

Appunti di Davide Vella 2024/2025

Note importanti prima di leggere

Questo file è solamente la fusione di tutte e 16 le lezioni svolte nel corso. Sono presenti 17 mila parole (109 mila caratteri), è facile che ci siano errori di battitura (possono essere segnalati su [Github](#)). Inoltre, possono esserci :

- informazioni errate
- informazioni mancati
- etc etc

Ho scritto questi appunti prima per me e poi per poterli condividere e aiutare chiunque ne abbia bisogno. Non mi prendo alcuna responsabilità per cose che sono scritte male, informazioni mancanti o errate (ogni errore è segnalabile su [Github](#)). Buona lettura.

Appunti sul corso (ed esame 2024/2025)

Note : La prova di esame sarà corretta e valutata dal prof del proprio corso di appartenenza. Non si può cambiare corso. Prima di iscriversi ad un esame, si deve valutare il corso.

Esame :

- Esonero 1 : domande a risposta multiple (divisi in turni che verranno notificati). Punti 0-8 (minimo 4 punti per l'esonero 2).
- Esonero 2 : Esercizi di laboratorio (programmi Assembly RISC-V. Quiz su moodle come programmazione 1, quindi ce un esercizio scritto e c'è il simulatore sotto dove testare il programma scritto (anche se il programma supera tutti i test, non vuol dire che vale il massimo dei punti)). Punti 0-10 (minimo 5 punti con almeno 1 punto su ogni esercizio).
- Orale : colloquio su tutti gli argomenti (solitamente 2 domande). Punti 0-14 (non c'è un punteggio minimo, lo giudica il professore in base alla qualità dell'orale). Ci si può presentare agli orali altrui per vedere l'orale

Note :

- Serve studiare dal libro, le slide non bastano
- seguire le esercitazioni
- seguire le lezioni

Libro di testo : Struttura e progetto dei calcolatori di "David A. Patterson e John L. Hennessy", non è da studiare tutto, solo alcune parti che verranno trattate durante le lezioni. Queste parti verranno poi pubblicate (ovvero verranno detti gli argomenti trattati e che saranno richiesti).

Obiettivi del corso :

- Lo studio dell'architettura dei calcolatori
- come è organizzato un calcolatore

- RISC-V

Conoscenza di base richieste

Ricordare le potenze di 2 fino a 2^{12} :

Base	Esponente	Risultato
2	0	1
2	1	2
2	2	4
2	3	8
2	4	16
2	5	32
2	6	64
2	7	128
2	8	256
2	9	512
2	10	1024
2	11	2048
2	12	4096

Ricordarsi le conversioni tra basi (binario, ottale, esadecimale). Conoscenza di una semplice ALU e della somma binaria tra 3 variabili. Complemento (a 2, a 1...).

Somma delle codifiche

Prendendo come esempio il complemento a 2 come funzione $f(n)$, $f(n) + f(m) = f(n + m)$. Se ho 5 (000101) e -12 (110100) la loro somma è = 111001, ovvero -7 (pensa a 7 come 000111, in complemento a 2 è 111001).

Complemento a 2

Piccola nota, per fare il complemento a 2 di un numero basta :

- porto in binario il numero seguendo il numero di bit che devo usare
- se il numero è positivo, ho finito
- • se il numero è negativo, leggo da destra verso sinistra il numero, riscrivo tutto ciò che leggo fino al primo 1 e poi invertto tutti i numeri

☰ Example

$$7 = 111, \text{ con 6 bit diventa : } 000111$$

$$-7 = 111001$$

Estensione del segno

Passare da un numero in binario da n bit a m bit ($m > n$), copio il segno verso sinistra fino a riempire tutto lo spazio :

☰ Example

$$72 \text{ (su 8 bit)} = 01001000$$

$$72 \text{ (su 32 bit)} = 00000000 00000000 00000000 01001000$$

$$-102 \text{ (su 8 bit)} = 10011010$$

$$-102 \text{ (su 32 bit)} = 11111111 11111111 11111111 10011010$$

Architettura di Von Neumann

Un'unica memoria che contiene programma e dati in esecuzione (RAM), c'è una memoria di massa dove vengono messi i dati e i programmi che non vengono usati in quell'istante. C'è inoltre, ovviamente, la control unit.

Linguaggio macchina e Linguaggio Assembly

Il linguaggio macchina è il linguaggio più basso e più vicino alla macchina. È ciò che viene letto dalla CPU quando esegue le operazioni. È totalmente illeggibile a noi (solamente in binario). L'assembly è un linguaggio a basso livello, ma più alto rispetto al linguaggio macchina.

Strati (livelli) di astrazione

Livello	Tipo livello	Tramite cosa scende
4	Linguaggi alto livello	Compiler
3	Assembly	Assembler
2	OS	Interpretazione parziale
1	Istruction set architecture (ISA)	Esecuzione fisica sull'hardware
0	Digital Logic	Non esistono livello sottostanti

Durante il corso studieremo il livello 0, 1 e un po' di 3 (dove si faranno effettivamente i programmi).

Organizzazione della CPU in una macchina di Von Neumann

- CPU : si compone di diverse parti distinte :
- Unità di controllo
- Unità aritmetico-logica
- registri
- Registri : l'unità aritmetico-logica e alcuni bus che li collegano compongono il data path. È una parte hardware che costituisce la CPU. È molto limitata (pochi MB), è estremamente veloce, è

molto costosa. Viene chiamata tecnicamente "Cache". Una costruzione ad esempio è 8 registri da 4 MB l'uno (un totale di 32MB di cache).

- Registri importanti : Program Counter (PC) e Instruction Register (IR)
- Main memory : contiene sia istruzioni sia dati usando sequenze di bit.

CPU e i suoi componenti

Unità di controllo

Si occupa di capire di che tipo di istruzione si sta per fare e poi (dopo aver settato i bit di controllo in modo opportuno) viene eseguita l'istruzione.

ALU

Circuito combinatorio che si occupa di eseguire le istruzioni mandate dall'unità di controllo.

Registri

Program Counter :

Contiene un indirizzo (NON l'istruzione) della prossima istruzione. Da qui capiamo che il programma che vogliamo eseguire si trova sulla RAM. Il programma arriva alla RAM quando viene eseguito partendo dal disco rigido.

Program register :

Contiene il codice macchine dell'istruzione che stiamo eseguendo.

Controller

Circuito che si occupa di comunicare con una parte fisica. C'è ne uno per la RAM, uno per i display port, GPU... (nell'attuale configurazione e un'architettura specifica (questa detta sopra è Intel)). Una volta c'erano due controller, uno chiamato "North bridge" e "South Bridge".

Data path

È l'organizzazione interna

Avvio di un programma

Quando avviamo un programma, allochiamo lo spazio sulla RAM per :

- Il codice
- Lo stack
- L'heap

Bus

Un mezzo di comunicazione, formato da fili elettrici. Si occupa di far comunicare le varie parti hardware del computer (CPU, RAM, disco rigido...). La velocità del bus è data dalla larghezza del bus e dalla

velocità del clock. Quelli interni alla bus sono più veloce (generalmente) rispetto a quelli che collegano CPU, RAM...

Esecuzione delle istruzioni

La CPU esegue ogni istruzione del livello 1 (ISA) per mezzo di una serie di passi elementari :

1. Prendi l'istruzione seguente dalla memoria e la mette nel registro delle istruzioni
2. Cambia il program counter per indicare l'istruzione seguente
3. Determina il tipo dell'istruzione appena letta (unità di controllo)
4. Se l'istruzione usa una parola in memoria, determina dove si trova (non avviene nell'architettura RISC-V)
5. Metti la parola, se necessario, in un registro della CPU
6. Esegui l'istruzione (a dipendenza dell'istruzione può essere eseguita tutta dalla ALU o dalla ALU e un altro circuito)
7. Torna al punto 1 e inizia a eseguire l'istruzione successiva
(Per tutto il corso, si pensa di avere sempre 1 macchina, 1 cpu, 1 core).

Ruolo del software

- Organizzazione a livelli
- Software applicativo
- Software di sistema

Ciclo di vita del software

Esiste prima un programma. Tramite il compilatore diventa un programma assembly. Ora attraverso l'assemblatore diventa linguaggio macchina. Facendo "gcc -S (programma)" otteniamo un file .S, ovvero in assembly (che usa l'istruzione register per la CPU del computer che stiamo usando, noi useremo quella di RISC-V).

Prestazioni

Le prestazioni si misurano in :

- Tempo di esecuzione / tempo di risposta : tempo fra l'inizio e il completamento di un programma.
- Throughput o la larghezza di banda : il numero di task eseguiti nell'unità di tempo.
Queste cose sono influenzate sia dal processore, ma anche dal tipo di task che vogliamo eseguire.
Ancora possiamo vedere :
 - Tempo di esecuzione della CPU : il tempo che la CPU utilizza nella computazione richiesta da un certo task, che si divide in :
 - Tempo di CPU utente : tempo effettivo speso dalla CPU sulla computazione della richiesta di un programma
 - Tempo di CPU di sistema : tempo speso dalla CPU per eseguire le funzioni del sistema operativo richieste per l'esecuzione di un programma
 - Cloak : dispositivo che genera un segnale (onda quadra) che serve a sincronizzare i componenti.
 - Ciclo di clock : detto anche colpo, colpo di clock, periodi di clock

- Periodo/frequenza di clock : durata di un ciclo di clock (250 picosecondi o 4GHz).
Architetture diverse, prendono cicli di clock diversi per eseguire una stessa istruzione (CPI - cicli per istruzione). Una frequenza di clock più elevata rende più veloce l'esecuzione di un'istruzione. (ad esempio le istruzioni "load" hanno un tempo di esecuzione più elevato). Per calcolare il tempo di CPU facciamo :

$$\text{Tempo di CPU} = \frac{\text{numero di istruzioni} \times \text{CPI}}{\text{frequenza di clock}}$$

Limiti fisici

Esistono alcuni limiti per la CPU, non possiamo aumentare, ad esempio, la velocità di clock all'infinito.
Alcuni limiti sono :

- costi
- fisici
- potenza elettrica

RISC-V (reduced instruction set computer)

Proposto e sviluppato dal 2010 da : Andrew waterman, Yunsup Lee, Krste Asanovic. Gli obiettivi erano :

- hardware semplice
- compilatori efficienti
- massimizzare costi e prestazioni
- minimizzare il consumo energetico
- **Standard aperto** (molto importante)
RISC-V si adatta bene sia per microcontrollori si per supercomputer.

ISA

L'ISA è semplice e può essere estesa con delle estensioni :

estensione	descrizione
i	integer
m	
a	
f	
d	
g	
c	

Principi di progettazione

1. La semplicità favorisce la regolarità
2. Minori le dimensioni, maggiore è la velocità
3. Un buon progetto richiede buoni compromessi

RISC-V ha poche istruzioni e semplici, che però se vengono ben usate posso eseguire tutti i programmi. CISC al contrario ha tante istruzioni, anche complesse, che possono eseguire sì tutti i programmi, ma prendono più tempo.

Nota

RISC-V non è un'azienda, né una CPU, è una struttura di CPU.

Registri e memoria

RISC-V ha la seguente struttura :

- 32 registri per gli interi (es : RV64I ha i registri a 64 bit, per RV32I (quella che viene usata durante il corso) sono a 32 bit). Questi registri si chiamano x_0, x_1, \dots, x_{31} .
- 32 registri per i numeri in virgola mobile
- Registro program counter
- 4096 control status registers
- Memoria centrale (ad esempio, 2^{64} indirizzi nella RV64I, 2^{32} indirizzi nella RV32I) (ogni indirizzo ha n)

Ruoli dei registri

I registri hanno dei ruoli specifici :

- x_0 : zero (utili per ottimizzare, molte volte inizializziamo una variabile a 0, questo lo rende più veloce).
- x_1 : return address
- ...
- $x_{10}-x_{17}$: registri usati per il passaggio di parametri nelle procedure e valori di ritorno
- $x_5-x_7, x_{28}-x_{31}$: registri temporanei, non salvati in caso di chiamata
- $x_8-x_9, x_{18}-x_{27}$: registri ... il contenuto va preservato se utilizzati dalla procedura chiamata

Istruzioni aritmetiche

Tutte le istruzioni aritmetiche hanno esattamente 3 operando, l'ordine di questi operando è fisso. Alcuni esempi :

- $a = b + c$ in C, diventa "add a, b, c" in RISC-V assembly
 - $a = b - c$ in C, diventa "sub a, b, c" in RISC-V assembly
- Nota, sopra in realtà non esistono variabili, ma i registri, quindi (ad esempio) :
- $a = b + c$ in C, diventa "add x5, x20, x21" in RISC-V assembly
 - $a = b - c$ in C, diventa "sub x5, x20, x21" in RISC-V assembly

Nota

- byte : 8 bit
- half word : 16 bit
- word : 32 bit
- doubleword : 64 bit

Associazione registri/variabili

Per passare da un linguaggio ad alto livello ad assembly, dobbiamo passare dalle variabili (come "tmp, a, b, c, somma,...") a indirizzi dei registri ("x5, x20, x21, ...").

Istruzioni aritmetiche

- Somma : add
- Sottrazione : Può succedere che si incontri la necessità di cambiare segno ad un valore.
L'istruzione "sub" può essere utilizzata per questa cosa. Dando $x19 = a$, da $a = -a$, diventa (in assembly) "sub x19, x0, x19" (così diventa $a = 0 - a$). La sottrazione avviene grazie al complemento a 2 del secondo operando e poi avviene la somma (inv(...) per invertire un registro)
Ogni istruzione di alto livello, può essere riscritta come più istruzioni di basso livello, ad esempio :

```
f = a + b - c
```

Diventa :

```
add x19, x20, x21
sub x19, x19, x22
```

Quindi facciamo prima la somma tra a e b (rispettivamente x20, x21) e messo in f (x19) e poi facciamo $f = f - c$ ($c = x22$)

Esercizio :

```
f = (g + h) - (i + j)
```

Var	Reg
f	x20
g	x21
h	x22
i	x23
j	x24

```
add x21, x21, x22
add x23, x23, x24
sub x20, x21, x23
```

```
g = g + h;  
i = i + j;  
f = g - i;
```

Accesso alla memoria

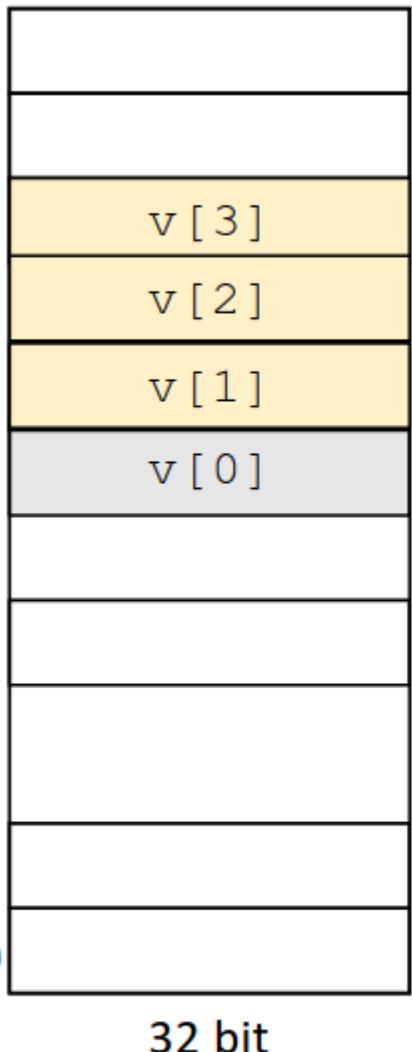
La memoria può essere vista come un grande array di celle (da 1 byte l'una). Prendiamo come esempio il seguente schema :

Valori	Cella
	...
	...
	...
	5
	4
	3
00111010	2
10111011	1
11111110	0

Istruzione Load

L'istruzione load permette di copiare un dato dalla memoria ad un registro. Prendiamo il seguente esempio :

Memoria

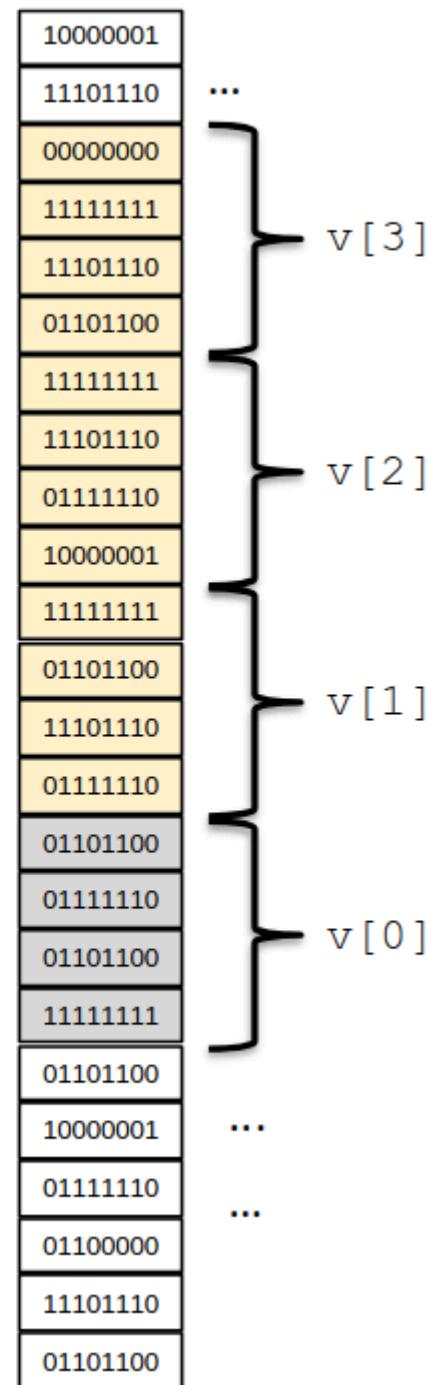


$x21+3*4$
base + offset
 $v \rightarrow x21$ base

...

...

8 (byte 8-11)
4 (byte 4-7)
0 (byte 0-3)



Se cerchiamo di fare

```
a = v[3];
```

Dobbiamo usare l'istruzione **lw** (load word) nel seguente modo :

lw (destinazione della lettura), offset (indirizzo di base)

Un esempio è :

Var	registri
a	x5
v (array int)	x21

```
lw x5, 12(x21)
```

L'offset di 12 ci serve perché dobbiamo andare al primo indirizzo di dove è salvato il valore v(3). Prendiamo lo spazio occupato da un int (4 byte) e moltiplichiamo per l'indirizzo a cui vogliamo arrivare (3), quindi abbiamo 12. **NOTA**, x21 in questo caso è un registro che contiene un indirizzo. L'offset può valere al massimo -2048 o +2047.

little endian

L'indirizzo della parola identifica il byte meno significativo

Istruzione store

Per scrivere in memoria usiamo l'istruzione **sw** (store word). La notazione è : sw (dato da scrivere in memoria), offset(indirizzo di base).

Istruzione sh

Come sw, sh serve per scrivere, ma a differenza di sw (che scrive 4 byte (1 word)), con sh scriviamo una halfword (2 byte), ovvero i meno significativi, sintassi : sh (dato da scrivere in memoria), offset(indirizzo di base).

esercizio scrittura/lettura

```
g = h + v[3]
v[6] = g - f
```

Var	Reg
g	x5
h	x9
f	x19
v (array)	x21

```
lw x2, 12(x21)
add x5, x9, x2
sub x2, x5, x19
sw x2, 24(x21)
```

In questo caso, usiamo x2 come registro d'appoggio.

Esempio

```
int d,i,j;
int v[10];
...
j=5
```

```
...  
v[i+d] = v[j+2];
```

d = x12
i = x9
j = x21
v = x19

```
addi x21, x0, 5  
addi x21, x21, 2  
slli x21, x21, 2  
add x21, x21, x19  
lw x21, 0(x21)
```

```
add x9, x9, x12  
slli x9, x9, 2  
add x9, x9, x19  
sw, x21, 0(x9)
```

operazioni logiche

AND

```
and x9, x22, x19
```

ovvero, : $x9 = x22 \& x19$

```
andi x9, x22, 5
```

OR

```
or x9, x22, x19
```

ovvero, : $x9 = x22 | x19$

```
ori x9, x22, 5
```

XOR

```
xor x9, x22, x19
```

ovvero, : $x9 = x22 \oplus x19$

```
xori x9, x22, 5
```

NOT

```
not x9, x22
```

ovvero, : $x9 = !x22$. È una pseudoistruzione, ovvero un'istruzione che non esiste in realtà e viene interpretata come un'altra, ovvero :

```
xori x9, x22, -1
```

Uno xor tra x22 e "1111....1111".

Formati usati dalle operazioni logiche

Le istruzioni "and, or e xor" si rappresentano in linguaggio macchina con il formato R. La "andi, ori, xori" si rappresentano in linguaggio macchina con il formato I.

Usi delle operazioni logiche

- OR : usato per settare alcuni bit specifici.
- AND : selezionare alcuni bit specifici, sapere che valore hanno quei bit.
- XOR : ---

Salti condizionali

Permettono di variare il flusso del programma (variando il valore del PC) al verificarsi di una condizione.

Istruzioni di salto

Salto incondizionato

```
j L1
```

Salta in modo incondizionato all'etichetta L1.

Salto condizionato

beq

```
beq rs1, rs2, L1
```

Si salta all'etichetta L1 se il valore del registro rs1 è uguale al valore del registro rs2.

bne

```
bne rs1, rs2, L1
```

Si salta all'etichetta L1 se il valore del registro rs1 è diverso dal valore del registro rs2.

blt

Si salta all'etichetta L1 se il valore del registro rs1 è minore a quello di rs2

bge

Si salta all'etichetta L1 se il valore del registro rs1 è maggiore a quello di rs2

bltu

Si salta all'etichetta L1 se il valore del registro rs1 (unsigned) è minore a quello di rs2 (uns.)

bgeu

Si salta all'etichetta L1 se il valore del registro rs1 (unsigned) è maggiore a quello di rs2 (uns.)

Nota

Non esiste "bequ, bneu", perché tanto stiamo già controllando se sono uguali (o diversi), prenderli unsigned non cambia.

Etichette

Una semplice parola che identifica un certo punto del codice. Si scrive :

```
addi x10, x0, 10
addi x9, x0, 0
loop :
addi x9, x9, 1
bne x9, x10, loop
```

in questo caso il codice dopo "loop" viene eseguito 10 volte. Il codice eseguito è semplicemente un incremento di 1 al valore di x9.

L'etichetta contiene l'offset tra l'istruzione da cui viene USATA (in questo caso quindi da "bne x9, x10, loop") e la prima istruzione dopo l'etichetta (in questo caso quindi, da "addi x9, x9, 1"). Questo offset può essere sia negativa che positiva, perché si può andare sia avanti che indietro.

Esempio salti

Esempio 1

```
if (i == j)
    f = g + h;
else
    f = g - h;
```

f = x19

g = x20

```
h = x21
i = x22
j = x23
```

```
sub x19, x20, x21
beq x22, x23, if
j else //o beq x0, x0, else
if
add x19, x20, x21
else
...
...
```

oppure

```
bnq x22, x23, else
add x19, x20, x21
beq x22, x23, endif
else
sub x19, x20, x21
endif
//altro codice
```

Esempio 2

```
for (int i = 0; i < 100; i++) {
```

```
}
```

```
addi x10, x0, 100
addi x19, x0, 0 //i
for :
bge x19, x10, endfor
...
addi x19, x19, 1
bne x19, x10, for
endfor :
```

Salti condizionati

```
int v[10], k, i;
while (v[i] == k) {
    ...
    i = i + 1;
}
```

```
i = x22
k = x24
v = x25
```

```
while :  
    slli x23, x22, 2  
    add x23, x23, x25  
    lw x26, 0(x23)  
    bne x26, x24, endwhile  
    ...  
    addi x22, x22, 1  
    j while  
endwhile :
```

Nota sui salti

Nelle condizioni, è sempre meglio usare la condizione opposta a quella che dobbiamo "tradurre".

✓ Esempio

```
while (v[i] == k) {  
    ...  
}
```

Non controlliamo per "v[i] == k", ma controlliamo "v[i] != k"

```
bne x26, x24, endwhile //dove x26 contiene v[i] e x24 k
```

SLT

Istruzione che scrive 1 in rd se $rs1 < rs2$. È di tipo R.

```
slt x21, x19, x20
```

Si può pensare di usare dopo slt l'istruzione "beq rs1, x0, L1" per saltare ad un certo punto se rs1 è minore di rs2.

☰ Esempio

```
if (i < j)  
    k = 1;  
else  
    k = 0;
```

i = x19

j = x20

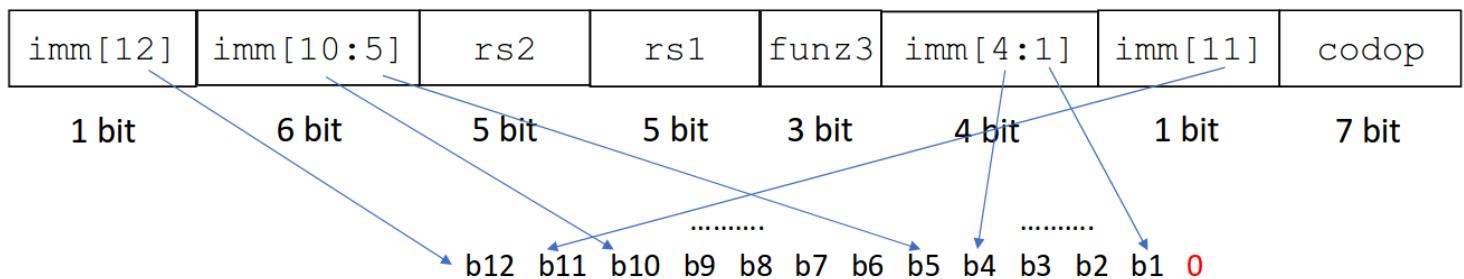
k = x21

```

slt x21, x19, x20 //salviamo in k il risultato di i < j
beq x21, x0, L1 //saltiamo a L1 se x21 == 0, dove x21 = 0 quando rs1 > rs2

```

Formato SB



- codop : codice operazione
 - imm[11] : contiene una parte dell'offset
 - imm[4:1] : (dal 1 al 4) contiene una parte dell'offset
 - funz3 : differenzia le varie istruzioni di salto
 - rs1/rs2 : registri di input
 - imm[10:5] : (dal 5 al 10) contiene una parte dell'offset
 - imm[12] : contiene una parte dell'offset
- Il formato può rappresentare indirizzi di salto da -4096 a +4094, in multipli di due.

Moltiplicazione

Per poter moltiplicare il valore di due registri si può usare (in alcuni casi, non sempre è implementata) l'istruzione "mul".

In C :

$$a = b * c$$

In assembly :

```
mul x10, x12, x12
```

Questo potrebbe essere un problema in caso in cui il risultato della moltiplicazione in overflow per un registro. Per questo si può usare "mulh", la quale usa 2 registri.

Divisione

Ugualmente per sopra, si può usare l'istruzione "div" per dividere il valore di due registri.

Procedure (funzioni)

Porzioni di codice associate ad un nome che possono essere invocate più volte e che eseguono un compito specifico, avendo come input una lista di parametri e come output un valore di ritorno.

Pro

- Astrazione

- Riusabilità del codice
- ..
- ..

Esempio C

Programma (procedura) (chiamante) :

```
f = f + 1;
risultato = somma(f, g);
```

Procedura somma() (chiamata) :

```
int somma(int x, int y) {
    int rst;
    rst = x + y;
    return rst;
}
```

Si può pensare di implementare la funzione in assembly con dei jump e delle etichette. Questo non sarebbe sbagliato, però dobbiamo salvarci anche il punto di ritorno (return rst e deve diventare un jump a chiamante).

Passi da seguire

Chiamante :

- Mettere i parametri in un luogo accessibile alla procedura
 - Trasferire il controllo alla procedura
- Chiamato :
- Acquisire le risorse necessarie per l'esecuzione della procedura
 - Eseguire il compito richiesto
 - Mettere il risultato in un luogo accessibile al programma chiamante
 - Restituire il controllo al punto di origine

Istruzione di invocazione : jal

Per saltare all'etichetta usiamo "jal" :

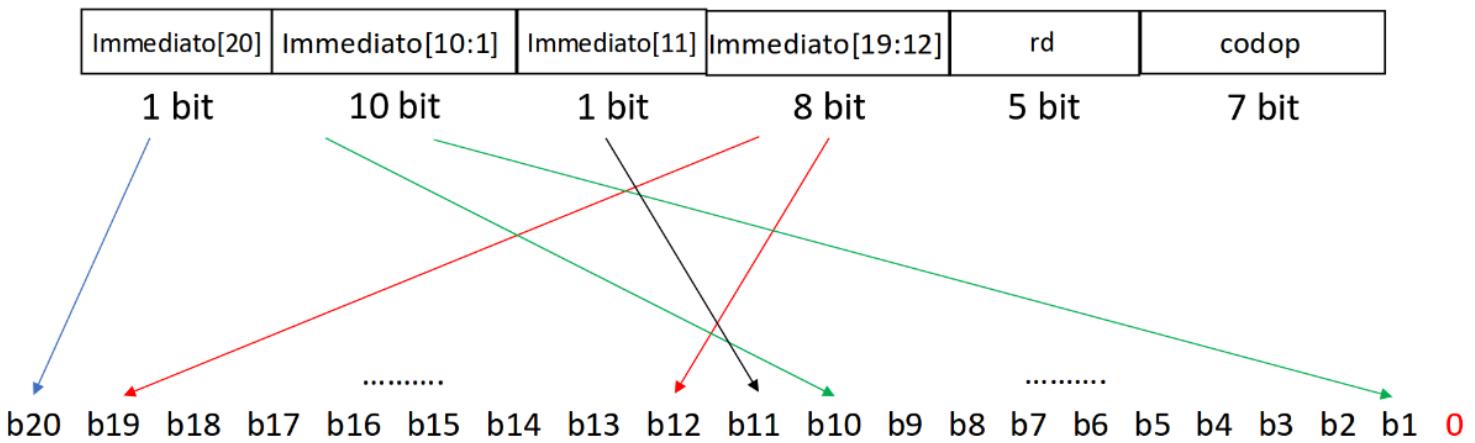
jal IndirizzoProcedura

In realtà "jal" è una pseudo istruzione, la vera istruzione è :

jal x1, IndirizzoProcedura

Dove x1 è il registro dove salviamo l'indirizzo della istruzione a cui saltare quando finisce la procedura. Si usa sempre x1, nessun'altra.

Formato J



- codop :
- rd :
- imm[19:12]
- imm[11]
- imm[10:1]
- imm[20]

Istruzione ritorno al chiamante : jalr

Salta ad un indirizzo qualsiasi :

jalr rd, offset(rs1)

Per tornare indietro quindi facciamo :

jalr x0, 0(x1)

Problemi

1. Si deve fare attenzione, i registri sono condivisi, non c'è spazio infinito. Può succedere che il registro x5 contiene un valore usato dal chiamante per qualcosa, ma magari il chiamato usa x5 per mettere il risultato. Come si può rimediare al problema? Possiamo salvare il valore di x5 in memoria (con l'istruzione sw) prima di utilizzarlo e poi lo ripristiamo alla fine (sw/lw sono istruzioni oerose, quindi è meglio evitarle).
2. Se chiamiamo una procedura dentro una procedura, andiamo a sovrascrivere x1, quindi non sappiamo più dove tornare. Come possiamo rimediare al problema? Di nuovo, possiamo salvare il valore di x1 prima di chiamare la procedura e dopo che la procedura finisce, possiamo caricare il valore che abbiamo salvato prima.
3. Può succedere che i parametri e le variabili di una procedura superano il numero di registri disponibili. Come facciamo? Salviamo di nuovo in memoria.

Stack

Area di memoria dinamica. Per accedere si usa il "Last In First Out"

Operazioni

- PUSH : aggiunge un elemento in cima allo stack.

- POP rimuove un elemento dalla cima dello stack.
- SP : la cima delle stack è identificato dallo stack pointer.

Convenzione RISC-V

- Grow-down : lo stack cresce da indirizzi di memoria alti verso indirizzi di memoria bassi
- last-full : lo stack pointer (SP) contiene l'indirizzo dell'ultima cella di memoria occupata nello stack,
- Il valore di SP è salvato nel registro x2 (o sp).

Push

```
addi sp, sp, -4 // -4 perché vogliamo salvare una word, sennò -2 se una hw...
sw x20, 0(sp)
```

Pop

```
lw x20, 0(sp)
addi sp, sp, +4 // -4 perché vogliamo salvare una word, sennò -2 se una hw...
```

Esempio

```
int somma(int x, int y) {
    int rst;
    rst = x + y + 2;
    return rst;
}
...
f=f+1
risultato=somma(f,g)
...
```

x = x10
y = x11
rst = x20
f = x6

```
somma :
addi sp, sp, -8 //allochiamo due interi nello stack
sw x5, 0(sp)
sw x20, 4(sp) //salviamo due interi nella memoria allocata prima
add x5, x10, x11
addi x20, x5, 2
addi x10, x20, 0
lw x5, 0(sp)
lw x20, 4(sp)
addi sp, sp, 8 //liberiamo la memoria
jalr x0,0(x1)
...
```

```

...
addi x6, x6, 1
...
jal somma

```

Convenzione sui registri

Registro	Nome	Utilizzo
x0	zero	Costante zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Puntatore a thread
x8	s0 / fp	Frame pointer (il contenuto va preservato se utilizzato dalla procedura chiamata)
x10-x11	a0-a1	Passaggio di parametri nelle procedure e valori di ritorno
x12-x17	a2-a7	Passaggio di parametri nelle procedure
x5-x7 x28-x31	t0-t2 t3-t6	Registri temporanei, non salvati in caso di chiamata
x9	s1	Registri da salvare: il contenuto va preservato se utilizzati dalla procedura chiamata
x18-x27	s2-s11	

Chi è responsabile di salvare i registri quando si effettuano chiamate di funzioni?

- La funzione chiamante conosce quali registri sono importanti per sé e che dovrebbero essere salvati
 - La funzione chiamata conosce quali registri userà e che dovrebbero essere salvati prima di modificarli
- Bisogna evitare le inefficienze → Minimo salvataggio dei registri:
- La funzione chiamante potrebbe salvare tutti i registri che sono importanti per sé, anche se la procedura chiamata non li modificherà
 - La funzione chiamata potrebbe salvare tutti i registri che si appresta a modificare, anche quelli che non verranno poi utilizzati dalla procedura chiamante una volta che la procedura chiamata le avrà restituito il controllo

Convenzione nell'uso e salvataggio dei registri

Spiegazione del problema

1. Chi è responsabile di salvare i registri quando si effettuano chiamate di funzioni? Dipende :
 - La funzione chiamata conosce i registri che essa userà, quindi prima dell'uso li salva, li usa e poi carica i vecchi valori dei registri.
 - La funzione chiamante conosce quali registri sono importanti per essa, quindi li salva prima di chiamare una funzione.
2. Bisogno evitare le inefficienze, ovvero salvare il minor numero di volte qualcosa nei registri (sw, lw, ...) :

- La funzione chiamante potrebbe salvare tutti i registri che sono importanti per sé, anche se la procedura chiamata non li modificherà.
 - La funzione chiamata potrebbe salvare tutti i registri che si appresta a modificare, anche quelli che non verranno poi utilizzati dalla procedura chiamante una volta che la procedura chiamata le avrà restituito il contenuto.
- Entrambe queste soluzioni vanno quasi bene, ma non sono ben ottimizzati, allora si può fare un altro modo, la convezione.

Convenzione (IMPORTANTE)

1. I registri x10-x17 (a0-a7), x5-x7 e x28-x31 (t0-t6) :
 - Possono essere modificati dal chiamato senza nessun meccanismo di ripristino.
 - Il chiamante se necessario dovrà salvare i valori dei registri prima dell'invocazione della procedura.
2. I registri x1 (ra), x2(sp), x3 (gp), x4 (tp), x8 (fp/s0), x9 e x18-x27 (s1-s11) :
 - Se modificati dal chiamato devono essere salvati e poi ripristinati prima del ritorno al chiamante.
 - Il chiamante non è tenuto al loro salvataggio e ripristino.

Fasi di invocazione di procedura

Fase 1 - Pre chiamata del chiamante

Se uno (o più) dei registri che sono allocati alle procedure (x10-x17 (a0-a7), x5-x7 e x28-x31 (t0-t6)) contiene qualcosa che ci serve, dobbiamo salvarlo prima di chiamare la procedura.

Preparazione degli argomenti (parametri) della procedura

I primi 8 argomenti vengono posti in x10-x17 (a0-a7). Eventuali altri argomenti vanno salvati nello stack (EXTRA_ARGS), così che si trovino subito sopra il frame della funzione che chiamata.

Fase 2 - Invocazione della procedura

Usiamo l'istruzione "jal NOME_PROCEDURA".

Ricorda, "jal" vuol dire "jump and link". Salta alla procedura e poi linka, imposta il PC.

Fase 3 - Prologo lato chiamato

Eventuale allocazione del call-frame sullo stack (aggiornare sp) :

- Salvataggio di x1 (ra) nel caso in cui la procedura non sia foglia (ovvero una procedura che non chiama altre procedure)
- Salvataggio di x8 (fp), solo se utilizzato all'interno della procedura
- Salvataggio di x9 e x18-x27 (s1-s11) se utilizzati all'interno della procedura (il chiamante si aspetta di trovarli intatti)
- Salvataggio degli argomenti x10-x17 (a0-a7) solo se la funzione li riuserà successivamente a ulteriori chiamate a funzione (procedure innidate)
- Eventuale inizializzazione di fp : punta al nuovo call-frame

Fase 4 - Corpo della procedura

La procedura fa quello che deve fare.

Fase 5 - Epilogo lato chiamato

1. Se si deve restituire un valore, lo si salva in x10-x11 (a0-a1).
2. I registri (se salvati) devono essere ripristinati :
 - x9e x18x-x27 (s1-s11)
 - x1 (ra)
 - x8 (fp)
3. SP devo solo aumentare dell'opportuno offset.

Fase 6 - Ritorno al chiamante

Istruzione "jalr x0, 0(x1) (o pseudo-istruzione jr ra).

Fase 7 - Post chiamate lato chiamante

- Eventuale uso del risultato della funzione in x10 e x11 (a0-a11).
- Ripristino dei valori x5-x7 e x28-x31 (t0-t6), x10-x17 (a0-a7) vecchi, se erano stati scritti prima di chiamare la funzione (procedura).

Fasi di una invocazione di una procedura foglia

Le fasi descritte precedentemente si applicano a tutti i tipi di procedura, ma possono essere semplificate, ovvero, se una procedura è foglia (ovvero non chiama a sua volte procedure), nel codice della procedura :

- non è necessario salvare e ripristinare il registro "ra".
- non è necessario salvare sullo stack i registri a0-a7 e t0-t6 (non c'è nessuna invocazione di procedura nella procedura chiamata).

Fase 1 - Pre chiamata del chiamante

- In caso siano usati del chiamante, si devono salvare i registri (solo quelli usati) tra a0-a7 e t0-t6 (potrebbero essere sovrascritti dal chiamato). Vanno salvati nello stack.
- Si devono salvare gli argomenti (o parametri) della funzione. Fino a 8 argomenti possono essere salvati in a0-a7, gli altri vanno salvati nello stack in modo tale che si trovino subito sopra al frame della funzione chiamata.

Fase 2 - invocazione della procedura

Istruzione jal NOME_ISTRUZIONE.

Fase 3 - prologa lato chiamato

Eventuale allocazione del call-frame sullo stack

- salvataggio di x8 (fp), solo se utilizzato all'interno della procedura
- Salvataggio di x9 e x18-x27 (s1-s11) se utilizzati all'interno della procedura (il chiamante si aspetta di trovarli intatti)
- Eventuale inizializzazione di fp L punta al nuovo call-frame

Fase 4 - Corpo della procedura

Esecuzione effettiva della procedura.

Fase 5 - Epilogo lato chiamato

- In caso si debba restituire un output, lo si salva in a0/a1.
- In caso si siano utilizzati (e quindi per forza salvati per quanto visto sopra), si ripristinano i registri s1-s11 e fp.

Fase 6 - ritorno al chiamante

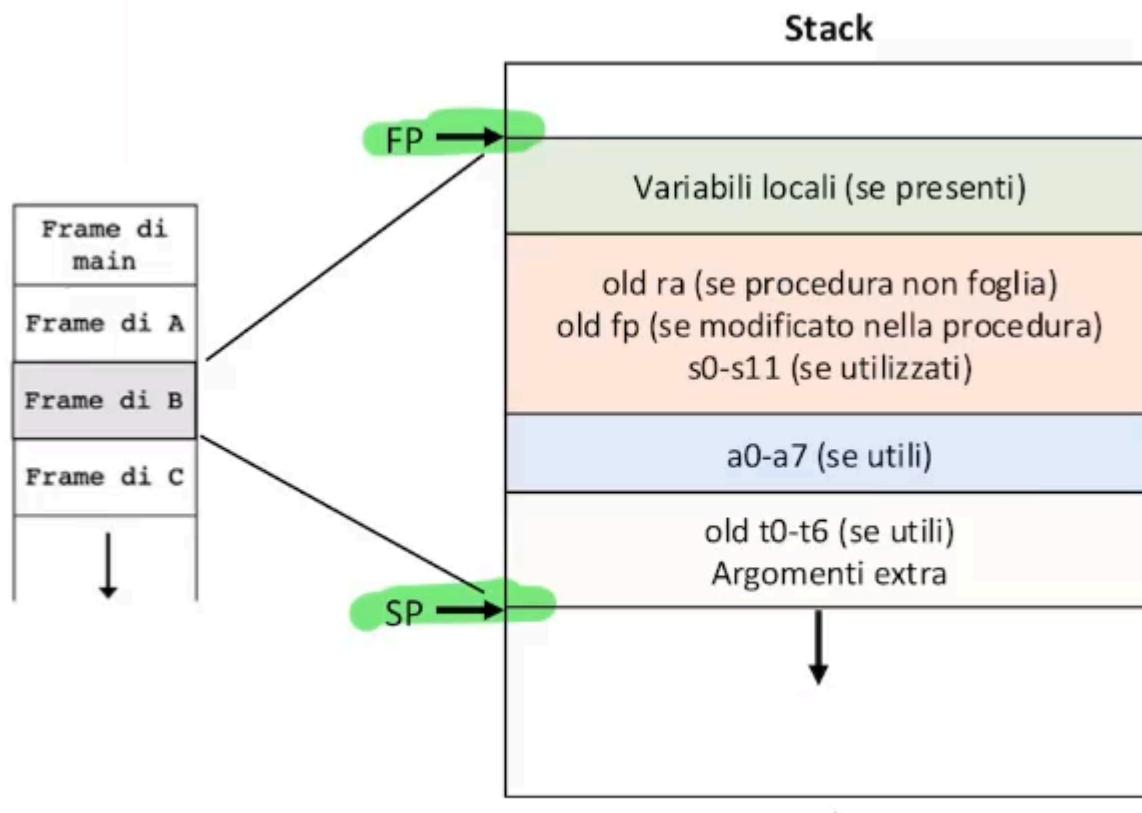
Istruzione jalr x0, 0(x1).

Fase 7 - post chiamata lato chiamante

- Eventuale uso del risultato della funzione (salvato in a0/a1).
- Ripristino dei registri a0-a7 o/e t0/t6 se usati prima di chiamare la funzione.

Record di attivazione - struttura

Fp : se utilizzato, viene inizializzato al valore di sp all'inizio della chiamata. Fp permette di avere un riferimento alle variabili locali che non muta con l'esecuzione della procedura.



Lezione 8

Operandi immediati ampi

Problema : è possibile caricare in un registro una costante a 32 bit?

LUI

Load upper immediate, tipo U. Carica 20 bit più significativi della costante nei bit da 12 a 31 di un registro. Perché è importante? Perchè la semplice LI carica solo i primi 12 bit di un registro (valori da -2048 a +2047 infatti).

Altro problema, abbiamo messo i 20 bit più significativi sì, ma manca i primi 12, come si va? Basta usare un ORI (or immediate). I bit già impostati dalla LUI non vengono modificati e i primi 12 bit a 0 prendono il giusto valore.

lui x5, 0x12345	x5	00010010 00110100 01010000 00000000
-----------------	----	-------	-------------------------------------

ori x5, x5, 0x678	x5	00010010 00110100 01010110 01111000
-------------------	----	-------	-------------------------------------

Salti con offset grandi

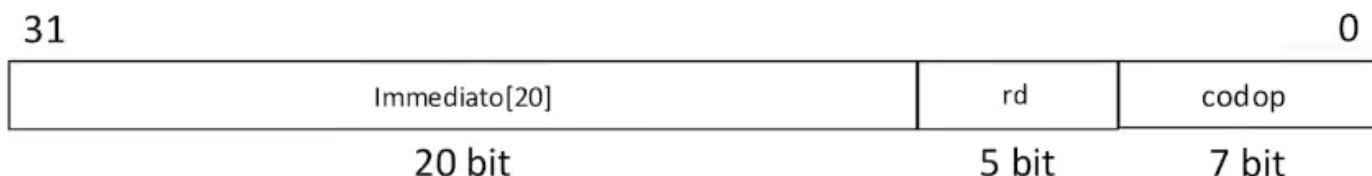
In RISC-V c'è la possibilità di saltare in un intervallo pari a 2^{32} grazie all'istruzione "auipc" (Add Upper Immediate PC) :

auipc rd, offset

Example

auipc x5, 0x12345	→	x5 = PC + 0x12345000
-------------------	---	----------------------

Formato U



- immediato : 20 bit, quelli che vanno sommati al PC
- rd : register destination, da dove prendiamo i 20 bit
- codOp : codice univoco della operazione

Riassunto dei formati visti (importante per l'esame)

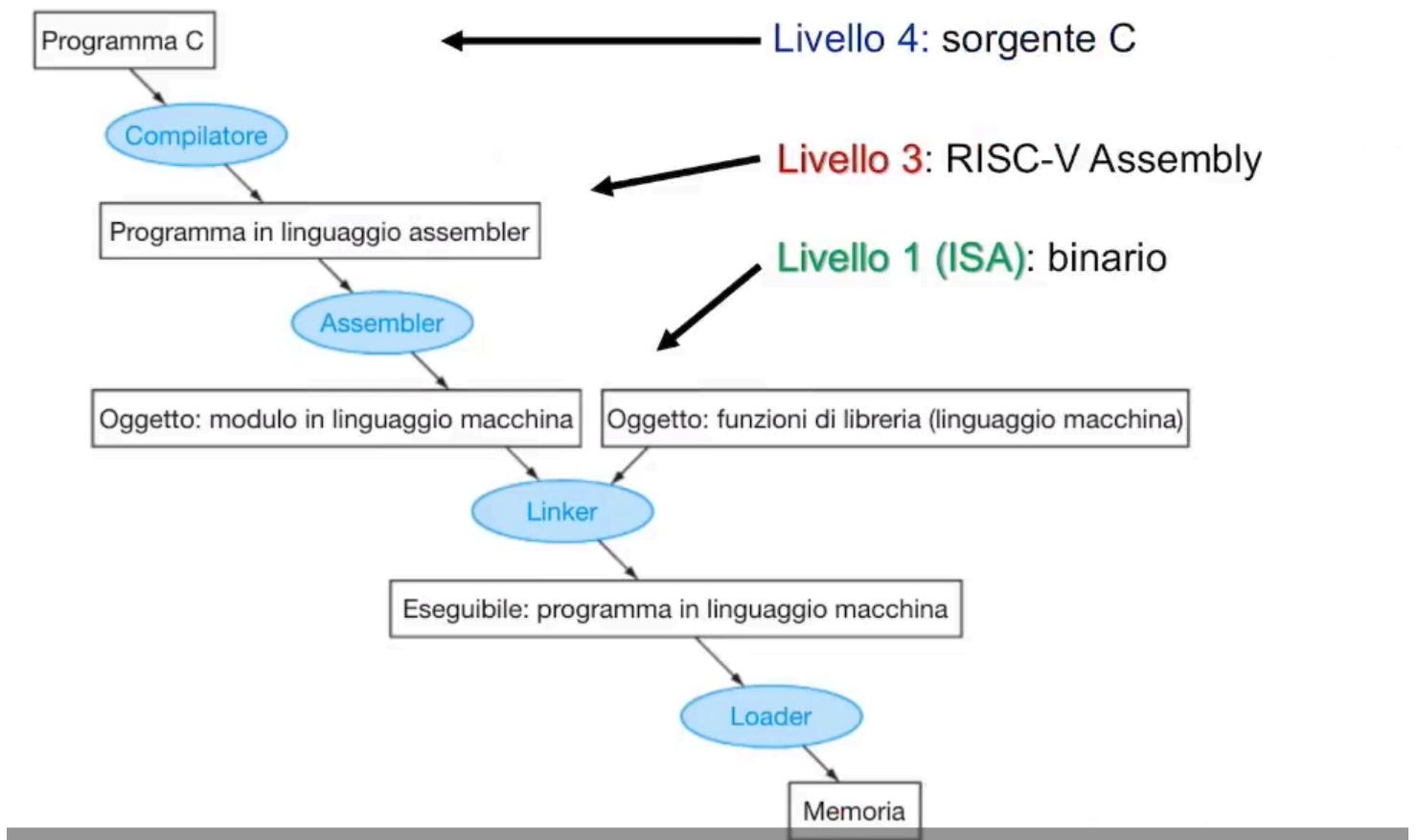
Nome (dimensione del campo)	Campi						Commenti
	7 bit	5 bit	5 bit	3 bit	5 bit	7 bit	
Tipo R	funz7	rs2	rs1	funz3	rd	codop	Istruzioni aritmetiche
Tipo I	Immediato[11:0]		rs1	funz3	rd	codop	Istruzioni di caricamento dalla memoria e aritmetica con costanti
Tipo S	immed[11:5]	rs2	rs1	funz3	immed[4:0]	codop	Istruzioni di trasferimento alla memoria (store)
Tipo SB	immed[12, 10:5]	rs2	rs1	funz3	immed[4:1,11]	codop	Istruzioni di salto condizionato
Tipo UJ	immediato[20, 10:1, 11, 19:12]				rd	codop	Istruzioni di salto incondizionato
Tipo U	immediato[31:12]				rd	codop	Formato caricamento stringhe di bit più significativi

⚠ Warning

Non si devono ricordare a memoria tutte le posizioni di tutti i bit, ma è importante ricordare le considerazioni fatte. Ad esempio : "Perché il registro sorgente sta sempre nella stessa posizione? Per ragioni strutturali fisiche, è più facile costruire così", "Perché i campi dei registri sono grandi 5 bit? Ci sono 32 registri in tutto, $\log_2 32 = 5$, ovvero, ci bastano 5 bit per rappresentare tutti e 32 i registri in modo univoco", "perché codOp è sempre nella stessa posizione? CodOp, in modo particolare, è sempre nella stessa posizione perché è un campo in comune tra tutte le operazioni di tutte i formati, quindi è comodo averlo sempre nella stessa posizione in modo da sapere sempre dove si trova", "Perché l'immediato è grande 12 bit? Cosa significa avere 12 bit di immediato in un'istruzione di salto? In una di tipo I?" ...

Sequenza di passi di traduzione per il C

Sequenza di passi di traduzione per il C



Assemblatore (o Assembler)

L'assemblatore è un software che si occupa di trasformare il programma in assembly (a sua volta tradotto dal compilatore) in un modulo oggetto, cioè un file che contiene la codifica in linguaggio macchina delle istruzioni che abbiamo scritto in linguaggio mnemonico.

Linker

Software utilizzato dopo l'assemblatore per linkare i moduli in linguaggio macchina con le funzioni di librerie (linguaggio macchina) chiamate nel codice scritto. Un esempio può essere la printf() o la scanf(). Fatto ciò, il linker produce un eseguibile.

Loader

Software che si occupa di caricare l'eseguibile prodotto dal linker.

☰ Example

0000000000010078 <scambia>:

10078:	00359313	slli	x6,x11,0x3
1007c:	932a	c.add	x6,x10
1007e:	00033283	ld	x5,0(x6)
10082:	00833383	ld	x7,8(x6)
10086:	00733023	sd	x7,0(x6)
1008a:	00533423	sd	x5,8(x6)
1008e:	00008067	jalr	x0,0(x1)

0b 0000000 00111 00110 011 00000 0100011

Livello 1 (ISA):
sequenza di byte
in memoria

Istruzione	Formato	immediato	rs2	rs1	funz3	immediato	codop
sd (memorizzazione di parola doppia)	S	indirizzo	reg	reg	011	indirizzo	0100011

Un esempio di come vengono tradotte le istruzioni (i codici sulla sinistra sono scritti in esadecimale)

Nota sul formato binario

Ovviamente, architetture diverse usano formati binari diversi, quindi incompatibili. Di seguito un esempio chiaro tra RISC-V e MIPS ISA :

Registro-registro

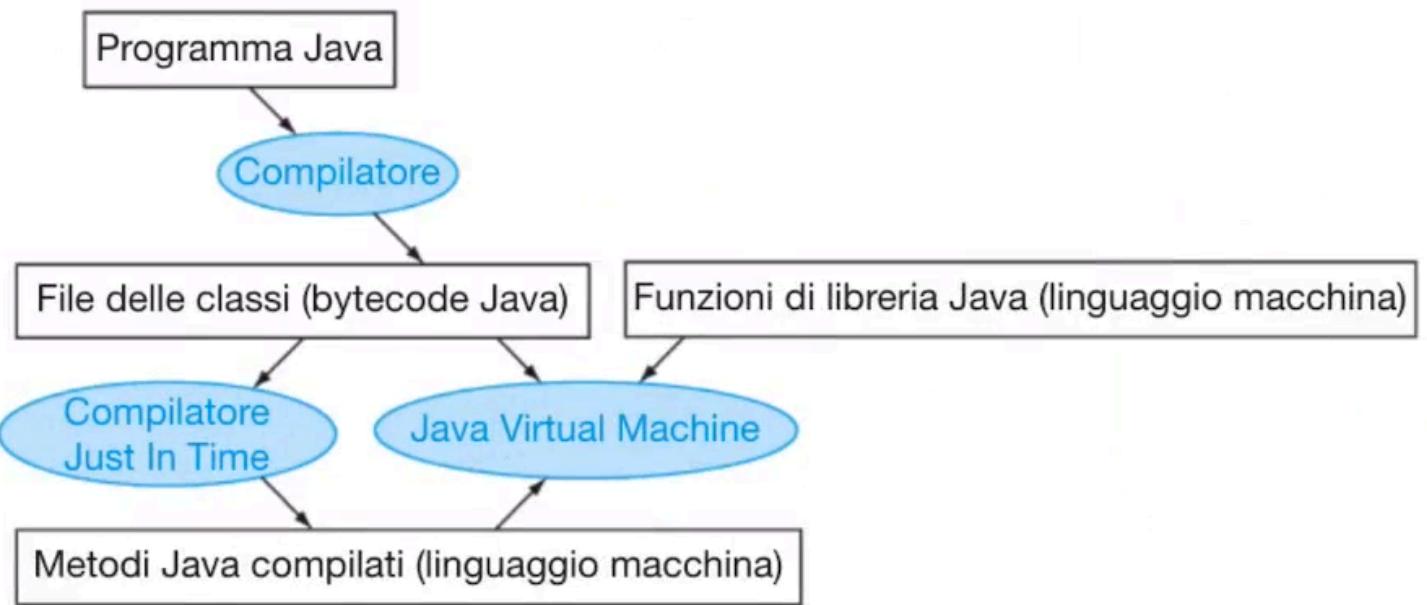
	31	25 24	20 19	15 14	12 11	7 6	0
RISC-V		funz7(7)	rs2(5)	rs1(5)	funz3(3)	rd(5)	codop(7)
MIPS	31	26 25	21 20	16 15	11 10	6 5	0
	Op(6)	Rs1(5)	Rs2(5)	Rd(5)	Cost(5)	Opx(6)	

Trasferimento dalla memoria

	31	20 19	15 14	12 11	7 6	0
RISC-V		immed(12)	rs1(5)	funz3(3)	rd(5)	codop(7)
MIPS	31	26 25	21 20	16 15		0
	Op(6)	Rs1(5)	Rs2(5)		Cost(16)	

Nota secondaria (non necessaria)

Linguaggi diversi hanno maniere diverse di venire tradotte, etc etc... Ad esempio, java non ha la stessa sequenza di passi di traduzione del C. Java usa una macchina virtuale per far girare il programma, quindi in Java avremo una struttura del genere :



Il programma in Java è quindi eseguito da un interprete (JVM - Java Virtual Machine), la quale può invocare il compilatore (JIT - Just In Time), il quale compila i metodi del linguaggio Java nel linguaggio macchina del calcolatore sul quale è in esecuzione. Ovviamente questo porta dei vantaggi sì, ma anche dei grandi svantaggi (ovvero l'inefficienza rispetto ad un programma scritto in C).

Linguaggio Assembly

Quando si parla di linguaggio Assembly si intende un linguaggio le cui istruzioni sono ottenute dalle istruzioni ISA sostituendo i codici binari con codici mnemonici; Il linguaggio Assembly è quindi molto vicino al linguaggio macchina. C'è sostanzialmente una corrispondenza uno-uno tra le istruzioni ISA e le istruzioni del linguaggio Assembly.

Ci sono delle facilitazioni nell'usare il linguaggio Assembly anziché scrivere in linguaggio macchina, ovvero :

- Si possono usare delle etichette simboliche per variabili e indirizzi
- La possibilità di usare primitive per allocazioni in memoria di variabili
- Costanti
- Si possono definire delle macro

⚠ Warning

Attenzione, non ci si deve confondere :

- Assembler : è il nome del software traduttore, NON è Assembly, sono due cose diverse.
- Linguaggio macchina : NON è il linguaggio assembly o le istruzioni ISA, sono cose diverse.
- Linguaggio Assembly : NON è uguale e univoco. È diverso per ogni architettura. Esiste quello Intel x86 (approfondimento disponibile sul libro di testo), esiste quello RISC-V (il quale ad esempio mette a disposizione il concetto di pseudo-istruzione, ovvero un'istruzione che in fase di traduzione, viene trasformata in più istruzioni, le direttive...).

Macro (non viste in laboratorio, non essenziali in pratica, ma si devono sapere per teoria)

Una definizione di macro è un modo per assegnare un nome ad una sequenza di istruzioni. Dopo aver definito una macro il programmatore può scrivere il nome al posto della sequenza di istruzioni. Per definire una macro serve :

- un header della macro che indica il nome della macro da definire
- il testo che comprende il corpo della macro
- una "assembly directive" che indica la fine della definizione

☰ Example

```
# swap macro
.macro swap reg1, reg2, reg3 ← parametri
    add \reg3, \reg2, zero
    add \reg2, \reg1, zero
    add \reg1, \reg3, zero
.endm

li s1, 10
li s2, 20
li s3, 30
swap s1, s2, t0 → tradotto come
swap s2, s3, t0
add t0, s2, zero
add s2, s1, zero
add s1, t0, zero
add t0, s3, zero
add s3, s2, zero
add s2, t0, zero
```

⚠ Warning

Non stiamo parlando di procedure. Le macro non vengono tradotte in run time (ovvero quando eseguiamo il programma), ma è l'assemblatore che si occupa di tradurle.

Item	Macro call	Procedure call
When is the call made?	During assembly	During program execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a procedure call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	One

Linking e loading

- Programmi : insieme di procedure (moduli) tradotti separatamente dall'assemblatore (o compilatore). Ogni modulo oggetto ha il suo spazio di indirizzamento separato.

Linker

È un programma che esegue la funzione di collegamento dei moduli oggetti in modo da formare un unico modulo eseguibile relativo alla macchina specificata. Cosa vuol dire? Su un computer che ha un processore Apple, è possibile creare un eseguibile per RISC-V, tuttavia non è possibile eseguirlo.

Compiti del linker

Il linker fonde gli spazi di indirizzamento dei moduli oggetto in uno spazio lineare unico nel modo seguente :

- Costruisce una tabella di tutti i moduli oggetto e le loro lunghezze
- Assegna un indirizzo di inizio ad ogni modulo oggetto
- Trova tutte le istruzioni che accedono alla memoria e aggiunge a ciascun indirizzo una relocation constant corrispondente all'indirizzo di partenza del suo modulo
- Trova tutte le istruzioni che fanno riferimento ad altri moduli e le aggiorna con l'indirizzo corretto.
Se ho un programma che usa solo funzioni interne allo stesso programma (tutte definite dentro ed unico file), quello che viene fatto è semplicemente una chiamata a funzione dove l'assemblatore va a specificare lo spostamento per poter saltare alla funzione nella fase di JAL (jump and link).
Invece ci sono altri casi, ovvero quando usiamo qualcosa di una libreria esterna al nostro codice, basta pensare a quando usiamo una printf oppure una variabile globale (ad esempio PI).

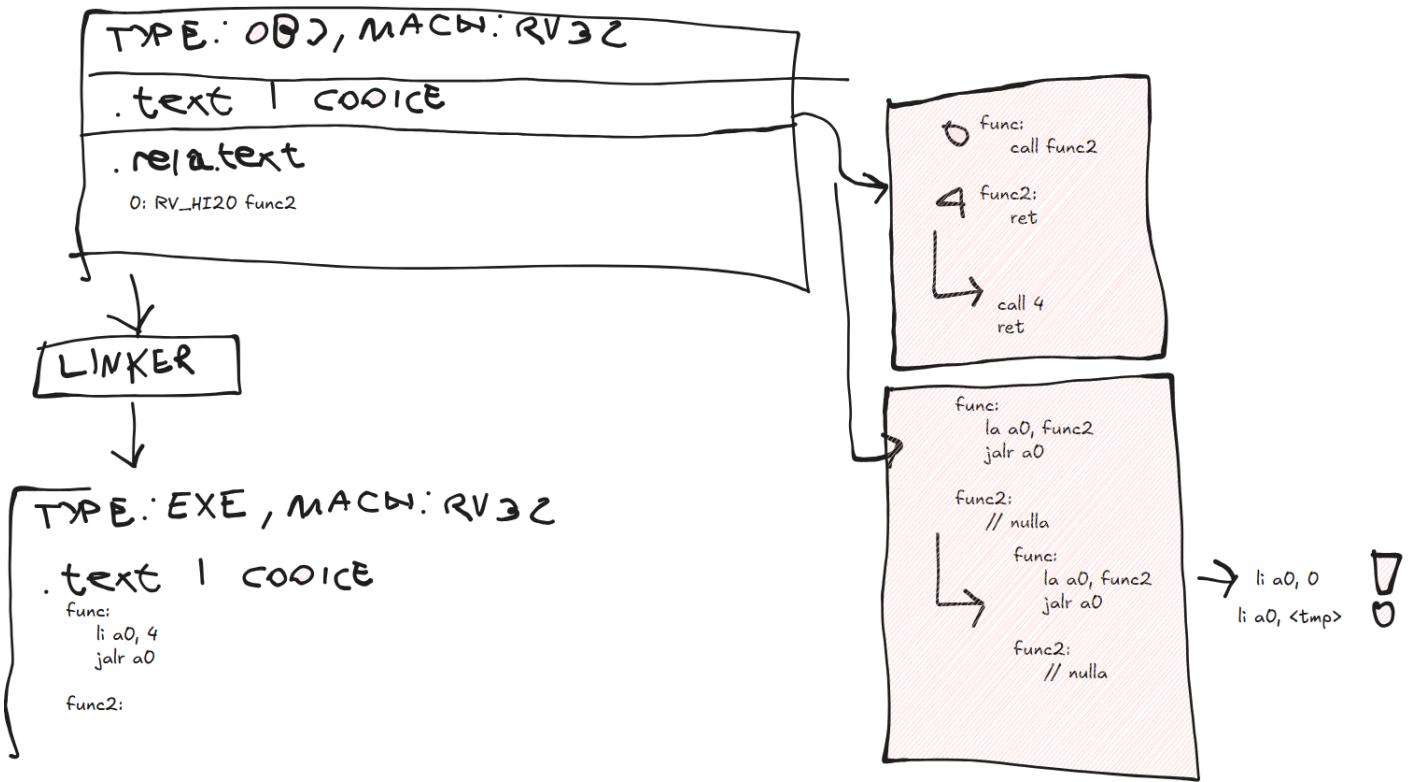
Veloce riassunto

Abbiamo il compilatore. Questo prende in input un file sorgente C e produce un file sorgente assembly. Preso questo file assembly, lo si passa per l'assembler il quale produce un file oggetto. Questo file

oggetto viene poi passato al linker

Composizione file oggetto

- header (es. "type : obj, macchina : rv32")
- sezioni/segmenti (la sezione .text ad esempio dove abbiamo il codice)



Loader

- Una volta creato l'eseguibile (ad opera del linker) esso viene memorizzato su un supporto di memoria secondaria
- Al momento dell'esecuzione il sistema operativo lo carica in memoria centrale e ne avvia l'esecuzione
- Il loader (che è un programma del sistema operativo) si occupa di:
 1. Leggere l'intestazione per determinare la dimensione del programma e dei dati
 2. Riservare uno spazio in memoria sufficiente per contenerli
 3. Copiare programma e dati nello spazio riservato
 4. Copiare nello stack i parametri (se presenti) passati al main
 5. Inizializzare tutti i registri e lo stack pointer
(ma anche gli altri del modello di memoria)
 6. Saltare ad una procedura che copia i parametri dallo stack ai registri e che poi invoca il main

Binding e allocazione dinamica

Se si **spostano** in memoria programmi per cui è già stato fatto il collegamento e il calcolo degli indirizzi, tutti gli **indirizzi** di memoria risultano sbagliati e le informazioni di rilocazioni sono state scartate da tempo. Il problema di spostare in memoria programmi è connesso con la scelta del momento in cui effettuare il collegamento (**binding**) tra nomi simbolici e indirizzi fisici (**binding time**).

Quando fare il collegamento? Ci sono alcune scelte possibili:

- Al momento della **scrittura** del programma
- Al momento della **traduzione** del programma
- Al momento del **linking** (ma prima del loading)
- Al momento del **loading**
- Al momento dell'**esecuzione** (uso di un registro di base)

Collegamento statico:

- Le funzioni di libreria diventano parte del codice eseguibile
- Se viene rilasciata una nuova versione, un programma che carica staticamente le librerie continua a utilizzare la vecchia versione
- La libreria può essere molto più grande del programma; i file binari diventano eccessivamente grossi

Collegamento dinamico:

- DLL, Dynamically Linked Libraries
- Le funzioni di libreria non vengono collegate e caricate finché non si inizia l'esecuzione del programma
- DLL con collegamento lazy: ogni procedura viene caricata solo dopo la sua prima chiamata

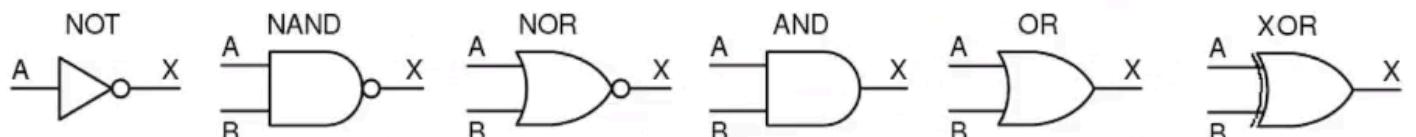
Circuiti Digitali

I circuiti digitali sono gli elementi di base con cui si sono costruiti i calcolatori. Sono dispositivi che utilizzano solo due valori logici 0 e 1 (oppure rispettivamente un segnale tra 0 e 1 volt e uno tra 2 e 5. Questi valori non sono universali, possono essere anche diversi).

Un circuito digitale trasforma segnali (binari) di ingresso $x_1, x_2, x_3, \dots, x_n$ nei segnali (binari) di uscita $z_1, z_2, z_3, \dots, z_n$.

Porte logiche

Sono circuiti costituiti da transistor, etc. Questo è il livello più "basso" che vedremo.



A	X
0	1
1	0

A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

A	B	X
0	0	1
0	1	0
1	0	0
1	1	0

A	B	X
0	0	0
0	1	1
1	0	1
1	1	1

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

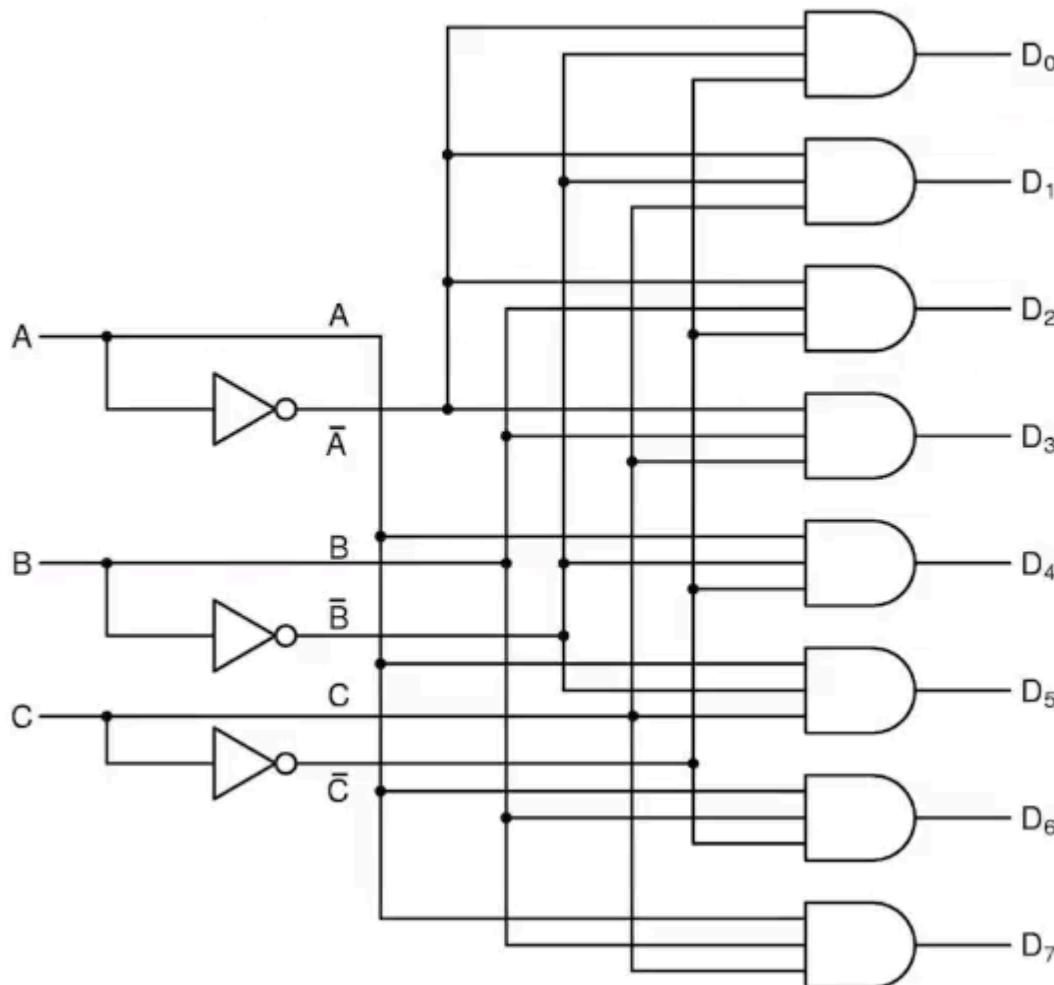
A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

Circuiti combinatori

Nei circuiti combinatori l'output viene determinato solo dagli input. Per convenzione, l'incrocio tra due linee non implica alcuna connessione a meno che non sia presente il simbolo • nel punto di intersezione.

☰ Example

Esempio di circuito combinatorio :



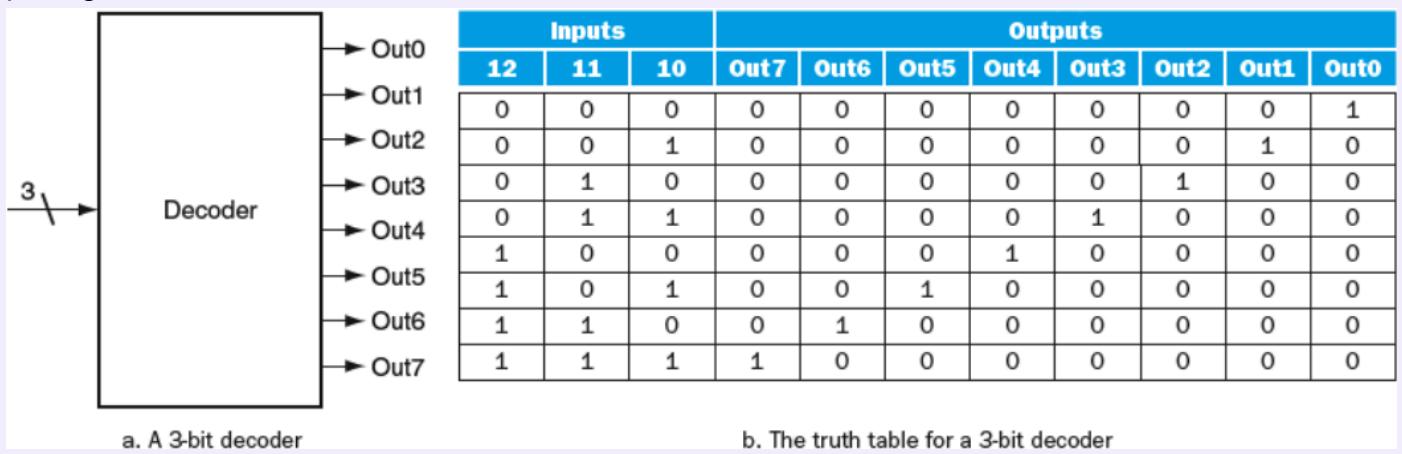
Decoder

Il decoder è un circuito combinatorio che permette di scegliere uno di 2^n output, avendo n input. I decoder sono utili per selezionare qualcosa. Vengono usati ad esempio per scrivere in un registro anziché in un altro.

☰ Example

Nell'immagine sotto, abbiamo 3 bit di input che permettono di selezionare uno di 8 bit di output. Interpretiamo gli ingressi A B C (o I₂ I₁ I₀) come le cifre di un numero in base 2 con A (I₂) quella

più significativa.

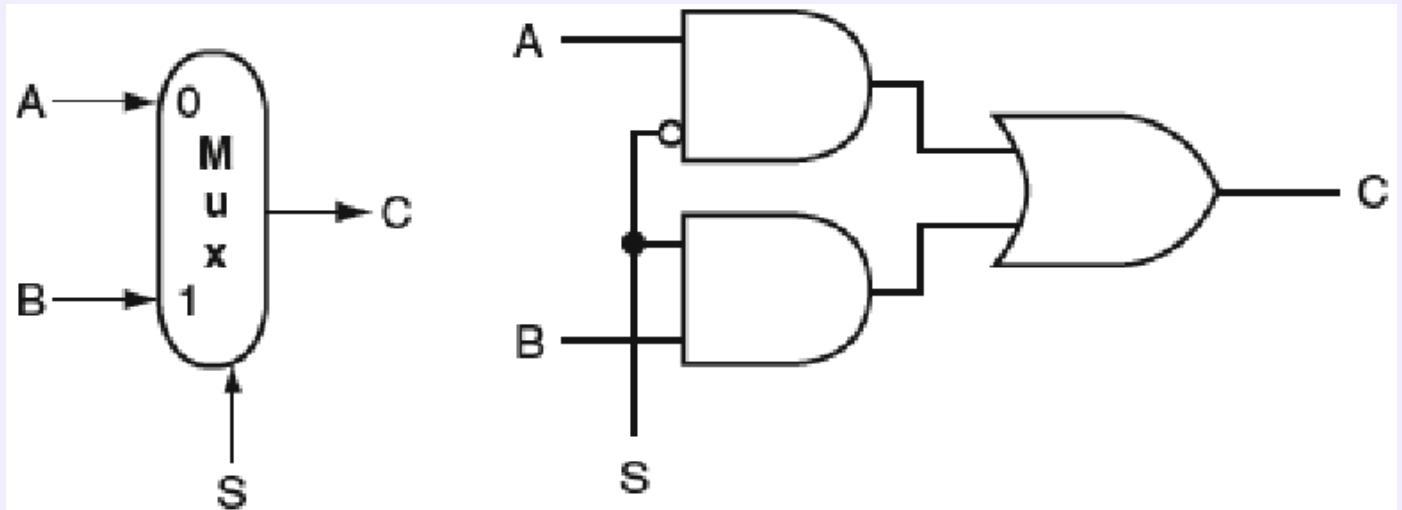


Multiplexer

Il multiplexer è un circuito combinatorio che dati 2^n input, permette di selezionare in uscita 1 solo di questi input avendo n ingressi di controllo. La linea di controllo viene chiamata anche "selettore". I multiplexer sono utili per selezionare qualcosa. Vengono usati ad esempio per prendere qualcosa da uno specifico registro anziché da un altro (magari una *lw* o una *add* di due registri). Per convenzione, quando il valore di ingresso è 0, viene scelto A.

Example

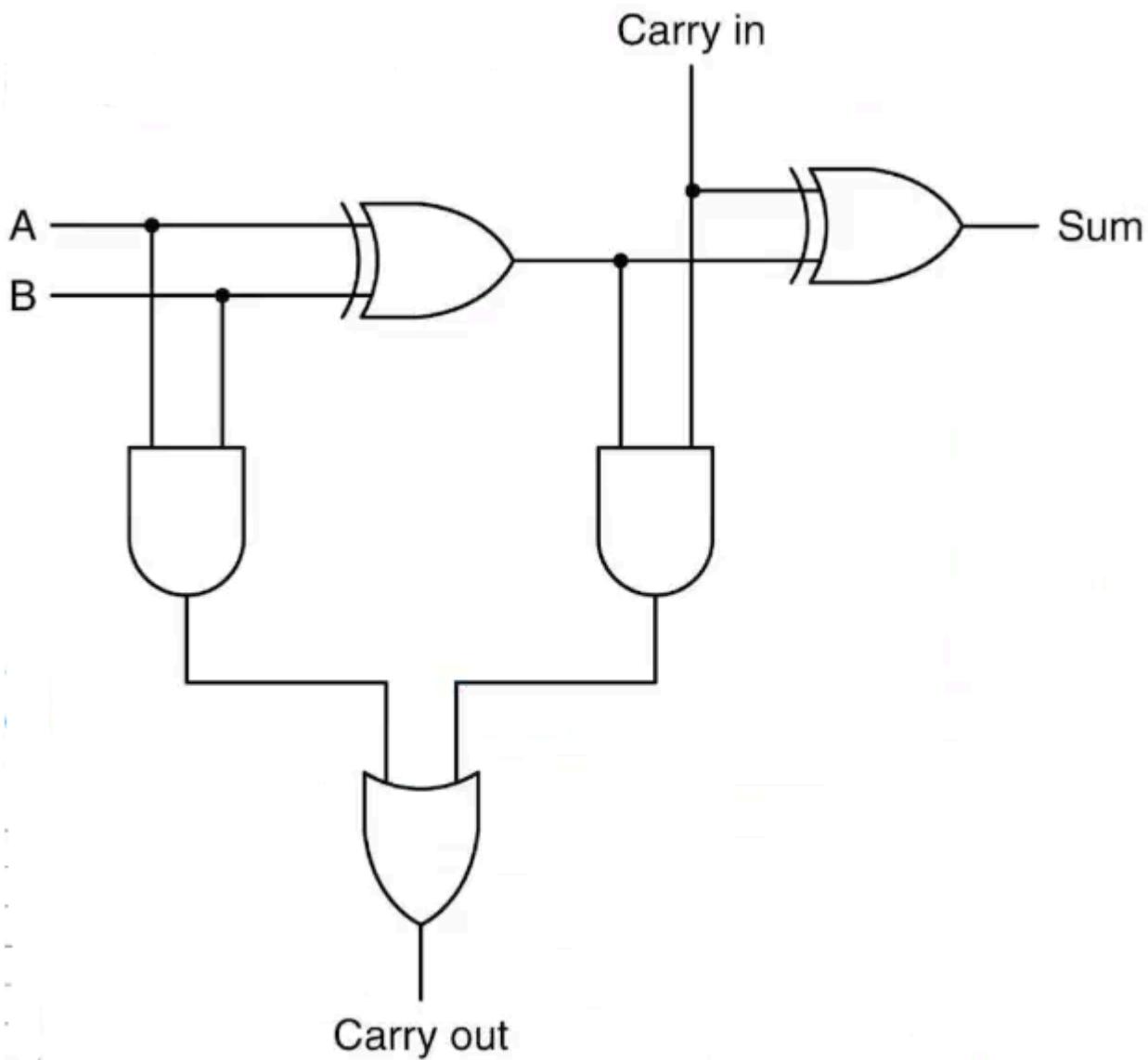
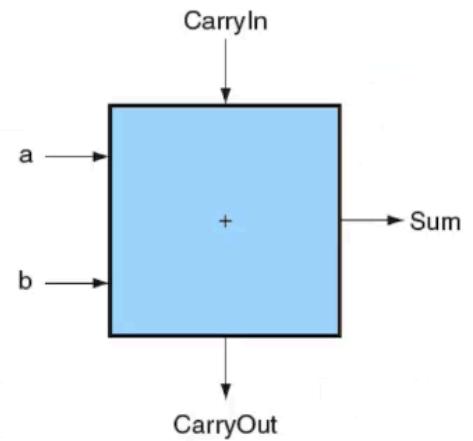
Nell'esempio sotto, abbiamo $n = 1$, $2^n = 2$. A destra anche il multiplexer sulla sinistra costruita con le porte logiche.



Addizionatore

- Riceve in ingresso due bit (cifre in base 2) da sommare.
 - Riceve in ingresso un bit (cifra in base 2) di riporto, il CarryIn.
 - Restituisce un bit in uscita che rappresenta il risultato, Sum.
 - Restituisce un bit in uscita che rappresenta il riporto CarryOut nello schema.
- Dato che c'è anche il CarryIn (il riporto), questo è un full adder

Inputs			Outputs	
a	b	CarryIn	CarryOut	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

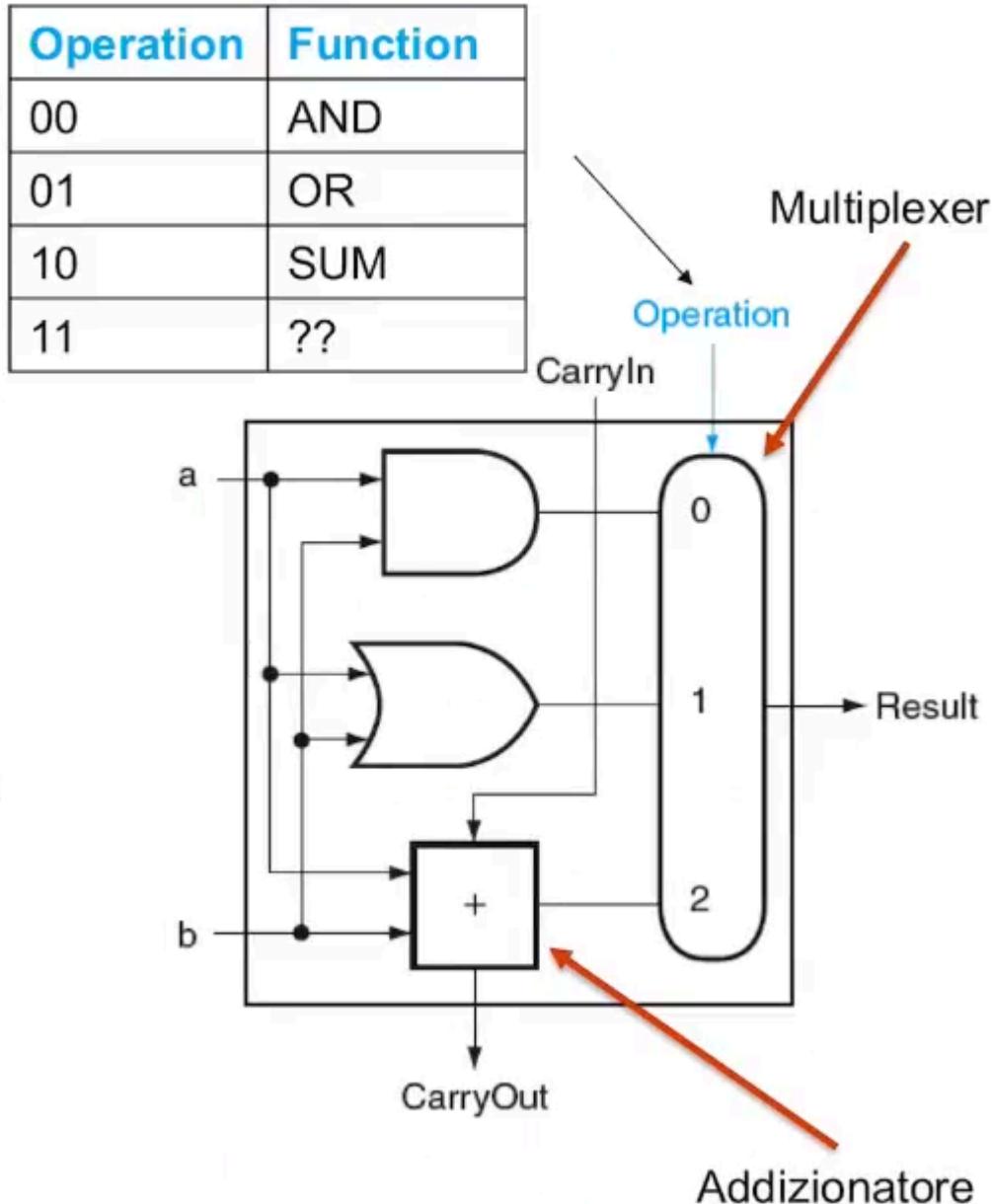


ALU (1 bit)

ALU (o Arithmetic Logic Unit) è un unità fondamentale di un processore. Permette di eseguire un certo calcolo tra input dati. Le operazioni che possono essere eseguite sono and, or e somma. La ALU è formata da :

- Gli ingressi degli input A e B.
- L'ingresso di CarryIn preso da un'altra ALU o impostato a 0 se è la prima ALU.
- L'ingresso di Operation, il quale permette di selezionare quale operazione si vuole svolgere.

- L'uscita di CarryOut dato dal full adder.
- L'uscita Result che restituisce il risultato dell'operazione selezionata.



Come si vede da sopra, le operazioni vengono eseguite tutte quante e poi viene scelto il giusto output tramite il multiplexer.

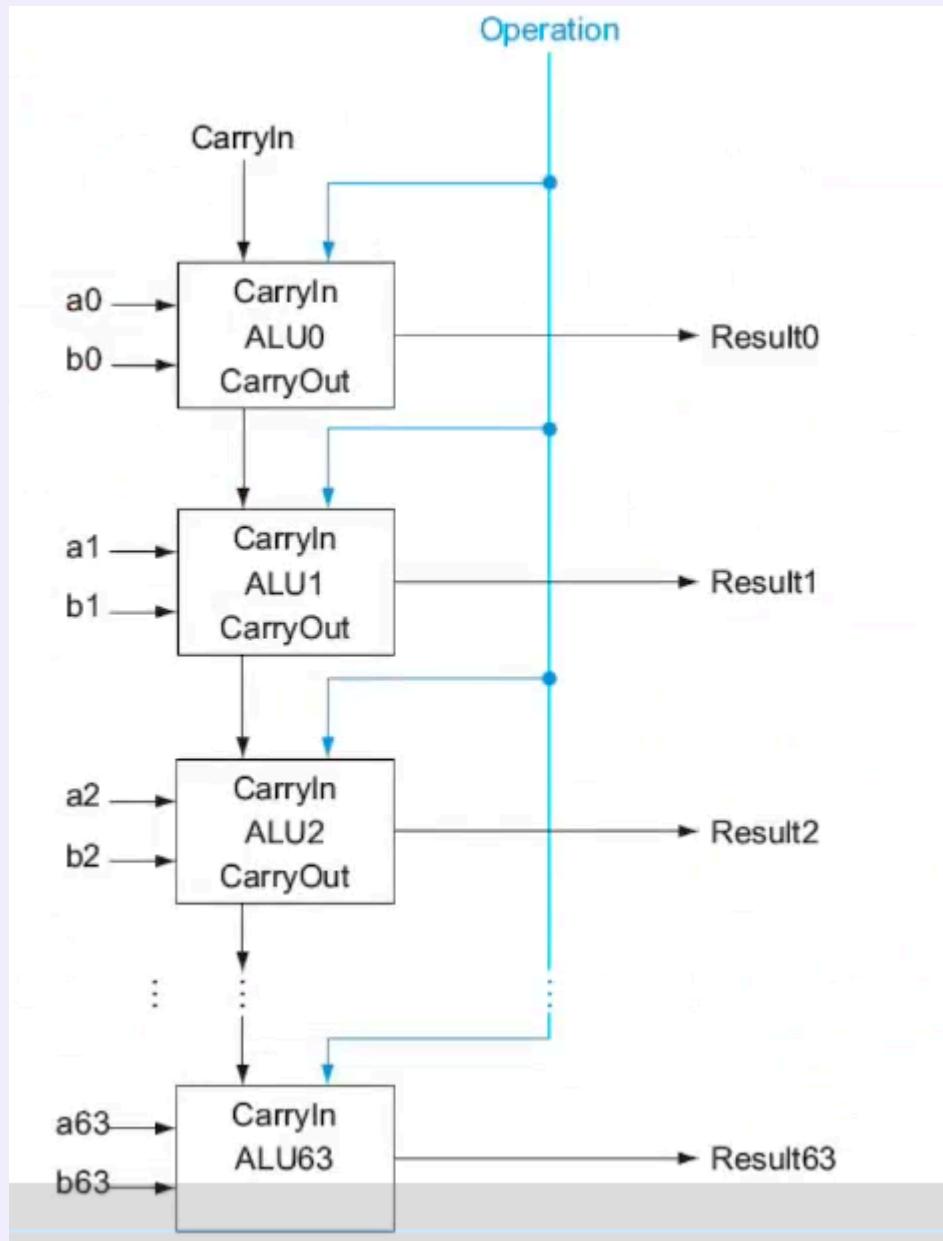
ALU (a 2^n bit)

Per svolgere un calcolo tra due registri da 64 bit non basta un ALU da 1 bit, ma avremo bisogno di 64 ALU da 1 bit collegate in serie. Il risultato di ogni ALU viene preso e messo nella giusta posizione di un altro registro a 64 bit.

Example

Sotto si può vedere un esempio. Si pensi ad A e B come due registri di cui si vuole fare la AND. Avremo a_n con $a \in A$ e con n l'ennesimo bit di A , stessa cosa per b_n . L'operation invece è la stessa

per tutti ovviamente.



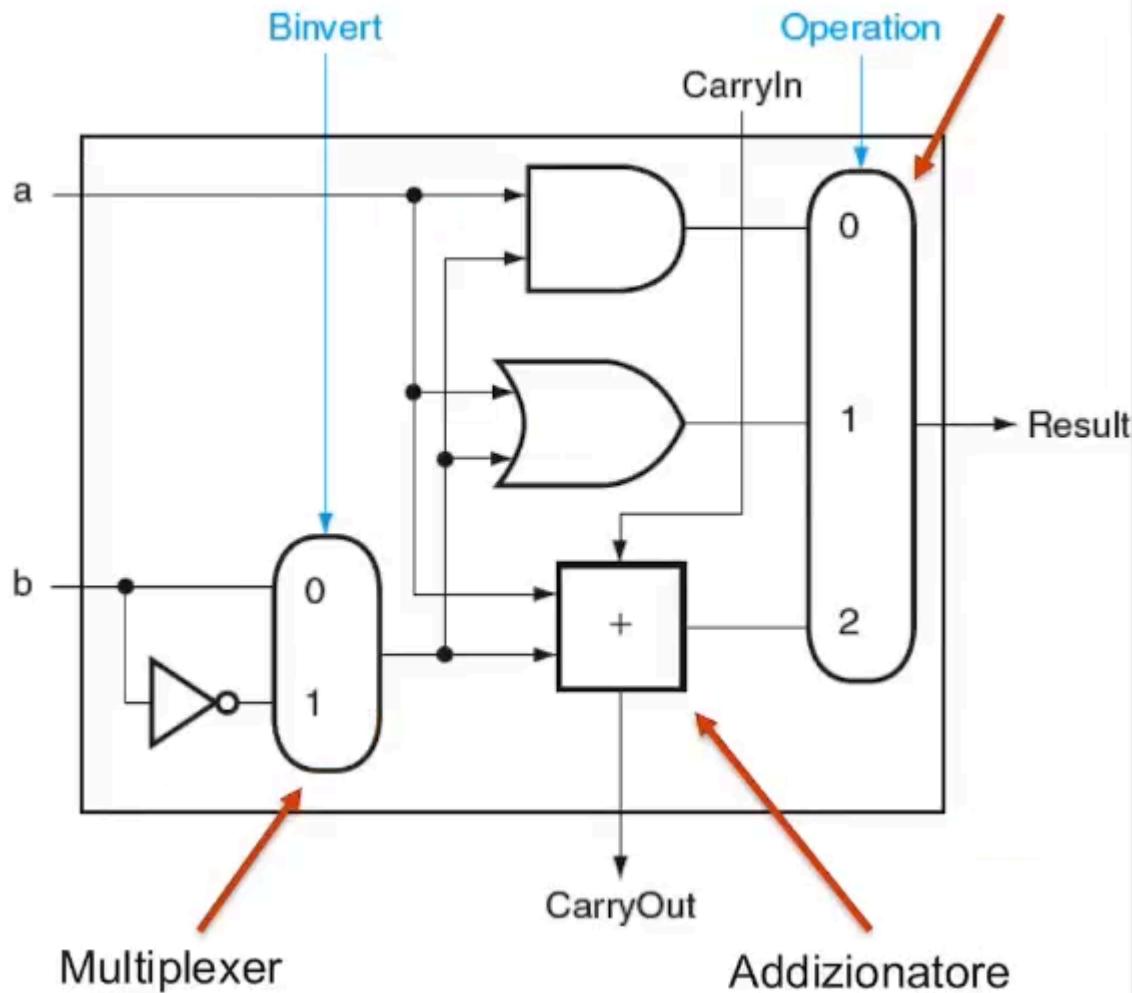
ALU con sottrazione

Binvert

Come l'abbiamo vista fin ora, la ALU non può fare la sottrazione, ma in realtà è una cosa che si può rimediare molto facilmente. La sottrazione non è altro che $A + (-B)$, ovvero $A +$ l'opposto di B . L'opposto si ottiene facilmente facendo il complemento a 2 di B , ovvero facendo il NOT di ogni bit di B e aggiungendo un 1 alla fine (aggiungere l'uno alla fine è molto facile, basta mettere il CarryIn del bit meno significativo ad 1). Allora, mettendo un nuovo input detto "Binvert" (B-Invert) e un multiplexer,

possiamo scegliere di sottrarre due numeri.

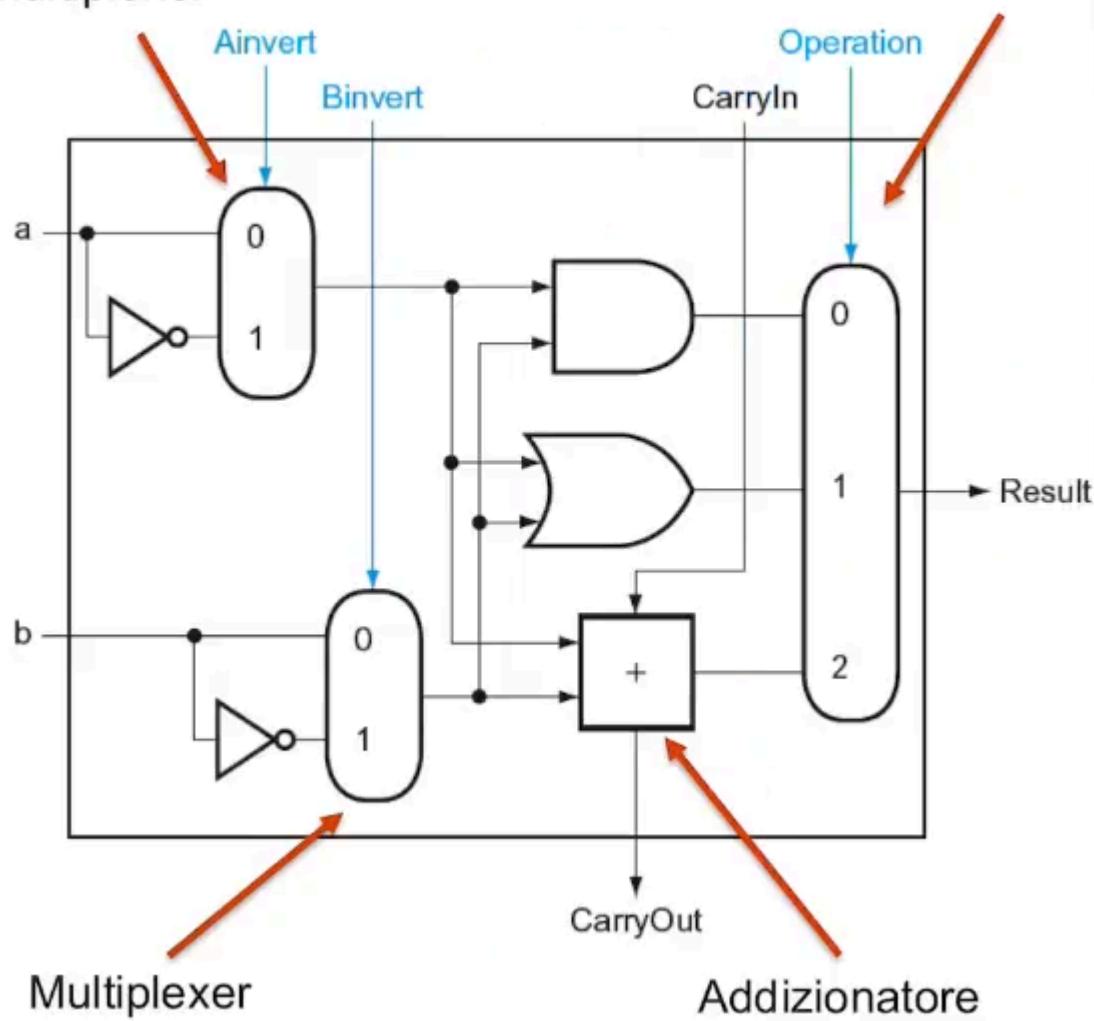
Multiplexer



Ainvert

Facendo lo stesso ragionamento, potremmo volere invertire anche A, quindi, facciamo la stessa cosa fatta per B e otteniamo la possibilità di fare $(\bar{a} \text{ OR } b)$ con l'uso di $(\bar{a} \text{ AND } \bar{b})$.

Multiplexer

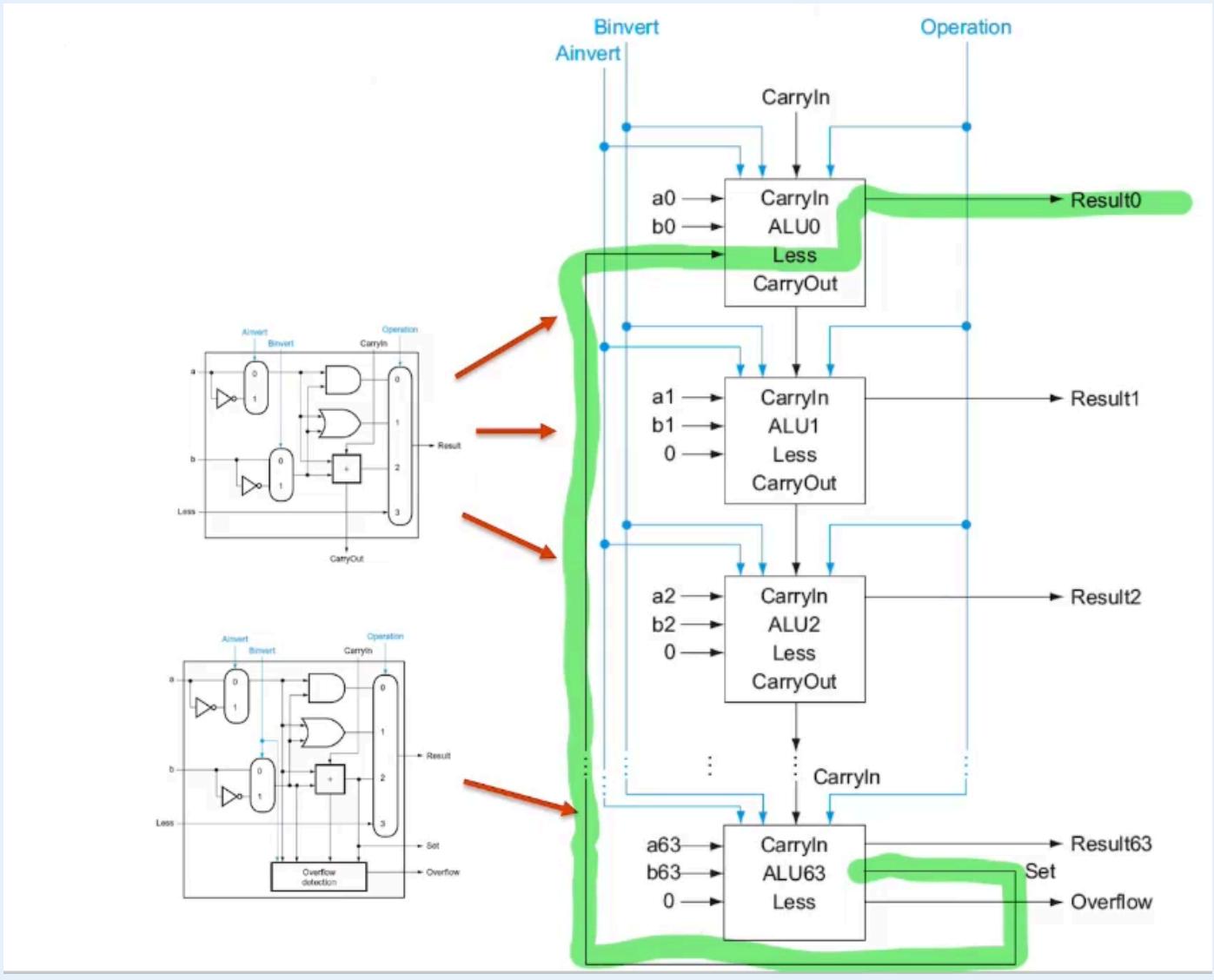


SLT

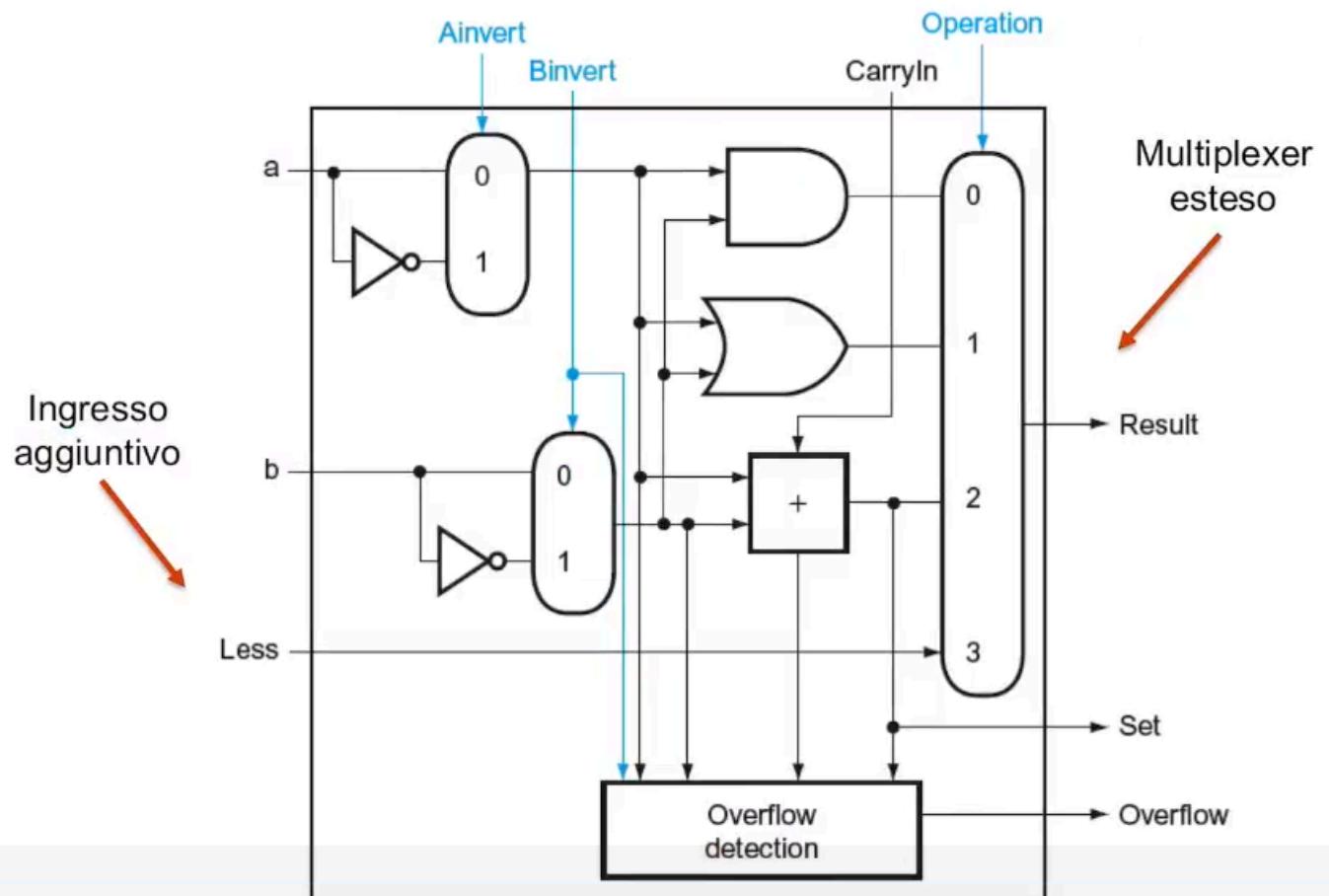
L'istruzione `slt` (set less than) restituisce 1 se $rs1 < rs2$ e 0 altrimenti. In output avremo che il valore di tutti i bit (tranne quello meno significativo) valgono 0, mentre quello meno significativo prende il valore dal confronto. Per poter implementare ciò, basta vedere se $a - b < 0$, allora ciò implica che $a < b$. Come possiamo farlo? Basta eseguire la sottrazione e sapendo che i risultati sono salvati con i segni, basta controllare il bit più significativo. Se il valore di questo è 1, allora $a < b$. In modo fisico, per fare ciò, basta prendere il risultato dell'ultima ALU che fa il calcolo. Adesso, facciamo uscire il risultato della sottrazione di questo ultimo bit e lo riportiamo alla prima ALU come campo "LESS" (nelle altre ALU, mettiamo 0).

Note

Struttura finale della ALU



Note



Overflow

Per capire se c'è stato un overflow, ci sono due modi :

- Se la somma di due numeri positivi fanno un numero negativo. Se la somma di due numeri negativi fanno un numero positivo.
- Se l'ultimo riporto ottenuto è discorda al penultimo riporto ottenuto.

Bnegate

Come visto nella scorsa lezione, ogni volta che vogliamo che l'ALU esegua sottrazioni, dobbiamo asserire (porre a 1) sia CarryIn sia Binvert

Operazione	Funzione	Binvert	CarryIn0
0	AND	0	0
1	OR	0	0
2	ADD	0	0
2	SUB	1	1
3	SLT	1	1

Notando che Binvert è ad 1, quando CarryIn0 (ovvero il CarryIn della prima ALU a 1 bit) è ad 1,

possiamo creare una sola linea che si chiama Bnigate, per poter risparmiare una linea (la quale porta ad una minore grandezza del bus, meno pin in un chip, meno costi...).

Beq, funzionamento

Beq realizza un salto se due registri sono uguali. Avviene quindi una sottrazione tra i due registri e se il risultato è uguale a 0, avviene il salto, altrimenti no.

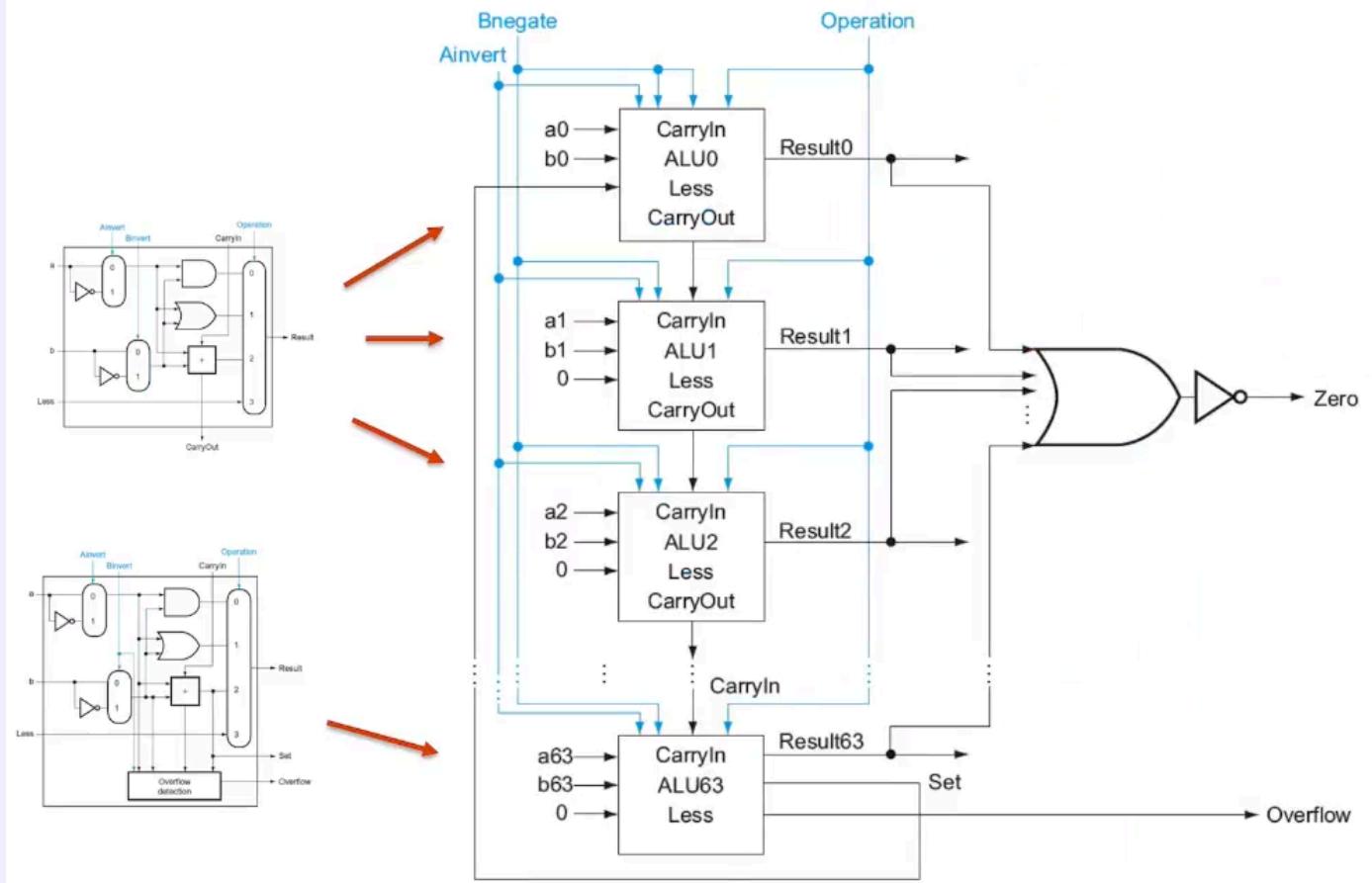
$$(a - b) == 0$$

L'operazione sopra viene eseguita nel seguente ordine :

1. Complemento a 1 di b
2. Aggiungo 1 a b (adesso ho $-b$)
3. Eseguo la somma tra 'a' e ' $-b$ '
4. Adesso ho il risultato. Devo controllarlo. Se il risultato vale 0, dobbiamo saltare, altrimenti no. Per far saltare però, dobbiamo dare come output '1'.
5. Neghiamo il risultato
6. Facciamo l'or. Se c'è un 1, vuol dire che $a == b$

Example

Come possiamo vedere sotto (una possibile soluzione), prendiamo l'or del risultato di tutti i bit. Se tutti valgono 0, allora passa 0 dalla OR e diventa 1 con la NOT e il risultato è zero = 1 (zero = 1 se i due numeri sono uguali!), altrimenti ogni altro valore di un result diverso da zero farà diventare zero = 0.



Controllo operazioni ALU

Per riassumere, nella nostra ALU avremo :

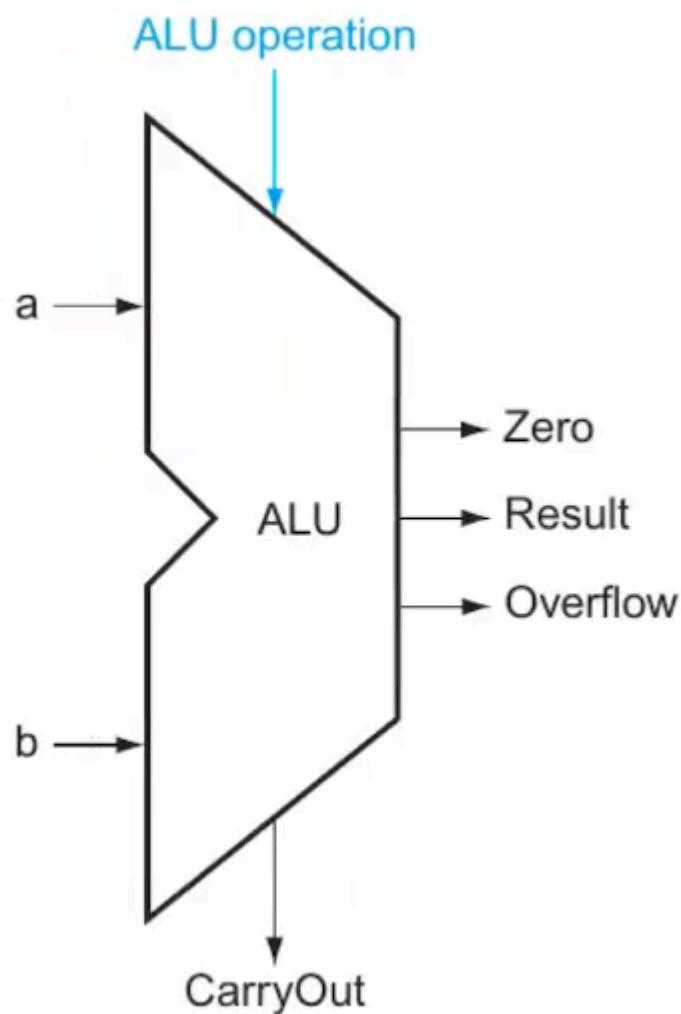
- 2 bit per il controllo dell'operazione (operation).
- 1 bit per l'ingresso di Ainvert (che usiamo per fare la nor).
- 1 bit per l'ingresso di Binvert (che usiamo per fare una sottrazione,slt e nor).

Ainvert	Bnigate	Operation	ALU control lines	Function
0	0	00	0000	AND
0	0	01	0001	OR
0	0	10	0010	add
0	1	10	0110	subtract
0	1	11	0111	set less than
1	1	00	1100	NOR

E ancora avremo :

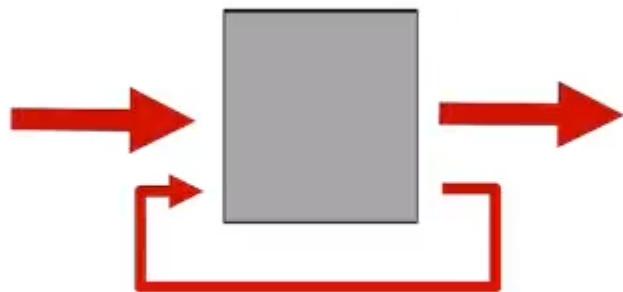
- 32 bit di ingresso per A (architettura a 32 bit)
- 32 bit di ingresso per B
- 32 bit di uscita per result
- 1 bit per zero
- 1 bit per overflow

- 1 bit per il CarryOut



Circuiti sequenziali

I circuiti sequenziali sono dispositivi in grado di calcolare funzioni che dipendono anche da uno stato interno, quindi dipendono anche da informazioni memorizzate in elementi di memoria interni. In generale, la funzione calcolata dal circuito ad un dato istante dipende dalla sequenza temporale dei valori in input al circuito.



Latch

Latch, un dispositivo a 1 bit che ha due ingressi (i e β) ed un'uscita (Q). Mantiene uno stato interno (s). La logica del latch è la seguente, se $\beta = 0$, lo stato futuro è uguale allo stato passato (non viene sovrascritto il bit salvato). Invece, se $\beta = 1$, lo stato futuro varia in base all'input i . Quindi :

- $b = 0$: hold, si mantiene il contenuto della memoria
- $b = 1$: store, si riscrivere il contenuto della memoria

β	i	s	$s' = o$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

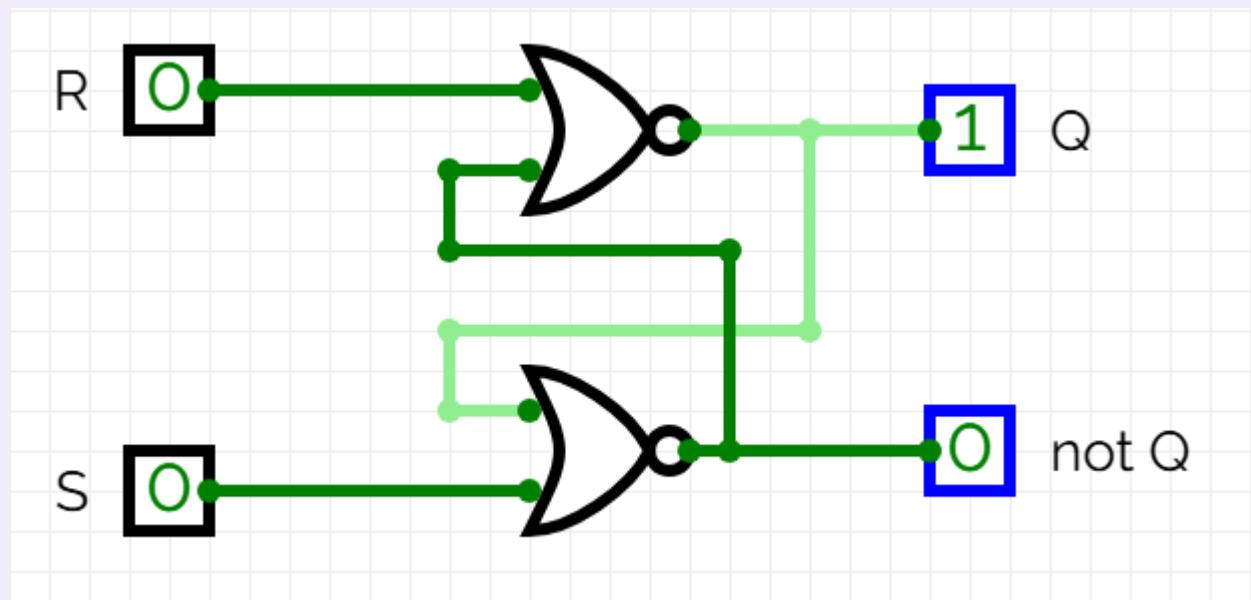
Latch SR

Latch SR (set-reset). Logicamente simile al latch visto sopra. Le fasi sono :

- Hold, quando $R = S = 0$, allora si mantiene lo stato salvato all'interno
- Set (store 1), $S = 1$ e $R = 0$, porta il latch allo stato 1
- Reset (store 0), $R = 1$ e $S = 0$, porta il latch allo stato 0

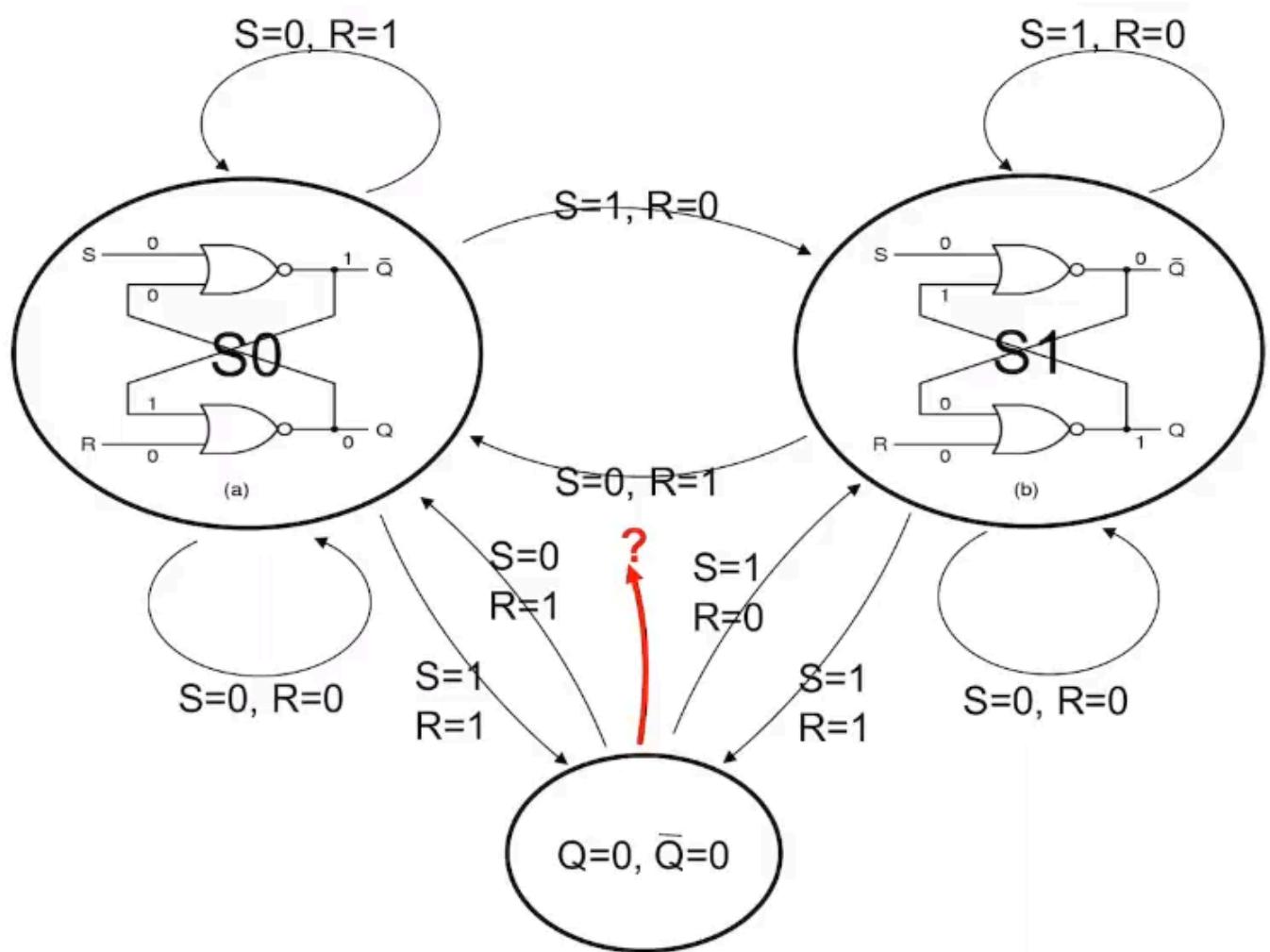
Example

Sotto un latch SR con stato salvato 1 e in fase di HOLD



Avere $S = R = 1$ è una cosa che non dovrebbe mai accadere. Lo stato non è stabile, per tanto Q diventa impredicibile e potrebbe nascere un'oscillazione.

Automa a stati finiti di Latch SR



La freccia rossa indica il momento in cui passiamo da $R = S = 1$ a $R = S = 0$, cosa che rende appunto impredicibile lo stato e quindi che non deve accadere.

Visto che non dobbiamo mai arrivare ad avere $S = R = 1$, dobbiamo cercare una soluzione in cui dobbiamo salvare il risultato di S e R sì, ma solo quando il loro segnale è stabile (S e R sono calcolati da circuiti combinatori, è possibile che il loro segnale non sia immediatamente stabile per varie ragioni), questa soluzione è il **clock**.

Latch SR sincronizzato

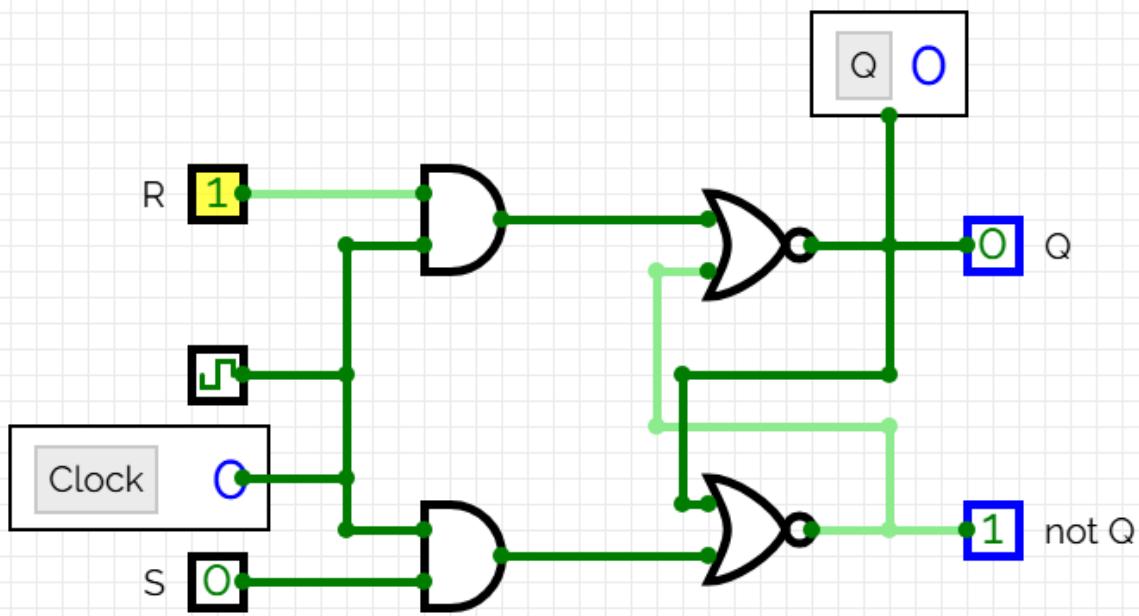
Un clock garantisce che il latch cambi stato solo in certi momenti specifici (momenti scelti cambiando il circuito. Si può decidere di salvare in fronte di salita, in discesa, in stato alto o stato basso).

Aggiungendo quindi un clock e delle porte AND, possiamo fare in modo che i dati vengono salvati solo durante lo stato alto :

Example

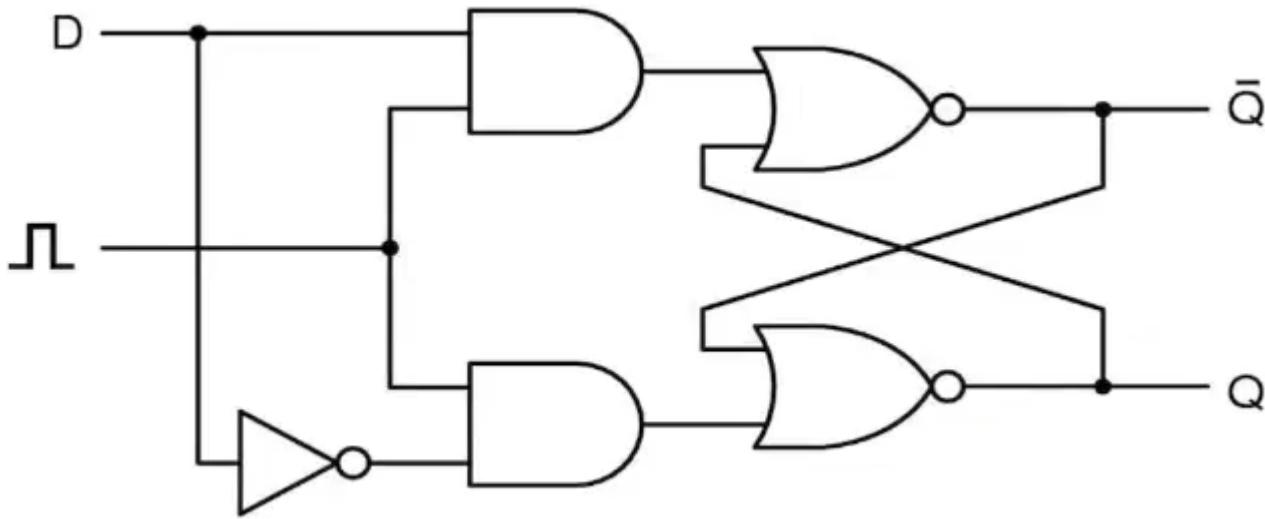
Di seguito un esempio realizzato con CircuitVerse. Possiamo notare sopra che il cambiamento in Q viene eseguito solo quando il Clock è HIGH (evidenziato in rosso).

Time	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361
Q																								
Clock																								



Latch di tipo D sincronizzato

Il latch di tipo D sincronizzato è un miglioramento del latch SR. Il latch di tipo D rimuove il problema di poter avere sia R che S ad 1 contemporaneamente, come? Semplicemente avendo un input "D" che va in S e che va in maniera negata in R. Essendo sincronizzato, presenta anche lui il clock.

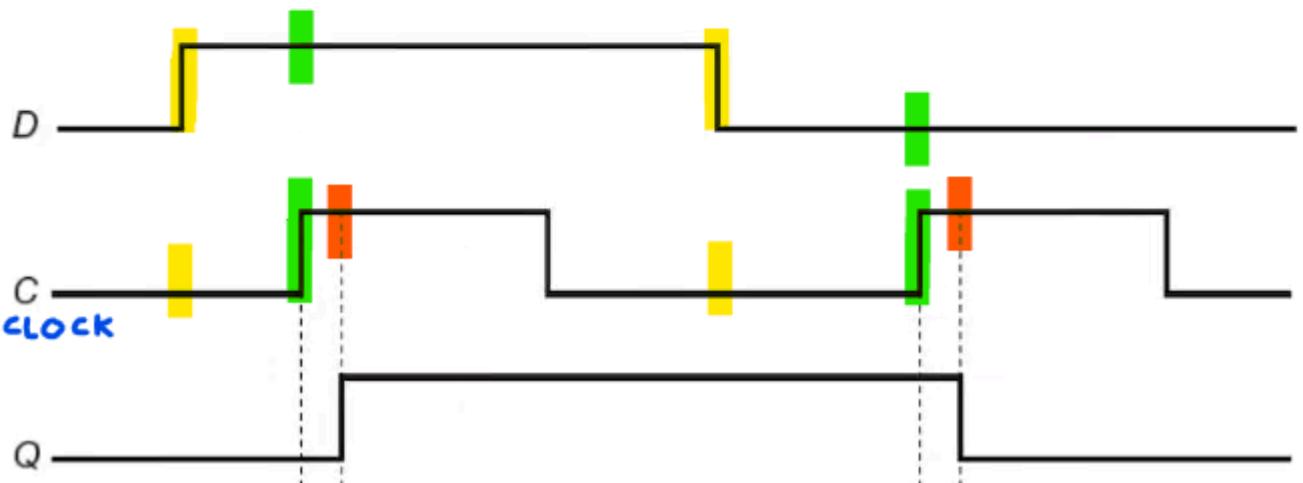
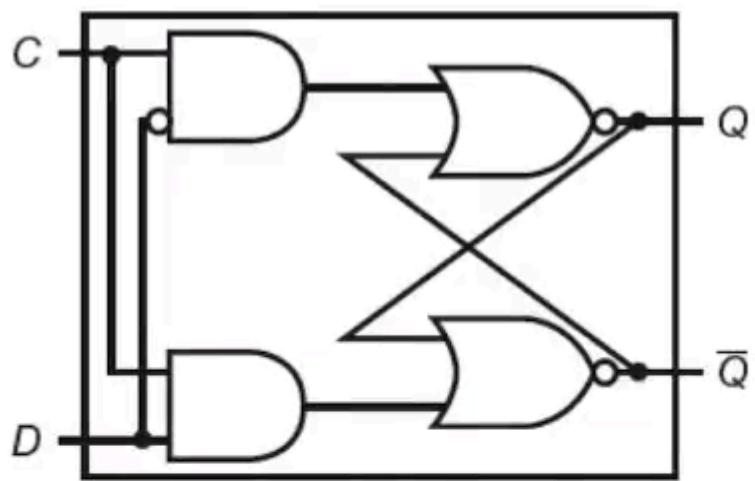


Con

Nel schema sopra, se D varia mentre il clock è 1 (HIGH), allora varia anche lo stato. Se D varia mentre il clock è 0, il primo cambiamento verrà salvato quando il clock torna ad 1.

Example

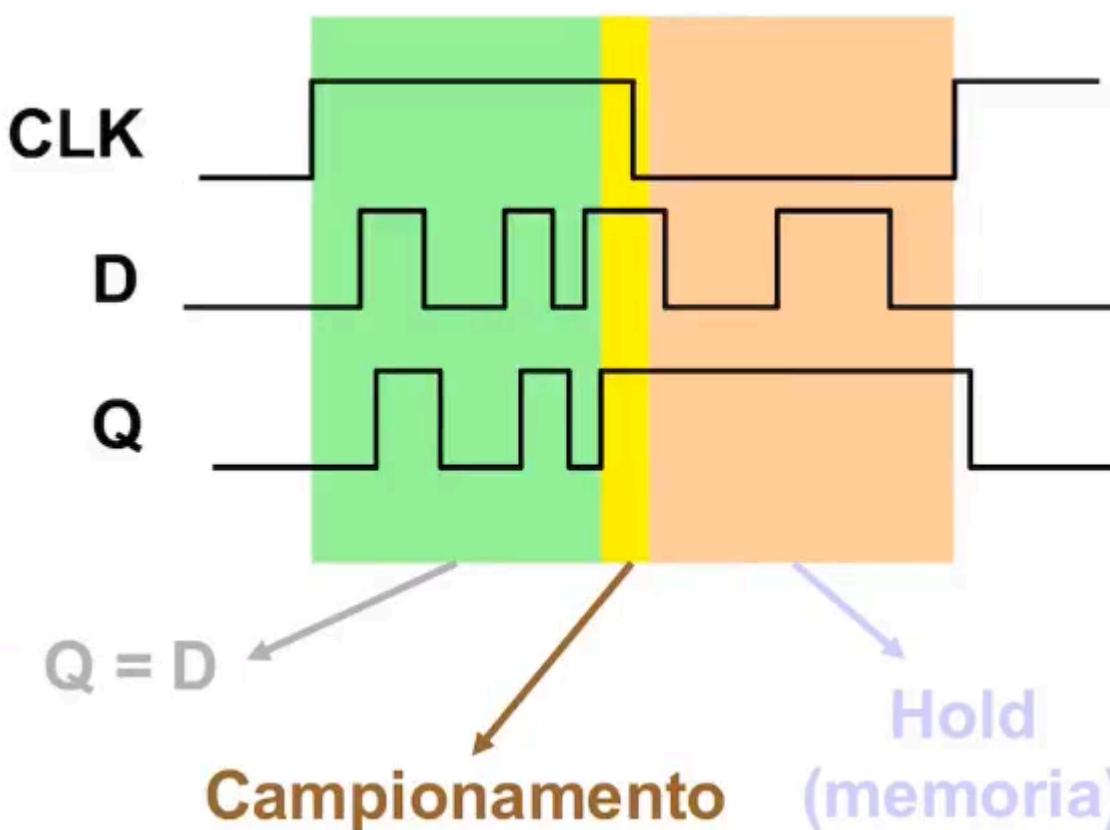
Qui un esempio. Troviamo in giallo il cambiamento di D, in verde lo stato 1 del clock nel quale verrà salvato il cambiamento di D in Q e in rosso il momento effettivo in cui viene cambiato Q.



Come si può notare, c'è una discrepanza tra il momento in cui il clock sale e il momento di salvataggio. Questa discrepanza è dovuta dal ritardo delle porte. Quindi nella parte in cui il clock è basso, il circuito calcola cosa memorizzare e quando il clock è alto, memorizza quello che ha precedentemente calcolato.

Problema della "trasparenza" (importante)

Con il clock = 1, il latch è "trasparente", ovvero, l'uscita segue l'ingresso istante per instante (con un leggero ritardo dovuto, come detto prima, al tempo di propagazione attraverso il latch).



Qual è il problema quindi? Se l'ingresso cambia mentre il latch è abilitato, anche l'uscita cambia immediatamente. Questo può causare **comportamenti imprevisti o instabilità**, soprattutto se l'uscita del latch influenza altri circuiti sensibili ai cambiamenti.

☰ Example

Pensiamo alla seguente istruzione :

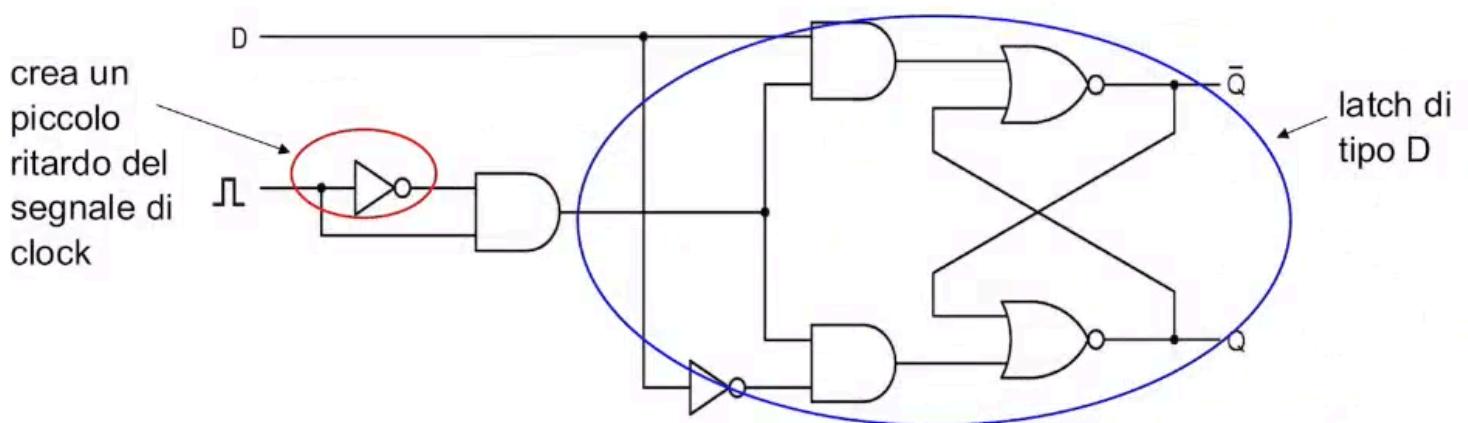
```
add x1, x1, x4
```

Qui prendiamo il valore di x1, lo sommiamo ad x4. Il risultato poi viene salvato in x1, il quale però può ancora tornare indietro creando un problema.

Come si può risolvere? Per evitare questo problema si preferisce usare **flip-flop edge-triggered**, che aggiornano l'uscita **solo sul fronte di salita o discesa del clock**, garantendo un comportamento più stabile e prevedibile nei sistemi digitali sincroni.

Flip-Flop di tipo D

Si aggiunge una porta NOT al segnale di clock e si sdoppia l'uscita del clock. Questi due segnali, vengono messi in AND. La porta NOT metterà ad 1 il segnale quando il clock vale 0. Questo per pochissimo tempo anche quando il valore del clock vale 1, perché c'è quel ritardo di cui parlavamo prima. Questo fa sì che la porta AND faccia passare 1 per un breve lasso di tempo.



In questo breve lasso di tempo, vogliamo salvare il risultato.

Nella condizione ottimale, il tempo in cui avviene il salvataggio dovrebbe essere solo 1, ma fisicamente è impossibile arrivare a questa ottimalità, quindi, si cerca di tenere questo tempo il più basso possibile.

💡 Important

Da ricordare bene. Il Flip-Flop D visto, salva SOLO sul fronte di salita.

Buffer (non) invertente

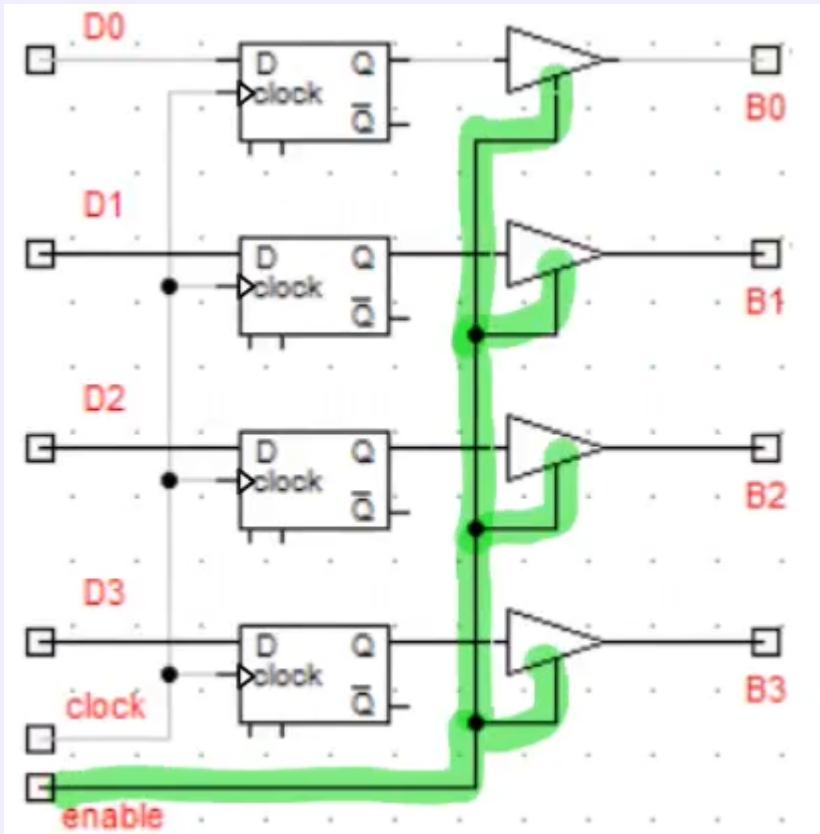
Circuito semplice che ha input, un output, un input di controllo e tre stati, ovvero :

- Quando il segnale di controllo vale 1, il circuito si chiude e l'input scorre fino all'output (valendo quindi o 0 o 1).
- Quando il segnale di controllo vale 0, il circuito si apre e l'output (a prescindere dal valore dell'input), non ha un segnale (terzo stato).

Può essere utile nel caso in cui abbiamo vari un registro (formato da vari flip-flop D) e vogliamo disconnetterli (o connetterli)

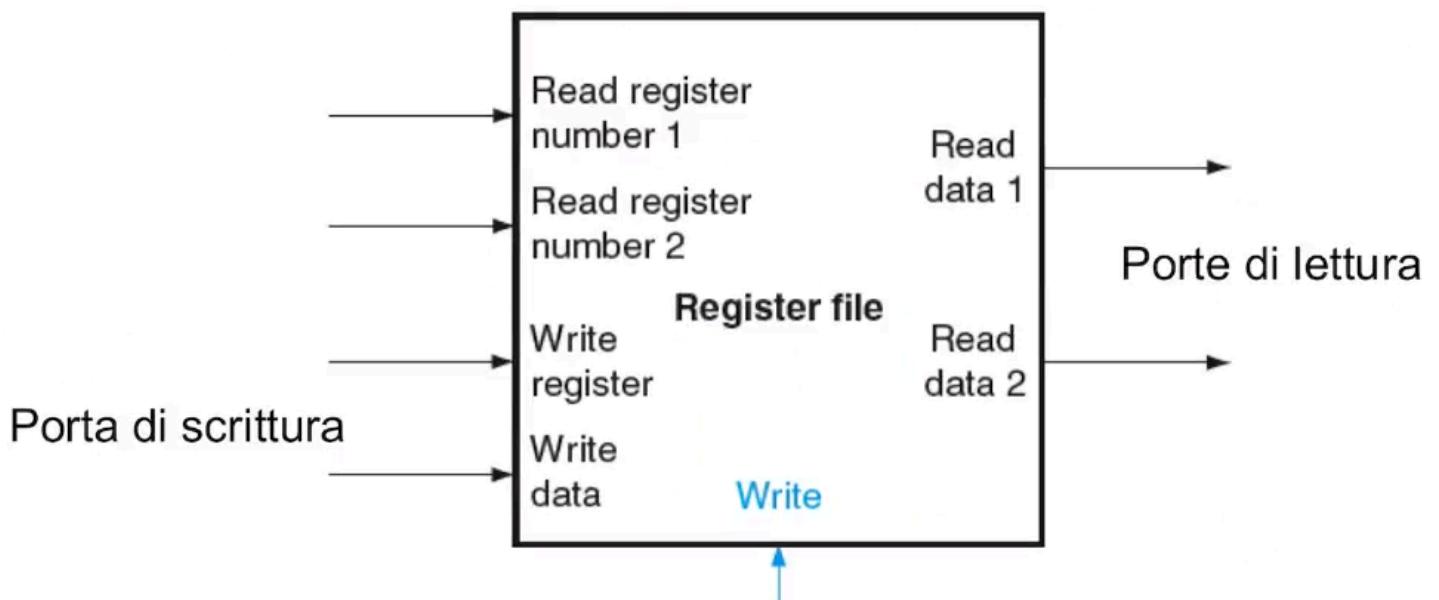
☰ Example

In verde l'enable comune che permette di far passare o meno un dato



Register file (blocco di registry)

Nell'architettura (RISC-V) che stiamo vedendo, il register file contiene 32 registri e la logica di controllo.



Guardando l'immagine sopra, abbiamo :

- Read register number 1 : che si occupa di leggere il primo registro selezionato. Legge 5 bit (sempre per il discorso che abbiamo 32 registri e quindi bastano 5 bit per specificare un numero da 0 a 31).
- Read register number 2 : che si occupa di leggere il secondo registro selezionato. Uguale al 1 in pratica.
- Read data 1 : contiene il valore del registro 1 letto. In un'architettura a 32 bit (quella che stiamo vedendo), restituisce 32 bit.
- Read data 2 : uguale a read data 1, ma contiene il valore del registro 2 letto.

- Write register : serve per specificare l'indirizzo dove voglio andare a scrivere l'eventuale output della ALU. Legge sempre 5 bit.
- Write data : serve per scrivere qual è il valore che voglio scrivere nel registro selezionato. Riceve 32 bit.
- Write (segnale di controllo) : serve per dire che vogliamo scrivere un dato in un registro. È solo 1 bit. Ci serve perché non tutte le istruzioni modificano un valore nei registri (ad esempio una sw, non salva nulla in alcun registro, però abbiamo comunque bisogno di accedere ai registri).

Exemple

Quindi, per chiarire, prendiamo questo esempio :

```
add x2, x4, x5
```

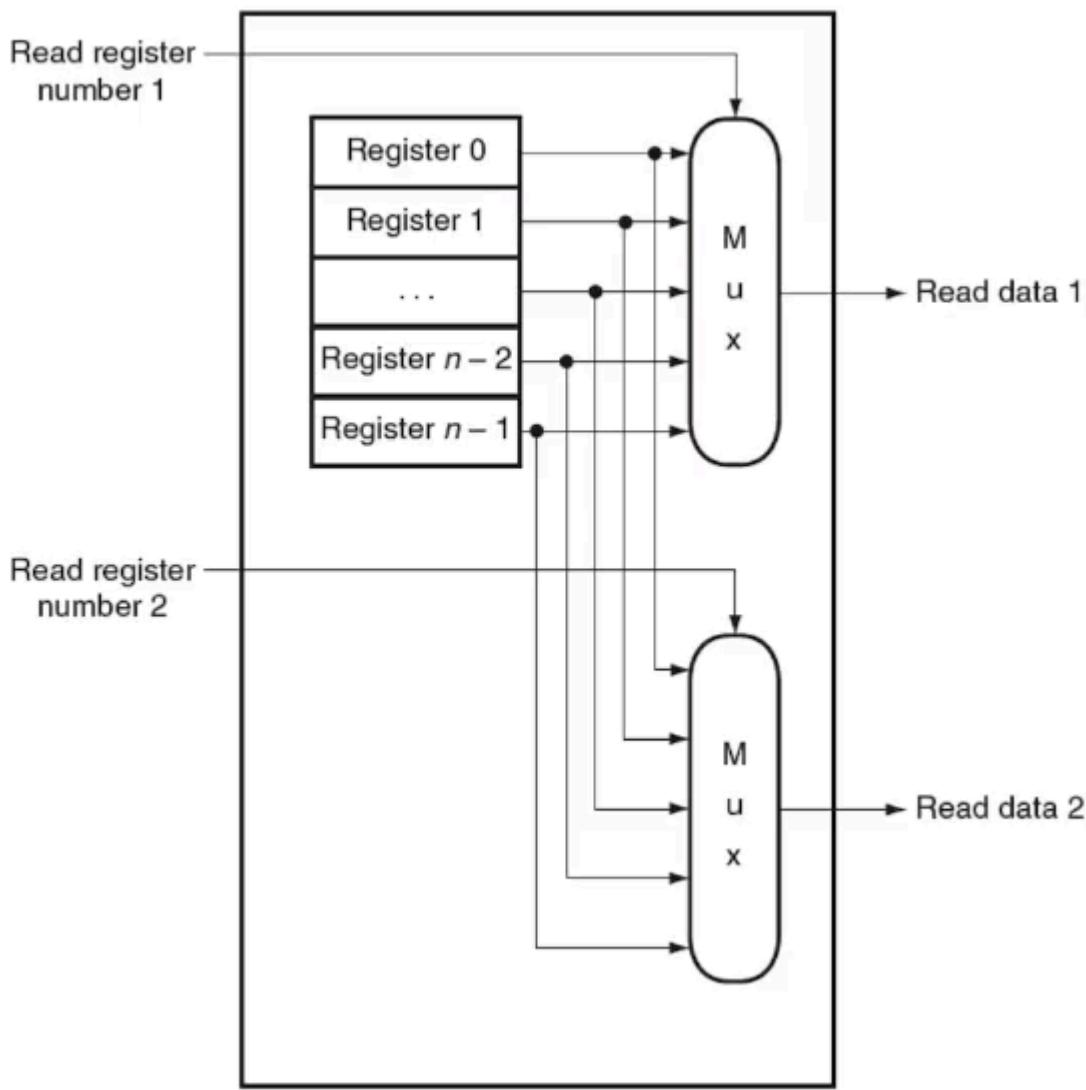
Gli step sono :

- Per prima cosa leggiamo tramite Read register number 1 e 2 i due registri (x4 e x5)
- Passiamo i loro valori, tramite "read data" 1 e 2, all'ALU che li somma
- L'ALU ritorna a "write data" il valore sommato
- Viene passato "x2" a "write register" per sapere dove dobbiamo scrivere il risultato della somma
- Impostiamo "write" ad 1
- Salviamo il valore ottenuto dalla ALU in x2.

Selezionamento dei registri

Abbiamo detto sopra che Read register number 1/2 selezionano dei registri, ma non come. Per selezionare dei registri quindi, si usa un multiplexer (lezione 9, un circuito combinatorio che dati 2^n input, permette di selezionare in uscita 1 solo di questi input avendo n ingressi di controllo)

Porte di lettura



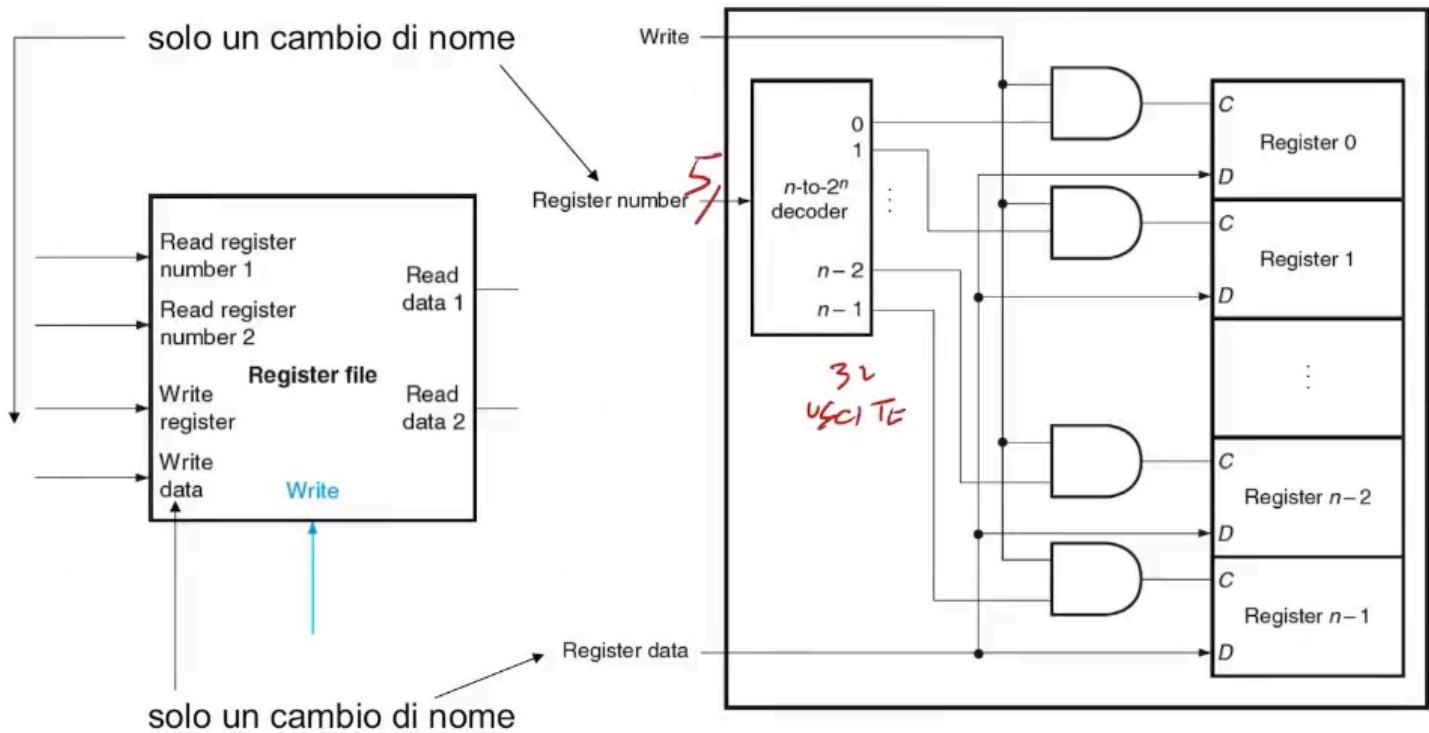
Se passiamo quindi al multiplexer 1, il valore "1" questo farà uscire da "read data 1" il valore contenuto in "register 1".

Scrittura di un registro

Abbiamo detto sopra che "write register" selezionano dei registri, ma non come. Per selezionare dei registri quindi, si usa un decoder (lezione 9, un circuito combinatorio che permette di scegliere uno di 2^n output, avendo n input). Ai decoder dobbiamo aggiungere, come già detto, write. Se mettiamo in AND il segnale restituito dal decoder con write (per ogni registro), riusciamo a scrivere solamente nel

registro selezionato (se non mettessimo il decoder, scriverebbero su tutti i registri).

Porta di scrittura



Tipi di memoria RAM

- SRAM : RAM statiche (flip-flop tipo D), estremamente veloci, utilizzate per realizzare le cache
- DRAM : RAM dinamiche (transistor con condensatore), vanno rinfrescati, offrono grandi capacità ma più lente

Macchine a stati finiti

Dispositivo automatico in grado di interagire con l'ambiente esterno. A fronte di uno stimolo di ingresso (input), esibisce un comportamento in uscita (output) che dipende anche da informazioni memorizzate in elementi interni (stati).

Le macchine che prendiamo in considerazione hanno memoria finita.

Comportamenti di una macchina a stati finiti

- Legge un simbolo in ingresso (che appartiene ad un insieme finito A)
- Produce un simbolo in uscita (che appartiene ad un insieme finito B)
- Cambia il proprio stato interno (la memoria è finita, quindi l'insieme Q degli stati interni è finito)

Descrizione di una macchina a stati finiti

Una macchina a stati finiti (detta anche automa) M è descritta da una quintupla : $M = \{A, B, Q, o, s\}$

- A : insieme finito di simboli di ingresso
- B : insieme finito di simboli di uscita
- Q : insieme finito di simboli di stato
- o : funzione di uscita.

- $A \times Q \rightarrow B$ funziona di uscita (per la macchina di Mealy - ovvero, la funzione di output dipende dallo stato e dall'input)
- $Q \rightarrow B$ per la macchina di Moore (ovvero, la funzione di output dipende solo dallo stato)
- $s : A \times Q \rightarrow Q$ funzione di cambiamento di stato

Esempio del distributore automatico di biglietti

Abbiamo un distributore automatico che :

- accetta solo monete grandi (MG) e monete piccole (MP)
- un biglietto viene emesso quando vengono ricevute una MP ed una MG
- l'ordine di immissione delle monete non è rilevante
- non viene dato resto, è necessario sempre introdurre MP e MG
- vengono restituire le monete "non adeguate"

Insieme simboli di ingresso

- $A = \{MP, MG\}$ (2 stati, quindi 1 bit per codificare)

Insieme simboli di uscita

- $B = \{\text{ancora}, \text{restituisci}, \text{emetti}\}$ (3 stati, quindi 2 bit per codificare)

Insieme degli stati

- $Q = (3 \text{ stati, quindi 2 bit per codificare})$
 - q_0 : non è stata inserita nessuna moneta
 - q_1 : è stata inserita una MP
 - q_2 : è stata inserita una MG

Non esiste q_3 perché quando inseriamo (ad esempio) MP e MP, l'ultima moneta messa viene restituita, quindi torniamo a $q_{1/2}$ (in base a che moneta abbiamo messo) (idem per quando inseriamo MP e MG, la macchina da un biglietto e ritorna a q_0).

Tabella di transizione (o tabella di stato)

	<i>MP</i>	<i>MG</i>
<i>q0</i>	<i>q1/ancora</i>	<i>q2/ancora</i>
<i>q1</i>	<i>q1/restituisci</i>	<i>q0/emetti</i>
<i>q2</i>	<i>q0/emetti</i>	<i>q2/restituisci</i>

Dalla tabella sopra, possiamo vedere ogni possibile stato della macchinetta

Dalla tabella sotto vediamo che :

- Abbiamo scelto (senza alcun criterio specifico) che la moneta piccola vale 0 e quella grande vale 1

Codifica delle **tre** possibili uscite (2 bit)

o₀	o₁	significato
0	0	ancora
0	1	emetti
1	0	restituisci
1	1	--

Codifica dei **tre** possibili stati (2 bit)

s₀	s₁	Significato
0	0	q0
0	1	q1
1	0	q2
1	1	--

Codifica dei **due** possibili ingressi (1 bit)

a	significato
0	MP
1	MG

L'insieme (finito) degli stati **Q** è formato da
 q0: non è stata inserita nessuna moneta
 q1: è stata inserita una MP
 q2: è stata inserita una MG

Macchina a stati finiti

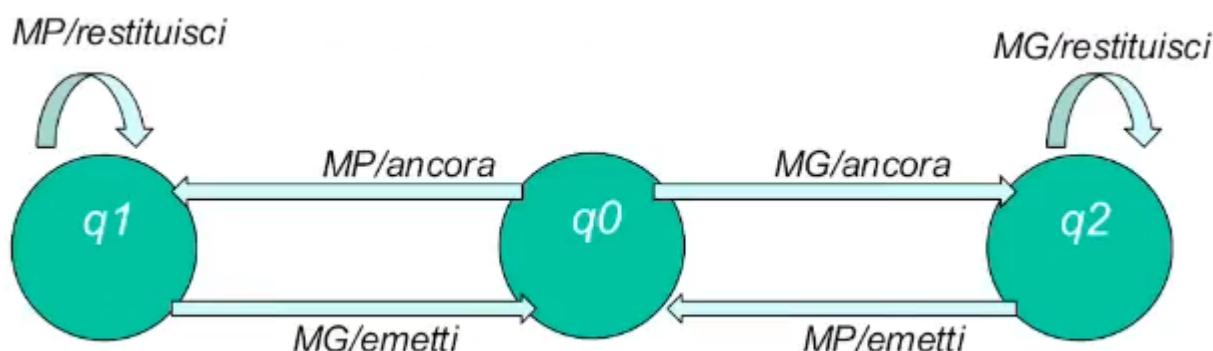


Tabella di verità

Come la scriviamo? Dobbiamo prendere in considerazione alcune cose, ovvero :

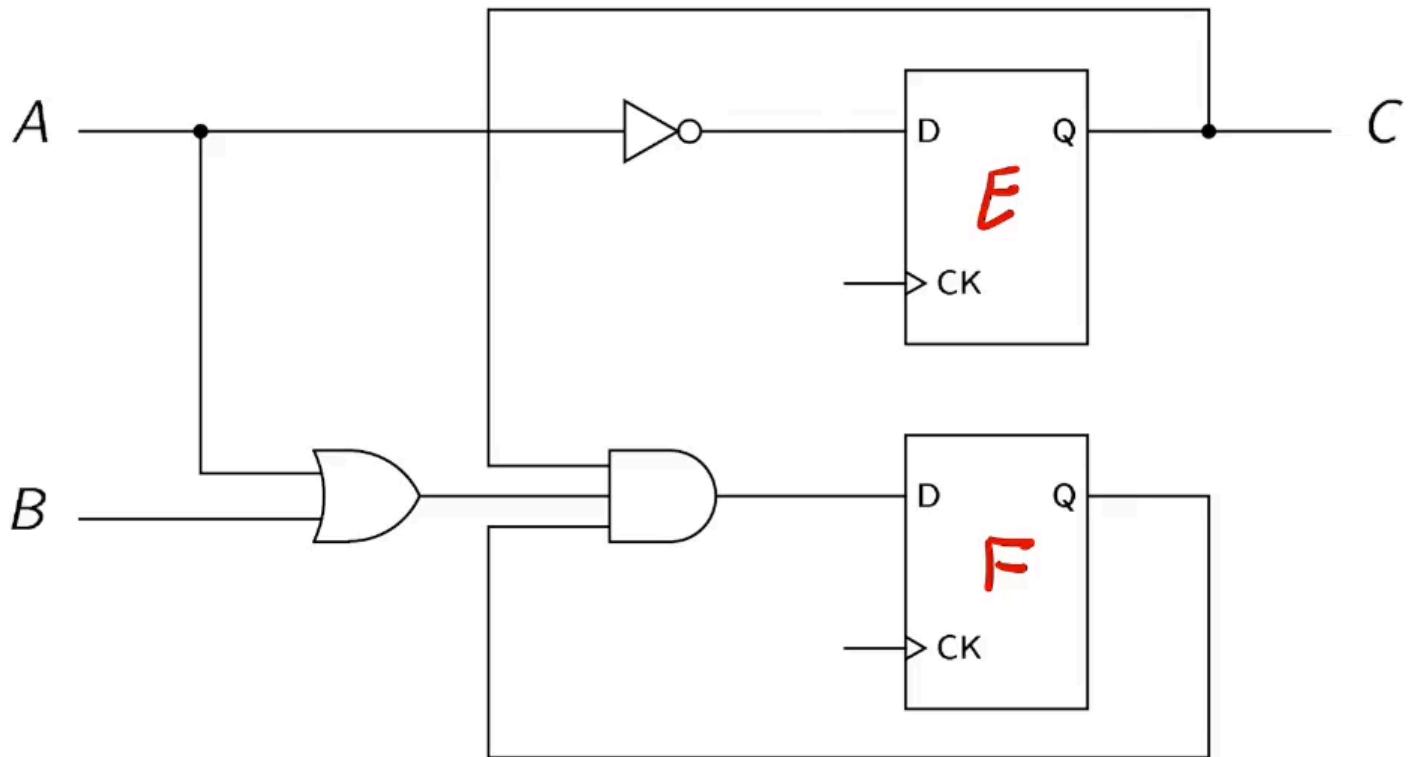
- Questo non è un normale circuito combinatorio, non abbiamo solo input, ma abbiamo anche degli stati interni.
- Quanti bit dobbiamo mettere? Dobbiamo mettere i bit dell'input (in questo caso solo 1 bit) più i bit per rappresentare gli stati interni (in questo caso 2 bit). Quindi dobbiamo mettere 3 bit (ovvero $2^3 = 8$ righe).
- Quali outputabbiamo? Abbiamo 4 bit per gli output, perché? Beh,abbiamo 2 bit per gli stati di output (Q) e 2 bit per gli stati futuri (s').

Adesso possiamo creare la tabella. Iniziamo mettendo tutti i possibili stati sugli input

	A	S_0	S_1	Q_0	Q_1	S'_0	S'_1	STATO S_m FUTURO
MP	0	0	0	0	0	0	1	→ PARTENDO DA NULLA E PICCOLA, "ANCORA"
	0	0	1	1	0	0	1	INSERENDO UNA MONETA DI JENTA LO STATO "ANCORA"
	0	1	0	-	-	-	-	OUTPUT : "RESTITUISCI"
M6	0	1	1	-	-	-	-	STATO NON USATO
	1	0	0	0	0	1	0	
	1	0	1	0	1	0	0	
	1	1	0	1	0	1	0	
	1	1	1	-	-	-	-	

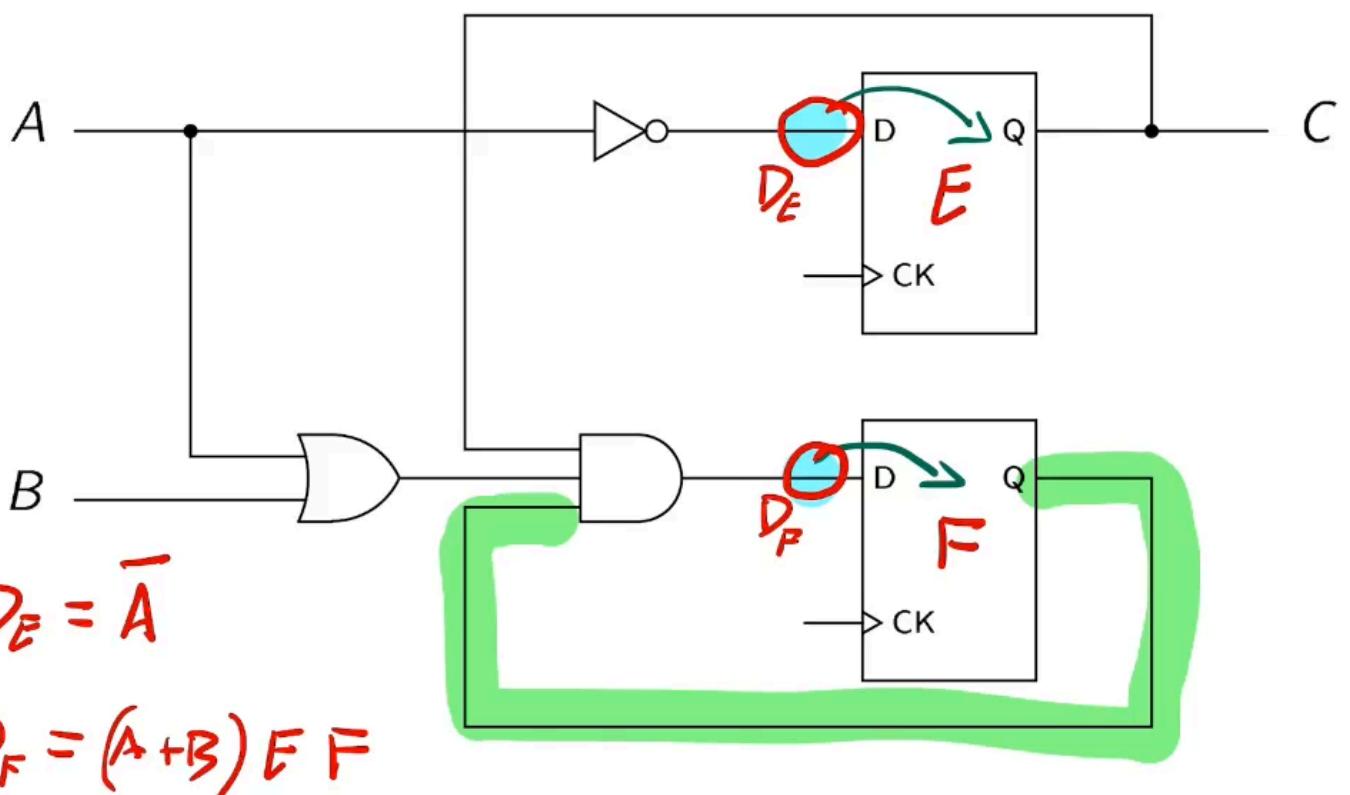
Analisi di reti sequenziali sincrone

Partendo da un circuito, dobbiamo analizzarlo e ottenere la tabella di verità. Guardiamo un esempio :



Si può analizzare seguendo questo ragionamento :

- Quanti input abbiamo? 2. Quindi, quante configurazioni (per gli input)? 4
 - Quanti bit in uscita abbiamo? 1. Quindi? 2 possibili uscite 1 o 0.
 - Cosa abbiamo nel circuito? 1 OR, 1 AND, 1 NOT e due Flip-Flop D (E e F).
 - Ci sono due Flip-Flop D, quindi, il valore "D" andrà in "Q", quando il clock è sul fronte di salita.
- Analizziamo adesso la rete e cerchiamo di ottenere i vari punti :



Come vediamo dall'immagine, possiamo calcolare :

$$D_E = \bar{A}$$

$$D_F = E(A + B)F$$

$$C = E$$

Le prime due (D_n) servono per calcolare lo stato futuro a partire dagli input A e B e dallo stato attuale. L'ultima ($C = E$) serve per calcolare l'output.

Partendo dall'ultimo esempio, ecco come possiamo sintetizzare :

Stato presente		Ingressi		Stato futuro		Uscite
E	F	A	B	D_E	D_F	C
0	0	0	0	1	0	0
0	0	0	1	1	0	0
0	0	1	0	0	0	0
0	0	1	1	0	0	0
0	1	0	0	1	0	0
0	1	0	1	1	0	0
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	1	0	1
1	0	0	1	1	0	1
1	0	1	0	0	0	1
1	0	1	1	0	0	1
1	1	0	0	1	0	1
1	1	0	1	1	1	1
1	1	1	0	0	1	1
1	1	1	1	0	1	1

Sintesi di reti sequenziali sincroni

Partendo dall'automa, sviluppare il circuito. (approfondire meglio l'argomento in modo autonomo)

Processore RISC-V

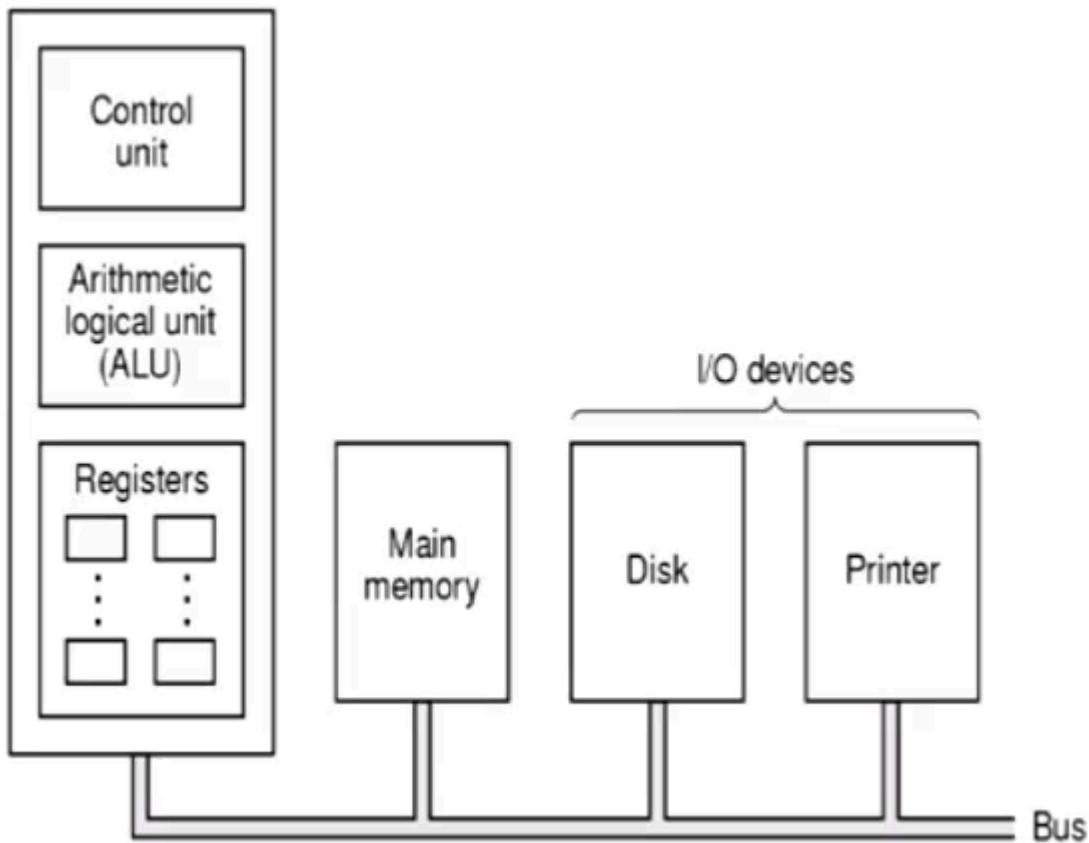
In maniera didattica, vedremo come fare un semplice processore RISC-V. Più avanti vedremo un processore completo che può eseguire tutte le istruzioni, per ora ci interessa che faccia :

- add
- sub
- lw
- sw

Cosa ci serve se vogliamo fare una macchina di Von Neumann? Di seguito un'immagine per

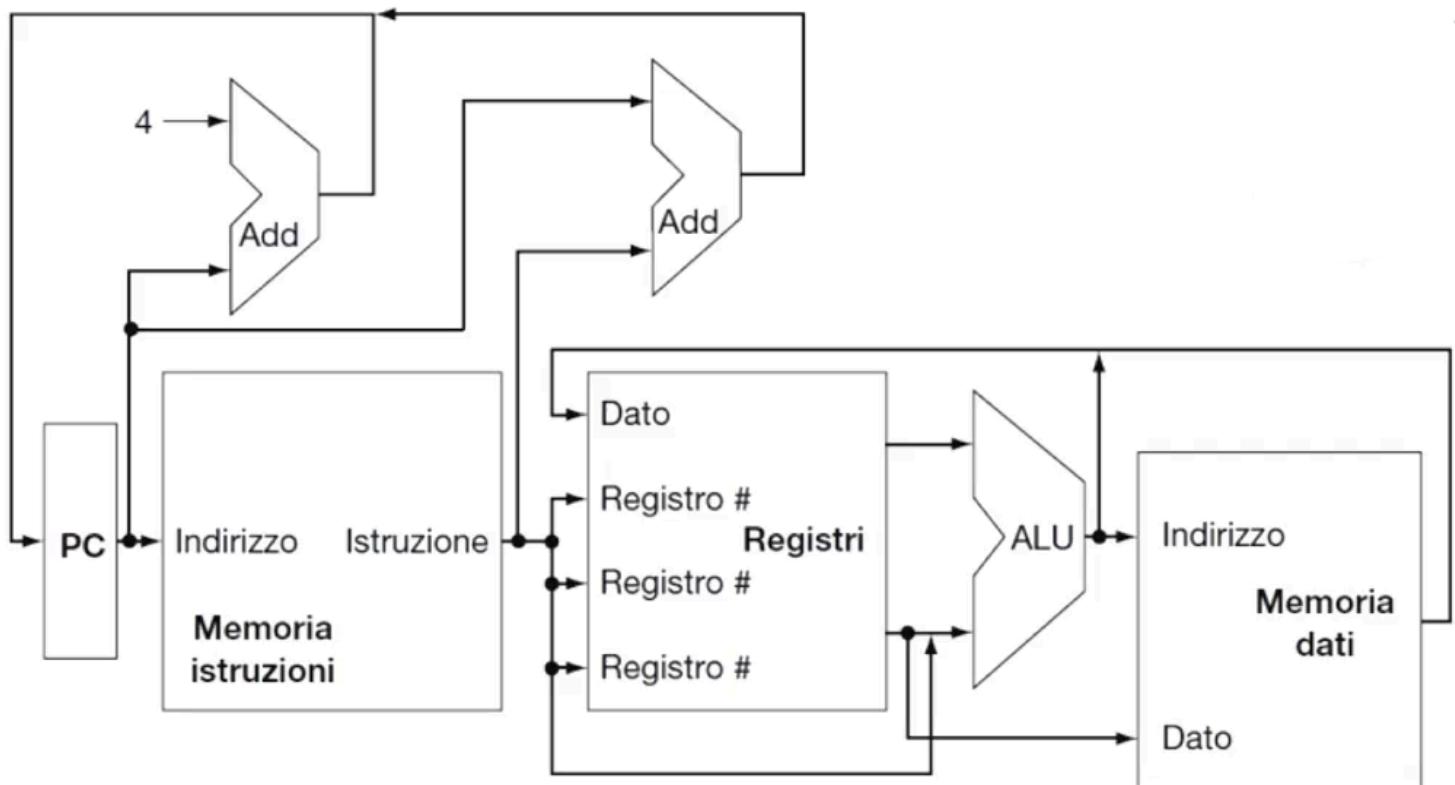
ricordare cosa serve :

Central processing unit (CPU)



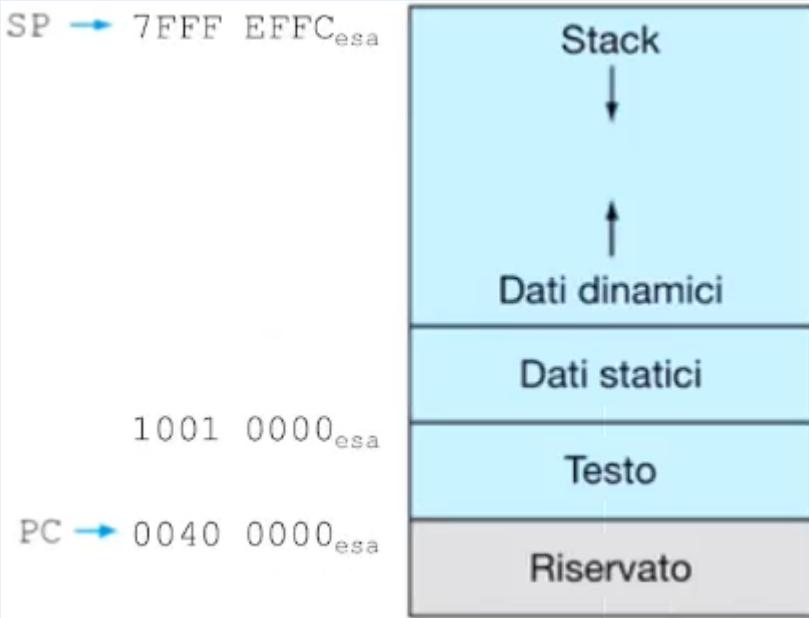
Data-Path

Il data path è formato dai registri, l'ALU e alcuni bus. È il percorso dei dati che permettono al processore di eseguire un'istruzione.



Note

Nota bene, non ci sono in realtà due memorie diverse (istruzioni e dati), ma è una sola memoria divisa poi internamente. È il modello di pila già visto. Le due memoria nello schema sopra ci servono solo per lo scopo di disegnare.



La "memoria delle istruzioni" fa riferimento alla parte "testo". Il PC infatti fa riferimento sempre a qualcosa dentro questa area "testo".

☰ Example

Ad esempio, se dobbiamo fare un'add prenderemo il valore di 2 registri, li sommeremo e poi scriveremo il risultato su un registro.

Fetch

- Il PC (program counter) fornisce un indirizzo della memoria delle istruzioni
- Per semplificazione, assumiamo che la "memoria delle istruzioni" sia "read-only"
- Assumiamo anche che ci siano 2 memorie diverse (una per le istruzioni ed un'altra per i dati)

Decode

- La ALU deve "capire" il tipo di istruzione (ad esempio, il numero d'ordine dei registri contenenti gli operandi può essere letto nei campi opportuni dell'istruzione stessa)
- L'unità di controllo (non mostrata nella figura sopra) è la parte che effettivamente decodifica le istruzioni

Execute

Una volta caricati gli operandi, si può :

- Eseguire un calcolo effettivo (operazioni aritmetico-logiche su interi)
- Elaborare il loro contenuto per determinare un indirizzo di memoria (nel caso di load o store)
- Eseguire un confronto (salto condizionato)

Metodologia di temporizzazione

Il segnale di clock determina quando gli elementi di stato scrivono la loro memoria interna. Gli ingressi a un elemento di stato devono raggiungere un valore stabile prima che il fronte attivo del clock provochi l'aggiornamento dello stato.

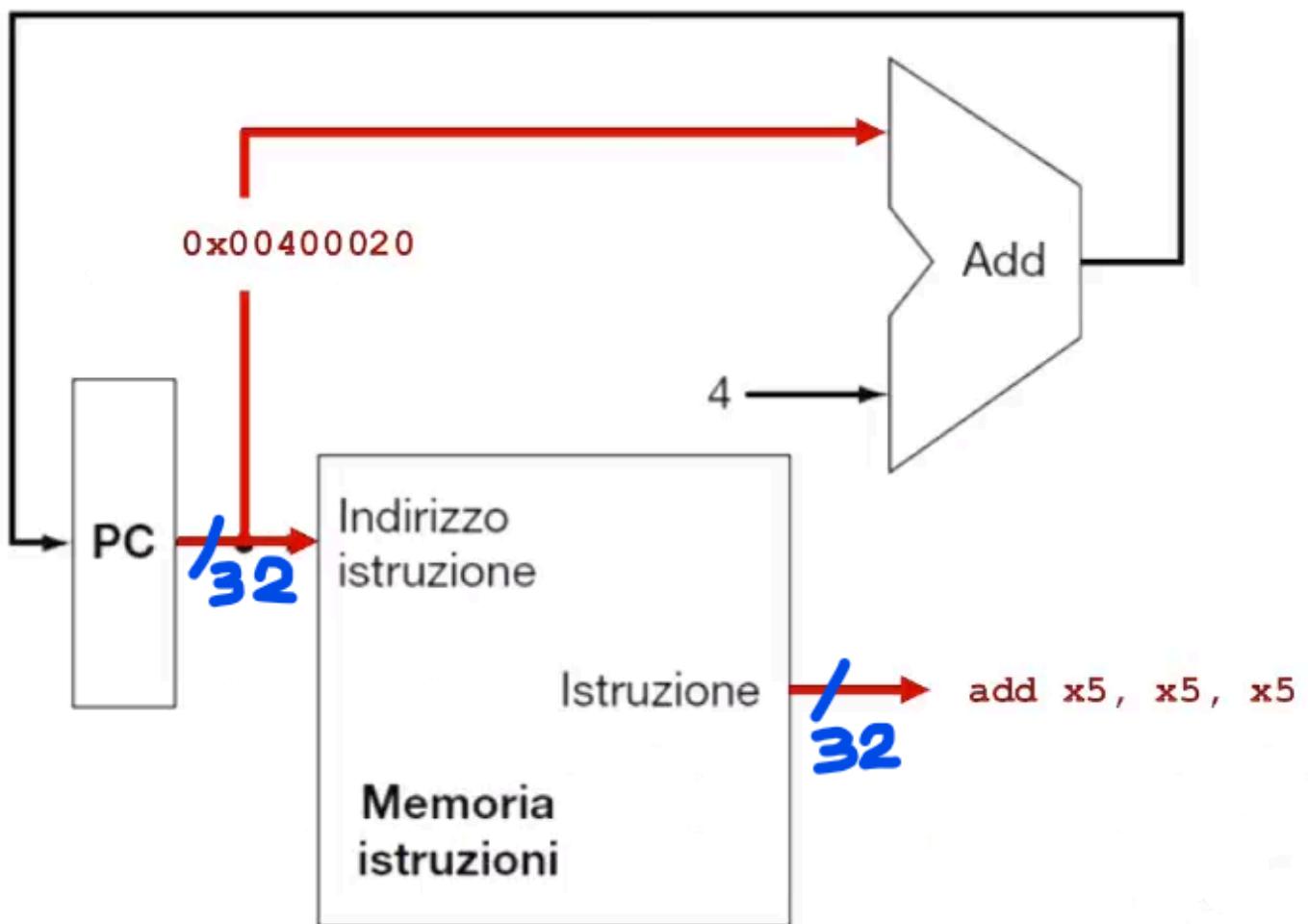
La metodologia sensibile ai fronti (edge-triggered) permette di leggere e scrivere un elemento di stato nello stesso ciclo di clock. Il ciclo di clock deve avere una durata sufficiente a garantire che gli ingressi siano stabili quando arriva il fronte attivo del clock.

Incremento del program counter

Il program counter manda l'indirizzo dell'istruzione corrente in due posti :

1. Memoria istruzione : si occupa di "capire l'istruzione".
2. Adder : aumenta, staticamente, l'indirizzo per avere l'indirizzo della prossima istruzione e il risultato lo rimanda al program counter (in un'architettura a 32/64 bit, l'indirizzo viene incrementato di 4 byte (32 bit)).

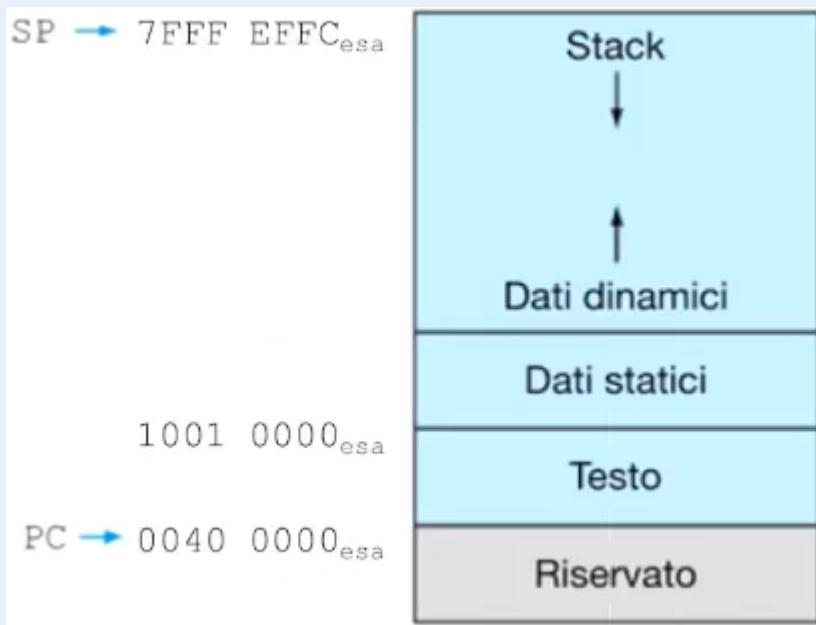
L'incremento avviene in contemporanea con l'accesso alla memoria (per la figura sotto, si intende che quando l'indirizzo esce da PC, va contemporaneamente sia a "Memoria istruzioni" sia all'add (ALU) che incrementa staticamente di 4).



Note

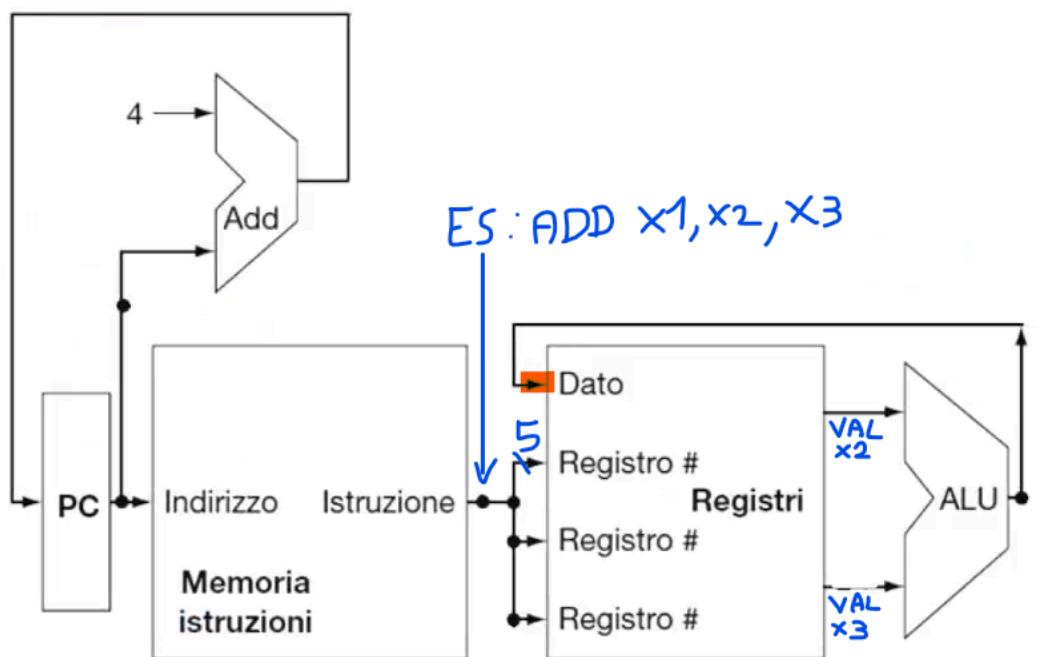
- Il PC non cambia il suo contenuto fino al primo fronte di salita.

- La "memoria istruzioni" non ha un segnale di controllo per scrivere, perché non si può scrivere nel segmento di testo. Guarda figura sotto per capire. (Dividiamo concettualmente "memoria istruzioni" e "memoria dati", ma fisicamente sono una cosa sola).



Data path per le istruzioni di tipo R

Nel data path per le istruzioni di tipo R troviamo sempre il circuito come sopra e in più troviamo il register file e l'ALU. Il register file ci serve per prendere prima i valori contenuti nei due registri salvati e poi per salvare il risultato dell'operazione calcolato dall'ALU nell'apposito registro indicato.



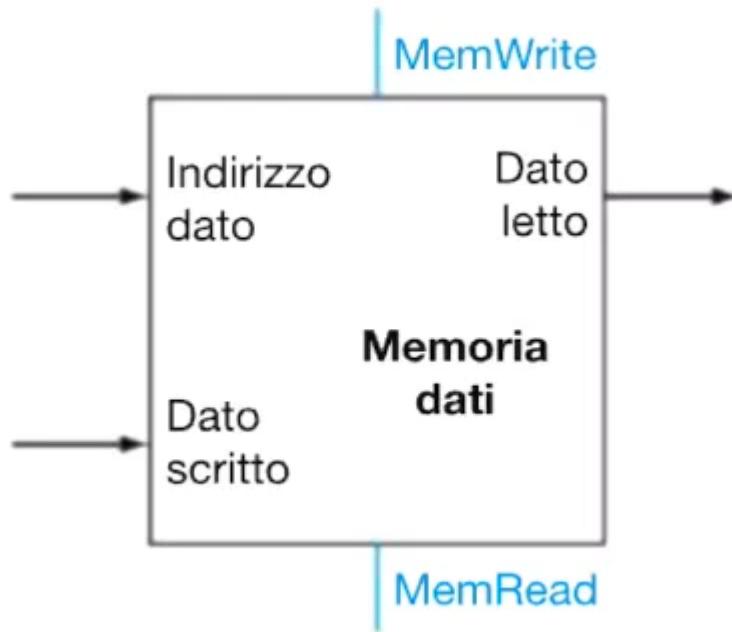
Gli ingressi con su scritto "Registro #" prendono 5 bit in ingresso sempre per il discorso che avendo 32 registri, ci servono 5 bit per selezionarli ($\log_2 32 = 5$).

L'ingresso "Dato" è evidenziato in rosso per far notare che li verrà salvato il risultato dell'operazione fatta dalla ALU, ma solo sul fronte di salita successivo.

Memoria dati

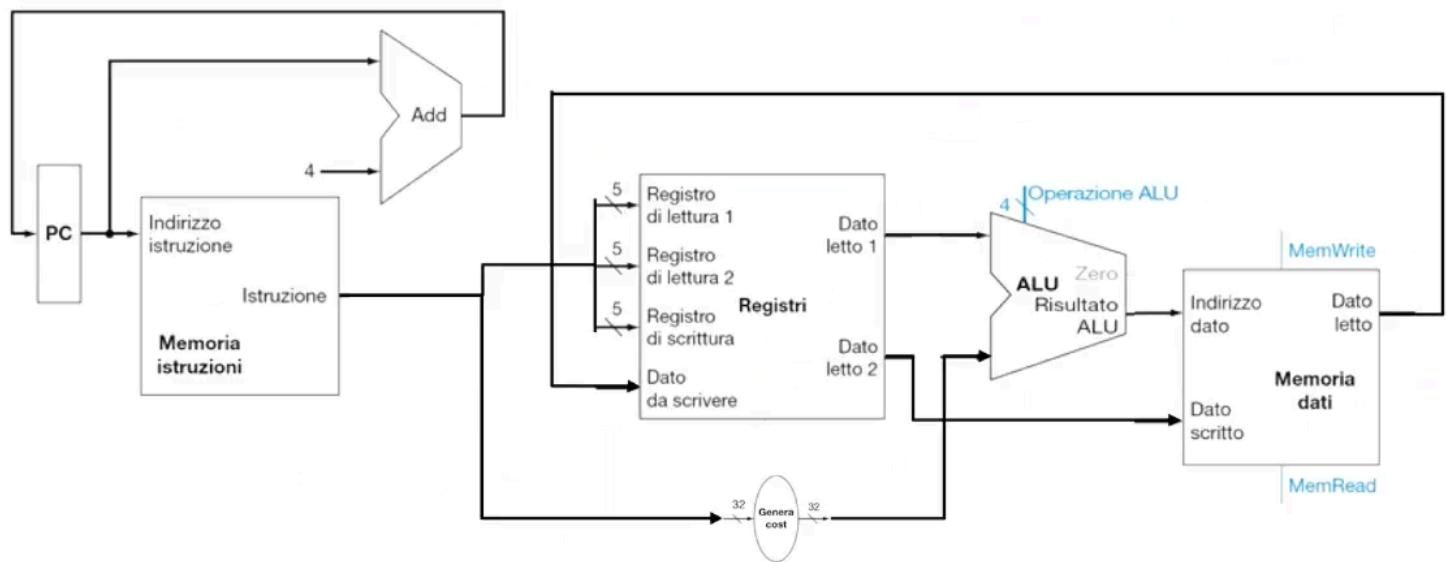
Questa è la memoria RAM. L'ingresso "indirizzo dato" è grande 32 bit. "Dato letto" invece è variabile, se facciamo "lb" o "lh" o... otteniamo risultati diversi (rispettivamente 8 e 16 bit). Troviamo anche due

segnali di controllo ("MemWrite" e "MemRead") che ci permetto di scrivere o leggere.



Genera cost (unità di estensione del segno)

A genera cost passiamo l'istruzione intera. Questa si occuperà di estrarre "cost" composta da 12 bit in molti tipi di istruzioni (tipo I, S) e poi ne estende il segno (perchè la ALU prende in ingresso sempre 32 bit, non ne bastano 12. Inoltre, in input, prende 32 bit perché la posizione dei vari "cost" nei vari tipi di istruzione è diverso, quindi gli si deve passare l'intera istruzione e poi lui si occuperà di prendere i bit giusti).



Questa sopra è il modo in cui "Genera cost" e "memoria dati" vengono implementati nel processore.

Supporto dell'istruzione beq (tipo SB)

BEQ è un'istruzione che richiede il confronto fra due valori e che consente il trasferimento del controllo a un altro indirizzo del programma a seconda del risultato del confronto. Le istruzioni di salto condizionato utilizzano il formato di tipo SB.



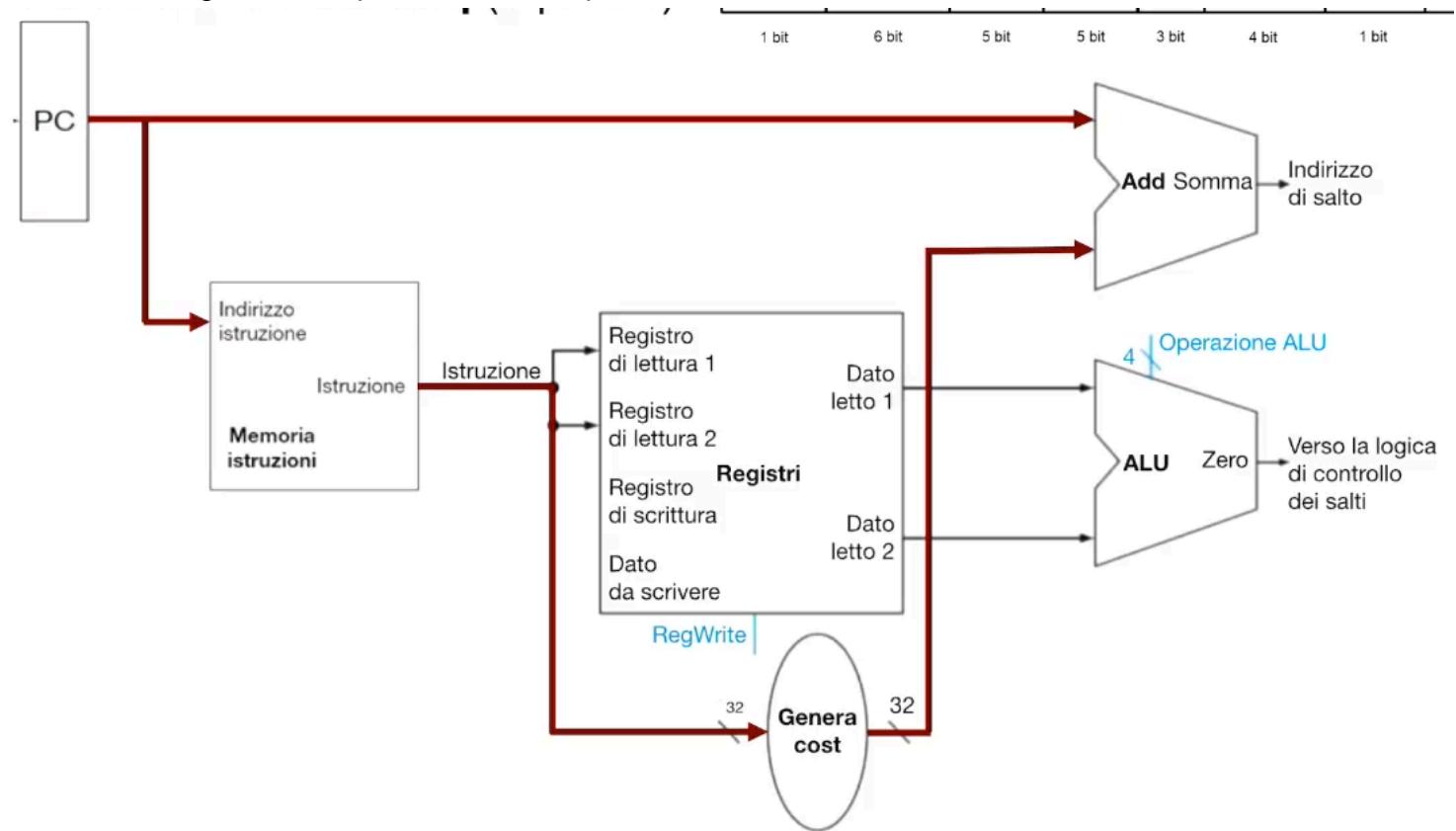
La struttura del formato SB

Come funziona?

Se due valori sono uguali, viene calcolata la distanza (può essere sia negativa che positiva, -4096/+4094) tra beq (l'attuale PC) e l'etichetta passata. Fatto il calcolo, si modifica il valore del PC.

Implementazione

Per implementare in modo hardware questa possibilità di salto abbiamo bisogno di alcuni componenti che abbiamo già visto sopra. Ecco come possiamo farlo :



Come possiamo vedere, dobbiamo prendere da "Genera cost" (che si occupa di fare l'estensione del segno e uno shift a sinistra di 1 bit) e sommarlo al program counter. Avremo poi bisogno di un qualche controllo per dire che se "zero" della ALU = 1 (ovvero i due numeri sono uguali), allora dobbiamo prendere il risultato dell'indirizzo di salto, altrimenti il normale incremento di 4 sul PC.

Note

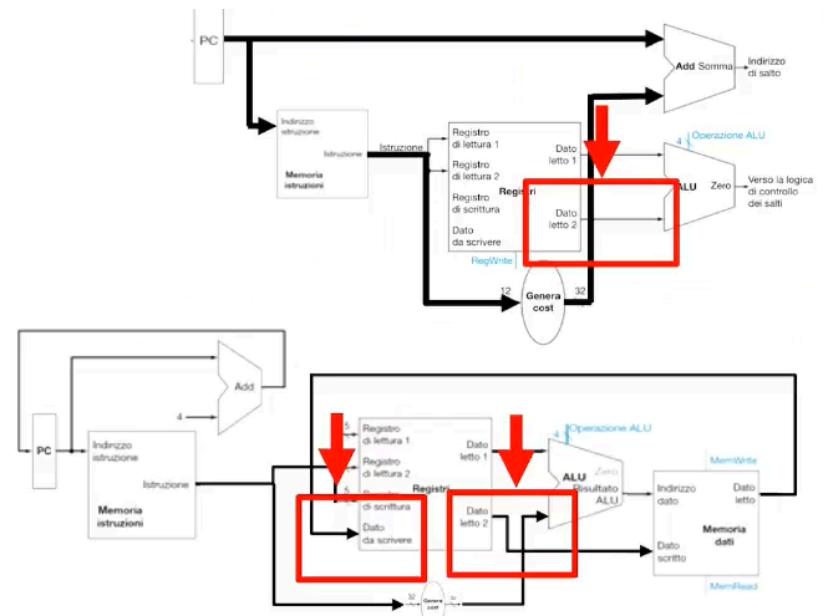
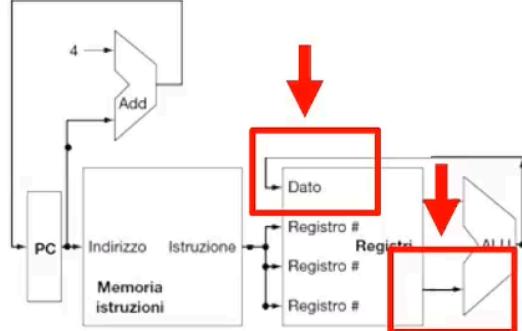
A cosa serve shiftare a sinistra di 1 bit il risultato? Beh, serve perché non ci servono mai salti da 1 bit, quindi si shifta di uno per avere più spazio. Infatti si può notare che ci sono solo 12 bit (quindi $2^{12} = 4096$), ma il range è da -4096 a +4094. Shiftando abbiamo appunto questo bit "in più" che ci permette di raddoppiare l'estensione del salto.

Unità di elaborazione unificata

Finora abbiamo visto come fare le istruzioni di tipo R, I, S, i salti e come usare genera cost. Abbiamo sempre usato circuiti diversi, ma molto simili tra di loro. Implementare ogni volta questi circuiti sarebbe uno spreco (oltre che poco fattibile a volte, non possiamo sdoppiare il Register file ad esempio), quindi dobbiamo trovare un modo di unificare questi circuiti.

Dove sono le differenze?

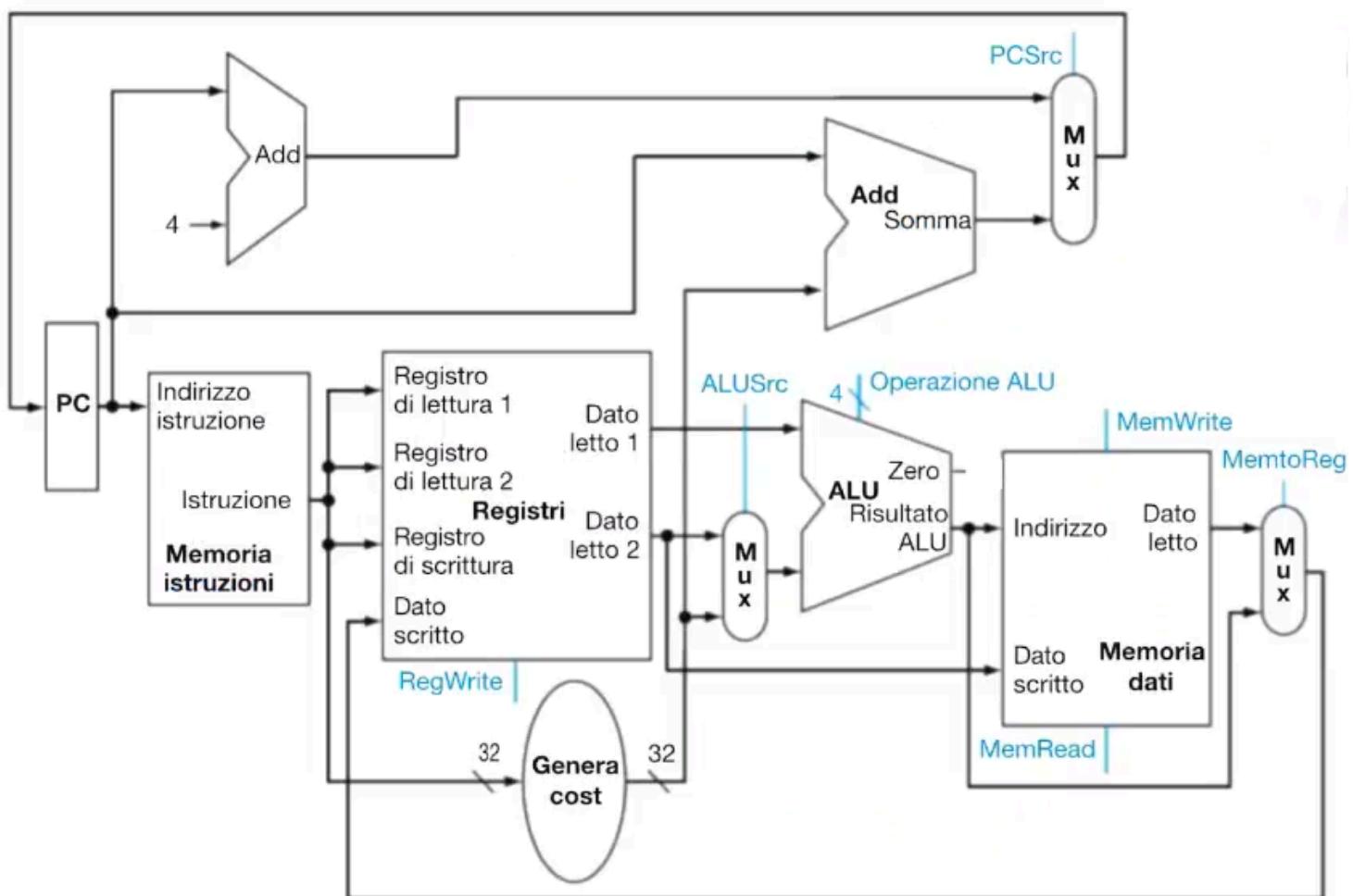
Ne vediamo alcune qua sotto :



Si nota che se unissimo tutto assieme, avremmo dei problemi. Prendiamo in esempio l'ingresso 2 della ALU. Nelle istruzioni R è direttamente collegato al register file, mentre nelle istruzioni di tipo S/I è collegato a genera cost, come possiamo fare? Beh, la soluzione è semplice, basta usare un multiplexer per decidere quale valore far passare.

Risultato

Collegando quindi tutti gli output e input che vanno in contrasto in modo corretto, otteniamo qualcosa del genere.



Notiamo però, che abbiamo molte linee di controllo che non abbiamo visto (ALUSrc, MemtoReg, PCSrc), chi le impone queste linee? Queste linee vengono impostate dall'unità di controllo.

Note

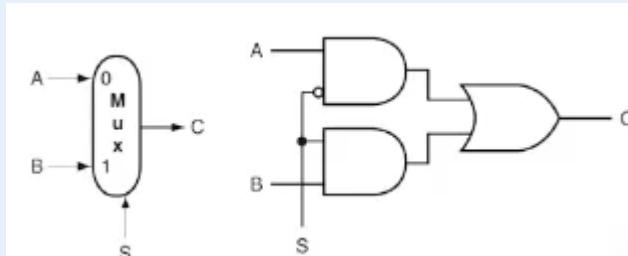
Nell'immagine sopra, i circuiti combinatori sono:

- Le varie ALU (ALU e 2 add, le quali sono ALU semplificate)
 - I multiplexer
 - Genera cost

Invece, sono sequenziali tutte le memorie (PC, memoria istruzioni/dati e register file), perché contengono i dati, quindi ovviamente l'output varia in base ad uno stato interno.

Note

Ricordo che un multiplexer (con 2 ingressi e 1 di controllo) è formato in questo modo :



Unità di controllo

L'unità di controllo è un circuito combinatorio che si occupa di prendere in ingresso un'istruzione da eseguire e da esse derivarne :

- Un segnale di scrittura per ciascun elemento di stato.
- Un segnale di selezione per ciascun multiplexer.
- I segnali di controllo per l'ALU (il controllo dell'ALU è particolare, conviene quindi progettarlo prima delle altre parti dell'unità di controllo).

L'unità di controllo è un componente essenziale e costruirne la tabella di verità sarebbe molto complesso, per questo non verrà trattato, ma cercheremo comunque di capire come funziona.

Ragionamento per la costruzione

Come possiamo quindi ottenere tutte le informazioni che vogliamo dall'istruzione? Beh, per prima cosa dobbiamo guardare il codop. In questo modo sappiamo di che istruzione si tratta. Inoltre dobbiamo vedere anche i possibili campi "funz3" e "funz7".

Istruzione (R)	funz7	rs2	rs1	funz3	rd	codop	Esempio
add	0000000	00011	00010	000	00001	0110011	add x1, x2, x3
sub (sottrazione)	0100000	00011	00010	000	00001	0110011	sub x1, x2, x3
	0000000	rs2	rs1	000	rd	0110011	ADD
	0100000	rs2	rs1	000	rd	0110011	SUB
	0000000	rs2	rs1	001	rd	0110011	SLL
	0000000	rs2	rs1	010	rd	0110011	SLT
	0000000	rs2	rs1	011	rd	0110011	SLTU
	0000000	rs2	rs1	100	rd	0110011	XOR
	0000000	rs2	rs1	101	rd	0110011	SRL
	0100000	rs2	rs1	101	rd	0110011	SRA
	0000000	rs2	rs1	110	rd	0110011	OR
	0000000	rs2	rs1	111	rd	0110011	AND

Come possiamo notare da questa immagine, add e sub (e anche altre istruzioni) hanno lo stesso codop, quindi per differenziali, abbiamo bisogno di funz3/7 (e da questi poi possiamo controllare la ALU ad esempio per mettere Binvert ad 1, etc...).

Nell'immagine sotto, possiamo vedere varie istruzioni e i loro valori in vari campi. Possiamo notare che lw e sw ad esempio, non hanno nulla nel campo "funz3/7", perché non servono. Invece, l'ingresso nella ALU sia per lw che sw sono uguali, perché? Beh, perché devono fare sostanzialmente la stessa cosa, ovvero la somma dell'offset.

Notiamo invece, che per istruzioni diverse, gli ingressi della ALU cambiano parecchio.

Controllo Operazioni della ALU

Termine **indifferente (don't care)**: una variabile di ingresso di una funzione logica che non ha effetto sull'uscita

Codice operativo istruzione	ALUOp	Operazione eseguita dall'istruzione	Campo funz7	Campo funz3	Operazione dell'ALU	Ingresso di controllo alla ALU
Iw	00	load di 1 parola	XXXXXXX	XXX	somma	0010
sw	00	store di 1 parola	XXXXXXX	XXX	somma	0010
beq	01	salto condizionato all'uguaglianza	XXXXXXX	XXX	sottrazione	0110
Tipo R	10	add	0000000	000	somma	0010
Tipo R	10	sub	0100000	000	sottrazione	0110
Tipo R	10	and	0000000	111	AND	0000
Tipo R	10	or	0000000	110	OR	0001

I 4 bit di controllo della ALU possono essere generati utilizzando una piccola unità di controllo che riceve in ingresso i campi funz7 e funz3 dell'istruzione e un campo di controllo su 2 bit, chiamato ALUOp :

- ALUOp = 00, somma per le istruzioni di load e store
- ALUOp = 01, sottrazione per le beq
- ALUOp = 10, l'operazione viene determinata dal contenuto dei campi funz7 e funz3

Per fare una tabella di verità derivata da questo ragionamento, abbiamo bisogno di 12 bit in input (2 bit per ALUOp, 7 per funz7 e 3 per funz3) e 4 in output (i 4 bit di "ingresso di controllo alla ALU"), ovvero una tabella da 4096 (2^{12}) righe.

Controllo operazioni della ALU

Ci sono molteplici livelli di decodifica :

1. L'unità di controllo principale imposta i bit ALUOp, poi utilizzati come ingresso dal secondo livello
2. Unità di controllo dall'ALU, usa i bit di ALUOp precedentemente impostati e genera i segnali effettivi dell'ALU.

Più livelli di controllo possono ridurre le dimensioni dell'unità di controllo principale, riducendo la latenza dell'unità di controllo (spesso l'unità di controllo è un elemento critico per la definizione della durata del ciclo di clock, quindi livelli diminuire la latenza fa diminuire anche la durata del ciclo di clock).

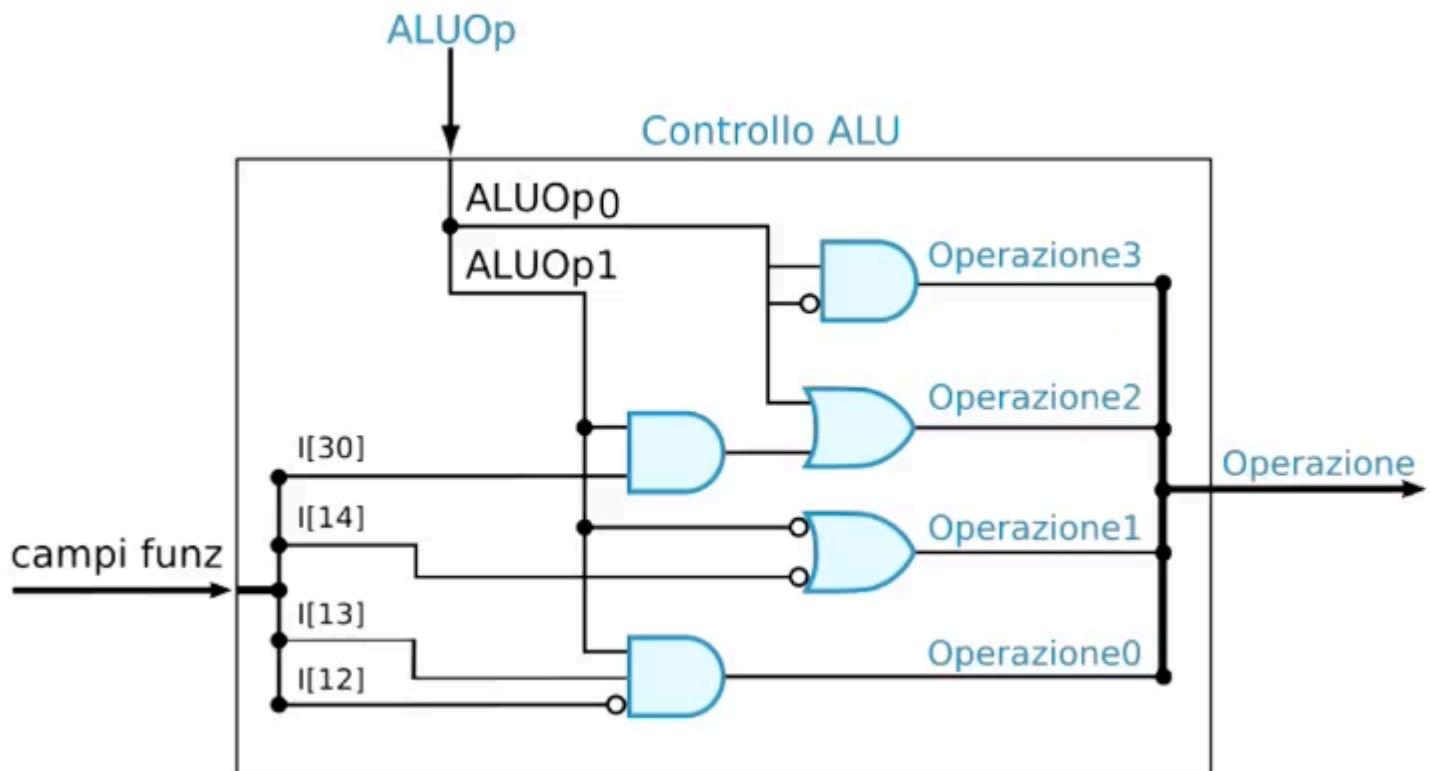
Tabella di verità per ottenere il controllo dell'ALU

Termini indifferenti
(don't care)

lw / sw	ALUOp		Campo funz7							Campo funz3			Operazione
	ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
lw / sw	0	0	X	X	X	X	X	X	X	X	X	X	0010
beq	X	1	X	X	X	X	X	X	X	X	X	X	0110
add	1	X	0	0	0	0	0	0	0	0	0	0	0010
sub	1	X	0	1	0	0	0	0	0	0	0	0	0110
and	1	X	0	0	0	0	0	0	0	1	1	1	0000
or	1	X	0	0	0	0	0	0	0	1	1	0	0001

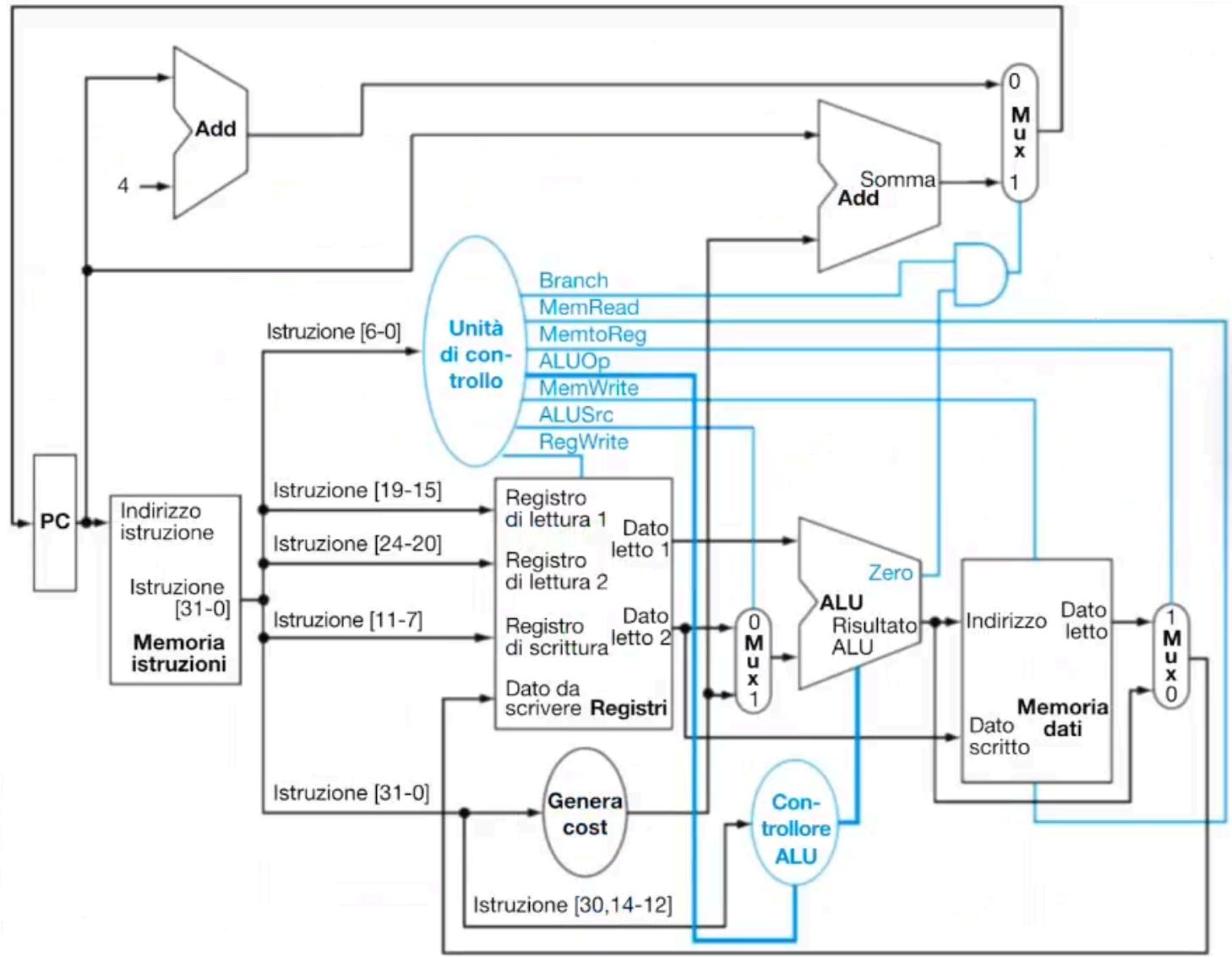
Tabella di verità: Corrispondenza tra i 2 bit del campo ALUOp e i bit dei campi funz con i 4 bit di controllo della ALU che selezionano l'operazione

Non è importante saperla. Non viene approfondita più di così, però è utile avere un'idea generale di come potrebbe funzionare.



L'unità di elaborazione con segnali di controllo

Detto tutto questo, possiamo vedere il circuito finito, il quale si comporrà nel seguente modo :

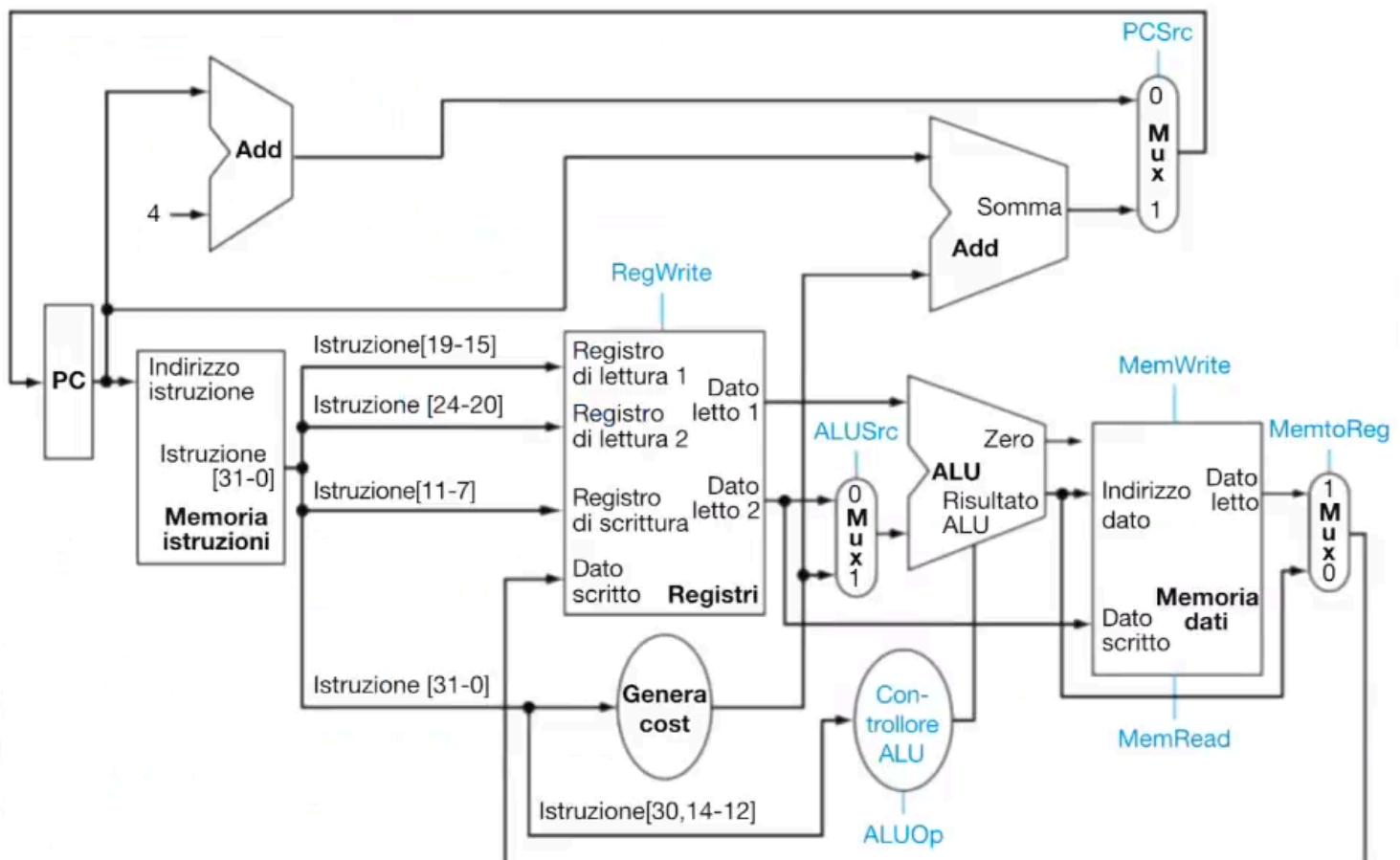


Recap formato delle istruzioni

Nome (posizione dei bit)	Campi					
	31:25	24:20	19:15	14:12	11:7	6:0
(a) Tipo R	funz7	rs2	rs1	funz3	rd	codop
(b) Tipo I	immediate[11:0]		rs1	funz3	rd	codop
(c) Tipo S	immed[11:5]	rs2	rs1	funz3	immed[4:0]	codop
(d) Tipo SB	immed[12,10:5]	rs2	rs1	funz3	immed[4:1,11]	codop

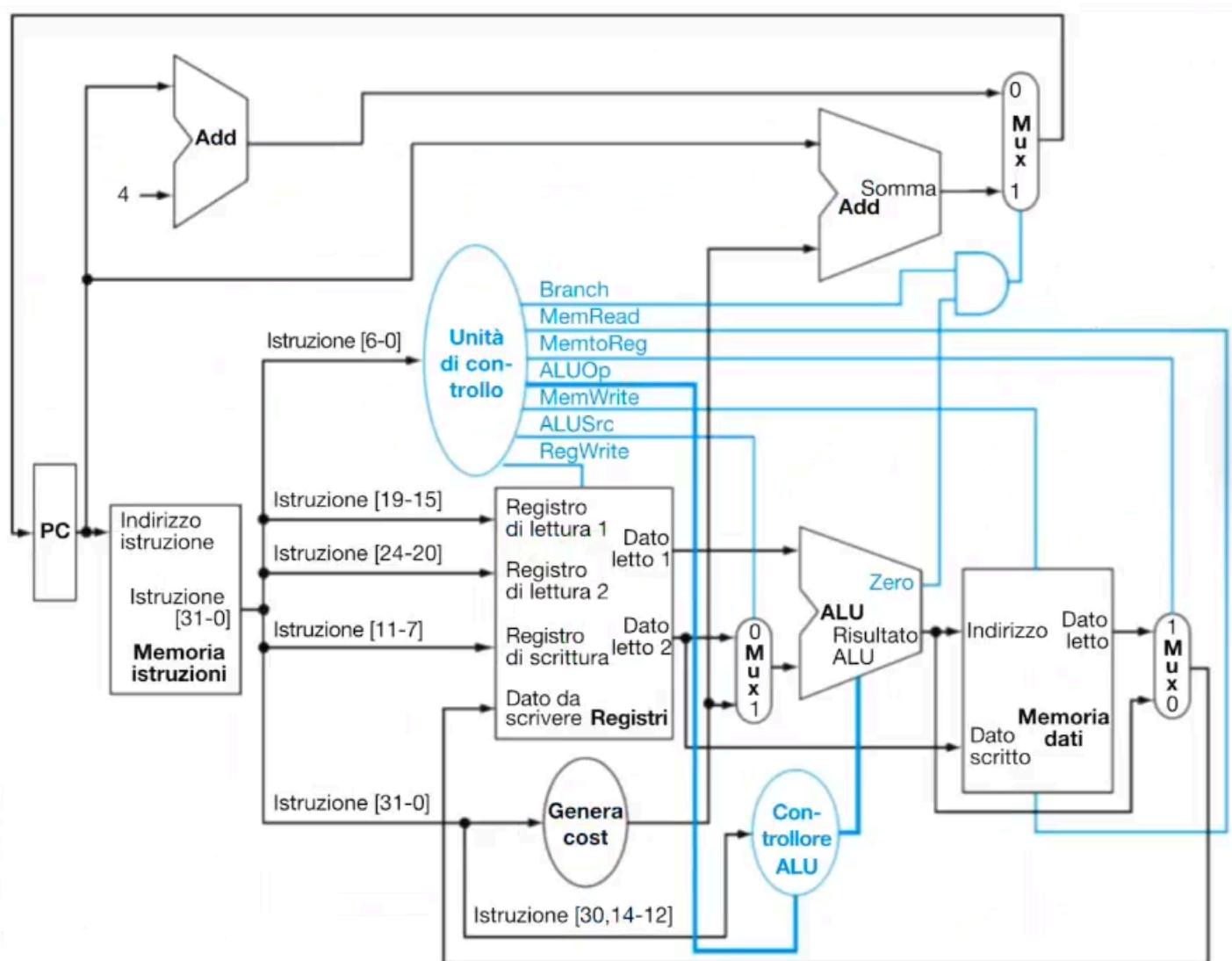
- Tipo R:** i campi rs1 e rs2 contengono il numero dei registri sorgenti e rd contiene il numero del registro destinazione. L'operazione da eseguire è codificata nei campi funz3 e funz7
- Tipo I, load:** rs1 è il registro base il cui contenuto viene sommato al campo immediato di 12 bit per ottenere l'indirizzo del dato in memoria. Il campo rd è il registro destinazione per il valore letto
- Tipo S, store:** rs1 è il registro base il cui contenuto viene sommato al campo immediato di 12 bit (suddiviso in 2 gruppi) per ottenere l'indirizzo del dato in memoria. Il campo rs2 è il registro sorgente il cui valore viene copiato nella memoria
- Tipo SB:** I registri rs1 e rs2 vengono confrontati. Il campo indirizzo immediato di 12 bit viene preso, il suo bit di segno esteso, fatto scorrere a sinistra di una posizione e sommato al PC per calcolare l'indirizzo di destinazione del salto

Recap segnali di controllo



Dall'immagine sopra, ecco un recap dei segnali di controllo da 1 bit :

- **RegWrite** : se vale 1, prende il dato in ingresso da "Dato scritto" e lo scrive in "registro di scrittura".
- **MemRead** : se vale 1, indica alla memoria se deve leggere un dato contenuto in "indirizzo dato" (serve perché l'indirizzo presentato in "indirizzo dato" potrebbe non essere valido, quindi c'è bisogno di un controllo esplicito che mi dica che voglio leggere da quell'indirizzo).
- **MemWrite** : se vale 1, devo scrivere in memoria il dato che trovo in "dato scritto" nell'indirizzo contenuto in "indirizzo dato".
- **MemtoReg** : è il segnale di controllo di un multiplexer che decide se prendere il dato letto dalla memoria o restituito dalla ALU (ad esempio, quando facciamo una load, vogliamo 1, perché leggiamo un dato in memoria. Invece, se facciamo una add, vogliamo 0, perché facciamo la somma tra due valori e mettiamo poi il risultato in un registro)
- **PCSrc** : è il segnale di controllo di un multiplexer che vale 0 se il PC deve essere incrementato semplicemente di 4 oppure 1 se il PC deve essere incrementato di un offset generato da "genera cost" (ovvero quando facciamo un salto). (NOTA, questa linea è collegata in AND alla linea di uscita della ALU "zero" e all'uscita dell'unità di controllo "branch", vedi immagine sotto*).
- **ALUSrc** : è il segnale di controllo di un multiplexer che pilota l'ingresso 2 della ALU. Vale 0 se dobbiamo prendere un dato da un registro (quindi dal register file) oppure 1 se prendiamo un dato estrapolato dall'immediato dell'istruzione (e quindi generato da "genera cost").



Note

Come fa l'unità di controllo a sapere che un'istruzione è di branch/load/store...? Guardando il codop.

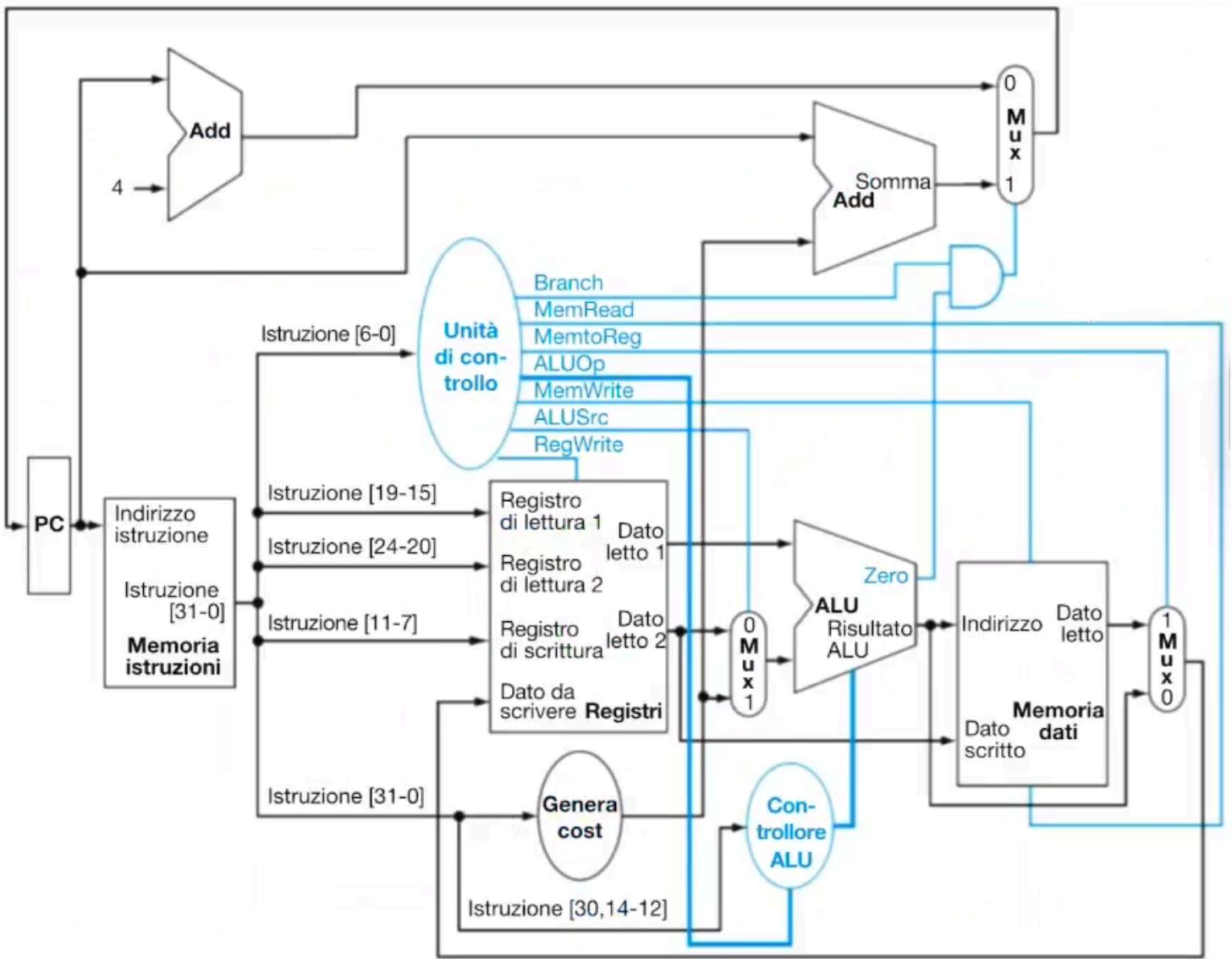
Valore delle linee di controllo per ogni caso

Di seguito un'immagine per ricapitolare il valore di ogni linea di controllo per ogni istruzione (vista ora) :

Istruzione	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
Tipo R	0	0	1	0	0	0	1	0
ld	1	1	1	1	0	0	0	0
sd	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

Esempio (utile per l'esame)

Durante l'esame, può essere che capiti un esercizio del genere, ovvero, data un'istruzione e l'immagine del circuito, venga chiesto cosa succede :



Istruzione :

```
add x1, x2, x3
```

Partiamo con dei ragionamenti ovvi che possiamo fare, ovvero ciò che non viene usato :

- Genera cost : non viene sicuramente usato perché "add" è un'istruzione di tipo R e quindi non contiene immediati
- La ALU "add" che somma un immediato al PC : non viene usato perché l'istruzione "add" non è un salto
- Memoria dati : non viene usata perché "add" non accede alla "memoria dati" (ram) perché i valori cercati (x_2 , x_3) sono contenuti nei registri (cache).

Poi possiamo procedere :

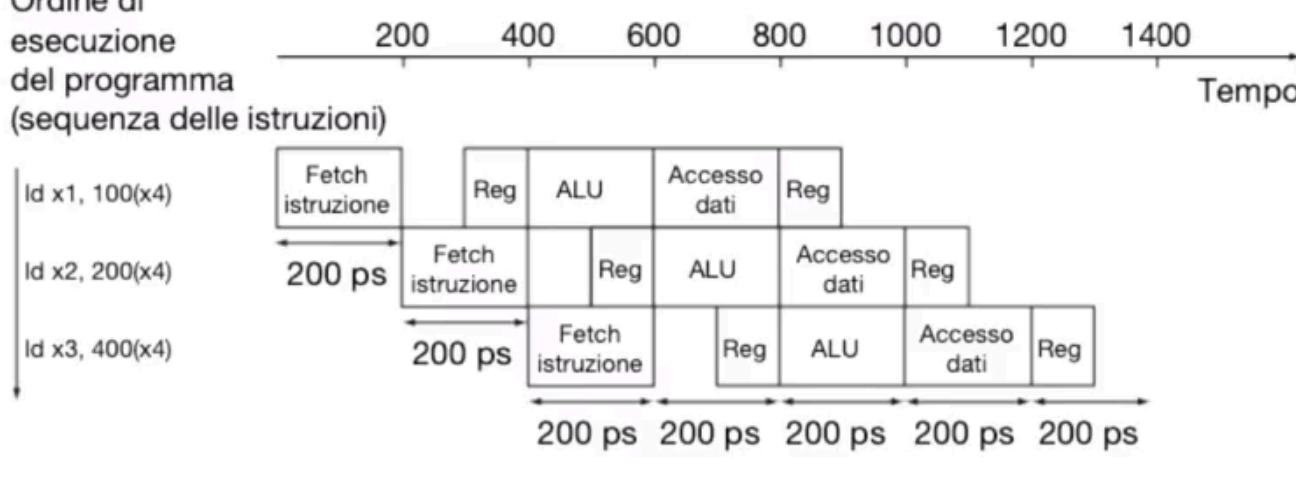
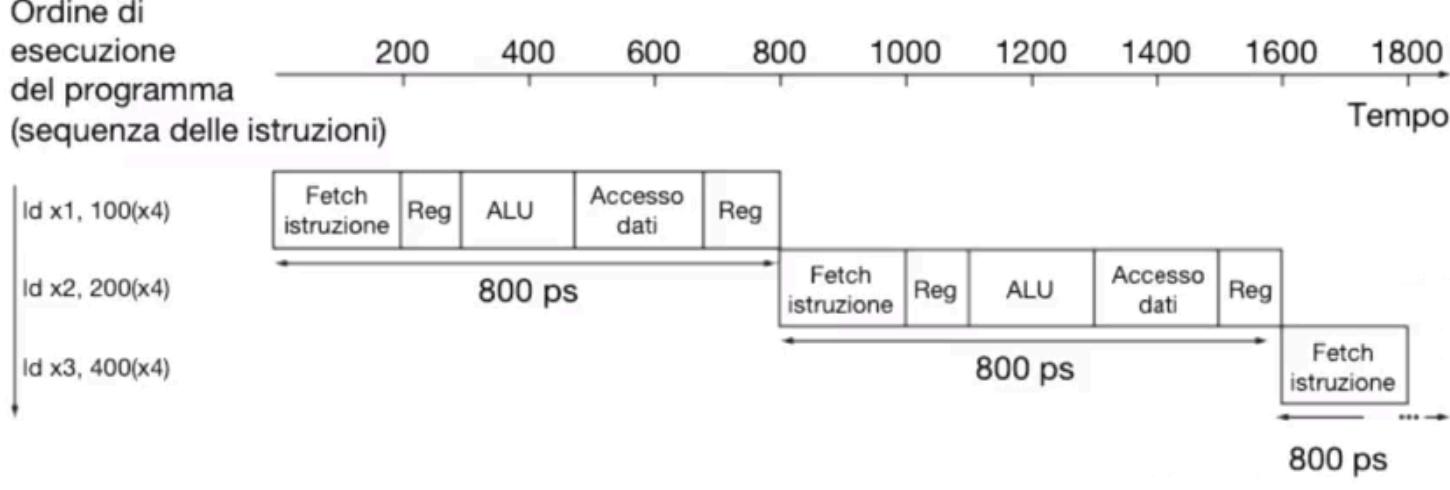
1. Il PC espone il proprio contenuto alla memoria delle istruzioni in "indirizzo di istruzione" e incomincia la fase di "fetch".
2. L'istruzione esce dalla memoria delle istruzioni e si dirama in vari punti (registro di lettura 1 (x_2 , ovvero "00010") registro di lettura 2 (x_3 , ovvero "00011"), registro di scrittura (x_1 , "00001"), unità di controllo, controllore ALU e genera cost (anche se non viene usata, è pur sempre collegata)).
3. L'unità di controllo ora sta facendo "decode" dell'istruzione e sta mettendo i valori adeguati a tutte le linee di controllo (es : branch = 0, MemRead = 0, MemtoReg = 0 (noi vogliamo che al register file

torni il valore calcolato dall'ALU, non dalla memoria dati), ALUOp = (qualcosa), MemWrite = 0, ALUSrc = 0 (vogliamo dato letto 2), RegWrite = 1)

4. Il register file presenta alla ALU da "dato letto 1" e "dato letto 2" i due valori da sommare.
5. Se il controllore ALU ha presentato all'ALU i valori dei 4 bit, allora l'ALU fa la somma dei valori. Altrimenti, se i valori non sono ancora stabili, la ALU presenterà il valore della somma con i vecchi 4 bit fino a quando però, i 4 bit diventano stabili e la ALU fa il giusto calcolo.
6. Il valore calcolato dall'ALU va verso il multiplexer con ingresso "MemtoReg" che vale 0, quindi fa passare il valore e arriva al register file.
7. Siccome RegWrite vale 1, il valore in "dato da salvare", viene salvato nel registro preso da "registro di scrittura". ATTENZIONE : il valore viene salvato al prossimo fronte di salita, non prima. È finito questo esempio. Possono ovviamente essere richieste cose diverse (ad esempio, una load, una store, un salto...).

Pipeline (approfondimento non richiesto)

In maniera semplice, il pipelining è l'ottimizzazione dell'esecuzione del programma facendo in modo parallelo più istruzioni. Di seguito un immagine per chiarire un po' il concetto :

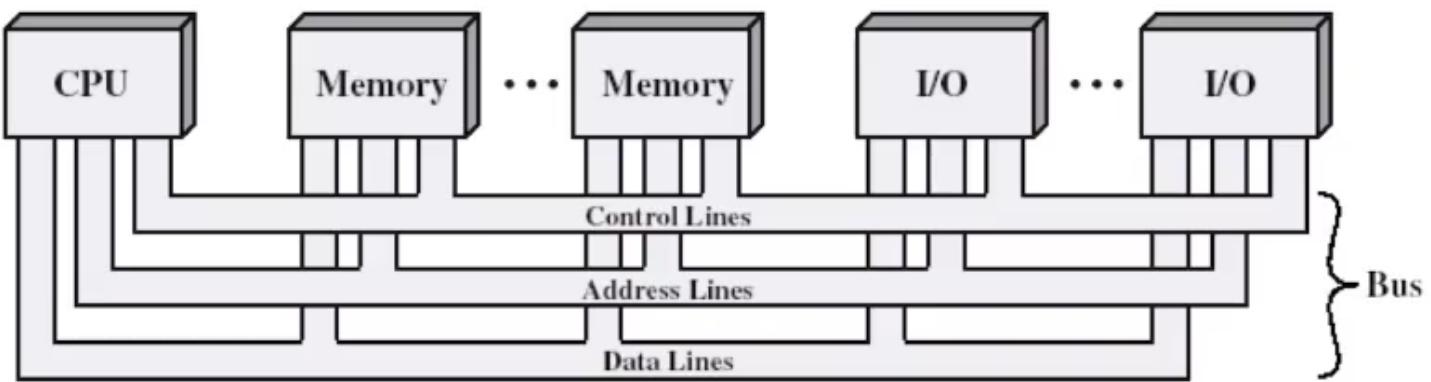


I BUS

I bus sono dei canali di trasporto che portano delle informazioni da un componente all'altro (non da una memoria all'altra, sono cose diverse).

Ci sono molteplici bus. Variano sia in architettura, in sistemi moderni o tradizionali... Noi vedremo un'astrazione di bus, non vedremo una cosa molto moderna.

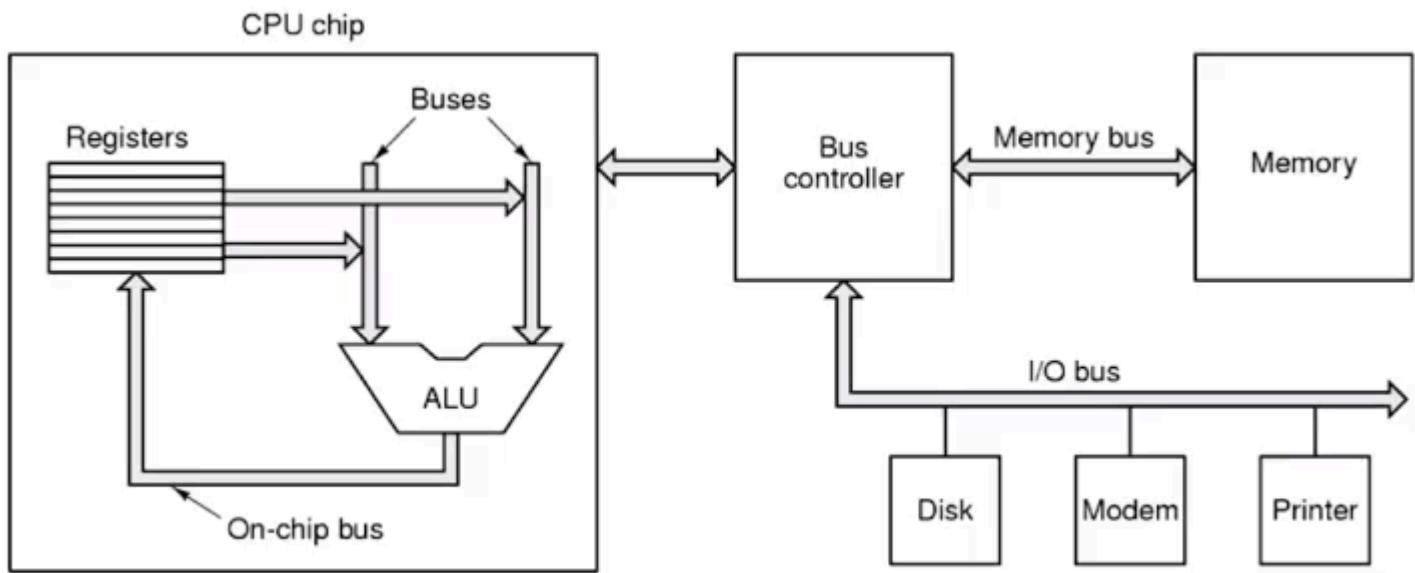
Di seguito un esempio di utilizzo di bus in una architettura di Von Neumann :



Note

Di cavi bus ne esistono tantissimi. Molte cose possono essere definite bus. Ad esempio troviamo : bus interni al processore, bus esterni che collegano vari componenti (PCIe), bus esterni al calcolatore (cavi USB, cavi ethernet)... Banalmente, anche il WIFI, e quindi l'aria, può essere visto come un bus di comunicazione

Un altro esempio di quanto detto sopra nella nota è questa immagine che chiarisce meglio la struttura :



Tipi di bus

I bus possono essere linee di :

- Dati : il numero di linee (larghezza del data bus) determina il numero di bit che possono essere trasmessi alla volta, ha un impatto sulle prestazioni del sistema.
- Indirizzo : permettono di individuare la sorgente/destinazione dei dati trasmessi sul data bus.
- Controllo : controllano l'accesso e l'utilizzo delle linee di dati e di indirizzo.

Dispositivi attivi e passivi

I dispositivi collegati ad un bus si dividono in :

- Attivi, possono decidere di iniziare un trasferimento, in genere sono collegati al bus per mezzo di un particolare chip detto bus driver
- Passivi, rimangono in attesa di richieste, in genere sono collegati per mezzo di un chip detto bus receiver
I dispositivi che si comportano sia come attivo che come passivo (es la CPU) sono collegati attraverso un chip combinato, il bus transceiver

Progettazione dei bus

I principali problemi nella progettazione di un bus riguardano :

- Larghezza del bus (numero di linee)
- Arbitraggio, come scegliere tra due dispositivi che vogliono diventare contemporaneamente arbitri dello stesso bus
- Funzionamento del bus, come avviene il trasferimento dei bit

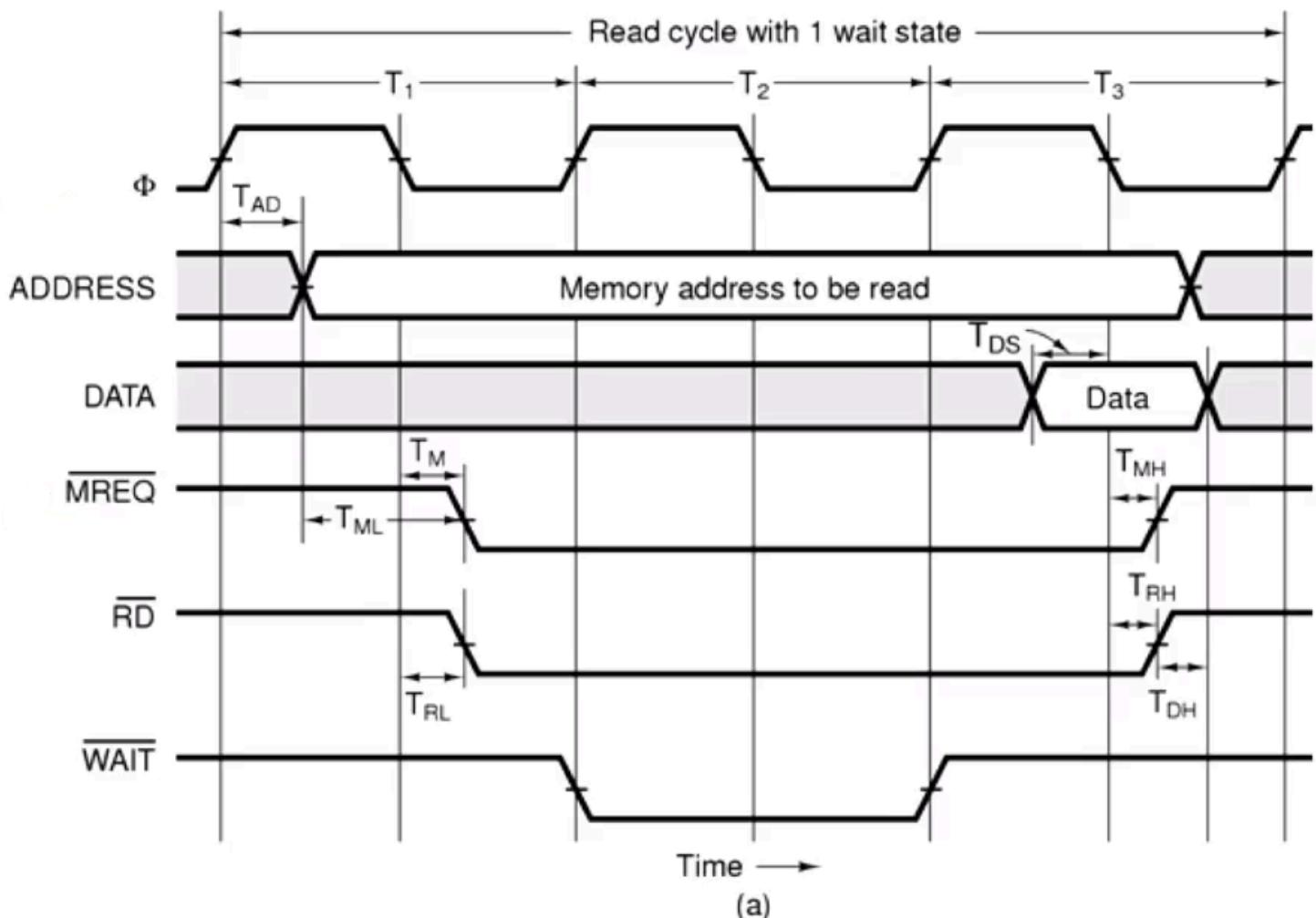
Categorie dei bus

I bus si possono dividere in due categorie :

- Bus sincroni, usano una linea di clock comune (pilotata da un oscillatore), con tutte le attività che avvengono in multipli interi del ciclo di clock. Esempi classici includono ISA, PCI e DDR. Le frequenze possono variare da pochi MHz (ISA) a diversi GHz (DDR5, PCIe).
- Bus asincroni, non usano un clock globale. Ogni transazione può richiedere tempi diversi, sincronizzandosi tramite segnali di handshaking.

Bus sincrono

Vediamo un esempio di bus sincrono :



Prendiamo il caso in cui la CPU (master) deve leggere qualcosa dalla memoria (slave). Chiariamo delle cose (alcune convenzioni):

- I fronti del clock sono obliqui per rispecchiare un po' di più la realtà, ma a noi questa cosa non interessa.
- Possiamo notare chiaramente la differenza tra alcune fasce.
 - Fasce "ampie" : queste fasce (come address e data) sono formate da più bit. In queste fasce, quando il segnale è colorato di grigio, vuol dire che il segnale non è stabile in quel tratto, non c'è il valore che mi serve in quel momento. Quando è bianco, vuol dire che il valore è stabile.
 - Fasce "stette" : queste fasce (come MREQ, RD, WAIT) sono formate da solo 1 bit. Queste fasce possono presentare alcune righe sopra. Quando una riga è presente sopra un segnale, vuol dire che viene asserito quando il valore vale 0 e non è asserito quando il valore vale 1. Ad esempio, prendiamo RD. Quando RD (read) vale 1, noi NON vogliamo leggere, mentre quando vale 0, noi vogliamo leggere.

Specificato ciò, vediamo cosa fa il processore per leggere un valore dalla memoria :

- Il processore mette l'indirizzo sul bus degli indirizzi.
- Il processore mette a 0 (asserisce) il valore di MREQ (memory request) e RD (read) per dire che vuole accedere alla memoria e che vuole leggere.
- La memoria si accorge che MREQ e RD sono stati asseriti (la memoria non presta caso al cambiamento dell'indirizzo nel bus apposito, potrebbero esserci diverse cose che fanno cambiare quell'indirizzo, non per forza un accesso alla memoria).
- Adesso la memoria asserisce il valore di WAIT, per dire al processore che deve aspettare il dato stabile.

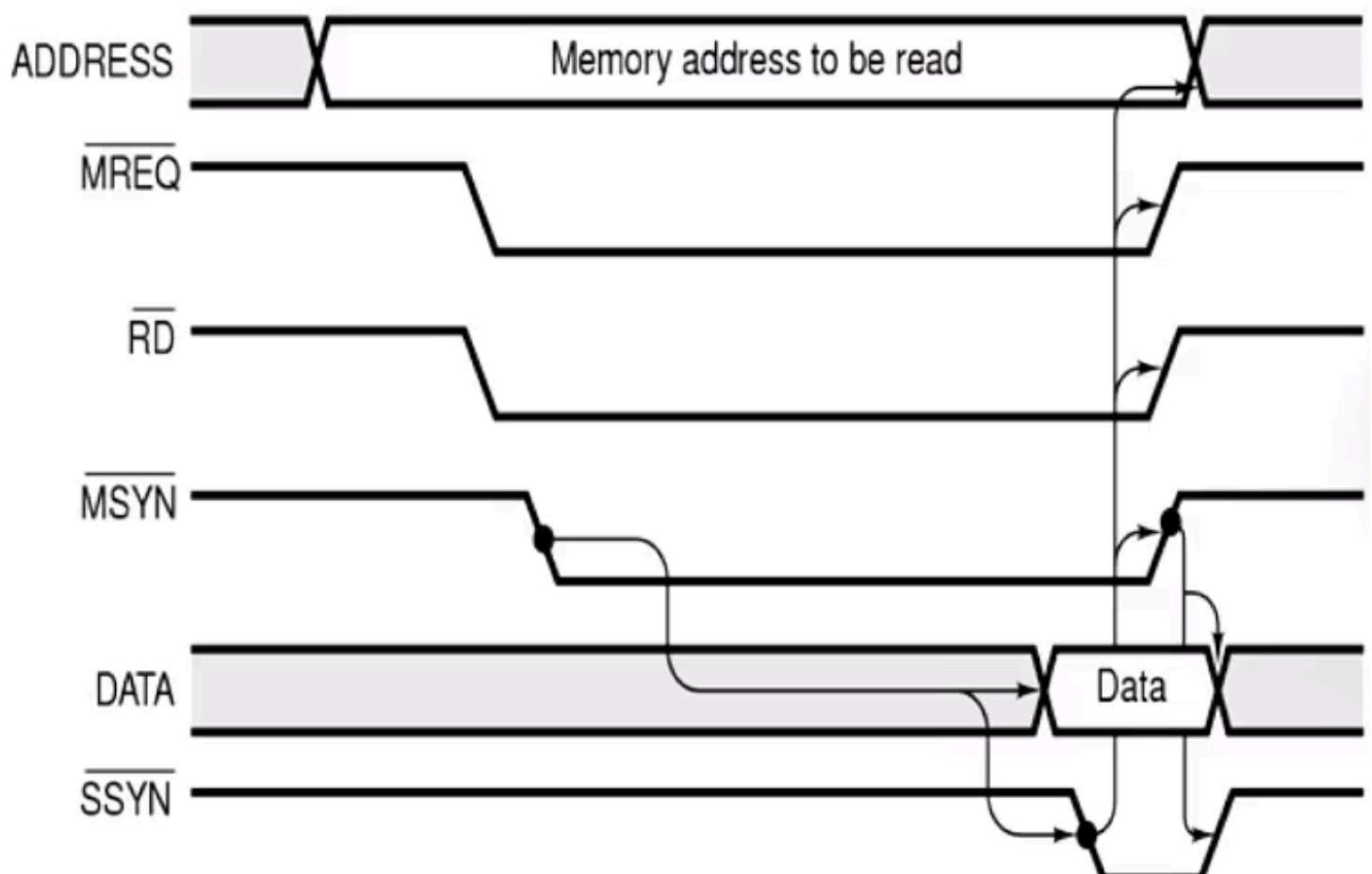
5. La memoria legge il dato richiesto all'indirizzo richiesto e lo carica sul bus "DATA" e mette WAIT ad 1 (per dire che è disponibile il dato).
6. Il processore a questo punto legge il dato e mette ad 1 sia MREQ sia RD (per dire alla memoria che non gli serve più).

Il processo è finito, ma c'è una cosa da specificare. Possiamo notare che il segnale WAIT è messo ad 1, ma in realtà in quel momento il segnale nel bus DATA non è ancora stabile, perché? Perché il processore scrive il dato richiesto in un registro, ovvero in dei flip flop, i quali scrivono solo su un fronte. Allora, la memoria notifica al processore in un punto, ma poi la memoria è sicura che il dato sarà disponibile sul prossimo fronte (in questo caso un fronte di discesa).

Il segnale di WAIT non è essenziale (per come abbiamo strutturato noi ora questa parte, sì), ma se noi conosciamo le specifiche della memoria e sappiamo, ad esempio, che la memoria legge in 3 cicli di clock, il segnale non serve.

Bus asincrono

A differenza del bus sincrono, non abbiamo un clock che permette di scandire il tempo.



Prendiamo, di nuovo, il caso in cui la CPU (master) deve leggere qualcosa dalla memoria (slave) :

1. Il processore rende stabile il segnale nel bus degli indirizzi.
2. Il processore asserisce (mette a 0) MREQ (memory request) e RD (read).
3. Il processore asserisce MSYN (master synchronization), ovvero, il processore dice che è pronto a ricevere il dato.
4. La memoria nota i cambiamenti di MREQ, RD e MSYN (e sa che, implicitamente, il processore si impegnerà a non cambiare il valore nel bus degli indirizzi).
5. La memoria legge l'indirizzo e mette nel bus dei DATI il dato letto stabile.

6. La memoria asserisce SSYN (slave synchronization) per far capire al master (il processore) che il dato è pronto.
7. Una volta che il processore ha letto il dato, impone a 1 MSYN (e gli altri due segnali) e la memoria capisce che il dato è stato letto.
8. La memoria toglie il SSYN.

Note

Quando ADDRESS e DATA si "chiudono", non vuol dire che non diventano stabile, ma che possono variare senza influire la lettura/scrittura.

Pro e contro dei bus sincroni/asincroni

Sincrono

Pro

- Realizzazione device semplice.
- Se la durata di un'operazione è fissa, non occorre una linea di wait.

Contro

- Durata di un'operazione di comunicazione deve necessariamente avere una durata pari ad un numero intero di cicli.

Asincrono

Pro

- Flessibilità : la durata di un'operazione è determinata unicamente dalla velocità della coppia di device

Contro

- Per completare un'operazione di comunicazione sono sempre necessarie 4 azioni
- Occorre inserire nei device i circuiti necessari a rispondere opportunamente al protocollo

Arbitraggio del bus

Lo stesso bus può essere richiesto da più dispositivi contemporaneamente. Per risolvere questo problema è necessario implementare un meccanismo di arbitraggio del bus, per evitare ambiguità delle informazioni immesse sul bus stesso. Sono possibili due strade :

- arbitraggio centralizzato
- arbitraggio decentralizzato

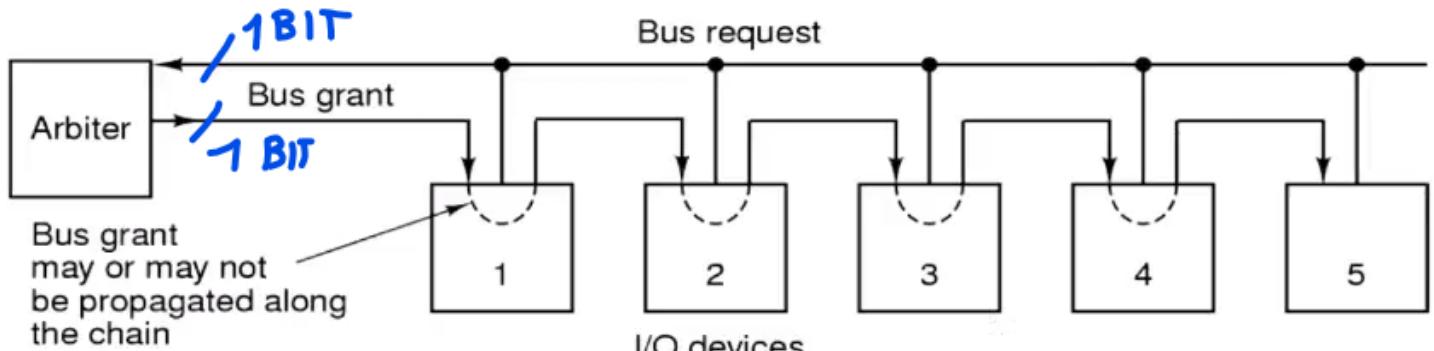
Meccanismo del grant

Mecccanismo per cui ho tanti dispositivi e io assegno il grant ad uno di questi. Quel dispositivo ora sa che può utilizzare quel bus. Il grant può essere assegnato in :

- grant esplicito, qualcuno assegna esplicitamente il grant.
- grant implicito, in base al contesto, un dispositivo si assegna autonomamente il grant.

Arbitraggio centralizzato

In questo modello, quando l'arbitro "nota" una richiesta, attiva la linea di grant del bus.



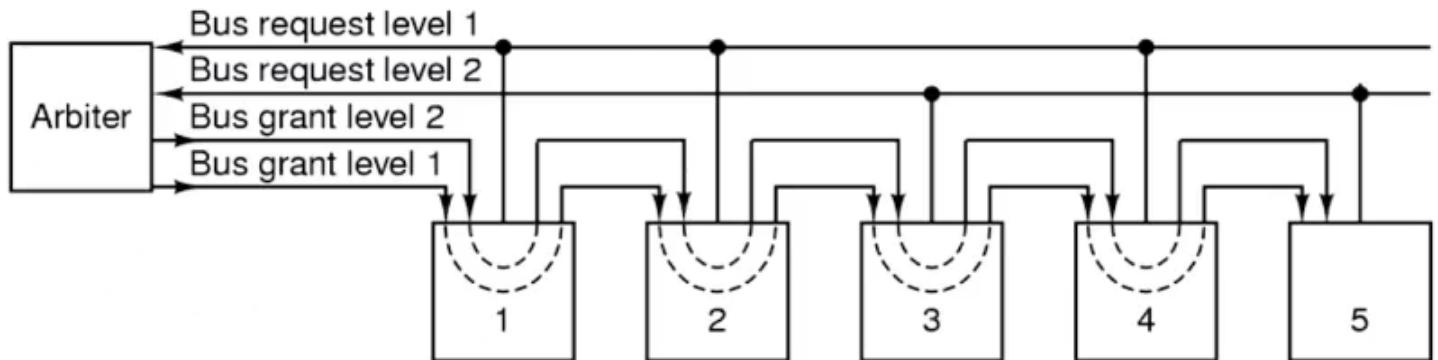
Il segnale di grant viene inviato seguendo la tecnica del "Daisy chaining". Ovvero, grant viene passato da ogni dispositivo finché un dispositivo non accetta l'assegnamento. Anche se il dispositivo 2 dovesse chiedere il grant, il dispositivo 1 potrebbe prenderlo quando passa da lui. Si dice che il dispositivo più vicino "vince".

Starvation

Questo modello presenta un grosso problema, ovvero che i dispositivi alla fine, più lontani, potrebbero non arrivare mai a ricevere il grant. Per risolvere questo problema si possono integrare più livelli di priorità.

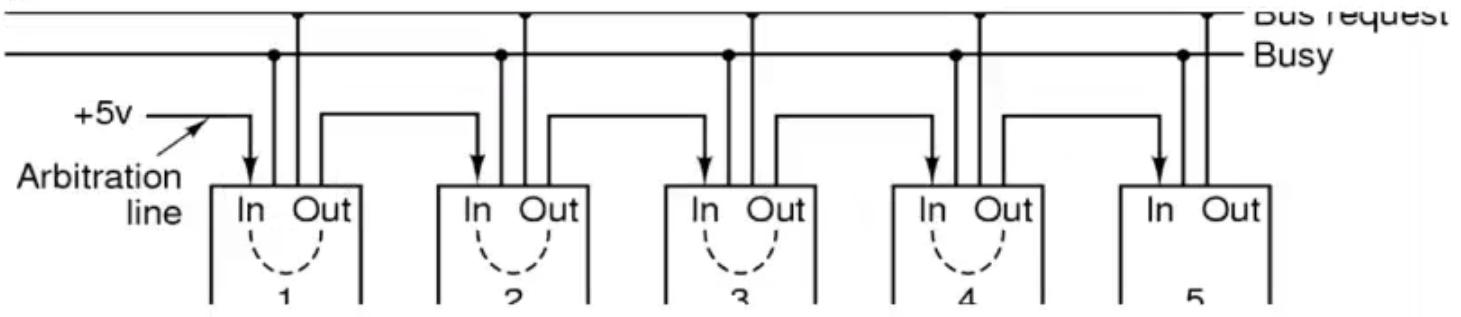
Priorità

Si possono integrare due, o più, livelli di priorità. In questo caso l'assegnamento segue lo stesso meccanismo del daisy chaining, ma i dispositivi priorità più alta hanno precedenza nell'assegnamento del grant da parte dell'arbitro.



Arbitraggio decentralizzato

In questo modello non c'è un arbitro che gestisce il bus, ma si permette ai dispositivi stessi di gestirlo.



Il dispositivo che vuole usare il bus ferma la propagazione di "arbitration line", abilità il segnale di "busy" (il quale notifica a tutti i dispositivi che il bus è occupato) e poi il bus viene usato. Una volta finito l'uso del bus, si toglie il segnale di "busy" e si lascia passare il segnale di "arbitration line". Questo sistema è più economico e veloce rispetto a "daisy chaining" centralizzato, però è poco flessibile (ma si risparmia il costo dell'arbitro).

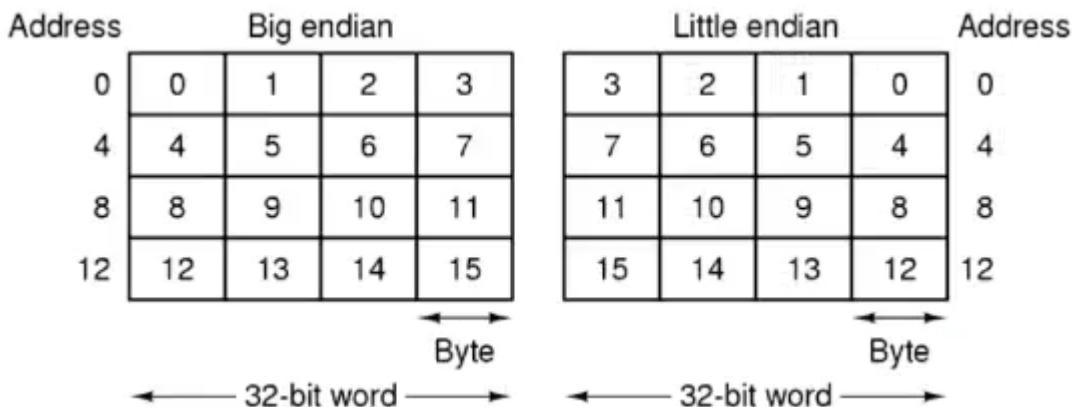
Memoria

Alcuni fatti per ripassare la memoria :

- La memoria è un insieme di celle aventi tutte la stessa dimensione
- Ogni cella di memoria è identificata tramite un indirizzo
- Se un calcolatore ha n celle, allora gli indirizzi vanno da 0 a $n - 1$, indipendentemente dalla dimensione della cella
- Il numero di bit nell'indirizzo determina il numero massimo di celle indirizzabili
- La cella è la minima unità indirizzabile (possiamo caricare gli 8 bit di una cella e poi vederne, ad esempio, 4, ma ne dobbiamo sempre caricare 8)
- In RISC-V una cella di memoria è grande 8 bit e ci sono 2^{32} celle.

Ordinamento dei byte

Abbiamo due modi di ordinare i byte :



- Big endian, la numerazione va da sinistra verso destra. Big end first (SPARC, mainframe IBM).
- Little endian, la numerazione va da destra verso sinistra. Little end first (Intel, RISC-V).

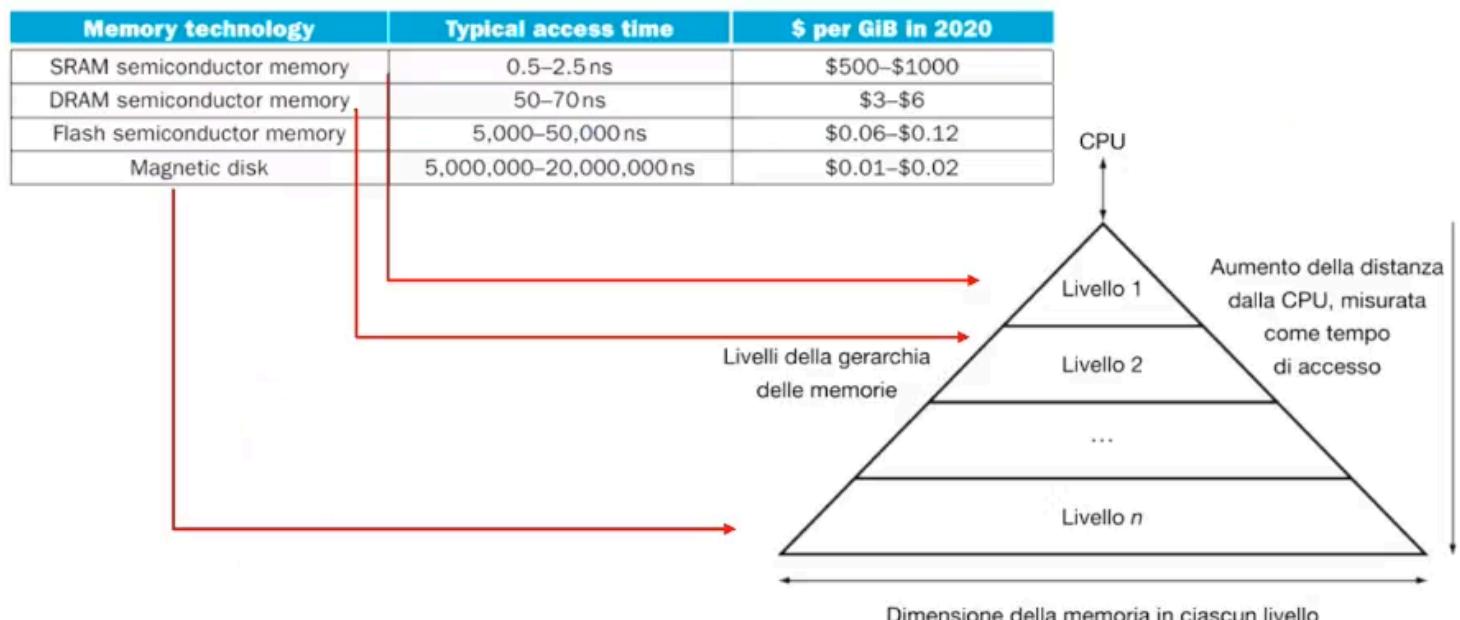
CPU e memoria

Le CPU sono più veloci delle memorie, questo può provocare del bottleneck (un rallentamento dell'interno processo per colpa di un componente). Infatti, quando la CPU effettua una richiesta alla

memoria, essa non ottiene la parola desiderata se non dopo molti cicli di CPU. Come si può risolvere questo problema? Cosa dovremmo fare, avere una memoria più piccola e molto più veloce (e molto più costosa) o una grande quantità di memoria, ma non abbastanza veloce (e relativamente economica)? Beh, si può fare una via di mezzo, ovvero, si può implementare una piccola quantità di **memoria cache** (attualmente lo standard sono pochi KB/MB in base al livello) e una media/grande quantità di memoria RAM.

#tyyyttt ## Gerarchia della memoria

Ecco uno schema della gerarchia delle memorie secondo la capacità di memorizzazione :



Come possiamo vedere c'è un collegamento diretto tra il costo delle memorie e la velocità di accesso. Per quanto un disco magnetico costi davvero poco, è impensabile di usarla memoria principale dinamica.

Memoria cache

La memoria cache è una memoria che si interpone tra la CPU e la memoria principale (RAM). L'uso della memoria cache è basata sul principio di località del codice (i programmi non accedono alla memoria a caso).

Se la CPU vuole leggere una word dalla memoria che non è salvata nella cache (nella cache possono essere salvate sia istruzioni che dati), questa word viene letta dalla memoria assieme a molte altre word (viene letto un blocco di memoria) e vengono salvate nella cache. Poi la cache restituisce la word richiesta alla CPU. Questa cosa viene fatta per i due seguenti fattori :

- Località temporale

Principio secondo il quale se si accede a una determinata locazione di memoria, è molto probabile allora che vi si acceda di nuovo dopo poco tempo. Un esempio banale al quale pensare è un ciclo (sia for che while), quando finisce un'iterazione del ciclo, si ritorna indietro alle stesse istruzioni.

- Località spaziale

Principio secondo il quale se si accede a una determinata locazione di memoria, è molto probabile che si acceda alle locazioni vicine a essa dopo poco tempo. Basta pensare ad un array contiguo.

Se accedo alla posizione i, è facile che io poi voglia avere i+1. Invece, per quanto riguarda le istruzioni, basta pensare a qualsiasi frammento di codice (dove non ci sono dei salti), noi vogliamo fare prima un'istruzione, poi un'altra e poi un'altra ancora tutte vicine tra di loro.

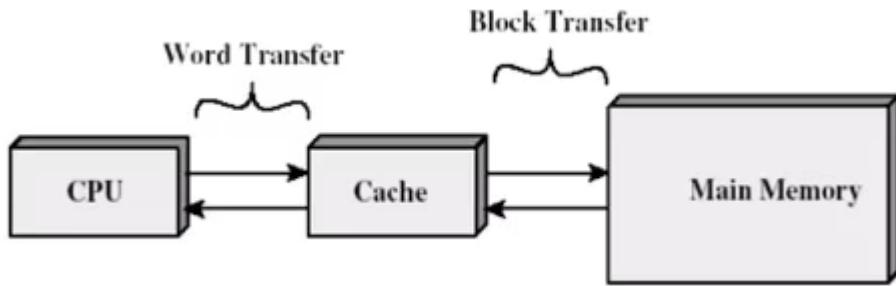
Blocchi di informazioni

La memoria principale consiste di 2^n byte indirizzabili. Logicamente possiamo pensare che questa memoria è divisa in blocchi di k byte ($M = 2^n/k$ blocchi), La memoria cache consiste in una quantità C di linee di k byte (blocchi).

Note

C (il numero di blocchi in cui è divisa la cache) è sempre minore a M (il numero di blocchi un cui è divisa la RAM). Questo è ovvio, perché altrimenti vorrebbe dire avere una cache che è grande quanto (o più) la RAM.

Questa divisione in blocchi è usata per adempiere al principio di località. Come vediamo dall'immagine sotto, tra la RAM e la cache si passano interni blocchi e poi dalla cache alla CPU si passano semplicemente le word richieste :



Un blocco viene caricato sulla cache quando la CPU richiede una word e la cache si accorge di non avere quella word, allora, viene caricato l'intero blocco dove è contenuta quella word (un blocco ovviamente contiene più di una word, ne contiene diverse).

Miss e hit della cache

Ci sono due possibilità per le word richiesta dalla CPU :

- hit : la word richiesta si trova nella cache, quindi il tempo è ottimo.
- miss : la word richiesta non si trova nella cache, pessimo per il tempo.

Si deve cercare di alzare al massimo le hit e abbassare le miss.

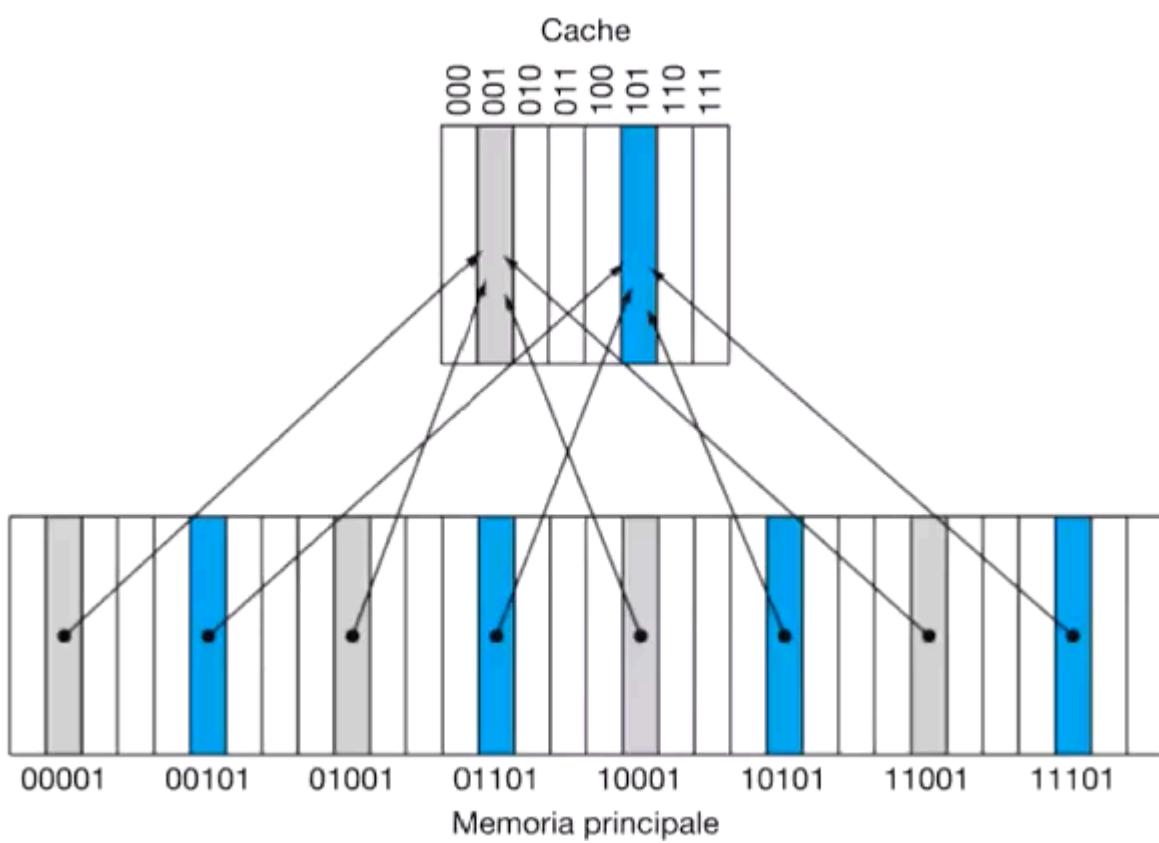
Problemi relativi alla progettazione della cache

Uno dei problemi da considerare è la dimensione della linea di cache, avere tante linee di pochi byte o poche linee di molti byte? Avere tanti blocchi, ma troppo piccoli fa generare tante miss. In modo contrario avere pochi blocchi troppo grandi, ci saranno tante sovrascritture dei blocchi.

Per questo ci sono diversi modi di mappare la cache :

Mappatura diretta (studiare)

Una cache in cui ad ogni locazione della memoria principale (ram) corrisponde una (e una sola) locazione della cache.



Si calcola con : (indirizzo del blocco) modulo (numero di blocchi nella cache)

☰ Example

Prendiamo come esempio il blocco "10001" nella memoria principale. Per sapere in quale blocco della cache verrà salvato con la mappatura diretta facciamo :

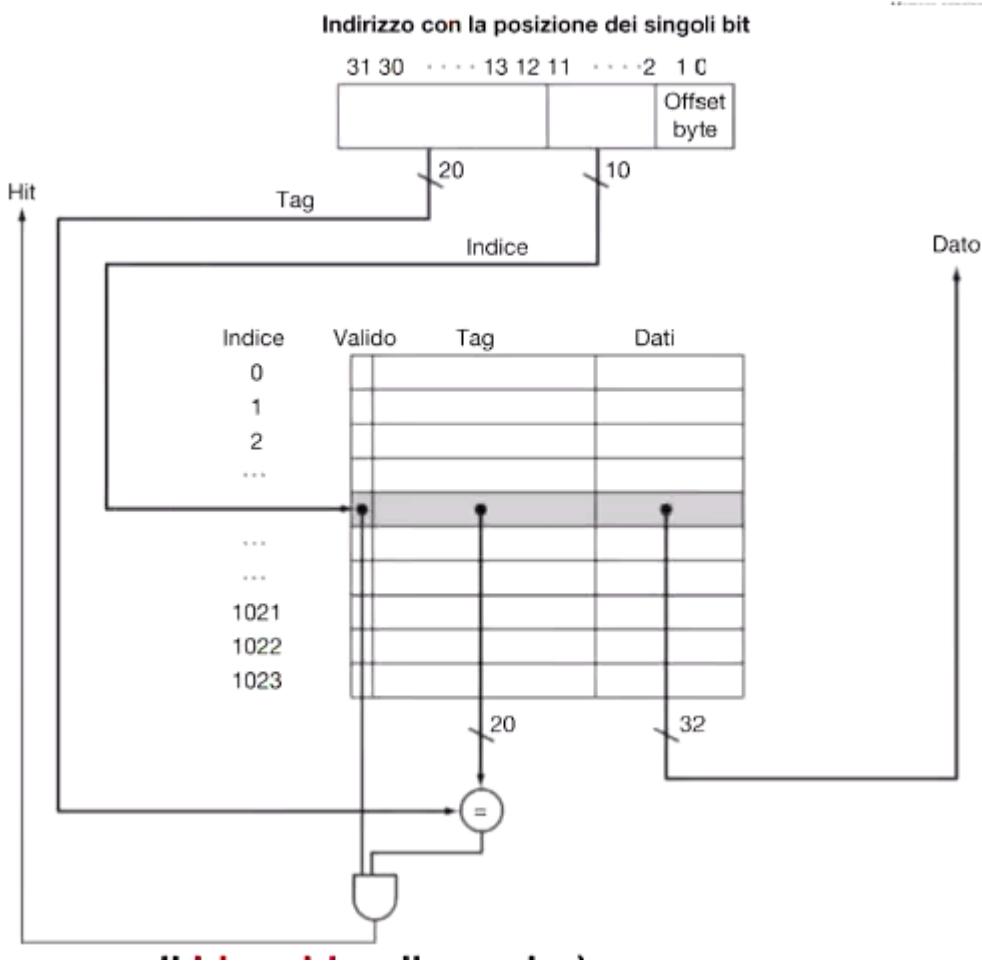
1. Da binario a decimale, $10001 \rightarrow 17$.
2. Faccio il modulo con il numero di blocchi della cache (in questo caso 8), $17 \% 8 = 1$. (Da questo esempio notiamo che anche 9, 25, 33... finisco nel blocco 001)
3. Trasformo 1 (risultato del modulo) in binario e ho l'indirizzo, $1 \rightarrow 001$.

Un altro modo per ottenere l'indirizzo in cui salvare della cache è quello di prendere le ultime n cifre dell'indirizzo della memoria principale (con n si intendo il numero di bit necessari a rappresentare gli indirizzi nella cache, in questo caso è 3) :

- Prendiamo gli ultimi 3 bit di $10001 = 001$, ecco che abbiamo lo stesso risultato (da questo capiamo anche che il calcolo è praticamente immediato dal punto di vista computazionale)

Come vediamo dall'immagine sopra, più blocchi della memoria principale vengono salvati sullo stesso blocco (NON contemporaneamente, uno sovrascrive l'altro), come si fa a capire allora quale blocco è stato sovrascritto? Si può capire a quale blocco della memoria principale corrisponde una locazione

della cache salvando i primi due bit del blocco che salviamo. Questi bit salvati si chiamano **tag**.



Cosa si salva nella cache?

Si salva :

- Bit di validità : ci dice se è salvato qualcosa in questo indirizzo è salvato qualcosa (1 bit).
- Tag : tag unico che permette di capire da quale blocco della memoria abbiamo preso i dati e salvati nella cache
- Dati : i dati salvati dalla memoria principale

Esempio (importante)

guarda slide/registrazione

Es :

- 8 blocchi
- 11111 (indirizzo) = 31
- $31 / 8 = 3$ resto 7
- Metto il dato al blocco 111 (7 in binario) e metto i tag a 11 (i primi due bit a sinistra dell'indirizzo).

Vantaggi

L'algoritmo è molto efficiente per quanto riguarda l'efficienza di calcolo. Il modulo e la ripresa di un dato è molto veloce.

Svantaggi

È possibile che capiti che vogliamo salvare un blocco nella cache in un posto in cui c'è già salvato qualcosa, quindi dobbiamo sovrascrivere, rischiando di ottenere tante miss (magari il dato tolto ci serviva per qualcosa).

Esempio (importante, viene facilmente chiesto agli esami)

Di seguito un esempio di occupazione della cache seguendo il modello di occupazione diretta. Lo leggiamo dall'alto verso il basso, pensando che il processore voglia una word dai blocchi che vediamo a sinistra :

Indirizzo decimale del dato nella memoria principale	Indirizzo binario del dato nella memoria principale	Hit o miss nell'accesso alla cache	Blocco della cache corrispondente (dove trovare o scrivere il dato)
22	10110 _{due}	Miss (5.9b)	(10110 _{due} mod 8) = 110 _{due}
26	11010 _{due}	Miss (5.9c)	(11010 _{due} mod 8) = 010 _{due}
22	10110 _{due}	Hit	(10110 _{due} mod 8) = 110 _{due}
26	11010 _{due}	Hit	(11010 _{due} mod 8) = 010 _{due}
16	10000 _{due}	Miss (5.9d)	(10000 _{due} mod 8) = 000 _{due}
3	00011 _{due}	Miss (5.9e)	(00011 _{due} mod 8) = 011 _{due}
16	10000 _{due}	Hit	(10000 _{due} mod 8) = 000 _{due}
18	10010 _{due}	Miss (5.9f)	(10010 _{due} mod 8) = 010 _{due}
16	10000 _{due}	Hit	(10000 _{due} mod 8) = 000 _{due}

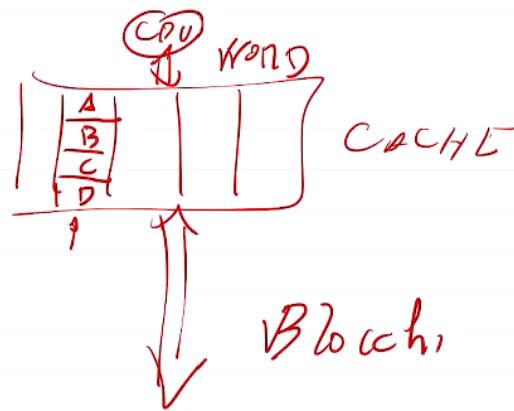
Come possiamo notare la prima è ovviamente una miss. La memoria cache è ancora vuota. Salviamo il valore nella cache. Dove lo salviamo? Facciamo $22 \% 8 = 6_{10} = 110_2$. Salviamo come tag "10". Andiamo avanti, di nuovo, controlliamo l'indirizzo. facciamo il modulo e otteniamo 010, c'è qualcosa in questo blocco della cache? No, salviamo il blocco all'indirizzo 010 e mettiamo come tag 11. Segniamo una miss

Ancora, facciamo il modulo, vediamo all'indirizzo "110" della cache, c'è qualcosa? Sì, il tag qual è? 10, allora è proprio quello che vogliamo, hit.

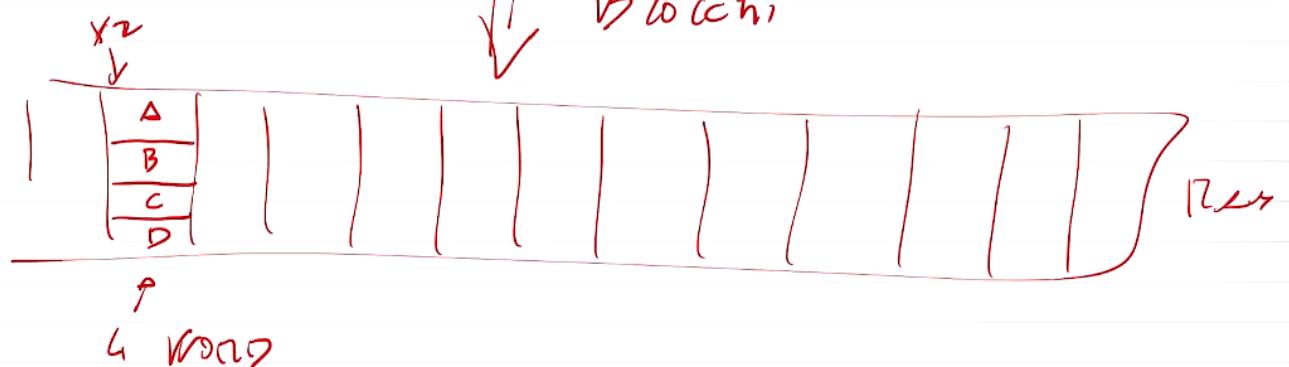
Procediamo così per tutto il resto dell'esempio. In caso il modulo di un indirizzo ci dia un risultato che già c'è nella cache, ma il tag è diverso, dobbiamo segnare una miss e sovrascrivere il contenuto della cache (guarda il penultimo punto).

Dimostrare che la cache velocizza l'esecuzione di un programma

LW X1, 0(X2)
 LW X1, 4(X2)
 LW X1, 8(X2)
 LW X1, 12(X2)



MISS \times
 HIT $\times \times \times$



(Esempio proposto dal professor Schifanella)

Mappatura diretta : vantaggi e svantaggi

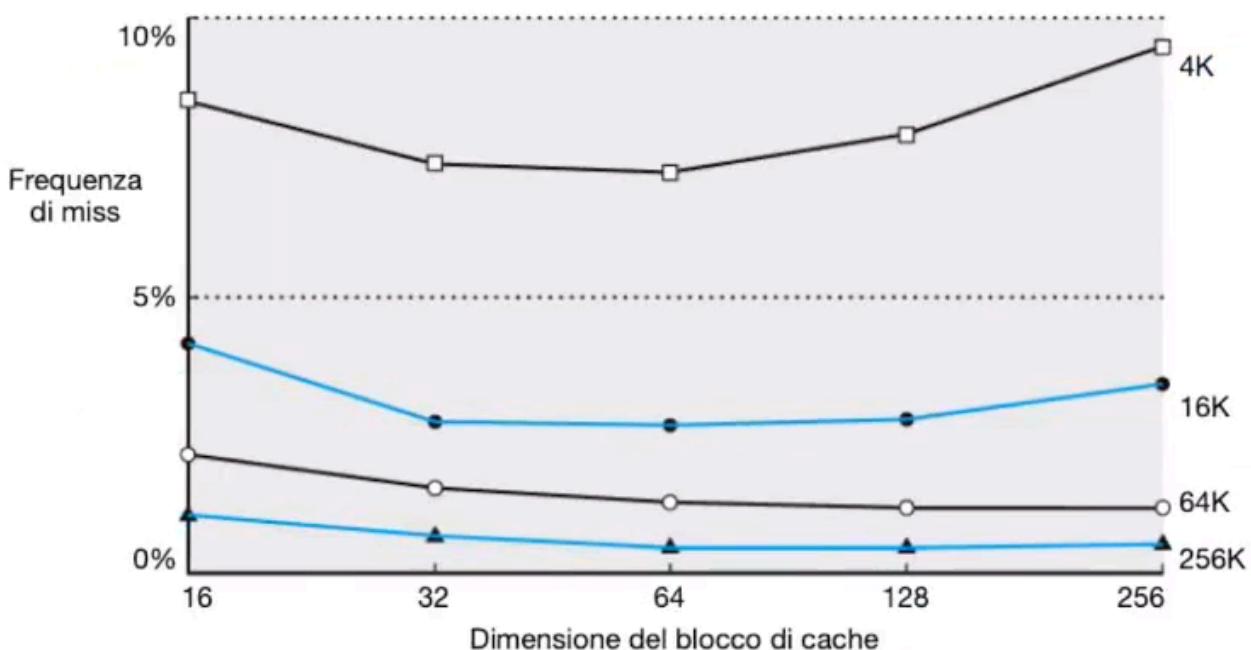
Vantaggi :

- Efficienza, è molto semplice passare dall'indirizzo del blocco dalla memoria all'indirizzo nella cache (basta considerare gli ultimi n bit e aggiungere il tag)

Svantaggi :

- Capita spesso che dei blocchi salvati sulla cache debbano essere sovrascritti. Questo è influito anche dalla grandezza dei blocchi.

Grafico per mostrare come aumentare la dimensione del blocco non è sempre la soluzione ottima :



Completamente associativa

Un altro metodo di mappatura dalla memoria cache. Questo metodo permette di scrivere nella cache in un punto qualsiasi.

C'è un controllo, la ricerca va effettuata su tutti gli elementi della cache. Questo influisce sia la ricerca che la scrittura di un blocco. Quando vogliamo scrivere dobbiamo cercare un blocco sulla cache libero oppure dobbiamo trovare un blocco da sovrascrivere. Ci sono varie criteri che si possono seguire per sovrascrivere un blocco, ovvero :

- Il primo blocco scritto
- L'ultimo blocco scritto
- Il blocco meno letto fra quelli salvati
- Il blocco nella cache con l'accesso più vecchio (LRU - Last Recently Used)

Come possiamo scegliere quale usare? Molto semplicemente basta fare dei test di benchmark per capire quello più performante.

Abbiamo ancora il problema che con questa implementazione dobbiamo controllare cella per cella, come si può risolvere? Possiamo risolvere questo problema parallelizzando la ricerca con dei **comparatori** (i quali fanno aumentare i costi, i consumi, il calore).

Cache set-associativa

Quale è meglio tra il metodo "completamente associativa" e "mappatura diretta"? Attualmente, quasi mai, nessuna delle due, viene usata la "Cache set-associativa". Questo metodo è una via di mezzo tra le prime due.

Funzionamento

Ci sono delle linee che contengono più blocchi (detto numero di vie). Individuata la linea, il blocco viene ricercato all'interno degli slot della linea. Di seguito uno schema di molteplici costruzioni di questo modello :

Cache set-associativa a una via (a mappatura diretta)

Blocco	Tag	Dato
0		
1		
2		
3		
4		
5		
6		
7		

Cache set-associativa a due vie

Linea	Tag	Dato	Tag	Dato
0				
1				
2				
3				

Cache set-associativa a quattro vie

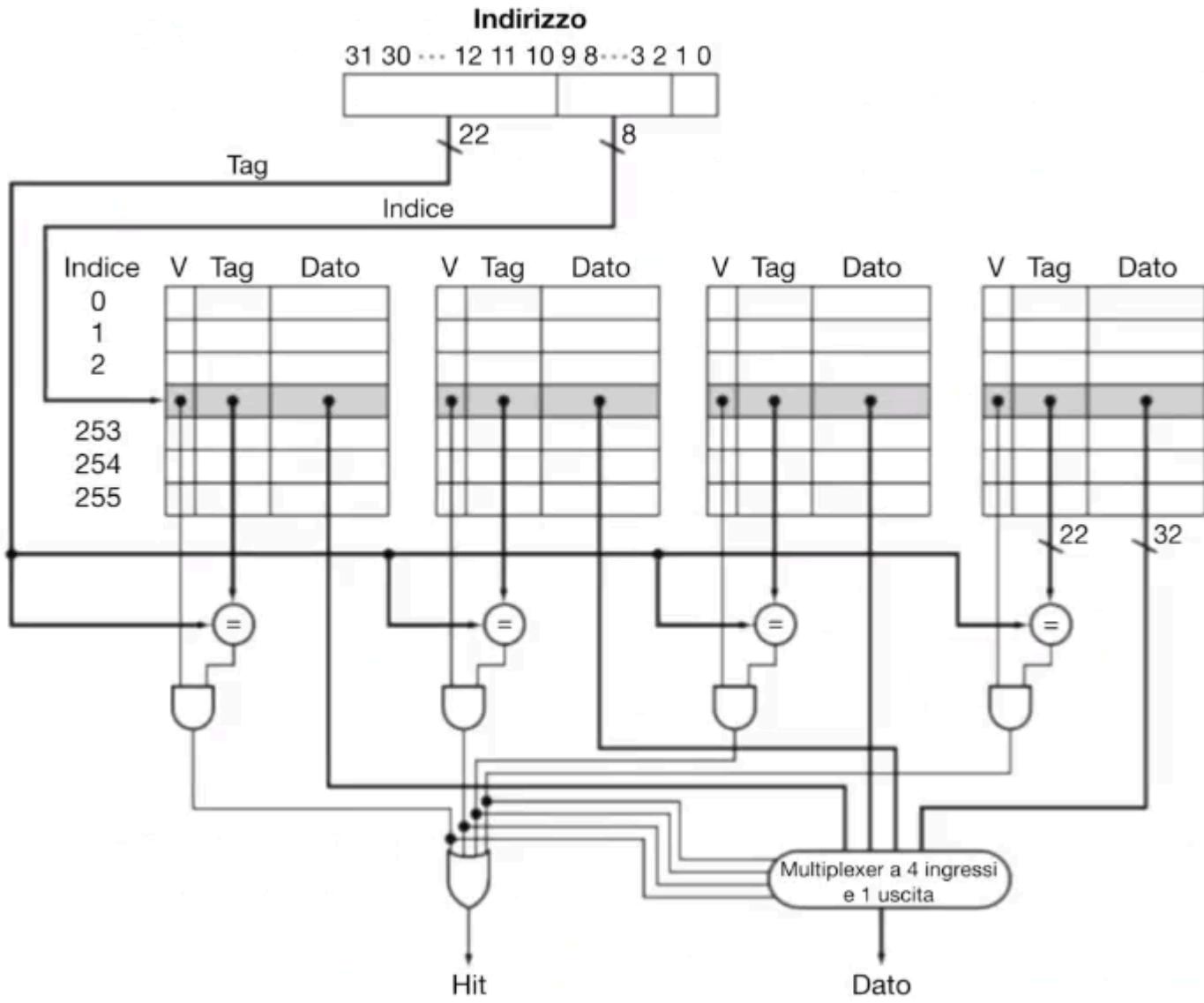
Linea	Tag	Dato	Tag	Dato	Tag	Dato	Tag	Dato
0								
1								

La linea che dobbiamo scegliere è scelta tramite la mappatura diretta. Trovata la via corretta, si ricerca il dato all'interno della via usando l'algoritmo completamente associativo (ricerca esaustiva).

Un esempio delle prestazioni :

Associatività	Frequenza di miss
1	10.3%
2	8.6%
3	8.3%
4	8.1%

L'implementazione della cache sopra è la seguente :



Gestione delle miss

La gestione delle miss e hit necessita di una modifica all'unità di controllo del processore (approfondimento disponibile sul libro di testo 5.9, non richiesto).

Gestione della scrittura

Fino ad adesso abbiamo sempre parlato della lettura della RAM/cache e di come ottimizzarla, ora ci poniamo il problema della scrittura. La scrittura però porta dei nuovi problemi. Quando scriviamo (e il dato è in cache), dove scriviamo? in cache e poi riportiamo il dato nella RAM? non lo riportiamo? Che strategie possiamo usare?

Write-through

Ad ogni scrittura in cache, si modifica il valore anche in memoria. Se il dato da scrivere non è presente in cache, viene prima caricato, poi modificato in cache e in memoria.

Le prestazioni non sono molto buone (stiamo usando un buffer di scrittura).

Write-back

Il dato viene scritto solo in cache. La scrittura al livello inferiore avviene solo quando il blocco nella cache deve essere rimpiazzato. È indubbiamente più veloce, perché modificare la cache è molto più veloce e poi modifichiamo una sola volta la ram, però è anche più complesso da implementare.

Input e Output

La CPU comunica con i device di I/O tramite i controllori, che hanno il compito di trasformare i comandi della CPU in segnali elettrici per le periferiche e i segnali delle periferiche in dati per la CPU.

Ogni controllore ha al suo interno dei registri identificati da un indirizzo, che può :

- Appartenere allo stesso insieme di indirizzi di memoria (memory mapped I/O, dove l'OS decide l'indirizzo associato ad un controllore)
 - Essere un indirizzo dedicato (isolated I/O).
- La comunicazione con i controllori è simile agli accessi in memoria.

Registri dei controllori

Cosa possono contenere? Ecco :

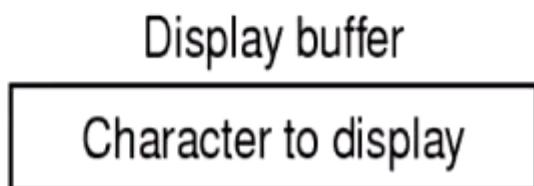
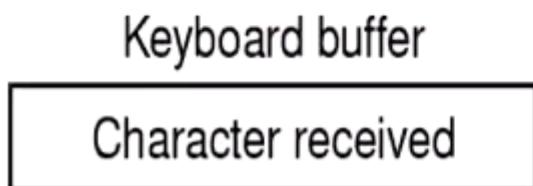
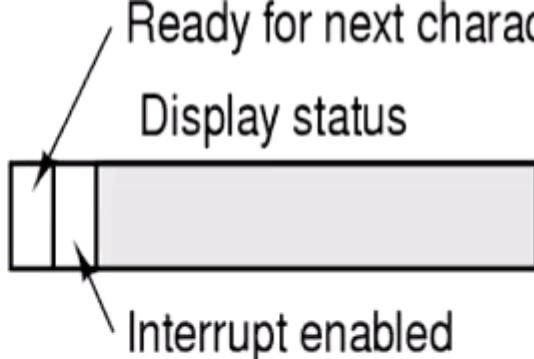
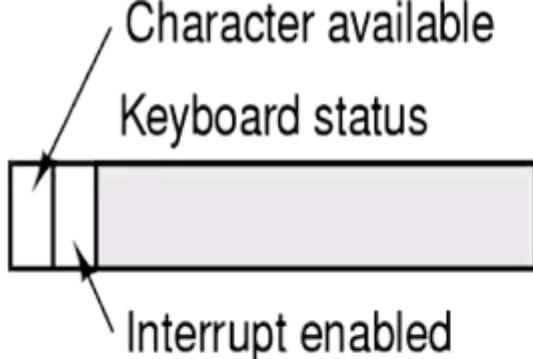
- dati (di ingresso e uscita). Es, per una stampante possono essere i dati da stampare.
- comandi (dalla CPU alla periferica). Es, i comandi di stampa.
- informazioni sullo stato della periferica (dalla periferica alla CPU). Es, informazioni sulla stato della stampante o sull'avanzamento dei lavori.

Modalità di input/output

I/O programmato con Busy waiting

La CPU controlla lo stato di avanzamento del I/O e lo stato della periferica ispezionando in un ciclo il bit del registro di stato del controllore READY fino a che questo non segnala di essere pronto per un nuovo comando di I/O.

Supponiamo di avere una tastiera. La CPU dovrà usare il tasto premuto sulla tastiera. Ogni tasto è codificato tramite un codice che sarà contenuto all'interno di un registro del controllore della tastiera e quando viene schiacciato deve arrivare alla CPU.



Come fa quindi la tastiera a sapere che è stato premuto un tasto? Un modo è proprio usando la tecnica di busy waiting, cioè la CPU chiede alla tastiera "è stato premuto un tasto?" e la tastiera risponde di conseguenza. Come avviene la richiesta? Tramite un bit di controllo, che nel caso di sopra è segnato in "keyboard status".

Questo controllo la CPU lo fa in modo ciclico. Viene detto "polling" (verifica ciclica appunto dei dispositivi di I/O tramite i bit di controllo).

L'esempio della tastiera vale per la lettura. Per la scrittura invece possiamo fare l'esempio di voler stampare qualcosa a video. Con lo stesso modo, la CPU chiede al display qualcosa tipo "sei pronto a stampare la prossima informazione" e poi il display risponde. Se dice sì, ottimo, altrimenti la CPU (come prima) torna dopo del tempo e gli chiede di nuovo.

Questo metodo fa sprecare delle risorse alla CPU, non è ottimale (ma può essere comunque usato in alcuni casi in cui questo schema risulta comodo).

Interrupt

Il dispositivo quando ha finito genera un segnale che avverte la CPU (tramite una linea di bus direttamente collegata alla CPU) di aver completato il proprio lavoro. La CPU può abilitare l'interrupt a 1 un opportuno bit.

Per poter sapere chi ha generato l'interrupt abbiamo bisogno di un "vettore di interrupt", il quale ci fa sapere chi ha generato un interrupt.

Funzionamento

L'interrupt interrompe il programma in atto e trasferisce il controllo ad un gestore di interrupt che eseguirà le azioni appropriate. Una volta terminata la gestione dell'interrupt, il controllo torna al programma interrotto.

Trasparenza

Gli interrupt sono asincroni rispetto al programma e devono essere gestiti in modo trasparente, ovvero, lo stato dell'esecuzione dopo la gestione dell'interrupt deve tornare esattamente come era prima dell'interrupt stesso. Quindi, lo stato dei vari registri deve tornare esattamente come prima (o salvando in memoria tutti i registri in memoria e poi ripristinarli oppure usando altri registri, detti "shadow registers" che servono per salvare una copia dei registri "primari" per poi ripristinarli).

DMA (direct memory access)

I/O programmato, ma svolto da un'apposito componente con accesso diretto al bus. Il DMA è impostato direttamente dal software o il sistema operativo inizializzando opportuni registri. La CPU e il DMA si contendono l'uso del bus (e questa è l'unica cosa di cui si deve fare attenzione durante l'uso di questo metodo).

Trap (o eccezioni/interrupt sincroni)

Sono chiamate a procedure automatiche che possono essere causate dal verificarsi di qualche condizione eccezionale, oppure da istruzioni apposite per richiedere servizi di base del sistema operativo. Queste trap sono generate dal programma.

☰ Example

Alcuni esempi che POSSONO generare una trap (o eccezioni/interrupt sincroni) :

- Tentativo di accesso ad un area di memoria non consentita
- Divisione per 0
- Un overflow

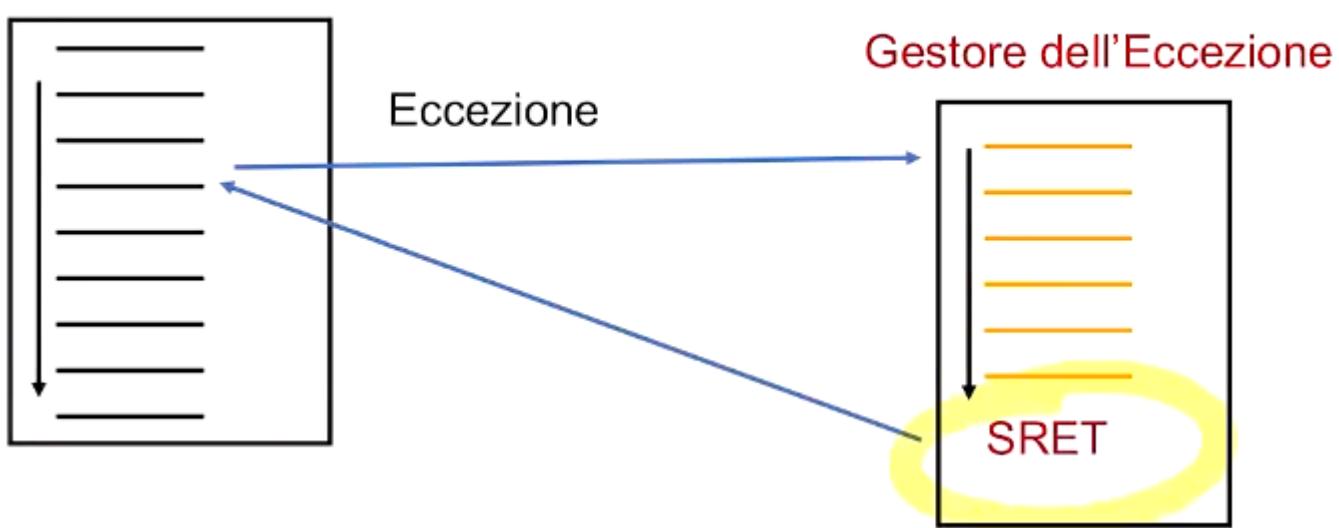
✍ Note

Se si usa il termine "interrupt sincrono" (anziché gli altri due), si intendo che l'interrupt va gestita subito, in modo sincrono al programma.

Gestione della trap

Il sistema gestisce l'eccezione eseguendo il "gestore dell'eccezione".

Applicazione



SRET = Supervisor return. È un'istruzione speciale a conclusione del gestore.

Tipi di eccezioni

Troviamo due tipi di eccezioni :

- utente (user mode), le applicazioni vengono eseguite in modalità user, senza poter fare accesso alle risorse privilegiate hardware del calcolatore
- supervisor (kernel mode), durante l'eccezione, il calcolatore esegue codice del sistema operativo
Andiamo in modalità utente quando, ad esempio, chiudiamo un programma. Invece, in modalità supervisor quando c'è un errore (es. overflow), una "Environment Call" (o "trap" o "system call") o ancora "Environment break" (debug).

Come si comporta il calcolatore

- Il programma in esecuzione deve essere sospeso e poi riattivato.
 - Il calcolatore salva il punto del codice (PC) in cui si è verificate l'eccezione (1).
- Il controllo passa quindi al "gestore dell'eccezione" (Exception Handler)
 - il gestore dell'eccezione deve rimediare alla situazione
 - A seconda del tipo di eccezione, il gestore esegue azioni diverse
 - Il calcolatore identifica e salva la causa dell'eccezione (2)
- **RISC-V : registri speciali SEPC e SCAUSE**
 - (1) SEPC, l'indirizzo del codice in cui si verifica l'eccezione
 - (2) SCAUSE, la cause dell'eccezione

⚠ Warning

NOTE :

- Il gestore dell'eccezione deve evitare di modificare lo stato dell'applicazione (i registri...)
- Il gestore dell'eccezione deve essere eseguito in modalità protetta (kernel mode)

Implementazione gestione delle eccezioni

In RISC-V troviamo 2 modi di gestione :

- Salto diretto all'indirizzo della routine di gestione
 - Viene generata un'eccezione
 - Esiste un registro che contiene l'indirizzo del gestore dell'eccezioni
 - Nella routine di gestione c'è un codice che va a trovare la causa dell'eccezione (che viene salvata in SCAUSE)
 - Il sistema operativo decide dove in memoria deve essere il gestore delle eccezioni.
 - L'indirizzo che contiene l'indirizzo di base si chiama "SVTEC"
- Vettore di interruzione
 - SVTEC contiene l'indirizzo di base
 - In memoria abbiamo un vettore delle interruzioni, cioè in memoria c'è un array che contiene gli indirizzi di tutte le routine di gestione di tutte le eccezioni.
 - Ogni causa ha un codice, si moltiplica per 4 (perché dipende dalla dimensione della causa e degli indirizzi) e lo si somma all'indirizzo di base.

- Abbiamo l'indirizzo della routine di gestione della causa del nostro errore che punta alla memoria dove ci sono effettivamente le istruzioni.

Domande poste durante la lezione (utili per esame)

- Strumenti per trasformare un programma da un linguaggio ad alto livello ad assembly? Il compilatore. Chi fa passare da Assembly a linguaggio macchina? L'assemblatore.
- Di cosa si occupa la CPU? Eseguire le istruzioni.
- Perché usiamo i registri associate alle variabili ($a = x_5, b = x_{20\dots}$): Usiamo i registri associati alle variabili perché la ALU ha come input solo il contenuto dei registri (in RISC-V).
- Tra cosa si fa la somma? : Somma i valori contenuti nei registri, qualsiasi cosa essi abbiano (valori, puntatori).
- Quando usiamo i registri? : Se non siamo in un contesto particolare, possiamo usare tutti e 31 ($x_0 = 0$ sempre).
- Con l'istruzione Load (lw), perché dobbiamo usare l'offset? : Perchè altrimenti (se guardiamo l'esempio sopra), dobbiamo aggiornare ogni volta il valore di x_{21} , es :

```
a = v[0];
b = v[1];
c = v[2];
```

In assembly sarebbe (senza offset) :

```
lw x5, x21
addi x21, x21, 4
lw x6, x21
addi x21, x21, 4
lw x7, x21
```

Con offset diventa :

```
lw x5, 0(x21)
lw x5, 4(x21)
lw x5, 8(x21)
```

- Perché con sh scriviamo prima i bit meno significativi e poi quelli più significativi? : perché in un intero, occupiamo prima i bit iniziali, poi quelli dopo.
- Perché usando le istruzioni di formato SB possiamo saltare solo di numeri pari? Perché tanto le istruzioni occupano 4 byte, quindi se vogliamo saltare (per esempio) 1 istruzione vogliamo saltare sicuramente un numero pari. Non esistono istruzioni da 1 byte.
- Perchè possiamo saltare di 2 in 2 nelle istruzioni formato SB e non di 4 in 4 (visto che le istruzioni occupano 32 bit = 4 byte)? Perché, anche se non le vedremo, l'architettura prevede di supportare anche istruzioni da 2 byte. (Dopo questa domande, non sono state più segnate altre domande, ma sono ugualmente state fatte (molteplici ad ogni lezione))

Note per l'esame

Dal lato pratico, all'esame verrà valutato quanto riusciamo a rendere efficiente il programma (usare meno volte possibile la ram ad esempio e quante più volte i registri).