qua in avanti è una produzione *RAM JAM!!!*

Proseguiamo con una variazione sul tema, caricate `Lezione8e.s`. Questa è la stessa routine, con una lieve modifica che consiste nell'aggiunta di una componente dominante aggiuntiva, la quale fa assumere al disegno quella tonalità. Può essere utile per dare un'aria da carnevale al tutto.

Infine, sto per presentarvi una routine che può passare da qualsiasi colore a qualsiasi altro! In pratica occorrono due tabelle, una con i colori iniziali, per esempio se si partisse dal nero, una tabella con tanti zeri, e un'altra con i colori finali. Ne risulta che per fare il fade di prima, ossia dal nero ai colori normali, si deve mettere come prima tabella una tutta azzerata e come seconda quella coi colori, per passare dai colori al nero, (FADE OUT), come prima tabella si deve mettere quella coi colori normali, e come seconda una tutta azzerata.

A questo punto ecco le innovazioni: per esempio possiamo fare un fade dal BIANCO ai colori normali, mettendo come prima tabella tutti \$FFF e come seconda quella coi colori normali. Esageriamo: possiamo passare da una colorazione ad un'altra! Basta mettere nella prima tabella i colori come vogliamo che siano all'inizio, e come seconda come li vogliamo alla fine. In questo modo possiamo passare da una tonalità verde ad una bluastra, eccetera. Caricatevi `Lezione8f.s` e provate la routine, che mostra proprio gli esempi che ho fatto. Per quanto riguarda il funzionamento della routine, è piuttosto complicato, e non ho voglia di rivederla, se volete provare a capirla leggete i miei (pochi) vecchi commenti. Comunque almeno imparate ad usarla per i vostri scopi!

Ora voglio proporvi tre di listati, opera di altrettanti "allievi" partiti da zero con il mio corso, proprio come voi, incoraggiante no?

LEZIONE8g.s parallasse 10 livelli (di Federico Stango)

LEZIONE8h.s pannello di controllo con gadgets (di Michele Giannelli)

LEZIONE8h2.s scrolltext 8X8 (di Lorenzo Di Gaetano)

Questi tre listati utilizzano le sole conoscenze del disco 1 del corso. Io ho cambiato solamente la startup, inserendo la `startup1.s` al posto del vecchio modo di inizializzare del disco1. Spero che anche voi stiate facendo delle prove "autonome", altrimenti a che serve leggersi tutto come un romanzo? Sveglia!!! E se avete fatto qualcosa di carino vedete di spedirmelo, almeno lo metto nelle prossime lezioni e diventate famosi come Fiorello.

Proseguiamo ora con una domanda frequente: "Come funzionano gli equalizzatori presenti nella piccola demo `AMIGAET.EXE` del disco 1 del corso?". Ebbene, ho "tagliato" quel pezzo di listato, potete vedere come funziona il tutto in `lezione8i.s`.

Attenzione: la routine `music.s` del disco 2 non è la stessa di quella del disco 1. Le 2 modifiche sono la rimozione di un BUG che alle volte causava una guru all'uscita del programma, e il fatto

che `mt_data` è un puntatore alla musica, e non LA musica. Questo permette di cambiare la musica più facilmente, per creare music disks, come si vede in `lezione8i2.s`.

Siamo arrivati a fare gli equalizzatori, ma non abbiamo ancora visto come stampare un punto, ossia “plottare un dot”. Rimediamo subito con `Lezione8l.s`. (Poi mettere plottata di diversi planes da `3d_stars.s`)

Ok, ora che sappiamo stampare i punti, stampiamone tanti uno accanto all’altro per fare delle “linee”, in `Lezione8m.s` e `Lezione8m2.s`.

Ebbene, se si possono fare linee, si possono fare anche curve paraboliche, basta moltiplicare $X \times X$, in `Lezione8m3.s`, `Lezione8m4.s`, `Lezione8m5.s`.

Vediamo ora come “ottimizzare” la routine di stampa dei punti. Come avete visto, ha una moltiplicazione, il che è molto male perché le moltiplicazioni sono lente. Come fare per “toglierla”? Dobbiamo moltiplicare per 40, dunque basta “eseguire” tutte le moltiplicazioni possibili, cioè i primi 256 multipli di 40, e scrivere i risultati in una tabella. Ora abbiamo in questa tabella tutti i “risultati” della moltiplicazione in questione a seconda dei vari casi. Basta fare in modo che venga “preso” dalla tabella il risultato giusto ogni volta, come prendiamo la X o la Y giusta dalle tabelle delle coordinate per gli sprite. Vediamolo in pratica in `Lezione8n.s`.

Verifichiamo se effettivamente la nuova routine è più veloce di quella vecchia, scrivendo e cancellando tutto lo schermo, in `Lezione8n2.s`

Dato che abbiamo visto come azzerare un punto (basta mettere un `BCLR` al posto del `BSET`), proviamo ad “animare” un punto come abbiamo fatto per gli sprite, scrivendolo e cancellandolo ogni frame a posizioni diverse, in `Lezione8n3.s`.

Provate a farvi versioni modificate, a più biplanes, con più di un punto alla volta, eccetera. Per stampare su 2 bitplanes, ossia 4 colori, potete fare così: `color0` è lo sfondo, mentre abbiamo 3 colori diversi per plottare. Considerando di avere i 2 bitplanes con i nomi `Bitplane1` e `Bitplane2`, potreste farvi 3 routines, una che plotta nel `bitplane1`, una che plotta nel `bitplane2`, e una che plotta in entrambi i bitplanes, e saltare ad una di queste 3 routines per stampare in uno dei 3 colori.

Incredibile! Lorenzo di Gaetano ha scritto al volo un suo listato! vedetevelo: `Lezione8n4.s`

Mi immagino che abbiate fatto un programma che studia funzioni megacomplesse, che disegna onde come la sigla di Quark. Allora si può fare un breve stacco pubblicitario per i `wait` del copper, che non sono stati usati per le routines dei punti. Date un’occhiata a cosa possono fare dei semplici `wait` e `color0`, senza l’ausilio di nessun bitplane, in `Lezione8o.s`. Non ci sono trucchi, solo che la `copperlist` viene “costruita”, oltre che modificata, ecco la routine che “crea” il pezzo saliente della `copperlist`:

```

1 ; INITCOPPER crea la parte di copperlist con tanti WAIT e COLOR0 di seguito
2
3 INITCOPPER:
4     lea     barcopper,a0      ; Indirizzo dove creare la copperlist
5     move.l  #$3001fffe,d1    ; Prima wait: linea $30 – WAIT in d1
6     move.l  #$01800000,d2    ; COLOR0 in d2
7     move.w  #coplines-1,d0   ; numero di linee copper
8 initloop:
9     move.l  d1,(a0)+         ; metti il WAIT
10    move.l  d2,(a0)+         ; metti il COLOR0
11    add.l   #$02000000,d1    ; prossimo wait, aspetta 2 linee più in basso
12    dbra   d0,initloop
13    rts

```

Come si può vedere, il risultato di questa routine è di creare:

```

1 barcopper:
2     dc.l   $3001fffe        ; wait linea $30
3     dc.l   $01800000        ; color 0
4     dc.l   $3201fffe        ; wait linea $32
5     dc.l   $01800000        ; color 0
6     dc.l   $3401fffe        ; wait linea $34

```

```

7      dc.l    $01800000      ; color 0
8      ....

```

Pensate quanto spazio e quanto tempo risparmiamo in questo modo.

Per terminare la lezione, credo sia il caso di trattare una caratteristica del processore che, nonostante sia importantissima, fino ad ora non è stata discussa. Infatti **credevate** di sapere abbastanza sul 68000, ma in realtà fino ad ora è stato studiato “all’acqua di rose”, il minimo indispensabile per fare delle routines. Infatti non sono stati nominati i Codici di Condizione, e con essi il CCR, il quale fa parte dello SR (Status Register). Ecco i 16 bit che compongono il registro:

```

SR:

15      T - TRACE                --- \
14      - non usato dal 68000    |
13      S - SUPERVISOR          |
12      - non usato dal 68000    | - SYSTEM BYTE
11      -                        |
10      I2 \                     |
9       I1 > INTERRUPT MASK     |
8       I0 /                     --- /
7       -                        \
6       -                        |
5       -                        |
4       X - EXTENSION           | - USER BYTE (Condition Code Register)
3       N - NEGATIVE            | (contiene i flag aritmetici)
2       Z - ZERO                |
1       V - OVERFLOW (eccesso)  |
0       C - CARRY (RIPORTO)     --- /

```

Ebbene, questo misterioso registro contiene dei bit riguardanti i FLAG di condizione, per la precisione il suo byte basso, detto CCR (Condition Code Register) contiene tali FLAG. Il byte alto dello SR lo tratteremo più avanti, quando parleremo di INTERRUPT e di MODO SUPERVISORE. Per ora posso solo anticiparvi che il processore può funzionare in due modi, uno *UTENTE* (USER) e uno *SUPERVISORE*. Normalmente i programmi che scriviamo sono eseguiti in modo UTENTE. Quando ci serviranno gli interrupt vedremo come passare da modo Supervisor a modo User e viceversa, ma ricordatevi che alcune istruzioni si possono eseguire solamente in modo SUPERVISORE, se provate ad eseguirle in modo USER va tutto in coma profondo. Queste istruzioni sono dette PRIVILEGIATE, fate attenzione! Per ora ci basterà CAPIRE il byte basso dello SR, il CCR.

Ogni istruzione, agendo, può influenzare i flag, per esempio se una sottrazione da un risultato negativo si setta il flag N, se da zero si setta il flag Z, se una addizione ha come risultato un numero più grande, ad esempio, di quello contenibile in D0.1, si setterà il bit V, overflow, che indica l'impossibilità di contenere il risultato nella destinazione. Questo vale anche per il Carry, ossia il riporto, che si setta in caso di riporto. Si potrebbero controllare i flag stessi testando il byte CCR, ma siccome il 68000 è il miglior processore del mondo, sono presenti istruzioni a sufficienza per sapere lo stato dei flag: si tratta di Bcc, dove cc sta per Condition Codes e può essere sostituito con CS, EQ, GE, GT, HI, LE, LS, LT, MI, PL. . . Vi ricordate che parlando del modo di funzionamento delle istruzioni CMP seguite da BEQ e BNE giustificammo il fatto che il BEQ/BNE sapesse come era andata al CMP perché il risultato del CMP veniva scritta su un “foglietto”? Ebbene, il “foglietto” dove il CMP scrive il risultato per il BEQ/BNE è il CCR, il byte basso di SR!! In realtà questo foglietto è composto da 4 bit, più un quinto, detto eXtend, che serve a scopi particolari. Tramite questi 4 bit, si possono creare un bel pò di “situazioni”, non solo BEQ e BNE, ma si può sapere se un numero è più grande o più piccolo di un altro, se due numeri sono uguali, se si verifica un riporto in una operazione, se il risultato è negativo, eccetera. Ecco tutti i Bcc:

```

1      bhi.s label ; > per numeri senza segno
2      bgt.w label ; > per num. con segno
3      bcc.s label ; > detto anche BHS, Carry = 0 (senza segno)
4      bge.s label ; >= per num. con segno
5      beq.s label ; = per tutti i numeri
6      bne.w label ; <> per tutti i numeri
7      bls.w label ; <= per num. senza segno
8      ble.w label ; <= per num. con segno
9      bcs.w label ; < per num. senza segno; detto anche BLO,
10     ; significa che il Carry = 1
11     blt.w label ; < per num. con segno
12     bpl.w label ; Se Negative = 0 (PLus)
13     bmi.s label ; Se Negative = 1, (Minus) num. con segno
14     bvc.w label ; V=0, no OVERFLOW (risultato contenibile)
15     bvs.s label ; V=1 OVERFLOW (risultato troppo grande per
16     ; essere contenuto nella destinazione)

```

Ora vediamo come usare i Bcc dopo CMP.x OP1,OP2

```

1      beq.s label ; OP2 = OP1 - per tutti i numeri
2      bne.w label ; OP2 <> OP1 - per tutti i numeri
3      bhi.s label ; OP2 > OP1 - senza segno
4      bgt.w label ; OP2 > OP1 - con SEGNO
5      bcc.s label ; OP2 >= OP1 - senza segno, detto anche *BHS*
6      bge.s label ; OP2 >= OP1 - con SEGNO
7      bls.w label ; OP2 <= OP1 - senza segno
8      ble.w label ; OP2 <= OP1 - con SEGNO
9      bcs.w label ; OP2 < OP1 - senza segno, detto anche *BLO*
10     blt.w label ; OP2 < OP1 - con SEGNO

```

Ed ora come usarli dopo un TST.x OP1

```

1      beq.s label ; OP1 = 0 - per tutti i numeri
2      bne.w label ; OP1 <> 0 - per tutti i numeri
3      bgt.w label ; OP1 > 0 - con SEGNO
4      bpl.s label ; OP1 >= 0 - con SEGNO (oppure BGE)
5      ble.w label ; OP1 <= 0 - con SEGNO
6      bmi.w label ; OP1 < 0 - con SEGNO (oppure BLT)

```

Come si vede dopo un CMP si possono sapere un bel pò di cose! Si possono notare i segni > (maggiore), >= (maggiore o uguale), =, <> (diverso), <= (minore o uguale), < (minore), e per di più esiste un Bcc di queste comparazioni per numeri normali, e uno per i numeri Signed (con segno). Per quanto riguarda i numeri negativi, fino ad ora abbiamo solo accennato che, ad esempio, -1 è \$FFFFFFF, -5 è \$FFFFFFB, stabilendo più o meno che il bit alto, ossia il 31 se siamo in longword, il 15 se in .w e il 7 se in .b, è quello del segno, ossia che se è ad 1 il numero è negativo, e procede come se tornasse “indietro” da \$FFFF che è -1, a \$FFFE che è -2, \$FFFD per -3 eccetera, fino ad arrivare (in campo .w) a \$8001, cioè -32767, seguito da \$8000, cioè -32768, che è il numero più negativo possibile in una word con segno, e corrisponde a %1000000000000000, ossia il bit alto del segno settato e gli altri tutti azzerati: eravamo partiti da -1, cioè %111111111111111. Questo sistema usato per avere numeri negativi in binario è detto complemento a due. Sappiamo già che il bit più significativo, ossia quello più a sinistra, rappresenta il segno: se = 0 è positivo, se = 1 è negativo. Questo sistema vale sia per numeri .byte (il bit è il 7), che per quelli .word (il bit è il 15), che per quelli .longword (il bit è il 31). Vediamo ora in dettaglio come funziona il complemento a due: abbiamo notato che non basta cambiare il bit più significativo per passare da positivo a negativo, facciamo l'esempio di +26 e -26 in campo .word:

```

;5432109876543210
+26  %0000000000011010    ($001A)
-26  %1111111111100110    ($FFE6)

```

Il bit 15 in +26 è azzerato e in -26 è settato, ma non è evidentemente l'unica modifica da fare per passare da -26 a +26!!! Occorre fare il complemento a due di %0000000000011010, che

consiste nell'*invertire* tutti i bit e *aggiungere* 1 al risultato. Proviamo se è vero: invertendo tutti i bit si ottiene:

```
%1111111111100101
```

Aggiungiamo 1:

```
%1111111111100101 +
                   1 =
-----
%1111111111100110
```

Se vi confonde la fila di 1, isolate i 6 bit bassi: %100101 è 25, se si aggiunge 1 = %100110, cioè 26, con i bit dal 7 al 15 tutti ad 1, cioè -26. Se vogliamo -26 in un byte, basta %11100110, ossia \$E6. Se vogliamo -26 in una long: %11111111111111111111111111100110 = \$FFFFFFE6. Si può scegliere di usare i nostri byte, word o long come vogliamo, con segno o senza segno, dipende dalle istruzioni che usiamo e dal nostro programma. Per chiarire, ecco quanto può contenere un .b, un .w o un .l a seconda del sistema usato, se “normale” o “complemento a 2”:

```
BYTE con segno .8 bit - da -128 ($80) a +127 ($7f)
BYTE senza segno .8 bit - da 0 ($00) a 255 ($ff)
WORD con segno .16 bit - da -32768 ($8000) a +32767 ($7fff)
WORD senza segno .16 bit - da 0 ($0000) a 65535 ($ffff)
LONG con segno .32 bit - da -2147483648 ($80000000) a +2147483648 ($7fffffff)
LONG senza segno .32 bit - da 0 ($00000000) a 4294967295 ($ffffffff)
```

Come si può vedere, nel campo SIGNED BYTE i numeri da 128 a 255 sono considerati come i valori da -128 e -1, in campo SIGNED WORD i valori che vanno da 32768 a 65535 sono considerati come i valori da -32768 e -1. Lo stesso valore per la notazione .longword. Ricapitolando, ecco 2 sistemi per ottenere un numero negativo dal positivo:

Sistema 1: Dato il numero N=%00110 (6 decimale) per trovare -N si esegue la negazione bit a bit di N, ottenendo N=%11001 (-7 decimale) e poi si somma 1 al risultato:

```
N=%11001+%00001=%11010 (-6 decimale)
```

Sistema 2: Dato il numero N=%00110 (6 decimale) per trovare -N si esegue la negazione bit a bit di N fino all'1 meno significativo N=%11010 (-6 decimale).

Se nella nostra routine non accade mai di scendere sotto lo zero, è bene usare un byte per i suoi 255 valori, se invece volessimo andare da -50 a +50, occorre usare istruzioni come BGT, BLE, BLT, che confrontano numeri con segno, al posto di BHI e BLS, ad esempio, che confrontano numeri senza segno. Addizioni e sottrazioni funzionano sia con numeri con segno che senza segno, mentre le moltiplicazioni e le divisioni no, infatti sono presenti due tipi di istruzione per numeri con o senza segno: MULU e DIVU per i numeri senza segno, MULS e DIVS per i numeri con segno.

Chiariti i numeri negativi, vediamo i bit del CCR, ossia i flag, uno ad uno:

bit 0 - Carry (C) settato ad 1 quando il risultato di un'addizione genera un riporto ('carry'), o quando un sottraendo è maggiore del minuendo, cioè quando una sottrazione ha richiesto un "prestito". Il bit di Carry contiene inoltre il bit più o meno significativo di un operando sottoposto ad uno shift o ad una rotazione. Viene posto a zero quando l'ultima operazione eseguita non ci sono né riporti, né "prestiti". Per esempio, un modo per settare il flag CARRY è questo:

```

1      move.l  #$FFFFFF, d0
2      ADDQ.L  #1, d0

```

Il risultato è d0=00000000, con il flag CARRY e ZERO settati, perché abbiamo ecceduto il massimo contenibile in .l, e il risultato è pure ZERO!

bit 1 - Overflow (V) viene settato se il risultato dell'ultima operazione tra numeri dotati di segno il risultato è troppo grande per poter essere contenuto nell'operando destinazione, ad esempio se tale risultato supera i limiti -128...+127 nel campo byte. Ad esempio, la somma.b 80+80 genera un oVerflow, avendo superato +127. In campo .w i limiti sono -32768...+32767, e in campo .l sono -/+ 2 miliardi. Da notare che la somma 80+80 in campo byte non setta il flag di Carry ed eXtend, ma solamente quello di oVerflow, dato che 160 non supera 255, il massimo contenibile in un byte per numeri normali.

bit 2 - Zero (Z) settato quando l'operazione genera il risultato zero (utile anche per controllare il decremento di un contatore), nonché quando si confrontano due operandi uguali.

bit 3 - Negative (N) viene settato ad 1 se in una operazione il bit alto del numero, in formato complemento a due, è settato. In pratica se il risultato è un numero negativo questo bit è settato, altrimenti azzerato. Il complemento a due si ottiene facendo il complemento a uno dell'operando (ossia invertendo tutti i bit), aggiungendo quindi 1; ad esempio, +26 in binario è %000110010; il suo complemento a uno è %11100101 (inversione dei bit 0 in bit 1 e viceversa); aggiungendo 1 si ottiene %11100110. Il bit 7, detto bit di segno, viene copiato nel bit 3 dello Status Register; Nel caso di -26, ad esempio, N viene settato, indicando un numero negativo.

bit 4 - Extend (X) è una ripetizione del bit di Carry, e viene usato in operazioni effettuate in notazione BCD (Binary Coded Decimal: il numero decimale 20, ad esempio, non viene rappresentato con 00010100, ma nella forma due decine, zero unità 0010 0000) ed in operazioni binarie "estese" come ADDX e SUBX, versioni particolari delle istruzioni di addizione e sottrazione ADD e SUB.

Alla luce di queste nuove conoscenze, vedetevi il testo di riferimento su tutte le istruzioni del processore, con relativi effetti sui FLAG del CCR, il 68000-2.TXT, un'"evoluzione" del vecchio 68000.TXT del primo disco, che ormai è roba da bambini per voi (o no?).

Prima di iniziare la LEZIONE9.TXT, sarebbe bene che vi leggeste tutto il 68000-2.TXT, almeno sarete veramente ferrati sulle istruzioni della CPU! Consideratelo come una LEZIONE8b.TXT, "fatevelo" tutto, carpitene l'essenza. Ammetto che può spaventarvi (se siete mezze cartucce) leggerlo tutto, ma una volta presa confidenza con quello che è scritto in quel bel testo da 100K potrete finalmente dire in giro che sapete programmare il 68000. Tra l'altro, se più avanti trovate istruzioni che non conoscete, non potete lamentarvi, perché sono spiegate nel 68000-2.TXT!!

Come prima cosa, andatevi a vedere le istruzioni CMP e Bcc, in cui i vari tipi di Bcc sono spiegati più diffusamente, poi partite dall'inizio e arrivate alla fine, magari rileggendolo più volte, facendo delle pause tra una lettura e l'altra, mangiandovi un panino. Questo 68000-2.TXT è il secondo macigno che dovete superare; il primo era la LEZIONE2.TXT dove avete imparato le prime basi, gli indirizzamenti. Molti si sono fermati a quella collina. Ora vi si presenta una montagna, e altrettanti non avranno il fegato per superarla. Ma chi la supererà, potrà cercare di arrivare alla vetta!

Lo avete letto almeno una volta? Avete chiari i *Condition Codes*? Ecco degli esempi che verificheranno se avete capito. Sono stati gentilmente scritti da Luca forlizzi (the Dark Coder) e da Antonello Pardi (Deathbringer), permettendomi di velocizzare la scrittura delle lezioni sull'AGA e sul 3d.

```

Lezione8p1a.s  -> CC nell'istruzione MOVE
Lezione8p1b.s  -> CC in MULU/MULS
Lezione8p1c.s  -> CC in DIVU/DIVS
Lezione8p2a.s  -> CC e registri indirizzi Ax
Lezione8p2b.s  -> Estensione del segno nei registri indirizzi Ax
Lezione8p3.s   -> CC in TST
Lezione8p4.s   -> CC in AND,NOT,OR,EOR
Lezione8p5.s   -> CC in NEG
Lezione8p6.s   -> CC in ADD
Lezione8p7.s   -> CC in CMP
Lezione8p8.s   -> CC in ADDX
Lezione8p9.s   -> CC in lsr,asr,lsl,asl

```

Per finire caricate il mio Lezione8p9b.s, che contiene anche un “quesito”.

Prima di passare alla prossima lezione, ci sarebbero un paio di cose che vorrei dirvi. Quel mio amico che programma l'avventura, Michele, mi ha chiesto delle cose l'ultima volta che mi è venuto a trovare, e suppongo che potrebbero interessare anche a molti di voi. Lui ha fatto un pannello di controllo in basso, simile a Lezione8h.s, e nella parte alta visualizza le varie figure, che carica dal dischetto (vedremo più avanti come caricare files con la libreria di sistema dos.library). Il problema è che aveva i .raw delle figure, ma la palette di ogni figura la doveva tenere nel programma principale in tabelle preparate, una per figura, e una routine si occupava di copiare i colori della tabella giusta in copperlist a seconda della figura caricata. Questo però ingarbugliava il codice, dato che le figure sono dozzine. Allora mi sono ricordato che con gli iffconverter, compreso il KefCon, si può salvare ANCHE LA PALETTE in fondo al .RAW! Basta cambiare l'opzione CMAP OFF in BEHIND, e in fondo al .raw viene attaccata la palette, dal color0 all'ultimo, word dopo word. Si potrebbe anche scegliere BEFORE, che attacca la palette prima della pic, ma in tal caso occorrerebbe puntare a “dopo la palette”, saltandola. Stabilito che conviene salvare con la CMAP BEHIND (in fondo), vediamo cosa cambia nel file .raw salvato.

Il file è uguale, ma più lungo, nel caso del logo di questa lezione è più lungo di 16 words, infatti ha 16 colori .w in fondo, come in questo esempio (per capirci):

```

1  inizio_pic:
2      incbin  'logo320*84*16c.raw'      ; bitplanes.raw normali
3      dc.w $000,$fff,$200,$310          ; palette
4      dc.w $410,$620,$841,$a73
5      dc.w $b95,$db6,$dc7,$111
6      dc.w $222,$334,$99b,$446
7  fine_pic:

```

Ho opportunamente risalvato il logo in questo formato, vediamo con quale semplice routine si può copiare la palette in copperlist, in Lezione8q.s. Da notare che la pic se puntata normalmente funziona anche nei listati precedenti, infatti abbiamo solo delle word “in più” che non sono visualizzate essendo “dopo” la fine dell'ultimo bitplane.

Altra cosa che mi è stata chiesta, è: come si fa a sapere quale processore e quale kickstart è presente sulla macchina? In Lezione8r.s questo mistero è svelato...basta consultare degli appositi bit dedicati allo scopo!

Dunque, se siete convinti di aver capito ogni cosa fino a qui, potete passare a caricare la LEZIONE9.TXT, che vi introdurrà FINALMENTE al blitter, che a questo punto sicuramente stavate domandandovi se veramente esiste.

Una nota: se sapete leggere l'inglese, vi sarà certamente utile avere questi libri fondamentali:

- La seconda edizione del manuale dell'hardware di Amiga: “Amiga Hardware Reference Manual” codice ISBN: 0-201-18157-6
- PER QUANTO RIGUARDA IL 680x0:

- Motorola, “MC68020 32-bit Microprocessor User Manual, fourth edition”, Prentice Hall ISBN 0-13541657-4
- Motorola, “MC68030 Enhanced 32-bit Microprocessor User Manual, second edition” Prentice Hall ISBN 0-13-566951-0, Motorola ISBN 0-13-566969-3.
- Motorola, “MC68040 32-bit Microprocessor User Manual”

Forse non vi conviene prendere l’user manual del 68000, né quello del 68040, dato che il 68000 è spiegato (spero) abbastanza bene nel 68000-2.txt, e il 68040 per ora è appannaggio di pochi fortunati, dunque demo o giochi che vanno su solo 68040 avrebbero poca diffusione. Inoltre le differenze maggiori ci sono tra 68000 e 68020, mentre tra 68020 e 68030 le differenze sono poche, lo stesso vale per il 68030 rispetto al 68040. Le maggiori differenze comunque sono nella MMU e nelle istruzioni di controllo delle CACHE, ma programmando demo e NON sistemi operativi ciò non ci interessa più di tanto.

LEZIONE 9 - IL BLITTER

Autori: Luca Forlizzi, Alvise Spanò, Fabio Ciucci

(Directory Sorgenti5) - quindi scrivere `<V Assembler2:sorgenti5>`

9.1 IL BLITTER

In questa lezione inizieremo a parlare del blitter. Chiunque possieda un Amiga avrà sicuramente sentito parlare di questo speciale circuito posto all'interno del suo computer che risulta esserne uno dei maggiori punti di forza qualora lo si confronti con altri computer. Non tutti però sanno che cosa il blitter sia in realtà e per quali motivi sia così tanto utile. In effetti la maggioranza degli effetti speciali che potete ammirare nelle demo (come per esempio gli scrolltext sinusoidali o i vectorballs) fanno uso del blitter. E allora, vi chiederete, come mai si possono realizzare questi effetti anche sui dei PC che non hanno il blitter? Il motivo è che in realtà tutto ciò che il blitter può fare, potrebbe essere fatto con il microprocessore, ed è appunto in questo modo che fanno i PC. Il blitter, però è in grado di svolgere i suoi compiti in maniera molto più veloce, in certi casi anche 10 volte più veloce. È grazie al blitter che effetti speciali che con un PC si possono realizzare solo se si ha a disposizione un 386 veloce o addirittura un 486, mentre sono ordinaria amministrazione per un Amiga 500 il cui processore (68000 a 7Mhz come sapete bene) è molto più lento dei 386 e 486. Quindi capirete che per chi voglia programmare demo o giochi sull'Amiga, la conoscenza del blitter sia indispensabile. Cominceremo lo studio delle capacità del blitter partendo dalle più semplici, che a prima vista potrebbero sembrare misere, ma che scopriremo via via nascondere la potenza che ha permesso la creazione dei giochi e delle demo più spettacolari. Occorre però notare che i programmi scritti per 68020+ spesso tendono ad usare la CPU anziché il blitter, dato che quest'ultimo non aumenta di velocità.

9.2 Funzioni del blitter

La parola *blitter* è un'abbreviazione di *B*lock *I*mage *T*ransfer*ER* ovvero "copiatore di blocchi di immagine". Il blitter è dunque uno strumento che ci permette di spostare "pezzi" di immagini. In realtà, come scoprirete in seguito, questa è solo una delle capacità del blitter, che è in grado di

effettuare anche operazioni più complesse. Come sapete, un'immagine all'interno dell'Amiga è costituita semplicemente da una zona di memoria che contiene i dati che definiscono il colore di ogni singolo pixel. Se non vi ricordate bene come sono formate le immagini è bene che andiate a ripassare le lezioni 4 e 5 prima di proseguire oltre. Quando il blitter effettua un'operazione su un "pezzo" di immagine, lavora in realtà sulla zona di memoria che forma il "pezzo" di immagine in questione. In effetti il blitter opera semplicemente su zone di memoria, indipendentemente dal fatto che esse contengano un'immagine grafica, un suono o il codice di un programma. Questo significa che il blitter può essere utilizzato anche in compiti che non riguardano la grafica. È importante precisare, però che il blitter, al pari del copper dei circuiti audio e di tutto il resto dei chip "custom" dell'Amiga, non è in grado di operare su tutta la memoria disponibile, ma solo su una parte di essa denominata "chip ram".

Per accedere alla memoria il blitter utilizza i canali DMA di cui si è parlato in termini generici nella lezione 8, a cui vi rimando in caso di dubbi. Il blitter ha a disposizione ben 4 canali DMA, di cui 3 (denominati A, B e C) servono per *leggere* dati dalla RAM (e per questo vengono detti canali "sorgente") mentre il quarto (canale D) serve per *scrivere* nella memoria (e pertanto è detto canale "destinazione"). Come tutti i canali DMA, quelli del blitter trasferiscono una word di dati alla volta.

Lo schema generale di un'operazione blitter (detta "BLITTATA") è molto semplice: il blitter, attraverso i canali A, B e C, legge dati dalla memoria, effettua delle operazioni su di essi e scrive i risultati in memoria attraverso il canale D. Per eseguire una blittata è dunque necessario specificare le seguenti informazioni:

1. quali canali usare per questa operazione
2. che operazione effettuare sui dati letti
3. per ogni canale usato, l'indirizzo da dove iniziare a leggere e scrivere
4. quanti dati leggere o scrivere

Notate che la quantità di dati letta (o scritta) durante un'operazione è la stessa per tutti e quattro i canali: se in un'operazione uso i canali A, B e D, il numero di words che vengono lette attraverso il canale A è uguale al numero di words che vengono lette attraverso il canale B e al numero di words che vengono scritte attraverso il canale D.

Queste informazioni vengono specificate attraverso alcuni registri hardware. I registri che controllano il blitter sono, come tutti i registri hardware, a 16 bit. Vi sono però molti registri che hanno indirizzi consecutivi. Questo fatto rende possibile accedervi a coppie utilizzando delle `move.l` invece che delle `move.w`, analogamente a quanto abbiamo visto per le coppie di registri `BPLxPT ($dff0e0...)` e `COPxLC ($dff080...)`.

Prima di iniziare a scrivere nei registri, è necessario però essere certi che il blitter sia fermo, cioè che non stia già compiendo un'altra operazione. È indispensabile aspettare che l'ultima "blittata" sia finita prima di farne un'altra, altrimenti si potrebbero causare esplosioni e crolli nel raggio di 100 metri, un vero cataclisma, paragonabile ad un bombardamento aereo.

Per sapere se il blitter è fermo o sta "blittando", basta controllare lo stato di un bit (il bit 6) del registro `DMACONR ($dff002)`. Se tale bit vale 1 allora il blitter sta lavorando, mentre se vale 0 vuol dire che il blitter ha finito. In pratica, quindi basta una semplice istruzione assembler:

```

1 AspettaBlit:
2 btst    #6,$dff002      ; dmaconr - il blitter ha finito?
3 bne.s   AspettaBlit    ; Non andare avanti fino a che non ha finito

```

Purtroppo, a complicare le cose c'è un fastidiosissimo BUG hardware nelle prime versioni del chip Agnus (il chip che contiene il blitter) a causa del quale la prima volta che viene effettuata

una lettura del bit in questione si ha un risultato sbagliato: bisogna effettuare una lettura a vuoto prima di poter conoscere con esattezza lo stato del bit.

Dopo esserci assicurati che il blitter sia fermo, possiamo scrivere nei registri le informazioni che gli sono necessarie per la blittata e che abbiamo elencato sopra.

Vediamo ora in dettaglio come si procede.

1. Per ogni blittata possiamo abilitare o disabilitare indipendentemente i canali DMA, in modo da usare solo quelli che ci interessano, mediante dei bit di abilitazione che, se vengono settati a 1, abilitano il canale; se invece vengono azzerati lo disabilitano. I bit di abilitazione si trovano nel registro di controllo BLTCON0 (\$dff040):

canale	nome bit di abilitazione	posizione del bit in BLTCON0
A	SRCA	8
B	SRCB	9
C	SRCC	10
D	DEST	11

2. Per specificare quale operazione effettuare si usano i bit da 0 a 7 del registro di controllo BLTCON0, detti *MINTERMS*. Il valore che assumono tali bit determina l'operazione effettuata dal blitter. Il funzionamento dei MINTERMS è abbastanza complicato, e lo spiegheremo in dettaglio in seguito.
3. Vediamo ora come indicare gli indirizzi di partenza dei canali. Ad ogni canale è connesso un puntatore alla chip RAM che serve appunto per memorizzare l'indirizzo di partenza di un'operazione. Durante l'operazione il valore contenuto nel puntatore varierà automaticamente, indicando di volta in volta l'indirizzo della word che il blitter legge o scrive. Un puntatore è costituito (come per i canali DMA degli sprites e dei planes) da una coppia di registri a 16 bit, uno che contiene i 16 bit meno significativi (cioè più bassi) e uno che contiene i rimanenti (alti). In questa tabella sono riassunti i nomi e gli indirizzi dei puntatori:

canale		registro alto		registro basso	
nome	indirizzo	nome	indirizzo		
A	BLTAPTH	\$DFF050	BLTAPTL	\$DFF052	
B	BLTBPTH	\$DFF04C	BLTBPTL	\$DFF04E	
C	BLTCPTH	\$DFF048	BLTCPTL	\$DFF04A	
D	BLTDPTH	\$DFF054	BLTDPTL	\$DFF056	

Chiaramente, queste coppie di registri possono essere trattate come singoli registri a 32 bit – come per i puntatori alle CopperList ed ai Plane –, e, quindi, essere scritti con una singola istruzione `move.l` all'indirizzo di BLTxPTH. Pertanto d'ora in avanti li considereremo come singoli registri a 32 bit, usando i nomi di BLTxPT e riferendoci agli indirizzi \$dff050, \$dff04c, \$dff048 e \$dff054 (salvo eventuali eccezioni che verranno opportunamente segnalate).

I registri di puntatore dovrebbero essere scritti con un indirizzo in byte, ma poiché il blitter lavora solo su WORDS, il bit meno significativo del nostro indirizzo viene ignorato, dunque occorre ricordare che gli indirizzi devono essere PARI, ossia allineati a WORDS. Quindi occorre ricordarsi che si possono scrivere solo indirizzi PARI della memoria CHIP, sia per le sorgenti che per la destinazione.

NOTA: Assegnate i bit non usati a zero, specialmente quelli che non hanno nessuna funzione nemmeno in ECS, dato che in versioni future potrebbero essere usati per chissà quali scopi e i risultati sarebbero imprevedibili.

4. L'ultima operazione da effettuare è indicare la quantità di dati che devono essere letti o scritti. Ciò viene fatto tramite il registro BLTSIZE (\$dff058). Questo registro permette al blitter di considerare i dati che legge e scrive non come una semplice sequenza di word, ma come una sorta di rettangolo bidimensionale composto da words. Per esempio il blitter considera una sequenza di 8 words, come un rettangolo largo 8 words e alto 1 linea:

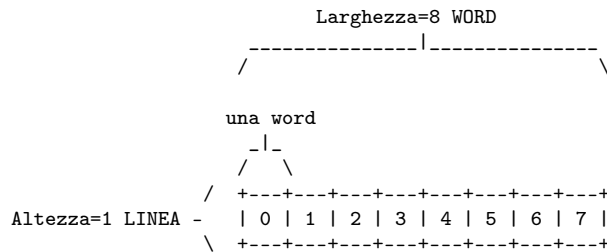


fig. 1 rettangolo di words 8*1

Facciamo un altro esempio: una sequenza di 50 words può essere considerata come un rettangolo di 10 word X 5 linee:

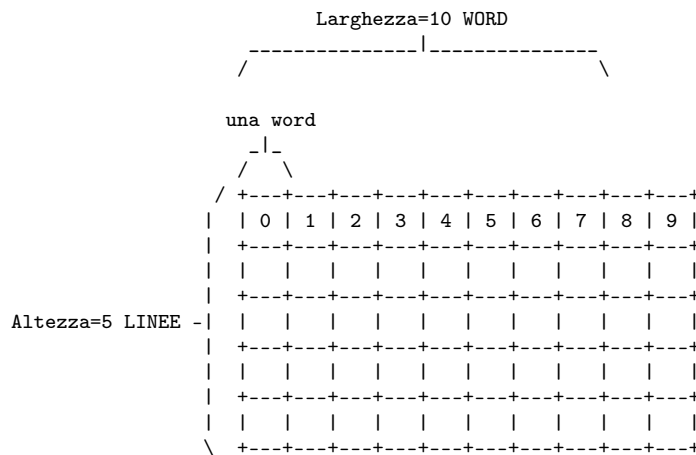


fig. 2 rettangolo di words 10*5

Questo fatto che a prima vista può sembrare un inutile complicazione è in realtà una delle caratteristiche che rendono il blitter tanto potente. Tra un attimo vedremo bene per quale motivo. Prima però vediamo come funziona BLTSIZE. Per specificare la quantità di dati coinvolta nella blittata, si scrivono in BLTSIZE le dimensioni del rettangolo di words che i dati formano. Nei 6 bit bassi va espressa la dimensione orizzontale, ovvero il *numero di word* che costituisce ogni linea orizzontale; nei 10 bit alti va espresso *numero di linee* orizzontali che costituiscono il rettangolo: in sostanza, nei 6 bit bassi va la larghezza in X del rettangolo, nei 10 bit alti va l'altezza in Y del suddetto rettangolo. È da notare che se il valore dei 10 bit alti (altezza) è 0, il blitter blitterà 1024 linee, e se il valore dei 6 bit bassi (larghezza in word) è 0, il blitter blitterà 64 word: la più grande blittata, dunque, si ottiene scrivendo `move.w #$0000,$dff058`. Essa sarà di 64 word X 1024 linee (=64*2*1024=128 Kb). Il registro BLTSIZE ha inoltre un'altra importantissima funzione: **scrivendoci si attiva il blitter**, dando inizio all'operazione specificata. Per questo motivo, **si deve scrivere nel registro BLTSIZE sempre dopo aver scritto in tutti gli altri registri del Blitter**, altrimenti

Abbiamo visto che con il blitter possiamo copiare dati da un punto all'altro della memoria. Se copiamo dati all'interno di un bit-plane, i valori da noi copiati verranno usati per formare l'immagine sullo schermo. Il blitter, poiché come abbiamo detto lavora su dati di dimensione WORD (16 bit), ci permette di modificare l'immagine a gruppi di WORD, cioè a gruppi di 16 pixel. Per esempio, se con il blitter scriviamo sopra la 21-esima word del bitplane mostrato in figura, modificheremo i 16 pixel più a sinistra della seconda riga dell'immagine. Supponiamo ora di avere un'immagine alta una sola riga e larga un certo numero L di pixel. Proprio per il fatto che il bit-plane è diviso in word, che contengono 16 pixel, è conveniente che la larghezza in pixel della nostra immagine, cioè L , sia un numero multiplo di 16, in modo che l'immagine sia contenuta esattamente in $L/16$ word. Ciò può essere ottenuto aggiungendo dei pixel di valore 0 alla fine della nostra immagine, come illustrato dal seguente esempio:

Questa è un'immagine larga 20 pixel e alta una sola riga.

```
11001101010100011001
 \-----/
  |
 20 pixel
```

Non è comoda da gestire perché 20 non è un multiplo di 16. Allora aggiungiamo dei pixel di valore 0 alla fine in modo da rendere la larghezza pari a 32 pixel, cioè pari ad un multiplo di 16.

```
11001101010100011001000000000000
 \-----/
  |
 32 pixel
```

La nostra immagine è memorizzata tra i dati nel nostro programma. Per farla apparire sullo schermo, dobbiamo copiarla nella zona di memoria dedicata al bit-plane. L'immagine assumerà sullo schermo una posizione corrispondente alle word del bit-plane nelle quali la copieremo. Supponiamo di voler disegnare l'immagine sullo schermo in maniera che il primo pixel di essa, cioè il pixel più a sinistra, assuma le coordinate X e Y (vi ricordo che il sistema di coordinate dello schermo ha l'origine, cioè il punto di coordinate $X=0$ e $Y=0$, nell'angolo superiore sinistro, le coordinate X crescono andando verso destra mentre le Y crescono andando verso il basso). Tale pixel sarà contenuto in una word del bit-plane.

Per il momento limitiamoci a considerare il caso in cui X sia anch'esso multiplo di 16. Ciò ci assicura che il nostro pixel sia il primo (cioè il pixel più a sinistra) della word a cui appartiene. In questo modo, una volta calcolato l'indirizzo di tale word, potremo copiarci (con il blitter) la prima word dell'immagine. Le altre word che formano la nostra immagine verranno copiate naturalmente nelle successive word del bit-plane.

Tutto questo, visto che il blitter è in grado di copiare sequenze di word, si può fare con una singola blittata che abbia come indirizzo sorgente la prima word dell'immagine, e come indirizzo destinazione, l'indirizzo della word del bit-plane a cui appartiene il pixel di coordinate X e Y . Vediamo come si fa per calcolare questo indirizzo.

Numeriamo le word del bit-plane a partire da 0, come mostrato in figura, e calcoliamo il numero della word che ci interessa: da tale numero risaliremo poi all'indirizzo vero e proprio.

Cominciamo col calcolare il numero della prima word della riga Y , ricordando ancora una volta che ogni riga è formata da 20 word e che le righe sono numerate a partire da 0. Potete notare dalla figura che la prima word della riga 0 (la prima riga) ha numero 0, la prima word della riga 1 (la seconda riga) ha numero 20, la prima word della riga 2 ha numero 40, la prima word della riga 3 ha numero 60 e così via.

In generale quindi, la prima word della riga Y ha numero $Y*20$. I numeri delle altre word della riga sono consecutivi a quello della prima: la seconda word della riga ha numero $Y*20+1$, la terza word della riga ha numero $Y*20+2$ e così via.

Possiamo chiamare “distanza” di una certa word R dalla prima word della riga cui R appartiene, la quantità che bisogna aggiungere al numero della prima word della riga per ottenere il numero della word R: in pratica, poiché la seconda word della riga ha numero $Y*20+1$, diciamo che essa ha “distanza” 1 dalla prima word della riga; allo stesso modo la terza word della riga, che ha numero $Y*20+2$ ha distanza 2 dalla prima word della riga, e così via. Possiamo dire inoltre che la prima word della riga ha distanza 0 da se stessa. È molto semplice calcolare la distanza tra la word che contiene il pixel di coordinata X e la prima word della riga, come vedremo aiutandoci con la seguente figura:

riga Y		Y*20+0		Y*20+1		Y*20+2		Y*20+3		-	-
Distanza dalla prima word		0		1		2		3		-	-
Pixel contenuti:		0-15		16-31		32-47		48-63		-	-

fig. 4 riga di words

La coordinata X del nostro pixel rappresenta la distanza (in pixel) tra esso e il primo pixel della riga. Poiché ogni word contiene 16 pixel, la prima word di una riga contiene i primi 16 pixel della riga, cioè quelli che hanno una coordinata X (=una distanza dal bordo) da 0 a 15. La seconda word invece contiene i pixel la cui coordinata X varia da 16 a 31, la terza word i pixel con X che varia da 32 a 47 e così via: ogni 16 pixel abbiamo una word.

Quindi per calcolare la distanza tra le word, basta dividere la distanza in pixel (cioè il valore di X) per 16. Poiché abbiamo scelto X multiplo di 16, il risultato sarà un intero. Per esempio, se $X=32$, la distanza in word vale $32/16=2$. Infatti, come vedete nella figura il pixel 32 della riga Y è il primo pixel della seconda word della riga, il cui numero è proprio $Y*20+1$. Con lo stesso calcolo vediamo che il pixel che ha $X=64$ è contenuto nella word che dista $64/16=4$, word il cui numero è $Y*20+3$. Questo calcolo funziona anche se $X=0$: infatti abbiamo distanza $0/16=0$ cioè la word di numero $Y*20+0$ che è appunto la prima word della riga.

In totale, quindi la word che contiene il pixel X,Y è la word con numero N dato dalla seguente formula:

$$N=(Y*20)+(X/16)$$

Questa formula è valida per bit-plane nei quali una riga è formata da 20 word. In generale la formula è:

$$N=(Y*NUMERO_WORD_CHE_FORMANO_UNA_RIGA)+(X/16)$$

Dal numero della word possiamo risalire all'indirizzo corrispondente: basta conoscere l'indirizzo della prima word del bit-plane e aggiungerci il numero della word moltiplicato per 2 (la moltiplicazione è necessaria perché l'indirizzo è espresso in byte e 1 word = 2 byte):

$$\text{Indirizzo word}=(\text{Indirizzo bitplane})+N*2 .$$

Nell'esempio Lezione9b1.s troverete l'applicazione di tutto quanto abbiamo detto. Nell'esempio Lezione9b2.s vedrete una serie di blittate in diverse posizioni dello schermo.

Iniziamo ora ad occuparci di immagini che abbiano altezza maggiore di una riga. Abbiamo visto quando abbiamo parlato del registro BLTSIZE, come il blitter consideri i dati su cui deve operare come dei "rettangoli" di words. Questa caratteristica è molto utile, perché gli consente di lavorare agevolmente con immagini rettangolari. Supponiamo ad esempio di voler copiare all'interno di un bitplane un'immagine larga 32 pixel e alta 2 linee. Questa piccola immagine andrà ad occupare una piccola porzione del bitplane, evidenziata nella figura dalle linee oblique.

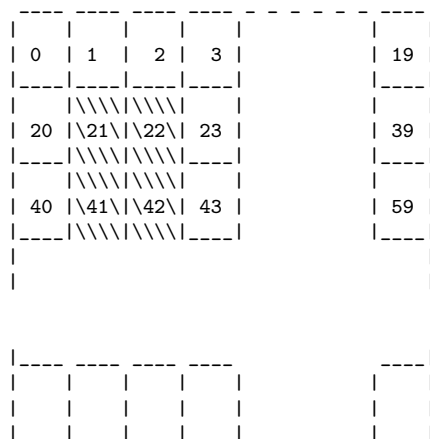


Fig. 5 Un bit-plane con evidenziata la porzione sulla quale blitteremo

Si tratta di un piccolo rettangolo largo 2 words (cioè 32 pixel) e alto 2 linee. Capirete immediatamente che per effettuare la copia è necessario specificare in BLTSIZE le dimensioni del rettangolo. Ma ciò non è sufficiente. Per rendercene conto, mettiamoci per un momento nei panni del blitter e proviamo a eseguire noi la copia, puntando la nostra attenzione per il momento solo sulla fase di scrittura.

Sappiamo (perché è scritto in BLTDPT) l'indirizzo della word in alto a sinistra del rettangolo (la word 21 nella figura). Inoltre sappiamo (è scritto in BLTSIZE) le dimensioni del rettangolo da copiare. Molto bene. Leggiamo la prima word e la copiamo all'indirizzo della word 21. Ora dobbiamo copiare la seconda word della prima riga. Sappiamo che questa word è consecutiva alla prima word, quindi aggiungiamo 2 all'indirizzo della prima word (che è scritto in BLTDPT) e sappiamo l'indirizzo della seconda word da scrivere. La scriviamo e abbiamo finito la prima riga. Molto soddisfatti ci prepariamo a scrivere la seconda riga. E qui ci accorgiamo che c'è un piccolo problema: la prima word della seconda riga **non è consecutiva** all'ultima word della prima riga! Infatti come potete vedere dalla figura l'ultima word della prima riga è la word 22 mentre la prima della seconda riga è la word 41.

Come facciamo a calcolare l'indirizzo della prima word della seconda riga? Nella figura è rappresentato un bitplane largo 20 word, ma è solo un esempio. Come fa il povero blitter a sapere quante word è largo il bitplane? Infatti potremmo essere in presenza di un bitplane più largo dello schermo visibile! Anzi a pensarci bene chi l'ha detto al blitter che lo stiamo usando per copiare un rettangolo sullo schermo? E se invece stessimo semplicemente copiando dei dati in una copperlist? È evidente che da solo il blitter non sa trarsi d'impaccio. Ma non c'è problema, lo aiutiamo noi.

Quello che al blitter serve di sapere è semplicemente come fare per calcolare l'indirizzo della prima word di una riga sapendo l'indirizzo dell'ultima word della riga precedente. Se guardate un attimo la figura vi convincerete il blitter deve semplicemente "saltare" le word da 23 a 40 comprese. Ciò può essere fatto aggiungendo all'indirizzo della word 22 (cioè l'indirizzo dell'ultima word della prima riga, che il blitter già conosce) il numero di bytes di differenza rispetto alla word 42 (che è appunto la prima word della nuova riga). Tale numero di bytes, che si chiama MODULO, è ovviamente uguale al numero di words da "saltare" moltiplicato per 2 (poiché come ben sapete una word occupa 2 bytes).

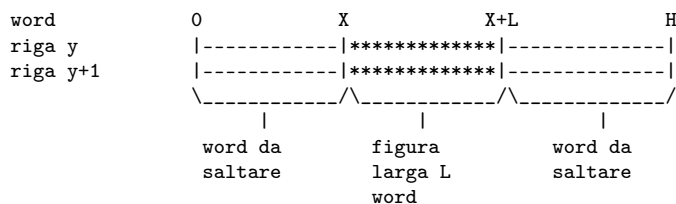


Fig. 6 Modulo

In generale, se dobbiamo copiare un rettangolo largo L words all'interno di una bitmap larga H words, il MODULO espresso in bytes si ottiene con la seguente formula:

$$\text{MODULO} = (H-L)*2$$

Il calcolo $H-L$ ci darebbe il modulo espresso in words, la moltiplicazione per 2 serve per esprimerlo in bytes. Nel nostro esempio il MODULO vale $(20-2)*2$. Se vi ricordate avevamo già incontrato il concetto di modulo relativamente ai bit-planes. Il modulo del blitter funziona esattamente allo stesso modo.

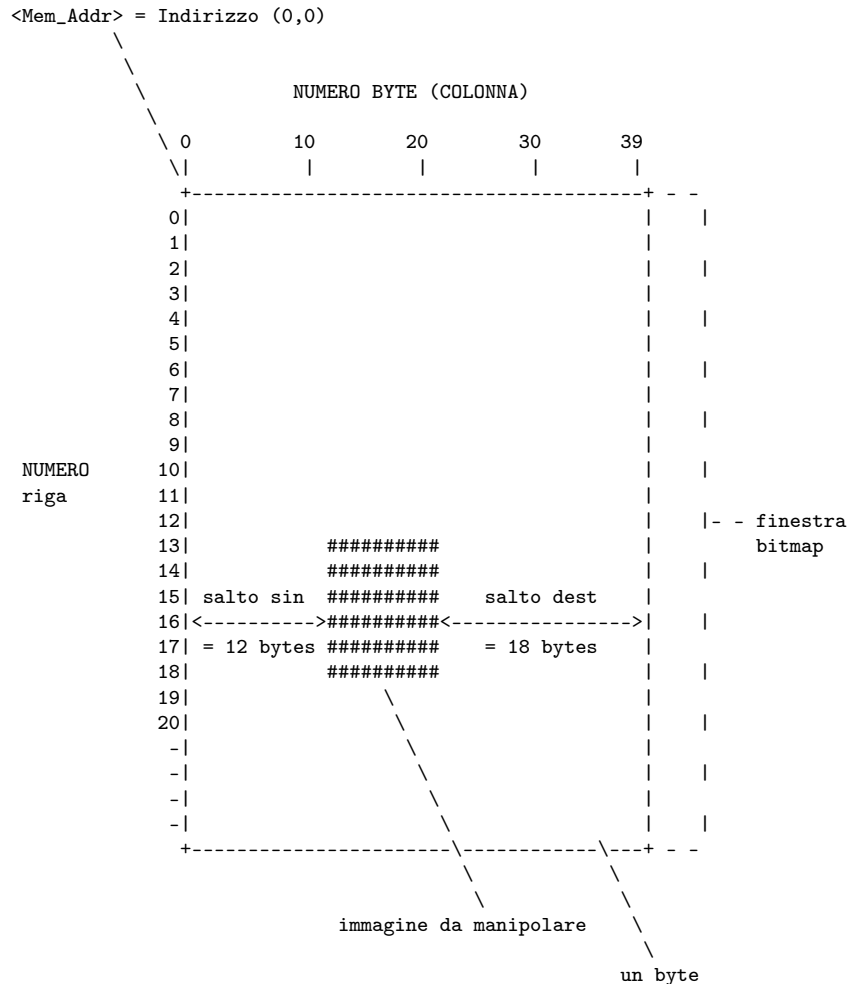
È possibile assegnare un modulo diverso per ogni canale DMA. In questo modo i dati possono essere copiati e spostati fra bitplanes di differenti larghezze. Il valore del modulo viene scritto in 4 appositi registri, uno per ogni canale DMA: BLTAMOD per il canale A (\$dff064), BLTBMOD per il B (\$dff062), BLTCMOD per il C (\$dff060), BLTDMOD per il D (\$dff066). I valori del modulo sono in byte, non words. Siccome il blitter può operare solo su words, il bit meno significativo è ignorato, questo significa che il valore del modulo deve essere pari. Il valore, positivo o negativo, viene aggiunto automaticamente ai registri che puntano agli indirizzi (BLTxPT) ogni volta che il blitter ha finito di copiare una riga, in modo da calcolare l'indirizzo della prima word della riga successiva. Valori negativi del modulo possono essere utili in molti casi, ad esempio per ripetere una linea settando il modulo come la larghezza del bitplane al negativo. Abbiamo già visto nella lezione 5 come replicare una linea mettendo il modulo copper BPL1MOD/BPL2MOD a -40, o comunque $-LunghezzaLinea$.

A questo punto sappiamo come copiare un rettangolo all'interno di una bitmap. Riassumiamo con un esempio tutti i calcoli necessari.

Supponiamo di voler operare su una sezione di un bitmap 320x200, che inizia alla riga 13, word 6 (dove entrambi sono numerati da zero) larga 5 word. Per prima cosa dobbiamo calcolare l'indirizzo della prima word del rettangolo, e poi scriverlo nel registro BLTxPT del canale che ci interessa. Il calcolo viene fatto nel modo seguente: prendiamo l'indirizzo del primo word del bitplane, aggiungiamo $13*20*2$ bytes per calcolare l'indirizzo del primo byte della riga 13 (infatti ogni riga occupa 20 words=40 bytes) e infine aggiungiamo 12 byte (=6 words) per arrivare alla giusta posizione orizzontale. La larghezza è di 5 words (10 byte). Alla fine di ogni riga, dobbiamo saltare 30 byte per arrivare all'inizio della riga seguente, quindi usiamo un

modulo di 30. In generale, la larghezza (in words) raddoppiata più il valore modulo (in byte) dovrebbe equivalere alla piena larghezza, in byte, del bitplane che contiene l'immagine.

Questi calcoli sono illustrati nella figura che mostra i valori richiesti usati nei registri blitter BLTxMOD e BLTxPTR (BLTxPTH e BLTxPTL).



$$\begin{aligned} \text{BLTxPTR} &= \text{<Mem_Addr>} + (40 \times 13) + 12 \\ &= \text{<Mem_Addr>} + 532 \end{aligned}$$

$$\begin{aligned} \text{BLTxMOD} &= 12 + 18 \\ &= 30 \text{ bytes} \end{aligned}$$

Fig. 7 calcoli per BLTxPTR e BLTxMOD

A questo punto è bene fare una sosta e guardarsi un pò di esempi.

In lezione9c1.s e lezione9c2.s trovate semplici esempi di copia di aree rettangolari. Studiateli attentamente, concentrando sul calcolo degli indirizzi e dei moduli usati nelle blittate.

In lezione9c3.s c'è un esempio nel quale viene effettuata una blittata con modulo negativo.

In lezione9d1.s e lezione9d2.s vedrete i primi esempi di animazione con il blitter.

L'idea è molto semplice, per dare l'idea del movimento basta disegnare la nostra figura ogni volta in una posizione diversa, un pò come si faceva con gli sprites. Diversamente da allora, però,

prima di disegnare la figura nella nuova posizione, dovremo cancellarla dalla vecchia, altrimenti otterremmo un effetto "scia". In questi 2 esempi spostiamo di volta in volta la figura in basso di una riga, aggiungendo ogni volta 40 bytes all'indirizzo BLTxPT.

In lezione9d3.s applichiamo la stessa tecnica per spostare la figura in orizzontale. Notate però che modificare l'indirizzo equivale a spostare il rettangolo a destra (o a sinistra) di una o più word. Poiché una word corrisponde a 16 pixel, in questo modo possiamo spostare orizzontalmente la figura solo a scatti di 16 pixel, il che rende, come potete vedere nell'esempio il movimento poco fluido, e troppo veloce.

Finora ci siamo limitati a disegnare figure con il pixel più a sinistra in una posizione multipla di 16. Per avere un movimento fluido, invece, è necessario poter disegnare la figura in una posizione arbitraria dello schermo. Facciamo un esempio, immaginando di avere l'immagine di un'auto che vogliamo spostare sullo schermo. Calcolando opportunamente l'indirizzo del rettangolo che la contiene, possiamo "blittare" la nostra auto a partire da una qualunque delle word che formano lo schermo. Se la nostra immagine di auto, per esempio, ha lo sportello a 5 pixel dalla sua estrema sinistra, potremo spostarlo, assieme con la macchina, a 5 pixel dall'inizio di una qualche word dello schermo. Se vogliamo spostarla verso destra, possiamo "blittarla" a partire dalla word successiva. Il risultato sarebbe uno "scatto" di 16 pixel ogni volta. Ma se vogliamo far scorrere quell'auto a destra o a sinistra di un pixel alla volta, o comunque vogliamo blittarla in una posizione orizzontale che non sia multipla di 16, come si fa?

Dobbiamo fare in modo che i pixel che formano l'immagine vengano copiati NON a partire dal primo bit della prima word, a partire da un bit arbitrario all'interno di tale word, come illustrato dalla seguente figura.

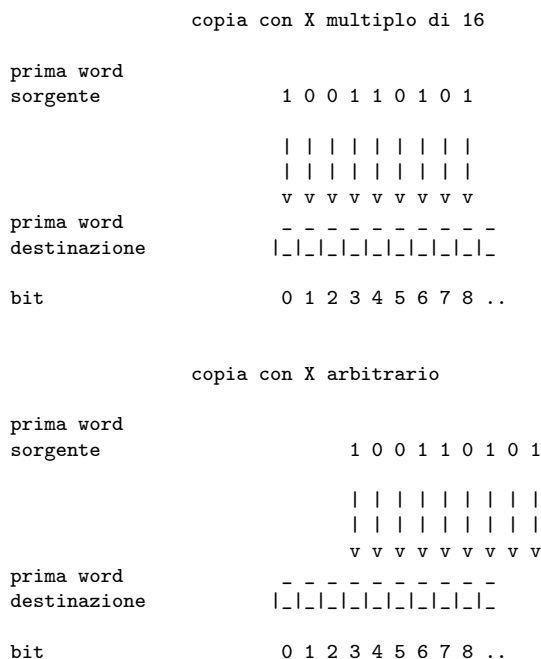


Fig. 8 Shift

In pratica dobbiamo shiftare i bit che compongono la figura da destra verso sinistra. Il blitter possiede degli shifter hardware per i canali A e B, che shiftano a destra tutti i bit delle word che vengono lette dai canali A e B. I bit vengono shiftati di numero di posizioni che può variare da 0

a 15. Shiftare di 0 posizioni equivale a non shiftare affatto: tutte le blittate che abbiamo visto (e fatto) finora erano blittate con shift di 0 posizioni. Il valore di shift per il canale A è assegnato con i bit dal 15 al 12 del registro BLTCON0 (\$dff040); il valore di shift del canale B è assegnato con i bit dal 15 al 12 di BLTCON1 (\$dff042). Se vi ricordate finora avevamo sempre lasciato questi bit al valore 0, che indica appunto uno shift di 0 posizioni. Il canale C invece è un proletario, non ha shifter. (Per chi se lo è dimenticato, shiftare dei bit significa “scorrere” dei bit verso destra o verso sinistra...) L'operazione di shifting viene eseguita contemporaneamente alla normale copia e non influenza la velocità del blitter: qualunque sia il valore di shift, il tempo impiegato per la blittata è sempre lo stesso.

Grazie allo shift, possiamo disegnare una figura avente il pixel più a sinistra in una posizione X arbitraria. Infatti, calcolando come di consueto l'indirizzo della destinazione, possiamo disegnare la figura ad una posizione X multipla di 16. Attivando contemporaneamente lo shifter, possiamo spostarla ulteriormente verso destra per farla raggiungere la posizione desiderata. Per esempio, supponiamo che si voglia una posizione X di 38 pixel. Mediante il calcolo dell'indirizzo possiamo spostare la figura 32 pixel (32 è multiplo di 16) a destra del bordo, 0 e possiamo spostarci a destra di altri 6 bits ($38-32=6$) impostando uno shift di 6.

In generale se X non è multiplo di 16, facendo la divisione intera $X/16$ otteniamo un risultato intero (che usiamo per calcolare l'indirizzo destinazione) e un resto che ci dice di quanto deve essere lo shift. (Ricordo che la divisione intera è una divisione nella quale non vengono calcolate le cifre decimali del risultato e viene ottenuto un resto, come si fa in prima elementare; per esempio $7/3=2$ col resto di 1). Nel caso di una posizione orizzontale $X=100$, abbiamo $100/16=6$ con il resto di 4 (infatti $16*6=96$ e $100-96=4$); dunque la distanza tra la prima word destinazione e la prima word della riga è pari a 6 words, ossia 12 bytes, e il valore di shift è di 4 bit.

Prima di iniziare ad usare lo shift dobbiamo però capire bene come funziona. Per cominciare, alcuni bit naturalmente sono shiftati a destra fuori delle word a cui appartenevano. Da sinistra deve essere shiftato dentro qualcosa per rimpiazzare i bit usciti. Cosa in particolare? Per la prima word della blittata, sono shiftati dentro degli zeri; per ogni successiva word della stessa blittata, i bit shiftati dentro una word sono quelli shiftati fuori dalla word precedente. Insomma, quello che esce da una parte (la destra) rientra dall'altra (sinistra!) nella word seguente. Facciamo un piccolo esempio, aiutandoci con una figura per capirlo meglio. Supponiamo di copiare 3 words (possono formare un rettangolo alto una riga e largo 3 word, oppure alto 3 righe larghe 1 word, non fa differenza dal punto di vista dello shift), applicando un valore di shift pari a 3. Guardiamo cosa succede:

SORGENTE		
word 1	word 2	word 3
1000110001010101	0001001001000110	1010101010101010
DESTINAZIONE		
word 1	word 2	word 3
0001000110001010	1010001001001000	1101010101010101
---	---	---
questi 3 bit sono gli zeri shiftati dentro la prima word	questi sono i 3 bit shiftati fuori dalla prima word e rientrati nella seconda word	questi sono i 3 bit shiftati fuori dalla word 2 e rientrati nella word 3

Fig. 9 shift

Notate che gli ultimi 3 bit della word 3 della sorgente **non** vengono copiati da **nessuna parte**! Ad esempio, consideriamo una blittata larga tre words e alta due words, con uno shift di 4 bit. Per semplicità, assumiamo che sia una copia normale da A a D. La prima word che sarà scritta

in D è la prima word presa da A, shiftata a destra di quattro bit con 4 bit azzerati shiftati dentro dalla sinistra. La seconda word sarà la seconda word presa da A, shiftata a destra, con i quattro bit meno significanti (a destra) della prima word shiftati dentro. Dopo, scriverà la prima word della seconda riga presa da A, shiftata quattro bit, con i quattro bit meno significativi dell'ultima word dalla prima riga shiftati dentro. Questo continuerà finché la blittata non è terminata.

In lezione 9e1.s potete vedere un esempio di uso dello shift, che permette ad una figura di muoversi di un pixel alla volta verso destra. Il risultato però non è molto buono a causa del fatto che i bit che vengono shiftati fuori da una word, vengono shiftati dentro la word successiva, che si trova una riga più in basso. Quindi i bit che escono a destra rientrano da sinistra nella riga successiva! La situazione è illustrata dalla seguente figura, assumendo uno shift di 4 bit:

SORGENTE

```
word 1      1000001111100000
"  2        110011111111000
"  3        111111111101100
"  4        111111111111110
"  5        110011111111000
word 6      1000001111100000
```

DESTINAZIONE

```
word 1      0000100000111110
"  2        000011001111111
"  3        100011111111110
"  4        110011111111111
"  5        111011001111111
word 6      1000100000111110
```

~~~~~

queste 4 colonne di bit sono costituite dai bit entrati da sinistra: come vedete (tranne che per la prima riga) in ogni riga entrano i bit usciti dalla riga precedente.

Fig. 10 Shift di un rettangolo

Fortunatamente questo problema si risolve in modo molto semplice. Se ci pensate bene, quello che noi vorremmo, è che i bit che escono a destra da una word, rientrino da sinistra **non** nella riga successiva, ma piuttosto **nella word più a destra!** Dobbiamo quindi “coinvolgere” nella blittata anche le word più a destra. Ciò si può fare semplicemente aumentando la larghezza della figura aggiungendo a destra una “colonna” di word di *valore zero*. In questo modo, la colonna in più è invisibile, ed inoltre i bit shiftati fuori dalle word che la compongono saranno tutti zeri e perciò non daranno fastidio rientrando nelle word della riga seguente. Per chiarirvi le idee ecco cosa succede:

## SORGENTE

```
word 1      word 2
riga 1      10000011111000000000000000000000
"  2        11001111111100000000000000000000
"  3        11111111110110000000000000000000
"  4        11111111111110000000000000000000
"  5        11001111111100000000000000000000
"  6        10000011111000000000000000000000
```

~~~~~

Questa è la colonna di words aggiunta

DESTINAZIONE

```
word 1      word 2
riga 1      00001000001111100000000000000000
```

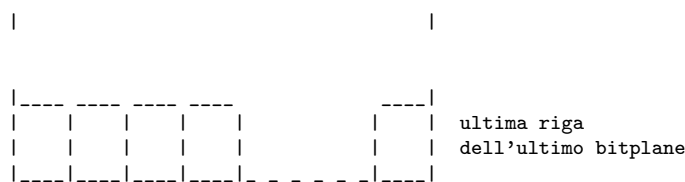
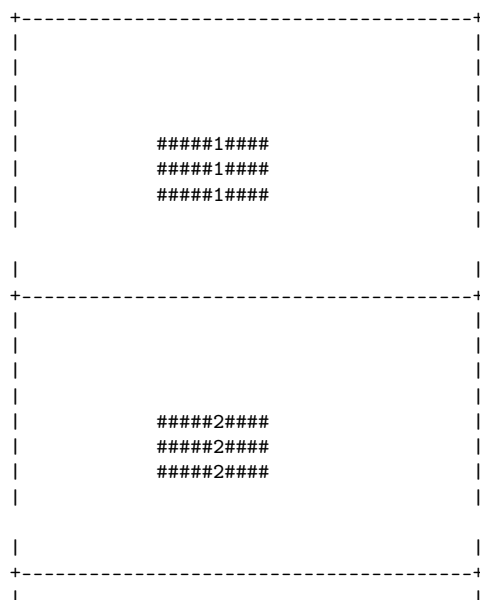



Fig. 12 Rappresentazione in memoria di un'immagine a più bitplane (ogni quadrato è una word)

Come già sapete, un bit-plane largo H words e alto V righe, occupa $H \cdot V$ words, ovvero $2 \cdot H \cdot V$ bytes (normalmente $H=20$ e $V=256$, quindi un bit-plane occupa $40 \cdot 256$ bytes). Questo significa che, poiché i bit-plane sono disposti in memoria uno di seguito all'altro, se il bit-plane 1 inizia all'indirizzo $PLANE1$, il bit-plane 2 inizierà all'indirizzo $PLANE2 = PLANE1 + 2 \cdot H \cdot V$. Analogamente il bit-plane 3 inizia all'indirizzo $PLANE3 = PLANE2 + 2 \cdot H \cdot V$ e così via. La stessa formula vale per determinare l'indirizzo di una word del secondo bit-plane conoscendo l'indirizzo della word corrispondente del primo bit-plane: per esempio la settima word del primo bit-plane ha indirizzo $INDIRIZZO1 = PLANE1 + 2 \cdot 7$, mentre la settima word del secondo bit-plane ha indirizzo $INDIRIZZO2 = PLANE2 + 2 \cdot 7 = PLANE1 + 2 \cdot H \cdot V + 2 \cdot 7$. Ma siccome $PLANE1 + 2 \cdot 7 = INDIRIZZO1$, abbiamo la seguente formula:

$$INDIRIZZO2 = INDIRIZZO1 + 2 \cdot H \cdot V.$$

Questa formula ci sarà molto utile tra poco. Un immagine rettangolare contenuta in uno schermo con N bitplane, sarà costituita da N rettangoli, uno per bitplane. Quindi, per manipolarla con il blitter, basta eseguire una blittata per ogni bit-plane. Nella figura sottostante potete vedere uno schermo con 3 bitplane, con evidenziata un'immagine alta 3 righe. In memoria, le righe di ogni bitplane, costituiscono un diverso rettangolo di words (abbiamo indicato in ogni riga dell'immagine il bitplane a cui essa appartiene). Come vedete le righe di ogni bitplane sono vicine tra loro e distanti dalle righe degli altri planes, pertanto devono essere manipolate con blittate diverse.



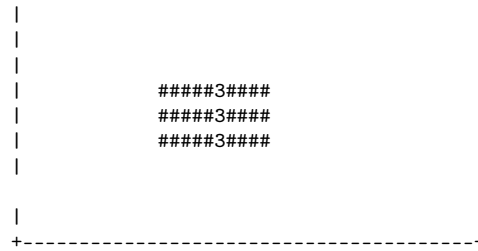


Fig. 13 Schermo con evidenziata un'immagine.

Per esempio, se abbiamo una figura da disegnare sullo schermo, prima blittiamo il primo plane della figura nel primo plane dello schermo, poi il secondo plane della figura nel secondo plane dello schermo, poi facciamo lo stesso con il terzo plane e via via con gli altri. Di solito quindi si fa un loop di blittate, tipo il seguente:

```

1  move.w  #NUMEROPLANES-1,d1      ; contatore del loop
2  LOOP:
3  waitblit:                        ; aspetta che il blitter abbia finito
4  btst    #6,2(a5)                 ; la blittata precedente
5  bne.s    waitblit
6
7  move.l   #$09f00000,$40(a5)      ; bltcon0 e BLTCON1 - copia da A a D
8
9  ;      carica gli altri registri del blitter
10
11 ;      avvia la blittata
12
13  dbra     d1,LOOP                ; fai il loop

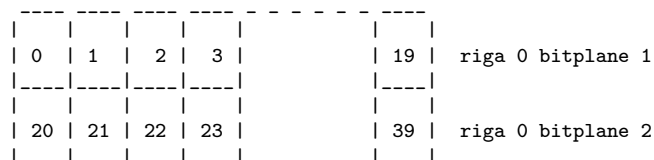
```

I valori da caricare nei registri del blitter sono sempre gli stessi ad ogni blittata, tranne ovviamente per quanto riguarda i registri BLT_xPT, perché gli indirizzi dei vari bit-plane sono diversi. A questo punto entra in gioco la formula che abbiamo visto. Mediante tale formula infatti, conoscendo gli indirizzi da scrivere nei registri BLT_xPT per la prima blittata (cioè per la blittata relativa al primo bit-plane), siamo in grado di calcolare gli indirizzi da scrivere nei registri BLT_xPT per le blittate successive (cioè relative ai bit-plane successivi). È sufficiente mettere in una variabile l'indirizzo relativo al primo bit-plane, e di aggiungere a tale indirizzo $2 \cdot H \cdot V$ ad ogni loop.

Nell'esempio lezione9f1.s potete vedere questa tecnica applicata. Non sempre comunque si usano loop di questo tipo.

Negli esempi lezione9f2.s e lezione9f3.s ci sono altri esempi di blittate "a colori".

Esiste però un'altro modo di disporre in memoria i bitplane, che ci consente di blittare in un colpo solo tutti i bitplanes di un'immagine, chiamato *Interleaved Bitmap* ovvero "bitmap" interlacciata. Come suggerito dal nome, questa tecnica consiste nel "mischiare" tra loro le righe dei vari planes. Invece di mettere prima tutte le righe del primo plane, poi quelle del secondo e così via, mettiamo prima la riga 0 (la prima) del primo bitplane, poi la riga 0 del secondo bitplane e poi in ordine le righe 0 degli altri planes; dopo le righe 0 di tutti i planes, mettiamo la riga 1 del primo plane, poi la riga 1 del secondo, e poi tutte le righe 1 degli altri planes; poi continuiamo così con le altre righe. Per capirlo bene guardate la figura seguente e confrontatela con la figura 12 dove è illustrata la disposizione normale dei bit-planes.



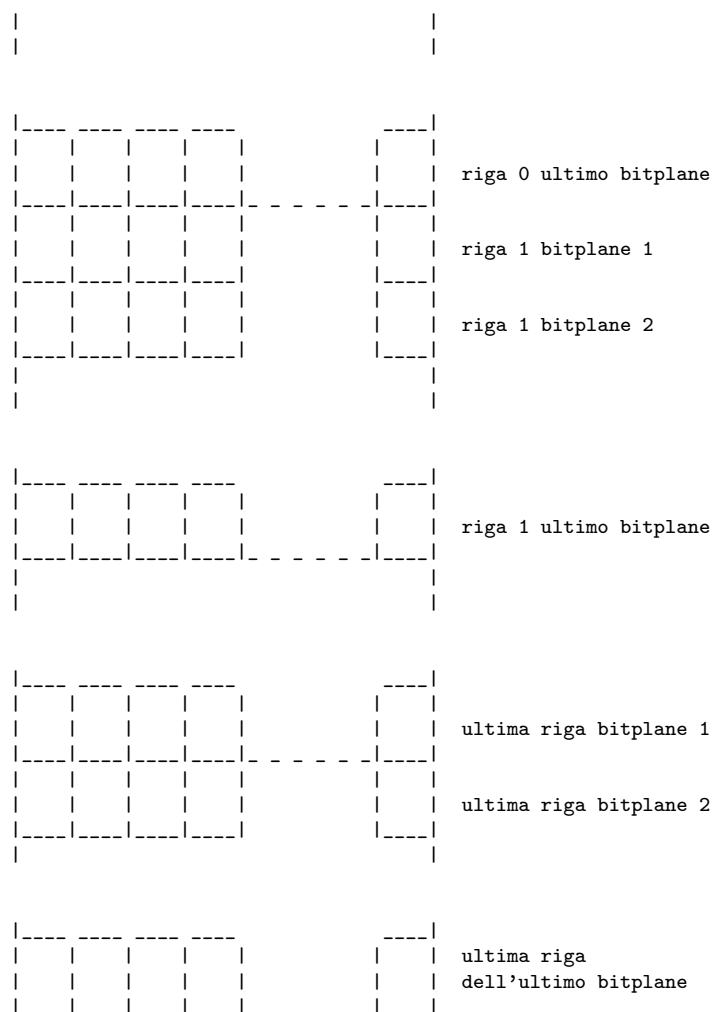


Fig. 14 Rappresentazione in memoria di un'immagine a più bitplane (ogni quadrato è una word) con la tecnica INTERLEAVED (o RAWBLIT).

Innanzitutto vediamo come si possono visualizzare immagini in questo formato, lasciando da parte un attimo il blitter. La quantità di words che compongono le righe è sempre la stessa. Quello che cambia è la disposizione relativa delle righe. Ciò per noi comporta 2 modifiche alla procedura che usiamo di solito per visualizzare i bitplanes. La prima riguarda il modo in cui calcoliamo gli indirizzi da mettere nei registri BPLxPT. Normalmente, per puntare i bitplane nella copper list, calcoliamo gli indirizzi dei bitplane successivi al primo, a partire dall'indirizzo del primo, aggiungendo ad esso ogni volta il numero di bytes occupati da una riga, moltiplicato per il numero di righe che formano il bitplane. Questo perché la prima riga di un bitplane è memorizzata dopo l'ultima del bit-plane precedente, e quindi "dista" dalla prima riga del bit-plane precedente un numero di righe pari all'altezza del bitplane stesso. Con la disposizione interleaved, invece, la riga 0 di un bitplane è memorizzata subito dopo la riga 0 del bitplane che lo precede.

Questo vuol dire che nel loop che calcola gli indirizzi dei bitplane dovremo aggiungere ogni volta all'indirizzo di un bitplane semplicemente il numero di bytes occupati da UNA riga, per ottenere l'indirizzo del bitplane seguente. Dobbiamo osservare inoltre che, a differenza del caso normale, le righe che formano un bitplane NON sono disposte consecutivamente in memoria. Infatti, tra la riga Y e la riga Y+1 ci sono le righe degli altri bitplane. Ciò significa che il puntatore al bitplane, ogni volta che arriva alla fine di una riga, deve "saltare" le righe degli altri bitplanes, per andare a puntare l'inizio della prossima riga.

Come avrete già intuito, per farlo saltare dobbiamo utilizzare il modulo. Vi ricordo infatti che anche i bitplanes hanno i loro moduli, contenuti nei registri BPLxMOD (dove x=1 per i bitplanes dispari e x=2 per i pari). Con la disposizione normale dei bitplanes, siccome subito dopo la fine di una riga inizia la riga successiva, mettiamo il modulo a 0 (a meno che non vogliamo fare l'effetto flood o abbiamo un'immagine più grande dello schermo). Vediamo invece il valore da mettere con la disposizione interleaved. Indichiamo con N il numero di bitplane che usiamo. Consideriamo il bitplane 1: all'inizio della riga Y il registro BPLPT1 punta alla prima word della riga Y del bitplane 1. Mentre la riga Y viene visualizzata sul monitor, il registro BPLPT1 si sposta puntando le words seguenti. Alla fine della riga Y, BPLPT1 punta alla prima word della riga Y del bitplane 2. A questo punto gli viene sommato il modulo. Noi vogliamo che BPLPT1 vada a puntare la prima word della riga Y+1 del bitplane 1. Dobbiamo quindi far saltare al puntatore le righe 2, 3, ecc... fino a N. In totale si tratta di N-1 righe (per esempio se abbiamo 4 bitplane, dobbiamo saltare la riga Y dei bitplanes 2, 3 e 4, cioè 3 righe). Quindi se una riga occupa L words, ovvero $2*L$ bytes, il valore corretto del modulo è $2*L*(N-1)$.

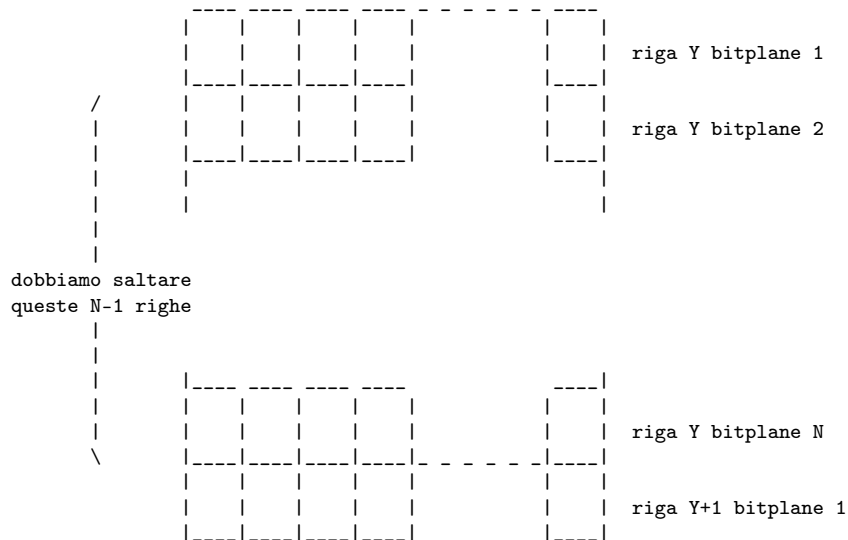


Fig. 15 Valore del modulo con la tecnica INTERLEAVED.

Naturalmente tutte le immagini che vogliamo visualizzare sullo schermo dovranno avere i bitplane disposti nel formato interleaved. Se un'immagine è definita direttamente nel nostro sorgente (tramite delle DC.w...), dobbiamo disporre le righe come previsto dal formato. Se invece vogliamo tenere l'immagine in un file esterno da includere con la direttiva INCBIN, dobbiamo convertirla **non** in formato RAW (che è il formato normale) ma in formato interleaved. Tutti i programmi di conversione supportano questo formato, anche se molti lo chiamano con altri nomi. In particolare il KEFRENS CONVERTER che abbiamo usato nel corso, chiama questo formato "RAW-BLIT". Altri converter lo chiamano "RASTER MODULO". Fate quindi attenzione a

convertire l'immagine nel giusto formato, altrimenti non vedrete nulla e passerete ore a cercare nel vostro programma un BUG inesistente !

In lezione9g1.s vedete un esempio di visualizzazione di una bitmap interleaved.

Vediamo ora perché questo formato è conveniente nell'uso del blitter. Nella figura seguente, viene mostrato uno schermo interleaved con evidenziata al suo interno un'area rettangolare. Come potete vedere, le righe che formano i vari bitplanes sono "mischiate" tra loro, e formano un unico rettangolo in memoria (abbiamo indicato in ogni riga dell'immagine il bitplane a cui essa appartiene). Confrontate questa figura con la figura 13 che mostrava una situazione analoga in uno schermo "normale". Nel caso normale, le righe degli N bitplanes dell'immagine, formano N distinti rettangoli di words, ognuno alto tante righe quante sono le righe dell'immagine. Nel caso interleaved invece, le righe degli N bitplanes, mischiandosi, formano un unico rettangolo di word. Notate che questo rettangolo ha altezza pari all'altezza dell'immagine moltiplicata per il numero di bitplanes che la formano. Nella figura abbiamo infatti un'immagine di 3 bitplanes alta 3 linee. Il rettangolo di words ha 9 righe.

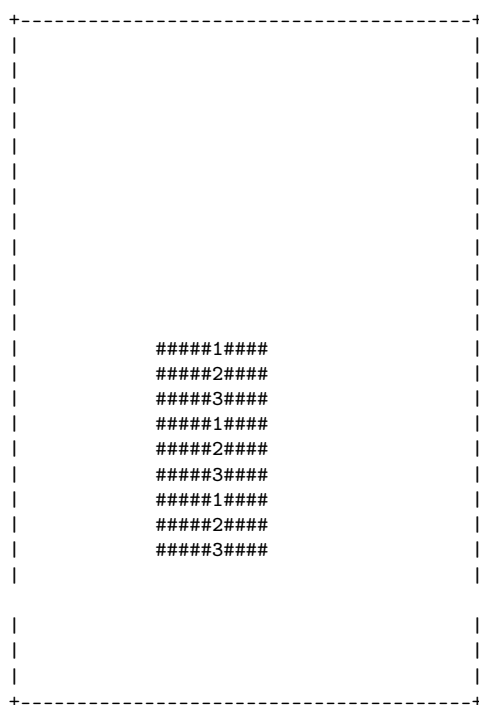


Fig. 16 Schermo INTERLEAVED con evidenziata un'immagine.

Il fatto che nel formato interleaved le righe dei bitplanes di un'immagine formino un unico rettangolo in memoria, è molto importante perché ci consente di operare sull'immagine mediante una sola blittata. Naturalmente, questa blittata è diversa dalle blittate che facciamo nel caso normale. Innanzitutto, è diversa la dimensione della blittata. Infatti nel caso normale ogni blittata ha altezza pari all'altezza dell'immagine, mentre nel caso interleaved il rettangolo di words ha un'altezza pari all'altezza dell'immagine moltiplicata per il numero di bitplanes che la formano, e quindi tale deve essere l'altezza della nostra blittata.

In secondo luogo, è diverso il modo in cui calcoliamo gli indirizzi delle blittate, in particolare dobbiamo cambiare il modo di calcolare l'indirizzo della prima word di una riga. Nel caso normale, abbiamo visto che se il rettangolo da blittare inizia alla riga Y , la “distanza” (offset)

della prima word della riga Y dall'inizio del bitplane è pari a $Y \cdot (\text{NUMERO DI BYTES OCCUPATI DA UNA RIGA})$. E questo è logico, perché in uno schermo normale le righe di un bitplane sono consecutive in memoria. In uno schermo INTERLEAVED, invece, le cose sono diverse perché le righe di un bitplane non sono consecutive. Infatti, come sapete, dopo la riga Y del primo bitplane, ci sono le righe Y degli altri bitplane, e dopo di esse la riga Y+1 del primo bitplane. Quindi, la distanza tra la prima word della riga Y del primo bitplane e la prima word della riga Y+1 del primo bitplane, è uguale al numero di bytes occupati dalle righe Y di tutti i bitplane della figura. Con lo stesso ragionamento capite facilmente che la distanza tra la prima word della riga Y del primo bitplane e l'inizio dello schermo è pari a:

$$Y \cdot (\text{NUMERO DI BYTES OCCUPATI DA UNA RIGA}) \cdot (\text{NUMERO DI PLANES})$$

In conclusione, quindi, il calcolo dell'indirizzo per blittare un rettangolo che inizia alle coordinate X e Y per uno schermo INTERLEAVED diventa:

$$\text{Indirizzo_word} = (\text{Indirizzo_bitplane}) + N \cdot 2$$

con:

$$N = (Y \cdot (\text{NUMERO_WORD_CHE_FORMANO_UNA_RIGA}) \cdot (\text{NUMERO_PLANES})) + (X/16) .$$

Fare una sola blittata invece che tante, oltre a rendere più semplice il programma, lo rende anche più veloce. Si badi bene che il tempo impiegato dal blitter è (più o meno) lo stesso, in quanto è vero che facciamo una sola blittata, ma essa ha altezza pari alla somma delle altezze delle blittate del caso normale, e quindi richiede lo stesso tempo, perché la velocità del blitter è determinata in sostanza dal numero di words che esso deve manipolare, e cioè dalla dimensione della blittata. Fare una sola blittata, però, avvantaggia notevolmente il processore, come potete capire dal seguente schema, che confronta le operazioni da effettuare nei 2 casi (schermo formato da 3 bitplanes):

	SCHERMO NORMALE	SCHERMO INTERLEAVED
1)	attendi la fine della (eventuale) blittata precedente	attendi la fine della (eventuale) blittata precedente
2)	carica i registri del blitter per la prima blittata	carica i registri del blitter per la prima e unica blittata
3)	attendi la fine della prima blittata	
4)	carica i registri del blitter per la seconda blittata	
5)	attendi la fine della seconda blittata	
6)	carica i registri del blitter per la terza blittata	

Come vedete, nel caso di schermo interleaved, il processore deve fare meno operazioni, e soprattutto deve attendere una sola volta che il blitter finisca, mentre nel caso di schermo normale deve attendere un numero di volte pari al numero dei bitplanes. Poiché durante un attesa il

processore non fa nulla di utile e non ha bisogno di riposo, è opportuno farlo lavorare il più possibile diminuendo il numero di attese.

L'esempio lezione9g2.s è la versione INTERLEAVED dell'esempio lezione9f1.s. Guardateli insieme, notando le differenze che essi presentano.

L'esempio lezione9g3.s, invece, è la versione INTERLEAVED dell'esempio lezione9f3.s. Confrontate anche questi.

9.5 Maschere

Il blitter ha la possibilità di mascherare la prima e l'ultima word di ogni riga che passa attraverso il canale A. Mascherare vuol dire leggere solo alcuni bit di tali word e ignorare gli altri. Questa operazione viene effettuata grazie a due registri, che finora avevamo usato senza spiegarne il significato. Questi due registri sono chiamati BLTAFWM (\$dff044) e BLTALWM (\$dff046), e servono rispettivamente per mascherare la prima e l'ultima word di ogni riga letta attraverso il canale A. Ognuno di essi contiene una word, detta maschera. Il blitter quando legge la prima o l'ultima word di una riga esegue un'operazione logica di AND tra la word letta e la maschera corrispondente. I bit della word letta dal canale A in corrispondenza dei quali c'è un bit settato a 0 nella maschera verranno cancellati. Vediamo qualche esempio:

```
word letta dal
canale A      %1001101100010111

maschera      %1111111100000000
-----
risultato     %1001101100000000
```

in questo modo abbiamo selezionato solo gli 8 bit più a destra della word.

```
word letta dal
canale A      %1001101100010111

maschera      %1111110000111111
-----
risultato     %1001100000010111
```

in questo modo abbiamo azzerato i 4 bit al centro della maschera. Se azzeriamo completamente la maschera, cancelliamo tutta la word:

```
word letta dal
canale A      %1001101100010111

maschera      %0000000000000000
-----
risultato     %0000000000000000
```

Se invece poniamo la maschera al valore \$ffff=%1111111111111111=-1 la maschera non cancella nulla, ovvero "fa passare" tutta la word:

```
word letta dal
canale A      %1001101100010111

maschera      %1111111111111111
```

```
-----
risultato      %1001101100010111
```

In tutti gli esempi che abbiamo visto finora non abbiamo avuto bisogno di mascherare nulla e infatti abbiamo inizializzato entrambe le maschere al valore \$ffff.

La prima word di ogni riga (cioè la word più a sinistra) è “ANDizzata” con BLTAFWM, e l’ultima word (la word più a destra) è “ANDizzata” con BLTALWM. Potete facilmente ricordarlo perché la F nel nome BLTAFWM indica “First” che come tutti sanno significa “prima” e la L in BLTALWM indica “Last”, cioè ultima. Naturalmente le 2 maschere possono essere diverse tra loro (senno a che ci servirebbero 2 registri?). Se la larghezza della riga è una singola word, entrambe le maschere vengono applicate simultaneamente alla stessa word. Poiché i 2 registri BLTAFWM e BLTALWM hanno indirizzi consecutivi è possibile inizializzarli con una sola istruzione `MOVE.L #maschera,$dff044`. È importante notare che le maschere vengono applicate ai dati **prima** di eseguire lo SHIFT. I canali B e C non hanno invece la possibilità di mascherare le words lette.

Nell’esempio `lezione9h1.s` mostriamo l’effetto delle maschere con semplici operazioni di copia.

In `lezione9h2.s` abbiamo una dimostrazione dell’utilità delle maschere nell’estrarre da un’immagine solo la parte che ci interessa.

In `lezione9h3.s` e `lezione9h4.s` presentiamo 2 nuovi effetti realizzati con l’ausilio delle maschere.

Gli esempi `lezione9h2r.s`, `lezione9h3r.s` e `lezione9h4r.s` sono le versioni in formato rawblit (interleaved) di `lezione9h1.s`, `lezione9h2.s` e `lezione9h3.s`. Fate un confronto incrociato, notando tutte le differenze che ci sono (in particolare notate che tutte le routines in versione interleaved non hanno la necessità di fare un loop per blittare su ogni plane, e pertanto hanno una struttura molto più semplice).

Dopo aver visto i nuovi effetti, torniamo ad occuparci di uno vecchio, cioè del pesce che nuota sullo schermo, per scoprire che, con le nostre nuove conoscenze sul blitter, possiamo realizzare un’importante miglioria. Abbiamo visto, infatti che per shiftare correttamente una figura, è necessario aggiungere a destra della figura una “colonna” di word azzerate. Questo fatto ci costringe a sprecare più memoria del necessario per memorizzare le figure. Ma ora, grazie alle maschere, possiamo evitare questo spreco. Per shiftare è necessario che l’ultima word di ogni riga della figura sia azzerata.

Invece di leggere direttamente dalla memoria una word azzerata, possiamo leggere una word di qualsiasi valore e azzerarla tramite la maschera. Siccome il mascheramento viene effettuato **prima** dello shift, al circuito di shift arriverà comunque l’ultima word di ogni riga azzerata, e tutto si svolgerà come se la word azzerata fosse stata letta dalla memoria. Visto che non ha importanza il valore dell’ultima word della riga, possiamo leggere una word di qualsiasi valore.

Proviamo allora a fare il seguente giochino: non aggiungiamo nessuna word a destra dell’immagine, ma senza dirlo al blitter, cioè settiamo la larghezza della blittata come se ci fosse una word in più a destra della figura. Il blitter, quindi, dopo aver letto l’ultima word di una riga, penserà di dover leggere ancora una word, e pertanto leggerà la word successiva a l’ultima della riga. Che cos’è questa word? Se usiamo un’immagine in formato normale, sarà la prima word della riga successiva dello stesso bitplane, mentre se l’immagine è in formato interleaved sarà la prima word di una riga di un altro bitplane. In ogni caso sarà comunque una word non nulla, ma per noi non c’è problema perché la possiamo azzerare con la maschera. A questo punto abbiamo solo un problemino: siccome abbiamo letto una word di troppo, il puntatore della sorgente si è spostato in avanti di una word, pertanto quando inizierà a leggere la prossima riga partirà dalla seconda word invece che dalla prima. Come si può far tornare indietro il puntatore? Naturalmente con il vecchio trucco del modulo negativo! Settando il modulo della sorgente a -2 (il

modulo si esprime in bytes) il blitter si riposiziona sulla prima word della riga seguente. Riassumiamo tutto tornando all'esempio del pesce che abbiamo usato per illustrare lo shift. Abbiamo dunque un'immagine di un solo bitplane, larga 1 word e alta 6 righe. Come abbiamo detto, NON aggiungiamo la colonna di word sulla destra.

```

SORGENTE
      word 1
riga 1    1000001111100000
"  2      1100111111111000
"  3      1111111111101100
"  4      1111111111111110
"  5      1100111111111000
"  6      1000001111100000

```

Fig. 17 NON aggiungiamo nessuna colonna di words

Tuttavia, facciamo finta che la colonna in più ci sia, e quindi blittiamo un rettangolo largo 2 words e alto 6 righe. Il blitter legge quindi 2 words per ogni riga, prendendo come seconda word la prima word della riga successiva. Vediamo in particolare, con l'aiuto della figura seguente, cosa accade durante la lettura della prima riga:

```

SORGENTE
      word 1
riga 1    1000001111100000-----
"  2      1100111111111000-----+-----
"  3      1111111111101100      |      |
"  4      1111111111111110      |      |
"  5      1100111111111000      |      |
"  6      1000001111100000      |      |
                                |      |
                                V      V

WORDS LETTE
DAL CANALE A
                                1000001111100000      1100111111111000
                                |      |
                                V      V

L'ULTIMA WORD DELLA
RIGA VIENE MASCHERATA
                                1000001111100000      0000000000000000
                                |      |
                                V      V

SHIFT (2 pixel)
                                0010000011111000      0000000000000000
                                |      |
                                V      V

                                Scritta al canale D      Scritta al canale D

```

Fig. 18 Shift con azzeramento dell'ultima word.

Come vedete la seconda word letta viene azzerata prima di essere shiftata. Dopo lo shift le 2 word vengono scritte attraverso il canale D. Nel frattempo il puntatore al canale A si è spostato in avanti di 2 words, e punta cioè alla prima word della terza riga. Noi invece dobbiamo farlo puntare alla prima word della seconda riga, cioè dobbiamo farlo tornare indietro di una word. Usiamo quindi un modulo pari a -2. Gli spostamenti del puntatore sono illustrati dalla figura seguente:

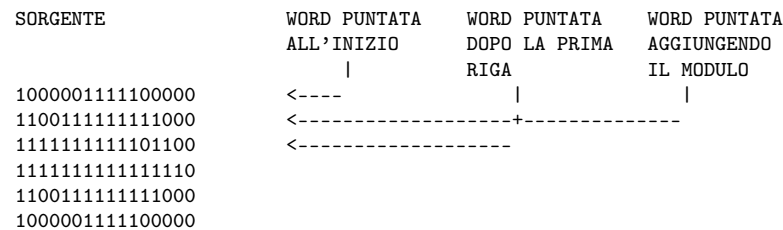


Fig. 19 Movimento del puntatore alla sorgente.

Per vedere il nostro pesce in azione consultate l'esempio `lezione9i1.s`.

Ormai sappiamo muovere molto bene delle figure sullo schermo usando il blitter. Queste figure vengono chiamate *BOB* che è un'abbreviazione del termine inglese *Blitter Object*, ovvero oggetti creati dal blitter. Con i BOB Possiamo fare le stesse cose che sappiamo fare con gli sprites hardware. I BOB sono più lenti degli sprites, perché il blitter impiega comunque un certo tempo per copiare dati. Per contro, però i BOB non soffrono delle limitazioni degli sprites riguardo a dimensione, colori e numero massimo. Infatti un BOB può essere grande quanto vogliamo noi (è ovvio però che al crescere delle dimensioni cresce la quantità di memoria occupata, e di conseguenza il tempo necessario al blitter per spostarlo), e può avere un numero di colori pari a quello dello schermo. Inoltre non c'è nessun limite per quanto riguarda il numero di bob contemporaneamente sullo schermo (ovviamente però, più bob ci sono, più tempo perdiamo per disegnarli). “Che bello”, direte voi, “possiamo iniziare a fare un gioco!”. Un attimo, non esaltiamoci troppo. Siamo proprio sicuri di saper fare con i BOB le stesse cose che possiamo fare con gli sprites?

Guardiamo `lezione9i2.s` e il suo “gemello” in formato interleaved `lezione9i2r.s`.

Abbiamo un BOB colorato che spostiamo liberamente con il mouse sullo schermo. Però c'è un problema... muovendo il BOB cancelliamo lo sfondo! Questo con gli sprite non accade, in quanto gli sprite sono dei piccoli bitplanes separati dai bitplanes dello sfondo. I BOB invece vengono disegnati proprio sui bitplane dell'immagine di sfondo, quindi in parte la sovrascrivono.

Una prima soluzione al problema la presentiamo negli esempi `lezione9i3.s` e `lezione9i3r.s` (naturalmente il secondo è la versione rawblit del primo).

Come vedrete, però non è ancora soddisfacente.

Nell'esempio `lezione9i4.s` proviamo un'altra soluzione, ma anch'essa presenta problemi.

Nell'esempio `lezione9i5.s`, invece vediamo un esempio di bob mosso dal joystick che esce parzialmente dallo schermo.

Abbiamo iniziato a conoscere i BOB, ma per ora non abbiamo raggiunto un risultato soddisfacente, cioè riuscire a fare con i BOB le operazioni tipiche videogiochi a causa del problema dello sfondo. Purtroppo con quello che sappiamo finora non si può fare di meglio.

Ma non preoccupatevi: ci sono ancora molte cose da imparare sul blitter, e una di queste ci aiuterà a risolvere il problema! Forza e coraggio dunque, la strada è ancora lunga!

9.6 Copia di zone di memoria sovrapposte

Illustreremo ora un'altra caratteristica del blitter prendendo spunto dalla copia di rettangoli, operazione che ormai conosciamo bene. Cosa succede se la sorgente e la destinazione della blittata sono sovrapposte, ovvero sono 2 rettangoli di word che hanno delle parti in comune? E' ovvio che la blittata modificherà tutta la destinazione, comprese quindi le parti in comune con la sorgente. La copia tra zone sovrapposte consiste quindi nel mettere il contenuto della sorgente PRIMA della copia nella destinazione. Dopo la copia, il contenuto della sorgente sarà cambiato.

Pertanto, dopo la copia, la destinazione NON sarà uguale alla sorgente!! Piuttosto, ripetiamo, essa sarà uguale a come era la sorgente PRIMA della copia!

Insomma immaginate che la destinazione sia una fotografia scattata alla sorgente, e che durante il tempo impiegato dal fotografo per sviluppare la foto, la sorgente sia invecchiata rapidamente tanto da apparire molto diversa da come appare nella foto. È sempre effettuare una copia in tali condizioni? Dobbiamo studiare bene il problema. Vediamo cosa succede con un esempio di una copia di un rettangolo alto 2 righe e largo 3 words. Supponiamo che la sorgente si trovi più in basso della destinazione, come illustrato dalla figura seguente:

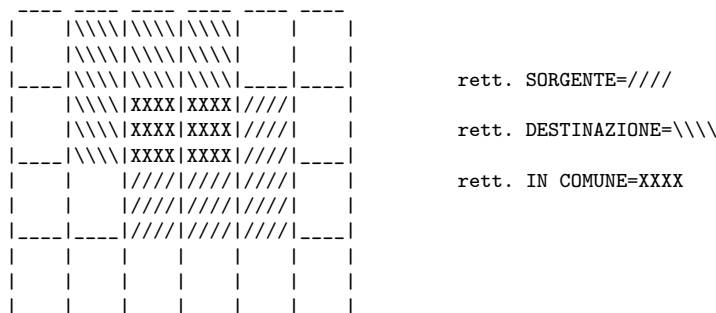


Fig. 20 Blittata tra rettangoli sovrapposti

Analizziamo, con l'aiuto di una serie di figure, le successive fasi dell'operazione. Indichiamo con le lettere A, B, C, D, E, F il contenuto delle 6 words che vogliamo copiare, e con il simbolo “?” il contenuto delle word che non ci interessano, e che quindi possiamo anche cancellare. Prima di iniziare la copia, abbiamo questa situazione:

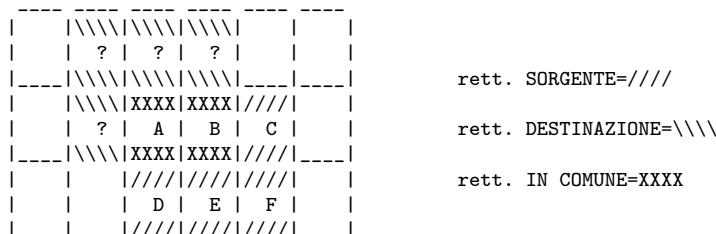
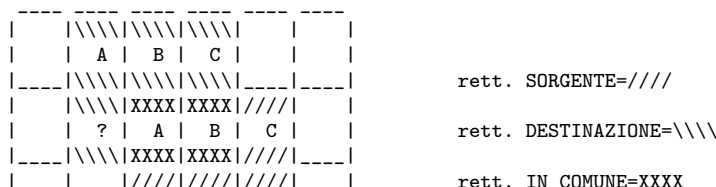


Fig. 21a Blittata tra rettangoli sovrapposti

Come sappiamo il blitter copia le words una alla volta partendo da quella più in alto a sinistra e proseguendo verso il basso e verso destra. La prima riga viene letta e copiata su una zona della destinazione non in comune, e che quindi possiamo sovrascrivere tranquillamente. Ecco la situazione dopo la copia della prima riga:



```

|   |   |   | D | E | F |   |
|___|___|___|___|___|___|___|

```

Fig. 21b Blittata tra rettangoli sovrapposti

A questo punto dobbiamo copiare la seconda riga. La seconda riga della destinazione si sovrappone con la prima riga della sorgente. Questo significa che quando scriveremo i dati nella destinazione, sovrascriveremo una parte della sorgente, distruggendone il contenuto. Osservate però che i dati sovrascritti appartengono alla PRIMA riga della sorgente, che noi abbiamo già copiato, e che quindi non ci serve più. Pertanto non ci sono problemi. La situazione dopo la copia della seconda (e ultima) riga è la seguente:

```

|   |   |   |   |   |   |   |
|   | A | B | C |   |   |   |
|___|___|___|___|___|___|___|
|   |   |   |   |   |   |   |
|   | D | E | F | C |   |   |
|___|___|___|___|___|___|___|
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|___|___|___|___|___|___|___|

```

```

rett. SORGENTE=////
rett. DESTINAZIONE=\\
rett. IN COMUNE=XXXX

```

Fig. 21c Blittata tra rettangoli sovrapposti

Abbiamo ottenuto proprio ciò che volevamo, in quanto ora il rettangolo destinazione è la copia esatta del contenuto del rettangolo sorgente PRIMA che iniziassimo la blittata. Notate che ora, invece il contenuto della sorgente è cambiato, ma ciò era inevitabile.

Potete vedere tutto ciò in pratica nell'esempio lezione911.s.

Sembrerebbe dunque che la sovrapposizione tra sorgente e destinazione non crei problemi. Proviamo però ad esaminare il caso in cui la destinazione si trovi più in basso della sorgente:

```

|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|___|___|___|___|___|___|___|
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|___|___|___|___|___|___|___|
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|___|___|___|___|___|___|___|

```

```

rett. SORGENTE=////
rett. DESTINAZIONE=\\
rett. IN COMUNE=XXXX

```

Fig. 22 Blittata tra rettangoli sovrapposti

Prima della blittata, la situazione è la seguente:

```

|   |   |   |   |   |   |   |
|   | A | B | C |   |   |   |
|___|___|___|___|___|___|___|
|   |   |   |   |   |   |   |
|   |   |   |   |   |   |   |
|___|___|___|___|___|___|___|

```

```

rett. SORGENTE=////

```

		D		E		F		?			rett. DESTINAZIONE=\\
	----		////		XXXX		XXXX		\\\\		rett. IN COMUNE=XXXX
			\\\\		\\\\		\\\\		\\\\		
			?		?		?				
	----		----		\\\\		\\\\		\\\\		

Fig. 23a Blittata tra rettangoli sovrapposti

Iniziamo con il copiare la prima riga. La prima riga della destinazione è parzialmente sovrapposta con la seconda riga della sorgente, che non è ancora stata copiata. Ecco quello che si ottiene:

	----		////		////		////				rett. SORGENTE=///
		A		B		C					rett. DESTINAZIONE=\\
	----		////		XXXX		XXXX		\\\\		rett. IN COMUNE=XXXX
		D		A		B		C			
	----		////		XXXX		XXXX		\\\\		
			\\\\		\\\\		\\\\		\\\\		
			?		?		?				
	----		----		\\\\		\\\\		\\\\		

Fig. 23b Blittata tra rettangoli sovrapposti

Come potete notare ci siamo persi i valori E ed F! Sembra proprio che stavolta la copia non riuscirà bene! Comunque copiamo anche la seconda riga, e vediamo che succede.

	----		////		////		////				rett. SORGENTE=///
		A		B		C					rett. DESTINAZIONE=\\
	----		////		XXXX		XXXX		\\\\		rett. IN COMUNE=XXXX
		D		A		B		C			
	----		////		XXXX		XXXX		\\\\		
			\\\\		\\\\		\\\\		\\\\		
			D		A		B				
	----		----		\\\\		\\\\		\\\\		

Fig. 23c Blittata tra rettangoli sovrapposti

Ecco fatto. La blittata è finita ma il risultato non è quello che volevamo. Siete convinti?

No? Allora, guardatevi l'esempio lezione912.s e convincetene!

Cerchiamo di capire perché la prima volta ha funzionato e stavolta no. Il problema nasce quando scriviamo sulle parti della destinazione che si sovrappongono con la sorgente, perché così facendo sovrascriviamo alcuni dati. Nel primo caso non ci sono stati problemi perché i dati sovrascritti li avevamo già copiati. Ciò è accaduto perché la sorgente si trova più in basso (ad indirizzi maggiori) della destinazione, e la sovrapposizione si verifica tra la prima riga della sorgente e la seconda riga della destinazione. Siccome il blitter copia partendo dalla prima riga, i dati della prima riga della sorgente vengono copiati PRIMA di essere sovrascritti dalla seconda riga della destinazione.

Nel secondo caso, invece, la sorgente si trova più in alto (ad indirizzi minori) della destinazione, e la sovrapposizione si verifica tra la seconda riga della sorgente e la prima della destinazione. I dati della seconda riga della sorgente, quindi, vengono sovrascritti durante la copia della prima riga, e cioè PRIMA di essere copiati a loro volta, pertanto vengono persi. Per risolvere questo problema, bisognerebbe copiare prima la seconda riga e poi la prima.

Ciò è possibile utilizzando il *modo discendente* del blitter. Quando si utilizza questo modo, il blitter esegue la copia (o una qualunque altra operazione) in senso inverso a quanto fa di solito, cioè parte dalla word più in basso a destra del rettangolo e prosegue verso sinistra e verso l'alto. Le words che blitta seguendo questo percorso hanno indirizzo via via minore. Si dice perciò che il blitter *discende* lungo la memoria, da cui il nome del modo di funzionamento (per contrasto, il modo normale viene detto anche MODO ASCENDENTE, infatti normalmente vengono blittate words con indirizzi via via crescenti). Prima di esaminare nel dettaglio come si usa il blitter in modo discendente, ritorniamo al problema della copia di regioni sovrapposte e verifichiamo che il modo discendente è la giusta soluzione. La situazione di partenza è la seguente:

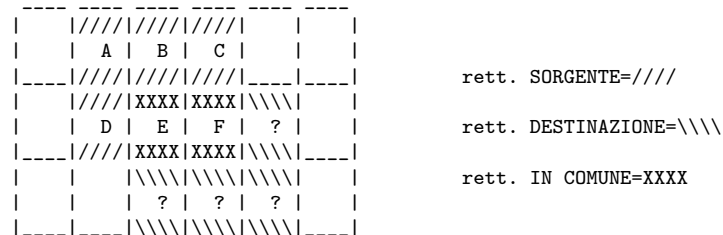


Fig. 24a Blittata tra rettangoli sovrapposti

Questa volta usiamo il modo discendente, perciò iniziamo a copiare a partire dall'ultima riga. In questo modo all'inizio non scriviamo sulla parte sovrapposta:

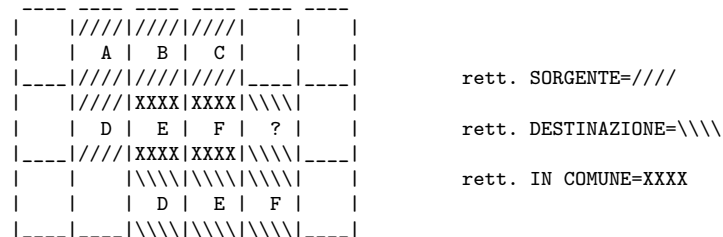


Fig. 24b Blittata tra rettangoli sovrapposti

Adesso copiamo la prima riga. Nel farlo sovrascriviamo la seconda riga della sorgente, ma poiché l'abbiamo già copiata non è un problema:

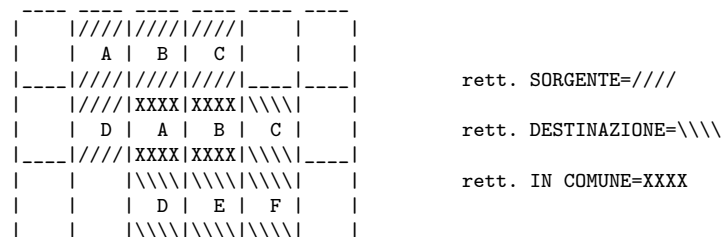


Fig. 24c Blittata tra rettangoli sovrapposti

OK! Questa volta ci siamo. Ora la destinazione ha lo stesso aspetto della sorgente prima di effettuare la blittata. Per concludere, possiamo quindi dire che quando effettuiamo una copia con sorgente e destinazione sovrapposte, se la sorgente si trova ad indirizzi di memoria maggiori

della destinazione, si deve fare la blittata in modo normale (*ascendente*), se invece la sorgente si trova ad indirizzi di memoria minori, si deve utilizzare il modo *discendente*.

A questo punto possiamo scendere nel dettaglio del modo discendente. Innanzitutto, il modo discendente va attivato mediante un bit di controllo. Si tratta del bit 1 del registro BLTCON1, che se settato ad 1 attiva il modo discendente, mentre quando viene azzerato (come abbiamo fatto finora) attiva il modo ascendente. Come abbiamo già detto, in modo discendente il blitter va “all’indietro” cioè si sposta tra locazioni di memoria di indirizzo via via minore. Per questo è necessario che i puntatori dei canali DMA puntino all’inizio della blittata alla word della blittata che ha indirizzo maggiore di tutte, cioè la prima word che verrà blittata. Si tratta, come sapete, della word più in basso e più a destra del rettangolo di words che verrà blittato. Per esempio, nel caso in cui si voglia blittare un rettangolo largo 3 words e alto 2 righe, si dovranno inizializzare i puntatori con l’indirizzo della terza word della seconda riga del rettangolo, che nella figura è indicata con due asterischi (**)

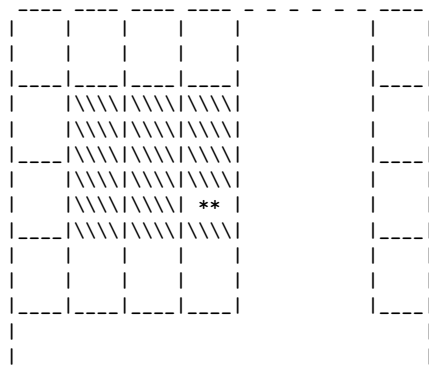


Fig. 25 Rettangolo di word con evidenziata la word da puntare all’inizio della blittata

Per calcolare l’indirizzo di tale word si segue un ragionamento simile a quello fatto nel caso ascendente. Dobbiamo calcolare la distanza (offset) di tale word dall’inizio del bitplane. Supponiamo di conoscere le coordinate X_a e Y_a del pixel più in alto a sinistra del rettangolo, ed anche la larghezza in words L e l’altezza A del rettangolo. La word che ci interessa appartiene all’ultima riga del rettangolo che ha coordinata $Y_b = Y_a + A$. L’offset della prima word di tale riga è dato dalla seguente formula:

$OFFSET_Y = 2 * (Y_b * NUMERO_WORDS_PER_RIGA)$ nel caso normale e

$OFFSET_Y = 2 * (Y_b * NUMERO_WORDS_PER_RIGA * NUMERO_PLANES)$ nel caso interleaved.

Ora dobbiamo calcolare la distanza tra la prima word della riga e l’ultima word del rettangolo. Come sappiamo tale distanza è data da $2 * (X_a / 16)$. D’altronde tra la prima e l’ultima word del rettangolo ci sono $L-1$ words, che equivalgono ad una distanza (che si esprime in bytes) di $2 * (L-1)$. Sommando le 2 differenze abbiamo:

$OFFSET_X = 2 * (X_a / 16 + L - 1)$.



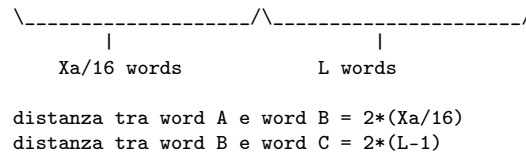


Fig. 26 Calcolo OFFSET_X

Quindi l'indirizzo da scrivere nei puntatori dei canali DMA è dato da:

```
INDIRIZZO_WORD = INDIRIZZO_BITPLANE+OFFSET_Y+OFFSET_X.
```

Per quanto riguarda i moduli e la dimensione della blittata, non ci sono differenze rispetto al caso ascendente, vengono calcolati tutti con le stesse formule. Ora possiamo finalmente copiare correttamente 2 regioni rettangolari sovrapposte anche quando la sorgente inizia ad un indirizzo di memoria minore di quello della destinazione: si tratta dell'esempio lezione9l3.s.

In modo discendente le maschere e lo shift si comportano diversamente rispetto al modo ascendente. Le maschere funzionano sempre nello stesso modo, ma cambiano le word a cui si applicano. La maschera contenuta in BLTAFWM viene applicata, come per il caso ascendente, alla prima word che blittiamo di ogni riga. Siccome però in modo discendente blittiamo al contrario, la prima word è la word più a destra del rettangolo, mentre in modo ascendente è la word più a sinistra. Allo stesso modo, la maschera contenuta in BLTALWM viene sempre applicata all'ultima word blittata di ogni riga, solo che in modo discendente tale word è la word più a sinistra. In sintesi:

- In modo ascendente (normale) BLTAFWM si applica alla word più a sinistra e BLTALWM alla word più a destra.
- In modo discendente BLTAFWM si applica alla word più a destra e BLTALWM alla word più a sinistra.

Se guardiamo l'immagine come appare sul video, passando al modo discendente le maschere si scambiano le colonne su cui operano. Per verificarlo caricate ed eseguite l'esempio lezione9m1.s che fa esattamente le stesse cose di lezione9h1.s, solo che opera in modo discendente. Vedrete che le maschere producono gli stessi effetti ma scambiando le colonne.

Lo shift, in modo discendente presenta una differenza fondamentale: viene fatto verso SINISTRA, anziché verso destra. Se specifichiamo un valore di shift pari per esempio a 2, la sorgente viene shiftata di 2 pixel **verso sinistra**. Utilizzando questa caratteristica possiamo realizzare l'effetto di scorrimento di un'immagine verso sinistra. Lo trovate nell'esempio lezione9m2.s.

A questo punto, siamo finalmente in grado di realizzare uno degli effetti più classici delle demo: lo *ScrollText*, ovvero un testo che scorre sullo schermo da destra verso sinistra.

Un semplice ma significativo esempio è la lezione9n1.s, nella quale troverete tutte le spiegazioni. Mi raccomando studiate con particolare attenzione questo esempio perché saper fare uno scrolltext è assolutamente fondamentale per un demo-coder!

Nell'esempio Lezione9n2.s troverete lo scrolltext dell'intro del disco1.

Avete capito come funziona lo scrolltext? Se la risposta è affermativa, potete iniziare ad essere soddisfatti di quanto avete imparato. Ormai conoscete il funzionamento di base del blitter. Nella prossima lezione scopriremo i segreti più nascosti di questo potente amico, a partire dal più grande, il più difficile da capire, con il quale abbiamo avuto a che fare per tutta questa lezione ma che abbiamo sempre evitato: il funzionamento dei *MintTerms*!

CAPITOLO 10

LEZIONE 10 - BLITTER AVANZATO

In questa lezione apprenderemo l'uso delle caratteristiche più avanzate del blitter.

10.1 I MINTERMS

Nella lezione 9 abbiamo detto che il blitter ci permette di effettuare diversi tipi di operazioni. Abbiamo anche detto che il tipo di operazione è definito dai *MINTERMS*, che sono i bit da 0 a 7 del registro BLTCON0, ovvero il byte basso (detto byte LF - Logic Function) di tale registro. A seconda del valore che viene scritto in tali bit, cambia l'operazione realizzata dal blitter. Per esempio sappiamo che per cancellare la memoria il byte LF va settato al valore \$00, mentre per copiare dal canale A al canale D al valore \$f0. Questi valori non sono stati scelti a casaccio dai progettisti del blitter, ma seguono una logica ben precisa, che ora spiegheremo.

Innanzitutto precisiamo che le operazioni effettuabili dal blitter sono operazioni LOGICHE, ovvero NOT, AND e OR, che ormai dovrete conoscere bene (in realtà c'è anche chi riesce a farci operazioni aritmetiche, ma ne parleremo, forse, nel prossimo disco!). Il blitter inoltre può combinare diverse operazioni di questo tipo in una unica blittata. Ma andiamo con ordine. Come sapete il blitter ha 3 canali di ingresso e uno di uscita. Per il momento non preoccupiamoci della abilitazione o disabilitazione dei canali. Una blittata è un'operazione logica che prende 3 valori in ingresso attraverso i 3 canali A,B,C e produce un risultato attraverso il canale D. Come tutte le operazioni logiche, essa viene effettuata bit-a-bit, anche se il blitter legge (e scrive) sempre delle word, esattamente come fa il 68000 con un'istruzione logica tipo AND. Quindi ogni bit della word in uscita viene calcolato in base ai valori dei corrispondenti bit delle word in ingresso. I 3 bit in ingresso possono dar luogo a 8 diverse combinazioni. Un'operazione blitter viene definita stabilendo, per ogni possibile combinazione dei bit in ingresso, se il risultato in uscita sarà 0 oppure 1. In pratica, ad ognuno degli 8 minterms (bit da 0 a 7 di BLTCON0) viene associata una diversa combinazione dei bit in ingresso; se il minterm vale 0, vuol dire che la combinazione in ingresso produce come risultato 0, se invece vale 1, il risultato sarà 1.

Questo può essere visualizzato con una tavola di verità, come mostrato sotto. Sono listati i tre canali di sorgente, e i valori possibili per un singolo bit di ognuno. A fianco è riportato il bit associato ad ogni combinazione.

A	B	C	posizione BLTCONO
-	-	-	-----
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

Fig. 27 MINTERMS

Per esempio, se vogliamo che una blittata produca un'uscita pari a 1 quando l'ingresso A vale 0, il B vale 1 e il C vale 0, e che invece produca un uscita pari a 0 in tutti gli altri casi, dobbiamo settare a 1 il minterm 2, e azzerare tutti gli altri minterms. Quindi scriveremo il valore \$04 nel byte LF. Per un altro esempio, il valore \$80 (= 1000 0000 binario) in LF pone a 1 solo i bit della destinazione per i quali i bit corrispondenti delle sorgenti A, B, e C sono tutti settati a 1. Tutti gli altri bit della destinazione a cui corrispondono altre combinazioni per A, B, e C, sono azzerati. Questo perché i bit dal 6 allo 0 del byte LF assumono il valore 0. Naturalmente è possibile settare a 1 più di un minterm contemporaneamente. Per esempio se poniamo LF al valore \$42 (= 0100 0010 in binario) "accendiamo" 2 minterms. Quindi con questo valore avremo un uscita pari a 1 in 2 casi: nel caso in cui A=0, B=0 e C=1 (corrispondente al bit 1 di LF) e nel caso A=1, B=1 e C=0 (corrispondente al bit 6 di LF). Negli altri casi avremo un uscita pari a 0. Cerchiamo ora di capire il significato dei valori dei minterms che abbiamo usato per la cancellazione e la copia. Nel caso della cancellazione si ha LF=\$00.

Tutti i minterms valgono 0. Questo significa che per qualunque combinazione dei canali sorgente, viene prodotto in uscita sempre uno 0. In pratica qualsiasi cosa leggiamo, scriviamo sempre 0, cioè cancelliamo (In realtà durante la cancellazione non leggiamo nulla perché non abilitiamo i canali A, B e C, ma dobbiamo comunque mettere LF=\$00, spiegheremo il perché in seguito). Per eseguire una copia da A a D, poniamo, come sapete, LF=\$F0 (= %11110000). In questo modo l'uscita vale 1 in corrispondenza di 4 diverse combinazioni, mentre vale 0 nelle restanti 4. Come potete leggere nella tabella di fig.27, le combinazioni corrispondenti ai minterms che abbiamo settato a 1, sono tutte le combinazioni possibili con A=1, e allo stesso modo le combinazioni corrispondenti ai minterms settato a 0, sono quelle con A=0. Ciò significa che tutte le volte che A=1, l'uscita vale 1 e quando invece A=0 l'uscita vale 0, indipendentemente dal valore di B e di C. In pratica cioè l'uscita assume lo stesso valore del canale A, e quindi ne è la copia esatta. Se volessimo invece copiare dal canale B al canale D, dovremmo usare un diverso valore di LF, ponendo a 1 i minterms che corrispondono alle combinazioni con B=1 (che come si legge nella fig.27 sono i minterms 2,3,6 e 7) e azzerare gli altri (minterms 0, 1, 4 e 5), ottenendo LF=\$CC (= %11001100). Programmando opportunamente i minterms si possono fare molte operazioni con il blitter.

Supponiamo per esempio di voler settare ad 1 tutti i pixel di un rettangolo (in pratica l'operazione inversa della cancellazione che invece setta tutti i bit a 0). Come per la cancellazione utilizziamo solo il canale di uscita. Quello che vogliamo è che l'uscita sia sempre 1, per qualun-

que combinazione degli ingressi. Per ottenere questo risultato poniamo ad 1 tutti i minterms, ottenendo $LF = \$FF$.

Potete vedere questa operazione nell'esempio lezione10a1.s.

Nell'esempio lezione10a2.s mostriamo invece l'operazione NOT. Vi rimandiamo al listato per la spiegazione.

Passiamo ora ad un esempio di operazione a 2 operandi, per esempio l'OR. Vogliamo che l'uscita sia pari all'OR dei canali A e B. Ripensando alla tabella della verità dell'OR, si capisce che l'uscita deve valere 1 in tutti i casi in cui $A=1$ e in tutti i casi in cui $B=1$. Come potete vedere dalla fig. 27 in totale si tratta di 6 casi che danno luogo a $LF = \$FC$.

L'esempio lezione10b1.s mostra appunto un'operazione di OR, mentre l'esempio lezione10b2.s effettua un'operazione di AND.

Un altro modo per calcolare il byte LF che realizza una particolare operazione è attraverso l'uso di diagrammi di Venn:

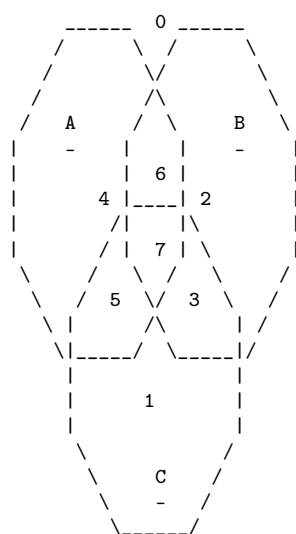


Fig. 28 Diagramma di Venn

Illustriamo l'uso di tale diagramma attraverso alcuni esempi

1. Per selezionare una funzione $D=A$ (cioè destinazione = sorgente A solamente), selezionare solo i minterms che sono totalmente inclusi dal cerchio A nella figura sopra. Questa è la serie di minterms 7, 6, 5, e 4. Quando sono scritti come una serie di 1 per i minterm selezionati e di 0 per quelli non selezionati, il valore diventa:

Numero Minterm	7	6	5	4	3	2	1	0
Minterm selezionati	1	1	1	1	0	0	0	0

	F				0	ossia \$F0		

2. Per selezionare una funzione combinazione di due sorgenti, cercare i minterms da entrambi dei cerchi (la loro intersezione). Per esempio, la combinazione A "AND" B è rappresentato dall'area comune ai cerchi A e B, ossia i minterms 7 e 6.

$D = \bar{A}C$	\$50	$D = \bar{\bar{A}}\bar{\bar{C}}$	\$11
$D = \bar{A}\bar{C}$	\$0A	$D = \bar{A} + \bar{B}$	\$F3
$D = \bar{A}C$	\$05	$D = \bar{A} + \bar{B}$	\$3F
$D = A + B$	\$FC	$D = \bar{A} + \bar{C}$	\$F5
$D = \bar{A} + B$	\$CF	$D = \bar{A} + \bar{C}$	\$5F
$D = A + C$	\$FA	$D = \bar{B} + C$	\$DD
$D = \bar{A} + C$	\$AF	$D = \bar{B} + \bar{C}$	\$77
$D = B + C$	\$EE	$D = AB + \bar{A}C$	\$CA
$D = \bar{B} + C$	\$BB		

Fig. 29 Mintems più usati

NOTA: Per individuare il valore desiderato di LF per i vostri scopi potete usare anche l'utility "minterm", programmata da Deftronic, lo stesso del Trash'M'One. La breve utility in questione la trovate in questo disco. La sintassi è questa: per il NOT, si mette la lettera del canale non shiftata (lowercase), ad esempio "abc". Per il canale normale si usa la lettera shiftata (uppercase). Due lettere adiacenti significano un AND tra i canali, mentre se sono separate dal "+" significa un OR tra i canali.

Esempio: se si vuole $\bar{\bar{A}}\bar{\bar{B}}C$:

minterm Abc

risultato: \$10

Esempio2: se si vuole solo la sorgente A:

minterm A

risultato: \$F0 (come volevasi dimostrare)

Esempio3: se si vuole solo (A AND B) OR C:

minterm AB+C

risultato: \$DA.

10.2 I BOBs

Siamo quasi arrivati al piatto forte della lezione, ovvero i BOB. Prima di affrontarli è necessario presentare un'altra idea: il bit-plane maschera. Si tratta semplicemente di un bitplane che costituisce "l'ombra" di una figura, cioè un bitplane delle stesse dimensioni di una figura che ha settati ad 1 i pixel corrispondenti a pixel della figura, colorati con un colore diverso dallo sfondo, e invece settati a 0 i pixel che corrispondono al colore di sfondo della figura. Per esempio consideriamo la seguente tabella di numeri:

```
0020
0374
5633
0130
```

essa rappresenta un'immagine ad 8 colori (3 bitplanes) larga 4 pixel e alta 4 righe. Ogni numero indica il colore associato al pixel. La maschera di tale immagine è la seguente:

```
0010
0111
1111
0110
```

Osserviamo che i colori diversi dallo 0 (lo sfondo) hanno almeno un bitplane settato ad 1.

Pertanto la maschera può essere costruita a partire dalla figura facendo l'OR di tutti i bitplanes, come illustrato negli esempi `lezione10c1.s` e `lezione10c2.s` che vi permettono anche di ripassare l'utilizzo del blitter per fare operazioni logiche. In particolare, in `lezione10c2.s` mostriamo per la prima volta una blittata che usa tutti e 4 i canali del blitter.

Il Kefrens Converter, comunque, ha un'opzione per creare automaticamente la maschera di una figura. I bitplanes maschera sono utili perché ci permettono di visualizzare delle parti di un'immagine, in base alla forma di un'altra immagine.

Ne vediamo esempi in `lezione10c3.s` e `lezione10c4.s`, dove utilizziamo una maschera a forma di cerchio per realizzare un riflettore che illumina un'immagine rendendone visibile una parte.

I 2 esempi, benché realizzino lo stesso effetto utilizzano tecniche molto diverse, come spiegato nei commenti. Studiate particolarmente bene `lezioneq4.s`, che è indispensabile per capire poi i BOB. In questo esempio, il bitplane maschera viene utilizzato per "selezionare" delle parti di un'immagine di 5 bitplanes. La selezione avviene effettuando una operazione di AND tra il bitplane maschera e i 5 bitplanes che costituiscono l'immagine. Poiché l'immagine è in formato normale, vengono effettuate 5 blittate distinte, una per ogni plane. La maschera, ovviamente è sempre la stessa per ogni blittata (è formata da un solo bitplane). Volendo applicare la tecnica dell'esempio `lezione10c4.s` ad uno schermo in formato interleaved, ci troviamo di fronte ad un problema. Quando operiamo in questo formato, infatti, blittiamo tutti i planes contemporaneamente. La maschera però ha la dimensione di un plane, e quindi non può essere usata in una blittata che ha una dimensione pari al numero di planes di cui si compone l'immagine. Per risolvere questo problema dobbiamo modificare la nostra maschera. Siccome ogni riga della maschera deve selezionare la riga corrispondente di **tutti** i bitplanes della figura, dobbiamo ripetere la riga tante volte quanti sono i bitplanes. In formato interleaved, quindi, dobbiamo usare un bitplane maschera che ha ogni riga ripetuta tante volte quanti sono i bitplane della figura. Nel caso della figura che abbiamo visto prima (3 planes) la nostra maschera interleaved è la seguente:

```
0010\
0010 | - prima riga della maschera normale ripetuta 3 volte
0010/
0111
0111
0111
1111
1111
1111
0110
0110
0110
```

Come potete notare, poiché la figura ha 3 bitplanes, ogni riga della maschera in formato normale è stata ripetuta 3 volte per ottenere la maschera interleaved. Il formato interleaved, quindi, ci costringe ad utilizzare una maschera che occupa più memoria di quella richiesta dal formato normale.

L'esempio `lezione10c5.s` è la versione interleaved di `lezione10c4.s`, e ci permette di vedere in pratica quanto detto.

Se avete capito bene il funzionamento delle maschere, siete pronti per risolvere una volta per tutte il problema dello sfondo con i BOBS. Come sicuramente ricordate, nell'esempio `lezione9i3.s` siamo andati abbastanza vicini alla soluzione del problema. Lo sfondo viene salvato e successivamente ridisegnato al suo posto. L'unico problema è che nel rettangolo che racchiude la figura del BOB viene cancellato lo sfondo, e sostituito con il colore 0. In realtà quando disegniamo un BOB usiamo il colore 0 non come un colore qualsiasi ma semplicemente per denotare i pixel del rettangolo che non appartengono alla figura del BOB. E' esattamente la stessa cosa che facciamo con gli sprite, usiamo il colore 0 come "trasparente". Quando disegniamo il BOB sullo schermo vorremmo che al posto dei pixel colorati con il colore 0 apparisse lo sfondo, in pratica dovremmo poter scrivere sullo schermo solo i pixel di colore diverso da 0. Ciò non è possibile perché come sapete il blitter scrive (e legge) SEMPRE delle word INTERE.

Si adotta dunque una diversa strategia. Invece di fare una semplice copia del BOB sulla destinazione, facciamo una blittata più complicata. Leggiamo dalla memoria, oltre al BOB, anche lo sfondo, li "mischiamo" assieme, in modo che al posto dei pixel di colore 0 del BOB appaiano i pixel dello sfondo, e scriviamo il risultato sullo schermo. La strategia è illustrata nella figura seguente, nella quale abbiamo un BOB e un pezzo di sfondo di 6*8 pixel. Il simbolo "." rappresenta un pixel di colore 0, il simbolo "#" rappresenta un pixel del BOB di diverso colore, e il simbolo "o" rappresenta un pixel dello sfondo di diverso colore:

BOB	SFONDO
.....	...0....
..####..	...oo...
.#.#.#.	..oooo..
..####..	..ooooo.
...##...	.ooooooo
..#..#..	oooooooo

\

 \

 \

/

 /

 /

BOB sovrapposto a SFONDO

```

...0....
..####..
.#o#o#.#
..####o.
.oo#ooo
oo#oo#oo
  
```

Fig. 30 Bob e sfondo

In questo modo otteniamo l'effetto desiderato. Resta da capire in che modo "mischiare" il BOB con lo sfondo. Per "mischiare" correttamente dobbiamo sapere quali pixel del BOB sono di colore 0 e quali no. Queste informazioni sono contenute nel bitplane maschera del BOB, che come sapete ha un bit a 0 per ogni pixel di colore 0 del BOB e un bit a 1 per ogni pixel di altro colore. L'operazione di mescolamento avviene dunque nel modo seguente:

- Per ogni pixel, leggiamo la maschera
- Se la maschera ha valore 1, copiamo il corrispondente pixel del BOB
- Se la maschera ha valore 0, copiamo il corrispondente pixel dello sfondo.

Possiamo realizzare questa procedura mediante una sola blittata, operando nel modo seguente: leggiamo la maschera attraverso il canale A del blitter, il BOB attraverso il canale B, lo sfondo attraverso il C, utilizziamo la maschera per selezionare i pixel da copiare (o dallo sfondo o dal BOB) e scriviamo il risultato nel canale D (l'assegnamento dei canali non è casuale). La selezione viene effettuata mediante la seguente equazione logica:

$$D = (A \text{ AND } B) \text{ OR } ((\text{NOT } A) \text{ AND } C)$$

Questa equazione si comporta esattamente come la procedura di selezione descritta in precedenza. Quando infatti la maschera $A = 1$ (cioè abbiamo un pixel del BOB di colore DIVERSO da 0) l'equazione si semplifica nel modo seguente:

$$D = (1 \text{ AND } B) \text{ OR } ((\text{NOT } 1) \text{ AND } C) = B \text{ OR } (0 \text{ AND } C) = B \text{ OR } 0 = B$$

Quindi viene copiato il pixel del BOB. Quando invece $A = 0$ (cioè abbiamo un pixel del BOB di colore 0) l'equazione diventa:

$$D = (0 \text{ AND } B) \text{ OR } ((\text{NOT } 0) \text{ AND } C) = 0 \text{ OR } (1 \text{ AND } C) = 0 \text{ OR } C = C$$

Quindi viene copiato il pixel dello sfondo. Questa equazione logica viene eseguita dal blitter (come potete calcolare voi stessi) ponendo $LF = \$CA$, valore noto come *cookie cut* ovvero “taglio del biscotto”. Come abbiamo accennato prima, l'assegnamento dei canali è stato fatto accuratamente sulla base delle caratteristiche dei canali stessi. Infatti per effettuare spostamenti fluidi orizzontali è necessario usare per il BOB e per la maschera lo shift del blitter; per questo il canale C (che non può fare lo shift) viene usato per lo sfondo. Inoltre, applichiamo il trucco di mascherare l'ultima word al bitplane maschera, in modo che l'ultima word di esso venga azzerata, facendo così che nell'ultima word venga blittato lo sfondo.

Gli esempi `lezione10d1.s` e `lezione10d1r.s` mostrano (rispettivamente in versione normale e interleaved) il tanto atteso BOB che si muove su uno sfondo.

10.3 La velocità del Blitter (e non solo)

È giunto il momento di occuparci di una questione molto importante: la velocità del blitter. Infatti come sapete, il blitter impiega una certa quantità di tempo per portare a termine i suoi compiti, ed è necessario tenere conto di ciò quando si programmano effetti complessi. Per misurare la velocità del blitter useremo una tecnica molto semplice, nota come “copper monitor”, che ci mostra il risultato sullo schermo in tempo reale. La tecnica è semplicissima: utilizziamo un certo colore (di solito il nero) come sfondo. Poi, subito prima di iniziare la blittata cambiamo il colore di sfondo col processore, tramite un `MOVE.W #$xxx,$dff180`. Quando la blittata finisce rimettiamo lo sfondo al colore iniziale. In questo modo sappiamo che la blittata impiega un tempo proporzionale alla porzione di schermo colorata diversamente. Da notare che questa tecnica è usata per misurare qualsiasi tipo di routine, e in particolare è utilissima per capire quando diventa più veloce o più lenta a seguito di una modifica, ad esempio un'ottimizzazione. Un esempio è illustrato in `lezione10e1.s`.

In questo esempio usiamo il blitter per copiare un rettangolo sullo schermo. Sulla base di questo esempio possiamo iniziare a fare un pò di considerazioni sulla velocità del blitter. Innanzitutto, come avevamo già accennato, la velocità dipende dalle dimensioni della blittata. Provate nell'esempio a modificare l'altezza e/o la larghezza del rettangolo, e ve ne renderete conto da soli. Ciò è ragionevole, in quanto più è grande il rettangolo, maggiore è la quantità di words da spostare. Allo stesso modo, il numero di bitplanes influenza la velocità (provate in lezione10e1.s a cambiare il numero di iterazioni della routine "DisegnaOggetto"), in quanto più bitplanes ci sono e maggiore è la quantità di dati da spostare. L'esempio lezione10e1r.s è la versione rawblit dell'esempio precedente.

Eseguendolo noterete che esso risulta più veloce ma di pochissimo. Ma allora, vi chiederete, tutto il vantaggio del rawblit? In realtà, come abbiamo già detto, la tecnica rawblit è conveniente non tanto perché accelera il blitter, ma piuttosto perché fa risparmiare tempo al processore. Nei 2 esempi che abbiamo visto finora abbiamo misurato solo il tempo impiegato dal blitter.

Negli esempi lezione10e2.s e lezione10e2r.s, invece, usiamo diversi colori per evidenziare sia il tempo impiegato dal blitter che quello impiegato dal processore.

Il confronto tra questi esempi ci mostra appieno i vantaggi del modo rawblit: con questa tecnica il processore è impiegato pochissimo, giusto il tempo di caricare i registri del blitter, e poi è libero di eseguire altri compiti, a differenza di quanto accade con il modo normale, dove il processore deve attendere la fine di una blittata per lanciare la blittata del plane successivo. È chiaro che per sfruttare il vantaggio della tecnica rawblit è necessario che la routine successiva alla blittata NON impieghi il blitter. Infatti se (come accade negli esempi) dopo una blittata c'è subito una routine che impiega il blitter, il processore dovrà comunque attendere che il blitter termini il suo compito, e non avremo dunque nessun vantaggio. Quindi un criterio da seguire per ottimizzare i programmi è quello di mettere, quando possibile, le routine che usano il blitter "distanti", ovvero intervallate da altre routine che non ne facciano uso, in modo che il blitter e il processore procedano in parallelo. C'è da dire però che questo criterio è valido soprattutto su macchine dotate di memoria fast, in quanto se il processore deve accedere alla memoria chip si generano dei conflitti nell'accesso alla memoria, di cui parleremo meglio tra un attimo.

Per il momento notiamo un'altra cosa sugli esempi lezione10e2.s e lezione10e2r.s: il blitter impiega circa lo stesso tempo per la cancellazione (schermo verde) e per il disegno (schermo rosso). Se ci pensate bene, questo fatto dovrebbe sembrarvi strano: infatti è vero che le 2 blittate hanno la stessa dimensione ma bisogna considerare che la cancellazione è una blittata che usa un solo canale, mentre la copia ne usa 2. È chiaro che all'aumentare del numero di canali, aumenta il numero di words lette e scritte dal blitter, quindi la blittata dovrebbe richiedere più tempo. Guardate però l'esempio lezione10e3.s.

Questo esempio è simile ai precedenti, ma invece di fare una semplice copia della figura effettua un'operazione di OR tra la figura e un plane azzerato. Naturalmente l'effetto è sempre lo stesso, ma potete notare come ora la routine, che effettua una blittata a 3 canali (D=A OR B) sia notevolmente più lenta. La velocità dipende da quali e quanti canali sono usati in una maniera abbastanza complicata, che può essere sintetizzata dalla seguente tabella:

bit 8-11 di BLTCONO	Canali usati	Sequenza di accesso alla memoria
F	A B C D	A0 B0 C0 - A1 B1 C1 D0 A2 B2 C2 D1 D2
E	A B C	A0 B0 C0 A1 B1 C1 A2 B2 C2
D	A B D	A0 B0 - A1 B1 D0 A2 B2 D1 - D2
C	A B	A0 B0 - A1 B1 - A2 B2
B	A C D	A0 C0 - A1 C1 D0 A2 C2 D1 - D2
A	A C	A0 C0 A1 C1 A2 C2
9	A D	A0 - A1 D0 A2 D1 - D2

8	A	AO - A1 - A2
7	B C D	B0 C0 - - B1 C1 D0 - B2 C2 D1 - D2
6	B C	B0 C0 - B1 C1 - B2 C2
5	B D	B0 - - B1 D0 - B2 D1 - D2
4	B	B0 - - B1 - - B2
3	C D	C0 - - C1 D0 - C2 D1 - D2
2	C	C0 - C1 - C2
1	D	D0 - D1 - D2
0	nessuno	- - - -

Questa tabella mostra per ogni combinazione di canali attivi, la sequenza di accessi alla memoria operata dal blitter, nel caso di una blittata di 3 words. Per ogni accesso è indicato il canale che lo effettua, e i trattini indicano cicli di bus non sfruttati dal blitter. Per esempio la stringa:

AO B0 - A1 B1 - A2 B2

Indica che prima accede al bus il canale A (A0) poi il B (B0), poi il blitter non utilizza un ciclo di bus (permettendo al processore di accedere alla memoria), poi tocca di nuovo al canale A (A1) e così via.

La tabella riportata in realtà è solo indicativa, perché non tiene conto di molti fattori, quali l'utilizzo di modi speciali del blitter e la competizione con il processore e con gli altri canali DMA (consultate al riguardo la lezione 8). Ciononostante è molto utile per avere un'idea di quali sono le combinazioni di canali migliori. Tenete presente che questa tabella è relativa ad una blittata di 3 words. Per blittate di più words il blitter ripete tante volte la sequenza di accessi che nella tabella sono "al centro". Per esempio, una blittata di 5 words che utilizza i canali A e D ha la seguente sequenza:

AO - A1 D0 A2 D1 A3 D2 A4 D3 A5 D4 - D5

Lo studio della tabella ci consente alcune osservazioni interessanti. Se guardiamo la sequenza relativa all'uso del solo canale D, vediamo che il blitter sfrutta il bus un ciclo sì e uno no. Al contrario, quando vengono usati i canali A e D, il blitter sfrutta (tranne che nei casi della prima e dell'ultima word) tutti i cicli di bus. Questo fatto ci spiega come mai negli esempi la routine di cancellazione (canale D) ha circa la stessa velocità della routine di disegno (canali A e D). Notate però che se facciamo una copia da B a D, le cose vanno diversamente.

Lo potete vedere in pratica nella lezione10e4.s.

Analogamente, consultando la tabella, si vede che nel caso di blittate con 2 sorgenti conviene usare A e B oppure A e C, ma non B e C perché vengono sprecati più cicli.

Dovete comunque ricordare che la velocità del blitter dipende anche dagli eventuali conflitti con altri canali DMA (video, audio, copper, processore) che possono "rubargli" cicli ritardandolo. Infatti, come abbiamo spiegato nella lezione 8, il blitter nell'accesso al bus ha priorità solo sulla CPU. Questo significa che se un altro dispositivo (es. il copper) vuole accedere alla RAM contemporaneamente al blitter, la precedenza spetta all'altro dispositivo. L'unico fesso che dà la precedenza al blitter è il processore. Anche qui però la priorità non è totale. Infatti il blitter, dando prova di grande generosità, se si accorge che il processore per 3 volte consecutive ha provato ad accedere al bus ma non c'è riuscito perché qualcun altro gli ha preso la precedenza, gli dice: "Passa tu per stavolta, vah" e gli concede il bus per un ciclo.

Questo meccanismo riduce la possibilità che in casi di sovraccarico del DMA il processore sia bloccato in attesa del bus troppo a lungo. È comunque possibile reprimere i moti di generosità del blitter. Settando a 1 il bit 10 (detto `blitter_nasty`, cioè blitter cattivo) del registro `DMA CON` il blitter non si comporterà più in questo modo, ma si prenderà la precedenza sul processore

ogni volta. Nel caso in cui le routine del nostro programma utilizzino tutte il blitter, quindi il processore non fa altro che caricare i registri e mettersi in attesa, conviene senz'altro settare a 1 tale bit. Ovviamente questo discorso ha senso in caso il cui il programma sia contenuto in memoria chip ed in assenza di caches, perché in caso contrario non si verificano conflitti tra il processore e il blitter per l'accesso alla RAM. Un esempio sul bit Blitter Nasty si trova in `lezione10e5.s`.

Per ottimizzare al massimo l'uso del blitter, dovete velocizzare al massimo la scrittura dei registri ad esso relativi. Negli esempi che abbiamo fatto sinora e anche in quelli che faremo nel resto della lezione infatti, per aumentare la chiarezza, non abbiamo ottimizzato la scrittura dei registri come avremmo potuto. Durante una blittata, gli unici registri che variano sono i registri `BLTxPT` e il `BLTSIZE`. I registri `BLTCONx`, `BLTxMOD` e `BLTxWM` rimangono costanti. Ciò significa che se il contenuto di tali registri non viene modificato da altre routine, non serve riscriverli all'inizio di ogni blittata. Un accorgimento da adottare per ottimizzare le routine nel caso in cui ci sono loop di blittate è quello di porre i valori da scrivere nei registri blitter in registri del processore, e sostituire all'interno del loop le `MOVE.W #YYY,$DDFxxx` con delle `MOVE.W Dx,$DDFxxx` che sono più veloci. Queste ottimizzazioni nella scrittura dei registri prese una per una danno incrementi di velocità davvero minimi, che con il copper monitor è difficile notare. Però in una demo con tanti effetti complessi, messe insieme hanno il loro peso.

A titolo di esempio guardate il listato `lezione10e6.s` che è una versione ottimizzata con questi trucchi di `lezione10c3.s`.

10.4 Il double buffering

Tutti gli esempi che abbiamo visto finora relativi ai bobs avevano sempre un solo bob che si muoveva sullo schermo. Proviamo ora a metterne di più. Per esempio, proviamo ad applicare la tecnica dello sfondo "finto": usiamo un bitplane per lo sfondo e 3 planes dove muovere i bobs. Poiché tutti i bobs si muovono sugli stessi bitplanes dovremo comunque disegnarli usando la tecnica del bitplane maschera. Avremo comunque il vantaggio di non dover salvare e ripristinare lo sfondo, perché i bitplanes dei bob inizialmente sono azzerati. Sarà quindi sufficiente cancellare questi plane ad ogni frame, prima di ridisegnare i bobs nelle nuove posizioni. Questa tecnica è applicata nell'esempio `lezione10f1.s`.

Eseguendo questo programma avrete però una brutta sorpresa: i bobs vengono disegnati correttamente solo nella parte bassa dello schermo, mentre in alto non vengono disegnati correttamente. Come mai? C'è qualche bug nelle nostre routines? No, le nostre routines vanno bene. Il problema è che sono troppo lente. Come ben sapete, mentre il nostro programma viene eseguito, il pennello elettronico disegna l'immagine sullo schermo.

Per far apparire un'immagine stabile, si cerca di modificare lo schermo (cioè cancellare, disegnare bobs, linee ecc.) durante il Vertical Blank, ossia nel periodo di tempo in cui il pennello elettronico è inattivo. Se però dobbiamo fare molte modifiche sullo schermo, può accadere che le nostre routines non siano abbastanza veloci da svolgere il proprio compito durante il Vertical Blank. È appunto quello che succede in questo caso. Aumentando il numero di bobs, aumenta il tempo necessario a disegnarli e di conseguenza non si riesce più a farlo durante il Vertical Blank. Il risultato è che a volte i bobs vengono disegnati sullo schermo DOPO che il pennello elettronico ha disegnato quella parte di schermo, e quindi i bobs non vengono visualizzati. Poiché il pennello elettronico va dall'alto verso il basso, più i bob sono disegnati in alto e più spesso ciò accade. Se guardate attentamente l'esempio, vedrete che la zona di schermo nella quale tutti i bobs vengono disegnati bene, è quella che viene visualizzata DOPO che le routines di disegno hanno finito il loro lavoro, come viene evidenziato dal copper monitor.

La tecnica del “double buffering” ci consente di risolvere questo problema. Si tratta di una tecnica di uso generale che potete impiegare con qualsiasi effetto, non solo con i bobs. In particolare lo useremo per le routines 3d. Questa tecnica consiste nell'utilizzare due schermi (detti appunto buffer) invece che uno solo. I due buffer vengono visualizzati alternativamente, un fotogramma l'uno e un fotogramma l'altro. Mentre viene visualizzato uno dei buffer, noi possiamo disegnare liberamente sull'altro, senza preoccuparci della stabilità, visto che l'immagine che viene visualizzata è quella del primo buffer che noi non modifichiamo. Quando si verifica il successivo Vertical Blank, i 2 buffer vengono scambiati. Quello sul quale noi abbiamo disegnato in precedenza viene visualizzato, mostrando le modifiche che abbiamo fatto, mentre il buffer che prima era stato visualizzato è ora a nostra disposizione per disegnarci sopra. Ripetendo lo scambio ad ogni Vertical Blank, avremo sempre a disposizione un buffer non visualizzato sul quale disegnare, senza preoccuparci di quello che fa il pennello elettronico. Grazie a questa tecnica, l'unica limitazione di tempo delle nostre routines di disegno è che esse devono concludersi prima che il pennello elettronico raggiunga la fine dello schermo. Questo ci da un tempo pari ad 1/50-esimo di secondo (in Pal, 1/60 in NTSC).

10.5 Uso dei canali Blitter non attivati

Vi sono casi in cui è utile far “partecipare” alla blittata anche i canali non attivi. Per capire bene cosa significa dovete sapere ancora una cosa sul blitter. Quando un canale di ingresso (A, B o C) è attivo, legge words dalla memoria. Ogni word dopo essere stata letta viene copiata in un apposito registro, detto registro dati del blitter. Ogni canale ha il suo registro dati, nel cui nome compare la lettera che identifica il canale: abbiamo dunque BLTADAT (canale A, \$DFF074), BLTBDAT (canale B, \$DFF072), BLTCDAT (canale C, \$DFF070) e BLTDDAT (canale D \$DFF000). La word dal registro dati viene successivamente sottoposta ad operazioni logiche con le words provenienti dagli altri canali, e il risultato viene scritto in memoria attraverso il canale D. Facciamo un esempio per capire bene. Consideriamo il caso di una blittata che esegua un AND tra i canali B e C. All'interno del blitter accadono le seguenti cose:

1. Il canale B legge una word e la copia in BLTBDAT
2. Il canale C legge una word e la copia in BLTCDAT
3. Viene eseguito un AND tra il contenuto di BLTBDAT e quello di BLTCDAT
4. Il risultato viene scritto attraverso il canale D
5. Si ripetono i passi da 1 a 4 per le successive words.

In realtà le cose funzionano un pò diversamente, perché alcune operazioni sono eseguite in parallelo per velocizzare il blitter, ma a livello logico le cose funzionano così, ed è quello che ci serve sapere. Che succede quando un canale è disabilitato? Naturalmente esso non leggerà nulla dalla memoria, pertanto il registro BLTxDAT corrispondente non verrà cambiato. Il contenuto di tale registro viene conservato, e può comunque essere usato in operazioni logiche. Inoltre tale registro può essere scritto anche dalla CPU, il che ci permette di settarlo a valori opportuni (non il registro BLTDDAT!). La situazione è simile a quella che abbiamo visto nella lezione 7 per gli sprite. Anche gli sprite hanno dei canali DMA (i registri SPRxPT) che copiano i dati letti in registri dati (SPRxDAT). In alcune applicazioni però è utile scrivere nei registri dati direttamente con il processore (o con il copper). Vediamo ora l'utilità di questa caratteristica del blitter. Per esempio consideriamo il caso in cui vogliamo riempire una serie di locazioni di memoria con un valore

costante, per esempio per disegnare sullo schermo un rettangolo non pieno, ma “a righine”, o come dicono i grafici con un “pattern” (cioè una trama).

Potremo risolvere il problema memorizzando il nostro rettangolo nella sezione dati del nostro programma e copiandolo con il blitter, esattamente come se fosse una figura come le altre. Una soluzione migliore, però, ci viene offerta dalla possibilità di disabilitare i canali del blitter. Per risolvere il problema possiamo infatti eseguire una copia dal canale A al D, TENENDO IL CANALE A disabilitato, e scrivendo il “pattern” nel registro BLTADAT. In questo modo otteniamo 2 vantaggi: non dobbiamo memorizzare il rettangolo tra i dati del nostro programma, quindi risparmiamo memoria, e, poiché il canale A è disabilitato, effettuiamo meno accessi alla memoria di quanti ne faremmo in caso di copia normale da A a D, dando pertanto al processore più possibilità di accesso alla RAM.

Per vedere in pratica questa applicazione caricate la lezione10g1.s.

È possibile applicare questa tecnica non solo per semplici copie di un valore costante, ma anche in operazioni logiche più complesse nelle quali un operando sia costante.

Trovate 2 esempi in lezione10g2.s e lezione10g3.s.

10.6 Il flag Zero e le collisioni

Questa è l'ultima caratteristica hardware del blitter da spiegare! Il blitter ha un flag, detto flag Zero, che ha un funzionamento analogo al flag Zero del processore. Questo flag è il bit 13 del registro DMAONR. Se una blittata produce come risultato TUTTI ZERI, il flag Zero viene settato a UNO. Al contrario, se almeno un bit in una delle word risultato ha valore 1, il flag assume valore ZERO. Il flag si comporta in questo modo anche nel caso in cui il risultato della blittata NON viene scritto in memoria, cioè quando il canale D è disabilitato.

Questo fatto è molto utile perché ci aiuta a rilevare collisioni tra un bob e un disegno sullo schermo (che può essere un altro bob già disegnato). Supponiamo per il momento di lavorare con immagini ad un solo bitplane. Per rilevare le collisioni effettuiamo (con il blitter) un'operazione di AND tra il bob e la parte di schermo su cui il bob si dovrebbe posizionare, MA non scriviamo il risultato da nessuna parte. Questa blittata serve solo per testare la collisione. Cosa succede quando eseguiamo un AND? Come sapete il risultato di un AND tra 2 bit è 1 solo nel caso in cui entrambi i bit operandi valgono 1. Nel nostro caso significa che un bit del risultato può valere 1 SOLO nel caso in cui coincidono nella stessa posizione un bit del bob di valore 1 e un bit dell'immagine di valore 1. Ma ciò vuol dire che tali bit producono una collisione. Quindi se c'è una collisione, almeno un bit del risultato avrà valore UNO, e in corrispondenza il flag Zero assumerà valore ZERO. Al contrario, se non si verifica collisione, nessun bit del bob coincide con un bit dello sfondo, quindi l'AND vale SEMPRE ZERO, e quindi il flag Zero assume valore UNO. Quindi il flag Zero ci può segnalare quando c'è una collisione e quando no.

Quando abbiamo a che fare con immagini con più bitplanes, le cose si complicano in quanto potrebbe accadere che si verifica una collisione tra 2 pixel di colori diversi che considerati plane per plane non coincidono. Per esempio, se si verifica una collisione tra un pixel di colore 1 (plane 1 = 1 e tutti gli altri a 0) e un pixel di colore 2 (plane 2 = 1 e tutti gli altri a 0) facendo un AND plane a plane, il risultato è sempre 0. In questi casi conviene usare i bitplane maschera. Essi infatti hanno un bit a 1 ogni volta che il corrispondente pixel del bob ha un colore diverso dallo sfondo. Quindi facendo l'AND tra 2 bitplane maschera si rilevano collisioni qualsiasi sia il colore dei pixel (è come rilevare la collisioni tra le “ombre” dei 2 bob, che sono immagini ad 1 plane).

Potete vedere un esempio in lezione10h1.s.

[illegible]

Fig. 31 lettera A deformata da un sine-scroller da 1 pixel

Come vedete ogni colonna di pixel si trova in una posizione verticale diversa dalle altre. Un sine-scroller da 2 pixel invece produce il seguente risultato:

[illegible]

Fig. 32 lettera A deformata da un sine-scroller da 2 pixel

Come vedete coppie di colonne adiacenti hanno la stessa posizione verticale. In un sine-scroller da 4 pixel, come avrete capito, le colonne di pixel son raggruppate a 4 a 4 e ogni gruppo assume una posizione diversa da un altro gruppo. Ora dovrete aver capito cosa si intende per sine-scroll da "1 pixel" o da "2 pixel".

Il metodo per realizzare un sine-scroller è molto semplice. Si parte da una normale routine di text scrolling, come quelle che abbiamo visto in precedenza. Però, invece di disegnare e scrollare il nostro testo sullo schermo visibile, lo facciamo in un buffer di dati allocato da qualche parte in memoria. Questo buffer di scroll non è mai visibile. Da questo buffer noi prendiamo delle "fettine" verticali di scroller e le copiamo nello schermo visibile. Ogni "fettina" viene copiata ad una differente posizione verticale, in base ad i valori della sinusoide. Lo spessore delle "fettine" determina la qualità del sine-scroller. Se esse sono spesse 1 pixel, abbiamo un sine scroller da 1 pixel, se sono spesse 2 pixel abbiamo una routine da 2 pixel e così via. Vediamo più in dettaglio come effettuare la copia delle "fettine".

Poiché le fettine sono molto sottili, faremo blittate larghe una sola word. Per selezionare all'interno della word solo la fettina (cioè solo le colonne di pixel) che ci interessano, useremo uno dei registri maschera del canale A (questo significa che siamo obbligati ad usare il canale A per la lettura) che ci permette di cancellare tutte le colonne di pixel che non fanno parte della fettina che ci interessa. Naturalmente, il valore della maschera varierà a seconda della "fettina" da leggere. La scrittura, come abbiamo già detto, avviene ogni volta ad una diversa posizione verticale. Quando effettuiamo la scrittura, non basta fare una semplice copia da A a D: se facessimo in questo modo, copiando una "fettina" cancelleremmo una parte delle "fettine" copiate in precedenza che appartengono alla stessa word della "fettina" attuale. Infatti, anche se le altre "fettine" non si sovrappongono alla nostra (perché si trovano una accanto all'altra) siccome la nostra blittata è larga una word, con una copia semplice copieremmo sullo schermo anche le colonne di pixel azzerate dalla maschera che si trovano a fianco della "fettina" attuale. Per risolvere questo problema, facciamo un OR tra la nostra word e lo sfondo sulla quale la scriviamo. In questo modo i pixel azzerati della word attuale non sovrascrivono quelli dello sfondo.

Per realizzare il sine-scroller è sufficiente copiare dal buffer allo schermo, mediante questo procedimento, tutto lo scrolltext una "fettina" alla volta. Ovviamente tutta la procedura deve essere ripetuta ad ogni fotogramma, perché lo scrolltext si è spostato e ogni volta, prima di effettuarla, è necessario cancellare lo schermo. Notate che maggiore è l'ampiezza della sinusoide e maggiore è l'area di schermo coinvolta nell'operazione, e che dobbiamo ogni volta cancellare. Quindi conviene usare una sinusoide poco ampia per migliorare le prestazioni.

In lezione10i1.s e lezione10i2.s troverete rispettivamente un sine-scroller da 2 pixel e uno da 1 pixel.

10.8 Animazione

Concludiamo la lezione con una breve spiegazione su come creare animazioni con il blitter. Un animazione è costituita da una serie di immagini (fotogrammi) che devono essere mostrati secondo una certa sequenza. Di solito tra un fotogramma e l'altro non varia tutta l'immagine, ma solo delle parti di essa. Per esempio potremmo avere un castello con delle bandiere che si muovono a causa del vento. Chiaramente solo la parte di schermo sulla quale sono disegnate le bandiere cambia tra un fotogramma e l'altro.

Per risparmiare memoria non conviene memorizzare tutte le immagini dell'animazione: basta memorizzare la prima immagine e poi i "pezzi" delle altre immagini che contengono le differenze con la prima. In questo modo per realizzare l'animazione basta copiare i nuovi "pezzi" di immagine sulla vecchia. Per questo scopo ci è molto utile il blitter che come sapete è molto più veloce

del 68000 (di base) nel copiare dati. In sostanza per realizzare un'animazione bisogna fare delle copie con il blitter, cosa di cui ormai siamo maestri. Le animazioni possono essere divise in due tipi a seconda di come è strutturata la sequenza di fotogrammi. Nelle animazioni del primo tipo, dette animazioni *cicliche*, i fotogrammi vengono disegnati uno dopo l'altro in base ad un ordine predeterminato. Dopo che è stato disegnato l'ultimo, l'animazione continua ripartendo dal primo fotogramma. Anche nelle animazioni del secondo tipo (animazioni *avanti-indietro*) i fotogrammi vengono disegnati in base ad un ordine. Però, dopo che è stato disegnato l'ultimo fotogramma, l'animazione continua ridisegnando i fotogrammi in ordine inverso, dal penultimo fino a tornare al primo. A questo punto l'animazione procede di nuovo in ordine diretto fino all'ultimo, poi ancora in ordine inverso e così via. In base al tipo di animazione si dovrà usare una diversa routine di gestione dei fotogrammi.

Vi presentiamo 2 esempi di animazione (uno per ogni tipo) nei listati `lezione10l1.s` e `lezione10l2.s`.

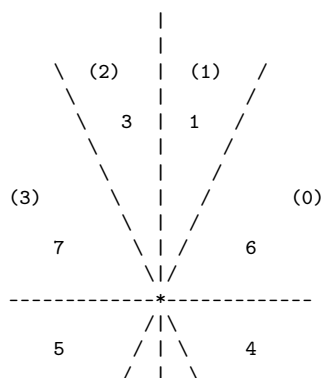
È possibile anche realizzare dei bob animati. Si tratta di bob che cambiano forma ogni volta che vengono disegnati. Naturalmente anche per i bob abbiamo a disposizione una serie di fotogrammi che vengono presentati in sequenza, in base ad una delle 2 tecniche di cui abbiamo parlato prima. Ogni volta che il bob deve essere disegnato bisogna utilizzare una figura diversa. È molto comodo dunque poter disporre di una routine universale, capace di disegnare come un bob qualsiasi figura, di dimensioni variabili.

Troverete una routine del genere per schermi in formato normale nell'esempio `lezione10m1.s` e per schermi in formato INTERLEAVED nell'esempio `lezione10m2.s`.

10.9 I modi speciali del Blitter

In aggiunta a tutte le funzioni descritte fin qui, il blitter ha anche la possibilità di disegnare linee e quella di "riempire" delle aree, cioè di settare a 1 tutti i bit di una determinata regione di un bit-plane. Queste capacità aggiuntive sono ottenute mediante degli speciali modi di funzionamento del blitter.

Iniziamo a parlare del tracciamento di linee. Quando il blitter opera in modo di tracciamento di linee (detto *line-mode*) esso disegna una linea da un punto dello schermo (che chiamiamo P1) a un altro (che chiamiamo P2). Indichiamo con X1 e Y1, rispettivamente l'ascissa e l'ordinata di P1, e con X2 e Y2 l'ascissa e l'ordinata di P2. In "line-mode" molti registri funzionano in maniera completamente differente rispetto a quanto visto finora ed è necessario settarli in maniera opportuna. Alcuni settaggi dipendono dalla posizione di P1 e P2. Prima di descrivere l'uso dei registri è necessario fare delle considerazioni preliminari. Durante il tracciamento il blitter considera lo schermo diviso in *ottanti* rispetto al punto P1. Per capire meglio guardate la seguente figura:



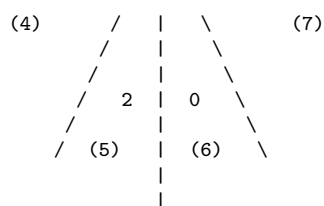


Fig. 1 Ottanti

Nella figura l'asterisco (*) rappresenta il punto P1. Il blitter considera lo schermo diviso nelle 8 regioni (dette ottanti) rappresentate in figura. La linea da tracciare appartiene ad uno degli ottanti, quello nel quale si trova P2. I numeri tra parentesi servono per numerare gli ottanti secondo la notazione di solito usata da noi "umani" (cioè in senso antiorario). Il blitter invece li numera in una maniera un pò strana che è indicata dai numeri senza parentesi. Di questa divisione dello schermo terremo conto in seguito. Dobbiamo inoltre definire alcune quantità che dovremo usare per preparare la blittata. Chiamiamo DiffX la differenza tra le ascisse di P2 e P1, cambiata di segno nel caso in cui venga negativa, in modo che sia comunque positiva. In formule poniamo:

$$\text{DiffX} = \text{abs}(X2 - X1)$$

dove "abs" indica la funzione che calcola il valore assoluto di un numero. Facciamo la stessa cosa con le ordinate ponendo:

$$\text{DiffY} = \text{abs}(Y2 - Y1).$$

A questo punto definiamo DX e DY rispettivamente come massimo e minimo tra DiffX e DiffY. In formule:

$$\begin{aligned} \text{DX} &= \max(\text{diffX}, \text{diffY}) \\ \text{DY} &= \min(\text{diffX}, \text{diffY}). \end{aligned}$$

Cominciamo ora a vedere come vanno settati i registri del blitter, a cominciare da BLTCON1 che permette di attivare il line-mode. Il bit 0 di BLTCON1 serve appunto a questo scopo. Quando è settato a 1 si attiva il line-mode. Il bit 1 permette di disegnare delle linee "speciali" che permettono il successivo riempimento di aree del blitter. Ne parleremo più avanti, per ora lo lasciamo a 0 (linee normali). Nei bit 2,3 e 4 va scritto il numero dell'ottante nel quale si trova il punto P2. Naturalmente dovremo usare la numerazione del blitter. Per convertire facilmente la normale numerazione in senso antiorario in quella usata dal blitter potete consultare la seguente tabella:

Valore Bit di BLTCON1	Numero Ottante
-----	-----
4 3 2	
- - -	
1 1 0	0
0 0 1	1
0 1 1	2
1 1 1	3
1 0 1	4
0 1 0	5
0 0 0	6
1 0 0	7

Il bit 6 di BLTCON1 (detto bit SIGN) va settato a 1 nel caso in cui risulti che $4*DY-2*DX < 0$. Altrimenti (cioè se $4*DY-2*DX > 0$) va settato a 0. I bit da 12 a 15 di BLTCON1 contengono la posizione di partenza del “pattern” della linea. Infatti è possibile disegnare non solo linee “solide”, ma anche linee tratteggiate, mediante un “pattern” che viene ripetuto lungo tutta la linea (abbiamo già visto esempi di pattern nella lezione 9). I bit da 12 a 15 di BLTCON1 indicano il pixel a partire dal quale deve essere usato il pattern. Naturalmente (abbiamo solo 4 bit) deve essere uno dei primi 16 pixel della linea. Tutti gli altri bit di BLTCON1 vanno lasciati a 0.

Veniamo ora a BLTCON0. Il byte basso di tale registro (LF, quello dei minterms) permette di selezionare 2 diverse modalità di disegno. Ponendo $LF = \$4A$ viene eseguita un'operazione di OR-esclusivo tra la linea e lo sfondo su cui viene tracciata. In pratica i pixel attraversati dalla linea vengono invertiti. Ponendo invece $LF = \$CA$ viene eseguita un'operazione di OR semplice tra la linea e lo sfondo. In pratica i pixel attraversati dalla linea vengono accesi. I canali da attivare per la blittata sono A, C e D. Quindi i bit 8, 9 e 11 devono essere settati a 1, mentre il 10 a 0. I bit da 12 a 15 di BLTCON0 devono invece contenere i 4 bit meno significativi (cioè più bassi) di X1, l'ascissa del punto P1. I settaggi degli altri registri sono fortunatamente più semplici. I registri BLTAFWM e BLTALWM devono essere settati al valore \$FFFF (non mascherano nulla). Il registro BLTADAT deve contenere invece il valore \$8000, che rappresenta il pixel da disegnare. Il registro BLTBDAT invece contiene il “pattern” della linea, a cui abbiamo accennato prima. Un valore \$FFFF fa disegnare una linea continua.

Nel tracciamento di linee viene usato solo la parte bassa di BLTAPT, ovvero solo il registro 16 bit BLTAPTL, che deve essere settato al valore $4*DY-2*DX$. Il registro BLTAMOD, invece va settato al valore $4*DY-4*DX$. Il registro BLTBMOD va settato al valore $4*DY$. I registri BLTCPT e BLTDPT devono contenere l'indirizzo della word dello schermo che contiene il pixel P1. I registri BLTCMOD e BLTDMOD devono contenere la larghezza dello schermo espressa in bytes.

Infine il registro BLTSIZE deve essere settato in maniera da eseguire una blittata larga 2 words e alta un numero di linee pari a $DX+1$. Il che vuol dire che i bit da 0 a 5 devono contenere il numero 2 mentre i bit da 6 a 15 il valore $DX+1$. Come accade di solito, scrivendo nel registro BLTSIZE si attiva il blitter. Per questo motivo, tale registro deve essere scritto per ultimo.

In sintesi, i valori da caricare nei registri sono:

```
BLTADAT = $8000
BLTBDAT = pattern linea ($FFFF per una linea solida)

BLTAFWM = $FFFF
BLTALWM = $FFFF

BLTAMOD = 4 * (dy - dx)
BLTBMOD = 4 * dy
BLTCMOD = larghezza del bitplane in byte
BLTDMOD = larghezza del bitplane in byte

BLTAPT = (4 * dy) - (2 * dx)
BLTBPT = non usato
BLTCPT = puntatore alla word che contiene il primo pixel della linea
BLTDPT = puntatore alla word che contiene il primo pixel della linea

BLTCON0 bit 15-12 = i 4 bit più bassi di X1
BLTCON0 bit 11 (SRCA), 9 (SRCC), e 8 (SRCD) = 1
BLTCON0 bit 10 (SRCB) = 0
BLTCON0 LF byte di controllo = $4A (per linea in EOR)
                                = $CA (per linea in OR)

BLTCON1 bit 0 = 1
BLTCON1 bit 4-2 = numero ottante (dalla tavola)
BLTCON1 bit 15-12 = bit iniziale per pattern linea
```

```

BLTCON1 bit 6 = 1 se  $(4 * dy) - (2 * dx) < 0$ 
                = 0 altrimenti
BLTCON1 bit 1 = 0 (per linee normali)
                = 1 (per linee speciali per il fill)

BLTSIZE bit 15-6 = dx + 1
BLTSIZE bit 5-0 = 2

```

Un esempio di tracciamento di linea è contenuto in `lezione10n.s`. Si tratta di una routine semplificata al massimo, senza ottimizzazioni particolari, per facilitare la comprensione a scapito della velocità di esecuzione.

10.10 Modo di Riempimento aree

Oltre a copiare dati, il blitter può simultaneamente eseguire un'operazione di fill (riempimento) durante la copia. Questo modo può essere attivato con una qualsiasi blittata standard (COPA, AND, OR, ecc.) e viene effettuato **dopo** tutte le altre operazioni che già conoscete (shift, mascheramenti, ecc.). Per capire come funziona il riempimento immaginate che il blitter scriva in uscita un bit alla volta (il che è falso, come sapete, perché scrive sempre UNA WORD alla volta) e che stia effettuando una semplice operazione di copia. Finché legge bit di valore 0, li copia normalmente. Ad un certo punto gli arriva un bit di valore 1. Lo copia ugualmente nell'uscita, ma a partire da questo momento, invece di continuare a copiare i bit seguenti, manda in uscita tutti bit di valore 1. Quando però legge un secondo bit di valore 1, riprende il comportamento normale. Quando poi legge un terzo bit di valore 1, ricomincia a mandare degli 1 in uscita, fino al successivo 1 in ingresso, e così via. Vediamo cosa succede ai dati copiati, mostrando una sequenza di bit in ingresso di esempio e la corrispondente uscita:

```

ingresso    000100010010010001000001000110010010
uscita      000111110011110001111111000110011110

```

In pratica i bit di valore 1 sono considerati i bordi dell'area e quindi il blitter riempie (cioè setta ad 1) i bit compresi dentro i bordi. Vediamo ora i dettagli tecnici del fill-mode. Come abbiamo già detto esso può essere usato in combinazione con una qualsiasi blittata, in quanto il riempimento viene effettuato dopo che i dati letti dalle 3 sorgenti sono stati combinati tra loro in base alla funzione logica selezionata dai minterms. Il fill-mode, però può essere usato solo con blittate effettuate in modo discendente. Vi sono 2 diversi tipi di fill, detti inclusivo ed esclusivo. Ogni tipo di fill ha un suo bit di abilitazione. Per attivare il fill-mode bisogna settare ad 1 uno dei 2 bit di abilitazione. Non è possibile attivare contemporaneamente i 2 diversi fill. Vediamo le differenze tra i 2 tipi di fill. Il modo di riempimento inclusivo riempie tra le linee, lasciandole intatte. Il modo esclusivo riempie tra le linee, ma pur lasciando la linea di delimitazione di destra, cancella quella di sinistra. Dunque il fill esclusivo produce forme riempite un pixel più strette dello stesso pattern (contorno) riempito con fill inclusivo.

Per esempio, il pattern:

```
00100100-00011000
```

riempito con fill inclusivo, produce:

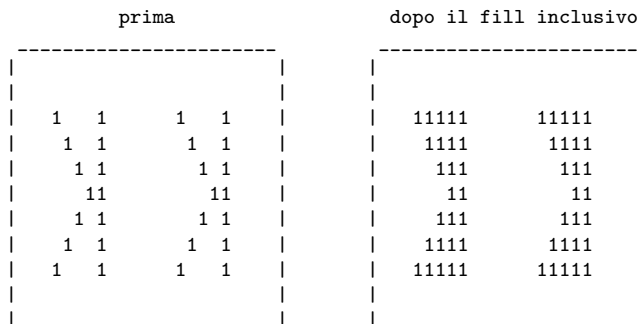
```
00111100-00011000
```

con fill esclusivo, il risultato sarebbe:

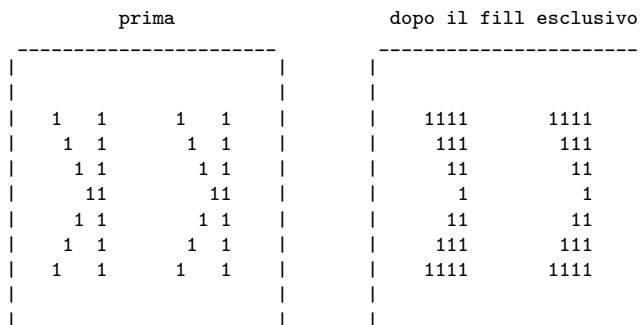
```
00011100-00001000
```

(Naturalmente, i riempimenti sono sempre fatti su piene word di 16-bit.) Facciamo un altro esempio con l'aiuto di disegni:

fill inclusivo:



fill esclusivo:



come potete vedere, con il fill esclusivo sono state cancellate le linee sinistre della figura. In questo modo si ottengono figure con bordi più affilati. Il bit di abilitazione del fill inclusivo è il bit 3 di BLTCON1, mentre quello del fill esclusivo è il bit 4, sempre di BLTCON1.

C'è un altro bit che serve a controllare il riempimento. Si tratta del bit 2 di BLTCON1 (detto FILL_CARRYIN) che, quando viene settato ad 1, forza il riempimento delle zone esterne alle linee, anziché di quelle interne. Torniamo al primo esempio che abbiamo fatto e vediamo cosa succede alla nostra riga di bit quando il bit FILL_CARRYIN è settato a 1. La riga di partenza era:

```
00100100-00011000
```

Con fill inclusivo e FILL_CARRYIN=1, l'output sarebbe:

```
11100111-11111111
```

Con fill esclusivo e FILL_CARRYIN=1, l'output sarebbe:

```
11100011-11110111
```

Vediamo cosa succede nel caso del secondo esempio con fill inclusivo e FILL_CARRYIN=1.

prima	dopo
<pre> 1 1 1 1 1 1 1 1 1 1 1 1 11 11 1 1 1 1 1 1 1 1 1 1 1 1 </pre>	<pre> 111 1111111 11 1111 11111111 11 11111 111111111 11 1111111111111111 11111 111111111 11 1111 11111111 11 111 1111111 11 </pre>

fill inclusivo e bit FCI = 1

Il fill-mode viene usato soprattutto per il riempimento di poligoni. I bordi dei poligoni vengono tracciati usando il line-mode del blitter. Un primo esempio molto semplice è presentato nel listato `lezione10o.s`, nel quale sono illustrati i vari tipi di fill. Quando l'area da riempire è delimitata da linee aventi pendenza minore di 45 gradi, sorge un problema. In questo caso, infatti, accade che una linea è formata da pixel che possono trovarsi adiacenti sulla stessa riga orizzontale dello schermo. La situazione è mostrata dalla seguente figura nella quale gli asterischi (*) rappresentano dei pixel di valore 1.

```

      *
      *
      *      linea con pendenza > di 45 gradi
      *
      *

      *
      **
      **      linea con pendenza < di 45 gradi
      *
      *
      **

```

Come vedete quando una linea ha pendenza maggiore di 45 gradi non capita mai che 2 dei suoi pixel siano affiancati sulla stessa riga dello schermo. Al contrario ciò accade quando la pendenza della linea è minore di 45 gradi. Questo fatto crea il problema nel riempimento. Infatti quando il blitter incontra 2 pixel affiancati sulla stessa riga durante il riempimento, li considera come 2 bordi distinti, e quindi non riempie i pixel che si trovano a destra della linea. trovate un esempio di questo problema nel listato `lezione10p.s`. Per ovviare a questo problema, i progettisti del blitter ci hanno messo a disposizione una speciale modalità di tracciamento linee (a cui avevamo accennato in precedenza) che produce linee aventi un solo pixel per ogni riga orizzontale. Chiaramente se tracciate una linea in questa modalità senza poi fare il fill, essa vi apparirà “spezzettata”. Nel listato `lezione10q.s` trovate la soluzione al problema mostrato in `lezione10p.s`. Nell'esempio `lezione10r.s` proviamo a tracciare e riempire un poligono chiuso formato da molte linee. Notiamo che si riscontra anche qui in piccolo problema. Il problema nasce dal fatto che i vertici del poligono sono in comune ad una coppia di linee. Quando disegniamo delle linee in modalità EOR invertiamo i pixel dello sfondo. I vertici vengono invertiti 2 volte e quindi alla fine risultano azzerati. Quindi c'è un “buco” nel bordo del poligono, a causa del quale il riempimento viene effettuato male. Se invece disegniamo le linee in modalità OR, i vertici rimangono al valore 1. Ciò crea problemi con i vertici in alto e in basso, in quanto essi si trovano isolati sulla riga a cui appartengono e pertanto il riempimento inizia a partire da essi ma non termina mai. Per capire meglio osservate la seguente figura (riferita al vertice basso):

```

*      *
*      *
*      *
*      *
*
~
+---- vertice in basso

*****
*****
*****
*****
*****
*****
~
+---- vertice in basso

```

Prima del FILL

Dopo del FILL

Come vedete sulla riga su cui giace l'ultimo vertice il riempimento non termina perché non c'è un altro pixel settato a 1 che faccia da bordo sinistro. Nel caso delle linee in modalità EOR questo problema non si presenta perché il vertice viene azzerato (cioè per il fenomeno che ci crea problemi per i vertici intermedi). Insomma in qualsiasi modo facciamo c'è sempre un vertice che ci fa sballare il fill!

Vediamo come trarci d'impaccio. Conviene disegnare le linee in modalità EOR, in modo da eliminare il problema dei vertici alto e basso. Inoltre facciamo in modo di disegnare le linee sempre dall'alto verso il basso e, prima di disegnarle invertiamo (con una BCHG) il primo pixel. In questo modo tale pixel sarà invertito 2 volte (dalla BCHG e poi dalla blittata) e risulterà dunque immutato. In questo modo il problema è risolto. Infatti (siccome abbiamo ordinato i punti) ogni vertice intermedio è disegnato una volta come ultimo pixel di una linea (e pertanto viene settato a 1) e una volta come primo pixel dell'altra linea (e pertanto rimane immutato, quindi a 1). Questa tecnica è presentata nell'esempio `lezione10s.s`.

Torniamo ora ad occuparci del trattamento di linee, per illustrare una particolarità. È possibile tracciare linee larghe 2 pixel semplicemente cambiando il valore di inizializzazione di `BLTBDAT`. La tecnica è illustrata nell'esempio `lezione10t1.s`. Nell'esempio `lezione10t2.s`, invece viene presentata una routine di tracciamento linee migliore di quella usata finora. Questa routine infatti sfrutta molti particolarità dell'assembler 68000 per ottimizzare il calcolo e il caricamento dei registri del blitter.

Per concludere la lezione presentiamo alcuni effetti realizzati mediante tracciamento di linee e fill, nei listati `lezione10u1.s`, `lezione10u2.s`, `lezione10v.s`, `lezione10x.s`. In particolare nell'ultimo vedrete una delle tecniche principali della leggendaria demo *"State of the Art"*!!

LEZIONE 11 - INTERRUPT, CIAA/CIAB, DOSLib

(Directory Sorgenti7) - quindi scrivere `<V Assembler3:sorgenti7>`

Ora che avete approfondito il funzionamento del blitter, potete affermare con sicurezza di conoscere l'hardware di Amiga, dato che 68000, blitter e copper li sapete programmare. Però non si finisce mai di imparare, e in questa lezione vedremo delle informazioni avanzate sul 68000, come gli interrupt, nonché dei listati che mostrano utilizzi particolari di Blitter e Copper, e infine l'utilizzo dei chip CIAA e CIAB, che per ora abbiamo usato solo per testare la pressione del mouse. Direi di cominciare con le nuove informazioni sul 68000, in modo da fare una startup migliore di quella che abbiamo fatto nel corso della LEZIONE8. Le informazioni su vettori di eccezione, interrupt ecc. che saranno spiegate non saranno tutte utili e indispensabili per la programmazione di giochi e demo, ma solo una parte di esse ci servirà. Non lasciatevi dunque intimorire dalla quantità di cose accennate: in pratica è poco ciò che useremo! Innanzitutto occorre parlare dei due modi operativi del 680x0, ossia del modo utente e del modo supervisore. Già nel 68000-2.txt si accennava, senza spiegare, che ci sono delle istruzioni "privilegiate", che devono essere eseguite in modo supervisore, ossia durante una eccezione o un interrupt. Per ora abbiamo fatto eseguire le nostre routines sempre in modo USER, ossia nel modo utente, perché non era necessario eseguire istruzioni privilegiate, e perché non abbiamo sfruttato gli Interrupt del processore.

Devo premettere che la velocità di esecuzione non cambia tra modo utente e modo supervisore, per cui far eseguire il proprio programma come una exception o un interrupt non turbizzerà di certo l'esecuzione. D'altronde una "eccezione" è chiamata così proprio perché si tratta di una cosa che deve avvenire solo in casi "eccezionali", per esempio un software failure, noto come GURU. Da notare che ci sono dei guru che sono causati dal sistema operativo Amiga, (come errori di exec) e altri che sono programmati direttamente dalla Motorola, autrice dei 680x0. Per esempio, gli errori di bus, di divisione per zero, ecc, ossia i vettori, sono di questo tipo. Avrete notato che il sistema operativo riesce a far aggiornare la posizione della freccia puntatore del mouse anche se stiamo eseguendo una nostra routine, a patto che non si disabiliti il multitasking con una chiamata al `Disable()`. Ebbene, se il processore sta eseguendo un nostro loop, come fa ogni fotogramma ad aggiornare altre cose? Ricorderete che se si disabilitano gli interrupt di

sistema si blocca tutto. Infatti si serve degli interrupt, che “interrompono” l'esecuzione del nostro programmino ogni fotogramma, eseguono una loro routine, e riprendono l'esecuzione del nostro programma dove l'avevano lasciata, il tutto senza che ce ne accorgiamo! Oltre agli interrupt, ci sono altri modi per eseguire delle routines in supervisore, ad esempio le istruzioni TRAP, o un qualsiasi errore in un programma. Ad esempio quando avviene una divisione per zero, o il processore trova dei dati che non corrispondono a nessuna istruzione, vengono eseguite in modo supervisore le routines dei guru meditation e software failure. Gli emulatori di altri processori, per esempio, fanno in modo che ad ogni valore binario di istruzione del processore emulato, ad esempio un 80286 o il 6502 del commodore 64, corrisponda una routine che faccia le operazioni che avrebbe fatto quell'istruzione per quel processore (grossomodo!). Ora, non è negli scopi di questo corso imparare a programmare sistemi operativi o emulatori, dato che il sistema operativo dell'Amiga è già il migliore al mondo, e gli emulatori su Amiga sono insuperabili, tanto che chi ha un Amiga può far eseguire i programmi di un MacIntosh e di un MSDOS, e giocare ai giochi dei vecchi C64 e Spectrum. Ma anche se si vuole programmare un demo o un gioco, o una utility come il protracker/octamed, è **utilissimo** gestire qualche interrupt.

Molte demo e giochi di vecchio tipo cominciavano mandando in modo supervisore il 68000, scrivendo subito sullo SR e facendo giochi strani con lo Stack. Ebbene, molti di questi giochi non funzionano su computer con 68020, perché lo Status Register nei processori più evoluti ha delle funzioni in più, che tali programmatori ignoravano, e settando o azzerando bit alla rinfusa commisero un enorme sbaglio. Inoltre in modo supervisore lo stack (SP) è uno stack “privato” del modo supervisore, mentre si accede allo stack di utente con il registro USP (User Stack Pointer). Insomma in modo supervisore è meglio andarci coi piedi di piombo, a meno che non si conoscano perfettamente tutte le nuove caratteristiche del 68020/30/40 e anche del 68060! Infatti il modo utente serve proprio a evitare di eseguire istruzioni che potrebbero provocare effetti diversi su processori diversi, che solo i sistemi operativi devono eseguire. Però i coder si sono sempre sentiti più tosti a mettere sottosopra i registri in modo supervisore, col risultato di farsi la fama di “programmatori incapaci di codice non compatibile”.

Ma vediamo di eseguire qualche istruzione in modo supervisore, prima di continuare con la teoria. Tra i tanti modi che ci sono per far eseguire una routine in exception, il più “sicuro” è quello di usare l'apposita funzione del sistema operativo `exec.library/Supervisor`, che richiede in entrata di mettere in A5 l'indirizzo della routine da eseguire:

```

1      move.l 4.w,a6          ; ExecBase in a6
2      lea   SuperCode(PC),a5 ; Routine da eseguire in supervisor
3      jsr   -$1e(a6)         ; LvoSupervisor - esegui la routine
4                               ; (non salva i registri! attenzione!)
5      rts                    ; esci, dopo aver eseguito la routine
6                               ; "SuperCode" in supervisor.
7
8 SuperCode:
9      movem.l d0-d7/a0-a6,-(SP) ; Salva i registri nello stack
10     ...                       ; istruzioni da eseguire
11     ...                       ; come fosse una subroutine....
12     movem.l (SP),d0-d7/a0-a6 ; Riprende i registri dallo stack
13     RTE ; Return From Exception: come l'RTS, ma per le eccezioni.
```

Come si può notare, non c'è niente di più facile. La routine da eseguire la potete considerare come una subroutine da chiamare con JSR o BSR, solo che si chiama con JSR `-$1e(a6)` dopo aver messo il suo indirizzo in A5, e naturalmente l'ExecBase in A6. Al termine della breve “subroutine supervisor”, anziché mettere un RTS per ritornare, occorrerà mettere un RTE, apposito per tornare da eccezioni e interrupt. A questo punto il processore tornerà ad eseguire in modo utente l'istruzione sotto il JSR `-$1e(a6)`, proprio come fosse stato un BSR o un JSR. La funzione Supervisor non salva i registri né li ripristina col MOVEM, per cui se se ne modifica qualcuno durante la routine in supervisor tali valori rimarranno al ritorno da quella routine. A questo proposito vi consiglio di salvare e ripristinare i registri manualmente all'inizio e alla fine della

routine supervisor. Da notare che accedendo a SP o A7 in modo supervisor si salva nello stack di supervisor, e non in quello utente, richiamabile come USP. Questo non è un problema, perché lo stack di supervisor salva e ripristina come quello utente, ma in caso di programmazione pericolosa dello stack potreste prendere dei granchi con impiantamento generale. Eseguiamo delle istruzioni privilegiate nell'esempio lezione11a.s. Ecco le istruzioni privilegiate eseguibili solo in modo supervisor:

```

1      ANDI.W  #xxxx,SR
2      ORI.W   #xxxx,SR
3      EORI.W  #xxxx,SR
4      MOVE.W  xxxx,SR
5      MOVE.W  SR,xxxxx
6      MOVEC   registro,registro      ; 68010+ - registri speciali per
7                                      ; controllo cache, MMU, vettori come:
8                                      ; CAAR, CACR, DFC, ISP, MSP, SFC, USP, VBR.
9      RTE

```

Ci sarebbero anche MOVES, RESET, STOP, ma non ci interessano. Interessa poco la possibilità di agire sullo Status Register, perché è molto pericoloso (data la differenza dei bit dello SR tra 68000 e altri 680x0), e non è indispensabile disturbarlo. Tra l'altro quando si salta ad una eccezione viene salvato nello stack, oltre al Program Counter, anche lo Status Register, e al momento dell'RTE viene ripristinato il vecchio valore del Program Counter, per tornare sotto il JSR -\$1e(a6), e il vecchio SR, cosicché non ci sono cambiamenti, se non durante l'esecuzione dell'exception. Invece l'istruzione MOVEC ci interesserà molto di più, perché per poter usare un interrupt in modo che funzioni anche su processori 68020+ è necessario sapere dove si trova il VBR (Vector Base Register). Per quanto riguarda il controllo delle CACHE, credo sia meglio non interferire, in modo che l'utente possa attivarle o disattivarle con delle utility prima di eseguire il nostro demo o gioco, in modo da valutare le differenze di settaggio. Se invece decidiamo per lui quali cache devono essere attivate e quali disattivate, rischiamo anche di fare codice che non funziona sul 68060 e eventuali nuovi processori RISC che potrebbero emularlo. L'istruzione MOVEC, disponibile solo dal 68010 in avanti, serve per copiare il contenuto di un registro An o Dx in un registro speciale, o viceversa. Vediamo i registri speciali presenti nel 68020:

```

CAAR   - CAche Address Register
CACR   - CAche Control Register
VBR    - Vector Base register

```

Ci sono anche DFC (Destination Function Code), SFC (Source Function Code), ISP (Interrupt Stack Pointer), MSP (Master Stack Pointer), USP (User Stack Pointer), ma non ci interessano, perché servono solo a chi programma sistemi operativi, emulatori eccetera. A noi in questo momento serve sapere il valore del registro speciale VBR, vedremo poi il perché. Per ottenerlo basta un MOVEC VBR,d0, ad esempio. Ma cosa è il Vector Base Register? Innanzitutto occorre spiegare cosa è un vettore (da non confondere con i vettori in matematica, o i vectors 3d!). Prima vediamo la tabella dei vettori, poi spieghiamoli:

NUM. VETTORE	OFFSET	Assegnazione e significato
0	\$0	Serve solo al momento del reset (SSP start)
1	\$4	Serve solo al reset, infatti c'è ExecBase (PC start)
2	\$8	GURU/soft. failure: Errore di BUS
3	\$c	GURU/soft. failure: Errore di Indirizzo
4	\$10	GURU/soft. failure: Istruzione illegale
5	\$14	GURU/soft. failure: Divisione per zero
6	\$18	Eccezioni generate da istruzioni CHK,CHK2 (68020+)
7	\$1c	Eccezioni gen. da istruzioni TRAPV (68020+ TRAPCC)
8	\$20	GURU/soft. failure: Violazione di privilegio
9	\$24	Traccia (trace exception)

\$A	\$28	GURU/soft. failure: Emulatore di linea %1010 (LINE-A)
\$B	\$2c	GURU/soft. failure: Emulatore di linea %1111 (LINE-F)
\$C	\$30	non usato
\$D	\$34	Violazione di protocollo coprocessore (68020+)
\$E	\$38	Errore di formato (solo 68020, dopo CALLM,RTM)
\$F	\$3c	Interruzione non inizializzata
...	...	
\$18	\$60	Interruzione spuria

; Ecco gli autovettori di interruzione: sono questi quelli che ci interessano!

\$19	\$64	INTERRUPT livello 1 (softint,dskblk,tbe)
\$1a	\$68	INTERRUPT livello 2 (ports: I/O,ciaa,int2)
\$1b	\$6c	INTERRUPT livello 3 (coper,vblanc,blit)
\$1c	\$70	INTERRUPT livello 4 (canali audio aud0/aud1/aud2/aud3)
\$1d	\$74	INTERRUPT livello 5 (rbf,dsksync)
\$1e	\$78	INTERRUPT livello 6 (exter: ciab,int6 + inten)
\$1f	\$7c	INTERRUPT livello 7 (schede hardware esterne: NMI)
...		
\$20	\$80	Vettore richiamabile con TRAP #0
\$21	\$84	Vettore richiamabile con TRAP #1
\$22	\$88	Vettore richiamabile con TRAP #2
\$23	\$8c	Vettore richiamabile con TRAP #3
\$24	\$90	Vettore richiamabile con TRAP #4
\$25	\$84	Vettore di TRAP #5, eccetera, fino TRAP #15
...		

seguono vettori per errori dell'eventuale coprocessore matematico e dell'MMU, che non ci interessano.

Immaginiamo la situazione di un processore 68000, per il quale non occorre cercare il valore del registro VBR, (non esiste nemmeno!). In questo caso l'offset è proprio la locazione di memoria! \$0 = \$00000000 All'indirizzo \$4 troviamo l'Execbase, mentre la prima long a \$0 di solito è azzerata. Ma ecco che "dentro" la locazione \$8 troviamo l'indirizzo della routine che fa apparire la scritta *GURU MEDITATION/SOFTWARE FAILURE* in caso di *bus error*. Infatti tale guru, come visto nel 68000-2.TXT, ha come numero di identificazione #00000002, ossia il suo numero di vettore.

Il terzo vettore contiene l'indirizzo della routine che fa apparire il guru di ADDRESS ERROR (#00000003), e così via. In pratica, quando il processore trova uno di questi errori, salta al vettore corrispondente, che contiene l'indirizzo della routine da eseguire in modo supervisore (data la gravità della situazione!). Al momento del reset vengono scritti dalla ROM tutti gli indirizzi nei vettori dal primo all'ultimo. Se avete dei programmini che modificano i messaggi dei software failure o dei guru, sappiate che fanno "puntare" i vettori alle sue routines, anziché alle routines normali di sistema. Naturalmente ci sono modi "legali" per cambiare i vettori, ossia passando da strutture e routines del sistema operativo. Scrivere brutalmente l'indirizzo della propria routine nel vettore può risultare inefficace o incompatibile. Ad esempio:

```

1      MOVE.L  #MiaDivisionePerZero,$14.w      ; sostituisco il vettore
2                                              ; del guru da div. per zero.
3      rts
4
5  MiaDivisionePerZero:
6      ...
7      RTE

```

NON FATE MAI COME IN QUESTO ESEMPIO, PER UN PAIO DI RAGIONI. LA PRIMA è CHE NEL 68020+ NON È DETTO CHE TALE VETTORE SI TROVI ALL'INDIRIZZO \$14, LA SECONDA è CHE È UN METODO INCOMPATIBILE CON MMU E STRUTTURE DEL SISTEMA OPERATIVO

AMIGA. Comunque, in teoria questo sistema dovrebbe funzionare, e su Amiga500 funziona quasi sempre, a patto che la routine supervisor sia scritta bene. La modifica dei vettori di errore/guru però ci interessa minimamente, perché il nostro programma non dovrebbe contenere errori da correggere al momento del salto al guru/software failure! Anche i vettori delle istruzioni TRAP #xx ci interessano poco. Tali vettori erano usati in passato per andare in eccezione, ma abbiamo già visto un modo più sicuro, quello tramite sistema operativo. Comunque, per vostra curiosità, il modo “antico” era:

```

1      move.l  $80.w,OldVector      ; Salva il vecchio vettore TRAP #0
2      move.l  #SuperCode,$80.w    ; Routine da eseguire in supervisor
3                                     ; posta nel vettore TRAP #0
4      TRAP    #0                    ; Esegui Supercode come eccezione
5      move.l  OldVector(PC),$80.w  ; ripristina il vecchio vettore TRAP #0
6      rts                          ; esci, dopo aver eseguito la routine
7                                     ; "SuperCode" in supervisor.
8
9 OldVector:
10     dc.l    0
11
12 SuperCode:
13     movem.l  d0-d7/a0-a6,-(SP)    ; Salva i registri nello stack
14     ...                                     ; istruzioni da eseguire
15     ...                                     ; come fosse una subroutine...
16     movem.l  (SP),d0-d7/a0-a6     ; Riprende i registri dallo stack
17     RTE      ; Return From Exception: come l'RTS, ma per le eccezioni.

```

Come vedete si può facilmente capire la funzione dell'istruzione TRAP: in pratica se si esegue un TRAP #0 viene eseguita come eccezione la routine il cui indirizzo è contenuto nella .l all'indirizzo \$80, mentre con l'istruzione TRAP #1 si esegue quella in \$84, e così via. Allo stesso modo, gli interrupt contengono l'indirizzo della routine da eseguire in caso di interrupt. Il modo antico per settare un interrupt era:

```

1      move.l  $6c.w,OldInt6c      ; Salva il vecchio int livello 3
2      move.l  #MioInt6c,$6c.w    ; Mia routine per int livello 3

```

Alla fine del programma veniva ripristinato il vecchio interrupt in \$6c. In questo interrupt di solito viene messo il BSR.w MT_MUSIC, dato che tale interrupt (VERTB) viene eseguito una volta per fotogramma.

11.1 Il registro VBR nei processori 68010 e superiori

Vi starete domandando cosa c'entra il VBR con i vettori. Ebbene, il Vector Base Register è l'indirizzo di BASE a cui aggiungere gli offset per trovare l'indirizzo dei vettori. Se il VBR=0, allora l'interrupt livello 3 si troverà all'indirizzo \$6c, come nel 68000, e allo stesso modo il TRAP #0 si troverà sempre in \$80, e gli esempi visti sopra funzionerebbero. Ma se il VBR fosse a \$10000, l'interrupt di livello 3 si troverebbe non più a \$6c, ma a VBR+\$6c, ossia a \$1006c! Lo stesso per tutti gli altri vettori. Dunque, in linea di massima:

$$\text{indirizzo del vettore} = \text{VBR} + \text{OFFSET}$$

Sul processore 68000 la base è sempre \$0000, tanto che non esiste il registro VBR né l'istruzione privilegiata MOVEC. Ma su 68010 e superiori il VBR può essere spostato in altri luoghi, anche in FAST RAM. Dopo un reset, il VBR è comunque sempre azzerato, sia su A3000 che su A1200 o A4000. È eseguendo il SETPATCH o altre utility che il VBR viene spostato. Infatti, molte demo/giochi su file che funzionano se fatte partire da sole da un disco senza caricare prima il setpatch, non funzionano se caricate dallo shell del workbench. Alcune funzionano ma sono “mute”, proprio perché scrivono il loro interrupt, che suona solo la musica, in \$6c, quando invece il VBR punta più avanti di \$0000. Dato che sappiamo questo, basta controllare se il processore è un 68000 o

un 68010+, e nel caso sia un 68010 o superiore prelevare il valore del VBR e sommarlo al vettore che si vuole cambiare. Ecco come fare in pratica:

```

1      move.l 4.w,a6      ; ExecBase in a6
2      btst.b #0,$129(a6) ; Testa se siamo su un 68010 o superiore
3      beq.s  IntOK      ; è un 68000! Allora la base è sempre zero.
4      lea    SuperCode(PC),a5 ; Routine da eseguire in supervisor
5      jsr    -$1e(a6)      ; LvoSupervisor - esegui la routine
6                      ; (non salva i registri! attenzione!)
7      bra.s  IntOK      ; Abbiamo il valore del VBR, continuiamo...
8
9      ;*****CODICE IN SUPERVISORE per 68010+ *****
10     SuperCode:
11         movem.l a0-a1,-(SP) ; Salvo a0 ed a1 nello stack
12         dc.l    $4e7a9801   ; Movec Vbr,A1 (istruzione 68010+).
13                      ; è in esadecimale perchè non tutti gli
14                      ; assembleri assemblano il movec.
15         lea    BaseVBR(PC),a0 ; Label dove salvare il valore del VBR
16         move.l a1,(a0)      ; Salvo il valore.
17         movem.l (SP)+,a0-a1 ; Ripristino i vecchi valori di a0 ed a1
18         RTE      ; Ritorna dalla eccezione
19     ;*****
20
21     BaseVBR:
22         dc.l    0
23
24     IntOK:
25         move.l BaseVBR(PC),a0 ; In a0 il valore del VBR
26         move.l $64(a0),OldInt64 ; Sys int liv 1 salvato (softint,dskblk)
27         move.l $68(a0),OldInt68 ; Sys int liv 2 salvato (I/O,ciaa,int2)
28         move.l $6c(a0),OldInt6c ; Sys int liv 3 salvato (coper,vblanc,blit)
29         move.l $70(a0),OldInt70 ; Sys int liv 4 salvato (audio)
30         move.l $74(a0),OldInt74 ; Sys int liv 5 salvato (rbf,dksync)
31         move.l $78(a0),OldInt78 ; Sys int liv 6 salvato (exter,ciab,inten)
32
33         movem.l d0-d7/a0-a6,-(Sp) ; Salva i registri nello stack
34         bsr.s  START      ; Esegui la routine principale
35         movem.l (sp)+,d0-d7/a0-a6 ; Riprendi i registri dallo stack
36
37         move.l BaseVBR(PC),a0 ; In a0 il valore del VBR
38         move.l OldInt64(PC),$64(a0) ; Sys int liv1 salvato (softint,dskblk)
39         move.l OldInt68(PC),$68(a0) ; Sys int liv2 salvato (I/O,ciaa,int2)
40         move.l OldInt6c(PC),$6c(a0) ; Sys int liv3 salvato (coper,vblanc,blit)
41         move.l OldInt70(PC),$70(a0) ; Sys int liv4 salvato (audio)
42         move.l OldInt74(PC),$74(a0) ; Sys int liv5 salvato (rbf,dksync)
43         move.l OldInt78(PC),$78(a0) ; Sys int liv6 salvato (exter,ciab,inten)
44         rts
45
46     START:
47         move.l BaseVBR(PC),a0 ; In a0 il valore del VBR
48         move.l #MioInt6c,$6c(a0) ; metto la mia rout. int. livello 3.
49         ...
50         eseguo il programma
51         ...
52         rts

```

Da notare che anche se si volesse usare l'istruzione TRAP occorrerebbe mettere in a0 il BaseVbr e fare l'offset \$80(a0). Lo stesso dicasi per tutti i vettori. è presente da questa lezione in avanti una startup nuova, startup2.s, da includere al posto della startup1.s. L'unica differenza è che contiene le istruzioni viste sopra ed è disponibile la label BaseVbr per modificare gli interrupt propriamente su tutti i microprocessori. Il salvataggio dei vecchi interrupt ed il ripristino alla fine sono fatti dalla startup, assieme al salvataggio e al ripristino dei canali DMA e di INTENA. Un'altra modifica è l'aggiunta di una routine che blocca l'input del mouse e della tastiera al sistema operativo, vedremo che caricando i file servirà. Ora che sappiamo come sostituire un interrupt di sistema con uno nostro, dobbiamo vedere come fare il nostro interrupt. Per anticipare le pagine seguenti, suoniamo la musica in interrupt nel listato esempio Lezione11b.s. Capirete meglio come funziona continuando a leggere!

11.2 Come si costruisce una routine di interrupt

Il sistema delle interruzioni (interrupt) consente ad un dispositivo esterno o ad un chip custom di interrompere l'esecuzione del processore, per farlo saltare in modo user alla routine il cui indirizzo si deve trovare in uno degli autovettori di interruzione (ad esempio \$6c). Questi interrupt hanno dei livelli diversi di *priorità*, che vanno da un livello minimo (1) al livello massimo (7). Queste priorità servono nel caso che si presenti l'occorrenza di una o più interruzioni durante l'esecuzione di un interrupt. Se per esempio il programma normale in modo user viene interrotto da un interrupt di livello basso, diciamo 2, e mentre viene eseguita questa routine si presenta la richiesta di un interrupt di livello più alto, ad esempio 5, l'interrupt 2 viene interrotto a sua volta dall'interrupt 5, con maggiore priorità, e una volta terminato quest'ultimo interrupt il controllo torna all'interrupt di livello 2, il quale poi lo ripassa al programma normale in modo user. In questo modo più routines interrupt possono rimanere in attesa di terminare l'esecuzione, e a seconda del loro livello saranno finite di eseguire prima. La necessità di introdurre gli interrupt fin dai primi microprocessori è legata al fatto che spesso si utilizza male la potenza della CPU, a causa di loop di attesa molto lunghi. Se per esempio si dovesse attendere l'inizio del vertical blank, si dovrebbe fare un loop che controlla la linea raggiunta, e fino a che tale linea non è raggiunta il processore non fa altro che bloccarsi in quel ridicolo loop. Se l'attesa per un certo segnale fosse di qualche secondo, immaginatevi come sarebbe sprecata la potenza del processore! Per questo esistono gli interrupt, che permettono al processore di eseguire dei programmi senza preoccuparsi di attendere gli "eventi". Se si genera un interrupt di livello 3 ogni vertical blank, si potrà far calcolare al processore un frattale o una immagine 3d, e quando si presenterà l'interrupt, al momento del vertical blanking, sarà interrotto il calcolo del solido 3d, verrà eseguita la routine da eseguire all'inizio del Vblank, poi si tornerà a continuare la routine 3d dove si era lasciata. Il multitasking stesso è possibile grazie a questo: poter stampare o leggere dal disk drive mentre si fa qualcos'altro è possibile perché, a differenza del PC MSDOS, il processore può svolgere un compito che sarà interrotto al momento giusto dall'interrupt del disco o della porta seriale/parallela, e che sarà ripreso appena eseguiti tali interrupt. L'Amiga assegna 6 dei 7 livelli di interrupt disponibili, a segnali prodotti dai chip custom (blitter, copper, cia) in determinate situazioni. Il settimo livello è usato da schede esterne come la Action Replay, dato che le linee IPL2-IPL0 che lo generano sono portate alla porta di espansione. I primi 6 livelli di interrupt sono generati dai chip custom, ad esempio in occasione del completamento di una blittata o di un vertical blank video.

11.3 Come si usano INTENA ed INTENAR

È possibile, tramite il registro INTENA (\$dff09a), mascherare alcuni di questi interrupt, ossia evitare che siano generati. Esiste anche un registro per richiedere interrupt, l'INTREQ (\$dff09c). Questi registri funzionano come il DMACON (\$dff09a), infatti il bit 15 decide se gli altri bit specificati devono essere settati o azzerati. Come abbiamo fatto per il DMACON/DMACONR nella lezione 8, vediamo la "mappa" dei registri INTENA(\$dff09a) a sola scrittura e INTENAR(\$dff01c) a sola lettura:

INTENA/INTENAR (\$dff09a/\$dff01c)

BIT	NOME	LIV.	DESCRIZIONE
15	SET/CLR		Bit di controllo "Set/clear". Determina se i bit ad 1 devono essere azzerati o settati, come in DMACON. I bit=0 non saranno nè settati nè azzerati
14	INTEN		Master interrupt (interrutt. generale di abilitazione)

13	EXTER	6 (\$78)	Interrupt esterno, connesso alla linea INT6
12	DSKSYN	5 (\$74)	Generato se il registro DSKSYNC corrisponde ai dati letti dal disco nel drive. Serve per i loader hardware.
11	RBF	5 (\$74)	Buffer UART di ricezione della porta seriale PIENO.
10	AUD3	4 (\$70)	Lettura di un blocco di dati del can. audio 3 finita.
09	AUD2	4 (\$70)	Lettura di un blocco di dati del can. audio 2 finita.
08	AUD1	4 (\$70)	Lettura di un blocco di dati del can. audio 1 finita.
07	AUD0	4 (\$70)	Lettura di un blocco di dati del can. audio 0 finita.
06	BLIT	3 (\$6c)	Se il blitter ha finito una blittata si setta ad 1
05	VERTB	3 (\$6c)	Generato ogni volta che il pennello elettronico è alla linea 00, ossia ad ogni inizio di vertical blank.
04	COPER	3 (\$6c)	Si può settare col copper per generarlo ad una certa linea video. Basta richiederlo dopo un certo WAIT.
03	PORTS	2 (\$68)	Input/Output Porte e timers, connesso alla linea INT2
02	SOFT	1 (\$64)	Riservato agli interrupt inizializzati via software.
01	DSKBLK	1 (\$64)	Fine del trasferimento di un blocco dati dal disco.
00	TBE	1 (\$64)	Buffer UART di trasmissione della porta seriale VUOTO.

Come si può notare, l'analogia con DMACON/DMAONR è evidente:

- Il bit 15 è importantissimo: se esso è acceso allora i bit settati a 1 in scrittura nel \$dff09A servono ad abilitare i relativi interrupt, se il bit 15 è a 0, allora gli altri bit a 1 nel registro servono a disabilitare, ossia a mascherare, i relativi interrupt. Per abilitare o disabilitare uno o più interrupt, come in DMAONR, è comunque necessario impostare ad 1 i relativi bit; quello che determina se quegli interrupt devono venir abilitati o disabilitati è il bit 15: se è ad 1 si abilitano, mentre se è a 0 si spengono (sempre indipendentemente dal loro precedente stato). Diciamo che si sceglie su quali bit OPERARE, poi si decide se attivarli(0) o disattivarli(1) in base al bit 15. I bit 0 non sono nè settati nè azzerati.

Facciamo un esempio:

```

1      ;5432109876543210
2      move.w #01000000111000000,$dff09A ; sono ABILITATI i bits 6,7 e 8
3      ;5432109876543210
4      move.w #00000000100100000,$dff09A ; sono DISABILITATI i bits 5 e 8.
```

- Il bit 14 funge da interruttore generale (come fa il bit 9 nel DMAONR). Lo si può azzerare ad esempio per disabilitare momentaneamente tutti i livelli di interrupt, senza ricorrere all'azzeramento dell'intero registro.

Vi ricorderete della vecchia Lezione3a.s che con un:

```
1      MOVE.W #$4000,$dff09a ; INTENA — Ferma gli interrupt
```

Si bloccavano tutti gli interrupt, mentre con:

```
1      MOVE.W #$C000,$dff09a ; INTENA — Riabilita gli interrupt
```

Si riabilitavano tutti. Ebbene, \$4000 = %0100000000000000, ossia si azzerava il bit MASTER, il 14. Invece \$c000 = %1100000000000000, cioè si riabilita il bit MASTER, e con esso tutti gli interrupt. In Lezione11b.s per abilitare VERTB:

```

1      move.w #$c020,$9a(a5) ; INTENA — abilito interrupt "VERTB" del
2      ; livello 3 ($6c), quello che viene generato
3      ; una volta al fotogramma (alla linea $00).
4
5      ; 5432109876543210
6      Infatti, $c020 = %1100000000100000 — bit 5, VERTB, settato assieme al MASTER.
```

Come avrete notato, gli interrupt spaziano dalla gestione della porta seriale alla lettura della sincronia delle tracce del disk drive, senza risparmiare nè blitter nè CIA, nè COPPER. Ridefinirsi

tutti i livelli di interrupt è molto pericoloso dal punto di vista della compatibilità, ed è anche molto difficile e specifico di chi voglia farsi un proprio sistema operativo. Ai programmatori di demo e giochi interessa soltanto l'interrupt \$6c, ossia quello di livello 3, che riguarda la generazione di interrupt sincronizzati col pennello elettronico (VERTB - bit 5), o generabili a particolari linee video con il copper (COPER - bit 4). Più raramente può succedere di dover avere a che fare con interrupt per la gestione della tastiera o altro. In particolare il caricamento da disco tramite loader hardware è fuori moda, perché occorre fare giochi e demo installabili su hard disk, per cui gli interrupt del disk drive non ci interesseranno. Inoltre, anche se voleste fare un gioco che sfrutti la porta seriale per giocare in doppio su 2 computer collegati via cavo o via modem, sarebbe meglio usare le chiamate legali da sistema operativo del SERIAL.DEVICE, anziché farsi degli interrupt poco compatibili con eventuali schede multiseriali o nuovo hardware.

11.4 Come si usano INTREQ e INTREQR

In Lezione 11b.s abbiamo visto anche INTREQ/INTREQR. Di cosa si tratta? Avrete notato come è strutturato l'interrupt \$6c:

```

1 MioInt6c:
2   btst.b  #5,$dff01f      ; INTREQ - il bit 5, VERTB, è azzerato?
3   beq.s   NointVERTB      ; Se sì, non è un "vero" int VERTB!
4   movem.l d0-d7/a0-a6,-(SP) ; salvo i registri nello stack
5   bsr.w   mt_music        ; suono la musica
6   movem.l (SP)+,d0-d7/a0-a6 ; riprendo i reg. dallo stack
7 nointVERTB:
8   move.w  #%1110000,$dff09c ; INTREQ - cancello rich, BLIT,COPER,VERTB
9   ; dato che il 680x0 non la cancella da solo!!!
10  rte      ; uscita dall'int COPER/BLIT/VERTB

```

NOTA: INTREQR è la word \$dff01e/1f. In questo caso agiamo sul suo byte \$dff01f anziché su \$dff01e, ma si tratta sempre del byte basso di INTREQR.

La mappa di INTREQ/INTREQR è uguale a quella di INTENA/INTENAR:

INTREQ/INTREQR (\$dff09c/\$dff01e)

BIT	NOME	LIV.	DESCRIZIONE
15	SET/CLR		Bit di controllo "Set/clear". Determina se i bit ad 1 devono essere azzerati o settati, come in DMACON. I bit=0 non saranno né settati né azzerati
14	INTEN	6 (\$78)	interrupt livello 6 CIAB
13	EXTER	6 (\$78)	Interrupt esterno, connesso alla linea INT6
12	DSKSYN	5 (\$74)	Generato se il registro DSKSYNC corrisponde ai dati letti dal disco nel drive. Serve per i loader hardware.
11	RBF	5 (\$74)	Buffer UART di ricezione della porta seriale PIENO.
10	AUD3	4 (\$70)	Lettura di un blocco di dati del can. audio 3 finita.
09	AUD2	4 (\$70)	Lettura di un blocco di dati del can. audio 2 finita.
08	AUD1	4 (\$70)	Lettura di un blocco di dati del can. audio 1 finita.
07	AUD0	4 (\$70)	Lettura di un blocco di dati del can. audio 0 finita.
06	BLIT	3 (\$6c)	Se il blitter ha finito una blittata si setta ad 1
05	VERTB	3 (\$6c)	Generato ogni volta che il pennello elettronico è alla linea 00, ossia ad ogni inizio di vertical blank.
04	COPER	3 (\$6c)	Si può settare col copper per generarlo ad una certa linea video. Basta richiederlo dopo un certo WAIT.
03	PORTS	2 (\$68)	Input/Output Porte e timers, connesso alla linea INT2
02	SOFT	1 (\$64)	Riservato agli interrupt inizializzati via software.
01	DSKBLK	1 (\$64)	Fine del trasferimento di un blocco dati dal disco.
00	TBE	1 (\$64)	Buffer UART di trasmissione della porta seriale VUOT0.

A cosa serve un registro per la richiesta di interrupt? A richiedere interrupt, naturalmente. E anche a **disdirli**, dato che una volta che viene richiesto un interrupt, automaticamente (dai chip custom) o manualmente (dal nostro programma), viene eseguito tale interrupt, ma non viene cancellata la “richiesta”, per cui alla fine di ogni interrupt occorre cancellare gli interrupt già svolti dalla lista degli interrupt da fare.

L'INTREQ (\$dff09c) è usato dal 680x0 per forzare l'esecuzione di un interrupt, di solito l'interrupt software, oppure dal COPPER per eseguire l'interrupt COPER ad una certa linea video. Naturalmente una volta settata una richiesta di interrupt, se tale interrupt non è abilitato in INTENA si può attendere anche una vita. Quando un bit settato in INTREQ è contemporaneamente settato anche in INTENA si verifica l'interrupt corrispondente a quel bit. Attenzione alla particolarità che se il bit 14 di INTREQ è settato si verifica un interrupt di livello 6 (a patto che il corrispondente bit in INTENA, Master Enable, sia anch'esso settato) Altrimenti è usato per cancellare i bit di richiesta degli interrupt già eseguiti, dato che le richieste di interrupt non sono azzerate automaticamente. Occorre stare attenti a questo fatto, perché se si dimentica di cancellare i bit di richiesta alla fine di ogni interrupt eseguito, il processore lo eseguirà nuovamente!! Ora dovreste capire la parte finale dell'interrupt:

```

1      ;6543210
2      move.w  #01110000,$dff09c ; INTREQ - cancello rich. BLIT,COPER,VERTB
3      ; dato che il 680x0 non la cancella da solo!!!
4      rte      ; uscita dall'int COPER/BLIT/VERTB

```

L'INTREQR (\$dff01e) è a sola lettura, al contrario di INTREQ che è a sola scrittura. Serve per sapere quale chip ha richiesto l'interrupt. Infatti, se viene eseguito l'interrupt di livello 3 (\$6c), può essere “colpa” del blitter, del vertical blank o del copper. Testando i bit di INTREQR capiamo quale di queste 3 è la causa, e determiniamo quale routine eseguire, o se eseguire la routine, nel caso ci interessi solo una di queste 3 eventualità. Il bit 15 non ha significati nell'INTREQR, dato che è il Set/Clr. Rivediamo ora l'utilizzo fatto in Lezione11b.s:

```

1      btst.b  #5,$dff01f ; INTREQR - il bit 5, VERTB, è azzerato?
2      beq.s   NointVERTB ; Se sì, non è un "vero" int VERTB!

```

In questo caso, dato che il BTST su un indirizzo può essere solo .BYTE, è testato il \$dff01f, ossia il byte basso della word, anziché il \$dff01e. Nel caso che si salti a Noint, è evidente che l'interrupt è stato generato dal copper o dal blitter, e sarebbero settati i bit 4 o 6. A tal proposito occorre disdire anche queste richieste di interrupt, per evitare che ogni microsecondo si ritorni ad eseguire l'interrupt per nulla:

```

1      ;6543210
2      move.w  #01110000,$dff09c ; INTREQ - cancello rich. BLIT,COPER,VERTB
3      ; dato che il 680x0 non la cancella da solo!!!
4      rte      ; uscita dall'int COPER/BLIT/VERTB

```

Vi sembrerà strano che, nonostante sia abilitato con INTENA soltanto VERTB, possa succedere che vengano richiesti ed **eseguiti** interrupt di COPER o BLIT. Effettivamente non dovrebbero essere richiesti, né eseguiti... Ma per motivi legati probabilmente alla MMU o alla velocità del processore, su computer più veloci del 1200 base, come l'A4000, può accadere benissimo, e ciò infatti causa dei problemi a delle demo, anche alcune recenti per AGA. Infatti su A1200 base può succedere che l'interrupt funzioni anche senza il BTST del bit VERTB, ma su A4000 o A1200 turbizzato non è così. Ammetto che “in teoria” dovrebbe funzionare, ma il fatto è che molte demo per A1200, se eseguite su A4000 suonano la musica 2 volte per fotogramma, e sono ridicole. Dunque siate categorici nel testare sempre i bit di INTREQ prima di eseguire l'interrupt, anche se sul vostro computer funziona tutto, o vi troverete con un gioco/demo che fa le bizzes su a4000 e company.

Per riassumere, ecco le cose da fare per settare un nostro interrupt:

- Ottenere l'indirizzo del VBR, salvare il vecchio interrupt e ripristinarlo prima di uscire. Questo compito è svolto bene da `startup2.s`, non c'è problema: l'indirizzo del VBR è nella label `BaseVBR`.
- Azzerare tutti gli interrupt con `INTENA`. Anche questo compito è svolto dalla `startup2.s`, con un `MOVE.W #$7fff,$9a(a5)`.
- Mettere l'indirizzo del nostro interrupt nell'autovettore giusto.
- Abilitare solo l'interrupt, o gli interrupt che ci servono

Ed ecco cosa ricordarsi di mettere nella nostra routine `interrupt`:

- Salvare e ripristinare tutti i registri con un bel `MOVEM`, dato che se si “sporcasse” qualche registro, immaginatevi alla fine dell'interrupt cosa succederebbe, quando si torna ad eseguire un programma interrotto in chissà quale situazione e con chissà quali valori nei registri!
- Testare subito il `$dff01e/1f` (`INTREQR`), per sapere chi o che cosa ha generato un interrupt di quel livello. Per esempio, un interrupt di livello 3 può essere generato da `COPER`, `VERTB` o `BLITTER`; un interrupt di livello 4 da `AUD0`, `AUD1`, `AUD2` o `AUD3`, eccetera. Attenzione al fatto che anche se alle volte sembra funzionare senza questo test, su `A4000` o simili andrà tutto sfasato come se la CPU fosse ubriaca (può essere un effetto speciale, pero!)
- Cancellare i bit di `INTREQ` (`$dff09c`) che hanno causato l'interrupt eseguito, dato che non sono cancellati automaticamente. Se ci si dimentica di fare questo il processore avrà la richiesta fissa di interrupt e sarà eseguito continuamente.
- Terminare l'interrupt con un `RTE`, come si termina una subroutine con `RTS`.

Alla luce di queste considerazioni, vi ripropongo il primo interrupt:

```

1 MioInt6c:
2     btst.b    #5,$dff01f      ; INTREQR — il bit 5, VERTB, è azzerato?
3     beq.s     NointVERTB      ; Se sì, non è un "vero" int VERTB!
4     movem.l   d0-d7/a0-a6,—(SP) ; salvo i registri nello stack
5     bsr.w     mt_music         ; suono la musica
6     movem.l   (SP)+,d0-d7/a0-a6 ; riprendo i reg. dallo stack
7 nointVERTB: ;6543210
8     move.w    #%1110000,$dff09c ; INTREQ — cancello rich, BLIT,COPER,VERTB
9                                     ; dato che il 680x0 non la cancella da solo!!!
10    rte          ; uscita dall'int COPER/BLIT/VERTB

```

11.5 Gli interrupt e il sistema operativo

Il 680x0 ha solo 7 livelli di `INTERRUPT`, ma allora come è possibile che gli interrupt in pratica siano 15? Ebbene, il chip Paula si occupa di dividere in pseudointerrupt i 7 livelli “reali” di interrupt. Per esempio fa saltare all'interrupt di livello 3 in tre casi: `COPER`, `VERTB` e `BLIT`, e l'unico modo di sapere quale di queste tre eventualità ha generato l'interrupt è di consultare un registro collegato al chip Paula stesso, ossia `INTREQR`! D'altronde, essendo solo 7 i livelli di interrupt “reali” del 680x0, non è possibile che interrupt “sdoppiati” da Paula dello stesso livello possano interrompersi tra loro. Mentre un interrupt di livello 5, come il `DSKSYNC`, può interrompere l'esecuzione di un'interrupt di livello 3, come `COPER`, non è possibile per `BLIT` interrompere `COPER`, anche se ha “priorità Paulesca” maggiore, dato che si trovano nello stesso livello fisico del 680x0. Per questo se durante l'esecuzione di un interrupt si verifica la richiesta di interrupt di un altro pseudo-livello di Paula nello stesso livello 680x0, come ad esempio uno

BLIT mentre si esegue un COPER, al termine dell'int che esegue il COPER verrà eseguito subito un'altra volta l'interrupt di livello 3, questa volta eseguendo la routine per COPER (secondo il BTST fatto sull'INTREQR verrà identificato quale tipo di "sottoint" eseguire). Ecco le priorità dei livelli di interrupt nel sistema operativo, ossia nell'Exec.library, che come vedete segue la priorità hardware:

livello 1: (\$64) MINIMA PRIORITÀ		
1	buffer di trasmissione vuoto	TBE
2	blocco del disco trasferito	DSKBLK
3	interrupt software	SOFTINT
livello2: (\$68)		
4	porte esterne INT2 & CIAA	PORTS
livello3: (\$6c)		
5	copper	COPER
6	intervallo di vertical blank	VERTB
7	blittata finita	BLIT
livello4: (\$70)		
8	canale audio 2	AUD2
9	canale audio 0	AUD0
10	canale audio 3	AUD3
11	canale audio 1	AUD1
livello5: (\$74)		
12	buffer di ricezione pieno	RBF
13	sync del disco trovata	DSKSYNC
livello6: (\$78) MASSIMA PRIORITÀ		
14	external INT6 & CIAB	EXTER
15	speciale (abilitazione master)	INTEN
livello7: (\$7c) (schede esterne come la Action Replay)		
-	interrupt non mascherabile	NMI

Il fatto che il sistema operativo gestisca sue routines interrupt rende più pericolosa la sostituzione di alcuni di essi da parte nostra. Per quanto riguarda la priorità 6, la graphics.library usa l'interrupt del timer CIAB Time Of Day (TOD) per controllare lo schermo. Nella priorità 5, DSKSYNC è usato dal TrackDisk e RBF dal serial.device. Nel livello 4, ci sono i canali audio, usati dall'audio.device. Nel livello 3, l'interrupt BLIT, che avviene quando il blitter ha finito una sua operazione, spesso vengono messe routines che si occupano di riutilizzare i dati appena scritti dal blitter, per evitare di perdere tempo. Nel livello 2, del chip CIAA il Timer.device usa l'interrupt TimerA per l'handshake di tastiera, il TimerB per il timer a microsecondi, e l'interrupt TOD alarm a 50/60Hz. Esiste anche l'INT2 per eventuali schede hardware esterne. Nel livello 1, il più basso, l'interrupt TBE è usato dal Serial.device, l'interrupt DSKBLK è usato dal TrackDisk.device. Gli interrupt SOFTINT, ossia software, sono definibili via sistema operativo, ad esempio con la funzione Cause dell'Exec o facendo una message port di tipo SOFT_INT.

11.6 Gli interrupt COPER chiamati da COPPERList

Se si vuole chiamare l'interrupt COPER del livello 3 (\$6c) ad una certa linea video, basta scrivere \$8010 nell'intreq (\$dff09c), dopo un wait che aspetti quella linea video:

```

1 COPPERLIST:
2   dc.w   $100,$200      ; BPLCONO — no bitplanes
3   dc.w   $180,$00e      ; color0 BLU
4   dc.w   $a007,$fffe    ; WAIT — attendi la linea $a0
5   dc.w   $9c,$8010      ; INTREQ — Richiedo un interrupt COPER, il
6                       ; quale fa agire sul color0 con un "MOVE.W".
7   dc.w   $FFFF,$FFFE    ; Fine della copperlist

```

Infatti il valore \$8010 = \$8000 + %10000, ossia si setta il bit 4, COPER. Vediamo in Lezione11c.s un esempio pratico.

Naturalmente l'interrupt si può chiamare anche a diverse linee, ogni volta cambiando “effetto”. Verifichiamolo in Lezione11d.s.

Data la particolare complessità degli interrupt, per ora non faremo esempi ulteriori, riguardo ad interrupt dei dischi, della porta seriale ecc. Alle applicazioni che ci interessano, ossia DEMO e GIOCHI, bastano spesso solo i due tipi di interrupt di livello 3 (\$6c) che abbiamo visto, ossia il VERTB, che viene eseguito ogni fotogramma, e il COPER, richiamabile dal copper a una qualsiasi linea video. Le applicazioni degli altri interrupt saranno commentate man mano che saranno trovate nei listati esempio, dato che spaziano in ogni campo! Per ora, possiamo anticipare l'uso di tutti i livelli di interrupt: nei listati Lezione11e.s e Lezione11f.s sono ridefiniti TUTTI gli interrupt, e sono abilitati TUTTI i livelli, ma naturalmente ci sono routines solo nel livello 3. Questo esempio può essere utile come “partenza” per definirsi un qualsiasi livello interrupt: potete “ritagliare” il livello che vi interessa e metterci le routines “dentro”.

11.7 Informazioni avanzate sul COPPER - Uso del solo COLOR0 (\$180) - No BitPlanes

Come recita il titolo, ora vedremo le applicazioni possibili usando solo una copperlist senza bitplanes. Ossia creare con soli WAIT e MOVE dei disegni o delle animazioni. Naturalmente, se aggiungete bitplanes potete “incrociare” e “sovrapporre” questi effetti, modificando molte volte per copperlist il color2 o il color3, oltre al color0. Per iniziare, però, occorre spiegare delle cose che non sono ancora state trattate. Si tratta del “tempo” impiegato dal copper per eseguire un suo comando MOVE. Abbiamo già visto come cambiare l'intera palette, anche di 32 colori, ad una certa linea video, per far comparire su schermo qualche centinaio di colori, col solo “32” o “16” colori ufficialmente settati nel BPLCON0. Ebbene, in una linea siamo riusciti a cambiare 32 colori, ossia ad eseguire 32 MOVE del copper:

```

1   dc.w   $180,xxx      ; 1 move nel color0
2   dc.w   $182,xxx      ; 1 move nel color1
3   ...                ; ecc.

```

Ebbene, in realtà se cominciamo a cambiare i colori dalla posizione orizzontale \$07, o anche \$01, tutti i 32 colori saranno effettivamente cambiati solamente verso la metà dello schermo, perché ogni MOVE richiede 8 pixel lowres per essere eseguito. Per questo conviene sempre cambiare i colori 1 linea prima che inizi veramente il disegno. Se mettessimo una cinquantina di MOVE di seguito, arriveremmo con l'ultima alla linea sotto! D'altronde, sarebbe materialmente impossibile eseguire decine di MOVE in meno di 1 cinquantesimo di cinquantesimo di secondo! Comunque possiamo usare questa apparente limitazione per i nostri scopi, ad esempio per cambiare colore in senso orizzontale ogni 8 pixel, senza usare i WAIT, ma semplicemente affiancando una cinquantina di COLOR0 per linea. Vediamo un esempio pratico di questo in Lezione11g1.s.

Un utilizzo di questo listato potrebbe essere quello di cambiare il \$182, e non il \$180, in uno schermo a 1 bitplane: in questo modo le eventuali “scritte” in sovraimpressione sarebbero sfumate da sinistra verso destra anziché dall’alto verso il basso, come facciamo di solito con la copperlist.

In Lezione11g2.s e Lezione11g3.s ci sono versioni più colorate di questo effetto, tanto che possono essere una base per degli effetti “PLASMA”.

A proposito, se “roteassimo” o “ciclassimo” i colori di una linea di questo tipo cosa avremmo? Un effetto molto noto, usato dalle intro già negli albori dell’Amiga: vediamo l’effetto “supercar” in Lezione11g4.s.

Forse ciclare solo 2 linee non esalta. Vediamo di ciclarne di più, magari facendo un effetto di tipo “annodato”, in Lezione11g5.s.

Un’altra “fantasia” che sfrutta l’“affollamento” di colorXX messi di seguito, nell’esempio Lezione11g6.s.

Ora vediamo un modo un pò “economico” per fare un simil-plasma: anziché cambiare i contenuti dei molti color0, mettiamo un wait all’inizio di ogni linea, ognuna delle quali conterrà 52 color0: possiamo spostare la linea a destra e a sinistra semplicemente cambiando la posizione orizzontale dei vari wait! In pratica questo è in Lezione11g7.s.

11.8 Uso della COPPER2 (COP2LC/COPJMP2)

Avrete notato che oltre al \$dff080 e il \$dff088 per puntare e far partire la copper 1, esistono il \$dff084 e il \$dff08a per puntare e far partire la copper2. Ma come funziona la copper2? E in che cosa può servirci? Ogni inizio frame il copper fa partire la copper 1, il cui indirizzo viene letto dal \$dff080. Noi a volte lo facciamo partire “al volo”, senza attendere nemmeno la fine del frame, scrivendo nel COPJMP1, ossia \$dff088. Se mettessimo una copperlist nel \$dff084, (COP2LC), dovremmo anche farla partire scrivendo nel COPJMP2 (\$dff08a). Ma alla fine del frame ripartirebbe la copper1. Ora, questa caratteristica può servire per fare diverse copperlist a cui saltare, come facciamo per le istruzioni 680x0 con i JMP. Per esempio, se volessimo eseguire fino a metà schermo la copper1, dopodiché saltare ad eseguire l’altra metà dalla copper 2, basterebbe puntare la copper 2 all’inizio, e farla partire dalla copperlist tramite COPJMP2:

```

1      move.l  #copper1,$dff080      ; COP1LC
2      move.l  #copper2,$dff084      ; COP2LC
3      ...
4
5      section copperissime,data_C
6
7  copper1:
8      ...      ; istruzioni varie...
9      dc.w    $a007,$fffe      ; Aspetta linea $a0
10     dc.w    $8a,0            ; COPJMP2 — fai partire la copper 2
11
12
13  copper2:
14     ...      ; istruzioni varie
15     dc.w    $ffff,$fffe      ; Fine della copperlist, si ripartirà con
16                                ; la copper1!
```

Il copper, arrivato al dc.w \$8a,0 salta (come BRA o JMP) a copper2, a patto che questa fosse messa in \$dff08a preventivamente. Da notare che SALTA, e non fa come un BSR, per cui non torna mai sotto il dc.w \$8a,0 della copper1. Vediamo ora un paio di utilizzi pratici della copper2. Uno è quello di fare le cosiddette copper dinamiche, ossia composte da 2 copperlist scambiate ogni fotogramma, come un “double buffering” dei bitplane. Questo serve per rendere più “lisce” le sfumature, infatti se scambiate 2 colori ogni fotogramma, avrete un effetto tipo l’interlace, che

farà “vedere” il colore intermedio. Basta prepararsi 2 copperlist con la stessa sfumatura, ma un poco “sfasata”, e scambiarle continuamente.

Vediamo in pratica una *Dynamic Cop* in *Lezione11h1.s*. Avete notato la differenza? Si può spacciare per una sfumatura AGA! E dire che le copper dinamiche sono state usate in pochi giochi, nonostante non siano poi così difficili da fare. Tra i giochi che hanno copper dinamiche devo ricordare AGONY, e il recente SHADOW FIGHTER, il picchiaduro italiano dei NAPS TEAM.

Ora vediamo un'altra applicazione della copper2. Anziché scambiare un paio di copperlist, possiamo ciclarne qualche decina, per cui possiamo dire che si può “precalcolare” un effetto copper calcolando 1 copperlist per ogni “fase” dell'effetto: dato che poi l'effetto sarà ciclico, basterà puntare ogni volta alla copperlist “dopo”. In questo modo otteniamo l'effetto copper, ma in termini di tempo risparmiamo TOTALMENTE quello che sarebbe stato usato dalle routines 68000! Per cui si può dire che l'effetto copper in questione è “GRATIS”, e possiamo farci girare sopra una routine che mangia tutto il resto del tempo.

Vediamo una routine “normale”, in *Lezione11h2.s*, e la versione che precalcola i fotogrammi “copper”, ossia *Lezione11h3.s*. L'unico inconveniente della versione “turbizzata/precalcolata” è che serve della memoria aggiuntiva per memorizzare tutte le copperlist-frames.

Visto che volete superare l'esame di copper livello 2, dovete sapere anche che si può “mascherare” la coordinata Y dei WAIT. In pratica, un WAIT con la Y mascherata è così:

```
1 dc.w      $0007,$80FE      ; Wait ad Y "mascherata"
```

E significa: non controllare la linea Y, ma aspetta la posizione \$07 X della linea attuale. è un WAIT “handicappato”, che non sa leggere la posizione Y. In realtà non sa leggere i 7 bit bassi della posizione Y, per cui funziona dopo la linea \$80. Ma allora, a cosa ci serve un wait mascherato che non controlla la posizione Y prima della posizione Y \$80? Se avessimo da muovere una mitica barretta, di quelle della lezione 3, dovremmo cambiare tutti i wait che la compongono. Se invece mettiamo un wait normale all'inizio, e sotto tutti wait mascherati, basterà cambiare il primo wait e gli altri “seguiranno”. Il risparmio di istruzioni 680x0 è evidente: con un solo ADD/SUB si sposta un'intera barretta.

Vediamo un'implementazione in *Lezione11h4.s* (mitica barretta di *Lezione3!*)

Da notare che funziona anche sotto la linea verticale \$FF, dato che riparte la numerazione da \$00. Ora che lo sapete, se vi capita di spostare qualche cosa in quella zona potete sfruttare questo trucchetto.

Ora si potrebbe parlare dell'istruzione SKIP, ma dato che non la ho mai vista usare da nessuno, e io stesso non vedo a cosa possa servire (si possono fare tranquillamente tutte le cose usando la copper2 per i salti...), tralascio questo argomento, spero crediate nella totale inutilità di tale comando.

Ora, per finire l'argomento *solo Copper senza bitplanes*, vi propongo 6 listati che riassumono i più comuni effetti di questo tipo.

Lezione11i1.s è uno scroll di colori a tutto schermo. *Lezione11i2.s* è una pseudo paralasse a 3 livelli di barre. Può servire come sfondo per un gioco platform durante una “salita”, ad esempio. *Lezione11i3* è una fantasia in COP minore... *Lezione11i4.s* è una sfumatura pseudocasuale, che mixa i valori della posizione orizzontale del pennello elettronico (di solito valori diversi), per farne i colori della sfumatura copper. *Lezione11i5.s* è una copperlist che cicla i colori in un modo che sembri 3d.

11.9 Informazioni avanzate sul COPPER - Anche i BitPlanes abilitati

Avete visto che con i soli MOVE&WAIT del copper possiamo fare un bel po' di cose? Ma cosa succede se facciamo copperlist complicate con i bitplanes abilitati? Possiamo cambiare il BPLMOD

ad ogni linea per allungare le figure, o usare il bplcon1 (\$dff102) per ondeggiarle, o addirittura cambiare ad ogni linea i puntatori ai bitplanes!!!

Nei listati seguenti, tra l'altro, è usato un sistema particolare per calcolare il DIWSTART/DIWSTOP, e DDFSTART/DDFSTOP, ossia tramite alcuni EQUATE, che sono usati per calcolare i valori grazie agli operatori di shift “<” e “>”, nonché “&” (and), e i comuni “*”, “/”, “+”, “-”. Se volete fare uno schermo in 320*256 normale, si fa prima a mettere i valori normali, o cambiarli a mano. Se invece voleste fare uno schermo di grandezza particolare, ad esempio 256*256, può risparmiare tempo.

```

1  scr_bytes      = 40      ; Numero di bytes per ogni linea orizzontale.
2                        ; Da questa si calcola la larghezza dello schermo,
3                        ; moltiplicando i bytes per 8: schermo norm. 320/8=40
4                        ; Es. per uno schermo largo 336 pixel, 336/8=42
5                        ; larghezze esempio:
6                        ; 264 pixel = 33 / 272 pixel = 34 / 280 pixel = 35
7                        ; 360 pixel = 45 / 368 pixel = 46 / 376 pixel = 47
8                        ; ... 640 pixel = 80 / 648 pixel = 81 ...
9
10 scr_h           = 256     ; Altezza dello schermo in linee
11 scr_x           = $81     ; Inizio schermo, posizione XX (normale $xx81) (129)
12 scr_y           = $2c     ; Inizio schermo, posizione YY (normale $2cxx) (44)
13 scr_res         = 1       ; 2 = HighRes (640*xxx) / 1 = LowRes (320*xxx)
14 scr_lace        = 0       ; 0 = non interlace (xxx*256) / 1 = interlace (xxx*512)
15 ham             = 0       ; 0 = non ham / 1 = ham
16 scr_bpl         = 1       ; Numero Bitplanes
17
18 ; parametri calcolati automaticamente
19
20 scr_w            = scr_bytes*8      ; larghezza dello schermo
21 scr_size         = scr_bytes*scr_h  ; dimensione in bytes dello schermo
22 BPLCO           = ((scr_res&2)<<14)+(scr_bpl<<12)+$200+(scr_lace<<2)+(ham<<11)
23 DIWS            = (scr_y<<8)+scr_x
24 DIWSt           = ((scr_y+scr_h/(scr_lace+1))&255)<<8+(scr_x+scr_w/scr_res)&255
25 DDFS            = (scr_x-(16/scr_res+1))/2
26 DDFSSt          = DDFS+(8/scr_res)*(scr_bytes/2-scr_res)

```

Poi, in copperlist metteremo:

```

1      dc.w      $8e,DIWS      ; DiwStrt
2      dc.w      $90,DIWSt     ; DiwStop
3      dc.w      $92,DDFS      ; DdfStart
4      dc.w      $94,DDFSSt    ; DdfStop
5      dc.w      $100,BPLCO     ; BplCon0

```

Comunque, non è “infallibile”, se volete fare schermi di grandezze strane potrebbe non funzionare, e sarà meglio andare “a mano”. Si può anche usare per calcolare il valore, controllandolo dopo l'assemblaggio con ? DIWS o ? xxxx, poi scrivere il valore a mano. Ecco i listati di questa sezione:

Lezione1111.s – Cambia ad ogni linea sia il COLOR0 che il BPLCON1 (\$dff102), causando l'ondulazione dei bitplanes.

Lezione1112.s – Si cambiano per ogni linea ben 3 colori su 4 (2 bitplanes).

Lezione1113a.s, Lezione1113b.s e Lezione113c.s – Sono 3 passaggi per arrivare a fare l'effetto di ondeggiamento del logo *AMIGA ET* della piccola demo presente nel disco 1. A forza di pezzi la abbiamo descritta tutta! La figura ondeggia grazie ai moduli negativi alternati a quelli azzerati.

Lezione1114.s – Questo è un altro modo per ondeggiare: sono ridefiniti i bplpointers ad ogni linea!

Lezione1115.s – Se aveste una piccola immagine larga 40*29 pixel, e voleste riempirci lo schermo intero cosa potreste fare? Provare a fare uno zoom di 8 volte, trasformandola in 320*232. è quello che fa questo listato, usando per l'allungamento verticale il modulo, per quello orizzontale una routine che testa ogni bit e lo “trasforma” in un byte (8 bit).

Lezione1115b.s – È una versione ottimizzata del listato precedente, che usa una tabella contenente le 256 possibili combinazioni di un byte “espanso” a 8 byte. Richiede meno della metà di tempo per eseguire! Imparate a farvi questo tipo di ottimizzazioni: routines che sembrerebbero impossibili da rendere più veloci, alle volte possono subire una turbizzazione così!

11.10 Come fare uno schermo in interlace (lungo 512 linee)

Il modo interlacciato permette di visualizzare il doppio di dati video. Questo è possibile raddoppiando il numero di linee visualizzate. Normalmente sono possibili 256 linee verticali, mentre con l'interlace si può arrivare a 512, sia in lowres che in hires. Ci sono però delle particolarità, infatti non basta puntare i bitplanes e settare il bit di interlace (il bit 2 del BPLCON0). Per quanto riguarda la figura RAW, basta convertire un normale disegno interlacciato con l'iffconverter e salvarlo, ossia una figura in 320x512 o in 640x512. Anche un brush più piccolo va bene, ma considerate sempre che l'immagine non deve risultare “schiacciata” per la doppia risoluzione verticale. Come è noto l'interlacciato “sfarfalla”, “traballa”. Questo è un male, ma anche un bene, infatti su normali TV o monitor non sarebbe possibile una risoluzione verticale superiore a 256 linee, occorrerebbe un monitor “VGA”, ossia multisync o multiscan. Il “truccetto” è fatto visualizzando una volta solo le 256 linee dispari, e l'altra le altre 256 linee pari. Lo scambio avviene ogni frame per cui inganna decentemente l'occhio, a parte lo sfarfallio, (che, se sono scelti bene i colori, diminuisce molto). Questo scambio però non è del tutto “automatico”, occorre fare qualcosina noi “a mano”.

QUADRO 1 (linee dispari)	QUADRO 2 (linee pari)
LINEA 1: ---> xxxxxxxxxxxx	LINEA 2: ---> xxxxxxxxxxxx
LINEA 3: ---> xxxxxxxxxxxx	LINEA 4: ---> xxxxxxxxxxxx
LINEA 5: ---> xxxxxxxxxxxx	LINEA 6: ---> xxxxxxxxxxxx
LINEA 7: ---> xxxxxxxxxxxx	LINEA 8: ---> xxxxxxxxxxxx
LINEA 9: ---> xxxxxxxxxxxx	LINEA 10: --> xxxxxxxxxxxx
[...]	[...]
LINEA 311: -> xxxxxxxxxxxx	LINEA 312: -> xxxxxxxxxxxx
LINEA 313: -> xxxxxxxxxxxx	

Per il modo interlacciato occorre ridefinire il modulo mettendolo a 40 se siamo in lowres, o a 80 se siamo in hires. In pratica occorre mettere nel modulo la lunghezza di una linea, per saltarla: essendo il modulo un valore che viene aggiunto alla fine di ogni linea video, se saltiamo la lunghezza di una intera linea ecco cosa succede: viene letta e visualizzata la prima linea, alla fine viene saltata la seconda, e sotto viene visualizzata la terza; alla fine di questa viene saltata una linea e viene visualizzata la quinta, eccetera. In pratica abbiamo fatto in modo che siano

visualizzate solo le linee dispari. Lo schermo per motivi hardware non può visualizzare più di 256 linee, ma nessuno ci impone quali 256, nè da quando farle iniziare a visualizzare. Se una volta saltiamo le linee pari, e quella dopo saltiamo le linee dispari, possiamo visualizzare una schermata di 512 linee in una da 256 “alternata”!

Ricapitoliamo: abbiamo una pic, per esempio, in 640x512 interlacciata, che abbiamo convertito in RAW e vogliamo visualizzare. Abbiamo puntato la pic opportunamente e messo il modulo a -80, nonché settato il bit di interlace (oltre a quello di hires) nel bplcon0 (\$dff100). Cosa otteniamo? **La figura come fosse in lowres! Alta 256 linee, come se l'avessimo abbassata di risoluzione!**

Allora, qua è il momento di fare quella piccola parte “a mano” per permettere l’interlacciamento. Esiste un apposito bit, che va controllato, che ci indica se visualizzare le linee dispari o quelle pari a ogni fotogramma. Questo è il bit 15 del VPOSR (\$dff004), detto LOF, o Long Frame, che indica se siamo nel “frame lungo” o no. Ecco un esempio di routine:

```

1 LACEINT:
2     MOVE.L #BITPLANE,D0 ; Indirizzo bitplane
3     btst.b #15-8,$dff004 ; VPOSR LOF bit?
4     Beq.S FaiDispari ; Se sì, tocca alle linee dispari
5     ADD.L #80,D0 ; Oppure aggiungi la lunghezza di una linea,
6 ; facendo partire la visualizzazione dalla
7 ; seconda: visualizzate linee pari!
8 FaiDispari:
9     LEA BPLPOINTERS,A1 ; PLANE POINTERS IN COPLIST
10     MOVE.W D0,6(A1) ; Punta la figura
11     SWAP D0
12     MOVE.W D0,2(A1)
13     RTS

```

Come vedete, se il bit LOF è azzerato si parte a visualizzare dalla prima linea, quindi per effetto del modulo saranno visualizzate le linee 1,3,5,7... eccetera, ossia quelle dispari. Altrimenti, si salta una linea, facendo partire la visualizzazione dalla seconda, quindi 2,4,6,8... eccetera: pari!

C'è chi fa 2 copperlist, una che punta in un modo e una che punta nell'altro, e poi a seconda del bit LOF punta una o l'altra ogni fotogramma. Credo però sia più “furbo” puntare solo i bitplanes... comunque potete fare come volete, basta aver capito il metodo.

Lezione1116.s – è un esempio in 640x512 ad 1 bitplane

Lezione1116b.s – è un esempio in 320x512 a 4 bitplanes

Per terminare questo corso di laurea in “copper”, livello 2, non ci rimane che fare un esempio più complesso anche per gli sprite. Vi ricordate come li “riutilizzammo” nella lezione7 per fare le stelline? Ebbene, cosa succede se riutilizziamo gli sprite ogni 2 linee?

Lezione1117.s – Per vedere un mega riutilizzo degli sprite (128 volte ognuno)

11.11 I due chip 8520, detti CIAA e CIAB

Se smontate l'Amiga troverete, oltre ad Agnus, Paula, Denise, il 680x0 ecc., anche un paio di chippetti 8520, detti CIA. Questi chip hanno 16 pin di Input/Output ognuno, un registro di shift seriale, tre timer, un pin di solo output e uno di solo input. Di conseguenza entrambi hanno 16 registri che si possono raggiungere accedendo ai loro relativi indirizzi:

Mappa di indirizzi del CIAA

Byte	Register	Data bits							
Address	Name	7	6	5	4	3	2	1	0

\$BFE001	pra	/FIR1 /FIRO /RDY /TKO /WPRO /CHNG /LED OVL
\$BFE101	prb	Porta parallela
\$BFE201	ddra	Direzione per port A (BFE001);1=output (normalmente \$03)
\$BFE301	ddrb	Direzione per port B (BFE101);1=output (può essere in/out)
\$BFE401	talo	CIAA timer A byte basso (.715909 Mhz NTSC; .709379 Mhz PAL)
\$BFE501	tahi	CIAA timer A byte alto
\$BFE601	tblo	CIAA timer B byte basso (.715909 Mhz NTSC; .709379 Mhz PAL)
\$BFE701	tbhi	CIAA timer B byte alto
\$BFE801	todlo	Timer a 50/60 Hz - bits 7-0 (VSync or line tick)
\$BFE901	todmid	Timer a 50/60 Hz - bits 15-8
\$BFEA01	todhi	Timer a 50/60 Hz - bits 23-16
\$BFEB01		Non usato
\$BFEC01	sdr	CIAA serial data register (connesso alla tastiera)
\$BFED01	icr	CIAA interrupt control register
\$BFEE01	cra	CIAA control register A
\$BFEF01	crb	CIAA control register B

Nota: il CIAA può generare un interrupt INT2, ossia liv.2, \$68.

Mapa di indirizzi del CIAB

Byte Address	Register Name	7	6	5	4	3	2	1	0
\$BFD000	pra	/DTR	/RTS	/CD	/CTS	/DSR	SEL	POUT	BUSY
\$BFD100	prb	/MTR	/SEL3	/SEL2	/SEL1	/SEL0	/SIDE	DIR	/STEP
\$BFD200	ddra	Direction for Port A (BFD000);1 = output (normalm. a \$FF)							
\$BFD300	ddrb	Direction for Port B (BFD100);1 = output (normalm. a \$FF)							
\$BFD400	talo	CIAB timer A byte basso (.715909 Mhz NTSC; .709379 Mhz PAL)							
\$BFD500	tahi	CIAB timer A byte alto							
\$BFD600	tblo	CIAB timer B byte basso (.715909 Mhz NTSC; .709379 Mhz PAL)							
\$BFD700	tbhi	CIAB timer B byte alto							
\$BFD800	todlo	Timer di sync orizzontale - bits 7-0							
\$BFD900	todmid	Timer di sync orizzontale - bits 15-8							
\$BFDA00	todhi	Timer di sync orizzontale - bits 23-16							
\$BFDB00		Non usato							
\$BFDC00	sdr	CIAB serial data register (non usato)							
\$BFDD00	icr	CIAB interrupt control register							
\$BFDE00	cra	CIAB Control register A							
\$BFDF00	crb	CIAB Control register B							

Nota: CIAB può generare un INT6, ossia liv 6: \$78.

Da questa "mappa" si può vedere come i compiti dei 2 CIA vadano dalla lettura della tastiera, alla gestione della porta seriale, (per scambio dati tra 2 computer o tra computer e modem), alla gestione della porta parallela (per la stampante, ad esempio), al controllo delle testine del disk drive, inoltre ha degli "orologi" che possono contare microsecondi o ore.

In realtà però non ci interesseranno tutte queste caratteristiche, per vari motivi. Innanzitutto, la parte riguardante l'hardware dei disk drive può essere omessa, dato che ogni buon gioco/demo che si rispetti deve essere installabile su Hard Disk (o CD-ROM!), come ad esempio Brian The Lion. Per quanto riguarda la gestione della porta parallela e della porta seriale, gli utilizzi potrebbero essere quello di stampare qualche istruzione per il gioco o qualche frase detta da un personaggio, oppure per la seriale la possibilità di giocare in due con i computer in rete, ossia connessi con un cavo. Però occorre dire che la gestione della stampante è bene farla fare al sistema operativo, usando il `parallel.device`. Lo stesso dicasi della porta seriale: il `serial.device` è certamente più sicuro di routines scritte via hardware, specialmente per gli Amiga futuri o

per le schede multiseriali. Per quanto riguarda i timers, dato che il sistema operativo ne utilizza diversi per le proprie mansioni, vedremo se e quali utilizzare. Ma allora, ci interessa quasi esclusivamente la lettura da tastiera? Ebbene sì, infatti se sbirciaste nel codice di un videogioco, notereste che vengono ridefiniti solo gli interrupt \$6c (COPER/VERTB/VBLANK) e \$68 (INT2 del ciaa di tastiera): il livello 3 (\$6c) serve per mettere la musica o altre routines in sincronia col pennello elettronico, mentre il livello 2 (\$68) per leggere la tastiera. Naturalmente con la lettura del tasto sinistro del mouse o di altre cose si accede a registri cia, ma si tratta di semplici controlli o settaggi di BIT, non occorre fare lunghe dissertazioni su `btst.b #6,$bfe001`. Nelle demo, addirittura, è facile che non sia ridefinito alcun interrupt, oppure che sia usato solo il \$6c per metterci la musica.

Quindi, cominciamo la spiegazione del CIAA dalla gestione della tastiera, in cui sono coinvolti i registri \$bfec01 (SDR), \$bfed01 (ICR), \$bfee01 (CRA) e l'interrupt di livello 2 (\$68). Prima vediamo i 3 registri separatamente, poi facciamo degli esempi sul loro corretto utilizzo. Premetto che quando un tasto viene premuto o rilasciato viene inviato dalla tastiera un codice a 8 bit attraverso \$bfec01, e viene generato un interrupt di livello 2 (\$68) nel quale occorre "dire" alla tastiera che si è ricevuto il codice di quel tasto. Da notare che tale codice NON è il codice ascii del carattere premuto, ma un codice con la posizione del bottone premuto nella tastiera.

```
*****
;* BFEC01    sdr        CIAA sdr (serial data register - connesso alla tastiera)
*****
```

è un registro di shift a 8 bit sincrono, connesso alla tastiera. Esso può funzionare in 2 modi: INPUT o OUTPUT, e la selezione tra questi due modi si può fare agendo sul bit 6 di \$bfee01 (CRA). Nel modo INPUT i dati ricevuti dalla tastiera sono inseriti nel registro, un bit alla volta e, quando sono "arrivati" tutti gli 8 bit che formano il carattere premuto, si genera un'interrupt INT2 (\$68), dal quale occorre vedere di quale tasto si tratta e annotarlo in qualche variabile. In questo caso si legge il byte corrispondente al codice del carattere: Es:

```
1  move.b $bfec01,d0
```

Nel modo OUTPUT invece si scrive sul registro, ad esempio `clr.b $bfec01`.

```
*****
;* BFED01    icr        CIAA interrupt control register
*****
```

Questo registro controlla gli interrupt generabili dal CIAA. I Cia infatti generano degli interrupt in varie occasioni, per esempio quando è finito un conto alla rovescia di un timer, o quando la porta seriale ha finito un trasferimento. A noi in particolare interessa l'interrupt INT2, di livello2, ossia il vettore di offset \$68, che viene generato quando è premuto un tasto. Il funzionamento degli icr (\$bfed01 per CIAA e \$bfdd00 per CIAB) è molto particolare, infatti consistono in una "maschera" a sola scrittura di un registro dati a sola lettura. Ma cosa significa questo? Innanzitutto che è molto facile sbagliarsi e far impazzire gli interrupt del CIA, il che è poco augurabile. Ogni interrupt risulta abilitato se il corrispondente bit della maschera è settato ad 1, infatti ogni interrupt CIAA, come farebbe con un INTREQ (\$dff09c), setta il suo bit di richiesta in questo registro. A questo punto, se tale interrupt è abilitato, si setta il bit 7 (IR), che è una specie di SET/CLR bit, come in DMACON, ossia quando tale bit è azzerato gli altri 6 bit settati sono azzerati, quando il bit 7 è settato, invece, gli altri bit settati sono settati, mentre quelli a zero non vengono modificati. La cosa che può confondere è che quando si legge il registro il suo contenuto viene azzerato, sia che si faccia un `tst.b $bfed01` che qualsiasi azione di lettura; azzerando il registro si elimina anche la richiesta di interrupt, in modo analogo all'azzeramento

dei bit di INTREQ (\$dff09c). Ora ci interessa solo la sua funzione per l'interrupt di tastiera, per cui vediamo in breve i suoi bit in modo lettura, con commenti solo dove interessa:

CIAA ICR (\$bfed01)

BIT	NOME	DESCRIZIONE
07	IR	Bit che indica, se settato, che c'è un interrupt in corso
06	0	
05	0	
04	FLG	
03	SP	Se settato, siamo in un interrupt generato dalla tastiera
02	ALRM	
01	TB	
00	TA	

Ricordarsi che se si legge il registro questo si azzerà, per cui se volete sapere quali bit fossero settati, dovete copiarlo in un registro Dx e fare dei controlli su tale registro: rileggendo \$bfed01 i bit sono azzerati.

```
*****
;* BFEE01   cra   CIAA cra (control register A)
*****
```

Questo registro è detto “di controllo” proprio perché i suoi bit controllano la funzione di altri registri. Ecco una sua “mappa”, con i commenti solo ai bit che ci interessano per la lettura da tastiera:

CIA Control Register A

BIT	NOME	FUNZIONE
---	----	-----
0	START	Timer A
1	PBON	Timer A
2	OUTMODE	Timer A
3	RUNMODE	Timer A
4	LOAD	Timer A
5	INMODE	Timer A
6	SPMODE	Se è a 1 = registro (\$bfec01) output (per scriverci) Se è a 0 = registro (\$bfec01) input (per leggerlo)
7	Non usato	

Come si vede, l'unico bit che ci interessa è il 6, che “decide” la funzione del \$bfec01, ossia se la sua direzione sia “verso la tastiera” (output), per cui possiamo scriverci, o “dalla tastiera verso l'Amiga” (input), per cui possiamo leggere il carattere relativo al tasto che è stato premuto. Per cambiare modalità basta fare questo:

```
1      bset.b  #6,$bfec01      ; CIAA cra - sp ($bfec01) output
2      ....
3      bclr.b  #6,$bfec01      ; CIAA cra - sp (bfec01) input
```

Oppure, se vi pare più elegante, potete usare AND e OR per farlo:

```
1      or.b    #$40,$bfec01    ; SP OUTPUT (%0100000, settiamo il bit 6!)
2      ...
3      and.b   #$bf,$bfec01    ; SP INPUT  (%10111111, azzeriamo il bit 6!)
```

Potreste anche muovere “0000” in un registro, moltiplicarlo per 5, dividerlo per 5, aggiungere 20, sottrarre 10, aggiungere 1, sottrarre 11, settare o azzerare il bit 6, e fare un and o un or con il \$bfec01, l'assembler permette di usare infinite strade per fare la stessa cosa. Ma il BSET/CLR

può bastare! Comunque occorre precisare che tra modo input e modo output occorre attendere una novantina di microsecondi, dato che l'hardware del CIAA e del chip della tastiera non può temporizzarsi da solo in modo input. Gli 8 bit del carattere corrispondente al tasto premuto sono trasferiti serialmente un bit alla volta dal chip della tastiera al CIAA. Quando tutti e 8 i bit sono stati trasferiti, **dobbiamo abbassare la linea KDAT per almeno una novantina di microsecondi (o 3/4 linee raster) per confermare alla tastiera che abbiamo ricevuto i dati.** Il "filo" KDAT si controlla dal bit SP/SPMODE, e in pratica dobbiamo fare questo:

```

1      move.b  $bfec01,d0      ; CIAA sdr - Leggiamo il carattere attuale
2      bset.b  #6,$bfee01      ; CIAA cra - sp ($bfec01) output, in modo da
3                               ; abbassare la linea KDAT per confermare che
4                               ; abbiamo ricevuto il carattere.
5
6      st.b    $bfec01         ; $FF in $bfec01 - uè! ho ricevuto il dato!
7
8
9      ; Qua dobbiamo mettere una routine che aspetti 90 millisecondi perchè la
10     ; linea KDAT deve stare bassa abbastanza tempo per essere "capita" da tutti
11     ; i tipi di tastiere. Si possono, per esempio, aspettare 3 o 4 linee raster.
12
13     bclr.b  #6,$bfee01      ; CIAA cra - sp (bfec01) input nuovamente.
14

```

Leggendo la tastiera via hardware, occorre stare molto attenti alla temporizzazione che attende 90 millisecondi, per 2 motivi:

1. La routine di temporizzazione deve aspettare lo stesso tempo su tutti i processori, dal 68000 al 68060. Per questo potete usare il pennello elettronico, o anche un timer del CIA, ma **mai** fare un semplice loop DBRA eseguito molte volte, o una serie di NOP, perché su 68020+ a causa della cache sarà eseguito in un battibaleno.
2. Una volta che la nostra routine "aspetta" bene su tutti i 680x0, occorre anche considerare il fatto che non tutte le tastiere sono uguali! Per esempio per una tastiera potrebbero bastare 2 linee raster mentre per un'altra ne potrebbero occorrere 4! Infatti le tastiere contengono un chip che le controlla, e questo può essere diverso nei diversi modelli di Amiga.

Per esempio nell'A1200 la tastiera è "economica", infatti si differenzia dalle tastiere Amiga normali (Mitsumi in genere) per il fatto che non si può registrare più di una pressione alla volta. . . se si tiene premuto un tasto e nel contempo se ne preme un altro, rilasciando il primo non compare il secondo. La routine di wait che deve aspettare tra: `or.b #$40` o `bset.b #6, $bfee01` e `and.b #$bf` o `bclr.b #6, $bfee01` determina se il vostro programma leggerà la tastiera correttamente o si inchiederà alla pressione di un tasto su alcuni computer.

A tal proposito vediamo in che modo aspettare correttamente, usando il VBLANK:

```

1      ; Se non volete "sporcare" registri indirizzi:
2
3
4      moveq   #4-1,d0          ; we wait 4 rasterlines (3+random...!)
5  waitlines:
6      move.b  $DFF006,d1
7  stepline:
8      cmp.b   $DFF006,d1
9      beq.s   stepline
10     dbra    d0,waitlines
11
12
13     ; Se invece volete "sporcare" anche un registro indirizzi:
14
15
16     lea      $dff006,a0        ; VHPOSr
17     moveq    #4-1,d0          ; Numero di linee da aspettare = 4 (in pratica 3 più
18                               ; la frazione in cui siamo nel momento di inizio)

```

```

19 waitlines:
20     move.b    (a0),d1 ; $dff006 — linea verticale attuale in d1
21 stepline:
22     cmp.b     (a0),d1  ; siamo sempre alla stessa linea?
23     beq.s     stepline ; se si aspetta
24     dbra      d0,waitlines ; linea "aspettata", aspetta d0-1 linee
25

```

Caratteristiche del codice carattere trasmesso in \$bfec01

Abbiamo già detto che il codice trasmesso non è un codice ascii, ma una informazione sul tasto che è stato premuto. Questo anche perché a seconda delle diverse tastiere, in lingua inglese, italiana o altro, molti tasti hanno una lettera diversa stampata sopra. Ma se dico: il terzo tasto della seconda fila, non ci si può sbagliare. Comunque gli 8 bit (1 byte) che prendiamo dal \$bfec01 contengono 7 bit relativi all'identificatore del tasto, più un bit che stabilisce se il tasto è stato premuto o rilasciato. Infatti, l'identificatore del tasto viene inviato sia quando si preme che quando si rilascia, con la differenza che il bit più alto, l'ottavo, una volta è azzerato (rilasciato) o settato (premuto).

Come non bastasse, tutti i codici trasmessi sono ruotati di un bit a sinistra prima della loro trasmissione. L'ordine della trasmissione quindi è 6-5-4-3-2-1-0-7. Comunque basta usare l'istruzione "ROR.B #1,xxx" per riportare l'ordine 7-6-5-4-3-2-1-0. La trasmissione di un bit impiega 60 microsecondi, dunque l'intero byte che costituisce il carattere viene trasferito in 480 microsecondi, per cui si possono trasferire 17000 bit al secondo. Ma che ci importa? Nulla! Vediamo invece come si fa a riconoscere se il tasto premuto è la A, la B o qualche altro. Nell'hardware manual c'è una lista con un codice per tasto, dove la particolarità è che al codice \$01 corrisponde il tasto "1". Per ottenere questi codici occorre eseguire un NOT del byte, oltre che fare una rotazione dei bit con un ROR per riportare l'ordine 76543210. In pratica, quello che dobbiamo fare è:

```

1     move.b    $bfec01,d0 ; CIAA sdr (serial data register — connesso
2                          ; alla tastiera — contiene il byte inviato dal
3                          ; chip della tastiera) LEGGIAMO IL CHAR!
4     NOT.B     D0          ; aggiustiamo il valore invertendo i bit
5     ROR.B     #1,D0       ; e riportando la sequenza a 76543210.

```

Ora in d0 abbiamo il byte con la sequenza di bit 76543210 anziché 65432107, e in più tutti i bit sono invertiti, in modo da far partire il "conto" dal primo tasto in alto a sinistra (non ESC, ma quello accanto all'1). Ecco la sequenza dei codici con il relativo carattere (normale e shiftato, ma considerate che qua la tastiera descritta è quella USA). (TASTI PREMUTI)

cod.	\$00	; ' - ~
cod.	\$01	; 1 - !
cod.	\$02	; 2 - @
cod.	\$03	; 3 - #
cod.	\$04	; 4 - \$
cod.	\$05	; 5 - %
cod.	\$06	; 6 - ^
cod.	\$07	; 7 - &
cod.	\$08	; 8 - *
cod.	\$09	; 9 - (
cod.	\$0A	; 0 -)
cod.	\$0B	; - - _
cod.	\$0C	; = - +
cod.	\$0D	; \ -
cod.	\$0E	; << vuoto
cod.	\$0F	; 0 tastierino numerico
cod.	\$10	; q - Q
cod.	\$11	; w - W
cod.	\$12	; e - E

```

cod.  $13      ;r - R
cod.  $14      ;t - T
cod.  $15      ;y - Y
cod.  $16      ;u - U
cod.  $17      ;i - I
cod.  $18      ;o - O
cod.  $19      ;p - P
cod.  $1A      ;[ - {
cod.  $1B      ;] - }
cod.  $1c      ; << non usato
cod.  $1D      ;1  tastierino numerico
cod.  $1E      ;2  tastierino numerico
cod.  $1F      ;3  tastierino numerico
cod.  $20      ;a - A
cod.  $21      ;s - S
cod.  $22      ;d - D
cod.  $23      ;f - F
cod.  $24      ;g - G
cod.  $25      ;h - H
cod.  $26      ;j - J
cod.  $27      ;k - K
cod.  $28      ;l - L
cod.  $29      ;; - :
cod.  $2A      ;' - "
cod.  $2B      ;(solo in tast. internazionali) - vicino al return
cod.  $2c      ; << non usato
cod.  $2D      ;4  tastierino numerico
cod.  $2E      ;5  tastierino numerico
cod.  $2F      ;6  tastierino numerico
cod.  $30      ;< (shift sin. solo in tastiere internazionali)
cod.  $31      ;z - Z
cod.  $32      ;x - X
cod.  $33      ;c - C
cod.  $34      ;v - V
cod.  $35      ;b - B
cod.  $36      ;n - N
cod.  $37      ;m - M
cod.  $38      ; , - <
cod.  $39      ; . - >
cod.  $3A      ; / - ?
cod.  $3b      ; << non utilizzato
cod.  $3C      ; .  tastierino numerico
cod.  $3D      ;7  tastierino numerico
cod.  $3E      ;8  tastierino numerico
cod.  $3F      ;9  tastierino numerico
cod.  $40      ;space
cod.  $41      ;back space <-
cod.  $42      ;tab ->|
cod.  $43      ;return tastierino numerico (enter)
cod.  $44      ;return <- '
cod.  $45      ;esc
cod.  $46      ;del
cod.  $47      ; << non usato
cod.  $48      ; << non usato
cod.  $49      ; << non usato
cod.  $4A      ; -  tastierino numerico
cod.  $4b      ; <<
cod.  $4C      ;cursore alto  ^
cod.  $4D      ;cursore basso v
cod.  $4E      ;cursore destra  >
cod.  $4F      ;cursore sinistra <
cod.  $50      ;f1

```

```

cod.  $51      ;f2
cod.  $52      ;f3
cod.  $53      ;f4
cod.  $54      ;f5
cod.  $55      ;f6
cod.  $56      ;f7
cod.  $57      ;f8
cod.  $58      ;f9
cod.  $59      ;f10
cod.  $5A      ;( tastierino numerico
cod.  $5B      ;) tastierino numerico
cod.  $5C      ;/ tastierino numerico
cod.  $5D      ;* tastierino numerico
cod.  $5E      ;+ tastierino numerico
cod.  $5F      ;help
cod.  $60      ;lshift (sinistro)
cod.  $61      ;rshift (destra)
cod.  $62      ;caps lock
cod.  $63      ;ctrl
cod.  $64      ;lalt (sinistro)
cod.  $65      ;ralt (destra)
cod.  $66      ;lamiga (sinistro)
cod.  $67      ;ramiga (destra)

```

Come vedete, la sequenza rispetta grossomodo l'ordine dei tasti, partendo dalla fila con 1,2,3,4,5... poi la fila sotto con q,w,e,r,t,y... eccetera. Questi codici si riferiscono ai tasti **premuti**, dove il bit 7, quello che determina se un tasto è stato premuto o rilasciato, è a zero. Infatti abbiamo eseguito un NOT sul byte, che ha invertito pure il bit 8: se un tasto è **premuto**, ora il bit 8=0, se un tasto è rilasciato, il bit 8=1. Se il tasto è rilasciato, il bit 7 (l'ottavo) va considerato come settato, per cui la tabella di prima diventerebbe:

```

cod.  $80      ;' - ~
cod.  $81      ;1 - !
cod.  $82      ;2 - @
...

```

Attenzione al fatto che gli Amiga600 non hanno il tastierino numerico, per cui se usate i tasti del tastierino su tali computer non c'è modo di premerli! Vi consiglio quindi di evitare i tasti del tastierino. Alla luce di queste considerazioni, possiamo vedere come sarà l'interrupt di livello 2 (\$68), che ci permetterà di salvare nella variabile "ActualKey" il codice dei tasti premuti:

11.12 Routine di interrupt \$68 (livello 2) - gestione tastiera

```

1 ;03  PORTS  2 ($68) Input/Output Porte e timers, connesso alla linea INT2
2
3 MioInt68KeyB:  ; $68
4   movem.l d0-d1/a0,-(sp) ; salva i registri usati nello stack
5   lea     $dff000,a0      ; reg. custom per offset
6
7   MOVE.B  $BFED01,D0      ; C'è icr - in d0 (leggendo l'icr causiamo
8                           ; anche il suo azzeramento, per cui l'int è
9                           ; "disdetto" come in intreq).
10  BTST.l  #7,D0           ; bit IR, (interrupt cia autorizzato), azzerato?
11  BEQ.s   NonKey          ; se sì, esci
12  BTST.l  #3,D0           ; bit SP, (interrupt della tastiera), azzerato?
13  BEQ.s   NonKey          ; se sì, esci
14
15  MOVE.W  $1C(A0),D0      ; INTENAR in d0
16  BTST.l  #14,D0          ; Bit Master di abilitazione azzerato?
17  BEQ.s   NonKey          ; Se sì, interrupt non attivi!

```

```

18      AND.W    $1E(A0),D0      ; INREQR — in d1 rimangono settati solo i bit
19                                ; che sono settati sia in INTENA che in INTREQ
20                                ; in modo da essere sicuri che l'interrupt
21                                ; avvenuto fosse abilitato.
22      btst.l   #3,d0           ; INTREQR — PORTS?
23      beq.w    NonKey          ; Se no, allora esci!
24
25      ; Dopo i controlli, se siamo qua significa che dobbiamo prendere il carattere!
26
27      moveq    #0,d0
28      move.b   $bfec01,d0      ; CIAA sdr (serial data register — connesso
29                                ; alla tastiera — contiene il byte inviato dal
30                                ; chip della tastiera) LEGGIAMO IL CHAR!
31
32      ; abbiamo il char in d0, lo "lavoriamo"...
33
34      NOT.B    D0              ; aggiustiamo il valore invertendo i bit
35      ROR.B    #1,D0           ; e riportando la sequenza a 76543210.
36      move.b   d0,ActualKey    ; salviamo il carattere
37
38      ; Ora dobbiamo comunicare alla tastiera che abbiamo preso il dato!
39
40      bset.b   #6,$bfec01      ; CIAA cra — sp ($bfec01) output, in modo da
41                                ; abbassare la linea KDAT per confermare che
42                                ; abbiamo ricevuto il carattere.
43
44      st.b     $bfec01         ; $FF in $bfec01 — uè! ho ricevuto il dato!
45
46      ; Qua dobbiamo mettere una routine che aspetti 90 microsecondi perchè la
47      ; linea KDAT deve stare bassa abbastanza tempo per essere "capita" da tutti
48      ; i tipi di tastiere. Si possono, per esempio, aspettare 3 o 4 linee raster.
49
50      moveq    #4-1,d0 ; Numero di linee da aspettare = 4 (in pratica 3 più
51                                ; la frazione in cui siamo nel momento di inizio)
52      waitlines:
53      move.b   6(a0),d1        ; $dff006 — linea verticale attuale in d1
54      stepline:
55      cmp.b    6(a0),d1        ; siamo sempre alla stessa linea?
56      beq.s    stepline        ; se si aspetta
57      dbra     d0,waitlines     ; linea "aspettata", aspetta d0-1 linee
58
59      ; Ora che abbiamo atteso, possiamo riportare $bfec01 in modo input...
60
61      bclr.b   #6,$bfec01      ; CIAA cra — sp (bfec01) input nuovamente.
62
63      NonKey: ; 3210
64      move.w   #%1000,$9c(a0)  ; INTREQ toglie richiesta, int eseguito!
65      movem.l  (sp)+,d0-d1/a0  ; ripristina i registri dallo stack
66      rte
67
68

```

Non dovrete aver notato niente di nuovo, è solo una “sintesi” delle cose che abbiamo già spiegato. Poi in definitiva sono solo poche linee e usiamo solo i registri d0 ed a0, non è mica una routine *complicata*!. L’unica cosa che dovete ricordarvi è di mettere questo interrupt al vettore \$68+VBR, e di abilitarlo, settando il bit 3 di INTENA (\$dff09a). Per esempio, se state usando un’interrupt di livello 3 (\$6c) per suonare la musica, usando il solo VERTB (bit 5), potete scrivere:

```

1      ; 5432109876543210
2      move.w   #%1100000000101000,$9a(a5) ; INTENA — abilito solo VERTB
3                                          ; del livello 3 e il livello2

```

O in altri termini `move.w #$c028,$dff09a`. Possiamo vedere il corretto uso di questo interrupt in `Lezione11m1.s`.

In questo listato il codice della tastiera viene immesso nel `COLOR0` per far “vedere” l’effettivo funzionamento della routine stessa. Per uscire naturalmente occorre premere un tasto: la barra di spazio.

Per comodità, in `Lezione11m2.s` è inclusa una routine rudimentale di conversione dei codici di tastiera in ASCII, che può servire se volete fare in modo che si possa stampare quello che viene scritto con la tastiera, per esempio se volete farvi una utility, o semplicemente scrivere il vostro nome nell'high score del vostro gioco.

11.13 I timers del CIAA e del CIAB

Questi timers in realtà sono usati pochissimo nei giochi, e nelle demo quasi mai, giusto in certe (complicate) routines che suonano la musica, che comunque basta includere e suonano per i fatti loro. Tra l'altro questi timers sono usati anche dal sistema operativo, per cui usandoli si può rischiare di far impazzire tutto all'uscita del nostro programma. Inoltre, aspettando le linee raster con il `$dff006` si possono fare tutte le attese che servono, senza rischiare queste eventualità. Per questo, nella lezione sono presenti solo un paio di listati che usano i timer, come esempio. In listati di lezioni più avanzate avremo modo di trovare applicazioni di questi timer, e li vedremo caso per caso.

`Lezione11n1.s` – Uso del timer A del CIAA o CIAB

`Lezione11n1b.s` – Uso del timer B del CIAA o CIAB

`Lezione11n2.s` – Uso del timer TOD (Time of day)

Nell'usare i timer del CIA considerare che il sistema operativo usa questi per i seguenti scopi: (meglio usare il CIAB!)

CIAA, timer A	Utilizzato per l'interfacciamento con la tastiera
! CIAA, timer B	Utilizzato dall'exec per lo scambio dei task ecc.
CIAA, TOD	timer a 50/60 Hz utilizzato dal <code>Timer.device</code>
CIAB, timer A	Non utilizzato, disponibile per i programmi
CIAB, timer B	Non utilizzato, disponibile per i programmi
CIAB, TOD	Utilizzato dalla <code>graphics.library</code> per seguire le posizioni del pennello elettronico.

Se dovete usare timer che servono anche al sistema operativo, fatelo solo se avete disabilitato multitasking e interrupt di sistema, se avete cioè preso il controllo completo del sistema. Mai il CIAA, timer B!

11.14 Il caricamento di files con la `dos.library`

Per concludere questa lezione piena di perfezionamenti e argomenti vari, non c'è miglior argomento del **caricamento dei dati**. Se volete programmare qualcosa di "grosso", dal punto di vista della mole di disegni, musiche e dati vari, non si può semplicemente includere tutto con l'`incbin` e salvare il mega eseguibile con "WO", perchè il file verrebbe troppo grande per poter essere caricato in memoria. Supponiamo, per esempio, di voler fare uno slideshow, ossia un programma che mostri una serie di figure una dopo l'altra, e che queste figure siano ben 30, lunghe 100Kb l'una. Vengono 3MB di figure. Non potendo fare una serie di 30 `INCBIN` per salvare un file di 3MB e passa, non ci resta che trovare un modo per "caricarne" una alla volta. Ma quale modo usare? Ce ne sono 2 principalmente:

Caricamento AutoBoot dalle tracce del dischetto ossia una modalità non compatibile col DOS, infatti potrete notare che molti dischi di giochi, se inseriti nel drive una volta caricato il workbench, non sono leggibili con comandi come "DIR", e risultano NON DOS, o ERRATI... insomma sembrano dischi marci! Se sottoposti a copia, tramite copiatori come XCOPY o DCOPY, alcuni di questi giochi non dos appaiono a tracce "ROSSE", ossia non riconoscibili nemmeno dal copiatore, mentre altri nonostante risultino illeggibili dal dos appaiono come "SANI", ossia a tracce verdi. Devo precisare che i giochi CRACKATI (sprotetti e distribuiti dai pirati) sono tutti del secondo tipo, ossia a tracce VERDI, infatti la sprotezione spesso sta nel trasformare le tracce incopiabili in tracce copiabili, ma spesso rimangono illeggibili dal dos. Le *TRACKMO* sono la gran parte delle demo, e sono a tracce "copiabili", ma non leggibili via dos. Una caratteristica è quella che occorre scrivere codice per indirizzi assoluti, non rilocabile, per cui solitamente viene usato solo il primo mega di CHIP RAM, o per al200 i primi 2 mega, ed eventuali espansioni di FAST RAM non sono utilizzate, a parte quelli che usano COMPLESSI LOADER CON RILOCATORI che somigliano a mini sistemi operativi, che però spesso fanno cilecca sul 68040 per le eccessive "esuberanze" di programmazione. Questo sistema ha il "pregio" di essere leggermente più veloce, sui floppy disk, del dos normale, ma lo svantaggio di non poter installare su Hard-Disk il programma, nè poterlo convertire per CD32 eccetera.

Caricamento "legale" usando la dos.library ossia un modo molto simile a quello usato da un qualsiasi programma che usa il sistema operativo, compilato con un qualsiasi linguaggio come il C, l'AMOS eccetera. In realtà, mantenendo una nostra copperlist e operando sui registri hardware facciamo un sistema "ibrido", ossia usiamo la dos.library in uno stato "particolare" con la nostra copperlist e i nostri interrupt. Una caratteristica dei programmi che usano questo sistema è che il sistema operativo deve essere lasciato "intatto" e il codice deve essere totalmente rilocabile, (potendo accedere all'eventuale FAST RAM). Questo sistema ha il pregio di poter essere usato su Hard Disk, CD-ROM e qualsiasi drive supportato dal sistema, anche future periferiche.

Benché il primo sistema possa sembrare più accattivante per uno che vuole programmare a livello hardware, in realtà si tratta di un modo **vecchio, spesso incompatibile** e limitante per l'impossibilità di installare su HD il programma (o demo che sia). Finché parliamo di una demo o di un gioco per Amiga500 che sia ad 1 solo disco, forse è accettabile l'opzione del trackloader, ma da 2 dischi in sù il sistema porta solo a far arrabbiare i possessori di HardDisk, che saranno sempre di più. Un gioco installato su HD caricherà sempre più velocemente di uno da disco con il loader più turbo possibile.

Poi c'è il discorso della FAST RAM: per poterla utilizzare con un trackloader occorrerebbe fare un mini sistema operativo che trovi dove si trova e rilochi il codice al giusto indirizzo. Non ho intenzione di proporvi il listato di uno di questi loader + rilocatori, per non indurvi sulla cattiva strada. Pensate alla soddisfazione di poter convertire il vostro gioco per il CD32, o di vederlo funzionare sul 68060 e su qualsiasi HardDisk, e invece alla delusione di veder fallire il rilocatore "a mano" o di notare che il programma non sfrutta la fast ram... vi ho convinto?

C'è anche un'altra cosa: sarebbe bene fare uso del comando `assign` per le nostre produzioni che devono caricare files. Ad esempio, se facciamo un gioco ad un disco, dando il nome "Cane" al disco, potremo caricare i file con `Cane:file1`, `Cane:file2`, `Cane2/oggetti/ogg1`, e così via. Nel caso si voglia installare su Hard Disk, basterà fare una directory, copiarci il contenuto del disco, e aggiungere alla startup-sequence:

```
assign Cane: dh0:giococane      ; ad esempio...
```

Se il gioco è a più dischi, basterà copiare tutti i dischi nella directory e fare l'assign di ogni disco:

```
assign Cane1: dh0:giococane
assign Cane2: dh0:giococane
assign Cane3: dh0:giococane
```

Per aggiungere “automaticamente” alla startup-sequence o all'user-startup gli assign necessari, durante l'installazione del gioco, si possono usare le opzioni dell'installer commodore o altri sistemi, ma questo esula dal corso.

Bene, vediamo allora come caricare un file “path:xx” in una destinazione in memoria. Ci sono vari modi. Il più semplice è questo:

```
1 CaricaFile:
2     move.l #filename,d1      ; indirizzo con stringa "file name + path"
3     MOVE.L #$3ED,D2          ; AccessMode: MODE_OLDFILE - File che esiste
4                               ; già, e che quindi potremo leggere.
5     MOVE.L DosBase(PC),A6
6     JSR -$1E(A6)             ; LVOOpen - "Apri" il file
7     MOVE.L D0,FileHandle     ; Salva il suo handle
8     BEQ.S ErrorOpen         ; Se d0 = 0 allora c'è un errore!
9
10    MOVE.L D0,D1              ; FileHandle in d1 per il Read
11    MOVE.L #buffer,D2         ; Indirizzo Destinazione in d2
12    MOVE.L #42240,D3          ; Lunghezza del file (ESATTA!)
13    MOVE.L DosBase(PC),A6
14    JSR -$2A(A6)             ; LVORead - leggi il file e copialo nel buffer
15
16    MOVE.L FileHandle(pc),D1   ; FileHandle in d1
17    MOVE.L DosBase(PC),A6
18    JSR -$24(A6)             ; LVOClose - chiudi il file.
19 ErrorOpen:
20     rts
21
22
23 FileHandle:
24     dc.l 0
25
26 ; Stringa di testo, da terminare con uno 0, a cui dovrà puntare d1 prima di
27 ; fare l'OPEN della dos.lib. Conviene mettere l'intero path.
28
29 Filename:
30     dc.b "Assembler3:sorgenti7/amiet.raw",0      ; path+nomefile
31     even
```

Questo è perfetto se si conosce la lunghezza esatta del file da caricare. Trattandosi del nostro programma, si suppone che si sappia quando sono lunghi i nostri file dati! Vediamo un esempio in Lezione11o1.s.

Però a noi interessa maggiormente caricare un file mentre stiamo visualizzando una nostra copperlist, e magari suonando una musica in interrupt. Come si concilia un caricamento “legale” con un sistema operativo del tutto disabilitato? Intanto consideriamo il fatto che gli interrupt di sistema devono essere tutti riattivati, mentre la copperlist di sistema non serve, e possiamo tenere la nostra. Allora come continuare a suonare la musica, o fare qualcos'altro, mentre sta avvenendo un caricamento? I sistemi sono molteplici. Potremmo aggiungere delle nostre routines in modo “legale” all'interrupt di sistema, con un AddIntServer(). Oppure potremmo eseguire un nostro interrupt, che poi salti ad eseguire quello di sistema.

Un modo un pò meno rispettoso, che però funziona e preferisco usare, anche perché l'ho visto usare nei giochi per CD32. In pratica ecco cosa dobbiamo fare: ripristinare i vecchi interrupt e il vecchio stato DMA/INTENA, riabilitare il multitasking eccetera, come facciamo all'uscita, ma lasciare la nostra copperlist e “infilare” il nostro interrupt \$6c in più a quello di sistema. Poi caricare il file, e **attendere qualche secondo per essere sicuri che la spia del drive o dell'hard disk o del cd-rom si sia spenta**, poi richiudere tutto e tornare a battere nel metallo senza pietà.

Insomma, prima e dopo il caricamento occorre riabilitare e ridisabilitare il sistema operativo, lasciando la nostra copperlist. L'unico particolare è l'interrupt: come facciamo ad eseguire il nostro, poi saltare a quello vecchio? Voglio proporvi un sistema da veri contrabbandieri, che però funziona, a patto che si chiami la routine `ClearMyCache`, che azzerà l'istruzione cache del processore (68020+). Infatti useremo per la prima (e ultima) volta, codice **automodificante**! Non andrebbe mai usato, ma voglio farvi vedere uno dei pochi casi in cui funziona ed è utile, giusto per informazione. Avete presente che ogni istruzione quando viene assemblata diventa una serie di valori esadecimali? ad esempio RTS diventa \$4e75, e così via. Noi dobbiamo *jumpare* al vecchio interrupt, dopo aver eseguito il nostro. Dunque, un `JMP $12345`, ad esempio, diventa \$49f900012345, ossia \$4ef9, seguito dall'indirizzo a cui saltare, che è una long:

```
1      dc.w    $4ef9          ; val esadecimale di JMP
2 Crappyint:
3      dc.l    0             ; Indirizzo dove Jumpare, da AUTOMODIFICARE...
```

Ora, se mettessimo in `CrappyInt` l'indirizzo dell'interrupt di sistema con:

```
1      move.l  oldint6c(PC),crappyint ; Per DOS LOAD – salteremo all'oldint
```

Avremmo il `JMP oldint6c` che cercavamo... allora l'interrupt finale è:

```
1 *****
2 ; Routine di interrupt da mettere durante il caricamento. Le routines che
3 ; saranno messe in questo interrupt saranno eseguite anche durante il
4 ; caricamento, sia che avvenga da floppy disk, da Hard Disk, o CD ROM.
5 ; DA NOTARE CHE STIAMO USANDO L'INTERRUPT COPER, E NON QUELLO VBLANK,
6 ; QUESTO PERCHÈ DURANTE IL CARICAMENTO DA DISCO, SPECIALMENTE SOTTO KICK 1.3,
7 ; L'INTERRUPT VERTB NON È STABILE, tanto che la musica avrebbe dei sobbalzi.
8 ; Invece, se mettiamo un "$9c,$8010" nella nostra copperlist, siamo sicuri
9 ; che questa routine sarà eseguita una volta sola per fotogramma.
10 *****
11
12 myint6cLoad:
13     btst.b  #4,$dff01f      ; INTREQR – il bit 4, COPER, è azzerato?
14     beq.s   nointL         ; Se sì, non è un "vero" int COPER!
15     move.w  #%10000,$dff09c ; Se no, è la volta buona, togliamo il req!
16     movem.l d0-d7/a0-a6,-(SP)
17     bsr.w   mt_music       ; Suona la musica
18     movem.l (SP)+,d0-d7/a0-a6
19 nointL:
20     dc.w    $4ef9          ; val esadecimale di JMP
21 Crappyint:
22     dc.l    0             ; Indirizzo dove Jumpare, da AUTOMODIFICARE...
23                               ; ATTENZIONE: il codice automodificante non
24                               ; andrebbe usato. Comunque se si chiama un
25                               ; ClearMyCache prima e dopo, funziona!
```

Come vedete, basta puntare in \$6c+VBR questo interrupt per eseguire `mt_music` e il vecchio interrupt di sistema, ottenendo la musica+il caricamento in contemporanea. Vediamo un esempio in `Lezione11o2.s`.

A questo punto vi potete immaginare a cosa può servire la routine che blocca l'input di intuition: quando carichiamo un file, riabilitiamo multitasking e interrupts di sistema quindi, anche se viene visualizzata la nostra copperlist, il workbench funziona perfettamente, tanto che se durante un caricamento si muove "alla cieca" il mouse, si può anche azionare qualche menù o clickare qualche icona, o dare dei comandi da tastiera per il cli. Pensate ad un videogiocatore che ha il vizio di muovere e pigiare il mouse durante i caricamenti per non innervosirsi: all'uscita del gioco potrebbe accorgersi di aver clickato l'icona dell'Hard Disk e aver scelto per caso l'opzione format dal menù del WB che non vedeva, e magari premendo accidentalmente la tastiera potrebbe avergli dato un nome osceno, anche. Quindi, anche se una volta disabilitato il sistema operativo chiamare la routine `InputOff` non è indispensabile, nel caso si carichino files o si facciano altre operazioni è bene che non sia possibile fare danni!

Per terminare la lezione, vediamo come fare a caricare un file di cui non sappiamo a priori la lunghezza, prendendo anche l'occasione per spiegare le routines di AllocMem e FreeMem. Per sapere la lunghezza di un file, basta eseguire una apposita funzione, detta Examine, a patto che si abbia lockato il file. Ciò non è molto difficile, basta fare qualche JSR in più. Da notare che Examine non fa altro che riempire un buffer lungo \$104 bytes con i vari dati del file, ecco un esempio:

```

1      cnop      0,4      ; Attenzione! Il FileInfoBlock deve essere allineato
2                                ; a longword, non basta che sia ad un indirizzo pari!
3
4  fib:
5      dcb.b     $104,0   ; Struttura FileInfoBlock: offsets.
6                                ; 0 = fib_DiskKey
7                                ; 4 = fib_DirEntryType (<0 = file , >0 = directory)
8                                ; 8 = FileName (max 30 caratteri, terminato con 0)
9                                ; $74 = fib_Protection , $78 = fib_EntryType
10                               ; $7c = fib_Size , $80 = fib_NumBlocks
11                               ; $84 = fib_Date (3 longs: Days, Minute, Tick)
12                               ; $90 = comment (termina con uno 0)

```

Come vedete, all'offset \$7c troviamo la lunghezza. Le altre cose non ci interessano... che ce ne facciamo della data o del commento? Comunque, dato che dobbiamo allocare la memoria per il file, la allocheremo anche per il FileInfoBlock, in modo da risparmiarci questo dcb.b \$104,0. Una volta saputa la lunghezza del file, dovremo creare un buffer in memoria lungo quanto il file, per caricarlo dentro. Questo si fa con AllocMem, che richiede in entrata il numero di bytes da allocare e il tipo di memoria, se chip o no, in modo analogo alle section con _C o no. A differenza delle sections, però, alla fine del programma dobbiamo liberare manualmente tutti i blocchi allocati tramite la funzione FreeMem.

11.15 AllocMem

Questa routine di Exec serve per richiedere un blocco di memoria da usare per i nostri scopi. Basta indicare il tipo di memoria richiesto (in pratica se deve essere CHIP ram o no), e la lunghezza in bytes di tale blocco. La routine **alloca** il pezzo di ram libera per il nostro uso esclusivo, in quanto ne prendiamo possesso, dato che il sistema operativo non scriverà più in quel pezzo di memoria, fino a che non glielo "rendiamo", con FreeMem. Infatti il sistema multitasking Amiga funziona con questo sistema: ogni programma richiede quanta memoria gli serve tramite AllocMem, il sistema operativo gli riserva dei pezzi di ram libera, poi ad un altro programma che carica in multitasking saranno allocate altre parti di ram libera. Per ora abbiamo usato le SECTION BSS per gli spazi di memoria azzerata che ci servivano, in quanto sapevamo la loro grandezza in partenza. Ed è meglio usare le BSS per i bitplanes o i buffer di grandezza certa, per varie ragioni, come il non dover chiamare routines e il poter mettere le label qua e là nel buffer, a differenza della mem allocata a cui dovremmo accedere per forza tramite offsets dall'inizio del blocco. Nel nostro listato, carichiamo in memoria un file di cui non sappiamo la lunghezza, per cui qua è obbligatorio usare l'AllocMem, dopo che abbiamo saputo quanto spazio occuperà il file. Vediamo in dettaglio la funzione:

```

1      move.l    Grandezza(PC),d0 ; Grandezza del blocco in bytes
2      move.l    TypeOfMem(PC),d1 ; Tipo di Memoria (chip,public...)
3      move.l    4,w,a6
4      jsr      -$c6(a6)           ; Allocmem
5      move.l    d0,FileBuffer     ; Indirizzo inizio del blocco di mem. allocata
6      beq.s     FineMem           ; d0=0? Allora errore!
7      ...

```

Se non è indispensabile allocare chip mem (ossia se nel buffer allocato non ci andrà né grafica né suono), allocate sempre MEMF_PUBLIC, che significa: *memoria fast se c'è, o se proprio non c'è*

allora chip. Ricordo per l'ennesima volta che è bene risparmiare chip mem, e che la FAST memory è più veloce della chip.

All'uscita, in D0 ci sarà l'indirizzo del blocco di memoria richiesto, che tra l'altro sarà allineato a long word (ossia allineato a 32 bit). Se invece D0 = 0, non è stato possibile allocare un blocco di questo tipo! Testate sempre questa cosa, o in caso di fine mem copiereste tutto in \$0!!!

Possiamo anche richiedere che la memoria richiesta sia azzerata, basta settare il bit MEMF_CLEAR, il 16 (\$10000). Ecco i parametri più utili da mettere in d1, per richiedere i vari tipi di memoria:

```
MEMF_CHIP      =      2      ; Richiesta Chip Ram
MEMF_FAST      =      4      ; Richiesta Fast Ram (non usatelo)
MEMF_PUBLIC    =      1      ; Richiesta Fast, ma se non c'è va bene chip!
```

E, naturalmente, se volete che i blocchi siano azzerati:

```
CHIP           =      $10002
FAST           =      $10004 ; non usatelo...
PUBLIC        =      $10001
```

Vi sconsiglio di richiedere MEMF_FAST, perché la fast non è presente su tutte le macchine. Usate sempre MEMF_PUBLIC, a parte quando la memoria allocata deve essere usata come bitplane, copperlist o audio, ossia MEMF_CHIP. Da notare che la lunghezza del blocco che immettiamo sarà arrotondata dal sistema operativo ad un multiplo dei blocchi (chunk) del sistema. Questo non è un problema per noi, infatti se immettiamo 39, probabilmente alloca 40, ma i 39 richiesti ci sono tutti, quindi a noi non interessa. All'uscita dal programma ricordatevi di liberare il blocco di memoria!

11.16 FreeMem

Questa è la routine da chiamare per liberare i blocchi di memoria allocata. è richiesto l'indirizzo del blocco in A1, e la lunghezza in bytes in D0. ATTENZIONE: Se si tenta di liberare un blocco che non era stato allocato veramente, causerete un casino pazzesco con Guru Meditation/soft Failure! Ecco come liberare il blocco di memoria di prima:

```
1      move.l  Grandezza(PC),d0 ; Grandezza del blocco in bytes
2      move.l  FileBuffer(PC),a1 ; Indirizzo del blocco di mem. allocata
3      move.l  4.w,a6
4      jsr    -$d2(a6)          ; FreeMem
```

A questo punto possiamo anche vedere il programma: Lezione11o3.s.

LEZIONE 12 - LA COMPATIBILITÀ DEL CODICE

In questa lezione leggerete la tecniche per ottenere la *compatibilità del codice*, una cosa importantissima: pensate quanto sia importante che il gioco o la demo che programmate funzioni su tutti i modelli Amiga!!!

Questo non è assolutamente difficile, anche se è risaputo che moltissimi giochi e demo vecchi non funzionano su kick 2.0, 68020, a1200, o addirittura esistono delle cose che funzionano solo su A500 inespanso 1.3, basta mettere la fast ram, o il kickstart nuovo che non funzionano più. Tutti questi problemi derivano da poche cause, le stesse stupide cause, infatti il 99% del codice di un demo o di un gioco che va solo su a500 1.3 funzionerebbe su tutti gli Amiga, se non fosse per quelle 2 o 3 linee di codice “sporco” che fanno inchiodare tutto, generalmente al boot.

Personalmente ho sempre cercato di capire tutti questi BUG, ossia errori di programmazione visibili soltanto su macchine superiori al 500 1.3, e molto spesso sono riuscito a *fixare*, ossia a “riparare” dei giochi o delle demo che non funzionavano, semplicemente disassemblando il codice e modificando gli errori ricorrenti. In questo modo ho unito l’utile al dilettevole: da una parte ho fatto funzionare ad amici con A1200 o altri computer il gioco o la demo che tanto piacevano quando avevano sempre il 500 1.3, prima di venderlo, dall’altra mi sono fatto una discreta cultura sulle cause degli “inchiodamenti” con guru spettacolare, e ho constatato che sono sempre i soliti pochi “vizietti” di programmazione, che elencherò.

Per colpa di questi vizietti la maggior parte del software scritto in assembler diretto sull’hardware Amiga si è guadagnato la fama di essere incompatibile e poco sicuro, per cui l’assembler stesso è stato colpevolizzato di essere un linguaggio insicuro, specialmente quello “hardware direct”, il “metalbashing”.

Tutti questi problemi invece possono essere facilmente evitati, basta *non fare* certe cose, e sicuramente il gioco/demo girerà su tutti gli Amiga. Infatti tutti i listati di questo corso, per esempio, funzionano sia su amiga 500 1.3 che su amiga 4000/040, e naturalmente vanno anche su tutti gli altri computer intermedi, con qualsiasi configurazione.

Naturalmente non posso garantire la compatibilità con ipotetici modelli di amiga con RISC o chipset AAA, ma in tal caso non funzionerebbe NEMMENO UN GIOCO, e tantomeno programmi come il protracker. Spero infatti che sia mantenuta la serie 680x0 e la compatibilità verso il basso con l’ECS (almeno), o ci troveremmo con dei PC, chiamati “Amiga”, ma non MSDOS compatibili.

Farei piuttosto dei computer basati su 68060 a 150Mhz, che non ha niente da invidiare ad un RISC, e un “local bus” della chip ram, ossia una velocizzazione dell’accesso a questo tipo di memoria, che sui modelli attuali è troppo lenta (maledetti ingegneri C= dell’ultim’ora).

Ho deciso di trattare questo argomento solo ora, anche perché per poter fare degli errori è necessario almeno saper programmare, dunque non era logico mettere questa lezione prima di aver spiegato le basi della programmazione. Dopo questa lezione forse riuscirete a far funzionare il vostro vecchio gioco incompatibile!

Nel corso avete visto **come bisogna programmare**, con tutti i procedimenti giusti, per cui ignorate le cavolate che venivano fatte anni fa. Ecco una lista degli errori “alla paperissima” che ho trovato in giro per i programmi che non funzionano su tutte le macchine: (test su A4000/040)

12.1 Errori riguardanti COPPERList e Blitter

1. Tra gli errori “rimediabili” troviamo il meno grave, che poi non è un errore nelle vecchie produzioni, dato che non potevano sapere dell’AGA: è quello di “dimenticarsi” di resettare l’AGA con queste 3 istruzioni dopo aver puntato la copperlist: (Non prima!)

```

1      lea      $dff000,a5      ; Indirizzo CUSTOM di base in A5 per offsets
2      move.l   #copper,$80(a5) ; COP1LC - Punta la copperlist
3      move.w   d0,$88(a5)     ; COPJMP1 - fai partire la copperlist
4
5      ;      disabilitiamo l'AGA:
6
7      move.w   #0,$1fc(a5)     ; reset sprites wide and DISABLE 64 bit burst
8      MOVE.W   #$c00,$106(A5) ; reset AGA palette , sprite resolution
9                        ; and double playfield palette
10     MOVE.W   #$11,$10c(A5)   ; reset AGA sprite palette

```

A questo errore si può rimediare dal boot del computer premendo entrambi i tasti del mouse, e selezionando l’emulazione del vecchio chipset. D’altronde i problemi di copper non finiscono qua, infatti basta dimenticarsi di definire uno qualsiasi dei registri COPPER, che l’errore si affaccia! Infatti ho trovato molte copperlist di vecchi demo/giochi che non definivano i moduli, per cui RIMANGONO I VALORI della copperlist di sistema che non si sa mai come possono essere. Infatti l’errore più comune è quello di non mettere i moduli (\$108 e \$10a), dando per scontato che siano azzerati. QUESTO ERA VERO PER IL KICKSTART 1.3!!! MA DAL 2.0 IL MODULO NON È ZERO!!! Dunque le intro/demo/giochi si vedono a “strisciate”, a meno che non si carichi il kickstart 1.3. Lo stesso vale per DiwStart/DiwStop eccetera. Ricordatevi SEMPRE di mettere nella copperlist tutti i registri, anche se sono azzerati, per evitare che rimangano i valori incerti del sistema operativo!!!

```

1      dc.w     $108,0          ; Bpl1Mod
2      dc.w     $10a,0          ; Bpl2Mod
3      dc.w     $8e,$2c81       ; DiwStrt
4      dc.w     $90,$2cc1       ; DiwStop
5      dc.w     $92,$38         ; DdfStart
6      dc.w     $94,$d0         ; DdfStop
7      dc.w     $102,0          ; BplCon1
8      dc.w     $104,0          ; BplCon2

```

Ho trovato anche altri errori balordi spulciando per le vecchie copperlist. Alcuni anziché NON SETTARE dei registri, misteriosamente NE SETTAVANO TROPPI!!! Infatti **non bisogna mai accedere ad un registro sconosciuto, o non ancora utilizzato, né bisogna settare o azzerare bit riservati o senza funzione in registri conosciuti**, in questo modo si rischia di attivare strane funzioni in chipset futuri. Per ora l’evoluzione più grossa la abbiamo avuta da ECS ad AGA, e sono più di quanto mi aspettassi gli errori di “settaggio alla cieca”. Per esempio, ho trovato questa stranezza nella copperlist di una vecchissima intro *Ackerlight*:


```

1      ....
2      dc.w    $100,$5000      ; BPLCON0
3      dc.w    $0092,$30      ; DDFSTRT
4      ;————> dc.w    $106,$FE5      ; Perché hanno aggiunto un registro a quel
5                                     ; tempo inesistente? Sull'AGA falsa la palette
6      dc.w    $102,$CC      ; BPLCON1
7      dc.w    $108,$A8      ; BPL1MOD
8      dc.w    $10A,$A8      ; BPL2MOD
9      ....

```

Questo errore è passato inosservato fino a che non sono usciti gli AGA, e badate bene che questo errore è ASSOLUTAMENTE INELIMINABILE, infatti, prima di disassemblare il materiale che non funziona, faccio tutte le prove possibili per intuire il problema: tolgo la fastmemory, carico il kick 1.3, disabilito le cache, la MMU, azzero il VBR eccetera. . . Questo errore si presentava comunque: i colori erano sbagliati, tutto il resto funzionava. Infatti l'unico modo per correggere questi errori è di scovarli e di toglierli nel codice. Ho sostituito quel \$106,\$fe5 con un \$92,\$30, ossia ho replicato la linea precedente, e tutto ha funzionato a meraviglia. Forse il coder non si è nemmeno accorto di quel move, può essere un errore di battitura, forse voleva scrivere \$108 e non \$106, chissà, ma state attenti a lavorare solo su registri o bit conosciuti, o avrete la delusione di vedere la vostra produzione non funzionare sull'Amiga 9000 dei vostri nipoti, e questo per una sola, stupida linea di copperlist.

State attenti anche alle strutture degli sprite, dato che già da OCS a ECS ci sono dei bit in più nel quarto byte di controllo: tanti sprite che appaiono “sporchi”, o allungati fino alla fine dello schermo sono in quello stato SOLO su macchine ECS/AGA, e non su vecchi a500/a2000, perché la routine di gestione dello sprite anziché azzerare quel bit lo lasciava settato. I progettisti consigliano di lasciare AZZERATI i bit non usati dei registri conosciuti, e di NON ACCEDERE assolutamente a quelli non ancora conosciuti. Vi consiglio vivamente di non dimenticarvelo.

2. Non usate l'istruzione CLR su registri \$dffXXX, perché tale istruzione ha un comportamento diverso su processori 68000 e 68020/30/40, infatti su 68000 causa una lettura ed una scrittura, cioè 2 accessi, mentre su 68020/30/40 causa un solo accesso. Per evitare risultati diversi su processori diversi, ricordatevi di accedere in altro modo ai registri di tipo STROBE (COPJMP1 = \$dff080, COPJMP2 = \$dff088 ecc.). Un modo può essere un MOVE.W d0,\$dff080. Ho riscontrato problemi anche quando viene utilizzato il CLR su altri registri \$dffXXX, per esempio \$dff064 (BLTAMOD)

```

1  Esempio1:
2      MOVE.W  #0,$DFF088      ; mai fare CLR.W $dff088!
3  oppure:
4      MOVE.W  d0,$dff088
5      ...
6
7  Esempio2:
8      MOVE.W  #0,$DFF064      ; mai fare CLR.W $dff064!
9  oppure:
10     MOVEQ   #0,d0
11     MOVE.W  d0,$dff064
12     ...

```

Per sicurezza, quindi, vi consiglio di accedere tramite registri o valori diretti (#0,\$dffxxx), mai usare un CLR su un registro \$dffxxx.

3. Nel 1988-1989 era in uso un modo alquanto stupido di puntare le copperlist, che poi si è rivelato incompatibile con le versioni del sistema operativo dal 2.0 in avanti, dato che venivano date per scontate delle strutture di sistema che, naturalmente, non essendo

documentate dalla Commodore, sono cambiate lasciando “i furbi” con i loro sorgenti che non puntavano più le copperlist su A500+ e A600. Purtroppo qualcuno poco esperto di programmazione ha continuato a “rubacchiare” pezzi di listati vecchi che ha trovato in giro, contenenti questo ridicolo codice di puntamento della copperlist, per cui anche alcune demo del 1990-91 richiedono il kickstart 1.3 per poter funzionare a causa di questa leggerezza. Riporto il penoso codice inventato da qualche “furbo” pioniere del coding:

```

1      move.l 4.w,a6      ; execbase
2      move.l (a6),a6     ; ???
3      move.l (a6),a6     ; HAHAHA! GFXBASE??? Solo in kick1.3!
4      move.l $26(a6),OLDCOP ; HAHAHA! SALVA VECCHIA COPLIST???
5      move.l #MYCOP,$32(a6) ; DOPPIO HAHAHHA! PUNTA COPLIST???
6      ...

```

Questo pezzo di codice purtroppo è comunissimo nei vecchi listati in giro nelle banche dati, e nelle intro (es. quella degli *ORACLE*). Ricordatevi SEMPRE di non commettere il **duplice** errore contenuto in queste 4 linee di **sporchissimo** codice: Innanzitutto il GFXBASE si trova aprendo la `graphics.library` come viene fatto negli esempi del corso, e non certo facendo due volte `move.l (a6),a6`, questo avviene casualmente solo nei kickstart 1.2 e 1.3 per la particolare struttura della vecchia libreria. Il secondo errore è quello di puntare la copperlist mettendo il suo indirizzo nella struttura GFXBASE anziché nel registro `$dff080`, questo causa infiniti disastri, puntate sempre la copperlist con un bel:

```

1      MOVE.L #Copperlist,$dff080 ; COP1LC
2      move.w d0,$dff088          ; COPJMP1

```

4. Un'usanza della vecchia generazione di coder era anche quella di non aspettare la fine di una blittata prima di farne un'altra. Questo si trova molto facilmente nel codice scritto prima del 1990, ma ci sono alcuni che continuano tuttora a sorvolare le routine di `WaitBlit`. Effettivamente quando gli Amiga avevano solo il 68000 come processore c'erano dei casi in cui, tra una blittata e l'altra, il processore doveva fare tante di quelle operazioni che il blitter aveva già finito. I programmatori di demo (e purtroppo anche di giochi) spesso pensavano che era inutile aspettare il blitter se, anche senza mettere la routine di `Wait`, il tutto funzionava. Un esempio è il gioco *PANG*... Ma se hanno risparmiato 2 linee di codice nel loro demo/gioco, non solo non hanno aumentato la velocità di esecuzione (non sono certo un paio di `btst #6,$dff002` a rallentare...), ma non hanno considerato che su processori più veloci il tempo tra una blittata e l'altra si è accorciato, dato che è risaputo che il 68020 è più veloce del 68000, per cui il crash è totale.

Purtroppo il blitter è rimasto, su ECS ed AGA, della stessa lentezza (su a4000 o a1200 accelerato addirittura è più lento del normale!). Per risolvere i guai prodotti da tali “leggerezze” di programmazione, alle volte basta togliere le cache e la fast ram, per cui anche un 68020, se lavora in chip ram con le cache disattivate, alle volte rallenta abbastanza da evitare una blittata sopra un'altra già in esecuzione. La cosa brutta è che, per motivi di sincronizzazione hardware, su computer come A4000 o A1200 accelerati con 68030, il blitter è *più lento che nel vecchio A500*, per cui, anche se riusciamo a rallentare il processore come un 68000 base, è il blitter più lento che rende il crash inevitabile e non certo gradito. Tra l'altro tenete presente che anche se ci sono tutte le routine che aspettano la fine della blittata prima di farne un'altra, il blitter, essendo più lento su A4000, può causare delle “scattosità” orrende in giochi o demo che vanno fluide (a 50 fotogrammi al secondo) su un A500, rendendo nervoso l'incredulo possessore di A4000 che, invece, credeva di veder girare più velocemente il gioco o la demo. Quindi SEMPRE e COMUNQUE aspettate che il blitter abbia finito:

```

1      LEA      $dff000,a5
2 WaitBlit0:
3      BTST.B   #6,2(a5)
4 WaitBlit1:
5      BTST.B   #6,2(a5)      ; controlla 2 volte per un errore nell'A1000
6      BNE.S    WaitBlit1

```

P.S: Alle volte potete trovare dei btst #14,\$dff002 anziché dei btst #6,\$dff002, ma l'effetto è lo stesso, dato che viene testato sempre il bit 6. Infatti 6+8 fa 14. Il BTST lavora solo su byte e testa comunque il sesto bit. Si preferisce per motivi estetici (e di logica) usare btst #6 e non btst #14!!

Per darvi un'idea di quanto rallenti il blitter su macchine accelerate, considerate che, per esempio, una routine che blittava 14 bob per fotogramma su un A1200 base, ne blittava solo 12 su un A4000 e solo 9 su un A1200 con scheda acceleratrice GVP 030 a 40Mhz!!!! Considerate quindi che, quando è possibile, è meglio usare il processore che il blitter. Inoltre è sempre bene lasciare qualche linea raster “libera”, anziché blittare fino all'ultimo milisecondo. Infatti, in quest'ultimo caso, con il rallentamento del blitter su a4000 o a1200 accelerati non ce la farebbe più in un frame, e il tutto diverrebbe mega scattoso.

12.2 Errori riguardanti CIAA/CIAB, tastiera, timers, trackloaders

5. Routines che fanno lampeggiare il led del tasto caps lock non funzionano su A1200 perché ha una tastiera economica diversa da quelle standard. Tali routines sono presenti in certe demo per “bellezza”, eccone una qua di seguito, provatela e noterete il flash su A500/a2000/a3000/a4000, e, invece, un bel reset su un A1200:

```

1 CAPSLOCK:
2      LEA      $BFE000,A2
3      MOVEQ    #6,D1      ; bit 6 of $bfee01—input—output bit of $bfec01
4      CLR.B    $801(A2)    ; reset TODLO — bit 7—0 of 50—60hz timer
5      CLR.B    $C01(A2)    ; CLear the SDR (synchronous serial shift
6                          ; connected to the keyboard)
7 DOFLASH:
8      BSET     D1,$E01(A2)  ; Output
9      BCLR     D1,$E01(A2)  ; Input
10     CMPI.B   #50,$801(A2) ; Wait 50 blanks (CIA timer)
11     BGE.S    DONE
12     BSET     D1,$E01(A2)  ; Output
13     BCLR     D1,$E01(A2)  ; Input
14     MOVE.W   $DFF01E,D0   ; Intregr in d0
15     ANDI.W   #%00000010,D0 ; checks I/O PORTS
16     BEQ.S    DOFLASH
17 DONE:
18     RTS

```

L'amiga 1200 ha un controller della tastiera economico, provate a fare questa prova per verificare: premete il tasto <r>, lasciatelo premuto, e premete un'altro tasto, ad esempio la <u>. Su un a1200 non succede nulla, mentre su un altro computer la <u> appare sullo schermo. Dunque non scherzate con le routine che gestiscono la tastiera! Una delle demo che non funzionano su A1200 per questa routine è ODISSEY.

Per quanto riguarda le routines che “muovono” le testine dei drives, l'errore fondamentale è quello di sbagliare nelle routines di sincronizzazione, facendole con semplici loop “a vuoto” o serie di NOP che, su processori più veloci, vengono eseguite troppo in fretta per attendere abbastanza. Temporizzate col VBLANK o col CIA!

12.3 Errori riguardanti i processori 68010/20/30/40/60

6. Innanzitutto bisogna ricercare gli errori anche nelle utility che usiamo, e non solo nel nostro eseguibile. Infatti mi è successo spesso di far funzionare demo o intro (che “guravano” subito) semplicemente scompattandole e ricompattandole con un cruncher moderno, ad esempio Powerpacker o lo StoneCracker 4. Infatti molti dei vecchi compattatori (crunchers) ad indirizzi assoluti non funzionano su 68010+, per cui anche se la demo in sè funziona, il solo fatto di essere compattata con un vecchio ByteKiller o TetraPacker la fa andare in guru, prima di partire, durante il decrunch. Come prima cosa, dunque, non compattate il vostro programma con vecchi crunchers, usate Stone Cracker4, PowerPacker o Titan Cruncher. Inoltre è sempre meglio fare codice rilocabile, che codice ad indirizzi assoluti!!!
7. Errori di indirizzo: Alcune vecchie produzioni contengono degli accessi agli indirizzi della ROM, ad esempio:

```
1 JSR $fce220
```

Ebbene, il kickstart 1.2/1.3 è localizzato, nei vecchi Amiga, alle locazioni di memoria `$fc0000`, fino a `$ffffff`, per un totale di 256k. è ovvio che in questi kickstart ogni routine ha il suo indirizzo: come abbiamo già visto esiste una “tabella di JMP” all’indirizzo di Execbase, ossia sappiamo che, ad esempio, avendo l’execbase in a6, troveremo, `$84` bytes prima, il JMP che salta in ROM ad eseguire il Forbid:

```
1 jsr -$84(a6) ; Forbid, disabilita multitask
```

per esempio nel kickstart 3.0 (Version 39.106) questa è la tabella di JMP dell’execbase (una sua parte disassemblata):

```
1 ...
2 JMP $00F815CC ; ...
3 JMP $00F815A2 ; -$96(a6)
4 JMP $00F81586 ; -$90(a6)
5 JMP $00F8286C ; -$8a(a6) - routine del permit
6 → JMP $00F82864 ; -$84(a6) - Routine del FORBID
7 JMP $00F817F8 ; -$7e(a6)
8 JMP $00F817EA ; ...
9 ...
```

Su un computer con kickstart V39.106, si può ottenere un FORBID con un:

```
1 JSR $F82864 ; Forbid su kickstart V39.106 di A1200/A4000
```

Ma se, ad esempio, il kickstart V39.106 è caricato via software e non è in ROM, la tabella dei JMP punterà agli indirizzi della RAM dove è stato caricato il kick. Dunque MAI e poi MAI accedere al kickstart in questo modo, o la vostra produzione funzionerà solo sul vostro computer. Con questo esempio, comunque, potete intuire che si possono modificare le tabelle dei JMP sostituendo l’indirizzo in ROM con uno nostro, in modo da far eseguire nostre routines modificate. È in questo modo che agiscono i programmi che modificano il sistema operativo, ad esempio le utility che aggiungono un gadget alle finestre o che aumentano opzioni al workbench. È per questa “relatività” del sistema operativo che MAI bisogna saltare nella ROM. L’unico indirizzo fisso del sistema operativo Amiga è `$0004`, ossia l’EXECBASE, che contiene l’indirizzo da cui fare gli offset. Per cui, se volete avere a che fare col sistema operativo, seguite sempre le indicazioni standard. E anche se non ci volete avere a che fare! Questo tipo di errori è letale, tanto che molte vecchissime demo fatte su Amiga500 Kick1.2, non funzionano su Amiga500 Kick1.3, o superiori, neppure caricando via software il kickstart 1.2.

Altri errori di indirizzo, di poco meno gravi, sono quelli che danno per scontato che la fast ram sia a \$c00000. Originariamente Amiga aveva 512k di CHIP RAM, successivamente si diffuse l'espansione interna che la portava a 1MB, ed è noto che i 512k di fast ram aggiuntivi sono da \$c00000 a \$c80000. Anche le demo e i giochi allora cominciarono ad essere progettati per riempire l'intero megabyte di memoria, e siccome in quel periodo la grande maggioranza dei programmi era ad indirizzi assoluti, i coder pensarono di assemblare il programma in fast ram, nelle locazioni da \$c00000 a \$c80000, e di caricare la grafica e la musica in chip ram, da \$00000 a \$80000. Per cui il programma, oltre ad essere scompattato negli indirizzi assoluti \$c00000, avevano le istruzioni allocate per quella zona:

```

1  ...
2  MOVE.L  #$c23b40,d0
3  jsr     $c32100
4  ...

```

Queste demo o giochi funzionarono sugli A500 espansi internamente con la classica scheda, ma quando uscirono gli A500 plus, dotati sempre di 1MB di memoria, ma di sola CHIP, tutti questi programmi risultarono inutilizzabili. Questo è avvenuto perché con 1MB di chip, la memoria è disposta in questo modo: i primi 512k si trovano sempre da \$00000 a \$80000, ma i secondi 512k sono di seguito da \$80000 a \$100000!!! Per cui un JSR \$c32100 non porta da nessuna parte, comunque porta di sicuro ad un crash spettacolare con i fuochi d'artificio sullo schermo. In seguito a questo, i giochi e le demo successive hanno usato metodi diversi per sfruttare la memoria oltre i primi 512k. Uno di questi è quello di abbandonare del tutto l'indirizzamento assoluto, dimenticando i comandi ORG e LOAD, e anche i programmi in autoboot, dato che quelli, con un loro loader, devono per forza essere messi ad indirizzi assoluti. Se da una parte molti demo/giochi successivi caricabili via DOS divennero rilocabili tramite le SECTIONS al 100%, senza parti caricate ad indirizzi fissi, altri non vollero rinunciare all'autoboot e agli indirizzi fissi, per usare fino all'ultimo byte di memoria. Questi ultimi risolsero il problema in un paio di modi: uno è quello di assemblare due programmi principali, uno fixato con ORG e LOAD a \$c00000, se veniva accertato che il computer aveva mezzo mega di CHIP e mezzo di FAST, e un altro fixato all'indirizzo \$80000, da caricare invece nel caso che il computer avesse 1MB o più di CHIP. In questo modo al boot una routine controlla in quale caso siamo, e carica l'uno o l'altro programma principale all'indirizzo giusto, mentre i dati come grafica e suoni sono poi caricati in seguito dal programma principale. Questo sistema ha lo svantaggio di richiede lo spreco di spazio su disco per le due versioni del programma principale. Altri più "bravi" si sono invece programmati un piccolo sistema operativo, il quale al boot prende nota di quali segmenti di memoria sono presenti nel computer, e tramite una routine di allocazione propria riallocano le varie parti del programma all'indirizzo dove trovano la FAST RAM. Questo è sicuramente il miglior modo di fare un programma AUTOBOOT, anche se è piuttosto difficile, e i vantaggi sono questi: immaginate di caricare su un A4000 una demo o un gioco col sistema dei due programmi principali: al boot il programma riconosce la memoria e, avvedendosi che non è presente la memoria a \$c00000, carica in CHIP RAM a \$80000 il codice. Lo stesso demo/gioco, invece, viene modificato per caricare col mini sistema operativo: al boot questo riconosce che ci sono due blocchi di memoria, quella CHIP da \$000000 a \$200000 e quella FAST da \$7c00000 a \$7ffffff, di conseguenza riloca tutte le parti di codice in FAST RAM e carica la grafica e il suono in CHIP RAM; come è noto il codice in FAST RAM è molto più veloce che in CHIP RAM, specialmente su processori TURBO come il 68040, di conseguenza la demo o il gioco andranno molto più veloci con il codice rilocato in FAST. Però è da notare che coloro che hanno usato loro sistemini operativi hanno visto andare in crash la loro demo con l'avvento dei 68040, o anche con l'avvento

del semplice 68020, perché la motorola garantisce la compatibilità completa verso il basso SOLO in modo user, e non in modo supervisor: infatti il 68040 ha proprie istruzioni per il modo supervisor, e anche il 68060 è compatibile 100% solo in usermode... Figuriamoci processori o computer del futuro, che magari emuleranno il 680x0... **non andate mai in supervisor e non fatevi sistemini operativi**, per darvi un'idea, la bellissima demo WOC 92 dei Sanity, a causa del suo sistemino operativo, non funziona su 68040... e lo stesso è successo alla demo italiana *IT CAN'T BE DONE*, di un mio amico, e in questo ultimo caso sono stato io a trovargli l'errore: la routine supervisor!!! Tutto sommato, penso che sia più facile e SICURO usare le SECTIONS per fare codice eseguibile, anche perché c'è il vantaggio di poterlo installare su HardDisk, e sicuramente nei prossimi anni Amiga dovrà sempre più rendersi competitivo con l'MSDOS, e con questo intendo che il computer di BASE dovrà avere l'HardDisk e la FAST RAM, altrimenti rimarremo in pochi a vedere i giochetti da 1MB che caricano dal dischetto in autoboot, e che non sfruttano nemmeno la velocità del processore non caricando il codice in FAST RAM!!

Fino qua ho indicato come sia meglio fare codice rilocabile, ma cosa succede se usiamo indirizzi assoluti per un file eseguibile caricabile da dos? Ebbene l'introduzione dell'A500+, con 1MB di CHIP ha portato a non far funzionare anche molte delle produzioni a codice "misto", ossia con codice rilocabile, creato con le "SECTION", ma con l'uso di buffer per la grafica NON allocati tramite Section BSS o AllocMem, bensì stabiliti arbitrariamente:

```

1      lea      $30000,a0      ; Indirizzo bitplane buffer
2      bsr.s    PrintText     ; Stampa il testo a $30000

```

In questo caso non è il codice ad essere non rilocabile, ma il buffer grafico. Di conseguenza non esiste nemmeno la routine di puntamento dei bitplane in copperlist, perché viene messo direttamente il valore \$30000 dal programmatore: (ORRORE!!!)

```

1      ...
2      dc.w     $e0,$0003      ; bpl0pth
3      dc.w     $e2,$0000      ; bpl0ptl
4      ...

```

Vediamo cosa succede sui computer vecchi, quelli con 512k di CHIP e 512k di FAST: supponendo che la intro abbia una section CODE di 20k e una CHIP di 40k (contenente il FONT dei caratteri e la musica), la prima section viene caricata in FAST e la seconda in CHIP, per cui non si arriva alla locazione \$30000, ma, mettiamo, a \$2a000. In questo caso tutto funziona, purché questa intro sia la prima cosa che viene caricata dal dos. Sulle macchine più recenti, con SOLO 1MB o 2MB di chip, non essendoci la FAST MEM, viene caricato tutto in CHIP, sia la section CODE che l'altra, in questo modo gli ultimi Kb di codice (o di musica, grafica ecc.) si trovano oltre l'indirizzo \$30000. Immaginatevi che bel CRASH avviene quando la routine stampa i caratteri sopra il codice! La disperazione dei coder "leggerini", all'uscita di A500+ e a600, era anche che non riuscivano a correggere i listati su tali computer, in quanto l'ASMONE stesso veniva caricato in CHIP RAM, oltrepassando le locazioni \$30000 o \$40000 usate come buffer assoluti, per cui al JMP poteva anche funzionare (PER CASO) il listato, ma all'uscita l'ASMONE si ritrovava PERFORATO dalle routines e il CRASH era inevitabile. Questo ha insegnato anche ai produttori di intro che bisogna farsi un bel buffer rilocabile:

```

1      SECTION BufferOK,BSS_C
2
3      ds.b     10000

```

Per concludere la serie degli errori riguardanti gli indirizzi, riporto ora degli errori che difficilmente avreste fatto, dato che si tratta di azioni illogiche, ma per sicurezza è bene sapere che:

alcuni coder furboni hanno alle volte usato il byte alto degli indirizzi per scrivere messaggi o semplicemente per il gusto di scrivervi.

Dovete sapere che le CPU a 16 bit come il 68000 o il 68010 ignorano il byte alto di un indirizzo, per cui fare:

```
1      JSR      $00005a00
2      JSR      $00120d00
3      JSR      $00c152b0
4      JSR      $00013cd0
```

è equivalente a scrivere

```
1      JSR      $C0005a00
2      JSR      $DE120d00
3      JSR      $FEc152b0
4      JSR      $DE013cd0
```

Si legge chiaramente nei primi byte un “CODE-FEDE”, che può essere un messaggio lasciato da un Emilio Fede coder di tanti anni fa, che si firmava in questo modo. Da notare che con gli esadecimali si possono formare molte parole (A,B,C,D,E,F, e lo 0 come una “O”), ad esempio: FEDE, AFA, ABAC0, FACCE, FOCA, CACCA, CADE, CODE, ...

Questi furboni dunque lasciavano messaggi interi, poesie, lettere d’amore nei byte alti degli indirizzi nelle serie delle subroutine o in altri luoghi, dove chi disassemblava poteva leggersi anche insulti! Questo giochetto è durato assai poco, per fortuna, ma le intro/demo che li hanno non funzionano su processori a 32 bit, infatti su tali processori l’indirizzamento massimo è aumentato, per cui il JSR cerca veramente quelle strane locazioni. Tra l’altro avrete notato come la FAST RAM dei vecchi A500 sia a *\$00c00000*, mentre quella dell’A4000 sia a *\$07c00000*, cioè fuori del raggio di indirizzamento di un 68000.

L’ultimo tra gli errori di indirizzo, e non meno insolito di quello precedente, è quello della memoria CHIP da 512k, che viene “ripetuta” quattro volte nel bus indirizzi, nel senso che nonostante che questa si trovi da *\$00000* a *\$7ffff*, ci si può accedere anche operando su *\$80000-\$FFFFF*, oppure *\$100000-\$17ffff*, o *\$180000-\$1FFFFF*. In pratica i *\$80000* bytes (512k), sono 1/4 dei *\$200000* (2MB) del bus, e sulle macchine OCS (vecchi amiga che potevano indirizzare solo 512k di CHIP, il resto FAST), ogni byte di memoria CHIP è accessibile da quattro indirizzi diversi, distanziati l’uno dall’altro da 512kb. Questa naturalmente è una proprietà che sulle macchine ECS ed AGA, cioè quelle che possono indirizzare 1MB o più di memoria, si è persa.

Facciamo un esempio: se scriviamo alla locazione *\$0* il valore *\$12345678*, possiamo “ripestare” quel valore anche da *\$0 + \$80000*, *\$0 + \$80000*2*, nonché *\$0 + \$80000*3* e *\$0 + \$80000*4*. Vediamo un listato:

```
1      move.l  #$12345678,$0      ; mettiamo nei primi 4 bytes questo valore
2
3      move.l  $80000,d0          ; d0 = $12345678
4      move.l  $100000,d1         ; d1 = $12345678
5      move.l  $180000,d2         ; d2 = $12345678
```

Leggendo da *\$80000*, *\$100000* e *\$180000* è come se leggessimo da *\$0!!!!* Purtroppo qualche stupido ha usato questa strana proprietà per le sue routines, e questo causa il cattivo funzionamento di a500+ e a600 di varie cose, e badate bene che nonostante che il processore sia sempre il 68000, l’errore si presenta anche con kickstart 1.3 in ROM.

Dunque, ora conoscete tutti gli errori riguardanti gli indirizzi che sono stati fatti nel passato. Vedete di non farli e di non inventarne di nuovi!!!!

8. Problemi con lo SR nei processori 68010 e superiori: Uno dei problemi di incompatibilità più frequenti nei processori 68010 e superiori rispetto al codice 68000 è quello delle istruzioni `MOVE SR,dest` ad esempio `MOVE SR,d0` o `MOVE SR,$1234` o `MOVE SR,LABEL`. Infatti queste istruzioni su 68000 base possono essere usate normalmente in modo utente (MODE USER) come qualsiasi altra istruzione: programmi come gli emulatori (PC Transformer, C64 Emulator ecc.) non funzionano su 68010+ proprio perché eseguono questa operazione in modo USER, il che su 68010+ non è più possibile e causa un guru di “*Privilege Violation*”.

Anche molti giochi e demo si inchiodano per la presenza di questa istruzione nella parte iniziale delle routines che prendono il controllo del sistema. La Motorola decise di aggiungere ai processori dal 68010 in avanti la possibilità di simulare il funzionamento di nuovi sistemi operativi per macchine ancora non disponibili, questo comportò la necessità di rendere l'istruzione `MOVE SR,dest` privilegiata, ossia eseguibile soltanto in modo supervisore. Altrimenti il risultato è una GURU di “*Privilege Violation*”. Per accedere allo SR in modo utente, comunque, i progettisti Motorola hanno aggiunto dai processori 68010 in avanti l'istruzione `MOVE CCR,dest` da usare al posto di `MOVE SR,dest`, che però non era disponibile su 68000, per cui alcuni programmi per il 68000 di Amiga furono scritti utilizzando l'istruzione `MOVE SR,dest` in modo utente, ed ora lo verifichiamo quando un gioco o una demo si inchiodano al boot con un sinistro messaggio di GURU MEDITATION o di SOFTWARE FAILURE su A1200/A3000/A4000 o A2000 accelerati.

In realtà lo “sbaglio” lo ha fatto la Motorola, in quanto gli ignari che hanno usato fiduciosamente `MOVE SR,dest` in modo utente (USER) non si aspettavano certo che divenisse un'istruzione da eseguire solamente in modo supervisore! (dopo un TRAP o una EXCEPTION del processore).

Comunque l'importante è saperlo, ed ora possiamo essere sicuri di non incorrere nel problema, e questo è possibile ricordandosi di eseguire sempre l'istruzione `MOVE SR,dest` in modo SUPERVISORE, ossia dopo un TRAP, o in un INTERRUPT, eccetera. In questo modo l'istruzione funzionerà su tutti i processori. Un'altra soluzione potrebbe essere quella di controllare che processore c'è sulla macchina, ed eseguire il codice appropriato, ossia un `MOVE SR,dest` su 68000, oppure un `MOVE CCR,dest` su 68020, entrambi in modo USER per evitare di doverle eseguire in modo SUPERVISOR, ma ritengo che la soluzione più veloce e ragionevole sia quella di eseguire sempre il `MOVE SR,dest` in modo SUPERVISOR. Recapitolando:

CPU	Modo USER (UTENTE)	Modo SUPERVISORE
68000	<code>MOVE SR,dest</code>	<code>MOVE SR,dest</code>
68010/20/30/40	<code>MOVE CCR,dest</code>	<code>MOVE SR,dest</code>

Converrete che conviene eseguire sempre il vecchio `MOVE SR,dest` in modo supervisor, ciò risparmia tempo e routines. Se invece il gioco/programma/demo che state programmando è destinato solo a processori 68010+, ad esempio se la demo è solo AGA, potete usare il nuovo `MOVE CCR,dest` in modo utente, dato che siete su un 68020+, ma ricordatevi anche che tale istruzione non esiste su 68000, per cui non viene assemblata da assembleri 68000 base, come questo TRASH'M-One; per poter assemblare istruzioni 68010+ come questa dovete usare il TFA ASMONE o il DEVPAC 3.

D'altronde, vi consiglio proprio di **non usare mai questa istruzione**, e di **non andare mai in modo supervisore**. . . a che vi serve? Per rischiare? Quando avviene un errore di questo

tipo, il numero del *SOFTWARE FAILURE* da sistema operativo è #80000008, non è difficile identificarlo.

Comunque per programmare demo o giochi agire sul registro SR non ha una utilità fondamentale, per cui vi consiglio VIVAMENTE di non accedere MAI a questo registro, anche perché i suoi bit sono diversi da processore a processore, ed è facilissimo causare problemi di incompatibilità.

9. Col 68010, oltre al comando *MOVE CCR,SR* che abbiamo visto, sono state introdotte altre novità, che se non conosciute possono causare errori di incompatibilità. Si tratta del VBR, ossia del VECTOR BASE REGISTER, che significa “registro di base del vettore”. Abbiamo visto questo registro nella lezione sugli interrupt, infatti sappiamo che quando avviene un interrupt o una trappola (istruzione *TRAP #xx*), il processore INTERROMPE la lettura del programma che stava eseguendo in modo USER, passa al modo SUPERVISORE ed esegue la routine all'indirizzo che trova nel VETTORE specifico, che può essere uno dei livelli di interrupt, oppure uno dei TRAP, eccetera.

Nei nuovi processori, oltre ad essere stati usati vettori che nel 68000 non avevano funzioni (si vedano, ad esempio, \$18 e \$1c), è stata implementata la possibilità di spostare la BASE di questi vettori. Mentre su un 68000 siamo sicuri che l'interrupt del VBLANC è sempre all'indirizzo \$6c, su un 68010 o superiori non ne possiamo essere sicuri. Questo perché la base di questi OFFSET può non essere più \$000000. Infatti basta eseguire, in supervisore, un *MOVEC d0,VBR*, e cambia tutto. Naturalmente al momento del boot il VBR è azzerato, per cui i vettori sono tutti allo stesso posto del 68000. È il *SetPatch* dell'AmigaDos che sposta il VBR, normalmente in FAST RAM, copiando gli indirizzi dei vettori al nuovo indirizzo. Oppure lo “spostamento” viene fatto da altre utility. Di fatto, quindi, una volta caricato il WorkBench su un computer con 68010+ è molto probabile che il VBR non sia a zero, per cui le vecchie demo e i giochi (non solo quelli vecchi!), se caricati dallo Shell o dal WorkBench, spesso non hanno la musica o proprio si inchiodano, perché mettono le routine di interrupt in \$6c, quando lo dovrebbero mettere in *VBR+\$6c*. Quindi MAI si deve fare una cosa del genere:

```
1  MOVE.L  #IntRoutine , $6c
```

Per prima cosa, si poteva “ottimizzare” in:

```
1  MOVE.L  #IntRoutine , $6c.w
```

Ma la cosa più importante è che funziona solo se caricata al boot, prima di eseguire il *SetPatch* o altre utility. Per ovviare al problema, bastano poche righe di codice, che controllino se è presente un 68000 o un 68010+, e in quest'ultimo caso leggano il valore del VBR per eseguire i dovuti offset. Alla fine del programma, basterà ricordarsi di fare lo stesso per rimettere a posto il vecchio interrupt. Questo accorgimento è presente nella *startup2.s* usata nei listati avanzati del corso. Da notare che l'istruzione *MOVEC VBR,A1*, essendo 68010+, non viene assemblata da tutti gli assembleri (compreso questo ASMON), per cui è meglio metterlo tramite il suo equivalente esadecimale. Nessuno infatti vi impedisce di scrivere *dc.w \$4e75* al posto degli RTS!

10. Ora vediamo le leggerezze di programmazione che sono state rese evidenti con l'introduzione delle *INSTRUCTION CACHE* dei processori dal 68020 in avanti. Per “rese evidenti” intendo che tali errori portano ad un crash del sistema spaventoso. Fortunatamente molti di questi errori possono essere risolti disattivando le CACHE tramite utility software.

Vediamo in breve cosa sono queste CACHE: si tratta di memoria molto veloce che si trova DENTRO il processore anziché fuori, al contrario della CHIP o FAST ram, che per essere raggiunte occorre passare dall'autoBUS.

Abbiamo già visto i REGISTRI dati e indirizzi, che non sono altro che LONG di memoria interna al processore, che possiamo leggere e scrivere. Ebbene le CACHE sono banchi di memoria simili, che però non possiamo leggere o scrivere con delle istruzioni, vengono lette e scritte automaticamente dal processore tramite un apposito hardware. Lo scopo delle cache è di velocizzare i LOOP, ossia le routines che vengono eseguite ciclicamente molte volte. Premetto che su 68020 e 68030 l'INSTRUCTION cache è di 256 byte, mentre su 68040 è di 4096 byte. Su 68060 credo sia di 8192, e in futuro chissà... Ebbene, immaginate questo loop:

```

1      ...
2      MOVEQ    #100,d0
3
4 Loop1:  move.w  LABEL1(PC),d2
5         add.w  d3,d2
6         ....
7         altre  istruzioni
8         ....
9         DBRA   d0,loop1
10        ...

```

Anche aumentando la velocità del processore, questo loop richiede la lettura delle istruzioni tra la label loop1: e il DBRA d0,loop1 ogni ciclo, e la lettura da RAM, specialmente se è CHIP RAM, è molto lenta. I progettisti Motorola quindi hanno escogitato questo trucco: “e se mettessimo automaticamente nella cache memory gli ultimi 256 byte che sono stati eseguiti?? Otterremmo che quando si presenta un loop più piccolo di 256 byte, tutte le istruzioni del loop stanno nella CACHE e il processore può leggerlo le restanti volte dalla veloce memoria CACHE anziché dalla RAM!”. Così più o meno funziona la Instruction CACHE.

Il loop di prima sarebbe letto dalla RAM solo la prima volta poi, raggiunto il DBRA, il processore “si accorge” che Loop1: è abbastanza vicino da essere contenuto sempre nella CACHE, e le restanti 99 volte il tempo di lettura delle istruzioni si abbassa notevolmente essendo eseguito dalla CACHE anziché dalla CHIP/FAST RAM.

Vi chiederete: ma allora quali errori si possono verificare???? Il più comune è quello di chi ha “temporizzato” certe routines basandosi sul tempo che occorre al 68000 base per fare un certo numero di loop “a vuoto”, (ora si ritrova con un listato da buttar via). Vediamo i “furbi” come hanno fatto a “perdere del tempo” per aspettare, ad esempio, che le testine del disk drive si spostassero, o per temporizzare una musica, o altro ancora:

```

1      ....
2      MOVE.W   #2500,d0
3
4 Aspettatempo: dbra    d0,Aspettatempo
5               ...

```

Questi esempi di programmazione impacciata ed approssimativa sono, purtroppo, molto frequenti nei track loaders e nelle routines che suonano le musiche. La routine music.s presente nel corso aveva in origine un paio di questi “loop a vuoto”, che ho prontamente sostituito con routines “perditempo” affidabili, che ora vedremo.

Se avete delle replay routines per noisetracker/protracker quasi sicuramente troverete dei loop stupidi di questo tipo, che causano la perdita di alcune note durante l'ascolto della martoriata musica. Una nota: anche nei listati del corso di Gerardo Proia ci sono loop stupidi di questo tipo, spero non li abbiate assunti come esempio!

Cercate nei listati che avete trovato in giro questi loop maledetti, e se li trovate buttate via quei sorgenti schifosi o sostituite quelle parti con routines che usino il CIA o il VBLANK, per temporizzare. Il principio di funzionamento di un loop a vuoto è che il processore deve leggere, in questo caso 2500 volte, l'istruzione DBRA dalla memoria, sottrarre #1 a d0 e saltare indietro ad Aspettiamo:.

Su un computer con la cache attiva si può far leggere anche 50000 volte, ma, essendo finito in CACHE, il DBRA verrà eseguito in una frazione di secondo comunque, di conseguenza il drive non legge le tracce, e la musica “taglia” delle note. Oltretutto anche il 68010, per la verità, ha una piccola CACHE di 3 word per velocizzare i piccoli loop DBRA come questo, per cui tali loop “funzionano” solo su 68000 a 7Mhz. Dato che nel corso MAI viene insegnato a temporizzare con loop a vuoto, spero che nessuno cominci autonomamente a fare simili *CAZZATE*.

In generale si può dire che MAI bisogna prendere come riferimento la velocità di esecuzione delle istruzioni da parte della CPU 680x0, dato che varia da processore a processore, e addirittura a seconda di quale tipo di memoria viene letta. Le uniche certezze, dal punto di vista della temporizzazione, sono **il refresh video del VBLANK**, che in standard PAL sarà sempre eseguito 50 volte al secondo, e **i timer del CIA**, che è un chip uguale per tutti gli Amiga, per cui 1 millisecondo è uguale su un A500 come su un A4000. Badate di non basarvi neppure sulla velocità del blitter, perché varia di velocità a seconda del processore del computer.

Ecco come temporizzare trackloaders, segnali per le periferiche esterne o bracci robot collegati alla porta parallela:

Innanzitutto vediamo come aspettare qualche “linea raster” col VBLANK:

```

1      LEA      $DFF000,A5      ; Custom register base in a5
2  PerdiTempo:
3      MOVE.w  #LINEE-1,D1      ; NUMERO DI LINEE da ASPETTARE (304=1 frame)
4  VBWAITY:
5      MOVE.B  6(A5),D0         ; $dff006, VHPOSR.
6  WBLAN1:
7      CMP.B   6(A5),D0         ; VHPOSR
8      BEQ.S   WBLAN1
9  WBLAN2:
10     DBRA    D1,VBWAITY

```

Se non state usando i timer del CIA per altre routines, allora potreste usarli, anche se è meglio toccarli il meno possibile, dato che li usa il sistema operativo. Nella lezione11 trovate sorgenti sull'argomento. Nell'usare i timer del CIA considerate che anche il sistema operativo ne utilizza alcuni per certi scopi: (meglio usare il CIAB!)

CIAA, timer A	Utilizzato per l'interfacciamento con la tastiera
CIAA, timer B	Utilizzato dall'exec per lo scambio dei task ecc.
CIAA, TOD	timer a 50/60 Hz utilizzato dal Timer.device
* CIAB, timer A	Non utilizzato, disponibile per i programmi
* CIAB, timer B	Non utilizzato, disponibile per i programmi
CIAB, TOD	Utilizzato dalla graphics.library per seguire le posizioni del pennello elettronico.

Se dovete usare timer che servono anche al sistema operativo, fatelo solo se avete disabilitato multitasking e interrupt di sistema, se avete cioè preso il controllo completo del

sistema. Comunque è sempre più pericoloso usare il CIA che il VBLANK, perché all'uscita dalla nostra produzione se abbiamo sballato un timer chissà cosa potrebbe succedere.

Oltre al problema dei loop di temporizzazione, c'è anche quello dato da un altro vizio di programmazione, che però al giorno d'oggi è quasi scomparso, fortunatamente. Si tratta del leggendario e misterioso codice “*automodificante*”: questo tipo di codice è detto automodificante proprio perché modifica se stesso. Infatti è possibile fare delle “creature” che, oltre a modificare dei dati, modificano anche le proprie istruzioni durante l'esecuzione.

Questo tipo di programmazione purtroppo è stato usato fin dall'antichità, probabilmente perché sembrava un modo per scrivere codice più veloce o più potente. In realtà quello che viene fatto con codice automodificante può essere riscritto con codice normale al 100% e, alle volte, si può guadagnare in velocità. Per cui dimenticatevi di scrivere codice di questo tipo, se non proprio a scopo di esperimento, perché tale codice non funziona con le CACHE del 68020/30/40/60 attive.

Chi ha un A1200 si rende certamente conto di quanti siano i giochi e le demo che non funzionano solo a causa delle cache! In effetti, credo che la maggior parte degli errori nel vecchio software sia di questo tipo. Per riconoscere un gioco o una demo che ha codice automodificante, basta provare se funziona togliendo le cache (senza caricare vecchi kick), poi riprovarla con le cache attivate. Se a questo punto non funziona, è ovvio che sono solo le cache attive a causare il problema, e le cache causano solo due tipi di errore: quello dell'annullamento dei cicli di ritardo DBRA, che abbiamo già visto, e quelli dovuti al codice automodificante. Ecco come si può presentare un listato contenente codice automodificante:

```

1      ...
2      divu.w    #3,d0
3 MYLABEL:
4      moveq    #0,d0
5      ...

```

Che viene assemblato in memoria in questo modo:

```

1      ...
2      dc.l      $80FC0003      ; DIVU.W #$0003,D0
3 MYLABEL:
4      dc.w      $7000          ; MOVEQ    #$00,D0
5      ...

```

Per ora abbiamo modificato dei dati in memoria, nelle copperlist, abbiamo messo valori nei registri custom *\$dffXXX*, ma non abbiamo mai agito DENTRO una istruzione!!! Questo proprio perché è una cosa **incompatibile**. (In realtà abbiamo modificato un JMP alla fine di un interrupt nel listato sul caricamento da dos in *Lezione11.txt*, ma è l'unico caso “utile!”). Immaginatevi che, più avanti nel listato, ci sia questa istruzione:

```

1      ...
2      move.w    #5,MYLABEL-2
3      ...

```

Cosa succede? La word che si trova prima della label MYLABEL è il *\$0003* del DIVU #3,d0, che diventa *\$0005*, per cui il DIVU #3,d0 diventa DIVU #5,d0!!! Allo stesso modo si possono modificare tutte le altre istruzioni. In altro modo si può scrivere:

```

1      ...
2      divu.w    #3,d0
3 MYLABEL:      EQU      *-2
4      moveq     #0,d0
5      ...

```

Ora per modificare il numero del DIVU basta un MOVE.W #xxxx,MYLABEL, infatti tramite l'EQU *-2 si fa corrispondere MYLABEL con la label -2. L'asterisco si può tradurre in "questo punto", per cui *-2 diventa "questo punto meno 2 bytes". Supponiamo ora di avere questa situazione in un listato con codice automodificante:

```

1      ...
2      divu.w  #0,d0    ; da modificare nel numero voluto
3 MYLABEL:
4      EQU    *-2
5      ...

```

Immaginate cosa accade quando la ICACHE è attiva: nella cache va l'istruzione così come è in memoria, ossia DIVU.W #0,d0, probabilmente prima che sia modificata, per cui al momento che viene eseguito il:

```

1      move.w  #5,MYLABEL

```

Viene modificata in RAM l'istruzione DIVU, ma non in CACHE! Infatti tale istruzione verrà eseguita così come era, ossia DIVU.W #0,d0, il che causa un bel crash di sistema per DIVISIONE PER ZERO!!

```

1      divu.w  #0,d0    ; il valore "0" sarà sostituito prima che questa
2 MYLABEL:           ; istruzione sia eseguita, ma con la ICACHE attiva
3      EQU    *-2      ; questa istruzione sarà letta dalla cache come era
4                  ; ; originariamente, e un bel guru DIVISION BY ZERO
5                  ; ; fermerà il nostro divertimento.

```

Allo stesso modo un:

```

1      JMP     0        ; l'indirizzo sarà messo in questo punto da un move,
2 MYLABEL:           ; ma con la cache avremo un bel JUMP a 0!! (guru!!)
3      EQU    *-6      ; (EQU *-4, EQU *-8, a seconda della grandezza
4                  ; ; dell'indirizzo o della "modifica" stupida.

```

In alcuni casi anziché un vero e proprio GURU si verificano dei problemi di malfunzionamento delle routines, per esempio un loop di questo tipo:

```

1      ...
2      MOVE.W  #100,d0
3 Loop1:
4      ...
5      divu.w  #2,d2
6 MYLABEL:
7      EQU    *-2
8      ...
9      addq.w  #1,MYLABEL
10     DBRA    d0,loop1
11     ...

```

Non si verifica alcun crash del sistema, ma mentre il divu.w ogni ciclo dovrebbe cambiare in DIVU.W #3,d2, DIVU.W #4,d2 eccetera, invece rimane sempre DIVU.W #2,d2 (essendo letto dalla CACHE e non dalla RAM). Se per esempio questa era una routine che faceva muovere un solido 3d, state certi che il solido rimarrà fermo o non si visualizzerà nemmeno.

Come avremo dovuto fare. La stessa cosa si sarebbe potuta fare dividendo per una LABEL o per un REGISTRO anziché per un valore assoluto (divu.w LABEL(PC),d2 anziché divu.w #xxx,d2), e si sarebbe potuto fare poi l'ADD sulla label, mantenendo la compatibilità con le cache e senza perdere in velocità.

Dunque **non siate stupidi**: fare codice automodificante non è una cosa di cui vantarsi, perché non è né difficile né utile, bensì è incompatibile. Vi prego di non usare vecchi listati che hanno codice automodificante. Esiste comunque un modo, oltre a quello di

disabilitare le cache, per far funzionare codice automodificante: si può infatti AZZERARE, ossia PULIRE la cache con una apposita istruzione, in questo modo verrà cancellata la vecchia istruzione dalla CACHE e il processore dovrà leggere quella modificata dalla RAM. Se volete fare programmi “batterici” o di intelligenza artificiale o chissà di che tipo, che si automodifichino, basta mettere un BSR.w CACHECLR prima dell'esecuzione dell'istruzione modificata. Su kickstart 2.0+ esiste un'apposita funzione della ExecLib:

```

1 ClearMyCache:
2     movem.l d0-d7/a0-a6, -(SP)
3     move.l 4.w, a6
4     MOVE.W $14(A6), D0      ; lib version
5     CMP.W #37, D0          ; è V37+? (kick 2.0+)
6     blo.s nocaches         ; Se kick1.3, il problema è che non può
7                             ; nemmeno sapere se è un 68040, per cui
8                             ; è rischioso.. e si spera che uno
9                             ; stupido che ha un 68020+ su un kick1.3
10                            ; abbia anche le caches disabilitate!
11     jsr     -$27c(a6)       ; cache clear U (per load, modifiche ecc.)
12 nocaches:
13     movem.l (sp)+, d0-d7/a0-a6
14     rts

```

Nella startup2.s è presente questa subroutine, che può essere eseguita. Nota: se siamo su kick 1.3 esce senza fare il JSR, in questo modo non ci sono problemi su computer vecchi.

Eseguite questa routine anche dopo aver caricato con un trackloader dei dati in memoria, o dopo aver fatto altre modifiche a zone di memoria contenente codice. Fate attenzione che alle volte un programma con codice automodificante può funzionare per caso su 68020/68030, perché ci sono più di 256byte di distanza tra l'istruzione modificata e quella che modifica, per cui l'istruzione viene letta dalla RAM, ma su A4000 la cache è di 4096 bytes, e chissà su processori futuri quanto può essere aumentata! Dunque MAI cadere nell'automodifica, come invece continuano a fare alcuni coder di demo per a1200, che ottengono solo di non far funzionare le loro creature su a4000.

Nei processori 68030 e 68040 esiste anche la DATA CACHE, che fa la stessa cosa della INSTRUCTION CACHE, ma sui dati (tipo le tabelle), ma solo se tali dati sono in FAST RAM. Su CHIP RAM la DATA cache non agisce. Errori per la DATA CACHE sono meno frequenti, infatti è difficile che vengano cambiate, ad esempio, delle tabelle. Comunque per sicurezza potete fare un CACHECLR, anche perché su 68040 è presente il copyback, un “potenziamento” della DATA CACHE che è veramente malvagio, tanto che non funzionano nemmeno alcuni programmi compilati in linguaggio C. Per questo ogni tanto fatevi un “ClearMyCache”, che non fa male.

11. Un altro problema di incompatibilità che ho riscontrato è quello degli interrupt su A4000. Notavo che molte demo, anche AGA, che funzionavano benissimo su A1200, su A4000 suonavano la musica a doppia velocità, andavano a scatti e a volte inchiodando il sistema. Mi sono reso conto che questo avveniva per una dimenticanza, che come tutte le dimenticanze si fanno notare quando il codice gira su A4000. Date un'occhiata a questo interrupt di livello 3 (il \$6c per intenderci):

```

1 INTERRUPT:
2     MOVEM.L D0-D7/A0-A6, -(SP)
3     BSR.W ROUTINE1
4     BSR.W ROUTINE2
5     BSR.W ROUTINE3
6     BSR.W ROUTINE4
7     MOVEM.L (SP)+, D0-D7/A0-A6
8 NOINT:
9     move.w #$20, $dff09c    ; INTREQ - vertb (bit 5 - $20 = %100000)
10    rte

```

Cosa è che non va?? Badate che è stato messo bene in VBR+\$6c, per cui viene eseguito regolarmente. La soluzione è: **manca il test del bit in INTREQR!!!!** Ecco come deve essere modificato:

```

1  INTERRUPT:
2      btst.b  #5,$dff01f      ; INTREQR - vertb int? (bit5)
3      beq.s   NOINT          ; non un vero interrupt VERTB
4      MOVEM.L D0-D7/A0-A6,-(SP)
5      BSR.W   ROUTINE1
6      BSR.W   ROUTINE2
7      BSR.W   ROUTINE3
8      BSR.W   ROUTINE4
9      MOVEM.L (SP)+,D0-D7/A0-A6
10 NOINT:
11      move.w  #$20,$dff09c    ; INTREQ - vertb (bit 5 - $20 = %100000)
12      rte

```

Effettivamente molto spesso su a500/a1200 l'interrupt funziona bene anche senza il controllo del bit in INTREQR, ma su A4000 va SEMPRE messo, altrimenti l'interrupt viene eseguito un miliardo di volte di troppo. Quindi MAI e poi MAI dimenticarsi di testare i bit che hanno generato un interrupt in INTREQR! Come abbiamo già visto, per ogni livello di interrupt c'è un bit del *\$dff01f* (INTREQR) da testare.

Un promemoria:

```

INT $64      LEVEL1  bits 0 (soft) ,1 (dskblk) ,2 (serial port tbe)

INT $68      LEVEL2  bit 3 (ports)

INT $6c      LEVEL3  bits 4 (copper) ,5 (verticalblank) ,6 (blitter)

INT $70      LEVEL4  bits 7 (aud0) ,8 (aud1) ,9 (aud2) ,10 (aud3)

INT $74      LEVEL5  bits 11 (serial port rbf) ,12 (disksyn)

INT $78      LEVEL6  bit 13 (external int)

```

Ricordatevi di fare sempre il btst all'inizio dell'interrupt, e in caso il bit non sia settato (beq) uscite senza eseguire le routines. All'uscita bisogna sempre agire sul *\$dff09c* per rimuovere la richiesta di interrupt, in pratica "segnamo" che l'interrupt è stato eseguito.

State attenti a non commettere l'errore, che pure io ho fatto in passato, di fare, ad esempio, un BTST #11,\$dff01f. In questo caso in realtà si testa il bit 11 di 8, ossia il bit 3 del *\$dff01f*. Ricorderete la vicenda del waitblit, per cui scrivere btst #6,\$dff002 è uguale a scrivere btst #14,\$dff002. Alcuni assembleri infatti assemblano anche istruzioni BTST su indirizzi con numero di bit superiore a 7, nonostante che siano inutili dato che scala di 8 il numero di tale bit. Altri assembleri, come il Devpac 3, danno errore e non permettono di assemblare tali inutili BTST .b. (NOTARE il .BYTE! = da 0 fino a 7 max!)

GUAI A CHI FA UNO DEGLI ERRORI DESCRITTI NELLA LEZIONE!!!!!!!!!!!!

LEZIONE 13 - OTTIMIZZAZIONE DEL CODICE ASSEMBLY

Autori: Fabio Ciucci, Ugo Erra

Ringraziamenti: Michael Glew, 2-Cool/LSD, Subhuman/Epsilon

Scrivere routine in assembly non significa necessariamente che il proprio codice girerà al massimo della velocità. Infatti non sempre il codice assembly può essere classificato come il meglio ottenibile in termini di velocità. Consideriamo infatti i numerosi demo che esistono in circolazione ed esattamente quelli che trattano grafica 3d, nella maggior parte delle volte (quasi sempre) le routine che stanno sotto ad effetti quali rotazioni, zoom, esplorazioni di mondi, etc, sono le stesse, ma la loro implementazione in codice assembly è differente, poiché ogni programmatore cerca di implementarle nel modo migliore possibile cioè in modo che girino al massimo della velocità. Ciò è realizzato con tecniche di ottimizzazione che ogni buon coder assembly deve sapere. Le tecniche sono numerose e sicuramente ci vuole un bel pò di tempo prima che si inizino ad utilizzare in modo del tutto naturale. Esistono vari tipi di ottimizzazione e molte di queste tecniche che andrò a spiegare sono valide per il 68000 ma le stesse sono anche inutili in microprocessori quali il 68040 o il 68060. La prima cosa che bisogna avere disponibile è una tavola dei cicli macchina di ogni istruzione del 68000, che troverete sintetizzata in questa lezione: dando un rapido sguardo a questa tavola vi potreste meravigliare osservando il “tempo” che impiega ogni istruzione ad essere eseguita, e forse fino a questo punto credevate che ogni istruzione fosse eseguita nello stesso tempo; ebbene vi sbagliavate!!! Infatti, come primo approccio, notate il tempo che impiega un’istruzione di moltiplicazione (MULU) rispetto ad una di addizione (ADD), e capirete immediatamente per quale motivo l’ottimizzazione è importante:

ADD ; tempo di esecuzione: dai 6 ai 12+ cicli di clock

MULS ; tempo di esecuzione: 70+ cicli di clock

Quindi, si capisce facilmente come ottimizzare questa istruzione:

lento: MULU.W #2,D0 ; 70+ cicli

ottimizzato: ADD.W d0,d0 ; 6+ cicli

Vi anticipo che le moltiplicazioni e le divisioni sono le due istruzioni più lente. Vediamo un approssimativo elenco delle istruzioni ordinate dalle più veloci alle più lente: (i cicli sono nella migliore ipotesi!)

EXT, SWAP, NOP, MOVEQ ; 4 cicli -> le più veloci!

TST, BTST, ADDQ, SUBQ, AND, OR, EOR ; 4 + indirizzamento, velocine...

MOVE, ADD, SUB, CMP, LEA ; 4+ indirizzamento, ma spesso gli
; indirizzamenti sono "pesanti" da eseguire

Poi abbiamo BCLR/BCHG/BSET con 8+, LSR/LSL/ASR/ASL/ROR/ROL con 6 +2n, dove n è il numero di shifts da fare, infine abbiamo:

MULS/MULU ; 70+ !
DIVU ; 140+ !!
DIVS ; 158+ !!!

Occorre anche ricordare che:

BEQ,BNE,BRA... ; 10
DBRA ; 10
BSR ; 18
JMP ; 12
RTS ; 16
JSR ; 16/20

Quindi, attenzione a non fare troppe chiamate alle subroutine, perché ogni BSR + RTS per tornare vi mangia 18+16=34 cicli almeno! Mettete sempre nel loop principale le subroutines corte, è uno spreco perdere 34 cicli di BSR + RTS per eseguire una manciata di istruzioni!

```

1  EXAMPLE:
2  BSR.S  ROUT1
3  BSR.S  ROUT2
4  BSR.S  ROUT3
5  RTS
6
7  ROUT1:
8  MOVE.W d0,d1
9  RTS
10 ROUT2:
11 MOVEQ  #0,d2
12 MOVEQ  #0,d3
13 RTS
14 ROUT3:
15 LEA    label1(PC),A0
16 RTS

```

Versione che salva 34*3= 96 cicli:

```

1  EXAMPLEFIX:
2  MOVE.W d0,d1
3  MOVEQ  #0,d2
4  MOVEQ  #0,d3
5  LEA    label1(PC),A0
6  RTS

```

Oltre all'istruzione in sé, conta anche il modo di indirizzamento usato. Per esempio:

```

1  MOVE.L  (a0),d0

```

è più lento di:

```

1  MOVE.L  $12(a0,d1.w),LABEL1

```

Eppure si tratta sempre di istruzioni MOVE. Comunque può apparirvi molto logico il perché della maggior lentezza della seconda istruzione rispetto alla prima: il processore deve calcolare l'offset sommando ad A0 il valore di D1 più il \$12, poi fare la copia, e dove? In memoria, ad una label, anziché in un registro, cosa ben più lenta dato che i registri sono DENTRO il processore, mentre la memoria è fuori, e per raggiungerla il dato deve passare dai fili della scheda madre!!!!

13.1 Ottimizzazioni di primo livello: lo “scambio” e la “scelta” di istruzioni

Ecco i modi di indirizzamento ordinati dal più veloce al più lento: NOTA: i numeri dopo il ; sono i cicli di clock da aggiungere al tempo usato dall'istruzione, nei casi di byte-word/longword

Registro dati diretto	Dn/An	; 0
Registro indirizzi indiretto (o con Post-Incremento)	(An)/(An)+	; 4/8
Immediato	#x	; 4/8
Registro indirizzi indiretto con Pre-Decremento	-(An)	; 6/10
Registro indirizzi indiretto con Offset (max 32767)	w(An)	; 8/12
Absoluto corto	w	; 8/12
Program Counter con Offset (calcolato dall'asmone)	w(PC)	; 8/12
Program Counter con Offset e Indice	b(PC,Rx)	; 10/14
Registro indirizzi indiretto con Offset e Indice	b(An,Rx)	; 10/14
Absoluto lungo	l	; 12/16

Come si può notare, mentre un `MOVE.L LABEL1, LABEL2` solo di indirizzamento porta via $16+16 = 32$ cicli, un `MOVE.L #1234, d0` porta via solo $8+0 = 8$ cicli. Appare evidente come le istruzioni `.W` siano più veloci di quelle `.L`, per esempio l'indirizzamento (An), se `.W` impiega 4 cicli, se `.L` 8 cicli!

Comunque questi esempi sono **molto** indicativi, infatti anche studiando con le tabelle alla mano è difficile calcolare veramente il tempo di esecuzione della routine. Ma saremo sempre sicuri che il BSR è più veloce del JSR, che l'ADDQ è più veloce dell'ADD, e soprattutto che ogni volta che si riesce a sostituire un MULU/DIVU/MULS/DIVS con qualcos'altro abbiamo certamente velocizzato il tutto!

Qua stiamo parlando di “cambi di istruzione”, ossia di piccole modifiche fatte sostituendo istruzioni lente con altre più veloci. Però l'arte delle ottimizzazioni, vera regina della scena demo, comporta anche l'utilizzazione di una tabella “precalcolata” invece di implementare una mega funzione che dà gli stessi risultati, e altre infinite cose.

Però c'è anche il rovescio della medaglia: il codice megaottimizzato con tabelle e altri stratagemmi spesso diviene meno leggibile e capibile, e meno “modificabile”. Quindi, state attenti ad evitare l'errore nel quale molti di noi sono caduti, ossia nel voler ottimizzare la routine prima di averla finita, passo passo, ad ogni costo. Questo non fa che rallentare lo sviluppo della routine in questione, specialmente se si è alle prime armi, infatti a che serve una routine megaottimizzata che calcola la prospettiva, se non riusciamo più a scriverci “intorno” la routine di disegno e rotazione del solido? O addirittura non capiamo più come mai sta funzionando?

Mai passare sotto ottimizzazione una routine che non è completamente terminata e funzionante; inoltre, una volta che è pronta per l'ottimizzazione, ricordarsi di tenere le copie dei listati dei vari passaggi dell'ottimizzazione, in quanto spesso occorre “tornare indietro” e modificare qualcosa!!! Poi riottimizzeremo la versione modificata!

Questo avvertimento vi suonerà strano, perché sembra che un listato, una volta ottimizzato, diventi irriconoscibile e incomprensibile anche per l'autore. Ebbene, se è *molto* ottimizzato, questo può succedere! Comunque ricordatevi che le ottimizzazioni vanno effettuate in parti del listato che effettivamente richiedono molto tempo per essere eseguite: ad esempio, è inutile stare ad ottimizzare una routine che viene eseguita una sola volta alla startup, o una sola volta per

fotogramma. Le prime routines da ottimizzare sono quelle che vengono eseguite molte volte per fotogramma, ossia quelle nei loop DBRA, o comunque in cicli vari. Ad esempio, vediamo questo listatino:

```

1  Bau:
2  cmp.w  #$ff,$dff006      ; Aspetta il Wblank
3  bne.s  Bau
4  bsr.s  routine1
5  bsr.s  routine2
6  btst   #6,$bfe001        ; Aspetta il mouse
7  bne.s  Bau
8  rts
9
10 Routine1:
11 move.w #label2,d6
12 move.w d0,d1
13 move.w d2,d3
14 and.w  d4,d5
15 rts
16
17 Routine2:
18 move.w #200,d7
19 lea    label2(PC),a0
20 lea    label3(PC),a1
21 loop1:
22 move.w (a0)+,d0
23 move.w (a0)+,d1
24 add.w  d0,d5
25 add.w  d0,d6
26 move.w d5,(a1)+
27 move.w d5,(a2)+
28 dbra  d7,loop1
29 rts

```

In questo caso, appare evidente che il 99% del tempo lo si perde eseguendo 200 volte il loop della routine2. Di conseguenza, se si ottimizzasse questo loop rendendolo veloce il doppio, l'intero programma girerebbe al doppio della velocità, mentre se si facesse andare anche al triplo o al quadruplo della velocità la routine2, non si noterebbe nemmeno la differenza!!!! Per vedere quante “linee raster” occupa una routine, basta usare il vecchio sistema di cambiare colore all'inizio della routine, e cambiarlo nuovamente alla sua fine. In questo modo la “striscia” del colore cambiato indicherà il tempo in “linee video” usato per l'esecuzione:

```

1  Bau:
2  cmp.w  #$90,$dff006      ; Aspetta il Wblank
3  bne.s  Bau
4  bsr.s  routine1
5  move.w #$F00,$dff180     ; Color0: ROSSO
6  bsr.s  routine2
7  move.w #$000,$dff180     ; Color0: NERO
8  btst   #6,$bfe001        ; Aspetta il mouse
9  bne.s  Bau
10 rts

```

In questo caso, aspettiamo la linea \$90, verso la metà dello schermo, facciamo eseguire la routine1, poco importante, poi cambiamo colore (rosso), facciamo eseguire la routine2, e ricambiamo colore (nero). Apparirà a video una strisciata rossa... quello è il “tempo” in cui viene eseguita routine2. Per vedere se si migliora o si peggiora la velocità, basterà vedere se la strisciata si allunga o si accorcia. Alcuni maniaci (come il mio amico hedgehog), appiccicano un pezzo di nastro adesivo sul monitor all'altezza dell'ultima linea colorata, in modo da notare ad ogni modifica ogni lieve miglioramento o peggioramento. Io, personalmente, ci metto un dito o vado ad occhio... fate voi! Abbiamo comunque già visto questo sistema nella lezione sul blitter e in Lezione11n1.s e seguenti, per “visualizzare” il tempo aspettato tramite i chip CIAA/CIAB. A proposito, potreste usare anche i timer per calcolare i tempi “numericamente”, ma il sistema del cambio di colore è più immediato.

Ma prima cominciamo con le ottimizzazioni elementari, che occorrerebbe saper fare “in diretta” mentre si scrive. La cosa più semplice è sapere quale istruzione scegliere tra quelle possibili, quando si vuol fare un dato compito. Infatti la stessa operazione si può fare in più modi! Ad esempio, vediamo questo listato:

```
1 lea    LABEL1,a0
2 move.l 0(a0),d0
3 move.l 2(a0),d1
4 ADD.W  #5,d0
5 SUB.W  #5,d1
6 MULU.W #2,d0
7 MOVE.L #30,d2
8 RTS
```

La stessa cosa, si può fare scegliendo queste istruzioni:

```
1 lea    LABEL1(PC),a0    ; Indirizzamento (PC) più veloce
2 move.l (a0),d0          ; Non occorre l'offset 0!!
3 move.l 2(a0),d1         ; Questa si lascia così
4 ADDQ.W #5,d0            ; numero minore di 8, si può usare ADDQ!
5 SUBQ.W #5,d1            ; idem, per SUBQ!
6 ADD.W  d0,d0            ; salvo 60 cicli!! D0*2 è uguale a D0+D0!!!
7 MOVEQ  #30,d2          ; numero minore di 127, posso usare MOVEQ!
8 RTS
```

La routine è mooolto più veloce, ed è sempre leggibilissima. Quindi, la prima cosa da imparare è stare attenti ad usare le istruzioni Quick dedicate come ADDQ/SUBQ/MOVEQ nel caso il numero sia piccolo abbastanza, a togliere le moltiplicazioni e le divisioni quando possibile, ad usare indirizzamenti relativi al (PC) o a registri+offset, anziché a LABEL nude e crude, eccetera. Con un pò di esperienza, vi verrà naturale scegliere le istruzioni più veloci, e scriverete al primo colpo listati come il secondo, anziché listati come il primo presentato, che spero già ora non scriviate!!!! Ecco un altro esempio di ottimizzazione “a scambio” di istruzioni:

```
1 Move.l #3,d0            ; 12 cicli
2 Clr.l  d0               ; 6 cicli
3 Add.l  #3,a0            ; 16 cicli
4 ;
5 Move.l #5,Label         ; 28 cicli
```

Versione ottimizzata “a scambio”:

```
1 Moveq  #3,d0            ; 4 cicli
2 Moveq  #0,d0            ; 4 cicli
3 Addq.w #3,a0            ; 4 cicli
4 ;
5 Moveq  #5,d0            ; 4 cicli
6 Move.l d0,Label         ; 20 cicli, totale 24 cicli
```

Potrei continuare ancora per molto con esempietti del genere, ma non dovete conoscere a memoria tutti i casi possibili, naturalmente! Occorre piuttosto capire “il metodo”, la filosofia del coding ottimizzato. Esistono, ad esempio, tecniche per velocizzare il caricamento di valori a 32 bit nei registri:

```
1 move.l #$100000,d0      ; 12 cicli
```

Versione ottimizzata:

```
1 moveq  #10,d0           ; 4 cicli
2 Swap   d0               ; 4 cicli, totale 8 cicli
```

Altra cosa **importantissima** è che l'accesso alla memoria (ossia alle label) è molto più LENTO dell'accesso ai registri dati ed indirizzi. Quindi, è una buona abitudine il tendere ad usare tutti i registri e badare di toccare il meno possibile le label. Ad esempio il listato:

```
1 MOVE.L #200,LABEL1
2 MOVE.L #10,LABEL2
3 ADD.L  LABEL1,LABEL2
```

Si può ottimizzare MOLTISSIMO scrivendo:

```
1  move.l  #200,d0
2  moveq   #10,d1
3  add.l   d0,d1
```

Non fate caso alla stupidità dell'esempio, ma al fatto che mentre nel primo abbiamo fatto 4 accessi alla lentissima RAM, facendo passare i dati per i fili aggrovigliati della scheda madre, nel secondo caso tutto si è svolto all'interno della CPU, turbizzando il tutto. Se finite i registri dati, usate anche i registri indirizzi per tenere dati, piuttosto che accedere a label! Inoltre, se è possibile, usate istruzioni .W anziché .L, per esempio il listato di prima si potrebbe riottimizzare in:

```
1  move.w  #200,d1
2  moveq   #10,d0
3  add.w   d0,d1
```

In questo caso le istruzioni occupano 8 cicli anziché 12...e non è poco! State però attenti che la word alta sia azzerata e/o non serva mai!!

Comunque, le ottimizzazioni “a scambio” più proficue sono quelle che eliminano istruzioni di moltiplicazione (70 cicli) e di divisione (158 cicli), e si può dire che è nata una scienza in proposito. Il caso più semplice è quando dobbiamo dividere o moltiplicare per numeri che sono potenze di 2, perché possiamo adoperare le istruzioni di shift che impiegano come cicli macchina esattamente:

```
1  Lsl.w   6+2n          ; n = numero di shifts
2  Asr.w   6+2n
3  Lsr.l   8+2n
4  Asr.l   8+2n
```

Qui n sta ad indicare il numero di bit, ed il numero dei cicli è riferito a quando si adoperano i registri. La regola da seguire in genere è la seguente: (per MULS o MULU)

Nota: alle volte ci vuole un EXT.L D0 prima degli ASL che sostituiscono i MULS, mentre prima di quelli che sostituiscono i MULU può servire una pulizia della word alta con swap d0, clr.w d0, swap d0.

```
1  MULS.w  #2,d0          | ADD.L d0,d0 ; mi pare chiaro!
2
3  MULS.w  #4,d0          | ADD.L d0,d0 ; anche questo!
4  | ADD.L d0,d0
5
6  MULS.w  #8,d0          | ASL.l #3,d0 ; da 8 a 256 conviene l'asl
7  MULS.w  #16,d0         | ASL.l #4,d0
8  MULS.w  #32,d0         | ASL.l #5,d0
9  MULS.w  #64,d0         | ASL.l #6,d0
10 MULS.w  #128,d0        | ASL.l #7,d0
11 MULS.w  #256,d0        | ASL.l #8,d0
```

Se ci sono problemi coi MULU, si potrebbe pulire la word alta:

```
1  mulu.w  #n,dx -> swap dx      ; n is 2^m, 2..2^8
2  clr.w   dx                ; (2,4,8,16,32,64,128,256)
3  swap    dx
4  asl.l   #m,dx
```

Per il muls può bastare mettere un ext.l prima dell'asl.

```
1  muls #n,dx -> ext.l dx      ; n is 2^m, 2..2^8
2  asl.l #m,dx
```

Mentre per le DIVISIONI:

```
1  DIVS.w  #2,d0          | ASR.L #1,d0 ; attenzione: IGNORA IL RESTO!!!!!!
2  DIVS.w  #4,d0          | ASR.L #2,d0
3  DIVS.w  #8,d0          | ASR.L #3,d0
4  DIVS.w  #16,d0         | ASR.L #4,d0
```

```

5 DIVS.w #32,d0 | ASR.L #5,d0
6 DIVS.w #64,d0 | ASR.L #6,d0
7 DIVS.w #128,d0 | ASR.L #7,d0
8 DIVS.w #256,d0 | ASR.L #8,d0
9 DIVU.w #2,d0 | LSR.L #1,d0 ; attenzione: IGNORA IL RESTO!!!!!!
10 DIVU.w #4,d0 | LSR.L #2,d0
11 DIVU.w #8,d0 | LSR.L #3,d0
12 DIVU.w #16,d0 | LSR.L #4,d0
13 DIVU.w #32,d0 | LSR.L #5,d0
14 DIVU.w #64,d0 | LSR.L #6,d0
15 DIVU.w #128,d0 | LSR.L #7,d0
16 DIVU.w #256,d0 | LSR.L #8,d0

```

Come sapete, dopo una divisione nella word bassa rimane il risultato e in quella alta il resto; se sostituite il DIVS/DIVU con uno shift invece avrete il risultato nella word bassa e la word alta azzerata. . . quindi **non è la stessa cosa**, state attenti! Nel caso peggiore in cui $n=8$ otterrete un numero di cicli esattamente di $6+2*8=22$ cicli per le word e $8+2*8=24$ cicli per le longword, quindi il risparmio è garantito. Inoltre sappiate che su un 68020 il numero dei cicli per le istruzioni di shift è lo stesso indipendentemente dal numero di bit da spostare. Inoltre tenete anche presente l'istruzione Swap, che impiega 4 cicli ad essere eseguita, poiché ci può far comodo in molte situazioni in cui il numero di bit da spostare è consistente. Vediamo a questo proposito una serie di esempi:

```

1 ; Shift di 9 bit a sinistra
2
3 Lsl.l #8,d0
4 Add.l d0,d0
5
6 ; Shift di 16 bit a sinistra
7
8 Swap d0
9 Clr.w d0
10
11 ; Shift di 24 bit a sinistra
12
13 Swap d0
14 Clr.w d0
15 Lsl.l #8,d0
16
17 ; Shift di 16 bit a destra
18
19 Clr.w d0
20 Swap d0
21
22 ; Shift di 24 bit a destra
23
24 Clr.w d0
25 Swap d0
26 Lsr.l #8,d0

```

Come potete vedere le tecniche per shiftare non mancano e se ne possono ricavare moltissime, come sempre spetta a voi entrare nell'ottica giusta e cercare di fare l'ottimizzazione che cercate. Quindi per potenze di 2 non avete grossi problemi a moltiplicare e a dividere in un tempo decente. I problemi potrebbero sorgere nel caso in cui il numero non sia una potenza di due; in effetti questo è vero, ma per molti valori possiamo ancora aggirare il problema. Infatti consideriamo il caso in cui dobbiamo moltiplicare il valore, contenuto in un registro, per 3: ebbene, pensate al fatto che dovete eseguire una espressione del genere $3*x$, che potete anche scrivere $2*x+x$. A questo punto avete risolto il vostro problema perché il vostro codice sarà:

```

1 Move.l d0,d1
2 Add.l d0,d0 ; d0=d0*2
3 Add.l d1,d0 ; d0=(d0*2)+d0

```

Consideriamo un altro caso ad esempio per $n=5$, allora abbiamo $5*x$, ossia $4*x+x$: come codice avremo che:

```

1 Move.l d0,d1
2 Asl.l #2,d0 ; d0=d0*4
3 Add.l d1,d0 ; d0=(d0*4)+d0

```

Consideriamo infine un altro caso in cui $n=20$, allora abbiamo $20*x$, ma $20*x = 4*(5*x) = 4*(4*x+x)$

```

1 Move.l d0,d1
2 Asl.l #2,d0 ; d0=d0*4
3 Add.l d1,d0 ; d0=(d0*4)+d0
4 Asl.l #2,d0 ; d0=4*((d0*4)+d0)

```

In breve possiamo tentare di fare una cosa del genere, se scomponendo il numero in fattori primi notiamo che ci sono molti 2; ma fatevi sempre un conticino sul numero dei cicli per vedere se vi conviene oppure no. Molti di voi potrebbero meravigliarsi nel vedere qui trattato il modo per poter ottimizzare una semplice MULU o DIVU, ma pensate i casi in cui queste si trovino in cicli, in questo caso queste tecniche sono realmente molto utili, comunque anche se la MULU non si trova in un ciclo, che vi costa sostituirla con qualcosa di migliore? Poiché siamo in argomento parliamo molto brevemente delle implementazioni delle espressioni in Assembly. Quello che vi dirò non è niente di particolare ma spesso non si presta attenzione ad un fatto banale. Quando dobbiamo implementare una funzione, di solito quello che facciamo è di caricare i valori nei registri ed effettuare tutte le operazioni. In generale, per risparmiare tempo macchina nella valutazione della funzione, conviene utilizzare i metodi di raccoglimento che si imparano alle superiori, infatti consideriamo una espressione banale:

$a*d0+b+d1+a*d3+b*d5$

può essere scritta come:

$a*(d0+d3)+b*(d1+d5)$

In questo modo risparmiamo due moltiplicazioni.

Dato che per saper scegliere la giusta istruzione basta conoscere in ogni coppia di istruzioni equivalenti quale è la più veloce, vi presento una tabella simile a quella alla fine di 68000-2.txt, con istruzioni “lente”, ed equivalenti “veloci” da usare:

ISTRUZIONE esempio	EQUIVALENTE, MA PIÙ VELOCE
add.X #6,XXX	addq.X #6,XXX (massimo 8)
sub.X #7,XXX	subq.X #7,XXX (massimo 8)
MOVE.X LABEL,XX	MOVE.X LABEL(PC),XX (se nella stessa SECTION)
LEA LABEL,AX	LEA LABEL(PC),AX (se nella stessa SECTION)
MOVE.L #30,d1	moveq #30,d1 (min #-128, max #+127)
CLR.L d4	MOVEQ #0,d4 (solo per i registri dati)
ADD.X/SUB.X #12000,a3	LEA (+/-)12000(a3),A3 (min -32768, max 32767)
MOVE.X #0,XXX	CLR.X XXX ; muovere #0 è stupido!
CMP.X #0,XXX	TST.X XXX ; il TST dove lo lasci?
Per azzerare un reg. Ax	SUB.L A0,A0 ; meglio di "LEA 0,a0".
JMP/JSR XXX	BRA/BSR XXX (Se XXX è vicino)
MOVE.X #label,AX	LEA label,AX (solo registri indirizzi!)
MOVE.L 0(a0),d0	MOVE.L (a0),d0 (togli l'offset se è 0!!!)
LEA (A0),A0	HAHAHAHA! ; Toglila, non ha effetti!!
LEA 4(A0),A0	ADDQ.W #4,A0 ; fino ad 8
addq.l #3,a0	addq.w #3,a0 ; Solo reg. indirizzi, max 8
Bcc.W label	Bcc.S label ; Beq,Bne,Bsr... dist. <128

Per le moltiplicazioni e divisioni di multipli di 2 convertite in ASL / ASR vedete la tabella sopra. Seguono dei casi particolari per cambiare MULS / MULU in qualcos'altro. NOTA: Se si tratta di un

MULS, spesso occorre aggiungere un `ext.l dx` come prima istruzione, per estendere il segno a longword.

```

mul*.w #3,dx -> move.l dx,ds
add.l dx,dx
add.l ds,dx
-----
mul*.w #5,dx -> move.l dx,ds
asl.l #2,dx
add.l ds,dx
-----
mul*.w #6,dx -> add.l dx,dx
move.l dx,ds
add.l dx,dx
add.l ds,dx
-----
mul*.w #7,dx -> move.l dx,ds
asl.l #3,dx
sub.l ds,dx
-----
mul*.w #9,dx -> move.l dx,ds
asl.l #3,dx
add.l ds,dx
-----
mul*.w #10,dx -> add.l dx,dx
move.l dx,ds
asl.l #2,dx
add.l ds,dx
-----
mul*.w #12,dx -> asl.l #2,dx
move.l dx,ds
add.l dx,dx
add.l ds,dx
-----
mulu.w #12,dx -> swap dx          ; HEI! spesso occorre azzerare la word
clr.w dx          ; alta per i MULU... considerate questo anche
swap dx ; per i mulu #3, #5, #6....

asl.l #2,dx          ; normale mulu #12
move.l dx,ds
add.l dx,dx
add.l ds,dx
-----

```

Se dovete azzerare la word alta dei registri molte volte, potete usare anche:

```

1  move.l #$0000FFFF,ds ; serve 1 registro per tenere $FFFF
2
3  and.l ds,dx          ; questo è più veloce dello swappaggio, ma
4  ; richiede un registro che contenga $0000FFFF,
5  ; altrimenti "AND.L #$FFFF,dx" non è più
6  ; veloce...

```

In sintesi, ricordatevi che in caso di MULS, dato che è SIGNED, può essere necessario eseguire un EXT.L all’inizio. Invece, nel caso dei MULU, può essere necessario azzerare la word alta del registro.

Ora degli scambi “composti”:

```

asl.x #2,dy -> add.x dy,dy
add.x dy,dy
-----
asl.l #16,dx -> swap dx

```

```

clr.w dx
-----
asl.w #2,dy -> add.w dy,dy
add.w dy,dy
-----
asl.x #1,dy -> add.x dy,dy
-----
asr.l #16,dx -> swap dx
ext.l dx
-----
bsr label -> bra label
rts
-----
clr.x n(ax,rx) -> move.x ds,n(ax,rx)      ; ds deve essere 0, naturalmente!
-----
lsl.l #16,dx -> swap dx
clr.w dx
-----
move.b #-1,(ax) -> st (ax)
-----
move.b #-1,dest -> st dest
-----
move.b #x,mn -> move.w #xy,mn
move.b #y,mn+1
-----
move.x ax,ay -> lea n(ax),ay              ; -32767 <= n <= 32767
add.x #n,ay
-----
move.x ax,az -> lea n(ax,ay),az           ; az=n+ax+ay, n<=32767
add.x #n,az
add.x ay,az
-----
sub.x #n,ax -> lea -n(ax),ax              ; -32767 <= n <= -9, 9 <= n <= 32767
-----

```

A questo punto vedetevi il tempo di esecuzione delle varie istruzioni. Al tempo di esecuzione dell'istruzione, va aggiunto il tempo speso per i vari indirizzamenti, il cui tempo di esecuzione è stato visto prima. State attenti al fatto che si tratta dei tempi di esecuzione del normale 68000! Per esempio, nel 68040 i MULS/MULU sono implementati via hardware e prendono pochi cicli!

>>>	MOVE.B e MOVE.W												<<<
	DESTINAZIONE												
SORG.	Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xn)*	(xxx.W)	(xxx.L)				
Dn / An	4	4	8	8	8	12	14	12	16				
(An)	8	8	12	12	12	16	18	16	20				
(An)+	8	8	12	12	12	16	18	16	20				
-(An)	10	10	14	14	14	18	20	18	22				
(d16,An)	12	12	16	16	16	20	22	20	24				
(d8,An,Xn)*	14	14	18	18	18	22	24	22	26				
(xxx).W	12	12	16	16	16	20	22	20	24				
(xxx).L	16	16	20	20	20	24	26	24	28				
(d16,PC)	12	12	16	16	16	20	22	20	24				
(d8,PC,Xn)*	14	14	18	18	18	22	24	22	26				
#(data)	8	8	12	12	12	16	18	16	20				

+-----+-----+-----+-----+-----+-----+-----+-----+
 * La grandezza del registro indice (Xn) (.w o .l) non cambia la velocità.

>>>		MOVE.L										<<<	
		DESTINAZIONE											
SORG.													
		Dn	An	(An)	(An)+	-(An)	(d16,An)	(d8,An,Xn)*	(xxx.W)	(xxx).L			
Dn o An		4	4	12	12	12	16	18	16	20			
(An)		12	12	20	20	20	24	26	24	28			
(An)+		12	12	20	20	20	24	26	24	28			
-(An)		14	14	22	22	22	26	28	26	30			
(d16,An)		16	16	24	24	24	28	30	28	32			
(d8,An,Xn)*		18	18	26	26	26	30	32	30	34			
(xxx).W		16	16	24	24	24	28	30	28	32			
(xxx).L		20	20	28	28	28	22	34	32	36			
(d,PC)		16	16	24	24	24	28	30	28	32			
(d,PC,Xn)*		18	18	26	26	26	30	32	30	34			
#(data)		12	12	20	20	20	24	26	24	28			
* La grandezza del registro indice (Xn) (.w o .l) non cambia la velocità.													

* La grandezza del registro indice (Xn) (.w o .l) non cambia la velocità.

Ed ora le altre istruzioni.

Note:

- Operando Immediato
 An - Registro indirizzi
 Dn - Registro Dati
 ea - Un operando specificato da un Indirizzo effettivo
 M - Indirizzo effettivo
 + - Aggiungere il tempo speso a calcolare l'indirizzo (indirizzamento)

Istruzione	Size	op<ea>,An 1	op<ea>,Dn	op Dn,<M>
ADD/ADDA	Byte,Word	8+	4+	8+
	Long	6+	6+	12+
AND	Byte,Word	-	4+	8+
	Long	-	6+	12+
CMP/CMPPA	Byte,Word	6+	4+	-
	Long	6+	6+	-
DIVS	-	-	158+	-
DIVU	-	-	140+	-
EOR	Byte,Word	-	4	8+
	Long	-	8	12+

MULS/MULU	-	-	70+	-
	Byte,Word	-	4+	8+
OR	Long	-	6+	12+
	Byte,Word	8+	4+	8+
SUB	Long	6+	6+	12+

Istruzione	Size	op #,Dn	op #,An	op #,M
ADDI	Byte,Word	8	-	12+
	Long	16	-	20+
ADDQ	Byte,Word	4	4	8+
	Long	8	8	12+
ANDI	Byte,Word	8	-	12+
	Long	14	-	20+
CMPI	Byte,Word	8	-	8+
	Long	14	-	12+
EORI/SUBI	Byte,Word	8	-	12+
	Long	16	-	20+
MOVEQ	Long	4	-	-
ORI	Byte,Word	8	-	12+
	Long	16	-	20+
SUBQ	Byte,Word	4	8	8+
	Long	8	8	12+

Istruzione	Size	Register	Memory
NBCD	Byte	6	8+
CLR/NEG	Byte,Word	4	8+
NEGX/NOT	Long	6	12+
Scc	Byte,False	4	8+
	Byte,True	6	8+
TAS	Byte	4	14+
TST	Byte,Word,Long	4	4+

LSR/LSL	Byte, Word	6 + 2n	8+
ASR/ASL			
ROR/ROL	Long	8 + 2n	-
ROXR/ROXL			

nota: n è il numero di shifts!

Bit Manipulation Istruzione Execution Times

Istruzione	Size	Dynamic		Static	
		Register	Memory	Register	Memory
BCHG/BSET	Byte	-	8+	-	12+
	Long	8	-	12	-
BCLR	Byte	-	8+	-	12+
	Long	10	-	14	-
BTST	Byte	-	4+	-	8+
	Long	6	-	10	-

Istruzione	Displacement	Branch Taken	Branch Not Taken	
Bcc	Byte	10	8	
	Word	10	12	
BRA	Byte	10	-	
	Word	10	-	
BSR	Byte, word	18	-	
DBcc	cc true	-	12	
	cc false, Count Not Expired	10	-	
	cc false, Counter Expired	-	14	

Ins. Sz (An) (An)+ - (An) (d16, An) (d8, An, Xn)+ (x).W (x).L (d16, PC) (d8, PC, Xn)*
JMP - 8 - - 10 14 10 12 10 14
JSR - 16 - - 18 22 18 20 18 22
LEA - 4 - - 8 12 8 12 8 12
PEA - 12 - - 16 20 16 20 16 20
Word 12+4n 12+4n - 16+4n 18+4n 16+4n 20+4n 16+4n 18+4n

```

|MOVEM+---+---+---+---+---+---+---+---+---+---+
|M->R |Long|12+8n|12+8n| _ |16+8n| 18+8n |16+8n|20+8n| 16+8n | 18+8n |
| | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+
| | |Word| 8+4n| _ |8+4n|12+4n| 14+4n |12+4n|16+4n| _ | _ |
| | | | | | | | | | | | | | | |
|MOVEM+---+---+---+---+---+---+---+---+---+---+
|R->M |Long| 8+8n| _ |8+8n|12+8n| 14+8n |12+8n|16+8n| _ | _ |
| | | | | | | | | | | | | | | |
+---+---+---+---+---+---+---+---+---+---+

```

nota: n è il numero di registri da muovere.

```

EXT/SWAP/NOP    4
EXG             6
UNLK            12
LINK/RTS        16
RTE             20

```

Considerate infine che le exceptions richiedono 44 cicli se si tratta di un interrupt, 34 se si tratta di un TRAP. Più 20 per l'RTE!!! Mi raccomando di commentare **sempre** una ottimizzazione, ad esempio, supponiamo vogliate ottimizzare questa routine:

```

1  movem.l  label1(PC),d1-d4
2  mulu.w   #16,d1
3  mulu.w   #3,d2
4  muls.w   #5,d3
5  divu.w   #8,d4
6  rts

```

Ottimizzandola, il risultato sarebbe:

```

1  movem.l  label1(PC),d1-d4
2  asl.l    #4,d1          ; mulu.w #16,d1
3  move.l   d2,d5          ; \
4  add.l    d2,d2          ; > mulu.w #3,d2
5  add.l    d5,d2          ; /
6  move.l   d3,d5          ; \
7  asl.l    #2,d3          ; > muls.w #5,d3
8  add.l    d5,d3          ; /
9  asr.l    #3,d4          ; divu.w #8,d4
10 rts

```

Oltre ad aver usato il registro d5, abbiamo reso più difficile da leggere il listato. A prima vista, se non avessimo messo i commenti, si capirebbe cosa succede ai registri d1,d2,d3 e d4? E immaginatevi se avessimo dovuto anche pulire la word alta prima dei MULU e estendere prima dei MULS:

```

1  movem.l  label1(PC),d1-d4
2  swap     d1
3  clr.w    d1
4  swap     d1
5  asl.l    #4,d1
6  swap     d2
7  clr.w    d2
8  swap     d2
9  move.l   d2,d5
10 add.l    d2,d2
11 add.l    d5,d2
12 ext.l    d3
13 move.l   d3,d5
14 asl.l    #2,d3
15 add.l    d5,d3
16 asr.l    #3,d4
17 rts

```

Oppure, si può azzerare la word alta nel modo più veloce:

```

1  move.l  #$FFFF,d6
2  ...
3  movem.l  label1(PC),d1-d4
4  and.l  d6,d1
5  asl.l  #4,d1
6  and.l  d6,d2
7  move.l  d2,d5
8  add.l  d2,d2
9  add.l  d5,d2
10 ext.l  d3
11 move.l  d3,d5
12 asl.l  #2,d3
13 add.l  d5,d3
14 asr.l  #3,d4
15 rts

```

Se tornate al vostro listato, dopo 1 mese dalla scrittura, quanto ci mettereste a capire che tutte queste istruzioni incomprensibili non fanno altro che 3 moltiplicazioni e una divisione? **Ci mettereste molto**, o dovrete addirittura cancellare il listato e ripartire da capo in caso di modifica. Non ho messo i commenti a quest’ultima versione proprio per farvi capire come sia FONDAMENTALE mettere i commenti alle ottimizzazioni, come nel listato precedente. Quindi: **commentate sempre le ottimizzazioni!!!!!!!!!!!!**

Altro esempio: vedetevi queste 3 istruzioni:

```

1  move.l  a1,a0
2  add.w  #80,a0
3  add.l  d0,a0

```

La stessa cosa si può fare con:

```

1  lea     80(a1,d0.l),a0 ; oppure d0.w se basta la word bassa di d0.

```

13.2 Ottimizzazioni di secondo livello: il “tabellamento”

Parliamo ora delle tabelle, un argomento tra i più importanti per l’Ottimizzazione, quella con la O maiuscola, che permette di andare più veloci di qualsiasi compilatore C, BASIC, eccetera. Le tabelle per l’ottimizzazione sono “simili” a quelle usate per contenere le coordinate dell’ondeggiamento degli sprite o altro, che abbiamo visto nelle lezioni precedenti: in quel caso si può dire che “precalcolavamo” le varie posizioni che avrebbero assunto gli oggetti, ma qua la tabella viene usata per “precalcolare” i risultati di una data moltiplicazione, divisione, o di intere funzioni matematiche, quindi il caso è un pò diverso. Facciamo un esempio concreto. Supponiamo di avere una routine che elabora una serie di valori compresi tra 0 e 100, ed a un certo punto bisogna eseguire una moltiplicazione per una costante c. Ora, se quella routine deve essere eseguita molte volte, allora quella moltiplicazione ci farà perdere un bel pò di tempo. Come aggirare il problema? Ci creiamo una tabella contenete tutti i valori del nostro “range” (serie) da 0...100 già moltiplicati per c, cioè una cosa del genere:

```

1  Table:
2  dc.w  0*c
3  dc.w  1*c
4  dc.w  2*c
5  dc.w  3*c
6  .
7  dc.w  n*c
8  .
9  dc.w  100*c

```

A questo punto è facile accedere alla tabella, perché dato il valore da moltiplicare per c in d0, avremo che:

```

1  Lea     Table,a0      ; Indirizzo della tabella
2  Add.w  d0,d0          ; d0 * 2, per trovare l’offset nella tabella,

```

```

3 ; dato che ogni suo valore è lungo 1 word.
4 Move.w (a0,d0.w),d0 ; Copia il valore giusto dalla tabella in d0

```

Facile, no? L'unico svantaggio è che abbiamo il listato 100 words più lungo, per contenere la tabella. Se tale tabella poi non fosse più lontana di 256 bytes, potremmo scrivere:

```

1 Add.w d0,d0 ; d0*2, ogni val. 1 word, ossia 2 byte
2 Move.w Table(pc,d0.w),d0 ; copia dalla tab. il valore giusto

```

Se il listato fosse per 68020+, basterebbe una sola istruzione:

```

1 Move.w Table(pc,d0.w*2),d0 ; istruzione del 68020 o superiori

```

Quest'ultima però è una anticipazione, infatti le ottimizzazioni specifiche per 68020 le tratteremo in seguito. Comunque, la soluzione più adottata per tabelle "corte" è quella di costruirle in una section BSS tramite una routine. In questo modo, il file eseguibile non risulta più lungo, ma occupa solamente un poco più di memoria (a meno che non facciate una tabella lunga 500Kb, in tal caso occupa MOLTA più memoria, heheheeh!)

Se siete stati attenti, nelle lezioni precedenti abbiamo già "tabellato" un paio di listati: uno per togliere una MULU.W #40, molto frequente dato che 40 è la lunghezza di una linea di schermo lowres. Rivedetevi attentamente quell'esempio, si tratta di Lezione8n2.s, dove sono presenti sia la versione ottimizzata che quella normale a confronto. Rivedetevi anche i listati precedenti per vedere le routine normale e ottimizzata da sole. Il problema era un:

```

1 mulu.w #largschermo,d1 ; Ossia mulu.w #40,d1

```

Per "risolverlo", ecco lo stratagemma:

```

1 ; PRECALCOLIAMO UNA TABELLA CON I MULTIPLI DI 40, ossia della larghezza dello
2 ; schermo, per evitare di fare una moltiplicazione per ogni plottaggio.
3
4 _._ _._ _._ _._ _._ _._ _._ _._ _._
5
6 lea MulTab,a0 ; Indirizzo spazio di 256 words dove scrivere
7 ; i multipli di 40...
8 moveq #0,d0 ; Iniziamo da 0...
9 move.w #256-1,d7 ; Numero di multipli di 40 necessari
10 PreCalcLoop
11 move.w d0,(a0)+ ; Salviamo il multiplo attuale
12 add.w #LargSchermo,d0 ; aggiungiamo larghschermo, prossimo multiplo
13 dbra d7,PreCalcLoop ; Creiamo tutta la MulTab
14 ....
15
16 SECTION Precalc,bss
17
18 MulTab:
19 ds.w 256 ; notare che la section bss, composta da zeri, non
20 ; allunga la lunghezza effettiva del file eseguibile.

```

Questo per quanto riguarda il calcolo della tabella. Poi, al posto del MULU:

```

1 lea MulTab,a1 ; Indirizzo della tabella con i multipli della
2 ; largh. schermo precalcolati in a1
3 add.w d1,d1 ; d1*2, per trovare l'offset nella tabella
4 add.w (a1,d1.w),d0 ; copia il multiplo giusto da tab a d0

```

Questo, in breve, è il metodo per tabellare una moltiplicazione. Naturalmente, qua sapevamo che d1 poteva andare solo da 0 a 255, di conseguenza abbiamo precalcolato solo 256 multipli. Se invece d1 avesse avuto un range da 0 a 65000, avremmo dovuto fare una tabella lunga 128Kb, e ciò potrebbe anche non convenire! Se il risultato massimo nella tabella non supera \$FFFF, ossia 65535, basta fare una tabella con valori .Word. Se invece i valori più alti superano tale valore, la tabella deve essere fatta di longword. In questo caso, dovremo cambiare il modo di trovare l'offset: non più *2, ma *4!


```

1 lea    MulTab,a1      ; Indirizzo della tabella con i multipli della
2      ; largh. schermo precalcolati in a1
3 add.w  d1,d1          ; d1*4, per trovare l'offset nella tabella
4 add.w  d1,d1          ;
5 move.l (a1,d1.w),d0   ; copia il multiplo giusto da tab a d0

```

Per quanto riguarda il tabellamento delle divisioni, la cosa è analoga, basta farsi una routine con un loop che divide ogni ciclo per un numero crescente e salvare i risultati nella tabella. In questo caso si può scegliere di salvare solo la word bassa, con il risultato, o anche quella alta con il resto, se serve al nostro scopo.

Una cosa fondamentale è di crearsi “in loco” la tabella, **mai inserire una tabella, specialmente se lunga diversi KB, già calcolata**. Per esempio, se precalcolassimo una multab da 20KB, immaginatevi la differenza tra un eseguibile che la calcola alla startup, e una che la ha inclusa con incbin o include già precalcolata: (esempio)

```

file1  ->    lunghezza = 40K      ; calcola la tab all'inizio
file1  ->    lunghezza = 60K      ; ha la tab inclusa con incbin

```

A livello di consumo della memoria sono in parità, ma se doveste fare una 40K intro o una 64K intro immaginatevi l'immenso risparmio di spazio, a scapito di 1 secondo o 2 di precalcolo all'inizio. Ma anche se aveste fatto un gioco o un programma, il fatto di essere più costo di 20k (o più) vi permetterebbe di inserire più cose nel disco e una maggior diffusione nelle bbs data la sua brevità. Poi c'è ancora un incentivo a precalcolare sul posto le tabelle: il fatto che si può facilmente modificare il listato, ad esempio volendo moltiplicare per 80 anziché per 40. Il FESSO che ha incluso con l'INCBIN una tabellona di multipli di 40, dovrebbe riscrivere la routine di moltiplicazione per 80, eseguirla, salvare il file binario, mentre il FURBO che ha la routine di creazione nel listato deve cambiare semplicemente 40 in 80, e fa tutto da se. Infine, specialmente per precalcoli di routine complesse, appare **molto** più chiaro il funzionamento se si ha sott'occhio la routine originaria che crea la tabella. Quindi, **precalcolate sempre “sul posto” le tabelle in spazi azzerati, specialmente in SECTION BSS se sono tabelle grandi**.

Il consiglio che vi posso dare è cercare sempre di tabellare **tutto**.

Se siete stati attentissimi, dovrete ricordarvi anche che nella lezione11 un listato ha subito una ottimizzazione da tabellamento, ben più azzardata di quella vista ora. Infatti si tabella un'intera routine, anziché una sola moltiplicazione. Non a caso l'ho messa nella Lezione11 e non nella 8! Il listato “normale” è Lezione1115.s, quella “tabellata” Lezione1115b.s. Rivedetevi come è avvenuta la forte ottimizzazione, che ripropongo.

Questa è la routine “normale”:

```

1 Animloop:
2 moveq  #0,d0
3 move.b (A0)+,d0      ; Prossimo byte in d0
4 MOVEQ  #8-1,D1       ; 8 bit da controllare e espandere.
5 BYTELOOP:
6 BTST.l D1,d0         ; Testa il bit del loop attuale
7 BEQ.S  bitclear      ; è azzerato?
8 ST.B   (A1)+         ; Se no, setta il byte (= $FF)
9 BRA.S  bitset
10 bitclear:
11 clr.B  (A1)+         ; Se è azzerato, azzerla il byte
12 bitset:
13 DBRA  D1,BYTELOOP   ; Controlla ed espandi tutti i bit del byte
14 DBRA  D7,Animloop   ; Converti tutto il fotogramma

```

Non si è fatto altro che precalcolare tutte le possibilità:

```

1 *****
2 ; Routine che precalcola tutti i possibili 8 bytes abbinati ai possibili
3 ; 8 bit. Per tutti si intende $FF, ossia 255.
4 *****
5

```

```

6 PrecalcoTabba:
7 lea    Precalctabba, a1 ; Destinazione
8 moveq  #0,d0           ; Parti dal valore zero
9 FaiTabba:
10 MOVEQ #8-1,D1          ; 8 bit da controllare e espandere.
11 BYTELOOP:
12 BTST.l D1,d0           ; Testa il bit del loop attuale
13 BEQ.s  bitclear        ; è azzerato?
14 ST.B   (A1)+           ; Se no, setta il byte (=$FF)
15 BRA.s  bitset
16 bitclear:
17 clr.B  (A1)+           ; Se è azzerato, azzerla il byte
18 bitset:
19 DBRA   D1,BYTELOOP     ; Controlla ed espandi tutti i bit del byte:
20 ; D1 calando ogni volta fa fare il btst di
21 ; tutti i bit.
22 ADDQ.W #1,D0           ; Prossimo valore
23 CMP.W  #256,d0         ; Li abbiamo fatti tutti? (max $FF)
24 bne.s  FaiTabba
25 rts

```

E cambiare la routine “esecutiva”:

```

1 Animloop:
2 moveq  #0,d0
3 move.b (A0)+,d0        ; Prossimo byte in d0
4 lsl.w  #3,d0           ; d0*8 per trovare il valore nella tabba
5 ; (ossia l'offset dal suo inizio)
6 lea    Precalctabba, a2
7 lea    0(a2,d0.w),a2   ; In a2 l'indirizzo nella tabba degli 8 byte
8 ; giusti per "l'espansione" degli 8 bit.
9 move.l (a2)+,(a1)+     ; 4 bytes espansi
10 move.l (a2),(a1)+      ; 4 bytes espansi (totale 8 bytes!!)
11
12 DBRA   D7,Animloop     ; Converti tutto il fotogramma

```

Come vedete, qua stiamo entrando in un tipo di ottimizzazione che per essere fatto occorre avere una certa esperienza, e un certo intuito. Meccanicamente è facile dire: “provo a tabellare tutte le moltiplicazioni e le divisioni, e metto tutti gli ADDQ e i MOVEQ possibili”. Ma quando ci so trovano davanti routines “strane” tipo quella già vista che fa dei BTST di tutto un byte e lo espande ad 8 bytes, occorre avere l’occhio di lince per intuire come ottimizzarla. È questo occhio di lince che fa la differenza tra una routine 3d che va a scatti quando gira 10 punti, e una che va al cinquantesimo di secondo pur ruotandone 8192. E naturalmente non si può fare un elenco di tutte le possibili routines con accanto tutte le possibili ottimizzazioni. Occorre farsi l’occhio di lince vedendo i pochi esempi presentati.

13.3 Ottimizzazioni varie - gruppo misto

Consideriamo il caso in cui dobbiamo eseguire una determinata routine per ogni valore in d0, ed inoltre supponiamo che questi possibili valori siano compresi tra 0 e 10. Ebbene, potremmo essere tentati di fare una cosa del genere:

```

1 Cmp.b  #1,d0
2 Beq.s  Rout1
3 Cmpi.b #2,d0
4 Beq.s  Rout2
5 ...
6 Cmp.b  #10,d0
7 Beq.s  Rout10

```

è una pessima idea, infatti come minimo avremmo potuto fare così:

```

1 Subq.b #1,d0 ; togliamo 1. Se d0=0, allora si imposta il flag Z
2 Beq.s  Rout1 ; Di conseguenza d0 era 1, e saltiamo a Rout1
3 Subq.b #1,d0 ; eccetera...
4 Beq.s  Rout2

```

```

5  ...
6  Subq.b  #1,d0
7  Beq.s   Rout10

```

In effetti così è già meglio, ma noi siamo perfezionisti e con l'aiuto di una tabella facciamo così:

```

1  Add.w   d0,d0           ; \ d0*4, per trovare l'offset nella tabella,
2  Add.w   d0,d0           ; /      costituito da longwords (4 bytes!)
3  Move.l   Table(pc,d0.w),a0 ; In a0 l'indirizzo della routine giusta
4  Jump     (a0)
5
6  Table:
7  dc.l     Rout1         ; 0 (valore in d0 per richiamare la routine)
8  dc.l     Rout2         ; 1
9  dc.l     Rout3         ; 2
10 dc.l     Rout4         ; 3
11 dc.l     Rout5         ; 4
12 dc.l     Rout6         ; 5
13 dc.l     Rout7         ; 6
14 dc.l     Rout8         ; 7
15 dc.l     Rout9         ; 8
16 dc.l     Rout10        ; 9

```

In questo modo non facciamo confronti, ed è ovvio che è un'ottima tecnica nel caso in cui conosciamo i valori da confrontare e sono consecutivi. Vorrei farvi notare inoltre che se utilizziamo in modo intensivo le tabelle potremmo addirittura lavorare con le potenze del due, risparmiando quindi quelle due ADD.w. Quindi, quando si vuole la routine 1, occorrerà d0=0, quando si vuole Rout2 d0=4, quando si vuole Rout3 d0=8, eccetera.

Ci sono anche delle varianti di questo sistema, ad esempio:

```

1  move.b   Table(pc,d0.w),d0      ; Prendi dalla tavola l'offset giusto
2  jmp      Table(pc,d0)           ; aggiungilo a Table, e salta!
3
4  Table:
5  dc.b     Rout1-Table          ; 0
6  dc.b     Rout2-Table          ; 1
7  dc.b     Rout3-Table          ; 2
8  ...
9  even

```

Con questo sistema non dobbiamo moltiplicare d0, perché abbiamo fatto una tabella di offset delle routines dalla tabella stessa. Qua sono offset .byte, perché si suppone che le routines siano piccole e vicine. Altrimenti gli offset potranno essere .word:

```

1  add.w    d0,d0             ; d0*2
2  move.w    Table(pc,d0.w),d0   ; Prendi dalla tavola l'offset giusto
3  jmp      Table(pc,d0)         ; aggiungilo a Table, e salta!
4
5  Table:
6  dc.w     Rout1-Table         ; 0
7  dc.w     Rout2-Table         ; 1
8  dc.w     Rout3-Table         ; 2
9  ...

```

Il vantaggio di questo sistema è che non occorre moltiplicare per 4 il registro d0, ma solo per 2. Se non riuscite a mettere la tabella abbastanza vicina, potete fare così:

```

1  add.w    d0,d0             ; d0*2
2  lea      Table(pc),a0
3  move.w    (a0,d0.w),d0
4  jmp      (a0,d0.w)
5
6  Table:
7  dc.w     Rout1-Table         ; 0
8  dc.w     Rout2-Table         ; 1
9  dc.w     Rout3-Table         ; 2
10 ...

```

Già che abbiamo implementato il salto alle routine utilizzando SUBQ.b #1,d0 seguito da BEQ, senza né CMP né TST, occupiamoci degli usi di tale particolarità, legata ai Condition Codes

(rivedetevi bene in 68000-2.txt) Noi programmatori in assembly ci possiamo prendere il lusso di testare tre condizioni alla volta, infatti consideriamo l'esempio:

```

1 Add.w  #x,d0      ; i cc sono settati in qualche modo
2 Beq.s   Zero      ; il risultato è zero
3 Blt.s   Negativo   ; il risultato è minore di zero
4 ...           ; Altrimenti il risultato è positivo ...

```

Quindi, se dovete testare qualche risultato, cercate sempre di farlo dopo l'ultima operazione matematica, e non alla fine quando i cc indicheranno tutt'altra cosa. Sarebbe bene se conoscestes quali cc influenzano le varie istruzioni. Inoltre vi consiglio di piazzare le BCC a seconda della loro probabilità di essere eseguite per prime, cioè praticamente quelle che fanno trasferire il controllo con più probabilità. Ad esempio un altro caso interessante è questo: abbiamo un certo numero di valori, non sappiamo quanti, ma sappiamo che terminano con uno zero... supponiamo che dobbiamo copiarli da una zona di memoria ad un'altra. Potremmo fare una cosa del genere:

```

1 Lea     Source,a0
2 Lea     Dest,a1
3 CpLoop:
4 Move.b  (a0)+,d0      ; sorgente -> d0
5 Move.b  d0,(a1)+      ; d0 -> destinazione
6 Tst.b   d0            ; d0=0?
7 Bne.s   CpLoop        ; Se non ancora, continua

```

Ma possiamo fare di meglio nel seguente modo:

```

1 Lea     Source,a0
2 Lea     Dest,a1
3 CpLoop:
4 Move.b  (a0)+,(a1)+    ; sorgente -> destinazione
5 Bne.s   CpLoop        ; flag 0 settato? Se non ancora, continua!

```

Come potete vedere, il 68000 in questo caso fa tutto da solo.

Parliamo ora delle chiamate alle subroutine, e quindi delle MOVEM. L'utilizzo delle subroutine è evidentemente utilissimo nella stesura dei programmi, ma al momento di ottimizzare il vostro codice occorre notare che invece di utilizzare la coppia di istruzioni BSR label / RTS potreste anche utilizzare BRA label, seguita alla fine della subroutine con una un'altra BRA che vi riporta alla istruzione immediatamente successiva a JMP label, ma questa ottimizzazione è a vostra discrezione. Comunque utilizzate se potete sempre BSR invece di JSR ed analogamente BRA invece di JMP, sempre che sia possibile. Ritornando comunque all'utilizzo delle routine, capita spesso volte di dover azzerare i contenuti dei registri prima di iniziare a lavora su di essi, tuttavia possiamo risparmiarci ogni volta una sfilza di MOVEQ #0,Dx e SUB.l Ax,Ax, infatti facciamo questa operazione all'inizio del programma principale e vediamo cosa accade quando chiamiamo le nostre subroutine, esempio:

```

1 Moveq   #0,d0      ;
2 Moveq   #0,d1
3 ...
4 Moveq   #0,d7
5 Move.l  d0,a0
6 ..
7 Move.l  d0,a6
8 Main:
9 Bsr.s   Pippo
10 Bsr.s   Pluto
11 Bsr.s   Paperino
12 ...
13 Bra.s   Main

```

Ebbene se salviamo ad ogni chiamata i contenuti dei registri utilizzati avremo che ogni volta che termina una routine si andrà alla prossima già con i registri "puliti", è ovvio che comunque bisogna organizzare il proprio codice per bene. Altrimenti, potreste con 1 sola istruzione pulire tutti i registri, e precisamente:

```

1  movem.l  TantiZeri(PC),d0-d7/a0-a6
2
3  TantiZeri:
4  dcb.b    15,0

```

Veniamo ora all'istruzione MOVEM ed esaminiamo i suoi pregi e difetti. Osserviamo innanzitutto il numero di cicli macchina della MOVEM, soprattutto nei trasferimenti di longword: nei trasferimenti dai registri alla memoria impiega $8+8n$, dove n indica il numero di registri, osserviamo inoltre il numero di cicli che invece impiega una semplice MOVE.l Dx, (Ax): 12 cicli. Il solito ingegnere si potrebbe porre allora la seguente domanda: nel caso devo trasferire più longword contenute in registri tutti differenti, fino a che punto mi conviene utilizzare la classica MOVE.l Dx, (Ax) ? Ebbene anche questa volta l'ingegnere a fatto una giusta osservazione, consideriamo infatti un caso limite in cui dobbiamo trasferire il contenuto dei registri D0..D7 e A0..A6: avremmo bisogno esattamente di $8+7=15$ MOVE.l per un totale di $15*12=180$ cicli. Invece, se utilizziamo la MOVEM, avremmo $8+8*15=128$ cicli, cioè un risparmio di 52 cicli! È evidente a questo punto che bisogna utilizzare la mastodontica MOVEM quando bisogna trasferire grosse quantità di dati, tuttavia nel caso siano coinvolti solamente due registri si può utilizzare ancora la MOVE.l normale. Vediamo a questo punto una serie di applicazioni pratiche che partono da un codice non ottimizzato fino ad arrivare ad uno ottimizzato. Supponiamo per esempio di dover azzerare 1200 byte a partire dalla locazione Table; i principianti farebbero in questo modo:

```

1  Lea      Table,a0          ; 12 cicli
2  Move.w   #1200-1,d7        ; 8 cicli
3  CleaLoop:
4  Clr.b    (a0)+              ; 8 cicli
5  Dbne     d7,CleaLoop

```

Questo tipo di codice è orrido!! Infatti vediamo quanto impiega...le prime due istruzioni impiegano 20 cicli, invece la CLR.b dovrà essere eseguita 1200 volte quindi $1200*8=9600$ cicli, inoltre c'è da aggiungere la DBNE che dovrà essere eseguita ben $10*1199=11990$ cicli più 14 alla fine, dunque ricapitolando $20+9600+11990+14=21624$ cicli!!! Bè tutto ciò non merita commento. Avremmo almeno potuto fare una cosa del genere:

```

1  Lea      Table,a0
2  Move.w   #(1200/4)-1,d7 ; numero di bytes diviso 4, per il clr.L!!
3  Clr:
4  Clr.l    (a0)+           ; azzeriamo 4 bytes alla volta...
5  Dbra     d7,Clr          ; e facciamo 1/4 di loops.

```

Infatti, con una CLR.l, almeno cancelliamo in un colpo 4 byte, e poiché ne dobbiamo cancellare 1200, faremmo $1200/4=300$ cicli, risparmiando un bel pò rispetto a prima (fatevi voi i calcoli per pietà). Per ottimizzare ancora di più, possiamo fare questo:

```

1  Lea      Table,a0
2  Move.w   #(1200/16)-1,d7 ; numero di bytes diviso 16, per il clr.L!!
3  Clr:
4  Clr.l    (a0)+           ; azzeriamo 4 bytes
5  Clr.l    (a0)+           ; azzeriamo 4 bytes
6  Clr.l    (a0)+           ; azzeriamo 4 bytes
7  Clr.l    (a0)+           ; azzeriamo 4 bytes
8  Dbra     d7,Clr          ; e facciamo 1/16 di loops.

```

Tuttavia anche questo tipo di codice può essere classificato come pessimo, proviamo ad ottimizzarlo di più, usando un registro dati:

```

1  Lea      Table,a0
2  moveq    #0,d0           ; più veloce "move.l d0" di un "CLR"!
3  Move.w   #(1200/32)-1,d7 ; numero di bytes diviso 32
4  Clr:
5  move.l    d0,(a0)+        ; azzeriamo 4 bytes
6  move.l    d0,(a0)+
7  move.l    d0,(a0)+
8  move.l    d0,(a0)+

```

```

9  move.l  d0,(a0)+
10 move.l  d0,(a0)+
11 move.l  d0,(a0)+
12 move.l  d0,(a0)+
13 Dbra    d7,Clr          ; e facciamo 1/32 di loops.

```

Con questa versione abbiamo aumentato l'ottimizzazione dovuta al calo dei DBRA da eseguire, e abbiamo sfruttato il fatto che usare i registri è megaveloce, anche più del CLR.

Veniamo ora ad utilizzare la MOVEM e vediamo cosa avviene:

```

1  movem.l  TantiZeri(PC),d0-d6/a0-a6      ; azzeriamo tutti i registri
2  ; tranne d7 e a7, naturalmente,
3  ; che è lo stack. Potete
4  ; azzerarli così o con
5  ; tanti moveq #0,Dx...
6
7  ; Ora abbiamo 7+7=14 registri azzerati, per un totale di 14*4=56 byte.
8  ; Dobbiamo fare 1200byte/56byte = 21 trasferimenti, ma 21*56 = 1176 byte, e ne
9  ; restano da fare altri 1200-1176 = 24byte che faremo a parte.
10
11 Move.l  a7,SalvaStack ; Salviamo lo stack in una label
12 Lea    Table+1200,a7 ; Mettiamo in A7 (o SP, è lo stesso registro)
13 ; l'indirizzo della fine dell'area da pulire.
14 Moveq  #21-1,d7      ; Numero di movem da fare (2100/56=21)
15 CleaLoop:
16 Movem.l d0-d6/a0-a6,-(a7) ; Azzeriamo "all'indietro" 56 bytes.
17 ; Se vi ricordate, il movem in scrittura
18 ; lavora "all'indietro", per lo stack.
19 Dbra    d7,CleaLoop
20 Movem.l d0-d5,(a7)+      ; Azzeriamo gli ultimi 24 bytes
21 Move.l  SalvaStack(PC),a7 ; Rimettiamo a posto lo stack in SP
22 rts
23
24 SalvaStack:
25 dc.l    0

```

Facciamo un pò di conti, la MOVEM interna occuperà esattamente $8+8*14=120$ cicli, dovrà essere eseguita 21 volte, quindi $21*120=2520$ cicli, a cui dovremo aggiungere tutta la fase di inizializzazione e di chiusura ma non vi preoccupate non supererà mai i casi di sopra. Possiamo essere ancora più perfezionisti espandendo il codice, cioè eliminando il cicli e piazzando tante MOVEM quante ne abbiamo bisogno; non vi spaventate, l'espansione del codice è una tecnica molto utilizzata, soprattutto quando non si sa più cosa ottimizzare, vedremo in seguito una serie di esempi. Nel caso di prima, comunque, ecco cosa succederebbe:

```

1  Move.l  a7,SalvaStack ; Salviamo lo stack in una label
2  Lea    Table+1200,a7 ; Mettiamo in A7 (o SP, è lo stesso registro)
3  ; l'indirizzo della fine dell'area da pulire.
4  CleaLoop:
5
6  rept    20              ; ripeto 20 movem...
7  Movem.l d0-d7/a0-a6,-(a7) ; Azzeriamo "all'indietro" 60 bytes.
8  endr
9
10 Move.l  SalvaStack(PC),a7 ; Rimettiamo a posto lo stack in SP
11 rts

```

Da notare che, avendo eliminato il DBRA, possiamo usare anche il registro d7, che ci fa azzerare 4 bytes in più per ogni MOVEM. In questo modo, 1200/60 fa 20 esatto. Le demo solitamente usano questo sistema, il più veloce!

Vediamo meglio la tecnica di espansione del codice. Osservate questa routine:

```

1  ROUTINE2:
2  MOVEQ   #64-1,D0          ; 64 cicli
3  SLOWLOOP2:
4  MOVE.W  (a2),(a1)
5  ADDQ.W  #4,a1
6  ADDQ.W  #8,a2
7  DBRA    D0,SLOWLOOP2

```

Ed ecco la routine molto velocizzata:

```

1 ROUTINE2:
2 MOVE.W (a2),(a1)
3 MOVE.W 8(a2),4(a1)
4 MOVE.W 8*2(a2),4*2(a1)
5 MOVE.W 8*3(a2),4*3(a1)
6 MOVE.W 8*4(a2),4*4(a1)
7 MOVE.W 8*5(a2),4*5(a1)
8 MOVE.W 8*6(a2),4*6(a1)
9 MOVE.W 8*7(a2),4*7(a1)
10 .....
11 MOVE.W 8*63(a2),4*63(a1)

```

Abbiamo tolto il tempo usato per il DBRA e i 2 ADDQ! Comunque, occorre dire che nei processori 68020 e superiori sono presenti le instruction cache, che velocizzano i loop lunghi meno di 256 bytes. Quindi, può succedere di ottimizzare per 68000, e rendere meno veloce su un 68020. Di conseguenza, sarebbe bene fare una mediazione di questo tipo:

```

1 ROUTINE2:
2 MOVEQ #4-1,D0 ; solo 4 cicli (64/16)
3 FASTLOOP2:
4 MOVE.W (a2),(a1) ; 1
5 MOVE.W 8(a2),4(a1) ; 2
6 MOVE.W 8*2(a2),4*2(a1) ; 3
7 MOVE.W 8*3(a2),4*3(a1) ; 4
8 MOVE.W 8*4(a2),4*4(a1) ; 5
9 MOVE.W 8*5(a2),4*5(a1) ; ...
10 MOVE.W 8*6(a2),4*6(a1)
11 MOVE.W 8*7(a2),4*7(a1)
12 MOVE.W 8*8(a2),4*8(a1)
13 MOVE.W 9*9(a2),4*9(a1)
14 MOVE.W 8*10(a2),4*10(a1)
15 MOVE.W 8*11(a2),4*11(a1)
16 MOVE.W 8*12(a2),4*12(a1)
17 MOVE.W 8*13(a2),4*13(a1)
18 MOVE.W 8*14(a2),4*14(a1)
19 MOVE.W 8*15(a2),4*15(a1) ; 16
20 ADD.W #4*16,a1
21 ADD.W #8*16,a2
22 DBRA D0,FASTLOOP2

```

Questo anche per il CLEAR con i MOVEM e le altre routine dove ripetiamo a tappeto.

Facciamo ora un paio di osservazioni utili: il metodo di indirizzamento indiretto con autoincremento è qualcosa da tenere sempre in considerazione. Infatti l'indiretto, sia senza incremento che con incremento, impiega lo stesso numero di cicli, un ottimo caso è nell'utilizzo del Blitter, ed un esempio di questo genere lo vedremo in seguito. Il secondo metodo che abbiamo utilizzato per copiare i 1200 byte tuttavia non è proprio da buttare completamente: nel caso dovessimo fare una copia possiamo fare molto meglio, ma pensate al caso in cui dovessimo mascherare 1200 byte: siamo per forza costretti ad utilizzare un ciclo DBCC. In questi casi cercate di abusare dell'istruzione DBCC e ricordate che su un 680xx dotato di cache questi tipi di cicli sono eseguiti a velocità TURBO. Inoltre le istruzioni DBCC sono ottime anche per comparare, ecco un esempio:

```

1 Move.w Len(PC),d0 ; Max lunghezza in cui cercare <> 0
2 Move.l String(PC),a0
3 Moveq #Char,d1 ; Carattere da cercare
4 FdLoop:
5 Cmp.b (a0)+,d1
6 Dbne.s d0,FdLoop

```

Il seguente ciclo controlla due cose contemporaneamente, infatti i cc EQ saranno settati se abbiamo esaminato tutti i Len (numero caratteri), oppure se il carattere è stato trovato, in questo caso saremmo anche in grado di dire in che posizione si trova. Vorrei a questo punto fare gli ultimi esempi sulla MOVEM ed esattamente sulla copia di zone di memoria: a differenza dell'azzeramento, qui dobbiamo prelevare dei dati e poi scaricarli, ma vediamo subito un esempio:

```

1  Lea      Start ,a0
2  Lea      Dest ,a1
3  FASTCOPY:                                ; Impiego 13 registri
4  Movem. l (a0)+,d0-d7/a2-a6
5  Movem. l d0-d7/a2-a6,(a1)
6  Movem. l (a0)+,d0-d7/a2-a6
7  Movem. l d0-d7/a2-a6,$34(a1)      ; $34
8  Movem. l (a0)+,d0-d7/a2-a6
9  Movem. l d0-d7/a2-a6,$34*2(a1)    ; $34*2
10 Movem. l (a0)+,d0-d7/a2-a6
11 Movem. l d0-d7/a2-a6,$34*3(a1)
12 Movem. l (a0)+,d0-d7/a2-a6
13 Movem. l d0-d7/a2-a6,$34*4(a1)
14 Movem. l (a0)+,d0-d7/a2-a6
15 Movem. l d0-d7/a2-a6,$34*5(a1)
16 Movem. l (a0)+,d0-d7/a2-a6
17 Movem. l d0-d7/a2-a6,$34*6(a1)
18 Movem. l (a0)+,d0-d7/a2-a6

```

Innanzitutto qui abbiamo adottato la tecnica (se si può chiamarla così) dell'espansione del codice: può risultare esagerato, ma è molto efficiente. Bene, cosa abbiamo fatto ? Preleviamo 13*4 byte dalla locazione di memoria puntata ad a0, e li copiamo nella locazione di memoria puntata in a1, prestando attenzione al fatto di aumentare l'offset di a1 dopo ogni copia. Nel caso in cui vogliate espandere il codice ma, vi da fastidio vedere tutte quelle istruzioni, potete usare la direttiva REPT:

```

1  REPT      100
2  And. l   (a0)+,(a1)+
3  ENDR

```

Sarà poi l'assemblatore a generarle per voi. Infine vediamo un esempio legato ai registri colori:

```

1  Lea      $dff180,a6
2  Movem. l Colours(pc),d0-a5      ; carichiamo 14 longword o 28 word
3  Movem. l d0-a5,(a6)              ; setta 28 colori in un solo colpo!!
4
5  Colours:      dc.w      ...

```

Oppure quando all'inizio di una routine dovete caricare molti registri:

```

1  MOVE. L   #$4232,D0
2  MOVE. W   #$F20,D1
3  MOVE. W   #$7FFF,D2
4  MOVEQ    #0,D3
5  MOVE. L   #123456,D4
6  LEA      $DFF000,A0
7  LEA      $BFE001,A1
8  LEA      $BFD100,A2
9  LEA      Schermo,A3
10 LEA      BUFFER,A4
11 ...

```

Tutto questo si può riassumere con 1 sola routine:

```

1  MOVEM. L   VariaRoba(PC),D0-D4/A0-A4
2  ...
3
4  VariaRoba:
5  dc.l      $4243      ; d0
6  dc.l      $f20       ; d1
7  dc.l      $7fff      ; d2
8  dc.l      0          ; d3
9  dc.l      $123456    ; d4
10 dc.l      $dff000    ; a0
11 dc.l      $bfe001    ; a1
12 dc.l      $bfd100    ; a2
13 dc.l      Schermo    ; a3
14 dc.l      Buffer      ; a4

```


Sull'istruzione MOVEM potremmo fare molti altri esempi, ma credo che abbiate capito la sua convenienza in dati casi.

Le chiamate relative al Program Counter (PC) sono più veloci di quelle normali alle label perché sono più “piccole”. Infatti quelle normali devono contenere l'indirizzo lungo 32bit della label, mentre quelle (PC) contengono solo l'offset a 16 bit dal registro PC, cosa che risparmia 2 bytes e del tempo. Purtroppo, è proprio il fatto che l'offset sia a 16 bit che non permette di rendere relative al PC label più lontane di 32k in avanti o indietro. Veniamo ora ad un trucchetto per rendere relativo al (PC) tutto il programma, cosa che velocizza anche l'esecuzione. Come sapete, è possibile fare:

```
1 move.l label1(PC),d0
```

Ma è impossibile rendere relativa al PC questa istruzione:

```
1 move.l d0,label1
```

Come fare? Non è un problema importantissimo, ma mettiamo che abbiamo in un loop eseguito molte volte questa istruzione. Se non possiamo rendere la label relativa al PC, possiamo renderla relativa ad un comune registro indirizzi! Il metodo più evidente, è questo:

```
1 move.x XXXX,label    ->    lea    label(PC),a0
2 move.x XXXX,(a0)
3
4 tst.x label          ->    lea    label(PC),a0
5 tst.x label
```

Da notare, che si risparmia anche a sostituire i valori #immediati con valori caricati in registri dati, sempre che i valori siano tra -80 e +7f per permettere il MOVEQ:

```
1 move.l #xx,dest      ->    moveq   #xx,d0
2 move.l d0,dest
3
4
5 ori.l #xx,dest       ->    moveq   #xx,d0
6 or.l d0,dest
7
8
9 addi.l #xx,dest      ->    moveq   #xx,d0
10 add.l d0,dest
```

In particolare, se è possibile caricare tutti i registri prima di un loop, per poi risparmiare sul caricamento, si possono anche fare MOVE.L #xx,Dx tranquillamente, il loop senza #immediati ci ripagherà!

Esempio:

```
1 RoutineSchifosa:
2 move.w #1024-1,d7      ; numero di loops
3 LoopSquallido:
4 add.l #$567,label2
5 sub.l #$23,label3
6 move.l label2(PC),(a0)+
7 move.l label3(PC),(a0)+
8 add.l #30,(a0)+
9 sub.l #20,(a0)+
10 dbra d7,LoopSquallido
11 rts
```

Questa si può ottimizzare così:

```
1 RoutineDecente:
2 moveq #30,d0           ; carichiamo i registri necessari...
3 moveq #20,d1
4 move.l #$567,d2
5 moveq #23,d3
6 lea label2(PC),a1
7 lea label3(PC),a2
8 move.w #1024-1,d7      ; numero di loops
```

```

9  LoopNormale:
10 add.l  d2,(a1)
11 sub.l  d3,(a2)
12 move.l (a1),(a0)+
13 move.l (a2),(a0)+
14 add.l  d0,(a0)+
15 sub.l  d1,(a0)+
16 dbra   d7,LoopNormale
17 rts

```

Per esagerare, possiamo infine risparmiare sul numero di DBRA da eseguire:

```

1  RoutineOK:
2  moveq   #30,d0
3  moveq   #20,d1
4  move.l  #$567,d2
5  moveq   #23,d3
6  lea     label2(PC),a1
7  lea     label3(PC),a2
8  move.w  #(1024/8)-1,d7      ; numero di loops = 128
9  LoopOK:
10
11 rept    8                  ; replica 8 volte il pezzo...
12
13 add.l  d2,(a1)
14 sub.l  d3,(a2)
15 move.l (a1),(a0)+
16 move.l (a2),(a0)+
17 add.l  d0,(a0)+
18 sub.l  d1,(a0)+
19
20 endr
21
22 dbra   d7,LoopNormale
23 rts

```

Comunque, per rendere velocemente tutto PC relative, c'è un sistema. Se in un registro indirizzi stabilito, ad esempio a5, mettiamo l'indirizzo dell'inizio del programma, o comunque un indirizzo conosciuto nel nostro programma, basterà indicare la nostra label come a5+offset per trovare la label in questione. Ma dovremmo fare questo **a mano**???? Nooooo! Ecco un sistema molto veloce per fare questo:

```

1  S:                      ; Label di riferimento
2  MYPROGGY:
3  LEA     $dff002,A6      ; In a6 abbiamo il custom register
4  LEA     S(PC),A5        ; In a5 il registro per l'offset delle label
5
6  MOVE.L  #$123,LABEL2-S(A5) ; label2-s = offset! Es: "$364(a5)"
7
8  MOVE.L  LABEL2(PC),d0    ; Qua agiamo normalmente
9
10 MOVE.L  d0,LABEL3-S(A5)  ; stesso discorso.
11
12 move.l  #$400,$96-2(a6)  ; Dmacon (in a6 c'è $dff002!!!)
13
14 ...
15
16 ; mettiamo di aver "sporcato" il registro A5... basterà ricaricarlo!
17
18 LEA     S(PC),A5
19 move.l  $64(a1),OLDINT1-S(A5)
20 CLR.L   LABEL1-S(A5)

```

Mi pare chiaro no? La label la potevate chiamare BAU: anzichè S:, ma credo che sia utile chiamarla S:, E:, I:, che è più corto da scrivere. L'unica limitazione è che se la label dista più di 32K dalla label di riferimento, andiamo fuori dei limiti di indirizzamento. Questo non è un problema insormontabile, infatti basta mettere una label di riferimento ogni 30K, e riferirsi alla più vicina, esempio:

```

1  B:

```

```

2  ...
3  LEA      B(PC),A5
4  MOVE.L  D0,LABEL1-B(A5)
5  ...
6
7  ; passano 30K
8
9  C:
10
11 LEA      C(PC),A5
12 MOVE.L  (a0),LABEL40-C(A5)
13 ...

```

Questo sistema inoltre rende difficile da disassemblare il vostro codice, nel caso che qualcuno voglia “rubarvi” le vostre routine con un disassemblatore.

Altra cosa che può esservi utile è l’uso dei bit come flag. Per esempio, se nel nostro programma abbiamo delle variabili che devono essere VERE o FALSE, ossia ACCESE o SPENTE, è inutile sprecare un byte per ognuna di esse. Basterà un bit, e risparmieremo spazio. Per esempio:

```

1  Opzione1      =      0
2  VaiDestra     =      1      ; Vai a Destra o a Sinistra?
3  Avvicinamento =      2      ; Avvicinamento o Allontanamento?
4  Music         =      3      ; Musica Accesa o Spenta?
5  Candele       =      4      ; Candele accese o spente?
6  FirePremuto   =      5      ; qualcuno ha premuto fire?
7  Acqua         =      6      ; il laghetto sotto?
8  Cavallette    =      7      ; Ci sono le cavallette?
9
10 Controllo:
11 move.w  MieFlags(PC),d0
12 btst.l  #Opzione1,d0
13 ...
14
15
16 CambiaFlags:
17 lea     MieFlags(PC),a0
18 bclr.b  #Opzione1,(a0)
19 ...
20
21 MieFlags:
22 dc.b    0
23 even

```

Comunque, se non amate i BTST e i BCLR / BSET / BCHG si può fare così:

```

bset.l  #Opzione1,d0  ->    or.b    #1<<Opzione1,d0

bclr.l  #Opzione1,d0  ->    and.b    ~(1<<Opzione1),d0

bchg.l  #Opzione1,d0  ->    eor.b    #1<<Opzione1,d0

```

Da notare l’utilità delle funzioni dell’asmone di shift “>” e “<”, nonché l’eor “^”.

Per terminare la sezione sulle ottimizzazioni della CPU, faccio presenti alcuni accorgimenti che velocizzano solo su 68020 e superiori, ma dato che farli non costa niente, può essere utile per veder schizzare maggiormente le nostre routines su computer più veloci. Prima di tutto, ci sono le cache, che permettono di caricare loops lunghi fino a 256 bytes, per cui dal secondo ciclo in avanti saranno letti dalla memoria interna alla cpu!!!!!!!!!!!!!! E non dalla lenta memoria (specie se chip-ram!). Di conseguenza, è bene ripetere le operazioni come abbiamo visto, nei vari loop, in modo che siano grandi sui 100-150 bytes. In questo modo, su 68020+ gireranno molto più veloci di routines in cui invece si sono messi in fila tante istruzioni quanti erano i loop da fare. Per intendersi, se abbiamo:

```

1  Routine1:
2  move.w  #2048-1,d7
3  loop1:

```

```

4 < blocco di istruzioni >
5 dbra    d7,loop1
6
7 Possiamo ottimizzarlo in:
8
9 Routine1:
10 rept    2048
11 < blocco di istruzioni >
12 endr

```

Che su un 68000 di base è molto più veloce, ma su un 68020 è più lento! Per fare un'ottimizzazione che sia veloce al massimo in tutti i casi:

```

1 Routine1:
2 move.w  #(2048/16)-1,d7
3 loop1:
4 rept    16
5 < blocco di istruzioni >
6 endr
7
8 dbra    d7,loop1

```

Supponiamo che il blocco di istruzioni sia lungo 12 bytes, allora $12 \cdot 16$ fa 192, che sta nella cache, e va velocissimo su 68020, mentre su 68000 è impercettibile la differenza con la versione con 2048 di REPT, e si risparmia anche in lunghezza dell'eseguibile. Attenzione solo a non fare i loop lunghi proprio 250 o 256 bytes, perché la cache può essere riempita solo secondo certe condizioni di “blocchi” e di “allineamento”. Quindi state sempre sotto i 180-200 bytes, per sicurezza.

Altra cosa da tener presente, è che se è possibile bisogna evitare di accedere alla memoria consecutivamente. Ad esempio:

```

1 move.l  d0,(a0)
2 move.l  d1,(a1)
3 move.l  d2,(a2)
4 sub.l   d2,d0
5 eor.l   d0,d1
6 add.l   d1,d2

```

Andrebbe “riformulato” in:

```

1 move.l  d0,(a0)
2 sub.l   d2,d0
3 move.l  d1,(a1)
4 eor.l   d0,d1
5 move.l  d2,(a2)
6 add.l   d1,d2

```

Infatti, quando si accede alla memoria (specie se chip), ci sono i cosiddetti WAIT STATE, ossia tempi d'attesa prima di poterci riscrivere. Nel primo esempio, tra una scrittura e l'altra c'è un tempo morto in cui il processore attende che si possa riscrivere in ram. Nel secondo caso, invece, dopo aver scritto in ram viene eseguita una operazione tra registri, interna alla cpu, dopodiché si accede nuovamente alla chip ram, a tempo d'accesso passato. Se si accede a FAST RAM a 32bit il problema è molto meno forte, ma esiste.

Infine, i 68020+ gradiscono molto le routines e le label allineate ad indirizzi multipli di 32, ossia allineate a longword. Per allineare a 32 bit, baste un:

```

1 CNOP    0,4

```

Prima della routine o della label. Su 68000 non ci sono miglioramenti, ma ci sono su 68020+, specialmente se il codice allineato va in fast ram o in cache. Ecco un esempio:

```

1 Routine1:
2 bsr.s   rotazione
3 bsr.s   proiezione
4 bsr.s   disegno
5 rts

```

```

6
7  cnop      0,4
8  rotazione:
9  ...
10 rts
11
12 cnop      0,4
13 proiezione:
14 ...
15 rts
16
17 cnop      0,4
18 disegno:
19 ...
20 rts

```

Per le label, vedete di non accedere ad indirizzi dispari, cosa che rallenta molto, piuttosto, allineate anche queste a long:

Versione originale:

```

1 Label1:
2 dc.b    0
3 Label2:
4 dc.b    0      ; indir. dispari! il "move.b xx,label1" sarà lento!
5 Label3:
6 dc.w    0
7 Label4:
8 dc.w    0
9 Label5:
10 dc.l    0
11 Label6:
12 dc.l    0
13 Label7:
14 dc.l    0

```

Versione allineata:

```

1  cnop      0,4
2  Label1:
3  dc.b      0
4  cnop      0,4
5  Label2:
6  dc.b      0
7  cnop      0,4
8  Label3:
9  dc.w      0
10 cnop      0,4
11 Label4:
12 dc.w      0
13 cnop      0,4
14 Label5:
15 dc.l      0
16 Label6:
17 dc.l      0 ; queste 2 sono allineate sicuramente, non occorre cnop
18 Label7:
19 dc.l      0

```

Per verificare se una label è allineata a 32 bit, assemblete, poi verificate a che indirizzo si trova tale label col comando `<M>`, poi dividete l'indirizzo per 4, e moltiplicate di nuovo il risultato per 4. Se torna l'indirizzo originario, significa che è un multiplo di 4, e tutto è OK, se viene diverso significa che c'è un resto e non è un multiplo di 4. Allora mettete dei `DC.w 0` sopra l'indirizzo e provate ad allinearlo “a mano” e mandate a quel paese l'assemblatore, che è un pò suonato. Comunque, se la vostra routine funziona già al cinquantesimo, senza scatti su un a500, risparmiatevi di mettere tutti quei `CNOP 0,4` a incasinarvi il listato. Cnoppate solo i listati con routines molto pesanti che non vanno entro un frame, come routines frattali, o 3d “esagerati”, eccetera.

13.4 Ottimizzazioni del Blitter

Alla fine faremo un altro esempio legato al Blitter. Questo genere di ottimizzazioni che abbiamo trattato fino ad adesso erano riferite solamente al 68000, quindi indipendenti dalla macchina a cui si riferivano, ora tratteremo delle ottimizzazioni legate all'hardware dell'Amiga, precisamente al Blitter. Come ben sapete il blitter è un potente coprocessore per lo spostamento di dati ben più veloce del 68000 di base (attenzione però che è più lento di un 68020+!). È bene trarre il massimo dal Blitter. In genere una filosofia che si adotta per il blitter è che prima inizio il trasferimento dei dati e prima finisco. Tuttavia dovete tenere sempre ben presente il bit chiamato blit nasty che è in grado di dare maggiore priorità al Blitter rispetto alla CPU, in pratica il bus per il trasferimento dei dati sarà per la maggior parte del tempo del Blitter, vediamo un esempio:

```

1  a6=$dff000
2  ; Supponiamo di aver inizializzato tutti i registri
3
4  Move.w d0,$58(a6)      ; BLTSIZE - Il Blitter inizia
5  Wblit:
6  Move.w #$8400,$96(a6)  ; Abilitiamo il blit nasty
7  Wblit1:
8  Btst #6,2(a6)          ; Attendiamo che il blitter abbia finito
9  Bne.s Wblit1
10 Move.w #$400,$96(a6)   ; Disabilitiamo il blit nasty
11 ....

```

Questo è un caso banale, perché mentre il blitter lavora la CPU potrebbe fare qualche altra cosa, quindi quel ciclo di attesa è antiproduttivo. Infatti, in computer con sola CHIP RAM questa funzione blocca del tutto il processore, e forse non andrebbe mai usata. Ma il caso in cui possiamo e dobbiamo abilitare il blit nasty è nei casi in cui dobbiamo copiare a video un bitplane bob per bitplane, allora, poiché di solito la CPU deve attendere tra una blittata ed una altra, possiamo tranquillamente abilitare il blit nasty. Vediamo un esempio:

```

1  BLITZ:                      ; I registri sono già stati abilitati
2  Move.w #$8400,$96(a6)      ; Abilitiamo il nasty
3  Move.l Plane0,$50(a6)      ; Puntatore al canale A
4  Move.l a1,$54(a6)          ; Puntatore al canale D
5  Move.w d0,$58(a6)          ; Go Blitter!!!
6  WBL1:
7  Btst #6,2(a6)              ; Qui la CPU deve attendere la fine...
8  Bne.s WBL1                 ; quindi il blitter deve andare al massimo!
9  Move.l Plane1,$50(a6)      ; Puntatore al canale A
10 Move.l a2,$54(a6)          ; Puntatore al canale D
11 Move.w d0,$58(a6)          ; Go Blitter!!!
12 WBL2:
13 Btst #6,2(a6)              ; Come sopra
14 Bne.s WBL2
15 Move.l Plane2,$50(a6)      ; Idem
16 Move.l a3,$54(a6)
17 Move.w d0,$58(a6)
18 WBL3:
19 Btst #6,2(a6)
20 Bne.s WBL3
21 Move.w #$400,$96(a6)       ; A questo punto il nasty può anche essere
22 Rts                        ; disabilitato.

```

Questo esempio mi dà la possibilità di farvi notare una caratteristica del blitter che è quella di non modificare alcuni valori dei suoi registri, ad esempio nei registri di modulo (BltAMod, BltBMod, etc...). Troveremo al termine della blittata gli stessi valori, quindi non c'è bisogno di inizializzarli se il modulo sarà lo stesso per la prossima blittata. La stessa cosa vale per i registri tipo BltCon0, BltCon1, BltFWM, BltLWM, ma ciò non è più valido per i registri puntatori poiché questi lavorano con un indirizzamento con incremento. Ciò ci suggerisce la seguente cosa: supponiamo di avere un bob di 5 bitplane da piazzare uno per uno in un bitplane "video", quindi ogni volta carichiamo il puntatore al bitplane "video" nel registro D e il puntatore al bob in A: dopo la prima blittata il registro D sarà caricato con lo stesso valore più una certa

quantità per puntare al bitplane successivo, ma lo stesso sarà inutile farlo con il canale A, poiché se il nostro bob è stato memorizzato in memoria come bitplane successivi, allora dopo la prima blittata il canale A punterà automaticamente al secondo bitplane del bob. Possiamo ottenere validi risultati facendo anche nel seguente modo. Riserviamo una zona di memoria con tutti i valori da passare ai registri del blitter (nel nostro caso la zona parte da DataBlit). Quindi in alcuni registri indirizzi carichiamo gli indirizzi dei registri del blitter in modo tale da poterci accedere più velocemente, e copiamo i dati preconfezionati per far partire il blitter, accedendo direttamente ai registri della CPU. Vediamo un esempio:

```

1  Lea    $dff002,a6      ; a6 = DMAConR
2  Move.l DataBlit(pc),a5 ; quindi a5 punta a una tabella di valori
3  ; precalcolati
4
5  ; Carichiamo ora i registri indirizzi
6
7  Lea    $40-2(a6),a0     ; a0 = BltCon0
8  Lea    $62-2(a6),a1     ; a1 = BltBMod
9  Lea    $50-2(a6),a2     ; a2 = BltApt
10 Lea    $54-2(a6),a3     ; a3 = BltDpt
11 Lea    $58-2(a6),a4     ; a4 = BltSize
12 Moveq  #6,D0           ; d0 costante per il controllo dello stato
13 ; del blitter.
14 Move.w (a5)+,D7         ; Numero di blittate
15 Move.w #$8400,$96-2(a6) ; Abilitiamo il nasty
16 BLITLOOP:
17 Btst    d0,(a6)          ; Attendiamo come sempre la fine di qualche
18 Bne.s   BLITLOOP        ; operazione.
19 ; Prima di guardare qui sotto facciamo un
20 ; osservazione, se in a0 ho il valore $40000
21 ; ed eseguo le istruzioni in tre casi distinti
22 ; a) Move.b #"1", (a0)
23 ; b) Move.w #"12", (a0)
24 ; c) Move.l #"1234", (a0)
25 ; otterrò la seguente cosa:
26 ;      (a)      (b)      (c)
27 ; $40000      "1"      "1"      "1"
28 ; $40001      "0"      "2"      "2"
29 ; $40002      "0"      "0"      "3"
30 ; $40003      "0"      "0"      "4"
31 ; Ora faremo una cosa del genere...
32 Move.l (a5)+,(a0)        ; $dff040-42 cioè Bltcon0-Bltcon1
33 Move.l (a5)+,(a1)        ; $dff062-64 cioè BltBMod-BltAMod
34 Move.l (a5)+,(a2)        ; $dff050 - Canale A
35 Move.l (a5)+,(a3)        ; $dff054 - Canale D
36 Move.l (a5)+,(a4)        ; $dff058 - BLTSIZE... START!!
37 Dbra    d7,BLITLOOP      ; Questo per d7 volte.

```

In questo esempio abbiamo adoperato varie tecniche di ottimizzazione, di cui abbiamo già parlato, in ogni caso vediamone qualcuna. Innanzitutto quando dobbiamo eseguire un gran numero di volte un ciclo ed all'interno c'è una operazione che coinvolge una costante (cioè un dato immediato), conviene mettere questo valore in un registro che non si userà nel ciclo, quindi effettuare l'operazione che coinvolge questo valore direttamente col registro che lo contiene, evitando un accesso alla memoria. Nel nostro caso abbiamo adoperato questa strategia caricando il valore del bit da testare, per controllare se il blitter aveva terminato il suo compito, nel registro d0. In pratica abbiamo adottato una delle prime regole di cui vi ho parlato all'inizio cioè cercare sempre di tenere i valori nei registri. Inoltre, abbiamo caricato \$dff002 come base e non \$dff000. Questo viene fatto spesso, per eliminare il tempo usato nel waitblit a calcolare l'offset:

```

1  Btst    #6,2(a6)        ; a6 = $dff000

```

è più lento di:

```

1  btst    d0,(a6)         ; a6 = $dff002, d0 = 6

```

Basta ricordarsi di mettere un -2 prima di (a6) per ottenere l'offset giusto:

```

1 $54-2(a6)      ; BltDpt
2 $58-2(a6)      ; BltSize
3 $96-2(a6)      ; DmaCon
4 ...

```

è importante che il waitblit sia veloce, in quanto prima si “accorge” che la blittata è finita, prima si comincia quella dopo! Per questo, evitate di chiamare il waitblit con un BSR, bensì mettetelo sempre sul posto, anche ripetendolo ogni volta che serve.

Lo stesso discorso che abbiamo fatto ora lo abbiamo applicato anche per i registri del blitter caricandoli nei registri della CPU, evitando accessi alla memoria (in pratica alla memoria ci accediamo comunque per inizializzare il blitter, ma evitiamo ogni volta di prelevare l'indirizzo dalla memoria). Inoltre abbiamo usato un trucco che chiunque programmi giochi o demo adopera, cioè invece di tenere in memoria le dimensioni del bob per poi calcolarsi il valore del bltsize, teniamo direttamente il valore del bltsize, questo lo abbiamo fatto tramite la tabella DataBlit. Tuttavia, come vi ho accennato sopra, mentre il blitter lavora il 68000 può fare qualche altra cosa, ad esempio se il blitter sta cancellando una zona di memoria, il 68000 da buon cristiano può dargli una mano, ad esempio:

```

1 btst    #6,2(a6)
2 WaitBlit:
3 btst    #6,2(a6)
4 bne.s   WaitBlit
5 Moveq   #-1,d0
6 Move.l  d0,$44(a6)          ; -1 = $ffffff
7 Move.l  #9f00000,$40(a6)
8 Moveq   #0,d1
9 Move.l  d1,$64(a6)
10 Move.l  a0,$50(a6)
11 Move.l  a1,$54(a6)
12 Move.w  #$4414,$58(a6)     ; Il blitter inizia a pulire...
13 Move.l  a7,OldSp
14 Movem.l CLREG(pc),d0-d7/a0-a6 ; Puliamo i registri
15 Move.l  Screen(pc),a7      ; Indirizzo della zona da cancellare
16 Add.w   #$a8c0,a7          ; andiamo alla sua fine (+$a8c0)
17
18 Rept    1024                ; Il 68000 inizia a pulire
19 Movem.l d0-d7/a0-a6,-(a7)   ; Pulisci 60 bytes per 1024 volte
20 EndR
21
22 Lea     $dff000,a6
23 Movea.l OLDSP(pc),a7
24 Rts
25
26 CLREG:
27 ds.l    15

```

Come vedete, qua il blitter e la cpu puliscono “in contemporanea” mezzo schermo per uno. Naturalmente in questo caso il nasty bit non deve essere settato, o la cpu non può pulire in pace.

Il miglior metodo per aumentare le prestazioni del proprio programma rimane però quello di migliorare i propri algoritmi, molto spesso. Non pensiate ad esempio che implementare in assembly un pessimo algoritmo di ordinamento, come il Bubble Sort, sia più veloce del miglior algoritmo di ordinamento, come il Quick Sort, implementato in C. Se il vostro algoritmo non vuole proprio girare più velocemente anche dopo aver adoperato le migliori tecniche di ottimizzazione, bè, allora cancellatelo e riscrivetelo completamente con un algoritmo migliore **in partenza**. Ed anche se siete in possesso del miglior algoritmo, cercate sempre di ottimizzarlo in modo da farlo girare su delle macchine anche non veloci, non come nel mondo dei PC dove un programmatore di un 486 si sente soddisfatto solo se il suo codice gira velocemente sulla propria configurazione. Che ci vuole a fare routines veloci, se poi sulla confezione del gioco o del programma si legge: CONFIGURAZIONE MINIMA: PENTIUM 60Mhz con 8MB di RAM.

LEZIONE 14 - FONDAMENTI DI ACUSTICA E AUDIO DIGITALE

Autore: Alvise Spanò

Come tutti senz'altro saprete, il suono altro non è che *un'onda*, ovvero, secondo la definizione fisica, la “*propagazione di una perturbazione in un mezzo*”; nel caso dei suoni che siamo abituati ad ascoltare, la perturbazione (onda) viene inizialmente emessa dalla vibrazione della materia (molecole e/o atomi), ed il mezzo è l'aria (un'onda è una oscillazione, un continuo scambio di energia cinetica e potenziale, quindi meccanica, tra molecole e/o atomi (fisicamente è la variazione di pressione tra particelle di materia), e pertanto *necessita* di materia per esistere e diffondersi: nel vuoto, ad esempio, non si propaga alcuna onda sonora, né alcun altro tipo di vibrazione tra due corpi distinti e separati (le radiazioni elettromagnetiche soltanto - almeno sinora, le uniche di cui se ne è alla conoscenza - riescono a muoversi anche nel vuoto grazie alla loro duplice natura fisica sia di corpuscolo dotato di massa - seppur piccolissima - (fotone) sia di onda)). Il modello fisico adottato per descrivere questo continuo cedimento di energia da un punto all'altro del mezzo presenta il grafico della funzione goniometrica *SENO* (= sen = sin), ovvero di una *SINUSOIDE*.

Funzione d'onda $y = f(x \pm v * t)$ Ad esempio l'equazione d'onda armonica:

$$y = a \sin(k(x \pm vt)) = a * \sin(k * (x \pm v * t))$$

- y = *variabile dipendente* in un grafico cartesiano bidimensionale (x,y), l'ordinata y rappresenta le “quote” di ogni punto x dell'oscillazione.
- a = *coefficiente di amplificazione dell'onda* come ben saprete, $-1 \leq \sin(x) \leq 1$ (seno di x (x = qualsiasi numero reale) compreso tra -1 e 1 , estremi inclusi), quindi per ottenere un'oscillazione che vari da $-a$ ad a ($-a \leq \sin(x) \leq a$) è necessario moltiplicare $\sin(x)$ per un numero reale a .
- k = *frequenza dell'onda* variando questo parametro si varia la frequenza, e, in modo inversamente proporzionale, il *periodo* dell'onda, ovvero l'intervallo minimo, lungo l'asse della

variabile indipendente (in questo caso x) per cui la sinusoidale è ciclabile, ovvero il minimo intervallo dopo il quale l'onda assume le medesime caratteristiche. Introduciamo a questo punto il concetto di *lunghezza d'onda* (= spazio percorso nel periodo = distanza tra due creste di cicli adiacenti), ed approfondiamo quello di frequenza: numero di volte in cui vengono assunte le stesse caratteristiche dall'onda in una data unità di tempo, ovvero quante volte al secondo viene letto il periodo (= cicli al secondo); di solito si considera il minuto secondo [s] come unità di tempo e l'Hertz [$\text{Hz} = [\text{s}]^{-1} = \frac{1}{[\text{s}]}$] come unità di misura della frequenza.

- $x = \text{variabile indipendente}$ in un grafico cartesiano bidimensionale (x, y) , l'ascissa x rappresenta un punto lungo una dimensione spaziale rettilinea prestabilita in un dato istante; x appartiene ai numeri reali e, teoricamente, non ha limitazioni, ovvero abbraccia tutta la retta provocando nel piano considerato (x, y) una sinusoidale che prosegue all'infinito sia a sinistra che a destra dell'origine degli assi $O(0, 0)$: ciò fa facilmente intuire come la funzione seno goda di una periodicità. Per esempio, se $k = 1$, il periodo dell'onda è di 360 gradi (= $2 * \pi$ radianti) e la frequenza è 1 Hz; se $k = 2$, il periodo è 180° (= π rad) e la frequenza è pari a 2 Hz; e così via.
- $v = \text{velocità di propagazione}$ indica la velocità in senso cinematico con cui si sposta nello spazio un punto dell'onda. Da notare che $v = \text{LUNG_ONDA} * \text{FREQ.}$
- $t = \text{istante di tempo}$ tenete presente che $v * t = s == \text{spazio}$, ed s è la distanza tra due punti corrispondenti della stessa perturbazione presa in t diversi, pertanto, aggiungendo/-sottraendo s a/da x si ottiene nel tempo una propagazione, un movimento nello spazio dell'onda. È da notare che con $(x + s)$ l'onda si muoverà a sinistra, e con $(x - s)$ a destra lungo l'asse delle ascisse.

P.S.: mi scuso per la frettolosità delle spiegazioni e l'assenza di dimostrazioni, ma non mi sembra questa l'occasione adatta per dilungarsi eccessivamente su argomenti che non riguardano direttamente l'assembler ed il coding in generale; per cui vi prego di prendere così come sono le suddette delucidazioni e non preoccupatevi troppo se non avete capito nel profondo la fisica delle onde: non vi servirà alla fin fine per inserire una musica in un gioco o demo.

N.B.: Tenete presente che x e y rappresentano due qualsiasi caratteristiche del fenomeno di propagazione. Nel caso di onde sonore, noi considereremo x e y come due dimensioni spaziali che descrivono su di un piano l'onda esaminandone una sezione.

Ora siamo in grado di trasportare quanto appreso dagli accenni di fisica generale dei fenomeni di propagazione in acustica vera e propria, definendo il concetto di ARMONICA: un suono - tra parentesi, inesistente in natura e riproducibile solo con strumenti elettronici, quali il computer appunto - che presenta la forma d'onda (= grafico (x, y) della perturbazione lungo tutta la sua durata) di una sinusoidale. Possiamo dire che l'armonica è il modello del *suono base*, che unendosi a molti altri crea tutti i suoni *non puri*. Sempre la fisica distingue in un suono 3 qualità affinché possa essere descritto:

1. *altezza* che si distingue di suoni *puri* (costituiti da una sola armonica - inesistenti in natura) e *naturali* (costituiti da più armoniche sovrapposte in uno stesso intervallo di tempo - alcuni di quelli che esistono in natura presentano migliaia di armoniche di periodo diverso), e ne qualifica la frequenza, alterando la quale vengono prodotte le varie note.
2. *intensità* che può essere vista come una sorta di "volume" del suono, e, nel caso di armoniche, è direttamente proporzionale all'amplificazione a (valore assoluto delle quote Y delle creste) della sinusoidale.

3. *timbro* che qualifica la forma d'onda del suono a prescindere dai precedenti due parametri, quindi descrive sostanzialmente lo strumento musicale o, più generalmente, ciò per cui noi distinguiamo un suono da un altro indipendentemente dalla sua altezza e intensità.

Lasciamo perdere le varie formule per calcolare l'intensità sonora ed i livelli di pressione e di intensità [decibel = dB], che comunque non toccano direttamente la musica elettronica ma solo l'acustica come ramo della fisica che si occupa della diffusione dei suoni nell'ambiente e della progettazione di apparecchi in grado di riprodurre (altoparlanti, ecc.) o captare (microfoni, ecc.) suoni, e soffermiamoci sull'altezza ed il timbro.

Intanto per cominciare, il timbro, di per se, è un parametro inesistente: il computer non distingue i suoni, e converte i dati digitali che ha in memoria in segnali analogici (che trasferiscono intensità elettriche e non bit) secondo una frequenza di lettura ed un dato valore di sottoamplificazione, non curandosi della differenza tra suono e rumore – due concetti che soltanto noi uomini abbiamo introdotto con significati diversi e riconducibili al senso estetico.

Del timbro come parametro, dunque, non si può propriamente parlare – perlomeno a livello elettronico –, anche se esistono sofisticati algoritmi di identificazione e confronto del timbro di forme d'onda differenti che potrebbero tornare utili a chi è intenzionato a programmare routine di riconoscimento vocale, ad esempio; ad ogni modo, non è certo questa la sede adatta a discutere di tanto complesse applicazioni del mondo della sintesi e della elaborazione sonora.

Passiamo quindi al parametro più importante - per quanto ci riguarda - : l'altezza dei suoni è direttamente legata, nel caso di suoni NATURALI, alla frequenza dell'armonica fondamentale (*frequenza fondamentale*) che li distingue, e, nel caso di suoni *puri*, alla frequenza dell'*unica* armonica che li costituisce. Facciamo un esempio: abbiamo un suono puro (armonica) di periodo (=durata) X; esso può essere emesso a qualsiasi frequenza, dipende solo dalla "*velocità di lettura*" del suono da parte dell'emittente: per esempio, se questo "legge" l'armonica per tutto il suo periodo 2 volte al secondo la frequenza sarà pari a 2 Hz; in un secondo momento, si presenta pure il problema della diffusione acustica del dato suono nell'aria, che, però, è presto risolto: la velocità di propagazione del suono nell'aria è di 380 m/s, e quindi, sapendo che $VELOCITÀ = LUNGH.ONDA * FREQ.$, è estremamente semplice ottenerne la lunghezza d'onda, che coincide, in termini di dimensioni spaziali, al periodo (che è espresso in secondi). A questo punto entra in gioco un nuovo parametro, per trattare gli strumenti di generazione del suono: la velocità di lettura, o *velocità / frequenza di campionamento*. Ora però, per proseguire, è necessario spiegare come gli strumenti elettronici trattano il suono e come lo processano prima di passarlo all'amplificatore.

Tramite l'uso di un campionatore (digitalizzatore audio) ed un software adeguato, è possibile convertire i suoni provenienti da una sorgente sonora (microfono, CD, ecc.) in dati numerici (campioni digitali) ognuno dei quali descrive la "quota" di un intervallo (di una "minuscola fettina" – per parlare in termini scientifici –) lungo le ascisse del grafico (x,y) della forma d'onda. Più sono i campioni che "catturiamo", più definita e vicina alla realtà fisica sarà il suono digitale. Ad esempio, se abbiamo un suono naturale (quindi, non armoniche) della durata di 2 secondi (che assumiamo come periodo, visto che non è possibile individuare un intervallo minimo inferiore per cui l'onda si cicli), e siamo in grado di ricavarne la frequenza fondamentale, è sufficiente campionare in memoria con una DOPPIA frequenza di campionamento (teorema di Nyquist, spiegato più avanti), per ottenere una riproduzione fedele e definita acusticamente del suono; se, ad esempio, la frequenza fondamentale è di 2 kHz (= 2000 Hz), dovremmo registrare 2000 campioni al secondo, per un totale di 4000 campioni ($2000 \text{ camp/s} * 2 \text{ s} = 4000 \text{ camp}$).

Ogni campione (= sample), in memoria occupa 8 bit (= 1 byte), nell'Amiga, che adotta una definizione sonora ad 8 bit, appunto. Col numero espresso ad 8 bit è possibile descrivere quote Y comprese tra -128 ($= -(2^8)/2 = -2^{(8-1)} = -2^7$) e 127 ($= (2^8)/2 - 1 = 2^{(8-1)} - 1 = 2^7 - 1$) (lo zero ha segno positivo, in binario: in effetti i numeri positivi (da 0 a 127) sono 128 come quelli negativi), estremi inclusi, di ogni singolo campione, per un totale di 256 ($= 2^8$) valori esprimibili.

N.B.: è importante notare che la forma d'onda oscilla tra il I (+) ed il II (-) quadrante, divisi dall'ascissa di quota 0; NON considerate -128 come 0 e +127 come 255: NON è possibile traslare l'onda sul I quadrante e rendere tutto positivo, *i conti non tornerebbero nè al chip sonoro, nè ad un'eventuale elaborazione del suono per effetti speciali via software.*

Considerate ogni sample come un byte ad 8 bit con segno (= MSB = bit 7)

È importante sottolineare come sintesi digitale ad un numero di bit maggiore di 8 offrirebbero una qualità sonora superiore, a parità di frequenza di campionamento. Per esempio, i lettori CD leggono sample a 16 bit (= 2 byte = 1 word), il che significa che il range delle quote varia da -32768 ($= -(2^{16})/2 = -2^{(16-1)} = -2^{15}$) e 32767 ($= (2^{16})/2 - 1 = 2^{(16-1)} - 1 = 2^{15} - 1$), per un totale di 65536 ($= 2^{16}$) valori esprimibili.

Ciò non vuol dire, però, che il valore 32767 (picco positivo) del CD corrisponde ad una quota maggiore di quel sample rispetto al valore 127 dello stesso campione reso ad 8 bit: l'uscita sonora sarà uguale, solo che a parità di range fisico la sintesi a 16 bit offre molta più definizione (in sostanza, ad 8 bit ho 256 numeri per esprimere un suono, compreso tra due picchi positivo e negativo fisicamente costanti, che però a 16 bit viene sintetizzato con una gamma di valori molto superiore (65536) e quindi *con più precisione, con meno approssimazione delle quote*). In un certo senso possiamo affermare che gli 8 bit del primo esempio corrispondono agli 8 bit alti (15:8) dei 16 bit del secondo, e gli 8 bit bassi corrispondono ad una sorta di approssimazione dopo una virgola fittizia posta tra il byte alto e quello basso della word del sample.

Torniamo ora a parlare della frequenza di campionamento, citando un celebre ed importante – tanto quanto complicato per quanto riguarda la dimostrazione – che enuncia che *"la risposta di frequenza è pari alla metà della frequenza di campionamento* (teorema di Nyquist): sostanzialmente, significa che se noi campioniamo a 10 kHz, verranno riprodotti fedelmente solo i suoni con frequenza minore od uguale a $10/2 = 5$ kHz (ecco spiegato il misterioso *doppio* scritto poco sopra circa la frequenza di campionamento necessaria per campionare il dato suono, conoscendone la frequenza fondamentale). È fondamentale campionare i suoni ad una frequenza adeguata per non sentire l'*aliasing*, che taglia le frequenze al di sopra della metà della frequenza di campionamento rendendole con uno sgradevole effetto di "disturbato".

Anche se il Paula (per la cronaca, nome proprio del chip sonoro dell'Amiga) adotta una precisione digitale a soli 8 bit, è possibile riprodurre suoni ugualmente di buona qualità campionando alle frequenze giuste ed evitando l'*aliasing*, anche se, comunque, non si può raggiungere la qualità di un CD, che campiona 16 bit a 44.1 kHz (44100 Hz) per ottenere una risposta di frequenza che spazia da 20 Hz a 22 kHz circa, che corrisponde circa al range delle frequenze udibili dall'orecchio umano (soggettivo: qualcuno potrebbe arrivare fino a 20 kHz ca.)

Colgo l'occasione per dire che i suoni di frequenza (fondamentale, sott'inteso) minore di 20 Hz vengono chiamati *infrasuoni*, e quelli di frequenza maggiore di 20-22 kHz *ultrasuoni*: entrambi **non** sono udibili dall'uomo.

Comunque, la maggior parte dei suoni naturali non ha una frequenza fondamentale superiore ai 15-16 kHz; è dunque sufficiente campionare a 32 kHz al massimo per riprodurre fedelmente quasi tutti i suoni esistenti? Ebbene, no ! Per un motivo molto semplice: poco sopra è stato spiegato che i suoni naturali sono costituiti da molte armoniche tra le quali ne è individuabile una fondamentale: potrebbe anche accadere che la frequenza fondamentale (che noi utilizziamo di solito per calcolare quella di campionamento) sia effettivamente la frequenza dell'armonica di periodo minore (quindi, di frequenza maggiore); in tal caso tutte le armoniche di frequenza maggiore di quella che noi assumiamo come fondamentale verrebbero tagliate e riprodotte con aliasing, abbassando sensibilmente la qualità sonora globale.

Sarebbe opportuno quindi, campionare a frequenza doppia rispetto alla frequenza dell'armonica più alta che compone il dato suono naturale

Dunque, l'intensità contraddistingue il "volume" del suono, ed essa **non** è costante rispetto alla frequenza: a frequenze molto alte o molto basse è necessario amplificarlo per essere percepito con la stessa intensità rispetto a suoni di frequenza media dall'orecchio, che l'evoluzione biologica (conseguenza dell'abitudine) lo ha evidentemente portato a sentire meglio quelli più diffusi in natura; cosa contraddistingue invece l'altezza di un suono? In termini puramente musicali, è presto detto: le *note*. Come senz'altro saprete le note musicali formano scale di 7 note per *ottava*, ognuna delle quali comincia con la nota DO (C, per la notazione anglosassone) e finisce con il SI (B, per la notazione anglosassone - il LA è la A); ad ogni ottava si raddoppia la frequenza, quindi ogni DO è di frequenza doppia rispetto a DO precedente (notate dunque che l'incremento di frequenze non è rettilineo, ma *esponenziale* in base 2). All'interno dell'ottava i rapporti tra le note della scala sono i seguenti:

DO	RE	MI	FA	SOL	LA	SI		(DO)
-----+-----+-----+-----+-----+-----+-----+-----+-----								
1	9/8	5/4	4/3	3/2	5/3	15/8		2

Nel caso in cui bisogni campionare suoni che poi verranno usati come strumenti in programmi di editing musicale (tipo SoundTracker, NoiseTracker o ProTracker), sarebbe sufficiente campionare ad una frequenza doppia rispetto alla frequenza fondamentale del suono, che, se si campiona da uno strumento musicale, corrisponde alla frequenza della nota suonata, almeno per praticità: se, ad esempio, necessitiamo di un pianoforte per la composizione di un modulo, possiamo campionare il LA3 (LA della terza ottava) a 880 Hz (LA3 = 440 Hz) e comunicare al tracker che la frequenza di campionamento corrisponde al LA3, penserà lui, dopo, a calcolare le frequenze giuste relative a quella di campionamento in base alle note che poniamo sullo spartito con quello strumento. Ora, sicuramente, vi porrete una domanda: ma basta campionare a 880 Hz, per non ottenere un aliasing? La risposta è no. Come abbiamo detto prima, bisognerebbe campionare al doppio della frequenza dell'armonica più acuta per riprodurre fedelmente la timbrica del pianoforte, ma è assai complicato ricavare tale frequenza (per non dire impossibile). Cosa fare, allora? Beh, provare e riprovare a campionare a varie frequenze (onestamente, ben più alte di 880 Hz) finché si ottiene una riproduzione ottimale dello strumento a quella nota e comunicare al tracker la frequenza di lettura del sample per tale nota. Come vedete, dunque, la faccenda è più complicata nella pratica che nella teoria!

Dopo questi inevitabili (e – spero – interessanti) cenni di audio digitale, passo alla spiegazione più specifica dell'hardware sonoro dei chip Original ed AA dell'Amiga (il Paula è l'unico chip custom che non ha mai subito miglioramenti dall'uscita del primo Amiga (1985, per la cronaca)). L'hardware presenta 4 canali DMA dedicati alle 4 voci del chip sonoro; queste 4 voci sono totalmente indipendenti e sono raggruppate a 2 a 2 per cassa ottenendo le voci 1+4 per

la via sinistra e 2+3 per la destra in stereo. Tutte e 4 le voci, inoltre, possiedono propri registri hardware:

AUDxLCH \$dff0y0 =	Locazione dei dati da leggere (word alta)
AUDxLCL \$dff0y2 =	Locazione dei dati da leggere (word bassa)
AUDxLEN \$dff0y4 =	Lunghezza del DMA (in word)
AUDxPER \$dff0y6 =	Periodo di campionamento in lettura
AUDxVOL \$dff0y8 =	Volume
AUDxDAT \$dff0ya =	Dato del canale (2 byte = 2 sample alla volta)

N.B.: Per ogni 'x' sostituite un numero da 0 a 3, corrispondente alla voce desiderata; per ogni 'y' sostituite un numero esadecimale da \$a a \$d relativo alle voci da 0 a 3.

AUDxLCH-AUDxLCL Costituiscono il valore di latch, non il puntatore del DMA ai dati, pertanto, una volta impostato, non incrementa come accade per i plane, per gli sprite o per i canali del blitter, ma è simile ai registri di locazione del copper, il valore dei quali viene automaticamente riinserito nei registri interni di puntatore quando ce n'è bisogno. * Visto che questi due registri a 16 bit sono adiacenti, è comodo impostarli con un singolo MOVE.L del 68000 del tipo:

```
1 MOVE.L #miosample,AUDxLCH *.
```

N.B.: d'ora in poi, con AUDxLC mi riferirò alla coppia dei due registri, ad una sorta di registro di locazione unico a 32 bit.

AUDxLEN Esprime la lunghezza in word del sample da suonare. Se, per esempio, abbiamo in memoria un sample di 500 byte, bisogna impostare questo registro (di uno dei 4 canali desiderato) con un valore di 250. N.B.: Come per il blitter, scrivendo 0 questo registro vengono letti 128 kB di sample.

AUDxPER Questo registro serve a specificare la frequenza di lettura del DMA in modo un pò bizzarro – apparentemente –, ma che torna comodo e veloce all'hardware: bisogna impostarlo con il PERIODO DI CAMPIONAMENTO di ogni singolo campione del suono, un valore che esprime il tempo (in cicli di CLOCK del DMA di sistema = 3546895 Hz (PAL), 3579545 Hz (NTSC)) che il DMA deve attendere (funziona da decrementatore: -1 per ciclo di clock) prima di trasferire un'altro campione. Ecco una formula per calcolare il valore da inserire in questo registro, data la frequenza di campionamento (che è molto più pratica da gestire): $PER = \text{CLOCK} / \text{freq. [Hz]}$ Per esempio, se dobbiamo campionare un LA3 armonica – ammesso che ne troviamo un sorgente naturale... – di frequenza 440 Hz, di periodo 1 secondo, dobbiamo adottare una frequenza di campionamento di 880 Hz, per cui ecco il periodo di campionamento da inserire nel registro AUDxPER per leggere alla giusta frequenza in 1 secondo il sample in memoria: $PER = 3546895 / 880 = 4030$ (PAL) N.B.: I dati audio vengono fetchati dal DMA in 4 slot del color clock (16bit=2 sample per canale) per linea di scansione orizzontale. Le linee di scansione PAL sono 312.5 (312=SHF, 313=LOF) per raster, e ci sono 50 raster al secondo; per cui, la frequenza massima di lettura (leggendolo a tutti i cicli assegnati disponibili) sarebbe = $2 \text{ BytePerLinea} * 312.5 * 50 = 31300$ Hz circa: **ma questa velocità è solo teorica**, in quanto è impossibile impostare un giusto periodo di campionamento che coincida perfettamente con i cicli assegnati al DMA audio ad ogni linea di raster: potrebbe verificarsi la fine di un conteggio del periodo dal DMA nel bel mezzo di una linea di scansione (o, sfiga totale, il ciclo dopo lo slot assegnato alla data voce), e l'hardware è costretto ad attendere la linea dopo per avere i dati da suonare, mentre, se il periodo di campionamento è breve, la successiva fine di conteggio potrebbe

avvenire nella linea stessa, in assenza di nuovi dati. In sostanza, il periodo minimo deve permettere all'hardware di percorrere **almeno** un'intera linea di raster: il suono non esce quando viene letto dal DMA, ma quando termina il conteggio interno dal valore del periodo in AUDxPER: il DMA audio legge i dati solo durante i cicli assegnati (peraltro ad altissima priorità – come quelli del DMA dei disk drive –, per evitare distorsioni e rallentamenti dovuti a “sovraffollamento” di canali – vedi: “bit plane che rompono sempre”) e li conserva in AUDxDAT fino alla fine del periodo di campionamento, *che può avvenire in qualsiasi momento*, e viene generato il suono; *per questo motivo non si può abbassare il periodo minimo di campionamento oltre 123 (=28836 Hz): per permettere al DMA di leggere comunque almeno un'altro dato prima di suonare, alla linea dopo*. A questo punto la domanda è d'obbligo: “Come mai si pone 123 come periodo minimo, quando i cicli di clock per linea (ovvero il numero di decrementi) sono 226.5 (226=LOF, 227=SHF)?”. Ecco la risposta: prima è stato accennato che il DMA trasferisce 16 bit (=2 byte) “al colpo” per voce, per cui possiede 2 sample suonabili ad ogni linea, e, suonato il primo, può suonare anche il successivo durante la stessa linea di raster, per il fatto che non lo ha già letto; con 123, infatti, possono accadere, al massimo, 2 fine-conteggi durante una stessa linea, ed il problema non si pone. Il periodo minimo teorico, dunque, dovrebbe essere di 227 (per tenerci larghi) / 2 = 114 (sempre approssimando per eccesso), che coincide più o meno (la corrispondenza tra periodo e frequenza di campionamento NON è biunivoca: c'è sempre una certa approssimazione) con la frequenza massima teorica di 31300 Hz circa. Ma, come si è detto sopra, non è raggiungibile con precisione dall'hardware. *123 è il periodo minimo impostabile = 28836 Hz*

AUDxVOL In questo registro va specificato il volume dell'emissione del suono nel relativo canale con valori compresi tra 0 e 64 (inserendo '64' non vi è alcuna diminuzione di dB = volume massimo).

CONSIGLIO!!! Quando campionate cercate di sfruttare tutta la fascia di valori da -128 a 127 anche per suoni a bassa intensità al fine di avere sempre la massima precisione, e, tutt'al più, abbassate il volume in questo registro.

AUDxDAT è il buffer momentaneo che il DMA utilizza prima che i dati vengano spediti ai convertitori D/A (Digitale/Analogico) e che vengano emessi i segnali all'esterno dell'Amiga. Esso contiene 2 byte di dati audio (il DMA trasferisce 16 bit alla volta dalla RAM – ecco spiegato il motivo per cui l'AUDxLEN deve essere espresso in word !) che vengono inviati 1 ad 1 al DAC (Digital-Analogical Converter).

SCONSIGLIO!!! è anche possibile impostare questi registri con la CPU quando il DMA è spento e far suonare ugualmente il computer : (.

Un esempio di impostazione dei registri per suonare un sample di 23 kB ad una frequenza di 21056 Hz posto in memoria alla locazione \$60000 (chip RAM) a volume massimo in stereo, con la voce 2 e la 3 (terzo e quarto canale):

```

1 PlaySample:
2     lea     $dff000,a0      ; base dei chip custom in a0
3     move.l #$60000,$c0(a0) ; punta AUD2LC a $60000
4     move.l #$60000,$d0(a0) ; punta anche AUD3LC a $60000
5     move.w #11776,$c4(a0)  ; AUD2LEN = 23 kB = 23*1024 = 23552 B [...]
6     move.w #11776,$d4(a0)  ; anche AUD3LEN = [...] = 23552/2 = 11776 word
7     move.w #168,$c6(a0)    ; AUD2PER = 3546895/21056 = 168
8     move.w #168,$d6(a0)    ; imposta AUD3PER come AUD2PER
9     move.w #64,$c8(a0)     ; volume massimo per AUD2VOL

```

```

10      move.w    #64,$d8(a0)      ; volume massimo anche per AUD3VOL
11      move.w    #$800c,$96(a0)   ; accende il DMA dei canali 2 e 3 in DMACON

```

Passiamo ora a spiegare cosa accade quando si accendono i DMA dei canali (oltre al fatto che – chiaramente – si sente il sample...):

1. Il valore contenuto in AUDLC viene inserito nei registri interni di puntamento ed il DMA comincia a trasferire nei registri dati 2 byte alla volta. Da adesso il registro AUDLC **può** anche venire cambiato: l'hardware, appena finito di trasferire tutto il sample, ricomincerà daccapo (LOOPANDO ALL'INFINITO).
2. Appena inserito il valore di AUDLC nei registri interni viene sparato un interrupt di LIVELLO 4, che nei registri INTENA e INTREQ si suddivide in 4 sottointerrupt, uno assegnato ad ognuno dei 4 canali audio:

LIVELLO	IRQ	BIT INTENA/INTREQ	CANALE
4		10	3
4		9	2
4		8	1
4		7	0

Grazie a questi interrupt è possibile, ad esempio, impostare un nuovo sample da suonare appena finito di suonare quello corrente semplicemente puntando i registri locazione ad un altro sample, ed ottenendo un perfetto collegamento fra i due (a patto che le due forme d'onda rispettivamente terminino e comincino similmente).

3. alla fine del trasferimento ricomincia tutto dal punto (1). *I registri non vengono mai alterati.*

Certo, tutto questo è interessante – voi direte –, ma come fare per far suonare al computer una canzone di 10 minuti senza campionarsi decine di megabyte di dati?

Proprio per questo scopo sono stati inventati i Tracker: programmi atti a scrivere musica che richiedono che soltanto gli strumenti di base siano campionati ricavando le varie note variando la frequenza di lettura degli stessi. Sono inoltre dotati di un editor dove comporre lo spartito suddiviso in 4 tracce (una per voce), ognuna delle quali può suonare qualsiasi strumento in qualsiasi momento, ma pur sempre uno alla volta (in totale si può ottenere un massimo di 4 strumenti effettivamente in contemporanea).

Sarebbe assai complicato spiegare qui tutte le varie possibilità di un tracker, pertanto vi invito a procurarvene uno (tipo il ProTracker: attualmente il miglior tracker a 4 tracce – ebbene sì, ne esistono anche a più tracce, che mixano in tempo reale le note di più tracce in un'unica voce; purtroppo, però, tale procedimento è estremamente lento e non potrebbe essere adottato per suonare la musica di un demo o di un videogioco, visto che la macchina, in simili casi, ha ben altro da fare che perdere tutto il tempo a suonare...). La "filosofia" dei tracker, comunque, non cambia: tutti forniscono delle – cosiddette – music routine, che sono dei sorgenti asm che sfruttando spesso interrupt di livello 6 (collegati al CIAAB) o loop di attesa sincronizzati col pennello elettronico, e suonano in tempo reale i moduli (song + samples = spartito + strumenti) creati con il relativo tracker (o compatibili, ovvero, che salvano la struttura del modulo allo stesso modo). Adattare tali routine ai propri sorgenti è semplicissimo: in linea di massima – ognuna ha le sue convenzioni: leggetevi i .doc del vostro tracker - è sufficiente lanciare una subroutine di inizializzazione che imposta gli interrupt ed i canali DMA – alcune anche i timer del CIAAB; il CIAA resta intatto visto che viene usato dall'Exec per temporizzare i processi/task – e lanciare la

subroutine di play ad ogni raster (vi conviene farlo all'inizio del codice di interrupt del vertical blank - se siete sotto sistema operativo aggiungete un server di interrupt 5 (VBLANK, livello 3) a priorità alta, per fare tutto durante l'intervallo di vertical blank, prima che i plane comincino a fetchare ed a rallentare tutto); prima di quittare il vostro demo/gioco - o programma che sia – **ricordate** di lanciare la subroutine di restore degli interrupt, dei DMA e dei timer all'OS.

Un altro importante paragrafo sull'hardware audio dell'Amiga riguarda la modulazione del suono proveniente dal DMA delle 4 voci. Cos'è la *modulazione*? Avrete senz'altro udito in moltissime canzoni l'effetto *didissolvenza audio* (il progressivo e lento abbassarsi del volume): ebbene, questo semplice effetto è un particolare tipo di modulazione.

La MODULAZIONE consiste nell'alterare uno o più parametri di un suono durante ed oltre il suo periodo; i parametri in questione sono, ovviamente, intensità (ampiezza) ed altezza (frequenza).

A quali effetti corrispondono – a livello di percezione sonora – modulazione in ampiezza e modulazione in frequenza? La prima, come abbiamo già accennato, trova una comune applicazione nelle dissolvenze solitamente al principio ed al termine di un brano musicale; un familiare esempio di modulazione in frequenza è lo slide sulle corde di una chitarra (od uno strumento a corda): in sostanza, una sfumata fusione di note adiacenti partendo da una data frequenza ed arrivando ad un'altra passando gradualmente per tutte quelle intermedie con una certa velocità (od addirittura una certa accelerazione). È anche possibile modulare sia in ampiezza che in frequenza contemporaneamente, ottenendo uno strano effetto riconducibile ad un fenomeno di esperienza quotidiana: *l'effetto Doppler*.

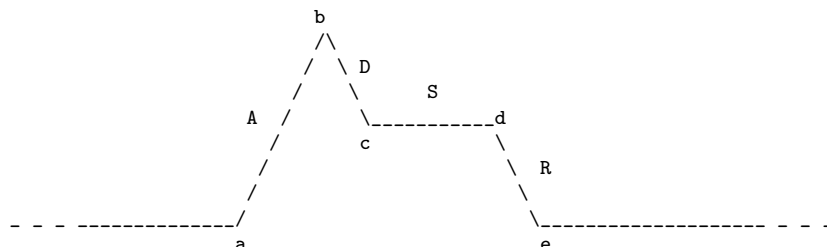
Brevemente, consiste nel cambiamento (appunto, modulazione) di intensità ed altezza dei suoni provenienti da una fonte in moto relativo rispetto all'ascoltatore: quando state camminando per strada, notate che il rumore delle macchine che vi si avvicinano e poi vi sorpassano non mantiene i medesimi parametri nelle diverse posizioni dell'auto (fonte) rispetto a voi (ascoltatore) ma, innanzitutto, si fa più alto di volume in modo inversamente proporzionale alla distanza tra voi e la fonte, e, se ci badate bene, l'intensità non è l'unica a mutare nel tempo: anche la frequenza del rumore emesso dal motore è minore quando la macchina è distante.

Non penso sia il caso di riportare l'equazione che descrive il fenomeno in funzione della velocità dei due corpi e della distanza, in quanto il problema non tocca da vicino l'argomento "modulazione su Amiga"; equazione che, comunque, potrete facilmente trovare in qualunque libro di fisica o di acustica generale anche delle scuole medie superiori.

Passando, appunto, alla modulazione sull'Amiga, sono costretto a deludervi subito: sebbene il Paula possieda dei particolari modi operativi per modulare i suoni provenienti da un canale sia in ampiezza che in frequenza, non si usa mai utilizzare questa soluzione hardware perché presenta una terribile restrizione: per modulare intensità ed altezza il DMA deve leggere dalla RAM dei valori da inserire rispettivamente nei registri AUDxVOL e/o AUDxPER mentre un altro DMA legge i valori veri e propri del sample da suonare e poi distorcere; tale processo ha la limitazione che il DMA che legge dalla tabella dei valori di modulazione deve essere uno dei canali audio, per cui per modulare, ad esempio, il suono letto dal canale 0 sia in frequenza che in ampiezza siamo costretti ad usare i canali 1 e 2 per leggere le rispettive tabelle, col risultato di sprecare 3 canali per generare un solo suono modulato. Tutti gli effetti di modulazione utilizzati dai tracker sono gestiti dalla CPU, che imposta "di cattiveria" i registri di volume e periodo della voce desiderata mentre il DMA legge ignaro il suo sample. Così facendo, non viene sprecato alcun canale, anche se la CPU resta per un pò impegnata a calcolare in tempo reale gli effetti sonori.

L'Amiga NON possiede nemmeno un sintetizzatore FM (Frequency Modulation) in grado di creare timbri diversi partendo da una stessa forma d'onda. Questi applicano modulazioni sia in

ampiezza che in frequenza secondo 4 parametri chiamati ADSR, dalle iniziali delle 4 fasi principali di un suono sintetizzato: Attack, Decay, Sustain, Release. Il grafico di questa modulazione è il seguente:



La prima fase è l'Attack, che consiste nel portare il volume e/o la frequenza dell'onda da 'a' a 'b' (ovvero da 0 ad un valore di picco massimo); dopodiché, il grafico scende per il tratto del Decay fino ad una quota 'c', alla quale si stabilizza per la durata del Sustain; infine ritorna a 0 da 'd' ad 'e' con il Release. "Giocando" acutamente con la posizione dei punti 'a','b','c','d' ed 'e' e con le durate delle varie fasi è possibile generare un'infinità di suoni anche partendo dal sample di una banale armonica. Purtroppo, queste nozioni non vi saranno utili per la programmazione del chip sonoro dell'Amiga, perciò passiamo alla descrizione di quei bit che mai vennero impostati nella storia di Amiga e del suo hardware... :) Il registro in questione è il famigerato ADKCON (\$dff09e), che possiede pure una copia di lettura (ADKCONR) all'indirizzo \$dff010:

bit	-	7:	USE3PN	Usa il canale 3 per non modulare nulla
		6:	USE2P3	Usa il canale 2 per modulare il PERIODO del 3
		5:	USE1P2	Usa il canale 1 per modulare il PERIODO del 2
		4:	USE0P1	Usa il canale 0 per modulare il PERIODO dell' 1
		3:	USE3VN	Usa il canale 3 per non modulare nulla
		2:	USE2V3	Usa il canale 2 per modulare il VOLUME del 3
		1:	USE1V2	Usa il canale 1 per modulare il VOLUME del 2
		0:	USE0V1	Usa il canale 0 per modulare il VOLUME dell' 1

Avrete senz'altro capito come funziona la faccenda: se, ad esempio, dovete modulare in ampiezza il canale 2, potete farlo solamente utilizzando l'1 come lettore dei valori da inserire nel registro AUD2VOL, per cui, dovete puntare i canali ai relativi dati e dare loro una frequenza di lettura. La modulazione, comunque, è un effetto semplice ma importante, che deve essere simulato via CPU - come già detto - per non occupare qualcuno dei già pochi canali audio di Amiga...

Concluderei permettendomi di affermare che la conoscenza di acustica digitale e del funzionamento dei chip sonori in generale, non è fondamentale come quella dell'hardware grafico o dell'asm di una CPU, ed è certo che la padronanza di questi è necessaria a chiunque, audiofili compresi; è altrettanto vero, però, che la vera cultura in materia prevede anche la conoscenza approfondita della teoria del suono digitale, che tanto viene decantata in questi ultimi tempi, quanto è oggetto di ignoranza dalla maggior parte della gente, che ragiona proprio come quei "coder" che snobbano - per non dire "saltano" - completamente le fonti di informazione su tale argomento perché "tanto, le music routine basta chiamarle dall'interrupt...".

14.1 Le replay routines sofisticate (Autore: Fabio Ciucci)

A proposito di tali music routines, per ora abbiamo visto solo quella standard fornita con il pro-tracker, ma ce ne sono anche altre più sofisticate. Vedremo ora una delle migliori, il player6.1a,

che richiede anche un programma di conversione (il p61con, in questo disco), con cui dobbiamo trasformare un modulo normale in uno ottimizzato per la replay routine.

NOTA: Questo player è copyright dell'autore:

Jarno Paananen / Guru of Sahara Surfers.

Quindi, se userete la sua replay routine in un prodotto commerciale, ad esempio in un gioco, dovete ricevere la sua autorizzazione scritta e dargli qualcosa (in marchi finlandesi!) come percentuale. Già si arrabbierà con me che non ho incluso tutto l'archivio!!!

Questo player ha moltissime opzioni, vediamo intanto di farci la cosa più semplice: suonare un modulo come abbiamo fatto con la routine standard fornita assieme al programma protracker.

Le cose da fare sono queste:

1. Convertire il modulo nel formato P61, usando l'utility "P61CON". Questa utility richiede la `reqtools.library` e la `powerpacker.library` nella directory `libs`: per funzionare. Nelle preferences del programma, non toccate niente, lasciando settata solo l'opzione "tempo". Annotatevi al salvataggio l'USECODE, che andrà specificato nel listato all'equante `USE =`. Questo serve per risparmiare codice.
2. In questo modo, abbiamo ottenuto il modulo convertito **non compresso**. Nonostante ciò, spesso il modulo si accorcia per l'ottimizzazione che viene fatta automaticamente.
3. Ora basta fare come con le routines precedenti: chiamare `P61_Init` prima di suonare, poi `P61_Music` ad ogni fotogramma, e `P61_End` alla fine. L'unica richiesta in più è l'abilitazione dell'interrupt di livello 6.

Vediamo un esempio pratico in Lezione14-10a.s.

Noterete come sia più veloce di quella standard, ma anche che usa il timer A del CIAB e l'interrupt di livello 6 (\$78). Ci sono poi degli equates, di cui occorre conoscere il significato:

```
1 fade = 0 ;0 = Normal, NO master volume control possible
2 ;1 = Use master volume (P61_Master)
```

Questo va messo ad 1 se si vuol controllare il volume per fare un fade, agendo sulla label `P61_Master`. Vedremo dopo un esempio. Se non ci serve tale opzione, mettiamo 0, in modo da risparmiare codice, infatti questi equate non sono altro che degli assemblaggi condizionati che usano le direttive dell'assemblatore `ifeq, ifne, endc...`

```
1 jump = 0 ;0 = do NOT include position jump code (P61_SetPosition)
2 ;1 = Include
```

Anche questa opzione va lasciata a zero, se non si usa la routine di salto ad una specifica posizione del modulo. Vedremo dopo un esempio.

```
1 system = 0 ;0 = killer
2 ;1 = friendly
```

Questa opzione è bene lasciarla a 0, se si fa codice "malvagio" e si usa la `startup2.s`. State solo attenti quando caricate da dos!!! (dovete anche lasciare l'interrupt \$78 (level6) al suo posto, senza ripristinare quello di sistema, nel caso carichiate con questa replay routine attiva!).

```
1 CIA = 0 ;0 = CIA disabled
2 ;1 = CIA enabled
```

Questa opzione va tenuta a 0 per usare la routine in modo "standard". Se si mette ad 1, non occorrerà più chiamare `P61_Music` ad ogni fotogramma, perché la temporizzazione avverrà totalmente col CIAB. Vedremo un esempio.

```

1  exec = 1          ;0 = ExecBase destroyed
2  ;1 = ExecBase valid

```

Qua va lasciato 1, dato che in \$4.w lasciamo l'execbase valido... non siamo mica dei maniaci!!! E la startup che la facciamo a fare?

```

1  opt020 = 0        ;0 = MC680x0 code
2  ;1 = MC68020+ or better

```

Questo è chiaro: se il vostro gioco/demo è AGA only, potete mettere questo ad 1, altrimenti lasciatelo a zero. Attenti a non metterlo ad 1 quando non serve!!!! **Solo se il gioco/demo va solo su AGA (quindi 68020+).**

```

1  use = $2009559    ; Usecode (mettete il valore dato dal p61con al salvataggio
2  ; diverso per ogni modulo!)

```

Qua il commento spiega tutto... annotatevi sempre l'usecode su un foglietto (senza perdere il foglietto, naturalmente), e mettetelo qua. Serve per far assemblare solo le routines degli effetti usati nel modulo, risparmiando spazio. Mettere -1 significa assemblare tutto (puah!).

Il programma di conversione permette anche di comprimere il modulo, ma questo fa perdere un pò la qualità. Quindi vi sconsiglio di compattarli... A meno che non dobbiate fare una 40k intro e abbiate le spalle al muro in termini di spazio, è sempre bene usare il modulo "normalmente" convertito. Comunque, il programma permette di scegliere quali sample compattare e quali no... e di ascoltare con le nostre orecchie se si perde troppo di qualità!!!

Ecco cosa si deve fare per comprimere e risuonare un modulo compresso:

1. Convertire il modulo nel formato P61 compresso, con "P61CON"- Nelle preferences del programma, occorre attivare l'opzione "pack samples". Da notare che potete scegliere quali sample comprimere e quali no. Ecco cosa vedrete per ogni sample:

Original Suona il sample originale (Stop con tasto destro mouse)

Packed Suona il sample come verrebbe compresso. Se notate che si perde troppo in qualità, ripensateci...!

Pack Segna questo sample come "da comprimere"

Pack rest Compatta tutti gli altri sample da qua in avanti

Don't pack Non compattare questo sample

Don't pack rest Non compattare tutti gli altri sample (da qua in avanti)

Annotatevi al salvataggio l'USECODE, come sempre. In più, annotatevi anche il "sample buffer length"!!!!!!

2. In questo modo, abbiamo ottenuto il modulo convertito e **compresso**.
3. Ora ci sono 2 cose in più da fare: innanzitutto, il modulo ha i sample compressi, che vanno scompattati in un buffer. Per fare ciò, occorre fare 2 cose: il buffer, lungo quanto indicato dal programma come "sample buffer length", e mettere il suo indirizzo in a2 prima di chiamare P61_Init, che provvederà alla decompressione. Per il resto (play ed end) è uguale.

Vediamo in pratica il tutto:

```

1      movem.l d0-d7/a0-a6,-(SP)
2      lea     P61_data,a0      ; Indirizzo del modulo in a0
3      lea     $dff000,a6      ; Ricordiamoci il $dff000 in a6!
4      sub.l   a1,a1           ; I samples non sono a parte, mettiamo zero
5      *****
6      >>>> lea     samples,a2    ; modulo compattato! Buffer destinazione per
7      *****                ; i samples (in chip ram) da indicare!
8      bsr.w   P61_Init        ; Nota: impiega alcuni secondi per decompress!
9      movem.l (SP)+,d0-d7/a0-a6

```

Per il modulo e il buffer, ecco le modifiche:

1. Il modulo non occorre più che sia caricato in chip ram:

```

1      Section modulozzo,data    ; Non occorre sia in chip ram, perchè è
2                                ; compresso e sarà scompattato altrove!
3      P61_data:
4      incbin "P61.stardust"    ; Compresso, (opzione PACK SAMPLES)

```

2. Il buffer deve essere caricato in CHIP RAM, e lungo quanto specificato:

```

1      section smp,bss_c
2
3      samples:
4      ds.b    132112    ; lunghezza riportata dal p61con

```

Come noterete, oltre a perdere qualità nei samples, si usa anche più memoria, in quanto abbiamo in più il buffer, anche se il modulo è più corto. Vediamo un esempio in Lezione14-10b.s.

Ora che abbiamo visto le 2 implementazioni principali, possiamo vedere tutte le varianti. Innanzitutto l'opzione CIA, che si abilita con l'equante. Potete vederne 2 esempi in Lezione14-10c.s e Lezione14-10d.s. Per finire, vediamo l'uso di 2 optional:

Il fade audio: basta attivare l'equante "FADE", e agire sulla apposita label P61_Master, che va da 0 a 64. Esempio in Lezione14-10e.s.

La possibilità di saltare a posizioni arbitrarie del modulo: basta attivare l'equante JUMP, e chiamare la routine P61_SetPosition, con la posizione nel registro D0. Esempio in Lezione14-10f.s.

Ci sarebbero anche altri optional, che possiamo riassumere nelle preferences:

Two files:	Questa opzione salva separatamente i sample e il song in 2 file. Può servire se usiamo più moduli con gli stessi sample...
P61A sign:	mette il segno P61A all'inizio del modulo... può servire solo a rendere più facile ai malintenzionati ripparlo!!! Non settatelo mai!
No samples:	Serve quando si salvano molti moduli che hanno gli stessi sample: la prima volta si mette "two files", e si salvano i moduli e il primo song. Poi si mette questa opzione e si salvano tutti gli altri songs.
Tempo:	Per far usare l'opzione "Tempo" al player.
Icon:	Se si vuol salvare un icona assieme al modulo
Delta:	Compressione a 8-bit anziché a 4-bit (ho notato che non cambia quasi niente... bah!)
Sample packing:	Da settare per la compressione dei samples con l'algoritmo delta a 4-bit (PERDENDO DI QUALITÀ!!!).

Buon ascolto a tutti!

LEZIONE 15 - IL CHIPSET AGA

Questa è l'attesissima lezione sul nuovo chipset AGA, presente nell'A1200 e nell'A4000. Quando uscì l'Amiga4000, alla fine del 1992, un mio amico lo comprò subito, e in pratica andai a stare a casa sua, tanto che lo ho usato di più io che lui. In quei primi mesi disassemblai le copperlist del sistema operativo e pezzi interi di KickStart, perché la defunta Commodore non dava a nessuno documentazione sull'AGA. Strano, ma vero. Comunque a forza di prove cominciai a capire qualcosa, ma mancava anche l'iffconverter AGA, e dovetti "convertire" a mano le figure da IFF a RAW. L'unico programma che era in grado di visualizzare schermate AGA in quel tempo era il nuovo DeLuxePaint, per cui caricavo una figura a 256 colori, poi in multitasking mi caricavo l'asmone e cercavo in memoria la figura .raw e la copperlist per salvarle. In un secondo tempo ricaricavo il raw, lo puntavo nella copperlist e incrociavo le dita. Comunque non riuscii ad essere il primo a fare una demo AGA, la prima la fecero gli *ABYSS*, una piccola demo che però visualizzava i fatidici 256 colori. Niente di eccezionale, ma erano stati i primi. Più o meno però ero allo stesso punto degli *Abyss* nella scoperta dell'AGA, e non mi scoraggiai. Era ormai Febbraio 1993, ero quasi pronto per una intro con un logo in 640*256 a 256 colori che ondeggiava con la fluidità di 1/4 di pixel (usando il nuovo BPLCON1), quando uscì la prima **vera** demo AGA, ossia *PLANET GROOVE* dei *TEAM HOI*.

Chiamai subito la loro BBS in Olanda, lasciando un messaggio al coder, *Rhino*. Da quel giorno comincio un rapporto di (costosi) messaggi tra di noi, dove ci scambiavamo le ultime scoperte e la funzione degli ultimi bit sconosciuti. Poco prima era uscito *ZOOL AGA*, che in realtà non aveva proprio nulla di AGA, per cui l'unico codice decentemente AGHIZZATO era la demo di *Rhino*, il quale si programmò anche un Iffconverter AGA (il primo uscito), che usai con molto piacere.

Dato che non esisteva alcuna documentazione sull'hardware del 1200, e che di conseguenza non si vedevano demo né giochi AGA, misi insieme le informazioni che avevo scoperto assieme a *Rhino* in un AGADOC.TXT, ma quando ero quasi pronto per distrinuirlo nelle BBS uscì un piccolo testo, *hard1200.txt*, opera di *Yragael*, un coder francese. In questo testo c'erano alcune cose che non sapevo, ma mancavano molte cose che sapevo io. Chiamai un poco di BBS in Francia e riuscii a trovarlo, e seppi che stava programmando anche lui un iffconverter per l'AGA, che salvava anche gli sprites larghi 64 pixel. Quell'IffConverter storico è presente nel disco di utility del corso. Misi tutte le informazioni insieme, e feci un gran bel testo, farcito anche di informazioni

sul 68020. Questo testo circolò per le bbs, e anche per i party. In teoria ero pronto per fare una demo aga, e infatti ne feci una per lo SMAU dell'ottobre 1993 a Milano, ma in realtà si tratta di uno slideshow “molto tecnico” più che di una demo (lo programmai in 2 settimane, faccio sempre le cose all'ultimo minuto!). Comunque c'erano figure a 256 colori in hires interlacciato, fade AGA a 24 bit (come al cinema!), nonché una figura in HAM8 (credo sia la prima figura in HAM8 visualizzata in una demo!!!), e un effetto di fade “incrociato” a 24 bit che ha avuto molto successo. Oggigiorno escono moltissime demo AGA, e giochi come *SUPER STARDUST* o *BRIAN THE LION* sfruttano finalmente le nuove possibilità. Nonostante abbia programmato la prima demo AGA italiana, poi mi sono “fermato” e non ho più fatto niente, tanto che l'ultima demo che ho fatto era per A500.

Perché? Non lo so. Comunque col mio AGADOC.TXT e qualche consiglio ho contribuito alla programmazione della seconda demo AGA italiana, ossia *IT CAN'T BE DONE*, programmata da *EXECUTOR/RAM JAM*, che ha texture mapping vario. Mentre Executor ha messo nella sua demo un dovuto ringraziamento per l'aiuto che gli ho dato, ben poche tra le prime demo AGA straniere contengono saluti per me, ma credo che molti abbiano usato il mio prezioso (almeno in quei tempi) agadoc. Qualche tempo dopo la Commodore cominciò a mandare il manuale delle specifiche AGA alle software house, per cui qualcuno lo “rubò” e lo trascrisse (furono i *COMBAT 18*), di conseguenza il mio agadoc divenne meno “esclusivo”.

Questa era la storia della scoperta dell'aga, dove posso considerarmi tra i primi 10 pionieri, anche se mi chiedo tuttora se ne sia valsa la pena, dato che poi mi sono passati tutti avanti leggendo qualche mese dopo la documentazione bella e pronta. Vi propongo una traduzione in italiano del mio primo AGADOC, dato che lo scrissi in inglese.

Innanzitutto occorre precisare che per visualizzare immagini AGA non occorre usare istruzioni 68020, si potrebbe fare una demo aga con istruzioni tutte del 68000 base, dato che le differenze stanno nel COPPER. Questo significa che potete programmare anche col *TRASH'M'ONE*, che non supporta istruzioni 68020, ma ovviamente se le userete conviene passare al *TFA ASMONE 1.25* presente nel disco di Utility: tra l'altro ha l'help in linea dei registri aga, come nel *TRASH'M'ONE*, solo che anziché usare `<=C>` occorre usare `<=R>`, per esempio per vedere il registro `$dff106` (BPLCON3) basta digitare `<=R 106>`. Abbiamo già visto come “disabilitare” l'aga:

1	<code>move.w #0,\$1fc(a5)</code>	<code>; FMODE - disabilita fetch 64/32 bit.</code>
2	<code>move.w #\$c00,\$106(a5)</code>	<code>; BPLCON3 - disattiva palette 24 bit</code>
3	<code>move.w #\$11,\$10c(a5)</code>	<code>; BPLCON4 - palette normale.</code>

Ebbene, ora dovremo vedere come abilitare tutto! Cominciamo con un sommario delle nuove possibilità, giusto per invogliarvi a sapere come usarle: la palette ora anziché essere a 12 bit, ossia 4096 colori, è stata portata a 24 bit, ossia 16 milioni. Mentre prima per ogni componente RGB si poteva scegliere un numero da 0 a 15, ora si può scegliere un numero tra 0 e 255. Dunque: $16 \times 16 \times 16 = 4096$ colori possibili nel vecchio modo OCS e ECS, mentre $256 \times 256 \times 256 = 16777216$ colori tra cui scegliere nell'AGA.

Per esempio, prima si potevano fare 16 toni di grigio al massimo, ossia si immetteva nei registri colore `$0000`, `$0111`, `$0222`, `$0333` ecc, mentre ora si possono fare 256 toni di grigio. Anche i bitplanes disponibili sono aumentati, infatti ora possono essere anche 8, ossia 256 colori. (8 bit=256 possibilità). Esiste anche un modo HAM8 speciale, con 262144 colori “teorici” sullo schermo, ma alcune limitazioni (lievi “sbavature”), simili all'HAM6 normale. HAM8 sta per HAM con 8 bitplanes, mentre HAM6 è l'HAM normale a 6 bitplanes. Il nuovo Dual Playfield può avere fino a 4 bitplanes per playfield (16 colori un playfield e 16 l'altro), e il banco dei 16 colori nella palette di 256 è selezionabile indipendentemente per ogni playfield.

Come se non bastasse, anche gli sprites si sono “evoluti”. Ricordate il limite di larghezza di 16 pixel? Ebbene gli 8 sprites ora possono essere larghi anche 32 o 64 pixel ciascuno, e si può scegliere se devono essere in lowres o in hires, indipendentemente dalla risoluzione dello

schermo. Per esempio, si possono visualizzare 8 sprites in hires, larghi 64 pixel, su uno schermo in lowres a 256 colori. Sprite Attached sono disponibili sempre. Gli sprites pari e dispari possono usare il loro banco indipendente di 16 colori dalla palette dei 256 totali. Comunque uno sprite non attached ha sempre un massimo di 3 colori + sfondo e uno attached 15 colori + sfondo. Una novità è pure quella che gli sprites possono apparire anche nei bordi, ossia fuori dalla finestra DIWSTART-DIWSTOP, mentre normalmente non potevano. Per attivare questa possibilità basta settare il bit 1 del *\$dff106* (BPLCON3) Come se non bastasse, il posizionamento orizzontale è stato portato a 32ns, ossia anziché fare 320 “scatti” per percorrere lo schermo orizzontalmente, ora possono fare passi più piccoli, anche di un quarto di pixel, come se lo schermo fosse in 1280*256, e si facesse 1 pixel alla volta. Questo permette di far ondeggiare gli sprite come nessuna scheda SUPER VGA del pc msdos può fare.

La possibilità di uno scrolling fluidissimo a passi di 1/4 di pixel è stata implementata anche per i bitplanes, si tratta di bit “extra” nel *\$dff102*, nel buon vecchio BPLCON1. È possibile fare decine di livelli in parallasse con lo scrolling più incredibile della storia dei computer. Il nuovo *\$dff102*, oltre a permettere scroll a “scattini” di 1/4 di pixel alla volta, ora può scorrere fino ad un massimo di 64 pixel, anziché 16. Anche se ci interessa in modo minore, è possibile già dall'ECS gestire schermi a 31khz, ossia per monitor multisync. Col chipset AGA è possibile “deinterlacciare” gli schermi a 15Khz, bitplanes e sprite compresi, per i monitor SUPERVGA. Le demo e i giochi comunque di solito sono in PAL! Tutte queste novità comunque se non “azionate” non interferiscono sulla compatibilità con il vecchio chipset, come avrete verificato eseguendo i sorgenti OCS/ECS delle precedenti lezioni.

In particolare occorre azzerare il *\$dff1fc* (FMODE) e il bit 0 del BPLCON0. Questo bit lo abbiamo sempre tenuto azzerato nelle lezioni precedenti. Settandolo diventano operativi altri bit nel BPLCON3 (*\$dff106*), tra cui BRDRSPRT, quello degli sprite fuori dai “bordi”. Per rilevare le collisioni con i bitplanes 7 ed 8, che non sono supportati dal CLXCON, esiste il CLXCON2 (*\$dff10e*), che si resetta scrivendo nel vecchio CLXCON, permettendo una corretta segnalazione delle collisioni nei giochi OCS. Non si sa al momento attuale se eventuali Amiga che usciranno nel futuro supporteranno l'AGA o soltanto l'ECS, si è detto che forse supporteranno solo l'OCS/ECS in emulazione e avranno un sistema grafico diverso. Comunque per i problemi che ci sono stati per vendere la Commodore ecc. il ritardo ha portato ad allontanare l'uscita di questi nuovi modelli, per cui l'AGA durerà per molti anni, e questo probabilmente porterà a supportarlo nelle eventuali nuove macchine Amiga.

Comunque c'è anche il CD32 che supporta l'AGA. Se volete programmare giochi per CD32 considerate che ha 2 porte joystick che supportano 11 “bottoni”, per cui dovete adattare il codice a questo joy. Altre differenze del CD32 sono 1Kb di flash RAM, dove si può salvare l'HIGH SCORE o le password dei giochi, nonché il chip AKIKO, che dovrebbe essere in grado di convertire grafica da Chunky a Planar, ma pare non sia velocissimo. Convertire da Chunky (modo video come la VGA) a Bitplanes amiga serve per la grafica in texture mapping, vedi DOOM sul PC MSDOS. Più avanti forse ci faremo un nostro DOOM.

Come prima cosa occorre vedere se il computer ha l'AGA, abbiamo già visto la routine per il detect:

```

1  LEA      $DFF000,A5
2  MOVE.W  $7C(A5),D0      ; DeniseID (o LisaID AGA)
3  MOVEQ   #100,D7         ; Controlla 100 volte (per sicurezza, dato
4                          ; che il vecchio denise da valori casuali)
5  DENLOOP:
6  MOVE.W  $7C(A5),D1      ; Denise ID (o LisaID AGA)
7  CMP.B   d0,d1           ; Lo stesso valore?
8  BNE.S   NOTAGA          ; Non è lo stesso valore: Denise OCS!
9  DBRA    D7,DENLOOP
10 BTST.L  #2,d0            ; BIT 2 azzerato=AGA. è presente l'aga??
11 BNE.S   NOTAGA          ; no...
12 ST.B    AGA             ; si... settiamo il flag "AGA" allora.
```

13 NOTAGA: ; non AGA... o OCS/ECS o il futuro AAA...
 14 ...

15.1 La nuova palette a 24 bit

Ok, ora vediamo in pratica come visualizzare 128 o 256 colori, come fare delle sfumature col copper a “24 bit”, eccetera. Per prima cosa è importante capire come funziona la nuova palette, perché poi per il resto si tratta soltanto di settare qualche bit qua e là per aggiungere bitplanes o allargare gli sprites. Abbiamo detto che per ognuna delle 3 componenti ROSSO, VERDE e BLU si può dare un valore da 0 a 255 anziché da 0 a 15. Se prima per settare il giallo occorreva mettere $\$F$ di rosso, $\$F$ di verde e 0 di blu, ora occorre $\$FF$ di rosso, $\$ff$ di verde e $\$00$ di blu. Fin qua tutto chiaro. Se prima dovevamo mettere in $\$dff180$ il valore $\$0ff0$ per il giallo ($\$ORGB$), ora dove mettiamo $\$00FFFF00$? Nel registro, che è una word, non c’entra $\$00ffff00$, ossia $\$00RRGGBB$. I progettisti hanno trovato un modo per mantenere la compatibilità con l’OCS e per far entrare 256 colori a 24 bit nei vecchi 32 registri a 12 bit!! Vediamo intanto come hanno risolto il primo problema, ossia quello di far entrare 1 colore $\$RRGGBB$, ad esempio, nel $\$dff180$ ($\$dff180$). Facciamo questa considerazione: se avessimo il colore a 12 bit $\$f3020$, come sarebbe l’equivalente a 24 bit? Naturalmente $\$f03020$. Ora, si può notare che i colori a 4 bit normalmente usati nell’OCS/ECS sono i 4 bit alti, o in altri termini il nibble alto, dei colori ad 8 bit dell’AGA. Ed è proprio così! Se azzeriamo i registri dell’AGA e mettiamo nel $\$dff180$ o in un’altro registro colore un valore, cambiamo i 4 bit alti delle 3 componenti RGB, lasciando azzerati i 4 bit bassi, per cui il colore risultante è lo stesso di quello di un OCS. Avrete intuito che per settare un colore a 24 bit occorre mettere separatamente i bit alti ($\$RxGxBx$) nel $\$dff180$, poi “scambiare” qualcosa e mettere i bit bassi ($\$xRxGxB$) sempre nel $\$dff180$. Facciamo un esempio: abbiamo il colore a 24 bit $\$437efa$, ossia RED = $\$43$, GREEN = $\$7e$, BLU = $\$fa$. Ecco come facciamo in copperlist:

```
1 dc.w $180,$47f ; metto i nibble alti
2 "scambio"
3 dc.w $180,$3ea ; metto i nibble bassi
```

Per ora abbiamo messo “scambio”, vediamo in pratica cosa si fa per scambiare la funzione del $\$dff180$ da “ricettore di nibble bassi” a “ricettore di nibble alti” del colore a 24 bit. Per selezionare i bit alti, abbiamo messo il valore $\$c00$ nel $\$BPLCON3$ ($\$dff106$) infatti nell’emulazione ECS i registri colore valgono sempre come ricettori di bit alti del colore. In teoria si potrebbe mettere anche $\$000$ nel $\$dff106$, perché settare i bit 10 e 11 serve solo nel modo DUAL PLAYFIELD a resettare delle cose che vedremo dopo. Si intuisce dunque che quando “un certo bit” del $\$dff106$ è azzerato, i registri colore ricevono i bit alti, quando è settato invece ricevono i bit bassi. Può sembrarvi contorto spezzare i valori RGB in questo modo, ma dato che la palette per le figure la salva l’iffconverter già pronta, non c’è di che lamentarsi.

Inoltre si possono fare delle routines che fanno copperlist o che “spezzano” i colori in questo modo. Il bit del $\$dff106$ che si occupa di “scambiare” la funzione dei registri colore è il nono, detto LOCT. Siccome quando scriviamo nei nibble alti (MSB) quelli bassi (LSB) sono azzerati per la compatibilità con OCS/ECS, quando si vuol settare un colore a 24 bit occorre caricare prima i bit alti, poi quelli bassi.

Ecco uno schemino del colore $\%RRRRrrrrGGGGggggBBBBbbbb$ (binario), dove le lettere maiuscole sono i bit alti della tonalità, quelle minuscole i bassi.

BIT#	11,10, 9, 8	7, 6, 5, 4	3, 2, 1, 0
----	-----	-----	-----
LOCT=0	R7 R6 R5 R4	G7 G6 G5 G4	B7 B6 B5 B4
LOCT=1	r3 r2 r1 r0	g3 g2 g1 g0	b3 b2 b1 b0

R = RED G = GREEN B = BLUE

Si può dire che i registri colore AGA hanno due facce, e si gira la faccia settando o azzerando il bit 9 di `$dff106`. Il bit 9 settato produce il valore `$200` (`%0000001000000000`). Per cui si può sostituire “scambio” con `$106, $200`:

```

1      dc.w    $106,$000      ; Seleziono i nibble alti
2      dc.w    $180,$47f      ; metto i nibble alti
3      dc.w    $106,$200      ; Seleziono i nibble bassi
4      dc.w    $180,$3ea      ; metto i nibble bassi

```

Molti settano anche i bit 10 ed 11, che come abbiamo detto servono solo per il dual playfield, comunque non fanno male:

```

1      dc.w    $106,$c00      ; Seleziono i nibble alti
2      dc.w    $180,$47f      ; metto i nibble alti
3      dc.w    $106,$e00      ; Seleziono i nibble bassi
4      dc.w    $180,$3ea      ; metto i nibble bassi

```

Dunque `$c00` per selezionare i bit alti, poi `$e00` per selezionare i bit bassi. Naturalmente se si debbono settare 10 colori non mettiamo ogni volta il `BPLCON3` tra un colore e l'altro, ma semplicemente:

```

1      dc.w    $106,$c00      ; Seleziono i nibble alti
2
3      dc.w    $180,$47f      ; metto i nibble alti di tutti i colori
4      dc.w    $182,$123
5      dc.w    $184,$456
6      dc.w    $186,$789
7      dc.w    $188,$abc
8      dc.w    $18a,$def
9
10     dc.w    $106,$e00      ; Seleziono i nibble bassi
11
12     dc.w    $180,$3ea      ; metto i nibble bassi di tutti i colori
13     dc.w    $182,$111
14     dc.w    $184,$444
15     dc.w    $186,$888
16     dc.w    $188,$434
17     dc.w    $18a,$abc

```

Ora è il momento di verificare in pratica se tutto questo funziona, proviamo a fare delle “barrette” tipo la lezione3, ma AGA: vedetevi `Lezione15a.s`

Noterete che è mooolto lungo scriversi la copperlist in AGA. Quindi per certe sfumature o cose ripetitive si fa prima a farsi una routine. In particolare vedetevi `Lezione15b.s` per fare delle sfumature.

15.2 I nuovi modi a 128 e 256 colori

Vediamo ora, invece, come è possibile “caricare” 256 colori se i registri colore sono soltanto 32. Infatti sappiamo che ogni registro colore ha 2 facce, che vedono una i nibble bassi e l'altra i nibble alti, ma sappiamo fare al massimo una figura a 32 colori, anche se tali colori sono scelti da una palette di 24 bit. Ebbene, c'è un altro trucco, anche questo nel `$dff106`.

I registri colore dovrebbero essere 256, e ce ne sono 32, ossia un ottavo di quelli che ci servono. Se azzerando il `$dff106` si accede ai primi 32 colori, si intuisce che ci deve essere un bit che, se settato, fa accedere ai registri dei colori dal 33 al 64, scrivendo sempre nel `$dff180-$dff1be`. Infatti ci sono 8 banchi con 32 registri colore ciascuno, e si deve decidere (con i bit 13,14 e 15 del `$dff106`) a quale degli 8 banchi accedere scrivendo nei registri colore:

```

----- bit --- $dff106 (BPLCON3) -----

15      BANK2   | Con questi 3 bit si seleziona uno degli 8 banchi
14      BANK1   | di registri per accedere ai 256 colori AGA

```

13 BANKO |

La selezione del “banco” funziona in modo analogo alla selezione del bitplane nel BPLCON0 (\$dff100), infatti questi 3 bit sono letti “assieme” e, a seconda del numero che contengono, selezionano il banco corrispondente:

valore dei 3 bit - banco di colori corrispondente - valore del \dff106

000	COLORE 00 - COLORE 31	\$c00 / \$e00
001	COLORE 32 - COLORE 63	\$2c00 / \$2e00
010	COLORE 64 - COLORE 95	\$4c00 / \$4e00
011	COLORE 96 - COLORE 127	\$6c00 / \$6e00
100	COLORE 128 - COLORE 159	\$8c00 / \$8e00
101	COLORE 160 - COLORE 191	\$ac00 / \$ae00
110	COLORE 192 - COLORE 223	\$cc00 / \$ce00
111	COLORE 224 - COLORE 255	\$ec00 / \$ee00

Questa tabella esplica come riusare i vecchi registri colore da \$180 a \$1be per accedere ai 256 colori. A destra sono riportati i valori che devono assumere i bit 13, 14, 15 del \$dff106 (BPLCON3) per accedere ai vari banchi. Facciamo un esempio: Se si vuol cambiare il colore 33, occorre fare:

```

1      DC.W    $106,$2C00    ; SELEZIONA PALETTE 1 (32-63), NIBBLE ALTI
2      dc.w    $182,$47f    ; metto i nibble alti
3      DC.W    $106,$2E00    ; SELEZIONA PALETTE 1 (32-63), NIBBLE BASSI
4      dc.w    $182,$3ea    ; metto i nibble bassi

```

Infatti si deve scegliere il banco che va dal colore 32 al 63, e di conseguenza scrivere nel \$dff180 significherà scrivere nel colore 32; scrivere nel \$dff182 significherà scrivere nel colore 33, e così via, fino al \$dff1be, che normalmente sarebbe il colore 31, ma che in questo caso diventa il colore 63, ossia 31+32. Se avessimo scelto il banco che va dal colore 64 al 95 il \$dff182 sarebbe stato il colore 65, eccetera.

Ecco la lista dei valori per \$dff106 pronti da mettere in copperlist, può esservi utile per operazioni di “taglia e incolla” con <Amiga+b+c+i>:

```

1      DC.W    $106,$c00    ; SELEZIONA PALETTE 0 (0-31), NIBBLE ALTI
2      DC.W    $106,$e00    ; SELEZIONA PALETTE 0 (0-31), NIBBLE BASSI
3      DC.W    $106,$2C00    ; SELEZIONA PALETTE 1 (32-63), NIBBLE ALTI
4      DC.W    $106,$2E00    ; SELEZIONA PALETTE 1 (32-63), NIBBLE BASSI
5      DC.W    $106,$4C00    ; SELEZIONA PALETTE 2 (64-95), NIBBLE ALTI
6      DC.W    $106,$4E00    ; SELEZIONA PALETTE 2 (64-95), NIBBLE BASSI
7      DC.W    $106,$6C00    ; SELEZIONA PALETTE 3 (96-127), NIBBLE ALTI
8      DC.W    $106,$6E00    ; SELEZIONA PALETTE 3 (96-127), NIBBLE BASSI
9      DC.W    $106,$8C00    ; SELEZIONA PALETTE 4 (128-159), NIBBLE ALTI
10     DC.W    $106,$8E00    ; SELEZIONA PALETTE 4 (128-159), NIBBLE BASSI
11     DC.W    $106,$AC00    ; SELEZIONA PALETTE 5 (160-191), NIBBLE ALTI
12     DC.W    $106,$AE00    ; SELEZIONA PALETTE 5 (160-191), NIBBLE BASSI
13     DC.W    $106,$CC00    ; SELEZIONA PALETTE 6 (192-223), NIBBLE ALTI
14     DC.W    $106,$CE00    ; SELEZIONA PALETTE 6 (192-223), NIBBLE BASSI
15     DC.W    $106,$EC00    ; SELEZIONA PALETTE 7 (224-255), NIBBLE ALTI
16     DC.W    $106,$EE00    ; SELEZIONA PALETTE 7 (224-255), NIBBLE BASSI

```

Tutto sembrerebbe perfetto. Ma manca un particolare! Come si fa a scegliere 8 bitplanes nel BPLCON0? C'è il posto solo per 7 bitplanes. Infatti, sono disponibili i bit 12,13 e 14, che possono andare da %000 per zero bitplanes a %111 per 7 bitplanes, ossia 128 colori. Occorrerebbe poter avere un bit alto in più, in modo da ottenere %1000, ossia 8. Nessun problema, tale bit è stato assegnato come il quarto bit di \$dff100. Per settare 8 bitplanes, dunque, occorre azzerare i bit 12,13,14 e settare il quarto, e il gioco è fatto. Ad esempio:

```

1      ;5432109876543210
2      dc.w    $100,%00000001000010001 ; 8 bitplanes lowres (320*256)
3      dc.w    $100,%10000001000010001 ; 8 bitplanes hires (640*256)
4      dc.w    $100,%01110010000000001 ; 7 bitplanes lowres (320*256)

```

Da notare che lasciamo sempre il bit 9 settato, per il genlock, e settiamo il bit 0, ECSENA, che abilita dei bit speciali che vedremo in seguito. Da notare che si possono avere anche 6 bitplanes non extra half bright, cioè 64 colori di cui si può scegliere la palette normalmente, basta scegliere 6 bitplanes e settare il bit 9 (KilleHB) del BPLCON2 (\$dff104). Se questo bit non è settato viene emulato il vecchio EHB, con 32 colori + 32 “scuriti”. Ora, per verificare quanto abbiamo detto, apprestiamoci a visualizzare una figura a 256 colori, in Lezione15c.s

La pic è mia. Ammetto di essermi ispirato allo stile di giochi come *AGONY* e *SHADOW OF THE BEAST*, niente di artisticamente innovativo, ma mi pare che regga, no? Comunque serve bene allo scopo del listato. Avrete notato che prima della copperlist e della figura ci sono dei:

```
CNOP    0,8    ; allineo a 64 bit
```

In realtà con FMODE (\$dff1fc) azzerato non “serve”, vedrete dopo perché. Come abbiamo visto per le figure non aga, si può “attaccare” la palette in fondo alla figura, da mettere in copperlist con una routine. Tale routine è un pò più complessa, ma non poi troppo: Lezione15c2.s

Ora che abbiamo quella routine, vi sarà più facile capire come ottenere un fade a 24 bit, modificando la routine di fade vista nella lezione 8. Vedetela in Lezione15c3.s. Ora vediamo di “ottimizzarla”, in Lezione15c4.s. Infine la facciamo in “realtime” al 100%: Lezione15c5.s.

Ora potete provare a convertire la vostra figura in 320*256 a 128 o 256 colori, come preferite, usate il PicCon, l'IffConv o l'AgaConv nel disco di utility. Vi consiglio vivamente di leggere le istruzioni del PicCon presenti nel disco.

15.3 FMode

Siete riusciti a visualizzare la vostra figura AGA? Ebbene, se provaste a visualizzare una figura in hires 640*256, nonostante includiate il RAW e la PALETTE giusta e settiate il bit 15 del BPLCON0, non otterreste altro che uno schermo nero... Questo perché abbiamo lasciato \$dff1fc (FMODE) azzerato. Tale registro controlla il BURST, ossia il modo con cui vengono trasferiti i dati dalla memoria al “video”.

Normalmente il trasferimento è a 16 bit, ma per visualizzare la grafica più “spinta” occorre settare il trasferimento a 32 bit o a 64 bit. Se il trasferimento avviene a 16 bit, ciò che deve essere trasferito deve essere ad un indirizzo pari, ossia allineato a WORD (16 bit). Infatti non si deve puntare un bitplane ad un indirizzo dispari! Ora, se il burst è a blocchi di 32 bit, i dati devono essere ad un indirizzo allineato a 32 bit, ossia a longword! Per esempio un indirizzo come \$16dfc è un multiplo di 4 (4*23423) e come tale multiplo di 4 bytes da 8 bit ossia di 4*8=32 bit. Insomma è un indirizzo allineato a 32 bit.

Per allineare dei dati ad indirizzi a 32 bit esiste la direttiva CNOP 0,4. Mentre EVEN, ossia CNOP 0,2, allinea a 2 byte, ossia 16 bit, cnoppare 0,4 allinea a 4 bytes, ossia 32 bit.

Se il burst è a 64 bit occorre mettere dei CNOP 0,8 prima delle copperlist, degli sprites e dei bitplanes, per assicurarci l'allineamento a 64 bit. Se l'assemblatore facesse delle storie e non allineasse, la figura apparirebbe come “affettata”, ossia a strisce verticali, dato che i blocchi a 32 o 64 bit non corrispondono all'inizio della figura. Per verificare se una label è allineata a 64 bit, assemblate, poi verificate a che indirizzo si trova tale label col comando “M”, poi dividete l'indirizzo per 8, e moltiplicate di nuovo il risultato per 8. Se torna l'indirizzo originario, significa che è un multiplo di 8, e tutto è OK, se viene diverso significa che c'è un resto e non è un multiplo di 8. Allora mettete dei dc.w 0 sopra l'indirizzo e provate ad allinearlo “a mano”.

Naturalmente, sarebbe bene tenere sempre attivato il burst (bandwidth) al massimo, ossia a 64 bit. Questo si può fare mettendo il valore 3 nel *\$dff1fc*. Comunque dovete stare attenti al fatto che se volete allargare i bitplanes, devono essere allargati a “blocchi” di 8 bytes alla volta. Per esempio abbiamo visto come convenga in certi casi avere “a lato” un pezzo di bitplane fuori dalla finestra video, per esempio per gli scroll e per i textscroll col blitter. In questo caso non potremmo aggiungere 2 bytes solamente, ma 8. Altro fatto è che non si deve usare MAI l’Allocmem per trovare spazio in memoria per i bitplanes, perché da degli indirizzi allineati a 16 bit, che solo per caso possono essere allineati anche a 64 bit. Già nei primi sorgenti della lezione, anche se non era necessario, è stata seguita la regola dell’allineamento:

```

1      CNOP 0,8      ; allineo ad indirizzo a 64 bit
2  sprite:
3      incbin "agasprite1"
4
5      CNOP 0,8      ; allineo ad indirizzo a 64 bit
6  pic:
7      incbin "AGABitplanes"
```

Vediamo un pò meglio i primi due bit del registro FMODE (*\$dff1fc*):

```

bit 1  BPAGEM | Bitplane Modo pagina (doppio CAS)
bit 0  BLP32  | Bitplane largo 32 bit
```

Abbiamo detto che, se i bit sono entrambi azzerati, il burst è “emulazione OCS/ECS”, ossia il trasferimento è a 16 bit. E se sono tutti e due settati il modo è 64bit. Vediamo i 4 casi in cui si possono trovare i primi 2 bit:

```

[x1]    %00    - Trasferimento bitplane data di 2 bytes alla volta (16bit)
              Cicli di memoria: CAS normale
              Larghezza del bus 16 bit
              Richiesto: Bitplanes allineati a 16 bit

[x2]    %01    - Trasferimento bitplane data di 4 bytes alla volta (32bit)
              Cicli di memoria: CAS normale
              Larghezza del bus 32 bit
              Richiesto: Bitplanes allineati a 32 bit (Double)
                      Modulo = Modulo -4

[x2]    %10    - Trasferimento bitplane data di 4 bytes alla volta (32bit)
              Cicli di memoria: CAS DOPPIO
              Larghezza del bus 16 bit
              Richiesto: Bitplanes allineati a 32 bit (Double)
                      Modulo = Modulo -4

[x4]    %11    - Trasferimento bitplane data di 8 bytes alla volta (64bit)
              Cicli di memoria: CAS DOPPIO
              Larghezza del bus 32 bit
              Richiesto: Bitplanes allineati a 64 bit (Quadruple)
                      Modulo = Modulo -8
```

Direi che va benissimo usare sempre *%11*, ossia *\$3*. l’unico problema che può presentarsi è un aggrovigliamento dei DMA se eventualmente il blitter e il processore (non dotato di FAST RAM) dovessero inciampare nel fiume a 64bit del trasferimento ipergalattico dei chip AGA. In caso di queste turbolenze, potreste optare per un *%01* o *%10*, se vedete miglioramenti. Vediamo ora la bandwidth minima necessaria per le varie risoluzioni grafiche AGA (anche se tenteremo sempre di metterla a 64bit!).

Come già visto, per il 320*256 lowres a 8 bitplanes bastano 16bit (*\$1fc,0*):

LORES (320x256),	Per 64, 128, 256 colori o HAM8, bastano 16bit
HIRES (640x245),	Per 32, 64, 128, 256 colori o HAM8, occorrono 32bit
SUPERHIRES (1280x200)	Per 2, 4 colori bastano 16 bit Per 8, 16 colori occorrono 32 bit Per 32, 64, 128, 256, HAM8 occorrono 64 bit

Intanto, potremmo cominciare mettendo il BURST al massimo nella visualizzazione della nostra tranquilla figura in lowres. Anche se visibilmente non accadrà nulla, il trasferimento sarà più GALATTICO. Occorre però una precisazione IMPORTANTISSIMA: Cambiare il FETCH comporta anche una correzione del MODULO, per circostanze hardware. Dunque, se l'FMODE è azzerato, e il trasferimento avviene a 16 bit, il modulo deve essere 0, o comunque è normale. Se invece il BURST è a 32 bit, il modulo è uguale al modulo -4, per cui se era zero, occorre mettere -4 in BPL1MOD/BPL2MOD per compensare. Se il BURST è a 64 bit, il modulo è uguale al modulo normale -8:

```
BANDWIDTH 1: dc.w $1FC,0      ; Allora i bitplanes devono essere allineati
                                ; almeno a word (16 bit), e il modulo è
                                ; quello normale.

BANDWIDTH 2: dc.w $1FC,1 o 2  ; Allora i bitplanes devono essere allineati
                                ; almeno a long (32 bit), e il modulo è
                                ; uguale al modulo normale meno 4

BANDWIDTH 4: dc.w $1FC,3      ; Allora i bitplanes devono essere allineati
                                ; almeno a quadword (64 bit), e il modulo è
                                ; uguale al modulo normale meno 8
```

Per verificare il tutto, ricaricate Lezione15c.s, e provate a modificare l'FMODE in copperlist mettendoci il valore 1 o 2, attivando il burst a 32 bit. Noterete che se non fate altre modifiche la figura appare con il modulo sbagliato. Modificate allora anche il modulo, mettendolo = -4, e vedrete che la pic si "riaddrizza". Allo stesso modo, provate a mettere il burst a 64 bit, mettendo il valore \$3 al dc.w \$1fc (FMODE) in copperlist. Ora dovreste mettere il modulo, sia \$108 che \$10a, a -8 per vedere la PIC.

Chiarito questo fatto, tenete sempre l'FMODE a \$3, ossia settate sempre i primi 2 bit, e potrete visualizzare anche hires a 256 colori.

C'è un'ultimo particolare riguardo agli effetti del burst a 32 o 64 bit. Anche i valori del DDFSTRT e DDFSTOP sono modificati. Con un burst normale a 16 bit per aprire uno schermo in hires che partisse alla posizione orizzontale MIOX, si determinava con la "formula":

$$DDFSTRT=(MIOX-9)/2$$

Con il burst a 32 bit, invece, occorre fare:

$$DDFSTRT=(MIOX-17)/2$$

Perchè viene letta una longword anzichè una word. Comunque se usate schermi a larghezza standard non ci sono problemi, e se ci fossero potete andare a tentativi! Comunque, in pratica, con il burst attivo, se visualizzate una figura in hires non occorre settare il DDFSTART e il DDFSTOP a \$003c e \$00d4, ma allo stesso modo del lowres:

```
1      dc.w    $92,$0038      ; DdfStart lowres, adatto per HIRES con burst
2      dc.w    $94,$00d0      ; DdfStop lowres, come sopra
```

Questo a causa dei cicli di memoria richiesti per un trasferimento “turbo” dalla ChipRam al Chip Lisa.

Vediamo di visualizzare una figura in hires a 256 colori in Lezione15d.s La figura in questione è opera di Cristiano Evangelisti, handle *KREEX*, un “indipendente” che sta facendo la grafica ad un gioco adventure che sta programmando un mio allievo.

15.4 HAM8

Il buon vecchio HAM a 6 bitplanes è stato “sbancato” dal nuovo HAM8, a 8 bitplanes. 6 bitplanes sono usati per i colori e 2 per i bit di controllo. Inoltre è disponibile in tutte le risoluzioni, non solo in LowRes. Per attivarlo, basta settare 8 bitplanes e il bit dell’HAM nel BPLCON0 (\$100). Degli 8 bit, i 6 bit alti sono usati come 64 registri colore base a 24bit, o come un valore di MODIFY a 6 bit, più i 2 bit bassi per il modo hold o modify a 18bit. Questo permette di visualizzare più di 256000 colori. I 2 planes di controllo e i 6 piani colore sono “internamente” fusi negli 8 bit dell’HAM8, rovesciando però l’ordine: prima i piani 3, 4, 5, 6, 7, 8 poi 1 e 2. Questo causa degli scambi di bitplanes che vedremo.

Ecco un confronto tra il vecchio HAM6 e il nuovo HAM8.

Funzione dei bitplanes di controllo 5 e 6 nell’HAM6:

BP6	BP5	RED	GREEN	BLUE
0	0	selez.il nuovo reg.base (1 dei 16)		
0	1	hold	hold	modify
1	0	modify	hold	hold
1	1	hold	modify	hold

Nell’HAM8 i bitplane di controllo sono l’1 e il 2:

BP2	BP1	RED	GREEN	BLUE
0	0	selez.il nuovo reg.base (1 dei 64)		
0	1	hold	hold	modify
1	0	modify	hold	hold
1	1	hold	modify	hold

Questi 2 bit BASSI sono il comando: nuovo registro di base o altera una delle componenti RED, GREEN, BLU. Attenzione al fatto che i 2 bit bassi del colore non possono essere modificati, per cui la palette iniziale deve essere scelta bene. (Comunque questo spetta ai grafici e ai programmi come AdPro o ImageFX).

Ora, vediamo in pratica come visualizzare una figura HAM8. Innanzitutto la palette è di 64 colori, non di 256: bastano infatti solo quei “pochi” colori per generare l’ham, grazie ai bit di controllo che “holdano” o “modificano” le componenti RGB. Per attivarlo basta settare in BPLCON0 8 bitplanes e il modo HAM, ossia settare il bit 4 e il bit 11. C’è però un’ultima “particolarità”. Abbiamo già detto che i bitplanes 1-2 sono internamente “scambiati” con i bitplanes 3-4-5-6-7-8.

Ebbene, in effetti al momento di “puntare” i bitplanes c’è questo problema. Se salvate il RAW con il PicCon potete puntare regolarmente la figura, come avete fatto per una figura a 256 colori. Questo perchè il PicCon scambia già l’ordine nel RAW, in modo che poi “torni”. Se invece salvate il raw con l’AgaConv o con altri iffconverters, il raw sarà salvato “come è”, per cui dovrete far puntare i primi 6 bitplanes come fossero i bitplanes 3,4,5,6,7,8, e infine puntare i plane 1 e 2.

```

1 ; Questo è l'ordine dei bitplanes se salvate il RAW con AgaConv o con un
2 ; altro iffconverter che non "rovescia" i plane da solo.
3
4 BPLPOINTERS:
5     dc.w $e8,0,$ea,0      ; terzo      bitplane
6     dc.w $ec,0,$ee,0      ; quarto     "
7     dc.w $f0,0,$f2,0      ; quinto     "
8     dc.w $f4,0,$f6,0      ; sesto      "
9     dc.w $f8,0,$fa,0      ; settimo    "
10    dc.w $fc,0,$fe,0      ; ottavo     "
11    dc.w $e0,0,$e2,0      ; primo      "
12    dc.w $e4,0,$e6,0      ; secondo     "

```

Nel listato di esempio il RAW è salvato con PicCon, per cui i planes sono puntati in modo normale. Caricate `Lezione15e.s`

Ora possiamo fare un confronto tra l’HAM8 e i normali 256 colori. Vedete e giudicate in `Lezione15e2.s`

Da notare che cambiare l’intera palette AGA è un’operazione che richiede una decina di linee raster! In questo esempio cambiamo “solo” 64 colori, per cui bastano 2 o 3 linee, ma se volessimo fare, ad esempio, un gioco adventure con una pic a 256 colori nella parte alta dello schermo, e un pannello di controllo nella parte bassa, al momento del cambio della palette dovremmo lasciare 10 pixel “neri”, in attesa che la palette cambi del tutto, comunque occorre considerare il tempo di “repalettamento”. Vi ricordate che ogni MOVE del copper impiega 8 pixel lowres, e circa 52 move occupano una linea?...

15.5 Sprite

Le novità per quanto riguarda gli sprites sono molteplici. Come prima cosa è possibile deciderne la larghezza, scegliendo tra 16, 32 o 64 pixel. Come sapete normalmente la larghezza massima era di 16 pixel! Inoltre lo sprite può essere visualizzato in lowres, hires o superhires, indipendentemente dalla risoluzione della pic sullo sfondo. Vediamo come si fanno, in pratica, queste cose.

La risoluzione dello sprite si decide con i bit 6 e 7 del registro BPLCON3 (`$dff106`), ed è indifferente la larghezza degli sprite:

bit 7	bit 6	
0	0	LOW RES, emulazione OCS/ECS (140ns)
0	1	LOW RES, (140ns) (Non è il modo ECS standard!)
1	0	HIRES (70ns)
1	1	SUPER HIRES (35ns)

Questi due bit si chiamano SPRES0 e SPRES1, tanto per cambiare. Vediamo subito un esempio di sprite in hires, dato che basta settare il bit 7 del `$dff106`, in `Lezione15f.s`

Sprites larghi 32 o 64 pixel

Ora occorre vedere come è possibile fare sprites larghi 32 o 64 pixel. Innanzitutto è necessario avere un iffconverter che salvi sprites di questo tipo! Il PicCon o AgaConv li salvano adeguatamente, non ci sono problemi. Come al solito ci sono un paio di bit che decretano la larghezza.

Questi sono i bit 3 e 2 del registro FMODE (\$dff1fc), detti SPAGEM e SPR32. I bit SPAGEM e SPR32 decidono la larghezza dello sprite, e di conseguenza se i dati da passare a SPRxDATA devono essere a 16, 32 o 64 bit, in modo analogo a come viene fatto per i bitplanes. È analogo anche il fatto che gli sprite a 32 bit devono essere allineati con un cnop 0,4, e quelli a 64 bit con un cnop 0,8. Questo per il risaputo fatto che i trasferimenti a 16 bit richiedono una bandwidth *1, mentre quelli a 32 bit ne richiedono una *2, di conseguenza quelli a 64 ne richiedono una *4. Nel caso degli sprite però variano le word di controllo, che si “allargano” assieme al resto dello sprite, nei casi sia a 32 o 64 bit.

Ma vediamo una tabella dei valori dei bit SPAGEM e SPR32 dell’FMODE:

bit 3	bit 2	Larghezza	Word di controllo
0	0	16 pixel	2 word (normale) - richiede cnop 0,2
1	0	32 pixel	2 longword - richiede cnop 0,4
0	1	32 pixel	2 longword - richiede cnop 0,4
1	1	64 pixel	4 longword - richiede cnop 0,8

Gli sprite “allargati” non sono disponibili nel caso in cui il DMA non ce la faccia, specialmente in superhires overscan interlacciato a 256 colori.

Dunque, avendo salvato uno sprite largo 32 o 64 pixel con l’iffconverter, e avendolo allineato ad un indirizzo multiplo di 4 o di 8, possiamo accedere alle sue word di controllo nello stesso modo di uno sprite largo 16 pixel? NO di certo, ecco perché:

Questa è la struttura di uno sprite normale, largo 16 pixel:

```

1 MIOSPRITE16:
2 VSTART:
3     dc.b $50          ; Posizione verticale di inizio sprite (da $2c a $f2)
4 HSTART:
5     dc.b $90          ; Posizione orizzontale di inizio sprite (da $40 a $d8)
6 VSTOP:
7     dc.b $5d          ; $50+13=$5d    ; posizione verticale di fine sprite
8 VHBITS:
9     dc.b $00          ; bit
10
11     dc.w %0000000000000000,%0000110000110000 ; dati
12     dc.w %0000000000000000,%0000111001110000
13     ...
14     dc.w 0,0          ; 2 word azzerate definiscono la fine dello sprite.
```

Ossia:

```

1
2     dc.w 0,0          ; 2 word di controllo
3     dc.w dataPlane1,dataPlane2 ; 2 word (16 bit - 16 pixel) con i 2 "plane"
4     dc.w dataPlane1,dataPlane2 ; 2 word (16 bit - 16 pixel) con i 2 "plane"
5     dc.w dataPlane1,dataPlane2 ; 2 word (16 bit - 16 pixel) con i 2 "plane"
6     ....
7     dc.w 0,0          ; 2 word azzerate per terminare
8
```

Ora, la struttura degli sprite larghi 32 pixel è questa:

```

1
2     dc.l 0,0          ; 2 longword di controllo
3     dc.l dataPlane1,dataPlane2 ; 2 longword (32 bit/pixel) con i 2 "plane"
4     dc.l dataPlane1,dataPlane2 ; 2 longword (32 bit/pixel) con i 2 "plane"
5     dc.l dataPlane1,dataPlane2 ; 2 longword (32 bit/pixel) con i 2 "plane"
6     ....
7     dc.l 0,0          ; 2 longword azzerate per terminare
8
```

Mentre quella degli sprite larghi 64 pixel è questa:

```

1
2 dc.l 0,0,0,0 ; 2 doppie longword di controllo
3 dc.l data1a,data1b,data2a,data2b ; 2 doppie longword (64 bit/pixel)
4 dc.l data1a,data1b,data2a,data2b ; 2 doppie longword (64 bit/pixel)
5 dc.l data1a,data1b,data2a,data2b ; 2 doppie longword (64 bit/pixel)
6 ....
7 dc.l 0,0,0,0 ; 2 doppie longword = 0 per terminare
8

```

Ora, quello che ci interessa è trovare i nostri bit nelle nuove word di controllo estese a longword e a doppia longword. Per quanto riguarda gli sprite larghi 32 bit:

```

1
2 SPRITE32:
3 VSTART:
4 dc.b $50 ; Posizione verticale di inizio sprite (da $2c a $f2)
5 HSTART:
6 dc.b $90 ; Posizione orizzontale di inizio sprite (da $40 a $d8)
7 DC.W 0 ; Word "aggiunta" nello sprite largo 32 pixel
8 VSTOP:
9 dc.b $5d ; $50+13=$5d ; posizione verticale di fine sprite
10 VHBITS:
11 dc.b $00 ; bit
12 DC.W 0 ; Word "aggiunta" nello sprite largo 32 pixel
13
14 dc.l %00000000000000011110000000000000,%000000000000100000000000000000 ; dati
15 dc.l %00000000000000011111100000000000,%00000000000010111100000000000000
16 ...
17 dc.l 0,0 ; Fine dello sprite (2 longword anzichè 2 word).
18

```

Come si vede, le 2 word di controllo sono diventate 2 long, e i bit di controllo sono rimasti quelli della word alta.

Vediamo ora il caso dei pixel larghi 64 pixel:

```

1
2 SPRITE64:
3 VSTART:
4 dc.b $50 ; Posizione verticale di inizio sprite (da $2c a $f2)
5 HSTART:
6 dc.b $90 ; Posizione orizzontale di inizio sprite (da $40 a $d8)
7 dc.w 0 ; Word + longword aggiunte per raggiungere la doppia
8 dc.l 0 ; longword nello sprite largo 64 pixel (2 long!)
9 VSTOP:
10 dc.b $5d ; $50+13=$5d ; posizione verticale di fine sprite
11 VHBITS:
12 dc.b $00 ; bit
13 dc.w 0 ; Word + longword aggiunte per raggiungere la doppia
14 dc.l 0 ; longword nello sprite largo 64 pixel (2 long!)
15
16 dc.l data1a,data1b,data2a,data2b ; 2 doppie longword (64 bit/pixel)
17 dc.l data1a,data1b,data2a,data2b ; 2 doppie longword (64 bit/pixel)
18 dc.l data1a,data1b,data2a,data2b ; 2 doppie longword (64 bit/pixel)
19 ...
20 dc.l 0,0,0,0 ; Fine dello sprite (2 doppie longword!).
21

```

Da qui ne deriva che occorre fare delle piccole modifiche a UniMuoviSprite, in modo che acceda ai byte spostati della seconda word di controllo.

Un esempio di sprite largo 32 pixel è Lezione15f2.s

Un esempio di sprite largo 64 pixel è Lezione15f3.s

Avete visto che insettone? E ne potete fare 8 così, oppure 4 a 16 colori in modo attached.

Da notare che il PicCon salva gli sprite attached senza il bit 7 (attach) settato allo sprite dispari, per cui lo dovete settare “a mano” se lo salvate con questo IffConverter.

15.6 Nuovo posizionamento orizzontale ad 1/4 di pixel

Un quarto di pixel? Ebbene sì! Sono stati aggiunti 2 bit “bassi” alla posizione orizzontale dello sprite, rendendola possibile a “scattini” 4 volte più piccoli, dunque più fluidi. Vediamo dove sono stati messi questi bit, analizzando SPRxCTL, la serie di registri che è un “equivalente” della seconda word di controllo dello sprite:

\$dff142/14A/152/15A/162/16A/172/17A - SPRxCTL - Controllo e posiz. sprite

BIT	nome	FUNZIONE
15-08	EV7-0	VSTOP - Gli 8 bit bassi della posiz. vert. di fine)
07	ATT	Bit di controllo dell'attached (solo sprite dispari)
06	SV9	Decimo bit della posizione di inizio verticale
05	EV9	Decimo bit della posizione di fine verticale
04	SH1=0	Posiz. orizzontale, incremento 70nS (mezzo pixel)
03	SH0=0	Posiz. orizzontale, incremento 35nS (1/4 di pixel)
02	SV8	Nono bit della posizione verticale di inizio (vstart)
01	EV8	Nono bit della posizione verticale di fine (vstop)
00	SH2	Posiz. orizzontale, incremento 140nS (1 pixel)

I bit che ci interessano sono SH0, SH1, SH2, ossia Start Horizontal. Come vedete, oltre ai noti bit SV8, EV8, ossia gli ottavi bit di VSTART e VSTOP, troviamo anche due nuovi bit che riguardano l'HSTART: oltre al bit “basso” che ci permette lo scroll di un pixel per volta, ne sono stati aggiunti un paio di ancora più “bassi”, che ci permettono di fare scroll di mezzo pixel o di 1/4 di pixel alla volta. Per 140ns (nanosecondi) si intende il “tempo” video di scorrimento. è comunque più chiaro dire che 140ns corrispondono ad 1 pixel lowres, mentre $140/2 = 70$ ns corrispondono ad un pixel hires, (o mezzo pixel lowres), infine appare ovvio che $70/2 = 35$ ns equivalgono ad 1/4 di pixel lowres, oppure ad un pixel in risoluzione 1280*xxx, ossia superhires.

Ma come si fa allora a far spostare a scattini di 1/4 di pixel lo sprite? Una via è quella di modificare la routine UniMuoviSprite, in modo che in entrata accetti una posizione X da 0 a 1280, anziché da 0 a 320, in questo modo se aggiungiamo 1 ogni volta lo scroll sarà ad 1/4 di pixel, se aggiungiamo 2 sarà a mezzo pixel oppure aggiungendo 4 per volta avremo uno scroll di un pixel alla volta. Semplice, no?

Vedetevi l'implementazione in Lezione15f4.s e Lezione15f5.s

In pratica la posizione orizzontale AGA è un numero a 11 bits, anziché 9.

15.7 Il bit BRDRSPRT

Il bit BRDRSPRT, quando è settato, rende possibile la visualizzazione degli sprite anche fuori dai bordi definiti da DIWSTART/DIWSTOP. Da notare che con questo bit abilitato gli sprite sono visualizzati anche se i bitplanes sono disabilitati in BPLCON0! Occorre però ricordarsi di settare anche il bit 0 del bplcon0 (\$dff100), che abilita anche altri bit speciali. Il bit in questione è il secondo (01) del \$dff106 (BPLCON3).

Vediamo una sua implementazione in Lezione15f6.s

15.8 Il modo attached

Gli sprites possono essere attached in qualsiasi modo, escluso il modo ECS SHRES (1280*xxx, 35ns).

15.9 La palette degli sprite AGA

Si può usare come palette per gli sprite un qualsiasi banco di 16 colori preso dalla palette di 256. I bit da ESPRM7 a ESPRM4 permettono di “rilocare” la colormap degli sprite pari, mentre i bit da OSPRM7 a OSPRM4 permettono di rilocare la colormap degli sprite dispari. Negli OCS/ECS i 16 colori degli sprite erano sempre e obbligatoriamente dal color16 (\$dff1a0) al color31 (\$dff1be), per cui una figura che aveva più di 16 colori doveva per forza condividere i colori dal 16 al 31 con gli sprite. Con l'AGA invece è possibile spostare questo banco di 16 colori in un qualsiasi segmento dei 256. Se per esempio avessimo una figura a 128 colori, potremmo spostare i colori degli sprite alla posizione dal color129 in avanti, in modo da non dover condividere la palette con la figura. L'utilità quindi si riscontra con le figure con più di 16 colori. Comunque se i bitplanes sono 8 e i colori 256, possiamo scegliere quale banco di 16 usare, ma tali 16 colori saranno sempre in comune con la figura. Ecco come sono assegnati nell'OCS i colori della palette agli sprite:

Sprites	Colors	
0-1	00-03	; \$dff1a0/1a2/1a4/1a6
2-3	04-07	
4-5	08-11	
6-7	12-15	

Quindi ci sono 4 coppie di sprite a 3 colori. Per esempio, per definire i 3 colori del primo sprite:

```

1      dc.w    $1A2,$462      ; color17, nibble bassi
2      dc.w    $1A4,$2e4      ; color18, nibble bassi
3      dc.w    $1A6,$672      ; color19, nibble bassi

```

Nel chipset AGA, invece, oltre a poter scegliere quale parte della palette a 256 colori usare per gli sprite, è possibile selezionare 2 palette, una per gli sprite pari e una per gli sprite dispari, avendo un totale di 32-8 colori, ossia 24, dato che il color0 è il trasparente e non conta. Ricapitolando, mentre in OCS avevamo 8 sprite a 3 colori effettivi ciascuno, ma legati da un rapporto di "coppia", i colori totali erano $3*4=12$, nell'AGA gli sprite sono sempre a 3 colori, ma non condividono la palette a coppie! Comunque, se gli sprite sono attaccati, viene usata per tutti la stessa palette da 16 colori, quella assegnata agli sprite dispari.

Dunque, nella palette AGA a 256 colori, abbiamo 16 palette da 16 colori tra cui scegliere, usando il byte basso del BPLCON4 (\$dff10c). I bit dal 7 al 4 sono usati per scegliere il “numero” della sottopalette di 16 da usare per gli sprite pari, mentre i bit dal 3 allo 0 sono per scegliere la sottopalette degli sprite dispari.

Vediamo gli 8 bit bassi del registro BPLCON4 (\$dff10c):

bit	"nome"
0	ESPRM7 \ Scegli la sottopalette da
1	ESPRM6 \ usare per gli sprite PARI
2	ESPRM5 /
3	ESPRM4 /
4	OSPRM7 \ Scegli la sottopalette da
5	OSPRM6 \ usare per gli sprite DISPARI
6	OSPRM5 /
7	OSPRM4 /

Ed ecco una tabella di riferimento per scegliere la palette:

bit 3	bit 2	bit 1	bit 0	Sprites pari
bit 7	bit 6	bit 5	bit 4	Sprites dispari
0	0	0	0	\$180/palette 0 (colore 0)
0	0	0	1	\$1A0/palette 0 (colore 16)
0	0	1	0	\$180/palette 1 (colore 32)
0	0	1	1	\$1A0/palette 1 (colore 48)
0	1	0	0	\$180/palette 2 (colore 64)
0	1	0	1	\$1A0/palette 2 (colore 80)
0	1	1	0	\$180/palette 3 (colore 96)
0	1	1	1	\$1A0/palette 3 (colore 112)
1	0	0	0	\$180/palette 4 (colore 128)
1	0	0	1	\$1A0/palette 4 (colore 144)
1	0	1	0	\$180/palette 5 (colore 160)
1	0	1	1	\$1A0/palette 5 (colore 176)
1	1	0	0	\$180/palette 6 (colore 192)
1	1	0	1	\$1A0/palette 6 (colore 208)
1	1	1	0	\$180/palette 7 (colore 224)
1	1	1	1	\$1A0/palette 7 (colore 240)

Ecco come si usa: se per esempio volessi scegliere sia per gli sprite pari che per quelli dispari la seconda palette, quella dal color16 al color31, dovrei mettere `%0001` nei bit dallo 0 al 3 per gli sprite pari, e `%0001` nei bit dal 4 al 7 per gli sprite dispari. Per cui il byte basso sarebbe `%00010001`. Ora, questa mia preferenza corrisponde con la modalità dell'OCS/ECS, per la quale la palette degli sprite è sempre dal color16 al color31. Infatti, `%00010001` in esadecimale è `$11`, ed è per questo che facciamo:

```
1 move.w #$11,$10c(a5) ; BPLCON4 resettato
```

Per resettare la palette degli sprite!!! Svelato questo mistero, vediamo di cambiare il settaggio in un modo più utile per eventuali usi: spostiamo la palette degli sprite in fondo, ossia decidiamo che sia dal color240 al color256. In questo caso abbiamo `%11111111`. Ora potremmo però scegliere un banco da 16 per gli sprite pari diverso da quello per gli sprite dispari! Per esempio, assegnamo agli sprite pari i colori dal 224 al 240, e agli sprite dispari dal 240 al 256. Il risultato in `$dff10c` è `%11101111`.

Mettiamo in pratica questo in `Lezione15f7.s`

15.10 Nuovo scroll orizzontale superfluido (1/4 di pixel) per i bitplanes

Lo scrolling orizzontale ad 1/4 di pixel è stato implementato anche per i bitplanes. E indovinate come? Aggiungendo dei bit al `BPLCON1` (`$dff102`). Come si è visto per gli sprites, sono stati aggiunti un paio di bit "bassi" del valore di scroll. In più ne sono stati aggiunti anche due alti, che permettono scatti di 16 pixel alla volta, per un massimo di 64 pixel. Da notare che gli scatti di 16 e 32 pixel sono "abilitati" solo quando il burst (`FMODE-$dff1fc`) è a 32 o 64 bit, rispettivamente. Dunque ora lo scroll può andare da 0 a 64 pixel, a scatti di 1/4 di pixel. Ma recapitoliamo: Se prima il valore di scostamento orizzontale di ciascun playfield poteva andare da 0 a 15 (`%1111`), e sono stati aggiunti 2 bit bassi e due alti, ora può andare da 0 a `%11111111`, ossia da 0 a 255 (un valore a 8 bit!), da intendersi però come scatti da 1/4 di pixel, per cui lo scroll massimo se misurato in pixel lowres è di $256/4=64$. Ma vediamo in che posizione questi bit sono stati "incastrati" nel byte alto del vecchio `bplcon1` (`$dff102`):

BIT	"nome"	descrizione
15	PF2H7	\ bit alti (6 e 7) del valore scroll playfield 2

14	PF2H6	/
13	PF2H1	\ bit bassi (0 e 1) del valore scroll playfield 2
12	PF2H0	/
11	PF1H7	\ bit alti (6 e 7) del valore scroll playfield 1
10	PF1H6	/
09	PF1H1	\ bit bassi (0 e 1) del valore scroll playfield 1
08	PF1H0	/
07	PF2H5	\
06	PF2H4	\ bit "medi" (2,3,4,5) del val. scroll playfield 2
05	PF2H3	/
04	PF2H2	/
03	PF1H5	\
02	PF1H4	\ bit "medi" (2,3,4,5) del val. scroll playfield 1
01	PF1H3	/
00	PF1H2	/

Nota:

Il bit PFxH0 scolla di 1/4 di pixel (35ns)
 Il bit PFxH1 scolla di 1/2 pixel (70ns)
 Il bit PFxH2 scolla di 1 pixel (140ns)
 Il bit PFxH3 scolla di 2 di pixel
 Il bit PFxH4 scolla di 4 di pixel
 Il bit PFxH5 scolla di 8 di pixel
 Il bit PFxH6 scolla di 16 pixel (deve essere attivo il burst 32 bit)
 Il bit PFxH7 scolla di 32 pixel (deve essere attivo il burst 64 bit)

Come vedete il byte basso è lo stesso, mentre quello alto è AGA only.

Ora, supponiamo di voler far scorrere un bitplane verso destra a scattini di 1/4 di pixel, fino al massimo possibile con il bplcon1, ossia di 256 posizioni equivalenti a 64 pixel lowres. Dovremmo scomporre il valore di scroll, da 0 a 255, in 3 “pezzi”: i due bit bassi dovrebbero essere messi in PHxH0/1, i 4 “centrali” in PFxH2-5, e i due bit alti in PFxH6/7. Questo si può fare facilmente con qualche AND e LSL/LSR.

Vediamo una implementazione in Lezione15g1.s (1 playfield)

Vediamo un’implementazione in Lezione15g2.s (2 playfield)

Ora proviamo a fare un’effetto “onda” con la precisione di 1/4 di pixel, conventendo una sintab in valori per BPLCON1 e cambiando quest’ultimo in copperlist una volta per linea: Lezione15g3.s

Una particolarità: se la figura è in hires, non “funziona” il bit più alto dello scroll, per cui i valori possono andare da 0 a 127. Lezione15g4.s

15.11 Una nuova possibilità per ciclare la palette

Abbiamo già visto la funzione dei bit bassi del BPLCON4. I bit alti, invece, servono per “scambiare” colori nella palette, senza dover cambiare il contenuto dei registri della palette stessa.

BPLTCON4 (\$dff10c)

BIT	NOME
15	BPLAM7
14	BPLAM6
13	BPLAM5
12	BPLAM4
11	BPLAM3
10	BPLAM2

09 BPLAM1
08 BPLAM0

BPLAMx = This 8 bit field is XOR'ed with the 8 bit plane color address, thereby altering the color address sent to the color table (x=1-8) Bits 15 thru 8 of BPLCON4 comprise an 8-bit mask for the 8 bitplane address, XOR'ing the individual bits. This allows the copper to exchange colour maps with a single instruction.

Vediamo un esempio pratico di scambio tra il colore A e il colore B:

- Il contenuto del registro colore hardware non è modificato
- Tutti i pixel che venivano visualizzati usando il colore A ora sono visualizzati usando il colore B, e tutti i pixel che erano visualizzati col colore B è visualizzato col colore A. (in pratica: SCAMBIATI!)
- Il gruppo di 2^n colori dal colore 00 al colore $(2^n) - 1$ è scambiato col gruppo di 2^n colori dal colore 2^n al colore $2^n + (2^n) - 1$
- Il gruppo di 2^n colori dal colore $2 * 2^n$ al colore $2 * 2^n + (2^n) - 1$ è scambiato con il gruppo di 2^n colori dal colore $3 * 2^n$ al colore $3 * 2^n + (2^n) - 1$

L'operazione di scambio finisce quando l'hardware non trova altri gruppi di colore da scambiare.

Facciamo un esempio: se settiamo il secondi bit, BPLAM1 (bit 9 del BPLCON4), ecco come appare la palette prima e dopo l'operazione:

PRIMA		DOPO
Color 00		Color 02
Color 01		Color 03
Color 02		Color 00
Color 03		Color 01
Color 04		Color 06
Color 05		Color 07
Color 06		Color 04
Color 07		Color 05
...		...

I colori sono stati scambiati, usando gruppi di $2^1 = 2$ colori.

Non si può scambiare un solo colore. Se si modifica un bit BPLAMx, si cambia tutta la palette.

Le operazioni di scambio comunque possono essere combinate. Se più di un bit BPLAMx è settato, le operazioni di scambio per ogni bit saranno eseguite una dopo l'altra, partendo dal bit BPLAM0 fino a BPLAM7.

Esempio: *\$dff10c* contiene *\$0500* (%0000010100000000). I bit BPLAM0 e BPLAM2 sono settati. Prima saranno scambiati usando gruppi di 2^0 colori, POI sarà scambiata la palette risultante usando gruppi di 2^2 colori come in questa tabella:

PRIMA		Scambio BPLAM0		Scambio BPLAM1
Color 00		Color 01		Color 05
Color 01		Color 00		Color 04
Color 02		Color 03		Color 07
Color 03		Color 02		Color 06
Color 04		Color 05		Color 01
Color 05		Color 04		Color 00
Color 06		Color 07		Color 03

Color 07		Color 06		Color 02
Color 08		Color 09		Color 13
Color 09		Color 08		Color 12
Color 10		Color 11		Color 15
Color 11		Color 10		Color 14
Color 12		Color 13		Color 09
Color 13		Color 12		Color 08
Color 14		Color 15		Color 11
Color 15		Color 14		Color 10
...	

In pratica gli 8 bit BPLAM0-7 del BPLCON4 sono una maschera per l'indirizzo degli 8 bitplanes, dato che viene fatto uno XOR (EOR) di ogni bit.

Vediamo un esempio, solo “didattico” degli effetti di questi bit: `Lezione15h.s`

15.12 Dual playfield AGA

Il nuovo Dual Playfield può avere fino a 4 bitplanes per playfield (16 colori un playfield e 16 l'altro), e il banco dei 16 colori nella palette di 256 è selezionabile indipendentemente per ogni playfield.

Per attivare il double Playfield, occorre settare il bit 10 del BPLCON0 come al solito, scegliere i bitplanes, (per 8 planes azzerare i bit 12, 13, 14 del BPLCON0 e settare il 4, altrimenti da 2 a 6 usare i bit 12, 13, 14 e azzerare il 4). Ora occorre puntare le 2 figure, una nei bplpointer pari e una nei bplpointers dispari. Poi si deve scegliere quali banchi di colori usare per le 2 pic, in mood simile a quanto visto per gli sprite.

Questo si decide con i bit 10, 11, 12 del BPLCON3 (`$dff106`):

PF20F	BITPLANE COINVOLTI												OFFSET
2 1 0 8 7 6 5 4 3 2 1	(decimale)												
0 0 0 - - - - - - - - - 0													
0 0 1 - - - - - - - 1 - 2													
0 1 0 - - - - - - 1 - - 4													
0 1 1 - - - - - - 1 - - 8 (default)													
1 0 0 - - - 1 - - - - - 16													
1 0 1 - - 1 - - - - - - 32													
1 1 0 - 1 - - - - - - - 64													
1 1 1 1 1 - - - - - - - 128													

Si intende Playfield 2 prioritario rispetto al Playfield 1. Come vedete la situazione di default avviene quando i bit 10 e 11 sono settati, infatti per default mettiamo `$c00 (%110000000000)` in `$dff106`.

15.13 VGA/Productivity 640x480 non interlacciata

Le demo e i giochi funzionano normalmente in risoluzione PAL o NTSC, che sono supportate dai televisori o dai monitor come il 1084. La frequenza verticale pal è di 50Hz, mentre quella NTSC di 60Hz. La frequenza orizzontale è di 15Khz. Come sapete, per scegliere fra una di queste due frequenze si fa così:

```

1  move.w  #$20,$dff1dc      ; BEAMCON0 — modo PAL
2
3  move.w  #$00,$dff1dc      ; BEAMCON0 — modo NTSC
```

Questo però non funziona sui vecchi computer Amiga, fabbricati prima del 1990 o 1991. In pratica gli A1000 e i primi a500/a2000 non possiedono il FAT AGNUS, che "possiede" il BEAMCONO, mentre tale registro ha cominciato a comparire negli A500/a2000 kickstart 1.3 fabbricati dopo il 1990-91. Comunque una macchina AGA ha SICURAMENTE anche questo registro.

Avrete notato che dal workbench 2.0 in avanti è possibile scegliere il tipo di monitor, e settare una frequenza video anche "VGA" non interlacciata, ossia 640x480 a 31KHz, o anche 800x600 e altre risoluzioni particolari.

NTSC (525 linee, 227.5 colorclocks per scan line) 15Khz

PAL (625 linee, 227.5 colorclocks per scan line) 15Khz

VGA (525 linee, 114.0 colorclocks per scan line) 31Khz

Comunque per visualizzare queste risoluzioni serve un monitor almeno "VGA", oppure multisync/multiscan. La televisione e i monitor "normali" come il 1084 non riescono ad agganciare quelle frequenze.

Allora si potrebbe pensare: mi compro un monitor VGA/multisync almeno posso vedere sia la risoluzione PAL/NTSC che quella non interlacciata 31Khz! Purtroppo la maggior parte dei monitor che riesce a visualizzare il 640x480 a 31Khz non visualizza la risoluzione "televisiva" a 50/60Hz, per cui dovrete avere due monitor, uno per vedere una risoluzione e uno per vedere l'altra. Per questo occorre stare attenti! Se voleste comprare un monitor multisync/multiscan, accertatevi prima che visualizzi correttamente anche il 320x256 PAL dei videogiochi/demo, come fa il C= 1950, ad esempio.

Programmare i vari modi video 800x600 o simili è complicato, e non compatibile con tutti i monitor, per questo vedremo solo come fare il 640x480, supportato comunque anche dai peggiori monitor VGA del PC MSDOS.

Intanto vediamo un po' di nuovi registri per la sincronizzazione:

VSSTRT	- Posizione linea verticale per VSYNC start.
VSSTOP	- Posizione linea verticale per VSYNC stop.
HSSTRT	- Posizione linea orizzontale per HSYNC start.
HSSTOP	- Posizione linea orizzontale per HSYNC stop.
HCENTER	- Posizione orizzontale per VSYNC nell'interlace.

E altri per il blanking programmabile:

HBSTRT	- Posizione linea orizzontale per HBLANK start.
HBSTOP	- Posizione linea orizzontale per HBLANK stop.
VBSTRT	- Posizione linea verticale per VBLANK start.
VBSTOP	- Posizione linea verticale per VBLANK stop.

I dati che abbiamo del nostro modo video sono:

VGA (525 linee, 114.0 colorclocks per scan line) 31Khz

Dobbiamo quindi mettere in VTOTAL il numero di linee-1 (524) e in HTOTAL il numero di colorclocks per scanline-1 (113), più altri settaggi.

Per cambiare la frequenza orizzontale dai 15Khz (TV, monitor 1084), ai 31Khz dei monitor VGA/Multiscan/Multisync è necessario agire su un bel pò di registri, non solo il BEAMCONO (che serve ad abilitare altri registri):

```

1      LEA      $DFF000,A5
2
3      ;5432109876543210
4      MOVE.W  #0001101110001000,$1DC(A5) ; BEACON0 — lista dei bit settati:
5
6      ; 3 — BLANKEN — COMPOSITE BLANK OUT TO CSY PIN
7      ; 7 — VARBEAMEN — VARIABLE BEAM COUNTER COMP. ENABLED
8      ;      Abilita i comparatori variabili di beam per
9      ;      operare nel contatore orizzontale principale,
10     ;      e disabilita lo stop hardware del display in
11     ;      orizzontale e in verticale.
12     ; 8 — VARHSYEN — VARIABLE HORIZONTAL SYNC ENABLED
13     ;      Attiva i registri HSSTRT/HSSTOP (var. HSY)
14     ; 9 — VARVSYEN — VARIABLE VERTICAL SYNC ENABLED
15     ;      Attiva i registri VSSTRT/VSSTOP (var. VSY)
16     ; 11— LORDIS — DISABLE LONGLINE/SHORTLINE TOGGLE
17     ;      Disabilita lo scambio tra linee lunghe/corte.
18     ; 12— VARVBEN — VARIABLE VERTICAL BLANK ENABLED
19     ;      Attiva i registri VBSTRT/VBSTOP, e disabilita la
20     ;      "fine" hardware della finestra video.
21
22     MOVE.W  #113,$1C0(a5) ; HTOTAL — HIGHEST NUMBER COUNT, HORIZ LINE
23     ;      Color clock massimo per linea orizzontale:
24     ;      Il VGA ha 114 colorclocks per scan line!
25     ;      Il valore va da 0 a 255: 113 va bene!
26
27     MOVE.W  #01000,$1C4(a5) ; HBSTRT — HORIZONTAL LINE POS FOR HBLANK START
28     ;      I bit 0–7 contengono le posizioni di start
29     ;      e di stop del blanking orizzontale in
30     ;      incrementi di 280ns. I bit 8–10 servono per
31     ;      un posizionamento a 35ns (1/4 di pixel).
32     ;      In questo caso abbiamo settato 2240ns.
33
34     MOVE.W  #14,$1DE(a5) ; HORIZONTAL SYNC START — Numero di color
35     ;      clocks per il Sync-start.
36
37     MOVE.W  #28,$1C2(a5) ; HORIZONTAL LINE POSITION FOR HSYNC STOP
38     ;      Num. di color-clocks per Sync-stop.
39
40     MOVE.W  #30,$1C6(a5) ; HORIZONTAL LINE POSITION FOR HBLANK STOP
41     ;      Linea orizzontale di stop Horiz BLANK
42
43     MOVE.W  #70,$1E2(a5) ; HCENTER — POS. ORIZZ. di VSYNCH in interlace
44     ;      nel caso di beam counters variabili.
45
46     MOVE.W  #524,$1C8(a5) ; VTOTAL — HIGHEST NUMBERED VERTICAL LINE
47     ;      Massima linea verticale numerata, ossia
48     ;      la linea alla quale resettare il contatore
49     ;      diposizione verticale.
50     ;      Sappiamo che il modo VGA ha 525 linee.
51
52     MOVE.W  #0,$1CC(a5) ; VBSTRT — VERTICAL LINE FOR VBLANK START
53     MOVE.W  #3,$1E0(a5) ; VERTICAL SYNC START
54
55     MOVE.W  #5,$1CA(a5) ; VERTICAL LINE POSITION FOR VSYNC STOP
56     MOVE.W  #29,$1CE(a5) ; VBSTOP — VERTICAL LINE FOR VBLANK STOP
57
58     MOVE.W  #00000110000100001,$106(a5) ; 0 — external blank enable
59     ;      ; 5 — BORDER BLANK
60     ;      ; 10–11 AGA dual playfiled fix

```

Ora basta puntare la nostra copperlist in `$dff080`, ricordandosi che il bit 0 del `BPLCON0` (`$dff100`) deve essere settato, e che se si vogliono più di 1 bitplane occorre abilitare il burst a 32/64 bit con `FMODE` (`$dff1fc`).

Ad esempio:

```

1  COPPERLIST:
2      dc.w  $8E,$1c45 ; diwstrt
3      dc.w  $90,$ffe5 ; diwstop
4      dc.w  $92,$0018 ; ddfstrt
5      dc.w  $94,$0068 ; ddfstop
6      dc.w  $1e4,$100

```

```

7      dc.w    $108,0      ; modulo (non -8??)
8      dc.w    $10A,0
9
10     ; Puntate una figura 640x480.
11
12     BPLPOINTERS:
13         dc.w $e0,0,$e2,0      ; primo      bitplane
14         dc.w $e4,0,$e6,0      ; secondo   "
15         dc.w $e8,0,$ea,0      ; terzo     "
16         dc.w $ec,0,$ee,0      ; quarto    "
17         dc.w $f0,0,$f2,0      ; quinto    "
18         dc.w $f4,0,$f6,0      ; sesto     "
19         dc.w $f8,0,$fa,0      ; settimo   "
20         dc.w $fc,0,$fe,0      ; ottavo    "
21
22         dc.w  $100,$1241      ; bplcon0 (non settare bit hires, solo il
23                                ; numero dei planes e i bit 0-9 e SHRES (6))
24
25     ; qua la palette
26
27         dc.w  $180,$000
28
29         dc.w  $1fc,$8003      ; sprite scan doubling???
30         dc.w  $FFFF,$FFFE    ; Fine Coplist

```

Vediamo un esempio pratico in Lezione15i.s (Se non avete un monitor capace di visualizzare 31Khz vedrete solo delle “strisciate”).

Una nota: nessuno ha mai fatto una demo o un gioco a 31Khz, perché sono pochi gli utenti Amiga con un monitor VGA+. Se decideste di aggiungere l'opzione di visualizzare grafica in questa modalità, dovrete però prima far apparire una finestrella chiedendo se usare la frequenza normale o 31Khz!

15.14 Collisioni

Essendo stati aggiunti i bitplanes 7 e 8, occorre un CLXCON2 che potesse registrare le collisioni con questi 2 plane.

```

CLXCON2 $dff10e      - Extended collision control - controlla (se i
                    bitplane 7 e 8 sono inclusi nel detect!)
                    Questo registro è resettato quando si scrive
                    nel vecchio CLXCON - La funzione dei bit è
                    analoga a quelli dei CLXCON

```

BIT	NOME	DESCRIZIONE
15-08		Non usati
07	ENBP8	Abilita controllo bit plane 8
06	ENBP7	Abilita controllo bit plane 8
05-02		Non usati
01	MVBP8	Match value per la collisione bitplane 8
00	MVBP7	Match value per la collisione bitplane 8

Nota: disabilitare i bitplanes non previene le collisioni: se tutti i plane sono disabilitati, le collisioni sono "continue".

15.15 Blitter ECS+

Il blitter ha avuto dei potenziamenti già con l'ECS, ma per compatibilità è bene blittare sempre in modo OCS. Invece se si detecta l'AGA si è sempre sicuri di poter blittare ECS+, sempre che ci sia utile.

In pratica sono stati aggiunti il BLTSIZV (\$dff05c) e il BLTSIZH (\$dff05E), che sono, in pratica, due registri in cui mettere la grandezza VERTICALE e ORIZZONTALE della blittata, anziché

nel classico BLTSIZE (\$dff058). Prima si deve scrivere nel BLTSIZV, poi nel BLTSIZH, e parte la blittata. Nel BLTSIZV va immessa l'altezza in linee, che può andare da 0 a 32767. Se si fanno delle blittate di seguito con la stessa altezza non occorre riscrivere nel BLTSIZV (\$dff05c), rimane l'ultimo valore immesso. La blittata parte quando si scrive nel BLTSIZH (\$dff05e), in cui occorre scrivere la grandezza orizzontale della blittata in word (da 0 a 2047, ossia fino a 32768 pixel). Mettere zero in questi 2 registri equivale al massimo, come il "vecchio" BLTSIZE. La massima blittata, dunque, è stata portata a 32768*32768, rispetto alla vecchia massima blittata di 1024x1024.

Ci sono poi un paio di cosucce meno importanti:

1. Il byte \$dff05b (BLTCONOL) è un "fac simile" del byte LF dei minterms, ossia il byte basso di BLTCONO (\$dff040). Pare che renda alcune blittate leggermente più veloci, specialmente se il byte alto di BLTCONO è sempre uguale e se ne cambia quello basso scrivendo qua... Comunque non ho notato velocizzazioni particolari.
2. Il bit 7 del BPLCON1 (\$dff042), detto DOFF, quando è settato disabilita l'output dell blitter nel canale D. Questo comunque permette un input ai canali A, B e C o delle eventuali modifiche di indirizzo, senza che questo venga "scritto" nel canale D.

Spero di essere stato abbastanza chiaro e di aver detto tutto quello che serviva per programmare l'AGA. Ora non avete scuse! **dovete** fare qualcosa per il chipset AGA.

Comunque, se avete l'AGA avete anche un 68020+, quindi potrebbe risultarvi utile leggere la prossima lezione, che si occupa proprio di questo!

Parte II

Pratica

CAPITOLO 16

DALLA TEORIA ALLA PRATICA

16.1 Introduzione

La Seconda parte del Corso di Assembler per Amiga, ovvero la “Pratica” contiene tutti i listati e gli esempi, con relative immagini che mostrano il risultato del codice della lezione assemblato, suddivisi per ogni singola lezione. Questi listati, con relative immagini e commenti, occupano 1466 pagine. In questa versione cartacea del libro abbiamo pensato di allegare su CD-ROM i floppy originali (in formato ADF, Amiga Disk Format) e il libro in versione E-Book (PDF), contenenti tutti i files ed i listati, con la possibilità di caricarli direttamente nell’assemblatore Amiga o di editarli con nuovi tool di sviluppo cross-platform anche con un semplice copia&incolla, contribuendo così a salvare diversi alberi dalla distruzione.

Se non disponeste del CD ROM allegato, il libro completo di 1872 pagine in versione E-Book e i dischetti in formato ADF sono scaricabili gratuitamente dal sito:

<http://corsodiassembler.ramjam.it>

Vi lasciamo quindi con la parte di approfondimenti e le appendici, che completano questa versione da stampa del Corso di Assembler per Amiga di Fabio Ciucci.

Alessandro Sperindé e Stefania Calcagno

CAPITOLO 17

LICENZA D'USO

17.1 Introduzione

Tutti i sorgenti e gli esempi pubblicati possono essere utilizzati liberamente nei propri progetti, esattamente come era possibile farlo all'epoca della prima scrittura di questo corso di programmazione. Al giorno d'oggi esistono una moltitudine di possibilità per pubblicare il proprio codice tutelando al contempo il proprio lavoro dal “lamer” di turno.

Per la riedizione del 2016 di questo corso si è deciso “proteggere” le innumerevoli ore spese dalle molte persone che hanno contribuito alla sua realizzazione adottando per tutto il codice scritto una licenza di tipo *open source*. Questo non vuol dire che si sia deciso di limitarne in alcun modo il suo uso all'interno di future produzioni, significa solo che se tale codice troverà una sua utilità all'interno di demo, giochi o altro sarà necessario citare la fonte.

La licenza scelta è la *BSD 3-clause*, che non pone vincoli di riutilizzo anche all'interno di produzioni commerciali o l'inclusione in software di tipo *closed source*. L'unica “imposizione” è che venga sempre citata da qualche parte (che sia nei sorgenti, nella documentazione, in una scritta all'avvio del software ecc. . .) la provenienza del codice utilizzato.

Nella sezione seguente verrà riportata la licenza nella sua interezza ma non verrà ripetuta in tutti i pezzi di codice solo per una questione di editing. Si deve comunque trattare tutto il codice che non riporti altri tipi di licenza, come se fosse preceduto dal testo sotto riportato.

17.2 BSD 3-clause

BSD 3-Clause License

Copyright (c) 2016, Fabio Ciucci, RamJam
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of "Corso completo di programmazione in due dischi" nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Parte III

Approfondimenti

TEXTURE MAPPING

Autore: Alberto Longo

18.1 Premessa

Questo articolo ha lo scopo (e la pretesa) di spiegare nella maniera più semplice possibile i principi alla base del texture mapping in real time, con particolare riferimento ad alcune delle tecniche da me utilizzate nella realizzazione del motore del videogioco BREATHLESS. Esso è stato scritto nel poco tempo lasciandomi libero dalla programmazione di Breathless, dagli impegni di lavoro e dalla mia ragazza, per cui non può e non deve essere considerato come una fonte inesauribile di conoscenza, ma solo come un ottimo punto di partenza per un argomento così affascinante ed attuale. Ciò nonostante, posso assicurare senza dubbio alcuno che i lettori di questo articolo risparmieranno una notevole quantità di notti insonni, notti che io stesso ho passato nel disperato tentativo di capire come hanno fatto quelli della Id software a realizzare quel capolavoro che è Doom.

Data la complessità dell'argomento si presuppone una certa esperienza nella programmazione in assembly e, comunque, un'abbondante dose di buona volontà. Il mio consiglio è quello di leggere più volte l'articolo, nonché i sorgenti e la documentazione allegata. Per ogni approfondimento, relativo anche al 3D in generale, rimando a libri ed articoli specifici.

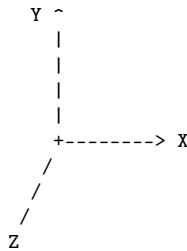
Per una buona lettura di questo articolo non è possibile prescindere da una buona conoscenza della struttura di Amiga e dell'assembly 68000+. Si suppone quindi che il lettore abbia una certa dimestichezza con tali argomenti, essendo impossibile, per ovvi motivi, soffermarsi su di essi.

L'articolo è volutamente scritto in maniera molto semplice, cercando di evitare disquisizioni troppo complesse o troppo vicine al rigido formalismo matematico, questo per permettere una agevole lettura al maggior numero possibile di persone.

Se non diversamente specificato, tutti i riferimenti riguardanti la parte hardware sono relativi alle macchine AGA.

Gli esempi di codice riportati, sono scritti in pseudo-linguaggio, oppure in assembly, e hanno scopo esclusivamente didattico. Questo significa che non sono ottimizzati nel migliore dei modi e che non sono stati testati, per cui la presenza di errori non è esclusa.

Gli assi del sistema di riferimento nello spazio sono orientati come segue:



18.2 Cenni sul formato dei numeri in virgola fissa

Nell'elaborazione di oggetti tridimensionali si ha spessissimo a che fare con numeri non interi. Tale tipo di dato è comunemente implementato nei linguaggi ad alto livello come il C e il BASIC, tramite il formato in virgola mobile. Con tale formato è possibile rappresentare un ampio insieme di numeri decimali, perdendo precisione solo quando strettamente necessario.

In Assembly le cose stanno in maniera decisamente diversa e, a meno che non si abbia a disposizione una FPU, è assolutamente improponibile l'utilizzo del formato in virgola mobile dei numeri, perché anche una semplice addizione dovrebbe essere realizzata da una routine di una certa complessità.

Si ricorre allora al formato in virgola fissa con il quale è possibile rappresentare numeri decimali utilizzando i normali numeri interi e quindi i registri del 68000. Risulta però necessario decidere in anticipo quanti bit si vogliono dedicare alla parte intera e quanti alla parte frazionaria e ciò si traduce in una scarsa flessibilità di questo tipo di rappresentazione. Il nostro scopo è però quello di eseguire il più velocemente possibile le operazioni sui numeri decimali, per cui bisogna sottostare a qualche compromesso.

Indicando che un numero è nel formato a virgola fissa, bisogna quindi specificare di quanti bit è composta la parte intera e di quanti bit è composta la parte frazionaria. In genere si utilizza la notazione $x.y$, dove x è il numero di bit dedicati alla parte intera e y il numero di bit per la parte frazionaria. Quindi, ad esempio, scrivendo 24.8 si vuole indicare che il numero è composto da 24 bit per la parte intera e da 8 bit per la parte frazionaria. In genere il formato più utilizzato è 16.16 (16 bit di parte intera e 16 di parte frazionaria), ed è quello a cui verrà fatto riferimento qui di seguito. Per ovvi motivi è conveniente che la somma del numero di bit dedicati alle due parti sia pari al numero di bit della più lunga parola che il processore è in grado di trattare, nel nostro caso 32.

La conversione da un numero decimale ad un numero in virgola fissa e viceversa si effettua tenendo conto della semplice formuletta:

$$\text{virgola_fissa} = \text{INT}(\text{decimale} * 2^{\text{bit_parte_frazionaria}})$$

Ad esempio il numero decimale 12.3456, convertito nel formato 16.16 è pari a $12.3456 * 65536 = 809081$ (la parte frazionaria ovviamente si perde), mentre all'inverso si avrà $809081 / 65536 = 12.34559631$. Come si può notare c'è una perdita di precisione che è tanto più contenuta quanto più alto è il numero di bit dedicati alla parte frazionaria.

La somma di due numeri in virgola fissa si effettua senza nessun particolare accorgimento rispetto ai numeri interi. L'istruzione

```
1 add.l d0,d1
```

è tutto quello che serve per sommare due numeri in virgola fissa contenuti rispettivamente in d0 e d1.

Il discorso è un pò più complesso per quel che riguarda le divisioni e le moltiplicazioni. Mi si conceda l'approssimazione contenuta nella seguente affermazione: un numero decimale A, nella sua forma in virgola fissa è pari ad $A \cdot K$, dove K vale $2^{\text{bit_parte_frazionaria}}$. Il prodotto dei due numeri decimali A e B nel formato in virgola fissa, vale:

$$A \cdot K * B \cdot K = (A * B) * K^2$$

Per ottenere il risultato cercato ($A \cdot B \cdot K$), si rende quindi necessaria una divisione per K (o meglio, uno shift a destra di 16 bit).

Analogamente, la divisione di due numeri A e B, vale:

$$A \cdot K / B \cdot K = A / B$$

Per evitare che la parte frazionaria venga annullata è sufficiente moltiplicare il dividendo per K:

$$A \cdot K \cdot K / B \cdot K = (A / B) * K$$

I microprocessori dal 68020 in poi sono particolarmente versatili rispetto all'implementazione dei numeri in virgola fissa, in quanto dotati di istruzioni di moltiplicazione e divisione a precisione estesa. Non bisogna però dimenticare che tali istruzioni sono comunque più lente di quelle normali, per cui in certi casi può essere preferibile utilizzare numeri in virgola fissa che entrino in una word piuttosto che in una long word.

18.3 Cos'è il Texture Mapping

Il texture mapping è una tecnica per "attaccare" un'immagine grafica in bitmap (brush) o un'immagine calcolata matematicamente (texture algoritmica) ai poligoni o, più in generale, ad una qualunque entità tridimensionale. La normale grafica vettoriale appare alquanto spoglia ed irreale, in quanto ogni oggetto è composto da un insieme di poligoni, ognuno dei quali è riempito con un unico colore. Il texture mapping aggiunge ai semplici poligoni un maggiore realismo ed una maggiore profondità, consentendo di realizzare ambienti virtuali molto più verosomiglianti e, quindi, più spettacolari.

Abbiamo quindi a disposizione un'immagine bidimensionale di dimensioni note (la nostra texture), a cui possiamo accedere tramite coordinate (u,v) per conoscere il colore di un punto. Per ovvi motivi la texture è conservata in memoria in formato chunky pixel (ogni pixel corrisponde ad un byte), ovvero come una matrice di byte. Volendo mappare la texture su un poligono nello spazio, dobbiamo trovare un sistema che ci permetta di associare ad ogni punto (x,y,z) del poligono nello spazio, un punto (u,v) della texture. Il poligono nello spazio appartiene ad un piano, a cui associamo un sistema di riferimento bidimensionale, definito da un'origine e da due versori. Se il poligono è un rettangolo, è sufficiente scegliere come origine il primo vertice e calcolare le componenti dei versori a partire dai due lati che hanno in comune il primo vertice.

Detti quindi:

```
P(x,y,z)  il generico punto del poligono di cui vogliamo calcolare il
           colore;

T(u,v)    il punto della texture associato a P;

O         l'origine del sistema di riferimento del poligono;

i, j      i versori del sistema di riferimento del poligono;

*         il simbolo di prodotto scalare;
```

possiamo scrivere:

$$T = ((P-O)*i, (P-O)*j)$$

Conosciamo ora le coordinate del punto $T(u,v)$ che possiamo usare per leggere dalla texture il colore del punto P .

Come si può notare i calcoli da effettuare per ogni punto sono troppo complessi per un'applicazione in real-time. Come è possibile semplificare e velocizzare il tutto ?

Un primo importante passo è la semplificazione del problema. Lo scopo che ci prefiggiamo è quello di realizzare un motore che ci permetta di "passeggiare" all'interno di un mondo tridimensionale, e per fare ciò è sufficiente avere la possibilità di muoversi e di ruotare lo sguardo a destra o a sinistra. Questo porta ad alcune semplificazioni che renderanno il motore notevolmente più veloce:

1. muri e pavimenti devono essere tra loro perpendicolari
2. è possibile variare la posizione dell'osservatore solo sugli assi X e Z e non sull'asse Y
3. è possibile ruotare lo sguardo solo intorno all'asse Y

Questo significa che il nostro mondo è in effetti bidimensionale. Il nostro intento è quello di farlo sembrare tridimensionale.

18.4 Come viene implementato il texture mapping in applicazioni real-time?

Un primo semplice esempio

Supponiamo di voler utilizzare come texture un brush dalle dimensioni di 128x128 pixel e di volerlo mappare su un poligono di forma quadrata che, una volta ruotato, traslato e proiettato in 2d, appare a video come un quadrato dalle dimensioni di 64x64 pixel. Banalmente, bisognerà tracciare sul quadrato a video solo un pixel ogni 2 ($2=128/64$). Se invece il quadrato a video ha dimensioni di 32x32 pixel, basta tracciare un pixel ogni 4 ($4=128/32$). E' quindi facile intuire l'importanza della semplice relazione:

$$\text{Step} = \text{BrushDim} / \text{ScreenDim}$$

dove:

Step : Passo (si tratta di un numero con virgola)
 BrushDim : Dimensione iniziale del brush
 ScreenDim : Dimensione a video del brush

Volendo infatti mappare la texture su un quadrato a video, di lato pari a ScreenDim, potremmo scrivere qualcosa del tipo:

```
Step = BrushDim / ScreenDim
for y=0 to ScreenDim
  v = y * Step;
  for x=0 to ScreenDim
    u = x * Step
    WriteScreenPixel(x,y,ReadTexturePixel(u,v))
  endfor
endfor
```

18.4. COME VIENE IMPLEMENTATO IL TEXTURE MAPPING IN APPLICAZIONI REAL-TIME 321

dove:

Step, u, v sono variabili in floating point

ReadTexturePixel(u,v) è la funzione che legge il colore del pixel di coordinate (u,v) della texture;

WriteScreenPixel(x,y,c) è la funzione che scrive un pixel di colore c a video, alle coordinate (x,y).

Ma quello che ci interessa è la velocità, e questa routine è ancora decisamente lenta. C'è prima di tutto bisogno di un accesso diretto alla memoria, e di eliminare dai cicli le istruzioni lente (le moltiplicazioni):

```
Step = BrushDim / ScreenDim
v = 0
for y=0 to ScreenDim
    u = 0
    screen = ScreenBase + 320 * y
    for x=0 to ScreenDim
        screen[x] = texture[v][u]
        u += Step
    endfor
    v += Step
endfor
```

dove:

ScreenBase è l'indirizzo di uno schermo in chunky pixel;

Come si può notare, le moltiplicazioni sono state sostituite da somme, mentre per la lettura di pixel dalla texture e per la scrittura a video sono stati utilizzati accessi diretti alla memoria, tramite gli array screen[] e texture[][]. Inoltre sia la texture che lo schermo sono organizzati in chunky pixel.

Per fare di meglio è più conveniente passare all'assembly:

```
1  ;a0 = ptr alla texture
2  ;a1 = ptr allo schermo chunky
3  ;d0 = u (nel formato 16.16, cioè 16 bit interi e 16 bit frazionari)
4  ;d1 = v (nel formato 16.16)
5  ;d2 = offset all'interno della texture
6  ;d4 = Step (nel formato 16.16)
7  ;d5 = ScreenDim * 320
8  ;d6 = x
9  ;d7 = y
10
11      moveq    #0,d1          ;v=0
12      move.w   ScreenDim,d5
13      mulu.w   #320,d5        ;Sapete come si ottimizza questo, no ?
14      moveq    #0,d7          ;inizializza x
15 loopy  moveq    #0,d0          ;u=0
16      move.l   ScreenBase,a0
17      add.l    d7,a0          ;a0=ptr alla riga attuale a schermo
18      move.l   d1,d2
19      clr.w    d2
20      swap     d2
21      lsl.w    #8,d2          ;d2=offset riga attuale della texture
22      move.w   ScreenDim,d6
23      subq.w   #1,d6          ;inizializza y
24 loopx  swap     d0
25      move.b   d0,d2
26      swap     d0
27      move.b   (a1,d2.l),(a0)+ ;Copia pixel da texture a schermo
28      add.l    d4,d0          ;u+=Step
```

```

29      dbra      d6, loopx
30      add.l     d4, d1          ; v += Step
31      add.l     #320, d7
32      cmp.l     d5, d7
33      bne      loopy

```

Come si può facilmente intuire, l'esempio qui riportato non effettua altro che lo zoom di un brush, al variare di ScreenDim, ma è estremamente significativo per la comprensione dei principi che sono alla base del texture mapping e di una sua implementazione in applicazioni real-time.

È importante fare attenzione al fatto che il quadrato di cui sopra viene suddiviso in una serie di trattini orizzontali. Ogni trattino è a sua volta composto da un certo numero di pixel. A questo punto i pixel da tracciare a schermo vengono "campionati" dal brush ad una distanza gli uni dagli altri pari a Step.

Si osservi questo semplice esempio in cui il brush originario ha dimensioni di 10x10 pixel, mentre le dimensioni a video sono di 5x5. Il valore di Step è, ovviamente, $10 / 5 = 2$ per cui, dal brush saranno scelti solo i trattini orizzontali di posto pari e, all'interno di ogni trattino, solo i pixel di posto pari:

Brush 10x10		A video 5x5
A . B . C . D . E .		
.		
F . G . H . I . J .		A B C D E
.		F G H I J
K . L . M . N . O .	--->	K L M N O
.		P Q R S T
P . Q . R . S . T .		U V W X Y
.		
U . V . W . X . Y .		
.		

Se, invece, le dimensioni a video sono di 4x4, Step avrà valore pari a $10 / 4 = 2.5$ e il risultato sarà:

Brush 10x10		A video 4x4
A . B . . C . D . .		
.		
E . F . . G . H . .		A B C D
.		E F G H
.	--->	I J K L
I . J . . K . L . .		M N O P
.		
M . N . . O . P . .		
.		
.		

Passiamo a qualcosa di più concreto

E' stato già detto che il mondo tridimensionale in cui abbiamo intenzione di muoverci ha solo muri verticali e l'unica rotazione permessa è quella intorno all'asse Y. Ogni muro è banalmente rappresentato da un poligono; lo stesso dicasi per ogni pezzo di pavimento o soffitto.

Supponiamo quindi di dover mappare la texture su un quadrato ruotato intorno all'asse Y. Il quadrato appare a video come un trapezio ruotato di 90 gradi e rappresenta un muro:

```

| \
| \

```



Come si può facilmente notare, questa figura è formata da una serie di trattini verticali di lunghezza decrescente, ovvero composti da un numero di pixel decrescente. Per ogni trattino verticale è sufficiente eseguire un ciclo simile al seguente:

```
loop    move.b  (a0,d0.w),(a1) ;Copia il pixel
        add.w   d3,d1          ;Somma la parte frazionaria
        addx.w  d2,d0          ;Somma la parte intera (+ riporto)
        adda.l  d4,a1          ;Sposta il ptr. allo schermo
        dbra    d7,loop
```

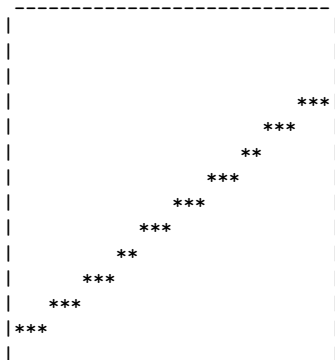
dove:

```
d0 = parte intera del contatore
d1 = parte frazionaria del contatore
d2 = parte intera dello Step
d3 = parte frazionaria dello Step
d4 = numero di pixel per ogni riga di schermo
d7 = numero di pixel da tracciare per il trattino corrente
a0 = ptr. alla colonna della texture corrispondente al trattino
a1 = ptr. al pixel corrente nello schermo
```

La texture è conservata in memoria come una matrice organizzata per colonne, questo per semplificare l'accesso ad ogni pixel della colonna corrente della texture. Il calcolo della colonna della texture da visualizzare, va fatto tenendo conto delle leggi della prospettiva.

Pavimenti e soffitti

Nel tipo di motore 3D che si intende realizzare, pavimenti e soffitti sono perfettamente orizzontali, nonchè perpendicolari ai muri. Il texture mapping di poligoni di questo genere è un pò più complesso di quello dei muri in quanto è necessario scorrere la texture secondo linee oblique. Inoltre è necessario effettuare il tracciamento per strisce orizzontali di pixel e non verticali, come avviene per i muri. Si osservi la seguente figura, rappresentante la texture (da 64x64 pixel) da mappare su un pezzo di pavimento (o soffitto):



Per ogni striscia orizzontale di pavimento (o soffitto) è necessario scorrere la texture secondo una linea non necessariamente orizzontale o verticale. Tale linea è rappresentata in figura per mezzo di asterischi (*). Il loop di texture mapping è qualcosa di simile al seguente:

```
for x = x1 to x2
  WriteScreenPixel(x,y,ReadTexturePixel(u & 63, v & 63))
  u += du
  v += dv
endfor
```

dove:

```
x1 = colonna iniziale della striscia orizzontale di pixel
x2 = colonna finale della striscia orizzontale di pixel
y  = riga su cui si trova la striscia di pixel
u, v = coordinate all'interno della texture
du = valore di somma per u
dv = valore di somma per v
```

Il problema, a questo punto, è costituito dal calcolo dei valori iniziali di u , v , du , dv . La tecnica adottata per il calcolo di tali valori dipende dall'approccio utilizzato nella realizzazione del motore.

Per chiarire un pò meglio le idee, si tenga presente che il poligono da tracciare appartiene al pavimento e, quindi, ad un piano. Su tale piano sono "incollate", una vicino all'altra, le texture da 64x64 pixel, in modo da coprirlo per intero. Ogni punto di tale piano è individuato tramite la coppia di coordinate (u,v) e, seguendo le regole della prospettiva, corrisponde ad un pixel a video di coordinate (x,y) .

Le coordinate a video del punto iniziale e del punto finale della striscia orizzontale di pixel sono noti e sono rispettivamente $(x1,y)$ e $(x2,y)$. A tali punti corrispondono i punti $(u1,v1)$ e $(u2,v2)$ nel piano di texture che vanno calcolati. Il valore iniziale della coppia (u,v) è proprio $(u1,v1)$, mentre il valore di (du,dv) è dato da:

$$du = (u2 - u1) / (x2 - x1 + 1)$$

$$dv = (v2 - v1) / (x2 - x1 + 1)$$

A titolo di esempio, consiglio di dare un'occhiata al sorgente AMOS presente nel file "TMap-Floor.lha".

18.5 Il calcolo della scena da tracciare

Riguardo al texture mapping sono molto utilizzati i due termini "ray-casting" e "BSP", ma non tutti sanno a cosa esattamente si riferiscano. Per tracciare a video una scena, non c'è solo bisogno di sapere come tracciarla, ma anche e soprattutto di sapere cosa tracciare. Il ray-casting e i BSP tree sono due dei metodi più diffusi per calcolare cosa tracciare in base al punto di vista dell'osservatore. In un classico labirinto alla Wolfenstein è contenuto un numero molto elevato di poligoni, e volerli analizzare tutti per decidere quali fanno effettivamente parte della scena da visualizzare, è assurdo. C'è bisogno di tecniche più veloci e sia il ray-casting che i BSP, ci vengono in aiuto.

Ray-Casting

Viene spontaneo notare che il ray-casting ha un nome simile al più famoso ray-tracing (l'algoritmo utilizzato per immagini 3D fotorealistiche), ed in effetti la somiglianza non si ferma al nome. L'algoritmo del ray-tracing consiste nel tracciare un raggio (una linea) tra l'osservatore ed ognuno dei pixel che compongono lo schermo. Per ognuno di questi raggi vengono poi calcolate collisioni e rifrazioni in modo da ricavare il colore del pixel corrispondente.

Il ray-casting non è altro che una semplificazione del ray-tracing: viene tracciato un solo raggio per ogni colonna dello schermo. Il guadagno in termini di velocità risulta subito evidente se si pensa che per calcolare un frame a 320x200 pixel sono necessari 320 raggi all'algoritmo del ray-casting contro i 64000 del ray-tracing !

Quella che potrebbe sembrare una esagerata approssimazione è invece un'idea tanto semplice quanto geniale. Non bisogna infatti dimenticare che il mondo che intendiamo visualizzare è soggetto a dei limiti per cui pavimenti e pareti sono tra loro perpendicolari. In più è possibile muoversi solo su un piano (la coordinata Y non può variare).

Si osservi il grafico seguente:

```

0   64  128  192  256  320  384  448  512 X
+---+---+---+---+---+---+---+---+---+---+
|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|
|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|
64 +---+---+---+---+---+---+---+---+---+
|XXX|   |   |   |   |   |   |XXX|
|XXX|   |   |   |   |   |   |XXX|
128 +---+---+---+---+---+---+---+---+---+
|XXX|   |XXX|XXX|   |XXX|   |XXX|
|XXX|   |XXX|XXX|   |XXX|   |XXX|
192 +---+---+---+---+---+---+---+---+---+
|XXX|   |XXX|   |   |XXX|   |XXX|
|XXX|   |XXX|   |   |XXX|   |XXX|
256 +---+---+---+---+---+---+---+---+---+
|XXX|   |XXX|   |   |XXX|   |XXX|
|XXX|   |XXX|   |   |XXX|   |XXX|
320 +---+---+---+---+---+---+---+---+---+
|XXX|   |XXX|XXX|XXX|XXX|   |XXX|
|XXX|   |XXX|XXX|XXX|XXX|   |XXX|
384 +---+---+---+---+---+---+---+---+---+
|XXX|   |   |   |   |   |   |XXX|
|XXX|0->|   |   |   |   |   |XXX|
448 +---+---+---+---+---+---+---+---+---+
|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|
|XXX|XXX|XXX|XXX|XXX|XXX|XXX|XXX|
512 +---+---+---+---+---+---+---+---+---+
|
Z |
V

```

Esso rappresenta una semplice mappa bidimensionale organizzata per blocchi che possono essere pieni o vuoti. Un blocco pieno non può essere attraversato, un blocco vuoto sì. Ogni blocco deve essere immaginato come una specie di cubo di dimensioni fisse, in genere 64 unità (o pixel) per lato, la cui faccia inferiore poggia sulla mappa, nella posizione indicata dalla griglia. La faccia inferiore e quella superiore rappresentano rispettivamente pavimento e soffitto dei blocchi vuoti. Le quattro facce perpendicolari al pavimento rappresentano altrettanti pezzi di muro. Ad ogni faccia è associata una texture di dimensioni 64x64 pixel.

Un blocco ha, quindi, sei parametri: un puntatore alla texture del pavimento, un puntatore alla texture del soffitto e quattro puntatori alle texture delle rimanenti quattro facce. Se il blocco

è pieno, non ha bisogno dei puntatori alle texture di soffitto e pavimento (sono posti a zero), mentre se è vuoto, non ha bisogno dei puntatori alle texture dei quattro pezzi di muro.

L'osservatore ha tre parametri: coordinata x, coordinata z e angolo di osservazione. Nel grafico precedente l'osservatore è rappresentato da una O e da una freccia che indica la direzione dello sguardo. Dividendo le coordinate x e z per la dimensione dei blocchi si calcola facilmente su quale blocco si trova l'osservatore.

Si immagini ora lo schermo come composto da 320 colonne da 200 pixel ciascuna.

L'algoritmo del ray-casting si può riassumere nei seguenti passi:

1. Tenendo conto dell'angolo di osservazione, calcolare il raggio (la retta) da tracciare tra l'osservatore e ognuna delle 320 colonne che compongono lo schermo.
2. A partire dal blocco su cui si trova l'osservatore e usando l'algoritmo di tracciamento delle linee di Bresenham, esaminare ogni blocco attraverso cui passa il raggio corrente finché non se ne trova uno pieno.
3. Se il blocco è pieno, vuol dire che c'è un'intersezione tra il raggio e due dei 4 lati del blocco. Calcolare le coordinate del punto di intersezione del lato più vicino all'osservatore. In questo modo, grazie ad una semplice operazione di and, è anche possibile calcolare quale dei 64 pixel del lato è stato colpito dal raggio. Questo dato è estremamente utile durante il texture mapping vero e proprio.
4. Calcolare la distanza tra l'osservatore ed il punto di intersezione.
5. Calcolare l'altezza del muro in pixel (tenendo conto delle leggi della prospettiva) nel punto di intersezione.
6. Tracciare a schermo, nella colonna corrente, tramite una routine di texture mapping, il pezzo di muro intersecato.
7. Tornare al punto 2 finché non sono stati tracciati i 320 raggi.

L'algoritmo, come si può notare, è molto semplice, ma implementarlo in maniera efficace è tutt'altro discorso. A tal proposito consiglio di studiare i sorgenti e, soprattutto, il file di documentazione `notes.txt` contenuti nell'archivio `ack3d.zip` allegato a questo articolo. Tale archivio contiene, oltre ai sorgenti, un'analisi approfondita di una delle possibili implementazioni dell'algoritmo del ray-casting. A quanti riuscissero ad implementare in un programma funzionante le soluzioni proposte dall'autore dell'archivio, consiglio di studiare nuove e più efficienti strade per raggiungere i medesimi risultati. Posso testimoniare in prima persona che si può fare molto di meglio.

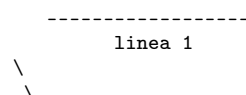
I sorgenti sono per PC, ma il loro valore didattico resta immutato.

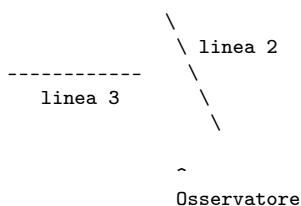
BSP Trees

Più complessa, invece, è l'idea di base dei BSP tree. Prima di tutto, BSP tree significa per esteso: Binary Split tree, ovvero alberi binari di divisione.

Vediamo come funzionano:

Si osservi la seguente figura, in cui 3 linee risiedono sul piano:



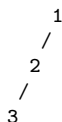


Volendo disegnare la scena utilizzando il classico algoritmo del pittore, bisognerebbe prima di tutto ordinare le linee in base alla distanza, quindi tracciarle in ordine dalla più lontana alla più vicina. Questa tecnica, oltre ad essere decisamente lenta, è anche soggetta a notevoli errori di imprecisione spesso difficili da eliminare.

Utilizzando i BSP, il calcolo dell'ordine di tracciamento viene fatto una volta per tutte al di fuori del motore 3D, creando un albero binario contenente tutte le informazioni necessarie a tracciare nell'ordine giusto le linee, qualunque sia la posizione dell'osservatore.

Prima di tutto si noti che, dato un qualunque punto (x,y), si può sempre dire se esso si trova su un lato o sull'altro di una linea. Se il punto dovesse appartenere alla linea, lo si può considerare come appartenente ad uno dei due lati.

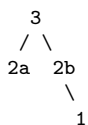
L'albero binario dei BSP è costituito da una serie di nodi che rappresentano le linee che si vogliono tracciare. Alla destra di ogni nodo si mettono tutte le linee che si trovano su un lato, mentre alla sinistra si mettono tutte le linee che si trovano sull'altro lato. Così, riguardo all'esempio precedente, l'albero potrebbe essere:



Ma è possibile utilizzare qualunque altra linea come nodo di testa:



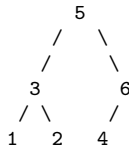
Volendo però utilizzare la linea 3 come nodo di testa, sorge un problema: su quale lato della linea 3 si trova la linea 2? La risposta è semplice: su entrambi. Si procede allora a suddividere la linea 2 in due parti, tagliate a metà dal prolungamento della linea 3. L'albero assume, quindi, questo aspetto:



Le linee 2a e 2b sono porzioni dell'originaria linea 2. Tracciando entrambe le linee a video, si otterrà esattamente la linea 2.

Durante la creazione di un albero BSP è necessario cercare di minimizzare il numero di suddivisioni di linee, pena un aumento spropositato delle dimensioni dell'albero stesso, e quindi del tempo necessario al tracciamento di una scena.

Per tracciare una scena bisogna partire dal nodo di testa e calcolare su quale lato della linea è posto l'osservatore. Si visita il nodo relativo all'altro lato, si traccia la linea corrente, e poi si visita il nodo relativo al lato su cui si trova l'osservatore, il tutto in maniera ricorsiva. Ad esempio, il seguente albero:



genera la seguente sequenza se l'osservatore si trova a destra di tutte le linee:

4 - 6 - 5 - 2 - 3 - 1

L'estensione di questi concetti al 3D è semplice. Si considerino al posto delle linee, dei poligoni e si supponga che il poligono 1 sia il nodo di testa. Per sapere dove inserire il poligono 2 nell'albero è sufficiente calcolare su quale lato si trovano tutti i suoi punti rispetto al poligono 1. Se una parte del poligono 2 dovesse essere su un lato e l'altra parte sull'altro lato del poligono 1, si dovrebbe dividere il poligono 2 in due parti. Per fare ciò è sufficiente prendere in considerazione la linea formata dall'intersezione tra il poligono 2 e il piano a cui appartiene il poligono 1 e suddividere il poligono 2 lungo questa linea.

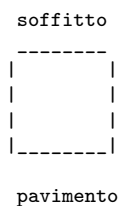
E' però da notare che per realizzare un motore simile a Doom, è sufficiente utilizzare i BSP tree nel caso bidimensionale. Il motivo risulterà chiaro dopo aver letto il paragrafo successivo.

Come si può facilmente comprendere, i BSP tree hanno notevoli vantaggi sul ray-casting: sono più veloci, danno la possibilità di tracciare muri obliqui e di ogni dimensione e, più in generale, danno la possibilità di realizzare ambienti più complessi e realistici. Per contro c'è da indicare un maggiore difficoltà d'uso

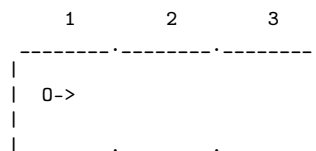
18.6 Saliamo le scale!

Le tecniche descritte fino a questo punto permettono di visualizzare scene tratte da un mondo sostanzialmente bidimensionale. Non si tratta cioè di vero 3D, ma solo di una parvenza.

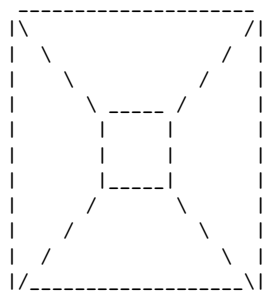
Un passo avanti nella realizzazione di un mondo tridimensionale più realistico può essere fatto con una tecnica abbastanza semplice. Si prenda in considerazione un blocco vuoto dell'algoritmo del ray-casting descritto poco prima. Se l'osservatore fosse nel blocco, ovviamente, si troverebbe tra pavimento e soffitto. La sezione laterale del blocco si presenta così:



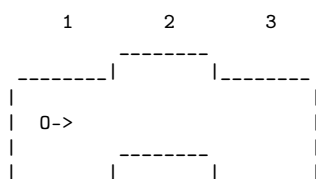
Sempre considerando una sezione laterale, si accostino 2 blocchi al precedente:



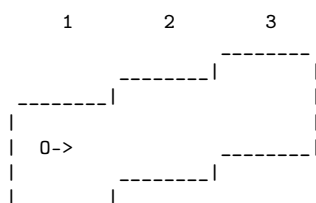
L'osservatore, che si trova sul blocco 1, vede (approssimativamente) questa scena (un corridoio):



I tre blocchi si trovano alla stessa altezza ma cosa succede se, ad esempio, il blocco centrale si trova più in alto rispetto agli altri due ?

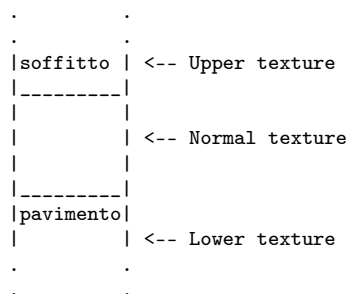


L'osservatore non vede più solo un semplice corridoio, ma vede anche un gradino. Innalzando il blocco 3 rispetto al blocco 2 si avrà:



L'osservatore, a questo punto, vede una piccola scalinata, composta da due gradini. Questo significa che ai sei parametri del blocco, bisogna aggiungere l'altezza del pavimento e l'altezza del soffitto.

C'è però un piccolo problema: tra due pavimenti (o soffitti) adiacenti che si trovano a differente altezza, rimane dello spazio che deve essere riempito in qualche modo. Si giunge allora ad una nuova definizione del blocco ed all'aggiunta di altri parametri. Si osservi la seguente figura, rappresentante la sezione laterale di un blocco in quella che è la sua nuova definizione:



Per ognuna delle quattro facce laterali (quindi per ogni muro), sono ora definite tre texture:

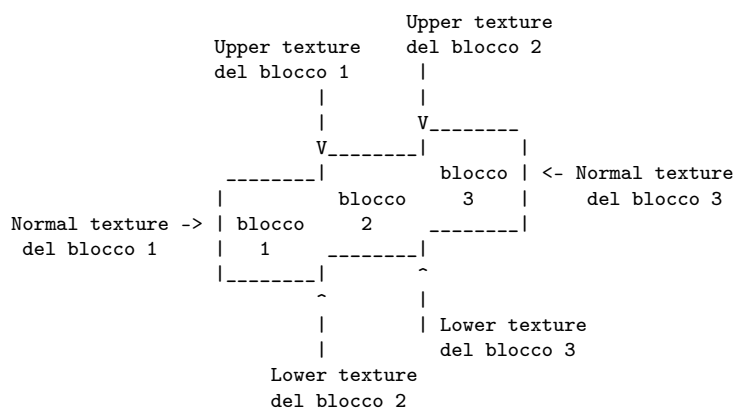
Normal è la texture visualizzata tra soffitto e pavimento, se il blocco è pieno;

Upper è la texture visualizzata tra due soffitti adiacenti che si trovano altezza differente;

Lower è la texture visualizzata tra due pavimenti adiacenti che si trovano altezza differente;

Il numero di parametri di ogni blocco è a questo punto salito a sedici.

Per chiarire meglio le idee, si osservi la seguente figura:



Come si può notare, il mondo risultante da questa nuova definizione, è anch'esso sostanzialmente bidimensionale. Il realismo è però sicuramente superiore. Ne è prova l'enorme successo riscosso da Doom.

L'applicazione ai BSP tree dei concetti descritti in questo paragrafo è semplice. Prima di tutto l'unità fondamentale non è costituita dai blocchi ma, ovviamente, dalle linee. Ogni linea rappresenta un muro ed è quindi dotata delle tre texture (upper, normal e lower). Le linee costituiscono i lati di poligoni detti settori. Ogni settore ha come parametri l'altezza e la texture di pavimento e soffitto.

Per un maggiore approfondimento consiglio la lettura delle specifiche dei file WAD di Doom, contenute nel file "DoomSpecs.guide".

La "tecnica del pittore"

Siamo ora a conoscenza delle due tecniche più usate per decidere cosa tracciare per visualizzare una scena. Utilizzando il ray-casting, ci ritroveremo con una lista di trattini verticali, ognuno relativo ad un muro, mentre utilizzando i BSP tree, ci ritroveremo con una lista di facce, scomponibili in trattini verticali.

[DA COMPLETARE]

18.7 Illuminiamo il nostro mondo

E' possibile dotare i blocchi (nel caso del ray-casting) o i settori (nel caso dei BSP) del parametro di illuminazione. Tale parametro viene utilizzato durante il texture mapping per accedere ad una tabella di illuminazione e variare la luminosità di ogni pixel delle texture del blocco.

Si supponga che le texture siano a 256 colori. Questo significa che ogni pixel di una qualunque texture può assumere valori tra 0 e 255. La palette di 256 colori deve essere composta da un certo numero di sfumature di un certo numero di colori di base (ad esempio 32 grigi, 32

marroni, 32 rossi, 16 blu, etc.), in modo da coprire un pò tutte le esigenze cromatiche di una qualunque immagine. Quindi ogni colore è presente più volte nella palette ma con intensità luminosa diversa.

E' quindi possibile costruire una tabella che associa ad ogni colore della palette un altro colore della palette stessa, ma con differente luminosità. Gli elementi di questa tabella assumono, ovviamente, valori tra 0 e 255.

Costruendo M tabelle di questo tipo, ognuna relativa ad una diversa luminosità (compresa tra 100% e 0%), si ottiene una matrice N x M, dove N è il numero di colori della palette (256). Con 32 livelli di illuminazione (e quindi 32 tabelle) si ottiene una matrice che occupa quindi $256 \times 32 = 8192$ bytes.

Supponendo che il colore 0 della palette sia il nero, la matrice può essere così presentata:

		COLORI DELLA PALETTE									
		0	1	2	3	253	254	255		
L U M I N O S I T A'	100%	0	1	2	3	253	254	255		
		
		
		
	75%		
		
		
		
	50%		
		
		
		
	25%		
		
		
	0%	0	0	0	0	0	0	0		

Come si può notare, la prima tabella (quella relativa alla luminosità 100%) fa corrispondere ad ogni colore della palette, il colore stesso, per cui in teoria si potrebbe evitare di accedere alla tabella. L'ultima tabella, relativa alla luminosità minima, fa invece corrispondere ad ogni colore della palette, il colore zero, cioè il nero. Le tabelle intermedie devono essere calcolate tramite una routine apposita. Allegato a questo articolo c'è un sorgente C adatto allo scopo. Il suo nome è: `MakeLTable.c`

Per gestire l'illuminazione, il ciclo di texture mapping va modificato in questo modo:

```

1  moveq    #0,d5
2  loop    move.b    (a0,d0.w),d5      ;Legge il pixel dalla texture
3         move.b    (a2,d5.l),(a1)    ;Legge dalla light table e scrive
4         add.w     d3,d1              ;Somma la parte frazionaria
5         addx.w    d2,d0              ;Somma la parte intera (+ riporto)
6         adda.l    d4,a1              ;Sposta il ptr. allo schermo
7         dbra      d7,loop

```

dove a2 è il puntatore alla tabella di illuminazione che deve ovviamente essere calcolato al di fuori del ciclo, tenendo conto della luminosità del mondo in quel punto e della distanza dall'osservatore. In un ambiente al massimo della luminosità, a2 punta sempre alla prima tabella, mentre in un ambiente totalmente buio punta all'ultima.

18.8 Texture mapping e Amiga

Penso sia noto a tutti che la Id, la casa di software che ha realizzato Doom, ha sempre creduto che fosse impossibile portare Doom su Amiga. Secondo loro, il 4000/40 sarebbe sufficientemente veloce per Doom se non fosse per la mancanza di una modalità grafica in chunky pixel e per l'elevato prezzo di tale macchina. Di realizzare Doom per un 1200, neanche a parlarne. Hanno ragione? In parte sì. Doom è realizzato principalmente in C, e solo una minima parte del codice è stata scritta in assembly, per cui solo i processori più veloci possono farlo girare ad una velocità decente. Riscriverlo interamente in assembly per l'Amiga non sarebbe affatto conveniente, per cui la Id ha preferito evitare di realizzare una conversione. Dal punto di vista della convenienza commerciale, quindi, la Id ha sicuramente ragione, ma dal punto di vista puramente tecnico? Come afferma la Id, i problemi sono sostanzialmente due: la scarsa diffusione di processori veloci (e di conseguenza, di costo elevato) e la mancanza di un modo grafico in chunky pixel.

Contrariamente a quello che si crede, la mancanza del chunky pixel non è il problema principale del texture mapping su Amiga. Sono in molti a credere che i modi grafici planari siano ben otto volte più lenti dei corrispettivi modi in chunky pixel, ma questo non è del tutto vero. E' vero che se si vuole scrivere un solo pixel per volta in uno schermo planare, bisogna accedere otto volte alla memoria per scrivere ognuno degli otto bit che compongono il pixel, ma è pur vero che è in certe applicazioni non è affatto necessario scrivere un solo pixel per volta. Esistono infatti tecniche che permettono di ottimizzare l'accesso alla memoria video e che permettono di avvicinarsi in maniera soddisfacente alle prestazioni delle modalità grafiche in chunky pixel. E' questo il caso della conversione "chunky to planar".

Conversione Chunky to Planar

Questa tecnica prevede l'uso di un finto buffer in chunky pixel, che il programma tratta come se fosse un vero schermo in chunky pixel. Terminata l'elaborazione di un frame, basta eseguire una routine che si occupa di convertire, il più velocemente possibile, il finto schermo chunky pixel nello schermo planare visualizzato dal chip-set di Amiga. Esistono diverse tecniche (e varianti delle stesse) per effettuare la conversione, ed è possibile trovare nel pubblico dominio un certo numero di routine già pronte e complete di sorgenti. In generale si può dire che esistono due grandi famiglie di routine di conversione: quelle che utilizzano il blitter e quelle che non ne fanno uso. Le prime sono ottime per le macchine non troppo veloci, come il 1200 base o il 1200 con fast. Le seconde sono, invece, preferibili su macchine veloci, quelle dotate di 68030 50Mhz o di 68040. Cerchiamo di capire perchè.

E' risaputo che la memoria chip è molto più lenta della fast, sia perchè rallentata dagli accessi DMA, sia perchè funzionante con un clock di soli 7 Mhz. Il 68040 25Mhz, che potrebbe accedere alla RAM in soli 2 cicli (80ns), ha bisogno di qualcosa come 16 cicli (640ns) per accedere alla chip (vengono cioè inseriti 14 cicli di attesa): il collo di bottiglia è proprio lì, nell'accesso alla chip RAM. Grazie alla pipeline, dopo una operazione di accesso in scrittura alla memoria, i processori dal 68020 in su possono passare subito ad eseguire altre istruzioni che non accedano alla memoria, senza attendere che la scrittura sia terminata, questo sempre che il codice da eseguire sia nella cache. Le istruzioni eseguite "all'ombra" dell'istruzione di accesso in scrittura alla memoria vengono comunemente chiamate "free instructions" o istruzioni gratuite. Data la maggiore velocità del 68040 rispetto al 68020 e tenuto conto del collo di bottiglia rappresentato dalla chip RAM, si nota facilmente come il 68040 sia in grado di eseguire un buon numero di istruzioni gratuite. Volendo fare un semplice esempio, l'insieme di istruzioni:

```
1  move.l d0,(a0)+
2  move.l d1,(a0)+
3
```

```

4  è veloce quanto il seguente:
5
6  move.l d0,(a0)+
7  add.l  d2,d0      <— istruzione gratuita
8  move.l d1,(a0)+

```

Il numero di istruzioni gratuite varia in dipendenza dal numero di cicli di attesa imposti al processore, e dalla velocità del processore stesso. Maggiori sono questi due parametri, maggiori sono le limitazioni imposte dal collo di bottiglia rappresentato dalla chip RAM. Di conseguenza il numero di istruzioni gratuite aumenta.

Tutto questo significa che insieme ad una semplice copia di dati dalla fast alla chip RAM, è possibile eseguire, su un processore veloce, anche un'elaborazione dei dati stessi, il tutto senza tempi aggiuntivi.

Purtroppo, per quanto riguarda la conversione chunky to planar, non è possibile scrivere tutto il codice in modo tale che tutte le istruzioni che non accedono alla memoria siano gratuite, ma è possibile andare vicino a questo risultato.

Come se non bastasse, sembra che il blitter sia sensibilmente più lento sulle macchine dotate di processori più veloci.

In conclusione, riporto un confronto dei tempi di esecuzione, su 1200 e 4000, della routine di chunky to planar che utilizzo per il 1200, e che sfrutta sia il blitter, sia il processore:

```

- A1200+fast:
    68020   :  41 msec
    Blitter :  66 msec
    -----
    Totale  : 107 msec

- A4000:
    68040   :  24 msec
    Blitter :  80 msec
    -----
    Totale  : 104 msec

```

Come si può notare, il Blitter del 4000 è sensibilmente più lento di quello del 1200, per cui le prestazioni globali di questa routine di chunky to planar sono quasi identiche tra 1200 e 4000. In verità sul 4000 le cose vanno comunque meglio che sul 1200, dato che la parte di conversione affidata al processore è di 17 msec più veloce.

Copper Chunky

Esiste un trucco per ottenere una specie di modalità grafica in chunky pixel su Amiga: si tratta del “copper chunky”. Vediamo un pò di cosa si tratta. Sappiamo bene che il copper è in grado di modificare il contenuto dei registri colore e, quindi, dei pixel sullo schermo. Proviamo allora a scrivere una copper list che cambi il colore del fondo in questo modo:

```

1  $0180, $0f00
2  $0180, $0000
3  $0180, $0f00
4  $0180, $0000
5  $0180, $0f00
6  $0180, $0000
7  $0180, $0f00
8  $0180, $0000
9  $0180, $0f00
10 $0180, $0000

```

Come si può notare, questo pezzo di copper list alterna il rosso e il nero come colore di fondo. Si potrebbe allora accedere alla seconda word di ogni istruzione copper come se fosse un pixel di

uno schermo chunky. La velocità del copper impone però un limite costituito dal numero di pixel di diverso colore visualizzabili e dalla dimensione degli stessi. Infatti, la copper list dell'esempio, cambia il colore del fondo ogni 8 pixel in bassa risoluzione, per cui potremmo ottenere uno schermo chunky di soli 40 pixel di dimensioni 8x1: praticamente inservibile, sia per il ridotto numero di pixel, sia per le dimensioni degli stessi. Possiamo allora provare a modificare anche gli altri registri colore, per è necessario uno schermo che contenga, su ogni riga, qualcosa del genere:

```
Colore0,Colore1,Colore2,Colore3,.....
```

La copper list corrispondente ad ogni riga dello schermo potrà quindi essere:

```
1 $0180, $0rgb
2 $0182, $0rgb
3 $0184, $0rgb
4 $0186, $0rgb
5 .....

```

Purtroppo il copper non riesce ad eseguire più di una cinquantina di istruzioni per riga, per cui il numero di pixel del nostro schermo chunky risulterà essere comunque troppo basso. Questo però significa anche che in due righe, il copper può eseguire un centinaio di istruzioni e, quindi, modificare un numero soddisfacente di registri colore (poco meno di 100). Possiamo quindi aprire uno schermo a 7 bitplane, in cui ogni riga sia, ad esempio, del tipo:

```
Colore0,Colore0,Colore1,Colore1,Colore2,Colore2,...,Colore95,Colore95
```

e scrivere una copper list che, ogni due righe, modifichi il contenuto dei 96 registri colore (ovviamente utilizzando il registro BPLCON3=\$dff106 per modificare il banco dei registri colore). Avremo così realizzato un schermo copper chunky con pixel da 2x2 che però, purtroppo, non funziona ancora nel migliore dei modi. Infatti ogni pixel da 2x2 non appare di un unico colore, proprio perchè il copper non riesce a cambiare i registri colore in maniera sufficientemente veloce. Servirebbe una specie di double-buffering. Per fortuna ci viene in aiuto una caratteristica del chipset AGA costituita dalla possibilità di cambiare l'insieme di colori utilizzati per la visualizzazione. Ogni volta che lo hardware video deve visualizzare un pixel, deve leggerne il colore RGB da uno dei 256 registri colore. Infatti il valore scritto nei bitplane non è altro che un indice nella tabella dei registri colore. Prima di effettuare l'accesso ai registri colore, viene effettuato un OR esclusivo tra il valore letto dai bitplane e il contenuto degli 8 bit alti del registro BPLCON4=\$dff10c. Gli 8 bit alti di BPLCON4 vengono chiamati BPLAMx (dove x = 1-8). E' facile quindi capire che, ponendo BPLAM=\$80, i colori visualizzati saranno quelli da 128 a 255, piuttosto che quelli da 0 a 127.

La copper list, quindi, sarà scritta in maniera da modificare i colori da 0 a 95, mentre sono visualizzati quelli da 128 in poi, e da modificare i colori da 129 a 224, mentre sono visualizzati quelli da 0 in poi:

```
1 $010c,$8000 ;visualizza i colori da 128 a 255
2 $0106,$0020 ;seleziona il primo banco di 32 colori
3 $0180,$0rgb ;modifica il colore del registro 0
4 $0182,$0rgb ;modifica il colore del registro 1
5 $0184,$0rgb ;modifica il colore del registro 2
6 $0186,$0rgb ;modifica il colore del registro 3
7 .....
8 $01be,$0rgb ;modifica il colore del registro 31
9 $0106,$2020 ;seleziona il secondo banco di 32 colori
10 $0180,$0rgb ;modifica il colore del registro 32
11 $0182,$0rgb ;modifica il colore del registro 33
12 $0184,$0rgb ;modifica il colore del registro 34
13 $0186,$0rgb ;modifica il colore del registro 35
14 .....

```



```

15  $01be,$0rgb      ;modifica il colore del registro 63
16  $0106,$020      ;seleziona il terzo banco di 32 colori
17  $0180,$0rgb      ;modifica il colore del registro 64
18  $0182,$0rgb      ;modifica il colore del registro 65
19  $0184,$0rgb      ;modifica il colore del registro 66
20  $0186,$0rgb      ;modifica il colore del registro 67
21  .....
22  $01be,$0rgb      ;modifica il colore del registro 95
23
24  $xx01,$fffe      ;attende la prossima riga da 2 pixel
25  $010c,$0000      ;visualizza i colori da 0 a 127
26  $0106,$020      ;seleziona il quinto banco di 32 colori
27  $0180,$0rgb      ;modifica il colore del registro 128
28  $0182,$0rgb      ;modifica il colore del registro 129
29  $0184,$0rgb      ;modifica il colore del registro 130
30  $0186,$0rgb      ;modifica il colore del registro 131
31  .....
32  $01be,$0rgb      ;modifica il colore del registro 159
33  $0106,$a020      ;seleziona il sesto banco di 32 colori
34  $0180,$0rgb      ;modifica il colore del registro 160
35  $0182,$0rgb      ;modifica il colore del registro 161
36  $0184,$0rgb      ;modifica il colore del registro 162
37  $0186,$0rgb      ;modifica il colore del registro 163
38  .....
39  $01be,$0rgb      ;modifica il colore del registro 191
40  $0106,$e020      ;seleziona il settimo banco di 32 colori
41  $0180,$0rgb      ;modifica il colore del registro 192
42  $0182,$0rgb      ;modifica il colore del registro 193
43  $0184,$0rgb      ;modifica il colore del registro 194
44  $0186,$0rgb      ;modifica il colore del registro 195
45  .....
46  $01be,$0rgb      ;modifica il colore del registro 223

```

Un buon esempio della tecnica del copper chunky è contenuta nel file `chunky.lha` allegato a questo articolo.

CAPITOLO 19

REAL-TIME COMPUTER GRAPHICS 3D

Autore: Cristiano Tagliamonte Aceman/RAMJAM

19.1 Prefazione

Questo breve testo vuol essere di aiuto a chi vuol intraprendere l'arduo ed affascinante cammino verso la programmazione di un motore grafico 3D, scoprendone inizialmente i concetti di base per poi affrontare i più complessi e spettacolari effetti realizzabili. Col termine "motore" si indica l'insieme di routine atte alla gestione e alla manipolazione di specifici dati che una volta elaborati nella maniera dovuta daranno come risultato la visualizzazione dell'ambiente 3D in tempo reale.

Il seguente testo non vuol essere un corso di programmazione orientato alle applicazioni 3D, bensì un'opera nella quale vengano forniti più semplicemente i concetti sui quali si fondano molti motori 3D.

Il tutorial è stato suddiviso in parti, in cui ogni parte è composta da capitoli tra loro indipendenti, ovvero (esclusi alcuni casi in cui certi argomenti sono legati tra loro intrinsecamente) la comprensione di un argomento citato in un determinato capitolo non dipende dagli altri capitoli. In questo modo si rende più efficace ed immediata la consultazione del presente testo. Naturalmente un capitolo potrebbe richiedere la conoscenza di particolari argomenti, pertanto all'inizio di ognuno di essi vengono elencati eventuali titoli dei capitoli in cui vengono fornite le informazioni necessarie affinché sia possibile comprendere appieno gli argomenti trattati. Inoltre, nella lista degli argomenti necessari alla comprensione, potranno essere presenti eventuali sigle "n.p." atte ad indicare che tali argomenti, a causa della loro natura, non sono presenti nel seguente testo.

Nel caso il lettore non abbia le dovute conoscenze del linguaggio C si raccomanda di porgere priorità alla consultazione dell'Appendice A, nella quale vengono forniti chiarimenti riguardo la sintassi dello pseudo-codice utilizzato, ripresa appunto dal C.

Per concludere si raccomanda la conoscenza di base della trigonometria, dell'algebra lineare e di un linguaggio di programmazione (meglio se non troppo evoluto).

19.2 Parte 0: Cenni di grafica 2D relativa agli 80x86 e VGA

Introduzione

In questa sezione non approfondiremo l'utilizzo delle VGA e SVGA, bensì tratteremo le più elementari funzioni atte alla manipolazione di strutture bidimensionali, indispensabili per poter realizzare progetti in computer grafica 3D.

In altre parole verranno date le nozioni minime per poter comprendere ed applicare i concetti sulla computer grafica 3D, senza esporre in maniera peculiare le caratteristiche delle VGA/SVGA, il cui studio richiederebbe una documentazione dedicata, che non è l'obiettivo di questo tutorial. A tal proposito verrà analizzato il solo modo video 320*200 a 256 colori, il più immediato da gestire.

Tutti gli argomenti presentati in questa sezione richiedono espressamente la conoscenza dell'Assembly relativo agli 80x86 presumendo di lavorare in modalità protetta 32 bit con un modello di memoria di tipo flat.

Apertura di uno schermo 320*200 a 256 colori

Analizziamo questa manciata di istruzioni assembly:

```

mov     eax,13h           ; imposta EAX al cui valore corrisponde
                           ; la modalità video da aprire. Nel nostro
                           ; caso 13h ci permette l'apertura di uno
                           ; schermo 320*200 a 256 colori.
int     10h               ; interrupt per la gestione della grafica.

```

Come possiamo vedere è molto semplice utilizzare una risoluzione 320*200 a 256 colori: basta eseguire giusto queste due istruzioni assembly e tale modo video viene aperto!

Nel caso volessimo ripristinare la vecchia modalità testo (standard del dos) non dovremo far altro che seguire queste istruzioni:

```

mov     eax,03h           ; imposta EAX al cui valore corrisponde
                           ; la modalità video da aprire. Nel nostro
                           ; caso 03h ci permette l'apertura di uno
                           ; schermo in modalità testo 80*25.
int     10h               ; interrupt per la gestione della grafica.

```

Questa coppia di istruzioni dovranno essere eseguite appena prima dell'uscita all'MS-DOS del nostro programma.

Organizzazione della memoria video

Uno schermo 320*200 a 256 colori ha come indirizzo fisico di partenza *000A0000h*. A partire da tale indirizzo ogni byte rappresenterà un pixel visualizzato su schermo e di conseguenza, dato che un byte può coprire un range di valori da 0 a 255, potremo utilizzare al massimo 256 colori.

Il byte corrispondente l'indirizzo *000A0000h* è il pixel estremo superiore a sinistra, mentre ai relativi pixel adiacenti corrisponderanno i byte immediatamente successivi a all'indirizzo *000A0000h*. Il 321esimo byte (a partire da *000A0000h*, quindi *000A0140h*) risulta il pixel estremo sinistro della seconda riga. Vediamo un semplice schema per chiarire le idee:

```

+---- 000A0001h           . = ipotetico pixel sul monitor
|+--- 000A0002h
||+-- 000A0003h
vvv

```


L'insieme delle componenti RGB rappresenta il colore che vogliamo ottenere. Una componente R, G o B non è altro che un byte in cui vengono presi in considerazione i 6 bit meno significativi; in altre parole per ogni componente abbiamo a disposizione un range di valori compreso tra 0 e 63, ovvero 64 combinazioni (infatti $64 \times 64 \times 64 = 262144$ ovvero l'intera gamma di colori a disposizione). Un valore basso indica una bassa intensità (tonalità scura), al contrario più il valore sarà alto e maggiore sarà l'intensità di quella componente (tonalità chiara).

Per settare la palette (ovvero la tavolozza di colori che si vuol utilizzare), ci serviranno solo le due porte hardware di indirizzo *03C8h* e *03C9h*. Ecco la procedura da seguire per settare un colore:

- scrivere nella porta *03C8h* un byte, il quale indicherà quale colore andremo a settare (0 è il primo colore, 255 è l'ultimo);
- scrivere nella porta *03C9h* il byte contenente la componente R (rosso);
- scrivere nella porta *03C9h* il byte contenente la componente G (verde);
- scrivere nella porta *03C9h* il byte contenente la componente B (blu).

Da notare che una volta settato un colore si potrà continuare a scrivere nella porta *03C9h* i successivi colori senza dover memorizzare (di volta in volta) il numero del colore successivo nella porta *03C8h*.

Tutto ciò tradotto in assembly porta ad un codice che può essere scritto nella seguente forma:

```

    lea     esi,[palette]    ; puntatore alla tabella dei colori
    mov     edx,03C8h        ; porta 03C8h
    xor     eax,eax          ; puliamo eax
    out     dx,al            ; iniziamo a settare il primo colore
    inc     edx              ; porta 03C9h
    mov     ecx,256          ; numero di iterazioni del loop
@@loop:
    mov     al,[esi]         ; leggiamo la componente Red...
    out     dx,al            ; ... e la memorizziamo
    mov     al,[esi+1]       ; leggiamo la componente Green...
    out     dx,al            ; ... e la memorizziamo
    mov     al,[esi+2]       ; leggiamo la componente Blue...
    out     dx,al            ; ... e la memorizziamo
    add     esi,3            ; prossimo colore da settare
    dec     ecx
    jnz     @@loop
    ...
    ...                    ; altro codice
    ...

palette:
    db      00, 00, 00      ; la tabella dei colori
                                ; colore 00h : R, G, B
    db      11, 23, 45      ; colore 01h : R, G, B
    ...
    ...                    ; altri colori della tavolozza
    ...
    db      63, 63, 63      ; colore FFh : R, G, B

```

Il sorgente appena esaminato non fa altro che leggere i byte nella tabella dei colori a partire dall'indirizzo "palette" e li scrive nella porta *03C9h* non prima di aver precisato nella porta *03C8h* che intendiamo iniziare a settare la palette a partire dal colore *00h*, fino ad arrivare all'ultimo, ovvero il colore *FFh*.

Sincronismo del processore col refresh video

Utilizzando una risoluzione come la 320*200 a 256 colori, lo schermo viene aggiornato 70 volte al secondo (70 Hz). In altre parole quel che si visualizza sul monitor viene tracciato 70 volte al secondo, ciò permette l'illusione di realizzare animazioni fluide. Il refresh non è che il procedimento fisico per cui lo schermo viene aggiornato (illuminando in maniera opportuna ogni pixel).

Mettiamo caso di voler realizzare l'animazione di un oggetto 3D renderizzato in tempo reale. Per rendere il movimento di tale oggetto più fluido possibile dovremo calcolare un intero frame in un 70esimo di secondo. Consideriamo che le routine che abbiamo realizzato siano abbastanza ottimizzate da permettere che l'elaboratore calcoli oltre 70 fotogrammi per secondo. In questo caso dovremo scrivere una routine che, una volta renderizzato un frame, permetta al processore di attendere il termine del refresh video prima di calcolare il prossimo fotogramma. Questo è ciò che si intende con "sincronizzazione del processore col refresh video".

Ora vediamo come realizzare il tutto in Assembly evitando di analizzare dettagliatamente il codice:

```

        mov     edx,3DAh
@@wait_refresh1:
        in      al,dx
        test    al,8
        jz      @@wait_refresh1
@@wait_refresh2:
        in      al,dx
        test    al,8
        jnz     @@wait_refresh2

```

In questo spezzone di codice il processore non fa altro che attendere (grazie al primo loop) che il pennello elettronico (che aggiorna lo schermo) giunga ad una ben determinata linea video, quindi (nel secondo loop) si aspetta che tale linea venga completamente tracciata.

Questa tecnica viene applicata per evitare di calcolare un nuovo fotogramma ancor prima di averlo visualizzato.

19.3 Parte 1: Geometry Engine

Introduzione

In questa sezione studieremo la geometria che si cela dietro la computer grafica 3D, in quanto ogni elemento elaborato da un motore 3D consiste in un insieme di valori numerici che attribuiscono una o più proprietà geometriche, quindi il movimento compiuto da un solido proiettato su monitor è frutto di opportune trasformazioni geometriche.

Trasformazioni prospettiche

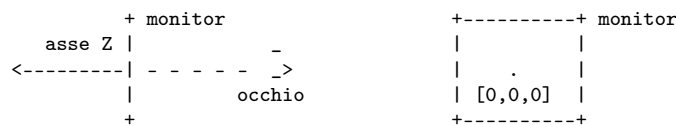
Concetto di prospettiva

Riguardo le nostre applicazioni (relative alla realizzazione di un motore 3d), alla base della prospettiva risiede il principio con cui una qualsiasi entità (punto, linea, poligono o oggetto che sia) da uno spazio tridimensionale possa essere rappresentata in uno spazio a due dimensioni. Difatti uno dei primissimi problemi che intercorre nella scrittura di un engine 3d è come visualizzare tali entità su uno spazio 2d, ossia il monitor, quando all'origine ogni cosa viene rappresentata matematicamente in uno spazio 3d. Per far fronte a questa esigenza viene in aiuto la prospettiva che ci permette di applicare la "conversione" da uno spazio 3d ad uno spazio 2d.

Innanzitutto va sottolineato che lo studio di trasformazioni prospettiche sarà limitato ai soli vettori. Infatti ogni entità è derivabile da un insieme ordinato di uno o più vettori (un punto è rappresentabile con un vettore; una linea si può rappresentare matematicamente come quella coppia di vettori coincidenti con gli estremi della linea stessa; un poligono è una sequenza ordinata di vertici, i quali possono essere indicati con vettori; infine un oggetto è un insieme ben definito di poligoni).

Dati un punto di vista e un piano di proiezione perpendicolare alla direzione di osservazione e posto di fronte al suddetto punto vista, il concetto di prospettiva è di tracciare un raggio passante sul punto di vista e sul vettore 3d di cui si vuol conoscere la proiezione sul piano (questo avviene esclusivamente a livello teorico, a livello pratico non si traccia nessun raggio!); quindi il vettore coincidente con il punto di intersezione tra il raggio ed il piano, rappresenterà la proiezione di quel vettore.

Dato un ipotetico punto di vista e un piano di proiezione (rispetto ai quali dovremo ricavare le coordinate 2d), consideriamo che la direzione di osservazione sia perpendicolare al nostro piano e che tale piano coincida con il piano XY del nostro spazio 3d (ovvero il piano di equazione $z = 0$). Inoltre la retta coincidente con la direzione di osservazione passa per l'origine dei tre assi del nostro sistema di riferimento XYZ. Questo implica che tale retta coincide con l'asse z, sul quale sarà quindi posto il punto di vista. Il monitor è inteso come quella parte (ovvero quel sottospazio) di piano visibile. Vediamo praticamente la nostra situazione:

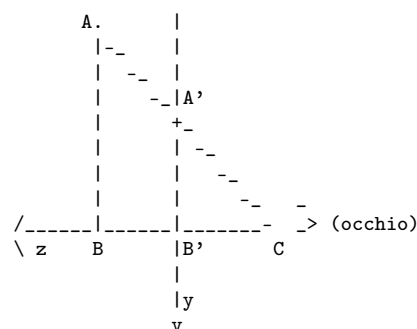


Questa situazione è molto importante, in quanto permette di semplificare concettualmente e algebricamente le trasformazioni prospettiche.

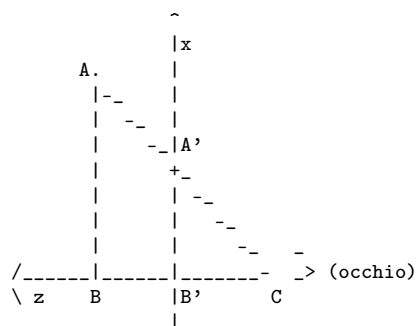
Implementazione della prospettiva

Abbiamo un vettore di cui conosciamo le coordinate $[x, y, z]$. Il nostro obiettivo consiste nel calcolare le coordinate $[xs, ys]$ proiettate sullo schermo, quindi dobbiamo applicare una trasformazione prospettica.

Consideriamo un punto, denominato A , nello spazio visibile dal video. Facciamo finta che il nostro monitor sia trasparente e che quel punto sia davvero presente nello spazio. Ora, la nostra situazione vista lateralmente è la seguente:



Mentre questo è il caso analogo visto dall'alto:



A = vettore nello spazio $[x, y, z]$ rappresentante il punto a cui si vuol applicare la trasformazione prospettica;
 B = vettore avente la stessa coordinata z del vettore A , ma con coordinate x e y uguali a zero, è del tipo $[0, 0, z]$;
 C = vettore coincidente con il punto vista, per comodità è posto sull'asse z (quindi ha le coordinate $x, y = 0$);
 A' = vettore coincidente con la proiezione su video del vettore A , è della forma $[xs, ys, 0]$;
 B' = vettore avente la stessa coordinata z del vettore A' , ma con coordinate x e y uguali a zero, è esattamente $[0, 0, 0]$, ciò significa che B' coincide con l'origine degli assi;

Si ricorda, come precedentemente definito, che il piano di proiezione è quello di equazione $z = 0$, ovvero il piano XY , su cui coincide, appunto, il monitor. Quindi sarà chiaro perché le coordinate $[xs, ys]$ sono esattamente le coordinate 2d rappresentanti il nostro vettore 3d su schermo.

Vediamo adesso come ricavare queste coordinate. Sia nella vista dall'alto che in quella laterale i triangoli ABC e $A'B'C$ sono simili in quanto hanno un angolo in comune ed entrambi hanno un angolo retto, quindi possiamo scrivere la seguente proporzione:

$$A'B' / AB = B'C / BC$$

che corrisponde a:

$$A'B' = AB * B'C / BC$$

Possiamo scrivere BC come $BB' + B'C$. Dato che B' coincide con l'origine, la distanza BB' equivale alla coordinata z del vettore B . Inoltre definiamo d come la lunghezza del lato $B'C$. La nostra formula si traduce nella seguente:

$$A'B' = d * AB / (z + d)$$

Rispetto alla vista laterale AB coincide con la coordinata y di A , mentre $A'B'$ coincide con ys ; sostituendo questi valori otteniamo proprio la formula per calcolare la y proiettata su schermo:

$$ys = d * y / (z + d)$$

Nel caso della vista dall'alto il discorso è analogo, ma relativo alla coordinata x proiettata:

$$xs = d * x / (z + d)$$

Il valore d rappresenta la distanza tra il punto di vista e l'origine; e come tale può essere fissato arbitrariamente. Un buon valore con cui istanziare d è, ad esempio, 256.

Bisogna tener conto che nel video le coordinate hanno come origine il punto in alto a sinistra, mentre per la trasformazione 3d->2d sono state considerate al centro del monitor. Per ovviare questo problema basta sommare, al termine dei calcoli per la proiezione, una costante a x_s e a y_s grazie alle quali i relativi assi verranno traslati di un numero di pixel equivalente alla costante. Riassumendo, ponendo $d = 256$, le coordinate proiettate sullo schermo sono equivalenti a:

```
xc = larghezza schermo/2, per traslare l'ascissa a destra
yc = altezza schermo/2, per traslare l'ordinata in basso

xs = 256 * x / (z + 256) + xc
ys = 256 * y / (z + 256) + yc
```

Attenzione al fatto che la coordinata z del punto non può coincidere con il punto di vista dato che si verificherebbe una divisione per zero. Nemmeno si deve porre la profondità di un punto inferiore a quella del punto di vista: sarebbe assurdo poter visualizzare punti posti dietro l'osservatore!

Finalmente siamo in grado di svolgere trasformazioni prospettiche, che tuttavia risultano veramente semplici da applicare.

Traslazione

Applicazione delle traslazioni

Dato un vettore V che rappresenta un punto nello spazio, traslare V rispetto ad un vettore T significa trovare un ulteriore vettore, chiamiamolo W , che rappresenta il vettore V spostato, in direzione coincidente con il verso di T , di uno spazio equivalente alla lunghezza di T .

Tutto ciò si traduce in una banalissima somma tra vettori, più precisamente tra il vettore V e T , quindi:

```
V[xv, yv, zv] = vettore da traslare
T[tx, ty, tz] = vettore coincidente la traslazione da applicare
W[xw, yw, zw] = vettore coincidente V traslato rispetto a T

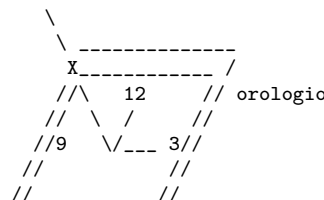
W = V + T

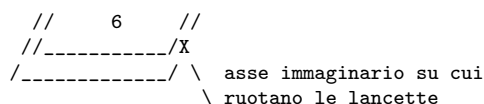
xw = xv + tx
yw = yv + ty
zw = zv + tz
```

Rotazione

Concetto di rotazione

Consideriamo un orologio analogico, le sue lancette ruotano intorno l'asse passante per il centro dell'orologio e perpendicolare l'orologio stesso:





Tecnicamente parliamo di rotazione di un punto su di un asse, quando il punto si muove sul piano appartenente al punto stesso e perpendicolare all'asse in modo da non alterare la distanza punto-asse (che rimane costante). In questo modo il punto compie un movimento circolare intorno l'asse, ossia ci ruota intorno, e la distanza punto-asse funge da raggio per la circonferenza descritta dalla rotazione del punto in questione.

La rotazione intorno l'asse y si può immaginare come la traiettoria percorsa da un punto che giri intorno l'asse y senza modificare la propria ordinata, che rimane appunto inalterata.

19.4 Appendici

Appendice D: coordinate polari

Introduzione

Le coordinate polari vengono utilizzate in questo tutorial al fine di comprendere il principio che sta alla base dell'applicazione delle rotazioni.

Appendice G: cenni di algebra lineare

Introduzione

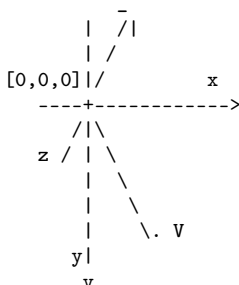
La seguente appendice è orientata a tutti coloro che non possiedono le nozioni di algebra lineare necessarie alla comprensione del testo ed in particolare della parte 1, relativa al geometry engine.

I vettori

Argomenti necessari alla comprensione: cenni di geometria piana (n.p.)

Definizione di un vettore

Un vettore non è altro che un oggetto matematico che rappresenta una quantità di valore con una direzione e un verso, o in termini spicci una linea. Nel contesto verranno sempre specificati vettori che vanno da un punto di coordinate $[0, 0, 0]$ (l'origine degli assi) verso un altro punto $[x, y, z]$, così facendo si può affermare che questo vettore ha come quantità $[x, y, z]$.



Un vettore viene indicato con una lettera, ad esempio nella precedente figura è rappresentato il vettore $V[x, y, z]$. Il sistema di riferimento usato è dato da tre variabili, immaginabile come un semplice piano cartesiano perpendicolare all'asse z . Per semplicità consideriamo la crescita della y verso il basso (e non verso l'alto) in modo da ridurre i calcoli da svolgere per la visualizzazione di un punto sullo schermo. L'asse z cresce quando esso si allontana dall'osservatore, mentre diminuisce nel caso si avvicini all'osservatore.

Utilizzeremo i vettori per definire ogni punto nello spazio (e in particolare i vertici di ogni oggetto 3d), ciò non significa in termini pratici che un vettore costituisca una linea visualizzata su schermo, bensì indica il segmento immaginario compreso tra l'origine degli assi ed un punto nello spazio.

Operazioni tra vettori

Innanzitutto va specificato che faremo uso fondamentalmente di vettori 2d e 3d in quanto il nostro motore gestirà punti nello spazio piano (per il tracciamento di poligoni su monitor) e tridimensionale (per eventuali trasformazioni geometriche), quindi vettori del tipo:

```
v2 = [ x , y ]      <--- generico vettore 2d
v3 = [ x , y , z ]  <--- generico vettore 3d
```

Tutte le operazioni tra vettori sono possibili solo tra vettori che hanno la stessa dimensione, quindi una qualsiasi operazione tra un vettore 2d ed uno 3d sono impossibili. Di seguito vediamo algebricamente quali sono queste operazioni:

ADDIZIONE il risultato della somma di due vettori è ancora un vettore della stessa dimensione dei due operandi:

```
v1 = [ x1 , y1 , z1 ]
v2 = [ x2 , y2 , z2 ]

v3 = v1 + v2 = [ x1+x2 , y1+y2 , z1+z2 ]
```

dove nel nostro caso $x1$, $x2$, $y1$, $y2$, $z1$ e $z2$ sono ovviamente numeri reali (che possono essere rappresentati come numeri in virgola fissa oppure preferibilmente come numeri in virgola mobile).

Da notare che il vettore $[0, 0, 0]$ coincide con l'operando nullo di questa operazione, ovvero addizionando tale vettore ad un qualsiasi vettore V otterremo come risultato sempre il vettore V . Inoltre tale operazione gode della proprietà commutativa.

Fisicamente è possibile immaginare la somma di due vettori 2d nella seguente maniera:

```
v1 = [ x1 , y1 ]
v2 = [ x2 , y2 ]
```

Ora immaginiamo di trovarci sul vettore $v1$. Quindi il risultato della somma $v1 + v2$ si può intendere come il risultato dello spostamento (dalla posizione indicata da $v1$) di $x2$ posizioni orizzontali e $y2$ posizioni verticali, questo implica la nostra posizione finale coinciderà col vettore $[x1 + x2, y1 + y2]$, ovvero il risultato dell'addizione.

NEGAZIONE il risultato della negazione di un vettore $v1$ è ancora un vettore della stessa dimensione di $v1$ e che è equidistante dall'origine rispetto a $v1$, ma è di verso opposto (in altre parole è simmetrico rispetto alla retta passante per l'origine, la quale è perpendicolare al vettore $v1$):

$$\begin{aligned} v1 &= [x, y, z] \\ v2 &= -v1 = [-x, -y, -z] \end{aligned}$$

Anche in questo caso l'elemento nullo di questa operazione coincide con il vettore dell'origine $[0, 0, 0]$.

PRODOTTO CON UN FATTORE il prodotto tra un vettore $v1$ ed un fattore (inteso come un numero reale o in virgola mobile) è ancora un vettore delle stesse dimensioni di $v1$:

$$\begin{aligned} v1 &= [x, y, z] \\ k &= \text{numero reale} \\ v2 &= k * v1 = [k*x, k*y, k*z] \end{aligned}$$

Il fattore k viene chiamato “scalare” (da non confondere con il prodotto scalare!), ed il vettore risultante $v2$ conserverà lo stesso verso di $v1$ ma la sua dimensione verrà scalata rispetto al valore di k . Questa operazione è commutativa.

PRODOTTO SCALARE il risultato prodotto scalare tra due vettori è uno scalare (quindi un numero reale o in virgola mobile):

$$\begin{aligned} v1 &= [x1, y1, z1] \\ v2 &= [x2, y2, z2] \\ k &= v1 * v2 = x1*x2 + y1*y2 + z1*z2 \end{aligned}$$

Geometricamente il prodotto scalare può essere definito come il prodotto tra le distanze dei due vettori con l'origine ed il coseno dell'angolo compreso tra i due vettori. Anche questa operazione gode della proprietà commutativa.

PRODOTTO IN CROCE il risultato del prodotto in croce tra due vettori è uno scalare (quindi un numero reale o in virgola mobile):

$$\begin{aligned} v1 &= [x1, y1] \\ v2 &= [x2, y2] \\ k &= v1 \times v2 = x1*y2 - x2*y1 \end{aligned}$$

Per semplicità citiamo solo la formula del prodotto in croce tra due vettori 2d dato che utilizzeremo questa operazione solo su tali vettori. Il prodotto in croce non gode della proprietà commutativa.

Vettore unitario

Un vettore unitario è un vettore cui lunghezza pari all'unità. La lunghezza di un vettore è data dalla distanza del punto di coordinate specificate dal vettore stesso con l'origine.

La lunghezza di un generico vettore si ricava dal teorema di Pitagora svolgendo la radice quadrata della sommatoria dei quadrati delle componenti del vettore, più semplicemente:

$$\begin{aligned} v2 &= [x2, y2] \\ v3 &= [x3, y3, z3] \\ l2 &= |v2| = \sqrt{x2*x2 + y2*y2} \\ l3 &= |v3| = \sqrt{x3*x3 + y3*y3 + z3*z3} \end{aligned}$$

Rendere un generico vettore v_1 come vettore unitario significa trovare quel vettore che conserva lo stesso verso di v_1 ma che ha lunghezza pari ad 1. Nella pratica si traduce nel dividere ogni componente del vettore per la relativa lunghezza; in relazione al precedente esempio possiamo affermare:

$$\begin{aligned} u_2 &= [x_2/12, y_2/12] \\ u_3 &= [x_3/13, y_3/13, z_3/13] \end{aligned}$$

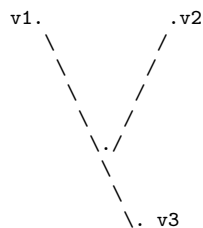
Che coincide col moltiplicare il vettore per uno scalare equivalente all'inverso della lunghezza del vettore:

$$\begin{aligned} v &= [x, y, z] \\ k &= 1 / |v| = 1 / \sqrt{x*x + y*y + z*z} \\ u &= k * v \end{aligned}$$

Vettore normale

Un vettore 3d normale su di un piano (quindi su uno spazio bidimensionale) significa che la retta passante per tale vettore è perpendicolare a quel piano. Un vettore normale su una retta r significa che la retta passante per tale vettore è perpendicolare alla retta r . Un vettore normale ad altro vettore implica che il primo vettore sia normale alla retta passante per il secondo vettore.

Vediamo un esempio di vettore normale ad un altro vettore:



In questo caso v_1 e v_3 sono vettori normali rispetto a v_2 , mentre quest'ultimo è vettore normale sia rispetto a v_1 che a v_3 . Infatti i vettori v_1 e v_3 giacciono sulla stessa retta.

Le matrici

Argomenti necessari alla comprensione: i vettori.

Definizione di matrice

Una matrice è una tabella di elementi (che nel nostro caso sono numeri razionali) del tipo:

$$M(n,m) = \begin{bmatrix} a(1,1) & a(1,2) & \dots & a(1,m) \\ a(2,1) & a(2,2) & \dots & a(2,m) \\ \dots & & & \\ \dots & & & \\ a(n,1) & a(n,2) & \dots & a(n,m) \end{bmatrix}$$

dove $a(i, j)$ è il generico elemento della matrice mentre $n * m$ rappresenta la dimensione della matrice, in cui valgono le seguenti regole:

$$\begin{aligned} n, m &> 0 \\ 0 < i &\leq n \\ 0 < j &\leq m \end{aligned}$$

n, m, i, j sono quindi numeri interi maggiori di zero. Il valore n è il numero di righe della matrice, mentre m è il numero di colonne. I valori i e j rappresentano rispettivamente gli indici di riga e di colonna di un elemento della matrice.

Nel nostro caso intendiamo ogni elemento $a(i, j)$ come un numero reale o un numero in virgola mobile.

Per semplicità si farà spesso riferimento ad una matrice denotando solo la lettera che la etichetta. Ovvero spesso si utilizzerà spesso la notazione M al posto di $M(n, m)$.

Casi particolari di matrici:

MATRICE QUADRATA $M(n, n)$ il numero di righe e di colonne sono uguali. Esempio:

$$M(n, n) = \begin{vmatrix} - & - \\ | 1 & 2 & 3 | \\ | 4 & 5 & 6 | \\ | 7 & 8 & 9 | \\ | - & - & - | \end{vmatrix}$$

La **DIAGONALE** principale della matrice quadrata è definita come quell'insieme di elementi della matrice stessa tali che gli indici di riga e colonna siano uguali, quindi:

$$D = \text{tutti gli } a(i, i) \text{ per } 0 < i \leq n$$

ovvero:

$$D = \{ a(1, 1), a(2, 2), \dots, a(n, n) \}$$

nel precedente esempio la diagonale principale è uguale a:

$$D = \{ 1, 5, 9 \}$$

MATRICE TRASPOSTA $Mt(n, n)$ data una matrice quadrata $M(n, n)$, la sua trasposta $Mt(n, n)$ è quella matrice che contiene gli stessi elementi della M , ma con diverso ordine. Tale ordine è determinato dagli indici di riga e colonna, i quali vengono scambiati. Più formalmente il generico elemento $a(i, j)$ appartenente a M coincide con l'elemento $t(i, j)$ di Mt tale che:

$$\begin{aligned} t(i, j) &= a(j, i) \\ 0 < i &\leq n \\ 0 < j &\leq n \end{aligned}$$

Per chiarire le idee vediamo un esempio pratico relativo ad una matrice di dimensione 3*3:

$$M = \begin{vmatrix} - & - \\ | 1 & 2 & 3 | \\ | 4 & 5 & 6 | \\ | 7 & 8 & 9 | \\ | - & - & - | \end{vmatrix} \quad \text{la sua trasposta è} \quad Mt = \begin{vmatrix} - & - \\ | 1 & 4 & 7 | \\ | 2 & 5 & 8 | \\ | 3 & 6 & 9 | \\ | - & - & - | \end{vmatrix}$$

Da notare che gli elementi posti sulla diagonale principale rimangono nella posizione originaria.

MATRICE RIGA $M(1, m)$ una matrice riga è coincide, a livello logico, con la definizione di vettore, quindi tale matrice è intesa come una matrice con una sola riga, quindi nella forma:

$$M(1, m) = [a(1, 1) \quad a(1, 2) \quad \dots \quad a(1, m)]$$

o più semplicemente:

$$M(m) = [a(1) \quad a(2) \quad \dots \quad a(m)]$$

Le ultime due definizioni di matrice per righe sono logicamente equivalenti, la differenza sostanziale sta nel secondo caso, in cui non si fa specificatamente riferimento alla notazione matriciale, la quale implica che si faccia riferimento ad entrambi gli indici di riga e colonna (infatti abbiamo il solo indice di colonna).

Etichettando gli elementi di una matrice per riga senza il relativo indice si ottiene un oggetto matematico equivalente alla matrice per riga che viene denominato *ennupla*. Vediamone un esempio:

$$E = (a, b, c, d)$$

MATRICE COLONNA $M(n, 1)$ è un modo alternativo (rispetto alla matrice riga) di rappresentare un vettore secondo la notazione matriciale, ovvero:

$$M(n, 1) = \begin{bmatrix} a(1, 1) \\ a(2, 1) \\ \dots \\ \dots \\ a(n, 1) \end{bmatrix}$$

o più semplicemente:

$$M(n) = \begin{bmatrix} a(1) \\ a(2) \\ \dots \\ \dots \\ a(n) \end{bmatrix}$$

MATRICE NULLA $M(n, m)$ è quella matrice in cui tutti gli elementi sono uguali a 0, ovvero:

$$\begin{aligned} a(i, j) &= 0 \\ 0 &< i <= n \\ 0 &< j <= m \end{aligned}$$

MATRICE IDENTITÀ $I(n)$ è un particolare caso di matrice quadrata in cui tutti gli elementi sono uguali a 0 fatta esclusione per quegli elementi presenti sulla diagonale principale che sono uguali a 1. O in termini più formali:


```

0 < i <= n
0 < j <= n
a(i,j) = 0   per i diverso da j
a(i,j) = 1   per i uguale a j

```

Faremo riferimento alla generica matrice identità $n * n$ con la notazione $I(n)$ (basta specificare un solo indice in quanto la matrice identità è sempre quadrata, quindi l'indice di colonna è sottointeso). Vediamo un esempio relativo alla matrice identità $4*4$:

```

      - -
      |   |
I(4) = | 1  0  0  0 |
      | 0  1  0  0 |
      | 0  0  1  0 |
      | 0  0  0  1 |
      - -

```

Operazioni tra matrici

In relazione all'obiettivo di realizzare un motore 3d possiamo anticipare che principalmente ci occuperemo di applicare operazioni tra matrici $3*3$, $4*3$ e $4*4$.

SOMMA la somma tra due matrici A e B è applicabile se e solo se A e B hanno lo stesso numero di righe e colonne. Il risultato è ancora una matrice, che per convenzione chiamiamo C , delle stesse dimensione degli operandi. Ogni elemento di C è la somma del corrispondente elemento di A e di B (che quindi si trovano nella stessa posizione dell'elemento di C):

```

A(n,m) = matrice cui generico elemento è a(i,j)
B(n,m) = matrice cui generico elemento è b(i,j)
C(n,m) = matrice cui generico elemento è c(i,j)

```

```

c(i,j) = a(i,j) + b(i,j)
0 < i <= n
0 < j <= m

```

```

      - -
      |   |
C = A + B = | a(1,1)+b(1,1) a(1,2)+b(1,2) ... a(1,m)+b(1,m) |
      | a(2,1)+b(2,1) a(2,2)+b(2,2) ... a(2,m)+b(2,m) |
      | ... |
      | ... |
      | a(n,1)+b(n,1) a(n,2)+b(n,2) ... a(n,m)+b(n,m) |
      - -

```

Questa operazione gode delle seguenti proprietà :

```

commutativa => A + B = B + A
associativa => (A + B) + C = A + (B + C)

```

NEGAZIONE la negazione di una matrice $A(n, m)$ è una matrice di dimensione ancora $n*m$ in cui ogni elemento di A è di segno opposto:

```

A(n,m) = matrice cui elementi sono a(i,j)
B(n,m) = matrice cui elementi sono b(i,j)

```

```

B = - A

```

```

b(i,j) = - a(i,j)
0 < i <= n
0 < j <= m

```

PRODOTTO PER UNO SCALARE svolgere il prodotto di una matrice A per uno scalare k significa moltiplicare ogni elemento di A con k , il quale non è altro che un numero reale. La matrice così ottenuta è ovviamente delle stesse dimensioni della matrice originaria:

```

k          = numero reale
A(n,m) = matrice cui elementi sono a(i,j)
B(n,m) = matrice cui elementi sono b(i,j)

B = k * A

b(i,j) = k * a(i,j)
0 < i <= n
0 < j <= m

```

PRODOTTO TRA MATRICI il prodotto tra due matrici A e B è applicabile se il numero di colonne di A è uguale al numero di righe di B . Il risultato sarà una matrice C con un numero di righe equivalente a quello di A ed un numero di colonne uguale al corrispondente di B . Ogni elemento $c(i,j)$ di C coincide con il prodotto scalare dei due vettori intesi come la i -esima riga di A e la j -esima colonna di B :

```

A(n,t) = matrice cui generico elemento è a(i,k)
B(t,m) = matrice cui generico elemento è b(k,j)
C(n,m) = matrice cui generico elemento è c(i,j)

C = A * B

c(i,j) = a(i,1)*b(1,j) + a(i,2)*b(2,j) + ... + a(i,t)*b(t,j)
0 < i <= n
0 < j <= m

```

che equivale a:

```

c(i,j) = a(i,k)*b(k,j)
0 < k <= t
0 < i <= n
0 < j <= m

```

Questa operazione gode della proprietà associativa, ovvero:

$$(A * B) * C = A * (B * C)$$

che, come vedremo in seguito, risulterà particolarmente utile per la progettazione del geometry engine. Inoltre il prodotto NON gode della proprietà commutativa. Da notare che l'elemento neutro di questa operazione è esattamente la matrice identità :

$$A(n,m) = I(n) * A(n,m) = A(n,m) * I(m)$$

Inoltre il prodotto per la matrice identità è l'unico caso in cui vale la proprietà commutativa.

19.5 Note finali

Spesso vagheggiano nella mia mente pensieri del tipo “Ne vale davvero la pena consumare tanto tempo per realizzare un motore 3d?”, oppure “Ora c'è da fare altro di più prioritario che starsene qui a scrivere questo engine”. . . , dubbi che affliggono molte di quelle persone che si propongono di programmare un motore 3d, in particolare chi è alle prime armi oppure chi ne ha avuto a che fare per molto (forse troppo) tempo.

Tristo è quel discepolo che non avanza il suo maestro. Leonardo.
AceMan

19.6 Rotazione

Adesso vediamo realmente come effettuare rotazioni. Utilizzando come sistema di riferimento un piano a 2 dimensioni è possibile convertire le coordinate cartesiane (x,y) di un punto in coordinate polari (r,t) :

	$V(x,y)=V'(r,t)$ $r=\sqrt{x^2+y^2}$ $t=\arctan(y/x)$ $x=r\cos(t)$ $y=r\sin(t)$		$r=\text{distanza}$ <p>punto-origine</p> $t=\text{angolo compreso}$ <p>tra il vettore</p> <p>ed il semiasse</p> <p>positivo x</p>
--	--	--	--

Dopodiché per ruotare il punto intorno l'origine si dovrà sommare l'angolo di cui si vuol ruotare il punto alla variabile t e convertire le risultanti coordinate polari in cartesiane. Questo metodo risulta troppo lento per applicazioni in real time in quanto sono presenti quadrati, arcotangenti e radici quadrate; inoltre introducendo la variabile z si andrebbe incontro ad una pesante gestione delle coordinate polari (utilizzo di integrali tripli, ecc.): meglio trovare una soluzione più semplice e veloce. Immaginiamo di avere un vettore con ordinata nulla $V(x,0)$ e vogliamo ruotarlo di un angolo a :

	<p>vettore</p> <p>coincidente</p> <p>l'ascissa</p> $V(x,0)$		<p>vettore</p> <p>ruotato</p> <p>di a</p> <p>radianti</p>
--	--	--	---

Se volessimo trasformare in polari le coordinate di V risulterebbe $r = x$ ($x = \sqrt{x^2+0^2}$) e $t = 0$ ($0 = \arctan(0/x)$). Per ruotare il vettore basta sommare a t l'angolo di rotazione a . Quindi:

```
(V = generico vettore in coordinate cartesiane)
(V' = generico vettore in coordinate polari)
(Vr = vettore ruotato di a radianti in coordinate cartesiane)
(Vr' = vettore ruotato di a radianti in coordinate polari)
(Vxr = vettore ruotato con la sola componente x in coordinate cartesiane)
(Vyr = vettore ruotato con la sola componente y in coordinate cartesiane)

V(x,y) = V'(r,t)
Vr(xr,yr) = Vr'(r,t+a) -> xr=r*cos(t+a) yr=r*sin(t+a)
Andiamo a sostituire xr e yr con le relative formule:
Vr(r*cos(t+a),r*sin(t+a)) -> r=x t=0
Andiamo a sostituire r con x e t con 0:
Vxr(x*cos(a),x*sin(a))
```

E questa è la formula per la rotazione di un vettore che non ha la componente y . Ora consideriamo un vettore che ha l'ascissa nulla $V(0,y)$:

```
r=y (=sqrt(0^2+y^2))
t=pi/2 (=arctan(y/0)=arctan(infinito))
Vyr(y*cos(pi/2+a),y*sin(pi/2+a))
```

Un paio di formule trigonometriche ci dicono:

```
cos(pi/2+a)=-sin(a)
sin(pi/2+a)=cos(pi/2)
```

ma siccome utilizziamo un sistema di riferimento con l'asse y "rovesciato" dobbiamo cambiare un segno ad entrambe le formule per poterle sfruttare, che quindi diventano:

$$\begin{aligned}\cos(\pi/2+a) &= \sin(a) \\ \sin(\pi/2+a) &= -\cos(a)\end{aligned}$$

Adesso la formula per ruotare un vettore senza componente x è uguale:

$$V_{yr}(y*\sin(a), -y*\cos(a))$$

Ma se guardiamo al caso generale, abbiamo un vettore V che ha entrambe le componenti x e y . Difatti un generico vettore con le componenti x e y è uguale a:

$$V_1(x,0)+V_2(0,y)=V(x+0,0+y)=V(x,y)$$

Ora possiamo usare le formule di rotazione dei singoli casi per calcolare il caso generale con un'addizione tra vettori:

$$\begin{aligned}& V_{xr}(x*\cos(a), x*\sin(a)) + \\ & V_{yr}(y*\sin(a), -y*\cos(a)) = \\ & \text{-----} \\ & V_r(x*\cos(a)+y*\sin(a), x*\sin(a)-y*\cos(a))\end{aligned}$$

Grazie a questa formula è possibile ruotare un qualsiasi vettore su uno spazio bidimensionale. In un ambiente 3D la formula appena descritta coincide con la rotazione intorno l'asse z (non c'è nessuna coordinata z cambiata). Per ruotare il punto intorno un altro asse basta lasciar fuori la relativa variabile e utilizzare le altre nella precedente espressione, il che si può sintetizzare con:

intorno l'asse z	intorno l'asse y	intorno l'asse x
-----	-----	-----
$xr=x*\cos(a)+y*\sin(a)$	$xr=x*\cos(a)+z*\sin(a)$	$yr=y*\cos(a)+z*\sin(a)$
$yr=x*\sin(a)-y*\cos(a)$	$zr=x*\sin(a)-z*\cos(a)$	$zr=y*\sin(a)-z*\cos(a)$

19.7 Ottimizzazione delle rotazioni

Dato un angolo di rotazione per ogni asse (x, y, z) con le precedenti formule sarebbero occorse ben 12 moltiplicazioni per poter ruotare un solo punto. Qui vedremo come effettuare rotazioni eseguendo 9 moltiplicazioni per ogni punto. Consideriamo:

ax = angolo di rotazione intorno l'asse x	$s1 = \sin(ax)$	$c1 = \cos(ax)$
ay = angolo di rotazione intorno l'asse y	$s2 = \sin(ay)$	$c2 = \cos(ay)$
az = angolo di rotazione intorno l'asse z	$s3 = \sin(az)$	$c3 = \cos(az)$

Ognuna delle variabili x, y, z influenza le rotazioni intorno a due assi (nella rotazione intorno al proprio asse la variabile rimane inalterata), possiamo così indicare con $x'y'ez'$ le variabili parzialmente ruotate (cioè dopo la prima rotazione) e con $x''y''ez''$ le variabili completamente ruotate. Detto ciò le formule viste in precedenza corrispondono a:

$$\begin{aligned}x' &= x*c1 + y*s1 \\ y' &= x*s1 - y*c1 \\ \\ x'' &= x'*c2 + z*s2 <- \text{coordinata } x \text{ ruotata completamente} \\ z' &= x'*s2 - z*c2 \\ \\ y'' &= y'*c3 + z'*s3 <- \text{coordinata } y \text{ ruotata completamente} \\ z'' &= y'*s3 - z'*c3 <- \text{coordinata } z \text{ ruotata completamente}\end{aligned}$$

che equivale a scrivere:

```
x'' = (x*c1+y*s1)*c2+z*s2=c2*c1 *x + c2*s1 *y + s2 *z

y'' = (x*s1-y*c1)*c3+((x*c1+y*s1)*s2-z*c2)*s3=
c3*s1 *x - c3*c1 *y + s3*s2*c1 *x + s3*s2*s1 *y - s3*c2 *z=
(s3*s2*c1+c3*s1) *x + (s3*s2*s1-c3*c1) *y + (-s3*c2) *z

z'' = (x*s1-y*c1)*s3-((x*c1+y*s1)*s2-z*c2)*c3=
s3*s1 *x - s3*c1 *y - c3*s2*c1 *x - c3*s2*s1 *y + c3*c2 *z=
(-c3*s2*c1+s3*s1) *x + (-c3*s2*s1-c3*c1) *y + (c3*c2) *z

z'' = (x*s1-y*c1)*s3-((x*c1+y*s1)*s2-z*c2)*c3=
s3*s1 *x - s3*c1 *y - c3*s2*c1 *x - c3*s2*s1 *y + c3*c2 *z=
(-c3*s2*c1+s3*s1) *x + (-c3*s2*s1-s3*c1) *y + (c3*c2) *z
~~~~~ zx ~~~~~ zy ~~~~~ zz
```

Dall'ultimo passaggio di ognuna di queste formule si può notare che non vengono calcolate coordinate parzialmente ruotate e che ogni coordinata ruotata equivale alla somma delle variabili (non ruotate) moltiplicate per un determinato fattore. Se precalcoliamo questi fattori potremo utilizzarli per tutti quei punti che devono essere ruotati nella stessa direzione. In questo modo avremo svolto semplicemente 9 moltiplicazioni per ogni punto (esclusi i precalcoli per i fattori). In sostanza dobbiamo calcolare prima queste costanti:

```
xx = c2*c1
xy = c2*s1
xz = s2
yx = c3*s1+s3*s2*c1
yy = -c3*c1+s3*s2*s1
yz = -s3*c2
zx = s3*s1-c3*s2*c1 = s2*c1+c3*s1
zy = -s3*c1-s3*s2*s1 = c3*c1-s2*s1
zz = c3*c2
```

poi per ogni punto si devono svolgere questi calcoli (sfruttando gli stessi fattori):

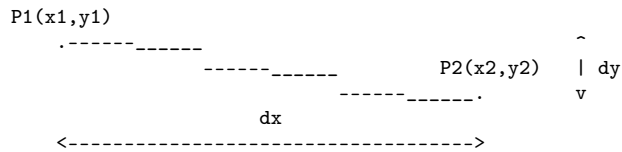
```
x'' = xx * x + xy * y + xz * z
y'' = yx * x + yy * y + yz * z
z'' = zx * x + zy * y + zz * z
```

Ed otterremo le tre coordinate ruotate.

Questo algoritmo risulterebbe meno efficiente del precedente se si utilizzassero pochi punti, mentre nel caso di una elevata quantità di vettori è possibile raggiungere notevoli risparmi in termini di calcolo.

19.8 Wireframe

Il wireframe è la più semplice ed antica tecnica atta a riprodurre poligoni. Consiste semplicemente nel tracciare linee che uniscano i vertici del poligono da rappresentare, nient'altro. Il tracciamento delle linee dovrà svolgersi sfruttando le coordinate proiettate dei punti (e non quelle con tre variabili xyz). Vediamo quindi come tracciare linee nel caso si stia programmando in un linguaggio che non permetta direttamente di svolgere questa funzione (come il C e l'Assembly). Analizziamo l'algoritmo di Bresenham. Abbiamo due punti $P1(x1, y1)$ e $P2(x2, y2)$ e vogliamo visualizzarne la linea che possa unirli:



consideriamo:

```

x2 > x1
y2 > y1
dx = x2-x1
dy = y2-y1
dx > dy

```

Tutti gli altri tipi di linee sono derivabili da questo tipo. Dopodiché andiamo a calcolare questi valori:

```

x1 = x1          -> ascissa attuale del punto
y1 = y1          -> ordinata attuale del punto
d = 2*dx-dy      -> variabile di decisione
d1 = 2*dy        -> incremento di d (se d<0)
d2 = 2*(dy-dx)   -> incremento di d (se d>0)

```

Finalmente vediamo l'algoritmo vero e proprio:

```

> loop di dx iterazioni
  > visualizza pixel alla posizione (x1,y1)
  > x1=x1+1
  > se d<0 allora:
    > d=d+d1
  > altrimenti:
    > d=d+d2
    > y1=y1+1
> prossima iterazione

```

Una linea è formata da un insieme di pixel, nel nostro caso il numero di pixel che compone la linea equivale a dx , quindi dobbiamo realizzare un loop che si ripete dx volte in cui per ogni iterazione bisogna visualizzare un punto. Ma quali coordinate dovrà avere questo punto? Indichiamo con x_l e y_l le coordinate del punto da proiettare su video, le quali inizialmente coincideranno con quelle di $P1(x1, y1)$. Al termine di ogni iterazione andiamo ad incrementare x_a , in questo modo uscendo dal ciclo x_a coinciderà con $x2$ (perché $x2 = x1 + dx$). Cosa succede invece alla ordinata del pixel? La incrementiamo semplicemente quando la variabile d è positiva. È possibile utilizzare anche un altro algoritmo per tracciare linee, il quale risulta spesso più efficiente di quello di Bresenham (soprattutto se realizzato in Assembly) che sfrutta il principio di interpolazione lineare, lo vedremo più dettagliatamente nel paragrafo relativo al fill e scan line.

19.9 Hidden Face

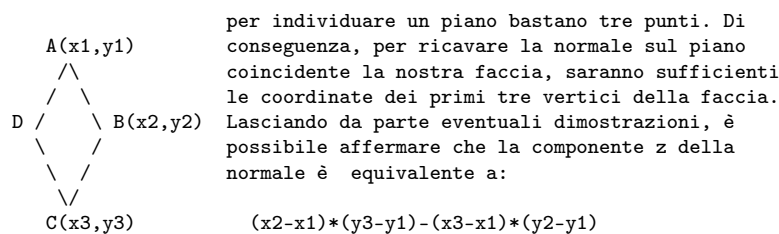
Hidden face significa faccia nascosta, in questo paragrafo vedremo come eliminarla. Difatti nella realtà in un solido non trasparente non sono visibili tutte le facce per ovvi motivi. Visualizzare sul monitor solo le facce visibili è sicuramente più realistico rispetto a tracciarle tutte.

Il più semplice e intuitivo algoritmo è quello del "pittore". Consiste nell'ordinare le facce che compongono l'oggetto in base alla componente z . In seguito si devono disegnare le facce partendo da quella più lontana sino a quella più vicina; in questo modo le facce disegnate per ultime

saranno visibili, mentre sulle nascoste verranno disegnate quelle visibili. A suo discapito questo algoritmo include un pesante spreco di tempo macchina, inoltre è praticamente inutilizzabile per grafica in wireframe, però permette a qualsiasi oggetto (che non sia wireframe) di essere visualizzato correttamente.

Un'altra via consiste nel calcolare la normale (la retta perpendicolare) su ogni faccia, controllare se punta verso l'osservatore ed in caso contrario non visualizzarla. Questo algoritmo è valido solo se i vertici che delimitano la faccia sono posti in memoria in senso orario, in quanto i calcoli che vedremo sfruttano questa caratteristica. Inoltre gli oggetti dovranno necessariamente essere convessi, ossia non devono esserci facce che possano "oscurare" (non nascondere!) altre facce. La retta normale la possiamo considerare una grandezza vettoriale che nello spazio viene indicata come un comune vettore con tre componenti.

La visibilità di un poligono dipende esclusivamente dalla sua orientazione lungo l'asse z . Essendo più precisi possiamo dire che la sola componente z è necessaria per sapere se la faccia è nascosta o meno. Consideriamo la nostra faccia come la seguente:



Se il risultato è minore o uguale a zero la faccia è nascosta, altrimenti è visibile. Per sapere semplicemente se questa componente z è maggiore o minore di zero potremmo anche utilizzare le coordinate proiettate, ovvero:

$$(xp2-xp1)*(yp3-yp1)-(xp3-xp1)*(yp2-yp1)$$

Questo è quanto basta per sapere se un poligono è o meno visibile.

19.10 Algoritmo del pittore

L'algoritmo del pittore consiste semplicemente nel visualizzare un oggetto (o una scena) 3D tracciando i poligoni partendo da quello più distante verso quello più vicino al punto di vista. Il suo scopo è di risolvere il problema causato dal tracciamento delle facce parzialmente oscurate da altre (ovvero visualizzare oggetti concavi).

Questo semplice principio è lo stesso che un qualsiasi pittore utilizza quando deve dipingere un quadro. Difatti si inizia sempre col disegnare (ad esempio) i monti (che si trovano più distanti dal punto di vista) sino a disegnare gli elementi più vicini al pittore, quali possono essere un ruscello o una persona.

Per realizzare l'algoritmo del pittore è sufficiente utilizzare un array monodimensionale a cui ogni coppia di posizioni corrispondano il puntatore o l'indice di ogni faccia e la componente z di quella faccia. Consideriamo di poter numerare le facce del nostro oggetto, quindi di poter indicare una singola faccia con un determinato numero. Il numero di elementi necessari al questo buffer sono equivalenti a:

$$\text{Dimensione Buffer} = 1 + (\text{numero facce oggetto}) * 2$$

Nella prima posizione viene salvato il numero di facce da tracciare. Invece nelle successive posizioni vengono salvati (per ogni poligono) il numero identificatore della faccia (che può essere un indice o un puntatore e la componente z di quella faccia di cui abbiamo appena parlato. La coordinata z relativa alla faccia è equivalente alla media aritmetica delle componenti z dei vertici di quel poligono.

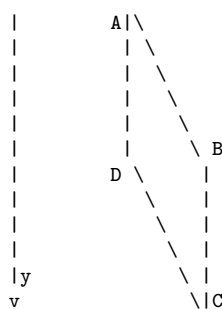
Successivamente dovremo ordinare in ordine crescente gli elementi di tale struttura in base alla coordinata z . Una volta ordinato l'array tratteremo i poligoni a video nella successione definita dagli identificatori delle facce presenti nel nostro array già ordinato (partendo dallo specificatore con la z maggiore sino ad arrivare allo specificatore della faccia con la z minore); così facendo saremo sicuri di visualizzare le facce dalle più lontane alle più vicine; ciò ci permette di rappresentare correttamente sullo schermo anche oggetti concavi.

Per snellire questo buffer possiamo calcolare la componente z della normale di ogni faccia (Nz) e, solamente se Nz risulterà positiva, salveremo l'identificatore e la z di quella faccia; in caso contrario (se $Nz < 0$) passiamo direttamente alla prossima faccia. In altre parole andiamo a considerare solo le facce orientate verso l'osservatore (consultare il paragrafo relativo all'hidden face per ulteriori approfondimenti).

Se applicato nella maniera dovuta (ovvero utilizzando un algoritmo di ordinamento ottimizzato), l'algoritmo del pittore può considerarsi uno dei metodi più veloci nella rimozione delle facce nascoste durante il tracciamento di un oggetto o di una scena 3D. I suoi principali svantaggi risiedono nella difficoltà di implementazione nel caso di intersezioni tra poligoni; nell'imprecisione visiva in certi mondi 3D complessi e nell'impossibilità di tracciare correttamente i poligoni in alcuni casi (si immagini di voler visualizzare una scena in cui vi siano un enorme poligono che rappresenti un pavimento su cui giace un piccolo cubo. Poniamo caso caso che la componente z del centro di tale cubo corrisponda con quella del poligono coincidente il pavimento. Dopo aver tracciato i poligoni su schermo può accadere che alcune facce del cubo vengano disegnate prima del pavimento, quindi alcuni poligoni, che dovrebbero essere visibili, non verranno visualizzati su schermo).

19.11 Filled vector e scan-line

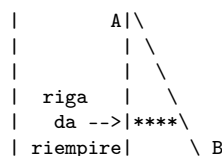
I filled vector non rappresentano altro che poligoni "riempiti" di un determinato colore. Realizzare una routine di fill significa colorare il contenuto di un poligono conoscendone le coordinate proiettate dei vertici. Immaginiamo che il poligono da riempire sia il seguente:



potremmo procedere nella seguente maniera:
a partire dalla coordinata y minore del poligono (in questo caso quella di A) e via via arrivando all'ultima posizione verticale (ossia y di D) coloriamo la riga delimitata dalle posizioni x dei lati in quella posizione y .

Il principio del fill si basa sul riempimento di righe orizzontali partendo dalla riga superiore del poligono sino a quella inferiore.

Vediamo un esempio relativo ad una sola riga:



dobbiamo riempire la riga indicata nella figura. Partiamo dalla coordinata x del lato AD. Iniziamo col colorare questo punto. Passiamo al punto successivo (ossia quello a destra) e coloriamo anch'esso; procediamo col


```

|      D \      |
|      \      |
|      \      |
|      \      |
|y      \      |
v        \C

```

colorare i pixel successivi fin quando arriviamo
a colorare il punto posto sul lato AB e passiamo
alla prossima riga.
Ciò significa che per ogni riga ci servono le
coordinate x del punto estremo a destra e del
punto estremo a sinistra.

Ora che abbiamo capito cosa fare vediamo come realizzare il tutto. Bisogna utilizzare due tabelle (matrici unidimensionali) in memoria di dimensioni equivalenti al numero di pixel verticali rappresentabili sullo schermo (es.: in una risoluzione 320*200 dobbiamo avere due tabelle di 200 valori ciascuna). Consideriamo ogni posizione delle tabelle una posizione y nello schermo ed il contenuto del primo array come la corrispondente componente x del punto estremo a sinistra mentre il valore del secondo array come la componente x del punto estremo a destra. Così facendo basterà colorare tutti i pixel alla riga corrispondente la posizione delle tabelle partendo dalla posizione x contenuta dalla prima tabella sino alla posizione x contenuta dalla seconda (tutti i punti di una riga hanno la stessa ordinata). Facciamo un esempio pratico:

```

0| x=0 ->A. <- x=0      per semplicità consideriamo i segmenti AB e CD
1| x=0 -> . . <- x=1    inclinati a 45 gradi . I vertici sono:
2| x=0 -> . . <- x=2    A(0,0) B(5,5) C(5,10) D(0,5)
3| x=0 -> . . <- x=3    le nostre due tabelle saranno:
4| x=0 -> . . <- x=4    +-----+
5| x=0 ->D. .B<- x=5    |TAB1| 0| 0| 0| 0| 0| 0| 1| 2| 3| 4| 5|..|..|
6| x=1 -> . . <- x=5    +-----+
7| x=2 -> . . <- x=5    |TAB2| 0| 1| 2| 3| 4| 5| 5| 5| 5| 5| 5|..|..|
8| x=3 -> . . <- x=5    +-----+
9| x=4 -> . . <- x=5    Per riempire il poligono basterà colorare i
10|y x=5 -> .C<- x=5    pixel compresi tra i corrispondenti valori
v                        delle due tabelle utilizzando come componente
                        y l'indice delle tabelle (che è lo stesso per
entrambe). A volte è possibile eliminare i due valori estremi delle
tabelle, quando in quelle righe vi è un solo pixel (come nel nostro
esempio). Adesso non ci rimane altro che ricavarci il contenuto di questi
due array.

```

Per definizione si indica col termine “scanline” una linea orizzontale su schermo del poligono. Un “edge” non è altro che uno dei due pixel ai bordi della scanline, quindi gli array di cui dovremo ricavarci il contenuto possiamo chiamarli “edge buffer”. Le tabelle contengono semplicemente le coordinate x di tutti i punti che compongono i lati del poligono; inoltre queste ascisse sono ordinate in base alla loro componente y . In pratica dobbiamo svolgere una routine di tracciamento di linee per tutti i lati della faccia, in cui non visualizziamo i pixel, bensì salviamo la componente x in un array la cui posizione equivale alla y di quel punto. Questa procedura è denominata “scan conversion” e praticamente consiste nel suddividere il poligono in un insieme di righe e colonne. Per realizzare la scan conversion è possibile utilizzare l'algoritmo di Bresenham, però conviene sfruttare il procedimento di interpolazione lineare, che risulta più efficiente. Vediamo sinteticamente in cosa consiste. Consideriamo due generici punti $A(x_1, y_1)$ e $B(x_2, y_2)$ dove $y_2 > y_1$. Ora calcoliamo:

```

dx = x2 - x1    <-- lunghezza della linea che unisce A e B
dy = y2 - y1    <-- altezza della linea che unisce A e B
stepx = dx / dy <-- numero di pixel orizzontali su ogni riga

```

Mentre l'algoritmo generale è :

```

> x = x1
> y = y1

```

```

> loop di dy iterazioni
  > se è libera la posizione y della tab1:
    > salva x in posizione y della tab1
  > altrimenti:
    > salva x in posizione y della tab2
  > x = x + stepx
  > y = y + 1
> prossima iterazione

```

Questo algoritmo permette di riempire parzialmente un edge buffer nel caso $y_2 > y_1$, se invece risulta $y_1 > y_2$ basterà scambiare entrambe le coordinate dei due punti (ossia considerare y_1 come y_2 e x_1 come x_2). Per riempire completamente l'edge buffer basterà ripetere tale procedura per ogni lato del poligono. La tab1 è l'array che contiene i punti estremi a sinistra mentre la tab2 punti estremi a destra. Col nostro algoritmo può capitare che alcuni i valori scritti nella tab1 appartengano alla tab2, e viceversa. Ciò significa che le coordinate x presenti nella tab1 potranno essere gli edge a destra mentre i valori presenti nella tab2 potranno essere gli edge a sinistra. Vediamo cosa fare per evitare questo inconveniente.

Se abbiamo i punti salvati in senso orario il tutto risulta più semplice e veloce. Dati due punti $A(x_1, y_1)$ e $B(x_2, y_2)$ posti in senso orario, se y_1 è maggiore di y_2 allora l'insieme dei punti che formano quel lato appartiene alla tab1 (quella contenente le posizioni x minori), in caso contrario quei punti apparterranno alla tab2 (contenente le x maggiori). Ecco l'algoritmo completo per la scan conversion relativo ad un singolo lato:

```

> confronta y1 con y2
  > se y1=y2:
    > esci dalla procedura!
  > se y1>y2:
    > la giusta tab è tab1
  > se y1<y2:
    > la giusta tab è tab2
    > scambia y1 con y2
    > scambia x1 con x2
> dy = y1 - y2
> dx = x1 - x2
> stepx = dx / dy
> x = x2
> y = y2
> loop di dy iterazioni
  > salva x in posizione y nella giusta tab
  > x = x + stepx
  > y = y + 1
> prossima iterazione

```

Una volta eseguita la scan conversion del poligono dovremo calcolare la componente y minore dei quattro punti che compongono i vertici del poligono ed l'altezza in pixel del poligono stesso. La coordinata y minore rappresenta l'indice delle tabelle da cui partire per riempire il poligono e quindi la posizione y superiore del poligono. L'altezza del poligono è equivalente alla differenza tra la y maggiore e la y minore e ci serve per sapere quante righe dovremo riempire per l'attuale poligono.

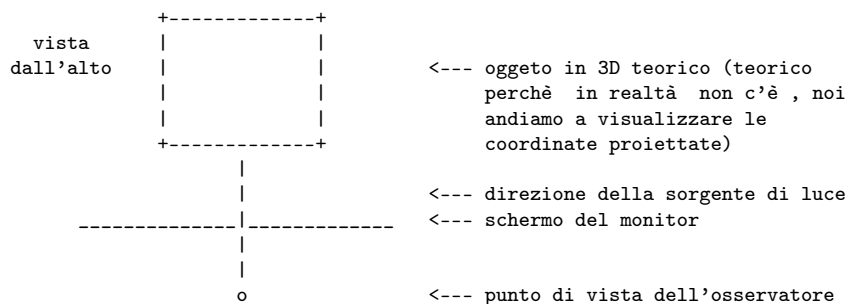
Riassumendo, per effettuare il fill di un poligono si devono svolgere i seguenti passi:

- definire in memoria due tabelle dimensionate ad ys valori (dove ys rappresenta l'altezza in pixel dello schermo);
- calcolare la y minore dei vertici e l'altezza del poligono;

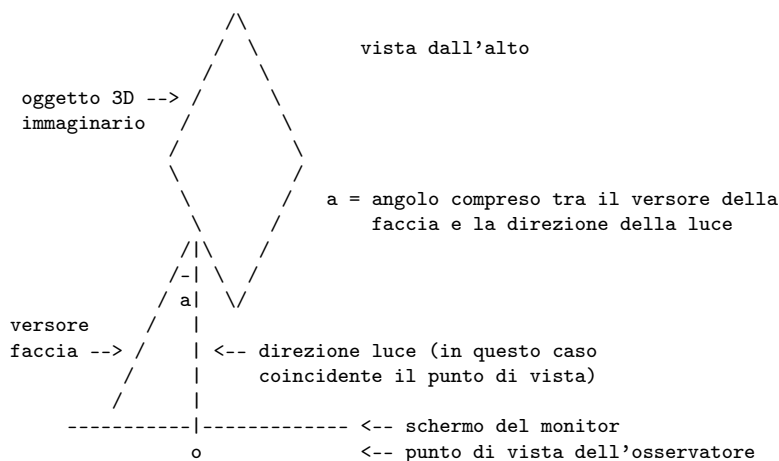
- svolgere la scan conversion del poligono salvando le coordinate x iniziali e finali (edge) di ogni scanline nell'edge buffer;
- partendo dalla posizione y minore, riempire la riga delimitata dalle posizioni x contenute dall'edge buffer stesso per un numero di volte equivalente all'altezza del poligono.

19.12 Flat shading

Siamo arrivati all'analisi del primo (e più semplice) algoritmo di shading, grazie al quale potremo attribuire ad ogni poligono comprendente l'oggetto una precisa intensità di luce, la quale verrà determinata in base all'orientazione della faccia rispetto alla sorgente di luce. Il flat shading permette di attribuire ad ogni faccia un solo colore che determinerà quanto il poligono sia illuminato. Facciamo un esempio:



Consideriamo che la sorgente di luce corrisponda con il punto di vista, la sua direzione è perpendicolare lo schermo. Definiamo l'angolo di inclinazione della faccia rispetto la sorgente di luce come l'angolo compreso tra la retta corrispondente la direzione della luce ed il versore della faccia (ovvero la retta normale). Più questo angolo è piccolo e più il poligono risulta orientato verso l'osservatore. Possiamo intuire che tanto la faccia è posta di fronte l'osservatore e tanto sarà maggiore l'intensità di luce applicata su quella faccia. Di conseguenza ad un angolo minore corrisponderà una luminosità maggiore della faccia. Ecco un altro esempio:



L'intensità di luce attribuibile al poligono è proporzionale al coseno di questo angolo. Sappiamo che il generico risultato di $\cos(a)$ è compreso tra -1 e 1 . Da notare che se il poligono

è visibile il nostro angolo varia da 0 a 90 gradi, altrimenti la faccia risulta nascosta (è possibile utilizzare questa caratteristica per l'eliminazione dell'hidden face!). Quindi il valore del coseno relativo al nostro angolo copre un range di valori da 0 e 1. Inoltre, utilizzando 256 colori, basterà moltiplicare il coseno per 256 (oppure effettuare uno shift di 8 bit a sinistra) e otterremo il pixel chunky col quale dovremo riempire la relativa faccia! Vediamo come ricavare questo valore.

Innanzitutto dobbiamo specificare la palette dei colori da utilizzare. Ciò è possibile definendo nella memoria video una tavolozza che parte dal colore di luminosità minore sino ad arrivare gradualmente al colore più chiaro. Per il calcolo del coseno conviene sfruttare la regola di Lambert, la quale afferma che il prodotto scalare tra due rette espresse come grandezze vettoriali è equivalente al prodotto della lunghezza dei relativi vettori e del coseno dell'angolo limitato dalle rette stesse, ovvero l'angolo a . Quindi, per conoscere $\cos(a)$, non dovremo far altro che svolgere questo prodotto scalare e dividere il risultato per il prodotto delle lunghezze dei due vettori.

Per calcolare il prodotto scalare di due vettori si esegue il prodotto delle corrispondenti componenti e poi si sommano i risultati, ad esempio:

```
H=(xh,yh,zh) ; K=(xk,yk,zk)
H*K=xh*xk+yh*yk+zh*zk      <-- prodotto scalare
```

Per ricavare una lunghezza di un vettore possiamo ricorrere al teorema di Pitagora grazie al quale si può affermare che la lunghezza è equivalente alla radice quadrata della somma dei quadrati di ogni componente.

Verifichiamo come calcolare i coefficienti x , y e z del versore della faccia:

```
Nx=(y2-y1)*(z3-z1)-(y3-y1)*(z2-z1)    <-+--- i tre coefficienti
Ny=(z2-z1)*(x3-x1)-(z3-z1)*(x2-x1)    <-|      della retta normale
Nz=(x2-x1)*(y3-y1)-(x3-x1)*(y2-y1)    <-+
```

N.B.: i punti devono essere posti in memoria in senso orario!

```
x1, y1, z1 = componenti del primo punto del poligono
x1, y2, z2 = componenti del secondo punto del poligono
x3, y3, z3 = componenti del terzo punto del poligono
```

Infine ecco la formula per calcolare il pixel chunky:

```

                                Nx*lx + Ny*ly + Nz*lz
cos(a)=-----
          sqrt(Nx*Nx+Ny*Ny+Nz*Nz) * sqrt(lx*lx+ly*ly+lz*lz)

pixel chunky = 256*cos(a)

a = angolo tra il versore e la direzione della sorgente di luce
lx = componente x della sorgente di luce
ly = componente y della sorgente di luce
lz = componente z della sorgente di luce
```

Le coordinate lx , ly e lz rappresentano la posizione della sorgente di luce. Nel caso la luce coincida con il punto di vista dell'osservatore, le relative coordinate risulteranno:

```
lx=0 ; ly=0 ; lz=-256
```

zl è uguale all'opposto della distanza tra l'osservatore e lo schermo (nel nostro caso la distanza tra l'osservatore e lo schermo è 256).

19.13 Ottimizzazioni per il calcolo della sorgente di luce

In questa parte vediamo come velocizzare il nostro motore 3D nel caso comprenda l'implementazione di una sorgente di luce reale.

Una prima ottimizzazione consiste nell'utilizzare un buffer dove andremo a precalcolare tutte le normali di ogni faccia (o di ogni vertice nel caso del gouraud shading); poi anziché calcolare ad ogni frame tutte le normali, ruotiamo i nostri versori precalcolati dello stesso angolo con cui ruotiamo i vertici dell'oggetto sfruttando la stessa identica procedura descritta in precedenza (utilizzando preferibilmente l'algoritmo a 9 moltiplicazioni).

Abbiamo detto che il prodotto scalare tra il vettore normale ed il vettore corrispondente la sorgente di luce moltiplicato 256 ci permette di conoscere il pixel chunky. Ora vediamo come eliminare subito le radici quadrate e la divisione. Analizziamo di nuovo la formula:

$$\cos(a) = \frac{N_x * l_x + N_y * l_y + N_z * l_z}{\sqrt{N_x * N_x + N_y * N_y + N_z * N_z} * \sqrt{l_x * l_x + l_y * l_y + l_z * l_z}}$$

Per attuare questa ottimizzazione dobbiamo rendere unitario il vettore normale ed il vettore corrispondente la sorgente di luce. Rendere unitario un vettore significa dividere ognuna delle componenti per la sua distanza dall'origine; ciò fa sì che le nuove componenti abbiano un range compreso tra -1 e $+1$, ecco perché si dice unitario. Vediamo algebricamente come rendere unitario un qualsiasi versore:

$$uN_x = \frac{N_x}{\sqrt{N_x * N_x + N_y * N_y + N_z * N_z}}$$

$$uN_y = \frac{N_y}{\sqrt{N_x * N_x + N_y * N_y + N_z * N_z}}$$

$$uN_z = \frac{N_z}{\sqrt{N_x * N_x + N_y * N_y + N_z * N_z}}$$

Più genericamente, dato un vettore $V(x, y, z)$, per calcolare le componenti del relativo vettore unitario $uV(ux, uy, uz)$:

$$u_x = \frac{x}{\sqrt{x * x + y * y + z * z}}$$

$$u_y = \frac{y}{\sqrt{x * x + y * y + z * z}}$$

$$u_z = \frac{z}{\sqrt{x * x + y * y + z * z}}$$

Per rendere unitaria la sorgente di luce possiamo anche evitare di dividere ogni sua componente per la lunghezza della medesima, infatti siamo noi a decidere dove si trova, quindi arbitrariamente possiamo assegnare coordinate unitarie. Ad esempio tornando al caso che la luce coincida col punto di vista:

$$u_lx=0 \quad ; \quad u_ly=0 \quad ; \quad u_lz=-1$$

Quindi la formula per calcolare l'intensità di luce viene ridotta a:

$$\begin{aligned}\cos(a) &= uNx*ulx + uNy*uly + uNz*ulz \\ \text{pixel chunky} &= 256*\cos(a)\end{aligned}$$

Rendere unitario un vettore e poi ruotarlo o ruotare un vettore e poi renderlo unitario significa svolgere la stessa funzione; quindi al momento in cui andiamo a precalcolare le normali possiamo renderle subito unitarie, in seguito andremo a ruotare i versori già resi unitari. In questo modo evitiamo di eseguire 2 radici quadrate e una divisione per vertice ad ogni frame!

N.B.: nel caso si voglia muovere la sorgente di luce, utilizzando questa ottimizzazione non si potranno attuare traslazioni della sorgente di luce, ma solo rotazioni, in quanto la distanza origine-luce deve restare costante.

L'ultima ottimizzazione consiste nel mantenere fissa in un punto la sorgente di luce, più precisamente diciamo che deve coincidere sempre con il punto di vista dell'osservatore. Naturalmente utilizzeremo coordinate unitarie per i versori e la luce. Vediamo la formula per calcolare il pixel chunky in questo particolare caso:

$$\begin{aligned}\text{pixel chunky} &= 256*(uNx*ulx + uNy*uly + uNz*ulz) = \\ &= 256*(uNx*0 + uNy*0 + uNz*(-1)) = \\ &= -256*uNz\end{aligned}$$

Ora il nostro pixel chunky dipende solo da uNz , quindi possiamo precalcolare per ogni vertice $-256 * uNz$ al posto del semplice uNz , ruotarlo e utilizzare subito questo valore come pixel chunky. In questo modo evitiamo 3 moltiplicazioni e 2 addizioni. Inoltre siccome ci serve solo uNz possiamo benissimo evitare di ruotare uNx e uNy , risparmiando la bellezza di altre 6 moltiplicazioni per faccia (o per vertice nel caso del gouraud). In totale risparmiamo ben 9 moltiplicazioni e 2 addizioni per faccia (o per vertice nel gouraud)! Naturalmente dovremo precalcolare oltre a $-256 * uNz$ anche uNx e uNy moltiplicati 256 ($256 * uNx$, $256 * uNy$) che servono per poter ruotare $-256 * uNz$. Inoltre se invertiamo la nostra palette potremo utilizzare $256 * uNz$ al posto di $-256 * uNz$.

19.14 Gouraud shading

Questo algoritmo di ombreggiatura permette di sfumare l'interno di ogni poligono al contrario del flat shading col quale si assegna un unico colore per faccia.

Innanzitutto bisogna calcolare il versore di ogni vertice dell'oggetto anziché di ogni poligono. Le componenti della normale sul vertice sono equivalenti alla media aritmetica delle componenti delle normali di tutte le facce che toccano quel vertice. Facciamo un esempio:

----	sia V un generico vertice di un cubo appartenente
/f2 /\	alle facce f1, f2 e f3. Consideriamo le normali di
/___/V \	codeste facce, chiamiamo questi versori N1,N2,N3.
\f1 \f3/	
___\	N1(Nx1,Ny1,Nz1) N2(Nx2,Ny2,Nz2) N3(Nx3,Ny3,Nz3)

Allora la normale su V è equivalente a:

$$NV((Nx1+Nx2+Nx3)/3, (Ny1+Ny2+Ny3)/3, (Nz1+Nz2+Nz3)/3)$$

In questo caso le facce appartenenti a V sono 3, a seconda dell'oggetto che si vuol utilizzare il numero di poligoni appartenenti ad un vertice cambiano.

Una volta precalcolate tutte le normali (preferibilmente già unitarie) su ogni spigolo dovremo calcolare la quantità di luce che cade su ognuno dei vertici, ovvero il pixel chunky, utilizzando la legge già studiata nel flat shading (magari sfruttando eventuali ottimizzazioni citate nel precedente paragrafo).

In seguito facciamo la scan conversion (spiegata nel paragrafo dedicato al fill e scan line) di tutti i poligoni visibili.

Adesso dobbiamo interpolare linearmente per ogni faccia i pixel chunky appartenenti ai vertici di quella faccia. In pratica dovremo svolgere una semplice scan conversion del poligono utilizzando i pixel chunky al posto delle coordinate x dei vertici, tutto qui. Naturalmente ciò va svolto solamente se la faccia risulta visibile.

Non rimane altro che effettuare il fill vero e proprio dei poligoni. Come per un normale fill bisogna eseguire un loop ad iterazioni equivalenti all'altezza in pixel del poligono. Ad ogni iterazione preleviamo di volta in volta le coordinate x iniziali e finali dalle tabelle delle scan line (come per un normale fill), però stavolta preleviamo anche i pixel chunky iniziali e finali. Ora dobbiamo interpolare il pixel chunky iniziale con quello finale partendo dalla coordinata x iniziale sino ad arrivare a quella finale. Per far ciò basta utilizzare l'algoritmo per tracciare una scan line con le seguenti modifiche:

- utilizzare il pixel chunky iniziale al posto della coordinata $x1$;
- utilizzare il pixel chunky finale al posto della coordinata $x2$;
- utilizzare la x iniziale al posto della coordinata $y1$;
- utilizzare la x finale al posto della coordinata $y2$;
- utilizzare la riga dello schermo chunky da "fillare" come tabella dove la scan line verrà salvata.

Ed ecco realizzato il gouraud shading!

19.15 Phong shading

Il phong shading permette di assegnare ad ogni pixel la sua reale intensità di luce, al contrario del gouraud nel quale si generano sfumature all'interno di ogni faccia tra le intensità di luce di ogni vertice dell'oggetto. La maggiore definizione che si ottiene col phong rispetto al gouraud comporta allo stesso tempo un drastico aumento delle operazioni che il processore deve svolgere. La pesantezza dei calcoli da eseguire è tale da impedire agli attuali elaboratori di tracciare soddisfacenti scene in phong shading in tempo reale.

Nel gouraud calcoliamo la reale intensità di luce su ogni vertice, dopodiché ogni colore viene interpolato lungo ogni lato del poligono, infine si interpolano i colori posti sui lati estremi a sinistra con i colori posti sui lati estremi a destra in modo da riempire l'intero poligono. Nel phong invece interpoliamo sempre le normali, non si interpolano mai i colori. Una volta determinati i versori su ogni vertice, questi devono essere interpolati lungo ogni lato; successivamente i versori posti lungo i lati estremi a sinistra verranno interpolati con quelli posti lungo i lati estremi a destra, quindi per ogni singolo verrà calcolato il colore sfruttando la tradizionale formula più volte studiata.

Il phong ci impedisce di sfruttare le diverse ottimizzazioni possibili col gouraud shading e col flat shading. Infatti nel phong è impossibile utilizzare normali unitarie in quanto nel momento in cui queste vengono interpolate la loro lunghezza (equivalente al risultato dell'espressione $\sqrt{Nx * Nx + Ny * Ny + Nz * Nz}$) può variare. Quindi occorre eseguire almeno una divisione ed una radice quadrata per pixel, il che non è poco.

19.16 Reflection mapping

Se un solido riflette sempre e solo una singola immagine (in gergo denominata “texture”) allora possiamo affermare che su quel solido è stato applicato il reflection mapping. Nel caso la texture corrisponda approssimativamente alla rappresentazione bidimensionale di una luce (es.: un cerchio il cui centro risulta molto chiaro mentre agli orli viene sfumato in una tinta più scura), è possibile raggiungere effetti simili (e a volte superiori) al phong e al gouraud.

Spesso questo effetto viene erroneamente confuso con l’environment mapping, il quale permette invece di riflettere un intero ambiente che circonda l’oggetto (il quale ambiente è spesso definito per semplicità come un cubo, quindi in questo caso sul solido verranno riflesse sei immagini). Tuttavia è possibile considerare il reflection mapping come un’approssimazione dell’environment mapping.

In questo paragrafo verrà descritto come realizzare il reflection mapping utilizzando esclusivamente texture di dimensioni 256*256 pixel. L’implementazione di texture di dimensioni differenti è facilmente derivabile.

Vediamo dettagliatamente come realizzare un comune oggetto in reflection mapping. Inizialmente ci andiamo a precalcolare tutti i versori unitari su ogni vertice (come per il gouraud) e li moltiplichiamo per 128 (oppure applichiamo un semplice scorrimento di 7 bit a sinistra), il che matematicamente si traduce in:

$$\begin{array}{l} \text{PV} \left| \begin{array}{l} PVx = 128 * Nx / \sqrt{Nx * Nx + Ny * Ny + Nz * Nz} \\ PVy = 128 * Ny / \sqrt{Nx * Nx + Ny * Ny + Nz * Nz} \\ PVz = 128 * Nz / \sqrt{Nx * Nx + Ny * Ny + Nz * Nz} \end{array} \right| \end{array}$$

Chiamiamo PV il vettore che ha come componenti questi 3 valori. La normale unitaria ha come coordinate 3 valori che comprendono numeri reali tra -1 e $+1$. Ora abbiamo PVx , PVy e PVz che rispetto ai versori unitari sono moltiplicati 128, questo vuol dire che copriranno un raggio di valori compresi tra -128 e $+128$ (anche se in realtà tali valori non superano mai $+127$). E qui finisce la fase di precalcolo.

In tempo reale dobbiamo ruotare il vettore PV per ogni vertice (casomai sfruttando gli stessi fattori di rotazione dei punti se si ha realizzato la rotazione a 9 moltiplicazioni). Del vettore PV ci servono solo le componenti x e y ruotate, quindi possiamo anche non ruotare PVz evitando così di svolgere almeno 3 moltiplicazioni e 2 addizioni per vertice (naturalmente è necessario precalcolare PVz per ogni vertice per poter ruotare PVx e PVy). Ad ognuna delle componenti ruotate x e y del vettore PV addizioniamo il valore 128. Al termine di questi calcoli, il range dei valori che PVx e PVy potranno coprire sarà compreso tra 0 e 255.

In realtà $((PVx_{ruotato}) + 128)$ e $((PVy_{ruotato}) + 128)$ rappresentano le coordinate della texture da mappare (ossia tracciare) sul poligono. Ciò significa che se abbiamo un poligono delimitato da 4 punti, dobbiamo mappare su quel poligono la parte di texture delimitata dai 4 relativi PVx e PVy ruotati (e sommati con 128). Quindi basterà mappare il “pezzo” di texture su quel poligono e ripetere il tutto per ogni faccia visibile per realizzare il reflection mapping!

Ora vediamo come tracciare la parte di texture una volta calcolati i nuovi PVx e PVy . Innanzitutto dobbiamo svolgere la scan conversion del poligono, in più bisogna interpolare PVx e PVy lungo tutti i lati della nostra faccia, il che significa fare 2 ulteriori scan conversion del poligono utilizzando i PVx e i PVy al posto delle coordinate x dei vertici. Quindi in tutto vi sono 3 scan conversion: la prima è quella tradizionale, la seconda viene fatta sostituendo PVx alle x dei vertici, mentre la terza utilizza i PVy al posto delle x (facciamo esattamente come per le normali nel phong, con la differenza che consideriamo 2 componenti (PVx e PVy) al posto di 3 (Nx , Ny e Nz)). Abbiamo eseguito la scan conversion del poligono, quel di cui abbiamo bisogno

è un algoritmo che permetta di associare ad ogni punto appartenente alla faccia un determinato pixel della texture. Consideriamo la seguente figura come la nostra faccia proiettata a video:

<pre> P1 -> . . . <- P2 </pre> <pre> P1 -> x1, y, PVx1, PVy1 P2 -> x2, y, PVx2, PVy2 dPVx = PVx1-PVx2 dPVy = PVy1-PVy2 dx = x1-x2 </pre>	<p>Dopo aver interpolato PVx e PVy e svolto la scan conversion, per ogni coppia di punti sulla stessa posizione y dello schermo, ad esempio $P1$ e $P2$, conosciamo la loro coordinata x assieme a PVx e PVy. Adesso dobbiamo interpolare PVx e PVy dal punto $P1$ al punto $P2$, in questo modo sapremo il valore di PVx e PVy per tutti i punti del poligono. Per interpolare questi 2 valori lungo una riga si deve applicare l'algoritmo generale per il tracciamento di una scan line utilizzando dx al posto di dy, $dPVx$ (per interpolare PVx) e $dPVy$ (per interpolare PVy) al posto di dx, proprio come abbiamo fatto nel gouraud per interpolare i pixel chunky.</p>
--	---

Come già accennato, PVx e PVy rappresentano le coordinate del pixel texture da tracciare. Appena svolte le 3 scan conversion conoscevamo PVx e PVy appartenenti ad ogni vertice della faccia. Quindi, interpolando PVx e PVy lungo tutto il poligono, ricaveremo le coordinate dei punti della texture per tutti i punti della faccia! Ora, con delle semplici operazioni di copia, potremo mappare il poligono associando, ad ogni suo punto, il pixel chunky della texture in posizione (PVx, PVy) .

19.17 Affine texture mapping

Questo effetto permette il tracciamento di un'intera immagine su di un poligono, in pratica è come se "incollassimo" su ogni faccia una texture (ovvero una semplice immagine chunky posta in memoria). In questo paragrafo tratteremo del texture mapping senza prospettiva (il cosiddetto "affine texture mapping"), il quale risulta essere uno degli algoritmi più veloci per mappare un'immagine su di un poligono, ma allo stesso tempo anche il meno realistico.

Per completezza definiamo gli assi relativi alla texture (posta in memoria, non quella visualizzata su schermo) hanno come origine il punto più in alto a sinistra. L'ascissa di tale sistema viene denominata $< u >$ mentre l'ordinata $< v >$; quindi un generico punto della texture può essere indicato come $P(u, v)$. Da notare che le componenti u e v non possono assumere valori negativi.

Consideriamo di utilizzare poligoni formati da 4 lati, ogni spigolo del poligono coincide con uno spigolo della faccia, ciò che dobbiamo fare è tracciare tutta la texture sul poligono. In pratica è come se dovessimo realizzare il reflection mapping sapendo che le coordinate della texture da mappare sono sempre costanti e coincidono esattamente con i 4 vertici della texture stessa. Se la texture è 256×256 pixel realizzeremo un algoritmo di reflection mapping sapendo che $PV1(0, 0)$, $PV2(255, 0)$, $PV3(255, 255)$, $PV4(0, 255)$ sono gli stessi per ogni poligono. In altre parole andiamo ad interpolare le coordinate x e y della texture (ovvero u e v) lungo tutto il poligono, in modo tale da sapere per ogni pixel appartenente alla faccia da tracciare quale sia il relativo punto della texture. Tutto qui.

Per chiarire meglio le idee osserviamo un semplice esempio:

<pre> A +-----+ B </pre>	<p>Abbiamo il nostro poligono ABCD in cui intendiamo applicarci una texture. Vogliamo che al punto più in alto a</p>
--	--

$\begin{array}{c} | \\ | \\ | \\ | \\ | \end{array}$
 $\begin{array}{c} | \\ | \\ | \\ | \\ | \end{array}$

sinistra della texture corrisponda il
 vertice A, al punto in alto a destra
 il vertice B, a quello in basso a
 destra C, mentre al punto in basso a
 sinistra il vertice D.
 Facciamo finta che le coordinate del
 poligono siano già state ruotate.
 Al punto A corrisponde il punto (0,0)
 della texture, al punto B le coordinate (255,0), a C corrisponde (255,255)
 ed infine a D corrisponde il pixel (0,255) della texture.

Quindi svolgiamo 2 ulteriori scan conversion della faccia (oltre a quella classica per “fillare” il poligono) utilizzando una volta le $\langle u \rangle$ iniziali e finali al posto delle x , mentre nella seconda volta si devono sfruttare le $\langle v \rangle$ iniziali e finali al posto delle x ; in questo modo (interpolando la $\langle u \rangle$ e la $\langle v \rangle$ per ogni scanline) sapremo le coordinate (u, v) della texture di ogni pixel della faccia.

19.18 Ottimizzazione del fill

In questo paragrafo studieremo un alternativo algoritmo per realizzare il fill dei poligoni nel caso avessimo bisogno di interpolare una o più variabili lungo tutto il poligono (come nel caso del texture mapping, del gouraud e del phong shading). L'unico vero svantaggio nell'utilizzare tale metodo consiste nel fatto che tutti poligoni dovranno per forza essere dei triangoli.

Consideriamo di dover mappare una texture (o parte di essa) su di un poligono. Applicando i metodi precedentemente descritti avremmo eseguito 2 divisioni per scanline, ovvero:

$du = u2 - u1$ dove $(u1, v1)$ rappresentano le coordinate
 $dv = v2 - v1$ della texture per l'edge a sinistra, mentre
 $stepu = du / dx$ $(u2, v2)$ valgono per l'edge a destra.
 $stepv = dv / dx$

L'ottimizzazione notevole che si ottiene con l'algoritmo che vedremo, sta nel calcolare $stepu$ e $stepv$, detti constant slope, una sola volta per tutto il poligono. In altre parole al posto di svolgere 2 divisioni per scanline, eseguiremo 2 divisioni per poligono, non male!

In un triangolo qualsiasi, gli $stepu$ e $stepv$ sono sempre costanti lungo tutto il poligono, quindi potremo calcolarci i constant slope una volta sola (per faccia) e riutilizzarli per tutte le scanline di tale poligono. Per ottenere risultati il più precisi possibile, calcoleremo gli $stepu$ e $stepv$ relativi alla scanline più lunga di tutto il poligono, poi utilizzeremo tali constant slope per tutte le scanline. Questo è il concetto fondamentale per poter ottimizzare il fill.

Consideriamo un generico triangolo:

$\begin{array}{c} A(x1, y1) \\ \swarrow \quad \searrow \\ L3 \quad \quad L1 \\ \swarrow \quad \searrow \\ C(x3, y3) \quad \quad B(x2, y2) \end{array}$

$\begin{array}{c} L2 \end{array}$

$A(x1, y1)$ è il vertice superiore,
 che sta più in alto;
 $B(x2, y2)$ è il vertice che sta a
 media altezza;
 $C(x3, y3)$ è il vertice inferiore,
 che sta più in basso;
 $L1$ è il lato AB; $L2$ è il lato BC
 mentre $L3$ è il lato CA.

La scanline più lunga è sempre quella che giace sul vertice B, ossia quel vertice vediamo come calcolarne la relativa lunghezza assieme ai constant slope:

```

temp = (y2-y1) / (y3-y1)
scan = temp * (x3-x1) + (x1-x2)      <- lunghezza scanline maggiore
stepu = (temp * (u3-u1) + (u1-u2)) / scan <- constant slope u
stepv = (temp * (v3-v1) + (v1-v2)) / scan <- constant slope v

```

Il valore di $\langle scan \rangle$ può essere positivo o negativo, il che ci permette di determinare se il vertice B è il punto estremo a destra o a sinistra dell'intero poligono:

```

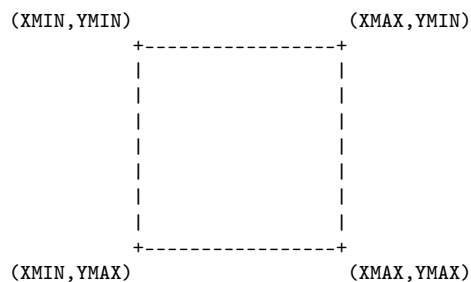
se (scan = 0) allora il poligono è vuoto, non tracciare il poligono;
se (scan > 0) allora il vertice B è posto a sinistra, quindi i lati L1
                  e L2 appartengono all'edge di destra, mentre solo
                  L3 appartiene all'edge di sinistra;
se (scan < 0) allora il vertice B è posto a destra, quindi i lati L1 e
                  L2 appartengono all'edge di sinistra, mentre solo
                  L3 appartiene all'edge di destra;

```

19.19 Clipping 2D

Nella visualizzazione di un oggetto o un mondo 3D, bisogna considerare il problema del tracciamento di poligoni o linee parzialmente visibili sullo schermo, ovvero di facce o linee che "escono" (anche parzialmente) fuori dal monitor. Il procedimento atto a risolvere questo problema è chiamato clipping 2D. Questo tipo di clipping è 2D in quanto ci si propone di "tagliare" i nostri oggetti solo rispetto al singolo piano coincidente lo schermo.

Come condizioni iniziali consideriamo i valori $XMIN$ e $XMAX$ come le componenti x estreme a sinistra e a destra dello schermo, mentre $YMIN$ e $YMAX$ come le componenti y estreme in alto e in basso dello schermo. Quindi i 4 vertici immaginari dello schermo avranno le seguenti coordinate:



Ad esempio, nell'ipotetico caso di uno schermo di risoluzione 320x200:

```

XMIN = 0 ; XMAX = 319 ; YMIN = 0 ; YMAX = 199

```

Analizziamo subito come clippare una singola linea, in seguito vedremo il clipping relativo ai poligoni (il quale si basa sullo stesso principio per tagliare una linea).

Caso 1 la linea esce dal bordo superiore dello schermo:

```

P1 .
  \
   \ P1c
+-----+-----+
|       \       |

```

consideriamo P1 il punto che sta più in alto
e P2 il punto che sta più in basso.
Quel che dobbiamo fare è calcolarci le nuove
coordinate (clippate) di P1 in modo tale che
risulti posizionato sul bordo superiore dello

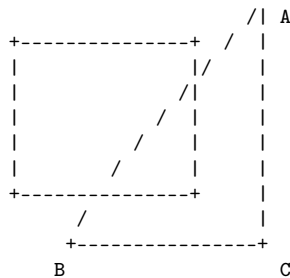
bisogno di 4 passi, uno per ogni lato della finestra di clipping. *SH* lavora su una lista di vertici e produce in output, ad ogni passo, una nuova lista di vertici che si trovano dalla parte visibile rispetto al lato di clipping. Siano $P1$ il vertice attuale nella lista, $P2$ il vertice seguente e T l'eventuale punto di intersezione fra la retta passante per $P1$ e $P2$ e il lato di clipping.

Bisogna considerare 4 casi:

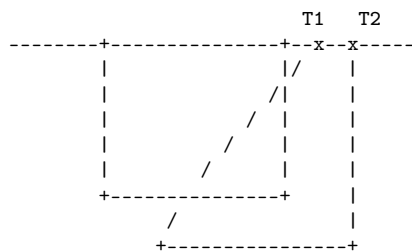
1. $P1$ interno $P2$ interno Nella lista di destinazione si inserisce $P2$.
2. $P1$ interno $P2$ esterno Nella lista di destinazione si inserisce T .
3. $P1$ esterno $P2$ interno Nella lista di destinazione si inseriscono T e $P2$.
4. $P1$ esterno $P2$ esterno Non si inserisce nessun vertice.

Le condizioni interno/esterno nel caso di clipping con una finestra rettangolare sono dei semplici confronti, nel caso di finestra convessa non rettangolare è necessario eseguire un prodotto scalare. Ad esempio il test $P1$ interno nel caso di clipping con il lato sinistro dello schermo è semplicemente $P1.x > 0$.

Un esempio di clipping di un triangolo con una finestra rettangolare:

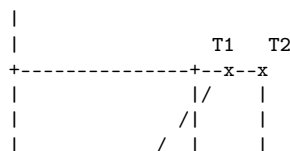


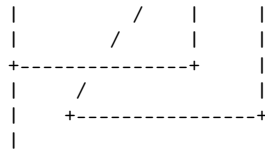
Passo 1 (clipping con il lato superiore) :



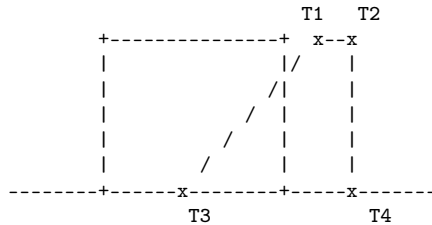
A è esterno e B è interno, si inseriscono T1 e B.
 B è interno e C è interno, si inserisce C.
 C è interno e A è esterno, si inserisce T2.

Si ottiene un nuovo poligono con i due vertici temporanei $T1$ e $T2$. Passo 2 (clipping con il lato sinistro):

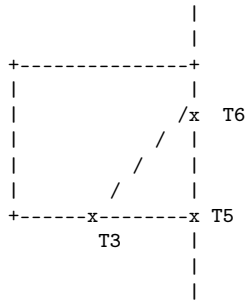




Passo 3 (clipping con il lato inferiore) :



Si ottiene un nuovo poligono con i due vertici temporanei $T3$ e $T4$. Passo 4 (clipping con il lato destro) :



Il poligono definitivo ha per vertici $T3$, $T5$ e $T6$.

CDS/DeathStar

19.20 Z-Buffer

Lo Z-Buffer è un'area di memoria dedicata all'eliminazione delle facce completamente o parzialmente nascoste. E' particolarmente utile per tracciare poligoni che si intersecano e non comporta limitazioni nella scena da tracciare (al contrario dell'algoritmo del pittore).

Lo spazio occupato in memoria dallo Z-Buffer coincide con le dimensioni dello schermo (es.: per uno schermo 320×200 avremo bisogno di un buffer di 64000 valori). Consideriamo lo Z-Buffer come il nostro schermo chunky, con la differenza che ogni posizione non rappresenta un pixel, bensì la componente Z di quel pixel; in questo modo possiamo sapere la coordinata Z di tutti i punti visualizzati su schermo.

Ora si procede nella seguente maniera. Per una ed una sola volta per ogni frame andiamo ad inizializzare lo Z-Buffer copiandoci su ogni posizione un valore arbitrario piuttosto elevato.

In seguito ci andiamo ad interpolare la coordinata Z di ogni vertice lungo tutto il poligono (ossia facciamo come nel gouraud, ma consideriamo Z al posto di N_z).

Nel loop nel quale tracciamo i pixel (ovvero il loop più interno del fill), confrontiamo la Z che stiamo interpolando con il valore corrispondente nello Z-Buffer (la cui posizione deve coincidere con quella dello schermo chunky) e, soltanto nel caso la nostra Z sia minore rispetto a quella

dello Z-Buffer, copiamo la nostra Z nello Z-Buffer e scriviamo il pixel nello schermo chunky. Nel caso in cui la nostra Z sia maggiore o uguale rispetto al valore dello Z-Buffer, allora non dobbiamo nè aggiornare lo Z-Buffer, nè tracciare il pixel su schermo (comunque in ogni caso dobbiamo continuare ad interpolare sia la Z che altre eventuali variabili).

19.21 Bump mapping 2D

Nel texture mapping abbiamo la possibilità di rivestire un oggetto con un'immagine qualunque, quindi la superficie dell'oggetto sarà data dalla texture ad essa applicata, quindi la superficie può considerarsi liscia, ovvero piana, completamente piatta. Il bump mapping ha lo scopo di rendere una texture "pseudo-tridimensionale", il che significa che l'immagine è solida (e non piatta) a livello ottico, ma in realtà viene elaborata come uno strumento 2D (in pratica non c'è nessun asse z per la texture). Per creare l'effetto "simil-3D" applichiamo sull'immagine ombre e luci, utilizzando casomai una sorgente di luce in movimento.

Per realizzare il bump mapping dovremo utilizzare una texture un pò particolare, la quale deve avere una determinata disposizione dei colori. Mettiamo caso che la nostra immagine utilizzi 256 colori, partendo dal primo colore sino al duecentocinquantaseiesimo specifichiamo le relative componenti RGB; nel nostro caso il primo colore dovrà essere il più scuro, il successivo dovrà essere quello un pò meno scuro, il terzo quello poco più chiaro del secondo e così via, in modo tale che l'ultimo colore sia il più chiaro di tutti. La texture dovrà trovarsi in memoria secondo una successione continua e ordinata di righe (ovvero: ogni riga di pixel dovrà trovarsi esattamente nella locazione di memoria successiva rispetto alla locazione in cui è presente l'ultimo pixel della riga precedente). Inoltre ogni pixel dovrà essere rappresentato come un singolo byte.

Premesso ciò, possiamo fare alcune osservazioni. Facciamo finta che la texture da "bumpare" sia la rappresentazione di una catena montuosa vista da un aereo, la cui tavolozza di colori è composta da 256 tonalità di grigio. Prendiamo un pixel qualunque di questa immagine, il pixel chunky è un byte, quindi un valore compreso tra 0 e 255; tanto questo valore sarà maggiore e tanto quel pixel risulterà chiaro (come colore). Un pixel più è chiaro e più rifletterà luce; i punti che riflettono più luce sono quelli maggiormente vicini alla sorgente di luce: in sostanza possiamo affermare che più è alto il valore del pixel chunky e più questo risulterà vicino alla sorgente di luce; al contrario possiamo dire che più è basso il valore del pixel e più sarà lontano dalla sorgente di luce.

Adesso vediamo come realizzare praticamente il bump mapping. L'algoritmo si può suddividere principalmente in 2 parti: nella prima svolgiamo un precalcolo, ovvero calcoliamo una tabella in memoria; la seconda parte consiste nel calcolo vero e proprio della texture "bumpata", la quale parte (a differenza della prima) viene svolta in tempo reale. Per una singola texture, il precalcolo della tabella viene svolto solamente una volta.

La tabella occupa esattamente 4 volte lo spazio occupato dalla texture. Per ricavarla dobbiamo fare l'intera scansione della nostra immagine. Supponiamo che $P1$ sia l'attuale pixel chunky scansionato, che $P2$ sia il successivo punto sulla destra di $P1$ e che $P3$ sia esattamente quello posto al di sotto di $P1$, quindi le coordinate di questi 3 punti saranno:

```
P1( x , y )      <- punto attuale
P2( x+1 , y )    <- punto sulla destra di P1
P3( x , y+1 )    <- punto al di sotto di P1
```

Ecco i calcoli da eseguire:

```
px = P1-P2      <- differenza di pixel chunky
py = P1-P3      <- differenza di pixel chunky
```

```

ox = h*sin(px)
oy = h*sin(py)

of = oy*tx+ox      <- valore della tabella da precalcolare

tx = 256           <- larghezza della texture in pixel
h      <- valore arbitrario tra 0 e 255

```

Da notare che px e py coprono un range di valori tra -255 e $+255$. Nel nostro caso -255 rappresenta $-\text{pigreco}/2$ (ovvero -90 gradi), mentre $+255$ indica $\text{pigreco}/2$ ($+90$ gradi). Se avessimo una funzione seno che accettasse in ingresso un angolo in radianti, $\sin(px)$ sarebbe equivalente a: $\sin(px * 512/PI)$. È consigliabile calcolare una tabella di $\sin(x)$ in cui x varia da -90 gradi a $+90$ gradi formata da 512 valori.

La tabella è composta semplicemente da un insieme di of , e precisamente uno per ogni pixel dell'immagine. of è un valore a 32 bit, quindi per una texture $256*256$ la tabella risulta lunga 256kb. Il valore $< h >$ è il coefficiente di perturbazione, cioè indica il grado dell'effetto bump. In altre parole tanto h sarà maggiore e tanto si noterà che la texture è tridimensionale, tanto h sarà minore e tanto la texture risulterà piatta. E qui finisce la fase di precalcolo.

Vediamo la parte da svolgere in tempo reale. Innanzitutto ci serve una ulteriore texture, la quale verrà utilizzata per "simulare" la luce applicata alla texture da bumpare. Questa immagine è la riproduzione vera e propria della luce; in pratica rappresenta una serie di cerchi concentrici, di cui il cerchio più interno (quindi il più piccolo) è riempito con il massimo valore attribuibile ad un pixel chunky (ovvero 255), mentre il cerchio più esterno (il più grande) è riempito con il minor valore attribuibile al pixel chunky (che sarebbe 0). E' possibile utilizzare la stessa texture che si applica nel reflection mapping per simulare una sorgente di luce. Un buon esempio di questo tipo di immagini è una sfera (di dimensioni arbitrarie) più luminosa al centro e resa scura ai bordi.

Consideriamo di utilizzare 3 variabili (o registri) che puntano rispettivamente al buffer dove salvare la bump-texture (che può anche essere lo schermo chunky), alla texture che rappresenta la luce e alla tabella precalcolata. Per ogni byte del buffer della texture da calcolare noi preleviamo sequenzialmente un valore dalla tabella precalcolata (ossia un valore della tabella per ogni byte del buffer). In seguito utilizziamo questo valore come offset da applicare al puntatore della texture che rappresenta la luce (ovvero addizioniamo al valore della tabella il puntatore della texture coincidente la sorgente di luce) e copiamo il relativo byte nel buffer destinazione. Dopo aver svolto il tutto per ogni pixel avremo ottenuto il bump mapping.

19.22 Notazione in virgola fissa

Nell'elaborazione di oggetti 3D si ha spesso a che fare con numeri reali, non interi. La maggior parte dei linguaggi evoluti permettono la diretta manipolazione di tali numeri, casomai sfruttando un eventuale coprocessore matematico oppure emulandoli via software. L'emulazione che apportano gli attuali compilatori risulta decisamente lenta per applicazioni in tempo reale, inoltre se si lavora in Assembly non si ha a disposizione la diretta gestione di numeri reali a meno che non venga utilizzato il coprocessore matematico. In alternativa alla FPU (che sfrutta numeri in virgola mobile) si può optare per il formato in virgola fissa che, nonostante una minore precisione rispetto al formato in virgola mobile, rimane la scelta migliore in quanto le operazioni svolte in tale formato risultano più veloci. In linea di principio, in un elaboratore elettronico, tutti i numeri (anche quelli reali) vengono rappresentati come interi, la notazione in virgola fissa si basa sulla diretta semplificazione di tale rappresentazione, vediamo come.

Un numero reale viene rappresentato come quel valore intero dato dal prodotto del numero reale moltiplicato per una costante definita a priori. E' proprio da questa costante che dipende la precisione con la quale possono essere rappresentati numeri non interi. Ecco un esempio:

```
3.25          <- numero reale
256           <- costante

3.25*256 = 832  <- 3.25 in virgola fissa
```

In questo modo possiamo rappresentare tutti i numeri reali con un discreto margine di errore che per le nostre applicazioni risulta praticamente ininfluente. È conveniente utilizzare come costante una potenza di 2 (es.: 256, 65536), con la quale è possibile velocizzare la manipolazione di numeri in tale notazione. Difatti è noto che il computer rappresenta un qualsiasi numero come una sequenza di bit, quindi, utilizzando come costante una potenza di 2, è possibile definire 2 campi di bit per ogni cifra: una dedicata alla parte intera mentre l'altra dedicata alla parte frazionaria. Se un numero in virgola fissa ha un numero di bit dedicati alla parte intera uguale ad $\langle a \rangle$ e un numero di bit dedicati alla parte frazionaria uguale a $\langle b \rangle$, allora si dice che quel numero è nel formato $a : b$. Bisogna inoltre specificare che la parte intera di una cifra in virgola fissa appartiene al campo di bit più alto, mentre la parte frazionaria appartiene al campo di bit più basso.

```
3.25          <- numero reale
256=2^8       <- costante
832           <- 3.25 in virgola fissa (ovvero 3.25*256)
3:8          <- formato del numero in virgola fissa. Se
               utilizziamo 1 word (16 bit) abbiamo gli 8 bit
               più significativi dedicati alla parte intera e
               gli altri 8 bit meno significativi per quella
               frazionaria.
```

Vediamo come convertire un numero intero nel formato in virgola fissa e viceversa:

```
numero intero      = (numero virgola fissa) / (costante)
numero virgola fissa = (numero intero)      * (costante)
```

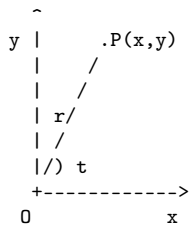
Infine occupiamoci di comprendere come effettuare le 4 operazioni con tali numeri:

```
(a:b) + (c:d) = impossibile!!
(a:b) + (a:b) = a:b
(a:b) * (c:d) = (a+c):(b+d)
(a:b) / (c:d) = (a-c):(b-d)
```

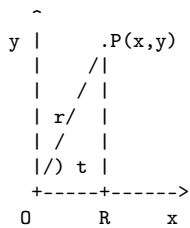
Ci accorgiamo subito che risulta impossibile sommare 2 numeri in virgola fissa di diverso formato, bisogna prima rendere omogenee le 2 cifre (significa che le 2 cifre devono avere lo stesso formato). Da notare che una qualsiasi cifra intera può essere intesa come un numero in virgola fissa nel formato "a:0"; quindi è possibile svolgere direttamente moltiplicazioni e divisioni tra numeri in virgola fissa ed interi.

19.23 Coordinate polari

Come già sappiamo, per rappresentare un generico punto su di un piano possiamo utilizzare gli assi cartesiani. Le componenti x ed y non rappresentano altro che le proiezioni del nostro punto sull'ascissa e sull'ordinata. Immaginiamo invece di voler indicare un punto utilizzando un altro sistema di riferimento, nel nostro caso le coordinate polari.



Consideriamo r la distanza tra il punto P e l'origine, mentre t come l'angolo compreso tra il segmento OP ed il semiasse positivo x . E' possibile indicare qualsiasi punto utilizzando queste 2 variabili (r e t), le quali rappresentano proprio le coordinate polari. Per ogni $P(x,y)$ corrisponde un $P'(r,t)$. Vediamo come effettuare queste conversioni.



Sia $R(x,0)$ la proiezione di $P(x,y)$ sull'asse x (ovvero il punto di ordinata 0 e di equivalente ascissa di P). Il triangolo ORP è rettangolo in R , quindi per il teorema di Pitagora:

$$r = OP = \sqrt{x^2 + y^2}$$

Possiamo anche affermare che:

$$\begin{aligned} PR &= r \sin(t) \Rightarrow \sin(t) = PR/r \\ OR &= r \cos(t) \Rightarrow \cos(t) = OR/r \end{aligned}$$

Se facciamo attenzione possiamo affermare che (considerando il punto $P(x,y)$) $PR = y$ e $OR = x$. La tangente di un angolo è equivalente al rapporto del seno di quell'angolo col relativo coseno, quindi:

$$\begin{aligned} \tan(t) &= \sin(t)/\cos(t) = (PR/r)/(OR/r) = PR/OR = x/y \\ x/y &= \tan(t) \end{aligned}$$

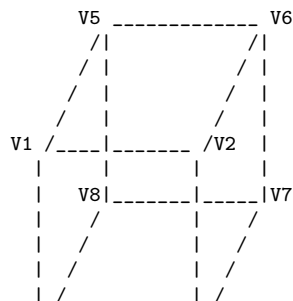
$$\begin{aligned} t &= \arctan(x/y) \\ r &= \sqrt{x^2 + y^2} \end{aligned}$$

$$\begin{aligned} x &= r \cos(t) \\ y &= r \sin(t) \end{aligned}$$

Ora sappiamo come convertire le coordinate cartesiane in polari e viceversa, ciò risulterà utile per comprendere come effettuare la rotazione di un punto.

19.24 Gestione degli oggetti

Vogliamo tracciare il nostro oggetto su schermo sia esso in wireframe, gouraud shading, texture mapping o in altra tecnica di rendering che più ci va a genio; sappiamo perfettamente come visualizzare una singola faccia, ma come potremmo gestire tutte le facce che compongono il solido tridimensionale? Bisogna definire un "formato" con cui l'oggetto risiede in memoria, in base al quale potremo visualizzare qualsiasi figura 3D utilizzando sempre le stesse routine che compongono il nostro motore 3D. Iniziamo col vedere un esempio pratico:



teniamo conto di dover definire come oggetto un semplice cubo, per prima cosa potremmo indicarne il numero di vertici e di facce di cui è composto; in seguito elenchiamo tutte le coordinate (x,y,z) dei vertici del cubo; infine indichiamo le caratteristiche di tutte le facce. Nel caso più semplice per definire una faccia basta indicarne i vertici (casamai ordinati in senso orario per facilitare la rimozione dell'hidden face e il calcolo della

|/-----|/ normale). Ecco come è possibile
V4 V3 definire in memoria un cubo:

```

8            <- numero di vertici dell'oggetto
6            <- numero di facce dell'oggetto
-50,-50,-50 <- coordinate x,y,z del vertice V1
+50,-50,-50 <- coordinate x,y,z del vertice V2
+50,+50,-50 <- coordinate x,y,z del vertice V3
-50,+50,-50 <- coordinate x,y,z del vertice V4
-50,-50,+50 <- coordinate x,y,z del vertice V5
+50,-50,+50 <- coordinate x,y,z del vertice V6
+50,+50,+50 <- coordinate x,y,z del vertice V7
-50,+50,+50 <- coordinate x,y,z del vertice V8
1,2,3,4      <- indici al buffer dei vertici della faccia 1
2,6,7,3      <- indici al buffer dei vertici della faccia 2
6,5,8,7      <- indici al buffer dei vertici della faccia 3
5,1,4,8      <- indici al buffer dei vertici della faccia 4
5,6,2,1      <- indici al buffer dei vertici della faccia 5
4,3,7,8      <- indici al buffer dei vertici della faccia 6

```

Analizziamo come vengono definiti i poligoni, prendiamo la faccia 1:

```

1,2,3,4      <- significa che la faccia è composta dai primi 4
               punti posti nella lista dei vertici, ovvero:

-50,-50,-50 <- coordinate x,y,z del vertice V1
+50,-50,-50 <- coordinate x,y,z del vertice V2
+50,+50,-50 <- coordinate x,y,z del vertice V3
-50,+50,-50 <- coordinate x,y,z del vertice V4

2,6,7,3      <- significa che la faccia è composta dai lati
               delimitati dai punti 2 e 6, 6 e 7, 7 e 3 e dai
               punti 3 e 2.

```

Ogni lato è rappresentato dalla congiunzione lineare di 2 vertici, nell'esempio appena proposto i lati della faccia 1 sono i segmenti delimitati dal primo al secondo punto ($V1eV2$), dal secondo al terzo ($V2eV3$), dal terzo al quarto ($V3eV4$) e dal quarto al primo punto ($V4eV1$). Nel nostro caso ogni faccia è un quadrilatero, naturalmente è possibile realizzare il proprio motore 3D che utilizzi un diverso numero di lati, l'importante è che ogni poligono sia convesso, altrimenti l'algoritmo di scan conversion risulterebbe decisamente più complesso di quello studiato nel paragrafo relativo al fill e scan line.

Parte IV

Appendici



MATEMATICA

Autore: Antonello Mincone

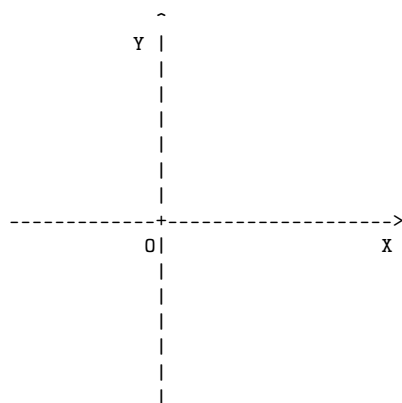
Per tutti quelli che aspirano a fare giochi tipo DOOM, ELITE, o comunque un qualsiasi gioco che richieda l'uso di poligoni, o addirittura di quella che è uno degli effetti ormai diventati uno standard nei DEMOS, cioè il texture-mapping, credo sia essenziale sapere alcune di quelle che sono le formule basilari di matematica, in particolare quelle relative alla geometria analitica e alla trigonometria.

Se a scuola non avete mai digerito questi argomenti, o semplicemente non li avete mai affrontati e ne avete sempre sentito parlare come qualcosa di incredibilmente complicato, posso assicurarvi che non è assolutamente vero. La vera difficoltà sta nel seguire attentamente l'argomento, ma soprattutto nel capirlo.

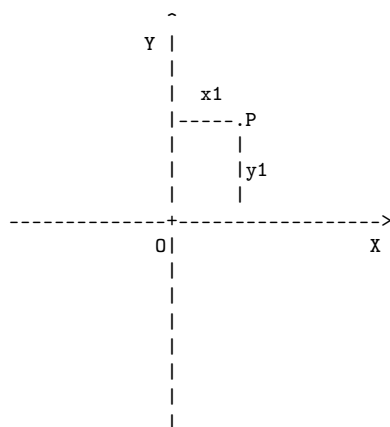
In pratica il consiglio che vi do è di fare (anche in base alle conoscenze che avete) poco alla volta sia con questa lezione che con il corso in generale, dato che in realtà potete possedere tutti i libri di informatica e matematica di questo mondo, ma questi costituiscono solo la base essenziale (ma non sufficiente) per scrivere un programma valido: si diventa bravi soprattutto con l'esperienza, provando e riprovando le routine, modificandole, facendo insomma degli esperimenti.

In base a ciò ho deciso di scrivere quest'articolo non insegnandovi una particolare tecnica di 3D (anche perché per quella c'è una lezione apposta), ma le basi che vi permetteranno di ricavarvi da soli le formule anche solo per farvi la tabella pre-calcolata adatta alle vostre esigenze. Partirò quindi da zero (beh non proprio, dato che spero conosciate le quattro operazioni fondamentali, anche perché se no, non dovete fare il corso ma le elementari), non si stupisca quindi tutta quella gente che conosce già l'argomento e magari avrebbe qualcosa da obbiettarci sullo stile di esposizione.

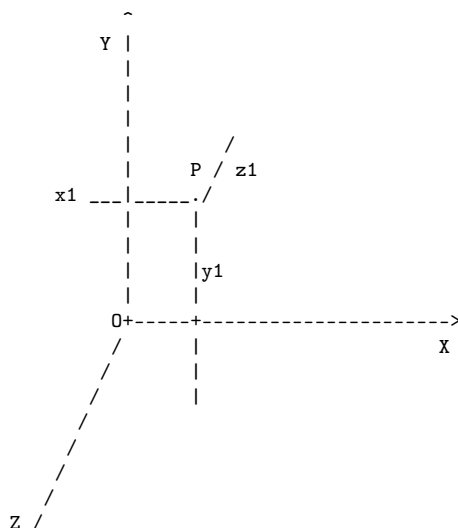
Inanzi tutto parliamo di quello che viene chiamato dagli esperti in materia *sistema di riferimento cartesiano ortogonale*. In realtà sotto questo nome si nasconde una cosa semplicissima: due comuni rette (si dice RETTA una linea di cui non c'è né un inizio e né una fine, si dice SEMI-RETTA una linea che ha un inizio ma non una fine, si dice SEGMENTO una linea che ha un inizio e una fine) disposte in modo che, incrociate tra loro, formino quattro angoli di 90 gradi, che hanno convenzionalmente i nomi di X e Y. In pratica disposte così:



Il punto O corrisponde all'incrocio degli assi ed è chiamato ORIGINE. Gli assi X e Y sono chiamati rispettivamente asse delle ascisse e asse delle ordinate, e servono a darci dei riferimenti ad ogni punto, che si ottengono tracciando la parallela all'asse X e quella all'asse Y, e che in pratica ci dicono l'altezza del punto rispetto ad X e la sua distanza da Y, che esprimeremo scegliendo un'unità di misura e vedendo quante volte questa unità entra nella misura considerata: quando cioè vogliamo indicare dove si trova un determinato punto dovremmo definire la sua distanza con l'asse X e con l'asse Y (più precisamente la sua ordinata e la sua ascissa). Immaginiamo ad esempio di avere un punto P :



Il segmento contrassegnato con x_1 indica la sua ascissa mentre quello contrassegnato con y_1 la sua ordinata. Tralascieremo l'unità di misura dato che con l'AMIGA userete il pixel. Badate bene che quando il punto si trova a destra dell'asse Y l'ascissa è positiva (è cioè maggiore di 0), mentre quando si trova a sinistra è negativa (è cioè minore di 0); caso limite quando si trova sull'asse Y, dove sarà uguale a 0. Allo stesso modo per l'asse X: quando il punto si trova sopra di questo l'ordinata è positiva, sotto è negativa, sull'asse x è 0. Se però rapportiamo il tutto alla realtà, ci accorgiamo che due dimensioni non sono sufficienti, dato che tutti gli oggetti, hanno, oltre ad una larghezza e una larghezza, anche una profondità. Abbiamo quindi bisogno di una terza dimensione, appunto la profondità, che ci permette di rapportare anche il singolo punto allo spazio. Un grafico quindi che ci voglia dare un completo quadro di un oggetto nello spazio sarà di questo genere:



Il nuovo asse, Z, indica la nuova dimensione. È da tener presente che, nello spazio, l'angolo situato tra l'asse X e l'asse Z e quello tra l'asse Y e l'asse Z, è retto, cioè di 90 gradi, che purtroppo si deformano in proiezione assonometrica (che è appunto quella con cui sono rappresentati tutti i grafici).

Finora abbiamo parlato solo di punti, mentre lo spazio che ci circonda è fatto da oggetti ben più complessi, che in genere sono fatti da linee che uniscono gli spigoli che li determinano. Molte volte è necessario dover rappresentare oggetti curvilinei, quali il semplice cerchio o curve più complesse, determinate da complicate formule goniometriche: in questi casi con l'AMIGA consiglio sempre di ridurre la curva a un poligono, magari anche con 20 lati, ma che risulta certamente più veloce da disegnare e da calcolare in rotazioni o traslazioni (movimenti che comportano solo lo spostamento), comunque troverete alla fine della lezione le formule delle più importanti curve.

Per unire i punti dei poligoni generalmente si può ricorrere alla funzione LINE del blitter, ma non sempre questa è la più veloce, e potrebbe essere necessario affidare questo lavoro al processore: è quindi utile conoscere le formule principali legate al tracciamento delle rette. Iniziamo col dire che ogni retta è individuata sugli assi cartesiani dalla formula:

$$Y = m \cdot X + q$$

La formula in questione ci dà le ordinate di tutti i punti della retta a seconda della sua ascissa, basta insomma sostituire la x con un qualsiasi valore per ottenere la corrispondente y della retta. I termini m e q che compaiono nella formula sono delle costanti: la prima m è detto coefficiente angolare e determina l'angolo che la retta forma con l'asse X (più precisamente è la tangente di quell'angolo, ma affronteremo questo argomento più avanti), più grande sarà m e più grande sarà l'angolo formato; q determina invece il punto dove la retta incrocia l'asse y, in sostanza il punto della retta che ha coordinate: (0,q), da ciò è facile capire che se q=0, allora la retta passa per l'origine degli assi.

Esiste poi una formula che a mio parere è importantissima per fare un programma 3D. Dato un punto P1 di coordinate (P1x,P1y), e un punto P2 di coordinate (P2x,P2y), possiamo ricavare la retta passante per questi due punti con la formula:

$$Y - P1y = (P2y - P1y) / (P2x - P1x) * (X - P1x)$$

Dalla formula in questione ricaviamo che:

$$Y = (P2y-P1y)/(P2x-P1x)*X + (-P1x*(P2y-P1y)/(P2x-P1x))+P1y$$

Questa è appunto la formula della retta passante per i punti presi in considerazione. Il termine che compare prima della X corrisponde alla m, mentre tutta la formulaccia che compare dopo sarebbe la q, ma naturalmente questo calcolo sarà fatto una sola volta per ogni retta. Applicazioni della stessa formula possono servire ad esempio, per una linea che esce dallo schermo: avendo gli estremi di questa possiamo trovare la formula che la determina e quindi, sostituendo alla X la ascissa dei bordi dello schermo, possiamo trovare le coordinate dei punti estremi del segmento visibile.

Altre formule che spesso si rivelano utili conoscendo le coordinate di due punti P1 (P1x,P1y) e P2 (P2x,P2y) sono:

1. Quella per trovare la loro distanza (che è in pratica un' applicazione del teorema di Pitagora che troverete più avanti):

$$\text{distanza} = \text{sqr}((P2x-P1x)^2 + (P2y-P1y)^2)$$

(sqr non è altro che l'istruzione usata dalla maggior parte dei linguaggi di alto livello per indicare la radice quadrata, mentre il simbolo ^ significa elevato: in questo caso leggerete quindi la distanza è uguale alla radice quadrata della differenza delle ascisse elevata al quadrato sommata alla differenza delle ordinate elevata al quadrato, in simboli:

$$\sqrt{(P2x-P1x)^2 + (P2y-P1y)^2}$$

Cercate di capire bene Sqr e perché in seguito li riuseremo spesso). Questa formula è utile ad esempio per trovare la lunghezza di un lato di un qualsiasi poligono conoscendo i due spigoli.

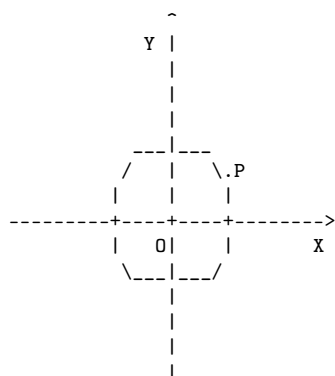
2. Conoscendo i soliti due punti P1(Px1,Py1) e P2(Px2,Py2) possiamo trovare le coordinate del punto medio M(XM,YM) con la formula:

$$XM = (Px1 + Px2)/2$$

$$YM = (Py1 + Py2)/2$$

A questo punto direi che potete anche fare la lezione di prospettiva, dato che ora siete capaci di rappresentare un qualunque oggetto nello spazio (basta infatti disegnarvi gli spigoli e unirli col Blitter in modo da formare una figura piana o un solido). Badate bene però che con queste conoscenze non potete ancora ruotare gli oggetti, ma solo zoomarli (per far questo basta aumentare o diminuire la Z di ogni punto).

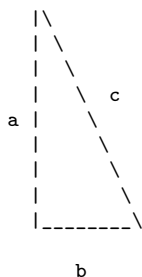
Per ruotare un punto dobbiamo entrare nella trigonometria introducendo il coseno e il seno. Questi due non sono altro che l'ascissa e l'ordinata di un punto che ha la caratteristica di trovarsi su una circonferenza che ha come centro l'origine.



Anche se quello che ho disegnato è un ottagono irregolare (ma che ci volete fare con i caratteri ASCII non sono riuscito a fare di meglio), con un pò di fantasia dovreste avere un'idea di quello che voglio dire. In breve il coseno è la distanza del punto P con l'asse Y, mentre il seno la distanza del punto P con l'asse X. Per convenzione (ma non solo per quello) il raggio del cerchio viene considerato pari ad 1. In questo modo sia il seno che il coseno oscilleranno sempre tra valori compresi tra 1 e -1 (in sostanza numeri decimali). Da notare inoltre che il punto P individua anche un angolo sulla circonferenza, l'angolo cioè formato tra l'asse X e la retta passante per il punto P e l'origine degli assi.

Se ad esempio diciamo che il seno di 30 gradi è 0.5, ciò significa che il punto P che, unito con O (origine degli assi), che forma con l'asse X un angolo di 30 gradi, dista dall'asse X 0.5. Per trovarci anche il coseno dell'angolo considerato possiamo fare una semplice osservazione, basandoci sul teorema di Pitagora (non il Coder). Per chi non conosce questo, che è uno dei principali teoremi di geometria ecco qui una veloce spiegazione:

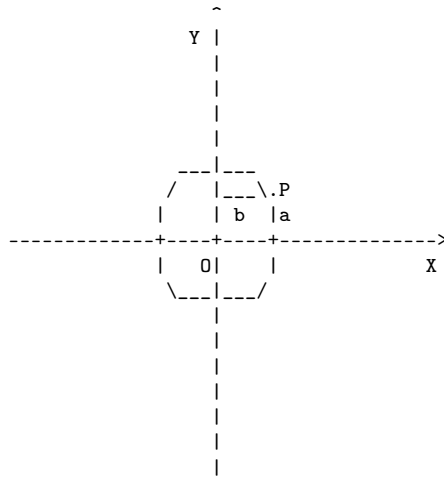
Definizione 1. dato un triangolo rettangolo (che ha cioè un angolo di 90 gradi), sapendo la lunghezza dei due cateti (che sarebbero i lati più corti), possiamo trovare l'ipotenusa (il lato più lungo), sapendo che questa è uguale alla radice quadrata della somma dei quadrati dei due cateti.



In questo caso a e b sono i cateti, per trovare c dovremo calcolare la radice quadrata di $a * a + b * b$ (che possiamo anche scrivere $a^2 + b^2$). In generale quindi:

$$c^2 = a^2 + b^2$$

Tornando alla circonferenza che stavamo considerando, notiamo anche qui la presenza di un triangolo rettangolo che ha come cateti l'ascissa e l'ordinata del punto P, e come ipotenusa il segmento OP, che in pratica è uguale al raggio, e quindi ad 1. Nell'esempio di prima, dove conosceamo il seno di 30 gradi, possiamo trovare il corrispondente coseno (che sarebbe in pratica l'ascissa):



In questo caso infatti $a=0.5$ ed OP (che non ho disegnato per motivi grafici) è uguale ad 1. Essendo quindi l'angolo tra a e b di 90 gradi, sostituendo i termini noti nell'equazione precedente abbiamo che :

$$1^2 = 0.5^2 + b^2$$

Sostituendo a 0.5 la forma frazionaria $1/2$ possiamo scrivere:

$$1 = 1/2^2 + b^2$$

Da cui :

$$1 = 1/4 + b^2$$

Quindi:

$$b^2 = 1 - 1/4$$

$$b^2 = 3/4$$

Possiamo concludere che $b = \sqrt{3}/2$ ($\sqrt{}$ non è altro che l'istruzione usata dalla maggior parte dei linguaggi di alto livello per indicare la radice quadrata, nel nostro caso leggete quindi b è uguale a radice di 3 fratto 2)



LICENZA PER DOCUMENTAZIONE LIBERA GNU

GNU Free Documentation License
Version 1.3, 3 November 2008

Copyright (C) 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc.
<<http://fsf.org/>>

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be

distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material

this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy

a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all

the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document

except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in

part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.
A copy of the license is included in the section entitled "GNU
Free Documentation License".
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

```
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.