

Architettura degli elaboratori

Simone Acuti

June 2023

Contents

1	Implementazione Algoritmo	6
1.1	Algoritmi	6
1.2	Implementazione Software	6
1.3	Implementazione Hardware	6
2	ISA e Micro-architettura	8
2.1	Livelli di trasformazione	8
3	Macchina di Von Neumann	9
3.1	Modello di Von Neumann	9
3.2	Memoria	9
3.2.1	Memoria word-addressable	10
3.2.2	Memoria byte-addressable	10
3.2.3	Endianess	11
3.2.4	Accessi in memoria	11
3.3	Unità di Elaborazione	12
3.4	Unità di controllo	12
3.5	Program visible state	12
3.6	Memoria dati e istruzioni	13
3.7	Proprietà fondamentali del modello	13
4	Istruzioni per il trasferimento dati dalla/verso la memoria	14
4.1	I registri	14
4.2	Lettura in memoria	14
4.3	Scrittura in memoria	15

5 Linguaggio Macchina	16
5.1 Dal linguaggio assembler al linguaggio macchina	16
5.2 Formati delle istruzioni	16
5.3 Utilizzo di costanti a 32 bits	17
6 Salti Incondizionati e Condizionati	18
6.1 Salti condizionati	18
6.2 Salti incondizionati	18
6.3 Istruzione jump	19
6.4 Soluzione estrema jump	19
6.5 Soluzione estrema branch	19
7 Chiamate a Procedura	20
7.1 Registri	20
7.2 Problemi	20
7.3 Lo stack	21
7.4 Complicazioni	21
7.4.1 Problema procedure innestate	22
7.4.2 Problema dello stack pointer	23
7.4.3 Problema degli argomenti	23
8 Rappresentazione dei numeri nei Calcolatori	24
8.1 Complemento a 2	24
8.2 Problemi di overflow	24
8.3 Come riconoscere gli overflow	25
9 Allineamento dei dati in memoria	26
9.1 Vincoli di allineamento	26
9.2 Regole di allineamento	26
9.3 Overhead	26
10 Sintesi di logica Combinatoria	27
10.1 Segnale analogico	27
10.2 Segnale digitale	27
10.3 Conversione analogico-digitale	27
10.4 Problema della risoluzione	27
10.5 Operatori logici e porte logiche	27
10.6 Tabelle	28
10.7 Logica combinatoria	29
10.8 Ciclo di progettazione ingegneristico	30
10.9 Princípio fondamentale	30

10.10 Definizioni	31
10.11 Esempio	31
11 Semplificazioni di Funzioni Logiche	32
11.1 Problema della semplificazione	32
11.2 Semplificazione	32
11.3 Algebra di Boole	32
11.4 Proprietà dell'algebra di Boole	33
12 Sintesi di Logica Combinatoria	34
12.1 Forma minima	34
12.2 Definizioni	34
12.3 Mappe di Karnaugh	34
12.4 Procedimento di semplificazione logica	35
12.5 Indifferenze	35
13 Componenti Combinatori	36
13.1 Decoder	36
13.2 Multiplexer	36
13.3 Differenza decoder e multiplexer	37
13.4 Circuito half-adder	37
13.5 Full adder	38
13.6 Adder a 32 bits	38
13.7 Ripple-carry adder	39
14 ALU	39
14.1 Realizzazione della sottrazione	39
14.2 ALU	40
14.3 Struttura	40
14.4 ALU-funzioni logiche	41
14.5 Funzioni aritmetiche	42
14.6 ALU ad n bitss	42
14.7 Bit di flag	43
15 Macchina a Stati Finiti	44
15.1 Logica sequenziale	44
15.2 Operatore sequenziale elementare	44
15.3 Sintesi di funzioni sequenziali	45
15.4 Clock	46
15.5 Specifica macchina a stati	46
15.6 Esempio: vending machine	46

15.7 State transition table	47
15.8 State transition diagram	47
15.9 Diagrammi ASM	48
15.10 Sintesi hardware di una rete sequenziale	48
16 Microarchitettura	49
16.1 Vista astratta di un microprocessore MIPS	49
16.2 Vista più dettagliata	50
16.3 Costruzione del datapath	51
16.3.1 Implementazione di tipo R	51
16.3.2 Implementazione LOAD/STORE	52
16.3.3 Implementazione di istruzioni di tipo R e LOAD/STORE	52
16.3.4 Implementazione di salti condizionati	52
16.3.5 Datapath completo	54
16.4 Control-path	55
16.5 Implementazione istruzione JUMP	56
17 Pipelining	57
17.1 Stadi di pipeline	57
17.1.1 Instruction fetch	57
17.1.2 Instruction decode	58
17.1.3 Execution	58
17.1.4 Memory access	59
17.1.5 Write back	59
17.2 Hazard	60
17.2.1 Hazard strutturali	60
17.2.2 Hazard dati	60
17.2.3 Hazard di controllo	63
17.3 Implementazione del pipelining nell'architettura MIPS	65
17.4 Main controller	66
18 Forwarding unit e Hazard detection unit	68
18.1 Dettaglio micro-architetturale	69
18.2 Stalli della pipeline	69
18.3 Dettaglio micro-architetturale	70
19 Branch prediction statica	71
19.1 Branch-hazard	71
19.2 Branch prediction	72
19.2.1 Missprediction penalty	72
19.2.2 Flush della pipeline	72

19.3	Predizione branch target address	74
19.3.1	Stadio di fetch con branch target buffer	75
19.4	Branch prediction statica	76
19.4.1	Always UNTAKEN	76
19.4.2	Always TAKEN	76
19.4.3	BTFN	77
19.4.4	Predizione su profiling del codice	77
19.4.5	Predizione basata sul programma	77
19.4.6	Predizione basata sul programmatore	77
20	Branch prediction dinamica	78
20.1	Last time predictor	78
20.1.1	Branch prediction buffer	79
20.1.2	Problema del last time predictor	79
20.2	Predittore a 2 bits	80
20.2.1	Limiti della predizione dinamica a 2 bits	81
20.3	Global Two-level prediction	82
20.3.1	Implementazione	82
20.3.2	Miglioramento dei predittori globali	83
20.4	Local Two-level prediction	83
20.4.1	Implementazione	84
20.4.2	Problemi della local prediction	84
20.5	Hybrid branch predictors	84
21	Cache	85
21.1	Gerarchia di memoria	85
21.2	Funzionamento della gerarchia hit e miss	86
21.3	L1 cache	87
21.4	Indirizzamento della cache	87
21.5	Cache direct mapped	87
21.5.1	Implementazione direct-mapped cache	88
21.6	Tipi di cache misses	89
21.7	Set associative cache	90
21.7.1	Implementazione set associative cache a 2 vie	90
21.8	Fully associative cache	90
21.8.1	Implementazione fully associative cache	91
21.9	Scrittura in cache	91
21.9.1	Opzioni write hit	92
21.9.2	Opzioni write miss	93

1 Implementazione Algoritmo

1.1 Algoritmi

Un *algoritmo* è uno strumento per risolvere problemi. Ha 3 proprietà fondamentali:

- È definito con precisione.
- È computabile.
- Termina.

Il metodo algoritmico esprime la capacità della ragione umana di trovare soluzioni computabili (accelerabili da un calcolatore) a determinate categorie di problemi.

1.2 Implementazione Software

Per implementare un’algoritmo sul software occorre codificarlo mediante un linguaggio di programmazione su un microprocessore.

Pro:

- È facilmente rirprogrammabile.
- Il programmatore può permettersi di non capire niente di assembler e di hardware.

Contro:

- Ricompilare milioni di righe di codice è dispendioso.
- Per ottimizzare certe prestazioni (per esempio il consumo di batteria) occorre avere delle conoscenze di hardware e/o assembler.

1.3 Implementazione Hardware

Si traduce l’algoritmo nel modello di un hardware che lo implementi. Ci sono due modelli di HW fondamentali: reti logiche combinatorie e reti logiche sequenziali.

Logica combinatoria: ad ogni combinazione del valore dei segnali di ingresso corrisponde uno ed un solo valore delle uscite.

Logica sequenziale: le sue uscite all’istante di tempo t sono definite non solo dal valore degli ingressi all’istante t ma anche dal valore degli ingressi agli istanti precedenti.

Per passare dal modello hardware in una implementazione hardware funzionante vi sono 2 soluzioni:

- **FPGA**: mediante un processo di compilazione, il modello hardware viene tradotto in una sequenza di bit di comando per l’hardware riconfigurabile.

Pro:

- È riconfigurabile.

Contro:

- È lento.
- Sono richieste due figure professionali: il programmatore software e il programmatore hardware.

- **ASIC**: Mediante un processo di sintesi, il modello hardware viene tradotto in un circuito elettronico dedicato che realizza esclusivamente l’algoritmo desiderato.

Pro:

- Implementazione algoritmica, quindi la più ottimizzata.
- il programmatore software si interfacci con un progettista hardware.

Contro:

- Costosa.
- Non riconfigurabile.

2 ISA e Micro-architettura

2.1 Livelli di trasformazione

Livello di astrazione	Esempi
Problema	Salvare un file
Algoritmo	JPEG
Software Applicativo	Programmi
Runtime System	Sistemi operativi
Architettura del set di istruzioni (ISA)	Fai somma
Micro-architettura	Memoria
Operatori Logici	AND gate, OR gate
Circuiti	Reti logiche
Dispositivi	Transistor, diodi
Fisica	Elettroni

Table 1: Livelli di astrazione

Il programmatore vede fino all'ISA, il resto viene astratto.

ISA: Interfaccia tra hardware e software; è un contratto che l'hardware promette di rispettare. È ciò che il programmatore deve sapere per scrivere e debuggare codice. È la concettualizzazione della macchina sottostante.

Micro-architettura: L'implementazione dell'ISA è la micro-architettura. Non è direttamente visibile al programmatore. Può essere radicalmente diversa da ciò che vede l'ISA, per ottimizzare performance o consumo.

3 Macchina di Von Neumann

3.1 Modello di Von Neumann

Costruiamo un *computer*. Per costruire un computer abbiamo bisogno di un modello; John Von Neumann propose un modello composto da cinque parti:

- Memoria
- Unità di elaborazione (Processing Unit)
- Input
- Output
- Unità di controllo (Control unit)

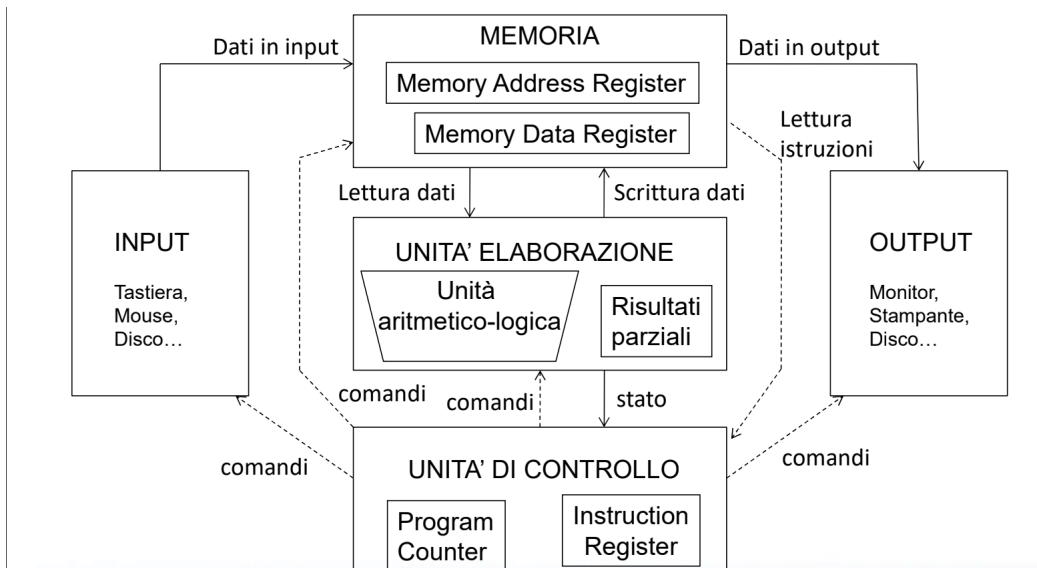


Figure 1: Modello di Von Nuemann

3.2 Memoria

La memoria memorizza: dati e programmi. L'unità di memorizzazione si chiama bit (unità di informazione); i bits sono raggruppati in bytes (8 bits) e words (4 bytes). Il modo in cui si accede ai bits determina la "indirizzabilità" (Es: processore *word addressable*). Il numero totale degli indirizzi si chiama spazio di indirizzamento, il MIPS questo spazio è di 2^{32} bits. Per rappresentare gli indirizzi usiamo un sistema in base 16.

3.2.1 Memoria word-addressable

Ogni "data word" ha un indirizzo univoco, in mips c'è un indirizzo univoco per 32 bit data word.

Word Address	Data	MIPS memory
.	.	.
.	.	.
.	.	.
00000003	D 1 6 1 7 A 1 C	Word 3
00000002	1 3 C 8 1 7 5 5	Word 2
00000001	F 2 F 1 F 0 F 7	Word 1
00000000	8 9 A B C D E F	Word 0

Figure 2: Memoria word addressable

3.2.2 Memoria byte-addressable

Ogni byte ha un indirizzo univoco (mips è byte addressable).

Byte Address of the Word	Data	MIPS memory
.	.	.
.	.	.
.	.	.
0000000C	D 1 6 1 7 A 1 C	Word 3
00000008	1 3 C 8 1 7 5 5	Word 2
00000004	F 2 F 1 F 0 F 7	Word 1
00000000	8 9 A B C D E F	Word 0

Figure 3: Memoria byte addressable

3.2.3 Endianess

L'endianess è l'ordine in cui memorizziamo i dati in memoria. Esistono due tipi di endianess:

- *Big Endian*: memorizzazione/trasmissione che inizia dal byte più significativo (quello di sinistra) per finire col meno significativo.
- *Little Endian*: memorizzazione/trasmissione che inizia dal byte meno significativo (quello di destra) per finire col più significativo.

Big Endian vs. Little Endian

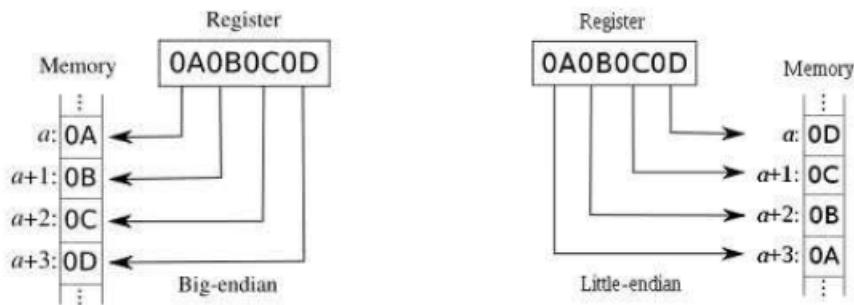


Figure 4: Big endian e Little endian

3.2.4 Accessi in memoria

Ci sono due modi per accedere alla memoria:

- Lettura (*loading*).
- Scrittura (*storing*).

Sono necessari due registri per accedere alla memoria: il MAR ("Memory Address Register") e il MDR ("Memory Data Register").

Per leggere:

- Step 1: caricare l'indirizzo nel MAR.
- Step 2: il dato è piazzato nel MDR.

Per Scrivere:

- Step 1: caricare l'indirizzo nel MAR e il dato sul MDR.
- Step 2: attiva il segnale di write enable.

3.3 Unità di Elaborazione

L'unità di elaborazione è composta di diverse unità funzionali.

L'unità funzionale più importante è la **ALU**, che esegue operazioni aritmetiche e logiche. La ALU elabora quantità riferite come *words*, in MIPS una word sono 32 bits \Rightarrow 4 bytes \Rightarrow 1 word.

La **memoria locale** contiene gli operandi e i risultati della ALU. Essa è composta dai registri.

3.4 Unità di controllo

L'unità di controllo si può paragonare al direttore d'orchestra. Esso dirige il processo di esecuzione di un programma passo dopo passo (istruzione dopo istruzione).

Ha bisogno di un registro che contiene l'indirizzo in memoria della istruzione attualmente in esecuzione: program counter (PC).

Deve tenere "memoria locale" dell'istruzione attualmente in esecuzione tramite un instruction register (IR).

3.5 Program visible state

Ogni volta che il programmatore scrive e compila un programma, lui vede il seguente stato della macchina:

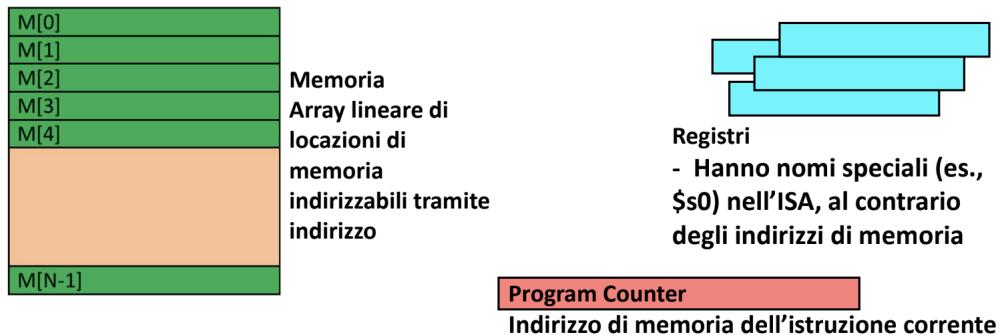


Figure 5: Program visible state

Molte cose sono nascoste al programmatore, come l'IR e tutta la micro-architettura.

3.6 Memoria dati e istruzioni

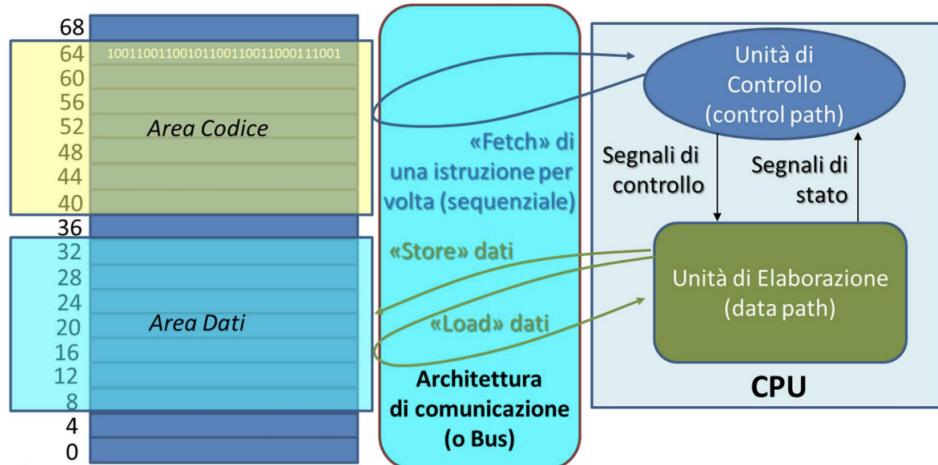


Figure 6: Memoria dati e istruzioni

L’unità di controllo legge un’istruzione per volta, e la manda all’unità di elaborazione, di modo che la possa eseguire. L’unità di elaborazione può (a seconda dell’istruzione) scrivere o leggere in memoria (load e store di dati).

3.7 Proprietà fondamentali del modello

Due proprietà:

- **Stored program:** Le istruzioni sono memorizzate in un array di dati di memoria lineare. La memoria dati e la memoria istruzioni sono unificate, quindi le istruzioni sono memorizzate come i dati.
- **Sequential instruction processing:** Le istruzioni vengono processate una alla volta nell’ordine del programma.

4 Istruzioni per il trasferimento dati dalla/verso la memoria

4.1 I registri

Un registro è un'unità di memoria scrivibile e leggibile in modo molto rapido. Il numero dei bits che compongono un registro può variare (es: 32 bits o 64 bits). Varia anche il numero di registri (es: 32 per l'architettura MIPS).

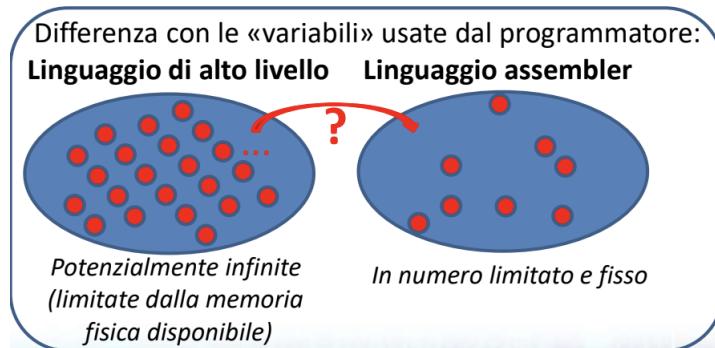


Figure 7: Registri

Pochi registri permettono di raggiungere frequenze di clock più elevate, e di minimizzare il numero di bit che servono per codificare le istruzioni.

I registri sono un piccolo pezzo di memoria, per di più vicino al microprocessore; dunque l'accesso ai registri avviene più rapidamente rispetto all'accesso alla memoria di massa per due motivi:

- Memorie più grandi sono anche più lente.
- La latenza di accesso è inferiore grazie alla vicinanza.

In MIPS i registri \$s0, \$s1, ..., \$s7 sono utilizzati per memorizzare le variabili ad alto livello del programma. I registri \$t0, \$t1, ..., \$t9 sono usati per valori temporanei.

4.2 Lettura in memoria

Assumiamo un'architettura **word-addressable** e un array immagazzinato in memoria dal compilatore all'indirizzo base $x+1$. Per effettuare una lettura in memoria si utilizza l'istruzione load-word (lw).

Istruzione C:

$$f = array[2];$$

Siccome il nostro indirizzo base è $x+1$ per andare a leggere $array[2]$, devo spostarmi all'indirizzo $x+3$, che nell'istruzione MIPS avremo un offset di 2.

Istruzione MIPS:

$$lw \$t0, 2(\$s3)$$

MIPS è però un architettura **byte-addressable**. Quindi la vera istruzione sarà:

$$lw \$t0, 8(\$s3)$$

4.3 Scrittura in memoria

È possibile copiare dati da un registro ad una specifica locazione di memoria mediante l'istruzione di store-word (sw).

Istruzione C:

$$array[2] = n;$$

Quindi la macchina deve copiare il valore di un registro (variabile n) dentro una locazione di memoria ($array[2]$).

Istruzione MIPS:

$$sw \$s0, 8(\$s3)$$

5 Linguaggio Macchina

5.1 Dal linguaggio assembler al linguaggio macchina

Per passare dal linguaggio assembler al linguaggio macchina occorre dare diversi formati alle istruzioni. Ogni formato di istruzione è formato da *fields* (gruppi di bits contigui) che assumono significato diverso a seconda dell'istruzione.

Esempio: add \$t0, \$s1, \$s2



Figure 8: Formato dell'istruzione somma

NB: tutte le istruzioni in MIPS sono lunghe 32 bits, cioè una word.

5.2 Formati delle istruzioni

Siccome ogni istruzione ha esigenze diverse (es: sommare una costante a 16 bits), dobbiamo definire formati diversi a seconda dello scopo dell'istruzione.

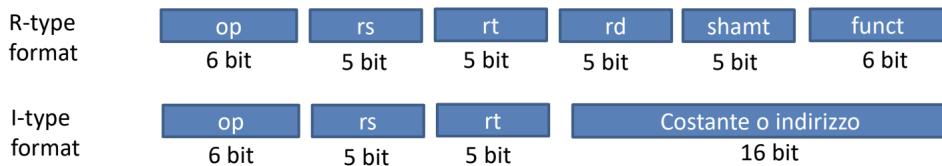


Figure 9: Formati R e I

- **OP**: OPCODE, è l'operazione di base.
- **RS**: primo registro operando.
- **RT**: secondo registro operando.
- **RD**: registro di destinazione.
- **SHAMT**: ammontare dello shift.
- **FUNC**: FUNCTION CODE, specifica la variante di OP.

I-Type: per istruzioni di tipo immediate o per il trasferimento di dati (load e store). L'istruzione di tipo addi può sommare una costante da -2^{15} a $2^{15} - 1$.

5.3 Utilizzo di costanti a 32 bits

Ipotizziamo di voler eseguire quest'istruzione: addi \$t0, \$t0, 11232769.

Il numero 11.232.769 non è rappresentabile con 16 bits, ma con 24 si. Il campo costante nell'istruzione I ha però a disposizione 16 bits.

Come fa il compilatore? È costretto ad inserire il numero in un registro a sé.

1. Rappresentiamo la nostra costante in esadecimale: 11232769 \Rightarrow 0xAB6601.
2. Carichiamo 0x00AB nella parte alta del registro (\$at): lui \$1, 0x00AB.
3. Carichiamo 0x6601 nella parte bassa del registro: ori \$1, \$1, 0x6601.
4. Infine eseguiamo una somma normale (R): add \$2, \$2, \$1.

6 Salti Incondizionati e Condizionati

6.1 Salti condizionati

MIPS ha 2 istruzioni per il salto condizionale (if-else):

- Branch-if-equal: beq register1, register2, label
- Branch-if-NOT-equal: bne register1, register2, label

Altre operazioni tipo branch-if-less-than (blt), non esistono in hardware perché sono troppo complicate da realizzare. Si preferisce spezzare l'operazione in più istruzioni, ma preservando la velocità di funzionamento.

Tutti i tipi di operazioni (compresi less-than-or-equal, greater-than, greater-than-or-equal) vengono realizzate mediante combinazioni delle operazioni di base slt, slti, beq, bne.

Siccome i salti condizionati sono delle istruzioni di tipo I, quest'ultimi hanno a disposizione solo 16 bits per l'indirizzo a cui devono saltare.

Significherebbe che un programma dovrebbe avere lunghezza massima di 2^{16} bits ($\simeq 64k$ istruzioni).

La soluzione è stata quella di cambiare addressing mode, trattando il campo finale non come indirizzo, ma come offset con segno.

Come punto a cui dare l'offset usiamo il PC (program counter)

6.2 Salti incondizionati

Le istruzioni MIPS per il salto incondizionato (jump), utilizzando il meccanismo dell'indirizzamento offerto da un nuovo tipo di istruzione: la **J-Type**.

OP	Address
6 bits	26 bits

Table 2: Istruzione di tipo J

IMPORTANTE: i salti incondizionati non usano l'offset come nei salti condizionati. Queste istruzioni codificano un *indirizzo assoluto*.

6.3 Istruzione jump

Il campo address è un indirizzo assoluto, cioè con la gradualità della word (1 istruzione MIPS = 1 word), non del byte. Per ottenere l'indirizzo completo a 32 bits occorre fare questi passaggi:

1. Moltiplicare il campo address per 4, cioè fare lo shift-left di 2 posizioni, inserendo 00 in posizione meno significativa (a sinistra).
2. I restanti 4 bits provengono dai bits più significativi del PC.

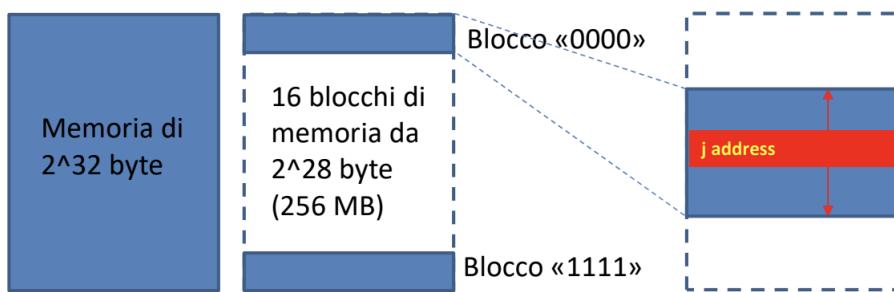


Figure 10: Blocco della jump

6.4 Soluzione estrema jump

Se serve andare oltre all'indirizzamento di 64M di istruzioni bisogna:

1. Formare l'indirizzo completo a 32 bits a cui saltare.
2. Memorizzazione dell'indirizzo in un registro (es: \$t0).
3. jr \$t0 (jump register).

6.5 Soluzione estrema branch

Se i 16 bit non sono sufficienti per creare il branch address il compilatore trasforma il salto condizionato in incondizionato, beneficiando così di un branch address di 26 bits.

7 Chiamate a Procedura

I programmatori di alto livello ricorrono spesso a procedure o funzioni per la miglior comprensibilità del programma, e per facilitare il suo riutilizzo.

Un **principio importante**: non deve rimanere traccia dell'esecuzione di una procedura nello stato del sistema, che deve tornare quello precedente l'invocazione della procedura.

Unica perturbazione: i valori di ritorno della procedura sono disponibili in locazioni pre-fissate.

7.1 Registri

- Passaggio argomenti: 4 registri da \$a0 a \$a3.
- Registri per valori di ritorno: \$v0, \$v1.
- Abbiamo un registro che punta all'indirizzo della prossima istruzione del chiamante: \$ra.

7.2 Problemi

La procedura potrebbe utilizzare dei registri che sono già in uso nel "main"; quindi è necessario fare lo spilling in memoria principale dei registri utilizzati nell'invocazione ed esecuzione della procedura.

Questa operazione manda i registri del processore alla memoria principale (PUSH) e viceversa (POP). Essa è come se creasse una copia di backup per il successivo ripristino.

Per facilitare PUSH e POP, una porzione della memoria viene identificata e gestita in modo particolare (STACK).

7.3 Lo stack

Una coda Last-In First-Out è la struttura di spilling ideale, chiamata STACK. Esso risiede in memoria principale.

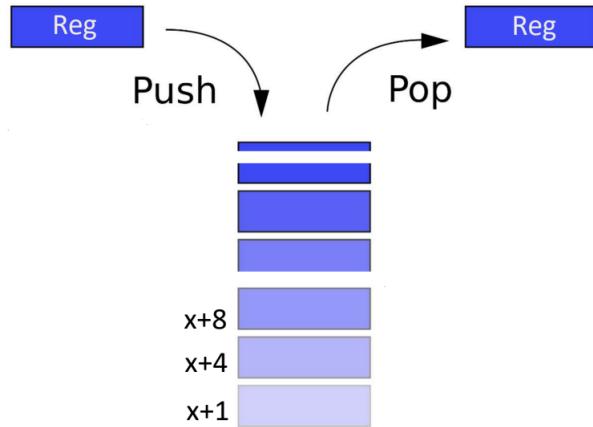


Figure 11: Struttura LI-FO

Per tenere traccia dei registri dobbiamo usare un puntatore, MIPS ci fornisce un registro apposito: \$sp (stack pointer).

7.4 Complicazioni

Il *frame* di una procedura sullo stack non serve solamente per salvare il valore dei registri da preservare, ma tipicamente anche per:

- Garantire la consistenza dell'esecuzione in caso di chiamate ricorsive a subroutines.
- Allocazione di variabili locali alla procedura.
- Passare un numero di argomenti > 4.

7.4.1 Problema procedure innestate

Problema: quando vogliamo chiamare una procedura dentro un'altra procedura i valori dentro certi registri si possono corrompere.

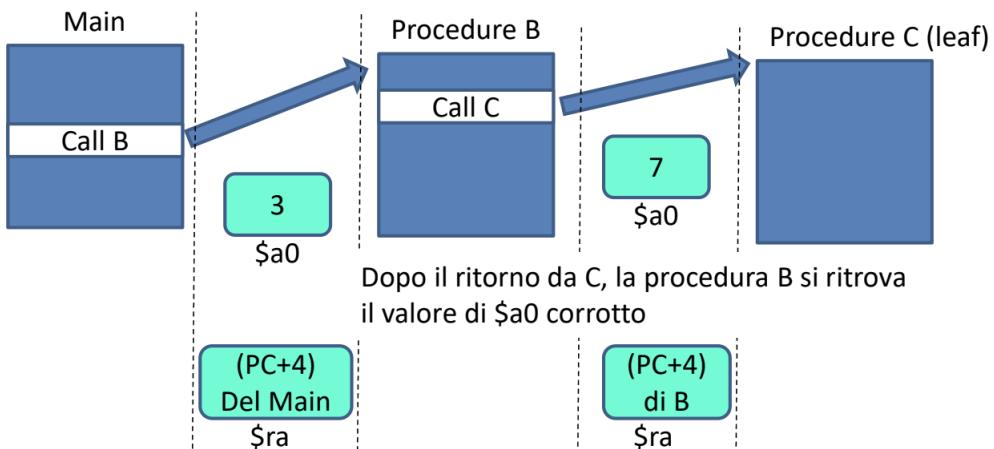


Figure 12: Problema procedure innestate

La **soluzione** è quella di fare il push anche dei registri \$ra e \$ai nello stack.

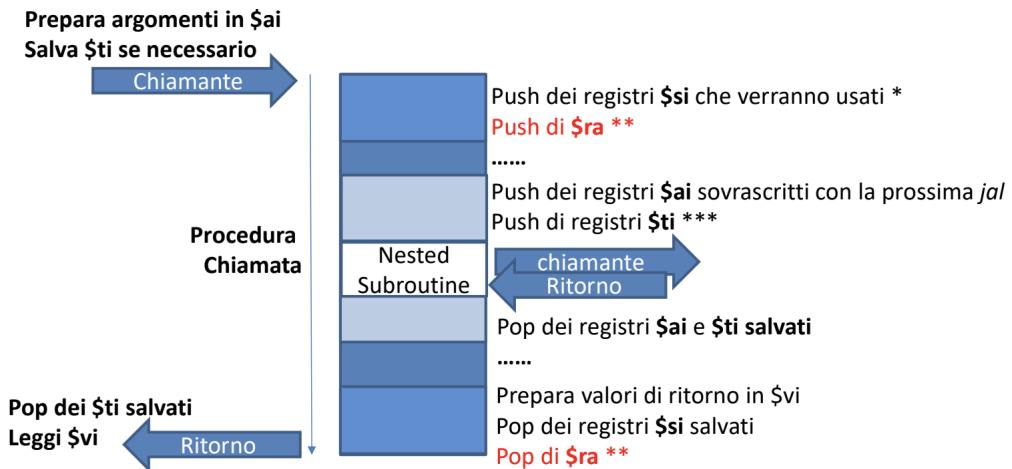


Figure 13: Problema procedure innestate

* Perché se ne deve garantire l'integrità al chiamante.

** SOLO SE la procedura chiamata è a sua volta chiamante.

*** Solo se nel caso io debba riusare il valore di alcuni registri \$ti dopo la chiamata.

7.4.2 Problema dello stack pointer

Le variabili locali ad una procedura, non gestibili direttamente sui registri del processore, sono salvate sullo stack.

Problema: durante l'esecuzione della procedura, \$sp potrebbe cambiare, dunque quelle variabili si trovano ad offset variabili in funzione della posizione istantanea dello Stack Pointer.

La **soluzione** è quella di utilizzare un registro chiamato *Frame-Pointer*, che punta stabilmente alla prima parola del frame.

- "Resizing" dello stack pointer.
- Push del vecchio \$fp sullo stack.
- add \$fp, \$sp, FRAMESIZE-4 # o simili.
- lw \$fp, xx(\$sp)
- "Resizing" dello stack pointer.

7.4.3 Problema degli argomenti

Di norma, sappiamo che in MIPS abbiamo a disposizione solamente 4 registri (\$a0, ..., \$a3), per passare gli argomenti. Ma se abbiamo bisogno di passarne più di 4?

La soluzione è quella di utilizzare il frame pointer.

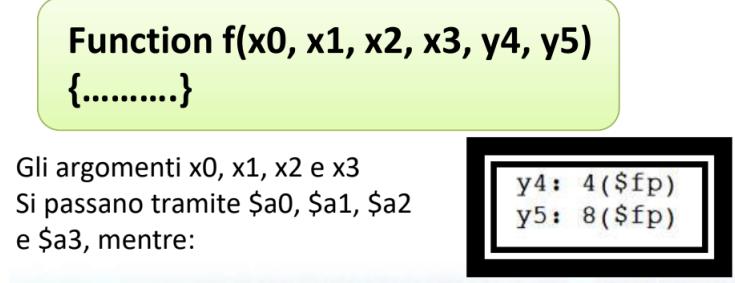


Figure 14: Passaggio argomenti con frame pointer

8 Rappresentazione dei numeri nei Calcolatori

Per rappresentare i numeri il computer usa il sistema binario. Anch'esso è un sistema posizionale, dove, ogni posizione rappresenta una potenza crescente del 2, a cominciare da 2^0 , e a crescere verso sinistra.

Esempi:

- $1111111_2 = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + \dots + 1 \times 2^0 = 255_{10}$
- $01111011_2 = 123_{10}$

8.1 Complemento a 2

Per rappresentare i numeri binari negativi si utilizza la cosiddetta: rappresentazione in complemento a 2. Questa rappresentazione è contraddistinta dal fatto che il suo ultimo peso è negativo. Dato un numero binario con

peso naturale	128	64	32	16	8	4	2	1
peso compl. a 2	-128	64	32	16	8	4	2	1

Figure 15: Complemento a 2

valore decimale X, si può ottenere la codifica binaria di (-X) mediante un procedimento in due passi:

- Inversione del valore di ogni bit (complemento a 1).
- Sommare +1.

8.2 Problemi di overflow

Spesso durante operazioni aritmetiche con segno possono esserci risultati anomali (es: $7+1 = -8$, in aritmetica con segno con 4 bits). Questi risultati sono detti overflow.

8.3 Come riconoscere gli overflow

L'hardware della ALU fa scattare un'eccezione hardware quando si accorge delle seguenti condizioni di overflow:

Operation	Operand A	Operand B	Result indicating overflow
$A + B$	≥ 0	≥ 0	< 0
$A + B$	< 0	< 0	≥ 0
$A - B$	≥ 0	< 0	< 0
$A - B$	< 0	≥ 0	≥ 0

Figure 16: Tabella degli overflow

ATTENZIONE: L'overflow NON viene normalmente segnalato per l'elaborazione di tipi di dato unsigned.

9 Allineamento dei dati in memoria

Gli accessi in memoria avvengono normalmente alla granularità delle word (32 bit), uguale alla dimensione dei registri, o di multipli di essa.

Perciò, indirizzi non allineati (se supportati dalla CPU) determinano una complicazione nell'accesso: sono generalmente necessari diversi accessi allineati per servire un accesso disallineato.

9.1 Vincoli di allineamento

Un dato di dimensione n byte deve essere memorizzato ad un indirizzo multiplo di N .

- Dati da 1 byte: sono memorizzabili ovunque.
- Dati da 2 bytes (1 halfword): memorizzabili ad indirizzi pari (ultimo bit: 0).
- Dati da 4 bytes (word): memorizzabili ad indirizzi multipli di 4 (ultimi due bits: 00).
- Dati da 8 bytes (double word): memorizzabili ad indirizzi multipli di 8 (ultimi tre bits: 000)

9.2 Regole di allineamento

1. Il Compilatore inserisce dei byte di padding (=inutilizzati) in modo che ogni tipo di dato sia allineato a multipli della sua grandezza.
2. Ulteriore padding può essere richiesto per far iniziare e finire la struttura ad indirizzi allineati alla dimensione del suo membro più grande.
3. Spesso si può minimizzare la frammentazione piazzando per primi i tipi di dato più grandi, successivamente si piazzano quelli di grandezza inferiore.

9.3 Overhead

Tipicamente nell'ISA ci sono istruzioni dedicate per ogni grandezza di dato (load-halfword, load-byte).

10 Sintesi di logica Combinatoria

10.1 Segnale analogico

Un segnale elettrico può assumere virtualmente infiniti valori in un certo intervallo.

Ad una variazione idealmente piccola nel valore della tensione corrisponde una differenza nell'informazione da esso rappresentata.

10.2 Segnale digitale

Un segnale digitale può assumere solo due valori: 1.5V e 0V / alto o basso / vero o falso.

La perdita di informazioni nel segnale digitale viene compensata dall'utilizzo di diversi segnali digitali per rappresentare una grandezza. Con n segnali digitali posso rappresentare 2^n valori.

10.3 Conversione analogico-digitale

Con 3 segnali digitali/bit, posso dividere un intervallo di tensione V_8 in 2^3 intervalli, a ognuno dei quali corrisponde una codifica binaria.

Il segnale analogico consente un'elaborazione più robusta del segnale analogico. Esso associa un valore della grandezza da rappresentare non ad una tensione ma ad un intervallo di tensioni.

La risoluzione di segnali digitali è potenzialmente infinita.

Codifica naturale	
v_8	111
v_7	110
v_6	101
v_5	100
v_4	011
v_3	010
v_2	001
v_1	000

10.4 Problema della risoluzione

Se provo a leggere un valore di tensione attraverso un multimetro, osservo che il valore misurato in diversi istanti di tempo è diverso. Questo succede perché i valori dei bits vengono corrotti dal "rumore".

10.5 Operatori logici e porte logiche

Ad un segnale digitalizzato può essere associata una variabile binaria. Queste variabili binarie possono essere elaborate tramite *funzioni logiche* o *booleane*, che utilizzano operatori logici.

La realizzazione fisica di un operatore logico prende il nome di porta logica.

Ci sono tre tipi di porte logiche fondamentali:

- AND
- OR
- NOT

10.6 Tabelle

Porta AND		
a	b	out
0	0	0
0	1	0
1	0	0
1	1	1

Table 3: Porta logica AND

Porta OR		
a	b	out
0	0	0
0	1	1
1	0	1
1	1	1

Table 4: Porta logica OR

Porta NAND		
a	b	out
0	0	1
0	1	1
1	0	1
1	1	0

Table 5: Porta logica NAND

Porta NOR		
a	b	out
0	0	1
0	1	0
1	0	0
1	1	0

Table 6: Porta logica NOR

Porta XOR		
a	b	out
0	0	0
0	1	1
1	0	1
1	1	0

Table 7: Porta logica XOR

Porta XNOR		
a	b	out
0	0	1
0	1	0
1	0	0
1	1	1

Table 8: Porta logica XNOR

Porta NOT	
a	out
0	1
1	0

Table 9: Porta logica NOT

10.7 Logica combinatoria

I circuiti digitali risolvono i problemi (cioè, eseguono algoritmi) applicando operatori logici a variabili di ingresso binarie/booliane. In questo modo, si applicano *funzioni booliane* a queste variabili. L'insieme delle funzioni booliane da origine alla cosiddetta logica combinatoria.

10.8 Ciclo di progettazione ingegneristico

1. Definizione delle specifiche.
2. Tabella di verità.
3. Funzioni o espressioni logiche.
4. Rete di porte logiche.

10.9 Principio fondamentale

Ogni funzione logica combinatoria può essere rappresentata mediante logica a due livelli: un livello di AND gates seguito da un livello di OR gates o viceversa.

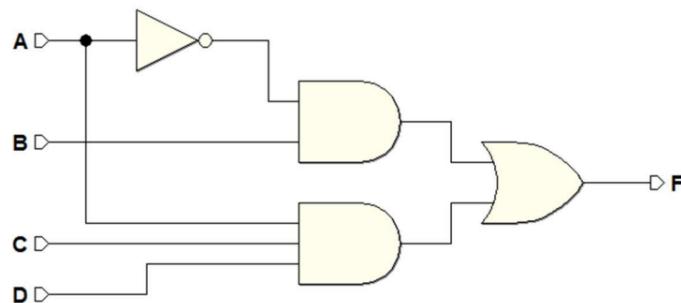


Figure 17: AND plane seguito da OR plane

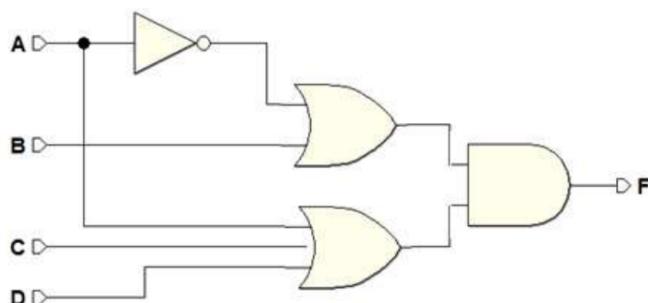


Figure 18: OR plane seguito da AND plane

Se abbiamo un AND plane seguito da un OR plane allora si parla di *somma di prodotti* (SOP). Se abbiamo un OR plane seguito da un AND plane allora si parla di *prodotto di somme* (POS).

10.10 Definizioni

- **MINTERM:** Data una funzione logica f di N variabili, un *minterm* di f è l'espressione di un prodotto logico in cui ciascuna delle N variabili d'ingresso di f compare una ed una sola volta, e può comparire diretta o negata.
- **MAXTERM:** Data una funzione logica f di N variabili, un *maxterm* di f è l'espressione di una somma logica in cui ciascuna variabile di ingresso di f compare una ed una sola volta, e può comparire diretta e negata.

10.11 Esempio

x	y	z	$f(x, y, z)$	OR's	AND's
0	0	0	0	$x + y + z$	
0	0	1	0	$x + y + \bar{z}$	
0	1	0	0	$\bar{x} + y + z$	
0	1	1	1		$\bar{x}yz$
1	0	0	0	$\bar{x} + y + z$	
1	0	1	1		$\bar{x}y\bar{z}$
1	1	0	0	$x + y + z$	
1	1	1	0	$\bar{x} + \bar{y} + \bar{z}$	

Figure 19: Espressioni SOP e POS

POS:

$$f = (x + y + z) \cdot (x + y + (\neg z)) \cdot (x + (\neg y) + z) \cdot ((\neg x) + y + z) \cdot ((\neg x) + (\neg y) + z) \cdot ((\neg x) + (\neg y) + (\neg z))$$

SOP:

$$f = (\neg x)yz + x(\neg y)z$$

11 Semplificazioni di Funzioni Logiche

11.1 Problema della semplificazione

Purtroppo la forma canonica non è quasi mai una forma "di costo minimo". Per ora, intendiamo come "costo" il numero di porte logiche utilizzate.

Per esempio, si può dimostrare che l'espressione booleana: $ab(\neg c) + abc$ è del tutto equivalente all'espressione semplificata: ab .

11.2 Semplificazione

Dal momento che le espressioni booleane rappresentano logica che il progettista vorrà implementare, è importante ottenere la rappresentazione booleana più "semplice".

Purtroppo, anche le espressioni SOP e POS potrebbero non essere le forme di rappresentazione più semplici di una funzione booleana.

È tuttavia possibile ridurre una espressione booleana alla sua forma più semplice utilizzando algoritmi di ottimizzazione e teoremi.

Questi algoritmi e teoremi si basano su postulati e proprietà che vanno complessivamente sotto il nome di *algebra di Boole*.

11.3 Algebra di Boole

L'algebra di Boole fornisce tutti gli strumenti per analizzare e sintetizzare le reti logiche. Essa comprende:

- Due elementi: 0 ed 1.
- Due operatori binari: AND e OR.
- Un operatore unario: NOT.

11.4 Proprietà dell'algebra di Boole

- $a + b = b + a$ (**Commutativa**). (1)
- $a + (b + c) = (a + b) + c$ (**Associativa rispetto alla somma**). (2)
- $a \cdot (b \cdot c) = (a \cdot b) \cdot c$ (**Associativa rispetto al prodotto**). (3)
- $a + (a \cdot b) = a$ (**Assorbimento rispetto alla somma**). (4)
- $a \cdot (a + b) = a$ (**Assorbimento rispetto al prodotto**). (5)
- $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ (**Distributiva rispetto al prodotto**). (6)
- $a + (b \cdot c) = (a + b) \cdot (a + c)$ (**Distributiva rispetto alla somma**). (7)
- $a + a = a$ (**Idempotenza rispetto alla somma**). (8)
- $a \cdot a = a$ (**Idempotenza rispetto al prodotto**). (9)
- $(a \cdot b) + (a \cdot (\neg b)) = a$ (**Combinazione rispetto alla somma**). (10)
- $(a + b) \cdot (a + (\neg b)) = a$ (**Combinazione rispetto al prodotto**). (11)
- $\neg(a \cdot b) = (\neg a) + (\neg b)$ (**De Morgan**). (12)
- $\neg(a + b) = (\neg a) \cdot (\neg b)$ (**De Morgan**). (13)
- $a \cdot 0 = 0$ (**\exists del minimo**). (14)
- $a + 1 = 1$ (**\exists del massimo**). (15)
- $a \cdot (\neg a) = 0$ (**\exists del complemento**). (16)
- $a + (\neg a) = 1$ (**\exists del complemento**). (17)
- $\neg(\neg a) = a$ (**Doppia negazione**). (18)

12 Sintesi di Logica Combinatoria

Per espressioni booleane più complesse, l'approccio "algebrico" basato sull'applicazione manuale delle proprietà e dell'algebra di Boole può essere molto complesso, se non impossibile.

In pratica si utilizzano tecniche algoritmiche di semplificazione, implementate mediante strumenti di CAD. (Es: le mappe di Karnaugh).

12.1 Forma minima

Lo scopo delle mappe di Karnaugh è di ricavare un'espressione SOP in forma minima, cioè: identificare una forma SOP che includa il numero minimo di termini prodotto e - a parità di numero di termini - i termini prodotto con il minimo numero di letterali.

12.2 Definizioni

- **Implicante:** data una funzione logica f , una espressione di prodotto logico si dice implicante di f se quando tale espressione vale 1 anche la f vale 1.
- **Implicante primo:** un implicante m di una funzione f si dice primo quando non esiste un altro implicante di f ove compaia sottoinsieme delle variabili di m .

12.3 Mappe di Karnaugh

Ai fini pratici, è utile per espressioni fino a 5 o 6 variabili, ed è un valido strumento che aiuta la comprensione del processo di semplificazione logica.

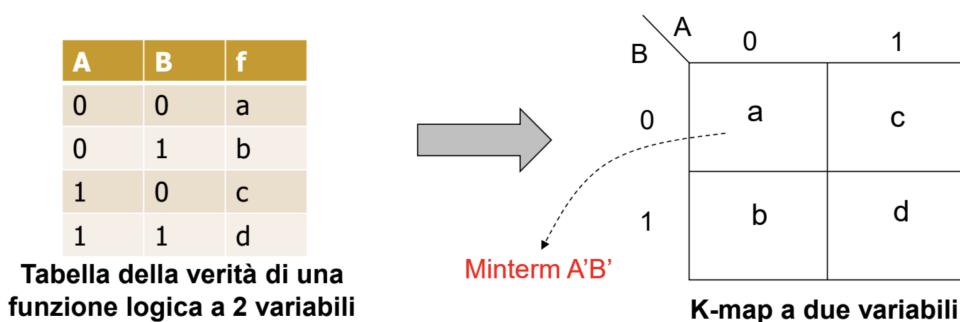


Figure 20: Esempio mappa di Karnaugh

12.4 Procedimento di semplificazione logica

- **Fase preparatoria:** formazione degli implicanti primi
 - Si rappresenta una funzione di N ingressi con una K-map.
 - Formazione degli implicanti primi. Si raccolgono i minterm implicanti (cioè, gli "1") in blocchi di dimensione 2^n caselle, partendo dalla massima dimensione e via via scendendo se non si trovano blocchi di quella dimensione.
 - Continua la procedura fino ad avere incluso tutti i minterm implicanti (cioè, gli "1" della mappa) in almeno un blocco.
- **Sintesi delle espressioni SOP di costo minimo**
 - Si scelgono tutti gli implicanti primi essenziali.
 - Si eliminano tutti gli implicanti primi che sono coperti da quelli essenziali.
 - Si seleziona il minor numero degli implicanti primi rimasti, al fine di coprire tutta la mappa.
 - Si scrive l'espressione OR di questi ultimi e di quelli essenziali.

12.5 Indifferenze

Spesso ci sono particolari configurazioni degli ingressi che non vengono mai utilizzate per come funziona il sistema. Per queste combinazioni, qualunque uscita va bene (si indica con "x" o "-").

La tecnica per trattare le indifferenze è dare un valore di comodo ai don't care, in modo da semplificare il più possibile la rete logica risultante.

13 Componenti Combinatori

Esistono funzioni logiche di particolare interesse ai fini delle applicazioni pratiche. Queste funzioni sono trattate, nelle reti logiche, come veri e propri componenti standard.

- Decoders.
- Encoders.
- Multiplexer.
- Half-adders.
- Full-adders.

13.1 Decoder

Un decoder è un circuito logico che ha n ingressi e 2^n uscite. Ogni uscita di un decoder sarà a 1 solo per 1 e una sola combinazione degli n ingressi.

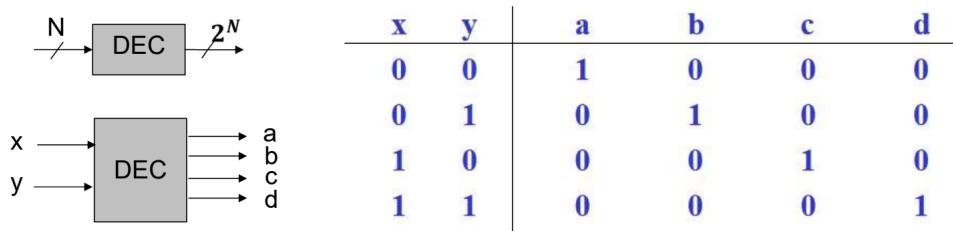


Figure 21: Esempio di decoder

Espressioni logiche

$$\begin{aligned}
 a &= (\neg x) \cdot (\neg y) \\
 b &= (\neg x) \cdot y \\
 c &= x \cdot (\neg y) \\
 d &= x \cdot y
 \end{aligned}$$

13.2 Multiplexer

Un multiplexer è un circuito logico combinatorio che ha 2^n ingressi, un indirizzo a n bits e un' uscita.

L'indirizzo a n bits viene codificato, di modo che il valore logico dell'ingresso selezionato venga portato in uscita.

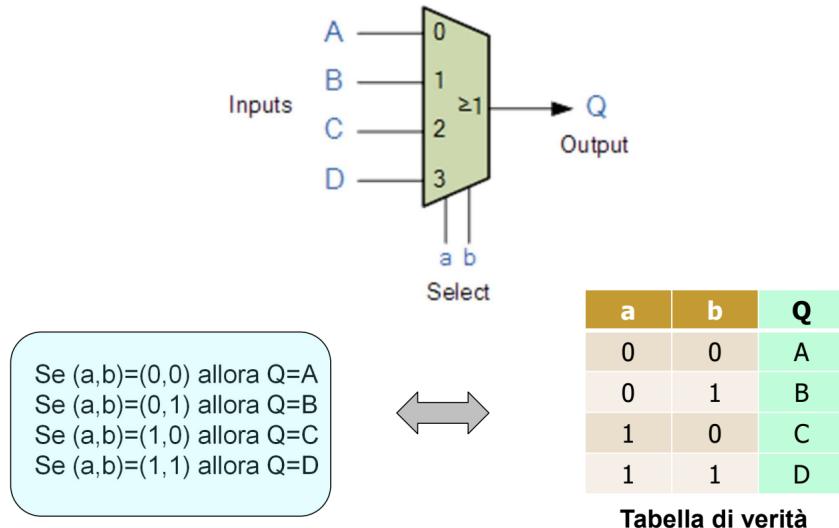


Figure 22: Esempio di multiplexer

13.3 Differenza decoder e multiplexer

Decoder:

- Un decoder ha n ingressi, chiamati "indirizzi".
- Un decoder ha 2^n uscite. Ogni uscita va a 1 solo per una specifica combinazione degli ingressi.

Multiplexer:

- Ha due set di ingressi: n bits di indirizzo, in aggiunta ha 2^n ingressi, uno dei quali è selezionato da ogni indirizzo per essere copiato sull'uscita.
- Il multiplexer ha solo un'uscita. L'uscita è il valore dell'ingresso selezionato dall'indirizzo.

13.4 Circuito half-adder

Poiché l'addizione viene fatta per colonne, ciò che serve per sommare due numeri ad n bits è l'utilizzo di un sommatore di colonna per ognuno degli n bits.

Il sommatore di colonna per la colonna più a destra si chiama half-adder (somma due bit senza alcun riporto in ingresso).

a	b	S	c_o	$S \text{ Exp}$	$c_o \text{ Exp}$
0	0	0	0		
0	1	1	0	\bar{ab}	
1	0	1	0	\bar{ab}	
1	1	0	1		ab

$S = \bar{ab} + \bar{ab}$

$C_o = ab$

Figure 23: Esempio di half adder

13.5 Full adder

L'Half-Adder non contempla un carry in, e in quanto tale serve solo come sommatore di colonna più a destra.

Tutti gli altri sommatori di colonna sono sommatori a 3 bit.

a	b	c_i	S	c_o	$S \text{ Exp}$	$c_o \text{ Exp}$
0	0	0	0	0		
1	0	0	1	0	\bar{abc}^*	
0	1	0	1	0	\bar{abc}	
1	1	0	0	1		\bar{abc}
0	0	1	1	0	\bar{abc}	
1	0	1	0	1		\bar{abc}
0	1	1	0	1		\bar{abc}
1	1	1	1	1	abc	abc

* Per semplicità, nella forma canonica «carry-in» viene indicato con «c»

Figure 24: Esempio di half adder

Forme SOP:

$$S = a(\neg b)(\neg c) + (\neg a)b(\neg c) + (\neg a)(\neg b)c + abc$$

$$C = ab(\neg c) + a(\neg b)c + (\neg a)bc + abc$$

13.6 Adder a 32 bits

32 sommatori di colonna a 3 bit sono collegati in serie in modo che ognuno di essi svolga il ruolo di sommatore di colonna per ognuna delle 32 colonne.

Un half-Adder può essere utilizzato nella prima colonna.

13.7 Ripple-carry adder

Sommatore a 4 bit dei numeri binari (A_3, A_2, A_1, A_0) con (B_3, B_2, B_1, B_0): Il

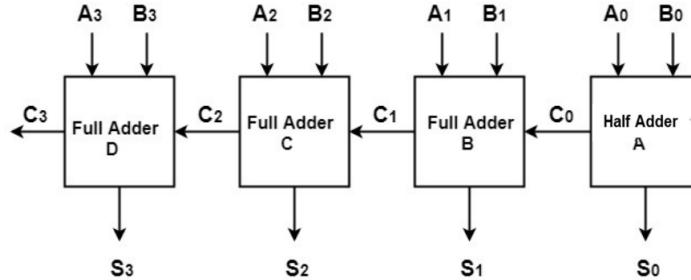


Figure 25: Ripple carry adder

percorso più lento (critico) passa attraverso il calcolo dei carry.

Il tempo di attesa totale è lineare nel numero di bit. Prima di quel tempo la somma non deve considerarsi valida

14 ALU

14.1 Realizzazione della sottrazione

Per realizzare la sottrazione si utilizza sempre il sommatore però bisogna invertire di segno b.

- Invertire b.
- Settare il carry-in del sommatore più a destra ad 1 (è necessario un full adder anche nella prima colonna a destra).
- Introdurre due segnali di selezione:
 - Inversione: dispone la selezione dell'ingresso b in forma vera o negata.
 - Selezione carry: dispone di settare il carry-in a 1, anziché a 0.
- Si può fare tutto con un unico segnale.

Tutto ciò è reso possibile dalla porta logica XOR, che in questo caso funge da invertitore programmabile. Cioè, se metto un ingresso a 1, ad esempio a , allora:

$$\begin{aligned} b = 0 &\Rightarrow \text{out} = 1 \\ b = 1 &\Rightarrow \text{out} = 0 \end{aligned}$$

Cioè $out = \neg b$. Se l'ingresso a è a 0, allora $out = b$.

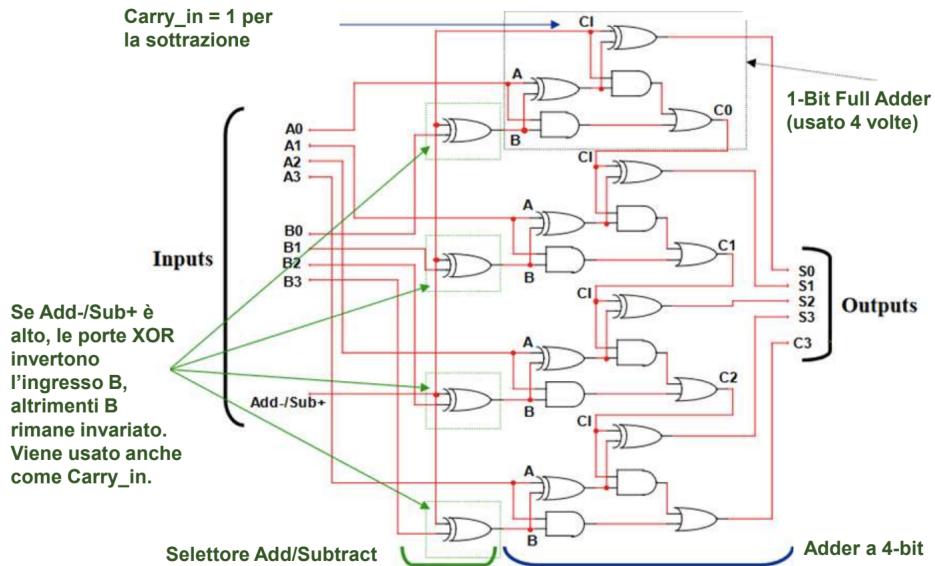


Figure 26: Rete per le somme e le sottrazioni

14.2 ALU

L'unità aritmetico logico è il cuore di un calcolatore. È un circuito combinatorio flessibile:

- Svolge diverse funzioni logiche combinatorie su ingressi ad n bits.
- Un comando (opportunamente decodificato) seleziona quale delle diverse funzioni calcolabili viene propagata un'uscita.
- Le funzioni sono di tipo logico e aritmetico.
- Genera anche bit ausiliari che rappresentano eccezioni o informazioni riguardo agli operandi o al risultato.
- In una CPU ci sono diverse ALU, anche se di complessità diversa.

14.3 Struttura

Sono stati proposti diversi tipi di ALU. Il tipo più diffuso è quello di tipo "bit-sliced", in cui una ALU a n bitss viene costruita a partire da "n-slice", ovvero n ALU a 1 bit ciascuna.

La ALU è essenzialmente costruita attorno ad una precisa architettura di sommatore ad n bitss, per esempio il Ripple Carry Adder.

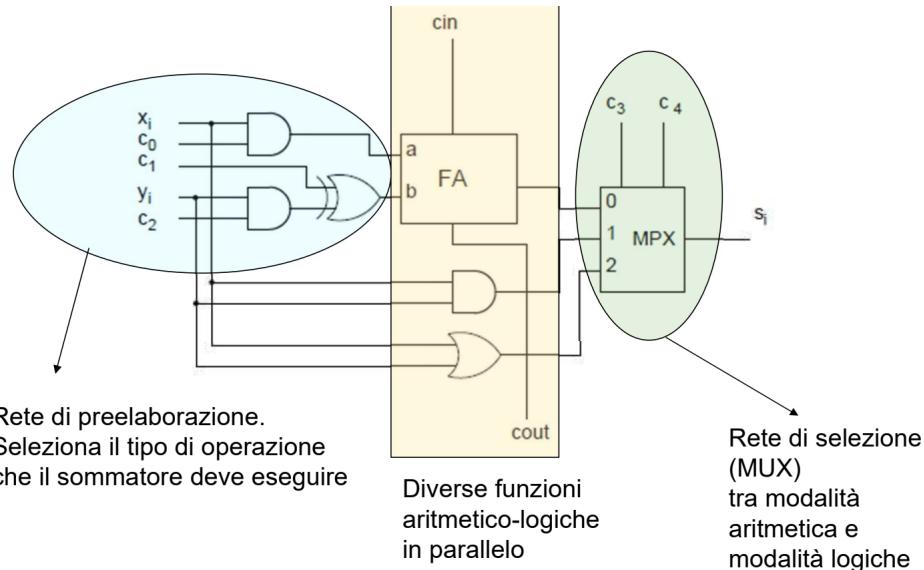


Figure 27: ALU a 1 bit

14.4 ALU-funzioni logiche

In questo esempio, abbiamo previsto solo due funzioni logiche in parallelo. Si tratta di operazioni logiche bit-per-bit il cui risultato è indipendente dal carry-in.

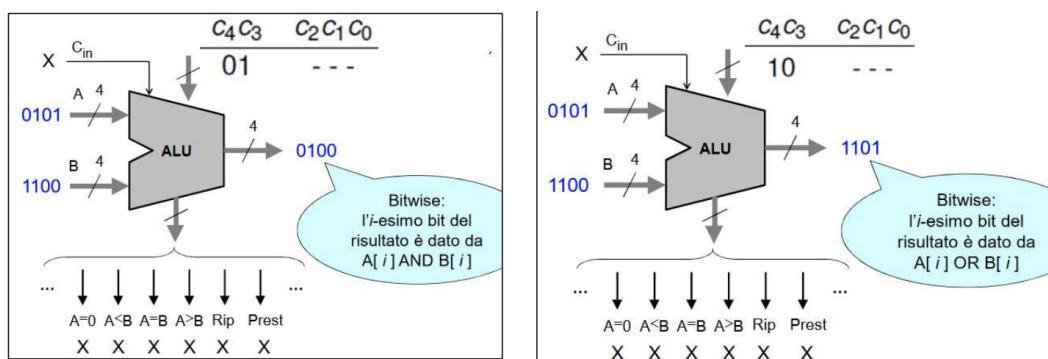


Figure 28: ALU a 1 bit

14.5 Funzioni aritmetiche

c4 e c3 pilotano il multiplexer: "00" seleziona le funzioni aritmetiche

$$\begin{aligned} a &= x_i AND c_0 & S &= (a \oplus b) \oplus c_{in} \\ b &= c_1 XOR(y_i AND c_2) & C_0 &= a AND b + (a \oplus b) AND c_{in} \end{aligned}$$

$c_4 c_3$	$c_2 c_1 c_0$	a	b
00	000	0	0
00	001	x_i	0
00	010	0	1
00	011	x_i	1
00	100	0	y_i
00	101	x_i	y_i
00	110	0	y'_i
00	111	x_i	y'_i

Figure 29: Tabella funzioni aritmetiche

14.6 ALU ad n bitss

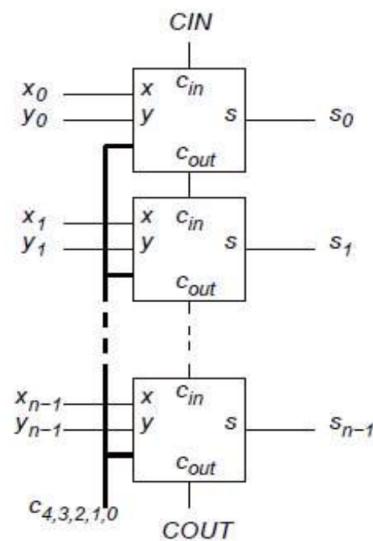


Figure 30: ALU a 1 bit

- Connessione di tante mini-ALU ad 1 bit logicamente in parallelo (il ritardo di propagazione invece è "seriale", dal CIN al COUT).
- Ogni uscita di ogni mini-ALU contribuisce un bit all'uscita della ALU complessiva.
- I segnali di controllo $c_4 c_3 c_2 c_1 c_0$ sono portati a tutte le mini-ALU.
- I carry-out di ogni mini-ALU sono collegati al carry-in della mini-ALU successiva.
- Si evidenziano un carry-in globale ed un carry-out globale.

Assumiamo che le configurazioni binarie in ingresso/uscita codifichino numeri interi rappresentati in complemento a 2.

ALU n BIT		
$c_4 c_3 c_2 c_1 c_0$	$S (CIN = 0)$	$S (CIN = 1)$
00 000	$(0)_2$	$(1)_2$
00 001	$(X)_2$	$(X + 1)_2$
00 010	$(-1)_2$	$(0)_2$
00 011	$(X - 1)_2$	$(X)_2$
00 100	$(Y)_2$	$(Y + 1)_2$
00 101	$(X + Y)_2$	$(X + Y + 1)_2$
00 110	$(Y)_2^{c1}$	$(Y^{c1} + 1)_2 = (Y)_2^{c2} = (-Y)_2$
00 111	$(X + Y^{c1})_2$	$(X + Y^{c1} + 1)_2 = (X - Y)_2$

Figure 31: Tabella funzioni aritmetiche

14.7 Bit di flag

- **Bit di zero:** vale 1 se il risultato $S = 0$. Serve quando in C scrivo $if(a == b)$. Basta sottrarre A-B in ALU e controllare il bit di flag.
- **Bit di parità:** fornisce la parità del risultato. Serve per codificare il risultato con un codice a rilevazione di errore e renderlo robusto ad eventuali guasti.
- **Bit di carry: carry-out.** Serve per gestire aritmetiche che rappresentano i numeri su una quantità di bit maggiore di quella macchina.

- **Bit di overflow:** vale 1 se il risultato non è contenuto in n bitss. Il risultato non è rappresentabile con i bits della ALU. Scattano routine software per la gestione della eccezione.
- **Bit di segno:** vale 1 se il risultato è negativo. Se ho per esempio $if(A > B)$.

15 Macchina a Stati Finiti

15.1 Logica sequenziale

- **Logica combinatoria:** il valore delle uscite dipende univocamente dal valore istantaneo degli ingressi.
- **Logica sequenziale:** il valore delle uscite O_i non è univocamente determinato dal solo valore degli ingressi I_i . Esiste anche un vettore di variabili di stato S_i che determina il valore delle uscite.

Le variabili di stato sono funzione degli ingressi e dello stato ad un tempo precedente (mantengono memoria della storia precedente):

La realizzazione hardware di una funzione logica sequenziale prende il nome di *macchina a stati finiti*.

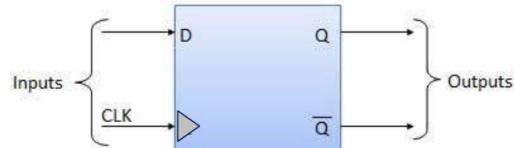
15.2 Operatore sequenziale elementare

Dall'istante in cui avviene una transizione da 0 a 1 dell'ingresso CLOCK, l'uscita Q assume lo stesso valore dell'ingresso D. In assenza di transizioni dell'ingresso CLOCK l'uscita rimane uguale a sé stessa, a prescindere dall'attività su D. L'uscita è disponibile anche in forma negata.

A causa della sua sensitività al clock, il flip-flop D viene anche chiamato "edge-triggered".

È un semplice circuito sequenziale: la sua uscita non dipende solo dal valore istantaneo degli ingressi di clock e D. Difatti, per uguali valori di clock e D in ingresso possiamo avere diversi valori di Q in uscita.

È un semplice "elemento di memoria": ricorda il valore dell'ingresso D dall'ultimo fronte di salita del clock.



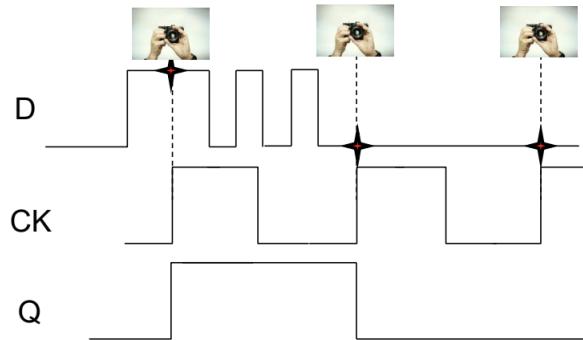


Figure 32: Clock e flip flop

È il componente base per realizzare i registri.

Combinando in parallelo n flip-flop pilotati dallo stesso clock, otteniamo un registro ad n bits.

15.3 Sintesi di funzioni sequenziali

Per la logica combinatoria, esistono forme canoniche in grado di esprimere qualunque funzione logica. Analogamente, esistono strutture canoniche in grado di esprimere qualunque funzione sequenziale. La più diffusa prende il nome di macchina a stati di Moore.

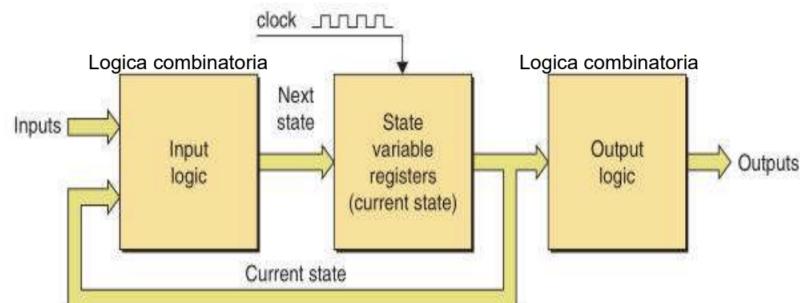


Figure 33: Macchina a stati di Moore

Abbiamo che l'output $O_i = f(S_i)$ e che lo stato è $S_i = g(I_{i-1}, S_{i-1})$.

Ad ogni fronte, lo stato futuro evolve in stato presente, avendo anche un nuovo effetto sulle uscite.

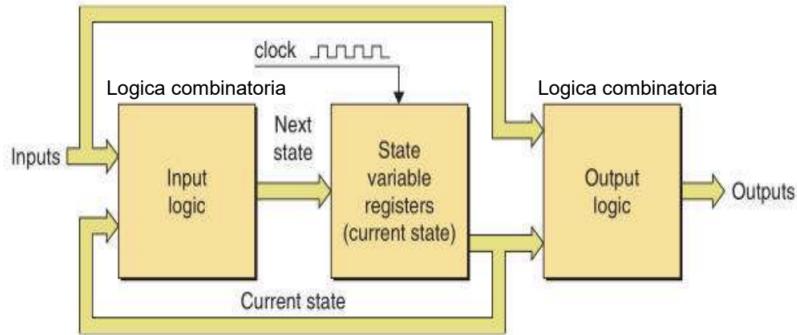


Figure 34: Macchina a stati di Mealy

Abbiamo che l'output è $O_i = f(I_i, S_i)$ e che lo stato è $S_i = g(I_{i-1}, S_{i-1})$.

Nei sistemi sequenziali sincroni, il cambiamento di stato avviene solamente in corrispondenza del fronte di un segnale di sincronizzazione globale di tutto il sistema.

15.4 Clock

Allo scoccare del nuovo segnale di clock ogni funzione logica compie una quantità di lavoro successivo tanto quanto basta per arrivare al fronte successivo senza sovrapporsi ad esso.

La frequenza di clock è determinata dal percorso di segnale più lento, chiamato critical path.

15.5 Specifica macchina a stati

La specifica di una funzione combinatoria si fa normalmente mediante una tabella di verità. Nel caso delle reti sequenziali, la specifica si fa più complicata. Esistono diversi formalismi per la specifica:

- Diagrammi ASM.
- Tabella di transizione degli stati.
- Diagramma di transizione degli stati.

15.6 Esempio: vending machine

- Se inserisci 10 cent, ti do la gomma.
- Se inserisci due monete da 5 cent, ti do la gomma.

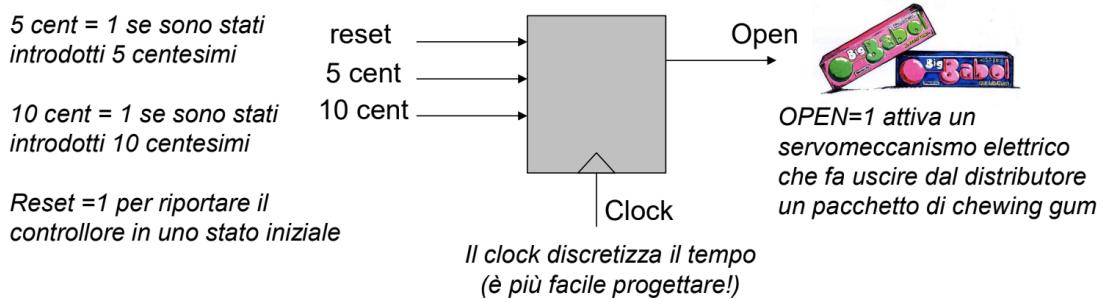


Figure 35: Macchina a stati di Mealy

- Se inserisci una moneta da 5 cent ed una da 10, ti do la gomma.
- Quando viene erogata la gomma, la macchina torna allo stato iniziale.

15.7 State transition table

Tabella di Transizione Degli Stati					
Reset	5 cent	10 cent	Present State	Next State	Open
1	-	-	-	Start	0
0	0	0	Start	Start	0
0	1	0	Start	Five	0
0	0	1	Start	Ten	0
0	1	0	Five	Ten	0
0	0	1	Five	Ten	0
0	0	0	Five	Five	0
0	-	-	Ten	Start	1

Figure 36: State transition table

15.8 State transition diagram

- Ogni nodo nel grafo è solo uno stato.
- Le frecce sono le transizioni di stato, con annotati gli ingressi che le provocano.

- Frecce non annotate rappresentano transizioni incondizionate.
- Gli stati riportano le uscite che prendono valore 1 (0隐式).

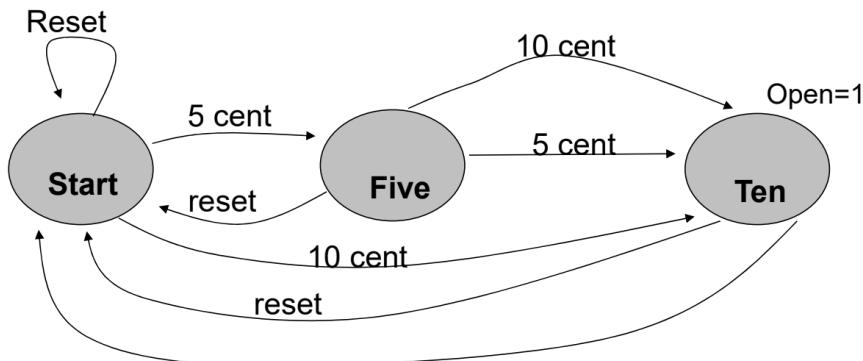


Figure 37: State transition diagram

15.9 Diagrammi ASM

Strumento grafico alternativo ai diagrammi degli stati che si presta a maggior flessibilità e maggiormente idoneo a rappresentare macchine complesse.

Un ASM, così come un STD, è sempre chiuso: non esiste uno stato in cui la macchina si ferma senza poter evolvere in un altro stato.

15.10 Sintesi hardware di una rete sequenziale

Il procedimento più intuitivo è il seguente: $\text{ASM} \Rightarrow \text{STT} \Rightarrow \text{STT codificato} \Rightarrow \text{K-map} \Rightarrow \text{sintesi}$.

STT codificato: assegnare una rappresentazione binaria a ciascuno stato.

Stato	X1	X0
Start	0	0
Five	0	1
Ten	1	0

Figure 38: STT codificato

16 Microarchitettura

16.1 Vista astratta di un microprocessore MIPS

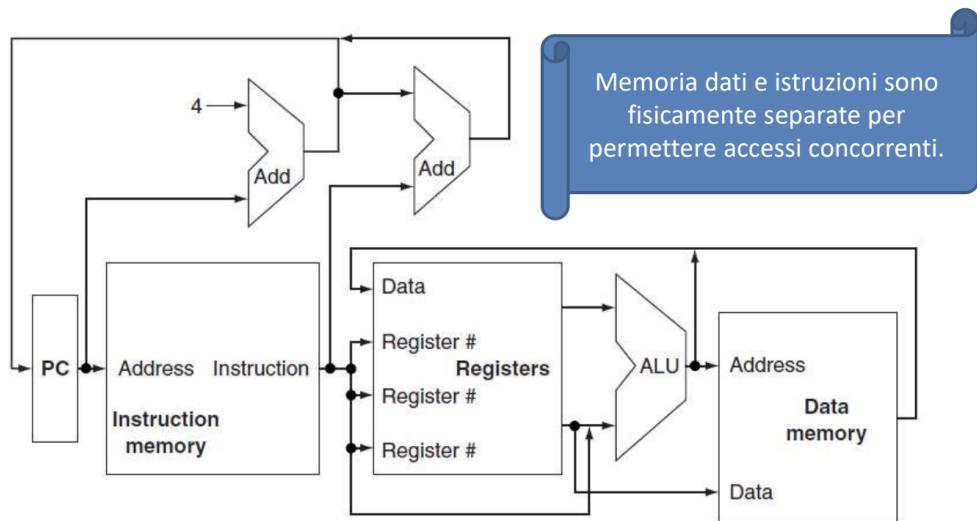


Figure 39: Microarchitettura

Ad eccezione della jump, tutte le istruzioni utilizzano la stessa unità aritmetico logica (ALU), dopo aver letto i registri.

- **Comune a tutte le istruzioni:** ad ogni istruzione viene fatto il fetch dalla memoria istruzioni. Viene fatto anche il fetch dei registri specificati nell'istruzione nella memoria istruzioni.
- **Istruzioni aritmetico-logiche:** utilizzo della ALU e poi scrittura del risultato nel register-file.
- **Istruzioni load e store:** la ALU calcola un indirizzo di memoria. Se effettuo una scritta (store) l'uscita della ALU è un indirizzo di scrittura che finisce nel data-memory. Per una lettura l'indirizzo generato dalla ALU finisce nel data-memory, il dato uscirà e andrà scritto nel register-file.
- **Istruzioni di branch:** eseguo il confronto tra i valori nella ALU. Se la condizione non è verificata allora il prossimo indirizzo sarà $PC+4$. Se la condizione è verificata il prossimo indirizzo sarà $PC+4+offset$.

16.2 Vista più dettagliata

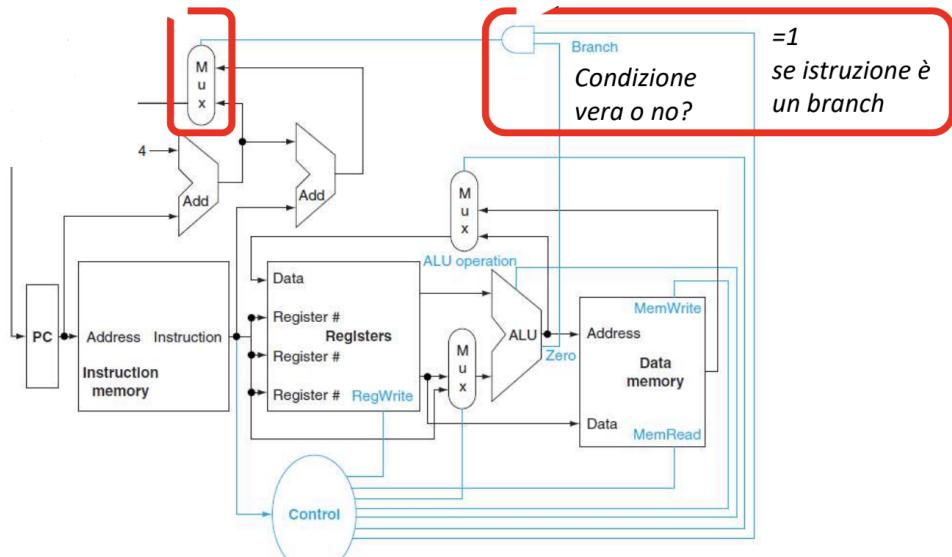


Figure 40: Microarchitettura

Il primo multiplexer serve per selezionare il prossimo valore del PC, se devo fare: $PC+4$ o $PC+4+off$. Il segnale di selezione proviene dall'unità di controllo (control).

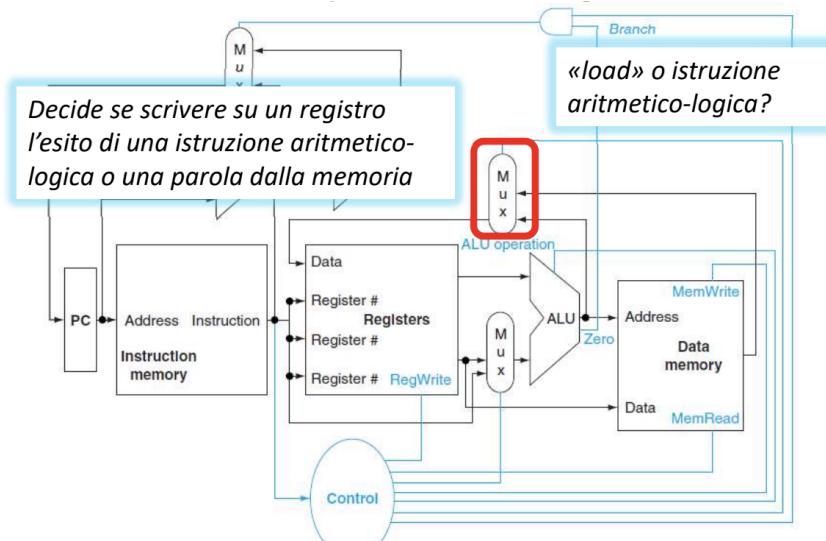


Figure 41: Microarchitettura

È un selettore che serve per decidere chi ha diritto di scrivere nel register-file (istruzione di tipo aritmetico-logica o load). Il segnale di selezione proviene dall'unità di controllo e va in base al OP-code dell'istruzione.

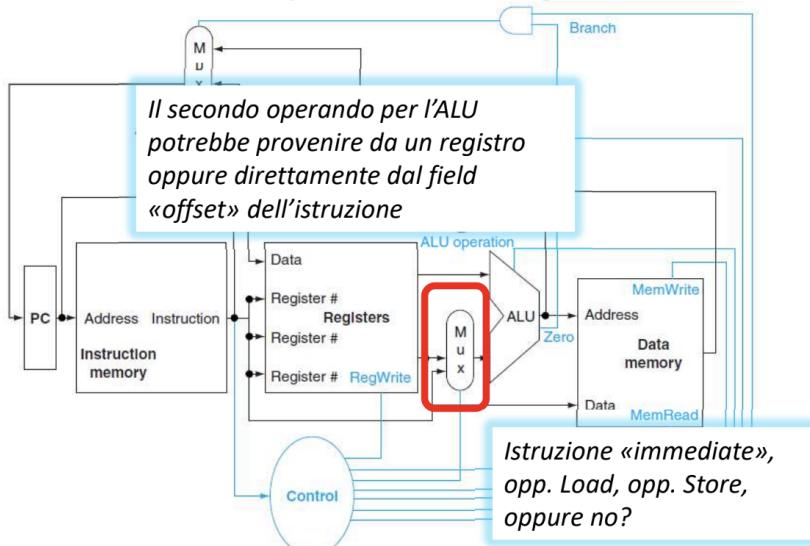


Figure 42: Microarchitettura

Nel caso di un'istruzione aritmetico-logica questo multiplexer mi serve per selezionare il registro nel caso non sia di tipo I. Nel caso di un'istruzione di tipo I prendo il valore immediate.

16.3 Costruzione del datapath

16.3.1 Implementazione di tipo R

Ingredienti:

- **Register-file:** struttura dove all'interno vi sono tutti i registri; il register file è una memoria multi-porta, la quale ha due porte di lettura e una porta di scrittura. Da qui vengono letti i registri per effettuare le operazioni aritmetico-logiche sulla ALU.
- **ALU.**

16.3.2 Implementazione LOAD/STORE

Ingredienti:

- **Register-file:** da qui dobbiamo estrarre il registro di base.
- **ALU:** dobbiamo sommare il registro di base con l'offset.
- **Data-memory:** memoria in cui andremo a leggere o a scrivere il dato. Ci sono due segnali di controllo che dicono alla memoria se il ciclo è di lettura o scrittura.
- **Sign-extend:** serve per l'estensione del segno da 16 a 32 bits.

16.3.3 Implementazione di istruzioni di tipo R e LOAD/STORE

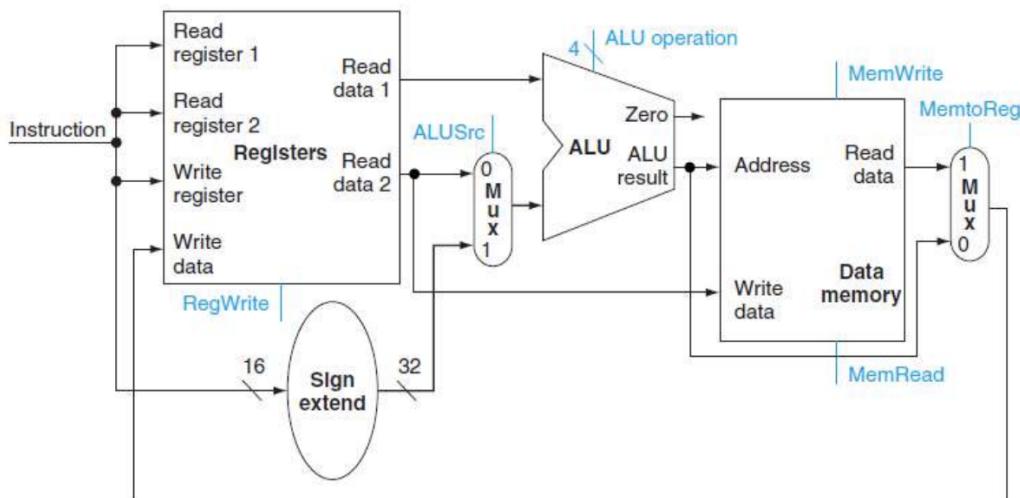


Figure 43: Implementazione datapath per istruzioni R e load/store

Il primo multiplexer serve per selezionare tra il secondo valore letto da register-file (per l'istruzione R) e l'offset della load/store.

Il secondo multiplexer serve per capire chi deve scrivere nel register-file. Potrebbe essere l'ALU (nel caso dell'istruzione R), oppure la data-memory (nel caso di una load).

16.3.4 Implementazione di salti condizionati

Se la condizione non è vera (in gergo: NOT TAKEN) allora il nuovo PC è $PC + 4$. Se la condizione è vera (TAKEN) il nuovo PC è il *branch target address*.

Per eseguire un branch bisogna effettuare due operazioni fondamentali.

- Calcolo di un nuovo branch target address.
- Confronto del contenuto dei due registri.

Queste due operazioni le eseguiamo in parallelo.

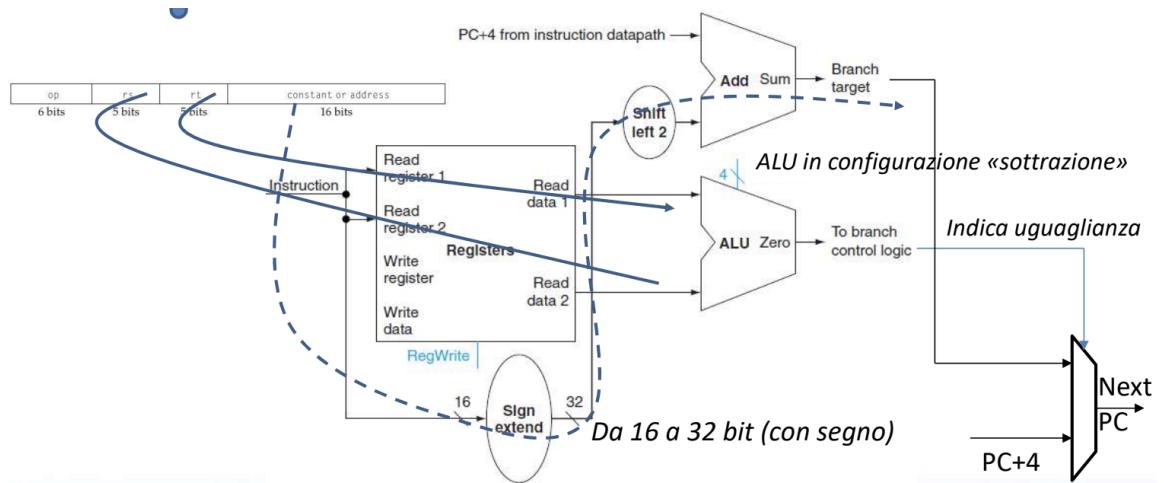


Figure 44: Implementazione datapath per i branch

I due registri vengono portati nel register-file, i cui valori vengono portati nella ALU per verificare la condizione. Se la condizione è vera si attiva il flag di controllo "zero".

Contemporaneamente, il valore immediate a 16 bits verrà esteso a 32 bits, moltiplicato per 4 (sll 2) e sommato al PC+4.

16.3.5 Datapath completo

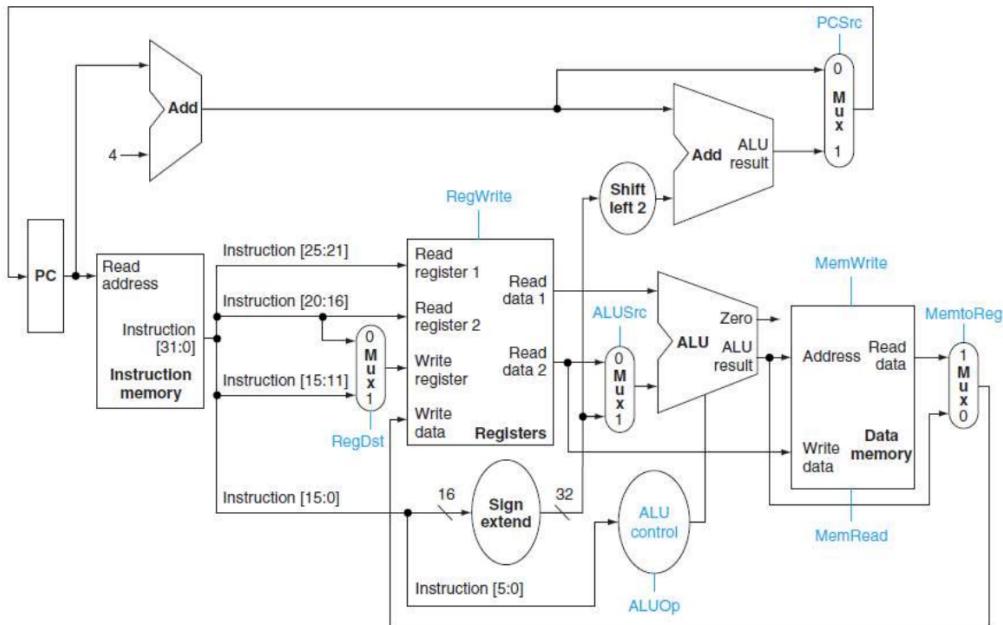


Figure 45: Datapath completo

16.4 Control-path

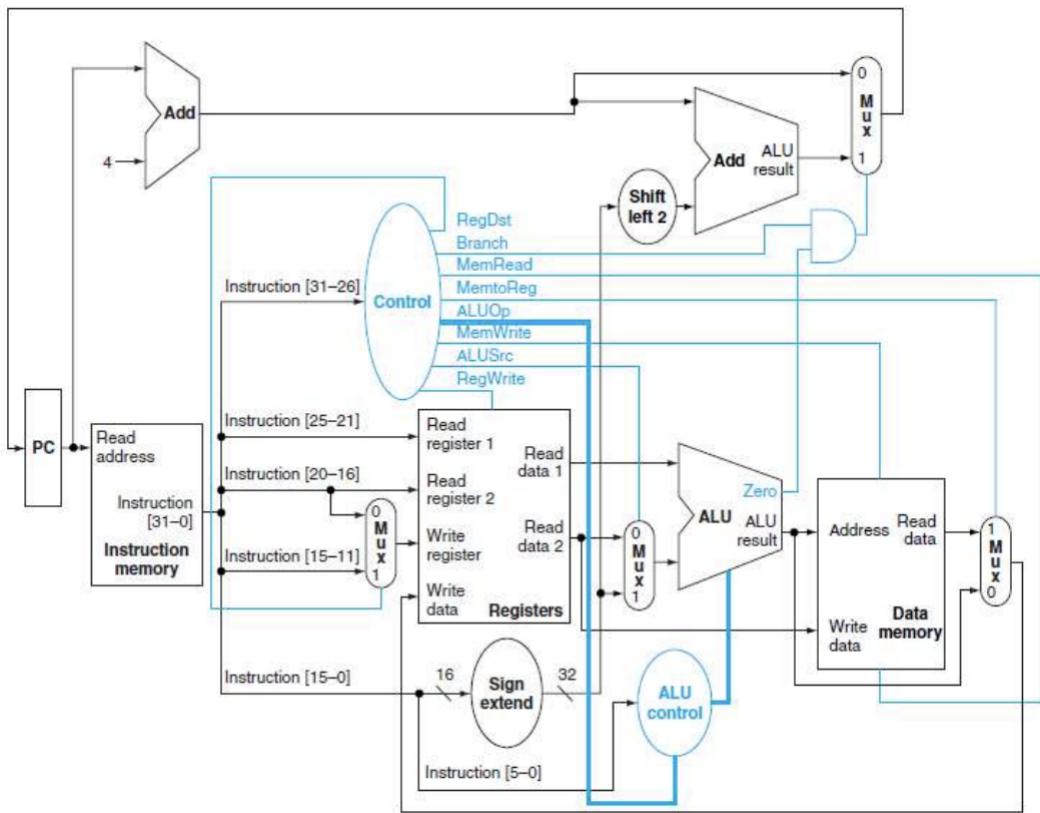


Figure 46: Implementazione control-path

16.5 Implementazione istruzione JUMP

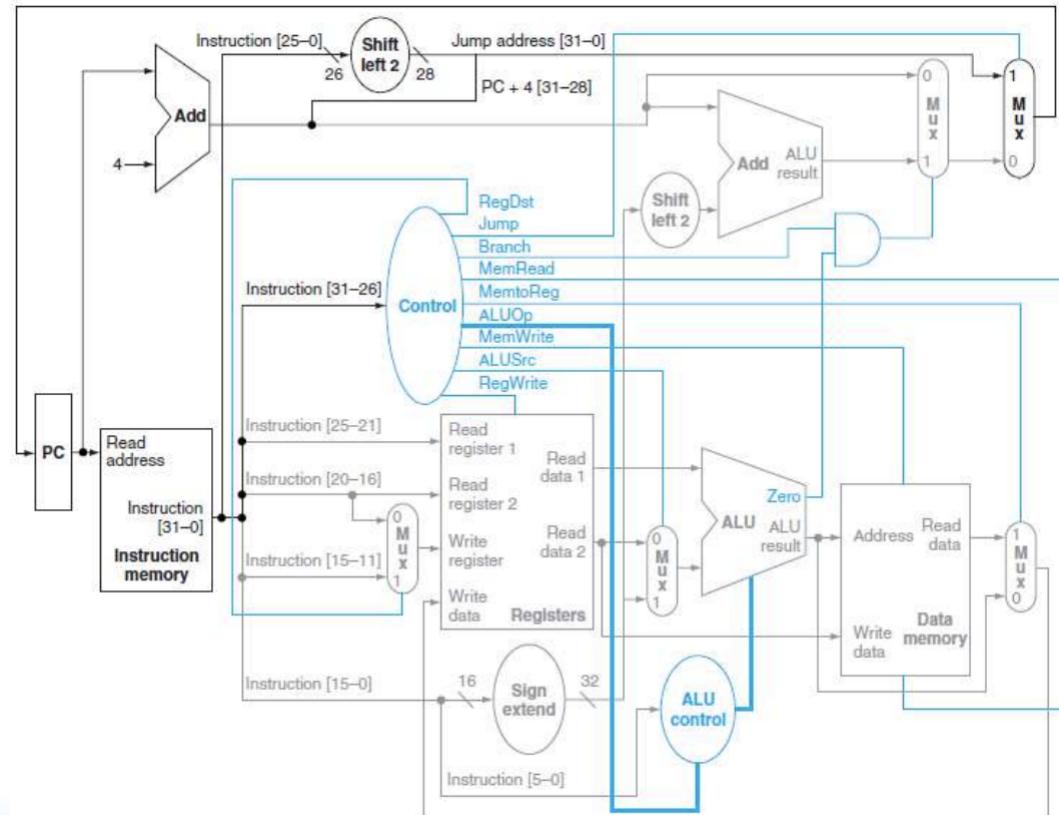


Figure 47: Implementazione istruzione jump

1. Moltiplichiamo il campo address per 4 con uno shift left di 2, ottenendo così un indirizzo a byte di 28 bits.
2. Per ottenere i restanti 4 bits concateniamo il nostro nuovo address con i primi 4 bits di PC+4.
3. Con l'ausilio di un altro multiplexer aggiorniamo il PC.

17 Pipelining

Nella precedente sezione abbiamo costruito un modello di microarchitettura; con il problema che però è in grado di eseguire un'istruzione per volta.

Con il pipelining risolviamo questo problema.

Il pipelining è una tecnica implementativa in cui l'esecuzione di istruzioni multiple viene sovrapposta nel tempo, sul modello di una "catena di montaggio". Il suo scopo è migliorare il throughput.

Idealmente una pipeline a N stadi ha un throughput che è N volte maggiore rispetto ad una architettura a singolo ciclo.

17.1 Stadi di pipeline

17.1.1 Instruction fetch

Lettura di un'istruzione e creazione del nuovo valore del PC.

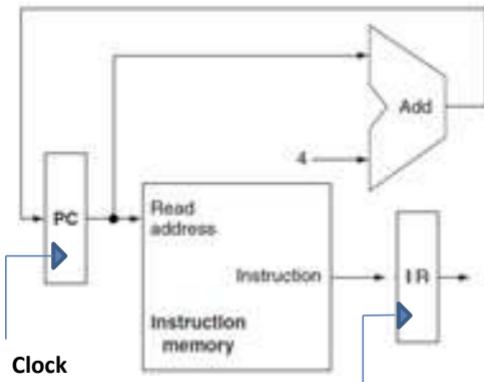


Figure 48: Instruction fetch

Per realizzare il pipelining è necessario aggiungere un registro in più per memorizzare l'istruzione.

In parallelo all'accesso alla memoria istruzioni, il prossimo valore del PC viene calcolato.

Il percorso più lento determina il periodo di clock supportato da questo "stadio di pipeline".

17.1.2 Instruction decode

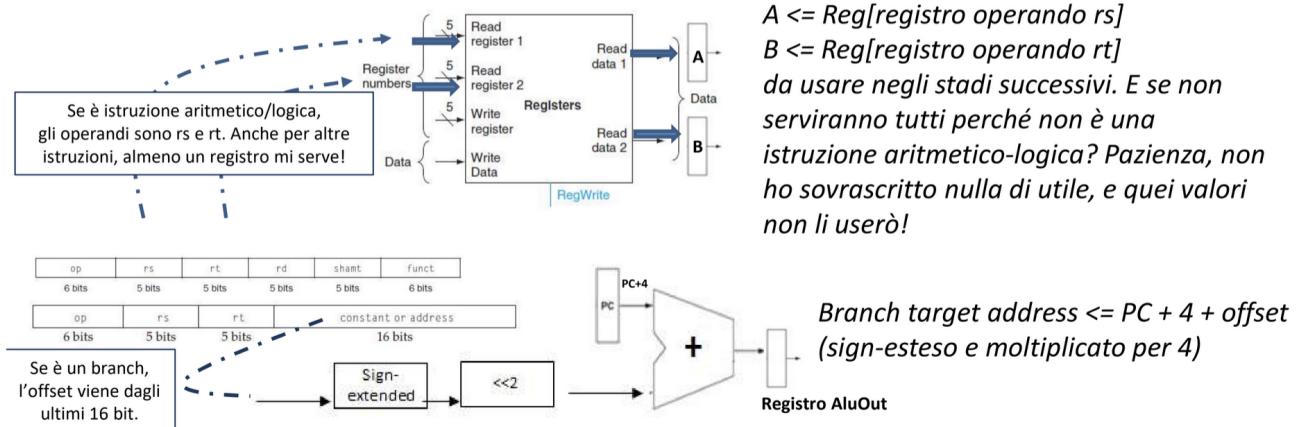


Figure 49: Instruction decode

Nuovi registri di memorizzazione: A, B, e AluOut.

17.1.3 Execution

- Load/store: il valore del registro A viene dato in ingresso all'ALU, e lo somma con l'offset di load/store. Il risultato viene immagazzinato in AluOut.
- Aritmetico-logica: i valori dei registri A e B vengono portati all'ALU, e il risultato viene portato su AluOut.
- Branch: viene fatta la differenza fra A e B e sulla base del flag di controllo "zero", pilota i multiplexer che abilitano AluOut piuttosto che PC+4.

17.1.4 Memory access

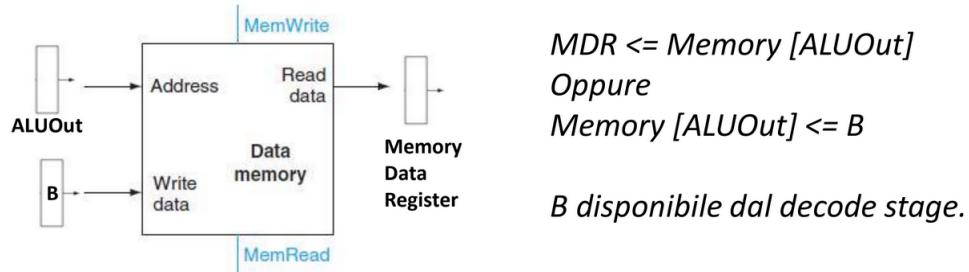


Figure 50: Memory access

Nel caso di una store allora l'istruzione è stata eseguita. Per la load c'è ancora un passaggio da fare.

Nuovi registri di memorizzazione: Memory Data Register.

17.1.5 Write back

Nel caso di una load:

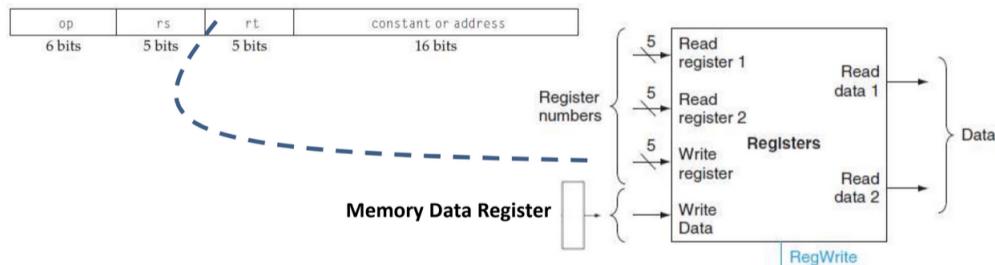


Figure 51: Write back A

Nel caso di un'istruzione di tipo R:

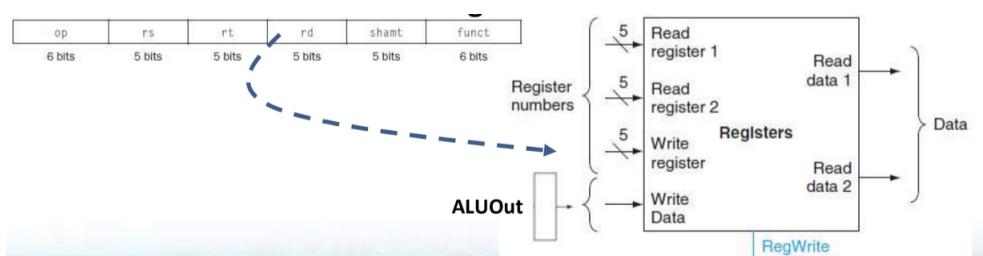


Figure 52: Write back B

17.2 Hazard

Ci sono situazioni col pipelining nelle quali la prossima istruzione non può essere eseguita nel ciclo di clock successivo (hazard).

- Hazard strutturali.
- Hazard dati.
- Hazard di controllo.

17.2.1 Hazard strutturali

Supponiamo di implementare il modello di Von Neumann, e che la memoria dati e memoria istruzioni coincidano.

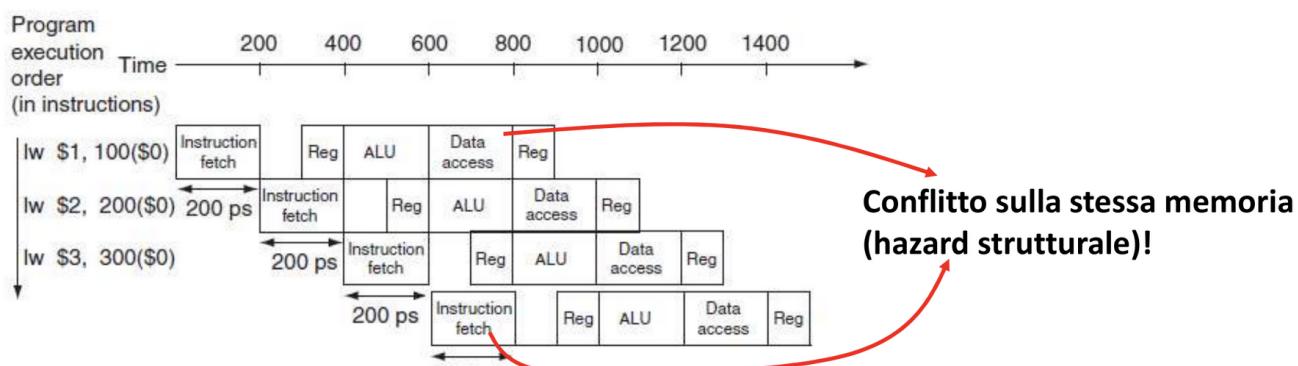


Figure 53: Hazard strutturale

Da qui ricaviamo una regola per evitare hazard strutturali: *ogni risorsa deve essere utilizzata in un solo stadio di pipeline.*

17.2.2 Hazard dati

Questi hazard capitano quando la pipeline deve essere fermata perché uno stadio ha bisogno del completamento di un altro.

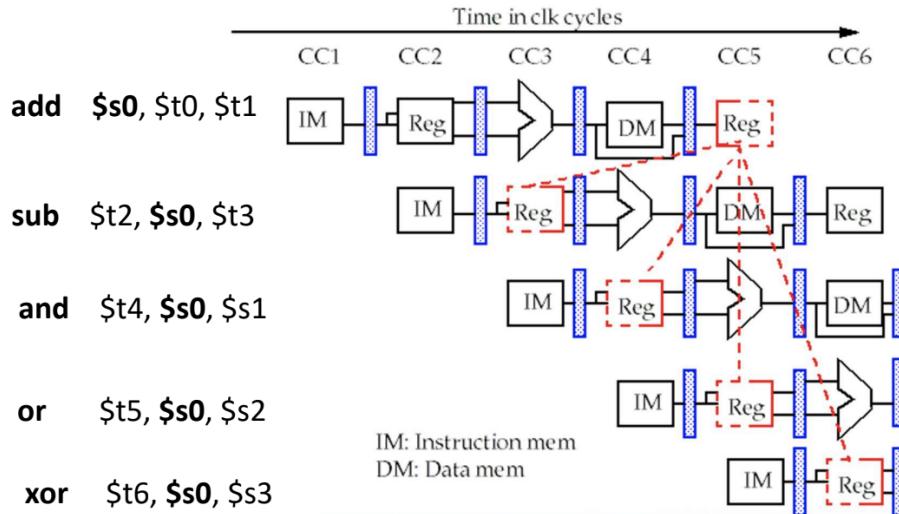


Figure 54: Hazard dati

In quest'immagine il problema è nella 2 e 3 istruzione, e sta nel fatto che noi stiamo operando su un valore non esatto di \$s0. \$s0 sarà aggiornato nell'istruzione 5 (con un register internal forwarding), ma nella 2 e 3 noi stiamo usando un valore vecchio.

Una soluzione al problema degli hazard dati potrebbe essere questa:

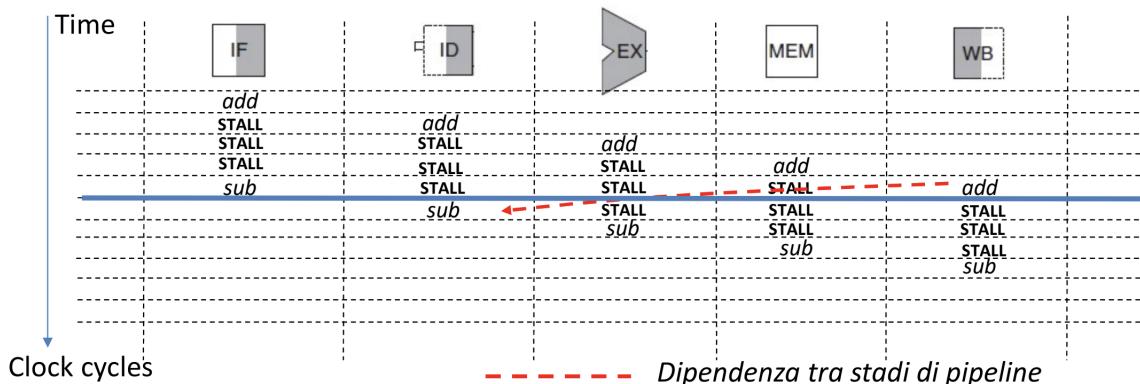


Figure 55: Soluzione hazard dati senza bypass

Per far sì che la sub ottenga il valore corretto di \$s0 dobbiamo aspettare che la prima istruzione scriva nel register-file il suo valore aggiornato. Quindi siamo costretti a mettere 3 istruzioni vuote (stall).

Un'altra soluzione, che è quella più intelligente, è implementare dei bypass:

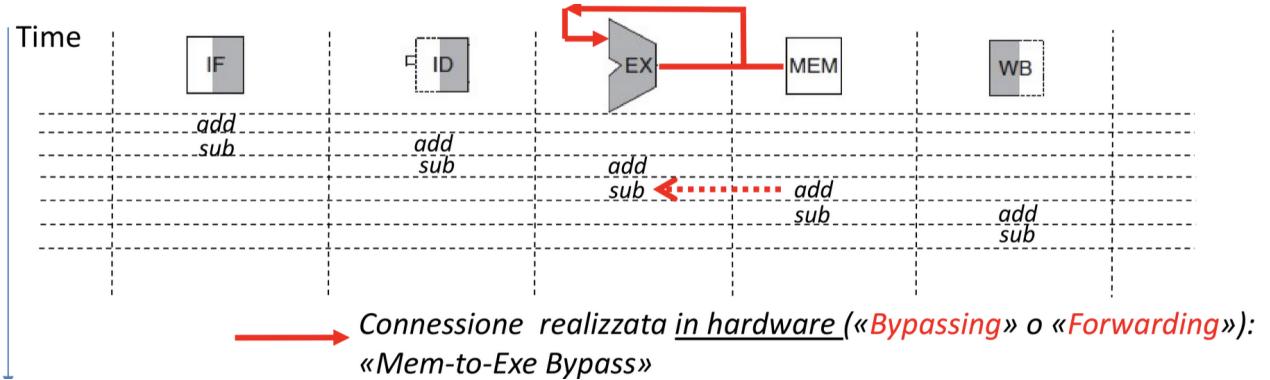


Figure 56: Soluzione hazard dati con bypass

Possiamo implementare un *bypass MEM-T0-EXE*, portando così il valore futuro di \$s0 dall'ingresso dello stadio MEM all'ingresso di EXE. In questo modo sovrapponiamo il valore sbagliato di \$s0 con quello corretto.

Non tutti gli hazard dati possono essere risolti con bypassing, e alla fine gli stalli di pipeline sono inevitabili.

Diamo per esempio le seguenti istruzioni:

lw \$s0, 1200(\$t1).

sub \$t2, \$s0, \$t3.

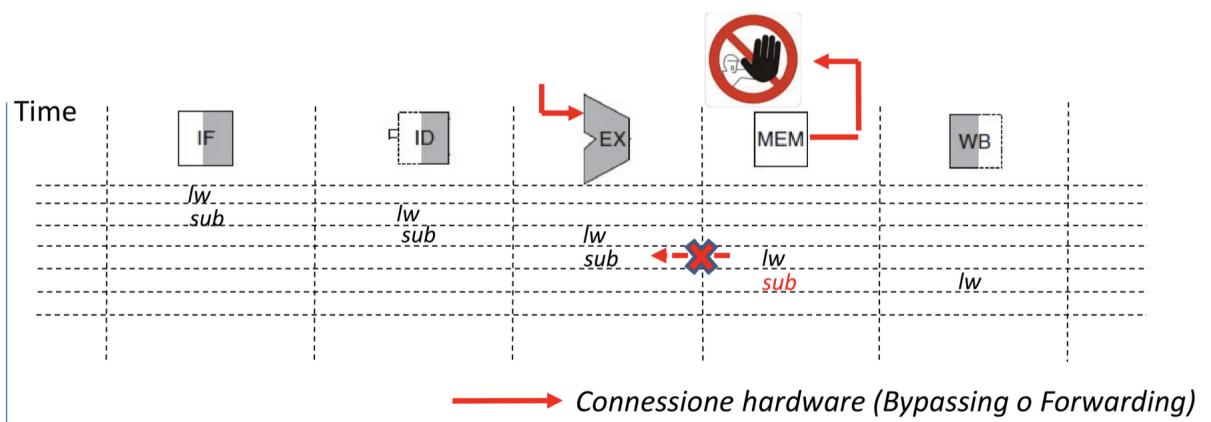


Figure 57: Hazard dati

Se proviamo ad implementare il bypass di prima ci troveremo con un problema: cioè che in questo caso, essendo l'istruzione una load-word, il nuovo valore di \$s0 non è disponibile all'inizio di MEM, ma alla fine di questa.

Per risolvere questo hazard l'unica soluzione è quella di mettere uno stall in mezzo alle 2 istruzioni, ed effettuare un bypass dall'ingresso di write back all'ingresso di execute WB-TO-EXE.

17.2.3 Hazard di controllo

Si verifica quando l'istruzione non può eseguire nel suo slot nominale nella pipeline, perché si è fatto il fetch dell'istruzione sbagliata.

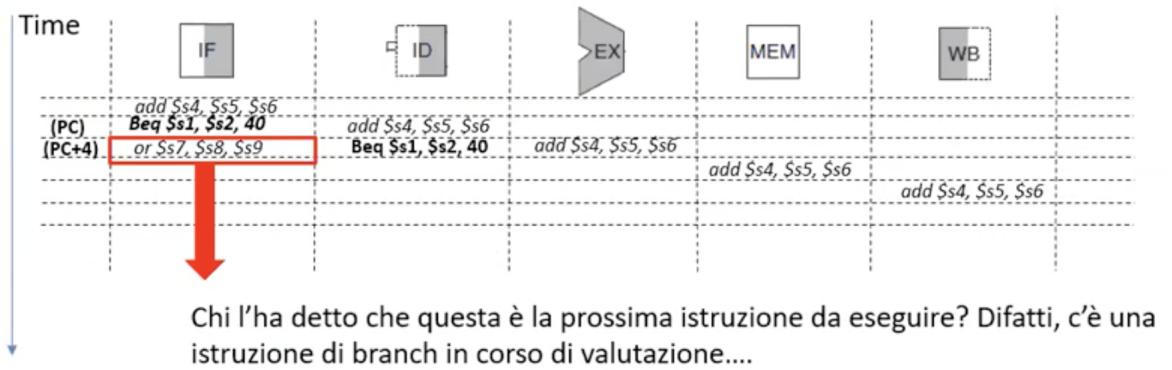


Figure 58: Hazard di controllo

Per risolvere il problema basterebbe inserire degli stall finché non viene testata la condizione. In questo caso basterebbe inserirne 2.

Questa soluzione però ha un costo troppo elevato.

Una soluzione intelligente è fare una branch-prediction. Cioè, quando faccio il fetch di un branch, prevedo se la condizione è verificata (TAKEN) oppure no (UNTAKEN).

Se voglio predire che il branch sarà UNTAKEN, allora mando subito nella pipeline l'istruzione successiva.

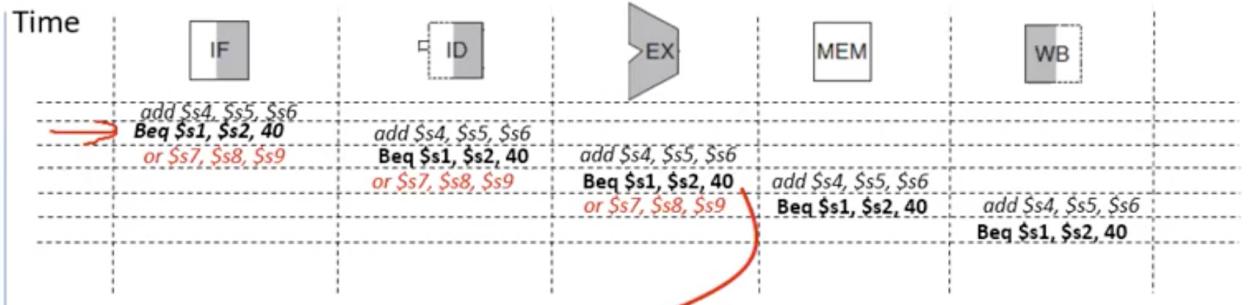


Figure 59: UNTAKEN corretto

In questo caso il branch è UNTAKEN quindi la mia previsione è corretta e così facendo non ho perso del throughput.

Se però la previsione non è corretta e il branch è TAKEN invece di un UNTAKEN:

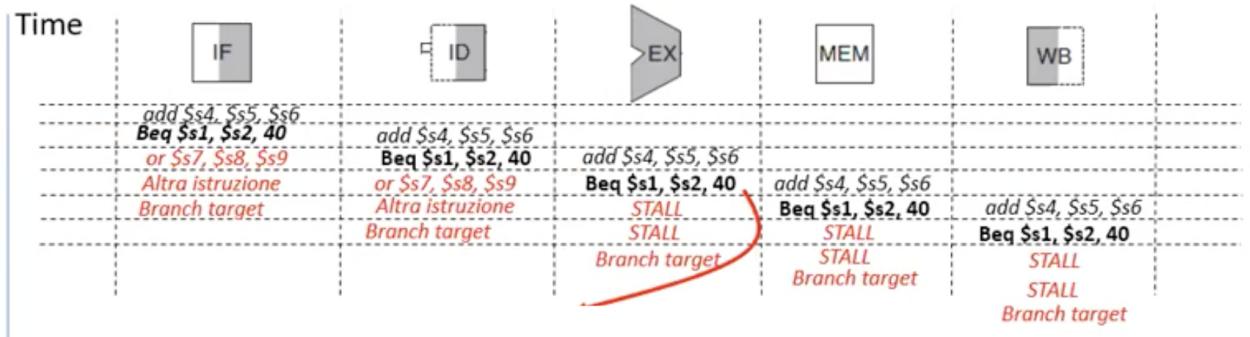


Figure 60: UNTAKEN non corretto

Il branch target (cioè l'istruzione giusta) è a PC+12, ma vengono comunque mandate le istruzioni alla pipeline. Per rimediare all'errore le istruzioni errate vengono trasformate in stall, così l'effetto di queste viene eliminato.

17.3 Implementazione del pipelining nell'architettura MIPS

In generale qualunque segnale viaggi in direzione opposta al data-path è sinonimo di hazard dati e strutturali.

Affinché l'informazione di uno stadio sia utile agli stadi successivi, è necessario salvarla in registri.

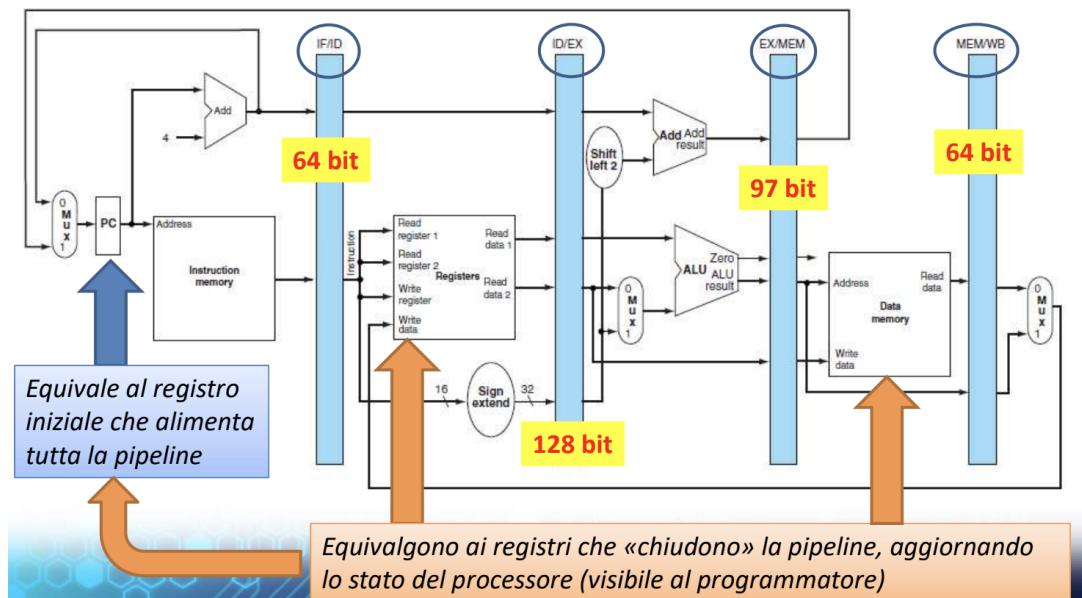


Figure 61: Registri della pipeline

Se il data-path è implementato così allora incorriamo subito in un hazard strutturale. Cioè se siamo nel caso di una load-word, quando arriviamo nella fase di WB dobbiamo usare la porta write register con il dato che proviene dalla data-memory. Ma se dopo viene un'altra istruzione la porta write register verrà corrotta dalla nuova istruzione, quindi si sta scrivendo su un registro sbagliato.

L'unico modo per risolvere il problema è agire sulle singole porte.

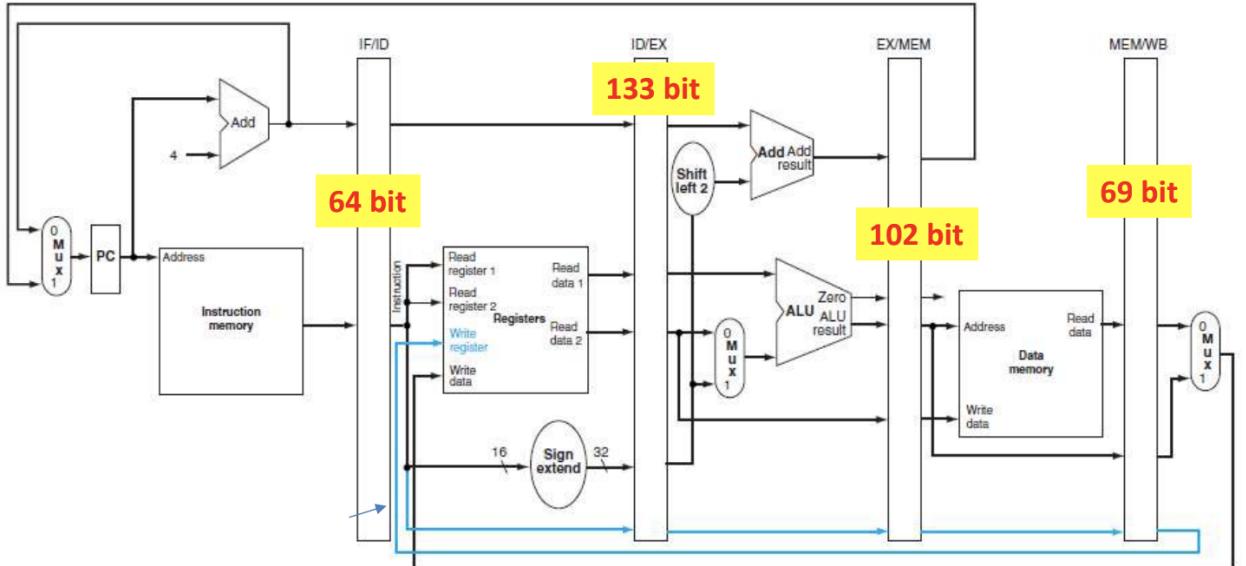


Figure 62: Soluzione hazard strutturale

L'indirizzo del registro su cui andare a scrivere viene portato in giro per tutti gli stadi di pipeline.

17.4 Main controller

Il main controller può settare tutti i segnali di controllo tranne uno (ALUOp) basandosi solamente sul campo OPCODE dell'istruzione.

L'unica differenza rispetto all'implementazione a singolo ciclo è che ora ogni segnale di controllo va settato durante il ciclo di pipeline corretto.

Si considera IF e ID come stadi "general purpose", cioè stadi dove l'istruzione passa sempre.

I primi segnali di controllo sono da settare nello stadio EXE. Quindi inseriamo il *main controller* nello stadio ID.

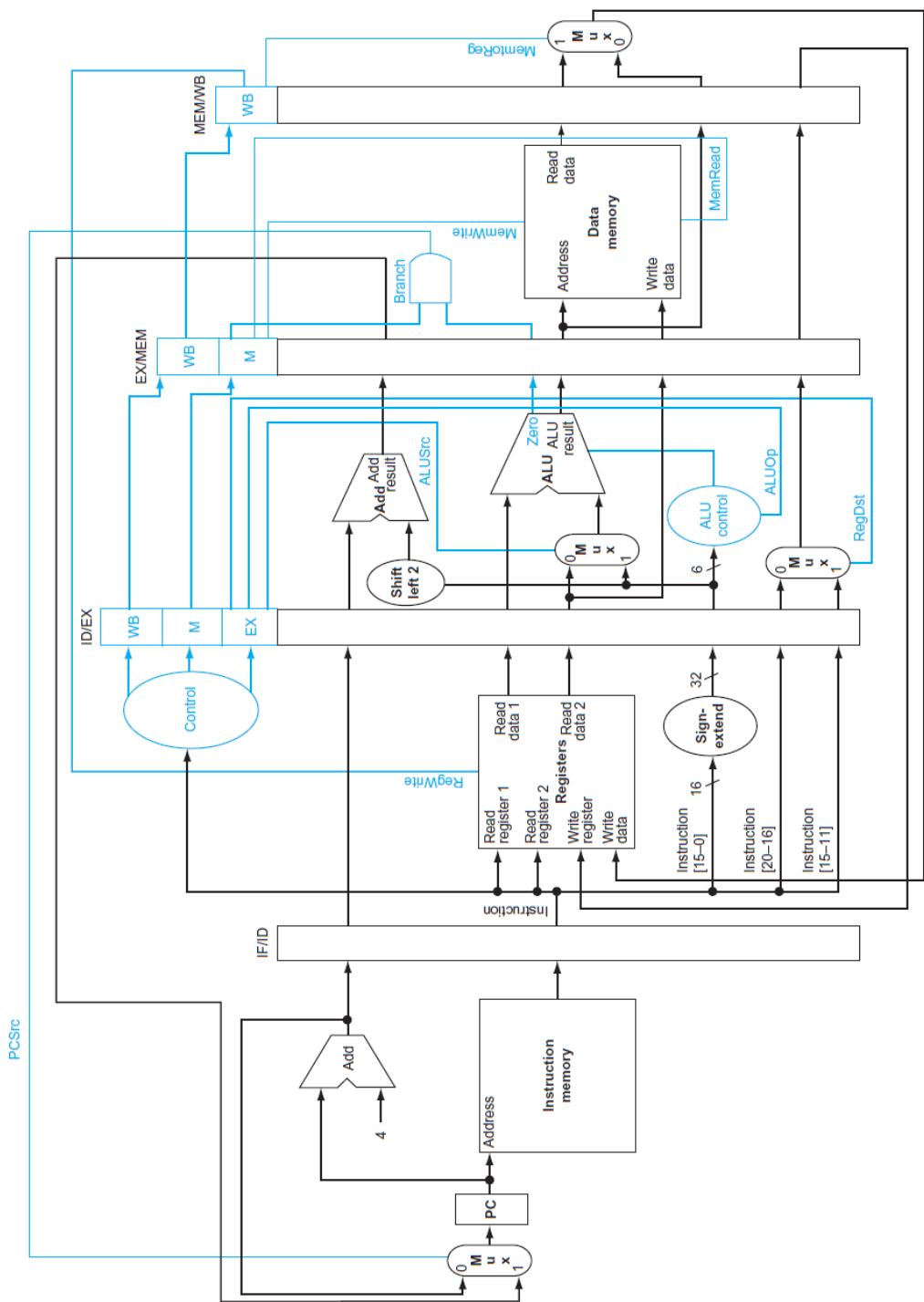


Figure 63: processore completo senza jump

18 Forwarding unit e Hazard detection unit

La forwarding unit si occupa di verificare quando c'è bisogno di fare un bypass o forwarding.

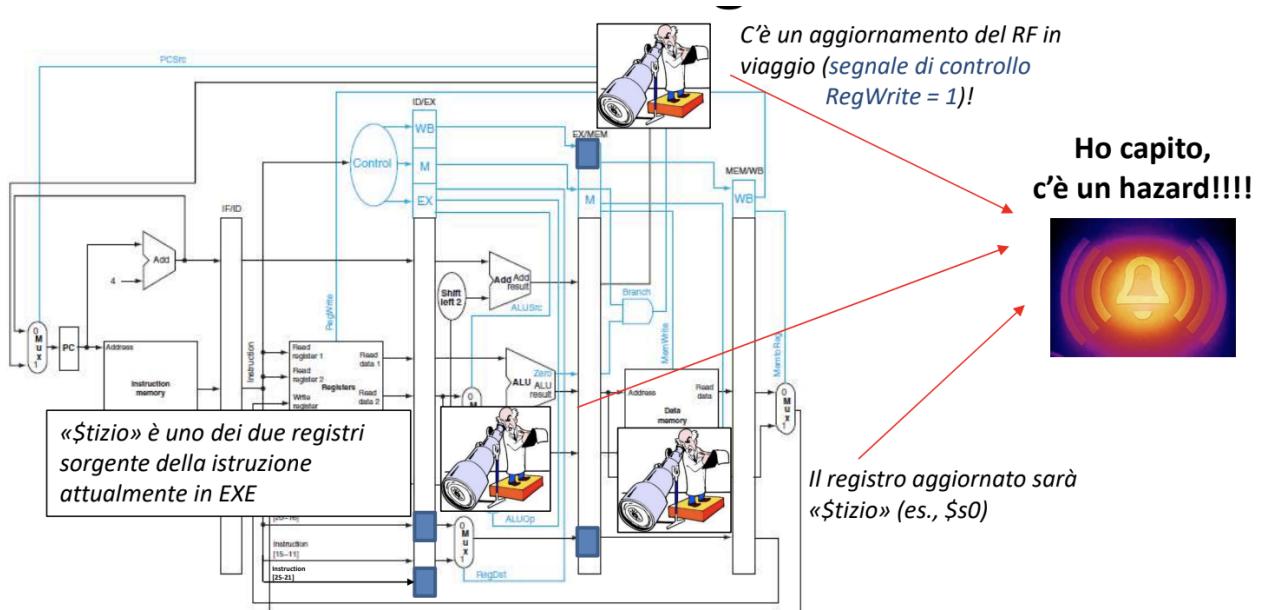


Figure 64: Forwarding unit

1. La forwarding unit monitora i sorgenti dell'istruzione nella fase di EXE.
2. Nello stadio di MEM andiamo a capire se nel ciclo precedente è stato generato un valore che deve essere scritto nel register-file; lo si capisce grazie al segnale di controllo "RegWrite".
3. Si verifica se il registro appena aggiornato è lo stesso registro presente nella fase di EXE.

C'è un registro destinazione nello stadio di MEM che sta viaggiando verso il register-file, per essere sovrascritto, e che coincide con un operando di EXE.

Da questo ragionamento la forwarding unit capisce che siamo in presenza di un hazard dati. Quindi va abilitato il ramo di bypass da MEM a EXE.

18.1 Dettaglio micro-architetturale

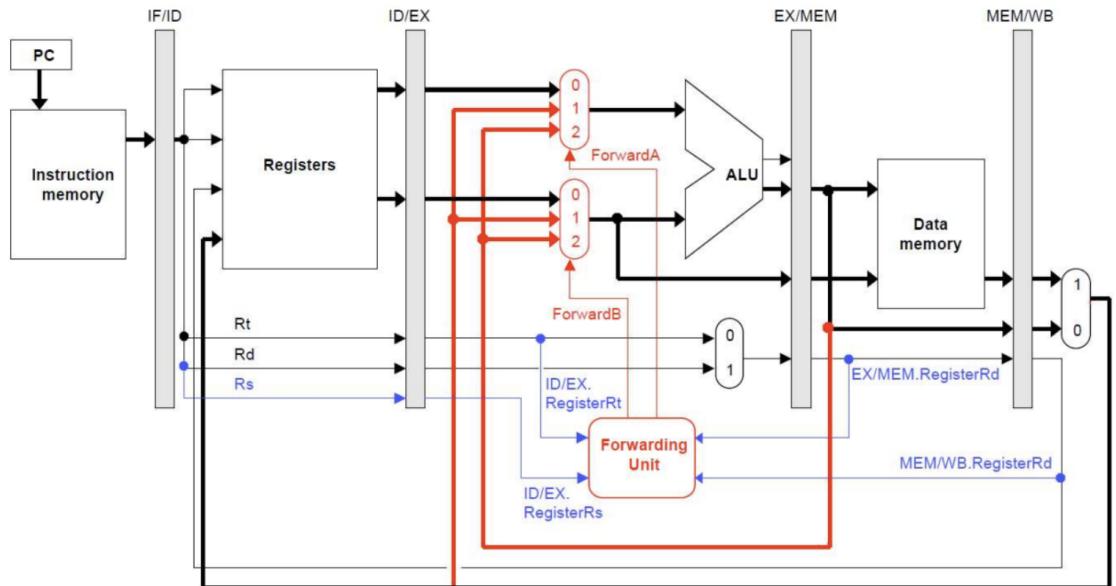


Figure 65: Forwarding unit nella micro-architettura

La forwarding unit si trova nello stato di EXE, perché è il punto giusto in cui cambiare gli operandi della ALU.

Gli ingressi dei multiplexer provengono dal register-file, dalla fase di MEM e dalla fase di WB.

La forwarding unit legge i registri operando dell'attuale fase di EXE, gli confronta con i registri che saranno aggiornati in futuro ma il cui valore è già stato generato nella pipeline, e decide se l'ALU prende in ingresso dal register-file con bypass dallo stadio di MEM o con bypass dallo stadio di WB.

18.2 Stalli della pipeline

Lo stall è inevitabile quando un'istruzione cerca di leggere un registro che una load sta cercando di scrivere.

Per inserire una stall dobbiamo ricorrere al *hazard detection unit*.

Per inserire una stall bisogna come prima cosa fare il fetch dell'istruzione, successivamente quando l'istruzione arriva nello stadio di EXE la hazard detection unit introduce una "bolla" per bloccare l'effetto dell'istruzione. Così si potrà eseguire l'istruzione nel ciclo dopo.

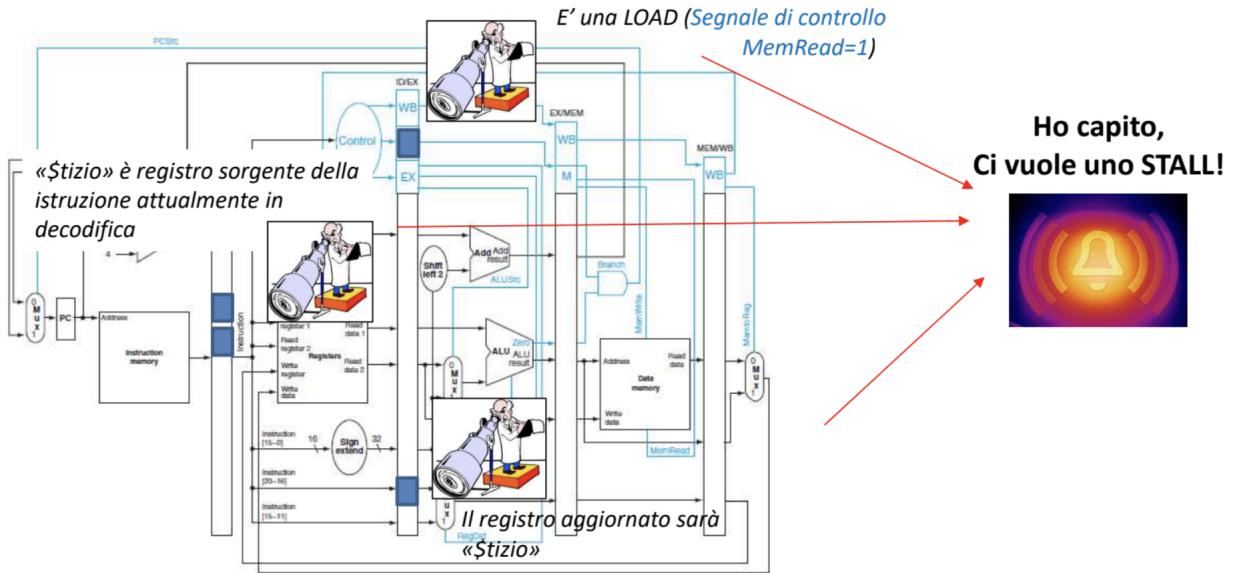


Figure 66: Hazard detectection unit

La hazard detection unit già nella fase di ID di un'istruzione, perché già da lì possiamo inserire uno stall tra la load (già in EXE) ed essa, che usa il risultato della load.

18.3 Dettaglio micro-architetturale

Il rilevatore di stall agirà nella fase di ID, perché può inserire una stall tra la load (già in EXE) e l'istruzione successiva che userà il suo risultato.

A livello micro-architetturale:

1. Quando la load è in EXE si manda un segnale di enable a 0 al PC e al registro IF/ID. Cioè l'istruzione successiva sarà la stessa istruzione di prima.
2. L'istruzione critica in fase di ID deve essere congelata. Questo si fa impedendo alla "barriera" IF/ID di mandare in avanti l'istruzione.
3. Il controllore globale manda tutti i segnali a 0, così facendo blocca l'istruzione critica in EXE.

19 Branch prediction statica

19.1 Branch-hazard

Solo alla fine di MEM so decidere quale sarà il nuovo PC.

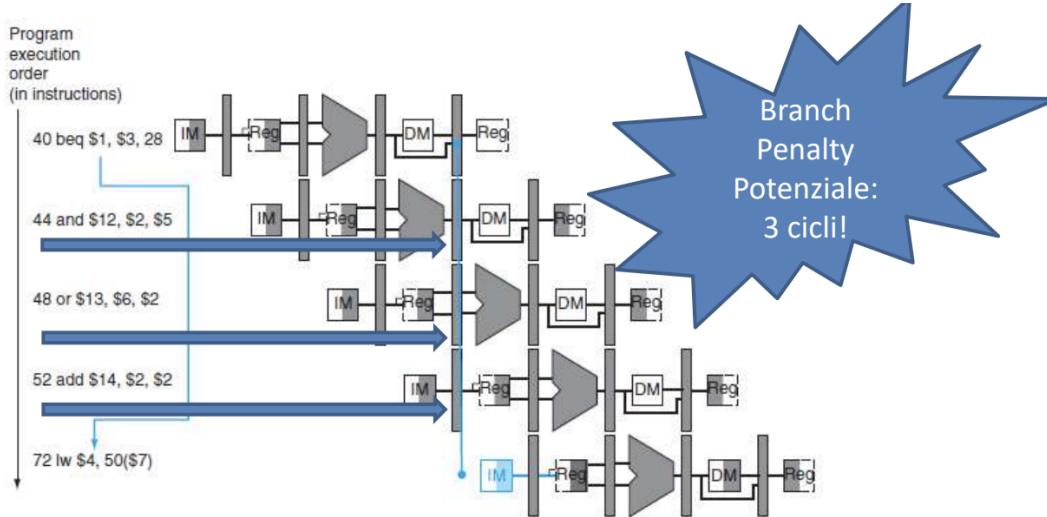


Figure 67: Branch hazard

La penalità aumenta in modo significativo con pipeline lunghe e con processori che fanno il fetch di più istruzioni contemporaneamente.

Terminologia:

- Branch target address: indirizzo a cui il branch salta.
- Branch taken: il controllo viene trasferito ad un indirizzo diverso (target) da quello dell'istruzione successiva.
- Branch not taken: viene eseguita l'istruzione successiva.
- Branch prediction accuracy: percentuale di branch accuratamente predetti.
- Fall-through path: percorso preso se il branch è NOT TAKEN.

La possibile soluzione a questo problema è cercare di fare una predizione di quello che il branch farà.

19.2 Branch prediction

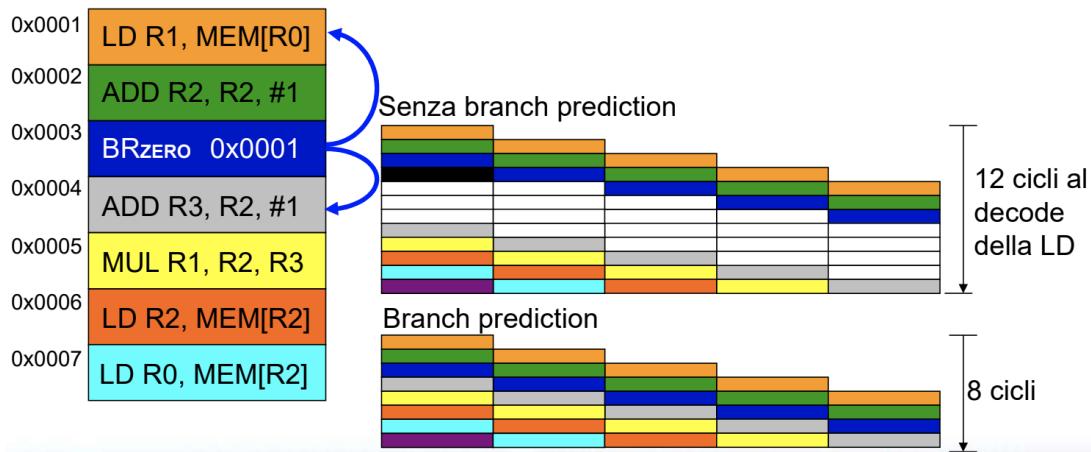


Figure 68: Branch prediction

Nel primo esempio la micro-architettura non è dotata di branch prediction, quindi dopo l'istruzione di branch si devono inserire delle stall, oppure eseguire delle istruzioni che non sono collegate al branch.

Nel secondo esempio la micro-architettura è dotata di branch prediction, quindi quando viene mandato il branch nella pipeline viene fatta una previsione. In questo caso la predizione di NOT-TAKEN è corretta.

19.2.1 Missprediction penalty

Nel caso in cui la predizione sia errata (lo si vede nello stadio di WB) si devono fare 2 operazioni:

- Eliminare tutte le operazioni in corso lungo la pipeline (in gergo si dice fare il flush della pipeline).
- Fatto il flush della pipeline si può fare il fetch delle istruzioni corrette.

19.2.2 Flush della pipeline

Rappresenta l'invalidazione di tutti gli stadi di pipeline che precedono la risoluzione di un branch.

Dobbiamo fare 2 assunzioni:

- Branch risolto nello stadio di decode.

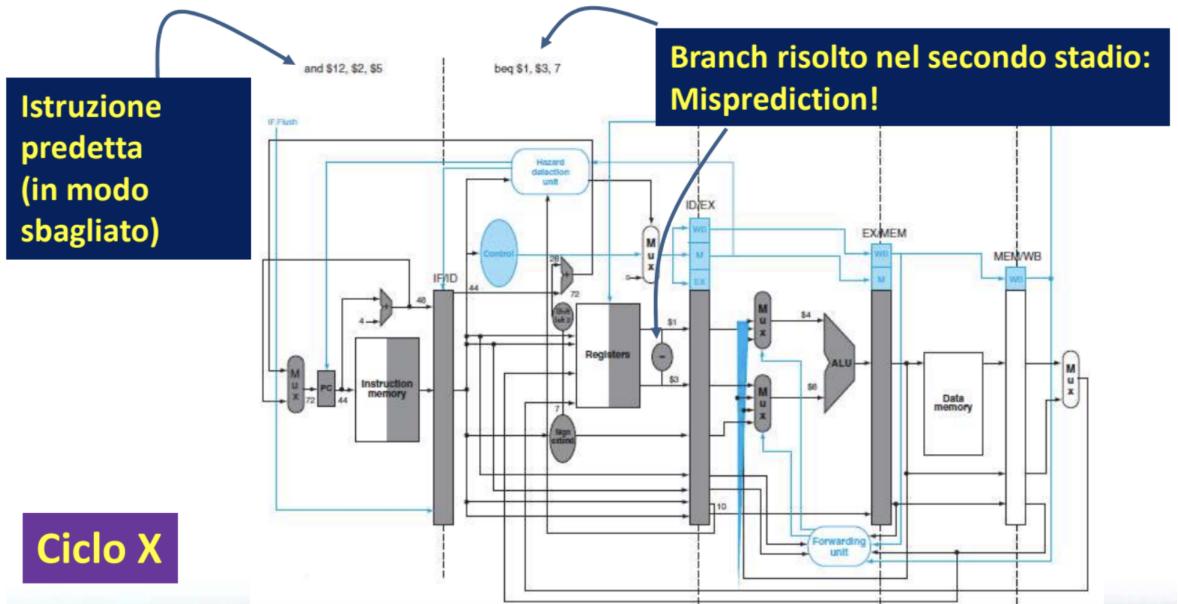


Figure 69: Flush della pipeline

- La missprediction penalty è solo di un ciclo (un'istruzione da invalidare).

Quando il branch si sposta da IF a ID fa la predizione. Quindi in IF si troverà la prossima istruzione determinata dalla predizione. Per verificare la condizione dobbiamo usare un comparatore di registri che si trova subito dopo il register-file.

Vediamo il caso in cui siamo in una missprediction:

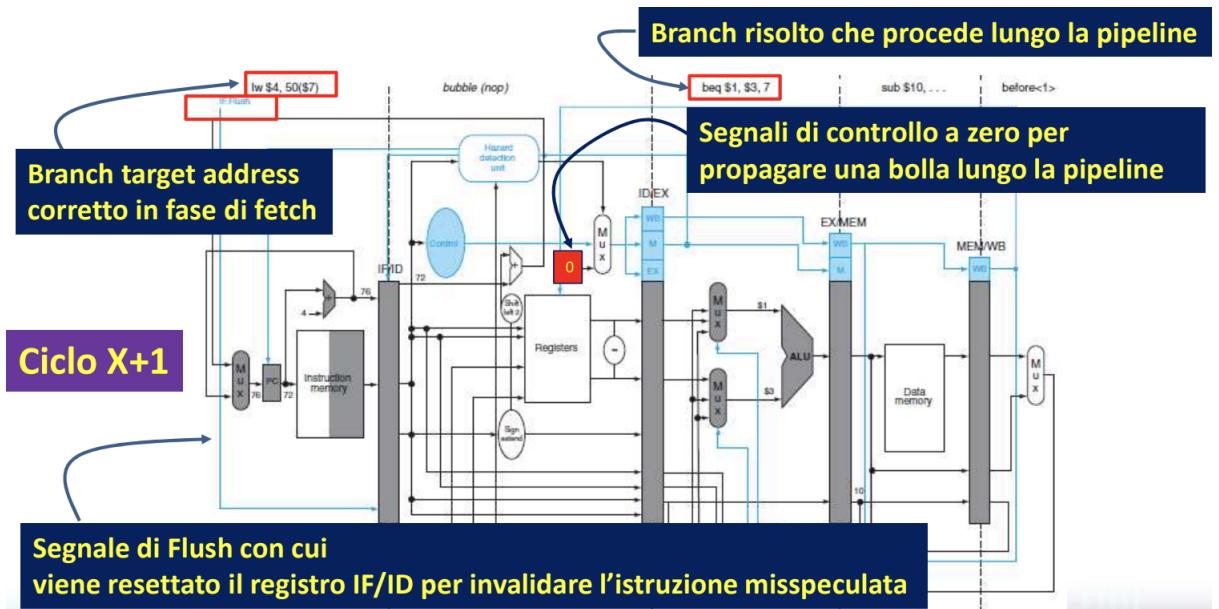


Figure 70: Flush della pipeline

Il branch avanza nello stadio di EXE, l'istruzione in ID viene trasformata in una stall e l'istruzione in IF sarà l'istruzione corretta.

Dal punto di vista dei segnali di controllo, un segnale di flush viene mandato al registro IF/ID che fa il reset dei registri (inserisce una stall). Successivamente questa stall viene propagata su tutta la pipeline.

19.3 Predizione branch target address

Assumiamo che la direzione (TAKEN/UNTAKEN) sia correttamente pre detta.

Il Target Address di un salto condizionato in caso di "branch taken" rimane lo stesso attraverso le sue invocazioni successive a tempo di esecuzione.

L'idea quindi è di memorizzare il target address del branch la prima volta che viene eseguito TAKEN, ed accedervi le volte successive tramite il PC se si predice nuovamente TAKEN.

Questa memoria di appoggio prende il nome di *branch target buffer* (BTB).

19.3.1 Stadio di fetch con branch target buffer

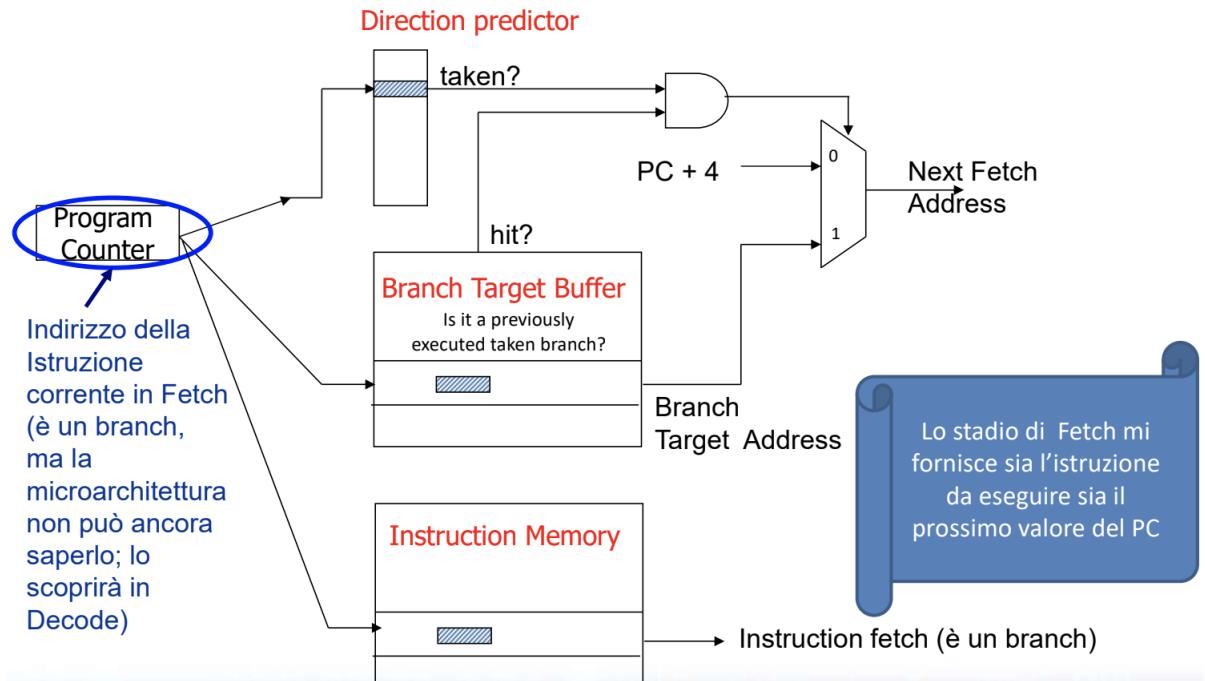


Figure 71: Stadio di fetch con BTB

Il PC punta alla instruction-memory e da lì viene fuori l'istruzione fetchata che poi andrà nello stadio di decode. Durante il fetch si predice che il branch sarà TAKEN e contemporaneamente accedo a alla branch-target-buffer. Se il branch è già stato eseguito ed è già risultato TAKEN in passato la branch-target-buffer avrà già allocato il branch-target-address. Quindi alla seconda invocazione del branch l'indirizzo sarà già pronto in memoria. Inoltre la BTB, tramite un segnale di hit/miss, andrà a dire se il branch è già stato registrato nel BTB; quindi se il branch non c'è nel BTB vuol dire che è la prima volta che viene eseguito il branch, oppure che il branch è sempre risultato NOT-TAKEN.

Si ha miss quando:

- L'istruzione non è un branch.
- L'istruzione è un branch ma non è mai stato TAKEN.
- Un altro branch condivide la stessa linea del BTB (*aliasing*).

19.4 Branch prediction statica

Per branch prediction statica si intende una predizione della direzione di un branch che è fissa per ogni esecuzione dello stesso branch. La predizione viene fatta al tempo di compilazione.

Le principali tecniche di branch prediction statica sono:

- Predizione "always UNTAKEN".
- Predizione "always TAKEN".
- Predizione BTFN (backward taken, forward UNTAKEN).
- Predizioni basate su "profiling".
- Predizioni basate sul programma o programmatore.
- Delayed branches.

19.4.1 Always UNTAKEN

È decisamente la tecnica di predizione più semplice. Consiste nel predire che la prossima istruzione della sequenza è la prossima istruzione che deve essere eseguita. Questo implica che ogni branch sarà sempre UNTAKEN. Successivamente si implementa il meccanismo di recovery da missprediction.

Non serve un branch target buffer perché non avremo mai bisogno del branch-target-address; essendo che l'istruzione prossima sarà sempre PC+4.

Il problema di questa tecnica è che non predice correttamente molti loop.

Per migliorare la probabilità di predizione accurata si agisce sul control flow graph, in modo che la prossima istruzione più probabile sia nel UNTAKEN path del branch.

19.4.2 Always TAKEN

Tecnica speculare del always UNTAKEN, non fornisce nessuna predizione, ma serve il BTB in quanto ci serve il branch-target-address.

Se un'applicazione ha molti loop allora conviene usare questa tecnica in quanto la maggior parte dei branch sono TAKEN.

19.4.3 BTFN

Questa tecnica mischia le prime due viste prima. In pratica se il branch punta in avanti (es: if/else) allora il branch è UNTAKEN, se invece punta all'indietro (for loop) allora il è TAKEN.

Questa tecnica quindi richiede il BTB in caso il compilatore abbia suggerito TAKEN.

19.4.4 Predizione su profiling del codice

Il compilatore determina la direzione più probabile per ogni branch utilizzando un'esecuzione di profilazione. Quindi capisce la prevalenza TAKEN o UNTAKEN di tutti i branch.

Questa prevalenza viene codificata attraverso un bits di suggerimento all'hardware nel formato istruzioni del branch.

Il vantaggio è che viene fatta una predizione che viene effettuata per ogni branch, quindi la predizione è più accurata. Gli svantaggi sono: la predizione è accurata solo se il data set di profiling è rappresentativo; l'accuratezza dipende dal comportamento dinamico del branch.

19.4.5 Predizione basata sul programma

L'idea è usare delle semplici euristiche basate sull'analisi del programma per determinare la direzione statica predetta. In breve il codice viene analizzato senza scegliere un data set rappresentativo.

Il vantaggio è che questa tecnica non richiede alcuna profilazione del codice, quindi nessun "run" di prova. Gli svantaggi sono: le euristiche potrebbero essere non rappresentative; richiede un'analisi da parte del compilatore.

19.4.6 Predizione basata sul programmatore

Il programmatore fornisce direttamente la direzione statica dei branch.

Lo fa attraverso delle "pragma" nel linguaggio di programmazione che qualificano un branch come TAKEN o UNTAKEN.

I vantaggi sono: non richiede una profilazione o analisi del programma; il programmatore conosce meglio alcuni branch rispetto a qualunque tecnica di analisi. Gli svantaggi sono: richiede supporto nell'ISA, nel linguaggio di programmazione e nel compilatore; grava in modo eccessivo sul programmatore.

Lo svantaggio comune di tutte queste tecniche è che non possono adattarsi a cambiamenti dinamici nel comportamento dei branch.

20 Branch prediction dinamica

L'idea della predizione dinamica è predire la direzione dei branch sulla base di informazioni dinamiche (raccolte a tempo di esecuzione).

Vantaggi:

- Predizione basata sulla storia dell'esecuzione dei branch e/o sul suo "conto".
- Capacità di adattamento della politica di predizione a variazioni dinamiche del comportamento dei branch.
- Non serve alcun profiling statico del codice.

Svantaggi:

- Maggiore complessità.

Tecniche di predizione dinamica:

- Last time prediction (singolo bit).
- Last time prediction with hysteresis (a due bits).
- Global Two-level prediction.
- Local Two-level prediction.
- Hybrid.

20.1 Last time predictor

Predice la direzione di un branch sulla base della sua ultima esecuzione.

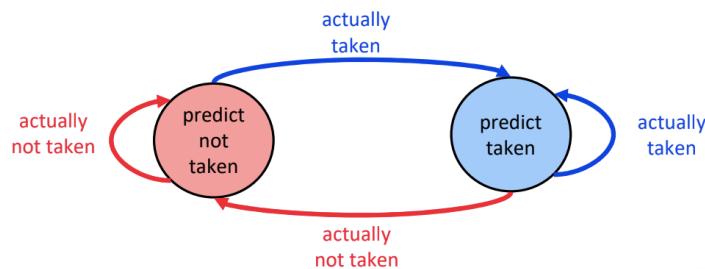


Figure 72: Macchina a stati di un last time predictor

Se un branch è risultato TAKEN la prima volta, allora anche le successive volte lo sarà; finché non si arriverà ad una missprediction e cambierà la sua predizione con UNTAKEN.

Questa tecnica richiede di memorizzare *n* bits di memoria per ogni branch nel BTB oppure in un buffer separato chiamato *branch prediction buffer*.

20.1.1 Branch prediction buffer

A differenza del BTB il BPB non ha il campo TAG. Il campo tag serve per capire se una entry del BTB è esattamente l'istruzione di branch che si sta valutando.

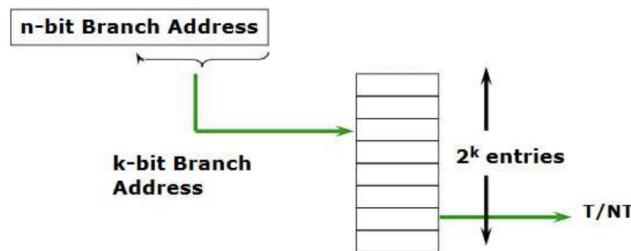


Figure 73: Branch prediction buffer

Si ha missprediction quando:

- La predizione per un certo branch non è corretta.
- La predizione non si riferisce a un branch ma ad un altro che condivide la stessa entry.

20.1.2 Problema del last time predictor

Un problema serio del last time predictor è che sbaglia sempre due volte la predizione di un loop (do-while) branch (alla prima iterazione e all'ultima iterazione).

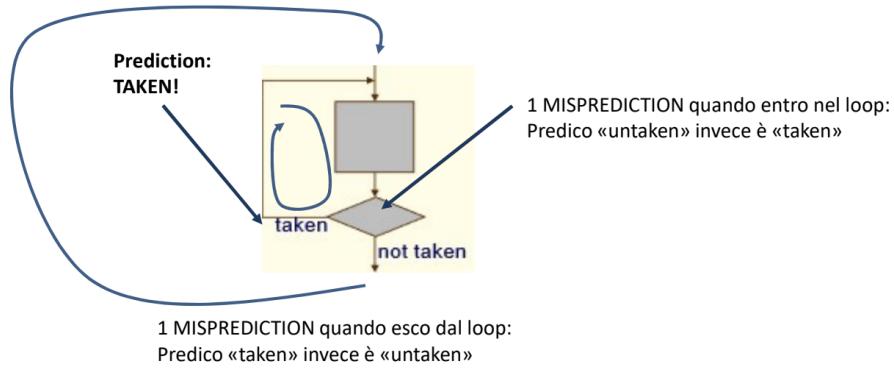


Figure 74: Problema last time predictor

20.2 Predittore a 2 bits

Il vero problema del last time predictor è che cambia il suo stato molto velocemente. Nel senso che se l'ultimo branch è stato TAKEN e quello nuovo è UNTAKEN cambia immediatamente la sua previsione sul prossimo.

L'idea per migliorare il last time predictor è quella di aggiungere dei "passaggi intermedi" di modo che non cambi subito il suo stato. Quindi invece di utilizziamo 1 bit ne utilizziamo 2 per tenere conto della storia delle previsioni per un branch.

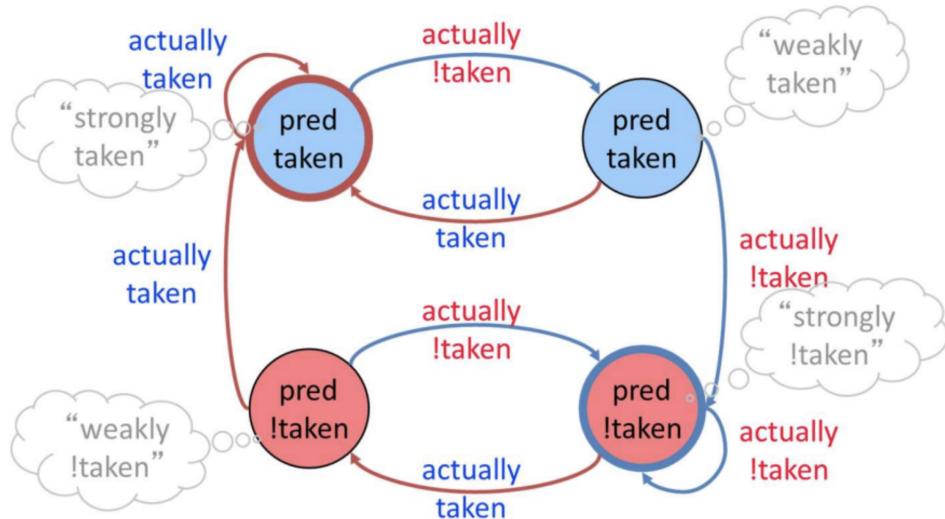


Figure 75: Predittore a 2 bits

Dal diagramma a stati finiti vediamo che la macchina inverte il suo stato dopo 2 missprediction consecutive.

Vantaggi:

- Maggiore accuratezza.

Svantaggi:

- Aumenta il costo hardware.

Il limite della predizione dinamica a n bits è che però non guardano alla storia globale degli altri branch, che si possono influenzare a vicenda.

20.2.1 Limiti della predizione dinamica a 2 bits

Il last time predictor a n bitss non guarda il "contesto" del branch, ma guarda la sua storia nelle ultime esecuzioni.

Contesto globale: si trova che un dato branch è influenzato dai branch che sono stati eseguiti di recente (*global branch correlation*).

Contesto locale: si trova che un dato branch è influenzato dalla sua storia di più lungo termine (*local branch correlation*).

In generale, si ottengono predizioni più accurate correlando la predizione di un branch al contesto (*correlating predictors*).

20.3 Global Two-level prediction

1 livello: consiste in un *global branch history register* un registro che tiene traccia degli ultimi n branch eseguiti.

2 livello: Una tabella 2-bits counter per ogni possibile pattern della storia globale che indica la direzione presa da un branch l'ultima volta che la storia è stata vista.

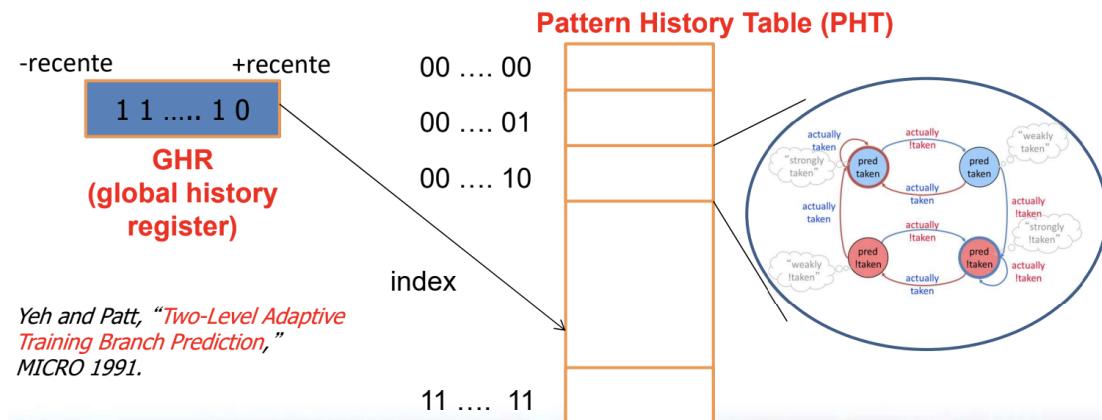


Figure 76: Global branch predictor

Quando il branch viene risolto, si aggiorna sia il GHR sia la entry nella PHT.

20.3.1 Implementazione

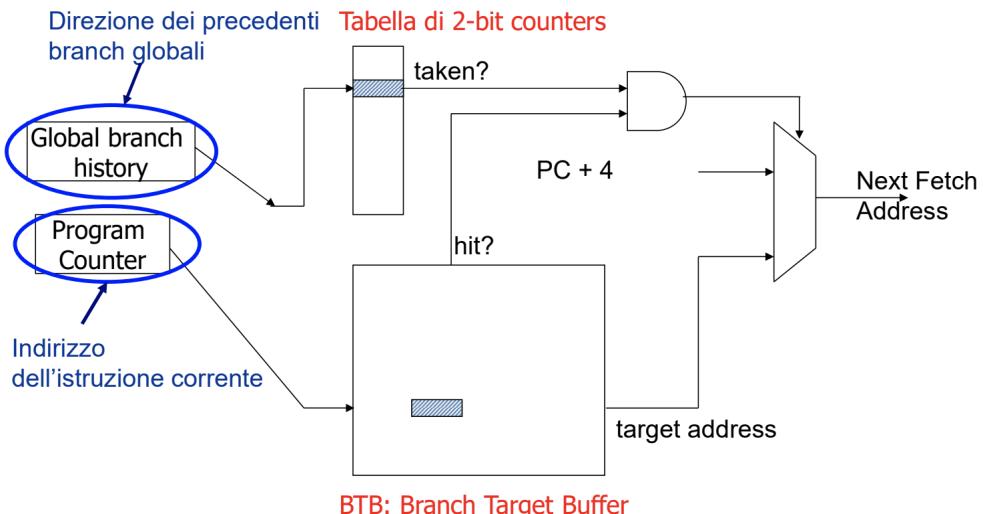


Figure 77: Implementazione global branch predictor

Implementiamo il global branch history nello stadio di IF.

Dal PC andiamo nella memoria istruzioni e contemporaneamente andiamo nel BTB per vedere se l'istruzione è un branch ed eventualmente il suo branch target address. Allo stesso tempo il global branch history indirizza una tabella di 2 bits counter, da cui ottengo la predizione TAKEN/UNTAKEN.

20.3.2 Miglioramento dei predittori globali

Si è osservato che non è vero che tutti i branch quando vedono la stessa storia globale si comportano allo stesso modo. Quindi occorre avere una maggiore visibilità del branch specifico che sto predicendo. Dal punto di vista micro-architetturale bisogna combinare il valore del GHR al valore del PC.

La soluzione più semplice è realizzare una combinazione mediante la porta logica XOR. Questo predittore prende il nome di *Gshare predictor*.

20.4 Local Two-level prediction

A volte sapere come si comporta di solito un branch aiuta a predirlo per il futuro (cioè la sua storia locale può essere più importante della storia globale). In particolare questa predizione è molto più efficace nei loop.

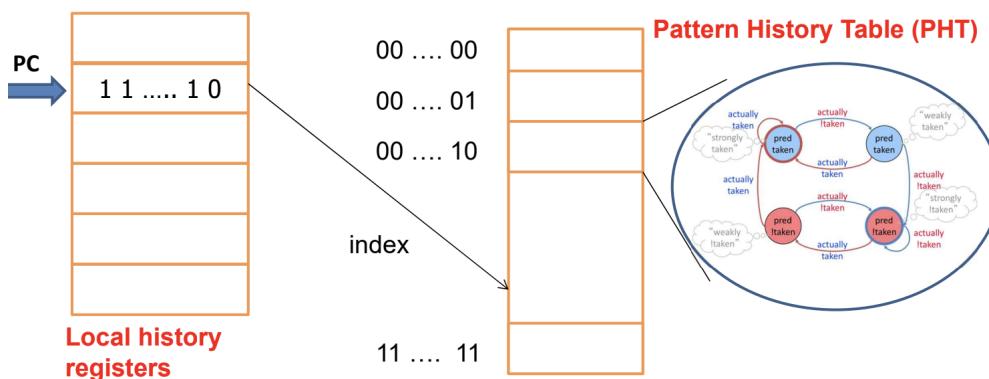


Figure 78: Local branch predictor

1 livello: un insieme di *local history registers* che riportano la storia di ogni branch. La selezione del registro giusto avviene tramite PC del branch.

2 livello: una tabella di 2-bits counters per ogni configurazione del local history register che indica la direzione del branch l'ultima volta che ha avuto una certa storia.

20.4.1 Implementazione

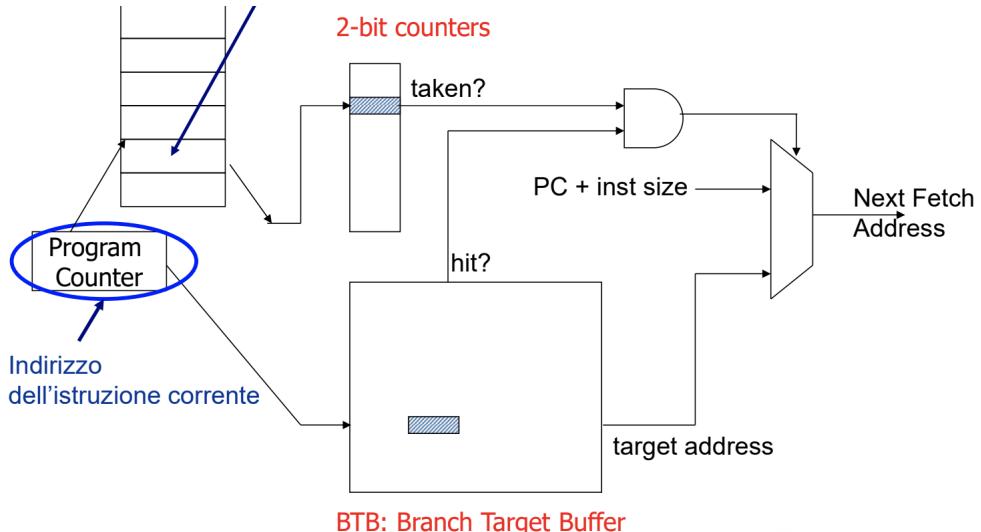


Figure 79: Implementazione local branch predictor

Il PC va ad indirizzare sia il BTB e sia un nuovo register file (local history registers) per la local branch prediction. Con il registro dentro la local history registers andremo a ricavarci un altro registro della PHT che ci dirà la predizione.

20.4.2 Problemi della local prediction

- La local history registers soffre di aliasing, cioè diversi branch confluiranno con la loro storia passata su un unico registro.
- Più di branch può puntare alla stessa locazione della PHT, quindi si sporca la predizione.

20.5 Hybrid branch predictors

A volte per certi branch è più utile guardare alla storia locale, mentre per altri alla storia globale. Dunque quello che fanno i processori più avanzati è avere entrambi i predittori. Quindi per ogni branch si seleziona l'algoritmo più appropriato.

21 Cache

La memoria dati e la memoria istruzioni non sono le vere memorie di massa, ma sono dispositivi intermedi.

Il problema è che il microprocessore vorrebbe memoria infinita con tempi di accesso molto veloci. Inutile dire che questo è irrealizzabile. Perché se una memoria è grande (megabytes, gigabytes) allora significa che sarà anche lenta; quando invece una memoria è piccola allora è anche veloce.

Il processore ha un comportamento particolare: tende a riutilizzare dati o istruzioni che sono stati utilizzati di recente, o vicini a quelli utilizzati di recente.

Parliamo di *località temporale* quando un processore che accede a un dato o istruzione, rifare l'accesso ad esso entro breve tempo.

Parliamo di *località spaziale* quando il processore dopo aver acceduto ad un elemento in memoria (dato o istruzione), entro breve tempo accederà agli elementi vicini.

21.1 Gerarchia di memoria

Si memorizzano su una memoria piccola, veloce e vicina al processore sia i dati/istruzioni acceduti più frequentemente sia i dati/istruzioni vicini (il working set).

Il working set può essere aggiornato dinamicamente attingendo dati/istruzioni dai livelli di gerarchia superiori.

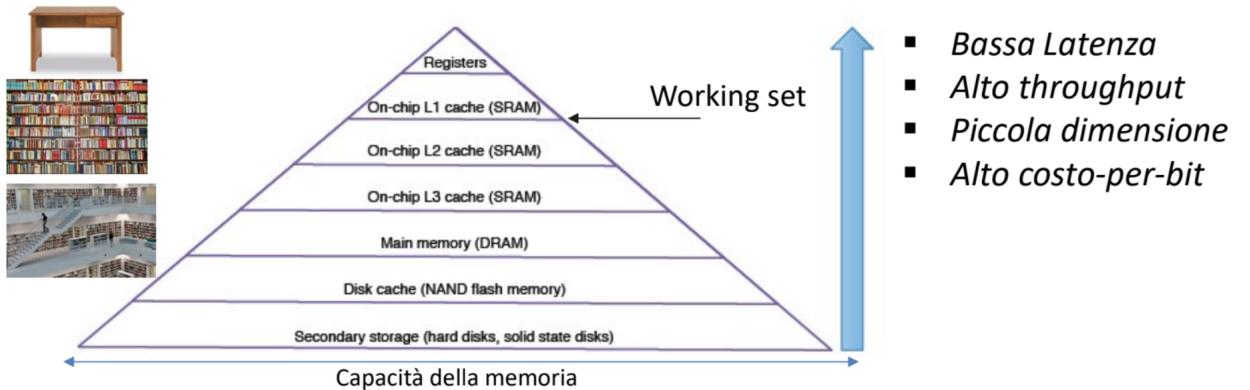


Figure 80: Gerarchia delle memorie

Implementando il sistema di memoria come una gerarchia, il microprocessore ha l'illusione di avere una memoria grande come l'ultimo livello, ma veloce come il primo.

21.2 Funzionamento della gerarchia hit e miss

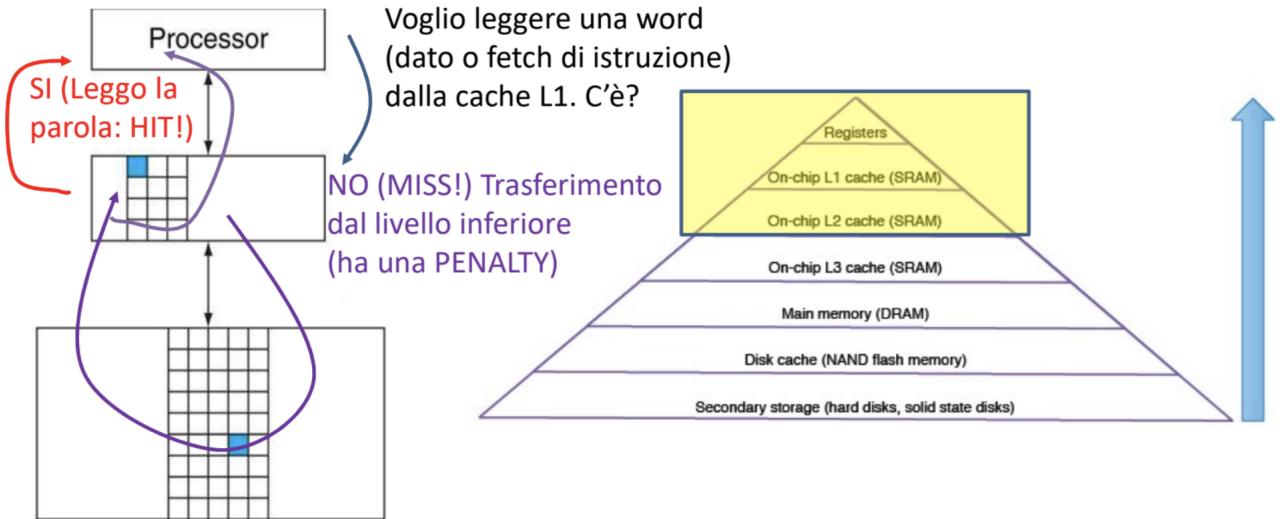


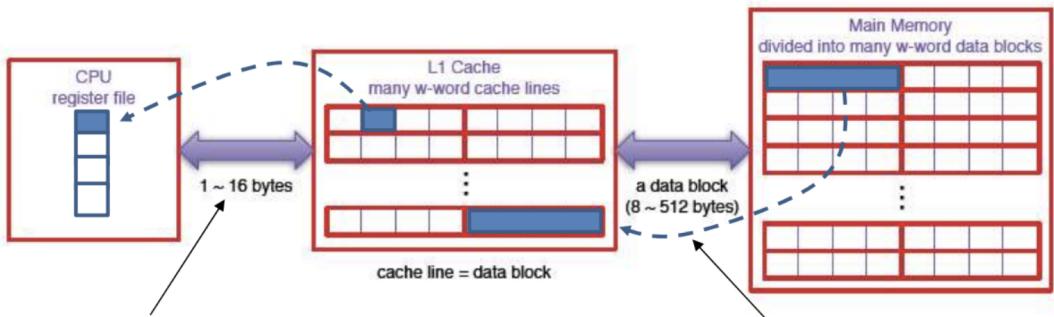
Figure 81: Gerarchia delle memorie

Quando il processore deve fare una load o store di una parola di memoria la va a cercare nel livello di gerarchia più prossimo, in questo caso nei registri e cache. Se il dato/istruzione è presente nella memoria in cui andiamo a cercarlo allora l'esito sarà *hit*, se invece non è presente occorre andarlo a cercare nelle cache ai livelli sottostanti della gerarchia, e quindi, per quanto riguarda la prima memoria si ha una *miss*.

La miss, per il processore, ha un costo. Nel senso che durante il tempo in cui andiamo a cercare il dato/istruzione nelle memorie sottostanti, il processore non sa cosa fare quindi produce degli stall.

21.3 L1 cache

La L1 cache è una memoria di primo livello, frapposta tra i registri del processore e la memoria principale (o i livelli di caching intermedi).



- La CPU vuol leggere una «word». La sua dimensione dipende dalla bitwidth dei registri
- «Refill» in L1 di una intera «linea» di cache o «blocco» di dati (diverse word)

Figure 82: L1 cache

Quando la CPU accede ad L1 mediante delle load/store, trasferiscono una certa quantità di dati che è pari al bit-with dei registri (per noi 32 bits). Quando però ci muoviamo dalla main memory alla L1 non si trasferisce solo il singolo dato/istruzione, ma un blocco di dati/istruzioni più grande.

Dunque questo blocco prende un nome e si chiama *linea di cache*. Quindi la main memory è logicamente suddivisa in linee di cache.

21.4 Indirizzamento della cache

La locazione del blocco in cache dipende dal suo indirizzo nella memoria principale (es: 0xFFA4568x). Da qui capiamo se il blocco è in cache.

Per trovare la posizione esatta del blocco usiamo diversi metodi. La soluzione più semplice è quella di una cache *direct mapped*.

21.5 Cache direct mapped

Assunzioni:

- Memoria principale divisa in 32 blocchi da n bytes.
- Per indirizzare un blocco in memoria ci vogliono 5 bit.
- Cache di dimensione 8 blocchi.
- Bastano 3 bit per selezionare Il blocco di cache.

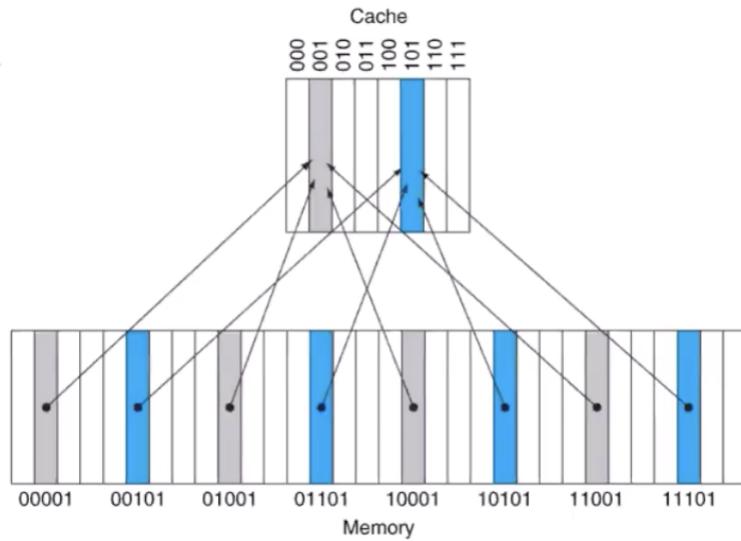


Figure 83: Direct mapping

Indirizzi di un blocco: 5 bits. 2 bits di TAG e 3 bits di INDEX.

In pratica, il blocco di cache dipende dagli ultimi 3 bit (INDEX). Risponde alla domanda: in quale blocco di cache devo cercare l'elemento?

I bit più significativi residui identificano univocamente il blocco (TAG). Risponde alla domanda: quale blocco di memoria, fra i tanti possibili che condividono una locazione di cache, è mappato su quel blocco di cache? Lo capisco confrontando il TAG.

21.5.1 Implementazione direct-mapped cache

Assumiamo una byte-addressable memory a 32 blocchi \rightarrow 8 bytes blocks \rightarrow 265 bytes.

- Load/store si riferiscono a parole di 4 bytes.
- Memoria principale da 256 bytes \rightarrow indirizzi da 8 bit.
- Blocchi di memoria da 8 byte (2 parole).
- Numero di blocchi in memoria: 32 blocchi ($256/8$).
- Ogni blocco è indirizzabile tramite 5 bits.
- Assumiamo una cache direct-mapped con 8 blocchi.

- INDEX: seleziona un blocco in cache (per indirizzare 8 blocchi ci vogliono 3 bits).
- Il TAG è dato dai rimanenti 2 bits.
- Rimangono 3 bits bit per indirizzare gli 8 byte di un blocco.

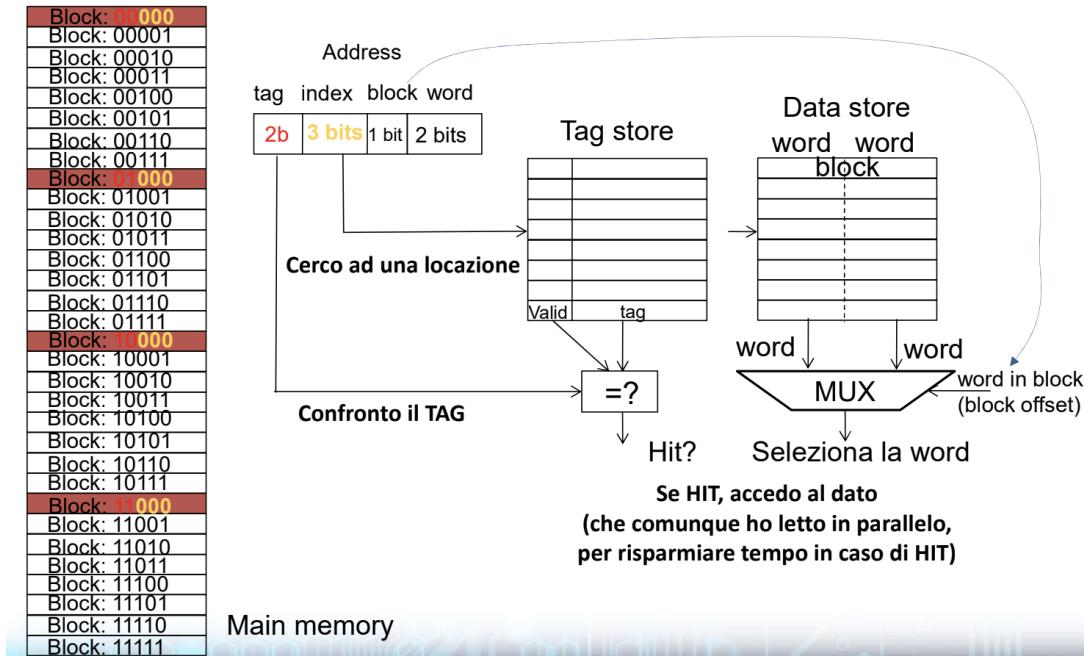


Figure 84: implementazione direct mapping

La cache è composta a sua volta da altre 2 memorie: tag store e data store.

Con INDEX si cerca una riga dentro il tag store. Contemporaneamente si legge anche la locazione del data store associata. Confronto tra il tag del tag store e il tag che fa parte dell'indirizzo che si sta cercando. Se i 2 tag coincidono allora sarà hit, se non coincidono allora è miss. Nel caso sia hit allora il multiplexer consente di selezionare il dato desiderato.

21.6 Tipi di cache misses

- **Cold miss:** quando la cache è vuota.
- **Conflict miss:** più blocchi di memoria condividono lo stesso blocco (o linea) di cache. Problema critico nelle cache direct mapped.
- **Capacity miss:** Quando il working set (insieme di blocchi attivi in un periodo di tempo) è più grande della dimensione della cache.

21.7 Set associative cache

Gli elementi possono essere in più locazioni (linee/blocchi) della cache. Ciò comporta: meno collisioni tra blocchi che condividono potenzialmente la stessa linea di cache rispetto al direct mapping; maggiore complessità nella ricerca e nel rimpiazzamento dei blocchi.

Aumentando il grado di associatività di una cache aumenta la complessità della stessa ma calano le conflict miss.

21.7.1 Implementazione set associative cache a 2 vie

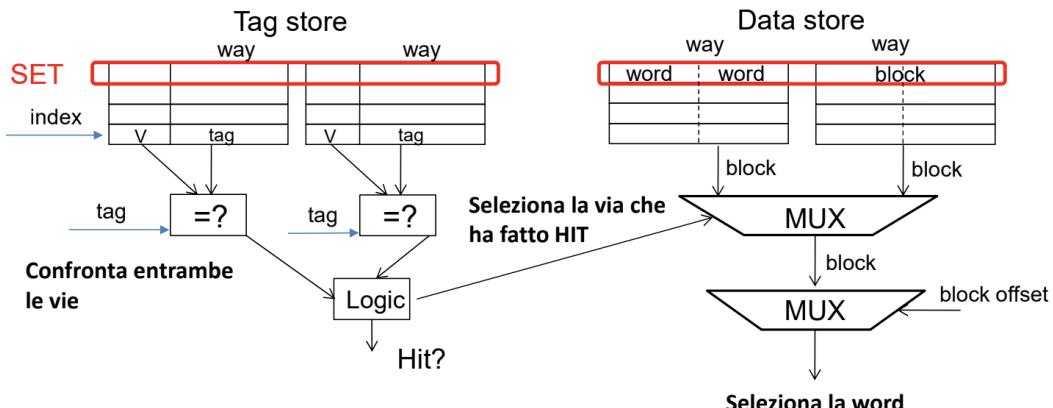
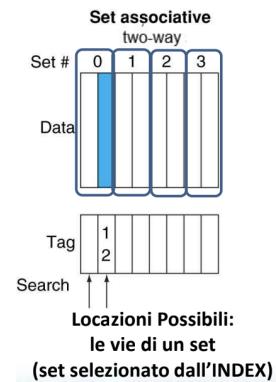


Figure 85: implementazione set associative a 2 vie

Indirizzamento: memoria principale 256 bytes → set a 2 vie → ogni blocco è 2 parole di byte.

Tag e data store sono a loro volta suddivise in 2 memorie più piccole. Dove ogni memoria rappresenta una via. La tag store cerca il tag in entrambe le vie e ci sarà una logica combinatoria che dirà se è hit o miss. Nel caso sia hit bisogna dire dov'è che si è fatto hit, quindi c'è bisogno di un multiplexer che serve per selezionare una delle 2 vie. Successivamente il secondo multiplexer estrae solo la quantità desiderata in una linea di cache.

21.8 Fully associative cache

La fully associative cache è un tipo di set associative cache dove gli elementi possono essere cercati in tutte le locazioni della cache.

In questo tipo di cache non esiste un campo INDEX, quindi abbiamo un campo TAG da bits.

21.8.1 Implementazione fully associative cache

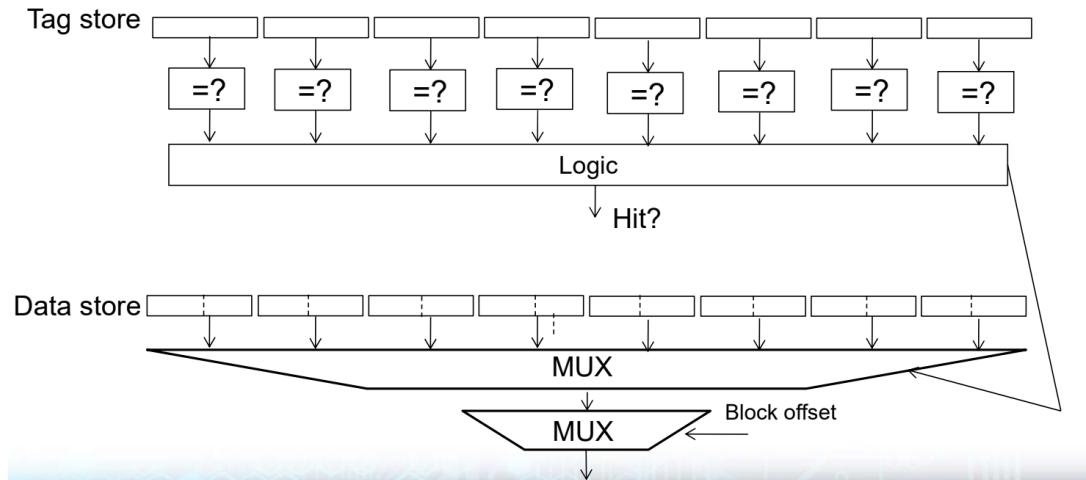


Figure 86: implementazione fully associative cache

21.9 Scrittura in cache

Per le scritture in cache la modifica del blocco non può iniziare finché non sia finito il confronto del tag.

Ci sono diverse opzioni sia per il write hit sia per il write miss.

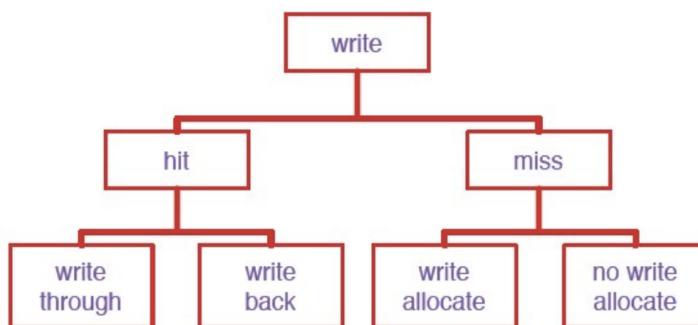


Figure 87: Scritture in cache

21.9.1 Opzioni write hit

- **Write through:** si modifica sia il blocco in cache sia il blocco nel livello di memoria superiore nel livello di memoria inferiore.

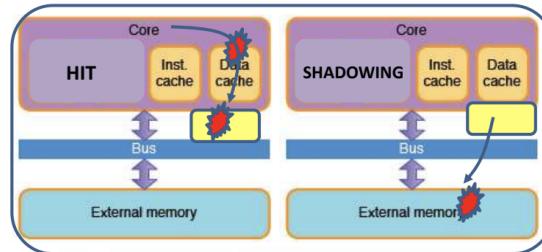


Figure 88: Write Through

- **Write back:** si modifica solo il blocco in cache.

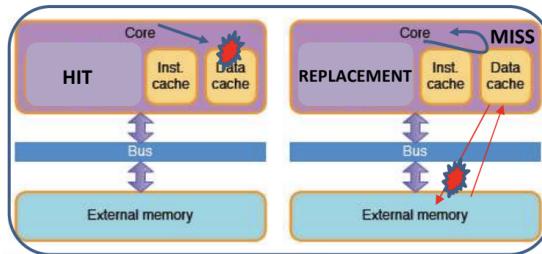


Figure 89: Write back

21.9.2 Opzioni write miss

- **Write allocate:** il blocco è caricato in cache durante una write miss.

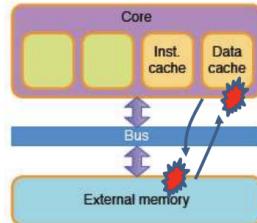


Figure 90: Write Through

- **No write allocate:** il blocco è modificato nella memoria di livello superiore e non caricato in cache

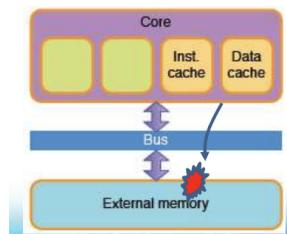


Figure 91: Write back