



ACSO

Architettura dei Calcolatori



AA 2016/2017
ANDREA DONATI
Ing. Inf. Politecnico di Milano: CM – IM
Prof. Luca Breveglieri Oddone

Linguaggio macchina del processore MIPS

Ogni architettura di processore ha il suo linguaggio macchina: ISA.

ISA: insieme delle istruzioni (instruction set) che possono essere interpretate direttamente dall'architettura del processore.

E' necessario definire:

- elementi a disposizione delle istruzioni macchina: modello della memoria e registri
- insieme delle istruzioni macchina
- formato e dimensione
- riferimenti agli operandi
- in memoria e/o nei registri del processore
- tipi di indirizzamento
- tipi di dati e dimensioni
- modalità operative

Linguaggio Assemblatore:

Linguaggio simbolico che consente di programmare un calcolatore utilizzando le istruzioni del linguaggio macchina. Le istruzioni del linguaggio assemblatore (ASSEMBLER) sono in corrispondenza (quasi) 1:1 con quelle del linguaggio macchina, tranne le **pseudo-istruzioni**, che vengono tradotte attraverso insiemi di istruzioni (ad esempio: *move ; li (load immediate)*).

Il linguaggio simbolico consente di rappresentare istruzioni, registri, dati e riferimenti alla memoria senza usare gli indirizzi numerici esadecimali, rendendo la programmazione più semplice per il programmatore e il programma human-readable.

Una volta scritto il programma in ASSEMBLER, perché il processore possa interpretarlo è necessaria una traduzione in linguaggio macchina, che viene effettuata dall'**assemblatore**. Se nel codice sono presenti etichette e riferimenti simbolici definiti in moduli esterni a quello assemblato o che dipendono dalla rilocazione del modulo è necessario anche l'intervento del **linker**.

Istruzioni e variabili in linguaggio macchina:

Le **ISTRUZIONI** sono costituite da:

- codice operativo (opcode) che identifica in modo univoco l'istruzione e definisce l'operazione associata
- riferimento/i all'operando/i su cui agisce l'istruzione. Il riferimento può essere
 - implicito nel codice operativo:
 - direttamente il valore numerico dell'operando
 - un registro del processore
 - in modo diretto o indiretto una locazione di memoria

Le **VARIABILI** accessibili dal processore sono di due tipi:

- riferimento: rappresentano un indirizzo di memoria (o di registro)
- valore: codifica opportuna del contenuto della parola di memoria associata all'indirizzo contenuto nella variabile (tipo puntatore).

Architettura del processore MIPS:

Il processore MIPS è una CPU del tipo RISC (*Reduced Instruction Set Computer*), il cui obiettivo è semplificare il più possibile le istruzioni e la struttura della CPU in modo da massimizzare la frequenza di completamento delle istruzioni.

La dimensione delle istruzioni infatti è fissa, il che rende molto veloce il fetch.

ACSO - Architettura dei Calcolatori

E' un processore con architettura LOAD/STORE: gli operandi della ALU non possono provenire dalla memoria ma devono essere prima inseriti nei registri del processore tramite le istruzioni di lettura (**load**). Una volta terminata l'operazione, se essa produce un risultato e lo scrive in un registro, tale risultato può essere salvato in memoria tramite l'operazione di scrittura (**store**).

Memoria – Struttura e Indirizzamento:

La memoria è indirizzata a byte, le parole di memoria occupano 4 bytes (32 bits).

L'indirizzo di una parola di memoria è a 32 bits, quindi lo spazio di indirizzamento a byte è di 4 GByte (1 Gparola).

Siccome la memoria è indirizzabile a bytes e una parola ha dimensione 4 bytes, gli indirizzi delle parole di memoria sono multipli di 4. La parola assume l'indirizzo che ha il byte più significativo all'interno di essa.

Indirizzo di byte

Parola 0	0	1	2	3
Parola 4	4	5	6	7
Parola 8	8	9	10	11
...	MS byte			LS byte
Parola 2^k-4	2^{k-4}	2^{k-3}	2^{k-2}	2^{k-1}

big-endian alla parola viene assegnato lo stesso indirizzo del suo byte più significativo

Registri:

I registri in architettura MIPS sono elementi interni alla CPU composti da parole di memoria da 32 bit. Una CPU con architettura MIPS ha 32 registri da 32 bit, accessibili anche a byte (load upper immediate) organizzati in un banco di registri. Per trovare un registro in mezzo a tutti è possibile apporre un \$ al "nome" del registro in assembler, mentre è identificato da 5 bit in linguaggio macchina.

I diversi banchi di registri (Address, Temporary...) possono avere un utilizzo specifico in termini di convenzione di programmazione, ma sono tutti trattati in modo omogeneo.

Istruzioni, formato delle istruzioni e modalità di indirizzamento in MIPS:

Classi di istruzioni tipiche in linguaggio macchina:

- aritmetico-logiche
- trasferimento da e in memoria (e tra registri)
- modifica del flusso di esecuzione: salto condizionato, incondizionato
- salto a sottoprogramma, ritorno da sottoprogramma
- trasferimento dati da e in periferica (istruzioni di I/O)
- istruzioni speciali (controllo)

Tutte le istruzioni in linguaggio macchina hanno la stessa dimensione (**dimensione fissa: 32 bit**) e quindi occupano una parola di memoria ciascuna.

Ogni istruzione ha un diverso codice operativo (**6 bit**) ed è così che vengono riconosciute dal processore. Le istruzioni MIPS sono di 3 formati:

- (**R**) Register: istruzioni aritmetico-logiche con operandi esclusivamente registri
- (**I**) Immediate: istruzioni di accesso alla memoria o istruzioni aritmetico-logiche contenenti almeno un valore costante
- (**J**) Jump: Istruzioni di salto condizionato/incondizionato.

Le modalità di indirizzamento previste in linguaggio macchina MIPS sono:

- Immediato
- A registro
- Con base e spiazzamento
- Relativo al Program Counter
- Pseudo-diretto (rispetto al Program Counter)

La modalità di indirizzamento è associata in modo univoco al formato dell'istruzione (cioè a come vengono interpretati i bit nell'istruzione in linguaggio macchina):

- Tipo R (register): a registro
- Tipo I (immediate): immediato, con base e spiazzamento, relativo al Program Counter
- Tipo J (jump): pseudo-diretto

Una singola istruzione «logica» può usare più di una modalità di indirizzamento, ad es. **add** e **addi** (che sono di due formati diversi, R e I rispettivamente e hanno anche due codici operativi diversi).

Le Istruzioni in MIPS:

Istruzioni aritmetico-logiche:

In MIPS un'istruzione aritmetico-logica ha 3 operandi:

- 2 registri sorgente
- 1 registro destinazione

In assemblatore il formato istruzione è

OPCODE DEST, SORG1, SORG2

dove **DEST, SORG1, SORG2** sono registri referenziabili del MIPS

Le istruzioni aritmetico-logiche sono: AD, SUB, ADDI, ADDU, ADDIU, SLL, SRL, AND, OR con tutte le varianti immediate/unsigned.

Istruzioni di trasferimento dati LOAD/STORE:

- LW: Load Word, trasferisce una parola di memoria in un registro.
- SW: Store Word, trasferisce il contenuto di un registro scrivendolo in una parola di memoria.

Entrambe le operazioni hanno 2 operandi:

- Registro destinazione (lw) o sorgente (sw)
- Indirizzo parola di memoria destinazione/sorgente.

NOTA: SW è l'unica istruzione di MIPS che usiamo che ha come primo operando una SORGENTE dell'operazione, mentre tutte le altre istruzioni hanno come primo operando l'indirizzo di destinazione.

L'**indirizzo** della parola di memoria coinvolta nel trasferimento viene sempre specificato attraverso la modalità **offset(registro_base)**:

- Offset è uno spiazzamento su 16 bit (256 byte)
- L'indirizzo della parola di memoria è dato dalla somma tra il valore immediato offset e il contenuto del registro base, che di solito è il GLOBAL POINTER (registro che punta a metà dell'area dati).

Tuttavia, in MIPS possiamo esprimere l'indirizzo di destinazione in modo più flessibile, sarà l'assemblatore a occuparsi di correggere/interpretare/completare il campo.

Istruzioni di salto condizionato e incondizionato:

Alterano l'ordine di esecuzione delle istruzioni, permettono di realizzare i costrutti di controllo condizionali e ciclici.

Nell'istruzione di salto in ASSEMBLER l'istruzione di destinazione del salto viene specificato tramite un nome simbolico: un'etichetta (*label*) associata all'istruzione di destinazione.

In MIPS le istruzioni di salto **condizionato** sono di tipo I, mentre quelle di salto **incondizionato** sono di tipo J

Salto condizionato:

- **BEQ** (Branch EQual): BEQ r1, r2, label -> Salta all'istruzione associata all'etichetta se i valori contenuti nei registri r1 e r2 sono uguali
- **BNE** (Branch Not Equal): BNE r1, r2, label -> Salta all'istruzione associata all'etichetta se i valori contenuti nei registri r1 e r2 *NON* sono uguali.

Salto incondizionato:

- **J** (Jump): semplice salto incondizionato (goto)
- **JR** (Jump Register): salto di ritorno da sottoprogramma (return)
- **JAL** (Jump And Link): salto di chiamata di sottoprogramma

Nella traduzione dal C, questi salti si usano per tradurre costrutti if-then-else, switch, cicli for e while.

Formato delle istruzioni in MIPS:

Come già detto, tutte le istruzioni sono composte da 32 bit suddivisi in codice operazione e operandi. Ma ogni specifico tipo di istruzione ha diversi campi e diversi utilizzi di essi, di seguito un riassunto:

Formato istruzioni di tipo R - aritmetiche

Formato usato per istruzioni aritmetico-logiche

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

Dove **shamt** sta per shift amount (scorrimento) e **funct** sta per function ed indica per ogni tipo di istruzione la variante specifica. Ad esempio ADD e ADDI hanno lo stesso OPCODE, ma un diverso parametro funct.

Formato istruzioni di tipo I – lw/sw

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

Nel caso di istruzioni load/store, i campi hanno il seguente significato:

- **op (opcode)** identifica il tipo di istruzione; (**35 o 43**)
- **rs** indica il registro base;
- **rt** indica il registro destinazione dell'istruzione load o il registro sorgente dell'istruzione store;
- **indirizzo** riporta lo spiazzamento (offset)

Con questo formato, un'istruzione lw(sw)può indirizzare parole nell'intervallo $-2^{15} \dots +2^{15}-1$ rispetto all'indirizzo base

Formato istruzioni di tipo I – con immediato

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

Nel caso di istruzioni **con immediati**, i campi hanno il seguente significato:

- **op (opcode)** identifica il tipo di istruzione;
- **rs** indica il registro sorgente;
- **rt** indica il registro destinazione;
- **indirizzo** contiene il valore dell'operando immediato

Con questo formato, un'istruzione con immediato può contenere costanti nell'intervallo $-2^{15} \dots +2^{15}-1$

Formato istruzioni di tipo I - branch

op	rs	rt	indirizzo
6 bit	5 bit	5 bit	16 bit

Nel caso di salti condizionati, i campi hanno il seguente significato:

- **op (opcode)** identifica il tipo di istruzione (**4 = beq**)
- **rs** indica il primo registro;
- **rt** indica il secondo registro;
- **indirizzo** riporta lo spiazzamento (offset)

Per l'offset si hanno a disposizione solo 16-bit del campo **indirizzo** ⇒ rappresentano un indirizzo di **parola** relativo al PC (**PC-relative word address**)

Istruzioni di salto condizionato (tipo I)

I 16-bit del campo indirizzo esprimono **l'offset** rispetto al PC rappresentato in complemento a due per permettere salti in avanti e all'indietro

L'offset varia tra **-2¹⁵** e **+2¹⁵-1**

Esempio: **bne \$s0, \$s1, L1**

L'assemblatore sostituisce l'etichetta **L1** con l'offset **di parola** relativo a PC: **(L1- PC)/4**

- PC contiene già l'indirizzo dell'istruzione successiva al salto
- La divisione per 4 serve per calcolare l'offset di parola

Il valore del campo **indirizzo** può essere negativo (salti all'indietro)

Formato istruzioni di tipo J

È il formato usato per le istruzioni di salto incondizionato (*jump*), per esempio **j L1**

op	indirizzo
6 bit	26 bit

In questo caso, i campi hanno il seguente significato:

- **op (opcode)** indica il tipo di operazione **(2)**
- **indirizzo** (composto da **26-bit**) riporta una parte (26 bit su 32) dell'indirizzo **assoluto** di destinazione del salto

I 26-bit del campo **indirizzo** rappresentano un indirizzo di parola (**word address**)

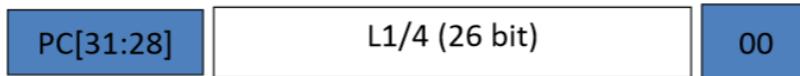
Istruzioni di salto incondizionato (tipo J)

L'assemblatore sostituisce l'etichetta **L1** con i 28 bit meno significativi traslati a destra di 2 (divisione per 4 per calcolare l'indirizzo di parola) per ottenere 26-bit

- in pratica elimina i due 0 finali
- si amplia lo spazio di salto:
 - si salta tra 0 e 2^{28} byte (2^{26} word)

I 26-bit di indirizzo nelle jump rappresentano un indirizzo di parola (word address) \Rightarrow corrispondono ad un indirizzo di byte (byte address) composto da 28 bit.

Poiché il registro PC è composto da 32 bit \Rightarrow l'istruzione jump rimpiazza solo i 28 bit meno significativi del PC, lasciando inalterati i rimanenti 4 bit più significativi.



C'è un problema, però, sulla gestione delle costanti su 32 bit nelle istruzioni di tipo I, essendoci solo 26 bit di spazio, ad esempio, per l'istruzione di load.

Esiste la pseudo istruzione load immediate (LI), ma è, appunto, una pseudo istruzione che viene tradotta con un load upper immediate (LUI) che carica i 16 bit più significativi dei 32 bit della costante da caricare, successivamente viene fatto un ORI e siccome i primi 16 bit sono 0 vengono caricati i 16 bit meno significativi della costante da 32 bit in questione. Siccome vengono caricati in un registro questo poi è pienamente utilizzabile per eseguire qualsiasi tipo di operazione.

La pseudo-istruzione LI è del tutto simile alla pseudo-istruzione LA (Load Address), l'unica differenza tra le due è che la prima andrebbe usata con le costanti e la seconda con gli indirizzi, ma la traduzione è identica. (Un'altra possibile variazione è quella del registro di destinazione che nel primo caso potrebbe essere un registro del banco S o del banco T, mentre nel secondo caso dovrebbe essere del banco A).

Struttura del programma e direttive dell'assemblatore:

Un programma in esecuzione ha 3 segmenti essenziali:

- Codice: main e funzioni utente
- Dati: variabili globali e dinamiche
- Pila: aree di attivazione con indirizzi, parametri, registri salvati e variabili locali

Codice e Dati sono segmenti dichiarati nel programma, mentre il segmento "Pila" (STACK) viene creato al lancio del processo e aggiornato durante l'esecuzione.

Direttive del compilatore:

Per dichiarare i diversi segmenti del codice si usano delle direttive per l'assemblatore. Queste non sono propriamente istruzioni, servono solo per informare il compilatore che quanto segue va interpretato, in fase di compilazione, in un modo piuttosto che in un altro.

L'assemblatore usa queste direttive per generale il file oggetto.

Le variabili, convenzioni e metodi di dichiarazione:

In generale le variabili del programma sono collocate in posti diversi in memoria a seconda del loro "ruolo" nel programma:

- Variabili globali in memoria ad un indirizzo fissato
- Variabili locali nei registri del processore
- Parametri passati ad un modulo nell'area della pila
- Dinamiche in memoria centrale

Siccome le istruzioni aritmetico-logiche operano sui registri, per operare con una variabile è necessario copiarla in un registro, se non è già presente.

ACSO - Architettura dei Calcolatori

In MIPS la variabile è un elemento di memoria (byte, parola, regione di memoria), ha una “collocazione” con NOME e MODO per indirizzarla, e viene manipolata tramite indirizzo simbolico (come con un’etichetta) o tramite il nome del registro in cui risiede.

E’ necessario però distinguere i metodi di dichiarazione e trattamento dei diversi tipi di variabili (globale, locale, parametro).

- Variabile GLOBALE:
collocata in memoria ad un indirizzo fisso (stabilito da assemblatore e linker).
L’indirizzo simbolico della variabile è il suo **nome** e viene assegnato come un’etichetta.
L’ordine di dichiarazione e l’ordine di disposizione in memoria delle variabili coincidono.
Le variabili globali sono allocate in memoria a partire dall’indirizzo 0x1000 0000.
- Parametri in ingresso ad una funzione e valore restituito:
I primi 4 parametri vanno passati nei primi 4 registri del banco A (a0, a1, a2, a3). Questo tipo di passaggio funziona per tutti i tipi di dati, perché i registri sono a 32 bit, quindi posso passare scalari e puntatori (nei puntatori è compreso il vettore). Per il passaggio di tipi di dati complesso (*struct*) il metodo è complesso e vari da compilatore a compilatore.
Se una funzione ha più di 4 parametri (raro), essi vengono inseriti nella pila dal chiamante prima di eseguire il salto di chiamata.
Il valore restituito va inserito nel registro v0, se è di tipo double si usa anche v1.
- Variabile LOCALE:
La variabile locale ha diversi metodi di gestione che dipendono dal tipo e dall’utilizzo che ne viene fatto nel codice.
 - _ Una variabile scalare o puntatore può essere inserita in un registro del banco S (da s0 a s7), oppure nell’area di attivazione della funzione nella pila.
 - _ Una variabile che viene acceduta tramite l’utilizzo di un puntatore va necessariamente inserita dell’area di attivazione della funzione nella pila perché per l’utilizzo del puntatore richiede che la variabile che punta abbia un indirizzo.
 - _ Una variabile di tipo array nella pila.

Sottoprogrammi (funzioni):

Nei linguaggi di alto livello, per esempio in linguaggio C, la chiamata a sottoprogramma ha come effetto la creazione di un’area (o record) di attivazione sulla pila (stack). È importante notare che il frame (l’area di attivazione) è allocata con dimensioni diverse. Non c’è uno standard di dimensione fissa per ogni area di attivazione, ma può sempre essere diverso a seconda del tipo di funzione. Anche le singole parti di area di attivazione contenenti le informazioni necessarie sono di grandezza diversa.

A ogni chiamata di sottoprogramma viene creata un’area di attivazione, quando il sottoprogramma termina l’area viene rilasciata dalla pila.

Con sottoprogrammi annidati le aree vengono tutte messe sulla pila e l’ultima messa (ossia quella in cima alla pila) corrisponde al sottoprogramma correntemente in esecuzione. (LIFO)

L’area di attivazione di main è la prima ad essere creata quando il programma viene lanciato e l’ultima a essere rilasciata (deallocata).

L’area di attivazione del sottoprogramma è associata in modo opportuno alla informazioni seguenti:

- Parametri formali (con i loro valori) passati al sottoprogramma
- Indirizzo di ritorno al programma (o sottoprogramma chiamante)
- Informazioni per gestire lo spazio allocato per l’area di attivazione
- variabili locali del sottoprogramma
- valore restituito al programma (o sottoprogramma) seguente

A livello ISA la chiamata a sottoprogramma è espressa in più istruzioni, eseguite da i diversi “processi” coinvolti:

- il Chiamante (**caller**) gestisce la parte relativa al passaggio dei parametri e attiva il sottoprogramma tramite ***l'istruzione di chiamata ISA***
- all'esecuzione dell'istruzione di chiamata viene autonomamente gestita la parte relativa al salvataggio dell'indirizzo della istruzione successiva da eseguire alla fine del sottoprogramma (PC)
- il chiamato (**callee**) gestisce l'allocazione delle variabili locali e del valore restituito

Modello di chiamata a sottoprogramma in MIPS:

L'ISA di MIPS è implementata in modo tale da strutturare rigidamente il modello di chiamata e ritorno da funzione:

- Vincola il salvataggio dell'indirizzo di ritorno tramite l'istruzione di chiamata (non deve implementarlo il programmatore)
- Definisce delle convenzioni per il passaggio di parametri e del valore restituito
- Caratterizza i gruppi di registri in base al fatto che i valori corrispondenti siano o meno da considerare preservati dal chiamato. In tal caso il callee deve salvare i valori di tali registri per poi restituirli integri.

Istruzione di chiamata a sottoprogramma: JAL label (Jump And Link)
 -> \$ra = PC+4 (parola successiva a quella corrente del PC)
 PC = indirizzo corrispondente a label

Istruzione di ritorno da sottoprogramma: JR \$ra (Jump Register)

Passaggio dei Parametri: Per il passaggio dei parametri si veda la parte relativa alle convenzioni e dichiarazioni delle variabili

Convenzioni per il salvataggio dei registri:

L'esecuzione di un sottoprogramma non deve compromettere l'esecuzione del modulo chiamante.

Lo stato dei registri precedente alla chiamata deve poter essere ripristinato al termine dell'esecuzione.

Gestione della pila in MIPS:

Esiste un registro che punta sempre all'inizio dell'area di attivazione del programma corrente: **\$sp**.

L'**inserimento di un dato nella pila** avviene decrementando il registro \$sp per allocare lo spazio, ed eseguendo l'istruzione **sw** per inserire il dato.

Il **prelevamento di un dato nella pila** (operazione di pop) avviene eseguendo l'istruzione **lw** e incrementando il registro \$sp (per eliminare il dato), riducendo così la dimensione della pila.
 Naturalmente per sapere di quanto decrementare il registro \$sp è necessario calcolare la dimensione in bytes dell'area di richiesta.

Convenzioni ACSO: il salvataggio di un qualsiasi registro viene fatto solo se necessario. Ad esempio:

- fp è salvato solo se lo si usa per referenziare parole di memoria in pila
- ra non viene salvato in procedure foglia.

In conclusione deve esserci questo schema nella chiamata a funzione:

- Prologo del chiamante: si esegue il salvataggio dei parametri da passare al calle di registri \$a0-\$a3, si salano sulla pila i registri dei banchi T e V che si vogliono avere inalterati, se necessario salva sulla pila anche gli argomenti \$a0-\$a3, per mantenerli preservati.
- Salto di chiamata.
- Prologo del chiamato: crea l'area di attivazione (addiu \$sp, \$sp, -dimensione area in byte), salva il registro \$fp se esso è in uso, salva \$ra se non è una funzione foglia, salva sulla pila i registri \$s0-\$s7 assegnati a variabili locali.
- Corpo elaborativo del chiamato: contiene le istruzioni del chiamato
- Rientro a chiamante
- Epilogo del chiamante: ripristina dalla pila: i registri \$a0-\$a3 se erano stati preservati, i registri del banco T che erano stati preservati, trova nel registro \$v0 il valore di uscita dalla funzione chiamata.

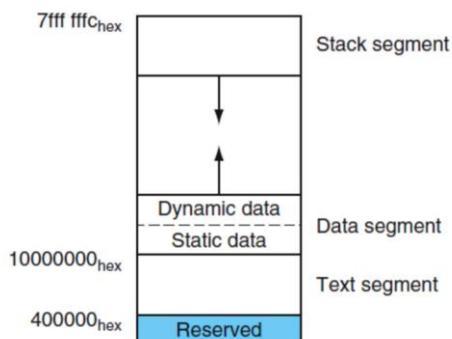
Assemblatore e collegatore (Linker):

- 1) Per ogni file sorgente (possono essere più di uno, suddivisi in moduli tipo C) l'assemblatore genera un file oggetto. I riferimenti simbolici vengono trasformati negli indirizzi corrispondenti e le pseudo istruzioni vengono espanso. Viene formata la tabella dei simboli che contengono gli indirizzi associati alle etichette.
- 2) Nel modulo oggetto assemblato gli indirizzi delle istruzioni con etichetta vengono messi tutti a 0x000000000 (perché non sono conosciuti), mentre successivamente nella tabella di rilocazione viene tenuta traccia delle etichette con indirizzi delle istruzioni diverse.
- 3) Ogni modulo ha il primo indirizzo di offset(byte) istruzione/variabile a 0x00000000: il LINKER si occupa di tenere in memoria RAM la successione dei moduli dove:
 - Il primo modulo parte da 0x00400000 per le istruzioni e da 0x10000000 per le variabili
 - il secondo modulo parte da 0x00400000 + spazio usato dal primo per le istruzioni e ci aggiunge le sue istruzioni, mentre per le variabili parte da 0x10000000 + spazio usato dal primo, e ci aggiunge le sue variabili.
 - per i moduli successivi il procedimento è identico

Il **file oggetto** creato dall'assemblatore alla fine della sua esecuzione contiene:

- Intestazione: descrive le dimensioni del segmento di testo e del segmento dati del modulo
- Segmento testo: codice in linguaggio macchina che potrebbe essere non completamente eseguibile a causa di riferimenti non risolti
- Segmento dati statici: rappresentazione binaria dei dati definiti nel file sorgente. Anche questa parte potrebbe essere incompleta causa etichette irrisolte.
- Informazioni di rilocazione: identificano le istruzioni e le parole di dati che dipendono da **indirizzi assoluti all'interno del file eseguibile** del programma completo. La posizione di queste istruzioni e dati deve essere modificata se parti del programma vengono spostate in memoria
- Tabella dei simboli: associa un indirizzo alle etichette e contiene l'elenco dei riferimenti non risolti nel modulo
- Informazioni di debug

Organizzazione della memoria:



Accesso all'area dati statici:

Si può fare ottenendo l'indirizzo con la composizione di *lui* e *lw*, ma si può molto più semplicemente usare il registro **global pointer \$gp**, che è inizializzato all'indirizzo 0x10008000 (metà dei primi 64 kB dell'area dati statici), operando con esso mediante la modalità di indirizzamento offset(\$gp). Questa operazione è molto più veloce della composizione delle due precedenti, tuttavia velocizza e ottimizza solamente i tempi di accesso ai primi 64 kB, per questo in quest'area vengono posizionate le variabili con valore scalare.

Processo di collegamento – Le operazioni del Linker:

Il **linker** ha il compito di generare un solo programma binario eseguibile (in formato rilocabile) partendo dai file oggetto dei diversi moduli. Ha l'obiettivo di creare un solo spazio di indirizzamento per tutto il programma.

Opera in base a queste informazioni:

- Lunghezza del segmento testo e segmento dati
- Indirizzi di impianto

Il collegatore (linker) calcola gli indirizzi dei riferimenti non risolti e completa la traduzione delle istruzioni presenti nelle tabelle di rilocazione.

Le operazioni svolte dal calcolatore sono:

- Determinare la posizione in memoria (indirizzo base) delle sezioni di codice e dati dei diversi moduli
- Determinare il nuovo valore di tutti gli indirizzi simbolici che risultano modificati dallo spostamento della base, ossia creare una tabella globale dei simboli
- Correggere in tutti i moduli i riferimenti agli indirizzi simbolici che sono stati modificati, in base alle tabelle di rilocazione

1 – Determinazione della posizione in memoria dei moduli:

L'assemblatore alloca la sezione testo e la sezione dati a partire dall'indirizzo base 0, ma i moduli non possono essere tutti caricati nella stessa zona di memoria e in più devono rispettare la suddivisione di essa come da convenzione. Il linker quindi rialloca la sezione testo a partire dall'indirizzo 0x0040 0000 e la sezione dati a partire dall'indirizzo 0x1000 0000. I diversi moduli devono inoltre essere ordinati in modo sequenziale.

2 – Creazione della tabella globale dei moduli:

E' costituita dall'unione delle tabelle dei simboli di tutti i moduli da collegare, modificati (riallocati) in base all'indirizzo di base del modulo a cui appartengono. Una variabile che prima, ad esempio, si trovava nell'area dati del MAIN che prima partiva dall'indirizzo 0x0000 0000, dopo questa operazione del linker si troverà nella parte di memoria dedicata all'area dati (che parte da 0x1000 000).

3 – Correzione dei riferimenti nei moduli:

Correzione dei riferimenti nei moduli

Siano:

- ISTR un'istruzione riferita dalla tabella di rilocazione di un modulo M, con simbolo S e indirizzo IND
- IADDR l'indirizzo di tale istruzione nell'eseguibile finale:

$$\text{IADDR} = \text{IND} + \text{BASE_M}$$

dove BASE_M è l'indirizzo di base del modulo M

 - VS il valore di S nella tabella globale dei simboli
 - GP il valore del registro *global pointer*

Regole da applicare in base al tipo di istruzione:

- ISTR è in formato J: inserire VS / 4 nell'istruzione
- ISTR è di salto in formato I: inserire (VS – (IADDR + 4)) / 4
- ISTR è aritmetico-logica in formato I:
 - inserire i 16 bit meno significativi di VS (VS_low)
- ISTR è di tipo *load o store*: inserire VS – GP
- ISTR è l'istruzione **Iui**:
 - inserire i 16 bit più significativi di VS (VS_high)

Il file eseguibile di un programma completo, infine, contiene il codice binario eseguibile dalla CPU e informazioni su come caricarlo in memoria e gestirlo. Non è una semplice sequela di istruzioni, indirizzi e valori finiti, contiene:

- L'Intestazione: necessaria per "guidare" l'avvio del programma, specifica le dimensioni del codice eseguibile, i valori iniziali dei dati e l'indirizzo dell'istruzione iniziale del programma.
- Il codice eseguibile: la lista delle istruzioni del programma in forma numerica completa.
- I valori iniziali dei dati statici (variabili globali): lista dei valori iniziali da dare ai dati, al lancio del programma i valori vengono caricati in memoria nel segmento dati **agli indirizzi corrispondenti**.
- La tabella globale dei simboli.

MODULO MAIN	MODULO PROC
<pre>.data VETT: .space 40 BYTE: .space 4 COPY: .space 4 .text (0x 0040 0000) .globl MAIN MAIN: J SUBMOD lw \$t1, VETT sl \$t1, BYTE li \$t0, VETT sw \$t0, COPY addi \$t0, \$zero, \$zero MEND: syscall</pre>	<pre>.data X: .word 5 .text .globl SUBMOD SUBMOD: lw \$t3, COPY lw \$t4, VETT LOOP: sub \$t4, \$t4, VETT bne \$t4, \$zero, LOOP sw \$t4, X SMEND: beq, \$zero, \$zero, MAIN</pre>

Calcolo le dimensioni del testo e dei dati dei moduli:

MAIN:

- Testo: 32 Byte -> 7 istruzioni da 32 Byte, ma **li** è una pseudo-istruzione, ne richiede 2 -> 8 x 4 Byte **0x20**
- Dati: 48 Byte -> **0x 30**

MAIN - oggetto assemblato;

Calcolo indirizzi e spiazzamenti dell'area di testo

```
0x 0040 0000  J, 0x 0000 0000 -> non ha ancora l'indirizzo di submod e quindi mette 0, poi lo corregge
0x 0040 0004  lw $t1, 0000($gp) -> spiazzamento di VETT non ancora noto
0x 0040 0008  sl $t1, 0000($gp) -> spiazzamento di BYTE non ancora noto
0x 0040 000C  lui $t0, 0000 -> spiazzamento non ancora noto
0x 0040 0010  ori $t0, 0000 -> spiazzamento non ancora noto
0x 0040 0014  sw $t0, 0000 -> spiazzamento non ancora noto
0x 0040 0018  addi $t0, $zero, 0000 -> spiazzamento non ancora noto
0x 0040 001C  syscall
```

Calcolo indirizzi e spiazzamenti dell'area di dati

```
0x 0000 0000  non inizializzato VETT
0x 0000 0018  non inizializzato BYTE
0x 0000 002C  non inizializzato COPY
```

Tabella dei Simboli di Main			Tabella di Rilocazione di Main		
MAIN	TESTO	0x 0000 0000	0x 0000 0000	J	SUBMOD
MEND	TESTO	0x 0000 001C	0x 0000 0000	SW	VETT
VETT	DATI	0x 0000 0000	0x 0000 0000	SL	BYTE
BYTE	DATI	0x 0000 0018	0x 0000 0000	LUI	VETT
COPY	DATI	0x 0000 002C	0x 0000 0000	ORI	VETT
			0x 0000 0000	SW	COPY

PROC (SUBMOD):

- Testo: 24 Byte -> **0x18**
- Dati: 4 Byte -> **0x04**

PROC - oggetto assemblato;

Calcolo indirizzi e spiazzamenti dell'area di testo

```

0x 0000 0000  lw $t3, 0000($gp) -> spiazzamento di COPY non ancora noto
0x 0000 0004  lw $t4, 0000($gp) -> spiazzamento di VETT non ancora noto
0x 0000 0008  sub $t4, $t4, $t3
0x 0000 000C  bne $zero, -2 -> -2 decimale, salta indietro di 2 istruzioni (noto perché interno al modulo)
0x 0000 0010  sw $t4, 0000($gp) -> spiazzamento non ancora noto
0x 0000 0014  beq $t0, 0x 0000 0000

```

Calcolo indirizzi e spiazzamenti dell'area di dati

0x 0000 0000 0x 0000 0005

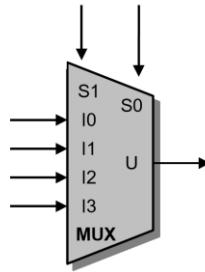
Tabella dei Simboli di Proc			Tabella di Rilocazione di Proc		
SUBMOD	TESTO	0x 0000 0000	0x 0000 0000	LW	COPY
LOOP	TESTO	0x 0000 0008	0x 0000 0000	LW	VETT
SMEND	TESTO	0x 0000 0014	0x 0000 0000	SW	X
X	DATI	0x 0000 0000	0x 0000 0000	BEQ	MAIN

Tabella Globale dei Simboli			
Etichetta	Offset	Indirizzo Base	Indirizzo Globale
MAIN	0x 0000 0000	0x 0040 0000	0x 0040 0000
MEND	0x 0000 001c	0x 0040 0000	0x 0040 001C
VETT	0x 0000 0000	0x 1000 0000	0x 1000 0000
BYTE	0x 0000 0028	0x 1000 0000	0x 1000 0028
COPY	0x 0000 002C	0x 1000 0000	0x 1000 002C
SUBMOD	0x 0000 0000	0x 0040 0020	0x 0040 0020
LOOP	0x 0000 0008	0x 0040 0020	0x 0040 0028
SMEND	0x 0000 0014	0x 0040 0020	0x 0040 0034
X	0x 0000 0000	0x 1000 0030	0x 1000 003C

LIVELLO LOGICO DIGITALE**Blocchi funzionali combinatori:****Multiplexer (MUX):**

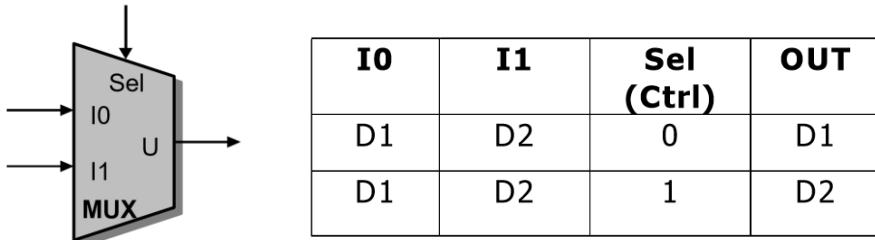
Ha:

- ingressi dati (numerati): 2^n
- ingresso/i selezione
- uscita



Viene "inviaato" all'uscita l'ingresso corrispondente al numero indicato dai bit degli ingressi di selezione.

Esempio a 2 bits:



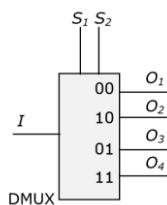
Se invece si volessero avere più di 2 ingressi dati è necessario implementare più ingressi selezione.

E' facile capire che per poter gestire 2^n ingressi dati è necessario avere n ingressi selezione.

Demultiplexer (DMUX):

Ha:

- Un ingresso dati
- n ingressi selezione
- 2^n uscite

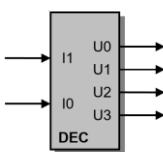


Il bit dell'ingresso dati viene inviato all'uscita corrispondente al numero binario indicato dai bit di selezione, le uscite che non sono considerate in questo processo sono messe a 0.

Decodificatore (Decoder):

Ha:

- n ingressi
- 2^n uscite



Il decodificatore, brevemente, è un DMUX senza gli ingressi di selezione.

Se negli ingressi è presente il numero k espresso in binario, la k^{esima} uscita viene messa a 1, tutte le altre a 0.

Il suo utilizzo all'interno del calcolatore è interno alla memoria, il Decodificatore si occupa di decodificare gli indirizzi delle parole di memoria.

Comparatore (Comparator):

Ha:

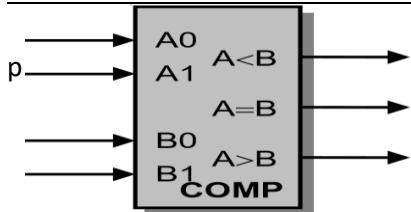
- 2 ingressi dati (a n bit)
- 3 uscite: $A > B$, $A = B$, $A < B$

Il comparatore confronta i due ingressi dati e mette a 1 l'uscita corrispondente alla relazione tra i due numeri.

L'esempio con ingressi a 1 bit è banale, a 2 bit è già più complicato, la tabella delle verità ha 2^{2n} righe.

HALF-ADDER			
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

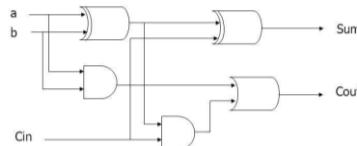
Sum = $A \oplus B$
Carry = AB

Blocchi aritmetici fondamentali: VEDI TRASPARENZE (blocchi funzionali combinatori).**Half-adder:**

L'half-adder è il primo blocco aritmetico fondamentale e il più semplice. Ha due ingressi, sull'uscita viene riportata la somma e per la prima volta si nota il bit di carry (riporto) implementato a livello fisico

Full-adder:

FULL ADDER					
A	B	Carry In	Sum	Carry Out	
0	0	0	0	0	
0	0	1	1	0	
0	1	0	1	0	
0	1	1	0	1	
1	0	0	1	0	
1	0	1	0	1	
1	1	0	0	1	
1	1	1	1	1	



$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + C(A \oplus B)$$

Il full-adder (sommatore completo), è una implementazione dell'half-adder con una modifica.

Il full-adder permette di eseguire operazioni avendo un riporto in entrata, il suo utilizzo si può facilmente immaginare come il secondo passo di una somma binaria con 2 bit da sommare in ingresso e il carry a riporto.

Nota sul full adder: per le operazioni in 32 bit si usano 32 full adder divisi in gruppi di 16. Se riesco a capire se c'è un trabocco (overflow) nei primi 16 bit posso effettivamente spezzare la somma in 2 e farla molto veloce, altrimenti dovrei aspettare il completamento della prima somma per utilizzare l'eventuale bit di overflow.

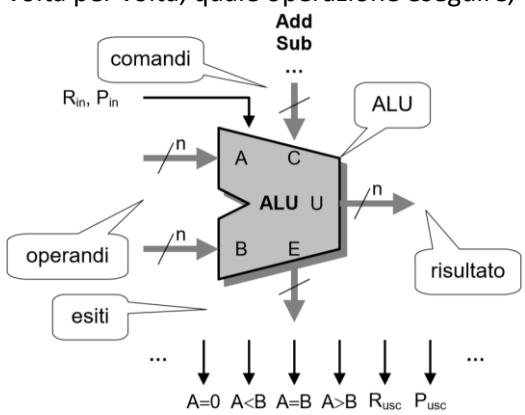
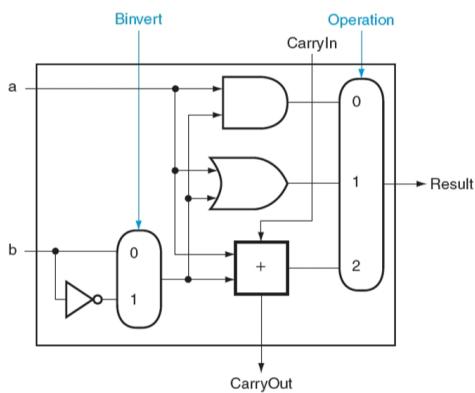
ALU (Arithmetic Logic Unit):

Unico blocco contenente una combinazione dei blocchi funzionali precedenti. Consente di eseguire tutte le operazioni di cui la CPU necessita. Ovviamente per distinguere, volta per volta, quale operazione eseguire, sono necessarie dei **comandi**.

Nel caso del **MIPS** abbiamo bisogno di poche operazioni:

- Add
- Sub
- And
- Or
- Not

L'ALU del MIPS è schematizzata di seguito:



Ed è in grado di eseguire tutte le operazioni necessarie al MIPS.

Overflow detection.

Libreria di blocchi sequenziali:

I blocchi sequenziali presenti nell'architettura MIPS sono:

- Registro parallelo
- Registro a scorrimento
- Banco dei registri
- Memoria

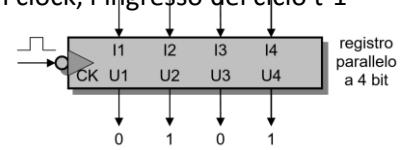
Ognuno di questi, poi, ammette diverse versioni e varianti.

Registro parallelo:

Vettore di $n \geq 1$ flip-flop di tipo D, con n ingressi (In) ed n uscite(Un) e un ingresso di clock CK (ovviamente, essendo un flip-flop *edge-triggered*).

La sua funzione è quella di salvare una parola di memoria per un ciclo di clock, l'⁰ingresso del ciclo t-1^{esimo} viene riportato sull'uscita del ciclo t^{esimo}.

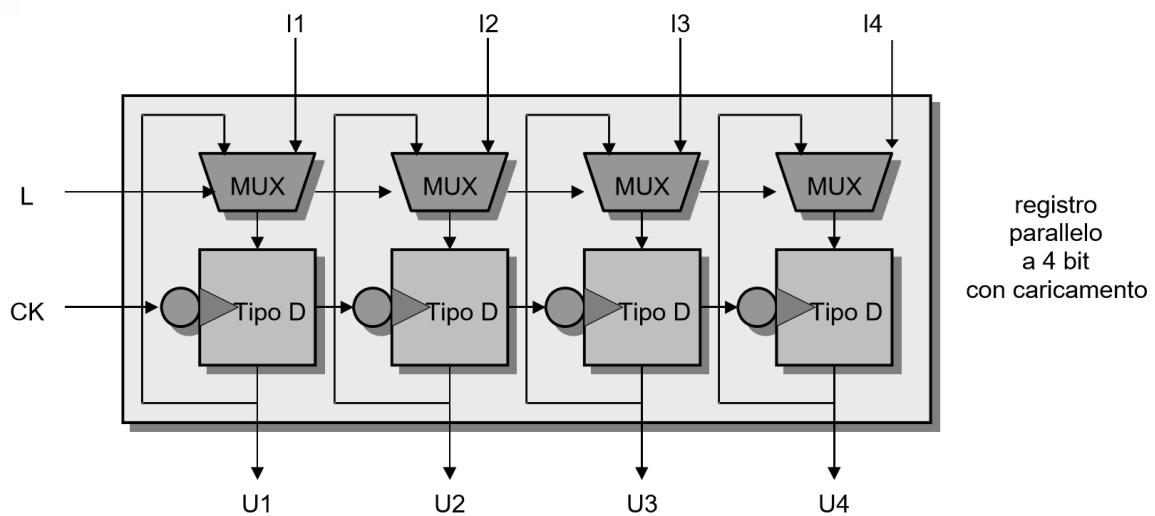
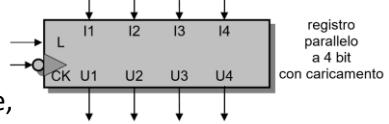
A ogni ciclo avviene una scrittura, anche se la parola in ingresso non viene cambiata.



Per questo si aggiunge un tipo di registro parallelo con **ingresso di caricamento**, che è un registro parallelo a cui si aggiunge un ingresso L detto di **load**, che quando viene attivato impone al registro di scrivere la nuova parola in uscita.

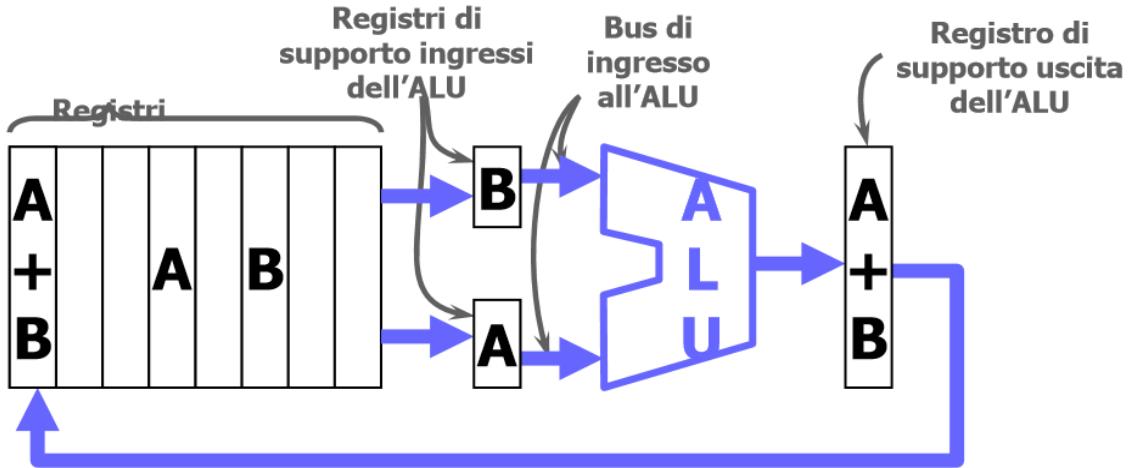
Viene realizzato con 4 (o n) flip-flop di tipo D sincroni sul fronte di discesa.

Allo schema del registro parallelo "semplice" vengono aggiunti 4 MUX a 2 bit, uno per ogni flip-flop, che in ingresso hanno lo stato del bit precedente, lo stato del nuovo bit da caricare (se presente) e come bit di selezione (SEL) il comando L.



Banco di registri (Register file):

Insieme di registri paralleli con comando di caricamento di dimensioni uguali (32 bit in MIPS).



IL PROCESSORE:

MIPS ha architettura RISC (Reduced Instruction Set Computer), ossia esegue solo istruzioni semplici con massimo 3 operandi.

E' strutturata con un'architettura LOAD/STORE: gli operandi dell'ALU possono solo provenire dai registri, non direttamente dalla memoria. Per questo sono necessarie operazioni di **caricamento** (*load*) per copiare un dato della memoria in un registro e operazioni di **memorizzazione** (*store*) per memorizzare i risultati delle operazioni.

Implementa anche l'architettura **pipeline**: una tecnica per migliorare le prestazioni basata sulla sovrapposizione dell'esecuzione di più istruzioni appartenenti ad un flusso di esecuzione sequenziale.

L'implementazione MIPS che analizzeremo con il seguente ISA:

- Istruzioni aritmetico-logiche
- Istruzioni di trasferimento da/verso la memoria (*load/store*)
- Istruzioni di salto (condizionato e incondizionato) (*beq/bne/j..*)

Le istruzioni hanno 3 formati: **R**, **I** e **J**.

Le istruzioni **aritmetico logiche** possono essere di tipo R e di tipo I.

Le istruzioni di **trasferimento da/verso la memoria** possono essere di tipo R e I.

Le istruzioni di **salto condizionato e incondizionato** sono solamente di tipo J.

Esecuzione delle istruzioni:**Passi svolti durante l'esecuzione delle istruzioni:****Aritmetico logiche di tipo R (4 passi):**

- 1) Prelievo istruzione dalla memoria istruzione e incremento del PC
- 2) Lettura dei 2 registri sorgente dal banco dei registri, utilizzando i bit [25-21] e [20-16] per selezionare i registri
- 3) Operazione dell'ALU sui dati letti dal banco dei registri, utilizzando il campo **function** per realizzare la funzione aritmetico logica (differenza add, addi, addiu)
- 4) Scrittura del risultato dell'ALU nel banco dei registri utilizzando i bit [15-21] dell'istruzione per selezionare il registro di destinazione.

Non serve precisamente sapere i bit di selezione dei registri, vedi il formato istruzione.

Load (5 passi):

- 1) Prelievo dell'istruzione dalla memoria istruzioni e incremento del PC
- 2) Lettura del registro base dal banco dei registri, bit [25-21]
- 3) Operazione dell'ALU per calcolare la somma del valore letto dal registro base e dei 16 bit meno significativi dell'istruzione estesi in segno (campo offset)
- 4) Prelievo del dato nella memoria dati utilizzando come indirizzo di lettura il risultato dell'ALU
- 5) Scrittura del dato proveniente dalla memoria nel banco dei registri; il registro destinazione (\$x) è indicato dai bit [20-16] dell'istruzione.

Store (4 passi):

- 1) Prelievo istruzione dalla memoria istruzioni e incremento del PC
- 2) Lettura del registro base ($\$y$), bit [25-21], e del registro sorgente ($\$x$) dal banco dei registri; il registro sorgente è indicato dai bit [20-16] dell'istruzione
- 3) Operazione dell'ALU per calcolare la somma del valore letto dal registro base e dei 16 bit meno significativi dell'istruzione estesi in segno (offset)
- 4) Scrittura del dato proveniente dal registro sorgente ($\$x$) nella memoria dati utilizzando come indirizzo di scrittura il risultato dell'ALU

Addi (4 passi):

- 1) Prelievo istruzione dalla memoria istruzioni e incremento del PC
- 2) Lettura del registro sorgente ($\$y$) dal banco dei registri, bit [25-21]
- 3) Operazione dell'ALU per calcolare la somma del valore letto dal registro sorgente e dei 16 bit meno significativi dell'istruzione estesi in segno (campo immediato)
- 4) Scrittura del risultato dell'ALU nel banco dei registri; il registro destinazione ($\$x$) è indicato dai bit [20-16] dell'istruzione.

Salto condizionato (4 passi):

- 1) Prelievo istruzione dalla memoria istruzioni e incremento del PC
- 2) Lettura dei 2 registri sorgente dal banco dei registri
- 3) Operazione dell'ALU per effettuare la sottrazione tra i valori letti dal banco dei registri. Il valore $(PC+4)$ viene sommato ai 16 bit meno significativi dell'istruzione estesi in segno (offset); il risultato è l'indirizzo di destinazione del salto (Branch Target Address)
- 4) L'uscita Zero dell'ALU viene utilizzata per decidere quale valore debba essere memorizzato nel PC: $(PC+4)$ oppure $(PC+4+offset)$

Istruzioni aritmetico-logiche: op \$x, \$y, \$z

Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente $\$y$ e $\$z$	Op. ALU sui Dati Letti ($\$y$ op $\$z$)	Scrittura nel Reg. Destinazione $\$x$
----------------------------------	--	--	--

Istruzioni di caricamento (*load*): lw \$x, offset(\$y)

Prelievo Istruz. & Increm. PC	Lettura Registro Base $\$y$	Op. ALU ($\$y+offset$)	Prelievo Dato $M(\$y+offset)$	Scrittura nel Reg. Destinazione $\$x$
----------------------------------	--------------------------------	-----------------------------	----------------------------------	--

Istruzioni di memorizzazione (*store*): sw \$x, offset(\$y)

Prelievo Istruz. & Increm. PC	Lettura Registri Base $\$y$ & Sorg. $\$x$	Op. ALU ($\$y+offset$)	Scrittura Dato $M(\$y+offset)$
----------------------------------	--	-----------------------------	-----------------------------------

Istruzioni di salto condizionato: beq \$x, \$y, offset

Prelievo Istruz. & Increm. PC	Lettura Registri Sorgente $\$x$ e $\$y$	Op. ALU ($\$x-\y) & $(PC+4+offset)$	Scrittura nel PC
----------------------------------	--	--	---------------------

Si può notare che per tutte le istruzioni sopra descritte i primi due passi svolti sono sempre il prelievo dell'istruzione e la lettura dei registri sorgente. Per le operazioni che producono un risultato c'è sempre la scrittura del risultato nel registro destinazione.

Dopo i primi due passi, le azioni necessarie per lo svolgimento e la conclusione dell'operazione dipendono dal tipo di istruzione (codice operativo).

Si differenziano ulteriormente nei passi seguenti all'utilizzo dell'ALU:

- Le istruzioni aritmetico-logiche devono scrivere nel registro destinazione il risultato dell'ALU
- Le istruzioni di *load* richiedono l'accesso in lettura alla Memoria Dati ed eseguono il caricamento del dato letto nel registro destinazione
- Le istruzioni di *store* richiedono l'accesso in scrittura alla Memoria Dati ed eseguono la memorizzazione del dato proveniente dal registro sorgente
- Le istruzioni di salto condizionato, possono cambiare l'indirizzo dell'istruzione successiva in base al risultato del confronto.

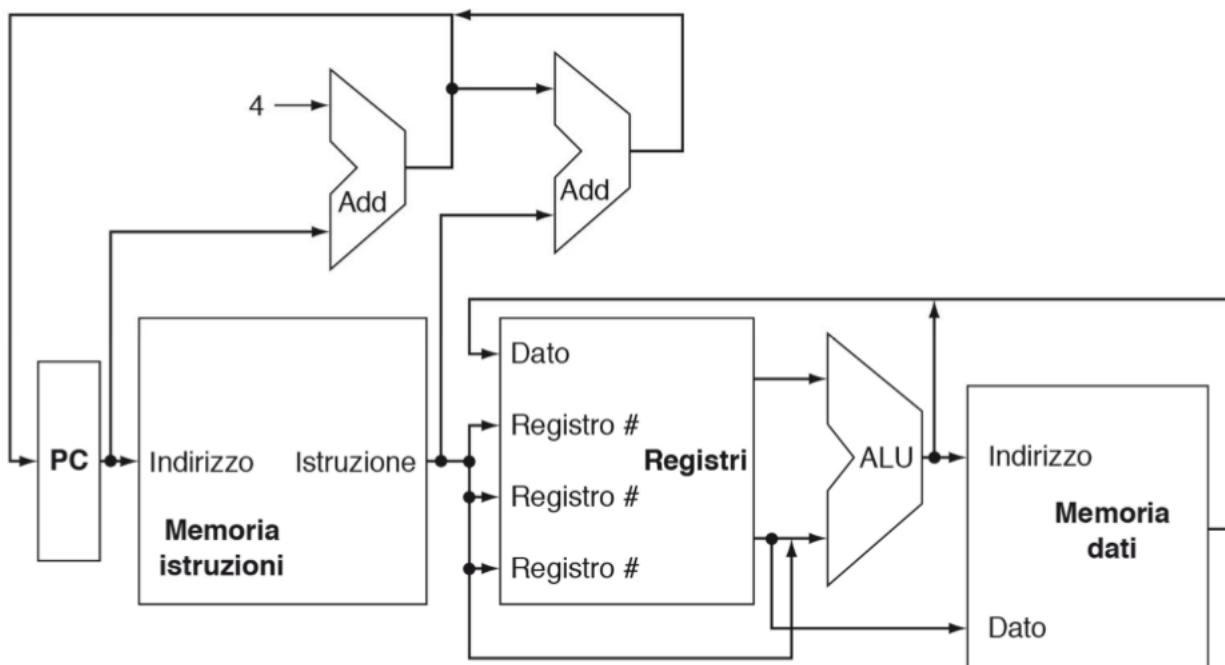
Tempo di esecuzione delle istruzioni:

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

Attenzione: se si prevede di eseguire ciascuna di queste istruzioni in un solo ciclo di CLOCK è necessario che questo non sia minore di 800ps (tempo di esecuzione dell'istruzione più "lenta").

CALCOLA LA FREQUENZA CORRISPONDENTE A CIASCUNA RIGA E AL CICLO DI CLOCK.

Struttura base del processore MIPS:



Nota: La memoria istruzioni è usata solo in lettura.

Register File:

I 32 registri sono organizzati in un **Register File (RF)** con **2 porte di lettura e 1 porta di scrittura**, i campi dell'istruzione indicano direttamente i registri da utilizzare come operandi dell'istruzione e vengono perciò collegati alle porte di lettura e scrittura del Register File (ecco perché 2 e 1, se molte istruzioni hanno 3 registri come operandi, 2 sorgente e 1 destinazione).

Nel Register file, per realizzare le due porte di lettura e la porta di scrittura, sono presenti **4 ingressi e 2 uscite** così suddivise:

- 3 ingressi sono collegati ai campi dell'istruzione che specificano i registri sorgente o base e il registro destinazione (2 per porta in lettura e 1 per porta in scrittura)
- 1 ingresso è per i dati che possono essere scritti nel registro destinazione (per la porta di scrittura)
- 2 uscite del banco dei registri riportano il contenuto dei 2 registri letti (per le porte di lettura)

Gli operandi dell'ALU sono utilizzati per:

- Calcolare un risultato aritmetico (istruzione aritmetico-logica)
- Calcolare un indirizzo di memoria tipo offset(registro) load e store, salto incondizionato(?)
- Effettuare un confronto (istruzioni di salto condizionato)

Il risultato prodotto dall'ALU è utilizzato per:

- Scrittura in un registro di destinazione (istruzione aritmetico-logica)
- Come indirizzo di lettura/scrittura della Memoria Dati (load/store)
- Calcolare l'indirizzo della prossima istruzione, secondo un'apposita logica di controllo (salto condizionato/incondizionato(?)

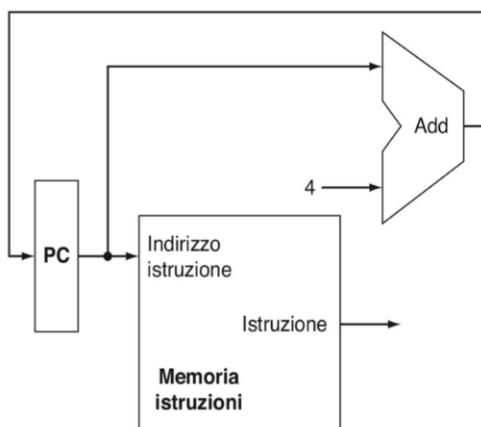
Per il calcolo dell'indirizzo di salto viene preso il PC + 4 (istruzione successiva a quella corrente), viene elaborato l'offset espresso nell'istruzione o l'etichetta in questione e viene sommato al dato precedente. Ovviamente se c'è un salto condizionato ci sarà un controllo che stabilisce se bisogna saltare o no.

LE DIVERSE ALU PRESENTI NEGLI SCHEMI SONO EFFETTIVAMENTE DIVERSE ALU.

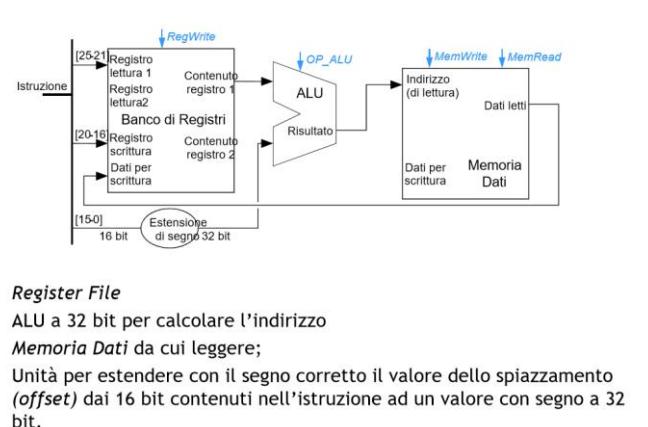
In tutto sono 3, ma perché? Perché sono utilizzate contemporaneamente.

- Una è quella sempre usata per incrementare il PC di 4 (molto semplificata). È Solamente un sommatore costante di 4 all'ingresso.
- Una per il calcolo degli indirizzi.
- Una per i confronti.

Fase di FETCH (caricamento) delle istruzioni



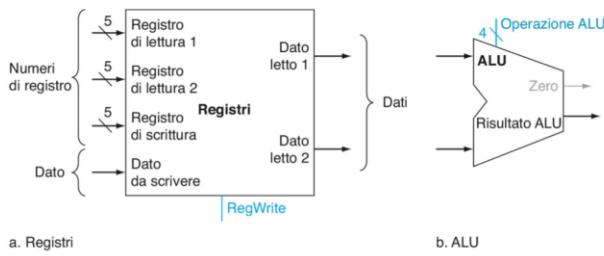
Esecuzione delle istruzioni di load



Il sommatore presente nello schema con ingresso fisso 4 indica un'operazione che viene SEMPRE eseguita, ovvero l'incremento del PC di 4 per passare all'istruzione successiva.

Esecuzione dei diversi tipi di istruzione

Esecuzione delle istruzioni aritmetico-logiche



- > **Register File**
- > ALU a 32 bit che riceve 2 ingressi da 32 bit e che restituisce un risultato da 32 bit.

Esecuzione delle istruzioni aritmetico-logiche (cont.)

Le istruzioni aritmetico-logiche hanno come operandi **3 registri**: per ogni istruzione si devono leggere due parole di dati dal banco di registri e se ne deve scrivere una

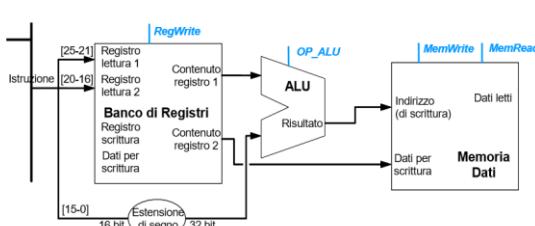
Per ogni parola letta dai registri sorgente sono necessari un ingresso (5 bit) al banco di registri, per specificare il numero del registro che si vuole leggere, ed un'uscita dal banco (32 bit) per il valore letto dal registro

Per scrivere una parola nel registro destinazione sono necessari due ingressi: un ingresso (5 bit) per specificare il numero del registro in cui si vuole scrivere ed un ingresso (32 bit) per il dato da scrivere

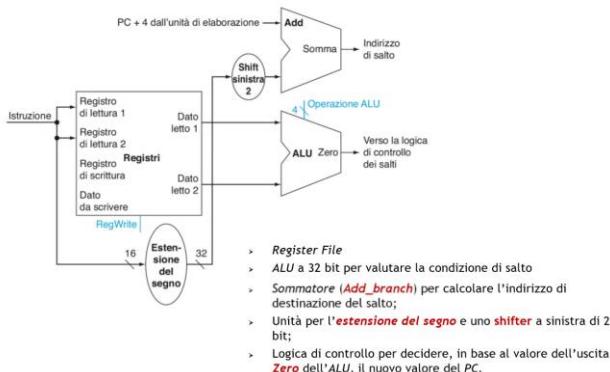
Il banco di registri fornisce sempre in uscita il contenuto dei registri di lettura, mentre le scritture sono controllate da un apposito segnale di controllo della scrittura (RegWrite)

Il segnale di controllo OP_ALU provvede a specificare all'ALU il tipo di operazione.

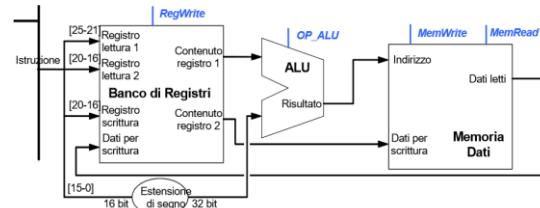
Esecuzione delle istruzioni di store



Esecuzione delle istruzioni di salto condizionato



Esecuzione delle istruzioni di load/store



Il valore dello spiazzamento (**offset**), dopo l'estensione di segno, è utilizzato

Esecuzione delle istruzioni di salto condizionato (cont.)

La base per il calcolo dell'indirizzo di destinazione di un salto condizionato è l'indirizzo dell'istruzione che segue quella di salto (PC + 4).

Poiché ogni istruzione occupa 4 byte, prima di effettuare la somma dello spiazzamento (**offset**) con il contenuto di (PC+4), bisogna **moltiplicare per 4** il valore contenuto nell'istruzione
 => **scorrimento a sinistra di 2 bit**.

L'operazione di salto condizionato è costituita da due operazioni:

- Calcolo dell'indirizzo di destinazione del salto attraverso un sommatore che effettua la somma tra il (PC+4) e il valore contenuto nell'istruzione dopo avere esteso il segno e fatto scorrere a sinistra di 2 bit;
- Confronto nell'ALU del contenuto dei registri operandi letti dal RF.
 Se il segnale di uscita Zero dell'ALU è asserito => la condizione di salto è verificata e l'indirizzo di destinazione del salto diventa il nuovo PC.
 Se invece la condizione non è verificata => il PC incrementato sostituisce il PC attuale.

Realizzazione del processore completo:

Si vuole combinare tutti gli elementi richiesti da ogni tipo di operazione in un'unica unità di elaborazione.
 Si assume che:

- Tutte le istruzioni siano eseguite in un solo ciclo di clock
- Nessuna risorsa può essere utilizzata più di una volta per istruzione
- Qualsiasi risorsa di cui si ha bisogno più di una volta deve essere duplicata

La Memoria Istruzioni deve essere quindi **distinta** dalla Memoria Dati, perché se l'istruzione si esegue in un unico ciclo di clock e la Memoria Dati può essere utilizzata per le operazioni di load/store (e solo una volta) non potrei usare la Memoria Istruzioni per prelevare il codice dell'istruzione.

Alcune unità funzionali potrebbero essere duplicate nel momento in cui si combinano le varie unità di calcolo definite nella precedente sezione, mentre altre unità possono essere condivise da differenti flussi di

istruzioni.

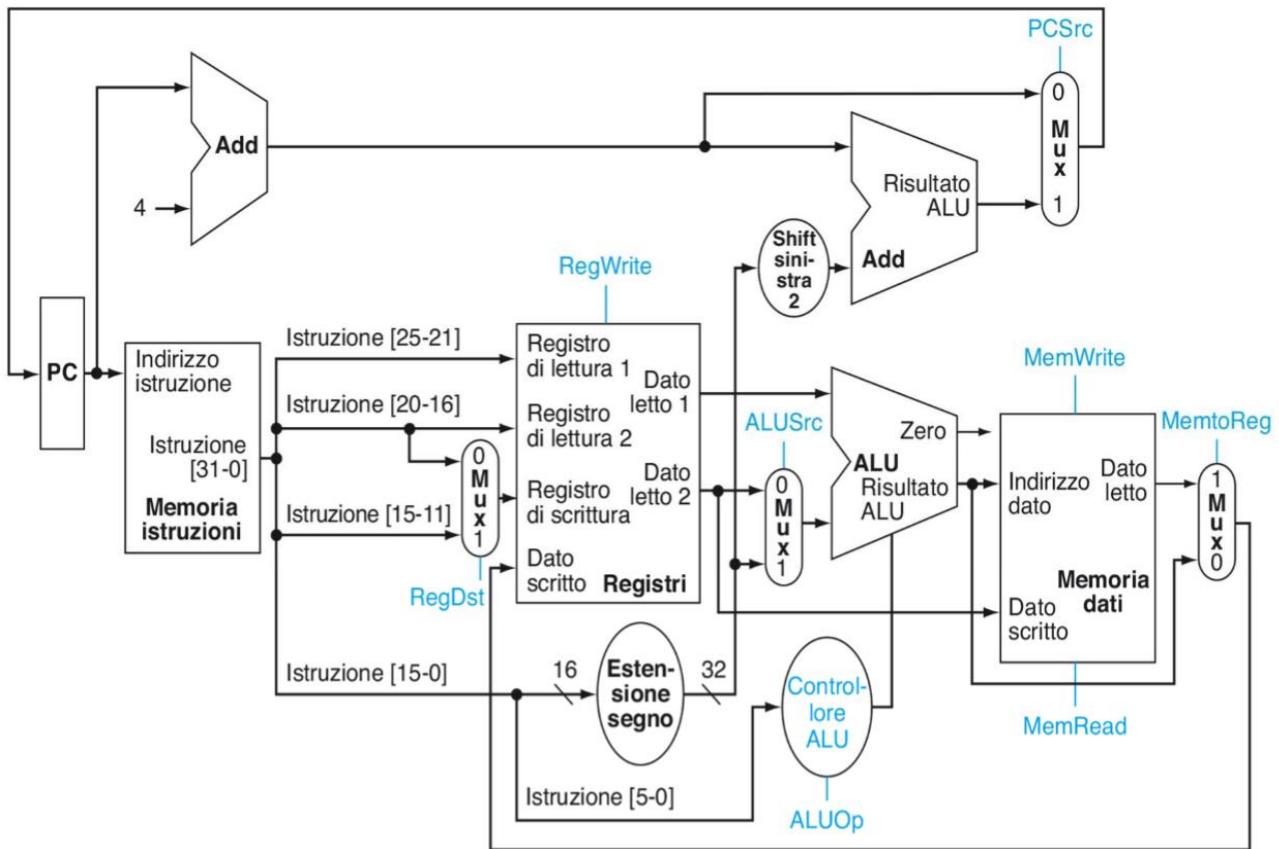
Per condividere un elemento tra due diversi tipi di istruzione, si deve introdurre un **multiplexer** di dati per permettere connessioni multiple all'ingresso di un elemento e selezionare uno tra i vari ingressi in base alla configurazione delle linee di controllo.

Esecuzioni delle istruzioni aritmetico-logiche e load/store – utilizzo del multiplexer per la risoluzione della condivisione:

- 1- Il secondo ingresso dell'ALU è il contenuto di un registro (istruzione di tipo R) oppure la metà meno significativa dell'istruzione (istruzione di load/store o aritmetiche immediate).
Si utilizza quindi un MUX al secondo ingresso dell'ALU (Mux A).
- 2- Il valore scritto nel registro destinazione proviene dal risultato dell'ALU (istruzione tipo R o aritmetica immediata) oppure dalla Memoria Dati (istruzione di load).
Si utilizza quindi MUX all'ingresso dei Dati per Scrittura del RF (Mux C).
- 3- Il numero del registro in cui si vuole scrivere il risultato è indicato da diversi campi (i bit [15-11] per le istruzioni di tipo R e bit [20-16] per istruzioni di load/store aritmetiche immediate).
Si utilizza quindi MUX all'ingresso del Registro Scrittura del RF (Mux D).

! Nel caso delle istruzioni di salto condizionato, è presente un **MUX** che permette di decidere se eseguire il salto o meno.

STRUTTURA DELLA CPU



In azzurro sono scritti i segnali di controllo, la loro funzione è espressa nella seguente tabella.

Non tutte le combinazioni di segnali sono però dotate di senso, quindi deve sempre esservi un senso logico per l'esecuzione di un'operazione.

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

VEDI NOTA SULLA TEMPORIZZAZIONE fino a fine pdf.

Abbiamo infine una micro architettura del processore MIPS di tipo monociclo, in un ciclo esegue esattamente un'istruzione macchina.

La suddivisione dei cicli di clock è effettuata sul fronte di salita del segnale.

PIPELINING:

Il **pipelining** è una tecnica per **migliorare le prestazioni** del processore basata sulla sovrapposizione dell'esecuzione di più istruzioni appartenenti ad un flusso di esecuzione sequenziale. L'obiettivo finale è quello di aumentare la *frequenza* di esecuzione delle istruzioni.

È importante notare che con questa tecnica NON si riduce il tempo di esecuzione delle istruzioni, la singola istruzione mantiene sempre il suo tempo di esecuzione che non viene alterato. Quello che cambia è che vengono eseguite più istruzioni "in parallelo" e quindi aumenta la frequenza di esecuzione delle istruzioni. In realtà il pipelining comporta un minimo aumento del tempo di esecuzione delle singole istruzioni.

È una tecnica *trasparente* al programmatore, implementata direttamente dalla CPU, comparabile ad una catena di montaggio. Il lavoro della CPU (*pipeline*) per eseguire un'istruzione è diviso in passi: **stadi di pipeline**, che richiedono una frazione del tempo necessario al tempo totale richiesto per svolgere un'intera istruzione. Gli stadi sono sovrapposti per formare la pipeline: le istruzioni entrano da una estremità, vengono elaborate attraverso gli stadi, escono dall'altro estremo.

Il tempo necessario per far avanzare un'istruzione di uno stadio lungo la pipeline corrisponde idealmente ad un **ciclo di clock**.

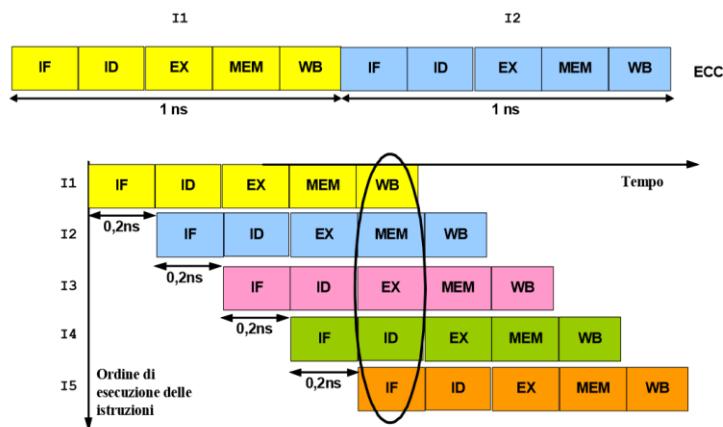
Poiché gli stadi di pipeline sono collegati in successione, devono tutti operare in modo sincrono: la durata di un ciclo di pipeline è determinata dal tempo richiesto per lo stadio più lento della pipeline.

Il **vincolo fondamentale** alla base di questa tecnica è che nello stesso ciclo di clock non si può eseguire più di uno stadio che utilizza determinate risorse, ossia non posso avere nello stesso ciclo di clock due stadi che utilizzano la memoria dati o tre stadi che richiedono operazioni dell'ALU.

Idealmente l'architettura del processore MIPS suddivide le istruzioni in 5 passi, quindi è 5 volte più veloce dell'architettura monociclo, perché esegue 5 stadi di 5 diverse istruzioni in un solo ciclo di clock, in realtà il parallelismo non è bilanciato perfettamente e quindi non si riuscirà esattamente a quintuplicare le prestazioni. Inoltre, l'intervallo di tempo che intercorre tra il completamento di 2 istruzioni è superiore al valore minimo possibile.

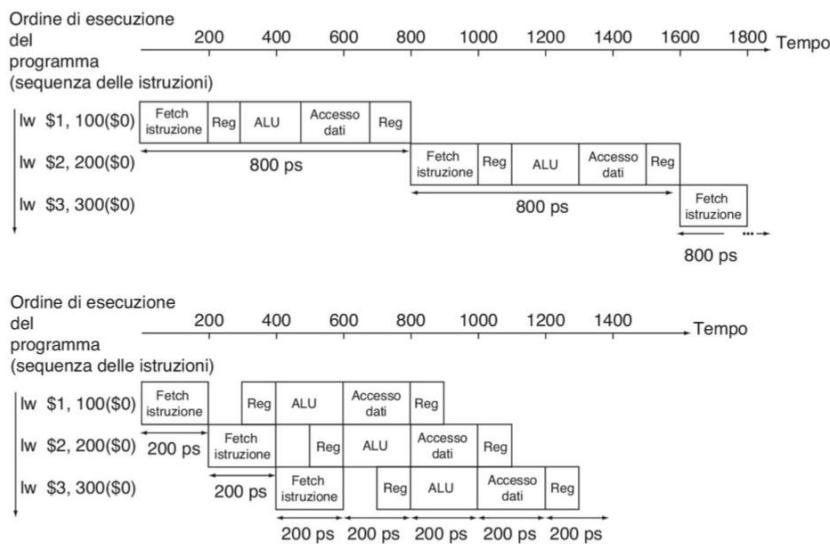
Di seguito è riportato un diagramma temporale che permette di comprendere meglio l'implementazione di questa tecnica:

Esecuzione sequenziale vs. pipelining



Ogni stadio di pipeline ha una durata prefissata (ciclo di pipeline), che deve essere sufficientemente lunga da consentire l'esecuzione dell'operazione più lenta: ad esempio si deve adottare un ciclo di almeno 200 ps, sebbene alcuni stadi richiedano solo 100 ps. In genere ogni ciclo di pipeline ha una durata diversa rispetto agli altri. MA, mentre nell'architettura monociclo un ciclo di clock aveva durata 800 ps perché questo era il risultato della somma dei tempi di esecuzione di ogni fase di una istruzione, nel **pipelining** ogni ciclo di clock può avere durata diversa e in genere molto minore degli 800 ps precedenti.

Confronto tra una esecuzione a singolo ciclo di clock e pipeline



Il miglioramento non è sempre lineare e uguale a 4 per le 5 pipeline di MIPS. In realtà basta fare un esempio per capirlo. Il tempo di esecuzione totale di 3 istruzioni di load comporta un miglioramento più modesto:

- 3 istruzioni di load senza pipeline: $3 \times 800 \text{ ps} = 2400 \text{ ps}$
- 3 istruzioni di load con pipeline: $2 \times 200 + 1000 \text{ ps} = 1400 \text{ ps}$
- Fattore di miglioramento: $2400/1400 = 1,71$

Questa differenza è provocata dal tempo necessario a riempire e svuotare la pipeline: servono 4 stadi per riempire la pipeline.

Al crescere del numero di istruzioni, però, il fattore di miglioramento torna a incrementarsi, raggiungendo asintoticamente 4.

Esempio asintotico con un milione di istruzioni:

- 1 000 000 istruzioni di load senza pipeline: $1 000 000 \times 0,8 \text{ ns} = 800 000 \text{ ns}$
- 1 000 000 istruzioni di load con pipeline: $1 000 000 \times 0,2 \text{ ns} + 0,8 \text{ ns} = 200 000,8 \text{ ns}$
- $800 000 \text{ ns} / 200 000,8 \text{ ns} = 3,999984$

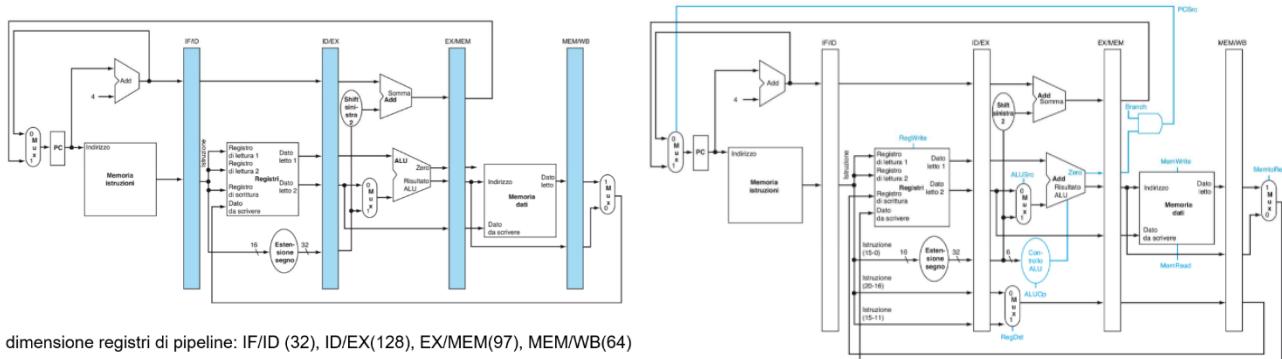
Per 1000 istruzioni $800 \text{ ns} / 200,8 \text{ ns} = 3,984$

Siccome questi casi sono molto più verosimili rispetto all'esempio con 3 istruzioni. Prendiamo il pipelining come una buona tecnica di miglioramento delle prestazioni.

Considerazioni sulla struttura delle istruzioni MIPS:

- Le istruzioni MIPS hanno tutte la stessa lunghezza: semplificazione per le fasi di prelievo e decodifica delle istruzioni
- Le istruzioni MIPS hanno un numero molto ridotto di formati diversi e, non considerando il registro destinazione delle istruzioni aritmetico-logiche, gli altri due registri sono sempre specificati nella stessa posizione (rs e rt) dell'istruzione codificata
- Gli operandi residenti in memoria sono presenti solo nelle istruzioni di load e store
- L'allineamento degli operandi in memoria è obbligatorio: il trasferimento dati con la memoria avviene sempre con un solo accesso

Struttura pipeline della CPU MIPS (cont.) Segnali dell'unità di controllo della pipeline



Nella trasparenza di *sinistra* vengono introdotti (in azzurro) i registri per distinguere i diversi stadi e per trasferire i dati da uno stadio all’altro. Prendono il nome di **registri di stadio** e sono di tipo *edge-triggered*, ovvero eseguono la scrittura del dato in entrata sull’uscita in corrispondenza del fronte di salita del clock.

Alternativamente: Implementazione dei **registri di pipeline** in barre azzurre, suddividono gli stadi memorizzando le informazioni sul fronte di salita e riportandole all’uscita per lo stadio seguente, mentre nella porzione precedente viene processato un nuovo stadio di una nuova istruzione.

Nella trasparenza di *destra* vengono introdotti i segnali di controllo della pipeline. Questi segnali sono indispensabili per il funzionamento corretto dell’architettura pipelined poiché:

- Non è sufficiente implementare solamente i registri lasciando inalterata la struttura della CPU per avere una struttura pipeline correttamente funzionante
- Avendo più istruzioni in esecuzione sulla CPU, tutti i dati appartenenti ad un’istruzione devono essere mantenuti all’interno dello stadio o passati da uno stadio all’altro in stato non alterato rispetto a quello di entrata e con temporizzazione corretta per rientrare nello stadio di esecuzione in cui sono richiesti.

I valori dei segnali vengono tutti generati a partire dal contenuto del registro IF/ID che –dopo ogni fetch –contiene l’istruzione da eseguire e il valore del PC incrementato.

I segnali di controllo presenti nell’architettura pipelined di MIPS sono i seguenti:

Codice operativo istruzione	ALUOp	Operazione associata all’istruzione	Codice funzione	Operazione della ALU	Input di controllo della ALU
LW	00	load word	XXXXXX	somma	0010
SW	00	store word	XXXXXX	somma	0010
Branch equal	01	branch equal	XXXXXX	sottrazione	0110
Tipo R	10	somma	100000	somma	0010
Tipo R	10	sottrazione	100010	sottrazione	0110
Tipo R	10	AND	100100	AND	0000
Tipo R	10	OR	100101	OR	0001
Tipo R	10	set less than	101010	set less than	0111

E i loro effetti sono i seguenti (come CPU monociclo):

Nome del segnale	Effetto quando non asserito (0)	Effetto quando asserito (1)
RegDst	Il numero del registro di scrittura proviene dal campo rt (bit 20-16)	Il numero del registro di scrittura proviene dal campo rd (bit 15-11)
RegWrite	Nulla	Il dato viene scritto nel registro (del register file) individuato dal numero del registro di scrittura
ALUSrc	Il secondo operando della ALU proviene dalla seconda uscita del register file (Dato letto 2)	Il secondo operando della ALU proviene dall’estensione del segno dei 16 bit meno significativi dell’istruzione
PCSrc	Nel PC viene scritta l’uscita del sommatore che calcola il valore di PC + 4	Nel PC viene scritta l’uscita del sommatore che calcola l’indirizzo di salto
MemRead	Nulla	Il dato della memoria nella posizione puntata dall’indirizzo viene inviato in uscita sulla linea «dato letto»
MemWrite	Nulla	Il contenuto della memoria nella posizione puntata dall’indirizzo viene sostituito con il dato presente sulla linea «dato scritto»
MemtoReg	Il dato inviato al register file per la scrittura proviene dalla ALU	Il dato inviato al register file per la scrittura proviene dalla Memoria Dati

ESERCIZIO su pdf BEEP**Monociclo:**

and t2, t0, t1

beq t2, t0, 257

- CALCOLO T2 dopo and?

0x 0E5FE371

0x FFFF4821

0x 0E5F 4021 -> t2

PC associato alla istruzione?

- PC prodotto al prelievo della istruzione -> PC incrementato di 4

0x 0040 0304 (perché viene dato come origine il PC quando estrae "and" = 0x 0040 0300)

(INDIRIZZO)# reg lettura 1? -> **0A**

(INDIRIZZO)#reg lettura 2? -> **08**

Vedi il fascicolo d'esame per gli indirizzi del banco dei registri

Ingresso 0 del MUX D? secondo registro sorgente della beq, basta vedere il cod op di beq (in formato I) e il fascicolo di esame per vedere quali bit dell'istruzione vanno nell'ingresso 0 del MUX D. -> **08**

Ingresso Estensione Segno? -> **0x 0101 (257 = 256+1)** -> rappresenta la distanza di salto in istruzioni con base di partenza istruzione successiva a beq (PC+4)

Ingresso 2 ALU BRENCH? (Nota -> la ALU BRENCH è solamente un sommatore, il cui risultato viene preso solo se c'è un salto, è usata per calcolare l'indirizzo dell'istruzione destinazione del salto)

0x 0404 (SHIFT a sinistra di 2 del risultato dell'estensione di segno calcolata prima)

Uscita ALU BRENCH?

0x 0404

0x 0308 -> PC + 4

0x 070C

Uscita MUX B? **0x 0308**

ACSO - Architettura dei Calcolatori

Valori dei segnali di controllo a bit singolo? (0,1,X -> indifferenza)

ALUsrc	=	0
Zero	=	0
Jump	=	0
RegDest	=	X
RegWrite	=	0
MemRead	=	0
MemtoReg	=	X

Pipeline: (vedi sempre fascicolo esame per struttura processore)

Indirizzo iniziale: 0X000 0800

0x 000 0800: lw t0, 00A0(t2)

0x 000 0804: add t2, t2, t1

0x 000 0808: sw t2, 000A(t1)

0x 000 080C: beq t0, t1, 128

0x 000 0810: sw t0, 000C(t1)

t0 = 0x 000F C422

t1 = 0x 0004 08C0

t2 = 0x 0040 0800

Considerare il ciclo di clock nel quale l'esecuzione delle istruzioni è il seguente

Iw	WB
add	MEM
sw	EX
beq	ID
sw	IF

Ciclo clock?? **Ciclo 5** (dall'ordine degli stadi di esecuzione delle istruzioni -> 5 istruzioni, la quinta in fetch)

Indirizzo di lettura Iw? calcolo spiazzamento 00A0(t1) = t1 + 00A0 (esteso in segno)

0x 0040 0800 +

0x 0000 00A0

0x 0040 08A0

Indirizzo destinazione del salto? 128 = n istruzioni, allora devo moltiplicarlo x 4 per avere la distanza in parole di memoria -> 0200

Indirizzo istruzione successiva a beq: PC + 4 0 x 0000 0810. Li sommo.

Indirizzo destinazione = **0x 0000 0A10**

Somma contenuto t1 + contenuto t2? (t1) + (t2) = ?

0x 0004 08C0 +

0x 0040 0800

0x 0044 10C0

Segnali dei registri di inter stadio in ingresso prima del fronte di salita?

segnali all'ingresso dei registri di interstadio (subito prima del fronte di SALITA del clock)			
IF	ID	EX	MEM
registro IF/ID	registro ID/EX	registro EX/MEM	registro MEM/WB
	.WB.MemtoReg <i>X</i>	.WB.MemtoReg <i>X</i>	.WB.MemtoReg <i>0</i>
	.WB.RegWrite <i>0</i>	.WB.RegWrite <i>0</i>	.WB.RegWrite <i>1</i>
	.M.MemWrite <i>0</i>	.M.MemWrite <i>1</i>	
	.M.MemRead <i>0</i>	.M.MemRead <i>0</i>	
	.M.Branch <i>1</i>	.M.Branch <i>0</i>	
.PC <i>0000 0814</i>	.PC <i>0000 0810</i>	.PC *****	
.istruzione <i>SW</i>	.(Rs) <i>FFFF AAAA</i>		
	.(Rt) <i>0C04 08C0</i>	.(Rt) <i>0040 0800 (t2 iniziale)</i>	
	.Rt <i>09 - t1</i>	.R *****	.R <i>0A - t2</i>
	.Rd *****		
	.imm/offset esteso <i>0000 0080</i>	.ALU_out <i>0C04 08CA (t1 + offset)</i>	.ALU_out <i>0C44 10C0 (t1+t2)</i>
	.EX.ALUSrc <i>0</i>	.Zero <i>0</i>	.DatoLetto *****
	.EX.RegDest <i>X</i>		

PIPELINING E HAZARD:**Problema dei conflitti:**

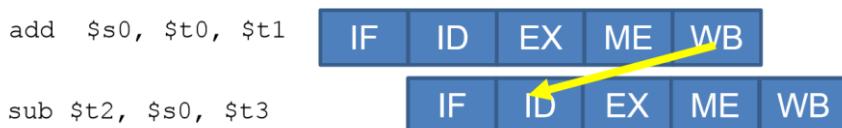
Il pipelining, se progettato male può comportare certi tipi di problemi:

- Conflitti strutturali: tentativo di usare la stessa risorsa da parte di diverse istruzioni in modi diversi nello stesso intervallo di tempo.
- Conflitti sui dati: tentativo di utilizzare un risultato prima che sia pronto.
- Conflitti sul controllo: tentativo di prendere una decisione sulla prossima istruzione da eseguire prima che la condizione sia valutata.

L'architettura MIPS è costruita in modo tale da non causare conflitti strutturali, è anche per questo che si differenzia la memoria istruzioni dalla memoria dati.

Un conflitto di tipo già più complicato è il conflitto di dati: **data hazard**.

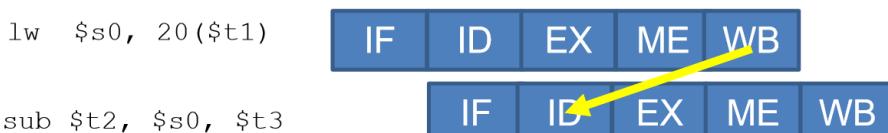
Il data hazard può avvenire nel caso in cui ci sono due istruzioni aritmetico-logiche in sequenza e il risultato della *seconda* delle due dipende dal *risultato* della prima.

Conflitto di dati (data hazard): R/R

Non c'è bisogno che aspetti la fine del ME per scrivere il risultato della add. In questo caso la sub userebbe un risultato senza senso.

Per ovviare al problema si usa una tecnica di **Propagazione/bypassing**: l'uscita dello stadio EX viene anche portata direttamente ai suoi ingressi (propagazione EX/EX).

Questo stesso errore viene riscontrato anche con altri tipi di istruzioni, ad esempio **lw** e **sub** (o più generalmente aritmetico logiche).

Conflitto di dati (data hazard): Load/R

- il valore corretto è disponibile già all'uscita dello stadio **MEM** di **lw**
- circuito di **Propagazione/bypassing**: l'uscita dello stadio MEM viene anche portata direttamente agli ingressi dello stadio EX (MEM/EX), ma non basta
- si deve inserire un ciclo di ritardo nell'esecuzione di **sub**

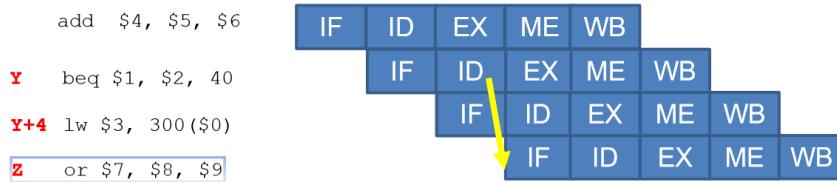
Si deve inserire un ciclo di ritardo nell'esecuzione di sub.

In questo caso il registro \$s0 fa come una "bolla", resta sospeso per un ciclo di clock. (Quindi si pausa, è uno dei fattori di degrado che fanno sì che il fattore di accelerazione della pipeline non sia esattamente uguale al numero di pipeline implementate).

ACSO - Architettura dei Calcolatori

Stesso ragionamento va fatto con le istruzioni di salto condizionato ad esempio **beq**.

Conflitto di controllo (control hazard): **beq**



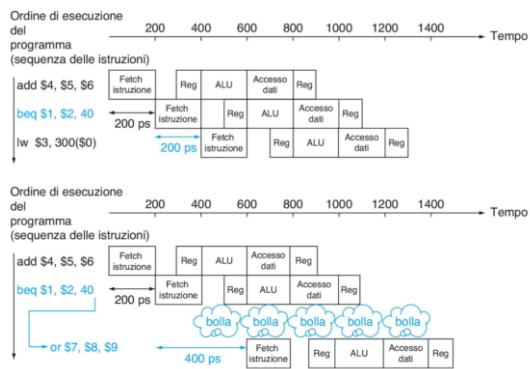
Pipeline ottimizzata

- esito confronto registri e calcolo indirizzo destinazione di salto disponibili alla fine della fase di **ID**

ma non basta

- se **branch untaken** la pipeline lavora "a pieno regime": esecuzione dell'istruzione in sequenza a **Y+4**
- se **branch taken** si deve inserire un ciclo di ritardo prima di eseguire l'istruzione destinazione di salto per avere il PC corretto (**Z**)
se esito e indirizzo salto disponibili alla fine della fase di **EX**, allora due cicli di ritardo

Conflitto di controllo (control hazard): **beq** (cont.)



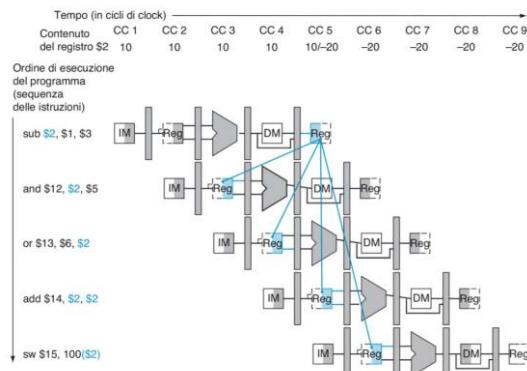
Problema del conflitto di dati

Abbiamo visto che se le istruzioni eseguite nella pipeline sono **dipendenti** tra loro possono nascere problemi dovuti a **conflitti di dati**

Esempio: le ultime quattro istruzioni dipendono dal risultato scritto nel registro **\$2** dalla prima istruzione, ma solo le ultime due istruzioni accedono al valore corretto:

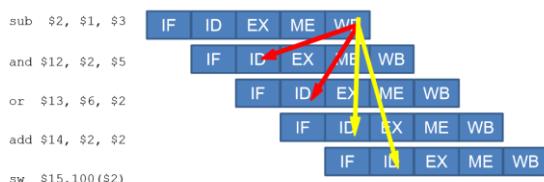
```
sub $2, $1, $3    # Reg. $2 scritto dalla istruzione sub
and $12, $2, $5   # Il 1° operando($2) dipende dalla sub
or $13, $6, $2    # Il 2° operando($2) dipende dalla sub
add $14, $2, $2    # 1° ($2) & 2° ($2) dipendono dalla sub
sw $15, 100($2)  # Il reg. indice($2) dipende dalla sub
```

Dipendenze di dato, risorse e registro coinvolto



Dipendenze di dato e stadi

Le dipendenze di dato diventano **conflitti** se "vanno all'indietro nel tempo"



Soluzioni al problema del conflitto di dati

Tecniche di compilazione del codice:

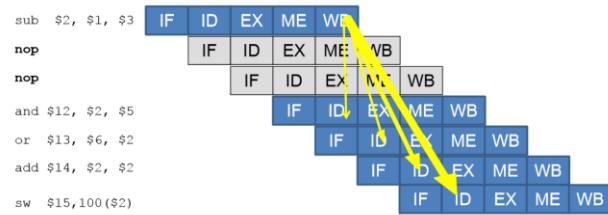
- Inserimento di istruzioni **nop** (*no operation*).
- *Scheduling* o riordino delle istruzioni in modo da impedire che istruzioni correlate siano troppo vicine.
 - Il compilatore cerca di inserire tra le istruzioni correlate (che presentano dei conflitti) delle istruzioni *indipendenti* dal risultato delle precedenti operazioni, facendo così scomparire tutti i conflitti.
 - Quando il compilatore non riesce a trovare istruzioni indipendenti deve inserire istruzioni **nop**.

Tecniche di tipo hardware:

- Inserimento di "bolle" o stalli nella pipeline.
- Propagazione dei dati in avanti i dati (*forwarding or bypassing*)

Inserimento di **nop**: Esempio

Il compilatore deve inserire tra le istruzioni **sub** e **and** due istruzioni **nop**, facendo così scomparire tutti i conflitti.



Scheduling: Esempio

Esempio:

sub \$2, \$1, \$3	sub \$2, \$1, \$3
and \$12, \$2, \$5	add \$4, \$10, \$11
or \$13, \$6, \$2	and \$7, \$8, \$9
add \$14, \$2, \$2	and \$12, \$2, \$5
sw \$15, 100(\$2)	or \$13, \$6, \$2
add \$4, \$10, \$11	add \$14, \$2, \$2
and \$7, \$8, \$9	sw \$15, 100(\$2)
lw \$16, 100(\$18)	lw \$16, 100(\$18)
lw \$17, 200(\$19)	lw \$17, 200(\$19)



Inserimento di bolle o stalli: Esempio

Una possibile soluzione di tipo *hardware* consiste nell'inserire delle "bolle" nella pipeline, cioè *bloccare il flusso di ingresso di istruzioni* nella pipeline fino a quando il conflitto non è risolto. Lo stato in cui si trova la CPU quando le istruzioni sono bloccate è indicato con il termine "stallo".

Nell'esempio significa inserire 2 bolle o stalli per fermare le istruzioni che seguono l'istruzione **sub** affinché possano essere letti i dati corretti.

Esecuzione delle istruzioni di salto condizionato (cont.)

La base per il calcolo dell'indirizzo di destinazione di un salto condizionato è l'indirizzo dell'istruzione che segue quella di salto (PC + 4).

Poiché ogni istruzione occupa 4 byte, prima di effettuare la somma dello spiazzamento (*offset*) con il contenuto di (PC+4), bisogna **moltiplicare per 4** il valore contenuto nell'istruzione
⇒ **scorrimento a sinistra di 2 bit**.

L'operazione di salto condizionato è costituita da due operazioni:

- Calcolo dell'indirizzo di destinazione del salto attraverso un sommatore che effettua la somma tra il (PC+4) e il valore contenuto nell'istruzione dopo avere esteso il segno e fatto scorrere a sinistra di 2 bit;
- Confronto nell'ALU del contenuto dei registri operandi letti dal RF. Se il segnale di uscita Zero dell'ALU è assertivo ⇒ la condizione di salto è verificata e l'indirizzo di destinazione del salto diventa il nuovo PC. Se invece la condizione non è verificata ⇒ il PC incrementato sostituisce il PC attuale.

Realizzazione del processore completo

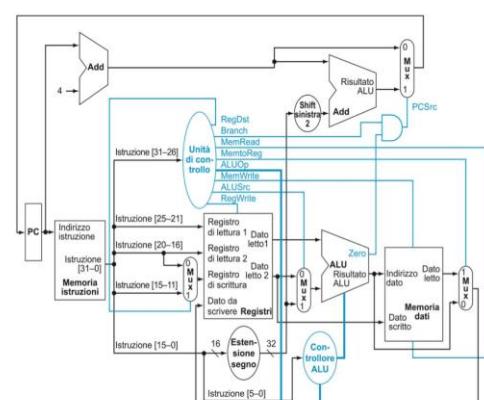
Esaminati gli elementi richiesti da ogni tipo di operazione, è possibile combinarli in un'unica unità di elaborazione.

Si assume che tutte le istruzioni siano eseguite in un solo ciclo di clock

- Nessuna risorsa può essere utilizzata più di una volta per istruzione
- Qualsiasi risorsa di cui si ha bisogno più di una volta deve essere duplicata

Occorre quindi una *Memoria Istruzioni* **distinta** dalla *Memoria Dati*.

Struttura della CPU con unità di controllo



Esecuzione delle istruzioni aritmetico-logiche e load/store

Da risolvere:

- Il secondo ingresso dell'ALU è il contenuto di un registro (istruzione di tipo R) oppure la metà meno significativa dell'istruzione (istruzione di *load/store* o *aritmetiche immediate*)
⇒ **MUX al secondo ingresso dell'ALU (Mux A)**
- Il valore scritto nel registro destinazione proviene dal risultato dell'ALU (istruzione tipo R o aritmetica immediata) oppure dalla Memoria Dati (istruzione di *load*)
⇒ **MUX all'ingresso dei Dati per Scrittura del RF (Mux C)**
- Il numero del registro in cui si vuole scrivere il risultato è indicato da diversi campi (i bit [15-11] per le istruzioni di tipo R e bit [20-16] per istruzioni di *load/store* e *aritmetiche immediate*)
⇒ **MUX all'ingresso del Registro Scrittura del RF (Mux D)**

GERARCHIA DI MEMORIA:

Il problema fondamentale delle memorie fisiche contemporanee è che non riescono a soddisfare contemporaneamente i requisiti fondamentali ipotizzati nella costruzione del processore:

- Accedere a una parola di memoria in un solo ciclo di clock del processore
- Essere così grande da contenere qualsiasi programma (cioè avere uno spazio di indirizzamento fisico uguale allo spazio di indirizzamento virtuale)

Per superare questo problema è necessario unire diverse tecnologie di costruzione delle memorie stabilendo una **gerarchia**.

L'idea di base della costruzione della gerarchia è quella di dividere le memorie nel modo seguente.

Avendo una memoria A veloce, ma piccola e una memoria B grande, ma lenta:

- Tengo tutto il codice e i dati del programma nella memoria B
- Porto nella memoria A solo le porzioni di programma che servono durante una certa fase dell'esecuzione

Nota che questa idea funziona solamente se il programma usa relativamente a lungo la porzione di programma contenuta nella memoria veloce prima di richiedere di sostituirla con un'altra.

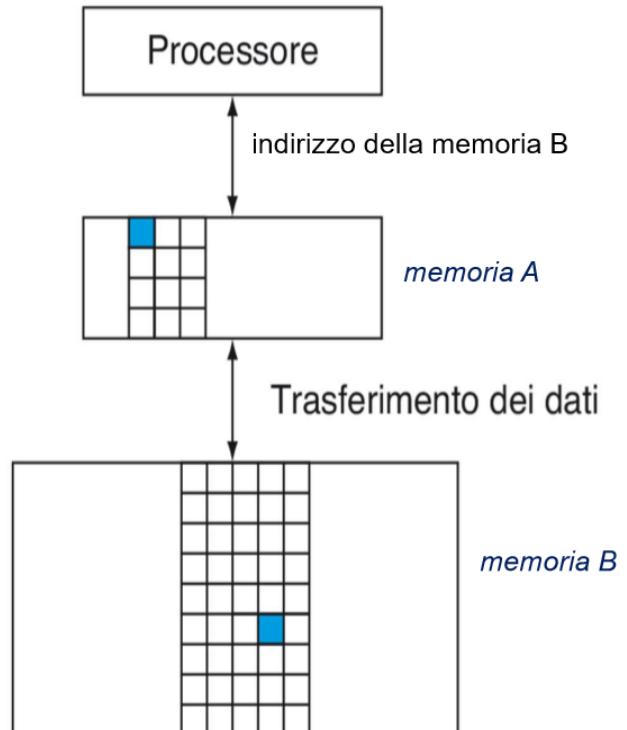
Esempio di **gerarchia a 2 livelli**:

Il processore genera indirizzi della memoria B

Deve essere possibile derivare da un indirizzo della memoria B il "corrispondente" della memoria A

Il processore lavora sulla memoria A, più piccola e veloce

Quando richiede un indirizzo che risulta non presente in memoria A, un intero blocco viene trasferito dalla memoria B alla memoria A



Il processore non si accorge in nessun caso della divisione, casomai viene ogni tanto "messo in pausa", quindi si arresta temporaneamente l'esecuzione delle istruzioni. Un caso di esempio può essere:

- In memoria A non è presente l'istruzione di cui fare il *fetch*
- Il processore viene messo in pausa per permettere di trasferire un blocco di istruzioni nella memoria A
- L'esecuzione riprende normalmente

Naturalmente se il processore viene arrestato troppo spesso c'è qualcosa che non va con il sistema della memoria.

Per sopperire al problema dell'arresto temporaneo troppo frequente è necessario che le memorie siano ben organizzate nello spostamento dei blocchi da A a B. Ma come si può fare? Esiste il principio di **località** che viene molto utile in questo:

Località temporale: indica che un programma tende ad accedere agli stessi indirizzi di memoria a cui ha già acceduto di recente (*cicli*). - istruzioni

Località spaziale: indica che un programma accede con maggior probabilità agli indirizzi vicini a quelli a cui ha già acceduto in passato (*strutture dati lineari: liste*). – dati

(Il principio di località temporale è generalmente più forte del secondo, siccome un programma ha per forza almeno un ciclo, dato che altrimenti sarebbe una lista di istruzioni da eseguire in modo sequenziale che finisce subito)

La gestione della memoria è quindi schematizzata nel modo seguente:

- Carica il programma nella memoria B
- Quando il programma richiede una parola ad un certo indirizzo, trasferisci il **BLOCCO** di parole che contiene la parola all'indirizzo richiesto dalle memoria B alla memoria A (blocco per località spaziale)
- Grazie alla località spaziale e temporale il programma chiederà di accedere molte volte a parole del blocco caricato in memoria A; questi accessi sono definiti **HIT**.
- Prima o poi il programma chiederà di accedere ad un indirizzo non presente in memoria A: questo accesso è detto **MISS**
- Quando si verifica un **MISS** un nuovo blocco, contenente la parola a cui accedere, viene caricato dalla memoria B alla memoria A
- La memoria veloce non è costituita da un unico blocco, ma da molti blocchi, perché la località temporale può riguardare dati non contigui in memoria
- Fino a quando la memoria veloce ha blocchi liberi, in risposta a una **MISS** viene caricato un nuovo blocco nello spazio libero
- Quando però la memoria veloce è tutta occupata, per caricare un blocco è necessario scegliere un blocco già caricato da liberare.

È quindi necessaria una politica di sostituzione dei blocchi. Quale posso scegliere?

- Casuale
- **LRU** (Last Recently Used): sostituisce il blocco **utilizzato meno di recente**, perché in base al principio di località spaziale, questo ha la probabilità più bassa di essere richiesto nel prossimo futuro
- **FIFO** (First In First Out): sostituisce sempre il blocco **caricato meno di recente**, indipendentemente da quando si sia fatto riferimento a tale blocco

Qual è la migliore da implementare? La LRU sarebbe ideale, ma è troppo difficile da realizzare (anche se le memorie moderne la usano), è necessario definire dei parametri secondo cui scegliere la migliore implementazione della politica di sostituzione scelta.

Parametri:

- **h = HIT rate:** è la percentuale di accessi che vengono soddisfatti dalla memoria A
- **m = MISS rate:** è la percentuale di accessi che ha richiesto di caricare un nuovo blocco dalla memoria B alla memoria A ($n^{\circ}MISS/n^{\circ}\text{totale accessi}$) **m = 1 - h**
- **M = MISS penalty:** è il tempo richiesto per svolgere l'operazione di caricamento di un blocco dal livello di memoria più lento

L'idea base funziona in pratica se la località dei programmi e la dimensione della memoria veloce permettono di raggiungere *HIT rate* sufficientemente elevati da compensare *ampiamente* le *MISS penalty*.

Attualmente la gerarchia a 2 livelli è superata, è considerata non sufficiente a coprire le esigenze dei programmi. Sono quindi state costruite **gerarchie a più livelli**, anche se per ogni coppia di livelli si applicano in linea di principio le considerazioni fatte per l'architettura a 2 livelli (differiscono ovviamente nei piccoli dettagli).

Tecnologie di memoria:

Le tecnologie di memoria attualmente utilizzate sono le seguenti, con le loro caratteristiche

livello	velocità	tipiche dimensioni
SRAM	0,2 – 2,5 ns	decine di Kbyte
DRAM	50 – 70 ns	alcuni Gbyte
Flash	0,005 – 0,05 ms	centinaia Gbyte
Dischi	5 – 20 ms	molti Tbyte

Livello	Tecnologia	Nome dell'Unità trasferita	Dimensione tipica dell'unità trasferita	Dimensione tipica della memoria
Cache (1° liv)	SRAM			16 – 64 Kb
Cache (2° liv)	SRAM (+ lente e + grandi)	Blocco	16 - 64	125 – 2000 Kb
Memoria Centrale	DRAM	Blocco	64 - 128	< 4 Gb
Memoria Virtuale	Flash o Disco	Pagina	> 4 Kbyte	

Nota che è anche importante la dimensione dei collegamenti delle memorie ad esempio cache di primo livello, perché in termini di collegamento i picosecondi contano.

MEMORIA CACHE: (copia d'uso della memoria centrale)

La memoria cache contiene **copie di blocchi della memoria centrale**, o **blocchi liberi** inizialmente vuoti.

Ad ogni blocco è associato un **bit "valid"** che indica se il blocco è una copia valida o meno.

Il sistema di gestione della cache è in grado di operare copie e ricopie dei blocchi di memoria, può caricare blocchi dalla memoria centrale e scaricare blocchi nella memoria centrale.

Il **processore** accede sempre e comunque prima alla memoria cache., può anche **scriverci**, ed è per questo che si scaricano i blocchi dalla cache alla memoria centrale.

Con riferimento all'architettura MIPS studiata, supporremo che esistano due cache di 1° livello, una per le istruzioni e una per i dati.

Funzionamento base:

Istruzioni:

- il processore deve leggere un'istruzione
- Se il blocco che contiene l'istruzione da prelevare **si trova** in memoria cache l'istruzione viene letta ed eseguita (**HIT**)
- Se l'istruzione **non si trova** in cache:
 - a. Il processore sospende l'esecuzione (stallo di pipeline)
 - b. Il blocco contenente l'istruzione viene **caricato** dalla memoria centrale in un blocco libero della memoria cache
 - c. Il processore **preleva** l'istruzione dalla cache e prosegue l'esecuzione

Dati:

- Il processore deve **leggere** un dato dalla memoria
- Se il blocco che contiene il dato da prelevare **si trova** in memoria cache il dato viene letto ed utilizzato (**HIT**)
- Se il dato **non si trova** in cache:
 - a. Il processore sospende l'esecuzione (stallo di pipeline)
 - b. Il blocco contenente il dato viene **caricato** dalla memoria centrale in un blocco libero della memoria cache
 - c. Il processore **preleva** il dato dalla cache e prosegue l'esecuzione
- Il processore deve **scrivere** un dato in memoria:
 - a. Se il dato non è presente in memoria cache, il blocco che lo contiene deve essere caricato in cache (per mantenere la coerenza del dato da scrivere con gli altri dati nel suo blocco in cache)
 - b. Il dato viene scritto in cache: scrittura differita (**Write Back**) o non differita (**Write Through**: il blocco viene scritto direttamente anche in memoria)

Write Through:

L'informazione viene scritta sia nel blocco del livello superiore (cache) sia nel blocco di livello inferiore della memoria (memoria centrale o cache di 2° livello).

Per evitare uno stallo dovuto alla scrittura sul livello inferiore viene utilizzato un buffer di scrittura e la scrittura effettiva avviene senza bloccare il processore.

Write Back:

L'informazione viene scritta solo nel blocco di livello superiore (cache). Il livello inferiore viene aggiornato solo quando avviene la sostituzione del blocco.

Per ogni blocco di cache è necessario mantenere l'informazione sulla scrittura (bit MODIFICA che indica se il blocco in cache è stato modificato o meno e va quindi copiato in memoria in caso di sostituzione di questo blocco in cache con un altro blocco)

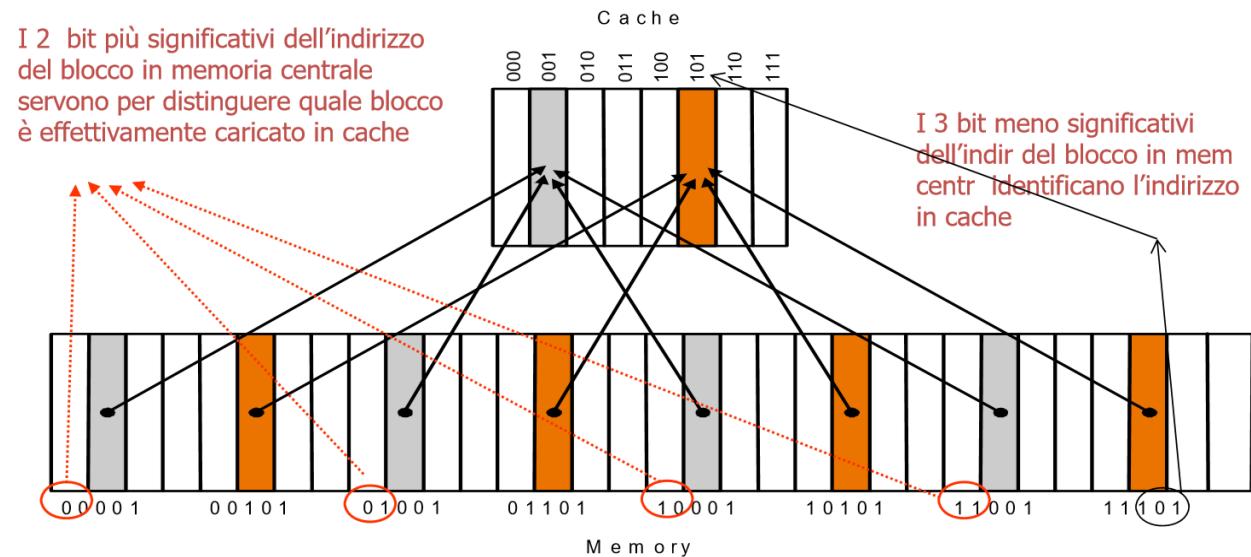
Cache a indirizzamento diretto:

Ogni blocco della memoria centrale è caricabile in un solo blocco della cache. Più blocchi di memoria centrale potrebbero dover essere caricati nello stesso blocco di memoria cache, in alternativa si verificherà un **conflitto**.

Per evitare i conflitti si utilizza una tecnica di gestione della cache detta tecnica di **Mapping**: il blocco di indice (indirizzo) j della memoria centrale è caricabile solo nel blocco di cache di indice (indirizzo) = **resto div intera di j / n blocchi della cache**.

Visto che la divisione è un'operazione complicata da implementare, anche in esadecimale a livello di istruzione, si usano indirizzi di mapping **binari**, tali che il resto della divisione di j per 2^n sono i primi n bit meno significativi di j .

Questa semplificazione «operativa» richiede che il numero di blocchi in cache, il numero di parole nel blocco e il numero di byte nella parola siano tutti potenze di 2.



DA QUI FINO A PAG 11 NON HAI SEGUITO IDIOTA LEGGILE

L'idea della cache a indirizzamento diretto è molto semplice.

Cache Associative (opposte a indirizzamento diretto):

Totalmente associativa: Un blocco di memoria centrale può essere memorizzato in un qualunque blocco della cache (pochi conflitti di allocazione). Non esiste una relazione tra indirizzo di memoria del blocco e posizione in cache. Non esiste quindi problema di mapping. C'è un completo rilassamento dei vincoli, anche quando devo sostituire un blocco in cache con un blocco della memoria centrale scelgo un qualunque blocco in cache (o scelgo il dato meno utilizzato di recente). Ci sono meno conflitti e l'indirizzo è analizzato in modo più semplice.

MA la ricerca di un blocco di memoria in cache dove andare a prendere il dato che mi serve si allunga perché devo andare a vedere tutte le etichette di tutti i blocchi, perché non ho un sistema di mapping per sapere a priori dove cercare. Questa ricerca deve essere eseguita in modo parallelo perché altrimenti

vanificherei il principio della cache che è avere una memoria molto veloce. Per sopperire a questo problema devo usare delle memorie di tipo **associativo** che svolgono da sole la ricerca del blocco di memoria basandosi sull'indirizzo. Queste memorie sono costose e non consentono la realizzazione di memorie molto capienti.

Set - Associative a n vie: I blocchi della cache sono divisi in **gruppi**. n indica la dimensione (in blocchi) del gruppo. La cache si indirizza per gruppi. Ogni blocco della memoria centrale può essere caricato in un solo gruppo(prefissato), in uno qualsiasi degli n blocchi.

Questo metodo di gestione della cache è un misto tra l'indirizzamento diretto e l'indirizzamento totalmente associativo. Se non sapessi dell'esistenza dei blocchi all'interno dei gruppi la modalità sarebbe quella a indirizzamento diretto, mentre i blocchi di ogni gruppo vengono gestiti in modo totalmente associativo.

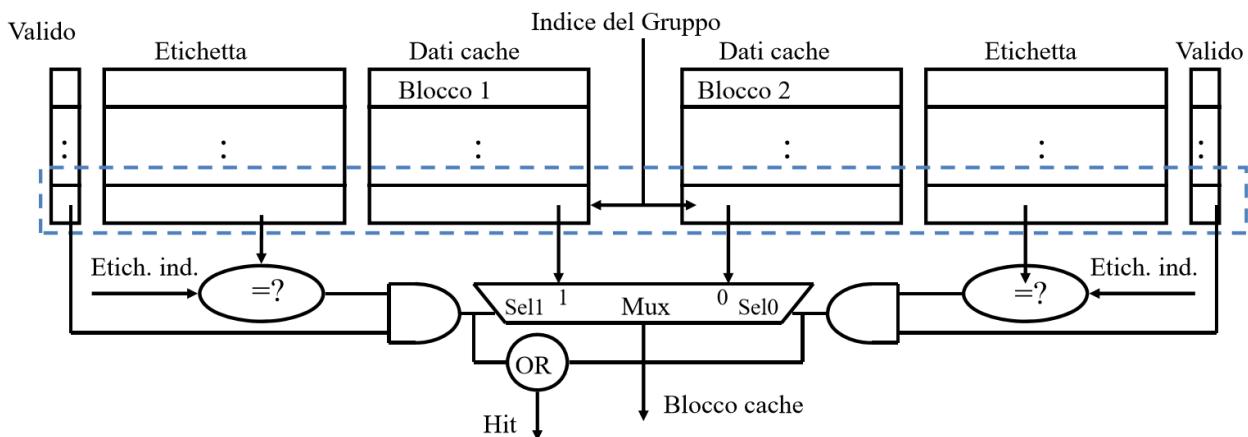
Come politica di sostituzione dei blocchi si usa LRU (Last Recently Used) in modo approssimativo. Nel caso di una cache *set-associative a 2 vie* è necessario solamente un bit d'uso che funziona come flag. Nel caso in cui tutti e 2 i flag fossero attivi posso operare una scelta casuale, oppure, prefissato un lasso di tempo standard, posso azzerare i bit d'uso allo scadere di ogni lasso di tempo. Nel caso di più di 2 vie questo sistema non è applicabile perché rallenterebbe tutto, quindi lo si usa approssimativamente e se ho dei **miss** pazienza.

Un indirizzo di memoria di N bit è suddiviso in 4 campi:

- B bit meno significativi per individuare il byte all'interno della parola
- K bit per individuare la parola all'interno del blocco
- M bit per individuare il gruppo, indice del gruppo (indirizzamento diretto per il gruppo)
- $N-(M+K+B)$ come etichetta, quale blocco nel gruppo (completamente associativa nel gruppo)

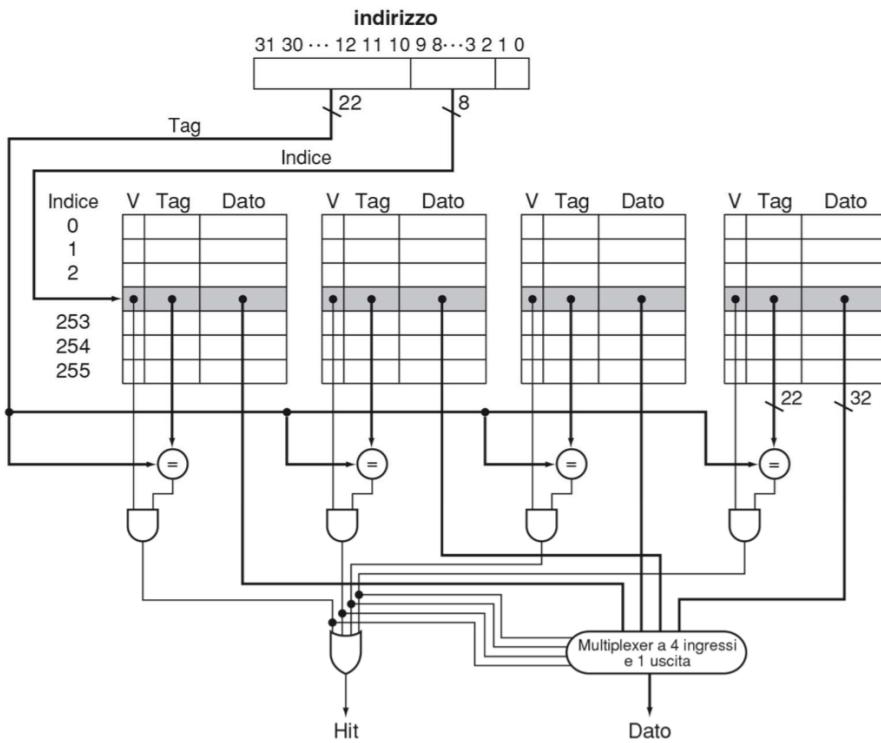
Nel caso della **cache set-associative a 2 vie** in pratica è come avere due cache a indirizzamento diretto identiche affiancate che vengono usate insieme, accedendo ai due blocchi corrispondenti nelle due cache, e scegliendo quello più conveniente dei due.

Un buon esempio grafico di funzionamento di una cache set-associative a 2 vie è il seguente:



Esempio di indirizzamento a 4 vie:

- Bit 0 e 1 per indirizzare i byte
- Numero blocchi nella cache = dimensioni della cache/dimensione del blocco = $2^{10}/1 = 2^{10}$
- Numero di gruppi nella cache = numero di blocchi cache/ dimensione del gruppo
 $= 2^{10}/2^2 = 2^8 = 256$ gruppi
- Bit 2-10 indice del gruppo nella cache
- Bit 31-11 etichetta

Esempio di struttura di memoria a 4 vie

Il problema che risolve è: nell'architettura a 2 vie posso caricare al massimo 2 blocchi di memoria di indice pari in un blocco, per motivi legati all'indirizzamento e gli indirizzi in memoria, mentre a 4 vie raddoppio il numero di blocchi memorizzabili.

Conclusioni: una cache a indirizzamento diretto è utilizzata molto frequentemente per la **cache istruzioni**, perché tutti i programmi rispettano la località temporale (essendo per la maggior parte composti da cicli). Le cache set-associative vengono utilizzate per la memoria dati, perché essa è molto probabile che risetti il principio di località spaziale.

Nel caso di cache set-associative a 8 vie ho una frequenza di miss dell' 8,1%.

Prestazioni:**Definizioni:**

- $h = \text{HIT rate}$ (h_I , h_D per distinguere tra istruzioni e dati, se necessario)
- $m = \text{MISSrate} = 1 - h$ (m_I , m_D per distinguere tra istruzioni e dati, se necessario)
- $M = \text{MISS penalty}$
- $\text{StalliM}(P) = \text{numero di stalli dovuti alla memoria nella prova } P$

il tempo di esecuzione del programma diventa:

$$T(P) = (\text{Cicli}(P) + \text{StalliM}(P)) * CK$$

Esempio di esercizio:

StalliM(P) = numero di stalli dovuti alla memoria
 = stalli in lettura + stalli in scrittura

Ipotesi - penalità di Miss in lettura e scrittura (write through) identiche, quindi

$$\text{StalliM}(P) = \text{accessi alla memoria}(P) * m * M$$

La miss penalty **M** (in cicli) può essere calcolata come:

$$M = \text{dimensione del blocco (in parole)} * \text{tempo di accesso (in cicli) a una parola della Memoria di livello inferiore}$$

Dati:

$$\begin{aligned} m_I &= 2\%, \quad m_D = 4\%, \quad M = 100 \text{ cicli}, \\ \text{CPI (esclusi stalli memoria)} &= 2 \text{ cicli} \\ \text{percentuale } Iw \text{ e } sw &= 36\% \end{aligned}$$

determiniamo

$$\begin{aligned} \text{cicli per miss I} &= NI * 2\% * 100 = 2 * NI \\ \text{cicli per miss D} &= NI * 36\% * 4\% * 100 = 1,44 * NI \\ \text{cicli totali per miss} &= 3,44 * NI; \\ \text{CPI (totale)} &= 2 + 3,44 = 5,44 \end{aligned}$$

Confronto con cache ideale (priva di miss)

$$\begin{aligned} \text{CPI (con cache reale) / CPI (cache ideale)} &= 5,44/2 = 2,72 \\ \text{tempo speso in stalli di memoria} &= 3,44/5,44 = 63\% \end{aligned}$$

Se acceleriamo il processore:

- $\text{CPI} = 1 \rightarrow \text{con cache reale: } 4,44 / 1 = 4,44$
- $\text{tempo speso in stalli di memoria} = 3,44/4,44 = 77\%$

Tempo medio di accesso alla memoria AMAT: **Average Memory Access Time****AMAT = tempo di hit + freqmiss * penalità**

MEMORIA VIRTUALE:

Lo scopo della memoria virtuale è quello di separare il concetto di spazio di indirizzamento di un programma eseguibile e dimensione effettiva della memoria fisica. Lo spazio di indirizzamento è definito dal numero di parole indirizzabili e dipende esclusivamente dal numero di bit dell'indirizzo e non dal numero di parole di memoria effettivamente disponibile. La dimensione della memoria fisica è pari al n° di byte (parole) che la costituiscono, questo è piuttosto limitativo.

Il processore produce indirizzi virtuali, quindi il programma in esecuzione fa riferimento a **indirizzi virtuali**, che sono indirizzi generati dal linker a partire dall'indirizzo 0. Questi indirizzi però non si riferiscono a quel reale indirizzo, ma sono **rilocabili**.

Lo spazio di indirizzamento virtuale è quello definito dalla lunghezza degli indirizzi virtuali (x64 (con solo 48 bit utilizzabili per l'indirizzo) -> 256 TB).

In un programma è possibile quindi definire la **dimensione virtuale** (in termini di indirizzi virtuali).

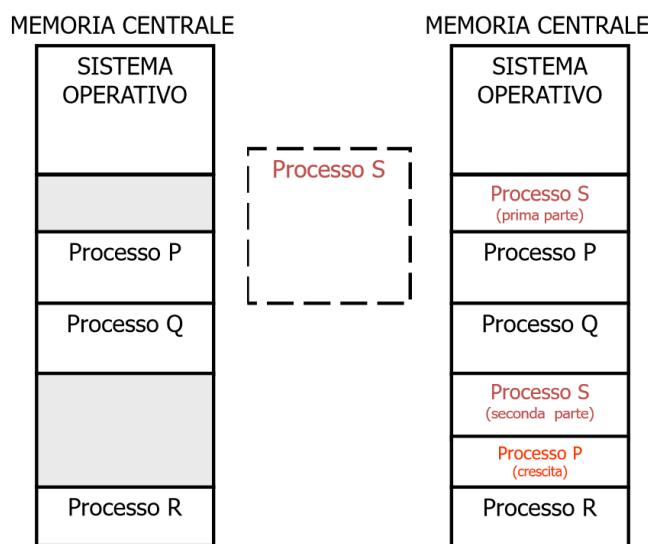
La memoria fisica e la memoria virtuale NON coincidono. E siccome gli indirizzi in memoria centrale sono **fisici**, deve esserci un meccanismo di traduzione da indirizzo virtuale a indirizzo fisico da trasmettere al sistema di memoria. Questo meccanismo di traduzione dell'indirizzo è detto **memory mapping**.

La traduzione automatica degli indirizzi virtuali in indirizzi fisici (rilocazione) è attuata durante l'esecuzione del programma a ogni riferimento a memoria (dinamica), è svolta dal MMU (Memory Management Unit) ed è completamente trasparente a programmatore, assemblatore e linker. Questa traduzione consente il caricamento del programma in una qualsiasi locazione della memoria fisica. La traduzione deve avvenire ogni volta che il processore genera un indirizzo virtuale.

Processi:

Un processo è una sorta di macchina virtuale realizzata combinando SW e HW che viene a costituire l'esecutore per uno specifico programma. Nel processo c'è il programma inteso come codice, ma altri elementi (risorse) come informazioni sul programma, sulle periferiche ecc. Quando un processo è in esecuzione e sta eseguendo un programma può anche mettersi a eseguire un altro codice, senza mutare in nessuna sua parte (tranne in quella del codice ovviamente). È creato dal Sistema Operativo al lancio di un programma. Il concetto di processo consente un vero e proprio parallelismo di esecuzione di diversi programmi.

Dal punto di vista della memoria questo significa che la memoria è frammentata e suddivisa tra tanti processi che eseguono programmi diversi (o anche lo stesso programma).



Attenzione: dato che più processi e il Sistema Operativo possono risiedere contemporaneamente in memoria indipendentemente dalle dimensioni effettive della memoria centrale e la dimensione di memoria virtuale di un singolo programma e/o di più programmi in memoria può essere **maggior**e della dimensione della memoria fisica.

È quindi necessario che **un programma in esecuzione non risieda completamente in memoria**.

- le parti di programma caricate in memoria sono dette residenti
- in ogni momento vengono rese residenti, cioè sono caricate in memoria, le parti di programma che servono per l'esecuzione
- il meccanismo di caricamento in memoria delle parti del programma e la traduzione degli indirizzi siano trasparenti al programmatore

Il metodo utilizzato per realizzare questa architettura della memoria virtuale è detto metodo di **paginazione**. La memoria virtuale del programma viene suddivisa in porzioni di lunghezza fissa dette **pagine virtuali** aventi una lunghezza che è una potenza di 2. Mentre la memoria fisica viene anch'essa suddivisa in pagine fisiche della stessa dimensione delle pagine virtuali. Ma le pagine virtuali di un programma da eseguire vengono caricate in altrettante pagine fisiche, prese arbitrariamente e non necessariamente contigue.

La paginazione ha l'effetto di ridurre il fenomeno della frammentazione della memoria.

È possibile gestire facilmente la crescita di memoria di un processo durante l'esecuzione.

È possibile rendere **Residenti** o **Non Residenti** singole pagine del processo.

Per realizzare la paginazione è necessario stabilire un sistema di **scomposizione dell'indirizzo virtuale**.

L'**indirizzo virtuale** può essere visto come

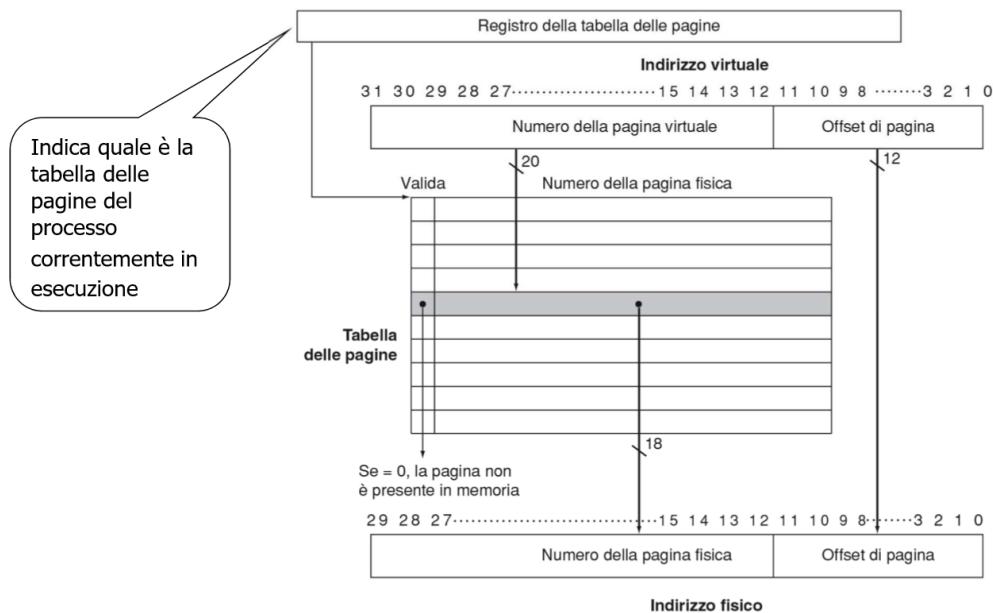
Numero Pagina Virtuale (NPV)	Spiazzamento nella pagina
------------------------------	---------------------------

Lo spazio di indirizzamento virtuale di ogni programma è lineare (indirizzi virtuali contigui) ed è suddiviso in un numero intero di pagine di dimensione fissa e potenza di 2, la dimensione tipica di una pagina è > 4 KB.

L'unico modo per **mappare** la corrispondenza memoria virtuale – memoria fisica è quello di creare una **tabella delle pagine**. Ne esiste una per ogni processo e appartiene concettualmente al descrittore di esso. Contiene una riga per ogni pagina virtuale dello spazio di indirizzamento del processo. NPV può essere quindi utilizzato come indice della tabella delle pagine.

Questo metodo consente facilmente anche la **condivisione delle pagine**. Un processo P a pagina virtuale 100 può avere lo stesso valore che ha un processo Q a pagina virtuale 123.

Come la MMU realizza fisicamente la traduzione indirizzo virtuale – fisico.

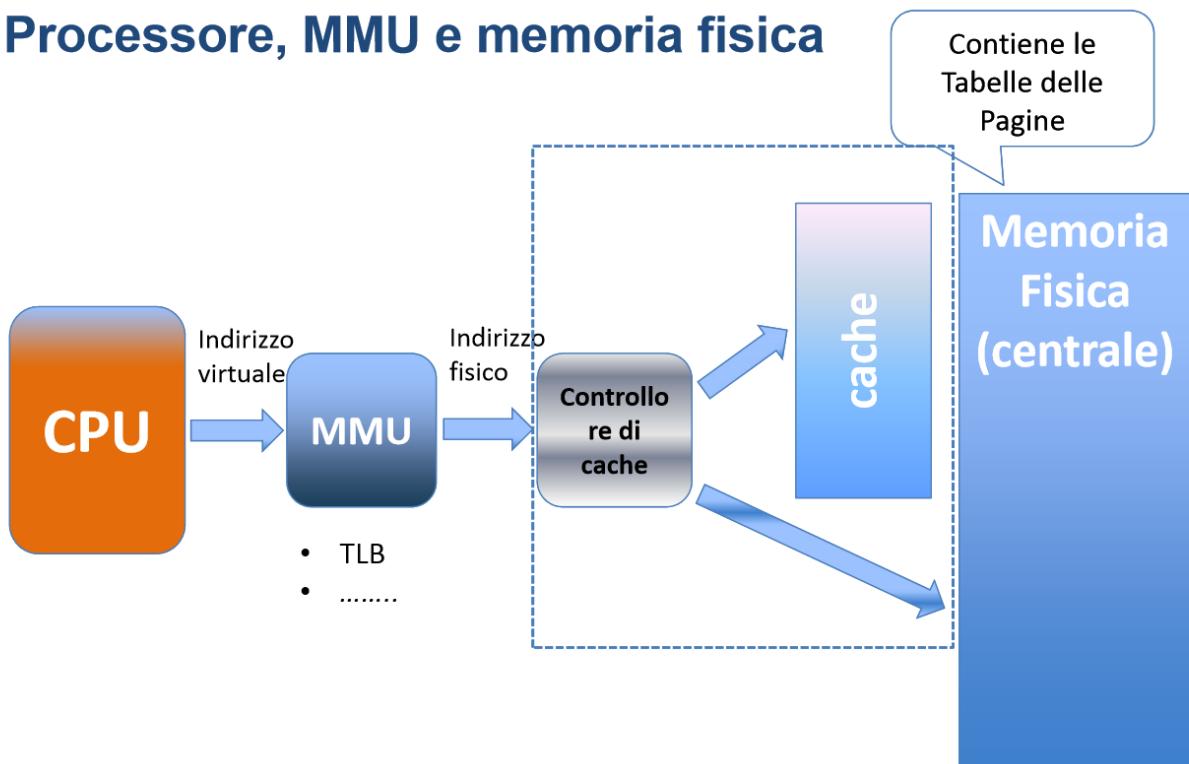


Il fatto è che la tabella delle pagine assume dimensioni enormi perché un processo può avere *milioni* di pagine virtuali. Pr il momento prendiamo questo metodo per buono, perché esistono tabelle di grandi dimensioni anche nella realtà delle implementazioni. (Anche se la MMU realmente non può accedere a tutta la tabella, accede solo alla parte che ci interessa, ovvero a dove essa si riferisce alle pagine **residenti**).

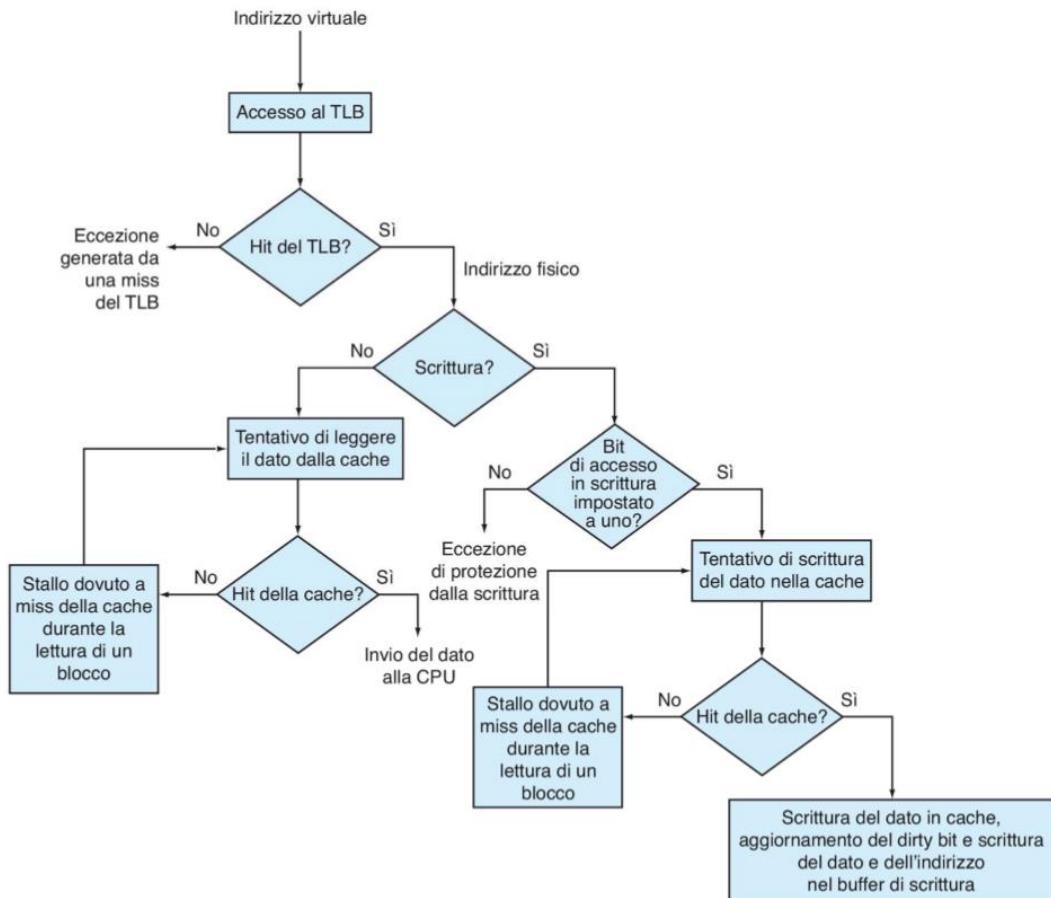
TLB (Translation Lookaside Buffer):

Dispositivo fisico presente nel processore che sfrutta il principio di località degli accessi. Contiene una piccola porzione della tabella delle pagine per velocizzare il processo di traduzione degli indirizzi. Se non è presente la porzione di tabella cercata interviene il SO a sostituire il contenuto del TLB. Il concetto è simile a quello della cache in linea di principio (anche se la cache è completamente hardware e questa parte è hardware solo nel TLB).

Processore, MMU e memoria fisica



Algoritmo da indirizzo virtuale a indirizzo fisico:



ESERCIZI:**Memoria Cache (da materiali esercitazione):****Esercizio 1 pdf:**

La memoria cache è indirizzata a livello del singolo byte, cosa vuol dire? Riceve un indirizzo di 10 bit (sufficiente a indirizzare gli 1KB). L'indirizzo di memoria centrale è di 2^{12} Byte, quindi usa indirizzi a 12 bit. I blocchi sono da 256 Byte. Lo spiazzamento del byte nel blocco richiede 8 Bit ($256 = 2^8$), che sono gli 8 bit meno significativi dell'indirizzo da 12 bit. La parte rimanente deve contemplare il formato dell'indirizzamento diretto, numero del blocco e etichetta.

Numero di blocchi in cache: $1K/256 = 2^2 = 4$.

Numero di blocchi in memoria centrale: $4K/256 = 2^4 = 16$ (da 0 15)

Visto che ho la capacità della cache = 4 blocchi per indirizzare il numero del blocco in cache userò **2 bit**.

I restanti 2 bit formano l'etichetta.

Simulare un sistema di memoria cache a indirizzamento diretto che riceve in ingresso l'indirizzo indicato in tabella e riportare i valori richiesti.

La situazione iniziale descritta dalla prima riga indica lo stato iniziale della cache: la cache è già piena, con blocchi validi e invalidi.

Ricorda che nel blocco 0 (secondo le politiche di indirizzamento della cache a indirizzamento diretto:

- il blocco 0 può contenere solo b0, b4, b8 e b12)
- il blocco 1 può contenere solo b1, b5 b9 e b13)
- il blocco 2 può contenere solo b2, b6, b10 e b14)
- il blocco 3 può contenere solo b3, b7, b11 e b15)

Quando un blocco non è valido non riportare nessun altro dato.

Esempio:

Indirizzo 1: 1110 1110 0010

- 11 -> etichetta
- 10 -> indice di blocco
- 1110 0010 -> spiazzamento in byte all'interno del blocco

Soluzione:

passo	indirizzo richiesto	esito	blocco 0_cache			blocco 1_cache			blocco 2_cache			blocco 3_cache			azione
			valido	etichetta	dati										
0			1	00	0	0	-	-	1	01	6	0	-	-	situazione iniziale
1	1110 1110 0010	M	1	00	0	0	-	-	1	11	14	0	-	-	carica blocco 14 in 2_cache
2	0001 1110 0101	M	1	00	0	1	00	1	1	11	14	0	-	-	carica blocco 1 in 1_cache
3	0001 0011 1001	H	1	00	0	1	00	1	1	11	14	0	-	-	accesso a blocco 1_cache
4	1100 0000 0010	M	1	11	12	1	00	1	1	11	14	0	-	-	carica blocco 12 in 0_cache
5	0011 1111 1110	M	1	11	12	1	00	1	1	11	14	1	00	3	carica blocco 3 in 3_cache
6	1001 0100 0111	M	1	11	12	1	10	9	1	11	14	1	00	3	carica blocco 9 in 1_cache

Esercizio 2 pdf:

La memoria cache è di tipo cache *set-associative a 2 vie*.

- La dimensione della cache è di 512 B
- La memoria centrale è di 4 KB, indirizzata a livello del singolo byte
- Ogni blocco della cache contiene 128 B, quindi la cache contiene 4 blocchi, denotati dalle lettere **A, B, C, D**
- I blocchi A e B appartengono all'**insieme 0 (G0) – Blocchi Pari**
- I blocchi C e D appartengono all'**insieme 1 (G1) – Blocchi Dispari**
- Numero gruppi = num_blocchi_cache / num_vie = 4/2 = 2
- La politica di sostituzione adottata nella cache è di tipo **LRU (Last Recently Used)**, quindi, visto che sono a 2 vie, è necessario un bit per indicare se il blocco è in uso o meno.

L'indirizzo di memoria centrale è ancora a 12 bit e viene diviso in questo modo

Spiazzamento: 128 byte \rightarrow **7 bit** dell'indirizzo.

Indice di gruppo: 2 gruppi \rightarrow **1 bit**

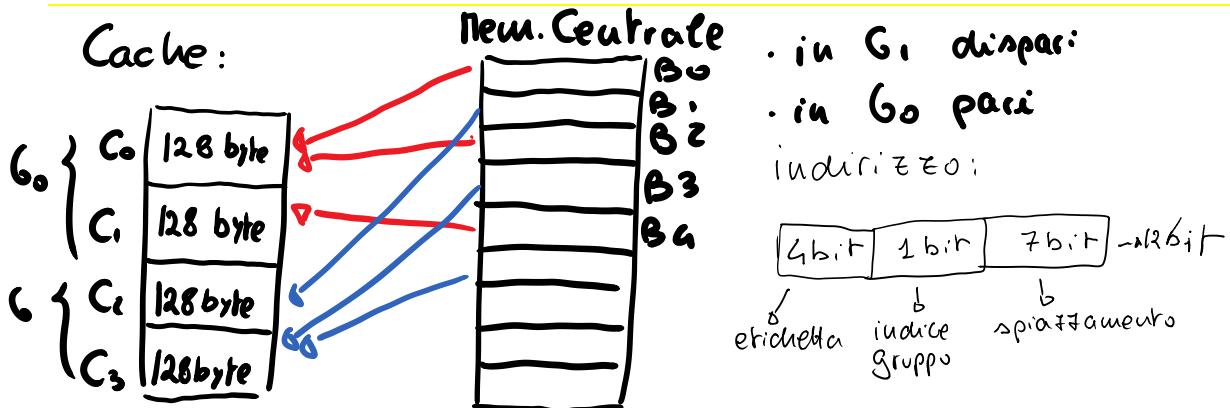
Etichetta: **4bit** \rightarrow perché, in un blocco all'interno di un gruppo in cache, posso mettere fino a 16 blocchi diversi della memoria centrale

Esempio:

Indirizzo 1: 0101|1010 0100

- 0101 \rightarrow etichetta
- 1 \rightarrow indice di gruppo
- 010 0100 \rightarrow spiazzamento in byte all'interno del gruppo

Passo	Indirizzo richiesto	Esito	Blocco A			Blocco B			Blocco C			Blocco D			Azione
			Valido	Etichetta	Dati										
0			1	1000	16	0	-	-	1	0001	3	0	-	-	Situazione iniziale
1.	0101 1010 0100	M	1	1000	16	0	-	-	1	0001	3	1	0101	11	Carica blocco 11 in D
2.	0001 1010 1000	H	1	1000	16	0	-	-	1	0001	3	1	0101	11	Accesso a C
3.	0001 0111 1110	M	1	1000	16	1	0001	2	1	0001	3	1	0101	11	Carica blocco 2 in B
4.	0000 0001 0100	M	1	0000	0	1	0001	2	1	0001	3	1	0101	11	Carica blocco 0 in A
5.	1001 1010 0100	M	1	0000	0	1	0001	2	1	0001	3	1	1001	19	Carica blocco 19 in D
6.	0111 1000 1100	M	1	0000	0	1	0001	2	1	0111	15	1	1001	19	Carica blocco 15 in C



Esercizio n° 3 da tema d'esame 7/09/2016:

Cache *set-associative* a 4 vie:

- Blocchi da 64 Byte (o 16 parole da 32 bit)
- La cache ha 32 blocchi
- $32 / 4 = 8$ gruppi
- Siamo in MIPS: indirizzo in memoria centrale: 32 bit

L'indirizzo è analizzato nel seguente modo:

- Blocco da 64 Byte -> 6 bit per lo spiazzamento in Byte;
Si potrebbe anche dividere questi 6 bit in 4 + 2 -> 4 per la parola e 2 per il singolo byte.
Questa ulteriore divisione (accessoria) non provoca nessun cambiamento al sistema di indirizzamento.
- 8 gruppi -> 3 bit per l'indice del gruppo
- I restanti 24 bit per l'etichetta

Si immagini che MIPS **esegue in sequenza** l'accesso a tutti gli indirizzi da 0x ABCD 0000 a 0x ABCD 08BF estremi inclusi (lettura oppure scrittura di **UN SINGOLO BYTE** a ogni accesso).

Quanti sono gli accessi ai byte?

0x ABCD 08BF -

0x ABCD 0000

0x 0000 08BF +1 (per estremi inclusi) -> 0x000 08C0 = **2240**

A quante parole si accede?

Visto che una parola è formata da 6 bit: $2240 / 4 = \textbf{560 parole}$

Quanti blocchi corrispondono a questa sequenza di accessi ai byte?

Posso fare il calcolo in byte o in parole: byte -> blocchi da 64 byte -> $2240 / 64 = \textbf{35}$

parole ->blocchi da 16 parole da 32 bit -> $560 / 16 = \textbf{35}$

Si supponga **che all'inizio la cache sia vuota**, e che la sequenza di accessi venga **eseguita due volte di seguito**. Mostrare come i blocchi si inseriscano nella cache, riempiendo con i simboli dei blocchi 0, 1, 2 ... le caselle della tabella sottostante. (I BLOCCHI SONO NUMERATI IN ESADECIMALE).

Se in una casella un blocco ne sostituisce un altro, **lasciarli** indicati entrambi, **cancellando** con un tratto di penna quello sostituito (**politica: LRU**; se un gruppo ha più blocchi vuoti, li si riempia nell'ordine A, B, C e D).

	A (C0)	B (C1)	C (C2)	D (C3)
000 (G0)	0 20 18	8 0 20	10 8	18 10
001 (G1)	1 21 19	9 1 21	11 9	19 11
010 (G2)	2 22 1A	A 2 22	12 A	1A 12
011 (G3)	3	B	13	1B
100 (G4)	4	C	14	1C
101 (G5)	5	D	15	1D
110 (G6)	6	E	16	1E
111 (G7)	7	F	17	1F

All'inizio devo caricare i primi 4 blocchi della memoria (partendo da 0x ABCD 0000) negli altrettanti blocchi dei gruppi della cache. Per il primo ciclo di riempimento della cache la procedura è immediata.

Arrivo a sostituire i primi 3 blocchi caricati a causa della politica LRU. Per il secondo ciclo di lettura tengo la

politica LRU e sostituisco i Last Used coi dati che mi servono, quando trovo un dato che è presente (perché non sostituito in precedenza) allora passo al dato successivo, ho un HIT. Ricorda che ogni nuovo caricamento che devo fare nella cache corrisponde a un MISS.

Quanti sono i MISS?

Quanti sono gli accessi alla memoria?

Quanto vale il MISS RATE?

Quanto vale lo HIT RATE?