

Architetture degli elaboratori II

Index

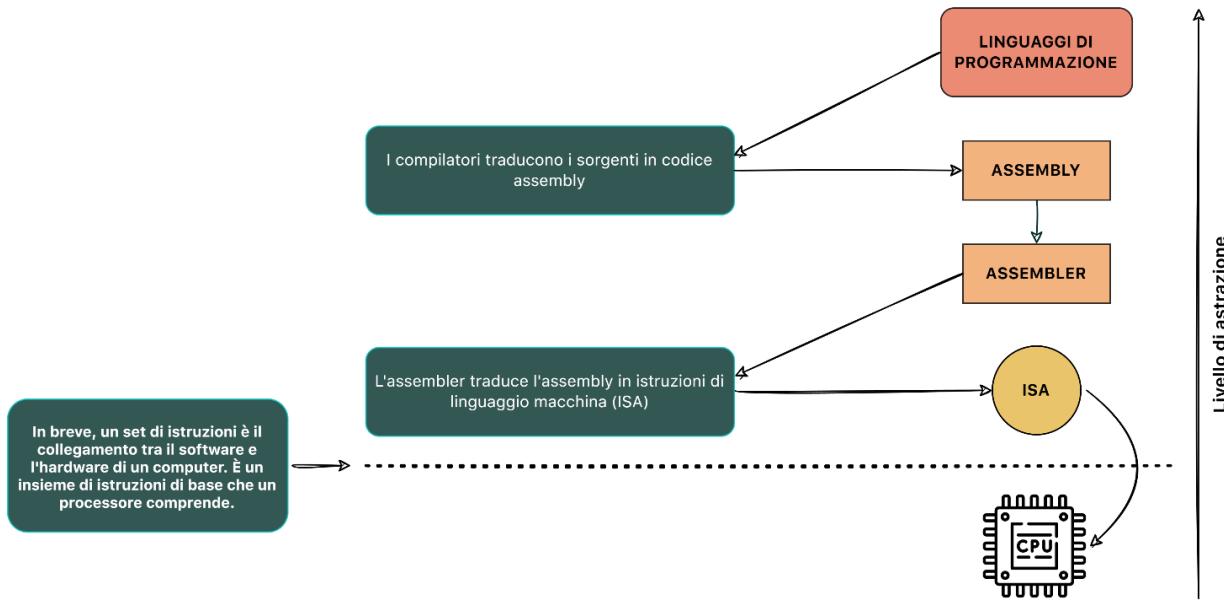
- Concetti Fondamentali
 - Instruction set architecture (ISA)
 - Microarchitecture
 - Restricted instruction set architecture & complex instruction set architecture
 - Processori Subscalari, scalari e superscalari
 - Monocycle Processor & Multicycle processor
 - Elementi di stato ed elementi combinatori
 - Clock per instruction (CPI) & Instruction per clock (IPC)
- MIPS Instruction Set
 - R instruction
 - I instruction
 - Jump and branch instruction
- MIPS Microarchitecture
 - Una versione Monociclo
 - MIPS Registers
 - Accesso ai registry
 - Control Unit in MIPS Monocycle
 - ALUOp & ALU Control
 - Transizione al Multiciclo
 - Registri di pipelining
 - Comromessi nel multiciclo
 - Control Unit in Multicycle
 - MIPS - Instruction Fetch Phase (IF)
 - MIPS - Instruction decode Phase (ID)
 - MIPS – Execute Phase (EX)
 - MIPS – Memory Access Phase (MEM)
 - MIPS – Write Back Phase (WB)
- Instruction level Parallelism
- Hazards and dependencies
 - Data Dependences
 - Name dependences – Anti-dependence
 - Name dependences – Output dependence
 - Control depependences
 - Structural Hazard
 - Stall della pipeline
- Dynamic instruction level parallelism
 - Scheduling dinamico & algoritmo di Tomasulo
 - Issue Phase
 - Struttura delle reservation station
 - Execute Phase
 - Features dell'algoritmo di Tomasulo
 - Dynamic Branch Prediction
 - Speculative execution

- Reorder buffer
 - Issue Phase
 - Execute Phase
 - Write-back Phase
 - Commit Phase
- Multiple Issue
 - Superscalar Processor
 - Increasing instruction fetch bandwidth
 - VLIW Processor
 - ILP statico
 - Loop Unrolling
- Cache Memory
 - SRAM & DRAM
 - Cache Hierarchy
 - Principio di Località
 - Fully Associative Cache
 - Replacement Policy
 - Vantaggi e svantaggi
 - Direct Mapped cache
 - Vantaggi e svantaggi
 - Set-Associative cache
 - Gestione delle scritture
 - Miss Penalty
 - Influenza della cache sugli algoritmi
- Multithreading & TLP
 - Hardware & Software Multithreading
 - Coarse-Grain Multithreading
 - Block Multithreading in out of order cores
 - Fine-Grained Multithreading
 - Simultaneous Multithreading
- Multiprocessing
 - Cache Coherence
 - Cache Coherence protocols
 - Snooping protocols
 - MSI Protocol
 - MESI Protocol
 - MOESI & MESIF
 - Limitazioni degli SMP
 - Crossbar Switch
 - Multistage Switching Network
 - Directory-Based Protocol
 - Sincronizzazione tra processi
 - Consistenza della memoria
 - Gestione dei processi negli SMP

Concetti Fondamentali

Instruction set architecture (ISA)

In informatica, un'instruction set architecture (ISA) è un modello astratto di computer. Un dispositivo che esegue le istruzioni descritte da quell'Instruction set, come una CPU, è chiamato implementazione.



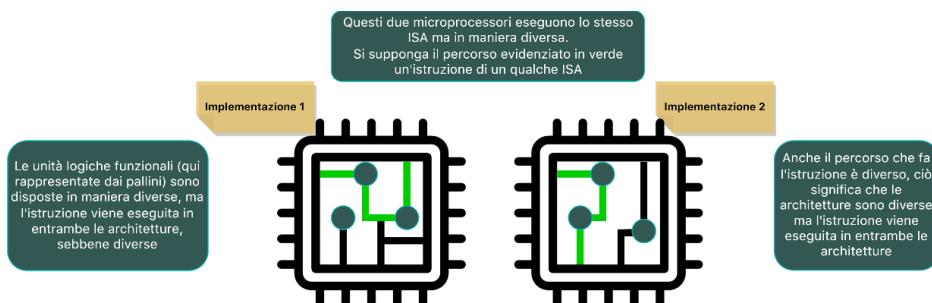
È definito come un modello astratto di computer in quanto l'Instruction set definisce, in sostanza, come lavora quella macchina: definisce i tipi dei dati, la quantità dei registri e come viene indirizzata la memoria. Inoltre, descrive l'insieme di istruzioni macchina visibili a basso livello al programmatore e agisce come interfaccia tra l'hardware e il software, specificando sia le capacità del processore sia come vengono eseguite le operazioni

Microarchitettura

La microarchitettura invece rappresenta il progetto a livello logico di unità funzionali di un microprocessore. Di solito viene mostrata tramite una progettazione a blocchi in modo da indicare le dipendenze tra le varie unità funzionali (definisce l'organizzazione di un particolare processore).

Per microarchitettura ci si riferisce a com'è definita l'architettura dell'implementazione (quindi della CPU) in termini delle tecnologie utilizzate, le risorse ed i metodi per eseguire uno specifico set di istruzioni (ISA).

Quindi è possibile avere due microarchitetture diverse ma che eseguono lo stesso Instruction set.



Possiamo avere due processori differenti basati sullo stesso Instruction set ma con differenti microarchitetture e differenti prestazioni, dimensioni fisiche e costi monetari diversi.

Pertanto, una macchina con prestazioni inferiori può essere sostituita con una macchina con prestazioni superiori senza dover sostituire il software (con stesso ISA). Inoltre, consente la

progressione delle microarchitetture delle implementazioni di tale ISA, in modo che un'implementazione più recente e con prestazioni più elevate di un ISA possa utilizzare software che gira su generazioni precedenti di implementazioni.

Un esempio reale di differenti microarchitetture ma che eseguono lo stesso ISA sono Intel e AMD. Ci sono alcune istruzioni speciali (opzionali) offerte da AMD e Intel che non sono comuni tra i due, che i programmi sanno come riconoscere e utilizzare quando necessario. Ma per il 90%, il loro set di istruzioni è lo stesso.

Restricted Instruction set architecture & complex instruction set architecture (RISC vs CISC)

Un ISA può essere classificato in diversi modi. Una classificazione comune è per complessità architetturale: **ISA CISC** o **ISA RISC**.

CISC (Complex instruction set computer) fu il primo concetto di instruction set (ma ancora in uso). L'architettura CISC utilizza un set di istruzioni complesse, che richiedono più cicli di clock per essere eseguite, ma che permettono di eseguire operazioni complesse in un'unica istruzione. Ciò significa che ad ogni istruzione si eseguono più operazioni (fetch, evaluating, storing) riducendo drasticamente la dimensione dei programmi. Tuttavia, ciò aumenta il numero di cicli di clock per istruzione (CPI) e rende l'hardware più complesso.

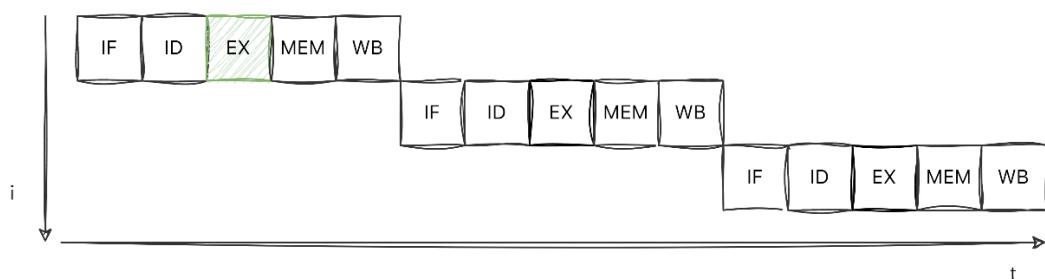
RISC (Reduced instruction set computer) fu il secondo concetto di instruction set ideato durante il corso della storia. Questo cerca di semplificare la complessità dell'hardware riducendo il numero di istruzioni totali e rendendoli di lunghezza fissa. Inoltre, ogni istruzione, in un'architettura RISC, è eseguita in un singolo ciclo di clock (a meno di stalli, che vedremo).

L'eleganza e la relativa semplicità delle architetture RISC (rispetto alle architetture che hanno pian piano sostituito, le CISC appunto) ha permesso di implementare o sfruttare meglio alcune tecniche fondamentali per aumentare la velocità di esecuzione delle istruzioni macchina di cui sono composti i programmi. Queste tecniche sono comuni a tutte le architetture moderne, anche se non tutti i processori le implementano tutte insieme:

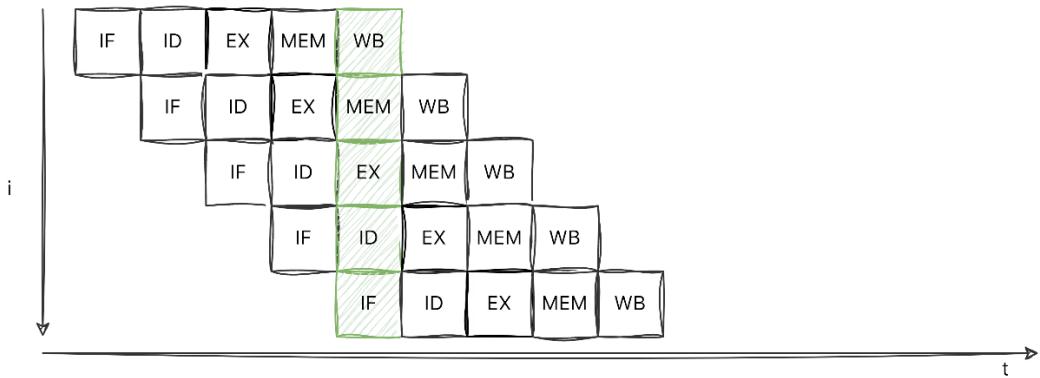
- **Pipelining.** L'esecuzione delle istruzioni avviene in catena di montaggio: si inizia ad eseguire una nuova istruzione prima di aver terminato l'esecuzione della istruzione precedente.
- **Esecuzione in parallelo di istruzioni.** Più istruzioni fra loro indipendenti vengono letteralmente eseguite in parallelo.
- **Branch prediction dinamico:** Il risultato di un salto condizionato viene ipotizzato in anticipo per sapere prima dove prosegue il programma in esecuzione.
- **Speculazione hardware.** Alcune istruzioni vengono eseguite prima di sapere se dovessero esserlo veramente, ed eventualmente poi l'effetto della loro esecuzione viene annullato.

Processori subscalari, scalari e superscalari

Le prime generazioni di processori erano interamente sequenziali. Ogni istruzione ricevuta dal processore doveva essere completata per intero prima che potesse essere avviata quella successiva. Questi processori prendono il nome di **processori subscalari**. Ci sono cinque fasi per la maggior parte delle istruzioni: Instruction fetch, Instruction decode, Execute, Memory access e Writeback.

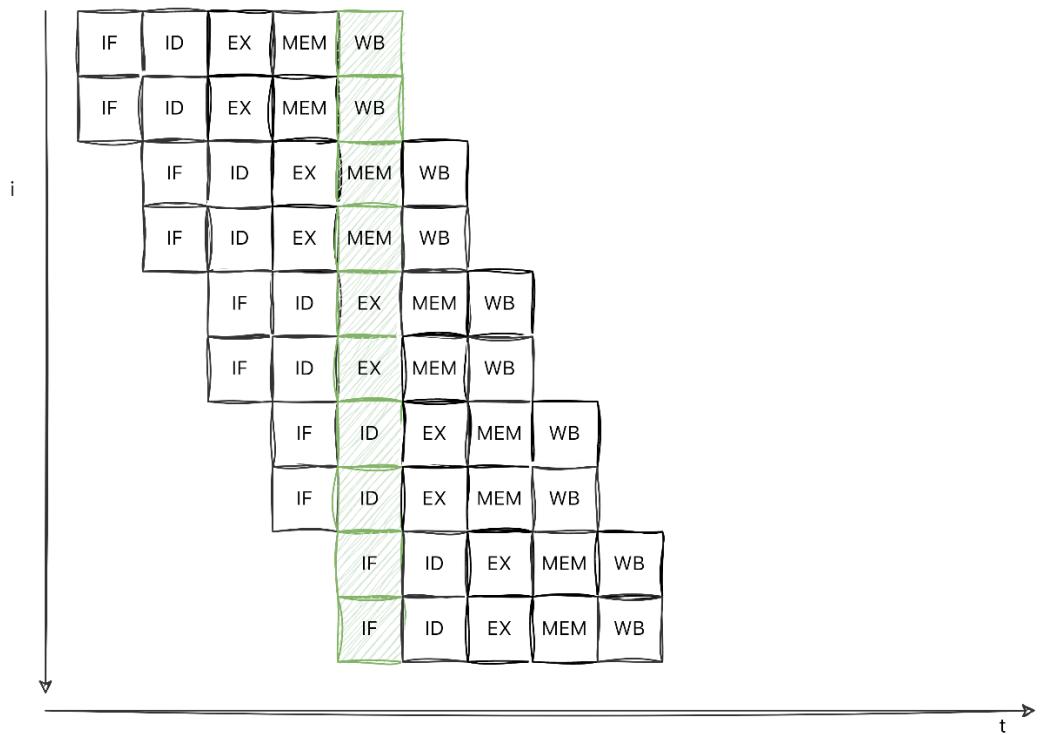


Un **processore scalare** può essere ottenuto partendo da un processore subscalare ed applicando una pipeline al sistema. Un processore scalare è classificato come processore SISD (Single Instruction, Single Data) nella tassonomia di Flynn.



In un **processore pipelined scalare**, si elabora un'istruzione alla volta su un singolo pezzo di dati per ciclo di clock. Ciò consente un throughput massimo di un'istruzione completata per ciclo.

Un **processore superscalare** è un tipo di processore in grado di eseguire più di una istruzione per ciclo di clock. Questo grazie al multiple issue che inietta nel processore più di una istruzione per ciclo di clock. In questo modo due istruzioni possono trovarsi in ogni fase della pipeline in ogni ciclo. Ciò si traduce ovviamente in una maggiore complessità del design poiché l'hardware viene duplicato, tuttavia offre eccellenti prestazioni.



Dato che si è voluto sempre raggiungere potenze di calcolo più elevate, e dato che non è possibile continuare a raddoppiare (ma nemmeno incrementare linearmente ogni anno) la velocità del clock, si è optati per sfruttare al meglio il parallelismo delle istruzioni di un programma. Quindi i processori moderni si sono evoluti in modo che le CPU possano utilizzare una varietà di unità di esecuzione in parallelo, diminuendo sempre di più la dimensione delle unità funzionali, aumentando le cache ai vari livelli, ed ottimizzando la performance per watt.

Monocycle Processor & Multicycle processor

La differenza tra un'architettura monociclo e una mult ciclo riguarda il modo in cui viene eseguita un'istruzione all'interno del processore.

In un'architettura **monociclo**, un'istruzione viene eseguita in un singolo ciclo di clock, che comprende tutte le fasi del ciclo di vita dell'istruzione. Questo ricorda il processore subscalare, infatti un processore monociclo è utilizzato in un'architettura subscalare, poiché può eseguire solo un'istruzione alla volta.

Un processore monociclo è solitamente più lento di un processore a mult ciclo, poiché ogni istruzione richiede più tempo per essere eseguita.

Inoltre, nel processore monociclo, ogni istruzione viene eseguita in un singolo ciclo di clock; quindi, il ciclo deve essere abbastanza lungo da permettere all'istruzione di completare tutte le sue operazioni necessarie nel datapath. Se il ciclo fosse troppo breve, l'istruzione potrebbe non avere il tempo di completare tutte le sue operazioni e dovrà attendere il prossimo ciclo per continuare l'elaborazione. Ciò può rallentare notevolmente il processore. Di conseguenza, il ciclo di clock di un processore monociclo viene solitamente progettato in base all'istruzione che richiede più tempo per essere eseguita. Ciò significa che il ciclo di clock dovrà quindi essere abbastanza lungo da permettere all'istruzione più lenta di completare tutte le sue operazioni, anche se questo significa che alcune istruzioni più veloci possono essere eseguite in modo inefficiente.

In un'architettura **mult ciclo**, invece, un'istruzione viene eseguita in più cicli di clock, ognuno dei quali corrisponde a una fase del ciclo di vita dell'istruzione. Se il processore mult ciclo implementa anche la pipeline allora può essere considerato un processore scalare.

L'architettura mult ciclo è più complessa rispetto all'architettura monociclo, ma consente di eseguire istruzioni più complesse e di utilizzare risorse hardware in modo più efficiente.

Sebbene le macchine a ciclo singolo possano funzionare correttamente, non vengono usati nelle implementazioni moderne, poiché sono inefficienti. Infatti, le diverse istruzioni di un processore hanno tempi di esecuzione diversi, a seconda della quantità di lavoro necessario all'interno del datapath (ossia a seconda della porzione di datapath da attraversare).

Per trasformare la macchina monociclo in una macchina mult ciclo, il progettista scomponerà l'esecuzione di ciascuna istruzione in un insieme di passi, ognuno dei quali sia eseguibile in un singolo ciclo di clock. È opportuno quindi che, ogni passo, richieda più o meno lo stesso tempo di esecuzione. Infatti, se un passo è più lungo degli altri, la durata del clock dovrà necessariamente essere sufficiente all'esecuzione del passo più lungo. A questo punto, il ciclo di clock può essere ridimensionato sulla base del tempo necessario ad attraversare la singola fase del datapath, e non più il datapath completo: se il datapath è stato suddiviso in N fasi, il nuovo ciclo di clock potrà essere lungo all'incirca un N-esimo della lunghezza del ciclo di clock della macchina monociclo.

Da monociclo a mult ciclo è necessario che ogni fase termini con il salvataggio dell'informazione elaborata in registri "nascosti" del datapath. Questi registri nascosti, invisibili al livello ISA, servono per implementare il funzionamento dell'architettura mult ciclo: i registri di output di ogni fase costituiscono l'input della fase successiva.

I registri nascosti sono quindi utilizzati come buffer per memorizzare temporaneamente i dati elaborati in ogni fase dell'elaborazione dell'istruzione, in modo che possano essere trasmessi alla fase successiva senza perdita di informazioni. Ciò consente di eseguire l'elaborazione dell'istruzione in modo efficiente e preciso, poiché i dati elaborati in ogni fase sono conservati in modo sicuro prima di essere utilizzati nella fase successiva.

Inoltre, i registri nascosti consentono una maggiore flessibilità nella progettazione del processore, poiché consentono di suddividere l'elaborazione dell'istruzione in fasi più piccole e gestibili. Ciò consente di progettare processori più complessi e potenti che possono elaborare una vasta gamma di istruzioni in modo efficiente.

Il fatto di utilizzare un processore multiciclo rispetto al monociclo, quindi, è vantaggioso non solo perché non utilizziamo un ciclo di clock per ogni istruzione (che quindi andrebbe a sprecare risorse per quelle istruzioni che richiedono meno tempo), ma anche perché la suddivisione in fasi, permette l'implementazione del Pipelining, e quindi della sovrapposizione delle istruzioni nel datapath, ma sempre in sequenza (ad ogni ciclo di clock, tutte le fasi sono impegnate ad eseguire lavoro su un'istruzione diversa).

La tecnica del pipelining è una delle tante che vengono utilizzate per permettere un parallelismo a livello di istruzione (in particolare quella del pipeline è una forma di ILP dinamico, che vedremo).

Elementi di stato ed elementi combinatori

In un circuito elettronico, gli elementi di stato sono componenti che memorizzano informazioni sullo stato del circuito.

Un esempio di elemento di stato potrebbe essere un flip-flop: può essere considerato come un elemento di stato, in quanto è in grado di memorizzare un bit di informazione e di mantenerlo per un certo periodo di tempo. Un flip-flop può essere utilizzato per sincronizzare i segnali in un circuito o per implementare un contatore, ma non modifica la forma d'onda o il contenuto informativo dei segnali che gli vengono forniti in ingresso.

Il combinatorio, d'altra parte, si riferisce al modo in cui gli elementi del circuito sono connessi tra loro. Ad esempio, in un circuito combinatorio, il valore di uscita di un componente dipende solo dai valori di ingresso attuali e non dallo stato passato del circuito. In sostanza elaborano e modificano i segnali elettrici in base alle loro regole di funzionamento (porte logiche, resistenze, etc...). Gli elementi combinatori sono spesso utilizzati nei contesti in cui è richiesta una risposta rapida e affidabile, come ad esempio nei circuiti di calcolo (ALU) o di controllo (Multiplexer).

Un multiplexer, invece, può essere considerato come un elemento combinatorio, in quanto elabora e modifica i segnali di ingresso selezionandone uno e facendolo passare attraverso l'uscita. Un multiplexer non immagazzina energia, ma modifica il contenuto informativo dei segnali elettrici in base alla sua configurazione e al segnale di ingresso selezionato (ad esempio con porte logiche).

In sintesi, gli elementi di stato immagazzinano energia mentre gli elementi combinatori elaborano e modificano i segnali elettrici.

Clock per instruction (CPI) & Instruction per clock (IPC)

CPI e IPC sono due aspetti per misurare le prestazioni di un processore.

Il **Clock per instruction** (o cycles per instruction) è il numero medio di cicli di clock che il processore impiega per completare un programma (o un frammento di esso).

Se assumiamo la seguente suddivisione del datapath:

- Instruction fetch (IF).
- Instruction decode/Register fetch (ID).
- Execution/Effective address (EX).
- Memory access (MEM).
- Write-back cycle (WB).

Per un processore subscalare, una nuova istruzione viene recuperata nella fase 1 solo dopo che l'istruzione precedente termina nella fase 5, quindi il numero di cicli di clock necessari per eseguire un'istruzione è cinque (CPI = 5).

Per un processore scalare invece, una nuova istruzione viene recuperata a ogni ciclo di clock sfruttando il parallelismo a livello di istruzione, quindi, teoricamente si potrebbero avere cinque istruzioni contemporaneamente nelle cinque fasi della pipeline (un'istruzione per fase), un'istruzione

diversa completerebbe la fase 5 in ogni ciclo di clock e in media il numero di cicli di clock necessari per eseguire un'istruzione è 1 (CPI = 1).

Tuttavia, con un processore superscalare, è possibile ottenere valori CPI ancora migliori (CPI < 1): Ad esempio, con due unità di esecuzione, due nuove istruzioni vengono recuperate ogni ciclo di clock sfruttando il parallelismo a livello di istruzione; quindi, due istruzioni diverse completerebbero lo stadio 5 in ogni ciclo di clock e in media il numero di cicli di clock necessari per eseguire un'istruzione è 1/2 (CPI = 1/2 < 1).

Per calcolare la media è necessario conoscere, per ogni categoria di istruzione, il numero di cicli necessari per eseguirla. La formula è definita come segue:

$$CPI = \frac{\sum_i (IC_i)(CC_i)}{IC}$$

Dove IC_i è il numero di istruzioni per un dato tipo di istruzione, CC_i sono i cicli di clock per quel tipo di istruzione ed infine $IC = \sum_i (IC_i)$ è il conteggio totale delle istruzioni.

Ipotizzando per una MIPS multiciclo (quindi senza pipeline), 5 tipi di istruzioni:

- Load (5 cycles)
- Store (4 cycles)
- R-type (4 cycles)
- Branch (3 cycles)
- Jump (3 cycles)

Ipotizzando un programma che ha

- 50% load
- 25% store
- 15% R-type
- 8% branch
- 2% jump

Il CPI medio per quel programma (su quella microarchitettura) è:

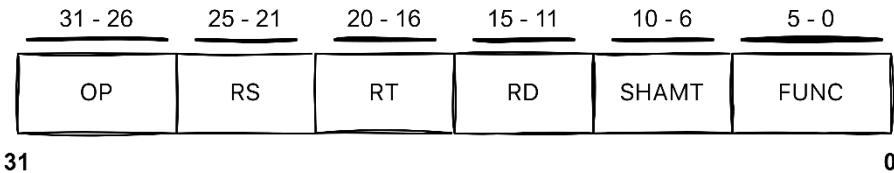
$$CPI = \frac{5 \times 50 + 4 \times 25 + 4 \times 15 + 3 \times 8 + 3 \times 2}{100} = 4.4$$

L'IPC (instruction per clock, instruction per cycle), invece, è il numero di istruzioni eseguite per ciclo di clock. Per calcolare l'IPC, basta invertire il valore del CPI.

Mips Instruction Set

Le architetture MIPS fino alla versione V (quindi non 64 bit) utilizzavano WORD di 32bit;

R instruction



- **OP** (opcode): campo a 6 bit che specifica l'operazione da eseguire.
- **RS** (source register 1): campo a 5 bit che specifica il primo registro sorgente dell'operazione.
- **RT** (source register 2): campo a 5 bit che specifica il secondo registro sorgente dell'operazione.
- **RD** (destination register): campo a 5 bit che specifica il registro di destinazione dell'operazione.
- **SHAMT** (shift amount): campo a 5 bit che specifica il numero di bit da spostare in un'operazione di shift.
- **FUNC** (function code): campo a 6 bit che specifica la variante dell'operazione op richiesta.

Le istruzioni di tipo R sono un tipo di istruzioni MIPS che vengono utilizzate per eseguire operazioni aritmetiche tra registri e di confronto

Ecco alcuni esempi di istruzioni di tipo R:

- **add \$t0, \$t1, \$t2**: esegue l'addizione dei valori contenuti nei registri \$t1 e \$t2 e memorizza il risultato nel registro \$t0.
- **sub \$t0, \$t1, \$t2**: esegue la sottrazione del valore contenuto nel registro \$t2 dal valore contenuto nel registro \$t1 e memorizza il risultato nel registro \$t0.
- **slt \$t0, \$t1, \$t2**: confronta i valori contenuti nei registri \$t1 e \$t2 e memorizza 1 nel registro \$t0 se il valore contenuto nel registro \$t1 è minore di quello contenuto nel registro \$t2, altrimenti memorizza 0.

Spesso si utilizza il prefisso 'D' o 'F' per indicare che l'istruzione di tipo R lavora su Interi o Float: Dadd, Dsub, Fadd, etc...

Quindi:

- OPCode = Operazione aritmetica
- Func = Somma, sottrazione, etc...

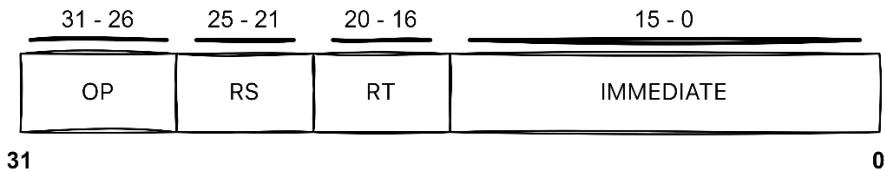
I valori dei campi op e funct saranno usati dalla control unit per pilotare adeguatamente l'esecuzione dell'istruzione all'interno del datapath.

I instructions

Le istruzioni di tipo I sono un tipo di istruzioni MIPS che vengono utilizzate per eseguire operazioni aritmetiche immediate (un registro con una costante), di caricamento, di memorizzazione e di branching. Quindi anche le load e le store sono di tipo I.

Tra le istruzioni di tipo I quindi, troviamo ad esempio le istruzioni Immediate: sono un tipo di istruzioni che vengono utilizzate per eseguire operazioni aritmetiche su numeri interi. Sono chiamate "istruzioni immediate" perché utilizzano il campo "immediate" dell'istruzione per fornire un valore intero che viene utilizzato come operando dell'istruzione (non utilizzano il campo Func, anzi, viene sostituito completamente da "immediate").

Le istruzioni immediate, quindi, risultano avere una struttura diversa rispetto ad altri tipi di istruzioni di questo instruction set. Sono sempre istruzioni a 32bit ma così formate:



Il campo immediate in sostanza sostituisce i campi RD, SHAMT e FUNC.

Per scrivere e leggere dalla memoria si utilizzano le istruzioni (sempre di tipo I), "LW" e "SW" per leggere e scrivere WORD (quindi 32 bit). Per scrivere e leggere 16bit invece, si utilizza "LH" ed "SH", mentre per leggere e scrivere Byte utilizziamo "LB" ed "SB".

Esempio: lw \$t, offset(\$s)

Dove:

- **\$t** è il registro dove verrà salvato il valore letto dalla memoria
- **offset** è l'offset (cioè il numero di byte) rispetto all'indirizzo contenuto nel registro **\$s** da cui il valore deve essere letto
- **\$s** è il registro che contiene l'indirizzo di memoria da cui il valore deve essere letto

Nota: le istruzioni sw e lw possono essere utilizzate solo con indirizzi di memoria allineati a 4 byte. Ciò significa che l'indirizzo di memoria deve essere divisibile per 4. Se l'indirizzo di memoria non è allineato a 4 byte, verrà generata un'eccezione di errore di allineamento.

Esempio di errore di allineamento:

- ```
1. li $s0, 0x10010001 # carico l'indirizzo 0x10010001 in $s0 (non allineato a 4 byte)
2. lw $t0, 0($s0) # tentativo di caricare il valore contenuto nella posizione di memoria indicata da $s0 in $t0
```

In questo esempio, l'indirizzo di memoria 0x10010001 non è divisibile per 4, quindi verrà generata un'eccezione di errore di allineamento.

Ora, se consideriamo l'indirizzo di memoria 0x10010001, notiamo che l'ultimo carattere esadecimale (che rappresenta l'ultimo gruppo di 4 bit) è 1. In binario, 1 rappresenta 0001. Se guardiamo l'ultimo carattere di altri indirizzi di memoria allineati a 4 byte (ad esempio 0x10010004), notiamo che l'ultimo carattere è 4, che in binario rappresenta 0100. Quindi, se l'ultimo carattere di un indirizzo di memoria è 0, 4, 8 o C (che rappresentano rispettivamente i numeri 0, 4, 8 e 12 in formato esadecimale), l'indirizzo di memoria è allineato a 4 byte. Altrimenti, l'indirizzo di memoria non è allineato a 4 byte.

In sintesi, per capire se un indirizzo di memoria è allineato a 4 byte in MIPS, basta controllare che l'ultimo carattere esadecimale sia 0, 4, 8 o C. Ad esempio, gli indirizzi di memoria 0x10010004, 0x20020000 e 0x30030000 sono tutti allineati a 4 byte, mentre gli indirizzi di memoria 0x10010001, 0x20020002 e 0x30030003 non lo sono.

## Jump and branch instructions

Le istruzioni di salto vengono utilizzate per modificare il flusso di esecuzione del programma, saltando a una determinata posizione del codice in base a un indirizzo specifico (non si controlla nessuna condizione, quelli sono i branch).

Le istruzioni di tipo J sono così formate:



Nella MIPS, ci sono diverse istruzioni di salto, come ad esempio:

- j: salta all'indirizzo specificato.
- jr: salta all'indirizzo specificato nel registro specificato.
  - L'istruzione jr (jump register) in MIPS viene utilizzata per effettuare un salto incondizionato ad un'altra istruzione del programma, utilizzando il valore di un registro come destinazione del salto.
- jal: salta all'indirizzo specificato e salva il valore del registro PC+4 nel registro \$ra.
  - L'istruzione jal (jump and link) in MIPS è simile all'istruzione jr, ma oltre a effettuare un salto incondizionato ad un'altra istruzione del programma, salva anche l'indirizzo dell'istruzione successiva in un registro specificato, in modo da potervi fare ritorno in seguito.

Esempio di jal:

```

1. main:
2. # istruzioni del programma principale
3. jal subroutine # salto incondizionato alla sottofunzione e salvataggio dell'indirizzo dell'istruzione
successiva nel registro $ra
4. # altre istruzioni ...
5.
6. subroutine:
7. # istruzioni della sottofunzione
8. jr $ra # ritorno all'indirizzo salvato nel registro $ra

```

Le istruzioni di branch, invece, come la beq (salta se i registri sono uguali) e la bne (salta se i registri sono diversi), sono di tipo I.

Esempio di jump:

```
1. j target # salta all'indirizzo specificato
```

Esempio di Branch

```
1. beq $s1, $s2, target # salta all'indirizzo specificato se i registri $s1 e $s2 sono uguali
```

Di seguito la tabella con l'instruction set completo:

| Mnemonic | Meaning      | Type | Opcode | Funct |
|----------|--------------|------|--------|-------|
| add      | Add          | R    | 0x00   | 0x20  |
| addu     | Add Unsigned | R    | 0x00   | 0x21  |
| and      | Bitwise AND  | R    | 0x00   | 0x24  |

|              |                                        |   |      |      |
|--------------|----------------------------------------|---|------|------|
| <b>div</b>   | Divide                                 | R | 0x00 | 0x1A |
| <b>divu</b>  | Unsigned Divide                        | R | 0x00 | 0x1B |
| <b>jalr</b>  | Jump and Link Register                 | R | 0x00 | 0x09 |
| <b>jr</b>    | Jump to Address in Register            | R | 0x00 | 0x08 |
| <b>mfhi</b>  | Move from HI Register                  | R | 0x00 | 0x10 |
| <b>mthi</b>  | Move to HI Register                    | R | 0x00 | 0x11 |
| <b>mflo</b>  | Move from LO Register                  | R | 0x00 | 0x12 |
| <b>mtlo</b>  | Move to LO Register                    | R | 0x00 | 0x13 |
| <b>mfc0</b>  | Move from Coprocessor 0                | R | 0x10 | NA   |
| <b>mult</b>  | Multiply                               | R | 0x00 | 0x18 |
| <b>multu</b> | Unsigned Multiply                      | R | 0x00 | 0x19 |
| <b>nor</b>   | Bitwise NOR (NOT-OR)                   | R | 0x00 | 0x27 |
| <b>xor</b>   | Bitwise XOR (Exclusive-OR)             | R | 0x00 | 0x26 |
| <b>or</b>    | Bitwise OR                             | R | 0x00 | 0x25 |
| <b>slt</b>   | Set to 1 if Less Than                  | R | 0x00 | 0x2A |
| <b>sltu</b>  | Set to 1 if Less Than Unsigned         | R | 0x00 | 0x2B |
| <b>sll</b>   | Logical Shift Left                     | R | 0x00 | 0x00 |
| <b>srl</b>   | Logical Shift Right (0-extended)       | R | 0x00 | 0x02 |
| <b>sra</b>   | Arithmetic Shift Right (sign-extended) | R | 0x00 | 0x03 |

|              |                                      |   |      |      |
|--------------|--------------------------------------|---|------|------|
| <b>sub</b>   | Subtract                             | R | 0x00 | 0x22 |
| <b>subu</b>  | Unsigned Subtract                    | R | 0x00 | 0x23 |
| <b>j</b>     | Jump to Address                      | J | 0x02 | NA   |
| <b>jal</b>   | Jump and Link                        | J | 0x03 | NA   |
| <b>addi</b>  | Add Immediate                        | I | 0x08 | NA   |
| <b>addiu</b> | Add Unsigned Immediate               | I | 0x09 | NA   |
| <b>andi</b>  | Bitwise AND Immediate                | I | 0x0C | NA   |
| <b>beq</b>   | Branch if Equal                      | I | 0x04 | NA   |
| <b>blez</b>  | Branch if Less Than or Equal to Zero | I | 0x06 | NA   |
| <b>bne</b>   | Branch if Not Equal                  | I | 0x05 | NA   |
| <b>bgtz</b>  | Branch on Greater Than Zero          | I | 0x07 | NA   |
| <b>lb</b>    | Load Byte                            | I | 0x20 | NA   |
| <b>lbu</b>   | Load Byte Unsigned                   | I | 0x24 | NA   |
| <b>lhу</b>   | Load Halfword Unsigned               | I | 0x25 | NA   |
| <b>lui</b>   | Load Upper Immediate                 | I | 0x0F | NA   |
| <b>lw</b>    | Load Word                            | I | 0x23 | NA   |
| <b>ori</b>   | Bitwise OR Immediate                 | I | 0x0D | NA   |
| <b>sb</b>    | Store Byte                           | I | 0x28 | NA   |
| <b>sh</b>    | Store Halfword                       | I | 0x29 | NA   |

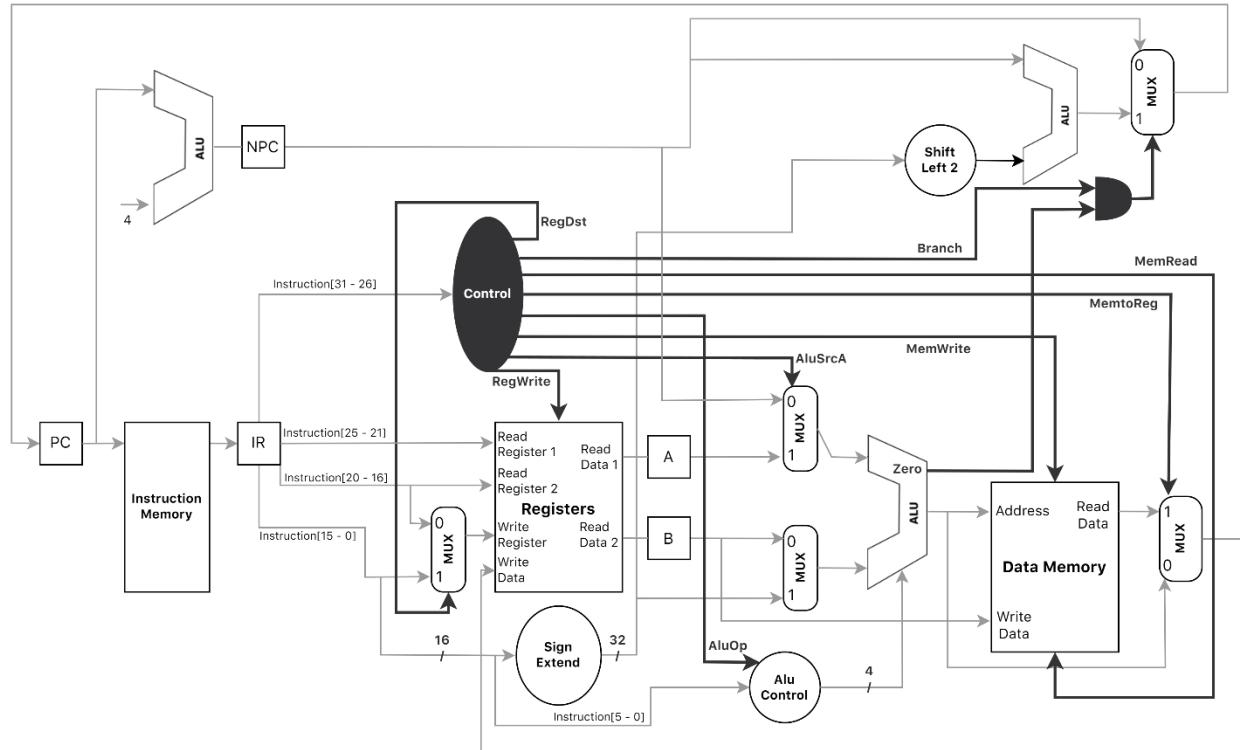
|              |                                          |   |      |    |
|--------------|------------------------------------------|---|------|----|
| <b>slti</b>  | Set to 1 if Less Than Immediate          | I | 0x0A | NA |
| <b>sltiu</b> | Set to 1 if Less Than Unsigned Immediate | I | 0x0B | NA |
| <b>sw</b>    | Store Word                               | I | 0x2B | NA |

# MIPS Microarchitecture

La MIPS è un'architettura che nasce già includendo tante specifiche che vengono utilizzate nelle architetture moderne. Partiamo però semplificando la MIPS, ed introducendola in versione Monociclo per poi trasformarla in Multiciclo ed aggiungendo per grado tutte le funzionalità di una CPU moderna.

## Una versione Monociclo

Questa è l'architettura della MIPS ad alto livello:



Nella versione monociclo, l'esecuzione di ciascuna istruzione avviene in un unico ciclo di clock: è sufficiente che il ciclo sia abbastanza lungo da permettere a ciascuna istruzione di percorrere tutta la parte del datapath necessaria per la sua esecuzione.

I primi due passi di ogni istruzione sono praticamente sempre identici:

1. Usa il PC per prelevare dalla memoria di istruzioni (che, attenzione, non è la RAM) la prossima istruzione da eseguire.
2. Decodifica l'istruzione mentre, contemporaneamente (capiremo meglio fra poco), leggi uno o due registri, usando i campi relativi dell'istruzione per selezionare quale/quali.

Dopo i primi due passi, le azioni richieste dipendono dal tipo di istruzione:

- LOAD/STORE: accesso alla memoria
- Operazione aritmetico / logico
- Branches & Jump

La parte finale delle istruzioni invece si diversifica:

- Load e Store accedono alla memoria dati (che, attenzione, non è la RAM), e nel caso della Load aggiornano un registro
- Le istruzioni logico-aritmetiche aggiornano un registro
- Le istruzioni di salto alterano (solo eventualmente, nel caso dei branch) il valore del PC

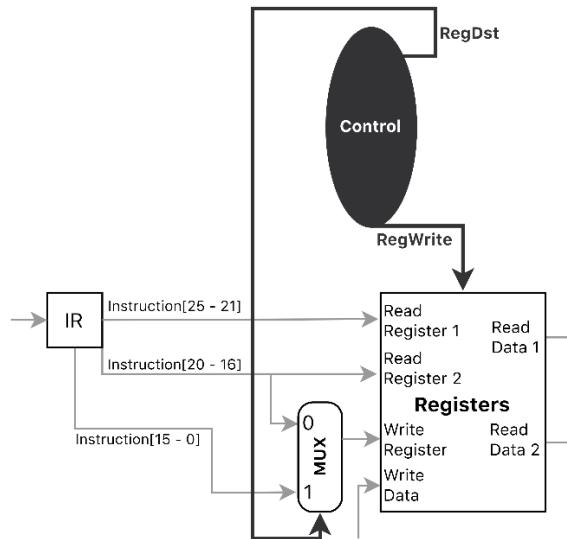
## MIPS Registers

Nei datapath di figura sopra, i registri della CPU sono rappresentati da una unità funzionale (più specificamente un elemento di stato) detta "registers": una unità di memoria, interna alla CPU, molto piccola e molto veloce. La MIPS ha una serie di registri di uso generale che possono essere utilizzati per l'esecuzione di istruzioni e per la memorizzazione di dati.

Ci sono 32 registri di uso generale, noti come \$0 - \$31, e sono identificati con nomi come \$t0, t1, s0, s1, v0, v1, a0, a1, e così via. I registri \$0 - \$31 sono tutti a 32 bit e possono essere utilizzati per immagazzinare numeri interi o puntatori. Il registro \$0 è sempre zero e le scritture su di esso vengono scartate.

Inoltre, la MIPS ha anche altri registri speciali che vengono utilizzati per scopi specifici, come il registro di program counter (PC), il registro di istruzione (IR), il registro di status (SR) e il registro di accumulatore (HI, LO).

### Accesso ai registri



L'accesso in scrittura ai registri interni è pilotato dalla control unit grazie al segnale RegWrite. Questo segnale viene utilizzato per indicare se una determinata istruzione sta scrivendo o meno un valore in un registro di uso generale. Se il segnale RegWrite è impostato su 1, significa che l'istruzione sta scrivendo un valore in un registro di uso generale.

Inoltre, il segnale RegWrite, per effettuare la scrittura, viene utilizzato in combinazione con altri segnali:

- **RegDst:** questo segnale viene utilizzato per specificare il registro in cui il risultato dell'operazione deve essere scritto
  - Se RegDst è impostato su 1, il risultato dell'operazione viene scritto nel registro specificato dall'indirizzo di destinazione dell'istruzione (RD).
  - Se RegDst è impostato su 0, il risultato dell'operazione viene scritto nel registro specificato dall'indirizzo di sorgente dell'istruzione (RT).
- **MemtoReg:** questo segnale viene utilizzato per indicare se il risultato di un'operazione deve essere preso dalla memoria o dal registro di risultato dell'operazione.
  - Se MemToReg è impostato su 1, il risultato dell'operazione viene preso dalla memoria.
  - Se MemToReg è impostato su 0, il risultato dell'operazione viene preso dal registro di risultato dell'operazione.

Ad esempio, con una generica ADD (che ha la seguente sintassi):

**ADD RD, RS, RT**

ecco come verrebbero interpretati i segnali di controllo per questa istruzione:

- **RegWrite**: Impostato a 1 per indicare che il risultato dell'operazione di ADD dovrà essere scritto nel registro di destinazione specificato.
- **RegDst**: Impostato a 1 per indicare che il registro di destinazione dell'istruzione di ADD sarà specificato dal campo rd dell'istruzione. Il valore del campo rd verrà utilizzato come destinazione del risultato dell'operazione di ADD.
- **MemToReg**: Impostato a 0 perché l'istruzione di ADD non richiede che il risultato venga trasmesso dalla memoria al registro di destinazione, ma dalla ALU.

Esempio di MemToReg settato ad 1:

### LW RS, offset(RT)

L'istruzione LW carica una WORD (32bit) di memoria dall'indirizzo di memoria specificato dal contenuto del registro RT (più l'offset) e la scrive nel registro RS.

Ecco come verrebbero interpretati i segnali di controllo per questa istruzione:

- **RegWrite**: Viene impostato su 1 per indicare che il risultato dell'istruzione di LOAD deve essere scritto in un registro. Ciò significa che il valore letto dalla memoria sarà memorizzato in un registro specificato dal campo di destinazione dell'istruzione (RS).
- **RegDst**: viene impostato su 0 perché l'istruzione di LOAD utilizza il campo rs dell'istruzione per specificare il registro di destinazione, e non il campo rd. Quindi il valore del registro rs viene utilizzato come destinazione del risultato dell'istruzione di LOAD.
- **MemToReg**: 1 (il risultato dell'operazione deve essere preso dalla memoria)

Le letture (dal banco dei registri) invece sono "immediate".

## Control Unit in MIPS Monocycle

Come notato, la control unit (unità di controllo) è un componente fondamentale di un processore che ha il compito di gestire e coordinare le operazioni del processore (come abbiamo visto per la scrittura sul register file). La control unit riceve le istruzioni dalla memoria e le decodifica; quindi, genera i segnali di controllo necessari per eseguire l'istruzione.

In particolare, la control unit riceve un frammento dell'istruzione, la analizza per determinare quali operazioni devono essere eseguite e quali dati devono essere utilizzati ed in base a questa analisi, genera i segnali di controllo necessari per eseguire l'istruzione, come il segnale "MemRead" per indicare al modulo di memoria di prepararsi a fornire i dati richiesti o il segnale "RegWrite" per indicare al modulo di registri di immagazzinare i dati.

Ecco una lista dei segnali più utilizzati:

| Segnale di controllo | Scopo                                                                                                   |
|----------------------|---------------------------------------------------------------------------------------------------------|
| MemRead              | Indica al modulo di memoria di prepararsi a fornire i dati richiesti                                    |
| RegDst               | seleziona il registro di destinazione per il risultato dell'operazione                                  |
| MemtoReg             | indica al modulo di registri di utilizzare i dati letti dalla memoria come risultato dell'operazione    |
| RegtoMem             | indica al modulo di memoria di utilizzare i dati immagazzinati in un registro come indirizzo di memoria |
| MemWrite             | Indica al modulo di memoria di immagazzinare i dati forniti dal processore                              |
| RegWrite             | Indica al modulo di registri di immagazzinare i dati forniti dal processore                             |
| PCWrite              | Indica al modulo di program counter di caricare un nuovo indirizzo di istruzione                        |

|         |                                                    |
|---------|----------------------------------------------------|
| ALUSrcA | Seleziona il primo operando per l'unità ALU        |
| ALUSrcB | Seleziona il secondo operando per l'unità ALU      |
| ALUOp   | Seleziona l'operazione da eseguire con l'unità ALU |

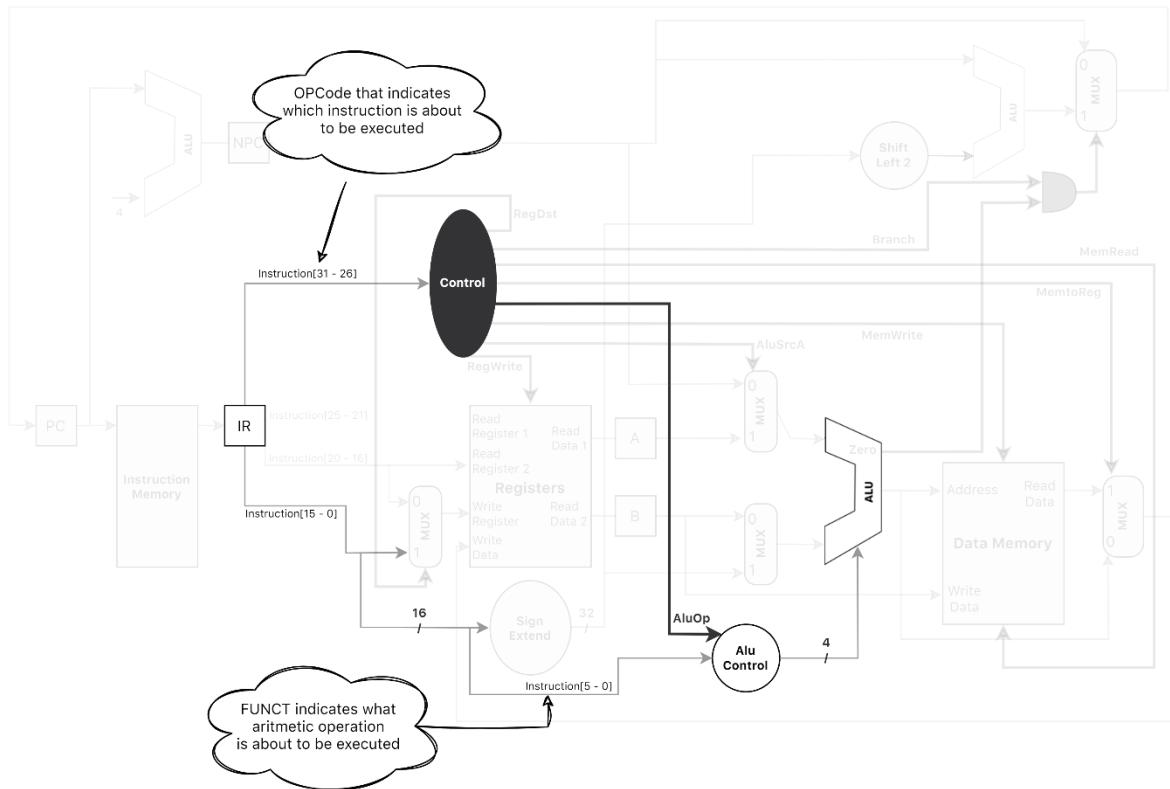
Ovviamente la lista dipende dalla versione dell'architettura.

### ALUOp & ALU Control

il segnale "ALUOp" (Arithmetic Logic Unit Operation) ha lo scopo di specificare il tipo di operazione che l'unità di elaborazione aritmetica e logica (ALU) deve eseguire. Il numero di istruzioni che l'architettura è in grado di eseguire andrà anche a definire il numero di bit dell'ALUOp. Ad esempio, se si esegue solo:

- LOAD/STORE
- BEQ
- R-Type

ALUOp sarà di 2 bit (sufficienti per discriminare 3 casi).



Il segnale di ALUOp non istruisce direttamente la ALU, ma il segnale viene inoltrato all'ALU Control che istruirà a sua volta l'ALU all'operazione corretta. La ALU Control, per istruire la ALU, oltre al segnale ALUOp, riceve in input anche il campo FUNCT dell'istruzione corrente. Questo per discriminare le operazioni aritmetico/logiche da eseguire; le istruzioni R-Type possono essere di qualsiasi tipo: ADD, SUB, etc... quindi solo i due bit di ALUOp che indicano che l'istruzione è una R-Type non sono sufficienti per istruire l'ALU.

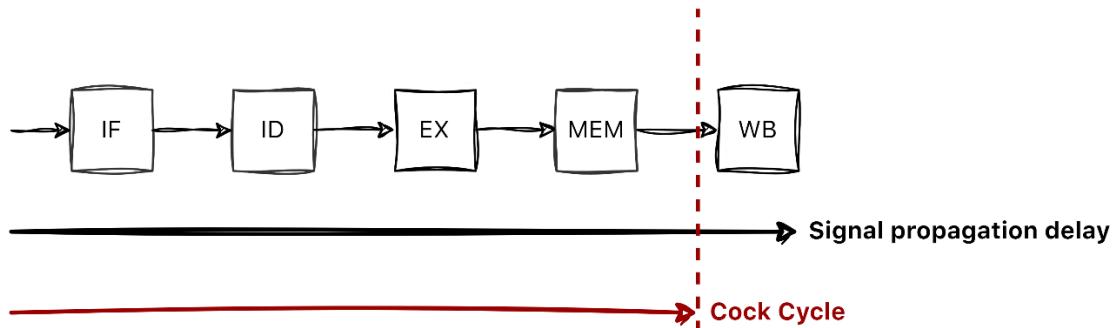
| Instruction<br>OPCode | ALUOp | Instruction<br>Operation | FUNCT  | ALUControl<br>Output | ALU<br>Operation |
|-----------------------|-------|--------------------------|--------|----------------------|------------------|
| LOAD                  | 00    | Load word                | XXXXXX | 0010                 | addition         |
| STORE                 | 00    | Store word               | XXXXXX | 0010                 | addition         |
| BEQ                   | 01    | branch equal             | XXXXXX | 0110                 | subtract         |
| R-TYPE                | 10    | add                      | 100000 | 0010                 | addition         |
| R-TYPE                | 10    | subtract                 | 100010 | 0110                 | subtract         |
| R-TYPE                | 10    | AND                      | 100100 | 0000                 | and              |
| R-TYPE                | 10    | OR                       | 100101 | 0001                 | or               |

Un esempio di come i segnali ALUOp e il campo FUNCT sono utilizzati per produrre l'input che istruisce l'ALU all'operazione corretta. Le due colonne Blu sono i segnali in input all'ALU Control, la colonna gialla indica il segnale che produce l'ALU Control e che verrà inviato all'ALU.

## Transizione al Multiciclo

Come detto nell'introduzione, sebbene le macchine a ciclo singolo possano funzionare correttamente, non vengono usati nelle implementazioni moderne, poiché sono inefficienti.

In un'implementazione Monocycle, un progettista hardware progetterebbe ciascuna "fase" come un singolo modulo e disporrà ciascuno dei moduli in linea retta, il tutto in un circuito chiuso:



Ora, questo design sembra piuttosto semplice, e lo è. Tuttavia, questa configurazione implica che la velocità di clock del processore sarà molto, molto lenta. Ma Perché? La prima regola dell'elettronica digitale pratica è che i **segnali di tensione elettrica non si propagano istantaneamente attraverso il circuito**. Ciò significa che ci sarà un notevole ritardo tra il momento in cui l'istruzione viene emessa e il momento in cui i segnali finalmente attraversano l'intero circuito per dare il risultato finale di quell'istruzione. Il clock della CPU non può essere più veloce del tempo di ritardo totale del circuito. Se il clock scorre più velocemente di quanto i segnali di tensione possano arrivare alla fine del circuito, significa che il circuito viene interrotto prima che esegua le ultime fasi, il che significa che i "risultati" del calcolo saranno da buttare. Per questo motivo, nelle architetture monociclo la velocità di clock (e, per estensione, la velocità con cui è possibile eseguire le istruzioni) è limitata dai tempi di ritardo combinati di tutti gli stadi del circuito.

Inoltre, le diverse istruzioni di un processore hanno tempi di esecuzione diversi, a seconda della quantità di lavoro necessario all'interno del datapath (ossia a seconda della porzione di datapath da attraversare). Ma il ciclo del clock deve ovviamente essere stabilito a priori, e deve avere una durata sufficiente a permettere l'esecuzione dell'istruzione con la durata maggiore, e di conseguenza si spreca tempo quando viene eseguita una istruzione che richiede meno tempo.

Come ottimizziamo quindi tutto questo, per aggirare il problema del ritardo del segnale? Utilizzando il pipelining. Se introduciamo la pipeline, non dobbiamo aspettare che il segnale si propaghi attraverso

I'intero circuito prima di avviare il successivo ciclo di clock: tutto ciò che dobbiamo fare è farlo propagare attraverso un singolo stadio. Dopo che l'istruzione 1 ha superato il primo stadio, possiamo iniziare il successivo ciclo di clock caricando l'istruzione 2 in quello stadio che ora è vuoto, e così via, e così via. Mantenendo la pipeline "piena", evitiamo la pratica dispendiosa di dover attendere che l'istruzione passi attraverso l'intero circuito prima di avviare il successivo ciclo di clock.

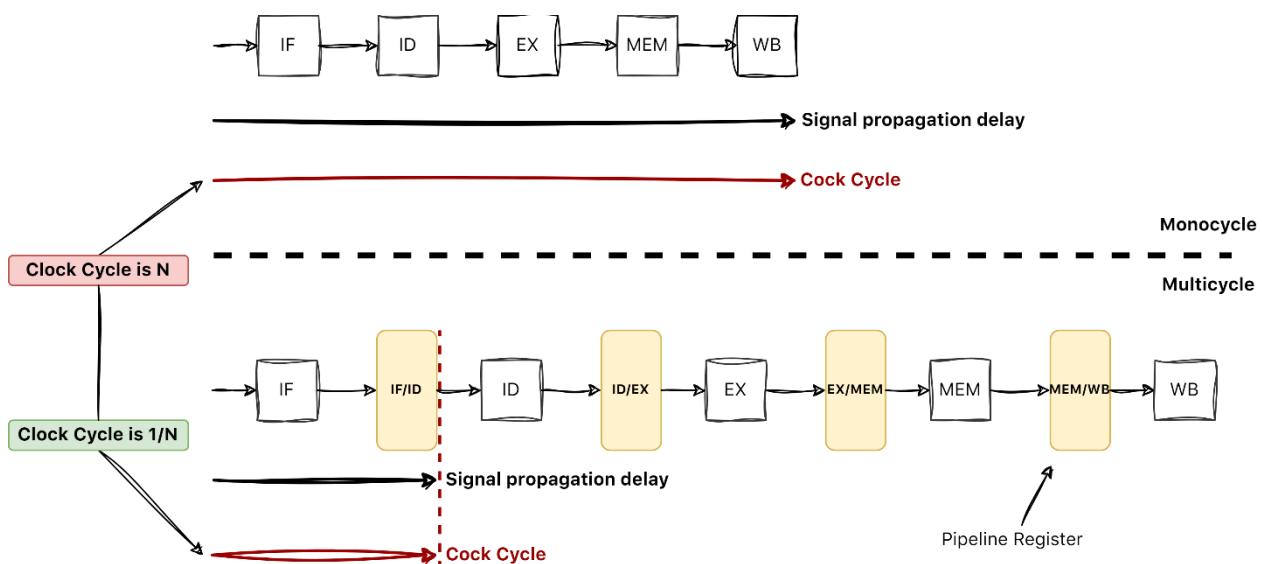
E' opportuno che ogni passo richieda più o meno lo stesso tempo di esecuzione. Infatti, se un passo è più lungo degli altri, la durata del clock dovrà necessariamente essere sufficiente all'esecuzione del passo più lungo.

Nelle architetture multiciclo con pipeline quindi, il ciclo di clock viene ridimensionato sulla base del tempo necessario ad attraversare la singola fase del datapath, e non più il datapath completo: se il datapath è stato suddiviso in N fasi, il nuovo ciclo di clock potrà essere lungo all'incirca un N-esimo della lunghezza del ciclo di clock della macchina monociclo.

Ecco perché aumentando la profondità della pipeline è possibile aumentare il clock della CPU: più fasi, meno lavoro per fase, la fase è più breve e di conseguenza il clock è alto.

### Registri di pipelining

Per permettere alla pipeline di funzionare correttamente, dato che ogni fase ad ogni ciclo di clock riceve un'istruzione diversa, è necessario che ogni fase immagazzini in un registro nascosto (a livello ISA), chiamato registro di pipeline, il risultato appena elaborato, per poi metterlo a disposizione alle fasi successive.



I registri di pipelining sono dei buffer temporanei utilizzati per trasportare risultati intermedi calcolati nella pipeline tra due stage non adiacenti della pipeline. Ogni stadio della pipeline scrive i suoi risultati intermedi in un registro di pipeline, che viene poi utilizzato da uno stadio successivo per continuare l'elaborazione dell'istruzione.

Inoltre, i registri di pipelining vengono anche utilizzati per gestire gli Hazard (o dipendenze in italiano). Perché funzioni tutto ciò è necessario che il processore sia implementato in modo tale che due istruzioni non si contengano le risorse (sia di dati che unità funzionali).

### Compromessi del Multiciclo

L'aggiunta del pipelining a un'architettura multiciclo comporta alcuni compromessi o svantaggi che è importante considerare.

In primo luogo **l'aumento di complessità dell'architettura**: L'implementazione del pipelining richiede una progettazione più complessa rispetto ad un'architettura monociclo semplice. È necessario suddividere l'esecuzione delle istruzioni in fasi e coordinare il flusso di dati attraverso di esse inserendo

registri di pipelining intermedi. Ciò comporta un aumento del numero di componenti e della complessità del circuito.

In secondo luogo, uno dei maggiori problemi è che, per vari motivi che vedremo, non tutte le combinazioni di istruzioni possono essere convogliate insieme all'interno della pipeline (**alcune istruzioni sono dipendenti tra loro**). Nel pipelining, le dipendenze tra le istruzioni possono causare quelli che vengono chiamati Hazard. Ad esempio, se un'istruzione dipende da un risultato prodotto da un'istruzione precedente ancora in esecuzione, potrebbe verificarsi un Hazard. Quando la CPU incontra questi Hazard, deve capire cosa fare al riguardo. Una soluzione è "bloccare" la pipeline: la CPU lascerà che ogni istruzione che genera Hazard "scorra" da sola lungo la pipeline mentre il resto attende. Nelle architetture Monociclo questo non può accadere in quanto ogni istruzione viene eseguita completamente prima che la successiva inizi, non si verificano situazioni in cui l'istruzione successiva dipende dai risultati di un'istruzione precedente ancora in esecuzione. Di conseguenza, non ci sono hazard come hazard di dati, hazard di controllo o hazard di dipendenza strutturale nelle architetture monociclo.

## Control Unit in Multicycle

Come cambia l'unità di controllo della versione multicyclo della MIPS? Sicuramente è più complessa, perché deve specificare quali segnali devono essere asseriti in ciascuna delle fasi, e anche quale sia la fase successiva della sequenza, per l'istruzione in esecuzione.

Ogni fase sarà quindi descritta da una tabella di verità i cui input sono:

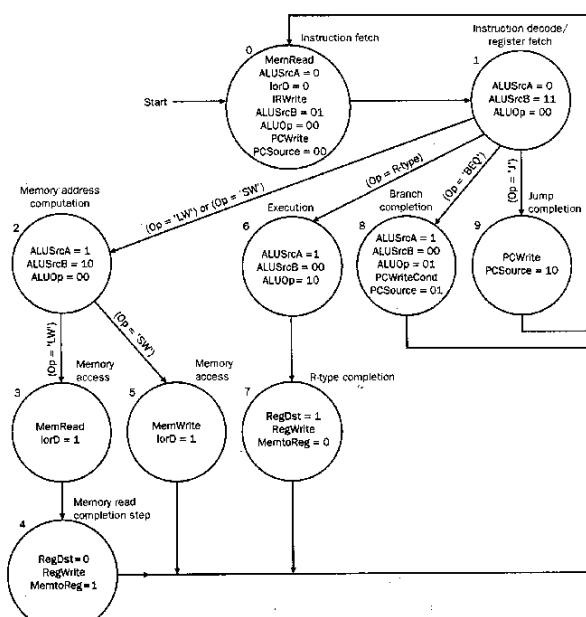
1. il tipo di istruzione in esecuzione (ossia il solito campo op)
2. La fase corrente nella sequenza di esecuzione dell'istruzione

L'output della tabella di verità di ciascuna fase sarà invece:

1. l'insieme dei segnali da asserire
2. La fase successiva nella sequenza di esecuzione dell'istruzione

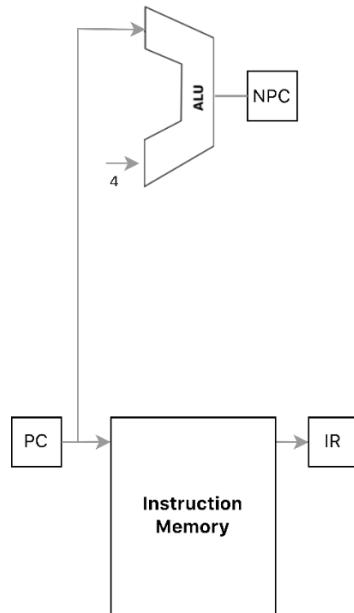
Ma questa non è altro che la descrizione informale di una Macchina di Moore: un tipo automa a stati finiti in cui:

- ad ogni stato è associato un output che dipende solo da quello stato (nel nostro caso i segnali da asserire)
- la transizione nello stato successivo dipende solo dallo stato corrente e dall'input (come in ogni automa a stati finiti)



## MIPS - Instruction fetch Phase (IF)

In questa fase, il processore legge l'istruzione da eseguire dalla memoria. Viene anche determinato l'indirizzo di memoria della prossima istruzione da eseguire.



In particolare:

- Il program counter (PC), che contiene l'indirizzo dell'istruzione attualmente in esecuzione, viene incrementato di 4 (poiché ogni istruzione MIPS occupa 4 byte di memoria).
- L'indirizzo di memoria specificato dal PC viene utilizzato per leggere l'istruzione dalla memoria.
- L'istruzione attuale viene caricata in un registro nascosto del processore chiamato instruction register (IR).
- L'indirizzo di memoria del prossimo istruzione da eseguire viene determinato ed inserito nel NPC (new program counter)

Una volta che l'istruzione è stata letta e memorizzata nell'IR, il processore passa alla fase di decode, dove l'istruzione viene decodificata e i suoi operandi vengono determinati.

## MIPS - Instruction decode/register fetch (ID)

In questa fase, il processore decodifica l'istruzione letta nella fase di fetch e determina gli operandi dell'istruzione.

Durante la fase di decode, il processore esegue le seguenti operazioni:

- L'istruzione viene decodificata utilizzando analizzando i bit dell'istruzione stessa. I bit più significativi (opcode) indicherà il tipo di istruzione (ad esempio, somma, moltiplicazione, caricamento da memoria, etc.), mentre gli altri bit specificano i dettagli dell'istruzione, come gli operandi e gli indirizzi di memoria.
- Vengono determinati gli operandi dell'istruzione. Gli operandi possono essere registri del processore o valori di memoria.
  - Non si sa ancora se i registri andranno usati, cioè se questa è un'istruzione di tipo-R, ma non è dannoso leggerli, e potrebbero servire ai passi successivi, per cui vengono letti dal register file e memorizzati nei registri nascosti A e B (invisibili al livello ISA).
  - Il register file viene quindi letto ad ogni ciclo di clock, e i registri nascosti A e B vengono sovrascritti ad ogni ciclo a prescindere dal tipo di istruzione

- Vengono determinati i valori degli operandi. Se gli operandi sono registri del processore, i loro valori vengono letti dai registri. Se gli operandi sono valori di memoria, vengono letti dalla memoria.
- Viene calcolato l'eventuale indirizzo di destinazione dei branch, e memorizzato nel registro interno ALUoutput, da cui verrà prelevato al ciclo di clock successivo se l'istruzione è effettivamente di branch.

## MIPS – Execute Phase (EX)

In questa fase, il processore esegue l'operazione specificata nell'istruzione.

Durante la fase di esecuzione, il processore esegue le seguenti operazioni:

- L'opcode dell'istruzione viene utilizzato per determinare il tipo di operazione da eseguire (La Control Unit pilota la ALU usando l'opcode - estratto dall'instruction register - e la ALU combina gli operandi - preparati al ciclo precedente). Ad esempio, se l'opcode indica un'operazione di somma, il processore eseguirà l'operazione di somma sugli operandi specificati. Se l'opcode indica un'operazione di confronto, il processore eseguirà l'operazione di confronto sugli operandi specificati.
- Vengono eseguiti i calcoli necessari per eseguire l'operazione. Ad esempio, se l'operazione è una somma, verranno sommati i valori degli operandi. Se l'operazione è un confronto, verrà determinato se i valori degli operandi sono uguali o diversi.
- Il risultato dell'operazione viene memorizzato in un registro del processore o in una locazione di memoria, a seconda della specifica dell'istruzione.

## MIPS – Memory Access Phase (MEM)

In questa fase, il processore può accedere alla memoria per leggere o scrivere dati.

Durante la fase di memoria, il processore esegue le seguenti operazioni:

- Viene determinato se l'istruzione richiede l'accesso alla memoria. Se l'istruzione non richiede l'accesso alla memoria, il processore passa direttamente alla fase di writeback.
- Se l'istruzione richiede l'accesso alla memoria, il processore utilizza l'indirizzo di memoria specificato nell'istruzione per accedere ai dati nella memoria.
- Se l'istruzione è un'operazione di lettura, il processore legge i dati dalla memoria e li memorizza in un registro del processore o in una locazione di memoria, a seconda della specifica dell'istruzione.
- Se l'istruzione è un'operazione di scrittura, il processore scrive i dati specificati nell'istruzione nella memoria all'indirizzo di memoria specificato.

# Instruction level parallelism

L'Instruction level parallelism (ILP) si riferisce alla tecnica di progettazione di microarchitettura che mira a sfruttare al meglio la capacità di eseguire più istruzioni allo stesso tempo, al fine di aumentare la velocità di esecuzione delle istruzioni. Si sfruttano in sostanza una serie di strategie per massimizzare al massimo il parallelismo a livello di istruzione.

Avendo gli Hazard da controllare, l'IPC ideale di una pipeline non sarà comunque 1. Se immaginiamo uno scenario in cui il problema degli Hazard non fosse presente, ci sarebbero due modi per aumentare le prestazioni di una CPU:

1. Incremento della frequenza del clock
2. Incremento del numero di istruzioni **eseguite** in parallelo

La prima soluzione richiederebbe di aumentare la profondità del datapath, suddividendo il lavoro in un numero maggiore di fasi. Aumentando la profondità ci saranno di conseguenza più istruzioni in esecuzione contemporaneamente, in fasi diverse della pipeline, e in una data unità di tempo verrà completato un maggior numero di istruzioni.

Tuttavia, questa soluzione non può protrarsi all'infinito, in quanto:

- Maggiore è il numero di fasi, e maggiore è la complessità della pipeline, e quindi della sua control Unit.
- Frequenze di clock maggiori producono interferenze tra le piste, consumi e consequenti problemi di dissipazione del calore

La seconda soluzione, quindi di aumentare il numero di istruzioni eseguite in parallelo, prende anche il nome di **Multiple Issue**. Utilizzando il multiple issue, si suppone ovviamente, che le istruzioni eseguite in parallelo non siano dipendenti tra loro.

Questa seconda tecnica può essere vista come un insieme di pipeline che lavorano in parallelo, e quindi richiede un datapath più ampio, in grado cioè di trasportare le varie informazioni di tutte le istruzioni lanciate in parallelo da uno stadio della pipeline al successivo.

Aumentando il numero di istruzioni eseguite in parallelo, avendo ad esempio due pipeline in parallelo, non avremmo più un IPC ideale uguale a 1, ma dato che abbiamo due pipe, avremmo che ad ogni istruzione verranno eseguite 0,5 istruzioni. In generale:

$$IPC = \frac{1}{K}$$

Dove  $K$  è il numero di Pipeline **perfette**

Ovviamente con l'introduzione degli Hazard non si hanno mai Pipeline perfette, quindi non ci sarà mai un  $IPC = 1/K$ . Inoltre, le architetture multiple issue vengono di solito dette superscalari, anche se, più correttamente, questo termine va riservato alle architetture a multiple issue "dinamico".

Per implementare il multiple issue è necessario determinare quali e quante istruzioni possono essere avviate all'esecuzione in un dato ciclo di clock. In particolare, la ricerca va fatta tra le istruzioni in attesa di essere eseguite, e ovviamente ci sono dei limiti pratici a quante istruzioni possono essere analizzate per trovare quelle indipendenti fra loro.

Una volta individuate, si dice che le istruzioni vengono impacchettate in un **issue packet**, e avviate all'esecuzione nello stesso **issue slot** (ossia nello stesso ciclo di clock).

I processori multiple issue si possono dividere in due grandi categorie, a seconda di come e quando vengono risolti questi due problemi (individuazione delle istruzioni ed impacchettamento):

1. **processori a multiple issue statico**: In questo tipo di architettura, il compilatore è responsabile dell'identificazione delle istruzioni indipendenti e della loro "impacchettamento" in un issue

packet, che verrà poi eseguito dal processore in parallelo in un unico ciclo di clock. In particolare:

- a. Quando il processore preleva dalla Instruction Memory un pacchetto di istruzioni preparato dal compilatore, sa già che potrà eseguire quelle istruzioni in parallelo senza rischi di hazards presenti tra queste istruzioni.
- b. Il numero di istruzioni nel pacchetto è stabilito a priori, nella fase di progettazione del processore, e si parla di architettura a **issue statico**.
- c. Tuttavia, l'issue statico dipende dalla capacità del compilatore di individuare le istruzioni indipendenti e può essere limitato dal numero di istruzioni che il processore è in grado di eseguire in parallelo in un unico ciclo di clock.

2. **processori a multiple issue dinamico**: è la CPU che analizza a run time le istruzioni e decide quali mandare in esecuzione in parallelo nello stesso ciclo di clock.

- a. Ovviamente c'è un limite al numero di istruzioni che la logica del processore è in grado di esaminare per trovare pacchetti di istruzioni indipendenti, e il numero di istruzioni individuate può cambiare ad ogni ciclo di clock, fino al numero massimo (3 o 4 istruzioni sono i valori più frequenti) consentito dall'architettura.
- b. Si parla in questo caso, invece, di **issue dinamico**.

Quindi, quando tutto il lavoro di ricerca del parallelismo tra le istruzioni è fatto dal compilatore siamo in presenza di **Instruction Level Parallelism Statico**, mentre se lo stesso lavoro è svolto dalla CPU, siamo in un contesto di **Instruction Level Parallelism Dinamico**. Ovviamente l'ILP non è solo l'issue statico/dinamico ma un'insieme di strategie che insieme contribuiscono alla miglior forma di parallelismo.

A livello terminologico, è ai processori pipelined che implementano il multiple issue dinamico che spetta la definizione di processori superscalari.

# Hazards and dependencies

Determinare in che modo un'istruzione dipende da un'altra è fondamentale per determinare quanto parallelismo esiste in un programma e come tale parallelismo può essere sfruttato. In particolare, per sfruttare il parallelismo a livello di istruzione, dobbiamo determinare quali istruzioni possono essere eseguite in parallelo. Se due istruzioni sono parallele, possono essere eseguite simultaneamente in una pipeline di profondità arbitraria senza causare stalli, assumendo che la pipeline disponga di risorse sufficienti (e quindi non esistano pericoli strutturali). Se due istruzioni sono dipendenti, non sono parallele e devono essere eseguite in ordine, anche se spesso possono essere parzialmente sovrapposte. La chiave in entrambi i casi è determinare se un'istruzione dipende da un'altra istruzione.

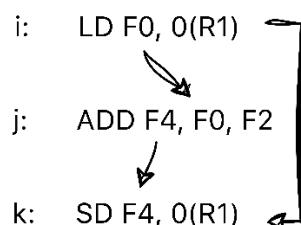
Ci sono 3 tipi di dipendenze tra le istruzioni di un programma:

- (true) Data Dependences
- Name dependences
  - Anti-dependence
  - Output-dependence
- Control Dependences

Un hazard invece è un evento che impedisce o ritarda l'esecuzione di un'operazione in una pipeline. Mentre la dipendenza è una cosa intrinseca del programma, l'hazard è un qualcosa che dipende dall'hardware (dalla microarchitettura). Ad esempio se si ha una dipendenza sui dati tra l'istruzione i e l'istruzione j, non è detto che questa dipendenza provochi un hazard sia nell'architettura x che in quella y, dipende se le architetture implementano misure per mitigare la dipendenza ed evitare l'hazard.

## Data Dependences

Questo tipo di dipendenza riguarda una situazione in cui un'istruzione del programma fa riferimento ai dati di un'istruzione precedente. In particolare, la dipendenza sui dati si verifica quando un'istruzione dipende da un dato che non è ancora stato prodotto da un'istruzione precedente o per transitività da una istruzione a monte.



In questo caso, l'istruzione 'k' è (true) data dependent da un istruzione 'i' precedente:

- 'i' produce un risultato che 'k' utilizzerà
- 'k' è data dependent da 'j' che a sua volta è data dependent da 'i'

Se non trattata, la (true) data dependence potrebbe risultare in una Read-After-Write (RAW) Hazard

## Name dependences – Anti-Dependence

L'anti-dependence si verifica quando un'istruzione 'i' legge un dato che dovrà essere scritto successivamente da un'istruzione 'j'.



In questo esempio, l'istruzione 'i' e l'istruzione 'j' sono anti-dependence tra di loro in quanto 'i' legge un dato che 'j' deve scrivere successivamente. In principio, non è possibile quindi riordinare 'i' e 'j' in quanto riordinandole si alterebbe il comportamento del programma.

L'anti-dependence è anche chiamata Name Dependence in quanto un'istruzione 'i' utilizza lo stesso nome (registro o locazione di memoria) di un'altra istruzione 'j' ma tra di essi non c'è uno scambio di dati. Infatti un modo per rimuovere l'anti-dependence è introdurre il renaming dei registri. Le istruzioni coinvolte in una dipendenza sui nomi possono essere eseguite simultaneamente se il nome del registro usato nelle istruzioni viene cambiato in modo da evitare conflitti.

Se non trattata, l'anti-dependence potrebbe risultare in un Write-After-Read (WAR) Hazard

## Name dependences – Output Dependence

L'output dependence si verifica quando un'istruzione 'i' e un'altra istruzione 'j' scrivono lo stesso dato. Si verifica quando l'ordinamento delle istruzioni influenzera il valore di output finale di una variabile.



Anche in questo caso, non è possibile riordinare 'j' e 'k', altrimenti si alterebbe il comportamento del programma.

Anche la Output Dependence è una Name Dependence, e come l'anti-dependence è possibile rimuoverla introducendo il renaming dei registri. In quel caso, l'istruzione j può essere eseguita prima o in parallelo rispetto a k o i;

Se non trattata, l'output dependence potrebbe risultare in un Write-After-Write (WAW) Hazard

## Data Hazard solutions

Una tecnica utilizzata per risolvere gli hazard sui dati è chiamata forwarding: Il **forwarding**, o "short-circuiting", è una tecnica che consiste nel bypassare il normale flusso di dati all'interno della CPU e "inoltrare" direttamente il risultato di un'operazione ad un altro registro o istruzione che ne ha bisogno, senza dover aspettare che il risultato venga memorizzato in un registro nel banco dei registri e poi ripreso in seguito. In questo modo, si evita il ritardo causato dall'accesso alla memoria e dal trasferimento dei dati tra i diversi componenti della CPU, riducendo così il tempo di esecuzione delle istruzioni. Tuttavia, il forwarding richiede una maggiore complessità hardware e logiche di controllo per gestire correttamente il flusso dei dati e assicurare la correttezza del risultato finale.

Il forwarding può aiutare quindi a ridurre il numero di stall dovuti ad Hazard sui dati, ma non può eliminarli completamente. Ad esempio, il forwarding non può risolvere i stall causati dall'accesso alla memoria, poiché richiede comunque tempo per leggere o scrivere i dati nella memoria. Ad esempio:

```

lw $t0, 0($s0) # $t0 = mem[$s0]
add $t1, $t0, $s1 # $t1 = $t0 + $s1
sw $t1, 0($s2) # mem[$s2] = $t1

```

Il forwarding non può essere utilizzato per eliminare questo stall causato dall'istruzione "lw", poiché il valore 0 (\$s0) deve essere prima caricato dalla memoria e quindi passare la fase di WB. L'hardware di controllo della pipeline deve accorgersi della situazione e fermare le istruzioni successive, in sostanza, la ADD e la SW non possono avere il dato pronto e quindi attenderanno.

La true dependency non c'è modo di aggirarla: Non c'è modo di evitarli davvero, ecco perché i pericoli RAW sono chiamati **vere** dipendenze. Le istruzioni devono attendere che i loro input siano pronti prima di poter essere eseguite.

Inoltre, il forwarding può essere limitato dalla disponibilità di registri e dalle risorse di elaborazione della CPU, che possono essere insufficienti per gestire tutte le istruzioni in modo efficiente.

WAR (antidipendenza) e WAW (dipendenza dall'output) invece, sono dipendenze dei nomi (name dependency): è possibile evitare il conflitto cambiando i nomi delle variabili.

La rinominazione può essere fatta staticamente dal compilatore o dinamicamente dalla CPU mentre esegue le istruzioni.

Esempio:

```
DIV $F0, $F2, $F4
ADD $F6, $F0, $F8
SUB $F8, $F10, $F14
MUL $F6, $F10, $F12
```

In questo esempio abbiamo due dipendenze sui nomi:

- Antidipendenza (WAR) tra ADD che deve leggere \$F8, e SUB che deve scrivere in \$F8 (Write after Read)
- Dipendenza in output (WAW) tra ADD e MUL

È possibile usare la tecnica del register renaming per bypassare questo problema:

```
DIV F0, F2, F4
ADD F6, F0, F8
SUB F9, F10, F14
MUL F11, F10, F12
```

Naturalmente, perché la rinominazione sia possibile occorre che **F11** ed **F9** siano disponibili. Occorre anche tener traccia di queste modifiche per le istruzioni che seguono quelle modificate. Da notare che la true dependency tra ADD e DIV permane (\$F0 non sarà disponibile finché non sarà scritto – RAW).

Le dipendenze vengono risolte ormai tramite degli algoritmi implementati nei processori come lo scoreboarding e l'algoritmo di Tomasulo che implementano a loro volta anche una sorta di rinominazione dei registri.

## Control dependences

Gli hazard di controllo si verificano quando viene eseguita un'istruzione di branch, che può cambiare il valore del program counter.

Ad esempio:

```
LD R1, 0 (R4)
BNE R0, R1, else
DADD R1, R1, R2
JMP next
else: DSUB R1, R1, R3
```

```
next: OR R4, R5, R6
```

In questo caso, si verifica un hazard di controllo poiché, quando sappiamo se il branch verrà eseguito, la DADD ha già completato la fase IF.

### Control hazards solution

Un modo semplice di gestire la situazione è di eseguire comunque la fase di fetch dell'istruzione fisicamente successiva al branch (ossia la DADD). Alla fine della fase di ID del branch, si sa se il salto verrà eseguito o no:

- Se non viene eseguito, tutto bene, la DADD prosegue con la fase ID.
- Se il salto va eseguito, si deve eseguire la DSUB, che riparte ovviamente dalla fase IF, e si è quindi sprecato un ciclo di clock (la fase IF della ADD).

In questo esempio, anche assumendo una implementazione aggressiva che esegue un branch in due cicli di clock, se il salto dovesse essere eseguito, il PC non verrebbe aggiornato se non alla fine del secondo ciclo di clock, nello stage ID, dopo che è stato calcolato l'indirizzo da inserire nel PC. Quindi, dato che siamo in un contesto in cui è presente la pipeline, l'istruzione "fisicamente" successiva al branch (ossia quella all'indirizzo PC+4, quindi la ADD) ha già terminato la fase IF. Ma se il salto va effettuato non è quella la prossima istruzione da eseguire.

Una delle tecniche utilizzate (anche quella meno efficace) è quella della **predizione statica dei branch**. Nella predizione statica dei branch, la CPU assume che la decisione di prendere o meno un'istruzione di salto sarà sempre la stessa (la previsione non cambia durante l'esecuzione del programma, essa viene fissata a priori) ad esempio:

- Previsione che tutti i salti siano sempre presi oppure,
- Previsione che solo le istruzioni di salto con determinati OPCode siano presi mentre le altre siano non presi oppure,
- Previsione che tutti i salti all'indietro (backwards branch) siano presi e tutti i salti in avanti (forward branch) non presi oppure,
- Previsione che la prima volta che si verifica una certa istruzione di salto essa si consideri sempre presa

Se però l'assunzione è errata, l'istruzione di cui si è iniziata l'esecuzione erroneamente deve essere trasformata in una no-op, e bisogna ripartire dal fetching dell'istruzione giusta.

Se la maggior parte delle previsioni si rivela corretta, lo spreco di cicli di clock risulta abbastanza limitato. Ad esempio, in un loop eseguito più volte, il salto all'indietro per ricominciare il loop, viene eseguito più volte. Quello in avanti invece (quello per uscire dal loop) solo quando il loop termina, e quindi una volta sola. Oppure i salti in avanti sono spesso usati in caso di condizioni di errore, per trasferire il controllo al codice di gestione dell'errore (che spesso viene messo alla fine di un programma). Se si assume che gli errori siano rari, anche i salti in avanti saranno rari, e dunque basso il costo di eventuali previsioni errate.

Naturalmente, questo tipo di previsione non funziona per dei semplici if then else.

Un'altra soluzione sarebbe quella di adottare i **delayed branch**. Questa tecnica consiste nello spostare dopo un branch una istruzione che è comunque necessario eseguire, indipendentemente dall'esito del branch. L'istruzione viene quindi portata a termine mentre la CPU ha il tempo di completare la valutazione della condizione di salto. Quando è terminata l'istruzione, si saprà se il salto dovrà essere eseguito o no.

Questa tecnica richiede l'intervento del compilatore, e può essere usata solo in una CPU progettata esplicitamente per implementare il delayed branch. Inoltre, non è sempre di facile applicazione.

## Structural Hazards

Questo tipo di Hazard si verifica quando alcune particolari combinazioni di istruzioni non possono essere eseguite simultaneamente nella pipeline a causa del numero limitato di unità funzionali disponibili (se ad esempio c'è una sola ALU non può essere usata nello stesso ciclo di clock da due diverse istruzioni)

- Alcune risorse hardware all'interno del datapath non sono duplicate, e ovviamente non possono essere contemporaneamente usate da due istruzioni nella pipeline.
- Ad esempio, due istruzioni nella pipeline non possono usare la stessa ALU nello stesso ciclo di clock.
- Per questa ragione nelle moderne CPU (ma anche in quelle vecchie), molte unità funzionali, in particolare quelle combinatorie, come le ALU, sono duplicate più volte

Una conseguenza comune ad un hazard è uno stall della pipeline.

## Stall della pipeline

Uno stall si verifica quando il processore è costretto a interrompere temporaneamente l'esecuzione delle istruzioni in modo da risolvere un hazard (questo perché non ci sono abbastanza tecniche per mitigare l'hazard). Durante lo stall, il processore non può continuare a eseguire le istruzioni e deve attendere che l'hazard sia risolto prima di poter riprendere l'esecuzione. Un tale evento è spesso chiamato bolla - bubble, per analogia con una bolla d'aria in un tubo del fluido.

Lo stall può avere un impatto significativo sulla velocità di esecuzione delle istruzioni, poiché il processore non può utilizzare al meglio la sua capacità di esecuzione in parallelo.

# Dynamic instruction level parallelism

Le tecniche utilizzate nel parallelismo dinamico (oltre all'issue dinamico) sono le seguenti:

- scheduling dinamico della pipeline
  - branch prediction
  - speculazione hardware

che insieme cercano di implementare il miglior parallelismo possibile, con lo scopo di minimizzare l'effetto negativo degli Hazard ed avvicinare le prestazioni della pipeline al suo valore ideale.

Va comunque precisato che ILP dinamico e statico non sono completamente distinti. Come sarà chiaro quando avremo visto anche le tecniche dell'ILP statico, i processori di una categoria adottano sempre almeno qualche tecnica di base dell'altra categoria.

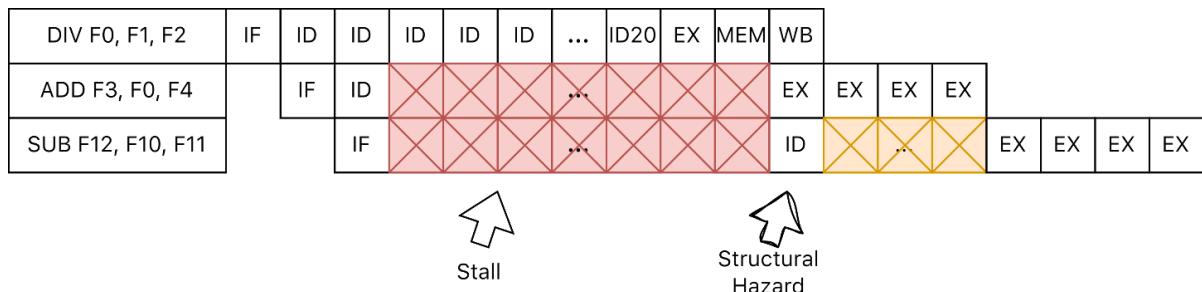
Sebbene un processore pianificato dinamicamente non possa modificare il flusso di dati, cerca di evitare lo stall quando sono presenti dipendenze. Al contrario, la programmazione statica della pipeline da parte del compilatore cerca di minimizzare gli stalli separando le istruzioni dipendenti in modo che non portino ad Hazards.

## Scheduling dinamico e algoritmo di Tomasulo

Uno dei principali limiti delle semplici tecniche di pipelining è che prelevano ed eseguono le istruzioni in ordine: le istruzioni vengono emesse nell'ordine del programma e se un'istruzione è bloccata nella pipeline, nessuna istruzione successiva può procedere. Pertanto, se c'è una dipendenza tra due istruzioni ravvicinate nella pipeline, ciò porterà ad un Hazard e ne risulterà uno stall e se sono presenti più unità funzionali, queste unità potrebbero rimanere inattive.

Lo scheduling dinamico della pipeline (o pianificazione dinamica delle istruzioni) è una delle tecniche che viene utilizzata nell'ILP dinamico al fine di ridurre gli Hazards sui dati e consiste nel modificare dinamicamente l'ordine di esecuzione delle istruzioni all'interno di una pipeline per cercare di tenerla il più possibile attiva minimizzando gli stall della pipeline dovuti alle dipendenze sui dati. Dunque, essendo ILP dinamico, la tecnica di scheduling dinamico è implementata dal processore a run-time.

Per giustificare il dynamic scheduling, si consideri questo scenario:

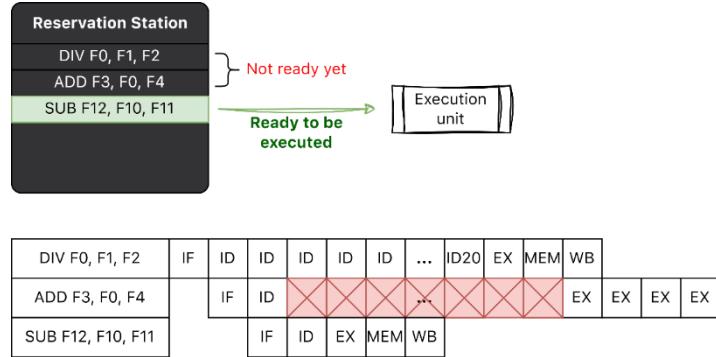


In questo scenario, la DIV impiega 20CC nella sua fase di ID (magari per un cache miss). Dato che la ADD è (true) data dependent, questo non può che trasformarsi in un RAW Hazard, stallando la ADD per 18CC + EX + WB. Il problema è che anche la SUB stalla per altrettanti cicli, anche se non dipende da nessuna delle istruzioni precedenti. L'idea principale del dynamic scheduling è proprio quella di bypassare questi stall dovuti alla dipendenza dai dati, eseguendo le istruzioni out-of-order. Da notare che c'è anche un Hazard strutturale in questo esempio. Questo è dovuto al fatto che esiste solo una ALU nel sistema in cui sono state eseguite queste tre istruzioni, la SUB dovrà aspettare che si liberi.

Per effettuare l'esecuzione fuori ordine, è necessario introdurre **l'algoritmo di Tomasulo** che è il cuore dell'out of order execution. L'algoritmo di Tomasulo è un algoritmo hardware per implementare la pianificazione dinamica delle istruzioni sviluppato nel 1967 per l'IBM360/91 da Robert Tomasulo, ricercatore dell'IBM. Tomasulo volle trovare un metodo per avere delle migliori Floating Point

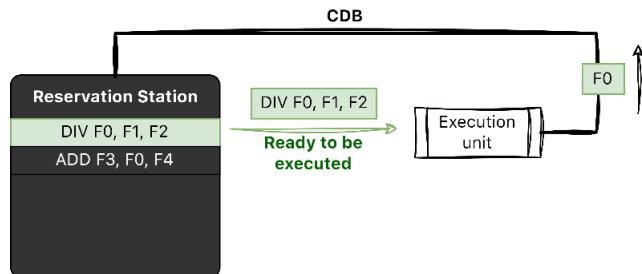
performance senza ricorrere a compilatori speciali che ottimizzassero il codice, utilizzando l'IBM360/91. Il problema è che questa macchina era molto limitata e non riusciva a raggiungere alte performance in FP, in particolare:

- Pochi registri Floating Point (4)
  - Questo motivò Tomasulo a trovare un modo per avere più registri effettivi, e lo fece implementando la ridenominazione dei registri a livello hardware
- Accessi alla memoria molto lunghi
  - Che è lo scenario di prima. Questo è il motivo principale dell'algoritmo.

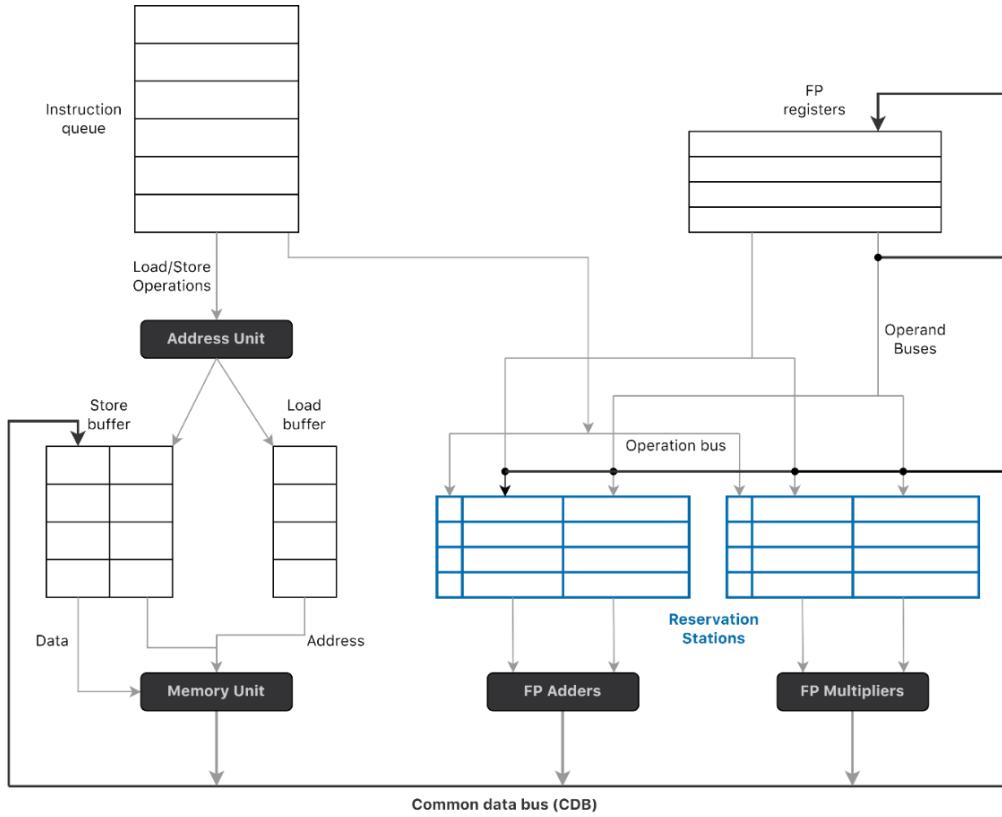


Per permettere alla SUB di essere eseguita prima della ADD, bisogna "parcheggiare" le istruzioni in dei buffer, che nell'algoritmo di Tomasulo prendono il nome di Reservation Station. Queste stazioni controlleranno gli operandi di tutte le istruzioni all'interno, accorgendosi del fatto che la SUB può ricevere gli operandi dal banco dei registri senza dover aspettare la DIV e la ADD.

Per permettere poi alla ADD di procedere è necessario informare la RS (reservation station) quando un nuovo operando è disponibile. Questo è implementato da Tomasulo dal Common Data Bus (CDB).



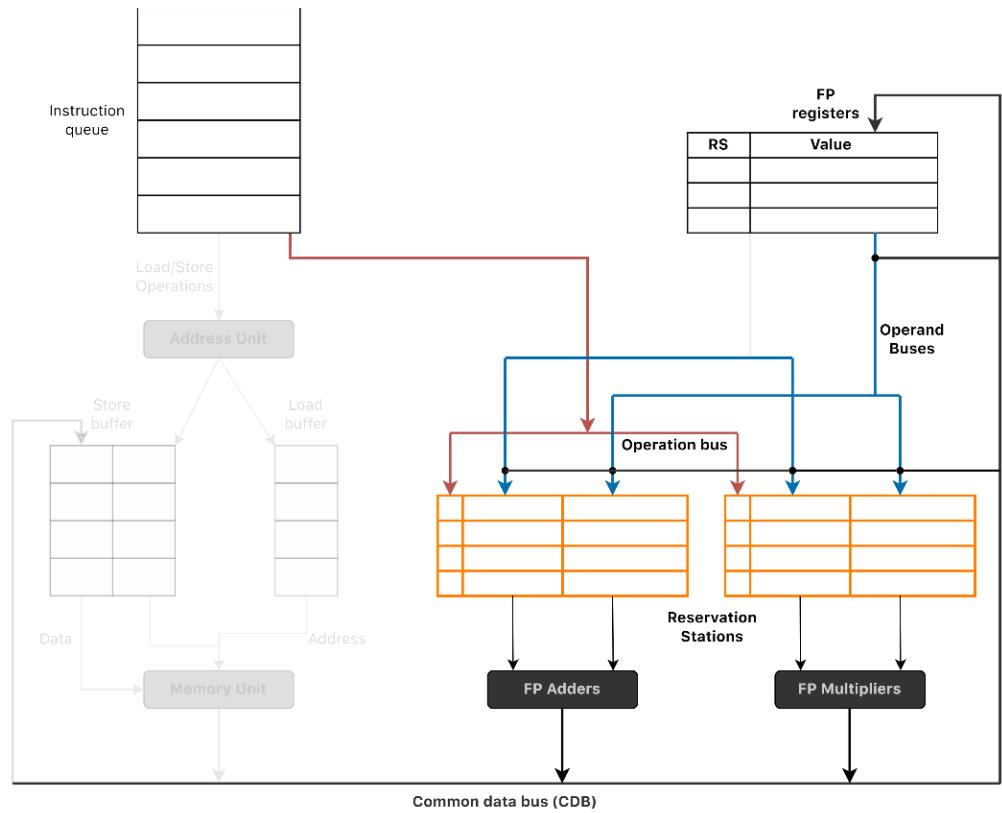
Schema di Tomasulo applicato alla MIPS:



Nello schema di Tomasulo, l'esecuzione di una istruzione è suddivisa in due macropassi fondamentali (ciascuno dei quali può essere suddiviso in più fasi della pipeline):

- ISSUE Phase
- EXECUTE Phase

### ISSUE Phase



La fase di ISSUE dell'algoritmo di Tomasulo è responsabile di prelevare, decodificare e inserire le istruzioni in una entry della stazione di prenotazione associata all'unità funzionale che dovrà eseguirle. In dettaglio, durante questa fase:

1. L'istruzione viene prelevata dalla memoria e decodificata per determinare il tipo di operazione da eseguire e gli operandi necessari.
2. Viene quindi cercata una entry vuota nella **stazione di prenotazione associata all'unità funzionale che eseguirà l'istruzione**.
3. Una volta trovata una entry vuota, l'istruzione viene inserita in essa e i suoi operandi vengono letti dal register file.
4. **Se tutti gli operandi sono disponibili immediatamente** (nel register file, o in qualche altra stazione di prenotazione) **vengono prelevati e mandati nella entry in cui è stata messa l'istruzione** e successivamente inviata all'unità funzionale per l'esecuzione. Altrimenti, l'istruzione rimane in attesa nella stazione di prenotazione finché gli operandi necessari non sono disponibili.
  - a. Per ogni operando che manca, nella entry dell'istruzione viene scritto l'identificativo della entry/stazione di prenotazione in cui è presente l'istruzione che dovrà produrre quell'operando.
5. Se non ci sono entry vuote disponibili, la pipeline viene stallata fino a quando una entry diventa disponibile.

Per sapere che gli operandi di una istruzione possono trovarsi in una entry di una qualche stazione di prenotazione oppure devono ancora essere prodotti da una istruzione che attualmente si trova in una entry di una qualche stazione di prenotazione, la logica di controllo della CPU, nella fase di ISSUE, ha dovuto tenere conto delle istruzioni prelevate in precedenza e già instradate verso le varie stazioni di prenotazione, e da cui l'istruzione corrente potrebbe dipendere.

Al fine di mantenere e tracciare lo stato di ogni istruzione, le RS sono composte da 7 campi.

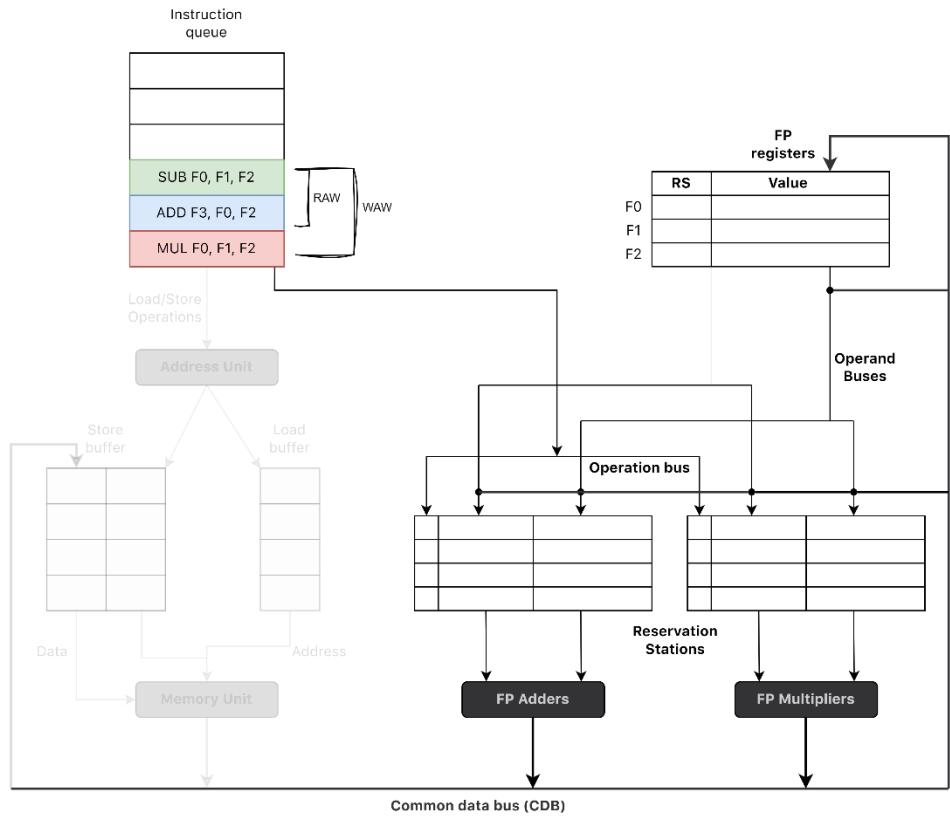
### Struttura delle Reservation Station

L'informazione di ogni entry di ogni stazione di prenotazione è organizzata in campi specifici per supportare l'algoritmo di Tomasulo. I campi:

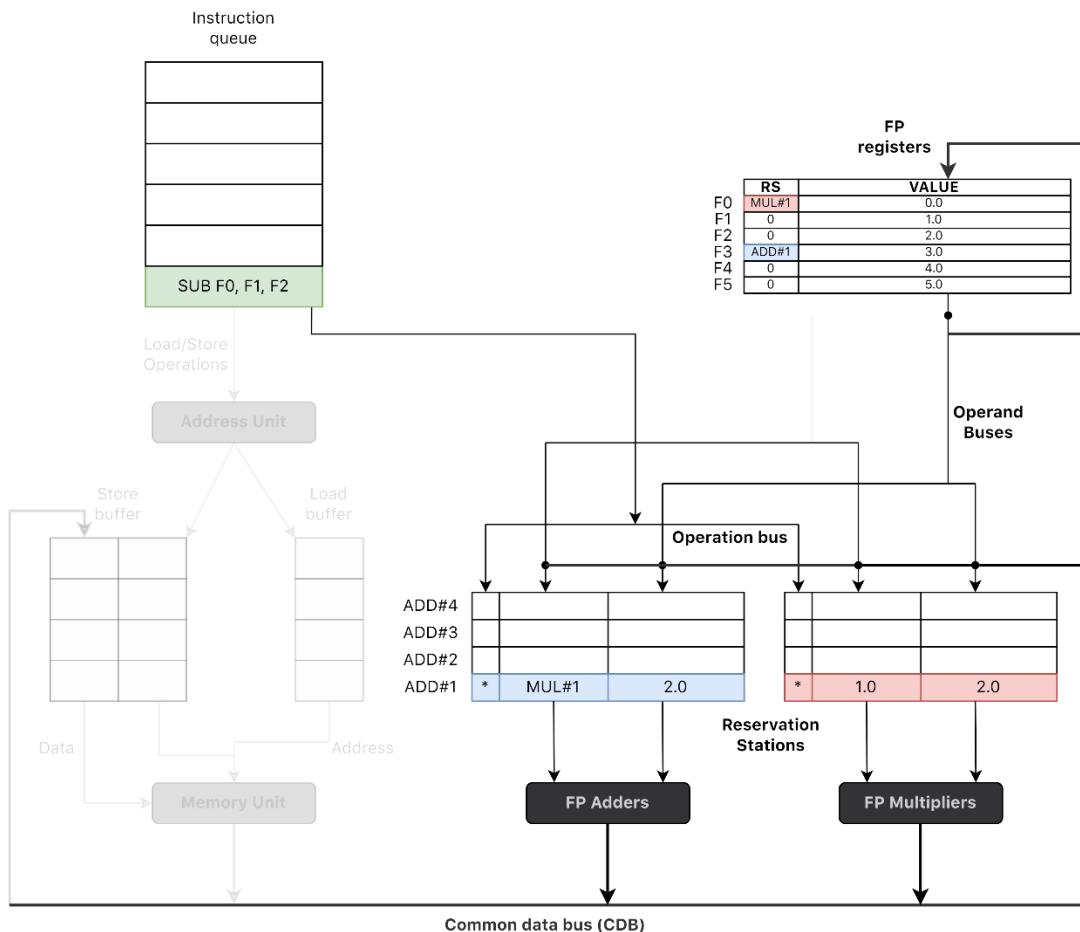
- **Op:** indica l'operazione da eseguire sugli operandi.
- **Q<sub>j</sub>, Q<sub>k</sub>:** indicano le entry delle stazioni che produrranno il risultato atteso da Op. Un valore di zero indica che l'operando è già presente in V<sub>j</sub> o V<sub>k</sub>.
- **V<sub>j</sub>, V<sub>k</sub>:** indicano i valori dei due operandi.
- **A:** presente solo nelle stazioni di prenotazione per load/store, contiene prima il valore immediato per la LOAD o STORE, e dopo che è stato calcolato, l'indirizzo effettivo in RAM in cui leggere/scrivere il dato.
- **Busy:** indica che la stazione è attualmente in uso.

Inoltre, ad ogni registro del banco dei registri è associato un campo **Q<sub>i</sub>**, che indica l'entry della stazione di prenotazione che contiene l'istruzione il cui risultato dovrà andare in quel registro. Se Q<sub>i</sub> vale 0, significa che non c'è alcuna istruzione che sta calcolando un valore che deve andare in quel registro.

Esempio:



Ci sono 3 istruzioni, la ADD e la SUB sono (true) data dependent, mentre la SUB e la MUL sono output dependent. L'algoritmo di tomasulo aiuta ad avivare glie ventuali RAW, WAR e WAW Hazard.



Questa immagine illustra i primi due clicli di clock.

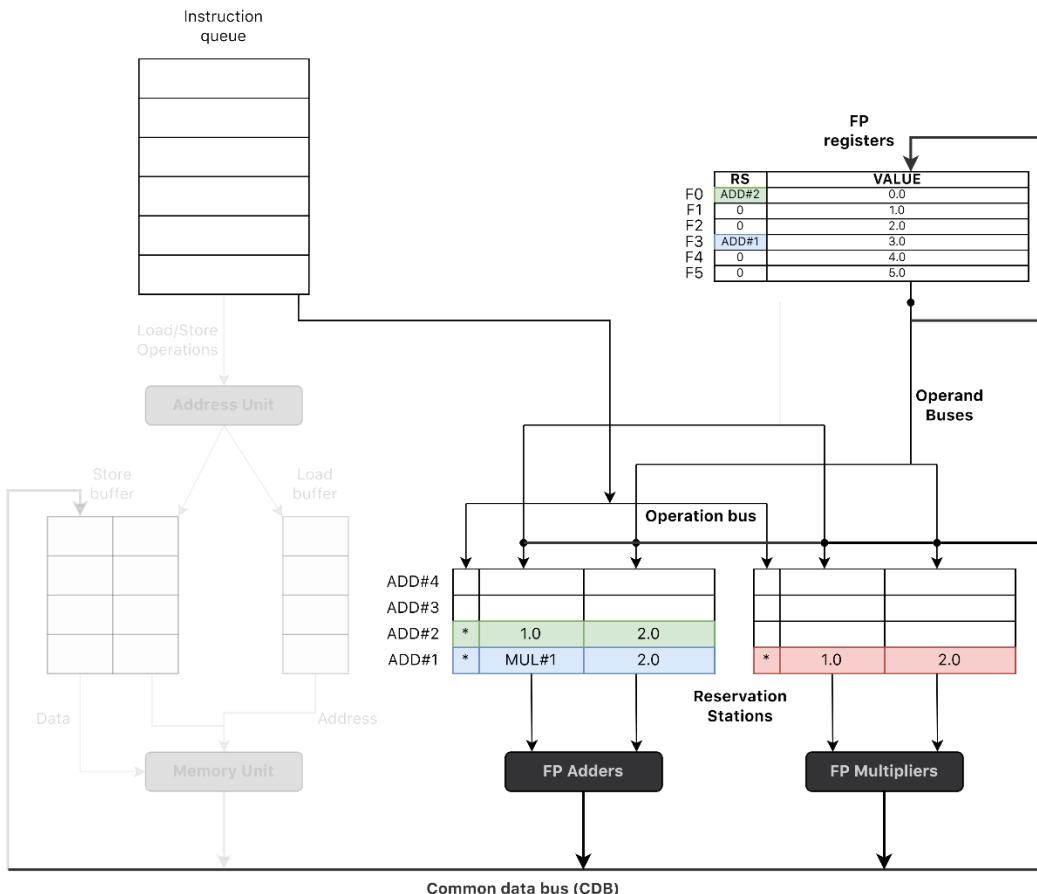
Primo ciclo di clock, come descritto dall'ISSUE Phase:

1. L'istruzione (prima la MUL) viene inviata alla stazione di prenotazione corrispondente.
2. Se ci sono degli operandi già disponibili nel register file, questi vengono inoltrati alla entry della MUL attraverso l'operand bus (sia F1 che F2 sono disponibili).

Secondo ciclo di clock, come descritto dall'ISSUE Phase:

1. L'istruzione successiva alla MUL (ADD) viene inviata alla stazione di prenotazione corrispondente.
2. Se ci sono degli operandi già disponibili nel register file, questi vengono inoltrati alla entry della ADD attraverso l'operand bus (solo F2 è disponibile).
  - a. Per ogni operando che manca, nella entry dell'istruzione viene scritto l'identificativo della entry/stazione di prenotazione in cui è presente l'istruzione che dovrà produrre quell'operando. In questo caso la ADD punta alla prima entry della MUL RS, il quale dovrà produrre il dato.

Si nota che in questo modo è stato implementato in modo implicito il forwarding: la ADD non dovrà aspettare che F0 sia disponibile dal banco dei registri, ma grazie al CDB, appena la MUL produrrà il risultato verrà inoltrato alla entry della ADD.



Terzo ciclo di clock, come descritto dall'ISSUE Phase:

1. L'istruzione successiva alla ADD (SUB) viene inviata alla stazione di prenotazione corrispondente.
2. Se ci sono degli operandi già disponibili nel register file, questi vengono inoltrati alla entry della SUB attraverso l'operand bus (sia F1 che F2 sono disponibili).

Da notare che l'entry #0 del banco dei registri registro F0 che contiene l'identificativo della RS che dovrà scrivere il dato su di esso, è cambiato. Non è più la MUL#1 ma è la ADD#2. Questo significa che il

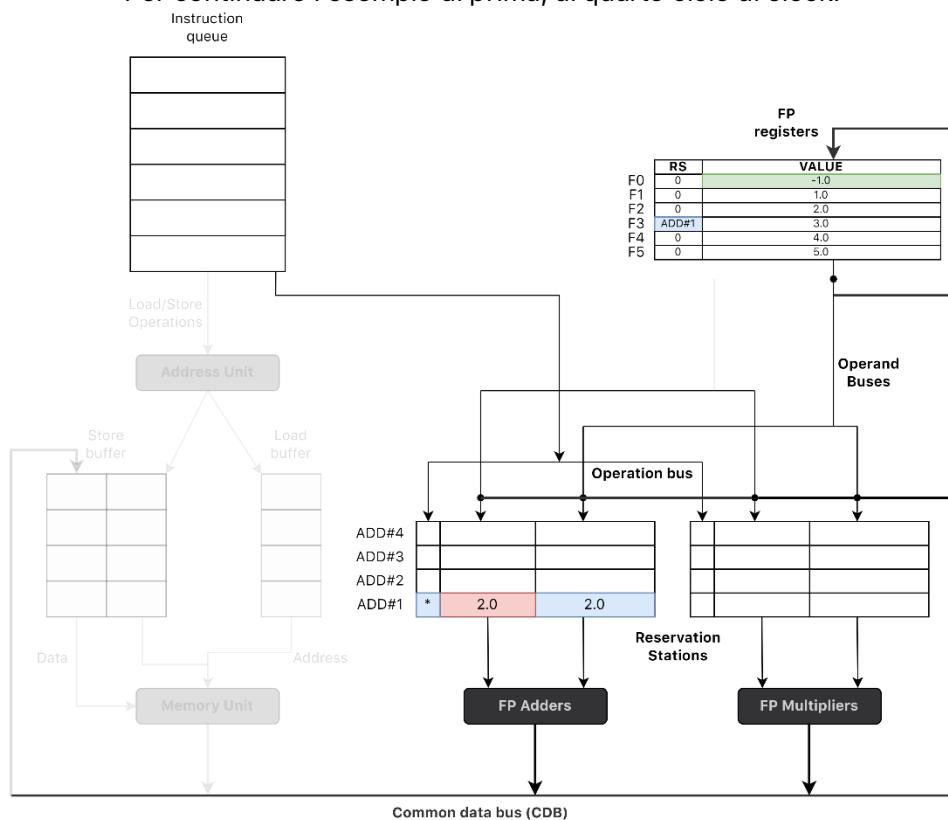
risultato della MUL non sarà mai scritto nel banco dei registri. In sostanza è l'ultima istruzione in ordine sequenziale ad avere il compito di scrivere il risultato sul registro. Questo è come Tomasulo rimuove le Output Dependencies, e quindi evita i WAW hazard, in quanto, nel caso in cui la SUB finisse prima la sua esecuzione della MUL, non dovrà aspettare la MUL per scrivere sul registro F0.

## EXECUTE phase

La fase di EXECUTE consente all'algoritmo di Tomasulo di gestire l'esecuzione delle istruzioni in parallelo, riducendo i tempi di attesa causati dalle dipendenze tra le istruzioni. Inoltre, l'uso del Common Data Bus (CDB) consente di distribuire il risultato dell'esecuzione a tutte le istruzioni che lo stanno aspettando, in dettaglio:

1. La logica di controllo verifica se tutti gli operandi necessari per l'istruzione sono disponibili. Se tutti gli operandi sono disponibili, l'istruzione può essere eseguita.
2. L'istruzione viene inviata all'unità funzionale associata per l'esecuzione. Ogni unità funzionale ha un certo numero di cicli di clock necessari per completare l'esecuzione di un'istruzione, quindi la durata della fase di esecuzione dipende dal tipo di istruzione e dall'unità funzionale che la esegue.
  - a. Se più istruzioni sono contemporaneamente pronte ad una determinata unità funzionale vengono eseguite una dopo l'altra in pipeline
3. Una volta completata l'esecuzione, il risultato viene scritto sul Common Data Bus (CDB) e da lì inoltrato al register file e alle entry delle stazioni di prenotazione che stanno attendendo quel risultato.
4. L'entry della stazione di prenotazione associata all'istruzione diventa disponibile per un'altra istruzione (Busy = 0).

Per continuare l'esempio di prima, al quarto ciclo di clock:



Quarto ciclo di clock, come descritto dall'EXECUTE Phase:

1. La logica di controllo verifica se tutti gli operandi necessari per l'istruzione sono disponibili. Se tutti gli operandi sono disponibili, l'istruzione può essere eseguita (Sia la MUL che la SUB hanno gli operandi pronti)

2. L'istruzione viene inviata all'unità funzionale associata per l'esecuzione. Ogni unità funzionale ha un certo numero di cicli di clock necessari per completare l'esecuzione di un'istruzione, quindi la durata della fase di esecuzione dipende dal tipo di istruzione e dall'unità funzionale che la esegue (la MUL viene inviata all'FP multiplier mentre la SUB all'FP Adders).
3. Una volta completata l'esecuzione, il risultato viene scritto sul Common Data Bus (CDB) e da lì inoltrato al register file e alle entry delle stazioni di prenotazione che stanno attendendo quel risultato (La MUL invia sul CDB il valore 2.0 che viene preso in considerazione dalla entry ADD#1, a cui mancava l'operando mentre la SUB usa il CDB per scrivere il valore sul registro)
4. L'entry della stazione di prenotazione associata all'istruzione diventa disponibile per un'altra istruzione (Sia la entry MUL#1 che ADD#2 vengono liberate e settate a zero).

Per preservare il comportamento delle eccezioni, nessuna istruzione può iniziare l'Execute phase finché non è stato completato un ramo che precede l'istruzione nell'ordine del programma. In un processore che utilizza la branch prediction (come fanno tutti i processori schedulati dinamicamente), ciò significa che il processore deve sapere che la previsione del ramo era corretta prima di consentire l'inizio dell'esecuzione di un'istruzione successiva al ramo. La speculazione fornisce un metodo più flessibile e più completo per gestire le eccezioni.

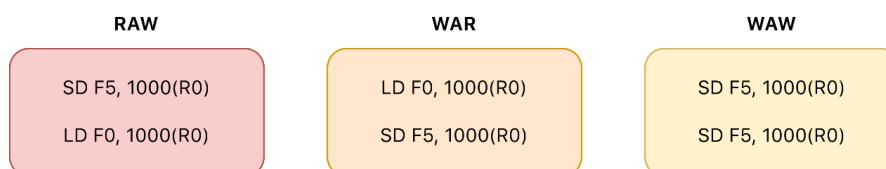
### Features dell'algoritmo di Tomasulo

Le feature che introduce l'algoritmo di Tomasulo sono quindi le seguenti:

- L'algoritmo implementa in modo implicito, attraverso le reservation station (RS), il renaming dei registri
  - Eliminando le dipendenze sui nomi (quindi WAR e WAW)
- Distribuzione più efficiente degli operandi.
  - In un singolo ciclo di clock si inoltra a tutte le unità funzionali che lo richiedono, l'operando appena calcolato attraverso il CDB.
- Bypassing/Forwarding
  - Attraverso il CDB, un risultato viene inoltrato direttamente dall'unità di esecuzione a tutte le RS che lo attendono
- Le istruzioni vengono inoltrate alle stazioni In-order
- Mentre l'esecuzione può avvenire out-of-order appena disponibili gli operandi
- L'esecuzione out-of-order implica che anche le scritture sui registri sono out-of-order. Questo implica che l'algoritmo di Tomasulo non supporta una gestione precisa delle eccezioni hardware.

Infine anche l'ultima istruzione, la ADD#1 scriverà il risultato sul CDB che verrà recapitato anche dal register file e quindi verrà salvato il valore della computazione nel registro F3.

le istruzioni di LOAD e STORE invece, vengono trattate in modo diverso durante le due fasi. Questo perché gli operandi, in questo contesto, non sono disponibili finchè l'indirizzo effettivo non è stato calcolato: Load e Store possono essere eseguiti out-of-order, ma solo nel caso in cui non scrivano/leggano nello stesso indirizzo di memoria. Se questo accade, si verifica un Hazard, ad esempio:

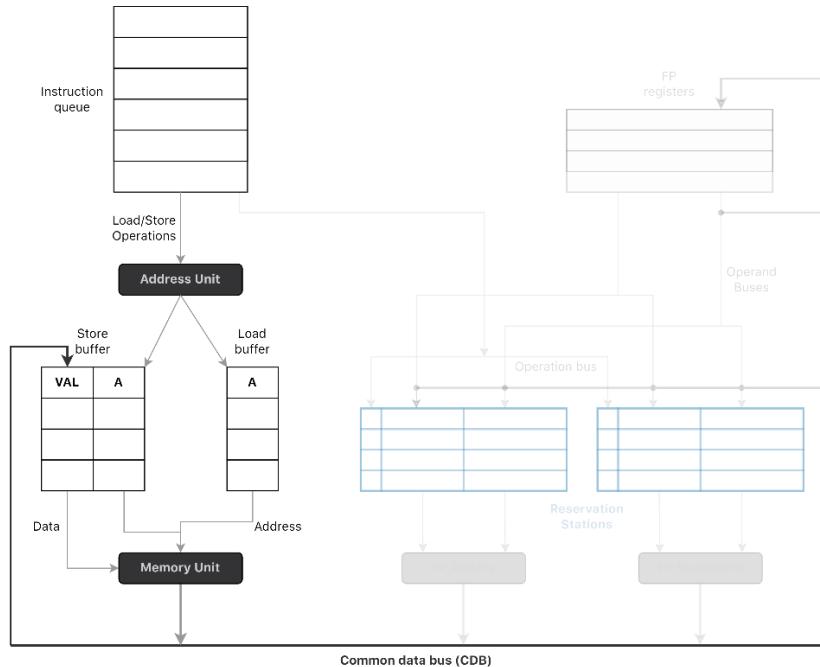


Il problema è che nella realtà non si legge e scrive utilizzando lo stesso registro, tipo:

### RAW or not RAW?

SD F5, 1000(R0)  
 LD F0, 1000(R1)

Quindi finché non si è calcolato il valore effettivo dell'indirizzo, non si può sapere se si è verificato un Hazard. Inoltre, è necessario che le istruzioni prima della LD (in questo caso solo la SD) abbiano già calcolato l'indirizzo effettivo per sapere se la LD causa Hazard. Schema che gestisce LOAD/STORE:



Lo schema che gestisce LOAD e STORE si comporta in questo modo: le istruzioni che lasciano la instruction queue identificate come LOAD e STORE vengono inoltrate all'Address Unit. L'Address Unit calcola l'indirizzo effettivo, verifica che l'istruzione corrente non provochi Hazard e la inserisce nel buffer corretto. Quando un valore caricato da una LOAD ritorna dalla memoria (sul CDB), questo viene inviato a tutte le RS che attendono il dato, ed al banco dei registri.

Entrambe le istruzioni (LOAD e STORE) quindi consistono in due fasi:

1. Calcolo dell'indirizzo effettivo attraverso l'Address Unit
2. E l'accesso alla memoria (Memory Unit)

I buffer contengono due valori: '**A**' che identifica il valore effettivo calcolato e '**VAL**' che (solo per le store) indica il valore da memorizzare. Quest'ultimo può anche essere identificato dall'indice di qualche RS che dovrà produrre il risultato, per questo motivo lo store buffer è collegato al CDB.

Durante la fase di ISSUE per le LOAD:

1. L'Address Unit calcola l'indirizzo di memoria a cui l'istruzione LOAD dovrà operare. L'Address Unit compara l'indirizzo effettivo appena calcolato con il campo '**A**' di tutte le entry dello store buffer.
  - a. Se viene trovata una corrispondenza, si è in presenza di un RAW dato che esiste una STORE che scrive sullo stesso indirizzo e la LOAD non verrà immessa nel buffer finché il conflitto non è completato.
2. Viene memorizzata la LOAD nel LOAD buffer

Durante la fase di ISSUE per le STORE:

1. L'Address Unit calcola l'indirizzo di memoria a cui l'istruzione STORE dovrà operare. L'Address Unit compara l'indirizzo effettivo appena calcolato con il campo 'A' di tutte le entry dello store e del load buffer
  - a. Se viene trovata una corrispondenza nel LOAD buffer si è in presenza di un WAR Hazard dato che esiste una LOAD che legge dallo stesso indirizzo
  - b. Se viene trovata una corrispondenza nello STORE buffer si è in presenza di un WAW Hazard dato che esiste una STORE che legge dallo stesso indirizzo
2. Viene memorizzata la STORE nello STORE buffer

Per le LOAD non viene comparato l'indirizzo effettivo appena calcolato perché non esistono i RAR Hazard.

Durante la fase di EXECUTE:

1. Le LOAD, una volta entrate nel buffer, possono essere eseguite immediatamente prelevando il dato dalla memoria dati e distribuendolo tramite il Common Data Bus (CDB) al registro di destinazione ed alle RS che stanno aspettando il dato.
2. Le STORE, possono dover attendere il valore da depositare in memoria prima di poter essere eseguite. Questo verrà recapitato attraverso il CDB.

Per far funzionare tutto, le entry delle stazioni di prenotazione e dei load/store buffer devono essere composte ognuna da più registri ad accesso veloce in grado di memorizzare le informazioni necessarie a gestire l'intero meccanismo.

## Dynamic Branch Prediction

Nelle architetture di tipo RISC, analizzando varie tipologie di programmi si è visto che, statisticamente, ogni 4-7 istruzioni viene eseguito un branch.

Nel caso di una pipeline a 5 stadi con implementazione aggressiva dei branch, eseguiti in soli 2 cicli di clock, la pipeline va fermata per un ciclo di clock in attesa del risultato del confronto, sprecando in questo modo all'incirca 1/5 dei cicli di clock a disposizione. In molti casi reali le pipeline sono più profonde, il risultato del confronto è noto solo dopo più cicli di clock, e lo spreco è maggiore.

Una delle soluzioni sarebbe, come detto, di adottare una predizione statica:

- Se la predizione è giusta non si sprecano cicli di clock
- In caso di errore, la pipeline va svuotata, mediante opportuni segnali della CU, di tutte le istruzioni nella pipeline successive al branch (potrebbero esserne state prelevate più di una), che non dovevano essere eseguite
  - In caso di errore di predizione lo spreco di cicli di clock è tanto maggiore quanto più è profonda la pipeline, e quanto più il risultato del confronto nel branch viene prodotto ben in profondità nella pipeline.

Purtroppo qualsiasi assunzione di tipo statico sul comportamento dei branch non è sufficiente a garantire un grado di errore abbastanza basso.

Una soluzione più moderna è quella di adottare una **predizione dinamica** e non statica.

La branch prediction si basa sul principio che molte istruzioni di branch seguono un comportamento predecibile e ripetitivo. Ciò significa che se un salto viene eseguito più volte, c'è una buona probabilità che venga eseguito nuovamente nello stesso modo. Ciò si applica in particolare ai cicli, dove il branch che controlla la condizione del ciclo viene eseguito più volte e di solito ha lo stesso comportamento.

Inoltre, per i branch eseguiti una sola volta, non è possibile effettuare alcuna previsione, in questo caso il processore esegue l'istruzione senza utilizzare la tecnologia di branch prediction (if then else).

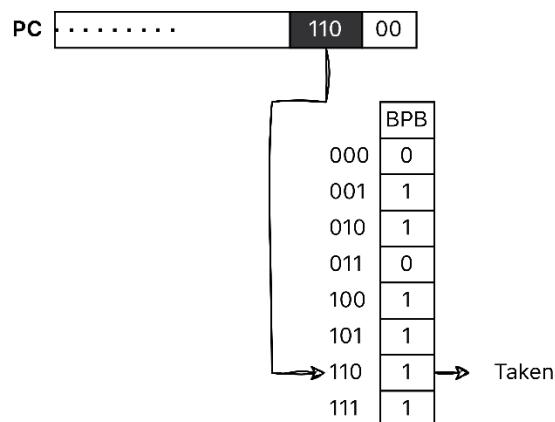
L'idea è quindi quella di Predire il risultato del branch basandosi su dati storici del branch stesso ed iniziando a prelevare (fetching) le istruzioni dal ramo predetto: Se la predizione è corretta non ci sono ripercussioni (branch penalty). Se la predizione è errata, bisogna cestinare le istruzioni precedentemente fetchate ed iniziare a prelevare istruzioni dalla parte opposta del branch. In questo caso c'è una branch penalty più severa.

### 1-bit Branch Prediction Buffer

La forma più semplice di branch prediction utilizza un **branch prediction buffer** a 1 bit. Questa struttura è sostanzialmente una tabella di entry formate da 1 singolo bit in cui il singolo bit indica se la volta precedente in cui è stata eseguita quell'istruzione di branch il salto è stato preso o meno. Le entry sono indirizzate dagli ultimi bit dell'istruzione di branch. Quindi:

- Se l'entry è 1: l'ultima volta in branch è stato preso (Taken - T)
- Se l'entry è 0: l'ultima volta il branch non è stato preso (Not Taken - NT)

Esempio di Branch prediction buffer:

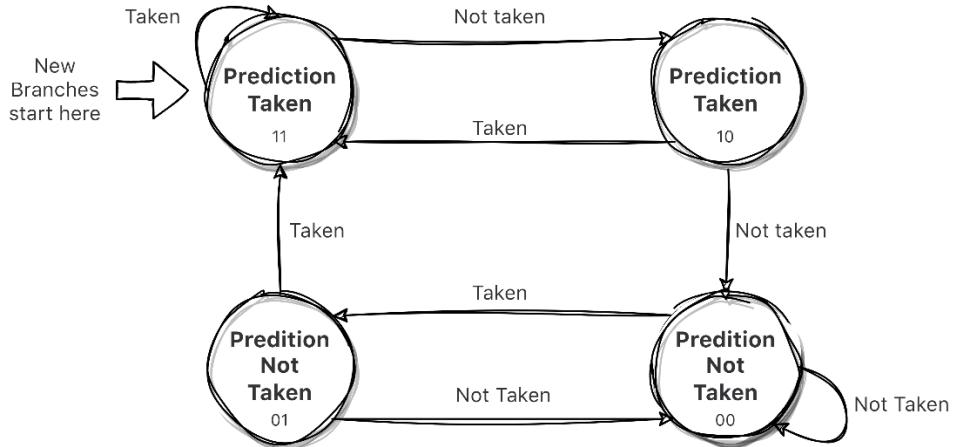


In questo esempio si usano i bit 2,3,4 del PC per indirizzare il BPB. In generale:

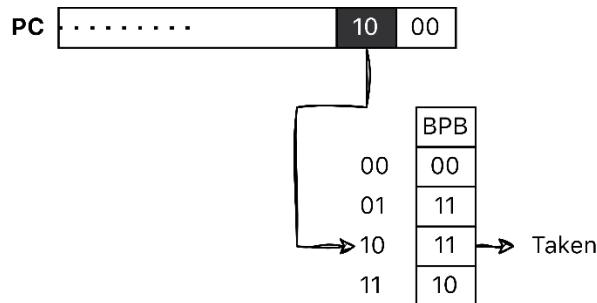
- Quando un processore esegue un salto, cerca prima nel buffer per vedere se ha già eseguito un salto simile in passato.
- Se il buffer contiene informazioni su quel salto, il processore utilizza il bit di predizione per prevedere il risultato del salto e procedere con l'esecuzione delle istruzioni.
- Se la previsione si rivela sbagliata, la pipeline viene svuotata e si ricomincia dall'istruzione a cui salta il branch. Il bit di predizione viene complementato.
- Se il buffer non contiene informazioni su quel salto, il processore esegue la previsione da solo e aggiorna il buffer con il risultato.

C'è un problema però con la BPB ad un singolo bit di predizione. Il problema è il seguente: Si immagini un ciclo che viene eseguito più volte ma poi termina (ad esempio 10 volte). In questo caso, alla fine della decima iterazione ci sarà sicuramente un errore di previsione, poiché il bit di predizione dice di eseguire ancora il ciclo (non è stato ancora complementato), ma il ciclo è terminato. Il bit viene ora complementato. Tuttavia, se il programma dovesse rientrare nel ciclo, la prima previsione sarà di nuovo sbagliata.

Per evitare questi problemi, si aumentano i bit di predizione. Con un numero di bit di predizione pari a due, avremo una sorta di macchina a stati finiti così definita:



In questo schema, una predizione deve essere sbagliata due volte consecutive prima di venire modificata. Ciò significa che se una previsione è sbagliata, non si passa direttamente ad uno stato Not Taken, ma viene tenuto conto del fatto che la previsione è stata sbagliata. Se la stessa previsione viene sbagliata nuovamente, si passa nel dominio Not Taken.



Questo è un esempio di BPB con 2 bit di predizione, che utilizza i bit 2 e 3 per indicizzare il BPB.

Quindi nell'esempio del ciclo che viene eseguito 10 volte, nel caso di 2 bit di BPB, avremmo che, all'inizio del primo ciclo, la predizione è 1 (presa) con stato Strongly taken (11).

Arrivati alla 10 iterazione avremo una missprediction (in quanto il ciclo si interrompe, ma la prediction è ancora Taken con bit 11), quindi avremo un cambio di stato: da 11 a 10 (ma lo stato 10 è ancora nel dominio Taken). Se successivamente si rientra nel ciclo, la predizione indica di saltare, in quanto ci si trova con stato (10), e dato che verrà effettuato un salto effettivo, si passerà dallo stato (10) allo stato (11). Quindi in totale avremo solo un missprediction

## Speculative execution

L'esecuzione speculativa, è una delle altre tecniche utilizzate per migliorare l'efficienza dell'elaborazione parallela a livello di istruzione (ILP).

Man mano che si cerca di sfruttare più parallelismo a livello di istruzione, mantenere le dipendenze dal controllo diventa un onere crescente. La Branch prediction riduce gli stalli diretti attribuibili ai rami, ma per un processore che esegue più istruzioni per clock, questo potrebbe non essere sufficiente per generare la quantità desiderata di parallelismo a livello di istruzione. Un processore di ampia portata potrebbe dover eseguire un ramo ogni ciclo di clock per mantenere le massime prestazioni. Quindi sfruttare più parallelismo richiede di superare la limitazione della dipendenza dal controllo.

Il superamento della dipendenza dal controllo avviene speculando sull'esito dei rami ed eseguendo il programma come se le nostre ipotesi fossero corrette. Questo meccanismo rappresenta un'estensione sottile, ma importante, rispetto alla Branch prediction. In particolare, con la speculazione, recuperiamo,

emettiamo ed eseguiamo istruzioni, come se le nostre previsioni di ramo fossero sempre corrette; la Branch prediction si limita a prelevare ed emettere le istruzioni che seguono la previsione.

Quindi La speculazione hardware combina tre idee chiave:

1. Branch Prediction: per scegliere quali istruzioni prelevare ed emettere nel processore
2. Speculazione: per consentire l'esecuzione delle istruzioni prelevate ed emesse, prima che le dipendenze sul controllo siano risolte (però con la capacità di annullare gli effetti di una speculazione errata)
3. Scheduling dinamico per gestire la schedulazione di diversi blocchi: lo scheduling dinamico senza speculazione riesce a sovrapporre solo parzialmente blocchi perché richiede che un ramo venga risolto prima di eseguire effettivamente qualsiasi istruzione nel blocco successivo

Supponiamo di avere il seguente codice:

```
LD F4, 100(R4) // value not in cache...
BLE F4, #0.66666, jump // branch if less or equal
FADD F1, F1, #0.5
DADD R1, R1, #2
jump: FADD F1, F1, #0.25
 DADD R1, R1, #1
```

Se si suppone che il valore 100(R4) non sia disponibile in nessun livello di cache, è possibile che ci vogliano più di 100 cicli di clock prima che il dato sia disponibile alla pipeline. Anche utilizzando una branch prediction perfetta ed utilizzando lo scheduling dinamico, sarà possibile solamente eseguire il fatch e l'issue delle istruzioni (fino ad un Hazard strutturale dovuto al riempimento delle RS). Questo perché anche se si va avanti con i blocchi del ciclo, questi non potranno essere eseguiti fino a che la BLE non è passata alla fase di execute e quindi calcolato la condizione di salto. La speculazione hardware permette proprio di risolvere gli Hazard sul controllo in quanto le istruzioni verrano si, prelevate ed emesse nelle RS, ma anche eseguite.

Per estendere l'algoritmo di Tomasulo al supporto della speculazione, bisogna separare il momento in cui l'istruzione produce il risultato dall'effettivo completamento (commit) di un'istruzione. Effettuando questa separazione, si consentire ad un'istruzione di essere eseguita finché non si sa che l'istruzione non è più speculativa. Quando un'istruzione non è più speculativa, le si permette di aggiornare il file di registro o la memoria (si effettua il commit).

### Reorder Buffer

Nello schema di Tomasulo, la speculazione hardware prevede l'utilizzo di una unità di commit, nota come Reorder Buffer (ROB), in cui vengono parcheggiate le istruzioni eseguite speculativamente, in attesa di sapere se effettivamente dovevano essere eseguite (commit). Quando quindi la CPU "sa" che una istruzione nel ROB deve essere effettivamente eseguita, ne esegue il commit, togliendola dal ROB e permettendo che il registro di destinazione (o memoria per le store) vengano aggiornate. Se invece la CPU si rende conto che l'istruzione non doveva essere eseguita o eseguita con altri argomenti, semplicemente la rimuove dal ROB, annullando ogni effetto.

Sebbene l'esecuzione delle istruzioni possa avvenire out-of-order, il commit deve avvenire nell'ordine in cui le istruzioni sono entrate nella CPU. Questo vincolo è gestito tramite l'utilizzo del Reorder Buffer (ROB), che è gestito come una coda: ogni istruzione eseguita viene inserita al fondo della coda, e fa scalare di una posizione verso la cima le istruzioni già presenti nel ROB. Questo vincolo diminuisce la quantità di lavoro necessario per il controllo delle dipendenze tra istruzioni e rende più facile la gestione delle eccezioni, che altrimenti sarebbero molto complesse nella speculazione.

Ogni entry del Reorder Buffer (ROB) è composta di 4 campi:

- **type:**
  - branch, che non produce un risultato;
  - store, che scrive in memoria dati;

- ALU o load, che scrivono in un registro.
- **destination**: ossia il numero del registro o l'indirizzo della locazione di memoria modificati dall'istruzione, se questa riuscirà a passare la fase di commit.
- **value**: che memorizza temporaneamente il risultato dell'esecuzione (ovviamente quando questo sarà stato prodotto) dell'istruzione fino al commit.
- **ready**: che indica che l'istruzione ha terminato l'esecuzione (ossia la fase EX) e il contenuto del campo valore è valido.

Questi campi permettono alla CPU di tenere traccia delle istruzioni eseguite speculativamente e dei loro risultati.

Con la speculazione, l'esecuzione di una istruzione viene divisa in **quattro** "macropassi" fondamentali (senza l'esecuzione speculativa erano 3). I primi tre sono gli stessi visti nello schema di Tomasulo senza l'esecuzione speculativa:

### **ISSUE Phase**

- ...
- L'istruzione viene anche inserita in fondo al ROB, facendo scalare verso la cima tutte le istruzioni già presenti nel ROB.
- Se disponibili, gli operandi dell'istruzione vengono inviati alla entry che la contiene, prelevandoli da uno dei registri, da un'altra stazione di prenotazione, o da una entry del ROB
- il numero della entry del ROB che contiene l'istruzione viene scritto nella entry stazione di prenotazione che contiene l'istruzione (così alla fine della fase EX dell'istruzione, il risultato di quest'ultima potrà essere inviato a quella entry del ROB). In sostanza c'è un associazione 1:1 tra la entry del ROB e quella della stazione di prenotazione.

### **EXECUTE Phase:**

- Nessun cambiamento rispetto alla versione senza esecuzione speculativa

### **WRITEBACK Phase**

- ...
- quando il risultato è disponibile, viene scritto sul CDB (con il tag ROB inviato al momento dell'emissione dell'istruzione) e dal CDB al ROB, nonché a tutte le stazioni di prenotazione in attesa di questo risultato.
- Sono richieste azioni speciali per le istruzioni STORE:
  - Se il valore da memorizzare è disponibile, viene scritto nel campo value della entry ROB della STORE.
  - Se il valore da archiviare non è ancora disponibile, il CDB deve essere monitorato fino a quando tale valore non viene trasmesso.

### **COMMIT Phase**

Alcune implementazioni chiamano la fase di Commit anche "graduate" o "complete".

Ci sono tre diverse sequenze di azioni al momento del commit a seconda che l'istruzione di commit sia un branch con una previsione errata, una STORE o qualsiasi altra istruzione (commit normale):

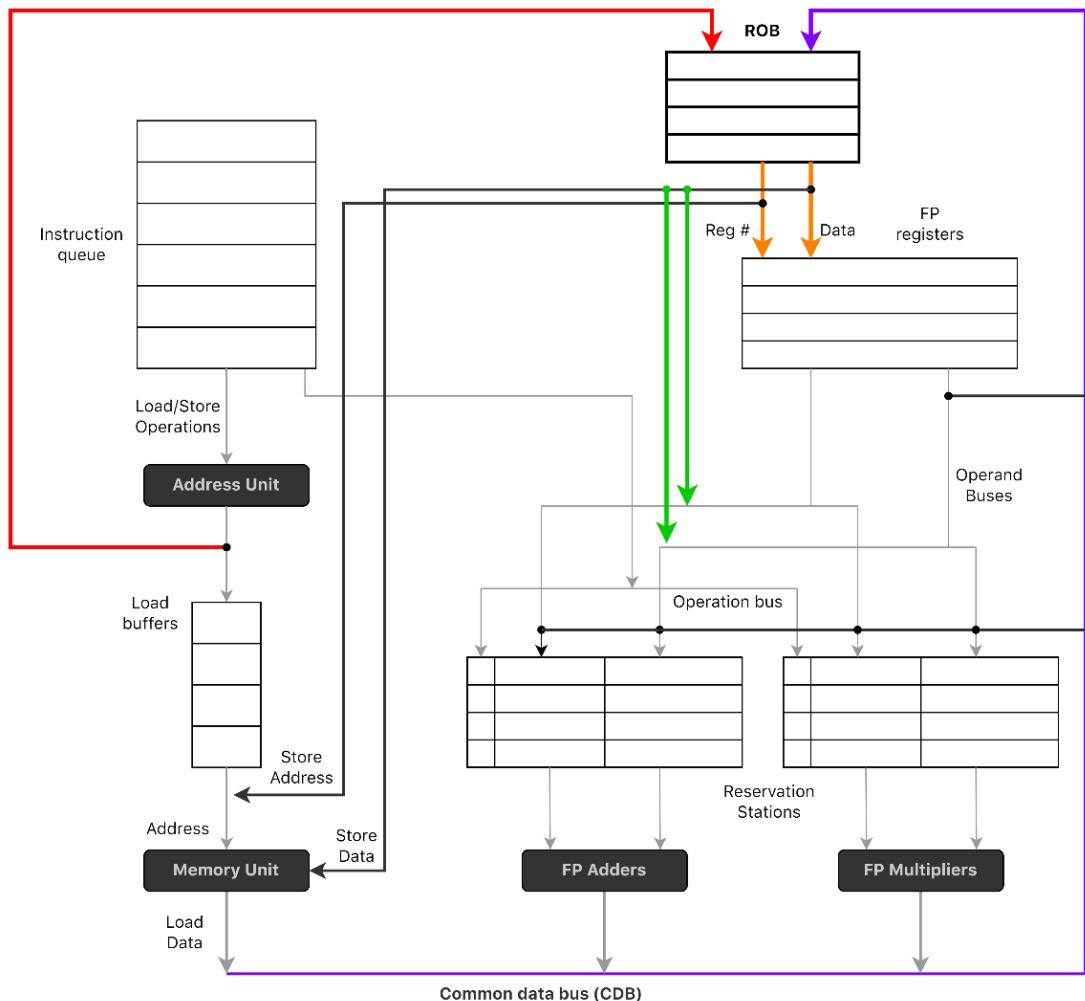
1. Il normale caso di commit si verifica quando un'istruzione raggiunge l'intestazione del ROB e il suo risultato è presente nel buffer. A questo punto il processore aggiorna il registro con il risultato e rimuove l'istruzione dal ROB.
2. Il commit di una STORE è simile, tranne per il fatto viene aggiornata la memoria anziché un registro.
3. Quando un branch con previsione errata raggiunge la testa del ROB, indica che la speculazione era sbagliata, il ROB viene svuotato e l'esecuzione viene riavviata dalla parte opposta del ramo.

Si suppone a questo punto che sappiamo il risultato del branch e che quindi l'esecuzione del branch sia stata completata. Possiamo fare questa assunzione (del fatto che il branch sia già stato eseguito) in

quanto il numero di entry del ROB sono dimensionate in modo tale che una volta che il branch arriva in cima, avrà già passato tutte le fasi della pipeline e sarà già stato eseguito.

Nell'effettiva implementazione della speculazione, nel momento in cui la CU scopre che la predizione relativa ad un branch nel ROB è sbagliata, quel branch viene rimosso dal ROB anche se non si trova in cima alla coda. Vengono rimosse anche le istruzioni che lo seguono (eseguite erroneamente), mentre vengono mantenute quelle prima del branch stesso.

Lo schema di tomasulo con implementato l'esecuzione speculativa:



Rispetto allo schema senza speculazione hardware, abbiamo che:

- Il banco dei registri è scritto dal ROB invece che dal CDB. Questo per garantire che le istruzioni eseguite speculativamente che lasciano l'unità di esecuzione non scrivano direttamente il banco dei registri (ma solo dopo commit)
- Ora, oltre ai registri ed al CDB, anche il ROB contribuisce alla distribuzione degli operandi alle RS. Se gli operandi si trovano sia nei registri che nel ROB, bisogna utilizzare quelli del ROB, in quanto più aggiornati.
- Lo STORE buffer è stato eliminato, ora il suo lavoro lo fa il ROB. Inoltre c'è un collegamento tra l'Address Unit ed il ROB. Questo perché l'Address Unit calcola gli indirizzi effettivi degli STORE, che dovranno poi essere inoltrati al ROB.
- I risultati inoltrati sul CDB dalle RS devono essere inoltrate insieme alla entry del ROB che contiene quell'istruzione

Come l'algoritmo di Tomasulo, bisogna evitare gli Hazard sui dati (oltre a quelli sul controllo). I pericoli WAW e WAR vengono eliminati con la speculazione perché l'effettivo aggiornamento della memoria avviene in ordine. I pericoli RAW attraverso la memoria sono mantenuti da due restrizioni:

1. Non consentire ad una LOAD di accedere alla memoria se una qualsiasi entry del ROB attiva è occupata da una STORE che ha un campo destination che corrisponde al valore del campo A della LOAD
2. Mantenere l'ordine di un indirizzo effettivo di una LOAD rispetto a tutte le STORE precedenti altrimenti si leggerebbero dati obsoleti

Insieme, queste due restrizioni assicurano che qualsiasi LOAD che accede a una posizione di memoria scritta da una STORE precedente non possa eseguire l'accesso alla memoria finché la STORE non ha scritto i dati.

# Multiple Issue

Le tecniche precedenti possono essere utilizzate per eliminare i gli Hazard principali, controllare gli stalli e ottenere un IPC ideale pari ad uno. Per migliorare ulteriormente le prestazioni, vogliamo ridurre l'IPC a meno di uno, ma l'IPC ideale non può essere ridotto al di sotto di uno se si emette solo un'istruzione ogni ciclo di clock (single-issue). Per rompere questa barriera ed avere un  $IPC < 1$ , è quello di emettere più istruzioni (multiple-issue) ad ogni ciclo di clock.

Ci sono diversi approcci per implementare il Multiple-Issue:

- Dynamically scheduled superscalar
- VLIW Processor (Very long instruction word)
- Statically scheduled superscalar

I due tipi di processori superscalari emettono un numero variabile di istruzioni per ciclo di clock e utilizzano: l'esecuzione in ordine, se sono schedulati staticamente o l'esecuzione fuori ordine, se sono schedulati dinamicamente. I processori VLIW, al contrario, emettono un numero fisso di istruzioni formattate come un'unica grande istruzione (da qui VLIW) o come un pacchetto di istruzioni fisso con il parallelismo pianificato staticamente dal compilatore. Quando Intel e HP crearono l'architettura IA-64, introdussero anche il nome EPIC (explicitly parallel instruction computer) per questo stile architettonico.

## Superscalar Processors

Una delle challenge che bisogna superare per implementare un processore superscalare, è quella di riuscire a prelevare (fetching) più istruzioni per ciclo di clock dalla instruction memory: questo è molto complesso in quanto il flusso delle istruzioni prelevate può contenere branches, questo implica che istruzioni successive nel flusso di istruzioni prelevato dalla IM, non sono necessariamente consecutive in memoria. Ad ogni modo, isolando solo il problema al "prelevare più istruzioni" è possibile aumentando la bandwidth della fase di IF.

La challenge però più importante è quella di riuscire ad emettere (issue) più istruzioni per ciclo di clock, dalla IM verso più RS. Questo per il semplice motivo che più istruzioni possono dipendere l'una dall'altra. Ci sono due approcci per risolvere questa challenge:

1. Pipeline issue logic: Questo permetterebbe di eseguire ogni issue in una frazione di clock (se sono due istruzioni prelevate, nella prima metà effettuare la prima issue, e nella seconda metà la seconda). Questo però non scala bene, in quanto può essere fatto solo per un multiple-issue pari a 2.
2. Widen issue logic: in questo approccio, 2 o più istruzioni possono essere gestite in parallelo incrementando la logica stessa della fase di issue.

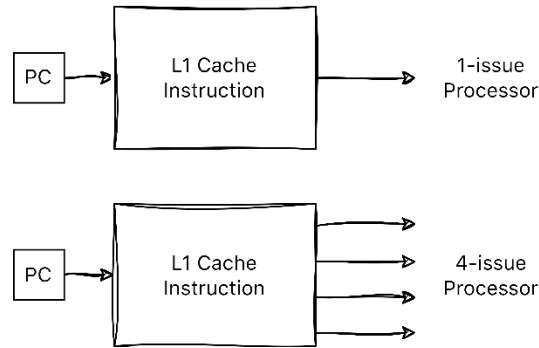
I processor superscalari attuali utilizzano entrambi gli approcci.

Un'altra challenge è quella di aggiungere molti più bus (sia operand che operation) ma anche bus di lettura dal banco dei registri.

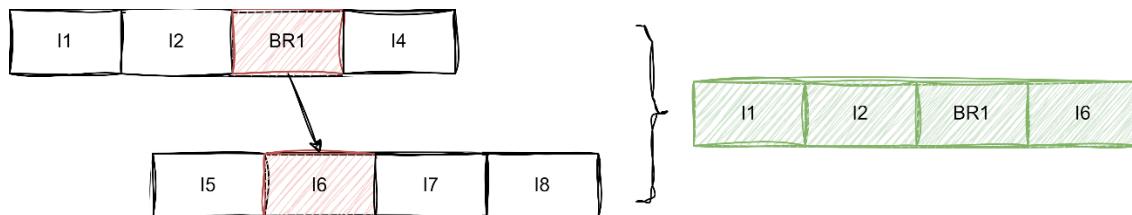
L'ultima challenge è quella di avere più bus per la fase di Write Back, dato che ad ogni ciclo di clock più istruzioni potrebbero scrivere risultati sul CDB. In altre parole, è necessario aumentare la dimensione del CDB. Inoltre tutte le RS devono implementare una logica più avanzata di comparison per capire se i risultati che arrivano dal CDB servono o meno.

## Increasing Instruction Fetch Bandwidth

Ovviamente un processore N-issue deve essere capace di prelevare (fetch) più istruzioni dalla Instruction memory pari ad N ogni ciclo di clock, ed inserirle nella instruction queue.

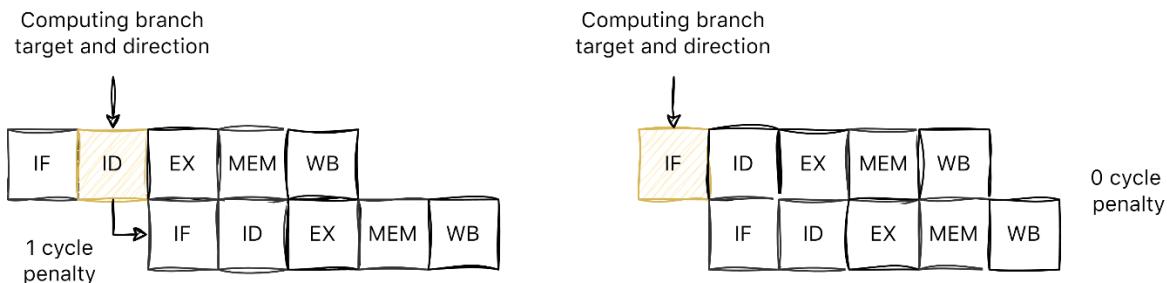


E come detto prima, la challenge più difficile in questa fase, è quella di riuscire a schedulare dinamicamente le istruzioni prelevate in quanto il flusso delle istruzioni prelevate può contenere branches:



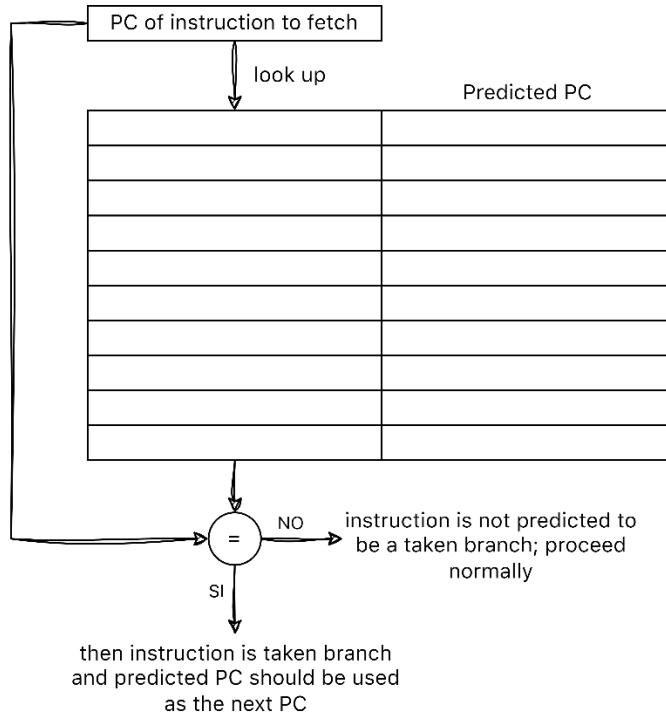
Il primo blocco di 4 istruzioni contiene un branch che salta all'istruzione del secondo blocco di istruzioni prelevate. Questo è un problema, bisogna riordinarle. Ma prelevare due blocchi di istruzioni, confrontarle tra di loro ed effettuare un merge in un singolo ciclo di clock è altamente complesso. Prima di tutto bisognerebbe identificare le istruzioni successive ai branch nella fase di Fetch delle stesse.

Per implementare un meccanismo che identifichi la prossima istruzione in fase di fetch, è necessario migliorare la logica della branch prediction. In principio è necessario spostare la logica della branch prediction nella fase di fetch. Questo perché nella pipeline classica, sia la predizione del ramo e l'effettiva istruzione a cui saltare, sono entrambe calcolate nella fase di ID.



Quindi è necessario avere sia la predizione che il target a cui saltare nello stesso ciclo di clock, nella fase di Fetch. Dopo aver spostato la logica di predizione (branch prediction) nella fase di Fetch, è necessario introdurre una nuova struttura chiamata **Branch Target Buffer**.

Il Branch target buffer, è una Branch Prediction Table che contiene anche il Target Address di ogni branch:



Il PC dell'istruzione recuperata viene confrontato con un insieme di indirizzi di istruzione memorizzati nella prima colonna; questi rappresentano gli indirizzi dei branch conosciuti. Se il PC corrisponde ad una di queste voci, l'istruzione è un branch Taken ed il secondo campo (predicted PC) contiene la previsione per il PC successivo dopo il branch. Il prossimo fetch inizia immediatamente a quell'indirizzo. Nel look-up sono immagazzinati solo i branch con predizione Taken perché un ramo non preso andrebbe a recuperare l'istruzione sequenziale successiva, come se non fosse un branch.

## Considerazioni

Notare che architetture superscalari non per forza implementano tutte le tecniche di ILP. Può esistere una CPU Superscalare ma che non implementa l'esecuzione speculativa, però sarebbero molto limitati.

Inoltre, per implementare un'architettura superscalare è necessario aumentare la profondità del datapath e aumentare il numero di unità funzionali (Ad esempio, più ALU, più unità di moltiplicazione intera/Floating Point, etc...).

Deve anche essere possibile indirizzare in parallelo più registri della CPU, e deve essere possibile leggere e/o scrivere i registri usati da diverse istruzioni in esecuzione nello stesso ciclo di clock.

Nelle architetture moderne, abbiamo un massimo di 4/5 istruzioni eseguite in parallelo per clock cycle (purchè implementino lo scheduling dinamico): l'eventuale assenza di scheduling dinamico della pipeline limiterebbe fortemente il numero di istruzioni che possono essere effettivamente eseguite in parallelo:

- una istruzione indipendente C che nella pipeline viene immediatamente dopo una coppia di istruzioni fra loro dipendenti A e B subisce comunque lo stall della pipeline causato da A e B

Infatti, per questa ragione, i processori con scheduling statico della pipeline hanno di solito un multiple issue limitato al più a due istruzioni per ciclo di clock, tanto non riuscirebbero a sfruttare un multiple issue più elevato.

Presumibilmente il numero di istruzioni che la CPU riesce ad avviare all'esecuzione è inferiore al numero di istruzioni preleva dalla IM.

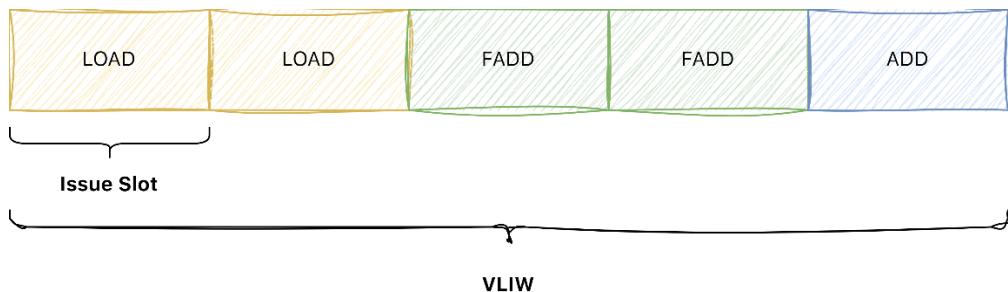
A regime quindi, la IQ tende riempirsi di istruzioni, e se ad un certo ciclo di clock M istruzioni vengono avviate all'esecuzione, al massimo  $M \leq N$  altre istruzioni possono essere prelevate dalla IM al successivo ciclo di clock.

Come caso limite, se la IQ è piena, e a causa delle dipendenze nessuna istruzione è stata inviata alla fase EXECUTE al ciclo precedente, nessuna istruzione potrà essere prelevata dalla IM al ciclo successivo.

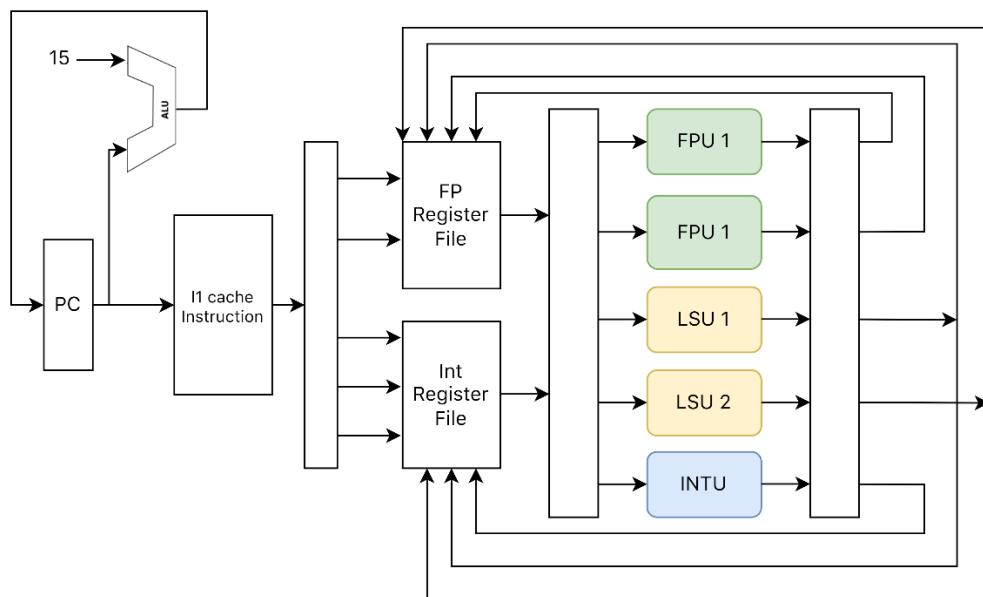
Notiamo anche che se è implementata la speculazione hardware la CPU deve anche essere in grado di eseguire il commit di più istruzioni nel ROB per ciclo di clock, che altrimenti diviene il collo di bottiglia del sistema.

## VLIW Processor

Nell'approccio VLIW si impacchettano le istruzioni indipendenti in un'unica istruzione molto lunga (Very long instruction word). Queste istruzioni vengono impacchettate dal compilatore in una VLIW che specifica un gruppo di operazioni che verranno eseguite in parallelo. Queste istruzioni all'interno della VLIW vengono anche chiamate operation slots. Quando il compilatore non riesce a trovare istruzioni per riempire gli slots, riempie con NOPs. Naturalmente, nel manipolare le istruzioni macchina generate, il compilatore deve lasciare inalterato il funzionamento del programma scritto dal programmatore.



Questa VLIW è composta da due load/store, 2 operazioni Floating point e una intera. Ciò significa che il processore che la elaborerà, dovrà sicuramente avere unità funzionali parallele sufficienti per riuscire ad eseguire tutte e 5 le istruzioni:



Ad esempio se assumiamo che la VLIW è lunga 15 byte, al PC non verrà più sommato 4, ma la lunghezza della VLIW. Comparato allo schema superscalare, nei processori VLIW, si può notare che:

- non c'è supporto hardware per il controllo delle dipendenze, questo perché il lavoro di controllo delle dipendenze è già stato fatto a monte dal Compiler.

- La fase di ISSUE è molto semplice, comparata con lo schema superscalare in cui si utilizza lo schema di tomasulo.
- Anche la fase di Fetch delle istruzioni è semplice, in quanto il Compiler ha impacchettato istruzioni consecutive, rispetto al caso superscalare in cui bisognava effettuare la fase di branch prediction.

C'è anche un utilizzo elevato di bus verso i register files. Ma questo avviene anche nel caso superscalare per permettere l'elaborazione parallela.

### VLIW vantaggi e svantaggi

VLIW utilizza un hardware più semplice rispetto ad un superscalare: richiede meno unità funzionali, meno transistor e quindi il power consumption è ridotto.

D'altro canto, i processori VLIW sono incompatibili a livello binario in quanto anche solo se cambia il numero di issue slots per VILW, numero di Unità funzionali parallele o ancora peggio il CPI delle istruzioni, il binario va ricompilato. Questo è un grosso problema, soprattutto nel mondo general purpose anche perché le persone acquistano e scaricano binari, non sorgenti da compilare.

L'approccio VLIW è particolarmente usato nei processori embedded usati in dispositivi elettronici come ad esempio foto e video camere, e più specificamente i dispositivi alimentati a batteria. Questi dispositivi devono consumare poco, e quindi non si possono permettere l'enorme numero di transistors necessari per implementare tutte le tecniche superscalari.

Inoltre è necessario un efficace supporto da parte di compilatori.

Proprio perché queste tecniche cercano di evidenziare il parallelismo presente nelle istruzioni di un programma prima che questo vada in esecuzione, tali tecniche vengono raggruppate sotto il nome di ILP statico.

Idea di base dell'ILP statico è: tenere il più possibile occupata la pipeline cercando, nella fase di compilazione del programma, sequenze di istruzioni fra loro indipendenti che possano essere eseguite nella pipeline senza generare stall. **Ma in un programma molte istruzioni dipendono le une dalle altre.** Dunque se una istruzione B dipende da una istruzione A, allora B deve essere separata da A di una "distanza" in cicli di clock pari almeno al numero di cicli di clock necessari ad A per produrre il risultato che dovrà essere usato da B.

Naturalmente, i cicli di clock che separano A e B non vanno sprecati, e il compilatore cerca di inserire tra A e B altre istruzioni, utilizzando sempre lo stesso criterio adottato per A e B.

Ma per sapere di quanti cicli di clock separare due istruzioni dipendenti A e B, il compilatore deve sapere esattamente come è fatta la CPU su cui A e B verranno eseguite: quanto è lunga la sua pipeline, e quanti cicli di clock sono necessari per eseguire ciascuna istruzione.

Inoltre, il confine tra ILP statico e dinamico non è netto: in generale le architetture moderne usano una qualche combinazione dei due gruppi di tecniche.

### Loop Unrolling

Un altro esempio di tecnica di ILP statico è il loop unrolling.

prendiamo ad esempio un ciclo che somma i valori di un vettore di lunghezza n. In una implementazione standard del ciclo, l'istruzione di somma verrebbe eseguita n volte, una per ogni elemento del vettore, e l'istruzione di controllo del ciclo verrebbe eseguita n+1 volte. In un'implementazione con srotolamento dinamico, il corpo del ciclo verrebbe ripetuto più volte, ad esempio 4, e l'istruzione di somma verrebbe eseguita 4 volte per ogni iterazione. Ciò ridurrebbe il numero di volte che l'istruzione di controllo del ciclo deve essere eseguita, aumentando così l'efficienza del processore.

```
Loop standard
add $s0, $zero, $zero # inizializza la somma a 0
```

```

loop:
 lw $t0, 0($s1) # carica un elemento del vettore in $t0
 add $s0, $s0, $t0 # somma l'elemento al totale
 addi $s1, $s1, 4 # incrementa l'indice del vettore
 addi $s2, $s2, -1 # decrementa il contatore del ciclo
 bne $s2, $zero, loop

Loop con srotolamento dinamico
add $s0, $zero, $zero # inizializza la somma a 0
loop:
 lw $t0, 0($s1) # carica un elemento del vettore in $t0
 lw $t1, 4($s1) # carica un secondo elemento del vettore in $t1
 lw $t2, 8($s1) # carica un terzo elemento del vettore in $t2
 lw $t3, 12($s1) # carica un quarto elemento del vettore in $t3
 add $s0, $s0, $t0 # somma gli elementi al totale
 add $s0, $s0, $t1
 add $s0, $s0, $t2
 add $s0, $s0, $t3
 addi $s1, $s1, 16 # incrementa l'indice del vettore di 4 volte
 addi $s2, $s2, -4 # decrementa il contatore del ciclo di 4
 bne $s2, $zero, loop

```

Lo srotolamento dinamico, in questo caso, permette di eseguire 4 operazioni di somma per ogni iterazione del ciclo, riducendo il numero di volte che l'istruzione di controllo del ciclo deve essere eseguita.

Da notare che LOAD e STORE appartenenti a iterazioni successive possono essere eseguite in ordine qualsiasi se accedono a indirizzi diversi,ma se usano lo stesso indirizzo si verifica:

- Una antidipendenza, se la STORE viene dopo la LOAD ma le due istruzioni vengono eseguite in ordine inverso
- Una dipendenza sui dati, se la LOAD viene dopo la STORE ma le due istruzioni vengono eseguite in ordine inverso.
- Dipendenza in output: se due STORE vengono scambiate

Per determinare se una LOAD può essere eseguita, la Control Unit deve controllare se una STORE non ancora completata e che precede la LOAD indirizza la stessa locazione in RAM (possibile true data dependence - Read after Write - RAW)

Analogamente una STORE deve attendere fino a che non ci sono altre STORE o LOAD precedenti e non ancora completate che indirizzano la stessa locazione in RAM (possibile dipendenza in output e antidipendenza)

Per individuare i potenziali problemi, la CPU deve aver calcolato l'indirizzo in RAM associato ad ogni LOAD e STORE precedente a quella in corso.

Il funzionamento è il seguente:

- Per le LOAD: quando il calcolo dell'indirizzo usato da una LOAD è stato completato, la CU esamina il campo A di tutti gli STORE buffer attivi per individuare eventuali conflitti di indirizzi.
  - Se l'indirizzo di una LOAD è uguale all'indirizzo di una qualche entry attiva in uno STORE buffer, la LOAD non viene inviata al LOAD buffer fino a che il conflitto non scompare (ossia, la STORE relativa non è terminata).
- Per le STORE: come la LOAD, ma la CU controlla sia gli STORE che i LOAD buffer attivi, perché le STORE che usano un dato indirizzo non possono essere riordinate rispetto ad altre STORE o LOAD attualmente attive (ossia non ancora completate).

# Cache Memory

La cache è una tecnologia sviluppata per migliorare le prestazioni dei computer. In particolare, la gestione della memoria rappresenta uno dei punti cruciali per garantire l'efficienza delle operazioni eseguite dalla CPU.

Fino ad ora, utilizzando la pipeline a 5 stadi, nelle fasi di accesso alla memoria (sia dati che istruzioni) si è assunto che impiegasse 1 ciclo di clock. Nella realtà questo è assolutamente irrealistico: La memoria (DRAM) ha dei tempi di accesso molto più alti.

La memoria principale è stata sempre più lenta della CPU: Fin dai primi computer sperimentali degli anni '40 e '50, lo sviluppo di un modo per memorizzare e rendere velocemente disponibili al processore le istruzioni e i dati del programma in esecuzione ha rappresentato una difficile e fondamentale sfida tecnologica. Inoltre, il gap di prestazioni si è ampliato nel tempo.

Inoltre, il flusso di dati e istruzioni tra CPU e RAM deve passare attraverso il bus, il quale può causare ulteriori ritardi.

Una prima soluzione sarebbe quella di utilizzare una SRAM anziché una DRAM, grande Gigabyte ad accesso veloce, così da risolvere il problema. Ma non è così semplice, infatti se osserviamo questa tabella:

| Tecnologia di memoria | Tempo di accesso      | Costo in dollari per Gbyte nel 2012 |
|-----------------------|-----------------------|-------------------------------------|
| SRAM                  | 0.5 ÷ 2.5 ns          | 500 ÷ 1000                          |
| DRAM                  | 50 ÷ 70 ns            | 10 ÷ 20                             |
| Flash Memory          | 5000 ÷ 50000 ns       | 0.75 ÷ 1                            |
| Dischi Magnetici      | 5000000 ÷ 20000000 ns | 0.05 ÷ 0.1                          |

Per costruire una memoria SRAM di 1GB ci partirebbero dai 500 ai 1000\$, inoltre, le memorie costruite con tecnologia SRAM, sono si più veloci ma occupano più spazio.

Il secondo approccio è quello di utilizzare una gerarchia di memorie (memory hierarchy). In particolare, utilizzare gerarchie di Cache SRAM all'interno dello stesso chip (CPU), che contengano copie di dati ed istruzioni contenuti in memoria principale più frequentemente utilizzati. In questo modo, la CPU può accedere rapidamente alla cache piuttosto che alla memoria principale per eseguire le operazioni richieste.

## SRAM e DRAM

La DRAM (Dynamic Random access memory) è la tecnologia di memoria più comune utilizzata come memoria principale nei computer.

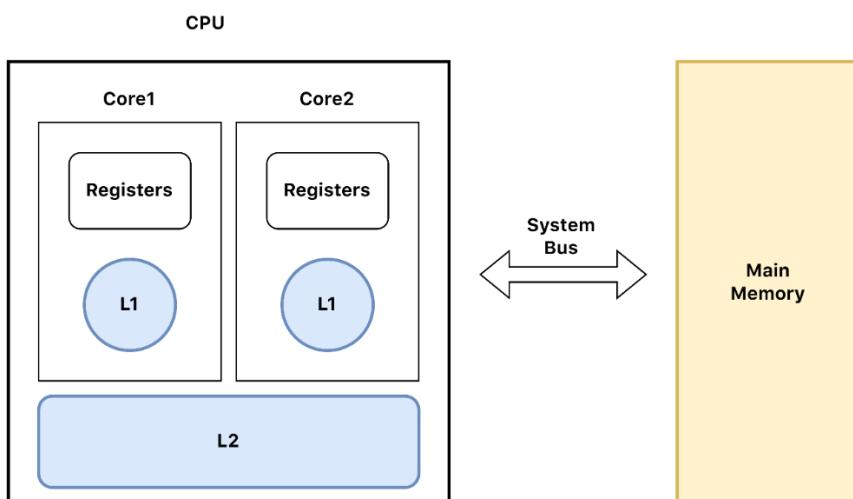
È costituita da celle di memoria composte da un condensatore e un transistor. Il condensatore memorizza l'informazione sotto forma di carica elettrica, mentre il transistor funge da interruttore per controllare l'accesso alla cella di memoria. Tuttavia, a causa della natura dinamica della tecnologia DRAM, la carica elettrica deve essere costantemente "refreshata" per mantenere l'informazione memorizzata, il che richiede un consumo energetico significativo e aumenta la latenza di accesso ai dati.

La SRAM, d'altra parte, utilizza una tecnologia di memoria statica basata su circuiti logici complessi, anziché su celle di memoria. In pratica, la SRAM memorizza l'informazione in una configurazione di transistor e flip-flop (6 transistor per un bit, per questo è più costosa e richiede più spazio),

mantenendo lo stato anche senza refresh periodici. Questo la rende più veloce rispetto alla DRAM e con una latenza di accesso molto bassa, tuttavia, come detto, è più costosa e richiede più spazio fisico rispetto alla DRAM per la stessa quantità di memoria.

## Cache hierarchy

Quindi l'idea per cercare di limitare il problema della lentezza della RAM è di utilizzare una gerarchia di memorie. Già nel 1946 A. Burks, H. Goldstine e J. von Neumann nel "Preliminary Discussion of the logical Design of Electronic Computing Instrument", scrissero che: "Ideally, one would desire an indefinitely large memory capacity such that any particular...word would be immediately available... We are forced to recognize the possibility of constructing a hierarchy of memories, each which has greater capacity than the preceding but which is less quickly accessible".



Infatti, come scrissero nel 1946 A. Burks, H. Goldstine e J. von Neumann, anche i moderni processori utilizzano (in gerarchia) solitamente tre livelli di cache, denominati L1, L2 e L3:

- La cache di primo livello (L1) è la cache più vicina alla CPU ed è integrata direttamente nel processore. Quello che era la Instruction Memory e Data memory in realtà era la Cache di primo livello. La cache di primo livello (L1) è infatti divisa in due parti distinte:
  - la cache per dati (data cache): cache per dati memorizza i dati usati frequentemente dalla CPU
  - la cache per istruzioni (instruction cache): cache per istruzioni memorizza le istruzioni del programma in esecuzione.

In questo modo, in un datapath pipelined, nello stesso ciclo di clock è possibile:

- prelevare una istruzione dalla cache di istruzioni (quella che fino ad ora avevamo chiamato la Instruction Memory)
- leggere/scrivere dei dati (relativi ad un'altra istruzione) dalla cache di dati (quella che chiamavamo la Data Memory).
- La cache di secondo livello (L2) è posizionata tra la CPU e la memoria principale. La L2 ha una capacità maggiore rispetto alla L1 e ha un tempo di accesso leggermente più lento.
- La cache di terzo livello (L3) è opzionale e può essere presente in alcuni processori. Se presente, si trova al di fuori della CPU ed è condivisa tra tutti i core del processore. La L3 ha una capacità ancora maggiore rispetto alla L2, ma ha anche un tempo di accesso più lento.

In alcuni casi, possono esistere anche cache di livello superiore, come la L4, tuttavia queste sono meno comuni e solitamente si trovano solo in sistemi server o workstation di fascia alta.

Le diverse cache possono essere costruite con tecnologie leggermente diverse per ottimizzare le prestazioni in base al loro utilizzo:

1. La cache di primo livello (L1) utilizza solitamente SRAM ad alta velocità, con tempi di accesso molto bassi.
2. Le cache di livello successivo, come la L2 e la L3, utilizzano anche SRAM, ma sono ottimizzate per la capacità di memorizzazione piuttosto che per la velocità di accesso. Questo significa che le cache di livello successivo sono più grandi ma leggermente più lente rispetto alla L1.

Inoltre, la posizione fisica della cache rispetto alla CPU influisce anche sulle prestazioni. Poiché la L1 è integrata direttamente nel processore, ha un tempo di accesso molto veloce rispetto alla L2 e alla L3, che sono posizionate più lontano dalla CPU.

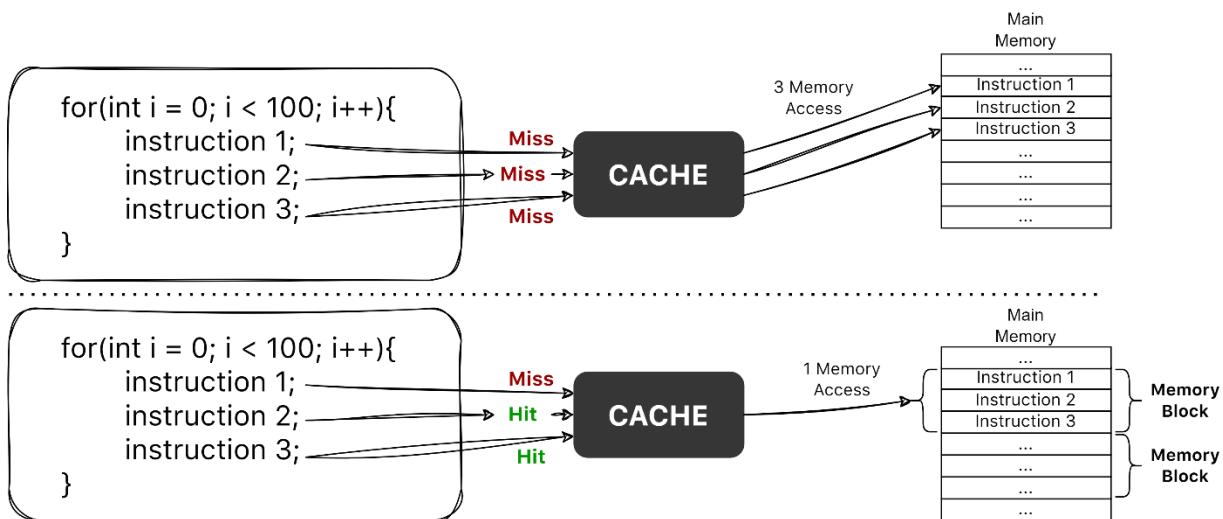
## Principio di Località

Il principio di località (principle of locality or locality of reference) è un concetto fondamentale nell'architettura dei computer e si riferisce alla tendenza dei programmi a riferirsi frequentemente ad una porzione (limitata) della memoria in un determinato intervallo di tempo.

Il principio di località è composto da due sotto-principi principali:

1. **Località spaziale**: elementi in memoria (istruzioni e dati) che vengono utilizzati in un determinato momento sono spesso vicini tra di loro nello spazio di indirizzamento della memoria. Ad esempio, un programma che sta eseguendo un ciclo su un array di dati, accederà frequentemente a elementi adiacenti dell'array (o istruzioni adiacenti). Ciò significa che quando viene caricato un dato (o istruzione) dalla memoria principale nella cache, è probabile che anche i dati (o istruzioni) adiacenti vengano utilizzati presto e possano quindi essere memorizzati nello stesso blocco di cache.
2. **Località temporale**: elementi in memoria (istruzioni e dati) che vengono utilizzati in un determinato momento è molto probabile che saranno utilizzati di nuovo in un futuro prossimo. Ad esempio, un programma che esegue un ciclo su un array, probabilmente utilizzerà gli stessi dati (o istruzioni) più volte durante l'iterazione del ciclo. Ciò significa che una volta che un dato (o un istruzione) viene caricato nella cache, è probabile che venga utilizzato di nuovo prima che la cache lo sostituisca.

Per giustificare il suo utilizzo, prendiamo come esempio un loop scritto in un linguaggio ad alto livello:



Nella versione più in alto, il principio di località temporale è soddisfatto, in quanto alla seconda iterazione del ciclo, le 3 istruzioni avranno un hit. Il primo esempio però non rispetta la località spaziale, infatti, quello che succede è che ogni accesso alla cache di istruzioni produrrà un miss. Questo perché si recuperano istruzioni (questo esempio istruzioni ma poteva essere anche cache dati) con granularità troppo fine (1). Perché quindi non recuperare dalla memoria un gruppo di elementi (che siano dati o istruzioni) al fine di avere più hit in cache e limitare l'accesso in memoria primaria?

Quello che si fa quindi, nelle architetture moderne, è quello di suddividere la memoria principale in blocchi. In termini di accessi in memoria, questo si traduce in 3 accessi in memoria per il primo scenario, ed 1 accesso in memoria per il secondo scenario (il che, è molto più conveniente). Quindi le cache moderne non immagazzinano un elemento di memoria per linea di cache, ma ne immagazzinano ben di più. Questo soddisfa il principio di località spaziale.

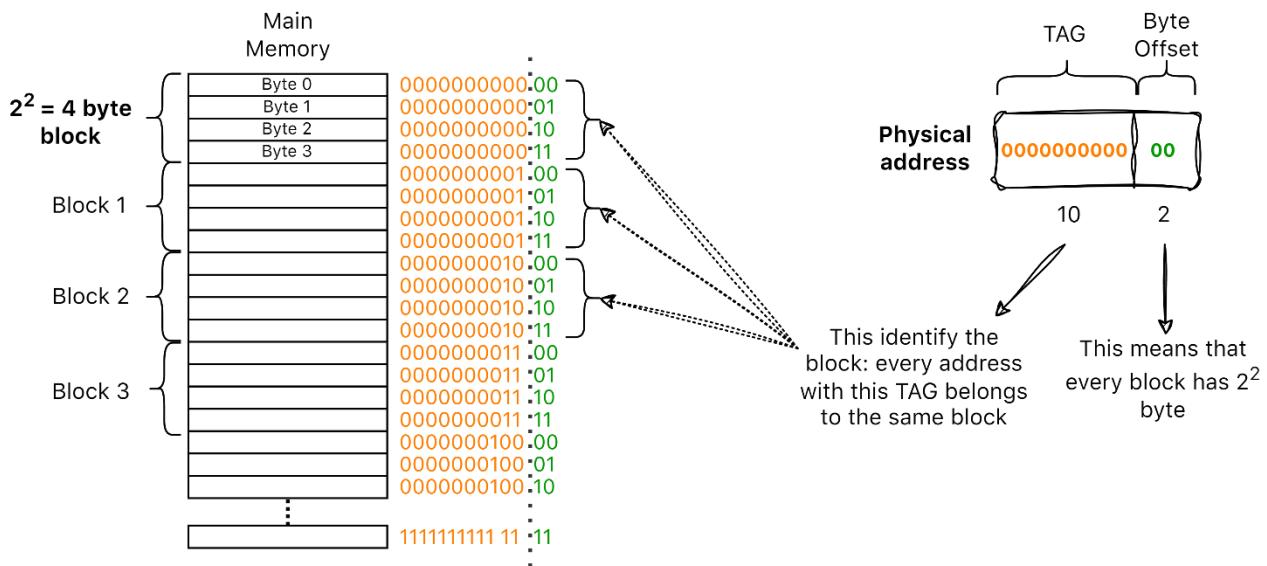
Il principio di località è alla base delle tecniche di caching.

## Fully Associative Cache

Le cache fully associative, non hanno restrizioni sulla posizione dei blocchi di dati nella cache. Ogni blocco di dati può essere memorizzato in qualsiasi posizione disponibile.

Nell fully associative cache, l'indirizzo fisico è scomposto in due sotto parti:

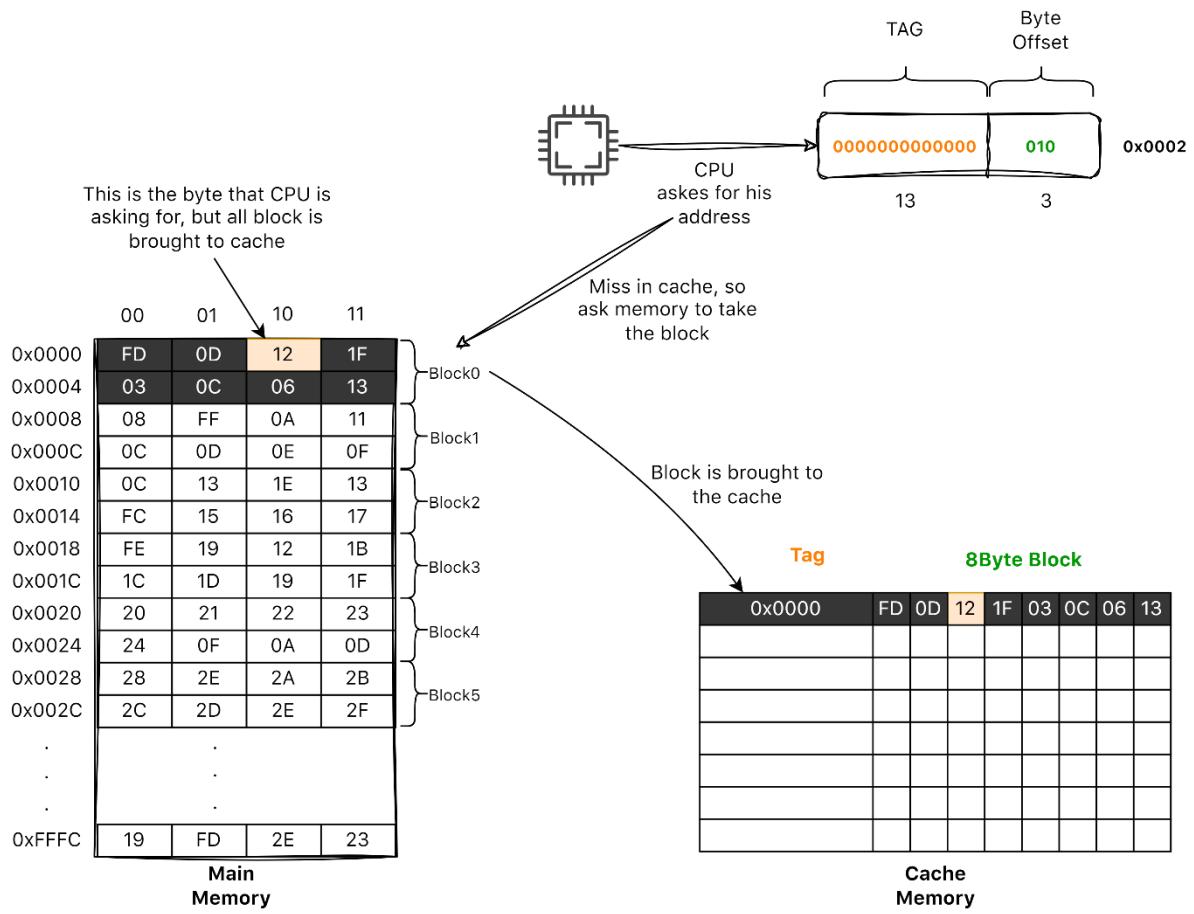
- La parte di TAG che indica a quale blocco quell'indirizzo appartiene
- La parte di Byte Offset (o Word-ID) che indica in che offset all'interno del blocco risiede quel dato



Qui un esempio di una memoria con spazio di indirizzamento fisico  $2^{12} = 4\text{Kbyte}$ . Se si usano 2 bit per il byte offset si suddividerà la memoria in blocchi da 4 byte ciascuno (o viceversa, se si sceglie di suddividere la memoria in blocchi da 4 byte, si utilizzeranno 2 bit di byte offset). Il TAG identifica un blocco, ad esempio, il TAG con 10 zeri identifica sempre e solo il primo blocco (di 4 byte sempre), infatti tutti e 4 i byte che hanno rispettivamente offset (00, 01, 10, 11) appartengono allo stesso blocco.

Quando la CPU effettua un lookup nella cache fully associative, lo fa confrontando in un solo colpo tutte le entry cache. Questo comporta un hardware più costoso e complesso in quanto è necessario implementare un compare instantaneo per tutte le linee di cache.

Quando invece la CPU necessita di un dato che non si trova nella cache (cache miss), recupera non solo il byte richiesto, ma l'intero blocco di cui quel byte fa parte, dalla memoria principale (sempre per il principio di località).



## Replacement Policy

Cosa succede quando la Cache si riempie? È necessario fare spazio per un dato (e di conseguenza blocco) richiesto. Ed è qui che entrano in gioco le Replacement Policy che determina quale blocco di dati presente nella cache viene rimosso per far spazio al nuovo blocco. Le politiche di sostituzione comuni includono l'uso di algoritmi come il Least Recently Used (LRU) o il Random Replacement (sostituzione casuale).

Di algoritmi ne esistono tanti ma i più usati sono quelli basati sull'utilizzo recente (Simple recency-based policies) e l'algoritmo di Bélády. C'è da dire che comunque non esiste un algoritmo perfetto, dipende dall'applicazione (% rapporto b/n lettura e scrittura, quanto spesso si effettuano riscrittire, dimensioni della cache..etc) e cosa si sta cercando di ottimizzare (Potenza/Prestazioni). La scelta perfetta dell'algoritmo di replacement corrisponde ad un minor miss-cache rate.

## Vantaggi e svantaggi

Uno dei vantaggi principali è il fatto che **si eliminano i conflitti**: nelle cache fully associative, ogni blocco di dati può essere memorizzato in qualsiasi posizione disponibile nella cache. Ciò significa che non ci sono restrizioni sulla posizione di archiviazione dei blocchi, eliminando così i conflitti di cache: ogni blocco di dati può essere memorizzato in una posizione unica, senza la necessità di sostituzioni forzate a causa di conflitti di mappatura.

Lo svantaggio di questo approccio è la **complessità hardware**: L'implementazione di una cache fully associative richiede un hardware più complesso rispetto ad altri tipi di cache. È necessario utilizzare circuiti e logiche aggiuntive per effettuare la ricerca di un blocco di dati all'interno dell'intera cache in un colpo solo. Ciò può comportare un aumento dei costi di produzione e un consumo energetico più elevato.

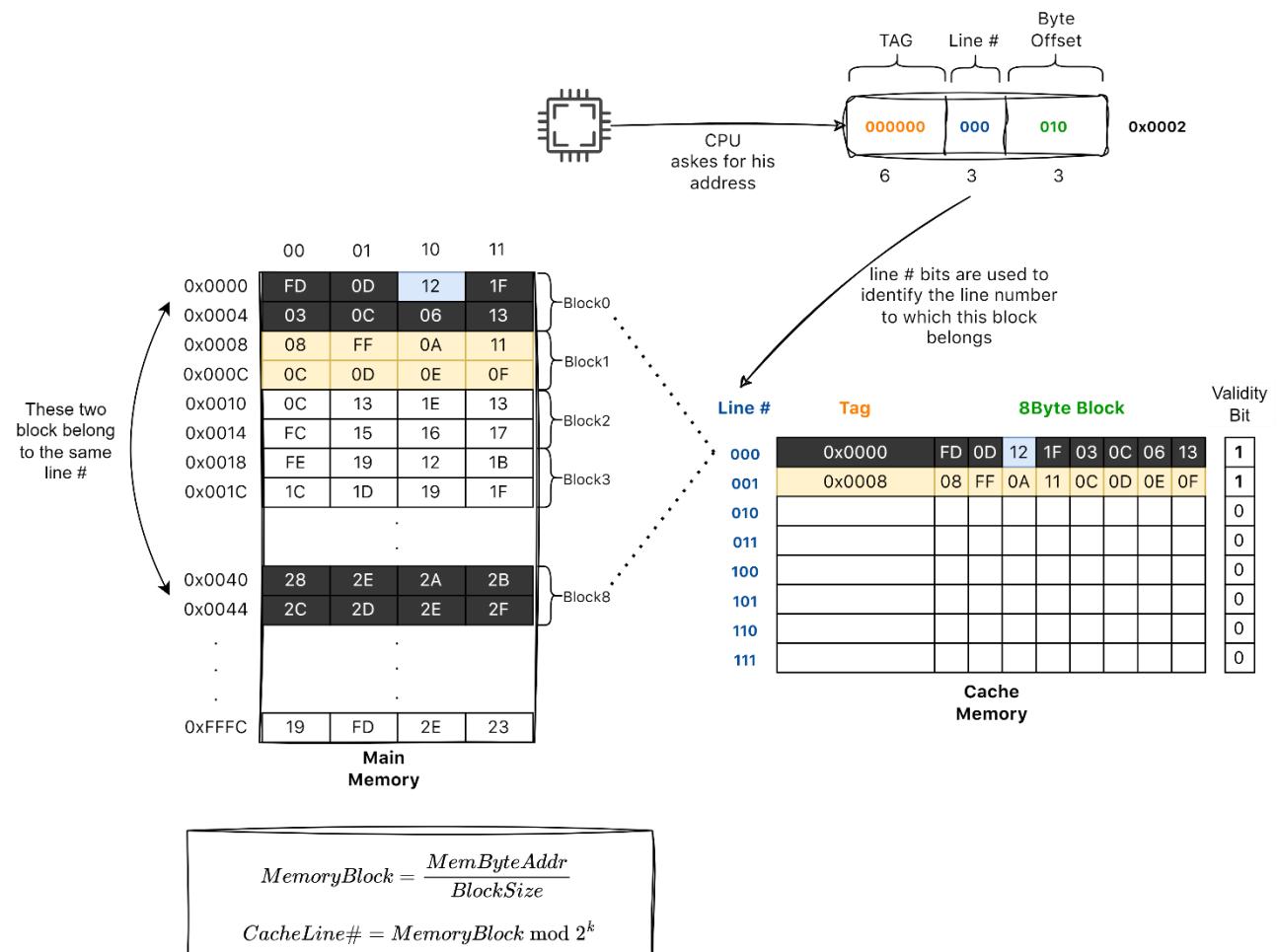
Un altro svantaggio è il **tempo di accesso potenzialmente più lungo**: A causa della complessità dell'hardware di ricerca, i tempi di accesso alle cache fully associative potrebbero essere più lunghi

rispetto ad altri tipi di cache. La necessità di esaminare tutte le posizioni della cache per trovare un blocco di dati richiesto può comportare una maggiore latenza.

## Direct Mapped Cache

Nella fully associative, ogni blocco può essere memorizzato in qualsiasi linea di cache. La Direct Mapped invece, specifica esattamente quale linea di cache immagazzinerà quel blocco.

La prima cosa che differenzia la Direct Mapped con la fully associative è il fatto che ogni linea di cache è numerata.



Ogni blocco di RAM appartiene ad una specifica linea di cache. Ovviamente i blocchi in RAM saranno maggiori delle linee in cache. Questo significa che due blocchi che sono sufficientemente spaziati, andranno mappati nella stessa linea di cache. Infatti ogni  $2^k$  byte in memoria, si ha un conflitto. Come mostra l'esempio, il primo byte (0x0000) crea conflitto con il byte che sta a  $2^6 = 64$  bit posizioni più avanti (0x0040). Oppure più semplicemente ogni  $2^n$  blocchi si ha un conflitto, dove  $2^n$  è il numero di entry della cache, nell'esempio ogni  $2^3 = 8$  blocchi si ha un conflitto.

Una volta immagazzinato il blocco 0 nella linea 000, se la CPU richiederà uno tra i byte che risiedono tra 0x0040 e 0x0044 (che è il blocco che fa conflitto), bisognerà sostituire il blocco precedente in cache con quello nuovo.

Ora, se due blocchi in RAM possono essere mappati nella stessa linea di cache, non è sufficiente utilizzare i bit che identificano la linea per capire se un dato si trova in quella linea (cache hit) oppure no (cache miss). Qui vengono in aiuto i bit del tag. Il tag è necessario per differenziare i vari blocchi che possono essere mappati nella stessa linea di cache.

Un dettaglio che viene aggiunto alle cache table è il **bit di validità**. Questo è semplicemente un bit che indica se il blocco/linea della cache è valida (se è vuota o meno).

Al boot ovviamente tutte le entry avranno il bit a Zero, man mano che le entry vengono fillate, il bit di validità corrispondente verrà settato ad Uno.

### Vantaggi e svantaggi

Un primo vantaggio sta nel fatto che non necessita di un hardware complesso, quindi

**Implementazione semplice**: relativamente facili da implementare rispetto ad altri tipi di cache.

Richiedono meno circuiti e logiche per gestire la mappatura dei blocchi di dati nella cache. Di conseguenza si ha un **costo ridotto**.

Quando un blocco effettua un conflitto, non si applica nessuna politica di sostituzione, il blocco precedente viene eliminato. Questo rende le direct mapped molto **veloci della risoluzione dei conflitti**.

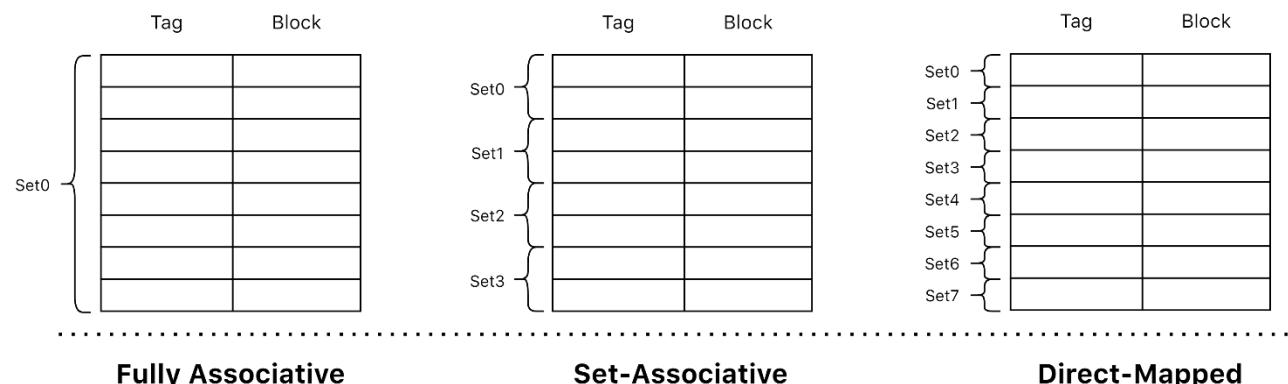
Invece, uno degli svantaggi principali è che aumenta il numero di conflitti, quindi **conflitti di cache elevati**: La mappatura diretta dei blocchi di dati può causare un alto grado di conflitti di cache. Quando diversi blocchi di dati vengono mappati nella stessa posizione, si verifica una cache miss e uno dei blocchi deve essere sostituito, anche se potrebbe essere ancora utilizzato. Se due blocchi finiscono per contestarsi la stessa linea, si va incontro al fenomeno del trashing: blocchi di memoria concorrono per una entry cache, in questo modo il sistema è occupato per il 90% del suo tempo a swappare blocchi di memoria all'interno delle entry cache, causando un collasso delle prestazioni. Come ridurre questo fenomeno? Aumentando semplicemente il numero di linee.

Un altro svantaggio è l'**utilizzo inefficiente dello spazio**: Poiché ogni blocco di dati è mappato in una posizione specifica, potrebbero verificarsi spazi non utilizzati o frammentazioni all'interno della cache. Ciò può portare a un utilizzo inefficiente dello spazio disponibile, riducendo l'efficienza complessiva della cache.

## Set-Associative Cache

Abbiamo visto le Fully associative e Direct Mapped che sono uno l'opposto dell'altro: Fully associative può inserire un blocco ovunque nella cache, non c'è il problema del trashing. Direct Mapped sono più veloci, meno costose, ogni blocco è mappato in una sola linea di cache, è sensibile al trashing.

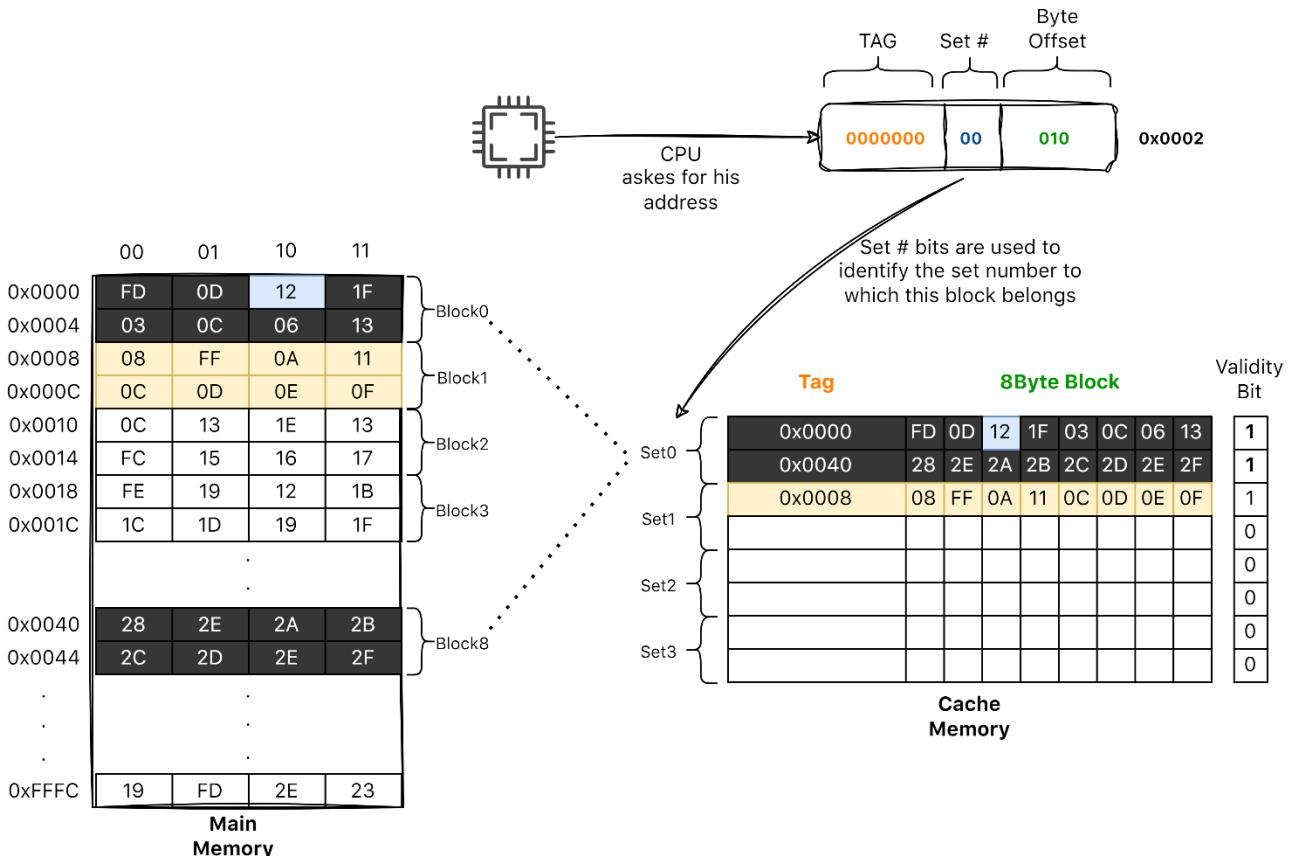
Quello che è un compromesso tra le due tecniche (ed è anche la più usata nelle architetture moderne) è la Set-Associative. In questa tecnica, si divide la cache in Set in k-way, dove k è il numero di linee che vengono raggruppate per Set. Se ad esempio si ha una cache 2-way associative, ogni set avrà 2 linee di cache.



Questa figura mostra come le 3 tecniche possono essere viste sotto forma di raggruppamenti in Set. Nella direct Mapped ogni set ha esattamente una linea di cache, infatti non sono presenti i set ma le linee vengono semplicemente chiamate linee. Nella fully associative si ha un Set che comprende tutte

le linee. E nella set-associative si hanno k-linee (nell'esempio 2-way) per set. In generale si dice che una cache è k-way associative dove k è il numero di entry per set.

Che vantaggi porta il raggruppamento di linee in set? Risolve in maniera efficiente il problema del trashing e quindi due blocchi di memoria che si contestano una linea di cache (problema che si ha nel direct mapped). Questo perché ora, due linee che vengono mappate nello stesso set, hanno k-linee da poter scegliere. Nel caso in cui tutte le entry del set fossero occupate sarà necessario tornare all'utilizzo di Algoritmi di replacement (come nella fully associative).



Ora, il blocco che prima nella Direct-Mapped veniva sostituito (0x0000) per far spazio a quello nuovo (0x0040) non viene più trashed ma rimane in cache ed il blocco nuovo trova spazio nel set.

L'hardware necessario per implementare un tale tipo di cache è più sofisticato, ma dà prestazioni migliori rispetto ad una cache DirectMapped, si riesce a ridurre di qualche punto percentuale la % di cache Miss.

## Cache e Prestazioni

Le prestazioni di una cache dipendono, oltre che dal numero di linee di RAM che è in grado di memorizzare, anche dalla dimensione del blocco per linea. In generale, linee più grandi consentono di diminuire i cache miss, perché sfruttano meglio il principio di località spaziale. Tuttavia, oltre una certa dimensione non conviene andare, perché se le linee sono troppo grandi, una cache di dimensione prefissata ne può contenere poche, e aumenta la probabilità che una linea debba essere rimossa dalla cache (perché si è verificato un cache miss) prima che molti dei dati contenuti in quella linea siano stati usati.

## Gestione delle scritture

La gestione delle scritture è più delicata, perché richiede (prima o poi) la modifica dei dati anche in RAM.

la gestione delle scritture nella cache è più delicata rispetto alla gestione delle letture, poiché ogni scrittura nella cache richiede che il dato venga aggiornato anche nella memoria principale (RAM), in modo da garantire la coerenza dei dati tra cache e RAM.

Esistono diversi modi per gestire le scritture nella cache, ma una delle soluzioni più comuni è il write-through: con questo metodo, ogni volta che viene eseguita una scrittura nella cache, il dato viene immediatamente propagato anche alla RAM, in modo da mantenere i dati coerenti tra le due memorie.

Il write-through è una soluzione efficace per garantire la coerenza dei dati tra la cache e la RAM, ma può comportare una riduzione delle prestazioni in caso di scritture ripetute sulla stessa linea di cache. In questo caso, infatti, ogni scrittura richiede la propagazione dei dati sia alla cache che alla RAM, causando un rallentamento delle prestazioni. Per ovviare a questo problema, è possibile utilizzare altre tecniche di gestione delle scritture nella cache, come ad esempio il write-back o il write-around.

Nello schema write-back i dati vengono modificati solo nella cache e la loro scrittura sulla RAM viene posticipata, fino a quando la linea di cache contenente il dato viene rimpiazzata. In questo modo, le operazioni di scrittura sono più veloci, poiché vengono eseguite solo sulla cache, ma è necessario gestire in modo accurato la coerenza della cache per evitare che i dati presenti nella RAM e nella cache diventino incoerenti. Quando una linea di cache viene rimpiazzata, se i dati in essa contenuti sono stati modificati, questi vengono scritti sulla RAM per garantire la coerenza.

## Miss Penalty

Il miss penalty è il tempo richiesto per recuperare un blocco di dati dalla memoria principale in caso di cache miss. Quando la CPU richiede un blocco di dati che non è presente nella cache, la cache deve recuperare i dati dalla memoria principale, il che richiede un certo tempo, solitamente molto maggiore rispetto al tempo necessario per accedere ai dati presenti nella cache. Questo tempo di recupero dei dati mancanti viene chiamato miss penalty e rappresenta un ritardo nel completamento dell'operazione richiesta dalla CPU.

La durata della penalità da fallimento è dovuta alle tre operazioni fondamentali necessarie per gestire un cache miss:

1. Inviare l'indirizzo della linea mancante alla RAM.
2. Accedere alla RAM per recuperare la linea.
3. Inviare alla cache la linea recuperata.

Il miss penalty può essere influenzato da diversi fattori, come la velocità della memoria principale, la dimensione della cache e l'efficienza dell'algoritmo di sostituzione della cache.

L'operazione più costosa è l'accesso alla RAM, dato che una singola lettura/scrittura può richiedere l'equivalente di 15-20 cicli di clock (del bus di sistema, che è un po' più lento del clock della CPU).

Inoltre, una linea della cache è di solito molto più grande della quantità di dati che un banco di RAM è in grado di fornire ad ogni accesso. Ad esempio, una linea può essere grande 16 byte, mentre con un accesso alla RAM possiamo leggere in tutto una word da 32 bit, ossia 4 byte, e abbiamo quindi bisogno di 4 accessi per recuperare l'intera linea, ossia l'equivalente di 60-80 cicli di clock.

Per contro, l'invio di dati dalla cache alla RAM e viceversa può essere fatto in uno o comunque pochi cicli di clock.

Se ad esempio consideriamo allora un sistema a 32 bit (e quindi con un bus sulla scheda madre a 32 bit). Se supponiamo un tempo di accesso alla RAM di 15 cicli di clock per leggere una word da 32 bit, per servire un cache miss con linee da 16 byte impiegheremo:

- 1 ciclo di clock per inviare l'indirizzo del blocco mancante alla RAM
- $4 \times 15$  cicli di clock per leggere l'intero blocco
- $4 \times 1$  cicli di clock per inviare l'intera linea alla cache

Per un totale di 65 cicli di clock (del bus).

Una soluzione per migliorare la situazione sarebbe quella di aumentare la banda passante della RAM (ossia la quantità di dati che possiamo prelevare dalla RAM con un unico accesso in modo che una intera linea possa essere prelevata con una sola interrogazione della RAM).

Ciò può essere ottenuto organizzando la RAM in banchi in modo da leggere o scrivere più word contemporaneamente, aumentando la quantità di dati che possiamo prelevare dalla RAM con un unico accesso. In particolare, se un blocco (o linea di ram) è formata da  $n$  word, la linea viene distribuita su  $n$  banchi di RAM, ciascuno dei quali memorizza una delle  $n$  word della linea (allo stesso indirizzo di tutti gli altri banchi). In questo modo, gli  $n$  banchi possono essere letti in parallelo, estraendo così in una sola lettura tutte le word di cui è composta una linea. Questo schema è detto memoria interlacciata (interleaved memory) e permette di risparmiare sul tempo necessario a prelevare un'intera linea dalla RAM.

Questo schema è detto a memoria interlacciata (interleaved memory) e permette di risparmiare sul tempo necessario a prelevare un'intera linea dalla RAM (un principio abbastanza simile alla distribuzione degli strip nei sistemi RAID).

Usando la memoria interlacciata, con l'esempio di prima avremo:

- 1 ciclo di clock per inviare l'indirizzo della linea mancante
- 15 cicli di clock per prelevare le 4 word della linea (i quattro banchi vengono ora acceduti in parallelo)
- $4 \times 1$  cicli di clock per inviare l'intera linea alla cache

Per un totale di 20 cicli di clock, contro i 65 della soluzione precedente.

Un altro modo per ridurre il costo dei cache miss e di non dover andare fino in RAM per recuperare una linea mancante, mediante l'uso di più livelli di cache (che è ciò che succede appunto nelle architetture moderne).

In questo modo, si riesce tra l'altro a raggiungere un accettabile trade-off tra l'avere una cache veloce quanto la CPU (ma costosa, e quindi piccola) e una cache sufficientemente grande da contenere una buona parte della RAM (più lenta, ma più economica).

## L'influenza della cache sugli algoritmi

Normalmente, gli algoritmi sono valutati solo a livello teorico, in base al numero di operazioni da compiere per risolvere un problema, rispetto alla "dimensione" del problema.

A livello pratico tuttavia, il modo in cui un computer funziona può influenzare enormemente le effettive prestazioni di un programma.

Ci sono algoritmi che possono sembrare efficienti a livello teorico, ma che in realtà possono fornire prestazioni peggiori se consideriamo l'impatto della cache.

Nel caso specifico del Radix Sort, che è noto per essere più efficiente di QuickSort per array molto grandi, l'impatto della cache può comportare prestazioni peggiori rispetto al Quick Sort. Questo è dovuto alla presenza di cache miss generati dal Radix Sort, che possono comportare un aumento dei tempi di accesso alla memoria principale.

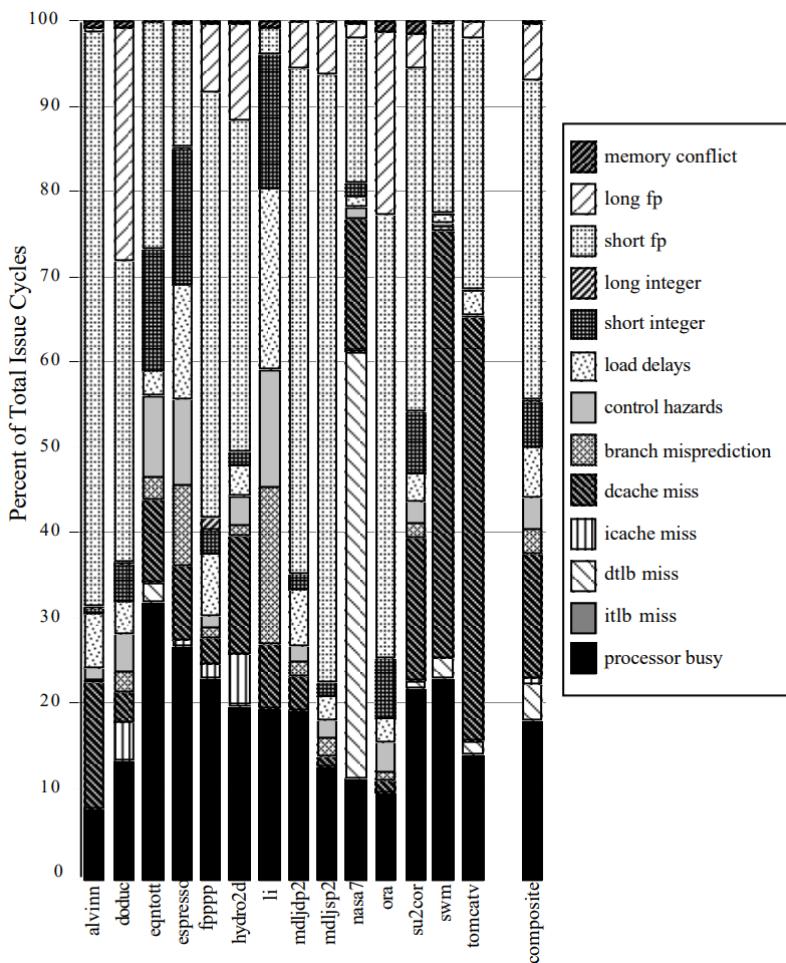
Per ovviare a questo problema, sono state sviluppate nuove versioni di Radix Sort e di altri algoritmi che tengono in considerazione la gerarchia di memorie dei computer. In questo modo, gli algoritmi possono essere ottimizzati per limitare le inefficienze causate dalla cache e migliorare le prestazioni complessive.

# Multithreading & TLP

Nell'architettura, il multithreading è la capacità di un'unità di elaborazione centrale (o di un singolo core in un processore multi-core) di fornire più thread di esecuzione contemporaneamente. Storicamente è nato su un core singolo, non per forza è necessario più di un core, anzi.

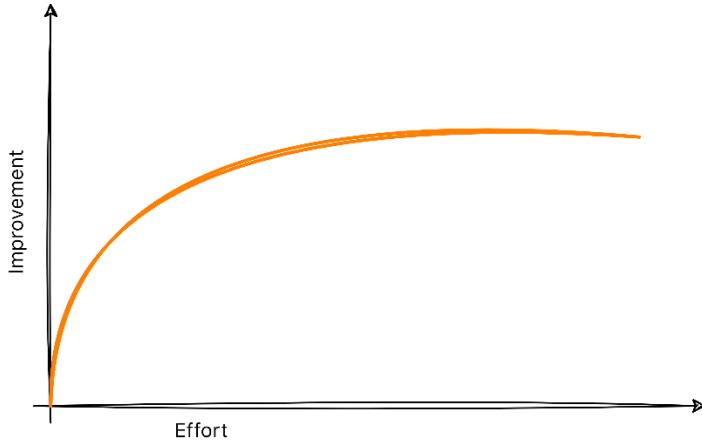
In parte si può dire che l'idea del multithreading sia nata dalla necessità di risolvere il problema del cache miss e dell'attesa necessaria per recuperare l'informazione mancante in RAM. Tuttavia, il multithreading è stato sviluppato anche per sfruttare appieno le risorse disponibili delle CPU e per migliorare le prestazioni dei programmi. Questo perché un cache miss produce una "lunga" attesa necessaria per recuperare l'informazione mancante in RAM. Se non c'è un'altra istruzione indipendente da poter eseguire, o se non è implementato lo scheduling dinamico della pipeline, la pipeline va in stall.

Per motivare l'introduzione del Multithreading e del Parallelismo a livello di Thread (Thread Level Parallelism – TLP), osserviamo questa immagine di un articolo "Simultaneous Multithreading by Dean Michael Tullsen - 1996":



L'immagine illustra come un singolo core superscalare 8-issue sfrutta le risorse della CPU. Si osserva che in media si riesce a sfruttare un 20% delle risorse. La maggior parte del tempo, i programmi sono soggetti a cache miss, branch misprediction, load delays, etc.

Le potenziali soluzioni sarebbero ad esempio incrementare le dimensioni delle cache, implementare dei branch predictors migliori, aumentare la window instruction per exploitare più ILP nelle istruzioni. Queste sono sicuramente delle opzioni, infatti negli anni si è sempre andato ad aumentare la dimensioni delle cache, l'obiettivo è sempre quello di rendere l'esecuzione di un singolo core il più veloce possibile.



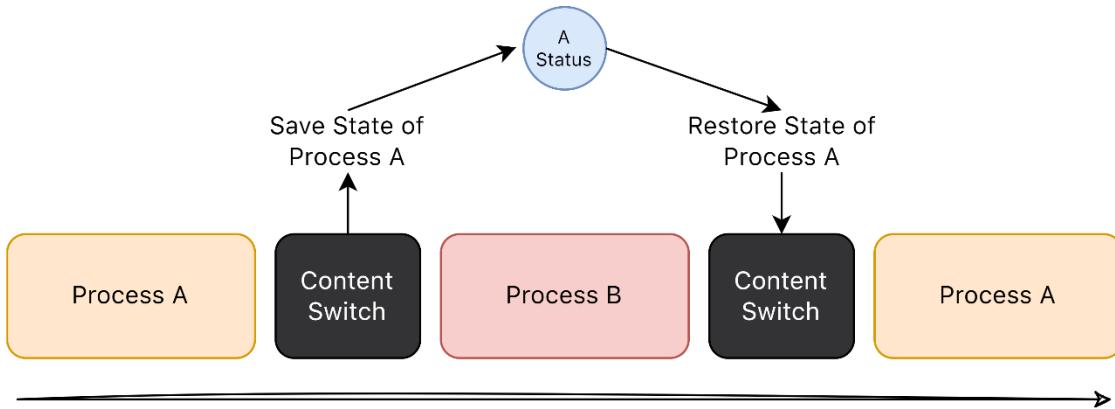
Il problema è che in questo caso per la legge dei rendimenti decrescenti, siamo arrivati ad un punto (per il singolo core) che per aggiungere anche solo un 5% di improvement è necessaria un effort enorme.

Si può dire che l'ILP è stato spremuto quasi ai suoi limiti. Questo ha portato allo sviluppo del TLP.

Il multithreading è anche un paradigma di programmazione che ovviamente è una conseguenza del fatto che esistono CPU in grado di supportare il multithreading hardware. Se il programmatore sfrutta tutto ciò, sviluppando programmi in grado di girare su una CPU che supporta il multi-threading potranno sfruttarne al meglio le caratteristiche architettoniche di quest'ultima. Al giorno d'oggi, le CPU sono sostanzialmente tutte multi-core, ciascuno dei quali implementa il multi-threading.

## Hardware & Software Multithreading

L'idea del Multithreading risale a tanto tempo fa, prima ancora che fosse applicata a livello hardware sulle CPU. Infatti, se pensiamo ai sistemi operativi Multitasking:



La latenza dovuta, ad esempio, da un'operazione di I/O di un processo A, è mascherata / assorbita dal fatto che viene fatto content switch con un altro processo B. In un sistema operativo che utilizza uno scheduling preemptive, questo si traduce in:

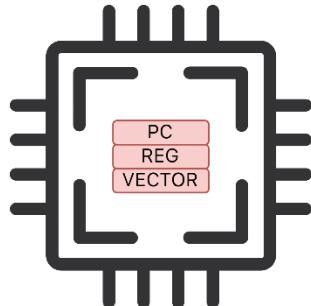
1. Il processo A, appena effettua l'I/O, viene spostato in coda di wait
2. Il processo B viene scelto dallo scheduler per andare in esecuzione
3. Il processo B effettua un'operazione di I/O, viene spostato in coda di wait (nel frattempo, il processo A ha concluso l'I/O, è stato spostato in coda di ready)
4. Il processo A viene riscelto per andare in esecuzione

Il problema del software Multithreading è che si ha la necessità di effettuare content switch. Questo necessita di molto tempo in quanto è necessario salvare gli stati del processo da qualche parte in

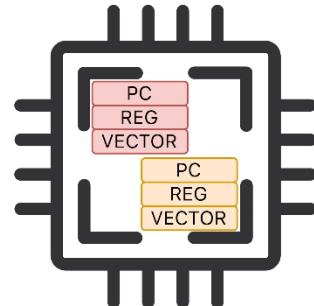
memoria che tipicamente prende il nome di PCB (process control block che include il PC, i registri ed il vettore di interrupt) e poi ripristinare questo stato.

Quello di cui si ha bisogno quindi, è rendere il costo di un context switch meno impattante a livello di tempistiche. Per farlo, è necessario implementare il context switch a livello hardware (**Hardware Multithreading**).

Il supporto hardware necessario per implementare il context switch a livello hardware è quello di replicare fisicamente le risorse. Quindi ogni thread deve avere il proprio context hardware:



**Single-Thread Processor**



**Multi-Thread Processor**

Questo significa che ogni core deve possedere più set di registri, più Programm counter, etc; per ogni hardware thread.

### Tipi di Hardware multithreading

Quando avviene il context switch quindi? Si potrebbe adottare la stessa idea del software multithreading, quindi quando un thread è "bloccato", e quindi:

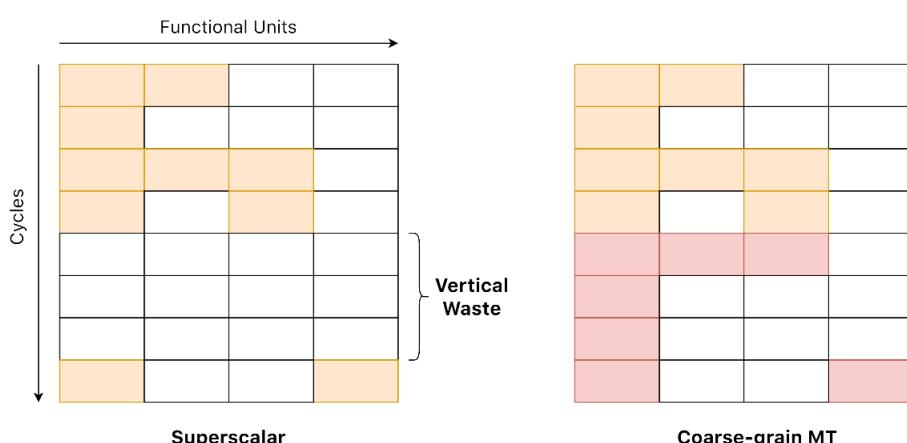
- L1 o L2 cache miss
- TLB miss
- Latenza dovuta all'attesa di un operando (ad esempio per Floating point operation)

In base a come vengono schedulati i Thread, si possono identificare tre tipi di hardware multithreading:

1. Coarse-grain MT
2. Fine-grain MT
3. Simultaneous MT

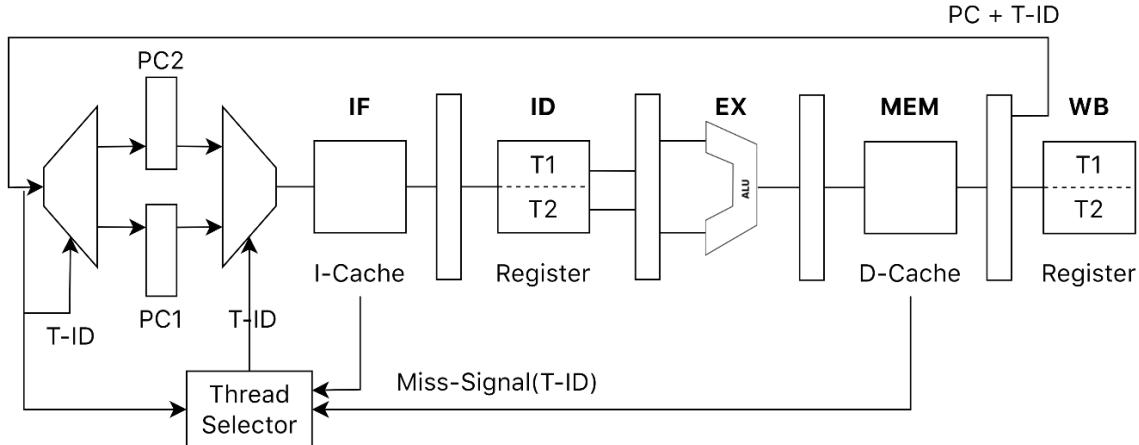
### Coarse-Grain Multithreading

L'idea principale dietro al Coarse-grain MT (anche chiamato Block Multithreading) è semplice ed è quella che si avvicina di più al Software multithreading: ogni Thread continua la sua esecuzione finché non viene bloccato da un evento che provoca un attesa lunga e prolungata.



A sinistra si vede un processore superscalare che non utilizza MT. Si nota come è presente quello che è chiamato un Vertical Waste, quindi cili di clock in cui il processore non sta utilizzando unità funzionali. Nel block multithreading, quello che andiamo a fare è cercare di riempire i vertical waste con un thread diverso (destra). Ovviamente questo cambio di contesto avrà un costo, ma il fatto che sia fatto a livello hardware, lo rende molto più veloce di un context switch software.

A livello hardware, un coarse-grain multithreading potrebbe essere implementato in questo modo:



Lo vediamo applicato alla classica pipeline a 5 stadi con 2 Thread. Prima di tutto è necessario avere 2 Program Counter in quanto bisogna recuperare le istruzioni per ogni thread. Inoltre, ogni istruzione porta con sé un bit che identifica il thread. Questo bit è il TID, o thread context ID. Da notare anche che il file dei registri è replicato.

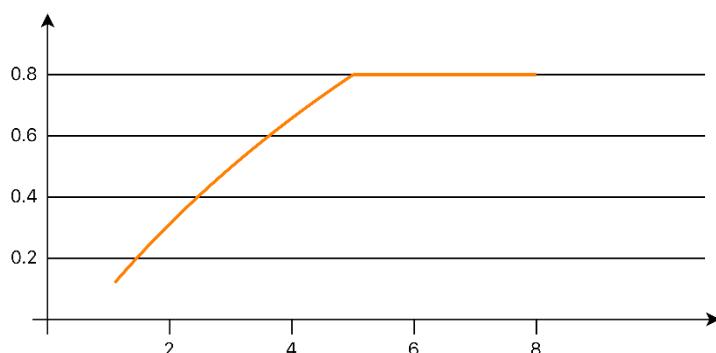
Successivamente è necessario un Thread Selector, il quale indirizza la pipeline nel fetch/decode/write-back sul Thread corretto. Un MISS nella Cache L1, triggerà il context switch: si può notare che i segnali di Miss-Signal che provengono dalle due cache (Istruzioni e dati) sono spediti al Thread Selector: questo permette al Thread Selector di "parcheggiare" il thread corrente e permettere ad un altro thread di andare in esecuzione. In caso di Cache-Miss è necessario Flushare la pipeline.

L'Instruction cache rimane identica, ma sicuramente è necessario averne una più capiente in quanto è necessario immagazzinare le istruzioni di più thread.

È necessario anche avere una TLB che include il Thread ID, in modo da separare la traduzione degli indirizzi virtuali-fisici.

### Considerazioni

Considerando questa immagine presa dal libro "*Parallel Computer Architecture: A Hardware/Software Approach*":



Nell'asse verticale si ha la % di utilizzo della CPU, mentre nell'asse orizzontale, il numero di Thread. Con un singolo Thread, si nota un utilizzo di meno del 20%. Duplicando i Thread, si ha quasi il doppio

dell'utilizzo. L'incremento è quasi lineare fino a 5 core, si ha un'appiattimento all'80% in quanto il context switch hardware ha comunque un costo.

I costi Hardware nell'implementare un meccanismo simile sono molteplici:

- Replicare lo stato hardware
- Avere 5 Thread implica un Thread Selector più complesso
- Incrementare la dimensione del TLB in quanto è necessario immagazzinare traduzioni per 5 Thread.

### **Block Multithreading in Out of order Cores**

Nei core che utilizzando l'esecuzione out of order, dato che il numero di istruzioni iniettate per ciclo è alto, il costo del context switch è molto alto perché il numero di istruzioni che deve essere flushato è molto alto. Di conseguenza il nuovo Thread che entra, impiegherà tempo a preparare tutte le stazioni di prenotazione.

Quindi il costo di context switch nel block multithread su un core che utilizza lo scheduling dinamico è ingestibile ed è anche per questo che non esistono esempi di block multithreading in core out of order.

## **Fine-Grained Multithreading**

Una tipologia di multithreading molto spinto prevede che il processore effettui lo switching tra i vari peer-thread ad ogni ciclo di clock, indipendentemente dal fatto che l'istruzione del thread in esecuzione abbia generato un cache miss. Questo tipo di multithreading richiede un'implementazione hardware molto sofisticata, in quanto la CPU deve essere in grado di effettuare lo switch tra i thread in modo molto efficiente e praticamente senza overhead.

Nel fine-grained multithreading, le istruzioni dei vari peer-thread sono eseguite secondo una politica di scheduling a round-robin, che prevede che ogni thread abbia la stessa opportunità di eseguire le proprie istruzioni. Questo significa che ogni thread ha la stessa priorità rispetto agli altri thread e che ogni istruzione viene eseguita per un certo periodo di tempo, prima di passare al thread successivo.

Questa tipologia di multithreading dovrebbe rimuovere la dipendenza dai dati dei singoli thread e quindi dovrebbe azzerare o comunque ridurre gli stalli della pipeline dovuta alla dipendenza dai dati. Dato che ogni thread dovrebbe funzionare in modo indipendente i singoli thread eseguiranno programmi non correlati e quindi vi saranno poche probabilità che le istruzioni di un thread necessitino dei risultati elaborati da un'istruzione di un altro thread in esecuzione in quel momento.

In generale, col fine-grained multithreading, in una architettura pipelined, se:

1. la pipeline ha k stage
2. ci sono almeno k peer-thread da eseguire
3. la CPU esegue lo switch tra thread ad ogni ciclo di clock

allora non ci può essere mai più di una istruzione per thread nella pipeline in un dato istante, le dipendenze sui dati si risolvono automaticamente, e la pipeline non va mai in stall (assumendo che un cache miss non richieda più cicli di clock del numero di thread in esecuzione).

A parte la necessità di riuscire ad implementare il context switch tra thread in maniera molto efficiente, lo scheduling fine-grained fra i thread ad ogni istruzione fa sì che un thread venga rallentato anche quando potrebbe proseguire la sua esecuzione perché non sta generando alcun stall.

Inoltre, ci potrebbero essere meno thread che stage della pipeline (anzi, è questo il caso più frequente), per cui può non essere così facile mantenere la CPU sempre occupata.

Tenendo presente questi problemi, un approccio complementare viene adottato nel multithreading coarse-grained (a "grana grossa")

## Simultaneous Multithreading

Le moderne architetture superscalari, multiple issue e a scheduling dinamico della pipeline danno la possibilità di sfruttare contemporaneamente il parallelismo insito nelle istruzioni di un programma (ILP) con il parallelismo insito in un insieme di peer threads: il Thread Level Parallelism (TLP): ILP + TLP = Simultaneous Multi-Threading (SMT)

La ragione per implementare l'SMT risiede nell'osservazione che le moderne CPU multiple-issue hanno più unità funzionali di quante siano mediamente sfruttabili dal singolo thread in esecuzione.

Sfruttando il register renaming e lo scheduling dinamico, istruzioni appartenenti a thread diversi possono essere eseguite insieme

nell'SMT, ad ogni ciclo di clock vengono avviate più istruzioni, potenzialmente appartenenti a thread diversi, aumentando così l'utilizzo delle varie risorse della CPU.

Nelle CPU superscalari senza multithreading, il multiple issue può venire vanificato dalla mancanza di sufficiente parallelismo tra le istruzioni di ogni thread, e/o da un lungo stall (ad esempio un cache miss su L3) che lasciano l'intero processore idle.

Nel coarse-grained MT, i lunghi stall sono mascherati dal passaggio ad un altro thread, ma la mancanza di parallelismo tra le istruzioni di ciascun thread limita il grado di utilizzo delle risorse della CPU (ad esempio, non possono essere usati tutti gli slot di issue disponibili)

Anche nel fine-grained MT, la mancanza di ILP in ciascun thread può limitare l'utilizzo delle risorse della CPU

SMT: le istruzioni appartenenti a thread diversi sono (quasi) certamente indipendenti, se possiamo lanciarle assieme aumentiamo il grado di utilizzo delle risorse della CPU.

Anche nell'SMT comunque, non si riesce sempre ad avviare il massimo numero possibile di istruzioni per ciclo di clock, a causa del numero limitato di unità funzionali disponibili e di stazioni di prenotazione disponibili, della capacità della cache di istruzioni di fornire le istruzioni dei vari thread, e del numero di thread presenti.

E' evidente, che l'SMT può essere adeguatamente supportato solo se sono disponibili registri rinominabili in abbondanza.

Inoltre, nel caso di una CPU che supporti la speculazione, sarà importante avere (almeno logicamente) un ROB distinto per ogni thread, in modo da gestire in modo indipendente l'operazione di commit.

# Architetture Parallelle

Come si sa già, programmatore scrive un programma per risolvere un problema, che viene poi eseguito dalla CPU come una sequenza di istruzioni. Tuttavia, non si sa come il compilatore (static-issued) o la CPU (dynamic-issued) manipoleranno queste istruzioni per ridurre il tempo di esecuzione del programma.

E se il programma continua ad essere lento? Si può passare ad un processore più veloce o riscrivere il programma, ma se questo non funziona, l'unica soluzione è utilizzare un'architettura parallela dotata di più unità di elaborazione per eseguire parti del codice in parallelo.

Il programmatore a questo punto deve quindi scegliere una soluzione algoritmica adeguata per sfruttare l'architettura parallela. Molti problemi hanno una soluzione che richiede l'esecuzione parallela di diversi programmi, ma l'incremento delle prestazioni ottenuto tramite l'uso di più CPU non è lineare rispetto al numero di CPU disponibili.

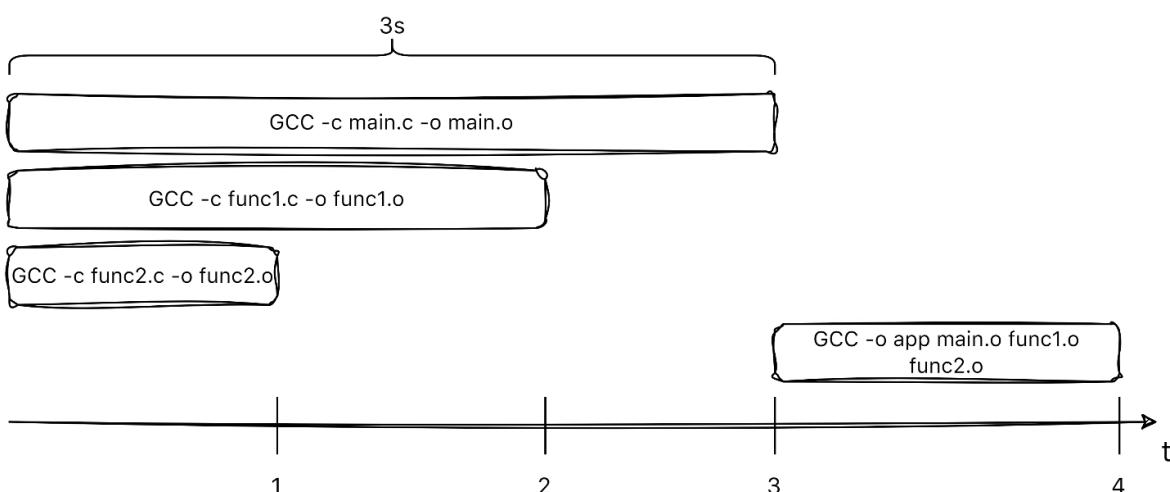
Ciò è dovuto al fatto che i programmi che girano in parallelo dovranno prima o poi sincronizzarsi per condividere i dati elaborati da ciascuno o eseguire una fase comune di inizializzazione. Ad ogni modo, ci sarà sempre una parte del lavoro che non può essere svolta in parallelo con tutte le altre operazioni.

Pensiamo ad esempio di lanciare la compilazione su una macchina monoprocessoresso, e che ci vogliano:

- 3 secondi per compilare main.c
- 2 secondi per compilare func1.c
- 1 secondo per compilare func2.c
- 1 secondo per linkare gli oggetti (main.o, func1.o, func2.o)

Eseguendo tutto in sequenza, andremmo a finire il tutto dopo 7 secondi.

Con 3 CPU a disposizione ad esempio, i tre sorgenti possono essere compilati in parallelo per generare i corrispondenti moduli oggetto. Quando gli oggetti sono stati prodotti, possono essere linkati assieme usando uno dei tre processori.



Ma l'operazione di linking può essere eseguita solo dopo che tutti e tre i file oggetto sono stati prodotti, ossia solo dopo 3 secondi.

Il tempo necessario per generare l'output sarà quindi  $3+1=4$  secondi, contro i 7 secondi del sistema monoprocessoresso, per uno incremento delle prestazioni (speedUp) di  $7/4 = 1.75$ , pur avendo usato un numero triplo di processori.

## SpeedUp e Legge di Amdahl

Lo speedUp è un numero che misura le prestazioni relative di due sistemi che elaborano lo stesso problema. Più tecnicamente, è il miglioramento della velocità di esecuzione di un compito eseguito su due architetture simili con risorse diverse (nel caso di prima le risorse che cambiavano erano i Core fisici).

La nozione di SpeedUp (accelerazione) è stata stabilita dalla [legge di Amdahl](#). La legge di Amdahl, viene usata per trovare il miglioramento atteso massimo in una architettura di calcolatori o in un sistema informatico quando vengono migliorate solo alcune parti del sistema.

La legge di Amdahl che era particolarmente focalizzata sull'elaborazione parallela per predire il miglioramento atteso massimo di velocità che si ottiene usando più processori.

Nel caso speciale del parallelismo, la legge di Amdahl afferma che se  $P$  è la porzione di un calcolo che è parallelizzabile (cioè che può beneficiare dal parallelismo), e  $(1 - P)$  è la porzione che non può essere parallelizzata, allora l'aumento massimo di velocità che si può ottenere usando  $N$  processori è:

$$\frac{1}{(1 - P) + \frac{P}{N}}$$

↑                           ↑  
Serial part =  $1 -$       Parallel job is  
Parallel Part( $P$ )        divided by  $N$  cores

Al limite, al tendere di  $N$  all'infinito, lo speedUp tende a  $1/(1-P)$ . In pratica, il rapporto prestazioni/prezzo scende rapidamente al crescere di  $N$ , dato che  $P/N$  diventa piccolo rispetto a  $(1-P)$ .

Quindi  $(1-P)$  è il massimo speedup che possiamo aspettarci, a prescindere dal numero di processori. Se la percentuale di codice seriale è il 10% il massimo speedup ottenibile sarà 10 a prescindere dal numero di processori utilizzato.

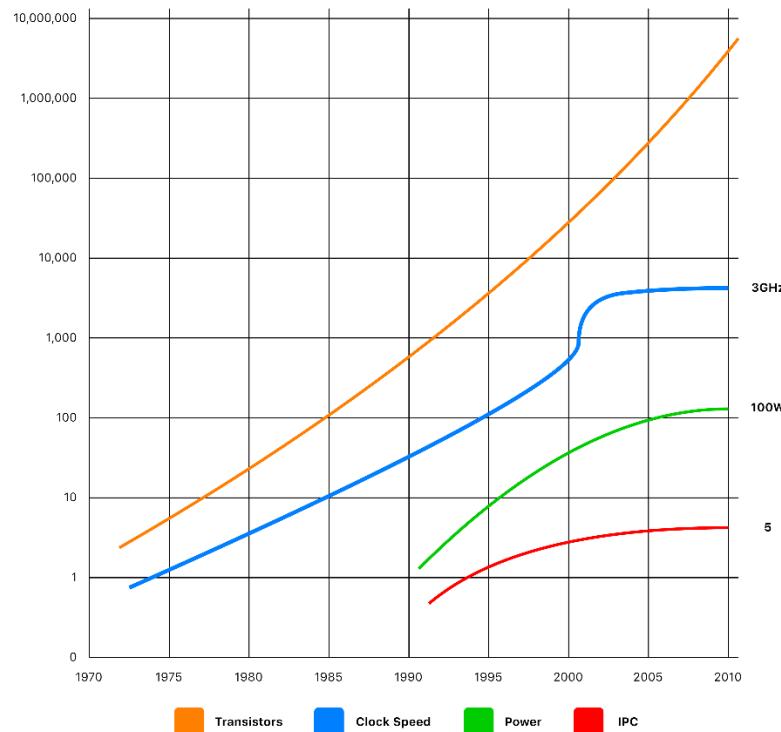
Naturalmente, per molte applicazioni avere uno speed-up meno che lineare, ad esempio di  $n$  usando  $2n$  processori è comunque più che accettabile, visto il costo sempre più basso delle CPU. Inoltre, l'aumento della potenza computazionale non è l'unica motivazione che porta a progettare architetture multi-processore:

- La presenza di più processori aumenta l'affidabilità del sistema: se anche uno si guasta, altri possono svolgere il lavoro al suo posto.
- Servizi che per loro natura sono forniti su ampia scala geografica, devono essere implementati con una architettura distribuita. Se il sistema fosse centralizzato in un unico nodo, l'accesso di tutte le richieste a quest'unico nodo costituirebbe probabilmente un collo di bottiglia in grado di rallentare enormemente il servizio fornito.

# Multiprocessing

Un sistema Multiprocessore è un sistema di elaborazione con più processori e più precisamente un numero di unità di elaborazione centrale collegate insieme per consentire l'elaborazione parallela. In altre parole, sono sistemi che consentono a più processori di lavorare insieme per eseguire un lavoro in modo più efficiente rispetto a un singolo processore.

Perché si dovrebbe puntare ad avere più processori?



Questa immagine mostra l'andamento dei processori intel nel corso degli anni.

Quello che si nota è che la velocità del clock si è sempre più appiattita con il tempo. Questa è legata alla potenza che il processore è in grado di ricevere, ma non è il motivo per il quale si è sempre più appiattita. Infatti, il problema non è tanto mandare più potenza all'unità di calcolo, ma quanto la sua densità di potenza (power density) indicata in Watt/cm<sup>2</sup>.

La power density è la quantità di potenza per unità di volume, che si trasforma in energia e quindi calore. Più cresce la potenza, più cresce la frequenza del clock e di conseguenza la power density. Di conseguenza è necessario avere dei sistemi di raffreddamento sempre più performanti. Questo non è scalabile.

Siamo quindi di fronte ad un muro che non è facilmente valicabile: un singolo core non può scalare all'infinito, in quanto limitato da:

- **Limite dovuto alla power density:** limita il processore a certe velocità del clock, in quanto il clock cresce al crescere della potenza.
- **Limite dovuto al parallelismo:** Non è possibile estrarre più di tanto parallelismo dai programmi.
- **Limite dovuto alla memoria:** Le memorie sono molto lente rispetto alla CPU.

La cosa positiva è che la legge di Moore è ancora valida: è possibile inserire sempre più transistor in un circuito integrato. Come è possibile quindi, a confronto di questi limiti, avendo solo la legge di Moore in aiuto, a rendere le CPU più veloci? La soluzione che la comunità scientifica ha scelto è quella di implementare CPU Multicore. Quindi anziché rendere un singolo core più veloce, se ne inseriscono di più insieme, in un singolo chip.

Un sistema multiprocessore (o in generale un Architettura parallela), è una collezione di unità di elaborazione che cooperano e comunicano tra di loro per risolvere in modo rapido problemi di larga complessità.

A livello architetturale, un architettura parallela è composta da due elementi:

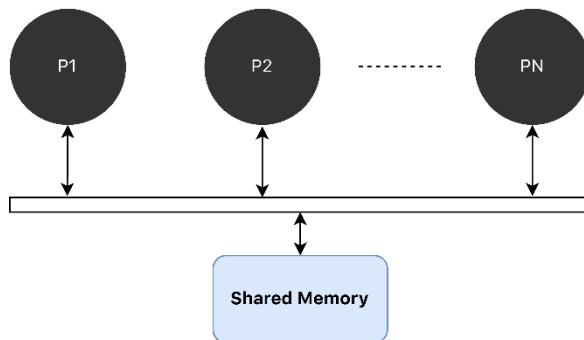
- Computing model: Definisce come i core (o unità di elaborazione) sono organizzati e come i dati vengono processati.
- Communication model: Definisce come i core comunicano tra di loro.

Come communication model ne esistono di due tipi: uno basato su scambio di messaggi, l'altro attraverso una memoria condivisa (shared memory). Come communication model ci concentriamo su quelle a memoria condivisa.

Le architetture parallele a memoria condivisa possono essere classificate in base all'organizzazione fisica della memoria:

- Symmetric Multiprocessor (SMP)
- Distributed Shared-Memory (DSM)

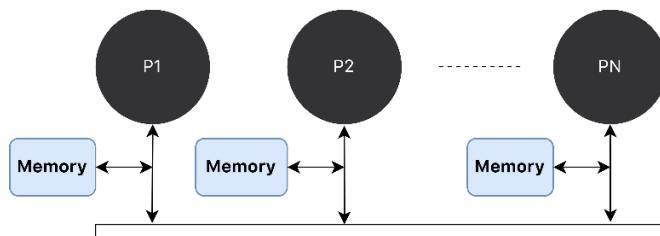
Nelle SMP la shared memory è centralizzata ed unisce i vari nodi attraverso un bus di sistema. Tuttavia, l'accesso concorrente dei nodi alla memoria può causare congestione del bus, limitando le prestazioni del sistema.



Il termine "Symmetric" deriva appunto dal fatto che la memoria è simmetrica per tutti i processori. Per questo motivo gli SMP sono anche chiamati Unified Memory Access (UMA), in quanto l'accesso alla memoria è uniforme indipendentemente da quale locazione di memoria viene acceduta.

Siccome tutte le CPU vedono lo stesso spazio di indirizzamento, è sufficiente una copia del sistema operativo. Inoltre, quando un processo termina o va in wait per qualche ragione, il S.O. può cercare nella coda di ready un altro processo a cui dare la CPU idle.

Le DSM d'altra parte, ha un organizzazione distribuita della memoria.



In questa organizzazione, ogni processore è direttamente collegato ad modulo di memoria locale e l'aggregato di tutte le memorie locali, formano la shared memory. Inoltre, la memoria locale ha dei tempi di accesso molto più brevi rispetto ad una accesso ad una memoria remota. Per questo motivo DSM sono anche chiamati Non Uniform Memory Access (NUMA).

## Cache Coherence

La cache, come detto nella sezione dedicata alle Cache, è necessaria in quanto riduce la latenza dovuta ai tempi di accesso lunghi in memoria centrale.

Nei sistemi multiprocessori però, introduce un problema chiamato Cache Coherence (coerenza della cache) in quanto le varie unità di calcolo condividono dati: se ci sono più copie dello stesso dato, in più cache, come ci assicuriamo che le copie distribuite siano le stesse (coerenti)?

Quindi una memoria è definita coerente se e solo se, tutti i processori, in ogni istante di tempo, hanno una vista consistente dei dati scritti, per ogni locazione di memoria. In particolare, un sistema è coerente se rispetta 3 proprietà:

1. **Preservare l'ordine del programma:** Una lettura da parte del processore P alla posizione X che segue una scrittura da P a X, senza che si verifichino scritture di X da parte di un altro processore tra la scrittura e la lettura da parte di P, restituisce sempre il valore scritto da P.
2. **Vista coerente della memoria:** Una lettura da parte di un processore nella posizione X che segue una scrittura da parte di un altro processore in X restituisce il valore scritto se la lettura e la scrittura sono sufficientemente separate nel tempo e non si verificano altre scritture in X tra i due accessi.
3. **Scritture serializzate:** Le scritture nella stessa posizione vengono serializzate; ovvero, due scritture nella stessa posizione da parte di due processori qualsiasi vengono visualizzate nello stesso ordine da tutti i processori. Ad esempio, se i valori 1 e poi 2 vengono scritti in una posizione, i processori non possono mai leggere il valore della posizione come 2 e successivamente leggerlo come 1.

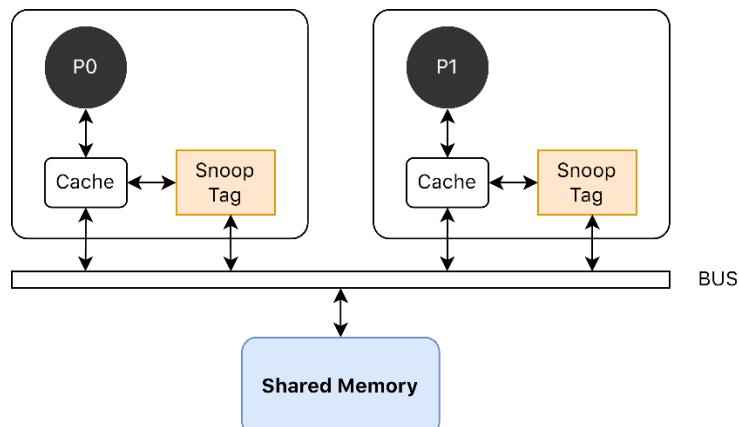
## Cache coherence protocols

I protocolli utilizzati per mantenere la coerenza in sistema multiprocessore, vengono chiamati Cache Coherence Protocols. Ci sono due principali tipi di protocolli di coerenza:

- Snooping protocols
- Directory-Based protocols

## Snooping protocols

In questo tipo di protocollo, ogni core (più precisamente, il cache controller), monitora il bus di sistema: il cache controller monitora ogni transazione sul bus di sistema. Se sul bus appare una transazione che modifica un blocco il quale indirizzo è anche nella cache locale, lo snooper esegue un'azione per garantire la coerenza della cache.



I protocolli di snooping possono essere ancora divisi in:

- Write-Invalidate protocols
- Write-Update (or Write broadcast) protocols

Nei protocolli **Write-Invalidation** quando un processore scrive su un blocco di cache (modificando dei dati condivisi), tutte le copie condivise nelle altre cache vengono invalidate tramite lo snooping del bus. Per risolvere il problema della congestione del bus di sistema (che si ha invece con write-update), il blocco non viene propagato subito in memoria primaria, ma verrà propagato dopo un tot di modifiche.

Il protocollo **write-update (o write broadcast)** invece, non invalida tutte le copie nelle altre cache locali, ma le aggiorna con il valore aggiornato.

Quando un processore scrive la stessa Word in modo iterativo (variabile contatore che controlla la ripetizione di un ciclo FOR)

prevede che ogni scrittura su una linea di cache debba essere propagata anche sulla memoria principale in modo immediato, quindi ogni volta che si scrive in una linea di cache, si scrive anche sulla memoria principale. In questo modo, ogni processore che condivide la stessa linea di memoria condivisa avrà sempre la stessa versione dei dati. Tuttavia, questo metodo può essere lento in quanto ogni scrittura deve essere propagata anche alla memoria principale.

L'azione che viene attuata alla entry cache di cui ci si accorge avere dati incoerenti, dipende dall'implementazione. In alcune implementazioni si marchia la entry con un dirty bit, il quale indica che la entry non è più valida, in altre si aggiorna direttamente la entry con i nuovi dati.

| Scenario                                                                    | Write-Invalidation | Write-Update                    |
|-----------------------------------------------------------------------------|--------------------|---------------------------------|
| Scritture multiple sulla stessa word: Wr(A), Wr(A), ...                     | 1 invalidation     | Scritture multiple in broadcast |
| Scritture su diverse Word nello stesso cache block: Wr(A[0]), Wr(A[1]), ... | 1 invalidation     | Scritture multiple in broadcast |
|                                                                             | Lento              | Rapido                          |

Come mostrato dalla tabella, il problema del protocollo write-update, è che genera un traffico sul bus esagerato: se un processore scrive in modo iterativo la stessa Word, write-validate dovrà invalidare una volta mentre write-update, per ogni modifica, dovrà aggiornare il dato. Stesso concetto si applica a diverse word sullo stesso cache block.

Write-update consuma molta più bandwidth sul bus di sistema rispetto a write-validate, infatti, è molto difficile trovare questo tipo di implementazione su CPU attuali.

La filosofia che sta dietro ai due protocolli però, è la stessa: quando dati specifici sono condivisi da più cache e un processore modifica il valore dei dati condivisi, la "notifica" della modifica deve essere propagata a tutte le altre cache. Questa propagazione delle modifiche impedisce al sistema di violare la coerenza della cache.

## MSI Protocol

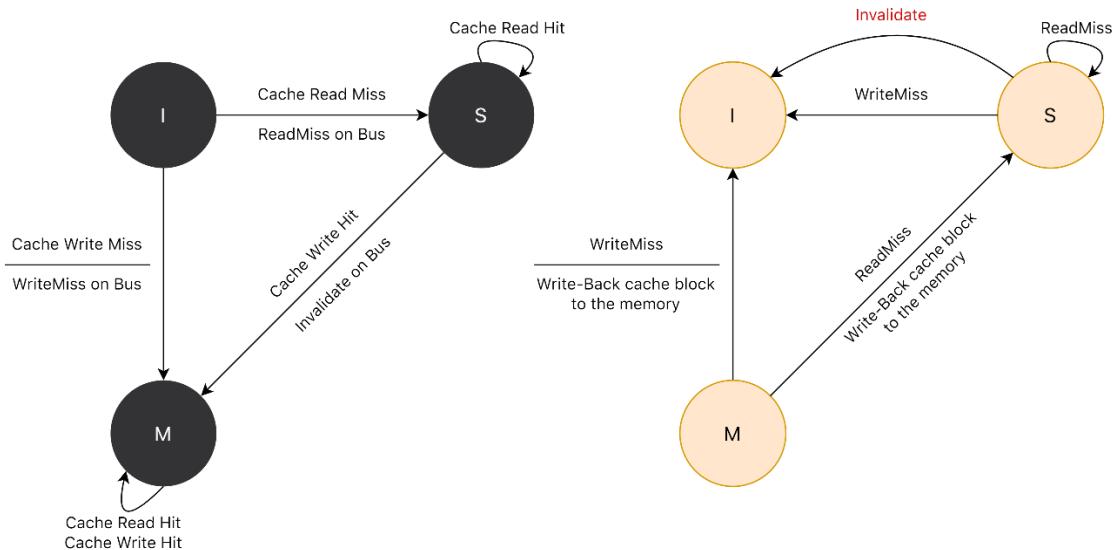
MSI è uno dei primi protocolli Write-validate. In questo protocollo di coerenza, ogni blocco contenuto in cache può avere 3 stati (le lettere del nome del protocollo identificano i possibili stati in cui può trovarsi una riga della cache):

- **Modified (M):** Il blocco è stato modificato ed è presente solo in questa cache. La cache che contiene in modo esclusivo questo blocco quindi può effettuare letture e scritture senza comunicarlo. Inoltre, se un'altra cache richiede questo blocco, dovrà recuperarlo da questa cache (i dati nella cache sono quindi incoerenti con la memoria principale). Una cache con un

blocco nello stato "M" ha la responsabilità di scrivere il blocco nella memoria quando viene rimosso.

- **Shared (S):** questo blocco è presente in una o più cache locali. Chiunque abbia la copia di questo blocco può leggerla senza comuncarlo. La cache inoltre, può eliminare i dati senza scriverli in memoria.
- **Invalid (I):** questo blocco non è presente nella cache corrente o è stato invalidato da una richiesta precedente e deve essere recuperato dalla memoria o da un'altra cache.

Diagrammi di stato:



Il diagramma a stati di sinistra mostra la transizione degli stati di un blocco di cache, ad esempio: Se un blocco si trova nello stato 'I' e la CPU legge quel blocco (Cache Read Miss) inoltrerà sul bus un segnale di ReadMiss passando allo stato shared appena arriverà il blocco richiesto.

Il diagramma di destra mostra le transizioni di un blocco rispetto a cosa lo snooper riceve dal bus, ad esempio: Se lo snooper riceve un invalidate per un blocco che è nello stato 'S', allora quel blocco passa allo stato Invalid 'I'. Oppure, se lo snooper riceve un ReadMiss per un blocco in stato 'M', dato che è l'unica cache ad avere quel blocco, effettuerà il write back alla memoria centrale, dicendo alla cache che ha richiesto il blocco di abortire la richiesta alla memoria centrale.

## MESI Protocol

Per giustificare l'aggiunta di uno stato, utilizzando il protocollo di coerenza MESI, si consideri questo scenario:

| Time  | Thread1 of CPU1 | Thread2 of CPU2 |
|-------|-----------------|-----------------|
| $t_0$ |                 |                 |
| $t_1$ | Read A          |                 |
| $t_2$ | Write A         |                 |
| $t_3$ |                 | Read B          |
| $t_4$ |                 | Write B         |

| Time  | BusRequest | state of A in CPU1 cache | state of B in CPU2 cache |
|-------|------------|--------------------------|--------------------------|
| $t_0$ |            | I                        | I                        |
| $t_1$ | ReadMiss A | S                        | I                        |
| $t_2$ | invalidate | M                        | I                        |
| $t_3$ | ReadMiss B | M                        | S                        |
| $t_4$ | invalidate | M                        | M                        |

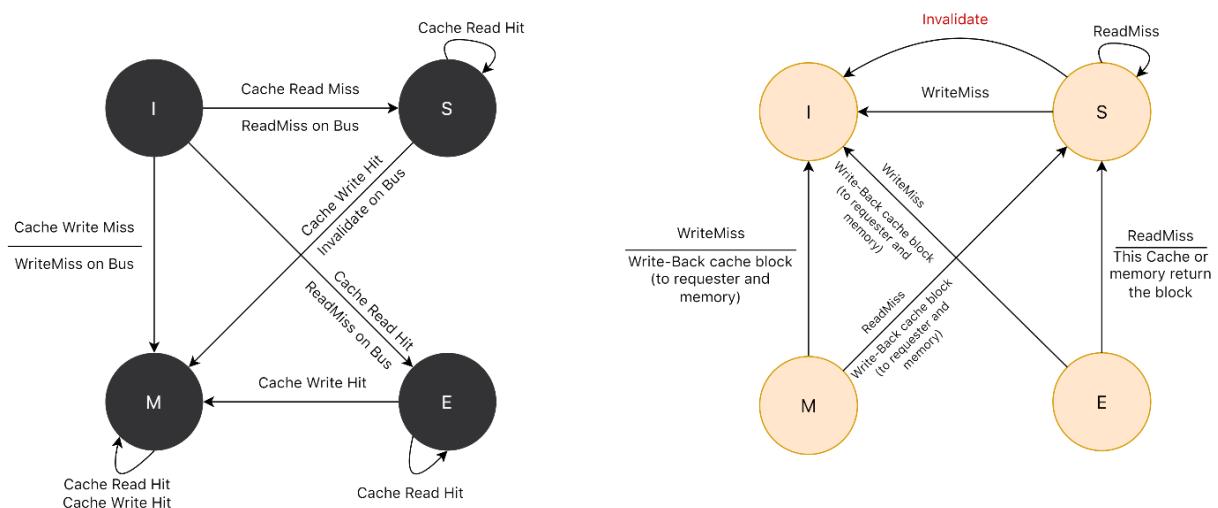
Al tempo  $t_2$ , il Thread1 effettua una scrittura all'interno del blocco A. Per il protocollo MSI, dato che il blocco è nello stato 'S' ed effettua un write hit, bisogna inoltrare un segnale di invalidate di quel blocco sul BUS, per tutte le altre cache. È necessario inoltrare un invalidate sul bus, se si è gli unici a possedere quel blocco? La risposta è No, in quanto non ci sono altre copie di quel blocco in altre cache. (stessa cosa vale al tempo  $t_4$ )

Il protocollo MESI, aggiungendo uno stato (Exclusive) evita il broadcast di invalidate per blocchi esclusivi.

Le lettere nell'acronimo MESI rappresentano quattro stati di coerenza con cui può essere contrassegnata una riga della cache (codificata utilizzando due bit aggiuntivi):

1. Modified (M): il blocco è stato modificato e non è coerente con la memoria principale o con altre cache. Solo il processore che detiene il blocco di cache in modalità Modified può accedervi e può scriverci senza invalidare i dati nella cache degli altri processori.
2. Exclusive (E): un blocco di cache in modalità Exclusive è valido e coerente solo nella cache di un singolo processore. Nessun altro processore ha lo stesso blocco di cache nella sua cache e quindi non ci sono conflitti di coerenza.
3. Shared (S): un blocco di cache in modalità Shared è stato letto da almeno due processori e si trova in una cache coerente. I processori che detengono questo blocco di cache possono solo leggerlo e non possono scriverci.
4. Invalid (I): un blocco di cache in modalità Invalid non è valido e non contiene dati coerenti.

Diagrammi di stato:



## MOESI e MESIF

Uno dei problemi che affronta il MESI, è che, se ci fossero operazioni continue di lettura e scrittura, da varie cache, su un particolare blocco, i dati devono essere scaricati ogni volta in memoria. Pertanto, la memoria principale rimarrà in uno stato pulito, ma questo non è un requisito ma solo un sovraccarico aggiuntivo. Questa sfida è stata superata dal protocollo MOESI.

MOESI consente di inviare linee di cache "sporche" direttamente tra le cache, invece di riscrivere ogni volta in memoria, e quindi leggere da lì ogni volta. Il protocollo MOESI è utilizzato da AMD.

Il MESIF consente alle cache di inoltrare una copia di una riga di cache pulita a un'altra cache, così che altre cache debbano rileggerla dalla memoria per ottenere un'altra copia condivisa. Il MESIF è utilizzato invece da INTEL.

## Limitazioni degli SMP

I sistemi SMP a bus condiviso, sono efficienti fino ad un numero limitato di nodi. Questo perché il bus non scala bene con il numero di nodi: la bandwidth per nodo diminuisce all'aumentare dei nodi stessi. Ed aumentando il numero di nodi, aumenterebbe la lunghezza del bus e di conseguenza si avrebbero problemi di latenza. L'uso di un bus singolo su cui si affacciano tutti i processori che comunicano condividono la memoria, limita quindi la dimensione di sistemi SMP ad un limite di nodi di circa 32.

I sistemi SMP sono spesso utilizzati in sistemi a basso costo, come i desktop e i server entry-level, dove la gestione della memoria non è critica e le prestazioni non devono essere estremamente elevate. Tuttavia, possono essere limitanti in sistemi di elaborazione ad alte prestazioni, come i server di

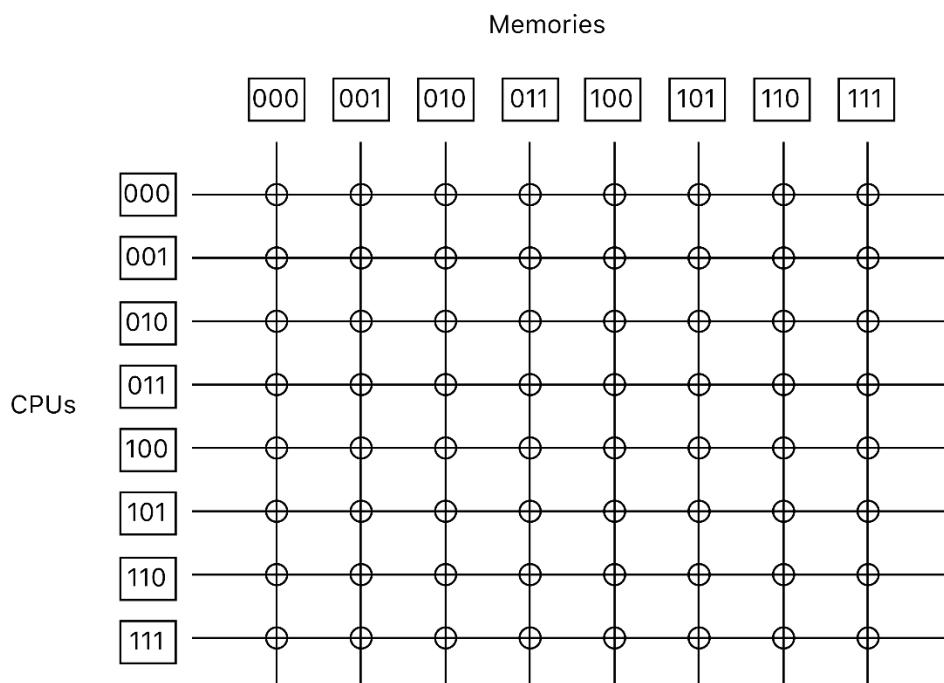
database, i cluster di calcolo e le macchine di elaborazione grafica, dove la gestione della memoria è critica e le prestazioni devono essere elevate.

Esistono altre strutture di interconnessione che non sono basate su bus condiviso, lo schema più semplice per connettere n nodi a k memorie è a commutatori incrociati (crossbar switch), un sistema simile a quello usato per decenni nelle centrali telefoniche.

### Crossbar switch

Il sistema di interconnessione crossbar switch nei sistemi SMP è una tecnologia utilizzata per collegare multiple CPU (Central Processing Unit) e dispositivi di memoria condivisa, consentendo loro di accedere alle stesse risorse di memoria condivisa in modo uniforme e rapido.

In pratica, il crossbar switch è un'interfaccia di collegamento che consente alle CPU e ai dispositivi di memoria di comunicare tra loro attraverso un meccanismo di commutazione. Il sistema di interconnessione crossbar switch prevede la presenza di una matrice di commutazione bidimensionale, dove ogni nodo rappresenta una CPU o un dispositivo di memoria e ogni collegamento rappresenta una connessione tra questi nodi.



Quando una CPU o un dispositivo di memoria vuole accedere a una determinata posizione di memoria condivisa, invia una richiesta al crossbar switch, che seleziona il collegamento corrispondente e reindirizza la richiesta alla CPU o al dispositivo di memoria appropriato. Questo processo avviene in modo simultaneo e indipendente per ogni richiesta in arrivo, consentendo a più CPU e dispositivi di memoria di accedere alle risorse di memoria condivisa in modo uniforme e senza conflitti.

Il numero di switch necessari per realizzare questo schema però cresce quadraticamente col numero di CPU (memorie) coinvolte: n CPU ed n memorie richiedono  $n^2$  switch.

La cosa è accettabile per sistemi di media grandezza (vari sistemi multiprocessore della Sun usano questo schema), ma certamente non è usabile in un sistema con, ad esempio, 256 CPU (sarebbero necessari 256<sup>2</sup> switch).

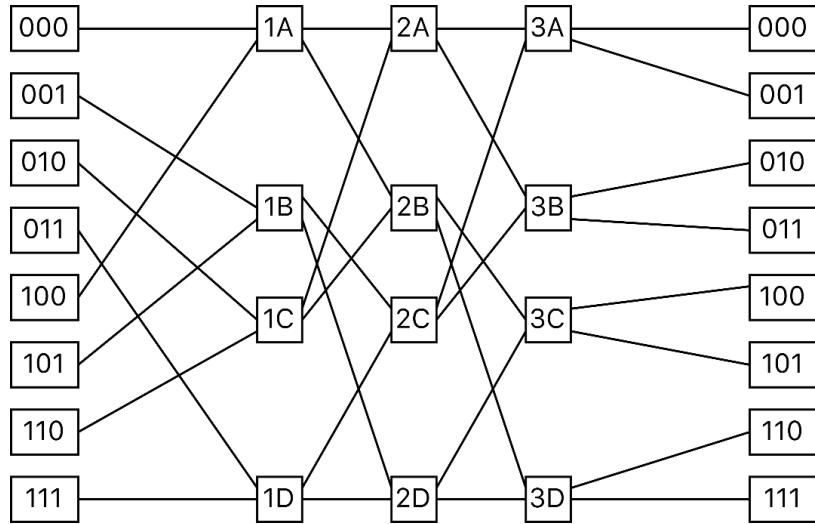
### Multistage Switching Network

Dunque, quando il numero di CPU è alto, si può usare un sistema basato su switch bidirezionali con due ingressi e due uscite: in questi switch ciascun ingresso può essere rediretto su ciascuna uscita



| Module | Address | Opcode | ? Value |
|--------|---------|--------|---------|
|--------|---------|--------|---------|

Esistono diversi tipi di reti di interconnessione multistadio. Una delle più usate è quella che viene chiamata Rete Omega:



Una rete Omega è costituita da più stadi di elementi di commutazione. Ogni ingresso ha una connessione dedicata a un'uscita.

Una rete omega  $N \times N$  ha un numero  $\log(N)$  di stadi e un numero  $N/2$  di elementi di commutazione in ogni stadio per uno shuffle perfetto tra gli stadi. Ogni rete utilizza il proprio algoritmo di commutazione.

Nel caso di sistemi UMA che utilizzano switch bidirezionali, i messaggi scambiati tra le CPU e la memoria sono suddivisi in quattro parti fondamentali, ovvero:

- Module: La parte module specifica quale modulo di memoria deve essere utilizzato per soddisfare la richiesta della CPU e quale CPU ha emesso la richiesta
  - Address: specifica l'indirizzo all'interno del modulo di memoria a cui la CPU sta cercando di accedere
  - Opcode: specifica l'operazione richiesta dalla CPU, ad esempio READ o WRITE
  - Value (opzionale): specifica il valore che deve essere scritto nella memoria in caso di operazione di scrittura (WRITE).

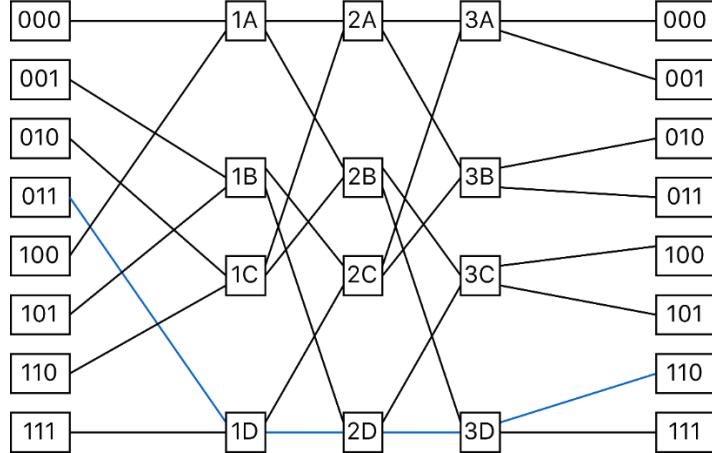
Lo switch bidirezionale può essere programmato in modo da analizzare la parte module del messaggio e determinare su quale output instradare il messaggio, ovvero quale modulo di memoria deve ricevere la richiesta della CPU. In questo modo, lo switch può garantire che ogni richiesta di accesso alla memoria venga instradata verso il modulo di memoria corretto.

Una volta che la richiesta è stata instradata verso il modulo di memoria corretto, la memoria stessa può accedere ai dati richiesti e restituire il valore richiesto alla CPU. In questo modo, lo switch bidirezionale funge da intermediario tra la CPU e la memoria, garantendo che le richieste di accesso alla memoria vengano soddisfatte in modo efficiente e preciso.

L'algoritmo di instradamento, per ogni switch preleva il bit più significativo (quello più a sinistra) e lo usa per l'instradamento: 0 instrada la richiesta sull'output superiore, 1 instrada la richiesta sull'output inferiore, poi passa al bit successivo.

Un esempio: La CPU 011 vuole leggere un dato nel modulo di RAM 110:

| Memory bankCPU |          |      |     |  |
|----------------|----------|------|-----|--|
| 110011         | 0x012... | READ | NUL |  |



È possibile che ci siano più comunicazioni in parallelo, ma non qualsiasi tipo di comunicazione: Consideriamo invece cosa accade se la CPU 000 vuole accedere il modulo 000. La sua richiesta confliggerebbe con la richiesta della CPU 001 sullo switch 3A: una delle due richieste deve attendere.

Al contrario di quello che accade con le reti che usano crossbar switch, le reti omega sono reti bloccanti: non tutte le sequenze di richieste possono essere servite contemporaneamente. I conflitti possono verificarsi sull'uso di una connessione, di uno switch, o in una richiesta alla memoria o una risposta ad una CPU. Varie tecniche possono essere usate per connettere CPU e memorie in modo da minimizzare la possibilità di conflitti e massimizzare il parallelismo delle comunicazioni CPU-memoria.

## Directoy-based Protocol

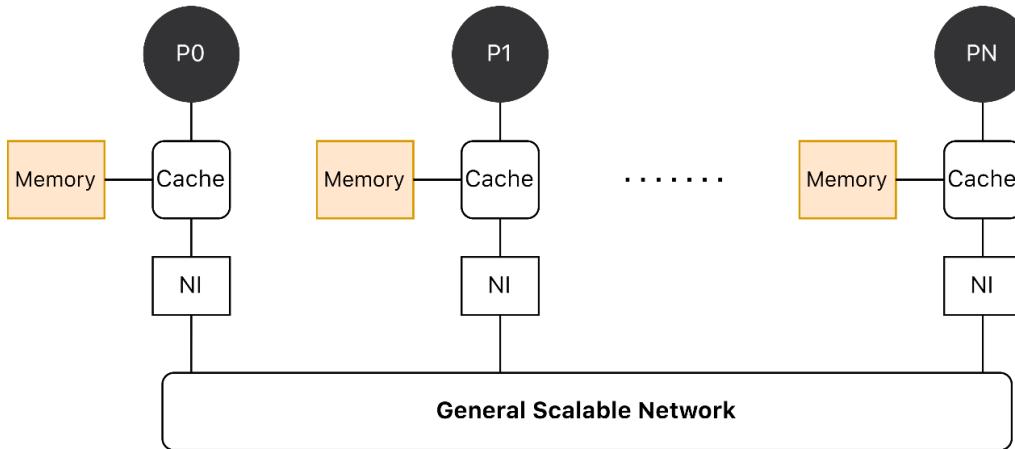
Pur utilizzando tecniche di crossbar e multistage switch, gli SMP sono limitati ad un nodi che non scala molto bene. Ci vengono in aiuto i sistemi **DSM (Distributed shared memory)**. Per supportare un numero maggiore di nodi, la memoria deve essere distribuita tra i nodi piuttosto che centralizzata; in caso contrario, il sistema non sarebbe in grado di supportare le richieste di larghezza di banda di un numero maggiore di processori senza incorrere in una latenza di accesso eccessivamente lunga.

Le DSM rimpiazzano il bus con una rete di interconnessione scalabile con il numero di nodi. Inoltre, ogni nodo è connesso ad un frammento della shared memory, che dal punto di vista del nodo a cui è connessa, viene chiamata memoria locale. Un frammento della memoria che non si trova all'interno del nodo, viene chiamata memoria remota. Ciò significa che i tempi di accesso alla memoria possono variare in base alla posizione fisica della memoria rispetto al processore che cerca di accedervi. Per questo motivo, i sistemi DSM sono anche chiamati NUMA (Non unified memory access).

Inoltre, i meccanismi di gestione della memoria in un sistema DSM sono decentralizzati, poiché ogni processore ha il controllo della propria memoria locale. Ciò significa che ogni processore può allocare, deallocare e gestire la propria memoria locale senza interferire con gli altri processori.

Nei DSM, ogni CPU deve far girare la propria copia del sistema operativo, e i processi possono comunicare solo attraverso lo scambio di messaggi. Tutte le CPU vedono lo stesso spazio di indirizzamento ma, ogni processore è dotato di una sua propria memoria locale, vista anche da tutti gli altri processori. Esistono due tipi di sistemi NUMA:

- Cache-Coherent NUMA (CC-NUMA)
- Non-Caching NUMA (NC-NUMA)

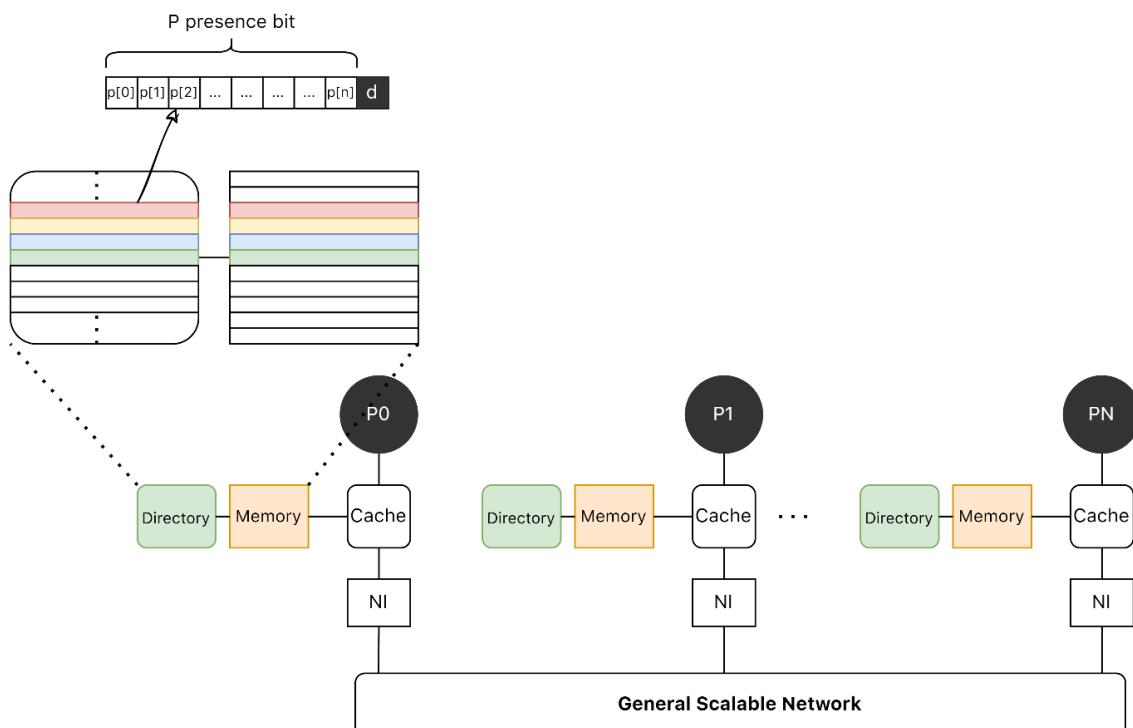


Questo approccio DSM è anche chiamato CC-NUMA (Cache Coherence Non Uniform Memory Access).

La distribuzione della memoria tra i nodi aumenta la memory bandwidth e riduce la latenza della memoria locale: separa il traffico della memoria locale dal traffico della memoria remota, riducendo le richieste sull'intero sistema di memoria e sulla rete di interconnessione. Però, a meno che non si elimini la necessità da parte del protocollo di coerenza di trasmettere i miss e i write della cache, la distribuzione della memoria ci farà guadagnare poco.

La soluzione è quella accennata in precedenza, ed è quella di utilizzare un protocollo directory-based anziché un protocollo basato su snoopers.

Nei protocolli directory based, ogni nodo ha una directory che consiste in una mappatura 1:1 con i blocchi di memoria locale.

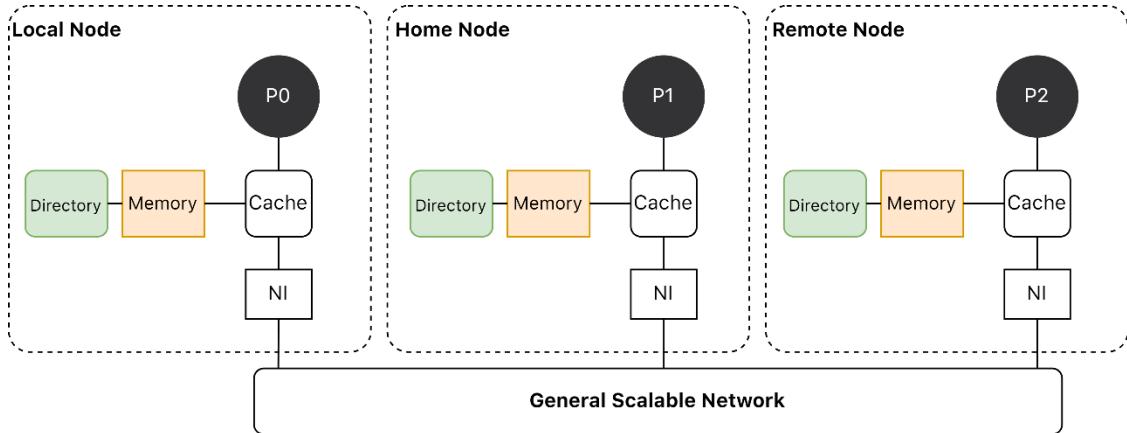


Ogni nodo ha una directory che mappa la propria memoria locale. Per ogni blocco in memoria c'è una entry nella directory che indica per quel blocco quali nodi posseggono una copia di quel blocco:  $p[i] = 1$  indica che il nodo 'i' possiede una copia di quel blocco. Inoltre, un dirty bit è presente per ogni entry e, se settato ad 1, indica che quel blocco è stato modificato e ci sarà un solo  $p[i] = 1$  che conterrà il dato modificato.

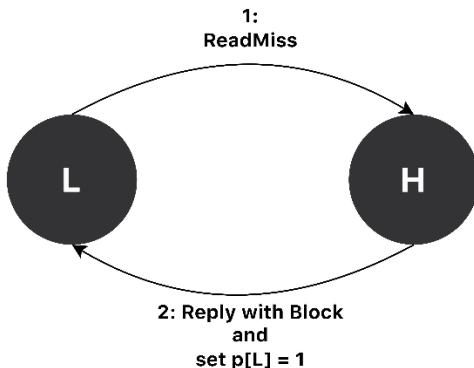
Nei protocolli full directory based, durante lo scambio di messaggi, vengono coinvolti 3 attori:

- **Local node:** è il nodo che effettua la richiesta
- **Home node:** è il nodo che contiene l'entry directory del blocco richiesto
- **Remote Node:** è il nodo (o nodi) che contiene la copia del blocco

Quindi, se ad esempio un Local Node effettua una richiesta di ReadMiss, questa richiesta viene inoltrata sulla rete e ricevuta dall'Home Node. Una volta ricevuta dall'Home Node, questo consulta la directory per controllare lo stato del blocco. Supponiamo che il blocco si trovi in un nodo  $p[i]$  casuale, la richiesta viene inoltrata al nodo  $p[i]$  che ritornerà la richiesta all'home node che a sua volta la ritornerà al local node.

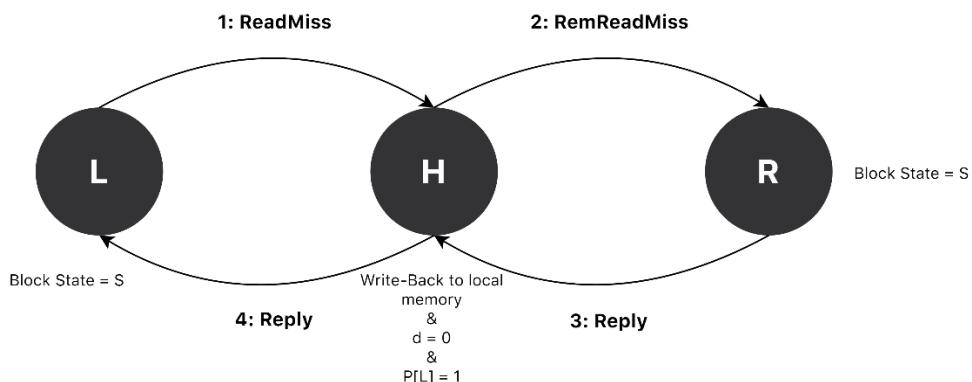


Come per i protocolli di snooping, bisogna tenere traccia dello stato di ogni blocco. Ad esempio usando MSI. Usando MSI, ad esempio, se un LocalNode effettua un **ReadMiss di clean block**:



In questo caso, il blocco è pulito e si trova solo nell'Home Node. Il Local Home inoltra la richiesta verso di esso, e l'Home Node imposta  $p[L] = 1$  ed inoltra il blocco. L'Home Node che riceve il blocco lo immagazzina nella cache privata con stato 'S' Shared.

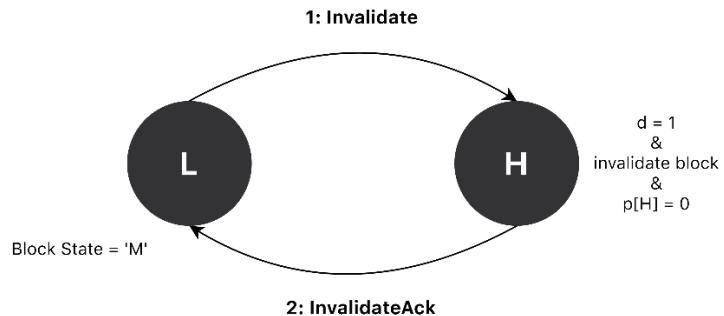
Mentre per un **ReadMiss di un dirty block**:



In questo scenario, il Local Node effettua un ReadMiss che inoltra all'Home Node, il quale consulta la directory, in particolare  $P = \{\}$  in cui ci sarà un certo  $p[i]$  che contiene il blocco modificato. L'Home Node inoltra la richiesta a  $p[i]$  Remote Node tramite una RemReadMiss (Remote Read Miss, necessaria al protocollo per disambiguare) il quale risponde con il blocco modificato all'Home Node e modifica lo stato di quel blocco nella sua cache privata da 'M' ad 'S'. L'Home Node che riceve il blocco modificato, effettua un write-back verso la sua memoria locale, ripulisce il dirty bit ed inoltra il blocco al Local Node impostando  $P[L] = 1$ . Il Local Node si salva il blocco in cache privata in stato 'S'.

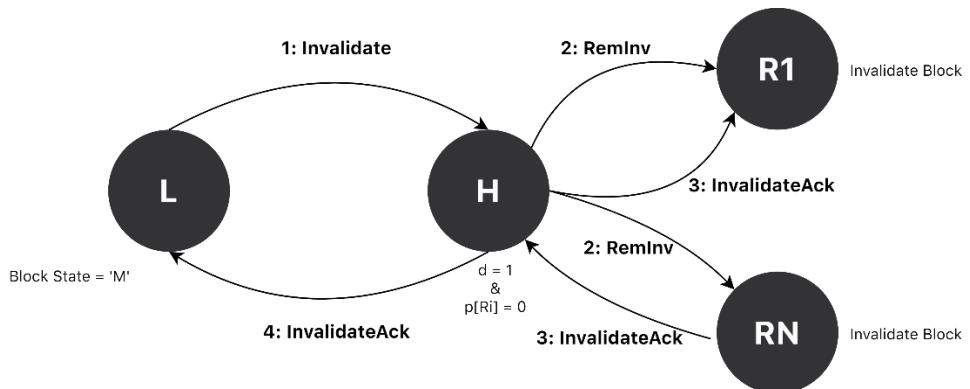
Quando il dirty bit è impostato, solo uno dei  $p[i]$  è impostato su 1, dato che solo uno dei Nodi del sistema può avere un blocco modificato con l'ultima modifica apportata.

Il caso di una **WriteHit** di un Local Node (che è l'unico a contenere il blocco) invece:



In questo caso, il Local Node ha già la copia del blocco ed effettua una scrittura, quindi WriteHit. Invia quindi all'Home Node un Invalidate. Quando l'Home Node la riceve, imposta il dirty bit ad 1, invalida il blocco nella propria cache privata (se presente) e si rimuove dai nodi possessori del blocco. L'Home Node invia un Ack al Local Node che a sua volta imposta il proprio blocco modificato sullo stato da 'S' ad 'M'.

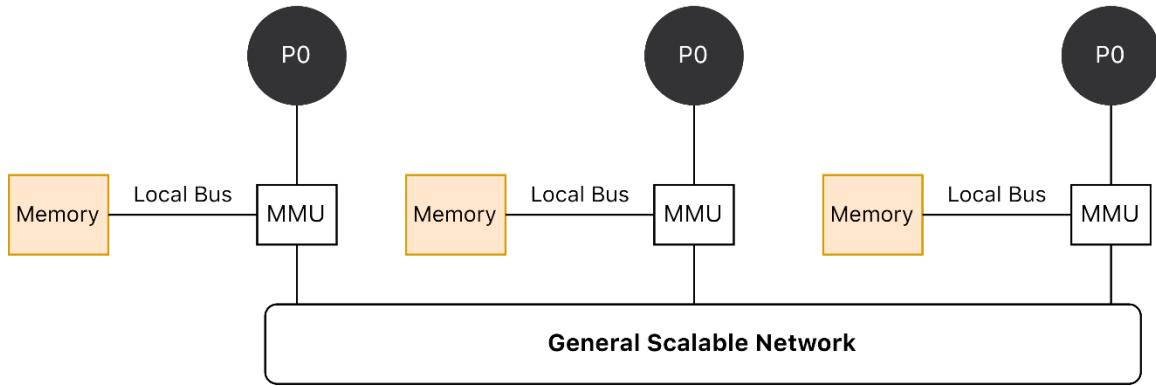
In caso di un WriteHit di un Local Node ma dove il blocco è contenuto in multipli Remote Node:



In questo caso le l'Home Node inoltra le Remote Invalidate ad ogni  $R_i$  che contiene il blocco condiviso.

## NC-NUMA

In un sistema NC-NUMA i processori non hanno cache locale. Ogni accesso alla memoria è gestito da una MMU dedicata, che controlla se la richiesta è diretta alla memoria locale o a un modulo remoto, nel qual caso la richiesta viene instradata al nodo che contiene il dato richiesto.



Evidentemente, i programmi che usano dati remoti (rispetto alla CPU su cui girano) risulteranno molto più lenti che se i dati fossero memorizzati nella memoria locale del processore su cui vengono eseguiti.

Nei sistemi NC-NUMA il problema della coerenza della cache non esiste perché non c'è alcuna forma di caching. Ogni dato nella memoria è presente esattamente in una locazione precisa e non viene mantenuta alcuna copia locale nei nodi.

Tuttavia, l'accesso alla memoria remota può essere meno efficiente rispetto all'accesso alla memoria locale. Per questo motivo, i sistemi NC-NUMA possono utilizzare software sofisticato per spostare le pagine di memoria da un modulo all'altro in modo da minimizzare gli accessi remoti e massimizzare le prestazioni complessive.

Un page scanner può essere utilizzato per analizzare le statistiche sull'indirizzamento della memoria e spostare le pagine di memoria tra i nodi per cercare di migliorare le prestazioni. Il page scanner può attivarsi ogni pochi secondi per rilevare le pagine di memoria utilizzate più di frequente e spostarle nei nodi con il processore che le utilizza maggiormente. In questo modo, è possibile ridurre il numero di accessi remoti alla memoria e migliorare le prestazioni complessive del sistema NC-NUMA.

In realtà, nei sistemi NC-NUMA, ogni processore può avere anche una memoria locale privata e una cache, e solo i dati privati del processore (ossia quelli nella memoria locale privata) possono risiedere nella cache. Questa soluzione aumenta ovviamente le prestazioni di ciascun processore, ed è adottata ad esempio nel Cray T3D/E. Tuttavia, il tempo di accesso ai dati remoti rimane molto alto, nel Cray T3D/E è di 400 cicli di clock del processore, contro solo due cicli necessari per accedere un dato nella cache locale.

## Sincronizzazione tra processi

In un sistema multiprocessore, i processi devono essere in grado di sincronizzarsi tra loro per evitare problemi come la concorrenza e la race condition, dove due o più processi cercano di accedere alle stesse risorse contemporaneamente.

Ma come fare in modo che due o più Thread non provino in modo simultaneo ad aggiornare variabili condivise, e quindi entrino in una zona critica in mutua esclusione? È necessario utilizzare delle primitive di sincronizzazione chiamati Lock.

Un Lock può assumere due valori:

- Zero: il lock è libero e il thread può entrare
- Uno: il lock è bloccato da un altro thread

E due operazioni possono essere applicate sui Lock:

- Acquire:
- Release: libera il lock, così che un altro thread possa entrare in sezione critica

Se dovessimo usare le istruzioni MIPS per implementare un lock, utilizzeremmo qualcosa di questo tipo:



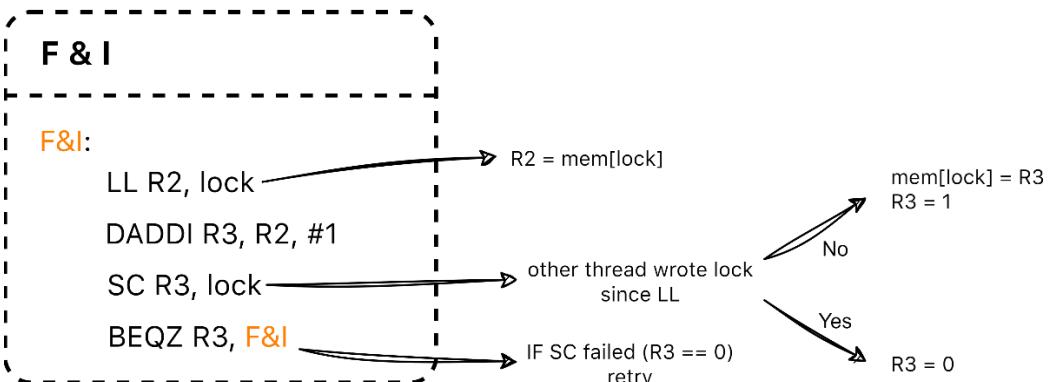
Questo purtroppo però, è un esempio errato di implementare uno spin-lock: si immagini ad esempio lo scenario in cui due thread entrino nella zona "acquire" nello stesso istante: entrambi i thread caricano il lock in R2, ed entrambi pensano che il lock sia libero perché la Store di un Thread non è ancora avvenuta prima della Load di un altro thread.

Il problema è che il codice tra la Load e la Store non è eseguito in modo atomico. È necessario quindi utilizzare delle operazioni macchina Atomiche chiamate anche Read-Modify-Write instructions (RMW).

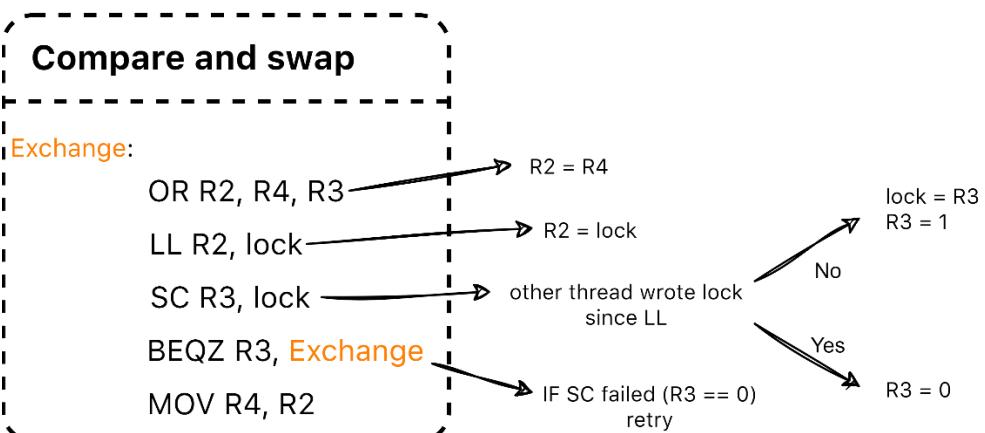
Le operazioni atomiche più utilizzate nelle architetture moderne vengono chiamate Load Linked e Store conditional. load-linked/store-conditional (LL/SC), a volte noto come load-reserved/store-conditional (LR/SC), sono una coppia di istruzioni macchina utilizzate nel multithreading per ottenere Read-Modify-Write atomiche:

- **Load-link:** restituisce il valore corrente di una posizione di memoria ,
- **Store-conditional:** nella stessa posizione di memoria, memorizza un nuovo valore solo se non si sono verificati aggiornamenti in quella posizione dal Load-Link precedente.

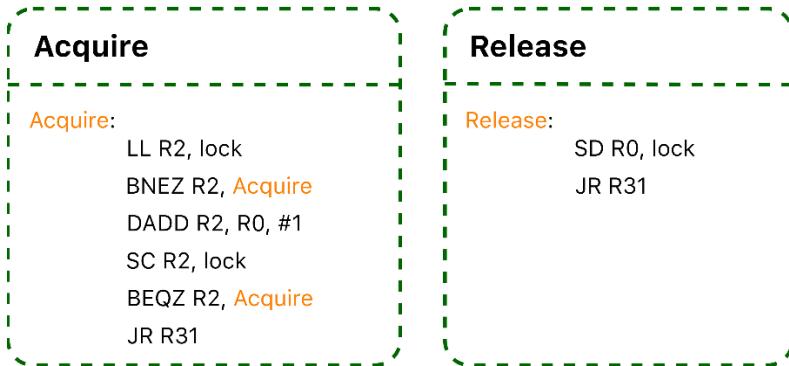
Con LL/SC è possibile implementare ad esempio alcune operazioni atomiche, come la fetch and increment (F&I) in modo atomico:



Oppure una compare and swap (o exchange):



Una volta ottenuta un'operazione atomica, possiamo utilizzare LL/SC per implementare gli spin lock:



A livello hardware, per implementare LL/SC, su un sistema multiprocessore SMP BUS based, il cache controller tiene traccia degli indirizzi utilizzati dalle istruzioni LL affiancandoci un bit LL.



In questo modo, quando si esegue un'istruzione LL, si crea una entry (LLbit-ADDRESS) in cui nell'LL Address viene inserito l'indirizzo a cui lavora la LL ed l'LL-bit viene impostato ad 1. Successivamente, lo snooper controlla le transazioni sul bus. Se si accorge che qualche Store di qualche altro thread modifica lo stesso indirizzo, azzera l'LL-bit. In questo modo, quando il thread corrente arriva alla SC, controlla l'LL-bit:

- Se == 0: qualche altro thread ha scritto sullo stesso indirizzo della LL precedente alla SC, errore.
- Se == 1: nessuno ha scritto, la SC ha successo.

## Consistenza della memoria

In un sistema multiprocessore tutte le CPU hanno accesso ad uno spazio di memoria primaria condiviso, e quando una CPU modifica una cella di memoria, un'altra CPU vedrà prima o poi quella modifica. Dunque, un altro aspetto fondamentale di cui si deve occupare l'hardware che fa funzionare il sistema è il seguente: in che ordine una CPU deve vedere le modifiche effettuate sulle celle di RAM effettuate dalle altre CPU?

La questione non è banale, dato che il sistema può essere costituito da centinaia di CPU e centinaia o migliaia di banchi di memoria, il tutto messo in comunicazione da reti a tipologia diversa, con conseguenti tempi variabili di completamento delle scritture/lettura in RAM

Persino in un semplice sistema UMA con poche CPU che si affacciano su un unico bus di comunicazione, il problema esiste.

Consideriamo infatti tre CPU che eseguono nell'ordine, in rapida sequenza, le seguenti operazioni sulla cella di memoria X:

- CPU A: write #1, X
- CPU B: write #2, X
- CPU C: read X

Che valore legge CPU C? In mancanza di assunzioni a priori, potrà leggere 1, 2 o addirittura il valore precedente alla prima scrittura, a seconda della distanza delle tre CPU rispetto al banco di memoria che contiene la cella indirizzata.

La forma più ovvia di consistenza è che qualsiasi lettura ad una cella X restituisce sempre il valore dovuto alla scrittura più recente effettuata nella stessa cella: è la cosiddetta consistenza stretta. Sfortunatamente, è anche la forma di consistenza più difficile e inefficiente da garantire: è necessaria

un'interfaccia tra le varie CPU e la memoria che gestisca tutte le richieste di accesso alla memoria stessa in modalità first come first served

La memoria diviene così un collo di bottiglia che rallenta in modo inaccettabile un sistema costruito per lavorare il più possibile in parallelo. Questa forma di consistenza non viene di fatto implementata.

Una forma di consistenza molto meno vincolante della consistenza stretta, e dunque spesso implementata, è la consistenza del processore, descritta dalle seguenti due proprietà:

1. Le scritture da parte di una qualsiasi CPU sono viste dalle altre CPU nell'ordine in cui sono state avviate. Se CPU 1 scrive A, B e C in una locazione var, una CPU 2 che legga var in sequenza più volte leggerà prima A, poi B e poi C.
2. Per ogni locazione di memoria, qualsiasi CPU vede tutte le scritture effettuate da ogni singola CPU in quella locazione nello stesso ordine.

Vediamo con un esempio cosa significano queste due condizioni: poniamo che CPU 1 scriva in una locazione var A, B e C, mentre CPU 2, concorrentemente con CPU 1, scrive in var X, Y e Z.

Diverse CPU che leggano più volte var potranno leggere sequenze diverse. Ciò che viene garantito è che nessuna CPU vedrà una sequenza in cui, ad esempio, B viene prima di A o Z viene prima di Y: l'ordine in cui ciascuna CPU esegue le sue scritture, viene visto allo stesso modo da tutte le altre.

Quindi, ad esempio, CPU 3 potrà leggere: A, B, C, X, Y, Z. Mentre CPU 4 potrà leggere X, Y, Z, A, B, C

Ma nessuna CPU potrà leggere una sequenza come B, A, C, X, Y, Z

## Gestione dei processi negli SMP

Nei sistemi SMP (shared-memory multiprocessing), ogni processore è collegato allo stesso bus di sistema e può accedere alla stessa memoria condivisa, di conseguenza, è necessario che anche il sistema operativo supporti sistemi SMP. Ad oggi ormai tutti i moderni sistemi operativi, come Windows, Linux, MacOS e Solaris, supportano la tecnologia SMP.

In un sistema SMP, i processi "ready to run" possono essere gestiti in due modi diversi:

1. Tutti i processi sono inseriti in una sola coda di attesa e ogni processore accede alla coda per selezionare il prossimo processo da eseguire.
2. Ogni processore ha la propria coda di attesa separata, in modo che i processi vengano distribuiti tra i vari processori in modo equilibrato.

Tra le due opzioni viene ovviamente privilegiata la seconda: non ha infatti senso avere un sistema con più CPU se poi i vari processi da eseguire non sono distribuiti più o meno omogeneamente tra i vari processori. Inoltre, nella coda unificata, il bilanciamento del carico è solitamente automatico: quando un processore è inattivo, il suo scheduler prenderà un processo dalla coda comune e lo manderà in esecuzione su quel processore. In questo modo però è necessario sincronizzarsi per accedere all'unica coda di ready quando devono inserire o prelevare un processo dalla coda, per evitare che un processo vada in esecuzione su due processori distinti.

L'obiettivo principale della tecnologia SMP però, è quella di bilanciare il carico di lavoro tra i processori in modo equo, in modo che ogni processore venga utilizzato al massimo delle sue capacità, quindi la seconda opzione è quella adottata.

Per applicare la seconda opzione però, i moderni sistemi operativi SMP utilizzano un meccanismo di bilanciamento del carico che può prendere un processo in attesa nella coda di un processore sovraccarico e spostarlo nella coda di un processore scarico. Ad esempio, in Linux SMP, il meccanismo di bilanciamento del carico si attiva ogni 200 millisecondi, o ogni volta che si svuota la coda di un processore.

Notare che la migrazione di un processo da un processore ad un altro può non essere conveniente nel caso in cui ciascun processore abbia una cache privata: migrando su un altro processore infatti, il

processo genererebbe inizialmente un elevato numero di cache miss. Per questa ragione, alcuni SO, come Linux, mettono a disposizione delle system call per poter specificare che un processo non deve cambiare processore, indipendentemente dal carico del processore stesso.

Nel caso di processori multi-core , l'architettura SMP si applica ai core, trattandoli come processori separati.

**GPU**