Architetture degli Elaboratori - Sommario

Introduzione intensiva da 4h sulle architetture degli elaboratori

1. Rappresentazione delle Informazioni

Rappresentazione dell'Informazione nei Calcolatori

Rappresentazione dell'informazione nei calcolatori. Principio zero: l'esistenza di solo due simboli. Rappresentazione dei numeri, del testo e delle immagini in binario.

1. Principio zero (uno)

IL BINARIO. La *rappresentazione dell'informazione* nei *calcolatori* si basa su un *principio cardine*, ovvero che abbiamo *solo* due simboli: vengono (solitamente) denominati come $0, 1$.
Ci sono state alcune eccezioni, come con i *computer sovietici* che avevano tre simboli, ma questo è un dettaglio storico minore.
Quindi consegue che *ogni tipologia di informazione*, tra cui numeri, foto, testo, documenti, istruzioni, eccetera... vanno *rappresentati* con questi due soli simboli. Ora vedremo come sarà possibile superare questo apparente limite del binario.

2. La rappresentazione dei numeri

I BIT E I BYTE. Definiamo un *bit* come una *singola cifra binaria* che può solo assumere *uno dei valori* $0, 1$. Si definisce un *byte* come *otto bit*. Segue che ogni *byte* può rappresentare 2^8 numeri.
Dopodiché si può prendere i *multipli* del *byte*, tra cui il *kilobyte*, *megabyte*, eccetera...

Tuttavia è possibile ci sono due *tipi di multipli del byte*: uno è di base 10^n, l'altro è di base 2^{10n}. Dato che gli ordini di grandezza sono più o meno uguali, è possibile fare uno fraintendimento tra questi multipli.
Dunque indicheremo quelli con base 10^n con i prefissi *kilo*, *mega*, *giga*, *tera*, *peta*, ...; invece per quelli di base 2^{10n} vengono indicati con *kibi*, *mebi*, *gibi*, *tebi*, *pebi*.
(*Notare che queste sono semplici convenzioni, e non sono necessariamente rispettate da tutti!*)

I numeri Naturali

I NUMERI NATURALI. Ora vediamo come *"tradurre"* un *byte* (o più) in un numero naturale. Generalmente, supponendo che un numero binario è della forma

$$\langle x_1 x_2 \dots x_n \rangle$$

Allora possiamo ottenere il suo numero *in base decimale* facendo il seguente calcolo:

$$\sum_{i=1}^{n} x_i 2^{n-i}$$

Ovvero *"sommiamo le potenze di 2^n, a seconda della posizione delle cefire"*.
Si può usare altre basi, tra cui l'*ottale* e l'*esadecimale*.

OPERAZIONI TRA NUMERI IN BINARIO. Per sommare *due numeri in binario*, lo si fa come facciamo di solito in *decimale*, solo che abbiamo solo *due cifre* (quindi abbiamo più riporti del solito).
Ad esempio,

$$1101\ 1001 + 0010\ 0011 = 1111\ 1100$$

Similmente si può definire la *moltiplicazione di numeri in binario*.

OVERFLOW. Notiamo che con queste nozioni, abbiamo una limitazioni: queste operazioni (in somma) hanno un range possibile tra $[0, 2^{n-1}]$. Ma cosa succede se tentiamo di sommare *"due numeri troppo grandi"*? Ovvero se tentiamo di sommare ad esempio

$$1111\ 1111 + 1111\ 1111 = \ ?$$

Qui abbiamo infatti il cosiddetto *"overflow"*: tentando di sommare due numeri tali che la somma venga maggiore di 2^{n-1}, non abbiamo abbastanza bit per rappresentare il nuovo numero.
Dunque si fa la somma come di consueto, troncando via le parti *"in eccesso"*. In questo caso ho

$$\begin{aligned}&1111\ 1111+\\&1111\ 1111=\\&1111\ 1110\end{aligned}$$

Quindi paradossalmente stiamo *"sottraendo"* il primo numero con 1!

I numeri Interi

I NUMERI INTERI. Adesso, vogliamo rappresentare anche i *numeri negativi*, dal momento che potrebbero risultare utili per certi contesti, tra cui i debiti, conti negativi, eccetera...
Storicamente ci sono stati più approcci per formalizzare i *numeri negativi* nel *binario*, tra cui quello del *segno e modulo*, del *complemento-due* e dell'*eccesso N*.

SEGNO E MODULO. Per sostituire il segno $-$ del numero negativo, basta usare *uno dei bit* del numero in binario per rappresentare il segno. Ovvero, se ad esempio ho

$$1000\ 1011$$

questo non si legge più come $2^7+2^3+2^1+2^0$, bensì come $-\cdot(2^3+2^1+2^0)$.
Questo sembra un buon approccio, dal momento che ampliamo il *range dei numeri rappresentabili* in $[-2^{n-1}-1, 2^{n-1}-1]$. Tuttavia ci sono due problematiche:

- In questo modo abbiamo due rappresentazioni del numero zero: ovvero $1000\ 000$ e $0000\ 000$, così in un modo *"sprechiamo"* uno slot.
- La somma tra *numeri positivi e negativi* non è più definibile in una maniera intuitiva: bisogna vedere dei *casi specifici* per poter sommarli. Tutto sommato, questo approccio è quello *"naïve"*, dal momento che l'idea sottostante è molto semplice.

IL COMPLEMENTO 2. In questo caso il *primo bit* diventa -2^{n-1}, ovvero adesso il numero lo leggiamo come

$$1000\ 0000 \to -2^7$$

E il restante delle caselle vengono lette come di consueto. In questo modo abbiamo molti vantaggi:

- Il range dei numeri diventa $[-2^n, 2^{n-1} - 1]$
- Il primo bit indica comunque il *segno* (o la *"negatività"*) del numero
- Possiamo sommare numeri positivi e negativi normalmente

ECCESSO N. Qui si tratta di *"sostituire il zero con un numero negativo"*, ovvero di *"contare da un numero più basso"*. Ad esempio, impostando $N = 128$, leggiamo il numero 0 come -128. Allora si ha

$$1000\ 000 = 0$$
$$1111\ 111 = 127$$

I numeri razionali

I NUMERI RAZIONALI (O REALI). Adesso voglio rappresentare i numeri *razionali*, ovvero quelli con la *virgola*; oppure vogliamo rappresentare pure i *numeri reali*, anche se sarebbe realisticamente impossibile per la *densità dei razionali nei reali*: ci servirebbero una quantità infinita di bit!
Quindi è necessaria una *buona approssimazione* per questi numeri, dal momento che una *vera e propria fedele rappresentazione* sarebbe fisicamente impossibile.
Storicamente ci sono state molte convenzioni per rappresentare questi numeri, oggi si usa lo *standard IEEE 754*, che ha come *"idea di base"* quella di *"simulare"* la *notazione scientifica*: si ha un *bit per il segno*, dei *bit* per l'*esponente delle cifre significative* in 2^n che viene moltiplicata per la *mantissa*.
Questo standard ci offre più *"tipologie di numeri"* con certi *livelli di precisione*, tra cui i numeri a *precisione singola e doppia*.
Inoltre, questa convenzione ci offre una possibilità per definire l'*infinito* $\pm\infty$ e il risultato di un'operazione *indefinita* NaN.
Per una visualizzazione di questa convenzione, vedere il sito https://bartaz.github.io/ieee754-visualization/.

3. La rappresentazione del testo

IL TESTO. Per quanto riguarda invece il *testo*, storicamente ci sono state più convenzioni per *"trasformare"* i *bit* in caratteri.
Inizialmente c'è stata la convenzione *ASCII*, dove si usava un *byte* per

rappresentare un singolo carattere. Si nota che in realtà *sette bit* erano già sufficienti per rappresentare tutti i caratteri nell'alfabeto anglo-sassone (ovvero latino escludendo gli accenti), quindi c'era un *"bit in eccesso"*.
Da qui si hanno più *"varianti"* di *ASCII*, dove si usa il *bit in eccesso* per rappresentare altri caratteri, tra cui *lettere accentate*, *nuovi caratteri caratteristici della lingua*, eccetera...; il variante dipende dal *linguaggio di riferimento*.
Tuttavia questa *molteplicità* rappresenta un limite dell'ASCII, dal momento che in questo modo diventa *impossibile* scrivere documenti in *più linguaggi*.
Oppure con l'ASCII rimane comunque impossibile *rappresentare* la scrittura di certi linguaggi, come ad esempio quello del *giapponese*, *cinese*, ...
Quindi oggi si usa la convenzione *Unicode*, che non solo comprende *una buona parte dei linguaggi*, ma anche gli *emoticon*.

4. La rappresentazione delle immagini

LE IMMAGINI. Qui basta considerare la convenzione *RGB*, dove si usa una *terna* di *byte* per rappresentare l'*intensità della luce* per il colore *rosso*, *giallo* e *blu*.

2. Architetture Base dei Calcolatori

Architettura Base del Calcolatore

Architettura base (generalizzata) di un calcolatore. Esecuzione di un programma: linguaggio di programmazione ad alto livello, linguaggio assembly e codice macchina.

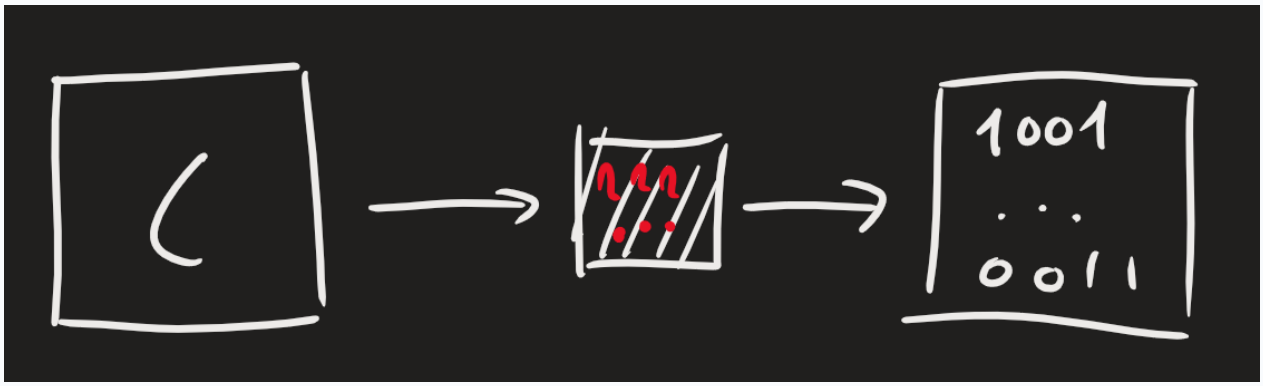
---

1. Il black box

IL BLACK BOX. Sapendo che *tutto dev'essere codificato nel linguaggio binario* nel calcolatore, vogliamo capire come il nostro *computer* è in grado di *eseguire le istruzioni*, che sono chiaramente scritte in un linguaggio *non binario* (come ad esempio *C* oppure *Python*).
Infatti quando vogliamo eseguire un programma, dobbiamo *convertire* il *codice*

sorgente del programma in un linguaggio che sia leggibile dalla macchina: tuttavia questo processo non è esplicitamente visibile, dal momento che possiamo vedere solo l'*input* e l'*output*. Quindi c'è questa specie di *"black box"* che converte il codice sorgente in codice macchina.
Il nostro obbiettivo è di chiarire questo *"black box"*.

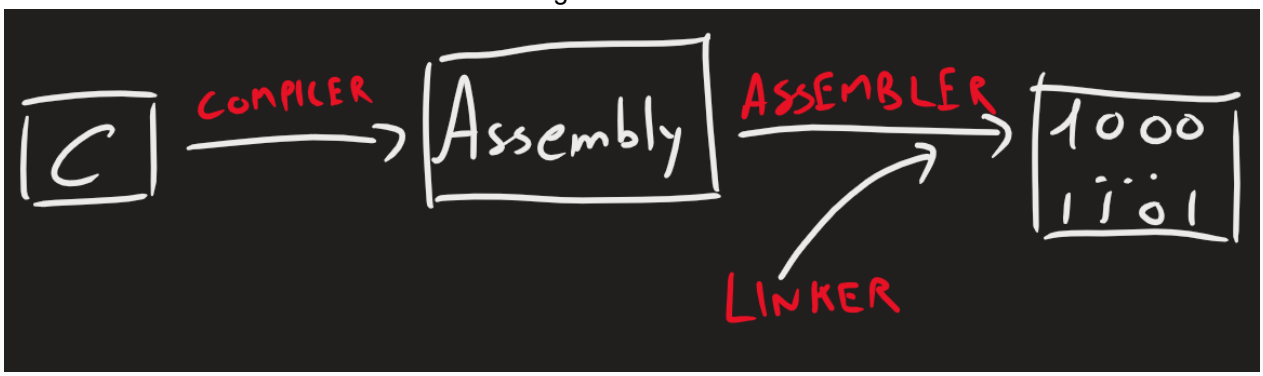
FIGURA 1.1. (*Il black box*)



2. Compilatore e Assembler

IL COMPILATORE. Ci ricordiamo che il linguaggio C è un linguaggio di programmazione che va *compilato*, ovvero trasformato in un linguaggio di programmazione che è più *"leggibile"* dal computer. Stiamo parlando del linguaggio *Assembly*, che è *poco astratto*.
Tuttavia, le istruzioni che caratterizzano questo linguaggio dipende dalla *singola architettura dell'elaboratore* (ovvero del *processore*). Infatti ci sono *molti* approcci diversi per il linguaggio *Assembly*. Quindi questo linguaggio si dice *non portatile*.
L'ASSEMBLER. Tuttavia il linguaggio *Assembly* non è sufficiente per essere compreso dalla macchina, dal momento che rimane ancora testo. Vogliamo quindi trasformare un'istruzione in una singola sequenza di cifre $0, 1$: su questo processo interviene l'*assembler* per trasformare le istruzioni in valori numerici (ovvero in *forma statica*) e interviene anche il *linker* per le *librerie*. Per vedere più dettagli su questo processo, vedere il sito https://godbolt.org/.

FIGURA 2.1. (*Schema generale della conversione di un programma*)



3. Architettura di Von-Neumann

Architettura di Von-Neumann

Caso specifico di un calcolatore generico: l'architettura di Von-Neumann. Struttura base di un calcolatore: CPU, Memoria primaria, periferiche e bus. Strutture specifiche di ogni singolo componente. Ciclo fetch-decode-execute della CPU.

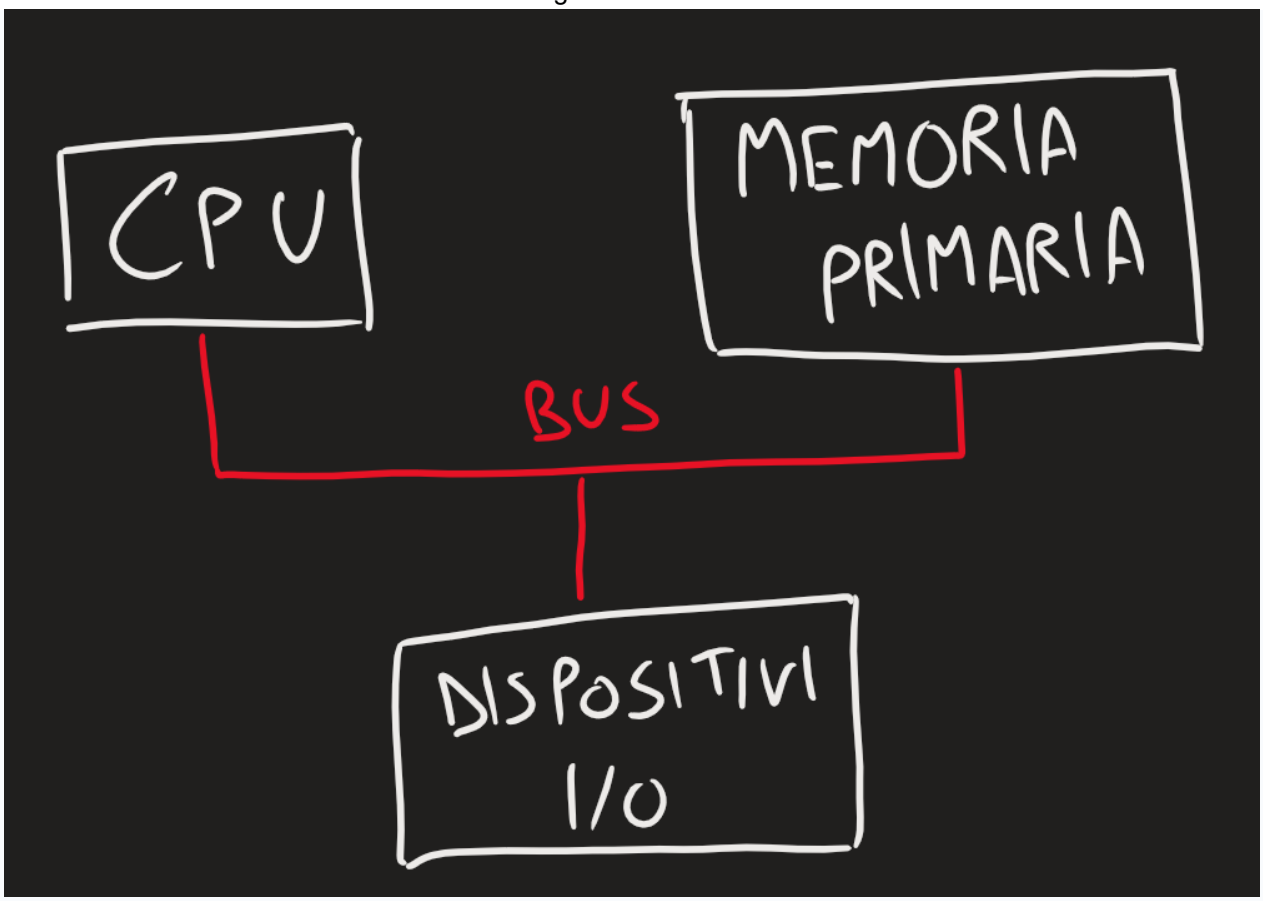
---

1. Layout di un Architettura di Von-Neumann

IL LAYOUT GENERALE. L'*architettura di Von-Neumann* è una *disposizione specifica* di un *calcolatore*, ed è formate principalmente da quattro componenti:

- La *CPU*, ovvero la *Central Processing Unit* che *esegue* le istruzioni
- La *memoria primaria*, della quale l'unicità è caratteristica dell'*architettura di Von-Neumann*; contiene sia *dati* che *istruzioni*
- Le *periferiche*, ovvero i *dispositivi I/O* che permettono l'interazione del calcolatore con l'*esterno*: comprendono la *memoria di massa*, *dispositivi di input e output* come la *tastiera* o il *monitor*
- Il *bus* che mette in *comunicazioni* le diverse componenti appena viste.

FIGURA 1.1. (*Architettura di Von-Neumann*)



2. Strutturazione specifica dei componenti

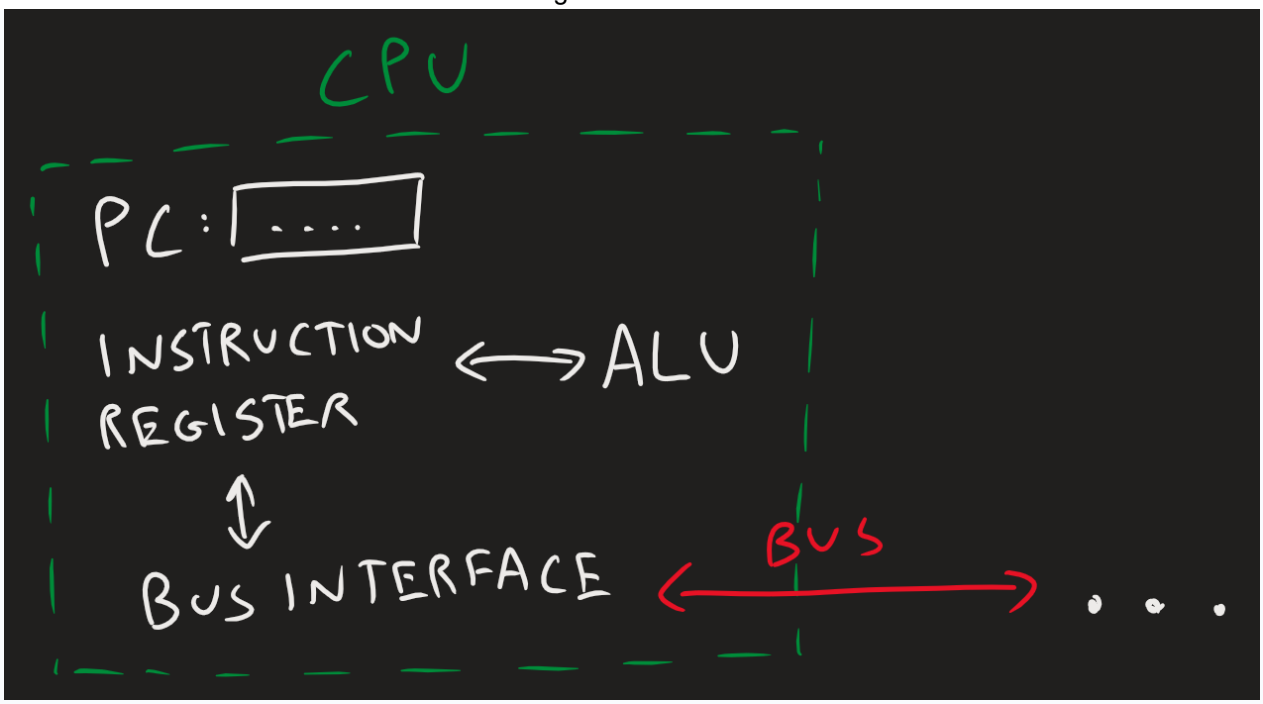
CPU. Specificamente la *CPU* è formata da altri componenti interni.

- Il *Program Counter*, abbreviato come *PC*: serve a *tenere conto* dell'*istruzione da eseguire* in un programma. In parole brevi, controlla il *flusso di esecuzione* di un programma.
- Il *Register file*, che tiene conto delle istruzioni.
- L'*ALU*, che esegue i singoli *calcoli aritmetici*.
- Il *Bus Interface*, che interfaccia il *register* con il *bus*.

MEMORIA PRIMARIA. La memoria primaria è organizzata in *modo lineare*, come uno *stack* di *locazioni di memoria* associati a degli *indirizzi di memoria*. Ogni *locazione di memoria* contiene un singolo *valore* numerico.

DISPOSITIVI I/O. I dispositivi *I/O* permettono di consentire la *comunicazione* del *calcolatore* col *mondo esterno*. Si può trattare ad esempio di *dati in arrivo*, che possono venire dalla *tastiera* o dal *mouse*; similmente si può parlare anche di *dati in uscita*, che sono visualizzabili col *monitor* o periferiche simili. Oppure si può trattare anche di un'*archiviazione larga di dati*, ovvero di *"mass storage"*, che può essere fatto con componenti come l'*HDD*, l'*SSD*, ...

FIGURA 2.1. (*Struttura particolare della CPU*)



3. Ciclo fetch-decode-execute della CPU

IL CICLO. La *CPU* segue delle istruzioni, e lo fa effettuando un ciclo di *tre operazioni*: fetch, decode e infinite execute.

FETCH. In questo stato il processore *incrementa* il *PC (program counter)*, ricava l'*istruzione* dalla *memoria primaria* e salva questa istruzione nell'*instruction register*.
DECODE. In questo processo la *CPU* interpreta il valore numerico in un'istruzione, il cui significato cambia al *variare dell'architettura*.
EXECUTE. In questo ultimo stato il processore esegue le operazioni contenute nell'*istruzione*, avvalendosi dell'*ALU*.

4. Architettura ARM

Architettura ARM

Esempio specifico-pratico di architettura Von-Neumann: ARM. Notizie storiche, introduzione generale, struttura di ARM-v7a, concetto di registri.

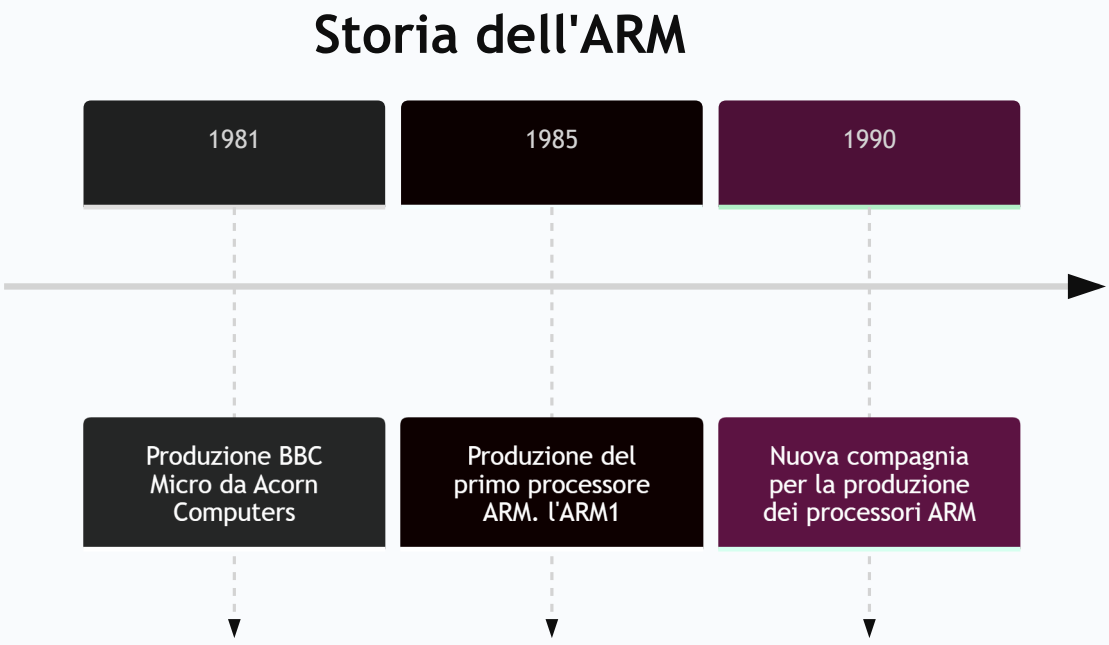
---

1. Introduzione Generale

UN ESEMPIO PRATICO. Finora abbiamo solo visto *strutture generali di architetture*: abbiamo capito come un *calcolatore* legge un *programma*, e una struttura *particolare* di un tale calcolatore. Adesso vediamo un esempio *pratico* di *un'architettura*: l'ARM, presente ancora oggi nei nostri dispositivi, tra cui cellulari, tablet, portatili, ...

NOTIZIE STORICHE. Il primo *processore* dell'*architettura ARM* venne prodotto nel *1985*, dalla *Acorn Computers*, una società britannica. Dopodiché la progettazione dei *processori* venne delegata ad una diversa compagnia, nel *1990*. Attualmente l'*ARM* non produce più i processori, ma li progetta e si limita a *distribuire i diritti di proprietà intellettuale*. Nel corso della storia, il significato di *ARM* è passado a *"Advanced Risc Machine"*. Ma che vuol dire *"Risc"*? Lo vedremo con i *paradigmi CISC-RISC* delle istruzioni (Paradigmi CISC e RISC).

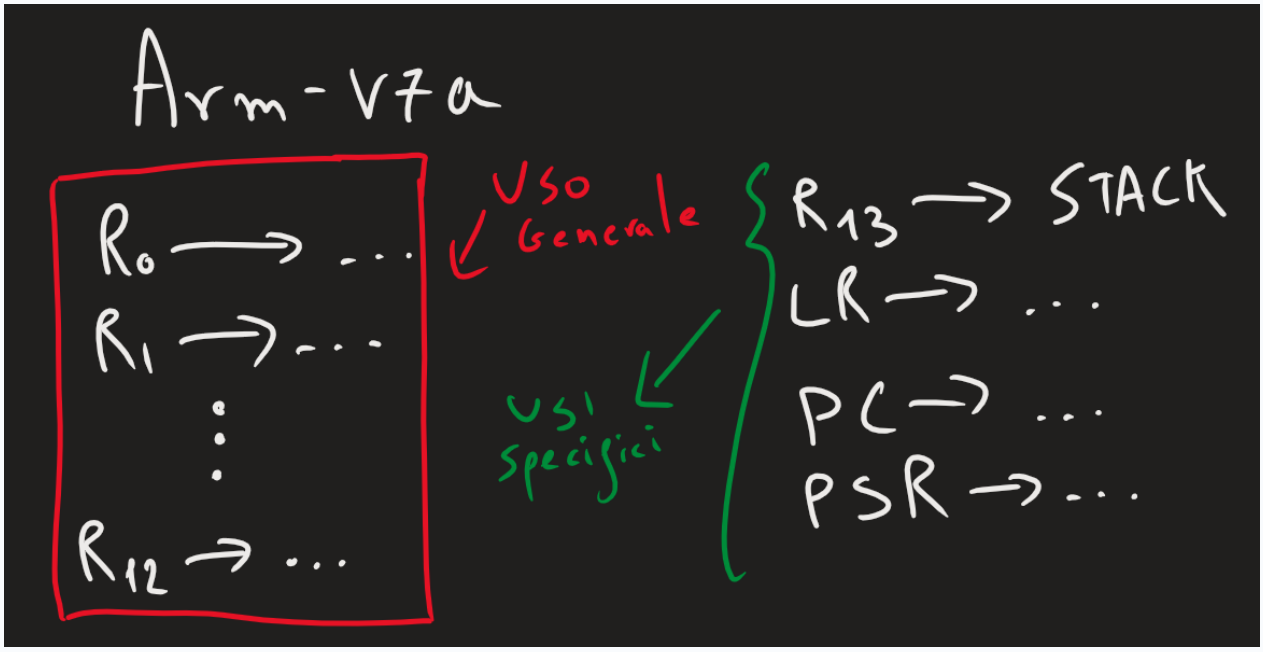
FIGURA 1.1. (*Linea temporale*)



2. Struttura di Arm-V7a

Arm-v7a. Ora vediamo una *versione specifica* dell'ARM. In questa versione dell'*ARM* abbiamo *tredici registri* per l'*uso generale*, ognuno dotato di *32 bit* (ovvero *4 byte*).
Dopodiché ci sono altri *quattro registri specifici*, tra cui *due* sono usati per *lo stack delle chiamate*, uno per il *program counter* e uno per il *program status register*.

FIGURA 2.1. (*Arm-v7a*)



5. Paradigmi CISC e RISC

Paradigmi CISC e RISC

Paradigmi CISC e RISC per le istruzioni del linguaggio Assembly. Filosofia generale degli approcci, svantaggi e vantaggi e spiegazione della necessità storica di due paradigmi diversi.

---

1. Introduzione Storica

INTRODUZIONE STORICA. Nel corso della storia sono stati sviluppati *due approcci* per codificare le *istruzioni Assembly*: una è il *CISC* (*Complex Instruction Set Computers*) e l'altra è *RISC* (*Reduced Instruction Set Computers*). La più tradizionale è la *CISC*, con le architetture Intel negli anni '70, e la più nuova e la *RISC*. Nei paragrafi successivi si vedrà il motivo.

2. RISC e CISC

CISC. Il *primo paradigma* è il *CISC*, da cui il nome ci dice che si vuole svolgere *tante operazioni complesse*: questo semplifica tantissimo la vita dei *programmatori Assembly*, dato che devono scrivere meno istruzioni! Tuttavia,

con questo approccio si ha un *tempo di esecuzione* più lento (in particolare le istruzioni richiedono più di un *ciclo di clock*) dato che le istruzioni diventano *complesse* da *decodificare*. Questo è stato l'approccio preferito dall'inizio, dato che il *compilatore* non esisteva, oppure non era ancora abbastanza *"affidabile"*.
RISC. Con l'*avvento dei compilatori*, si vuole iniziare a semplificare tutto: tanto con il linguaggio di programmazione C, non bisognava trattare più con *registri*! Con questo approccio si vuole dare un *"set"* di istruzioni *poche* ma *semplici*, e di solito ogni istruzione ha un suo solo compito preciso. Nell'avvenire del tempo, l'approccio *RISC* è diventato più popolare dal momento che è *più veloce* e che non serviva più *programmare in Assembly*.

6. Linguaggio Assembly

Linguaggio Assembly

Funzionamento base del linguaggio Assembly: codifica delle istruzioni, program status register. Alcune istruzioni ARM del linguaggio Assembly. Istruzione MOV, ADD, SUB, B, BL, CMP, BEQ, LOAD, STORE. Chiamate di funzione con Assembly.

---

1. Codifica delle Istruzioni

LA CODIFICA DELLE ISTRUZIONI IN ASSEMBLY. Come detto prima, ogni informazione dev'essere rappresentata come un numero (in particolare binario). Questo vale anche per le istruzioni del linguaggio Assembly (in particolare v7a): ogni istruzione occupa esattamente *4 byte*. Ora vediamo i *"ruoli"* di questi bit individuali

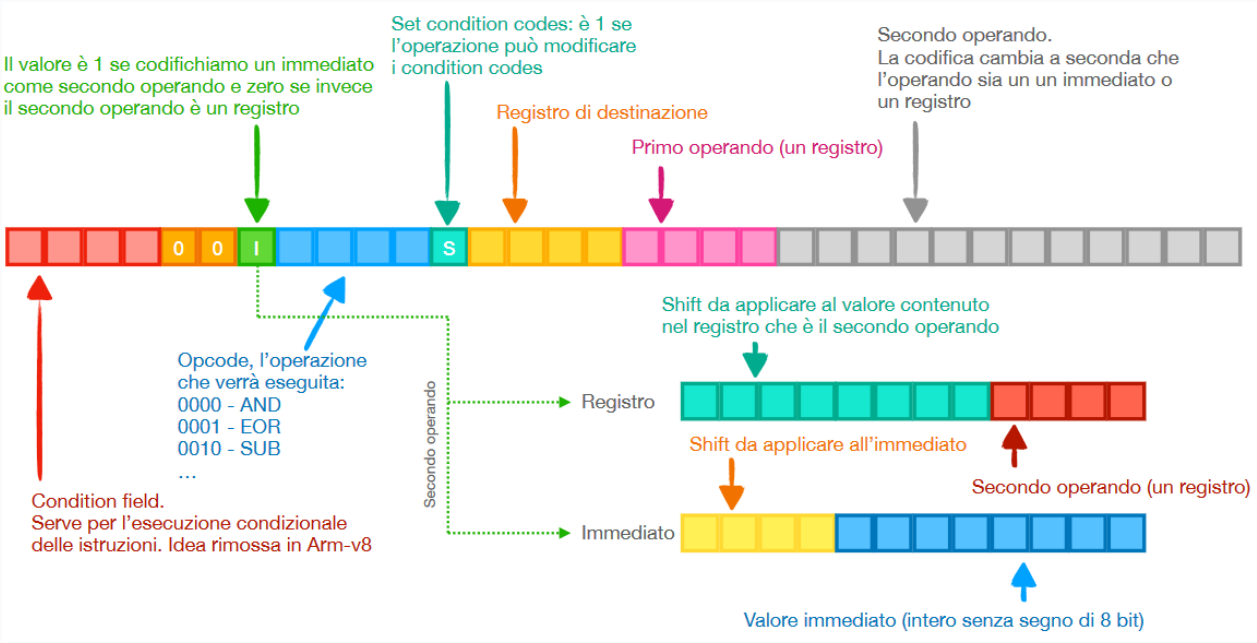
BIT 1-6. Sono dei bit di cui possiamo non occuparcene, dato che si trattano di dettagli.
BIT 7. (*Segno*) Questo bit è una specie di *"flag"* che ci dice se questa *"stringa"* binaria di numeri tratta il *secondo operando* come un *registro* o un *immediato*.
BIT 8-11. (*Opcode*) Indica l'operazione da eseguire
BIT 12. (*Set condition codes*) Dettagli.
BIT 13-16. (*Registro di destinazione*) Indirizzo del registro in cui salvare il

risultato del calcolo.

BIT 17-20. (*Primo operando*) Sempre un registro

IL RESTO DEI BIT. (*Secondo operando*)

FIGURA 1.1. (*Schema riassuntivo*)



2. Program Status Register

L'IDEA. Il program status register è utile per *tenere conto* di operazioni di *comparazione*. I bit di questa componente verrà controllata dai comandi *branch condizionali*. In particolare tengono conto di quattro risultati: *negative result*, *zero result*, *carry set* e *overflow*.

3. Istruzioni Assembly

MOV. L'istruzione `MOV` permette di copiare un *valore* da un *registro* a un *altro registro* (oppure inserire un valore numerico in un registro). Viene scritta come `MOV dest, source`.

ADD. L'istruzione `ADD` permette di *sommare due valori* contenuti in due registri, o di *sommare un valore di un registro* con un *valore numerico*. Viene scritta come `ADD dest, op1, op2`.

SUB. L'istruzione `SUB` è analoga all'istruzione `ADD`, solo che al posto di *aggiungere numeri* si *sottrae* dei numeri.

BRANCH. L'istruzione branch `B` ci permette di saltare ad un'altra istruzione nel codice, che viene contrassegnata con un *label*. Viene scritta come `B label`. Notare che con *sola* questa istruzione, si può avere solo *cicli infiniti*.

CMP. L'istruzione `CMP` ci permette di comparare *due numeri* contenuti in *due registri* (o un numero di un registro con un valore numerico). Dal momento che *non serve* di nessuna destinazione di scrittura (infatti l'esito viene automaticamente salvato nel *PSR*), viene scritta solamente come `CMP op1 op2`.

BEQ. L'istruzione `BEQ` è *una delle istruzioni* di *branch condizionale*, ovvero del *branch* effettuata solo in certi casi. In questo caso, `BEQ` effettua un branch solo nel caso dell'uguaglianza di due valori precedentemente comparati con l'istruzione `CMP`.
Di altri tipi ce ne sono `BNE` che verifica la *disuguaglianza* di due valori.

LOAD. L'istruzione `LDR` permette al calcolatore di *accedere* alla *memoria del lavoro*. Dato che gli *indirizzi della memoria del lavoro* sono diversi da quelli dei *registri* (infatti se fossi così, avremmo solo 2^{24} locazioni di memoria che equivalgono a circa *16 MB*), l'indirizzo della memoria viene indicata mediante un *registro che contiene l'indirizzo da cui prendere il valore*. Ovvero si ha `LDR dest, [addr]`.

STORE. Analogamente al comando `LDR`, il comando `STR` ci permette di *salvare* un valore numerico di un registro in una locazione di memoria. Si ha quindi `STR src, [addr]`.

4. Chiamate di funzioni

LO STACK. Per effettuare una *"chiamata di funzione"* in Assembly, bisogna prima decidere le convenzioni per *"tradurre"* le seguenti:

- Come passare gli argomenti
- Come salvare lo stato prima della chiamata della funzione
- Come salvare lo stato della funzione
- Come ritornare il valore

Per far ciò usiamo una struttura particolare chiamata *"stack"*. Questa struttura di indirizzi inizia con un indirizzo di memoria fissato, noto come *"top of the stack"*; dopodiché il registro *R13* indica la *prima posizione libera sullo stack*.

5. Esempio

ESERCIZIO. Effettuare una specie di *"conversione"* dal C in Assembly:

```c
int x = 5;
int y = 10;
while (y>0)
{
      x = x + y + 3;
      y = y - 1;
}
```

SOLUZIONE.

```
MOV R0, #5
MOV R1, #10

loop:

CMP R1, 1
BEQ end
ADD R0, R0, R1
ADD R0, R0, #3
SUB R1, R1, #1
B loop

end:
```

7. Polling e Interrupt

Comunicazione Architettura e l'Esterno

Approcci PMIO, MMIO per la comunicazione tra architettura e strutture I/O. Metodi polling e interrupt per gestire la comunicazione tra CPU e I/O. Direct Memory Access per trasferimento di grandi quantità di dati.

1. Le questioni fondamentali

LE DOMANDE. Per poter aver un'idea di come poter far *comunicare* il nostro *calcolatore* con le strutture *I/O esterne*, dobbiamo porci le seguenti domande.

- Cosa succede se vogliamo comunicare con altri dispositivi? Come facciamo ad inviare e ricevere informazioni? Dove li salviamo? Come gestiamo la comunicazione asincrona?
- Come possiamo accedere ai dispositivi di memorizzazione di massa?

2. Collegamento delle informazioni

PMIO. Una prima idea per poter approcciarsi alla prima domanda è di usare l'approccio *"Port Mapped I/O"*, dove semplicemente creiamo delle *istruzioni speciali* per comunicare con dei dispositivi input-output. In particolare, sono delle istruzioni simili a dei *load* e *store*. Da notare che gli *indirizzi dei dispositivi I/O* sono diversi da quelli del *calcolatore*.
Quindi, in questo caso *separiamo* la *memoria di lavoro* dai *dispositivi I/O*.

MMIO. Un'idea più aderente ad un approccio *RISC* è quello di usare *istruzioni già esistenti*: quindi dedichiamo alcuni *istruzioni già esistenti* della memoria per *operazioni di I/O*. In particolare servirà un *componente* che sarà in grado di *decodificare* se un *indirizzo* è dedicato al *calcolatore* o al *dispositivo I/O*.

3. Esecuzione delle Operazioni I/O

LA SECONDA DOMANDA. Ora, per risolvere la seconda domanda, ovvero *come gestire la comunicazione effettiva* tra il calcolatore e l'I/O, abbiamo due approcci per gestire casi in cui *i dispositivi* lavorano con *tempi diversi*, ovvero quando un dispositivo è più veloce dell'altro.
I due metodi principali sono il *polling* e l'*interrupt.*

POLLING. L'idea del *polling* è quella di controllare ad *intervalli regolari* se il dispositivo *I/O* sia pronto o meno. Il svantaggio di questo approccio è il *dispendio del tempo*, dedicato per fare il *polling*; oppure in certi casi, è stato effettuato in un tempo troppo tardivo.

INTERRUPT. Una seconda idea è quello di *"avvisare"* il *CPU* quando un dispositivo è pronto. Questo avviene tramite *supporto hardware*, che

"interrompe" il processore forzandolo ad un pezzo di codice per gestire il dispositivo I/O.

4. Gestione di Grande Quantità di Dati

L'ULTIMA DOMANDA. Per gestire invece un *trasferimento di grandi quantità di dati*, gli approcci di *polling* e di *interrupt* sono inefficaci, dal momento che da un lato spostiamo sempre dati e dall'altro il processore viene continuamente interrotto. Allora l'idea è quello di *istituire un specifico* fatto apposta per questo caso, ovvero la *direct memory access* (DMA). Questo permette di copiare i dati dal dispositivo I/O alla memoria di lavoro, senza dover interrompere la *CPU*.