Indice

Tipi di calcolatori e loro caratteristiche

Nonostante calcolatori molto diversi tra loro condividano la stessa tecnologia hardware, nella maggior parte dei casi le soluzioni utilizzate non sono identiche. Infatti queste applicazioni sono caratterizzate da requisiti di progetto differenti che implicano un diverso utilizzo dell'hardware.
A grandi linee, i calcolatori possono essere raggruppati in tre classi ben distinte.

I **personal computer** rappresentano il tipo di calcolatore più conosciuto; essi offrono buone prestazioni a un singolo utente mantenendo il costo limitato; inoltre vengono spesso utilizzati per eseguire software scritto da terze parti.

I **server** sono la forma moderna di quelli che un tempo erano calcolatori di dimensioni decisamente maggiori e, di norma, ad essi si accede solo attraverso la rete. Essi sono orientati all'elaborazione di carichi di lavoro di grosse dimensioni. I server sono realizzati con le stesse tecnologie di un PC, ma offrono una maggiore potenza di calcolo, una maggiore velocità di input/output e una maggiore capacità della memoria.

I **calcolatori embedded** (cioè dedicati) sono i più numerosi e coprono un ampio spettro di applicazioni e prestazioni. I sistemi di calcolo di questo tipo sono progettati per eseguire una singola applicazione o un insieme di applicazioni correlate tra loro; queste applicazioni sono di norma integrate con l'hardware e si presentano all'utente come un sistema monolitico.
Le applicazioni di tipo embedded richiedono spesso prestazioni limitate con vincoli stringenti sul costo e sulla potenza assorbita dal dispositivo.

Legge di Moore

Una delle costanti dei calcolatori è la rapida evoluzione, descritta principalmente dalla **legge di Moore**, la quale stabilisce che le risorse messe a disposizione dai circuiti integrati vengano duplicate ogni 18-24 mesi.

Che cosa c'è dietro un programma

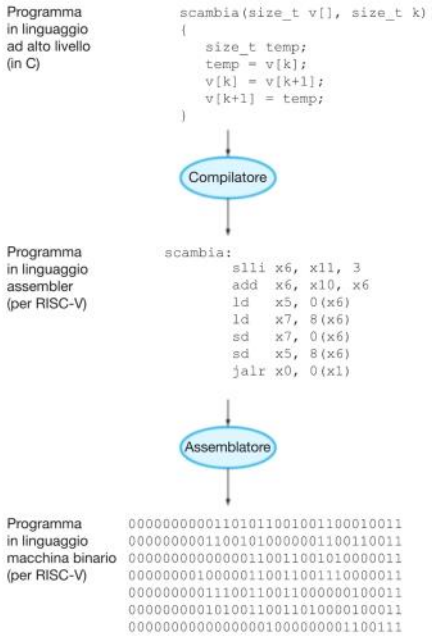
Il calcolatore può eseguire solo istruzioni di basso livello estremamente semplici. Passare da un'applicazione complessa alle semplici istruzioni comprensibili al calcolatore è un processo che coinvolge diversi strati di software, organizzati principalmente in maniera gerarchica.

Nel cerchio più esterno compaiono le applicazioni, mentre i diversi componenti del **software di sistema** sono posizionati nel cerchio intermedio tra l'hardware e le applicazioni software.
Il software di sistema ha diversi componenti, ma sono due quelli essenziali per tutti i calcolatori moderni: il sistema operativo e il compilatore.

Il **sistema operativo** permette di interfacciare i programmi utente con l'hardware del calcolatore, fornendo un gran numero di servizi e funzioni di supervisione.

I **compilatori** eseguono la traduzione di un programma scritto in linguaggio ad alto livello in istruzioni eseguibili dall'hardware, il che è una funzione complessa.

Da un linguaggio ad alto livello a un linguaggio hardware



Per parlare con una macchina elettronica è necessario inviare segnali elettrici; i segnali che un calcolatore può comprendere facilmente sono *on* e *off*, quindi utilizziamo un alfabeto binario fatto di *0* e *1*.
Il **linguaggio macchina** è la composizione di numeri binari, dove ogni cifra (*binary digit*) è un **bit**.
Una **istruzione** è una stringa (o insieme) di bit che l'hardware del computer comprende ed esegue.
Il **compilatore** è il programma che traduce le istruzioni scritte in linguaggio ad alto livello in istruzioni assembler.

Componenti di un calcolatore

Il **processore** (**CPU**) è la parte attiva del calcolatore, quella che esegue fedelmente le istruzioni di un programma: è in grado di effettuare somme tra numeri, fare test su essi, inviare segnali per attivare dispositivi di I/O e così via.
Il processore comprende due componenti principali: l'**unità di controllo** e l'**unità di elaborazione dati** (**datapath**). L'unità di controllo invia i comandi all'unità di elaborazione dati, alla memoria e ai dispositivi di I/O secondo le istruzioni del programma.
Il datapath provvede a eseguire le operazioni aritmetico-logiche sui dati.
La **memoria** è il luogo dove vengono tenuti i programmi in esecuzione assieme ai loro dati.
DRAM e **RAM** sono memorie **ad accesso casuale** e l'accesso richiede lo stesso tempo indipendentemente dalla particolare area di memoria a cui si accede.
La **memoria cache** consiste in una memoria piccola ma veloce che funge da tampone nei confronti della DRAM che invece è più grande e più lenta. La cache è costruita usando una tecnologia di memorie di tipo **SRAM**, ovvero **statica ad accesso casuale**.
L'**ISA** (*Instruction Set Architecture*) è l'**interfaccia astratta** tra hardware e il livello più basso del software del calcolatore. Comprende tutte le informazioni necessarie per scrivere un programma in linguaggio macchina funzionante in modo corretto, comprese istruzioni, registri, gli accessi alla memoria, I/O etc.

Gerarchia delle memorie

Una **memoria volatile** è in grado di mantenere i dati solamente se è alimentata. Un esempio di memoria volatile è la DRAM.
Una **memoria non volatile**, invece, conserva i dati anche quando viene a mancare l'alimentazione; viene utilizzata per conservare i dati tra un'esecuzione e l'altra.

La **memoria principale**, detta anche **memoria primaria**, viene utilizzata per contenere i programmi durante la loro esecuzione.
La **memoria di massa**, detta anche **memoria secondaria**, è una memoria non volatile utilizzata per conservare i programmi e i dati tra un'esecuzione e l'altra.

Prestazioni

Tempo di risposta: detto anche **tempo di esecuzione**, è il tempo totale richiesto da un calcolatore per completare un task; esso comprende gli accessi a disco, gli accessi a memoria, le attività di I/O, il tempo richiesto dal sistema operativo, il tempo di esecuzione della CPU etc.

Throughput: detto anche **larghezza di banda**, rappresenta il numero di programmi completati per unità di tempo.

Per massimizzare le prestazioni vogliamo minimizzare il tempo di esecuzione richiesto da un dato task. Possiamo quindi mettere in relazione le prestazioni con il tempo di esecuzione; per un generico calcolatore X varrà la relazione:

$$\text{Prestazioni}_X = \frac{1}{\text{Tempo di esecuzione}_X}$$

Misurare le prestazioni

Il calcolatore che esegue un certo lavoro nel tempo minore è il più veloce.
I calcolatori lavorano spesso in condivisione e può accadere che un processore stia lavorando su più programmi contemporaneamente; per questo motivo si distingue tra tempo assoluto di esecuzione di un programma e tempo durante il quale il processore ha effettivamente lavorato su quel programma
Il **tempo di esecuzione della CPU** è il tempo effettivamente speso dalla CPU nella computazione richiesta dal programma e non comprende il tempo speso per le operazioni di I/O o nell'esecuzione di altri programmi.
Il tempo di CPU può essere ulteriormente suddiviso in **tempo di CPU utente** (tempo effettivamente speso dalla CPU nella computazione richiesta da un programma) e **tempo di CPU di sistema** (tempo speso dalla CPU per eseguire le funzioni del sistema operativo richieste per l'esecuzione di un programma).

Quasi tutti i calcolatori sono costruiti utilizzando un segnale che sincronizza le varie funzioni implementate nell'hardware; questo segnale è periodico nel tempo e i relativi intervalli di tempo sono i **cicli di clock**. Il **periodo di clock** è il tempo necessario per completare un intero ciclo di clock; la ***frequenza di clock*** è il suo inverso.

Prestazioni della CPU

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

o analogamente

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

Prestazioni

Misura delle prestazioni associate alle istruzioni

Il numero di cicli di clock necessari per l'esecuzione di un programma si può scrivere come:

$$\text{Cicli di clock della CPU} = \frac{\text{Numero di istruzioni del programma}}{} \times \frac{\text{Numero medio di cicli di clock per istruzione}}{}$$

Cicli di clock per istruzione (**CPI**): Numero medio di cicli di clock che le diverse istruzioni richiedono per essere completate.

Dato che istruzioni diverse possono richiedere un tempo di esecuzione differente in funzione del compito che svolgono, il CPI è una quantità utile a confrontare due calcolatori diversi che condividono la stessa architettura dell'insieme di istruzioni.

Possiamo quindi scrivere questa equazione fondamentale in funzione del numero di istruzioni, del CPI e del periodo di clock:

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

o analogamente:

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

Componente hardware o software	Che cosa influenza?	Come?
Algoritmo	Numero di istruzioni, eventualmente il CPI	L'algoritmo determina il numero di istruzioni del programma sorgente e quindi il numero di istruzioni in linguaggio macchina che vengono eseguite dal processore. L'algoritmo può anche influenzare il CPI, favorendo l'utilizzo di istruzioni più o meno veloci. Per esempio, se l'algoritmo utilizza più operazioni di divisione, tenderà ad avere un CPI più elevato
Linguaggio di programmazione	Numero di istruzioni, CPI	Il linguaggio di programmazione influenza certamente il numero di istruzioni, dal momento che i costrutti del linguaggio ad alto livello sono tradotti in istruzioni in linguaggio macchina e queste determinano il numero di istruzioni eseguite. Il linguaggio ad alto livello può anche influenzare il CPI a seconda delle sue caratteristiche; per esempio, un linguaggio con un esteso supporto per i dati astratti (per es. Java) richiederà chiamate indirette a funzione, che sono caratterizzate da un CPI più alto
Compilatore	Numero di istruzioni, CPI	L'efficienza del compilatore influenza sia il numero di istruzioni sia il numero medio di cicli per istruzione, dal momento che il compilatore traduce le istruzioni dal linguaggio sorgente ad alto livello nelle istruzioni in linguaggio macchina. Il ruolo del compilatore può essere molto complesso e può influenzare il CPI in maniera complessa
Architettura dell'insieme delle istruzioni	Numero di istruzioni, frequenza di clock, CPI	L'architettura dell'insieme di istruzioni influenza tutti e tre i fattori delle prestazioni della CPU, dato che influenza le istruzioni richieste da una data funzione, il costo in numero di cicli di ogni istruzione e la frequenza del processore

Il linguaggio dei calcolatori

Operazioni svolte dall'hardware

Qualsiasi calcolatore deve saper eseguire le operazioni aritmetiche.
A differenza di altri linguaggi di programmazione, in un linguaggio Assembler ciascuna linea può contenere al massimo una istruzione.
Il numero di operandi per una operazione come la somma è pari a tre: i due numeri da sommare e il riferimento alla locazione in cui memorizzare il risultato.

```
add a, b, c    // la somma di b e c è posta in a
```

Il fatto di richiedere che tutte le istruzioni abbiano esattamente tre operandi è conforme alla filosofia di mantenere l'hardware semplice.
Il primo principio per la progettazione dell'hardware è: *la semplicità favorisce la regolarità*.

Operandi dell'hardware

Gli operandi delle istruzioni aritmetiche del RISC-V devono obbedire ad alcune restrizioni: devono essere scelti tra un numero limitato di locazioni particolari, chiamate *registri*.

Operandi RISC-V

Nome	Esempio	Commenti
32 registri	x0–x31	Accesso veloce ai dati. Nel RISC-V gli operandi devono essere contenuti nei registri per potere eseguire delle operazioni. Il registro x0 contiene sempre il valore 0
2^{61} parole di memoria	Memoria[0], Memoria[8], … Memoria[18 446 744 073 709 551 608]	Alla memoria si accede solamente attraverso istruzioni di trasferimento dati. Il RISC-V utilizza l'indirizzamento al byte, perciò due variabili ampie due parole (double word) hanno indirizzi in memoria a distanza 8. La memoria consente di memorizzare strutture dati, vettori, o il contenuto dei registri

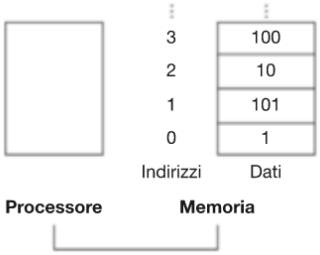
I registri rappresentano sia le primitive utilizzate nella progettazione dell'hardware sia gli elementi visibili al programmatore.
La dimensione dei registri nell'architettura RISC-V è di 64 bit; gruppi di 64 bit prendono il nome di **doubleword**, a cui si accede naturalmente in un calcolatore.
Una **word** è invece è il numero di bit a cui si accede più naturalmente in un calcolatore ed è costituita da 32 bit.

Una delle differenze più importanti fra le variabili utilizzate nei linguaggi di programmazione e i registri è il numero limitato di questi ultimi; infatti sono esattamente 32 nei calcolatori RISC-V.
La ragione di questa limitazione si trova nel secondo principio fondamentale per la progettazione dell'hardware: *minori sono le dimensioni, maggiore è la velocità*.
Un numero molto elevato di registri potrebbe aumentare la durata del ciclo di clock semplicemente perché i segnali elettrici impiegherebbero un tempo maggiore a compiere il percorso assegnato.

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

Operandi allocati in memoria

Benché il processore possa contenere un numero limitato di dati nei registri, la memoria può contenere miliardi di dati. Di conseguenza le strutture dati (vettori e strutture) vengono allocate in memoria.
Nel RISC-V le istruzioni aritmetiche richiedono che gli operandi siano memorizzati nei registri; l'assembler RISC-V deve quindi contenere delle **istruzioni di trasferimento dati** che trasferiscono dati fra la memoria e i registri.
Per accedere a una word o doubleword in memoria, l'istruzione deve fornire l'**indirizzo** di memoria corrispondente.
La memoria può essere vista come un grande vettore monodimensionale, con l'indirizzo che funge da indice e parte a contare da zero.



Dato che il RISC-V utilizza l'indirizzamento della memoria al singolo byte e ogni doubleword contiene 8 byte, gli indirizzi sono multipli di 8.
La figura sopra quindi deve essere corretta tenendo conto di questo **vincolo di allineamento**:

L'istruzione che sposta un dato dalla memoria a un registro è la `load` (carica).

Esempio: Sia A un vettore di doubleword contenuto in memoria, voglio trasferire il contenuto di `A[8]` in un registro:

```
ld x9, 64(x22)   // il valore A[8] viene caricato nel
                 // registro x9
```

L'indirizzo dell'elemento `A[8]` è dato dalla somma dell'indirizzo base (`x22`) e il numero che permette di selezionare l'elemento 8, ovvero $8*8 = 64$.
`x22` è detto *registro base* e la costante si chiama *offset*.

L'istruzione che invece sposta un dato da un registro alla memoria è la *store*.

Esempio: Voglio memorizzare in $A[12]$ (che sta in memoria) la somma tra l'elemento `A[8]` e la variabile contenuta in `x21`.

```
ld x9, 64(x22)    // il valore A[8] viene caricato nel
                  // registro x9
add x9, x21, x9   // il registro x9 assume il valore della
                  // somma
sd x9, 96(x22)   // memorizza la somma in A[12]
```

L'indirizzo dell'elemento `A[12]` è dato dalla somma dell'indirizzo base (`x22`) e il numero che permette di selezionare l'elemento 12, ovvero $8*12 = 96$.
Come per la load, `x22` è detto *registro base* e la costante si chiama *offset*.

Operandi immediati o costanti

Spesso i programmi utilizzano all'interno di una operazione una costante, per esempio per incrementare l'indice di un contatore in modo da puntare all'elemento successivo di un vettore.
In più della metà delle operazioni aritmetiche RISC-V uno degli operandi è una costante.

La versione dell'operazione di somma in cui un operando è una costante è chiamata `addi` (*add immediate*).

```
addi x22, x22, 4   // x22 = x22 + 4
```

Le operazioni su costanti sono molto frequenti. Inserendo le costanti all'interno delle istruzioni aritmetiche, le operazioni risultano molto più veloci e richiedono meno energia rispetto al caso in cui le costanti siano caricate in memoria.

La costante *zero* ha il ruolo di semplificare l'insieme delle istruzioni, consentendo di realizzare delle utili varianti.
Per esempio, si può **negare** il contenuto di un registro utilizzando l'operazione `sub` con zero come primo operando.
Per questo motivo, i RISC-V dedicano il registro `x0` a contenere il valore prefissato di zero.

Rappresentazione delle istruzioni nel calcolatore

Anche le istruzioni nel calcolatore sono memorizzate come una sequenza di segnali elettrici e vengono rappresentate con stringhe di bit.
L'istruzione viene scomposta in campi di numeri binari secondo quello che viene chiamato **formato dell'istruzione**.
L'istruzione RISC-V richiede esattamente 32 bit, una word.
Ricordando che **linguaggio macchina** è la rappresentazione binaria utilizzata per la comunicazione all'interno dei calcolatori, una sequenza di istruzioni in linguaggio macchina viene definita **codice macchina**.
Per evitare la lettura e la scrittura di lunghe stringhe in binario, si ricorre all'uso della numerazione **esadecimale** (numeri in base 16), che può essere convertita facilmente in binario.

Esadecimale	Binario	Esadecimale	Binario	Esadecimale	Binario	Esadecimale	Binario
0_{esa}	0000_{due}	4_{esa}	0100_{due}	8_{esa}	1000_{due}	c_{esa}	1100_{due}
1_{esa}	0001_{due}	5_{esa}	0101_{due}	9_{esa}	1001_{due}	d_{esa}	1101_{due}
2_{esa}	0010_{due}	6_{esa}	0110_{due}	a_{esa}	1010_{due}	e_{esa}	1110_{due}
3_{esa}	0011_{due}	7_{esa}	0111_{due}	b_{esa}	1011_{due}	f_{esa}	1111_{due}

Figura 2.4 Tabella di conversione tra binario ed esadecimale. È sufficiente sostituire una cifra esadecimale con le corrispondenti quattro cifre binarie e viceversa. Se la lunghezza del numero binario non è un multiplo di 4, si procede da destra a sinistra.

Campi delle istruzioni RISC-V

Ai diversi campi delle istruzioni viene associato un nome:

funz7	rs2	rs1	funz3	rd	codop
7 bit	5 bit	5 bit	3 bit	5 bit	7 bit

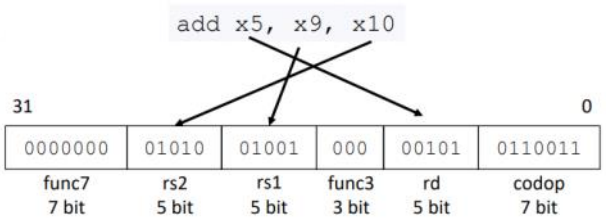
Con il seguente significato:

- *codop*: **codice operativo** che specifica operazione e formato dell'istruzione stessa;
- *rd*: registro destinazione: riceve il risultato dell'operazione;
- *funz3*: un codice operativo aggiuntivo;
- *rs1*: registro contenente il primo operando sorgente;
- *rs2*: registro contenente il secondo operando sorgente;
- *funz7*: un codice operativo aggiuntivo.

Può nascere un problema quando un'istruzione richiede campi di dimensioni maggiori rispetto a quelle sopra specificate, come nel caso della load, che richiede di specificare due registri e una costante: se la costante venisse inserita in un campo da 5 bit, non potrebbe superare il valore di $2^5 - 1$, cioè 31.
Nasce un conflitto fra il desiderio di mantenere la stessa lunghezza per tutte le istruzioni e quello di avere un unico formato. Introduciamo quindi il terzo principio fondamentale della progettazione dell'hardware: *un buon progetto richiede buoni compromessi*.

Nel RISC-V si è deciso di mantenere uguale la lunghezza di tutte le istruzioni, predisponendo formati diversi per tipi di istruzioni diverse.
Il formato descritto sopra è chiamato di *tipo R* (R sta per registro).

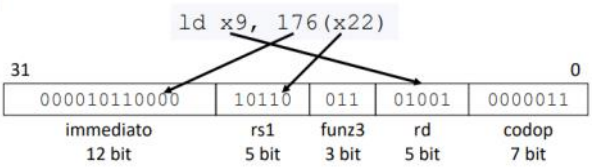
Esempio

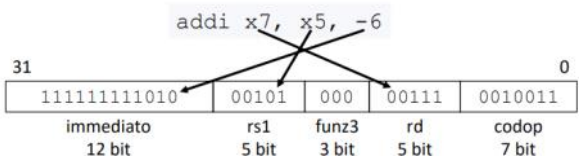


Un altro tipo di formato è detto *tipo I* (I sta per immediato) e viene utilizzato dalle operazioni in cui un operando è una costante:

immediato	rs1	funz3	rd	codop
12 bit	5 bit	3 bit	5 bit	7 bit

Esempi





Questa soluzione consente di mantenere i campi *rs1* e *rs2* nella stessa posizione in tutti i formati di istruzioni; inoltre i campi *codop* e *funct3* hanno la stessa dimensione e si trovano nella stessa posizione in tutte le istruzioni.
Si possono distinguere i due formati in base al codice operativo: a ciascun formato è assegnato un insieme di valori del campo codop, in modo tale che l'hardware sappia esattamente come deve trattare i rimanenti bit dell'istruzione.

Operazioni logiche

Sono operazioni utili a operare su gruppi di bit o su singoli bit di una word.

Operazioni logiche	Istruzioni RISC-V
Shift a sinistra	`sll, slli`
Shift a destra	`srl, srli`
Shift a destra aritmetico	`sra, srai`
AND bit a bit	`and, andi`
OR bit a bit	`or, ori`
XOR bit a bit	`xor, xori`
NOT bit a bit	`xori`

La prima tipologia di queste operazioni è la **shift** (*scorrimento*) e consiste nello spostare tutti i bit di una word a sinistra o a destra, riempiendo i bit vuoti con degli zeri.

Esempio supponiamo che il registro x19 contenga:

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001001_{due} = 9_{dec}$

ed eseguiamo l'istruzione di shift a sinistra di 4; il numero ottenuto sarà:

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 10010000_{due} = 144_{dec}$

Questa operazione è eseguibile tramite l'istruzione `slli` (*shift left logical immediate*):

```
slli x11, x19, 4    // reg x11 = x19 << 4 bit
```

L'operazione duale a quella riportata sopra è la `srli` (*shift right logical immediate*).

funz6	immediato	rs1	funz3	rd	codop
0	4	19	1	11	19

Queste operazioni di shift utilizzano il formato di tipo I.
Dato che non serve far scorrere i bit di un registro formato da 64 bit per più di 63 posizioni, solamente i 6 bit meno significativi del campo immediato vengono effettivamente utilizzati.
I rimanenti 6 bit vengono utilizzati come un campo aggiuntivo di codice operativo.

L'operazione di shift logico fornisce una ulteriore funzionalità: lo scorrimento di un numero a sinistra di *i* cifre produce lo stesso risultato di una moltiplicazione per 2^i.
Allo stesso modo, vediamo un terzo tipo di operazione di shift: lo *scorrimento a destra* aritmetico `srai`, simile alla `srll` tranne che invece di riempire con degli zeri, i bit che si liberano vengono riempiti copiando il bit del segno.
Il RISC-V fornisce anche una variante per ognuna di queste operazioni che, anzi che utilizzare un immediato, prendono il numero con cui fare lo scorrimento da un registro: `sll`, `srl` e `sra`.

Un'altra operazione logica, che permette di **isolare** i campi di una word, è **AND**.
L'operazione di AND è un'operazione logica bit a bit su due operandi, che restituisce 1 se entrambi gli operandi sono uguali a 1, 0 altrimenti.

Esempio supponiamo che il registro x11 contenga:

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001101\ 11000000_{due}$

e il registro x10 contenga:

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00111100\ 00000000_{due}$

il valore contenuto nel registro x9 sarebbe:

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00001100\ 00000000_{due}$

Questa operazione si scrive con l'istruzione:

```
and x9, x10, x11    // x9 = x10 & x11
```

L'operazione di AND bit a bit può essere usata per **forzare a 0** i bit di una word fornendo in input all'AND una parola contenente zeri in quelle posizioni (questa parola viene chiamata *maschera*).

Esempio:

```
and x5,x6,x7
```

x6	00100100 00010101 00001011 10100110	Sorgente
x7	00000100 00000110 00000010 00010010	Maschera
x5	00000100 00000100 00000010 00000010	Risultato

L'**OR** è un'operazione bit a bit su due operandi, che restituisce 1 se *almeno uno dei due operandi* è uguale a 1.

Esempio supponiamo di avere gli stessi registri visti sopra:

il risultato dell'operazione `or x9, x10, x11  // x9 = x10 | x11` sarà:

$00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00000000\ 00111101\ 11000000_{due}$

L'operazione di OR bit a bit può essere usata per **forzare a 1** i bit di una word fornendo la maschera adatta.

Esempio:

```
or x5,x6,x7
```

x6	00100100 00010101 00001011 10100110	Sorgente
x7	00000100 00000110 01000010 00010010	Maschera
x5	00100100 00010111 01001011 10110110	Risultato

L'ultima operazione logica è la **negazione**: l'operazione di **NOT** prende un operando e inverte i bit (tutti gli 1 diventano 0 e viceversa).
Per mantenere il formato dell'operazione a tre operandi, nel RISC-V abbiamo l'operazione **XOR** (OR esclusivo) invece della negazione semplice.
Dato che XOR produce 1 quando i valori dei due operandi sono diversi, si può ottenere NOT tramite `xor` di un numero con 111…111.

Istruzioni per prendere decisioni

Il calcolatore è capace di prendere decisioni: in base ai dati in ingresso e ai valori calcolati durante l'elaborazione, possono essere eseguite istruzioni diverse.

Il RISC-V è in grado di implementare un processo decisionale simile al costrutto *if* tramite istruzioni di **salto condizionato** *(conditional branches)*:

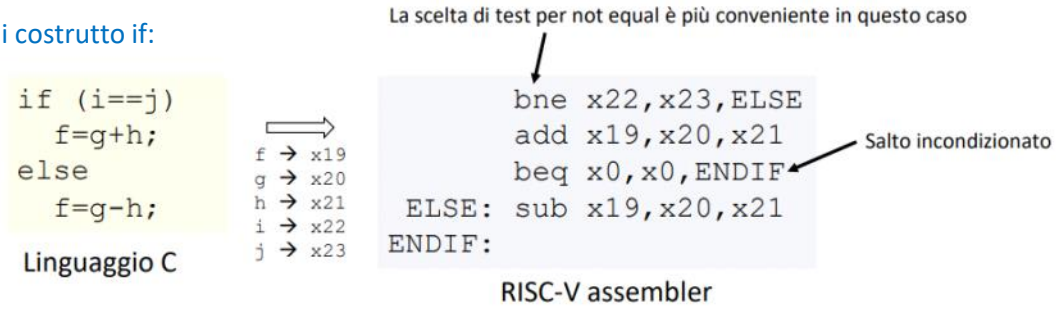
```
beq rs1, rs2, L1
```

e significa: vai all'istruzione etichettata `L1` se il valore contenuto in `rs1` corrisponde a quello contenuto in `rs2`;
`beq` significa *branch if equal* (salta se uguale).
Analogamente, abbiamo l'istruzione bne, che significa *branch if not equal* (salta se non uguale):

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

e significa: vai all'istruzione etichettata `L1` se il valore contenuto in `rs1` *non* corrisponde a quello contenuto in `rs2`.

Esempio di costrutto if:



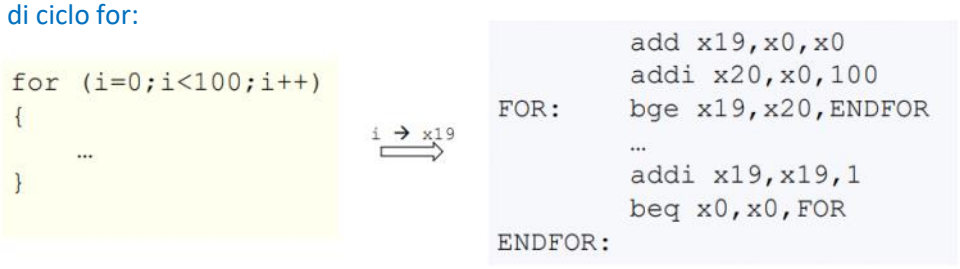
N.B.	L'assembler evita al compilatore o al programmatore il compito di calcolare gli indirizzi dei salti.

L'insieme dei confronti possibili prevede, oltre l'uguaglianza e disuguaglianza viste sopra, anche: $<, \leq, >, \geq$.
La comparazione di stringhe di bit deve prevedere sia i numeri dotati di segno sia quelli senza.
Questi controlli possono essere effettuati tramite:

- salta se minore: `blt` (*branch if less than*)
- salta se minore o uguale: `ble` (*branch if less than or equal*)
- salta se maggiore: `bgt` (*branch if greater than*)
- salta se maggiore o uguale: `bge` (*branch if greater than or equal*)
- salta se minore di (senza segno): `bltu`
- salta se maggiore o uguale di (senza segno): `bgeu`

Le stesse istruzioni assembler possono essere utilizzate anche per l'implementazione di **cicli**.

Esempio di ciclo for:



Procedure

Una **procedura** è un sottoprogramma utilizzato in modo da rendere l'intero programma più comprensibile e permettere il riutilizzo del codice.
Le procedure consentono ai programmatori di concentrarsi su una parte del problema alla volta; l'interfaccia fra la procedura e il resto del programma e dei dati è costituita dai *parametri*, i quali permettono di passare dei valori alla procedura e di restituire i risultati al programma chiamante.
Per eseguire una procedura, un programma deve eseguire questi sei passi:

1. mettere i parametri in un luogo accessibile alla procedura;
2. trasferire il controllo ad essa;
3. acquisire le risorse necessarie per la sua esecuzione;
4. eseguire il compito richiesto;
5. mettere il risultato in un luogo accessibile al programma chiamante;
6. restituire il controllo al punto di origine, dato che la stessa procedura può essere chiamata in diversi punti del programma.

Il software RISC-V per le chiamate a procedura utilizza i registri secondo queste convenzioni:

- `x10-x17` (`a0-a7`): registri argomento per il passaggio dei parametri o la restituzione dei valori calcolati;
- `x1` (`ra`): registro contenente l'indirizzo di ritorno per tornare al punto di origine.

L'istruzione per passare alla procedura è `jal` (**jump and link**), la quale esegue un salto all'indirizzo della procedura e contemporaneamente salva nel registro `ra` l'indirizzo dell'istruzione successiva, detto **indirizzo di ritorno**:

```
jal ra, EtichettaProcedura
```

L'indirizzo di ritorno è necessario, perché la stessa procedura può essere chiamata da diversi punti del programma.

Utilizzo di più registri

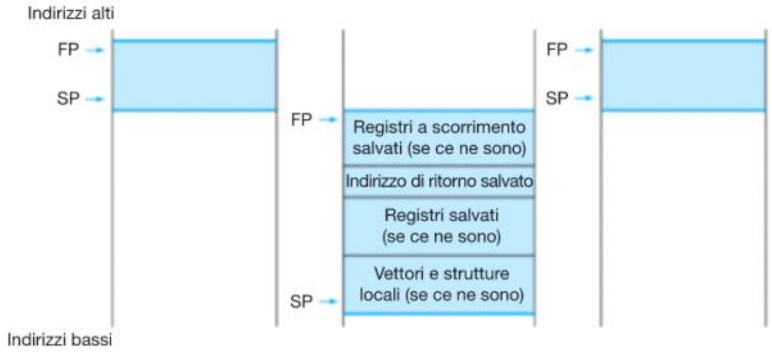
Supponiamo che il compilatore abbia bisogno, all'intero della procedura, di un numero maggiore di registri rispetto agli 8 elencati sopra.
Il contenuto dei registri utilizzati dal chiamante deve essere ripristinato con il valore *precedente* alla chiamata. Per questo motivo è necessario copiare il contenuto dei registri in memoria, più precisamente nello **stack**, cioè una coda di tipo LIFO (*first-in-first-out*).
Lo stack ha bisogno di un puntatore all'indirizzo dell'ultimo dato introdotto per indicare il punto in cui la procedura successiva può salvare il contenuto dei registri e da dove poi possa recuperarli per ripristinarli.
Nel RISC-V il puntatore allo stack (**stack pointer**) è il registro `x2` (`sp`).
Lo stack pointer viene incrementato o decrementato di una doubleword ogni volta che si toglie (**pop**) o si inserisce (**push**) il contenuto di un registro.
Lo stack "cresce" a partire da indirizzi di memoria alti verso indirizzi di memoria bassi: quando si inseriscono contenuti lo stack, il valore dello stack pointer diminuisce; quando i dati vengono estratti, lo stack pointer aumenta, riducendo la dimensione dello stack.

Per evitare di salvare e ripristinare registri il cui valore non verrà mai utilizzato, si suddividono 19 dei registri in due gruppi:

- `x5-x7` (`t0-t2`) e `x28-x31` (`t3-t6`): registri temporanei, che non sono salvati in caso di chiamata di una procedura;
- `x8-x9` (`fp-s1`) e `x18-x27` (`s2-s11`): registri da salvare il cui contenuto deve essere preservato in caso di chiamata a procedura.

Allocazione dello spazio nello stack

Lo stack può essere utilizzato anche per memorizzare le variabili locali della procedura che non trovano spazio nei registri. Il segmento dello stack che contiene i registri salvati da una procedura e le variabili locali prende il nome di **record di attivazione**, o **frame della procedura**.



Allocazione dello stack prima, durante e dopo la chiamata della procedura.

Il **frame pointer** (`fp`) è il valore che individua la posizione dei registri salvati e delle variabili locali di una data procedura.
Lo stack pointer dovrebbe cambiare durante l'esecuzione di una procedura; in questo caso il riferimento alle variabili locali in memoria potrebbe assumere offset diversi a seconda della loro posizione nella procedura. L'utilizzo del frame pointer è vantaggioso proprio perché fa sì che i riferimenti alle variabili in stack di una procedura mantengano lo stesso offset.

Che cosa succederebbe se si volessero passare più di otto parametri a una procedura?

Per convenzione i parametri aggiuntivi vengono messi nello stack al di sopra dell'indirizzo puntato dal frame pointer: la procedura si aspetterà i primi otto parametri nei registri appositi e i restanti nell'area di stack, indirizzabili attraverso `fp`.

Indirizzamento nei salti

Le istruzioni di salto condizionato RISC-V utilizzano il formato di *tipo-SB*.
Questo formato può rappresentare indirizzi di salto da -4096 a 4094 **in multipli di 2**: è possibile saltare solo a indirizzi pari.
Il formato di tipo SB consiste in 7 bit di codice operativo, 3 bit di codice funzione, due registri operandi su 5 bit e un campo immediato di indirizzo. Quest'ultimo è implementato con una codifica insolita, che semplifica l'elaborazione da parte della CPU ma complica l'assembler.

L'istruzione di salto incondizionato *jump-and-link* (`jal`) è l'unica che utilizza il formato di *tipo UJ*, che consiste in un codice operativo di 7 bit, un registro operando di destinazione su 5 bit e un indirizzo immediato su 20 bit. L'indirizzo dell'istruzione successiva viene scritto nel campo rd.
Come per il formato di tipo SB, l'operando che contiene l'indirizzo in questo formato utilizza una codifica insolita e non può codificare gli indirizzi dispari.

Se gli indirizzi del programma trovassero posto in questo campo a 20 bit, risulterebbe che nessun programma potrebbe avere una dimensione superiore a 2^{20}, troppo piccola per essere utilizzata nelle applicazioni reali. Una valida alternativa consiste nello specificare un registro il cui contenuto deve essere sommato all'indirizzo del salto; l'istruzione di salto dovrebbe quindi effettuare il seguente calcolo:

$$\text{Program counter} = \text{Registro} + \text{Spiazzamento del salto}$$

Questa somma consentirebbe al programma di indirizzare 2^{64} posizioni pur continuando a utilizzare i salti condizionati, risolvendo il problema della dimensione dell'indirizzo di salto.

Indirizzamento nei salti

Il metodo di indirizzamento utilizzato si chiama **indirizzamento relativo al program counter** (*PC-relative addressing*). Grazie a questo metodo, il RISC-V consente di effettuare salti molto lunghi a uno qualsiasi tra 2^{32} indirizzi utilizzando una sequenza di due istruzioni: `lui` scrive i bit da 12 a 31 in un registro temporaneo e `jalr` somma i 12 bit meno significativi all'indirizzo ottenuto con la somma.
Dato che le istruzioni RISC-V sono ampie 4 byte, le istruzioni di salto sono state progettate per ampliare il loro spazio di indirizzamento definendo l'indirizzo relativo al PC in termini di numero di *word* tra l'istruzione corrente di salto e l'istruzione di destinazione del salto, invece che in termini di numero di byte. Tuttavia gli architetti RISC-V hanno voluto supportare la possibilità che le istruzioni siano ampie 2 byte, per cui lo spiazzamento viene definito nelle istruzioni di salto in termini di *half word* che intercorrono tra l'indirizzo dell'istruzione corrente e quello di destinazione del salto.
Quindi il campo di indirizzi di 20 bit nell'istruzione `jal` può codificare una distanza di $\pm 2^{19}$ half word a partire dal valore attuale del PC.
Analogamente, anche il campo di 12 bit delle istruzioni di salto condizionato è espresso anch'esso in termini di half word; questo vuol dire che rappresenta un indirizzo di 13 bit in termini di byte.

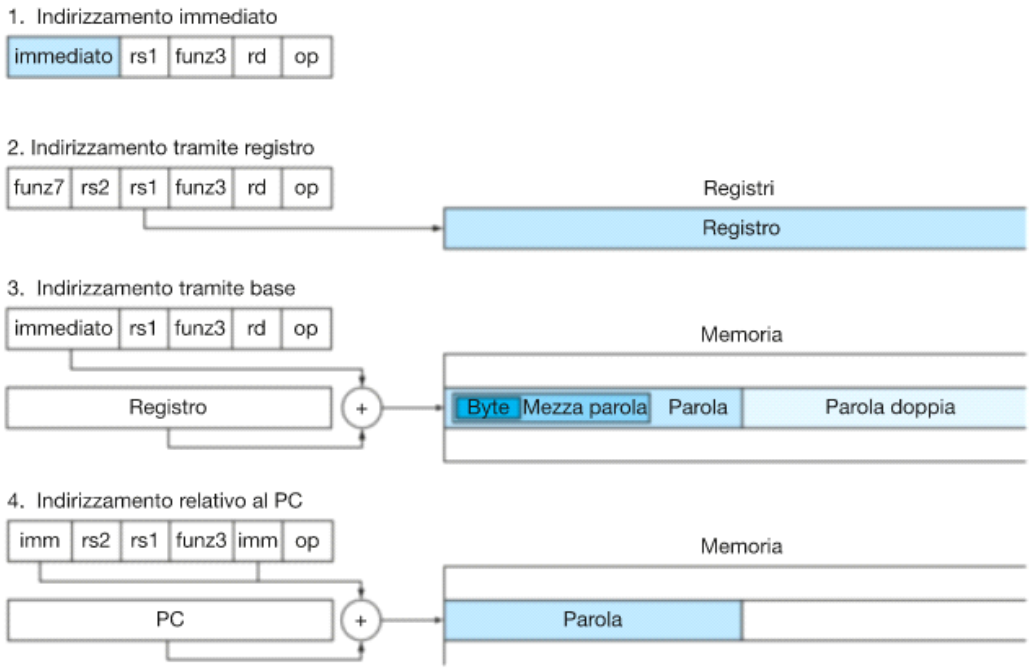
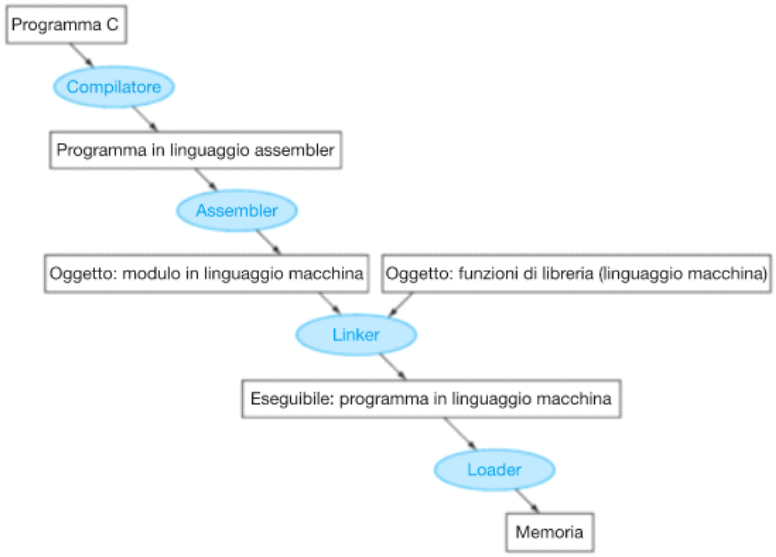


Figura 2.17 Illustrazione delle quattro modalità di indirizzamento del RISC-V. Gli operandi sono evidenziati in blu. L'operando della modalità 3 si trova in memoria, mentre quello della modalità 2 si trova in un registro. Si noti che le varianti delle istruzioni di load e store possono accedere al byte, alla mezza parola, alla parola o alla parola doppia. Nella modalità 1, l'operando è contenuto nell'istruzione stessa. La modalità 4 viene utilizzata per indirizzare le istruzioni in memoria, aggiungendo un indirizzo ampio al PC. Si noti che un'operazione può utilizzare diverse modalità di indirizzamento: la somma, per esempio, può avere sia un operando immediato (`addi`) sia tutti gli operandi nei registri (`add`).

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

Tradurre e avviare un programma



Un programma scritto in linguaggio ad alto livello viene prima di tutto compilato in linguaggio assembler e poi assemblato per ottenere un modulo oggetto in linguaggio macchina. Il linker unisce uno o più moduli tra loro e con le procedure contenute nelle librerie, e risolve tutti i riferimenti incrociati. Il loader, infine, carica il codice macchina nell'opportuna area di memoria, in modo che possa essere eseguito dal processore.
Per accelerare il processo, alcuni passi possono essere saltati o eseguiti in parallelo.

Compilatore

Trasforma il programma C (o qualsiasi altro di alto livello) in un *programma in linguaggio assembler*, cioè in una forma simbolica di ciò che il calcolatore è in grado di comprendere.

Assemblatore

Dato che il linguaggio assembler rappresenta l'interfaccia verso il software di livello più alto, l'assembler può anche trattare varianti delle istruzioni in linguaggio macchina come se fossero istruzioni vere e proprie, semplificando la traduzione e la programmazione e aumentando la leggibilità del codice.
Queste istruzioni sono chiamate **pseudoistruzioni**.

Alcuni esempi:

```
li x9, 123    // load immediate (carico in x9 una costante)
```

sarà convertito dall'assemblatore in

```
addi x9, x0, 123
```

```
mv x10, x11  // move (sposta il valore di x11 in x10)
```

diventa

```
addi x10, x11, 0
```

```
j Etichetta  // salto incondizionato
```

corrisponde all'istruzione

```
jal x0, Etichetta
```

Riassumendo, le pseudoistruzioni consentono all'assembler di avere un insieme di istruzioni più ricco di quello implementato in hardware.
Inoltre, gli assemblatori accettano numeri espressi in basi diverse (oltre a binario, abbiamo visto che è valida la notazione decimale, quella ottale e quella esadecimale), che poi verranno convertite in sequenza di bit.

- Per produrre la versione in linguaggio macchina di ogni istruzione, l'assemblatore deve determinare gli indirizzi corrispondenti a tutte le etichette: esso tiene traccia di tutte le etichette utilizzate nei salti e nei trasferimenti di dati scrivendole in una tabella, detta **tabella dei simboli** (*symbol table*), che contiene coppie di tipo simbolo-indirizzo.

Linker

Per evitare che la modifica anche di una sola linea di codice richieda di ricompilare e riassemblare l'intero programma, con un conseguente spreco di risorse computazionali, è necessario un sistema che permetta di compilare e assemblare ciascuna procedura indipendentemente dalle altre. In tal modo, ciascuna modifica di una linea di codice rende necessario ricompilare e riassemblare *solo* la procedura a cui la linea di codice appartiene.
Per fare ciò è necessario un nuovo programma di sistema, il **link editor** o **linker**; esso prende *tutti* i programmi (le procedure) in codice macchina che sono stati assemblati indipendentemente e li unisce.
Il motivo per cui il linker è particolarmente utile è che risulta molto più veloce correggere il codice piuttosto che ricompilarlo e riassemblarlo di nuovo.
Il linker esegue tre passi:

1. inserisce in memoria in modo simbolico il codice e i moduli dati;
2. determina gli indirizzi dei dati e delle etichette che compaiono nelle istruzioni;
3. corregge i riferimenti interni ed esterni.

In pratica il linker utilizza le informazioni di rilocazione e la tabella dei simboli di ciascun modulo oggetto per risolvere tutte le etichette non definite (nelle istruzioni di salto e negli indirizzi dei dati).
Una volta risolti tutti i riferimenti esterni, il linker determina le locazioni di memoria che ciascun modulo dovrà occupare.
Il linker produce un **file eseguibile** (programma nel formato dei file oggetto, che non contiene riferimenti non risolti) che può essere eseguito su un calcolatore.

Loader

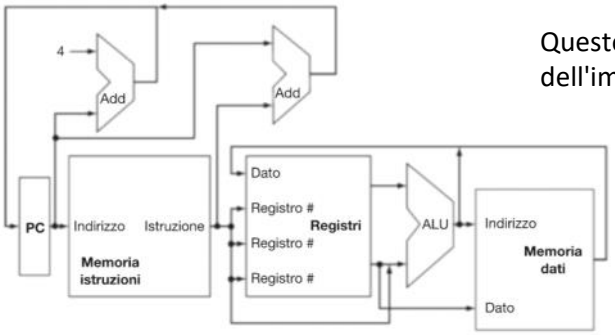
Una volta che il file eseguibile è stato memorizzato su disco, il sistema operativo può leggerlo e trasferirlo tramite il loader in memoria per avviarne l'esecuzione.

Il processore

Esamineremo un'implementazione che comprende le seguenti istruzioni di base del RISC-V:
- le istruzioni di riferimento alla memoria `ld` e `sd`;
- le istruzioni aritmetico-logiche `add`, `sub`, `and` e `or`;
- le istruzioni di salto condizionato dal risultato di un test di uguaglianza, `beq`.

Per ogni istruzione i primi due passi sono identici:
1. inviare il contenuto del PC alla memoria che contiene il programma e prelevare l'istruzione dalla memoria (fase di *fetch*);
2. leggere il contenuto di uno o due registri utilizzando i campi dell'istruzione per selezionare i registri.

Dopo queste due fasi, le azioni richieste per completare l'esecuzione delle istruzioni dipendono dalla loro tipologia. Tutti i tipi di istruzioni, eccetto i salti condizionati, utilizzano l'unità aritmetico-logica (ALU) dopo aver letto i registri.



Questo è uno schema ad alto livello di astrazione dell'implementazione di un RISC-V.

Convenzioni del progetto logico

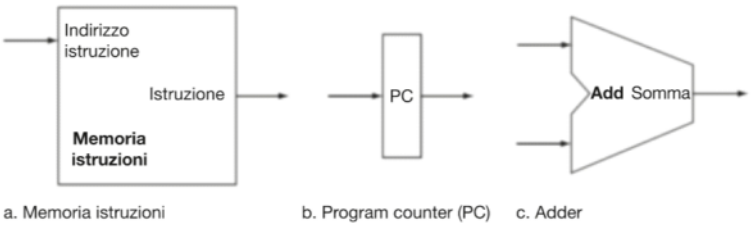
Gli elementi funzionali che costituiscono l'unità di elaborazione del RISC-V sono costituiti da die diverse classi di elementi logici: elementi che operano sui dati, detti **combinatori**, ed elementi che contengono lo stato, detti **sequenziali**.
La ALU raffigurata sopra è un esempio di elemento combinatorio, ovvero che in ogni istante i suoi output dipendono solo dagli input ricevuti nello stesso istante.
Gli elementi sequenziali contengono lo *stato* e, per questo, hanno al loro interno elementi di memoria.
La memoria istruzioni, la memoria dati e i registri della figura sopra sono esempi di elementi di stato (sequenziali).
Un elemento di stato possiede almeno due ingressi e un'uscita. Gli ingressi richiesti sono il valore da scrivere nell'elemento e il clock, che determina quando scrivere. L'uscita di un elemento di stato è il valore contenuto al suo interno, scritto in un ciclo di clock precedente.

Metodologia di temporizzazione

La **metodologia di temporizzazione** definisce quando i segnali possono essere scritti e quando possono essere letti. È importante temporizzare le operazioni di lettura e scrittura perché, se un segnale venisse letto e contemporaneamente scritto, il valore letto potrebbe non corrispondere a quello atteso.
Utilizziamo una metodologia di **temporizzazione sensibile ai fronti** (*edge-triggered*): essa garantisce che il valore memorizzato all'interno di un elemento sequenziale venga aggiornato solamente in corrispondenza di un fronte del segnale di clock.
Poiché solo gli elementi sequenziali possono memorizzare i dati, un qualsiasi circuito combinatorio deve ricevere l'input da un insieme di elementi di stato. Gli ingressi sono i valori che erano stati scritti in un ciclo di clock precedente, mentre gli output del circuito combinatorio sono i valori che potranno essere utilizzati in un ciclo di clock successivo.
La metodologia sensibile ai fronti permette di leggere il contenuto di un registro, inviare il valore attraverso uno o più blocchi di logica combinatoria e scrivere lo stesso registro nello stesso ciclo di clock.
In questo modo non si rischia di innescare una retroazione all'interno dello stesso ciclo di clock, e il circuito funziona correttamente.

Quasi tutti gli elementi di stato e combinatori dell'architettura RISC-V a 64 bit hanno ingressi e uscite di ampiezza pari a 64 bit, essendo l'ampiezza della maggior parte dei dati elaborati dal processore.

Realizzazione di un'unità di elaborazione



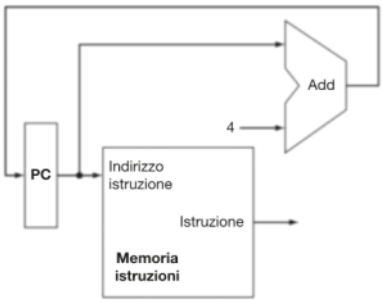
I primi elementi dell'**unità di elaborazione** (*datapath elements*) sono:

a. un'unità di memoria in cui salvare le istruzioni del programma e che sia in grado di fornire in uscita l'istruzione associata all'indirizzo dato in ingresso;

b. Il program counter, che è un registro utilizzato per memorizzare l'indirizzo dell'istruzione corrente;

c. Un sommatore per incrementare il PC (di 4 byte nel caso più semplice) e ottenere l'indirizzo dell'istruzione successiva (può essere costruito a partire dalla ALU).

Si possono combinare i tre elementi per formare una unità di elaborazione che prelevi le istruzioni e incrementi il PC per ottenere l'indirizzo dell'istruzione successiva del programma.



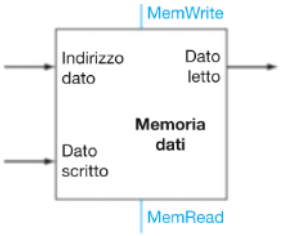
Istruzioni in formato R (add, sub, and, or)

Tutte le istruzioni di questo tipo leggono due registri, eseguono un'operazione con la ALU sul contenuto di questi due registri e, alla fine, scrivono il risultato in un registro.
I registri universali a 32 bit del processore sono raccolti nel **register file**, un insieme di registri in cui ciascuno di essi può essere letto o scritto specificando il numero ad esso associato all'interno dell'insieme. Il register file contiene lo stato dei registri del calcolatore. Avremo inoltre bisogno di una ALU per operare sui valori letti.
Dato che le istruzioni di tipo R hanno tre registri come operandi, per ciascuna istruzione dovremo leggere due dati di una word ciascuno dal register file e poi scrivere il risultato, sempre nel register file.
Per scrivere un dato di una word serviranno due ingressi: il primo deve specificare il *numero del registro di scrittura*, il secondo deve fornire il *dato* da scrivere.
Il register file fornisce in qualsiasi momento in uscita il contenuto del registro letto; la scrittura, invece, viene controllata da un segnale di controllo esplicito, "RegWrite".
Quindi serviranno complessivamente quattro ingressi e due uscite.
Gli ingressi che specificano il numero dei registri hanno ampiezza di 5 bit in modo da specificare 32 registri,
Mentre i bus dei dati in ingresso e in uscita sono di 64 bit ciascuno.

Qui è mostrata una ALU che riceve due input di 64 bit e produce un risultato su 64 bit; inoltre produce un segnale a 1 bit che vale 1 se il risultato dell'operazione è 0.

Istruzioni di caricamento di un registro (load) e di trasferimento alla memoria (store)

Queste istruzioni hanno la forma generale `ld x1, offset(x2)` e `sd x1, offset(x2)` e calcolano un indirizzo di memoria sommando il contenuto del registro base (x2) al campo offset di 12 bit.
Se l'istruzione è una store il dato da memorizzare deve essere letto dal register file (dove risiede in x1), mentre se è una load il valore letto dalla memoria deve essere scritto all'interno del register file nel registro specificato (x1).
Per eseguire queste istruzioni occorrono sia il register file che la ALU.
Inoltre saranno necessarie anche una unità per l'**estensione del segno** del campo offset (da 12 a 64 bit) e un'unità di memoria dati in cui scrivere o da cui leggere il dato.



La memoria dati viene scritta dalle istruzioni di store, e quindi sarà dotata dei segnali di controllo sia di lettura sia di scrittura; riceverà in ingresso, inoltre, l'indirizzo e il dato che deve essere scritto.

Istruzioni di salto condizionato (beq)

La beq ha tre operandi: due registri il cui contenuto viene confrontato per determinare se è uguale, e un offset di 16 bit utilizzato per calcolare l'**indirizzo di destinazione del salto** (sommando il campo offset dell'istruzione, dopo averlo esteso a 32 bit con segno, al PC).

- L'indirizzo di base per il calcolo dell'indirizzo di salto è quello dell'istruzione di salto stessa.
- Il campo offset viene spostato di 1 bit a sinistra, in modo tale che l'offset non codifichi lo spiazzamento in numero di byte ma in numero di half word. Tale spostamento aumenta lo spazio di indirizzamento dell'offset di un fattore 2 rispetto alla codifica dello spiazzamento in byte.

Per gestire questa ultima complicazione è necessario far scorrere il campo offset di 1 bit a sinistra.
Oltre a calcolare l'indirizzo di destinazione del salto, bisogna determinare se l'istruzione da eseguire dopo sia quella nella posizione di memoria successiva oppure quella contenuta all'indirizzo di destinazione del salto. Quando la codifica del salto è vera (i due operandi sono uguali) l'indirizzo di destinazione del salto diventa il nuovo valore del PC e si parla di **salto condizionale eseguito** (*branch taken*); se il contenuto degli operandi è diverso, il valore del PC viene incrementato di 4 e diventa il valore corrente nel PC; in questo caso si parla di **salto condizionato non eseguito** (*branch not taken*).
Per il calcolo dell'indirizzo di destinazione si utilizzano una unità di estensione del segno e un sommatore.
Il confronto viene fatto dalla ALU.