Dispense di Architettura dei Calcolatori

In queste note vengono presentati gli elementi principali di un calcolatore. Per la sua maggior semplicità di presentazione è stata presa come riferimento un'architettura RISC (*Reduced Instruction Set Computer*), quella del processore MIPS[1], la cui descrizione più dettagliata può essere trovata nel seguente testo:

> David A. Patterson, John L. Hennessy,
> *Computer Organization and Design: the Hardware/Software Interface*,
> Morgan Kaufmann, San Mateo, California, 1994.

La descrizione del processore parte con la presentazione di un insieme semplificato delle istruzioni e prosegue poi con la realizzazione di un semplice modello dello hardware necessario per l'esecuzione di queste istruzioni. Il modello del processore così ottenuto viene poi modificato in modo da adattarlo alla esecuzione delle istruzioni in pipeline. Vengono infine presentate le caratteristiche principali delle memorie cache, necessarie per realizzare un sistema di memoria in grado di soddisfare le esigenze dei processori più recenti.

[1] La sigla MIPS viene spesso utilizzata anche come acronimo di *Mega Instruction Per Second* ovvero *Milioni di Istruzioni Per Secondo* che rappresenta un indice delle prestazioni dei calcolatori molto utilizzato negli anni '80, specialmente nel campo commerciale. In queste dispense il termine MIPS™ fa sempre riferimento al processore prodotto dalla MIPS Technology, Inc.

1. L'architettura di riferimento dei calcolatori

Come illustrato in figura 1, gli elementi principali di un calcolatore sono l'unità centrale di elaborazioe (*Central Processing Unit — CPU*), la memoria di lavoro (detta anche memoria centrale) e il bus di sistema che provvede a collegarle. Al bus vengono inoltre collegate le interfacce per tutti i dispositivi di ingresso/uscita (*Input/Output — I/O*) necessari per collegare il calcolatore al mondo esterno, come, per esempio, il terminale (composto da video e tastiera) e la memoria di massa (di solito si tratta di dischi magnetici), entrambi riportati nella figura 1.

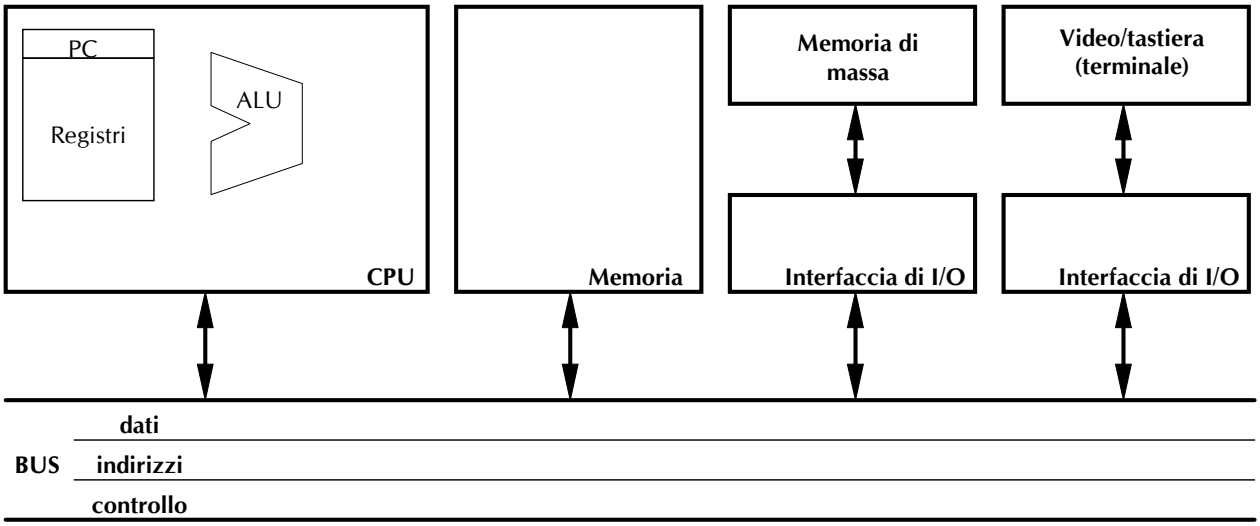


Figura 1 Architettura base di un calcolatore, che viene spesso presentata come architettura di Von Neumann dal nome del progettista che per primo la descrisse. Elementi fondamentali sono la CPU, la memoria e il bus di sistema. Il collegamento con il mondo esterno è assicurato da periferiche di I/O quali il terminale, le stampanti, …, ciascuna collegata al bus di sistema attraverso una opportuna interfaccia di I/O.

Consideriamo dapprima la CPU. Questa unità provvede ad eseguire le istruzioni che compongono i diversi programmi elaborati dal calcolatore. Le istruzioni, insieme ai dati a cui queste istruzioni fanno riferimento, si trovano nella memoria centrale e vengono trasferite alla CPU passando attraverso il bus. La CPU deve quindi provvedere a prelevare le istruzioni dalla memoria (*Fetch*) e quindi ad eseguirle.

Gli elementi principali che fanno parte della CPU sono:

- un registro per conservare l'indirizzo dell'istruzione corrente (*Program Counter — PC*), da aggiornare durante l'evoluzione del programma, in modo da prelevare dalla memoria la corretta sequenza di istruzioni;
- un insieme di registri (a cui si fa riferimento in genere con il termine di *banco di registri*[2]) ad accesso molto rapido, in cui memorizzare i dati di utilizzo più frequente;
- un'unità per l'esecuzione delle operazioni aritmetico-logiche (*Arithmetic-Logic Unit — ALU*) come la somma, la sottrazione, …, inoltre, in aggiunta al risultato dell'operazione, quest'unità fornisce anche una serie di informazioni aggiuntive associate ad un insieme di bit che spesso vengono raccolti in un registro di stato:
  - il risultato è uguale a (diverso da) zero;
  - il risultato è positivo (negativo);
  - il risultato supera le capacità di rappresentazione della CPU (*overflow*);
  - … … …;

  i dati che vengono forniti all'ALU per effetuare le operazioni desiderate possono provenire dai registri oppure direttamente dalla memoria, a seconda delle modalità di indirizzamento previste per le istruzioni aritmetico-logiche;

[2] Per esempio nell'architettura MIPS sono presenti 32 registri (a cui si fa riferimento tramite i nomi $0, $1, …, $31) di 32 bit (4 byte) ciascuno.

- un insieme di unità aggiuntive per elaborazioni particolari (unità aritmetiche per dati in virgola mobile, sommatori ausiliari, …);
- un insieme di dispositivi di controllo, che provvedono a coordinare le operazioni svolte da tutti gli elementi della CPU.

Le architetture delle CPU sono classificate in due categorie:

1. **CISC** (*Complex Instruction Set Computer*), ovvero CPU caratterizzate da
   - elevata complessità delle istruzioni eseguibili, ed elevata dimensione dell'insieme delle istruzioni (sono disponibili molte istruzioni, anche molto potenti);
   - ampia libertà di indirizzamento per gli operandi dell'ALU che possono provenire dai registri oppure dalla memoria, in questo ultimo caso l'indirizzamento può essere diretto (l'indirizzo è contenuto nell'istruzione), indiretto (l'indirizzo è contenuto in un registro), con registro base (l'indirizzo contenuto nell'istruzione deve essere sommato al contenuto di un registro), …;
   - elevata complessità della CPU stessa (cioè dell'hardware relativo) in termini degli elementi che la compongono con la conseguenza di rallentare i tempi di esecuzione delle operazioni (quanto più è complesso l'hardware tanto più si rallenta l'esecuzione di una singola operazione);
   - dimensione variabile delle istruzioni (a seconda della modalità di indirizzamento scelta per ciascuno degli operandi può variare la lunghezza dell'istruzione) e relativa complessità di gestione della fase di prelievo (a priori la CPU non sa quanto è lunga l'istruzione da caricare);
2. **RISC** (*Reduced Instruction Set Computer*), ovvero CPU ispirate al principio di eseguire soltanto istruzioni semplici (le operazioni complesse vengono scomposte in una serie di istruzioni più semplici), con l'obiettivo di farlo a una velocità molto elevata, in modo da superare le prestazioni che si possono ottenere dalle CPU CISC. In particolare, una CPU RISC ha le seguenti caratterisitiche:
   - le istruzioni eseguibili sono molto semplificate;
   - gli operandi dell'ALU possono provenire dai registri e non dalla memoria, per il trasferimento dei dati dalla memoria ai registri si utilizzano delle apposite operazioni di caricamento (*load*) e di memorizzazione (*store*);
   - la CPU è relativamente semplice quindi si riducono i tempi di esecuzione delle singole istruzioni (che però sono meno potenti di quelle eseguite da un CPU CISC);
   - le istruzioni hanno una dimensoine fissa, perciò diventa più semplice gestire la fase di prelievo e la coda delle istruzioni da eseguire.

Attualmente le CPU RISC (MIPS, SPARC, ALPHA, PowerPC, …) consentono di raggiungere prestazioni superiori rispetto a quelle ottenibili dalle CPU CISC (Intel Pentium o 80x86, Motorola 680x0, …).

Passiamo ora ad esaminare la memoria di lavoro. Il suo compito principale consiste nel contenere il sistema operativo ed i processi in esecuzione (completi di codice e dati). Una sua caratteristica fondamentale è la dimensione complessiva. L'unità di misura della capacità di memoria è il bit, che corrisponde ad un elemento base capace di conservare un valore binario (0 o 1). In genere però, per indicare le dimensioni delle memorie si adotta il byte (che corrisponde a 8 bit, cioè a una sequenza di 8 valori binari) insieme ai suoi multipli: Kbyte (2^{10} = 1024 byte), Mbyte (2^{20} = 1 048 576 byte), Gbyte (2^{30} = 1 073 741 824 byte).

Oltre alla dimensione complessiva di una memoria, è importante indicare anche la dimensione di ogni singolo elemento (*parola*) che può essere trasferito da o verso la memoria. Nei calcolatori più recenti la dimensione della parola va dai 32 ai 128 bit (dai 4 ai 16 byte). Per esempio, nell'architettura MIPS, la memoria contiene parole di 32 bit (4 byte), anche se l'indirizzo di queste parole viene specificato indicando il numero del loro primo byte (l'indirizzo della prima parola è 0, quello della seconda è 4, per la terza è 12 e così via, infatti la differenza tra l'indirizzo di una parola e quello della successiva corrisponde appunto ai 4 byte contenuti nella prima delle due parole).

Salvo rare eccezioni, la dimensione della parola di memoria coincide con la dimensione dei registri contenuti nella CPU, in modo da poter caricare una parola di memoria in un registro della CPU.

La comunicazione tra CPU e memoria è assicuarata dal bus di sistema, che è generalmente composto da tre parti (come peraltro illustrato nella figura 1):

1. *Bus dati*, che comprende le linee lungo cui vengono trasferiti i dati da e verso la memoria. Salvo casi particolari, la dimensione del bus dati è tale da garantire il trasferimento contemporaneo di una o più parole di memoria.
2. *Bus indirizzi*, su cui la CPU provvede a trasmettere l'idirizzo di memoria da cui prelevare il dato nel caso di lettura dalla memoria, oppure in cui depositarlo nel caso di scrittura nella memoria.
3. *Bus di controllo*, dove transitano le informazioni ausiliarie necessarie alla corretta definizione delle operazioni da compiere (per esempio l'indicazione che si vuole effettuare una *lettura* piuttosto che non una *scrittura*) e alla sincronizzazione tra CPU e memoria.

Si consideri, per esempio, l'esecuzione di un'operazione di lettura dalla memoria. Come prima cosa la CPU fornisce l'indirizzo della parola desiderata, applicando le opportune tensioni ai fili che costituiscono il bus indirizzi, quindi viene richiesta l'operazione di lettura attivando i relativi fili del bus di controllo. Quando la memoria ha completato la lettura della parola richiesta, il dato viene trasferito sul bus dati e la CPU può da lì prelevarlo ed utilizzarlo nelle sue elaborazioni.

2. L'insieme delle istruzioni

Le istruzioni che sono comprese nel linguaggio macchina di ogni calcolatore possono essere classificate nei seguenti tre gruppi:

1. istruzioni aritmetico-logiche;
2. istruzioni di trasferimento con la memoria;
3. istruzioni di salto condizionato o non condizionato;

Questi gruppi vengono ora presentati più in dettaglio, facendo riferimento (come già detto) all'architettura del processore MIPS™.

2.1 Istruzioni aritmetico-logiche

Con la notazione « `add a,b,c` » si indica che il calcolatore effettua la somma del contenuto delle due variabili `b` e `c` e mette il risultato in `a`. Per un'operazione come l'addizione il numero naturale di operandi è tre: i due contenitori dei valori da sommare e il contenitore del risultato. Il linguaggio macchina qui presentato impone il vincolo che ogni istruzione di questo tipo abbia esattamente tre operandi con l'obiettivo di realizzare un sistema semplice. Operazioni con un maggior numero di operandi possono essere effettuate scomponendole in operazioni più semplici.

Se per esempio si vuole mettere la somma dei contenuti delle variabili `b`, `c`, `d` ed `e` nella variabile `a`, si può adottare la seguente sequenza di istruzioni:

```
add a,b,c                # la somma di b e c è posta in a
add a,a,d                # la somma di b, c e d è messa in a
add a,a,e                # la somma di b,c,d ed e è messa in a
```

Servono quindi tre istruzioni per ottenere la somma di quattro variabili. Come ulteriore esempio si consideri l'espressione

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

che può essere compilata nelle tre istruzioni:

```
add t0,g,h               # la variabile temporanea t0 contiene g+h
add t1,i,j               # la variabile temporanea t1 contiene i+j
sub f,t0,t1              # f riceve t0-t1 ovvero (g+h)-(i+j)
```

Il compilatore crea due nuove variabili, `t0` e `t1`, per riuscire a rappresentare il programma facendo uso di istruzioni che hanno solo tre operandi.

In realtà i tre operandi delle istruzioni aritmetico-logiche possono provenire soltanto dai registri contenuti nella CPU (l'architettura MIPS prevede 32 registri indicati con i nomi `$0`, `$1`, … `$31`). Uno dei compiti del compilatore consiste nell'associare le variabili del programma ai registri. Nel caso dell'espressione `f=(g+h)-(i+j)`, le variabili `f`, `g`, `h`, `i` e `j` possono essere assegnate rispettivamente ai registri `$16`, `$17`, `$18`, `$19` e `$20`:

```
add $8,$17,$18          # la variabile temporanea $8 contiene g+h
add $9,$19,$20          # la variabile temporanea $9 contiene i+j
sub $16,$8,$9           # f riceve $8-$9 ovvero (g+h)-(i+j)
```

2.2 Trasferimenti da e verso la memoria

Il numero dei registri non è comunque sufficientemente grande da consentire di memorizzarvi tutte le variabili di un programma, perciò ad ogni variabile deve essere comunque assegnata una locazione di memoria a cui trasferire il contenuto del registro quando questo stesso registro deve essere utilizzato per contenere un'altra variabile. Per questo motivo il processore deve possedere anche delle modalità di trasferimento dati fra la memoria ed i registri. L'istruzione che trasferisce dati dalla memoria ai registri è chiamata *load* e verrà abbreviata con il termine `lw` che corrisponde a *load word* (carica parola). Il formato di questa istruzione prevede quattro parti: il nome dell'operazione (`lw`), il registro in cui andare a scrivere il dato letto dalla memoria, un indirizzo di base da sommare al contenuto di un secondo registro in modo da ottenere l'indirizzo dell'elemento della memoria che si vuole trasferire nel registro specificato. L'indirizzo di memoria dell'elemento da caricare in un registro si ottiene dalla somma di una parte costante (contenuta nell'istruzione) e del contenuto di un registro. Per esempio, data l'istruzione

```
g = h + A[i]
```

dove `A` è una matrice di 100 elementi che inizia all'indirizzo `Astart`, se il compilatore associa le variabili `g`, `h` ed `i` ai registri `$17`, `$18` e `$19` si ottiene il seguente frammento di codice:

```
lw  $8, Astart($19)     # il registro temporaneo $8 riceve A[i]
add $17, $18, $8        # g=h+A[i];
```

L'istruzione di caricamento `lw` somma l'indirizzo di inizio della matrice `A` (`Astart`) con l'indice `i` contenuto nel registro `$19` per formare l'indirizzo dell'elemento `A[i]`. Il registro che viene sommato all'indirizzo è chiamato *registro indice*. Il processore legge il contenuto dell'indirizzo di memoria calcolato e lo pone nel registro `$8` che viene usato come variabile temporanea. L'istruzione `add` può quindi operare sul valore in `$8` (uguale ad `A[i]`) poiché esso è un registro. Il risultato, contenuto nel registro corrispondente a `g`, è la somma di `A[i]` con `h`. Siccome l'indirizzamento viene effettuato a livello di byte (e non di parola), per ottenere l'indirizzo corretto, il registro `$19` deve contenere `4i` in modo che la somma di `$19` e `Astart` selezioni `A[i]` e non `A[i/4]`.

L'operazione di *store* (memorizzazione) è duale di quella di *load*. Per questa istruzione viene adottata la sigla `sw` che indica *store word*. Il relativo formato prevede quattro parti: il nome della operazione (`sw`), il registro in cui andare a prelevare il dato da scrivere nella memoria, l'indirizzo di inizio dell'array e il registro che contiene l'indice dell'elemento dell'array in cui si vuole trasferire il contenuto del registro specificato. Analogamente al caso precedente, l'indirizzo è composto da una parte costante ed una legata ad un registro. Per esempio, data l'istruzione

```
B[i] = h + A[i]
```

dove `A` e `B` sono due matrici di 100 elementi, che iniziano rispettivamente agli indirizzi `Astart` e `Bstart`, e dove il compilatore associa alle variabili `g`, `h` ed `i` rispettivamente i registri `$17`, `$18` e `$19`, si ottiene il seguente frammento di codice:

```
lw  $8, Astart($19)     # il registro temporaneo $8 riceve A[i]
add $17, $18, $8        # g=h+A[i];
sw  $8, Bstart ($19)    # scrittura di h+A[i] in B[i]
```

2.3 Istruzioni di salto condizionato

Il processore MIPS dispone di due istruzioni di salto condizionato: `beq` (*branch on equal*) e `bne` (*branch on not equal*), illustrate anche dai seguenti esempi:

```
beq register1, register2, L1      # salta a L1 se registro1 è uguale a registro2
bne register1, register2, L1      # salta a L1 se registro1 è diverso da registro2
```

e due di salto incondizionato (dette anche istruzioni di *jump*)

```
j   L1                            # salta a L1
jr  registro1                     # salta all'indirizzo contenuto in registro1
```

Per esempio l'espressione `if (i==j) f = g + h; else f = g - h;` può essere tradotta in:

```
      bne $19, $20, Else    # se i≠j, vai a Else
      add $16, $17, $18     # f=g+h (saltata se i ≠ j)
      j   Exit              # vai a Exit
Else: sub $16, $17, $18     # f=g-h (saltata se i = j)
Exit:
```

Il seguente frammento di programma C

```
while (save[i]==k)
i=i+j;
```

nell'ipotesi che `i`, `j` e `k` corrispondano ai registri `$19`, `$20` e `$21`, che la matrice `save` inizi a `Startsave` e che il registro `$10` contenga il valore 4, può essere tradotto nelle seguenti istruzioni:

```
Loop:  mult $9,$19,$10           # Registro temporaneo $9 = i*4
       lw   $8,Startsave($9)     # Registro temporaneo $8 = save[i]
       bne  $8,$21, Exit         # salta a Exit se save[i] è diverso da k
       add  $19,$19,$20          # i = i + j
       j    Loop                 # salta a Loop
Exit:
```

Spesso la verifica di uguaglianza richiede il confronto con il valore 0, per rendere più veloce questo confronto, nel processore MIPS il registro $0 contiene sempre un valore pari a 0, perciò questo registro non può mai essere utilizzato per contenere altri dati.

La verifica di uguaglianza è probabilmente il test più diffuso, ma spesso è utile anche controllare quale è la più piccola fra due variabili. Tali situazioni sono gestite mediante l'istruzione `slt` (*set on less than*) che confronta due registri e pone un terzo registro ad 1 se il contenuto del primo è minore o uguale del secondo, oppure a 0 in caso contrario:

```
slt    $8, $19, $20         # $8 ← 1 se $19 < $20; $8 ← 0 se $19 ≥ $20
```

2.4 Rappresentazione delle istruzioni

Le istruzioni vengono manipolate dal calcolatore tramite una serie di impulsi elettrici del tipo alto o basso (bit) e, di conseguenza, ogni singolo pezzo costitutivo dell'istruzione può essere considerato come un singolo numero binario che, opportunamente unito agli altri, forma l'istruzione. Ad esempio l'istruzione:

```
add $8, $17, $18
```

è codificata dalla seguente combinazione di numeri decimali:

```
0     17    18    8     0     32
```

Ognuno di tali segmenti che compongono l'istruzione è chiamato *campo* (*field*), il primo e l'ultimo di essi (che nell'esempio contengono rispettivamente i valori 0 e 32), considerati in maniera congiunta, informano il processore che l'operazione da effettuare è una addizione. Il secondo campo rappresenta il numero del registro che contiene il primo operando sorgente (`$17`) e il terzo il registro indica il secondo operando sorgente dell'addizione (`$18`). Il quarto campo specifica il registro che dovrà contenere la somma (`$8`) mentre il quinto, in questa istruzione, rimane inutilizzato ed è azzerato. In sintesi,

l'istruzione somma i registri `$17` e `$18` mettendo il risultato in `$8`. Naturalmente è possibile avere anche una rappresentazione binaria dei vari campi, equivalente alla decimale

Nome campo	op	rs	rt	rd	shamt	funct
Dimensione	6 bit	5 bit	5 bit	5 bit	5 bit	6 bit
`add $8, $17, $18`	000000	10001	10010	01000	00000	100000

Questa strutturazione dell'istruzione è chiamata *formato dell'istruzione*. Come si può notare contando il numero dei bit, l'istruzione precedente ne occupa 32, la stessa dimensione di una parola di dati. In ossequio al principio secondo il quale la semplicità favorisce la regolarità, tutte le istruzioni del processore preso a riferimento in queste note hanno lunghezza pari a 32 bit. Ai vari campi sono stati assegnati dei nomi mnemonici per facilitarne la comprensione; il significato dei nomi dei vari campi è il seguente:

- *op:* operazione effettuata dall'istruzione;
- *rs:* registro contenente il primo operando sorgente;
- *rt:* registro contenente il secondo operando sorgente;
- *rd:* registro destinazione, esso conterrà il risultato dell'operazione;
- *shamt:* scorrimento (*shift amount*), utilizzato per alcune operazioni aritmetiche (nella nostra trattazione verrà trascurato);
- *funct:* funzione (seleziona una delle possibili varianti all'operazione specificata nel campo *op*).

Può nascere un problema qualora un'istruzione necessiti di campi più lunghi di quelli mostrati in precedenza. Ad esempio, l'istruzione di caricamento (`lw`) deve specificare due registri ed un indirizzo. Se per l'indirizzo si utilizzano solo cinque bit, come suggerito dal formato appena descritto, si potrebbero indirizzare solamente 32 locazioni di memoria. Per mantenere invariata la lunghezza delle diverse istruzioni si deve variare il formato. Il formato descritto in precedenza è detto *tipo R* (*registro*).

Un secondo tipo di formato istruzione, usato tra l'altro per le istruzioni di trasferimento dati fra CPU e memoria, è quello di *tipo I*, i cui campi sono:

Nome campo	op	rs	rt	indirizzo
Dimensione	6 bit	5 bit	5 bit	16 bit
`lw $8, Astart ($19)`	100011	10011	01000	0000 0100 1011 0000 (`Astart`)

In questo caso nel campo *rs* è posto 19, 8 in *rt* e `Astart`, che è il nome dell'indirizzo di inizio della matrice `A`, è posizionato all'interno del campo *indirizzo*. Si noti come il significato del campo *rt* sia variato: nella istruzione di caricamento indica il registro che deve contenere il risultato. Anche se la presenza di diversi formati rende più complessa la macchina fisica, tale inconveniente può essere attenuato mantenendo una similarità fra i vari formati. Ad esempio, i primi tre campi del tipo R e del tipo-I hanno lo stesso nome, ed il quarto campo del secondo tipo è lungo quanto gli ultimi tre campi del tipo R. Quando necessario, i formati possono essere riconosciuti tramite il valore del primo campo: ad ogni formato è assegnato un insieme di valori del primo campo (op) che indicano alla macchina se trattare la rimanente metà istruzione come composta da tre campi (tipo R) o da uno singolo (tipo I). Il campo usato per operare tale discriminazione è tradizionalmente noto come *codice operativo* (*opcode*).

Istr.	Formato	op	rs	rt	rd	shamt	funct	indirizzo
add	R	0	reg	reg	reg	0	32	*n.a.*
sub	R	0	reg	reg	reg	0	34	*n.a.*
lw	I	35	reg	reg	*n.a.*	*n.a.*	*n.a.*	indirizzo
sw	I	43	reg	reg	*n.a.*	*n.a.*	*n.a.*	indirizzo

Tabella 1 Codifica di alcune istruzioni. In questa tabella, *reg* rappresenta il numero di registro compreso fra 0 e 31, *indirizzo* un valore a 16 bit oppure n.a. (non disponibile) nei formati ove esso non compaia. Si osservi che le istruzioni `add` e `sub` contengono lo stesso valore nel campo *op*; la macchina considera quanto specificato nel campo *funct* per distinguere l'addizione dalla sottrazione.

Si consideri per esempio la seguente espressione in linguaggio C

```
A[i] = h + A[i];
```

Nell'ipotesi che `$19` contenga `4i` e che il registro `$18` contenga `h`, il risultato che si ottiene dalla compilazione è

```
lw  $8, Astart ($19)        # il registro temporaneo $8 riceve A[i]
add $8, $18, $8             # il registro temporaneo $8 riceve h+A[i]
sw  $8, Astart ($19)        # scrittura di h+A[i] in A[i]
```

Per semplicità rappresentiamo inizialmente le istruzioni usando i numeri decimali. Supponendo che l'indirizzo di inizio della matrice `A` sia 1200_{dieci} (ovvero $0000\ 0100\ 1011\ 0000_{due}$), le tre istruzioni sono:

Istruzione	Tipo	Forma decimale	Forma binaria
`lw $8, Astart ($19)`	I	35 19 8 1200	100011 10011 01000 0000 0100 1011 0000
`add $8, $18, $8`	R	0 18 8 8 0 32	000000 10010 01000 01000 00000 100000
`sw $8, Astart ($19)`	I	43 19 8 1200	101011 10011 01000 0000 0100 1011 0000

È da notare la similarità fra le rappresentazioni in binario della prima e dell'ultima istruzione; la sola differenza è nel terzo bit a partire da sinistra.

2.5 Altre modalità di indirizzamento

2.5.1 Costanti oppure operandi immediati

Nel corso delle operazioni si utilizzano spesso delle costanti, ad esempio per incrementare un indice al fine di puntare all'elemento successivo, per contare le iterazioni di un ciclo. A questo scopo sono introdotte delle istruzioni aritmetiche nelle quali un operando è costante ed il suo valore è contenuto all'interno dell'istruzione stessa. Per uniformità viene usato lo stesso formato delle istruzioni di trasferimento di dati. Il nome "I" dato al formato tipo-I, indica appunto un valore *immediato* che è il nome consueto per tale tipo di operando. Il campo che contiene la costante è lungo 16 bit. Per esempio l'istruzione di somma che ha una costante come operando è chiamata *somma con un operando immediato* (`addi`). Per aggiungere 4 al contenuto del registro $29 si deve scrivere

		op	rs	rt	indirizzo
`addi $29, $29, 4`	decimale	8	29	29	4
`addi $29, $29, 4`	binario	001000	11101	11101	0000 0000 0000 0100

2.5.2 Indirizzamento nei salti

Il più semplice indirizzamento è presente nelle istruzioni jump del MIPS. Esse fanno uso del formato detto di tipo J, che dedica 6 bit per il campo che specifica la operazione e dei rimanenti per il campo indirizzo. Ad esempio:

```
j   10000                 # salta alla locazione 10000
```

Nel caso del salto condizionato si debbono specificare due operandi oltre all'indirizzo di salto, lasciando solo 16 bit per l'indirizzo del salto. Se gli indirizzi di un programma debbono rientrare in tale campo di 16 bit, non è possibile avere programmi con dimensione superiore a 2^{16} byte. Una alternativa potrebbe essere quella di specificare un registro il cui valore debba essere sempre sommato all'indirizzo del salto. In tale ipotesi una istruzione di salto dovrebbe effettuare il seguente calcolo: *PC = registro + indirizzo di salto*.

Tale soluzione risolve il problema consentendo al programma di avere dimensioni fino 2^{32} byte, ferma restando la possibilità di usare i salti condizionati. Rimane da decidere quale sia il registro da utilizzare. I salti condizionati sono principalmente impiegati all'interno dei cicli e nei costrutti condizionali e, di conseguenza, il salto avviene verso istruzioni piuttosto vicine. Poiché il contatore di programma (PC) contiene l'indirizzo dell'istruzione corrente, usandolo come registro da sommare all'indirizzo è possibile

saltare in un'area di 2^{16} byte distante dall'istruzione corrente. Questa forma di indirizzamento viene detta *indirizzamento relativo al contatore di programma*. Consideriamo, per esempio, il *ciclo a condizione iniziale* riportato nell'esempio di pagina 6 che corrisponde al seguente codice:

```
Loop:  mult $9,$19,$10         # registro temporaneo $9 = i*4
       lw   $8,Startsave($9)   # registro temporaneo $8 = save[i]
       bne  $8,$21, Exit       # salta a Exit se save[i] ≠ k
       add  $19,$19,$20        # i = i + j
       j    Loop               # salta a Loop
Exit:
```

Se supponiamo che il ciclo sia posto in memoria alla locazione 80000 e che l'indirizzo `Startsave` si riferisca alla locazione 1000, il codice macchina corrispondente è

Istruzione	Posizione in memoria	Codifica (decimale)					
`Loop: mult $9,$19,$10`	80000	0	19	10	9	0	24
`lw $8,Startsave($9)`	80004	35	9	8		1000	
`bne $8,$21, Exit`	80008	5	8	21		8	
`add $19,$19,$20`	80012	0	19	20	19	0	32
`j Loop`	80016	2			80000		
`Exit: …`	80020	…	…	…	…	…	…

Si ricordi che viene adottato un indirizzamento a livello di byte; gli indirizzi di parole consecutive differiscono quindi del valore 4, corrispondente al numero di byte che compongono una parola. L'istruzione `bne` sulla terza riga somma 8 byte all'indirizzo dell'istruzione seguente (80012), specificando la destinazione del salto relativamente a tale istruzione (8), in luogo dell'indirizzo completo (80020). In alternativa si può delegare all'hardware la trasformazione dell'indirizzo di parola in indirizzo di byte, come verrà illustrato nel seguito. L'istruzione di salto dell'ultima riga utilizza l'indirizzo completo (80000), corrispondente all'etichetta `Loop`.

Nome	Esempio	Commenti
32 registri	$0, $1, $2, . . . , $31	Accesso rapido ai dati. I dati debbono essere nei registri per potere effettuare operazioni aritmetiche. Il registro $0 ha sempre valore zero.
2^{30} parole di memoria	Memory[0], Memory[4], . . . , Memory[4294967292]	Sono usate solo dalle istruzioni di trasferimento dati. Il processore MIPS usa indirizzi a livello di byte perciò indirizzi di parole consecutive differiscono di un fattore 4. La memoria contiene le strutture dati, come le matrici e le copie dei registri, di uso corrente e salvate durante le chiamate di procedura.

Tabella 2 Operandi utilizzati dalle istruzioni del linguaggio assemblatore del processore MIPS.

	6 bit	5 bit	5 bit	5 bit	5 bit	6 bit	Commenti
Formato R	op [31-26]	rs [25-21]	rt [20-16]	rd [15-11]	shamt [10-6]	funct [5-0]	istruzioni aritmetiche su registri
Formato I	op [31-26]	rs [25-21]	rt [20-16]	indirizzo/immediato [15-0]			trasferimenti, salti condizionati, istruzioni aritmetiche con immediati
Formato J	op [31-26]	indirizzo destinazione [25-0]					istruzioni di salto

Tabella 3 Struttura dei tre diversi formati adottati per la codifica delle istruzioni MIPS. Si noti come tutte le istruzioni siano contenute in 32 bit. Tra parentesi quadre viene indicato l'insieme dei bit che contengono il campo corrispondente (per esempio nel formato R il campo *shamt* è contenuto nei bit che vanno dal 10 al 6).

Istruzione	Significato	Tipo	Codifica (decimale)						Commenti
`add $1,$2,$3`	$1 ← $2 + $3	R	0	2	3	1	0	32	3 operandi nei registri
`sub $1,$2,$3`	$1 ← $2 – $3	R	0	2	3	1	0	34	3 operandi nei registri
`addi $1,$2,100`	$1 ← $2 + 100	I	8	2	1	100			2 operandi nei registri e 1 immediato
`lw $1,100($2)`	$1 ← M [$2+100]	I	35	2	1	100			Dati dalla memoria nei registri
`sw $1,100($2)`	M [$2+100] ← $1	I	43	2	1	100			Dati dai registri nella memoria
`lui $1,100`	$1 ← 100 * 2^{16}	I	15	0	1	100			Costante nei 16 bit più significativi
`beq $1,$2,100`	se ($1 == $2) vai a PC+4+100	I	4	1	2	100			Verifica uguaglianza; salto relativo a PC
`bne $1,$2,100`	se ($1 != $2) vai a PC+4+100	I	5	1	2	100			Verifica di differenza; salto relativo a PC
`slt $1,$2,$3`	se($2 < $3) $1=1; altrimenti $1=0	R	0	2	3	1	0	42	Assegna 1 se un registro è minore di un altro
`slti $1,$2,100`	if ($2 < 100) $1=1; altrimenti $1=0	I	10	2	1	100			Assegna 1 se un registro è minore di una costante
`j 10000`	vai a 10000	J	2	10000					Salta all'indirizzo di destinazione
`jr $31`	vai a $31	R	0	31	0	0	0	8	Selezione multipla e ritorno da procedura
`jal 10000`	$31 = PC + 4; vai a 10000	J	3	10000					Chiamata di procedura

Tabella 4 Insieme semplificato delle istruzioni del processore MIPS.

In conclusione le possibili modalità di indirizzamento presentate finora sono:
1. *A registro*: l'operando è un registro.
2. *Base* (detto anche con *spiazzamento*): l'operando è una locazione di memoria il cui indirizzo si ottiene sommando un registro ad un indirizzo contenuto nell'istruzione.
3. *Immediato*: l'operando è una costante contenuta nell'istruzione.
4. *Relativo al PC*: l'indirizzo è la somma del PC e di una costante contenuta nell'istruzione.

Nelle tabelle 2, 3 e 4 vengono riportati rispettivamente gli operandi utilizzabili dalle istruzioni MIPS, i tre formati diversi adottati per le istruzioni MIPS e un insieme di istruzioni in cui sono comprese anche tutte quelle presentate fino ad ora.

3. La struttura del processore

3.1 Passi svolti per l'esecuzione delle diverse istruzioni

Invece di esaminare l'intera unità di calcolo come un unico blocco, è più semplice analizzare l'esecuzione di un'istruzione mediante una determinata sequenza di passi, focalizzando l'attenzione sulla parte di unità di calcolo associata ad ogni passo.

Un'istruzione di tipo R, ad esempio `add $x, $y, $z`, viene eseguita in quattro passi:
1. L'istruzione è prelevata dalla memoria istruzioni e il PC viene incrementato.
2. Due registri, `$y` e `$z`, all'interno del banco di registri vengono letti.
3. La ALU opera sui dati letti dal banco dei registri, utilizzando il codice funzione (i bit 5-0 dell'istruzione) per realizzare la funzione della ALU.
4. Il risultato della ALU viene scritto nel banco di registri utilizzando i bit 15-11 dell'istruzione per selezionare il registro destinazione (`$x`).

È possibile illustrare in modo simile a quanto fatto per le istruzioni di tipo R, anche l'esecuzione di un'istruzione di caricamento di parola, quale `lw $x, offset($y)`. In questo caso sono però necessari cinque passi:
1. L'istruzione è prelevata dalla memoria istruzioni e il PC viene incrementato.
2. Il valore di un registro (`$y`) viene letto dal banco dei registri.
3. La ALU calcola la somma del valore letto dal banco di registri ed i 16 bit meno significativi dell'istruzione estesi in segno (`offset`).
4. La somma calcolata dalla ALU è utilizzata come indirizzo per la memoria dati.
5. Il dato proveniente dall'unità di memoria viene scritto nel banco dei registri; il registro destinazione è indicato dai bit 20-16 dell'istruzione (`$x`).

Infine, è possibile illustrare, con la stessa modalità, l'esecuzione dell'istruzione di salto condizionato, ad esempio `beq $x, $y, offset`. I passi sono molto simili a quelli dell'istruzione di tipo R, però l'uscita *Zero* della ALU viene utilizzata per determinare se il nuovo valore del PC debba essere PC + 4 oppure l'indirizzo destinazione del salto. I quattro passi dell'esecuzione sono:

1. Un'istruzione è prelevata dalla memoria istruzioni e il PC viene incrementato.
2. Due registri, `$x` e `$y`, vengono letti dal banco dei registri.
3. L'unità di calcolo effettua una sottrazione dei valori letti dal banco dei registri. Il valore PC + 4 viene sommato ai 16 bit meno significativi dell'istruzione estesi in segno (`offset`); il risultato è l'indirizzo destinazione del salto.
4. L'uscita *Zero* della ALU viene utilizzata per decidere quale valore debba essere memorizzato nel PC.

Tipo di istruzione	Unità funzionali utilizzate				
Tipo R	Memoria istruzioni	Banco registri	ALU	Banco registri	
Caricamento parola	Memoria istruzioni	Banco registri	ALU	Memoria dati	Banco registri
Memorizzazione parola	Memoria istruzioni	Banco registri	ALU	Memoria dati	
Salto condizionato	Memoria istruzioni	Banco registri	ALU		
Salto	Memoria istruzioni				

Tabella 5 Unità funzionali richieste per l'esecuzione delle diverse classi di istruzioni.

Nell'ipotesi che il tempo richiesto dalle principali unità funzionali per svolgere le relative operazioni sia 10 ns (nanosecondi) per le memorie, l'ALU e i sommatori ausiliari, 5 ns per i registri e 0 per i multiplexer, l'unità di controllo e il PC, il tempo di esecuzione di ogni istruzione è stabilito dal cammino critico, che dipende dal numero di operazioni richieste, come indicato nelle tabelle 5 e 6. La figura 2 riporta, come esempio di esecuzione, la temporizzazione di una sequenza di tre istruzioni di caricamento dati dalla memoria ai registri.

Tipo di istruzione	Memoria istruzioni	Lettura registro	Operazione ALU	Memoria dati	Scrittura registro	Totale
Tipo R (`add`, `sub`, `and`, `or`)	10	5	10	0	5	30 ns
Caricamento parola (`lw`)	10	5	10	10	5	40 ns
Memorizzazione parola (`sw`)	10	5	10	10		35 ns
Salto condizionato (`beq`)	10	5	10	0		25 ns
Salto	10					10 ns

Tabella 6 Durata minima di ogni tipo di istruzione. Se il processore deve effettuare ogni istruzione in un solo ciclo di clock, la durata del ciclo non deve essere inferiore ai 40 ns.

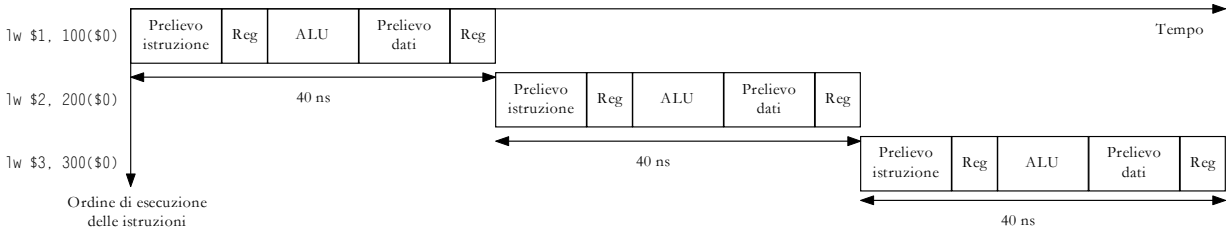


Figura 2 Esecuzione di una sequenza di operazioni di caricamento dalla memoria

3.2 Struttura generale del processore

Viene ora presentata una possibile realizzazione dell'unità centrale capace di eseguire un sottoinsieme delle istruzioni presentate prima, comprendente in particolare le seguenti istruzioni:

- quelle di riferimento a memoria per il caricamento e la memorizzazione delle parole (rispettivamente `lw` e `sw`);

- quelle aritmetico-logiche come `add`, `addi`, `sub`, `and`, `or`, `slt`, …
- quelle di salto condizionato (`beq`, `bne`).

Molto di ciò che deve esser fatto per realizzare queste istruzioni non varia da istruzione a istruzione. Per ogni istruzione, i primi due passi sono infatti identici:

1. Inviare il contenuto del contatore di programma (*Program Counter - PC*) ad una memoria che contiene il codice per prelevare l'istruzione.
2. Leggere uno o due registri utilizzando i campi dell'istruzione per selezionare i registri ai quali accedere. Per un'istruzione di caricamento (`load`) è necessario leggere un solo registro, ma per tutte le altre istruzioni è necessario leggerne due.

Dopo questi due passi, le azioni necessarie per concludere l'esecuzione dell'istruzione dipendono dal tipo di istruzione. Comunque, per ognuno dei tre tipi di istruzioni (riferimento a memoria, aritmetico-logico e salto condizionato), le azioni sono a grandi linee le stesse, indipendentemente dall'esatto codice operativo. Perfino tra diverse classi di istruzioni ci sono similitudini. Per esempio, tutti i tipi di istruzioni utilizzano la ALU dopo la lettura dei registri. Le istruzioni di riferimento a memoria utilizzano la ALU per il calcolo dell'indirizzo effettivo, le istruzioni aritmetico-logiche per l'esecuzione del codice operativo e i salti condizionati per valutare l'esito dei confronti.

Dopo aver utilizzato la ALU, le azioni richieste per completare le varie istruzioni si differenziano. Un'istruzione di riferimento a memoria richiederà l'accesso alla memoria dati per portare a termine la memorizzazione o la lettura di una parola che deve essere caricata. Un'istruzione aritmetico-logica deve scrivere nuovamente in un registro i dati provenienti dalla ALU. Infine, per una istruzione di salto condizionato, potrebbe essere necessario cambiare l'indirizzo dell'istruzione successiva sulla base del confronto.

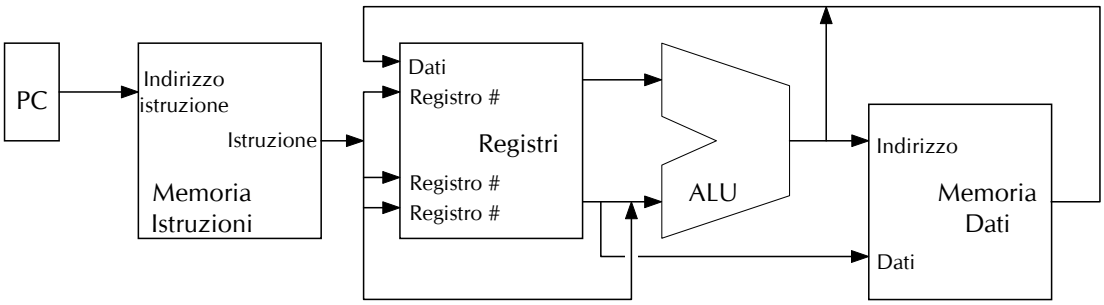


Figura 3 Rappresentazione astratta della unità centrale di elaborazione che esegue il sottoinsieme di istruzioni preso in esame. Tutte le istruzioni iniziano utilizzando il contatore di programma che fornisce l'indirizzo dell'istruzione corrente. Questo indirizzo viene utilizzato per prelevare l'istruzione dalla memoria. Alcune parti dell'istruzione indicano direttamente i registri che debbono essere utilizzati come operandi dell'istruzione stessa e vengono perciò collegate agli ingressi del banco dei registri. Si noti come il banco di registri sia dotato di quattro ingressi e di due uscite; a tre dei quattro ingressi sono collegati i campi dell'istruzione che specificano il numero dei due registri da leggere e del registro che eventualmente deve essere scritto, l'altro ingresso contiene i dati che possono essere scritti nel registro di destinazione. Le due uscite del banco dei registri riportano il contenuto dei due registri letti. Una volta individuati gli operandi, questi possono essere manipolati per calcolare un indirizzo di memoria (per un caricamento o una memorizzazione), per calcolare un risultato aritmetico (per un'istruzione aritmetico-logica), o per effettuare un confronto (per un salto condizionato). Se l'istruzione è aritmetico-logica, il risultato della ALU deve essere scritto in un registro. Se l'operazione è di caricamento o memorizzazione, il risultato della ALU viene usato come indirizzo per memorizzare un valore proveniente da un registro oppure per caricare un valore dalla memoria in un registro. Per questo il risultato proveniente dalla ALU o dalla memoria viene riscritto nel banco dei registri. I salti condizionati richiedono l'utilizzo dell'uscita della ALU per determinare l'indirizzo della prossima istruzione, che necessita di apposita logica di controllo, come verrà illustrato nel seguito.

3.3 Il caricamento delle istruzioni

Il primo elemento necessario è un luogo in cui memorizzare le istruzioni di un programma. Una unità di memoria serve per mantenere e fornire istruzioni una volta noto un indirizzo. L'indirizzo dell'istruzione deve essere memorizzato in un apposito registro, detto *Program Counter* (*PC*). Infine, è

necessario disporre di un apposito sommatore per incrementare il PC in modo da poter indirizzare l'istruzione successiva. Gli elementi dell'unità di elaborazione che si occupano di questa prima fase sono mostrati in figura 4.

3.4 Esecuzione delle istruzioni di tipo R

Si considerino ora le istruzioni di tipo R. Tutte leggono due registri, effettuano un'operazione mediante la ALU sul contenuto dei registri, e scrivono il risultato. Queste istruzioni vengono denominate *istruzioni di tipo* R oppure *istruzioni aritmetico-logiche*. I 32 registri del processore vengono organizzati in una struttura chiamata *banco di registri* (*register file*).

Visto che le istruzioni di tipo R hanno come operandi tre registri, per ogni istruzione si dovranno leggere due parole di dati dal banco di registri e se ne dovrà scrivere una. Per ogni parola letta dai registri sono necessari un ingresso al banco di registri, dove specificare il numero del registro che si vuole leggere, ed un'uscita dal banco, dove comunicare il valore letto dal registro. Per scrivere una parola sono necessari due ingressi: uno che specifichi il numero del registro in cui si vuole scrivere ed un altro per il dato da scrivere. Di conseguenza, sono richiesti un totale di quattro ingressi (tre per i numeri dei registri ed uno per i dati) e due uscite (entrambe per i dati), come già mostrato nella figura 3. Il banco di registri fornisce sempre in uscita il contenuto dei registri i cui numeri sono specificati sugli ingressi del registro di lettura. Le scritture, invece, sono controllate da un apposito segnale di controllo: il segnale di scrittura. Gli ingressi che indicano il numero del registro sono di 5 bit in modo da poter specificare 1 dei 32 registri ($32 = 2^5$), mentre l'ingresso e le due uscite che riguardano i dati sono di 32 bit. La ALU riceve due ingressi di 32 bit e restituisce un risultato di 32 bit.

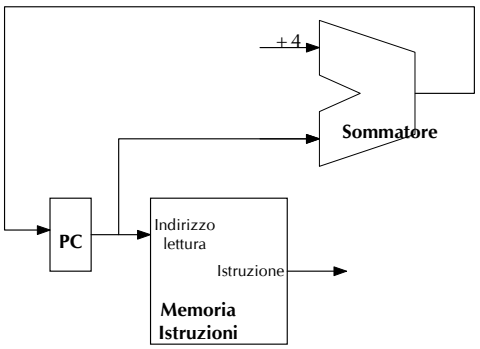


Figura 4 Elementi dell'unità di centrale elaborazione necessari per prelevare le istruzioni dalla memoria e per incrementare il contenuto del Program Counter.

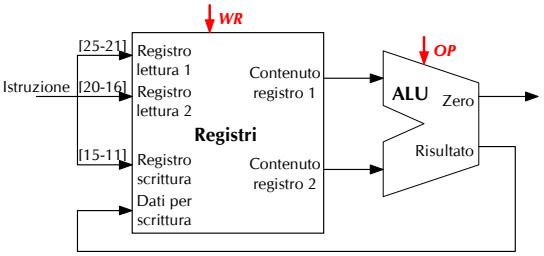


Figura 5 Elementi dell'unità di centrale elaborazione necessari per eseguire le istruzioni di tipo R. I numeri tra parentesi quadre indicano la posizione dei diversi campi all'interno dei 32 bit dell'istruzione. I segnali WR ed OP provengono dall'unità di controllo e provvedono rispettivamente a regolare la scrittura del risultato nel banco dei registri ed a specificare l'operazione che deve essere eseguita dall'ALU.

3.5 Esecuzione delle istruzioni di trasferimento da e verso la memoria

Si considerino ora le istruzioni di caricamento e memorizzazione, che hanno la forma generale: `lw $1, offset($2)` oppure `sw $1, offset($2)`. Queste istruzioni calcolano un indirizzo di memoria sommando il contenuto del registro base (`$2`) al valore di 16 bit con segno del campo di spiazzamento (*offset*) contenuto nell'istruzione. Se l'istruzione è una memorizzazione, il valore da memorizzare deve anche essere letto dal banco di registri (`$1`). Se l'istruzione è un caricamento, il valore letto dalla memoria deve essere scritto in un registro specifico (`$1`) del banco. Quindi, saranno necessari sia il banco di registri sia la ALU, entrambi richiesti per istruzioni tipo R e già mostrati in figura 5. Inoltre, sarà necessario introdurre una unità per estendere con il segno corretto il valore dello spiazzamento dai 16 bit contenuti nell'istruzione ad un valore con segno di 32 bit, e una unità di memoria dati da cui leggere o scrivere.

La figura 6 mostra come combinare tutti questi elementi per costruire un'unità di calcolo per l'istruzione di caricamento o di memorizzazione di una parola, supponendo che l'istruzione sia già stata

prelevata. Il numero del registro è prelevato direttamente dall'istruzione, così come il valore dello spiazzamento, che, dopo l'operazione di estensione del segno, viene utilizzato come secondo operando della ALU.

3.6 Esecuzione delle istruzioni di salto

Passiamo ora a considerare le istruzioni di salto condizionato. Per esempio l'istruzione `beq` ha tre operandi, due registri che vengono confrontati per verificarne l'uguaglianza, ed un valore di spiazzamento di 16 bit che indica la destinazione del salto, a patto di sommarlo prima al contenuto del PC. Ci sono due particolari dell'architettura dell'insieme di istruzioni a cui è necessario prestare una particolare attenzione:

1. la base per il calcolo dell'indirizzo di un salto condizionato è l'indirizzo dell'istruzione che segue quella di salto (un salto di 0 istruzioni corrisponde all'esecuzione dell'istruzione successiva a quella di salto e non alla ripetizione dell'istruzione di salto). Poiché nella fase di prelievo dell'istruzione viene calcolato PC + 4 (cioè l'indirizzo dell'istruzione seguente), è facile utilizzare questo valore come base per il calcolo dell'indirizzo di destinazione del salto condizionato;
2. se il salto è specificato in termini di istruzioni da saltare, visto che ogni istruzione occupa 4 byte, prima di effettuare la somma dello spiazzamento con il contenuto del PC, bisogna moltiplicare per 4 il valore contenuto nell'istruzione. Questo viene ottenuto tramite un semplice scorrimento a sinistra di 2 bit.

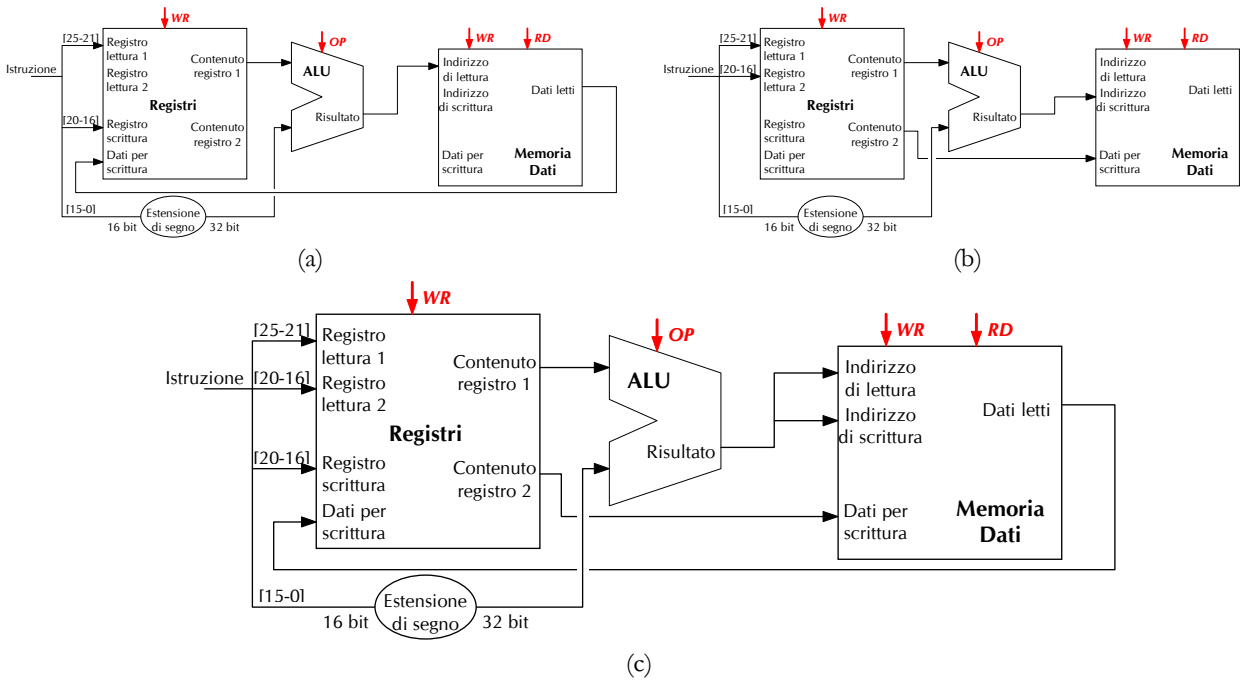


Figura 6 Elementi dell'unità di centrale elaborazione necessari per eseguire (a) le istruzioni di caricamento; (b) le istruzioni di memorizzazione e (c) entrambe le istruzioni di caricamento e memorizzazione. Ogni operazione è costituita dal calcolo di un indirizzo di memoria, da una lettura o scrittura della memoria, e (nel caso l'istruzione sia di caricamento) da una scrittura nel banco di registri. Si noti come il registro destinazione della scrittura sia ora indicato dai bit 20-16 dell'istruzione, mentre nelle istruzioni di tipo R venivano utilizzati i bit 15-11 (figura 5).

Oltre a calcolare l'indirizzo di destinazione del salto condizionato, l'unità centrale deve anche determinare quale sia l'istruzione da eseguire: quella immediatamente successiva oppure quella che si trova all'indirizzo di destinazione del salto? Quando la condizione è verificata (nel caso dell'istruzione `beq` questo capita quando gli operandi sono uguali), l'indirizzo di destinazione del salto condizionato diventa il nuovo PC. Se invece la condizione non è verificata, il PC incrementato sostituisce il PC attuale (come in ogni altra normale istruzione). Riassumendo, l'operazione di salto condizionato è costituita da due azioni: calcolo dell'indirizzo di destinazione del salto e confronto del contenuto dei registri. Per calcolare l'indirizzo di destinazione sono necessari un sommatore e una unità per l'estensione del segno.

Si deve inoltre modificare la parte dell'unità di calcolo relativa al prelievo dell'istruzione. Per effettuare il confronto è necessario utilizzare il banco di registri per fornire i due registri operandi (sebbene non sia necessario scrivere nel banco dei registri). Inoltre, il confronto può essere fatto dalla ALU che fornisce un'uscita attraverso cui indica se il risultato dell'operazione svolta è 0: si possono mandare i due operandi alla ALU ed effettuare una sottrazione, selezionando opportunamente i segnali di controllo. Se il segnale *Zero* d'uscita della ALU (quello che dice se il risultato è uguale a 0) viene posto ad 1, si saprà che i due operandi sono uguali. L'unità di calcolo per un salto condizionato combina tutti questi elementi, come mostrato in figura 7.

L'istruzione di salto non condizionato viene effettuata sostituendo una parte del PC con i 26 bit meno significativi dell'istruzione.

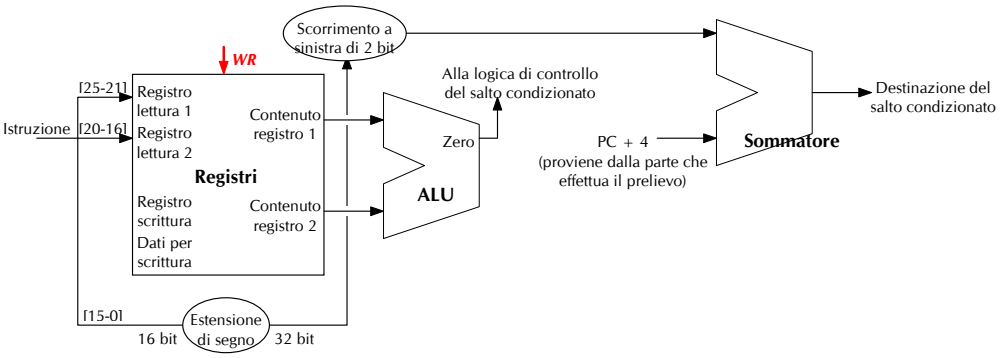


Figura 7 Elementi dell'unità di centrale elaborazione necessari per eseguire le istruzioni di salto condizionato. Viene utilizzata la ALU per valutare la condizione e un apposito sommatore per calcolare l'indirizzo di destinazione ottenuto dalla somma del PC (già incrementato nella parte della CPU che si occupa del prelievo delle istruzioni) e il valore contenuto nell'istruzione (dopo averne esteso il segno e averlo fatto scorrere a sinistra di 2 bit). La logica di controllo è utilizzata per decidere (in base al valore indicato dall'ALU sull'uscita *Zero*) se il nuovo valore del PC deve essere quello ottenuto dall'incremento del valore precedente oppure l'indirizzo indicato come destinazione del salto condizionato.

3.7 Realizzazione di un processore completo

Esaminati gli elementi dell'unità di elaborazione richiesti da ogni tipo di operazione, è possibile combinarli in un'unica unità di elaborazione. Per semplicità si ipotizza che tutte le istruzioni siano eseguite in un solo colpo. Ciò significa che nessuna risorsa dell'unità di calcolo può essere utilizzata più di una volta per istruzione e che qualsiasi elemento di cui si ha bisogno più di una volta deve essere duplicato. Occorre quindi una memoria istruzioni distinta dalla memoria dati. Alcune delle unità funzionali dovranno essere duplicate nel momento in cui si combinano le varie unità di calcolo definite nella precedente sezione, mentre molti elementi possono essere condivisi da differenti flussi di istruzioni. Per condividere un elemento dell'unità di calcolo tra due diversi tipi di istruzione, si dovrà consentire la presenza di connessioni multiple all'ingresso di un elemento ed avere un segnale di selezione tra gli ingressi. Questo viene generalmente realizzato mediante un elemento chiamato *multiplexer*, (potrebbe essere più propriamente chiamato *selezionatore di dati*). Il multiplexer seleziona uno tra i vari ingressi in base alla configurazione delle linee di controllo.

L'unità di calcolo che svolge le istruzioni aritmetico-logiche (quelle di tipo R) e quella dedicata alle istruzioni di caricamento e di memorizzazione, illustrate rispettivamente nelle figure 5 e 6 sono molto simili. Le differenze più significative sono:

- il secondo ingresso dell'unità ALU è un registro (istruzione di tipo R) oppure la metà meno significativa dell'istruzione (istruzione di riferimento a memoria);
- il valore scritto nel registro destinazione proviene dalla ALU (istruzione tipo R) oppure dalla memoria (istruzione di caricamento);
- il numero del registro in cui si vuole scrivere il risultato è indicato da diversi campi (i bit 15-11 per le istruzioni di tipo R e quelli 20-16 per le istruzioni di tipo I).

Per combinare le due unità di calcolo, utilizzando un unico banco di memoria ed un'unica ALU, bisogna consentire al secondo ingresso della ALU di avere due sorgenti distinte, così come sono necessarie due

sorgenti diverse per i dati memorizzati nel banco dei registri e per il numero di registro da scrivere. Di conseguenza, un multiplexer viene posizionato all'ingresso della ALU ed un altro all'ingresso dati del banco dei registri. La figura 8 mostra l'unità di calcolo complessiva.

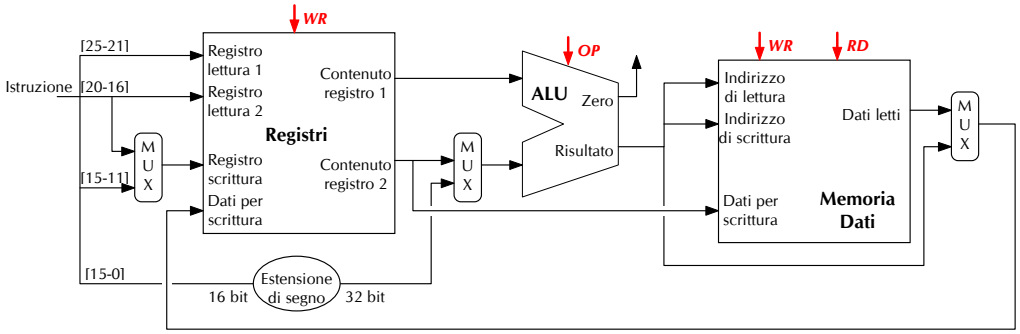


Figura 8 Elementi dell'unità di elaborazione in grado di eseguire le istruzioni di tipo R e quelle di riferimento alla memoria (caricamento e memorizzazione).

La parte di unità di calcolo dedicata al prelievo dell'istruzione, mostrata in figura 4, può facilmente essere aggiunta all'unità di calcolo di figura 8, in modo da ottenere il risultato riportato in figura 9.

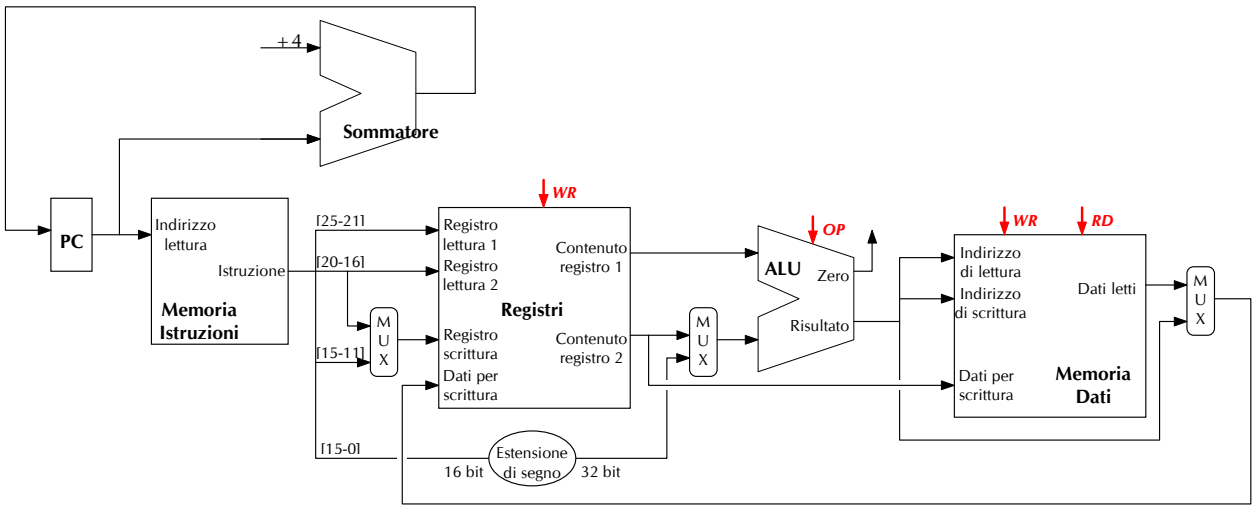


Figura 9 Elementi dell'unità di elaborazione che si occupano del prelievo dell'istruzione (presi dalla figura 4) e della gestione delle istruzioni di tipo R e di riferimento a memoria (presi dalla figura 8).

A questo punto si può procedere ad unire tutte le parti aggiungendo anche gli elementi richiesti dai salti condizionati (figura 7). Si ottiene così l'unità di elaborazione completa di figura 10. L'istruzione di salto condizionato utilizza la ALU principale per il confronto degli operandi, quindi si deve aggiungere un sommatore ausiliario per calcolare l'indirizzo di destinazione. Serve inoltre un multiplexer aggiuntivo per selezionare cosa scrivere nel PC: l'indirizzo dell'istruzione immediatamente seguente (PC + 4) o l'indirizzo di destinazione del salto condizionato.

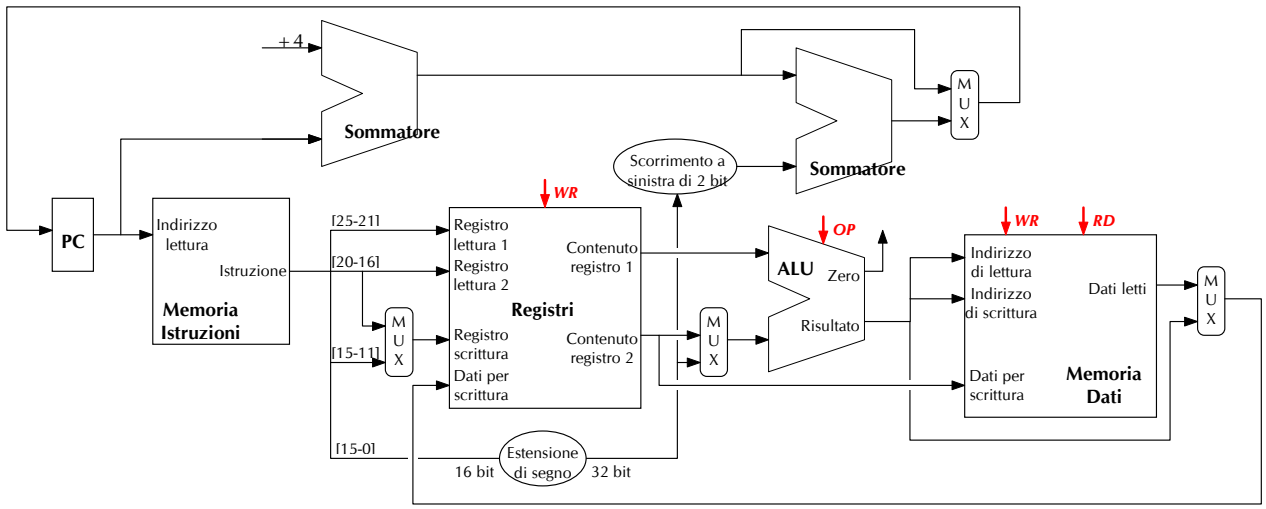


Figura 10 Combinazione di tutti gli elementi dell'unità di elaborazione richiesti dalle diverse classi di istruzioni. Questa unità di calcolo può eseguire le istruzioni elementari (caricamento/memorizzazione parola, operazioni ALU e salti condizionati) in un unico ciclo di clock.

4. Come migliorare le prestazioni utilizzando le pipeline

4.1 Introduzione

Il *pipelining* è una tecnica di realizzazione in cui si sovrappone l'esecuzione di più istruzioni. Una pipeline è come una catena di montaggio: in entrambe, ciascun passo completa una parte dell'intero lavoro. I lavoratori di una catena di montaggio di automobili hanno un compito ristretto, ad esempio montare i rivestimenti dei sedili. Il vantaggio della catena di montaggio deriva dal fatto che molti lavoratori svolgono ognuno un compito piccolo per produrre collettivamente molte automobili al giorno. In una catena di montaggio ben bilanciata, una nuova automobile esce dalla catena nel tempo che serve per svolgere uno dei molti passi. Si osservi che la catena di montaggio non riduce il *tempo* di realizzazione di una singola automobile; essa aumenta il numero di automobili che vengono costruite simultaneamente e pertanto la *frequenza* con cui le automobili vengono iniziate e completate.

Come in una catena di montaggio, il lavoro che deve essere fatto in una CPU pipeline per eseguire un'istruzione è spezzato in piccole parti, ciascuna delle quali richiede una frazione del tempo necessario al completamento dell'intera istruzione. Ciascuno di tali passi è chiamato stadio di pipeline o segmento di pipeline. Gli stadi sono giustapposti a formare la pipeline: le istruzioni entrano da una estremità, vengono elaborate attraverso gli stadi, escono dall'altro estremo. Il tempo necessario per far avanzare un'istruzione di un passo lungo la pipeline corrisponde idealmente ad un ciclo di clock. Poiché gli stadi di pipeline sono collegati in successione, devono tutti operare in modo sincrono e perciò la durata di un ciclo di clock è determinata dal tempo richiesto per lo stadio più lento della pipeline.

L'obiettivo dei progettisti è bilanciare la lunghezza di ciascuno stadio: se ciò non avviene, si crea un tempo d'attesa durante uno stadio. Se gli stadi sono perfettamente bilanciati, l'accelerazione dovuta al pipelining è pari al numero di stadi di pipeline; una CPU pipeline a cinque stadi è cinque volte più veloce della stessa CPU senza pipeline. Solitamente, comunque, gli stadi non sono perfettamente bilanciati. Inoltre, l'introduzione del pipelining determina qualche costo aggiuntivo. Così, l'intervallo di tempo fra il completamento di due istruzioni sulla macchina dotata di pipelining è superiore al valore minimo possibile, e l'incremento di velocità sarà minore del numero di stadi di pipeline introdotto (una pipeline a cinque stadi non riesce a quintuplicare le prestazioni).

Per rendere la discussione più concreta, concentriamo l'attenzione su un insieme limitato di sette istruzioni: caricamento parola (`lw`), memorizzazione parola (`sw`), somma (`add`), sottrazione (`sub`), and (`and`), or (`or`), e salto condizionato (`beq`). I tempi di completamento delle diverse istruzioni sono riportati

in tabella 6. Il progetto della CPU senza pipeline deve tener conto dell'istruzione più lenta tra quelle riportate nella tabella 6, perciò il tempo richiesto per ciascuna istruzione è di 40 ns. L'esecuzione di una sequenza di istruzioni di caricamento sarebbe simile a quanto illustrato in figura 2. Pertanto, il tempo tra l'inizio della prima istruzione e il completamento della terza istruzione è di 3 × 40 ns, cioè 120 ns.

Nel paragrafo 3.1, a pagina 10, le istruzioni erano suddivise al massimo in cinque passi, perciò una pipeline a cinque stadi può essere un buon punto di partenza. Cinque stadi significa che vengono eseguite cinque istruzioni simultaneamente, con un'istruzione in ognuno degli stadi della pipeline. Ciascuno stadio di pipeline ha una durata pari a un ciclo di clock. La sequenza di istruzioni di caricamento, nella realizzazione con pipeline, viene trasformata nel modo indicato in figura 11. Tutti gli stadi hanno una durata prefissata (*ciclo di pipeline*), che deve essere sufficientemente lunga da consentire l'esecuzione dell'operazione più lenta, quindi una realizzazione organizzata a pipeline deve adottare un ciclo di almeno 10 ns, sebbene alcuni stadi richiedano solo 5 ns. Questo consente comunque un miglioramento di quattro volte.

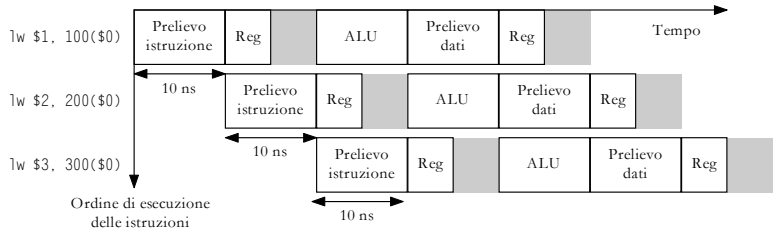


Figura 11 Esecuzione in pipeline della sequenza di operazioni presentata in figura 2.

Sebbene il miglioramento ideale sia di quattro volte, il tempo di esecuzione totale delle tre istruzioni riportate nelle figure 2 e 11 ha un miglioramento più modesto: 70 ns invece di 120 ns. Questa differenza è provocata dal tempo necessario a riempire la pipeline. Si pensi a quello che succede quando cresce il numero di istruzioni, per esempio con 1003 istruzioni il tempo totale di esecuzione diventa $1000 \times 10 \text{ ns} + 70 \text{ ns} = 10\,070 \text{ ns}$ nella pipeline, mentre senza pipeline si ottiene un tempo di $1000 \times 40 \text{ ns} + 120 \text{ ns} = 40\,120 \text{ ns}$. In queste condizioni, il rapporto tra i tempi totali di esecuzione dei programmi su macchine senza pipeline e su macchine organizzate a pipeline è vicino al limite ideale (stabilito dal rapporto dei tempi di completamento delle istruzioni):

$$\frac{40\,120 \text{ ns}}{10\,070 \text{ ns}} = 3.98 \cong \frac{40 \text{ ns}}{10 \text{ ns}}$$

Il pipelining è una tecnica di realizzazione che sfrutta il parallelismo tra le istruzioni di un flusso di esecuzione sequenziale. Essa ha il sostanziale vantaggio che, al contrario di altre tecniche di accelerazione, può essere completamente invisibile agli occhi del programmatore.

4.2 Un'unità di elaborazione organizzata a pipeline

La suddivisione dell'esecuzione di un'istruzione in cinque stadi (cioè l'organizzazione della CPU in una pipeline a cinque stadi) significa che, complessivamente, ad ogni singolo ciclo di clock ci saranno in esecuzione cinque istruzioni. Pertanto, occorre scomporre l'unità di calcolo in cinque parti, come illustrato nella figura 12. La figura 12 mostra l'unità di calcolo della figura 10 suddivisa in cinque parti, ciascuna corrispondente ad una delle fasi dell'esempio precedente; ognuna di queste parti prende il nome dalla stadio d'esecuzione dell'istruzione.

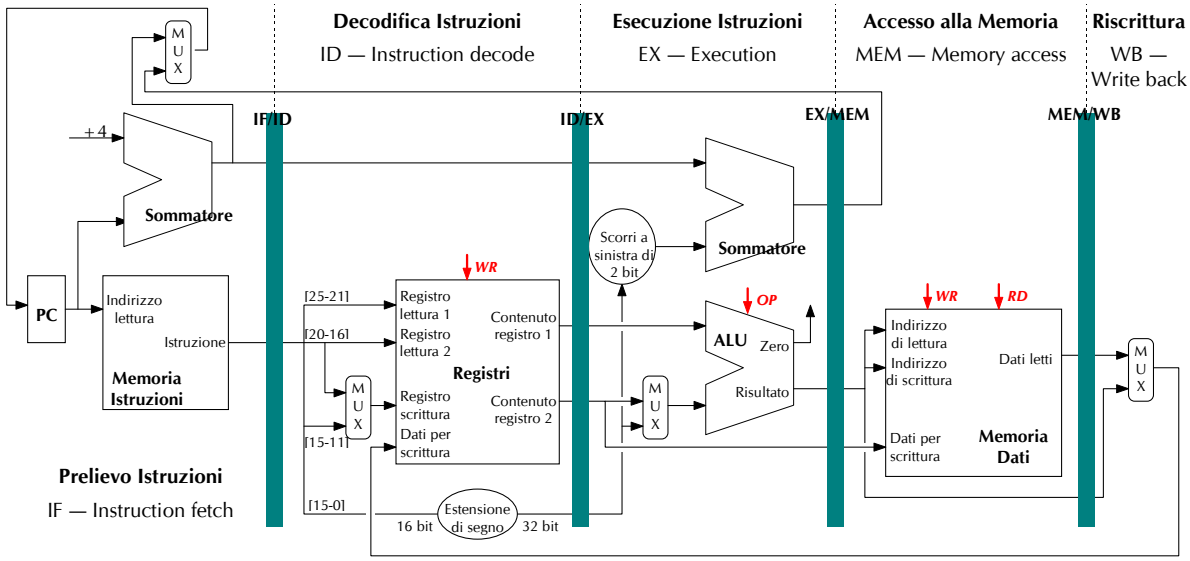


Figura 12 Versione organizzata a pipeline dell'unità di elaborazione riportata in figura 10. Ciascun passo di una istruzione può essere rappresentato come un trasferimento di dati da sinistra verso destra. Le uniche eccezioni sono i passi di aggiornamento del PC e di riscrittura del risultato; quest'ultimo invia a sinistra il risultato della ALU oppure un dato proveniente dalla memoria, per poterlo scrivere nel registro destinazione. I registri di pipeline, in colore, separano i diversi stadi. Questi registri devono essere sufficientemente grandi da poter memorizzare tutti i dati corrispondenti alle linee che li attraversano.

Nella struttura riportata in figura 12 c'è però un errore: l'istruzione memorizzata nel registro di pipeline IF/ID fornisce il numero del registro di scrittura, mentre i dati che vengono scritti sono quelli relativi all'istruzione che si trova nel registro MEM/WB. Quindi occorre trasmettere attraverso i registri di pipeline anche il numero del registro in cui debbono essere scritti i dati. La figura 13 mostra la versione corretta dalla CPU che trasferisce il numero di registro di scrittura prima al registro ID/EX, poi al registro EX/MEM, e infine al registro MEM/WB. Il numero di registro è utilizzato durante lo stadio WB per specificare il registro da scrivere.

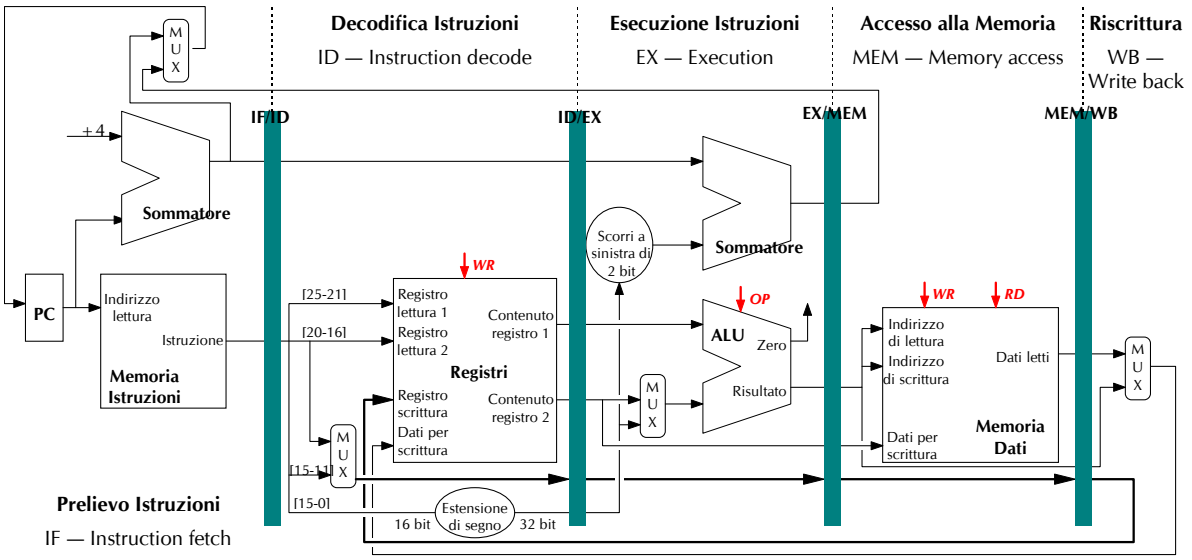


Figura 13 Unità di elaborazione corretta per l'istruzione di caricamento. Il numero del registro di scrittura proviene ora dal registro di pipeline MEM/WB insieme ai dati. Questo numero di registro viene fatto passare attraverso i diversi stadi intermedi fino a raggiungere il registro di MEM/WB. Il nuovo percorso è indicato dalle linee di maggior spessore.

4.3 Conflitti di dati

Se le istruzioni eseguite nella pipeline sono tutte indipendenti tra di loro, non sorgono problemi. Si consideri invece una sequenza di istruzioni in cui esistono numerose dipendenze, messe in evidenza dal carattere in grassetto:

```
sub     $2, $1, $3          # Il registro $2 è scritto dalla istruzione sub
and    $12, $2, $5          # Il 1° operando($2) dipende dalla istruzione sub
or     $13, $6, $2          # Il 2° operando($2) dipende dalla istruzione sub
add    $14, $2, $2          # 1°($2) & 2°($2) dipendono dalla istruzione sub
sw     $15,100($2)          # L'indice($2) dipende dalla istruzione sub
```

Le ultime quattro istruzioni dipendono tutte dal risultato scritto nel registro $2 dalla prima istruzione. Se il contenuto del registro prima dell'esecuzione dell'istruzione di sottrazione fosse 10 ed in seguito -20, il programmatore vorrebbe utilizzare il valore -20 nelle istruzioni che fanno riferimento a $2. La figura 14 illustra l'esecuzione di queste istruzioni in un'architettura a pipeline. Per comprendere meglio le modalità di esecuzione di questa sequenza, nella parte bassa della figura 14 è riportato il valore che il registro $2 assume all'inizio di ogni ciclo di clock.

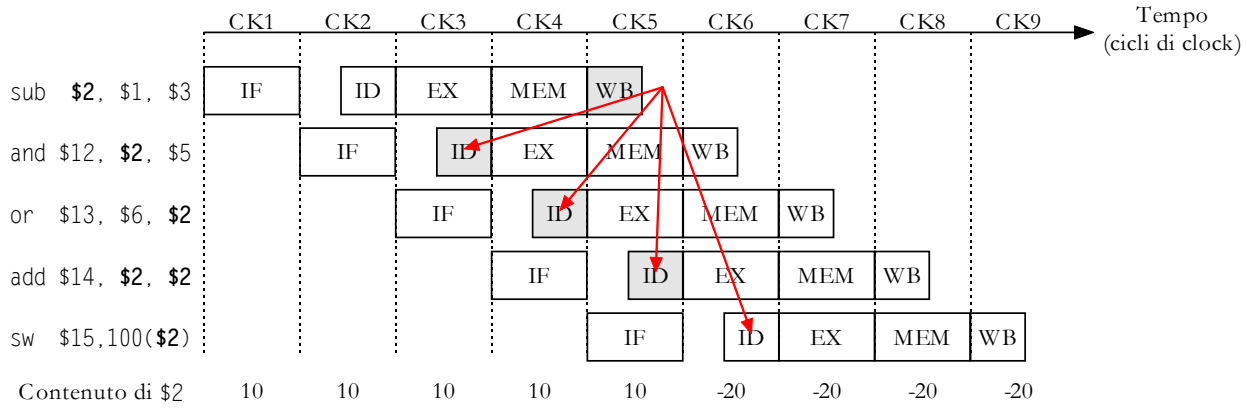


Figura 14 Sequenza di istruzioni correlate tra loro. Le dipendenze sono evidenziate con il grassetto, il tempo è indicato in termini di cicli di clock (*CKi* indica l'*i*-esimo ciclo di clock). La prima istruzione scrive nel registro $2, le istruzioni successive leggono $2, ma il registro viene aggiornato nel ciclo di clock numero 5, quindi il valore corretto non è disponibile prima del ciclo di clock 6. Le frecce che vanno dalla prima riga alle successive mostrano le dipendenze. Quelle che dirette verso cicli di clock precedenti indicano la presenza di *conflitti di dati* nella pipeline.

Per mantenere un corretto ordinamento temporale, il banco dei registri viene considerato come se fosse separato in due parti: i registri letti durante la fase ID e il registro scritto durante la fase WB. Questa separazione ha significato poiché le due parti sono logicamente unite solamente quando lo stesso registro viene letto e scritto da parte della stessa istruzione. Per ora è utile pensare alla parte letta e a quella scritta come a due elementi separati.

La figura 14 mostra come i valori letti nel registro $2 *non* corrispondano al risultato dell'istruzione `sub`, a meno che la lettura non avvenga nel ciclo di clock 6 o in cicli successivi. L'unica istruzione che accede al valore corretto (-20) è l'istruzione finale di memorizzazione; `and`, `or` e `add` utilizzano tutte il valore errato 10. Di conseguenza, ci sono problemi con le istruzioni `and`, `or` e `add` poiché queste dipendono da un valore scritto in seguito. Tali dipendenze prendono il nome di *conflitti sui dati*.

4.3.1 Le possibili soluzioni: impedire che istruzioni correlate siano troppo vicine

Una semplice strategia che consente di risolvere il problema dei conflitti sui dati, consiste nel proibire l'esistenza di questi conflitti: il compilatore non può generare sequenze come quella presentata sopra. Ad esempio, il compilatore dovrebbe inserire tre istruzioni indipendenti tra l'istruzione `sub` e l'istruzione `and`, facendo così scomparire tutti i conflitti. Quando non riesce a trovare istruzioni indipendenti, il compilatore inserisce istruzioni che sono certamente indipendenti da tutte le altre: le

istruzioni nop. L'abbreviazione significa «*no operation*» (nessuna operazione), poiché nop non legge registri, né modifica dati, né scrive risultati.

Il seguente codice utilizza delle istruzioni nop per garantire la correttezza del risultato:

```
sub  $2, $1, $3
nop
nop
nop
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw  $15, 100($2)
```

Le tre istruzioni nop occupano tre cicli di clock senza produrre risultati utili. Nel caso ideale il compilatore troverà istruzioni che effettuano calcoli utili con cui sostituire queste istruzioni d'attesa.

4.3.2 Le possibili soluzioni: inserire delle "bolle" nella pipeline

L'approccio hardware più semplice per risolvere conflitti di dati consiste nel bloccare il flusso di istruzioni nella pipeline fino a quando il conflitto non è risolto. Nell'esempio di figura 14 questo significa fermare le istruzioni *che seguono* l'istruzione sub finché i dati non possano essere letti. I primi progettisti di CPU pipeline utilizzarono il nome "*bolla*" per indicare il blocco delle istruzioni nella pipeline[3]. Per indicare lo stato in cui si trova la CPU quando le istruzioni sono bloccate si utilizza il termine di "*stallo*".

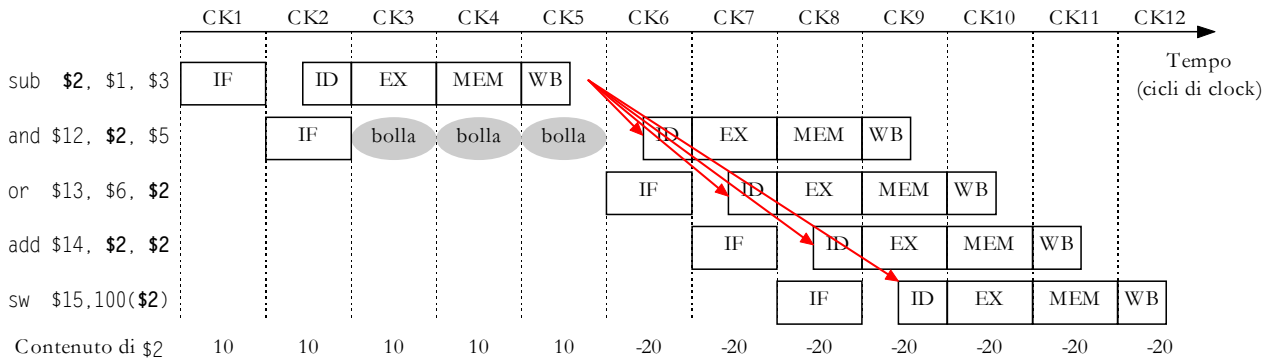


Figura 15 Sequenza di istruzioni mostrata in figura 14 con tre bolle inserite per risolvere il conflitto di dati. Si noti che risolvendo il conflitto dati per l'istruzione and, le bolle risolvono anche i conflitti per tutte le istruzioni che *seguono* l'istruzione and.

Il conflitto di dati si verifica quando un'istruzione, durante il proprio stadio ID, cerca di leggere un registro che una delle istruzioni precedenti ha intenzione di scrivere nel proprio stadio WB. Perciò, per risolvere un conflitto di dati basta bloccare l'istruzione che si trova nello stadio ID e dipende da una delle istruzioni immediatamente precedenti. Il blocco deve essere mantenuto finché l'istruzione che causa la dipendenza non viene completata. La figura 15 mostra come l'inserimento di tre bolle prima dello stadio ID dell'istruzione and rimuova il problema presentato in figura 14.

Se l'istruzione nello stadio ID viene bloccata, allora anche l'istruzione nello stadio IF deve essere bloccata, altrimenti l'istruzione prelevata andrebbe persa. Per impedire che queste istruzioni proseguano è sufficiente non cambiare né il registro PC né il registro di pipeline IF/ID. Purché il contenuto di questi registri venga conservato, l'istruzione nello stadio IF continuerà ad essere letta utilizzando lo stesso PC, ed i registri nello stadio ID continueranno ad essere letti utilizzando la stessa istruzione nel registro di pipeline IF/ID. La figura 15 è una rappresentazione sintetica di ciò che accade realmente nella CPU. Come una bolla d'aria in un condotto per l'acqua, una bolla di stallo procede nella CPU e ne

[3] Si ricordi che il termine *pipeline* ha il significato di oleodotto, dove le *bolle* d'aria scorrono attraverso i tubi, ma ne riducono la portata.

esce allo sbocco finale. L'istruzione `and` resta nel registro di pipeline IF/ID per tre cicli di clock, mentre vengono generate tre bolle separate nel condotto, come mostrato in figura 16.

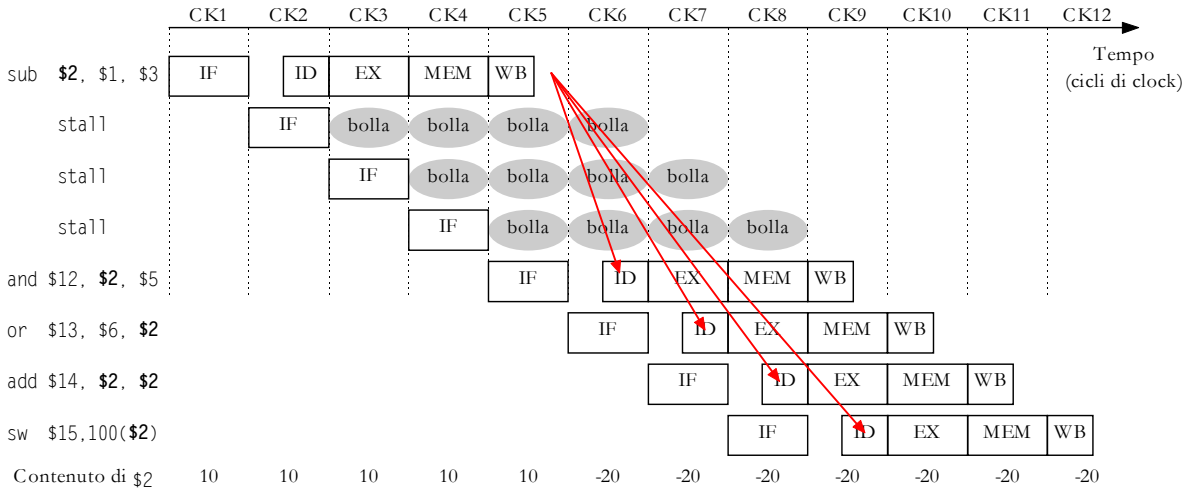


Figura 16 Versione della figura 15 che mostra il modo in cui le bolle vengono effettivamente inserite nel flusso di istruzioni che attraversano la pipeline. Poiché le dipendenze ora riguardano istanti successivi alla modifica del registro, non ci sono più conflitti.

4.3.3 Le possibili soluzioni: la propagazione in avanti dei dati

Un'analisi più attenta della figura 14 consente di osservare che il valore richiesto dall'istruzione `and`, è già disponibile nel campo *ALUResult* del registro di pipeline EX/MEM dell'istruzione `sub`. In modo analogo, l'ingresso della ALU per l'istruzione `or` successiva può essere trovato nel registro di pipeline MEM/WB dell'istruzione `sub`. Se si modifica il banco dei registri in modo tale che esso fornisca il valore appena scritto quando lettura e scrittura riguardano lo stesso registro, l'unità di calcolo può fornire anche l'operando per l'istruzione `add`. La figura 17 mostra le dipendenze tra i registri delle pipeline e gli ingressi della ALU per la stessa sequenza di codice riportata in figura 14.

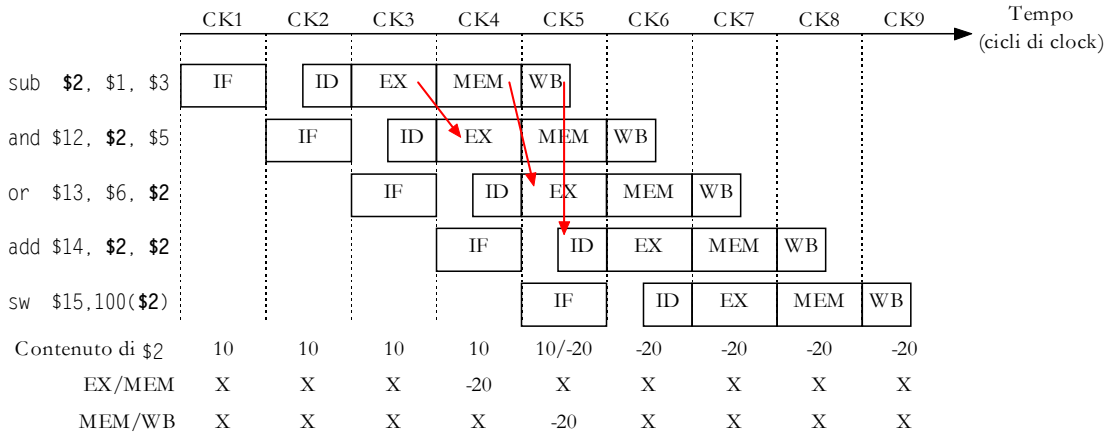


Figura 17 Le dipendenze tra i diversi registri di pipeline sono tutte orientate in avanti, si riferiscono cioè a istanti di tempo futuri, quindi è possibile fornire alla ALU gli ingressi richiesti dalle istruzioni `and` e `or` propagando in avanti i risultati che si trovano già nei registri di pipeline, senza aspettare che questi risultati vengano scritti nel banco di registri.

La modifica sta nel fatto che la dipendenza inizia da un registro di *pipeline* invece di attendere che lo stadio WB scriva i registri reali. I dati richiesti sono presenti nei registri di pipeline per essere utilizzati in future istruzioni, suggerendo una scorciatoia che possa ridurre il calo delle prestazioni dovute al ricorso alle bolle.

Se è possibile prelevare gli ingressi della ALU da qualsiasi registro di pipeline e non solo dal registro ID/EX, la pipeline può procedere senza stalli. Questa tecnica, che utilizza risultati temporanei invece di attendere che i registri vengano scritti, si chiama *propagazione in avanti dei risultati* (*forwarding*) o *bypassing*.

Aggiungendo multiplexer agli ingressi della ALU e utilizzando un apposito meccanismo di controllo è possibile far funzionare la pipeline a piena velocità anche in presenza di conflitti di dati.

4.4 I conflitti provocati dai salti condizionati

Fino ad ora l'attenzione è stata rivolta ai conflitti che coinvolgono operazioni aritmetiche e trasferimenti dati. Un altro tipo di conflitto tipico delle strutture a pipeline riguarda però i salti condizionati. La figura 18 mostra una sequenza di istruzioni in cui si verifica un salto condizionato. Per alimentare la pipeline si deve prelevare un'istruzione ad ogni ciclo di clock, però la decisione relativa al salto condizionata non viene presa fino allo stadio MEM. Questo ritardo nel determinare l'istruzione corretta da prelevare viene chiamato *conflitto di controllo* o *conflitto di salto condizionato*, in contrasto con i *conflitti di dati* precedentemente esaminati.

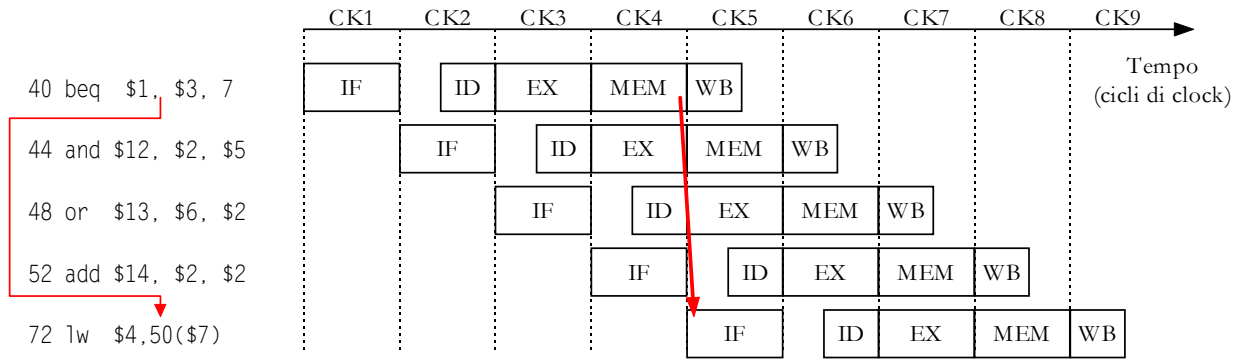


Figura 18 Esecuzione di un'istruzione di salto condizionato in una CPU pipeline. I numeri alla sinistra dell'istruzione (40, 44, ...) ne indicano l'indirizzo. Poiché l'istruzione di salto condizionato decide se effettuare tale salto nello stadio MEM (nel IV ciclo di clock dell'istruzione `beq`) le tre istruzioni immediatamente successive vengono prelevate e viene iniziata la loro esecuzione. Se non si interviene, queste tre istruzioni verranno completate prima dell'istruzione `lw` che si trova all'indirizzo di destinazione del salto (72).

4.4.1 Le possibili soluzioni: bloccare sempre la pipeline in uno stato di stallo

Una soluzione consiste nel bloccare il flusso di istruzioni (stallo della pipeline) fino a quando il salto condizionato non viene completato, come illustrato in figura 19. Questa soluzione comporta una penalizzazione di molti cicli di clock per ogni salto condizionato. Inoltre spesso la condizione non è verificata e dunque non si esegue il salto: il lavoro che si sarebbe potuto svolgere prelevando e decodificando le istruzioni successive deve essere comunque fatto.

4.4.2 Le possibili soluzioni: ipotizzare che il salto non venga eseguito

Un approccio migliore si basa sull'ipotesi che il salto non venga eseguito e che quindi si possa continuare l'esecuzione secondo il flusso naturale delle istruzioni. Nel caso in cui invece il salto debba essere eseguito, verranno scartate tutte le istruzioni prelevate e decodificate e l'esecuzione potrà proseguire dall'indirizzo di destinazione del salto. La figura 20 mostra come questa ottimizzazione cambi il flusso delle istruzioni. Se metà delle volte i salti non vengono eseguiti, e se è limitato il costo associato all'eliminazione delle istruzioni già prelevate, con questa ottimizzazione si riesce a dimezzare il costo dei conflitti di controllo.

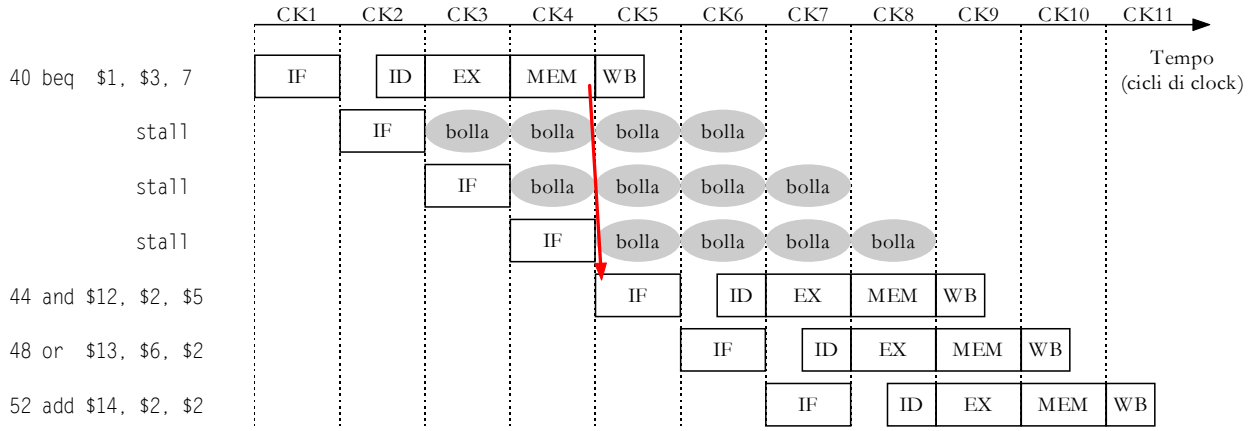


Figura 19 Salto condizionato con blocco delle istruzioni per risolvere il conflitto di controllo. La soluzione più semplice consiste nel bloccare tutte le istruzioni che seguono il salto condizionato fino a che la decisione non sia chiara, e poi proseguire con l'esecuzione delle istruzioni corrette. In pratica, utilizzare questa soluzione fa crescere da uno a quattro cicli di clock il costo di un salto condizionato.

Questo schema non è altro che una forma di previsione dei salti condizionati. In questo caso, si prevede che il salto non venga eseguito, svuotando la pipeline se si commette un errore. Con ulteriori strutture hardware è possibile utilizzare altri schemi di previsione dei salti condizionati. Per esempio è possibile esaminare l'indirizzo dell'istruzione e valutare se l'ultima volta che è stata eseguita il salto è stato effettuato o meno; in caso affermativo, si procede prelevando le nuove istruzioni dallo stesso indirizzo della volta precedente.

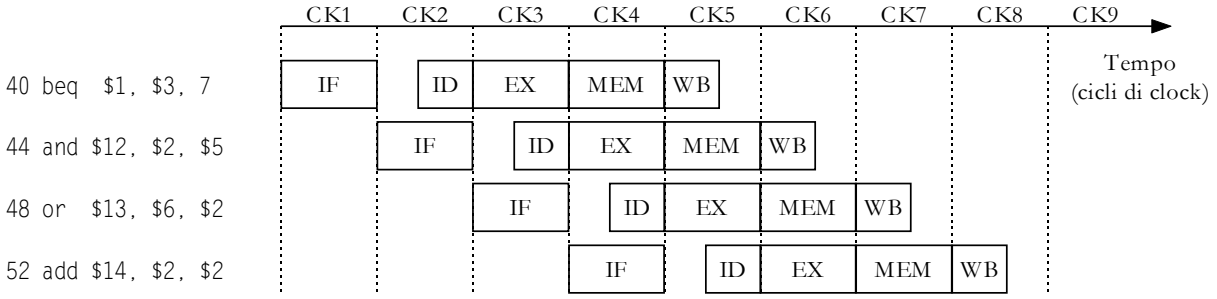


Figura 20 Contrariamente a quanto mostrato in figura 19, in questo approccio, quando il salto condizionato non viene eseguito, l'istruzione richiede un solo ciclo di clock. Solamente nel caso in cui il salto venga eseguito l'istruzione richiede quattro cicli di clock.

5. Le memorie cache

Fin dall'inizio, i programmatori hanno voluto avere a disposizione una quantità illimitata di memoria veloce. L'obiettivo delle tecniche che verranno illustrate qui di seguito è dare al programmatore l'illusione di poter usufruire di una memoria contemporaneamente veloce e grande. Queste tecniche si basano sul *principio di località*. Ci sono due diversi tipi di località:

- *Località temporale* (località nel tempo): quando si fa riferimento a un elemento, c'è la tendenza a far riferimento allo stesso elemento entro breve.
- *Località spaziale* (località nello spazio): quando si fa riferimento a un elemento, c'è la tendenza a far riferimento entro breve tempo ad altri elementi che hanno indirizzo vicino a quello dell'elemento corrente.

Per esempio, molti programmi contengono dei cicli, le cui istruzioni e i cui dati vengono richiesti ripetutamente, mostrando così un elevato livello di località temporale. Siccome inoltre le istruzioni di un programma vengono eseguite in sequenza, i programmi dimostrano un'elevata località spaziale. Anche l'accesso ai dati esibisce un comportamento intrinsecamente caratterizzato dalla località spaziale. Per esempio, è naturale che gli accessi agli elementi di una matrice o di un record mostrino un elevato grado di località spaziale.

Per sfruttare il principio di località, la memoria di un calcolatore viene realizzata come una *gerarchia di memoria*. Una gerarchia di memoria consiste in un insieme di livelli di memoria, ciascuno di diversa velocità e dimensione. A parità di capacità, le memorie più veloci sono più costose di quelle più lente, perciò sono di solito più piccole. La memoria principale viene in genere realizzata utilizzando componenti DRAM (dall'inglese *dynamic random access memory*, ovvero memoria dinamica ad accesso casuale), mentre i livelli più vicini alla CPU (cache) sono composti da SRAM (dall'inglese *static random access memory*, ovvero memoria statica ad accesso casuale). Il costo per bit della DRAM è più basso di quello della SRAM, anche se la prima è decisamente più lenta della seconda. La differenza di prezzo dipende dal fatto che le memorie DRAM utilizzano meno transistor per ogni bit da memorizzare, e così riescono a raggiungere capacità superiori a parità di area di silicio.

Viste le differenze di costo e di velocità, diventa conveniente costruire una memoria come una gerarchia di livelli, con la memoria più veloce posta più vicino al processore e quella più lenta, meno costosa, posta più distante. L'obiettivo è quello di fornire all'utente una quantità di memoria pari a quella disponibile nella tecnologia più economica, consentendo allo stesso tempo una velocità di accesso pari a quella garantita dalla tecnologia più veloce.

Una gerarchia di memoria può essere composta da più livelli, ma i dati vengono di volta in volta copiati solo tra due livelli adiacenti, perciò possiamo concentrare la nostra attenzione su due soli livelli. Il livello *superiore*—quello più vicino al processore—è più piccolo e veloce (infatti utilizza la tecnologia più costosa) del livello *inferiore*. La minima quantità di informazione che può essere presente o assente nella gerarchia composta da due livelli è indicata con il nome di *blocco*.

Se il dato richiesto dal processore compare in uno dei blocchi presenti nel livello superiore, si dice che la richiesta ha successo e si indica ciò con il termine inglese *hit*. Se il dato non si trova nel livello superiore, si dice che la richiesta fallisce, come indicato dal termine inglese *miss*. In questo caso, per trovare il blocco che contiene i dati richiesti, bisogna accedere al livello inferiore della gerarchia. La *frequenza dei successi* (in inglese *hit rate* oppure *hit ratio*) corrisponde alla frazione di accessi alla memoria che hanno trovato il dato desiderato nel livello superiore; spesso questo valore viene utilizzato come indice delle prestazioni della memoria gerarchica. La *frequenza dei fallimenti* (in inglese *miss rate* oppure *miss ratio*), pari a *1.0 – frequenza dei successi*, è la frazione di accessi alla memoria che non hanno trovato il dato desiderato nel livello superiore.

Siccome il motivo principale che ha condotto all'organizzazione gerarchica della memoria è l'aumento di prestazioni, la velocità degli accessi è importante in entrambi i casi di successo e di fallimento. Il *tempo di successo* (in inglese *hit time*) è il tempo di accesso al livello superiore della gerarchia di memoria, che comprende anche il tempo necessario a stabilire se il tentativo di accesso si risolve in un successo o in un fallimento. La *penalità di fallimento* è il tempo necessario a sostituire un blocco nel livello superiore con un altro blocco preso dal livello inferiore e a passare le informazioni contenute in questo nuovo blocco al processore. Siccome il livello superiore è più piccolo ed è costruito utilizzando componenti più veloci, il tempo di successo è molto inferiore al tempo necessario per accedere al secondo livello della gerarchia, che invece rappresenta la componente principale della penalità di fallimento.

Nell'esame di una gerarchia di memoria basate sugli stessi principi di località sono quattro le domande a cui bisogna rispondere:
1. Dove si può mettere un blocco?
2. Dove si trova un blocco?
3. Quale blocco deve essere sostituito in caso di fallimento?
4. Cosa succede in caso di scrittura?

5.1 Dove si può mettere un blocco?

Se ogni parola può andare in un solo posto della cache, allora sappiamo come trovarla, ammesso che si trovi effettivamente nella cache. La maniera più semplice per assegnare una locazione della cache ad ogni parola della memoria consiste nel definire una corrispondenza tra l'indirizzo in memoria della

parola e la locazione nella cache. Questa organizzazione della cache è detta a *indirizzamento diretto*, dato che ogni locazione di memoria corrisponde esattamente a una locazione della cache.

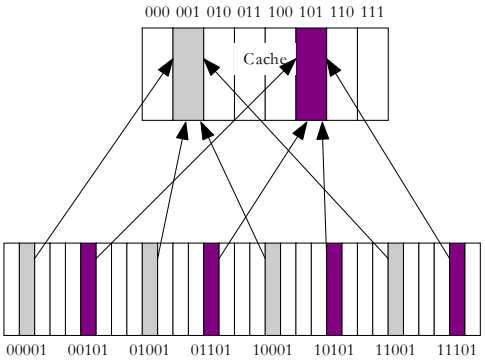


Figura 21 Esempio di cache a indirizzamento diretto con 8 elementi che illustra quali indirizzi di memoria compresi tra 0 e 30 corrispondono alle stesse locazioni della cache. Siccome la cache contiene 8 parole, un indirizzo *X* corrisponde all'elemento *X modulo 8* della cache. Cioè i $\log_2(8)=3$ bit meno significativi sono utilizzati come indice della cache. Perciò tutti gli indirizzi 00001_{due}, 01001_{due}, 10001_{due} e 11001_{due} corrispondono all'elemento 001_{due} della cache, mentre le parole di indirizzo 00101_{due}, 01101_{due}, 10101_{due} e 11101_{due} sono caricate nell'elemento 101_{due} della cache.

Per le cache a indirizzamento diretto, la corrispondenza tra indirizzo di memoria e locazione nella cache è di solito stabilita dalla seguente formula:

indirizzo del blocco *modulo* numero dei blocchi nella cache

Una corrispondenza realizzata secondo questa tecnica è molto comoda perché se il numero degli elementi nella cache è una potenza di due, allora l'operazione di modulo può essere effettuata semplicemente considerando i *$\log_2$(dimensione della cache in termini di blocchi)* bit meno significativi. Per esempio, la figura 21 mostra una cache a indirizzamento diretto di otto parole e gli indirizzi di memoria compresi tra 1 (00001_{due}) e 29 (11101_{due}) che corrispondono alle locazioni 1 (001_{due}) e 5 (101_{due}) della cache.

In alternativa si può prevedere invece che un blocco possa essere messo in un qualsiasi posto della memoria cache, questo schema viene chiamato *completamente associativo*.

In realtà esiste una serie di possibilità intermedie tra lo schema a indirizzamento diretto e quello completamente associativo. Gli schemi intermedi sono definiti come *set-associativi*. In una cache set-associativa, ogni blocco può essere messo in un numero prefissato di posizioni (almeno due); una cache set-associativa in cui un blocco può andare in *n* posizioni viene definita set-associativa a *n* vie. Una cache set-associativa a *n* vie è costituita da numerosi insiemi, ognuno dei quali comprende *n* blocchi. Ogni blocco della memoria corrisponde ad un unico *insieme* della cache ed il blocco può essere messo in uno *qualsiasi* degli elementi di questo insieme. Quindi un piazzamento set-associativo combina la modalità a indirizzamento diretto con quella completamente associativa: un blocco viene indirizzato all'insieme in maniera diretta, poi tutti i blocchi nell'insieme vengono esaminati per verificare la corrispondenza. Nella cache set-associativa, l'*insieme* che contiene il blocco viene individuato da

indirizzo del blocco *modulo* numero degli *insiemi* nella cache

Siccome il blocco può essere messo in un qualsiasi elemento dell'insieme, la ricerca deve essere effettuata su tutti questi elementi. In una cache completamente associativa, il blocco può essere messo in un posto qualsiasi, perciò tutti i blocchi della cache debbono essere esaminati durante la ricerca. La figura 22 mostra i posti dove il blocco 12 può essere messo in una cache che ha un totale di otto blocchi, a seconda che la politica di piazzamento sia a indirizzamento diretto, set-associativa o completamente associativa.

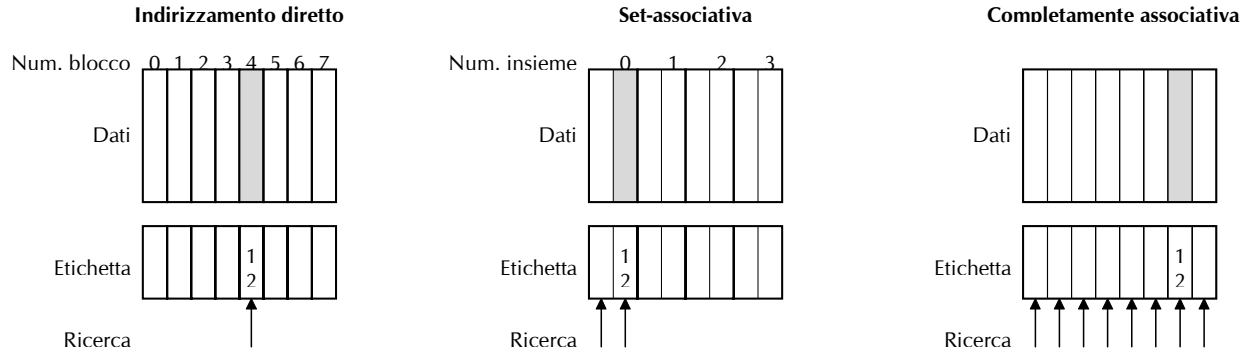
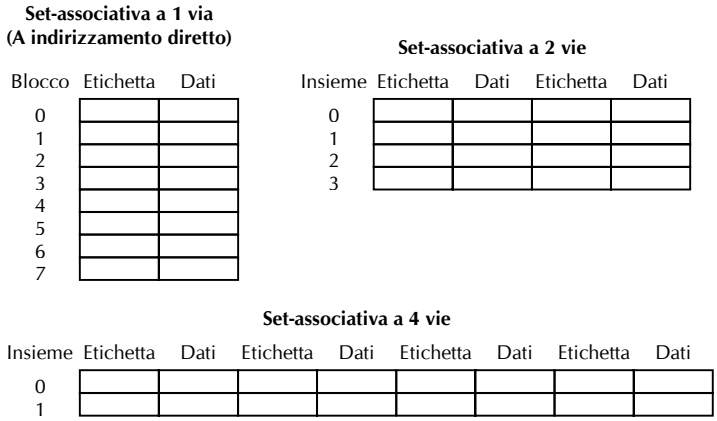


Figura 22 Il piazzamento del blocco di indirizzo 12 varia per le cache a indirizzamento diretto, set-associative e completamente associative. Nel piazzamento a indirizzamento diretto il blocco 12 può essere trovato in un solo blocco della cache e il numero di questo blocco si ottiene da (12 modulo 8) = 4. Nella cache set-associativa a due vie ci saranno quattro insiemi, ed il blocco 12 della memoria deve trovarsi nell'insieme (12 mod 4) = 0, in uno qualsiasi degli elementi che compongono quest'insieme. Nel piazzamento completamente associativo, il blocco di indirizzo 12 può trovarsi in uno qualsiasi degli otto blocchi.



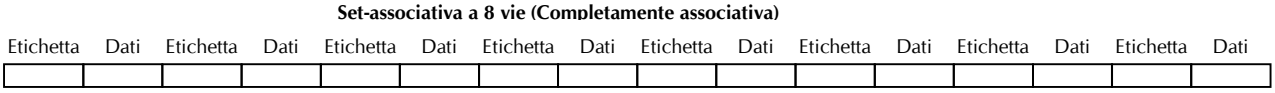


Figura 23 Una cache di otto blocchi configurata come cache a indirizzamento diretto, set-associativa a due vie, set-associativa a quattro vie e completamente associativa. La dimensione complessiva della cache in termini di blocchi è uguale al numero degli insiemi moltiplicato per l'associatività (ovvero la dimensione degli insiemi). Perciò, definita la dimensione della cache, al crescere dell'associatività diminuisce il numero degli insiemi, mentre cresce il numero di elementi compresi in un insieme. Con otto blocchi, una cache set-associativa a otto vie è uguale a una cache completamente associativa, anche se per cache di dimensioni reali queste due organizzazioni sono molto diverse.

Si può addirittura pensare che ogni politica di piazzamento sia una variazione di quella set-associativa. Una cache a indirizzamento diretto è semplicemente una cache set-associativa a una via: ogni elemento della cache contiene un blocco e forma un insieme di un solo elemento. Una cache di *m* elementi, completamente associativa, è semplicemente una cache set-associativa a *m* vie: c'è un solo insieme di *m* blocchi e un elemento può trovarsi in uno qualsiasi dei blocchi dell'insieme. La figura 23 mostra le strutture possibili per una cache di otto blocchi.

5.2 Dove si trova un blocco?

Poiché ogni elemento della cache può contenere le parole memorizzate in diverse locazioni di memoria, per sapere se una certa parola si trova nella cache bisogna aggiungere un insieme di *etichette* (in inglese *tag*) alla cache. Queste etichette contengono le informazioni necessarie a verificare se le parole presenti nella cache corrispondono o meno alla parola cercata.

Inoltre è necessario disporre di un metodo per riconoscere che un blocco della cache non contiene informazioni valide. Per esempio, quando un processore parte la cache è vuota e le informazioni nelle etichette non hanno nessun significato. La procedura più comune consiste nell'aggiungere un *bit di validità* per indicare se il corrispondente elemento della cache contiene dei dati validi. Se il bit non è attivato, allora non c'è nessun indirizzo della memoria che corrisponde a questo elemento della cache.

Indirizzo decimale del dato in memoria	**Indirizzo binario del dato in memoria**	**Successo o fallimento nell'accesso alla cache**	**Blocco della cache assegnato (dove trovare o mettere i dati)**
22	10110_{due}	Fallimento (8b)	$(10110_{due} \bmod 8) = 110_{due}$
26	11010_{due}	Fallimento (8c)	$(11010_{due} \bmod 8) = 010_{due}$
22	10110_{due}	Successo	$(10110_{due} \bmod 8) = 110_{due}$
26	11010_{due}	Successo	$(11010_{due} \bmod 8) = 010_{due}$
16	10000_{due}	Fallimento (8d)	$(10000_{due} \bmod 8) = 000_{due}$
4	00100_{due}	Fallimento (8e)	$(00100_{due} \bmod 8) = 100_{due}$
16	10000_{due}	Successo	$(10000_{due} \bmod 8) = 000_{due}$
18	10010_{due}	Fallimento (8f)	$(10010_{due} \bmod 8) = 010_{due}$

Tabella 7 Sequenza delle richieste fatte dal processore al sistema di memoria che provoca l'evoluzione della cache riportata in tabella 8.

La tabella 8 mostra il contenuto di una cache a indirizzamento diretto di otto parole mentre risponde a una serie di richieste che le giungono dal processore. Siccome ci sono otto blocchi nella cache, i 3 bit meno significativi indicano il numero di blocco. La sequenza di azioni compiuta per quanto riguarda i riferimenti alla memoria è riportata nella tabella 7.

Quando la parola che si trova all'indirizzo 18 (10010_{due}) viene portata nella cache, nel blocco 2 (010_{due}), la parola di indirizzo 26 (11010_{due}), che si trovava in precedenza nel blocco 2 (010_{due}) della cache, deve essere sostituita dai dati appena richiesti. Questo comportamento consente alla cache di trarre vantaggio dalla località temporale: le parole richieste più di recente sostituiscono quelle richieste meno di recente. In una cache a indirizzamento diretto c'è un solo posto dove poter inserire i dati appena richiesti e quindi non c'è nessuna possibilità di scegliere quale elemento sostituire.

Per ogni indirizzo possibile abbiamo quindi identificato un metodo per individuare in quale blocco della cache a indirizzamento diretto cercare la parola desiderata: i bit meno significativi dell'indirizzo possono essere utilizzati per individuare l'unico elemento della cache a cui l'indirizzo stesso corrisponde.

Indice	V	Etichetta	Dato
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. Stato iniziale della cache dopo l'accensione.

Indice	V	Etichetta	Dato
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	*S*	*10_{due}*	*Memoria(10110_{due})*
111	N		

b. Dopo la gestione di un fallimento di accesso a un indirizzo richiesto (10110_{due}).

Indice	V	Etichetta	Dato
000	N		
001	N		
010	*S*	*11_{due}*	*Memoria(11010_{due})*
011	N		
100	N		
101	N		
110	S	10_{due}	Memoria(10110_{due})
111	N		

c. Dopo la gestione di un fallimento di accesso a un indirizzo richiesto (11010_{due}).

Indice	V	Etichetta	Dato
000	*S*	*10_{due}*	*Memoria(10000_{due})*
001	N		
010	S	11_{due}	Memoria(11010_{due})
011	N		
100	N		
101	N		
110	S	10_{due}	Memoria(10110_{due})
111	N		

d. Dopo la gestione di un fallimento di accesso a un indirizzo richiesto (10000_{due}).

Indice	V	Etichetta	Dato
000	S	10_{due}	Memoria(10000_{due})
001	N		
010	S	11_{due}	Memoria(11010_{due})
011	N		
100	*S*	*00_{due}*	*Memoria(00100_{due})*
101	N		
110	S	10_{due}	Memoria(10110_{due})
111	N		

e. Dopo la gestione di un fallimento di accesso a un indirizzo richiesto (00100_{due}).

Indice	V	Etichetta	Dato
000	S	10_{due}	Memoria(10000_{due})
001	N		
010	*S*	*10_{due}*	*Memoria(10010_{due})*
011	N		
100	S	00_{due}	Memoria(00100_{due})
101	N		
110	S	10_{due}	Memoria(10110_{due})
111	N		

f. Dopo la gestione di un fallimento di accesso a un indirizzo richiesto (10010_{due}).

Tabella 8 Il contenuto della cache è illustrato dopo ogni tentativo di accesso risultato in un *fallimento*. I campi indice ed etichetta sono riportati in binario. All'inizio la cache è vuota con tutti i bit di validità (campo V) che hanno valore negativo (N). Il processore richiede i seguenti indirizzi: 10110_{due} (fallimento), 11010_{due} (fallimento), 10110_{due} (successo), 11010_{due} (successo), 10000_{due} (fallimento), 00100_{due} (fallimento), 10000_{due} (successo) e 10010_{due} (fallimento). Le figure mostrano il contenuto della cache dopo la gestione di ogni fallimento che si verifica nella sequenza. Quando il processore fa riferimento all'indirizzo 10010_{due} (18), il contenuto dell'elemento riservato alla parola di indirizzo 11010_{due} (26) deve essere sostituito, e una successiva richiesta dell'indirizzo 11010_{due} provocherà un nuovo fallimento. Si noti che il campo etichetta contiene solo la parte più significativa dell'indirizzo. L'indirizzo completo di una parola contenuta in questa cache, nel blocco *i*, con un campo etichetta pari a *j*, si ottiene da $8j+i$, oppure, in maniera equivalente, lo si ricava dalla concatenazione dei campi etichetta *j* e indice *i*. Per comprendere questo meccanismo basta esaminare l'indirizzo del blocco riportato nella colonna dei dati per ogni elemento della cache e confrontarlo con l'indice e l'etichetta corrispondenti. Per esempio nella cache riportata sopra al punto f, l'indice 010 ha come etichetta 10 e corrisponde all'indirizzo 10010.

La figura 24 illustra come un indirizzo è diviso in una parte che identifica l'indice della cache, utilizzato per selezionare il blocco, e un'altra che specifica l'etichetta, da confrontare con il contenuto del campo etichetta della cache. Siccome un certo indirizzo può comparire soltanto in una locazione, l'etichetta deve soltanto corrispondere alla parte più significativa dell'indirizzo, che non viene utilizzata per l'indice

della cache. Così, l'indice di un blocco della cache insieme al contenuto dell'etichetta di quello stesso blocco specificano in maniera completa l'indirizzo di memoria della parola contenuta nel blocco della cache. Siccome i bit della parte indice vengono utilizzati come indirizzo per accedere alla cache, il numero complessivo degli elementi della cache deve essere una potenza di 2. Nell'architettura MIPS, i due bit meno significativi di ogni indirizzo servono a specificare un byte all'interno di una parola e non vengono utilizzati per selezionare una parola nella cache.

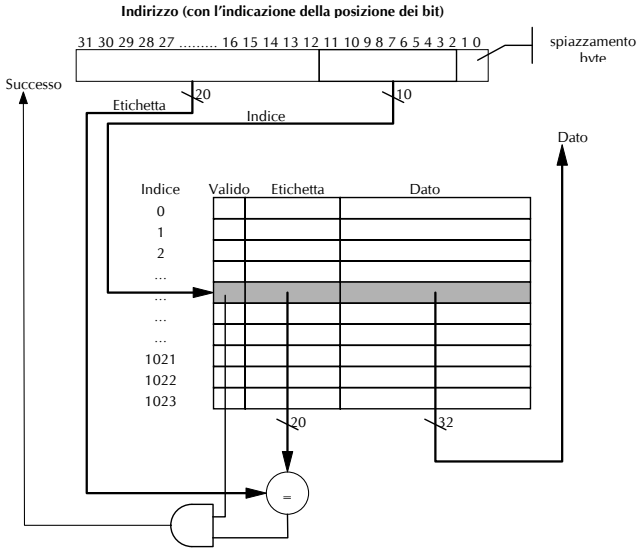


Figura 24 Per questa cache, la parte bassa dell'indirizzo viene utilizzata per selezionare un elemento della cache che consiste di una parola di dati e di un'etichetta. L'etichetta della cache viene confrontata con la parte alta dell'indirizzo per verificare se l'elemento della cache corrisponde all'indirizzo richiesto. Siccome la cache contiene 2^{10} (cioè 1024) parole e la dimensione di un blocco è di 1 parola, 10 bit sono utilizzati per l'indirizzo della cache, lasciando 32 – 10 – 2 = 20 bit per il confronto con l'etichetta. Se l'etichetta e i 20 bit più alti dell'indirizzo sono uguali e se il bit di validità ha valore positivo, allora la ricerca nella cache ha successo, e la parola viene fornita al processore. Altrimenti si verifica un fallimento.

La cache che abbiamo descritto fino ad ora, sebbene semplice, non sfrutta la località spaziale degli accessi visto che ogni parola ha un suo blocco. Come abbiamo già notato, la località spaziale è una caratteristica intrinseca dei programmi. Per trarre vantaggio dalla località spaziale è necessario che il blocco della cache sia più ampio, in modo da contenere più di una sola parola. Quando si verifica un fallimento, dalla memoria centrale vengono prelevate più parole adiacenti che hanno una alta probabilità di essere richieste a breve. La figura 25 mostra una cache che contiene 64 KB di dati, in blocchi di 4 parole (16 byte) ciascuno. In confronto alla cache di figura 24 nello schema di indirizzamento della cache illustrata in figura 25 è necessario un campo aggiuntivo per l'indice del blocco. Questo campo di indice nel blocco viene utilizzato per controllare il multiplexer (che si trova in basso nella figura), in modo da selezionare la parola richiesta tra le quattro parole che si trovano nel blocco individuato. Il numero totale di etichette è inferiore nella cache con blocchi di più parole, perché ogni etichetta è utilizzata per quattro parole. Questo migliora il grado di efficienza di utilizzo della memoria nella cache.

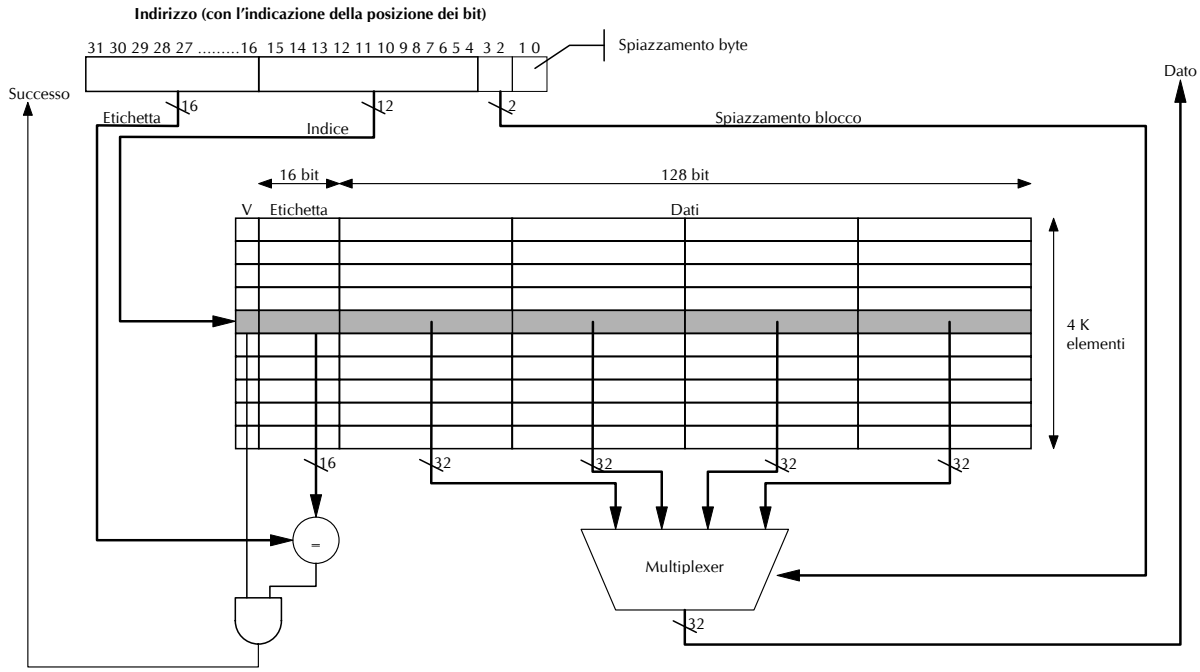


Figura 25 Cache di 64 KB che utilizza blocchi di quattro parole (16 byte). Il campo etichetta è largo 16 bit, quello indice è di 12 bit, mentre un campo di due bit (i bit 3–2) è utilizzato come indice del blocco e serve a selezionare la parola richiesta dal blocco utilizzando un multiplatore 4-a-1. In realtà, i bit meno significativi dell'indirizzo (in questo caso i bit 2 e 3) sono utilizzati per abilitare solo quelle RAM che contengono la parola desiderata. Questo elimina il bisogno del multiplexer. Questa tecnica funziona perché il valore dei bit che indicano lo spiazzamento del blocco è noto nello stesso istante in cui si hanno gli altri bit dell'indirizzo.

Consideriamo ora il compito di individuare un blocco in una cache set-associativa[4]. Ogni blocco della cache comprende un'etichetta che permette di individuare l'indirizzo del blocco. Per ogni blocco della cache che potrebbe contenere l'informazione cercata viene controllata l'etichetta per verificare se corrisponde all'indirizzo richiesto dalla CPU.

Etichetta	Indice	Spiazzamento nel blocco

Figura 26 Le tre parti di un indirizzo in una cache set-associativa oppure a indirizzamento diretto. L'etichetta viene utilizzata per controllare tutti i blocchi nell'insieme, mentre l'indice serve a identificare l'insieme. In una cache completamente associativa, il campo indice non serve (c'è un solo insieme). Lo spiazzamento nel blocco indica l'indirizzo dei dati desiderati all'interno del blocco.

La figura 26 illustra come viene scomposto l'indirizzo. Il valore dell'indice serve a selezionare l'insieme che contiene l'indirizzo desiderato, le etichette di tutti blocchi compresi in questo insieme debbono quindi essere controllate. Siccome è fondamentale la velocità di esecuzione, tutte le etichette dell'insieme selezionato vengono esaminate in parallelo. Una ricerca serializzata renderebbe troppo lento il tempo di successo in una cache set-associativa.

Se si mantiene costante la dimensione totale, al crescere dell'associatività aumenta anche il numero dei blocchi compresi nell'insieme, che corrisponde al numero dei confronti che debbono essere effettuati per realizzare una ricerca in parallelo: ogni volta che si raddoppia il grado di associatività, si raddoppia anche il numero di blocchi compresi in un insieme e si dimezza il numero degli insiemi. Analogamente, ogni incremento dell'associatività di un fattore due, fa diminuire di un bit la dimensione dell'indice e aumentare di un bit la dimensione dell'etichetta. Perciò, in una cache completamente associativa c'è un solo insieme e tutti i blocchi debbono essere esaminati in parallelo. Così non c'è indice, e tutto l'indirizzo, a parte lo spiazzamento nel blocco, viene confrontato con l'etichetta di ogni blocco. In altre parole, la ricerca viene effettuata su tutta la cache, senza nessun indice.

In una cache a indirizzamento diretto, come quella illustrata in figura 24, è necessario un solo comparatore, dato che l'elemento può essere solo in una posizione. In una cache set-associativa a

[4] Si ricordi che una cache completamente associativa può essere vista come una cache set-associativa con un solo insieme.

quattro vie, illustrata in figura 27, sono necessari quattro comparatori, oltre a un multiplexer da 4 a 1 per scegliere tra i quattro possibili membri dell'insieme selezionato. L'accesso alla cache si effettua utilizzando l'indice per individuare l'insieme giusto e poi esaminando gli elementi dell'insieme. Il costo di una cache associativa consiste nei comparatori aggiuntivi e nei ritardi imposti dalla necessità di confrontare e selezionare l'elemento desiderato tra quelli dell'insieme.

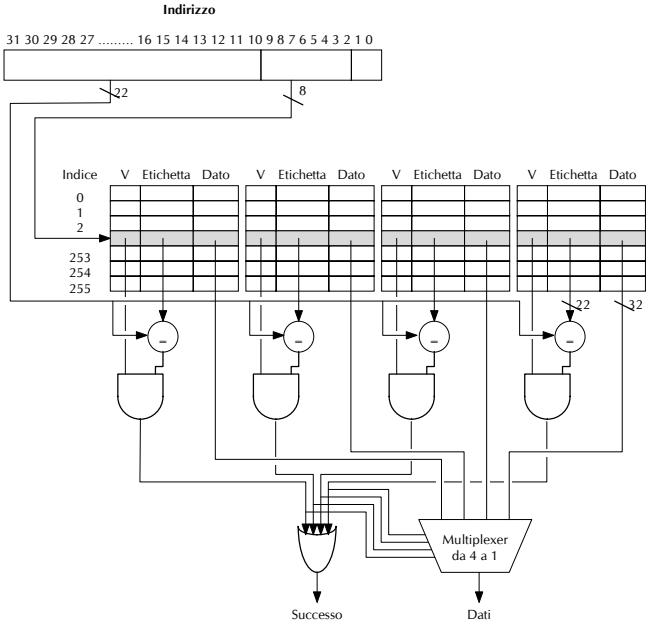


Figura 27 La realizzazione di una cache set-associativa a quattro vie richiede quattro comparatori e un multiplexer da 4 a 1. I comparatori individuano quale elemento dell'insieme (sempre che ne esista uno) corrisponde all'etichetta. L'uscita dei comparatori viene utilizzata per selezionare i dati da uno dei quattro insiemi, utilizzando un multiplexer. In alcune realizzazioni, il segnale di abilitazione dell'uscita, che si trova in quella parte dei chip dedicata alla memorizzazione dei dati, può essere utilizzato per selezionare l'elemento appartenente all'insieme che controlla l'uscita. Il segnale di abilitazione dell'uscita viene collegato direttamente all'uscita dei comparatori, in modo che solo l'elemento corrispondente alla richiesta riesca effettivamente a controllare l'uscita. Questo meccanismo elimina la necessità del multiplexer.

In ogni gerarchia di memoria, la scelta tra lo schema a indirizzamento diretto, quello set-associativo e quello completamente associativo dipende dal confronto tra il costo di un fallimento e quello di realizzazione dell'associatività, dal punto di vista sia del tempo che delle componenti aggiuntive.

5.3 Quale blocco sostituire in caso di fallimento durante l'accesso alla cache?

Quando si verifica un fallimento in una cache associativa, bisogna decidere quale blocco sostituire. In una cache completamente associativa ogni blocco è un potenziale candidato per la sostituzione. Se la cache è set-associativa, bisogna scegliere tra i blocchi compresi nell'insieme. Ovviamente la sostituzione è facile in una cache a indirizzamento diretto perché c'è un solo candidato.

Ci sono due principali strategie utilizzate per la scelta del blocco da sostituire:

- *A caso*: la scelta tra i blocchi candidati viene effettuata a caso, eventualmente utilizzando dei componenti hardware di supporto.
- *Utilizzato meno di recente* (in inglese *Last Recently Used — LRU*): il blocco sostituito è quello che è rimasto inutilizzato da più lungo tempo.

La selezione casuale offre il vantaggio di essere semplice da realizzare in hardware. Al crescere del numero di blocchi di cui bisogna tener traccia, la politica LRU diventa sempre più costosa e, in pratica, viene semplicemente approssimata. In una cache set-associativa a due vie, la sostituzione a caso ha una frequenza dei fallimenti pari a circa 1.1 volte quella ottenuta utilizzando la politica LRU. Al crescere delle dimensioni della cache, la frequenza dei fallimenti diminuisce per entrambe le strategie e la differenza in termini assoluti diventa minima. La sostituzione LRU presenta un vantaggio superiore con un più ampio grado di associatività, ma in quel caso è anche più difficile da realizzare.

5.4 Cosa succede in caso di scrittura?

Una caratteristica fondamentale di tutte le gerarchie di memoria è la gestione delle scritture. Le opzioni base sono due:

1. *Write-through*: l'informazione viene scritta nel blocco della cache *e* nel blocco della memoria principale.
2. *Write-back* (detto anche *copy back*): l'informazione viene scritta solo nel blocco della cache. Il blocco modificato viene scritto nel livello inferiore della gerarchia solo quando viene sostituito. Si noti che al termine della scrittura nella cache, la memoria conterrà un valore diverso da quello presente nella cache; in questo caso si dice che la memoria e la cache sono *inconsistenti* (cioè non sono coerenti).

Entrambi gli schemi hanno i relativi vantaggi. Gli aspetti positivi dell'opzione *write-back* sono:

- Le singole parole possono essere scritte dal processore alla frequenza a cui la cache, e non la memoria principale, è in grado di accettarle.
- Scritture multiple all'interno dello stesso blocco richiedono una sola scrittura al livello inferiore della gerarchia.
- Quando i blocchi vengono scritti, il sistema può trarre vantaggio dall'utilizzo di un'interfaccia più larga con il livello inferiore, visto che viene scritto un blocco intero. Un'interfaccia più larga consente anche di migliorare la gestione dei fallimenti in lettura.

Utilizzando lo schema *write-through*, si ottengono i seguenti vantaggi:

- I fallimenti in lettura sono meno costosi, infatti non richiedono mai la scrittura nel livello inferiore.
- È più facile realizzare uno schema *write-through* che uno *write-back*, anche se, per essere efficace in un sistema veloce, una cache *write-through* deve essere dotata anche di un tampone di scrittura.

Siccome le CPU continuano ad aumentare le loro prestazioni con un tasso di crescita superiore a quello delle memorie principali basate su DRAM, la frequenza con cui vengono generate le scritture da parte del processore sarà presto superiore a quella gestibile dal sistema di memoria, anche in presenza di memoria più larghe sia dal punto di vista fisico che da quello logico. Di conseguenza si può prevedere che, in futuro, sempre più spesso anche le cache utilizzeranno una strategia *write-back*.