
ARCHITETTURA DEGLI ELABORATORI

LAUREA TRIENNALE IN SCIENZE INFORMATICHE

Magliani Andrea
Perego Luca

Università degli studi di Milano-Bicocca

A.A. 2022/2023

INDICE

Sistemi numerici.....	5
1.1 Calcolatori.....	5
1.2 Bit.....	5
1.3 Rappresentazioni.....	5
1.4 Sistemi numerici.....	5
1.5 Rappresentazione posizionale.....	6
1.6 Byte.....	6
1.7 Sistema esadecimale.....	6
Rappresentazione di Naturali.....	7
2.1 Conversioni da n_r a n_{10}	7
2.2 Conversione da n_{10} a n_r	7
2.3 Naturali in un sistema limitato.....	7
Operazioni su Naturali.....	8
3.1 Somma.....	8
3.2 Sottrazione.....	8
Rappresentazione di Interi.....	9
4.1 Metodo del Modulo e Segno [MS].....	9
4.2 Complemento a 1 [CA1].....	9
4.3 Complemento a 2 [CA2].....	10
Operazioni su Interi.....	11
5.1 Conversione da CA2 a n_{10}	11
5.2 Operazioni in MS.....	11
5.3 Operazioni in CA2.....	11
5.4 Shift.....	12
Rappresentazione in Eccesso 2^{n-1}.....	13
6.1 Struttura.....	13
6.2 Regola Pratica.....	13
Rappresentazioni di reali.....	14
7.1 Underflow.....	14
7.2 Metodi di rappresentazione.....	14
7.3 Virgola fissa.....	14
7.4 Virgola mobile.....	15
7.5 Standard IEEE 754.....	16
Circuiti Digitali.....	17
8.1 Introduzione.....	17
8.2 Classificazione.....	17

Logica Combinatoria	18
9.1 Porte Logiche	18
9.2 Componenti elettronico-logici	19
9.3 Logica a 2 livelli	19
Arithmetic Logic Unit	20
10.1 Somma su ALU ad 1 bit	20
10.2 Input "operation"	21
10.3 Sottrazione su ALU ad 1 bit	21
10.4 NOR e NAND su ALU ad 1 bit	22
10.5 ALU a 32 bit - Ripple Carry	22
10.6 Set Less Than su ALU a 32 bit	23
10.6 Branch On Equal su ALU a 32 bit	23
Circuiti Sequenziali	24
11.3 Introduzione	24
11.2 Latch	24
11.3 Clock	25
11.4 D-Latch	25
Metodi di Timing	26
12.1 D Flip-Flop	26
Register File	27
13.1 Struttura & Componenti	27
13.2 Lettura	27
13.3 Scrittura	27
Memorie	28
14.1 Parametri	28
14.2 Tipi di Memorie	28
14.3 Calcoli sulla memoria	28
Macchine a Stati Finiti	29
15.1 Utilizzo	29
15.2 Tipi	29
Filosofia di Progettazione della CPU	30
16.1 Tipi principali	30
16.2 MIPS	30
16.3 R-Type	31
16.4 I-Type	31
16.5 J-Type	31
Catena Programmatica	32
17.1 Assembly	32
17.2 Debugger	32
17.3 Compiler	32
17.4 Assembler	33
17.5 Linker	33
16.6 Loader	33

Datapath.....	34
18.1 Realizzazione di un Datapath.....	34
18.3 Fetch.....	34
18.4 Decode.....	34
18.5 Execute.....	34
Datapath Multiciclo.....	35
19.1 Vantaggi e Svantaggi.....	35
19.2 Struttura delle istruzioni.....	36
Eccezioni e Interruzioni.....	37
20.1 Definizione.....	37
20.2 Esecuzione delle istruzioni.....	37
20.3 Gestione di eccezioni ed interrupt.....	38
20.4 Identificazione dell'errore.....	38
Eccezioni in MIPS.....	39
21.1 Metodo Utilizzato.....	39
21.2 Passi da eseguire.....	39
21.3 Gestione delle Eccezioni nel Datapath.....	39
Prestazioni.....	40
22.1 Banda passante.....	40
22.2 Latenza.....	40
Input/Output.....	41
23.1 Definizione.....	41
23.2 Tipi di Bus.....	41
23.3 Periferiche.....	41
23.4 I/O gestito da programma.....	42
23.5 I/O guidato da interrupt.....	42
23.6 DMA - Accesso diretto alla memoria.....	43
Pipeline.....	44
24.1 Introduzione.....	44
24.2 Suddivisione nel datapath.....	44
24.3 Simulazione di esecuzione in pipeline.....	45
Hazard.....	46
25.1 Definizione.....	46
25.2 Structural Hazard.....	46
25.3 Data Hazard.....	46
25.4 Control Hazard.....	47
Gerarchie di Memoria.....	48
26.1 Introduzione.....	48
26.2 Tipi di memoria.....	48
26.3 Principio di Località.....	49
26.4 Definizioni.....	49

Cache	50
27.1 Introduzione	50
27.2 Tecniche di indirizzamento	50
27.3 Direct Mapped Cache	51
27.4 Fully Associative Cache	52
27.5 Set Associative Cache	52
27.6 Tecniche di Sostituzione	53
27.8 Tecniche di Aggiornamento	53
27.9 Write-through	54
27.10 Write-back	54
27.11 Write Buffer	54
27.12 Write Miss	55

Sistemi numerici

1.1 Calcolatori

Utilizzano il sistema binario (0/1).

Per evitare problemi di lettura viene utilizzato uno **standard di codifica**.

1.2 Bit

E' definito come **unità di misura dell'informazione**.

Combinando più bit possiamo ottenere strutture più complesse:

1. **Byte** → 8 bit
2. **Nibble** → 4 bit
3. **Word** → 32 bit
4. **Halfword** → 16 bit
5. **Doubleword** → 64 bit

Dati **k bit** posso ottenere 2^k **combinazioni**.

1.3 Rappresentazioni

Modo per descrivere un'entità. Nei sistemi numerici abbiamo:

1. Entità = valore

2. Rappresentazione

Ex. $16_{10} \rightarrow$ Rappresentazione decimale (10), entità (16).

1.4 Sistemi numerici

Possono essere **posizionali** o **non posizionali**.

Sistema decimale → posizionale.

Sistema romano → non posizionale.

1.5 Rappresentazione posizionale

$$N = \sum_{i=-m}^{n-1} d_i \cdot v^i$$

1. d (digit)
2. r (base del sistema)
3. n (numero della parte intera)
4. m (numero parte frazionaria)

$$\text{Ex. } 123,45 = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0 + 4 \cdot 10^{-1} + 5 \cdot 10^{-2}$$

1.6 Byte

MSB (most significant bit): bit più a sinistra.

LSB (less significant bit): bit più a destra.

1.7 Sistema esadecimale

Sistema in base 16.

Base $r = 16$, cifre $d = [0, \dots, F]$.

$$\text{Ex. } A1_{16} = A \cdot 16^1 + 1 \cdot 16^0 = 160_{10}$$

Spesso si usa il **pedice** (h) al posto di (16) per indicare la base, oppure 0x davanti al numero.

Rappresentazione di Naturali

2.1 Conversioni da n_r a n_{10}

La conversione da una qualsiasi n_r a n_{10} avviene nel seguente modo:

$$N = \sum_{i=-m}^{n-1} d_i \cdot r^i$$

d = cifra

r = base del sistema

n = numero della parte intera

m = numero della parte frazionaria

La conversione da n_{10} a n_r avviene come segue:

Ripeti $\frac{n}{r}$ finché $n = 0$, il resto di ogni quoziente comporrà n_r

2.2 Conversione da n_p a n_k

La conversione da n_p a n_k avviene aggiungendo uno step intermedio, il numero viene prima convertito n_p in n_{10} e solo dopo alla n_k .

2.3 Naturali in un sistema limitato

In qualsiasi rappresentazione reale, il numero di cifre disponibili è limitato. Si ha **overflow** quando la macchina non può rappresentare un risultato a causa del numero di cifre insufficienti.

Conversioni di byte:

- $2^{10} = 1024 = \text{Kilobyte}$
- $2^{20} = 1048576 = \text{Megabyte}$
- $2^{30} = 1073741824 = \text{Gigabyte}$

Operazioni su Naturali

3.1 Somma

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$$1 + 1 = 0 \rightarrow \text{carry [1 bit]}$$

$$1 + 1 + 1 = 1 \rightarrow \text{carry [1 bit]}$$

$$010011 +$$

$$010001 =$$

$$\text{-----}$$

$$100100$$

Il riporto viene detto **carry**.

3.2 Sottrazione

$$0 - 0 = 0$$

$$0 - 1 = 0 \rightarrow \text{borrow [1 bit]}$$

$$1 - 0 = 1$$

$$1 - 1 = 0$$

$$1 - 1 - 1 = 0 \rightarrow \text{borrow [1 bit]}$$

$$11101 -$$

$$11110 =$$

$$\text{-----}$$

$$01111$$

Il prestito è detto **borrow**.

Rappresentazione di Interi

4.1 Metodo del Modulo e Segno [MS]

Per la rappresentazione di valori interi si utilizza il metodo del **Modulo e Segno** (MS).

Questo sistema rappresenta il valore assoluto di un numero e utilizza il MSB come segno, 0 se positivo, 1 se negativo.

$$10011_2 \rightarrow -3_{10}$$

Con il metodo del modulo e segno, con n bit a disposizione è possibile rappresentare $2^{n-1} - 1$ numeri negativi e $2^{n-1} - 1$ numeri positivi.

Il metodo MS rappresenta 2 volte lo 0 $[-0, +0]$.

4.2 Complemento a 1 [CA1]

Complemento: operazione che inverte il valore di ogni bit.

$$\overline{1001} = 0110$$

Nel complemento a 1, il MSB rappresenta il segno:

- Se il numero è positivo [MSB = 0], rimane invariato;
- Se il numero è negativo [MSB = 1], si effettua l'operazione di complemento, ottenendo il valore assoluto dell'originale;

Come il metodo MS, anche CA1 rappresenta 2 volte lo 0.

4.3 Complemento a 2 [CA2]

Il CA2 è utilizzato per risolvere il problema del doppio 0.

- Se il numero è positivo [MSB = 0], rimane invariato;
- Se il numero è negativo [MSB = 1], si effettua il processo del CA1 e si somma 1;

Esistono **3 metodi** per il calcolo dei negativi in CA2:

- $CA2(x) = 2^n - x$;
- $CA2(x) = CA1(x) + 1$;
- **Regola pratica:** trascrivere tutti gli 0 e il primo 1 e fare il CA1 di tutti i bit rimanenti;

Operazioni su Interi

5.1 Conversione da CA2 a n_{10}

- Se il numero è positivo [MSB = 0], rimane invariato;
- Se il numero è negativo [MSB = 1], si applica l'operazione di CA2, si converte come codice binario puro per poi aggiungere il "-";

5.2 Operazioni in MS

Nelle **operazioni in MS**, si utilizza il seguente metodo:

- Se i segni sono concordi, il risultato rimane invariato;
- Se i segni sono discordi, il segno del risultato sarà uguale a quello del numero con valore assoluto maggiore;

5.3 Operazioni in CA2

Per eseguire **operazioni in CA2**, si utilizza il seguente metodo:

- Si esegue la somma su tutti gli addendi, segno compreso;
- Si scarta un eventuale carry sul segno;
- Se i segni sono **concordi** bisogna verificare l'overflow;

La sottrazione in CA2 viene convertita in somma.

5.4 Shift

L'operazione shift consiste nello spostare verso destra o sinistra la posizione delle cifre di un numero, inserendo uno 0 nella posizione lasciata libera.

Left Shift: equivale a moltiplicare il numero per la base;

- In MS si esegue regolarmente;
- In CA2 se il segno cambia, indica l'avvenimento dell'overflow;

Right Shift: equivale a dividere il numero per la base;

L'operazione Left può andare in overflow, aggiungendo un bit alla destra del numero, mentre l'operazione right causa overflow, in quanto scarta il LSB.

Rappresentazione in Eccesso 2^{n-1}

6.1 Struttura

Nella rappresentazione **Eccesso** 2^{n-1} un numero x è rappresentato come:
 $x + 2^{n-1}$

Con n bit si rappresenta l'eccesso 2^{n-1} $[-2^{n-1}, 2^{n-1} - 1]$

6.2 Regola Pratica

I numeri in eccesso si ottengono cambiando il MSB di CA2

ex:

$$\begin{array}{ll} x = 5 & 5 + 128 = 133 \rightarrow 10000101_2 \\ x = -3 & -3 + 128 = 125 \rightarrow 01111001_2 \end{array}$$

La rappresentazione eccesso **inverte il segno**:

- positivo = 1;
- negativo = 0;

Rappresentazioni di reali

7.1 Underflow

Nella rappresentazione di numeri reali, può comparire il fenomeno dell'**underflow**: La parte frazionaria ha troppi pochi bit a disposizione per rappresentare un valore troppo piccolo.

7.2 Metodi di rappresentazione

1. Fixed point [Virgola fissa]
2. Floating point [Virgola mobile]

Un'associazione è una linea che collega le classi coinvolte.

7.3 Virgola fissa

La rappresentazione in **virgola fissa** si riserva un numero di bit alla parte intera e alla parte frazionaria.

La parte della virgola è implicita e uguale per tutti i numeri rappresentati.

Rappresentazione unsigned: Dati n bit, vengono dedicati i bit alla parte intera e d bit alla parte frazionaria, tale che $i + d = n$.

Tramite la codifica unsigned è possibile rappresentare solo i positivi.

L'intervallo di rappresentazione è $[0, 2^i - 1]$ per la parte intera e $[0, 2^d - 1]$ per la parte frazionaria.

Rappresentazione signed: Dati n bit, vengono dedicati **n bit**, viene dedicato **1 bit** al segno, **$i-1$ bit** alla parte intera e **d bit** alla parte frazionaria, tale che

$$i + d + 1 = n.$$

L'intervallo di rappresentazione è:

1. $[-2^{i-2} - 1, 2^{i-2} - 1]$ per la parte intera.
2. $[0, 2^d - 1]$ per la parte frazionaria.

Tramite la rappresentazione in virgola fissa, è possibile scegliere se dedicare più o meno bit alla parte intera e viceversa alla parte frazionaria.

ATTENZIONE:

1. Aumentando i bit della parte intera aumenta il rischio di underflow.
2. Aumentando i bit della parte frazionaria aumenta il rischio di overflow.

Ex. Virgola fissa: $12,5_{(10)} = 1100,0101_{(2)}$

7.4 Virgola mobile

Nasce per sopperire ai problemi di arrotondamento ed inefficienza del sistema in virgola fissa.

La notazione in virgola mobile divide i bit a disposizione in 4 componenti:

1. Segno = [s]
2. Mantissa = [m]
3. Base = [b]
4. Esponente = [e]

Esistono due forme di virgola mobile:

1. **Non formalizzata:** Ammette un valore qualsiasi per la mantissa
2. **Formalizzata:** Dedica solo 1 cifra intera alla mantissa

Utilizzando la notazione scientifica per la base 2:

1. **Precisione singola (32 bit):**
 - a. 1 bit segno
 - b. 8 bit esponente
 - c. 23 bit mantissa
2. **Precisione doppia (64 bit):**
 - a. 1 bit segno
 - b. 11 bit esponente
 - c. 52 bit mantissa

Il **bit iniziale (parte intera della mantissa)** in virgola mobile è sempre uguale a 1 (se normalizzato), è detto "bit nascosto".

La formula standard è: $(-1)^s \cdot (1 + 0.m) \cdot 2^e$

7.5 Standard IEEE 754

E' lo standard internazionale per la rappresentazione dell'aritmetica frazionaria adottato dal 1989.

Lo standard è **non proprietario**, ovvero non dipendente dall'architettura dell'elaboratore.

Rappresenta l'esponente in eccesso 127, con intervallo $[-127, 128]$.

Le configurazioni estreme sono riservate

ATTENZIONE:

Rappresentare un numero in virgola mobile causa un errore di approssimazione.

La virgola mobile rappresenta solo le n cifre più significative.

1. Errore assoluto: La grandezza dell'errore dipende dal numero di cifre significative e dall'ordine di grandezza del numero
2. Errore relativo: La grandezza dell'errore dipende dalla differenza rispetto al risultato corretto

Ascii standard: La codifica ascii standard contiene 128 [7 bit]

"caratteri", questi sono:

1. Lettere maiuscole [65-90]
2. Lettere minuscole [97-122]
3. Le cifre [48-57]
4. Caratteri di controllo [0-31]
5. Punteggiatura [spazi rimasti]

Ascii esteso: La codifica ascii estesa contiene 256 [8 bit] "caratteri".

Unicode: Dedica una quantità variabile di bit al carattere in base alla versione della codifica [da 8 a 32 bit].

UTF-8: è una codifica a lunghezza variabile [da 8 a 16 bit].

Circuiti Digitali

8.1 Introduzione

I **circuiti logici** sono realizzati come circuiti integrati su chip di silicio.

Porte e connessioni sono depositate sul chip, inserite in un package e collegate all'esterno tramite **pin**.

Nell'elettronica, input e output possono assumere solo 2 valori: **segnale alto** [1] oppure **segnale basso** [0].

8.2 Classificazione

I circuiti possono essere distinti per grado di integrazione:

- **Small Scale Integrated**: 1-10 porte;
- **Medium Scale Integrated**: 10-100 porte;
- **Large Scale Integrated**: 100-100.000 porte;
- **Very Large Scale Integrated**: >100.000 porte;

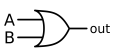
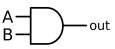
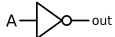
Logica Combinatoria

Un circuito è detto **combinatorio** se le sue uscite dipendono solamente dalla funzione logica applicata allo stato delle sue entrate in tempo reale.

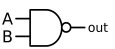
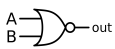
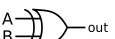
9.1 Porte Logiche

Le **porte logiche** sono componenti elettrici che permettono di eseguire operazioni logiche primitive.

Porte Logiche Fondamentali:

- OR: 
- AND: 
- NOT: 

Porte Logiche Derivate:

- NAND: 
- NOR: 
- XOR: 

Le porte NAND e NOR, svolgono la funzione di **inverter** e sono dette universali.

Le porte logiche possono essere disposte **a cascata** per permettere più ingressi ad un'operazione che normalmente sarebbe binaria.

9.2 Componenti elettronico-logici

Decoder:

Il **decoder** è un componente elettronico caratterizzato da n ingressi e 2^n uscite. Il suo scopo è impostare lo stato alto le uscite corrispondenti alla conversione in base 10 dei codici binari ricevuti in input.

Multiplexor:

Il **multiplexor**, o selettore, è un componente elettronico caratterizzato da 2^n ingressi principali, n ingressi di controllo e 1 uscita. Il valore del selettore determina quale degli input verrà trasmesso come output.

Se un multiplexor riceve n segnali, necessita $\log_2(n)$ selettori che consistono in:

- Un decoder che genera n segnali;
- Un array di n porte logiche AND;
- Una porta logica OR;

9.3 Logica a 2 livelli

Tramite le porte logiche fondamentali è possibile realizzare funzioni complesse. Si possono creare logiche a due livelli:

-**Somma di prodotti**: somma logica (OR) di prodotti (AND)

-**Prodotto di somme**: prodotto (AND) di somme (OR)

La somma di prodotti è nota come **Programmable Logic Array** (PLA). Questa è composta da:

- n input;
- I corrispondenti n input complementati;
- una logica a 2 stage:
 - un array di porte logiche AND;
 - un array di porte logiche OR;

La gran parte delle operazioni avvengono a 32 bit, svelando la necessità di creare **array di elementi logici**.

Un **Bus** è una collezione di linee di input trattati come singolo segnale.

Arithmetic Logic Unit

L'**Arithmetic Logic Unit**, o ALU è la componente del processore che svolge le operazioni aritmetico-logiche.

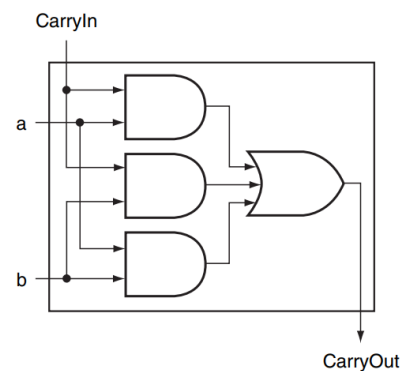
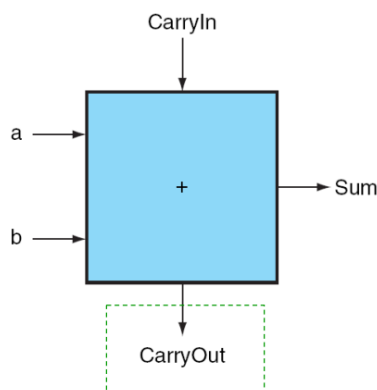
L'ALU è un insieme di circuiti combinatori:

- Operazioni aritmetiche;
- Operazioni logiche;

Gli **elementi fondamentali** della ALU sono:

- AND Gate;
- OR Gate;
- Inverter (NOT Gate);
- Multiplexor;

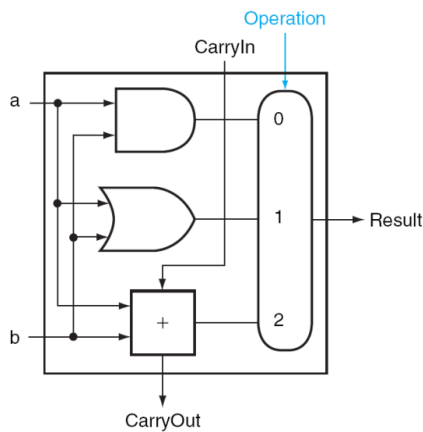
10.1 Somma su ALU ad 1 bit



Il risultato di una somma può essere scritto come:

$$CarryOut = (b \cdot CarryIn) + (a \cdot CarryIn) + (a \cdot b) + (a \cdot b \cdot CarryIn)$$

10.2 Input "operation"

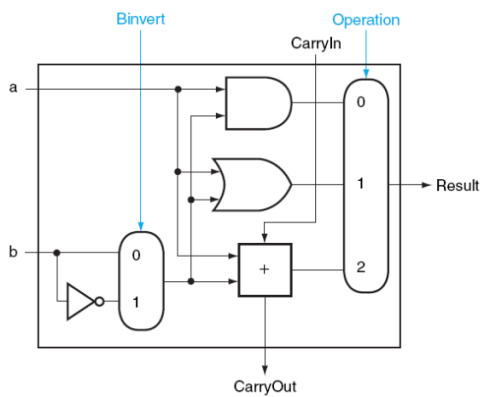


In base al valore di **operation**, il circuito esegue un'operazione diversa:

- 0: AND Gate;
- 1: OR Gate;
- 2: Circuito somma;

Operation riesce a trasmettere più di 2 valori utilizzando un bus.

10.3 Sottrazione su ALU ad 1 bit



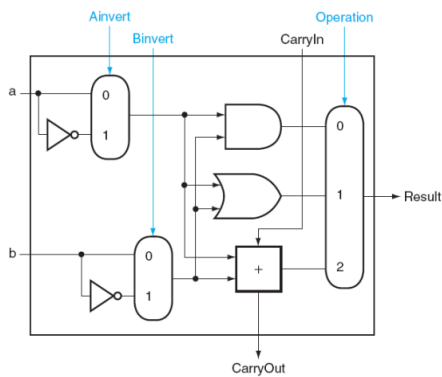
Il **Binverter** è un collegamento a 1 bit che, se positivo (1), inverte il valore di b permettendo di realizzare la **sottrazione in CA2**.

L'operazione realizzata da somma e Binverter è: $a - b = a + (\bar{b} + 1)$

10.4 NOR e NAND su ALU ad 1 bit

L'operazione NOR è implementabile nel circuito dell'ALU ad un bit tramite la **Prima Legge di De Morgan**:

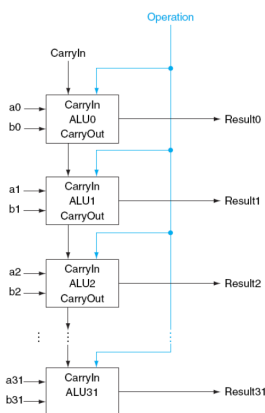
- $NOT (a OR b) = (NOT a) AND (NOT b)$;
- $NOT (a AND b) = (NOT a) OR (NOT b)$



L'**Ainverter** è un collegamento a 1 bit che, se positivo (1), inverte il valore di a .

L'inversione dei valori a e b tramite i due inverter permette di realizzare **NOR** e **NAND** applicando la prima legge di De Morgan.

10.5 ALU a 32 bit - Ripple Carry



Ponendo multiple ALU ad 1 bit in cascata è possibile realizzare una ALU che riesce a computare input fino a 32 bit.

Questo tipo di organizzazione è noto come **Ripple Carry**.

10.6 Set Less Than su ALU a 32 bit

Output: 1 se $a < b$, altrimenti 0;

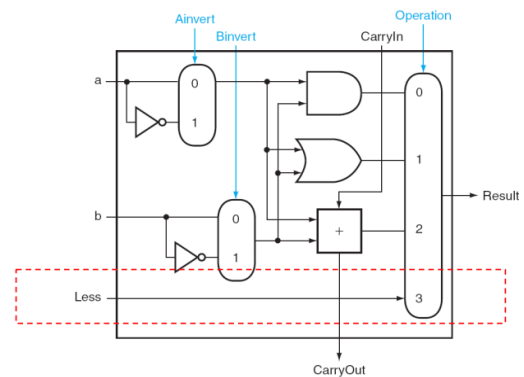
Funzionamento: imposta tutti i bit da 1 a 31 a valore 0, e assegna all'ALU 0 il risultato dell'operazione;

Se $a - b$ è negativo, l'operazione restituisce 1 all'ALU 0;

All'interno dell'ALU ad 1 bit, l'operazione SLT è detta **less**.

Il risultato dell'ALU 0 deriva dal valore dell'ALU 31 [segno].

Nell'ALU 31:
Se CarryIn = CarryOut, less = 0.



Durante l'operazione è necessario verificare l'overflow.

La formula per il **test dell'overflow** è:

$$overflow = (\bar{a} \cdot b \cdot result) + (a \cdot \bar{b} \cdot \overline{result})$$

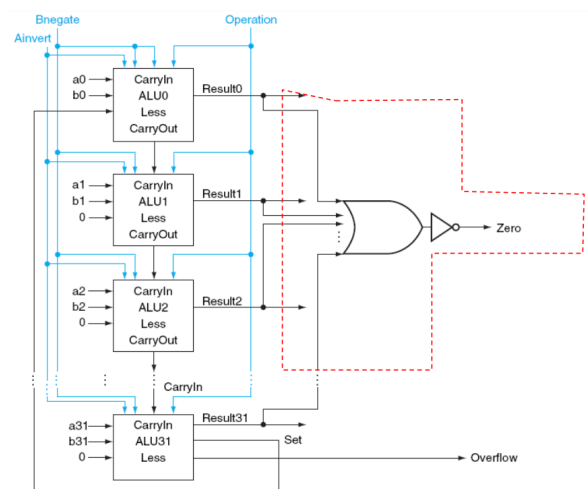
10.6 Branch On Equal su ALU a 32 bit

Output: 1 se $a = b$, altrimenti 0;

Funzionamento: $a - b = 0 \Leftrightarrow a = b$.

Il BEQ fa l'OR di tutte le uscite e poi fa il complemento del risultato.

$$0 = \overline{(R31 + R30 + \dots + R0)};$$



Circuiti Sequenziali

11.3 Introduzione

Esistono 2 tipi di **circuiti sequenziali**:

- Sincroni: hanno bisogno di un clock per allineare temporalmente le istruzioni svolte;
- Asincroni: non utilizzano il clock;

I Circuiti sequenziali sono composti da una **rete combinatoria** e degli **elementi di memoria**. In ogni istante, i circuiti combinatori hanno uno stato determinato dalle informazioni memorizzate.

11.2 Latch

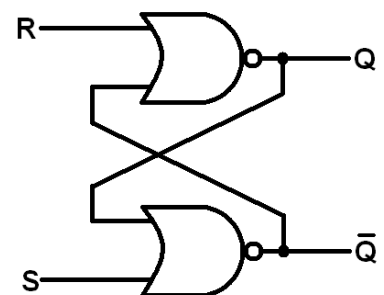
circuito in grado di **memorizzare** 1 bit.

S-R Latch (Set Reset Latch) è un tipo di circuito latch costituito da 2 porte NOR.

Input $[0,0]$ è detto **combinazione di riposo**, in quanto mantiene il valore memorizzato in precedenza.

Input $[1,1]$ è uno **stato non ammissibile**, in quanto Q e \bar{Q} ottengono entrambi valore 1 e infrangono il principio di complementarità.

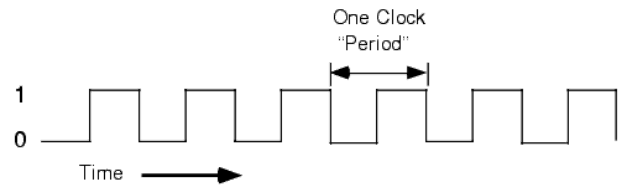
Il circuito SR Latch è utilizzato raramente, in quanto deve essere stabile ed è necessario evitare che gli output intermedi vengano memorizzati.



11.3 Clock

Un **segnale a scalino**, utilizzato per determinare il ritmo di esecuzione delle istruzioni.

Il clock lavora su un **periodo** T .



La **frequenza** di lavoro si misura in **Hertz** [Hz] e ha formula $F = \frac{1}{T}$.

La parte crescente del segnale è detta Rising Edge.

La parte decrescente del segnale è detta Falling Edge.

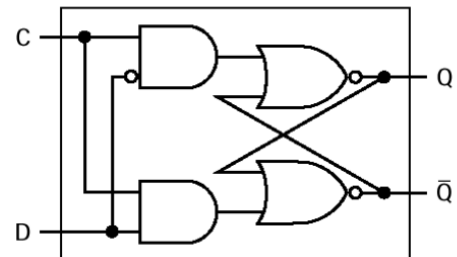
11.4 D-Latch

S-R Latch sincronizzato utilizzando un clock.

Quando il clock ha valore basso, il D-Latch mantiene il suo valore precedente anche se vengono eseguite altre istruzioni, altrimenti esegue una delle seguenti operazioni:

- D=1 corrisponde al **setting** $\rightarrow S=1, R=0$;
- D=0 corrisponde al **resetting** $\rightarrow S=0, R=1$;

Il **segnale D** è ottenuto tramite un circuito combinatorio.



Questo deve essere stabile quando $C=1$. Il segnale D deve avere margine d'errore prima e dopo il segnale alto di C per evitare errori:

questi sono detti **Setup Time** [prima] e **Hold Time** [dopo].

Quando il clock torna a valore 0, l'output Q si stabilizza.

Metodi di Timing

Il **Timing** può essere misurato in 2 modi:

- Level Triggered: sincronizza in base al **segnale alto** del clock;
- Edge Triggered: sincronizza in base alla **variazione di altezza** del segnale (salita o discesa);

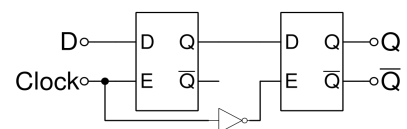
Con il sistema edge triggered la memorizzazione avviene istantaneamente ed un eventuale **segnale sporco** non fa in tempo ad arrivare a causa dell'istantaneità della memorizzazione.

I circuiti sequenziali edge triggered sono detti **Flip-Flop**.

12.1 D Flip-Flop

Effettua input e output durante lo stesso ciclo.

Questo circuito è realizzato ponendo 2 D-Latch in serie, il primo **master** ed il secondo **slave**.



Funzionamento:

- **segnale in salita**: il primo latch è aperto e memorizza D, viene dato Q' in output, ma il secondo latch è chiuso;
- **segnale in discesa**: il secondo latch si apre per memorizzare Q';
- Il **nuovo valore Q** entra nel circuito e il primo latch si chiude;

Register File

13.1 Struttura & Componenti

Un registro è costituito da n flip-flop.

I registri sono organizzati in una struttura detta **Register File**.

Nell'architettura MIPS ci sono 32 registri da 32 bit, per un totale di 1024 flip-flop. Il Register File permette la **lettura** di 2 registri e la **scrittura** di un registro.

Nel datapath il clock viene messo in relazione AND con un **segnale di controllo write**, che devono essere coordinati per permettere il cambio di valori nella memoria.

13.2 Lettura

Durante la lettura di un registro, il register file utilizza **2 segnali**: Read Reg1, Read Reg2. Si utilizza un multiplexer con 32 ingressi, uno per ogni registro.

13.3 Scrittura

La scrittura avviene utilizzando **3 segnali** in ingresso: Register Number, Register Data e Write.

Si utilizza un decoder per ottenere il numero del registro su cui scrivere, il segnale write (in AND con il clock) si combina in AND con l'output del decoder.

Se write non è a segnale alto, non viene effettuata nessuna modifica.

Memorie

14.1 Parametri

Oltre alle piccole memorie (registri) esistono altri tipi di memorie distinguibili in base ai parametri:

- **Dimensione:** quantità di bit memorizzabili;
- **Velocità:** tempo tra la richiesta di un dato e la sua restituzione;
- **Consumo:** corrente assorbita;
- **Costo;**

La memoria è organizzata in **gerarchie**, poste in ordine di velocità. Maggiore è la velocità di una memoria, più vicina sarà essa alla CPU.

14.2 Tipi di Memorie

A seguito diversi tipi di memorie e le loro caratteristiche:

- **RAM** - Random Access Memory: è una memoria volatile che permette accesso diretto a qualsiasi indirizzo di memoria con gli stessi tempi;
- **SRAM** - Static RAM: ogni cella di memoria è composta da un D-Flip Flop;
- **DRAM** - Dynamic RAM: ogni cella è costituita da un transistor e un condensatore, tramite il quale mantiene l'informazione;
- **SSRAM** - Synchronous Static RAM: SRAM sincronizzata tramite un clock. Uno degli utilizzi più comuni per la SSRAM è la cache;
- **SDRAM** - Synchronous Dynamic RAM: DRAM sincronizzata tramite un clock. La sua applicazione più diffusa è il suo utilizzo nei moduli DIMM per i PC.

14.3 Calcoli sulla memoria

L'**altezza della memoria** si calcola con l'operazione: $\frac{\text{dim. memoria}}{\text{dim. word}}$

La **lunghezza dell'indirizzo di memoria** è uguale all'esponente dell'altezza di memoria ridotta in base 2.

Macchine a Stati Finiti

15.1 Utilizzo

Le **Finite State Machine** sono usate per descrivere i circuiti sequenziali. Le FSM sono sincronizzate tramite clock.

15.2 Tipi

Sono composte da un set di stati e 2 funzioni:

- **Next-State Function:** Determina lo stato successivo partendo dallo stato corrente e dagli input;
- **Output Function:** Determina l'insieme di risultati partendo dallo stato corrente;

Esistono diversi tipi di FSM:

- **FSM di Moore:** utilizza lo stato corrente, usato come controller;
- **FSM di Mealy:** utilizza lo stato corrente e dagli input;

Filosofia di Progettazione della CPU

16.1 Tipi principali

RISC - Reduced Instruction Set Computing

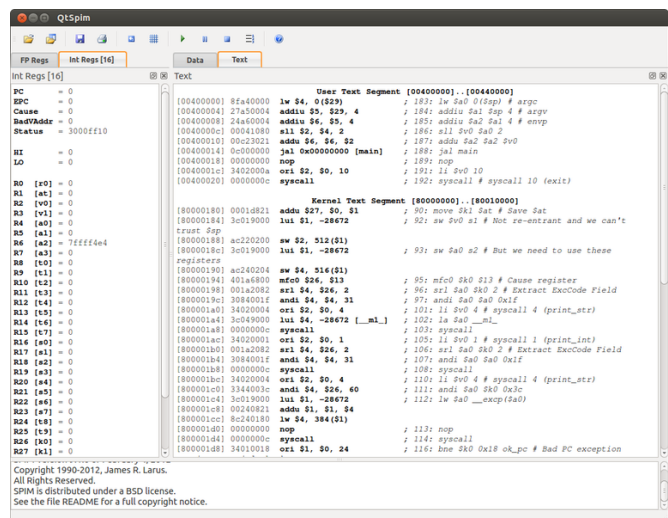
Utilizza poche istruzioni semplici. Esegue velocemente ogni singola istruzione, ma deve usarne molte per fare istruzioni più complesse (**MIPS** è un architettura RISC).

CISC - Complex Instruction Set Computing

Utilizza istruzioni più complesse. È strutturalmente complicato ed esegue le istruzioni basilari più lentamente, ma ne ha bisogno meno per eseguire istruzioni complesse.

16.2 MIPS

È un'architettura RISC con 32 registri da 32 bit. Può manipolare dati solo sui registri. Trasferisce i dati tra memoria e registri e permette l'alterazione del flusso di controllo tramite salti.



The screenshot shows the QtSpim MIPS simulator. On the left, the 'FP Regs' and 'Int Regs [16]' are listed, with values for registers \$0 through \$15. The main window displays assembly code with comments. The code is divided into sections: 'User Text Segment [00400000]..[00440000]', 'Kernel Text Segment [80000000]..[80010000]', and 'Data'. The assembly code includes instructions like 'lw \$4, 0(\$29)', 'addiu \$5, \$29, 4', 'addiu \$6, \$5, 4', 'ori \$2, \$0, 10', 'syscall', 'sw \$4, 516(\$1)', 'mf0 \$26, \$13', 'andi \$4, \$4, 31', 'ori \$2, \$0, 4', 'lui \$4, -26672', 'syscall', 'ori \$2, \$0, 1', 'ori \$4, \$26, 2', 'andi \$4, \$4, 31', 'ori \$2, \$0, 4', 'andi \$4, \$26, 40', 'addiu \$1, \$1, \$4', 'lw \$4, 384(\$1)', 'nop', 'syscall', and 'ori \$1, \$0, 24'. Comments explain the purpose of each instruction, such as 'move \$4: Save \$4', 'sw \$4: Not re-entrant and we can't', 'mf0 \$26: Cause register', 'andi \$4: Extract ExcCode Field', 'ori \$2: syscall', 'ori \$4: syscall 1 (print_int)', 'andi \$4: Extract ExcCode Field', 'ori \$2: syscall 4 (print_str)', 'andi \$4: syscall', 'lw \$4: syscall', 'nop', 'syscall', and 'ori \$1: syscall'.

Le istruzioni MIPS sono a 32 bit e si dividono in 3 categorie.

16.3 R-Type

- **op - Operation Code** [6 bit]: identificativo dell'operazione;
- **rs** [5 bit]: primo registro sorgente;
- **rt** [5 bit]: secondo registro sorgente;
- **rd** [5 bit]: registro di destinazione;
- **shamt** [5 bit]: operazione di shift;
- **funct** [6 bit]: variante dell'operazione;

add \$t0, \$s, \$9 → somma il reg8 + reg9 e inserisce il risultato nel reg10

16.4 I-Type

- **op - Operation Code** [6 bit]: identificativo dell'operazione;
- **rs** [5 bit]: primo registro sorgente;
- **rt** [5 bit]: secondo registro sorgente;
- **Valore Immediato** [16 bit];

addi \$t0, \$s, \$9 → somma il reg8 + reg9 e inserisce il risultato nel reg10

16.5 J-Type

Le istruzioni J-Type sono composte da:

- **op - Operation Code** [6 bit]: identificativo dell'operazione;
- **Jump word Address** [26 bit]: indirizzo di memoria dell'istruzione a cui bisogna saltare;

j \$t0 → salta all'indirizzo di memoria contenuto nel reg10

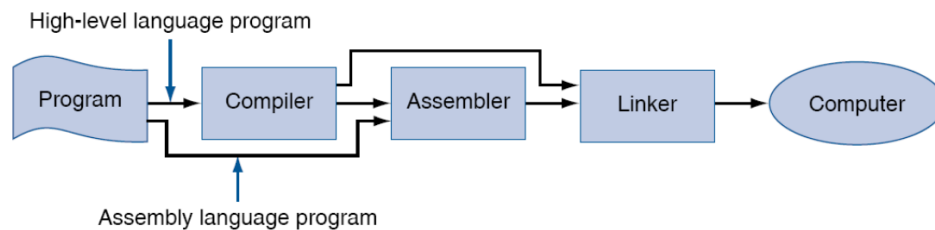
Catena Programmatica

17.1 Assembly

Assembly è un linguaggio a basso livello, simile al linguaggio macchina, ma più comprensibile all'uomo.

Il codice in assembly è scritto specificamente per una determinata architettura, questo permette:

- Maggiore efficienza;
- Programmi più compatti;
- Massimo sfruttamento delle potenzialità dell'hardware;



17.2 Debugger

Il **debugger** consente di eseguire il codice in modo controllato per la ricerca di errori. Il debugger consente:

- Esecuzione step-by-step del programma;
- Ispezione delle variabili;
- Interruzione nei **breakpoint**;
- Visualizzazione degli indirizzi di memoria;

17.3 Compiler

Il **compiler** si occupa di convertire il linguaggio di alto livello generato da un programmatore in linguaggio assembly, o linguaggio macchina a seconda del tipo di compiler.

17.4 Assembler

I programmi di alto livello vengono convertiti in assembly da un compilatore.

Dopo la fase di compilazione, il programma viene tradotto in linguaggio macchina dall'**assembler**.

L'assemblaggio esamina il codice riga per riga, traducendolo in linguaggio macchina una alla volta.

A causa delle istruzioni jump, l'assemblatore legge il codice 2 volte, per questo è detto **traduttore a due passi**.

L'assembler dispone di una **tabella dei simboli** utile per convertire il codice.

17.5 Linker

Le **etichette** possono essere locali o globali, queste ultime devono essere risolte dal **linker** e non dall'assembler.

Il linker si occupa di inserire in memoria in modo simboli il codice e i moduli dati, determinare gli indirizzi delle etichette, correggere i riferimenti interni ed esterni e generare il file eseguibile (l'insieme di tutti i file object creati durante l'assemblaggio).

16.6 Loader

Il loader si occupa dei seguenti compiti:

- Crea spazio di indirizzamento e copia istruzioni e dati in memoria;
- Copia parametri nello stack e inizializza lo stack pointer;
- Procedura di startup che copia i parametri nei registri ed esegue la procedura principale;
- Quando la procedura principale restituisce il controllo, la macchina termina il programma;

Datapath

18.1 Realizzazione di un Datapath

Si stabilisce il set di istruzioni da implementare.

Si identificano i componenti del datapath (ALU, Register File, ecc).

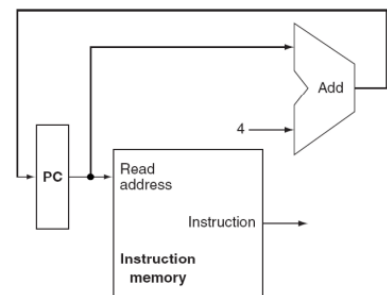
Si sceglie la metodologia di clocking:

- **datapath singolo ciclo:** 1 ciclo di clock esegue ogni operazione;
- **datapath multiciclo:** esegue più cicli di clock per più operazioni;

L'esecuzione di un'istruzione si divide in **3 fasi**.

18.3 Fetch

- I. Legge l'istruzione dalla memoria e la salva nell'Instruction Register;
- II. L'indirizzo di memoria che indica l'istruzione da leggere si trova nel registro Program Counter;
- III. Dopo la lettura dell'istruzione in PC viene incrementato di 4 per indicare la prossima istruzione da leggere;



18.4 Decode

Legge i vari campi dell'istruzione e identifica di che istruzione si tratta controllando opcode ed eventualmente funct code.

18.5 Execute

esegue le istruzioni "decodificate" dal decoder, per sapere su quali registri lavorare controlla i gruppi di 5 bit delle istruzioni.

Datapath Multiciclo

19.1 Vantaggi e Svantaggi

Il datapath multiciclo è realizzato con cicli di lunghezza fissa minore, tuttavia necessita più cicli per l'esecuzione di un'istruzione.

Questa suddivisione permette di avere istruzioni realizzate da un numero di cicli di clock variabile, e utilizzando più volte gli stessi componenti, permettendo meno **replicazione** nella struttura del datapath.

Per eseguire istruzioni multiciclo il processore necessita registri aggiuntivi per memorizzare gli **stati intermedi** dell'istruzione:

- **IR** - Instruction Register;
- **MDR** - Memory Data Register;
- **A, B** - Registri tra Register File e ALU;
- **ALUout** - Output dell'ALU;

Grazie alla possibilità di riutilizzo dei componenti, la struttura del datapath viene spogliata di elementi che risultano ridondanti.

La **ALU** viene utilizzata per calcolare i salti e incrementare il PC e la **memoria** è utilizzata per leggere istruzioni e leggere/scrivere dati.

Il datapath multiciclo deve inoltre bilanciare la quantità di lavoro ad ogni ciclo di clock e memorizzare i valori nei registri addizionali.

19.2 Struttura delle istruzioni

Un'istruzione in datapath multiciclo ha una lunghezza variabile tra i 3 e i 5 cicli di clock.

I 5 possibili passi di esecuzione di un'istruzione sono:

R-Type	Load Word e Store Word	Branch	Jump
IR = PC PC = PC+4			
A = Reg(IR [25:21]) B = Reg(IR [20:16]) ALUOut = PC + sign-extend (IR[15:0]<<2)			
ALUOut = A op B	ALUOut = A + sign-extend IR[15:0]	if (A == B) PC = ALUOut	PC = { PC[31:28], IR[25:0], 2'b00 }
	lw: MDR = Memory[ALUOut] sw: Memory[ALUOut] = B		
	lw: Reg[IR[20:16]] = MDR		

Eccezioni e Interruzioni

20.1 Definizione

Eccezione: evento sincrono generato dal processore a causa di problemi nell'esecuzione. Le eccezioni devono essere risolte da un gestore di eccezioni (exception handler).

Il programma continua solo se l'eccezione è risolvibile.

Interruzione: evento asincrono generato esternamente al processore. Le interruzioni vengono gestite al termine dell'istruzione in esecuzione, interrompendo l'esecuzione del programma e dando priorità alla gestione dell'interrupt.

Il datapath stesso deve essere lo strumento che rileva le eccezioni, ad esempio, l'overflow è un'eccezione rilevata dall'ALU durante la fase execute.

20.2 Esecuzione delle istruzioni

Esecuzione normale: le istruzioni vengono eseguite nel flusso preventivato.

Esecuzione eccezionale: succede qualcosa nel datapath o nelle periferiche che devia lo scorrimento delle istruzioni.

20.3 Gestione di eccezioni ed interrupt

Il controllo del processore deve gestire gli eventi inattesi, l'hardware deve quindi:

- Interrompere l'esecuzione del programma corrente;
- Salvarne lo stato (parzialmente), per poi riprenderne l'esecuzione se possibile;
- Saltare ad una routine dell'OS per gestire l'eccezione/interrupt;
- Se possibile, recuperare le informazioni e riprendere l'esecuzione del programma;

Se il programma è scritto rispettando le norme di utilizzo dei registri della CPU, è più probabile che venga effettuato un salvataggio delle informazioni corretto durante la gestione degli errori.

20.4 Identificazione dell'errore

- **Indirizzo Fisso:** registro dedicato

Il controllo della CPU, prima di saltare all'handler dell'OS, salta ad un registro interno contenente l'identificatore numerico che permette di riconoscere il tipo di eccezione.

- **Interruzioni Vettorizzate:**

Utilizza handler diversi per exception/interrupt differenti. Il controllo della CPU sceglie l'handler corretto, saltando al suo indirizzo.

Per questo viene predisposto un array di indirizzi, con uno di questi disponibile per ogni exception/interrupt e indirizzato tramite il codice numerico dell'errore.

Eccezioni in MIPS

21.1 Metodo Utilizzato

MIPS utilizza l'indirizzo fisso, detto Cause, per memorizzare il motivo dell'eccezione.

L'indirizzo di memoria dell'istruzione che ha causato l'errore si chiama **Exception Program Counter (EPC)**.

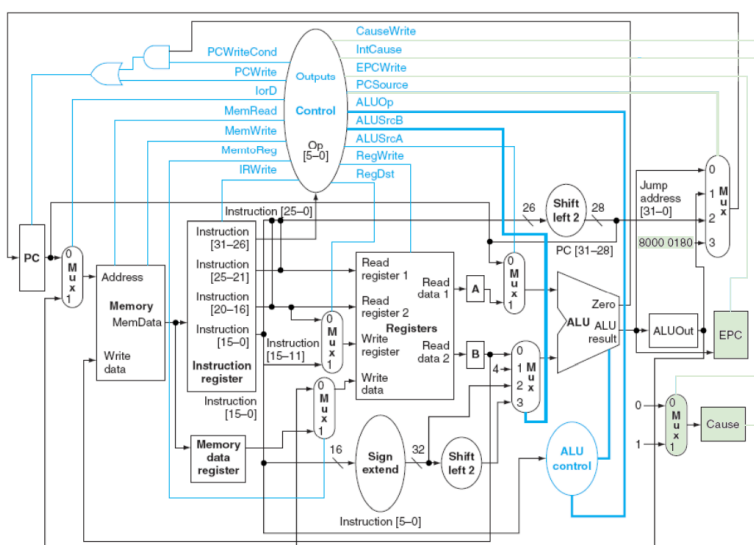
21.2 Passi da eseguire

La gestione delle eccezioni in MIPS si divide in 3 sezioni principali:

- Individuare l'evento inatteso e salvarlo in Cause;
- Salvare l'indirizzo dell'istruzione corrente \rightarrow EPC := PC-4
- Passare il controllo al gestore software, interrompendo l'esecuzione corrente;

MIPS salva solo il PC, in quanto la routine si deve occupare di salvare lo stato corrente del programma, se necessario.

21.3 Gestione delle Eccezioni nel Datapath



Verde: Componenti aggiunti/modificati al datapath per la gestione delle eccezioni

Prestazioni

22.1 Banda passante

Rappresenta la quantità dei dati che si può trasferire per unità di tempo, rappresenta una misura di flusso.

22.2 Latenza

Rappresenta il tempo che passa tra l'istante in cui una periferica è pronta per il trasferimento e l'istante in cui il dato viene trasferito.

Input/Output

23.1 Definizione

I/O - input/output: insieme di architetture e dispositivi per il trasferimento di informazioni da e verso l'elaboratore.

23.2 Tipi di Bus

Bus di Sistema: collega la CPU con la memoria e le periferiche;

Bus Dati: trasferisce data da e verso i dispositivi;

Bus di Controllo: trasporta informazioni per la definizione delle operazioni e la sincronizzazione dei dispositivi;

Bus degli Indirizzi: trasmette gli indirizzi alla memoria o alle periferiche per lettura/scrittura;

23.3 Periferiche

Dispositivi per l'I/O delle informazioni collegati tramite bus alla CPU. Le interfacce sono standardizzate per semplificare la comunicazione.

Le interfacce hanno una componente hardware e un componente software. Il componente software è diviso tra **firmware** (software della periferica) e **driver** (software del computer).

Le periferiche contengono anche registri di stato e di dati che sono mappati in memoria e non accessibili all'utente.

Dal punto di vista della CPU invece, i registri delle periferiche sono accessibili come l'accesso alla memoria.

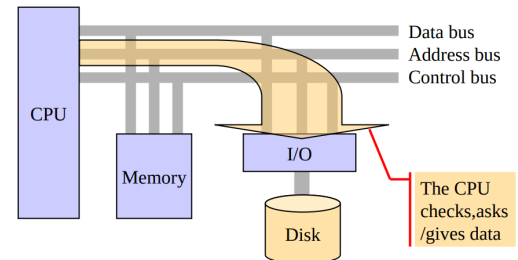
Parte della memoria è riservata alla comunicazione con le periferiche ed ognuna di esse ha una parte dedicata.

23.4 I/O gestito da programma

La CPU si occupa sia del controllo sia del trasferimento dei dati, la periferica ha un ruolo passivo, si occupa solo di eseguire gli ordini della CPU.

Vantaggi: risposta veloce al ready bit;

Svantaggi: La CPU bloccata in stato di busy wait;



Prestazioni:

Banda passante alta, la CPU trasferisce subito il dato e la gestione della periferica richiede poche istruzioni.

Latenza minima, la CPU nota subito lo stato della periferica.

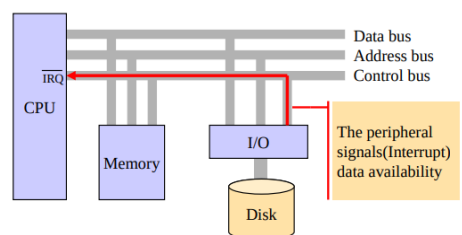
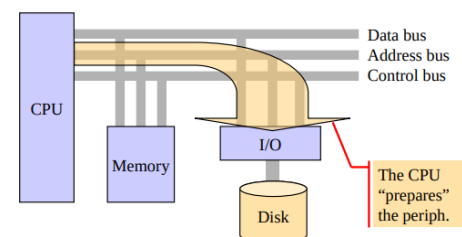
23.5 I/O guidato da interrupt

La periferica interrompe la CPU tramite il bus di controllo.

Quando il processore rileva l'interrupt risponde con un segnale di Acknowledge, interrompe l'esecuzione del programma ed esegue la procedura di interrupt.

Vantaggi: La CPU non fa busy waiting.

Svantaggi: La CPU deve gestire le operazioni di trasferimento.



Prestazioni:

Banda passante minima, ogni trasferimento necessita di più tempo.

Latenza massima, causata dal numero di operazioni da eseguire.

23.6 DMA - Accesso diretto alla memoria

In DMA, la periferica diventa autonoma nell'accesso alla memoria, in quanto gestisce lei stessa il trasferimento dei dati, senza intervento della CPU. Per realizzare ciò la periferica necessita 2 registri in più:

- Un registro che indica l'indirizzo da cui trasferire i dati;
- Un registro che indichi la quantità di dati da gestire;

Anche questi registri devono essere mappati in memoria. Alla fine del trasferimento la periferica invia un interrupt alla CPU per segnalare la fine del trasferimento.

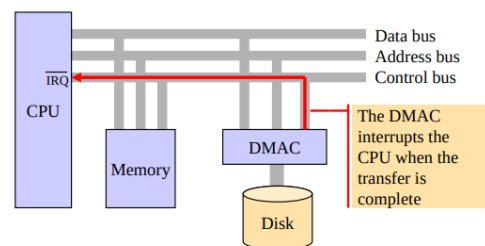
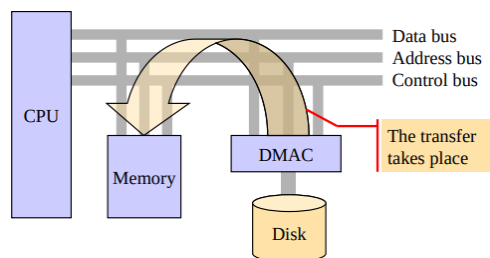
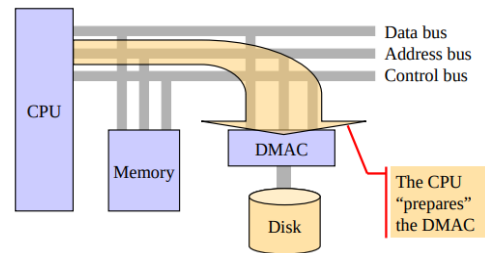
Vantaggi: La CPU non deve eseguire nessuna istruzione.

Svantaggi: La periferica ha bisogno di 2 registri per eseguire l'accesso alla memoria.

Prestazioni:

Banda passante massima, CPU non deve eseguire nessuna istruzione.

Latenza minima, nessuna istruzione è eseguita dalla CPU.



Pipeline

24.1 Introduzione

Il pipeline è una tecnica utilizzata nel datapath singolo ciclo utilizzato per eseguire più step di diverse istruzioni in un singolo ciclo di clock.

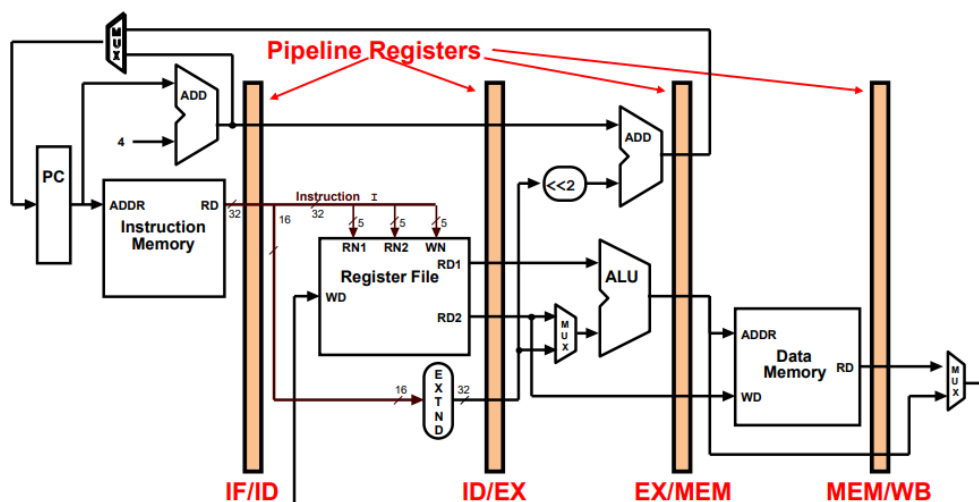
I 5 passaggi per eseguire un'istruzione sono:

- **IF:** Instruction Fetch;
- **ID:** Instruction Decode;
- **EX:** Execute;
- **MEM:** Memory read or write;
- **WB:** Writeback, o salvataggio di un risultato;

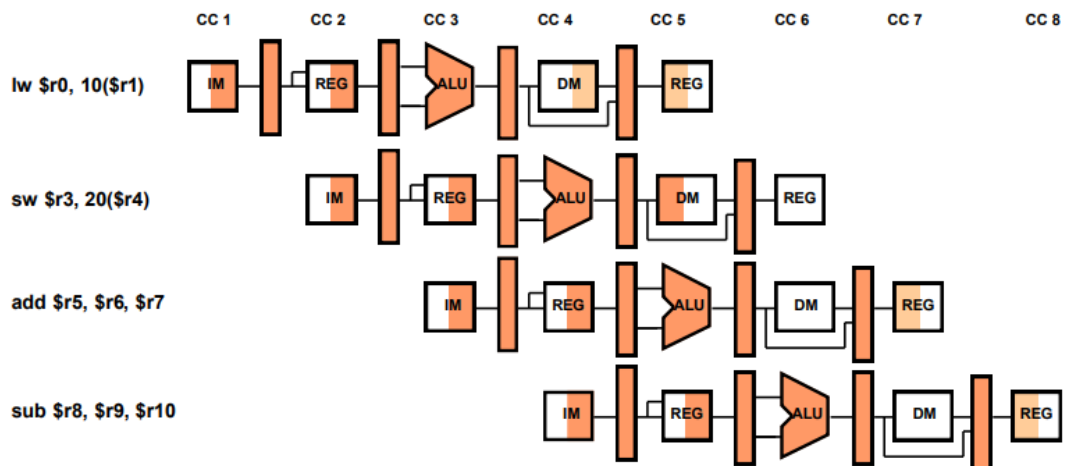
Ogni istruzione può richiedere solo alcuni di questi:

Istruzione	Step Richiesti				
beq	IF	ID	EX		
R-Type	IF	ID	EX		WB
sw	IF	ID	EX	MEM	
lw	IF	ID	EX	MEM	WB

24.2 Suddivisione nel datapath



24.3 Simulazione di esecuzione in pipeline



La durata di un ciclo di clock nel datapath pipelined equivale alla durata dell'operazione più lenta.

Hazard

25.1 Definizione

Si dice hazard, la situazione in cui un'istruzione non può seguire immediatamente l'istruzione precedente.

Una soluzione comune ad ogni tipo di hazard è lo **stallo**, ovvero l'attesa da parte dell'istruzione in coda.

25.2 Structural Hazard

Si dice hazard strutturale quando un'istruzione prova ad accedere ad una **risorsa occupata**.

Un esempio di hazard strutturale è il tentativo di accesso di due istruzioni alla stessa memoria.

Gli hazard strutturali possono essere risolti aggiungendo un ritardo di 1 ciclo di clock tra le 2 istruzioni in pipeline, oppure disponendo **memorie separate** per le istruzioni.

Quest'ultima soluzione è detta **Harvard Architecture** ed è la soluzione implementata da MIPS.

25.3 Data Hazard

Si dice hazard dati il tentativo di usare **informazioni** prima che siano disponibili.

Gli hazard dati possono essere risolti tramite diverse tecniche, tra cui il forwarding.

Utilizzando la tecnica del **forwarding**, tramite l'aggiunta di alcuni componenti al datapath, è possibile utilizzare i dati in output dell'ALU appena computati, che normalmente dovrebbero subire altri passaggi prima di essere disponibili.

25.4 Control Hazard

Si dice hazard di controllo un **cambiamento inaspettato del flusso** degli indirizzi delle istruzioni.

Alcuni esempi di hazard di controllo sono le istruzioni jump e branch.

Gli hazard di controllo possono essere risolti anticipando il punto di decisione nel pipeline, ritardando la decisione (necessita supporto dal compiler) o effettuando predict.

Il **predict** permette di assumere il risultato di un branch in attesa del risultato effettivo.

Il valore atteso nel branch è false (**predict not taken**), se la predizione è corretta, l'istruzione continua senza rallentamenti, altrimenti bisogna effettuare uno stallo del pipeline per poi ripartire dall'istruzione successiva al branch.

Per evitare che lo stato della macchina sia cambiato da un'istruzione annullata, le operazioni in grado di alterare lo stato sono poste come ultimo step del pipeline (MemWrite, RegWrite).

Gerarchie di Memoria

26.1 Introduzione

La memoria viene strutturata seguendo una **gerarchia**, che divide questa in livelli caratterizzati da velocità e dimensione.

Al crescere della **distanza** dalla CPU, aumenta la capienza della memoria ma ne diminuisce la velocità e il costo.

26.2 Tipi di memoria

La **memoria interna della CPU** [Registri] è costituita da registri e caratterizzata da velocità alta e dimensioni ridotte.

La **memoria centrale** [RAM] ha dimensioni molto maggiori, ma più lenta, come la memoria interna è accessibile direttamente dalla CPU.

La **memoria cache** esiste come via di mezzo tra memoria interna e centrale.

Le **memorie secondarie** [Mass Storage] hanno alta capacità e bassi costi, inoltre non sono volatili.

26.3 Principio di Località

Un programma accede solo ad una piccola parte dello spazio da lui occupato in un determinato lasso di tempo.

Questo rappresenta la base del comportamento di un programma.

Esistono 2 principi di località:

- **Località Temporale:** tende a far riferimento ad un elemento più volte in un breve lasso di tempo. Posiziona i dati in base alla gerarchia di distanza della memoria;
- **Località Spaziale:** tende a far riferimento ad elementi con indirizzi vicini;

I programmi non vedono la gerarchia, ma accedono ai dati come se fossero sempre in memoria centrale.

26.4 Definizioni

Blocco/Linea: più piccola quantità di informazioni in una gerarchia di memoria;

Hit: l'informazione richiesta si trova nel blocco superiore;

Miss: l'informazione richiesta non è presente nel livello superiore e occorre accedere ad un livello più distante;

Hit Rate: $\frac{Hit}{Numero\ di\ Richieste}$;

Miss Rate: $1 - Hit\ Rate$;

Tempo di Hit: tempo di accesso al livello superiore;

Tempo di Miss: tempo necessario a sostituire un blocco con quello di livello inferiore, trasferendo i dati di questo alla CPU;

Cache

27.1 Introduzione

La **memoria cache** è la memoria esterna più vicina alla CPU.

L'utilizzo della cache è nascosto al programmatore e viene utilizzato un **algoritmo di caching** per disporre i dati.

Questo si basa sui principi di località, mantenendo i dati richiesti vicino alla CPU e muovendo i blocchi contigui che contengono i dati richiesti.

27.2 Tecniche di indirizzamento

L'algoritmo di caching è basato sui principi di località, mantenendo i dati richiesti vicino alla CPU e muovendo i blocchi contigui che contengono i dati richiesti.

Esistono 3 tecniche di organizzazione della cache:

- Direct Mapped Cache;
- Fully Associative Cache;
- Set Associative Cache;

27.3 Direct Mapped Cache

Associa una sola locazione di memoria ad ogni blocco.

L'indirizzo che permette di identificare un blocco è costituito da:

- **Tag:** contenente informazioni necessarie a verificare se una parola corrisponde a quella cercata;
- **Indice:** utilizzato per selezionare il blocco della cache;
- **Offset:** bit necessari per selezionare il byte richiesto nella parola;

La posizione di una parola in cache è data da $\log_2(n_{LSB})$ dell'indirizzo in memoria principale.

Ogni linea di cache include 3 elementi fondamentali:

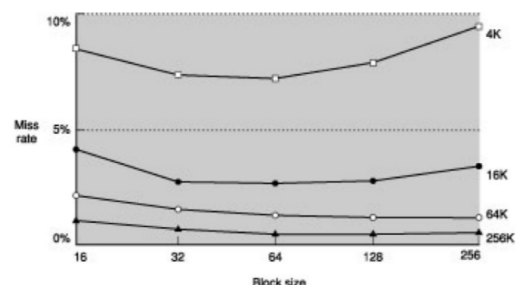
- **Bit di Validità:** indica se i dati sono nella linea di cache, il suo valore di default è miss (**compulsory miss**);
- **tag:** identificativo del blocco mappato nella linea di cache;
- **Blocco di dati:** formato da una o più parole;

È possibile effettuare la **mappatura** di un blocco con la formula:

$$\frac{\text{indirizzo}}{\text{dimensione blocco}} \bmod n_{\text{blocchi in cache}}$$

La dimensione dei blocchi è in **relazione diretta** al miss rate.

Una grande dimensione dei blocchi permette di massimizzare i vantaggi della località spaziale, ma aumenta il miss rate, il tempo di trasferimento dei blocchi e la **miss penalty**, ovvero il tempo speso per correggere un miss.



27.4 Fully Associative Cache

Ogni blocco di memoria può essere collocato in una posizione qualsiasi in cache e per trovarlo è necessario cercarlo in tutta la memoria. Per ottimizzarne il suo ritrovamento si fa uso della **ricerca parallela**.

27.5 Set Associative Cache

Ogni blocco ha a disposizione un numero fisso (≥ 2) di posizioni possibili in cache.

I blocchi sono raggruppati in set e la cache è detta set associativa ad N vie (o **N-way set associative**).

Ogni indirizzo di memoria corrisponde ad un set, per trovare un blocco all'interno di un set è necessario confrontarli in parallelo.

Svantaggi:

La set associativa cache necessita di più componenti della direct access cache, che ne aumentano la complessità e la lentezza, inoltre rende disponibile il blocco alla CPU dopo la decisione Hit/Miss e la selezione del set.

Vantaggi:

La set associativa cache diminuisce il miss rate.

L'**associatività della cache**, ovvero il numero di possibili posizioni possibili per un determinato blocco, è direttamente proporzionale a vantaggi e svantaggi di questa struttura.

27.6 Tecniche di Sostituzione

Nella cache a mappatura diretta, se una linea è già occupata, si elimina il contenuto e si rimpiazza con il nuovo blocco.

Nelle cache associative è necessario decidere quale blocco sostituire in caso di miss, in particolare nella cache fully associative ogni blocco è un candidato per la sostituzione.

Esistono diverse politiche di sostituzione:

- **Sostituzione Random:** politica diffusa nelle cache con alta associatività, per evitare algoritmi dispendiosi;
- **Least Recently Used [LRU]:** sostituisce il blocco inutilizzato da più tempo, massimizzando il principio di località temporale. Ad ogni blocco si associa un contatore, incrementa in caso di hit e diminuisce in caso di miss;
- **First In First Out [FIFO]:** sostituisce il blocco più vecchio, approssimando la politica LRU per minimizzare le risorse spese;

27.8 Tecniche di Aggiornamento

Se un blocco non è presente nella cache (miss) bisogna mettere in stallo l'intera CPU.

Al verificarsi di un miss, sono eseguiti i seguenti passaggi:

- inviare PC - 4 alla memoria;
- Lettura della memoria;
- Scrittura della cache;
- Riavvia l'istruzione che ha causato il miss;

Scrivere un dato in cache genera un'incoerenza temporanea tra le gerarchie di memoria.

Per risolvere questo problema esistono 3 tecniche: **Write-through**, **Write-back** e **write buffer**.

27.9 Write-through

Scrive i dati contemporaneamente in cache e nei livelli inferiori.

Vantaggi:

Elimina l'incoerenza tra le memorie ed è la tecnica più semplice da implementare.

Svantaggi:

Le operazioni di scrittura sono limitate alla velocità della memoria più lenta e aumenta l'utilizzo del bus di sistema.

27.10 Write-back

Scrive i dati solamente in cache, scrivendo ai livelli inferiori solo quando deve essere sostituito.

Vantaggi:

Le scritture avvengono alla velocità massima per la rispettiva memoria.

Svantaggi:

Ogni sostituzione provoca un trasferimento in memoria.

27.11 Write Buffer

Utilizza la tecnica write-through ponendo un **buffer** tra la cache e la memoria di livello inferiore.

I dati vengono scritti nella cache e nel buffer, e il controller di memoria si occuperà di copiarli dal buffer alla memoria stessa.

Il buffer è efficiente se: $scrittura < \frac{1}{cicli\ di\ scrittura\ DRAM}$, altrimenti viene saturato e genera overflow.

Per risolvere il problema di overflow è possibile separare la cache in più livelli.

27.12 Write Miss

Durante la scrittura è possibile generare dei **write miss**.

Esistono 2 soluzioni a questo problema:

- **Write Allocate:** il blocco viene caricato in cache, dove poi viene modificato;
- **No-Write Allocate:** il blocco viene scritto direttamente in memoria;