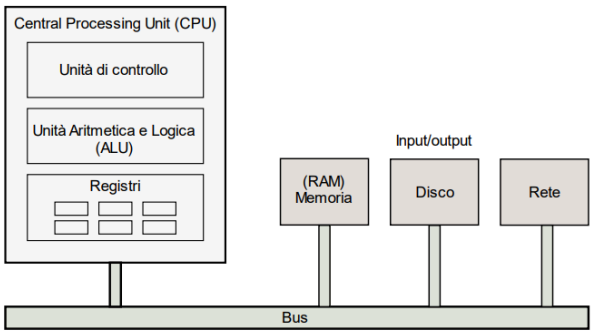
Architettura degli elaboratori

Lezione 02 - 21/02/2025

Organizzazione della CPU in una macchina di Von Neumann



- La CPU si compone di diverse parti distinte: unità di controllo, unità aritmetico-logica, registri
- I registri, l'unità aritmetico-logica e alcuni bus che li collegano compongono il data path
- Due registri importanti: Program Counter (PC) e Instruction Register (IR)
- La main memory contiene sia istruzioni sia dati usando sequenze di bit.

La CPU(processore), si occupa di eseguire le istruzioni del nostro programma; è composta da 3 blocchi:

- Unita di controllo: Capire di che tipo di istruzione si sta parlando e pilotare i bit di controllo di ALU e registri per eseguire l'istruzione.
- Unita aritmetica e Logica(ALU): è un circuito combinatorio(l'output dipende solo dall'input), si occupa di eseguire le operazioni comandate dall' unita di controllo.
- Registri: sono circuiti di memoria sequenziale(L'output dipende dallo stato della memoria), sono elementi molto costosi e veloci, presenti in numero molto

limitato all’interno del processore.

Ci sono 2 tipi di registri nella CPU:

- Program Counter(PC) che contiene **l’indirizzo** della prossima istruzione da eseguire;
- Instruction Register(IR) che contiene l’istruzione stessa.

Quando lancio un programma viene riservato lo spazio in memoria per: codice, stack, hip.

Tutte le periferiche sono collegate tra di loro dal **Bus**: un canale di comunicazione composto da fili elettrici.

Data path: organizzazione interna di una CPU(registri, ALU, bus interno).

Il nostro processore esegue un programma: FETCH-DECODE-EXECUTE:

1. Prende l’istruzione seguente dalla memoria e la mette nel registro delle istruzioni;
2. Cambia il PC per indicare l’istruzione seguente;
3. Determina il tipo d’istruzione appena letta;
4. se l’istruzione usa una parola in memoria, determina dove si trova;
5. Metti la parola in un registro della CPU(se necessario);
6. Esegui l’istruzione;
7. Torna al punto 1 per l’istruzione successiva.

i primi due punti sono la parte di FETCH, il terzo DECODE e il resto EXECUTE

Come si calcolano le prestazioni?
Tempo di esecuzione: identifica tempo tra inizio e completamento del programma

Throughput: numero di task eseguiti nell’unità di tempo

- tempo di esecuzione della CPU: tempo che la CPU utilizza nella computazione richiesta da un certo task;
- tempo di CPU utente: tempo effettivamente speso dalla CPU nella computazione richiesta da 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;
- Clock: genera un segnale costante che serve per sincronizzare le varie operazioni del nostro sistema periodo/frequenza di clock: durata di un ciclo di clock.

Per misurare le prestazioni si usano i Cicli di clock per istruzione(CPI): numero medio di cicli di clock per istruzione di un programma o di un frammento di programma.

$$\text{Tempo di CPU relativo a un programma} = \text{Cicli di clock della CPU relativi al programma} \times \text{Periodo di clock}$$

$$\text{Tempo di CPU relativo a un programma} = \frac{\text{Cicli di clock della CPU relativi a un programma}}{\text{Frequenza di clock}}$$

$$\text{Tempo di CPU} = \frac{\text{Numero di istruzioni} \times \text{CPI}}{\text{Frequenza di clock}}$$

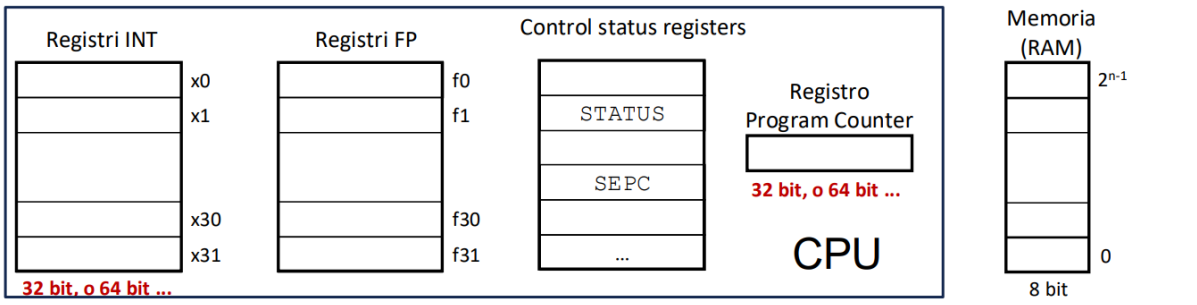
RISC-V instruction set

Ha uno standard aperto, ovvero posso costruire un processore che si basa su questa architettura senza dover pagare nulla a nessuno.

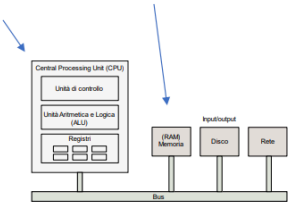
RISC sta per reduced instruction set computer, ovvero ho un set di istruzioni ridotto che svolgono compiti piu semplici che combinati opportunatamente permettono di eseguire tutte le altre operazioni.

RISC-V Registri e memoria

Parola (word): 32 bit
Parola doppia (doubleword): 64 bit



- 32 registri per gli interi (e.g., per RV64I i registri sono a 64 bit, se RV32I sono a 32 bit)
- 32 registri per i numeri in virgola mobile
  RV64D registri a 64 bit per "double precision floating points" e
  RV32F per registri a 32 bit e "single precision floating point"
- Registro Program Counter
- 4096 control status registers
- **Memoria centrale (ad esempio, 2^{64-1} indirizzi nella RV64I, 2^{32-1} indirizzi nella RV32I)**



RISC-V Instruction Set 7

In RISC-V una cella di memoria è grande 8 bit, quindi se ho un intero di 4 byte(32bit) e scrivo un istruzione di LOAD, come indirizzo specifico do un indirizzo che fa riferimento al byte meno significativo. OGNI INDIRIZZO REFERENZIA 1 BYTE.

In RISC-V ho a disposizione 2^n celle, dove n è il numero di bit che compongono l'indirizzo, quindi nel nostro caso 2^{32}.

Parola per noi referenzia un contenuto di 4 byte(32 bit), da qua deriviamo double world, half world...

Ogni registro(da x0 a x31) ha un ruolo diverso.

RISC-V Registri e memoria

Parola doppia (doubleword): 64

- Registri per gli interi
  - Quantità: 32, indicati con x0 .. X31
  - Dimensione: 64 bit se RV64I, 32 bit se RV32I
  - ....

x0	zero
x1	Return address (ra)
x2	Stack pointer (sp)
x3	Global pointer (gp)
x4	Thread pointer (tp)
x8	Frame pointer (fp)
x10-x17	Registri usati per il passaggio di parametri nelle procedure e valori di ritorno
x5-x7 , x28-x31	Registri temporanei, non salvati in caso di chiamata
x8-x9, x18-x27	Registri da salvare: il contenuto va preservato se utilizzati dalla procedura chiamata

RISC-V Instruction Set 8

Istruzioni aritmetiche

`add a, b, c` → `a = b + c`

`sub a, b, c` → `a = b - c`

in assembly, al posto di a, b, c ci saranno le celle di memoria(x5,x10...);

Lezione 03 - 25/02/2025

Istruzioni aritmetiche

quando si passa da un linguaggio ad alto livello ad un linguaggio Assembly, associo ad ogni operando un registro, scegliendo tra i 32.

`a → x5` ; `b → x20` ; `c → x21`

Usiamo i registri non per essere più veloci, ma perché è l'unico modo per fare l'addizione con un architettura RISC-V avendo un numero di istruzioni ridotte.

Quindi `add x5, x20, x21` non fa altro che mettere nel registro x5 la somma tra il contenuto di x20 e x21(posso anche sommare un indirizzo e un dato);

```
a → x5
b → x5
c → x21
add x5, x5, x21
b = b * 2
//in questo caso dovrei fare un altro accesso in memoria per prendere il valore di
//visto che nell'operazione di add l'ho sovrascritto
```

COMPILATORE: trasforma da linguaggio di alto livello in assembly

ASSEMBLATORE: trasforma il linguaggio da Assembly in binario

```
a = -a    →    sub x19, x0, x19
```

questo è un modo efficiente, perché x0 contiene sempre il valore 0;

```
f = a + b - c    →    add x19, x20, x21
                                  sub x19, x19, x22
//f → x19; a → x20; b → x21; c → x22
```

una semplice istruzione come quella sopra diventa più complicata: prima si registra il valore della somma in f, poi si sottrae c.

```
f = (g + h) - (i + j)    →    add x20, x20, x21  OPPURE  add x19, x20, x21
                                                  add x22, x22, x23          sub x19, x19, x22
                                                  sub x19, x20, x22             sub x19, x19, x23
//f → x19; g → x20; h → x21; i → x22; j → x23
// I DUE MODI HANNO LO STESSO NUMERO DI CALCOLI, QUINDI LA STESSA EF
```

Istruzioni di accesso alla memoria

`load` copia un dato dalla memoria ad un registro.

`lw` copia una parola dalla memoria ad un registro.(Attenzione, ogni elemento è grande 4 byte, quindi se devo passare da indirizzo 1 a indirizzo 2, devo fare +4).

```
lw x5, 12(x21)
// x5 è la destinazione della lettura
//12(x21) rappresenta l'indirizzo: 12 è l'offset e x21 l'indirizzo base. L 'indirizzo
//finale è la somma tra x21 e 12.
```

L'offset è compreso tra -2048 e 2047.

```
a = v[3];   →    lw x5, -12(x21)
//a → x5; v → x21
```

se specifico i dati in memoria in modo diverso l'offset può essere anche negativo.

`ld` (load doubleword) copia una parola doppia dalla memoria ad un registro.

```
ld x5, 24×21)
```

l'offset dev'essere di 8 ogni indirizzo visto che si tratta di `long int` e occupa 8 byte (ESISTE SOLO NELLE ARCHTETTURE A 64 BIT).

`sw` (store word) scrive il contenuto di un registro in memoria.

```
sw x5, 8(x21)
```

scrive in x21 + 8 il primo byte meno significativo di x5.

`sh` (store halfword) scrive in memoria il contenuto di un registro(parte dai bit meno significativi)

`sb` (store byte)

Esercizio:

```
int g,h,f;
int v[10];
...
g = h + v[3]
v[6] = g - f

⟶ g → x5; h → x9; v → x21; f → x19

lw x22, 12(x21)
add x5, x9, x22
sub x23, x5, x19
sw x23, 24(x21)
```

Lezione 04 - 26/02/2025

Si possono leggere dati senza considerare il segno, usando:

- `lbu`
- `lhu`

entrambi non eseguono l'estensione del segno.

Se uso `lh` una half word(occupa solo i due byte meno significativi) , i due byte più significativi effettuano l'estensione del segno, ovvero se la cella di memoria ha come byte più significativo 1 riempie gli altri due byte di 1 e viceversa.

Invece se uso `lhu` i bit più significativi saranno sempre 0.

Le letture in memoria avvengono secondo gruppi di byte.

Se ho un dato composto da 4byte e lo memorizzo in memoria il mio sistema farà 1 sola lettura in memoria; se invece spezzo il mio dato in 2 byte in una cella logica di

4byte e l'altro in d byte nella cella logica successiva avrò 2 letture in memoria(meno efficiente).

Operatori immediati e costanti

```
a = a + 2
a → x5
addi x5, x5, 2
```

- la costante può assumere valori tra -2048 e 2047
- non esiste la sottrazione immediata

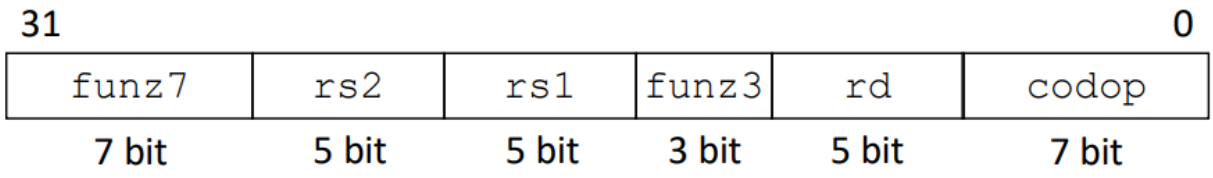
Il linguaggio macchina

Ogni istruzione RISC-V richiede esattamente 32 bit per la sua rappresentazione in linguaggio macchina. Per esempio per rappresentare in linguaggio macchina :

```
add x5, x6, x21
```

- codop → sequenza di n bit che specifica tipo di istruzione
- se ho 32 registri vuol dire che 5 bit sono sufficienti per indicare tutti i miei registri: `x0→00000` ; quindi per rappresentare i tre registri userò 15 bit.

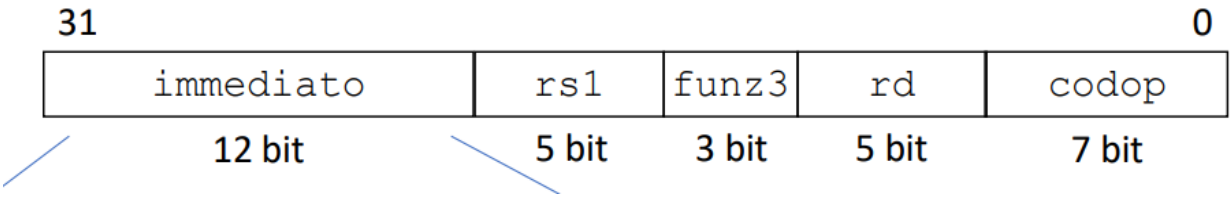
Formato di tipo R(Registro)



Permette di codificare istruzioni `add, sub, and, or, xor, ...`

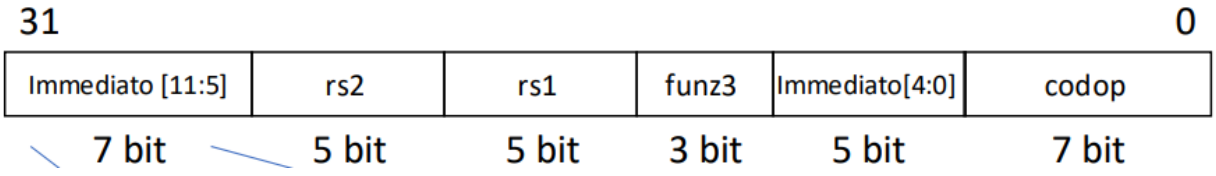
- codop: codice operativo dell'istruzione
- rd: registro di destinazione;
- rs1: registro che contiene il primo operando sorgente,
- rs2: registro che contiene il secondo operando sorgente;
- funz3,funz7: codici operatici aggiuntivi;

Formato di tipo I(immediato)



- permette id codificare istruzioni che richiedono il caricamento dalla memoria o una costante, come `load, addi, andi, ori, ...`
- sono presenti 12 bit per rappresentare la costante perché con 5 bit l'intervallo di rappresentazione per costanti e offset sarebbe stato troppo ridotto

Formato di tipo S



gli elementi sono gli stessi del tipo I ma disposti in maniera differente.

I due registri Immediato sono due registri sorgente(e non destinazione), perché leggo i dati.

Il campo immediato è diviso in due parti per facilitare la costruzione del circuito.

Operazioni logiche

Shift logico

Possono essere a dx o sx. Abbiamo istruzioni di tipo R e istruzioni di tipo I.

```
sll x9, x22, x19 → in x19 metto il valore

slli x9, x22, 5
```

```
srl x9, x22, x10
srli x9, x22, 5
```

Shift aritmetico

Esiste solo a destra: la differenza è che lo shift logico inserisce sempre 0 nel bit più a destra, mentre lo shift aritmetico inserisce 0 o 1 a seconda dell'ultimo bit più significativo.

Lo shift a sinistra esegue l'operazione di /2.

Esercizio:

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

i→x9; j→x21; v→x19

slli x6, x21, 2 // faccio uno shift a sinistra di 2 per moltiplicare per 4 il valore di j
                // perché il dato dista 4*j byte.
add x22, x19, x6//sommo in x22 l'indirizzo di v e il valore in byte di j, ottenendo v
lw x23, 0(x22)//ora in x23 ho il valore di v[j]
```

```
slli x7, x9, 2
add x22, x7, x19 // ottengo l'indirizzo di v[i]
sw x23, 0(x22)
```

Lezione 05 - 10/03/2025

```
int d,i,j;
int v[10];
...
j=5;
...
v[i+d]=v[j+2];

v→x19; d→x5; i→x9; j→x21

addi x6, x21, 2 //calcola indice j in x6
slli x6, x6, 2 //moltiplica per dimensione in byte
add x6, x6, x19 //somma l'indirizzo base e ottengo v[j+2]
lw x6, 0(x6) /*leggo l'elemento dalla memoria, l'offset è 0 perchè ho gia calcolat
l'indirizzo; se avessi dovuto fare v[i] = v[3] era motlo piu facile, bastava usareo
offset 12(ogni cella è di 4 byte→3*12)*/

add x7, x9, x5 //calcolo indice [i+d] in x7
slli x7, x7, 2 //moltiplica per dimensione in byte
add x7, x7, x19 //somma l'indirizzo base e ottengo v[i+d]
sw x6, 0(x7)
```

Operazioni logiche

Ogni espressione logica avviene confrontando bit per bit

- AND

  `and x9, x22, x19` →x9 = x22 & x19

`andi x9, x22, 5` →x9 = x22 & 5

- OR

  `or x9, x22, x19` →x9 = x22 | x19

  `ori x9, x22, 5` → x9 = x22 | 5

- XOR

  `xor x9, x22, x19` → x9 = x22 xor x19

  `xori x9, x22, 5` → x9 = x22 xor 5

- NOT (PSEUDOISTRUZIONE!)

  `not x5, x6` → `xori x5, x6, -1` → x5 = $\not{x}6$

L'AND si usa per selezionare i bit, si crea un maschera con tutti 0 tranne 1 nei bit selezionati e si confronta, ottenendo 1 solamente nei bit selezionati che valgono 1.

Salti condizionati

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

`beq rs1, rs2, L1` → "branch if equal": il programma continua all'istruzione con etichetta L1 se il valore del registro rs1 è uguale a quello di rs2.

`bne rs1, rs2, L1` →"branch if not equal": continua all'istruzione con etichetta L1 se il valore del registro rs1 è uguale a quello di rs2.

`blt rs1, rs2, L1` →"branch if less than": continua all'istruzione con etichetta L1 se il valore del registro rs1 è minore di quello di rs2.

`bge rs1, rs2, L1` →"branch if greater than or equal": continua all'istruzione con etichetta L1 se il valore del registro rs1 è minore di quello di rs2.

Nell' etichetta L1 si registra il valore dell' offset (positivo o negativo).

`bltu` → "branch if less than unsigned".

`bgeu` → “branch if grater than or equal unsigned”.

NON esistono varianti unsigned come `bequ` , non avrebbe senso.

Esercizio:

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

f → x19
g → x20
h → x21
i → x22
j → x23


beq x22, x23x THEN
beq x0, x0, ELSE
THEN:add x19, x20, x21
      beq x0, x0, ENDIF
ELSE:sub x19, x20, x21
ENDIF:

bne x22, x23, ELSE
add x19, x20, x21
beq x0, x0, ENDIF
ELSE: sub x19, x20, x21
ENDIF:
```

Esercizio:

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

i→x19

addi x3, x0, 100
addi x19, x0, 0
FOR: bge x19, x3, ENDFOR
...
addi x19, x19, 1
beq x0, x0 FOR
ENDFOR:
```

Lezione 06 - 11/03/2025

Esercizio(while):

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

i → x22
k → x24
v → x25

WHILE://è necessario che il loop parta da qua perchè sto controllando che v[i] si
    slli x10,x22,2 //shift a sinistra di 2^2 per moltiplicare per 4(numero di byte)
    add x10,x25,x10//sommando indirizzo di v e valore di i ottengo indirizzo della p
    lw x9, 0(x10)//metto il valore di v[i] in x9
```

```
    //se il confronto fosse stato tra i e k, avrei messo WHILE: in questa riga
    bne x9,x24,ENDWHILE //controllo che siano diversi, se lo sono non entro nel c
    addi x22,x22,1 //incremento di i
    beq x22,x22,WHILE
ENDWHILE:
```

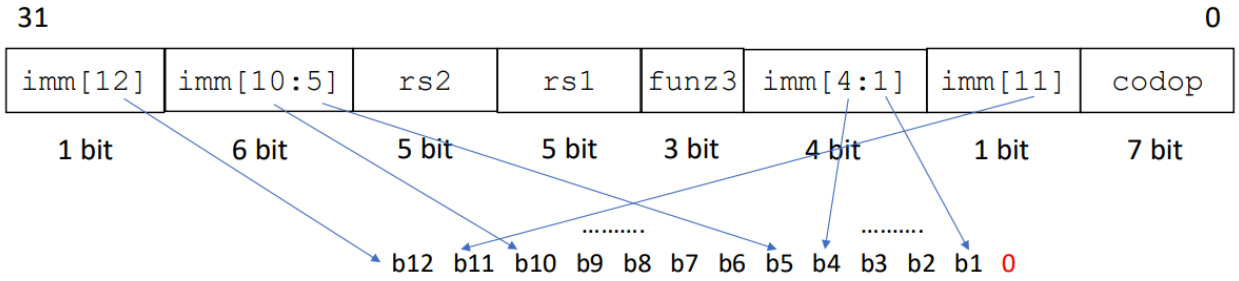
Salto condizionato

`slt x21,x19,x20` → “set less then”. Può essere utilizzata per creare strutture di controllo con istruzioni di salto generiche.

- Possiamo, ad esempio, inserire dopo `slt` l'istruzione `beq rs1,x0,L1`
  - per il confronto su ">=" basta invertire la condizione (`bne`)
  - per il confronto su ">" basta scambiare gli operandi della `slt`
  - per il confronto su "<=", inverto condizione e scambio operandi

`rs1` (nell’immagine) corrisponde a x21

Le istruzioni di salto condizionato utilizzano il formato di tipo SB



- rs2=registro
- rs1=registro
- codop
- funz3

- imm(immediato=sono usati per rappresentare l'offset dell'etichetta in questione, sono 12 bit

L'intervallo rappresentabile è -4096 a 4094: perchè abbiamo un 13 bit come bit meno significativo che vale 0. Di conseguenza tutti i valori delle etichette saranno pari(visto che siamo in 32 bit e i salti sono di numeri pari)

Istruzione aritmetiche:moltiplicazione

`mul a, b, c` →a=b*c Viene usato un registro interno nascosto all'utente, accessibile con l'istruzione:

`mulh a,b,c`

`div a,b,c` →a=b/c

`rem a,b,c` →a=b%c

Procedure (o funzioni)

Per chiamare una funzione usiamo le label, per il return devo salvarmi l'indirizzo dove metterlo.

Nella funzione chiamante:

- Mettere i parametri in un luogo accessibile alla procedura.
- Trasferire il controllo alla procedura.

Nella funzione chiamata:

- Acquisire le risorse necessarie per l'esecuzione della procedura(le variabili locali della funzione).
- Eseguire il compito richiesto.
- Mettere il risultato in un luogo accessibile al programma chiamante.
- Restituire il controllo al punto in cui è stata chiamata.

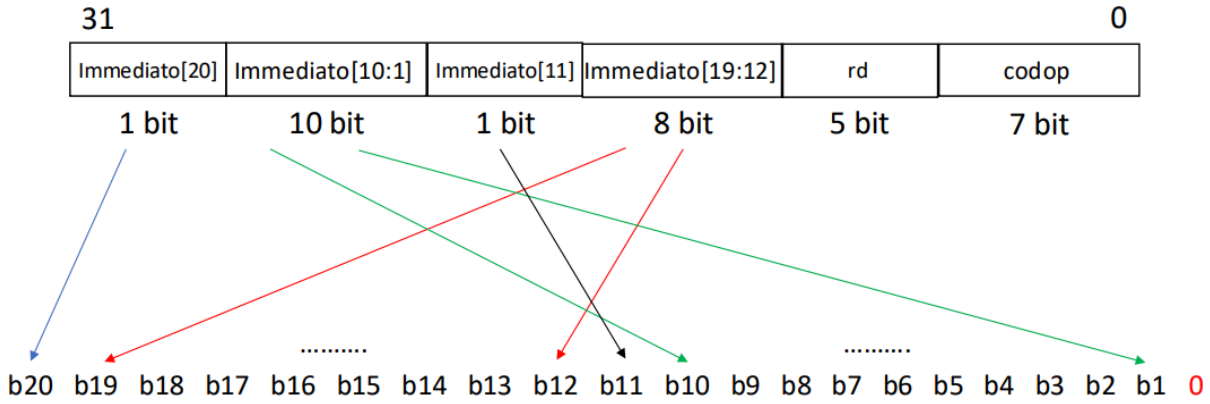
  (La stessa procedura può essere chiamata in punti differenti del programma).

`jal IndirizzoProcedura` "jump and link"

- Salta all'indirizzo(offset) con etichetta `IndirizzoProcedura`.
- Memorizza il valore dell'istruzione successiva PC+4 nel registro x1(return adress,ra).
- Pseudo-istruzione: abbreviazione di

  `jal x1, IndirizzoProcedura`

Questa istruzione è in formato J:



per dire return usiamo:

`jalr rd,offset(rs1)` "jump and link register" (Formato I).

Problemi:

Cosa succede se un registro contiene un valore usato dalla procedura chiamante?

Quando esaurisco i valori che possono stare i registri, li sposto nella ram. Una possibile soluzione del problema è salvare il valore del registro in memoria, e ripristinarlo prima del ritorno al chiamante.

In pratica faccio una store prima della chiamata, e dopo il ritorno dalla chiamata faccio una load.

Procedure annidate:

Anche qua nel caso di funzioni dentro a funzioni, devo salvare i return in memoria.

Parametri numerosi:

Se i parametri eccedono il numero di registri disponibili, anche qua salvo temporaneamente in memoria per caricarli nei registri prima del loro utilizzo all'interno della procedura.

Per salvare i registri in memoria evitando di perdere il valore uso lo stack.

Operazioni:

- PUSH: aggiunge un elemento in cima allo stack
- POP: rimuove un elemento dalla cima dello stack
- La cima dello stack è identificata dallo Stack Pointer(SP)

In RISC-V si usa la convenzione:

- grow-down: lo stack cresce da indirizzi di memoria alti verso indirizzi di memoria bassi.
- last-full: lo SP contiene l'indirizzo dell'ultima cella di memoria occupata nello stack

PUSH:

- Decremento lo SP: `addi sp, sp, -4` (lo decremento di 4 perchè il dato che voglio memorizzare è di 4 byte, volessi memorizzare una half-word avrei diminuito lo stack pointer di 2.
- Scrive in M[SP]: `sw x20, 0(sp)`

POP:

- Legge da M[SP]: `lw x20, 0(sp)`
- Incrementa SP: `addi sp, sp, 4` (sto facendo una cancellazione "logica").

Esempio:

```
SOMMA: addi sp,sp,-8
sw x5,0(sp) #salvo in memoria gli indirizzi che uso nella funzione
sw x20,4(sp) #salvo in memoria gli indirizzi che uso nella funzione
add x5,x10,x11
addi x20,x5,2
addi x10,x20,0
lw x5,0(sp) #ricarico nei registri i valori in partenza
lw x20,4(sp) #ricarico nei registri i valori in partenza
addi sp,sp,8
jalr x0,0(x1)
...
...
addi x6,x6,1
...
jal SOMMA
```

In questo caso li carico nella funzione, ma è buona norma caricare i registri in memoria quando sto chiamando la funzione, perchè magari i registri erano non utilizzati e in questo caso farei istruzione senza motivo.

`x2` è il registro che contiene lo Stack Pointer(sp)

`x8` è il registo che contiene il Frame Pointer(fp)

I registri e convenzioni sul loro uso

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	

RISC-V Instruction Set 131

La cosa importante è che si evitino le inefficienze → **Usare il minimo salvataggio dei registri**, dato che sono operazioni costose.

In generale la funzione chiamante potrebbe salvare i registri anche se la procedura chiamata non li modificherà.

Quella 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

---

Lezione 07 - 18/03/2025

Bisogna limitare salvataggi e caricamenti in memoria.

In assembly non esiste il concetto di scope di una variabile, i registri sono sempre quelli.

Convenzione nell' uso e salvataggio dei registri:

- I registri da `a0` a `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.

- 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, che non è tenuto al loro salvataggio e ripristino.

Il vantaggio di questa convenzione è che se posso usare quei registri senza preoccuparmi di salvarli, di conseguenza farò meno accessi in memoria.

Fasi di invocazione di una procedura

1. Pre-chiamata del chiamante
   - Eventuale salvataggio dei registri da preservare nel chiamante*(SI ASSUME CHE X10-X17, X5-X7 E X28-31 POSSANO ESSERE SOVRASCRITTI DAL CHIAMATO!)*

   Preparazione degli argomenti della funzione
   - I primi 8 argomenti vengono posti in x10-x17(10-17)
   - Gli altri argomenti oltre l'ottavo vanno salvati nello stack così che si trovino subito sopra il frame della funzione chiamata
2. Invocazione della procedura
   - Istruzione `jal NOME_PROCEDURA`
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(per esempio somma)
   - 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
   - Salvataggio degli argomenti `x10-x17(a0-a7)` solo se la funzione li riuserà successivamente a ulteriori chiamate a funzione
   - Eventuale inizializzazione di fp: punta al nuovo call-frame
4. Corpo della procedura
   - Istruzione che implementano il corpo della procedura

5. Epilogo lato chiamato
    - Se deve essere restituito un valore dalla funzione(in `x0 e x11`)
    - I registri(se salvati) devono essere ripristinati(`s1-s11, ra e sp`)
    - Notare che `sp` deve solo essere aumentato di opportuno offset
6. Ritorno al chiamante
    - Istruzione `jalr x0 0(x1)`
7. Post-chiamata lato chiamante
    - Eventuale uso del risultato della funzione(in `a0,a1`)
    - Ripristino dei valori `x5-x7(t0-t6) e x28-x31, x10-x17(a0-a7)` vecchi, eventualmente salvati

Se una procedura è **Foglia,** nel codice della pocedura:

- non è necessario salvare e ripristinare il registro `ra`
- non è necessario salvare sullo stack i registri `a0-a7 e t0-t6`

Esempio struttura foglia:

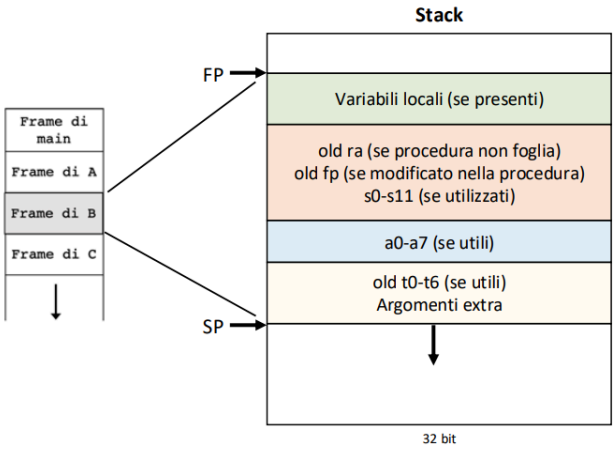
```
sommaArray:
    li t0,0//t0 = somma
    li t1,0//t1 = contatore indice i
loop:
    bge t1,a1,fine//i>= lunghezza, esci dal loop
    slli t2,t1,2//offset
    add t3, a0, t2//indirizzo elemento i
    lw t4, 0(t3)//carica v[i]
    add t0, t0, t4//aggiorna la somma
    addi t1, t1, 1//incrementa i
    j loop//ripeti il ciclo
fine:
    mv a0, t0//restituisci il risultato in a0
    jr ra//Ritorna
//Funzione main
_start:
```

```
la a0, array //Indirizzo dell'array in a0
lw a1, length // Lunghezza dell'array in a1
jal ra, sommaArray //chiamata a funzione

li a7, 93 # syscall exit (codice in a0)
ecall # Termina il programma
```

Record di attivazione: struttura

- Se utilizzato, il registro `fp` (frame pointer) viene inizializzato al valore di `sp` all'inizio della chiamata
- `fp` consente di avere un riferimento alle variabili locali che non muta con l'esecuzione della procedura
- Se lo stack non contiene variabili locali alla procedura, il compilatore risparmia tempo di esecuzione evitando di impostare e ripristinare il frame.



Esempio ricorsivo

```
int fact(int n){
    if (n==0){
        return 1;
    }else{
        return n* fact(n-1);
    }
}
→

fact:
//crea il call frame sullo stack(12 byte)
```

```
//lo stack cresce verso il basso

    addi sp, sp, -12 //allocazione del call frame nello stack
    sw a0, 8(sp) //salvataggio di n nel call frame
    sw ra, 4(sp) // salvataffio dell0indirizzo di
    sw fp, 0(sp) //salvataggio del precedente
    addi fp, sp, 8 //aggiorna,ento del frame

    //calcolo del fattoriale
    bne a0, zero, Ric //test fine ricorsione n!=0
Ric://chiamata riorsiva
    addi a0, a0, -1//a0 ← (n-1)
    jal fact //chiama fact(n-1) → risultato in a0
    lw t0, 0(fp) //t0 ← n
    j Fine//n! = (n-1)! * n

    //uscita dalla funzione
Fine:
    lw fp, 0(sp) //recupera li frame pointer
    lw ra, 4(sp) //recupera l'indirizzo di ritorno
    addi sp, sp, 12 //elimina il call frame dallo stack
    jr ra //ritorna al chiamante
```

Lezione 8 - 24/03/2025

Convenzioni di chiamata: esempio

- Data la funzione moltiplica che, dato un numero intero restituisca dalla sua moltiplicazione per 10 senza utilizzare istruzioni di moltiplicazione

```
moltiplica:
    slli t0, a0, 3 #t0 = a0*8
    slli t1, a0, 1 #t1 = a0*2
```

```
    add a0, t0, t1 # sommo
    ret
```

- scrivere una funzione `moltiplicaVettore` che
  - dato un vettore di interi(word)
  - Restituisce il vettore sostituendo ogni elemento con il suo prodotto per 10

  Ecco una versione implementata da chatGPT(con problemi):

```
moltiplicaVettore:
    beq a1, zero, fine //condizione di controllo, se a1 = zero finisce
    mv t1, a0 //copia il valore di a0 in t1
ciclo:
    lw t0, 0(t1)
    mv a0, t0 //sposto in a0 il valore caricato per prepararmi alla funzione
    jal ra, moltiplica //chiamata di funzione, in azione avrò l'elemento iesimo m
    sw a0, 0(t1) //salvo l'elemento
    addi t1, t1, 4
    addi a1, a1, -1 //sottraggo 1 all'indice
    bnez a1, ciclo //se a1 != 0 continua il ciclo
fine:
    ret
```

  I problemi sono che la funzione viene trattata come se fosse foglia: `ra` non viene salvato e ripristinato;

  Inoltre mi trovo in una funzione chiamante e sto utilizzando i registri `ra, t0, t1` ecc.. senza preoccuparmi di salvare il valore prima di chiamare la funzione.

  Quando torno dopo la chiamata di funzione, il valore di `t0` sarà diverso.

  BISOGNA SEMPRE RISPETTARE LE CONVENZIONI DI CHIAMATA!!!

  Versione con correzioni:

```
//versione corretta
moltiplicaVettore:
```

```
    addi sp, sp, -4 //salvo ra ad inizio funzione visto che non è funzione foglia
    sw ra, 0(sp)

    beq a1, zero, fine //condizione di controllo, se a1 = zero finisce
    mv t1, a0 //copia il valore di a0 in t1
ciclo:
    lw t0, 0(t1)

    addi sp, sp, -8 //aumento lo stack 4  byte per ogni elemento che devo salv
    sw t1, 0(sp) //salvo i due elementi nello stack per non perderli dopo la chia
    sw a1, 4(sp)

    mv a0, t0 //sposto in a0 il valore caricato per prepararmi alla funzione
    jal ra, moltiplica //chiamata di funzione, in azione avrò l'elemento iesimo m

    lw t1, 0(sp) //carico i due elementi dopo la chiamata della funzione
    lw a1, (4sp)
    addi sp, sp, 8 //ripristino lo stack pointer

    sw a0, 0(t1) //salvo l'elemento
    addi t1, t1, 4
    addi a1, a1, -1 //sottraggo 1 all'indice
    bnez a1, ciclo //se a1 != 0 continua il ciclo
fine:

    lw ra, 0(sp) //ripristino il valore di ra
    addi sp, sp, 4

    ret
```

Eventuali miglioramenti sono per esempio incrementare lo stack all'inizio di 12, in tal modo ottengo (2 istruzioni in meno) * (numero di iterazioni).

Posso inoltre usare `s1,s2` e salvarli una sola volta prima del ciclo e ripristinarli una sola volta alla fine del ciclo(a differenza di ora che salvo i registri `t1,a1` una

volta per ogni iterazione del ciclo), ottenendo quindi una funzione più efficiente.

Operandi immediati ampi

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

  Supponiamo di voler caricare nel registro x5 il valore 0x12345678

- Soluzione
  - si introduce una nuoca istruzione `lui` (load upper immediate, tipo U) che carica i 20 bit più significativi della costante nei bit da 12 a 31 di un registro.
  - Con una operazione di or immediao si impostano i 12 bit meno significativi rimasti

    ```
    lui x5, 0x12345
    ```

    ```
    ori x5, x5, 0x678
    ```

Riassunto dei formati delle istruzioni RISC-V

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

Figura 2.19 Formati delle istruzioni RISC-V.

Lezione 9 - 25/03/2025

Assemblatore

L’assemblatore agisce con un algoritmo a due passi:

La prima lettura serve per vedere tutte le etichette, così se ho un salto condizionato in avanti l’assemblatore sa che valore numerico dare alla label(valore di offset).

Un programma è formato da più file sorgenti che vengono tradotti in un modulo oggetto

Linker

Il **linker** è un programma che esegue la funzione di collegamento dei moduli oggetto in modo da formare un unico eseguibile.

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

Loader

Il loader prende l’eseguibile che risiede sul disco e lo fa diventare un processo, caricandolo in memoria centrale e ne avvia l’esecuzione.

Collegamento statico:

- Le funzioni di libreria diventano parte del codice eseguibile
- Tutto il codice oggetto delle librerie è all’interno dell’ eseguibile
- Il file eseguibile è molto più grande

Collegamento dinamico:

- DLL, Dynamically linked libraries
- Le funzioni di libreria non vengono collegate e caricate finché non si inizia l'esecuzione del programma
- La dimensione del file è minore: ogni procedura viene caricata solo dopo la sua prima chiamata(collegamento lazy)

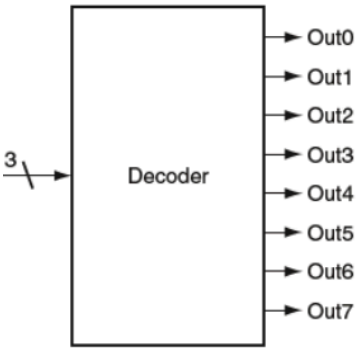
Circuiti logici digitali di base e algebra Booleana

Circuiti combinatori

Nei circuiti combinatori l'output viene determinato solo dall'input

Decoder: prende un numero n di bit come ingresso e lo usa per selezionare una delle 2^n linee di uscita.

Tabella di verità di un decoder:



a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

notiamo che abbiamo 3 input e 2^3 output.

Quando gli input `000` valgono 0 l'output con quel valore diventa 1

Ad esempio l'out6 in binario è `101` vale 6.

$\hat{I}_0 I_1 I_2$ (circuito di out6).

I_0 compare negato perchè l'output 0 di out6 vale 0.

Multiplexer: 2 ingressi, 1 uscita e 1 ingresso di controllo

Viene definito selettore, espone verso un'unica uscita sempre 1 solo input.

Addizionatore: riceve in ingresso 2 bit da sommare, un bit do riporto e restituisce un bit di ritorno con il risultato e un bit che rappresenta il riporto.

Lezione 10 - 07/04/2025

Overflow

Quando faccio una somma in complemento a 2, ci sono 2 criteri per accorgermi di overflow:

- Se sto sommando 2 numeri positivi e ottengo un numero che inizia con `1` o viceversa (Se sommo due numeri con segni opposti non otterrò **MAI** overflow, perché è impossibile uscire dall' intervallo di rappresentabilità.).
- L'ultimo riporto ottenuto è discorde dal penultimo riporto ottenuto.

Nel `beq` si realizza il salto eseguendo (a-b) e verificando se il risultato è uguale a 0-

La ALU fornisce il valore 1 quando tutti i bit del risultato sono a zero, in quanto: `(a-b) == 0→ a == b` .

Per ottenerlo quindi, faccio un OR di tutti i bit, ottenendo cosi 0 se tutti i bit valgono 0(a==b)e il NOT del risultato in modo da ottenere 1 come richiesto prima. Il segnale si chiama Zero, e vale 1 quando sono uguali.

Per il controllo dell'ALU ho una combinazione di 4 bit:

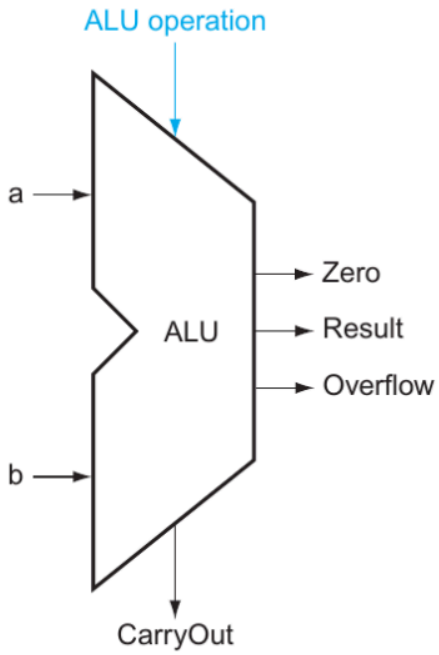
- 1 bit per l'ingresso Ainvert
- 1 bit per l'ingresso Binvert

- 2 bit per gli ingressi Operation

Ainvert	Bnegate	Operation
0	0	00
0	0	01
0	0	10
0	1	10
0	1	11
1	1	00

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR

- Il simbolo comunemente usato per rappresentare una **ALU**
- Anche comunemente usato per indicare un **addizionatore**, quindi è prassi indicare esplicitamente con una scritta quale di questi due componenti si intende.



ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR

è un circuito combinatorio: l'output riflette il risultato dei bit in input dopo un certo ritardo.

Circuiti sequenziali

I circuiti sequenziali l'output dipende dagli input e **dallo stato interno.** Quindi dipendono da informazioni memorizzate in elementi di memoria interni.

Latch

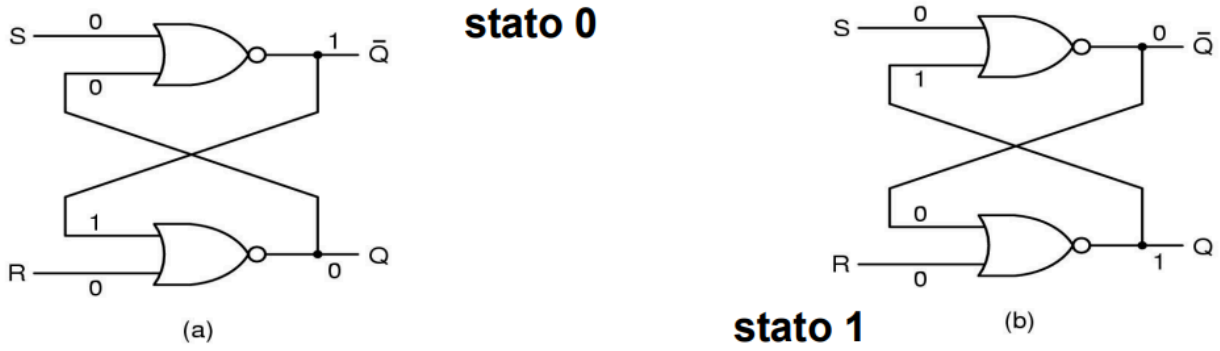
è un dispositivo a 1 bit:

- due ingressi i e β ed un'uscita o.
- mantiene uno stato interno s

  se $\beta = 1$(store),o ← s ← i Prendo l'ingresso e lo salvo

  se $\beta = 0$(hold),o ← s Qualsiasi sia l'input del latch, l'output non cambia

Latch di tipo SR



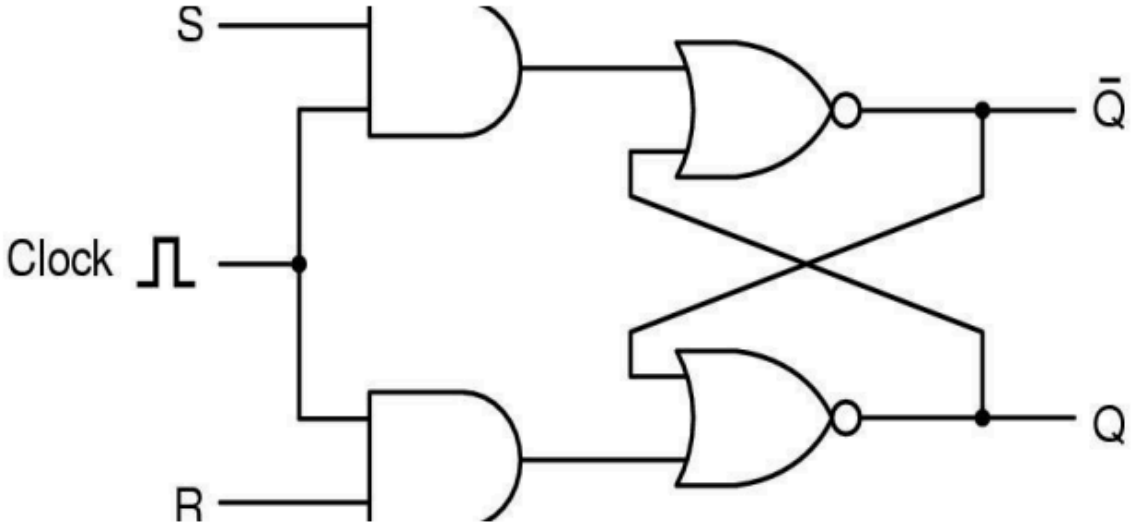
- Hold: $R = S = 0$(due stati stabili)
- Set (store 1): $S = 1$ e $R = 0$ porta il latch allo stato 1
- Reset (store o): $R = 1$ e $S = 0$ porta il latch allo stato 0

Se metto entrambi R e S a 1 otterrò
Q e $\hat{Q}$ uguali. Lo stato non è stabile, non deve mai accadere, neanche in fenomeni di transitorio. Si potrebbe innescare un' oscillazione.

Per risolvere il problema differenziamo 2 fasi: una nella quale non ci interessa il valore del segnale e un'altra in cui ci interessa, dando tempo al circuito di stabilizzare il risultato. Questo viene fatto da un **clock.**

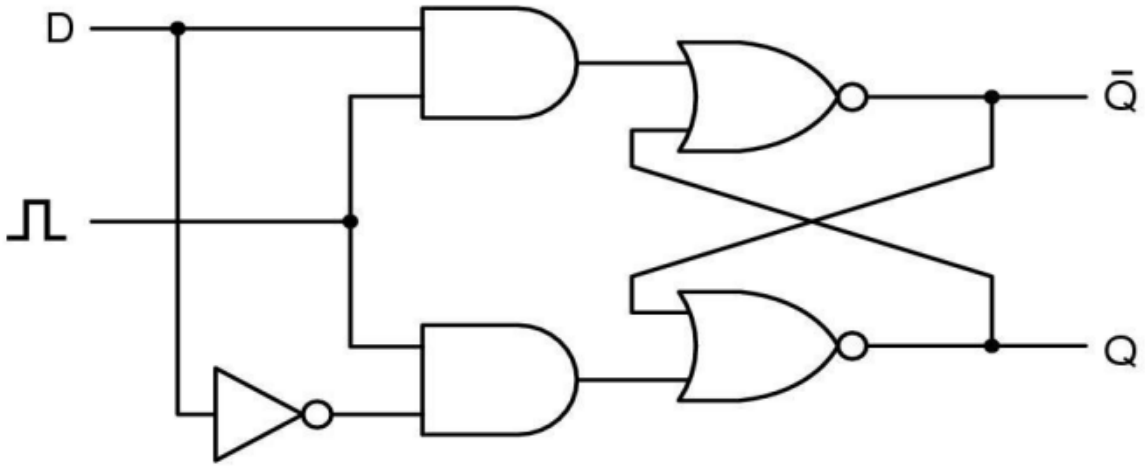
Latch di tipo SR sincronizzato

La differenza rispetto al circuito precedente è che abbiamo 2 porte and in più:



L’ effetto di queste due porte è che finché il clock è a 1 i segnali S e R si presentano alle porte nor, quando il clock è 0 qualunque sia il valore di S e R il risultato degli and sarà sempre 0, quindi posso fare qualsiasi cosa a S e R lasciando lo stato invariato.

Latch di tipo D sincronizzato



Questo ci permette di avere 1 solo input(magari quello di una ALU), memorizzando il valore di D quando il clock è 1, rende impossibile avere una combinazione di `11`.

ATTENZIONE: Questo rende impossibile lo stato di Hold perché non avrò mai `00` .

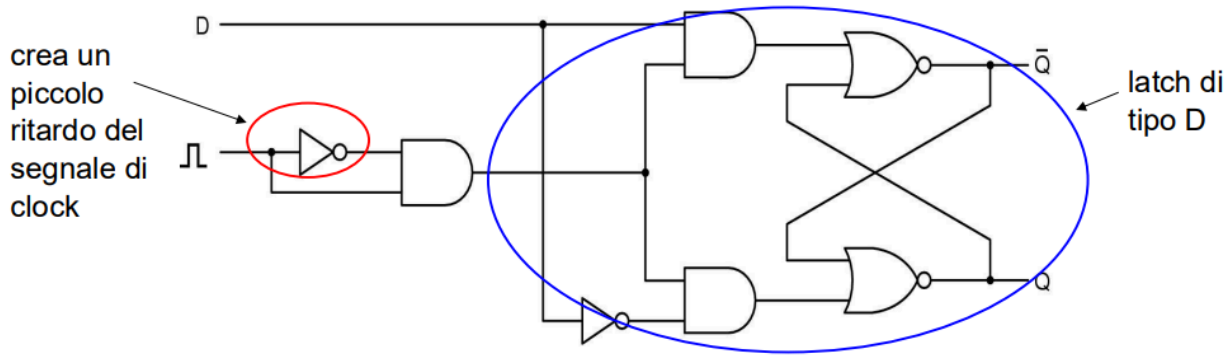
Problema della trasparenza

Con i Latch abbiamo il problema della “trasparenza”. L’input di un latch deve essere mantenuto invariato, altrimenti questo si ripropone come input e cambia il risultato.

Se l’output del circuito diventa poi l’input dello steso circuito otteniamo questo problema.

La soluzione è rendere più corto il periodo in cui il clock è a 1. Per far ciò modifico il circuito:

Flip-flop di tipo D



così facendo aggiungendo un and tra il clock e il clock negato, si ottiene 0, ma col not si crea un piccolo intervallo temporale di ritardo nel quale l’ and avrà un uscita che vale 1, memorizzando l’ingresso D. Dopo un piccolo lasso di tempo il clock varrà di nuovo 0.

Un latch viene azionato a livello, mentre un flip-flop è azionato dal fronte(di salita o discesa).

Lezione 11 - 15/04/2025

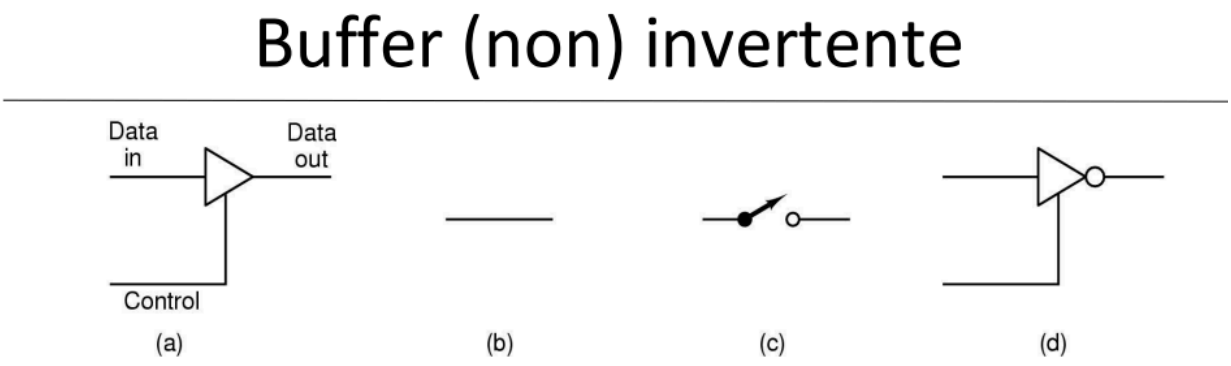
D-latch

La memorizzazione avviene in istanti diversi rispetto al segnale a gradino del clock:

- commutazione a livello
- commutazione dal fronte

Per trasformare il d-latch in circuito che commuta dal fronte, inseriamo(come si vede nell'immagine sopra) un not per creare un piccolo ritardo, ottenendo così il flop-flop di tipo D.

Buffer (non)invertente



- Il buffer (non) invertente (detto anche tri-state) si comporta come un filo quando l'ingresso Control è alto (caso b)
- Il buffer (non) invertente disconnette Datain e Dataout quando l'ingresso Control è basso (caso c)
- Buffer invertente (caso d)

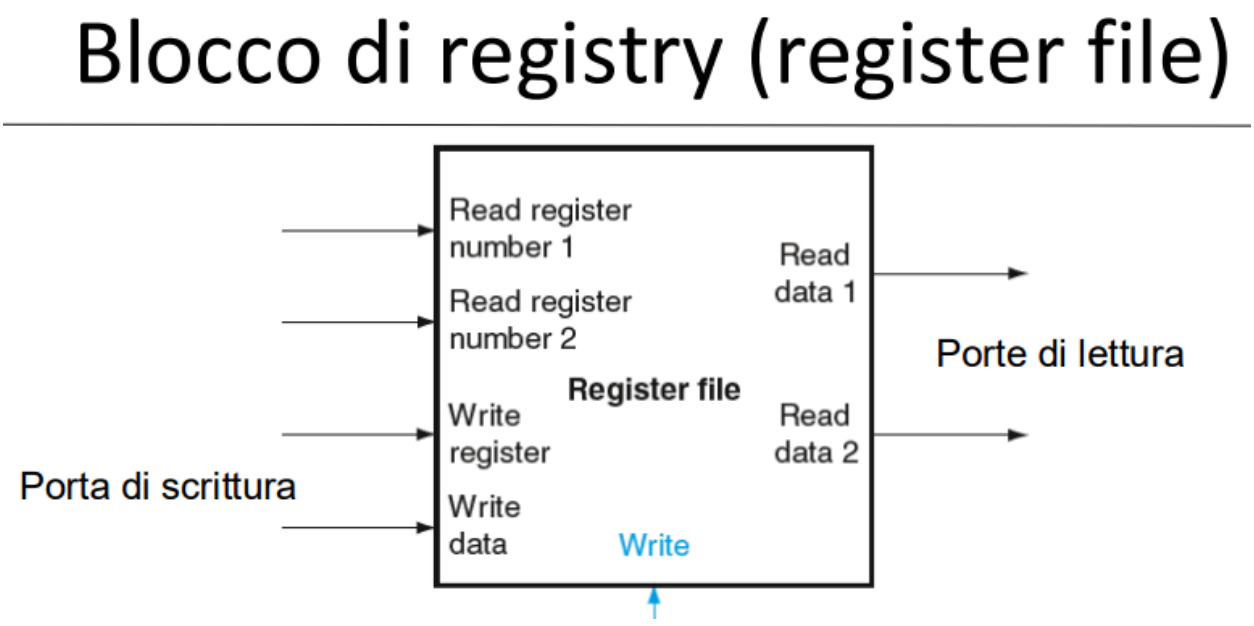
Ha un input e un output, si comporta come un dispositivo con 3 stati:

- il primo: quando il bit di controllo vale 1, il triangolo si comporta come un circuito chiuso, ovvero se ho 1 in input ottengo 1 in output e viceversa
- il secondo: quando il bit di controllo vale 0 il circuito è aperto, non ho output
- il terzo: assenza di segnale

Registri

- sono un insieme di flip-flop D raggruppati
- Lo stesso clock è in ingresso a tutti i flip-flop del registro
- L'ingresso enable permette di (dis)connettere il registro dal bus di output tramite buffer non invertenti.

Blocco di registry(register file)



Il nostro register file contiene 32 registri + logica di controllo.

Per esempio:

`add x1, x2, x3` (formato R, 5 dei bit dicono che voglio selezionare x2(00010), altri 5 che dicono che voglio selezionare x3(00011)).

I due input(read register number 1 e 2) sono quindi composti da 5 bit(x1,x2 in questo caso)

Le due uscite(read data 1 e 2) sono di grandezza 32 bit (se ci troviamo in un architettura a 32 bit)

Per quanto riguarda la scrittura abbiamo il write register, che contiene il registro dell' output(x1), composto sempre da 5 bit.

Write data invece è il valore che voglio scrivere all interno del registro in write register, essendo un valore è rappresentato da 32 bit.

Il segnale write (in blu) è grande 1 bit, serve per dire che in alcuni casi voglio scrivere nel registro (write == 1)selezionato, in altri casi no(write ==0).

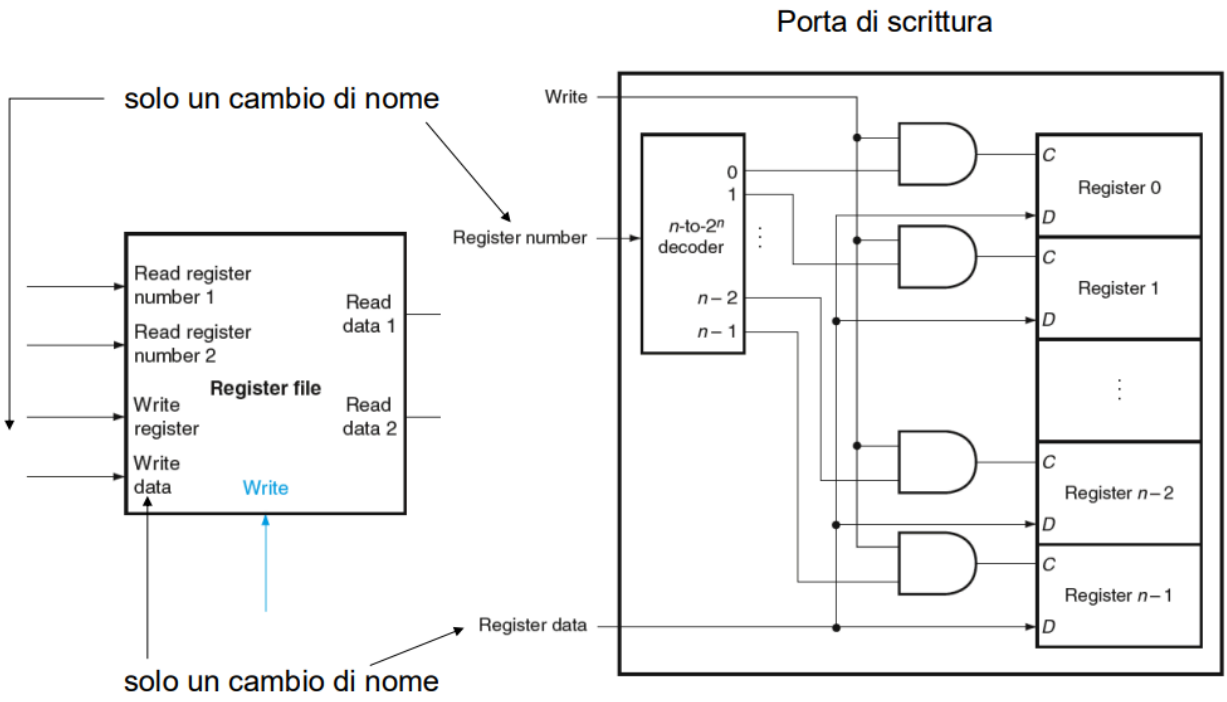
I casi in cui non scrivo nei registri sono le store ad esempio, perchè prendo un valore e lo salvo in un indirizzo(anche `beq x1, x2 jump` ad esempio).

Ho 32 registri, come faccio a selezionarne solo 1 con 5 bit? Lo faccio con un multiplexer che avrà $n = 5$ bit di controllo(perché $2^n = 32 -> n = 5$).

E invece come faccio a scrivere il valore di un registro in uno solo dei 32 registri? qui mi serve un

decoder, che ha n ingressi e 2^n uscite, nel nostro caso avrà 2^{32} bit di cui solo 1 varrà 1 lasciando a 0 gli altri

Blocco di registry (register file)



in questo modo scriviamo solo su un registro e leggiamo solo da un registro.

Chip di memoria

- n righe di indirizzi(A1-A18) corrispondono a 2^n righe di flip-flop.
- b linee di output(D0-D7)corrispondono a b colonne di flip flop.
- I segnali possono essere attivi quando il livello è basso o alto(specificato nel chip)
- Matrici $2^n * b$: selezione della colonna e della riga.