Corso di Informatica: Parte 1

Anno 2004-2005

Prof. Gilberto Filè

Architettura, Hardware e Software di un computer

Contenuto:

1 Introduzione

I calcolatori elettronici o computer furono sviluppati durante gli anni 40 negli Stati Uniti. Allora come adesso questi calcolatori erano costruiti assemblando un grande numero di componenti elettronici in modo da poter memorizzare informazioni ed eseguire programmi per manipolare le informazioni in modo da ottenere i risultati desiderati. Naturalmente, oggi le componenti sono diventate molto più piccole e conseguentemente i computer moderni sono enormemente più piccoli e più potenti dei loro antenati degli anni '40. Queste componenti elettroniche sono in grado di esprimere 2 condizioni diverse: presenza o assenza di corrente e convenzionalmente si associa il valore 1 ad uno dei 2 stati e 0 all'altro. Quindi tutte le informazioni che risiedono su un calcolatore sono codificate con sequenze di 1 e di 0, insomma con numeri in base 2 o *binari*.

Per poter essere eseguiti da un computer i programmi devono risiedere nella memoria del computer stesso, esattamente come i dati che il programma manipola. Quindi i programmi devono essere anch'essi codificati da numeri binari. Agli albori della programmazione, i programmi erano scritti direttamente in binario. Ovviamente questo era un lavoro molto arduo ed inoltre i programmi codificati in binario erano molto difficili sia da comprendere che da correggere.

Da quegli anni ad oggi si è fatto uno sforzo enorme per disegnare linguaggi di programmazione che permettessero ai programmatori di essere sempre piu'

svincolati dal computer usato, in modo da potersi concentrare sulla soluzione corretta dei loro problemi piuttosto che sulla sua codifica in binario. Nel seguito del corso vedremo il linguaggio di programmazione C++. Questo linguaggio è stato disegnato a partire dagli ultimi anni '80 ed oggi è uno dei linguaggi più evoluti e più usati al mondo. In effetti C++, come altri linguaggi simili, non solo rende il programmatore (il più possibile) indipendente dal computer che utilizza, ma gli mette a disposizione dei costrutti che guidano e facilitano il suo compito di analisi e soluzione dei problemi affrontati. Comunque, prima di studiare aspetti avanzati come il linguaggio C++, sarà utile capire il legame che comunque esiste tra i linguaggi di alto livello come il C++ ed il linguaggio macchina citato prima.

2 L'architettura di Von Neumann

Oggi come negli anni 40 un computer è costituito da 4 componenti principali: l'unità centrale di calcolo (CPU), la memoria principale o RAM, la memoria secondaria ed i dispositivi di ingresso uscita (input/output). Queste 4 componenti sono collegate da un insieme di connessioni, chiamato *bus*, che permette il passaggio di informazioni da una all'altra. Questo schema è noto come *architettura di Von Neumann*, dal nome del suo inventore. La Fig. 1 rappresenta questo schema. Nonostante che i computer di oggi continuino in genere a seguire l'architettura di Von Neumann, l'enorme sviluppo tecnologico che si è prodotto dagli anni 40 ad oggi permette di costruire computer di piccole dimensioni e con prestazioni impensabili solo qualche anno fa.

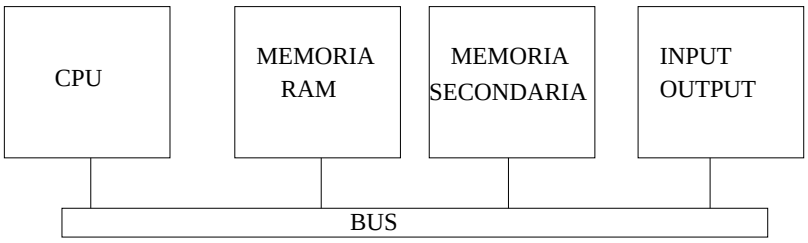


Fig.1 Architettura di Von Neumann

Il compito delle 4 componenti principali è il seguente:

- **La memoria RAM:** RAM sta per Random Access Memory, memoria ad accesso diretto in italiano. Spiegheremo tra poco il significato di questi termini. La memoria RAM serve a contenere informazioni che, come già detto, sono numeri binari.

  La RAM è costituita da una sequenza di **bytes**, dove ogni byte è costituito da 8 bits ed un **bit** rappresenta uno 0 oppure un 1. Ogni byte della memoria ha un indirizzo, da 0 ad un certo $N-1$, dove N è il numero

totale di bytes della memoria. Una sequenza di 2 o 4 bytes (a seconda del computer) contigui nella RAM forma una **parola di memoria** (word).

La memoria (RAM o secondaria) si misura in bit o byte. 2^{10} è 1024, cioè circa 1000, ed quindi 2^{10} bits viene chiamato un Kbit (K=Kilo). Un Kbyte è 2^{10} bytes. Analogamente 2^{20} è vicino ad un milione e quindi un tal numero di bits/bytes forma un Mbit/Mbyte (M=Mega). 2^{30} è circa 1 miliardo ed un tale numero di bits/bytes si indica con 1 Gbit/Gbytes (G=Giga). Infine, 2^{40} è un Tera e quindi avremo 1 Tbit/Tbyte.

La RAM serve ad immagazzinare informazioni (in binario) che potranno essere "scritte" nella RAM oppure "lette" dalla RAM (per essere manipolate). Queste operazioni vengono genericamente chiamate **accessi** alla RAM. La minima quantit di in formazione accessibile è 1 byte. Ogni byte è identificato dal suo indirizzo e viene acceduto tramite esso. La RAM è fatta in modo tale che il tempo necessario per accedere un qualunque byte è costante, cioè non dipende né dall'indizzo del byte da accedere, né da quello del byte acceduto subito prima. Random Access e accesso diretto stanno ad indicare questa caratteristica. Vedremo nel seguito che questo non è vero per la memoria secondaria.

Il tempo di accesso alla RAM è in genere dell'ordine di $10^{-7}, 10^{-8}$ secondi. A causa della minore rapidità di funzionalmento della RAM rispetto alla CPU, i computer moderni dispongono di una **memoria cache** più rapida della RAM e che in ogni momento contiene le informazioni che hanno maggiore probabilità di essere richieste dalla CPU. Ci possono essere vari livelli di memoria cache. La memoria cache ha dimensioni molto minori della RAM a causa del suo costo molto maggiore.

Una caratteristica importante della RAM è che essa è **volatile**, cioè, allo spegnimento del computer, le informazioni in essa contenute vanno perse. Ovviamente non è possibile che tutto vada perso ad ogni spegnimento e quindi, i computer hanno bisogno di memoria secondaria permanente per mantenere tutti i file importanti ed anche di avere una memoria simile alla RAM, ma permanente, su cui risiedono alcuni brevi programmi che "fanno partire" il computer nella fase di avvio. Questa fase è chiamata **bootstrap**. La memoria permanente che contiene questi programmi di avvio, è detta **ROM** (per Read Only Memory=memoria a sola lettura) di dimensioni molto ridotte rispetto alla RAM (pochi Kbytes). Il fatto che la ROM sia "read-only" serve a proteggere i programmi d'avvio da eventuali manomissioni.

- **CPU:** la CPU (Central Processing Unit) esegue programmi scritti in un linguaggio molto semplice (detto **linguaggio macchina**) ripetendo continuamente la seguente sequenza di operazioni, chiamato *ciclo ADE* (ciclo accesso-decodifica-esecuzione):

  1. accedere la memoria per leggervi la prossima istruzione del programma;

2. riconoscere di quale istruzione si tratti (tra quelle possibili).
3. eseguirla.

Per eseguire il ciclo ADE la CPU contiene una unità artitmetico-logica (UAL), dei registri ed una componente C_{is} per ogni possibile istruzione macchina is (ce ne sono poche). Ovviamente C_{is} si occupa di eseguire l'istruzione macchina is. Lo schema di una CPU è rappresentato in Fig.2.

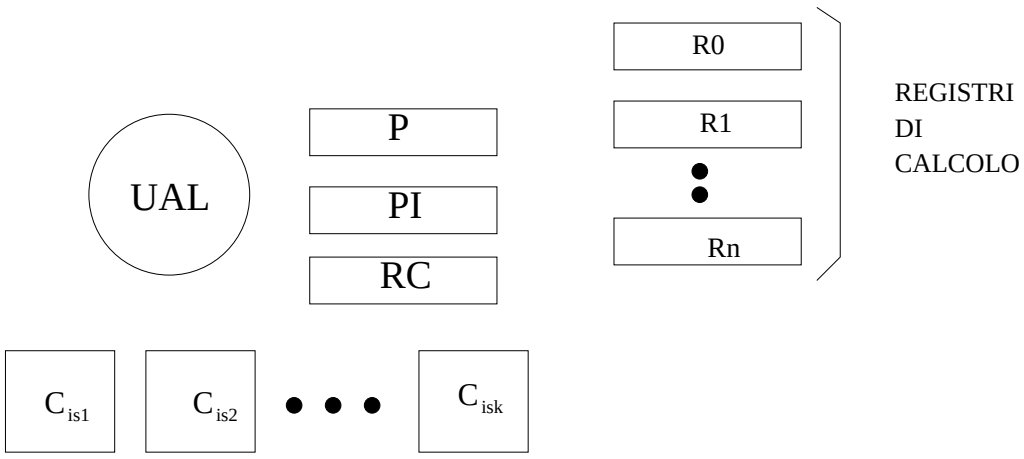


Fig.2 Schema della CPU

La maggior parte dei registri serve a fare i calcoli (contenere operandi e risultati parziali dei calcoli), ma ce ne sono due con un compito particolare. Innanzitutto il registro P che contiene l'indirizzo in memoria (RAM) della prossima istruzione macchina da eseguire. Durante il ciclo ADE, questa istruzione macchina viene messa nel registro PI in modo che la UAL la riconosca e, se si tratta dell'istruzione macchina is, la UAL attiva la corrispondente componente C_{is} che esegue l'istruzione is richiesta.

Questa esecuzione comporta in generale uno *scambio di informazioni* tra i registri della CPU e la RAM e la modifica del registro P. In generale nel registro P viene copiato l'indirizzo dell'istruzione macchina successiva a quella appena eseguita (un programma macchina è una sequenza di istruzioni macchina) a meno che l'istruzione appena eseguita non richieda esplicitamente di continuare l'esecuzione con un'altra istruzione (specificando l'indirizzo RAM di questa istruzione che verrà copiato in P). In questo caso si parla di **istruzioni di salto**. Queste istruzioni di salto usano uno speciale registro di calcolo RC il cui uso verrà illustrato nella Sezione 6.

Il tempo necessario per eseguire un ciclo ADE è nei moderni computer circa 10^{-9} secondi (nanosec). Questo tempo viene detto un **clock**. L'unità

di misura della velocità delle CPU è l'hertz, cioè i clock per secondo. Le CPU moderne si aggirano intorno ai 4 GHz. Questo significa che esse possono eseguire qualche miliardo di cicli ADE al secondo. Osservando che un ciclo ADE generalmente deve accedere la RAM e che un tale accesso richiede 10^{-7} secondi, si capisce immediatamente che la velocità della CPU rischia di restare imprigionata nel collo di bottiglia della RAM. La memoria cache è stata inventata per risolvere questo problema. Essa può essere acceduta con tempi simili alla CPU e viene gestita in modo che "contenga in anticipo" le informazioni che la CPU richiede di accedere nella RAM. Questo risultato che potrebbe sembrare "miracoloso" è ottenuto seguendo un'idea molto semplice: la cache mantiene le informazioni che la CPU ha usato più recentemente.

- **Memoria secondaria:** questa memoria è meno costosa della RAM e sopratutto è permanente. La memoria secondaria è costituita generalmente da dischi, nastri magnetici e lettori CD. Nella memoria secondaria verrà quindi memorizzato tutto quanto si vuole conservare nel computer anche dopo il suo spegnimento.

  La memoria secondaria è collegata alla RAM: leggere informazioni dalla memoria secondaria significa trasferirle nella RAM e viceversa scrivere informazioni nella memoria secondaria significa trasferirle dalla RAM nella memoria secondaria. La lettura di programmi e dati dalla memoria secondaria alla RAM è mecessario quando si vuole elaborare o eseguire questi oggetti. Dati e programmi vengono trasferiti dalla RAM alla memoria secondaria quando li si vuole rendere permanenti.

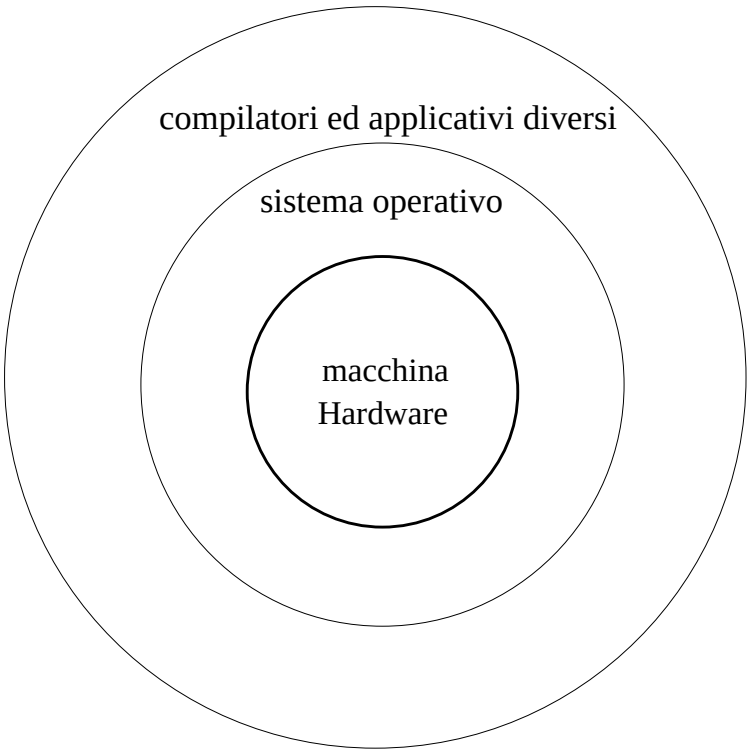
  Sulle memorie secondarie i dati vengono letti/scritti da testine di lettura/scrittura che devono quindi essere ogni volta posizionate all'inizio dell'informazione da leggere/scrivere. Questa operazione può avere una durata dell'ordine dei millisecondi e soprattutto essa dipende da dove si trova la testina di lettura subito prima dell'operazione, cioè dipende dalla lettura fatta in precedenza (al contrario della RAM). Quindi la memoria secondaria viene usata per lettura e scrittura di grosse quantità di informazioni posizionate in modo contiguo sul disco o nastro, ecco perchè questa memoria viene detta **sequenziale**.

- **input/output:** si tratta di dispositivi attraverso i quali è possibile far pervenire dati al computer dal mondo esterno (input) e, viceversa, far uscire i risultati dal computer al mondo esterno (output). Nei moderni PC, l'input è costituito dalla tastiera, mentre l'output è il video. Altri dispositivi di input/output diffusi sono le stampanti, i modem, le schede di rete. La velocità dell'output è dell'ordine dei decimi di secondo, mentre quella dell'input è dell'ordine dei secondi (in effetti dipende dai tempi di reazione dell'utilizzatore del programma).

3 Macchina Hardware e Macchina Software

Nei primi anni 40 il computer era essenzialmente una CPU collegata ad una piccola RAM. Quindi ogni operazione si volesse eseguire con esso, richiedeva che il programmatore la specificasse come sequenza di istruzioni macchina della CPU e che si occupasse di scrivere questa sequenza in binario direttamente nella RAM del computer.

Ovviamente questo limitava molto l'uso dei computer. Quindi già a partire da quegli anni, questo semplice modello di computer è stato arricchito con programmi che risiedono in modo permanente nel computer stesso e che facilitano il suo uso. Il principale di questi programmi si chiama **sistema operativo**. Il sistema operativo coordina tra loro le varie parti della macchina ed offre agli utilizzatori delle operazioni che consentono di inserire facilmente nel computer dati e programmi. In un computer moderno dobbiamo quindi distinguere tra la **macchina hardware** (cioè quella costituita dai puri circuiti elettronici) e la **macchina software** che consiste del sistema operativo e di altri programmi applicativi che consentono un uso semplice del computer stesso. In effetti si possono distinguere vari strati interdipendenti di software. Quindi un computer moderno viene rappresentato con il noto schema *a cipolla* della Figura seguente.



Deve comunque essere chiaro che le sole istruzione che la CPU di un computer può eseguire sono quelle del suo linguaggio macchina. D'altra parte, sarebbe

un compito troppo difficile scrivere programmi in linguaggio macchina. Per risolvere questo problema sono stati definiti dei linguaggi di programmazione ad alto livello (C++ è uno dei più recenti, ma ne esistono centinaia) che consentono ai programmatori di descrivere in modo più semplice le operazioni che vogliono eseguire. Una volta che questi programmi sono finiti, essi vengono tradotti in linguaggio macchina da appositi programmi di traduzione che si chiamano **compilatori**. Naturalmente per ogni linguaggio di programmazione ci sarà uno specifico compilatore. I compilatori sono una parte del software presente in un moderno computer. Insomma in questo modo, un programmatore può scrivere un programma (che risolve il suo problema specifico) in un qualsiasi linguaggio di programmazione (per esempio in C++), contando sul fatto che esso verrà tradotto in istruzioni macchina dall'apposito compilatore.

In pratica il sistema operativo risiede in piccola parte nella ROM del computer ed in massima parte nella memoria secondaria. Quando il computer viene avviato, i programmi che risiedono nella ROM si occupano di trasferire dalla memoria secondaria alla RAM le altre parti utili del sistema operativo. Essendo la RAM una risorsa in generale scarsa, questo processo è dinamico: nella RAM ci saranno sempre solo i programmi utili in quel momento. Quando serve un altro programma, esso viene caricato dalla memoria secondaria (se necessario, prende il posto di programmi diventati temporaneamente inutili). I compilatori risiedono nella memoria secondaria e vengono caricati in RAM quando l'utilizzatore del computer chiede di tradurre un suo programma.

Un'altro compito del sistema operativo è il seguente. Abbiamo visto che la velocità delle diverse componenti di un computer è estremamente diversa. Sarebbe quindi uno spreco inutile della CPU lasciare che un programma che richiede dati da memorie secondarie o input, mantenga il possesso della CPU stessa. Quello che avviene invece è che in questi casi, il programma viene messo in attesa e la CPU è usata per eseguire un altro programma. Quando le operazioni lente sono concluse, allora il vecchio programma torna a chiedere la CPU e la riceve per esempio quando il nuovo programma a sua volta chiede operazioni lente oppure quando qualche altra condizione è verificata. Questa tecnica di usare la CPU per eseguire molti programmi a turno (attenzione però: la CPU ne esegue sempre solo uno solo alla volta) si chiama **multi-programmazione**. La multiprogrammazione necessita dell'intervento del sistema operativo. Altre funzionalità del sistema operativo verranno studiate nella seconda parte del corso.

4 L'hardware di un computer

Hardware (parte dura) sta ad indicare l'insieme dei circuiti elettronici che costituiscono un computer. Essi sono ottenuti assemblando un grandissimo numero di componenti molto semplici detti *porte*.

Ci sono solamente tre tipi di porte:

AND questa porta compie l'operazione di E logico, cioè dati due input A e B che possono avere valore 0 o 1 restituisce 1 se e solo se (sse) entrambi A e

B sono 1. Una porta AND è raffigurata in Fig.3(a) dove il comportamento appena descritto è rappresentato da una tabella detta *di verità*.

OR questa porta compie l'O logico dei suoi input A e B, vedi Fig.3(b).

NOT complementa il suo unico input, vedi Fig.3(c).

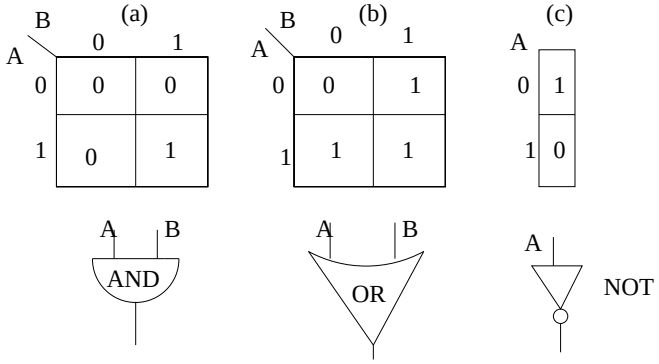
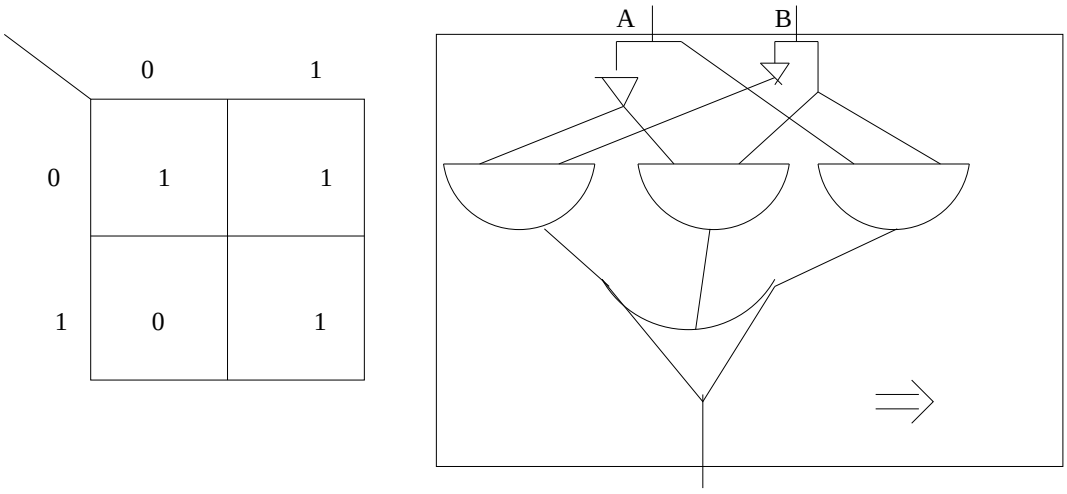


Figure 3. Le porte AND, OR e NOT

Componendo queste tre porte possiamo ottenere circuiti che eseguono operazioni anche molto complesse. Nel seguito consideriamo alcuni esempi semplici, ma significativi. In primo luogo esamineremo esempi di logica ed iniziamo costruendo un circuito che realizza l'implicazione logica.



Il significato di A *implica* B (in formula $A \Rightarrow B$) è che se A è vero allora anche B lo deve essere, mentre se A è falso allora qualunque sia il valore di B, la formula è vera. Per esempio, nella frase *se piove ci sono le nuvole* (in

formula *piove* $\Rightarrow$ *ci sono le nuvole*), vogliamo dire che se piove ci sono le nuvole, ma se non piove potrebbero esserci oppure no. Quindi, in questo caso, l'affermazione è vera (banalmente). L'operatore di implicazione è rappresentato dalla tabella di verità contenuto nella Figura precedente. Da questa tabella è facile costruire il circuito che si trova alla destra della tabella e che realizza la corrispondente la tabella stessa. Questo circuito lo abbiamo costruito seguendo il seguente **metodo generale**:
Per ogni 1 della tabella, siano a e b i corrispodenti valori di A e B (a e b sono o 0 o 1), il circuito contiene un AND i cui input sono A se $a = 1$ *e* $NOT(A)$ *se* $a = 0$ *e analogamente per B. I risultati di tutti gli AND prodotti in questo modo, entrano in un OR che da il risultato finale del circuito.*

È facile rendersi conto che questo circuito realizza sempre la tabella di partenza. Si osservi inoltre che generalizziamo l' OR ad avere 3 input, ma è facile realizzare un tale OR mettendo a cascata 2 OR normali.

Supponiamo ora di voler definire l'operatore logico di equivalenza. Si dice che A è equivalente a B se quando è vero A lo è anche B e viceversa (in formula $A \Leftrightarrow B$). Da questo è facile vedere che la tabella di verità di questo operatore è quella di Figura 4 che riporta anche il corrispondente circuito ottenuto applicando il metodo appena enunciato alla tabella stessa.

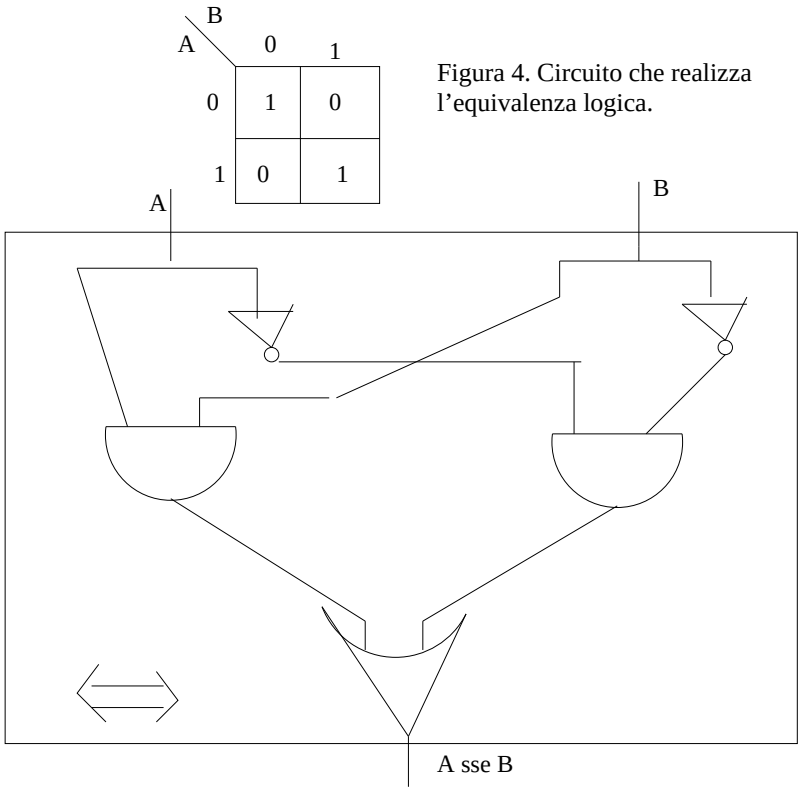
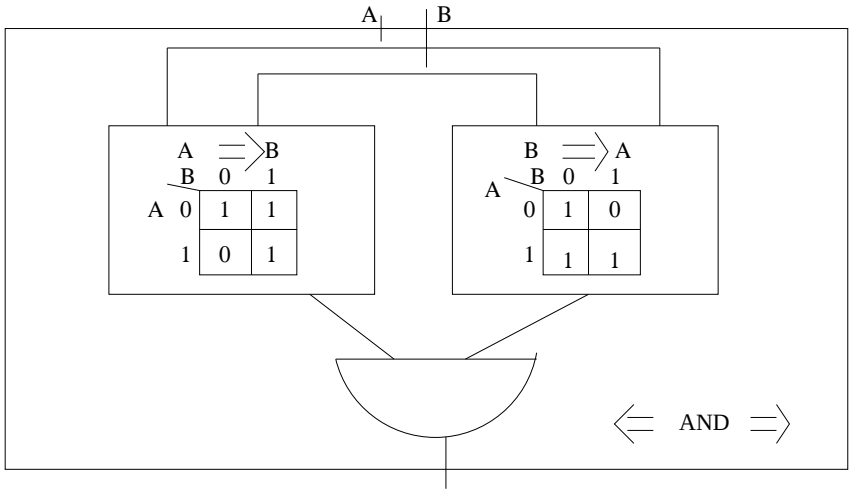


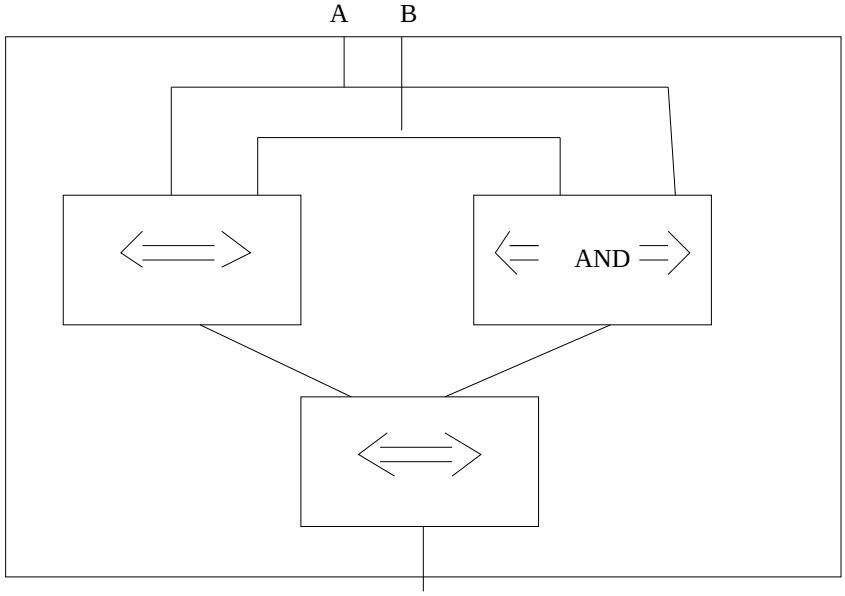
Figura 4. Circuito che realizza l'equivalenza logica.

Ora non è difficile capire che l'equivalenza può anche essere rappresentata da

una doppia implicazione. Cioè $A \Leftrightarrow B$ è la stessa cosa che $A \Rightarrow B \; AND \; B \Rightarrow A$. Assumiamo per il momento questo fatto, per verificarlo sperimentalmente in seguito. Il circuito che realizza $A \Rightarrow B \; AND \; B \Rightarrow A$ è facile da costruire componendo due circuiti che realizzano l'implicazione, come mostra la seguente Figura.



Possiamo ora ricavare la tabella di verit realizzata da questo circuito e verificare che è uguale a quella dell'equivalenza. È facile infatti vedere che i 2 circuiti dell'implicazione restituiscono 1 quando A e B sono o entrambi 0 o entrambi 1, mentre se A e B sono diversi, una delle 2 implicazioni restituisce 0 e quindi anche la risposta finale del circuito è 0.



Si osservi che in precedenza, formulando la congettura che l'equivalenza può essere espressa con 2 implicazioni, abbiamo usato l'espressione *è la stessa cosa che* per evitare di usare l'espressione *è equivalente*. In realtà potremmo verificare l'equivalenza dei 2 circuiti tramite il circuito della Figura precedente che realizza la formula: $(A \Leftrightarrow B) \Leftrightarrow (A \Rightarrow B \; AND B \Rightarrow A)$. La tabella di verità di questo circuito è facile da calcolare (provateci): essa contiene tutti 1. Questo sta appunto ad indicare che per ogni valore di A e B i 2 circuiti confrontati danno lo stesso risultato, cioè essi sono equivalenti.

Questo esempio mostra anche che ci possono essere circuiti molto diversi che realizzano la stessa tabella di verità.

In conclusione, gli esempi precedenti mostrano che componendo le operazioni (circuiti) di partenza possiamo definire nuove operazioni (circuiti) e possiamo anche dimostrare che 2 operazioni (circuiti) sono equivalenti (oppure no).

4.1 Circuiti per sommare

Consideriamo ora la realizzazione di operazioni più vicine alla macchina di von Neumann: vogliamo realizzare un circuito che riesce a sommare tra loro 2 numeri binari. Prima di affrontare il problema vediamo brevemente come funzionano i numeri binari e che rapporto hanno con quelli in base 10.

Siamo abituati a vedere i numeri interi rappresentati in base 10, ma è possibile utilizzare altre basi. In informatica è importante la rappresentazione in base 2, visto che questo è il modo in cui i numeri (e tutte le informazioni) sono rappresentate nella memoria di un computer. I numeri in base 2 vengono detti binari. I numeri binari seguono le stesse regole di quelli in base 10. Per esempio il numero binario 100101 rappresenta il numero in base 10, 37 (nel seguito denoteremo questo fatto con $100101 = 37_{10}$). Il valore 37 viene ottenuto dal numero binario nel modo seguente: $1*2^5+0*2^4+0*2^3+1*2^2+0*2+1*2^0 = 37$ (visto che gli 0 non contano per il valore finale, nel seguito mostreremo solo i bit a 1, quindi per il nostro esempio scriveremo: $2^5 + 2^2 + 2^0$). Dunque ogni 1 del numero binario corrisponde a 2^{p-1}, dove p è la sua posizione a partire dalla destra del numero. Da quanto appena detto, segue immediatamente il seguente Fatto:

Fatto 4.1 *1. Un numero binario B con n bits rappresenta un valore compreso tra 0 (se B contiene solo zeri) e $2^n - 1$ (se contiene solo uni).*

2. B rappresenta un valore maggiore di $2^{n-1} - 1$ sse il suo bit più a sinistra (quello in posizione n) è 1.

In informatica si usano spesso anche le rappresentazioni dei numeri in base 8 (ottali) e 16 (esadecimali). Nei numeri ottali si usano solo le cifre da 0 a 7, mentre negli esadecimali per avere 16 simboli, si aggiungono alle cifre 0...9 anche 6 lettere A..F, dove A rappresenta 10, B 11 e cosi via fino a F che rappresenta 15.

Torniamo ai numeri binari. Si osservi che un numero binario rappresenta un intero dispari sse il suo bit più a destra è 1. È facile vedere che se eliminiamo il

bit più a destra di un numero binario, il numero che otteniamo in questo modo è il quoziente della divisione intera per 2 del valore originale. Inoltre il valore (1 o 0) del bit eliminato è il resto della divisione stessa. In effetti questo fenomeno è l'analogo di quello che succede ad un numero in base 10 se lo dividiamo per 10: la cifra più a destra viene eliminata ed il valore di questa cifra è il resto della divisione.

Mostriamo questo fatto con un esempio: se eliminiamo il bit più a destra di 37_{10}=100101 otteniamo 10010 che rappresenta $2^4 + 2^1 = 18 = 37/2$. Inoltre il bit eliminato è 1 che è infatti il resto della divisione 37/2 (ovvio visto che 37 è dispari).

Questa osservazione ci suggerisce un algoritmo per calcolare la rappresentazione binaria di un qualsiasi numero in base 10: si divide il numero per 2 e si ricorda il resto, si divide il quoziente per 2 e si ricorda di nuovo il resto e si continua in questo modo finchè il quoziente diventa 0. La concatenazione dei resti da destra verso sinistra dà il valore binario cercato.

Esempio 4.2 *Dato il valore* 40_{10} *calcoliamo la sua rappresentazione binaria nel modo appena descritto:*
$40/2 = 20$ $resto = 0$
$20/2 = 10$ $resto = 0$
$10/2 = 5$ $resto = 0$
$5/2 = 2$ $resto = 1$
$2/2 = 1$ $resto = 0$
$1/2 = 0$ $resto = 1$

la codifica binaria di 40_{10} *è quindi 101000.*

In Fig.5 mostriamo un esempio di addizione tra due numeri binari di 6 bits.

7	6 5 4 3 2 1	colonne
1	1 1 1	riporti
	011100	+
	100111	=

1	000011	

Fig.5 Somma di numeri binari

Esercizio 4.3 *Trovare i valori in base 10 dei 2 numeri binari sommati in Fig.5 e del risultato della somma e verificare che la somma dà il risultato corretto.*

La somma in base 2 segue le stesse regole della somma in base 10 opportunamente modificate. In particolare, anche nella somma in base 2 si verificano

riporti. Nel nostro esempio si verifica un riporto a partire dalla colonna 4 da destra fino alla colonna 6. Questo causa il fatto che il risultato abbia 7 bits con un uno in colonna 7. Insomma il risultato è troppo grande per "starci" in 6 bits. Infatti con 6 bits possiamo rappresentare valori da 0 a $2^6 - 1$, cioè 63, vedi il Fatto 4.1(1).

Esempio 4.4 *Vogliamo ora progettare un circuito che calcoli somme come quella di Fig.5. A questo fine costruiamo prima un circuito che si occupa di sommare i due bits di una colonna qualsiasi dei due numeri da sommare. Oltre ai due bits da sommare che chiameremo X e Y, questo circuito avrà in input il riporto dalla colonna precedente, che chiameremo R. Esso dovrà produrre il bit che risulta dalla somma (lo chiameremo B) ed anche il riporto che andrà sulla colonna successiva (lo chiameremo Z).*

La tabella di verità di questo circuito è data in Fig.6.

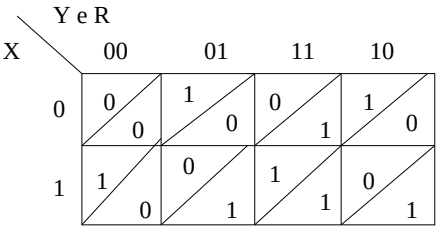
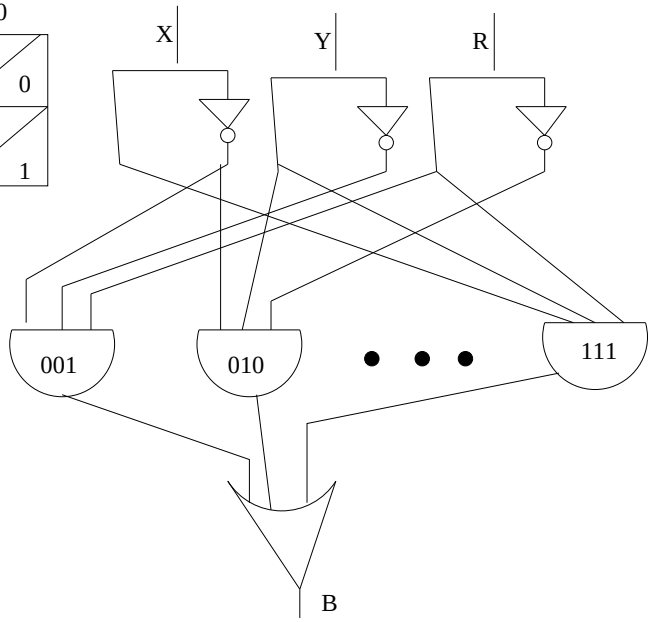


Fig. 6
Circuito 1-ADD che somma 2 bits con riporto.



La tabella è diversa da quelle viste finora in due cose: dato che ci sono 3 input, per rappresentarla in due dimensioni, 2 input (Y e R) sono considerati assieme sulle colonne della tabella, inoltre, ogni elemento della tabella contiene nella parte alta il valore di B ed in basso quello di Z. Dalla tabella è immediato ricavare il circuito che la realizza. Consideriamo le parti in alto degli elementi della tabella. Per ogni elemento 1 siano a_X, a_Y, ed a_R i corrispondenti valori di X, Y e R, allora il circuito contiene un AND (si noti a 3 ingressi, ma è ovvio come ottenerlo usando 2 AND normali) che ha in ingresso X se a_X è 1 e $\hat{X}$ (cioè la negazione di X) se a_X è 0 e lo stesso per Y e R. Basterà poi

far entrare le uscite di questi AND in un OR per avere il circuito che produce B, vedi Fig.6. Dovrebbe essere facile costruire in modo analogo un circuito che calcola il riporto Z. Chiameremo 1-ADD il circuito complessivo che calcola sia B che Z.

La Fig.7 mostra come ottenere un addizionatore per numeri binari di n bits, per un $n > 0$ qualunque, semplicemente mettendo in serie n circuiti 1-ADD.

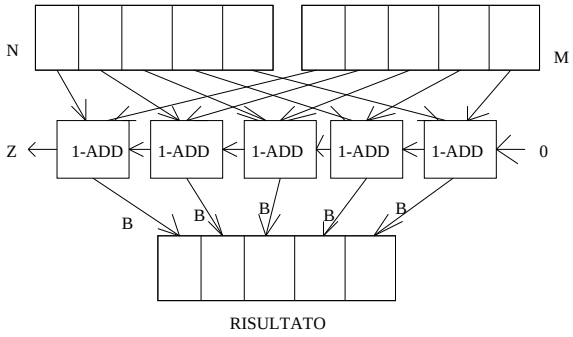


FIGURA 7. Circuito che somma 2 numeri binari di 5 bits

5 Rappresentazione degli interi, dei reali e dei caratteri

Dato che ogni computer ha, ovviamente, dimensione finita, nei computer solo alcuni valori interi e reali possono essere rappresentati. Generalmente un valore intero viene rappresentato con una parola, cioè con 32 bits, comunque in quanto segue assumeremo di avere n bits con un n qualunque maggiore di 0. In una sequenza di n bit, diremo che il bit più a destra è in posizione 1, quello alla sua sinistra in posizione 2, e così via fino a quello a sinistra che sarà in posizione n.

Con n bits, si può al massimo rappresentare numeri da 0 fino a $2^n - 1$, cf. il Fatto 4.1(1). C'è però il problema degli interi negativi. La maniera più semplice di rappresentare interi col segno è di riservare il bit in posizione n per il segno: se questo bit è 1, allora l'intero è negativo, se è 0 il numero è positivo. In questo modo possiamo rappresentare numeri da $-2^{n-1} - 1$ a $2^{n-1} - 1$. Questa rappresentazione soffre però di 2 problemi. Il primo problema è che ci sono 2 rappresentazioni dello 0: con segno + e -. Il secondo problema, più importante, è che le operazioni di somma e differenza sarebbero complicate dal fatto che si dovrebbe stabilire il segno del risultato. Per ovviare a questi problemi si adotta la rappresentazione degli interi detta *in complemento a 2.* In questa rappresentazione gli interi positivi da 0 a $2^{n-1} - 1$ si rappresentano normalmente (lasciando a 0 il bit in posizione n), mentre un numero negativo $k \in [-2^{n-1}, -1]$, viene rappresentato come $k + 2^n$. Un esempio pare utile.

Esempio 5.1 *Supponiamo per semplicità che $n = 6$. La rappresentazione in complemento a 2 di -4 è $-4+2^6 = -4+64 = 60$ che in binario è 111100. Provi-*

amo ora -32: -32+64=32 che in binario è 100000. Si osservi che la rappresentazione in complemento a 2 di un numero negativo ha sempre 1 in colonna 6. Questo segue dal fatto che i numeri negativi rappresentabili sono in $[-2^{n-1}, -1]$, *quindi nel nostro esempio di 6 bits, i numeri vanno da -32 a -1. Se sommiamo a questi numeri* 2^n, *otteniamo al minimo il valore* $2^n - 2^{n-1} = 2^{n-1}$, *che nel nostro esempio è 64-32=32. Ora questo numero deve necessariamente avere 1 in colonna 6, cf. il Fatto 4.1(2) (si ricordi che questo bit vale* $2^5 = 32$*) ed a maggior ragione avranno 1 in colonna 6 le rappresentazioni di tutti gli altri numeri negativi (che saranno maggiori di 32).*

Quindi con 6 bit rappresentiamo i numeri positivi da 0 a 31 (con i corrispondenti numeri binari in cui il bit in posizione 6 è sempre 0) e rappresentiamo i negativi da -32 a -1 con i numeri binari che rappresentano 32...63, rispettivamente.

Vediamo ora con un esempio come la somma tra numeri interi di segno diverso diventa banale con la rappresentazione in complemento a 2.

Esempio 5.2 *Supponiamo di voler sommare 20=010100 e -32 che sappiamo essere rappresentato in complemento a 2 da 32=100000. Faremo 20+32=52. Essendo 52 un numero più grande di 31 esso rappresenta un numero negativo. Ma quale numero? Per saperlo basta sottrargli* $2^6 = 64$, *cioè decodificarlo: 52-64=-12 il risultato atteso!!*

Proviamo ora con -20+(-5). -20 è rappresentato da -20+64=44=101010, mentre -5 è rappresentato da -5+64=59=111011. Facendo la somma si ottiene 44+59=103. abbiamo un problema: il risultato dell'addizione non è rappresentabile con 6 bit, esso è infatti 1100111. Ma anche questo problema è facile da risolvere: basta buttare via il bit in posizione 7 ottenendo quindi, 100111=39 che rappresenta il numero 39-64=-25 come desiderato.

Naturalmente il problema di poter ottenere valori troppo grandi positivi o troppo piccoli negativi per essere rappresentati con i bits a disposizione esiste. Questo problema è detto problema del **supero o overflow**. *Cercate per esempio (con i soliti 6 bits) di sommare -20 con -15. Il risultato (in base 10) è -35 che non è rappresentabile visto che il minimo numero rappresentabile (con 6 bits) è -32. Facciamo i conti per vedere cosa succede:*

```
-20 diventa -20+64= 44 in binario 101100
-15 diventa -15+64= 49 in binario 110001

La loro somma:
               1
                101100+
                110001=
               --------
               1011101
```

buttando via il bit in più (colonna 7) otteniamo un valore positivo! (0 in colonna 6). Ovviamente abbiamo un problema! Cioé abbiamo supero. Si osservi che in

questa somma c'è un unico riporto in colonna 7, e che non c'è invece riporto nella colonna 6. È possibile dimostrare che osservando i riporti nella colonna 6 e 7 (n ed $n+1$ in generale) è possibile accorgersi dell'occorrenza del supero.

L'esempio precedente introduce i seguenti 2 importanti fatti che enunciamo senza dimostrazione.

Teorema 5.3 *Siano a e b interi rappresentati in complemento a 2 con n bits,*

- *$a+b$ dà supero se e solo se i riporti in posizione n ed $n+1$ sono diversi.*
- *se $a+b$ non dà supero, allora i primi n bit (da destra) del risultato della somma è sempre il risultato corretto. Insomma, basta buttare via l'eventuale $n+1$-esimo bit come nella somma -20-5 dell'esempio precedente.*

In sostanza il Teorema precedente spiega come fare correttamente le operazioni di somma tra interi in complemento a 2. Controllate la correttezza del Teorema verificando che esso correttamente caratterizza tutti gli esempi di somme con e senza supero discussi prima. Da quanto detto è possibile capire che la somma ed il test di supero sono operazioni che possono essere realizzate in modo molto efficiente da un computer.

La sottrazione può essere eseguita facilmente usando la somma. Se dobbiamo calcolare $a-b$ con b positivo, basta trasformare b nel suo complemento a 2 e sommarlo ad a. Ottenere il complemento a due di un valore positivo è molto semplice: basta partire dalla posizione 1 e muovere verso sinistra lasciando tutto uguale fino al primo 1 (compreso) e complementare tutto il resto. Il motivo di questa regola è spiegato nel seguente esempio.

Esempio 5.4 *Seguendo la regola appena data, il complemento di 26=011010 è 100110: le prime 2 colonne sono uguali, mentre le 4 più a sinistra sono complementate. Il risultato è corretto visto che, 100110=38 che rappresenta 38-64=-26. In effetti il complemento a 2 di 26 è: 64-26, cioè 1000000-011010. Anche nella sottrazione binaria si seguono le stesse regole che in quella in base 10. La sottrazione inizia in colonna 1. fino a che entrambi i bit sono 0, il risultato è 0. Al primo 1 di 011010, si deve prendere un prestito da 1000000, ottenendo 0111120-011010. Osserva che il 2 in posizione 2 è dovuto al prestito. Esso corrisponde al 10 del prestito nella sottrazione in base 10. Quindi in colonna 2 si ottiene 1, mentre nelle colonne dalla 6 alla 3 si ottiene il complemento di quanto presente in 26. Infatti, 1-1=0 e 1-0=1.*

Esercizio 5.5 *La regola per trovare il complemento a 2 funziona anche per i numeri negativi?*

5.1 Reali

Consideriamo ora i numeri reali. Per prima cosa dobbiamo capire come si rappresenta in binario la parte decimale di un numero in base 10. Si ricordi che

per i numeri interi in base 10 abbiamo calcolato la loro rappresentazione binaria dividendo ripetutamente il numero per 2 (fino ad avere quoziente 0) e concatenando da destra a sinistra i resti delle divisioni. Questo metodo funziona perchè il resto della divisione per 2 è il bit più a destra della rappresentazione binaria del numero ed il rimanente della rappresentazione lo si ottiene trovando la rappresentazione binaria del quoziente e concatenandolo con questo resto.

Per calcolare la rappresentazione binaria della parte frazionaria di un numero in base 10, usiamo l'idea inversa. Sia $f = 0.q$ il numero decimale da rappresentare, la sua rappresentazione binaria f_b viene calcolata nel modo seguente: si moltiplica f per 2, se si ottiene un valore $1.d$ allora il primo bit B dopo la virgola di f_b è 1, se invece il risultato è $0.d$ allora B è 0. In entrambi i casi si cerca allo stesso modo la rappresentazione binaria d_b di $0.d$. La rappresentazione di f è $f_b = 0.B \cdot d_b$ (dove $\cdot$ rappresenta la concatenazione). Questo metodo di calcolo termina solo quando, dopo una moltiplicazione per 2 si ottiene un numero con parte decimale 0. Ci sono casi in cui questo non succederà mai, si consideri per esempio $f = 0.2$. Questo non è un problema per noi in quanto vogliamo rappresentare i valori in un computer e quindi con un numero fisso e finito di bits. Quindi nel nostro caso il processo di calcolo terminerà o quando otterremo parte decimale 0 oppure quando avremo eseguito tante moltiplicazioni per 2 quanti sono i bits che abbiamo a disposizione per la rappresentazione stessa. Naturalmente in questo caso avremo che f_b rappresenta un numero diverso da f, ma è comunque l'approssimazione migliore possibile di f con il numero di bit che abbiamo a disposizione.

Illustriamo il metodo appena introdotto con un esempio:

Esempio 5.6 *Consideriamo il valore decimale* $f = 0.650$*, calcoliamo la sua rappresentazione binaria, supponenedo di avere solo 8 bits a disposizione, con la seguente sequenza di 8 moltiplicazioni per 2:*
$0.650 * 2 = 1.300 \Rightarrow B = 1$
$0.3 * 2 = 0.6 \Rightarrow B = 0$
$0.6 * 2 = 1.2 \Rightarrow B = 1$
$0.2 * 2 = 0.4 \Rightarrow B = 0$
$0.4 * 2 = 0.8 \Rightarrow B = 1$
$0.8 * 2 = 1.6 \Rightarrow B = 1$ *siamo in un loop, la rappresentazione non sarà precisa!!*
$0.6 * 2 = 1.2 \Rightarrow B = 1$
$0.2 * 2 = 0.4 \Rightarrow B = 0$
Quindi la migliore rappresentazione binaria di $f = 0.650$ *con 8 bits per i decimali è:* $f_b = 0.10101110$*. E' ovvio che con un maggior numero (ma sempre finito) di bit a disposizione potremmo ottenere una migliore approssimazione di* f*, ma in nessun caso potremmo ottenere una rappresentazione esatta.*

Si osservi che nella rappresentazione binaria dei numeri decimali troviamo una situazione nuova rispetto alla rappresentazione binaria degli interi. Nel caso degli interi, il fatto di avere un numero finito di bit ci permetteva di rappresentare (in modo esatto!) tutti gli interi contenuti in un certo intervallo finito. Nel caso dei reali esiste ancora un intervallo di valori al di fuori del quale i val-

ori non sono rappresentabili, ma, anche all'interno di questo intervallo, ci sono valori che possono solamente essere approssimati.

Torniamo ora ai dettagli della rappresentazione dei reali all'interno di un computer. Assumiamo per semplicità che per rappresentarli venga usata una parola, cioè 4 bytes. La descrizione è facilmente estendibile a più bytes. I reali vengono rappresentati in generale in *virgola mobile* (floating point). In questa rappresentazione, la parola viene divisa in 3 parti come rappresentato in Fig. 8:

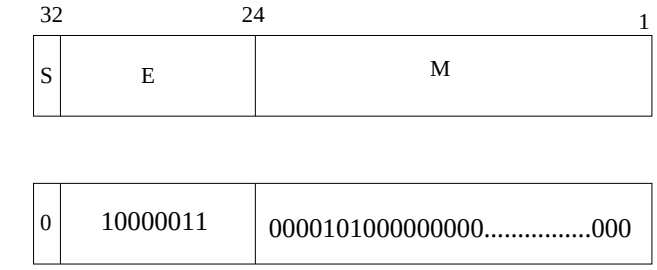


Fig 8. Rappresentazione in virgola mobile di 16.625

1. Il bit in colonna 32 indica il segno (0= positivo, 1=negativo);
2. la parte E di 8 bits contiene l'esponente, esso è letto come un intero senza segno, quindi può contenere valori in [0,255]; Per poter rappresentare esponenti anche negativi, al valore di E viene sempre sottratto -127, quindi i valori possibili degli esponenti sono in [-127,128].
3. La parte M di 23 bits rappresenta la mantissa. Nell'esempio di Fig.8 M=0000101 rappresenta $1 + 1/2^5 + 1/2^7 = 1 + 1/32 + 1/128$, dove l'1 intero è sempre implicito nella rappresentazione. Visto che l'esponente in Fig.8 è 131, l'esponente reale è 131-127=4, il valore binario rappresentato è quindi, $2^4 * 1.0000101 = 10000.101 = 16 + 1/2 + 1/8 = 16.625$.

E' importante osservare che la richiesta del punto 3 precedente che ogni valore reale sia rappresentato come $2^e * 1.d$, dove d è la parte decimale del valore e l'1 intero è sempre presente (e quindi viene omesso), causa il fatto che per ogni valore reale (rappresentabile) esiste un'unica rappresentazione che si chiama **normalizzata**. Supponiamo infatti di non richiedere questo fatto, allora per esempio il numero 4 avrebbe molte rappresentazioni, tra cui, $2^{130-127} * 0.1$ e $2^{131-127} * 0.01$ e cosi via. Mentre l'unica rappresentazione normalizzata di 4 è $2^{129-127} * 1.0$ (si noti che abbiamo scritto la parte intera in 1.0 solo per una questione di chiarezza mentre in realtà nella rappresentazione normalizzata, esso verrebbe omesso).

È importante anche notare che il valore reale 0.0 viene necessariamente trattato diversamente da tutti gli altri valori reali. Infatti la sua rappresentazione non può soddisfare il punto 3 precedente. Lo 0 in floating point viene rappresentato con tutti i bit della parola a 0.

Si noti che con una parola di 32 bits si possono rappresentare al più 2^{32} valori reali. Quindi il numero dei valori rappresentati è uguale al numero di interi (se anche per questi si usa 1 parola). Quello che cambia rispetto agli interi è che i reali non sono distribuiti uniformemente nell'intervallo dei valori rappresentati. Essi hanno massima concentrazione nell'intervallo [-1,1], dove si trovano circa la metà dei valori.

Esercizio 5.7 *Dare la rappresentazione normalizzata del valore reale 5.5*

Esercizio 5.8 *assumendo che la mantissa consista solo di 8 bits, dare la rappresentazione normalizzata del valore reale 1.2*

5.2 Caratteri

Un'altra classe di valori ha molta importanza nei computer: i caratteri. I caratteri sono importanti perchè lo scambio di informazioni tra esseri umani e computers avviene (oggi) generalmente tramite sequenze di caratteri. Oltre ai caratteri alfabetici normali (i 26 caratteri dell'alfabeto inglese minuscolo e maiuscolo), sono importanti i caratteri numerici, (0,...,9), i caratteri di punteggiatura, le parentesi di vario tipo, i simboli matematici (+,-,* eccetera) i caratteri greci ed anche molti caratteri di controllo come l'enter, il tab ed altri, Visto che ogni cosa in un computer deve essere rappresentata da sequenze di 0 ed 1, anche per i caratteri si usa una codifica in cui ogni carattere è rappresentato da 8 bits. Questa codifica standard usata da tutti i computer, si chiama codifica **ASCII**. Visto che 2^8 é 256, la codifica ASCII permette di gestire 256 caratteri diversi. Normalmente questo è sufficiente. In alcuni casi è invece necessario poter gestire molti più simboli (per esempio gli ideogrammi cinesi o giapponesi). Per questi casi è stata fissata un'altra codifica in cui 16 bits sono usati per ogni carattere. In questo caso si parla di **wide characters**, cioè caratteri larghi. Poichè $2^12 = 4096$, questo è il numero di wide characters.

6 Linguaggio macchina e linguaggio assembler

Considereremo ora un insieme di istruzioni macchina, cioè le istruzioni is_i per cui nello schema della CPU di Fig.2 è prevista una componente C_{is_i}. L'insieme di queste istruzioni forma il **linguaggio macchina**. Computer diversi hanno generalmente linguaggi macchina diversi. Le diversità sono però spesso solo superficiali ed il linguaggio macchina che presenteremo nel seguito, benchè sia molto semplificato rispetto ai linguaggi macchina dei computer reali, è comunque sufficiente ad illustrare i concetti più importanti relativi ai linguaggi macchina.

Nel nostro linguaggio macchina ci sono le seguenti istruzioni:

1. Istruzioni che trasferiscono il contenuto di una parola di RAM in un registro della CPU.

2. Istruzioni che trasferiscono il contenuto di un registro della CPU in una parola di RAM.

3. Operazioni aritmetiche (somma, sottrazione, moltiplicazione e divisione) che trovano gli operandi in 2 registri della CPU e mettono il risultato nel primo dei due.

4. Istruzioni di confronto e di salto. Esaminiamo queste operazioni assieme dato che esse cooperano strettamente. Un'istruzione di confronto paragona tra loro il contenuto di 2 registri R_i ed R_j della CPU ed a seconda della relazione esistente tra di loro, inserisce un valore in un apposito registro RC della CPU nel modo seguente:

   - Se il contenuto di R_i è maggiore di quello di R_j allora RC riceve il valore 1;.
   - se i due registri hanno contenuto uguale, allora RC riceve il valore 0;
   - infine, se il contenuto di R_i è minore di quello di R_j allora RC riceve il valore -1;

   È chiaro che i valori specifici assegnati ad RC da un'operazione di confronto, non sono importanti. Quello che importa è che RC descrive i 3 possibili risultati del test. Infatti, il registro RC viene usato dalle istruzioni di salto condizionato che possono compiere le seguenti azioni: saltare ad una particolare istruzione del programma (di cui deve essere specificato l'indirizzo), se RC=1 oppure saltare se RC=0 e così via. Esiste anche un'operazione di salto incondizionato, che salta sempre ad una data istruzione del programma (indipendentemente da RC). Cosa significhi "saltare ad un'altra istruzione del programma" diventerà chiaro (spero) tra breve.

5. L'istruzione che segnala la fine del programma.

Vediamo subito come esempio un piccolo (frammento di) programma macchina che somma 2 numeri interi che risiedono nella RAM e mette il risultato nella RAM. Un tale programma è illustrato nella Figura 9, in cui nella parte di sinistra sono descritte *a parole* le 4 istruzioni che eseguono quanto richiesto e nella parte destra si mostra come esse verrebbero rapprentate veramente nella RAM. In modo simile, nella parte sinistra di Figura 9 si mostrano i 2 numeri da sommare in base 10 mentre nella parte destra essi sono rappresentati in binario ed ancora nella parte sinistra gli indirizzi della RAM che contengono i valori e le istruzioni sono in base 10 e nella parte destra in binario. Questo è importante in quanto gli indirizzi fanno parte delle istruzioni di trasferimento.

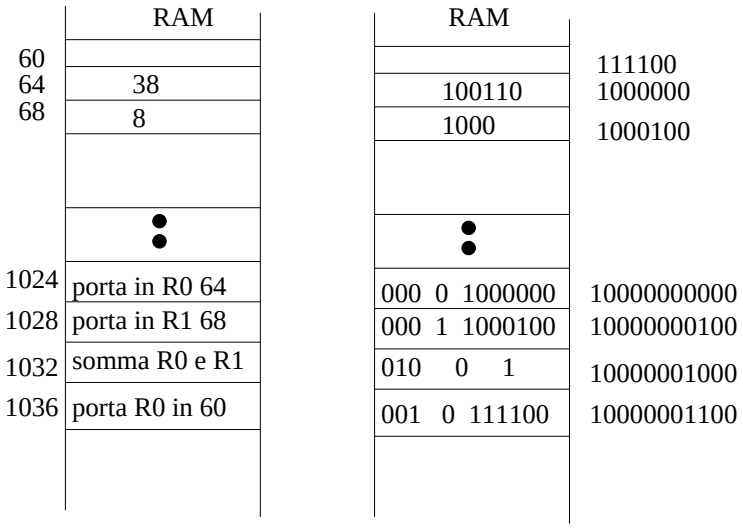


Figura 9. esempio di programma in linguaggio macchina

Questo esempio dovrebbe far capire che nel linguaggio macchina, ogni istruzione è rappresentata da un codice binario che la distingue dalle altre: nella Figura 9 l'istruzione di trasferimento dalla CPU ad un registro ha codice 000, quella che trasferisce da un registro alla RAM ha codice 001, quella di somma ha codice 010. Allo stesso modo i registri $R_0, R_1 \ldots$ sono rappresentati dalla codifica binaria del loro indice: il registro R_0 è rappresentato da 0, R_2 da 10 e così via. Infine gli indirizzi RAM da cui si vuole trasferire oppure in cui si vuole mettere informazioni, sono specificati in binario. Per esempio nella Figura 9, la prima istruzione specifica l'indirizzo 64 (in binario 1000000) per spostare il valore 38 che si trova in quell'indirizzo della RAM nel registro R_0.

L'esempio di Figura 9 pur nella sua semplicità mostra chiaramente i limiti del linguaggio macchina:

1. Il codice binario è difficile da scrivere e capire e modificare. Immaginate un programma di alcune migliaia di istruzioni!!

2. Il programmatore deve occuparsi di dove nella RAM vengono inseriti i suoi dati, visto che i loro indirizzi devono far parte del programma che deve scrivere.

Comunque, negli anni 40 si programmava in linguaggio macchina e programmi e dati venivano scritti "a mano" nella RAM. Ben presto però questo compito si è rivelato impossibile da eseguire senza introdurre errori. La teoria dei linguaggi di programmazione è nata in quegli anni allo scopo di definire linguaggi di programmazione che permettano al programmatore di costruire programmi concentrandosi sui problemi da risolvere senza preoccuparsi dei dattagli tecnici del particolare computer su cui verrà eseguito il suo programma. Il primo passo su questa lunga strada è stato compiuto rapidamente ed è rappresentato

dalla definizione (fine anni '40) dei linguaggi `assembler`. Parliamo al plurale di linguaggi assembler, dato che, essendo questi molto vicini ai linguaggi macchina, ne è stato definito uno per ogni tipo di computer. Nel seguito illustreremo un linguaggio assembler molto semplice (inventato da me) che comunque servirà ad illustrare i punti essenziali di questi linguaggi.

In un linguaggio assembler le istruzioni vengono indicate con un codice mnemonico (nel senso che esso richiama l'azione che l'istruzione deve compiere) al posto del loro codice binario. Per esempio l'operazione che trasferisce una parola dalla RAM ad un registro della CPU si chiama *LOAD* (carica), quella che dal registro trasferisce nella RAM è *STORE* (immagazzina), quella che somma 2 registri è *ADD* (somma). La lista completa dei codici mnemonici è specificata di seguito.

```
1)Istruzioni di trasferimento: LOAD e STORE
2)Istruzioni aritmetiche: ADD, SUB, MULT e DIV
3)Istruzione di confronto: COMPARE
4)Istruzioni di salto condizionato: BREQ (salta se =),
BRGT (salta se >) BRLT (salta se <) BRLE (salta se < o =)
5)Istruzione di salto incondizionato: BRANCH (salta sempre)
6)Fine programma: STOP
```

La seconda novità dell'assembler è il fatto che invece di scrivere nelle istruzioni gli indirizzi nella RAM dei dati, si possono usare nomi per identificare i dati (questi nomi sono infatti detti **identificatori**). Anche i registri possono essere specificati esplicitamente nelle istruzioni: se si vuole usare il registro R_{15} lo si scrive esplicitamente (invece di dover specificare l'indice 15 in binario come nelle istruzioni macchina).

Rivediamo subito l'esempio di Figura 9 tradotto nel nostro Assembler.

```
Z: ;
X: 38;
Y: 8;
LOAD R0 X;
LOAD R1 Y;
ADD R0 R1;
STORE R0 Z;
STOP;
```

`Figura 10. un semplice programma Assembler`

È necessario precisare alcune cose su questo esempio.

1. Il programma inizia con 3 **dichiarazioni**. Ognuna introduce un identificatore di un dato che serve nel programma (X, Y e Z nel nostro esempio) specificando o meno per esso un valore iniziale.

2. Le istruzioni del programma richiedono operandi di tipo diverso. Quelle di trasferimento (`LOAD e STORE`) richiedono come primo operando un registro e come secondo un identificatore. L'istruzione aritmetica `ADD` richiede

due registri, in quanto essa calcola la somma dei contenuti dei due registri e la mette nel primo registro. Quindi `ADD R0 R1;` nel nostro esempio calcola 38+8=46 e lo mette in R0. Finalmente, l'istruzione finale `STOP` non richiede operandi.

3. Dovrebbe essere chiaro che, nonostante che dal punto di vista del significato il programma assembler di Figura 10 sia molto simile a quello macchina di Figura 9, il programma assembler non può essere eseguito direttamente dall'hardware di alcun computer. Per essere eseguito esso deve essere **tradotto** nel corrispondente programma macchina. Come avevamo anticipato nella Sezione 3, questa traduzione viene eseguita per noi da un programma che si chiama in generale **compilatore**. Un compilatore per l'assembler viene chiamato **assemblatore**. Nelle conclusioni cercheremo di spiegare a grandi linee come funziona l'assemblatore. Per ora osserviamo solamente che un programma assembler può essere eseguito solo su di un computer che, al di sopra della macchina hardware, possieda alcuni strati software tali da contenere l'assemblatore ed anche tali da permetterci di inviare un programma assember come input del computer. Questo può avvenire attraverso la tastiera (come nei moderni PC), ma anche attraverso, per esempio, un floppy oppure un CD o un modem eccetera.

Nonostante esso sia così semplice, l'Assembler è sufficiente per eseguire qualunque calcolo, così come il linguaggio macchina. Ricordate infatti che qualunque programma noi scriviamo (in Java o Pascal o C++, eccetera), esso verrà sempre tradotto in una sequenza di istruzioni macchina e questa e solo questa verrà alla fine eseguita dal nostro computer.

Mostriamo ora l'uso delle istruzioni di confronto e di salto dell'Assembler con un esempio leggermente più complicato di quello della Figura 10. Invece di sommare 2 numeri e mettere il risultato in una terza posizione della RAM, vogliamo che il nostro programma metta il risultato nella RAM al posto del maggiore dei due numeri che vengono sommati. Ecco il programma. Dopo lo commentiamo.

```
X: 38;
Y: 8;
LOAD R0 X;
LOAD R1 X;
LOAD R2 Y;
ADD R1 R2;
COMPARE R0 R2;
BRGE pippo;
STORE R1 Y;
STOP;
pippo: STORE R1 X;
STOP;
```

Figura 11. Un programma Assembler che usa COMPARE e salto

Visto che l'operazione di somma scrive il risultato nel registro `R1`, dobbiamo mettere il valore di `X` anche nel registro `R0` per poterlo confrontare con `Y` (contenuto in `R2`). Per capire il resto del programma è necessario leggere con attenzione la descrizione dell'istruzione di confronto all'inizio di questa sezione. Essa inserisce 1, 0 o -1 nel registro `RC` a seconda del risultato del confronto. Cioè se (il contenuto) di `R0` è maggiore di quello di `R2`, `RC` riceve 1, se è uguale, `RC` riceve 0, altrimenti riceve -1. Quindi l'istruzione di salto condizionato successiva al `COMPARE`, cioè `BRGE`, salta all'istruzione etichettata `pippo` se RC contiene 1 o 0 (cioè se `X`$\geq$ `Y`) e continua con l'istruzione successiva al `BRGE` se invece `Y` $>$ `X`. Si osservi che in questo programma abbiamo dato un nome (`pippo`) ad un'istruzione verso la quale il programma contiene un'operazione di salto condizionato. Il nome `pippo` viene usato anche nell'istruzione di salto stessa. In generale, questi nomi dati alle istruzioni si chiamano **etichette**. La struttura di questo programma è rappresentata in modo facile da capire nel grafo della Figura 13. Una tale rappresentazione di un programma è detta **flowchart** in quanto rappresenta i possibili flussi dell'esecuzione all'interno del programma.

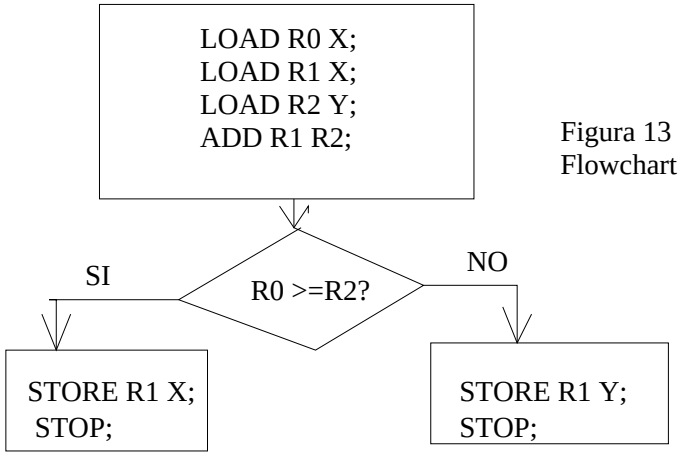


Figura 13
Flowchart con test

Vediamo ora un programma che impiega una struttura di controllo nuova per noi ed in generale di enorme importanza: **il ciclo o iterazione**. Vogliamo scrivere un programma assembler che dati due numeri (interi positivi) calcoli il resto della loro divisione e, per rendere interessante ed al contempo semplice l'esercizio, richiediamo che questo programma utilizzi solo la sottrazione per fare questo calcolo e non la divisione nè la moltiplicazione.

```
X: 356;
Y: 23;
RESTO: ;
LOAD R0 X;
LOAD R1 Y;
ciclo: COMPARE R0 R1;
BRLT fine;
```

```
SUB R0 R1;
BRANCH ciclo;
fine: STORE R0 RESTO;
STOP;
```

Il flowchart di questo programma è in Figura 14. La parte essenziale del programma è il ciclo che consiste di 4 istruzioni: `ciclo: COMPARE, BRLT, SUB` e `BRANCH ciclo;`. Come dice il nome *ciclo*, queste operazioni sono ripetute fino a quando il test COMPARE mette -1 nel registro RC e quindi `BRLT fine;` salta all'istruzione con etichetta `fine` e quindi il resto (cioè il contenuto di `R0`) viene messo nell'identificatore `RESTO` (dichiarato apposta per questo scopo).

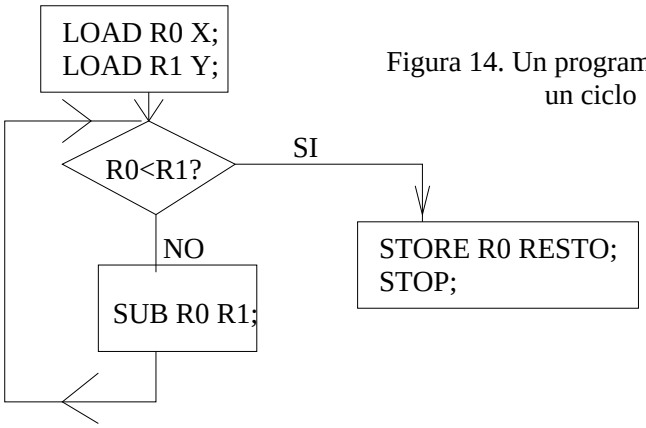


Figura 14. Un programma con un ciclo

7 Conclusioni

Terminiamo con alcune osservazioni volte a porre alcune delle nozioni viste in una luce più generale.

1. Nei programmi Assembler che abbiamo visto possiamo distinguere 3 tipi diversi di costrutti:

   - Istruzioni volte a calcolare risultati dai valori iniziali e ad assegnarli a certi identificatori appositi (per esempio, l'identificatore `RESTO` nell'esempio del resto). Quest'azione viene chiamata **assegnazione**.
   - Il costrutto di **test**, in cui una condizione viene testata ed a seconda del risultato ottenuto si continua il calcolo in modi diversi, si veda la Figura 13.
   - Il **ciclo** in cui una sequenza di istruzioni viene ripetuta fino a quando una condizione è soddisfatta.

   Questi 3 costrutti sono gli elementi di base della maggior parte dei Linguaggi di programmazione come Pascal Fortran, C++ eccetera. Quindi essi hanno una generalità che va ben al di là del linguaggio Assembler.

2. Nella Sezione precedente abbiamo detto che l'assemblatore (ed in generale ogni compilatore) compie automaticamente una traduzione da un programma Assembler ad un corrispondente programma in linguaggio macchina. Possiamo essere un poco più precisi di questo. Fondamentalmente l'assemblatore è costituito da 2 parti:

   - La parte di traduzione vera e propria che traduce i codici mnemonici delle operazioni e dei registri nei corrispondenti codici binari. Per trasformare anche gli identificatori e le etichette delle istruzioni ha bisogno del secondo modulo, descritto al punto seguente.
   - Il modulo di caricamento: trova delle parole RAM (disponibili, cioè libere da altri usi) in cui caricare i valori delle variabili e le istruzioni (tradotte) del programma. Questi indirizzi RAM vengono sostituiti agli identificatori ed alle etichette delle istruzioni.

   Attraverso queste 2 azioni combinate si ottiene il corrispondente programma macchina che verrà poi eseguito dalla CPU.

3. Si ricordi la figura *a cipolla* che nella Sezione 3 ci è servita per illustrare la nozione di un computer moderno in cui la parte hardware è arricchita da strati successivi di software che offrono operazioni sempre più sofisticate all'utente. Ora visto che un qualunque programma (Assembler, C++ o altro) viene sempre tradotto in una sequenza di istruzioni macchina, è chiaro che sarebbe possibile costruire un circuito che esegue qualunque programma come un'unica istruzione. Con questo vogliamo dire che il cuore della cipolla, cioè la parte hardware, non è fissa e sempre più o meno uguale in ogni computer. Ci sono computer in cui essa è in grado di eseguire solo operazioni semplici, simili a quelle viste nella Sezione precedente, e ci sono altri computer in cui la parte hardware è invece capace di eseguire operazioni più sofisticate e (necessariamente) in numero maggiore. Nel primo caso si parla di architetture RISC (Reduced Instruction Set Computers) e nel secondo caso di architetture CISC (Complex Instruction Set Computers).