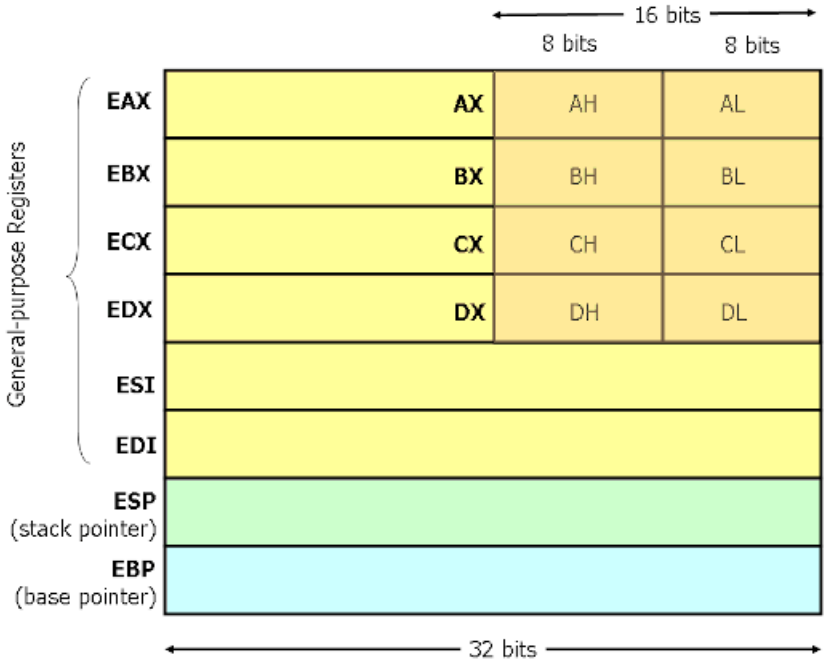


Architetture e programmazione dei sistemi di elaborazione

Indice

Capitolo 1

Introduzione

1.1 Architetture e programmazione dei sistemi di elaborazione

Lo scopo della materia è quello di fornire i principi base della progettazione dei moderni calcolatori elettronici, tra cui le macchine Multiciclo, Monociclo, Pipeline, Superscalare e Parallela.

In ogni PC coesistono varie forme di parallelismo, esso, infatti, può essere diviso in:

- **Implicito**: indicato con l'acronimo ILP (Instruction Level Parallelism), è la capacità della CPU di eseguire più parti del programma sequenziale in esecuzione;
- **Esplicito**: richiede l'intervento del programmatore secondo due diverse forme:
  - *MIMD*: più istruzioni vengono eseguite su più dati differenti, l'acronimo vuol dire Multiple Instruction Multiple Data;
  - *SIMD*: Single Instruction Multiple Data, indica un unica istruzione che opera su più dati. Ad esempio si ha un registro a 128 bit che viene utilizzato per memorizzare 4 floating point;

Si cerca inoltre di approfondire la conoscenza del linguaggio **Assembly** SIMD, con estensione SSE (SIMD Streaming Extensions), ossia con registri a 128 bit, oppure con la più recente estensione AVX (Advanced Vector Extensions) con registri vettoriali a 256 bit.

1.2 Calcolatore come macchina multilivello

La macchina è divisa in vari livelli, che sono:

- **Dispositivi**: sono i componenti elettronici del calcolatore, i transistor;
- **Logico digitale**: in questo livello si osservano le porte logiche di base, come ad esempio i registri e la ALU;
- **Microarchitettura**: in questo livello vengono assemblate tra loro le porte logiche in modo da ottenere un dispositivo più complesso, ossia la CPU;
- **Instruction Set Architecture**: l'ISA è il livello del repertorio d'istruzioni, sono presenti tutti i dettagli visibili dal programmatore a linguaggio macchina;
- **Sistema operativo**: contiene astrazioni come i processi e la memoria virtuale;
- **Linguaggio assembly**: indica il livello contenente le istruzioni del linguaggio assembly;
- **Linguaggio ad alto livello**: esistono vari linguaggi ad alto livello, tra questi java.

Di seguito verranno analizzati i livelli *Microarchitettura*, *ISA* e *Linguaggio Assembly*.

Questa suddivisione in livelli è avvenuta con l'aumentare della complessità dei calcolatori, mentre precedentemente non vi era una distinzione tra i livelli ISA, microarchitettura e logico digitale. Questa distinzione avvenne nei primi anni '50 con la Microprogrammazione (1951). Nei primi calcolatori si aveva una corrispondenza 1:1 tra hardware e istruzioni a disposizione, ma aggiungere funzioni vuol dire aggiungere componenti, andando ad incrementare le possibilità d'errore e di guasto del calcolatore.

La microprogrammazione tenta di ridurre l'aumento dell'hardware, esso infatti è un interprete hardware del linguaggio macchina, che si evolverà diventando Microarchitettura.

- **Architettura**: indica le caratteristiche del sistema visibili al programmatore in linguaggio macchina, ad esempio i registri, le istruzioni, il formato di queste ultime ecc;

- **Organizzazione**: coincide con la Microarchitettura, indica le relazioni strutturali che intercorrono tra le unità funzionali ed il modo in cui esse realizzano una data architettura, ossia come vengono realizzate le caratteristiche dell'architettura.

L'architettura sarebbe l'interfaccia mentre l'implementazione coincide con l'organizzazione. Questi due concetti vennero introdotti da IBM nel 1964 con l'immissione nel mercato dei calcolatori system/360, che differivano in prezzi e prestazioni ma erano tutti capaci di eseguire gli stessi programmi.

Con il termine **Retro compatibilità** si indica un sistema capace di eseguire programmi realizzati per le generazioni precedenti.

INTEL			
Architettura	**Processore**	**Anno**	**Bit**
x86	8086	1978	16
x86-32	80386	1995	32
x86-64	Pentium 4	2003/2005	64
IA-64	Itanium	2001	64

Tabella 1.1: Tabella d'evoluzione dell'architettura.

Venne successivamente aggiornata l'architettura con x86-32, in cui si raddoppiano i registri ma si dovette necessariamente supportare la retro compatibilità come estensione dell'architettura precedente, ampliando i registri da AX ad EAX (Extended AX) ad esempio.

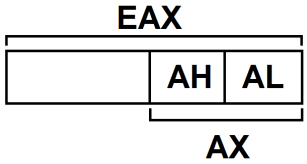


Figura 1.1: Estensione del registro AX in EAX.

Anche con l'architettura x86-64 si ha la retro compatibilità dato che i registri EAX sono stati estesi ulteriormente, passando da 32 bit a 64 bit diventando i registri RAX. Tutt'oggi si utilizza l'architettura basata su x86-64.

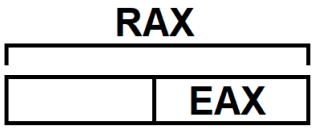


Figura 1.2: Estensione del registro EAX in RAX.

L'architettura IA-64 nasce per rimuovere o correggere tutte le operazioni diventate obsolete, ma non ha avuto molto successo questa scelta.

A causa della retro compatibilità, i processori sono molto più complessi del dovuto, con intere parti inutili dedicate alla sola retro compatibilità che non sarebbero presenti altrimenti nelle architetture moderne.

1.3 Evoluzione della tecnologia dei calcolatori

L'evoluzione della tecnologia dei calcolatori è dovuta sostanzialmente a due tipi di progressi:

- **Progressi nell'organizzazione e nell'architettura** (relativi ai livelli ISA ecc.);

- **Progressi nei dispositivi.**

1.3.1 Progressi nei dispositivi

I progressi nei dispositivi sono suddivisi in due ambiti principali che sono la *Tecnologia costruttiva* e la *Frequenza di funzionamento*.

Nella **Tecnologia costruttiva**, si suddividono i calcolatori in generazioni sulla base della tecnologia adottata.

- **Generazione zero:** (1642-1945) inizia nel 1642 con la nascita della prima calcolatrice di Pascal, si identifica con questa generazione la nascita dei computer meccanici;
- **Prima generazione:** (1945-1955) nel 1945 si ha la nascita del *Colossus* creato in Gran Bretagna alla fine della seconda guerra modiale, si introducono in questa generazione le Valvole;
- **Seconda generazione:** (1955-1965) vengono introdotti in questa generazione i Transistor;
- **Terza generazione:** (1965-1980) vengono inseriti più transistor all'interno di una stessa piastrina, creando i circuiti integrati;
- **Quarta generazione:** (1980-Oggi) vengono inseriti molti componenti all'interno dei calcolatori, Very Large Scale Integration (VLSI).

La scala indica il numero di componenti che si riesce a posizionare su un singolo chip e viene chiamata **Scala d'integrazione**.

La **Legge di Moore** descrive il tasso della crescita delle scale d'integrazione formulate nel 1965. Seconda questa legge la capacità d'elaborazione (ossia del numero di transistor all'interno del singolo chip) raddoppia ogni 12 mesi. La legge ha subito delle rivisitazioni infatti si sosteneva 12 mesi nel 1965, poi 24 mesi nel 1975, fino ad arrivare alla formulazione finale che afferma che la capacità d'elaborazione raddoppia ogni 18 mesi (versione del 1980).

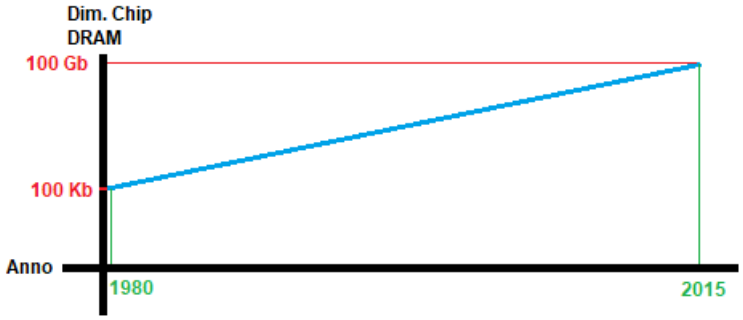


Figura 1.3: Andamento crescita d'elaborazione.

Questo tipo di crescita quindi segue un andamento esponenziale e questo andamento è stato seguito dalle memorie RAM, che aumentano la capacità d'elaborazione di 4 volte ogni 3 anni.

CPU	Anno	Transistor	Bit	Tecnologia
4004	1971	2.300	4	10 μm
8008	1972	3.500	8	
8080	1974	4.500	8	
8086	1978	29.000	16	3 μm
80386	1985	275.000	32	
80486	1989	1.200.000	32	
Pentium	1993	3.100.000	32	
Pentium 4	2000	42.000.000	32	180 nm
Pentium 4	2005	125.000.000	64	90 nm
Core i7	2010	1.170.000.000	64	32 nm
Core i7	2013	1.400.000.000	64	
Xeon 28 Core	2017	2.000.000.000	64	14 nm

Tabella 1.2: Numero di transistor per le principali CPU.

La CPU 4004 è stato il primo processore ad ospitare tutti i suoi componenti in una sola piastrina, essa venne realizzata da un fisico italiano. Era un processore a 4 bit perché progettato per la realizzazione di una calcolatrice.

Il processore 8086 è stato il primo processore ad introdurre l'architettura x86, mentre il processore Pentium 4 del 2005 introduce l'architettura x86-64.

Per le CPU non vale la legge di Moore perché hanno un andamento irregolare e la crescita è del 2,6 ogni 3 anni.

1.4 Processo di produzione dei circuiti a semiconduttori

Si parte da una barra in Silicio di lunghezza intorno ai 30/60 cm e diametro pari a 15/30 cm.

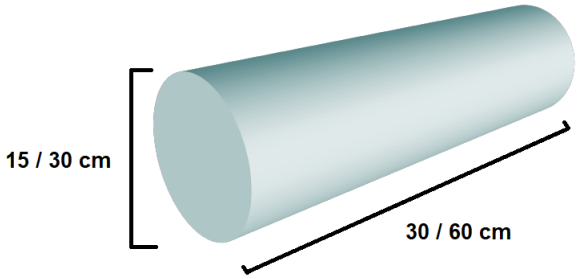


Figura 1.4: Dimensioni approssimative di una barra in Silicio.

Questa barra viene tagliata in **Wafer** di spessore pari a 2,5 cm, al quale viene applicata una *mascheratura*, ossia un deposito di materiali droganti applicato in più passate.

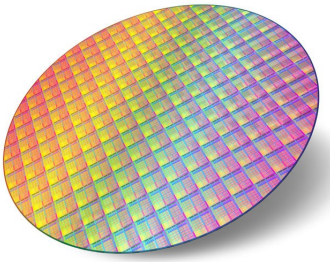


Figura 1.5: Wafer in silicio.

Si cercano difetti nella struttura cristallina che nel caso vengano trovati verranno scartati. Successivamente si cercano falle e sbavature come ulteriori difetti.

Viene tagliato il wafer in quadrati e vengono inseriti su di essi i vari componenti, in modo da creare un chip tramite saldatura, che a sua volta andrà a creare (così come grazie al taglio) alcuni difetti. Si può affermare che il rendimento del processo è inversamente proporzionale alla dimensione del chip.

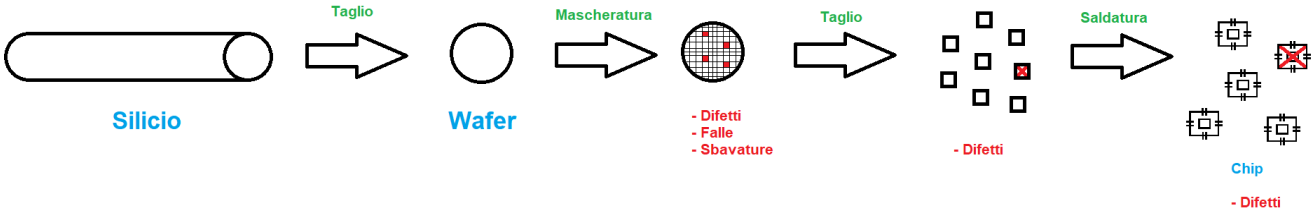


Figura 1.6: Processo produttivo dei chip.

Per incrementare il rendimento si esegue una miniaturizzazione con una tecnologia costruttiva a X metri, oggi è dell'ordine dei nanometri. La tecnologia è a 10 μm dal processore 4004 al processore 8080, successivamente si è passati a 3 μm dal processore 8086 al Pentium, 180 nm per il Pentium 4 a 32 bit, 90 nm per il Pentium 4 a 64 bit, 32 nm nel Core i7 del 2010 e 22 nm per il modello del 2013.

La legge di Moore è applicabile anche per le tecnologie precedenti la IV generazione.

	Capacità	Velocità
CPU	2,6 x 3 Anni	4 x 3 Anni
DRAM	4 x 3 Anni	2 x 10 Anni

Tabella 1.3: Prestazioni CPU e RAM.

Forbice delle prestazioni: Tra CPU e RAM esiste una forbice prestazionale, perché la CPU cresce in velocità di risposta molto più rapidamente rispetto alle memorie RAM. Per questo motivo, le memorie *Cache* nascono per fronteggiare il problema del collo di bottiglia di velocità d'accesso alla RAM da parte della CPU.

1.5 Prestazioni del calcolatore

Le prestazioni di un calcolatore vengono calcolate in base a due parametri principali:

- *Tempo d'esecuzione o Risposta:* è il tempo che intercorre dal momento in cui viene lanciato il programma al momento in cui esso termina, cioè coincide con il tempo d'esecuzione del singolo task e interessa il singolo utente;
- *Throughput o Bandwidth:* è il numero di task completati nell'unità di tempo, esso interessa il gestore del sistema che vuole servire più utenti possibile.

In riferimento al tempo d'esecuzione si ha una suddivisione in due sotto casi:

- **Tempo assoluto:** tiene conto anche del tempo degli altri task del sistema operativo, cioè del tempo impiegato dai task sullo stesso processore, incluso il sistema operativo;
- **Tempo d'esecuzione della CPU:** indica il tempo effettivamente dedicato dalla CPU per eseguire uno specifico task.

Per il calcolo del tempo della CPU vale la seguente formula, utile per analizzare le prestazioni di un calcolatore:

$$T_{CPU} = N_{ISTR} \cdot C_{PI} \cdot T_{CK} \tag{1.1}$$

- **Numero istruzioni:** il primo parametro dipende dal repertorio, fa parte del livello ISA. Maggiore e complesso è il repertorio, minore sarà il numero di operazioni necessarie;
- **Cicli per istruzione:** si calcola come la media pesata tra la frequenza d'utilizzo dell'i-esima operazione e il numero di cicli di clock dell'i-esima istruzione, $\sum x_i \cdot c_{pi}$;
- **Periodo di clock:** indica il maggiore ritardo tra le parti combinatorie e dipende dalla tecnologia costruttiva.

La tecnologia costruttiva impone una frequenza di funzionamento specifica. In presenza di una nuova tecnologia costruttiva, il costruttore della macchina può riproporre lo stesso progetto del calcolatore, senza variare nulla, avendo però caratteristiche migliori.

Anno	Processore	Frequenza	Potenza
1985	80839	16 MHZ	$\sim$ 4,1 W
1989	80486	25 MHZ	$\sim$ 4,9 W
1993	Pentium	66 MHZ	$\sim$ 10,1 W
1997	Pentium Pro	200 MHZ	$\sim$ 27,1 W
2001	Pentium 4	2 GHZ	$\sim$ 75,3 W
2004	Pentium 4	3.6 GHZ	$\sim$ 103 W

Tabella 1.4: Frequenze e consumi dei principali processori.

Le frequenze si sono assestate intorno al 2005 e questo fenomeno viene definito come **Barriera dell'energia**, mostrato nella figura 1.7 (*).

Esistono 3 periodi temporali in cui variano le prestazioni annue, è inferiore inizialmente rispetto al secondo periodo. Grazie alla legge di Moore l'aumentare dell'area dei chip è stata utilizzata per aumentare le prestazioni. Il parallelismo a livello d'istruzione viene introdotto nel secondo periodo dai costruttori, in modo tale che la

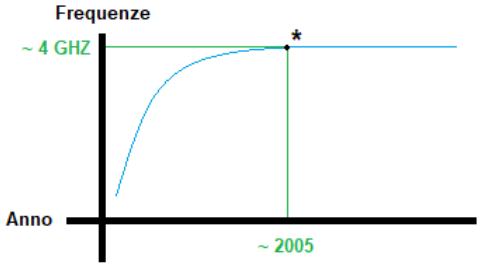


Figura 1.7: Barriera dell'energia.

Periodo	Incremento annuo prestazioni	Innovazioni organizzative
1978 - 1986	25 %	Circuiti aritmetici ad alte prestazioni
1986 - 2002	52 %	Parallelismo a livello di istruzioni (detto Instruction Level Parallelism o ILP)
2002 - Oggi	20 %	- Limiti ILP; -Barriera dell'energia; -Forbice tra CPU e RAM;

CPU riesca ad eseguire in parallelo le istruzioni in modo totalmente trasparente al programmatore. Questo aumenta la frequenza e crea un forte incremento delle prestazioni. Nel terzo periodo sono stati introdotti i circuiti di supporto per l'ILP, ma si raggiungono limiti intrinseci alla tecnica ILP. Questi risultati sono ottenuti tramite Benchmark, tra questi uno tra i più famosi è sicuramente *System Performance Evaluation Cooperative* (SPEC).

La legge di Moore però, nonostante tutto, continua a valere. A causa dei problemi del terzo periodo si è passati ai processori multi core per tentare di risolvere queste problematiche.

Si passa dal tempo d'esecuzione al throughput, con miglioramenti molto ridotti rispetto a prima.

Il pc moderno è un calcolatore parallelo con varie forme di parallelismo:

- **Implicito:** legato a ILP, ogni singola CPU riesce a parallelizzare l'esecuzione del programma in modo trasparente;
- **Esplicito:** richiede l'intervento del programmatore e si suddivide in:
  - **MIMD**: supportato dal multitasking e dal multi core e consiste nella divisione del codice in thread ad esempio;
  - **SIMD**: la macchina riesce ad eseguire la stessa operazione su più dati contemporaneamente, detto anche *Data Level Parallelism*. A livello hardware viene supportato dai nuovi registri vettoriali detti *Advanced Vector eXtension* (AVX) che sono registri a 256 bit che contengono diverse variabili contemporaneamente.

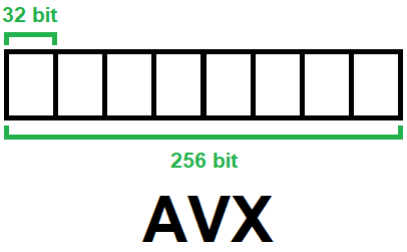


Figura 1.8: Struttura del registro AVX.

Le istruzioni SIMD nascono perché non si conosce un modo per sfruttare l'area in più dei chip, viene quindi migliorato un parametro prestazionale detto FLOPS (Floating Point Operations Per Seconds), ossia il numero di operazioni in virgola mobile al secondo.

Capitolo 2

Reti Logiche

2.1 Rete combinatoria

Una rete logica è un dispositivo in grado di elaborare dati, sono presenti vari morsetti associati a ciascuna variabile. Queste hanno un funzionamento unidirezionale, è un dispositivo su cui viaggiano segnali reali, ma si assume che siano ideali, ossia sono ammessi solo valori 0 e 1 nei vari istanti con variazioni istantanee tra essi, cosa che nella realtà non avviene.

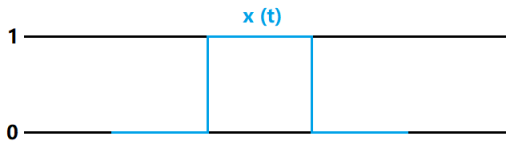


Figura 2.1: Andamento ideale del segnale x(t).

Ogni rete ha associato un ritardo, cioè un tempo d'attesa richiesto dalla rete prima che essa possa produrre un risultato, questo ritardo si indica con τ.

Le **Reti combinatorie** sono delle reti logiche in cui l'output dipende esclusivamente dall'input e vale la relazione:

$$z_i(t+\tau) = f_i(x_1(t), \ldots, x_n(t)) \tag{2.1}$$

questo a meno del ritardo.

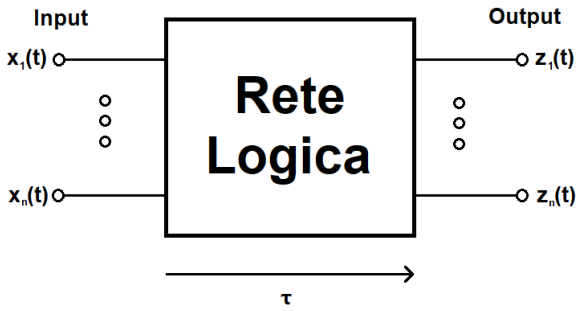


Figura 2.2: Esempio di rete combinatoria.

Per riconoscere una rete combinatoria bisogna valutare se sono presenti dei cicli, infatti *Tutte e sole le reti prive di cicli sono combinatorie.*

Viene definito **Flip-Flop** un unità di memoria che può essere di vari tipi. Esso (mostrato nella figura 2.3) è un unità sincrona, cioè segue un impulso periodico, il clock, che regola la vita del dispositivo, ossia scandisce quando è possibile variare le transizioni del dispositivo, che varieranno solo quando il clock sarà pari a 1.

Un Flip-Flop è costruito in modo che se all'arrivo dell'impulso questo è abilitato, allora esso cambierà il proprio stato ma continuerà a presentare in uscita sempre lo stesso risultato, che muterà solo al termine dell'impulso del clock.

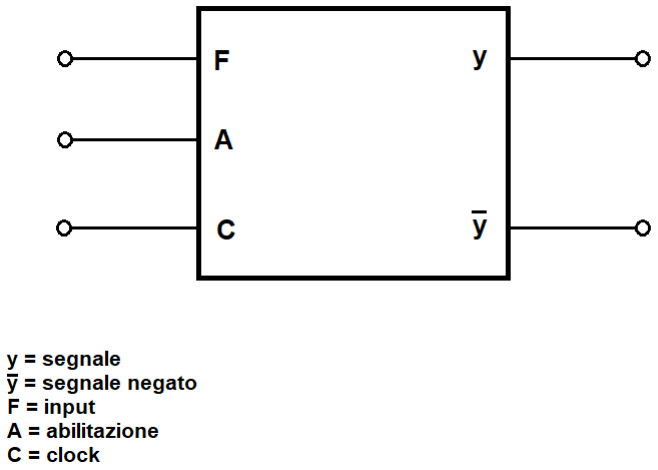


Figura 2.3: Struttura del Flip-Flop.

2.2 Reti sequenziali

Le **Reti sequenziali** sono delle reti che presentano almeno un ciclo. Bisogna individuare nel circuito una rete combinatoria che verrà rappresentata con i vari cicli, posizionati all'esterno della rete, questi cicli formano effettivamente la rete sequenziale che si vuole rappresentare.

Le y vengono dette variabili di stato *Futuro* per le y_i' e variabili di stato *Corrente* per le y_i.

Sono valide le sequenti relazioni:

$$z_i(t+\tau_i) = f_i(x_1(t), \ldots, x_n(t) + y_1(t), \ldots, y_k(t)) \tag{2.2}$$

$$y_i(t+\tau_i') = g_i(x_1(t), \ldots, x_n(t) + y_1(t), \ldots, y_k(t)) \tag{2.3}$$

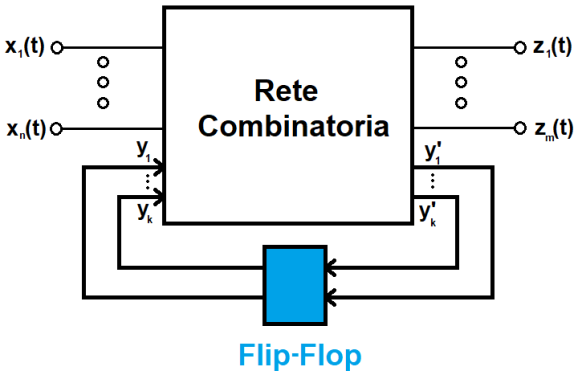


Figura 2.4: Esempio di rete sequenziale sincrona.

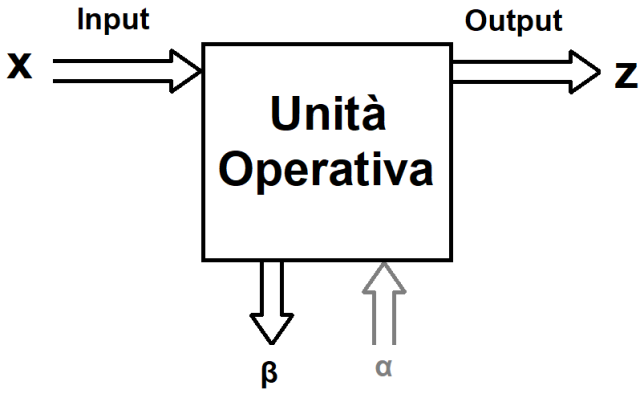
Nelle reti combinatorie τ è il massimo dei ritardi associati all'uscita, e quindi vale $\tau = max\{\tau_1, \ldots, \tau_n\}$. Le reti sequenziali sono asincrone, perché non si ha un clock e, per poterle rendere sincrone (come nella figura 2.4), bisogna inserire un Flip-Flop nella rete d'anello delle y. Questo Flip-Flop ha il compito di stabilizzare i valori e quindi le y saranno sempre stabili. Per funzionare correttamente bisogna fornire alla rete il tempo necessario per poter creare il risultato impostando un clock maggiore di τ, in modo da avere un output corretto.

Capitolo 3

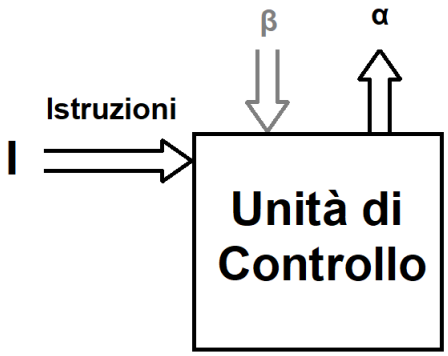
Calcolatore

3.1 Sistema di elaborazione dell'informazione

Il sistema di elaborazione dell'informazione nei calcolatori è diviso in due sottosistemi principali, ossia l'unità operativa e l'unità di controllo.



Nell'**Unità Operativa** sono presenti solitamente i registri, i circuiti aritmetici ecc. Essa esegue le operazioni sotto la guida di istruzioni codificate tramite segnali binari ricevuti dall'unità di controllo.



L'**Unità di Controllo** invia segnali α all'unità operativa e riceve i segnali β (o *condizione*) da quest'ultima.

L'unità di controllo e l'unità operativa sono due reti sincrone regolate dal segnale di controllo (o *impulso di controllo*) c noto come *Clock*.

L'unità di controllo contiene al suo interno la logica di controllo, infatti bisogna descrivere in esso preventivamente tutte le funzioni eseguibili dal sistema. Alcuni dei formalismi adottati prendono il nome di **Register Transfer Language** (RTL). Le operazioni elementari sono le micro operazioni, le quali sono istruzioni elementari eseguibili in un singolo colpo di clock. Ogni *μOperazione* ha la seguente forma:

<Dato> → <Registro>, in cui si sposta un dato nel registro;
<Funzione(<Dato>)> → <Registro>, ad esempio la funzione INC(A) → B;
<Dato1><Operazione><Dato2> → <Registro>, ad esempio A+B → C.

Viene chiamato $\mu Passo$ (micro passo) un insieme di istruzioni eseguibili nello stesso colpo di clock, questo è possibile nel momento in cui non ci siano conflitti, ossia non si utilizzano le stesse risorse.

Un micro passo si traduce in una $\mu Istruzione$ (micro istruzione), cioè la traduzione del passo in segnali di controllo α.

Viene utilizzata un'istruzione condizionale *if* del tipo:

if <Condizione> then <$\mu Sequenza$> else <Condizione> fi

dove la condizione è sempre un segnale β, il progettista deve garantire che lo stato della condizione deve essere costante per l'intera durata delle micro sequenze. Viene utilizzato il comando *goto* per simulare i salti non condizionati. Il progetto dell'unità operativa dipende dalle operazioni che si vogliono supportare, mentre il progetto dell'unità di controllo è solitamente standard. Una volta scelte le operazioni, si traducono in micro istruzioni che avranno associate un automa a stati finiti.

Ogni arco dell'automa avrà la forma Istruzione β / α, ognuno dei quali è rappresentato da un codice che li identifica, questo per ogni istruzione che si ha in uno stato. L'automa di un istruzione senza goto è un automa sequenziale, ossia un albero degenere, altrimenti si avranno archi di ritorno o diramazioni.

Per poter tradurre il circuito si hanno due possibili opzioni:

- **Controllo cablato**: vuol dire tradurre la tabella di verità in una rete minima a due livelli, perché le reti a due livelli sono le più veloci e tramite queste è possibile rappresentare ogni combinazione, ma una volta tradotto questo schema è immutabile perché avviene a livello macchina.
- **Controllo Microprogrammato**: consiste nella traduzione della logica di controllo in una memoria non volatile ma modificabile, ossia la ROM.

Il controllo cablato è più veloce ma immutabile, esso utilizza un decoder per individuare il valore cercato.

3.2 Macchina calcolatore

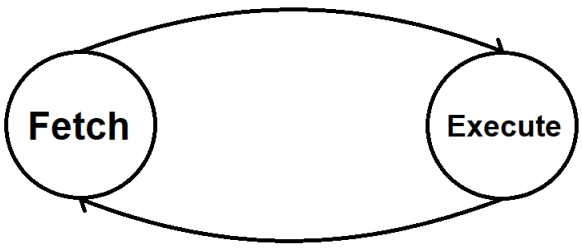
Si assume che la lettura nella RAM avvenga in un unico ciclo di clock, nella realtà ciò non è vero, ma si può supporre di dimensionare il clock in base al ritardo della CPU. Esistono due registri di memoria all'interno della CPU e sono il **Memory Address Register** (MAR) e il **Memory Buffer Register** (MBR).

Le operazioni di *scrittura* seguono la forma:

<Dato> → MBR, <Indirizzo> → MBR;
MBR → M[MAR];

Mentre le operazioni di *lettura* hanno la forma:

<Indirizzo> → MAR;
M[MAR] → MBR;



Le varie operazioni eseguite dal calcolatore seguono lo schema Fetch-Execute. La fase di Fetch è un operazione speciale, esegue la fase di recupero della prossima operazione e ha come codice operativo una sequenza di soli 0. Execute invece è la fase successiva alla Fetch ed esegue effettivamente l'operazione successiva. Vi è una fase di *Decode* che va dalla scrittura nell'**Instruction Register** (IR) alla stabilizzazione dei segnali α. Alla fase di Fetch è associato il seguente Automa:

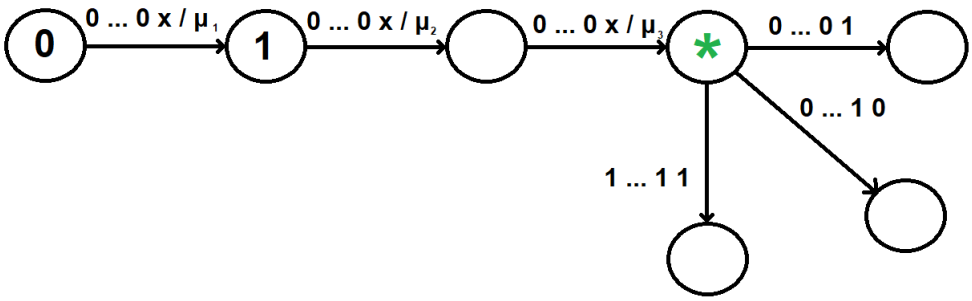


Figura 3.1: Automa Fetch.

Dopo la sovrascrizione dell'IR (* nella Figura 3.1) le varie operazioni che si possono eseguire termineranno tutte con il ritorno all'inizio dell'automa (stato 0) per poter consentire alla fase di Fetch di ripartire.

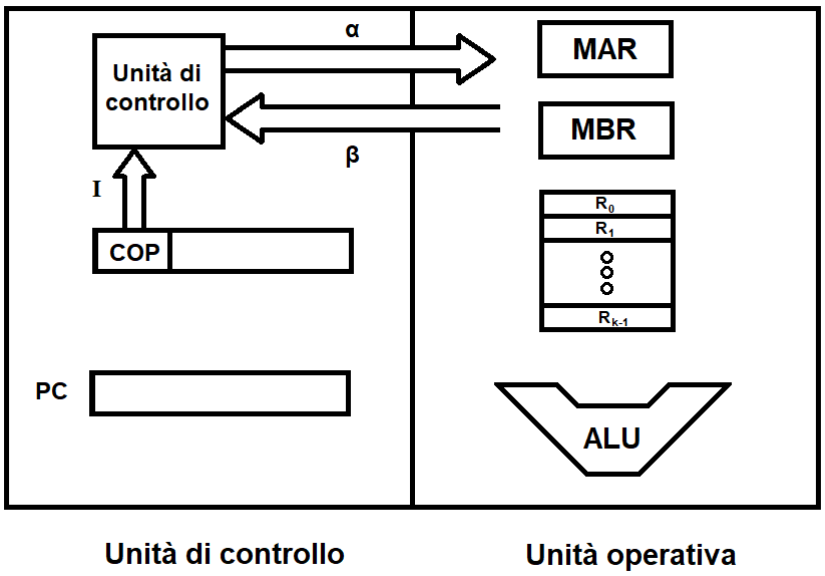


Figura 3.2: Schema della macchina calcolatore.

3.3 Modelli di repertorio delle istruzioni

Sono i principi su cui si basa la progettazione di repertorio, negli anni sono state utilizzate quattro diverse metodologie:

- Modello Memoria-Memoria;
- Modello Memoria-Registro;
- Modello Registro-Registro;
- Modello a stack;

Si può osservare adesso come l'istruzione $A = B + C$ sia formulata dai vari modelli. Nell'istruzione si eseguono i seguenti passi:

1. Leggere il valore posto in memoria all'indirizzo B;
2. Leggere il valore posto in memoria all'indirizzo C;

3. Sommare tramite ALU i valori ottenuti nei passi precedenti;
4. Scrivere il risultato in memoria nell'indirizzo A.

3.3.1 Modello Memoria-Memoria

Secondo questa tipologia di modello, le elaborazioni sono esprimibili tramite tre operandi, tutti e tre posizionati in memoria, di cui due sono sorgenti (B e C) e l'altro è la destinazione (A). La formula è la seguente ADD A, B, C. e si ottiene una rappresentazione molto compatta.

3.3.2 Modello Memoria-Registro

In questo modello le istruzioni hanno un massimo di due operandi, la sorgente può trovarsi in memoria. In questo caso, come mostrato dalla formula successiva, la rappresentazione è meno compatta rispetto al modello precedente.

```
LOAD    R0, B
ADD     R0, C
STORE   R0, A.
```

3.3.3 Modello Registro-Registro

Le uniche istruzioni che possono avere accesso alla memoria sono STORE e LOAD, tutte le altre devono necessariamente utilizzare i registri. Si hanno tre operandi ma devono essere registri, di cui due sono sergenti e l'altro fa da destinazione.

```
LD      R0, B
LD      R1, C
ADD     R2, R0, R1
ST      R2, A.
```

3.3.4 Modello a Stack

Questo modello è basato sugli elementi in cima allo stack.

```
PUSH    B
PUSH    C
ADD
POP     A.
```

Nel **Modello Memoria-Memoria**, si ha una maggiore compattezza del codice, ma la codifica è più onerosa, dal momento che bisogna indicare tre registri. Queste caratteristiche erano adatte quando le RAM erano piccole ma costose. Con l'aumentare della memoria e la forbice CPU-RAM, è diventato meno desiderabile questo modello che utilizza intensivamente la RAM.

Il **Modello a Stack** la situazione peggiora, perché aumenta il numero di accessi in RAM e tutte le operazioni avvengono li.

Il **Modello Memoria-Registri** è intermedio tra tutti e quattro i modelli, perché le istruzioni possono coinvolgere solo registri, ma non è perfetto. Infatti, potenzialmente, ogni operazione potrebbe accedere in memoria e queste sono operazioni critiche per una possibile parallelizzazione del codice. Anche se si cercano di utilizzare solo i registri non è fattibile, perché sono pochi e non si riesce a rappresentare ogni istruzione realizzabile, si basa sulla tecnica *Complex Instruction Set Computer* (CISC) che si oppone alla tecnologia *Reduced Instruction Set Computer* (RISC). I calcolatori CISC supportano centinaia di operazioni, mentre i calcolatori RISC poche istruzioni con ridotte varianti di formato.

Nel **Modello Registro-Registro** si ha una tecnologia RISC e si cerca di evitare la dipendenza dalla memoria con un maggiore quantitativo di registri a disposizione.

Capitolo 4

Macchina multiciclo

4.1 Introduzione

La **macchina Multiciclo** è una configurazione della famiglia CISC, le istruzioni vengono eseguite in più cicli di clock, ossia la singola istruzione viene divisa in più cicli. La macchina contiene sempre almeno un *bus* a cui sono collegate le unità funzionali.

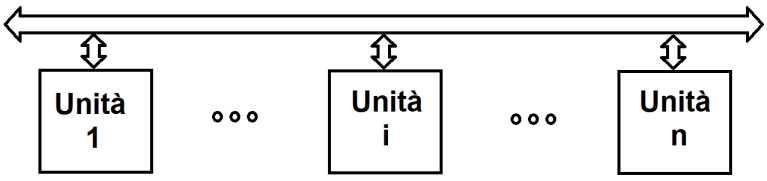


Figura 4.1: Esempio di unità funzionali collegate da un bus.

Ad ogni istante, uno o più unità funzionali possono essere in esecuzione, purché non siano in conflitto tra loro. Il periodo di clock sarà impostato come il massimo dei ritardi associati alle singole unità funzionali, ossia $T_{CLK} = max\{\tau_i\}$.

4.2 Architettura

Nella macchina multinastro è presente una memoria RAM a 64k indirizzi, detti *Locazioni*, a 16 bit, con 32 registri ad uso generale **General Purpose Register** (GPR) a 16 bit. Un registro di flag Z ad 1 bit, che varrà 1 se il risultato dell'ALU è pari a 0.

I dati supportati sono gli interi a 16 bit e il repertorio si ispira alla memoria a registri con operazioni a due operandi, ossia formule del tipo: <*Operazione*> <*Destinazione*>, <*Sorgente*>.

Sono disponibili quattro modalità di indirizzamento, di seguito specificate con riferimento all'istruzione LD (LOAD) di caricamento dalla memoria:

1. **Indirizzamento a registro:** LD R_i, R_j;
2. **Indirizzamento immediato:** LD R_i, # X, dove # X è un intero costante a 16 bit e non una variabile;
3. **Indirizzamento diretto:** LD R_i, X, in questo caso X è una variabile;
4. **Indirizzamento indicizzato:** LD R_i, X(R_j), qui X(R_j) è il valore somma tra X e R_j.

L'insieme delle istruzioni che si voglio implementare sono:

1. **LD** (LOAD) carica i dati;
2. **ST** (STORE) salva i dati;
3. **ADD**, esegue la somma;
4. **SUB**, esegue la sottrazione;
5. **CMP**, esegue il confronto;

6. **JP X**, esegue un salto incondizionato all'indirizzo X;

7. **JE X**, esegue un salto all'indirizzo X se uguale a 0 il bit nel registro di flag Z;

8. **JNE X**, esegue un salto all'indirizzo X se diverso a 0 il bit nel registro di flag Z;

Nella macchina sono presenti tre bus la disponibilità di più bus consente di aumentare il "parallelismo" del percorso dati (data path). Infatti, due bus possono essere utilizzati per portare in ingresso all'ALU due operandi ed il terzo bus per inviare il risultato dell'ALU ad un registro, il tutto in un solo ciclo di clock;

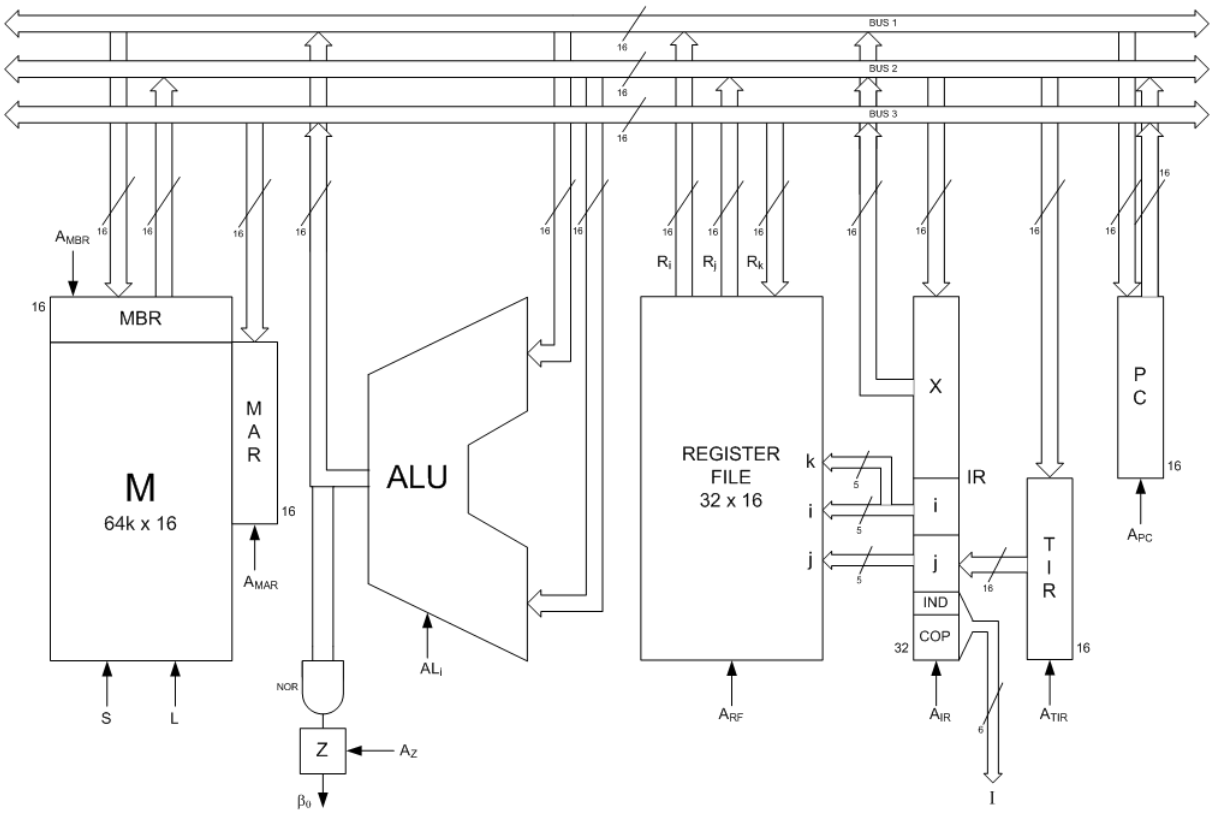


Figura 4.2: Architettura macchina multiciclo.

Register File (RF) è il banco dei registri, ossia un blocco funzionale contenente i registri ad uso generale. Permette di leggere il contenuto di due registri e di modificare il contenuto di un altro registro nello stesso ciclo di clock. Nello schema di riferimento, il RF invia i registri da leggere sui bus 1 e 2 e preleva il registro da scrivere dal bus 3;

Arithmetic Logic Unit (ALU) utilizzata per effettuare le elaborazioni. Nello schema di riferimento, preleva i due ingresso dai bus 1 e 2 ed invia il risultato al bus 3.

Instruction Register (IR) è un registro a 32 bit e suddiviso nei campi illustrati in precedenza. In particolare i campi i e j, il cui scopo è quello di specificare i registri R_i e R_j coinvolti nell'istruzione, sono utilizzati per comandare direttamente il RF sfruttando il formato istruzione ortogonale. Poiché la macchina adotta il modello di esecuzione registro-memoria, il campo i dell'IR identifica sia un registro sorgente che un registro destinazione e quindi il suo contenuto viene inviato agli ingressi i e k del RF (che individuano, rispettivamente, uno dei due registri da leggere del RF e l'unico registro da scrivere del RF), mentre il campo j dell'IR viene inviato in ingresso al campo j del RF (che individua l'altro dei due registri da leggere del RF).

Il registro TIR a 16 bit viene utilizzato per memorizzare temporaneamente la parte meno significativa del registro IR durante la fase di fetch. Si ricorda che la sovrascrittura del campo COP determina la fine della fase di fetch ed il passaggio all'esecuzione vera e propria dell'istruzione (fase di execute). Si tratta quindi dell'ultimo passaggio da eseguire durante la fase di fetch. Si noti che si sarebbe potuto scegliere di rappresentare l'istruzione in memoria ponendo nella locazione di indirizzo più basso il campo X, eliminando così la necessità del registro TIR. Sebbene questa soluzione sia praticabile per il formato istruzione corrente, non è adatta per formati istruzione a lunghezza variabile, in cui è necessario ispezionare innanzitutto il campo COP al fine di determinare la lunghezza effettiva dell'istruzione corrente.

4.3 Formato istruzioni

Si utilizza un unico formato per tutte le istruzioni in modo da semplificare le cose. Si identifica l'istruzione più lunga e complessa che sarà qualcosa del tipo *ADD* R_i, R_j *(X)*, ossia R_i=R_i+M[R_j+X].

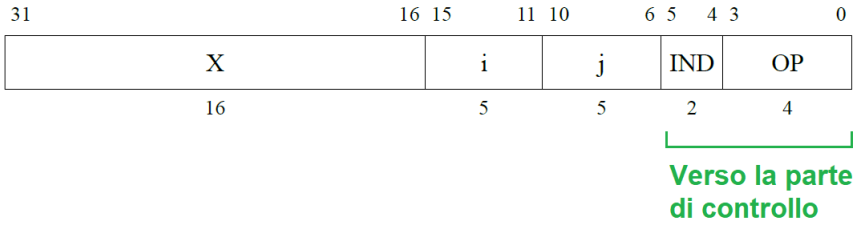


Figura 4.3: Formato istruzioni.

X è un campo a 16 bit, si utilizzano 32 registri e per indicizzarli sono necessari 5 bit. Ind è il campo di indirizzamento e COP è il codice operativo.

Indirizzamento	**Ind**
A registro	00
Immediato	01
Diretto	10
Indicizzato	11

Tabella 4.1: Codici indicizzazione.

Questo formato è detto **Ortogonale**, cioè i campi sono separati tra loro per definire il formato. Si hanno campi separati per le diverse tipologie di informazioni presenti.

4.4 Codice RTL delle istruzioni

Di seguito verrà mostrato il codice RTL delle istruzioni da implementare.

4.4.1 Fetch

Anziché progettare il PC come registro funzione a incremento, si sfrutta la disponibilità di 3 bus per incrementare il PC mediante la ALU. Il registro X coincide con i bit 16-31 del campo IR (X = IR16-31). Il registro TIR viene utilizzato per caricare il campo COP nell'ultimo micropasso della microsequenza associata alla FETCH.

PC $\rightarrow$ MAR, PC + 1 $\rightarrow$ PC;
M[MAR] $\rightarrow$ MBR, PC $\rightarrow$ MAR, PC + 1 $\rightarrow$ PC;
MBR $\rightarrow$ TIR, M[MAR] $\rightarrow$ MBR;
TIR $\rightarrow$ IR_{0-15}, MBR $\rightarrow$ X;

4.4.2 LD R_i, R_j

Il registro R_j viene inviato mediante il bus 2 alla ALU, che lo restituisce invariato in uscita sul bus 3 e viene quindi scritto nel RF. Occorre abilitare il RF in scrittura (A_RF asserito) per permettere la scrittura del registro R_i (ovvero R_k).

$R_j \rightarrow R_i$;

4.4.3 LD R_i, #X

Il contenuto del campo X dell'IR, che in questo caso rappresenta un numero intero (operando immediato), viene inviato al RF mediante il bus 3.

X $\rightarrow$ R_i;

4.4.4 LD R_i, X

Il contenuto della locazione di memoria di indirizzo X viene letto dalla RAM e scritto nel RF.

X $\rightarrow$ MAR;
M[MAR] $\rightarrow$ MBR;
MBR $\rightarrow$ R_i;

4.4.5 LD R_i, X(R_j)

Il contenuto della locazione di memoria di indirizzo X+R_j viene letto dalla RAM e scritto nel RF. L'indirizzo effettivo di memoria viene calcolato utilizzando la ALU, il campo X di IR viene letto dal bus 1, il registro R_j dal bus 2 e la loro somma inviata al MAR dalla ALU mediante il bus 3.

4.4.6 ST R_i, X

Il contenuto del registro R_i viene scritto in memoria nella locazione di indirizzo X. Il MAR (campo X mediante bus 3) e l'MBR (registro R_i mediante bus 1) vengono caricati nello stesso ciclo di clock.

X $\rightarrow$ MAR, R_i $\rightarrow$ MBR;
MBR $\rightarrow$ M[MAR];

4.4.7 ST R_i, X(R_j)

Il contenuto del registro Ri viene scritto in memoria nella locazione di indirizzo X+R_j. Il MAR (campo X mediante bus 3) e l'MBR (registro R_i mediante bus 1) vengono caricati nello stesso ciclo di clock. Questa volta i 3 bus vengono impegnati dal calcolo dell'indirizzo effettivo (primo micropasso) e quindi il caricamento dell'MBR richiede un ulteriore ciclo di clock. Complessivamente la store indicizzata risulta più lenta di quella diretta.

X + R_j $\rightarrow$ MAR;
R_i $\rightarrow$ MBR;
MBR $\rightarrow$ M[MAR];

4.4.8 ADD R_i, R_j

Calcola R_i + R_j e memorizza il risultato in R_i (ovvero R_k). R_i arriva all'ALU mediante il bus 1, R_j mediante il bus 2 e la somma viene presentata al RF mediante il bus 3. Il flag Z viene modificato di conseguenza (comando A_Z asserito).

R_i+R_j $\rightarrow$ R_i, NOR(R_i+R_j) $\rightarrow$ Z;

4.4.9 ADD R_i, #X

Somma a R_i (via bus 1) il campo X (via bus 2; operando immediato) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

R_i + X $\rightarrow$ R_i, NOR(R_i + X) $\rightarrow$ Z;

4.4.10 ADD R_i, X

Somma il contenuto del registro R_i (via bus 1) e della locazione di memoria di indirizzo X (provienente dall'MBR via bus 2) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

X$\rightarrow$ MAR;
M[MAR] $\rightarrow$ MBR;
R_i + MBR $\rightarrow$ R_i, NOR(R_i + MBR) $\rightarrow$ Z;

4.4.11 ADD R_i, X(R_j)

L'indirizzo effettivo (X+R_j) della locazione di memoria da addizionare viene calcolato mediante l'ALU. Somma il contenuto del registro R_i (via bus 1) e della locazione di memoria di indirizzo X+R_j (proviene dall'MBR via bus 2) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

X + R_j → MAR;
M[MAR] → MBR;
R_i + MBR → R_i, NOR(R_i + MBR) → Z;

4.4.12 SUB R_i, R_j

Calcola R_i - R_j e memorizza il risultato in R_i (ovvero R_k). R_i arriva all'ALU mediante il bus 1, R_j mediante il bus 2 e la somma viene presentata al RF mediante il bus 3. Il flag Z viene modificato di conseguenza (comando A_Z asserito).

$R_i$$R_j$ → R_i, NOR(R_i-R_j) → Z;

4.4.13 SUB R_i, #X

Sottrae da R_i (via bus 1) il campo X (via bus 2; operando immediato) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

R_i - X → R_i, NOR(R_i - X) → Z;

4.4.14 SUB R_i, X

Sottrae dal contenuto del registro R_i (via bus 1) il contenuto della locazione di memoria di indirizzo X (proviene dall'MBR via bus 2) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

X→ MAR;
M[MAR] → MBR;
R_i - MBR → R_i, NOR(R_i - MBR) → Z;

4.4.15 SUB R_i, X(R_j)

L'indirizzo effettivo (X+R_j) della locazione di memoria da sottrarre viene calcolato mediante l'ALU. Sottrae il contenuto del registro R_i (via bus 1) e della locazione di memoria di indirizzo X+R_j (proviene dall'MBR via bus 2) e scrive il risultato nel RF (via bus 3). Aggiorna il flag Z.

X + R_j → MAR;
M[MAR] → MBR;
R_i - MBR → R_i, NOR(R_i - MBR) → Z;

4.4.16 CMP R_i, R_j

Lo scopo dell'istruzione CMP è quello di settare i flag sulla base del confronto dei due operandi. Calcola R_i – R_j e non memorizza il risultato in R_i = R_k (comando A_Z non asserito). R_i arriva all'ALU mediante il bus1, R_j mediante il bus 2. Il flag Z viene modificato di conseguenza.

NOR(R_i - R_j) → Z;

4.4.17 CMP R_i, #X

Lo scopo dell'istruzione CMP è quello di settare i flag sulla base del confronto dei due operandi. Sottrae da R_i (via bus 1) il campo X (via bus 2; operando immediato). Aggiorna il flag Z, ma non il RF.

NOR(R_i - X) → Z;

4.4.18 CMP R_i, X

Lo scopo dell'istruzione CMP è quello di settare i flag sulla base del confronto dei due operandi. Sottrae dal contenuto del registro R_i (via bus 1) quello della locazione di memoria di indirizzo X (proveniente dall'MBR via bus 2). Aggiorna il flag Z, ma non il RF.

X $\rightarrow$ MAR;
M[MAR] $\rightarrow$ MBR;
NOR(R_i - MBR) $\rightarrow$ Z;

4.4.19 CMP R_i, X(R_j)

Lo scopo dell'istruzione CMP è quello di settare i flag sulla base del confronto dei due operandi. L'indirizzo effettivo (X + R_j) della locazione di memoria da sottrarre viene calcolato mediante l'ALU. Sottrae dal contenuto del registro R_i (via bus 1) quello della locazione di memoria di indirizzo X + R_j (proveniente dall'MBR via bus 2). Aggiorna il flag Z, ma non il RF.

X + R_j $\rightarrow$ MAR;
M[MAR] $\rightarrow$ MBR;
NOR(R_i - MBR) $\rightarrow$ Z;

4.4.20 JP X

Carica nel PC l'indirizzo della prossima istruzione da eseguire posto nel campo X.

X $\rightarrow$ PC;

4.4.21 JE X

Se Z vale 1 carica nel PC l'indirizzo della prossima istruzione da eseguire posto nel campo X.

if Z = 1
    then X $\rightarrow$ PC;
    else ϕ;
fi

4.4.22 JNE X

Se Z vale 0 carica nel PC l'indirizzo della prossima istruzione da eseguire posto nel campo X.

if Z = 0
    then X $\rightarrow$ PC;
    else ϕ;
fi

Le macchine CISC hanno formati istruzione complessi, mentre quelli della macchina proposta non lo sono. Le istruzioni complesse, solitamente, hanno lunghezza variabile nella realtà.

4.5 Modifiche al formato istruzione

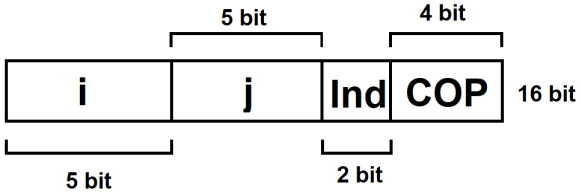
Volendo utilizzare le istruzioni a lunghezza variabile, si può inserire un altro indirizzamento aggiuntivo detto **Indirizzamento indiretto a registro**. L'obiettivo di questa aggiunta è quello di evitare spreco di memoria, ovvero evitare che l'istruzione occupi più spazio di quello strettamente necessario. Due bit non sono più sufficienti per distinguere gli indirizzamenti. Per mantenere l'ortogonalità del formato istruzione si rende necessario un terzo bit. Cambia così la tabella dei codici d'indirizzamento, che diventa:

Questi indirizzamenti possono essere raggruppati in 3 macro tipologie:

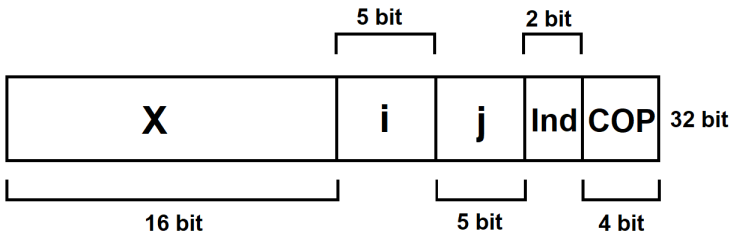
Indirizzamento	Ind
A registro	x00
Indiretto a registro	x01
Indicizzato	x10
Immediato	011
Diretto	111

Tabella 4.2: Codici indicizzazione per istruzioni a lunghezza variabile.

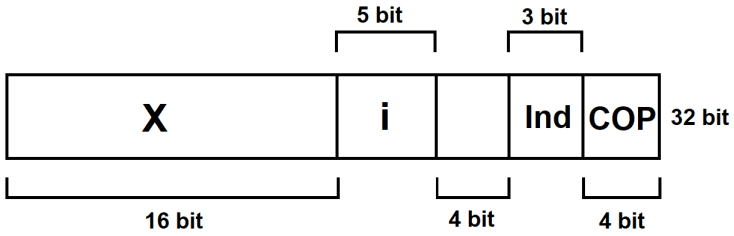
- **Tipologia A:** fanno parte di questa tipologia gli indirizzamenti a registro e gli indirizzamenti indiretti a registro e si utilizzano i campi i e j. Si utilizza la seguente forma:



- **Tipologia B:** solo l'indirizzamento indicizzato appartiene a questa tipologia. Vengono utilizzati i campi i, j e X. La forma è la seguente:



- **Tipologia C:** gli ultimi due indirizzamenti, ossia immediato e diretto, compongono questa tipologia. Vengono utilizzati i campi i e X. La forma è la seguente:



I 5 bit di j non vengono utilizzati nella tipologia C, quindi si unisce un bit di j al campo Ind per poter discriminare i due casi, immediato e diretto. Nei tipi A e B non ha senso eseguire questa operazione, infatti il terzo bit di Ind ha valore indeterminato x.

4.5.1 Nuova Fetch

La fase di fetch si modifica in modo da gestire le istruzioni di lunghezza variabile. Al fine di non sprecare cicli di clock durante il caricamento dell'istruzione, si aggiunge un segnale condizione sul bit 4 dell'MBR, segnale che utilizzeremo per testare il valore del campo LEN quando si legge dalla memoria la prima locazione relativa alla prossima istruzione da eseguire.

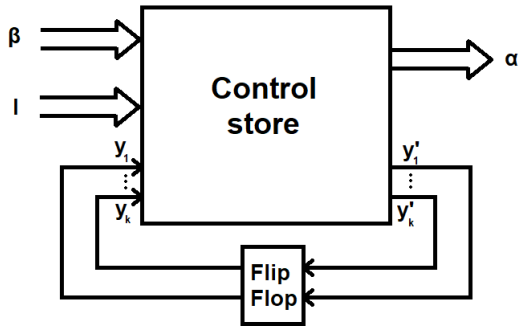
$PC \rightarrow MAR, PC + 1 \rightarrow PC;$
$M[MAR] \rightarrow MBR, PC \rightarrow MAR;$
if $MBR_4 = 0$
    then $MBR \rightarrow IR_{0-15}$;
    else $M[MAR] \rightarrow MBR, MBR \rightarrow TIR, PC + 1 \rightarrow PC;$
        $MBR \rightarrow X, TIR \rightarrow IR_{0-15}$;
fi

La nuova fetch impiega tre cicli di clock per caricare un'istruzione che occupa 1 locazione di memoria (MBR4 = LEN = 0) e quattro cicli di clock per caricare un'istruzione che occupa 2 locazioni di memoria (MBR4 = LEN = 1).

Per eseguire correttamente la fetch occorre poter caricare IR_{0-15} direttamente dall'MBR, aggiungendo un collegamento dal bus 2 verso IR_{0-15} ed un multiplexer (pilotato da un nuovo segnale di comando) in cui confluiscono il nuovo collegamento e l'uscita del TIR.

4.6 Ottimizzazione del controllo microprogrammato

Il controllo microporgrammato è utile ai fini di spostare la complessità del progetto dell'unità operativa a quella programmata. Purtroppo, però, all'aumentare della complessità del sistema, aumenta anche la complessità spaziale. Questo approccio necessita quindi un lavoro di ottimizzazione, soprattutto sul numero di locazioni.



Il numero di locazioni L è pari a 2^m, con $m = |\mathrm{I}|+|\beta|+|\mathrm{y}|$. In particolare, ogni segnale β raddoppia il numero di locazioni, quindi bisogna eliminare la dipendenza dai segnali.

I passi da compire sono la riduzione della parole in memoria, attuabile eliminando la dipendenza da I, β e y, oppure riducendo i segnali β che concorrono alla formazione dell'indirizzo. Bisogna inoltre ridurre il numero delle locazioni eliminando l'indicatore esplicito al prossimo stato o riducendo il numero di segnali di comando α.

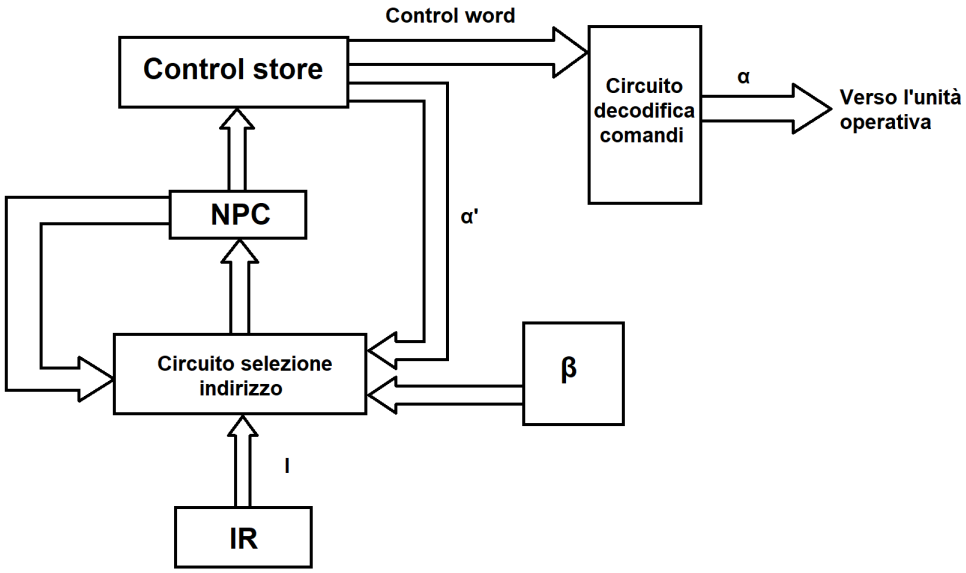


Figura 4.4: Possibile schema risolutivo.

Nello schema in figura 4.4, i segnali α' sono parte di α, ossia segnali di controllo dell'indirizzo.

Supponendo di avere un sistema con quattro istruzioni, sono necessari due segnali istruzione. Non è importante cosa svolgono queste istruzioni

4.7 Dimensione clock

Per dimensionare correttamente il clock, bisogna analizzare prima la formula del periodo della CPU:

$$T_{CPU} = N_{ISTR} \cdot C_{PI} \cdot T_{CK} \tag{4.1}$$

Con N_{ISTR} pari al numero di istruzioni, C_{PI} indica i cicli per istruzione e T_{CK} pari al periodo di clock. Fissato un linguaggio di riferimento si ha che $T_{CPU} \propto C_{PI} \cdot T_{CK}$, cioè il periodo della CPU è proporzionale ai cicli per istruzione moltiplicati per il periodo del clock, quindi il numero di istruzioni non è discriminante. Vale l'equazione:

$$C_{PI} = \sum_i x_i \cdot C_{PI} \Rightarrow T_{CPU} \propto (\sum_i x_i \cdot C_{PI}) \cdot T_{CK} \tag{4.2}$$

Il periodo del clock è dimensionato in modo da avere il massimo dei ritardi, ossia $T_{CK} = \max_i\{T_i\}$. Ad esempio, si supponga di avere le seguenti unità funzionali e istruzioni:

$$UF = \begin{cases} RAM \Rightarrow \tau_{RAM} : 30nsec \\ ALU \Rightarrow \tau_{ALU} : 12nsec \\ ALTRE \Rightarrow \tau_A : 5nsec \end{cases} \qquad Istruzioni = \begin{cases} Aritmetiche : X_{ARIT} = 45\%, UF : ALU \\ AccessoMemoria : X_{MEM} = 35\%, UF : ALU + RAM + A \\ Salti : X_{SAL} = 20\%, UF : A \end{cases} \tag{4.3}$$

Per dimensionare correttamente il clock, è possibile percorrere varie strade:

1. *Soluzione 1 : T_{CK} = ritardo della RAM*

   Nell'esempio precedente, il ritardo della RAM è pari a 30 nsec, e quindi si può dimensionare il periodo del clock in questo modo. Volendo ora calcolare il periodo della CPU, è noto che $C_{PI} = Istruzioni\ aritmetiche \cdot unita'\ utilizzate + Accesso\ in\ memoria \cdot unita'\ utilizzate + Salti \cdot unita'\ utilizzate = 0,45 \cdot 1 + 0,35 \cdot 3 + 0,2 \cdot 1 = 1,7$. Per questo motivo il periodo della CPU è pari a $T_{CPU} = 1,7 \cdot 30 = 51 \frac{nsec}{Istruzione}$. Solitamente questo approccio non viene utilizzato, perché la RAM è troppo lenta e si sceglie quindi un clock più basso.

2. *Soluzione 2 : Cicli di wait*

   Il clock viene quindi dimensionato in modo da essere più basso del massimo dei ritardi, questo viene definito come **Clock fine**. Un ciclo di wait è un ciclo in cui non avviene nulla, ma si attende che termini l'accesso alla RAM che è la componente più lenta. Ponendo $T_{CK} = 12nsec$ si può calcolare secondo la formula definita sopra $C_{PI} = 0,45 \cdot 1 + 0,35 \cdot (1 + 3 + 1) + 0,2 \cdot 1 = 2,4$. In questo caso è peggiorato, ma globalmente si ha che $T_{CPU} = 2,4 \cdot 12 = 28,8 \frac{nsec}{Istruzione}$ che è molto più basso del caso precedente, perché in media ogni istruzione impiega circa 30 nsec per essere eseguita.

3. *Soluzione 3 :*

   Ponendo $T_{CK} = 5nsec$ si ottiene $C_{PI} = 0,45 \cdot 3 + 0,35 \cdot (3 + 6 + 1) + 0,2 \cdot 1 = 5,05$ e quindi $T_{CK} = 5,05 \cdot 5 = 25,25 \frac{nsec}{Istruzione}$. Ciò dimostra che non sempre ridurre vuol dire ottenere un valore migliore.

4. *Soluzione 4:*

   Si pone $T_{CK} = 6nsec$ e si riduce la frequenza a $f = 166MHZ$. Da questo si può dedurre $C_{PI} = 0,45 \cdot 2 + 0,35 \cdot (2 + 5 + 1) + 0,2 \cdot 1 = 3,9$ e quindi $T_{CPU} = 6 \cdot 3,9 = 23,4 \frac{nsec}{Istruzione}$. Questo è il risultato migliore trovato, ma non sempre ridurre drasticamente la frequenza coincide con un aumento delle prestazioni, perché bisogna eliminare i ritardi e i cicli di wait per riuscire ad ottenere l'ottimo.

4.8 RISC e CISC

Inizialmente si utilizzava il controllo cablato, con un rapporto 1-a-1 tra istruzioni e hardware, cioè per ogni istruzione è presente un circuito che la realizza. Con l'aumentare delle istruzioni, si ha la necessità di hardware aggiuntivi e si andrebbero a creare dei costi troppo elevati. Per questo motivo, viene introdotta la **Microprogrammazione**, realizzata dalla famiglia IBM System 360, con famiglie di PC qualitativamente differenti ma capaci di eseguire tutte lo stesso codice, seppur con hardware diversi. Vengono quindi introdotti i **Repertori complessi** che garantiscono i seguenti vantaggi:

- Facilita la realizzazione di compilatori efficienti;
- Programmi più compatti in memoria centrale;

- Controllo microprogrammato competitivo in termini di prestazioni rispetto al controllo cablato:
  - Facilità di progettazione, perché scritto su memoria rescrivibile;
  - Aggiornamento della logica di controllo.

Le macchine a repertorio complesso vengono chiamate **VAX**, esse sono particolari famiglie a repertorio complesso. A metà anni '70 vennero eseguiti sul comportamento dei calcolatori, in cui emerse che l'80% delle istruzioni ad alto livello viene tradotta facendo uso del solo 20% delle istruzioni a basso livello. Quindi l'idea fu quella di concentrarsi su poche istruzioni massimizzando le prestazioni. Nel 1980, Patterson introdusse **Reduced Instruction Set Computer** (RISC) e nell'anno successivo Hannessy diede vita al progetto **Microprocessor with Interlocked Pipeline Stages** (MIPS), la sigla venne scelta per assonanza all'indice di prestazione *Million Instruction Per Second* (mips) in modo da enfatizzare questo obiettivo.

L'idea è quella di avere poche istruzioni semplici che potessero essere emesse velocemente. L'emissione avviene in fase di execute quindi si cerca di massimizzare questa fase. Idealmente queste macchine eseguono un istruzione per ciclo di clock, cercando di massimizzare la velocità le istruzioni che verranno eseguite in parallelo.

4.9 Principi di progettazione dei moderni calcolatori (o RISC)

Si possono evidenziare vari principi su cui si basa la progettazione dei moderni calcolatori:

- **Massimizzare la frequenza di emissione delle istruzioni**;
- **Istruzioni eseguibili in poco tempo**: devono utilizzare poco la CPU, perché deve eseguire più istruzioni parallelamente e se complesse intaccano le prestazioni, utilizzando istruzioni semplici;
- **Numero limitato di istruzioni**: per limitare la complessità della parte di controllo;
- **Istruzioni facili da decodificare**: la decodifica è la fase tra il caricamento e l'emissione dei segnali α. Le macchine hanno tutte la stessa lunghezza d'istruzione e tipicamente un formato costante;
- **Istruzioni eseguite direttamente in hardware**: si torna al controllo cablato per avere una decodifica più veloce possibile;
- **Modello Registro-Registro**: per evitare di introdurre troppi accessi in memoria;
- **Molti registri a disposizione**: per supportare il modello Registro-Registro;
- **Compilatori efficienti**;

Attualmente le due tipologie CISC e RISC coesistono in alcuni progetti come nell'architettura x86 (1978), ha un architettura CISC con molte istruzioni, ma sono RISC dal punto di vista dell'organizzazione. Questo ibrido nasce per supportare la retrocompatibilità, derivante dal CISC, massimizzando l'emissione di istruzioni, tramite RISC.

Nelle macchine CISC si ha un parametro migliore rispetto a N_{ISTR}, ma nelle RISC il parametro C_{PI} è idealmente 1, mentre nel CISC è > 1, infine τ_{CK} le RISC riescono ad ottenere periodo più basso per la presenza di meno istruzioni.

Capitolo 5

Macchina monociclo

5.1 Introduzione

La **Macchina monociclo** è l'organizzazione di base dei calcolatori RISC. La macchina, infatti, è progettata in modo da eseguire ogni operazione in un unico ciclo di clock, fetch compresa. Verrà analizzata una versione semplificata della macchina MIPS. Il modello utilizzato è Registro-Registro, il repertorio d'istruzioni è ridotto al minimo e possiede 32 registri da 32 bit ciascuno, mentre la memoria contiene 2^{32} locazioni da 8 bit, per un totale di 4 GB. Il repertorio deve gestire le seguenti istruzioni:

- **ADD R_i, R_j, R_k** che equivale a $R_k = R_i + R_j$;
- **SUB R_i, R_j, R_k** che equivale a $R_k = R_i$ - R_j;
- **LD R_j, X(R_I)** che equivale a R_J = M[x + R_i];
- **ST R_j, X(R_I)** che equivale a M[x + R_i] = R_J;
- **JEQ R_i, R_j, x** che equivale a if ($R_i == R_j$) ? PC = x : 0; ossia salta se $R_i == R_j$;

5.2 Formato istruzione

Tutte le istruzioni occupano 32 bit e hanno una struttura del tipo:

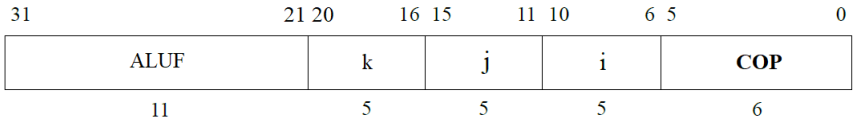


Figura 5.1: Struttura operazioni aritmetiche della macchina monociclo.

Le operazioni aritmetiche possiedono tutte lo stesso codice ma differiscono nel campo ALUF che specifica il tipo di operazione che dovrà eseguire la ALU.

Le operazioni di accesso alla memoria, invece, hanno la forma:

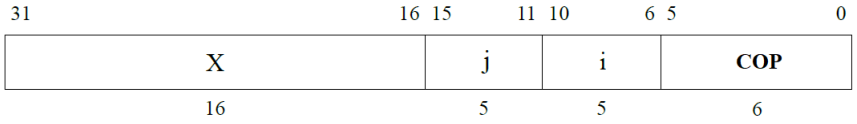


Figura 5.2: Struttura operazioni di accesso alla memoria della macchina monociclo.

Siccome il campo x è a 16 bit, mentre la memoria è a 32 bit, non si possono indirizzare tutte le locazioni in memoria, e bisogna quindi considerare come un offset che spazia da -32k a +32k dato che contiene il segno.

Per le operazioni di salto si ha la medesima forma delle operazioni di accesso alla memoria, e anche in questo caso si hanno solo 16 bit per il campo x, quindi viene considerato salto relativo che diventa una formula del tipo $4 \cdot x + PC + 4 \rightarrow PC$, questo perché le istruzioni sono allineate, cioè partono da posizioni multiple di 4 e per questo la x è un offset. Siccome però i primi due bit sono a 0, perché le istruzioni sono allineate e quindi multiple di 4, si avrà nella pratica un offset a 18 bit, mentre il +4 serve per l'incremento di 4 del PC, infine il ·4 serve per traslare e simulare i 18 bit a disposizione.

5.3 Modello di Harvard

Presenta due tipi di memoria separata, una per le istruzioni e una per i dati, secondo lo schema:

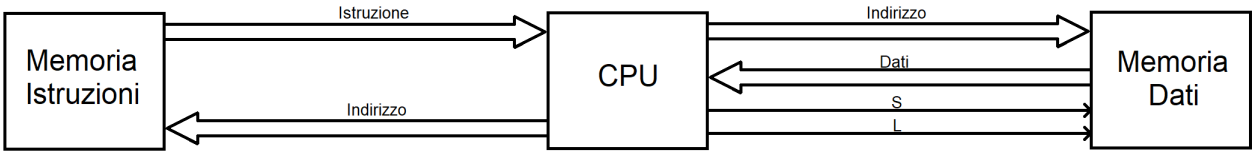


Figura 5.3: Schema di memoria del modello di Harvard.

Nella realtà, la memoria è unica, ma la distinzione è implementata nella CPU che comunica solo con parti della memoria in base alle richieste.

5.4 Architettura

Lo schema della macchina monociclo è il seguente:

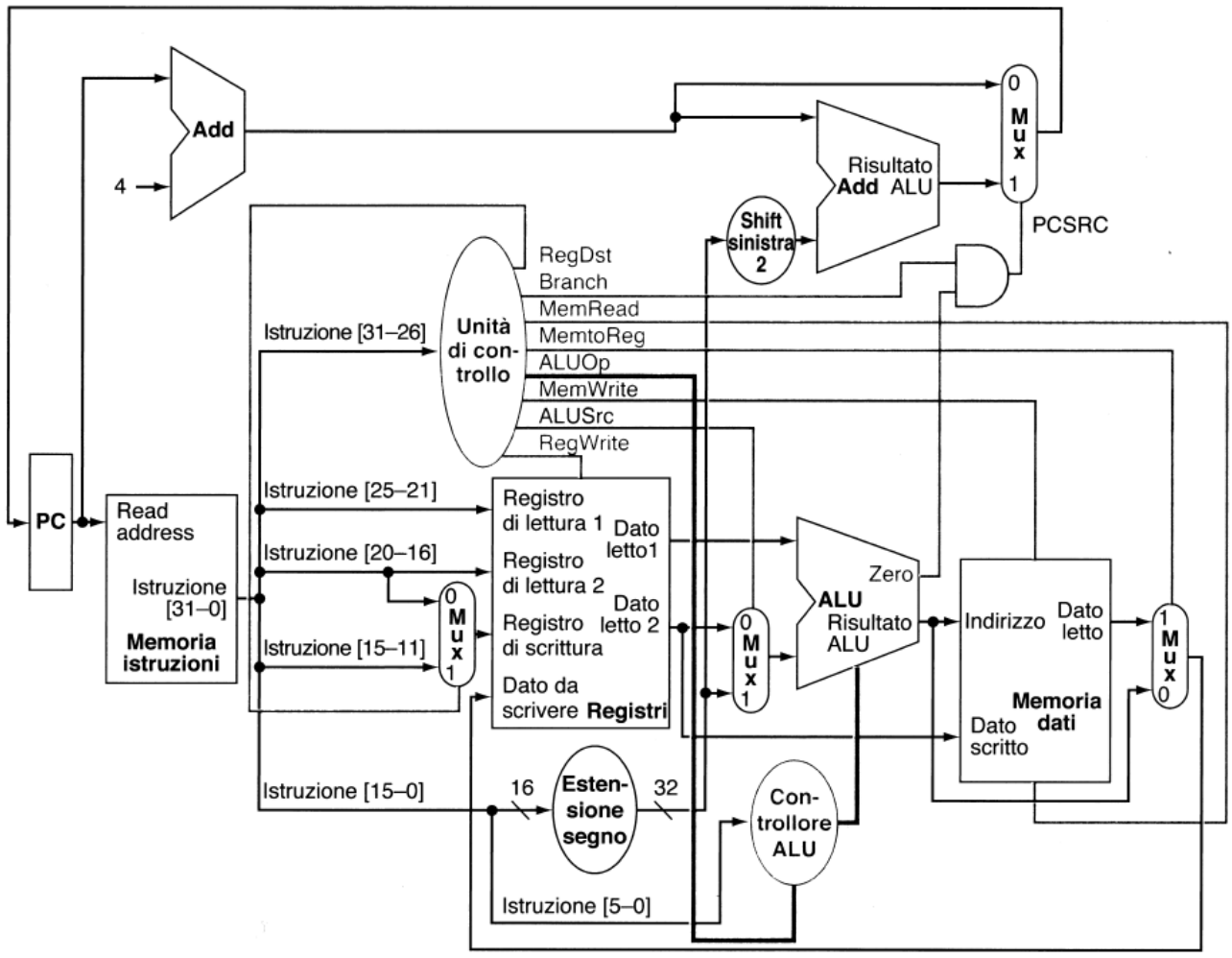


Figura 5.4: Schema della macchina monociclo.

Il **Register pool**, o register file, è il blocco che contiene al suo interno i 32 registri da 32 bit. Le componenti inserite nella macchina devono consentire l'esecuzione di un operazione in un solo ciclo di clock.

I **mux** sono i selettori per stabilire quale segnale inviare. Lo **Shift sinistra** server per trasformare i 16 bit di x in 18 bit. **Estensione segno** serve per trasformare x da 16 bit a 32 bit.

L'unità di controllo è semplicemente una rete a due livelli che si può descrivere direttamente tramite la sua tabella di verità:

Codice operativo	Branch	Registro dest	ALU SRC	Mem to Reg	Register Write	Memory Read	Memory Write	ALU OP	Descrizione
ADD / SUB	0	1	0	0	1	0	0	10	Utilizza ALUF
LD	0	0	1	1	1	1	0	00	Addizione
ST	0	X	1	X	0	0	1	00	Addizione
JEQ	1	X	0	X	0	0	0	01	Sottrazione

Per far funzionare correttamente la macchina, bisogna dare il tempo a ciascuna operazione di poter eseguire le proprie istruzioni. Per questo il clock è dato dalla sommatoria dei ritardi dei singoli componenti $T_{CLK} = \sum_i \tau_i$.

5.5 Confronto tra Monociclo e Multiciclo

Si supponga di avere quattro diverse componenti con differenti tempi d'esecuzione:

$$\begin{cases} \tau_1 = 65 \ ms \\ \tau_2 = 70 \ ms \\ \tau_3 = 90 \ ms \\ \tau_4 = 75 \ ms \end{cases} \tag{5.1}$$

Il tempo d'esecuzione della CPU per entrambe le macchine è il seguente:

$$\frac{T_{CPU}^{Mono}}{T_{CPU}^{Multi}} = \frac{\cancel{N_{ISTR}} \cdot \cancel{C_{PI}^{Mono}} \cdot T_{CK}^{Mono}}{\cancel{N_{ISTR}} \cdot C_{PI}^{Multi} \cdot T_{CK}^{Multi}} = \frac{T_{CK}^{Mono}}{C_{PI}^{Multi} \cdot T_{CK}^{Multi}} = \frac{\sum_i \tau_i}{C_{PI}^{Multi} \cdot \max_i\{\tau_i\}} \tag{5.2}$$

Si possono calcolare i singoli valori per stabilire quale sia il rapporto tra le due macchine:

$$\begin{cases} T_{CK}^{Mono} = \sum_i \tau_i = 300ms \\ T_{CK}^{Multi} = \max_i\{\tau_i\} = 90ms \end{cases} \tag{5.3}$$

Analizzando l'istruzione più lenta, in un ciclo di clock, la macchina monocilo impiega

$T_{CPU}^{Mono} = T_{CK}^{Mono} = 300ms.$

La macchina multiciclo invece necessita di quattro cicli di clock, per tutte e quattro le unità, quindi

$T_{CPU}^{Multi} = 4 \cdot T_{CK}^{Multi} = 4 \cdot 90 = 360ms.$

Per quanto riguarda il repertorio, si può supporre che il 60% delle istruzioni impiega due cicli di clock, mentre il restante 40% ne impiega quattro.

Il numero di cicli di clock medi per la macchina multiciclo è:

$C_{PI}^{Multi} = 0.6 \cdot 2 + 0.4 \cdot 4 = 2.8 \frac{Clock}{Istruzione}$

Mentre la macchina monociclo impiega:

$T_{CPU}^{Mono} = 2.8 \cdot 90 = 256nsec$

Quindi in media è meglio la macchina multiciclo, ma nel caso peggiore, ossia analizzando la componente più lenta, è meglio la macchina monociclo.

L'organizzazione monociclo è la base per un'altra organizzazione detta pipeline.

Capitolo 6

Macchina in Pipeline

6.1 Introduzione

La **Macchina in Pipeline** si ispira alla produzione dei beni materiali, in cui si divide in sequenze di sotto processi la produzione totale, ossia la catena di montaggio.

$$Materie \quad prime \rightarrow Sequenza_1 \rightarrow \cdots \rightarrow Sequenza_k \rightarrow Prodotto \quad finito$$

Il vantaggio, supponendo che τ sia bilanciato tra le sequenze, è che il tempo totale di lavorazione non diminuisce per la singola unità, ma aumenta il throughput, quindi $T = K_\tau$, ma ogni $\tau = \frac{T}{K}$, quindi ogni τ coincide con un semi prodotto.

Per la macchina in pipeline le materie prime sono le istruzioni e il prodotto finito coincide con l'esecuzione dell'istruzione. Tra una sequenza e l'altra, si utilizzano dei registri detti **Registri di pipeline** che garantiscono la corretta sequenzializzazione delle istruzioni. Ogni registro prende il nome dalle sequenze che lo affiancano.

Nella realtà non è vero che ogni sequenza ha lo stesso τ. Tutte le istruzioni devono necessariamente attraversare ogni sequenza, anche se non le necessita tutte.

Per ogni ciclo di clock bisogna dare il tempo ad ogni stato di terminare la propria esecuzione, quindi il tempo totale è $T_{CPU}^{Pipe} = \max_i\{\tau_i\} + \tau_{FF}$.

Il ritardo dei registri si evidenzia perché sono una peculiarità di questa organizzazione, i registri infatti sono tutti uguali e hanno tutti lo stesso ritardo.

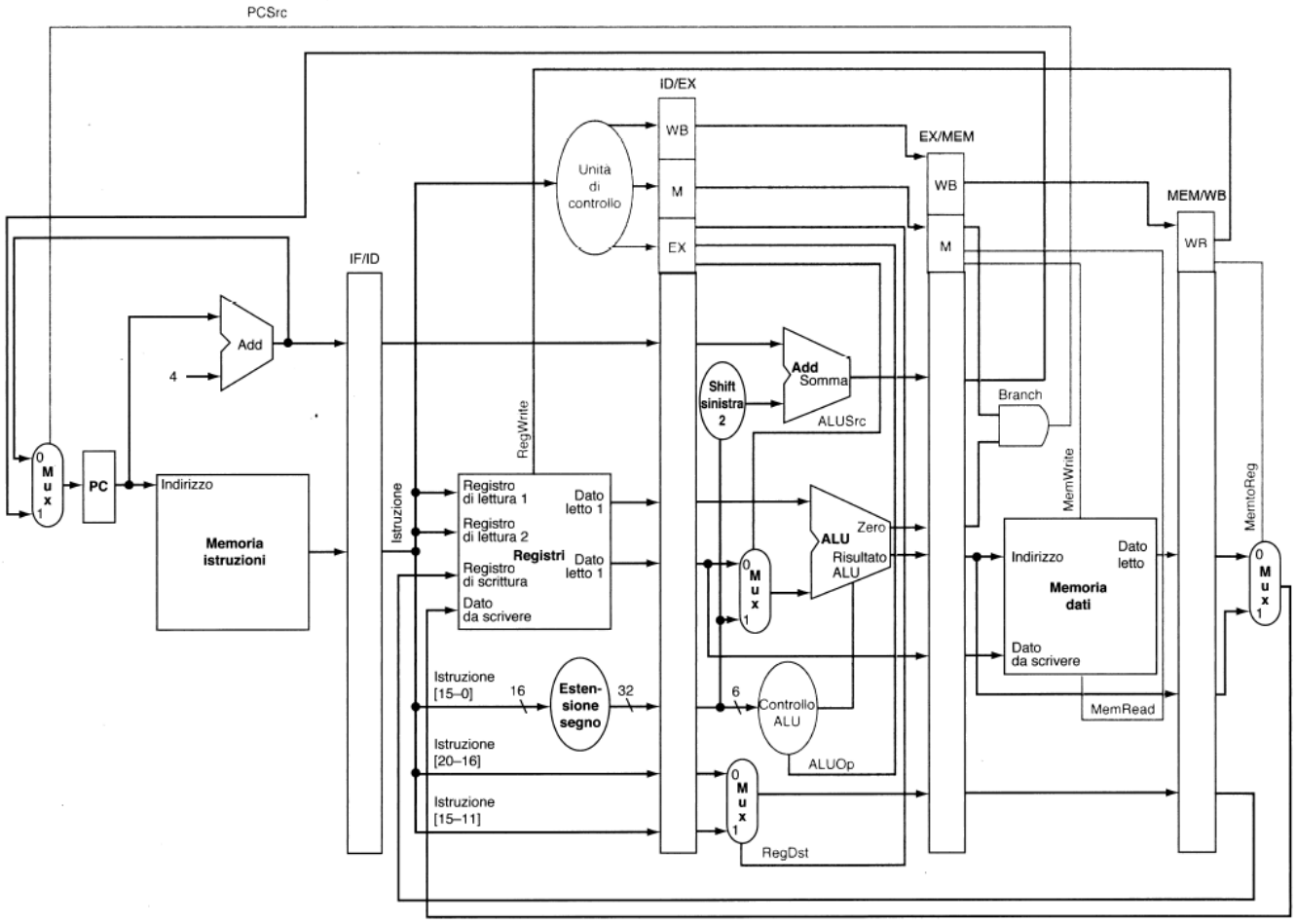
6.2 Architettura

Nella macchina in pipeline bisogna scomporre in vari stati, quindi bisogna inserire i registri di pipeline. L'elaborazione di un'istruzione da parte di un processore si compone di cinque passaggi fondamentali:

- **Instruction Fetch**: (IF) indica la lettura dell'istruzione da memoria;
- **Instruction Decode**: (ID) decodifica l'istruzione e la lettura degli operandi da registri;
- **Execution**: (EX) esecuzione dell'istruzione;
- **Memory**: (MEM) attivazione della memoria;
- **Write Back**: (WB) scrittura del risultato nel registro opportuno.

Il primo stadio è diverso dal secondo grazie al registro *Fetch IF/ID*. Il secondo registro è il decodificatore ID/EX, mentre il registro EX/MEM si occupa dello stato d'esecuzione delle istruzioni. Infine l'ultimo registro MEM/WB delimita lo stato *Memory* e il *Write Back*. Tutti i collegamenti passano dai registri, tranne l'ultimo nella Write Back.

Lo schema della macchina in pipeline è:



Supponendo di avere le seguenti istruzioni:

$$\begin{cases} i_1 & : & LD & R_{10}, 20\{R_1\} \\ i_2 & : & SUB & R_2, R_3, R_{11} \\ i_3 & : & ADD & R_3, R_4, R_{12} \\ i_4 & : & LD & R_{13}, 24\{R_5\} \\ i_5 & : & ADD & R_5, R_6, R_{14} \end{cases} \tag{6.1}$$

è possibile tracciare il diagramma temporale:

	$\mathbf{t}_1$	$\mathbf{t}_2$	$\mathbf{t}_3$	$\mathbf{t}_4$	$\mathbf{t}_5$
$\mathbf{i}_1$	IF	ID	EX	MEM	WB
$\mathbf{i}_2$		IF	ID	EX	MEM
$\mathbf{i}_3$			IF	ID	EX
$\mathbf{i}_4$				IF	ID
$\mathbf{i}_5$					IF

La macchina è divisa in stadi specifici, la Write Back gestisce la scrittura e i segnali di comando viaggiano insieme ai dati nei registri. Si possono quindi eseguire cinque istruzioni contemporaneamente.

6.3 Confronto tra Monociclo e Pipeline

Per confrontare le due macchine bisogna, ancora una volta, calcolare il rapporto dei tempi d'esecuzione delle due macchine:

$$\frac{T_{CPU}^{Mono}}{T_{CPU}^{Pipe}} = \frac{\cancel{N_{ISTR}} \cdot \cancel{C_{PI}^{Mono}} \cdot T_{CK}^{Mono}}{\cancel{N_{ISTR}} \cdot \cancel{C_{PI}^{Pipe}} \cdot T_{CK}^{Pipe}} = \frac{\sum_i \tau_i}{\max_i\{\tau_i\} + \tau_{FF}} \leq \frac{K \cdot \cancel{\tau}}{\cancel{\tau}} = K \tag{6.2}$$

Il rapporto precedente si confronta con K, perché si possono semplificare i τ solo nel caso ideale in cui i ritardi siano bilanciati, e quindi tutti uguali.

Questa relazione evidenzia τ_{FF} (che rappresenta il ritardo dei registri), perché una delle strategie della macchina Pipeline è proprio quella di aumentare gli stati e quindi i registri. Per questo motivo, se τ_{FF} aumenta, non si può ignorare, e ciò avviene all'aumentare dei registri.

Posto il ritardo associato ai registri pari a $\tau_{FF} = 10nsec$, il rapporto precedente diventa

$$\frac{T_{CPU}^{Mono}}{T_{CPU}^{Pipe}} = \frac{300}{90 + 10} = 3 \tag{6.3}$$

Che non coincide con il caso ideale. Questo è dovuto al non perfetto bilanciamento degli stati, infatti, la presenza dei registri della pipeline è un problema caratteristico delle istruzioni, cioé la presenza dei conflitti sui dati o sul controllo.

I prodotti sono indipendenti tra loro, ma in un programma reale ciò non è vero, perché possono esserci salti oppure operazioni sullo stesso dato. Finché il salto non è calcolato, non è possibile stabilire quale istruzione bisogna eseguire, si hanno quindi dei collegamenti all'indietro nella pipeline.

Nella gestione dei conflitti, la macchina si allontana dalla condizione di regime perché si svuota in genere metà macchina e si allontana dal valore di rapporto calcolato precedentemente.

La macchina, allontanandosi dalla condizione di regime, va in una condizione di pipeline vuota in cui non esegue istruzioni per qualche ciclo di clock.

$$\frac{T_{CPU}^{Mono}}{T_{CPU}^{Pipe}} = \frac{n \cdot T_{CK}^{Mono}}{(n + k - 1) \cdot T_{CK}^{Pipe}} \tag{6.4}$$

In base al valore di n si hanno valori diversi del rapporto e la gestione dei conflitti diventa fondamentale per la macchina affinché non si svuoti la pipeline, in modo da non allontanarsi dalla condizione di regime che l'allontanerebbe dal caso ideale.

n	$\frac{T_{CK}^{Mono}}{T_{CK}^{Pipe}}$
5	1.67
10	2.17
100	2.88
$+\infty$	3

6.4 Confronto tra Multiciclo e Pipeline

A regime il rapporto tra i tempi d'esecuzione della CPU per quanto riguarda le due macchine vale:

$$\frac{T_{CPU}^{Multi}}{T_{CPU}^{Pipe}} = \frac{\cancel{N_{ISTR}} \cdot C_{PI}^{Multi} \cdot T_{CK}^{Multi}}{\cancel{N_{ISTR}} \cdot \cancel{C_{PI}^{Pipe}} \cdot T_{CK}^{Pipe}} = \frac{C_{PI}^{Multi} \max_i\{\tau_i\}}{\max_i\{\tau_i\} + \tau_{FF}} \leq C_{PI}^{Multi} \tag{6.5}$$

Che è chiaramente in favore della macchina pipeline.

6.5 Conflitti nella pipeline

I conflitti nella pipeline sono legati al fatto che le istruzioni presentano dipendenze tra loro. Questi conflitti possono essere suddivise in tre categorie:

- **Conflitti strutturali**: si verificano quando la macchina non è in grado di supportare più istruzioni in uno stesso ciclo di clock, questo a causa dell'hardware. Non sono mai presenti nella macchina in pipeline;
- **Conflitti sui dati**: si verificano quando almeno due istruzioni accedono allo stesso dato e almeno uno dei due è in lettura;
- **Conflitti sul controllo**: si verificano in presenza di salti condizionati, perché finché non si verifica lo stato della condizione, non è possibile determinare in anticipo in che modo bisogna continuare.

6.6 Conflitti sui dati

Si supponga di avere due istruzioni del tipo:

i_1: ADD R_0, R_1, R_0 // $R_0 = R_0 + R_1$
i_2: SUB R_2, R_0, R_3 // $R_3 = R_2 + R_0$

Per capire cosa avviene, si analizza la tabella degli stati:

	t_1	t_2	t_3	t_4
i_1	IF	ID	EX	MEM
i_2		IF	ID	EX

i_2 esegue male l'esecuzione, perché utilizza un valore errato di R_0, perché lo legge prima dell'esecuzione a memoria da parte di i_1, quindi si ha un errore di tipo **Read After Write** (RAW).

Bisogna gestire il conflitto sui dati e il modo più comune è lo **Stallo**.

6.6.1 Stallo

Esso consiste nel bloccare l'avanzamento dell'istruzione di conflitto e far continuare le istruzioni che la precedono in modo da far riprendere l'esecuzione appena termina il conflitto. Nell'esempio si blocca i_2 finché non termina l'esecuzione di i_1.

Bisogna inserire un unità unità di rilevamento dei conflitti, che non è altro che un apposito circuito che verifica se ci sono situazioni di conflitto del tipo RAW, e nel caso va ad inserire degli stati aggiuntivi. Viene disabilitata la scrittura del Program Counter, congelando lo stato della pipeline in modo da bloccare i_2, ma consentire l'esecuzione di i_1. Si inserisce una bolla, ossia un istruzione che non fa nulla e non altera lo stato dei registri.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7
i_1	IF	ID	EX	MEM	WB		
i_2		IF	ID	ID	ID	ID	EX
i_3			IF	IF	IF	IF	ID

Le bolle bloccano i_2 e i_3 finché non viene scritto il dato, ciò avviene tra t_5 e t_6. Cosi si risolve il problema, ma il prezzo pagato è lo svuotamento della pipeline.

È la soluzione più drastica di tutte perché si hanno gravi perdite prestazionali, questa è una tecnica generale da utilizzare solo se non esistono alternative utili.

6.6.2 Propagazione

Un'altra tecnica risolutiva è la **Propagazione**. Essa è più efficiente e si basa sul fatto che se si analizzano le due istruzioni, è possibile notare che i_1 non ha ancora scritto R_0, però è anche vero che R_0 è stato già calcolato, quindi sarebbe possibile fornire a i_2 questo valore grazie al circuito di propagazione. Il compito di questo circuito è quello di prelevare il valore di un registro calcolato, ma non ancora sovrascritto, e passarlo all'istruzione che necessita il nuovo valore.

	t_1	t_2	t_3	t_4	t_5
i_1	IF	ID	EX	MEM	WB
i_2		IF	ID	EX	MEM

Il registro EX/MEM contiene il valore calcolato da i_1 per R_0 che viene rilevato dopo t_3, ma intanto i_2 avanza con un valore errato, però l'unità di propagazione cambia il valore di R_0 per i_2 e quando transita per il calcolo, esso conterrà il valore corretto. Non viene bloccata alcuna istruzione, ma semplicemente viene propagato un valore e quindi non si avranno blocchi e stalli, con una conseguente maggiore efficienza rispetto a prima.

Supponendo di avere due istruzioni in conflitto RAW:

i_1: LD $R_0, 0(R_1)$
i_2: SUB R_2, R_0, R_3

Il valore R_0 non è ancora pronto in t_3, quindi non si può propagare il valore e bisogna gestire il problema tramite stallo.

Risulta, però, possibile propagare dallo stato MEM di i_1 allo stato ID di i_2, quindi si può risolvere il conflitto tramite una combinazione di stallo e propagazione, in cui l'unità di propagazione e rilevamento dei conflitti operano in sinergia per risolvere il conflitto.

	t_1	t_2	t_3	t_4	t_5
i_1	IF	ID	EX	MEM	WB
i_2		IF	ID	ID	EX

Un altro esempio utile da analizzare si verifica con le seguenti istruzioni:

i_1: SUB R_1, R_3, R_2
i_2: ADD R_2, R_5, R_{12}
i_3: ADD R_6, R_2, R_{13}
i_4: ADD R_2, R_2, R_{15}
i_5: ST R_{14}, 100(R_2)

Si ha un problema dato che i_1 scrive su R_2 e tutte le altre istruzioni leggono questo valore dal registro. Tra i_1 e i_2 si può risolvere il conflitto tramite propagazione, come nel caso precedente, quindi non serve lo stallo. Stessa cosa accade per i_3 che viene risolto ancora una volta grazie alla sola propagazione.

Per i_4 bisogna introdurre uno stallo e bloccarla, però anche in questo caso il dato necessario è noto, quindi si potrebbe sfasare il segnale di abilitazione che va in AND con il clock. Si potrebbe anche suddividere il clock che viene ritardato in modo da permettere la lettura a metà ciclo di clock, dato che questo viene diviso in due semi periodi, uno per la lettura e uno per la scrittura. In questo modo i_1 scrive R_2 durante il ciclo di clock, in modo da ottenere una propagazione per *sovrapposizione ID/WB* e di conseguenza non esiste più alcuna dipendenza tra i_1 e i_5.

6.6.3 Riordinamento del codice

Tutte le tecniche viste vengono implementate dall'hardware e non dalla CPU, mentre la tecnica del **Riordinamento del codice** viene implementata dal compilatore e serve per annullare i conflitti.

Date le seguenti istruzioni:

i_1: LD R_1, 0(R_0) // R_1 = A
i_2: LD R_2, 4(R_0) // R_2 = B
i_3: ADD R_1, R_2, R_3 // R_3 = A + B
i_4: ST R_3, 12(R_0) // D = A + B
i_5: LD R_4, 8(R_0) // R_4 = C
i_6: ADD R_1, R_4, R_5 // R_5 = A + C
i_7: ST R_5, 16(R_0) // E = A + C

In memoria sono caricati i valori interi di A, B, C, D, E, bisogna eseguire le operazioni D = A + B e E = A + C. Per come è scritto il codice, richiede l'introduzione di due stalli, ma si potrebbe variare l'ordinamento delle istruzioni senza alterare il risultato, in modo da evitare conflitti spostando l'istruzione i_5 prima dell'istruzione i_3. Il nuovo codice è:

i_1: LD R_1, 0(R_0) // R_1 = A
i_2: LD R_2, 4(R_0) // R_2 = B
i_3: LD R_4, 8(R_0) // R_4 = C
i_4: ADD R_1, R_2, R_3 // R_3 = A + B
i_5: ST R_3, 12(R_0) // D = A + B
i_6: ADD R_1, R_4, R_5 // R_5 = A + C
i_7: ST R_5, 16(R_0) // E = A + C

6.7 Unità di rilevamento dei conflitti

Se ci sono delle istruzioni in cui una scrive su un registro e l'altra legge subito dopo, si ha un conflitto e bisogna far intervenire l'unità di rilevamento dei conflitti.

i_1: LD R_0, 1(R_1)
i_2: SUB R_2, R_0, R_3

La condizione da verificare per l'unità di rilevamento dei conflitti è:

$$C_{STALLO} = (ID/EX.MEMREAD) \quad AND \quad ((ID/EX.K = IF/ID.i) \quad OR \quad (ID/EX.K = IF/ID.j)) \tag{6.6}$$

Questa condizione va verificata ad ogni ciclo di clock e, nel caso in cui sia vera, bisogna introdurre una bolla.

	t_1	t_2	t_3
i_1	IF	ID	EX
i_2		IF	ID

6.8 Unità di propagazione

L'unità di propagazione entra in funzione in casi del tipo:

i_1: SUB R_1, R_3, R_2
i_2: ADD R_2, R_5, R_{12}
i_3: ADD R_6, R_2, R_{16}

	t_1	t_2	t_3	t_4	t_5
i_1	IF	ID	EX	MEM	WB
i_2		IF	ID	EX	MEM
i_3			IF	ID	EX

In questo caso il valore di R_2 è già noto, anche se non è stato ancora salvato. L'unità di rilevamento non fa nulla, perché non viene bloccato il codice, ma solo aggiornato il valore del registro R_2 in i_2 e poi in i_3. Le condizioni di propagazione sono due:

1. **Caso 1**:
   $C1_i = (EX/MEM.RegWrite) \quad AND \quad (EX/MEM.K = ID/EX.i)$
   $if(C1_i) \quad then \quad MEM/EX.ALUOUT \rightarrow R_i$
   $C1_j = (EX/MEM.RegWrite) \quad AND \quad (EX/MEM.K = ID/EX.j)$
   $if(C1_i) \quad then \quad MEM/WB.ALUOUT \rightarrow R_j$
2. **Caso 2**:
   $C2_i = (MEM/WB.RegWrite) \quad AND \quad (MEM/WB.K = ID/EX.i)$
   $if(C2_i \quad AND \quad MEM/WB.MemToReg) \quad then \quad MEM/WB.MEMOUT \rightarrow R_i$
   $if(C2_i \quad AND \quad NOT \quad MEM/WB.MemToReg) \quad then \quad MEM/WB.ALUOUT \rightarrow R_i$
   $C2_j = (MEM/WB.RegWrite) \quad AND \quad (MEM/WB.K = ID/EX.j) \quad AND \quad (NOT \quad C1_j)$

6.9 Conflitti sul controllo

Data l'istruzione di salto:

i_1: JE R_1, R_2, 1000

per stabilire quale sarà la prossima istruzione da eseguire, bisogna calcolare il salto, 1000 indica l'offset, e si eseguono due passi:

1. Calcolo della condizione;
2. Calcolo dell'indirizzo di destinazione.

Una prima soluzione per il proseguimento del programma è lo stallo, cioè si blocca i_2 finché non si ha la certezza che i_1 sia terminato e si può stabilire cosi se far continuare i_2 o eseguire l'istruzione in cui punta il salto di i_1.
Esistono varie strategie alternative allo stallo per risolvere questo problema.

6.9.1 Anticipazione del calcolo del salto

Tramite questa tecnica si blocca l'istruzione i_2 per un solo colpo di clock, quindi si ha un unico stallo e poi si inizia la fase esecutiva dell'istruzione corretta.

6.9.2 Svuotamento (Flush)

Si ipotizza che il salto non verrà effettuato, e si verificano due casi:

1. L'ipotesi è corretta e non bisogna eseguire azioni speciali;
2. L'ipotesi è errata e quindi bisogna svuotare gli stati della pipeline contenenti istruzioni errate.

Questo complica lo schema della macchina, ma per migliorare questa tecnica si può utilizzare la predizione.

6.9.3 Predizione

Dato un salto, con la **Predizione** si ipotizza l'esito di tale salto e si suddivide in due macro categorie:

1. **Statica**: Data una sequenza di salti, la predizione non cambia, cioè ogni volta che si incontra questa sequenza si ipotizza la stessa cosa, a sua volta si divide in:
   - **Invariabile**: quando non cambia mai, proprio come lo svuotamento (flush);
   - **Variabile**: quando dipende dal caso, ad esempio tutti i salti all'indietro si ipotizza avvengano mentre i salti in avanti che non avvengano mai.
2. **Dinamica**: L'ipotesi può variare nel tempo incontrando la stessa sequenza.

Per effettuare una predizione dinamica sono necessarie due caratteristiche:

1. **Statistica**: un gruppo di bit associati al salto e la CPU tiene traccia del salto. È un gruppo di bit x associati alla CPU che ad ogni istruzione di salto del programma tiene traccia della sua storia;
2. **Funzione di predizione**: $F(X) = \begin{cases} 1 \Rightarrow Predizione\ \ del\ \ salto\ \ effettuato; \\ 0 \Rightarrow Predizione\ \ del\ \ salto\ \ non\ \ effettuato. \end{cases}$

In un ipotesi di statistica ad 1 solo bit, si ha che $X = \begin{cases} 1 \Rightarrow salto\ \ effettuato; \\ 0 \Rightarrow salto\ \ non\ \ effettuato. \end{cases}$ cioè si tiene traccia della storia e vale $F(X) = X$.

6.9.4 Branch prediction buffer

Dalla predizione si genera la tabella, che deve garantire un accesso costante.

Indirizzo istruzione	Indirizzo destinazione	Statistiche di X	V
PC	PC+4+4*4	X	

Se l'accesso è troppo lento allora la Fetch sarà troppo lenta e viene vanificata la soluzione. La chiave d'accesso della tabella è l'indirizzo istruzione, ossia un suo particolare valore.

Capitolo 7

Macchina super scalare

7.1 Introduzione

Nella **Macchina super scalare**, in assenza di stalli, tutte le operazioni vanno a regime contemporaneamente grazie alla parallelizzazione di più pipeline.

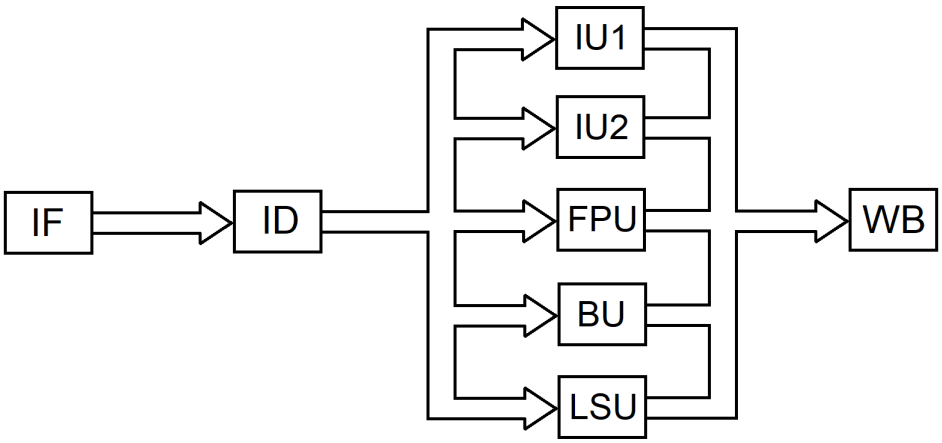


Figura 7.1: Schema della macchina super scalare.

Sono presenti varie unità funzionali in grado di portare a termine le operazioni, tra queste vi è IUx, che rappresenta *Integer Unit*, la FPU *Float Point Unit*, la BU *Branch Unit* e la LSU *Load Store Unit*, ossia lo stato di MEM scompare e LSU prende il suo posto.

Queste unità internamente conservano una pipeline, l'idea è quella di recuparare quell'unico vantaggio rimasto nelle macchine multiciclo, cioè si assegna un valore maggiore alle istruzioni più frequenti.

Sono presenti tre fasi principali:

- **Emissione**: Un istruzione viene emessa quando passa dalla fase di decode ad una delle unità funzionali;
- **Completamento**: Completa lo stato relativo all'unità funzionale;
- **Ritiro**: Completa la fase di Write Back.

Il numero di istruzioni che si possono completare insieme è ovviamente dipendente dalle unità libere. Il confronto tra la macchina pipeline e la super scalare, con h pari al numero di unità funzionali, è:

$$\frac{T_{CPU}^{SS}}{T_{CPU}^{Pipe}} \leq h \cdot K \tag{7.1}$$

7.2 Conflitti sui dati

Nella macchina super scalare tutti i possibili conflitti sui dati, tra cui:

- **Read After Write**: Avviene quando l'istruzione j legge un dato scritto da i;

- **Write After Read**: Avviene quando j scrive un dato letto da i, quindi i non aggiorna correttamente i dati;
- **Write After Write**: Avviene quando ci sono due scritture, una di seguito all'altra, sullo stesso dato.

Il **Reservation Shift Register** (RSR) è una tabella con un numero di righe pari al numero di stadi dell'unità più lenta.

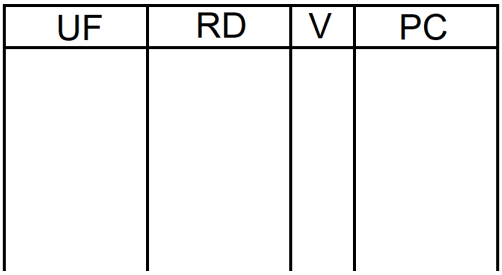


Figura 7.2: Schema RSR.

I campi di RSR rappresentano:

- **UF**: al completamento di una istruzione, ossia all'uscita di RSR, questo campo serve ad individuare da quale UF deve essere preso il risultato da trasmettere a RD;
- **RD**: individua il registro di destinazione del risultato, ma non c'è bisogno di tenere traccia di RD nelle pipeline delle UF;
- **V**: rappresenta il Bit di validità. Informa se la posizione contiene informazioni o se è da ritenersi vuota;
- **PC**: PC dell'istruzione. Necessario per ripristinare uno stato coerente in caso di predizione di salto errata.

7.3 Completamento in ordine

Per garantire il completamento in ordine, bisogna gestire l'RSR. Se le istruzioni in sequenza occupano un numero di ciclo cresente, non si hanno perdite di prestazioni, altrimenti le cose peggiorano.

7.4 Tecnica di riordinamento del buffer

Questa tecnica tenta di preservare un valore recente di istruzioni e crea un **ReOrder Buffer** (ROB). Se non ci fosse un problema di accesso al bus, basterebbe solo il ROB a tenere l'ordine.

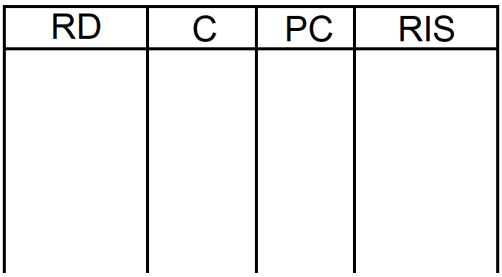


Figura 7.3: Tabella ROB.

Esistono varie strategie da poter applicare e sono le seguenti.

7.4.1 Strategia di riordinamento

La prima tecnica analizzata è la **Strategia di riordinamento**, date le seguenti istruzioni:

i_1: DIVF F_0, F_0, F_1 // Divide due float in FPU 2;
i_2: ADD R_0, R_0, R_1
i_3: SUB R_2, R_0, R_3

Si verifica una dipendenza RAW, e nel buffer di riordinamento si ha:

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
i_1	FPU2 1	FPU2 2	FPU2 3	FPU2 4	FPU2 5	WB			
i_2		IU1	WR				WB		
i_3			Non può essere emessa					IU1	WB

Il problema è che il valore che serve a R_0, non si trova dove dovrebbe essere, ossia nel registro, al momento in cui deve essere calcolato. Bisogna quindi sfruttare la propagazionee bisogna aggiungere una rete di Bypass per risolvere il problema.

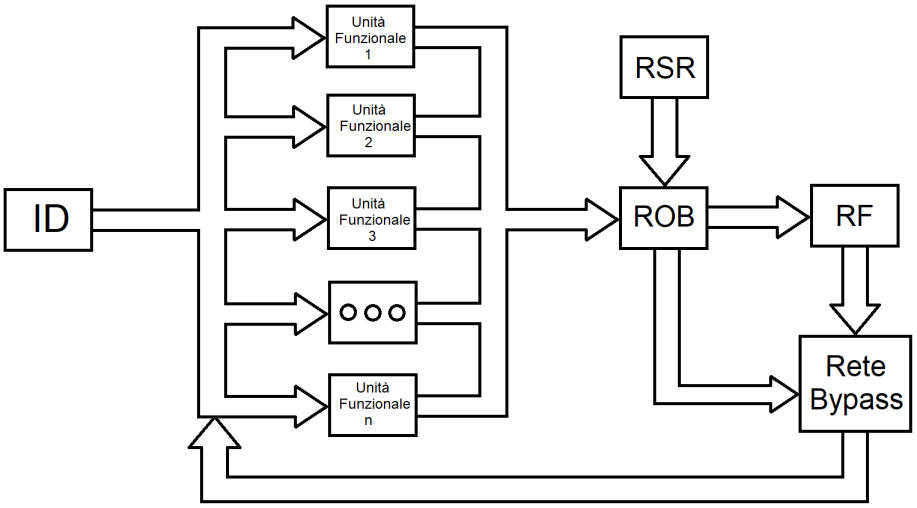


Figura 7.4: Schema Bypass.

La nuova tabella diventa quindi:

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
i_1	FPU2 1	FPU2 2	FPU2 3	FPU2 4	FPU2 5	WB			
i_2		IU1	WR				WB		
i_3			IU1 (Grazie al Bypass)	WR					WB

7.4.2 Completamento poliordine

Supponendo di eseguire il seguente codice su una macchina superscalare:

i_1: DIV R_1, R_2, R_3
i_2: ADD R_4, R_1, R_5
i_3: MUL R_{10}, R_{10}, R_{11}
i_4: MUL R_{12}, R_{12}, R_{13}

La tabella equivalente è:

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7
i_1	IU3 1	IU3 2	IU3 3	IU3 4	IU3 5	WB		
i_2						IU1	WB	
i_3								
i_4								

7.4.3 Esecuzione fuori ordine

Può accadere che un istruzione venga eseguita prima di alcune che la precedono, nel caso precedente si esegue i_3, dato che non presenta dipendenze con i_1 e i_2 finché i_1 non ottiene il valore e può fornirlo a i_2. Nella nuova configurazione compare il *Dispatcher*.

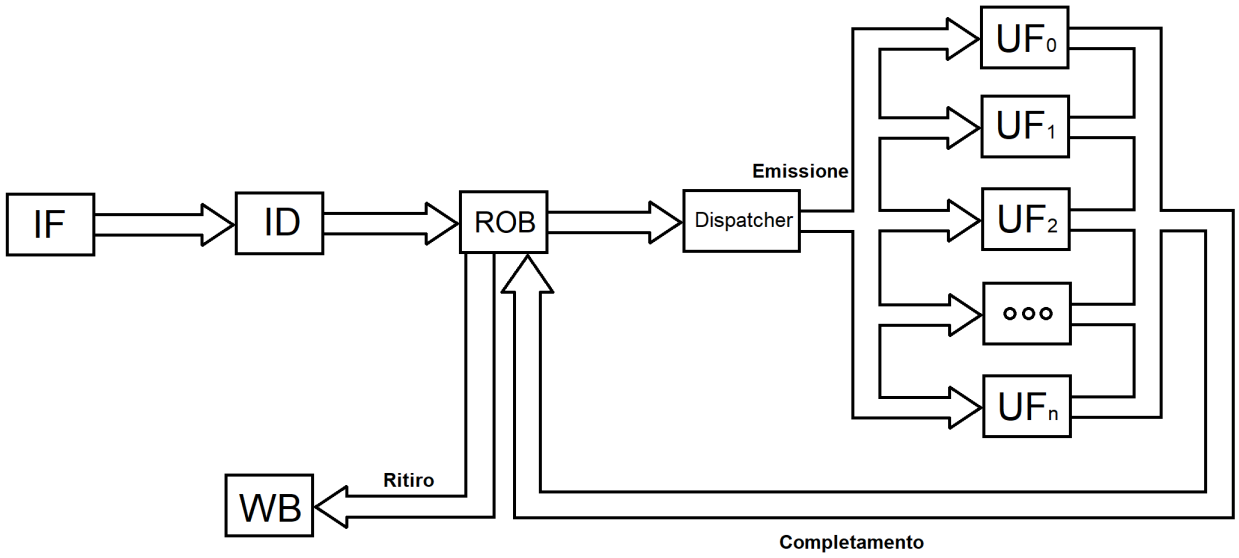


Figura 7.5: Schema esecuzione fuori ordine.

ROB sostituisce il registro d'emissione. Il ROB contiene le istruzioni in attesa d'esecuzione e le istruzioni in attesa di essere emesse. Il dispatcher seleziona la prima istruzione non emessa priva di conflitti strutturali o RAW, secondo una logica detta *Pseudo-FIFO*. Il ROB ha quindi un numero di righe maggiore, perché deve memorizzare anche le istruzioni che sono in attesa che termini il conflitto. Grazie al Dispatcher avviene un meccanismo detto **Ridenominazione dei registri**. Esiste quindi un circuito che identifica le dipendenze WAR e sono presenti anche dei registri nascosti, chiamati T_n, che servono a fare le operazioni in sicurezza fino alla risoluzione dei conflitti.

7.5 Conflitti WAR

Con questa termine si indicano i conflitti **Write After Read**, come avviene ad esempio nel codice:

i_1: DIV R_0, R_2, R_4
i_2: ADD R_6, R_0, R_8
i_3: SUB R_8, R_{10}, R_1
i_4: MUL R_6, R_{11}, R_8

In cui è presente una dipendenza RAW R_0 e una dipendenza prima WAR poi RAW per R_8.

Per far fronte a questo problema, viene utilizzata la tecnica detta **Register Renaming**, ossia ridenominazione dei registri. Viene aggiunta una nuova unità che rileva i conflitti RAW e sostituisce i registri in conflitto con dei nuovi registri, che hanno nomi temporanei creando dei registri nascosti. Nell'esempio, R_8 diventa T_0, ma appena termina le sue esecuzioni, T_0 torna ad essere R_8. Per fare questo serve una tabella che mantiene l'associazione del registro effettivo a quello temporaneo nascosto, e questa tabella viene chiamata **Register Alias Table**.

7.6 Conflitti sul controllo

Questa tecnica viene chiamata **Esecuzione speculativa** e avviene in codici del tipo:

i_1: LD R_1, (1000)
i_2: JE R_1, R_2, 12345
i_3: ADD R_3, R_3, R_4
i_4: ADD R_5, R_5, R_6

In una situazione simile, con un conflitto WAW tra i_1 e i_2, è possibile completare l'esecuzione di i_3 e di i_4 ancor prima di capire se poi vanno effettivamente eseguite o meno.

Capitolo 8

Intel

8.1 Micro architettura Intel

La micro architettura Intel segue due principali periodi:

- **Tick**: introduzione di un nuovo processo produttivo, abbinato ad un'organizzazione già esistente e matura (Die shrink);
- **Tock**: introduzione di una nuova organizzazione abbinata ad un processo produttivo già esistente e maturo.

Intel cerca di introdurre ogni anno un Tick, seguito da un Tock l'anno successivo.

65 nm		45 nm		32 nm		22 nm		14 nm		
Netburst	Tock	Tick	Tock	Tick	Tock	Tick	Tock	Tick	Tock	Tick
	Merom	Penryn	Nehalem	Westmere	Sandy Bridge	Ivy Bridge	Haswell	Broadwell	Skylake	Kabylake
	CORE		NEHALEM		SANDY BRIDGE		HASWELL		SKYLAKE	

Le generazioni precedenti a Netburst sono 8086 (prima generazione), 80286 (seconda generazione), 80386 (terza generazione, 1986), 80486 (quarta generazione, 1989), Pentium (quinta generazione, 1993) e Pentium PRO (sesta generazione).

Tra le architetture menzionate, tutte molto simili, analizziamo meglio l'architettura Haswell.

8.2 Microarchitettura Haswell

Il singolo core viene suddiviso in due parti:

- **Back-end**: Si occupa della Fetch e della decodifica;
- **Front-End**: Si occupa dell'esecuzione;

Il **Translation Lookaside Buffer** (TLB) è una cache che contiene il mapping tra indirizzi fisici e virtuali.

L'**Instruction Length Decoder**, segue un architettura CISC, serve per gestire la compatibilità con l'architettura RISC, ma effettivamente è CISC.

Il decodificatore, preleva le istruzioni dalla macchina e le traduce in sequenze RISC, secondo un repertorio interno alla macchina.

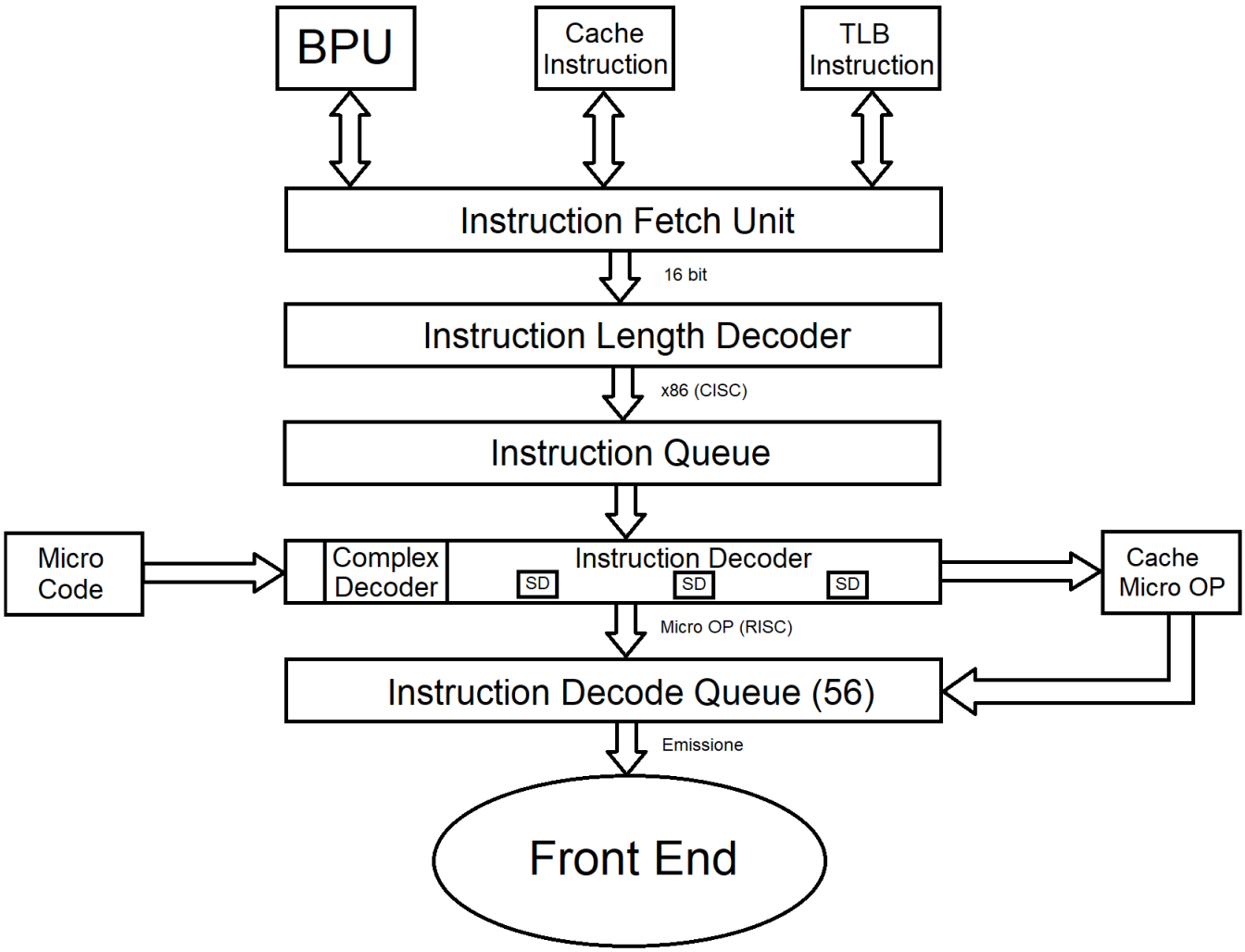


Figura 8.1: Micro architettura Haswell.

Capitolo 9

Macchine Parallele

9.1 Introduzione

Le **Macchine parallele** si suddividono in due famiglie, SIMD e MIMD. Le macchine appartenenti alla famiglia MIMD, a loro volta, sono divise in *Macchine multiprocessore*, dette anche macchine a memoria condivisa, e *Macchine multi computer*, dette anche macchine a scambio di messaggi.

Istruzioni / Dati	**SD**	**MD**
SI	SISD	SIMD
MI	MISD	MIMD

Le macchine MISD non esistono nella realtà, sono delle classiche macchine che operano sul singolo dato. Le macchine multi processore più diffuse sono sicuramente le UMA e le NUMA. Le macchine SIMD sono divise in macchine a processori vettoriali e unità di calcolo vettoriale.

9.2 Multi processori a memoria condivisa

È un sistema in cui sono presenti più CPU che comunicano tra di loro tramite la memoria principale, la quale è condivisa. La comunicazione avviene tramite un canale che nella sua forma più semplice corrisponde ad un BUS, ogni CPU ha la sua memoria cache. Le parti critiche sono la contesa del BUS e la coerenza della memoria, cioè i dati in memoria possono essere replicati in cache e bisogna garantire l'accesso ad una versione aggiornata del dato. Il sistema è *Coerente* se ogni CPU può accedere alla copia aggiornata del dato, indipendentemente da chi lo ha prodotto.

Per garantire la coerenza, si utilizzano i protocollo di *Snooping*, questi sono molto complessi ma in generale tutte le cache intercettano il traffico in viaggio sul BUS. Questo è possibile grazie:

- **Scrittura a invalidazione**: quando la CPU modifica un dato replicato, tutte le altre CPU che hanno quel dato devono invalidarlo. Ha senso quando viene adottata la politica del Write Back;
- **Scrittura a propagazione**: chi modifica il dato spedisce sia l'informazione sulla modifica di un dato, sia il nuovo valore creato propagandolo.

Le linee di cache invece possono essere **Private**, come avviene nella Write Back, oppure **Condivise**, come avviene nella Write Through.

Uno dei protocolli di snooping utilizzato è **Modified Exclusive Shared Invalid** (MESI). Per quanto riguarda la contesa del BUS, se il sistema ha un ordine dell'unità dei processori, influisce poco questo problema, ma se si hanno una decina di CPU le prestazioni crollano.

9.3 Griglia di commutatori

La **Griglia di commutatori** o *Crosspoint*, è una sorta di matrice in cui si collegano da una parte le CPU e dall'altra i moduli di memoria, viene creata cosi una griglia e nei punti d'intersezione vengono posizionati dei crosspoint controllati elettronicamente, che collegano le due linee.

Questo è un esempio di rete non bloccante, perché se la CPU vuole accedere ad un blocco di memoria, può farlo tranquillamente se quest'ultimo è libero, ossia non è acceduto da nessun altra CPU. Il problema di queste reti è il costo elevato, perché il costo è dato dal numero di crosspoint inseriti, quindi $O(n \cdot K)$.

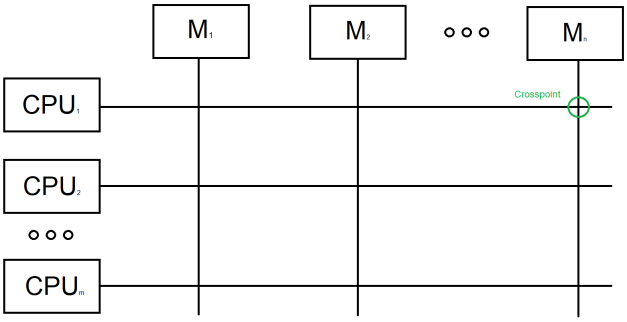


Figura 9.1: Griglia di commutatori.

9.4 Rete Omega

La **Rete Omega** è una rete bloccante, ed è un compromesso tra costi e prestazioni, non garantisce le funzioni della griglia, ma il costo è inferiore $O(\frac{n}{2} \cdot log\ n)$. Queste reti utilizzano commutatori 2D e permettono di creare, in un certo istante, una comunicazione tra un solo ingresso ed una sola uscita.

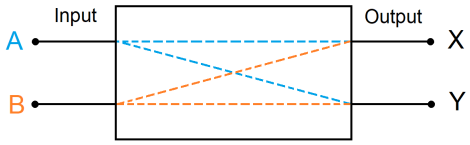


Figura 9.2: Commutatore 2D.

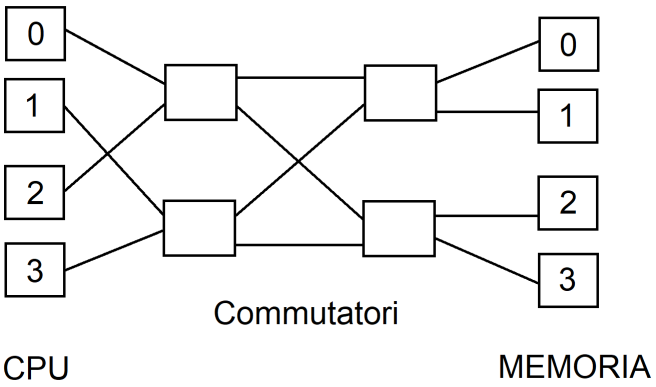


Figura 9.3: Rete Omega.

Le principali macchine parallele, come detto in precedenza, sono:

- **UMA**: le **Uniform Memory Access** comprendono tutti i processori visti fin ora. Il tempo di memoria non varia rispetto alla CPU e alla posizione del dato;
- **NUMA**: le **Non Uniform Memory Access**, sono macchine che non hanno tempi d'accesso costanti.

Nelle macchine NUMA, la memoria è divisa in vari moduli, un blocco più grande "Distante" e più blocchi piccoli "Vicini", uno per ogni CPU. Per accedere alla memoria distante, si utilizza un BUS globale condiviso, mentre per accedere alla propria memoria, si utilizza un BUS "Privato". Un'unità detta **Memory Managment Unit** (MMU) inoltra le richieste alla memoria locale o condivisa, in base alle esigenze. In generale per un programma si cerca di avere tutti i dati utili nella memoria locale e salvare successivamente i dati ottenuti nella memoria condivisa.

9.5 Multicore

I sistemi multi core sono dei sistemi multi processore allocati tutti in un unico dispositivo fisico. Essi hanno tre livelli di memoria cache e seguono il seguente schema:

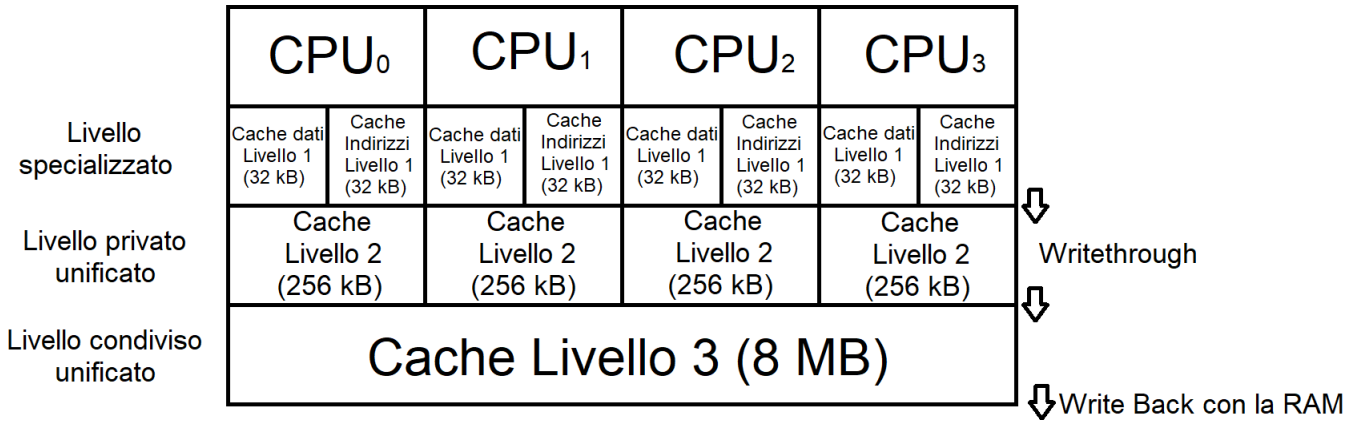


Figura 9.4: Schema multi core.

9.6 MultiThreading Hardware

La CPU riesce a supportare in maniera reale l'esecuzione di più Thread contemporaneamente. Per supportare questa tecnica necessita due copie del Register File, del Program Counter ecc. Ogni Thread ha il proprio stato replicato più volte nella macchina. Il *Context Switch* è immediato, perché si possiedono già tutti i dati richiesti (questa cosa non è vera a livello software) e i thread condividono le stesse unità funzionali.

Esistono tre strategie implementative per il multi threading:

- **Grana fine**: La CPU alterna, ad ogni ciclo di clock, l'esecuzione dei vari thread. Viene servita l'istruzione di un Thread diverso ad ogni colpo di clock. È efficiente in presenza di un elevato numero di thread, perché diminuisce la possibilità di avere stalli. Il lato negativo è la complessità dell'organizzazione, perché bisogna far avanzare tante istruzioni provenienti da thread diversi;
- **Grana grossa**: In ogni istanze sarà in esecuzione un unico thread e si passa all'altro thread solo quando si ha uno stallo nel thread in esecuzione;
- **Multithreading simultaneo**: Viene implementata nelle macchine super scalari e si parla di **Hyper-Threading** e serve ad aumentare l'utilizzo delle unità. Può accadere che un processo sotto utilizza le unità funzionali, quindi vengono eseguiti più thread e la fetch preleva dati da due processi simultaneamente. Sono presenti quindi 4 core fisici e 4 logici provenienti dal doppio processo presente in ogni core. Si ha un vantaggio prestazionale del 25% dato che il raddoppiamento dei processori è logico e non fisico.

9.7 Graphic Processing Unit

Le GPU sono dei dispositivi *Many core*, perché hanno un elevato numero di core. Indica le unità nate per operazioni grafiche ma poi modificate per sfruttare elaborazioni non prettamente grafiche. Le caratteristiche principali sono il supporto ad un gran numero di elaborazioni in virgola mobile (1.3 TFlops, nelle GPU NVIDIA Tesla 52090 del 2010), hanno un elevato numero di core (512 nelle NVIDIA Tesla), hanno un numero elevato di thread hardware (24576 thread, di cui solo 512 in esecuzione contemporaneamente a massimo nelle NVIDIA Tesla), quindi vengono dette *Heavily multi thread*, e hanno un elevata larghezza di banda (177 GB/s nelle NVIDIA Tesla).

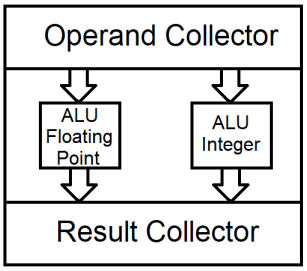


Figura 9.5: Graphic Process Unit.

Le GPU sono **Single Instruction Multiple Thread** (SIMT) ossia implementano una computazione SIMD ma supportata da più thread. Per esempio, le GPU NVIDIA utilizzano il linguaggio **Compute Unified Device**

Architecture (CUDA) che si basa sull'utilizzo del linguaggio C con l'aggiunta di alcune funzioni kernel eseguite dalla GPU, mentre tutto il resto si esegue nella CPU. Le funzioni Kernel sono anche scritte in C. In ogni istante viene eseguita una singola istruzione.

Capitolo 10

Assembly x86

10.1 Processori 8086

L'architettura nasce nel 1978, è un architettura dotata di registri generali a 16 bit. Questi registri sono otto: **AX**, **BX**, **CX**, **DX**, **SI**, **DI**, **BP**, **SP**, rispettivamente, accumilatore, base, contatore, dati, source index, destination, base pointer e stack pointer. I primi quattro registri sono divisi a loro volta in 2 registri a 8 bit, High e Low (es AH e AL).

Questa architettura subisce una prima estensione con il processore 80386 nel 1985, con l'architettura x86-32 e i registri diventano a 32 bit raddoppiando la dimensione dei precedenti otto che diventano **EAX**, **EBX**, **ECX**, **EDX**, **ESI**, **EDI**, **EBP**, **ESP**, dove E sta per *Extended*.

Un'altra estensione è chiamata x87 nel 1980 che introduce i numeri a virgola mobile che operano tramite un *Coprocessore* che esegue il calcolo di numeri a virgola mobile, vengono inoltre introdotti otto nuovi registri **ST(0)**, **ST(1)**, **ST(2)**, **ST(3)**, **ST(4)**, **ST(5)**, **ST(6)**, **ST(7)**, che sono registri a 80 bit, funzionano a stack.

Un ulteriore estensione è la **Multimedial EXtensions** MMX nel 1997, serve per garantire un supporto alle operazioni multimediali e introduce un nuovo set di otto registri a 64 bit, supportano il calcolo di timo SIMD e possono contenere più dati. Sono registri che lavorano con i soli interi e possono immagazzinare un intero da 64 bit, oppure due interi a 32 bit, fino a sei interi a 8 bit. Questi registri sono **MM0**, **MM1**, **MM2**, **MM3**, **MM4**, **MM5**, **MM6**, **MM7** ma sono limitati perché non supportano i numeri a virgola mobile, usati nelle operazioni multimediali. Questi registri sono fisicamente implementati sopra i registri ST degli x87 a virgola mobile, quindi non si possono utilizzare contemporaneamente.

Nel 1999 viene introdotta l'estensione **Simd Streaming Extensions** (SSE) per risolvere i problemi di MMX. Si aggiungono i registri a 128 bit, otto registri chiamati **XMM0**, **XMM1**, **XMM2**, **XMM3**, **XMM4**, **XMM5**, **XMM6**, **XMM7** che supportano il calcolo SIMD e a virgola mobile e non hanno nulla in comune con gli MMX se non il nome.

L'architettura x86-64 (per AMD) nel 2003/2004 estende ulteriormente i registri portandoli a 64 bit, supportano la retrocompatibilità dei registri x86-32 che si chiamano ora **RAX**, **RBX**, **RCX**, **RDX**, **RSI**, **RDI**, **RBP**, **RSP**, ma vengono anche raddoppiati in numero, i nuovi registri sono **R0**, **R1**, **R2**, **R3**, **R4**, **R5**, **R6**, **R7**, **R8**, **R9**, **R10**, **R11**, **R12**, **R13**, **R14**, **R15**. Si possono utilizzare i 32 bit meno significativi, tramite i registri **R8D**, ... , **R15D** dove D sta per *Doubleword*, oppure i 16 bit meno significativi, tramite **R8W**, ... , **R15W**, perché W indica *Word*.

Nel 2011 viene introdotto **Advanced Vector Extensions** (AVX) che amplia i registri XMM a 256 bit, che per estensione diventano **YMM0**, **YMM1**, **YMM2**, **YMM3**, **YMM4**, **YMM5**, **YMM6**, **YMM7**, e raddoppiano anche essi in numero aggiungendo i registri **YMM8**, **YMM9**, **YMM10**, **YMM11**, **YMM12**, **YMM13**, **YMM14**, **YMM15**, e per questione di retrocompatibilità, possono essere visti come **MM8**, ... , **MM15**.

I moderni processori utilizzano registri a 512 bit, hanno 32 registri totali per l'architettura AVX512, con i registri **ZMM0**, ..., **ZMM31**. Questi nuovi registri sono per ora implementati nei coprocessori e non effettivamente nei core.

10.2 Indirizzamento

Si utilizza un modello Registro-Memoria con operazioni del tipo <OP> <Registro>, <Registro/Memoria>, che equivale ad avere $Registro1 = Registro1 < OP > Registro2/Memoria$.

Per effettuare un indirizzamento, la sintassi da utilizzare è $< Base > + < Scala > \cdot < Indice > + < Offset >$, in cui Base risiede nei registri generali, Scala è una costante che può valere 1,2,3,4 oppure 8, Indice è un altro registro generale, mentre Offset è ancora una costante.

10.3 SSE

Per poter operare con i floating point, è necessario conoscere la precisione da utilizzare. Per la precisione singola si utilizzano 32 bit (float) mentre per la precisione doppia si utilizzano 64 bit (double).

	Precisione singola	**Precisione doppia**
Scalare	SS	SD
Packed	PS	PD

Queste sigle si utilizzano come suffissi nelle operazioni che specificano il modo di operare. Ad esempio, avendo a disposizione un registro a 128 bit, se si utilizza il suffisso PS, allora si intende utilizzare il registro come un vettore a quattro campi da 32 bit, mentre con il suffisso PD si utilizza il registro come un vettore a due elementi da 64 bit. Nella modalità Scalare SS si indica l'utilizzo del registro come vettore a 32 bit, mentre tramite SD si desidera utilizzare il registro come vettore con un unico elemento a 64 bit.

10.4 Istruzioni di trasferimento dati

Per effettuare gli spostamenti dei dati, si utilizza l'operazione **MOV** che indica *Move* e i casi possibili sono i seguenti:

MOV $\{^{A}_{U}\}$ $\{^{PS}_{PD}\}$ $\{^{<XMM>, <XMM, MEM128>}_{<XMM, MEM128>, <XMM>}\}$

La lettera A indica **Aligned**, ossia un dato allineato si ha quando l'indirizzo a partire dal quale il dato è memorizzato, risulta essere multiplo intero della sua dimensione espressa in termini di numero di locazioni. U invece indica **Unaligned**, un dato non allineato si ha quando la posizione di partenza non è multiplo della dimensione espressa in numero di locazioni.

Il supporto d'accesso ai dati allineati è più efficiente di quello non allineato. Se si cerca però di utilizzare una move allineata su dati non allineati, il programma termina con errore. Nella move scalare non vi è differenza tra dati allineati e non.

10.5 Istruzioni aritmetiche

La sintassi da utilizzare per le istruzioni aritmetiche è la seguente:

<OP> $\{^{PS}_{PD}\}$ <XMM>, <XMM, MEM128>

Il primo registro è la destinazione, il secondo registro e la memoria sono le sorgenti, inoltre la memoria è allineata. Le operazioni possibili sono : <OP> = {ADD, SUB, MUL, DIV, MAX, MIN, SQRT, RCP}. Le operazioni a loro volta si dividono in due gruppi:

- **Binarie**: hanno la forma <destinazione> = <destinazione> <OP> <sorgente>. Appartengono a questo gruppo le operazioni di ADD, SUB, MUL, DIV, MAX e MIN;
- **Unarie**: hanno la forma <destinazione> = <OP> <Sorgente> e fanno parte di questo gruppo le restanti due operazioni, ossia SQRT e RCP.

10.6 Code Vectorization

La **Code Vectorization** è una trasformazione nel calcolo SIMD che consiste nella sostituzione di sequenze di codice del tipo a[i] = b[i] + c[i], ... , a[i+p-1] = b[i+p-1] + c[i+p-1], ossia sequenze di istruzioni identiche che lavorano su dati diversi.

Si può supporre che la macchina sia in grado di lavorare su p operazioni parallelamente, ossia a[i ... i+p-1] = b[i ... i+p-1] + c[i ... i+p-1], e ciò viene detto vettorizzazione del codice.

Più in generale si ha un operazione che coinvolge n elementi, ad esempio:

```
for(i=0; i<n; i++)
        a[i] = b[i] + c[i];
```

Viene definita **Loop vectorization**, la scomposizione del ciclo in due cicli, il primo è detto ciclo quoziente e sarà:

```
for(i=0; i<n; i+=p)
        a[i] = b[i] + c[i];
        ...;
        a[i+p-1] = b[i+p-1] + c[i+p-1];
```

Ottenuto mediante la code vectorization, e a seguire il ciclo resto che opera sugli ultimi valori restanti:

```
for(i=p⌊n/p⌋; i<n; i++)
        a[i] = b[i] + c[i];
```

Supponendo p=4 e n=10 si otterrà un quoziente q=2 (Packed) e un resto r=2 (Scalar). Aggiungendo un numero di elementi fittizi, tutti posti pari a 0, per far in modo da avere un multiplo di p come numero di elementi di n si può sfruttare a pieno il Packed. Aggiungendo 2 elementi a 0 quindi si passa a n=12, che è multiplo di p, così da eliminare il ciclo Resto. Questa tecnica viene detta **Padding**.

10.7 Riduzione

Data una sequenza di valori $S = x_i, \ldots, x_n$ e un operatore binario associativo, la **Riduzione** di S restituisce un solo valore ottenuto applicando ripetutamente l'operatore agli elementi di S.

Ad esempio, sia $S = \sum_i x_i$, oppure $P = \prod_i x_i$, oppure $M = max\{x_i\}$, oppure $m = min\{x_i\}$, avendo a disposizione p unità che operano in parallelo, è possibile effettuare la riduzione di $n = 2 \cdot p$ elementi secondo una logica ad albero che ha una profondità pari a $\log_2 2 \cdot p = 1 + \log_2 p$. Questo perché si dimezza il numero di elementi ad ogni passo.

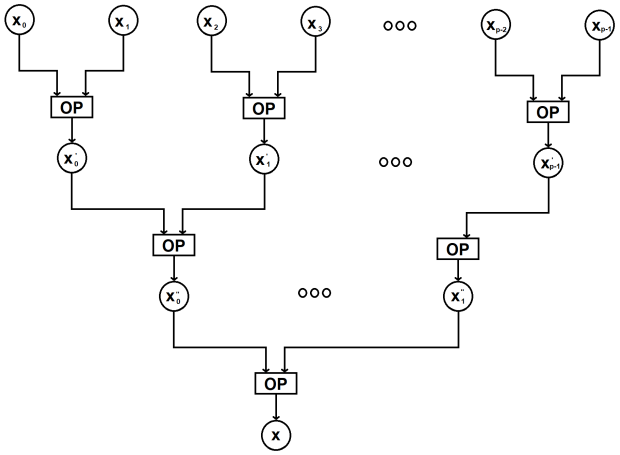


Figura 10.1: Albero di riduzione.

Avendo $n > 2 \cdot p$ si ha la sequenza $S = \sum_{i=0}^{n-1} x_i$ con $p = 4$ e servono n accumulatori. Però per riuscire ad effettuare la riduzione parallela in linguaggio macchina, serve una nuova istruzione di somma, ossia l'Halfadder.

10.8 Halfadder

L'istruzione **Halfadder** serve per sommare i componenti delle sorgenti ed inserirli nella destinazione secondo il seguente schema:

Mentre la sintassi è:

HADD $\{^{\text{PS}}_{\text{PD}}\}$ <XMM>, <XMM, MEM128>

10.9 Halfsubtractor

L'istruzione **Halfsubtractor** è identica all'Halfadder solo che sottrae gli elementi della sorgente e li inserisce nella destinazione. La sintassi è:

HSUB $\{^{\text{PS}}_{\text{PD}}\}$ <XMM>, <XMM, MEM128>

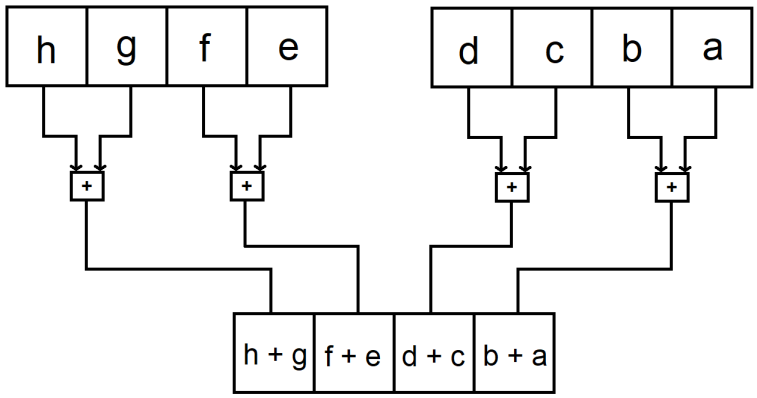


Figura 10.2: Schema Halfadder.

10.10 MOVLHPS / MOVHLPS

Se si vuole risolvere una produttoria del tipo $\prod_i x_i$, non è possibile utilizzare funzioni del tipo HADD per la moltiplicazione perché non esiste. Bisogna quindi introdurre due nuove operazioni:

MOVLHPS <XMM>, <XMM>

MOVHLPS <XMM>, <XMM>

Queste sono istruzioni che non accedono alla memoria, non spostano i campi a 32 bit ma blocchi di 64 bit. Le lettere L e H, rispettivamente Low e High, indicano che si vuole spostare la parte bassa della sorgente la parte alta della destinazione, e viceversa per HL. Si copia la parte bassa della sorgente nella parte alta della destinazione.

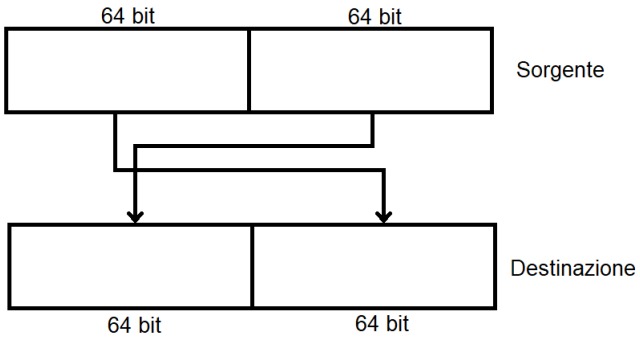


Figura 10.3: Istruzione MOVLHPS.

10.11 MIN / MAX

Per calcolare minimo e massimo si utilizzano le istruzioni **MINPD** e **MAXPD** per la versione Packed, mentre **MINSD** e **MAXSD** per la versione Scalar.

10.12 MOVSHDUP / MOVSLDUP

Le istruzioni **MOVSHDUP** e **MOVSLDUP** sono due istruzioni di movimento, si può intuire da MOV, che operano a precisione singola, intuibile da S. Le lettere L e H indicano rispettivamente la parte bassa, Low, e la parte alta, High. Entrambe operano tramite duplicazione, per questo il suffisso è DUP.

Si utilizzano quattro campi a 32 bit e si prendono le parti alte duplicandole nella destinazione, per quanto riguarda MOVSHDUP. L'istruzione duale MOVSLDUP, invece, copia le parti basse nella destinazione.

Le sintassi sono rispettivamente:

MOVSHDUP <XMM>, <XMM, MEM128>

MOVSLDUP <XMM>, <XMM, MEM128>

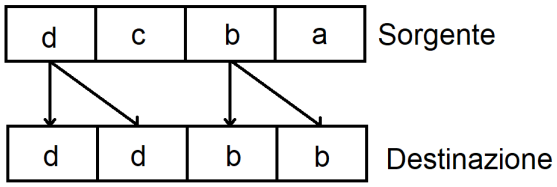


Figura 10.4: Istruzione MOVSHDUP.

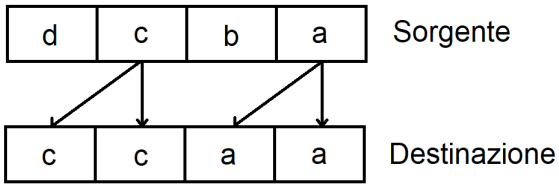


Figura 10.5: Istruzione MOVSLDUP.

10.13 MOVDDUP

Questa istruzione è analoga alle istruzioni precedenti, con l'unica differenza che questa opera su campi a 64 bit e non a 32. La sintassi è

MOVDDUP <XMM>, <XMM, MEM64>
Si indica MEM64 perché si può prendere un valore a 64 bit dalla memoria e inserirlo nella destinazione.

10.14 SHUFPD

L'istruzione **SHUFPD** è un operazione che mescola i valori, infatti SHUF sta per *Shuffle*. Opera su tre valori di cui l'ultimo è un numero a 8 bit codificato dentro l'istruzione, detto *immediato*. L'operazione lavora a 64 bit e la sintassi è:

SHUFPD <XMM>, <XMM, MEM128>, <IMM8>

L'operazione copia in base al bit meno significativo di IMM8, cioè, dato l'esempio in figura:

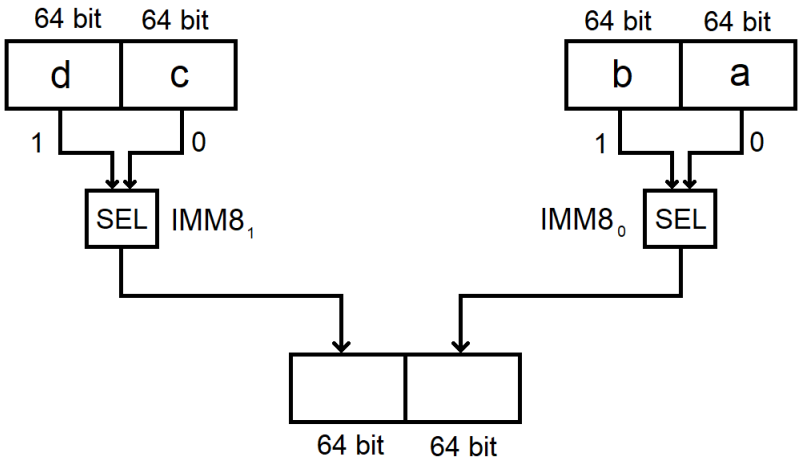


Figura 10.6: Istruzione SHUFPD.

Si analizza il bit meno significativo di IMM8, in modo che se $IMM8_0 \Rightarrow \begin{cases} 0 & copia & a \\ 1 & copia & b \end{cases}$. Vale la stessa cosa per la parte superiore, cioè se $IMM8_1 \Rightarrow \begin{cases} 0 & copia & c \\ 1 & copia & d \end{cases}$.

Scrivendo lo stesso registro sia nella destinazione che nella sorgente, si invertono parte alta e parte bassa.

10.15 SHUFPS

L'istruzione **SHUFPS** è la versione a 32 bit del caso precedente, ma è più complessa. La sintassi è:

SHUFPS <XMM>, <XMM, MEM128>, <IMM8>

Se la base non è specificata, allora l'immediato è inteso in base 10, altrimenti si può specificare un'altra base diversa.

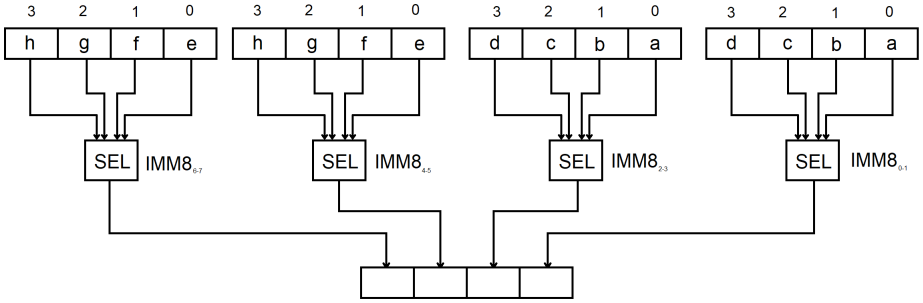


Figura 10.7: Istruzione SHUFPS.

I valori del campo IMM8, indicano cosa selezionare tra i quattro valori, ad esempio il per il primo valore si avrà 00, per il secondo 01, per il terzo 10 e per il quarto 11.

Si utilizza questa istruzione per invertire i valori di un registro, passando ad esempio da a, b, c, d alla configurazione d, c, b, a. Si può anche copiare il primo valore (quello in posizione 0) su tutti i campi del registro.

10.16 ADDSUB

Se si vuole eseguire sia la somma che la sottrazione tra i campi di un registro si utilizza l'istruzione **ADDSUB**. La sintassi è:

ADDSUB $\{^{\text{PS}}_{\text{PD}}\}$ <XMM>, <XMM, MEM128>

Quest'istruzione ha la seguente struttura:

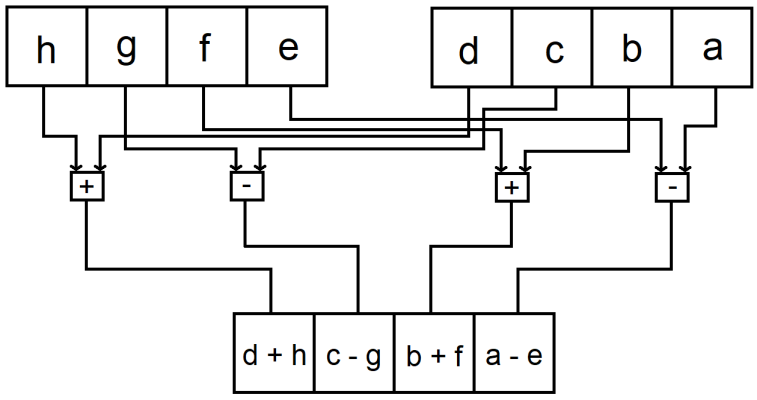


Figura 10.8: Istruzione ADDSUB.

10.17 MOV (64 bit)

Una nuova istruzione utile da analizzare è sicuramente la MOV capace di spostare blocchi da 64 bit. La sua sintassi è:

MOV $\{^{\text{L}}_{\text{H}}\}$ $\{^{\text{PS}}_{\text{PD}}\}$ $\{^{\text{<XMM>, <MEM128>}}_{\text{<MEM128>, <XMM>}}\}$

In alternativa si potrebbe utilizzare la MOVSD, che prende i 64 bit meno significativi, ma non esiste l'equivalente della MOVH. Utilizzare PS o PD non fa alcuna differenza. Utilizzando H, si vuole indicare da dove verrà preso il dato da spostare.

10.18 Istruzioni logiche

Le istruzioni logiche possibili sono **AND**, **ANDN**, **OR** e **XOR**, con le rispettive versioni PS, PD, SS ed SD.

<Istruzione logica> <Precisione> <XMM>, <XMM, MEM128>

Consentono di eseguire le operazioni logiche bit a bit tra la sorgente e la destinazione in modo scalare o vettoriale. $Dest_i = dest_i < OP > sorg_i \; \forall biti$ in cui $< OP > \in \{AND, OR, XOR\}$.

Utilizzando l'istruzione XOR, se la sorgente coincide con la destinazione, allora ciò consente di azzerare il registro.

10.19 Confronti

Le operazioni di confronto servono per eseguire confronti tra registri, oppure tra registro e memoria. La sintassi è:

CMP <Tipologia> $\{^{\text{PS}}_{\text{PD}}\}$ <XMM>, <XMM, MEM128>

Le tipologie di confronto sono le seguenti: **EQ** per l'uguaglianza, **NEQ** per la negazione, **LT** per il minore, **LE** per il minore uguale, **NLT** per il maggiore uguale ed infine **NLE** per il maggiore.

10.20 Principio di località

La CPU non comunica direttamente con la RAM, quindi lettura e scrittura avvengono tramite la memoria cache che è più piccola e più veloce. Viene chiamato **Cache hit** quando la ricerca di un dato richiesto va a buon fine nella cache, altrimenti se la ricerca fallisce si ha un **Cache miss** e la cache deve interfacciarsi con la RAM per reperire il dato.

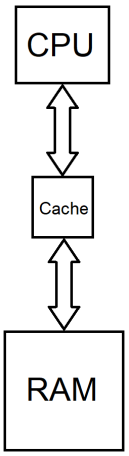


Figura 10.9: Connessioni CPU, cache e RAM.

La cache ha un sotto insieme dei dati della RAM e basa il proprio funzionamento sul **Principio di località** che si divide in due tipologie:

- **Località spaziale**: Se la CPU accede ad una certa locazione x, allora è altamente probabile che la prossima locazione ad essere acceduta sarà contigua a x;
- **Località temporale**: Se la CPU accede alla locazione x, allora è altamente probabile che nell'immediato futuro accederà nuovamente alla locazione x.

In presenza di cache miss, accede alla memoria RAM e porta al suo interno il dato per località temporale, ma porta anche un gruppo di locazioni contigue al dato x per località spaziale, in modo da favorire il tasso di cache hit.

10.21 Accessi sequenziali in memoria

Gli accessi sequenziali in memoria servono per sfruttare il principio di località. Ad esempio, dato il seguente codice:

```
const int n = 1000;
float a[n][n]; //matrice n·n float (4 Byte) in Raw Major Order (ordinamento per riga) allocata nello stack
for(int j = 0; j < n; j++)
    for(int i = 0; i < n; i++)
        a[i][j] = i * j;
```

Essendo linearizzata per righe, l'accesso avviene in locazioni non contigue tra loro e implica una forte presenza di miss. Se si esegue un **Loop interghange** cioè si inverte l'ordine dei for il codice diventa:

```
const int n = 1000;
float a[n][n]; //matrice n·n float (4 Byte) in Raw Major Order (ordinamento per riga) allocata nello stack
for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
        a[i][j] = i * j;
```

Si ottiene così un accesso sequenziale e si aumenta il tasso di hit massando da un miss dell'11% (nel primo caso) ad un miss dello 0,7% dopo l'interchange.

Si aumentano le prestazioni del codice creando matrici non array di array, anche matrici automatiche che hanno il problema della dimensione, ma si crea un array che contiene i dati secondo una convenzione scelta dal programmatore

10.22 Matrici in C

Si può creare un puntatore ad un puntatore, cioè una matrice array di array. Per allocarla in memoria si può utilizzare la funzione *calloc*, un codice generico possibile è:

```
float **a;
a = calloc(n, sizeof(float*));
for(int i = 0; i < n; i++)
    a[i] = calloc(n, sizeof(float));
```

Per evitare problemi dal punto di vista cache, conviene allocare un unico blocco di memoria contigua e bisogna scegliere come accedere ai dati, se scriverli per rigo a per colonna, passando da due indici ad un unico indice. Si analizzi lo pseudocodice del prodotto tra matrici:

```
void prod(float * A, float * B, float * C, int n);{
    int i, j
    for(int i = 0; i < n; i++)
        for(int j = 0; j < n; j++)
            float t = c[i][j];
            for(int k = 0; k < n; k++)
                t+ = A[i][k] * B[k][j];
            c[i][j] = t;
        }
}
```

La prima domanda da porsi è come acceder ai dati della matrice, per per riga o per colonna. L'indice i si muove per righe di A e C, mentre k scorre le colonne di A e le righe di B, infine j scorre le colonne di B. Lo scorrimento avviene lungo le colone di B per ogni riga di A:

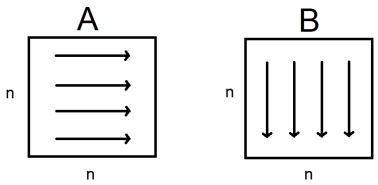


Figura 10.10: Scorrimento matrici A e B.

Si possono utilizzare le istruzioni SIMD per rendere vettoriale la matrice, per il momento si sta eseguendo il prodotto di una riga fissata di A per tutta la matrice B. È possibile parallelizzare il codice eseguendo p righe di A contemporaneamente, per la matrice B. Il nuovo codice è il seguente:

```
void prod(float * A, float * B, float * C, int n); {
    int i, j
    for(int i = 0; i < n; i+ = p)
        for(int j = 0; j < n; j + +)
            float[p] = t[0 ... p − 1];
            float t = c[i][j];
            for(int k = 0; k < n; k + +)
                t[0 ... p − 1]+ = A[i ... i + p][k] * B[k][j];
            c[i ... i + p − 1][j] = t[0 ... p − 1];
        }
}
```

10.23 Loop unrolling

Il **Loop unrolling** tenta di migliorare il tempo d'esecuzione a discapito della lunghezza del codice. Sfrutta il parallelismo implicito favorendone lo sfruttamento della pipeline. Il codice è più lungo ma vengono eseguite meno istruzioni. Dato il codice:

```
for(int i = 0; i < n; i + +) //costo istruzioni aggiornamento del ciclo pari al 30%
    a[i] = b[i] + c[i];
```

Si può trasformare il ciclo precedente in un nuovo ciclo con un fattore di unrolling detto $UNROLL$ e il codice diventa:

```
for(int i = 0; i < n; i+ = UNROLL)
    a[i] = b[i] + c[i];
    a[i + 1] = b[i + 1] + c[i + 1];
    ...
    a[i + UNROLL − 1] = b[i + UNROLL − 1] + c[i + UNROLL − 1];
```

Il loop unrolling serve per srotolare il ciclo, replicando istruzioni non correlate tra loro. Questa scomposizione porta uno sfruttamento del parallelismo implicito e vengono ridotte le istruzioni di gestione del ciclo.

Se le istruzioni d'aggiornamento del ciclo costano 30% (come nel primo ciclo), avendo un fattore di $UNROLL$ pari a 5, si ottiene un risparmio del $\frac{5(1-0.3)+0.3}{5} = 0.76$, ossia introno al 25% rispetto al 30% precedente.

Ponendo ora $UNROLL = p$ la sequenza di istruzioni può essere vettorizzata, trasformando il nuovamente il codice in:

```
for(int i = 0; i < n; i+ = p)
    a[i ... i + p − 1] = b[i ... i + p − 1] + c[i ... i + p − 1];
```

Infine applicando l'unrolling si ottiene la forma più generale possibile contenente sia la vettorizzazione chel'unrolling:

```
for(int i = 0; i < n; i+ = p * UNROLL)
    for(int u = 0; u < UNROLL; u + +)
        a[i + u * p ... i + (u − 1) * p − 1] = b[i + u * p ... i + (u − 1) * p − 1] + c[i + u * p ... i + (u − 1) * p − 1];
```

10.24 Advanced Vector Extensions

L'**Advanced Vector Extensions** (AVX) è un estensione a 256 bit di SSE. Si hanno a disposizione 8 registri a 256 bit chiamati YMM.

Si estendono i vecchi registri XMM, che continuano però ad essere disponibili essendo la parte bassa dei registri YMM, questo è vero se si lavora a 32 bit. Operando a 64 bit i registri raddoppiano in numero, diventando 16. Tutte le vecchie istruzioni di SSE sono ancora disponibili, ma sono applicabili ai soli registri XMM. Se per

256 bit	
128 bit	**128 bit**
YMM0	XMM0
YMM1	XMM1
YMM2	XMM2
YMM3	XMM3
YMM4	XMM4
YMM5	XMM5
YMM6	XMM6
YMM7	XMM7

Tabella 10.1: Registri a disposizione nella versione a 32 bit.

esempio si considera l'istruzione:

ADDPS XMM0, XMM1

la somma verrà applicata solo alla parte bassa del registro, mentre la parte alta non verrà alterata. Per poter utilizzare anche la parte alta, bisogna aggiungere il prefisso V alle vecchie istruzioni. L'istruzione precedente, volendo utilizzare i registri YMM diventa quindi:

VADDPS YMM0, YMM1

Questa applica una somma all'intero vettore, ma siccome il registro è a 256 bit, utilizzando PS si divide in otto campi da 32 bit e quindi otto elaborazioni in una singola istruzione. Se si utilizza invece l'istruzione

VADDPS XMM0, XMM1

si esegue la somma solo sulla parte bassa, ma vengono azzerati i 128 bit della parte alta nella destinazione, e nell'esempio quindi la parte alta di YMM0.

È buona norma utilizzare il prefisso V anche se si utilizzano i registri XMM, perché si andrebbe a pagare una penalità alternando operazioni a 128 bit con operazioni a 256 bit. Per quanto riguarda le operazioni scalari, senza il prefisso V esse operano sui 128 bit della parte bassa, utilizzando la V non si può specificare YMM, perché lavorano comunque sulla parte bassa ma senza azzerare la parte alta della destinazione.

Tutte le istruzioni che contengono il prefisso V, ammettono la semantica a tre operandi, quindi è possibile scrivere:

VADDPS YMM0, YMM1, YMM2

in cui si ha una destinazione, YMM0, e due sorgenti, YMM1 e YMM2, quindi coincide con l'operazione $YMM0 = YMM1 + YMM2$. È anche possibile utilizzare le operazioni a tre operandi sui registri XMM, e ovviamente le versioni a due operandi restano disponibili e non cambiano forma.

La forma più generale delle istruzioni a 3 operandi è:

V<OP> <Destinazione>, <Sorgente$_1$>, <Sorgente$_2$>

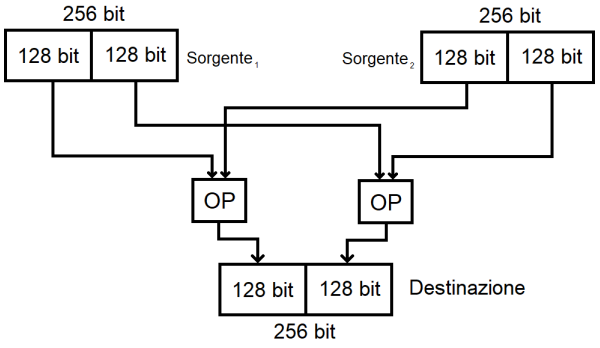


Figura 10.11: Schema istruzioni a tre operandi.

In alcune istruzioni però, l'estensione a 256 bit non è naturale, come ad esempio avviene per l'istruzione ADD.

Esiste inoltre un nuovo tipo a 128 bit chiamato **F128**, che viene utilizzato negli spostamenti ed indica un blocco da 128 bit. Un esempio di istruzione di spostamento a 128 bit è **VPERM2F128**, un istruzione a quattro

operandi che ha la forma:

VPERM2F128 <Destinazione>, <Sorgente$_1$>, <Sorgente$_2$>, <Immediato>

utilizzata per spostare o azzerare determinati campi a 128 bit.

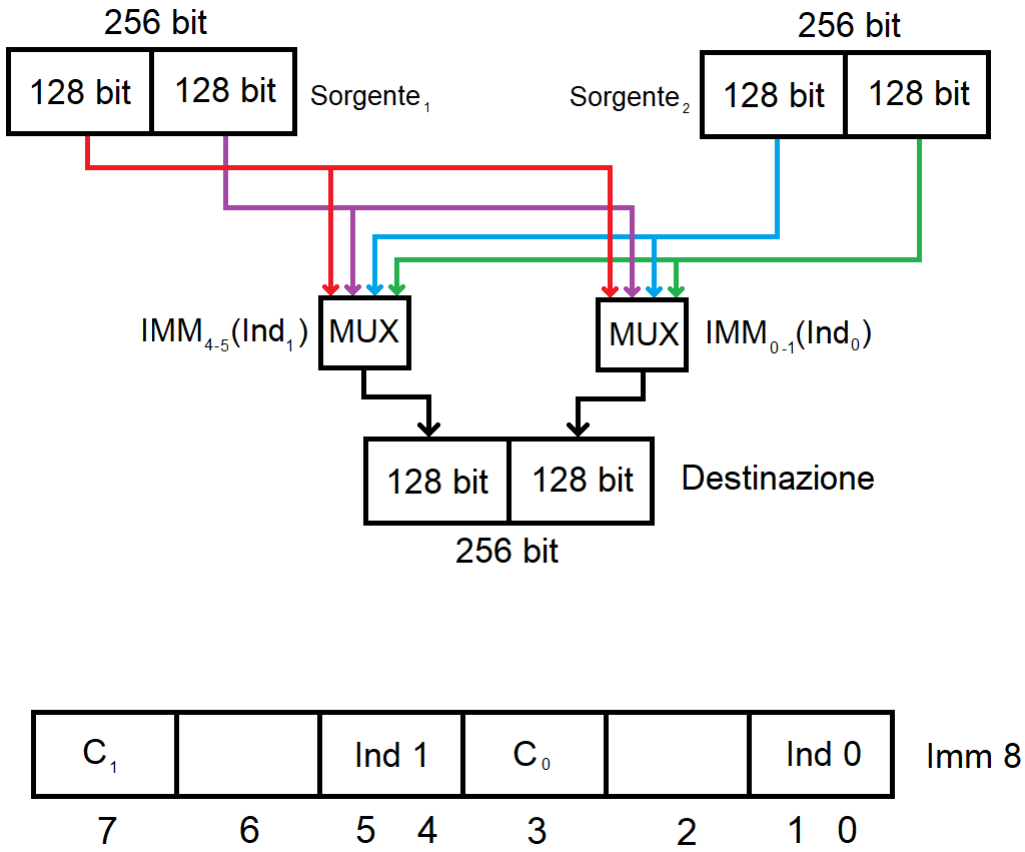


Figura 10.12: Schema istruzione VPERM2F128.

L'immediato indica cosa copiare nella parte bassa tramite i bit 1 e 0, mentre indica cosa copiare nella parte alta tramite i bit 5 e 4. Dopo l'assegnamento, se il bit 7 dell'immediato è a 1, allora si azzera la parte alta della destinazione, analogamente se il bit 3 è a 1 si azzera la parte bassa della destinazione. Ad esempio, volendo copiare la parte alta di YMM1 in YMM0 e azzerare la parte alta, il codice sarà:

VPERM2F128 YMM0, YMM0, YMM1, 10000011b

10.25 VHADDPS

È la versione AVX dell'istruzione HADD, somma coppie consecutive di dati conservando però la retrocompatibilità.