



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza
dell'Informazione

Corso di Laurea in
Informatica

DISPENSA DI CALCOLATORI
Dal materiale dei prof. Iacca, Palopoli e Abeni

Autori

Francesco Bozzo
Samuele Conti
Filippo Daniotti

Revisori

Emanuele Nardi
Michele Yin

Anno accademico 2018-2019

Indice

1	Introduzione ai calcolatori	1
1.1	Un po' di storia dell'informatica	1
1.1.1	Differenziazione dei calcolatori	2
1.2	Le prestazioni dei calcolatori	2
1.2.1	La memoria e il processore	3
1.2.2	Software di sistema	4
1.2.3	Periferiche di I/O	4
1.2.4	Misurazione delle prestazioni	5
1.2.5	La legge di Moore	6
1.2.6	La barriera dell'energia	6
2	Aritmetica dei calcolatori	7
2.1	Informazione nei computer	7
2.1.1	I transistor	7
2.1.2	Interpretazione delle informazioni	8
2.2	I numeri	8
2.2.1	Regole di conversione tra basi	8
2.2.2	I naturali	9
2.2.3	Gli interi	10
2.2.4	Aritmetica modulare	12
2.2.5	I reali	13
3	Codifica del testo	17
3.1	La codifica ASCII	17
3.2	La codifica ASCII estesa	17
3.3	La codifica UNICODE	17
4	Le reti logiche	19
4.1	Introduzione	19
4.2	L'algebra di Boole	19
4.2.1	Regole di semplificazione	20

4.2.2	La tavola di verità e i mintermini	20
4.3	Le porte logiche	21
4.4	Alcuni circuiti degni di nota	21
4.5	Il concetto di costo	24
4.6	Le reti sequenziali	25
5	Introduzione al linguaggio Assembly	27
5.1	Perché esiste Assembly	27
5.2	Come funziona Assembly	27
5.2.1	L'Assembler	27
5.2.2	Instruction Set Architecture	28
5.2.3	Funzionamento di una CPU	28
5.2.4	Diversi tipi di ISA	29
5.2.5	I tipi di istruzioni	30
5.2.6	Confronto tra CISC e RISC	30
5.2.7	Accesso alla memoria	31
5.2.8	Application Binary Interface	31
6	L'architettura MIPS	33
6.1	Breve riepilogo su ISA	33
6.2	Operazioni aritmetiche	34
6.3	I registri	35
6.4	La rappresentazione delle istruzioni	37
6.4.1	Le istruzioni register	38
6.4.2	Le istruzioni immediate	39
6.5	Istruzioni aritmetico-logiche	40
6.5.1	Operazioni di shift	40
6.5.2	Operazioni bitwise	41
6.6	Salti	43
6.7	Le procedure	46
6.7.1	Protocolli di chiamata MIPS	47
6.7.2	Gestione della memoria in MIPS	48
6.7.3	Svolgimento di una procedura in MIPS	49
6.7.4	Gestione delle variabili in MIPS	52
6.7.5	Elaborazione delle procedure	54
6.8	Le stringhe in MIPS	55
6.8.1	Possibile implementazione	56
6.8.2	Implementazione realistica	57

7 L'architettura Intel x86	59
7.1 Introduzione	59
7.2 La gestione dei registri	60
7.2.1 I registri general purpose	60
7.2.2 I registri specializzati	60
7.3 Le convenzioni di chiamata	61
7.4 Le modalità di indirizzamento	61
7.5 La sintassi delle istruzioni	62
7.6 Alcune istruzioni frequenti	63
7.6.1 Load Effective Address (LEA)	64
7.6.2 Incremento e decremento	64
8 L'architettura ARM	65
8.1 Introduzione	65
8.2 Registri e relativo utilizzo	65
8.3 Le convenzioni di chiamata	66
8.4 Le istruzioni ARM	66
8.4.1 Possibili operazioni sull'operando <r*>	68
8.5 Modalità di indirizzamento in ARM	69
9 Programmi Assembly comparati	71
9.1 Semplici espressioni aritmetico-logiche	71
9.2 Array e accesso alla memoria	72
9.3 Blocchi condizionali	73
9.4 Cicli	76
9.5 Invocazione di subroutine	79
9.6 Copia stringa	82
10 Toolchain	85
10.1 Introduzione	85
10.2 Processo: dal codice sorgente al file oggetto	86
10.3 Processo: dal file oggetto all'eseguibile	88
10.3.1 Linker e simboli	88
10.4 Linking visto approfonditamente	89
10.5 Librerie	89
10.5.1 Funzionamento di una libreria dinamica	90
10.6 Esempio finale	90
11 Central Processing Unit	94
11.1 Introduzione	94
11.2 Arithmetic-Logic Unit	94

11.3 Il datapath	95
11.3.1 Struttura del datapath	95
11.3.2 Il ruolo del multiplexer	96
11.4 La Control Unit	97
11.5 La temporizzazione	98
11.5.1 Breve riepilogo sulle reti logiche	98
11.5.2 Gestione della temporizzazione	99
11.6 Elaborazione delle istruzioni	100
11.6.1 Istruzioni R	100
11.6.2 Istruzioni di accesso alla memoria	101
11.6.3 Istruzioni di salto condizionato	102
11.7 Prima progettazione completa di una CPU	103
11.7.1 Progettazione dell'unità di controllo della ALU	104
11.7.2 Progettazione dell'unità di controllo principale	106
11.7.3 Esecuzione delle principali istruzioni	107
12 Pipeline	112
12.1 Introduzione	112
12.2 Esempio	113
12.2.1 Osservazioni	115
12.3 Vantaggi del RISC	116
12.4 Condizioni di Hazard	116
12.4.1 Hazard strutturale	117
12.4.2 Hazard sui dati	117
12.4.3 Hazard sul controllo	118
12.4.4 Registri di backup	119
12.5 Esercizio tipo	120
13 La gerarchia di memoria	121
13.1 Introduzione	121
13.2 La memoria principale	123
13.2.1 Le RAM	123
13.2.2 Le Static Random Access Memory (SRAM)	125
13.2.3 Le Dynamic Random Access Memory (DRAM)	126
13.3 Gestione della memoria	130
13.3.1 Struttura della gerarchia	131
13.4 La cache	132
13.4.1 Cache a mappatura diretta	133
13.4.2 La cache in MIPS	134
13.4.3 La ricerca di un compromesso	136
13.4.4 Gestione delle miss	137

13.4.5 Scritture	137
13.5 Cache associative	139
13.5.1 Cache completamente associativa	139
13.5.2 Cache set-associativa	139
13.6 Mappatura del blocco	141
13.7 Vantaggi dell'associatività	143
14 I/O	145
14.1 La necessità di comunicare	145
14.2 Connessione tra processore e periferiche	147
14.2.1 Bus sincrono	148
14.2.2 Bus asincrono	149
14.3 Gestione delle periferiche da parte del SO	152
14.3.1 Come impartire comandi ai dispositivi	153
14.4 Polling	154
14.4.1 Considerazioni finali sul polling	156
14.5 Input ad interruzione di programma	157
14.5.1 Considerazioni finali sull'interrupt driven I/O	159
14.6 Eccezioni	159
14.6.1 Supporto del MIPS per la gestione di eccezioni	160
14.6.2 Modifiche al processore per gestire le eccezioni	161
A Istruzioni e registri MIPS	163
B Come compilare Assembly Intel	168
B.1 Prerequisiti	168
B.1.1 Come utilizzare GCC	168
B.1.2 Come utilizzare nasm e ld	168
B.2 Un po' di esempi	169
B.2.1 Hello World	169
B.2.2 Stampa di una stringa di lunghezza variabile	169
B.2.3 Esempio di subroutine	170
C Ancora sui programmi Assembly	172
C.1 Ordinamento di array	172
C.2 Funzione 2^n	179
C.3 Successione di Fibonacci	182
C.4 Serie tail recursive	185

Introduzione

Quella che il lettore sta approcciando è una dispensa che raccoglie, riorganizza e riesponde l'intero contenuto del corso di *calcolatori* tenuto nell'Università degli studi di Trento durante il secondo semestre dell'anno accademico 2018/2019. La repository del progetto può essere raggiunta tramite il seguente link: <https://github.com/Samaretas/Appunti-Calcolatori>.

Le risorse utilizzate sono state le slides fornite dal professor Giovanni Iacca (corso dispari) durante le lezioni e integrate dalle sue spiegazioni, la dispensa "Piccola introduzione al linguaggio Assembly e simili amenità" del professor Luca Abeni, le lezioni dell'esercitatore Fabiano Zenatti e svariate risorse online per dirimere alcune lievi questioni non meritevoli del disturbo ai professori. L'elaborato è stato suddiviso in capitoli che seguono il partizionamento proposto dal professore e, successivamente, in sezioni e sottosezioni per questioni di agilità consultativa. I singoli argomenti vengono presentati nell'ordine e modalità proposti dal professore (all'infuori di sporadici casi in cui gli autori hanno proposto una maniera da loro ritenuta più fruibile per il target di riferimento), corredati da immagini e codici commentati dove possibile.

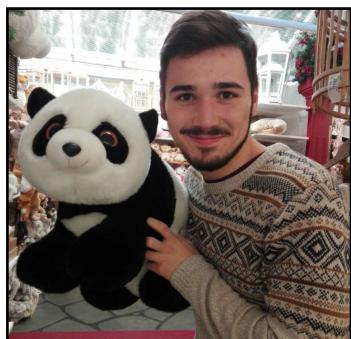
Nel momento della stesura questa dispensa si propone di essere autoconsistente nell'ottica della preparazione dell'esame di fine corso, ossia lo studente potrebbe potenzialmente usare questo testo come unica risorsa ed essere ampiamente in grado di superare con successo l'esame. Quest'ultimo, alla data attuale, consiste in 12 domande a risposta multipla, ciascuna delle quali del valore di 2.75 punti, per un totale di 33. Si tenga presente che ogni risposta errata comporta una decurtazione di 0.55 punti dal totale; risposte considerate nulle non comporteranno alcuna penalità.

Nota bene: a seguito di alcune modifiche apportate al programma del corso per l'anno accademico 2019-2020, questo elaborato non può più dirsi autoconsistente. Gli autori invitano pertanto i futuri studenti del corso che volessero contribuire ad aggiornare la dispensa a contattarli per ricevere istruzioni su come partecipare al progetto.

Ecco una breve presentazione degli autori; per raggiungere il profilo GitHub di ciascuno è sufficiente cliccare sul relativo nome; ognuno dei tre, a turno, ha curato

l'esposizione dei singoli capitoli.

- *Francesco Bozzo*: leader e responsabile del progetto, ha frequentato il liceo scientifico indirizzo scienze applicate G.B. Quadri di Vicenza e successivamente si è iscritto all'Università di Trento, corso di laurea in informatica. Nel progetto si è dedicato principalmente alla parte più tecnica, risolvendo ogni volta qualsiasi problema L^AT_EX potesse presentare, il che gli ha giustamente procurato il titolo di campione della Lega Pokémon della regione di Trentoh.



- *Samuele Conti*: ha frequentato scienze applicate a Verona, quindi non un liceo vero, e da un anno a questa parte frequenta l'Università di Trento dove ha raggiunto la notorietà per mezzo delle sue memorabili sbronze. Particolaramente notevoli i suoi errori di ortografia e la sua incapacità di formulare pensieri in modo ordinato. Tutto sommato però è un bravo ragazzo, ve lo raccomando.

- *Filippo Daniotti*: dopo aver frequentato il liceo classico Antonio Scarpa, si è iscritto al corso di laurea in informatica presso l'Università di Trento, già da questa premessa si deduce quindi il suo forte carattere masochista. Ha curato la revisione ortografica e morfosintattica del progetto, il commento dei codici e la stesura di quest'introduzione. Inoltre ha sempre dimostrato una grande passione per l'utilizzo del tool L^AT_EX (qui ritorna il carattere masochista) e molti ritengono che per questa ragione sia il classico tipo che alle feste rimane solo in un angolino a compiangere se stesso.



Capitolo 1

Introduzione ai calcolatori

“ ...noi che depositeremo in un computer non solo le nostre conoscenze, ma anche memorie, ricordi, emozioni. Addirittura la nostra consapevolezza. E così potremo vivere in eterno. Un’idea non solo assurda, sbagliata, ma che può portare a sviluppi pericolosi: se pensiamo di essere macchine, ci comporteremo come macchine. ”

Federico Faggin, 2017

1.1 Un po’ di storia dell’informatica

ENIAC, il primo computer della storia Commissionato nel 1946 dal Dipartimento di Difesa degli Stati Uniti, Electronic Numerical Integrator and Computer (ENIAC) è diventato il primo calcolatore della storia. Dotato di 18000 valvole termoioniche, esso occupava una stanza di 9×30 metri, consumando un ammontare spropositato di energia. Il suo impiego principale consisteva nel calcolare traiettorie dei proiettili di artiglieria. Infatti, è doveroso specificare come i primi calcolatori della storia sono stati concepiti per essere sfruttati in applicazioni belliche.

Apollo Guidance Computer Prodotto da IBM nel 1969, disponeva di 2800 circuiti integrati, un processore da 0.043 MHz e 152 KByte complessivi di memoria ROM/RAM. Presentava un’interfaccia display & keyboard: i comandi utilizzavano una sintassi del tipo «verbo + nome».

Programma 101 Nel 1964 l’Olivetti rilasciò il primo *personal computer* della storia che, sfortunatamente, non ebbe successo.

1.1.1 Differenziazione dei calcolatori

Seppur i calcolatori di oggi condividano la stessa idea di base, le soluzioni per ciascuna tipologia di applicazione sono piuttosto diverse. A seguire alcuni esempi di diverso tipo di calcolatori:

Calcolatore	Costo	Prestazioni	Applicazioni
Personale	Ridotto	Buone	Software prodotti da terze parti
Server	Elevato	Ottimali	Poche ma complesse (calcolo scientifico) o tante ma semplici (web server)
Embedded	Minimo	Minimi	Dedicate e molto specifiche; esiste un vasto spettro di scopi possibili

Per ovvi motivi, ogni tipologia di calcolatore è meglio adatta per differenti scopi. Ottenere delle buone prestazioni senza eccedere in prezzo e potenza è ciò che decreta il successo commerciale di un prodotto.

1.2 Le prestazioni dei calcolatori

Un buon programmatore, oltre a saper utilizzare degli efficienti paradigmi, deve comprendere la gerarchia di memoria e il concetto di parallelismo; conoscere a fondo il calcolatore è fondamentale. Fino a qualche tempo fa, le prestazioni di qualsiasi calcolatore erano in balia della disponibilità di memoria. Al giorno d'oggi invece risulta un problema risolto, tranne per qualche criticità per le applicazioni embedded. Ecco una lista degli elementi che influenzano le prestazioni del calcolatore e il loro ruolo:

- *algoritmi*: determinano il numero di istruzioni di alto livello e di operazioni di Input/Output (I/O);
- *linguaggi di programmazione, compilatori e architetture*: determinano il numero di istruzioni macchina per ogni istruzione di basso livello;
- *processore e sistema di memoria*: determina quanto velocemente è possibile eseguire ciascuna istruzione;
- *sistema di I/O (hardware e sistema operativo)*: determina quanto velocemente possono essere eseguite le istruzioni.

1.2.1 La memoria e il processore

La memoria È possibile classificare la memoria in:

- *volatile*: è costituita da vari banchi di (tipicamente) 8 chip di RAM dinamica. Il tipo più diffuso è appunto la DRAM;
- *permanente*: è costituita da memorie flash (es. *SSD*), dischi rigidi e CD/DVD.

La prima viene utilizzata per memorizzare dati e programmi mentre vengono eseguiti (per questo motivo viene chiamata anche *memoria principale*): allo spegnimento i dati vengono persi. La seconda viene usata per memorizzare grandi quantità di dati e programmi fra esecuzioni diverse.

Le memorie di massa Il principio di funzionamento dell'hard disk è di magnetizzare delle particelle metalliche distribuite su un substrato:

- i dischi sono organizzati in strutture sovrapposte (cilindri);
- le particelle vengono lette da un dispositivo meccanico (testina) che si sposta radialmente su un braccio (in grado di fare movimenti angolari);
- questa componente rallenta i tempi di accesso ma aumenta la densità di memorizzazione (è possibile arrivare facilmente ai TeraByte).

Diversamente dai dischi rigidi elettromeccanici, la memorizzazione su una memoria flash avviene intrappolando una carica elettrica in maniera permanente.

I dischi ottici funzionano sul principio di riflessione della luce: viene emesso un raggio laser che viene riflesso dai rilievi, nel caso bit 1, e assorbito dalle buche, nel caso bit 0. Nei dischi riscrivibili è inoltre presente un particolare substrato che, se riscaldato, torna alla condizione di partenza, eliminando tutti i dati memorizzati.

Il processore La Central Processing Unit (CPU) è l'unità centrale di ogni calcolatore. Si compone principalmente di:

- *datapath*: esegue operazioni aritmetiche sui dati;
- *control unit*: impedisce ordini al datapath, alla memoria e alle componenti I/O, sulla base di quanto stabilito dal programma.

Ciascun processore dispone di un'Instruction Set Architecture (ISA), ossia un'interfaccia che permette di utilizzarlo senza conoscerne i dettagli. Questa, assieme all'ulteriore interfaccia del sistema operativo, forma l'interfaccia binaria delle applicazioni Application Binary Interface (ABI). Ciò permette allo sviluppatore di svincolarsi dal livello hardware sottostante, secondo il principio di *astrazione*. Esistono due diversi tipi di architetture ISA:

- RISC: offre un numero ridotto di istruzioni, rivolta soprattutto a sistemi embedded;
- CISC: permette di usufruire di un maggior numero di istruzioni, rivolta prevalentemente ai personal computer.

1.2.2 Software di sistema

Le principali funzioni che deve svolgere un *sistema operativo*:

- gestione delle operazioni di I/O;
- allocazione della memoria;
- consentire e regolare il multitasking.

Il *compilatore* traduce da linguaggio di alto livello a linguaggio macchina. Ogni istruzione di quest'ultimo è costituita da una determinata sequenza di *bit*, ossia l'unità di base dell'informazione (0 o 1). L'utilità del compilatore consiste nel facilitare il lavoro del programmatore: non è più necessario implementare il software in codice binario, bensì è possibile utilizzare un linguaggio di programmazione più vicino al linguaggio naturale. Il primo linguaggio sviluppato per questo scopo è stato l'*Assembly*. Il software che traduce l'assembly in codice binario si chiama *Assembler*. È possibile riassumere la traduzione di un linguaggio di programmazione di alto livello in codice macchina in due semplici passi:

1. traduzione del linguaggio ad alto livello in linguaggio Assembly, per via del compilatore;
2. traduzione del linguaggio Assembly in linguaggio macchina, attraverso l'assemblatore.

Spesso fanno eccezione alcuni compilatori che trasformano direttamente il linguaggio ad alto livello in codice macchina.

1.2.3 Periferiche di I/O

Il mouse Il *mouse* è stato inventato nel 1967 da *Doug Engelbart* nei laboratori della *Xerox*. Attualmente sfrutta la tecnologia ottica: attraverso le variazioni di luce provocate da alcuni led che illuminano il piano, il mouse è in grado di rilevare gli spostamenti.

Il display LCD Gli schermi *LCD* (*Liquid Crystal Display*) sono costituiti da alcuni cristalli che galleggiano in un fluido: ciascuno di essi corrisponde ad un pixel. Attraverso un campo elettrico è possibile ruotare di 90 gradi ogni singolo cristallo, che di conseguenza può impedire o meno il passaggio della luce secondo il fenomeno fisico della *polarizzazione*.

L'immagine viene rappresentata da una matrice di pixel: ciascuno di essi ha associata una componente rossa, verde, blu (sistema *RGB*). Questa immagine viene memorizzata in un *frame buffer*, una RAM che viene aggiornata fino a 100 volte al secondo.

1.2.4 Misurazione delle prestazioni

I moderni processori sono costruiti usando un segnale periodico che ne sincronizza le operazioni. Il *periodo di clock* (detto anche ciclo di clock) è l'intervallo di tempo che intercorre tra due colpi di clock. La frequenza è definita come $\frac{1}{\text{periodo di clock}}$. Il periodo di clock è misurato in secondi, la frequenza in Hertz. Il tempo d'esecuzione di un programma dipende da tre fattori:

- il numero di istruzioni dell'algoritmo.
- i cicli di clock per istruzione (*CPI*).
- la frequenza di clock.

$$\text{Tempo CPU} = \text{N. di istruzioni} \cdot \text{CPI} \cdot \text{Periodo Clock} = \frac{\text{N. di istruzioni} \cdot \text{CPI}}{\text{Frequenza Clock}}$$

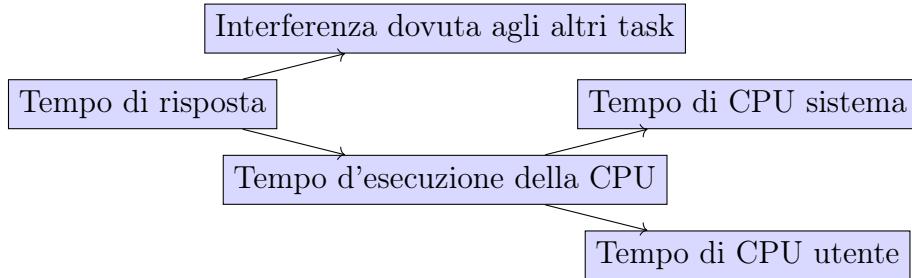


Figura 1.1: Diagramma tempo di risposta

Un algoritmo è efficiente se viene strutturato in modo da risparmiare istruzioni e, per una particolare architettura, se utilizza le istruzioni più efficienti (quelle con un basso CPI, poiché il CPI dei diversi tipi di istruzione varia in base all'architettura).

Il linguaggio di programmazione influenza il numero di istruzioni e il CPI: più di alto livello sono i costrutti, più lunghe sono le sequenze di istruzioni macchina ottenute traducendo il codice di partenza.

Il compilatore sicuramente influenza sia il numero di istruzioni che il CPI in base alla propria efficienza e ottimizzazione.

In base alla sua progettazione, anche l'ISA ha un impatto sul numero di istruzioni, sul CPI, e sulla frequenza di clock: essa può fornire istruzioni di basso o alto livello (più o meno istruzioni per eseguire un'operazione).

1.2.5 La legge di Moore

Negli scorsi anni, le prestazioni dei calcolatori sono aumentate costantemente, secondo l'andamento previsto da *Moore*. Di recente si è assistito a una diminuzione dell'incremento tra una generazione e la successiva, sintomo di un'evidente saturazione causata dai limiti fisici della materia.

1.2.6 La barriera dell'energia

Dagli anni Ottanta ad oggi, la frequenza media dei processori è incrementata di tre ordini di grandezza, con un conseguente aumento dei consumi di 30 volte.

$$\text{Potenza} = \text{capacità} \cdot \text{tensione}^2 \cdot \text{frequenza di commutazione}$$

dove il valore della frequenza di commutazione è strettamente legato alla frequenza di clock del processore.

Incrementare la frequenza senza eccedere in consumi è stato permesso grazie un abbassamento della tensione di alimentazione (termine quadratico, quindi il più influente) da 5 V ad un minimo di 1.2 V. Al di sotto di questo valore avvengono fenomeni di tipo eletrostatico che portano il sistema in una condizione anomala, dunque inutilizzabile.

C'è inoltre un limite alla capacità di estrarre la potenza prodotta dai processori (e il conseguente calore) tramite ventole o radiatori. Il processo di refrigerazione diventa molto costoso e difficilmente attuabile in dispositivi desktop e laptop.

La soluzione più adottata consiste nelle *architetture multicore*, attraverso il concetto di *parallelismo*. Ciò comporta una maggiore difficoltà e responsabilità durante la scrittura di codice da parte del programmatore (debugging, bilanciamento del carico di lavoro fra le CPU), ma porta a notevoli miglioramenti in termini di efficienza.

Capitolo 2

Aritmetica dei calcolatori

“ There are 10 people in the world, those who can read binary and those who cannot. ”

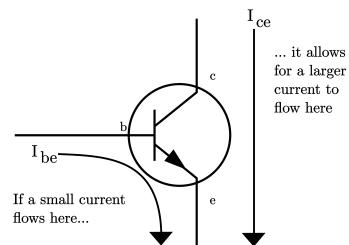
Anonymous

2.1 Informazione nei computer

2.1.1 I transistor

Tutti i computer moderni sono composti da *transistor* (detti anche triodi), delle strutture bistabili: possono assumere due stati come un interruttore, 0 (spento) e 1 (acceso). Molti *transistor* collegati tra loro in una sorta di matrice possono rappresentare delle serie di porte logiche, ed è questa la struttura che sta alla base dell'architettura dei moderni calcolatori.

Un computer memorizza (e manipola) solo sequenze di 0 e 1 (sequenze di bit); anche ENIAC funzionava allo stesso modo, solo che al posto dei *transistor* era composto da *valvole termoioniche*, che non erano differenti nel funzionamento (assumono sempre i due stati descritti in precedenza, solo più lentamente).



2.1.2 Interpretazione delle informazioni

In seguito le sequenze di bit che i computer elaborano/memorizzano possono essere interpretate in tantissimi modi diversi, tra cui numeri, caratteri, suoni, immagini, istruzioni e molti altri. Alla base di tutte le interpretazioni che si possono dare alle stringhe di simboli (che nel caso del computer sono proprio i due stati del bit) sta il concetto di codifica. La *codifica* è appunto uno schema, una legge, un *mapping* che permette prima di tradurre, poi di interpretare stringhe di simboli.

Nell'ambito dell'IT le codifiche sono fortemente caratterizzate dalla lunghezza (dal numero di bit) delle “parole” elementari della codifica. Ad esempio, per rappresentare tutti i caratteri (tabella ASCII estesa) utilizziamo 8 bit (256 caratteri). Si osservi che essendo i *transistor* sistemi bistabili la base 2 (con cifre 0 e 1) è perfetta per rappresentare la codifica dei dati memorizzati/elaborati da essi.

2.2 I numeri

Limitiamoci al caso dei numeri per spiegare il concetto di *codifica*. Ricordiamo che i metodi di rappresentazione numerica che studiamo sono posizionali, ovvero il peso di ogni cifra varia in base alla posizione che essa occupa. In generale se una base è composta da B elementi essa ha B cifre (da 0 a $B - 1$) utilizzate per scrivere ogni numero. Questa formula ci restituisce il valore di ogni numero scritto in una qualsiasi base B :

$$c_i c_{i-1} c_{i-2} \dots c_0 = c_i \cdot B^i + \dots + c_0 \cdot B^0$$

dove c_i è la cifra in posizione i .

2.2.1 Regole di conversione tra basi

Vediamo ora come operare la conversione tra le principali basi:

- *da base 2 a base 16*: partendo da destra, si dividano le cifre binarie in gruppi da 4 cifre ciascuno, e ognuno di questi corrisponderà a una cifra esadecimale; qualora il numero di cifre binarie non sia divisibile per 4, si completi con degli 0 in modo da ottenere gruppi da 4 (*esempio*: $(11011)_2$ diventerà $(0001\ 1011)_2$, ovvero $(1B)_{16}$);
- *da base 2 a base 8*: analogamente, si dividano le cifre binarie in gruppi da 3 cifre ciascuno, e ognuno di questi corrisponderà a una cifra ottale;
- *da una qualsiasi base B a base 10*: come visto sopra, si moltiplichli ogni cifra c_i per B^i , dipendentemente dalla sua posizione;

- da base 10 a una qualsiasi base B :

1. si consideri un numero x_{10} e una base B ;
2. si divida x per B ;
3. il resto della divisione è la cifra da inserire a sinistra nel numero convertito;
4. si assegni a x il quoziante della divisione;
5. si torni al punto 2 e si ripeta finché $x \neq 0$.

Alcuni esempi di conversione

$$1000_{10} = ?_2$$

X	$X/2$	$X \% 2$
1000	500	0
500	250	0
250	125	0
125	62	1
62	31	0
31	15	1
15	7	1
7	3	1
3	1	1
1	0	1

$$1000_{10} = 1111101000_2$$

$$1000_{10} = ?_{16}$$

X	$X/16$	$X \% 16$
1000	62	8
62	3	14(E)
3	0	3

$$1000_{10} = 3E8_{16}$$

2.2.2 I naturali

Nella codifica binaria un numero naturale è rappresentato su k cifre binarie, dove con k cifre si possono rappresentare i numeri tra 0 e $2^k - 1$. La codifica più comune per gli interi spesso usa i byte, sequenze di 8 bit, che tuttavia richiedono molte cifre per rappresentare un numero e quindi spesso, per semplificare la lettura, si usa scrivere i numeri in esadecimale (quindi scrivendo un quarto delle cifre rispetto alla binaria).

Somma e sottrazione Le operazioni somma e sottrazione funzionano allo stesso modo del sistema utilizzato nella numerazione decimale con il riporto. Si guardino gli esempi qui sotto riportati, già di per sé esplicativi del processo:

Moltiplicazione La moltiplicazione in base binaria si può semplificare con il metodo dello shifting, che è più facile illustrare con un esempio:

$$\begin{array}{r}
 \begin{array}{ccccccc|c}
 1 & 1 & 1 & 0 & 1 & 0 & 1 & + \\
 1 & 1 & 0 & 0 & 0 & 0 & 1 & = \\
 \hline
 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0
 \end{array}
 \end{array}$$

Tabella 2.1: Esempio di somma

$$\begin{array}{r}
 \begin{array}{ccccccc|c}
 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & - \\
 1 & 1 & 1 & 0 & 1 & 0 & 1 & = \\
 \hline
 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1
 \end{array}
 \end{array}$$

Tabella 2.2: Esempio di differenza

$$\begin{array}{r}
 \begin{array}{ccccc||c}
 & 1 & 1 & 0 & 1 & \times \\
 \begin{array}{c} 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{array} & & & & & = \\
 \hline
 & 1 & 1 & 0 & 1 & + \\
 & & & - & & + \\
 & & & - & & + \\
 & & & - & & + \\
 \hline
 & 1 & 1 & 0 & 1 & \\
 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & \parallel
 \end{array}
 \end{array}$$

Tabella 2.3: Esempio di prodotto

Si noti che quando è presente un 1 si riscrive il numero (nella posizione corrispondente) mentre quando è presente uno 0 semplicemente non si scrive nulla (e si va solo avanti con le posizioni).

2.2.3 Gli interi

Finora abbiamo visto come si codificano solo i numeri naturali, ma esistono metodi di codifica che permettono di rappresentare anche i numeri negativi (quindi l'insieme degli interi). Le tecniche più comuni sono: modulo e segno, complemento a 1 (CA1) e complemento a 2 (CA2).

Codifica con Modulo e Segno Si usano $k - 1$ bit per rappresentare il valore assoluto (modulo) del numero ed un bit per codificare il segno (0 per i positivi, 1 per i negativi). In questo modo con k bit si possono codificare valori tra $-2^{k-1} + 1$ e $+2^{k-1} - 1$.

NB: esistono due codifiche per lo 0, -0 e $+0$, e questo è un spreco.

Codifica in Complemento a 1 Il primo bit indica sempre il segno e per ottenere l'opposto di un numero positivo si invertono tutti gli 0 in 1 e gli 1 in 0 (e lo stesso si fa per ottenere l'opposto di un numero negativo). Si hanno ancora due rappresentazioni per $+0$ e -0 (rispettivamente, utilizzando 4 bit, 0000 e 1111). È più facile da sommare rispetto al modulo e segno ma solo se il bit significativo non dà riporto.

Quando si sommano due numeri in CA1:

- si deve verificare che i riporti delle prime due cifre significative siano uguali, altrimenti il numero che si ottiene non è rappresentabile in CA1 su k bit.
- alla fine si deve sommare il riporto della prima cifra al risultato ottenuto (ultima riga).

Esempio di somma in CA1 $6 + -3$ (codifica su 5 bit)

$$6 = 00110;$$

$$-3 = \overline{00011} = 11100;$$

$$\begin{array}{r} (\text{Riporti}) & 1 & 1 & 1 \\ & 0 & 0 & 1 & 1 & 0 \\ & 1 & 1 & 1 & 0 & 0 \\ \hline (\text{Risultato}) & 0 & 0 & 0 & 1 & 0 \end{array}$$

$$00010 + 1 = 00011 (= 3);$$

Codifica in complemento a 2 (CA2) Per tradurre un numero da intero con segno a complemento a 2 (CA2) basta invertire tutti i bit e poi sommare 1. Viceversa, per convertire da CA2 a binario, si sottrae 1 e poi si invertono tutti i bit. Ancora una volta il bit più significativo indica il segno, ma in questo caso la codifica dello 0 è unica, quindi con k bit si possono rappresentare i numeri da -2^{k-1} a $2^{k-1} - 1$.

Somma algebrica in complemento a 2 (CA2)

L'utilizzo della codifica CA2 permette di eseguire somme algebriche senza alcuna differenza operativa, ma rende necessario prestare attenzione all'*overflow*. Con il termine *overflow* si intende la situazione in cui, date in input delle informazioni codificate in un numero k di bit arbitrariamente scelto, lo stesso numero k di bit non è sufficiente a codificare le informazioni in output.

Diamo subito un esempio: supponiamo di voler eseguire $-64 - 8$ utilizzando il CA2 e $k = 7$ bit. Ricordiamo che:

$$-64 = 1000000_2 = \overline{1000000} + 1 = 0111111 + 1 = 1000000 \text{ in CA2}$$

$$-8 = 0001000_2 = \overline{0001000} + 1 = 1110111 + 1 = 1111000 \text{ in CA2}$$

A questo punto eseguiamo la somma algebrica:

$$(1) \quad \begin{array}{r} 1 & 0 & 0 & 0 & 0 & 0 & 0 & + \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & = \\ \hline 0 & 1 & 1 & 1 & 0 & 0 & 0 & \end{array}$$

Si noti che il numero fra parentesi indica l'ultimo riporto, che chiaramente non possiamo inserire poiché sforerebbe i 7 bit.

Osserviamo così che, per $k = 7$ bit, $-64 - 8 = 0111000_2 = 56_{10}$, che ovviamente non ci piace; concludiamo quindi che la somma algebrica $-64 - 8 = -72$ non è rappresentabile con soli $k = 7$ bit.

In generale possiamo dire che, dati due numeri a e b aventi lo stesso segno, avremo problemi di *overflow* se:

- $a + b > 2^{k-1} - 1$;
- $a + b < -2^{k-1}$.

ricordando che $-2^{k-1} \leq a, b \leq 2^{k-1} - 1$.

2.2.4 Aritmetica modulare

Inconsapevolmente utilizziamo l'*aritmetica modulare* quasi continuamente, nella nostra quotidianità; l'esempio più comune è quando sommiamo le ore, che sono organizzate in modulo 24 (da cui la denominazione ufficiosa di "Matematica dell'orologio"), ma anche minuti e secondi (modulo 60) e gli angoli (modulo 360).

Diamone una descrizione intuitiva: fissato un numero n finito di valori, arrivato al numero più grande ($n-1$), al successivo avrà che $(n-1)+1 = 0$. Per completezza, forniamo ora la relazione di equivalenza di due numeri in modulo n :

$$a \equiv b \iff a \% n = b \% n$$

Quando eseguiamo delle operazioni sui numeri in base 2 su k bit, stiamo lavorando in un ambiente in modulo 2^k , e calcolando $a + b$ stiamo in realtà trovando $(a + b)\%2^k$, e i due valori saranno uguali se e solo se $a + b < 2^k$, perché altrimenti ogni valore successivo ricomincerebbe da 0: questo è l'*overflow*. In particolare, quando lavoriamo su numeri interi in CA2, dobbiamo ricordare che, se $-2^{k-1} \leq a + b \leq 2^{k-1} - 1$ non avremo *overflow*, altrimenti:

- se $a + b > 2^{k-1} - 1$, otterremo $a + b + 2^k \rightarrow overflow$;
- se $a + b < -2^{k-1}$, otterremo $a + b - 2^k \rightarrow overflow$.

2.2.5 I reali

Non è sempre semplice codificare un numero reale $\alpha \in \mathbb{R}$, poiché di questo insieme fanno parte i numeri cosiddetti *irrazionali*, che presentano infinite cifre non periodiche dopo la virgola (si pensi ad esempio a $\pi = 3,14159\dots$, o $e = 2,71828\dots$, o ancora $\sqrt{2} = 1,41421\dots$). Alcuni numeri non possono quindi essere rappresentati con un numero finito di bit, per cui ne rappresenteremo solo un'approssimazione, e per farlo abbiamo sviluppato due metodi: la rappresentazione a *virgola fissa* (*fixed point*) e quella a *virgola mobile* (*floating point*).

Codifica in virgola fissa

Questo sistema di codifica prevede che, posto un numero k di bit per la rappresentazione del numero, si impieghino $k - f$ bit per rappresentarne la parte intera, e i restanti f bit per rappresentarne la parte razionale:

$$\underbrace{c_{k-1}c_{k-2}\cdots c_f}_{k-f} \cdot \underbrace{c_{f-1}\cdots c_0}_f$$

Possiamo scrivere una formula generale più compatta. Supponiamo di voler convertire un numero x_{10} :

$$x_{10} = \sum_{i=0}^{k-1} c_i B^{i-f} = \sum_{i=0}^{k-1} c_i B^i \cdot B^{-f} = (\sum_{i=0}^{k-1} c_i B^i) \cdot B^{-f}$$

Detta in altri termini, convertiamo il numero da base 10 come se non avesse la virgola e poi moltiplichiamo per B^{-f} . Presentiamo vantaggi e svantaggi di questo tipo di codifica:

- le operazioni si possono svolgere senza alcuna differenza, se non l'accortezza di scalare il risultato;
- il peso per la CPU è ridotto;
- non ci sono errori di approssimazione, *se il valore è rappresentabile*;
- iniziamo ad avere problemi quando lavoriamo con un ampio spettro di ordini di grandezza.

Codifica in virgola mobile

Un qualsiasi numero reale può essere scritto anche nella forma $x = M \cdot B^{exp}$ dove M si dice *mantissa*, *exp esponente* (effettivo dei calcoli) e B è la base della rappresentazione (che nel nostro caso è la leggendaria base 2).

Questo metodo di rappresentazione, molto simile alla notazione scientifica standard, è implementato anche nell'informatica in più modi diversi ma alla base di tutti si riconosce lo stesso schema: x può essere rappresentato su k bit utilizzando m bit per il campo mantissa M ed $e = k - m$ bit per il campo esponente E (esponente memorizzato, attenzione: è diverso dall'esponente effettivo dei calcoli); quest'ultimo permette in un certo senso di "spostare" la virgola; con esponenti "piccoli" si avranno x "piccoli" e viceversa con esponenti "grandi" si avranno x "grandi".

Lo standard più comunemente utilizzato è l'IEEE 754 che, nei numeri *float* a precisione singola (dove si usano 32 bit), ha questa struttura:

1 bit	8 bit	23 bit
Bit di segno	E (esponente memorizzato)	Mantissa

Il bit di segno ha la funzione che già conosciamo, mentre il campo E può contenere valori compresi tra 0 e 255 (se non si tratta di un *float* a precisione singola i valori sono compresi tra 0 e $2^{e-1} - 1$); questi ultimi due valori vengono utilizzati per funzioni speciali, mentre i valori da 1 a 254 compresi codificano l'*esponente effettivo dei calcoli*, che diremo exp^1 , in questo modo:

$$exp = E - (2^{e-1} - 1)$$

Il numero $2^{e-1} - 1$ è un *bias* e nel nostro caso vale 127, per cui $exp = E - 127$. L'ultimo campo, la *mantissa*, codifica un numero intero.

Come calcolare il valore di un float Il primo bit è utilizzato per indicare il segno, i successivi 8 per E e i restanti 23 per la mantissa. La decodifica in decimale assume due forme in base al valore di E :

- se $E = 0$, allora il numero in base 10 è $x_{10} = 0.M \cdot 2^{(exp+1=-126)}$;
- se $E > 0$, allora il numero in base 10 è $x_{10} = 1.M \cdot 2^{exp}$, dove exp è definito come qui sopra descritto.

¹La notazione *exp* è stata introdotta da noi autori della dispensa perché ci è sembrata più chiara ed agile di quella utilizzata nelle slides, chiediamo venia se questo genererà confusione.

Nel primo di questi due casi i numeri si dicono *denormalizzati*, essi rappresentano tutti i numeri compresi tra $1.4 \cdot 10^{-45}$ e $1.1754942 \cdot 10^{-38}$; nel secondo caso i numeri si dicono *normalizzati* [questa distinzione è stata aggiunta per migliorare la precisione della notazione con numeri molto vicini allo 0].

Per tradurre in decimale un numero in virgola mobile si deve quindi per prima cosa determinare il segno, poi si calcola l'esponente effettivo dei calcoli ($exp = E - bias$) e da qui, simmetricamente a prima, si procede in due modi:

- se $exp \neq -bias$ si parla di numero *normalizzato* e quindi si aggiunge un bit 1 alla mantissa in modo da ottenere il numero $1.M$ e si sposta la virgola di exp posizioni e per completare il tutto si traduce il numero binario (in virgola fissa) così ottenuto in decimale;
- se $exp = -bias$ si parla di numero *denormalizzato* e quindi si aggiunge un bit 0 alla mantissa in modo da ottenere $0.M$ e come prima si sposta la virgola di $(exp+1)$ posizioni e poi si decifra il numero (binario in virgola fissa) così ottenuto;

Ripetiamo ancora una volta il procedimento per chiarezza: stabilito quale delle precedenti due opzioni riguarda il nostro caso spostiamo la virgola di exp posizioni nel numero $1.M$ oppure $0.M$ (verso destra se $exp > 0$ e viceversa), ed in conclusione decodifichiamo il numero ottenuto come se fosse un binario in virgola fissa.

Tabella riassuntiva

Ecco una breve tabella riassuntiva dell'interpretazione delle stringhe nella rappresentazione IEEE 754:

Categoria	Esponente (E)	Mantissa
± zero	0	0
num. denormalizzati	0	non zero
num. normalizzati	1-254	qualunque
± infinito	255	0
nan (not a number)	255	non zero

Tabella 2.4: Riepilogo floating point

Esempio di decodifica di un numero in virgola mobile

Si traduca in IEEE 754 il numero $(0100001010011011100000000000000)_2$. Spacchettiamolo nei vari campi:

MSB	Esponente	Mantissa
0	10000101	001101110000000000000000

Procediamo ora alla traduzione.

1. Osserviamo che il numero è positivo, poiché il bit di segno è 0.
2. L'esponente effettivo dei calcoli exp è

$$E - 127 = E - \text{bias} = 10000101 - 01111111 = 133 - 127 = 6$$

L'esponente è diverso da zero, perciò si tratta di un numero normalizzato.

3. La mantissa va moltiplicata per $2^{exp} = 2^6$ e, poiché il numero è normalizzato, devo considerare $1.M$:

$$1.M \cdot 2^6 = 1.0011011100000000000000 \cdot 2^6 = 1001101.11$$

4. A questo punto convertiamo il risultato ottenuto come fosse in virgola fissa:

$$2^6 + 2^3 + 2^2 + 2^0 + 2^{-1} + 2^{-2} = 77.75$$

Capitolo 3

Codifica del testo

3.1 La codifica ASCII

Oltre alle codifiche per numeri naturali, interi e reali viste nel capitolo precedente, sono state introdotte delle codifiche per rappresentare sequenze di caratteri. Una delle più note è la codifica American Standard Code for Information Interchange (ASCII), che utilizza 7 bit per rappresentare ciascun carattere (quindi $2^7 = 128$ caratteri disponibili in totale). Attraverso la codifica ASCII è possibile rappresentare tutte le lettere dell'alfabeto anglosassone (lettere maiuscole, minuscole, numeri, punteggiatura, ecc.).

Questa codifica presenta uno svantaggio: seppur utilizzi solamente 7 bit, un byte è formato da 8 bit. Ciò implica che il primo bit di ciascun carattere viene impostato a 0 per default.

3.2 La codifica ASCII estesa

A differenza dell'ASCII tradizionale, l'*extended ASCII* sfrutta anche l'ottavo bit per codificare caratteri addizionali (quindi $2^8 = 256$ caratteri disponibili in totale).

Non esiste un unico standard esteso: sono state implementate diverse versioni, ciascuna in grado di supportare diversi alfabeti specifici. In particolare, i byte con valore minore di 128 sono comuni a tutte le implementazioni della codifica ASCII estesa al fine di garantire la compatibilità.

3.3 La codifica UNICODE

La necessità di avere una codifica univoca ha portato ad aumentare fino a 32 il numero di bit per la rappresentazione di un carattere attraverso la codifica

UNICODE. Anche qui non mancano differenti formati di codifica:

- *UTF-32*: ogni simbolo è composto da 32 bit.
- *UTF-16*: ogni simbolo è composto da 16 o più bit (lunghezza variabile).
- *UTF-8*: ogni simbolo è composto da 8 o più bit; mantiene la compatibilità con ASCII.

Input e output da file Come si è visto precedentemente, è dunque possibile codificare dati numerici in diverse modalità. Quando si scrive un programma che si interfaccia con file, si consiglia caldamente di salvare i dati numerici in formato binario e non in forma di stringa. Questa accortezza permette di ridurre la dimensione in byte del file finale prodotto.

Capitolo 4

Le reti logiche

4.1 Introduzione

Come ben sappiamo, a differenza di ENIAC, i computer moderni si compongono di un'enorme quantità di circuiti elettronici. Trattandosi di elettronica digitale, a livello hardware è possibile lavorare con due livelli fondamentali:

- *alto* (asserito); viene solitamente associato alla tensione di alimentazione Vdd . Si indica con 1.
- *basso* (negato); viene solitamente associato alla massa (tensione vicina a 0). Si indica con 0.

Nel passaggio da un livello all'altro si presenta lo stato *transitorio*, in cui la tensione assume valori intermedi fra zero e Vdd .

In particolare, per trasformare alcuni valori logici in ingresso in altri valori logici in uscita, vengono utilizzati i *circuiti logici*, costituiti dalle *porte logiche*. I circuiti logici si differenziano principalmente in:

- *reti combinatorie*: l'uscita dipende unicamente dal valore di ingresso (non sono dotati di memoria).
- *reti sequenziali*: l'uscita dipende dal valore di ingresso e dalla storia degli ingressi precedenti (sono dotati di memoria, detta "stato" della rete).

È possibile riassumere il comportamento di una rete logica attraverso una *tabella di verità*, che descrive i diversi output al variare dei diversi input.

4.2 L'algebra di Boole

Esistono principalmente tre operazioni di base:

- *AND*: produce 1 se entrambi gli operandi sono 1, altrimenti 0. Solitamente si indica con $A \cdot B$.
- *OR*: produce 0 se entrambi gli operandi sono 0, altrimenti 1. Solitamente si indica con $A + B$.
- *NOT*: inverte il valore logico. Solitamente si indica con \bar{A} .

4.2.1 Regole di semplificazione

Di seguito vengono riportate alcune regole di semplificazione:

- identità: $A + 0 = A$, $A \cdot 1 = A$
- "zero e uno": $A + 1 = 1$, $A \cdot 0 = 0$
- inversa: $A + \bar{A} = 1$, $A \cdot \bar{A} = 0$
- commutativa: $A + B = B + A$, $A \cdot B = B \cdot A$
- associativa: $A + (B + C) = (A + B) + C$, $A \cdot (B \cdot C) = (A \cdot B) \cdot C$
- distributiva: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$, $A + (B \cdot C) = (A + B) \cdot (A + C)$
- De Morgan: $\overline{A \cdot B} = \bar{A} + \bar{B}$, $\overline{A + B} = \bar{A} \cdot \bar{B}$

Proprio grazie alla regola di De Morgan, si può affermare che la porta *NAND* (not AND) e la porta *NOR* (not OR) sono universali. Ciò significa che utilizzando unicamente porte NAND (rispettivamente NOR) è possibile costruire tutte le altre porte logiche.

4.2.2 La tavola di verità e i mintermini

Input			Output		
A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Tabella 4.1: Esempio di tabella di verità

Si può verificare che:

$$\begin{aligned} D &= A + B + C \\ F &= A \cdot B \cdot C \\ E &= (A \cdot B \cdot \bar{C}) + (A \cdot \bar{B} \cdot C) + (\bar{A} \cdot B \cdot C) \end{aligned}$$

In particolare, l'ultima espressione è stata ottenuta come somme di prodotti attraverso i cosiddetti *mintermini*. Per ciascun 1 presente nella colonna E, si scrive il prodotto degli input (asseriti se pari a 1, negati se pari a 0); il risultato dell'espressione è pari alla somma di tali prodotti.

Spesso ritornerà il concetto di *somme di prodotti* o più brevemente *sp*, in particolare nella sezione 4.4 trattando dei *pla*.

4.3 Le porte logiche

A seguire vengono rappresentate, attraverso i corrispettivi simboli, le porte logiche implementate attraverso gli operatori booleani fondamentali:

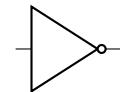
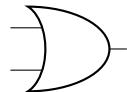
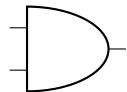


Figura 4.1: Porta AND Figura 4.2: Porta OR Figura 4.3: Porta NOT

Si ricorda inoltre che è possibile combinare fra loro diverse porte logiche:

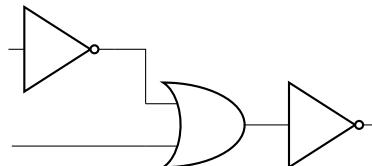


Figura 4.4: Combinazione di porte logiche

4.4 Alcuni circuiti degni di nota

Decoder

L'ingresso del *decoder* svolge il ruolo di selettore: ricevendo in input il valore n (codificato in binario), si accenderà l' n -esima uscita. A seguire l'implementazione di un decoder a 3 bit:

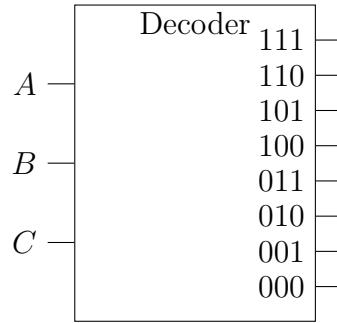


Figura 4.5: Decoder a 3 bit

Input			Output							
In2	In1	In0	Out0	Out1	Out2	Out3	Out4	Out5	Out6	Out7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Tabella 4.2: Tabella di verità per un decoder a 3 bit

Multiplexer

Prevede un input con valore S utilizzato come selettore. L'output del *multiplexer* è il valore dell'input S -esimo.

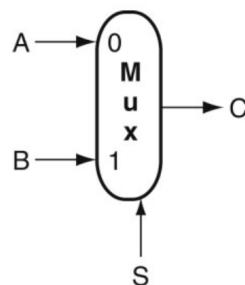


Figura 4.6: Multiplexer a 1 bit

È possibile costruire un multiplexer attraverso un decoder:

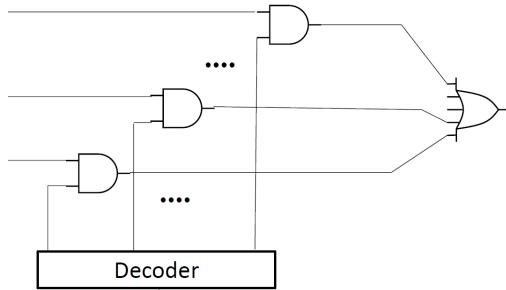


Figura 4.7: Multiplexer a n vie

Molto spesso, per operare con dati complessi, si compongono degli array di elementi elementari. Segue un esempio di come sia possibile costruire un multiplexer con *bus* a 32 bit utilizzando un array di multiplexer a 1 bit.

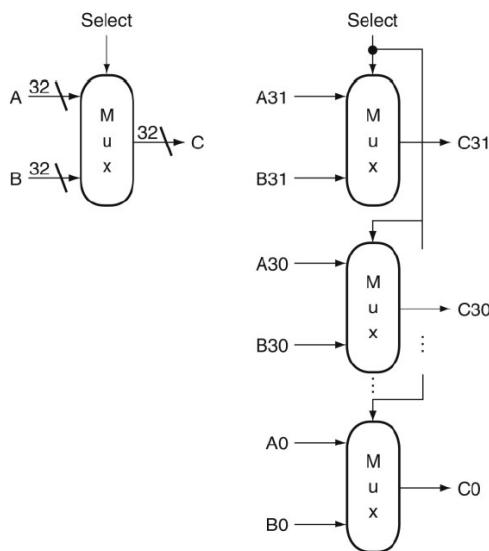


Figura 4.8: Multiplexer a 32 bit

Programmable Logic Array (PLA)

Il Programmable Logic Array (PLA) si compone di due strutture: una barriera di AND e una barriera di OR. La dimensione totale del PLA è data dalla somma del piano AND (numero di mintermini) e del piano OR (numero di uscite).

Inoltre, il PLA implementa porte logiche solamente per le configurazioni che producono 1 in uscita. In aggiunta, se un mintermine è condiviso tra varie uscite, lo si

può riutilizzare. Successivamente seguono due diverse implementazioni della rete logica descritta nel paragrafo 4.2.2.

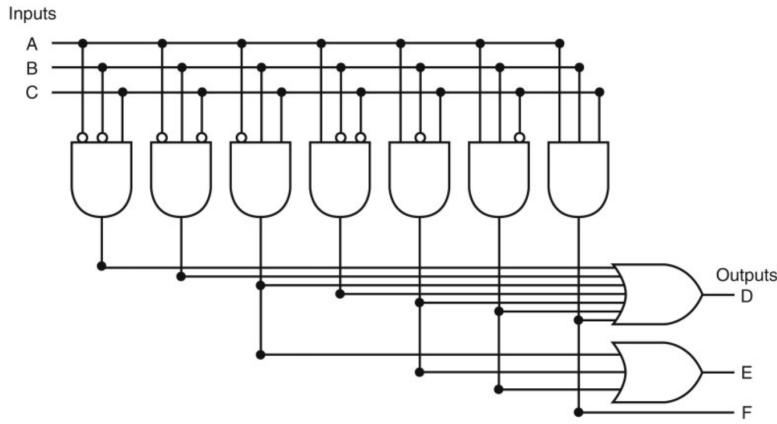


Figura 4.9: Esempio di rete logica

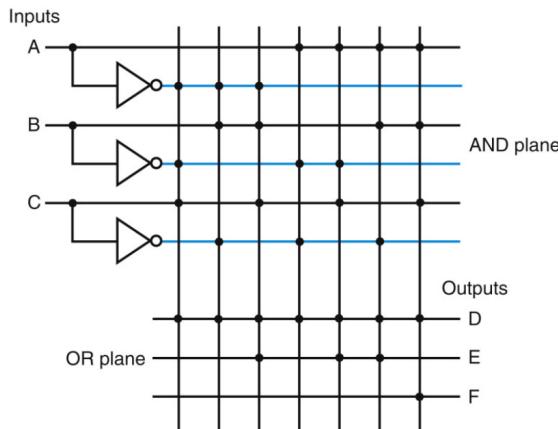


Figura 4.10: Esempio di rete logica implementata attraverso PLA

4.5 Il concetto di costo

Come appena visto, le reti logiche possono essere implementate in maniera differente. Si definisce *costo* di una rete logica la somma del numero di porte e del numero di ingressi della rete. Quando si analizza una rete, è importante saperne trovare l'implementazione con costo minimo. Principalmente si può procedere in due modalità:

- regole di semplificazione matematiche dovute all'algebra di Boole (vedi paragrafo 4.2.1).

- metodi basati su rappresentazioni grafiche (mappe di *Karnaugh*), nei casi più semplici.

4.6 Le reti sequenziali

Come si è visto, le reti combinatorie non hanno memoria degli stati del passato: in ogni istante di tempo l'uscita dipende solamente dagli ingressi nell'istante considerato. Tuttavia, in certe applicazioni è necessario introdurre una vera e propria memoria nel sistema.

La memoria in una rete logica si ottiene attraverso una *retroazione*, che consiste nel ridirezionare alcune porte di uscita in ingresso ad altre porte del medesimo circuito, in maniera tale da formare un anello.

Seppur questo meccanismo offra delle potenzialità enormi, l'analisi e la sintesi di una rete logica è molto più complessa rispetto a quella delle reti combinatorie.

Latch S-R

A seguire un'implementazione di una rete sequenziale *latch S-R* (*set-reset*) costruita attraverso due porte *NOR*:

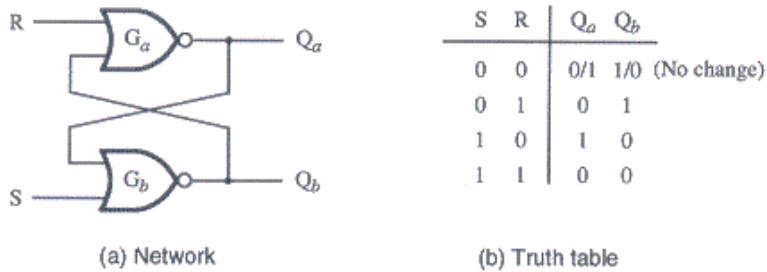


Figura 4.11: Esempio di latch a porte NOR

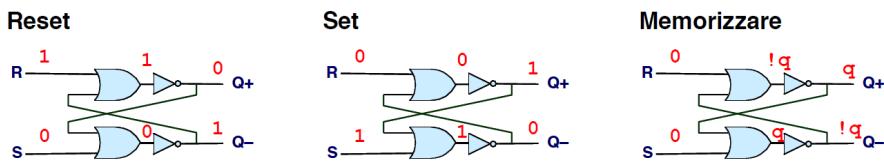


Figura 4.12: Operazioni di un latch a porte NOR

Ma, considerando che i segnali non si propagano in tempo nullo, ciò potrebbe causare dei problemi circa la correttezza del segnale stesso. La soluzione è quella di aggiungere un ulteriore segnale, in grado di temporizzare il circuito: il *clock*. Un latch dotato di temporizzazione viene chiamato *gated latch*.

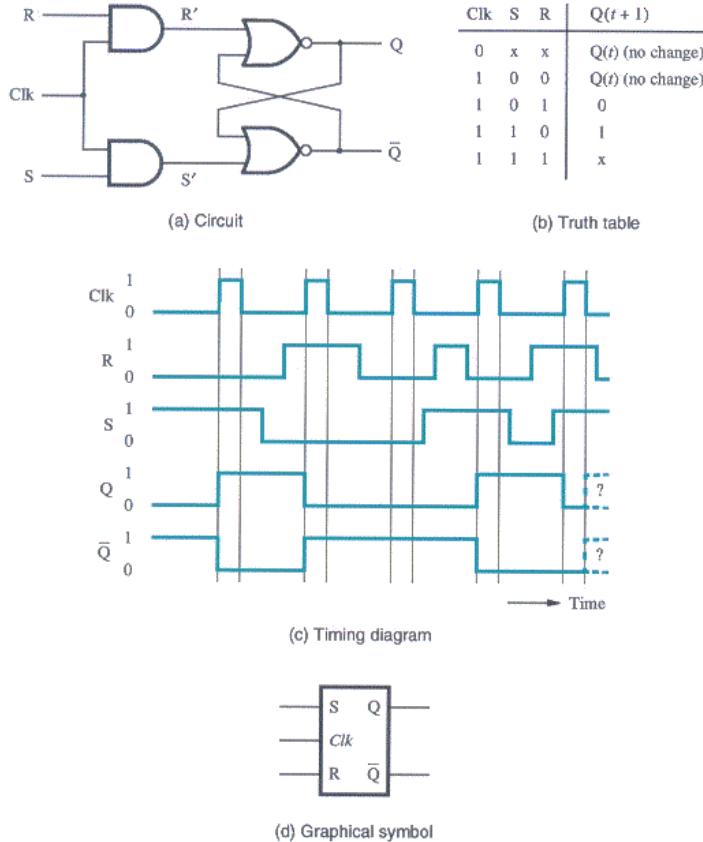


Figura 4.13: Esempio di gated latch

Il difetto principale di un latch R-S è quello di avere uno stato indecidibile: l'uscita non può essere nota con certezza quando entrambi gli ingressi sono a 1.

Una soluzione è implementata attraverso *latch* tipo D: gli ingressi al circuito sono ottenuti da un'unica variabile (di cui si fa il negato). Altrimenti, attraverso circuiti *flip-flop master-slave* è possibile ottenere che l'uscita del circuito commuti esattamente al termine dell'impulso di clock. In particolare, i latch di tipo D vengono utilizzati per implementare i registri della CPU.

Capitolo 5

Introduzione al linguaggio Assembly

“ It’s hardware that makes a machine fast. It’s software that makes a fast machine slow. ”

Craig Bruce

5.1 Perché esiste Assembly

Le CPU capiscono esclusivamente le istruzioni scritte nel loro specifico linguaggio macchina, che corrisponde ad una sequenza di bit 0 e 1. Al giorno d’oggi si programma praticamente solo ad alto livello, poiché programmare in linguaggio macchina sarebbe molto più complesso e dispendioso; il mezzo che ci permette di intermedicare tra l’alto livello e la macchina è l’Assembly. Il vantaggio di Assembly è l’utilizzo di codici mnemonici (le keyword del linguaggio) che al programmatore risultano più semplici di stringhe di 0 e 1.

Si deve però notare che Assembly, essendo di così basso livello, è un linguaggio molto specifico, che si adatta solo ad una certa struttura (anche se esistono architetture che possono essere gestite da più versioni di Assembly). Ad esempio, un linguaggio Assembly per ARM non può essere usato su architetture Intel.

5.2 Come funziona Assembly

5.2.1 L’Assembler

Una volta che il programmatore ha scritto il suo algoritmo e questo è stato codificato in Assembly dal calcolatore, chi si occupa della traduzione dell’Assem-

bly? L'Assembler, che è appunto il compilatore dell'Assembly. Ad esempio, avendo un'istruzione del tipo `add a, b, c`, è l'Assembler che la trasforma in codice binario).

L'insieme di tutte le istruzioni specifiche conosciute da un'architettura è detto Instruction Set Architecture (ISA).

5.2.2 Instruction Set Architecture

L'ISA è l'insieme di regole e procedure che specifica come ogni istruzione deve essere strutturata, codificata ed anche eseguita; in pratica esso specifica sia la semantica che la sintassi, che cosa si deve fare per compiere delle istruzioni e come si fa. L'ISA specifica inoltre anche i registri e la loro tipologia (*general purpose* o *specializzati*), come accedere ad essi (sono molto vicini alla CPU contengono i dati che devono essere utilizzati in maniera super-rapida) ed alla memoria (indirizzamento).

Per comprendere bene le istruzioni che compongono l'ISA conviene ripassare le fasi di funzionamento di una CPU.

5.2.3 Funzionamento di una CPU

Il funzionamento di una CPU segue il seguente schema:

- *fetch*: in questa fase viene letta l'istruzione da eseguire.
 1. l'indirizzo dell'istruzione da eseguire, contenuto nel Program Counter (PC), viene copiato nel Memory Address Register (MAR) (indirizzo di accesso al bus di sistema);
 2. i dati dell'istruzione vengono quindi trasferiti dalla RAM al Memory Data Register (MDR) (che tiene conto dei dati da leggere o scrivere);
 3. i dati vengono salvati in un registro invisibile al programmatore;
 4. viene incrementato il PC, facendolo puntare all'istruzione successiva;
- *decode*: l'istruzione qui dev'essere decodificata (spacchettata in una serie di campi) che contengono le varie caratteristiche dell'istruzione.
- *execute*: alla fine l'istruzione viene eseguita, essa può essere aritmetica, logica, di gestione della memoria, ecc. Nel caso di istruzioni di salto è possibile modificare il valore del PC.

Ricapitolando, il ciclo dell'esecuzione delle istruzioni segue lo schema lineare *fetch* > *decode* > *execute*, scandito dal clock del processore.

Ritornando ad Assembly

Nell'Assembly di fatto non esistono istruzioni come i cicli, però queste vengono implementate attraverso istruzioni che modificano il Program Counter (PC). Complessivamente un programma in Assembly può essere descritto come una lista di istruzioni di tre tipi:

- operazioni aritmetiche/logiche (o anche algebriche e booleane).
- operazioni di gestione dati (istruzioni di indirizzamento).
- operazioni di controllo del flusso del programma (operazioni sul PC).

Generalmente possiamo utilizzare le istruzioni in due modi diversi: possiamo inserire direttamente i dati nel corpo dell'istruzione, ma questa è una grande limitazione poiché le istruzioni hanno dimensioni preimpostate (o comunque limitate), oppure possiamo inserire nelle istruzioni gli indirizzi di alcuni registri (dove posso stoccare i dati che necessito).

Esiste un'ultima possibilità, che ci permette di memorizzare nelle istruzioni gli indirizzi dei dati che stocchiamo in memoria (così da poter sfruttare le quantità enormi di memoria di un calcolatore) e poi prelevare i dati necessari dalla memoria e caricarli nei registri (così da avere la velocità di accesso dei registri).

Si deve tenere in conto che però i registri sono limitati (in alcune architetture ARM sono addirittura solo 4) e devono anche essere ripuliti dopo ogni utilizzo (a questo pensa la CPU).

Registri Esistono diversi tipi di registro, sia generici che specifici; un esempio di registro specifico è il registro *\$zero*, che contiene tutti zeri: viene utilizzato per semplificare e diminuire le istruzioni dell'ISA. Per copiare un registro in un altro, invece che implementare una funzione specifica, si somma il registro con il *registro zero* e si salva il risultato in un terzo registro (che sarà la copia del primo).

5.2.4 Diversi tipi di ISA

Nel tempo si sono sviluppati due diversi approcci verso gli ISA, il *Complex Instruction Set Computer* (CISC) ed il Reduced Instruction Set Computer (RISC). A seguire alcune loro caratteristiche:

- CISC: comprende molte istruzioni, anche complesse; ha poca necessità di registri e la scrittura di programmi Assembly risulta più facile; tuttavia, per implementare le sue istruzioni (come già detto complesse), aumenta la difficoltà nella progettazione di hardware adatti.
- RISC: comprende solo istruzioni di base o comunque poche istruzioni e relativamente semplici; in questo modo semplifica l'implementazione della CPU

e della sua architettura, ma ha bisogno di molti più registri per caricare le istruzioni temporanee (le intermedie necessarie per compiere istruzioni più complesse). Inoltre complica la programmazione in Assembly (il suo linguaggio conosce solo le istruzioni basilari, quindi quelle complesse dovranno essere descritte dal programmatore).

5.2.5 I tipi di istruzioni

Istruzioni aritmetiche/logiche Le istruzioni aritmetiche e logiche spettano alla Arithmetic-Logic Unit (ALU), che è il “chip” che svolge effettivamente (che esegue) le istruzioni.

Istruzioni di gestione dati Le istruzioni di gestione (o movimento) dati sono svolte da altri elementi specifici adibiti proprio allo spostamento di dati tra registri, da registri a memoria e viceversa.

Quando la CPU esegue un programma necessita di due elementi: gli operandi e la destinazione (dove salvare il risultato). Le due tipologie di ISA offrono due alternative diverse: l'ISA RISC, per semplicità e velocità, salva tutto nei registri, mentre l'ISA CISC riesce ad accedere/salvare questi elementi anche nella memoria (rendendo anche la programmazione in Assembly più semplice).

Istruzioni condizionali/di controllo del flusso Per esprimere questo tipo di istruzioni, alcune ISA *saltano* nel codice (modificando il Program Counter (PC)) sfruttano il contenuto di registri generici; ad esempio, eseguono l'istruzione successiva solo se un certo registro è uguale ad un altro (questo però richiede l'implementazione del confronto tra registri, che è una classica operazione complessa da CISC); altri ISA controllano se sono presenti degli elementi detti flag contenuti in un apposito registro (*registro flag*) che segnalano determinate condizioni.

Le istruzioni di controllo di flusso sono svolte da alcuni elementi che gestiscono il PC: quando non ci sono *salti* incrementano il PC di 4 in 4 (o 8 in 8 a seconda dell'architettura), quando si verificano *salti* invece le reti logiche hardware che gestiscono il PC devono cambiarlo ma tenere in memoria il suo indirizzo effettivo, per riprendere da lì al termine della subroutine.

Curiosità: nell'architettura ARM tutte le istruzioni sono condizionali, mentre altre architetture hanno solo poche istruzioni dedicate al salto condizionale.

5.2.6 Confronto tra CISC e RISC

Una volta osservato tutto ciò potrebbe venire spontaneo chiedersi quale delle due ISA sia la migliore, ma la risposta ovviamente dipende dal contesto; le due modalità si sono sviluppate per essere ottimali in ambiti diversi e hanno obiettivi

differenti.

Esistono tuttavia ISA ibride, che derivano da entrambe le architetture, come ad esempio l'ARM, che è presente sulla maggior parte degli smartphone odierni. Questi ibridi sono detti RISC "pragmatici" e combinano parte della flessibilità e complessità CISC con parte della semplicità e regolarità RISC.

5.2.7 Accesso alla memoria

Per svolgere questa funzione in RISC vi sono solo le istruzioni *load* e *store*, mentre per CISC sono presenti istruzioni più generiche; in ogni caso l'argomento dell'istruzione è sempre *<memory location>*.

Ci sono varie modalità di indirizzamento che propongono che in questo campo siano scritti indirizzi diversi:

- *indirizzamento assoluto*: nel campo (operando dell'istruzione) si indica l'indirizzo di memoria (difetto: si possono indicare pochi indirizzi poiché si hanno pochi bit).
- *indirizzamento indiretto*: l'indirizzo di memoria contenuto in un registro (nell'argomento *<memory location>*).
- *base + spiazzamento*: l'indirizzo è ottenuto "shiftando" (o manipolando) il contenuto di un registro: si ha un registro base con un indirizzo base ed un secondo registro che viene shiftato (a partire dal base), e come argomento *<memory location>* la quantità di cui devo shiftare questo secondo registro.
- *combinazioni* varie delle modalità precedenti, che permettono di accedere a locazioni di memoria in maniera molto fine (es.: base + indice e base + indice + spiazzamento).

Se si vuole, questo può essere osservato come un esempio di confronto tra i due tipi di ISA: nell'ISA CISC si può svolgere "base + spiazzamento" direttamente in una stessa istruzione (espressione), mentre nell'ISA RISC si necessita di più istruzioni.

5.2.8 Application Binary Interface

L'ISA definisce numero, nome e funzione dei registri (*general-purpose* o specializzati), ma questo non basta, perché non abbiamo risposte a domande come: quali registri posso usare? Quali posso modificare durante una *subroutine*? Come passo i valori dei parametri e di ritorno?

Per questo c'è il bisogno dell'Application Binary Interface (ABI), che fornisce una

sorta di protocollo dell'utilizzo dei registri (sostanzialmente una vera e propria raccolta di regole sull'utilizzo dei registri).

Un programma è composto da tanti metodi e funzioni, e questi vengono mappati in *subroutine* (un pezzo monolitico di istruzioni) dal linguaggio Assembly. Per esempio, l'istruzione $r = \text{funz(arg1, arg2, arg3, ...)}$; richiama la *subroutine* *funz*. Si deve decidere quali registri utilizzare per immagazzinare gli input della funzione, dove si può salvare il suo output, quali registri essa può modificare liberamente. Le risposte a queste questioni vengono risolte dall'ABI che definisce a livello software le *convenzioni di chiamata*, che specificano come utilizzare i registri all'invocazione di una subroutine.

Nota tecnica: dato un ISA, questo può essere servito da ABI diversi, poiché l'ABI influenza solo la gestione dei servizi.

Capitolo 6

L'architettura MIPS

“Assembly programmers are the only programmers who can truly claim to be the masters, and that's a truth worth meditating on.”

Jeff Duntemann, *Assembly Language Step-by-Step: Programming with DOS and Linux*

Dopo la breve introduzione del precedente capitolo, affronteremo ora lo studio dell'architettura *MIPS*; iniziamo con questa perché, seppur non molto diffusa, è molto simile ad altre architetture che vedremo in seguito (soprattutto ARM), ed è quindi propedeutica.

6.1 Breve riepilogo su ISA

Come sappiamo già, ogni processore possiede un proprio linguaggio macchina, che non è altro che una sorta di "vocabolario di istruzioni" detto *instruction set architecture* (o ISA per gli amici); queste istruzioni sono solitamente le operazioni aritmetiche fondamentali (a parte quelle che lavorano su floating point) e quelle logiche.

Nonostante ogni processore possieda un proprio ISA, le differenze non sono poi così grandi (motivo per cui conoscere MIPS ci agevolerà molto nell'apprendimento di altre ISA); una similitudine adeguata può essere fatta con i dialetti di una lingua maggiore (per esempio, veneziano, trevigiano e veronese sono tutte varianti locali mutualmente intelligibili della Grande Lingua Veneta).

Lo scopo di avere questi sistemi di istruzioni è poter controllare e sfruttare la potenza di calcolo fornita dai nostri calcolatori senza conoscerne i dettagli; inoltre,

come già fatto notare da Von Neumann nel 1947, la progettazione di un'architettura influisce enormemente sull'efficienza dell'hardware, ed è quindi fondamentale che si ricerchino istruzioni *semplici, chiare e veloci*.

Un organico di istruzioni andrà a costruire un *programma memorizzato* che, come approfondiremo, altro non è che un insieme di istruzioni mappato sotto forma di numero binario, non diversamente da numeri o qualsiasi altro dato.

Cos'è MIPS

Come già detto, nonostante non sia diffusissima, inizieremo a studiare l'architettura MIPS in quanto propedeutica. È una cosiddetta architettura RISC. Per quanto semplice e vicino al linguaggio macchina un linguaggio potesse essere, era chiaro fin dai tempi di Von Neumann che non dovevano mancare le operazioni aritmetiche e logiche fondamentali; oltre a quelle, MIPS possiede inoltre alcuni comandi per il controllo di flusso e per l'accesso alla memoria.

6.2 Operazioni aritmetiche

Per dare una chiave di lettura al percorso che faremo su MIPS useremo alcuni principi di progettazione software e hardware che ogni informatico segue fin dagli albori, e naturalmente, introducendo le operazioni aritmetiche, inizieremo così:

Primo principio di progettazione
La semplicità favorisce la regolarità.

In linea con questo principio, MIPS consente di effettuare operazioni aritmetiche solo nella loro forma più semplice, ossia con tre operandi: $a = b + c$. Quest'istruzione, comune in C, Java, Python o altri linguaggi, in MIPS sarà scritta così:

¹ `add a, b, c`

Notiamo subito che l'operando "di destinazione" viene messo come primo elemento (in altre ISA non è sempre così, tuttavia).

Naturalmente, mentre nei linguaggi ad alto livello possiamo agevolmente scrivere espressioni matematiche complesse, in MIPS verranno tutte mappate in espressioni semplici, sicché una linea di codice come $f = (g + h) - (i + j)$, per quanto banale, può essere srotolata in diverse righe assembly:

```

1 add t0, g, h    # t0 (variabile temp) = g + h
2 add t1, i, j    # t1 (variabile temp) = i + j
3 sub f, t0, t1   # f = t0 - t1

```

La sintassi per scrivere commenti consiste nell'anteporre un `#` al testo che si vuole inserire. Come per `//` in C, anche `#` agisce solo su una riga.

Esistono però alcune eccezioni: ad esempio `gcc`, l'assembler di Unix, usa i commenti come il C e l'operando di destinazione delle operazioni va in fondo.

6.3 I registri

Cosa sono i registri

Fino ad ora abbiamo considerato gli operandi come fossero normali variabili, ma in realtà questi altro non sono che dei *registri*, ossia delle particolari locazioni di memoria interne al processore che possono essere reperite molto rapidamente, in un solo colpo di clock.

MIPS possiede 32 registri, ciascuno di 32 bit, per cui ognqualvolta si debba eseguire delle operazioni è necessario caricare i dati dalla RAM con un'operazione di *load*. Questo processo può apparire dispendioso ed è lecito domandarsi come mai non ci si possa dotare di più registri più capienti, e la risposta è espressa dal secondo principio di progettazione.

Secondo principio di progettazione
Minori sono le dimensioni, maggiore è la velocità.

Di fatto, se avessimo più registri, aumenterebbe notevolmente il tempo di accesso a questi, in quanto gli impulsi elettrici impiegherebbero fisicamente più tempo per passare da un registro all'altro, e la velocità di clock sarebbe sensibilmente compromessa.

Gestione dei registri

Notazione A questo punto, tornando all'esempio in 1.2, illustriamo la sintassi per richiamare un registro:

```

1 add $t0, $s1, $s2  # al registro temporaneo t0 viene assegnato il
   ↳ valore come somma di s1 + s2
2 add $t1, $s3, $s4  # come prima
3 sub $s0, $t0, $t1  # f = t0 - t1

```

Come si evince, è sufficiente anteporre un `$` al nome del registro.

Movimenti

Naturalmente, i registri non sono minimamente sufficienti per contenere tutti i dati di un programma complesso (soprattutto perché alcuni vengono usati dal kernel e dal sistema operativo), ma le operazioni possono essere eseguite solo fra registri. Diventa quindi necessaria l'implementazione delle funzioni *load* che, come anticipato, carica i dati dalla RAM a un registro, e *store*, che passa il dato dal registro alla RAM.

La memoria è solitamente organizzata in gruppi da 8 bit (ossia 1 byte) ciascuno associato ad un indirizzo lineare progressivo, ma il prelievo e il salvataggio dei dati segue un cosiddetto *vincolo di allineamento*: viene fissato un *offset (spiazzamento)* di 4 byte (quindi della stessa dimensione dei registri), dimodoché ogni informazione che passa abbia la medesima dimensione di un registro (anche il program counter viene incrementato di 4 byte ogni fetch) e sia possibile accedere solamente ad indirizzi multipli di 4 byte. In questo contesto, ogni informazione di 4 byte che viene spostata tra RAM e registri viene detta *parola*.

L'indirizzo di ogni parola viene quindi espresso con una *base*, specifica per registro, e con lo *spiazzamento* costante di cui abbiamo parlato prima.

Load and store Ecco un esempio di istruzione di load (analogamente per la store):

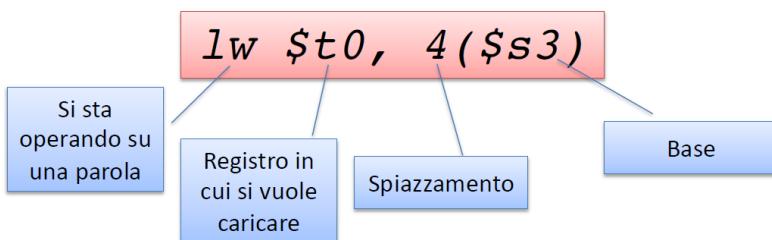


Figura 6.1: Descrizione di una load

Adesso prendiamo invece una semplice riga di codice:

1 `A[12] = h + A[8];`

E vediamone ora la traduzione fatta dal compilatore:

```

1  lw $t0, 32($s3)  # la parola 8 comincia all'indirizzo 32(=8*4) e
   ↳ il puntatore A è in $s3
2  add $t0, $s2, $t0 # h è in $s2
3  sw $t0, 48($s3)  # memorizzo il contenuto di $t0 in A[12]

```

Register Spilling Come già anticipato, i 32 registri non sono assolutamente sufficienti per contenere tutte le variabili usate da un programma complesso. Quello che succede a livello assembly, eseguito dal compilatore, è un continuo susseguirsi di load e store, chiamato *register spilling*. Il compilatore stesso stima il *working set* ed inserisce nel codice assembly le operazioni di load/unload appropriate.

Le costanti

Molto spesso ci si trova a lavorare con le costanti. Nonostante ogni costante possa essere salvata su un registro, questa soluzione è parecchio inefficiente, perché le costanti e le operazioni relative sono molto comuni.

Terzo principio di progettazione
Rendi più veloci possibile le operazioni comuni.

Quindi ci conviene assolutamente costruire una sintassi dedicata all'esecuzione di operazioni con costanti, dove la *i* sta per *immediate*. Ad esempio $f = f + 4$ diventa: `addi $s3, $s3, 4`.

Per dare un altro esempio, è molto conveniente implementare un registro zero (`$zero`), perché rende molto più semplici le operazioni di copia, che di fatto possono essere ridotte a una somma di un qualsiasi registro con il registro zero.

6.4 La rappresentazione delle istruzioni

Convenzioni sulla scrittura esadecimale

Prima di indicare come vengono codificate le istruzioni, ricordiamo che 1 byte è agevolmente rappresentabile con due cifre esadecimale (essendo composto da 8 bit e sapendo che ogni cifra esadecimale corrisponde a 4 bit). Ad esempio, 10011101_2 diventa $9D_{16}$. Inoltre, quando si scrive in esadecimale, si usa di solito precedere la scrittura con `0x`. Ad esempio: `0xEA01BD1C`.

Quando si devono codificare parole di 4 byte è importante decidere dove vada il byte più significativo:

- se il byte più significativo è posto per primo, la notazione è detta *big endian* (usata nei processori Motorola e protocolli internet).
- se il byte meno significativo è posto per primo, la notazione è detta *little endian* (usata nei processori Intel).

6.4.1 Le istruzioni register

Come anticipato, le istruzioni vengono codificate come numeri binari. In particolare, ogni istruzione dovrà essere mappata utilizzando parole di soli 32 bit, ma come viene impostato questo processo?

Prendiamo come esempio l'istruzione di somma `add $t0, $s1, $s2`: avremo bisogno di un codice per l'istruzione di somma e uno per ogni registro; l'operazione sarà scritta nei primi 6 bit e negli ultimi 6, mentre i 20 in bit in mezzo saranno divisi in gruppi da 5 bit ciascuno: i primi tre gruppi ospiteranno i registri, e l'ultimo sarà usato per comandi speciali (come lo shift, che vedremo in seguito).

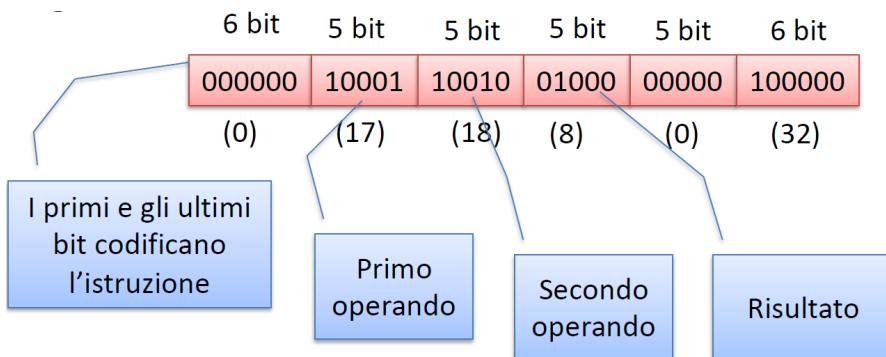
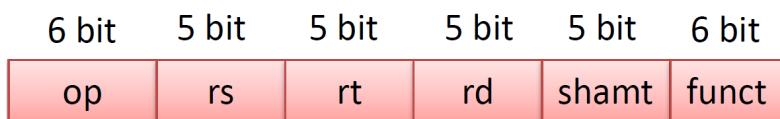


Figura 6.2: Mappatura di un'istruzione R, l'operazione di somma

In generale questo sistema, detto R (da registro), ogni campo dell'istruzione ha un nome:

- *op*: codice operativo dell'istruzione.
- *rs*: primo operando sorgente.
- *rt*: secondo operando sorgente.
- *rd*: operando di destinazione.
- *shamt*: shift amount (specifico per alcune istruzioni).
- *funct*: definisce la funzionalità specifica dell'istruzione (insieme ad *op*). Ad esempio la sottrazione è un particolare tipo di addizione.



A questo punto è lecito domandarsi come mai si usino solo 32 bit per rappresentare le istruzioni. La risposta si trova nel quarto principio di progettazione.

Quarto principio di progettazione*Un buon progetto richiede buoni compromessi.*

Utilizzando una quantità limitata di registri e istruzioni, è dunque possibile guadagnare moltissimo in efficienza!

6.4.2 Le istruzioni immediate

Nei casi di indirizzamento immediato e di operazioni con costanti, MIPS mette a disposizione una diversa mappatura, detta appunto *I*. Forniamo qui un esempio e invitiamo il lettore a confrontare la seguente mappatura con l'equivalente R.

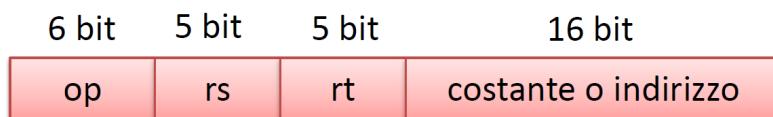


Figura 6.3: Mappatura di un'istruzione I

Prendiamo come esempio la riga di codice $A[300] = h + A[300]$. Essa verrà tradotta in MIPS come:

- 1 **lw \$t0, 1200(\$t1)**
- 2 **add \$t0, \$s2, \$t0**
- 3 **sw \$t0, 1200(\$t1)**

Osserviamo nel dettaglio cosa succede:

Istruzione	op	rs	rt	rd	Ind/Shamt	Funct
lw \$t0, 1200(\$t1)	35	9	8		1200	
add \$t0, \$s2, \$t0	0	18	8	8	0	32
sw \$t0, 1200(\$t1)	43	9	8		1200	

op	rs	rt	rd	Ind/Shamt	Funct
100011	01001	01000		0000 0100 1011 0000	
000000	10010	01000	01000	00000	100000
101011	01001	01000		0000 0100 1011 0000	

Figura 6.4: Trascrizione del codice sopra riportato, prima in decimale e poi in binario

6.5 Istruzioni aritmetico-logiche

Inizialmente i calcolatori erano capaci di operare unicamente su parole intere, ma ben presto si manifestò la necessità di andare ad agire solo su determinate porzioni di parole, o anche sui singoli bit. A seguire presenteremo alcune delle istruzioni che MIPS mette a disposizione qualora si presentasse la suddetta situazione.

6.5.1 Operazioni di shift

Shift logico a sinistra Consideriamo lo *shift logico a sinistra*: l'idea consiste nell'inserire degli zeri nella posizione meno significativa e traslare tutto a sinistra perdendo, nel caso di overflow, i bit più significativi.

¹ `sll $t2, $s0, 4 # memorizza in t2 il contenuto del registro s0
→ shiftato a sinistra di 4`

Lo shift massimo che è possibile eseguire è di 32 bit (infatti, come precedentemente visto, vengono allocati 5 bit per il campo di *shift amount*). Ciò non risulta un problema in quanto il vincolo di allineamento definito dall'implementazione MIPS che affronteremo è costituito da 32 bit.

Eseguire uno shift a sinistra di k bit di un numero n positivo, consiste nel moltiplicare n per 2^k . Fanno eccezione solo alcuni casi in cui si lavora con numeri interi codificati in complemento a 2: in questo frangente, si potrebbe uscire dal *range di rappresentazione* del CA2.

Shift logico a destra In maniera analoga, lo *shift logico a destra* consiste nell'inserire degli zeri nella posizione più significativa, traslando tutto a destra. Come per lo shift logico a sinistra, anche qui è possibile effettuare uno shift di al massimo 32 bit.

```
1 srl $t2, $s0, 1 # memorizza in t2 il contenuto del registro s0
    ↳ shiftato logicamente a destra di 1
```

Appicare uno shift logico a destra di k bit ad un numero n positivo è equivalente a dividere (divisione intera) n per 2^k . Lo shift logico a destra non è utilizzabile in nessun caso con un numero negativo codificato in CA2 (proprio per questo è stato introdotto lo shift aritmetico a destra).

Shift aritmetico a destra Dunque, per trovare una soluzione all'implementazione della divisione intera anche per i numeri negativi codificati in CA2, è stato introdotto lo *shift aritmetico a destra*.

```
1 sra $t2, $s0, 1 # memorizza in t2 il contenuto del registro s0
    ↳ shiftato aritmeticamente a destra di 1
```

Piuttosto che inserire 0 come bit più significativo, come avviene nello shift logico, lo shift aritmetico a destra inserisce bit uguali al bit di segno. Dunque risolve i problemi con CA2.

Si noti come non abbia senso parlare di shift aritmetico a sinistra, in quanto o il risultato è rappresentabile (attraverso lo shift logico) o non lo è (a causa di un overflow).

6.5.2 Operazioni bitwise

Vengono definite *bitwise* quelle operazioni logiche che operano su ciascun bit.

Bitwise AND L'operazione di *bitwise AND* viene applicata principalmente per forzare alcuni bit a 0 usando come operando una maschera (i cui bit corrispondenti a quelli che si vogliono annullare sono settati a 0).

<code>t1 = 0000 0000 0000 0000 0000 1101 0000 0000</code>

<code>t2 = 0000 0000 0000 0000 0011 1100 0000 0000</code>

<code>and \$t0, \$t1, \$t2</code>

Risultato:

<code>t0 = 0000 0000 0000 0000 0000 1100 0000 0000</code>

Figura 6.5: Funzione "maschera" del bitwise AND

Bitwise OR In maniera analoga, è possibile settare a 1 alcuni bit applicando l'operazione di *bitwise OR* con un operando che abbia 1 nella posizione corrispondente (funzione di maschera). Ad esempio, per impostare ad 1 i 4 bit più significativi di \$s0:

```

1 addi $t0, $zero, 0x000F
2 sll $t0, $t0, 28
3 or $s0, $s0, $t0

```

Un ulteriore esempio riguarda quello della *rotazione* dei 4 bit più significativi di una parola sui suoi bit meno significativi:

```

1 # si ottengono i 4 bit più significativi, spostandoli nei 4 meno
  → significativi di $t0
2 srl $t0, $s0, 28
3
4 # si azzerano i 4 bit più significativi di $s0
5 sll $s0, $s0, 4
6 srl $s0, $s0, 4
7
8 # operazione di maschera
9 or $s0, $s0, $t0

```

Bitwise NOR L'operazione di *NOR* viene definita come segue:

Input	Output
0 0	1
0 1	0
1 0	0
1 1	0

Tabella 6.1: Tabella di verità NOR

A seguire la sintassi MIPS dell'istruzione di *bitwise NOR*:

```

1 nor $t0, $t1, $t2

```

Bitwise NOT Dato il fatto che l'operatore di *NOT* è unario, esso non è implementato in MIPS, per non interrompere la regolarità delle istruzioni a tre operandi. Un modo per ottenere il *NOT*, grazie alle leggi di De Morgan, è il seguente:

```

1 nor $t0, $t1, $zero # equivale a: not $t1

```

OR esclusivo (XOR) L'*OR esclusivo* produce 1 se i due bit di ingresso sono diversi, altrimenti 0 se sono uguali. Segue la tabella di verità:

Input		Ouput
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 6.2: Tabella di verità XOR

A seguire la sintassi MIPS dell'operazione di *XOR*:

```
xor $t0, $t1, $t2 # $t0 = $t1 xor $t2
```

Curiosità: se si esegue lo *XOR* di un numero con se stesso, il risultato è pari a 0 per ciascun bit.

6.6 Salti

Oltre alle istruzioni aritmetico-logiche e di lettura e scrittura in memoria, il linguaggio MIPS offre anche alcune istruzioni per effettuare dei "salti" all'interno del codice. In particolare, si dicono *istruzioni di salto* quei comandi che possono modificare il valore del registro Program Counter (PC).

Esistono sostanzialmente due differenti tipi di salto:

- *jump*: salto non condizionato.
- *branch*: salto condizionato.

A differenza delle istruzioni di *jump*, i *branch* vengono eseguiti solo se una determinata condizione è verificata: uguaglianza di due registri (nel caso di `beq`) e disuguaglianza di due registri (nel caso di `bne`). Si definiscono perciò come operazioni che hanno la capacità di modificare il flusso del programma.

A seguire la sintassi delle istruzioni di salto offerte da MIPS:

```

1 # Branch Equal
2 beq reg1, reg2, L1 # vai a L1 se reg1 == reg2
3
4 # Branch Not Equal
5 bne reg1, reg2, L1 # vai a L1 se reg1 != reg2

```

```

6
7 # Jump
8 j LABEL # salta incondizionatamente a LABEL
9
10 # Jump Register
11 jr $t0 # salta incondizionatamente all'indirizzo contenuto nel
   ↳ registro t0

```

Come si può osservare, in fase di scrittura del codice si utilizzano delle *label* (ossia etichette) piuttosto che veri e propri indirizzi di memoria. L'arduo compito di tradurre queste etichette in veri e propri indirizzi di memoria spetta al compilatore.

Blocco di base Viene data la seguente definizione di *blocco di base*: sequenza di istruzioni che non contiene né istruzioni di salto (con l'eccezione dell'ultima) né etichette di destinazione (con l'eccezione della prima); sostanzialmente, una porzione di codice compresa tra due salti. In fase di compilazione ricoprono un ruolo non indifferente, in quanto una delle prime analisi eseguite sul codice consiste nel cercare proprio queste porzioni di codice. Inoltre, tutti i cicli dei linguaggi di programmazione ad alto livello vengono implementati con l'ausilio di blocchi base.

Confronti per minorazioni

Esistono inoltre delle istruzioni che permettono di settare dei bit nei registri, per utilizzarli come registri di *flag*, sulla base di una determinata condizione. In modo particolare osserviamo l'istruzione **slt** e la sua controparte immediata, sia in versione signed che unsigned:

```

1 slt $t0, $s3, $s4 # setta t0=1 se s3<s4
2 slti $t0, $s3, 10 # setta t0=1 se s3<10
3 sltu $t0, $s3, $s4 # unsigned: setta t0=1 se s3<s4
4 sltui $t0, $s3, 10 # unsigned: setta t0=1 se s3<10

```

Queste istruzioni vengono molto utilizzate dai compilatori MIPS per ottenere salti su condizioni di minore o maggiore uguale, con l'ausilio di **beq** e **bne**. A seguire viene presentato un esempio di traduzione da codice C ad assembly MIPS.

```

1 if (i < j){
2     f = g + h;
3 } else {
4     f = g - h;
5 }

```

```

1 INIZIO:
2     slt $t0, $s3, $s4      # setta $t0 se i < j
3     beq $t0, $zero, ELSE   # salta a ELSE se $t0 = $zero
4     add $s0, $s1, $s2      # f = g + h
5     j ESCI                 # salto incondizionato a ESCI
6 ELSE:
7     sub $s0, $s1, $s2      # f = g - h
8 ESCI:

```

Un'ulteriore utilità di queste istruzioni può essere espressa utilizzando i vettori: supponiamo di volere controllare se un indice (`$s1`) è fuori dal limite di un array (`[0, $t2]`).

```

1 sltu $t0, $s1, $t2  # importante l'uso dell'unsigned
2 beq $t0, $zero, FUORI_LIMITE

```

Infatti, se `$s1` è maggiore di `$t2`, ovviamente avremo 0 in `$t0`. D'altra parte, se `$s1` è negativo, interpretato come unsigned, sarà maggiore di `$t2` (che ha il bit più significativo a 0, essendo `$t2` necessariamente un numero positivo).

Implementazione del costrutto if

Supponiamo di voler tradurre il seguente listato di codice in assembly MIPS:

```

1 if (i == j){
2     f = g + h;
3 } else {
4     f = g - h;
5 }

```

A seguire viene presentata una possibile soluzione:

```

1 IF:
2     bne $s3, $s4, ELSE  # salta a ELSE se $s3 != $s4
3     add $s0, $s1, $s2    # f = g + h
4     j ESCI                 # salto incondizionato a ESCI
5 ELSE:
6     sub $s0, $s1, $s2    # f = g - h
7 ESCI:

```

Si noti che l'espressione di controllo consiste nel verificare il negato della condizione voluta: tale stratagemma risulta infatti essere più efficiente.

Implementazione del costrutto while

Supponiamo di voler tradurre il seguente listato di codice in assembly MIPS:

```

1 while (salva[i] == k){
2     i += 1;
3 }
```

A seguire viene presentata una possibile soluzione:

```

1 CICLO:
2     sll $t1, $s2, 2      # registro temp. $t1 = 4*i
3     add $t1, $t1, $s6    # ind. di salva[i] in $t1
4     lw $t0, 0($t1)       # carica salva[i] in $t0
5     bne $t0, $s5, ESCI   # esci se raggiunto limite
6     addi $s2, $s2, 1     # i = i+1
7     j CICLO
8 ESCI:
```

Si noti come il registro `$s2` venga utilizzato come contatore, e che `$s6` è l'offset di base del vettore `salva[]`.

Implementazione del costrutto switch/case

Piuttosto che eseguire una cascata di `if-else`, esiste una tecnica per implementare un costrutto `switch/case`: memorizzare i vari indirizzi del codice da eseguire in una tabella, caricando l'indirizzo a cui saltare in un registro, per poi utilizzare un `jr`.

A seguire un esempio di switch in codice C trasformato in assembly MIPS:

```

1 switch(a) {
2     case 1: <code 1>
3     case 2: <code 2>
4 }
5
1 sll $t0, $a0, 2  # moltiplica per 4
2 lw $t0, TABLE($t0)
3 jr $t0
```

6.7 Le procedure

L'utilità dei calcolatori sarebbe molto limitata se questi non avessero la possibilità di svolgere delle procedure; queste possono essere immaginate a tutti gli effetti

come delle funzioni che dato un certo input eseguono un determinato task a cui sono dedicate. Un primo vantaggio dell'utilizzo di questi costrutti deriva dalla modularizzazione che introducono in un programma, aumentandone la facilità di lettura e di scrittura.

L'aspetto fondamentale che si deve curare per dare la possibilità ai calcolatori di svolgere procedure è la definizione di un protocollo di chiamata delle procedure che deve stabilire con precisione questi aspetti:

- come caricare i parametri di input della procedura in locazioni note
- come trasferire il controllo alla procedura, che deve:
 - acquisire le risorse necessarie
 - eseguire il task affidatole
 - caricare gli output in locazioni note (sicché il chiamante della procedura sappia dove trovare i risultati)
 - restituire il controllo al chiamante
- come salvare il valore di ritorno della procedura e "fare pulizia" (ovvero eliminare i registri temporanei e ripristinare quelli da preservare)

I protocolli per la gestione delle procedure sono diversi in base all'architettura che si utilizza ed alle convenzioni di chiamata del compilatore, noi per ora studiamo il caso del MIPS.

6.7.1 Protocolli di chiamata MIPS

L'idea fondante del protocollo implementato da MIPS è di utilizzare ogniqualvolta sia possibile i registri, dato che essi sono il meccanismo più veloce a disposizione per la gestione dei parametri delle procedure. Ecco una tabella riassuntiva delle convenzioni sui registri definite nel MIPS:

Nome	Numero	Utilizzo	Preservare
\$zero	0	Valore costante 0	Costante
\$v0-\$v1	2-3	Valori di ritorno funzioni e valutazione espressioni	No
\$a0-\$a3	4-7	Argomenti	No
\$t0-\$t7	8-15	Temporanei	No
\$s0-\$s7	16-23	Salvati	Sì
\$t8-\$t9	24-25	Altri temporanei	No
\$gp	28	Global pointer	Sì
\$sp	29	Stack pointer	Sì
\$fp	30	Frame pointer	Sì
\$ra	31	Return address	Sì

Tabella 6.3: Convenzioni sui registri in MIPS

L'assembly MIPS mette a disposizione dell'utente l'utilissima istruzione *jump and link* (`jal`) che effettua un salto all'indirizzo di inizio della procedura specificata e contemporaneamente memorizza in `$ra` l'indirizzo di ritorno della procedura, dove si ritornerà una volta terminata la funzione. Questo significa che quando si chiama la `jal` si salva automaticamente in `$ra` il PC incrementato di 4 (l'istruzione successiva).

Alla fine della procedura sarà così sufficiente compiere il salto `jr $ra` per riprendere lo svolgimento del programma principale da dove lo si era interrotto.

6.7.2 Gestione della memoria in MIPS

Abbiamo già visto come i dati utilizzati da un programma in MIPS debbano essere caricati nei registri affinché il calcolatore possa usarli, ma cosa succede quando la memoria dei registri non basta?

In questo caso si utilizza lo stratagemma dello stack (esatto, proprio il buon vecchio stack, la nostra struttura dati di tipo LIFO preferita) implementato nella memoria del calcolatore; in MIPS vi si caricano i dati a partire da una posizione nota che viene puntata dal registro dedicato `$fp` (*frame pointer*, che indica quindi la base dello stack).

Viene utilizzato poi il registro `$sp` per puntare alla testa dello stack (dove vengono inseriti i nuovi elementi). I dati vengono caricati nello stack tramite operazioni di *push* ed, una volta utilizzati, eliminati tramite una operazione di *pop*.

All'interno di questa struttura dati si possono salvare variabili locali di procedure come anche registri che sarà necessario ripristinare in seguito.

6.7.3 Svolgimento di una procedura in MIPS

Per rendere in modo migliore la spiegazione dello svolgimento di una procedura in MIPS osserveremo come la seguente funzione viene tradotta dal linguaggio C in assembly MIPS:

```

1 int esempio(int g, int h, int i, int j){
2     int f;
3     f = (g+h)-(i+j);
4     return f;
5 }
```

Quando chiamiamo una procedura in MIPS per prima cosa il compilatore sceglie un'etichetta associata all'indirizzo di entrata della procedura (in questo caso *esempio*); in fase di collegamento l'etichetta è collegata ad un indirizzo.

La prima operazione è quella di salvare in memoria tutti i registri che la procedura sovrascriverà, in modo da poterli ripristinare in seguito; tale fase è chiamata *prologo* e potrebbe richiedere di allocare nello stack anche spazio per le variabili locali (in caso di spazio insufficiente nei registri).

Nell'esempio supponiamo, con lo scopo di osservare il metodo di salvataggio di un registro, di dover tenere in memoria i valori di *\$s0*, *\$t1* e *\$t0*.

ESEMPIO:

```

1 # decremento $sp di 12 byte creando lo spazio per 3 words
2 addi $sp, $sp, -12;
3
4
5 # salvo i registri nello stack prima che la funzione li sovrascriva
6 sw $t1, 8($sp)
7 sw $t0, 4($sp)
8 sw $s0, 0($sp)
9
10 # fine del prologo, inizio della procedura effettiva
11 # si suppone che g,h,i,j siano già salvate nei registri $a...
12 add $t0, $a0, $a1 # $t0=g+h"
13 add $t1, $a2, $a3 # $t1=i+j
14 sub $s0, $t0, $t1 # $s0=(g+h)-(i+j)
15
```

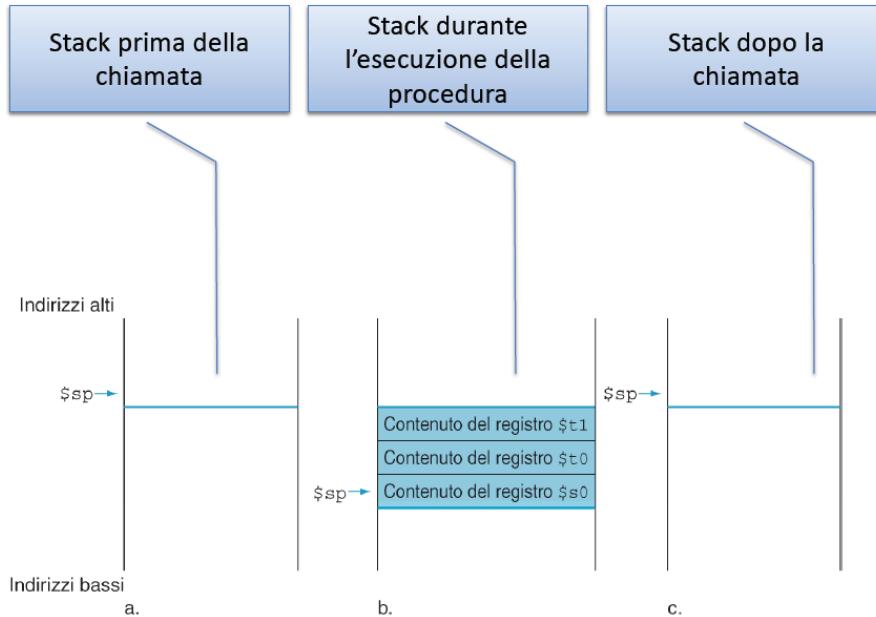
```

16      # imposto il valore di ritorno
17      add $v0, $s0, $zero  # $v0=$s0
18
19      # ($s0 è il registro che contiene il valore di f)
20      # pulizia: dopo le operazioni resetto i registri
21      lw $s0, 0($sp)
22      lw $t0, 4($sp)
23      lw $t1, 8($sp)
24
25      # ora posso liberare lo spazio dello stack dove stavano i registri
26      addi $sp, $sp, 12
27
28      # infine "salto" all'istruzione successiva alla chiamata di esempio
29      jr $ra

```

A titolo esplicativo si inserisce anche un'immagine dell'evoluzione dello stack durante lo svolgimento di questa funzione *esempio*.

Figura 6.6: Evoluzione dello stack durante l'esecuzione di *esempio*. L'ampliamento dello stack avviene sottraendo word a \$sp proprio perché lo stack cresce "verso il basso".



Una volta visto l'esempio vanno fatte alcune precisazioni. Infatti un compilatore effettivo non avrebbe *mai* svolto così la procedura, perché:

- i registri temporanei *non* devono essere preservati (quindi di norma non si usano istruzioni **sw** e **lw** su registri **\$t...**)
- l'utilizzo di **\$s0** non era necessario (si poteva usare anche solo **\$v0**)

Vediamo ora un esempio più complesso, dove si affrontano ostacoli come variabili locali e procedure annidate (a detta di tutti temutissime); la soluzione che si applica in questo caso è di salvare nello stack appunto tutte le variabili locali e tutti i valori del registro **\$ra** (per poi risalire la catena delle chiamate).

```

1 int fact(int n){
2     if (n < 1)
3         return 1;
4     else
5         return n*fact(n-1);
6 }
```

Si nota che implementando questa funzione il registro **\$ra** ed il registro **\$a0** vengano sovrascritti *n* volte. Vediamo quindi la traduzione del seguente esempio da C ad assembly MIPS:

```

1 FACT:
2     # decrementa $sp di 8 creando lo spazio per 2 words
3     addi $sp, $sp, -8;
4
5     # salva i registri $ra e $a0 nello stack
6     # ricorda: $ra contiene l'indirizzo di ritorno post-funzione
7     # ricorda: $a0 contiene il dato, in questo caso n
8     sw $ra, 4($sp)
9     sw $a0, 0($sp)
10
11    # fine del prologo, inizio della procedura effettiva
12    slti $t0, $a0, 1      # se $a0 < 1 setta $t0=1
13
14    # $t0 settato a 1 se n < 1
15    beq $t0, $zero, L1  # se $t0 == 0 si va a L1 (ricorsiva)
16
17    # altrimenti si prosegue con l'esecuzione finale
18    addi $v0, $zero, 1  # $v0 = 1
19    addi $sp, $sp, 8    # si ripulisce lo stack
20
21    # ritorna all'indirizzo dopo la chiamata (n-esima) di fact
```

```

22      jr $ra
23      # da questa istruzione parte la risalita della sequenza di $ra
24  L1:
25      addi $a0, $a0, -1  # decrementa n
26      jal fact          # chiama fact(n-1)
27
28      # a questo punto si ripristina $a0 e $ra e si ripulisce lo stack
29      lw $a0, 0($sp)    # ripristina $a0
30      lw $ra, 4($sp)    # ripristina $ra
31      addi $sp, $sp, 8  # ripulisce lo stack
32
33      # infine moltiplica n*fact(n-1), che è in $v0, e ritorna
34      mul $v0, $a0, $v0
35      jr $ra

```

6.7.4 Gestione delle variabili in MIPS

Le variabili in C sono in genere associate a locazioni di memoria e si caratterizzano per:

- *tipo*: (`int`, `char`, `float`, ecc.);
- *storage class*:
 - *automatic*: variabili locali che hanno un ciclo di vita collegato alla funzione
 - *static*: variabili globali o statiche che sopravvivono anche al termine della funzione.

Le variabili *statiche* sono memorizzate in una zona di memoria specifica (nel MIPS accessibile attraverso il registro global pointer, `$gp`).

Le variabili *automatiche* invece possono essere memorizzate nei registri, tuttavia quando questi non bastano, come già visto, le variabili vengono salvate all'interno dello stack; il segmento di stack che le contiene viene chiamato *record di attivazione* (o *stack frame*). Le variabili locali vengono individuate tramite un offset a partire da un puntatore allo stack.

Contrariamente a come si penserebbe, non si utilizza `$sp` come puntatore, poiché esso può variare all'interno dello svolgimento di una funzione, di conseguenza si utilizza il puntatore dedicato `$fp`.

L'immagine qui sotto rappresenta l'evoluzione dei due puntatori `$sp` e `$fp` durante la chiamata di una funzione, per capire meglio i vantaggi dell'utilizzo di `$fp`.

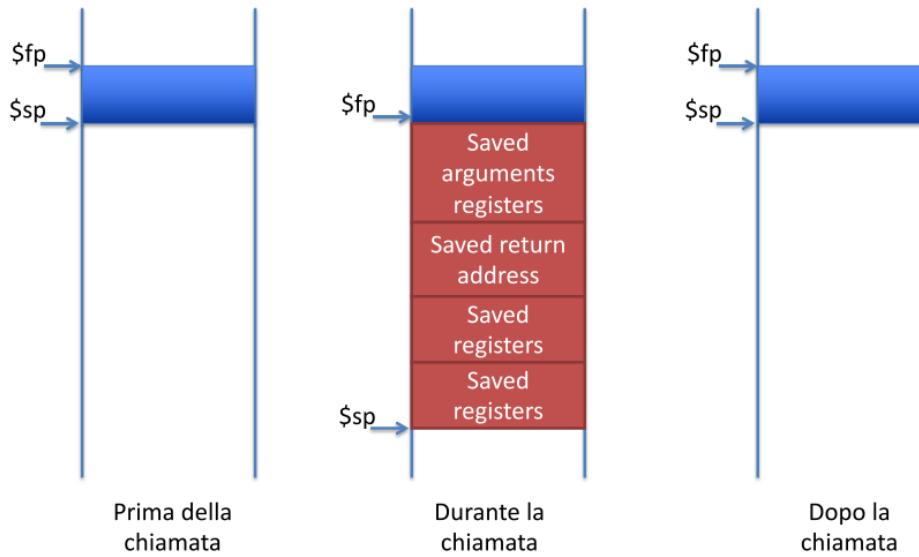


Figura 6.7: Confronto tra puntatore \$sp e \$fp

Esiste inoltre un ultimo tipo di variabili (oltre alle globali ed alle locali), quelle dinamiche, che vengono allocate secondo lo schema della figura sottostante dal MIPS; da notare il fatto che lo stack procede "decrescendo", mentre i dati allocati dinamicamente "crescono" partendo da dove terminano i dati statici (e quindi si spiega perché negli esempi precedenti abbiamo decrementato \$sp per inserire word nello stack).

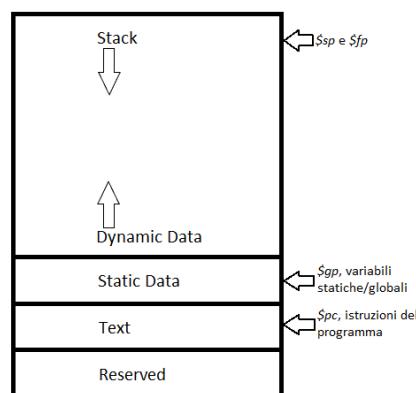


Figura 6.8: Schema allocazione dei dati in MIPS

6.7.5 Elaborazione delle procedure

Per spiegare come vengono elaborate le procedure complesse da MIPS utilizziamo come esempio la seguente funzione:

```

1 int sum(int n){
2     if (n > 0){
3         return sum(n-1) + n;
4     } else {
5         return n;
6     }
7 }
```

Se si analizza la funzione si nota come il valore di ritorno viene costruito risalendo la catena di chiamate e quindi ogni frame di attivazione deve essere preservato per produrre il risultato corretto. Ecco come compare il codice assembly della procedura *sum*:

```

1 SUM:
2     add $v0, $a1, $zero      # sposta $a1 in $v0
3     blez $a0, $L5            # se $a0<=0, salta a L5
4     addi $sp, $sp, -32        # crea spazio nello stack
5     sw   $ra, 28($sp)        # salva $ra
6     add  $a1, $a1, $a0        # (acc = acc + n)
7     addi $a0, $a0, -1         # (n = n - 1)
8     jal  SUM                 # ricorsione (aggiorna $ra)
9
10    # parte di risalita della catena di $ra
11    lw   $ra, 28($sp)        # ripristina $ra
12    addi $sp, $sp, 32          # ripulisce lo stack
13 L5:
14    j $ra                   # torna al programma chiamante
```

Come possiamo immaginare il procedimento scritto così è molto dispendioso in termini di memoria, ma possiamo, con un po' di esperienza e scaltrezza, riscrivere nel seguente modo il programma:

```

1 int sum(int n, int acc){
2     if (n > 0){
3         return sum(n-1, acc+n)
4     } else {
5         return acc;
6     }
7 }
```

In questo caso si applica il metodo di ricorsione in coda, che comporta un grande vantaggio dal punto di vista dell'utilizzo di memoria poiché una volta terminata una chiamata della funzione si può eliminare tutto il suo ambiente (che non risulterà più utile) e utilizzare sempre solo un ambiente unico ed un unico `$ra`, vediamo come:

```

1  SUM:
2      add $v0, $a1, $zero    # sposta $a1 in $v0
3      blez $a0, $L5          # se $a0 <= 0, salta a L5
4
5      add $a1, $a1, $a0      # (acc = acc + n)
6      addi $a0, $a0, -1       # (n = n - 1)
7      j  SUM                # ricorsione
8      # si osservi come non si usa più jal dato che non si
9      # aggiorna più $ra ora che si lavora su un unico ambiente
10     L5:
11         j $ra               # torna al programma chiamante

```

Alcuni compilatori sono in grado di ottimizzare automaticamente il programma in questo modo così da renderlo molto più veloce e leggero, il `gcc` è in grado di aggiustare, ove possibile, questo tipo di procedure con una complessità pari a $O(2)$.

6.8 Le stringhe in MIPS

Nel linguaggio C le stringhe erano degli array di caratteri che terminava in '`\0`' (`NULL` character) e, secondo la codifica ASCII, ogni carattere veniva codificato con un byte. Presentiamo qui una funzione che copia una stringa:

```

1 void copia_stringa(char *d , const char *s){
2     int i = 0;
3     while((d[i] = s[i]) != 0){
4         i+=1;
5     }
6 }
```

Appare chiaro che qui sovviene la necessità di copiare *bytes*, non *parole*. Vediamo quindi come implementare questa funzione in MIPS.

Load byte e store byte

Operare su byte anziché su parole intere può tornare utile anche in altre situazioni, per cui in MIPS disponiamo di istruzioni dedicate: *load byte* e *store byte*, analoghe a quelle che operano sulle words. Eccone un esempio:

```

1  lb $t0, 0($sp)  # leggi un byte dalla cima dello stack ed
   ↳ inseriscilo nei bit meno significativi di $t0
2
3  sb $t0, 0($gp)  # inserisci gli otto bit meno significativi di $t0
   ↳ in $gp

```

6.8.1 Possibile implementazione

Prologo

Come il lettore a questo punto saprà, i due parametri `d` e `s` sono contenuti in `$a0` e `$a1`. Supponiamo di voler utilizzare `$s0` per salvare il contatore `i` (anche se naturalmente un compilatore non lo farebbe *mai*, lo salverebbe su un registro temporaneo). Il registro `$s0` andrà dunque salvato nello stack e successivamente inizializzato:

```

1  COPIA_STRINGA:
2      addi $sp, $sp, -4      # crea spazio nello stack per salvare $s0
3      sw $s0, 0($sp)        # salva $s0
4      add $s0, $zero, $zero # i = 0

```

Loop

A questo punto procediamo all'implementazione del ciclo *while* che eseguirà la copia di `s[i]` in `d[i]`. Iniziamo mettendo l'indirizzo di `s[i]` nel registro `$t1`. (Nota: `&s[i] = s + i`, non `s + i * 4`).

A questo punto carichiamo `s[i]` in `$t2` e salviamolo in `d[i]`.

```

5  L1:
6      add $t1, $s0, $a1    # inizio loop
7      lb $t2, 0($t1)
8      add $t3, $s0, $a0    # metti (d+i) in $t3
9      sb $t2, 0($t3)

```

Ora che il corpo dell'istruzione è stato svolto, bisogna controllare se `s[i] == '\0'` e, in caso di risposta affermativa, il loop terminerà; altrimenti si incrementerà il contatore (che sta in `$s0`) e torna al ciclo.

```

10     beq $t2, $zero, L2  # se la stringa è finita, vai a L2 e termina
11     addi $s0, $s0, 1     # incrementa il contatore
12     j L1                 # torna al ciclo

```

Epilogo

In questa fase viene implementata la fine del ciclo e il ritorno al chiamante; si recupera il valore precedente di `$s0`, viene ripristinato `$sp` e tutto ritorna.

```

13  L2:
14      lw $s0, 0($sp)
15      addi $sp, $sp, 4
16      jr $ra

```

Mettendo insieme il tutto

Ricapitolando, ecco come sarà la nostra procedura:

```

1  COPIA_STRINGA:
2      addi $sp, $sp, -4
3      sw $s0, 0($sp)
4      add $s0, $zero, $zero
5  L1:
6      add $t1, $s0, $a1
7      lb $t2, 0($t1)
8      add $t3, $s0, $a0
9      sb $t2, 0($t3)
10     beq $t2, $zero, L2
11     addi $s0, $s0, 1
12     j L1
13  L2:
14      lw $s0, 0($sp)
15      addi $sp, $sp, 4
16      jr $ra

```

6.8.2 Implementazione realistica

Nessun compilatore avrebbe mai lavorato in questo modo. Abbiamo presentato la procedura così affinché fosse didatticamente più efficace, adesso invitiamo il lettore a dare un'occhiata a quella che potrebbe essere la traduzione di un vero compilatore, in particolare `gcc`:

```

1  COPIA_STRINGA:
2      lb $v0, 0($a1)
3      sb $v0, 0($a0)
4      beq $v0, $zero, L5
5      move $v0, $zero

```

```
6  L3:  
7      addiu $v0, $v0, 1  
8      addu $v1, $a1, $v0  
9      lb $v1, 0($v1)  
10     addu $a2, $a0, $v0  
11     sb $v1, 0($a2)  
12     bne $v1, $zero, L3  
13 L5:  
14     jr $ra
```

Capitolo 7

L'architettura Intel x86

“ “ “Do you prog in Assembly?”, she asked. “*NOP*”, he said.

” ” ”

Anonymous

7.1 Introduzione

Le CPU *Intel* della famiglia *x86* si basano su un'architettura di tipo CISC. Esse vengono utilizzate principalmente su laptop, desktop e server, a partire dagli anni Settanta. Un'importante complicazione dell'architettura *Intel* è dettata dal mantenimento della retrocompatibilità: le moderne CPU a 64 bit di ultima generazione sono infatti in grado di eseguire il vecchio codice a 8 bit. A differenza del *MIPS* infatti le istruzioni *Intel* non sono codificate in una singola parola, bensì possono occupare da 8 a 64 bit.

Come già ampiamente trattato nei precedenti capitoli, un'architettura di tipo CISC è in grado di offrire un notevole set di istruzioni e un potente meccanismo di indirizzamento; ciò implica che load e store non sono più le uniche operazioni che permettono di accedere alla memoria.

Per semplicità, in questa dispensa si tratterà l'ISA più moderna, chiamata *x86-64*, nota anche come *amd64*. Essa è caratterizzata da parole a 64 bit e da 16 registri. Fra le varie ABI verrà utilizzata quella di Linux, che differisce per qualche aspetto da quella dei sistemi Microsoft e Apple.

7.2 La gestione dei registri

Per indicare i registri si antepone un % al nome del registro, mentre per indicare le costanti si antepone un \$ al valore della costante stessa.

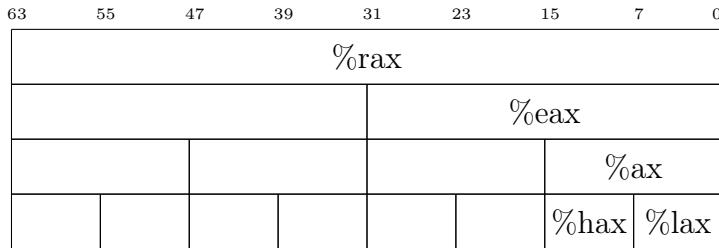


Figura 7.1: Il registro `%rax` e i suoi sottoregistri

7.2.1 I registri general purpose

La versione di *Intel* analizzata in questa dispensa è caratterizzata da 16 registri a 64 bit *general purpose*. Si possono dividere in:

- derivanti dall'architettura 8080: `%rax` (accumulatore), `%rbx`, `%rcx`, `%rdx`;
- derivanti dall'architettura 8086: `%rsi` (source index), `%rdi` (destination index), inizialmente concepiti per la copia di array;
- puntatore allo stack frame (base pointer): `%rbp`;
- puntatore allo stack pointer: `%rsp`;
- introdotti in *x86-64*: `%r8`, `%r9`, `%r10`, `%r11`, `%r12`, `%r13`, `%r14`, `%r15`.

7.2.2 I registri specializzati

Oltre ai registri general purpose, si aggiungono i registri specializzati:

- `%rip` (*instruction pointer*, ossia il program counter per *Intel*);
- `%rflags` (*flag register*, che contiene i *flag* per i salti condizionali).

Si noti che `%rflags` estende `%eflags`, il quale a sua volta estende `%flags`. Alcuni flag contenuti nel registro di flag sono: CF, ZF, SF, OF, IF.

7.3 Le convenzioni di chiamata

Come già ripetuto in molteplici occasioni, le convenzioni di chiamata vengono specificate nell'ABI. In particolare, noi utilizzeremo quella di Linux. Principalmente andremo a definire i meccanismi per il passaggio di argomenti, per la restituzione di valori e quali registri da preservare:

- il passaggio di argomenti si compone di:
 - i primi 6 argomenti nei registri: `%rsi`, `%rdi`, `%rcx`, `%rdx`, `%r8`, `%r9`;
 - ulteriori argomenti possono essere salvati nello stack;
- i valori di ritorno vengono salvati nei registri `%rax` e `%rdx`;
- i registri da preservare sono: `%rbp`, `%rbx`, `%r12`, `%r13`, `%r14`, `%r15`.

Esistono due modalità per richiamare una funzione attraverso l'istruzione `call`:

- `call address`: invoca la subroutine presente all'indirizzo indicato;
- `call *register`: invoca la subroutine presente all'indirizzo contenuto nel registro indicato.

Una caratteristica di *Intel* è quella di salvare automaticamente l'*instruction pointer* (che ha un funzionamento analogo al *link register* di MIPS) nello stack quando una funzione viene richiamata, facilitando di molto la scrittura di funzioni ricorsive¹. In maniera analoga, l'istruzione `ret`, che viene utilizzata per terminare una funzione, preleva automaticamente dallo stack l'indirizzo da utilizzare nell'instruction pointer.

7.4 Le modalità di indirizzamento

Esistono svariate modalità di indirizzamento in *Intel*:

- `<displacement>;`
- `[<displacement>](<base register>);`
- `[<displacement>](<index register>, [<scale>]);`
- `[<displacement>](<base register>, <index register>, [<scale>]).`

¹Si ribadisce che in MIPS esiste un vero e proprio registro che si occupa di questo (*link register* appunto). Nel caso di funzioni non foglia però si deve provvedere manualmente a salvarne e recuperarne il valore dallo stack

dove `<displacement>` e `<scale>` sono valori immediati mentre `<base register>` e `<index register>` stanno ad indicare due registri generici. L'indirizzo di memoria corrispondente viene calcolato come segue:

$$<\text{displacement}> + <\text{base}> + <\text{index}> \cdot <\text{scale}>$$

dove:

- `<displacement>` è una costante a 8, 16 o 32 bit;
- `<base>` è un registro;
- `<index>` è un registro;
- `<scale>`² è 1, 2, 4 o 8.

7.5 La sintassi delle istruzioni

In generale si può affermare che molte istruzioni si presentano nella forma:

$$<\text{opcode}>[<\text{size}>] <\text{source}>, <\text{destination}>$$

Dunque, il secondo elemento è sia un operando che la destinazione del risultato dell'operazione: in quanto tale esso deve essere un registro o un indirizzo di memoria. Il primo operando, oltre a poter essere un registro e un indirizzo di memoria, può essere anche un valore immediato. In particolare esistono però due vincoli:

- i due operatori non possono contemporaneamente essere indirizzi di memoria;
- non è possibile specificare due operandi e una destinazione diversa.

Il suffisso `<size>` può essere usato per indicare l'ampiezza in bit degli operandi:

- `b` significa ampiezza di 8 bit (*byte*);
- `w` significa ampiezza di 16 bit (*word*);
- `l` significa ampiezza di 32 bit (*long word*);
- `q` significa ampiezza di 64 bit (*quad word*).

Esso è opzionale quando uno dei due operandi è un registro, mentre diventa obbligatorio nel caso l'istruzione non presenti nessun registro come operando. Un esempio in cui `<size>` è obbligatorio: `notl 200(%rbp)`.

²Attenzione: al momento della pubblicazione del nostro elaborato, la dispensa "Piccola introduzione al Linguaggio Assembly e altre Simili Amenità" di Abeni riporta la notazione errata.

7.6 Alcune istruzioni frequenti

A seguire un elenco delle istruzioni aritmetico-logiche più frequenti:

- **mov** per scambiare dati fra memoria e registri e viceversa (la destinazione non può essere un valore immediato);
- **push** e **pop** per leggere e salvare dati sullo stack, senza dover modificare lo stack pointer come avviene in MIPS;
- **add** (somma) e **addc** (somma con carry);
- **sub** (sottrazione) e **subc** (sottrazione con carry);
- **mul** (moltiplicazione con segno) e **imul** (moltiplicazione senza segno);
- **div** (divisione con segno) e **idiv** (divisione senza segno);
- **inc** (somma 1) e **dec** (sottrae 1);
- **and**, **or**, **xor** e **not**: operazioni logiche bit a bit;
- **lea**: *load effective address*;
- **rcl**, **rcr**, **rol**, **ror**: varie forme di *rotate*;
- **sal**, **sar**, **shl**, **shr**: *shift* aritmetico e logico;
- **jmp**: salto incondizionato;
- **je** (jump if equal), **jnz** (jump if not zero), **jc** (jump if carry), **jnc** (jump if not carry);
- **neg**;
- **cmp**: setta i flag facendo una sottrazione, ma senza salvarne il risultato;
- **call** e **ret**: indirizzo di ritorno sullo stack;
- **nop**;
- eventuali istruzioni condizionali.

7.6.1 Load Effective Address (LEA)

Nonostante questa istruzione sia stata implementata per calcolare indirizzi (con indirizzamento indiretto) senza effettuare accessi, viene utilizzata principalmente per un secondo scopo. Essa infatti risulta molto utile per effettuare una somma di due registri, salvandone il risultato in un terzo (anche con eventuali shift). Ricordiamo che, mentre in MIPS lavorare con tre registri è la normalità, in Intel per ciascuna istruzione possono esserne specificati al massimo due.

Ad esempio:

```
1 lea (%rbx, %rcx), %rax
```

esegue la somma di `%rbx` e `%rcx` e la salva nel registro `%rax`.

Si noti come nonostante la sintassi sia identica a quella per l'indirizzamento, in questo caso non si sta eseguendo un accesso alla memoria. La stessa sintassi, in un'altra istruzione, avrebbe portato ad accedere al blocco di memoria contenuto in $(%rcx + \%rbx)$.

7.6.2 Incremento e decremento

Come avete potuto notare dalla lista delle istruzioni comuni in Intel, sono presenti le istruzioni `inc` e `dec`, che rispettivamente incrementano e decrementano di una unità il valore del registro specificato. Ma per quale motivo non si utilizza la semplice istruzione `add $1, register`?

In Intel si è visto che le istruzioni possono essere codificate con una sequenza di bit di lunghezza variabile. Utilizzare la funzione di incremento piuttosto che quella di somma significa non dover memorizzare un valore immediato, con il conseguente risparmio di 16 bit.

Capitolo 8

L'architettura ARM

“ When someone says: "I want a programming language in which I need only say what I wish done", give him a lollipop. ”

Alan J. Perlis

8.1 Introduzione

ARM nasce negli anni Ottanta come architettura RISC pragmatica, ovvero tenta di migliorare la strategia RISC implementando delle istruzioni e delle tecniche appartenenti ad *Intel*: è proprio questa ricerca di un compromesso tra prestazioni e complessità il suo punto di forza.

Un'altra caratteristica peculiare dell'*ARM* è la presenza di una modalità *low power*, che limita di molto il consumo di energia restringendo le istruzioni utilizzabili solo a quelle più semplici (e quindi meno costose in termini di risorse). È proprio per questa ragione che l'*ARM* ha avuto un'enorme diffusione come architettura per i dispositivi *mobile* e *wearable* oltre che per i sistemi *embedded*.

Per amor di precisione ci teniamo a specificare che in realtà "architettura ARM" non è una dicitura univoca in quanto esistono più CPU *ARM* con ISA differenti (per non parlare del numero spropositato di ABI anche molto diverse tra loro che si possono trovare per questa architettura).

8.2 Registri e relativo utilizzo

In ARM ci sono 16 registri a 32 bit e sono nominati da **r0** a **r15** (da notare che in ARM per indicare un registro non serve un prefisso). Alcuni di essi hanno

generalmente funzioni specifiche, ovvero:

- **r13**: *stack pointer*, contiene il puntatore allo stack;
- **r14**: *link register*, che contiene l'indirizzo di ritorno delle subroutine (come **\$ra** in MIPS);
- **r15**: *program counter* (nei bit più significativi contiene anche dei flags).

Nonostante sia molto diffuso, questo standard non è assoluto: di fatto esistono ABI che implementano l'utilizzo dei registri in modo diverso.

8.3 Le convenzioni di chiamata

Le convenzioni di chiamata, che vengono specificate appositamente dall'ABI, servono per "mettere d'accordo" diversi compilatori/librerie e altre parti del sistema operativo: sono esse che determinano se passare i parametri solo tramite registri o anche via stack, quali registri preservare, quali registri utilizzare liberamente e come gestire l'indirizzo di ritorno. A seguire ecco cosa stabilisce l'ABI più diffusa del mondo ARM:

- **r0, r1, r2, r3, r12** sono registri temporanei;
- da **r4** a **r11** sono da preservare, con l'eccezione di **r9** che in alcune implementazioni non viene preservato.

8.4 Le istruzioni ARM

Le istruzioni dell'Assembly ARM sono molto simili a quelle del buon vecchio MIPS, tuttavia sono meno regolari in quanto alcune di esse presentano solamente due argomenti; è inoltre notevole il fatto che, nonostante venga classificato come architettura RISC, ARM presenta una lista di istruzioni piuttosto corposa.

Come nel MIPS, le istruzioni a tre argomenti prevedono il primo come destinazione e i successivi come operandi; inoltre l'operando di sinistra dev'essere un registro mentre quello di destra può essere sia un registro che un valore immediato. Da notare che i dati in memoria non possono essere usati direttamente come operandi: risolveremo questo problema in seguito, introducendo delle operazioni specifiche *load/store register*.

Tutte le istruzioni in *ARM* permettono esecuzione condizionale grazie all'inserimento di determinati suffissi al termine del nome identificativo dell'istruzione:

l'esecuzione dell'istruzione stessa avviene solo se i bit del registro di flag corrispondono alla condizione desiderata; qui è riportata una tabella con i principali suffissi ed il loro funzionamento:

Suffisso	Condizione verificata
<code>eq</code>	equal
<code>ne</code>	not equal
<code>lo</code>	lower
<code>hs</code>	higher or same
<code>mi</code>	minus (i.e. negative)

Tabella 8.1: Suffissi condizionali ARM

I flag stessi possono essere modificati da apposite operazioni come `cmp` (*compare*), o da operazioni a cui si aggiunge il suffisso opzionale `-s`, che va a settare i bit del registro di flag se si registrano determinate condizioni (ad esempio un overflow durante una somma).

Lo schema base di un'istruzione è il seguente:

`<opcode>[<-cond>] [<-s>] <rd>, <rl>, <r*>`

dove `<-cond>` ed `<-s>` sono facoltativi, `<rd>` è il registro di destinazione, `<rl>` è il registro di sinistra e `<r*>` è il registro di destra. Si noti inoltre che `<r*>` può essere sia un registro che un valore immediato che un registro su cui si effettuano operazioni di modifica (si veda 8.4.1). Tuttavia, a differenza di MIPS, ARM non ha bisogno di due opcode distinti per definire l'operazione di somma: in ARM è possibile utilizzare `add` sia per sommare due registri che per sommare un valore immediato ad un registro.

Se il secondo operando è un registro, si possono inserire al termine dell'operazione dei comandi ulteriori che permettono di eseguire operazioni di shifting o rotazione su di esso (che avvengono prima dell'operazione), ad esempio `lsl/asl`, `lsr/asr`, `ror`, `rrx` (si veda sempre 8.4.1).

Le istruzioni più comuni

Vediamo ora una lista delle operazioni più comuni:

- `add/adc`: rispettivamente somma e somma con carry;

- **sub/sbc**: rispettivamente sottrazione e sottrazione con carry;
- **rsb/rsc**: *reverse sub/reverse sub with carry* che permettono una sottrazione invertendo il primo ed il secondo operando. In questo modo risulta possibile sottrarre un registro ad un numero;
- **and/orr/eor/bic**: operazioni booleane bit a bit. Si noti che **bic r0, r1, r** calcola $r0 = (r1 \text{ and } (\text{not } r))$;
- **mul/mla** e simili sono per varie forme di moltiplicazione¹;
- **b/bl** rispettivamente *branch* e *branch and link* (da notare che è possibile aggiungere qualsiasi suffisso condizionale);
- **bx** è implementato solo in alcune CPU, corrisponde a **move r15, r**;
- **cmp** setta i flags comparando due operandi, ma senza risultato; da usarsi prima di salti condizionali. Funzionano similmente anche **tst** (attraverso un and) e **teq** (attraverso uno xor);
- **ldr/str** sono rispettivamente *load register* e *store register*. Rispecchiano rispettivamente **lw** e **sw** dell'assembly MIPS;
- **mov/mvn**: equivalente agli omonimi move e move not in MIPS; move not sposta il complemento a 1 del registro sorgente (utilizzata per implementare il not).

8.4.1 Possibili operazioni sull'operando <r*>

Prima di mostrare le operazioni possibili su <r*>, ci teniamo a specificare che queste sono operazioni di shifting e di rotating e vengono applicate al secondo operando prima dello svolgimento dell'operazione descritta dall'istruzione.

Ecco un esempio di un'operazione in diverse versioni, per capire al meglio la potenza di ARM:

```

1  add r0, r1, #1
2  adds r0, r1, r2
3  addeq r0, r1, r2, lsl #2
4  addeqs r0, r1, r2, lsl r3

```

Analizziamo le righe di codice una ad una:

1. $r0 = r1 + 1$;

¹Non esistono comandi per la divisione in assembly ARM.

2. $r0 = r1 + r2$ e setta dei flag eventualmente;
3. $r0 = r1 + r2$: esegue la somma solo se il flag **eq** è settato, prima dell'operazione esegue uno shift (temporaneo) a sinistra di 2 del registro **r2**;
4. $r0 = r1 + r2$: esegue la somma solo se il flag **eq** è settato, setta flag eventualmente, prima dell'operazione esegue uno shift (temporaneo) a sinistra di **r3** del registro **r2**.

Istruzioni di load/store multiple

L'Assembly ARM permette il salvataggio in memoria o il caricamento da memoria di più registri con una sola istruzione, **ldm** e **stm** per l'appunto. Le istruzioni appena citate hanno la seguente struttura:

```
ldm [-mode] [-cond] rb[!], (lista dei registri)
```

dove **rb** è il registro che contiene la base, **-cond** è l'eventuale suffisso condizionale di cui si è già discusso in precedenza, mentre il suffisso (facoltativo) **mode** può essere:

- **ia**: *increment after*, incrementa il registro base dopo l'esecuzione di **ldm**;
- **ib**: *increment before*, incrementa il registro base prima dell'esecuzione;
- **da**: *decrement after*, decrementa il registro base dopo l'esecuzione;
- **db**: *decrement before*, decrementa il registro base prima dell'esecuzione.

Il **!**, facoltativo anch'esso, serve per rendere definitive le modifiche apportate dai suffissi appena elencati, dopo l'operazione.

8.5 Modalità di indirizzamento in ARM

Introduciamo ora le modalità d'indirizzamento in ARM, ricordando che si tratta di istruzioni che rendono possibile modificare l'indirizzo indicato dal secondo registro di operazioni come **ldr** e **str**; si tenga presente che le operazioni di indirizzamento sono diverse dalle operazioni di shifting o rotating.

In modo semplificato esistono due metodologie di indirizzamento in ARM, ovvero **base+offset** (spiazzamento, valore immediato) o **base+index** (indice, eventualmente scalato). La struttura dell'istruzione è la seguente:

```
1 ldr rd, rb (offset) ; base + spiazzamento  
2 ldr rd, rb (index) [shift] ; base + indice shiftato
```

dove **rd** è il registro destinazione, **rb** è il registro contenente l'indirizzo base, **offset** è un immediato codificato su 12 bit e **index** è un valore in registro, eventualmente shiftato o ruotato.

Una possibilità molto interessante offerta da ARM è l'utilizzo dei meccanismi di *pre indexing* e *post indexing*, vediamo come funzionano:

- *pre indexing con offset*: **[rb, #i]!**, in questo caso la somma tra **rb** e **i** viene effettuata prima dell'indirizzamento, il **!** è facoltativo e se è presente finalizza la somma (il valore di **rb** dopo l'esecuzione è dato da **rb + i**);
- *post indexing con offset*: **[rb], #i!**, in questo caso la somma tra **rb** e **i** viene effettuata dopo l'indirizzamento, il **!** se è presente finalizza la somma (il valore di **rb** dopo l'esecuzione è dato da **rb + i**);
- *pre indexing con indice e shift*: **[rb, ri, shift]!**, in questo caso la somma tra **rb** e **ri** shiftato (si ricorda che lo shift è facoltativo) viene effettuata prima dell'indirizzamento, il **!** se è presente finalizza la somma;
- *post indexing con indice e shift*: **[rb], ri, shift !**, in questo caso la somma tra **rb** e **ri** eventualmente shiftato viene effettuata dopo dell'indirizzamento.

Tirando le somme sui metodi di indirizzamento, si può dire che sono senz'altro più potenti di quelli offerti da *MIPS*, soprattutto grazie alla possibilità di aggiornare automaticamente il registro di base, che diventa super comoda quando si devono gestire degli array. Rispetto a x86, invece, ARM ha la limitazione di non poter usare contemporaneamente spiazzamento ed indice.

Capitolo 9

Programmi Assembly comparati

“Talk is cheap. Show me the code.”

”

Linus Torvalds

Ora andiamo a riprendere alcuni esempi già portati nel capitolo dedicato all’Assembly MIPS e andremo a confrontare le implementazioni proposte con le corrispettive per Assembly x86 e ARM. Per completezza, specifichiamo che il codice che mostriamo è stato ottenuto con il compilatore `gcc`, lievemente modificato affinché risulti più leggibile (principalmente sull’ordine logico delle istruzioni e sulla nomenclatura dei registri).

9.1 Semplici espressioni aritmetico-logiche

Cominciamo con delle semplici operazioni aritmetiche, che evidenzieranno come le architetture RISC (MIPS e ARM) permettono di eseguire questo tipo di istruzioni in modo molto naturale, al contrario dell’implementazione leggermente più macchinosa di x86. L’espressione è la seguente:

$$f = (g + h) - (i + j)$$

Implementazione MIPS

In MIPS assumiamo che le variabili `g`, `h`, `i` e `j` vengano mappate nei 4 registri dedicati agli argomenti, rispettivamente `$a0`, `$a1`, `$a2` e `$a3`, e decidiamo di salvare il risultato in `$v0`. A questo punto la conversione Assembly è immediata:

```

1 add $a0, $a0, $a1    # memorizza  $g + h$  in $a0
2 add $v0, $a2, $a3    # memorizza  $i + j$  in $v0
3 sub $v0, $a0, $v0    # memorizza  $(g + h) - (i + j)$  in $v0

```

Implementazione x86

Siano g , h , i e j mappate rispettivamente in `%rdi`, `%rsi`, `%rcx` e `%rdx` e si voglia salvare il risultato in `%rax`.

Il problema di x86 sta nel fatto che le operazioni sono solo a due operandi, con il secondo che funge da destinazione, e quindi non è possibile specificare un diverso registro di destinazione.

Possiamo aggirare questa limitazione sfruttando l'istruzione `lea`: questa infatti ha come argomenti `<addr>`, `<dst>` e calcola l'indirizzo `<addr>` secondo base, spiazzamento e indice shiftato, e salva tutto in `<dst>`. Sarà quindi sufficiente impostare spiazzamento e shift a 0 per riuscire a sommare agevolmente base (per noi `%rdi`) a indice non shiftato (`%rsi`) e salvarne il risultato in `%rax`.

```

1 leaq (%rdi, %rsi), %rax    # %rax =  $g + h$  con il trick visto sopra
2 addq %rcx, %rdx            # %rdx =  $i + j$ 
3 subq %rdx, %rax           # %rax =  $(g + h) - (i + j)$ 

```

Implementazione ARM

Analogamente a MIPS, anche l'implementazione ARM è molto semplice e naturale, una volta mappate g , h , i e j in `r0`, `r1`, `r2` e `r3` e decidendo di salvare il tutto in `r0`.

Qui vengono usate `adds` e `subs`, che aggiornano i flag, ma usare le normalissime `add` e `sub` non avrebbe cambiato nulla.

```

1 adds r1, r0, r1 ; r1 =  $g + h$ 
2 adds r3, r2, r3 ; r3 =  $i + j$ 
3 subs r0, r1, r3 ; r0 =  $(g + h) - (i + j)$ 

```

9.2 Array e accesso alla memoria

Ora invece prenderemo come esempio una semplice istruzione che agisce su un array per mostrare diverse modalità di accesso alla memoria: questa volta, x86 permetterà di ridurre notevolmente il numero di righe di codice, mentre le due ISA RISC richiederanno molte più istruzioni.

`a[12] = h + a[8];`

Implementazione MIPS

Come già visto, in MIPS sarà necessario utilizzare le istruzioni di `lw` e `sw` per accedere all'indirizzo dell'array; inoltre, ricordiamo che lo shift necessario a ottenere l'indirizzo desiderato andrà sempre moltiplicato per lo spiazzamento di 4 byte, sicché ogni parola risulti lunga 32 bit.

Assumendo quindi che `h` stia in `$a0` e l'indirizzo dell'array `a` in `$a1` otteniamo:

```

1 lw $v0, 32($a1)      # carica in $v0 il contenuto di a incrementato
   ↳ di 4 * 8
2 add $a0, $v0, $a0    # salva in $a0 a[8] + h
3 sw $a0, 48($a1)    # ripone il contenuto di $a0 nell'indirizzo di a
   ↳ incrementato di 4 * 12

```

Implementazione x86

La capacità di accedere direttamente alla memoria grazie a istruzioni come `addl` e `movl` semplifica di molto questa operazione per l'ISA Intel.

Assumendo che `h` stia `%edi` e l'indirizzo di `a` in `%rsi`, otteniamo:

```

1 addl 32(%rsi), %edi  # %edi = a[8] + h
2 movl %edi, 48(%rsi) # sposta a[8] + h in a[12]

```

Implementazione ARM

La versione ARM è molto simile a quella MIPS.

Assumendo che `h` stia in `r0` e l'indirizzo di `a` in `r1`, otteniamo:

```

1 ldr r3, [r1, #32] ; carica a[8] in r3
2 add r0, r3, r0      ; salva in r0 a[8] + h
3 str r0, [r1, #48] ; riponi r0 in a[12]

```

9.3 Blocchi condizionali

Vediamo ora la gestione di istruzioni condizionali. Solitamente queste vengono gestite da istruzioni di salti condizionati, ma alcune ISA hanno modalità di gestione decisamente peculiari.

Condizioni di uguaglianza

Andiamo ad analizzare le traduzioni del seguente codice C:

```

1 if (i == j){
2     f = g + h;
3 } else {
4     f = g - h;
5 }
```

Implementazione MIPS

In MIPS assumiamo che le variabili g , h , i e j vengano mappate nei 4 registri dedicati agli argomenti, rispettivamente $\$a_0$, $\$a_1$, $\$a_2$ e $\$a_3$, e decidiamo di salvare il risultato in $\$v_0$.

Utilizzeremo quindi **bne** per operare un confronto fra due registri general purpose per verificare $i == j$; qualora la condizione non fosse verificata, faremo il salto a L2 e alla sottrazione $g - h$. Differentemente, verrà eseguita la somma $g + h$ e un salto incondizionato **j** farà terminare la sequenza.

```

1 bne $a2, $a3, L2    # salta a L2 se $a2 e $a3 sono diversi
2 add $v0, $a0, $a1
3 j L3                 # salta a L3
4 L2:
5 sub $v0, $a0, $a1
6 L3:
```

Implementazione x86

Siano g , h , i e j mappate rispettivamente in $\%rdi$, $\%rsi$, $\%rcx$ e $\%rdx$ e si voglia salvare il risultato f in $\%rax$.

L'implementazione risulta più complessa rispetto a quella di MIPS, poiché dovremo usare **lea** al posto di **add** e soprattutto dovremo affiancare alla **sub** una **mov** per far sì che il risultato vada salvato in $\%rax$. Inoltre, è necessaria una **cmpq** per settare i flag e solo dopo possiamo lanciare l'istruzione di salto.

```

1 cmpq %rcx, %rdx  # confronta i e j e setta il flag
2 jne L2            # se i != j, salta a L2
3 leaq (%rdi, %rsi), %rax  # salva in %rax g + h
4 jmp L3
5 L2:
6 movq %rdi, %rax  # sposta g da %rdi a %rax
7 subq %rsi, %rax  # salva in %rax
8 L3:
```

Tutto questo può però essere semplificato dalla magica istruzione **cmove** (**mov** condizionale) di x86:

```

1 leaq (%rdi, %rsi), %rax    # salva g + h in %rax
2 subq %rdi, %rsi           # salva g - h in %rsi
3 cmpq %rcx, %rdx          # i == j
4 cmovne %rsi, %rax        # sposta %rsi in %rax se il i è diverso da j

```

Implementazione ARM

L'implementazione ARM ci permette di evitare istruzioni di salto grazie alla sua versione condizionale delle operazioni aritmetiche. Assumendo che g , h , i e j vengano mappate rispettivamente in $r0$, $r1$, $r2$ e $r3$ e che il risultato vada in $r0$, otteniamo:

```

1 cmp r2, r3      ; setta i flag
2 addeq r0, r0, r1 ; se i == j, salva g + h in r0
3 subne r0, r0, r1 ; altrimenti, salva g - h in r0

```

Condizioni di disuguaglianza

Vediamo ora invece una condizione di disuguaglianza:

```

1 if (i < j){
2     f = g + h;
3 } else {
4     f = g - h;
5 }

```

Implementazione MIPS

Assumendo che le variabili vengano mappate come sopra, il codice MIPS non appare troppo diverso, se non per l'utilizzo dell'istruzione **slt**:

```

1 slt $a2, $a2, $a3    # setta $a2 se i < j
2 beq $a2, $zero, L2   # se $a2 == 0 (non settato), vai a L2
3 add $v0, $a0, $a1
4 j L3                 # salta a L3
5 L2:
6 sub $v0, $a0, $a1
7 L3:

```

Implementazione x86

L'implementazione è quasi identica a quella della condizione di uguaglianza, cambia solamente l'istruzione di salto.

```

1  cmpq %rcx, %rdx  # confronta i e j e setta il flag
2  jge L2            # se i < j, salta a L2
3  leaq (%rdi, %rsi), %rax  # salva g + h in %rax
4  jmp L3
5 L2:
6  movq %rdi, %rax  # sposta g da %rdi a %rax
7  subq %rsi, %rax  # salva g - h in %rax
8 L3:

```

Anche l'implementazione con `cmove` è simile:

```

1  leaq (%rdi, %rsi), %rax  # salva g + h in %rax
2  subq %rsi, %rax          # salva g - h in %rax
3  cmpq %rcx, %rdx          # flag
4  cmove %rdi, %rax         # sposta %rdi in %rax se il i < j

```

Implementazione ARM

Ancora una volta, le indicazioni condizionali delle istruzioni ARM ci semplificano la vita:

```

1  cmp r2, r3      ; setta i flag
2  addlt r0, r0, r1 ; se i < j, salva g + h in r0
3  subge r0, r0, r1 ; altrimenti, salva g - h in r0

```

9.4 Cicli

Ora analizziamo l'implementazione di un ciclo con condizione di terminazione basata sui valori contenuti in un array di interi. Vedremo che l'implementazione usata sarà sempre quella del salto condizionale, nonostante ARM coi suoi suffissi potrebbe prestarsi ad implementazioni alternative. Il codice C è il seguente:

```

1 int i = 0;
2 while (a[i] == k){
3     i += 1;
4 }

```

Implementazione MIPS

In MIPS mappiamo `k` in `$a0`, l'indirizzo di `a` in `$a1` e `i` in `$v0`. Notiamo che per poter operare il confronto `a[i]==k` sarà necessario servirsi di un registro temporaneo `$t0` di appoggio dove caricare l'elemento del vettore `a`; inoltre, poiché le modalità

d'indirizzamento di MIPS sono molto poco potenti, sarà necessario operare lo shift logico a sinistra a ogni ciclo per mantenere la regolarità delle words.

```

1  start:
2      move $v0, $zero      # i = 0
3  L1:
4      sll $t1, $v0, 2      # salva 4 * i in $t1
5      add $t1, $t1, $a1    # ottieni l'indirizzo di a[i]
6      lw $t0, 0($t1)       # carica in $t0 il contenuto di a[i]
7      bne $t0, $a0, L2     # salta a l2 se a[i] è diverso da k
8      addi $v0, $v0, 1
9      j L1
10 L2:
```

L'implementazione di gcc è leggermente più complessa e aggira la scarsa potenza sull'indirizzamento di MIPS usando non uno, ma due registri come contatori: \$v0 contiene *i* e \$a1 contiene il contatore per l'elemento successivo dell'array; quest'ultimo registro inoltre viene sempre incrementato di 4, assumendo che ogni elemento dell'array, in quanto **int**, abbia dimensione 4 byte.

```

1  start:
2      lw $a2, 0($a1)      # carica in a2 il contenuto di a[0]
3      bne $a2, $a0, L2    # se a[0]==k, salta a L2
4      addi $a1, $a1, 4     # incrementa a di 4
5      move $v0, $zero      # inizializza il contatore
6  L1:
7      addi $a1, $a1, 4     # incrementa a di 4
8      lw $v1, -4($a1)      # carica in v1 il contenuto di a decrementato
9      ← di 4
10     addi $v0, $v0, 1      # incrementa il contatore
11     beq $v1, $a0, L1      # se il contenuto di a in questa posizione è
12     ← uguale k torna al ciclo
13     j L3                  # termina
14 L2:
15     move $v0, $zero      # inizializza il contatore
16 L3:
```

Implementazione x86

L'implementazione di Intel è più compatta. Mappiamo le variabili come segue: *i* in %rax, l'indirizzo di base *a* in %rsi e *k* in %edi; notiamo che possiamo mappare *k* in un registro a 32 bit poiché andrà confrontato con un elemento di un array,

che abbiamo detto essere lungo 4 byte. Notiamo inoltre che la `cmpl` alla riga 7 permette di confrontare `k` direttamente con un elemento in memoria, senza bisogno di operazioni di load; inoltre la stessa istruzione ci permette di moltiplicare per 4 l'indice in `%rax`, senza bisogno di un ulteriore registro di appoggio.

```

1 start:
2     cmpl (%rsi), %edi    # confronta k e il primo elemento di a
3     jne L2                 # se non sono uguali, vai a L2
4     movq $0, %rax         # inizializza l'indice
5 L1:
6     addq $1, %rax          # incrementa i
7     cmpl %edi, (%rsi, %rax, 4) # confronta k con a[i]
8     je L1                  # se k==a[i], torna a L1
9     jmp L3                 # salto incondizionato a L3
10 L2:
11    movq $0, %rax
12 L3:
```

Implementazione ARM

Assumiamo che le variabili siano mappate in questo modo: `k` in `r0`, l'indirizzo di `a[0]` in `r1` e `i` in `r3`. Notiamo nella riga 8 l'utilizzo dell'indirizzamento *pre indexing*, che incrementa `r1` di 4 prima di accedervi e come siano utilizzati due registri, `r3` e `r2` per il valore del contatore e della locazione di memoria cui accedere.

Si noti che in `L2` viene settato a 0 il valore del contatore: questa operazione risulta necessaria in quanto se il flusso d'esecuzione arriva a `L2` sicuramente non si è entrati nel ciclo (e dunque non è avvenuta l'inizializzazione del puntatore).

```

1 start:
2     ldr r3, [r1]      ; carica a[0] in r3
3     cmp r0, r3        ; confronta a[0] con k
4     bne L2            ; se sono diversi, salta a L2
5     mov r3, #0         ; ora r3 contatore: inizializza i=0
6 L1:
7     add r3, r3, #1    ; incrementa i
8     ldr r2 [r1, #4]!  ; carica a[i] in r2
9     cmp r2, r0        ; confronta a[i] con k
10    beq L1           ; se sono uguali torna al ciclo
11    b L3              ; altrimenti salta a L3
12 L2:
13    mov r3, #0         ; setta il contatore a 0
```

```

14 L3:
15     mov r0, r3

```

9.5 Invocazione di subroutine

I prossimi esempi serviranno ad esaminare le ABI e le convenzioni di chiamata delle tre architetture che stiamo analizzando.

Funzioni foglia

Iniziamo con una funzione foglia, ossia una funzione che non chiama alcuna subroutine, che conseguentemente non rende necessario salvare il **return address register**. Di seguito il codice C.

```

1 int esempio_foglia(int g, int h, int i, int j){
2     int f;
3     f = (g + h) - (i + j);
4     return f;
5 }

```

Implementazione MIPS

Mappatura: *g, h, i j* in *\$a0, \$a1, \$a2, \$a3*, e il valore di ritorno va salvato in *\$v0*. Nulla di particolare da segnalare, se non il ritorno implementato con un salto incondizionato all'indirizzo del **return address register**.

```

1 esempio_foglia:
2     addu $a0, $a0, $a1    # somma g + h e salva in a0
3     addu $v0, $a2, $a3    # somma i + j e salva in v0
4     subu $v0, $a0, $v0    # sottrae a0 a v0
5     jr $ra                # salto all'indirizzo di ritorno

```

Implementazione x86

L'implementazione Intel è immediata, se si tiene conto del fatto che le variabili verranno mappate nei "registri e" (%edi, %esi, %ecx, %edx) a 32 bit, anziché nei "registri r" (%rdi, %rsi, %rcx, %rdx) offerti da x86-64; questo succede perché i valori che dobbiamo modificare sono di tipo **int**, quindi di dimensione appunto 32 bit. Notiamo inoltre l'istruzione **ret** preposta a restituire il ritorno della subroutine.

```

1 esempio_foglia:
2     leal (%rdi, %rsi), %eax    # salva in eax g + h
3     addl %ecx, %edx           # salva in edx i + j
4     subl %edx, %eax          # sottrae le due somme
5     ret

```

Implementazione ARM

Anche l'implementazione ARM è molto semplice. Notiamo l'utilizzo di `rsb`, che sottrae l'operando di destra a quello centrale, e l'uso di `bx` (una macro per `move r15, r`) per saltare all'indirizzo contenuto nel link register `lr`.

```

1 esempio_foglia:
2     add r0, r0, r1 ; salva in r0 la somma g + h
3     add r3, r2, r3 ; salva in r3 la somma i + j
4     rsb r0, r3, r0 ; salva in r0 il risultato di r0 - r3
5     bx lr

```

Funzioni non foglia

Com'era lecito aspettarsi, l'implementazione di funzioni non foglia (che invocano altre funzioni) è più complessa. Analizziamo un esempio semplice:

```

1 int inc(int n){
2     return n + 1;
3 }
4
5 int f(int x){
6     return inc(x) - 4;
7 }

```

Implementazione MIPS

La conversione di `inc` è banale, `f` invece ha notevoli complicazioni; è necessario infatti salvare il contenuto di `$ra` prima di chiamare `inc`, e possiamo farlo decrementando di una word (4 byte) lo stack pointer per così salvare `$ra` in `$sp` nello spazio appena creato, per poi recuperarlo (e reincrementare `$sp`) prima del ritorno.

```

1 inc:
2     addiu $v0, $a0, 1    # incrementa
3     jr $ra
4

```

```

5   f:
6       addiu $sp, $sp, -4    # decrementa di 4 byte sp
7       sw $ra, 0($sp)        # salva ra all'indirizzo di sp
8       jal inc               # chiama inc
9       addiu $v0, $v0, -4    # sottrae 4 a v0
10      lw $ra, 0($sp)        # recupera ra
11      addiu $sp, $sp, 4     # ripristina lo stack pointer
12      jr $ra               # ritorna

```

L'implementazione di `gcc` è ancora diversa, perché l'ABI da noi usata prevede che il record di attivazione sia un multiplo di 8, per cui `$sp` verrà decrementato di 8 bytes; altre ABI avrebbero lavorato ancora diversamente, dal momento che la gestione dello stack è appunto demandata alla *Application Binary Interface* utilizzata.

```

1 inc:
2     addiu $v0, $a0, 1
3     jr $ra
4
5 f:
6     addiu $sp, $sp, -8    # sp viene decrementato di 8
7     sw $ra, 4($sp)        # ra viene salvato in sp con offset di 4
8     jal inc
9     addiu $v0, $v0, -4
10    lw $ra, 4($sp)        # recupero ra
11    addiu $sp, $sp, 8     # ripristino sp
12    jr $ra

```

Implementazione x86

Fortunatamente per noi, le funzioni `call` e `rec` di x86 gestiscono magistralmente il movimento dello stack pointer e non richiedono alcun intervento extra da parte nostra; inoltre, la sempre fresca `lea` ci regala la libertà di cui abbiamo bisogno per salvare i risultati di una somma in un operando terzo.

```

1 inc:
2     leal 1(%rdi), %eax  # salva rdi + 1 in eax
3     ret
4
5 f:
6     call inc              # chiama inc
7     subl $4, %eax         # sottrae 4 a eax
8     ret

```

Implementazione ARM

ARM si rivela molto interessante: le sue possibilità di indirizzamento *pre indexing* ci consentono di effettuare il decremento dello stack pointer e il salvataggio del link register in una sola riga di codice; inoltre, dal momento che in ARM pc è un registro referenziabile come fosse un general purpose, possiamo direttamente prelevare il link register dallo stack e caricarlo direttamente nel Program Counter (PC) e, sempre nella stessa riga, reincrementare **sp** di 4 grazie all'indirizzamento *post indexing*.

```

1 inc:
2     add r0, r0, #1
3     bx lr
4
5 f:
6     str lr, [sp, #-4]! ; decrementa SP di 4 e salva lr
7     bl inc             ; chiama inc
8     sub r0, r0 #4       ; sottrae 4 a r0
9     ldr pc, [sp], #4    ; carica lr in PC e recupera SP

```

9.6 Copia stringa

Infine consideriamo una funzione che copia un array di caratteri:

```

1 void copia_stringa(char *d , const char *s){
2     int i = 0;
3     while((d[i] = s[i]) != 0){
4         i += 1;
5     }
6 }

```

Implementazione MIPS

Abbiamo già discusso l'implementazione MIPS di questa funzione in 6.8, per cui riporteremo solamente il codice, generato da gcc:

```

1 copia_stringa:
2     lb $v0, 0($a1)      # carica in v0 il contenuto di s[0]
3     sb $v0, 0($a0)      # salva v0 in d[0]
4     beq $v0, $zero, L5  # se v0 == 0, salta a L5
5     move $v0, $zero      # inizializza v0 a 0
6 L3:

```

```

7      addiu $v0, $v0, 1    # incrementa il contatore v0
8      addu $v1, $a1, $v0  # salva s[0 + contatore] in v1
9      lb $v1, 0($v1)     # carica *(s + contatore) in v1
10     addu $a2, $a0, $v0  # salva d[0 + contatore] in a2
11     sb $v1, 0($a2)     # ripone v1 in *(d + contatore)
12     bne $v1, $zero, L3 # se v1 non è il carattere nullo torna al
                           ↵  ciclo
13 L5:
14     jr $ra               # termina e ritorna

```

Implementazione x86

```

1 copia_stringa:
2     movzbl (%rsi), %eax
3     movb %al, (%rdi)
4     testb %al, %al
5     je L1                      # se uguali, ritorna e termina
6     movl $0, %eax              # inizializza a 0 eax
7 L3:
8     addl $1, %eax              # incrementa il contatore
9     movslq %eax, %rcx
10    movzbl (%rsi, %rcx), %edx
11    movb %dl, (%rdi, %rcx)
12    testb %dl, %dl
13    jne L3
14 L1:
15     ret

```

Implementazione ARM

Ecco invece il codice ARM; notiamo come i suffissi condizionali e soprattutto il *pre indexing* vengono sfruttati per snellire un codice in realtà molto simile a quello di MIPS.

```

1 copia_stringa:
2     ldrb r3, [r1]          ; carica s[0] in r3
3     strb r3, [r0]          ; salva r3 in d[0]
4     cmp r3, #0             ; r3 è 0?
5     bxeq lr                ; se sì, ritorna e termina
6 L3:
7     ldrb r3, [r1, #1]!    ; carica in r3 il contenuto di s[]
                           ↵  incrementato di 1 e salva le modifiche

```

```
8      strb r3, [r0, #1]! ; salva r3 in d[] incrementato di 1 e salva
→    le modifiche
9      cmp r3, #0           ; la stringa è terminata?
10     bne L3              ; se no, torna al ciclo
11     bx lr               ; altrimenti ritorna e termina
```

Capitolo 10

Toolchain

10.1 Introduzione

La *toolchain* è la catena di strumenti che permette di tradurre il codice di alto livello in linguaggio macchina: descriveremo in questo capitolo quali sono le sue componenti e come funzionano.

Come ben sappiamo, non è possibile per il calcolatore comprendere né i linguaggi di alto livello né il linguaggio Assembly: si utilizzano degli strumenti come compilatori ed assemblatori per tradurre i programmi in modo da renderli comprensibili alla macchina; è giunto finalmente il momento di scoprire come avviene questa traduzione. Partendo con un esempio di traduzione dal C, osserviamo le componenti che entrano in gioco nel processo.

1. *Preprocessore*: gestisce le direttive # (`#include`, `#define` etc.) generalmente sostituendo pezzi di codice ed eliminando i commenti;
2. *Compilatore*: traduce il codice da C ad assembly (da file .c a file .s);
3. *Assembler*: traduce da assembly in linguaggio macchina, da file .s a file .o;
4. *Linker*: collega tra loro diversi file .o per linkare al file oggetto del programma le librerie o gli altri file .o di cui ha bisogno, una volta terminato produce un file eseguibile.

Quando si è ottenuto il file eseguibile, questo può essere caricato in memoria con una system call ed essere eseguito. Si noti che normalmente un driver gestisce tutto questo processo in maniera automatica.

La toolchain GCC

Uno degli strumenti più diffusi per compiere queste operazioni è la toolchain Gnu Compiler Collection (GCC), che può compilare diversi linguaggi e generare linguaggio macchina per varie architetture CPU, MIPS compreso.

Il nostro `gcc` lavora in svariate fasi, operate da diversi componenti: un compilatore per il C++ (`cpp`), un compilatore per il C (`cc`), un assembler (`as`) e un linker (`ld`). Quando si utilizza `gcc` si può invocare la compilazione su un file con diversi parametri:

- `gcc -c <file>` si ferma dopo aver invocato `as` (genera quindi un file oggetto `.o`);
- `gcc -S <file>` si ferma dopo aver invocato `cc` (genera quindi un file assembly `.s`);
- `gcc <file>` invoca tutti i programmi necessari.

Vediamo ora nello specifico le fasi che caratterizzano tutto il processo svolto da `gcc`.

10.2 Processo: dal codice sorgente al file oggetto

Per comprendere bene il funzionamento di questi processi verrà ora illustrato il procedimento che porta da un file `.c` a un eseguibile.

Primo passo Dato un file `.c`, è invocare su di esso l'istruzione `gcc -S`, chiamando così il `cc` (compilatore), che traduce il file da linguaggio C ad Assembly generando un file `.s` del tipo esatto (poiché `cc` riconosce l'architettura per cui sta lavorando).

Da notare che spesso il codice Assembly generato dal compilatore è migliore di quello che potrebbe scrivere un programmatore, dato che è pensato per effettuare una serie di semplificazioni e ottimizzazioni al codice mentre lo traduce, così da renderlo più leggero e più performante (dispiegando loop, accorpando variabili, diminuendo l'uso di variabili temporanee...); i livelli di ottimizzazione si possono settare tramite l'opzione (facoltativa) `-O`, che permette di indicare uno tra i 3 livelli di semplificazione disponibili.

Secondo passo Si utilizza il comando `gcc -c` sul file (può essere invocato sia sul file `.c`, rendendo implicito lo svolgimento del primo passo, che sul file `.s` ottenuto dallo svolgimento primo passo). Questo significa che viene invocato l'assemblatore `as` per generare un file `.o`, ovvero un file oggetto.

Nello specifico `as` fa molto di più che tradurre da assembly in linguaggio macchina:

- converte le pseudo-istruzioni in sequenze di istruzioni standard;
- converte le istruzioni Assembly in linguaggio macchina;
- converte i numeri da decimale/esadecimale in binario;
- gestisce le label traducendole in indirizzi veri e propri;
- gestisce i salti: se l'indirizzo è troppo lontano e non sta nello spazio dedicato all'operazione `j`, questa va convertita in caricamento della destinazione in un registro ed in seguito in una `jr` al registro stesso;
- genera i metadati, informazioni di alto livello che serviranno in seguito al loader per caricare il codice binario.

Alla fine delle operazioni dell'`as` si ottiene quindi un file oggetto, composto da varie sezioni:

- *header*: contiene le posizioni di altri dati all'interno del file (segmenti, tabelle, etc.);
- *segmenti*: principalmente testo (codice) e dati (variabili globali);
- *tabella di rilocazione*: comprende tutti i simboli¹ con indirizzo relativo all'interno dell'area `.text` (ovvero l'indirizzo delle operazioni in cui compaiono tali simboli) e tipo di istruzione;
- *tabella dei simboli*: contiene tutti i simboli che non sono contenuti all'interno del file oggetto stesso (segnalati come `undefined`) ma saranno dedotti da file esterni e quindi saranno noti solo una volta che il linker avrà svolto il suo compito di collegamento tra file; contiene anche i simboli definiti nel file stesso ed il loro indirizzo globale (sempre relativo al file);
- altre informazioni (ad esempio per il debugging).

¹I simboli sono stringhe come etichette, nomi di funzioni e variabili che implicitamente rappresentano indirizzi.

10.3 Processo: dal file oggetto all'eseguibile

A questo punto non rimane altro che prendere il file oggetto e trasformarlo in eseguibile: questo significa invocare il comando `gcc` senza modificatori, ovvero compiendo anche l'ultimo passo, quello del linking. Il linker `ld` mette assieme uno o più file oggetto eseguendo le necessarie rilocazioni. Generalmente svolge le seguenti operazioni:

- decide come codice e dati sono disposti in memoria compattando tra loro le sezioni con funzioni uguali;
- associa indirizzi assoluti a tutti i simboli, risolvendo anche quelli contenuti nella tabella dei simboli, che quindi non sono locali e che deve andare a cercare nei file `.o` linkati;
- patcha le istruzioni di salto risistemando gli indirizzi, che sono stati modificati dopo lo svolgimento del primo punto di questo elenco.

Quindi, lo scopo principale di `ld` è quello di eliminare le tabelle dei simboli e di rilocazione, sostituendo tutti gli indirizzi relativi e indefiniti con indirizzi assoluti. Per fare ciò, il linker ha bisogno di collegare tra loro più file `.o`, dato che alcuni oggetti definiti in un file potrebbero essere richiamati all'interno di un altro file.

10.3.1 Linker e simboli

I tipi di simboli con cui il linker si trova ad avere a che fare sono tre:

- i simboli *locali* (*non esportati*) sono simboli definiti e visibili solamente all'interno del file;
- i simboli *definiti* (*defined*) sono i simboli associati ad un indirizzo relativo nella tabella dei simboli, tuttavia definiti all'interno del file stesso;
- i simboli *non definiti* (*undefined*) sono quei simboli presenti nella tabella dei simboli ma dichiarati in un file diverso, quidi presenti appunto solo come `undefined`.²

In tutti questi casi, il lavoro del linker rimane lo stesso, ovvero quello di associare ad ogni simbolo un indirizzo assoluto, perché solo in questo modo essi sono interpretabili dal calcolatore.

²Nel caso un simbolo non locale non venga trovato nei file `.o` linkati viene generato un errore di linking.

10.4 Linking visto approfonditamente

Il linking può essere spiegato più precisamente suddividendolo nei tre passi qui elencati.

1. Disporre in memoria i vari segmenti riordinandoli, ovvero prendendo le parti `.text` di tutti i file `.o` linkati ed unendole, facendo la stessa cosa con tutte le sezioni, così da ottenere un unico file `.o` che contiene tutto il codice e che sia strutturalmente ordinato.
2. Assegnare un indirizzo assoluto ad ogni simbolo presente nelle tabelle dei simboli, tenendo conto del passaggio precedente (che modifica necessariamente gli indirizzi relativi dei singoli file `.o`).
3. Infine modificare tutte le istruzioni con gli indirizzi appena calcolati, sistemandone definitivamente anche tutti i simboli della tabella di rilocazione.

Il file risultante viene finalmente "incapsulato" in un file eseguibile, che quindi si ritroverà a contenere i vari segmenti (testo, dati, etc.), le informazioni per il caricamento in memoria (indirizzo di caricamento dei segmenti, entry point del programma, etc.) ed altre informazioni aggiuntive (ad esempio per il debugging).

10.5 Librerie

Data la complessità dei programmi al giorno d'oggi, si è resa indispensabile la creazione di librerie, ovvero collezioni di file `.o` contenenti le funzioni standard che si rivelano utili praticamente in ogni programma. Questo è un grande vantaggio: invece di linkare un'enormità di file oggetto basta linkare una singola libreria per avere tutte le funzioni standard. Questo però comporta lo svantaggio di portarsi dietro un file relativamente pesante (la libreria standard del C pesa 2,5 Mb) quando magari si necessita solamente di poche funzioni fra tutte quelle contenute nella libreria. Anche in questo caso esiste una soluzione più ottimizzata: le *librerie dinamiche*. A seguire vengono presentate le principali differenze fra librerie statiche e dinamiche.

- *Librerie statiche* (`.a`): sono semplici collezioni di file `.o`; il linker `ld` compie tutto il lavoro di linkaggio sulla libreria intera, il che è svantaggioso su calcolatori dove la memoria è limitata (vedi sistemi embedded). Tuttavia la fase di esecuzione risulta molto semplice in quanto tutte le funzioni che serviranno nel programma sono già contenute nell'eseguibile;
- *Librerie dinamiche* (`.so`): il vero linking avviene a tempo di caricamento ed esecuzione, ovvero la libreria viene consultata solo a runtime, rendendo

così molto più leggero il file eseguibile del programma. Pur essendo molto utile quando la memoria è limitata, tuttavia l'implementazione è più difficile rispetto alla prassi con librerie statiche.

10.5.1 Funzionamento di una libreria dinamica

Nel momento del linking `ld` inserisce all'interno dell'eseguibile dei riferimenti alle librerie ed alle funzioni usate, ma non inserisce le librerie stesse, mantenendo così l'eseguibile leggiadro; inoltre `ld` inserisce all'interno dell'eseguibile un riferimento ad un linker dinamico (esempio: `/lib/ld-linux.so.2`) che viene caricato ed eseguito al momento dell'esecuzione passandogli come argomento l'intero programma. Sarà proprio questo linker dinamico a caricare ed eseguire le librerie dinamiche ed eseguire solo i linking necessari.

Come conseguenza dell'implementazione di questa strategia si ha che l'eseguibile risulta di per sé molto più leggero ed inoltre si ottiene una modularizzazione del programma stesso: se le librerie vengono aggiornate (rimanendo retrocompatibili), il programma ne utilizzerà in automatico la versione più aggiornata. Lo svantaggio in cui si incorre però è una complicazione notevole del processo di caricamento.

Esiste tuttavia una soluzione ancora più ottimale all'interno della categoria delle librerie dinamiche: questa è il processo di *lazy linking*, che consiste, in una maniera squisitamente informatica, nel rimandare il più possibile le operazioni complesse di linking, così da effettuarle solo se estremamente necessarie. In questo modo, se una libreria non viene mai effettivamente invocata, si può addirittura evitare di linkarla. La strategia utilizzata è quella di inserire al posto dell'indirizzo effettivo della funzione l'indirizzo di uno *stub* che solo se verrà invocato andrà a ricercare la funzione e a compiere il linking con la libreria in modo da poter utilizzare la funzione. Tutto questo lavoro viene svolto se e solo se la funzione è effettivamente utilizzata e quindi comporta potenzialmente un risparmio di spazio per il linking della libreria che la contiene. Inoltre, una volta che una funzione è stata effettivamente linkata, essa è già disponibile per tutto il resto del programma, così da non dover nemmeno ripetere più volte il lazy linking.

10.6 Esempio finale

Procediamo ora illustrando un esempio di linking composto da due file assembly:

File 1:

```
.comm x,4,4
...
.text
.globl f
f:
lw $a0, x($gp)
jal g
...
```

File 2:

```
.comm y,4,4
...
.text
.globl g
g:
sw $a1, y($gp)
jal f
...
```

Vediamo ora come vengono tradotti questi due file dall'as che ne ricava due file oggetto.

Sezione	Contenuti		
metadati	<i>campo</i>	<i>valore</i>	
	nome	file1	
	text size	100 ₁₆	
	data size	20 ₁₆	
.text	<i>indirizzo (rel.)</i>	<i>istruzione</i>	
	0	lw \$a0, 0(\$gp)	
	4	jal 0	
	8	...	
	
.data	<i>indirizzo (rel.)</i>	<i>simbolo</i>	
	0	x	
	
tabella simboli	<i>simbolo</i>	<i>indirizzo</i>	
	x	...	
	g	...	
	
tabella rilocazione	<i>indirizzo</i>	<i>tipo istruzione</i>	<i>simbolo</i>
	0	lw	x
	4	jal	g

Tabella 10.1: File oggetto 1

Sezione	Contenuti		
metadati	<i>campo</i>	<i>valore</i>	
	nome	file2	
	text size	200_{16}	
	data size	30_{16}	
.text	<i>indirizzo (rel.)</i>	<i>istruzione</i>	
	0	<code>sw \$a1, 0(\$gp)</code>	
	4	<code>jal 0</code>	
	8	...	
	
.data	<i>indirizzo (rel.)</i>	<i>simbolo</i>	
	0	y	
	
tabella simboli	<i>simbolo</i>	<i>indirizzo</i>	
	y	...	
	f	...	
	
tabella rilocazione	<i>indirizzo</i>	<i>tipo istruzione</i>	<i>simbolo</i>
	0	sw	y
	4	jal	f

Tabella 10.2: File oggetto 2

Ora è il turno della componente `ld`, che si occupa di unire i due file. Ricordiamo che per prima cosa il linker deve definire la struttura del file di output, e per fare questo sfrutta le intestazioni (i metadati) dei file che riceve in input; nello specifico è interessato a leggere le dimensioni delle varie sezioni per poter decidere quanta memoria dedicarvi.

Si deve sapere che in base all'architettura su cui si lavora esistono degli indirizzi di partenza prestabiliti per ogni sezione: nel nostro caso (architettura MIPS) la sezione `.text` inizia all'indirizzo $0x00400000$; l'`ld` così inserirà il `text` del primo file da $0x00400000$ a $0x400100$ e quello del secondo da $0x00400100$ a $0x00400300$. Stessa cosa con la sezione `.data` (che in *MIPS* inizia $0x10000000$), quella del primo file si troverà nel file di output finale tra $0x10000000$ e $0x10000020$ e quella del secondo tra $0x10000020$ e $0x10000050$.

In secondo luogo il linker assegna un indirizzo assoluto ai simboli, nel nostro esempio $f = 0x00400000$ e $g = 0x00400100$, mentre $x = 0x10000000$ e $y =$

0x10000020.

Infine `ld` patcha tutti gli indirizzi dei simboli tenendo conto che il `$gp` in MIPS punta alla posizione `0x10008000` (quindi, ad esempio, l'indirizzo di `x` diventa `0x8000` rispetto al `$gp`). Il linker in questa fase risolve anche gli indirizzi legati alle operazioni di salto incondizionato; si può vedere facilmente questa operazione nell'esempio sottostante.

Ecco finalmente il file che viene dato come output da `ld`:

Sezione	Contenuti	
Header	<i>campo</i>	<i>valore</i>
	text size	300_{16}
	data size	50_{16}

.text	<i>indirizzo (rel.)</i>	<i>istruzione</i>
	0x00400000	<code>lw \$a0, 0x8000(\$gp)</code>
	0x00400004	<code>jal 0x00400100</code>

	0x00400100	<code>sw \$a1, 0x7FE0(\$gp)</code>
	0x00400104	<code>jal 0x00400000</code>

.data	<i>indirizzo (rel.)</i>	<i>dato</i>
	0x10000000	x

	0x10000020	y

Tabella 10.3: File di output

Capitolo 11

Central Processing Unit

“ Computers don’t create computer animation any more than a pencil creates pencil animation. What creates computer animation is the artist. ”

John Lasseter

11.1 Introduzione

In questo capitolo andremo ad analizzare come il nostro processore elabora ed esegue le istruzioni che gli arrivano dai programmi Assembly; in particolare ci baseremo ancora su MIPS, in quanto super regolare e particolarmente utile a scopo didattico. Per come è stato progettato MIPS le istruzioni presentano dei tratti comuni, in particolare le prime due fasi del ciclo della CPU (vedi 5.2.3): il prelievo dell’istruzione e la lettura dei valori dei registri operandi sono comuni a ogni istruzione.

Noi in particolare, dopo aver spiegato i vari elementi che concorrono a far funzionare tutta la baracca, andremo ad analizzare le istruzioni aritmetico-logiche, quelle di accesso alla memoria e quelle di salto, dal momento che tutte le altre si implementano con tecniche simili.

11.2 Arithmetic-Logic Unit

A eccezione di [j](#), tutte le istruzioni MIPS fanno uso di Arithmetic-Logic Unit (ALU), una rete logica combinatoria abbastanza complessa preposta all’esecuzione

di tutti calcoli di cui abbiamo bisogno; sostanzialmente, è un'unione di blocchi fondamentali che fanno operazioni su singoli bit.

Al di là delle operazioni puramente aritmetiche (per le quali lo scopo di ALU è ovvio), viene usata nelle operazioni di accesso alla memoria (**sw** e **lw**) per il calcolo degli indirizzi e dalle operazioni di salto condizionato per effettuare i confronti; successivamente al passaggio per ALU, ognuna di queste tre categorie d'istruzione differisce.

11.3 Il datapath

Il *datapath* è quella parte del processore attraverso cui passano le istruzioni; è letteralmente un percorso da seguire ogniqualvolta si debba eseguire una qualsiasi istruzione.

11.3.1 Struttura del datapath

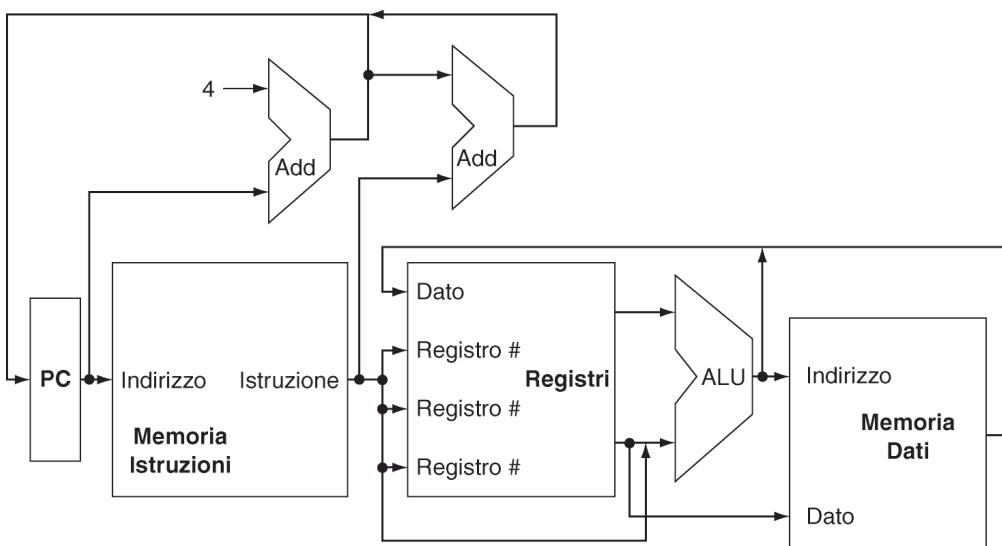


Figura 11.1: Schema base del datapath

Il datapath si compone essenzialmente di 5 fasi.

1. Assumendo che il programma sia già stato caricato in memoria (secondo le modalità viste nel capitolo 10), in questa fase viene prelevata l'istruzione da eseguire e viene spacchettata nei vari campi;

2. I registri interessati all'esecuzione vengono caricati nel *banco dei registri*; se l'istruzione è di tipo R, gli indirizzi dei registri sono contenuti nel corpo dell'istruzione;
3. A questo punto la ALU esegue i calcoli opportuni, ottenendo il valore (o l'indirizzo) desiderato;
4. I risultati ALU vengono utilizzati per salvare (o prelevare) in memoria dati un dato, o per memorizzare un registro;
5. Si incrementa il PC per mezzo di una ALU dedicata (di fatto, un addizionatore), e una successiva ALU calcola l'indirizzo verso cui ci si deve spostare in caso di salti incondizionati.

11.3.2 Il ruolo del multiplexer

La figura però è incompleta; manca qualcosa che dica ai blocchi cosa fare in caso di punti di decisione (momenti in cui i segnali arrivano da due diverse sorgenti e bisogna sceglierne una), come ad esempio l'incremento del program counter: normalmente, il suo valore proviene dall'addizionatore (e punta quindi alla word successiva a quella appena letta), ma in caso di salto l'indirizzo viene calcolato dallo spiazzamento contenuto nell'istruzione¹. Altro esempio: a seconda della tipologia di istruzione il secondo operando della ALU potrà provenire o dal banco dei registri (per le istruzioni R) o dal codice dell'istruzione stessa (per le istruzioni I).

Ed è qui che ci viene in soccorso il *multiplexer* che, come già detto in 4.4, è un circuito combinatorio che prende in input due segnali e decide quale di essi debba andare in output sulla base di un terzo segnale di controllo (un po' come un vigile che decide quale macchina far passare).

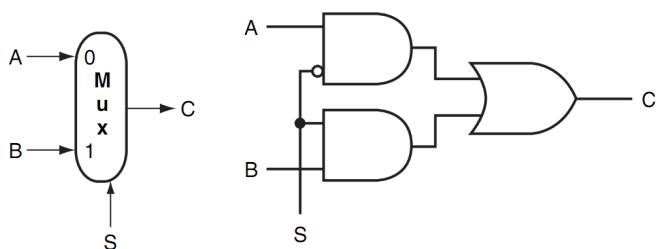


Figura 11.2: Schema di un multiplexer

¹Si ricordi che MIPS offre solo 16 bit per immagazzinare costanti all'interno di istruzioni immediate. Dunque, attraverso **beq** risulta necessario utilizzare riferimenti relativi (al PC + 4) e non assoluti per poter accedere ad una vasta gamma di indirizzi.

Altri elementi Oltre ai multiplexer, ci sono altri elementi che concorrono alla soluzione dei punti di decisione. Ad esempio, alla ALU viene notificato un segnale di controllo (nel nostro caso di 4 bit) per decidere quale operazione effettuare, il banco dei registri riceve dei flag per decidere se scrivere o meno un registro, la memoria dati ha degli espedienti per decidere se effettuare lettura o scrittura, e simili.

Chi stabilisce queste cose, oltre a settare il segnale del selettore dei vari multiplexer? È presto detto, l'unità di controllo!

11.4 La Control Unit

La *Control Unit* funge da vero e proprio "direttore d'orchestra" per il processore, stabilendo il valore S dei vari multiplexer e andando a sciogliere le questioni dei punti di decisione.

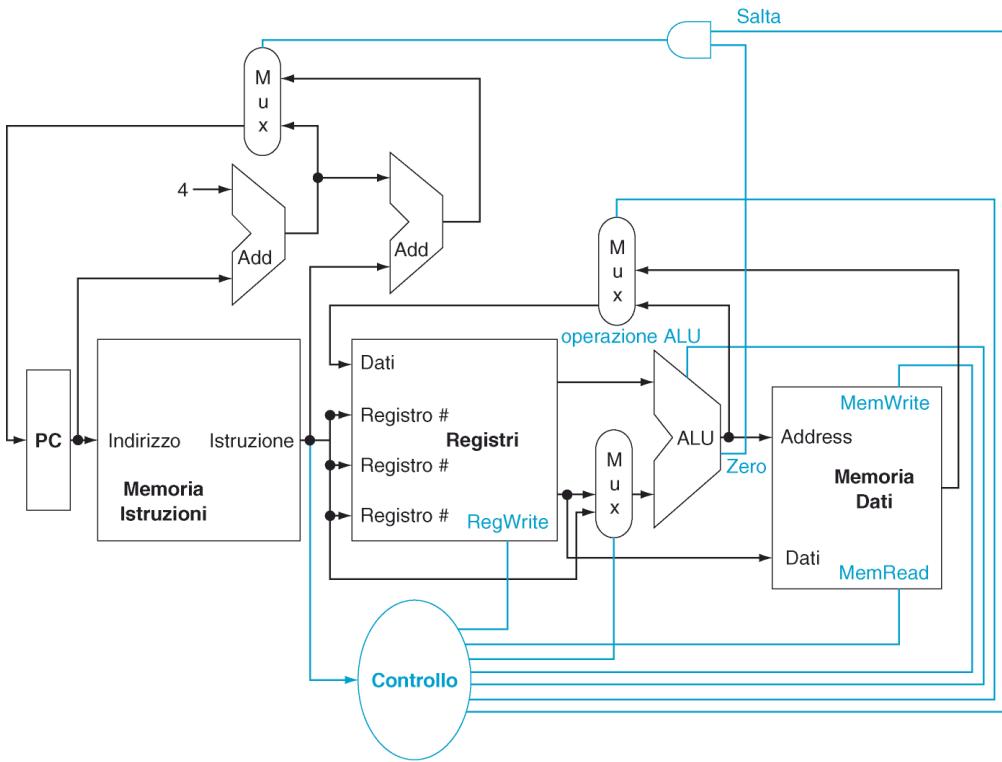


Figura 11.3: Schema di datapath e control unit

La figura è molto complessa, andiamo ad analizzarla con ordine.

- Innanzitutto notiamo il multiplexer prima del PC: questo gestisce la scelta su come viene aggiornato il PC, se secondo il valore del primo o del secondo

addizionatore. Per settare il valore S viene utilizzata una porta AND, il cui primo input arriva direttamente dalla Control Unit, mentre il secondo viene fornito dalla ALU (*zero*).

- Vediamo un secondo multiplexer che indica se memorizzare nel registro di destinazione un dato da ALU o dalla memoria.
- Abbiamo un ultimo multiplexer che precede il caricamento del secondo operando nell'ALU: dalla control unit arriverà l'indicazione per scegliere di caricare un registro (nelle istruzioni R) o una costante (per quelle I).
- Poi vediamo un segnale per la ALU: esso consiste di un bus di più bit per comunicare all'unità quale operazione eseguire.
- Alla memoria dati invece arrivano i segnali *MemWrite* e *MemRead*, che indicano la rispettiva operazione da svolgere.
- In base al valore di *RegWrite*, il risultato dell'operazione viene memorizzato nel registro di destinazione.

Questa è la rappresentazione di MIPS, che è una ISA RISC; inutile dire che gli schemi di funzionamento che stanno alla base di architetture CISC sono incredibilmente più complessi.

11.5 La temporizzazione

A questo punto abbiamo moltissimi segnali che viaggiano nel processore, sincronizzati dal clock. Per semplicità assumiamo che ciascuna istruzione venga eseguita durante un ipotetico singolo ciclo di clock, lungo abbastanza.

11.5.1 Breve riepilogo sulle reti logiche

Circuiti come i multiplexer sono detti *reti combinatorie* e producono un output secondo una funzione statica dell'input, senza memoria dei cambiamenti.

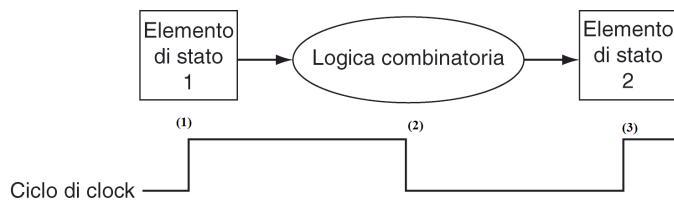
Risulta però chiaro che elementi come i registri e le memorie (detti elementi "*di stato*") necessitano di tenere traccia dei cambiamenti subiti. In questo caso giungono in nostro soccorso le *reti sequenziali*, dove l'output dipende dalla storia (sequenza) degli input precedenti. Si ha inoltre che gli elementi di stato hanno almeno due ingressi:

- il valore da immettere nello stato;
- il clock atto a sincronizzare le transizioni.

Flip-flop Il *flip-flop D-latch* è l'elemento base per memorizzare un bit, e registri sono di fatto vettori di 32 di questi circuiti (rivedi 4.6 per maggiori dettagli).

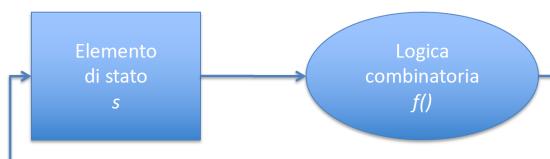
11.5.2 Gestione della temporizzazione

Avere un clock regolare ci permette di ottenere indicazioni precise sulla sequenza di operazioni svolte e avere sempre ragione di quale istruzione è avvenuta prima o dopo di un determinato colpo di clock. La metodologia di temporizzazione ci dà quindi indicazioni precise rispetto alla possibilità di lettura/scrittura dei segnali rispetto al clock, e i dati presi dagli elementi di stato sono sempre relativi ai cicli precedenti. Vediamo uno schema di come viene gestita la cosa:



1. l'elemento di stato 1 viene aggiornato a un determinato tempo t ;
2. il valore passa attraverso a una qualsiasi rete combinatoria;
3. il valore arriva all'elemento di stato 2 al tempo $t + T$.

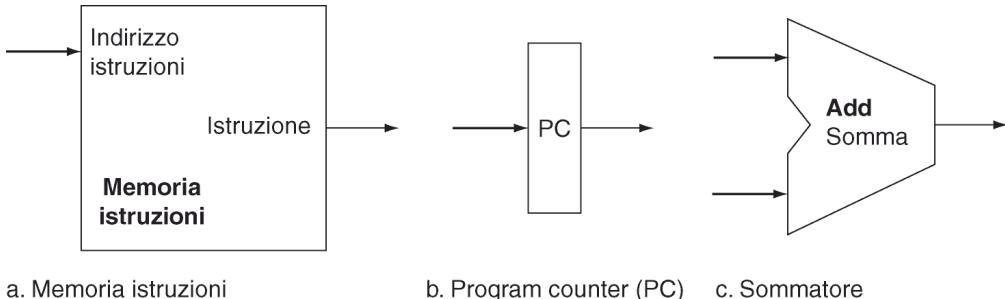
Il tempo T di clock deve naturalmente essere tale da dare tempo ai valori di attraversare i vari circuiti combinatori; inoltre il periodo del clock ci permette di regolarizzare quelli che altrimenti diventerebbero dei cicli di retroazione assolutamente non predibili.



Se non avessimo una metodologia di temporizzazione precisa avremmo che $s = f(s)$, che è un risultato assolutamente non prevedibile. Grazie alla temporizzazione sensibile al clock possiamo ottenere un andamento del tipo: $s(t + T) = f(s(t))$, decisamente più regolare ed efficiente per noi poiché deterministico e non più afflitto da incertezza.

11.6 Elaborazione delle istruzioni

Prima di addentrarci nel dettaglio dell'esecuzione di alcuni tipi di istruzioni, passiamo in rassegna le varie componenti che servono alla realizzazione di un datapath:



Ricordiamo che il sommatore (o addizionatore) è una ALU specializzata alla sola operazione di incremento del PC.

A questo punto la fase di fetch, ricordiamolo, prevede il prelievo dell'istruzione dal PC, il trasferimento della stessa agli elementi preposti all'esecuzione e l'incremento del PC.

11.6.1 Istruzioni R

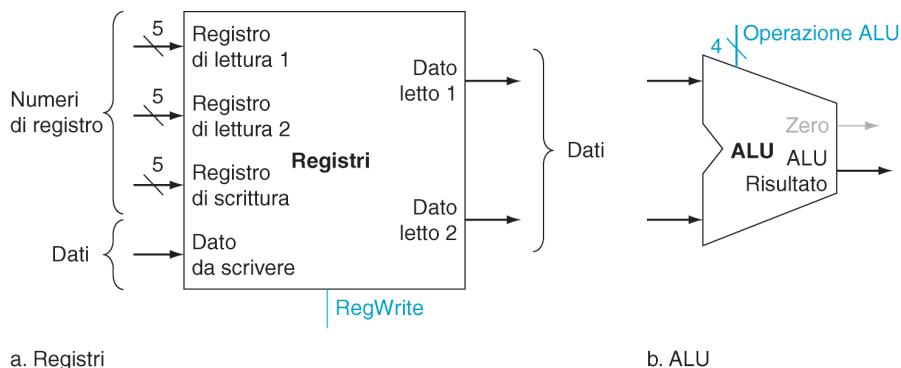
Iniziamo con un'istruzione di tipo R, come potrebbe essere:

¹ `add $t0, $s1, $s2`

la quale, come sappiamo, ha questo tipo di mapping:



Per effettuare questa operazione abbiamo bisogno di due blocchi funzionali che vanno ad aggiungersi ai tre di base che abbiamo visto prima.



- Il primo blocco è il *banco dei registri*, una struttura con 3 bus da 5 bit ciascuno in cui vengono caricati i registri da operare; per decidere se si dovrà scrivere il risultato nel registro di destinazione viene impiegato il segnale di controllo *RegWrite*, gestito direttamente dalla Control Unit.
- Il secondo blocco di cui abbiamo bisogno è la ALU di cui prima abbiamo tanto parlato. Questa presenta due controlli: uno di questi è il bus che arriva dal corpo dell'istruzione e indica qual è l'operazione da svolgere, mentre il secondo è il settaggio *zero* qualora il risultato dell'operazione fosse effettivamente 0; quest'ultimo valore viene poi mandato alla porta AND che decide il segnale di controllo per il multiplexer preposto alla scelta del blocco d'incremento del PC.

11.6.2 Istruzioni di accesso alla memoria

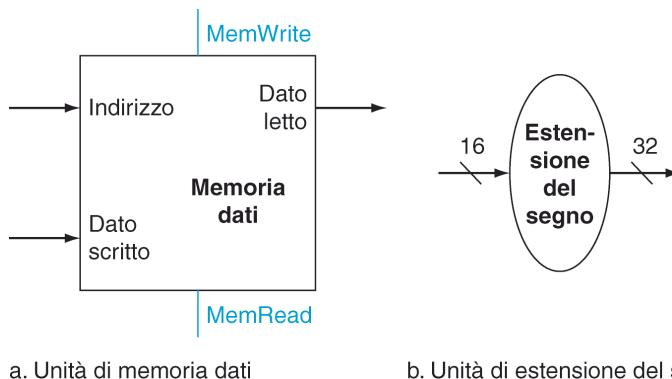
Adesso andiamo ad analizzare le seguenti istruzioni sulla memoria:

¹ `lw $t0, offset($t2)`
² `sw $t0, offset($t2)`

Ricordiamo che presentano la seguente mappatura:



In entrambi i casi avremo bisogno di calcolare l'indirizzo in memoria dato da `$t2` e dallo spiazzamento, per cui ci servirà nuovamente una ALU.



In aggiunta:

- avremo bisogno di una *memoria dati*, dove eventualmente potremo memorizzare i dati di `sw`;

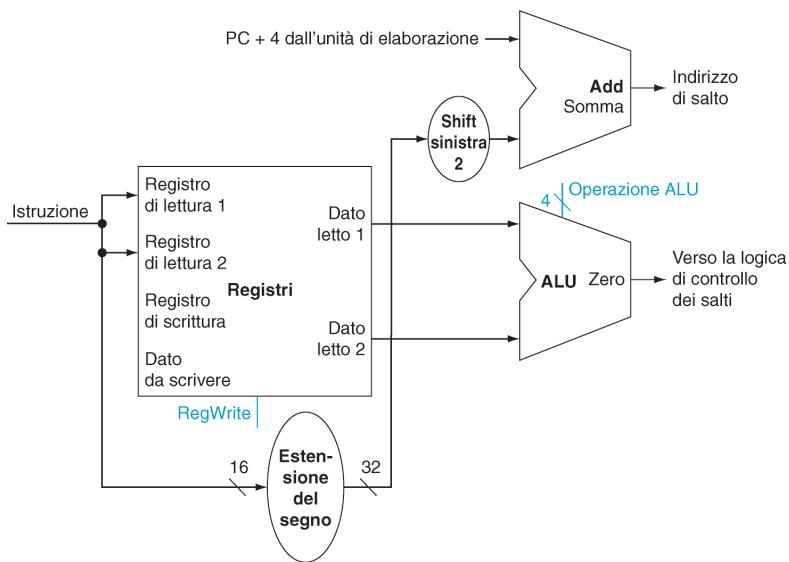
- abbiamo bisogno di due segnali di controllo appositi per indicare se l'operazione da eseguire è di lettura o scrittura;
- osserviamo che l'offset viene memorizzato in un campo a 16 bit che potrebbe essere necessario estendere a 32 bit replicando per 16 volte il bit di segno: per questo abbiamo bisogno di un blocco funzionale apposito.

11.6.3 Istruzioni di salto condizionato

Andiamo ora a vedere la seguente istruzione:

¹ **beq \$t1, \$t2, offset**

Ovviamente anche qui bisogna sommare l'offset, per cui di nuovo necessitiamo di una ALU. Osserviamo inoltre che il PC viene incrementato di 4 byte ogni ciclo, per cui quando effettuiamo un salto andiamo a riferire l'offset al PC già incrementato; inoltre MIPS shifta automaticamente gli indirizzi di istruzioni a sinistra di 2 bit, in modo da ragionare sempre in words ed espandere in numero di indirizzi raggiungibili.



Dallo schema emergono:

- una prima ALU, preposta al calcolo dell'indirizzo di salto da comunicare al PC;
- una seconda ALU, che effettua una sottrazione fra i due valori in ingresso. Se il risultato è 0 (e quindi i due sono uguali) lo comunica, insieme ad un

segnale che indica l'avvenimento di una istruzione di branch, alla porta AND. Il risultato di quest'operazione logica verrà infine inviata al multiplexer che si occupa di selezionare correttamente il valore del nuovo PC. In particolare:

- se l'AND risulta 0, $PC = PC + 4$;
- se l'AND risulta 1, $PC = PC + 4 + ind_relativo$, dove $ind_relativo$ è l'indirizzo relativo all'istruzione a cui effettuare il **beq**.

11.7 Prima progettazione completa di una CPU

Dopo aver visto in generale quali sono le principali componenti di una CPU e il loro funzionamento di base, possiamo cimentarci nella progettazione di una CPU completa.

Uno dei prerequisiti fondamentali per la nostra CPU è quello di essere in grado di eseguire ciascuna operazione in un singolo ciclo di clock: risulta perciò impossibile utilizzare una singola unità funzionale più di una volta all'interno di uno stesso ciclo. Proprio per questo motivo, come abbiamo visto, la memoria viene divisa in due: una zona viene dedicata alle istruzioni, l'altra ai dati.

In aggiunta, la nostra CPU dovrà implementare alcune semplici istruzioni aritmetico/logiche (**add**, **sub**, **and**, **or** e **slt**), istruzioni di lettura e scrittura in memoria (**lw** e **sw**) e istruzioni di salto (condizionato nel caso di **beq** e non condizionato nel caso di **j**).

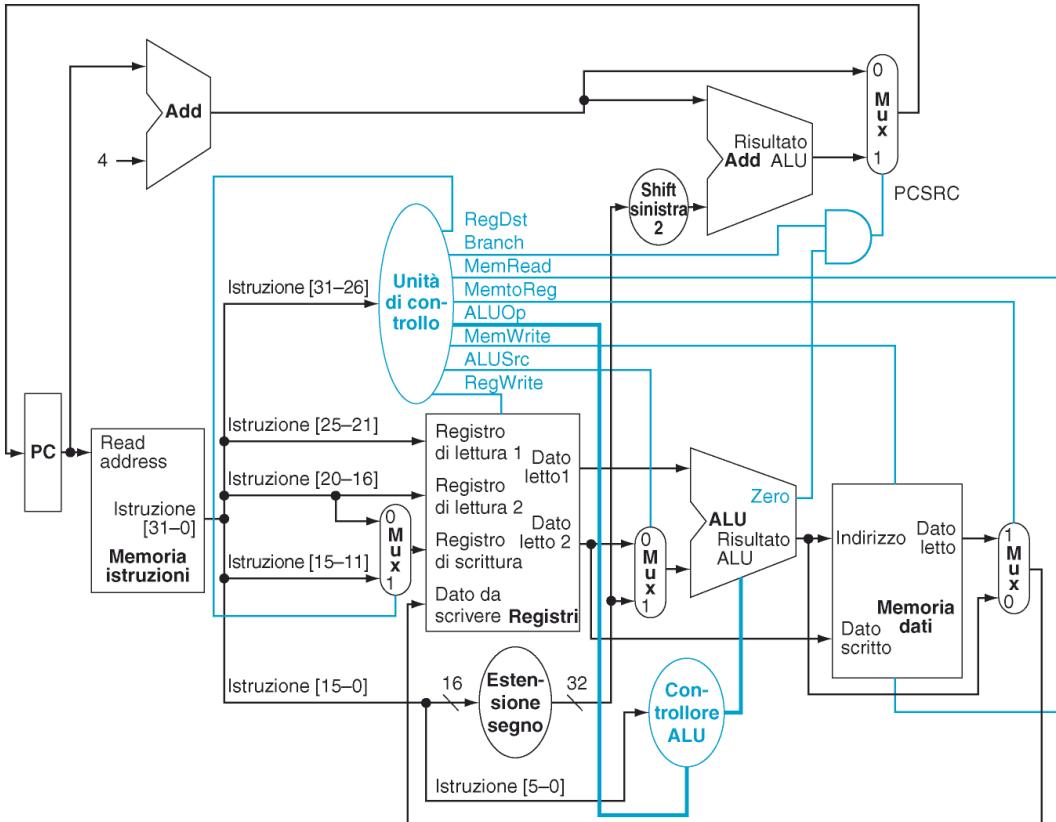


Figura 11.4: Schema complessivo di un'unità di elaborazione

Anche se ne parleremo più approfonditamente in futuro, da questa rappresentazione è già possibile notare come varino i segnali di controllo in base al tipo di istruzione che viene eseguita. Ad esempio:

- con istruzioni di tipo R: $\text{ALUSrc} = 0$, $\text{REGwrite} = 1$, $\text{MemtoReg} = 0$;
- con istruzioni `lw`: $\text{ALUSrc} = 1$, $\text{REGwrite} = 1$, $\text{MemtoReg} = 1$.

11.7.1 Progettazione dell'unità di controllo della ALU

L'*unità di controllo della ALU* è una rete logica combinatoria che ha lo scopo di informare la ALU riguardo al tipo di operazione da eseguire sui dati ricevuti come input. Questo è possibile solamente se viene assegnato ad ogni istruzione un codice univoco (che nella nostra CPU sarà a 4 bit), chiamato *codice di controllo ALU*.

Operazione	AND	OR	ADD	SUB	SLT	NOR
Codice di controllo ALU	0000	0001	0010	0110	0111	1100

Tabella 11.1: Istruzioni con il rispettivo codice di controllo ALU

Definiamo più chiaramente il codice di controllo ALU: esso viene calcolato dall'unità di controllo della ALU attraverso il campo *funct* (vedi 6.4.1) e il codice *ALUOp* dell'istruzione considerata. Brevemente:

- *funct*: corrisponde ai 6 bit meno significativi della rappresentazione in binario di un'istruzione MIPS;
- *ALUOp*: consiste in 2 bit che permettono di categorizzare l'istruzione in base al suo tipo. Essa viene generata dall'unità di controllo centrale e assume i seguenti valori:
 - 00: somma per istruzioni **lw** e **sw**;
 - 01: sottrazione per **beq**;
 - 10: operazione di tipo R.

Questo processo non è da considerarsi indifferente: in questo modo è possibile utilizzare solo 4 bit per indicare all'ALU quale operazione eseguire, piuttosto che degli 8 bit inizialmente necessari (rispettivamente 6 per il campo *funct* e 2 per il codice di controllo *ALUOp*).

A seguire vengono presentate delle tabelle che esplicitano la rappresentazione delle istruzioni considerate sia nella rappresentazioni a 4 bit che a 8².

Codice operativo istruzione	ALUOp	Operazione eseguita dall'istruzione	Campo Funct	Operazione dell'ALU	Ingresso di controllo alla ALU
Lw	00	load di 1 parola	XXXXXX	somma	0010
Sw	00	store di 1 parola	XXXXXX	somma	0010
Branch on equal	01	salto condizionato all'uguaglianza	XXXXXX	sottrazione	0110
Tipo R	10	somma	100000	somma	0010
Tipo R	10	sottrazione	100010	sottrazione	0110
Tipo R	10	AND	100100	AND	0000
Tipo R	10	OR	100101	OR	0001
Tipo R	10	set on less than	101010	set on less than	0111

Figura 11.5: Generazione del codice di controllo ALU

²I valori dove sono presenti le 'X' non vengono considerati.

ALUOp		Campo Funct						Operazione
ALUOp1	ALUOp2	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

Figura 11.6: Tabella verità dell'unità di controllo ALU

Tirando le somme, quello che abbiamo appena visto è un sistema di decodifica e generazione dei comandi detto *a due livelli*:

- *livello 1*: l'unità di controllo genera i segnali di controllo ALUOp per l'unità di controllo della ALU;
- *livello 2*: l'unità di controllo ALU genera i segnali di controllo per la ALU.

11.7.2 Progettazione dell'unità di controllo principale

L'*unità di controllo principale* è una rete combinatoria che prende come input il codice operativo (in termini tecnici *opcode*, vedi 6.4.1) dell'istruzione caricata e genera i segnali di controllo appropriati. A seguire un elenco dei principali segnali di controllo e dei loro effetti collaterali:

Nome del segnale	Effetto quando non asserito	Effetto quando asserito
RegDst	Il numero del registro di scrittura proviene dal campo rt (bit 20-16)	Il numero del registro di scrittura proviene dal campo rd (bit 15-11)
RegWrite	Nulla	Il dato viene scritto nel register file nel registro individuato dal numero del registro di scrittura
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2)	Il secondo operando della ALU proviene dall'estensione del segno dei 16 bit meno significativi dell'istruzione
PCSrc	Nel PC viene scritta l'uscita del sommatore che calcola il valore di PC + 4	Nel PC viene scritta l'uscita del sommatore che calcola l'indirizzo di salto
MemRead	Nulla	Il dato della memoria nella posizione puntata dall'indirizzo viene inviato in uscita sulla linea «dato letto»
MemWrite	Nulla	Il contenuto della memoria nella posizione puntata dall'indirizzo viene sostituito con il dato presente sulla linea «dato scritto»
MemtoReg	Il dato inviato al register file per la scrittura proviene dalla ALU	Il dato inviato al register file per la scrittura proviene dalla Memoria Dati

Figura 11.7: Segnali unità di controllo principale

Elenchiamo ora, per le istruzioni che abbiamo intenzione di implementare, il valore che assumono i segnali di controllo prodotti dall'unità di controllo principale:

Input o Output	Nome del segnale	Formato R	lw	sw	beq
Input	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Output	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp2	0	0	0	1

Figura 11.8: Schema complessivo di I/O di una CU principale

Lasciamo al lettore un ulteriore schema riassuntivo, in cui rimarchiamo ancora una volta la struttura delle istruzioni MIPS codificate in binario. Si noti che la prima colonna di questa tabella, ossia i 6 bit più significativi, corrispondono all'*opcode*:

Campo	0	rs	rt	rd	shamt	funct
Posizione dei bit	31-26	25-21	20-16	15-11	10-6	5-0
a. Istruzioni di tipo R						
Campo	35 or 43	rs	rt		indirizzo	
Posizione dei bit	31-26	25-21	20-16		15-0	
b. Istruzioni di load e store						
Campo	4	rs	rt		indirizzo	
Posizione dei bit	31-26	25-21	20-16		15-0	
c. Istruzioni di salto condizionato						

Figura 11.9: Campi istruzione

11.7.3 Esecuzione delle principali istruzioni

Istruzione di ADD

Per eseguire un'istruzione di somma del tipo `add $t1, $t2, $t3`, occorre:

1. prelevare l'istruzione dalla memoria e incrementare il PC di 4;
2. leggere \$t2 e \$t3 dal register file;
3. attivare la ALU passando in input i dati dal register file;
4. memorizzare il risultato nel registro destinazione.

Si ricordi che l'insieme di tutte queste operazioni va eseguito in un unico ciclo di clock.

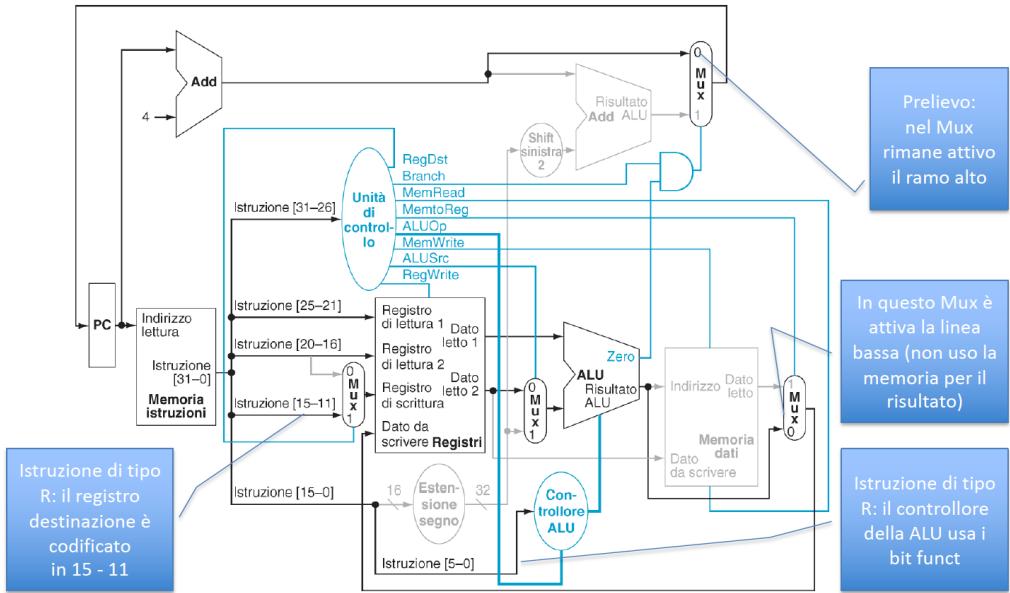


Figura 11.10: Esecuzione di ADD sulla CPU

Istruzione di LW

Per eseguire un'istruzione di `lw` del tipo `lw $t1, offset($t2)`, occorre:

1. prelevare l'istruzione dalla memoria e incrementare il PC di 4;
2. prelevare \$t2 dal register file;
3. sommare \$t2 ai campi offset dell'istruzione (i 16 bit meno significativi);
4. l'indirizzo calcolato viene inviato alla memoria dati;
5. il dato all'indirizzo calcolato viene prelevato dalla memoria dati e memorizzato in \$t1.

Si ricordi che l'insieme di tutte queste operazioni va eseguito in un unico ciclo di clock.

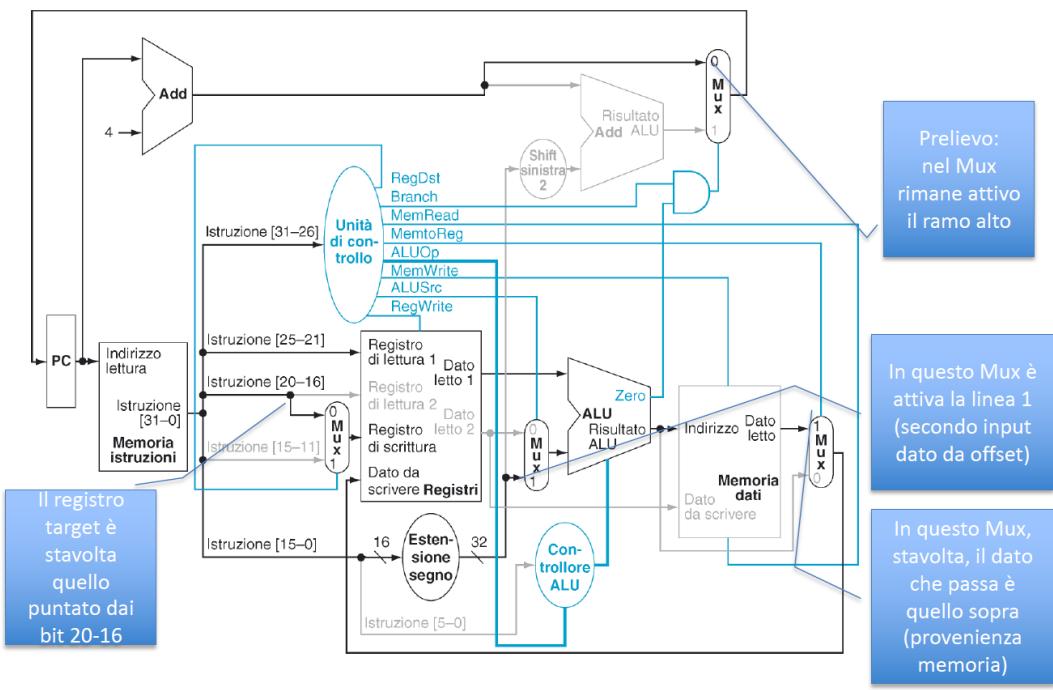


Figura 11.11: Esecuzione di LW sulla CPU

Istruzione di BEQ

Per eseguire un'istruzione di `beq` del tipo `beq $t1, $t2, offset`, occorre:

1. prelevare l'istruzione dalla memoria e incrementare il PC di 4;
2. prelevare `$t1` e `$t2` dal register file;
3. la ALU sottrae `$t1` da `$t2`. Il valore di $PC + 4$ viene sommato all'offset (esteso a 32 bit) e shiftato di due volte (per indirizzare word e non in byte);
4. il codice di controllo `zero` della ALU viene usato per decidere a cosa settare il PC.

Si ricordi che l'insieme di tutte queste operazioni va eseguito in un unico ciclo di clock.

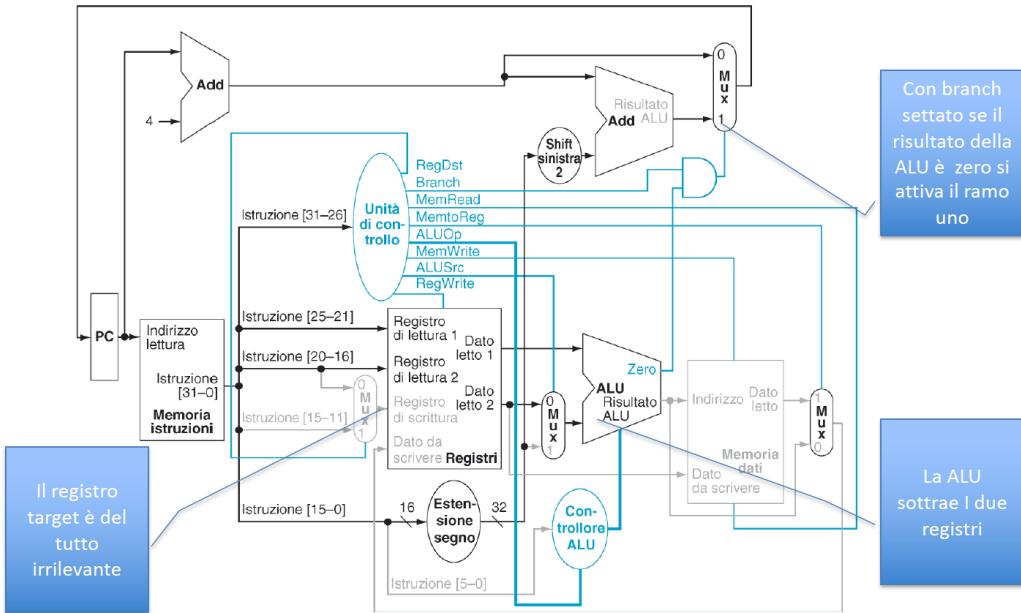


Figura 11.12: Esecuzione di BEQ sulla CPU

Istruzione di J

Prima di tutto bisogna sottolineare il fatto che l'istruzione **j** viene implementata in MIPS con una struttura molto particolare: i 6 bit più significativi costituiscono l'opcode, mentre gli ulteriori 26 bit rimanenti sono dedicati a memorizzare l'indirizzo. Per eseguire un'istruzione di **j** (salto incondizionato), occorre:

1. Prelevare l'istruzione dalla memoria e incrementare il PC di 4;
2. Sostituire al PC il nuovo valore così ottenuto:
 - i quattro bit più significativi del PC rimangono invariati;
 - i bit dal 27 all'1 (inclusi) del PC sono sostituiti con il campo indirizzo dell'istruzione **j**;
 - i due bit meno significativi sono messi a 0 (ragioniamo in word e non in byte).

Si ricordi che l'insieme di tutte queste operazioni va eseguito in un unico ciclo di clock. In particolare è da sottolineare che per implementare questa istruzione si necessita di alcune componenti hardware specifiche in aggiunta a quelle che abbiamo descritto precedentemente.

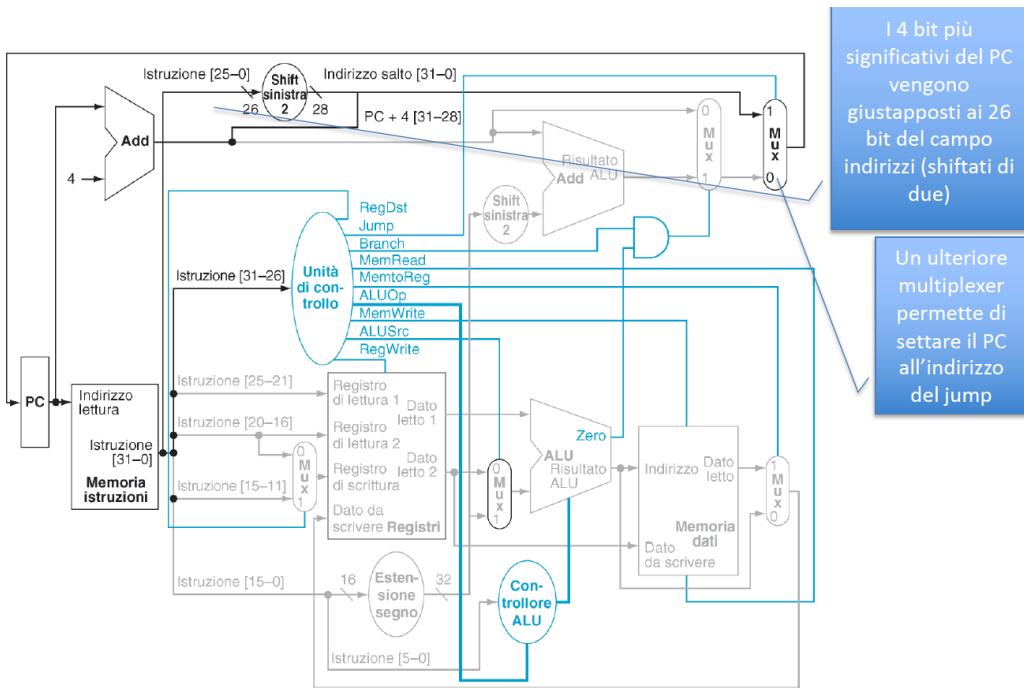


Figura 11.13: Esecuzione di J sulla CPU

Conclusione

Abbiamo appena visto come realizzare un semplice processore che esegue ciascuna delle istruzioni implementate in un singolo ciclo di clock. Solitamente però, si cerca di evitare un'implementazione di questo tipo, in quanto:

- a dettare il clock sono le istruzioni più lente (solitamente l'operazione più costosa è quella per accedere alla memoria);
- più aumenta la complessità delle istruzioni da implementare, più la progettazione diventa complessa (esempio: istruzioni dedicate ai floating point);
- non è possibile fare ottimizzazioni aggressive sulle operazioni che avvengono più frequentemente.

Capitolo 12

Pipeline

12.1 Introduzione

Nel capitolo precedente è stato presentato lo schema base di un semplice processore che esegue un'operazione per ogni ciclo di clock, tuttavia questo *modus operandi* è molto svantaggioso in quanto il clock deve essere calibrato in base alla durata delle operazioni più lente. Quindi, se si implementano istruzioni complesse, questo rallenta di molto la velocità del clock e di conseguenza quella del processore stesso, che dovrà aspettare la durata di un clock intero anche per istruzioni molto semplici che richiedono meno tempo. Tutto ciò rende impossibile compiere ottimizzazioni aggressive sulle operazioni più comuni.

Esistono, tuttavia, dei procedimenti che permettono di migliorare il rendimento della CPU. Uno dei metodi più semplici è quello di parallelizzare il lavoro, in una maniera molto simile a quella con cui Henry Ford migliorò la produzione delle fabbriche del primo Novecento.

Il punto centrale di questa strategia sta nel comprendere che mentre una componente della CPU sta lavorando sull'istruzione corrente, un'altra componente può iniziare a svolgere l'istruzione successiva; in tal modo lo svolgimento delle operazioni non è più prettamente sequenziale ma diventa parallelo, in un certo senso, perché lo svolgimento di un'operazione avviene nello stesso momento dello svolgimento di un'altra, anche se le due operazioni si trovano in fasi di completamento diverse.

Una misura del miglioramento che si introduce con metodi come il lavoro in parallelo è data dal *throughput*, il quale è definito come il numero di operazioni risolte per unità di tempo. In una catena di lavoro in parallelo composta da n stadi il *throughput* massimo raggiungibile è $\frac{1}{n}$; si intuisce che con un meccanismo di lavoro in parallelo ben funzionante a regime si possono fare n operazioni nello stesso tempo in cui normalmente se ne fa una sola, e questo è un miglioramento

notevole.

Pipeline

Il concetto di pipeline è esattamente simmetrico a quello di catena di montaggio: viene detta pipeline un'implementazione della CPU che prevede lo svolgimento di più operazioni in parallelo.

Tornando al caro vecchio MIPS, ricordiamo brevemente quali sono le fasi per l'esecuzione di un'istruzione:

1. fetch;
2. lettura registri e decodifica dell'istruzione;
3. esecuzione di un'operazione (di tipo R) o calcolo di un indirizzo;
4. accesso ad un operando nella memoria dati (nel caso di **lw** o **sw**);
5. scrittura del risultato in un registro (*writeback*).

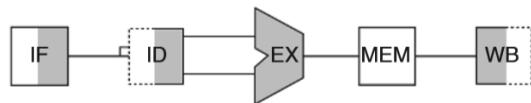


Figura 12.1: IF sta per instruction fetch, ID per instruction decode, EX per execution, MEM per accesso alla memoria dati e WB per *writeback*

Dunque, avendo un processo suddiviso in 5 step, la pipeline che useremo per parallelizzare il lavoro della CPU sarà composta da 5 stadi.

12.2 Esempio

Per comprendere meglio il concetto è presentato di seguito un esempio semplificato di pipeline, in cui le uniche operazioni possibili sono:

- **lw** e **sw** di accesso alla memoria;
- **and**, **or** e **slt** di tipo R;
- **beq** di tipo salto.

Naturalmente i tempi necessari per l'esecuzione di queste operazioni non sono uguali: è riportata quindi una tabella con annessi i tempi di esecuzione (veri solo nel nostro esempio) relativi ad ogni categoria di operazione.

Tipo di istruzione	Lettura dell'istruzione	Lettura dei registri	Operazione con la ALU	Accesso ai dati in memoria	Scrittura del register file	Tempo totale
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
Formato R (add, sub, and, or, slt)	200 ps	100 ps	200 ps		100 ps	600 ps
Salto condizionato (beq)	200 ps	100 ps	200 ps			500 ps

Figura 12.2: Tempi d'esecuzione per ciascuna fase (nel nostro esempio)

Si noti che l'istruzione più lenta di tutte è la `load word`¹; di conseguenza il tempo di clock viene stabilito in base ad essa.

Inoltre il tempo dedicato ad ogni singola fase del processo di esecuzione è definito in base al tempo della fase con durata massima, ovvero nel nostro caso 200ps, per rendere possibile il partizionamento in fasi della stessa durata (altrimenti il parallelismo non sarebbe possibile). Questo tuttavia implica che anche le fasi con durata inferiore verranno eseguite in un tempo di 200ps ed anche che la durata dell'esecuzione di una singola istruzione nella nostra pipeline è di (tempo singola fase) * (numero fasi) = $200ps \cdot 5 = 1000ps$.

Osserviamo quindi un diagramma temporale per sequenza istruzioni in cui si confronta lo svolgimento di operazioni con il metodo standard e attraverso l'implementazione della pipeline.

¹Si noti anche che questa è l'unica istruzione che esegue effettivamente tutte le 5 fasi descritte in precedenza.

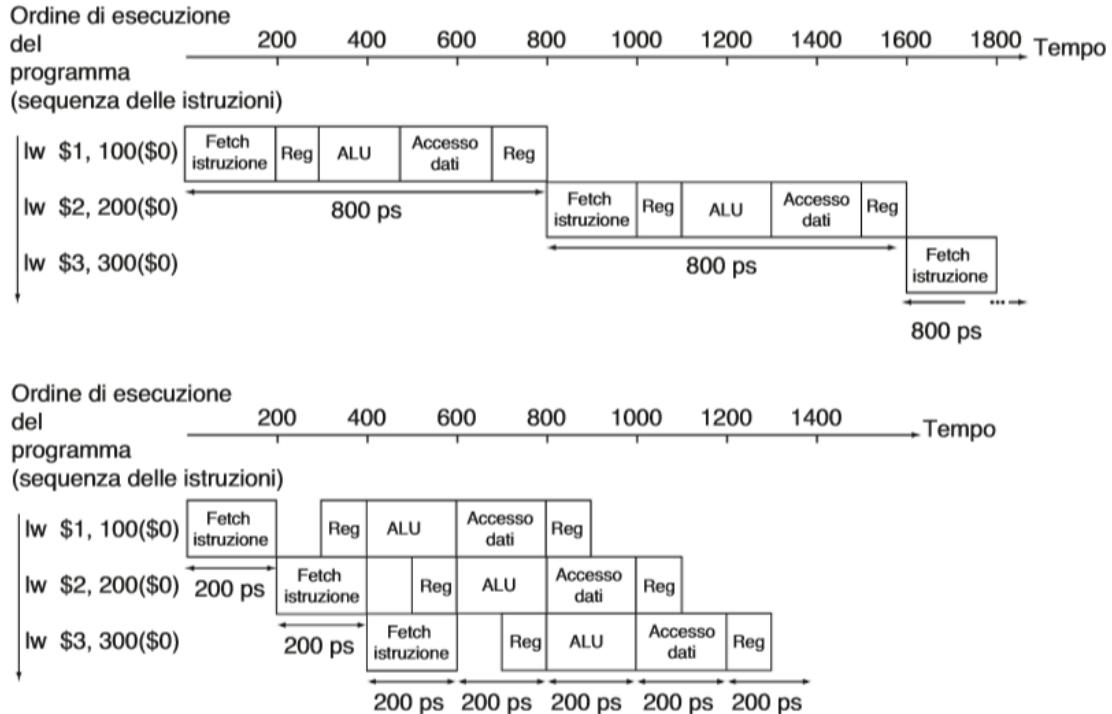


Figura 12.3: Il grafico in alto raffigura la normale esecuzione di un programma, il grafico sotto invece rappresenta lo svolgimento dello stesso con il meccanismo della pipeline; si noti che la fase reg è forzatamente fatta durare 200ps, ed inoltre si noti di quanto è ridotta la durata del programma.

12.2.1 Osservazioni

La teoria prevede che per avere una stima delle prestazioni della CPU una volta implementata una pipeline si possa usare la seguente formula:

$$\text{Tempo tra due istruzioni con pipeline} = \frac{\text{Tempo tra due istruzioni senza pipeline}}{\text{Numero di stadi della pipeline}}$$

Quindi teoricamente, secondo questa formula, nel nostro caso ci si aspetta un miglioramento delle prestazioni di un fattore 5; tuttavia si può notare nell'esempio in figura 12.3 che il tempo per eseguire 3 istruzioni passa da 2400 ps a 1400 ps, generando quindi un miglioramento di un fattore 1.7.

Le cause di questa discrepanza sono principalmente due: in primis si deve sottolineare che per ottenere una prestazione 5 volte più alta il tempo di clock dovrebbe essere uguale a quello di una singola fase del ciclo, ovvero $\frac{800}{5} = 160$ ps, mentre nella realtà non possiamo scendere sotto ai 200 ps poiché ci sono delle fasi che hanno tale durata e quindi impongono un limite minimo alla durata di una

fase. In secondo luogo abbiamo considerato poche istruzioni e questo implica che non abbiamo fatto in tempo a distribuire il costo di "avvio" e di "terminazione" della pipeline (i due periodi in cui la pipeline non è completamente a regime); il miglioramento si osserva meglio quando il numero di istruzioni è ben più alto del numero delle fasi della pipeline.

12.3 Vantaggi del RISC

L'architettura di tipo RISC presenta dei vantaggi sostanziali per quanto riguarda il processo di *pipelining*, i principali sono riassunti nell'elenco sottostante.

- Tutte le istruzioni hanno la stessa lunghezza, cosa che facilita di molto il prelievo, dal momento che si lavora sempre su word di dimensione fissata.
- I codici degli operandi sono in posizione fissa, questo permette di accedervi leggendo il file register in parallelo con la decodifica dell'istruzione.
- Gli operandi residenti in memoria sono possibili solo per `lw` e `sw`. Ciò permette di usare la ALU per il calcolo di indirizzi, cosa che non sarebbe possibile se dovessimo usare le ALU in due fasi della stessa istruzione, come invece richiesto in ISA più complesse.
- L'uso di accessi allineati fa sì che gli accessi in memoria avvengano sempre in un ciclo di trasferimento, impegnando quindi un solo stadio della pipeline.

12.4 Condizioni di Hazard

In condizioni normali una pipeline permette di eseguire un'istruzione per ciclo di clock, ma questo non è sempre possibile, soprattutto quando si verificano delle determinate condizioni critiche dette *hazard*. Ne esistono diversi tipi:

- hazard *strutturali*;
- hazard *sui dati*;
- hazard *sul controllo*.

Osserviamo quindi come vengono a verificarsi queste condizioni critiche e quali sono le soluzioni più comuni adottate per risolverle.

12.4.1 Hazard strutturale

Una condizione di hazard strutturale è una condizione che si verifica quando l'architettura dell'elaboratore rende impossibile l'esecuzione di alcune sequenze di istruzioni in pipeline. Ad esempio, se si disponesse di un'unica memoria, non si potrebbe nello stesso ciclo di clock caricare istruzioni e memorizzare (o prelevare) operandi dalla memoria: è proprio per questa ragione che la memoria dati è separata dalla memoria istruzioni.

12.4.2 Hazard sui dati

Per spiegare questo hazard ricorriamo al seguente esempio di codice MIPS:

```

1 add $s0, $t0, $t1
2 sub $t2, $s0, $t3

```

Andando ad analizzare come la pipeline implementerebbe questo codice si nota che `$s0` viene salvato nella quinta fase di `add`, ma lo stesso registro è necessario nella seconda fase di `sub`; volendo implementare la pipeline si dovrebbe lasciare il processore in stallo per il tempo necessario per la memorizzazione di `$s0`.

Esistono dei metodi per evitare queste tre intere fasi di attesa: il più semplice (e spesso anche il più funzionale) prevede un rimescolamento delle istruzioni. Esso consiste nell'allontanare tra loro le due istruzioni quanto basta per far sì che la prima termini in tempo per garantire la corretta esecuzione della seconda; tuttavia questa strada non è sempre percorribile.

Quella appena descritta è una delle operazioni che compilatori come il `gcc` compiono nella fase di traduzione da linguaggio ad alto livello ad Assembly. Una soluzione ancora più efficiente è quella dell'*operand forwarding* (detto anche propagazione o bypass): questa in casi come questo prevede di rendere il contenuto del registro `$s0` disponibile per l'operazione successiva ancora prima che questo sia effettivamente salvato nel registro stesso, dato che quel valore sarà già noto dalla fase "execute" dell'istruzione `add`. Si osservi lo schema riportato qui sotto per una maggiore comprensione del meccanismo di propagazione.

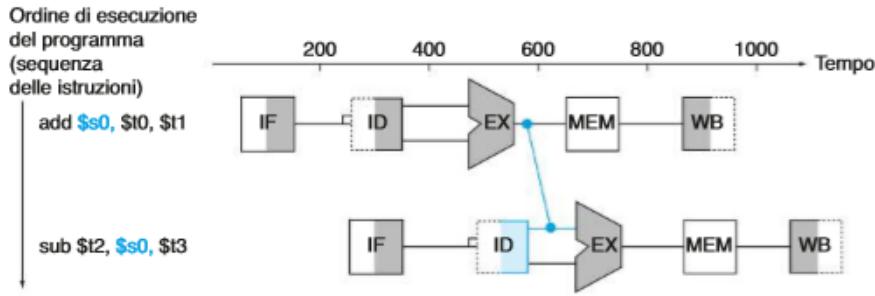


Figura 12.4: Meccanismo dell'operand forwarding

12.4.3 Hazard sul controllo

Questo tipo di hazard riguarda nello specifico le operazioni di salto condizionato, che generano problemi in quanto prima dell'effettivo termine della loro esecuzione non si può sapere quale sarà l'istruzione successiva. Se la pipeline è composta da tante fasi il tempo che si aspetterebbe prima di sapere come procedere potrebbe diventare eccessivo.

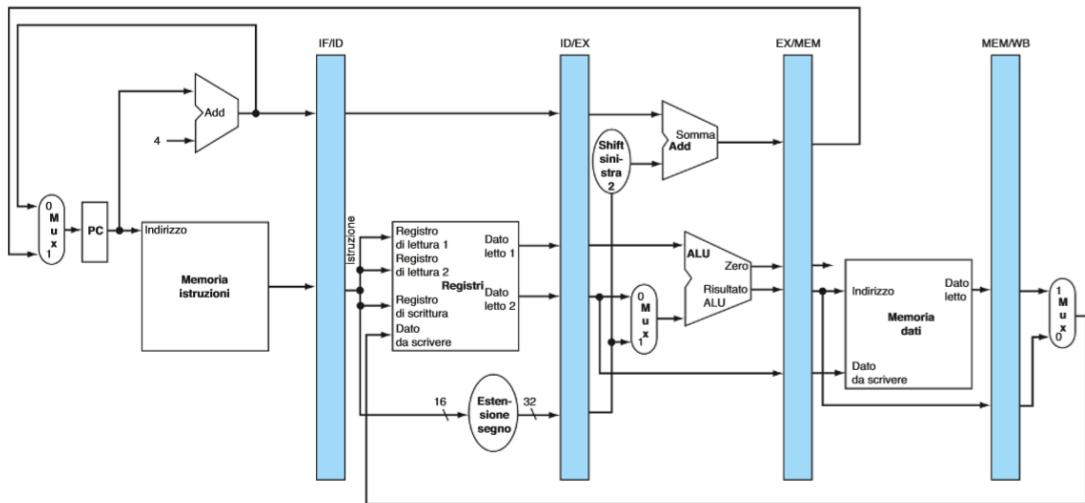
Per il resto del capitolo consideriamo di avere un circuito molto sofisticato che ci permette di calcolare l'indirizzo di salto già al secondo stadio (ID); tuttavia questo non basta poiché dobbiamo lasciare la pipeline in stallo per almeno una fase prima di sapere qual è l'istruzione successiva. Esistono tre tipi di soluzione all'hazard sul controllo, e sono riportate in seguito in ordine di complessità.

1. Il primo approccio prevede di risistemare l'ordine delle istruzioni (come si farebbe per risolvere un hazard sui dati) facendo eseguire subito dopo il branch delle istruzioni indipendenti dal salto stesso, che quindi dovrebbero essere eseguite indifferentemente; questo metodo è molto comodo, ma purtroppo non è sempre applicabile.
2. Il secondo approccio viene chiamato *untaken branch* e consiste nel considerare sempre che il salto non si verifichi e far quindi proseguire la pipeline regolarmente nell'esecuzione delle operazioni. Quando invece la condizione di salto viene verificata ci sono due casi: se il salto non deve avvenire allora il problema si è risolto da solo e la pipeline può proseguire tranquillamente il suo lavoro, altrimenti si deve ripristinare lo stato del processore a quello precedente all'istruzione del salto, questa volta eseguendo il salto stesso. Questo implica la necessità di creare dei *registri di backup* dove si va a salvare per l'appunto lo stato del processore ogni volta che si esegue il fetch di un'operazione di salto condizionale.

3. Il terzo approccio è una versione più raffinata del secondo, in cui è prevista l'implementazione di una circuiteria di *branch prediction*, che serve appunto per dare una stima della probabilità con cui avverrà il salto; questo permette di avere dei miglioramenti sull'efficienza del meccanismo *untaken branch*. Un circuito di *branch prediction* calcola a runtime la probabilità con cui si verificherà un salto condizionale e può decidere se settare il comportamento standard come *untaken branch* o come *taken branch*. Se ad esempio un programma prevede che un ciclo venga eseguito 1000 volte il circuito di *branch prediction* dopo qualche ripetizione setterà la risposta standard a *taken branch* e quindi risparmierà molte fasi di ripristino dello stato della CPU dovute ad errori di predizione (che sarebbero avvenuti con certezza usando il meccanismo *untaken branch*).

12.4.4 Registri di backup

Come detto in precedenza, per meccanismi come l'*untaken branch* e l'implementazione di circuiti di *branch prediction* si rendono necessari dei registri da utilizzare come backup dello stato del processore nel caso la predizione effettuata si riveli errata. Nell'immagine presente qui sotto si possono osservare tali registri; si noti che il nome dei registri viene dato a seconda delle fasi che separano (information fetch/information decode/execute...).



Un altro aspetto particolare di questa componente riguarda le dimensioni che i registri di backup devono avere: queste infatti sono molto variabili e dipendono strettamente da cosa il registro in causa deve memorizzare. Il registro IF/ID, per esempio, deve contenere 64 bit, 32 per l'istruzione prelevata dalla memoria ed altri

32 per il contenuto del PC incrementato di 4. Gli altri tre registri di pipeline contengono rispettivamente 128, 97 e 64 bit.

12.5 Esercizio tipo

È qui riportato un esempio di possibile esercizio da esame riguardante questi argomenti. Si consideri una CPU che impiega 600 ps per la fase di fetch, 600 ps per la fase di decodifica, 500 ps per eseguire operazioni con la ALU, 400 ps per la fase di accesso alla memoria e 700 ps per la fase di scrittura nel register file.

Qual è il massimo incremento che ci si può attendere usando una pipeline? Per prima cosa si calcola il tempo totale di un'operazione senza pipeline:

$$\text{Tempo totale} = 600 + 600 + 500 + 400 + 700 = 2800 \text{ ps}$$

In seguito si ricorda che il tempo di clock una volta implementata la pipeline sarà uguale al tempo della fase con durata più lunga, ovvero nel nostro caso 700 ps. Dunque il tempo di esecuzione di una singola operazione sarà ridotto (tramite il parallelismo, quindi virtualmente) a: $\frac{\text{Tempo totale}}{\text{Operazione più lenta}}$. Ciò significa che l'incremento delle prestazioni si calcolerà come segue:

$$\text{Incremento} = \frac{\text{Tempo totale}}{\text{Operazione più lenta}} = \frac{2800}{700} = 4$$

ovvero l'incremento finale sarà di 4 volte.

Capitolo 13

La gerarchia di memoria

“ *Idealmente si desidererebbe una memoria indefinitamente grande, in cui ogni particolare parola risulti immediatamente disponibile.* ”

Burks, Goldstine, Von Neumann, 1946

13.1 Introduzione

Sottolineare quanto il concetto di memoria sia importante e necessario nei dispositivi elettronici del giorno d'oggi risulta abbastanza scontato. D'altra parte, viene fatta subito una precisazione: non esiste un'unica soluzione che permette di ottenere la memoria "perfetta"; esistono infatti diverse implementazioni, ognuna con i suoi compromessi, che variano per costo, prestazioni e capacità.

Qualche definizione

Possiamo distinguere principalmente due tipologie di memoria in base alla modalità di accesso:

- *memoria indirizzata direttamente* (memoria principale, memoria cache):
 - è volatile, ossia il suo contenuto viene perso se viene spento il calcolatore;
 - è limitata per capacità dallo spazio di indirizzamento definito dall'architettura del processore;
 - i dati contenuti nella memoria principale sono disponibili in qualsiasi momento;

- *memoria indirizzata indirettamente* (memoria periferica):
 - è di tipo permanente, ossia mantiene il suo contenuto anche senza alimentazione;
 - ha uno spazio di indirizzamento software che non dipende dall'architettura del processore;
 - i dati contenuti nella memoria periferica devono essere trasferiti alla memoria principale prima di essere utilizzati (solitamente questo processo viene mediato dal software, tipicamente il sistema operativo);

A seguire un elenco di definizioni che utilizzeremo in seguito.

- *Tempo di accesso*: è il tempo richiesto per *una* operazione di lettura / scrittura nella memoria;
- *Tempo di ciclo*: è il tempo che intercorre fra l'inizio di due istruzioni consecutive; è composto dal tempo di accesso sommato al tempo di transito del dato con cui si lavora;
- *Latenza*: è il tempo di acceso ad una singola parola; indica quanto il processore dovrebbe poter aspettare un dato dalla memoria nel caso peggiore;
- *Velocità o "banda"*: velocità di trasferimento massima in FPM¹; oltre ad essere importante per le operazioni FPM che sono legate all'utilizzo della memoria cache interne ai processori, è significativa per le operazioni in DMA (ossia quando un dispositivo periferico viene collegato alla memoria senza passare per il processore);
- *Accesso casuale*: è quella modalità di accesso in cui non vi è alcun ordine o relazione fra i dati memorizzati; è tipico delle memorie a semiconduttori;
- *Accesso sequenziale*: è quella modalità che presuppone lo scorrimento ordinato di un blocco di dati per accedere ad un suo dato; il tempo d'accesso dipende dalla posizione fisica del dato nel supporto (tipicamente nastri e dischi);
- *Random Access Memory (RAM)*: è una memoria dotata di accesso casuale che permette scrittura e lettura; viene implementata attraverso semiconduttori;
- *Read Only Memory (ROM)*: è una memoria a semiconduttori che prevede solo un accesso in lettura; esistono implementazioni sia attraverso accesso casuale che sequenziale

¹Fast Page Mode, per la definizione si guardi la sezione 13.2.3.

13.2 La memoria principale

13.2.1 Le RAM

Con la seguente immagine descriviamo come avviene la connessione logica fra memoria RAM e CPU:

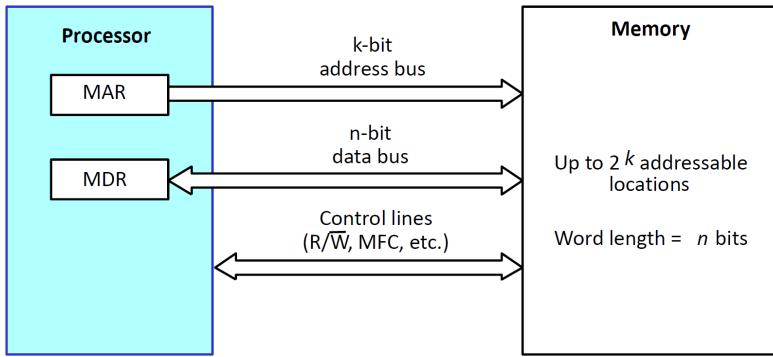


Figura 13.1: Modello della relazioni logiche fra CPU e RAM

Esistono principalmente 3 bus che permettono di far comunicare il processore con la memoria:

- *Memory Address Register (MAR)*: è un bus a k bit, dove 2^k è il numero massimo di indirizzi accessibili direttamente in memoria. Dal punto di vista pratico contiene l'indirizzo del dato (o istruzione) che si vuole caricare o scrivere;
- *Memory Data Register (MDR)*: è un bus a n bit, dove n è la lunghezza delle word nella memoria. Dal punto di vista pratico contiene il dato (o l'istruzione) che si vuole caricare o scrivere;
- *bus di controllo*: è un bus che contiene vari codici di controllo che permettono di pilotare le relazioni fra memoria e CPU. Alcune linee di controllo sono: il bit *MFC* (*Memory Function Completed*) che permette di stabilire se l'operazione di lettura o scrittura è stata completata e il bit R/\bar{W} , che distingue se si sta eseguendo un'operazione di lettura o scrittura.

Si noti che il MAR è stato rappresentato da una freccia unidirezionale: dopo aver definito l'indirizzo di memoria su cui lavorare, infatti, il processore non si aspetta il risultato. Vengono gestiti diversamente i bus di controllo e MDR, i quali sono rappresentati con una freccia bidirezionale, in quanto sia la memoria che la CPU possono occupare il ruolo di mittente del messaggio.

Una memoria RAM a semiconduttori principalmente immagazzina singoli bit, memorizzati normalmente in gruppi di byte e/o word per motivi di efficienza. La memoria non necessariamente deve essere strutturata monolicamente: è possibile che sia suddivisa in diversi blocchi, al fine di favorire il parallelismo; si noti comunque che la dimensione complessiva della memoria non varia se presa in un unico blocco o se separata. Si ricorda che l'organizzazione di una memoria influenza il numero di pin di I/O del circuito integrato: più saranno, più aumenterà il costo.

Organizzazione dei bit in un banco di memoria 16x8

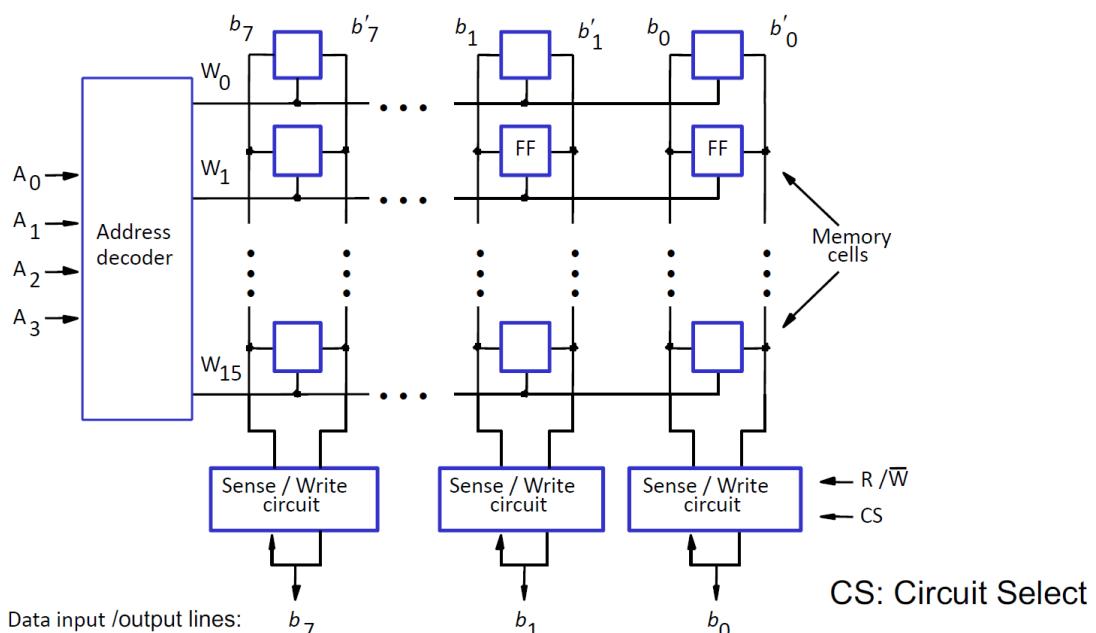


Figura 13.2: Organizzazione di un banco di memoria 16x8

Il caso presentato descrive il funzionamento di un banco di memoria formato da 16 righe e 8 colonne, con celle da un byte ciascuna. Si noti che per indirizzare un certo elemento all'interno della matrice del banco di memoria è necessario attivare una determinata riga e colonna (come avverrebbe in battaglia navale). Analizziamo ora, in ordine temporale, le operazioni che permettono di accedere/scrivere un determinato dato.

1. In base alla riga da selezionare, l'*address decoder* genera un 1 nell'uscita della riga corrispondente;
2. Avviene un meccanismo simile anche per le colonne, attraverso il *circuit select*, che decide se abilitare ciascuna colonna;

3. Ogni colonna è gestita da un *Sense/Write circuit* che riceve in input due linee di controllo: una definisce se eseguire l'operazione di lettura o scrittura (R/\bar{W}) mentre l'altra (CS) definisce se attivare o meno la colonna. Nel caso di una read, legge il valore contenuto nel bus della colonna (1 byte nel nostro esempio), altrimenti, nel caso di write, fa circolare il valore che si vuole scrivere nel bus della colonna. Per avere certezza dell'integrità dei dati viene propagato sia il dato stesso che si vuole scrivere che il suo negato. Ad esempio, nella colonna 0, b_0 contiene il dato asserito, mentre b'_0 contiene il dato negato;
4. In base a com'è stata implementata la memoria (SRAM o DRAM), viene attivata la cella selezionata dalla colonna e dalla riga designata: su di essa verrà eseguita l'operazione di lettura/scrittura richiesta.

Nelle prossime sezioni affronteremo le diverse implementazioni di cella di memoria: SRAM e DRAM.

13.2.2 Le Static Random Access Memory (SRAM)

Le SRAM, ossia le RAM statiche, sono delle memorie in cui i bit possono essere mantenuti indefinitivamente (posto il fatto che non manchi l'alimentazione). Seppur abbiano tempi di accesso molto ridotti (nell'ordine di pochi nanosecondi) e consumino poca corrente, il loro costo è molto elevato: per ciascuna cella di memorizzazione vengono impiegati molteplici componenti. Il meccanismo che sta alla base di una cella di SRAM è quello del flip flop (in particolare ai latch tipo D).

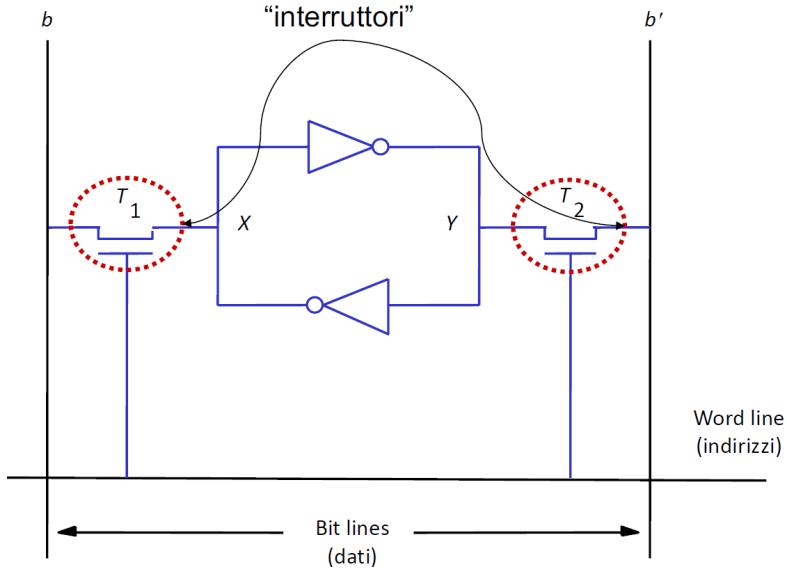


Figura 13.3: SRAM: memorizzazione di un bit

Nell'immagine, T_1 e T_2 sono due transistor²: in base al variare della tensione nel loro *gate* (ossia il collegamento inferiore) permettono o meno il passaggio di corrente fra i rami laterali. Il funzionamento ricorda molto quello di un interruttore: se il gate ha una tensione pari a V_{dd} , allora il circuito viene chiuso, altrimenti rimane aperto. Si noti che se la linea della word è bassa, allora gli interruttori T_1 e T_2 sono aperti e il consumo è praticamente nullo.

Dunque, nel caso venga chiuso il circuito, è possibile effettuare delle operazioni di lettura e di scrittura (gestite dal *Sense/Write circuit*). Invece, nel caso in cui il circuito venisse aperto, la corrente circolerebbe solamente all'interno del latch a doppio NOT. La presenza di due NOT permette di rinforzare il risultato e ridurre il tasso di errore, in quanto si vuole controllare la coerenza delle informazioni sia col segnale b che col segnale b' (si ricorda che b contiene il segnale asserito, mentre b' contiene il segnale negato: $b = \bar{b}'$).

13.2.3 Le Dynamic Random Access Memory (DRAM)

Le DRAM, ossia le RAM dinamiche, sono le memorie più diffuse nei computer. Sono caratterizzate da costi molto contenuti e di un'elevatissima densità, dovuta al fatto di dover impiegare un solo componente per ogni singola cella di memoria. Il concetto che sta alla base della DRAM è un circuito RC (la resistenza è quella data dal filo): la capacità di memorizzazione viene ottenuta attraverso la carica di

²Attraverso gli ultimi ritrovati dell'ingegneria, è possibile realizzare dei transistor nell'ordine di grandezza dei nanometri.

un condensatore. Presentano però un difetto che comporta consumi molto elevati: la cella di memoria necessita di un refresh continuo, altrimenti il proprio contenuto svanisce a causa delle correnti parassite e per la scarica del condensatore.

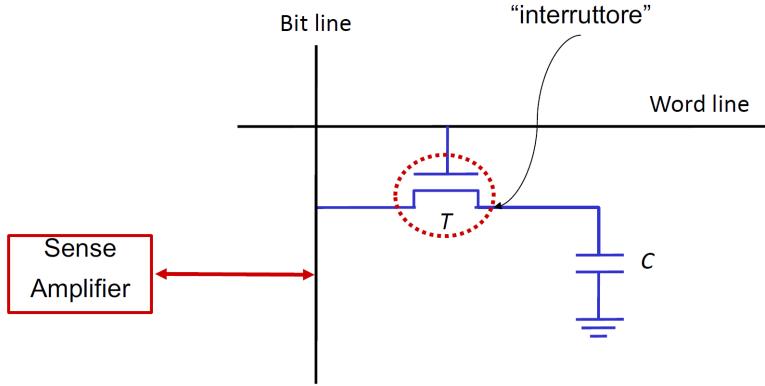


Figura 13.4: DRAM: memorizzazione di un bit

Nell'immagine T è un transistor: svolge una funzione di interruttore, in quanto permette di collegare o meno il circuito RC alla bit line e alla word line. Se il circuito viene chiuso il condensatore viene caricato, altrimenti si scarica (ricordiamo che il processo di carica/scarica di un condensatore è esponenziale). Si ricorda che se non viene attivata la bit line o la word line, allora il circuito RC è scollegato, con la conseguente scarica del condensatore a causa della resistenza applicata. Dunque, se si vuole mantenere in memoria il dato 1, c'è la necessità di continuare a scrivere 1 nella cella di memoria ad ogni intervallo t_0 , altrimenti la scarica del condensatore annullerebbe il valore. Se il valore che si vuole memorizzare è 0, basta semplicemente lasciare disattivato il circuito (con attenzione ad avergli lasciato il tempo necessario affinché si scarichi prima di riutilizzarlo). Dunque per "refresh-share" la memoria basta fare un ciclo di lettura: questo processo viene eseguito da un circuito di refresh, in maniera tale che l'utente non si debba preoccupare di questo meccanismo.

Si noti che nelle memorie DRAM non è implementato il latch a doppio NOT. Solitamente si può considerare un condensatore scarico nel momento in cui la tensione ai suoi capi è inferiore a $\frac{V_{dd}}{2}$. In particolare è proprio questo il compito del circuito *sense amplifier*, il quale determina lo stato asserito o negato del dato memorizzato in base alla soglia $\frac{V_{dd}}{2}$.

La multiplazione degli indirizzi

Facciamo un salto indietro di astrazione. L'immagine rappresenta l'organizzazione di una DRAM. A è l'indirizzo con cui si indica un dato in memoria: dalla

posizione 20 a 9 definisce il valore da dare in input al decoder delle righe, dalla posizione 8 alla 0 definisce il valore da dare al decoder delle colonne. Questa DRAM è caratterizzata da 4096 (2^{12} , dove 12 rappresenta il numero di bit di A per indirizzare la riga) righe e 512 (2^9 , dove 9 rappresenta il numero di bit di A per indirizzare la colonna) colonne:

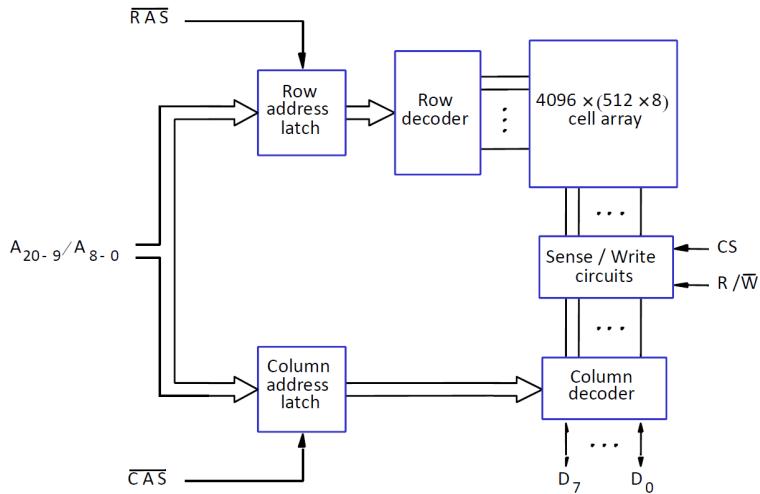


Figura 13.5: Organizzazione generale di una memoria

Si noti che sono stati inseriti due circuiti: *row address latch* e *column address latch*, che rispettivamente ricevono in input i codici di controllo \overline{RAS} e \overline{CAS} . Queste due linee di controllo permettono di decidere, in base al loro input, se è possibile accedere ad una riga/colonna.

Fast Page Mode (FPM)

Solitamente i trasferimenti da/per la memoria avvengono a blocchi piuttosto che per singole celle. Ciò comporta un notevole risparmio di tempo, in quanto sarebbe necessario solo un unico indirizzamento per le righe e colonne. Questo processo viene chiamato Fast Page Mode (FPM). Si sottolinea che questa modalità è del tutto automatica: l'utente non deve preoccuparsi di gestire questo meccanismo.

Le DRAM sincrone

Le DRAM che sono state descritte fin'ora sono dette *asincrone*, in quanto non esiste una precisa temporizzazione di accesso: la dinamica viene governata dai segnali \overline{RAS} e \overline{CAS} . Si noti che questa asincronicità può causare non pochi problemi, soprattutto mentre si esegue il refresh della memoria. Una soluzione

consiste nell'aggiungere dei buffer di memorizzazione degli ingressi e delle uscite, in maniera tale da riuscire a disaccoppiare la lettura e scrittura dal refresh della memoria, ottenendo automaticamente un accesso FPM pilotato dal clock. Questa implementazione della DRAM viene detta *sincrona*. A seguire un'immagine che contiene uno schema riassuntivo:

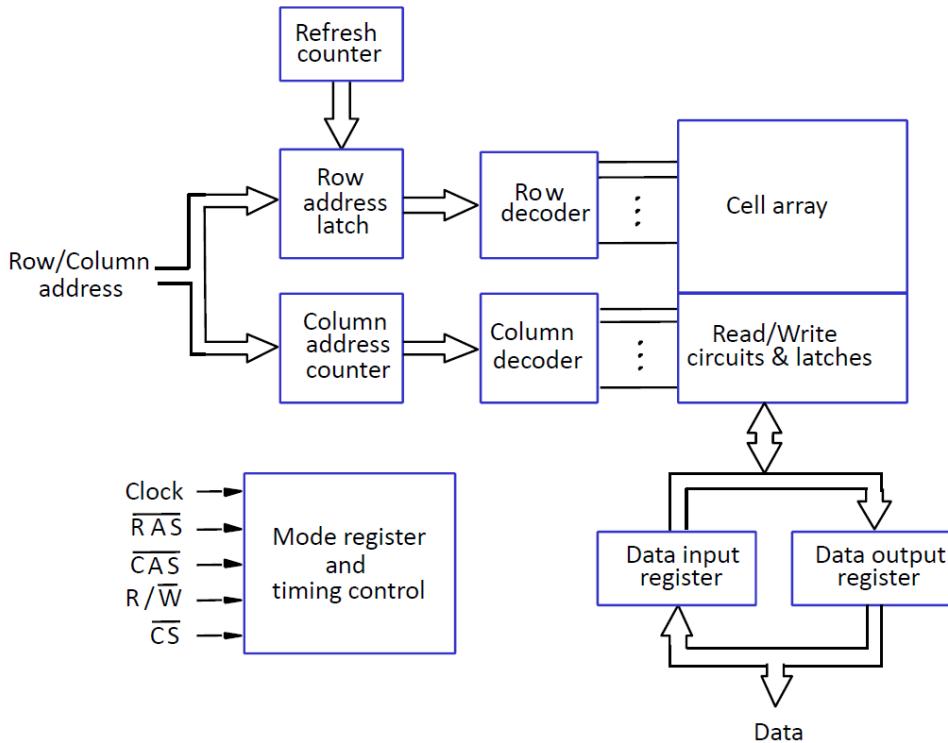


Figura 13.6: Organizzazione generale di una DRAM sincrona

Nella prossima immagine è possibile visualizzare un accesso sincrono in FPM:

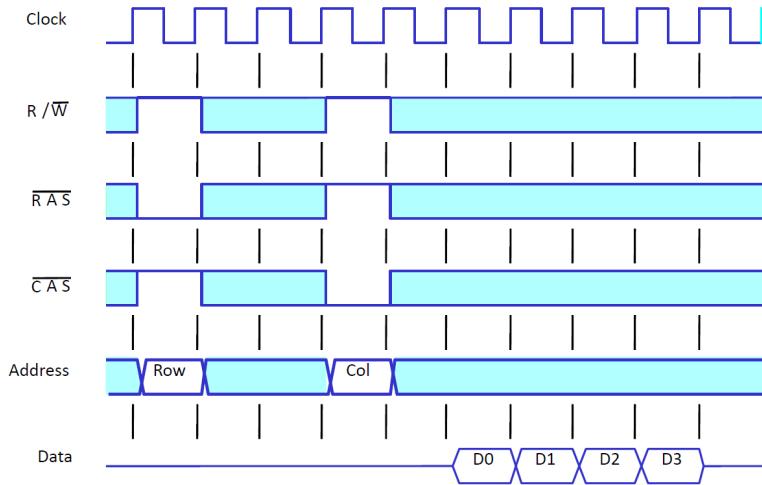


Figura 13.7: Funzionamento fast page mode

dove il segnale di clock rappresentato è quello della memoria RAM: le operazioni di lettura e scrittura sono sincronizzate col fronte di salita; D0, D1, D2, D3 sono i byte che arrivano in serie; i valori rimanenti sono le linee di controllo.

Le Double Data Rate Synchronous Dynamic Random Access Memory (DDR-SDRAM)

Un ulteriore miglioramento è effettuato dalla Double Data Rate Synchronous Dynamic Random Access Memory (DDR-SDRAM): essa permette il trasferimento dei dati sia sul fronte di salita che sul fronte di discesa del clock della memoria. In poche parole ciò permette di raddoppiare le prestazioni rispetto ad una normale DRAM sincrona, e nonostante la latenza rimanga invariata, la banda viene raddoppiata.

Sono ottenute organizzando la memoria in due banchi separati: uno contiene le posizioni pari (a cui si accede durante il fronte di salita), l'altro contiene quelle dispari (a cui si accede durante il fronte di discesa). Locazioni continue risultano divise fra i due banchi: grazie a questo è possibile effettuare un accesso in modo interlacciato.

13.3 Gestione della memoria

Andiamo adesso a domandarci quale possa essere il modo migliore di gestire la memoria che abbiamo a disposizione. È preferibile conservare i dati nella memoria RAM? Così facendo possiamo avere dei tempi di accesso ai dati estremamente

ridotti, ma di contro la quantità di dati di cui possiamo disporre sarà limitata. Dobbiamo quindi affidare i nostri dati alle memorie periferiche? In questo modo possiamo avere moltissimo spazio dove riporre i nostri dati, ma il tempo necessario a raggiungere di volta in volta quelli che ci servono sarà decisamente maggiore.

L'approccio vincente è quello di mantenere i dati archiviati nelle memorie periferiche (dove lo spazio abbonda), e ogniqualvolta siano necessari prelevarli e caricarli sulla RAM; quando questa sarà piena, i dati non più necessari verranno riarchiviati per far spazio a quelli nuovi. In questo modo possiamo avere botte piena e moglie ubriaca, perché possediamo una grande quantità di dati archiviati e possiamo accedere alla maggior parte di questi in modo veloce. Andiamo a formalizzare quanto appena detto nei due seguenti *principî di località*.

- *Principio di località temporale*: questo principio dice che se faccio uso di un dato in memoria è molto probabile che nel breve periodo dovrò servirmene di nuovo. Ad esempio, se ci troviamo all'interno di un ciclo, dovrò accedere alle stesse zone di memoria ad ogni iterazione.
- *Principio di località spaziale*: questo principio dice invece che quando accedo a una locazione di memoria è molto probabile che io debba accedere anche a zone a lei adiacenti (o contigue); questo per esempio succede ogni volta che dobbiamo lavorare per gli array, che sono ordinati per word successive.

13.3.1 Struttura della gerarchia

Osserviamo la seguente tabella:

Tecnologia di Memoria	Tempo di accesso tipico	\$ per GB (2008)
SRAM	0.5-2.5ns	\$2000 - \$5000
DRAM	50 – 70 ns	\$20 - \$75
Dischi magnetici	5 000 000 – 20 000 000 ns	\$0.2 - \$2

Al di là dei dati un po' desueti³, si può facilmente intuire quale sarà la gerarchia delle memorie: quelle più veloci ma costose saranno vicine al processore e gli forniranno i dati da elaborare, mentre invece quelle più lente (ma più capienti) staranno lontane e fungeranno da archivio.

³Verrebbe quasi da dire che fanno ridere i polli, in effetti.

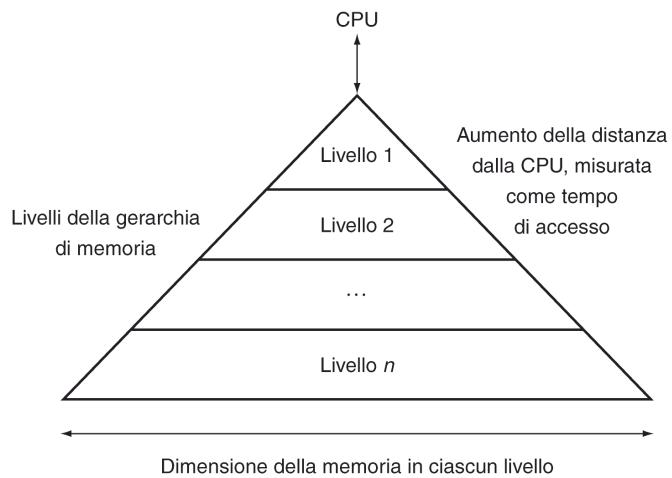


Figura 13.8: Struttura gerarchica delle memorie

13.4 La cache

La cache (dal francese *caché*, participio passato del verbo *cacehr*, "nascondere") è una memoria molto vicina al processore e invisibile al programmatore; dopo essere stata sperimentata negli anni '70 è diventata parte della progettazione di ogni tipo di calcolatore.

Altre definizioni

Forniamo ora una breve carrellata di definizioni cui bisogna prendere familiarità per affrontare la seguente sezione.

- **Blocco:** unità minima dell'informazione che può essere presente o assente in un livello superiore a quello di riferimento.
- **Hit rate:** frequenza di successo, ossia il rapporto tra il numero di accessi al livello superiore in cui trovo il dato e il numero totale di accessi che compio.
- **Miss rate:** $1 - \text{hit rate}$, frequenza degli insuccessi.
- **Hit time:** tempo che occorre a leggere il dato una volta individuato il blocco al livello superiore.
- **Miss penalty:** tempo che occorre ad accedere al dato se non trovo il blocco al livello superiore.

13.4.1 Cache a mappatura diretta

Ma come si può sapere se un determinato dato è nella cache? Com'è possibile eventualmente cercarlo?

Un sistema interessante per gestire la cache potrebbe essere quello detto *a mappatura diretta*: ad ogni indirizzo in memoria corrisponde una precisa locazione di cache; l'indirizzo in cache viene normalmente mappato come segue:

$$\text{indirizzo in cache} \equiv \text{indirizzo in memoria} \pmod{\text{numero blocchi cache}}$$

Qualora il numero di blocchi in cache fosse potenza di due sarà sufficiente usare i bit meno significativi del loro indirizzo in memoria. Quanti bit meno significativi bisogna prendere, di preciso? La risposta è in questa formula:

$$\text{Cache address} = \log_2(\text{cache dimension})$$

Se per esempio abbiamo una cache di 8 parole dobbiamo prendere i 3 bit meno significativi dell'indirizzo in memoria, come illustrato dalla seguente immagine. Si noti che l'operazione di modulo e di logaritmo restituiscono lo stesso risultato, sono perfettamente equivalenti.

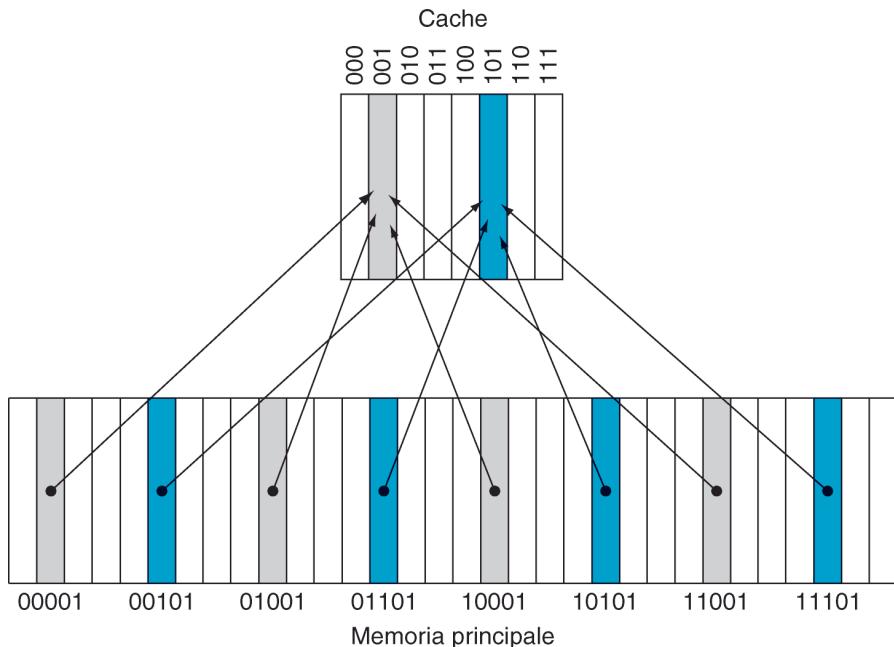


Figura 13.9: Esempio di mappatura da memoria a cache

Dal momento che molte parole in memoria possono ritrovarsi mappate nel medesimo blocco di cache sarà necessario ricorrere a dei *tag* identificativi; i bit

non utilizzati per l'indirizzamento adempiscono egregiamente allo scopo. L'ultimo elemento che vediamo è un cosiddetto *bit di validità*, fornito dal gestore della cache (quindi non prelevato dall'indirizzo del blocco) che ci dice se quello che in un dato momento abbiamo memorizzato in un blocco di cache sia valido o meno.

13.4.2 La cache in MIPS

Adesso mostriamo un esempio di come viene gestita la cache in Assembly MIPS. Assumiamo le seguenti informazioni:

- gli indirizzi sono vettori di 32 bit, naturalmente;
- la cache seguirà il criterio della mappatura diretta prima descritto;
- la cache avrà dimensione 2^n blocchi, di cui n bit usati per l'indice di ogni blocco;
- il blocco di cache sarà composto di 2^m parole, ossia 2^{m+2} bit⁴;
- la dimensione del *tag* sarà di $32 - (n + m + 2)$ bit.

Si analizzi attentamente la seguente immagine, tenendo presente che $n = 10$ e $m = 0$, quindi la dimensione del *tag* è di 20 bit.

⁴La ragione di quel $m + 2$ è naturalmente nel fatto che le parole sono sempre linee di 4 bytes, per cui il loro numero va sempre moltiplicato per 4, ossia 2^2 ; vediamo i 2 bit riservati a questo offset anche nell'immagine sottostante.

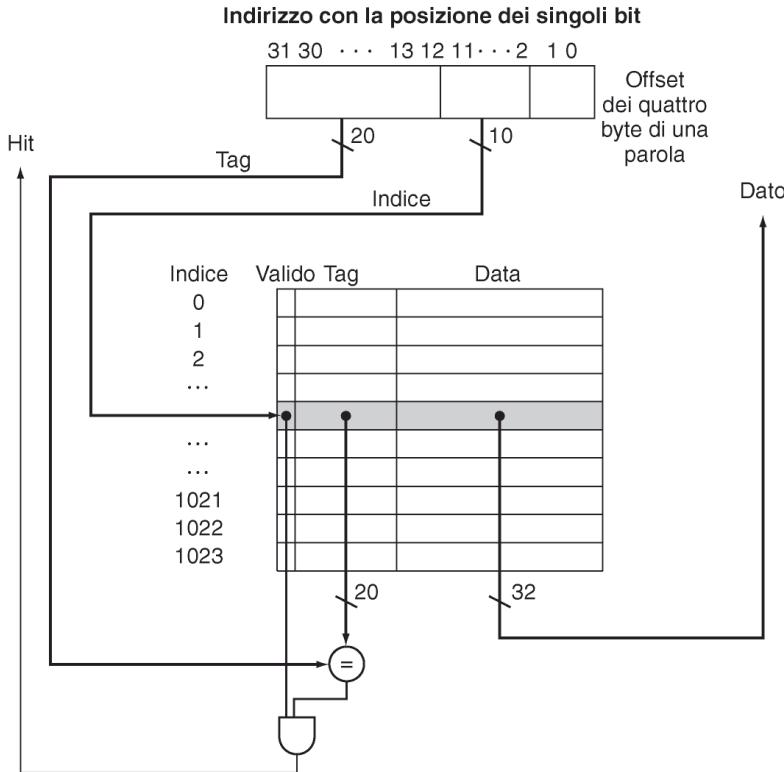


Figura 13.10: Schema risoluzione indirizzo

Il calcolo degli indirizzi

Mostriamo con un esempio come calcolare a quale blocco di cache corrisponde un determinato indirizzo in memoria.

Supponiamo di avere una cache con 64 blocchi di 16 byte ciascuno e di voler trovare a quale blocco cache corrisponde l'indirizzo 1200 in byte. Ricordiamo che l'indirizzo di ogni blocco è identificato da:

$$\text{Indirizzo blocco} \quad (\text{mod} \text{ Numero di blocchi della cache})$$

con l'indirizzo del blocco definito come segue:

$$\text{Indirizzo blocco} = \frac{\text{Indirizzo dato in byte}}{\text{Byte per blocco}}$$

- Calcoliamo l'indirizzo del blocco: $\frac{1200}{16} = 75$
- A questo punto calcoliamo l'indirizzo del blocco nella cache corrispondente: $75 \text{ (mod } 64) = 11$

Il blocco di cache cercato è quindi il numero 11.

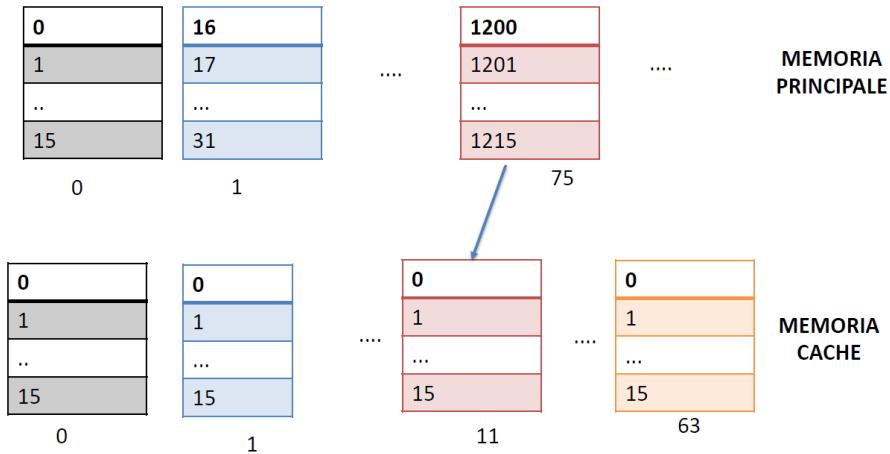
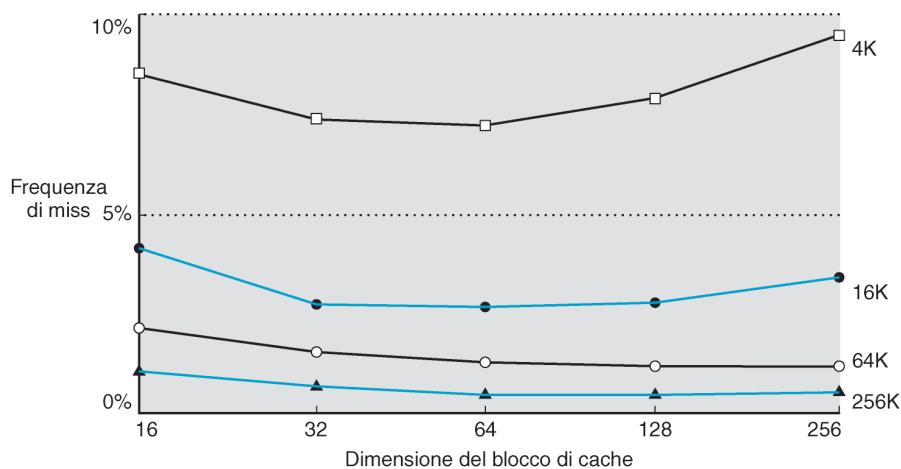


Figura 13.11: Esempio di calcolo blocco

13.4.3 La ricerca di un compromesso

A questo punto risulta evidente che avere blocchi di cache molto grandi ci permette di sfruttare ampiamente il principio della località spaziale, dal momento che disporrò di molto spazio in cui riporre gruppi di dati anche di dimensioni importanti; perché non si ricorre sempre a una soluzione di questo tipo, allora? Perché innanzitutto avere blocchi molto grandi porta a dei costi di gestione molto alti (di fatto di volta in volta ci troviamo a spostare molti byte), ma soprattutto perché a parità di dimensioni avere meno blocchi diminuisce l'efficacia della località temporale. Per questo motivo è molto importante trovare un buon compromesso tra numero di blocchi e dimensione degli stessi, in fase di progettazione.

La figura sottostante mostra come varia la frequenza delle miss (asse y) in relazione a diverse dimensioni di cache (le varie linee) e in numeri di blocchi (asse x). Osserviamo che le migliori prestazioni le abbiamo sempre per valori equilibrati, e che la maggior differenza la osserviamo per dimensioni di cache più piccole.



13.4.4 Gestione delle miss

Fintantoché non avviene una miss il funzionamento di una CPU con pipeline non viene influenzato dalla presenza della cache, ma quando una di queste viene riscontrata è necessario bloccare temporaneamente il flusso della pipeline e gestire il trasferimento dei dati dalla memoria principale alla cache. Vediamo come viene gestita una miss sulla memoria istruzioni.

1. Inizialmente bisogna raddrizzare il PC; ricordiamo che questo viene incrementato di 4 all'inizio di ogni istruzione, quindi l'eventuale miss sarà su $PC - 4$, e questo sarà infatti il valore che dobbiamo inviare alla memoria.
2. Dobbiamo inviare alla memoria il comando di eseguire una lettura ed attenderne il completamento.
3. A questo punto dobbiamo scrivere il dato proveniente dalla memoria in cache e aggiornarne i tag.
4. Infine possiamo rieseguire il fetch e trovare adesso l'istruzione in cache al posto giusto.

13.4.5 Scritture

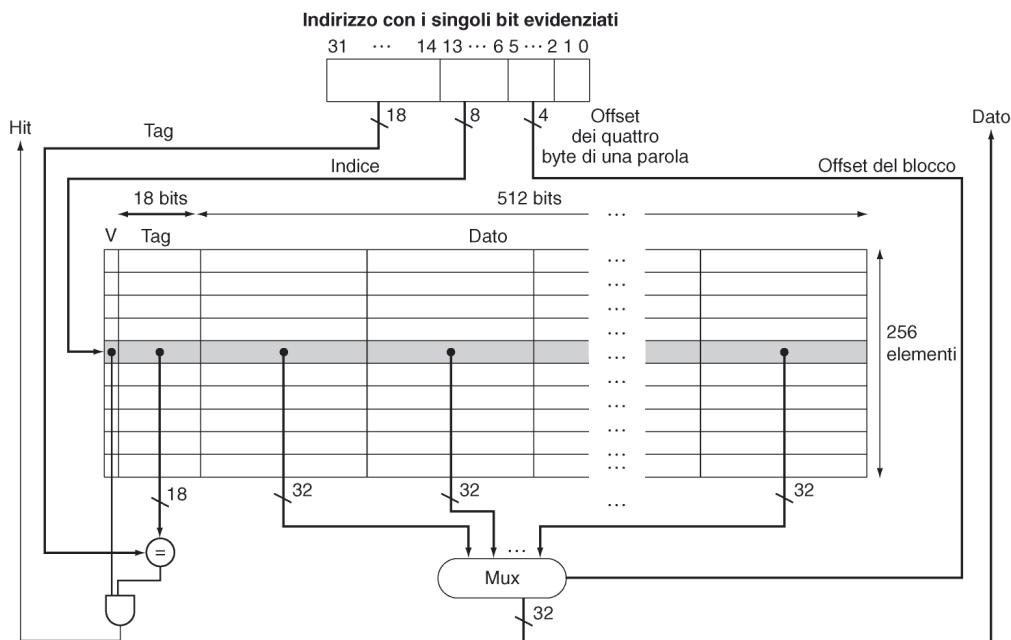
Se gli accessi in lettura avvengono con la logica precedente, quelli in scrittura sono più delicati e potrebbero causare problemi di consistenza, ossia potrei avere situazioni in cui si genera un disallineamento tra i dati presenti in cache e quelli presenti nella memoria principale.

- Un sistema che possiamo adoperare per arginare questo problema è il cosiddetto *write-through*, che consiste nel fatto che ogni scrittura viene effettuata

direttamente in memoria principale, a prescindere dal fatto che ci sia o meno una miss. In questo modo avrò cache e memoria sempre aggiornati, ma ben si capisce che si tratta di un'operazione costosa; una buona aggiunta potrebbe essere un buffer di scrittura, una sorta di "coda" dove tutte le scritture vengono messe in attesa, che ci permette quindi di accorpare il maggior numero di operazioni possibili e mantenere cache e memoria sempre allineate.

- Una seconda soluzione potrebbe essere la cosiddetta *write-back*: se il blocco richiesto è già presente in cache allora ogni scrittura verrà fatta solo localmente e la memoria principale viene aggiornata solo quando si cambia il blocco di riferimento (oppure quando un secondo processore tenta di accedere allo stesso blocco); chiaramente questa soluzione è pensata in situazioni molto affollate, quanto il processore non riesce a processare in modo efficiente le molte scritture richieste.

Riportiamo come esempio *FastMath*, una cache basata su MIPS che dispone di una capacità di 16K con 16 parole per blocco e la possibilità di lavorare sia in *write-through* che in *write-back*.



- Notiamo in basso a sinistra una porta AND: questa prenderà in input il bit di validità e il risultato del confronto tra il tag del blocco selezionato e l'indirizzo. Il risultato in output determinerà se avremo conseguito una *hit* oppure no.

- Infine osserviamo un multiplexer in cui confluiscono tutte le 16 parole del blocco; per determinare quale word debba andare in output viene impiegato come segnale di controllo un offset dall'indirizzo del blocco di cache.

13.5 Cache associative

Le cache a mappatura diretta sono piuttosto semplici da realizzare, tuttavia presentano un problema: se ho spesso bisogno di locazioni di memoria che si mappano sullo stesso blocco di cache⁵, ho cache miss in continuazione. Proprio per gestire queste situazioni è nata l'implementazione della cache completamente associativa.

13.5.1 Cache completamente associativa

Una cache completamente associativa è una cache in cui ogni blocco della memoria può essere mappato in un qualsiasi blocco della cache. Il problema in cui si incorre utilizzando cache completamente associative è che si deve cercare in tutta la cache il dato (il tag in questo tipo di cache è rappresentato da tutto l'indirizzo del blocco). Per effettuare la ricerca in maniera efficiente si deve quindi lanciarla su tutti i blocchi in parallelo.

Per questo motivo ho bisogno di n comparatori (uno per ogni blocco di cache) che operino in parallelo; di conseguenza la complessità di realizzazione cresce proporzionalmente al numero di blocchi ed inoltre il costo HW diventa proibitivo appena si aumenta di poco la dimensione della memoria.

13.5.2 Cache set-associativa

Le cache set-associative sono una via di mezzo tra le due che abbiamo visto finora. In una cache set-associativa, composta da n blocchi totali, tutti i blocchi vengono suddivisi in un determinato numero di gruppi, che vengono chiamati linee; ogni gruppo di conseguenza contiene un numero pari a $\frac{n}{\text{numero di linee}}$ di blocchi, che chiameremo vie.

Fatta questa suddivisione quello che si ottiene è una sorta di scacchiera in cui i blocchi possono essere visti in raggruppamenti di "linee" (righe) e di "vie" (colonne).

Riusciamo quindi ad ottenere una combinazione delle due diverse cache: mappiamo ciascun blocco della memoria centrale in una certa linea direttamente (vedi 13.4.1), in seguito questo può essere inserito in una qualsiasi delle vie in maniera associativa.

⁵Ricordiamo che questo significa che tali blocchi sono congrui in modulo n .

Quando si va a cercare un blocco quindi se ne calcola la linea in cui è mappato direttamente ed in seguito, all'interno di questa linea, si effettua una ricerca parallela su tutte le vie (come nelle cache completamente associative).

Il grande vantaggio è che abbiamo una memoria non troppo costosa, dato che la cache associativa viene implementata su blocchi di dimensioni ridotte; tuttavia si mantiene anche (almeno parzialmente) il vantaggio della strategia associativa con conseguente abbassamento del *miss rate*.

È ora riportato uno schema con una cache suddivisa in più modi differenti, ma sempre attraverso la strategia di cache set-associativa, come esempio esplicativo; si noti che ogni gruppo Tag-Dato rappresenta una via.

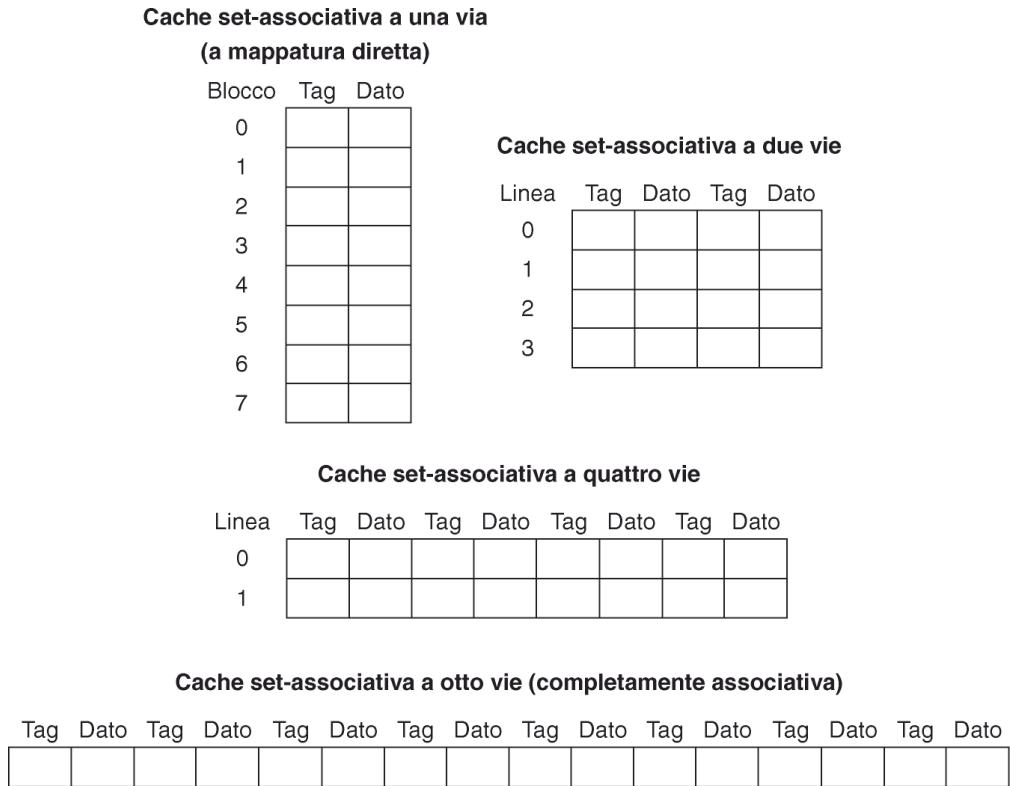


Figura 13.12: Mappatura set-associativa in varie forme

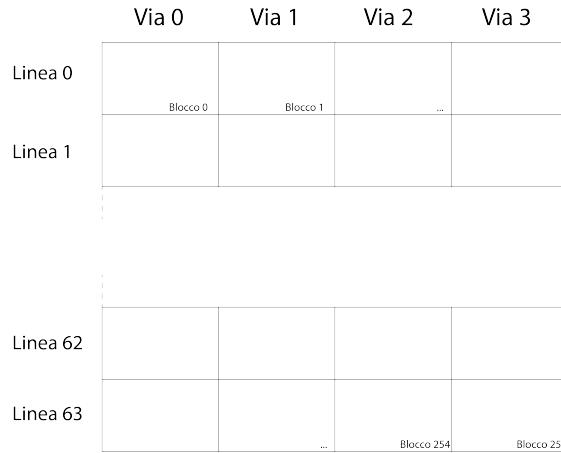
Inseriamo ora un ulteriore esempio esplicativo di come viene mappata una cache set-associativa. Immaginiamo di avere una cache di dimensioni 1KByte, composta da blocchi di grandezza 4 byte. Ora, con un rapido calcolo, troviamo il numero di blocchi presenti:

$$\text{Numero di blocchi} = \frac{\text{Dimensioni memoria}}{\text{Dimensioni blocco}} = \frac{1024}{4} = 256$$

Supponiamo ora di volerla sfruttare tramite un'implementazione set-associativa e di voler ottenere 4 vie, questo significa che il numero delle linee presenti sarà:

$$\frac{\text{Numero di blocchi}}{\text{Numero di vie}} = \frac{256}{4} = 64 \text{ linee}$$

Graficamente la nostra cache sarà quindi rappresentata da una scacchiera con 4 colonne, le vie, e 64 righe, le linee, la vediamo rappresentata qui sotto:



Ora è spiegato come viene memorizzato in cache un blocco di memoria.

Ad ogni blocco nella memoria centrale viene associata una linea con lo stesso meccanismo con cui in una cache a mappatura diretta si associa un blocco in cache.

Una volta che si identifica la linea a cui il blocco di memoria appartiene esso può essere memorizzato in uno qualsiasi dei blocchi di cache che compongono quella linea. Questo ci permette di avere in cache allo stesso momento due elementi che non sarebbero potuti essere compresi in una cache a mappatura diretta.

Tuttavia dato che il numero di posizioni (nella cache) possibili per un blocco è ridotto (è lo stesso numero delle vie) l'implementazione risulta molto più semplice rispetto ad una cache completamente associativa.

13.6 Mappatura del blocco

In una cache a mappatura diretta abbiamo già osservato come un blocco di memoria centrale venga mappato in un blocco di cache deciso dalla formula:

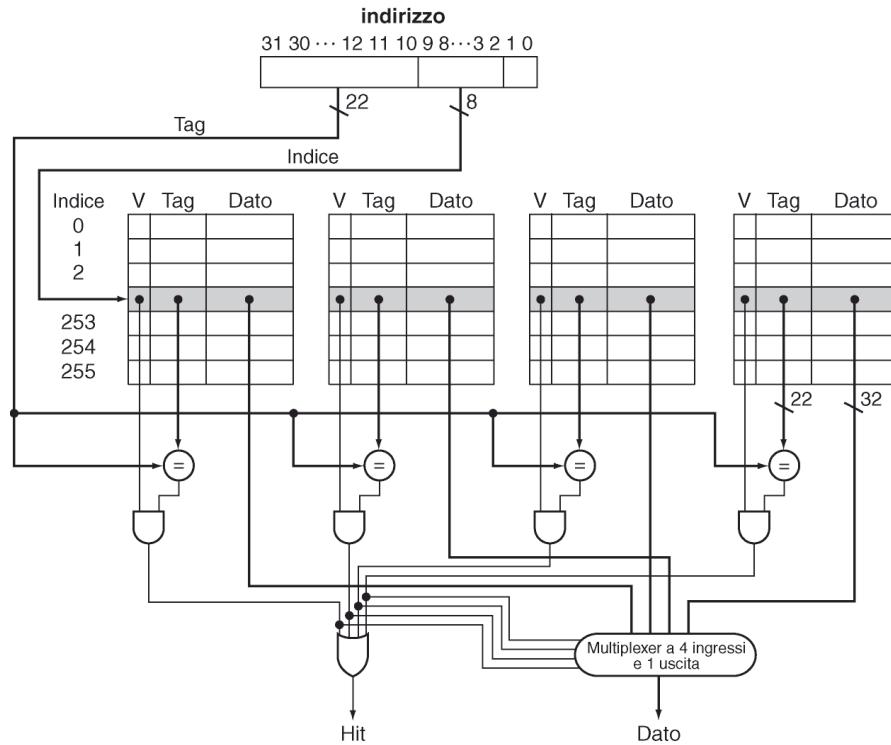
$$\text{Indirizzo blocco} \pmod{\text{Numero blocchi della cache}}$$

In maniera simile in una cache set-associativa un blocco di memoria viene mappato in una *linea* data da:

$$\text{Indirizzo blocco} \pmod{\text{Numero linee della cache}}$$

Quindi, per trovare il blocco all'interno della linea dobbiamo confrontare, in parallelo, il tag del blocco con tutti i tag dei blocchi di quella linea.

È ora inserito anche uno schema rappresentativo di una cache a 4 vie in cui ogni blocco contiene una word.



A seguire una breve descrizione dello schema.

I primi due bit dell'indirizzo (a destra) sono utilizzati per indicare, se necessario, quale singolo byte della word prelevare (sono due appunto perché rappresentano i 4 possibili byte contenuti dal blocco).

Dato che le linee possibili sono 256 i bit che servono per rappresentarle sono 8, per l'appunto sono i bit dal 2 al 9 dell'indirizzo che vengono utilizzati come indicizzatore della linea (sia per la scrittura che per la lettura).

I bit rimanenti vengono utilizzati come tag del blocco; si noti che il tag viene confrontato con tutti e quattro i tag che ritornano dalle 4 vie. Se il confronto va a buon fine vuol dire che il dato cercato è presente nella cache ed il mux a 4 ingressi si occupa di trasmetterlo al processore.

13.7 Vantaggi dell'associatività

Anche in questo caso è chiaro che la tecnologia che abbiamo trattato presenta vantaggi e svantaggi, è dunque giunto il momento di riassumerli e schematizzarli.

- **Vantaggi:** aumentando l'associatività abbiamo il grosso vantaggio di diminuire la frequenza di miss e velocizzare il processo di reperimento dei dati.
- **Svantaggi:** purtroppo con questa implementazione la complessità dell'hardware aumenta in proporzione lineare al numero di vie della cache, ed allo stesso modo aumenta il costo.

Esiste inoltre un ulteriore problema che non abbiamo trattato: nelle cache a mappatura diretta, nel momento in cui si riscontra una cache miss, sicuramente è possibile conoscere chi sostituire, ossia l'unico blocco in cui è possibile mappare il dato che cerco. Nelle cache associative invece ci sono più scelte, quindi se la linea è piena chi bisogna sostituire? In tal caso esistono vari approcci possibili, tra cui:

- *FIFO*, un immancabile classico;
- *LeastRecentlyUsed*, un metodo più sofisticato che però richiede una serie di bit in più per calcolare quando è avvenuto l'ultimo accesso.

Per concludere proponiamo al lettore anche un esempio pratico in cui si nota l'importanza della scelta di un buon meccanismo di cache mapping. Si noti come nel primo grafico sembrerebbe che il radix sort sia asintoticamente più efficiente del quick sort, in quanto il suo rapporto

$$\frac{\text{numero di istruzioni}}{\text{elemento}}$$

scende molto sotto quello del quick sort.

Tuttavia nel secondo grafico si nota stranamente che il rapporto

$$\frac{\text{numero di cicli di clock}}{\text{elemento}}$$

del radix sort rimanga sempre più alto di quello del quick sort, decretando il radix sort come algoritmo più lento.

Nel terzo ed ultimo grafico si nota il perché di questa situazione anomala: il radix sort, per come è implementato, comporta un numero assai maggiore di miss della cache per ogni elemento che tratta e questo lo porta a ripetere perdere molti cicli di clock e quindi a risultare più lento.

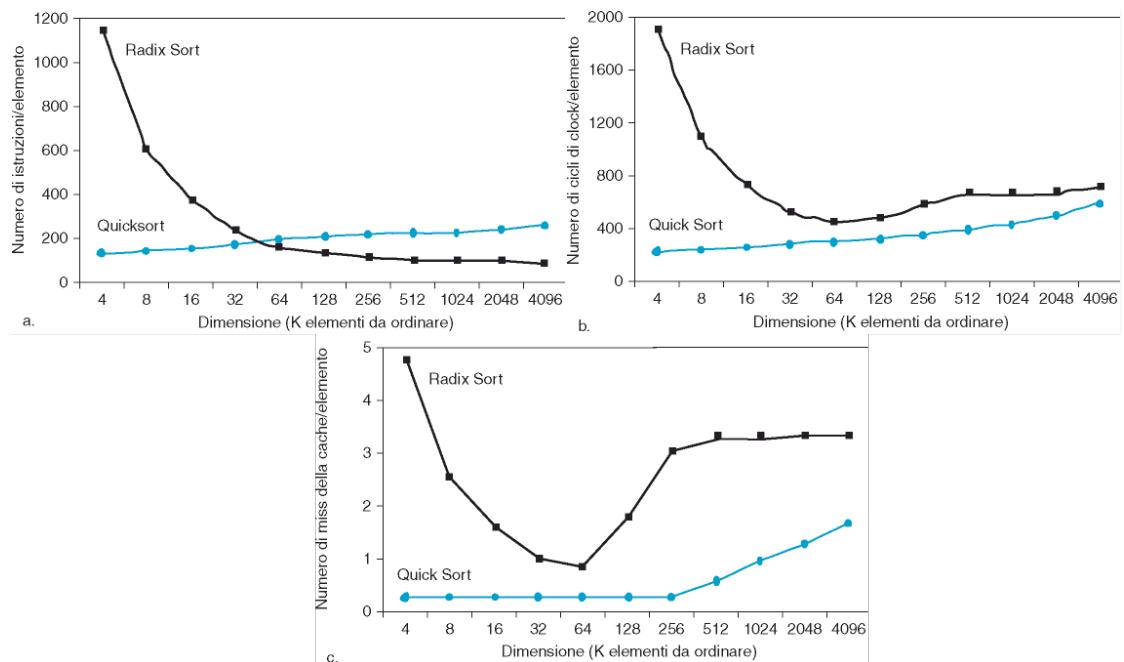


Figura 13.13: Grafici prestazioni dei due algoritmi radix e quick sort

Capitolo 14

I/O

“ I am thankful the most important key in history was invented. It’s not the key to your house, your car, your boat, your safety deposit box, your bike lock or your private community. It’s the key to order, sanity, and peace of mind. The key is “Delete”. ”

Elayne Boosler

14.1 La necessità di comunicare

Un calcolatore sarebbe praticamente inutile se non avesse la possibilità di interfacciarsi e comunicare con l'esterno, ed è proprio di questo che si occupano le periferiche di input/output (da qui in avanti I/O); questi dispositivi possono essere estremamente *eterogenei* tra loro e si diversificano a seconda del compito che sono chiamati a svolgere¹. Tuttavia devono avere una caratteristica comune: devono essere *espandibili*, ovvero deve essere prevista la possibilità di aggiungere o rimuovere i dispositivi di I/O.

Tre termini tecnici

Chiariamo ora questi termini tecnici definendone precisamente il significato poiché ci torneranno utili nel resto del capitolo.

- *Transizione di I/O*: invio indirizzo e spedizione o ricezione dei dati.

¹Si pensi che nella categoria di dispositivi I/O compaiono mouse, penne grafiche, desktop, microfoni e tanti altri ancora.

- *Input*: trasferimento di dati da una periferica verso la memoria dove il processore può leggerla.
- *Output*: trasferimento di dati dalla memoria ad un dispositivo.

Data l'enorme varietà di compiti che assolvono le periferiche di I/O a seconda del tipo di applicazione, posso essere interessato a diverse prestazioni:

- in alcuni casi la cosa più importante è il tempo di accesso/risposta (latenza), ad esempio quando si tratta di tastiere o mouse;
- in altri casi la caratteristica di spicco deve essere il throughput, vedi i sistemi di streaming;
- infine esistono molti altri casi meno comuni che necessitano di caratteristiche dedicate (ad esempio un sistema bancario può avere la necessità di massimizzare il numero di file di piccole dimensioni su cui opera contemporaneamente).

Dati questi esempi si possono introdurre le principali caratteristiche secondo cui vengono suddivise le suddette periferiche: operazioni possibili (R e/o W), *partner* (uomo o macchina) e velocità di trasferimento; in seguito viene riportata un tabella riassuntiva delle maggiori classi di dispositivi di questo tipo.

Dispositivo	Comportamento	Partner	Frequenza dati (Mbit/s)
Tastiera	Input (ingresso)	Uomo	0,0001
Mouse	Input (ingresso)	Uomo	0,0038
Input vocale	Input (ingresso)	Uomo	0,2640
Input audio	Input (ingresso)	Macchina	3,0000
Scanner	Input (ingresso)	Uomo	3,2000
Output vocale	Output (uscita)	Uomo	0,2640
Output audio	Output (uscita)	Uomo	8,0000
Stampante laser	Output (uscita)	Uomo	3,2000
Display grafico	Output (uscita)	Uomo	800,0000-8000,0000
Modem via cavo	Input o output	Macchina	0,1280-6,0000
Rete/LAN	Input o output	Macchina	100,000-10000,0000
Rete/LAN wireless	Input o output	Macchina	11,0000-54,0000
Disco ottico	Memoria	Macchina	80,0000-220,0000
Nastro magnetico	Memoria	Macchina	5,0000-120,0000
Memoria flash	Memoria	Macchina	32,0000-200,0000
Disco magnetico	Memoria	Macchina	800,0000-3000,0000

Figura 14.1: Classificazione delle periferiche

14.2 Connessione tra processore e periferiche

Nonostante le differenze di cui si è parlato nella sezione precedente, tutte le periferiche hanno una caratteristica comune: la modalità di collegamento al processore. Vediamo ora uno schema, molto semplificato, che rappresenta il modo in cui le periferiche sono collegate al processore:

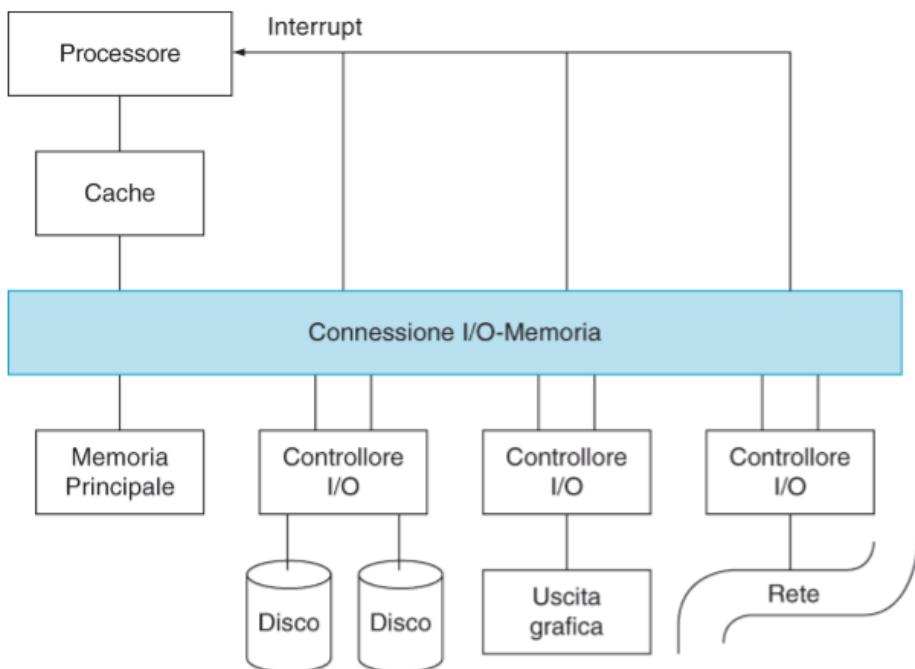


Figura 14.2: Schema del collegamento delle periferiche

Si noti quindi come ogni periferica non è collegata direttamente con il processore, la loro comunicazione è intermediata dai controllori I/O e dal bus, che descriveremo in seguito. Ciò implica che i dispositivi di I/O "non esistono" per il processore ma vengano rappresentati "virtualmente" come normali locazioni di memoria.

Tutte le connessioni avvengono attraverso strutture di comunicazione dette *bus*. Ne esistono principalmente due tipologie:

- *bus processore/memoria*: sono specializzati, corti e molto veloci;
- *bus I/O*: sono utilizzati per la comunicazione con periferiche generiche, possono essere relativamente lunghi e comunque non si interfacciano direttamente con il processore.

mente con la memoria, ma richiedono come intermediario un bus processore/memoria o un bus di sistema².

Nelle prime semplici architetture avevamo un unico grosso bus parallelo (non seriale) che collegava tutte le componenti, ma per problemi di clock e frequenze ora si usano architetture di comunicazione più complesse fatte di più bus paralleli condivisi e di bus seriali punto/punto³.

Nel caso in cui il processore comunichi con delle periferiche esterne, come anticipato in precedenza, esso in realtà comunica attraverso i controllori I/O, che traducono sia i comandi che i dati da e verso il processore stesso. Saranno ora discusse due implementazioni diverse del bus che convivono nei nostri calcolatori: *sincrona* ed *asincrona*.

14.2.1 Bus sincrono

Si tratta di un bus attraverso cui le informazioni vengono trasmesse in modo sincrono, quindi le comunicazioni sono necessariamente scandite da un segnale di bus clock (che gira a frequenza molto più bassa di quello del processore), che viene trasmesso in una linea di controllo del bus stesso.

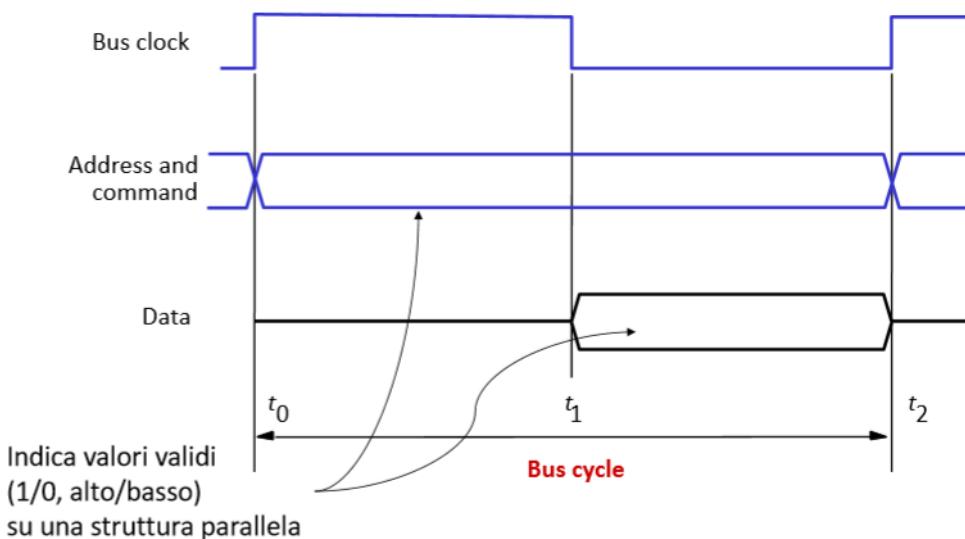


Figura 14.3: Schema rappresentativo del ciclo del bus sincrono

²Il bus di sistema è un "bus di alto livello", ovvero un bus dedicato a mettere in comunicazione tutti gli altri bus del sistema appunto.

³I bus seriali punto a punto sono bus in cui ogni periferica è collegata direttamente al processore tramite un bus dedicato.

In questo esempio si può osservare come avviene la trasmissione di un dato. Nell'*Address and command*, durante il fronte di salita del bus clock, sono trasmessi indirizzo bersaglio e comando da svolgere; sul fronte di discesa del bus clock viene effettivamente trasferito il dato. Il trasferimento termina proprio con il termine del ciclo di clock, in seguito il ciclo ricomincia; ripetendo ancora una volta per maggiore chiarezza, la prima metà del ciclo di bus clock è utilizzata per passare l'indirizzo e l'istruzione da svolgere, la seconda effettivamente per trasmettere il dato.

Riassumendo quindi i punti di forza del bus di tipo sincrono, si deve osservare che questo sistema è tanto semplice da implementare quanto veloce, data la presenza esigua di sequenze di controllo, che quindi non appesantisce lo svolgimento delle operazioni di trasmissione dati.

Di contro si può argomentare che è un sistema che dimostra poca robustezza al *drift* del clock⁴; inoltre utilizzando un sistema simile si costringono tutte le periferiche a lavorare alla stessa velocità del clock, ma questo non è sempre possibile.

14.2.2 Bus asincrono

Per ovviare agli inconvenienti appena discussi è stata sviluppata anche una versione asincrona della tecnologia del bus che sostanzialmente non si appoggia più a clock per il controllo delle transazioni ma utilizza un protocollo in cui le varie fasi sono determinate da *handshake*⁵.

Questo cambio di stratagemma libera dalla limitazione del tempo scandito dal clock ma comporta la creazione di apposite linee di controllo per segnalare inizio e fine delle transazioni; nonostante questa complicazione il bus asincrono garantisce un grande vantaggio, ovvero permette di collegare periferiche a velocità diversa.

⁴Per *drift* del clock si intende la situazione in cui il dato non si propaga in modo sincrono con il termine del bus clock e quindi viene trasmesso in modo scorretto.

⁵Gli handshake sono segnali che vengono utilizzati come segnali di controllo sulla ricezione o l'invio di dati.

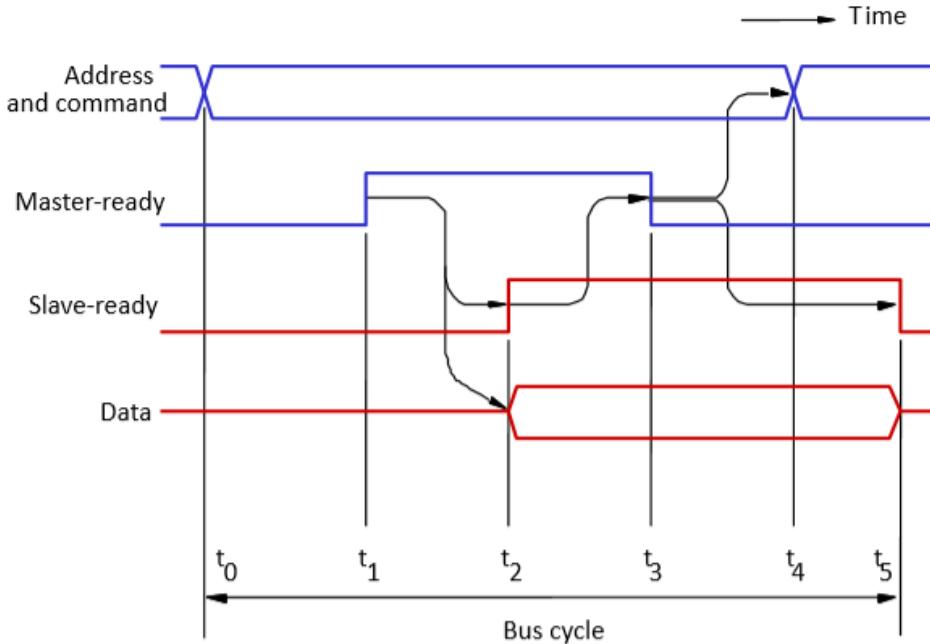


Figura 14.4: Schema rappresentativo del ciclo del bus asincrono

Dall'esempio qui riportato si osserva che in ogni transizione una parte riveste il ruolo del *master*, richiedendo delle operazioni, mentre l'altra parte fa da *slave*, eseguendo i comandi. Si nota quindi che esistono due linee di controllo dedicate (la master ready e la slave ready), che servono per trasmettere ai due componenti i segnali che indicano ad uno dei due che l'altro è pronto ed ha terminato la corretta trasmissione dei dati.

Riassumiamo ora i punti di forza del bus asincrono: è un meccanismo che permette di avere un sistema robusto relativamente agli eventuali ritardi di una periferica rispetto ad un'altra o al processore e che consente di comunicare con periferiche di tipo diverso.

Viceversa, osservando i fattori negativi, si deve indicare che risulta più lento del bus sincrono data la necessità di inviare e ricevere diversi segnali di controllo ed inoltre, aumentando la complessità del trasferimento I/O, aumenta anche la complessità della circuiteria che sta alla base.

In conclusione, dando uno sguardo alla realtà, si scopre che spesso si usano tecnologie ibride di bus, in cui c'è un segnale di clock ma che rimangono prevalentemente asincrone.

Caratteristica	Firewire (1394)	USB 2.0	PCI Express	Serial ATA	Serial Attached SCSI
Utilizzo previsto	Esterno	Esterno	Interno	Interno	Esterno
Numero dispositivi per canale	63	127	1	1	4
Larghezza base dei dati (numero di segnali)	4	2	2 per linea	4	4
Larghezza di banda di picco teorica	50 MB/s (Firewire 400) o 100 MB/s (Firewire 800)	0,2 MB/s (low speed), 1,5 MB/s (full speed), o 60 MB/s (high speed)	250 MB/s per linea (1x); le schede PCIe sono disponibili in versione 1x, 2x, 4x, 8x, 16x o 32x	300 MB/s	300 MB/s
Collegamento a caldo	Sì	Sì	Dipende dalle dimensioni	Sì	Sì
Lunghezza massima del bus (piste in rame)	4,5 metri	5 metri	0,5 metri	1 metro	8 metri
Nome dello standard	IEEE 1394, 1394b	Forum degli implementatori USB	SIG PCI	SATA-IO	Comitato T10

Figura 14.5: Principali tecnologie asincrone

Qui è riportato un esempio di implementazione reale di questo schema. L'esempio riguarda l'ormai vetusta architettura dell'x86. Notiamo che esiste una componente dedicata alla comunicazione con tutte le periferiche, detto hub dell'I/O; questo hub comunica poi in seguito con l'hub della memoria che si occupa di gestire le comunicazioni tra processore e memoria centrale, oltre che interpretare le comunicazioni che arrivano appunto dall'hub dell'I/O. Nei calcolatori moderni tutte queste componenti tendono ad essere inglobate nel processore stesso.

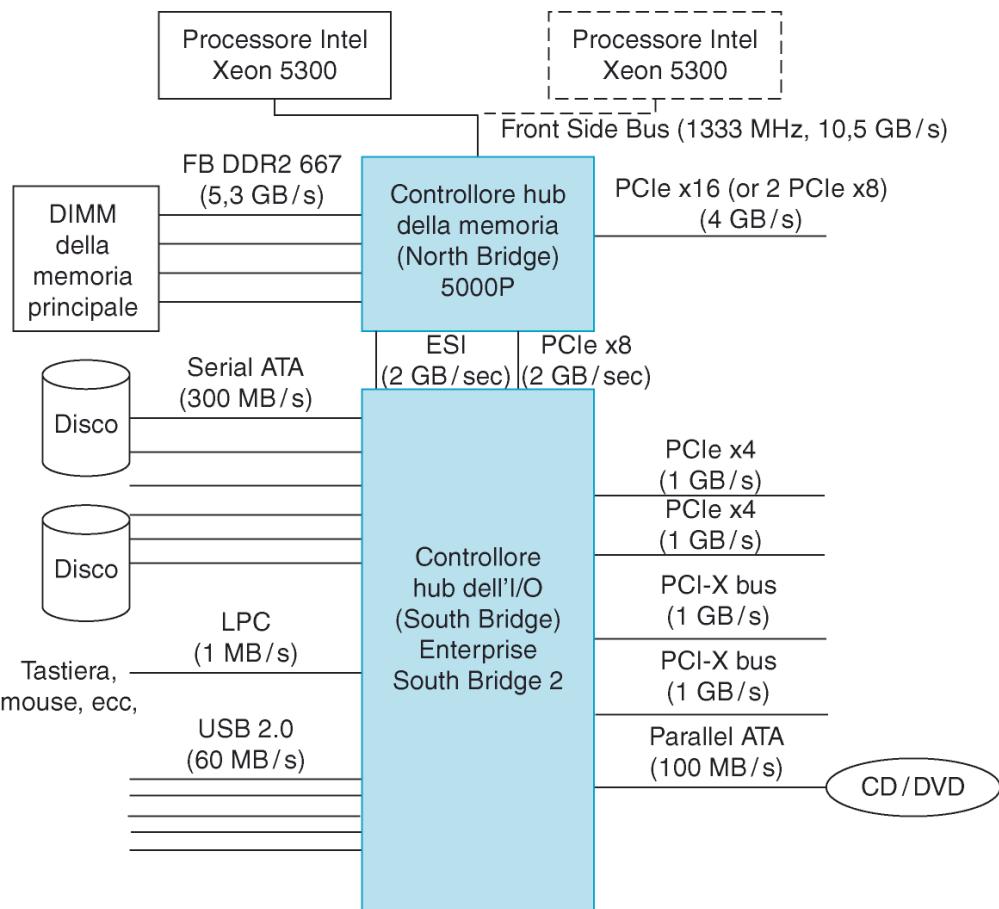


Figura 14.6: Schema collegamenti bus in x86

14.3 Gestione delle periferiche da parte del SO

Ora che abbiamo chiarito qual è il tramite fisico attraverso cui i sistemi operativi comunicano con le periferiche, rimangono ancora aperti i seguenti interrogativi: come trasformare una richiesta di I/O in un comando per la periferica? E come trasferire i dati? Di questi problemi si fa carico il sistema operativo (da qui in avanti SO), dato che è la parte di software dedicata alla gestione diretta del processore e delle sue risorse, quindi anche dei sistemi di I/O. Il SO deve fornire queste funzionalità per la gestione dell'I/O:

- garantire che un dato utente abbia accesso ai dispositivi di I/O cui ha i permessi per accedere;
- fornire dei comandi di alto livello per gestire le operazioni di basso livello, trasferimento dati nello specifico;

- gestire le interruzioni generate dai dispositivi di I/O (in maniera simile a quanto avviene con le eccezioni generate nei programmi);
- ripartire l'accesso a ciascun dispositivo in maniera equa tra i vari programmi che lo richiedono.

È chiaro quindi, guardando il terzo punto, che i trasferimenti dati hanno un impatto sulle funzionalità del SO e di conseguenza vengono eseguite in una particolare modalità del processore, la modalità kernel.

Questa modalità è utilizzata per garantire la sicurezza delle operazioni assicurando, tra l'altro, che le comunicazioni siano atomiche (cioè che non avvenga più di un'operazione sulla stessa area di memoria nello stesso tempo). Inoltre, è evidente che per implementare le funzionalità appena elencate, il SO deve poter inviare comandi alle periferiche, ricevere notifiche di corretta esecuzione dalle periferiche stesse e consentire trasferimenti diretti tra dispositivi e memoria. Nelle prossime sezioni sono discussi proprio i meccanismi che rendono tutto questo possibile.

14.3.1 Come impartire comandi ai dispositivi

Il sistema operativo impedisce comandi alle varie periferiche fornendo sulle relative linee di bus alcune "parole" di controllo attraverso due metodi possibili:

- scrivendo/leggendo in particolari locazioni di memoria (memory mapped I/O);
- tramite alcune istruzioni speciali dedicate all'I/O.

Per chiarire questo meccanismo conviene procedere tramite un esempio: memorizzando una particolare parola in una locazione di memoria associata al dispositivo il sistema di memoria ignora la scrittura, mentre il controllore di I/O intercetta l'indirizzo *particolare* e trasmette il dato al dispositivo sotto forma di comando.

Queste particolari locazioni di memoria sono inaccessibili ai programmi utente, solo il sistema operativo può operarvi: esso viene invocato tramite una chiamata di sistema, la quale fa commutare il processore in modalità kernel e rende quindi possibile la scrittura in tali locazioni. Inoltre la periferica stessa può usare queste locazioni per trasmettere dati o presegnalare il suo stato; ad esempio può richiedere la stampa di un carattere a terminale e, finita la stampa, un particolare bit di un registro di stato mappato in memoria verrà aggiornato.

Esistono principalmente due meccanismi differenti per il trasferimento dei dati, il polling e le interruzioni di programma. Descriviamo in prima battuta il più semplice dei due: il polling.

14.4 Polling

Detta anche modalità di *attesa attiva*, sostanzialmente consiste nell'inviare un comando di lettura/scrittura alla periferica e poi iniziare un ciclo di attesa monitorando il bit di stato per sapere quando il dato è pronto. È di seguito inserita un'immagine rappresentante il controllo del terminale nel simulatore SPIM.

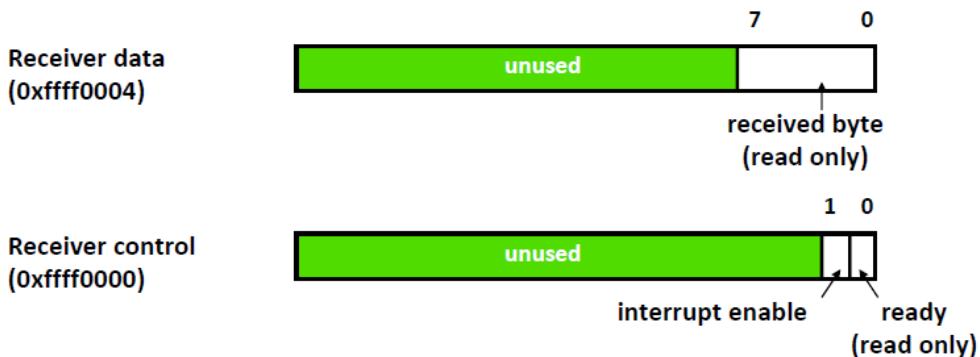


Figura 14.7: Controllo del terminale in SPIM (INPUT)

Le due parole di memoria 0xfffff0000 ed 0xfffff0004 sono riservate al SO. La prima è il *receiver control*, ovvero viene utilizzata per contenere il bit che segnala la presenza di un nuovo dato in ingresso (ready) e quello per stabilire se l'interrupt è abilitato (vedremo in seguito cosa significa). La seconda è il *receiver data* che conterrà gli effettivi dati forniti in input.

Ora che abbiamo chiarito l'utilizzo di queste due parole in memoria, possiamo osservare anche degli esempi di implementazione in MIPS di polling.

Di seguito è riportata l'implementazione in assembly MIPS di un ciclo di lettura input da tastiera e successivamente il ciclo di stampa di un dato.

Esempio 1: Input, lettura dalla tastiera in \$v0.

```

lui $t0, 0xfffff          # 0xfffff0000
Waitloop:
    lw $t1, 0($t0)        # control
    andi $t1,$t1,0x0001
    beq $t1,$zero, Waitloop
    lw $v0, 4($t0)         # data

```

Analizziamo ora il significato del codice: l'operazione `load upper immediate` carica nei bit più significativi del registro `t0` il valore `0xfffff`, così che esso contenga l'indirizzo del `Receiver control`. La `lw` carica in `$t1` il `receiver control`. Se è presente il bit ready significa che il dato è pronto e quindi viene caricato in memoria

(in \$v0 il carattere effettivamente premuto, che si trova in 0xfffff0004. Nel caso il bit ready non sia presente il **beq** farà ripetere il waitloop.

Esempio 2: Output, stampa del dato da \$a0.

```

lui $t0, 0xffff          # 0xfffff0000
Waitloop:
    lw $t1, 8($t0)        # control
    andi $t1,$t1,0x0001
    beq $t1,$zero, Waitloop
    sw $a0, 12($t0)        # data

```

Nel secondo esempio cambiano solamente le aree di memoria in cui si compie la verifica: l'area dedicata alla lettura da tastiera è infatti diversa da quella dedicata alla scrittura su video. Nonostante ciò, il procedimento è simmetrico a quello del primo waitloop osservato.

Osserviamo ora l'aspetto del costo in termini di risorse del meccanismo del polling con i seguenti esempi: consideriamo un processore a 500 Mhz e supponiamo che occorrono 400 cicli di clock per un'operazione di polling. Qual è il costo percentuale di questo meccanismo?

Esempio 1: Mouse Per non perdere movimenti da parte dell'utente occorre acquisire il dato 30 volte al secondo. Per stimare l'impatto del polling sul processore si deve calcolare il numero di clock al secondo spesi per il polling stesso, ovvero $30 \cdot 400 = 12000$ clocks/sec. Ora che abbiamo il numero di clock spesi per il polling ci basta calcolare a che percentuale di utilizzo del processore corrispondono, ricordando che 500Mhz significa $500 \cdot 10^6$ clocks/sec. Quindi complessivamente l'impatto del polling in percentuale è:

$$\frac{12 \cdot 10^3}{500 \cdot 10^6} = 0.002\%$$

È chiaro quindi che in questo caso l'impatto del polling è trascurabile; si noti tuttavia che questo overhead viene pagato sempre, sia che avvenga il trasferimento, sia che non avvenga.

Esempio 2: Hard disk I dati vengono trasferiti in blocchi di 16 byte ad una velocità di 8MB/s senza la possibilità di perdite.

Il numero di volte al secondo che occorre fare cicli di attesa per non perdere dati è:

$$\frac{8\text{MB/s}}{16\text{B}} = \frac{(8 \cdot 1024 \cdot 1024)\text{B/s}}{16\text{B}} = 529408 \text{ polls/sec}$$

Questo implica che il numero di cicli di clock al secondo dedicati al polling sono:

$$529408 \cdot 400 = 211763200 \text{ clocks/sec}$$

In percentuale quindi il peso di questo meccanismo è di $\frac{211763200}{500 \cdot 10^6} \approx 42\%$ il che è inaccettabile, anche perché, come nel primo esempio, questo prezzo in risorse viene pagato sempre.

14.4.1 Considerazioni finali sul polling

L'attesa attiva è un meccanismo che fa perdere tempo al processore dedicando cicli macchina a letture inutili, di conseguenza il polling può essere usato quando le operazioni di I/O avvengono con velocità di trasferimento predeterminata (ad esempio applicazioni di controllo) e comunque il processore ha poco altro da fare.

Sicuramente se i dati vengono trasferiti con elevati *bitrate*⁶ il ciclo di attesa attiva dura poco, in altri casi lo spreco derivante dal polling è inaccettabile e per questo motivo è stato inventato il sistema di I/O a interruzione di programma.

Osserviamo ora uno schema esemplificativo tratto dal simulatore SPIM per l'assembly MIPS.

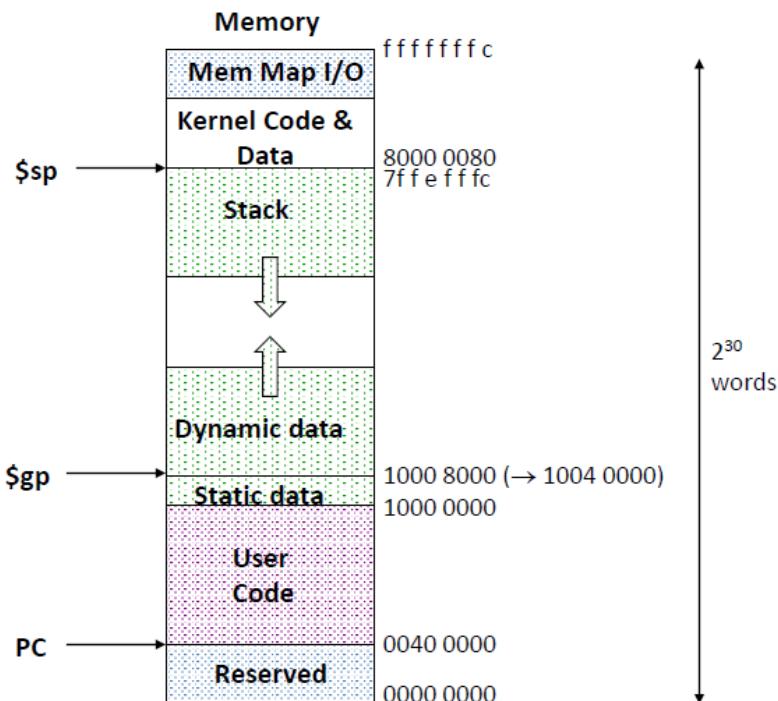


Figura 14.8: Organizzazione memoria SPIM (MIPS)

⁶Indica la quantità di dati che possono essere trasferiti su un canale di comunicazione in un dato intervallo di tempo.

La parte più in alto in questa rappresentazione della memoria è quella che contiene il kernel code e le parole riservate.

Il kernel code è la traduzione in linguaggio macchina del SO (che ricordiamo essere un programma esso stesso); all'interno di questa sezione si trovano inoltre le parole di memoria riservate all'I/O. È quindi ora chiaro cosa si intendeva quando si è detto che il processore vede solo virtualmente le periferiche, che fisicamente dal suo punto di vista altro non sono che dati.

Passiamo ora alla descrizione del secondo meccanismo di trasferimento dei dati: l'input ad interruzione di programma.

14.5 Input ad interruzione di programma

L'input ad interruzione di programma (conosciuto anche con il nome di *interrupt driven I/O*) funziona tramite il sollevamento di interruzioni.

Un'interruzione I/O è un segnale usato per indicare al processore che la periferica è pronta ad eseguire il trasferimento richiesto; occorre tuttavia un modo per segnalare al processore quale periferica richiede l'interruzione e per gestire il fatto che le interruzioni I/O sono sempre asincrone rispetto all'esecuzione delle istruzioni.

Non esistono particolari istruzioni Assembly per eseguire le interruzioni, queste possono presentarsi mentre una qualsiasi istruzione viene eseguita, ma si deve comunque dare modo di terminare l'esecuzione dell'istruzione corrente prima di passare alla gestione dell'interruzione. In tal senso il programmatore può decidere di posticipare l'esecuzione dell'interruzione a un momento più conveniente programmando sezioni non interrompibili nel codice del kernel; inoltre può classificare le interruzioni secondo grado di priorità.

Il maggior vantaggio che deriva dall'utilizzo di questa strategia è che non occorre interrompere l'esecuzione del programma se non quando il dato può essere effettivamente riferito in memoria.

Guardando invece il lato negativo, occorre un hardware speciale per permettere ai dispositivi di I/O di generare un'interruzione, rilevare l'interruzione, salvare lo stato del processore per eseguire una particolare routine di servizio, Interrupt Service Routine (ISR), e poi riprendere dal punto dove si era interrotto.

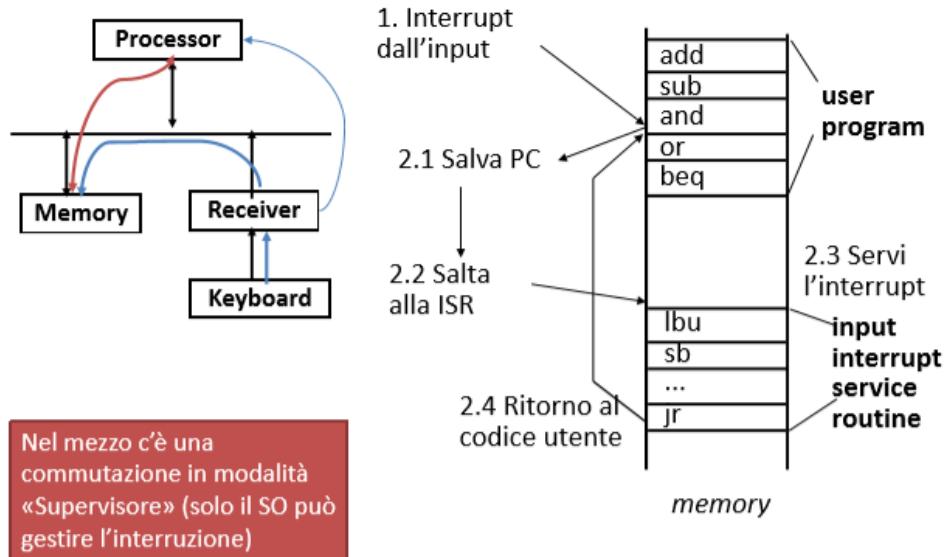


Figura 14.9: Meccanismo di input ad interruzione

È ora presentato un esempio di implementazione di gestione dell'I/O tramite interrupt in SPIM.

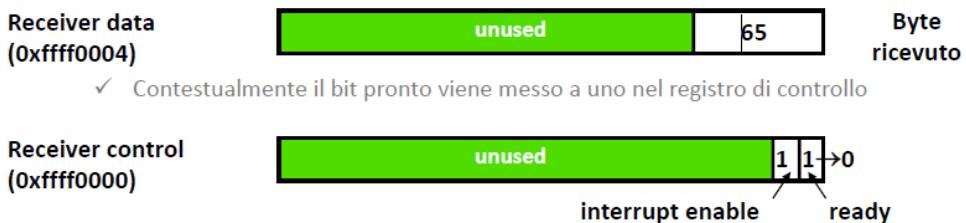


Figura 14.10: Esempio controllo terminale SPIM

In questo esempio si nota come vi sia ancora, come nel polling, una parola dedicata alla segnalazione della presenza di dati da scrivere, il receiver control (0xfffff0000); la parola successiva viene utilizzata per memorizzare effettivamente il dato disponibile, il receiver data (0xfffff0004).

La periferica indica, con un'interruzione (lanciata da una circuiteria di controllo del bit ready), di avere un nuovo carattere dalla tastiera nell'opportuno registro di ricezione.

Il processo utente viene interrotto trasferendo il controllo a una ISR che copia il dato in memoria utente; contemporaneamente alla lettura il bit ready viene resettato.

Nel dettaglio il bit *interrupt enable* serve per indicare se è abilitato il meccanismo di interrupt.

14.5.1 Considerazioni finali sull'interrupt driven I/O

Nonostante la complessità di questo meccanismo è facile osservare che in caso di trasferimenti di grandi moli di dati esso risulti molto più efficiente del polling, dimostriamolo riprendendo l'esempio dell'hard disk fatto in precedenza:

Esempio 2: Hard disk Supponiamo che un interrupt costi 500 cicli di clock, poiché è plausibile che costi di più del polling. Se le interruzioni venissero generate alla frequenza di polling avremmo che:

$$\frac{\text{Disk Interrupts}}{\text{sec}} = \frac{8\text{MB/s}}{16\text{B}} = 500 \cdot 10^3 \text{interrupts/sec}$$

e quindi

$$\frac{\text{Disk Polling Clocks}}{\text{sec}} = 500 \cdot 10^3 \cdot 500 = 250 \cdot 10^6 \text{clocks/sec}$$

Di conseguenza la percentuale di utilizzo del processore sarebbe $250 \cdot 10^6 / 500 \cdot 10^6 = 50\%$. Sembra che non ci sia guadagno, anzi.

Tuttavia se l'hard disk è attivo solo per il 5% del tempo, gli interrupt generati saranno il 5% e la spesa di processore sarà: $5\% \cdot 50\% = 2.5\%$.

Tutto questo proprio grazie al fatto che con il meccanismo di interrupt l'overhead si paga solo quando vengono effettivamente generate richieste.

14.6 Eccezioni

Le interrupts che abbiamo appena visto vanno inserite in una classe più grande di eventi detti *eccezioni*.

Un'eccezione consiste nel trasferimento del controllo del programma per l'avveramento di una condizione appunto eccezionale⁷. Per la gestione di questi eventi il sistema effettua delle azioni specifiche (come registrare il punto di interruzione e salvare lo stato) ed in seguito, a eccezione finita, riprendere dal punto immediatamente successivo al punto di interruzione.

⁷Da intendere qui come "fuori dalla norma".

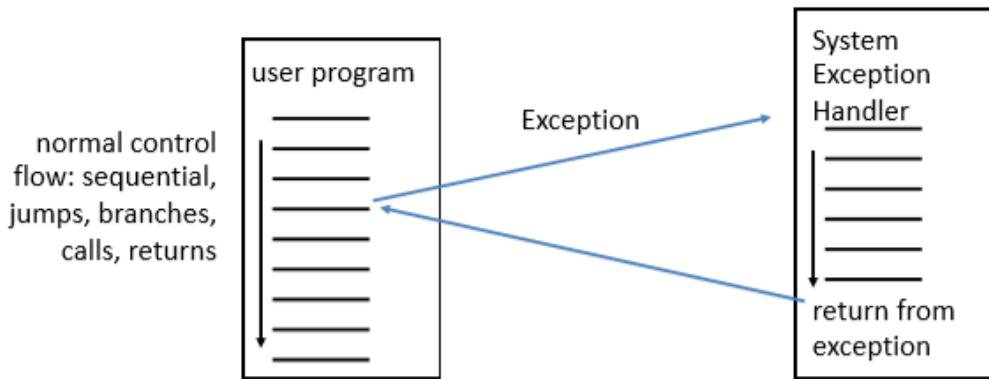


Figura 14.11: Schema processo delle eccezioni

Esistono due tipi di eccezioni:

- *interrupts*: sono causate da eventi esterni (I/O) e quindi sono asincrone; possono essere gestite nello spazio tra due istruzioni semplicemente sospendendo il programma e riprendendo, dopo la gestione, dal punto in cui era stato interrotto.
- *traps*: sono causate da eventi *interni* al programma come condizioni eccezionali (es. arithmetic overflow), errori (es. hardware malfunction) o fault (es. page fault). Esse sono sincrone all'esecuzione del programma. Vengono gestite da un *traphandler*; da notare che spesso è possibile riprovare ad eseguire l'istruzione che ha causato l'eccezione o abortire il programma.

14.6.1 Supporto del MIPS per la gestione di eccezioni

La componente che registra le informazioni necessarie alla gestione delle eccezioni in MIPS è il coprocessore 0, che utilizza dei registri a lui dedicati.

Quando si presenta un'eccezione in MIPS il registro Expected Program Counter (EPC) (registro 14) punta all'indirizzo successivo all'istruzione in esecuzione quando l'eccezione è avvenuta, mentre il registro di stato (registro 12) fa da maschera di abilitazione trap/interrupt.

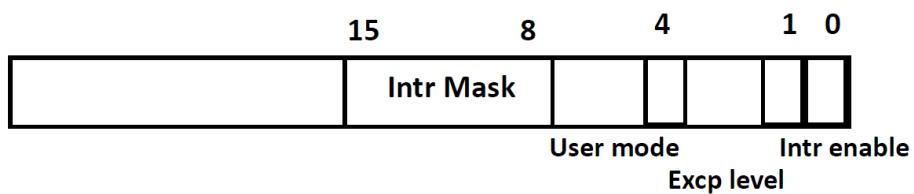


Figura 14.12: Registro maschera

In seguito una breve spiegazione dei campi di questo registro.

- *IntrMask*: 1 bit per ognuno dei 6 livelli di eccezioni hardware e 2 livelli software abilitabili.
- *User mode*: 0 se quando l'eccezione è avvenuta il sistema era in modo supervisore, 1 se era in modo utente.
- *Excp level*: settato a 1 (eccezioni disabilitate) mentre si sta gestendo un'eccezione, tipicamente resettata dopo che l'ISR ha terminato.
- *Intr enable*: 1 se si sta utilizzando il meccanismo delle interrupt, 0 altrimenti.

Il registro *BadVAddr* (registro 8) contiene l'indirizzo di memoria che ha causato un errore di memoria. Infine il registro *cause* (registro 13) contiene il tipo dell'eccezione ed i bit pendenti.

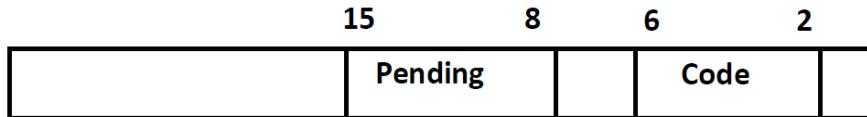


Figura 14.13: Registro cause

In particolare sono importanti i campi:

- *code*: codifica i motivi dell'eccezione;
- *pending*: gestisce i casi in cui avvengono una o più eccezioni mentre venivano risolte altre eccezioni.

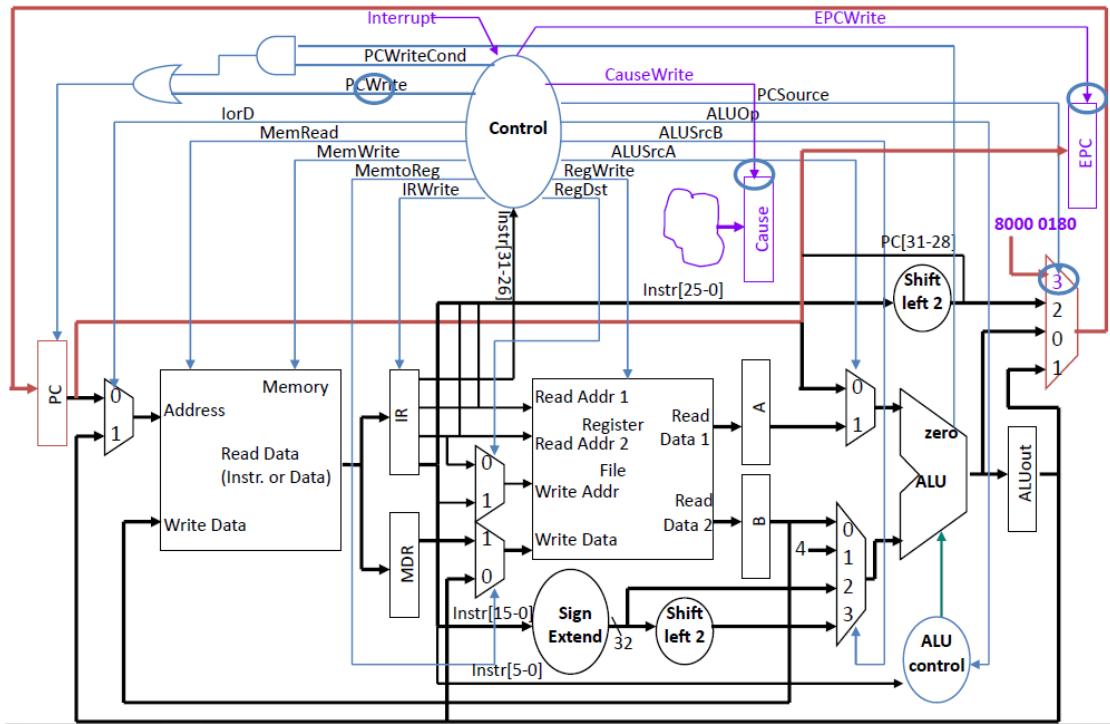
14.6.2 Modifiche al processore per gestire le eccezioni

Per implementare il complesso meccanismo di gestione delle eccezioni il processore deve essere munito di queste nuove componenti:

- segnali di controllo per scrivere EPC (*EPCWrite*), *Cause* (*CauseWrite*) e *Status*;
- hardware per registrare il tipo di interruzione in *Cause*;
- modifiche alla macchina a stati in modo che:
 - l'indirizzo del gestore dell'interruzione 0x80000180 possa essere caricato in PC (altro ramo nel multiplexer);

- sia salvato l’indirizzo della prossima istruzione da eseguire a ISR terminato in EPC.

Ed ecco infine un’immagine del processore con datapath modificato per supportare la gestione delle eccezioni.



Appendice A

Istruzioni e registri MIPS

Istruzioni di movimento dati dell'Assembly MIPS

Istruzione	Sintassi	Tipo	Funzione
lw	lw \$d, S(\$b)	I	load word: carica in \$d la parola memorizzata a b+S (\$b = base, S = spiazzamento)
lh	lh \$d, S(\$b)	I	load half word
lhu	lhu \$d, S(\$b)	I	load half word unsigned
lb	lb \$d, S(\$b)	I	load byte
sw	sw \$r, S(\$b)	I	store word: salva in b+S la parola contenuta \$r
sh	sh \$r, S(\$b)	I	store half word
sb	sb \$r, S(\$b)	I	store byte
lui	lui \$d, C	I	load upper register immediato: carica C nei 16 bit più significativi di \$d
mfhi	mfhi \$d	R	muove HI in \$d
mflo	mflo \$d	R	muove LO in \$d

Istruzioni di salto dell'Assembly MIPS

Istruzione	Sintassi	Tipo	Funzione
j	j C	J	jump: salta a C·4
jal	jal C	J	jump and link: salta a C·4 e salva il valore di PC in \$ra
jr	jr \$r	R	jump register: salta all'indirizzo contenuto in \$r
beq	beq \$s, \$t, C	I	branch equal: se s e t sono uguali, salta a PC+4+C·4
bne	bne \$s, \$t, C	I	branch not equal: se s e t sono diversi, salta a PC+4+C·4

Istruzioni di supporto ai salti condizionali dell'Assembly MIPS

Istruzione	Sintassi	Tipo	Funzione
slt	slt \$d, \$s, \$t	R	set if less than: setta \$d a 1 se s < t
sltu	sltu \$d, \$s, \$t	R	set if less than unsigned: setta \$d a 1 se s < t (confronto tra naturali)
slti	slti \$d, \$s, C	I	set if less than immediate: setta \$d a 1 se s < C (valore immediato)

Istruzioni aritmetico-logiche dell'Assembly MIPS

Istruzione	Sintassi	Tipo	Funzione
add	add \$d, \$s, \$t	R	somma, generando eccezione in caso di overflow
sub	sub \$d, \$s, \$t	R	sottrazione, generando eccezione se overflow
and	and \$d, \$s, \$t	R	and logico bit a bit
or	or \$d, \$s, \$t	R	or logico bit a bit
xor	xor \$d, \$s, \$t	R	or esclusivo bit a bit
nor	nor \$d, \$s, \$t	R	or logico bit a bit negato
addu	addu \$d, \$s, \$t	R	somma, ignorando overflow
subu	subu \$d, \$s, \$t	R	sottrazione ignorando overflow (no eccezioni)
addi	addi \$d, \$s, I	I	somma con valore immediato (eccezione se overflow)
andi	andi \$d, \$s, I	I	and logico bit a bit con valore immediato
ori	ori \$d, \$s, I	I	or logico bit a bit con valore immediato
addiu	addiu \$d, \$s, I	I	somma con valore immediato ignorando overflow (no eccezioni)
sll	sll \$d, \$s, i	R	shift logico a sinistra di i bit
srl	srl \$d, \$s, i	R	shift logico a destra di i bit
sra	sra \$d, \$s, i	R	shift aritmetico a destra di i bit
sllv	sllv \$d, \$s, \$t	R	shift logico a sinistra di \$t bit
srlv	srlv \$d, \$s, \$t	R	shift logico a destra di \$t bit
srav	srav \$d, \$s, \$t	R	shift aritmetico a destra di \$t bit
mult	mult \$s, \$t	R	moltiplicazione (risultato in registri speciali HI e LO)
multu	multu \$s, \$t	R	moltiplicazione tra unsigned (risultato in HI e LO)
div	div \$s, \$t	R	divisione (quoziente in LO e resto in HI)
divu	divu \$s, \$t	R	divisione tra unsigned (quoziente in LO e resto in HI)

Macro spesso implementate nell'Assembly MIPS

Macro	Sintassi	Espansa a	Funzione
Move	move \$rt, \$rs	add \$rt, \$rs, \$zero	$rt = rs$
Clear	clear \$rt	add \$rt, \$zero, \$zero	$rt = 0$
Not	not \$rt, \$rs	nor \$rt, \$rs, \$zero	$rt \neq rs$
Load Adress	la \$rd, Addr	lui \$rd, Addr[31:16] ori \$rd, \$rd, Addr[15:0]	$rd = Address$
Load Immediate	li \$rd, v	lui \$rd, v[31:16] ori \$rd, \$rd, v[15:0]	$rd = 32$ bit value
Branch	b Label	beq \$zero, \$zero, Label	PC = Label
Branch if greater	bgt \$rs, \$rt, Label	slt \$at, \$rt, \$rs bne \$at, \$zero, Label	if $rs > rt$ PC = Label
Branch if less	blt \$rs, \$rt, Label	slt \$at, \$rs, \$rt bne \$at, \$zero, Label	if $rs < rt$ PC = Label
Branch if greater or equal	bge \$rs, \$rt, Label	slt \$at, \$rs, \$rt beq \$at, \$zero, Label	if $rs \geq rt$ PC = Label
Branch if less or equal	ble \$rs, \$rt, Label	slt \$at, \$rt, \$rs beq \$at, \$zero, Label	if $rs \leq rt$ PC = Label
Multiply 32 bit	mul \$d, \$s, \$t	mult \$s, \$t; mflo \$d	$d = s \cdot t$
Division	div \$d, \$s, \$t	div \$s, \$t; mflo \$d	$d = s / t$
Reminder	rem \$d, \$s, \$t	div \$s, \$t; mfhi \$d	$d = s \bmod t$

Elenco completo dei registri dell'Assembly MIPS

Registri referenziabili

Nome	Numero	Utilizzo	Preservare
\$zero	0	Valore costante 0	Costante
\$at	1	Assembler-linker per pseudo-istruzioni e macro	No
\$v0-\$v1	2-3	Valori di ritorno funzioni e valutazione espressioni	No
\$a0-\$a3	4-7	Argomenti	No
\$t0-\$t7	8-15	Temporanei	No
\$s0-\$s7	16-23	Salvati	Sì
\$t8-\$t9	24-25	Altri temporanei	No
\$k0-\$k1	26-27	Dedicati al kernel, da NON USARE	No
\$gp	28	Global pointer	Sì
\$sp	29	Stack pointer	Sì
\$fp	30	Frame pointer	Sì
\$ra	31	Return address	Sì

Registri non referenziabili

Nome	Numero	Utilizzo
\$pc	/	Program Counter (PC)
\$hi	/	Risultato di moltiplicazione e divisione (32 bit più significativi)
\$lo	/	Risultato di moltiplicazione e divisione (32 bit meno significativi)

Appendice B

Come compilare Assembly Intel

B.1 Prerequisiti

Il sistema operativo che utilizzeremo è una qualsiasi distribuzione Linux. A seguire i software che andremo a utilizzare:

- *Gnu Compiler Collection (GCC)*: il compilatore C GNU;
- *ld*: il linker GNU;
- *nasm*: un assemblatore Intel x86.

B.1.1 Come utilizzare GCC

Per compilare il file C "main.c" attraverso GCC nell'eseguibile "main":

```
gcc main.c -o main
```

mentre per compilare un file C in assembly AT&T attraverso GCC:

```
gcc -S main.c -o main.s
```

B.1.2 Come utilizzare nasm e ld

Per compilare un file Assembly Intel "pluto.asm" nel file oggetto "pluto.o":

```
nasm -felf64 -o pluto.o pluto.asm
```

e successivamente per linkare il file oggetto e ottenere l'eseguibile "pluto":

```
ld -m elf_i386 -o pluto pluto.o
```

B.2 Un po' di esempi

Esistono principalmente due sezioni: `.data` che racchiude eventuali dati da salvare in memoria e `.text` che contiene il codice Assembly. In particolare, in quest'ultima sezione bisogna specificare un'etichetta come `global`: l'etichetta che andiamo a specificare indicherà l'inizio del programma.

B.2.1 Hello World

```

1 SECTION .data
2 msg db "Hello World", 0Ah # 0A permette di andare a capo riga
3
4 SECTION .text
5 global _start
6
7 _start:
8     mov edx, 13    # lunghezza stringa
9     mov ecx, msg  # puntatore alla stringa
10    mov ebx, 1     # seleziona come output lo std output
11    mov eax, 4     # system call per stampare
12    int 80h
13
14    mov ebx, 0     # return 0
15    mov eax, 1     # sys call close
16    int 80h

```

Da notare che se non si inserisce l'ultima sys call (`return 0`) verrà prodotto un errore di segmentation fault.

B.2.2 Stampa di una stringa di lunghezza variabile

```

1 SECTION .data
2 msg db "Hello World", 0Ah, 0h # 0h terminatore di stringa
3
4 SECTION .text
5 global _start
6
7 _start:
8     # esegue un ciclo while
9     mov ebx, msg
10    mov eax, ebx

```

```

11 nextchar:
12     cmp byte[eax], 0
13     jz finished
14     inc eax
15     jmp nextchar
16 finished:
17     sub eax, ebx
18     # stessa cosa di prima, semplicemente la lunghezza è variabile
19     → in eax
20     mov edx, eax
21     mov ecx, msg
22     mov ebx, 1
23     mov eax, 4
24     int 80h
25
26     mov ebx, 0
27     mov eax, 1
28     int 80h

```

B.2.3 Esempio di subroutine

```

1 SECTION .data
2 msg db "Scritta", 0Ah, 0h
3
4 SECTION .text
5 global _start
6
7 _start:
8     mov eax, msg
9     call strlen
10
11    mov edx, eax
12    mov ecx, msg
13    mov ebx, 1
14    mov eax, 4
15    int 80h
16    mov ebx, 0
17    mov eax, 1
18    int 80h
19 strlen:
20     push ebx

```

```
21      mov ebx, ea
22  nextchar:
23      cmp byte[eax], 0
24      jz finished
25      inc eax
26      jmp nextchar
27 finished:
28      sub eax, ebx
29      pop ebx
30      ret
```

Appendice C

Ancora sui programmi Assembly

Si consideri quest'appendice come un'estensione del capitolo 9; qui illustreremo tutti gli esempi presenti nella dispensa del professor Abeni, dimodoché anche il lettore più curioso possa trovare pane per i suoi denti. Enjoy.

C.1 Ordinamento di array

Mostriamo ora le diverse traduzioni della seguente funzione *insert sort* scritta in C:

```
1 void sposta(int v[], int i){
2     int j;
3     int appoggio;
4
5     appoggio = v[i];
6     j = i - 1;
7
8     while((j >= 0) && (v[j] > appoggio)){
9         v[j + 1] = v[j];
10        j = j - 1;
11    }
12
13    v[j + 1]
14}
15
16 void ordina(int v[], int n){
17     int i = 1;
18
19     while (i < n){
```

```

20         sposta(v, i);
21         i = i + 1;
22     }
23 }
```

Implementazione MIPS

Prima versione, "fatta a mano":

```

1 sposta:
2         # a0 = v, a1 = i
3         # t0 = j * 4, t1 = appoggio
4         sll $t0, $a1, 2
5         add $t2, $a0, $t0    # t2 = &v[i]
6         lw $t1, 0($t2)      # appoggio = v[i]
7         addi $t0, $t0, -4
8
9 ciclo:
10        slt $t3, $t0, $zero   # t3 = 1 se j < 0
11        bne $t3, $zero, out   # salta a out se j < 0
12        add $a2, $a0, $t0      # t2 = &v[j]
13        lw $t4, 0($t2)      # t4 = v[j]
14        slt $t3, $t1, $t4      # t3=1 se t1<t4 (appoggio < v[j])
15        beq $t3, $zero, out      # esce se appoggio >= v[j]
16        addi $t5, $t0, 4      # t5 = (j + 1) * 4
17        add $t2, $a0, $t5      # t2 = &v[j + 1]
18        sw $t4, 0($t2)
19        addi $t0, $t0, -4
20        j ciclo
21
22 out:
23        addi $t5, $t0, 4
24        add $t2, $a0, $t5      # t2 = &v[j + 1]
25        sw $t1, 0($t2)
26        jr $ra
27
28 ordina:
29         # a0 = v, a1 = n
30         # s0 = i
31         addi $sp, $sp, -12
32         sw $s0, 0($sp)
```

```

33      sw $ra, 4($sp)
34      sw $a1, 8($sp)
35
36      addi $s0, $zero, 1  # i = 1
37
38  loop_ordina:
39      slt $t0, $s0, $a1    # t0 = 1 se i < n
40      beq $t0, $zero, out_ordina
41      add $a1, $s0, $zero
42      jal sposta
43      lw $a1, 8($sp)
44      addi $s0, $s0, 1
45      j loop_ordina
46
47  out_ordina:
48      lw $s0, 0($sp)
49      lw $ra, 4($sp)
50      addi $sp, $sp, 12
51      jr $ra

```

Ecco invece l'implementazione, più complessa, che ci presenta il nostro fido `gcc`:

```

1  sposta:
2      sll $v0, $a1, 2
3      addu $v0, $a0, $v0
4      lw $t0, 0($v0)
5      addiu $v0, $a1, -1
6      blitz $v0, L2
7      move $a2, $v0
8      sll $v1, $v0, 2
9      addu $v1, $a0, $v1
10     lw $v1, 0($v1)
11     slt $a3, $t0, $v1
12     beq $a3, $zero, L2
13     sll $a1, $a1, 2
14     addu $a1, $a0, $a1
15     li $t1 , -1    # 0xffffffffffff
16
17  L3:
18      addiu $a2, $a2, 1

```

```
19      sll $a2, $a2, 2
20      addu $a2, $a0, $a2
21      sw $v1, 0($a2)
22      addiu $v0, $v0, -1
23      beq $v0, $t1, L2
24      move $a2, $v0
25      addiu $a1, $a1, -4
26      lw $v1, 4($a1)
27      slt $a3, $t0, $v1
28      bne $a3, $zero, L3
29
30  L2:
31      addiu $v0, $v0, 1
32      sll $v0, $v0, 2
33      addu $a0, $a0, $v0
34      sw $t0, 0($a0)
35      jr $ra
36
37  ordina:
38      addiu $sp, $sp, -16
39      sw $ra, 12($sp)
40      sw $s2, 8($sp)
41      sw $s1, 4($sp)
42      sw $s0, 0($sp)
43      move $s1, $a1
44      slt $v0, $a1, 2
45      bne $v0, $zero, L5
46      move $s2, $a0
47      li $s0, 1      # 0x1
48
49  L7:
50      move $a0, $s2
51      move $a1, $s0
52      jal sposta
53      addiu $s0, $s0, 1
54      bne $s0, $s1, L7
55
56  L5:
57      lw $ra, 12($sp)
58      lw $s2, 8($sp)
```

```

59      lw $s1, 4($sp)
60      lw $s0, 0($sp)
61      addiu $sp, $sp, 16
62      jr $ra

```

Implementazione x86

Ecco una possibile implementazione Intel, sempre artigianale:

```

1 sposta:
2         # rdi = v, rsi = i
3         # rax = j , r10d = appoggio
4         movq %rsi, %rax
5         movl (%rdi, %rax, 4), %r10d      # appoggio = v[i]
6         dec %rax
7
8 ciclo:
9         cmpq $0, %rax                  # confronta 0 e rax
10        jl out                      # esci se j < 0
11        movl (%rdi, %rax, 4), %r11d  # metti v[j] in %r11
12        cmpl %r10d, %r11d          # confronta v[j] e appoggio
13        jle out                      # se v[j] < appoggio, esci
14        movl %r11d, 4(%rdi, %rax 4)
15        dec %rax
16        jmp ciclo
17
18 out:
19        movl %r10d, 4(%rdi, %rax, 4)
20        ret
21
22 ordina:
23         # rdi = v, rsi = n
24         # rbx = i
25         pushq %rbx
26         movq $1, %rbx
27
28 loop_ordina:
29         cmp %rbx, %rsi
30         jle out_ordina
31         pushq %rsi
32         movq %rbx, %rsi

```

```

33    call sposta
34    popq %rsi
35    inc %rbx
36    jmp loop_ordina
37
38 out_ordina:
39    popq %rbx
40    ret

```

Implementazione di gcc:

```

1  sposta:
2      movslq %esi, %rax
3      leaq 0(%rax, 4 ), %r8
4      movl (%rdi, %rax, 4), %ecx
5      subl $1, %esi
6      js L2
7      movslq %esi, %rdx
8      movl -4(%rdi, %r8), %eax
9      cmpl %eax, %ecx
10     jge L2
11
12 L4:
13     movl %eax, 4(%rdi, %rdx, 4)
14     subl $1, %esi
15     cmpl $-1, %esi
16     je L2
17     movslq %esi, %rdx
18     movl (%rdi, %rdx, 4), %eax
19     cmpl %eax, %ecx
20     jl L4
21
22 L2:
23     movslq %esi, %rsi
24     movl %ecx, 4(%rdi, %rsi, 4)
25     ret
26
27 ordina:
28     cmpl $1, %esi
29     jle L12
30     pushq %r12

```

```

31      pushq %rbp
32      pushq %rbx
33      movl %esi, %ebp
34      movq %rdi, %r12
35      movl $1, %ebx
36
37  L8:
38      movl %ebx, %esi
39      movq %r12, %rdi
40      call sposta
41      addl $1, %ebx
42      cmpl %ebx, %ebp
43      jne L8
44      popq %rbx
45      popq %rbp
46      popq %r12
47
48  L12:
49      ret

```

Implementazione ARM

```

1  sposta:
2      mov r2, r1, asl #2
3      add r3 , r0, r2
4      ldr ip, [r0, r1, asl #2]
5      subs r1, r1, #1
6      bmi L2
7      ldr r2, [r3, #-4]
8      cmp ip, r2
9      bge L2
10
11 L3:
12      str r2, [r3], #-4
13      sub r1, r1, #1
14      cmn r1, #1
15      beq L2
16      ldr r2, [r3, #-4]
17      cmp ip, r2
18      blt L3
19

```

```

20 L2:
21     add r1, r1, #1
22     str ip, [r0, r1, asl #2]
23     bx lr
24
25 ordina:
26     cmp r1, #1
27     bxle lr
28     stmfd sp!, {r4, r5, r6, lr}
29     mov r5, r1
30     mov r6, r0
31     mov r4, #1
32
33 L8:
34     mov r1, r4
35     mov r0, r6
36     bl sposta
37     add r4, r4, #1
38     cmp r5, r4
39     bne L8
40     ldmfd sp!, {r4, r5, r6, pc}

```

C.2 Funzione 2^n

Via iterativa

```

1 int potenza_due(int n){
2     if (n == 0){
3         return 1;
4     } else {
5         int ret = 1;
6         int i;
7         for (i=0; i<n; i++){
8             ret = ret * 2;
9         }
10
11     return ret;
12 }
13 }
```

Implementazione MIPS

```

1  potenza_due:
2      beq $a0, $zero, L4
3      blez $a0, L5
4      move $v1, $zero
5      li $v0, 1 # 0x1
6
7  L3:
8      sll $v0, $v0, 1
9      addiu $v1, $v1, 1
10     bne $v1, $a0, L3
11     jr $ra
12
13 L4:
14    li $v0, 1 # 0x1
15    jr $ra
16
17 L5:
18    li $v0, 1 # 0x1
19    jr $ra

```

Implementazione x86

```

1  potenza_due:
2      testl %edi, %edi
3      jle L4
4      movl $0, %edx
5      movl $1, %eax
6
7  L3:
8      addl %eax, %eax
9      addl $1, %edx
10     cmpl %edx, %edi
11     jne L3
12     ret
13
14 L4:
15     movl $1, %eax
16     ret

```

Implementazione ARM

```

1  potenza_due:
2      subs r2, r0, #0
3      ble L4
4      mov r3, #0
5      mov r0, #1
6  L3:
7      mov r0, r0, asl #1
8      add r3, r3, #1
9      cmp r2, r3
10     bne L3
11     bx lr
12 L4:
13     mov r0, #1
14     bx lr

```

Via ricorsiva

```

1 int potenza_due (int n){
2     if (n < 1){
3         return 1;
4     } else {
5         return 2 * potenza_due(n - 1);
6     }
7 }

```

Implementazione MIPS

```

1  potenza_due:
2      blez $a0, L3
3      addiu $sp, $sp, -8
4      sw $ra, 4($sp)
5      addiu $a0, $a0, -1
6      jal potenza_due
7      sll $v0, $v0, 1
8      j L2
9
10 L3:
11     li $v0, 1    # 0x1
12     jr $ra

```

```

13
14 L2:
15     lw $ra, 4($sp)
16     addiu $sp, $sp, 8
17     jr $ra

```

Implementazione x86

```

1 potenza_due:
2     movl $1, %eax
3     testl %edi, %edi
4     jle L6
5     subq $8, %rsp
6     subl $1, %edi
7     call potenza_due
8     addl %eax, %eax
9     addq $8, %rsp
10
11 L6:
12     ret

```

Implementazione ARM

```

1 potenza_due:
2     cmp r0, #0
3     ble L3
4     stmfd sp!, {r4, lr}
5     sub r0, r0, #1
6     bl potenza_due
7     mov r0, r0, asl #1
8     ldmfd sp!, {r4, pc}
9
10 L3:
11     mov r0, #1
12     bx lr

```

C.3 Successione di Fibonacci

Presentiamo ora l'implementazione di una funzione che calcola l' n -esimo numero di Fibonacci:

```

1 int fibonacci(int n){
2     if (n < 2){
3         return 1;
4     } else {
5         return 3 * fibonacci(n - 1) - fibonacci(n - 2);
6     }
7 }
```

Implementazione MIPS

```

1 fibonacci:
2     addiu $sp, $sp, -16
3     sw $ra, 12($sp)
4     sw $s1, 8($sp)
5     sw $s0, 4($sp)
6     move $s0, $a0
7     slt $v0, $a0, 2
8     bne $v0, $zero, L3
9     addiu $a0, $a0, -1
10    jal fibonacci
11    move $s1, $v0
12    addiu $a0, $s0, -2
13    jal fibonacci
14    sll $v1, $s1, 1
15    addu $s1, $v1, $s1
16    subu $v0, $s1, $v0
17    j L2
18 L3:
19    li $v0, 1      # 0x1
20 L2 :
21    lw $ra, 12($sp)
22    lw $s1, 8($sp)
23    lw $s0, 4($sp)
24    addiu $sp, $sp, 16
25    jr $ra
```

Implementazione x86

```

1 fibonacci:
2     movl $1, %eax
3     cmpl $1, %edi
```

```

4      jle L6
5      pushq %rbp
6      pushq %rbx
7      subq $8, %rsp
8      movl %edi, %ebx
9      leal -1(%rdi), %edi
10     call fibonacci
11     movl %eax, %ebp
12     leal -2(%rbx), %edi
13     call fibonacci
14     leal 0(%rbp, %rbp, 2), %edx
15     subl %eax, %edx
16     movl %edx, %eax
17     addq $8, %rsp
18     popq %rbx
19     popq %rbp
20 L6:
21     ret

```

Implementazione ARM

```

1 fibonacci:
2     cmp r0, #1
3     ble L3
4     stmfd sp!, {r4, r5, r6, lr}
5     mov r5, r0
6     sub r0, r0, #1
7     bl fibonacci
8     mov r4, r0
9     sub r0, r5, #2
10    bl fibonacci
11    add r4, r4, r4, lsl #1
12    rsb r0, r0, r4
13    ldmfd sp!, {r4, r5, r6, pc}
14 L3:
15    mov r0, #1
16    bx lr

```

C.4 Serie tail recursive

Vediamo ora una funzione che calcola i primi n numeri naturali, in versione tail recursive:

```

1 int sum(int n, int acc){
2     if (n > 0){
3         return sum(n - 1, acc + n);
4     } else {
5         return acc;
6     }
7 }
```

Senza ottimizzazione di chiamate in coda

Vediamone inizialmente una traduzione fatta senza l'eliminazione delle chiamate ricorsive:

Implementazione MIPS

```

1 sum:
2     move $v1, $a0
3     move $v0, $a1
4     blez $a0, L5
5     addiu $sp, $sp, -8
6     sw $ra, 4($sp)
7     addiu $a0, $a0, -1
8     addu $a1, $a1 , $v1
9     jal sum
10    lw $ra, 4($sp)
11    addiu $sp, $sp,8
12 L5:
13    jr $ra
```

Implementazione x86

```

1 sum :
2     movl %esi, %eax
3     testl %edi, %edi
4     jle L6
5     subq $8, %rsp
6     addl %edi, %esi
```

```

7      subl $1, %edi
8      call sum
9      addq $8, %rsp
10     L6:
11     ret

```

Implementazione ARM

```

1  sum:
2      cmp r0, #0
3      ble L3
4      stmfd sp!, {r4, lr}
5      add r1, r0, r1
6      sub r0, r0, #1
7      bl sum
8      ldmfd sp!, {r4, pc}
9  L3:
10     mov r0, r1
11     bx lr

```

Chiamate in coda ottimizzate

Proponiamo ora la traduzione che `gcc` esegue attivando il comando `-foptimize-sibling-calls`, la cui funzione è ottimizzare le chiamate in coda:

Implementazione MIPS

```

1  sum:
2      move $v0, $a1
3      blez $a0, L2
4  L4:
5      addu $v0, $v0, $a0
6      addiu $a0, $a0, -1
7      bne $a0, $zero, L4
8  L2:
9      jr $ra

```

Implementazione x86

```

1  sum:
2      movl %esi, %eax
3      testl %edi, %edi

```

```
4      jle L2
5  L4:
6      addl %edi, %eax
7      subl $1, %edi
8      jne L4
9  L2:
10     ret
```

Implementazione ARM

```
1  sum:
2      cmp r0, #0
3      ble L2
4  L4:
5      sub r3, r0, #1
6      add r1, r1, r0
7      mov r0, r3
8      cmp r3, #0
9      bne L4
10 L2:
11     mov r0, r1
12     bx lr
```