



Esami A e B di Architetture degli Elaboratori

Soluzione

A.A. 2017-18 — I appello — 30 gennaio 2018

N.B.: il punteggio associato ad ogni domanda è solo una misura della difficoltà, e peso, di ogni domanda. Per calcolare il voto complessivo bisogna normalizzare a 30 (circa).

- Convertire il valore 203.42_5 in base 7.

R: (3 pt) Passiamo attraverso la base decimale:

$$203.42_5 = 2 \cdot 5^2 + 3 \cdot 5^0 + 4 \cdot 5^{-1} + 2 \cdot 5^{-2} = 50 + 3 + 0.8 + 0.08 = 53.88$$

$$\begin{array}{r} 53 \mid 7 \\ \hline 7 \mid 4 \\ 1 \mid 0 \\ 0 \mid 1 \\ \hline .88 \mid 7 \\ .16 \mid 6 \\ .12 \mid 1 \\ .84 \mid 0 \\ .88 \mid 5 \\ \dots \mid \dots \end{array}$$

da cui otteniamo immediatamente il valore nella nuova base: $104.\overline{6105}_7$.

- Sono dati i seguenti valori n_1 e n_2 codificati in complemento a 2 a 8 bit: $n_1 = 10110111$, $n_2 = 00000011$. Si calcoli il prodotto $n_1 \cdot n_2$ e, se possibile, si esprima il risultato nella stessa codifica.

R: (3 pt) Conviene calcolare il modulo del prodotto e, successivamente, ricordare che n_1 ha segno negativo. Il modulo di n_1 si ottiene per complementazione a 2: $-n_1 = 01001001$, da cui immediatamente $|n_1| \cdot n_2$:

$$\begin{array}{r} 01001001 * \\ 00000011 = \\ \hline 01001001 + \\ 01001001 = \\ \hline 11011011 \end{array}$$

Ora, cambiare di segno di questo risultato richiederebbe almeno 9 bit adoperando per esso la codifica in complemento a 2. Dunque il prodotto appena calcolato non può essere espresso nella codifica richiesta.

- [INF] Fornire il risultato dell'esercizio precedente in codifica *floating point* IEEE 754 a 32 bit.

R: (3 pt) Il risultato trovato sopra può essere subito messo nella forma $-1.1011011E7$. La codifica richiesta avrà dunque bit di segno asserito, esponente uguale a $127 + 7 = 134 = 10000110_2$ e infine mantissa uguale a 1011011 . Sistemando sui 32 bit previsti dallo standard IEEE 754 e convertendo alla base esadecimale:

$$1|1\ 0\ 0\ 0\ 0\ 1\ 1\ 0|1\ 0\ 1\ 1\ 0\ 1\ 1\ 0 \dots \\ C \quad | \quad 3 \quad | \quad 5 \quad | \quad B \quad | \quad 0 \dots$$

da cui la codifica richiesta: $C35B0000_{16}$.

- La legge di Moore afferma che il numero di transistor nell'unità di memoria quadruplica ogni tre anni. Se la densità in un *chip* di memoria è uguale a $3 \cdot 10^7$ transistor/cm², che densità dobbiamo aspettarci nello stesso *chip* dopo 4 anni e mezzo?

R: (3 pt) Per come è definito, il tasso di crescita appena visto è per sua natura *esponenziale*. Se il numero quadruplica ogni tre anni allora il raddoppio avviene ogni anno e mezzo. Quindi dopo quattro anni e mezzo assistiamo a tre raddoppi, dunque la densità attesa sarà $2 \cdot 2 \cdot 2 \cdot 3 \cdot 10^7 = 24 \cdot 10^7$ transistor/cm².

5. Trovare il circuito col minore numero di porte AND, OR, NOT, il quale realizza l'espressione

$$E = \overline{A}B\overline{C} + A\overline{B}\overline{C} + \overline{A}\overline{B}\overline{C}$$

R: (3 pt)

$$E = \overline{A}B\overline{C} + A\overline{B}\overline{C} + A\overline{B}\overline{C} = \overline{A}(B+\overline{B})\overline{C} + A\overline{B}\overline{C} = \overline{A}\overline{C} + A\overline{B}\overline{C} = \overline{A}\overline{C} + A(\overline{B}+\overline{C}) = \overline{A}\overline{C} + A\overline{C} + A\overline{B} = \overline{C} + A\overline{B}$$

da cui discende immediato il circuito contenente 2 porte NOT, una porta AND e una porta OR.

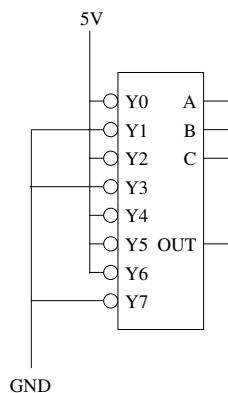
6. [INF] Verificare la minimizzazione ottenuta sopra con una mappa di Karnaugh.

R: (3 pt) Ci sono due regioni connesse, rispettivamente contenenti quattro e due simboli 1. La prima presenta la costanza del solo termine \overline{C} ; la seconda presenta la costanza dei termini A e \overline{B} .

BC	00	01	11	10
A				
----		--		
0		1 0	0	1
1		1 <>1	0	1
----		--		

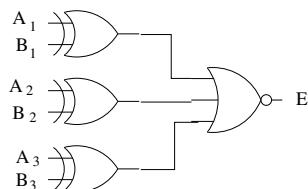
7. [INF] Riprodurre il circuito appena progettato adoperando un *multiplexer*.

R: (3 pt) È sufficiente scegliere un multiplexer a 8 ingressi e 3 controlli collegati rispettivamente ad ABC , e poi connettere a una sorgente in tensione (per esempio 5V) gli ingressi associati ai valori di controllo 000, 010, 100, 101, 110. Viceversa, gli altri 3 ingressi dovranno essere collegati a una tensione nulla (detta anche di massa, o GND).



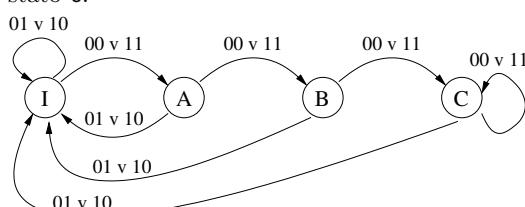
8. Progettare un comparatore di ingressi a 3 bit, cioè una rete combinatoria in grado di decidere se due ingressi $A_1A_2A_3$ e $B_1B_2B_3$ sono identici.

R: (3 pt)



9. [INF] Progettare la macchina a stati finiti (di Mealy oppure di Moore) che realizza il comparatore di cui sopra, supponendo che essa a ogni istante riceva per un tempo indefinitamente lungo coppie di bit $A_1B_1, A_2B_2, A_3B_3, \dots, A_nB_n, \dots$, e all'istante n -esimo debba stabilire se gli ingressi $A_{n-2}A_{n-1}A_n$ e $B_{n-2}B_{n-1}B_n$ sono identici.

R: (3 pt) La seguente macchina di Moore effettua la comparazione partendo dallo stato I, producendo un'uscita uguale a 0 in corrispondenza degli stati I, A e B; viceversa, produce un'uscita uguale a 1 in corrispondenza dello stato C.



10. Si vogliono codificare le cifre decimali $0, 1, \dots, 9$ con 4 bit adoperando un codice a *lunghezza variabile* il quale ottimizzi l'efficienza della codifica delle cifre decimali 0 e 1. Si dia un possibile codice che rispetta questo vincolo.

R: (3 pt) Quattro bit sono sufficienti, ma non necessari per codificare 10 simboli. Poichè con 3 bit possiamo codificare 8 simboli, assegniamo il valore 0 del bit più significativo alla codifica delle cifre 0 e 1; altrimenti occorrerà valutare i tre bit meno significativi per decodificare le restanti otto cifre decimali. Quindi, per esempio,

```

0 -> 00
1 -> 01
2 -> 1000
3 -> 1001
4 -> 1010
5 -> 1011
6 -> 1100
7 -> 1101
8 -> 1110
9 -> 1111

```

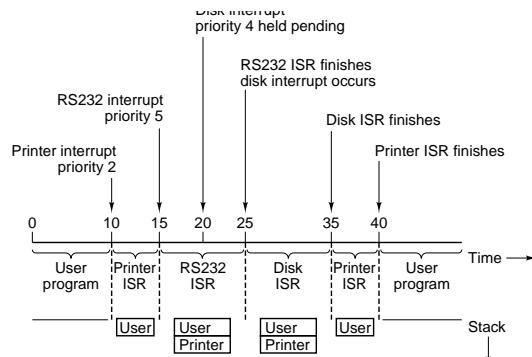
In tal modo, 0 e 1 necessitano di due soli bit per essere codificati mentre tutte le altre cifre necessitano di 4 bit.

11. Una CPU interagisce con 4 dispositivi esterni etichettati A, B, C, D. Ogni richiesta del dispositivo D dev'essere servita prima di quelle provenienti da C, e ogni richiesta del dispositivo B dev'essere servita prima di quelle provenienti da A e D. In quanti modi possono essere disposti i dispositivi in un bus che implementa il protocollo *daisy chain*? Elencarli.

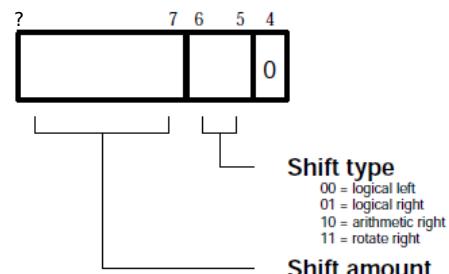
R: (3 pt) Il bus monterà, in ordine di prossimità alla CPU, i seguenti dispositivi: BADC, BDAC oppure BDCA.

12. Si dia schematicamente un esempio di *mascheramento dell'interrupt* durante il servizio a dispositivi esterni da parte della CPU.

R: (3 pt) Qualunque variazione della situazione descritta in figura è accettabile.



13. I campi in figura appartengono a una microistruzione per ARM che prevede la possibilità di traslare o ruotare il contenuto di un registro. Quale valore posizionale va sostituito al '?' in alto a sinistra nella figura? Quali sono le traslazioni o rotazioni minime e massime ammesse in tal caso sul registro?



R: (3 pt) Se parliamo di un'architettura a 32 bit, allora le traslazioni o rotazioni ammesse sul registro variano da un minimo di 0 bit a un massimo di 31 bit. Occorrendo 5 bit per specificare l'entità di queste operazioni, il campo più a sinistra in figura occuperà i bit dal 7 all'11.

14. In una memoria paginata, la *page table* si compone di righe ciascuna lunga 7 bit. Se la memoria principale paginabile è di 1 Mbyte, di quanti bit si compone il campo *offset* di ogni indirizzo fisico di memoria? Qual è la dimensione di ogni pagina?

R: Una memoria di 1 Mbyte necessita di 20 bit per essere indirizzata. Escludendo il bit di presenza/assenza della pagina in memoria, la *page table* contiene 6 bit per indirizzare ogni pagina. Dunque, restano $20 - 6 = 14$ bit per specificare l'*offset* nella stessa pagina, la quale dunque avrà dimensione $2^{14} = 16$ kB.

15. [INF] Scrivere un programma in assembly per ARM, il quale trova il valore massimo tra n elementi contenuti in un file testuale di nome *inputLista.txt*. Il file contiene il numero n di elementi nella prima riga e n valori interi nelle righe successive. Al termine dell'esecuzione il programma avrà restituito in memoria l'indice dell'elemento di valore massimo, oppure il valore -1 se il file contiene 0 elementi. Nel caso in cui nel file esista più di un elemento di valore massimo il programma restituirà l'indice del primo massimo presente nel file.

R: (9 pt)

```

.data
stringa:
    .asciiz "inputLista.txt"
output:
    .skip 4
    .text
main:
    ldr r0, =stringa      ; string address in r0
    mov r1, #0              ; read mode
    swi 0x66                ; open file in read mode
    mov r2, r0                ; save file handler
    swi 0x6c                ; read number of integers
    mov r3, r0                ; save number of integers in r3
    mov r5, #0x80000000      ; start with minInt in r5
    mov r4, #0                ; current element in r4
    mov r6, #0xFFFFFFFF      ; min element in r6
loop:
    subs r3, r3, #1          ; decrement r3 and set status
    blt exit                ; exit if negative
    add r4, r4, #1            ; increment r4
    mov r0, r2                ; load file handler
    swi 0x6c                ; read integer from file
    cmp r5, r0                ; if r0-r5 < 0...
    movlt r5, r0              ; ..then update minimum..
    movlt r6, r4              ; ..and its index
    b loop                  ; next element
exit:
    mov r0, r2                ; load input file handler
    swi 0x68                ; close file
    ldr r0, =output          ; output address in r0
    str r6, [r0]              ; save min element index
;; end
    swi 0x11                ; exit
    .end

```



Esame A di Architetture degli Elaboratori

Soluzione

A.A. 2017-18 — II appello — 20 febbraio 2018

N.B.: il punteggio associato ad ogni domanda è solo una misura della difficoltà, e peso, di ogni domanda. Per calcolare il voto complessivo bisogna normalizzare a 30 (circa).

- Convertire il valore $4.\bar{3}$ dalla base 10 alla base 3.

R: (3 pt)

$$\begin{array}{r} 4 \mid 3 \\ \hline 1 \mid 1 \\ 0 \mid 1 \end{array} \quad \begin{array}{r} .333\dots \mid 3 \\ \hline 0 \mid 1 \end{array}$$

e quindi $4.\bar{3} = 11.1_3$. Infatti, $3^{-1} = 1/3 = 0.\bar{3}$, da cui $4.\bar{3} = 1 \cdot 3^1 + 1 \cdot 3^0 + 1 \cdot 3^{-1}$.

- Sono date le seguenti codifiche in complemento a 2 a 8 bit: $n_1 = 10110111$, $n_2 = 11001100$. Si calcoli la differenza $n_1 - n_2$ e, se possibile, si esprima il risultato nella stessa codifica.

R: (3 pt) Complementando n_2 per cambiarne il segno: $-n_2 = 00110100$, eseguiamo successivamente la somma

$$\begin{array}{r} 10110111 + \\ 00110100 = \\ \hline 11101011 \end{array}$$

che, coinvolgendo un valore positivo e uno negativo, non dà *overflow* ed è quindi codificata come tale.

- [INF] Fornire il risultato dell'esercizio precedente in codifica *floating point* IEEE 754 a 32 bit.

R: (3 pt) Il risultato trovato sopra può essere subito messo nella forma $-1.0101E4$. La codifica richiesta avrà dunque bit di segno asserito, esponente uguale a $127+4 = 131 = 10000011_2$ e infine mantissa uguale a 0101. Sistemando sui 32 bit previsti dallo standard IEEE 754 e convertendo alla base esadecimale:

$$\begin{array}{r} 1|1\ 0\ 0\ 0\ 0\ 0\ 1\ 1|0\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ \dots \\ \text{C} \quad | \quad 1 \quad | \quad \text{A} \quad | \quad 8 \quad | \quad 0 \dots \end{array}$$

da cui la codifica richiesta: `0xC1A80000`.

- Si esprima il numero di bit presenti in una memoria a 512 MB come potenza di 2.

R: (3 pt) Ricordando che $1 \text{ MB} = 2^{20}$ byte: $8 \cdot 512 \cdot 2^{20} = 2^3 \cdot 2^9 \cdot 2^{20} = 2^{32}$ bit.

- Adoperando le regole di equivalenza booleana, calcolare quanto vale E nell'espressione seguente:

$$E = \overline{ABC} + \overline{AC}$$

R: (3 pt) Sfruttando le regole di De Morgan si ha $E = \overline{A} + \overline{B} + \overline{C} + \overline{\overline{A}} + \overline{C} = 1 + \overline{B} + \overline{C} = 1$

- [INF] Verificare il risultato ottenuto sopra con una mappa di Karnaugh.

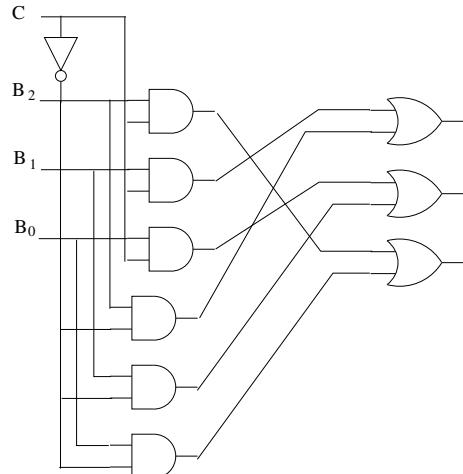
R: (3 pt) La tabella di verità porge una mappa che presenta solo simboli 1:

$$\begin{array}{ccccc} BC & 00 & 01 & 11 & 10 \\ A & & & & \end{array}$$

$$\begin{array}{ccccc} 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{array}$$

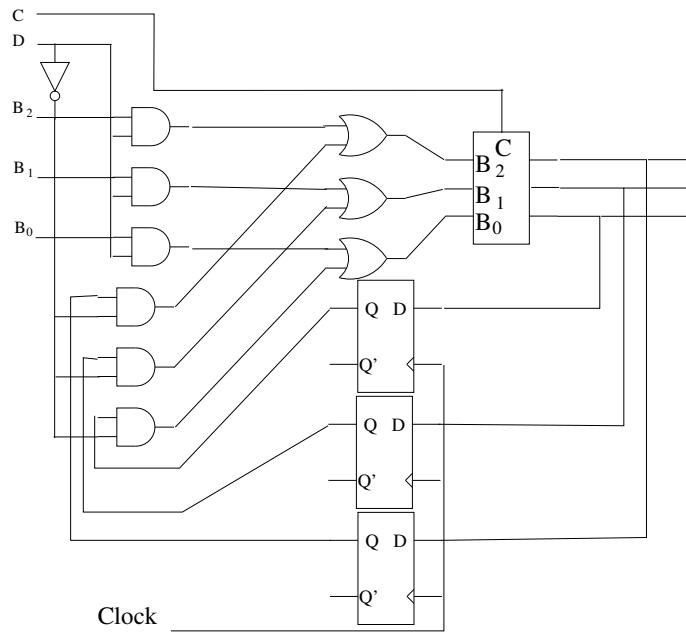
7. Progettare un traslatore rotazionale a 3 bit a sinistra, cioè una rete combinatoria in grado di ruotare un ingresso a 3 bit (B_2, B_1, B_0) verso sinistra di un bit oppure no, rispettivamente se un controllo C a un bit è asserito oppure no. In altre parole, se $C = 1$ allora (B_2, B_1, B_0) ruota in (B_1, B_0, B_2) .

R: (3 pt)



8. [INF] Avendo a disposizione il traslatore di cui sopra, che per comodità può essere denotato come un unico blocco, progettare un circuito sequenziale che, a seconda del valore assunto dal controllo C , a ogni ciclo di clock esegue la rotazione di un bit a sinistra oppure no dei bit (B_2, B_1, B_0) in ingresso oppure dei tre bit in uscita all'istante precedente. Quest'ultima decisione viene presa in base al valore assunto all'istante corrente da un bit etichettato come D .

R: (3 pt)



9. Un codice a ripetizione tripla del carattere codifica le cifre decimali 0,1,...,9 in corrispondenti triplette di cifre identiche. Il decodificatore restituisce la cifra c se le tre cifre costituenti la tripletta sono tutte uguali a c , altrimenti segnalando un errore di trasmissione e scartando la tripletta. Si dica in quanti casi una trasmissione errata viene riconosciuta come tale rispetto alla totalità delle codifiche che possono essere ricevute.

R: (3 pt) Su 1000 possibili triplette ricevute, una è quella corretta. Nove triplette contengono tre cifre identiche errate ma non vengono scartate, mentre tutte le altre vengono scartate. Quindi, in 990 casi su 999 una trasmissione errata viene riconosciuta come tale.

10. Un elementare sistema di I/O programmato prevede che ogni 100 ms la CPU impegni 3 ms del proprio tempo nel *polling* di un'unica periferica. Se la periferica non deve essere servita questo tempo viene sprecato, viceversa in caso di servizio i 3 ms vengono impiegati utilmente e la CPU lavora globalmente con un'efficienza del 100%. In queste ipotesi si calcoli quante volte la periferica dovrebbe mediamente richiedere un servizio affinché la CPU lavori con un'efficienza del 99%.

R: (3 pt) Posto x il numero medio di volte in cui la periferica richiede di essere servita, se la periferica non richiede mai il servizio ($x = 0$) l'efficienza è del $100 - 3 = 97\%$. Al contrario, se richiede sempre un servizio ($x = 1$) l'efficienza sale al 100%. Il numero medio di volte si ottiene dunque imponendo che

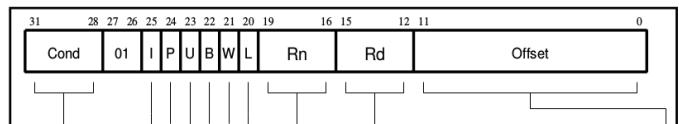
$$97 \cdot (1 - x) + 100 \cdot x = 99 \Rightarrow 97 - 97x + 100x = 99 \Rightarrow 3x = 2 \Rightarrow x = 2/3, \text{ cioè due volte su tre.}$$

11. Se la periferica del sistema di I/O presentato in precedenza inoltra mediamente 12 richieste di servizio al secondo, il *polling* può funzionare? Si motivi sinteticamente la risposta.

R: (3 pt) No, poichè il sistema in questione non può servire la periferica mediamente più di 10 volte al secondo.

12. I campi in figura appartengono a una microistruzione per ARM che prevede la possibilità di salvare il contenuto di un registro **Rd** nell'indirizzo di memoria specificato dal contenuto del registro **Rn** corretto da un valore di *offset* presente nell'omonimo campo. Quanti kB può valere al massimo lo stesso offset in base alle informazioni presenti in figura?

ARM Processor Instruction Set



13. Quali locazioni di una memoria principale di 64 kB può contenere la linea 1023 di una *cache* a 32 byte composta da 1024 *entry*?

R: Le locazioni dalla $32 \cdot 1023 = 32736$ alla $32 \cdot 1023 + 31 = 32767$ e dalla $32 \cdot 1024 + 32 \cdot 1023 = 65504$ alla $32 \cdot 1024 + 32 \cdot 1023 + 31 = 65535$.

14. [INF] Scrivere un programma in assembly per ARM il quale, caricati nei registri **r2** e **r3** rispettivamente due numeri m e n tali che $0 < n \leq m$, calcola il valore $m!/(m-n)! = m(m-1)\dots(m-n+1)$ attraverso una procedura fattoriale che esegue le $n-1$ moltiplicazioni e deposita il risultato nel registro **r1**.

R: (9 pt)

```

.text
main:
    mov r2, #5          ; read m
    mov r3, #3          ; read n
    subs r3, r3, #1     ; will multiply n-1 times
    blne fattoriale   ; if n!=0 then call
    mov r1, r2          ; copy result in r1
    swi 0x11            ; exit program

fattoriale:
    stmdfd sp!, {r4, lr} ; save registers
    mov r4, r2          ; m in r4
    sub r2, r2, #1      ; m <- m-1
    subs r3, r3, #1      ; decrement r3
    blne fattoriale   ; if r3!=0 call again
    mul r2, r4, r2      ; otherwise multiply
    ldmfd sp!, {r4, lr} ; restore registers
    mov pc, lr           ; restore PC

.end

```



Esame B di Architetture degli Elaboratori

Soluzione

A.A. 2017-18 — II appello — 20 febbraio 2018

N.B.: il punteggio associato ad ogni domanda è solo una misura della difficoltà, e peso, di ogni domanda. Per calcolare il voto complessivo bisogna normalizzare a 30 (circa).

- Convertire il valore $5.\bar{1}$ dalla base 10 alla base 9.

R: (3 pt)

$$\begin{array}{r} 5 \mid 9 \\ \hline 0 \mid 5 \end{array} \quad \begin{array}{r} .111\dots \mid 9 \\ \hline 0 \mid 1 \end{array}$$

e quindi $5.\bar{1} = 5.1_9$. Infatti, $9^{-1} = 1/9 = 0.\bar{1}$, da cui $5.\bar{1} = 5 \cdot 9^0 + 1 \cdot 9^{-1}$.

- Sono date le seguenti codifiche in complemento a 2 a 8 bit: $n_1 = 10110111$, $n_2 = 11001100$. Si calcoli la differenza $n_2 - n_1$ e, se possibile, si esprima il risultato nella stessa codifica.

R: (3 pt) Complementando n_1 per cambiarne il segno: $-n_1 = 01001001$, eseguiamo successivamente la somma

$$\begin{array}{r} 11001100 + \\ 01001001 = \\ \hline 100010101 \end{array}$$

che, coinvolgendo un valore positivo e uno negativo, dà falso *overflow* ed è quindi codificata come 00010101, scartando appunto il nono bit.

- [INF] Fornire il risultato dell'esercizio precedente in codifica *floating point* IEEE 754 a 32 bit.

R: (3 pt) Il risultato trovato sopra può essere subito messo nella forma 1.0101E4. La codifica richiesta avrà dunque bit di segno nullo, esponente uguale a $127 + 4 = 131 = 10000011_2$ e infine mantissa uguale a 0101. Sistemando sui 32 bit previsti dallo standard IEEE 754 e convertendo alla base esadecimale:

$$0|1\ 0\ 0\ 0\ 0\ 0\ 1\ 1|0\ 1\ 0\ 1\ 0\ 0\ 0\ 0\ \dots$$

4 | 1 | A | 8 | 0 \dots

da cui la codifica richiesta: 0x41A80000.

- Si esprima il numero di bit presenti in una memoria a 64 GB come potenza di 2.

R: (3 pt) Ricordando che $1 \text{ GB} = 2^{30}$ byte: $8 \cdot 64 \cdot 2^{30} = 2^3 \cdot 2^6 \cdot 2^{30} = 2^{39}$ bit.

- Adoperando le regole di equivalenza booleana, calcolare quanto vale E nell'espressione seguente:

$$E = \overline{\overline{A} + \overline{B} + \overline{C}} \overline{A + \overline{C}}$$

R: (3 pt) Sfruttando le regole di De Morgan si ha $E = (ABC)(\overline{AC}) = ABC\overline{AC} = A\overline{ABC} = 0BC = 0$

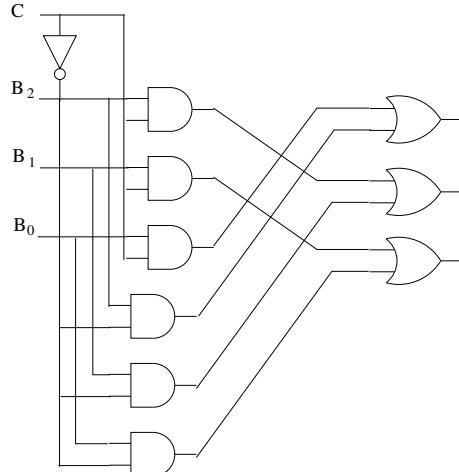
- [INF] Verificare il risultato ottenuto sopra con una mappa di Karnaugh.

R: (3 pt) La tabella di verità porge una mappa che presenta solo simboli 0:

BC	00	01	11	10
A				
0	0	0	0	0
1	0	0	0	0

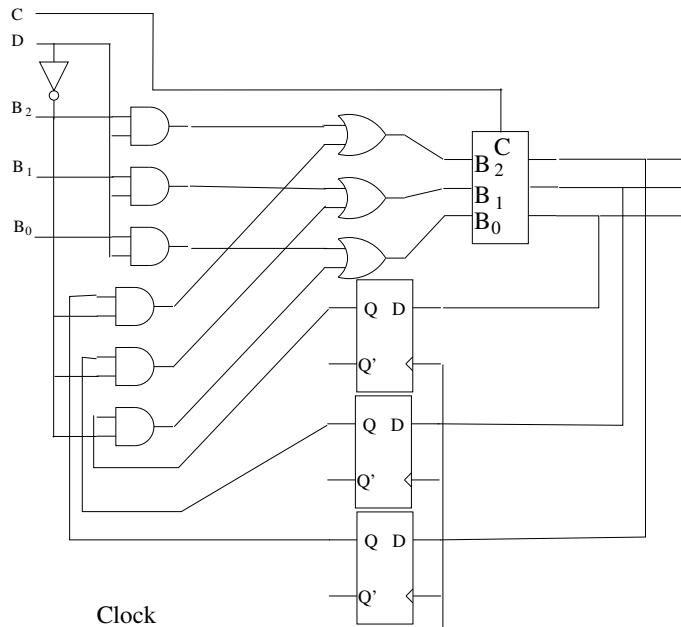
7. Progettare un traslatore rotazionale a 3 bit a destra, cioè una rete combinatoria in grado di ruotare un ingresso a 3 bit (B_2, B_1, B_0) verso destra di un bit oppure no, rispettivamente se un controllo C a un bit è asserito oppure no. In altre parole, se $C = 1$ allora (B_2, B_1, B_0) ruota in (B_0, B_2, B_1) .

R: (3 pt)



8. [INF] Avendo a disposizione il traslatore di cui sopra, che per comodità può essere denotato come un unico blocco, progettare un circuito sequenziale che, a seconda del valore assunto dal controllo C , a ogni ciclo di clock esegue la rotazione di un bit a destra oppure no dei bit (B_2, B_1, B_0) in ingresso oppure dei tre bit in uscita all'istante precedente. Quest'ultima decisione viene presa in base al valore assunto all'istante corrente da un bit etichettato come D .

R: (3 pt)



9. Un codice a ripetizione tripla del carattere codifica le cifre decimali $0, 1, \dots, 9$ in corrispondenti triplette di cifre identiche. Il decodificatore restituisce la cifra c se almeno due delle tre cifre costituenti la tripletta sono uguali a c , altrimenti segnalando un errore di trasmissione. Si dica in quanti casi il codice riesce a correggere l'errore rispetto alla totalità delle codifiche che possono essere ricevute.

R: (3 pt) Su 1000 possibili triplette ricevute, una è quella corretta. Ventisette triplette contengono due cifre identiche corrette. Quindi, in 27 casi su 999 una trasmissione errata viene corretta con successo.

10. Un elementare sistema di I/O programmato prevede che ogni 100 ms la CPU impegni 6 ms del proprio tempo nel *polling* di un'unica periferica. Se la periferica non deve essere servita questo tempo viene

sprecato, viceversa in caso di servizio i 6 ms vengono impiegati utilmente e la CPU lavora globalmente con un'efficienza del 100%. In queste ipotesi si calcoli quante volte la periferica dovrebbe mediamente richiedere un servizio affinché la CPU lavori con un'efficienza del 97%.

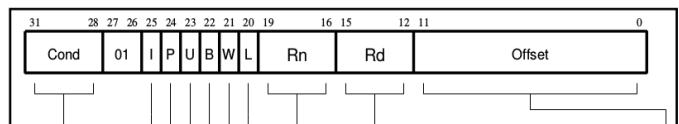
R: (3 pt) Posto x il numero medio di volte in cui la periferica richiede di essere servita, se la periferica non richiede mai il servizio ($x = 0$) l'efficienza è del $100 - 6 = 94\%$. Al contrario, se richiede sempre un servizio ($x = 1$) l'efficienza sale al 100%. Il numero medio di volte si ottiene dunque imponendo che $94 \cdot (1 - x) + 100 \cdot x = 97 \Rightarrow 94 - 94x + 100x = 97 \Rightarrow 6x = 3 \Rightarrow x = 1/2$, cioè una volta su due.

11. Qual è la richiesta media di servizi al minuto che la periferica non può oltrepassare nel sistema di I/O presentato in precedenza?

R: (3 pt) 600 servizi per ogni minuto.

12. I campi in figura appartengono a una microistruzione per ARM che prevede la possibilità di salvare il contenuto di un registro **Rd** nell'indirizzo di memoria specificato dal contenuto del registro **Rn** corretto da un valore di *offset* presente nell'omonimo campo. Se *offset* vale 2^8 , il contenuto del registro **Rn** può essere maggiore di *offset* in base alle informazioni presenti in figura?

ARM Processor Instruction Set



13. Quali locazioni di una memoria principale di 64 kB può contenere la linea 2 di una *cache* a 32 byte composta da 1024 *entry*?

R: Le locazioni dalla $32 \cdot 2 = 64$ alla $32 \cdot 2 + 31 = 95$ e dalla $64 + 32 * 1024 = 32832$ alla $95 + 32 * 1024 = 32863$.

14. [INF] Scrivere un programma in assembly per ARM il quale, caricati due numeri m e $n > 0$ rispettivamente nei registri **r2** e **r3**, calcola il valore $(m + n - 1)!/(m - 1)! = (m + n - 1)(m + n - 2) \dots (m + 1)m$ attraverso una procedura ricorsiva che esegue le $n - 1$ moltiplicazioni e deposita il risultato nel registro **r1**.

R: (9 pt)

```

.text
main:
    mov r2, #5          ; read m
    mov r3, #3          ; read n
    subs r3, r3, #1     ; will multiply n-1 times
    blne fattoriale   ; if n!=0 then call
    mov r1, r2          ; copy result in r1
    swi 0x11            ; exit program

fattoriale:
    stmdfd sp!, {r4, lr} ; save registers
    mov r4, r2          ; m in r4
    add r2, r2, #1      ; m <- m+1
    subs r3, r3, #1      ; decrement r3
    blne fattoriale   ; if r3!=0 call again
    mul r2, r4, r2      ; otherwise multiply
    ldmfd sp!, {r4, lr} ; restore registers
    mov pc, lr           ; restore PC

.end

```



Esame A di Architetture degli Elaboratori

Soluzione

A.A. 2017-18 — III appello — 19 giugno 2018

N.B.: il punteggio associato ad ogni domanda è solo una misura della difficoltà, e peso, di ogni domanda. Per calcolare il voto complessivo bisogna normalizzare a 32 (circa).

- Convertire il valore $13.\bar{2}_4$ alla base 8.

R: (3 pt) Trattandosi di basi che sono una potenza di due, conviene dapprima convertire alla base 2 e poi raggruppare le cifre risultanti per riscrivere il valore nella base 8:

$$13.\bar{2}_4 = \underbrace{01}_4 \underbrace{11}_4 \cdot \underbrace{\bar{1}0}_4 = \underbrace{01}_4 \underbrace{11}_4 \cdot \underbrace{\overline{101010}}_{\overline{222}_4} = 0 \underbrace{111}_7 \cdot \underbrace{\overline{10}}_{\overline{5}_8} \cdot \underbrace{\overline{010}}_{\overline{2}_8} = 7.\overline{52}_8.$$

- Sono date le seguenti codifiche in complemento a 2 a 8 bit: $n_1 = 10110100$, $n_2 = 00000100$. Se esiste, si calcoli il rapporto n_1/n_2 e se possibile lo si esprima nella stessa codifica.

R: (3 pt) Per sicurezza conviene calcolare $|n_1|/|n_2|$ e poi ricordare che n_1 codifica un valore negativo. Dunque, cambiando di segno n_1 si ha $-n_1 = 01001100$. Poiché n_2 vale 4, il calcolo del rapporto è immediato: basta traslare n_1 di due bit, ottenendo $|n_1|/|n_2| = |n_1|/n_2 = 00010011$. Complementando nuovamente si ha subito $n_1/n_2 = -|n_1|/n_2 = 11101101$.

Si noti che lo stesso risultato sarebbe uscito anche senza passare per il valore assoluto di n_1 .

- [INF] Fornire il risultato dell'esercizio precedente in codifica *floating point* IEEE 754 a 32 bit.

R: (3 pt) Il risultato trovato sopra può essere subito messo nella forma $-1.0011E4$. La codifica richiesta avrà dunque bit di segno asserito, esponente uguale a $127+4 = 131 = 10000011_2$ e infine mantissa uguale a 0011. Sistemando sui 32 bit previsti dallo standard IEEE 754 e convertendo alla base esadecimale:

$$\begin{array}{ccccccccc} 1 & | & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ & | & & & & & & & \\ C & & 1 & & 9 & & 8 & & 0 \dots \end{array}$$

da cui la codifica richiesta: `0xC1980000`.

- La *legge di Moore* afferma che il numero di transistor nell'unità di memoria quadruplica ogni tre anni. Se un chip di memoria di forma rettangolare oggi misura mediamente 1×0.5 cm, quanto misuravano mediamente i lati di un chip analogo 18 mesi fa, a parità di capacità di memoria?

R: (3 pt) Se il numero di transistor nell'unità di memoria quadruplica ogni tre anni allora il raddoppio della densità avviene ogni anno e mezzo. Quindi, a parità di capacità di memoria un anno e mezzo prima i lati erano mediamente più lunghi di un fattore $\sqrt[2]{2}$: $1 \cdot \sqrt[2]{2} \times 0.5 \cdot \sqrt[2]{2} \approx 1.4 \times 0.7$ cm.

- È dato il seguente listino prezzi in microeuro ($\mu\epsilon$): porta NAND 1 $\mu\epsilon$; porta NOR 5 $\mu\epsilon$; porta NOT 10 $\mu\epsilon$; porta AND 20 $\mu\epsilon$; porta OR 20 $\mu\epsilon$. Adoperando le regole di equivalenza booleana, dare il circuito che realizza l'espressione booleana $\overline{A} + \overline{B}$ al costo minimo, quantificando inoltre il costo stesso.

R: (3 pt) Adoperando le regole di De Morgan si ha $\overline{A} + \overline{B} = AB = \overline{AB}$, che adopera una porta NAND e una porta NOT costando dunque $1 + 10 = 11 \mu\epsilon$. Questa soluzione ha ricevuto punteggio pieno.

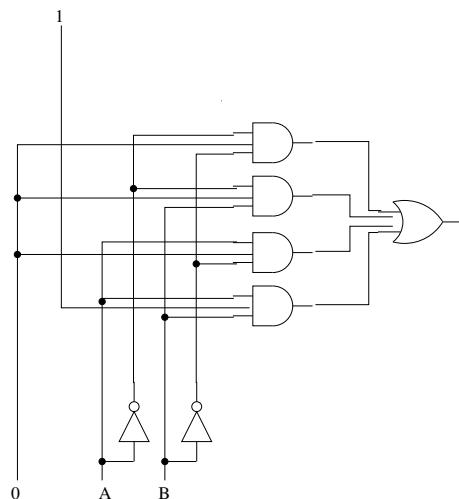
Il costo minimo in assoluto si trovava ricordando l'uguaglianza $E = E \cdot E \cdot \dots \cdot E$. Di qui l'ulteriore espressione $\overline{AB} = \overline{AB} \cdot \overline{AB}$ che, richiedendo solo due porte NAND, costa $1 + 1 = 2 \mu\epsilon$.

6. [INF] Si provi ad abbassare ulteriormente il costo del circuito all'esercizio precedente, ricordando che esso può accedere anche alle tensioni nulla e di alimentazione le quali rappresentano, rispettivamente, gli stati logici 0 e 1.

R: (3 pt) Continuando ad adoperare le regole di De Morgan si ha $\overline{A} + \overline{B} = \overline{AB} = \overline{AB} + 0 = \overline{AB} \cdot 1$, che adoperando due porte NAND costa $1 + 1 = 2 \mu\epsilon$. Chi all'esercizio precedente aveva già trovato la soluzione in assoluto migliore non poteva comunque scendere sotto il costo di $1 + 1 = 2 \mu\epsilon$.

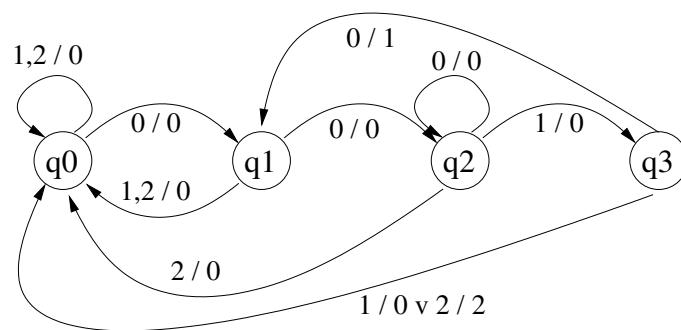
7. Realizzare l'espressione all'esercizio 5 adoperando un *multiplexer*, il cui circuito combinatorio dev'essere chiaramente esplicitato all'interno della rete booleana proposta.

R: (3 pt)



8. [INF] Disegnare il diagramma della macchina di Mealy che, leggendo simbolo dopo simbolo una sequenza indefinitamente lunga di caratteri appartenenti all'alfabeto $\mathcal{A} = \{0, 1, 2\}$, riconosce *ogni* istanza delle due sottosequenze 0012 e 0010 all'interno della sequenza letta. Per esempio, la sequenza 22001001022 contiene *due* istanze della sottosequenza 0010. A ogni ciclo di lettura la macchina in questione genera le seguenti uscite: 0 se non ha riconosciuto alcuna sottosequenza; 1 nell'istante in cui riconosce la sottosequenza 0010; 2 nell'istante in cui riconosce la sottosequenza 0012.

R: (3 pt)



9. Quanti bit contenenti dati informativi possiede un *codice di Hamming a N bit*?

R: (3 pt) I codici di Hamming a N bit contengono $\log_2(N + 1)$ bit di controllo. Di conseguenza i bit informativi sono $N - \log_2(N + 1)$. Per esempio, il codice di Hamming a 7 bit contiene $7 - 3 = 4$ bit informativi.

10. In un elementare sistema che deve servire un'unica periferica si deve scegliere tra un protocollo I/O, di tipo programmato oppure controllato. Di essi si sa che il primo serve la periferica in 3 ms, mentre il secondo necessita di 4 ms per espletare il medesimo servizio. In più si sa che la periferica può chiedere di essere servita esattamente ogni 100 ms, tuttavia questo avviene appunto solo se la periferica ha fatto richiesta; altrimenti essa non necessita di essere servita, e forse farà una richiesta trascorsi esattamente

altri 100 ms. Sulla base di questi dati si calcoli qual è il minimo numero di richieste che rende mediamente conveniente adoperare il protocollo I/O di tipo programmato.

R: (3 pt) Se la periferica necessita di essere servita ogni 100 ms allora il protocollo I/O di tipo programmato fa guadagnare al sistema 1 ms di tempo di calcolo ogni 100 ms. Il punto di parità tra i due protocolli avviene quando la periferica dev'essere mediamente servita tre volte su quattro, cioè il 75% delle volte. In tal caso anche il protocollo I/O di tipo controllato richiede 12 ms di tempo al sistema per le interruzioni. In definitiva il numero medio minimo di interruzioni che rende conveniente adoperare il protocollo I/O di tipo programmato è uguale a 3 ogni 400 ms.

11. Con riferimento all'esercizio precedente, si dica quante sono le richieste che saturano il sistema nel caso in cui si scelga di usare a) il protocollo I/O di tipo programmato e b) il protocollo I/O di tipo controllato.

R: (3 pt) Il protocollo I/O di tipo programmato è saturato quando la periferica richiede il servizio ogni 100 ms. Il protocollo I/O di tipo controllato non può essere mai saturato da una periferica che richiede il servizio ogni 100 ms, in quanto la serve in 4 ms.

12. Si dia un esempio in cui un'operazione di *logical shift right* (**lsr**) eseguita da un processore ARM non corrisponde all'operazione di *arithmetic shift right* (**asr**) eseguita sullo stesso dato.

R: (3 pt) Le due operazioni hanno esito diverso per esempio quando il dato rappresenta la codifica in complemento a due di un valore intero negativo.

13. Una memoria fisica di 1 Gbyte è paginata in pagine di 64 kB, che possono trovare posto in posizioni casuali di una memoria principale di 4 Mbyte. Non è prevista la presenza di memoria virtuale. Come dev'essere dimensionata in questo caso la *page table*?

R: La memoria fisica necessita di 30 bit per essere indirizzata. Di essi, 16 individuano l'*offset* all'interno di ogni pagina di $2^{10+6} = 64$ kB. La *page table* dunque deve possedere $2^{30-16} = 2^{14}$ locazioni corrispondenti alla parte più significativa dell'indirizzo. Ciascuna di esse dovrà contenere 22 bit per indirizzare ciascuna pagina casualmente in $2^{22} = 4$ Mbyte, più il bit di presenza/assenza della pagina in memoria principale per un totale di 2^{14} locazioni di 23 bit ciascuna.

Si noti che se per ottimizzare l'occupazione ogni pagina non finisse in una posizione casuale ma piuttosto multipla di 64 kB, allora in tal caso in memoria principale possono trovare posto $2^{22}/2^{16} = 2^6 = 64$ pagine, le quali a questo punto necessitano di soli 6 bit per essere indirizzate attraverso la *page table* all'interno della stessa memoria.

14. [INF] Con riferimento all'esercizio 8, scrivere un programma in assembly per ARM il quale legge dal file di testo **file.txt** una sequenza di caratteri appartenenti all'alfabeto $\mathcal{A} = \{0, 1, 2\}$ e quindi realizza la macchina sequenziale colà descritta. Il file di testo per comodità riporta nella prima riga il numero di caratteri presenti nel file a partire dalla seconda riga. L'output è scritto a ogni ciclo di lettura nel registro **r5**. Il programma termina allorquando il file è stato letto interamente.

R: (9 pt)

```
.data
stringa:
    .asciiz "input.txt"
    .text
main:
    ldr r0, =stringa      ; file name address in r0
    mov r1, #0             ; read mode
    swi 0x66               ; open file in read mode
    mov r2, r0              ; save file handler
    swi 0x6c               ; read number of integers
    mov r3, r0              ; save number of integers in r3
    mov r6, #1              ; r6 loaded with one
machine_q0:
    subs r3, r3, r6        ; decrement number of integers..
    beq end                ; ..and exit if zero
    mov r5, #0              ; output zero
    mov r0, r2              ; load file handler
```

```

        swi 0x6c          ; read integer from file
        cmp r0, #0         ; if input==0..
        beq machine_q1     ; ..jump to state q1
        b machine_q0        ; else stay in current state
machine_q1:
        subs r3, r3, r6    ; decrement number of integers..
        beq end             ; ..and exit if zero
        mov r5, #0           ; output zero
        mov r0, r2           ; load file handler
        swi 0x6c             ; read integer from file
        cmp r0, #0           ; if input==0..
        beq machine_q2       ; ..jump to state q2
        b machine_q0         ; else jump to state q0
machine_q2:
        subs r3, r3, r6    ; decrement number of integers..
        beq end             ; ..and exit if zero
        mov r5, #0           ; output zero
        mov r0, r2           ; load file handler
        swi 0x6c             ; read integer from file
        cmp r0, #1           ; if..
        beq machine_q3       ; ..input==1 jump to state q3
        bgt machine_q0       ; ..input==2 jump to state q0
        blt machine_q2       ; ..input==0 stay in current state
machine_q3:
        subs r3, r3, r6    ; decrement number of integers..
        beq end             ; ..and exit if zero
        mov r0, r2           ; load file handler
        swi 0x6c             ; read integer from file
        cmp r0, #0           ; if..
        bne non_zero         ; ..input!=0 jump to non_zero
        mov r5, #1           ; else output one
        b machine_q1         ; and jump to state q1
non_zero:
        cmp r0, #1           ; if..
        bne non_one          ; ..input!=1 jump to non_one
        mov r5, #0           ; else output zero
        b machine_q0         ; and jump to state q0
non_one:
        mov r5, #2           ; output two
        b machine_q0         ; jump to state q0
end:
        ;; end
        swi 0x11             ; exit
.end

```



Esame di Architetture degli Elaboratori

Soluzione

A.A. 2017-18 — IV appello — 3 luglio 2018

N.B.: il punteggio associato ad ogni domanda è solo una misura della difficoltà, e peso, di ogni domanda. Per calcolare il voto complessivo bisogna normalizzare a 32 (circa).

- Convertire il valore $AA.\overline{F}_{16}$ alla base 4.

R: (3 pt) Trattandosi di basi che sono una potenza di due, conviene dapprima convertire alla base 2 e poi raggruppare le cifre risultanti per riscrivere il valore nella base 4:

$$AA.\overline{F}_{16} = \underbrace{1010_2}_{A_{16}} \underbrace{1010_2}_{A_{16}} \cdot \underbrace{\overline{1111}_2}_{\overline{F}_{16}} = \underbrace{2_4}_{10_2} \underbrace{2_4}_{10_2} \underbrace{2_4}_{10_2} \underbrace{2_4}_{10_2} \cdot \underbrace{\overline{33}_4}_{\overline{1111}_2} = 2222.\overline{3}_4.$$

- Sono date le seguenti codifiche già in complemento a due a 8 bit: $n_1 = 00001010$, $n_2 = 11110101$. Se esiste, si calcoli il prodotto $n_1 \cdot n_2$ e se possibile lo si esprima in codifica *complemento a uno*.

R: (3 pt) Conviene calcolare $n_1 \cdot |n_2|$ e poi ricordare che n_2 codifica un valore negativo. Dunque, cambiando di segno n_2 si ha $-n_2 = 00001011$. Il calcolo del prodotto $n_1 \cdot |n_2|$ porge

$$\begin{array}{r} 00001010 \ x \\ 00001011 \\ \hline - \\ 00001010 \\ 00001010 \\ 00000000 \\ 00001010 \\ \hline - \\ 00001101110 \end{array}$$

che, restituendo il segno negativo in codifica complemento a uno, diventa 10010001.

- [INF] Fornire il risultato dell'esercizio precedente in codifica *floating point* IEEE 754 a 32 bit.

R: (3 pt) Il risultato trovato sopra può essere subito messo nella forma $-1.10111E6$. La codifica richiesta avrà dunque bit di segno asserito, esponente uguale a $127+6 = 133 = 10000101_2$ e infine mantissa uguale a 10111. Sistemando sui 32 bit previsti dallo standard IEEE 754 e convertendo alla base esadecimale:

$$\begin{array}{ccccccccccccc} 1 & | & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & \dots \\ & c & | & 2 & | & d & | & c & | & 0 & \dots \end{array}$$

da cui la codifica richiesta: 0xC2DC0000.

- La legge di Moore afferma che il numero di transistor nell'unità di memoria quadruplica ogni tre anni. Se un chip di memoria di forma quadrata di 256 MB oggi misura mediamente 1×1 cm, quale sarebbe la capacità di un chip delle stesse dimensioni tra 9 mesi se non esistessero vincoli di standardizzazione necessari per la compatibilità col resto dell'architettura?

R: (3 pt) Se il numero di transistor nell'unità di memoria quadruplica ogni 36 mesi allora il raddoppio della densità avviene ogni 18 mesi, e quindi dopo 9 mesi la memoria cresce di un fattore $\sqrt{2}$ a parità di dimensioni. In definitiva dunque il chip in questione avrebbe capacità uguale a $\sqrt{2} \cdot 256 = 362$ MB.

5. È data la tabella a destra, in cui C, D, E, F e A, B sono rispettivamente i 4 ingressi e le 2 uscite di una rete booleana. Realizzare ciascuna uscita con un numero minimo di porte, esclusivamente binarie, scelte tra AND, NAND, OR, NOR e XOR. Quante porte occorrono per realizzare la rete in questione?

C	D	E	F	A	B
1	0	0	0	1	0
0	0	1	0	1	0
0	1	0	0	1	1
0	0	0	1	1	0

R: (3 pt) Si ha: $A = \overline{C}D\overline{E}\overline{F} + \overline{C}\overline{D}\overline{E}F = \overline{C}\overline{E}(D\overline{F} + \overline{D}F) = \overline{C} + \overline{E}(D\overline{F} + \overline{D}F)$.

Analogamente: $B = \overline{C}\overline{D}E\overline{F} + \overline{C}D\overline{E}\overline{F} = \overline{C}\overline{F}(\overline{D}E + D\overline{E}) = \overline{C} + \overline{F}(\overline{D}E + D\overline{E})$.

Sia A che B sono realizzate unendo le uscite da una porta NOR e una XOR attraverso una porta AND. Quindi in totale occorrono 6 porte, 3 per ciascuna uscita.

Questo se si assumeva che le uscite per gli ingressi non specificati fossero nulle. Se le stesse si fossero considerate indeterminate, allora in totale erano sufficienti 2 porte XOR: $A = D\overline{F} + \overline{D}F$ e $B = \overline{D}E + D\overline{E}$.

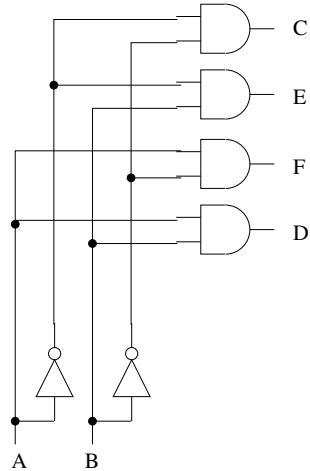
6. [INF] Si realizzi la rete all'esercizio sopra rinunciando alla porta AND. Quante porte occorrono stavolta?

R: (3 pt) Ricordando l'uguaglianza $E = \overline{\overline{EE}}$, è sufficiente sostituire la porta AND con due NAND per ottenere lo stesso risultato. Quindi, in totale occorrono 8 porte per realizzare le due uscite A e B.

Anche qui, se si assumeva che le uscite per gli ingressi fossero indeterminate allora in totale erano sufficienti le stesse 2 porte XOR: $A = D\overline{F} + \overline{D}F$ e $B = \overline{D}E + D\overline{E}$.

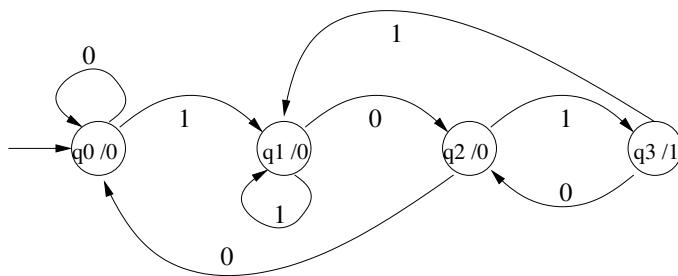
7. Invertire la tabella (cioè trasformare gli ingressi in uscite e viceversa) dell'esercizio 5. Realizzare la nuova tabella così ottenuta adoperando un *decodificatore*, il cui circuito combinatorio dev'essere chiaramente esplicitato all'interno della rete booleana proposta.

R: (3 pt)



8. [INF] Disegnare il diagramma della macchina di Mealy oppure di Moore che, leggendo simbolo dopo simbolo una sequenza indefinitamente lunga di caratteri appartenenti all'alfabeto binario $\mathcal{B} = \{0, 1\}$, riconosce *ogni* istanza delle due sottosequenze 1010 e 1011 all'interno della sequenza letta. Per esempio, la sequenza 10101011010 contiene *una* istanza della sottosequenza 1011 e *tre* istanze della sottosequenza 1010. A ogni carattere letto la macchina in questione genera le seguenti uscite: 0 se non ha riconosciuto alcuna sottosequenza; 1 nell'istante in cui riconosce la sottosequenza 1010 o la sottosequenza 1011.

R: (3 pt) Poichè la sequenza è indefinitamente lunga siamo certi che, dopo ogni sottosequenza 101, la macchina *sicuramente* genererà un'uscita uguale a 1. Dunque, per esempio sceglieremo la macchina di Moore schematizzata qui sotto:



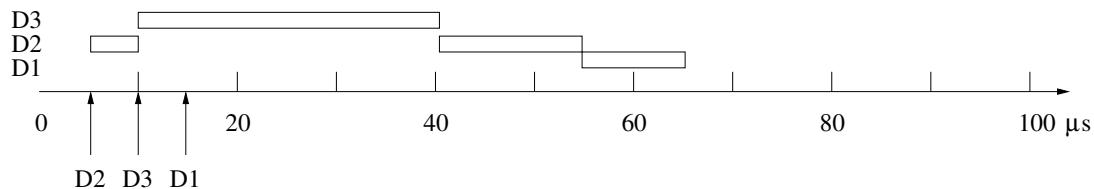
Alternativamente è corretto anche realizzare la macchina di Mealy che restituisce 1 in corrispondenza di ogni simbolo letto dopo avere riconosciuto la sottosequenza 101.

9. Si vogliono codificare le 21 lettere dell'alfabeto italico A, B, ..., Z adoperando un codice binario a lunghezza variabile, il quale ottimizzi l'efficienza della codifica delle vocali A, E, I, O, U minimizzando la lunghezza dei rispettivi codici. Si dia un codice che rispetta questo requisito.

R: (3 pt) Il codice binario dev'essere lungo almeno 5 bit. Essendo le vocali 5, le 16 consonanti possono essere codificate dai 4 bit meno significativi premessi per esempio dallo 0: 00000, 00001, ..., 01111. A questo punto ottimizziamo i codici delle vocali, che devono iniziare con 1: poiché occorrono almeno 3 bit per codificare 5 simboli, i cinque codici possono essere i seguenti: 1000, 1001, 1010, 1011, 1100. Infine, poiché non esistono due codifiche che iniziano con 11 l'ultimo codice può essere accorciato in 11.

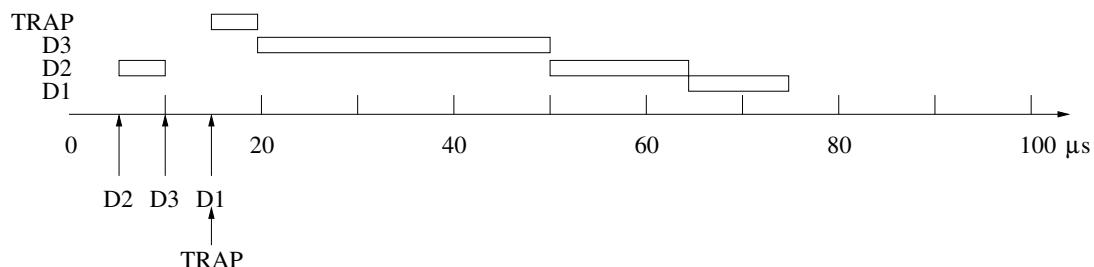
10. In un protocollo I/O di tipo controllato, tre dispositivi D1, D2 e D3 con priorità di servizio crescente competono per essere serviti dalla CPU. Il primo è servito in $10 \mu\text{s}$, il secondo in $20 \mu\text{s}$ e il terzo in $30 \mu\text{s}$. Posto a $t = 0 \mu\text{s}$ l'attimo in cui il sistema entra in funzione, accade che: a $t = 5 \mu\text{s}$ D2 inoltra una richiesta di servizio; a $t = 10 \mu\text{s}$ D3 inoltra una richiesta di servizio; a $t = 15 \mu\text{s}$ D1 inoltra una richiesta di servizio. Si tracci un diagramma temporale che illustra come e quando le richieste vengono servite dalla CPU.

R: (3 pt)



11. Si modifichi il diagramma tracciato in precedenza, tenendo presente che 1) a $t = 15 \mu\text{s}$ si verifica una *trap* a massima priorità di durata $5 \mu\text{s}$, e che 2) la CPU solo nel caso di servizi verso dispositivi esterni impiega $5 \mu\text{s}$ per sospendere un servizio a priorità più bassa a fronte della necessità di servire un dispositivo a priorità maggiore.

R: (3 pt)



12. Una memoria fisica di 4 Gbyte è paginata in pagine di 32 kB, che possono trovare posto in posizioni multiple della lunghezza della pagina all'interno di una memoria principale di 4 Mbyte. Non è prevista la presenza di memoria virtuale. Qual è la dimensione della *page table*?

R: La memoria fisica, di $2^{32} = 4$ Gbyte, necessita di 32 bit per essere indirizzata. Di essi, 15 individuano l'*offset* all'interno di ogni pagina di $2^{15} = 32$ kB. Poiché ogni pagina trova posto dentro la memoria principale in una posizione multipla di 32 kB, allora in uno spazio di $2^{22} = 4$ Mbyte possono trovare posto

$2^{22}/2^{15} = 2^7 = 128$ pagine, le quali in definitiva necessitano di 7 bit per essere indirizzate. La *page table* dunque deve possedere $2^{32-15} = 2^{17}$ locazioni corrispondenti alla parte più significativa dell'indirizzo. Ciascuna di esse dovrà contenere 7 bit per indirizzare ciascuna pagina in uno dei 128 possibili slot, più il bit di presenza/assenza della pagina in memoria principale. La dimensione della *page table* dunque è di $2^{17} \times 8$ bit, ovvero di 128 kB.

13. Si spieghi la differenza esistente tra gli indirizzamenti *immediato*, *diretto* e *indiretto*, possibili argomenti del set di istruzioni a bordo del Core i7.

R: (3 pt) L'indirizzo immediato è interpretato dalla CPU come un valore numerico; l'indirizzo diretto è interpretato come l'indirizzo di una locazione di memoria contenente un valore numerico; l'indirizzo indiretto infine è interpretato come l'indirizzo di una locazione di memoria, tipicamente il nome di un registro, in cui leggere l'indirizzo di una locazione di memoria contenente un valore numerico.

14. [INF] Con riferimento all'esercizio 8, scrivere un programma in assembly per ARM il quale legge dal file di testo *file.txt* una sequenza di caratteri appartenenti all'alfabeto $\mathcal{B} = \{0,1\}$ e quindi realizza la macchina che riconosce le sottosequenze colà descritte. L'output è scritto a ogni ciclo di lettura nel registro **r4**. In più, il programma mantenga nel registro **r7** la somma delle sottosequenze 1010 trovate e nel registro **r8** la somma delle sottosequenze 1011 trovate. Il file di testo per comodità riporta nella prima riga il numero di caratteri presenti nel file a partire dalla seconda riga. Il programma termina allorquando il file è stato letto interamente.

R: (9 pt)

```

.data
stringa:
    .asciiz "input.txt"
    .text
main:
    ldr r0, =stringa      ; file name address in r0
    mov r1, #0              ; read mode
    swi 0x66                ; open file in read mode
    mov r2, r0                ; save file handler
    swi 0x6c                ; read number of integers
    mov r3, r0                ; save number of integers in r3
    mov r6, #1                ; r6 loaded with one
    mov r7, #0                ; reset r7
    mov r8, #0                ; reset r8

machine_q0:
    subs r3, r3, r6          ; decrement number of integers..
    beq end                  ; ..and exit if zero
    mov r4, #0                ; output zero
    mov r0, r2                ; load file handler
    swi 0x6c                ; read integer from file
    cmp r0, #1                ; if input==1..
    beq machine_q1            ; ..jump to state q1
    b machine_q0               ; else stay in current state

machine_q1:
    subs r3, r3, r6          ; decrement number of integers..
    beq end                  ; ..and exit if zero
    mov r4, #0                ; output zero
    mov r0, r2                ; load file handler
    swi 0x6c                ; read integer from file
    cmp r0, #0                ; if input==0..
    beq machine_q2            ; ..jump to state q2
    b machine_q0               ; else jump to state q0

machine_q2:
    subs r3, r3, r6          ; decrement number of integers..
    beq end                  ; ..and exit if zero
    mov r4, #0                ; output zero
    mov r0, r2                ; load file handler

```

```
        swi 0x6c          ; read integer from file
        cmp r0, #1         ; if input==1
        beq machine_q3     ; ..jump to state q3
        b machine_q0        ; else jump to state q0
machine_q3:
        subs r3, r3, r6    ; decrement number of integers..
        beq end             ; ..and exit if zero
        mov r4, #1           ; output one
        mov r0, r2           ; load file handler
        swi 0x6c          ; read integer from file
        cmp r0, #0           ; if..
        bne non_zero        ; ..input!=0 jump to non_zero
        add r7, r7, #1       ; else count up 0010 in r7
        b machine_q2        ; and jump to state q2
non_zero:
        add r8, r8, #1       ; count up 0011 in r8
        b machine_q1        ; and jump to state q1
end:
        ;; end
        swi 0x11          ; exit
.end
```



Esame A di Architetture degli Elaboratori

Soluzione

A.A. 2017-18 — V appello — 25 settembre 2018

N.B.: il punteggio associato ad ogni domanda è solo una misura della difficoltà, e peso, di ogni domanda. Per calcolare il voto complessivo bisogna normalizzare a 32 (circa).

- Convertire il numero $333.\bar{3}_{10}$ alla base 3.

R: (3 pt) $333.\bar{3}_{10} = 110100.1_3$. Infatti,

$$\begin{array}{r}
 333 \mid 3 & .333\dots \mid 3 \\
 \hline
 111 \mid 0 & 0 \mid 1 \\
 37 \mid 0 & \\
 12 \mid 1 & \\
 4 \mid 0 & \\
 1 \mid 1 & \\
 0 \mid 1 &
 \end{array}$$

- Sono date le seguenti codifiche in complemento a 2 a 7 bit: $n_1 = 1001000$, $n_2 = 1111010$. Si calcoli la somma $n_1 + n_2$, si dica quanto vale e se possibile la si esprima nella stessa codifica.

R: (3 pt) Si può direttamente riprodurre i calcoli realizzati da un’ipotetica ALU che opera in aritmetica in complemento a 2 a 7 bit:

$$\begin{array}{r}
 1001000 + \\
 1111010 = \\
 \hline
 1000010
 \end{array}$$

Complementando a 2 il risultato si ottiene 0111110 , equivalente al valore 62. La somma cercata vale dunque -62 . Per la validità della codifica associata si veda l’esercizio 4.

- [INF] Fornire il risultato dell’esercizio precedente in codifica *floating point* IEEE 754 a 32 bit.

R: (3 pt) Il risultato trovato sopra può essere subito messo nella forma $-1.1111E5_2$. La codifica richiesta avrà dunque bit di segno asserito, esponente uguale a $127+5 = 132 = 10000100_2$ e infine mantissa uguale a 1111 . Sistemando sui 32 bit previsti dallo standard IEEE 754 e convertendo alla base esadecimale per semplicità di lettura:

$$\begin{array}{r}
 1|1\ 0\ 0\ 0\ 0\ 1\ 0\ 0|1\ 1\ 1\ 1\ 0\ 0\ 0\ 0\dots \\
 C \quad | \quad 2 \quad | \quad 7 \quad | \quad 8 \quad | \quad 0\dots
 \end{array}$$

da cui la codifica richiesta: $0xC27800000$.

- Con riferimento all’esercizio 2, si spieghi come la ALU decide la validità del risultato di un’operazione di somma in complemento a 2.

R: (3 pt) Se i termini da sommare differiscono nel bit più significativo allora hanno segno diverso e non possono dare luogo a *overflow*. Altrimenti, se il bit più significativo del risultato è diverso da quello dei due termini da sommare allora c’è certamente *overflow*, viceversa non può esserci *overflow*.

- Adoperando le regole di equivalenza booleana, calcolare quanto vale E nell’espressione seguente:

$$E = ABC \cdot \overline{AC}$$

R: (3 pt) Sfruttando le regole di De Morgan si ha $E = ABC \cdot (\overline{A} + \overline{C}) = A\overline{ABC} + ABCC = 0 + ABC$.

6. [INF] Verificare il risultato ottenuto sopra con una mappa di Karnaugh.

R: (3 pt)

BC	00	01	11	10
A				

0	0	0	0	0
1	0	0	1	0

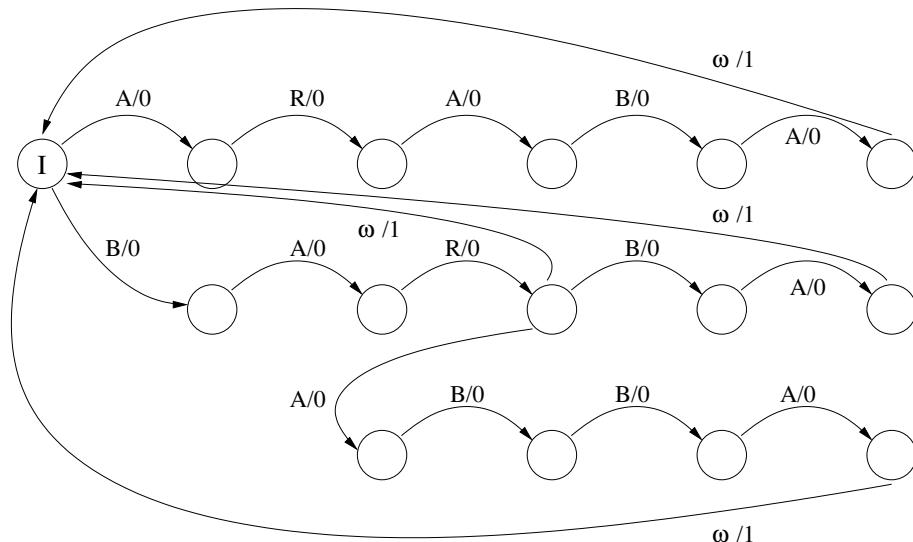
da cui $E = ABC$.

7. Si può realizzare l'espressione all'esercizio 5 avendo a disposizione solamente delle porte NOR a 3 ingressi? Motivare la risposta.

R: (3 pt) Sì. Con 4 porte: $ABC = \overline{A + B + C} = \overline{A + A + A + B + B + B + C + C + C}$.

8. [INF] Disegnare il diagramma della macchina di Mealy che, leggendo simbolo dopo simbolo una sequenza indefinitamente lunga di caratteri appartenenti all'alfabeto $\mathcal{A} = \{A, B, R, \omega\}$, riconosce le parole ARABA, BAR, BARABBA, BARBA. Il simbolo ω è adoperato esclusivamente per terminare ogni parola presente nella sequenza: per esempio, la sequenza ARABABA ω BAR $\omega\omega$ ARBA ω contiene la sola parola riconoscibile BAR. Per semplicità si omettano dal disegno tutti gli archi percorsi in presenza di parole non riconoscibili e le etichette degli stati non iniziali.

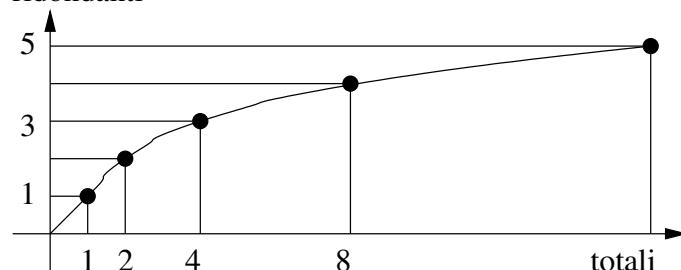
R: (3 pt) Se tutti gli archi percorsi in presenza di parole non riconoscibili producono in uscita per esempio il valore 2, allora la macchina disegnata qui sotto riconosce solo le parole richieste, che vengono segnalate da una sequenza di zeri terminata da 1, e non tutte le parole che terminano con quelle richieste (per esempio RRRBAR ω).



9. Si tracci una funzione che esprime il rapporto $\frac{\text{dati ridondanti}}{\text{dati totali}}$ dei codici di Hamming, fermandosi al valore in ascissa dati totali = 16.

R: (3 pt) Per esempio:

ridondanti



10. Un protocollo I/O di tipo programmato è sempre meno efficiente di uno di tipo controllato? Si motivi la risposta.

R: (3 pt) Poichè la gestione dell'interruzione in un protocollo I/O di tipo programmato è più semplice, rapida ed economica dell'interruzione in un protocollo I/O di tipo controllato, allora il primo può essere più efficiente del secondo a patto che la richiesta di servizio arrivi in modo periodico e altamente frequente, in modo da soddisfare tutte (o quasi tutte) le interruzioni.

11. Che cos'è il *bus skew*?

R: (3 pt) Il *bus skew* è l'arrivo temporalmente disallineato dei dati alla CPU, a causa di un problema di cattiva sincronizzazione presente nei canali che formano il bus.

12. Quali registri della CPU rendono possibile il costrutto di programmazione noto come *chiamata a procedura*?

R: (3 pt) I registri **lr** (*link register*) e **sp** (*stack pointer*).

13. In una regione di memoria segmentabile a 8 bit, grande 1 MB, trovano posto segmenti tutti uguali di 8 kB. Nel tentativo di scrivere un nuovo segmento in quella regione, il sistema operativo trova la memoria piena. Quanti segmenti stanno in quel momento occupando la regione in questione nel caso di occupazione in assoluto più efficiente? Quanti nel caso di occupazione in assoluto meno efficiente? Qual è la percentuale di utilizzo di quella regione nei due casi?

R: Se è piena, allora la regione di memoria in questione non contiene spazi liberi contigui di 8 kB. Nel caso di occupazione in assoluto più efficiente essa è occupata da $1024/8 = 128$ segmenti, corrispondenti a una percentuale di utilizzo del 100%, mentre nel caso di occupazione in assoluto meno efficiente esiste uno spazio di 8 kB meno 1 byte tra ogni coppia di segmenti occupati di 8 kB adiacenti. In questo caso, dunque, i segmenti presenti sono $2^{20}/(8192 + 8191) \approx 64.004$. In altre parole, i 64 segmenti occupano in totale $64 \cdot (8192 + 8191) = 1048512$ byte. La memoria libera ma inutilizzabile ammonta quindi al totale delle locazioni di 1 byte lasciate vuote: $64 \cdot 8191 + (2^{20} - 1048512) = 64 \cdot 8191 + 64 = 524288$ byte, equivalente esattamente al 50% della regione considerata.

14. [INF] Scrivere un programma in assembly per ARM il quale calcola la somma S dei primi $n + 1$ termini di una *serie geometrica* di ragione naturale positiva r : $S = \sum_{i=0}^n r^i$. Per esempio, se $r = 2$ e $n = 4$ avremo

$$S = \sum_{i=0}^4 2^i = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 = 1 + 2 + 4 + 8 + 16 = 31.$$

Per convenzione il programma legge r e n da due *word* consecutivi in memoria, e li carica rispettivamente nei registri **r1** e **r3**. Inoltre mantiene il registro **r4** per il risultato. Infine, salva il risultato nel word in cui inizialmente compariva r . Per semplicità si assuma in ogni caso $r \geq 0$ e $n \geq 0$, e non ci preoccupi del possibile *overflow* del risultato.

R: (9 pt)

```

.data
inout:
    .word 2,6           ; root, power
    .text
main:
    ldr r0, =inout      ; memory addr in r0
    ldr r1, [r0], #4     ; load root in r1, update r0
    mov r2, #1           ; set r2 to initial multiplier
    mov r4, #1           ; set result in r4 to power^0 = 1
    ldr r3, [r0]          ; load power in r3
    cmp r3, #0           ; if power null..
    beq fine            ; ..then go to end
loop:
    mul r2, r1, r2       ; r2 = r2*r1 = r1^n
    add r4, r4, r2       ; r4 = r4+r2

```

```
    subs r3, r3, #1          ; decrement power in r3
    bne loop                 ; loop again if power is not zero
fine:
    ldr r0, =inout           ; memory addr in r0
    str r4, [r0]              ; save output

    swi 0x11                 ; end
.end
```

Architettura degli elaboratori

Docenti:

- Federico Fontana - federico.fontana@uniud.it
- Yuri De Pra - depra.yuri@spes.uniud.it



Obiettivi del corso

- Spiegare come sono fatti e come funzionano **fisicamente** i calcolatori
- Illustrare i componenti base e le loro interazioni
- Nella tradizionale distinzione **hardware–software**:
questo è un corso di **hardware**.

Organizzazione: sdoppiamento

Il corso è diversificato tra Informatica e Internet of Things, Big Data & Web.

- **Informatica**: 12 CFU = 6 CFU di lezione + 3 CFU di laboratorio.
- **IBW**: 6 CFU di lezione.

Motivazione:

- i due corsi di laurea hanno obiettivi differenti.

Organizzazione: lezioni

- Alcuni argomenti (sistemi sequenziali, programmazione in Assembly) sono presentati solo a Informatica.
- Il laboratorio è svolto solo a Informatica.
- L'esame è differenziato.
- **INF**: 6 ore di lezione + 2 ore di laboratorio a settimana (lunedì, martedì e giovedì).
IBW: 4 ore di lezione a settimana (lunedì e martedì).
- Orario di ricevimento (meglio previa email): mercoledì 15:30 – 17:30, o quando mi trovate in studio (il venerdì se posso lavoro a PN).

Organizzazione esame

- Solo **scritto!**
- INF: 14 domande in 2h45m per 48 punti da riproporzionare in trentesimi + lode.
- IBW: 10 domande in 1h40m per 30 punti da riproporzionare in trentesimi + lode.
- Tipicamente: $\approx 30\%$ domande di **teoria**, $\approx 70\%$ **esercizi** da risolvere (!).
- Lo storico (un anno, 5 appelli) è reperibile del sito di e-learning. Durante il corso vedremo esercizi simili a quelli dati all'esame, ma **non c'è** una struttura univoca dello scritto.

Laboratorio - solo Informatica

- Solo per Informatica.
- Docente di riferimento: Yuri De Pra.
- Ma quasi sempre ci sarò anch'io.
- Inizio: giovedì 11 ottobre.
- 3 crediti: 24 ore di laboratorio (2 ore a settimana).
- È necessario registrarsi (ora), per poter accedere ai calcolatori
modulo scaricabile dalla vecchia pagina web:
[web.uniud.it/didattica/facolta/scienze/
info_dida/laboratorio/](http://web.uniud.it/didattica/facolta/scienze/info_dida/laboratorio/)
da consegnare nella stanza tra Lab1 e Lab2.

Contenuti

Corso in parte nozionistico - descrittivo:

- si descrivono le diverse parti del calcolatore e come interagiscono
- vengono presentati principi di funzionamento e idee base
- non sono dettagliati né spesso tecnologicamente all'avanguardia
- le nozioni vanno **comprese** in vista dell'esame, non solo memorizzate:
 - capire i meccanismi di funzionamento delle diverse componenti,
 - costruirsi una visione generale, collegare tra loro le diverse nozioni.

Contenuti

Aspetti più progettuali:

- rappresentazione dei numeri
- progettazione di circuiti logici e sequenziali
- organizzazione della memoria
- programmazione assembly del processore ARM.

Obiettivo di queste parti del corso: insegnare a *fare*, farvi acquisire abilità.

Progettazione circuiti e programmazione assembly vengono trattati anche a laboratorio.

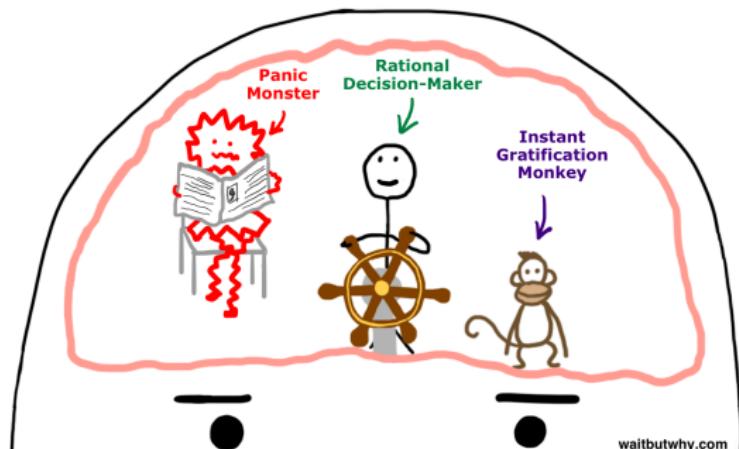
Considerazioni generali

- L'università fornisce pochi stimoli a studiare durante l'anno (interrogazioni, compiti in classe).
- Si richieda una maggiore maturità.
- È facile perdersi: 30% degli iscritti al primo anno non si iscrive al secondo.
- Tra le valutazioni a Fontana: "Si rifiuta di usare il microfono, non si sente nulla!"
Risposta: "Hai ragione. Alla sagra, dal palco adopero il microfono anch'io".

Pericolo: procrastinare lo studio

Un modello del cervello

My Brain in High School



Dal blog www.waitbutwhy.com,
Why procrastinators procrastinate.

Psicologia pratica

Fissare una serie di obiettivi a breve termine, verificabili.

Obiettivi a lungo termine o generali sono inutili.

- Ogni settimana seguire il 90% delle lezioni.
- Ripassare ogni giorno gli argomenti trattati a lezione.
- Ripassare a sufficienza gli argomenti trattati in modo da seguire agevolmente le lezioni.
- Da parte mia cercherò di mettere i lucidi nel web e-learning prima della lezione.
- (Se serve **chiedetemi** di usare il microfono).

Per questo corso

- Distribuire il carico di lavoro, non rimandare tutto alle due settimane precedenti l'esame.
- Non perdere il contatto con ciò che viene presentato a lezione.
- Per Informatica: seguire le lezioni di laboratorio.
- Sfruttate l'orario di ricevimento e, in generale, i servizi di tutorato.

Libro di testo

A. S. Tanenbaum, T. Austin *Architettura dei calcolatori - Un approccio strutturale*. Pearson, 2018.

È possibile utilizzare anche le edizioni precedenti del libro di testo, solo poche parti sono state aggiornate nella nuova edizione.

- L'autore è un autorità nella didattica dell'informatica e nel campo dei sistemi operativi. Insegnante di Linus Torvalds e autore di MINIX, un precursore di Linux.
- Il corso segue abbastanza da vicino il libro di testo.
- Non così aggiornato sulle tecnologie hardware in uso oggi.

Libro di testo

- Molto discorsivo. Vengono evitati gli argomenti difficili, tecnici, quantitativi.
- Ordine di presentazione degli argomenti un po' controintuitivi:
 - capitolo 2: descrizione superficiale del calcolatore, riassunto di tutti gli argomenti;
 - capitoli successivi: descrizione dettagliata.

Il portale dell'e-learning

- Occorre iscriversi all'insegnamento specifico!
- Lucidi presentati a lezione
- Pagine supplementari
- Esercizi svolti
- Testi d'esame.

È **molto** utile seguire la presentazione delle slide a lezione, integrando con appunti.

Azzeramento dei concetti

Elaboratore: macchina capace di eseguire una sequenza di istruzioni semplici (**istruzioni macchina**).

Programma: nel nostro corso, **sequenza** di istruzioni macchina.

I programmi, e i **dati** su cui il programma lavora sono organizzati in *file* che risiedono nella **memoria** del calcolatore.

Memoria

La memoria del calcolatore è di due tipi:

- operativa (**memoria principale, memoria RAM**): contiene i dati durante la loro elaborazione
- di massa (**memoria secondaria, disco rigido**): contiene i dati non in elaborazione, in modo permanente.

Memoria

La capacità delle memorie si misura in **byte**, o nei suoi multipli:

- **KB** (kilobyte = $2^{10} = 1024 \sim 1.000$ byte),
- **MB** (megabyte = $2^{20} \sim 1.000.000$ byte)
- **GB** (gigabyte = $2^{30} \sim 1.000.000.000$ byte).

Il byte è uno spazio di memoria che può assumere $2^8 = 256$ configurazioni distinte, in grado cioè di memorizzare un numero naturale tra 0 e 255, oppure una tra 256 diverse codifiche di un carattere (a, b, c, . . . , A, B, C, . . .), eccetera.

Processore

Il cuore dell'elaboratore è il **processore** o **CPU**, il circuito che fisicamente esegue le istruzioni.

Processore e memoria principale vengono realizzati mediante particolari circuiti chiamati **circuiti integrati** o **chip**. Un chip ha la dimensione di pochi centimetri e può contenere miliardi di *transistor*. Un solo chip è sufficiente per realizzare un processore.

Bus

Il processore riceve/trasmette i dati da/verso l'esterno attraverso le **periferiche**: schermo, tastiera, stampante, scheda di rete, scanner, lettore CD, . . . , Processore, memoria e periferiche sono collegati tra loro attraverso dei **bus**. L'insieme di tutte queste componenti forma il moderno **calcolatore**.

Sistema Operativo

L'interazione tra calcolatore e mondo esterno avviene tramite un particolare programma chiamato **sistema operativo**. Il sistema operativo è permanentemente in esecuzione e fornisce funzionalità agli altri programmi.

Attraverso il sistema operativo si possono eseguire **programmi applicativi**. Es.: web browser, word processor.

È ammessa una parziale ignoranza

- Il calcolatore oggi è un sistema troppo complesso per poterlo conoscere completamente.
- Un sistema di calcolo (calcolatore + software) è strutturato in maniera tale si possa interagire con esso conoscendo solo alcune funzionalità, e ignorando come queste sono realizzate.
- Lo studio e la progettazione di sistemi di calcolo possono essere affrontato a vari livelli (hardware, software).

Macchine virtuali

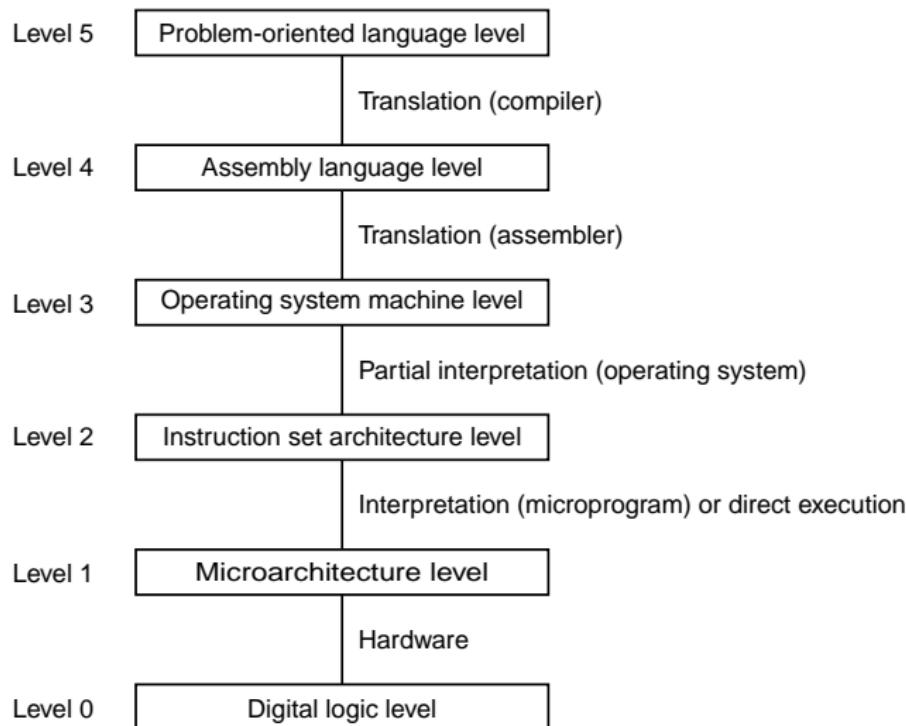
Uno dei concetti base dell'informatica.

Necessario per gestire l'enorme complessità di un sistema di calcolo.

Il calcolatore può essere visto come una **macchina a livelli**: un insieme stratificato di **macchine virtuali**.

Ciascuna fornisce una serie di funzionalità e viene realizzata a partire da una macchina virtuale sottostante (**livelli**).

Esempio di schema a livelli



Motivi per studiare l'hardware

- **Culturale:** avere una conoscenza complessiva dei sistemi di calcolo.
- **Utilità:**
 - valutare, scegliere o gestire l'hardware,
 - conoscere i fattori che determinano le prestazioni,
 - gestire i malfunzionamenti.

Più in dettaglio

- porte logiche
- circuiti logici, memorie
- processore
- programmazione assembly
- dispositivi periferici
- bus, collegamenti tra dispositivi
- calcolatori paralleli, sistemi multiprocessore.

Storia dei sistemi di calcolo

- 100 a.C. ? — Meccanismo di Anticitera



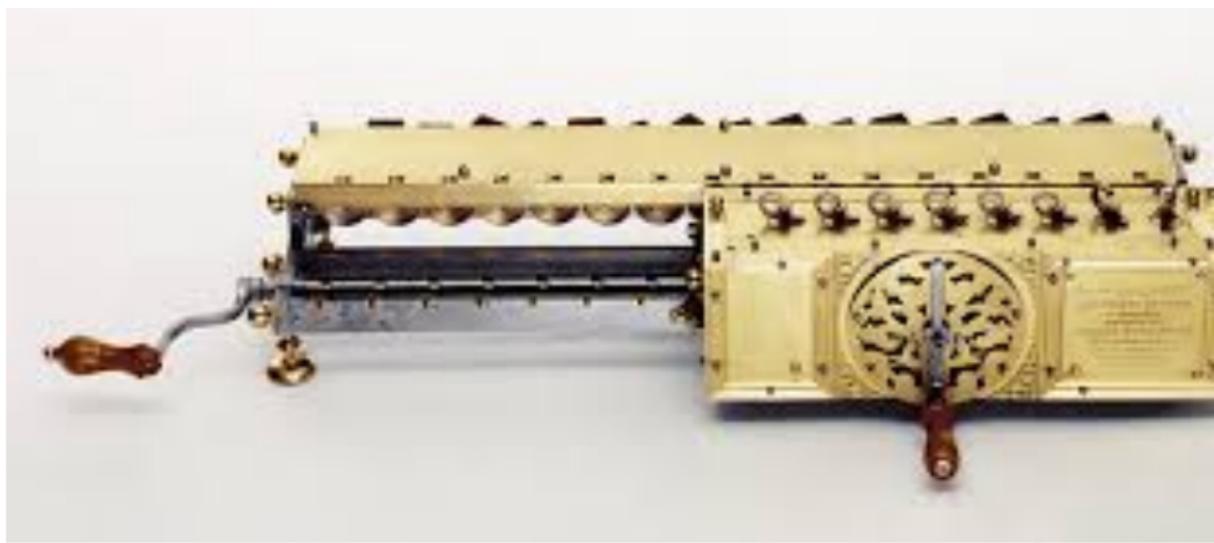
Storia dei sistemi di calcolo

- 1642 Pascal — calcolatrice meccanica per somma e sottrazione



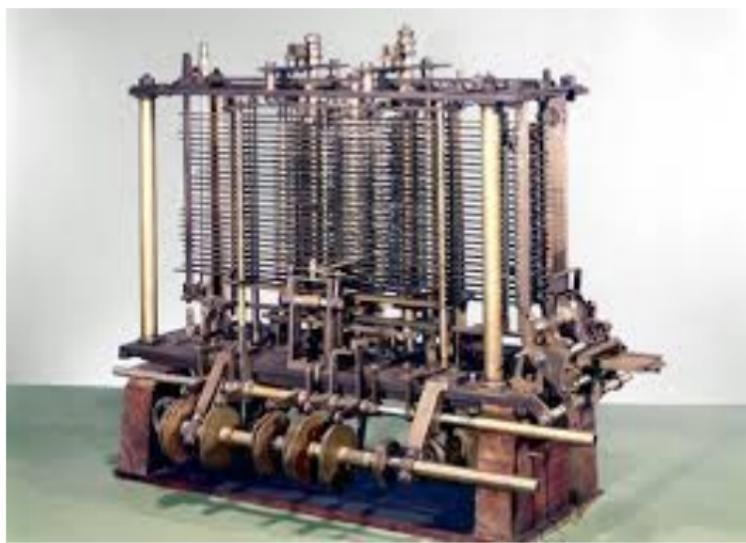
Storia dei sistemi di calcolo

- 1670 Leibniz — calcolatrice meccanica per prodotto e divisione



Storia dei sistemi di calcolo

- 1834 Babbage — calcolatrice meccanica programmabile



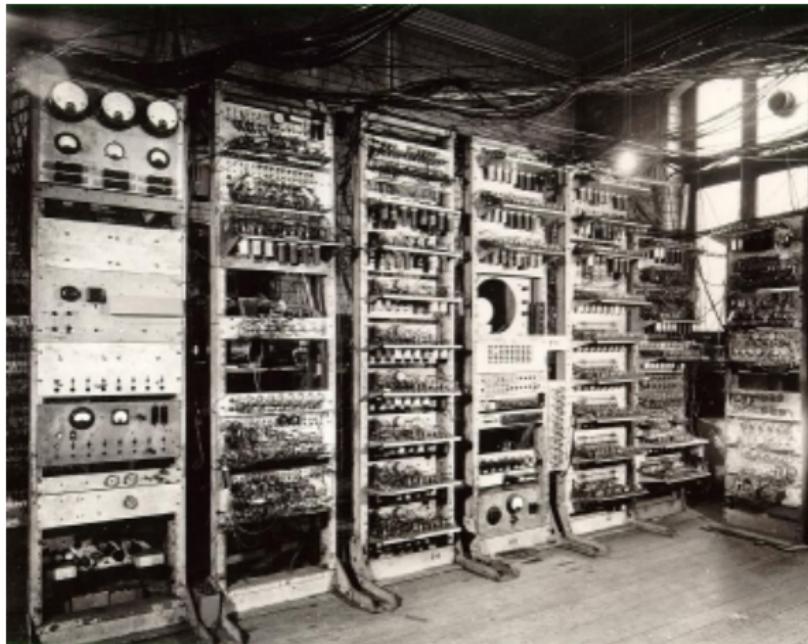
Storia dei sistemi di calcolo

- 1930 Zuse — Macchina calcolatrice a relè



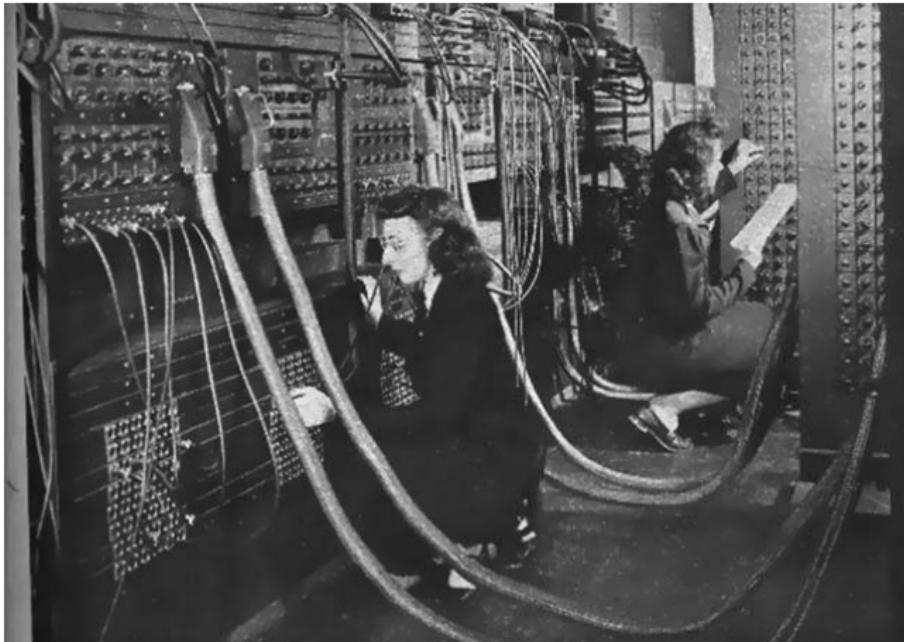
Storia dei sistemi di calcolo

- 1944 Aiken — Mark I — macchina programmabile a relè



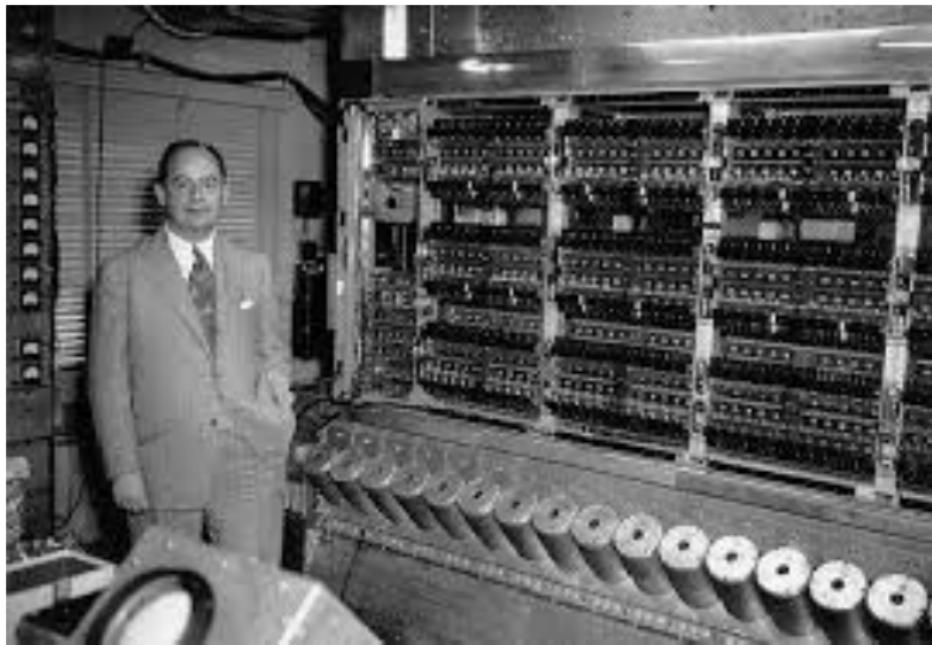
Storia dei sistemi di calcolo

- 1946 Eckert & Mauchly: Eniac — primo calcolatore a triodi (valvole)



Storia dei sistemi di calcolo

- 1951 Von Neumann — IAS — con architettura simile agli attuali computer



Storia dei sistemi di calcolo

- 1956 TX-0 — primo calcolatore interamente a transistor



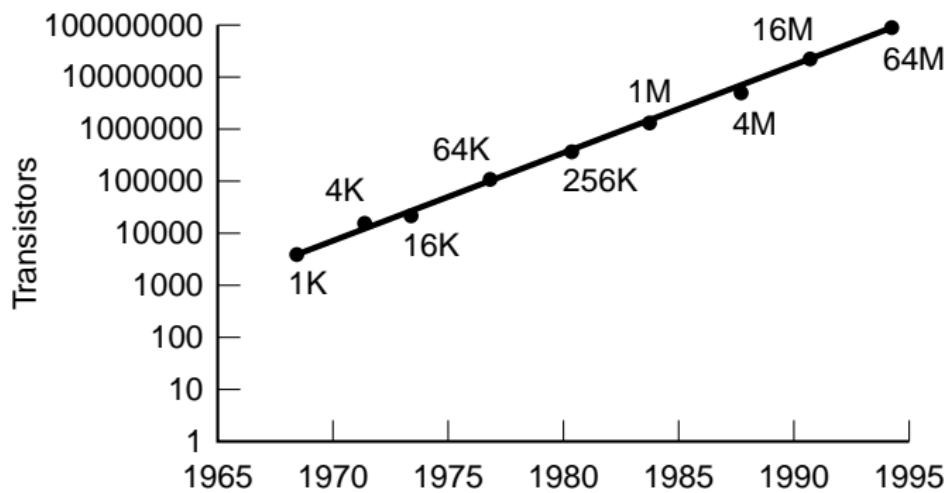
Storia dei sistemi di calcolo

- 1954 IBM — primi calcolatori a transistor
- 1958 — Calcolatori a circuiti integrati.

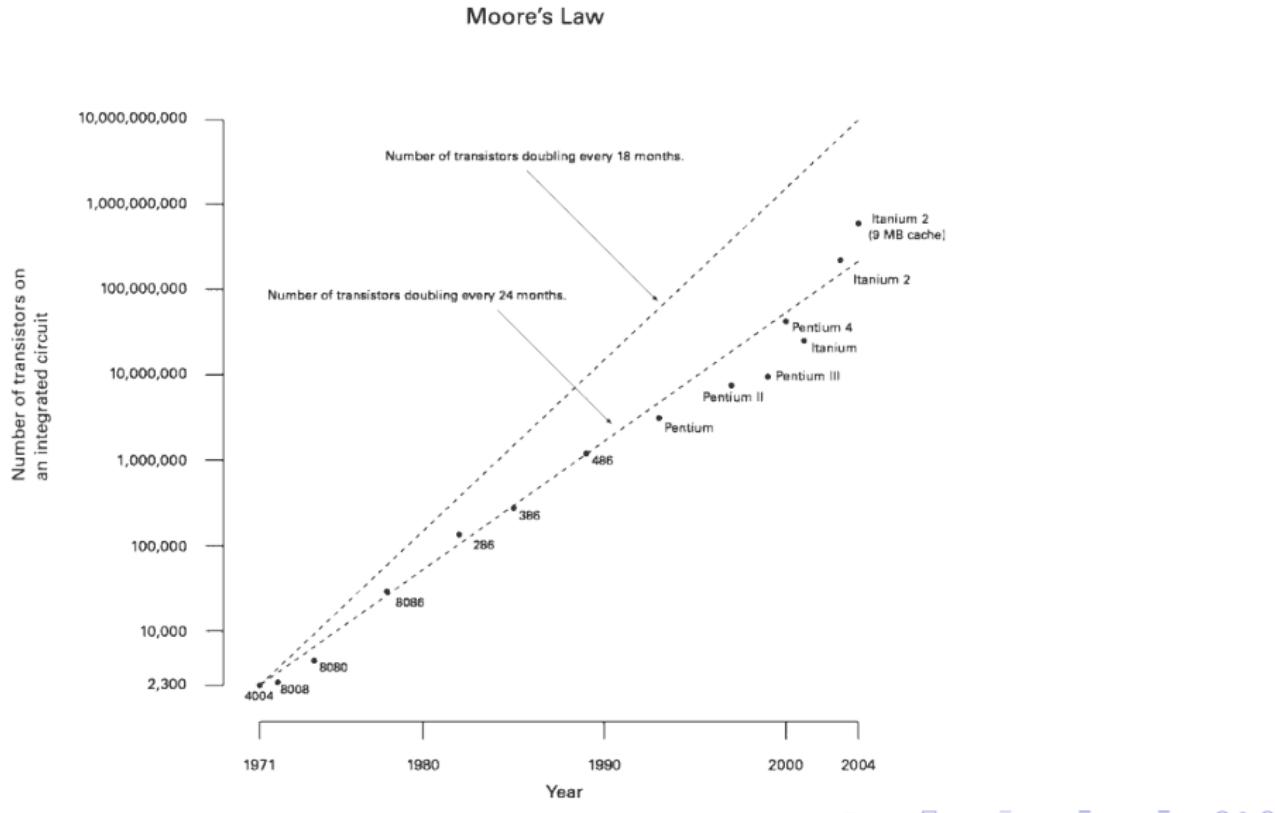


Progressi della tecnologia

Legge di Moore (Gordon Moore, 1965): il numero dei transistor all'interno di un chip (circuito integrato) di memoria quadruplica ogni tre anni, (raddoppia ogni 18 mesi).



Numero di transistor per processore



Progressi della tecnologia

Progressi analoghi anche per le altre componenti.

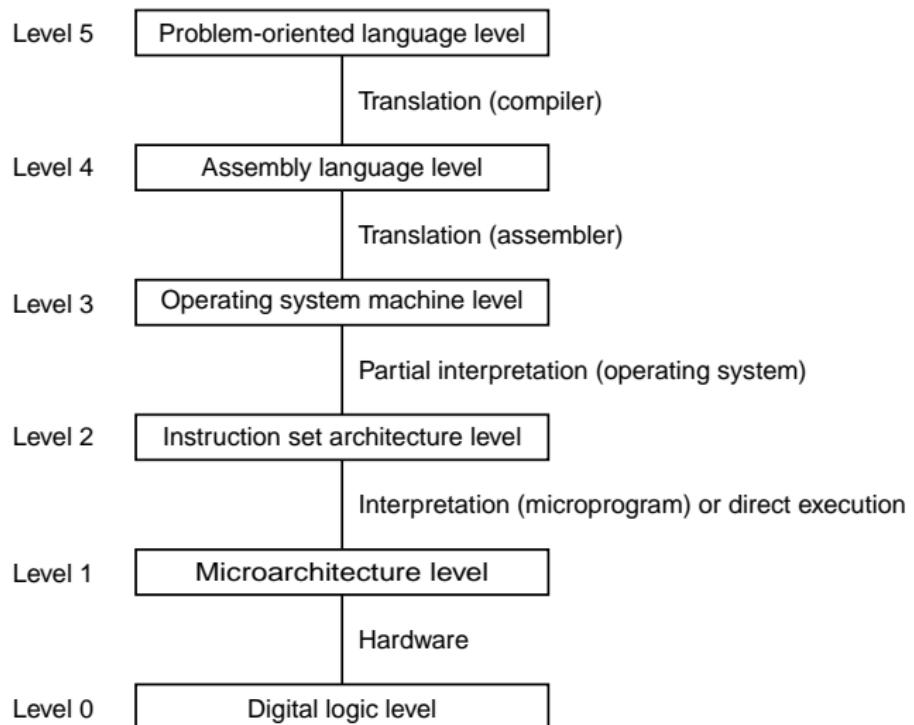
- capacità dei dischi magnetici, memorie flash.
- velocità dei bus e reti di interconnessione,
- risoluzione delle fotocamere e stampanti.

Pervasive computing

Ci si muove verso una frammentazione,
distribuzione dei sistemi di calcolo:

- Mainframe
- Minicomputer, server
- Workstation – Personal Computer
- Tablet – Smartphone
- Embedded Computer
- Internet of Things (IoT), Cloud.

Livello 0: logica binaria



Dominio analogico, dominio digitale

Nei calcolatori il segnale elettrico rappresenta l'informazione.

Due metodi per rappresentare informazione tramite segnali (es.: telefono vs. telegrafo):

- segnale modulato (**analogico**, ∞ livelli)
- segnale presente - assente (**digitale**, 2 livelli).

Il calcolatore moderno segue il paradigma digitale:

- tolleranza ai disturbi, affidabilità
- semplicità di progettazione
- rappresentazione astratta dell'informazione
- less is more.

Codifica dell'informazione binaria

Segnale digitale di tensione elettrica (Volt).

Logica positiva:

- tensione presente (2 - 5 Volt): valore 1
- tensione assente: (\sim 0 Volt): valore 0.

Logica negativa:

- tensione presente (2 - 5 Volt): valore 0
- tensione assente: (\sim 0 Volt): valore 1.

N.B.: Anche la tensione elettrica può essere negativa, indipendentemente dalla logica scelta.

Es. (logica positiva): -5 V, valore 1; \sim 0 V, valore 0.

Circuiti logici

Dispositivi con una serie di

- segnali digitali in ingresso (input)
- segnali digitali in uscita (output).



Combinatori – Sequenziali

Due classi di circuiti logici

- **COMBINATORI**: l'uscita y_n dipende solo dall'ingresso x_n attuale, non c'è memoria:
$$y_n = f(x_n)$$
- **SEQUENZIALI**: l'uscita y_n dipende anche dagli ingressi passati x_{n-1}, \dots (o dallo **stato** s_{n-1}), hanno memoria della storia passata:

$$y_n = f(x_n, x_{n-1}, \dots).$$

equivalente a

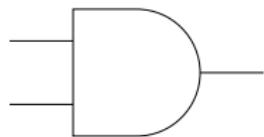
$$\begin{cases} y_n = g(x_n, s_{n-1}) \\ s_{n-1} = h(x_{n-1}, x_{n-2}, \dots) = q(x_{n-1}, s_{n-2}) \end{cases} .$$

Porte logiche

Semplici circuiti combinatori.

Elementi base della costruzione: ogni circuito combinatorio può essere ottenuto collegando in modo opportuno un insieme di porte logiche.

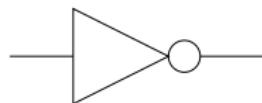
Porte logiche fondamentali



AND



OR



NOT

Comportamento delle porte logiche

- La porta logica **AND** ha uscita 1 se tutti gli ingressi sono a 1 (e zero altrimenti)
- La porta logica **OR** ha uscita 1 se almeno uno degli ingressi è a 1 (e zero altrimenti)
- La porta logica **NOT** ha uscita 1 se l'ingresso ha valore 0 (e zero altrimenti)

Le porte AND, OR possono avere **più di 2** ingressi (con il comportamento descritto sopra).

Come si realizzano le porte logiche?

Paragone: elettronica — idraulica

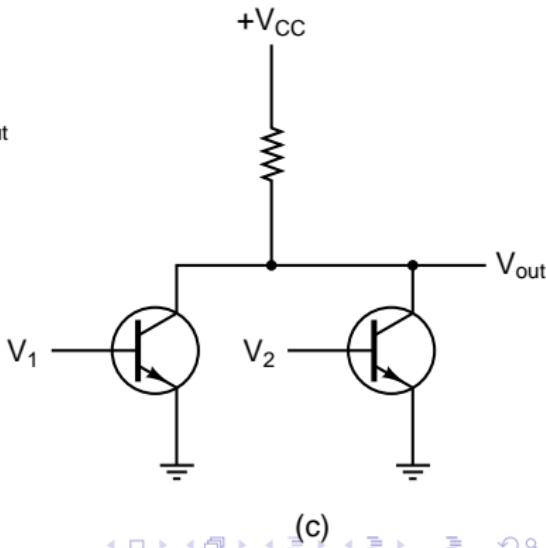
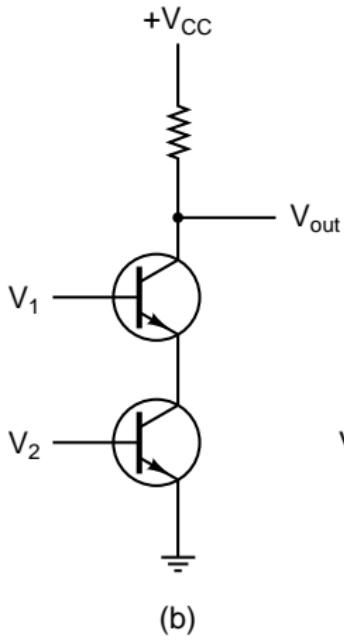
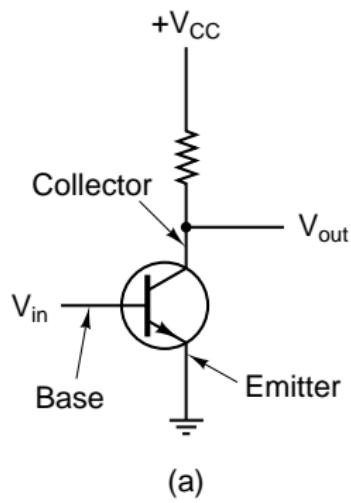
Fornire un'idea intuitiva del significato dei circuiti elettrici: reti idrauliche.

I due domini sono governate da leggi fisiche simili.

- corrente elettrica — portata (flusso d'acqua)
- fili elettrici — tubi di un sistema idraulico
- tensione elettrica — pressione (spinta)
- batteria/alimentatore — pompa idraulica
- resistenza — strozzatura, resistenza al flusso.
- interruttore — saracinesca
- condensatore — serbatoio
- transistor — valvola idraulica.

Realizzazione fisica porte logiche

Transistor (Shockley, Bardeen & Houser, Nobel '56 per la fisica): regola il segnale attraverso un altro segnale (di controllo).



Descrizione comportamento circuiti

Informale:

- La porta logica OR ha uscita 1 se almeno uno degli ingressi è a 1 (e zero altrimenti).
- D ha valore 1 se almeno uno degli ingressi ha valore 1 (D è vero se almeno uno degli ingressi è vero).
- E ha valore 1 se esattamente due ingressi hanno valore 1.
- F ha valore 1 se tutte tre gli ingressi hanno valore 1.

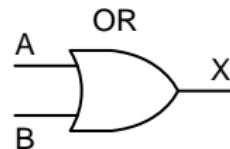
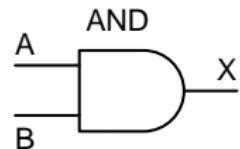
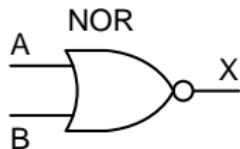
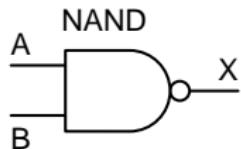
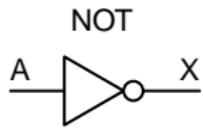
Formale: tavole di verità

- Descrizione esaustiva del comportamento: per ogni combinazione dei valori di ingresso, specifica i valori di uscita
- Una tabella con tante righe quante le combinazioni di ingresso
- Insieme finito di combinazioni di ingresso;
- Se ci sono n ingressi, le possibili combinazioni sono: 2^n .

Tabella di verità

A	B	C	D	E	F
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	1	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	0	1

Tabelle di verità per le porte logiche



A	X
0	1
1	0

(a)

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

(b)

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

(c)

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

(d)

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

(e)

Sintesi di circuiti

Altri metodi:

- metodo duale

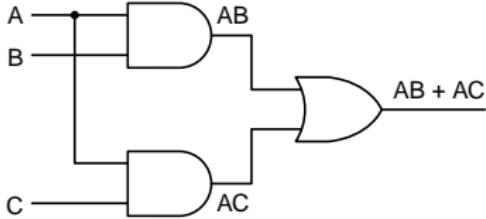
principio di dualità: scambio $0 \leftrightarrow 1$ e quindi è sufficiente calcolare l'uscita duale scambiando AND \leftrightarrow OR

- mappe di Karnaugh

permettono di realizzare reti con un numero **minimo** di porte logiche. Possono essere a loro volta dualizzate

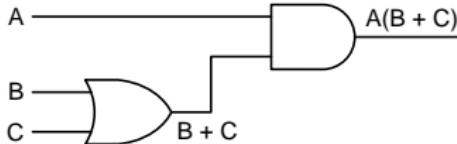
- vedremo un altro modo per minimizzare le reti.

Dal circuito alla tabella di verità



A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

(a)



A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

(b)

Mappe di Karnaugh

Il metodo precedente per la sintesi di circuiti crea circuiti con più porte logiche del necessario.

Le mappe di Karnaugh sono un metodo per minimizzare il numero di porte logiche.

Si basano sulla regola seguente:

- un OR che riceve porte AND con ingresso identico può essere sostituito da un'unica porta AND
- nel caso particolare in cui l'ingresso A sia unico e l'ingresso B entri alternativamente negato e non nelle porte AND:
$$(A \text{ AND } B) \text{ OR } (A \text{ AND } (\text{NOT } B)) = AB + A\bar{B} = A.$$

Mappe di Karnaugh

Tabelle di verità in forma di matrici:

	A 0	1
B 0		
1		

	AB 00	01	11	10
C 0				
1				

AB	00	01	11	10
CD	00			
01				
11				
10				

Notare l'ordine di scrittura delle combinazioni:
00, 01, 11, 10.

Cioè: righe e colonne adiacenti differiscono ciascuna per un bit. In tal modo le mappe sfruttano l'equivalenza $AB + A\bar{B} = A$.

Esempio:

		BC			
		00	01	11	10
A	0	0	0	1	0
	1	0	1	1	1

$$F = AC + BC + AB$$

infatti

$$A\bar{B}C + ABC = AC.$$

La variabile B, che assume entrambi i valori nella coppia di 1 adiacenti, può essere omessa.

Raggruppamenti di valori adiacenti

Bisogna raggruppare le adiacenze di uscite 1 che appaiono in un rettangolo (quadrato) di 2, 4, 8 ... (2^n) elementi.

In ogni rettangolo:

- alcune variabili assumono tutte le possibili combinazioni;
- altre assumono sempre lo stesso valore.

Nell'AND associato appaiono solo le seconde (le prime sono OR di valori complementari).

Es. (mappa 4×4):

$$ABCD + \overline{A}BCD + ABC\overline{D} + ABC\overline{D} = BCD + BC\overline{D} = BC.$$

Mappe di Karnaugh: gruppi di 4 “uni”

		v3 v4				
		00	01	11	10	
v1 v2		00	0	0	0	1
01	00	0	0	1	1	
	01	0	0	1	1	
11	00	0	0	1	1	
	10	0	0	1	1	

$$F = V_3 \overline{V_4} + V_2 V_3 + V_1 V_3$$

Mappe di Karnaugh: lati opposti.

I lati opposti vanno considerati contigui.

		YZ				
		00	01	11	10	
wx		00	1	0	0	1
wx	01	1	1	0	0	
	11	1	0	0	0	
wx	10	1	0	0	1	

$$F = \overline{Y}\overline{Z} + \overline{X}\overline{Z} + \overline{W}XY$$

Applicazione duale

AB	00	01	11	10
CD	1	1	0	1
00	1	0	0	1
01	0	0	1	1
11	1	1	1	1
10	1	1	1	1

$$S_n = \overline{A}CD + ABC\bar{C} + \overline{A}BD$$

AB	00	01	11	10
CD	1	1	0	1
00	1	0	0	1
01	0	0	1	1
11	1	1	1	1
10	1	1	1	1

$$S_p = \overline{A}\overline{C} + BC + AD$$

$$S_p = AC + \overline{B}\overline{C} + \overline{A}\overline{D} \text{ (correggere qui sopra \uparrow)}$$

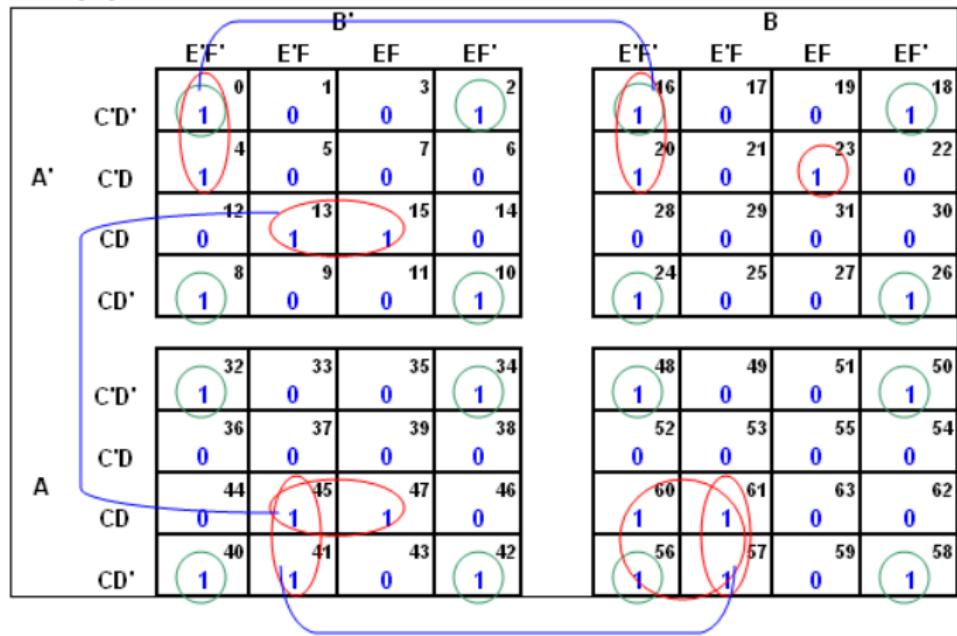
$$AB + A\overline{B} = A. \text{ Dualizziamo: } (\overline{A} + \overline{B})(\overline{A} + B) = \overline{A}.$$

$$S_n = (\overline{A} + \overline{B} + C)(A + \overline{B} + \overline{D})(A + \overline{C} + \overline{D}) = S_p.$$

Attenzione: quindi **non** ottengo il circuito duale.

Mappe di Karnaugh con 5 o 6 variabili

Si usa la terza dimensione. Disegnare 2 o 4 mappe di Karnaugh a 4 variabili, cassette corrispondenti in mappe adiacenti sono considerate adiacenti.



Specifiche incomplete, valori indefiniti

A volte alcune combinazioni possono restare indeterminate.

L'implementazione sceglie i valori che portano al circuito con meno porte.

AB \ CD	00	01	11	10
00			X	
01	X	1	X	1
11	1	1	X	X
10		1	X	X

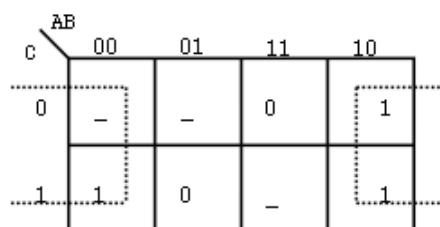
Determinazione della funzione minima e realizzazione dello schema logico corrispondente alla tabella della verità di [Figura 14](#).

Figura 14.

A	B	C	Y
0	0	0	-
0	0	1	1
0	1	0	-
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	-

La mappa di Karnaugh relativa alla tabella data è la seguente:

Figura 15.



Assumendo la condizione di indifferenza localizzata nel raggruppamento come 1 e le altre come 0, la funzione minima vale:

$$Y = \overline{B}$$

Riferimento

I lucidi presentati oggi riassumono la dispensa qui di seguito allegata (contiene diversi errori!).

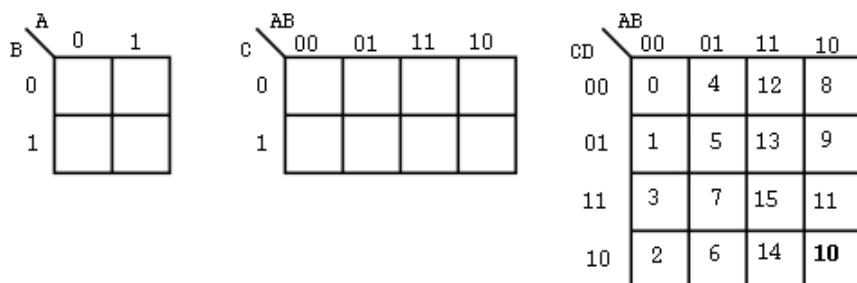
[Next](#) [Previous](#) [Contents](#)

3.1.4. Semplificazione delle funzioni logiche mediante le mappe di Karnaugh

3.1.4.1. Le mappe di Karnaugh

Le semplificazioni di una funzione logica possono essere effettuate mediante i teoremi dell'algebra di Boole. Esiste però un metodo molto più pratico di semplificazione che quello costituito dalle mappe di Karnaugh. Tale metodo di facile applicazione per funzioni di poche variabili, in genere fino ad un massimo di quattro o cinque, risulta alquanto difficoltoso se le variabili diventano numerose. In [Figura 1](#) sono riportate le mappe di Karnaugh (di forma quadra o rettangolare) per funzioni di due, tre o quattro variabili.

Figura 1. Mappe di Karnaugh



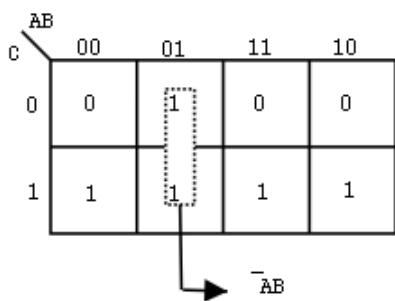
Mappe di Karnaugh

Ogni mappa contiene tante caselle quante sono 2^n combinazioni delle n variabili della funzione logica. Caselle che hanno un lato in comune sono dette adiacenti. Debbono essere considerate adiacenti anche le caselle all'estremità di una riga o di una colonna, come se la mappa fosse disegnata su una superficie chiusa su se stessa. Sono caselle adiacenti, ad esempio, le caselle 0 e 8, 10 e 8, 5 e 7; non lo sono invece le caselle 4 e 13, 1 e 13 etc. Le caselle inoltre sono disposte in modo tale che passando da una qualsiasi ad una adiacente sulla stessa riga o sulla stessa colonna cambia di valore una sola variabile. Per rappresentare una funzione Y sulla mappa basta scrivere 1 nelle caselle corrispondenti alle combinazioni per le quali la funzione vale 1. Ad esempio alla funzione:

$$Y = \overline{ABC} + \overline{ABC} + \overline{ABC} + ABC$$

(forma canonica della somma) corrisponde la seguente mappa di Karnaugh:

Figura 2.



Si considerino ora le due caselle comprese nel rettangolo tratteggiato; esse corrispondono alle combinazioni 010 e 011 delle variabili A, B, C e quindi nell'espressione algebrica della funzione alla somma del secondo e terzo termine che vale:

$$\overline{A} \overline{B} \overline{C} + \overline{A} B C = \overline{A} B (C + \overline{C}) = \overline{A} B$$

Il prodotto $\neg A B$ così ottenuto è evidenziato nella [Figura 2](#) dal rettangolino che racchiude i due 1 adiacenti. I due fattori che lo compongono sono dati da quelle variabili (A,B) che non cambiano di valore (0,1) nelle due caselle del rettangolino. [Questo prodotto può essere scritto direttamente dall'osservazione della mappa, assumendo come fattori le variabili che mantengono il loro valore, negando quelle a valore 0 e lasciando inalterate quelle a valore uno.]

Le considerazioni precedenti possono essere estese, riferendosi ancora alla figura2, al raggruppamento delle quattro caselle contigue dell'ultima riga ottenendo come risultato dei quattro 1 adiacenti il solo termine C. Infatti lungo tutta la riga la sola variabile che resta costante è la C (che non va poi negata perchè vale 1).

Poichè tutti gli uno della mappa sono stati inclusi nei rettangoli tratteggiati, la somma dei termini corrispondenti a detti rettangolino dà come risultato l'espressione minima della funzione:

$$Y = \overline{A} B + \overline{C}$$

Tale risultato può essere raggiunto, come può essere facilmente verificato, applicando i teoremi dell'algebra di Boole.

In generale, per funzioni logiche di n variabili si può dire che:

Due 1 adiacenti rappresentano un prodotto di n-1 variabili.

Quattro 1 adiacenti rappresentano un prodotto di n-2 variabili.

Otto 1 adiacenti rappresentano un prodotto di n-3 variabili.

Sedici 1 adiacenti rappresentano un prodotto di n-4 variabili.

Etc...

In definitiva per minimizzare una funzione logica mediante il metodo delle mappe di Karnaugh si opera nel modo seguente:

1. Si rappresenta la funzione logica sulla mappa;
2. Si localizzano sulla mappa i più grandi raggruppamenti possibili di 1 adiacenti che formano

potenze del 2;

3. Si sceglie il numero minimo di raggruppamenti che copre tutti gli 1 della mappa tenendo conto che eventuali termini isolati debbono essere riportati integralmente.

Esempio 1

Realizzare lo schema logico che soddisfa la seguente tabella di verità:

Figura 3.

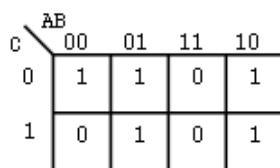
A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

La forma canonica della somma vale:

$$Y = \overline{ABC} + \overline{ABC} + \overline{ABC} + \overline{BC} + A\overline{BC}$$

e la rappresentazione della funzione sulla mappa di Karnaugh è la seguente:

Figura 4.



Dall'esame della [Figura 4](#) si può notare che sono possibili due diversi raggruppamenti di 1 adiacenti ([Figura 5](#) a,b) a cui corrispondono due diverse espressioni

$$Y_a = \overline{AB} + \overline{AC} + A\overline{B}$$

$$Y_b = A\overline{B} + A\overline{B} + \overline{BC}$$

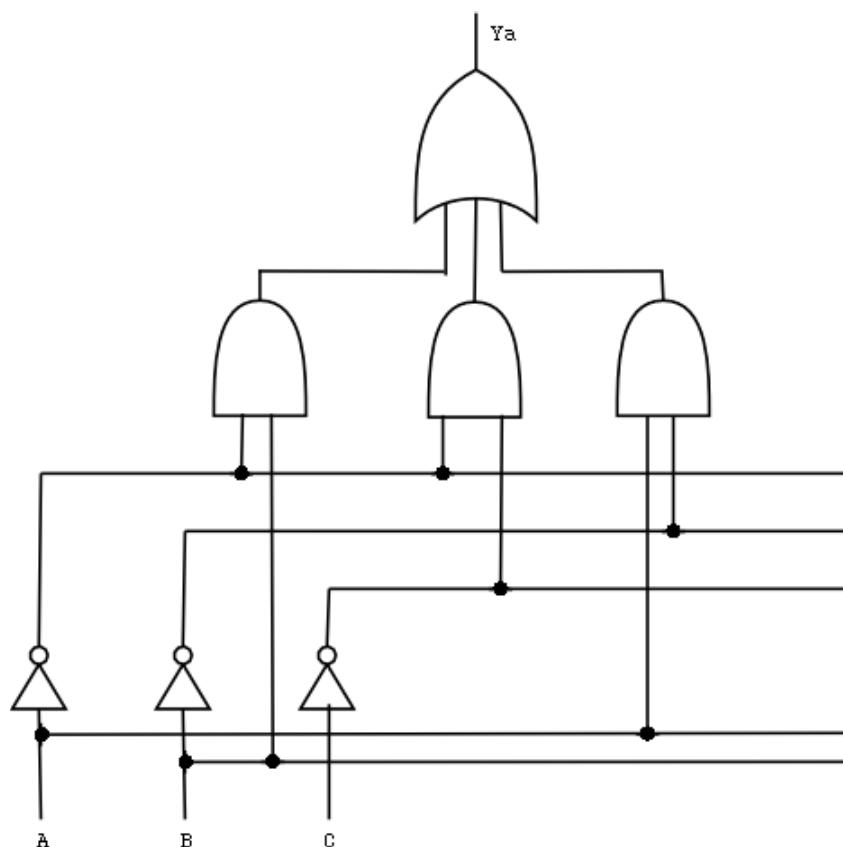
Figura 5.

		AB	00	01	11	10
		C	0	1	0	1
AB	C	00	1	1	0	1
		1	0	1	0	1

		AB	00	01	11	10
		C	0	1	1	0
AB	C	00	0	0	1	1
		1	0	1	0	1

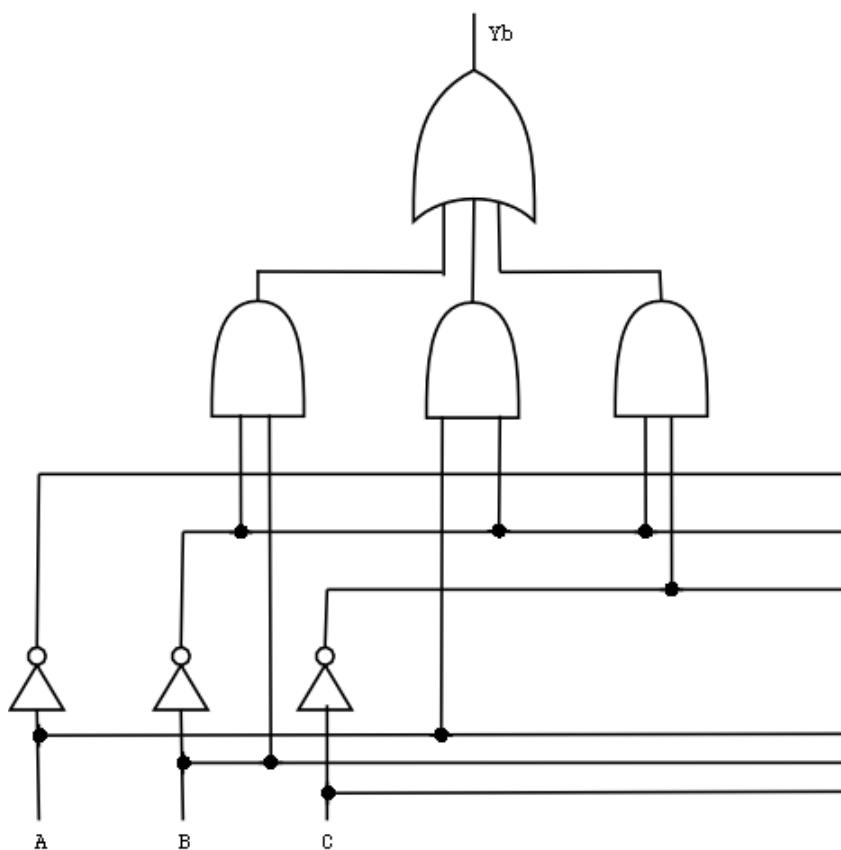
Alle due espressioni di Y_a e Y_b , entrambe minime, corrispondono gli schemi logici di [Figura 6](#) e [Figura 7](#), rispettivamente:

Figura 6.



Può essere facilmente verificato che i circuiti di [Figura 6](#) e [Figura 7](#) soddisfano alla medesima tabella della verità e quindi realizzano la stessa funzione logica pur partendo da espressioni diverse.

Figura 7.



In definitiva si può dire che da una mappa di Karnaugh è possibile ricavare funzioni minime diverse a seconda del raggruppamento scelto. Un rigoroso raggruppamento di 1 adiacenti comporta una espressione ulteriormente semplificabile e quindi non minima.

Ad esempio, facendo riferimento alla mappa di [Figura 4](#) al raggruppamento di 1 adiacenti di [Figura 8](#) corrisponde la funzione:

Figura 8.

AB		00	01	11	10
C		0	1	0	1
A	B	0	0	1	1
1	0	1	0	0	1

$$Y = \overline{AC} + A\overline{B} + \overline{ABC}$$

Che non è una funzione minima, infatti tramite i teoremi dell'algebra di Boole, si osserva che:

$$Y = \overline{A}(\overline{C} + BC) + A\overline{B} = \overline{A}(\overline{C} + B) + A\overline{B} =$$

$$\overline{AC} + \overline{AB} + A\overline{B} = Y_a$$

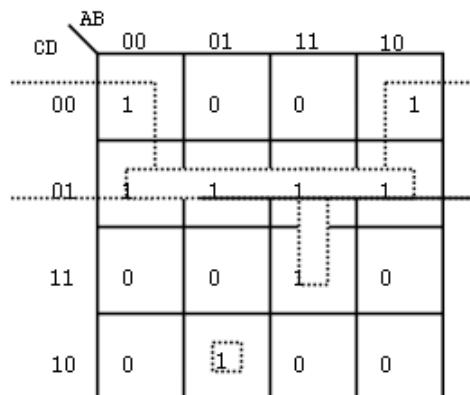
Esempio 2

Determinazione delle funzioni minime della mappa di Karnaugh di [Figura 9](#) e realizzazione dello schema logico corrispondente:

Figura 9.

		AB	00	01	11	10
		CD	00	01	11	10
00	00		1	0	0	1
			1	1	1	1
11	01		0	0	1	0
			0	1	0	0

Figura 10.



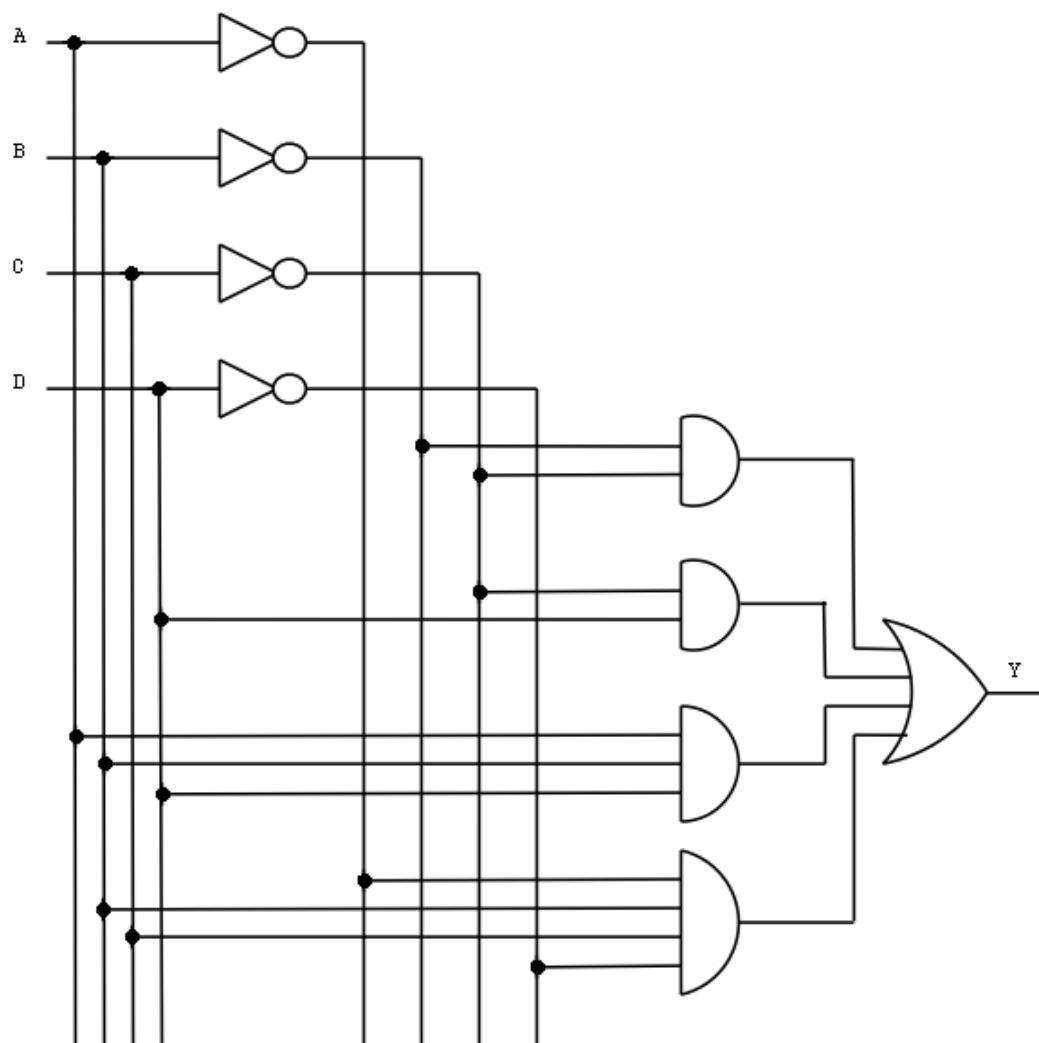
I raggruppamenti sono indicati in [Figura 10](#).

La funzione minima vale:

$$Y = \overline{BC} + \overline{CD} + ABD + \overline{ABC}\overline{D}$$

Lo schema che la realizza è quello di [Figura 11](#).

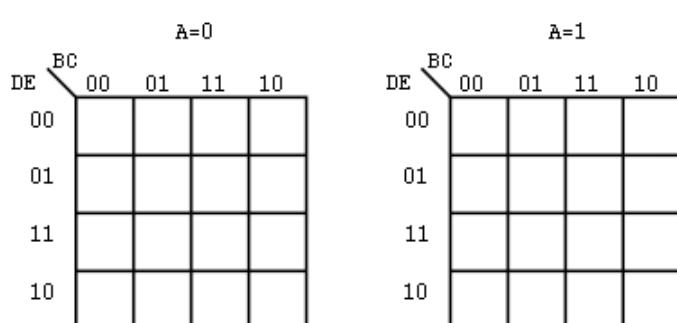
Figura 11.



3.1.4.2 Mappe di Karnaugh per più di quattro variabili

Le mappe di Karnaugh per più di quattro variabili binarie devono essere costruite sempre rispettando la regola che nel passaggio da una casella a quella adiacente sulla stessa riga o sulla stessa colonna deve cambiare una sola variabile. Per quanto riguarda la semplificazione di una funzione a cinque variabili essa può, essere fatta mediante due mappe di Karnaugh da 16. Le adiacenze possono essere ben localizzate pensando di sovrapporre le due mappe e considerando adiacenti le caselle che si corrispondono verticalmente.

Figura 12.



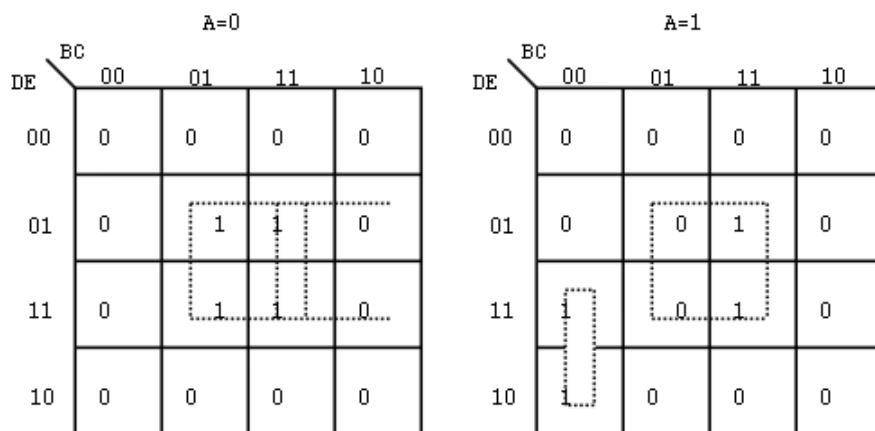
Esempio 3

Minimizzare la seguente funzione:

$$Y = \overline{ABC}\overline{D}E + \overline{ABC}\overline{D}E + \\ \overline{ABC}DE + \overline{ABC}DE + ABC\overline{D}E + \\ ABCDE + A\overline{B}\overline{C}DE + A\overline{B}\overline{C}\overline{D}E$$

La mappa di Karnaugh con la rappresentazione dei mintermini è rappresentata in [Figura 13.](#)

Figura 13.



Ne segue che:

$$Y = \overline{ACE} + BCE + A\overline{BCD}$$

Naturalmente all'aumentare del numero delle variabili della funzione da minimizzare aumenta il numero di caselle della mappa di Karnaugh corrispondente e di conseguenza anche la difficoltà dell'operatore nella ricerca di più ampi raggruppamenti possibili in ragione di 2^n . In realtà quando il numero delle variabili binarie risulta maggiore di cinque è preferibile passare ad altri sistemi di minimizzazione come per esempio quello di Quine Mc-Cluskey.

3.1.4.3. Condizioni di indifferenza

Accade, a volte, che il valore dell'uscita di un'assegnata tabella di verità non venga specificato per alcune combinazioni delle variabili d'ingresso, o perchè queste combinazioni non possono verificarsi oppure perchè più in generale, non interessa conoscere i valori dell'uscita corrispondenti a tali combinazioni. Si parla così di condizioni di indifferenza. In questa situazione l'uscita, che può assumere indifferentemente il valore 0 o 1, viene riportata sulla mappa di Karnaugh con il simbolo "-", simbolo quest'ultimo derivato dalla sovrapposizione di 0 e 1. Le condizioni di indifferenza possono essere sfruttate al fine di semplificare la funzione logica assegnando il valore 1 quando ciò è conveniente.

Esempio 4

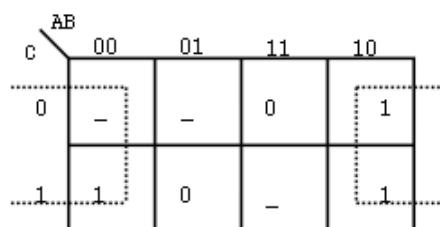
Determinazione della funzione minima e realizzazione dello schema logico corrispondente alla tabella della verità di [Figura 14](#).

Figura 14.

A	B	C	Y
0	0	0	-
0	0	1	1
0	1	0	-
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	-

La mappa di Karnaugh relativa alla tabella data è la seguente:

Figura 15.



Assumendo la condizione di indifferenza localizzata nel raggruppamento come 1 e le altre come 0, la funzione minima vale:

$$Y = \overline{B}$$

Algebra booleana

Describe un circuito con un'espressione algebrica, utile per:

- rappresentare sinteticamente un circuito,
- verificare l'equivalenza tra due circuiti,
- semplificare un circuito: trasformarlo in un circuito equivalente più semplice.

[George Boole (1815-1864), Claude Shannon (1916-2001)]

Algebra booleana

Algebra dei valori di verità:

- ideata nell'ambito della logica
- in logica: è necessario formalizzazione linguaggio;
- la valutazione della verità di una espressione viene ridotta ad un calcolo;
- si considerano solo due possibili valori di verità: **vero** (1), **falso** (0).

Formalizzazione del linguaggio

Frasi composte da **proposizioni** base (**atomiche**) come:

splende il sole — piove — nevica;

combinate con l'uso di **connettivi** (e, o, non):

- piove **e** splende il sole
- o piove **o** splende il sole
- **non** piove
- se piove allora – **non** splende il sole **e non** nevica.

Si calcola la verità di una frase a partire dalla verità delle proposizioni atomiche.

Algebra booleana

Operazioni (connettivi base)

AND	.	\wedge	prodotto logico	(binaria)
OR	\pm	\vee	somma logica	(binaria)
NOT		\neg	negazione	(unaria)

Valori (valori di verità)

TRUE	1	t
FALSE	0	f

Connettivi e valori hanno diverse scritture a seconda dell'ambito: logica, progettazione di circuiti.

Comportamento delle operazioni

Coincide con quello delle porte logiche:

- $A \text{ AND } B$ è vera se A è vera **e** B è vera
(es: piove **e** splende il sole)
- $(A \text{ OR } B)$ è vera se A è vera **o** B è vera
(es: piove **o** splende il sole)
- NOT A è vera se A **non** è vera (cioè se A è falsa)
(es: **non** piove)

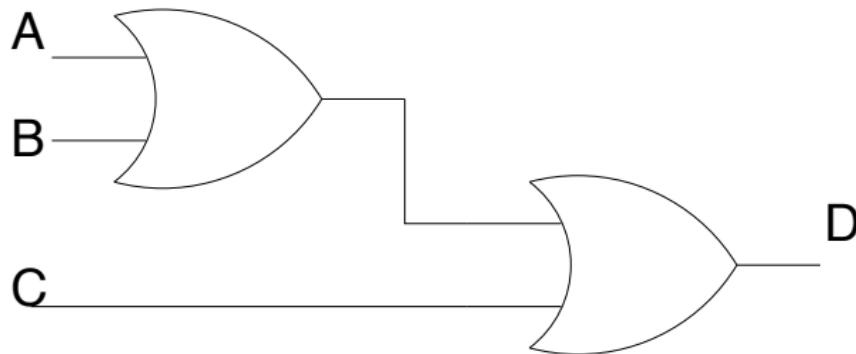
Applicazioni all'informatica

Rappresentare un circuito mediante un'espressione algebrica [Shannon]

- circuiti logici \Rightarrow funzioni, espressioni
- porte logiche \Rightarrow operazioni base
- input \Rightarrow argomenti (variabili)
- output \Rightarrow valore dell'espressione
- segnali intermedi \Rightarrow sotto-espressioni

Espressioni e circuiti

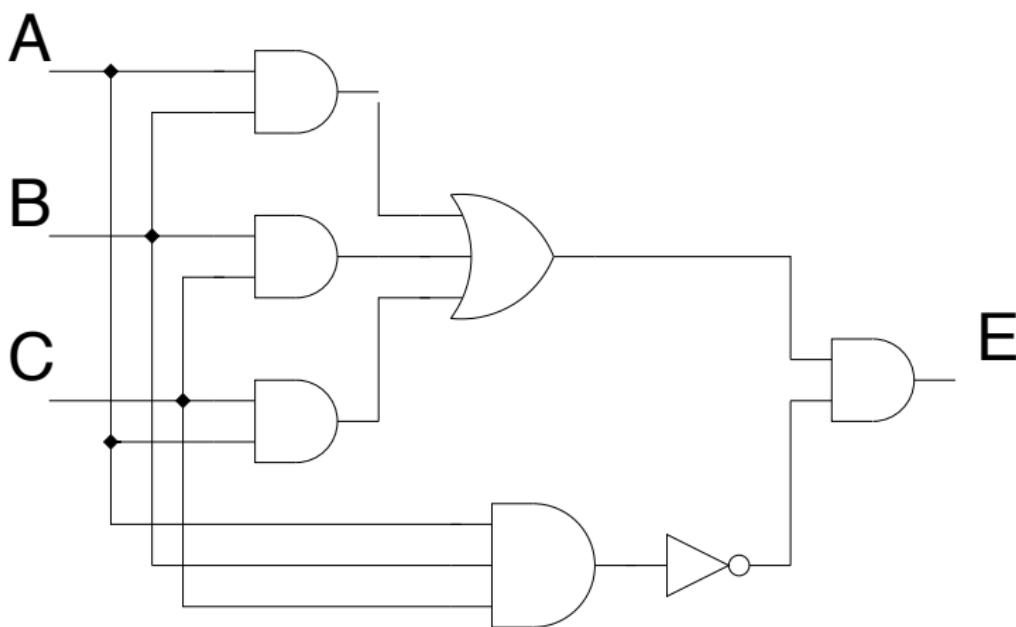
Ai circuiti formati da porte logiche posso associare espressioni (e viceversa)



$$D = (A + B) + C$$

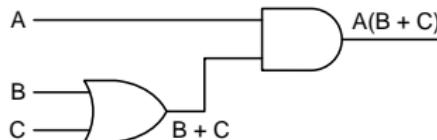
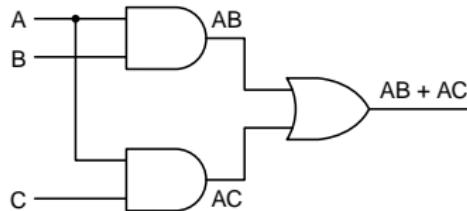
Espressioni e circuiti

$$E = ((A \cdot B) + (B \cdot C) + (A \cdot C)) \cdot (\overline{A} \cdot \overline{B} \cdot \overline{C})$$



Espressioni \subset Tabelle verità

A più circuiti/espressioni può corrispondere un'**unica** tabella di verità



A	B	C	AB	AC	AB + AC
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	0	0	0
1	0	1	0	1	1
1	1	0	1	0	1
1	1	1	1	1	1

(a)

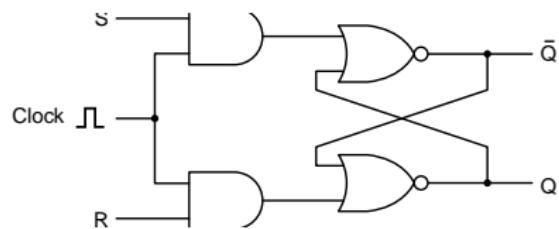
A	B	C	A	B + C	A(B + C)
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

(b)

Circuiti < Espressioni < Tabelle verità

Alcuni circuiti non hanno una corrispondente espressione algebrica:

- circuiti in cui un'uscita comanda più ingressi;
- circuiti con retroazione:



C'è maggiore libertà nel collegare fili che nello scrivere espressioni.

Convenzioni nella scrittura

Le usuali convenzioni dell'algebra:

- il segno di prodotto può essere **omesso**, oppure indicato con \times o $*$ (AB , $A \times B$, $A * B$)
- raramente la negazione è rappresentata con '
 $(A + B)' = \overline{A + B}$
- il prodotto ha **precedenza sulla somma**
- AND e OR sono operazioni **associative**:
 $(AB)C = ABC = A(BC)$
 $(A + B) + C = A + B + C = A + (B + C).$

Equivalenze algebriche

Ogni algebra è caratterizzata da un insieme di equivalenze.

Equivalenze booleane utili per:

- stabilire equivalenze tra circuiti;
- semplificare circuiti.

Equivalenze booleane

Elemento nullo	$0 + A = A$
Identità	$1 + A = 1$
Idempotenza	$A + A = A$
Inverso	$A + \bar{A} = 1$
Commutatività	$A + B = B + A$
Associatività	$(A + B) + C = A + (B + C)$
Distributività	$A(B + C) = AB + AC$
Assorbimento	$A + (A \cdot B) = A$
De Morgan	$\overline{A + B} = \overline{A} \cdot \overline{B}$
Negazione	$\overline{\overline{A}} = A$

Equivalenze duali

Identità	$1 \cdot A = A$
Elemento nullo	$0 \cdot A = 0$
Idempotenza	$A \cdot A = A$
Inverso	$A \cdot \bar{A} = 0$
Commutatività	$A \cdot B = B \cdot A$
Associatività	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$
Distributività	$A + BC = (A + B)(A + C)$
Assorbimento	$A \cdot (A + B) = A$
De Morgan	$\overline{A \cdot B} = \overline{A} + \overline{B}$
Negazione	$\overline{\overline{A}} = A$

Considerazioni

- Identità, commutatività, associatività, distributività (somma sul prodotto), elemento nullo, sono valide anche per l'algebra degli interi (le altre no).
- Tutte le equivalenze dell'algebra degli interi restano valide, tenuto conto che la negazione non ha **nessuna attinenza** con le operazioni di opposto o inverso.
- Equivalenze dimostrabili mediante tabelle di verità, oppure mediante argomenti di logica.

Semplificazione di espressioni

- Identità ed elemento nullo permettono di eliminare le costanti 0 e 1 dalle espressioni.
- Le regole di De Morgan e la negazione permettono di esprimere variabili negate.
- Commutatività, associatività, inverso, idempotenza e assorbimento permettono di riordinare i termini ed eliminare doppioni.
- La proprietà distributiva permette di riscrivere ogni espressione come somma di prodotti (**forma normale**). Controparte algebrica: polinomio.
- Assorbimento: si può derivare dalle proprietà di identità, distributività, elemento nullo.

Esempi di applicazione

Attraverso le regole indicate si possono semplificare le seguenti espressioni.

- De Morgan: $\overline{A \cdot B + \overline{C}}$, $\overline{A \cdot B \cdot C}$, $\overline{A + B + C}$.
- Distributiva: $((A + B) \cdot C + A) \cdot \overline{B}$
- Con alcune regole: $A \cdot (1 + B) + 1 \cdot A + B$

$$\frac{\overline{A \cdot \overline{B} \cdot \overline{C}} \cdot \overline{A \cdot B \cdot C}}{(A + B) \cdot (\overline{A \cdot \overline{B} \cdot \overline{C}}) \cdot \overline{(A \cdot B \cdot C)}}$$

Semplificare un'espressione non significa necessariamente minimizzare il corrispondente circuito.

Dualità e complementarietà

Ogni proprietà booleana resta valida se scambiamo

- la costante 0 con la costante 1
- l'operazione $+$ con l'operazione \cdot .

Formalmente: data un'espressione booleana E ,
l'espressione **duale** \tilde{E} si ottiene scambiando tra loro
le negazioni, le operazioni $+$ e \cdot e le costanti 0 e 1.

Attenzione: l'espressione \tilde{E} ottenuta scambiando tra loro **solo** le operazioni $+$ e \cdot e le costanti 0 e 1
complementa (**non dualizza!**) il risultato \overline{E} .

- $E = A + \overline{A} \cdot (B + 0)$
- $\overline{E} = \overline{A} \cdot (A + \overline{B} \cdot 1)$
- $\tilde{E} = A \cdot (\overline{A} + B \cdot 1)$.

Altre porte logiche

NAND NOR XOR NXOR

A	B	$A \text{ NAND } B$	$A \text{ NOR } B$	$A \text{ XOR } B$	$A \text{ NXOR } B$
0	0	1	1	0	1
0	1	1	0	1	0
1	0	1	0	1	0
1	1	0	0	0	1

Definizione mediante espressioni

$$A \text{ NAND } B = \overline{A \cdot B}$$

$$A \text{ NOR } B = \overline{A + B}$$

$$A \text{ XOR } B = A \cdot \overline{B} + \overline{A} \cdot B = (A + B) \cdot (\overline{A} + \overline{B})$$

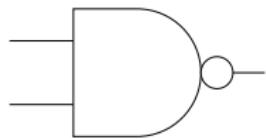
$$A \text{ NXOR } B = A \cdot B + \overline{A} \cdot \overline{B}$$

Le porte NAND e NOR possono essere generalizzate ad n ingressi.

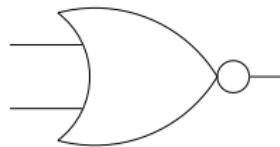
Una NAND con n ingressi restituisce il valore negato di una porta AND con n ingressi.

Esercizio: ha senso definire una porta XOR con n ingressi? Se “sì”, descriverne il comportamento.

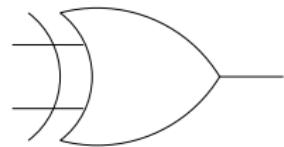
Rappresentazione grafica



NAND



NOR

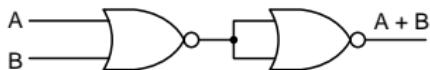
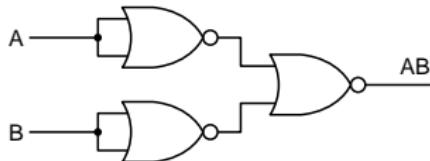
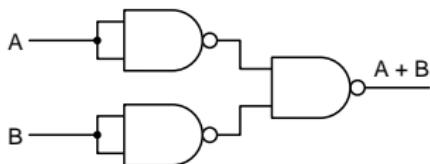
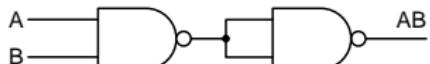


XOR

AND, OR e NOT da NAND e NOR



(a)



(b)

(c)

Completezza di NAND e NOR

Un'espressione in forma normale può essere sempre realizzata con porte NAND:

$$A_1 \cdot A_2 \cdot A_3 + B_1 \cdot B_2 + C_1 \cdot C_2 \cdot C_3 \cdot C_4 =$$

$$\overline{\overline{A_1 \cdot A_2 \cdot A_3 + B_1 \cdot B_2 + C_1 \cdot C_2 \cdot C_3 \cdot C_4}} =$$

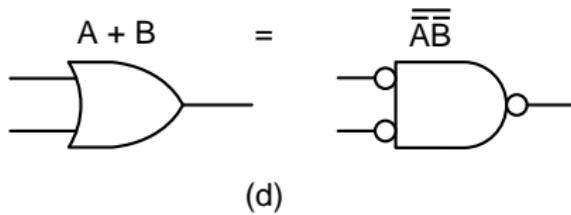
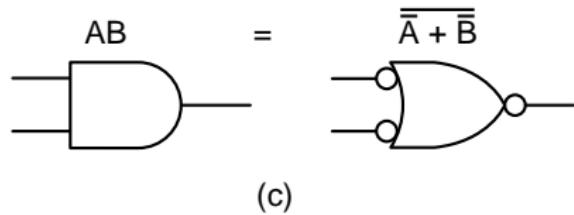
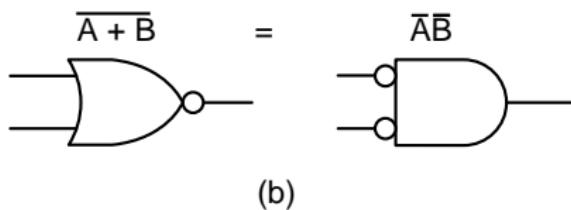
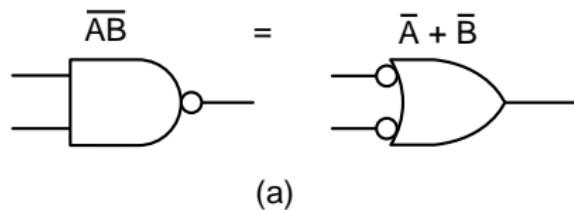
$$\overline{\overline{A_1 \cdot A_2 \cdot A_3}} \cdot \overline{\overline{B_1 \cdot B_2}} \cdot \overline{\overline{C_1 \cdot C_2 \cdot C_3 \cdot C_4}}$$

Nei circuiti integrati vengono impiegate esclusivamente porte NAND o NOR: più economiche da implementare.

Esercizio: come rappresentare, in modo efficiente, ogni circuito con porte NOR?

“Bolla” di inversione

Può apparire anche all'ingresso delle porte logiche



Esercizi

Semplificare le espressioni

- $A + \overline{A}B$
- $\overline{A} \overline{B} \overline{C} + A\overline{B} \overline{C} + \overline{A} \overline{B} C + ABC\overline{C}$
- $\overline{A} + \overline{\overline{B} + \overline{C}} + ABC$
- $\overline{ABC} + A\overline{BC} + \overline{AB}C + AB\overline{C}$

Algebraicamente e (solo per gli INF) attraverso le mappe di Karnaugh.

Circuiti logici di base

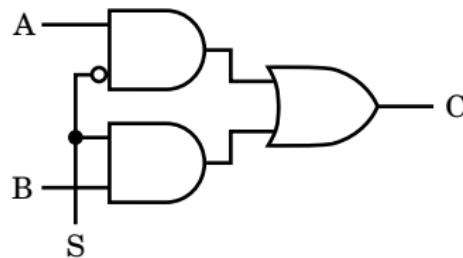
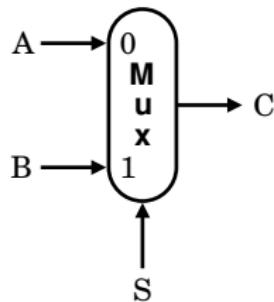
- Primo passo nella costruzione di circuiti combinatori e sequenziali.
- Funzioni di utilità universale.
- Progettazione (e descrizione) strutturata dei circuiti.
- Breve rassegna dei più significativi:
comportamento — implementazione — uso.

Multiplexer

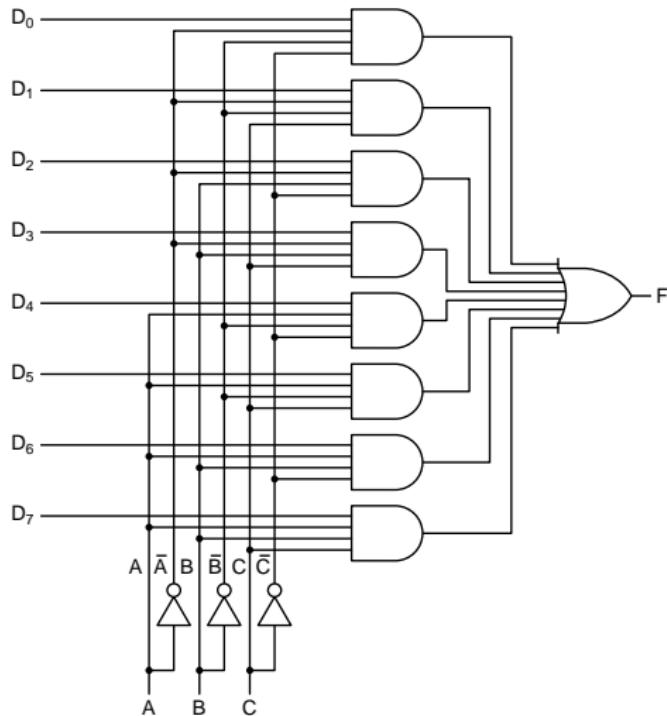
Due tipi di ingressi:

- n ingressi di controllo — 2^n ingressi segnale
- un unica uscita;

il controllo seleziona quale segnale d'ingresso mandare in uscita

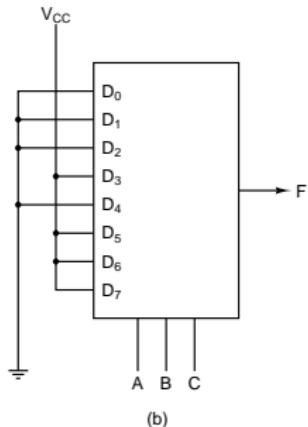
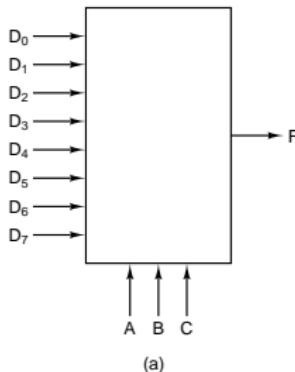


Implementazione



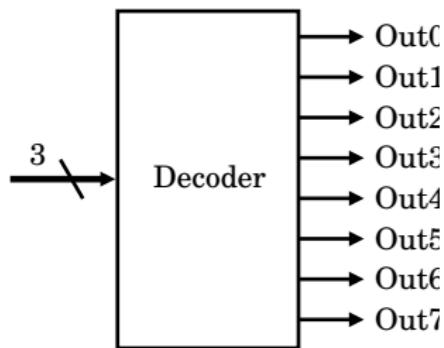
Uso

- trasformazione parallelo \Rightarrow seriale
- realizzare un generatore di tabelle di verità



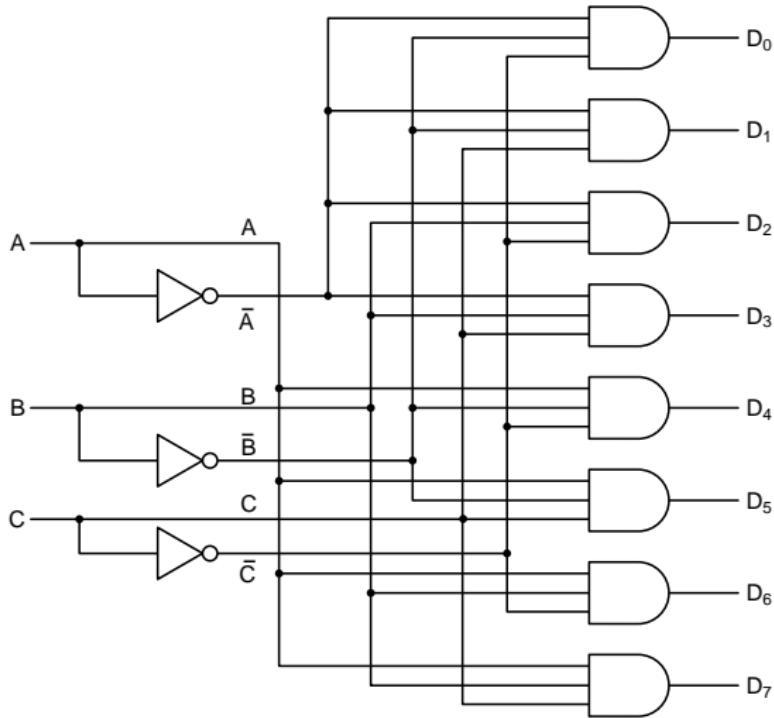
Decoder: decodificatore

- n ingressi — 2^n uscite
- l'ingresso seleziona *una* delle uscite
- l'uscita selezionata ha valore 1 tutte le altre 0.



a. A 3-bit decoder

Implementazione



Uso

Selezionare uno tra molti dispositivi, in cui ogni dispositivo contiene un segnale di attivazione.

Realizzare un **Demultiplexer**: un ingresso, n linee di controllo, 2^n uscite.

Esempio: selezionare un chip di memoria, tra gli 2^n presenti nel calcolatore.

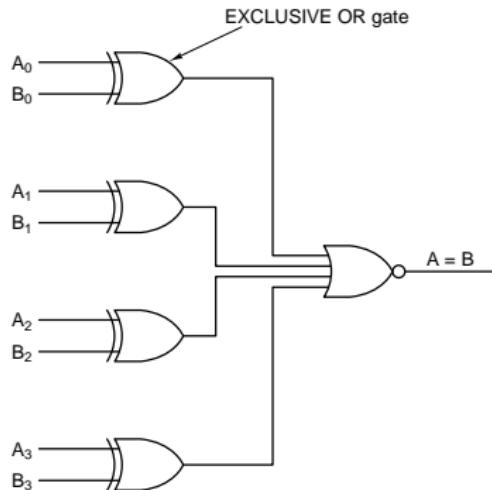
Nessuna parentela con il decoder televisivo.

Comparatore

2 ingressi di n bit: $A = (A_1, \dots, A_n)$, $B = (B_1, \dots, B_n)$

1 uscita

Controlla se gli ingressi sono uguali bit a bit



Uso: confronto di valori

Circuiti aritmetici

Presenteremo i seguenti circuiti:

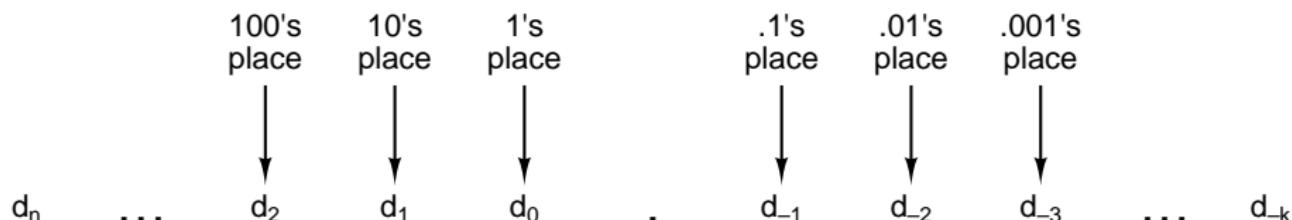
- mezzo sommatore
- sommatore completo
- shifter
- ALU

premessa, come viene realizzata l'aritmetica nel calcolatore.

L'aritmetica dei calcolatori

- come vengono rappresentati i numeri naturali.
- come vengono eseguite le operazioni aritmetiche.

Notazione posizionale: il peso di una cifra dipende dalla sua posizione:



$$\text{Number} = \sum_{i=-k}^n d_i \times 10^i$$

Notazione posizionale: basi diverse

Binary	1	1	1	1	1	0	1	0	0	0	0	1
	$1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$											
	1024	+ 512	+ 256	+ 128	+ 64	+ 0	+ 16	+ 0	+ 0	+ 0	+ 0	+ 1
Octal	3	7	2	1								
	$3 \times 8^3 + 7 \times 8^2 + 2 \times 8^1 + 1 \times 8^0$											
	1536	+ 448	+ 16	+ 1								
Decimal	2	0	0	1								
	$2 \times 10^3 + 0 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$											
	2000	+ 0	+ 0	+ 1								
Hexadecimal	7	D	1	.								
	$7 \times 16^2 + 13 \times 16^1 + 1 \times 16^0$											
	1792	+ 208	+ 1									

Notazione binaria

Il calcolatore utilizza base 2, motivi:

- un segnale rappresenta una cifra;
- semplificazione dell'hardware.

Operazioni aritmetiche

Gli algoritmi che realizzano l'operazione **non** dipendono dalla base scelta.

Algoritmo per la somma:

- si sommano le cifre di pari peso,
- a partire dalle meno significative,
- eventualmente si generano riporti.

I numeri in hardware

- Nel calcolatore i numeri rappresentati con un numero fisso di cifre binarie (bit).
- Nel caso dei naturali: 8 o 16 o 32 oppure 64 cifre.
- Non tutti i numeri naturali sono rappresentabili.

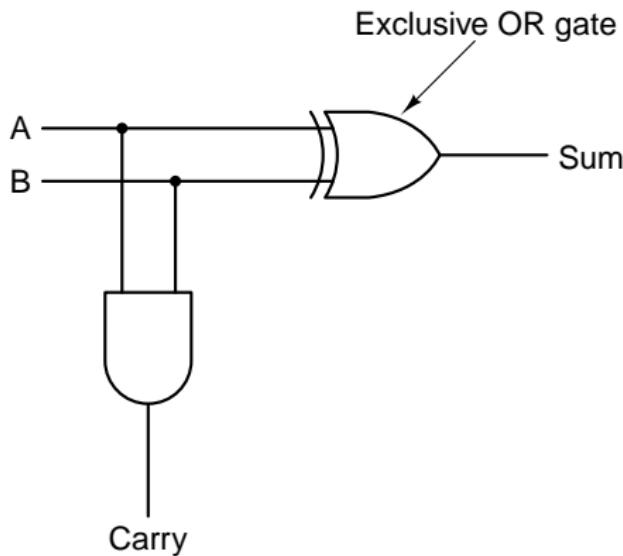
La somma:

- L'algoritmo di somma ripete la stessa operazione su cifre diverse.
- In hardware: tanti circuiti, ciascuno somma una diversa coppia di cifre.

Circuiti aritmetici

Mezzo sommatore

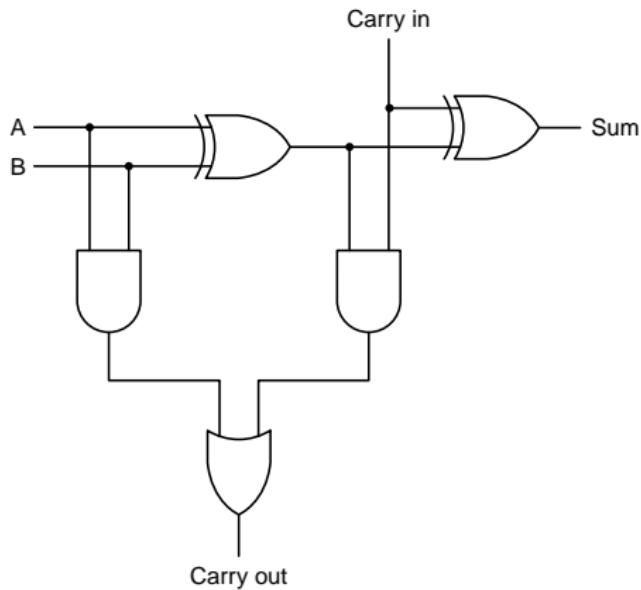
A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Sommatore completo

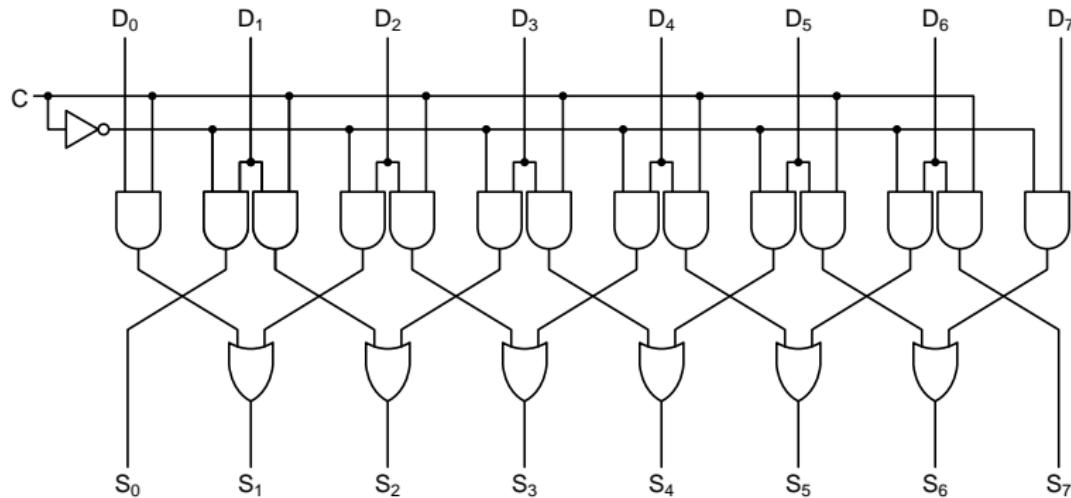
A	B	Carry in	Sum	Carry out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(a)



(b)

Shifter (traslatore)



Traslazione dei bit a dx ($C = 1$) oppure a sx ($C = 0$).
Uso: moltiplicatore per 2 oppure divisore per 2.

$$\left(\sum_{i=-k}^n d_i \cdot 2^i \right) \cdot 2 = \sum_{i=-k}^n d_i \cdot 2^{i+1} = \sum_{i=-k+1}^{n+1} d_{i-1} \cdot 2^i.$$

Laboratorio di architettura degli elaboratori

CIRCUITI COMBINATORI

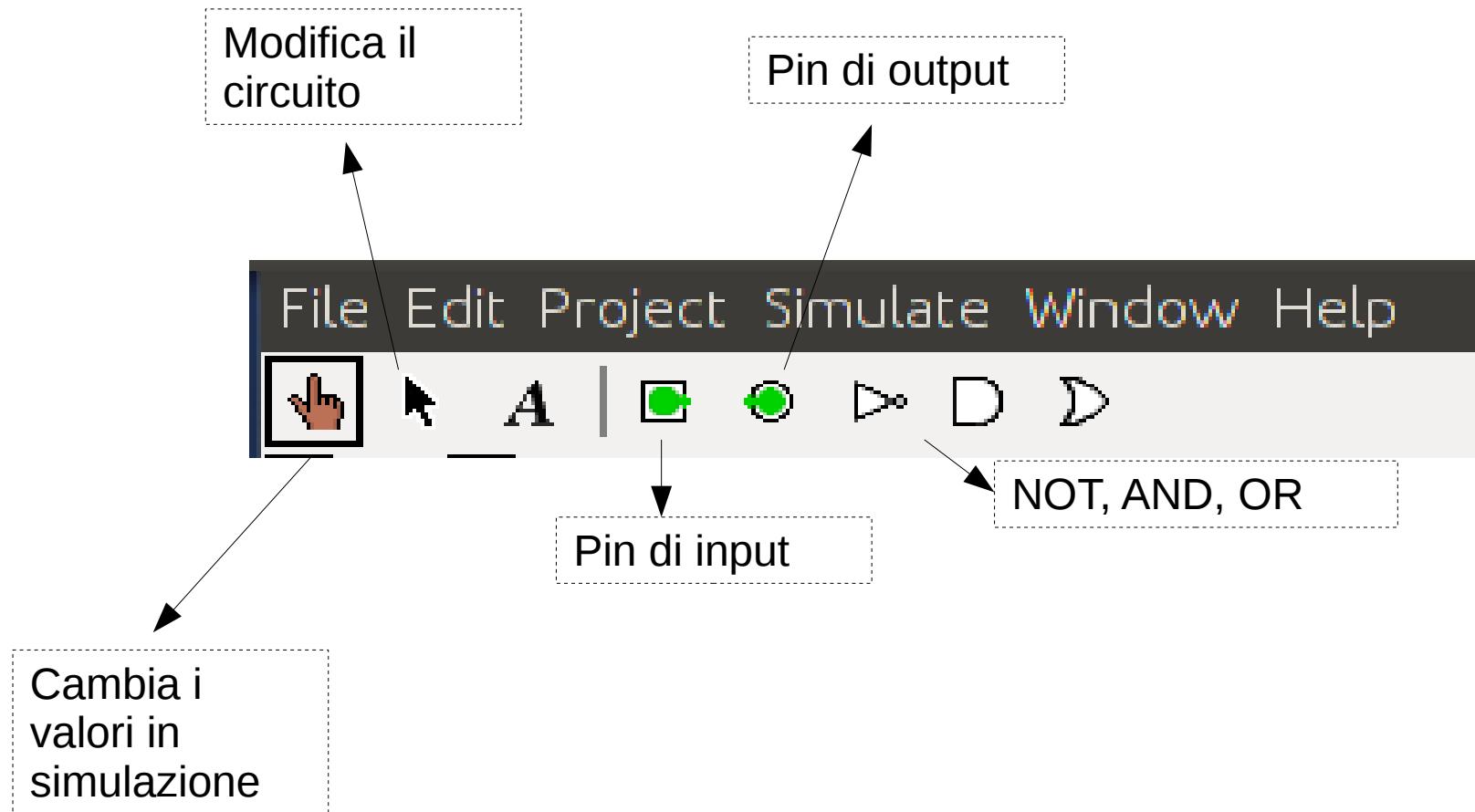
STRUMENTI SOFTWARE

- Logisim (<https://sourceforge.net/projects/circuit>)

CONTATTI

- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

LOGISIM



Esercizio 1

Si progetti un circuito combinatorio, con tre segnali di input, che calcola la minoranza. Il circuito fornisce in uscita 1 se almeno due ingressi sono 0, altrimenti genera 0 come uscita.

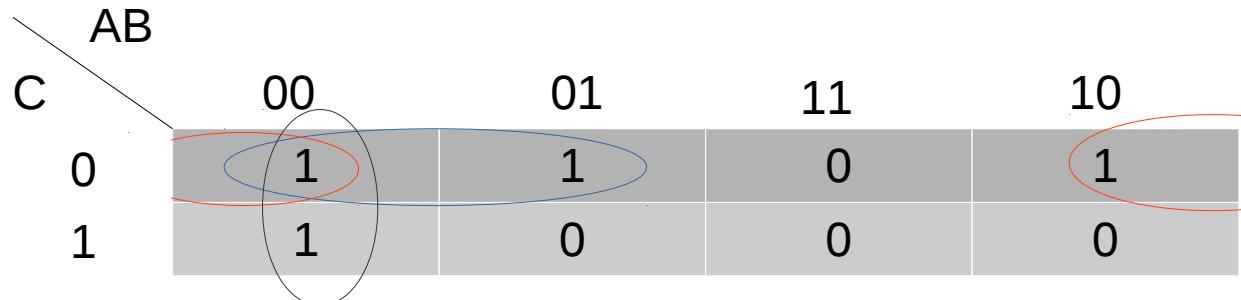
1) Forma normale del circuito

(sommatoria di termini, ciascuno dei quali costituito da una produttoria di letterali)

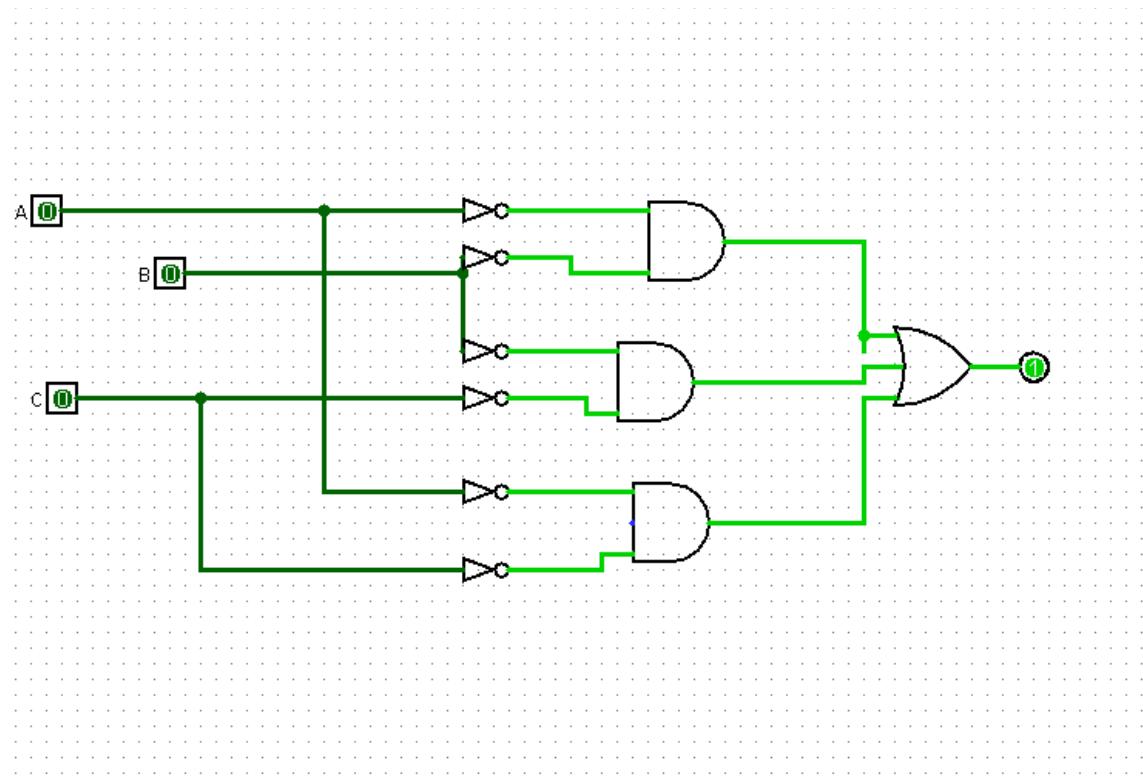
$$Y = (A' * B' * C') + (A' * B * C') + (A * B' * C') + (A' * B' * C)$$

2) Mappa di Karnaugh e riduzione (potenze di 2, scrivo valori invarianti)

$$Y = A'*B' + A'*C' + B'*C'$$



Esercizio 1



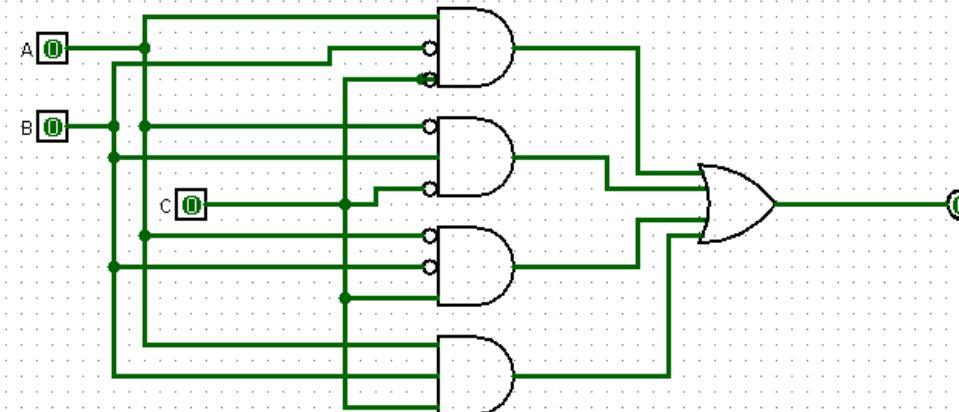
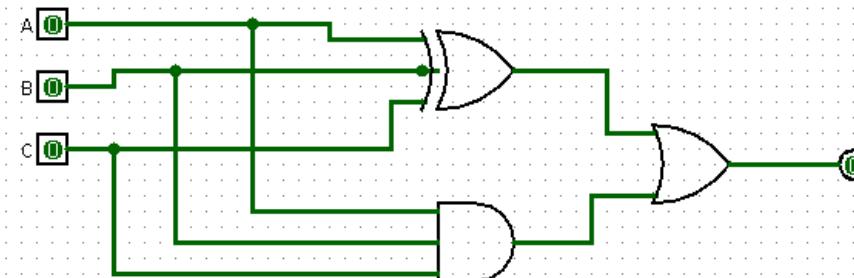
Esercizio 2

Si progetti un circuito combinatorio che simulala una lampadina comandata da tre diversi interruttori. I tre ingressi del circuito rappresentano lo stato degli interruttori e l'uscita rappresenta lo stato della lampadina. Il circuito deve soddisfare la condizione che ogni modifica allo stato di uno degli interruttori comporta un cambiamento dello stato della lampadina.

Per i più volenterosi: fornire due soluzioni per l'esercizio, la prima basata sulle sole porte AND, OR e NOT, quindi proporre una soluzione, più semplice, basata su porte XOR.

		AB	00	01	11	10
		C	0	1	0	1
0	0	0	1	0	1	
	1	1	0	1	0	

Esercizio 2



Esercizio 3

- Si progetti un circuito combinatorio che riceve come ingresso due numeri binari, ALPHA e BETA, di 2 bit ciascuno e genera una singola linea di uscita. Il circuito restituisce 1 se $\text{ALPHA} \leq \text{BETA}$, e 0 altrimenti.
- Si progetti quindi la versione duale del circuito, ossia un circuito dove l'uscita è data da una porta AND che riceve come ingresso le uscite di un certo numero di porte OR.

Circuito duale:

$$y = a + (b' * c)$$

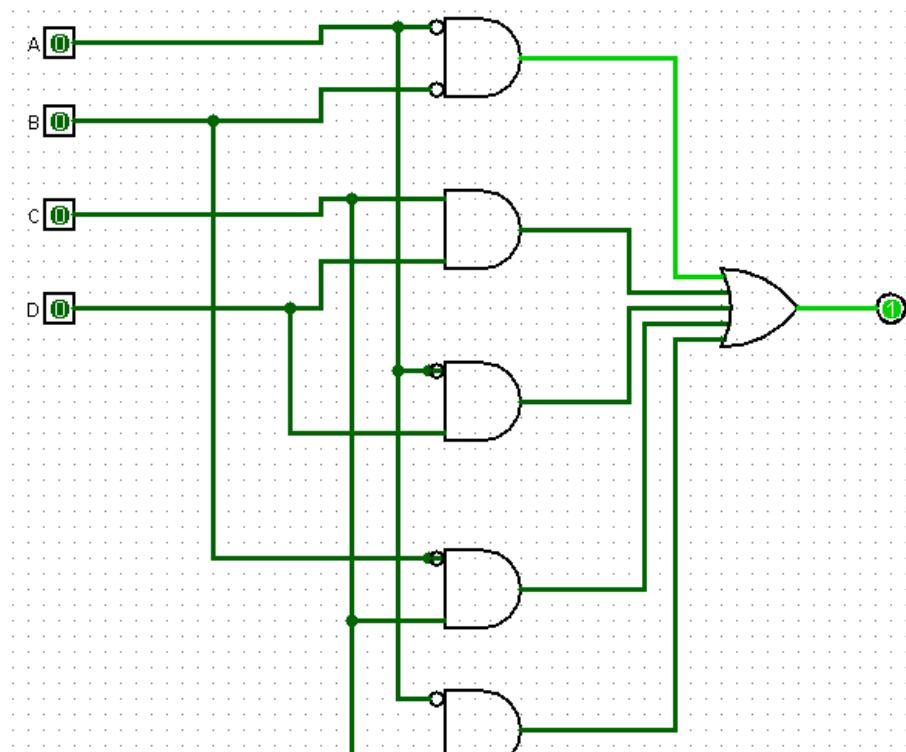
$$y' = a' * (b + c')$$

		AB				
		CD	00	01	11	10
		00	1	0	0	0
		01	1	1	0	0
		11	1	1	1	1
		10	1	1	0	1

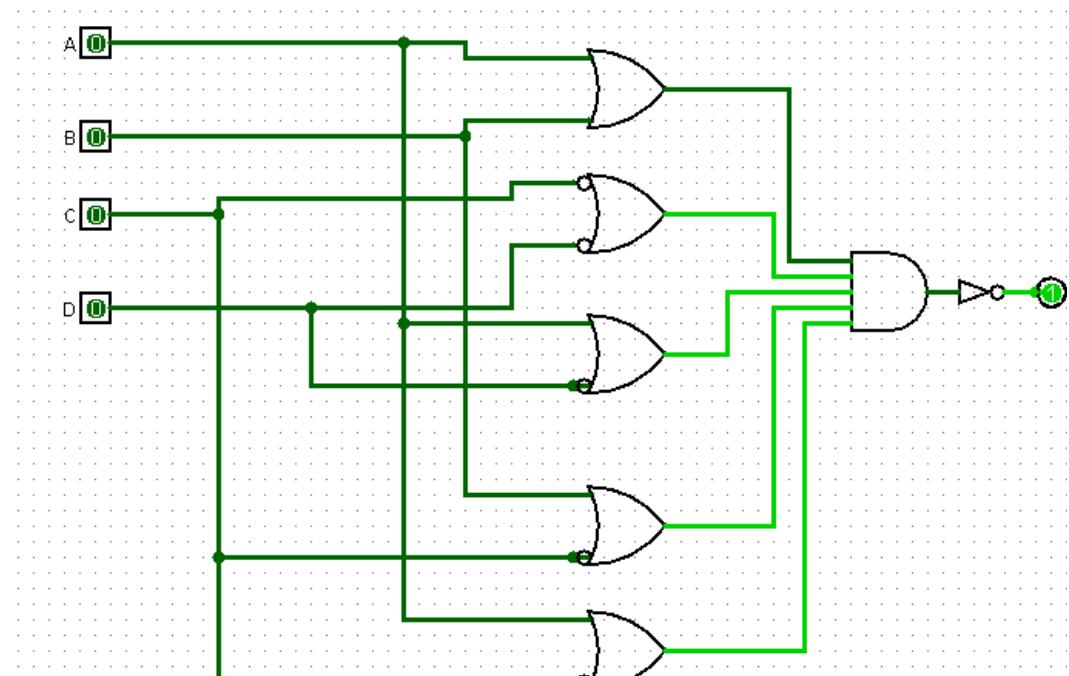
Progettazione duale:

Raggruppo gli zeri usando
Espressioni duali

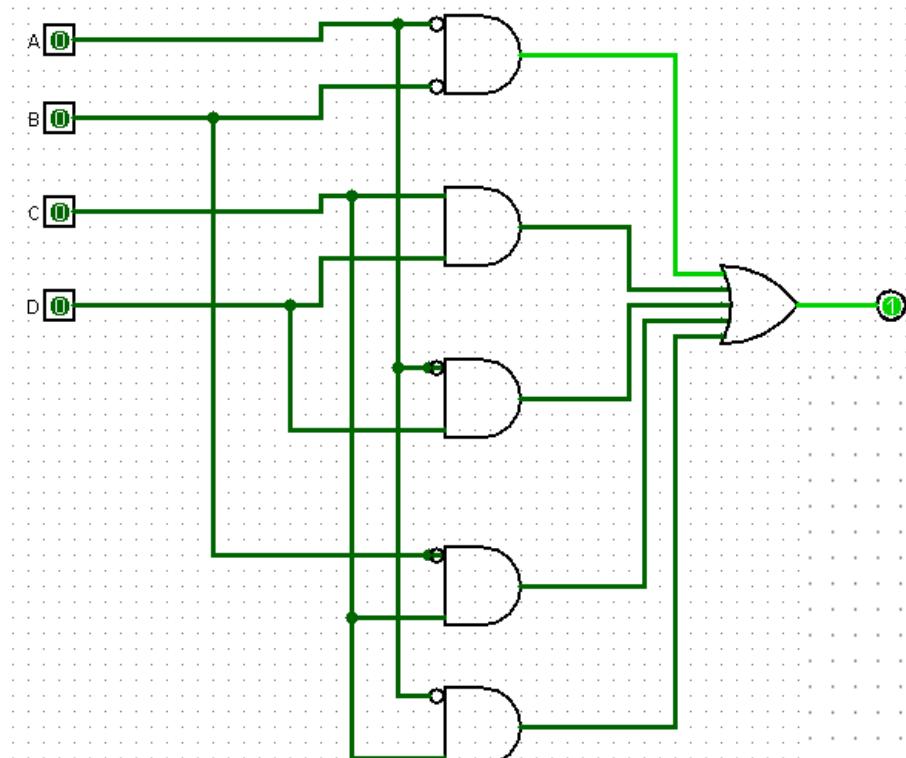
Esercizio 3



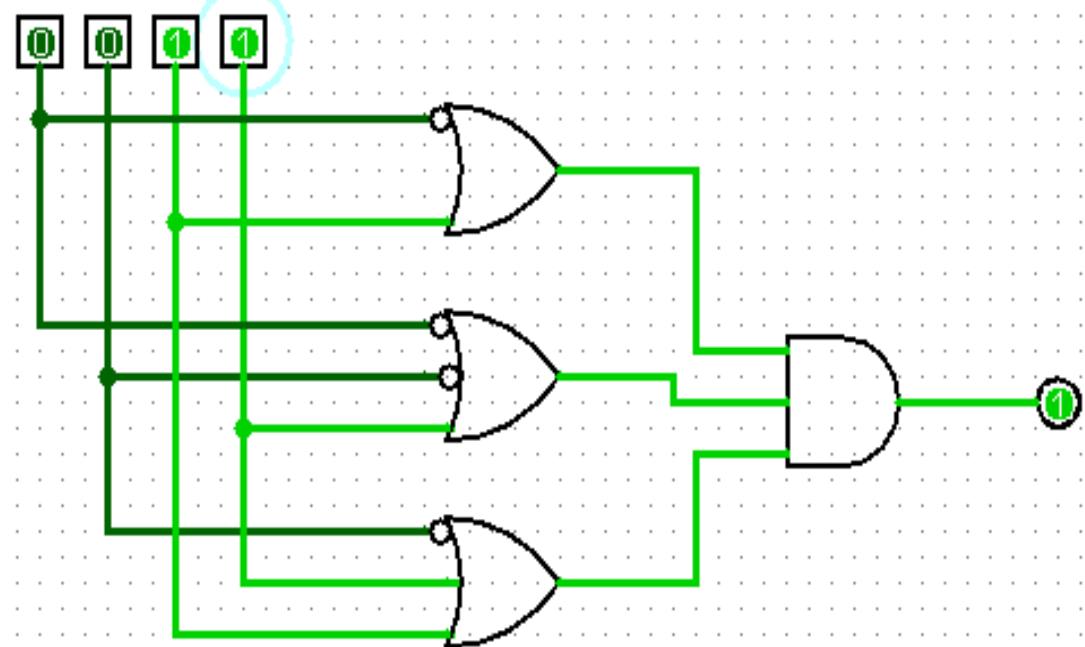
CIRCUITO DUALE NEGATO



Esercizio 3



PROGETTAZIONE DUALE



Laboratorio di Architettura degli Elaboratori

A.A. 2018/19 — Circuiti Logici

Per ogni lezione, sintetizzare i circuiti combinatori o sequenziali che soddisfino le specifiche date e quindi implementarli e testarne il comportamento mediante il programma simulatore Logisim.

Lezione 1

Mediante le mappe di Karnaugh, sintetizzare i circuiti *combinatori* che soddisfino le seguenti specifiche:

1.1 Esercizio

Si progetti un circuito combinatorio, con tre segnali di input, che calcola la minoranza. Il circuito fornisce in uscita 1 se almeno due ingressi sono 0, altrimenti genera 0 come uscita.

1.2 Esercizio

Si progetti un circuito combinatorio che simula una lampadina comandata da tre diversi interruttori. I tre ingressi del circuito rappresentano lo stato degli interruttori e l'uscita rappresenta lo stato della lampadina. Il circuito deve soddisfare la condizione che ogni modifica allo stato di uno degli interruttori comporta un cambiamento dello stato della lampadina.

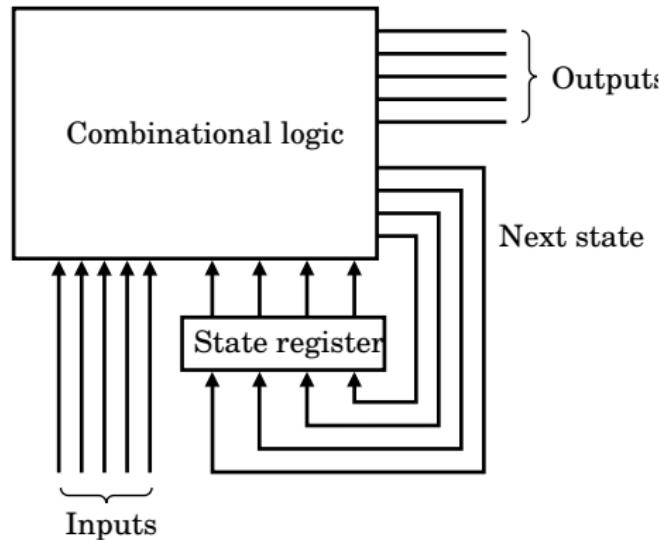
Per i più volenterosi: fornire due soluzioni per l'esercizio, la prima basata sulle sole porte AND, OR e NOT, quindi proporre una soluzione, più semplice, basata su porte XOR.

1.3 Esercizio

- Si progetti un circuito combinatorio che riceve come ingresso due numeri binari, A e B , di 2 bit ciascuno e genera una singola linea di uscita. Il circuito restituisce 1 se $A \leq B$, e 0 altrimenti.
- Si progetti quindi la versione duale del circuito, ossia un circuito dove l'uscita è data da una porta AND che riceve come ingresso le uscite di un certo numero di porte OR.

Progettazione di circuiti sequenziali

Circuito sequenziale tipico: combinatorio + memoria.



Esistono circuiti complessi, con svariati registri di memoria e circuiti combinatori.
Ci limitiamo a considerare circuiti semplici.

Macchina a stati finiti

È conveniente descrivere un circuito sequenziale come una **macchina a stati finiti** (MSF).

Stato: etichetta tutte le **configurazioni** in cui potrà trovarsi la macchina.

- In ogni istante la macchina si trova in una determinata condizione:
 - configurazione dello stato
 - valore delle variabili d'ingressodeterminano univocamente
 - valore delle uscite,
 - configurazione dell'**istante** successivo.

Una macchina a stati dunque introduce la nozione di **tempo discreto** sincronizzato da un **clock**.

Rappresentazioni grafiche MSF

La macchina a stati finiti viene rappresentata da un grafo: **nodi** connessi tramite **archi orientati**.

- Nodi: configurazioni dello stato.
- Archi orientati: transizioni dello stato.

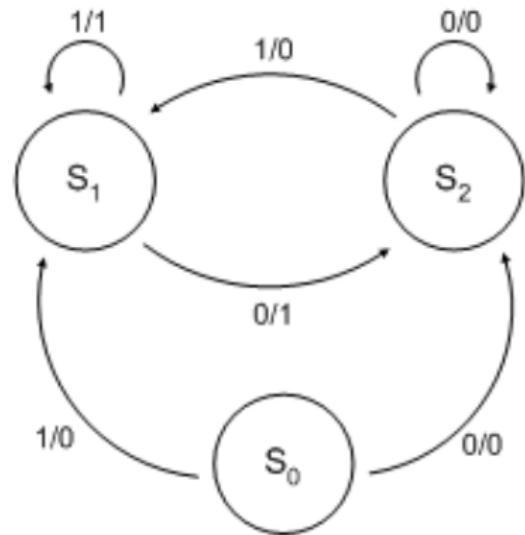
Nella **Macchina di Mealy** ogni arco è etichettato da

- valori di input
- valori di output, conseguenti al corrispondente input.

Rappresentazione del comportamento più intuitiva.

Utile nella progettazione della parte combinatoria.

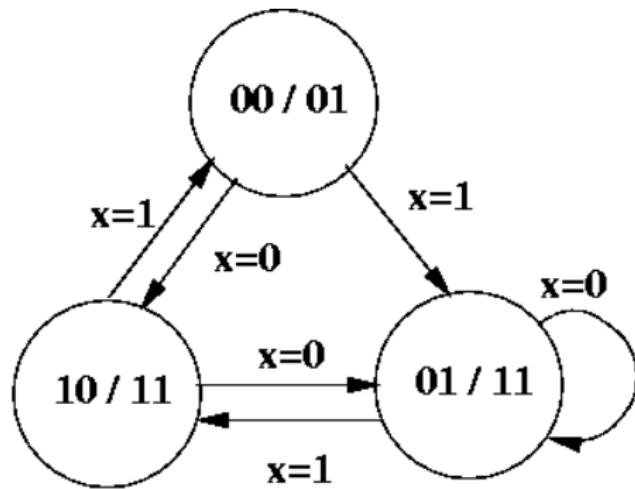
Grafo di una macchina di Mealy



Che comportamento ha?

Copia, con ritardo di un impulso del clock, l'ingresso nell'uscita.

Macchina di Moore



Nella **Macchina di Moore** il valore dell'uscita è associato allo stato, non all'ingresso.
Gli archi sono dunque etichettati con il solo input.
Non è meno potente della macchina di Mealy.

Progettazione

- Dalla descrizione del problema si determina una macchina a stati finiti che lo risolve
 - allocazione della memoria necessaria per descrivere il problema: lo stato (i nodi),
 - determinazione delle transizioni (gli archi)
- associo a ogni nodo un'etichetta binaria
- costruisco (ad esempio) le mappe di Karnaugh per le uscite del circuito combinatorio: stato successivo **e** uscita
- sintetizzo il circuito combinatorio
- è **sempre** necessario un segnale di clock per aggiornare lo stato in modo sincrono.

Esempi di progettazione

Circuito per il controllo di un semaforo, con rivelatori di presenza di traffico.

Comportamento:

- il semaforo può cambiare stato in corrispondenza al segnale di clock
- il semaforo cambia stato solo se sono presenti dei mezzi in attesa.

Semplificazione: due sole luci complementari, non esiste la luce arancio.

Segnali

- 2 ingressi: presenza di traffico sulla strada 1, presenza di traffico sulla strada 2
- 1 uscita: determina lo stato del semaforo (dall'unica uscita è possibile determinare le 4 luci del semaforo)
- 2 stati: luce verde sulla strada 1, luce rossa sulla strada 1.

Dal diagramma al circuito

Progetto del circuito combinatorio.

- Minimizzo la tabella di transizione per esempio adoperando le mappe di Karnaugh:
 - una mappa per ogni bit nel registro
 - una mappa per ogni variabile d'uscita.
- Dalle mappe di Karnaugh ricavo il circuito sequenziale.

Esercizi

- Contatore “up/down” a 2 bit:
 - 2 ingressi: x abilitazione al conteggio, ud ordine di conteggio
 - 2 uscite: numero binario.
- Circuito sequenziale per riconoscere una stringa (1100).
- Circuito sequenziale con:
 - 1 ingresso: numeri codificati come gruppi di 3 bit
 - 2 uscite: assumono il valore 00 in corrispondenza del primo e del secondo bit di ingresso; poi il numero (in binario) degli 1 ricevuti in ingresso.

Osservazioni

Macchina a stati finiti come architettura di calcolo dotata di una memoria limitata.

Concetto ricorrente in informatica.

Utilizzato per descrivere architetture in diversi contesti:

- componenti di calcolatori elettronici
- linguaggi formali
- organismi biologici.

128-bit word needs 8. This type of code is called a *Hamming code*, after R. Hamming, who described a method for creating such codes.

B.10 Finite State Machines

As we saw earlier, digital logic systems can be classified as combinational or sequential. Sequential systems contain state stored in memory elements internal to the system. Their behavior depends both on the set of inputs supplied and on the contents of the internal memory, or state of the system. Thus, a sequential system cannot be described with a truth table. Instead, a sequential system is described as a **finite state machine** (or often just *state machine*). A finite state machine has a set of states and two functions called the **next-state function** and the **output function**. The set of states correspond to all the possible values of the internal storage. Thus, if there are n bits of storage, there are 2^n states. The next-state function is a combinational function that, given the inputs and the current state, determines the next state of the system. The output function produces a set of outputs from the current state and the inputs. Figure B.10.1 shows this diagrammatically.

finite state machine A sequential logic function consisting of a set of inputs and outputs, a next-state function that maps the current state and the inputs to a new state, and an output function that maps the current state and possibly the inputs to a set of asserted outputs.

next-state function A combinational function that, given the inputs and the current state, determines the next state of a finite state machine.

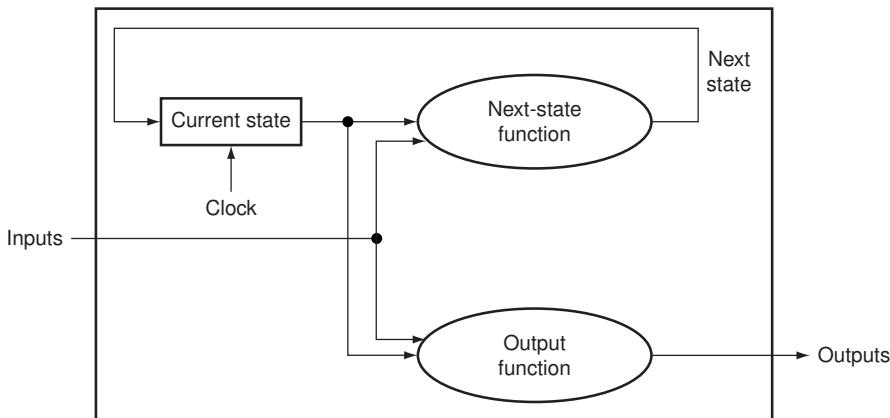


FIGURE B.10.1 A state machine consists of internal storage that contains the state and two combinational functions: the **next-state function** and the **output function**. Often, the output function is restricted to take only the current state as its input; this does not change the capability of a sequential machine, but does affect its internals.

The state machines we discuss here and in Chapters 5 and 6 are *synchronous*. This means that the state changes together with the clock cycle, and a new state is computed once every clock. Thus, the state elements are updated only on the clock edge. We use this methodology in this section and throughout Chapters 5 and 6, and we do not usually show the clock explicitly. We use state machines throughout Chapters 5 and 6 to control the execution of the processor and the actions of the datapath.

To illustrate how a finite state machine operates and is designed, let's look at a simple and classic example: controlling a traffic light. (Chapters 5 and 6 contain more detailed examples of using finite state machines to control processor execution.) When a finite state machine is used as a controller, the output function is often restricted to depend on just the current state. Such a finite state machine is called a *Moore machine*. This is the type of finite state machine we use throughout this book. If the output function can depend on both the current state and the current input, the machine is called a *Mealy machine*. These two machines are equivalent in their capabilities, and one can be turned into the other mechanically. The basic advantage of a Moore machine is that it can be faster, while a Mealy machine may be smaller, since it may need fewer states than a Moore machine. In Chapter 5, we discuss the differences in more detail and show a Verilog version of finite state control using a Mealy machine.

Our example concerns the control of a traffic light at an intersection of a north-south route and an east-west route. For simplicity, we will consider only the green and red lights; adding the yellow light is left for an exercise. We want the lights to cycle no faster than 30 seconds in each direction, so we will use a 0.033 Hz clock so that the machine cycles between states at no faster than once every 30 seconds. There are two output signals:

- *NSlite*: When this signal is asserted, the light on the north-south road is green; when this signal is deasserted the light on the north-south road is red.
- *EWlite*: When this signal is asserted, the light on the east-west road is green; when this signal is deasserted the light on the east-west road is red.

In addition, there are two inputs: *NScar* and *EWcar*.

- *NScar*: Indicates that a car is over the detector placed in the roadbed in front of the light on the north-south road (going north or south).
- *EWcar*: Indicates that a car is over the detector placed in the roadbed in front of the light on the east-west road (going east or west).

The traffic light should change from one direction to the other only if a car is waiting to go in the other direction; otherwise, the light should continue to show green in the same direction as the last car that crossed the intersection.

To implement this simple traffic light we need two states:

- *NSgreen*: The traffic light is green in the north-south direction.
- *EWgreen*: The traffic light is green in the east-west direction.

We also need to create the next-state function, which can be specified with a table:

Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

Notice that we didn't specify in the algorithm what happens when a car approaches from both directions. In this case, the next-state function given above changes the state to ensure that a steady stream of cars from one direction cannot lock out a car in the other direction.

The finite state machine is completed by specifying the output function:

Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

Before we examine how to implement this finite state machine, let's look at a graphical representation, which is often used for finite state machines. In this representation, nodes are used to indicate states. Inside the node we place a list of the outputs that are active for that state. Directed arcs are used to show the next-state function, with labels on the arcs specifying the input condition as logic functions. Figure B.10.2 shows the graphical representation for this finite state machine.

A finite state machine can be implemented with a register to hold the current state and a block of combinational logic that computes the next-state function and the output function. Figure B.10.3 shows how a finite state machine with 4 bits of state, and thus up to 16 states, might look. To implement the finite state machine in this way, we must first assign state numbers to the states. This process is called *state assignment*. For example, we could assign NSgreen to state 0 and

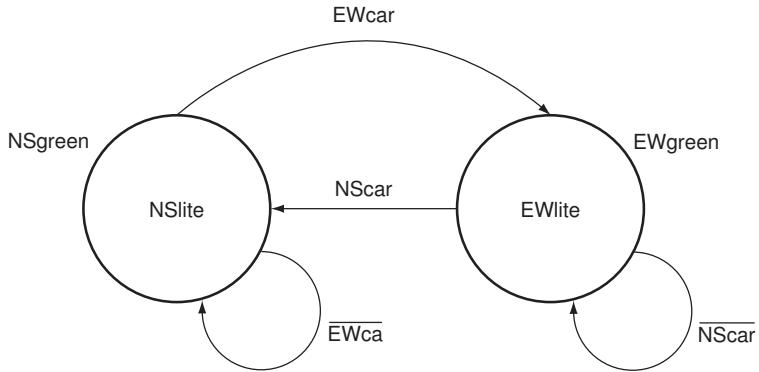


FIGURE B.10.2 The graphical representation of the two-state traffic light controller. We simplified the logic functions on the state transitions. For example, the transition from NSgreen to EWgreen in the next-state table is $(\overline{NScar} \cdot EWcar) + (NScar \cdot \overline{EWcar})$, which is equivalent to $EWcar$.

EWgreen to state 1. The state register would contain a single bit. The next-state function would be given as

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

where CurrentState is the contents of the state register (0 or 1) and NextState is the output of the next-state function that will be written into the state register at the end of the clock cycle. The output function is also simple:

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

The combinational logic block is often implemented using structured logic, such as a PLA. A PLA can be constructed automatically from the next-state and output function tables. In fact, there are computer-aided design (CAD) programs that take either a graphical or textual representation of a finite state machine and produce an optimized implementation automatically. In Chapters 5 and 6, finite state machines were used to control processor execution. Appendix C discusses the detailed implementation of these controllers with both PLAs and ROMs.

To show how we might write the control in Verilog, Figure B.10.4 shows a Verilog version designed for synthesis. Note that for this simple control function, a Mealy machine is not useful, but this style of specification is used in Chapter 5 to implement a control function that is a Mealy machine and has fewer states than the Moore machine controller.

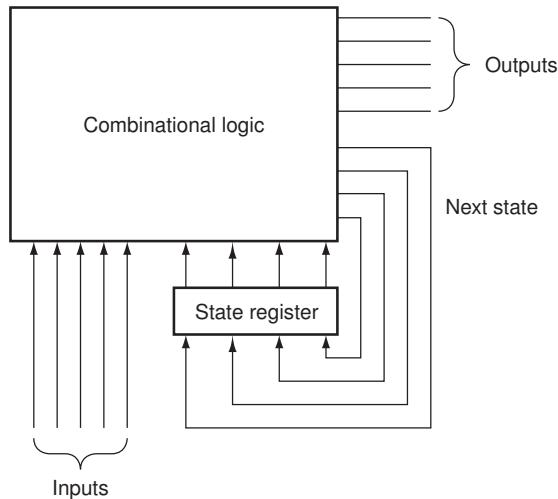


FIGURE B.10.3 A finite state machine is implemented with a state register that holds the current state and a combinational logic block to compute the next state and output functions. The latter two functions are often split apart and implemented with two separate blocks of logic, which may require fewer gates.

```
module TrafficLite (EWCAR, NSCAR, EWLITE, NSLITE, clock);
    input EWCAR, NSCAR, clock;
    output EWLITE, NSLITE;

    reg state;
    initial state=0; //set initial state
    //following two assignments set the output, which is based only on the state
    //variable
    assign NSLITE = ~ state; //NSLITE on if state = 0;
    assign EWLITE = state; //EWLITE on if state =1

    always @(posedge clock) // all state updates on a positive clock edge
        case (state)
            0: state = EWCAR; //change state only if EWCAR
            1: state = NSCAR; //change state only if NSCAR
        endcase
endmodule
```

FIGURE B.10.4 A Verilog version of the traffic light controller.

**Check
Yourself**

What is the smallest number of states in a Moore machine for which a Mealy machine could have fewer states?

- a. Two, since there could be a one-state Mealy machine that might do the same thing.
- b. Three, since there could be a simple Moore machine that went to one of two different states and always returned to the original state after that. For such a simple machine, a two-state Mealy machine is possible.
- c. You need at least four states to exploit the advantages of a Mealy machine over a Moore machine.

B.11**Timing Methodologies**

Throughout this appendix and in the rest of the text, we use an edge-triggered timing methodology. This timing methodology has the advantage that it is simpler to explain and understand than a level-triggered methodology. In this section, we explain this timing methodology in a little more detail and also introduce level-sensitive clocking. We conclude this section by briefly discussing the issue of asynchronous signals and synchronizers, an important problem for digital designers.

The purpose of this section is to introduce the major concepts in clocking methodology. The section makes some important simplifying assumptions; if you are interested in understanding timing methodology in more detail, consult one of the references listed at the end of this appendix.

We use an edge-triggered timing methodology because it is simpler to explain and has fewer rules required for correctness. In particular, if we assume that all clocks arrive at the same time, we are guaranteed that a system with edge-triggered registers between blocks of combinational logic can operate correctly without races, if we simply make the clock long enough. A *race* occurs when the contents of a state element depend on the relative speed of different logic elements. In an edge-triggered design, the clock cycle must be long enough to accommodate the path from one flip-flop through the combinational logic to another flip-flop where it must satisfy the set-up time requirement. Figure B.11.1 shows this requirement for a system using rising edge-triggered flip-flops. In such a system the clock period (or cycle time) must be at least as large as

$$t_{\text{prop}} + t_{\text{combinational}} + t_{\text{setup}}$$

for the worst-case values of these three delays, which are defined as follows:

Propagazione del ritardo

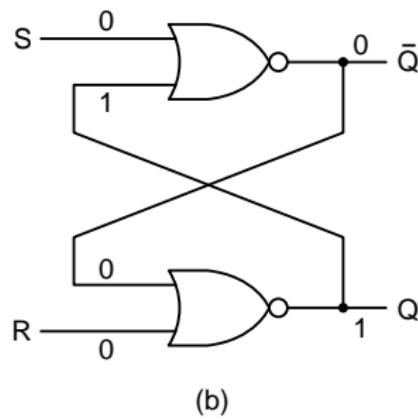
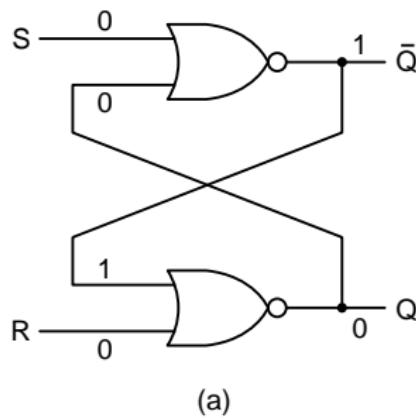
- I circuiti logici rispondono con un piccolissimo ritardo: attorno a 10^{-10} s
- in presenza di porte logiche **in cascata** i ritardi si sommano
- la realizzazione tradizionale del sommatore richiede una lunga cascata per la propagazione del riporto: implementazione **lenta**.

Per ottenere circuiti più veloci la somma usa circuiti più sofisticati.

Circuiti con memoria

Circuiti con memoria: l'input passato ha effetti sul comportamento corrente.

Il più semplice circuito con memoria:
latch Set–Reset (SR), sfrutta la **retroazione**.



A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0

Latch SR

- Il segnale S (Set) a 1 porta l'uscita Q a 1.
- Il segnale R (Reset) a 1 porta l'uscita Q a 0.

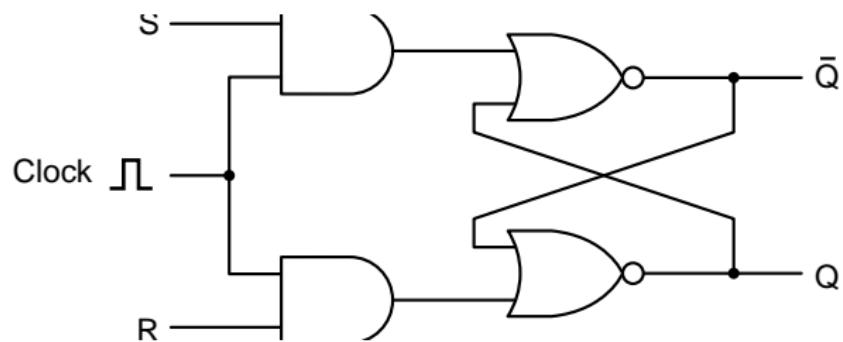
Posso **memorizzare un bit**.

Infatti, cessato il segnale d'ingresso, con input $S = 0$, $R = 0$ assume uno tra **due stati stabili complementari** in dipendenza dal passato.

Nota: l'input $S = 1$, $R = 1$ forza le due uscite complementari ad assumere lo stesso valore 0, incoerente rispetto al funzionamento del latch SR.

Latch sincronizzato

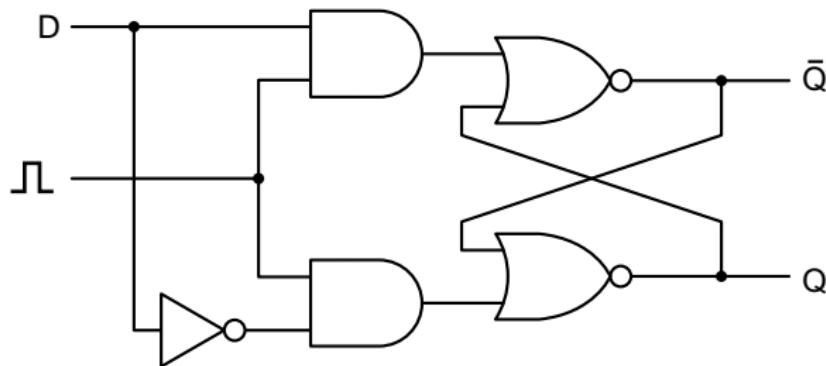
Segnale di clock (enable, strobe) per l'abilitazione alla scrittura.



Quando il segnale di clock è 0 la scrittura è disabilitata.

Latch di tipo D

Differisce per i segnali di controllo.

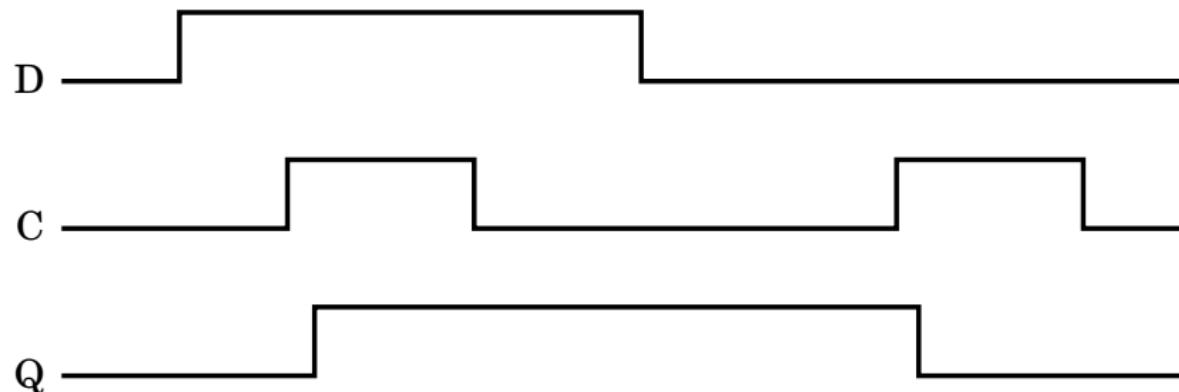


Quando il clock è abilitato (= 1) memorizza il segnale D.

Flip-flop

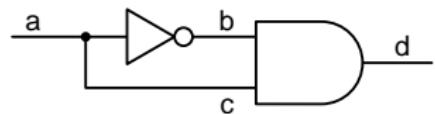
Diversi dai latch per il comportamento rispetto al clock: possono cambiare stato **solo nell'istante in cui il clock cambia valore.**

Esempio di comportamento:

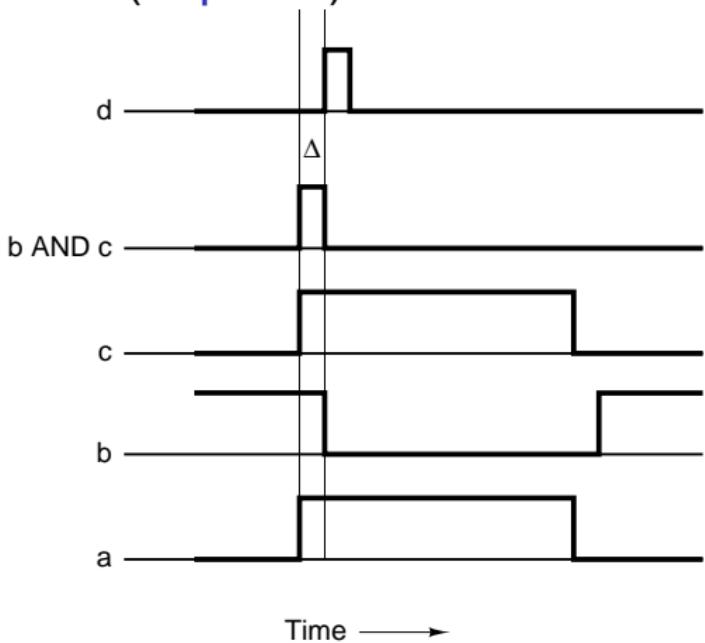


Implementazione con clock a impulsi

Si sfruttano i ritardi delle porte logiche per generare un segnale 1 brevissimo (**impulso**):



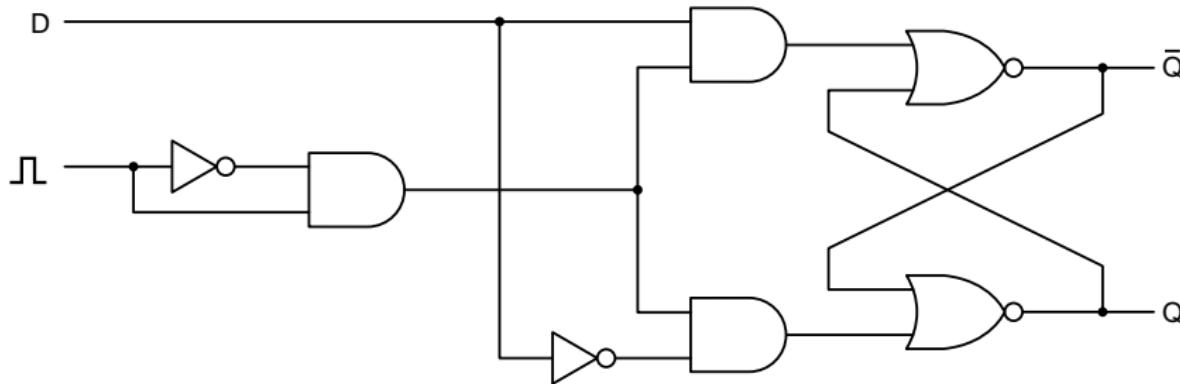
(a)



(b)

Flip-flop completo

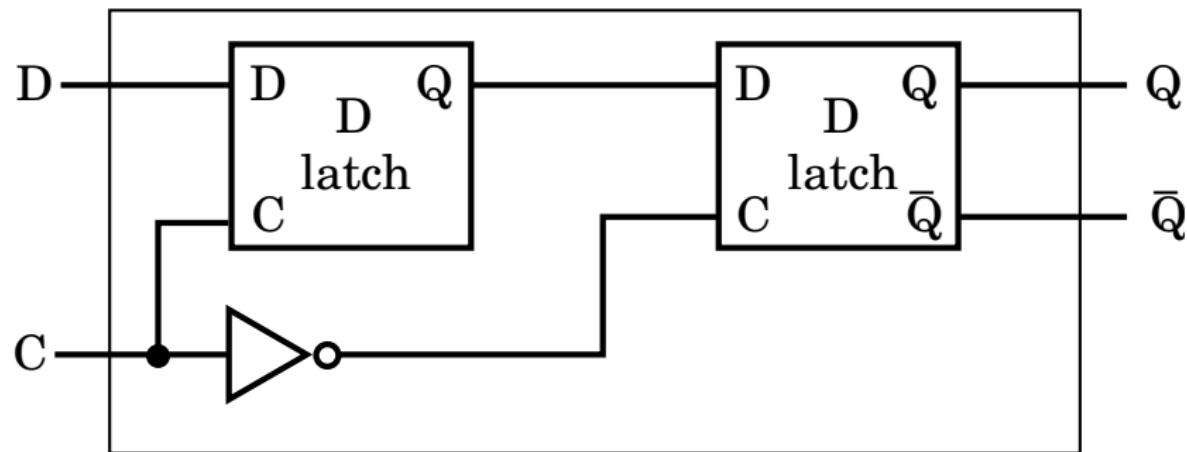
L'impulso abilita la scrittura:



Il flip-flop garantisce la persistenza dell'uscita secondo la sincronizzazione data dal clock.

Flip-flop Master-Slave

Implementazione alternativa, più affidabile:



È più un latch che un flip-flop.

Latch e Flip-flop

Classificazione:

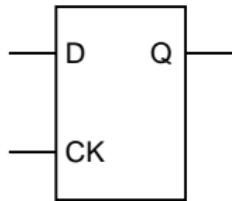
- Latch: level-triggered (azionato dal livello)
- Flip-Flop: edge-triggered (azionato dal fronte).

Vari tipi di latch/flip-flop:

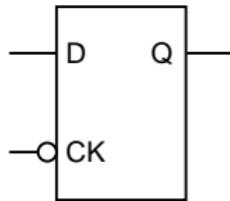
- S-R
- D
- J-K: come S-R ma cambia stato con $J=1, K=1$
- T: un solo ingresso, cambia stato con $T=1$.

Attenzione: non tutti concordano sulla classificazione qui adottata!

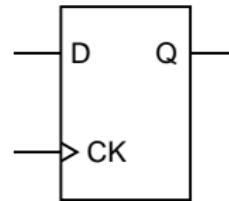
Rappresentazione grafica



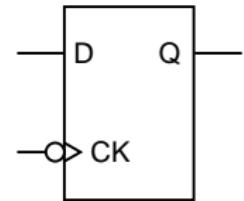
(a)



(b)



(c)



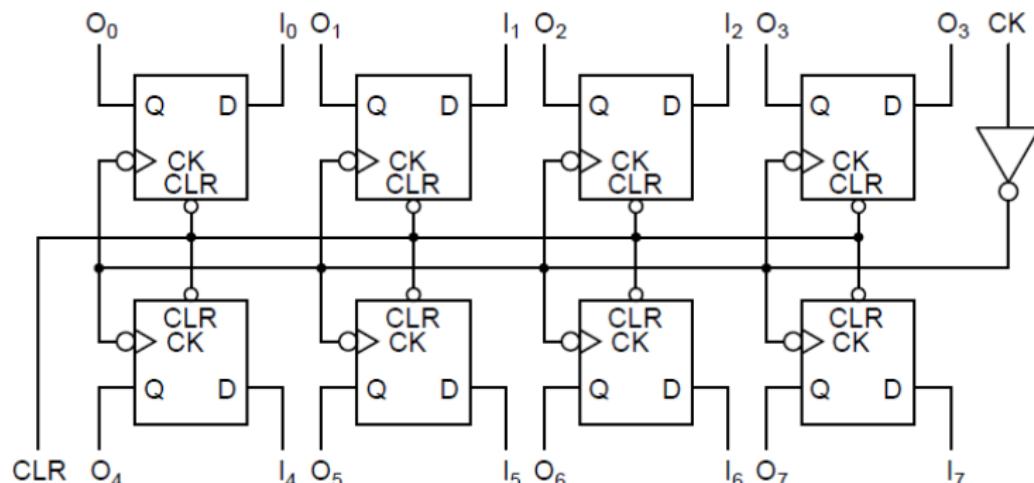
(d)

- (a), (b): latch con risposta complementare al segnale di clock
- (c), (d): flip-flop con risposta complementare al segnale di clock.

Registri

Collezione di elementi di un bit per memorizzare sequenze binarie di N bit (tipicamente $N = 8, 16, 32, 64$).

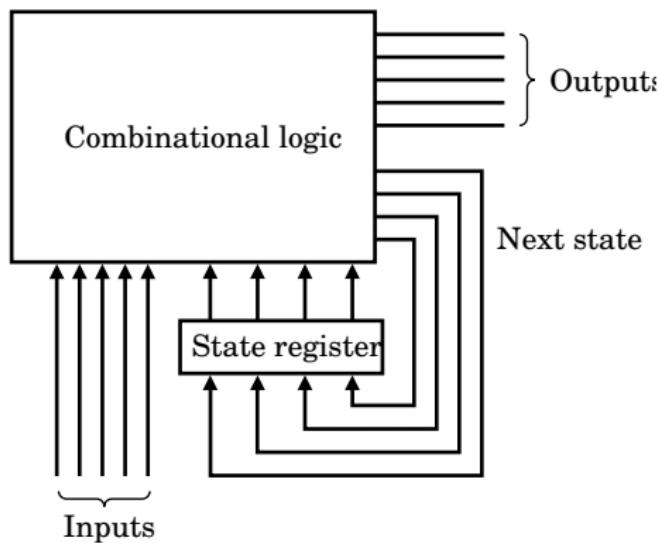
Realizzazione: sequenza di N flip-flop controllati da un comune segnale di clock.



Circuiti sequenziali

L'uscita dipende dall'ingresso attuale **e** dal passato:

$$\begin{cases} \mathbf{y}_n = g(\mathbf{x}_n, \mathbf{s}_{n-1}) \\ \mathbf{s}_n = h(\mathbf{x}_n, \mathbf{s}_{n-1}) \end{cases}$$



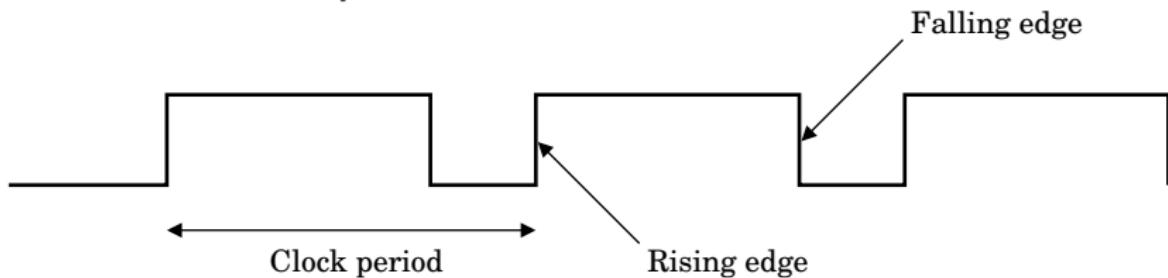
Funzionamento

Il circuito evolve secondo certe regole:

- attraverso passaggi da uno stato a quello successivo
- passaggi sincronizzati dall'impulso di clock, il quale abilita la scrittura nel registro
- il segnale di clock è **periodico**
- il passaggio di stato può avvenire solo quando il circuito si è **stabilizzato** (ritardi).

Segnale di clock

Segnale periodico che scandisce il funzionamento dei circuiti sequenziali.



Periodico: esiste un intervallo in cui si ripete identico.
Tipicamente ogni intervallo contiene un valore di tensione alto e uno basso.

Frequenza di clock = $1 / \text{periodo}$ [Hz].

In un calcolatore coesistono vari segnali di clock:
processore, scheda grafica, bus di sistema, . . .

Periodo di clock

Due esigenze contrapposte:

- massimizzazione teorica delle prestazioni rendendo il periodo di clock più breve possibile
- ogni circuito ha un **tempo di commutazione** che non può essere superiore al periodo di clock.

Ordini di grandezza del periodo: $1 \sim 10$ ns,
corrispondenti a una frequenza di $100\text{ MHz} \sim 1\text{ GHz}$.

Laboratorio di architettura degli elaboratori

CIRCUITI COMBINATORI lezione 2

CONTATTI

- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

Esercizio 2.1

Progettare un circuito che, ricevuti 4 segnali binari (bit) in ingresso, stabilisca se questi rappresentano nella notazione binaria un numero primo. (consideriamo 1 non primo)

Il circuito restituisce in uscita 1 se l'input rappresenta un numero primo, mentre restituisce 0 in caso contrario.

CDAB

0000 → 0

0001 → 0

0010 → 1

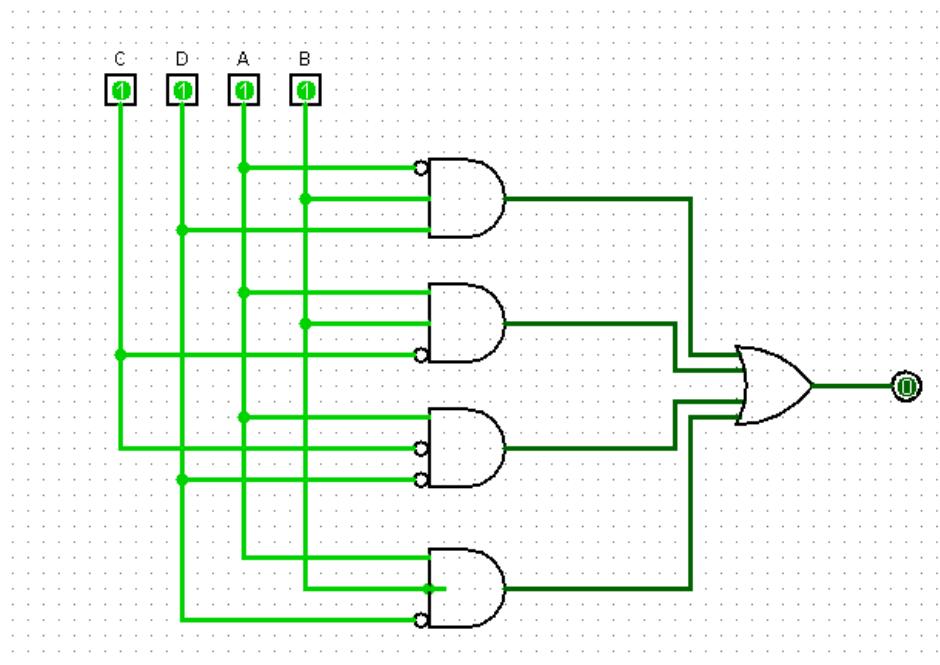
0011 → 1

...
1111 → 0

$$Y = A' * B * D + A * B * C' + A * C' * D' + A * B * D'$$

		AB				
		CD	00	01	11	10
CDAB	00	0	0	1	1	1
	01	0	1	1	0	0
	11	0	1	0	0	0
	10	0	0	1	0	0

Esercizio 2.1



Esercizio 2.2

Progettare un circuito che riceve in ingresso un numero binario di 4 bit. Il circuito restituisce 1 se l'ingresso rappresenta un numero decimale tra 0 e 9 divisibile per 2 o per 5. Viceversa restituisce 0 se lo stesso numero non è divisibile né per 2 né per 5.

Nel caso in cui l'ingresso sia un numero maggiore di 9, l'uscita può assumere un valore arbitrario.

$$2,4,5,6,8 \rightarrow 1$$

$$0,1,3,7,9 \rightarrow 0$$

$$10,11,12,13,14,15 \rightarrow 0 \text{ oppure } 1$$

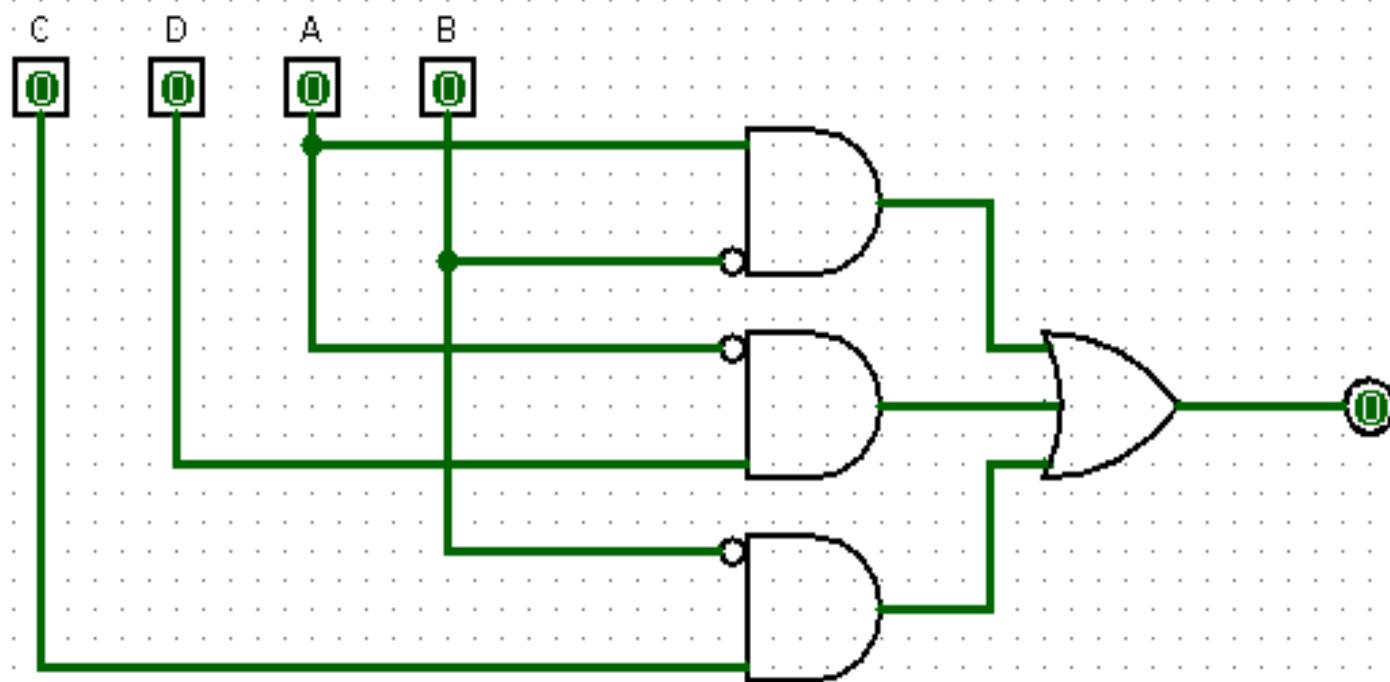
$$Y = A'B' + A'D + B'C$$

AB	CD					
	00	01	11	10		
00	0	0	0	1		
01	1	1	0	1		
11	X	X	X	X		
10	1	0	X	X		

The truth table shows the output Y for various binary inputs. Inputs are labeled AB (columns) and CD (rows). The output Y is 1 for inputs (0,0), (0,1), (1,0), and (1,1). It is X for inputs (1,1) and (1,0). It is 0 for inputs (0,0), (0,1), (1,0), and (1,1).

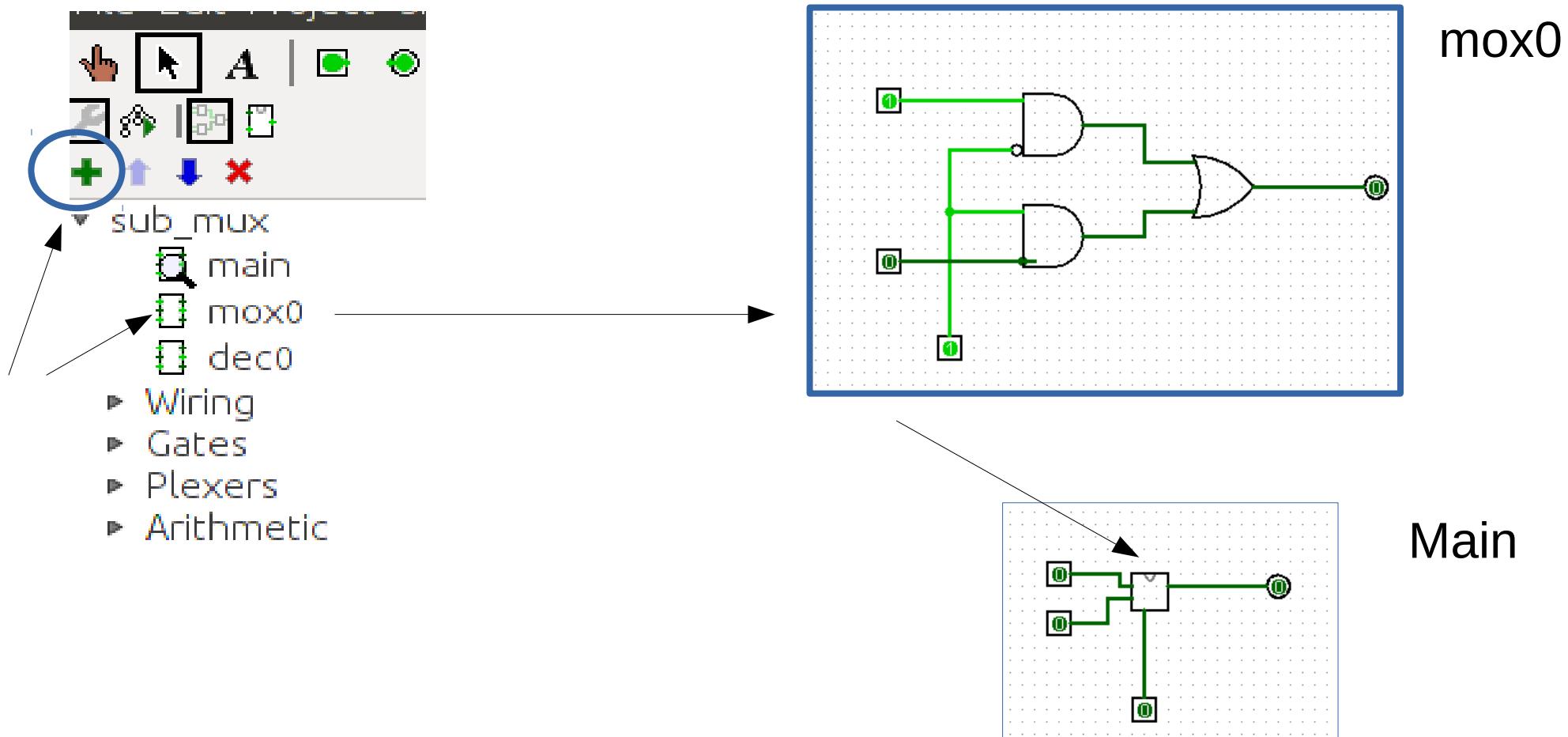
A blue rectangle highlights the row where CD=01 and the column where AB=01. A red rectangle highlights the row where CD=10 and the column where AB=01. A green oval encloses the row where CD=10 and the column where AB=10.

Esercizio 2.2



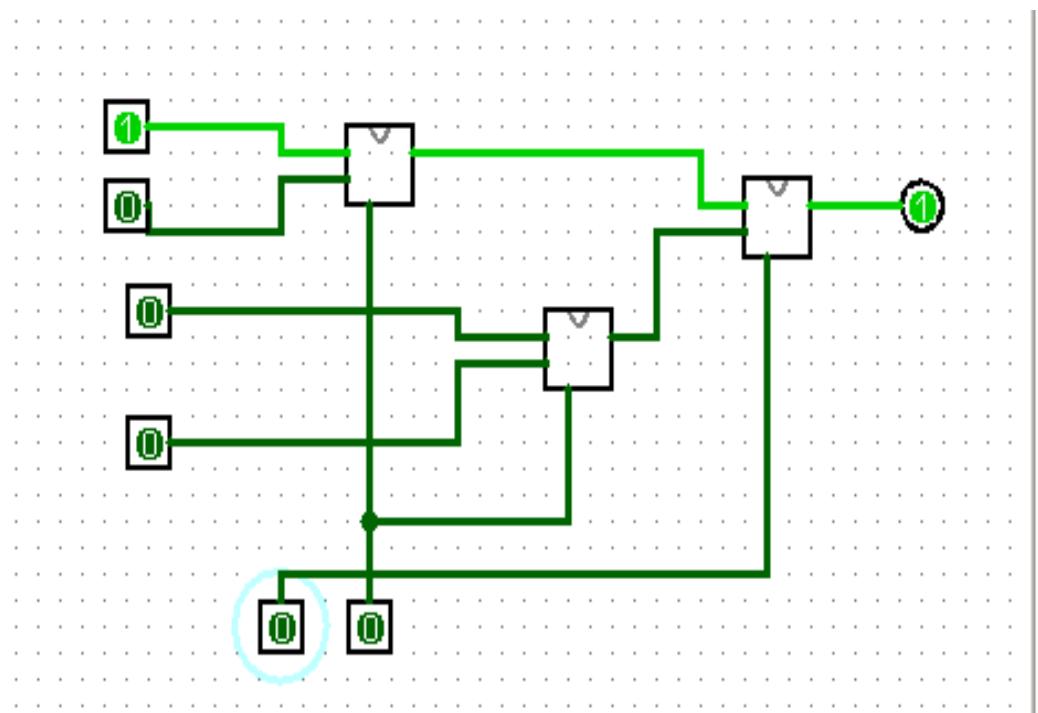
Esercizio 2.3.1

Costruire un **multiplexer**, con 1 ingresso di controllo, e realizzarlo come modulo Logisim.



Esercizio 2.3.2

Realizzare un multiplexer con 2 ingressi di controllo utilizzando tre multiplexer con 1 ingresso di controllo.



Lezione 2

2.1 Esercizio

- Progettare un circuito che, ricevuti 4 segnali binari (bit) in ingresso, stabilisca se questi rappresentano nella notazione binaria un numero primo. Il circuito restituisce in uscita 1 se l'input rappresenta un numero primo, mentre restituisce 0 in caso contrario.
- Progettare un circuito che riceva in ingresso un numero binario di 4 bit. Il circuito restituisce in uscita 1 se l'ingresso è una cifra decimale (ossia un valore tra 0 e 9) divisibile per 2 o per 5; restituisce 0 se l'ingresso è una cifra non divisibile né per 2 né per 5. Infine, nel caso in cui l'ingresso non rappresenti alcuna cifra decimale, l'uscita può assumere un valore arbitrario.

2.2 Esercizio

- Costruire un multiplexer con 1 ingresso di controllo, e realizzarlo come modulo Logisim.
- Utilizzando tre multiplexer con 1 ingresso di controllo realizzare un multiplexer con 2 ingressi di controllo.

2.3 Esercizio

- Progettare un decoder a 2 ingressi dotato di un segnale aggiuntivo di Enable. Se il segnale Enable vale 0 tutte le uscite valgono 0; se Enable vale 1 si comporta come un circuito decoder. Realizzare il circuito come modulo.
- Utilizzare il modulo del punto precedente per realizzare un decoder a 3 ingressi e uno a 4 ingressi.

Tecnologia dei circuiti integrati

Circuiti integrati (Integrated Circuit, IC, chip): unità contenenti circuiti logici.

- Piastrina di cristallo di silicio, tipicamente $\sim 1 \times 1$ cm.
- Sulla superficie vengono creati transistor, resistori, collegamenti, eventualmente condensatori.

Lavorazioni sul silicio

- Transistor: ottenuti drogando il **silicio**, cioè inserendo atomi estranei (boro, arsenico, fosforo) nella sua struttura cristallina.
Si espone il silicio, in forno, ai vapori di altre sostanze
- **collegamenti** tra i componenti elettronici, ottenuti depositando uno strato di materiale conduttore (rame o alluminio)
- **isolamenti** elettrici ottenuti ossidando il silicio, esponendolo in forno a ossigeno.

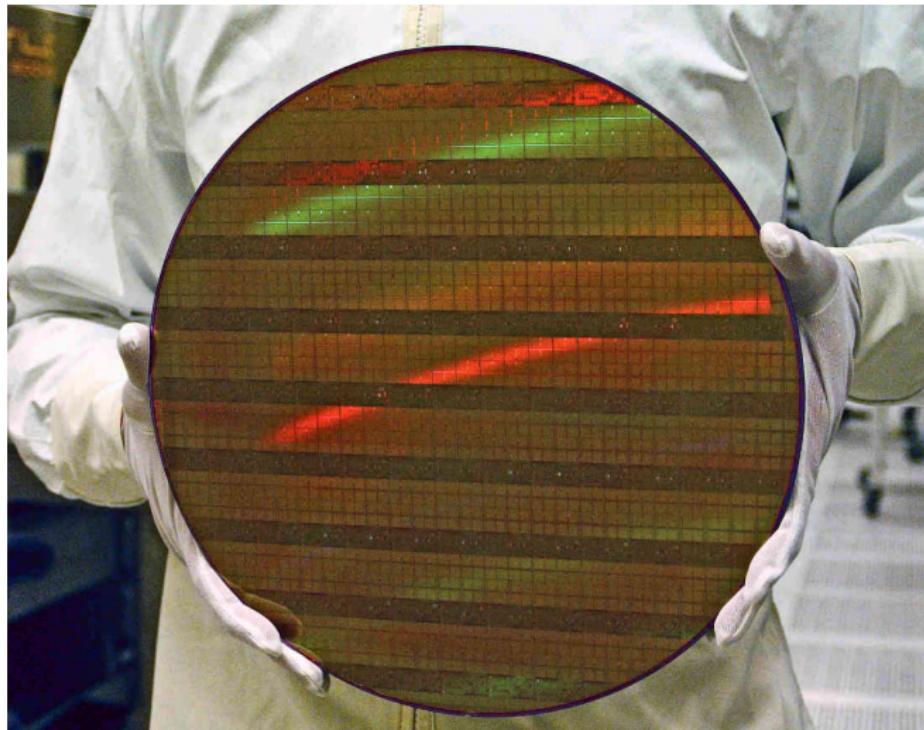
Tecniche di fotolitografia

Il silicio viene lavorato coprendolo selettivamente con uno strato di materiale fotosensibile. Illuminato in maniera differenziata,

- la parte illuminata solidifica
- la parte al buio viene rimossa.

Anche 50 diverse lavorazioni per singolo chip.

Wafer



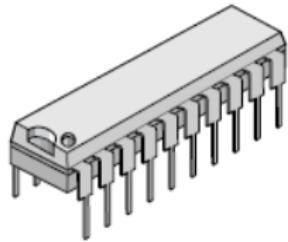
Package

Ogni chip è inglobato in un supporto di plastica:
package.

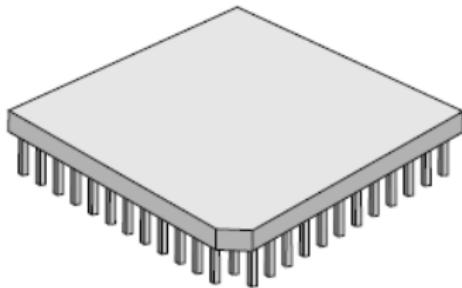
Connessioni mediante **piedini**:

- chip di memoria e chip semplici: due file di piedini (**dual in line package**)
- chip con processori: centinaia di connessioni, due file di piedini non sufficienti, pedinatura più complessa.

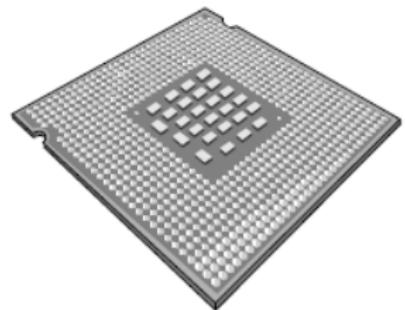
Package



(a)



(b)



(c)

Chip di memoria

Circuiti integrati contenenti un notevole numero di registri, raccolti in **locazioni**.

I singoli registri non sono tutti individualmente accessibili.

Per accedere ai dati

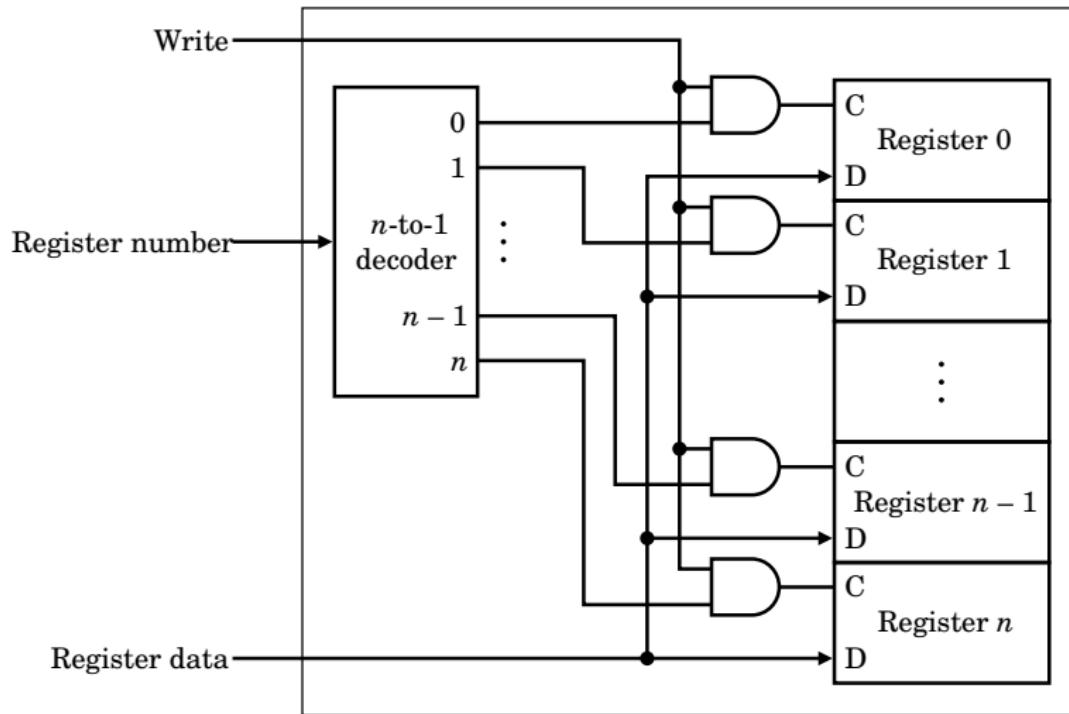
- si seleziona la locazione su cui operare, specificando il suo **indirizzo** binario
- si definisce l'operazione da eseguire (**lettura/scrittura**).

Chip di memoria

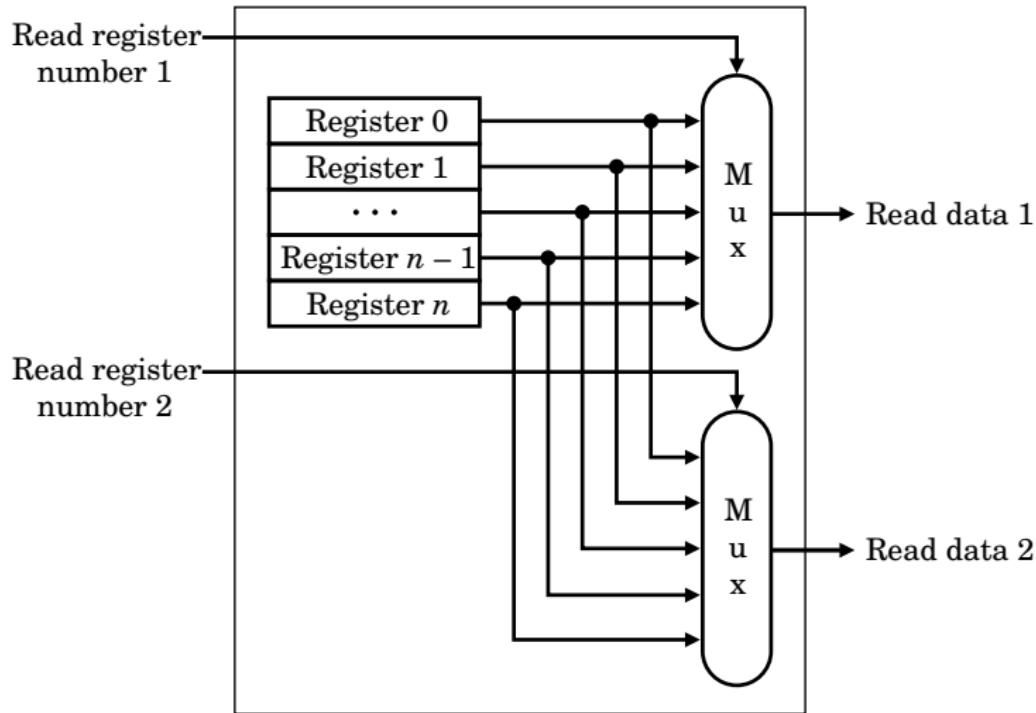
Segnali I/O:

- **indirizzo** (specifica il registro su cui operare)
- **dati in ingresso** (da scrivere nel registro)
- **segnali di controllo:**
 - **chip select** (CS) per attivare il chip di memoria
 - **read** (RD) specifica se vogliamo leggere o scrivere in memoria
 - **output enable** (OE)
- **dati in uscita** (nella pratica le linee d'uscita coincidono con gli ingressi).

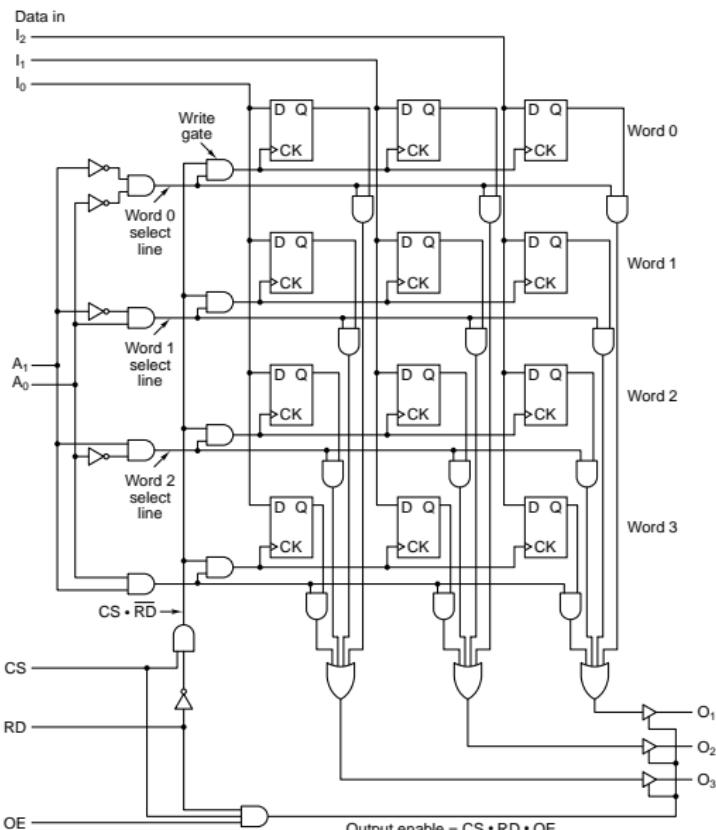
Schema di input



Schema di output



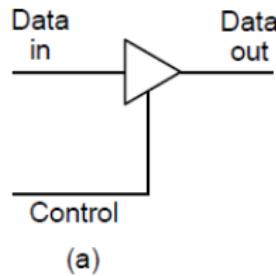
Implementazione



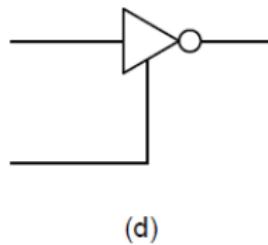
Buffer non invertenti, circuiti a tre stati

Per connettere tra di loro diverse uscite i **buffer non invertenti** sono più efficaci delle porte AND in quanto possono lasciare l'uscita indeterminata (non forzano un valore di tensione in uscita).

I **buffer invertenti** complementano il valore dell'uscita.



—



(d)

Memorie RAM

I circuiti di memoria riscrivibile vengono chiamati tradizionalmente **RAM** (Random Access Memory).

Due tipi:

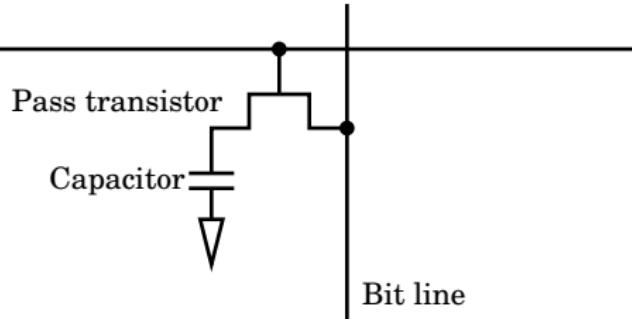
- RAM **statiche** (SRAM): i singoli bit vengono memorizzati con latch. Veloci e costose, sei transistor per memorizzare un bit. Utilizzate nella memoria **cache** (di basso livello)
- RAM **dinamiche** (DRAM): usano un diverso meccanismo di memorizzazione, lente e capienti. Costituiscono la **memoria principale** del calcolatore.

RAM Dinamiche (DRAM)

Un singolo transistor più un condensatore per memorizzare un bit.

Si posso inserire molte più celle di memoria in un singolo chip.

Word line



RAM Dinamiche (DRAM)

L'accumulo di carica nel condensatore rappresenta lo stato: la presenza di carica è associata a una tensione alta, e viceversa.

L'accesso al condensatore è controllato dal transistor, abilitato o no dalla **word** line.

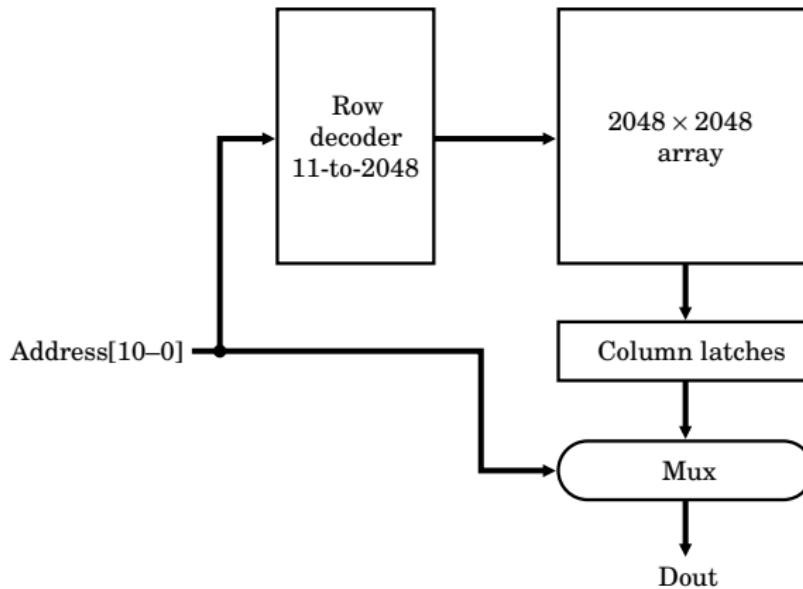
Pregi e difetti:

- più **economiche e compatte** delle SRAM
- più **lente** delle SRAM
- complessità del **controllo**: i condensatori perdono la loro carica in circa 1 ms; occorre una circuiteria dedicata di **refresh** della carica che impegnava circa il 10% di ogni ciclo di clock.

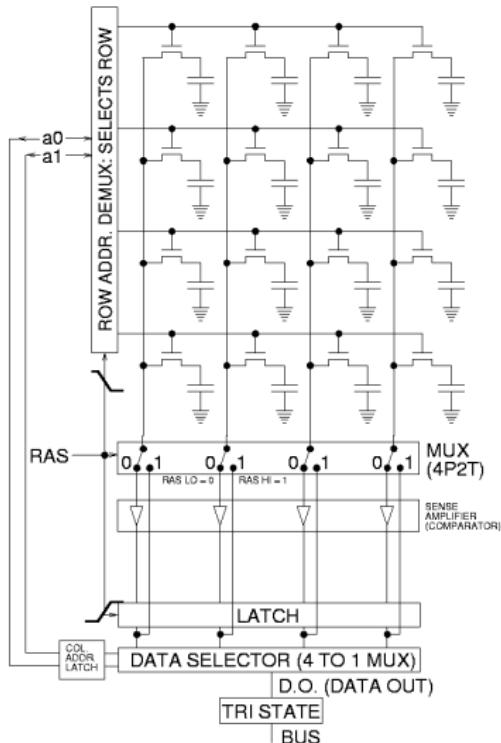
Indirizzamento DRAM

In due fasi:

- **RAS**: row access strobe
- **CAS**: column access strobe



DRAM



(by Glogger at English Wikipedia).

Struttura DRAM

Accesso alla memoria in **due fasi**,

- nella **prima fase** il contenuto di un intera riga viene copiato in un registro (latch),
- nella **seconda** vengono letti i bit selezionati della riga.

Accesso veloce a locazioni consecutive: non si ripete la prima fase, si usa il registro.

Tecnologie per le DRAM

I miglioramenti nei tempi di risposta delle DRAM sono state inferiori a quelli del processore; per un certo periodo di tempo: ($\sim 10\%$ vs $\sim 50\%$ l'anno).

La velocità relativa della memoria diminuisce: processore 100-1000 volte più veloce della DRAM.

Nuove tecnologie per le DRAM: sfruttano la possibilità di accedere a byte consecutivi più velocemente rispetto a byte causali.

Diverse tecnologie di DRAM

Evoluzione negli anni:

- FPM RAM (Fast page mode)
- EDO RAM (Extended data output)
- SDRAM (Synchronous DRAM)
- DDR3 SDRAM (Double Data Rate SDRAM)
- RDRAM (Direct Rambus DRAM)
- GDDR4 (Graphic Double Data Rate, schede grafiche)
- ...

Stessa struttura interna. Cambia l'interfaccia con il processore.

Double Data Rate Synchronous DRAM

- **Synchronous**: trasmissione sincrona, regolata da un segnale di clock.
Vengono trasmessi pacchetti di dati (locazioni consecutive).
Un nuovo pacchetto a ogni ciclo di clock, ma molti cicli di clock per il primo pacchetto.
- **Double Data Rate**: ad ogni ciclo di clock vengono spediti due pacchetti di dati.

Banda passante e tempo d'accesso

Le nuove DRAM migliorano più la **banda passante** rispetto al **tempo d'accesso**.

- **Banda passante**: quantità di dati consecutivi leggibili nell'unità di tempo.
- **Tempo d'accesso**: tempo necessario per un singola operazione in memoria.

Non sono necessariamente proporzionali.

Significato originale di Random Access Memory:
non si accede a tutti i dati con lo stesso ritardo.

Capacità e connessioni chip di memoria

- Capacità: 4^n , la crescita segue la legge di Moore
- le memorie più capienti sono più costose (per unità di memoria)
- una stessa quantità di memoria può essere distribuita su un numero variabile di locazioni.

Esempio

Un memoria da 1 Gbit può essere realizzata per esempio con:

- 1 G di locazioni di 1 bit
- 512 M di locazioni da 2 bit
- 256 M di locazioni da 4 bit
- 128 M di locazioni da 8 bit.

Distribuzioni diverse portano a diverse quantità di

- linee indirizzo
- linee di dato.

Capacità =

Esempio

Un memoria da 1 Gbit può essere realizzata per esempio con:

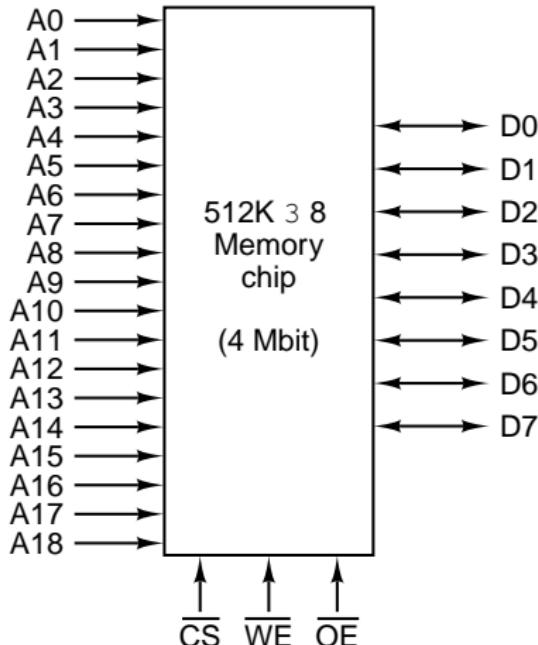
- 1 G di locazioni di 1 bit
- 512 M di locazioni da 2 bit
- 256 M di locazioni da 4 bit
- 128 M di locazioni da 8 bit.

Distribuzioni diverse portano a diverse quantità di

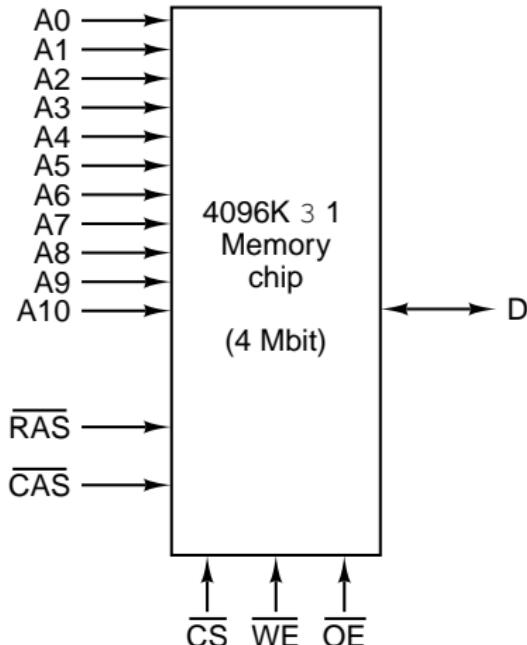
- linee indirizzo
- linee di dato.

Capacità = 2 **linee indirizzo** × **linee dato.**

Esempi

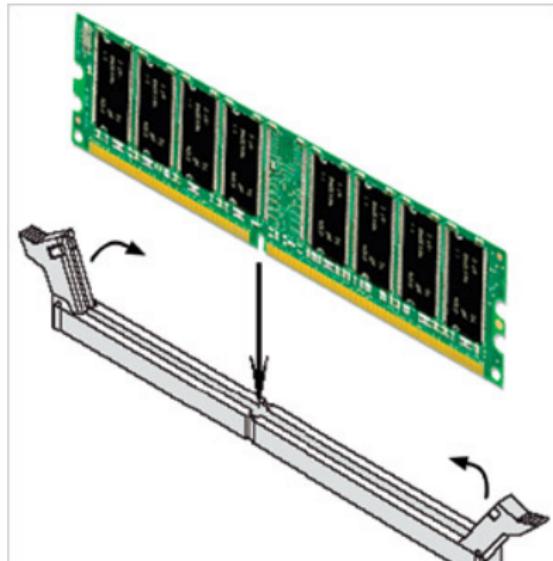
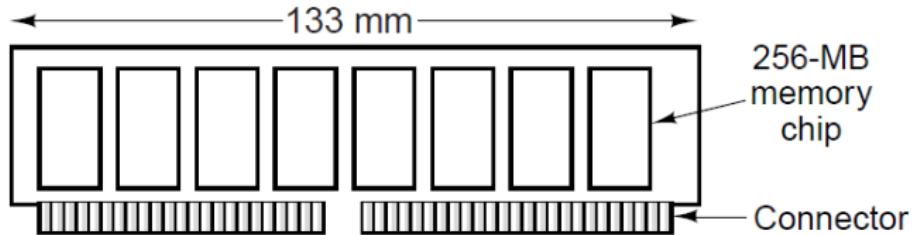


(a)



(b)

Moduli di memoria



Moduli di memoria

Schede di memoria:

- circuito stampato contenente la **RAM dinamica**
- distribuita su più chip
- si innesta in appositi slot (prese) sulla scheda madre
- diversi tipi di connessioni (moduli):
 - **DIMM** Double Inline Memory Module
 - **SO-DIMM** Small Outline DIMM.

Spesso incompatibili: architetture differenti adottano indirizzamenti e tempi d'accesso diversi.

Memorie permanenti

Le RAM perdono i dati se non alimentate.

Memorie permanenti necessarie per:

- calcolatori embedded semplici che eseguono sempre lo stesso codice, non memorizzano dati in modo permanente;
- calcolatori embedded a sostituzione disco magnetico: smartphone, tablet;
- calcolatori: memorizzare i programmi di avvio del calcolatore (bios, sistema operativo, software residente, . . .).

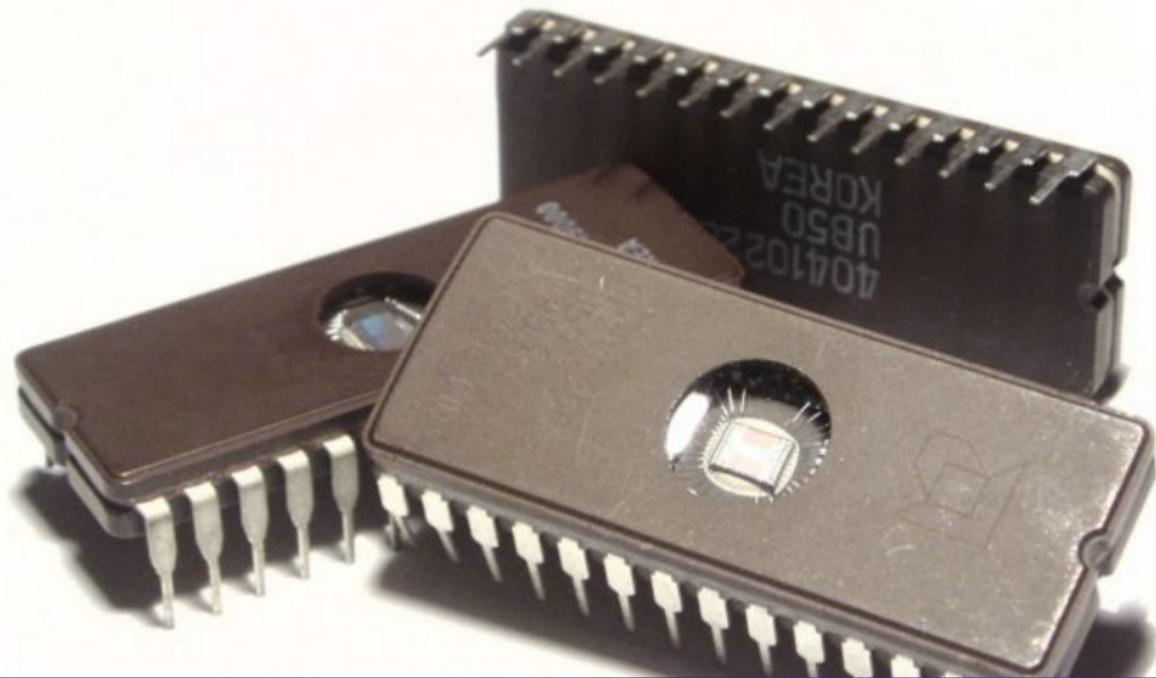
Memorie permanenti

- ROM (Read Only Memory) di sola lettura

Memorie permanenti

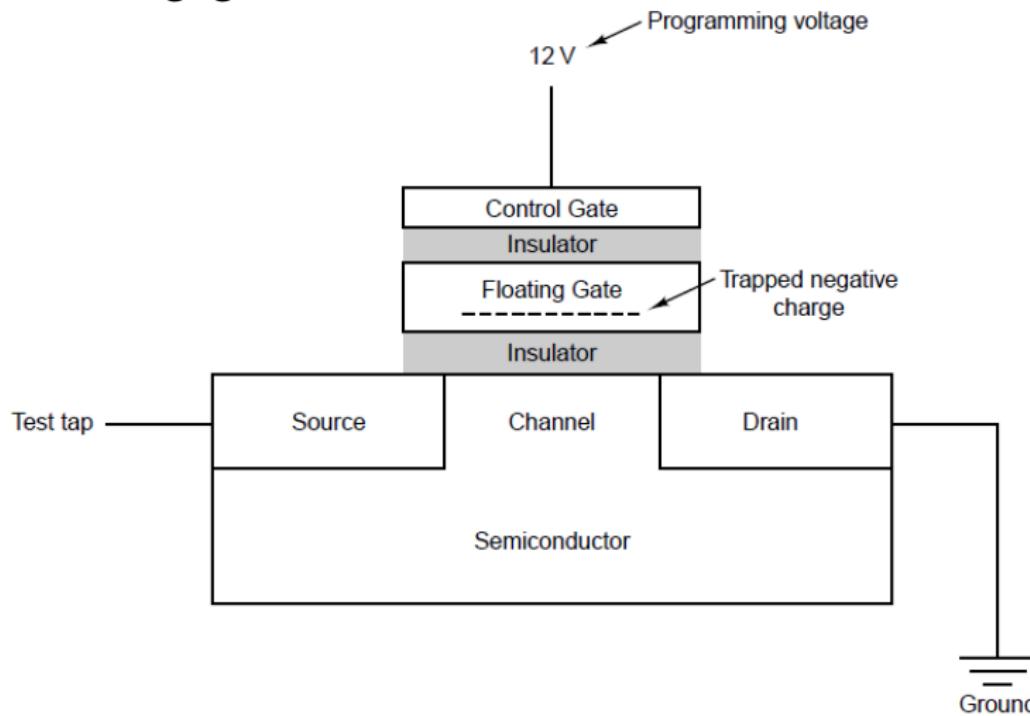
- **ROM** (Read Only Memory) di sola lettura
- **PROM** (Programmable ROM) scrivibili un'unica volta.
Bit: fusibile. Scrittura distruttiva
- **EPROM** (Erasable PROM) cancellabili mediante esposizione a raggi ultravioletti.
Bit: carica elettrica
- **EEPROM** (Electrically EPROM) cancellabili elettricamente (singolo bit).
Bit: carica elettrica
- **Memoria flash**: particolari EEPROM cancellabili a banchi. SSD dischi a stato solido.

Memorie EPROM



Memorie EEPROM, Flash

Floating-gate MOSFET



Classificazione delle memorie

Type	Category	Erasure	Byte alterable	Volatile	Typical use
SRAM	Read/write	Electrical	Yes	Yes	Level 2 cache
DRAM	Read/write	Electrical	Yes	Yes	Main memory (old)
SDRAM	Read/write	Electrical	Yes	Yes	Main memory (new)
ROM	Read-only	Not possible	No	No	Large-volume appliances
PROM	Read-only	Not possible	No	No	Small-volume equipment
EPROM	Read-mostly	UV light	No	No	Device prototyping
EEPROM	Read-mostly	Electrical	Yes	No	Device prototyping
Flash	Read/write	Electrical	No	No	Film for digital camera

Rappresentazione dell'informazione

I calcolatori gestiscono dati di varia natura:
testi, immagini, suoni, filmati.

Nei calcolatori i dati sono rappresentati con sequenze di bit mediante un'opportuna **codifica**.

Presentiamo le codifiche dei dati gestite dall'hardware: **numeri** e **caratteri**.

Argomenti trattati:

- numeri naturali, interi, reali
- operazioni aritmetiche nell'hardware
- caratteri e loro codici di rappresentazione
- codici di correzione degli errori
- organizzazione della memoria.

Codifica: Teoria generale

- insieme dei dati rappresentabili D
- alfabeto $A = \{0, 1\}$: insieme di simboli
- codifica $D \rightarrow A^*$: mappa tra dati e le sequenze di bit.

A^* è l'insieme di tutti i possibili sottoinsiemi che possiamo generare adoperando elementi di A :

$$A^* = \{\emptyset, \{0\}, \{1\}, \{1, 0\}, \dots\}.$$

In una codifica di lunghezza costante,
con n bit si rappresentano sino a 2^n dati diversi.

Esistono codifiche a lunghezza variabile:
i dati possono occupare un diverso numero di bit.

Proprietà di una codifica:

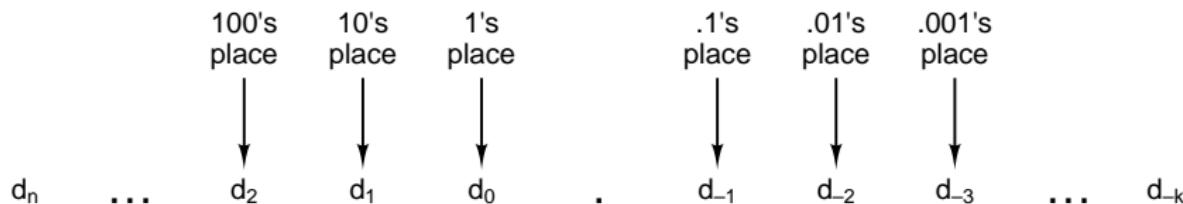
- **compatta**: limitare il numero di bit/byte necessari
- **pratica**: semplificare le operazioni di calcolo
- **accurata**: non perdere informazione, o perdere dati in quantità trascurabile.

Esigenze contrastanti: si cerca un compromesso.

L'aritmetica dei calcolatori

- codificare i numeri naturali, interi, razionali
- eseguire le operazioni.

Notazione posizionale: il peso (significatività) di una cifra dipende dalla sua posizione:



$$\text{Number} = \sum_{i=-k}^n d_i \times 10^i$$

Indipendenza dalla base

La notazione posizionale **non** dipende dalla *base scelta*.

Binary	1	1	1	1	1	0	1	0	0	0	0	1
	$1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$											
	1024	+ 512	+ 256	+ 128	+ 64	+ 0	+ 16	+ 0	+ 0	+ 0	+ 0	+ 1

Octal	3	7	2	1
	$3 \times 8^3 + 7 \times 8^2 + 2 \times 8^1 + 1 \times 8^0$			
	1536	+ 448	+ 16	+ 1

Decimal	2	0	0	1
	$2 \times 10^3 + 0 \times 10^2 + 0 \times 10^1 + 1 \times 10^0$			
	2000	+ 0	+ 0	+ 1

Hexadecimal	7	D	1	.
	$7 \times 16^2 + 13 \times 16^1 + 1 \times 16^0$			
	1792	+ 208	+ 1	

Notazione binaria

Il calcolatore utilizza la base 2:

- un valore basso/alto di tensione rappresenta una cifra
- semplificazione dell'hardware.

Alternativa: **BCD** (Binary Coded Decimal)

- si usa l'usuale notazione decimale
- ogni cifra rappresentata da 4 bit
- vantaggi: nessun cambio di precisione, interfaccia utente più diretta
- svantaggi: complica i circuiti per le operazioni, usa più bit del necessario.

Ambiguità e notazione

“Esistono 10 tipi di persone: quelle che capiscono la notazione binaria e quelle che non la capiscono.”

Se si usano diverse basi allora bisogna specificare la base utilizzata.

Diverse notazioni, in genere un pedice alla fine della sequenza di cifre.

Esempi: 257_{ten}, 257₁₀, 257_{eight}, 257₈.

Alternative: lettere all'inizio della sequenza di cifre:
H328C, **H**ABCD, **O**127.

Conversione di base da 2 a 10

Sommare il peso delle singole cifre.

$$101110110111_{two} =$$

$$2^{11} + 2^9 + 2^8 + 2^7 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0 =$$

$$2048 + 512 + 256 + 128 + 32 + 16 + 4 + 2 + 1 = 2999_{ten}$$

Il metodo applica la definizione di notazione posizionale. Non è computazionalmente efficiente.

Conversione di base da 2 a 10

Metodo più efficiente: **accumulare** il peso delle cifre nella sequenza.

101110110111_{two} :

$$1_{two} = 1_{ten}$$

$$10_{two} = 2_{ten} = 1 * 2 + 0$$

$$101_{two} = 5_{ten} = 2 * 2 + 1$$

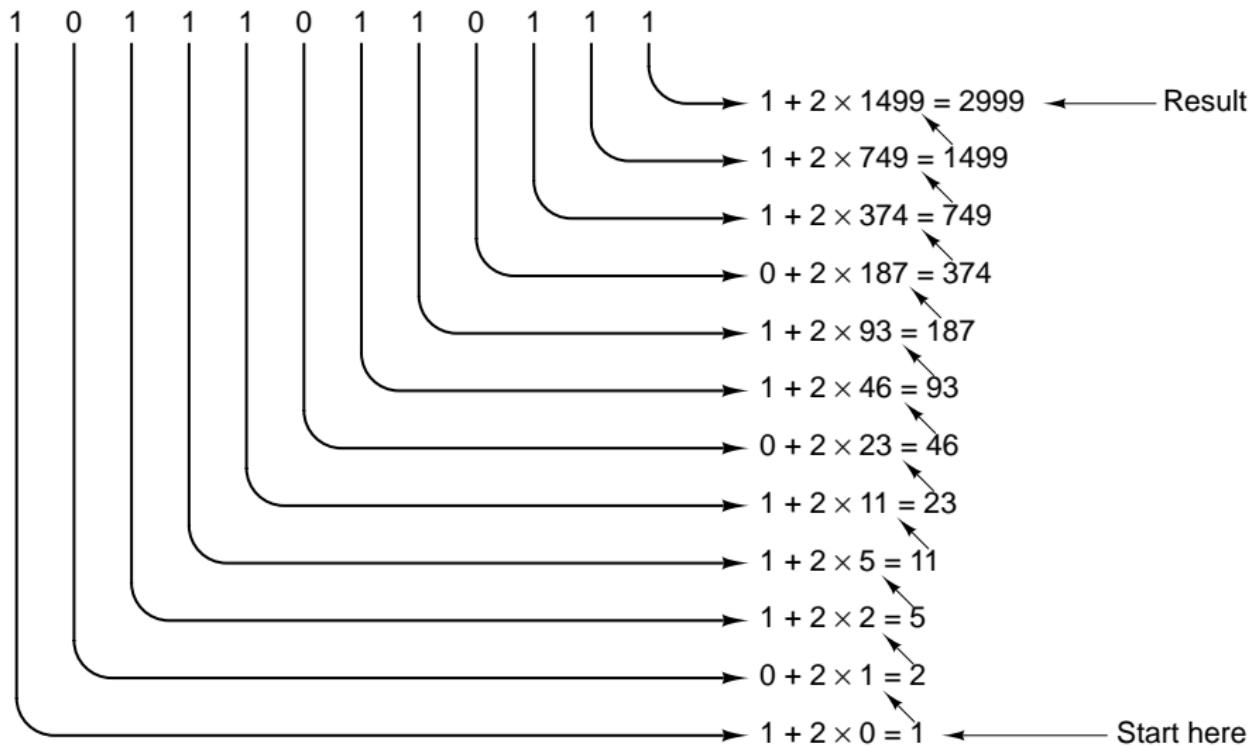
$$1011_{two} = 11_{ten} = 5 * 2 + 1$$

eccetera.

A ogni passo aggiorno il risultato:

“risultato” \leftarrow “risultato” * 2 + “nuova cifra”.

Accumulazione del risultato: esempio



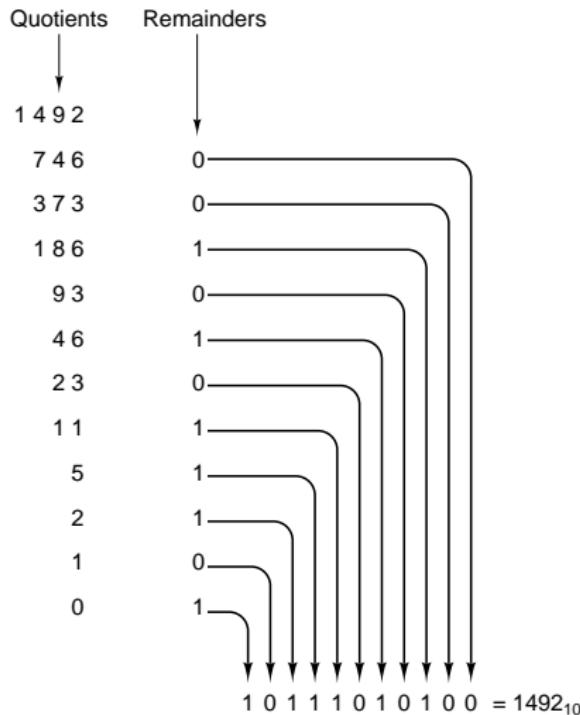
Vale ovviamente per qualsiasi base.

Conversione da base 10 a base 2

- Dividendo un numero decimale per 2 ottengo l'ultima cifra binaria come **resto** (sempre per definizione di notazione posizionale)
- a quel punto posso ripetere lo stesso procedimento sul risultato
- procedo fino a quando il numero da convertire non è nullo.

Il metodo visto converte un numero in base 10 in **qualsiasi base**.

Esempio



Base 8 (ottale) o 16 (esadecimale)

I dati binari possono essere lunghi decine di cifre.

La rappresentazione **ottale** o quella **esadecimale** sono più compatte.

Scegliamo basi che sono potenze di due in quanto **mappano n cifre binare in una cifra nella nuova base**.

Es.: tre in una (ottale); quattro in una (esadecimale).

Se una base è potenza di un'altra allora la conversione di base è immediata: $8 = 2^3$, $16 = 2^4$.

Ricordare: la notazione esadecimale richiede **cifre extra**: A (10), B (11), C (12), D (13), E (14), F (15).

Conversioni

Da base 2 a base 16 (8):

- dividere la sequenza binaria in gruppi di 4 (3), partendo dal bit meno significativo
- trasformare ciascun gruppo in una cifra nella nuova base
- ripetere il procedimento per la parte decimale, partendo dal bit più a sinistra.

Da base 16 (8) a base 2:

- trasformare la cifra in una sequenza binaria di 4 (3) cifre, partendo dal numero meno significativo
- riunire le sequenze così ottenute
- ripetere il procedimento per la parte decimale, partendo dal numero più a sinistra.

Esempi

Example 1

Hexadecimal

1 9 4 8 . B 6

Binary

0 0 0 1 1 0 0 1 0 1 0 0 1 0 0 0 . 1 0 1 1 0 1 1 0 0
1 4 5 1 0 . 5 5 4

Octal

Example 2

Hexadecimal

7 B A 3 . B C 4
0 1 1 1 1 0 1 1 1 0 1 0 0 0 1 1 . 1 0 1 1 1 1 0 0 0 1 0 0
7 5 6 4 3 . 5 7 0 4

Binary

Octal

Un argomento per la correttezza

$$(d_7 d_6 d_5 d_4 d_3 d_2 d_1 d_0)_{\text{two}} =$$

$$= 2^7 d_7 + 2^6 d_6 + 2^5 d_5 + 2^4 d_4 + 2^3 d_3 + 2^2 d_2 + 2^1 d_1 + 2^0 d_0$$

$$= (2d_7 + d_6)2^6 + (4d_5 + 2d_4 + d_3)2^3 + (4d_2 + d_1 + d_0)$$

$$= (2d_7 + d_6)2^{3*2} + (4d_5 + 2d_4 + d_3)2^{3*1} + (4d_2 + d_1 + d_0)$$

$$= (2d_7 + d_6)8^2 + (4d_5 + 2d_4 + d_3)8^1 + (4d_2 + 2d_1 + d_0)$$

$$= ((2d_7 + d_6)(4d_5 + 2d_4 + d_3)(4d_2 + 2d_1 + d_0))_{\text{eight}}$$

L'aritmetica del calcolatore

Tre diverse classi di numeri: **naturali** (unsigned), **interi** (signed), **razionali** (floating point, real).

Rappresentati con un numero fisso di cifre:

- naturali e interi: 16 o 32 o **64** bit
- floating-point: **32** o **64** o **128** bit.

L'insieme dei valori rappresentabili è **limitato**.

Conseguenze:

- esiste un valore massimo e uno minimo rappresentabile
- i valori sono approssimati, a seguito di operazioni oppure inizialmente
- i valori sono memorizzati in più formati.

Overflow

Errore di overflow: il risultato di un'operazione non è rappresentabile.

Esempio: la somma o il prodotto di due numeri vicini al massimo numero rappresentabile.

L'hardware segnala gli eventuali errori di overflow.

Esempio: nella somma di due numeri naturali, l'hardware controlla se i bit più significativi generano un riporto.

Operazioni aritmetiche sui naturali

Sottrazione:

- sottrazioni e prestiti
- hardware: calcola l'**opposto** ed esegue un'addizione. Il problema quindi è spostato sulla scelta di una codifica efficiente per i numeri negativi.

Moltiplicazione:

- serie di somme
- hardware: stessa idea ma parallelizzata in più somme contemporaneamente, meno cicli di somme (Dadda — Wallace multipliers).

Operazioni aritmetiche

Divisione:

- serie di sottrazioni
- hardware: serie di sottrazioni, oppure calcolo dell'inverso e prodotto (il problema è spostato sulla scelta di una codifica efficiente per l'inverso di un numero).

Codifiche di interi

Hardware: **quattro** possibili alternative.

- **segno e valore assoluto**

$$9_{\text{dieci}} = 0\ 0001001$$

$$-9_{\text{dieci}} = 1\ 0001001$$

- **complemento a 1**

$$9_{\text{dieci}} = 0\ 0001001$$

$$-9_{\text{dieci}} = 1\ 1110110$$

- **complemento a 2**

$$9_{\text{dieci}} = 0\ 0001001$$

$$-9_{\text{dieci}} = 1\ 1110111$$

- **eccesso 128**

$$9_{\text{dieci}} = 1\ 0001001 \quad (9 + 128 = 137)$$

$$-9_{\text{dieci}} = 0\ 1110111 \quad (-9 + 128 = 119).$$

Complemento a 2

È la più usata.

- **Definizione:** il numero negativo $-i$ si ottiene complementando i e poi aggiungendo 1
- **Definizione equivalente:** con n cifre binarie, il numero negativo $-i$ viene rappresentato da $2^n - i$. I due valori sono congruenti modulo 2^n :
 $-i \equiv_{2^n} (2^n - i)$
- **Motivazioni:** semplifica l'ALU (Arithmetic Logic Unit).

Somma algebrica in complemento a 2

- eseguo la somma come se i numeri fossero **naturali** in notazione binaria
- perché funziona? Proprietà dell'aritmetica modulo $m = 2^n$. Tre casi possibili, con $i \geq 0, j \geq 0$:
 - i) $i + j \equiv_{2^n} i + j$
 - ii) $i - j \equiv_{2^n} i + (2^n - j) \equiv_{2^n} (2^n + i - j)$
 - iii) $-i - j \equiv_{2^n} (2^n - i) + (2^n - j) \equiv_{2^n} (2^n + 2^n - i - j) \equiv_{2^n} (2^n - i - j)$
- **diverso** il controllo dell'**overflow**: c'è overflow **solo** se sommo termini con lo stesso segno ma il segno del risultato **non coincide**.

Operazioni in complemento a 2

- **Opposto**: complemento a 2 del numero
- **Complemento a 2 del complemento a 2 di n** : n stesso
- **Sottrazione**: calcolo l'opposto del secondo termine e poi sommo
- **Prodotto e divisione**: simili, ma non identici ai rispettivi algoritmi sui naturali
- **Estensione del numero di cifre**: aggiungo cifre uguali alla cifra segno
- **Conversione in base dieci**: se il numero è negativo prima calcolo l'opposto, poi adopero l'algoritmo già visto, infine ricordo il segno.

Numeri frazionari

Il peso di ogni cifra dopo la virgola è elevato a una potenza negativa della base b , dunque la somma dei pesi è **sempre** minore di uno. Ciò permette di tenere il calcolo della parte frazionaria separato da quello della parte intera.

Es. ($b = 2$): $1,011 = 1 + 1/4 + 1/8 = 1 + 3/8$.

Non tutti i valori sono rappresentabili nella nuova base con un numero finito di cifre. Es.: $0,4_{10}$ è periodico se rappresentato in base due.

Vale anche il viceversa: un valore periodico in qualche base può **non** esserlo se rappresentato in base diversa. Es.: $0,\overline{3}_{10}$ non è periodico in base 3.

Conversione di base della parte frazionaria

Si convertono quindi parte intera e parte frazionaria **separatamente**.

Conversione della parte frazionaria:

- da base due (o base b) a base dieci: applico direttamente la notazione posizionale
- da base dieci a base due (o base b) procedo come segue: se non è nulla, moltiplico la parte frazionaria per 2 (o per b); ora,
 - la parte intera del risultato è la nuova cifra frazionaria nella nuova base
 - la nuova parte frazionaria è da rimoltiplicare.

Nota: il procedimento può **non terminare**.

Operazioni su numeri frazionari

Gli algoritmi sono quelli usuali:

- somma e sottrazione: sommo (sottraggo) cifre associate allo stesso peso (allineamento della virgola)
- moltiplicazione e divisione: opero con i numeri senza virgola e poi valuto dove posizionare la virgola nel risultato.

Notazione floating-point

Per i numeri frazionari si può usare la tradizionale notazione scientifica (**floating-point**) mantissa più esponente.

Il numero X è rappresentato da una **coppia** (m, e) :

- $X = m \times 10^e$
- **mantissa** m : numero frazionario **normalizzato** (es.: $-1 < m < 1$)
- **esponente** e : numero **intero**.

Vantaggi: si rappresentano numeri grandi o piccoli in modo efficiente.

Floating-point nel calcolatore

Un numero binario X viene rappresentato da una sequenza di bit divisa in due parti:

- la mantissa m . La virgola è in posizione fissa e non viene rappresentata,
- una parte per l'esponente e (numero intero)

$$X = m \times 2^e$$

L'esponente indica di quante posizioni deve essere spostata la virgola nella mantissa.

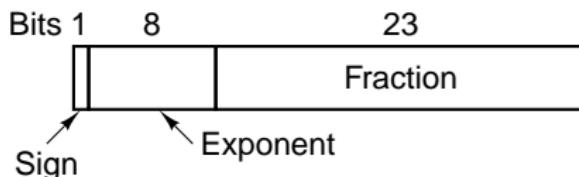
Nota bene: in questa notazione X inizia **sempre** con la cifra 1.

Floating-point nel calcolatore

Lunghezza in bit fissa per m ed e .

Al solito, numero **finito** di valori rappresentabili.

Esempio:



(a)

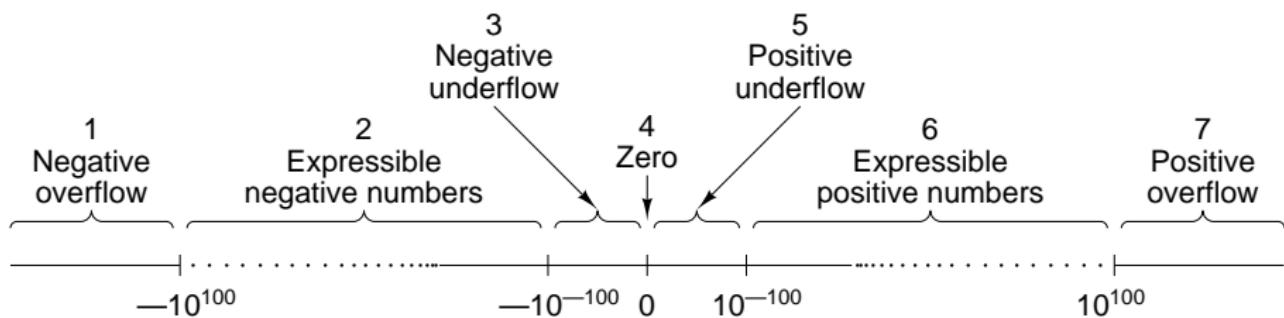


(b)

Floating-point nel calcolatore

A seconda di come partiziono la sequenza:

- più bit per $m \Rightarrow$ intervalli più densi
- più bit per $e \Rightarrow$ intervalli più ampi.



Floating-point nel calcolatore

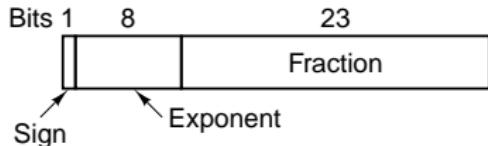
Difetti:

- **precisione**: ogni rappresentazione e ogni operazione può essere affetta da errore
- **overflow**: risultati non rappresentabili perché troppo grandi
- **underflow**: risultati non rappresentabili perché troppo vicini a 0 sono approssimati con 0.

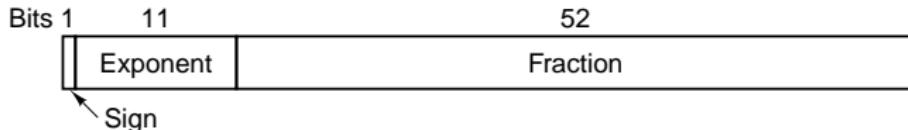
Notazione IEEE standard

- Fino agli anni 80: processori diversi usavano notazioni diverse, problemi nello scambio dati
- produttori affidano a William Kahan la definizione di uno standard
- notazione **IEEE 754** standard
- gestazione molto lunga
- ora in uso su tutti i calcolatori.

Notazione IEEE standard



(a)



(b)

segno 0: positivo; 1: negativo

esponente eccesso 127 (1023)

mantissa notazione normalizzata: la mantissa inizia sempre con 1, che quindi non occorre rappresentare.

Valori speciali

L'esponente minimo o massimo codifica valori speciali.

- 00000000: inizio隐式 della mantissa: 0; esponente: -126 (numeri denormalizzati)
- 11111111, con mantissa 00...00: infinito (gestisce l'overflow)
- 11111111, con mantissa diversa da 00...00: not a number (NaN), risultato non rappresentabile (es.: 0/0).

Normalized	\pm	$0 < \text{Exp} < \text{Max}$	Any bit pattern
Denormalized	\pm	0	Any nonzero bit pattern
Zero	\pm	0	0
Infinity	\pm	1 1 1...1	0
Not a number	\pm	1 1 1...1	Any nonzero bit pattern

↑ Sign bit

Laboratorio di architettura degli elaboratori

CIRCUITI COMBINATORI Lezione 3

- CONTATTI
- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

Esercizio 3.1.1

Progettare un **decoder** a 2 ingressi dotato di un segnale aggiuntivo di Enable. Se il segnale Enable vale 0 il circuito restituisce 0 in uscita. Realizzare il circuito come modulo.

Esercizio 3.1.2

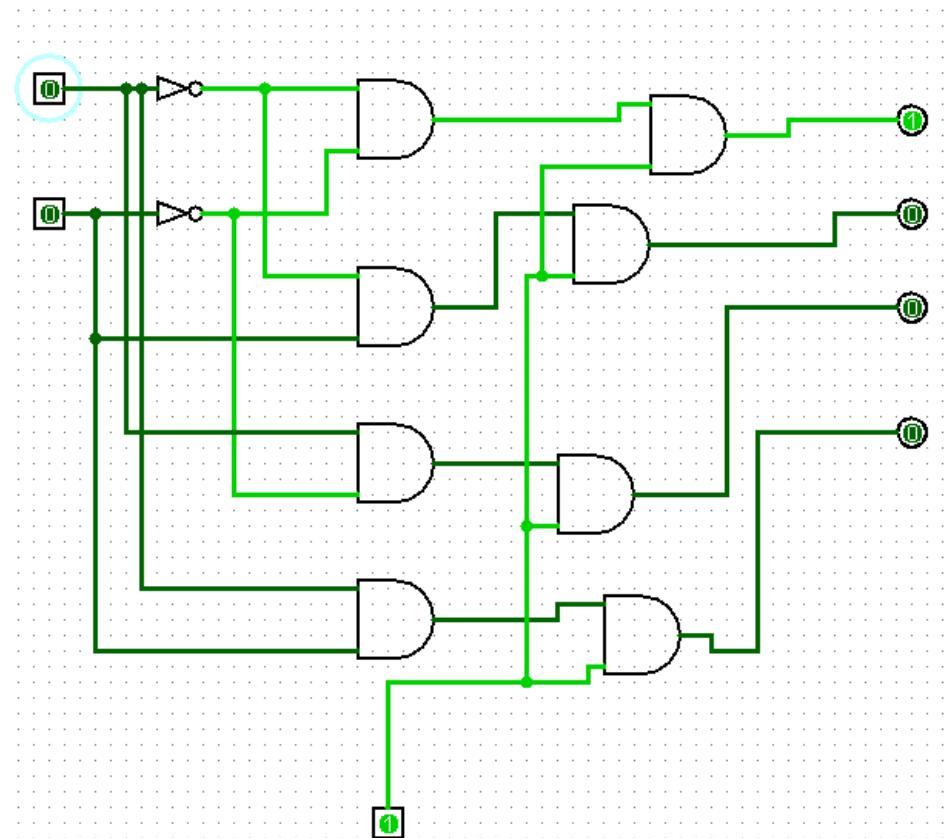
Utilizzare il modulo del punto precedente per realizzare un decoder a 3 ingressi

Esercizio 3.1.3

Utilizzare il modulo del punto 3.1.1 per realizzare un decoder a 4 ingressi

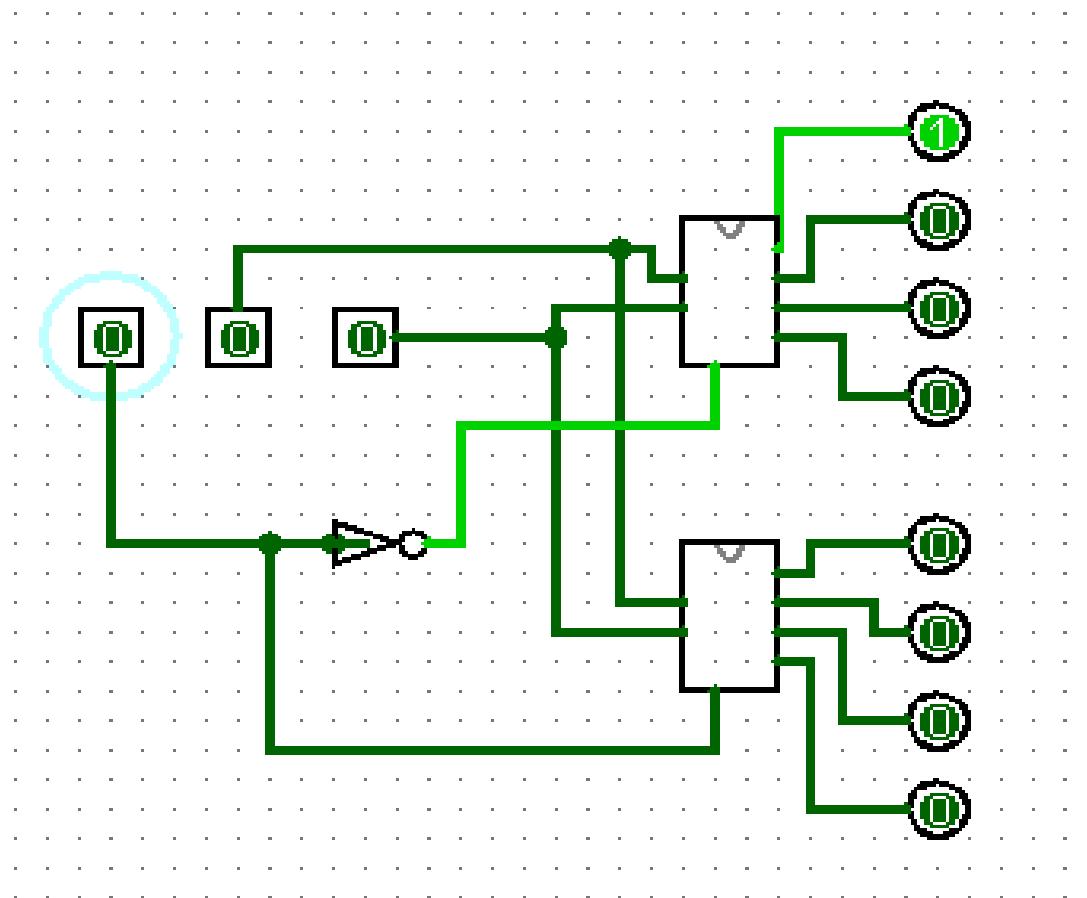
Esercizio 3.1.1

Progettare un **decoder** a 2 ingressi dotato di un segnale aggiuntivo di Enable, se il segnale Enable vale 0 tutte le uscite valgono 0, se Enable vale 1 si comporta come un circuito decoder.
Realizzare il circuito come modulo.



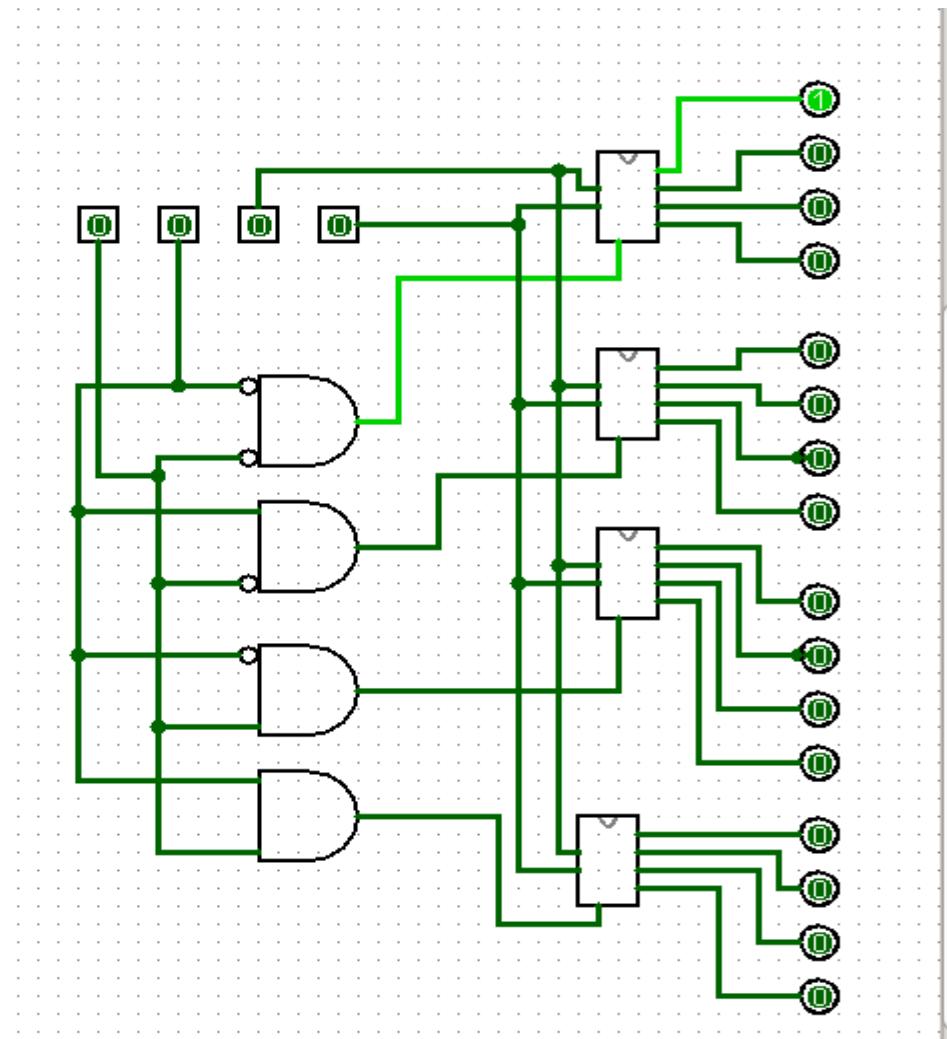
Esercizio 3.1.2

Utilizzare il modulo del punto precedente per realizzare un decoder a 3 ingressi



Esercizio 3.1.3

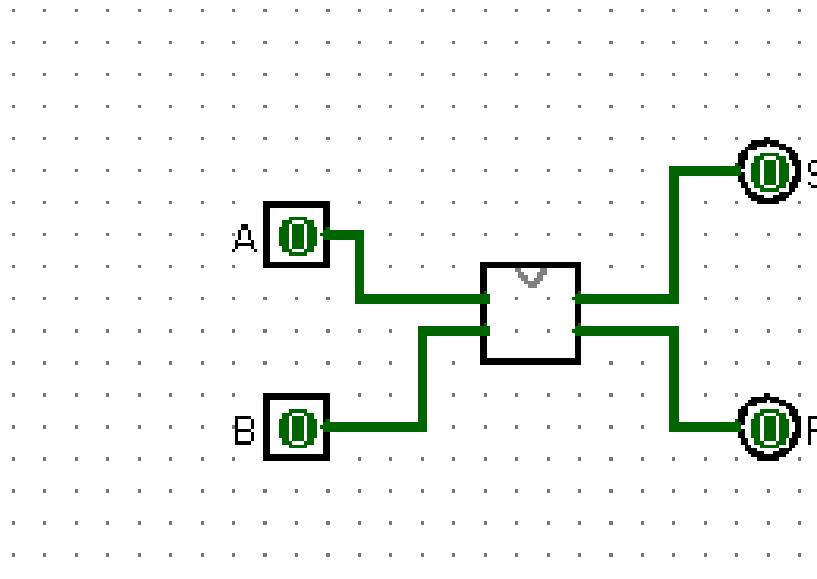
Utilizzare il modulo del punto precedente per realizzare un decoder a 4 ingressi



Esercizio 3.2.1

Progettare un **half-adder**, ossia un circuito combinatorio che somma due bit e genera il bit risultato ed un riporto.

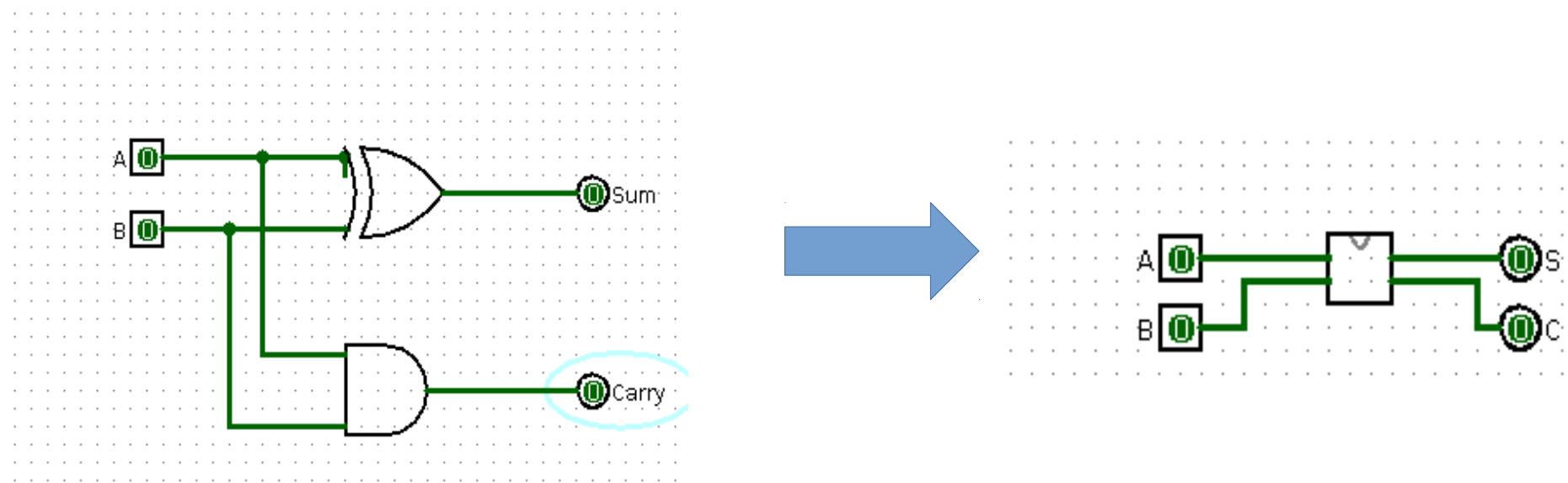
L'half-adder va realizzato come modulo (sottocircuito) Logisim.



Esercizio 3.2.1

Progettare un **half-adder**, ossia un circuito combinatorio che somma due bit e genera il bit risultato ed un riporto.

L'half-adder va realizzato come modulo (sottocircuito) Logisim.



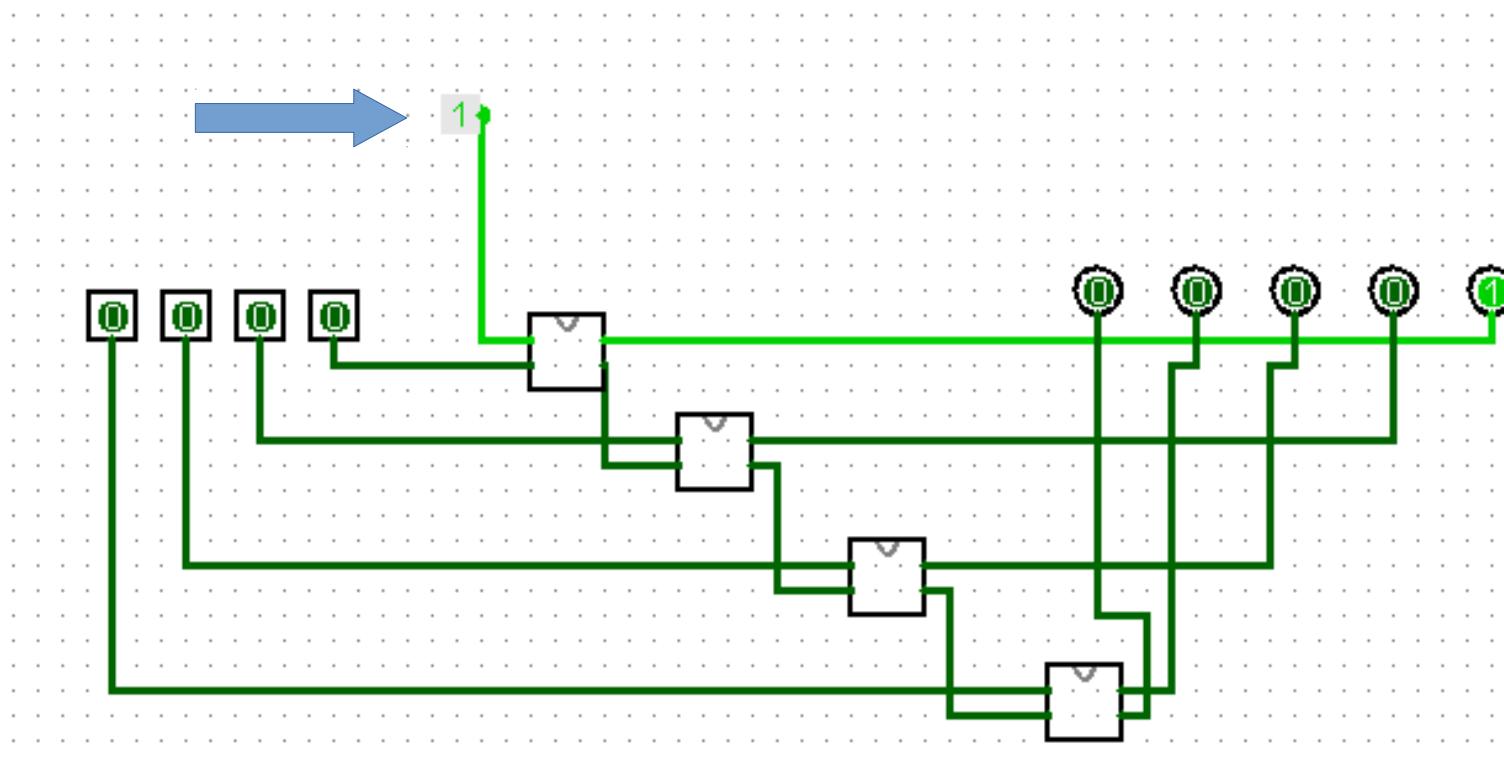
Esercizio 3.2.2

Utilizzando 4 moduli half-adder, costruire un circuito che ricevuto in ingresso un numero binario di 4 cifre, restituisca in uscita il numero binario successivo.

Implementare il circuito come modulo.

Esercizio 3.2.2

Utilizzando 4 moduli half-adder, costruire un circuito che ricevuto in ingresso un numero binario di 4 cifre, restituisca in uscita il numero binario successivo.



Esercizio 3.3.1

Utilizzando due moduli half-adder, progettare un full-adder, ossia un circuito combinatorio che somma due bit ed un riporto e genera il bit risultato ed un nuovo riporto.

Realizzare i full-adder come modulo Logisim.

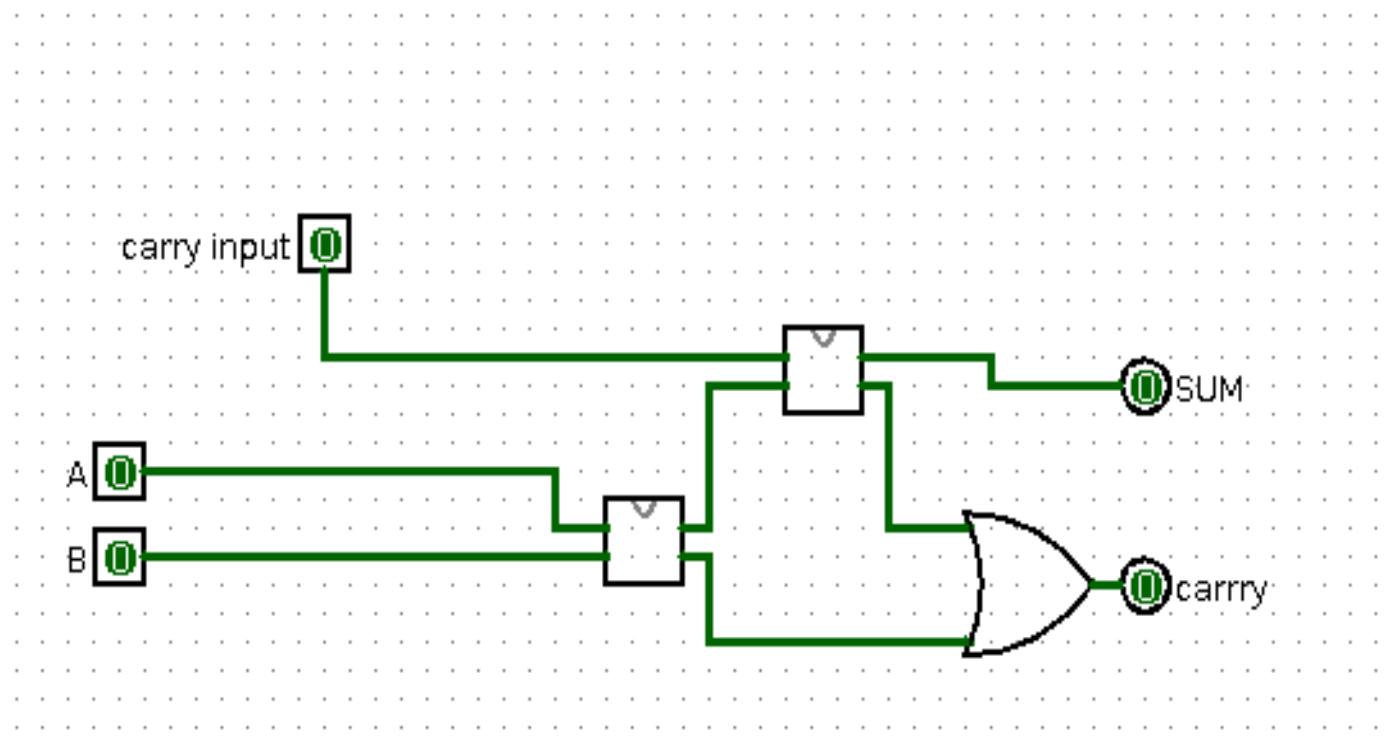
Esercizio 3.3.2

Progettare un circuito che calcoli la somma di due numeri binari di 4 bit ciascuno.

Esercizio 3.3.1

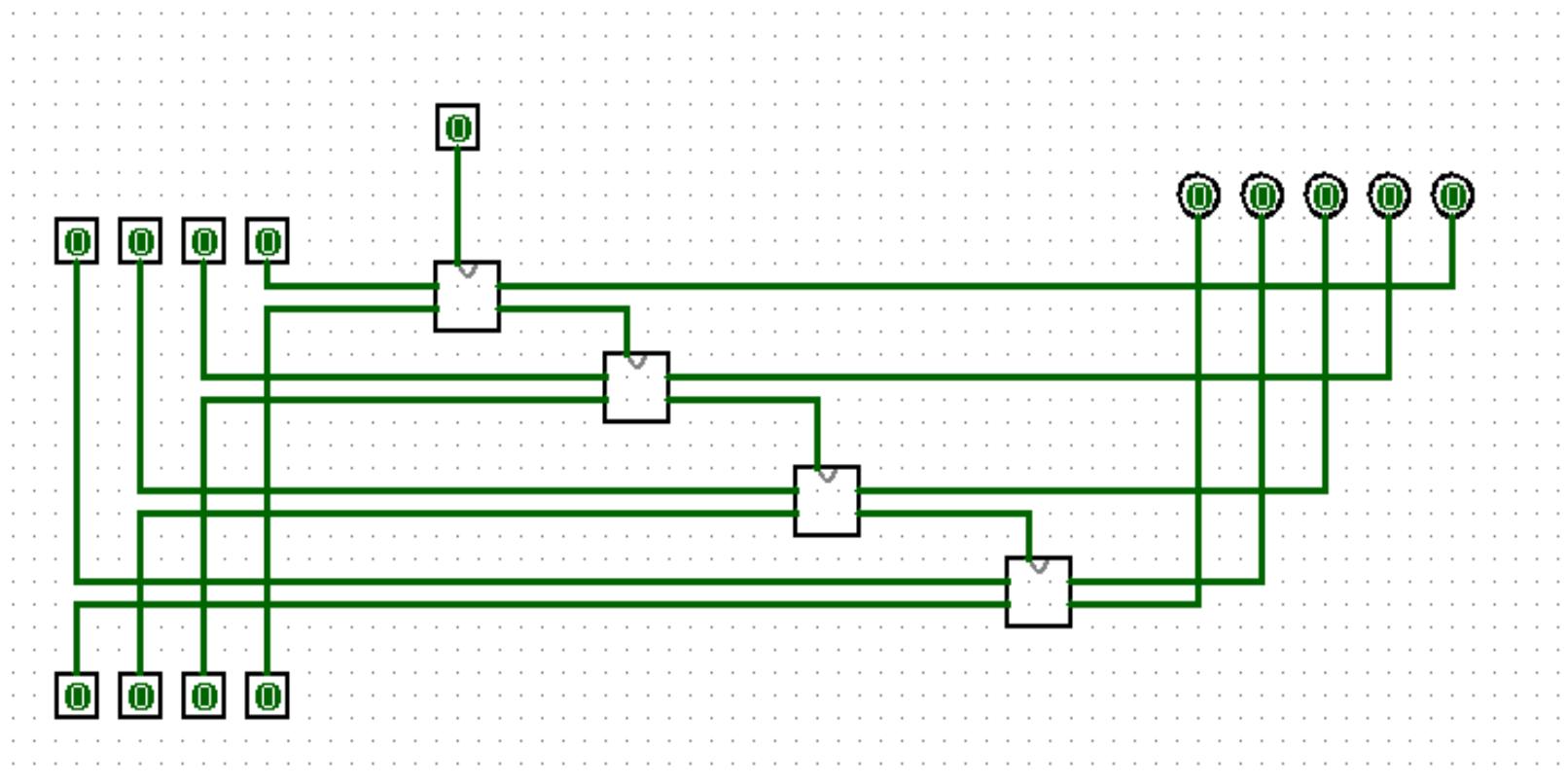
Utilizzando due moduli half-adder, progettare un full-adder, ossia un circuito combinatorio che somma due bit ed un riporto e genera il bit risultato ed un nuovo riporto.

Realizzare i full-adder come modulo Logisim.



Esercizio 3.3.2

Progettare un circuito che calcoli la somma di due numeri binari di 4 bit ciascuno.



Esercizio 3.1.1

Progettare un **decoder** a 2 ingressi dotato di un segnale aggiuntivo di Enable. Se il segnale Enable vale 0 il circuito restituisce 0 in uscita. Realizzare il circuito come modulo.

Esercizio 3.1.2

Utilizzare il modulo del punto precedente per realizzare un decoder a 3 ingressi

Esercizio 3.1.3

Utilizzare il modulo del punto 3.1.1 per realizzare un decoder a 4 ingressi

Esercizio 3.2.1

Progettare un **half-adder**, ossia un circuito combinatorio che somma due bit e genera il bit risultato ed un riporto. L'half-adder va realizzato come modulo (sottocircuito) Logisim.

Esercizio 3.2.2

Utilizzando 4 moduli half-adder, costruire un circuito che ricevuto in ingresso un numero binario di 4 cifre, restituisca in uscita il numero binario successivo.

Esercizio 3.3.1

Utilizzando due moduli half-adder, progettare un full-adder, ossia un circuito combinatorio che somma due bit ed un riporto e genera il bit risultato ed un nuovo riporto. Realizzare i full-adder come modulo Logisim.

Esercizio 3.3.2

Progettare un circuito che calcoli la somma di due numeri binari di 4 bit ciascuno

Esercizi

- Contatore “up/down” a 2 bit:
 - 2 ingressi: x abilitazione al conteggio, ud ordine di conteggio
 - 2 uscite: numero binario.
- Circuito sequenziale per riconoscere una stringa (1100).
- Circuito sequenziale con:
 - 1 ingresso: numeri codificati come gruppi di 3 bit
 - 2 uscite: assumono il valore 00 in corrispondenza del primo e del secondo bit di ingresso; poi il numero (in binario) degli 1 ricevuti in ingresso.

Esercizi su circuiti sequenziali

- Progettare un circuito sequenziale che è in grado di riconoscere tutte e sole le sequenze in una stringa binaria contenenti ripetizioni non vuote della sottostringa 01. Il circuito produce in uscita 1 non appena il riconoscimento di una di queste sequenze termina; 0 altrimenti.
Es.: ... 110101001010000111 ...
- Adoperando una macchina di Moore, progettare un circuito sequenziale che riconosce ogni ricorrenza disgiunta di caratteri doppi appartenenti all'alfabeto $A = \{a, b, c\}$. Il circuito produce 1 non appena il riconoscimento di due caratteri identici termina; 0 altrimenti.

Codifica dei caratteri

Caratteri: simboli contenuti in documenti testuali

- cifre, lettere, simboli di punteggiatura
- **simboli speciali:** '@', '#', '\$', '%', '&', ')', '(',
- **caratteri speciali:** ritorno a capo, tabulazione, *escape* (ESC), spostamento del cursore,
Informazioni di controllo non visibili, definiscono il formato, contengono informazione non stampabile.

Codici per caratteri

I caratteri vengono rappresentati mediante codici: si associa a ogni carattere una sequenza di bit (numero binario).

Associazione arbitraria, con alcune regole:

- cifre consecutive mappate su codici consecutivi
 $'N' = '0' + N,$
- lettere: ordine lessicografico
 $'C' = 'A' + \text{posizione lettera nell'alfabeto} - 1.$

Principali codici

- ASCII (standard 8 bit ed esteso),
- MS DOS,
- MAC OS Roman,
- UNICODE,
- UTF-8, UTF-7, UTF-16,
- EBCDIC,
- Morse.

Codice ASCII

American Standard Code for Information Interchange

Prima codifica a larga diffusione (anni '60), 7 bit per carattere, max 128 caratteri codificabili:

- codici da 0 a 31 dedicati al controllo del testo (carriage return, line feed, backspace, cancel, escape, ...) e del flusso (start of heading, end of transmission, ...)
- codici da 32 a 127 dedicati a 95 caratteri stampabili.

Codice ASCII

Hex	Name	Meaning	Hex	Name	Meaning
0	NUL	Null	10	DLE	Data Link Escape
1	SOH	Start Of Heading	11	DC1	Device Control 1
2	STX	Start Of TeXt	12	DC2	Device Control 2
3	ETX	End Of TeXt	13	DC3	Device Control 3
4	EOT	End Of Transmission	14	DC4	Device Control 4
5	ENQ	Enquiry	15	NAK	Negative AcKnowledgement
6	ACK	ACKnowledgement	16	SYN	SYNchronous idle
7	BEL	BELI	17	ETB	End of Transmission Block
8	BS	BackSpace	18	CAN	CANcel
9	HT	Horizontal Tab	19	EM	End of Medium
A	LF	Line Feed	1A	SUB	SUBstitute
B	VT	Vertical Tab	1B	ESC	ESCAPE
C	FF	Form Feed	1C	FS	File Separator
D	CR	Carriage Return	1D	GS	Group Separator
E	SO	Shift Out	1E	RS	Record Separator
F	SI	Shift In	1F	US	Unit Separator

Codice ASCII

Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char	Hex	Char
20	(Space)	30	0	40	@	50	P	60	'	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	\$	34	4	44	D	54	T	64	d	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(38	8	48	H	58	X	68	h	78	x
29)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	,	4B	K	5B	[6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	
2D	-	3D	=	4D	M	5D]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Codice ASCII

Problemi

- L'insieme di caratteri di controllo era pensato per le “telescriventi” (typewriter: tty), postazioni remote nei mainframe degli anni '60. Ancora supportate in UNIX e Linux.
- Ritorno a capo = carriage return + line-feed.
Sistemi operativi diversi gestiscono in diverso modo il ritorno a capo.
- Pochi caratteri rappresentabili: non ci sono i caratteri non appartenenti all'alfabeto occidentale internazionale: lettere accentate, segni diacritici.

Estensioni ASCII

Unità d'informazione il **byte** (sequenze di 8 bit).

Naturale usare 8 bit per codificare un carattere.

Diverse estensioni del codice ASCII standard: si conservano le prime 128 codifiche, si aggiungono 128 nuovi caratteri

- **ANSI** (UNIX),
- MS-DOS,
- MAC OS Roman.
- Lingue diverse usano caratteri diversi. Di qui la necessità di diversificare.

Standard 8859

Si mette ordine tra le diverse estensioni ASCII definendo un unico standard: **code page** (pagina di codice)

- IS 8859-1: ANSI, Latin 1, West Europe, IS 646;
- IS 8859-2: Latin 2, East Europe, lingue slave;
- IS 8859-3: Latin 3, South Europe, turco, esperanto, ... ;
- IS 8859-5: Cyrillic;
- IS 8859-6: Arabic;
- ...

ASCII esteso

Problemi:

- il software **deve sapere** su che pagina opera
- non si possono mescolare le lingue
- difficile gestire lingue **dinamiche** e/o con moltissimi caratteri: cinese e giapponese

Problemi superabili con un ampio insieme di caratteri codificabili.

UNICODE

Nato dall'accordo tra diverse aziende (IS 10646)

- due (o più) byte per carattere: $2^{16} = 65.536$ caratteri
- simboli matematici, musicali, grafici
- **code point**: il codice di un carattere. Per semplicità di traduzione tra code page spesso esistono più code point per lo stesso carattere
- code point lasciati vuoti per future estensioni.

La definizione non è stata ancora completata.

Problemi con UNICODE

65.536 code point non sono sufficienti: più di 200.000 simboli nel mondo!

- ulteriore estensione: Universal Character Set (UCS)
- in continua evoluzione, ogni anno nuovi gruppi di caratteri
- al 2015: 120.000 code point.

Codifiche a lunghezza variabile

Nella versione completa (UTF-32) occorrono 4 byte per carattere. Ciò rende i file spesso inutilmente grandi.

Codici UNICODE con lunghezze variabili: caratteri diversi usano un diverso numero di byte.

- **UTF-8** (UCS Transformation Format 8 bit): da 1 a 4 byte per carattere
- **UTF-16**: 2 o 4 byte per carattere.

UTF-8

È la codifica più diffusa: si possono rappresentare tutti i caratteri UCS e si generano file compatti

- 0-127 (un byte): caratteri ASCII standard
- $128 \sim 2^{11}$ (due byte): tutte le varie estensioni ASCII
- 3-4 byte: ideogrammi cinesi, altre lingue
- UTF-16: rappresentazione più compatta degli ideogrammi cinesi.

UTF-8

Bits	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6
7	0ddddddd					
11	110dddddd	10ddddddd				
16	1110dddd	10ddddddd	10ddddddd			
21	11110ddd	10ddddddd	10ddddddd	10ddddddd		
26	111110dd	10ddddddd	10ddddddd	10ddddddd	10ddddddd	
31	1111110x	10ddddddd	10ddddddd	10ddddddd	10ddddddd	10ddddddd

Conclusioni

La standardizzazione dei codici carattere è limitata

- è comune trovare documenti in diversi formati: Latin-1, UTF-8, UNICODE, MS-DOS, MAC-OS Roman
- problemi nello scambio di dati
- mail, documenti HTML devono specificare la codifica in cui sono scritti: necessità di un *header*.

Codici di correzione degli errori

Nella trasmissione e memorizzazione dei dati si verificano degli **errori**:

- disturbi sulla linea (trasmissione),
- imperfezioni del supporto (memoria disco),
- radioattività, fotoattività (DRAM).

Necessità di meccanismi di **protezione dagli errori**.
Nonostante ciò, alcuni errori dell'hardware sono inevitabili.

Codici di correzione degli errori

Codici di correzione: meccanismi per controllare ed eventualmente correggere errori nei dati.

Idea base: **ridondanza** dell'informazione

- si aggiunge ai dati dell'informazione di controllo
- si trasmettono (o memorizzano) più dati di quelli strettamente necessari
- chi riceve determina la presenza di errori sfruttando l'informazione ridondante.

Esempio dal parlato: sillabazione

Nella comunicazione vocale non tutti i suoni vengono compresi correttamente.

Sillabare: per comunicare un carattere si comunica un'intera parola nota al destinatario.

- “D come Domodossola!”
- Questo permette di correggere ‘T’, ‘B’ in ‘D’.

Linguaggio parlato

Nel linguaggio parlato c'è più informazione di quella strettamente necessaria. Il contesto in cui appare una parola nota all'ascoltatore infatti permette di correggere eventuali errori nella comprensione:

- “*oddimo*” viene subito corretto con “*ottimo*”
- “*testo*” può essere confuso con “*desto*” ma “*quali sono i libri di desto*” viene corretto in “*quali sono i libri di testo*”.

Semplici esempi di codici

Trasmetto un testo ripetendo ogni carattere due volte: casa ⇒ ccaassaa

Posso in generale **rilevare** l'errore:
ccaasraa ⇒ casa? cara?

Trasmetto un testo ripetendo ogni carattere tre volte.

Posso in generale **correggere** l'errore:

cccaaasrsaaa ⇒ casa

cccaaasrraaa ⇒ cara

cccaaasrvaaa ⇒ casa? cara? cava?

NON ci sono codici a prova d'errore.

Codici di parità

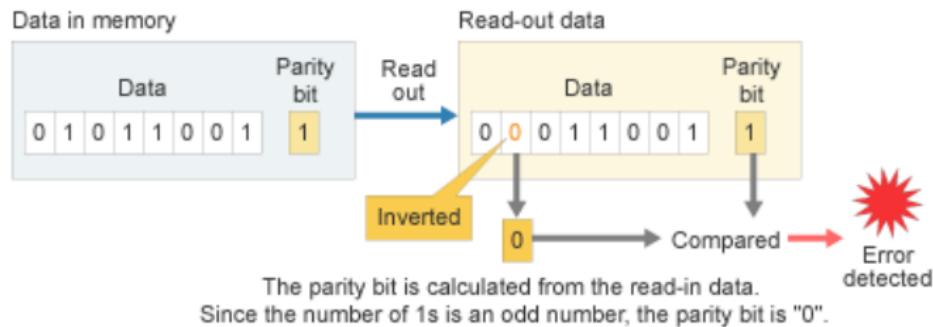
I codici precedenti sono troppo costosi: introducono troppa informazione ridondante.

Bit di parità: codice con un singolo bit ridondante, molto usato nella pratica.

- I dati sono divisi in pacchetti (byte, parole).
- Ad ogni pacchetto viene aggiunto un bit di controllo
 - in modo tale che il numero totale di bit 1 sia **pari**.
- Chi riceve il dato
 - rileva la presenza di errori che modificano un (numero dispari di) bit in una parola,
 - non può rilevare errori che modificano un numero pari di bit in una parola,
 - non può correggere eventuali errori.

Esempio

Original Data	Even Parity	Odd Parity
0 0 0 0 0 0 0 0	0	1
0 1 0 1 1 0 1 1	1	0
0 1 0 1 0 1 0 1	0	1
1 1 1 1 1 1 1 1	0	1
1 0 0 0 0 0 0 0	1	0
0 1 0 0 1 0 0 1	1	0



Questo esempio usa parità dispari.

Codice di correzione di Hamming

Il trasmettitore:

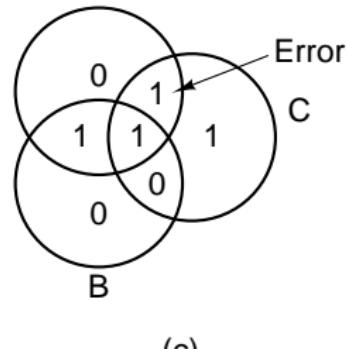
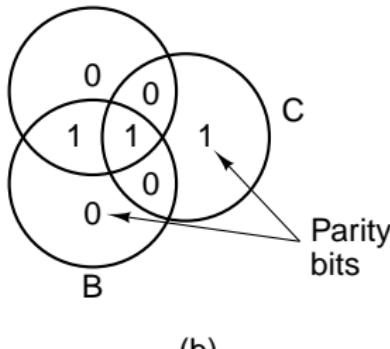
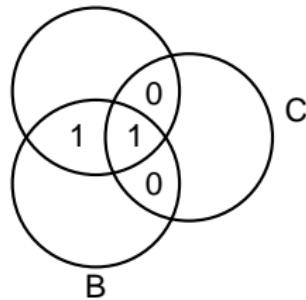
- divide la sequenza binaria di bit informativi in sottoinsiemi
- ogni bit è univocamente individuato dai sottoinsiemi a cui appartiene
- introduce un bit di parità per ogni sottoinsieme.

Il ricevitore:

- valuta la parità su ogni sottoinsieme
- in caso di errore elenca i bit di parità errati
- se possibile corregge l'errore sul bit che appartiene a tutti e soli i sottoinsiemi *errati*.

Codice di correzione di Hamming

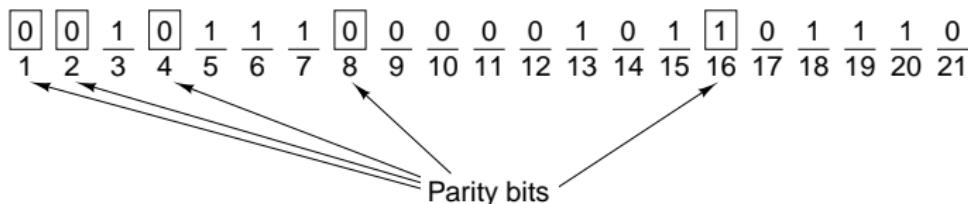
Implementazione a per 4 bit dati e 3 bit controllo.



Codice di Hamming a N bit

$$N = 2^k - 1 \text{ con } K = 2, 3, \dots$$

Memory word 1111000010101110



- etichetto in base 2 le posizioni dei bit **a partire dall'etichetta 1**
- uso come bit di parità quelli la cui etichetta (posizione) è una potenza di 2: $1_2, 10_2, 100_2, \dots$
- ogni bit di parità controlla i bit la cui etichetta contiene la cifra 1 nella stessa posizione
Es. ($N = 7$): il bit in posizione 010 controlla i bit nelle posizioni 011, 110 e 111.

Costo di un codice

Introdurre informazione ridondante costa: si utilizza più banda (trasmissiva, di memoria).

Costo:

$$\frac{\text{dati ridondanti}}{\text{dati utili}}.$$

Costo esempi precedenti:

- ripetizione doppia del carattere: costo 1, 100% dati aggiuntivi
- ripetizione tripla del carattere: costo 2, 200% dati aggiuntivi
- bit di parità: $\frac{1}{\text{dim. pacchetto} - 1}$.
Es. (pacchetti di 9 bit): 12,5% dati aggiuntivi.

Affidabilità di un codice

Nessun codice di rilevazione (correzione) errore garantisce un'**affidabilità** assoluta.

Se quasi tutti i bit trasmessi sono errati nessun codice funziona.

Una buona codifica in pratica rende trascurabile (ma non nulla!) la probabilità di non rilevare un errore presente nei dati.

Codice affidabile:

- altamente improbabile che un errore non venga rilevato
- funziona anche con errori multipli
- maggiore il numero di errori multipli gestibili, più affidabile è il codice.

Esempi precedenti

- ripetizione doppia del carattere: rileva 1 errore, non rileva 2 errori
- ripetizione tripla del carattere: rileva 2 errori, corregge 1 errore; non rileva 3 errori, non corregge 2 errori
- parità: rileva un numero dispari di errori sulla sequenza di bit.

Distanza di Hamming

Distanza: misura di quanto due elementi (punti, pacchetti, ...) sono “lontani” secondo qualche metrica: spazio, tempo, ...

In informatica (e in matematica), il concetto di distanza viene adoperato in diversi ambiti.

Parola di codice: sequenza di bit contenente l'informazione sul dato più il controllo.

La **distanza di Hamming** fornisce una misura di “lontananza” tra parole di codice distinte.

Nota: non c’entra **nulla** col codice di Hamming!

Distanza di Hamming tra due parole

La distanza di Hamming misura quanto due parole della stessa lunghezza sono **differenti** tra loro.

Definizione

Il numero di simboli non coincidenti tra le due parole.

Hamming distance = 3 —

A	1	0	1	1	0	0	1	0	0	1
			‡			‡		‡		
B	1	0	0	1	0	0	0	0	1	1

Interpretazione: numero di errori necessari per trasformare una parola nell'altra.

Parole valide e non valide

Dato un codice binario, distinguiamo tra

- parole **valide**: sequenze di bit ottenibili aggiungendo informazione di controllo a un dato iniziale secondo le regole del codice
- parole **non valide**: tutte le altre sequenze di bit.

In una comunicazione:

- chi trasmette genera solo parole valide
- chi riceve controlla se la parola è valida. Una parola non valida segnala un errore di trasmissione.

Esempi

Nei codici visti in precedenza:

- ripetizione doppia del carattere; valide: “aa” , “bb”, “ss”; non valide: “ab”, “as”, “cd”
- ripetizione tripla del carattere; valide: “aaa” , “bbb”, “sss”; non valide: “aab”, “ssa”, “cde”
- bit di parità; valide: “01001011” , “00100001”, “11100111”; non valide: “01001010” , “00100000”, “10100111”

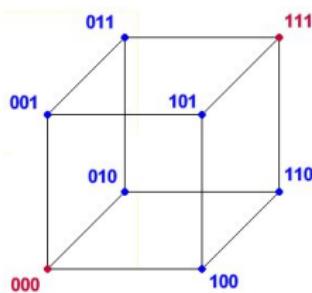
Distanza di Hamming di un codice

Definizione

Distanza di Hamming di un codice: distanza **minima** tra parole valide distinte.

Esempi

- parità: distanza di Hamming = 2;
- ripetizione tripla dei caratteri: distanza di Hamming = 3;



- codice di Hamming a 7 bit: distanza di Hamming = 3;
- codici Reed-Solomon (CD, DVD): distanza di Hamming > 5. Correggono errori multipli.

Proprietà fondamentali

Un codice con distanza di Hamming n :

- **rileva** errori che modificano sino a $n - 1$ bit in una parola
- **corregge** errori che modificano sino a $(n - 1)/2$ bit in una parola.

Memorizzazione dati

Memoria: divisa in **unità** (locazioni). Ogni indirizzo individua un'unità.

- Dimensione standard di una locazione di memoria: 8 bit (1 byte).
- Dimensione usuale per gli indirizzi: 32 bit.

I calcolatori operano su parole di 32-64 bit (4-8 byte)

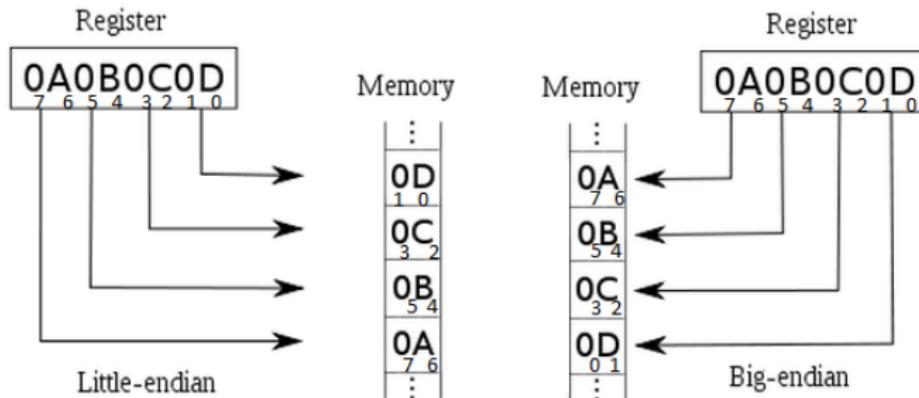
Come scrivere una parola che occupa più di un'unità in memoria?

Una parola viene distribuita su un gruppo di unità contigue.

Little-endian vs. Big-endian

Due modi possibili, scorrendo la memoria per indirizzi crescenti delle unità.

- **Little-endian**: la parola viene scritta in memoria partendo dal byte **meno** significativo.
- **Big-endian**: la parola viene scritta in memoria partendo dal byte **più** significativo.



Diverse scelte

Processori Intel: little-endian. Altri processori: big-endian. Gli stessi dati vengono memorizzati in memoria in modo diverso. Trasferimento:

- alcuni dati, come i numeri (interi, reali), sono memorizzati in modo diverso e quindi **devono** essere riordinati quando passano da un processore little-endian a uno big-endian e viceversa;
- altri dati, come le stringhe (sequenze di caratteri), sono memorizzate allo stesso modo e quindi **non devono** essere riordinati quando passano da little-endian a big-endian e viceversa.

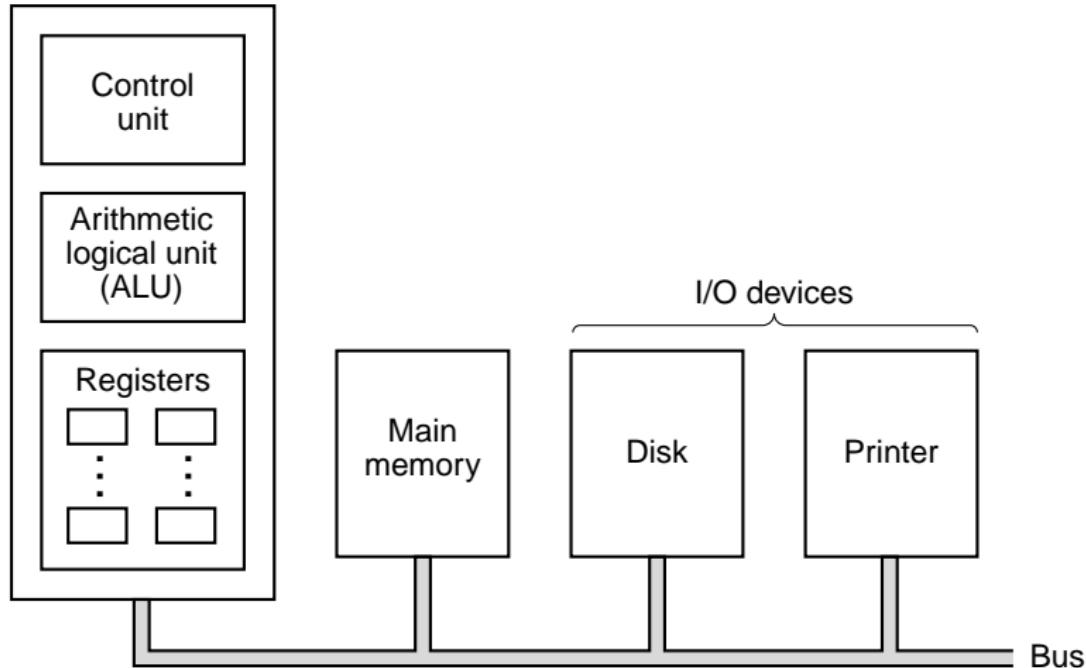
Processore

Presentazione a livelli del calcolatore. Dopo porte logiche, circuiti base e memorie, esaminiamo il **processore** (Central Processing Unit o **CPU**): cuore del calcolatore, l'unità che esegue le istruzioni macchina (cap. 2 e 4 del libro di testo).

Nel seguito presenteremo le altre componenti del calcolatore: bus, I/O, memoria di massa.

Struttura schematica di un calcolatore

Central processing unit (CPU)



Il processore (CPU)

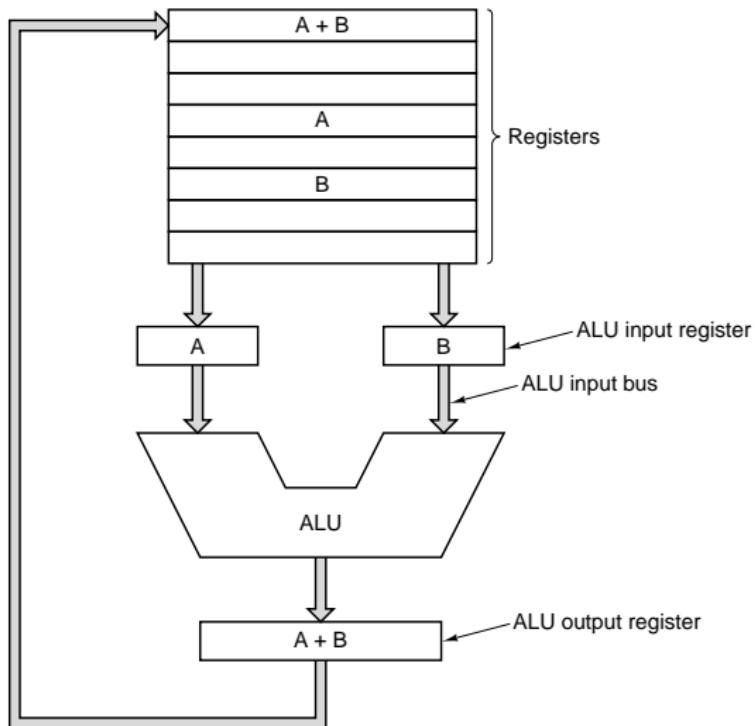
Compito del processore: eseguire il ciclo
fetch-decode-execute il più rapidamente possibile

- **fetch**: preleva un'**istruzione macchina** dalla memoria
- **decode**: determina il tipo di istruzione e i suoi argomenti
- **execute**: recupera gli argomenti, esegue un'operazione, memorizza i risultati.

Il ciclo si ripete.

Componenti processore: Data Path

Esempio: somma di A e B



Data Path

Formato da:

- una serie di **registri** (locazioni di memoria),
- un'unità aritmetica e logica (**ALU**),
- dei **bus** di collegamento.

Può eseguire **micro-operazioni**.

Micro-operazione

Operazione eseguibile nel data path in un **singolo** ciclo di clock.

Azioni realizzabili da una micro-operazione:

- una **singola operazione** aritmetica o logica (i cui argomenti e risultato risiedono nei registri)
- un **accesso alla memoria** (lettura o scrittura di una locazione).

Un'**istruzione macchina** si compone di una o più micro-operazioni.

Unità di controllo

Il funzionamento del data path viene regolato, mediante segnali, dall'**unità di controllo**.

L'unità di controllo è un circuito sequenziale che regola il funzionamento del processore:

- esamina l'istruzione corrente, contenuta nell'**Instruction Register** (IR)
- invia segnali di lettura e scrittura ai registri
- invia segnali di controllo alla ALU
- gestisce la comunicazione con la memoria principale.

Logica cablata o programmata

Due alternative.

- **Logica cablata:** si realizza in hardware un circuito sequenziale classico. Più complicato e costoso da realizzare, offre prestazioni migliori.
- **Logica programmata:** l'unità di controllo è a sua volta una micro-architettura capace di eseguire un micro-programma. Più semplice e flessibile, ma più lento.

Diversi esempi di questa dicotomia: istruzioni grafiche, elaborazione dei segnali, ...

Logica cablata o programmata?

I primi processori: poche istruzioni, logica cablata.

Anni '50:

- prime unità di controllo micro-programmate; permettono la realizzazione di calcolatori economici ma con un ricco insieme di istruzioni
- calcolatori con maggiori prestazioni usano la logica cablata.

Si costruiscono calcolatori molto diversi per prestazioni e costi con lo stesso insieme di istruzioni.
Es.: IBM 360.

Anni '70: micro-programmazione

Le potenzialità della micro-programmazione vengono sfruttate al massimo.

- Linguaggi macchina sempre più sofisticati, vicini ai linguaggi di programmazione standard.
- Il calcolatore apice di questa filosofia: VAX (Digital EC).
- Motivazioni tecnologiche: all'epoca i control store (memorie micro-programmate) sono molto più veloci della memoria principale (RAM).

Anni 80: processori RISC

Si cambia rotta: poche istruzioni, logica cablata.

RISC: Reduced Instruction Set Computer:

- istruzioni semplici ora sono molto più veloci
- si possono utilizzare controlli cablati
- la velocità della RAM si avvicina a quella del control-store.

Le prime macchine RISC

- IBM 801
- CPU-RISC (Patterson - Berkeley) ⇒ SUN Sparc
- MIPS (Hennessy - Stanford)
- Alpha (Digital), PowerPC (Motorola) IBM, ARM.

Anni 90: diatriba RISC – CISC

CISC: Complex Instruction Set Computer.

- Tutti i processori di nuova concezione sono RISC.
- Però: i processori usati nei PC Intel x86 (IA-32) sono CISC.
- Motivi: compatibilità con il software preesistente.

Attualmente

La contrapposizione RISC – CISC è più sfumata.

- La legge di Moore ha portato alla creazione di processori RISC con insiemi di istruzioni sempre più ampi.
- I processori IA-32 usano al loro interno un cuore RISC:
 - istruzioni semplici, ricorrenti: eseguite direttamente,
 - istruzioni complesse: scomposte in più istruzioni semplici,
 - istruzioni sofisticate: eseguite mediante micro-programma.

RISC – CISC oggi

- In linea di principio i processori RISC sono preferibili
- le architetture CISC hanno uno svantaggio in termini di efficienza del 20-30%
- per i processori x86 lo svantaggio tecnologico è compensato dalle economie di scala (vengono prodotti in gran numero, costi minori per unità prodotta).

Dal programma al codice macchina

Un programma ad alto livello (Java, C, C++, Scheme, Pascal, ...) deve essere **tradotto** in linguaggio macchina (istruzioni eseguibili dal calcolatore).

Due alternative:

- **compilatore**: programma **traduttore**, dal programma **sorgente** genera del codice macchina equivalente
- **interprete**: programma che traduce il programma sorgente mappando ogni riga/blocco su codice macchina (**non** genera codice intermedio).

I processori nel libro di testo

Per illustrare con maggior dettaglio il funzionamento dei processori, nel testo si mostra un esempio concreto di processore.

Più precisamente: si mostra come funziona il processore (**Mic**) che esegue le istruzioni del **bytecode** Java. Mic **esegue** Java. Altrove vedrete la *Java Virtual Machine* (JVM).

A lezione: presentazione sintetica. Descrizione dei principi, pochi dettagli.

Problema: non esiste un insieme di pagine del libro che non sia sovrabbondante (molti più argomenti di quelli presentati a lezione).

Processori Mic-(1,2,3,4)

Implementazione hardware, cablata (non virtuale o programmata) della JVM.

Mic è un processore capace di eseguire un **sottoinsieme** del Java bytecode.

Esistono delle vere implementazioni di Mic:

- picoJava (Sun Microsystems)
- ARM926EJ-S (architettura ARM estesa con il Java bytecode).

Il modello di memoria

Memoria standard processori, monolitica, uno stesso spazio per programmi e dati.

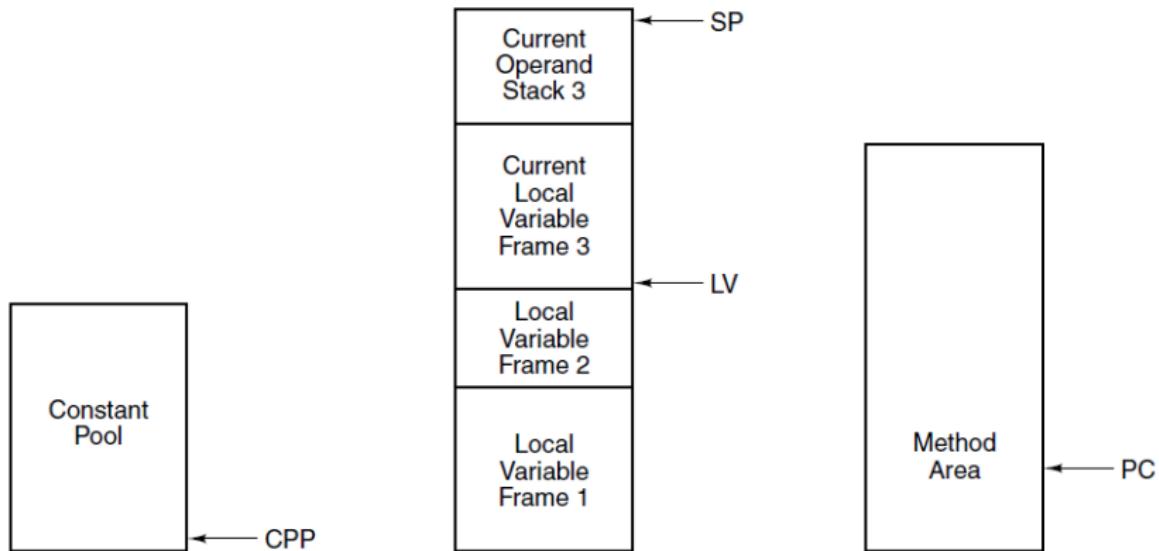
La memoria di Mic è organizzata in tre parti.

- area del codice (programma)
- area delle costanti (usate nel programma)
- area stack (dati usati in una procedura).

Lo stack è a sua volta suddiviso in frame che evolvono dinamicamente.

Es.: JVM valuta l'espressione $(3 + 5) \times (4 + 2)$ allocando prima un frame 1: $? \times ?$, poi un frame 2: $3 + 5$, poi un frame 2: $4 + 2$. A questo punto il frame 1 è risolubile e la procedura termina.

Il modello di memoria



Frame

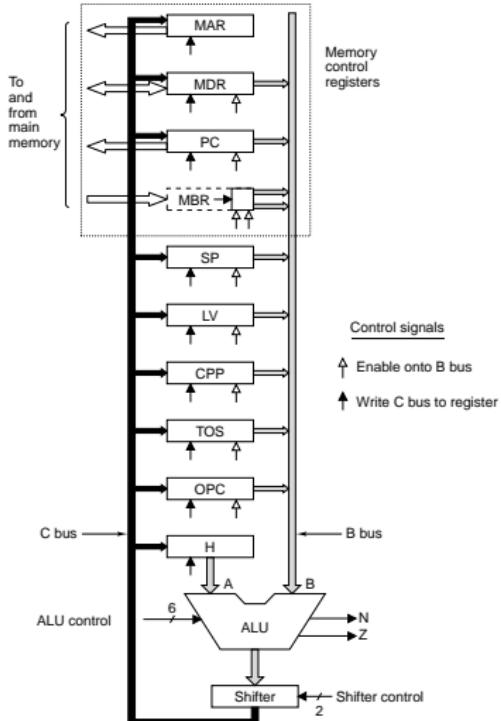
- Usati per gestire dinamicamente le **chiamate di procedura**
- ogni procedura ha le sue **variabili locali**
- a ogni chiamata di procedura si alloca un frame nello stack
- il frame viene rilasciato quando la procedura termina.

I-Java bytecode

Da ricordare solo indicativamente!

- Operazioni aritmetiche e logiche: IADD, IAND, IOR, ISUB, IINC vn con.
- Operazioni trasferimento dati, da e per la memoria: ILOAD vn, ISTORE vn, LDC_W i, (DUP, POP, SWAP, BIPUSH b).
- Operazioni per il controllo del flusso dell'esecuzione:
GOTO, IF_EQ os, IF_LT os, IF_ICM_EQ os,
Chiamate di metodi: INVOKEVIRTUAL d, IRETURN.
- Altro: NOP, WIDE.

Mic-1: data path



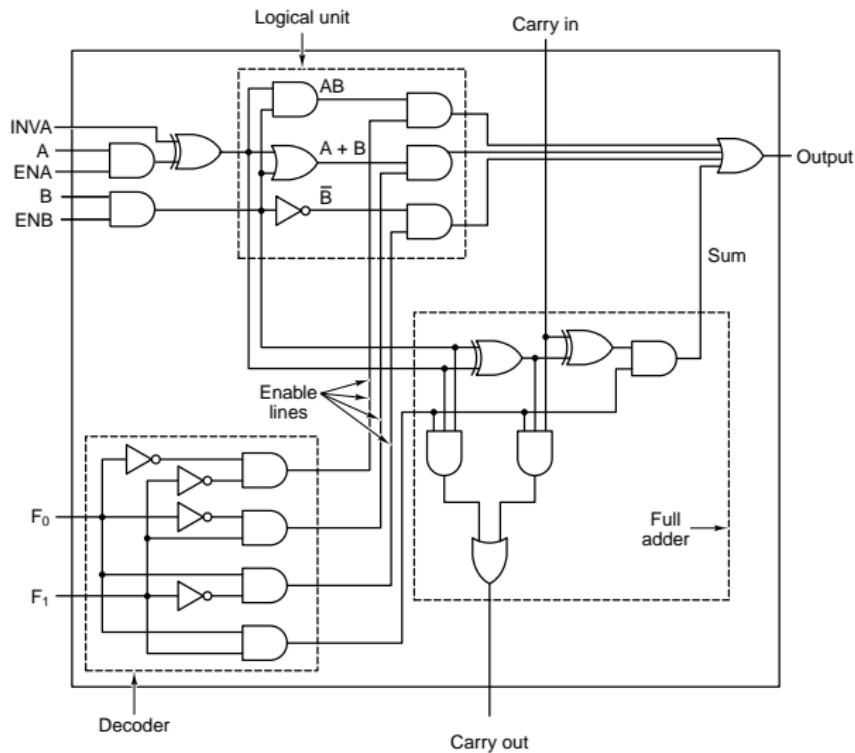
Basta ricordare a grandi linee!

Unità aritmetico-logica (ALU)

Mic-1 usa una semplice ALU.

- Poche operazioni: AND, OR, NOT, somma, sottrazione, incremento, decremento 1, opposto.
- Operazioni eseguite bit per bit (locali): ALU scomposta in unità elementari, ciascuna operante su una coppia di bit.
- Segnali di controllo: selezionano l'operazione da svolgere, abilitano o meno gli ingressi, forniscono il riporto per le cifre meno significative.

Mic-1: ALU (modulo a 1 bit)



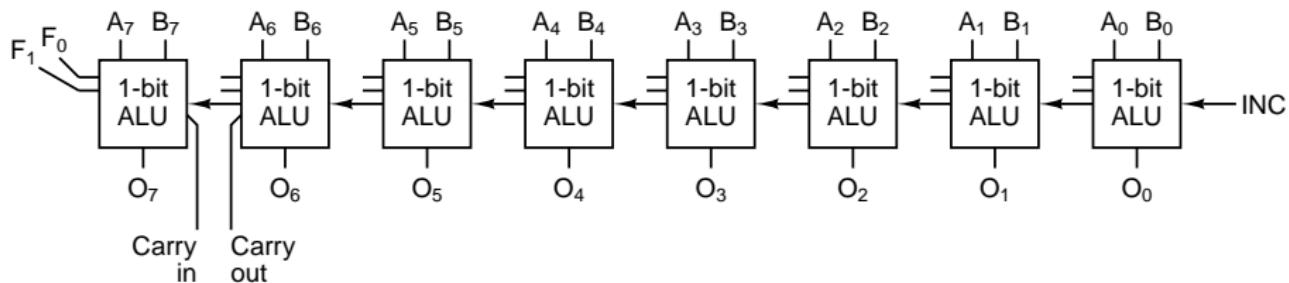
Operazioni aritmetiche nella ALU

Operazioni possibili per il sommatore:

- $A + B + 1$ con ENA=1, ENB=1, INVA=0, Carry=1
- $A + 1$ con ENA=1, ENB=0, INVA=0, Carry=1
- $B + 1$ con ENA=0, ENB=1, INVA=0, Carry=1
- $-A$ con ENA=1, ENB=0, INVA=1, Carry=1
- $B - A$ con ENA=1, ENB=1, INVA=1, Carry=1
- $B - 1$ con ENA=0, ENB=1, INVA=1, Carry=0.

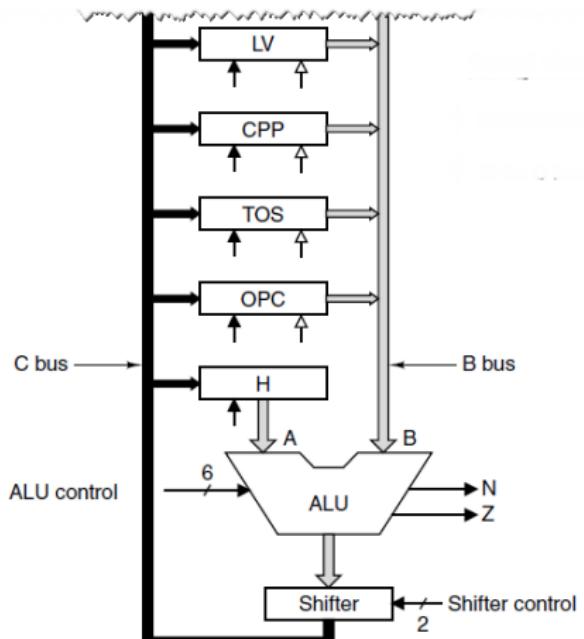
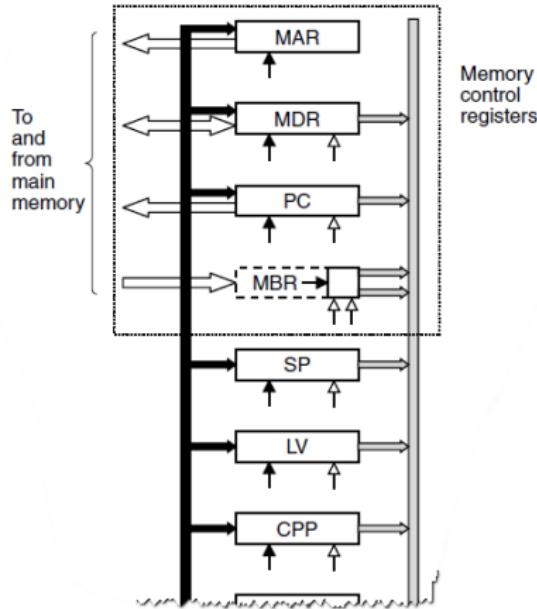
ALU da 8 (, 32, 64) bit

ALU completa formata da una cascata di ALU a 1 bit



Mic-1 contiene una ALU a 32 bit.

Mic-1: registri



Registri (specializzati)

- **MAR**: Memory Address Register
MDR: Memory Data Register (dati)
- **PC**: Program Counter
MBR: Memory Bytecode Register (istruzione)
- **SP**: Stack Pointer
TOS: Top Of Stack (dati)
- **LV**: Local Variable; indirizzo variabili
- **CPP**: Constant Pool Pointer; indirizzo costanti
- **OPC, H**: registri ausiliari.

Da ricordare solo indicativamente!

Esempio di implementazione

Ipotizzando di trovarsi in uno stack frame etichettato Main1, l'istruzione macchina IADD viene realizzata dalla sequenza di micro-istruzioni seguente:

- **Main1:** PC = PC + 1; fetch; goto (MBR)
- **iadd1:** MAR = SP = SP-1; rd
- **iadd2:** H = TOS
- **iadd3:** MDR = TOS = MDR+H; wr; goto Main1.

Le altre istruzioni sono implementate in maniera concettualmente analoga.

Circuito di controllo

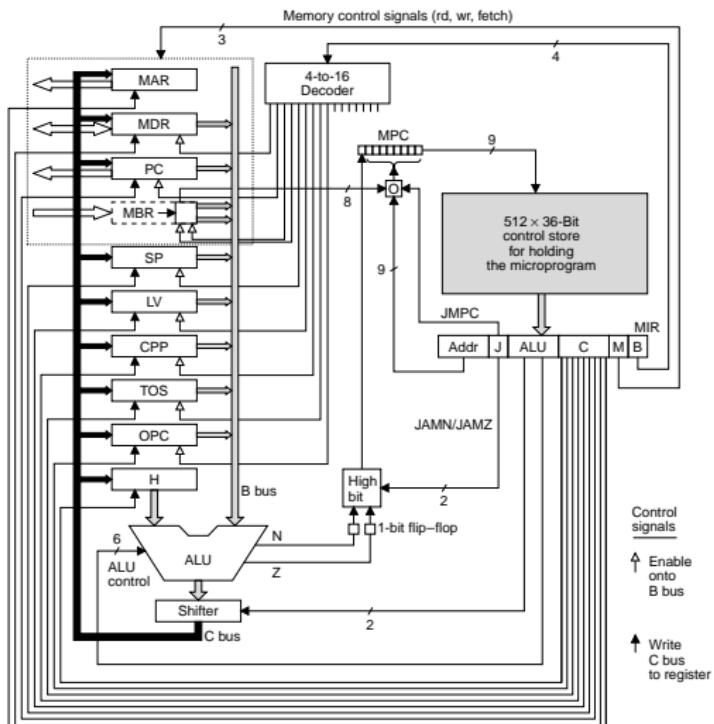
Micro-programmato. Memoria ROM (contiene il micro-codice), 2 registri, un multiplexer:

- semplice sia il circuito che la progettazione
- relativamente lento.

Micro-istruzioni di 36 bit:

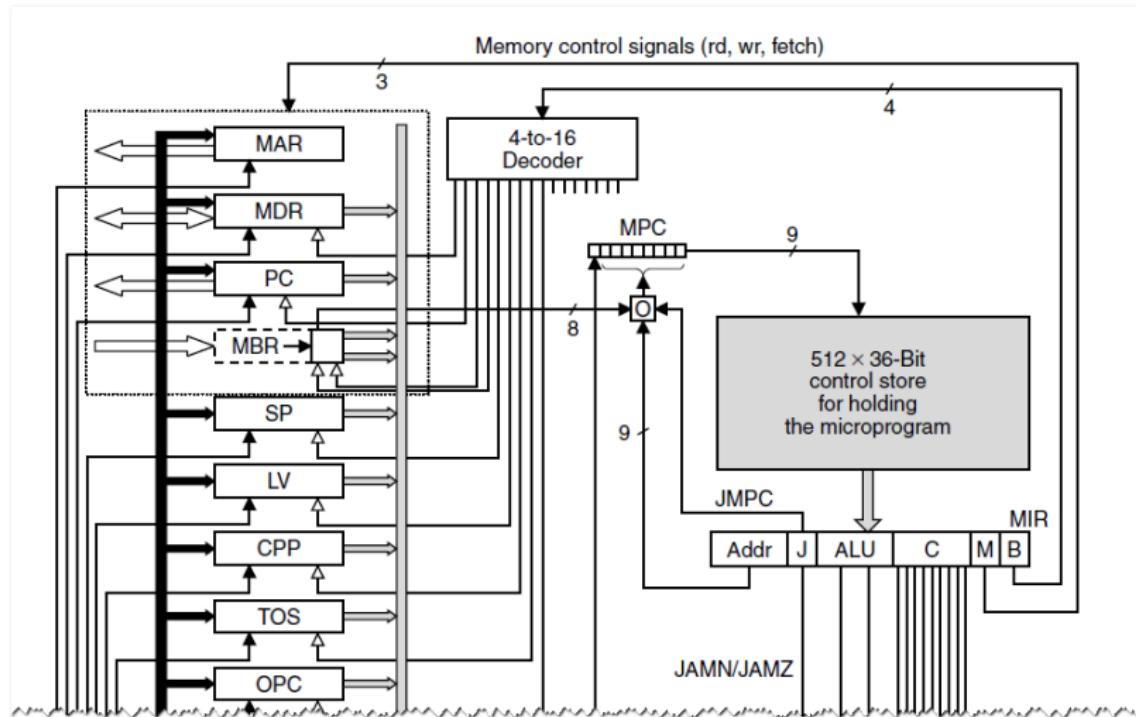
- in parte segnali di controllo
- in parte determinano la micro-istruzione successiva.

Circuito di controllo



Da ricordare solo indicativamente!

Circuito di controllo: Control store



Circuito di controllo

Invia:

- ai registri i segnali di lettura e scrittura;
- alla memoria i segnali read, write, fetch;
- alla ALU il codice dell'istruzione da eseguire.

Determina l'istruzione successiva via:

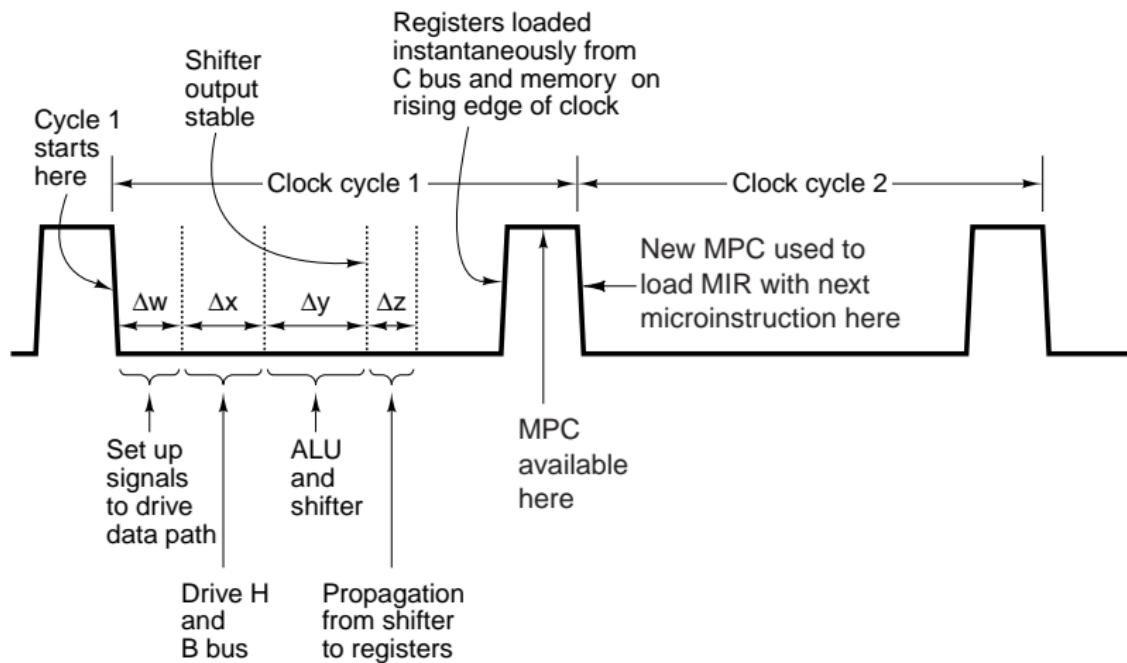
- MBR (prima micro-istruzione nell'esecuzione di un'istruzione macchina);
- la micro-istruzione corrente (micro-istruzioni successive);
- in alcuni casi, il primo bit dell'indirizzo determinato dall'uscita della ALU (per poter implementare i salti condizionati).

Miglioramento delle prestazioni

- Istruzioni macchina più potenti a parità di frequenza di clock.
- Ridurre il ciclo di clock.
- Più istruzioni nell'unità di tempo:
 - diminuire il numero di micro-istruzioni (cicli di clock) necessarie per eseguire un'istruzione macchina
 - aumentare le operazioni svolte nel ciclo di clock.

Ridurre il ciclo di clock

Il periodo di clock dev'essere maggiore della somma dei ritardi necessari alla CPU per funzionare.



Ridurre il ciclo di clock

Due modi:

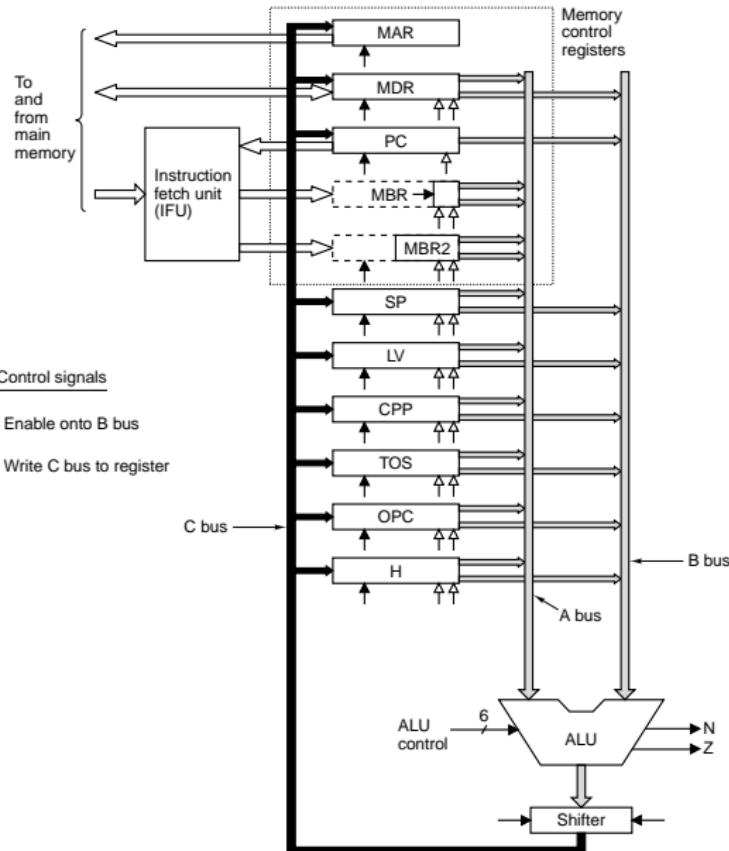
- migliorare la tecnologia dei circuiti integrati
(transistor più veloci)
- migliorare l'organizzazione dei circuiti integrati:
meno ritardi a parità di transistor impiegati.

Più operazioni per ciclo di clock

Aumentare la potenza di calcolo del data path;
micro-istruzioni più potenti:

- ALU con più funzioni disponibili
- più registri disponibili nel processore
- più possibilità di scambio dati (3 bus distinti)
- un'unità separata per il caricamento delle istruzioni: **Instruction Fetch Unit (IFU)**.

Mic-2: IFU

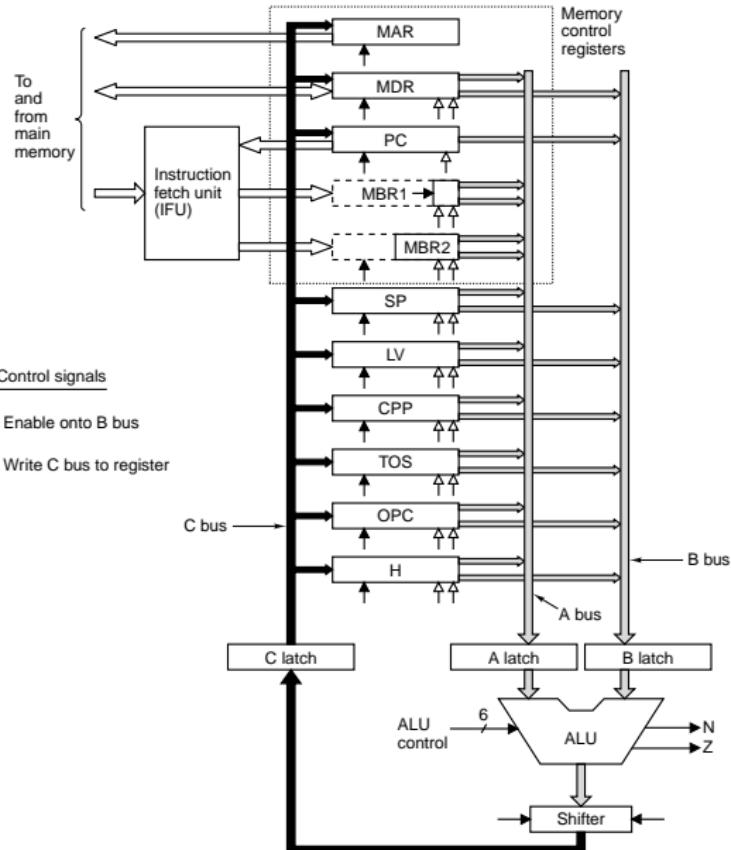


Mic-3: parallelismo sui bus

Pipeline:

- **esecuzione parallela**, spezzare l'esecuzione della micro-operazione in più **stadi**: tecnica del **pipelining**
- l'esecuzione della micro-operazione viene divisa in tre stadi
- ogni stadio è eseguito in parallelo
- i diversi stadi eseguono in contemporanea più istruzioni
- i percorsi lungo i bus sono in generale più brevi. Ciò permette di aumentare la frequenza di clock.

Mic-3: bus pipeline



Pipeline: esempio

Possibile paragone preso dalla vita comune.

Lavanderia:

vestiti da lavare	micro-operazioni
lavaggio	acquisire i dati di ingresso
asciugatura	calcolo del risultato
stiratura	memorizzazione del risultato.

Le varie fasi di lavoro sono portate avanti in parallelo.

Pipeline: altro esempio

Catena di montaggio:

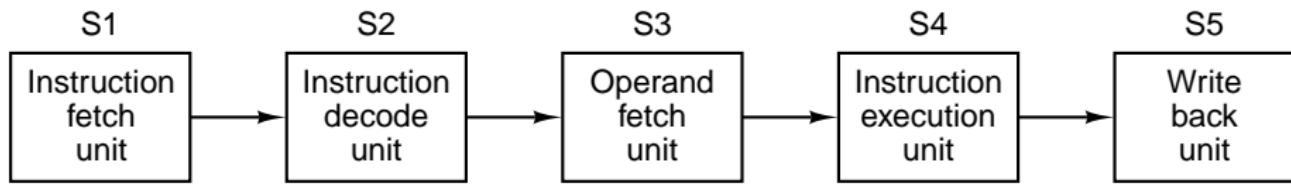
il lavoro viene scomposto in fasi, ognuna eseguita da un agente specifico; si svolgono le diverse fasi, su agenti diversi, in parallelo.

In un processore si parallelizza l'esecuzione delle micro-operazioni.

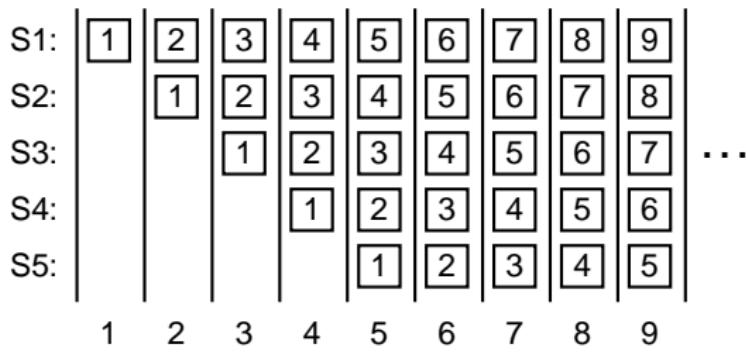
Tutto ciò migliora la **banda passante** ma non i **tempi di risposta**.

Tecnica utilizzata in tutti i processori. In ambito Intel: dal 486 in poi (1989).

Esempio di scomposizione

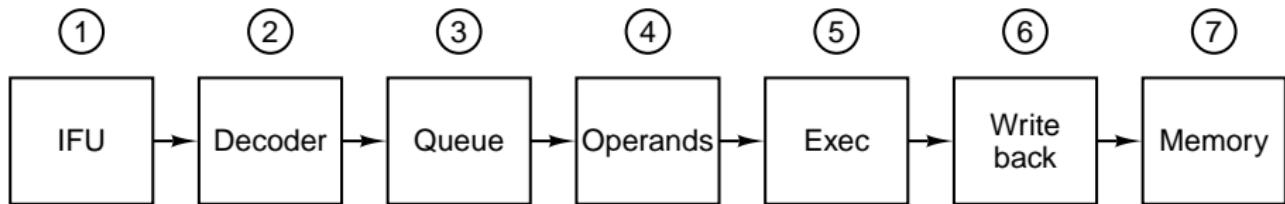


(a)



(b)

Altro esempio



Processori diversi usano scomposizioni e strutture della pipeline diverse.

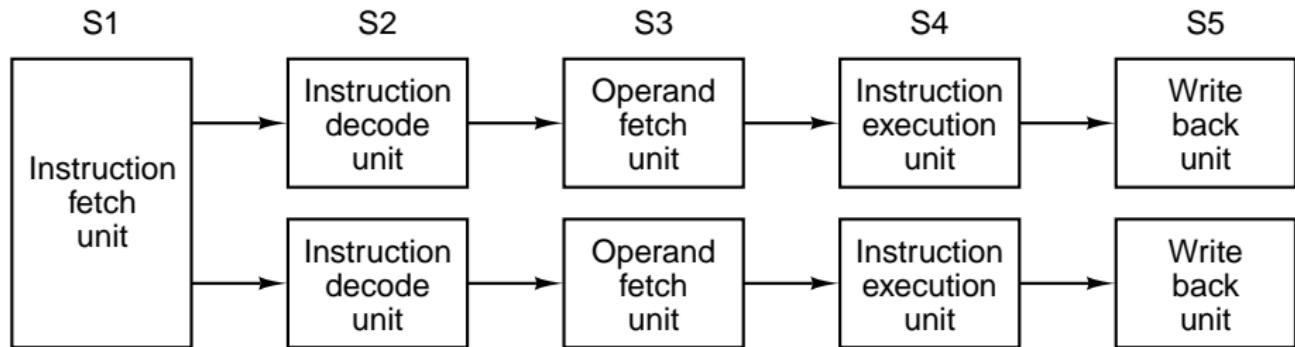
Lunghezza tipica di una pipeline: 7–14 stadi.

Caso limite (Pentium IV): 20 stadi (guerra dei GHz).

Processori superscalari

Aumentano ulteriormente il parallelismo: migliora il rapporto istruzioni / cicli di clock.

Iniziano contemporaneamente l'esecuzione di più istruzioni: più pipeline operanti in parallelo.

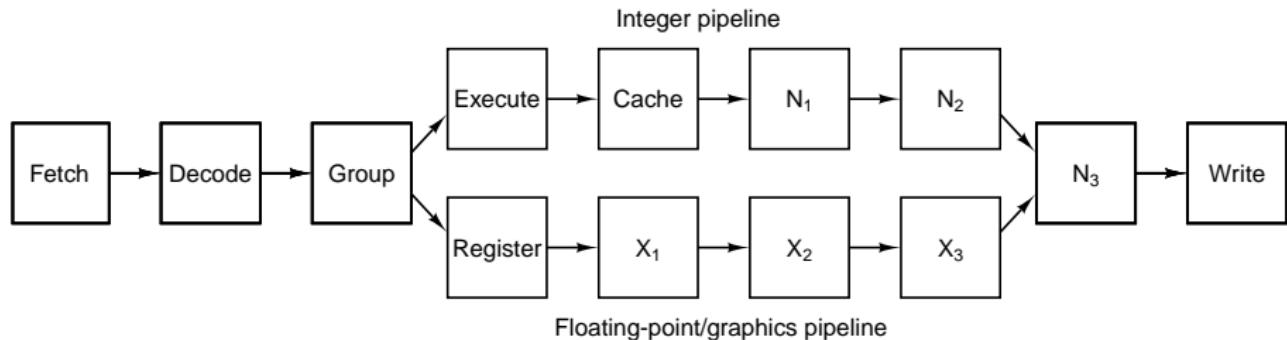


Processori superscalari

- I primi stadi (singoli) prelevano più istruzioni dalla memoria e le decodificano
- le istruzioni sono smistate su stadi successivi multipli
- dev'esserci uno stadio finale singolo o un sistema di controllo che termina ordinatamente le istruzioni.

Processore SPARC (Sun)

Pipeline specializzate.



Attualmente processori con 4-15 pipeline, decine di micro-istruzioni in contemporanea.

Problemi del parallelismo

I processori superscalari possono potenzialmente eseguire decine di istruzioni contemporaneamente.

Due vincoli impediscono un completo sfruttamento del parallelismo:

- dipendenza tra istruzioni
- istruzioni di salto.

Dipendenza tra istruzioni:

In un programma le istruzioni sono state pensate per essere eseguite in ordine.

Un'esecuzione parallela, senza controlli, può portare a risultati scorretti. Tre casi:

- **RAW Read After Write**

$$R0 = R1$$

$$R2 = R0 + 1$$

- **WAR Write After Read**

$$R1 = R0 + 1$$

$$R0 = R2$$

- **WAW Write After Write**

$$R0 = R1$$

$$R0 = R2 + 1.$$

Dipendenza tra istruzioni

Le dipendenze vengono rilevate mediante una **tabella delle dipendenze** (scoreboard): memoria interna al processore che conta per ogni registro le operazioni, di lettura e scrittura, in sospeso su quel registro.

Le istruzioni dipendenti devono essere sospese: si creano **bolle**, zone inattive, nella pipeline.

Gestire la dipendenza tra istruzioni

Tecniche per recuperare le prestazioni perse

- **esecuzione fuori ordine**: si mandano in esecuzione le istruzioni non dipendenti
- **registri ombra**: copie di registri su cui memorizzare temporaneamente i dati
- **register renaming**: nuovi registri, non specificati dal codice
- **multi-threading** (hyper-threading): si eseguono più programmi contemporaneamente. È necessario duplicare i registri. Primo passo verso processori multicore.

Istruzioni condizionate di salto

Problematiche per i processori con pipeline:

- il processore impiega alcuni cicli di clock per valutare una **condizione**
- nel frattempo, non sa quali istruzioni eseguire.

Due possibili soluzioni

- **stall**: non si inizia alcuna istruzione; corretta, ma con decadimento delle prestazioni
- **predizione di salto**: **taken, not taken**; si inizia l'esecuzione **condizionata** di alcune istruzioni. L'esecuzione è annullata se la previsione si rivela errata.

In un programma ci sono numerose istruzioni di salto. La predizione assicura migliori prestazioni.

Tecniche per la predizione di salto

Due tecniche:

- predizione **statica**, sul codice
 - semplice: si eseguono in anticipo condizionatamente **tutti** i salti all'indietro
 - con suggerimento del compilatore (profiling del codice) o del programmatore
- predizione **dinamica**, fatta durante l'esecuzione aggiornando una **history table** (ricorda il comportamento passato di alcune istruzioni di salto, poche istruzioni, pochi bit).

Esecuzione speculativa

Nel caso di salto condizionato **non** si tenta la predizione ma si eseguono entrambi i rami del codice che segue.

Pur di alimentare il processore, si eseguono anche alcune istruzioni che sicuramente verranno scartate.

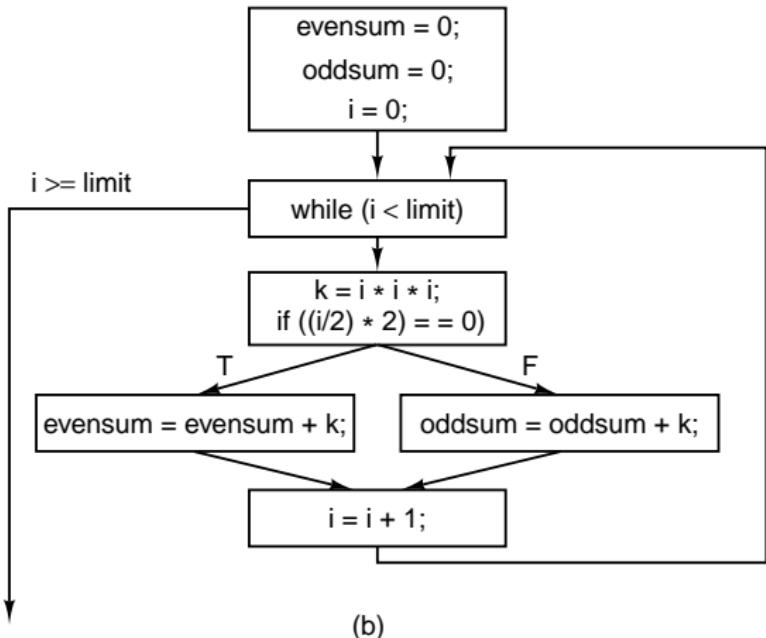
Problemi

- l'esecuzione deve essere **reversibile**: registri ombra, istruzioni che generano trap (**poison bit**)
- evitare l'esecuzione condizionata di costosi *fetch* dalla memoria principale: istruzione macchina **SPECULATIVE-LOAD**.

Esecuzione Speculativa

```
evensum = 0;  
oddsum = 0;  
i = 0;  
  
while (i < limit) {  
    k = i * i * i;  
    if ((i/2) * 2 == 0)  
        evensum = evensum + k;  
    else  
        oddsum = oddsum + k;  
  
    i = i + 1;  
}
```

(a)



(b)

Il **pre-load** delle variabili in memoria è delegato al compilatore. Il **pre-store** non è conveniente.

Errore if((i/2)*2 == 0) (corretto nel libro di testo).

La memoria principale troppo lenta

La differenza di velocità tra processore e memoria è aumentata col tempo.

L'accesso in memoria è un'operazione costosa: il processore deve attendere il dato anche per più di **una decina di cicli** di clock.

Memoria **cache**: memoria costosa e veloce: efficace se contiene i dati utilizzati più frequentemente.

La memoria cache

Funzionamento:

- prima si cerca il dato in cache (cache **hit**)
- in caso di fallimento (cache **miss**): si carica il dato in cache dalla memoria principale.

Migliori prestazioni solo con numerosi cache hit.

Sfruttamento della regolarità statistica della località spaziale e **temporale** dei dati.

Cache a più livelli

Per evitare troppi cache miss si possono prevedere più livelli di cache.

Cache di **secondo livello**:

- più ampia
- tecnologia meno costosa
- più lenta
- contiene un sovrainsieme della cache di primo livello.

Spesso sono presenti 3 livelli di cache.

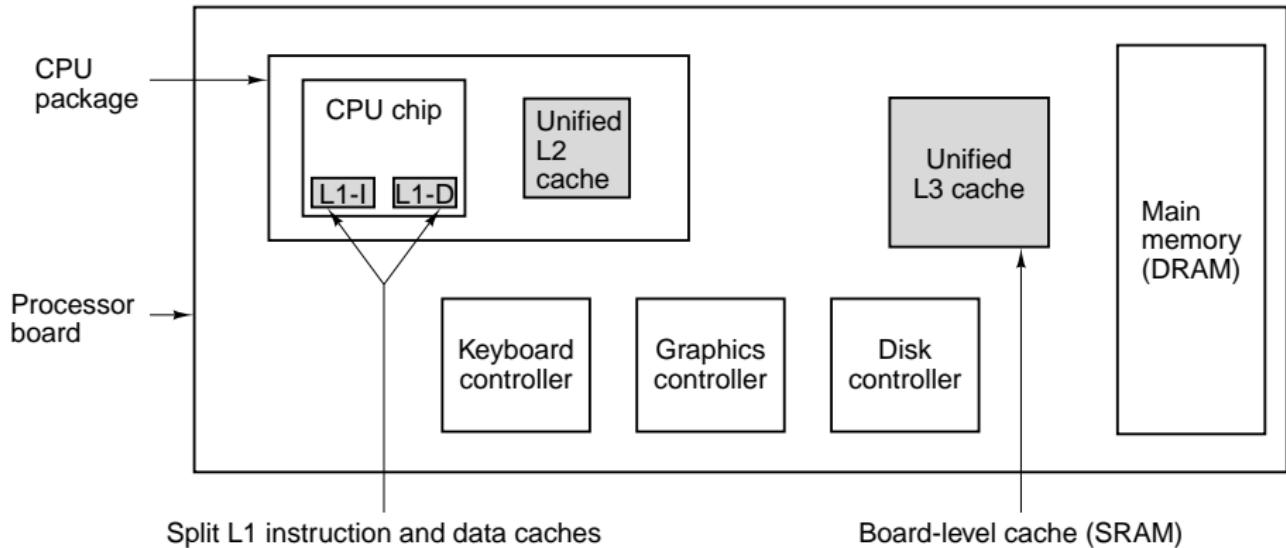
Es.: Intel Core i7 (Sandy Bridge): livello 1: 32KB; livello 2: 256KB, livello 3: 1–20MB.

Split cache

Le moderne cache sono divise in **due parti**: dati e istruzioni.

- Normalmente cache **L1** (di primo livello)
- accesso in memoria parallelizzato:
 - l'IFU accede alle istruzioni
 - l'unità Dispatch/Execute accede ai dati
- sostanziale raddoppio degli accessi alla memoria nell'unità di tempo.

Esempio di configurazione



Valutazione delle prestazioni

Le velocità di un calcolatore con processore superscalare dipende fortemente da quanto viene sfruttato il potenziale parallelismo:

- percentuale delle istruzioni non bloccate per dipendenze
- percentuale di predizioni di salto corrette
- percentuale cache hit.

Queste percentuali sono difficilmente valutabili a tavolino, dipendono dal tipo di programmi eseguiti. Una corretta valutazione delle prestazioni può essere fatta solo tramite test.

IEEE 754 FLOATING POINT NOTES AND EXAMPLE

POWER OF 2 TABLE

$$2^{10} = 1024$$

$$2^9 = 512$$

$$2^8 = 256$$

$$2^7 = 128$$

$$2^6 = 64$$

$$2^5 = 32$$

$$2^4 = 16$$

$$2^3 = 8$$

$$2^2 = 4$$

$$2^1 = 2$$

$$2^0 = 1$$

.

$$2^{-1} = 0.5$$

$$2^{-2} = 0.25$$

$$2^{-3} = 0.125$$

$$2^{-4} = 0.0625$$

$$2^{-5} = 0.03125$$

$$2^{-6} = 0.015625$$

The following is a step by step roadmap to go from a decimal number to its IEEE 754 32 bit floating point representation. The example will use -176.375 as an example

STEP 1: OBSERVE THE SIGN

For -176.375 the sign is negative. This means the first bit will be 1, if positive the first bit will be 0. Going forward the sign will be ignored, but then used again in the last step.

STEP 2: FROM DECIMAL TO BINARY

Transform the decimal to binary (ignoring the sign). To do this subtract the largest power of 2 relative to the decimal until you reach 0. Note that floating point is an approximation and cannot be perfectly represented – but the potential rounding error is very small.

<i>Calculations</i>				
Use 2^7	Use 2^5	Use 2^4	Use 2^{-2}	Use 2^{-3}
176.375	48.375	16.375	0.375	0.125
- <u>128.000</u>	- <u>32.000</u>	- <u>16.000</u>	- <u>0.250</u>	- <u>0.125</u>
48.375	16.375	0.375	0.125	0.000

As a sum : $2^7 + 2^5 + 2^4 + 2^{-2} + 2^{-3} = 176.375$

As binary : 10110000.011 (Place a 1 in each position used)

STEP 3: MOVE TO SCIENTIFIC NOTATION AND GET SIGNIFICANT

IEEE Floating points need to be in the format of $1.\text{xxxxx} * 2^y$. The significant is the *xxxxx* component (ignore the 1.) and has 23 bits. If your significant is shorter than 23 bits add trailing zeros.

$$10110000.011 = \underbrace{1.0110000011}_{\text{Significant}} * 2^7$$

Significant = 01100001100000000000000 (had to add 13 trailing zeros)

STEP 4: CALCULATE EXPONENT IN BINARY

The exponent is represented by 8 bits (256 states) and is shifted by 127. In our example ($1.0110000011 * 2^7$) the exponent is 7. So we need to express 134 (from $7+127$) in binary. Using the same technique as step 2:

As a sum : $128 + 4 + 2 = 134$

As a sum : $2^7 + 2^2 + 2^1 = 134$

As binary : 10000110 (Place a 1 in each position used)

STEP 5: COMBINE SIGN, EXPONENT, AND SIGNIFICANT

The format is:

Sign (1 bit)	Exponent (8 bits)	Significant (23 bits)
1	10000110	01100001100000000000000

Or: 11000011001100000110000000000000 (Hex : 0xc3306000)

Converting to/from IEEE 754 single-precision floating point format

-Ian! D. Allen - idallen@idallen.ca - www.idallen.com

For an online converter that lets you check your work,
see: <http://www.h-schmidt.net/FloatApplet/IEEE754.html>

Unless otherwise stated, convert using the IEEE 754 single precision format.

The IEEE 754 single precision format is 32 bits, which are (from left to right):

One bit for the sign;
8 bits for the exponent and;
23 bits for the mantissa/significand.

The IEEE 754 double precision format is 64 bits, which are (from left to right):

One bit for the sign;
11 bits for the exponent;
52 bits for the mantissa/significand.

(In this course, we don't work with double-precision numbers.)

The conversions in this file only work for non-zero numbers. The IEEE 754 bit pattern for zero is 00000000h. Negative zero is 80000000h.

EXAMPLE 1: Show that decimal 147.625 equals binary 4313A000h in IEEE 754 format.

Step 1: Convert the decimal number to its binary fractional form.

(Convert the parts on each side of the binary decimal point separately, using the techniques already learned in class.)

The decimal number 147.625 converted to binary is: 10010011.101

Note that the implied binary exponent multiplier here is "1" (2^{**0}).

Step 2: Normalize the binary fractional number.

Move the decimal point left or right so that only a single binary digit "1" is to the left of the binary decimal point. Compensate by adjusting the exponent in the opposite direction.

10010011.101 times 2^{**0} --> (move binary decimal point left 7)
--> 1.0010011101 times 2^{**7}

Moving the decimal left seven decreases the size of the number; so, we use an exponent of 7 to compensate and keep the number the same size.

(Moving right would increase the size of the number, and the exponent would have to be negative, e.g. 0.001101 --> 1.101 times $2^{**(-3)}$.)

Step 3: Convert the exponent to 8-bit excess-127 notation.

Add 127 to the exponent and convert it to 8-bit binary:

7 + 127 = 134 --> 10000110 (= 128 + 4 + 2)

Step 4: Convert the mantissa/significand to "hidden bit" format.

Since every binary floating-point number (except zero!) is normalized with "1." at the start, there is no need to store that leftmost "1". Remove the leading "1." from the mantissa/significand:

1.0010011101 --> 0010011101

Step 5: Write down the $1+8+23 = 32$ bits.

147.625 is positive - the sign bit is zero: 0
The next eight bits are the exponent: 100000110
The next 23 bits are the mantissa: 0010011101000000000000000

Binary result (32 bits): 01000011000100111010000000000000

Step 6: Convert the 32 bits to hexadecimal, starting on the right.

To convert 01000011000100111010000000000000 to hexadecimal, group the bits into chunks of four, starting from the right, and convert each four-bit chunk to its hexadecimal digit 0-9A-F:

0100 0011 0001 0011 1010 0000 0000 0000
Answer (hex): 4 3 1 3 A 0 0 0

Answer: 147.625 is 4313A000h in IEEE 754 single-precision format.

EXAMPLE 2: Show that decimal 2004 equals binary 44FA8000h in IEEE 754 format.

Step 1: decimal 2004 --> binary 11111010100.0 (no fractional part)

Step 2: normalize binary 11111010100.0 --> 1.11110101000 times 2**10

We shifted the binary decimal point 10 places to the left, making the number smaller. The exponent needed to compensate is therefore 2**10

Step 3: convert exponent 10 --> 10 + 127 = 137 --> binary 10001001 (=128+8+1)

Step 4: remove hidden digit from 1.11110101000 --> 11110101000

Step 5: Write down the 1+8+23 = 32 bits.

2004 is positive - the sign bit is zero: 0
The next eight bits are the exponent: 10001001
The next 23 bits are the mantissa: 1111010100000000000000000

Binary result (32 bits): 01000100111101010000000000000000

Step 6: 0100 0100 1111 1010 1000 0000 0000 0000
4 4 F A 8 0 0 0

Answer: 2004 is 44FA8000h in IEEE 754 single-precision format.

EXAMPLE 3: Show that decimal -20.5 equals binary C1A40000h in IEEE 754 format.

Step 1: decimal 20.5 --> binary 10100.1

Step 2: normalize binary 10100.1 --> 1.01001 times 2**4

Step 3: exponent 4 + 127 = 131 = binary 10000011

Step 4: remove hidden digit from 1.01001 --> 01001

Step 5:

-20.5 is negative - the sign bit is one: 1
The next eight bits are the exponent: 10000011
The next 23 bits are the mantissa: 0100100000000000000000000

Binary result (32 bits): 11000001101001000000000000000000

Step 6: 1100 0001 1010 0100 0000 0000 0000 0000
C 1 A 4 0 0 0 0

Answer: -20.5 is C1A40000h in IEEE 754 single-precision format.

EXAMPLE 4: Show that decimal -0.5 equals binary BF000000h in IEEE 754 format.

Step 1: decimal 0.5 --> binary 0.1

Step 2: normalize binary 0.1 --> 1.0 times 2^{*-1}

We shifted the binary decimal point 1 place to the right, making the number larger. The exponent needed to compensate is therefore 2^{*-1}

Step 3: exponent $-1 + 127 = 126 =$ binary 01111110

Step 4: remove hidden digit from 1.0 --> 0

Step 5:

-0.5 is negative - the sign bit is one: 1
The next eight bits are the exponent: 01111110
The next 23 bits are the mantissa: 0000000000000000000000000

Binary result (32 bits): 10111111000000000000000000000000

Step 6: 1011 1111 0000 0000 0000 0000 0000
B F 0 0 0 0 0 0

Answer: -0.5 is BF000000h in IEEE 754 single-precision format.

EXAMPLE 5: Show that decimal -1 equals binary BF800000h in IEEE 754 format.

Step 1: decimal 1 --> binary 1.0 (no fractional part)

Step 2: normalize binary 1.0 --> 1.0 times 2^{*0}

The binary number 1.0 is already normalized. No need to shift.
The exponent remains zero.

Step 3: exponent $0 + 127 = 127 =$ binary 01111111

Step 4: remove hidden digit from 1.0 --> 0

Step 5:

-1 is negative - the sign bit is one: 1
The next eight bits are the exponent: 01111111
The next 23 bits are the mantissa: 0000000000000000000000000

Binary result (32 bits): 10111111000000000000000000000000

Step 6: 1011 1111 1000 0000 0000 0000 0000
B F 8 0 0 0 0 0

Answer: -0.5 is BF800000h in IEEE 754 single-precision format.

EXAMPLE 6: Show that IEEE 754 binary 438F000h is decimal 286

We run the six steps in reverse order from above:

Step 6:

Convert the hexadecimal into binary:

4 3 8 F 0 0 0 0

0100 0011 1000 1111 0000 0000 0000 0000

Step 5: Split the binary into 1+8+23 bit pieces:

01000011100011110000000000000000
--> 0 10000111 000111100000000000000000

Number is positive - the sign bit is zero: 0
The next eight bits are the exponent: 10000111
The next 23 bits are the mantissa: 0001111000000000000000000

Step 4: add back the missing hidden digit to the mantissa/significand

000111100000000000000000 --> 1.000111100000000000000000

Step 3: convert the exponent to decimal and subtract 127

binary 10000111 (=128+4+2+1) = 135 --> 135 - 127 = 8 --> 2**8

Step 2: de-normalize the mantissa/significand (make exponent zero)

1.000111100000000000000000 times 2**8 --> (must move right 8 places)
--> 10001110.00000000000000 times 2**0

We moved the binary decimal point 8 places to the right, making the number larger, which allowed us to reduce the exponent by the same amount (to zero).

Step 1: binary 10001110.00000000000000 --> decimal 286 (=256+16+8+4+2)

Answer: IEEE 754 binary 438F0000h is decimal 286.

EXAMPLE 7: Show that IEEE 754 binary BF880000h is decimal -1.0625

Step 6:

Convert the hexadecimal into binary:

B F 8 8 0 0 0 0
1011 1111 1000 1000 0000 0000 0000 0000

Step 5: Split the binary into 1+8+23 bit pieces:

10111111000100000000000000000000
--> 1 01111111 000100000000000000000000

Number is negative - the sign bit is one: 1
The next eight bits are the exponent: 01111111
The next 23 bits are the mantissa: 0001000000000000000000000

Step 4: add back the missing hidden digit to the mantissa/significand

000100000000000000000000 --> 1.000100000000000000000000

Step 3: convert the exponent to decimal and subtract 127

binary 01111111 = (2**7)-1 = 127 --> 127 - 127 = 0 --> 2**0

Step 2: de-normalize the mantissa/significand (make exponent zero)

1.000100000000000000000000 times 2**0 --> (no need to move decimal)
--> 1.000100000000000000000000 times 2**0

An exponent of zero means no adjustment is needed.

Step 1: binary 1.000100000000000000000000 --> decimal 1.0625

(= 1 + 1*(2**(-4)) = 1 + 0.0625 = 1.0625)

Answer: IEEE 754 binary BF880000h is decimal -1.0625

EXAMPLE 8: Show that decimal 128.5625 equals binary 43009000h in IEEE 754 format.

Step 1: Convert the decimal number to its binary fractional form.

(Convert the parts on each side of the binary decimal point separately, using the techniques already learned in class.)

The decimal number 128.5625 converted to binary is: 10000000.1001

Note that the implied binary exponent multiplier here is "1" (2**0).

Step 2: Normalize the binary fractional number.

Move the decimal point left or right so that only a single binary digit "1" is to the left of the binary decimal point. Compensate by adjusting the exponent in the opposite direction.

10000000.1001 times 2**0 --> (move binary decimal point left 7)
--> 1.00000001001 times 2**7

Moving the decimal left seven decreases the size of the number; so, we use an exponent of 7 to compensate and keep the number the same size.

Step 3: Convert the exponent to 8-bit excess-127 notation.

Add 127 to the exponent and convert it to 8-bit binary:

7 + 127 = 134 --> 10000110 (= 128 + 4 + 2)

Step 4: Convert the mantissa/significand to "hidden bit" format.

Since every binary floating-point number (except zero!) is normalized with "1." at the start, there is no need to store that leftmost "1". Remove the leading "1." from the mantissa/significand:

1.00000001001 --> 00000001001

Step 5: Write down the 1+8+23 = 32 bits.

128.5625 is positive - the sign bit is zero: 0
The next eight bits are the exponent: 10000110
The next 23 bits are the mantissa: 00000001001000000000000

Binary result (32 bits): 01000011000000001001000000000000

Step 6: Convert the 32 bits to hexadecimal, starting on the right.

To convert 01000011000000001001000000000000 to hexadecimal, group the bits into chunks of four, starting from the right, and convert each four-bit chunk to its hexadecimal digit 0-9A-F:

0100 0011 0000 0000 1001 0000 0000 0000
Answer (hex): 4 3 0 0 9 0 0 0

Answer: 128.5625 is 43009000h in IEEE 754 single-precision format.

*) The IEEE 754 floating-point number 81234567h is negative. Without converting, give the hexadecimal for the same number, only positive.

Turn off the sign bit (8->0): 01234567h

- *) The IEEE 754 floating-point number 7EDCBA98h is positive. Without converting, give the hexadecimal for the same number, only negative.

Turn on the sign bit (7->F): FEDCBA98h

- *) Without converting, circle all the IEEE 754 negative numbers:
1837A654h 7A6A3B65h 87B5CDE2h 90A5B5EFh A0000037h D1B8765Ah F0000000h

Anything with the sign bit on is negative.
(Choose anything starting with hex digits 8..F)

--
| Ian! D. Allen - idallen@idallen.ca - Ottawa, Ontario, Canada
| Home Page: <http://idallen.com/> Contact Improv: <http://contactimprov.ca/>
| College professor (Free/Libre GNU+Linux) at: <http://teaching.idallen.com/>
| Defend digital freedom: <http://eff.org/> and have fun: <http://fools.ca/>

Assembly

Programmazione in **linguaggio macchina**: programmare scrivendo istruzioni direttamente eseguibili dal processore.

Programmazione in **assembly**: programmare utilizzando istruzioni **quasi** direttamente eseguibili dal processore.

Questa parte del corso si accompagna a lezioni in laboratorio: programmazione in assembly con uso di un simulatore.

Motivazioni

- Programmare in assembly aiuta a capire funzionamento e meccanismi base di un calcolatore.
- Fa meglio comprendere cosa accade durante l'esecuzione di un programma scritto ad alto livello.
- Insegna una metodologia, si acquisiscono delle abilità.
- In alcuni casi è efficace programmare in assembly.

Assembly: vantaggi

Controllo dell'hardware. Si definiscono esattamente le istruzioni da eseguire e le locazioni di memoria da modificare.

Utile per

- accesso alle risorse di basso livello del computer ([kernel](#), [driver](#))
- ottimizzare il codice: programmi anche molto più efficienti riscrivendo in assembly poche righe critiche di codice di alto livello.

Assembly: svantaggi

- scarsa portabilità: programmi eseguibili solo da una famiglia di processori
- scomodo: istruzioni poco potenti, programmi lunghi
- facile cadere in errore: programmi poco strutturati e leggibili.

Compilatori sempre più sofisticati producono codice efficiente (soprattutto per le architetture parallele): difficile fare meglio programmando manualmente in assembly.

La famiglia ARM

Tanti linguaggi macchina quante le famiglie di processori: x86 (IA-32 Intel), PowerPC (IBM, Motorola), Sparc, MIPS, ...

Linguaggi sintatticamente e strutturalmente simili, ma con molte differenze dipendenti dall'architettura su cui sono eseguiti.

In questo corso: processore ARM (Acorn RISC Machine), uno dei primi processori RISC (Acorn Computers, Wilson and Furber, 1985): poche istruzioni semplici e lineari.

Architetture ARM

Una famiglia di architetture (**ISA**: Instruction Set Architecture) e processori: ARMv1, ... ARMv8.

Diversi insiemi di istruzioni:

- nel tempo si sono aggiunte istruzioni (divisione, istruzioni vettoriali)
- esistono versioni ARM che inglobano ulteriori istruzioni, come
 - Java bytecode
 - Thumb 16 bit
- architetture a 32-bit: istruzioni, registri interni, indirizzi di memoria.

ARMv8 gestisce registri interni e indirizzi di memoria a 64-bit.

Processore ARM

- Produzione: $\approx 10^{10}$ pezzi l'anno
- architettura a basso consumo
- a bordo del 95% di smartphone, tablet
- ideale per sistemi embedded: televisori digitali, DVD, router, ADSL modem, ...
- Processore ampiamente usato
- insieme di istruzioni semplice ed elegante, facile da imparare
- simile agli altri RISC.

Possibili alternative

- **IA-32 (Pentium)**. Linguaggio difficile da apprendere e da usare. Istruzioni stratificate nell'arco degli anni. Problemi di legacy (compatibilità con linguaggi precedenti).
- **Assembly 8088** (descritto nel libro di testo). Superato, progenitore del IA-32. Diverso dai linguaggi attuali.
- **MIPS**. Istruzioni simili ad ARM, più semplici e lineari. Forse più adatto alla didattica, ma meno presente sul mercato.

Riferimenti

Ampia documentazione su ARM disponibile in rete,
spesso non troppo completa oppure troppo
complessa.

Facciamo riferimento anche a

- manuale completo delle istruzioni ARM (**non si può** portare all'esame)
- tabella sintetica istruzioni ARM (**Reference Chart**: **si può** portare all'esame).

Simulatore

Necessario per realizzare programmi, verificarne il comportamento, correggere il codice.

- **ArmSim** simula un processore ARM-v5:
<http://armsim.cs.uvic.ca/>
- sviluppato dalla University of Victoria (Canada):
progetto didattico di libero accesso

Esistono anche simulatori commerciali: più completi, più complessi da usare.

Simula l'esecuzione di un programma in linguaggio assembly:

- carica il programma
- controlla la correttezza sintattica
- assembra il programma in istruzioni macchina
- esegue virtualmente le istruzioni, anche **passo passo**
- mostra il contenuto di memoria e registri, anche **durante l'esecuzione.**

Essendo un simulatore permette di eseguire semplici chiamate al sistema operativo durante l'esecuzione del programma assembly.

Astrazione dell'hardware

La programmazione in assembly opera su un'astrazione del calcolatore:

- un processore
- un certo numero di registri
- la memoria principale.

La complessità dell'hardware sottostante non è direttamente visibile dall'ambiente assembly.

Componenti nascoste all'assembly

- Pipeline (processori superscalari)
- registri ombra
- memoria cache
- parte della memoria fisica
- memoria virtuale
- periferiche.

Modello di memoria

Le istruzioni assembly fanno riferimento a un modello di memoria formato da

- memoria dati
- memoria istruzioni
- registri contenuti nel data path, argomento e destinazione delle operazioni aritmetiche logiche.

Nota: la gestione di uno stack è responsabilità quasi completa del programmatore. Il modello Mic invece delegava alcune operazioni sullo stack alle micro-istruzioni del processore.

Struttura memoria ARM

- **Memoria principale**: 2^{32} locazioni della dimensione di **un byte**. Ogni locazione è individuata da un **indirizzo di 32 bit**.
- **Registri processore**: 16 registri **generici** di 32 bit (4 byte) r0, ..., r15.

Rappresentazione simbolica alternativa: i registri r13, r14, r15 possono essere indicati rispettivamente come **Stack Pointer** sp, **Link Register** lr, **Program Counter** pc.

Nota: l'assembler tradizionalmente non distingue il maiuscolo dal minuscolo (r0 = R0).

Registri ARM

- i registri r0 . . . r12 sono perfettamente analoghi per l'hardware
- il registro r13 per convenzione contiene l'indirizzo della cima dello stack (sp)
- il registro r14 contiene l'indirizzo di ritorno da una procedura (lr)
- il registro r15 contiene l'indirizzo dell'istruzione in esecuzione (pc)
- registri specializzati: **Current Program Status Register** (cpsr) contiene informazioni sullo stato del programma.

Operazioni aritmetiche

Accettano tre argomenti:

- `add r0, r2, r3` scrive sul registro `r0` la somma dei contenuti di `r2` e `r3`
- `add r0, r2, #7` pone in `r0` la somma di `r2` e 7
- `add r0, r2, #0xF` in `r0` la somma di `r2` e 15
- `sub r0, r2, r3` subtract $r0 = r2 - r3$
- `rsb r0, r2, r3` reverse subtract $r0 = r3 - r2$.

Operano su numeri interi codificati complemento a due.

In ogni istruzione il terzo argomento può essere una costante o un registro.

Operazioni aritmetiche

- `mul r0, r2, r3` multiply $r0 = r2 * r3$
`mul` non ammette argomento costante, solo registri.
Esistono altre istruzioni di moltiplicazione
- `adc r0, r2, r3` add with carry
 $r0 = r2 + r3 + c$, somma il bit di riporto `c` (`carry`) generato dall'istruzione precedente permettendo somme su 64 bit
- `sbc r0, r2, r3` subtract with carry
 $r0 = r2 - r3 + c - 1$, considera anche il bit di `carry` permettendo sottrazioni su 64 bit
- `rsc r0, r2, r3` reverse subtract with carry
 $r0 = r3 - r2 + c - 1$.

Esercizi:

Scrivere pezzi di codice ARM che inseriscano in r1 il valore delle espressioni:

- $r1 = r1 + r2 + r3$
- $r1 = r1 - r2 - 3$
- $r1 = 4 \times r2$
- $r1 = - r2$
- $r0\ r1 = r2\ r3 + r4\ r5$ (somma a 64 bit)
- $r0\ r1 = r2\ r3 - r4\ r5$ (sottrazione a 64 bit).

Operazioni logiche

Estese ai registri (sequenze di bit),

- `and r0, r2, r3` and bit a bit tra due registri
- `and r0, r2, #5` secondo argomento costante
- `orr r0, r2, r3` or
- `eor r0, r2, #5` exclusive or
- `bic r0, r2, r3` bit clear ($r0 = r2 \text{ and } (\text{not } r3)$)
- `mov r0, r2` funzione identità, move
- `mvn r0, r2` not, move negate.

Esercizio: calcolare il resto della divisione per 16.

Costanti rappresentabili

L'ultimo argomento di ogni operazione aritmetico-logica (esclusa la moltiplicazione) può essere una costante numerica scritta in

- decimale:

```
sub r0, r2, #15
```

- esadecimale:

```
sub r0, r2, #0xF.
```

Costanti numeriche — Immediate_8r

Costanti rappresentate in memoria con 8 + 4 bit:

- 8 bit di mantissa
- 4 bit rappresentano lo spostamento a sinistra **di due bit**. Il valore dell'esponente viene moltiplicato per 2: solo spostamenti pari.

Non tutte le costanti sono rappresentabili

- costanti rappresentabili (valid immediate 8_r)
0xFF 255 256 0xCC00 0x1FC00
- costanti non rappresentabili:
0x101 257 0x102 258

Istruzioni con costanti non rappresentabili generano errore.

Esercizi

Scrivere serie di istruzioni che realizzano le seguenti espressioni:

- $r1 = 57$
- $r1 = 1024$
- $r1 = 257$
- $r1 = \#0xAABBCCDD$
- $r1 = -1.$

Operazione di shift/rotate

In ogni istruzione aritmetico-logica, se l'ultimo argomento è un registro a questo si può applicare un operazione di **shift** o **rotate**:

`add r0, r1, r2, lsl #2` esegue
 $r0 = r1 + (r2 \ll 2)$.

La sequenza di bit in `r2` viene traslata di due posizioni verso sinistra **prima** di essere di essere sommata.

La costante di shift è contenuta nell'intervallo `[0..31]`.

Operazione di shift/rotate

È possibile specificare il numero di posizioni da traslare anche come contenuto di un registro:

`mov r0, r2, lsl r3.`

Solo gli 8 bit meno significativi del registro sono esaminati.

Esempio: l'esecuzione di

```
mov r0,#1
```

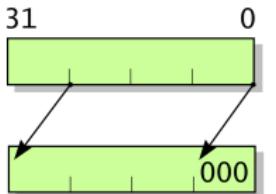
```
mov r1,#2
```

```
add r2,r0,r0, lsl r1
```

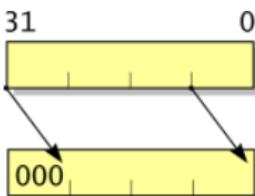
assegna a r2 il valore 5. Infatti r0 è traslato **prima** che la CPU esegua la add, la quale però a sua volta acquisisce il valore dal secondo operatore (sempre r0) **prima** di traslare il terzo.

5 tipi di shift/rotate

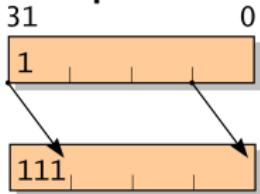
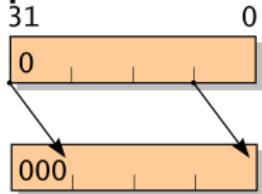
- **lsl logical shift left**



- **lsr logical shift right**

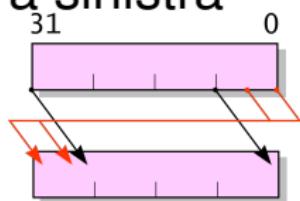


- **asr arithmetic shift right**, si inserisce a sinistra il bit di segno, esegue una divisione per una potenza di 2 in complemento a 2.

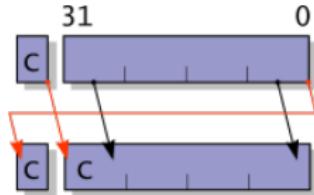


5 tipi di shift/rotate

- **ror rotate right**, i bit eliminati a destra rientrano a sinistra



- **rrx rotate right extended**, ruota a destra di una singola posizione coinvolgendo il bit di carry, non ammette argomento.



Esercizi

Scrivere le istruzioni assembly che calcolano le seguenti espressioni:

- $r1 = 8 * r2$
- $r1 = r2 / 4$
- $r1 = 5 * r2$
- $r1 = 3/4 * r2.$

Laboratorio di architettura degli elaboratori

CIRCUITI SEQUENZIALI Lezione 4

- CONTATTI
- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

Esercizio 4.1

Utilizzare il modulo del secondo punto dell'esercizio 3.2.2 per costruire un contatore binario ciclico a 4 cifre in forma di circuito sequenziale senza ingressi e con 4 uscite che, ad ogni ciclo di clock, incrementa il valore binario in uscita di una unità. Il contatore riparte da zero dopo avere raggiunto il valore massimo.

Esercizio 4.2

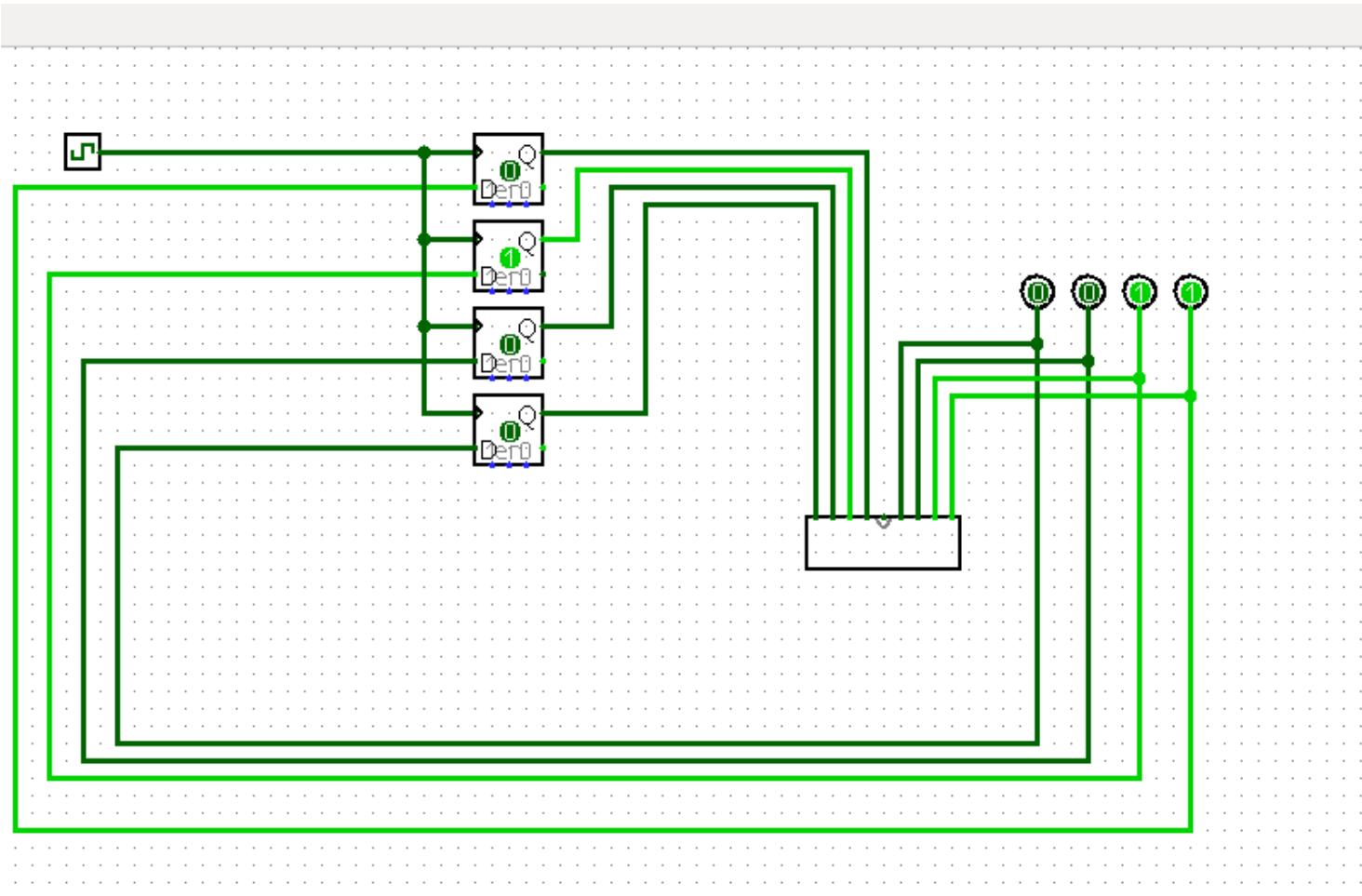
Modificare il circuito in modo che siano presenti due ingressi di controllo:

- un segnale **S** che, se è pari a 1, blocca il contatore nella posizione corrente. Se **S** viene successivamente posto a 0 il circuito deve ricominciare a contare
- un segnale **R** che, se è pari a 1, imposta il contatore al numero 0011. Se **R** viene successivamente posto a 0 il circuito deve ricominciare a contare partendo da 0011.

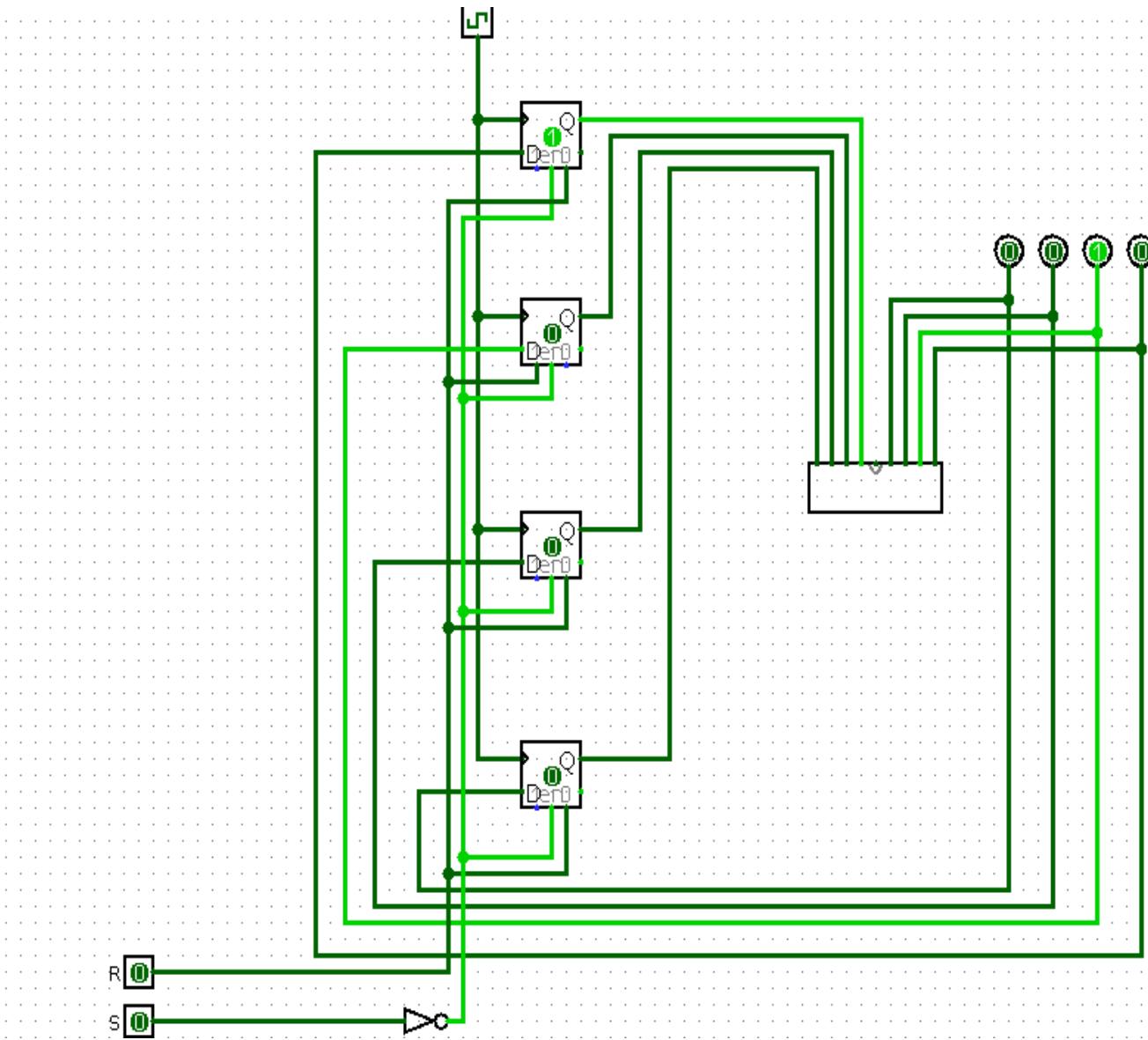
Esercizio 4.3

Modificare il circuito in modo che, una volta giunto al numero 13, il contatore riparta a contare dal numero 3 (0011).

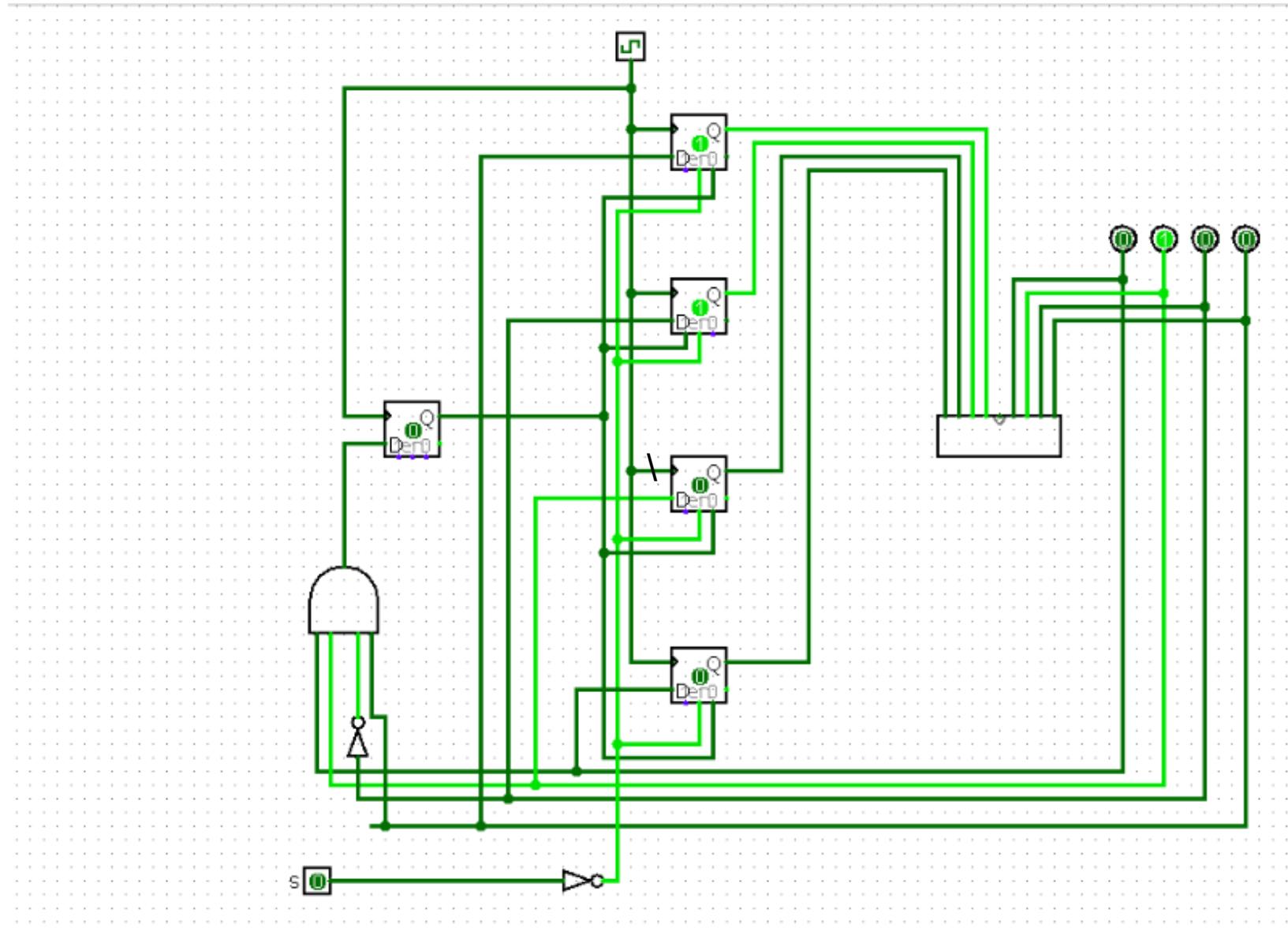
Esercizio 4.1



Esercizio 4.2



Esercizio 4.3



Laboratorio di architettura degli elaboratori

CIRCUITI SEQUENZIALI Lezione 4

- CONTATTI
- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

Esercizio 4.1

Utilizzare il modulo del secondo punto dell'esercizio 3.2.2 per costruire un contatore binario ciclico a 4 cifre in forma di circuito sequenziale senza ingressi e con 4 uscite che, ad ogni ciclo di clock, incrementa il valore binario in uscita di una unità. Il contatore riparte da zero dopo avere raggiunto il valore massimo.

Esercizio 4.2

Modificare il circuito in modo che siano presenti due ingressi di controllo:

- un segnale **S** che, se è pari a 1, blocca il contatore nella posizione corrente. Se **S** viene successivamente posto a 0 il circuito deve ricominciare a contare
- un segnale **R** che, se è pari a 1, imposta il contatore al numero 0011. Se **R** viene successivamente posto a 0 il circuito deve ricominciare a contare partendo da 0011.

Esercizio 4.3

Modificare il circuito in modo che, una volta giunto al numero 13, il contatore riparta a contare dal numero 3 (0011).

Livello ISA

Instruction Set Architecture: insieme delle istruzioni eseguibili dal processore (istruzioni macchina).

Presentazione generale dei linguaggi macchina.

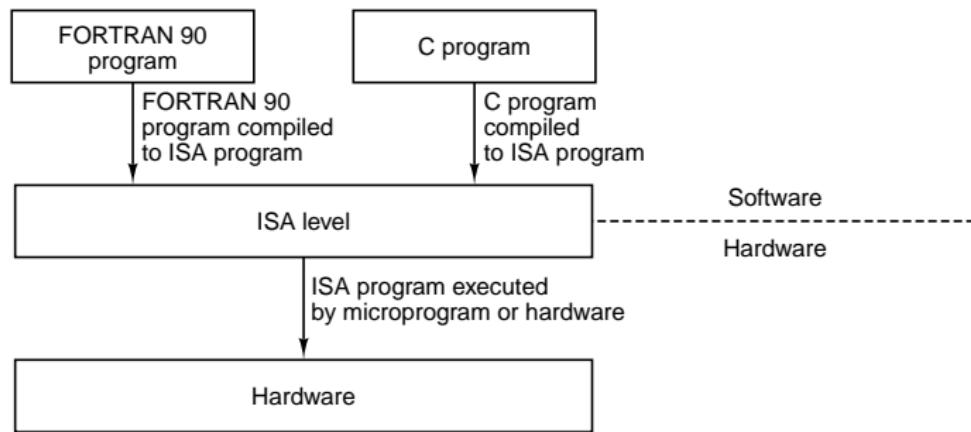
- Elenco degli aspetti salienti di un linguaggio macchina.
- Confronto tra linguaggi macchina.

Software – Hardware

ISA definisce l'interfaccia tra hardware e software.

Ogni codice dev'essere tradotto in istruzioni macchina.

L'hardware esegue istruzioni macchina.



Progettazione di un ISA

Approcci:

- CISC – linguaggio macchina simile ai linguaggi alto livello
- RISC – set di istruzioni semplice
- VLIW (**Very Long Instruction Word**) – sfruttare il parallelismo del processore.

Il livello di complessità di un ISA riflette un compromesso tra esigenze di

- progettisti hardware
- programmatore (soprattutto di sw per la compilazione).

Progettisti hardware

Requisiti sulle istruzioni:

- facilmente implementabili
- sfruttano le potenzialità della tecnologia
- ne evitano i difetti.

Esvti:

- solo operazioni aritmetico-logiche (nessun calcolo funzionale)
- uso dei registri per limitare l'accesso alla memoria
- istruzioni vettoriali (SSE, VLIW)
- istruzioni di salto condizionato, speculative,

Programmatori, compilatori

Requisiti sulle istruzioni:

- disponibilità di istruzioni fondamentali e anche utili
- comportamento chiaro e prevedibile (**semantica non ambigua**)
- efficienza (le istruzioni più ricorrenti nel codice vanno implementate in maniera più efficiente).

Compatibilità

Retrocompatibilità con software in linguaggio macchina di cui non è più disponibile il codice sorgente.

IA-32 (Intel Architecture a 32 bit) della famiglia x86:
Core i7 (64 bit) -> ... -> 80386 (32 bit) -> ... ->
80286 (16 bit) -> 8088 -> 8086, il quale a sua volta aveva parziale retrocompatibilità con 8080 (8 bit) -> 8008 -> 4004 (4 bit!)

Core i7 può (**deve**) emulare la CPU 8088!
Es.: esecuzione programmi MS-DOS da Windows 7.

Eccezioni alla compatibilità

- Apple: Motorola 68000 — PowerPc — IA-32
- Sun: Motorola 68000 — Sparc
- Intel: Pentium-Core (IA-32) — Itanium (IA-64)
- Video giochi, sistemi embedded.
Sony Play Station – PS1, PS2: MIPS, PS3: Cell,
PS4: x86.

Flessibilità

Un ISA può durare decenni.

Deve adattarsi allo sviluppo tecnologico.

Nella definizione di un ISA bisogna favorire e semplificare le future estensioni.

In futuro è probabile un aumento di

- spazio di memoria indirizzabile,
- numero di registri interni,
- lunghezza della parola,
- insieme di istruzioni eseguibili.

Caratteristiche dei linguaggi macchina

- specifica
- istruzioni
- tipi di dato
- modello memoria
- formato istruzioni
- indirizzamento
- modalità di funzionamento
- I/O.

Specifica

Definire il comportamento del processore:

- **formalmente**: SPARC V9, ARM
informative (descrizione intuitiva)
normative (descrizione formale e rigorosa)
alcune scelte lasciate all'implementazione
- **informalmente**: IA32 (Pentium-Core)
non sono state pubblicate delle specifiche formali.

Differenza dovuta a scelte commerciali.

Tipi di istruzioni

Quasi tutte rientrano in una di queste categorie:

- **movimento dati**: memoria e/o registri
LOAD reg add, STORE reg add, MOVE reg reg
- **operazioni**: aritmetiche, logiche, rotazioni
ADD reg reg reg, NOT reg reg,
SHIFT_LEFT reg reg n,
(in diversi ISA varia il numero di argomenti)
- **salto**: istruzioni condizionate e non
GOTO label, BRANCH label,
BRANCH_EQZ reg label
- **chiamate di procedura, al sistema operativo**
JUMPnLINK, INVOKE, SYSCALL.

Tipi di dati hardware

Numerici

- naturali: unsigned 8, 16, 32, 64 bit
- interi: signed 8, 16, 32, 64 bit
- virgola mobile: 32, 64, 128 bit
- BCD: binary code decimal format.

Non numerici

- caratteri, booleani
- riferimento (indirizzi di memoria)
- stringhe, array (sequenze di dati).

Tipi di dati: Intel e ARM

Intel

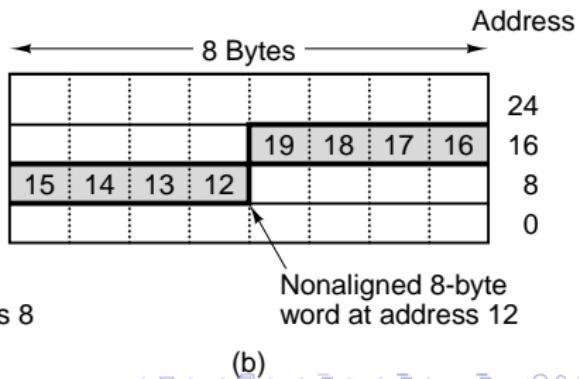
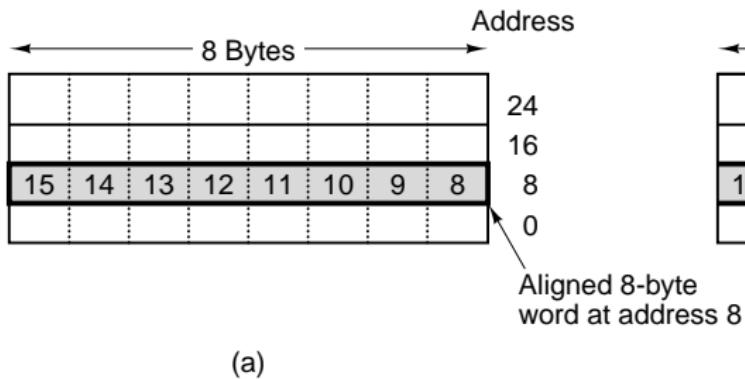
Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×	×	×	× (64-bit)
Unsigned integer	×	×	×	× (64-bit)
Binary coded decimal integer	×			
Floating point			×	×

ARM

Type	8 Bits	16 Bits	32 Bits	64 Bits
Signed integer	×	×	×	
Unsigned integer	×	×	×	
Binary coded decimal integer				
Floating point			×	×

Modello di memoria

- Spazio di memoria: unico oppure suddiviso in istruzioni e dati
- Lunghezza dell'unità di memoria: bit, byte, parola (word)
- Alcuni processori accedono solo alle parole se correttamente allineate:



Modello di memoria

Semantica della memoria

- **stretta**: accessi in memoria nell'ordine con cui sono scritti nel codice.
Semantica semplice, agevola il programmatore
- **lasca** (loose): permette accessi **fuori ordine** alla memoria, migliorando il parallelismo nel processore.
Semantica complessa, è necessario stabilire regole e meccanismi di sincronizzazione.

Registri

Riducono gli accessi alla memoria.

- Registri **generici**: memorizzano variabili temporanee.
- Registri **specializzati**: realizzano funzioni specifiche.

Nell'ISA IA-32 tutti i registri sono etichettati con un funzione specifica.

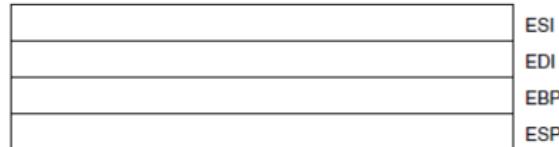
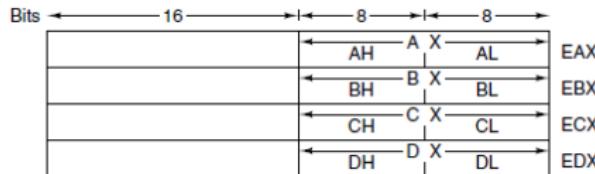
In tutti i processori troviamo:

- **PC**: program counter;
- **IR**: instruction register;
- **PSW**: program status word,

Describe lo stato del programma: codici di condizione, modalità macchina, livello di priorità, abilitazione degli interrupt.

Registri interni x86

Solo a titolo informativo.



Registri interni ARM

ARM è un'architettura **load/store**: l'aritmetico-logica avviene **solo** tra registri.

Register	Alt. name	Function
R0–R3	A1–A4	Holds parameters to the procedure being called
R4–R11	V1–V8	Holds local variables for the current procedure
R12	IP	Intraprocedure call register (for 32-bit calls)
R13	SP	Stack pointer
R14	LR	Link register (return address for current function)
R15	PC	Program counter

In più ci sono 32 registri dedicati all'aritmetica in virgola mobile.

Formato delle istruzioni

Rappresentazione delle istruzioni nel calcolatore.

Assembly è il linguaggio di programmazione più vicino a quello macchina.

Es.: per sommare due valori in ARM si deve **assemblare** l'istruzione ADD:

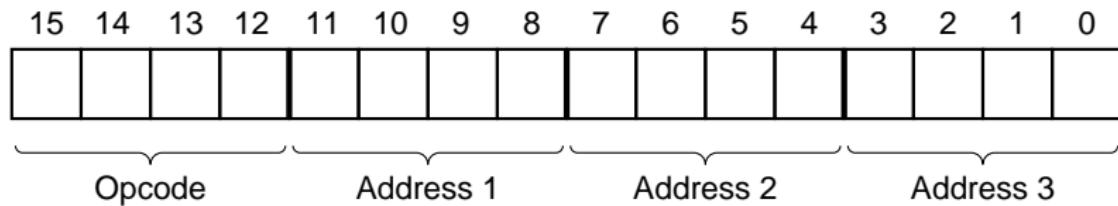
ADDS R2, R3, #4

La codifica in linguaggio macchina deve conservare

- l'istruzione: ADD
- gli argomenti su cui opera: r2 ove depositare la somma, r3 contenente il primo termine, il secondo termine costante 4
- eventuali altre indicazioni: S.

Formato delle istruzioni

Codice operativo espandibile



- semplifica la decodifica da parte del processore
- riduce la lunghezza delle istruzioni
- facilita l'aggiunta di nuove istruzioni.

Esempio: Sun SPARC

Si usano formati diversi a seconda del numero di argomenti dell'istruzione:

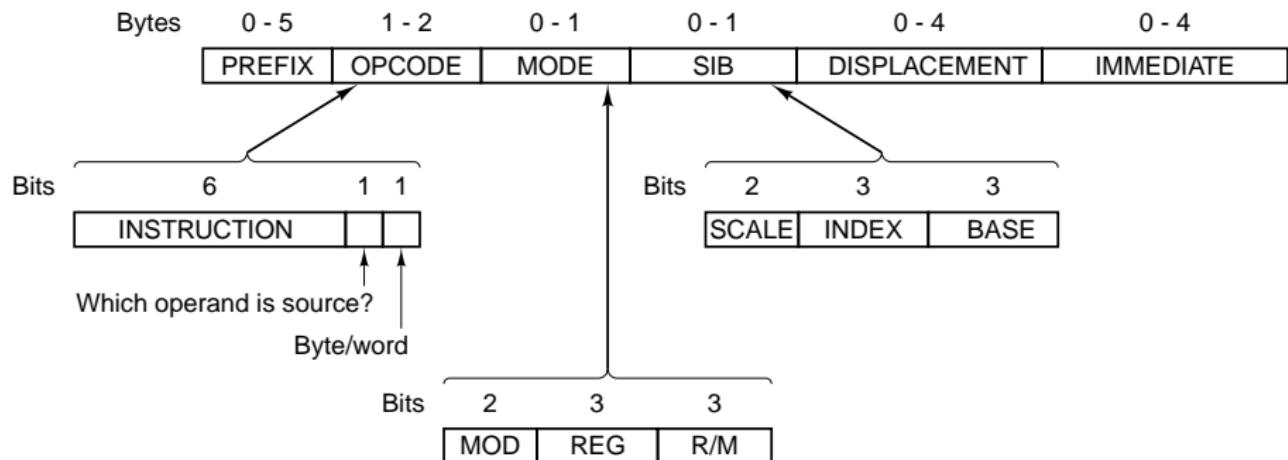
Format	1a	1b	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500	501	502	503	504	505	506	507	508	509	510	511	512	513	514	515	516	517	518	519	520	521	522	523	524	525	526	527	528	529	530	531	532	533	534	535	536	537	538	539	540	541	542	543	544	545	546	547	548	549	550	551	552	553	554	555	556	557	558	559	560	561	562	563	564	565	566	567	568	569	570	571	572	573	574	575	576	577	578	579	580	581	582	583	584	585	586	587	588	589	590	591	592	593	594	595	596	597	598	599	600	601	602	603	604	605	606	607	608	609	610	611	612	613	614	615	616	617	618	619	620	621	622	623	624	625	626	627	628	629	630	631	632	633	634	635	636	637	638	639	640	641	642	643	644	645	646	647	648	649	650	651	652	653	654	655	656	657	658	659	660	661	662	663	664	665	666	667	668	669	670	671	672	673	674	675	676	677	678	679	680	681	682	683	684	685	686	687	688	689	690	691	692	693	694	695	696	697	698	699	700	701	702	703	704	705	706	707	708	709	710	711	712	713	714	715	716	717	718	719	720	721	722	723	724	725	726	727	728	729	730	731	732	733	734	735	736	737	738	739	740	741	742	743	744	745	746	747	748	749	750	751	752	753	754	755	756	757	758	759	760	761	762	763	764	765	766	767	768	769	770	771	772	773	774	775	776	777	778	779	770	771	772	773	774	775	776	777	778	779	780	781	782	783	784	785	786	787	788	789	790	791	792	793	794	795	796	797	798	799	800	801	802	803	804	805	806	807	808	809	8010	8011	8012	8013	8014	8015	8016	8017	8018	8019	8020	8021	8022	8023	8024	8025	8026	8027	8028	8029	8030	8031	8032	8033	8034	8035	8036	8037	8038	8039	8040	8041	8042	8043	8044	8045	8046	8047	8048	8049	8050	8051	8052	8053	8054	8055	8056	8057	8058	8059	8060	8061	8062	8063	8064	8065	8066	8067	8068	8069	8070	8071	8072	8073	8074	8075	8076	8077	8078	8079	8080	8081	8082	8083	8084	8085	8086	8087	8088	8089	80810	80811	80812	80813	80814	80815	80816	80817	80818	80819	80820	80821	80822	80823	80824	80825	80826	80827	80828	80829	80830	80831	80832	80833	80834	80835	80836	80837	80838	80839	80840	80841	80842	80843	80844	80845	80846	80847	80848	80849	80850	80851	80852	80853	80854	80855	80856	80857	80858	80859	80860	80861	80862	80863	80864	80865	80866	80867	80868	80869	80870	80871	80872	80873	80874	80875	80876	80877	80878	80879	80880	80881	80882	80883	80884	80885	80886	80887	80888	80889	80890	80891	80892	80893	80894	80895	80896	80897	80898	80899	808100	808101	808102	808103	808104	808105	808106	808107	808108	808109	808110	808111	808112	808113	808114	808115	808116	808117	808118	808119	808120	808121	808122	808123	808124	808125	808126	808127	808128	808129	808130	808131	808132	808133	808134	808135	808136	808137	808138	808139	808140	808141	808142	808143	808144	808145	808146	808147	808148	808149	808150	808151	808152	808153	808154	808155	808156	808157	808158	808159	808160	808161	808162	808163	808164	808165	808166	808167	808168	808169	808170	808171	808172	808173	808174	808175	808176	808177	808178	808179	808180	808181	808182	808183	808184	808185	808186	808187	808188	808189	808190	808191	808192	808193	808194	808195	808196	808197	808198	808199	808200	808201	808202	808203	808204	808205	808206	808207	808208	808209	808210	808211	808212	808213	808214	808215	808216	808217	808218	808219	808220	808221	808222	808223	808224	808225	808226	808227	808228	808229	808230	808231	808232	808233	808234	808235	808236	808237	808238	808239	808240	808241	808242	808243	808244	808245	808246	808247	808248	808249	808250	808251	808252	808253	808254	808255	808256	808257	808258	808259	808260	808261	808262	808263	808264	808265	808266	808267	808268	808269	808270	808271	808272	808273	808274	808275	808276	808277	808278	808279	808280	808281	808282	808283	808284	808285	808286	808287	808288	808289	808290	808291	808292	808293	808294	808295	808296	808297	808298	808299	808300	808301	808302	808303	808304	808305	808306	808307	808308	808309	808310	808311	808312	808313	808314	808315	808316	808317	808318	808319	808320	808321	808322	808323	808324	808325	808326	808327	808328	808329	808330	808331	808332	808333	808334	808335	808336	808337	808338	808339	808340	808341	808342	808343	808344	808345	808346	808347	808348	808349	808350	808351	808352	808353	808354	808355	808356	808357	808358	808359	808360	808361	808362	808363	808364	808365	808366	808367	808368	808369	808370	808371	808372	808373	808374	808375	808376	808377	808378	808379	808380	808381	808382	808383	808384	808385	808386	808387	808388	808389	808390	808391	808392	808393	808394	808395	808396	808397	808398	808399	808400	808401	808402	808403	808404	808405	808406	808407	808408	808409	808410	808411	808412	808413	808414	808415	808416	808417	808418	808419	808420	808421	808422	808423	808424	808425	808426	808427	808428	808429	808430	808431	808432	808433	808434	808435	808436	808437	808438	808439	808440	808441	808442	808443	808444	808445	808446	808447	808448	808449	808450	808451	808452	808453	808454	808455	808456	808457	808458	808459	808460	808461	808462	808463	808464	808465	808466	808467	808468	808469	808470	808471	808472	808473	808474	808475	808476	808477	808478	808479</

Esempio: ARM

31	2827	1615	87	0	Instruction type
Cond	0 0	Opcode S	Rn	Rd	Operand2
Cond	0 0 0 0 0 0	A S	Rd	Rn	RS 1 0 0 1 Rm
Cond	0 0 0 0 1	U A S	RdHi	RdLo	RS 1 0 0 1 Rm
Cond	0 0 0 1 0	B 0 0	Rn	Rd	0 0 0 0 1 0 0 1 Rm
Cond	0 1 P U B W L		Rn	Rd	Offset
Cond	1 0 0 P U S W L		Rn		Register List
Cond	0 0 0 P U 1 W L		Rn	Rd	Offset1 1 S H 1 Offset2
Cond	0 0 0 P U 0 W L		Rn	Rd	0 0 0 0 1 S H 1 Rm
Cond	1 0 1 L				Offset
Cond	0 0 0 1	0 0 1 0	1 1 1 1	1 1 1 1	1 1 1 1 0 0 0 1 Rn
Cond	1 1 0 P U N W L		Rn	CRd	CPNum Offset
Cond	1 1 1 0	Op1	CRn	CRd	CPNum Op2 0 CRm
Cond	1 1 1 0	Op1 L	CRn	Rd	CPNum Op2 1 CRm
Cond	1 1 1 1				SWI Number

Esempio: x86

Solo a titolo informativo.



Argomenti di un'istruzione

A seconda del linguaggio macchina una stessa istruzione può avere un numero variabile di argomenti.

Per esempio, la somma in

- ARM, architetture RISC: **tre** argomenti
- IA-32: **due** argomenti
- architetture con accumulatore: **un** argomento
- Java bytecode: **nessun** argomento (Mic-1).

Indirizzamento

Modi per specificare gli argomenti di un'istruzione

- **immediato**: il valore esplicito – #5
- **registro**: un registro – \$r1
- **diretto**: un indirizzo di memoria – [0x1000]
- **indiretto tramite registro**: un registro che contiene un indirizzo di memoria – [\$r1]
- **indice**: offset + displacement – [\$r1,100]
- **base indicizzata**: displacement contenuto in un registro – [\$r1,\$r2].

Indirizzamento

Addressing mode	Core i7	OMAP4430 ARM	ATmega168 AVR
Immediate	×	×	×
Direct	×		×
Register	×	×	×
Register indirect	×	×	×
Indexed	×	×	
Based-indexed		×	

Indirizzamento

Indipendenza tra indirizzamento e istruzione.

Istruzioni analoghe usano le stesse modalità di indirizzamento. Es.: tutte le operazioni aritmetiche specificano gli argomenti nello stesso modo.

I processori RISC sono architetture load/store:

- operazioni aritmetiche-logiche solo con indirizzamento immediato o registri
- accesso ai dati in memoria solo con istruzioni di caricamento (load) e immagazzinamento (store) da/verso la memoria.

Modalità di funzionamento

Permettono meccanismi di protezione: si evita che un programma usi risorse non proprie

- **kernel** del sistema operativo: può eseguire qualsiasi istruzione. Es.: chiamate di sistema
- **utente**: privilegi limitati delle applicazioni utente.

Passaggio controllato dalla modalità utente alla modalità **supervisore**.

Es.: l'istruzione `syscall` è eseguita al sussistere di determinati privilegi.

Modalità di funzionamento utile anche per garantire la **retrocompatibilità**. Es. (x86): Core i7 -> 8088.

Modalità di funzionamento - Core i7

- **reale**: isa 8088
- **virtuale**: isa 8088, blocchi delegati al **S.O.**
- **protetta**: isa x86-64
 - livello 0: kernel
 - livello 1: S.O.
 - livello 2: librerie
 - livello 3: utente.

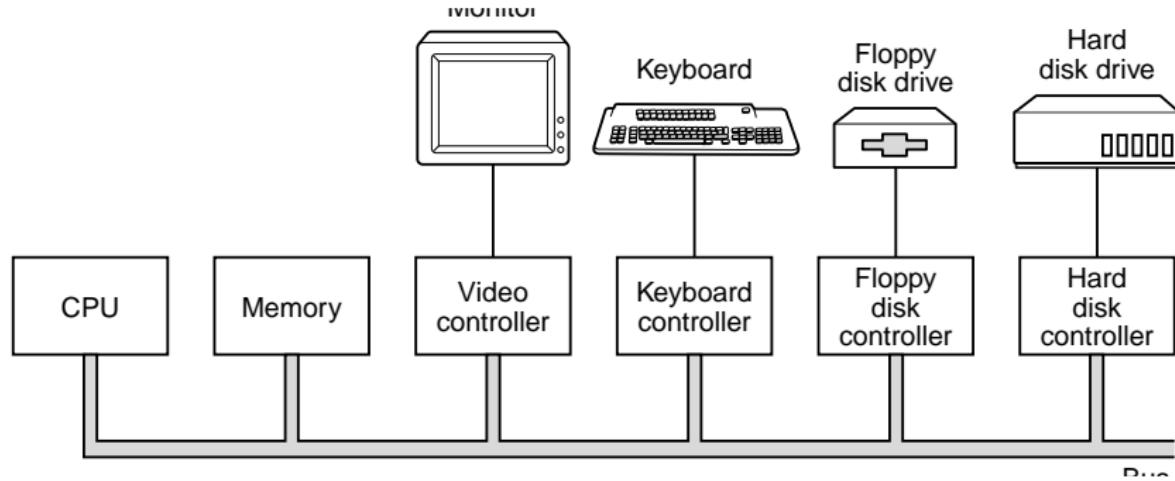
Modalità di funzionamento - ARM

Non occorre impararle a memoria.

Mode	Description
User	Normal program execution, no privileges
System	Runs programs with some privileges
FIQ	Fast interrupt handling
IRQ	Normal interrupt handling
Supervisor	Privileged mode for the operating system
Abort	For memory protection
Undefined	Facilitates emulation of co-processors.

Input/Output (I/O)

Meccanismi e istruzioni con cui la CPU comunica con le periferiche e la memoria di massa attraverso il bus.



Circuiti di controllo

La CPU comunica con **dispositivi di controllo I/O**: moduli (I/O channels, I/O processor) quali schede video, audio, di rete. Questi moduli

- risiedono nel calcolatore (interfacciano la CPU con le periferiche)
- trasformano
 - comandi della CPU in segnali elettrici di comando per le periferiche,
 - i segnali dalle periferiche in dati per la CPU
- permettono alla CPU di interracciarsi in maniera uniforme con dispositivi diversi tra loro.

Comunicazione CPU – controllori:

Attraverso dei registri **interni ai controllori**.

Indirizzamento. Due alternative:

- **Memory mapped I/O**: registri identificati come indirizzi della memoria principale.
Accesso uniforme. Si usano normali istruzioni di accesso alla memoria principale (load, store).
Sottraggono spazio di memoria indirizzabile.
- **Isolated I/O**: indirizzi e istruzioni dedicate.

La comunicazione con i controllori è comunque simile a quella con la memoria principale.

Esempio di Memory Mapped I/O

Si divide lo spazio di indirizzamento in settori, ciascuno dedicato a un diverso controllore.

Dispositivo	Address range	Size
RAM I/O	0000 - 7FFF	32 KB
General purpose I/O	8000 - 80FF	256 bytes
Sound controller	9000 - 90FF	256 bytes
Video controller	A000 - A7FF	2 KB
ROM I/O	C000 - FFFF	16 KB

A livello hardware, un circuito combinatorio aziona il controllore appropriato.

Comunicazione con i controllori

Si scambiano

- dati I/O
- comandi (dalla CPU alla periferica)
- informazioni di stato dalla periferica.

Comunicazioni diverse usano registri diversi.

Es.: stampante

- dati da stampare
- comandi di stampa
- informazioni sull'avanzamento del lavoro, stato della stampante (mancanza di carta, livello inchiostro).

Meccanismi di sincronizzazione

Necessari perché CPU e controllori funzionano in modo indipendente (compiti e velocità differenti). La CPU può inviare i comandi più velocemente di quanto la periferica riesca ad eseguirli.

Tre meccanismi principali:

- I/O programmato attraverso busy waiting
- I/O controllato da interrupt
- Direct Memory Access.

I/O programmato

Busy waiting (o **polling**).

La CPU controlla periodicamente lo stato di avanzamento e lo stato della periferica.

- Semplice: non richiede hardware dedicato.
- Inefficiente: spreca risorse di CPU.

Usato in architetture semplici, sistemi embedded, microcontrollori.

Esempio di busy waiting

Controllore stampante con registro di stato (con bit READY) e registro del carattere da stampare.

- CPU: trova/attende la stampante pronta ($\text{READY}=1$);
- CPU: inserisce carattere nel registro e dà il comando di stampa ($\text{READY}=0$);
- CPU: inizia a leggere ciclicamente il bit READY;
- controllore: riceve il comando e lo esegue;
- controllore: segnala l'esecuzione del comando ($\text{READY}=1$).

I/O controllato

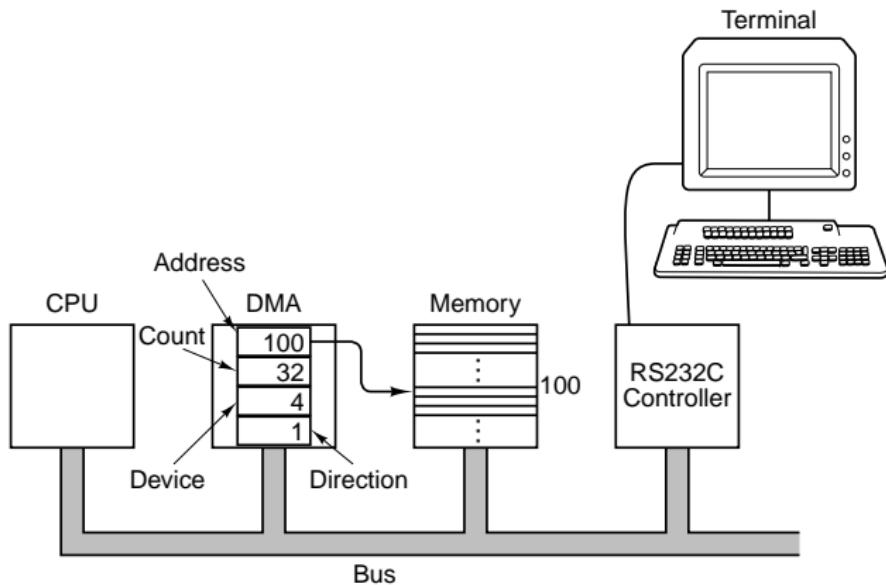
Interrupt: “campanello” con cui i controllori richiamano l’attenzione della CPU.

- La CPU dà il comando di I/O e poi esegue altri processi
- il controllore esegue il comando e quindi invia un interrupt
- ricevuto l’interrupt, la CPU può decidere di sospendere il processo in esecuzione e gestire l’I/O.

Migliora notevolmente le prestazioni, ma non risolve ancora la gestione di ogni dato I/O da parte della CPU che di conseguenza riceve continue richieste d’interruzione.

Direct Memory Access

La gestione dell'I/O viene in parte **delegata** a un dispositivo che ha accesso diretto alla memoria.



Direct Memory Access

- La CPU autorizza l'interfaccia DMA a svolgere l'operazione:
 - controllore di I/O coinvolto
 - indirizzo memoria dove prelevare - depositare i dati
 - quantità di dati da trasferire
- il dispositivo DMA gestisce il trasferimento memoria - controllore.

DMA: vantaggi e svantaggi

- Presenza di un chip DMA aggiuntivo
- minor lavoro per la CPU, meno interrupt
- potenziali conflitti per l'uso della memoria ([cycle stealing](#))
- il controllore DMA può gestire una o più periferiche
- esistono controllori DMA sofisticati dotati di un proprio codice macchina (co-processor I/O).

Eccezione (Trap)

Forza la CPU a gestire un evento **generato dalla CPU stessa** in presenza di una situazione anomala.

- Interrupt: generato da una periferica, **asincrono** (la periferica richiede l'attenzione della CPU).
- Trap: generata dal codice, **sincrona** (la CPU si isola per risolvere la situazione).

Casi tipici: overflow, opcode inesistente, indirizzo di memoria errato, page-fault.

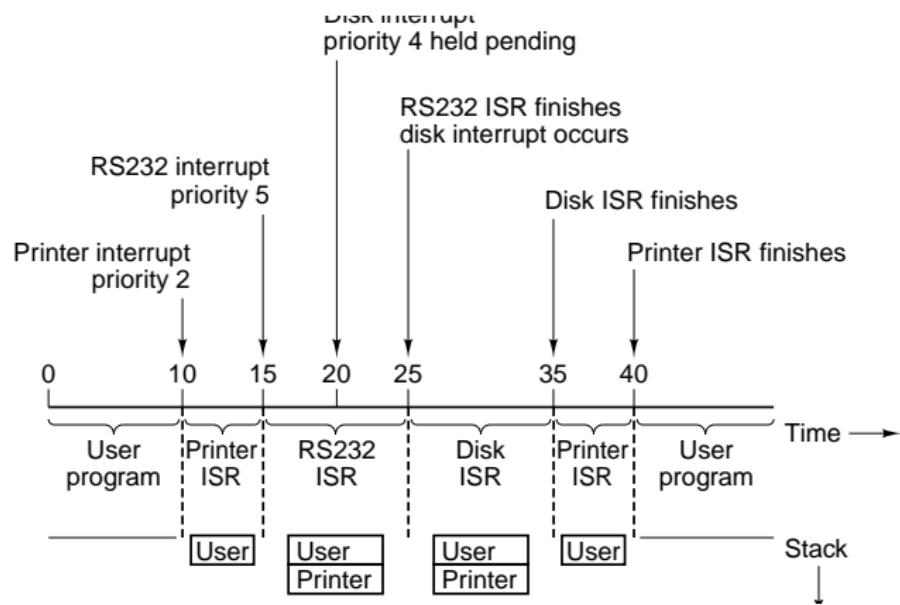
Vantaggi: le eccezioni permettono di delegare alla CPU la gestione degli errori software (si evita di dover inserire codice di controllo).

Gestione di interrupt (e trap)

- Il dispositivo genera un segnale di interrupt
- CPU rileva il segnale, identifica il dispositivo, richiede il **vettore dell'interrupt**
- il vettore dell'interrupt contiene tra l'altro l'indice per aggiornare PC dalla **tabella degli interrupt**
- si salva PC, PSW, **mascheramento** di nuovi interrupt, cambio modalità
- esecuzione della procedura (salvataggio dello stato, identificazione del dispositivo, disamina dello stato interno, . . .)
- ripristino programma sospeso e modalità.

Mascheramento dell'interrupt

Priorità nel servire le eccezioni: situazioni che possono attendere di meno hanno priorità più alta.



Precisione dell'interrupt

Per la gestione dell'interrupt la CPU deve fermarsi in istanti precisi dell'esecuzione.

A causa del parallelismo (es.: processori superscalari), questo vincolo è costoso:

- la CPU deve scegliere l'istruzione su cui interrompersi;
- completare le istruzioni precedenti;
- smettere di caricare le istruzioni successive, scaricare quelle eventualmente iniziate.

Precisione dell'interrupt:

ISA recenti hanno proposto interrupt imprecisi: non viene definita l'istruzione precisa su cui il programma può interrompersi:

- permettono implementazioni più efficienti
- complicano la vita al programmatore.

Un altro esempio di contrapposizione tra progettazione hardware e scrittura software.

Laboratorio di architettura degli elaboratori

CIRCUITI SEQUENZIALI Lezione 5

- CONTATTI
- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

Esercizio 5.1

Costruire un circuito sequenziale con un segnale di ingresso e un segnale di uscita che riconosca la stringa 1101; ossia l'uscita del circuito assume il valore 1 quando l'ingresso attuale assieme a quelli nei 3 cicli di clock precedenti forma, nell'ordine temporale, la sequenza 1101; l'uscita assume il valore 0 altrimenti.

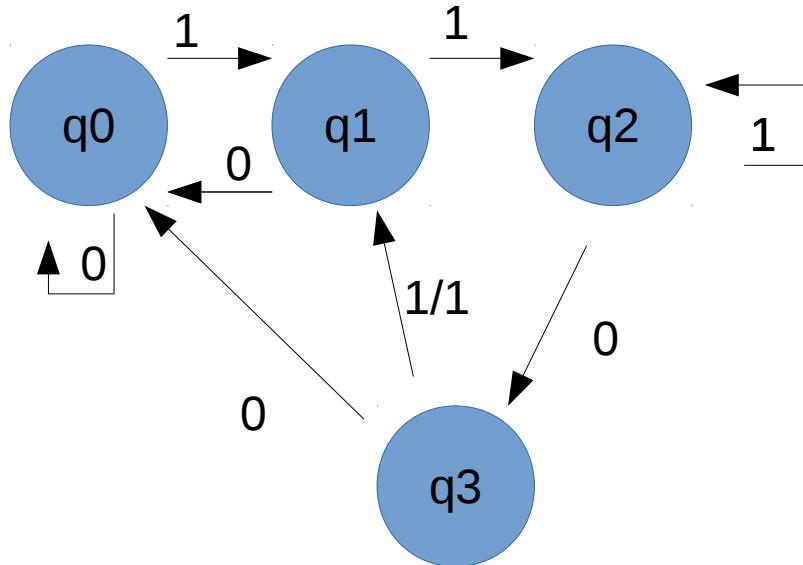
Esercizio 5.2

Costruire un circuito che simuli il funzionamento di un ascensore a due piani. Il circuito ha due segnali di ingresso che simulano i pulsanti di chiamata al piano, ha due segnali di uscita che segnalano la presenza dell'ascensore al piano. Si supponga inoltre che ascensore impieghi tre cicli di clock per passare da un piano all'altro.

Esercizio 5.3

Usando il circuito sommatore a 4 bit dell'esercizio 3.3.2, un registro e poche altre porte logiche (nonchè i concetti dell'aritmetica in complemento a 2), costruire un contatore up/down a 4 bit. Il circuito possiede due ingressi **e** ed **u**, e quattro uscite. L'ingresso **e** abilita il conteggio, il circuito non modifica l'uscita se l'ingresso **e** è a 0 mentre conta se l'ingresso è a 1. L'ingresso **u** determina il verso del conteggio: quando l'ingresso **u** è a 1, il contatore incrementa ad ogni ciclo di clock il valore dell'uscita; quando l'ingresso **u** è a 0, il valore dell'uscita viene decrementato

Esercizio 5.1



S1	S0	I	S1*	S0*	E
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	1	1	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	1	0	1	1

$S1^*$

	S1	S0	
	0	0	0
	0	1	1

$S0^*$

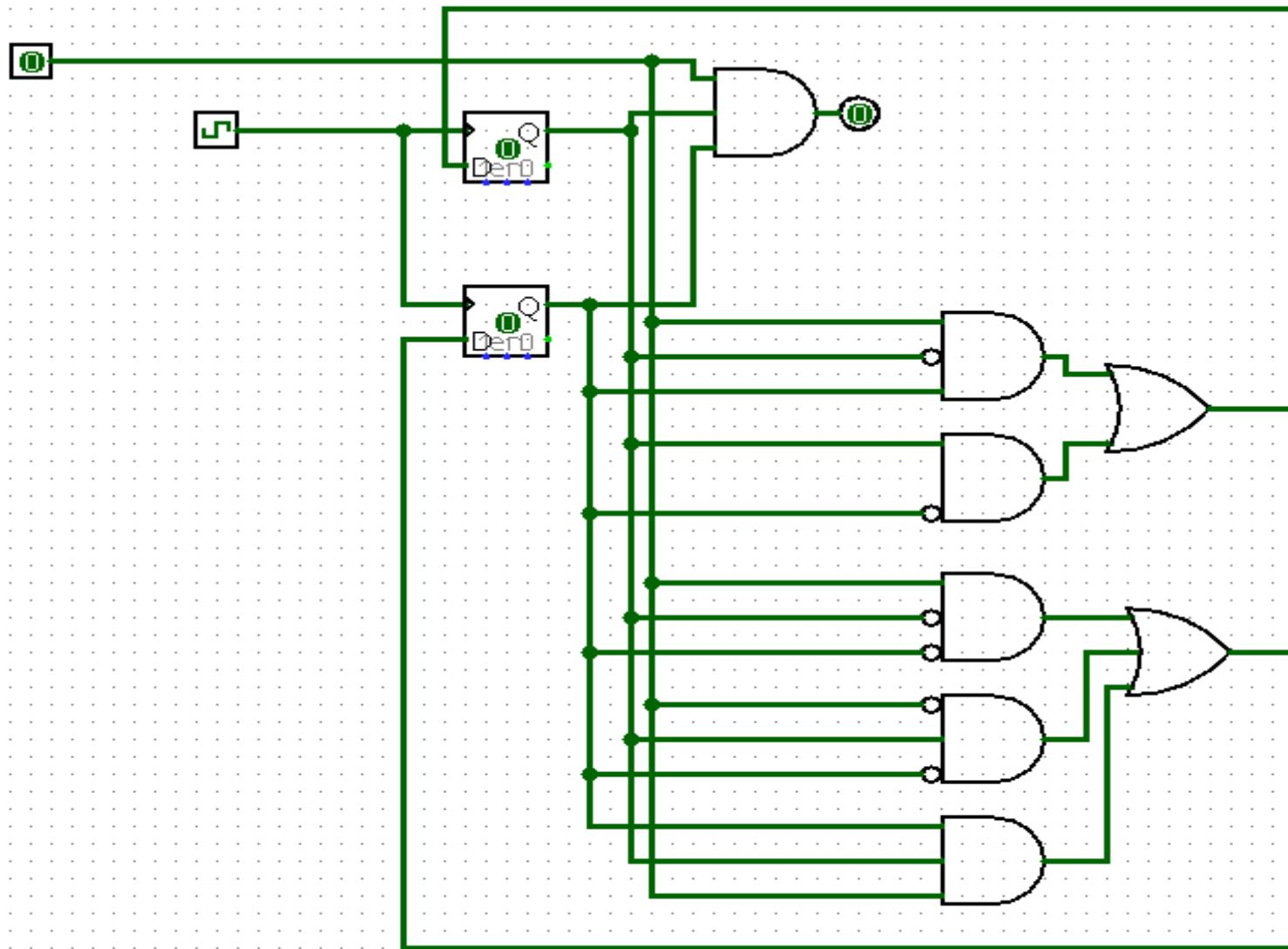
	S1	S0	
	0	0	0
	1	0	1

$$E = S1 * S0 * I$$

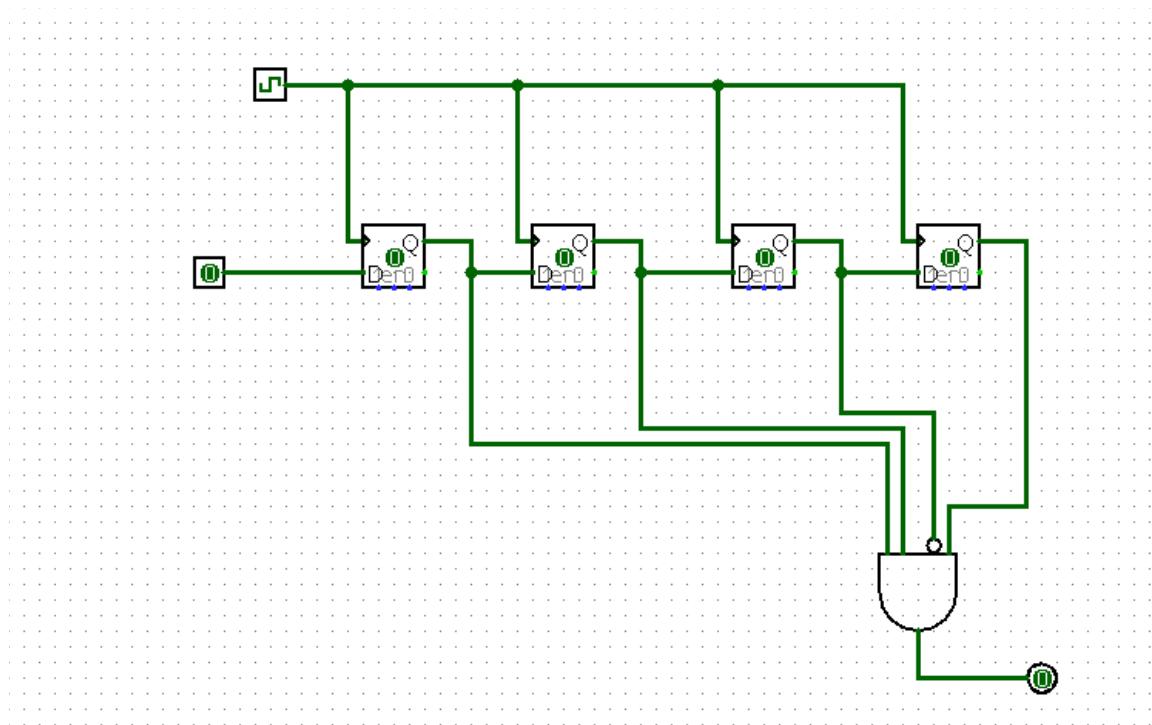
$$S1^* = S1^\wedge * S0 * I + S1 * S0^\wedge$$

$$S0^* = (S1^\wedge * S0^\wedge * I) + (S1 * S0^\wedge * I^\wedge) + (S1 * S0 * I)$$

Esercizio 5.1



Esercizio 5.1 (Alternativa)



Esercizio 5.2

S2	S1	S0	I1	I0	S2*	S1*	S0*	E1	E0
1	1	1	0	0	1	1	1	1	0
0	0	0	0	0	0	0	0	0	1
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	1	0	0
0	0	1	1	0	0	1	1	0	0
0	1	1	1	0	1	1	1	1	0
1	1	1	1	0	0	0	0	1	0
*	*	*	1	1	*	*	*	0	1
*	*	*	0	1	*	*	*	0	1

$$E0 = S2^{*\wedge} \times S1^{*\wedge} \times S0^{*\wedge}$$

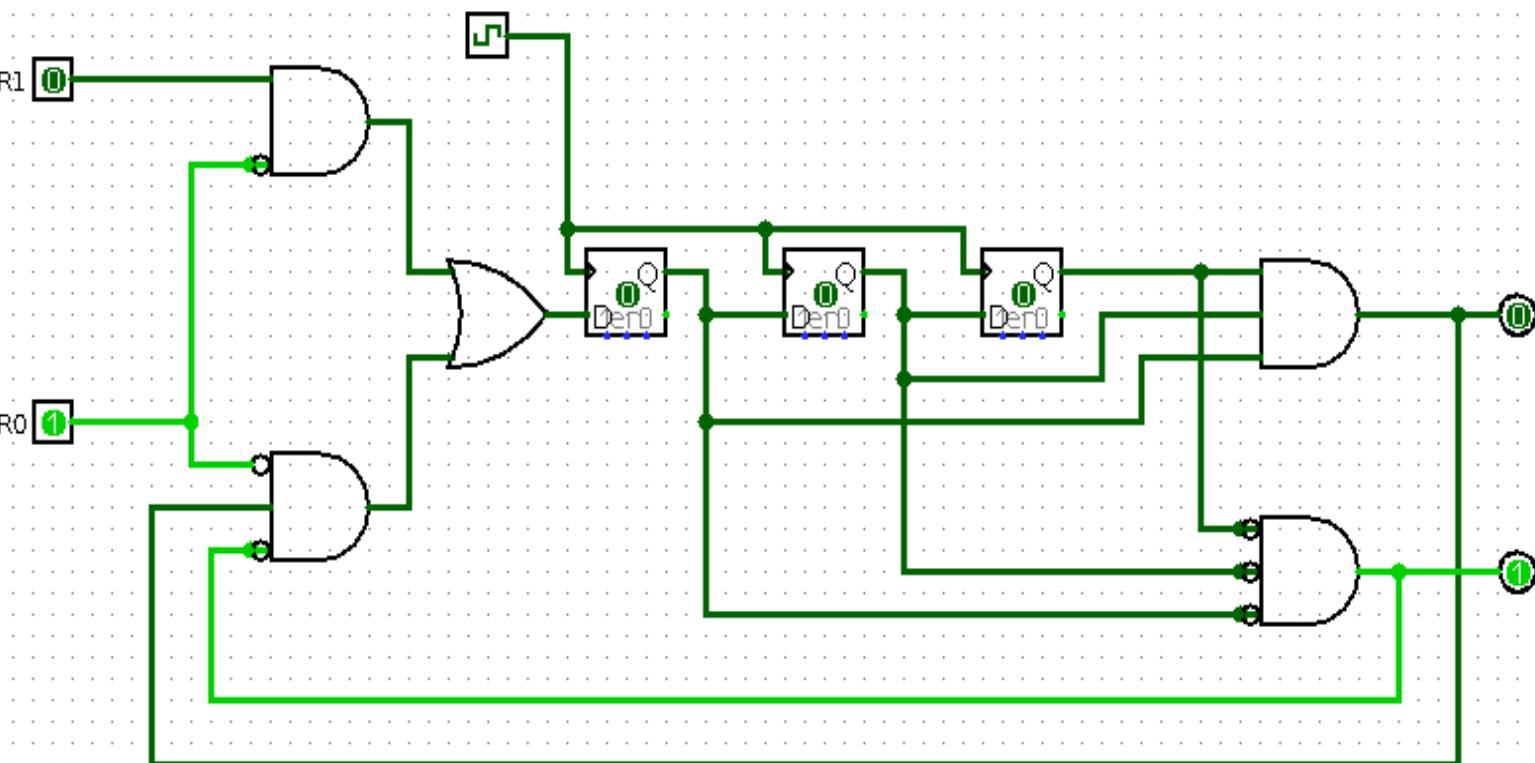
$$E1 = S2^* \times S1^* \times S0^*$$

$$S0 = (I1 * I0^\wedge) + (I1^\wedge * I0^\wedge * E1)$$

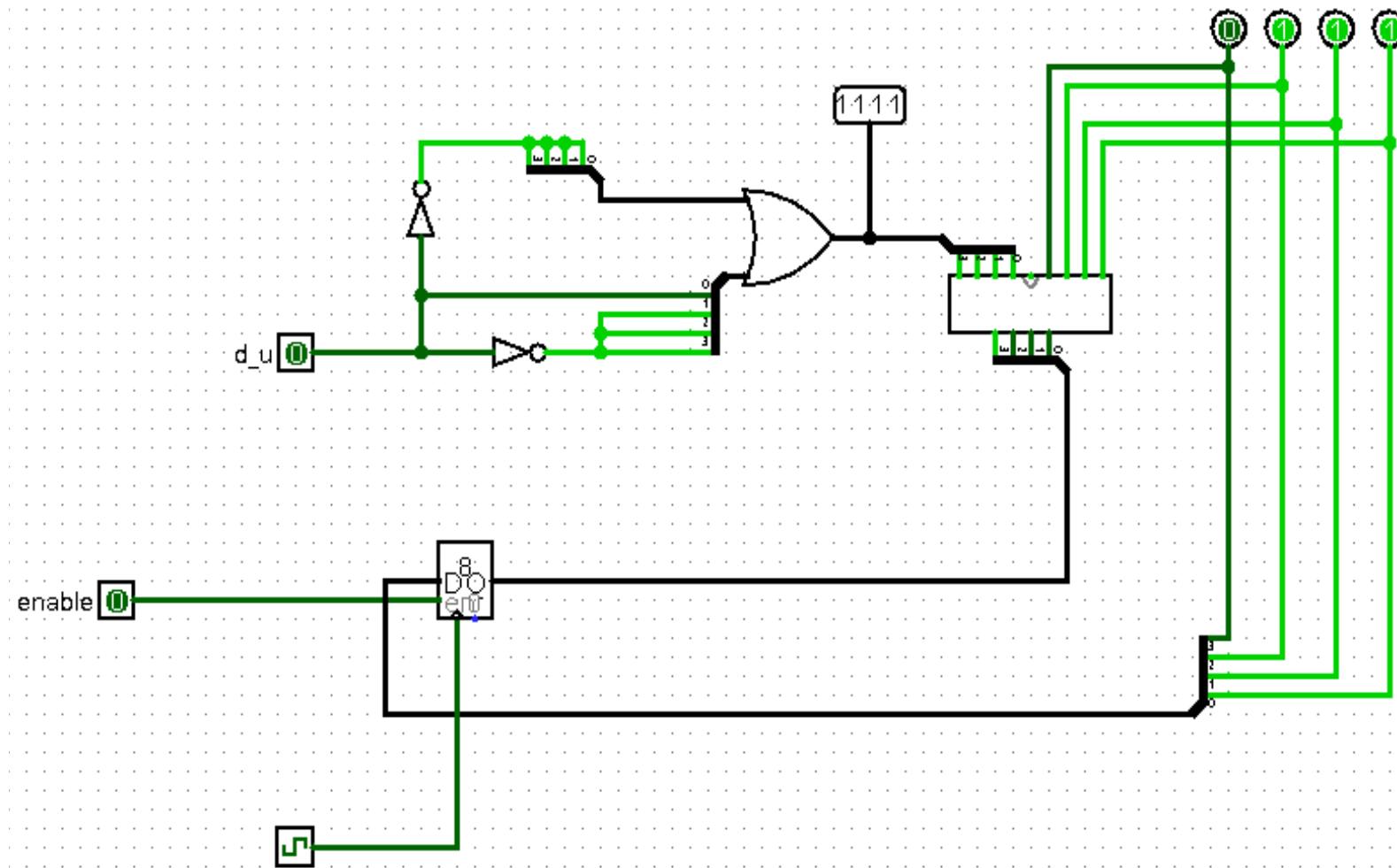
$$S1 = S0^*$$

$$S2 = S1^*$$

Esercizio 5.2



Esercizio 5.3



Laboratorio di architettura degli elaboratori

CIRCUITI SEQUENZIALI Lezione 5

- CONTATTI
- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

Esercizio 5.1

Costruire un circuito sequenziale con un segnale di ingresso e un segnale di uscita che riconosca la stringa 1101; ossia l'uscita del circuito assume il valore 1 quando l'ingresso attuale assieme a quelli nei 3 cicli di clock precedenti forma, nell'ordine temporale, la sequenza 1101; l'uscita assume il valore 0 altrimenti.

Esercizio 5.2

Costruire un circuito che simuli il funzionamento di un ascensore a due piani. Il circuito ha due segnali di ingresso che simulano i pulsanti di chiamata al piano, ha due segnali di uscita che segnalano la presenza dell'ascensore al piano. Si supponga inoltre che ascensore impieghi tre cicli di clock per passare da un piano all'altro.

Esercizio 5.3

Usando il circuito sommatore a 4 bit dell'esercizio 3.3.2, un registro e poche altre porte logiche (nonchè i concetti dell'aritmetica in complemento a 2), costruire un contatore up/down a 4 bit. Il circuito possiede due ingressi **e** ed **u**, e quattro uscite. L'ingresso **e** abilita il conteggio, il circuito non modifica l'uscita se l'ingresso **e** è a 0 mentre conta se l'ingresso è a 1. L'ingresso **u** determina il verso del conteggio: quando l'ingresso **u** è a 1, il contatore incrementa ad ogni ciclo di clock il valore dell'uscita; quando l'ingresso **u** è a 0, il valore dell'uscita viene decrementato

Trasferimenti dalla memoria

Scambio di dati tra i registri e la memoria principale.

- **load**: da memoria a registro
- **ldr r3, [r0]**
copia nel registro r3 4 byte a partire
dall'**indirizzo** contenuto in r0
- **ldr r3, [r0, #8]**
copia nel registro r3 4 byte a partire
dall'indirizzo in r0+8 (**base + offset**)
- **ldr r3, [r0, r1, lsl r2]**
copia nel registro r3 4 byte a partire
dall'indirizzo in r0 + r1 traslato del numero di
bit contenuto in r2 (**base + shifted register**).

Modalità di indirizzamento

- `ldr r3, [r0, #8]!` pre-incremento
 $r0 = r0 + 8; r3 = [r0]$
- `ldr r3, [r1, r0]!` pre-incremento da registro
 $r1 = r1 + r0; r3 = [r1]$
- `ldr r3, [r0], #8` post-incremento
 $r3 = [r0]; r0 = r0 + 8$
- `ldr r3, [r1], r0` post-incremento da registro
 $r3 = [r1]; r1 = r1 + r0$
- `ldr r3, [r1, -r0]` offset da registro
 $r3 = [r1 - r0]$
- `ldr r3, [r1, r0, lsl #2]` registro traslato
 $r3 = [r1 + 4 * r0].$

Parole allineate

`ldr` legge una parola (4 byte consecutivi in memoria)

legge solo **parole allineate**: l'indirizzo del primo byte è un multiplo di 4.

Endianness

I processori ARM normalmente sono **little-endian**:

byte meno significativi negli indirizzi più piccoli.

Ma possono funzionare in modalità big-endian: detti anche **bi-endian**.

Trasferimenti verso la memoria

Per inserire dati in memoria:

- **store register**: da registro a memoria
`str r0, [r4,#8]`

Tutte le modalità di indirizzamento di `ldr` restano valide anche per `str`.

Array (vettori)

Struttura dati molto comune. Definisce una sequenza indicizzata di elementi dello stesso tipo.
In assembly: sequenza di parole.

Supportati dai linguaggi alto livello.

In assembly:

- allocati nella memoria principale in locazioni consecutive;
- accessibili noto l'indirizzo base (quello del primo elemento).

Array

Per accedere all'elemento di indice i del vettore bisogna

- calcolare la sua posizione in memoria:
posizione = indirizzo base + offset
offset = indice \times dimensione parola
- leggerlo o scriverlo con `ldr`, `str`.

Esercizio

Scrivere in r1 la somma dei primi tre elementi del vettore (di interi) con indirizzo base r0

```
ldr r1, [r0]
ldr r2, [r0, #4]
add r1, r1, r2
ldr r2, [r0, #8]
add r1, r1, r2
```

Esercizio

Scrivere in r1 il valore $a[i] + a[i+1] + a[i+2]$ del vettore a, avente indirizzo base contenuto in r0

```
add r3, r0, r2, lsl #2
ldr r1, [r3]
ldr r4, [r3,#4]
add r1, r1, r4
ldr r4, [r3,#8]
add r1, r1, r4
```

Trasferire parti di una parola

Dalla memoria è possibile leggere o scrivere anche

- un singolo byte:

load register byte `ldr b` (logico)

load register signed byte `ldr sb` (aritmetico)

store byte `str b`

- un half-word (2 byte):

load register half word `ldr h`

load register signed half word `ldr sh`

store half word `str h`.

Per gli half-word l'indirizzo deve essere allineato, multiplo di 2.

Sintassi di un programma

Un programma assembly (file.s), oltre alle istruzioni contiene anche

- **etichette item:** : permettono di associare un identificatore a un indirizzo
- **direttive .text** : indicazioni all'assemblatore.

Le etichette sono seguite da due punti.

Es.: label_1:, label_R:.

Le direttive sono precedute da un punto.

Es.: .data, .globl.

Direttive principali

Alcune direttive compaiono in quasi tutti i programmi assembly:

- `.text` quello che segue è il testo del programma
- `.data` quello che segue sono dati da inserire in memoria
- `.globl` rende l'etichetta che segue visibile agli altri pezzi di codice
- `.end` specifica la fine del modulo sorgente.

Direttive di rappresentazione dati

Specificano il tipo di dati inserire in memoria:

- `.word` 34, 46, 0xAABBCCDD, 0xA01
ogni numero intero scritto con 4 byte
- `.byte` 45, 0x3a
ogni numero intero scritto con un byte
- `.ascii` "del testo tra virgolette "
i codici ascii dei caratteri della stringa
- `.asciiz` "altro esempio"
si aggiunge un byte 0 per marcare fine stringa
- `.skip` 64
vengono allocati 64 byte, inizializzati a 0.

Spesso etichettate per identificare la locazione di memoria contenente i dati.

Esempio di programma

```
.data
primes: .word 2, 3, 5, 7, 11
string: .asciiz "alcuni numeri primi"
.text
main: ldr r0, =primes
      ldr r1, [r0]
      ldr r2, [r0, #4]    @ leggo numero 3
      ldr r0, =string
      ldrb r3, [r0]
      ldrb r4, [r0, #3]    @ leggo carattere u
      swi 0x11            @ comando di terminazione
.end
```

Pseudo-istruzioni

`ldr r0, =primes` è una **pseudo-istruzione**. Carica in `r0` l'indirizzo di memoria etichettato con `primes`:

Una pseudo-istruzione è tradotta dall'assemblatore in sequenze di (1-4) istruzioni macchina. L'uso di pseudo-istruzioni semplifica la programmazione senza astrarre dal livello assembly.

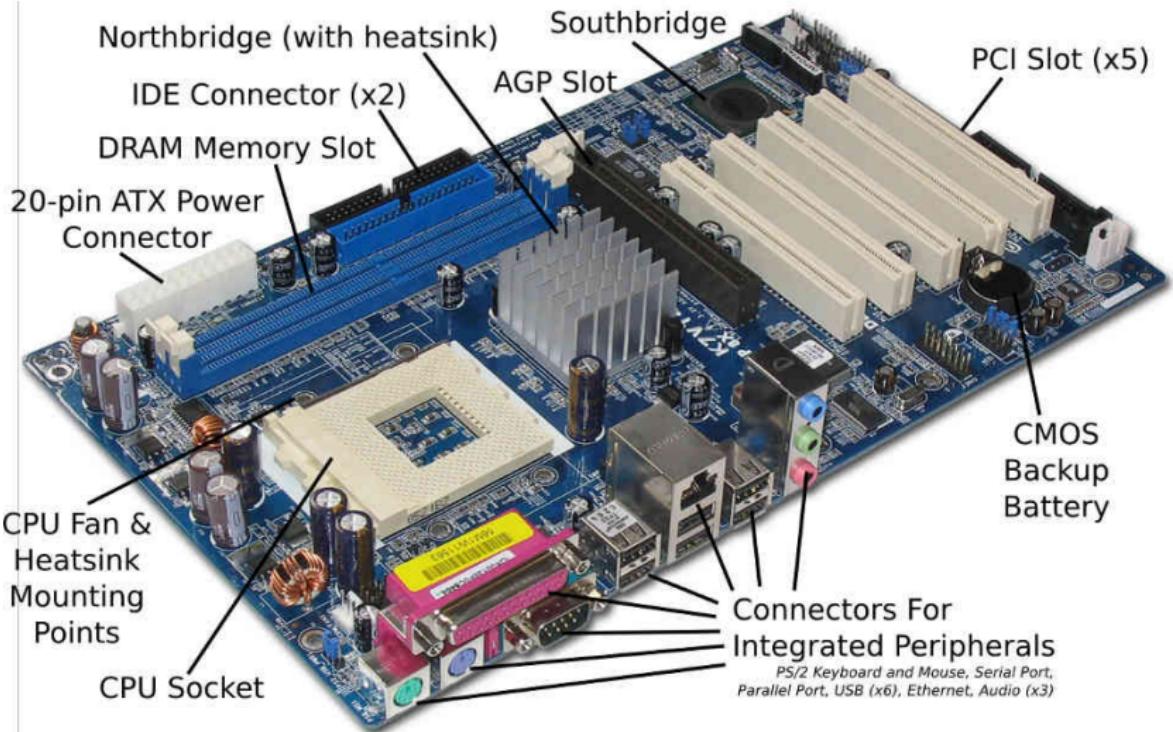
Non sono implementate dal processore, non fanno parte del linguaggio macchina per motivi di efficienza.

Struttura fisica di un calcolatore

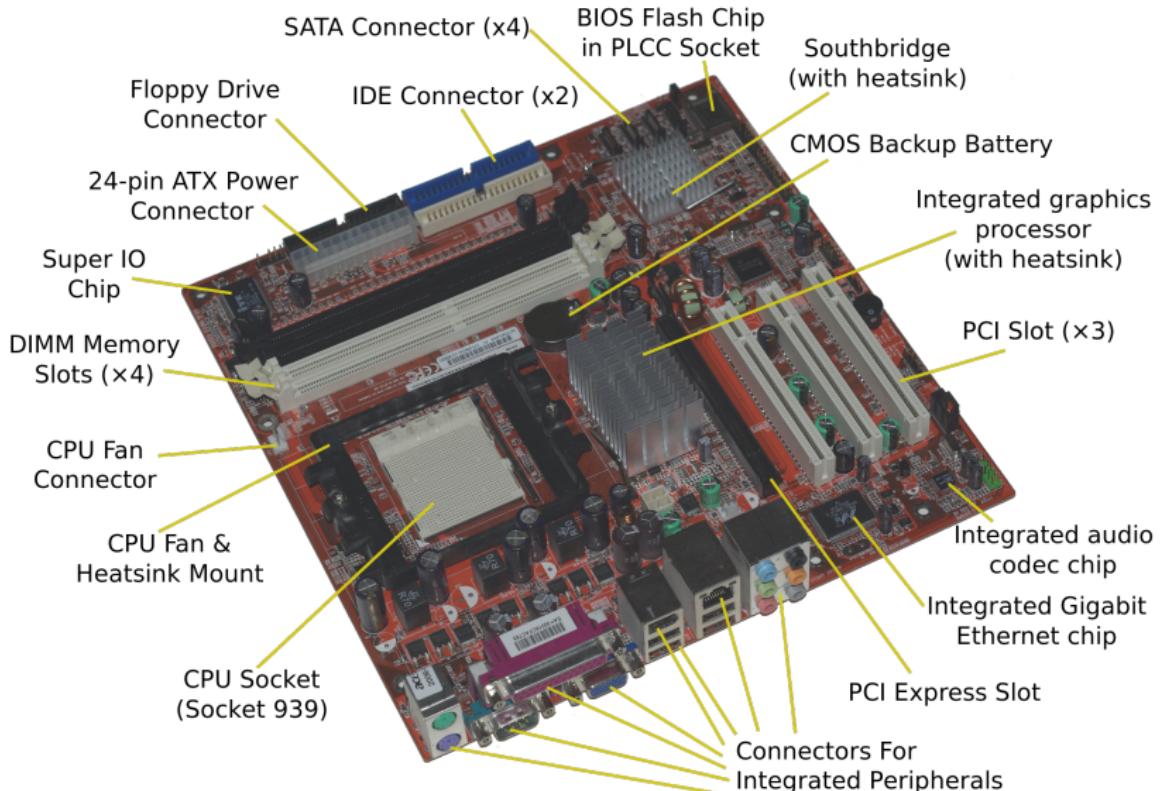
La CPU risiede su una **scheda madre**, un circuito stampato contenente:

- **bus**
- integrati per il controllo dei bus (**bridge**, chipset)
- alcuni circuiti di controllo e relative connessioni a periferiche (USB, tastiera, modem)
- un insieme di connettori per
 - memorie (moduli DIMM)
 - controllori per altre periferiche (schede video, audio, schede di rete, ...)
 - collegamento ai dischi (magnetici, ottici)
 - alimentazione
- elettronica (resistori, condensatori, induttori).

Esempio di motherboard: ASRock



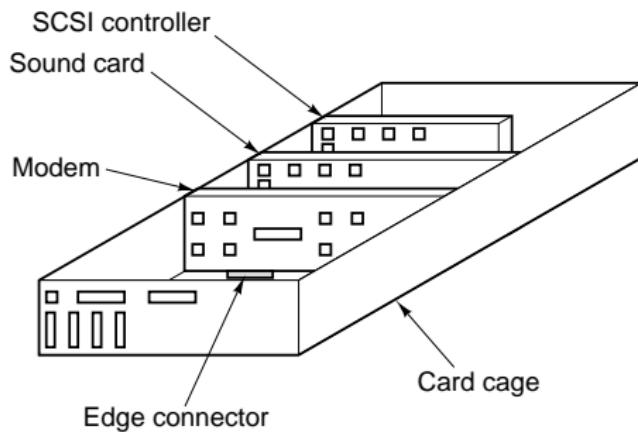
Esempio di motherboard: Acer



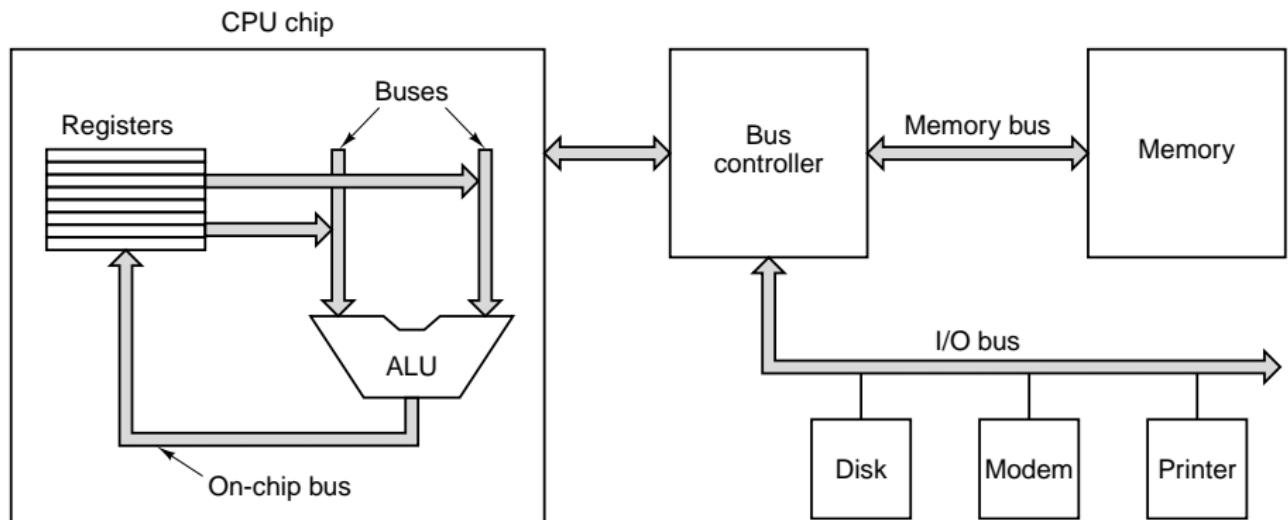
PS/2 Keyboard and Mouse, Serial Port,
Parallel Port, VGA, Firewire/IEEE 1394a,
USB (x4), Ethernet, Audio (x6)

Sulla scheda madre

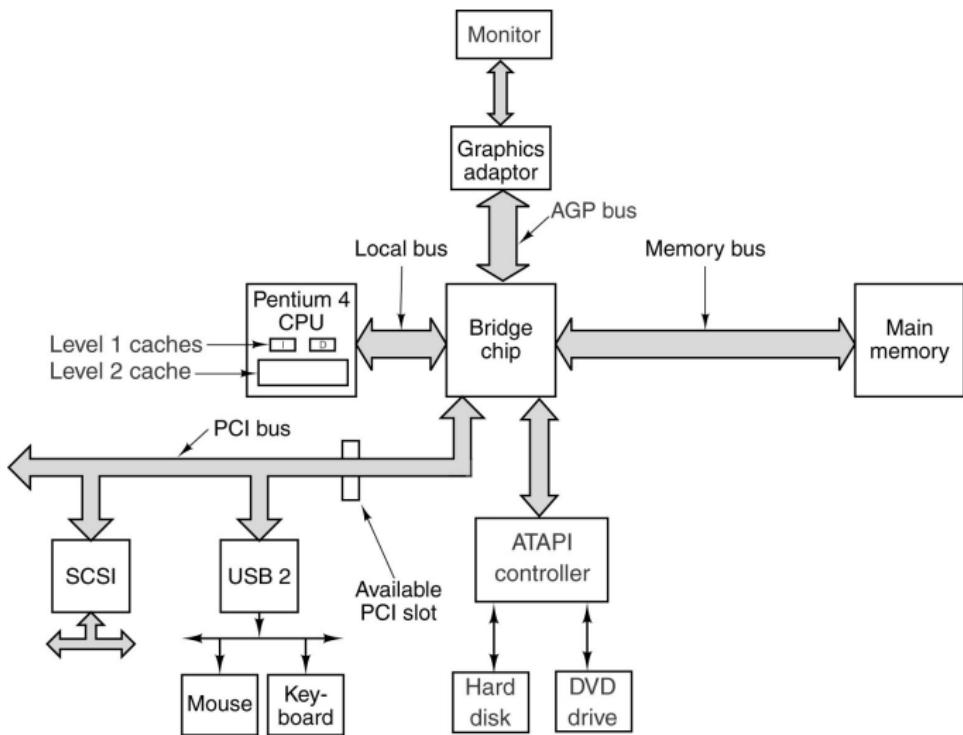
- schede di memoria
- schede controllori
- memoria permanente (hard disk, stato solido)
- alimentatore, circuiti di raffreddamento, case.



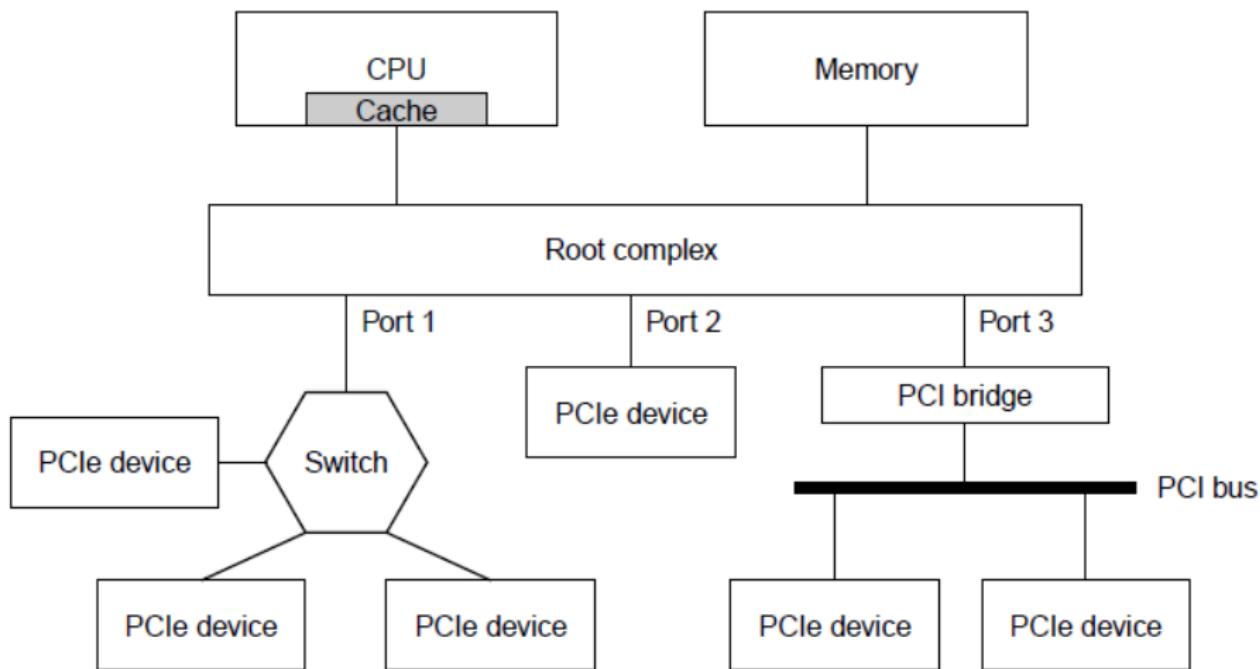
Struttura logica schematizzata



Struttura logica realistica



Strutture logiche più recenti



Bus

Bus: insieme di collegamenti tra due o più dispositivi.

- Economico: più dispositivi usano le stesse linee.
- Flessibile: è semplice aggiungere nuovi dispositivi.

Processore, memoria, controllori e periferiche sono collegati attraverso un sistema complesso di bus.

Per migliorare le prestazioni e gestire dispositivi con velocità diversa, si usano più bus.

Protocolli

Oltre agli aspetti fisici (**collegamenti**), un bus è caratterizzato dai suoi **protocolli**: regole di comunicazione.

- Bus **privati**: CPU, CPU - Memoria.
- Bus **pubblici**: CPU - I/O, BUS esterni.

Tecnologie storicamente importanti: Omnibus (PDP-8), PC IBM (PC/XT), ISA (PC/AT), PCI, PCI-Express (bus di sistema), ATA, SCSI, USB (periferiche esterne).

Caratteristiche fisiche di un bus

Realizzazione **fisica**: insieme di connessioni, da alcune unità ad alcune centinaia di collegamenti.

- Interni a un circuito integrato (es.: CPU): tracce di alluminio o rame
- su circuiti stampati (es.: scheda madre): tracce di rame
- esterni: cavi **schermati**.

Schermatura **differenziale** per diminuire i disturbi: coppie di cavi **intrecciati** o **coassiali**.

Connessione dei dispositivi

Poichè ci sono più dispositivi su una stessa connessione, solo un dispositivo deve inviare il segnale in assenza di interferenze:

- dispositivi **tri-state** (buffer invertente - non invertente)
- **open-collector**: wired-OR.

Necessità di **amplificare** il segnale: bus **driver** - bus **receiver** - bus **transceiver**.

Frequenza e banda passante

Frequenza del bus: volte al secondo in cui si può commutare un dato nel successivo.

Più alta la frequenza maggiori le prestazioni.

Massima **banda passante teorica** =
frequenza × numero di dati nella singola
commutazione (bit/sec).

La banda passante **reale** è minore a causa di fasi di inattività e di coordinamento/sincronizzazione tra i vari dispositivi.

Bus skew

Limiti fisici all'aumento della frequenze e alla diminuzione della potenza assorbita:

- trasmissioni a bassa tensione sono più esposte a interferenze (disturbi) esterni
- ad alta frequenza i ritardi di propagazione diventano significativi.

Bus skew: segnali su linee diverse sono recapitati in tempi diversi. Segnali allineati in partenza vengono ricevuti non allineati.

Bus skew nelle tecnologie attuali, difficilmente superabile: ~ 1 ns.

Il periodo = 1/frequenza del clock deve essere maggiore del bus skew: limite alla frequenza.

Tipologia di informazioni

In un bus parallelo tipicamente si riconoscono

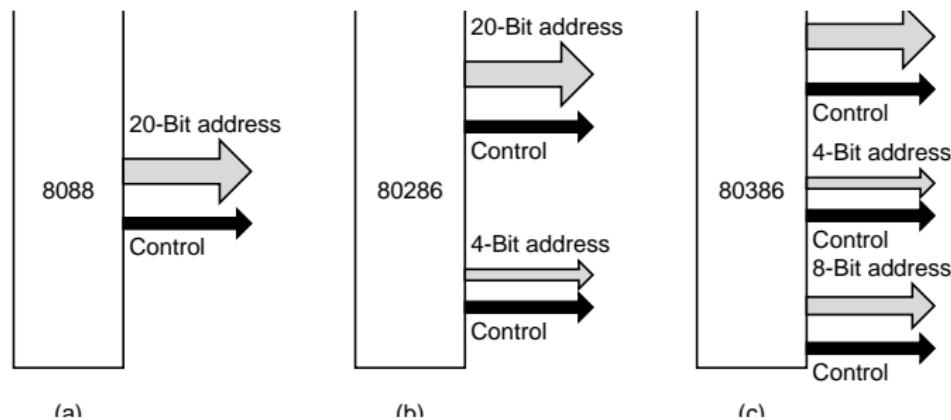
- linee **dati**: le informazioni da recapitare
- linee **indirizzi**: identificano locazioni di memoria o registri di dispositivi
- linee di **controllo**: recapitano comandi e informazioni sul funzionamento del bus.

Maggior numero di linee: più dati, più spazio indirizzabile. Maggiori costi.

Numero di linee

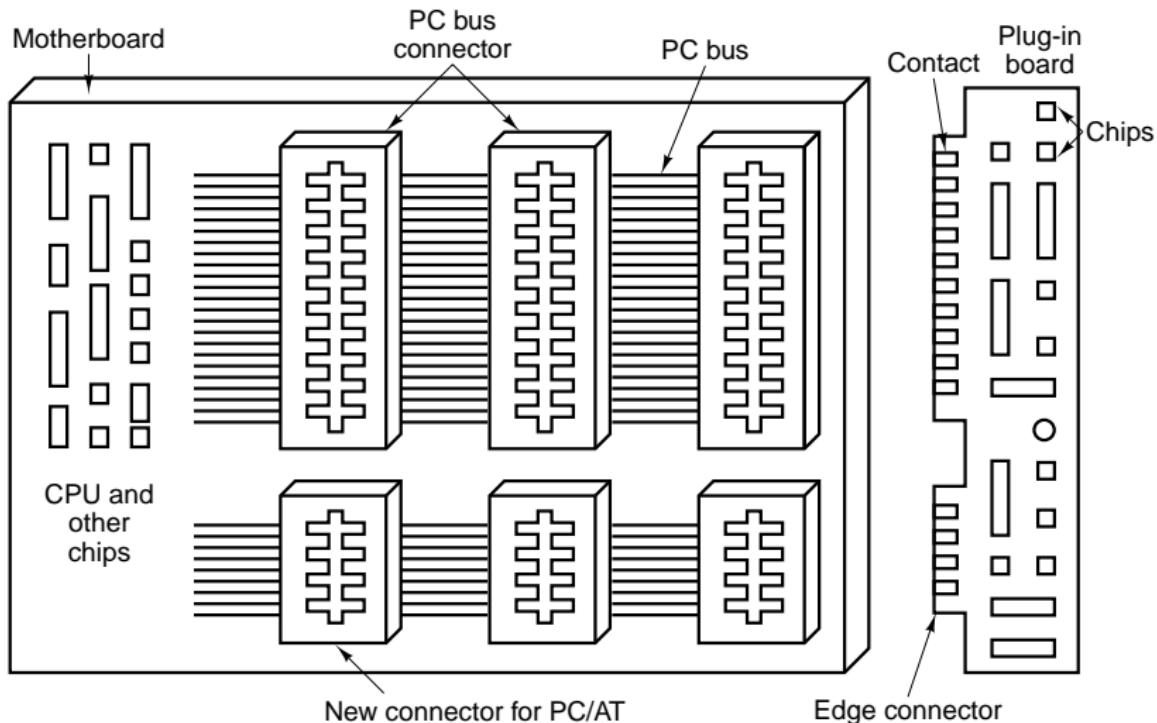
Aggiornare i bus richiede di aggiungere linee.
Problema della retrocompatibilità.

Es.: bus Intel x86:



Multiplexed bus: stesse linee utilizzate per dati diversi. Es.: indirizzamento della memoria a due passi.

Estensione delle schede



Codici di correzione degli errori

Una comunicazione via bus è esposta a errori (anche se poco frequenti).

- Bus paralleli: codici di parità (linee dedicate, errori per ipotesi non superiori a uno per pacchetto).
- Bus seriali: codici di controllo aggiunti in coda a ogni pacchetto. In grado di rilevare anche errori multipli.

Protocolli di comunicazione

Una **transazione** sul bus prevede le seguenti fasi:

- un dispositivo prende il controllo del bus
 - invia una richiesta di comunicazione a un secondo dispositivo
 - la richiesta viene soddisfatta dal secondo dispositivo
 - il bus viene liberato per un'altra comunicazione.
-
- **Master**: dispositivo che prende il controllo del bus e inizia l'interazione.
 - **Slave**: dispositivo che risponde al master.

Master o slave

In transazioni diverse, un dispositivo può essere a volte master e a volte slave.

Esempi:

Transazione	Master	Slave
Lettura, scrittura	CPU	Memoria
Inizio I/O	CPU	Controllore
Operazione DMA	Controllore	Memoria
Acquisizione operandi	Coprocessore	CPU

Arbitraggio

Il bus è responsabile della scelta di un master.

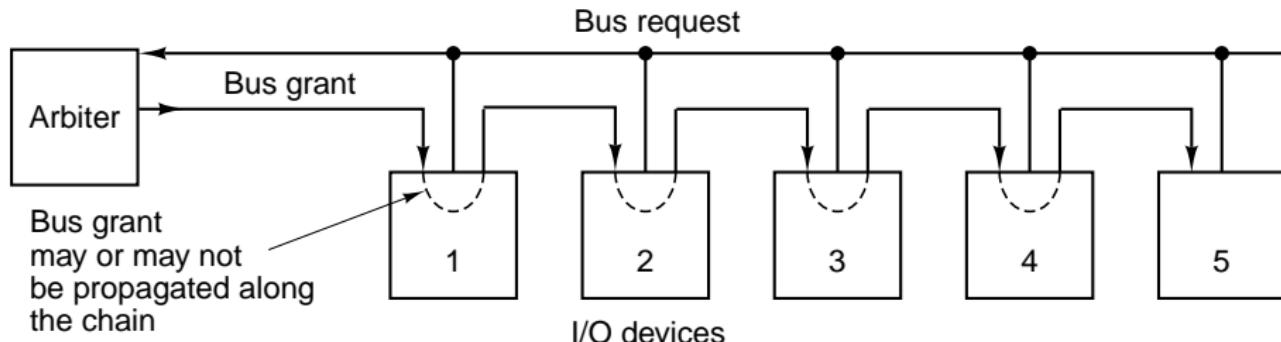
Meccanismi:

- **centralizzato**: un circuito fa da **arbitro**; i dispositivi richiedono all'arbitro l'accesso al bus
- **decentralizzato**: nessun arbitro; la scelta del master è determinata da un protocollo distribuito tra i dispositivi collegati al bus.

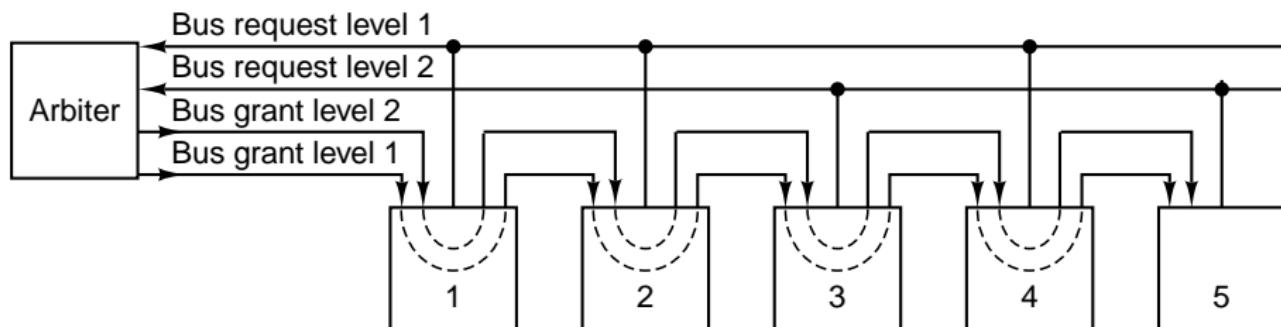
Criteri per l'arbitraggio

- **priorità** tra dispositivi: si privilegiano i dispositivi più importanti
- **fairness** (equità): ogni dispositivo deve poter accedere, prima o poi, al bus.

Daisy chain



(a)



(b)

Daisy chain

Esempio di protocollo centralizzato. Passaggi:

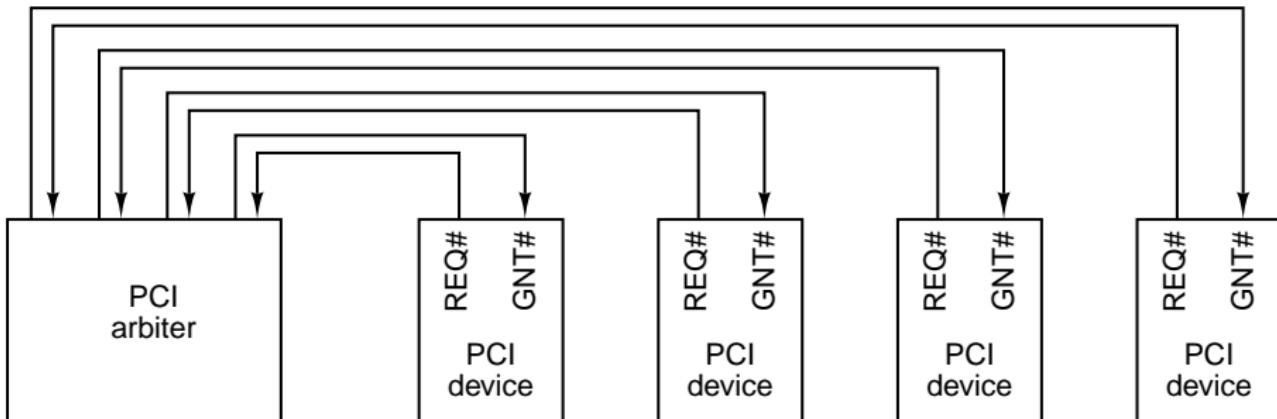
- richiesta del bus asserendo la linea di richiesta (connessione wired-OR col bus)
- se il bus è libero l'arbitro emette un token asserendo la linea di concessione (bus grant)
- il token è fatto transitare sino a raggiungere il primo dispositivo che ha richiesto il bus.

Semplice ma poco flessibile:

- priorità determinata dalla posizione lungo il bus grant
- non soddisfa i criteri di fairness.

Più gerarchie di priorità adoperando più linee di richiesta - grant. Aumenta la fairness.

Bus PCI

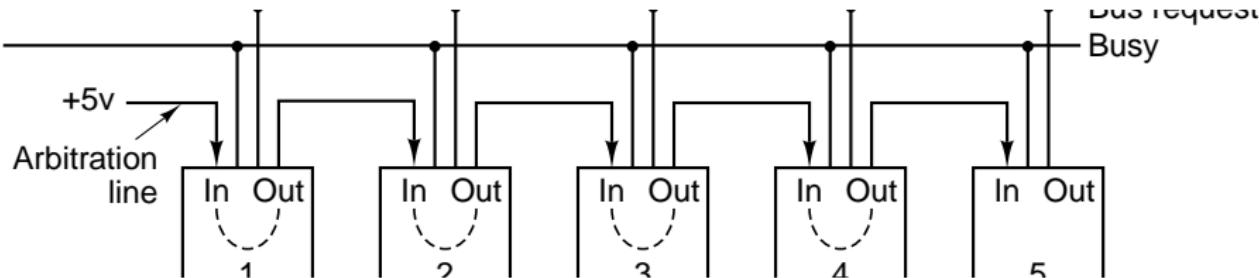


Costoso: due linee per ogni dispositivo.

Flessibile: l'arbitro ha un controllo completo della situazione.

Arbitraggio decentralizzato

Simile a daisy chain (stessa gerarchia di priorità):



Tre linee per i comandi: **busy**, **request**, **grant**.

- il bus è disponibile se linea busy non è asserita
- i dispositivi che vogliono trasmettere disattivano il grant verso il dispositivo alla loro destra
- solo il dispositivo che vede la linea grant asserita può richiedere il bus.

Protocolli decentralizzati

- SCSI: una linea di richiesta per dispositivo
- Ethernet:
nessun arbitraggio iniziale
si rilevano conflitti (sovraposizioni) sul bus
in caso di conflitto il dispositivo annulla la
comunicazione e, dopo un ritardo casuale,
ritenta.

Transazione sincrona e asincrona

Due approcci:

- Bus sincrono: **è presente** un segnale di **clock** che regola lo scambio dei dati.
- Bus asincrono: **non è presente** un segnale di **clock**.

Bus sincroni

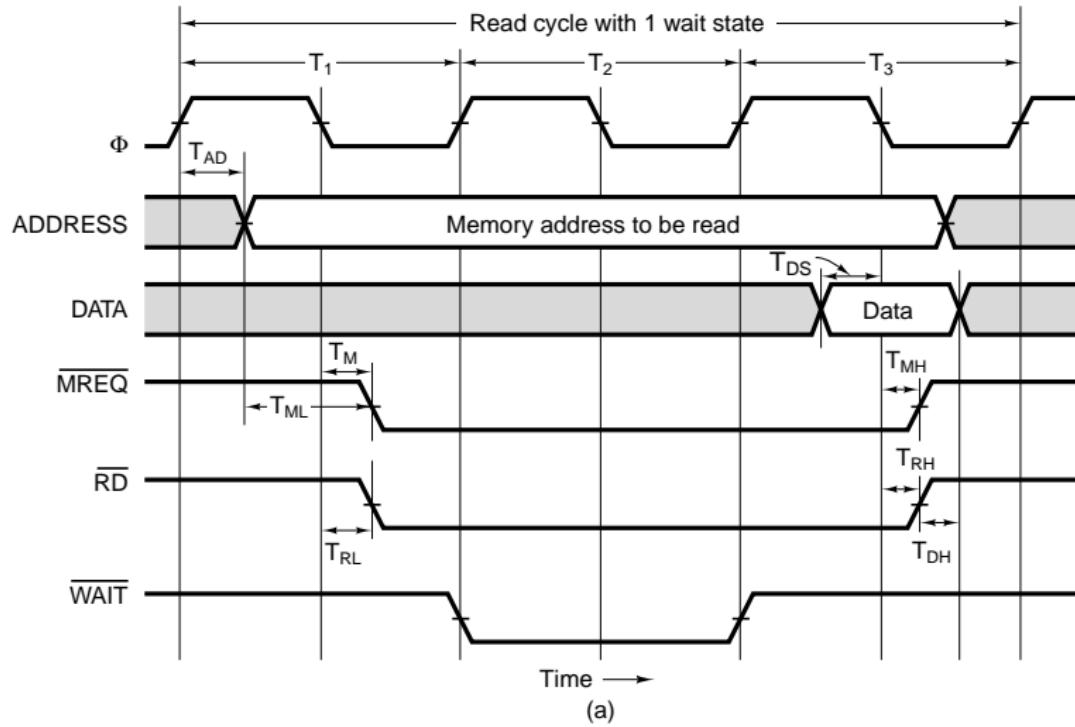
Il clock, presente su una delle linee del bus, vincola i tempi di risposta.

- **Vantaggi:** semplicità del protocollo di comunicazione, velocità.
- **Svantaggi:** più rigido.

Protocollo inadatto a collegare dispositivi con diversi tempi di risposta.

Inadatto a bus lunghi (il segnale di clock centralizzato raggiunge i dispositivi in istanti diversi).

Lettura sincrona di una locazione di memoria

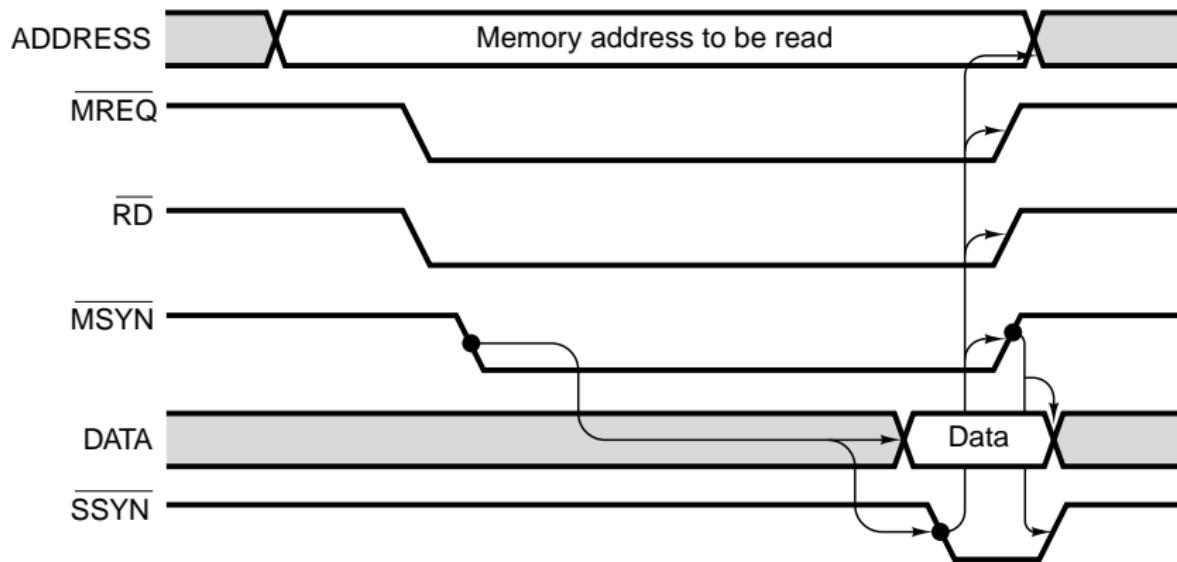


Bus asincroni

Nessun segnale di clock:
sincronizzazione mediante **handshaking**.

- **Vantaggi**: maggiore flessibilità; supporta dispositivi con diversi tempi di risposta; meno problemi su bus lunghi.
- **Svantaggi**: circuiti di connessione più complessi.

Lettura asincrona di una locazione di memoria

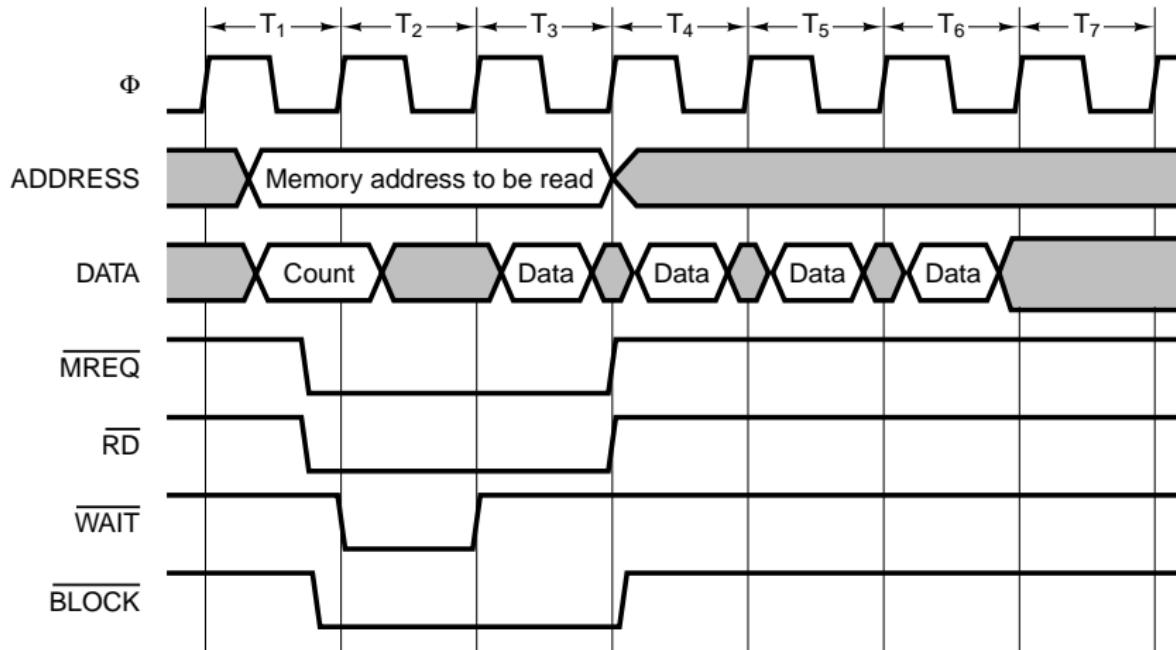


Transazioni: cicli di bus

Durante una comunicazione sul bus si possono realizzare diverse operazioni

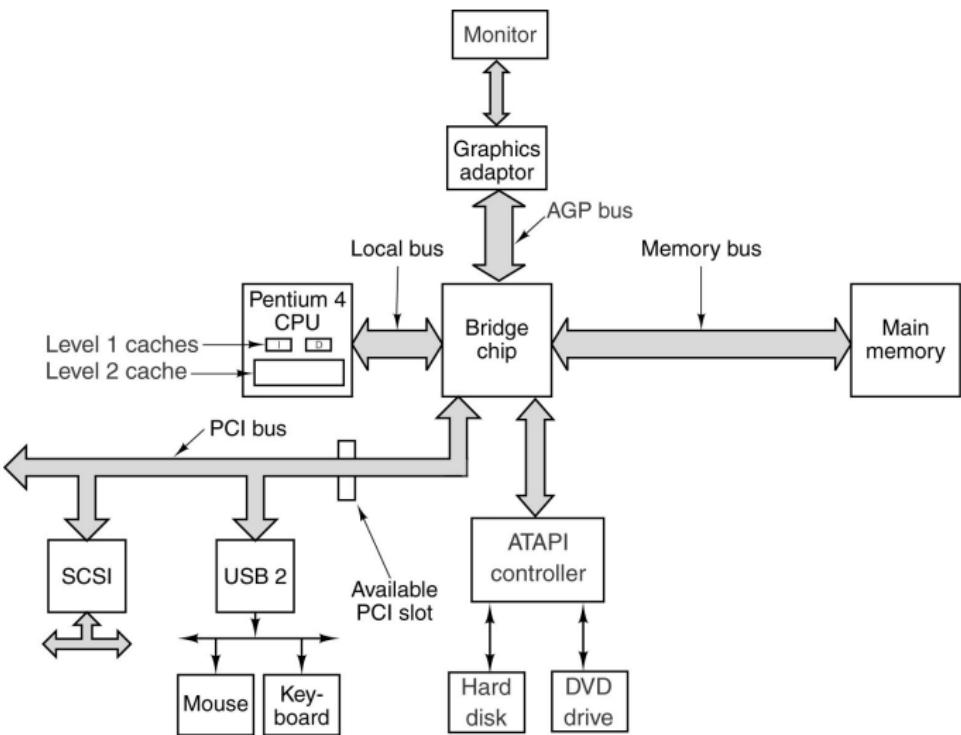
- lettura o scrittura di un registro o locazione di memoria
- lettura o scrittura di un **blocco** di dati
- **read-modify-write**: lettura e immediata scrittura di una locazione (gestione risorse condivise)
- interrupt: richiesta e invio vettore di interrupt
- configurazione e test dei controllori.

Lettura sincrona di un blocco di memoria



4 blocchi in 6 cicli invece dei 12 richiesti da 4 letture.

Struttura logica di un PC (rivediamo)



Bus a banda larga

Diversi bus per molteplici esigenze.

Alcuni bus con una larga banda passante.

Collegano le componenti più vicine al processore,
costosi e con restrizioni fisiche:

- bus locale, front-side bus
- bus di memoria
- bus della cache, back-side bus.

Altri bus

Altri a banda più stretta connettono le periferiche.

- Bus di sistema: **PCI, PCIe** è il bus principale per connettere dispositivi di controllo.
- Bus per la connessione di specifici controllori: **ISA, ATA, SCSI**.
- Bus per dispositivi esterni: **USB, FireWire**.
- Bus della scheda video con **Accelerated Graphics Port** (AGP).

Connessioni della CPU

Numero elevato di connessioni (qualche centinaio).

- Principalmente appartenenti al front side bus:
 - indirizzi
 - dati
 - comandi, arbitraggio
 - controllo degli errori.
- Centinaia di connessioni di alimentazione e massa: elevata intensità di corrente, eliminazione dei disturbi.
- Dissipazione del calore (Core i7: ~ 150 W di picco; ARM: < 1 W).
- Configurazione, diagnostica, gestione della temperatura.

Front-side bus

Collegamento unico del processore con il resto del sistema. Noto anche come North Bridge.

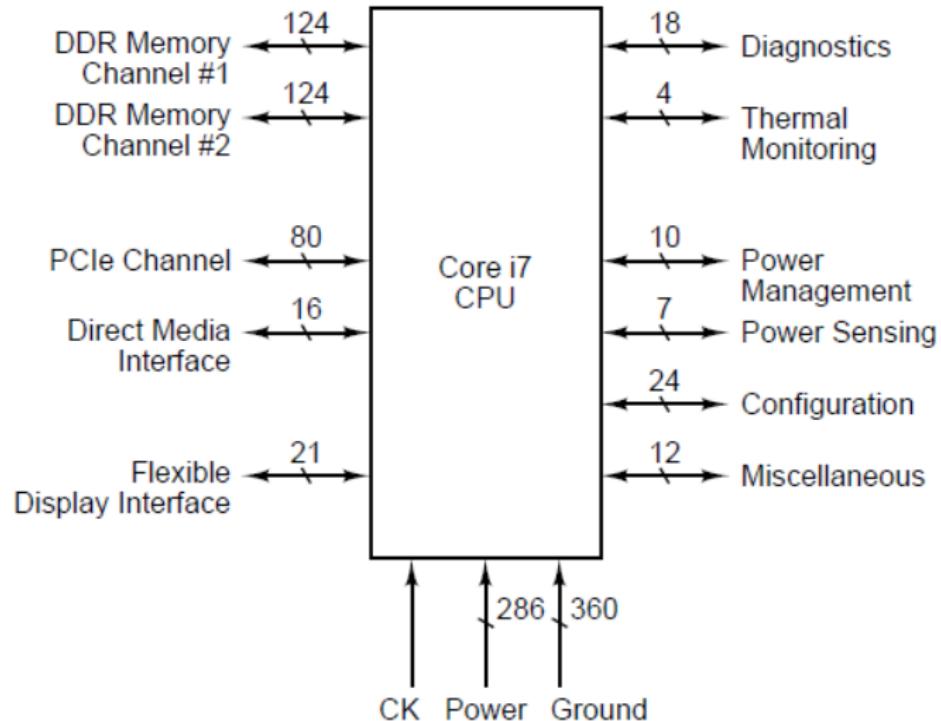
- Collega CPU a chipset. Proprietario, specifico per quel particolare processore.
- Veloce (133-400 MHz). In un singolo ciclo di clock 2 o 4 transazioni possibili. 32-64 linee dati.
- Migliori prestazioni si ottengono con bus seriali: HyperTransport (AMD), QuickPath Interconnect QPI (Intel).
- In processori recenti interno al chip del processore.

Connessioni Intel Core i7

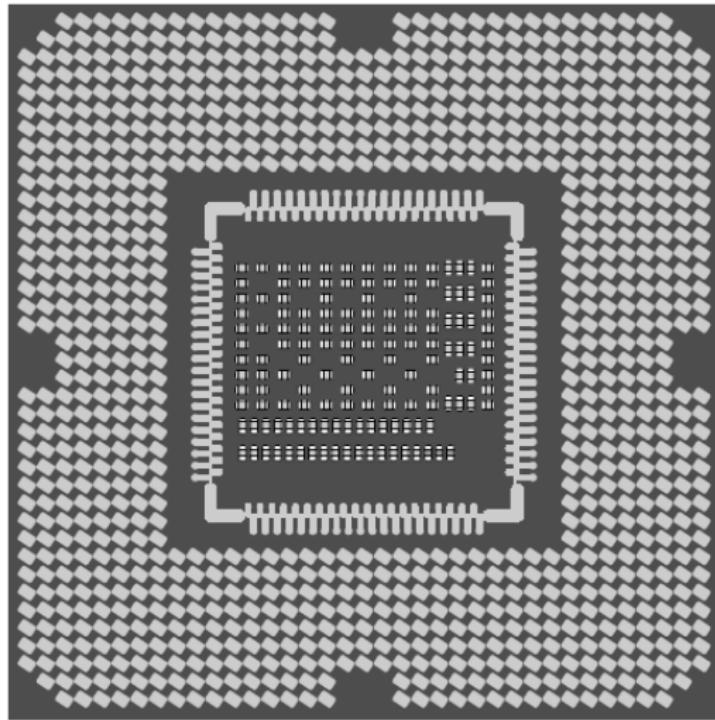
Un chipset interno al circuito integrato da cui escono:

- 2 bus per connessione a due banchi di memoria DDR3 SDRAM; ciascuno con 64 linee dati, 666 MHz con DDR (due transazioni per ciclo di clock): 20 GB/s
- 1 bus PCIe per la connessione alla scheda grafica: 16 GB/s
- 1 bus Direct Media Interface, bus proprietario simile a PCIe per connessione ad un chipset esterno a cui si collegano le restanti periferiche: 20 GB/s.

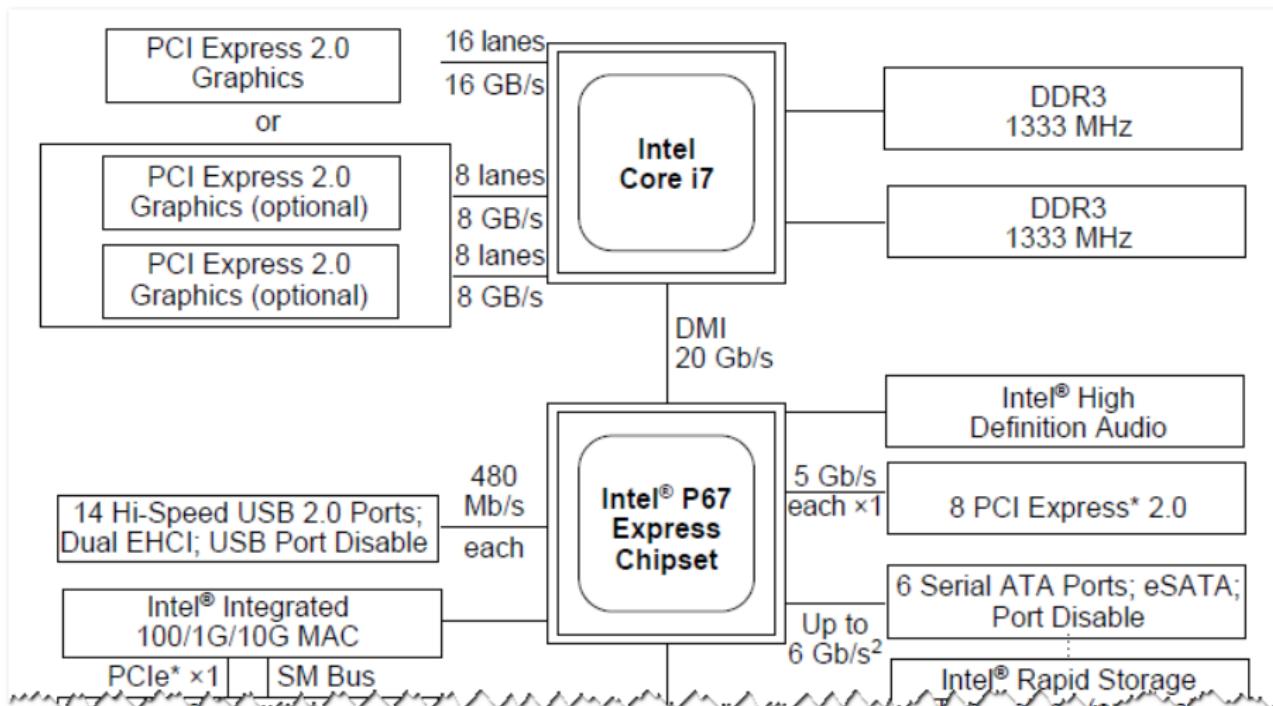
Core i7: schema connessioni



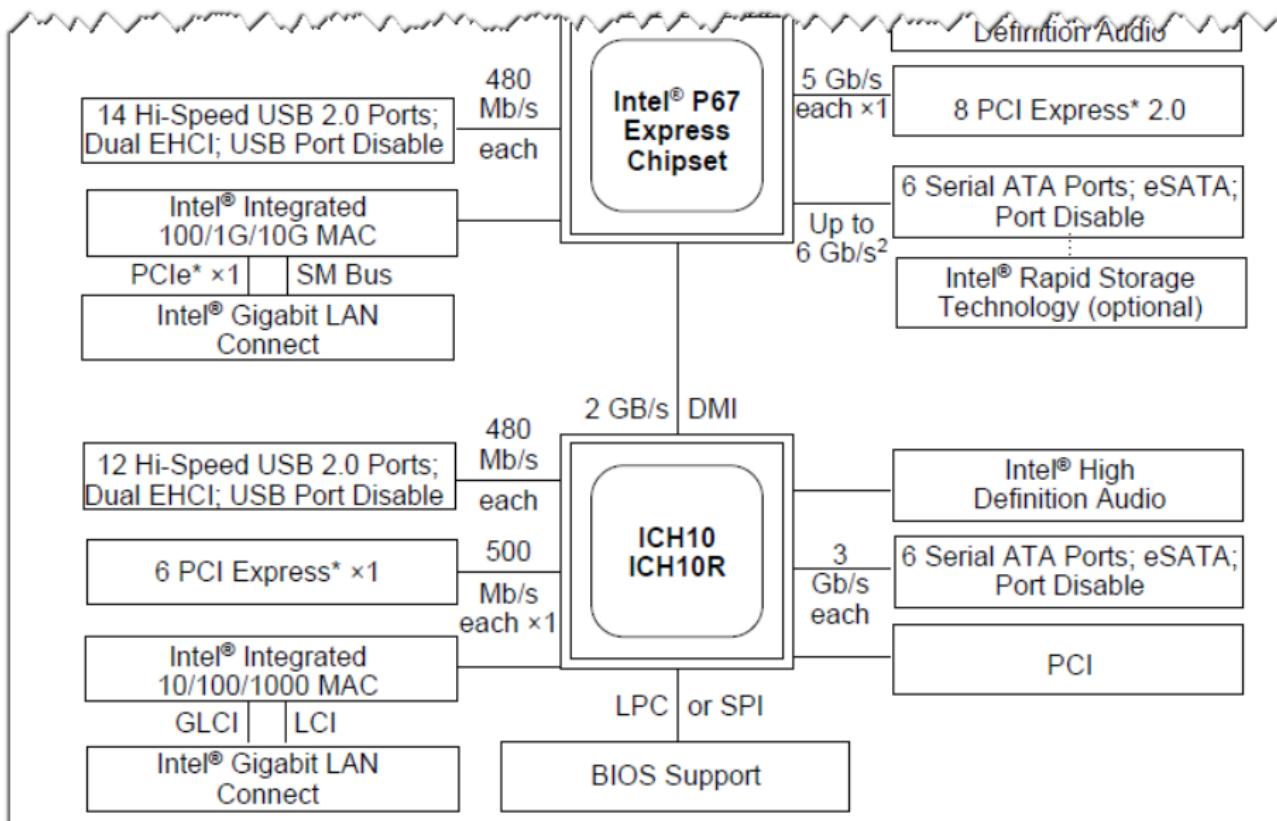
Pedinatura fisica del Core i7



Struttura logica di un PC con Core i7



Struttura logica di un PC con Core i7



Bus della memoria DDR3

Parallelizzazione accessi alla memoria (pipelining).

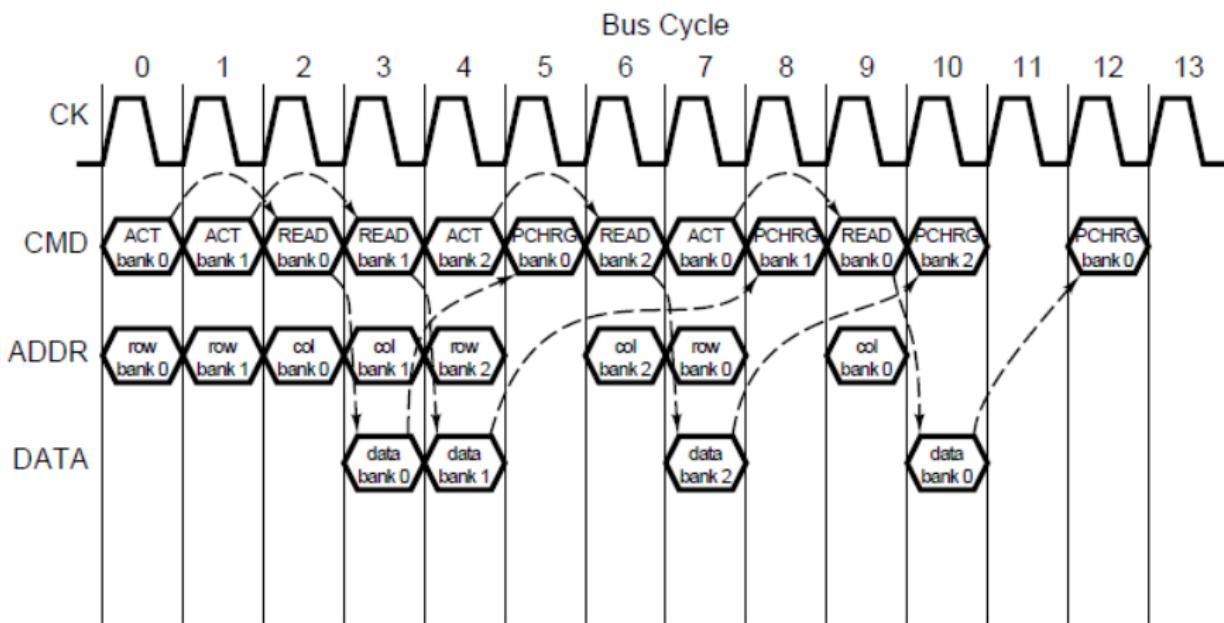
Accesso ai dati in 3 fasi

- **ACTIVATE**: si prepara la lettura, una riga della matrice di celle viene preparata all'accesso
- **READ/WRITE**: si eseguono accessi multipli a singole o sequenze di parole (**burst mode**) nella riga attivata
- **PRECHARGE**: chiude la riga corrente e prepara la memoria a una nuova ACTIVATE.

La memoria DDR è divisa in banchi (tipicamente 8); fino a 4 banchi possono essere attivati simultaneamente.

Pipeline sulla memoria

i segnali di comando, indirizzi, dati, possono operare su **transazioni diverse**:



Bus di sistema: ISA

ISA (Industry Standard Architecture), bus di sistema dei primi PC, evoluzione dei bus PC bus (IBM) e PC/AT bus (Intel 80286).

Contiene 64 + 36 linee:

20 + 4 linee indirizzi

8 + 8 linee dati

Sincrono con clock a 8.33 MHz.

I bus IDE, ATA sono una sua diretta derivazione.

A volte presente, per legacy, anche nei PC attuali oppure sostituito da LPC Bus che lo simula a livello software.

Bus di sistema: PCI

PCI (Peripheral Component Interconnect)

Intel 1992, in sostituzione bus ISA (EISA, VESA).
Brevetti resi pubblici da Intel.

Evoluzione gestita dal consorzio PCI-SIG (PCI Special Interest Group).

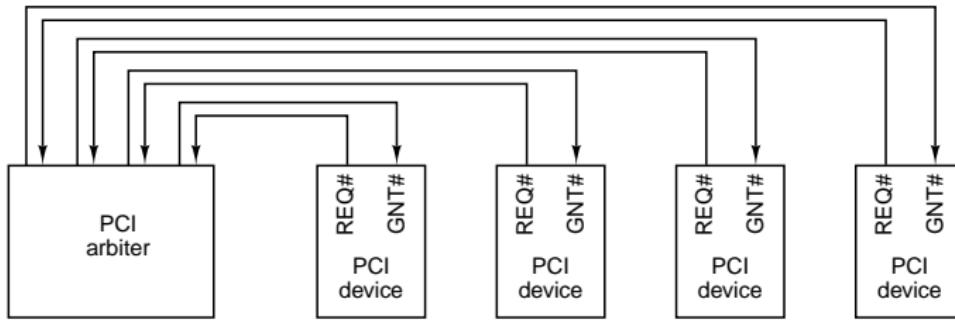
Diverse versioni: PCI, PCI 2.0, PCI 2.1, PCI 2.2, PCI-X, PCI-X DDR

- 32-64 linee dati/indirizzi (multiplexed);
- clock a 33, 66, 133, 266 MHz;
- alimentazione a 5 V e poi 3,3 V (entrambe possibili).

Bus PCI

Diversi tipi di scheda:

- numero di piedini;
- tensione di alimentazione (bus fisicamente diversi);
- frequenza (selezionata asserendo una linea);
- arbitraggio centralizzato nel bridge.



PCI: arbitraggio

- Vantaggi: velocità.
- Svantaggi: numero di linee.

REQ# = Richiesta.

GNT# = Grant (assegnazione).

Logica negata: motivazione anche pratica.

Politica di arbitraggio: libera, non vincolata dalle specifiche PCI.

Il master può usare il bus per più cicli se la linea di grant rimane abilitata.

Terminologia: Initiator (master), Target (slave).

PCI: segnali

CLK: clock (inizio periodo: fronte di discesa)

AD: (32 linee) indirizzi e poi dati - M o S

PAR: bit di parità per AD - M o S

C/BE: (4 linee) tipo di comando e poi byte validi nel word a 32 bit - M

FRAME: via libera, comandi e indirizzo validi - M

IRDY: M disponibile a leggere dati, o correttezza dei dati uscenti dal master - M

IDSEL: lettura configurazione dispositivo (P&P) - M

DEVSEL: risposta di disponibilità - S

TRDY: duale a IRDY - S

STOP: richiesta (inattesa) fine transazione - S.

PCI: segnali ausiliari e per chip 64 bit

PERR: errore sul controllo di parità - M o S

SERR: errore di sistema o di indirizzamento

REQ - GRN: richiesta - assegnazione bus

RST: reset manuale del sistema, errore irrimediabile

REQ64: richiesta transazione a 64 bit - M

ACK64: riscontro a REQ64 - S

AD: (32 linee) estensione indirizzi e dati - M o S

PAR64: extra bit di parità

C/BE: (4 linee) estensione del comando C/BE

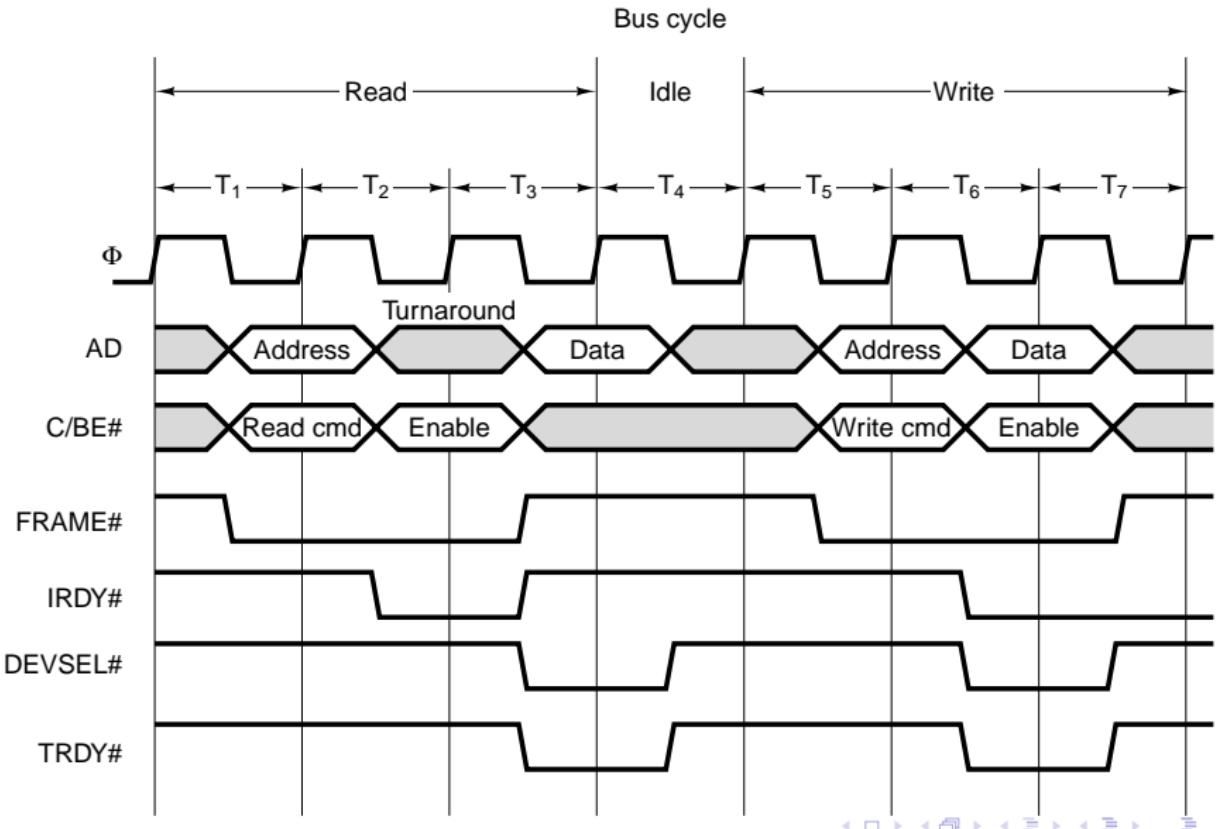
LOCK: lock per transazioni multiple

SBO - SDONE: snooping, sistemi multiprocessore

INTx: (4 linee) - richiesta di interrupt

M66EN: selezione frequenza del clock.

PCI: transazione lettura/scrittura



PCI Express (PCIe)

Evoluzione del bus PCI. In realtà non ne eredita quasi nulla.

Motivazioni

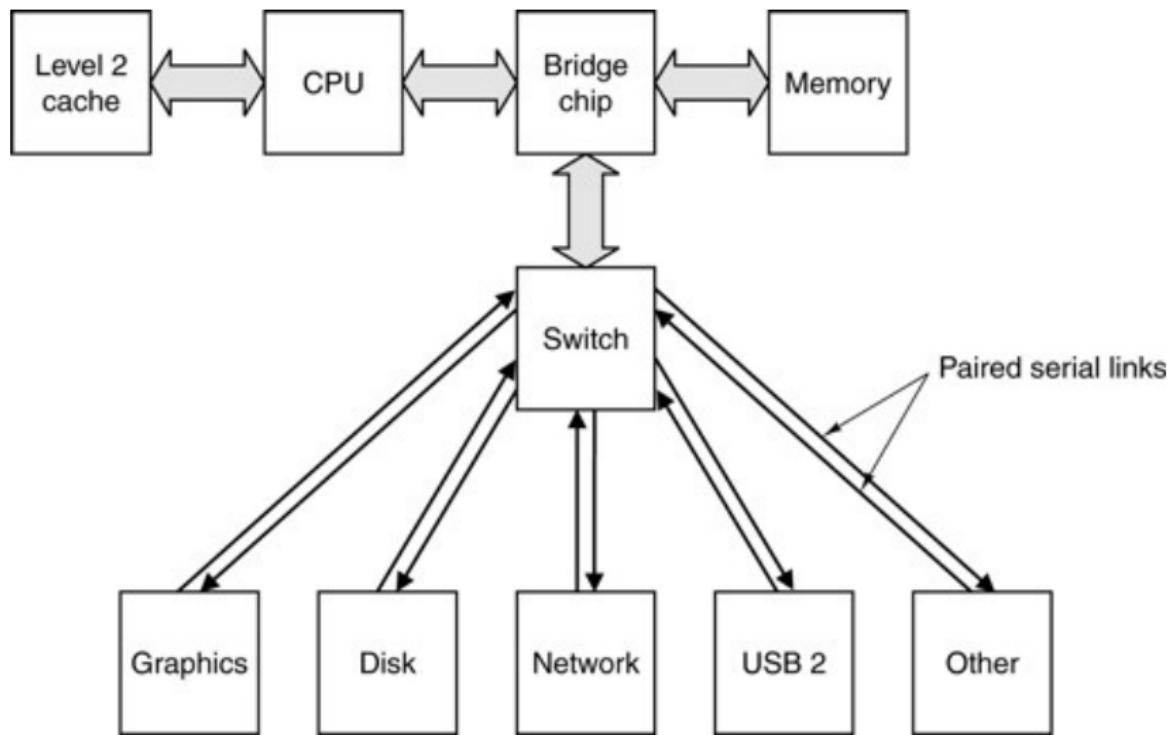
- la tecnologia del bus PCI ha raggiunto i suoi limiti: è difficile aumentare la frequenza e la banda passante (bus skew)
- avere un unico bus per tutte le periferiche: eliminare AGP, ATA, bus della memoria eccetera
- avere un bus fisicamente più compatto, con minori vincoli di lunghezza, connettere direttamente memoria esterna.

PCIe: tecnologia

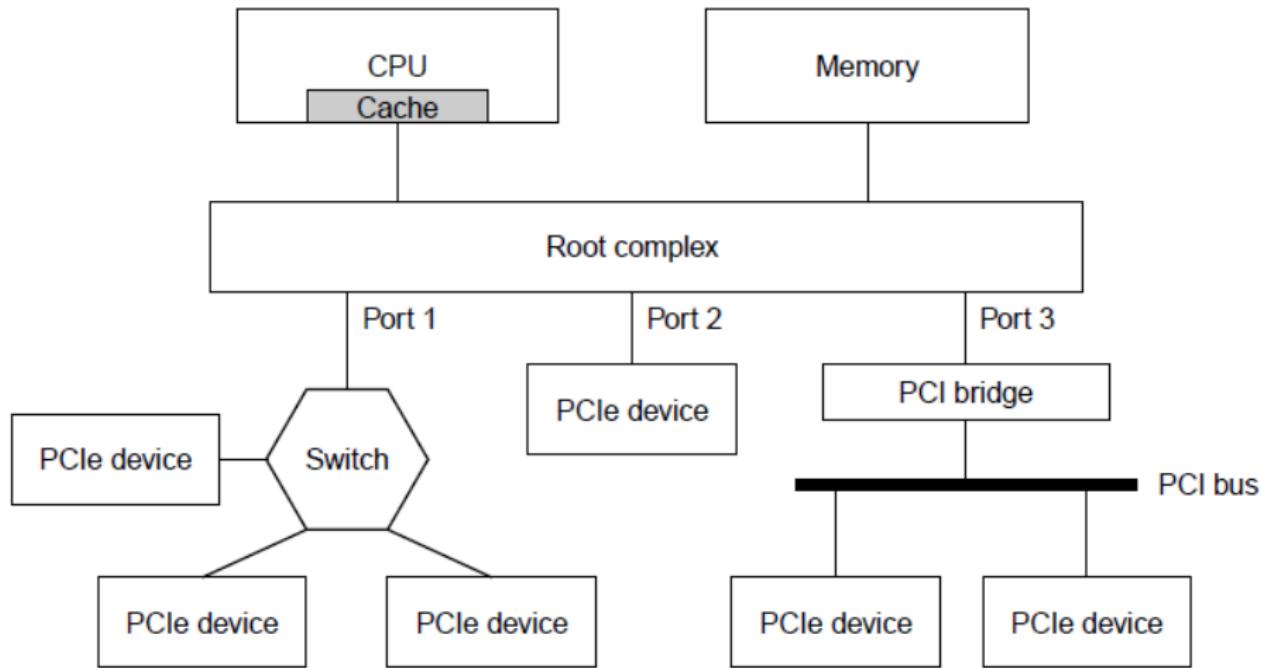
Strutturalmente molto diverso da PCI

- **seriale**: una linea di input e una di output, banda 2.5, 5, 8, 16 Gb/s
- connessione **punto a punto**: collegamento indipendente per ogni dispositivo, configurazione a stella (**switch**)
- dispositivi con più connessioni: 2/4/8/12/16, dati distribuiti su più linee
- trasmissione dati **a pacchetto**: **sequenze** di bit.

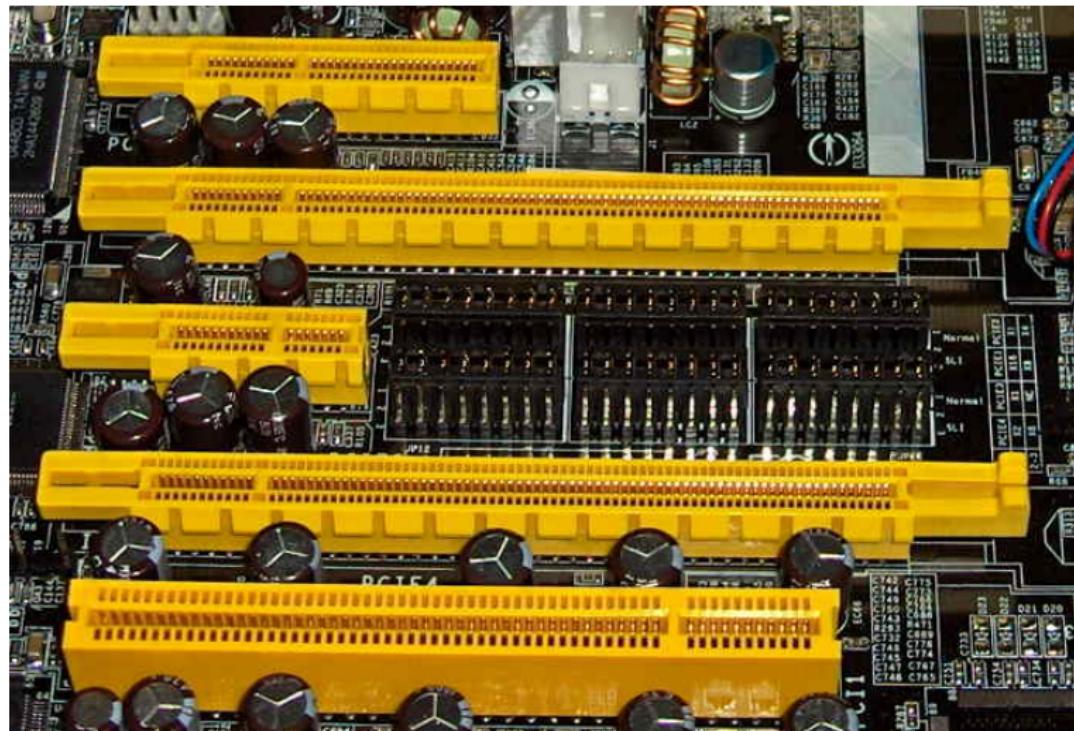
PCIe: struttura logica



PCIe: comunicazione



PCIe: slot di connessione

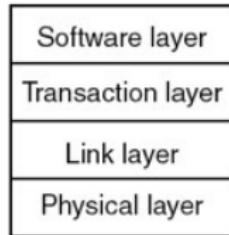


PCIe: funzionamento

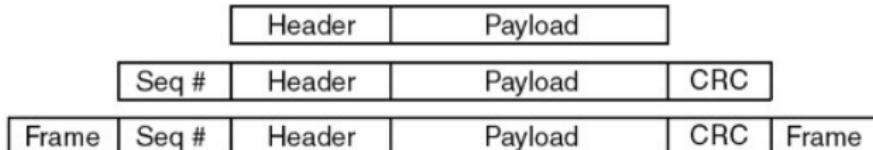
Comunicazione a più livelli (ispirata alle reti a pacchetto).

Ogni livello offre delle funzionalità al livello sovrastante.

Ogni livello introduce informazioni nel pacchetto.



(a)



(b)

PCIe: livelli di comunicazione

Livello fisico

- compatibilità tra slot e schede fisicamente diverse
- segnale differenziale con due linee intrecciate: meno sensibile alle interferenze
- bit di clock, codifica di bit 8b/10b (128b/130b da PCIe 3.0).

Livello di trasmissione

- trasmissione a pacchetto
- correzione degli errori (**codici a ridondanza ciclica**)
- riscontro (**acknowledgement**)
- gestione dell'interrupt.

PCIe: livelli di comunicazione

Livello di transazione

- sistema di **credit**i per adattare il flusso della comunicazione
- **circuiti virtuali**: suddividono la comunicazione tra due dispositivi su un certo numero di canali (fino a otto); comunicazioni di tipo diverso possono usare canali diversi
- denominato PCIe perché conserva il livello logico del bus PCI, uniformando le transazioni a livello di sistema operativo (retrocompatibilità).

PCIe: diffusione

Nelle schede madri coesistono bus PCI-Express, PCI, ATA (e ISA).

Parte delle periferiche continuano a usare i bus tradizionali.

PCI-Express ha sostituito il bus AGP.

Thunderbolt: bus dispositivi esterni (PCIe + Display Port).

Laboratorio di architettura degli elaboratori

CIRCUITI SEQUENZIALI Lezione 6

- CONTATTI
- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

Esercizio 6.1

Costruire un dispositivo sequenziale sincrono che riceve in ingresso una linea seriale sulla quale vengono trasmessi pacchetti di 3 bit. Il dispositivo genera come uscita un bit che indica se è stato spedito un pacchetto contenente la sequenza 110. L'uscita vale 0 in corrispondenza dei primi 2 bit di ogni pacchetto, sul terzo bit l'uscita vale 1 se è stata trasmessa la sequenza 110 e 0 altrimenti.

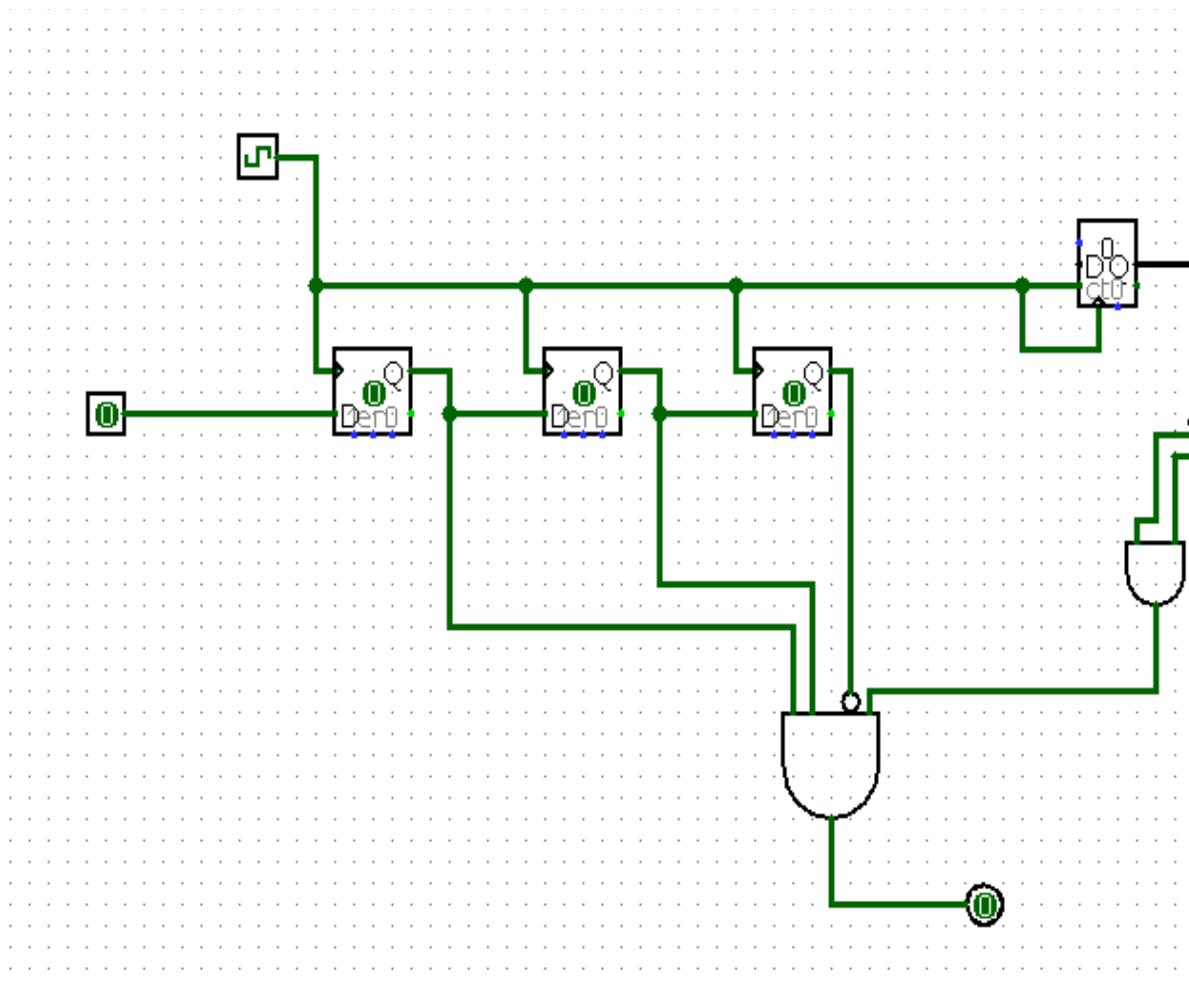
Esercizio 6.2

Utilizzando moduli half-adder e full-adder (esercizi 3.2.1, 3.2.2), costruire un circuito moltiplicatore; il circuito riceve in ingresso due numeri binari di 3 bit e genera in uscita il prodotto a 7 bit.

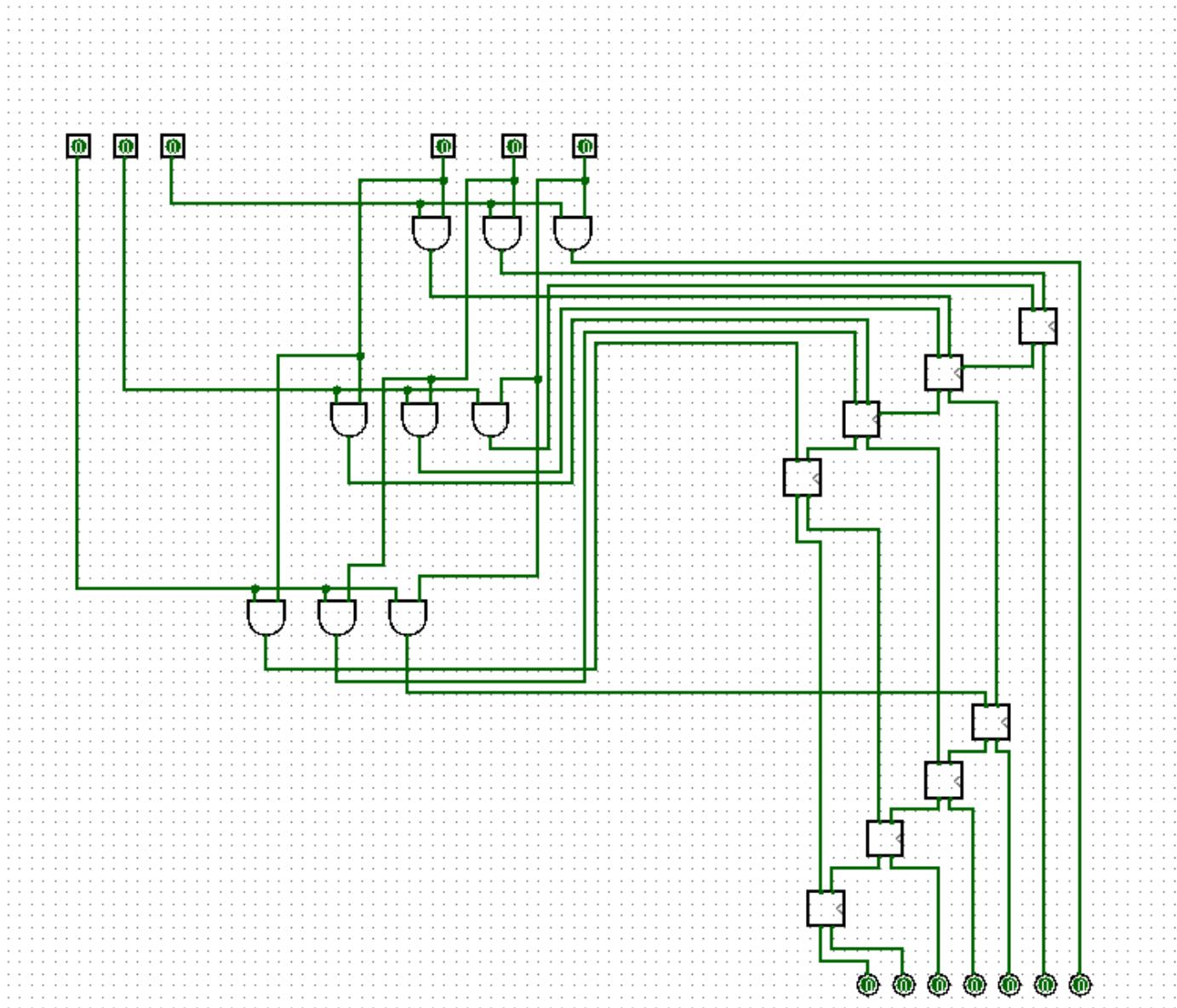
Esercizio 6.3

Modificare l'esercizio 5.2 (ascensore a due piani) in modo che la chiamata si disabiliti automaticamente quando l'ascensore raggiunge il piano selezionato. Implementare la chiamata utilizzando i “button” di Logisim.

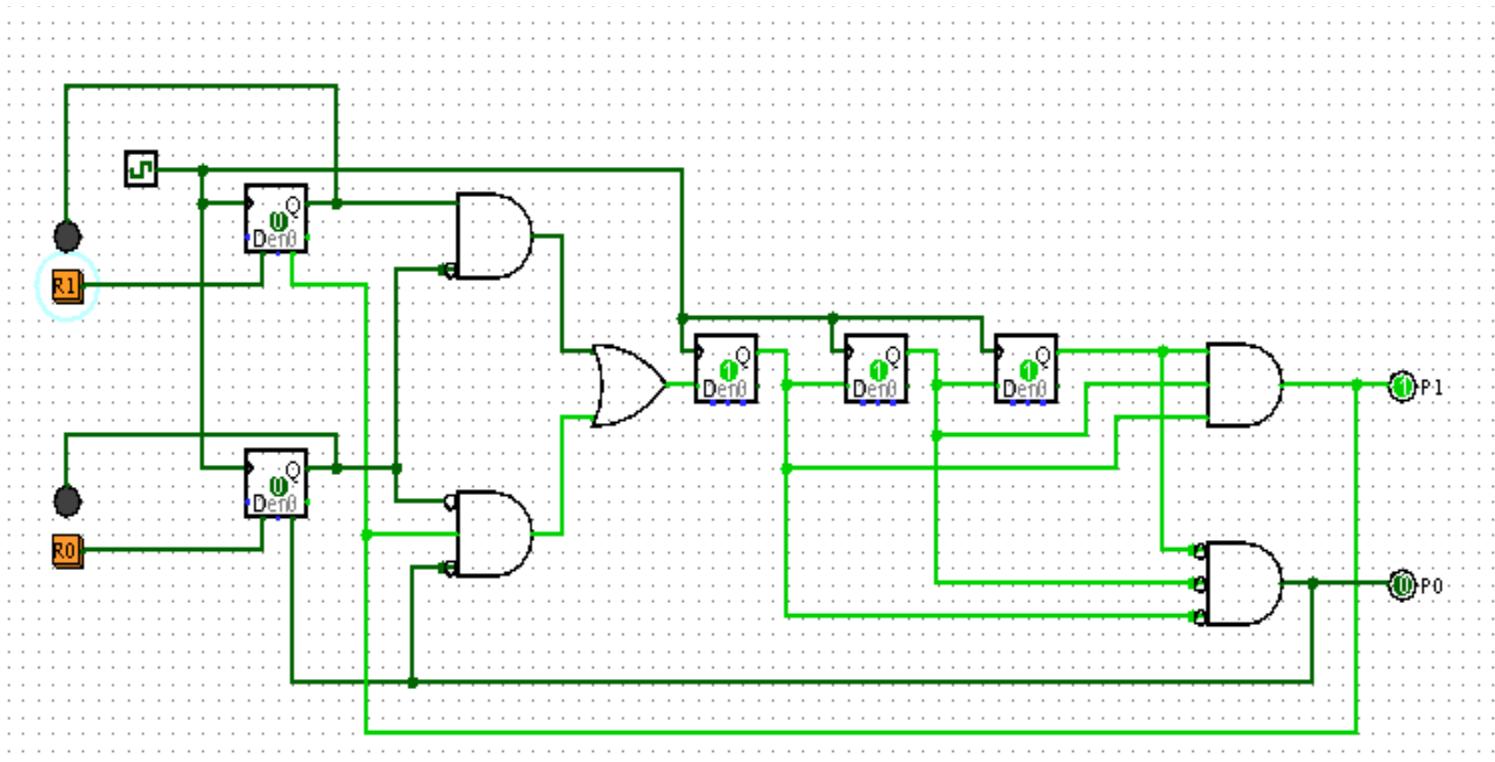
Esercizio 6.1



Esercizio 6.2



Esercizio 6.3



Laboratorio di architettura degli elaboratori

CIRCUITI SEQUENZIALI Lezione 6

- CONTATTI
- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

Esercizio 6.1

Costruire un dispositivo sequenziale sincrono che riceve in ingresso una linea seriale sulla quale vengono trasmessi pacchetti di 3 bit. Il dispositivo genera come uscita un bit che indica se è stato spedito un pacchetto contenente la sequenza 110. L'uscita vale 0 in corrispondenza dei primi 2 bit di ogni pacchetto, sul terzo bit l'uscita vale 1 se è stata trasmessa la sequenza 110 e 0 altrimenti.

Esercizio 6.2

Utilizzando moduli half-adder e full-adder (esercizi 3.2.1, 3.2.2), costruire un circuito moltiplicatore; il circuito riceve in ingresso due numeri binari di 3 bit e genera in uscita il prodotto a 7 bit.

Esercizio 6.3

Modificare l'esercizio 5.2 (ascensore a due piani) in modo che la chiamata si disabiliti automaticamente quando l'ascensore raggiunge il piano selezionato. Implementare la chiamata utilizzando i “button” di Logisim.

Controllo di flusso

Nei linguaggi ad alto livello il flusso dell'esecuzione è controllato dai costrutti

- **condizione** (if-then-else)
- **iterazione condizionata** (while, repeat)
- **iterazione incondizionata** (for).

In assembly i meccanismi di controllo del flusso sono elementari:

- **salto incondizionato**
- **salto condizionato.**

Salto incondizionato (branch)

b etichetta

salta all'istruzione etichettata con etichetta.

L'assemblatore crea un'associazione
etichetta-indirizzo:

label: add r0, r1, r1

Istruzioni condizionate

- Le istruzioni ARM possono essere eseguite **sotto condizione**.
- La condizione dipende da 4 bit contenuti nel registro di stato cprs.
- Inserendo il suffisso s al nome di un'istruzione, il registro di stato viene modificato a seguito della sua esecuzione.
- La condizione viene specificata da un ulteriore suffisso di due lettere. Es: ne
add addne addnes

Esempi

```
subs r0, r1, r2  
addeq r2, r2, #1  
beq label
```

L'Istruzione **sub**s modifica il registro cprs.

Se il risultato di **subs r0, r1, r2** è uguale a zero,
le istruzioni **addeq r2, r2, #1** e **beq label** sono eseguite.

Elenco condizioni

Le condizioni considerano 4 bit (**flag**):

- Z zero
- C carry
- N negative
- V overflow

Suffix	Description	Flags
eq	Equal / equals zero	Z
ne	Not equal	!Z
cs / hs	Carry set / uns. higher or same	C
cc / lo	Carry clear / unsigned lower	!C

Elenco condizioni

Suffix	Description	Flags
mi	Minus / negative	N
pl	Plus / positive or zero	!N
vs	Overflow	V
vc	No overflow	!V
hi	Unsigned higher	C and !Z
ls	Unsigned lower or same	!C or Z
ge	Signed greater than or equal	N == V
lt	Signed less than	N != V
gt	Signed greater than	!Z and (N == V)
le	Signed less than or equal	Z or (N != V)
al	Always (default)	any

Istruzioni di confronto

Modificano solo i flag del registro di stato:

- **compare** `cmp r0, r1`, confronta r0 con r1, aggiorna i flag come `subs r r0 r1`
- **compare negated** `cmn r0, r1`, confronta r0 con -r1, aggiorna i flag come `adds r r0, r1`
- **test** `tst r0, r1`, aggiorna i flag come `ands r r0 r1`
- **test equal** `teq r0 r1`, aggiorna i flag come `eors r r0 r1`.

Es.:

```
cmp r2, #7
```

```
ble label
```

salta a label solo se r2 è minore o uguale a 7.

Esempio: istruzione di test

```
if (i == j) then i = i + 1 else i = j fi
```

traduzione: i, j \Rightarrow r1, r2

```
        cmp r1, r2
        beq then
        mov r1, r2
        b fine
then:   add r1, r1, #1
fine:
```

codice alternativo **senza** salti:

```
        cmp r1, r2
        addeq r1, r1, #1
        movne r1, r2.
```

if-then-else in generale

```
if Bool then Com1 else Com2 fi
```

viene tradotta in:

```
Eval Bool
```

```
b_cond then
```

```
Com2
```

```
b fine
```

```
then: Com1
```

```
fine:
```

if-then in generale

if Bool then Com fi

viene tradotta in:

Eval not Bool

b_cond fine

Com

fine:

Esempio: ciclo

```
while (i != 0) do i = i - 1, j = j + 1 od
```

traduzione: i, j ⇒ r1, r2

```
while: cmp r1, #0
      beq fine
      sub r1, r1, #1
      add r2, r2, #1
      b while
fine:
```

while-do in generale

while Bool do Com od

viene tradotta secondo lo schema:

```
while:  Eval not Bool  
        b_cond fine  
        Com  
        b while
```

fine:

Assembly e linguaggio macchina

Linguaggio assembly: sintassi usata dal programmatore per scrivere, analizzare e rappresentare programmi in linguaggio macchina.

Linguaggio macchina: sintassi utilizzata dal calcolatore per memorizzare ed eseguire programmi. Ogni istruzione è rappresentata mediante una sequenza di bit.

Assembly e linguaggio macchina

L'istruzione assembly `subs r1, r2, r3` corrisponde alla seguente istruzione macchina:

cond	opcode	S	rn	rd	shift	2arg
al	sub	t	r2	r1	lsl 0	r3
1110	0000010	1	0010	0001	00000000	0011

Esiste una corrispondenza 1 a 1 tra istruzioni assembly e istruzioni macchina.

In ARM, ogni istruzione macchina utilizza 32 bit.

Istruzioni macchina di salto

- L'istruzione di salto

beq label

riserva 24 bit per specificare l'indirizzo di memoria a cui saltare.

- Il salto è **relativo**: si specificano quante istruzioni (di 32 bit!) saltare in avanti o indietro.
- Al registro r15 (program counter) viene sommato un numero intero (23 bit con segno).
- Numeri negativi: salti all'indietro.
- Indirizzi raggiungibili: $\pm 2^{23} \cdot 4 = \pm 32 \text{ MB}$.

Esercizi

- Calcolare e scrivere nel registro r1 l' n -esimo numero di Fibonacci $F(n)$, con n contenuto nel registro r0.

Nota: $F(0) = 0$, $F(1) = 1$,

$$F(n) = F(n - 1) + F(n - 2), n > 1.$$

- Calcolare, e scrivere nel registro r1 la somma degli elementi di un vettore V con 10 elementi, con indirizzo base il valore di r0,

Moltiplicazione estesa

Unsigned multiplication long: `umull r0, r1, r2, r4`

Calcola il prodotto di r2 e r4; deposita il risultato a 64 bit nei registri r1 (cifre più significative) e r0 (cifre meno significative).

Signed multiplication long: `smull r0, r1, r2, r4`

Calcola il prodotto in complemento a due tra r2 e r4; deposita il risultato a 64 bit nei registri r1 e r0.

Non esistono istruzioni per la divisione di interi.
Aritmetica floating-point disponibile attivando la
Floating Point Unit (FPU): NEON & VFP
programming.

Funzioni, metodi, subroutine

Costrutti base della programmazione.

- Strutturano il programma in parti invocabili.
- Modularizzano i programmi.
- Permettono il riutilizzo del codice.
- Necessarie per gestire codice complesso.
- Necessarie per realizzare la **ricorsione**.

Utilizzate in programmazione assembly.

Supportate dai linguaggi macchina, ma non automaticamente disponibili in assembly: una **chiamata di procedura** corrisponde a una sequenza di istruzioni macchina **non risolte automaticamente da istruzioni assembly**.

Procedure

Una chiamata di procedura comporta:

- passaggio del controllo al codice della procedura,
- passaggio di parametri,
- allocazione di spazio di memoria per le variabili locali.

L'uscita da una procedura comporta:

- recupero di spazio in memoria,
- restituzione del controllo e del risultato al chiamante.

Implementazione in ARM con istruzioni elementari.

Mostriamo come le chiamate di procedura vengono realizzate nella macchina.

Chiamata di procedure

Salto con memorizzazione dell'indirizzo di ritorno:

`bl label` branch and link

salta all'indirizzo di etichetta `label` e salva nel `link register lr` (r14) l'indirizzo di ritorno (cioè l'istruzione successiva a `bl label`).

Rientro dalla procedura:

`mov pc, lr`

Esempio: funzione fattoriale $n!$

$$n! = n \cdot (n - 1) \cdot \dots \cdot 2 \cdot 1$$

```
.text
main:      ...
          bl fattoriale
          ...
fattoriale: ...
          ...
          mov pc, lr
fibonacci: ...
          ...
          ...
```

Passaggio di parametri e risultati

Si usano i registri per il passaggio dei parametri (se pochi). **Convenzione** sull'uso dei registri:

- $r0, \dots, r3$ sono utilizzati per passare argomenti a una procedura.
- $r0, r1$ sono utilizzati per restituire al programma chiamante i valori risultato di una procedura.

Eventuali parametri aggiuntivi devono essere passati **attraverso la memoria**.

Esempio: funzione fattoriale

```
.text
main:    ...
          mov r0, #5
          bl fattoriale
          ...
fattoriale: ...
          movs r1, r0
          beq end
          ...
          mul r0, r4, r0
          mov pc, lr
fibonacci: ...
          ...
```

Interferenza sui registri

Programma principale e procedura agiscono sullo stesso insieme di registri.

Bisogna evitare interferenze improprie:

main:

```
    ...  
    add r4, r4, #1  
    mov r0, #5  
    bl fattoriale
```

```
    ...
```

fattoriale:

```
    ...  
    ...  
    move r4, r0  
    ...  
    mov pc, lr
```

Salvataggio dei registri

- Per convenzione, i registri r4-r14 devono essere **preservati** dalle procedure
- se una procedura vuole utilizzarli
 - li salva in memoria prima del loro utilizzo
 - ne ripristina il valore originale prima di restituire il controllo al programma chiamante.

Dualmente

- i registri r0-r3 sono considerati **modificabili** dalle procedure
- se contengono dati utili, il programma chiamante
 - li salva in memoria prima di chiamare una procedura
 - li ripristina dopo la chiamata.

Esempio: salvataggio di r4-r14

main:

```
    ...  
    add r4, r4, #1  
    mov r0, #5  
    bl fattoriale
```

```
    ...
```

fattoriale:

```
    ...  
    stmfd sp!, {r4-r5}  
    ...  
    move r4, r0  
    ...  
    ldmfd sp!, {r4-r5}  
    mov pc, lr
```

Esempio: salvataggio di r0-r3

```
main:    ...
        add r2, r2, #1
        mov r0, #5
        stmfd sp!, {r2,r3}
        bl fattoriale
        ldmfd sp!, {r2-r3}

        ...

fattoriale: ...
        ...
        move r2, r0
        ...
        mov pc, lr
```

Istruzioni di load e store multipli

`stmfd sp!, {r0, r4-r6, r3}`

- salva in locazioni decrescenti di memoria,
- a partire dall'indirizzo in $sp - 4$ ($r13 - 4$),
- il contenuto dei registri $r0, r4, r5, r6, r3$,
- aggiorna sp alla locazione contenente l'ultimo valore inserito:

$$r1 = r13 - 5*4$$

`ldmfd sp!, {r0, r4-r6, r3}`

ripristina il contenuto di tutti i registri (compreso sp).

Suffissi in load e store multiple

Il comando `stm` è spesso usato per manipolare lo stack.

Il registro `r13` punta alla **cima** dello stack.

Più possibilità:

- `stmia r13!, {...}` **incrementa** da `sp`
- `stmib r13!, {...}` **incrementa** da `sp+4`
- `stmda r13!, {...}` **decrementa** da `sp`
- `stmdb r13!, {...}` **decrementa** da `sp-4`.

Esistono i suffissi corrispondenti: `fd`, `fu`, `ed`, `eu`.

Allocazione spazio di memoria

Ogni procedura necessita di un'area di memoria per

- mantenere le variabili locali
- salvare i registri
- acquisire i parametri e restituire i risultati.

Tutta l'area è allocata in un frame dello stack.

Chiamate innestate generano una pila di frame consecutivi:

- una **chiamata** di procedura alloca un nuovo frame
- un'**uscita** dalla procedura libera l'ultimo frame.

Last in - first out: la procedura chiamata più recentemente è la prima a terminare.

Laboratorio di architettura degli elaboratori

CIRCUITI SEQUENZIALI Lezione 7

- CONTATTI
- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

Esercizio 7.1

Utilizzando i moduli per l'aritmetica progettati precedentemente, realizzare un circuito che ricevuti in ingresso due numeri interi, m ed n rappresentati in complemento a 2 con 4 bit e un ingresso di controllo s calcoli:

- La somma tra m ed n se $s = 0$.
- La differenza tra m ed n se $s = 1$.

Il circuito deve generare sia risultato a 4 bit che un segnale di eventuale overflow.

Esercizio 7.2

Un flip-flop di tipo T è un flip-flop con un singolo segnale di controllo T (oltre al segnale di clock) avente il seguente comportamento: se il segnale di T ha valore 0 il flip-flop non cambia stato, mentre se il segnale T ha valore 1 il flip-flop cambia stato (toggle) ad ogni segnale di clock. Utilizzando un flip-flop di tipo D e un circuito multiplexer, costruire un flip-flop di tipo T.

Per risolvere l'esercizio bisogna sfruttare il fatto che un flip-flop di tipo di D è provvisto di 2 uscite, Q e Q^{\wedge} .

Esercizio 7.3

Costruire un circuito che trasformi un segnale parallelo in uno seriale. Il circuito ha 4 linee di ingresso ed una di uscita. Durante il funzionamento le linee di ingresso vengono modificate ogni 4 cicli di clock. Tra una modifica dell'ingresso e la successiva, la linea di uscita, in quattro cicli di clock consecutivi, assume ciascuno dei 4 valori presenti nelle linee di ingresso.

Universal Serial Bus (USB)

Bus per il collegamento di periferiche esterne.
Sviluppato dal '95 dal consorzio USB Implementers Forum (USB-IF), a cui si sono aggiunte via via grandi compagnie. Enfasi sul computer, non sui dispositivi elettronici.

Obiettivi:

- economicità
- semplicità e flessibilità di utilizzo
 - un unico bus per tutte le periferiche
 - accesso a slot interni non più necessario
 - espandibilità
 - connettività “a caldo”
- supporto dispositivi a tempo reale (audio).

USB: evoluzione

Larghezza di banda crescente con gli aggiornamenti.

- USB 1.0: 1.5 Mb/s (1995)
- USB 1.1: 12 Mb/s (1998)
- USB 2.0: 480 Mb/s (2001)
- USB 3.0: 4.8 Gb/s (2010)
- USB 3.1: 10 Gb/s (2013)
- USB Type-C: più canali USB 3.1 (2015).

USB: connessioni

Bus seriale

- fino a USB 2.0, 4 connessioni:
2 linee per ingresso o uscita (**half-duplex**, differenziale),
1 linea di alimentazione a 5 V, 1 linea di massa
- USB 3.0 aggiunge una linea realizzando il **full-duplex**: 1 massa, 1 linea di input, 1 linea di output.
- USB Type-C (24 linee): 2 canali a bassa velocità, 4 canali ad alta velocità, configurazione e alimentazione; più tensioni di alimentazione, fino a $20\text{ V} \times 5\text{ A} = 100\text{ W}$ (! difficile da sostenere per i laptop).

USB: struttura della rete

Ad albero. Fino a 127 nodi

- radice (root hub) - connessa a bus PCI (o al South-Bridge)
- nodi hub di espansione (USB bay) - permettono ramificare le connessioni
- foglie periferiche - tastiera, mouse, scanner, webcam, memoria esterna, interfaccia audio...
- esistono anche “sharing hubs”.

USB: collegamento a caldo

Quando una nuova foglia viene inserita

- root hub identifica l'evento
- lancia un interrupt al sistema operativo
- interrogazione dispositivo: tipo di dispositivo, banda richiesta
- assegnazione indirizzo unico (1-127).

USB: comunicazione

- Connessione esclusivamente da root a dispositivo: le periferiche **non** comunicano tra loro.
- Comunicazione distribuita su diverse linee logiche (circuiti virtuali):
fino a 16 input + 16 output per dispositivo,
circuiti virtuali diversi possono trasmettere tipi di dati diversi.

USB: framing

La comunicazione è organizzata in **frame** (pacchetti strutturati di dati).

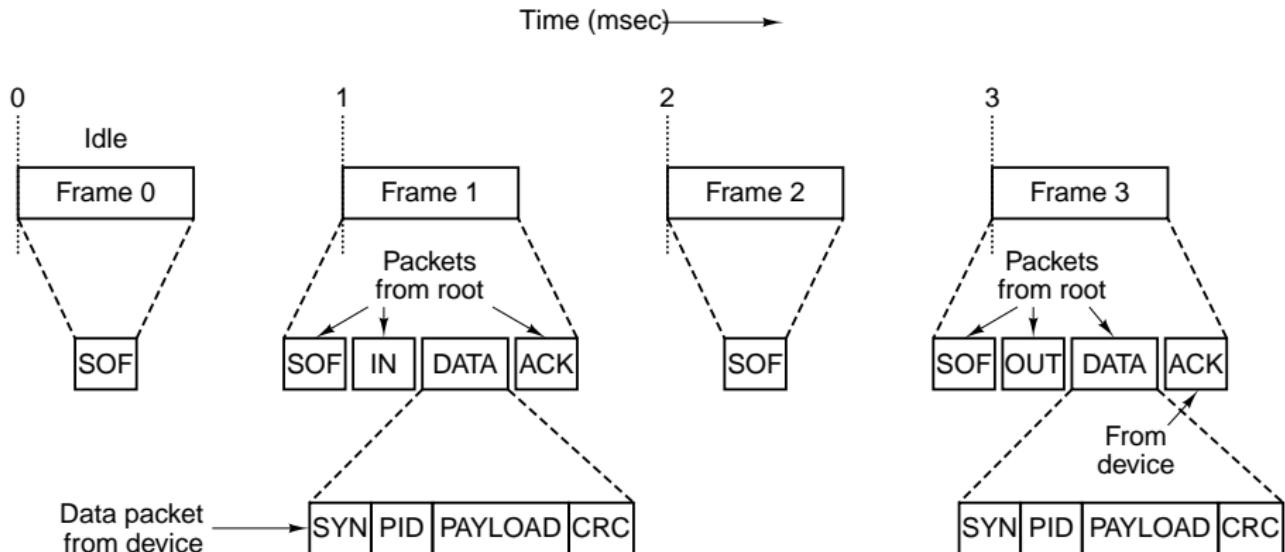
Per mantenere la sincronizzazione tra gli orologi interni, esattamente ogni 1 msec root hub invia un frame.

Trasmissione in **broadcast**: inviata a tutti i dispositivi.

4 tipi di frame:

- **control** comandi al dispositivo/diagnostica
- **bulk** dati
- **isochronous** dispositivi tempo reale
- **interrupt** simulazione interrupt.

USB: struttura dei frame

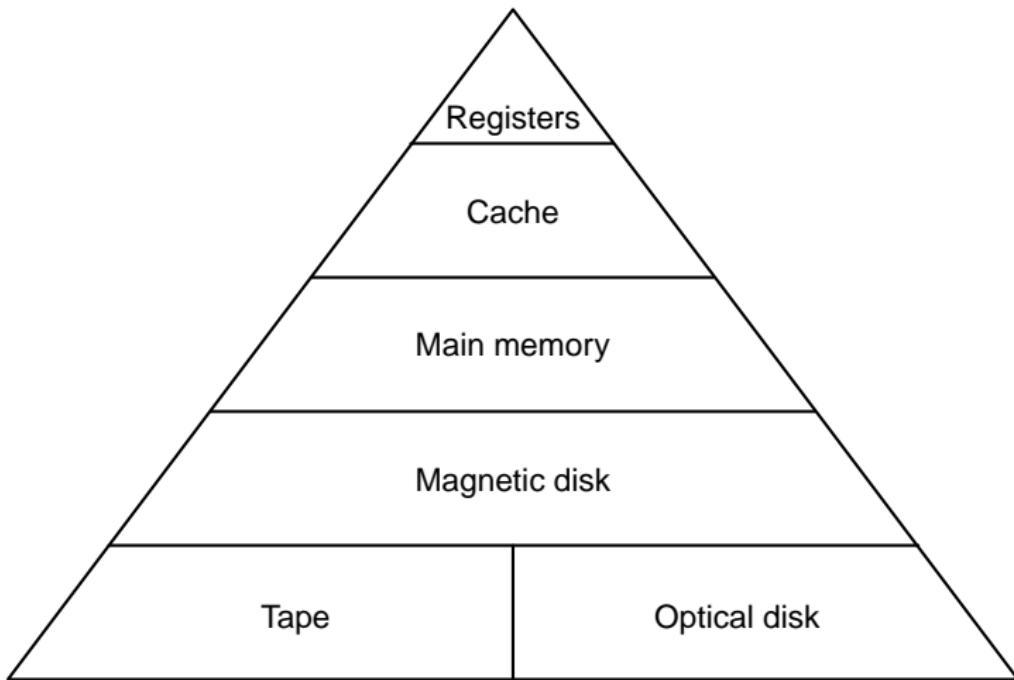


USB: struttura dei frame

Ogni frame contiene dei sottopacchetti

- **token** (da root al dispositivo) per il controllo della comunicazione: SOF, IN, OUT, SETUP
- **data** (nelle due direzioni) con formato: sincronizzazione, tipo di pacchetto, dati, controllo
- **handshake**: ACK, NAK, STALL
- **special**.

Connessioni tra livelli di memoria



Desiderata: memorie veloci e capienti

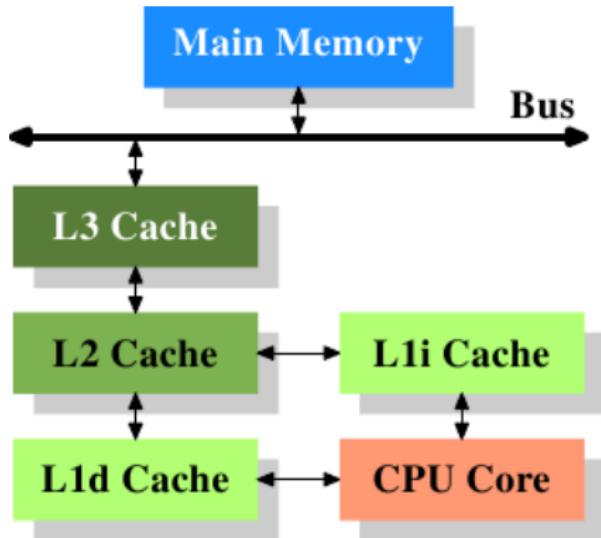
- I dati in uso risiedono nella memoria veloce.
- Le memorie capienti devono contenere il resto dei dati.

Spostamento dei dati

- **esplicito**: caricamento di un programma, scrittura di un registro
- **trasparente**: non visibile ai livelli più alti, gestito da hardware e sistema operativo.

Memoria cache e virtuale

- Memoria cache: tra registri e memoria principale.
- Memoria **virtuale**: tra memoria principale e memoria di massa.



Accesso alla memoria

La linea di cache contenente il dato è presente in memoria?

- sì (**cache hit**): accedo al dato
- no (**cache miss**):
 - scarico una linea dalla memoria cache
 - carico la linea contenente il dato
 - accedo al dato.

Rapporto hit/miss

La memoria cache è giustificata se i cache miss sono poco frequenti.

Dati:

h probabilità di cache hit

t_c tempo di accesso alla cache

t_p tempo di accesso alla memoria principale

Allora:

il tempo medio t_M di accesso alla memoria è

$$t_M = t_c + (1 - h) \times t_p$$

La cache è conveniente se $t_M < t_p$, dunque se

$$h > t_c / t_p$$

Memoria cache: politica

- Obiettivo: mantenere nella cache i dati che saranno più probabilmente usati.
- Metodo: si sfruttano due principi statistici
 - **località temporale**: i dati usati recentemente hanno maggior probabilità di essere usati
 - **località spaziale**: dati contigui a dati usati recentemente hanno buona probabilità di essere usati.

Memoria cache: linee

Si tengono in cache i dati utilizzati di recente e i dati contigui a questi.

- La memoria viene divisa in **linee di cache** (tipicamente 32-64 byte).
- La cache mantiene le linee utilizzate più di recente.

Memoria cache: tipologie

- Cache ad **accesso diretto**.
- Cache **associativa a n vie**.
- Ma anche cache associative, cache 2-way skewed associative . . .

Cache ad accesso diretto

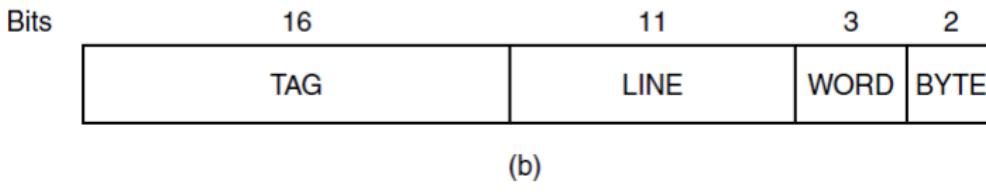
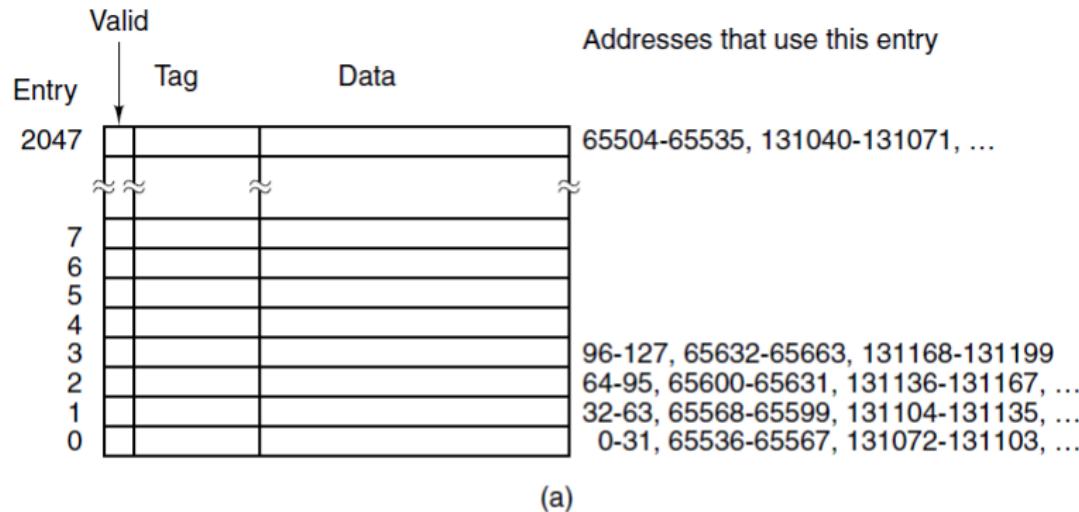
Ogni elemento della cache può memorizzare **una linea di cache** della memoria centrale (es.: 32 byte).

La posizione nella cache è determinata univocamente dall'indirizzo dei dati in memoria.

L'indirizzo **virtuale** contiene diverse parti:

- bit di **validità** dei dati nella linea
- **line**: individuato dalla posizione fisica della linea tra quelle possibili
- **TAG**: individua la posizione in memoria della linea
- **Word(+Byte)**: individuano il word (ed eventualmente il byte) all'interno della linea di cache.

Cache ad accesso diretto



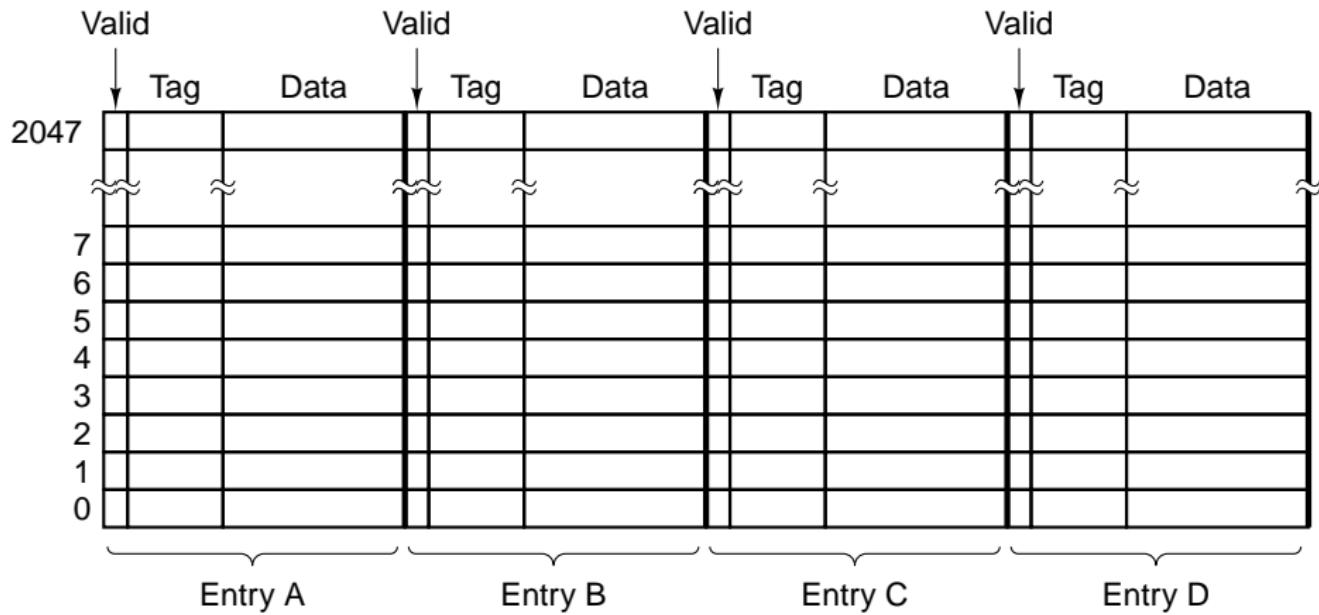
Accesso diretto: vantaggi/svantaggi

- vantaggi: semplice da implementare, veloce
- svantaggi: meccanismo rigido,
non è possibile scegliere la linea da scartare (in
base alla località temporale),
in alcuni casi genera un elevato numero di
cache miss.

Es. (critico): un programma accede
continuamente a due locazioni di memoria
 contenute in due linee distinte che mappano
sullo stesso elemento di cache (identico campo
line).

Cache: (n -way) set-associative

Unione di n tabelle di cache ad accesso diretto.
Ogni dato contenuto in una qualsiasi tabella



Cache set-associative: pro e contro

- Più flessibile dell'accesso diretto,
è possibile scegliere tra più alternative in cui inserire la linea di cache,
- Vengono evitati i casi più critici descritti della cache ad accesso diretto.
- Quale linea scartare dalle tabelle?
- Più complessa da realizzare: richiede più hardware.
- Architetture attuali:
 - Core i7: 4/8/16-way associative (a seconda del livello)
 - ARM Cortex A15: 2/16-way associative.

Set-associative: rimozione linee

- Quale linea scartare per far spazio a una nuova linea?
- In base alla località temporale, quella usata meno recentemente: **least recently used** (LRU).
- Implementazione: per ogni elemento della cache, una lista descrive l'ordine di accesso alle varie tavole.

Cache: scrittura dati

La scrittura dei dati è temporalmente meno critica rispetto alla lettura.

Infatti, non è necessariamente un'operazione bloccante: dopo un comando di scrittura il processore non deve attendere il completamento dell'operazione e può procedere immediatamente all'istruzione successiva.

Si costruisce una coda di operazioni di scrittura in memoria ancora da eseguire.

Scrittura: linea nella cache

Cosa fare in caso di scrittura di un dato in cache?

Due alternative:

- **Write through**: si scrive nella cache e nella memoria principale.

Logicamente semplice: mantiene la coerenza dei dati ma genera più accessi alla memoria.

- **Write deferred – Write back**: si scrive solo in cache e si aggiorna la memoria principale quando la linea cache viene scaricata (verifica del **dirty bit**).

Più efficiente, meno accessi in memoria, richiede di prevenire inconsistenze tra cache e memoria principale.

Scrittura: linea non nella cache

Cosa fare in caso di scrittura di un dato non in memoria cache?

Due alternative:

- **no-write allocate**: si modifica solo la memoria principale,
- **write allocate**: il dato viene prima portato nella memoria cache.

Usualmente

- write through si accompagna a no-write allocate
- write back si accompagna a write allocate.

5

ARM Processor Instruction Set

This chapter describes the ARM processor instruction set.

5.1	Instruction Set Summary	5-2
5.2	The Condition Field	5-2
5.3	Branch and Branch with Link (B, BL)	5-3
5.4	Data Processing	5-4
5.5	PSR Transfer (MRS, MSR)	5-13
5.6	Multiply and Multiply-Accumulate (MUL, MLA)	5-16
5.7	Single Data Transfer (LDR, STR)	5-18
5.8	Block Data Transfer (LDM, STM)	5-24
5.9	Single Data Swap (SWP)	5-32
5.10	Software Interrupt (SWI)	5-34
5.11	Coprocessor Instructions on the ARM Processor	5-36
5.12	Coprocessor Data Operations (CDP)	5-36
5.13	Coprocessor Data Transfers (LDC, STC)	5-38
5.14	Coprocessor Register Transfers (MRC, MCR)	5-41
5.15	Undefined Instruction	5-43
5.16	Instruction Set Examples	5-44
5.17	Instruction Speed Summary	5-47



ARM Processor Instruction Set

5.1 Instruction Set Summary

A summary of the ARM processor instruction set is shown in *Figure 5-1: Instruction set summary*.

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data Processing PSR Transfer	cond	0	0	I	opcode				S	Rn		Rd	operand 2																			
Multiply	cond	0	0	0	0	0	0	A	S	Rd		Rn	Rs		1	0	0	1	Rm													
Single data swap	cond	0	0	0	1	0	B	0		Rn		Rd	0		0	0	0	0	1	0	0	1	Rm									
Single data transfer	cond	0	1	I	P	U	B	W	L	Rn		Rd	offset																			
Undefined instruction	cond	0	1	1	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	1	x	x	x			
Block data transfer	cond	1	0	0	P	U	S	W	L	Rn		Register List																				
Branch	cond	1	0	1	L	offset																										
Coproc data transfer	cond	1	1	0	P	U	N	W	L	Rn		CRd	cp_num		offset																	
Coproc data operation	cond	1	1	1	0	CP opc				CRn		CRd	cp_num		CP	0	CRm															
Coproc register transfer	cond	1	1	1	0	CP opc		L	CRn		Rd	cp_num		CP	1	CRm																
Software interrupt	cond	1	1	1	1	ignored by processor																										

Figure 5-1: Instruction set summary

Note: Some instruction codes are not defined but do not cause the Undefined instruction trap to be taken; for instance, a Multiply instruction with bit 6 changed to a 1. These instructions shall not be used, as their action may change in future ARM implementations.

5.2 The Condition Field

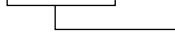
31	28	27	0																																
cond																																			
																																			
Condition Field																																			
<table><tbody><tr><td>0000 = EQ (equal)</td><td>- Z set</td></tr><tr><td>0001 = NE (not equal)</td><td>- Z clear</td></tr><tr><td>0010 = CS (unsigned higher or same)</td><td>- C set</td></tr><tr><td>0011 = CC (unsigned lower)</td><td>- C clear</td></tr><tr><td>0100 = MI (negative)</td><td>- N set</td></tr><tr><td>0101 = PL (positive or zero)</td><td>- N clear</td></tr><tr><td>0110 = VS (overflow)</td><td>- V set</td></tr><tr><td>0111 = VC (no overflow)</td><td>- V clear</td></tr><tr><td>1000 = HI (unsigned higher)</td><td>- C set and Z clear</td></tr><tr><td>1001 = LS (unsigned lower or same)</td><td>- C clear or Z set</td></tr><tr><td>1010 = GE (greater or equal)</td><td>- N set and V set, or N clear and V clear</td></tr><tr><td>1011 = LT (less than)</td><td>- N set and V clear, or N clear and V set</td></tr><tr><td>1100 = GT (greater than)</td><td>- Z clear, and either N set and V set, or N clear and V clear</td></tr><tr><td>1101 = LE (less than or equal)</td><td>- Z set, or N set and V clear, or N clear and V set</td></tr><tr><td>1101 = AL</td><td>- always</td></tr><tr><td>1111 = NV</td><td>- never</td></tr></tbody></table>				0000 = EQ (equal)	- Z set	0001 = NE (not equal)	- Z clear	0010 = CS (unsigned higher or same)	- C set	0011 = CC (unsigned lower)	- C clear	0100 = MI (negative)	- N set	0101 = PL (positive or zero)	- N clear	0110 = VS (overflow)	- V set	0111 = VC (no overflow)	- V clear	1000 = HI (unsigned higher)	- C set and Z clear	1001 = LS (unsigned lower or same)	- C clear or Z set	1010 = GE (greater or equal)	- N set and V set, or N clear and V clear	1011 = LT (less than)	- N set and V clear, or N clear and V set	1100 = GT (greater than)	- Z clear, and either N set and V set, or N clear and V clear	1101 = LE (less than or equal)	- Z set, or N set and V clear, or N clear and V set	1101 = AL	- always	1111 = NV	- never
0000 = EQ (equal)	- Z set																																		
0001 = NE (not equal)	- Z clear																																		
0010 = CS (unsigned higher or same)	- C set																																		
0011 = CC (unsigned lower)	- C clear																																		
0100 = MI (negative)	- N set																																		
0101 = PL (positive or zero)	- N clear																																		
0110 = VS (overflow)	- V set																																		
0111 = VC (no overflow)	- V clear																																		
1000 = HI (unsigned higher)	- C set and Z clear																																		
1001 = LS (unsigned lower or same)	- C clear or Z set																																		
1010 = GE (greater or equal)	- N set and V set, or N clear and V clear																																		
1011 = LT (less than)	- N set and V clear, or N clear and V set																																		
1100 = GT (greater than)	- Z clear, and either N set and V set, or N clear and V clear																																		
1101 = LE (less than or equal)	- Z set, or N set and V clear, or N clear and V set																																		
1101 = AL	- always																																		
1111 = NV	- never																																		

Figure 5-2: Condition codes

ARM Processor Instruction Set

All ARM processor instructions are conditionally executed, which means that their execution may or may not take place depending on the values of the N, Z, C and V flags in the CPSR.

The condition codes have meanings as detailed in *Figure 5-2: Condition codes*, for instance code 0000 (EQual) executes the instruction only if the Z flag is set. This would correspond to the case where a compare (CMP) instruction had found the two operands to be equal. If the two operands were different, the compare instruction would have cleared the Z flag and the instruction is not executed.

Note: *If the always (AL - 1110) condition is specified, the instruction will be executed irrespective of the flags. The never (NV - 1111) class of condition codes must not be used as they will be redefined in future variants of the ARM architecture. If a NOP is required it is suggested that MOV R0,R0 be used. The assembler treats the absence of a condition code as though always had been specified.*

5.3 Branch and Branch with Link (B, BL)

These instructions are only executed if the condition is true. The instruction encoding is shown in *Figure 5-3: Branch instructions*.

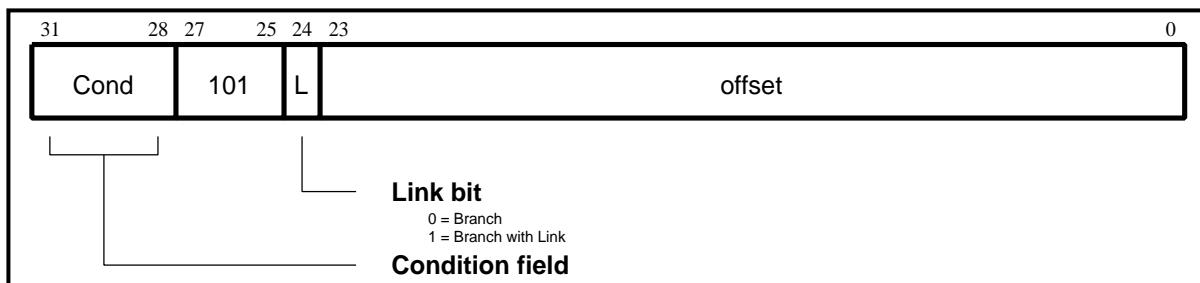


Figure 5-3: Branch instructions

Branch instructions contain a signed 2's complement 24-bit offset. This is shifted left two bits, sign extended to 32 bits, and added to the PC. The instruction can therefore specify a branch of +/- 32Mbytes. The branch offset must take account of the prefetch operation, which causes the PC to be 2 words (8 bytes) ahead of the current instruction. Branches beyond +/- 32Mbytes must use an offset or absolute destination which has been previously loaded into a register. In this case the PC should be manually saved in R14 if a branch with link type operation is required.

5.3.1 The link bit

Branch with Link (BL) writes the old PC into the link register (R14) of the current bank. The PC value written into R14 is adjusted to allow for the prefetch, and contains the address of the instruction following the branch and link instruction. Note that the CPSR is not saved with the PC.

To return from a routine called by Branch with Link use MOV PC,R14 if the link register is still valid or use LDM Rn!,{..PC} if the link register has been saved onto a stack pointed to by Rn.



ARM Processor Instruction Set

5.3.2 Instruction cycle times

Branch and Branch with Link instructions take 3 instruction fetches. For more information see *5.17 Instruction Speed Summary* on page 5-47.

5.3.3 Assembler syntax

B{L}{cond} <expression>

Items in {} are optional. Items in <> must be present.

{L}	requests the Branch with Link form of the instruction. If *absent, R14 will not be affected by the instruction.
{cond}	is a two-char mnemonic as shown in <i>Figure 5-2: Condition codes</i> on page 5-2 (EQ, NE, VS etc). If absent then AL (ALways) will be used.
<expression>	is the destination. The assembler calculates the offset.

5.3.4 Examples

```
here    BAL    here      ;assembles to 0xEFFFFFFE (note effect of PC  
;offset)  
        B      there     ;ALways condition used as default  
        CMP    R1,#0      ;compare R1 with zero and branch to fred if R1  
        BEQ    fred       ;was zero otherwise continue to next instruction  
        BL     sub+ROM    ;call subroutine at computed address  
        ADDS   R1,#1      ;add 1 to register 1, setting CPSR flags on the  
        BLCC   sub        ;result then call subroutine if the C flag is  
                      ;clear, which will be the case unless R1 held  
                      ;0xFFFFFFFF
```

5.4 Data Processing

The instruction is only executed if the condition is true, defined at the beginning of this chapter. The instruction encoding is shown in *Figure 5-4: Data processing instructions* on page 5-5.

The instruction produces a result by performing a specified arithmetic or logical operation on one or two operands.

First operand is always a register (Rn).

Second operand may be a shifted register (Rm) or a rotated 8-bit immediate value (Imm) according to the value of the I bit in the instruction.

The condition codes in the CPSR may be preserved or updated as a result of this instruction, according to the value of the S-bit in the instruction.

Certain operations (TST, TEQ, CMP, CMN) do not write the result to Rd. They are used only to perform tests and to set the condition codes on the result and always have the S bit set.

The instructions and their effects are listed in *Table 5-1: ARM data processing instructions* on page 5-6.

ARM Processor Instruction Set

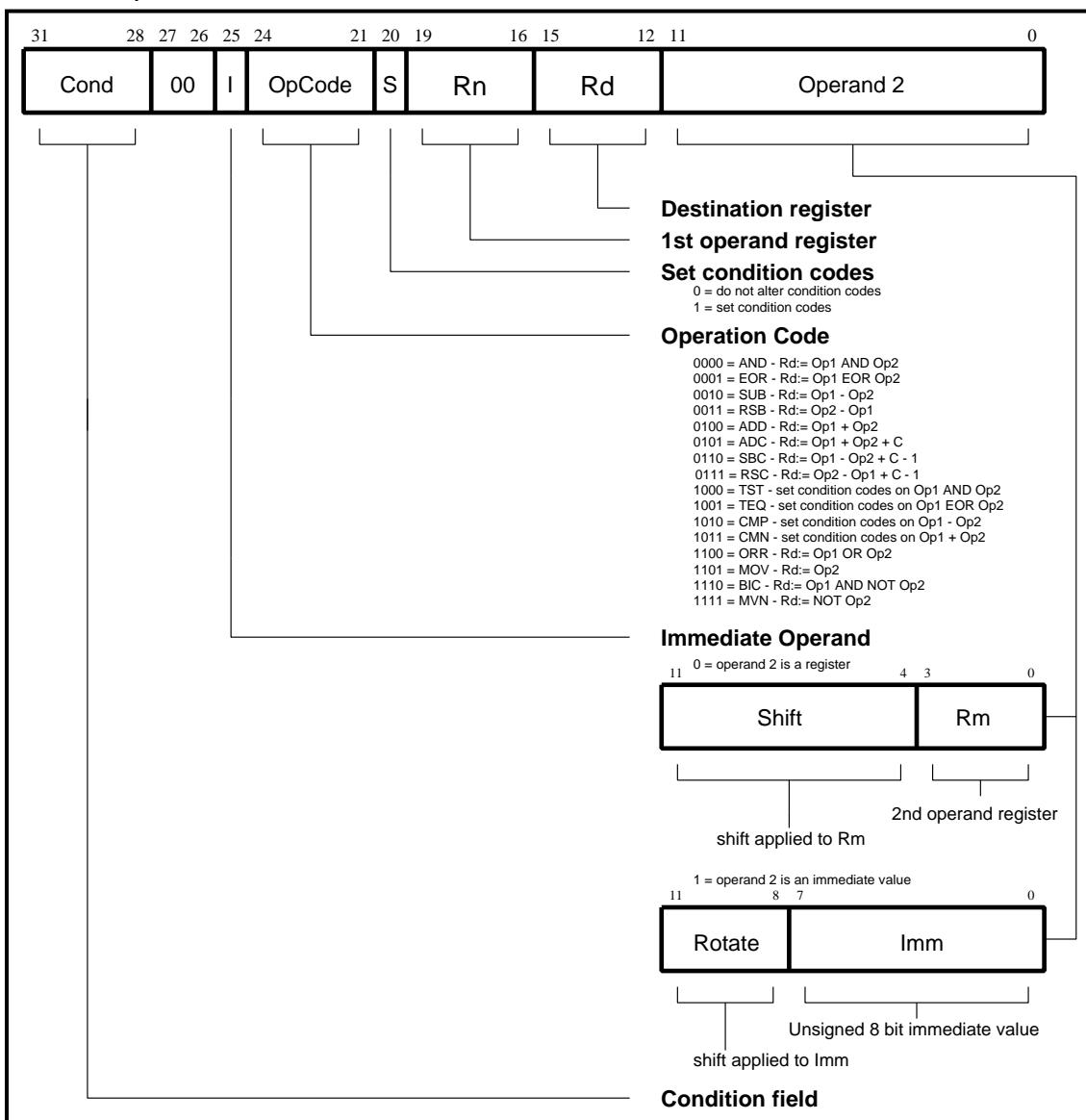


Figure 5-4: Data processing instructions

ARM Processor Instruction Set

5.4.1 CPSR flags

The data processing operations may be classified as logical or arithmetic. The logical operations (AND, EOR, TST, TEQ, ORR, MOV, BIC, MVN) perform the logical action on all corresponding bits of the operand or operands to produce the result.

If the S bit is set (and Rd is not R15):

- the V flag in the CPSR will be unaffected
- the C flag will be set to the carry out from the barrel shifter (or preserved when the shift operation is LSL #0)
- the Z flag will be set if and only if the result is all zeros
- the N flag will be set to the logical value of bit 31 of the result.

Assembler mnemonic	OpCode	Action
AND	0000	operand1 AND operand2
EOR	0001	operand1 EOR operand2
SUB	0010	operand1 - operand2
RSB	0011	operand2 - operand1
ADD	0100	operand1 + operand2
ADC	0101	operand1 + operand2 + carry
SBC	0110	operand1 - operand2 + carry - 1
RSC	0111	operand2 - operand1 + carry - 1
TST	1000	as AND, but result is not written
TEQ	1001	as EOR, but result is not written
CMP	1010	as SUB, but result is not written
CMN	1011	as ADD, but result is not written
ORR	1100	operand1 OR operand2
MOV	1101	operand2 (operand1 is ignored)
BIC	1110	operand1 AND NOT operand2 (Bit clear)
MVN	1111	NOT operand2 (operand1 is ignored)

Table 5-1: ARM data processing instructions

The arithmetic operations (SUB, RSB, ADD, ADC, SBC, RSC, CMP, CMN) treat each operand as a 32-bit integer (either unsigned or 2's complement signed, the two are equivalent).

ARM Processor Instruction Set

If the S bit is set (and Rd is not R15):

- the V flag in the CPSR will be set if an overflow occurs into bit 31 of the result; this may be ignored if the operands were considered unsigned, but warns of a possible error if the operands were 2's complement signed
- the C flag will be set to the carry out of bit 31 of the ALU
- the Z flag will be set if and only if the result was zero
- the N flag will be set to the value of bit 31 of the result (indicating a negative result if the operands are considered to be 2's complement signed).

5.4.2 Shifts

When the second operand is specified to be a shifted register, the operation of the barrel shifter is controlled by the Shift field in the instruction. This field indicates the type of shift to be performed (logical left or right, arithmetic right or rotate right). The amount by which the register should be shifted may be contained in an immediate field in the instruction, or in the bottom byte of another register (other than R15).

The encoding for the different shift types is shown in *Figure 5-5: ARM shift operations*.

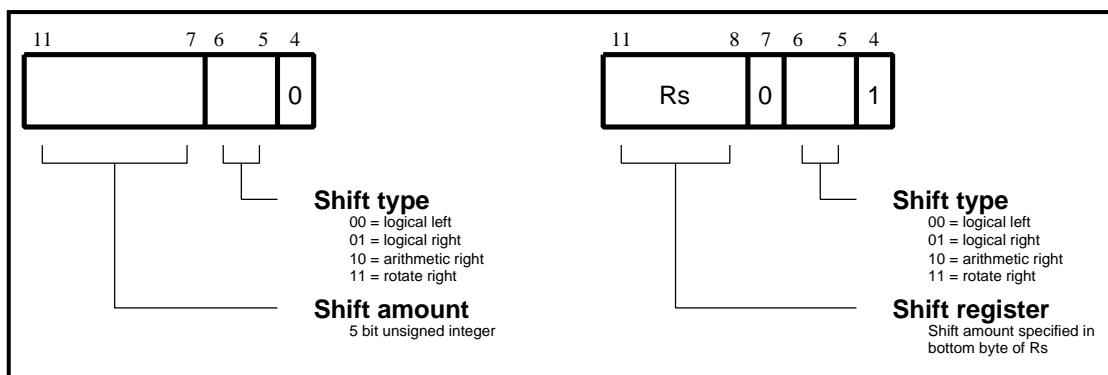


Figure 5-5: ARM shift operations

Instruction specified shift amount

When the shift amount is specified in the instruction, it is contained in a 5 bit field which may take any value from 0 to 31. A logical shift left (LSL) takes the contents of Rm and moves each bit by the specified amount to a more significant position. The least significant bits of the result are filled with zeros, and the high bits of Rm which do not map into the result are discarded, except that the least significant discarded bit becomes the shifter carry output which may be latched into the C bit of the CPSR when the ALU operation is in the logical class (see above). For example, the effect of LSL #5 is shown in *Figure 5-6: Logical shift left* on page 5-8.



ARM Processor Instruction Set

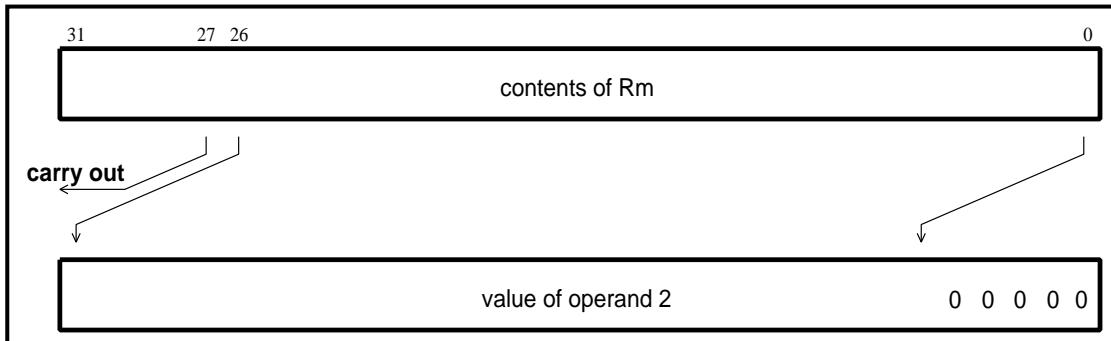


Figure 5-6: Logical shift left

Note: LSL #0 is a special case, where the shifter carry out is the old value of the CPSR C flag. The contents of Rm are used directly as the second operand.

Logical shift right

A logical shift right (LSR) is similar, but the contents of Rm are moved to less significant positions in the result. LSR #5 has the effect shown in Figure 5-7: Logical shift right.

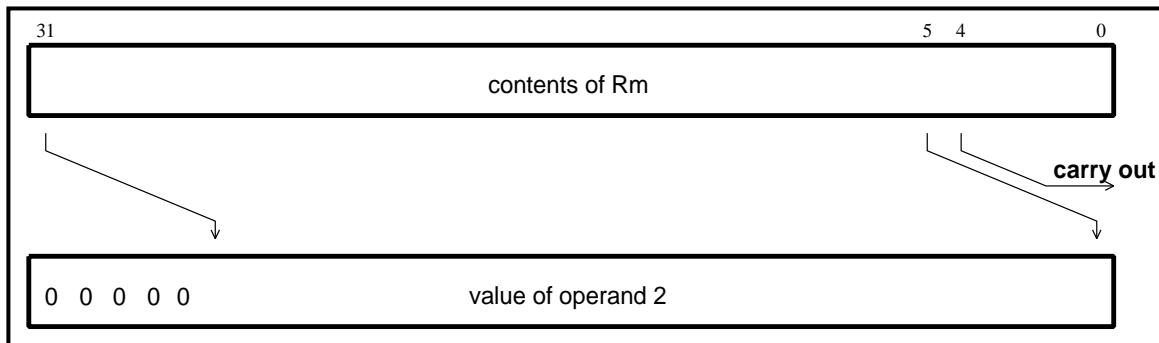


Figure 5-7: Logical shift right

The form of the shift field which might be expected to correspond to LSR #0 is used to encode LSR #32, which has a zero result with bit 31 of Rm as the carry output. Logical shift right zero is redundant as it is the same as logical shift left zero, so the assembler will convert LSR #0 (and ASR #0 and ROR #0) into LSL #0, and allow LSR #32 to be specified.

Arithmetic shift right

An arithmetic shift right (ASR) is similar to logical shift right, except that the high bits are filled with bit 31 of Rm instead of zeros. This preserves the sign in 2's complement notation. For example, ASR #5 is shown in Figure 5-8: Arithmetic shift right on page 5-9.

ARM Processor Instruction Set

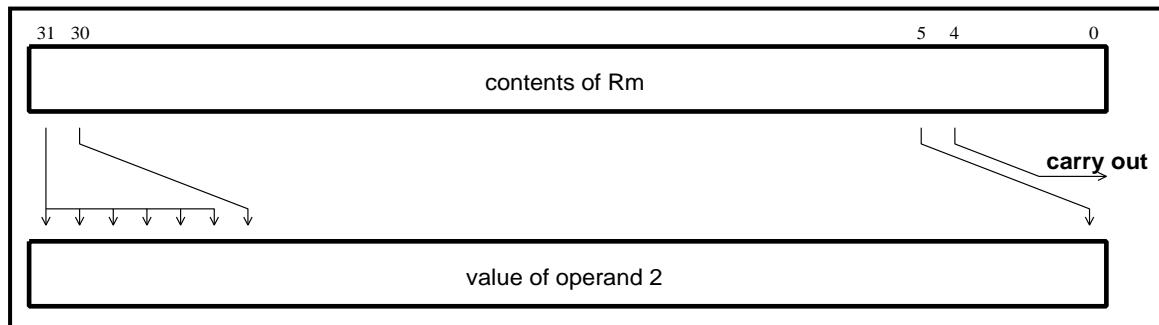


Figure 5-8: Arithmetic shift right

The form of the shift field which might be expected to give ASR #0 is used to encode ASR #32. Bit 31 of Rm is again used as the carry output, and each bit of operand 2 is also equal to bit 31 of Rm. The result is therefore all ones or all zeros, according to the value of bit 31 of Rm.

Rotate right

Rotate right (ROR) operations reuse the bits which 'overshoot' in a logical shift right operation by reintroducing them at the high end of the result, in place of the zeros used to fill the high end in logical right operations. For example, ROR #5 is shown in *Figure 5-9: Rotate right* on page 5-9.

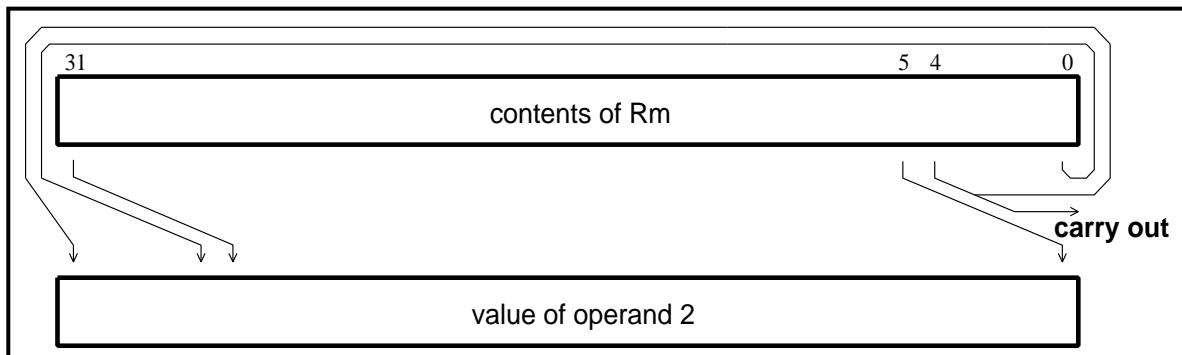


Figure 5-9: Rotate right

The form of the shift field which might be expected to give ROR #0 is used to encode a special function of the barrel shifter, rotate right extended (RRX). This is a rotate right by one bit position of the 33 bit quantity formed by appending the CPSR C flag to the most significant end of the contents of Rm as shown in *Figure 5-10: Rotate right extended* on page 5-10.



ARM Processor Instruction Set

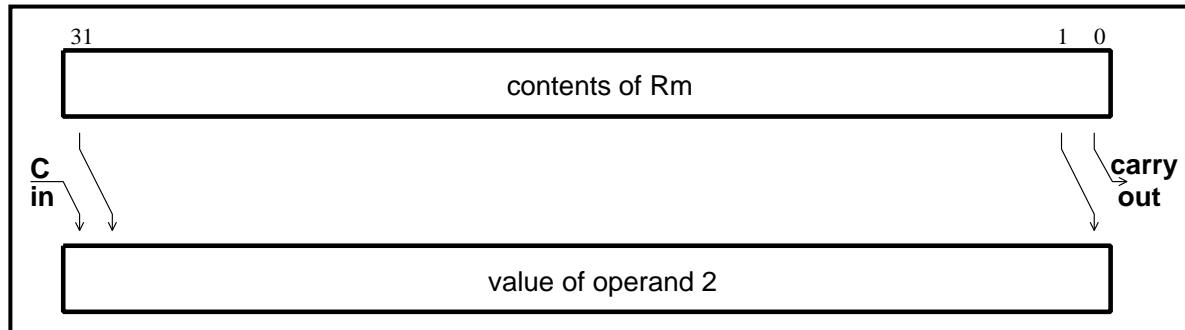


Figure 5-10: Rotate right extended

Register specified shift amount

Only the least significant byte of the contents of Rs is used to determine the shift amount. Rs can be any general register other than R15.

Byte	Description
0	Unchanged contents of Rm will be used as the second operand, and the old value of the CPSR C flag will be passed on as the shifter carry output
1 - 31	The shifted result will exactly match that of an instruction specified shift with the same value and shift operation
32 or more	The result will be a logical extension of the shift described above: <ol style="list-style-type: none">1 LSL by 32 has result zero, carry out equal to bit 0 of Rm.2 LSL by more than 32 has result zero, carry out zero.3 LSR by 32 has result zero, carry out equal to bit 31 of Rm.4 LSR by more than 32 has result zero, carry out zero.5 ASR by 32 or more has result filled with and carry out equal to bit 31 of Rm.6 ROR by 32 has result equal to Rm, carry out equal to bit 31 of Rm.7 ROR by n where n is greater than 32 will give the same result and carry out as ROR by $n-32$; therefore repeatedly subtract 32 from n until the amount is in the range 1 to 32 and see above.

Table 5-2: Register specified shift amount

Note: The zero in bit 7 of an instruction with a register controlled shift is compulsory; a one in this bit will cause the instruction to be a multiply or undefined instruction.

5.4.3 Immediate operand rotates

The immediate operand rotate field is a 4 bit unsigned integer which specifies a shift operation on the 8 bit immediate value. This value is zero extended to 32 bits, and then subject to a rotate right by twice the value in the rotate field. This enables many common constants to be generated, for example all powers of 2.

5.4.4 Writing to R15

When Rd is a register other than R15, the condition code flags in the CPSR may be updated from the ALU flags as described above.

When Rd is R15 and the S flag in the instruction is not set the result of the operation is placed in R15 and the CPSR is unaffected.

When Rd is R15 and the S flag is set the result of the operation is placed in R15 and the SPSR corresponding to the current mode is moved to the CPSR. This allows state changes which atomically restore both PC and CPSR.

Note: This form of instruction must not be used in User mode.

5.4.5 Using R15 as an operand

If R15 (the PC) is used as an operand in a data processing instruction the register is used directly.

The PC value will be the address of the instruction, plus 8 or 12 bytes due to instruction prefetching. If the shift amount is specified in the instruction, the PC will be 8 bytes ahead. If a register is used to specify the shift amount the PC will be 12 bytes ahead.

5.4.6 TEQ, TST, CMP & CMN opcodes

These instructions do not write the result of their operation but do set flags in the CPSR. An assembler shall always set the S flag for these instructions even if it is not specified in the mnemonic.

The TEQP form of the instruction used in earlier processors shall not be used in the 32-bit modes, the PSR transfer operations should be used instead. If used in these modes, its effect is to move SPSR_<mode> to CPSR if the processor is in a privileged mode and to do nothing if in User mode.

5.4.7 Instruction cycle times

Data Processing instructions vary in the number of incremental cycles taken as follows:

Instruction	Cycles
Normal Data Processing	1 instruction fetch
Data Processing with register specified shift	1 instruction fetch + 1 internal cycle
Data Processing with PC written	3 instruction fetches
Data Processing with register specified shift and PC written	3 instruction fetches and 1 internal cycle

Figure 5-11: Instruction cycle times

See 5.17 *Instruction Speed Summary* on page 5-47 for more information.



ARM Processor Instruction Set

5.4.8 Assembler syntax

- 1 MOV,MVN - single operand instructions
 $\langle\text{opcode}\rangle \{\text{cond}\} \{S\} \text{ Rd}, \langle\text{Op2}\rangle$
- 2 CMP,CMN,TEQ,TST - instructions which do not produce a result.
 $\langle\text{opcode}\rangle \{\text{cond}\} \text{ Rn}, \langle\text{Op2}\rangle$
- 3 AND,EOR,SUB,RSB,ADD,ADC,SBC,RSC,ORR,BIC
 $\langle\text{opcode}\rangle \{\text{cond}\} \{S\} \text{ Rd}, \text{Rn}, \langle\text{Op2}\rangle$

where:

$\langle\text{Op2}\rangle$	is $\text{Rm}\{ , \langle\text{shift}\rangle \}$ or $\langle\#\text{expression}\rangle$
$\{\text{cond}\}$	two-character condition mnemonic, see <i>Figure 5-2: Condition codes</i> on page 5-2
$\{S\}$	set condition codes if S present (implied for CMP, CMN, TEQ, TST).
$\text{Rd}, \text{ Rn}$ and Rm	are expressions evaluating to a register number.
$\langle\#\text{expression}\rangle$	if used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.
$\langle\text{shift}\rangle$	is $\langle\text{shiftname}\rangle \langle\text{register}\rangle$ or $\langle\text{shiftname}\rangle \# \text{expression}$, or RRX (rotate right one bit with extend).
$\langle\text{shiftname}\rangle$	is: ASL, LSL, LSR, ASR, ROR. (ASL is a synonym for LSL; they assemble to the same code.)

5.4.9 Example

```
ADDEQ R2,R4,R5          ;if the Z flag is set make R2:=R4+R
TEQS  R4,#3              ;test R4 for equality with 3
                           ;(the S is in fact redundant as the
                           ;assembler inserts it automatically)
SUB   R4,R5,R7,LSR R2;  ;logical right shift R7 by the number in
                           ;the bottom byte of R2, subtract result
                           ;from R5, and put the answer into R4
MOV    PC,R14             ;return from subroutine
MOVS   PC,R14             ;return from exception and restore CPSR
                           ;from SPSR_mode
```

5.5 PSR Transfer (MRS, MSR)

The instruction is only executed if the condition is true. The various conditions are defined in *5.2 The Condition Field* on page 5-2.

The MRS and MSR instructions are formed from a subset of the Data Processing operations and are implemented using the TEQ, TST, CMN and CMP instructions without the S flag set. The encoding is shown in *Figure 5-12: PSR transfer* on page 5-14.

These instructions allow access to the CPSR and SPSR registers. The MRS instruction allows the contents of the CPSR or SPSR_<mode> to be moved to a general register.

The MSR instruction allows the contents of a general register to be moved to the CPSR or SPSR_<mode> register. The MSR instruction also allows an immediate value or register contents to be transferred to the condition code flags (N,Z,C and V) of CPSR or SPSR_<mode> without affecting the control bits. In this case, the top four bits of the specified register contents or 32-bit immediate value are written to the top four bits of the relevant PSR.

5.5.1 Operand restrictions

In User mode, the control bits of the CPSR are protected from change, so only the condition code flags of the CPSR can be changed. In other (privileged) modes the entire CPSR can be changed.

The SPSR register which is accessed depends on the mode at the time of execution. For example, only SPSR_fiq is accessible when the processor is in FIQ mode.

Note: *R15 must not be specified as the source or destination register.*

A further restriction is that you must not attempt to access an SPSR in User mode, since no such register exists.

5.5.2 Reserved bits

Only eleven bits of the PSR are defined in the ARM processor (N,Z,C,V,I,F & M[4:0]); the remaining bits (= PSR[27:8,5]) are reserved for use in future versions of the processor.

Compatibility

To ensure the maximum compatibility between ARM processor programs and future processors, the following rules should be observed:

- 1 The reserved bit must be preserved when changing the value in a PSR.
- 2 Programs must not rely on specific values from the reserved bits when checking the PSR status, since they may read as one or zero in future processors.

A read-modify-write strategy should therefore be used when altering the control bits of any PSR register; this involves transferring the appropriate PSR register to a general register using the MRS instruction, changing only the relevant bits and then transferring the modified value back to the PSR register using the MSR instruction.



ARM Processor Instruction Set

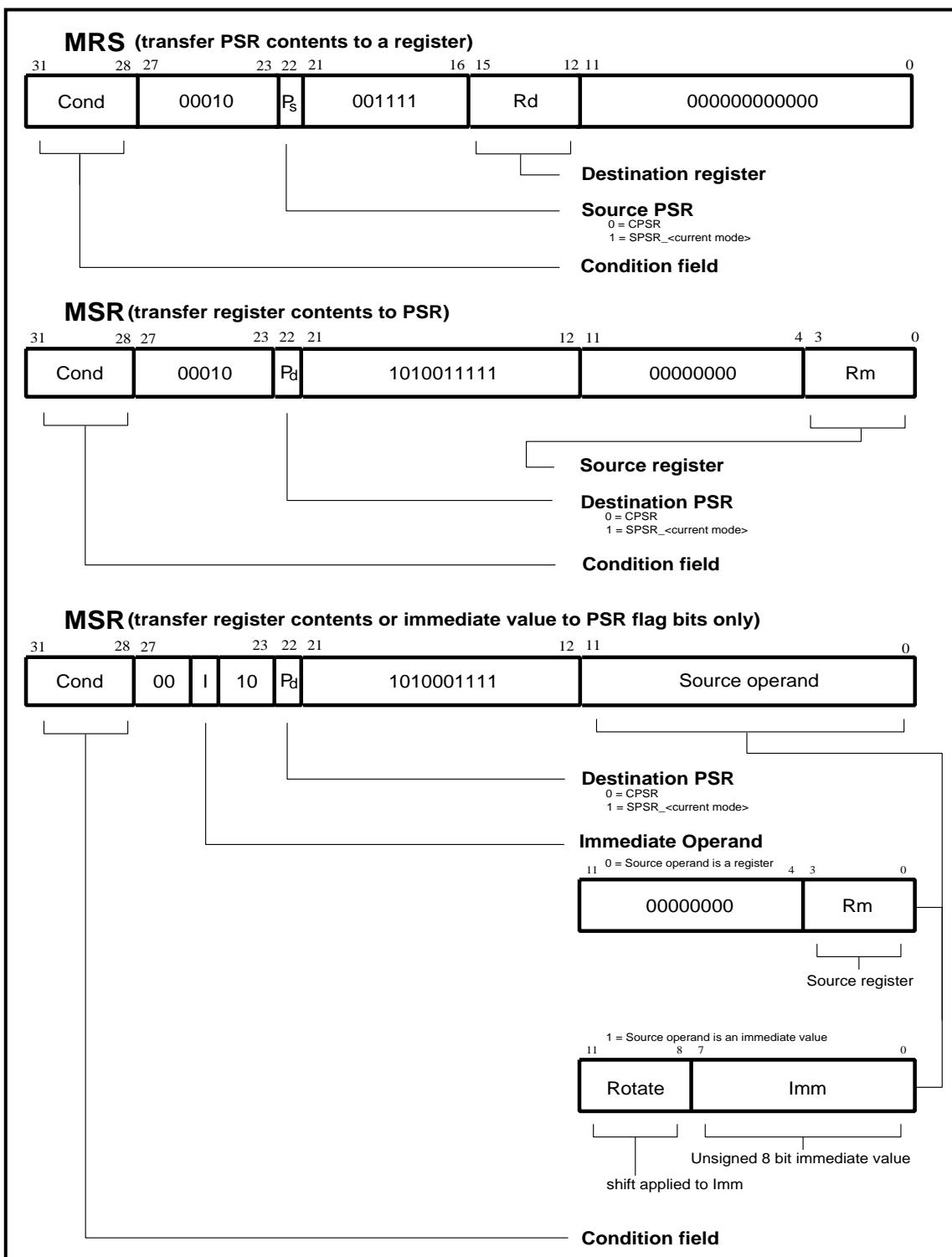


Figure 5-12: PSR transfer

For example, the following sequence performs a mode change:

```
MRS    R0,CPSR          ;take a copy of the CPSR  
BIC    R0,R0,#0x1F      ;clear the mode bits  
ORR    R0,R0,#new_mode  ;select new mode  
MSR    CPSR,R0          ;write back the modified CPSR
```

When the aim is simply to change the condition code flags in a PSR, a value can be written directly to the flag bits without disturbing the control bits. e.g. The following instruction sets the N,Z,C & V flags:

```
MSR    CPSR_flg,#0xF0000000  
                   ;set all the flags regardless of  
                   ;their previous state (does not  
                   ;affect any control bits)
```

Note: *Do not attempt to write an 8 bit immediate value into the whole PSR since such an operation cannot preserve the reserved bits.*

5.5.3 Instruction cycle times

PSR Transfers take 1 instruction fetch. For more information see [5.17 Instruction Speed Summary](#) on page 5-47.

5.5.4 Assembler syntax

1 MRS - transfer PSR contents to a register
 MRS{cond} Rd,<psr>

2 MSR - transfer register contents to PSR
 MSR{cond} <psr>,Rm

3 MSR - transfer register contents to PSR flag bits only
 MSR{cond} <psrf>,Rm

The most significant four bits of the register contents are written to the N,Z,C & V flags respectively.

4 MSR - transfer immediate value to PSR flag bits only
 MSR{cond} <psrf>,<#expression>

The expression should symbolize a 32-bit value of which the most significant four bits are written to the N,Z,C & V flags respectively.

where:

{cond} two-character condition mnemonic, see [Figure 5-2: Condition codes](#) on page 5-2

Rd and Rm expressions evaluating to a register number other than R15

<psr> is CPSR, CPSR_all, SPSR or SPSR_all. (CPSR and CPSR_all are synonyms as are SPSR and SPSR_all)

<psrf> is CPSR_flg or SPSR_flg

<#expression> where used, the assembler will attempt to generate a shifted immediate 8-bit field to match the expression. If this is impossible, it will give an error.



ARM Processor Instruction Set

5.5.5 Examples

In User mode the instructions behave as follows:

```
MSR    CPSR_all,Rm      ;CPSR[31:28] <- Rm[31:28]
MSR    CPSR_flg,Rm      ;CPSR[31:28] <- Rm[31:28]
MSR    CPSR_flg,#0xA0000000;
                  ;CPSR[31:28] <- 0xA
                  ;(i.e. set N,C; clear Z,V)
MRS    Rd,CPSR          ;Rd[31:0] <- CPSR[31:0]
```

In privileged modes the instructions behave as follows:

```
MSR    CPSR_all,Rm      ;CPSR[31:0] <- Rm[31:0]
MSR    CPSR_flg,Rm      ;CPSR[31:28] <- Rm[31:28]
MSR    CPSR_flg,#0x50000000;
                  ;CPSR[31:28] <- 0x5
                  ;(i.e. set Z,V; clear N,C)
MRS    Rd,CPSR          ;Rd[31:0] <- CPSR[31:0]
MSR    SPSR_all,Rm      ;SPSR_<mode>[31:0] <- Rm[31:0]
MSR    SPSR_flg,Rm      ;SPSR_<mode>[31:28] <- Rm[31:28]
MSR    SPSR_flg,#0xC0000000;
                  ;SPSR_<mode>[31:28] <- 0xC
                  ;(i.e. set N,Z; clear C,V)
MRS    Rd,SPSR          ;Rd[31:0] <- SPSR_<mode>[31:0]
```

5.6 Multiply and Multiply-Accumulate (MUL, MLA)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 5-13: Multiply instructions*.

The multiply and multiply-accumulate instructions use a 2-bit Booth's algorithm to perform integer multiplication. They give the least significant 32-bits of the product of two 32-bit operands, and may be used to synthesize higher-precision multiplications.

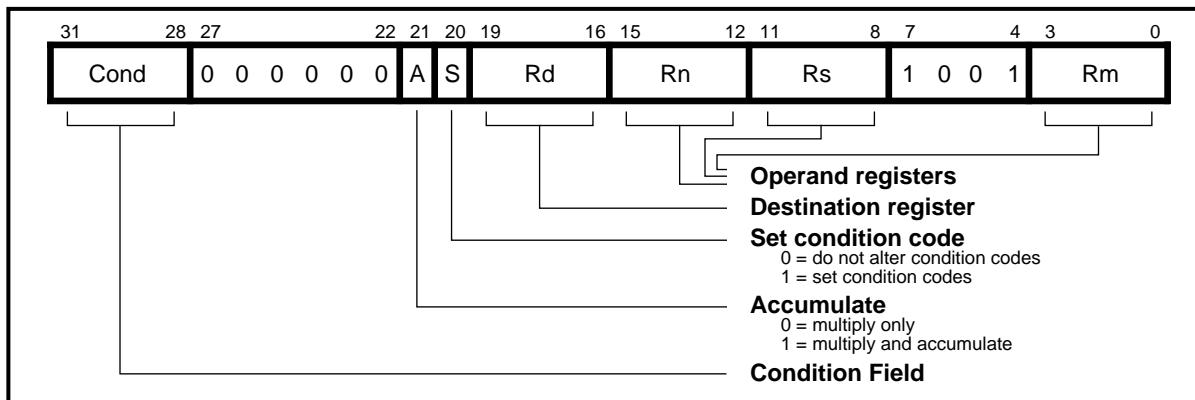


Figure 5-13: Multiply instructions

The multiply form of the instruction gives $Rd := Rm * Rs$. Rn is ignored, and should be set to zero for compatibility with possible future upgrades to the instruction set.

The multiply-accumulate form gives $Rd := Rm * Rs + Rn$, which can save an explicit ADD instruction in some circumstances.

The results of a signed multiply and of an unsigned multiply of 32-bit operands differ only in the upper 32 bits; the low 32 bits of the signed and unsigned results are identical. As these instructions only produce the low 32 bits of a multiply, they can be used for both signed and unsigned multiplies.

Example

For example consider the multiplication of the operands:

Operand A	Operand B	Result
0xFFFFFFF6	0x00000014	0xFFFFFFF38

If the operands are interpreted as signed, operand A has the value -10, operand B has the value 20, and the result is -200 which is correctly represented as 0xFFFFFFF38

If the operands are interpreted as unsigned, operand A has the value 4294967286, operand B has the value 20 and the result is 85899345720, which is represented as 0x13FFFFFF38, so the least significant 32 bits are 0xFFFFFFF38.

5.6.1 Operand restrictions

Due to the way multiplication was implemented, certain combinations of operand registers should be avoided. (The assembler will issue a warning if these restrictions are overlooked.)

The destination register (Rd) should not be the same as the operand register (Rm), as Rd is used to hold intermediate values and Rm is used repeatedly during multiply. A MUL will give a zero result if Rm=Rd, and an MLA will give a meaningless result. R15 must not be used as an operand or as the destination register.

All other register combinations will give correct results, and Rd, Rn and Rs may use the same register when required.

5.6.2 CPSR flags

Setting the CPSR flags is optional, and is controlled by the S bit in the instruction. The N (Negative) and Z (Zero) flags are set correctly on the result (N is made equal to bit 31 of the result, and Z is set if and only if the result is zero). The C (Carry) flag is set to a meaningless value and the V (Overflow) flag is unaffected.

5.6.3 Instruction cycle times

The Multiply instructions take 1 instruction fetch and m internal cycles, as shown in *Table 5-3: Instruction cycle times*. For more information see *5.17 Instruction Speed Summary* on page 5-47.

Multiplication by	Takes
any number between $2^{(2m-3)}$ and $2^{(2m-1)}-1$	1S+ml cycles for $1 < m > 16$.
Multiplication by 0 or 1	1S+1I cycles

Table 5-3: Instruction cycle times



ARM Processor Instruction Set

Multiplication by	Takes
any number greater than or equal to 2^{29}	1S+16I cycles.

Table 5-3: Instruction cycle times

m is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs

The maximum time for any multiply is thus 1S+16I cycles.

5.6.4 Assembler syntax

`MUL{cond}{S} Rd,Rm,Rs`

`MLA{cond}{S} Rd,Rm,Rs,Rn`

where:

{cond} two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2

{S} set condition codes if S present

Rd, Rm, Rs, Rn are expressions evaluating to a register number other than R15.

5.6.5 Examples

<code>MUL</code>	<code>R1,R2,R3</code>	<code>;R1:=R2*R3</code>
<code>MLAEQS</code>	<code>R1,R2,R3,R4</code>	<code>;conditionally</code> <code>;R1:=R2*R3+R4,</code> <code>;setting condition codes</code>

5.7 Single Data Transfer (LDR, STR)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 5-14: Single data transfer instructions*.

The single data transfer instructions are used to load or store single bytes or words of data. The memory address used in the transfer is calculated by adding an offset to or subtracting an offset from a base register.

The result of this calculation may be written back into the base register if “auto-indexing” is required.

ARM Processor Instruction Set

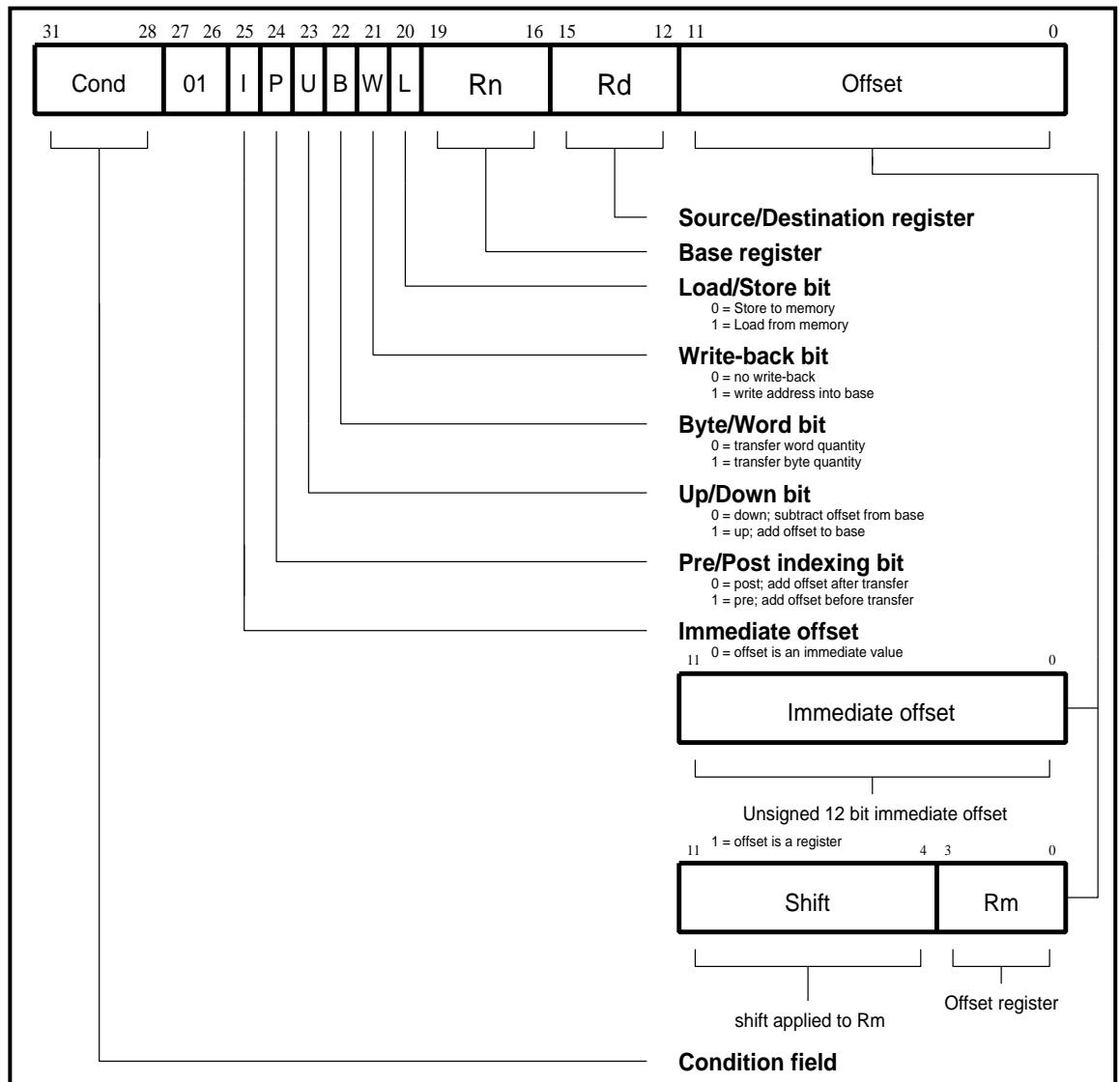


Figure 5-14: Single data transfer instructions

5.7.1 Offsets and auto-indexing

The offset from the base may be either a 12-bit unsigned binary immediate value in the instruction, or a second register (possibly shifted in some way). The offset may be added to ($U=1$) or subtracted from ($U=0$) the base register R_n . The offset modification may be performed either before (pre-indexed, $P=1$) or after (post-indexed, $P=0$) the base is used as the transfer address.

The W bit gives optional auto increment and decrement addressing modes. The modified base value may be written back into the base ($W=1$), or the old base value may be kept ($W=0$).



ARM Processor Instruction Set

Post-indexed addressing

In the case of post-indexed addressing, the write back bit is redundant and is always set to zero, since the old base value can be retained by setting the offset to zero. Therefore post-indexed data transfers always write back the modified base. The only use of the W bit in a post-indexed data transfer is in privileged mode code, where setting the W bit forces non-privileged mode for the transfer, allowing the operating system to generate a user address in a system where the memory management hardware makes suitable use of this hardware.

5.7.2 Shifted register offset

The 8 shift control bits are described in the data processing instructions section. However, the register specified shift amounts are not available in this instruction class. See 5.4.2 *Shifts* on page 5-7.

5.7.3 Bytes and words

This instruction class may be used to transfer a byte (B=1) or a word (B=0) between an ARM processor register and memory. The following text assumes that the ARM7500FE is operating with 32-bit wide memory. If it is operating with 16-bit wide memory, the positions of bytes on the *external* data bus will be different, although, on the ARM7500FE internal data bus the positions will be as described here.

The action of LDR(B) and STR(B) instructions is influenced by the 3 instruction fetches. For more information see 5.17 *Instruction Speed Summary* on page 5-47. The two possible configurations are described below.

ARM Processor Instruction Set

Little endian configuration

Byte load (LDRB) expects the data on data bus inputs 7 through 0 if the supplied address is on a word boundary, on data bus inputs 15 through 8 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register, and the remaining bits of the register are filled with zeros. See *Figure 4-1: Little-endian addresses of bytes within words* on page 4-2.

Byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0.

Word load (LDR) will normally use a word aligned address. However, an address offset from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 0 to 7. This means that half-words accessed at offsets 0 and 2 from the word boundary will be correctly loaded into bits 0 through 15 of the register. Two shift operations are then required to clear or to sign extend the upper 16 bits. This is illustrated in *Figure 5-15: Little Endian offset addressing* on page 5-21.

A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

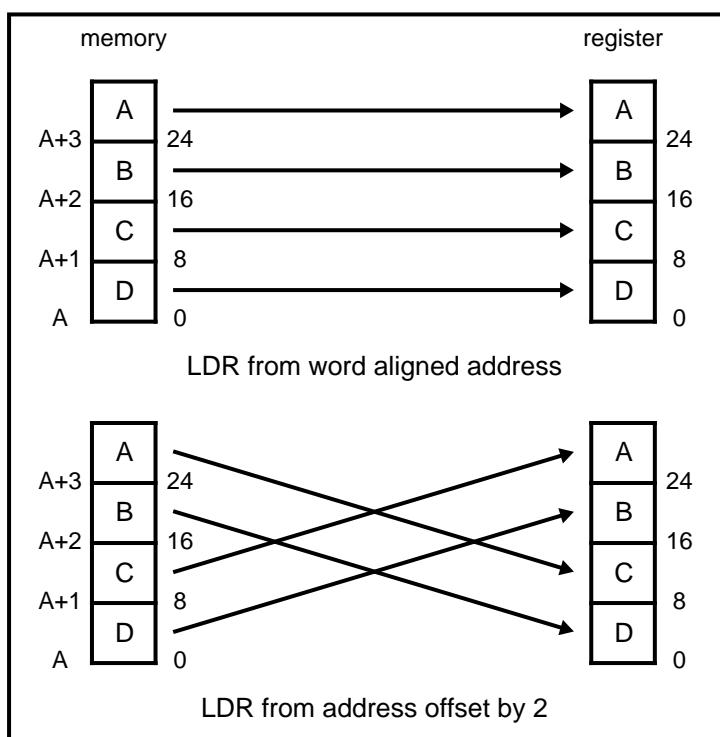


Figure 5-15: Little Endian offset addressing



ARM Processor Instruction Set

Big endian configuration

- Byte load (LDRB) expects the data on data bus inputs 31 through 24 if the supplied address is on a word boundary, on data bus inputs 23 through 16 if it is a word address plus one byte, and so on. The selected byte is placed in the bottom 8 bits of the destination register and the remaining bits of the register are filled with zeros. Please see *Figure 4-2: Big-endian addresses of bytes within words* on page 4-3.
- Byte store (STRB) repeats the bottom 8 bits of the source register four times across data bus outputs 31 through 0.
- Word load (LDR) should generate a word aligned address. An address offset of 0 or 2 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 31 through 24. This means that half-words accessed at these offsets will be correctly loaded into bits 16 through 31 of the register. A shift operation is then required to move (and optionally sign extend) the data into the bottom 16 bits. An address offset of 1 or 3 from a word boundary will cause the data to be rotated into the register so that the addressed byte occupies bits 15 through 8.
- A word store (STR) should generate a word aligned address. The word presented to the data bus is not affected if the address is not word aligned. That is, bit 31 of the register being stored always appears on data bus output 31.

5.7.4 Use of R15

Do not specify write-back if R15 is specified as the base register (Rn). When using R15 as the base register you must remember it contains an address 8 bytes on from the address of the current instruction.

R15 must not be specified as the register offset (Rm).

When R15 is the source register (Rd) of a register store (STR) instruction, the stored value will be address of the instruction plus 12.

5.7.5 Restriction on the use of base register

When configured for late aborts, the following example code is difficult to unwind as the base register, Rn, gets updated before the abort handler starts. Sometimes it may be impossible to calculate the initial value.

For example:

```
LDR    R0, [R1], R1  
<LDR | STR> Rd, [Rn], {+/-}Rn{, <shift>}
```

Therefore a post-indexed LDR|STR where Rm is the same register as Rn shall not be used.

5.7.6 Data aborts

A transfer to or from a legal address may cause the MMU to generate an abort. It is up to the system software to resolve the cause of the problem, then the instruction can be restarted and the original program continued.

5.7.7 Instruction cycle times

Instruction	Cycles
Normal LDR instruction	1 instruction fetch, 1 data read and 1 internal cycle
LDR PC	3 instruction fetches, 1 data read and 1 internal cycle.
STR instruction	1 instruction fetch and 1 data write incremental cycles.

Table 5-4: Instruction cycle times

For more information see 5.17 *Instruction Speed Summary* on page 5-47.

5.7.8 Assembler syntax

<LDR | STR> {cond} {B} {T} Rd, <Address>

LDR load from memory into a register

STR store from a register into memory

{cond} two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2

{B} if B is present then byte transfer, otherwise word transfer

{T} if T is present the W bit will be set in a post-indexed instruction, forcing non-privileged mode for the transfer cycle. T is not allowed when a pre-indexed addressing mode is specified or implied.

Rd is an expression evaluating to a valid register number.

<Address> can be:

1 An expression which generates an address:

<expression>

The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

2 A pre-indexed addressing specification:

[Rn] offset of zero

[Rn, <#expression>] {!} offset of <expression> bytes

[Rn, {+/-}Rm{, <shift>}] {!} offset of +/- contents of index register, shifted by <shift>

3 A post-indexed addressing specification:

[Rn], <#expression> offset of <expression> bytes

[Rn], {+/-}Rm{, <shift>} offset of +/- contents of index register, shifted as by <shift>.

Rn and Rm are expressions evaluating to a register number. If Rn is R15 then the assembler will subtract 8 from the offset value to allow for ARM7500FE pipelining. In this case base write-back



ARM Processor Instruction Set

	shall not be specified.
<shift>	is a general shift operation (see section on data processing instructions) but note that the shift amount may not be specified by a register.
{ ! }	writes back the base register (set the W bit) if ! is present.

5.7.9 Examples

```
STR    R1,[R2,R4]!      ;store R1 at R2+R4 (both of which are  
                      ;registers) and write back address to R2  
STR    R1,[R2],R4       ;store R1 at R2 and write back  
                      ;R2+R4 to R2  
LDR    R1,[R2,#16]     ;load R1 from contents of R2+16  
                      ; Don't write back  
LDR    R1,[R2,R3,LSL#2]  
                      ;load R1 from contents of R2+R3*4  
LDREQB  
      R1,[R6,#5]       ;conditionally load byte at R6+5 into  
                      ; R1 bits 0 to 7, filling bits 8 to 31  
                      ; with zeros  
STR    R1,PLACE        ;generate PC relative offset to address  
                      ;PLACE  
      .  
      .  
PLACE
```

5.8 Block Data Transfer (LDM, STM)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 5-16: Block data transfer instructions*.

Block data transfer instructions are used to load (LDM) or store (STM) any subset of the currently visible registers. They support all possible stacking modes, maintaining full or empty stacks which can grow up or down memory, and are very efficient instructions for saving or restoring context, or for moving large blocks of data around main memory.

5.8.1 The register list

The instruction can cause the transfer of any registers in the current bank (and non-user mode programs can also transfer to and from the user bank, see below). The register list is a 16 bit field in the instruction, with each bit corresponding to a register. A 1 in bit 0 of the register field will cause R0 to be transferred, a 0 will cause it not to be transferred; similarly bit 1 controls the transfer of R1, and so on.

Any subset of the registers, or all the registers, may be specified. The only restriction is that the register list should not be empty.

Whenever R15 is stored to memory the stored value is the address of the STM instruction plus 12.

ARM Processor Instruction Set

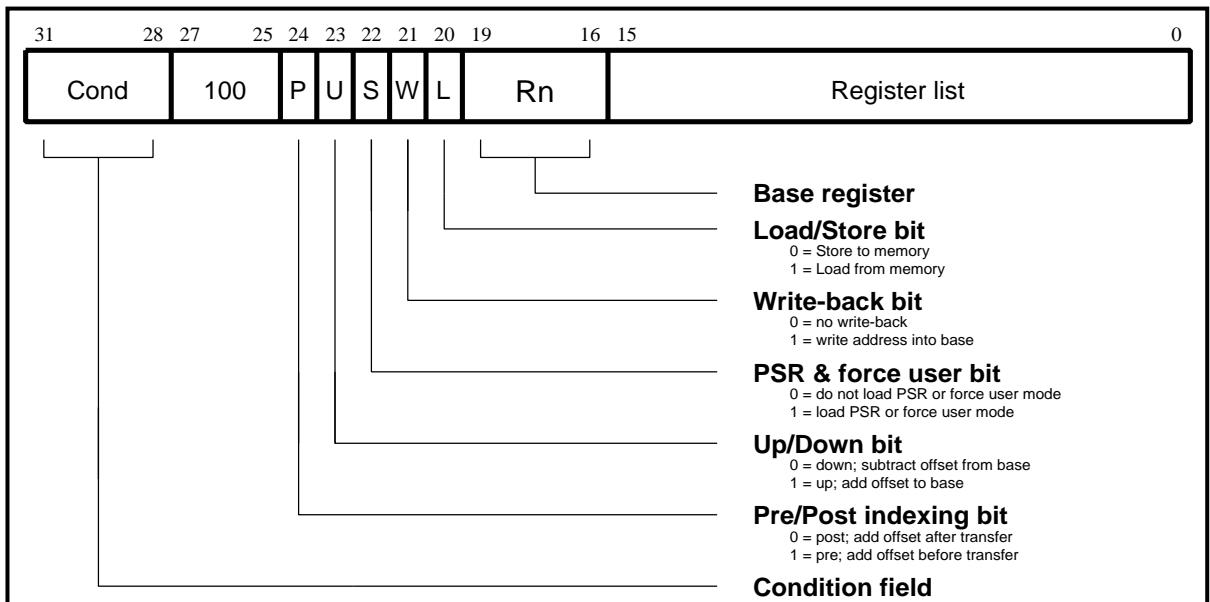


Figure 5-16: Block data transfer instructions



ARM Processor Instruction Set

5.8.2 Addressing modes

The transfer addresses are determined by:

- the contents of the base register (Rn)
- the pre/post bit (P)
- the up/down bit (U)

The registers are transferred in the order lowest to highest, so R15 (if in the list) will always be transferred last. The lowest register also gets transferred to/from the lowest memory address.

By way of illustration, consider the transfer of R1, R5 and R7 in the case where Rn=0x1000 and write back of the modified base is required (W=1).

Figure 5-17: Post-increment addressing, Figure 5-18: Pre-increment addressing, Figure 5-19: Post-decrement addressing, and Figure 5-20: Pre-decrement addressing on page 5-28, show the sequence of register transfers, the addresses used, and the value of Rn after the instruction has completed.

In all cases, had write back of the modified base not been required (W=0), Rn would have retained its initial value of 0x1000 unless it was also in the transfer list of a load multiple register instruction, when it would have been overwritten with the loaded value.

5.8.3 Address alignment

The address should always be a word aligned quantity.

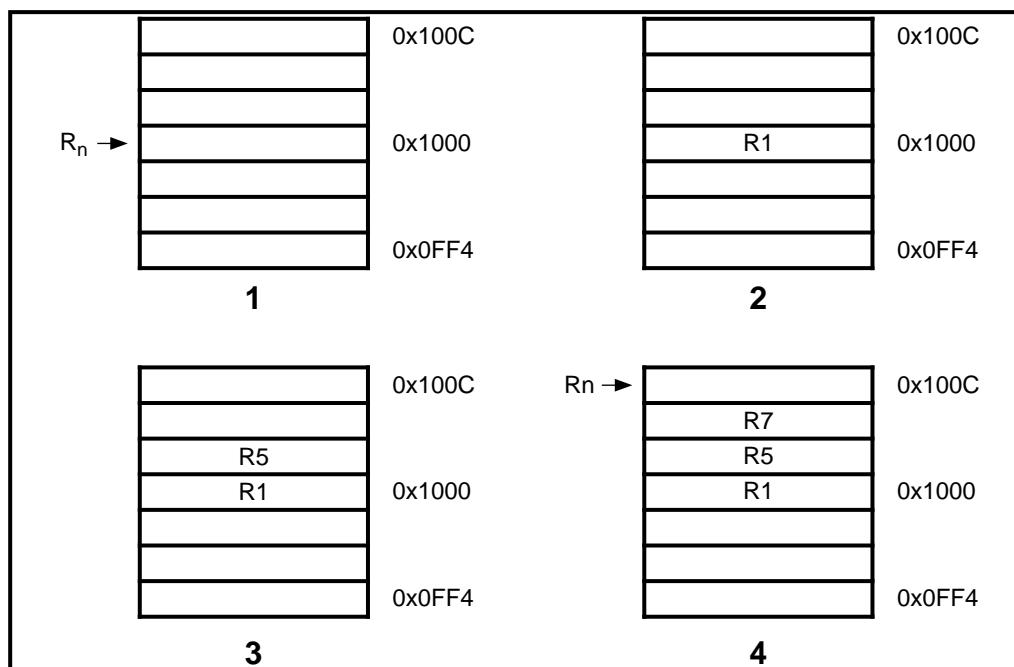


Figure 5-17: Post-increment addressing

ARM Processor Instruction Set

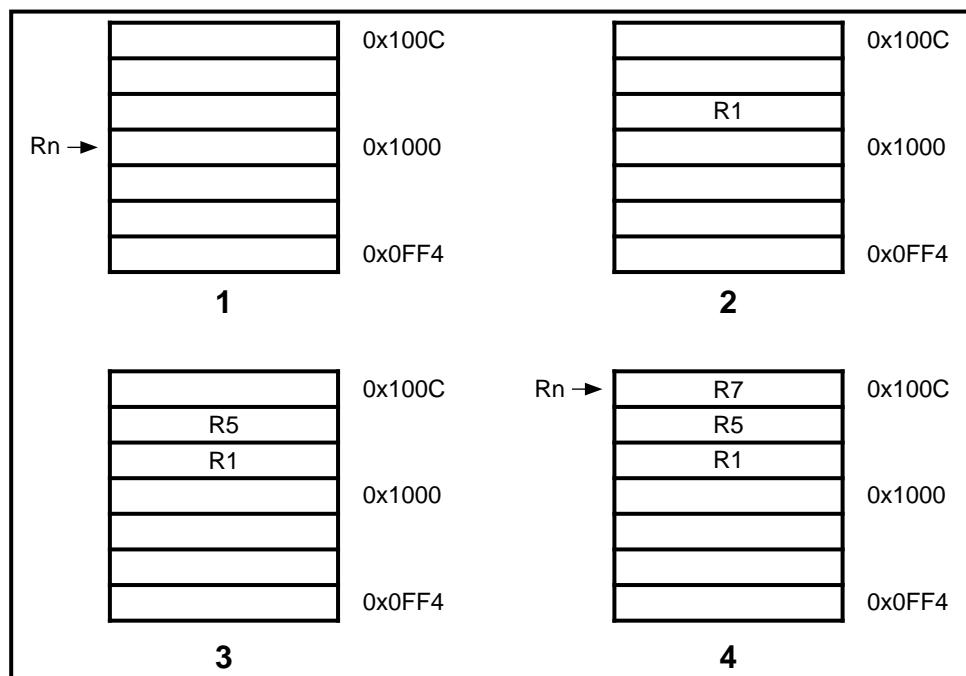


Figure 5-18: Pre-increment addressing

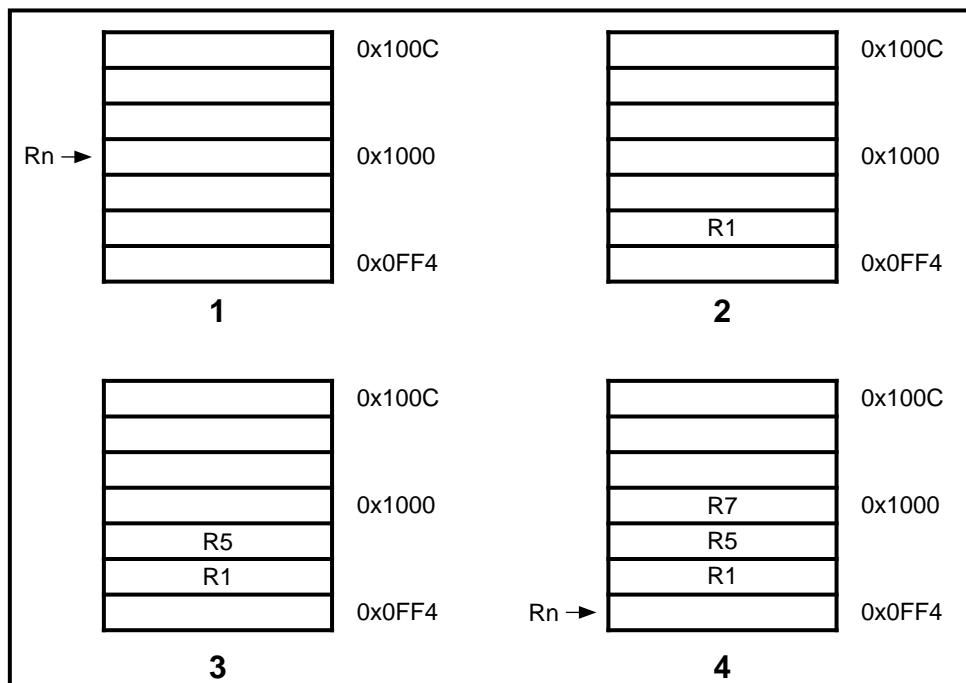


Figure 5-19: Post-decrement addressing



ARM Processor Instruction Set

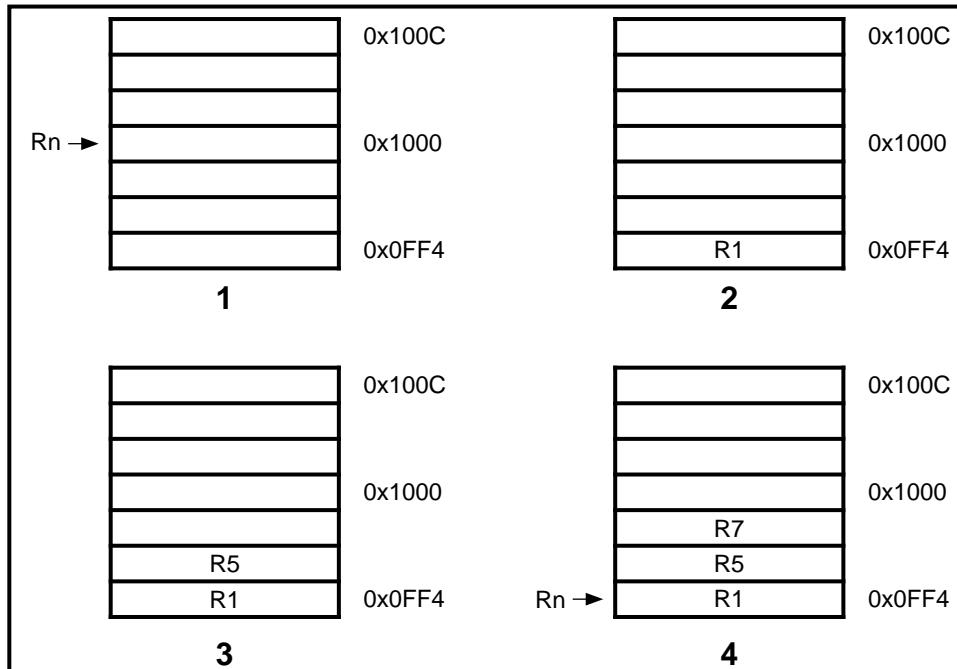


Figure 5-20: Pre-decrement addressing

5.8.4 Use of the S bit

When the S bit is set in a LDM/STM instruction its meaning depends on whether or not R15 is in the transfer list and on the type of instruction. The S bit should only be set if the instruction is to execute in a privileged mode.

LDM with R15 in transfer list and S bit set (Mode changes)

If the instruction is a LDM then SPSR_<mode> is transferred to CPSR at the same time as R15 is loaded.

STM with R15 in transfer list and S bit set (User bank transfer)

The registers transferred are taken from the User bank rather than the bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back shall not be used when this mechanism is employed.

R15 not in list and S bit set (User bank transfer)

For both LDM and STM instructions, the User bank registers are transferred rather than the register bank corresponding to the current mode. This is useful for saving the user state on process switches. Base write-back shall not be used when this mechanism is employed.

When the instruction is LDM, care must be taken not to read from a banked register during the following cycle (inserting a NOP after the LDM will ensure safety).

5.8.5 Use of R15 as the base register

R15 must not be used as the base register in any LDM or STM instruction.

5.8.6 Inclusion of the base in the register list

When write-back is specified, the base is written back at the end of the second cycle of the instruction. During an STM, the first register is written out at the start of the second cycle. An STM which includes storing the base, with the base as the first register to be stored, will therefore store the unchanged value, whereas with the base second or later in the transfer order, will store the modified value. An LDM will always overwrite the updated base if the base is in the list.

5.8.7 Data aborts

Some legal addresses may be unacceptable to the MMU. The MMU will then cause an abort. This can happen on any transfer during a multiple register load or store, and must be recoverable if ARM7500FE is to be used in a virtual memory system.

Aborts during STM instructions

If the abort occurs during a store multiple instruction, the ARM processor takes little action until the instruction completes, whereupon it enters the data abort trap. The memory manager is responsible for preventing erroneous writes to the memory. The only change to the internal state of the processor will be the modification of the base register if write-back was specified, and this must be reversed by software (and the cause of the abort resolved) before the instruction may be retried.

Aborts during LDM instructions

When the ARM processor detects a data abort during a load multiple instruction, it modifies the operation of the instruction to ensure that recovery is possible.

- 1 Overwriting of registers stops when the abort happens. The aborting load will not take place but earlier ones may have overwritten registers. The PC is always the last register to be written and so will always be preserved.
- 2 The base register is restored, to its modified value if write-back was requested. This ensures recoverability in the case where the base register is also in the transfer list, and may have been overwritten before the abort occurred.

The data abort trap is taken when the load multiple has completed, and the system software must undo any base modification (and resolve the cause of the abort) before restarting the instruction.

5.8.8 Instruction cycle times

Instruction	Cycles
Normal LDM instructions	1 instruction fetch, n data reads and 1 internal cycle
LDM PC	3 instruction fetches, n data reads and 1 internal cycle.
STM instructions	instruction fetch, n data reads and 1 internal cycle, where n is the number of words transferred.

Table 5-5: Instruction cycle times

For more information see 5.17 *Instruction Speed Summary* on page 5-47.



ARM Processor Instruction Set

5.8.9 Assembler syntax

<LDM | STM> {cond} <FD | ED | FA | EA | IA | IB | DA | DB> Rn{!}, <Rlist>{^}

where:

- {cond} is a two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
- Rn is an expression evaluating to a valid register number
- <Rlist> is a list of registers and register ranges enclosed in {} (e.g. {R0,R2-R7,R10}).
- {!} (if present) requests write-back (W=1), otherwise W=0
- {^} (if present) set S bit to load the CPSR along with the PC, or force transfer of user bank when in privileged mode

5.8.10 Addressing mode names

There are different assembler mnemonics for each of the addressing modes, depending on whether the instruction is being used to support stacks or for other purposes. The equivalencies between the names and the values of the bits in the instruction are shown in *Table 5-6: Addressing mode names*:

Key to table

FD, ED, FA, EA define pre/post indexing and the up/down bit by reference to the form of stack required.

- F Full stack (a pre-index has to be done before storing to the stack)
- E Empty stack
- A The stack is ascending (an STM will go up and LDM down)
- D The stack is descending (an STM will go down and LDM up)

The following symbols allow control when LDM/STM are not being used for stacks:

- IA Increment After
- IB Increment Before
- DA Decrement After
- DB Decrement Before

Name	Stack	Other	L-bit	P-bit	U-bit
pre-increment load	LDMED	LDMIB	1	1	1
post-increment load	LDMFD	LDMIA	1	0	1
pre-decrement load	LDMEA	LDMDB	1	1	0
post-decrement load	LDMFA	LDMDA	1	0	0
pre-increment store	STMFA	STMIB	0	1	1
post-increment store	STMEA	STMIA	0	0	1
pre-decrement store	STMFD	STMDB	0	1	0
post-decrement store	STMED	STMDA	0	0	0

Table 5-6: Addressing mode names

5.8.11 Examples

```

LDMFD SP!,{R0,R1,R2} ;unstack 3 registers
STMIA R0,{R0-R15}      ;save all registers
LDMFD SP!,{R15}          ;R15 <- (SP), CPSR unchanged
LDMFD SP!,{R15}^         ;R15 <- (SP), CPSR <- SPSR_mode (allowed
                         ;only in privileged modes)
STMFD R13,{R0-R14}^     ;save user mode regs on stack (allowed
                         ;only in privileged modes)

```

These instructions may be used to save state on subroutine entry, and restore it efficiently on return to the calling routine:

```

STMED SP!,{R0-R3,R14};           ;save R0 to R3 to use as workspace
                                ;and R14 for returning
BL    somewhere             ;this nested call will overwrite R14
LDMED SP!,{R0-R3,R15}          ;restore workspace and return

```



ARM Processor Instruction Set

5.9 Single Data Swap (SWP)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 5-21: Swap instruction*.

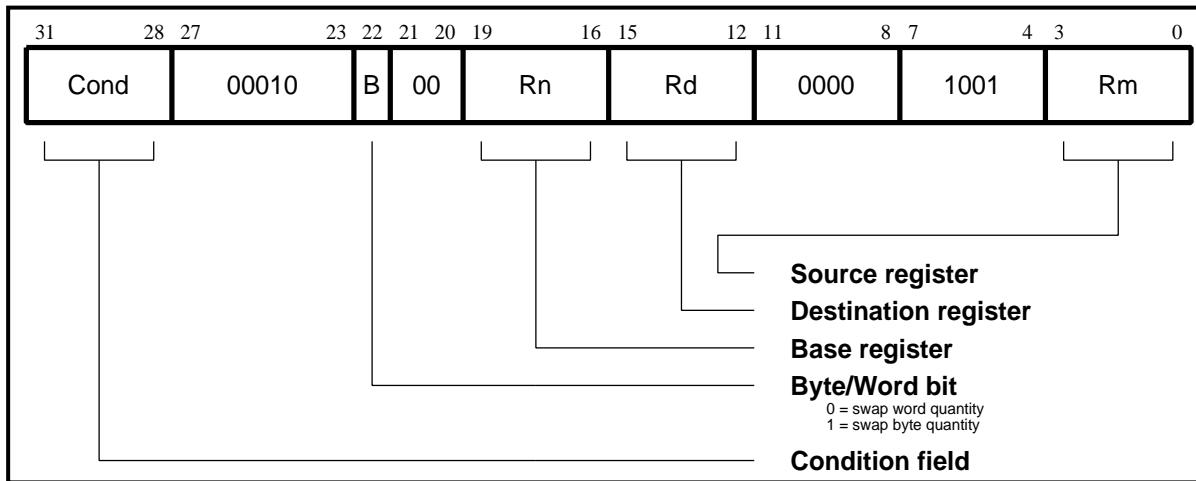


Figure 5-21: Swap instruction

Data swap instruction

The data swap instruction is used to swap a byte or word quantity between a register and external memory. This instruction is implemented as a memory read followed by a memory write which are “locked” together (the processor cannot be interrupted until both operations have completed, and the memory manager is warned to treat them as inseparable). This class of instruction is particularly useful for implementing software semaphores.

Swap address

The swap address is determined by the contents of the base register (Rn). The processor first reads the contents of the swap address. Then it writes the contents of the source register (Rm) to the swap address, and stores the old memory contents in the destination register (Rd). The same register can be specified as both the source and the destination.

ARM710 lock feature

The ARM7500FE does not use the lock feature available in the ARM710 macrocell. You must take care to ensure that control of the memory is not removed from the ARM processor while it is performing this instruction.

5.9.1 Bytes and words

This instruction class may be used to swap a byte (B=1) or a word (B=0) between an ARM processor register and memory. The SWP instruction is implemented as a LDR followed by a STR and the action of these is as described in the section on single data transfers. In particular, the description of Big and Little Endian configuration applies to the SWP instruction.

5.9.2 Use of R15

Do not use R15 as an operand (Rd, Rn or Rs) in a SWP instruction.

5.9.3 Data aborts

If the address used for the swap is unacceptable to the MMU, it will cause an abort. This can happen on either the read or write cycle (or both), and, in either case, the Data Abort trap will be taken. It is up to the system software to resolve the cause of the problem. The instruction can then be restarted and the original program continued.

5.9.4 Instruction cycle times

Swap instructions take 1 instruction fetch, 1 data read, 1 data write and 1 internal cycle. For more information see *5.17 Instruction Speed Summary* on page 5-47.

5.9.5 Assembler syntax

<SWP>{cond}{B} Rd,Rm,[Rn]	
{cond}	two-character condition mnemonic, see <i>Figure 5-2: Condition codes</i> on page 5-2
{B}	if B is present then byte transfer, otherwise word transfer
Rd,Rm,Rn	are expressions evaluating to valid register numbers

5.9.6 Examples

SWP R0,R1,[R2]	;load R0 with the word addressed by R2, and ;store R1 at R2
SWPB R2,R3,[R4]	;load R2 with the byte addressed by R4, and ;store bits 0 to 7 of R3 at R4
SWPEQ R0,R0,[R1]	;conditionally swap the contents of R1 ;with R0



ARM Processor Instruction Set

5.10 Software Interrupt (SWI)

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 5-22: Software interrupt instruction*. The software interrupt instruction is used to enter Supervisor mode in a controlled manner. The instruction causes the software interrupt trap to be taken, which effects the mode change. The PC is then forced to a fixed value (0x08) and the CPSR is saved in SPSR_svc. If the SWI vector address is suitably protected (by external memory management hardware) from modification by the user, a fully protected operating system may be constructed.

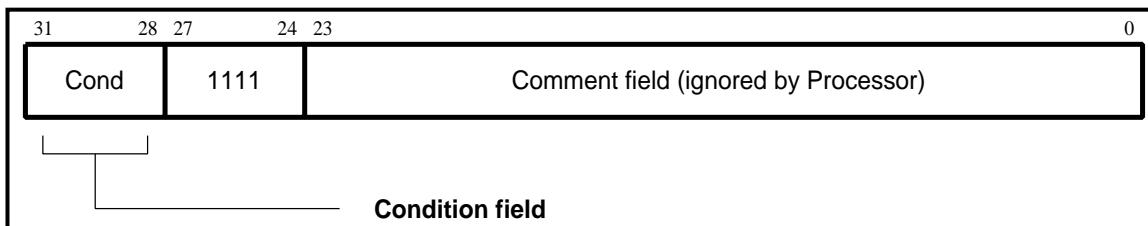


Figure 5-22: Software interrupt instruction

5.10.1 Return from the supervisor

The PC is saved in R14_svc upon entering the software interrupt trap, with the PC adjusted to point to the word after the SWI instruction. MOVS PC,R14_svc will return to the calling program and restore the CPSR.

Note: The link mechanism is not re-entrant, so if the supervisor code wishes to use software interrupts within itself it must first save a copy of the return address and SPSR.

5.10.2 Comment field

The bottom 24 bits of the instruction are ignored by the processor, and may be used to communicate information to the supervisor code. For instance, the supervisor may look at this field and use it to index into an array of entry points for routines which perform the various supervisor functions.

5.10.3 Instruction cycle times

Software interrupt instructions take 3 instruction fetches. For more information see *5.17 Instruction Speed Summary* on page 5-47.

5.10.4 Assembler syntax

SWI {cond} <expression>	
{cond}	two-character condition mnemonic, see <i>Figure 5-2: Condition codes</i> on page 5-2
<expression>	is evaluated and placed in the comment field (ignored by the ARM processor).

5.10.5 Examples

SWI ReadC	;get next character from read stream
SWI WriteI+"k"	;output a "k" to the write stream

ARM Processor Instruction Set

```
SWINE 0           ;conditionally call supervisor  
                  ;with 0 in comment field
```

The above examples assume that suitable supervisor code exists, for instance:

```
0x08 B Supervisor      ;SWI entry point
```

```
EntryTable          ;addresses of supervisor routines  
DCD ZeroRtn  
DCD ReadCRtn  
DCD WriteIRtn  
...  
Zero    EQU      0  
ReadC   EQU      256  
WriteI  EQU      512
```

Supervisor

```
;SWI has routine required in bits 8-23 and data (if any) in bits  
;0-7.  
;Assumes R13_svc points to a suitable stack
```

```
STMFD R13,{R0-R2,R14}; save work registers and return address  
LDR  R0,[R14,#-4]    ;get SWI instruction  
BIC  R0,R0,#0xFF000000;  
                  ;clear top 8 bits  
MOV  R1,R0,LSR#8     ;get routine offset  
ADR  R2,EntryTable  ;get start address of entry table  
LDR  R15,[R2,R1,LSL#2];  
                  ;branch to appropriate routine  
  
WriteIRtn          ;enter with character in R0 bits 0-7  
...  
LDMFD R13,{R0-R2,R15}^;  
                  ;restore workspace and return  
                  ;restoring processor mode and flags
```



ARM Processor Instruction Set

5.11 Coprocessor Instructions on the ARM Processor

The core ARM processor in the ARM7500FE, unlike some other ARM processors, does not have an external coprocessor interface. It supports 2 on-chip coprocessors:

- the FPA
- on-chip control coprocessor, #15, which is used to program the on-chip control registers

For coprocessor instructions supported by the FPA, see *Chapter 10: Floating-Point Instruction Set*.

Coprocessor #15 supports only the Coprocessor Register instructions MRC and MCR.

Note: *Sections 5.12 through 5.14 describe non-FPA coprocessor instructions only.*

All other coprocessor instructions will cause the undefined instruction trap to be taken on the ARM processor. These coprocessor instructions can be emulated in software by the undefined trap handler. Even though external coprocessors cannot be connected to the ARM processor, the coprocessor instructions are still described here in full for completeness. It must be kept in mind that any external coprocessor referred to will be a software emulation.

5.12 Coprocessor Data Operations (CDP)

Use of the CDP instruction on the ARM processor (except for the defined FPA instructions) will cause an undefined instruction trap to be taken, which may be used to emulate the coprocessor instruction.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 5-23: Coprocessor data operation instruction*.

This class of instruction is used to tell a coprocessor to perform some internal operation. No result is communicated back to the processor, and it will not wait for the operation to complete. The coprocessor could contain a queue of such instructions awaiting execution, and their execution can overlap other activity allowing the coprocessor and the processor to perform independent tasks in parallel.

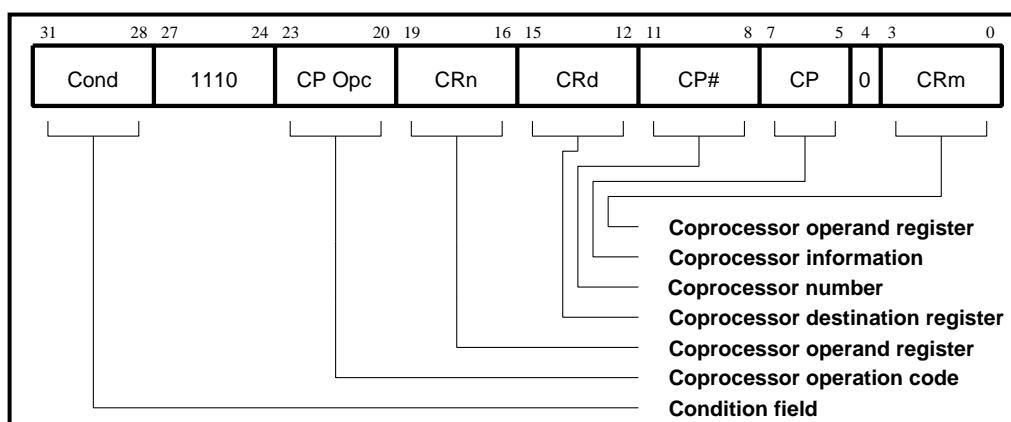


Figure 5-23: Coprocessor data operation instruction

5.12.1 The coprocessor fields

Only bit 4 and bits 24 to 31 are significant to the processor; the remaining bits are used by coprocessors. The above field names are used by convention, and particular coprocessors may redefine the use of all fields except CP# as appropriate. The CP# field is used to contain an identifying number (in the range 0 to 15) for each coprocessor, and a coprocessor will ignore any instruction which does not contain its number in the CP# field.

The conventional interpretation of the instruction is that the coprocessor should perform an operation specified in the CP Opc field (and possibly in the CP field) on the contents of CRn and CRm, and place the result in CRd.

5.12.2 Instruction cycle times

All non-FPA CDP instructions are emulated in software: the number of cycles taken will depend on the coprocessor support software.

5.12.3 Assembler syntax

CDP{cond} p#, <expression1>, cd, cn, cm{, <expression2>}	
{cond}	two character condition mnemonic, see <i>Figure 5-2: Condition codes</i> on page 5-2
p#	the unique number of the required coprocessor
<expression1>	evaluated to a constant and placed in the CP Opc field
cd, cn and cm	evaluate to the valid coprocessor register numbers CRd, CRn and CRm respectively
<expression2>	where present, is evaluated to a constant and placed in the CP field

5.12.4 Examples

```
CDP    p1,10,c1,c2,c3 ;request coproc 1 to do operation 10  
          ;on CR2 and CR3, and put the result in CR1  
CDPEQ p2,5,c1,c2,c3,2;  
          ;if Z flag is set request coproc 2 to do  
          ;operation 5 (type 2) on CR2 and CR3,  
          ;and put the result in CR1
```



ARM Processor Instruction Set

5.13 Coprocessor Data Transfers (LDC, STC)

Use of the LDC or STC instruction on the ARM processor (except for the defined FPA instructions) will cause an undefined instruction trap to be taken, which may be used to emulate the coprocessor instruction.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 5-24: Coprocessor data transfer instructions*.

This class of instruction is used to load (LDC) or store (STC) a subset of a coprocessors's registers directly to memory. The processor is responsible for supplying the memory address, and the coprocessor supplies or accepts the data and controls the number of words transferred.

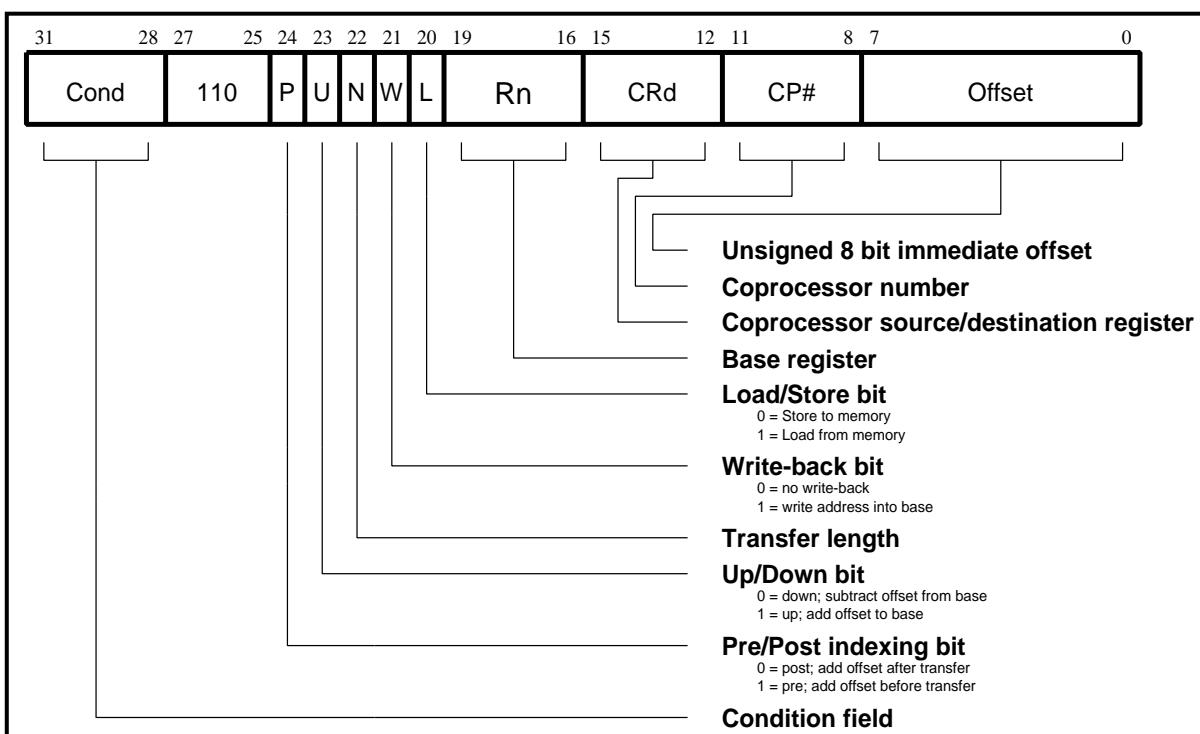


Figure 5-24: Coprocessor data transfer instructions

5.13.1 The coprocessor fields

The CP# field is used to identify the coprocessor which is required to supply or accept the data, and a coprocessor will only respond if its number matches the contents of this field.

The CRd field and the N bit contain information for the coprocessor which may be interpreted in different ways by different coprocessors, but by convention CRd is the register to be transferred (or the first register where more than one is to be transferred), and the N bit is used to choose one of two transfer length options.

For example:

- N=0 could select the transfer of a single register
- N=1 could select the transfer of all the registers for context switching.

5.13.2 Addressing modes

The processor is responsible for providing the address used by the memory system for the transfer, and the addressing modes available are a subset of those used in single data transfer instructions. Note, however, that the immediate offsets are 8 bits wide and specify word offsets for coprocessor data transfers, whereas they are 12 bits wide and specify byte offsets for single data transfers.

The 8 bit unsigned immediate offset is shifted left 2 bits and either added to (U=1) or subtracted from (U=0) the base register (Rn); this calculation may be performed either before (P=1) or after (P=0) the base is used as the transfer address. The modified base value may be overwritten back into the base register (if W=1), or the old value of the base may be preserved (W=0).

Note: Post-indexed addressing modes require explicit setting of the W bit, unlike LDR and STR which always write-back when post-indexed.

The value of the base register, modified by the offset in a pre-indexed instruction, is used as the address for the transfer of the first word. The second word (if more than one is transferred) will go to or come from an address one word (4 bytes) higher than the first transfer, and the address will be incremented by one word for each subsequent transfer.

5.13.3 Address alignment

The base address should normally be a word aligned quantity. The bottom 2 bits of the address will appear on **A[1:0]** and might be interpreted by the memory system.

5.13.4 Use of R15

If Rn is R15, the value used will be the address of the instruction plus 8 bytes. Base write-back to R15 must not be specified.

5.13.5 Data aborts

If the address is legal but the memory manager generates an abort, the data trap will be taken. The write-back of the modified base will take place, but all other processor state will be preserved. The coprocessor is partly responsible for ensuring that the data transfer can be restarted after the cause of the abort has been resolved, and must ensure that any subsequent actions it undertakes can be repeated when the instruction is retried.

5.13.6 Instruction cycle times

All non-FPA LDC instructions are emulated in software: the number of cycles taken will depend on the coprocessor support software.



ARM Processor Instruction Set

5.13.7 Assembler syntax

<LDC | STC> {cond} {L} p#,cd,<Address>
LDC load from memory to coprocessor
STC store from coprocessor to memory
{L} when present perform long transfer (N=1), otherwise perform short transfer (N=0)
{cond} two-character condition mnemonic, see *Figure 5-2: Condition codes* on page 5-2
p# the unique number of the required coprocessor
cd is an expression evaluating to a valid coprocessor register number that is placed in the CRd field

<Address> can be:

- 1 An expression which generates an address:

<expression>

The assembler will attempt to generate an instruction using the PC as a base and a corrected immediate offset to address the location given by evaluating the expression. This will be a PC relative, pre-indexed address. If the address is out of range, an error will be generated.

- 2 A pre-indexed addressing specification:

[Rn] offset of zero
[Rn,<#expression>]{!} offset of <expression> bytes

- 3 A post-indexed addressing specification:

[Rn],<#expression> offset of <expression> bytes

Rn is an expression evaluating to a valid processor register number.
Note, if Rn is R15 then the assembler will subtract 8 from the offset value to allow for processor pipelining.

{!} write back the base register (set the W bit) if ! is present

5.13.8 Examples

```
LDC    p1,c2,table      ;load c2 of coproc 1 from address table,  
                      ;using a PC relative address.  
STCEQLP2,c3,[R5,#24]!  ;conditionally store c3 of coproc 2  
                      ;into an address 24 bytes up from R5,  
                      ;write this address back to R5, and use  
                      ;long transfer  
                      ;option (probably to store multiple  
                      ;words)
```

Note: Though the address offset is expressed in bytes, the instruction offset field is in words. The assembler will adjust the offset appropriately.

5.14 Coprocessor Register Transfers (MRC, MCR)

Use of the MRC or MCR instruction on the ARM processor to a coprocessor other than to the FPA or to coprocessor #15 will cause an undefined instruction trap to be taken, which may be used to emulate the coprocessor instruction.

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction encoding is shown in *Figure 5-25: Coprocessor register transfer instructions*.

This class of instruction is used to communicate information directly between the ARM processor and a coprocessor. An example of a coprocessor to processor register transfer (MRC) instruction would be a FIX of a floating point value held in a coprocessor, where the floating point number is converted into a 32-bit integer within the coprocessor, and the result is then transferred to a processor register. A FLOAT of a 32-bit value in a processor register into a floating point value within the coprocessor illustrates the use of a processor register to coprocessor transfer (MCR).

An important use of this instruction is to communicate control information directly from the coprocessor into the processor CPSR flags. As an example, the result of a comparison of two floating point values within a coprocessor can be moved to the CPSR to control the subsequent flow of execution.

Note: *The ARM processor has an internal coprocessor (#15) for control of on-chip functions. Accesses to this coprocessor are performed during coprocessor register transfers.*

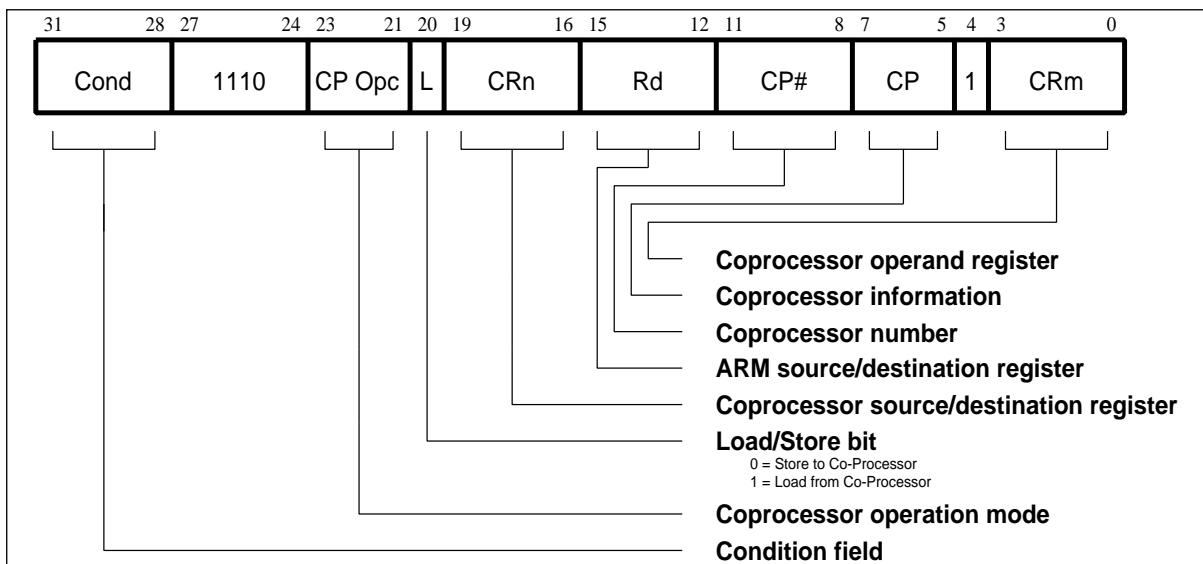


Figure 5-25: Coprocessor register transfer instructions

5.14.1 The coprocessor fields

The CP# field is used, as for all coprocessor instructions, to specify which coprocessor is being called upon.

The CP Opc, CRn, CP and CRm fields are used only by the coprocessor, and the interpretation presented here is derived from convention only. Other interpretations are allowed where the coprocessor functionality is incompatible with this one. The



ARM Processor Instruction Set

conventional interpretation is that the CP Opc and CP fields specify the operation the coprocessor is required to perform, CRn is the coprocessor register which is the source or destination of the transferred information, and CRm is a second coprocessor register which may be involved in some way which depends on the particular operation specified.

5.14.2 Transfers to R15

When a coprocessor register transfer to the ARM processor has R15 as the destination, bits 31, 30, 29 and 28 of the transferred word are copied into the N, Z, C and V flags respectively. The other bits of the transferred word are ignored, and the PC and other CPSR bits are unaffected by the transfer.

5.14.3 Transfers from R15

A coprocessor register transfer from the ARM processor with R15 as the source register will store the PC+12.

5.14.4 Instruction cycle times

Access to the internal configuration register takes 3 internal cycles. All non-FPA MRC instructions default to software emulation, and the number of cycles taken will depend on the coprocessor support software.

5.14.5 Assembler syntax

<MCR | MRC> {cond} p#, <expression1>, Rd, cn, cm{, <expression2>}

where:

MRC	move from coprocessor to ARM7500FE register (L=1)
MCR	move from ARM7500FE register to coprocessor (L=0)
{cond}	two character condition mnemonic, see <i>Figure 5-2: Condition codes</i> on page 5-2
p#	the unique number of the required coprocessor
<expression1>	evaluated to a constant and placed in the CP Opc field
Rd	is an expression evaluating to a valid ARM processor register number
cn and cm	are expressions evaluating to the valid coprocessor register numbers CRn and CRm respectively
<expression2>	where present is evaluated to a constant and placed in the CP field

5.14.6 Examples

```
MRC    2,5,R3,c5,c6 ;request coproc 2 to perform operation 5  
;on c5 and c6, and transfer the (single  
;32-bit word) result back to R3  
MCR    6,0,R4,c6      ;request coproc 6 to perform operation 0  
;on R4 and place the result in c6  
MRCEQ  3,9,R3,c5,c6,2 ;conditionally request coproc 2 to  
;perform  
;operation 9 (type 2) on c5 and c6, and  
;transfer the result back to R3
```

5.15 Undefined Instruction

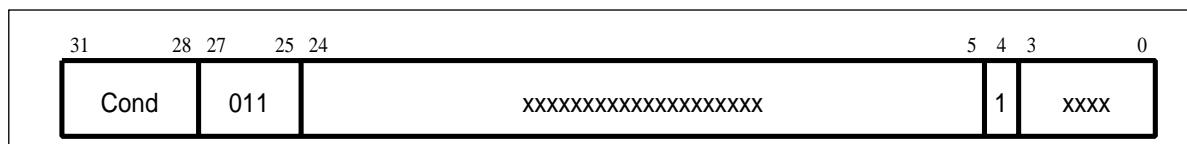


Figure 5-26: *Undefined instruction*

The instruction is only executed if the condition is true. The various conditions are defined at the beginning of this chapter. The instruction format is shown in *Figure 5-26: Undefined instruction* on page 5-43.

If the condition is true, the undefined instruction trap will be taken.

5.15.1 Assembler syntax

At present the assembler has no mnemonics for generating this instruction. If it is adopted in the future for some specified use, suitable mnemonics will be added to the assembler. Until such time, this instruction shall not be used.



ARM Processor Instruction Set

5.16 Instruction Set Examples

The following examples show ways in which the basic ARM processor instructions can combine to give efficient code. None of these methods saves a great deal of execution time (although they may save some), mostly they just save code.

5.16.1 Using the conditional instructions

- 1 using conditionals for logical OR

```
CMP      Rn,#p      ;if Rn=p OR Rm=q THEN GOTO Label
BEQ      Label
CMP      Rm,#q
BEQ      Label
can be replaced by
CMP      Rn,#p
CMPNE   Rm,#q      ;if condition not satisfied try other
                  ;test
BEQ      Label
```
- 2 absolute value

```
TEQ      Rn,#0      ;test sign
RSBMI   Rn,Rn,#0 ;and 2's complement if necessary
```
- 3 multiplication by 4, 5 or 6 (run time)

```
MOV      Rc,Ra,LSL#2;
                  ;multiply by 4
CMP      Rb,#5      ; test value
ADDCS   Rc,Rc,Ra ; complete multiply by 5
ADDHI   Rc,Rc,Ra ; complete multiply by 6
```
- 4 combining discrete and range tests

```
TEQ      Rc,#127 ;discrete test
CMPNE   Rc,#" -1;
                  ;range test
MOVLS   Rc,"." ;IF   Rc<=" " OR Rc=ASCII(127)
                  ;THEN Rc:=". "
```
- 5 division and remainder

A number of divide routines for specific applications are provided in source form as part of the ANSI C library provided with the ARM Cross Development Toolkit, available from your supplier. A short general purpose divide routine follows.

ARM Processor Instruction Set

```
;enter with numbers in Ra and Rb
;
MOV      Rcnt,#1 ;bit to control the division
Div1    CMP      Rb,#0x80000000;
        ;move Rb until greater than Ra
        CMPCC   Rb,Ra
        MOVCC   Rb,Rb,ASL#1
        MOVCC   Rcnt,Rcnt,ASL#1
        BCC     Div1
        MOV     Rc,#0
Div2    CMP      Ra,Rb ;test for possible subtraction
        SUBCS  Ra,Ra,Rb ;subtract if ok
        ADDCS  Rc,Rc,Rcnt;
        ;put relevant bit into result
        MOVS   Rcnt,Rcnt,LSR#1;
        ;shift control bit
        MOVNE  Rb,Rb,LSR#1;
        ;halve unless finished
        BNE    Div2
        ;
        ;divide result in Rc
;remainder in Ra
```

5.16.2 Pseudo random binary sequence generator

It is often necessary to generate (pseudo-) random numbers and the most efficient algorithms are based on shift generators with exclusive-OR feedback rather like a cyclic redundancy check generator. Unfortunately the sequence of a 32-bit generator needs more than one feedback tap to be maximal length (i.e. $2^{32}-1$ cycles before repetition), so this example uses a 33-bit register with taps at bits 33 and 20. The basic algorithm is newbit:=bit 33 or bit 20, shift left the 33-bit number and put in newbit at the bottom; this operation is performed for all the newbits needed (ie. 32 bits). The entire operation can be done in 5 S cycles:

```
;enter with seed in Ra (32 bits),
;Rb (1 bit in Rb lsb), uses Rc
;
TST      Rb,Rb,LSR#1 ;top bit into carry
MOVS   Rc,Ra,RRX ;33 bit rotate right
ADC     Rb,Rb,Rb ;carry into lsb of Rb
EOR     Rc,Rc,Ra,LSL#12;
        ;(involved!)
EOR     Ra,Rc,Rc,LSR#20;
        ;(similarly involved!)
;
;new seed in Ra, Rb as before
```



ARM Processor Instruction Set

5.16.3 Multiplication by constant using the barrel shifter

- 1 Multiplication by 2^n (1,2,4,8,16,32..)
MOV R_a, R_b, LSL #n
- 2 Multiplication by 2^{n+1} (3,5,9,17..)
ADD R_a, R_a, R_a, LSL #n
- 3 Multiplication by 2^{n-1} (3,7,15..)
RSB R_a, R_a, R_a, LSL #n
- 4 Multiplication by 6
ADD R_a, R_a, R_a, LSL #1; ;multiply by 3
MOV R_a, R_a, LSL#1; ;and then by 2
- 5 Multiply by 10 and add in extra number
ADD R_a, R_a, R_a, LSL#2; ;multiply by 5
ADD R_a, R_c, R_a, LSL#1; ;multiply by 2
;and add in next digit
- 6 General recursive method for R_b := R_a*C, C a constant:
 - a) If C even, say C = 2^n*D , D odd:
D=1: MOV R_b, R_a, LSL #n
D<>1: {R_b := R_a*D}
MOV R_b, R_b, LSL #n
 - b) If C MOD 4 = 1, say C = 2^n*D+1 , D odd, n>1:
D=1: ADD R_b, R_a, R_a, LSL #n
D<>1: {R_b := R_a*D}
ADD R_b, R_a, R_b, LSL #n
 - c) If C MOD 4 = 3, say C = 2^n*D-1 , D odd, n>1:
D=1: RSB R_b, R_a, R_a, LSL #n
D<>1: {R_b := R_a*D}
RSB R_b, R_a, R_b, LSL #n

This is not quite optimal, but close. An example of its non-optimality is multiply by 45 which is done by:

```
RSB Rb, Ra, Ra, LSL#2; ;multiply by 3
RSB Rb, Ra, Rb, LSL#2; ;multiply by 4*3-1 = 11
ADD Rb, Ra, Rb, LSL# 2; ;multiply by 4*11+1 = 45
```

rather than by:

```
ADD Rb, Ra, Ra, LSL#3; ;multiply by 9
ADD Rb, Rb, Rb, LSL#2; ;multiply by 5*9 = 45
```

5.16.4 Loading a word from an unknown alignment

```
BIC    Rb,Ra,#3           ;enter with address in Ra (32 bits)
LDMIA  Rb,{Rd,Rc}         ;uses Rb, Rc; result in Rd.
AND    Rb,Ra,#3           ;Note d must be less than c e.g. 0,1
;
BIC    Rb,Ra,#3           ;get word aligned address
LDMIA  Rb,{Rd,Rc}         ;get 64 bits containing answer
AND    Rb,Ra,#3           ;correction factor in bytes
MOVS   Rb,Rb,LSL#3        ;...now in bits and test if aligned
MOVNE  Rd,Rd,LSR Rb       ;produce bottom of result word
                           ;(if not aligned)
RSBNE  Rb,Rb,#32          ;get other shift amount
ORRNE  Rd,Rd,Rc,LSL Rb;    ;combine two halves to get result
```

5.16.5 Loading a halfword (Little-endian)

```
LDR    Ra,[Rb,#2]          ;get halfword to bits 15:0
MOV    Ra,Ra,LSL #16        ;move to top
MOV    Ra,Ra,LSR #16        ;and back to bottom
                           ;use ASR to get sign extended version
```

5.16.6 Loading a halfword (Big-endian)

```
LDR    Ra,[Rb,#2]          ;get halfword to bits 31:16
MOV    Ra,Ra,LSR #16        ;and back to bottom
                           ;use ASR to get sign extended version
```

5.17 Instruction Speed Summary

Due to the pipelined architecture of the CPU, instructions overlap considerably. In a typical cycle one instruction may be using the data path while the next is being decoded and the one after that is being fetched. For this reason the following table presents the incremental number of cycles required by an instruction, rather than the total number of cycles for which the instruction uses part of the processor. Elapsed time (in cycles) for a routine may be calculated from these figures which are shown in *Table 5-7: ARM instruction speed summary* on page 5-48.

These figures assume that the instruction is actually executed.

Unexecuted instructions take one instruction fetch cycle.



ARM Processor Instruction Set

Instruction	Cycle count
Data Processing - normal with register specified shift with PC written with register specified shift & PC written	1 instruction fetch 1 instruction fetch and 1 internal cycle 3 instruction fetches 3 instruction fetches and 1 internal cycle
MSR, MRS	1 instruction fetch
LDR - normal if the destination is the PC	1 instruction fetch, 1 data read and 1 internal cycle 3 instruction fetches, 1 data read and 1 internal cycle
STR	1 instruction fetch and 1 data write
LDM - normal if the destination is the PC	1 instruction fetch, n data reads and 1 internal cycle 3 instruction fetches, n data reads and 1 internal cycle
STM	1 instruction fetch and n data writes
SWP	1 instruction fetch, 1 data read, 1 data write and 1 internal cycle
B,BL	3 instruction fetches
SWI, trap	3 instruction fetches
MUL,MLA	1 instruction fetch and m internal cycles
CDP	1 instruction fetch and b internal cycles
LDC	1 instruction fetch, n data reads, and b internal cycles
STC	1 instruction fetch, n data writes, and b internal cycles
MCR	1 instruction fetch and b+1 internal cycles
MRC	1 instruction fetch and b+1 internal cycles

Table 5-7: ARM instruction speed summary

Where:

- n is the number of words transferred.
- m is the number of cycles required by the multiply algorithm, which is determined by the contents of Rs. Multiplication by any number between $2^{(2m-3)}$ and $2^{(2m-1)}-1$ takes 1S+ml cycles for $1 < m > 16$. Multiplication by 0 or 1 takes 1S+1l cycles, and multiplication by any number greater than or equal to $2^{(29)}$ takes 1S+16l cycles. The maximum time for any multiply is thus 1S+16l cycles.
- b is the number of cycles spent in the coprocessor busy-wait loop.

ARM Processor Instruction Set

The time taken for:

- an internal cycle will always be one FCLK cycle
- an instruction fetch and data read will be FCLK if a cache hit occurs, otherwise a full memory access is performed.
- a data write will be FCLK if the write buffer (if enabled) has available space, otherwise the write will be delayed until the write buffer has free space.
If the write buffer is not enabled a full memory access is always performed.
- memory accesses are dealt with elsewhere in the ARM7500FE datasheet.
- coprocessor instructions depends on whether the instruction is executed by:

the FPA	See <i>Chapter 10: Floating-Point Instruction Set</i> for details of floating-point instruction cycle counts.
coprocessor #15	MCR, MRC to registers 0 to 7 only. In this case $b = 0$.
software emulation	For all other coprocessor instructions, the undefined instruction trap is taken.



ARM Processor Instruction Set



ARM assembly language reference card

<code>MOVcdS</code>	<code>reg, arg</code>	copy argument (S = set flags)	<code>Bcd</code>	<code>imm12</code>	branch to imm_{12} words away
<code>MVNcdS</code>	<code>reg, arg</code>	copy bitwise NOT of argument	<code>BLcd</code>	<code>imm12</code>	copy PC to LR, then branch
<code>ANDcdS</code>	<code>reg, reg, arg</code>	bitwise AND	<code>BXcd</code>	<code>reg</code>	copy <code>reg</code> to PC
<code>ORRcdS</code>	<code>reg, reg, arg</code>	bitwise OR	<code>SWIcd</code>	<code>imm24</code>	software interrupt
<code>EORcdS</code>	<code>reg, reg, arg</code>	bitwise exclusive-OR	<code>LDRcdb</code>	<code>reg, mem</code>	loads word/byte from memory
<code>BICcdS</code>	<code>reg, reg_a, arg_b</code>	bitwise reg_a AND (NOT arg_b)	<code>STRcdb</code>	<code>reg, mem</code>	stores word/byte to memory
<code>ADDcdS</code>	<code>reg, reg, arg</code>	add	<code>LDMedium</code>	<code>reg !, mreg</code>	loads into multiple registers
<code>SUBcdS</code>	<code>reg, reg, arg</code>	subtract	<code>STMedium</code>	<code>reg !, mreg</code>	stores multiple registers
<code>RSBcdS</code>	<code>reg, reg, arg</code>	subtract reversed arguments	<code>SWPcdb</code>	<code>reg_d, reg_m, [reg_n]</code>	copies reg_m to memory at reg_n , old value at address reg_n to reg_d
<code>ADCcdS</code>	<code>reg, reg, arg</code>	add with carry flag			
<code>SBCcdS</code>	<code>reg, reg, arg</code>	subtract with carry flag			
<code>RSCcdS</code>	<code>reg, reg, arg</code>	reverse subtract with carry flag			
<code>CMPcd</code>	<code>reg, arg</code>	update flags based on subtraction			
<code>CMNcd</code>	<code>reg, arg</code>	update flags based on addition			
<code>TSTcd</code>	<code>reg, arg</code>	update flags based on bitwise AND			
<code>TEQcd</code>	<code>reg, arg</code>	update flags based on bitwise exclusive-OR			
<code>MULcdS</code>	<code>reg_d, reg_a, reg_b</code>	multiply reg_a and reg_b , places lower 32 bits into reg_d			
<code>MLAcdS</code>	<code>reg_d, reg_a, reg_b, reg_c</code>	places lower 32 bits of $reg_a \cdot reg_b + reg_c$ into reg_d			
<code>UMULLcdS</code>	<code>reg_l, reg_u, reg_a, reg_b</code>	multiply reg_a and reg_b , place 64-bit unsigned result into $\{ reg_u, reg_l \}$			
<code>UMLALcdS</code>	<code>reg_l, reg_u, reg_a, reg_b</code>	place unsigned $reg_a \cdot reg_b + \{ reg_u, reg_l \}$ into $\{ reg_u, reg_l \}$			
<code>SMULLcdS</code>	<code>reg_l, reg_u, reg_a, reg_b</code>	multiply reg_a and reg_b , place 64-bit signed result into $\{ reg_u, reg_l \}$			
<code>SMLALcdS</code>	<code>reg_l, reg_u, reg_a, reg_b</code>	place signed $reg_a \cdot reg_b + \{ reg_u, reg_l \}$ into $\{ reg_u, reg_l \}$			

reg: register

R0 to R15	register according to number
SP	register 13
LR	register 14
PC	register 15

um: update mode

IA/ED	increment, starting from <i>reg</i>
IB/FU	increment, starting from $reg + 4$
DA/ED	decrement, starting from <i>reg</i>
DB/FD	decrement, starting from $reg - 4$

cd: condition code

AL or omitted	always
EQ	equal (zero)
NE	nonequal (nonzero)
CS	carry set (same as HS)
CC	carry clear (same as LO)
MI	minus
PL	positive or zero
VS	overflow set
VC	overflow clear
HS	unsigned higher or same
LO	unsigned lower
HI	unsigned higher
LS	unsigned lower or same
GE	signed greater than or equal
LT	signed less than
GT	signed greater than
LE	signed less than or equal

arg: right-hand argument

<code>#imm₈*</code>	immediate (rotated into 8 bits)
<code>reg</code>	register
<code>reg, shift</code>	register shifted by distance

mem: memory address

<code>[reg, #±imm₁₂]</code>	<i>reg</i> offset by constant
<code>[reg, ±reg]</code>	<i>reg</i> offset by variable bytes
<code>[reg_a, ±reg_b, shift]</code>	reg_a offset by shifted variable reg_b [†]
<code>[reg, #±imm₁₂] !</code>	update <i>reg</i> by constant, then access memory
<code>[reg, ±reg] !</code>	update <i>reg</i> by variable bytes, access memory
<code>[reg, ±reg, shift] !</code>	update <i>reg</i> by shifted variable [†] , access memory
<code>[reg], #±imm₁₂</code>	access address <i>reg</i> , then update <i>reg</i> by offset
<code>[reg], ±reg</code>	access address <i>reg</i> , then update <i>reg</i> by variable
<code>[reg], ±reg, shift</code>	access address <i>reg</i> , update <i>reg</i> by shifted variable [†]

[†] shift distance must be by constant

shift: shift register value

<code>LSL #imm₅</code>	shift left 0 to 31
<code>LSR #imm₅</code>	logical shift right 1 to 32
<code>ASR #imm₅</code>	arithmetic shift right 1 to 32
<code>ROR #imm₅</code>	rotate right 1 to 31
<code>RRX</code>	rotate carry bit into top bit
<code>LSL reg</code>	shift left by register
<code>LSR reg</code>	logical shift right by register
<code>ASR reg</code>	arithmetic shift right by register
<code>ROR reg</code>	rotate right by register

Memoria Virtuale

Parte della memoria di massa utilizzata come memoria principale.

Trasferimento trasparente dei dati tra memoria di massa e memoria principale.

Motivazioni:

- **simulare** una memoria principale più ampia agevolando l'esecuzione di programmi di grosse dimensioni in sistemi operativi **multi-tasking**;
- realizzare meccanismi di protezione, impedendo a un processo l'accesso ai dati di un altro processo.

Cenni storici

Prime soluzioni alla carenza di memoria (anni 50, memoria disponibile di pochi KB):
memoria non sufficiente a contenere tutto il programma,
programmi divisi in **overlay** (moduli), **esplicitamente** caricati e scaricati dalla memoria.

Memoria virtuale nata per realizzare una gestione trasparente degli overlay.

- Ideazione: anni 60 (Manchester).
- Utilizzazione: anni 70.

Idea base

Con la memoria virtuale, si **diversificano**

- lo **spazio d'indirizzamento**: indirizzi utilizzabili dal programma, sostanzialmente illimitati
- **posizioni in memoria**: indirizzi fisici delle locazioni in memoria, limitati dalla memoria disponibile.

Diverse realizzazioni:

- **paginazione**
- **segmentazione**.

Paginazione

Similitudini con le tecniche di organizzazione della memoria cache.

Spazio di indirizzamento diviso in **pagine**

- **tutte** con la stessa dimensione
- in cui la dimensione è una **potenza di 2**.

Posizionamento fisico delle pagine:

- alcune presenti in memoria principale e in memoria di massa
- alcune presenti solo in memoria di massa
- altre vuote e non esistenti.

Paginazione: accesso alla memoria

Dato l'indirizzo di una locazione in memoria

- si determina la pagina contenente la parola
- si controlla se la pagina è presente nella memoria principale
- se non presente (**page fault**) la pagina viene caricata in memoria dal sistema operativo
- si calcola la posizione della locazione cercata e si accede a essa.

Meccanismo invisibile al programma utente.

Indirizzi di memoria virtuale

Si distingue tra:

- **indirizzo virtuale** utilizzato all'interno del programma
- **indirizzo fisico** utilizzato dall'hardware.

Parallelamente:

- **pagina virtuale** accessibile nella memoria
- **pagina fisica** residente in memoria.

Mappatura degli indirizzi

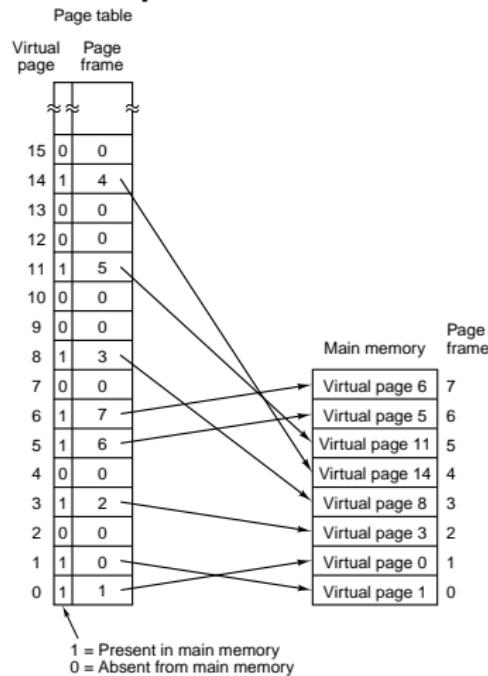
Meccanismi più sofisticati rispetto alla memoria cache gestiti a livello di sistema operativo: ogni pagina può risiedere in un qualunque luogo della memoria principale.

Viene utilizzata una **tabella delle pagine** (**Page Table**), chiamata anche **mappa della memoria** (**PMT: Page-Map Table**), che ad ogni pagina **virtuale** associa

- un bit di presenza in memoria principale
- se asserito, un indirizzo di inizio pagina (posizione fisica in memoria).

Tabella delle pagine

Associa a ogni numero di pagina:
bit di presenza, indirizzo base.

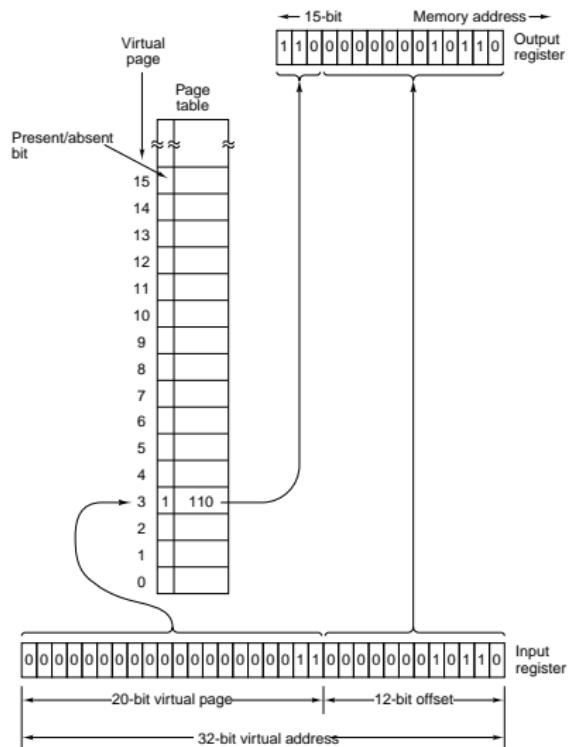


Accesso a una parola in memoria

- Si scomponete l'indirizzo in **numero di pagina** e **offset**
- con il numero di pagina si accede alla tabella delle pagine e si controlla se la pagina è presente in memoria principale
- se la pagina è presente, la tabella delle pagine fornisce l'indirizzo base della pagina (l'indirizzo in memoria della prima parola nella pagina)
- giustapponendo indirizzo base e offset si ottiene l'indirizzo in memoria della parola
- se la pagina è assente è sollevata una chiamata *page fault* al S.O.: si carica la pagina dalla memoria di massa; si aggiorna la tabella.

Esempio di calcolo indirizzo fisico

Indirizzo fisico = indirizzo di inizio pagina + offset.



Esempio di calcolo indirizzo fisico

- Indirizzi di memoria di 32 bit
- Spazio di indirizzamento: 4 GB
- Suddivisione: 1 M di pagine di 4 kB ciascuna.

Indirizzamento:

- i primi 20 bit identificano il **numero della pagina**
- i 12 bit rimanenti identificano la parola all'interno della pagina: **offset**.

Pagine: molto più ampie delle linee di cache.

Page frame

Regione della memoria principale contenente una pagina fisica.

Alcune regioni della memoria sono ad accesso diretto (senza paginazione). Es.: page table; indirizzi associati al memory mapped I/O.

Page	Virtual addresses
~	~
15	61440 – 65535
14	57344 – 61439
13	53248 – 57343
12	49152 – 53247
11	45056 – 49151
10	40960 – 45055
9	36864 – 40959
8	32768 – 36863
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

Page frame	Physical addresses
Bottom 32K of main memory	
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

Page fault

Ogni **page fault** genera una chiamata (trap) al sistema operativo che

- cerca un page frame vuoto
- se non esiste, svuota un page frame:
 - least recently used (**LRU**) o una sua approssimazione, si sfrutta la **località temporale**
 - first-in first-out (**FIFO**): più semplice da implementare
- se la pagina da scaricare è stata modificata (meccanismo simile a quello del dirty bit)
aggiorna la copia in memoria di massa
- carica la pagina cercata nella memoria principale.

Assegnazione delle pagine

Processo: istanza in esecuzione di un programma.
I page frame devono essere ripartiti tra i processi.

Working set: insieme di indirizzi di uso corrente,
tipicamente quelli usati nell'ultimo secondo.

Tutte le pagine del working set dovrebbero essere
contenute in memoria principale pena l'insorgenza
del **thrashing**: generazione di continui page fault.

Il thrashing può essere causato da una memoria
principale insufficiente e/o da algoritmi di
paginazione inadatti.

Paginazione: organizzazione

Costruzione della **page table**: **una per processo**.

All'avvio di un programma, due possibili scelte:

- **un sottoinsieme** di pagine viene caricato in memoria principale
- **nessuna** pagina caricata in memoria principale: **demanding paging**, paginazione su richiesta.

L'esecuzione del programma determina le pagine caricate. In condizioni ottimali esse costituiranno un sovrainsieme del working set.

Paginazione: realizzazione hw/sw

- In parte gestita a livello hardware da una **Memory Management Unit** (MMU): esegue la mappatura dell'indirizzo logico nell'indirizzo fisico.
MMU è contenuta nel chip del processore, ma dal punto di vista logico è un'unità indipendente.
- In parte gestita a livello software: page fault causa una trap che viene servita dal sistema operativo.

Memorizzazione della page table

La page table dev'essere accessibile dalla memoria principale, altrimenti ogni accesso alla memoria virtuale avrebbe il costo di almeno un accesso alla memoria di massa.

Massimizzazione dell'efficienza: parte della page table è contenuta in una memoria cache dedicata denominata **Translation Lookside Buffer (TLB)**, contenuta all'interno della MMU.

Ogni processo in esecuzione ha una sua page table di circa 10 MB. La cache **non** può contenere tutte le page table.

Dimensioni della pagina

Conviene avere pagine grandi per

- ridurre l'accesso al disco
- favorire la località spaziale
- avere page table piccole.

Conviene avere pagine piccole per

- sfruttare meglio la memoria
- limitare il costo dei page fault
- limitare la **frammentazione**: l'ultima pagina di ogni programma è utilizzata solo in parte.

Anni '70: 0.5-1 KB.

Attualmente: 4 KB ~ 4 MB.

Limiti della paginazione

Paginazione: spazio di indirizzamento virtuale unico e non strutturato, condiviso da tutti i processi.

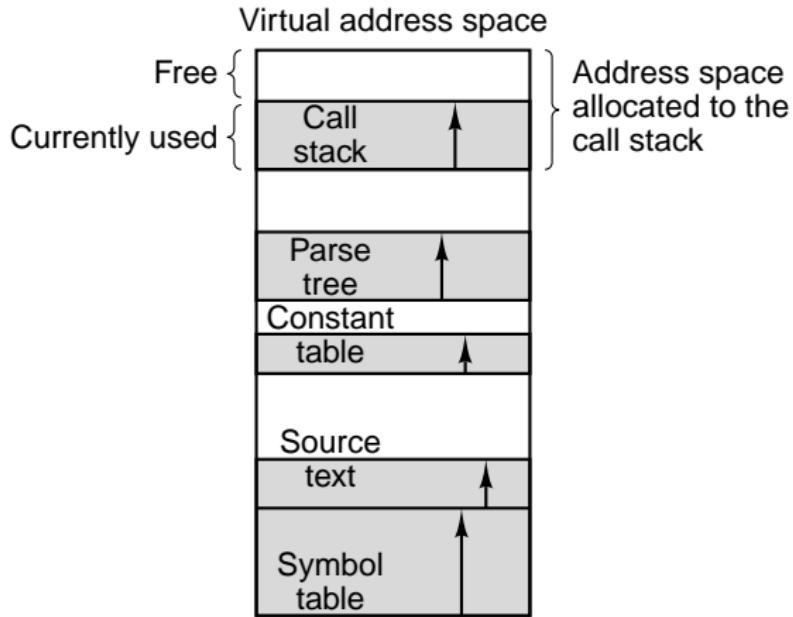
Viceversa, i programmi (e il S.O.) devono strutturare la memoria in parti logicamente distinte:

- aree di memoria dei programmi eseguiti dal programma utente JVM (4 regioni distinte)
- processi eseguiti dal sistema operativo
- procedure eseguite dai programmi utente
- utilizzo di strutture dati dinamiche.

In più, occorre realizzare meccanismi di **restrizione all'accesso alla memoria** da parte dei vari processi.

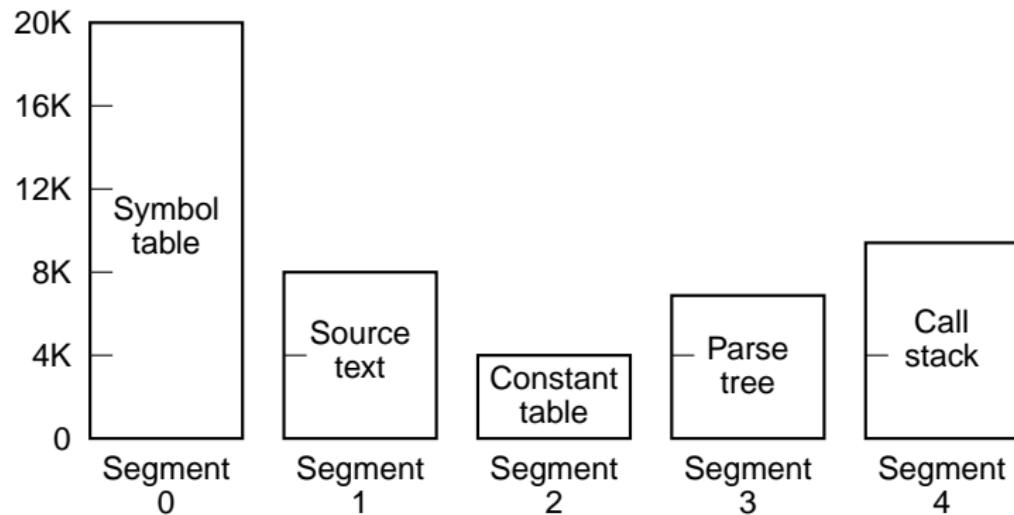
Suddivisione logica della memoria

Es.: spazio dei dati necessari a un compilatore.



Suddivisione logica della memoria

Segmenti logicamente indipendenti in regioni fisicamente separate della memoria



Segmentazione

Realizzazione di una memoria virtuale segmentata, dove

- il programma opera su dati suddivisi in unità logiche
- il meccanismo di virtualizzazione
 - mappa i segmenti sulla memoria fisica
 - li alloca a seconda delle necessità in memoria principale o di massa
- lo spazio virtualmente occupato dall'insieme dei segmenti può superare quello della memoria fisica (indirizzi virtuali non mappati sulla memoria fisica). Es.: heap per strutture dati dinamiche.

Segmentazione e paginazione

- Segmento — Pagina
- allocato dal — trasparente al programma
- unità logica — unità fisica
- ampiezza variabile — ampiezza fissa.

Il **descrittore** del segmento deve contenere

- il **numero** del segmento, che ne individua la posizione in memoria virtuale
- il **livello di protezione**, che ne individua l'accessibilità.

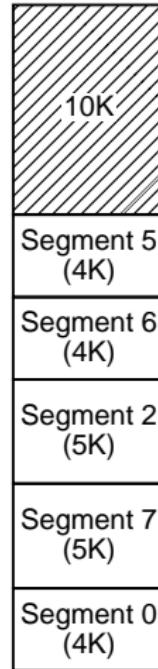
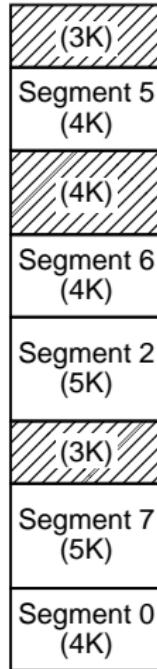
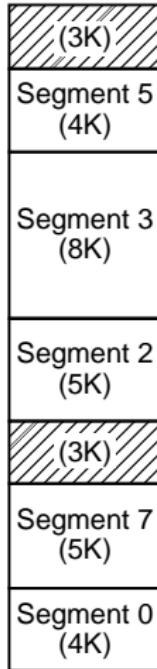
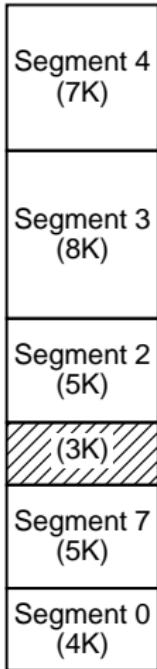
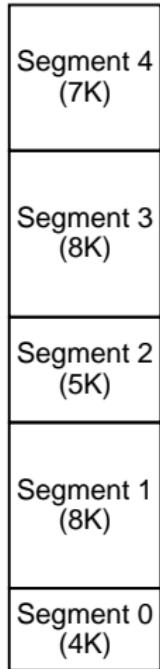
Swapping

In una memoria non paginata i segmenti quando necessario sono

- caricati **per intero** in memoria principale
- scaricati per far posto a nuovi segmenti.

Data la dimensione variabile dei segmenti, lo swapping solleva problemi dovuti alla ricerca di spazi liberi in memoria principale.

Swapping: esempio



(a)

(b)

(c)

(d)

(e)

Ricerca di spazi liberi

Necessità di opportune strategie di scelta degli spazi dove inserire un nuovo segmento.

Tra gli spazi utilizzabili si può selezionare

- quello più piccolo: **best fit**, complicato da implementare
- il primo spazio trovato: **first fit**, semplice e con buone prestazioni.

Rimozione dei segmenti dalla memoria principale: devono tener conto della lunghezza dei segmenti e del loro uso. Piuttosto complessi.

Frammentazione esterna in memoria

A causa dello swapping, la memoria principale viene divisa in zone non contigue in cui si creano spazi liberi di piccole dimensioni (**checkerboarding**).

Rimedi:

- unione di spazi liberi contigui
- spostamento e riunione di segmenti (**compattazione**).

La compattazione richiede tempo e non può essere utilizzata frequentemente.

La segmentazione non paginata è quindi **problematica**.

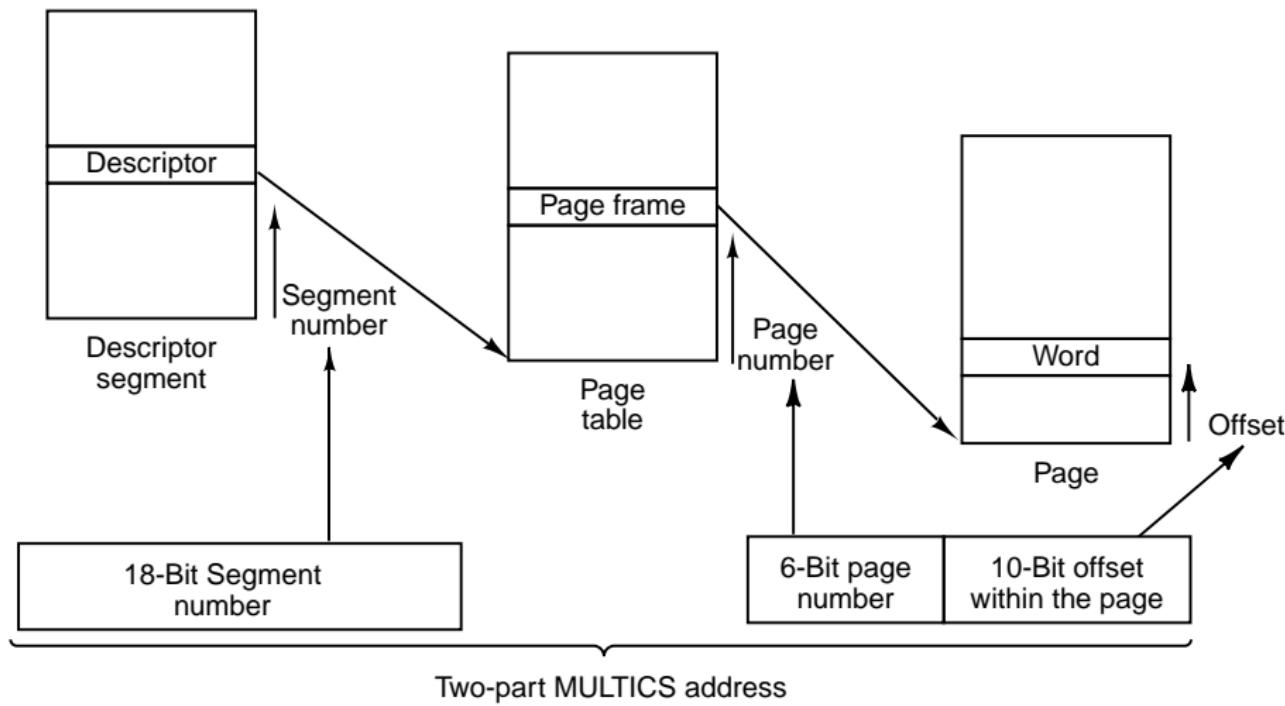
Segmentazione paginata

Combinazione di segmentazione e paginazione:
ogni segmento diviso in pagine.

- Risolve i problemi di frammentazione esterna.
- Introduce un ulteriore livello di suddivisione.
- Prima implementazione: MULTICS (1965).
- Idee fondamentali di MULTICS ereditate dalla famiglia x86.

Paging in MULTICS

Indirizzamento a partire da un segmento descrittore



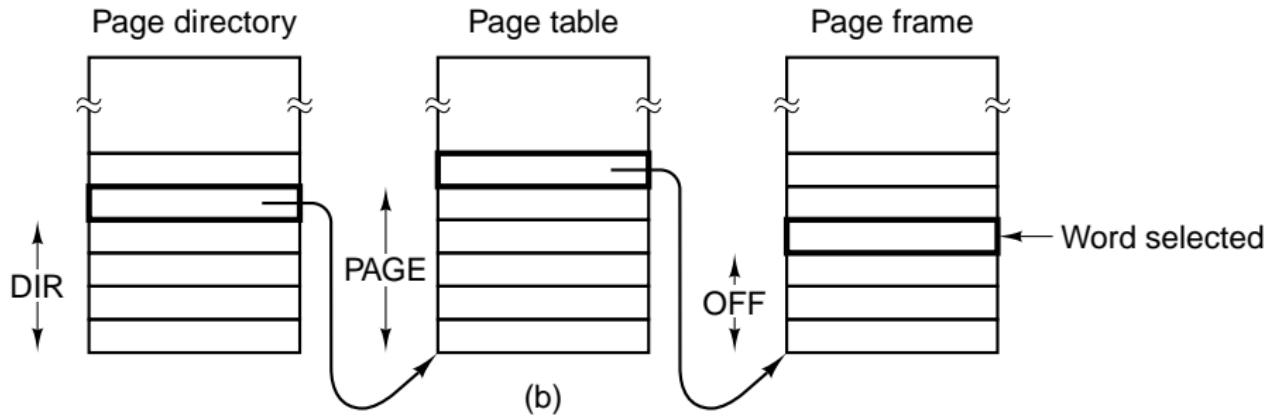
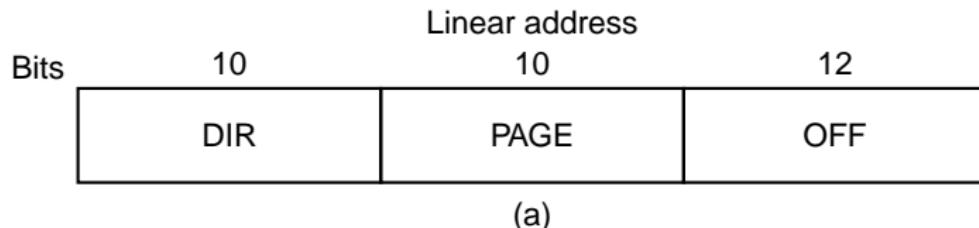
Modello di memoria nei processori x86 (cenni)

Modello di memoria retrocompatibile: memoria semplice oppure segmentata e/o paginata.

- Esistenza di **segmenti codice** e **segmenti dati**
- Segmenti **locali** al processo e **globali** nel sistema
- **Local Descriptor Table** (LDT)
- **Global Descriptor Table** (GDT)
- Paginazione non attivata: accesso diretto alla memoria (segmentazione pura)
- Paginazione attivata: accesso tramite paginazione (segmentazione paginata).

Calcolo dell'indirizzo lineare

Esistenza di una page directory (tabella delle tabelle)

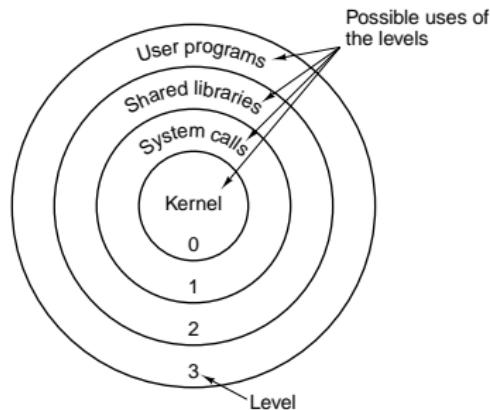


Organizzazione delle page tables

- Ogni segmento accede a **una** page table.
- Page table divisa in più sottotabelle.
- Problemi di dimensione: parte della page table contenuta in memoria secondaria.
- Tempi di accesso: nella MMU una memoria cache per i collegamenti segmento–pagina usati più recentemente.
- Utilizzo di un unico segmento → paginazione pura.

Protezione

Un processo che ha livello di privilegio N (come da PSW, program status word) **non** può accedere a segmenti con livello di protezione minore di N.



Un processo accede a livelli inferiori di protezione solo attraverso opportune **call gate**: procedure di sistema richiamabili da **punti di ingresso ufficiali**.

Laboratorio di architettura degli elaboratori

ASSEMBLY PER ARM Lezione 8

- CONTATTI
- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

Esercizio 8.1

Scrivere un programma assembly che, dopo aver immesso in memoria mediante una dichiarazione nella sezione .data quattro numeri interi $n1$, $n2$, $n3$, $n4$, inserisca nei registri r0 ,r1 ,r2 ,r3 ,r4 rispettivamente i seguenti valori:

- la somma dei valori $n1$, $n2$, $n3$, $n4$
- la media dei valori $n1$, $n2$, $n3$, $n4$ (arrotondando sempre verso $-\infty$)
- il risultato dell'operazione $(2^{10} + 1) * n1$
- il resto r della divisione di $n1$ per 16, inteso come $n1 = (n1/16)*16+r$
- il segno di $n1$, ossia il valore 0 se $n1$ è positivo o il valore 1 se $n1$ è negativo.

Esercizio 8.2

```
.text

mov r0, #0x1000
add r0, r0, #0x10
ldr r1, [r0]
str r1, [r0, #8]

mov r0, #0
mov r0, #1
mov r0, #2

;exit
swi 0x11

.end
```

Alla fine nel registro r0 che valore è presente? Perché?

Funzioni, metodi, subroutine

Costrutti base della programmazione.

- Strutturano il programma in parti invocabili.
- Modularizzano i programmi.
- Permettono il riutilizzo del codice.
- Necessarie per gestire codice complesso.
- Necessarie per realizzare la **ricorsione**.

Utilizzate in programmazione assembly.

Supportate dai linguaggi macchina, ma non automaticamente disponibili in assembly: una **chiamata di procedura** corrisponde a una sequenza di istruzioni macchina **non risolte automaticamente da istruzioni assembly**.

Procedure

Una chiamata di procedura comporta:

- passaggio del controllo al codice della procedura,
- passaggio di parametri,
- allocazione di spazio di memoria per le variabili locali.

L'uscita da una procedura comporta:

- recupero di spazio in memoria,
- restituzione del controllo e del risultato al chiamante.

Implementazione in ARM con istruzioni elementari.

Mostriamo come le chiamate di procedura vengono realizzate nella macchina.

Chiamata di procedure

Salto con memorizzazione dell'indirizzo di ritorno:

`bl label` branch and link

salta all'indirizzo di etichetta `label` e salva nel `link register lr` (r14) l'indirizzo di ritorno (cioè l'istruzione successiva a `bl label`).

Rientro dalla procedura:

`mov pc, lr`

Esempio: funzione fattoriale $n!$

$$n! = n \cdot (n - 1) \cdot \dots \cdot 2 \cdot 1$$

```
.text
main:      ...
          bl fattoriale
          ...
fattoriale: ...
          ...
          mov pc, lr
fibonacci: ...
          ...
          ...
```

Passaggio di parametri e risultati

Si usano i registri per il passaggio dei parametri (se pochi). **Convenzione** sull'uso dei registri:

- $r0, \dots, r3$ sono utilizzati per passare argomenti a una procedura.
- $r0, r1$ sono utilizzati per restituire al programma chiamante i valori risultato di una procedura.

Eventuali parametri aggiuntivi devono essere passati **attraverso la memoria**.

Esempio: funzione fattoriale

```
.text
main:    ...
          mov r0, #5
          bl fattoriale
          ...
fattoriale: ...
          movs r1, r0
          beq end
          ...
          mul r0, r4, r0
          mov pc, lr
fibonacci: ...
          ...
```

Interferenza sui registri

Programma principale e procedura agiscono sullo stesso insieme di registri.

Bisogna evitare interferenze improprie:

main:

```
    ...  
    add r4, r4, #1  
    mov r0, #5  
    bl fattoriale
```

```
    ...
```

fattoriale:

```
    ...  
    ...  
    move r4, r0  
    ...  
    mov pc, lr
```

Salvataggio dei registri

- Per convenzione, i registri r4-r14 devono essere **preservati** dalle procedure
- se una procedura vuole utilizzarli
 - li salva in memoria prima del loro utilizzo
 - ne ripristina il valore originale prima di restituire il controllo al programma chiamante.

Dualmente

- i registri r0-r3 sono considerati **modificabili** dalle procedure
- se contengono dati utili, il programma chiamante
 - li salva in memoria prima di chiamare una procedura
 - li ripristina dopo la chiamata.

Esempio: salvataggio di r4-r14

main:

```
    ...  
    add r4, r4, #1  
    mov r0, #5  
    bl fattoriale
```

```
    ...
```

fattoriale:

```
    ...  
    stmfd sp!, {r4-r5}  
    ...  
    move r4, r0  
    ...  
    ldmfd sp!, {r4-r5}  
    mov pc, lr
```

Esempio: salvataggio di r0-r3

```
main:    ...
        add r2, r2, #1
        mov r0, #5
        stmfd sp!, {r2,r3}
        bl fattoriale
        ldmfd sp!, {r2-r3}

        ...

fattoriale: ...
        ...
        move r2, r0
        ...
        mov pc, lr
```

Istruzioni di load e store multipli

`stmfd sp!, {r0, r4-r6, r3}`

- salva in locazioni decrescenti di memoria,
- a partire dall'indirizzo in $sp - 4$ ($r13 - 4$),
- il contenuto dei registri $r0, r4, r5, r6, r3$,
- aggiorna sp alla locazione contenente l'ultimo valore inserito:

$$r1 = r13 - 5*4$$

`ldmfd sp!, {r0, r4-r6, r3}`

ripristina il contenuto di tutti i registri (compreso sp).

Suffissi in load e store multiple

Il comando `stm` è spesso usato per manipolare lo stack.

Il registro `r13` punta alla **cima** dello stack.

Più possibilità:

- `stmia r13!, {...}` **incrementa** da `sp`
- `stmib r13!, {...}` **incrementa** da `sp+4`
- `stmda r13!, {...}` **decrementa** da `sp`
- `stmdb r13!, {...}` **decrementa** da `sp-4`.

Esistono i suffissi corrispondenti: `fd`, `fu`, `ed`, `eu`.

Allocazione spazio di memoria

Ogni procedura necessita di un'area di memoria per

- mantenere le variabili locali
- salvare i registri
- acquisire i parametri e restituire i risultati.

Tutta l'area è allocata in un frame dello stack.

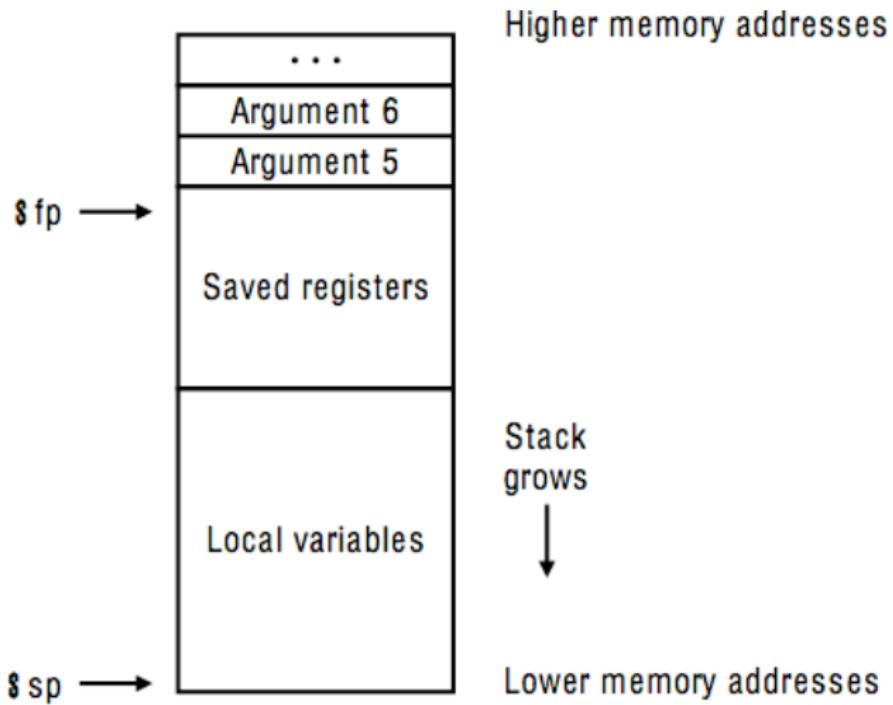
Chiamate innestate generano una pila di frame consecutivi:

- una **chiamata** di procedura alloca un nuovo frame
- un'**uscita** dalla procedura libera l'ultimo frame.

Last in - first out: la procedura chiamata più recentemente è la prima a terminare.

Stack chiamate di procedura

Formato di un frame dello stack:



Uso dello stack

Per convenzione, gestito mediante il registro r13 o sp (**stack pointer**), che punta in ogni momento alla prima parola libera oltre la cima dello stack.

Inizialmente una procedura **salva i registri** nel nuovo frame.

La stessa procedura prima di rientrare svuota il frame **ripristinando le condizioni iniziali**.

Per convenzione lo stack **cresce verso il basso**.

ArmSim inizializza sp in modo che punti a un'area della memoria su cui poter realizzare lo stack.

Chiamata di funzione: fattoriale

```
; calcolo di fattoriale(n) = n(n-1)...
.data
valore: .word 4
.text
main: ldr r4, =valore
      ldr r0, [r4]          @ carica il dato n in r0
      bl fattoriale        @ chiamata di funzione
      str r0, [r4]          @ salva il risultato
      swi 0x11              @ s.o. ferma il programma
.end
```

Soluzione iterativa

fattoriale:

```
        mov r1, #1      @ iniz. contatore
        mov r2, #1      @ iniz. risultato parziale
loop:   cmp r1, r0      @ inizio ciclo
        bge exit
        add r1, r1, #1  @ agg. contatore
        mul r2, r1, r2  @ agg. risul. parz.
        b loop
exit:   mov r0, r2
        mov pc, lr
```

Soluzione ricorsiva

fattoriale:

```
    stmfd sp!, {r4, lr} @ salva registri
    mov r4, r0
    sub r0, r0, #1
    cmp r0, #1           @ se r0 == 1 ...
    beq skip            @ va a svuotare lo stack
    bl fattoriale      @ chiamata ricorsiva
skip:   mul r0, r4, r0  @ accumula risultato
    ldmfd sp!, {r4, lr} @ ripristina registri
    mov pc, lr          @ rientra dalla procedura
```

Uso della memoria

ArmSim prevede il seguente uso della memoria:

- 0x0000 - 0x0FFF: riservata al sistema operativo
- 0x1000 - 0xNNNN : codice del programma (.text), costanti (.data)
- 0xNNNN - 0x5400: stack per chiamate di procedura
- 0x5400 - 0x11400: **heap** per allocazione di strutture dati dinamiche.

Valori modificabili con opportune direttive di assemblaggio.

Il registro r13, sp viene inizializzato a 0x5400.

Il registro r15, pc viene inizializzato a 0x1000.

Procedura per la divisione intera

Realizza la divisione intera tra binari.

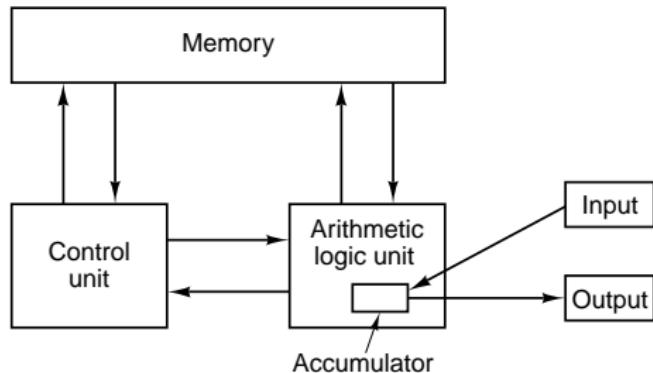
```
divisione:      ; r0 / r1 (interi positivi)
                ; fine: resto in r0, risultato in r1
    mov r3, #31          ; contatore
    mov r2, #0           ; inizializza r2 a 0
loop:   mov r2, r2, lsl #1 ; risultato parziale
        cmp r1, r0, lsr r3 ; se r1 < (r0>>) ...
        suble r0, r0, r1, lsl r3 ; ...sottrai <<r1 a r0
        addle r2, r2, #1   ; aggiorna risultato parziale
        subs r3, r3, #1   ; aggiorna contatore
        bge loop
    mov r1, r2
    mov pc, lr
```

Esercizi

- Scrivere una funzione che sommi tutti gli elementi pari di un vettore.
- Scrivere una procedura che, in un vettore, azzeri tutti i valori negativi.
- Scrivere una procedura che determini se un numero è primo.

Parallelismo: un punto di partenza

Modello di calcolo: macchina di von Neumann



- un'unica operazione in esecuzione
- una sola componente attiva.

Ricerca di nuove architetture con più operazioni in esecuzione allo stesso istante.

Approcci

- Reti neurali
- Data flow machine, Connection machine
- Calcolatori vettoriali
- Architetture bio-ispirate.

Calcolatori vettoriali: usati nel calcolo scientifico.

Supercomputer: costosi, architetture non standard, richiedono la presenza di personale specializzato e quindi sono perlopiù accessibili nei centri di calcolo. Stili di programmazione differenti e spesso dipendenti dall'architettura.

Problema più generale di usare efficientemente le **CPU multicore** a bordo degli attuali PC.

Sviluppo delle architetture parallele

Architetture parallele fisicamente sul mercato dagli anni '90.

- CPU con pipeline, CPU superscalari: eseguono programmi pre-esistenti.
- CPU multi-core, sistemi multiprocessore, cluster di calcolatori.

Soluzione ai limiti insormontabili dell'integrazione delle componenti nei chip.

Richiedono l'utilizzo di programmi per **ambienti concorrenti/distribuiti**.

Presentazione

- Problematiche e aspetti generali del parallelismo
- Classificazione delle architetture parallele
- Esempi di calcolatori paralleli, idee architetturali attualmente in uso.

Motivazioni per il parallelismo

Migliorare le prestazioni in ambiti in cui:

- un algoritmo richiede più potenza di calcolo possibile
- un programma tende a sfruttare tutte le risorse disponibili
- l'aumento delle prestazioni non può più più pesare sull'integrazione nel chip.

Ulteriori vantaggi:

- un'architettura parallela può aumentare la tolleranza ai guasti: ridondanza dei componenti (es.: RAID)
- scalabilità: aumento “proporzionale” delle prestazioni aggiungendo nuove unità di calcolo.

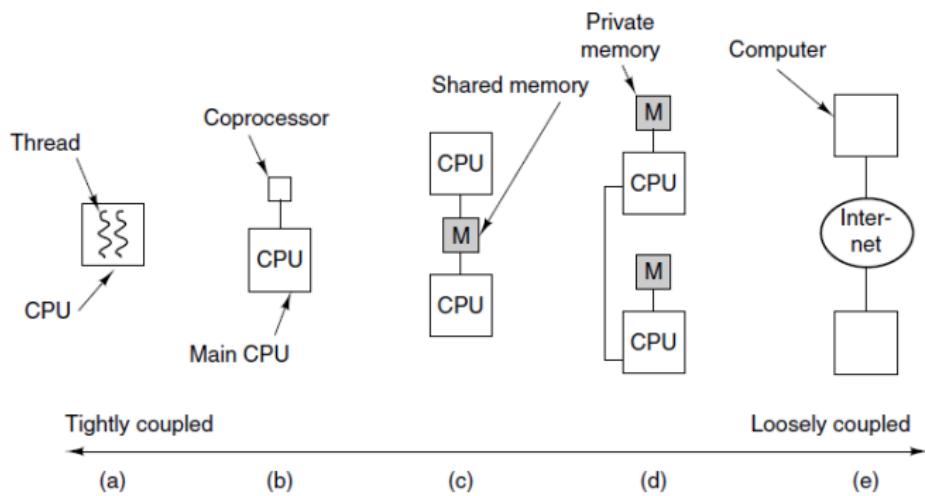
Catalogazione del parallelismo

Granularità: complessità delle operazioni eseguite in parallelo e dei circuiti che le eseguono

- **Fine:** semplici istruzioni, Instruction Level Parallelism (ILP).
Es.: processori con pipeline.
- **Grossa (coarse):** procedure, Process Level Parallelism (PLP).
Es.: multi-threading, multiprocessori, multicompiler.

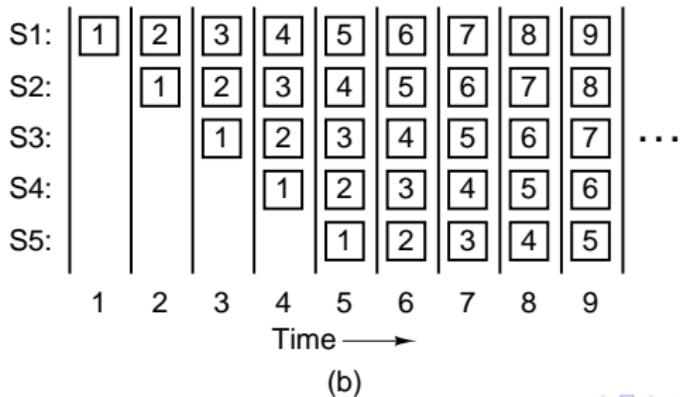
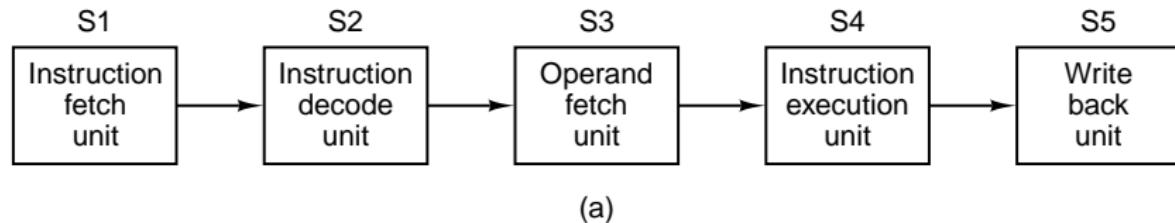
Livello di accoppiamento

- **Forte:** unità connesse, condivisione di dati.
Es.: processori superscalari, multiprocessori.
- **Debole:** unità più indipendenti, scambio di messaggi. Es.: multicomputer.



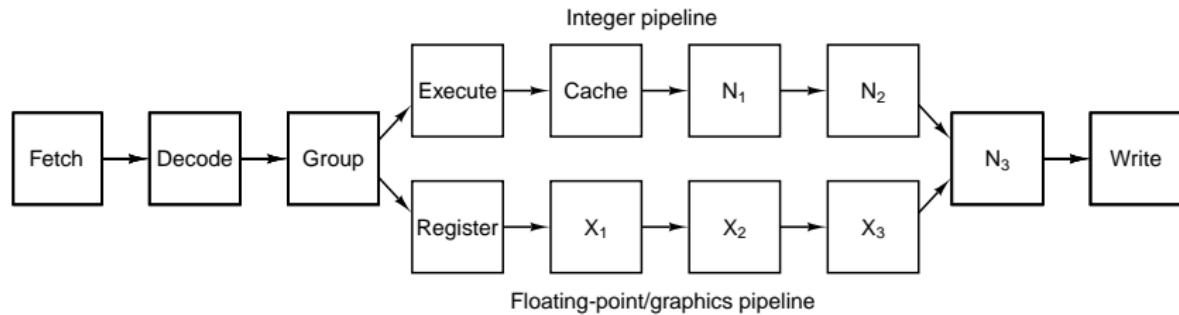
Parallelismo su un singolo chip

Pipeline: esecuzione della micro-istruzione su più stadi



Parallelismo su un singolo chip

Processori superscalari: più micro-istruzioni eseguite in contemporanea



Aumento delle prestazioni di circa **un ordine di grandezza**.

Parallelismo su singolo chip

Pipeline e superscalarità non rendono necessaria la riscrittura di un programma, ma hanno effetti limitati sulle prestazioni.

Altre tecniche per il parallelismo:

- Processori VLIW: Very Long Instruction Word
- On-chip multi-threading
- Single-chip multiprocessor
 - Multiprocessori omogenei
 - Multiprocessori eterogenei.

VLIW: Very Long Instruction Word

Istruzioni macchina adatte al calcolo parallelo.
Sfruttano al meglio le tecnologie esistenti.

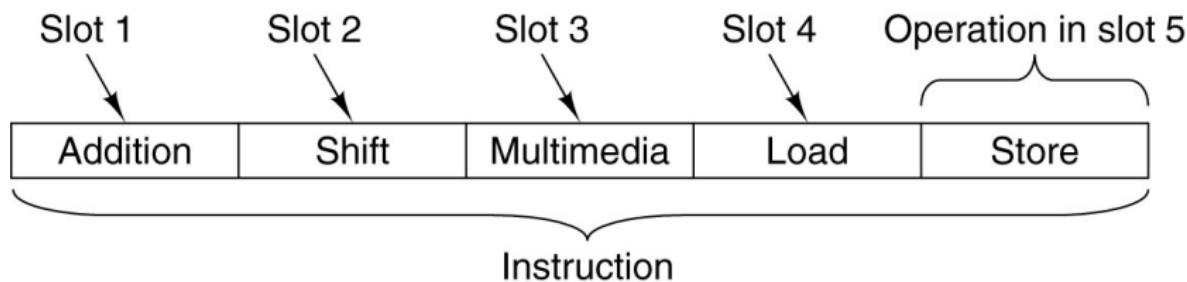
Istruzioni lunghe, ciascuna composta da una sequenza di istruzioni base.

Il rispetto delle dipendenze tra eventi in una istruzione VLIW è garantito a priori, in quanto delegato a compilatore/programmatore.

TriMedia

Processore Philips, evoluzione della filosofia DSP per applicazioni audio-video (DVD player - recorder, camcorder,)

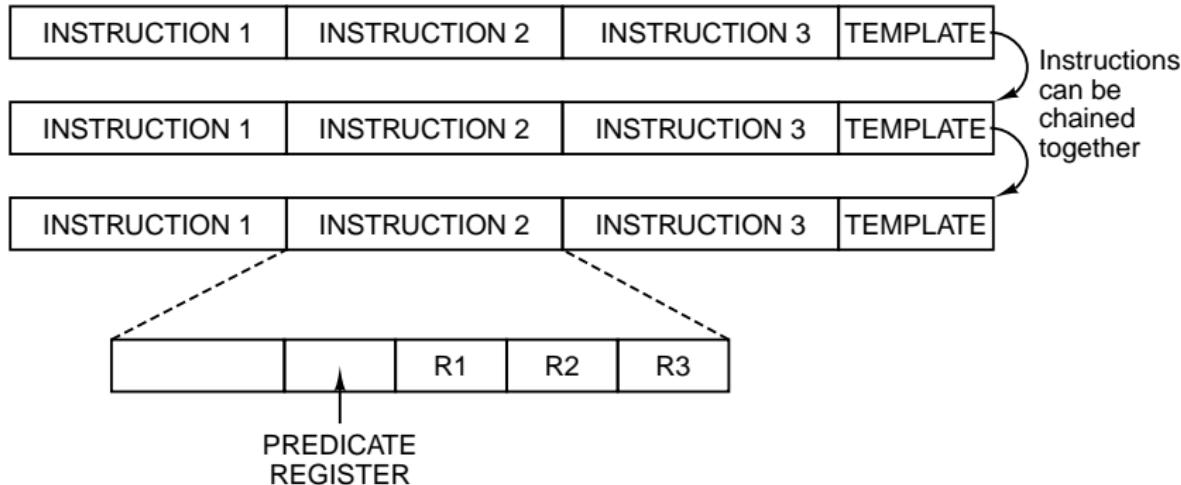
Una singola istruzione è composta da 5-8 sotto-istruzioni: operazioni aritmetiche, load-store, multimediali (vettoriali).



Aritmetica anche a saturazione (elaborazione dei segnali). Micro-operazioni predlicative.

Itanium IA-64

Candidato a sostituire l'Intel IA-32 (Pentium, Core).



Più istruzioni possono essere accodate ed eseguite contemporaneamente, senza controllarne l'indipendenza.

Itanium IA-64

- Istruzioni condizionate, per minimizzare i salti (operazioni predicative).
- Centinaia di registri, per minimizzare gli accessi in memoria e la dipendenza tra istruzioni.
- Caricamenti speculativi, anticipo gli accessi in memoria (a meno di page-fault).

Poco successo commerciale: prestazioni sotto le aspettative, difficile progettare i compilatori.

Il mercato ha preferito x86-64 (o AMD 64): scelta conservativa.

Multi-threading

Permette di sfruttare meglio le capacità di calcolo dei processori superscalari.

Il processore esegue più **thread** (micro-processi **con memoria condivisa**) contemporaneamente.

Utile nel caso un programma rallenti per:

- dipendenze tra istruzioni
- istruzioni che bloccano temporaneamente l'esecuzione (accessi alla memoria principale, alla cache di III livello).

Multi-threading

Può essere utilizzato anche in processori con singola pipeline. Più utile in processori superscalari.

Il processore **commuta** da un thread ad un altro:

- multi-threading **a grana fine**: commutazione su ogni istruzione. Anticipa possibili blocchi ma impone un numero elevato di commutazioni
- multi-threading **a grana grossa**: si eseguono più istruzioni per ogni thread. Riduce le commutazioni senza prevenire i blocchi
- multi-threading **simultaneo**: a grana grossa, ma nel caso di **stallo** commuta sul thread successivo. Massimizza l'impegno delle unità funzionali nella CPU.

Multi-threading

Processore con una singola pipeline:

(a)

A1	A2			A3	A4	A5			A6	A7	A8
----	----	--	--	----	----	----	--	--	----	----	----

(d)

A1	B1	C1	A2	B2	C2	A3	B3	C3	A4	B4	C4
----	----	----	----	----	----	----	----	----	----	----	----

(b)

B1			B2			B3	B4	B5	B6	B7	B8
----	--	--	----	--	--	----	----	----	----	----	----

(c)

C1	C2	C3	C4			C5	C6			C7	C8
----	----	----	----	--	--	----	----	--	--	----	----

(e)

A1	A2		B1		C1	C2	C3	C4	A3	A4	A5
----	----	--	----	--	----	----	----	----	----	----	----

Cycle →

Cycle →

Processore superscalare:

A1	B1	C1	A3	B2	C3	A5	B3	C5	A6	B5	C7
A2		C2	A4		C4		B4	C6	A7	B6	C8

Cycle →

(a)

A1	B1	C1	C3	A3	A5	B2	C5	A6	A8	B3	B5
A2		C2	C4	A4			C6	A7		B4	B6

Cycle →

(b)

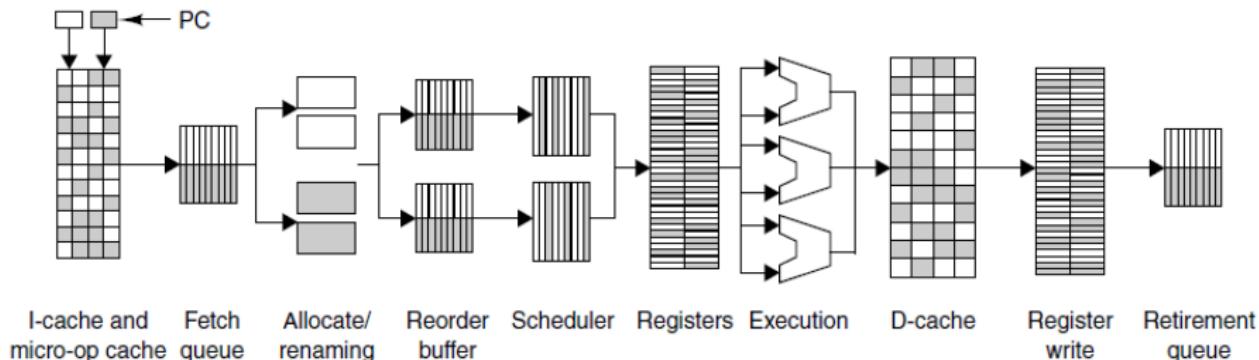
A1	B1	C2	C4	A4	B2	C6	A7	B3	B5	B7	C7
A2	C1	C3	A3	A5	C5	A6	A8	B4	B6	B8	C8

Cycle →

(c)

Multi-threading: ripartizione risorse

- Risorse **condivise**: memoria cache.
- Risorse **partizionate (tempo)**: pipeline.
- Risorse partizionate (spazio): registri.



Multi-threading: condivisione risorse

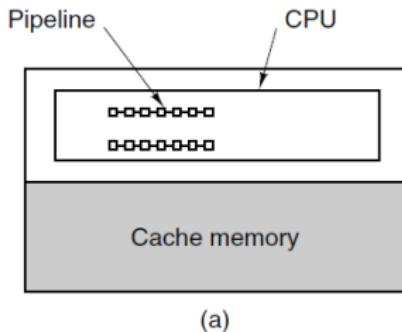
- Condivisione **ripartita**: ogni thread possiede un insieme privato di risorse. Rischio di sottoutilizzo delle risorse.
- Condivisione **totale**. Rischio di **starvation** di uno o più micro-processi.
- Condivisione **a soglia**. La condivisione totale è limitata alle risorse acquisibili da un thread.

Multi-threading usato su molti processori:

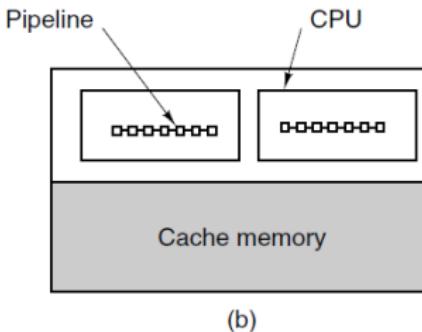
- Core i7: **hyper-threading**, 2 thread/core
- UltraSparc T3: 16 core, 8 thread/core.

Multiprocessori omogenei

Evoluzione del multi-threading: un core per ogni thread. Totale separazione delle risorse tra i thread. Più CPU su un singolo chip. Condivisione della sola memoria.



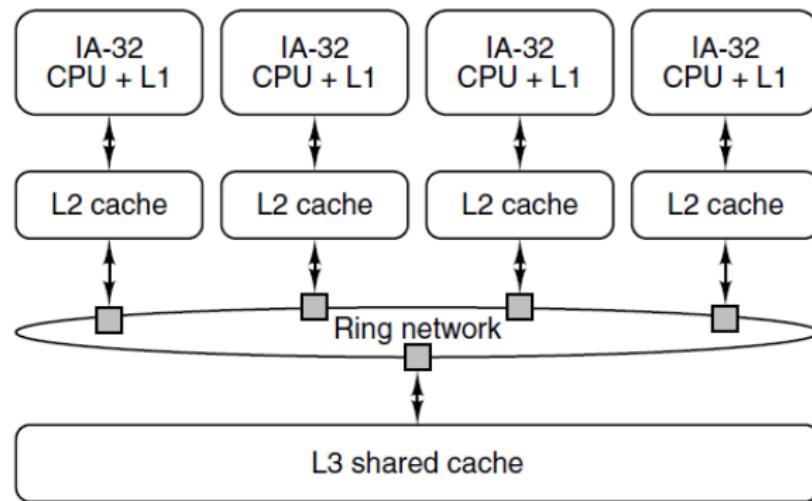
(a)



(b)

La tecnologia attuale permette di inserire più processori su di un chip.

Intel Core i7



Comunicazione attraverso una rete ad anello
(ispirata al modello **token ring**): garantisce semplicità
e consistenza dei dati a fronte di un moderato
spreco di tempo.

Co-processori

Parallelismo ottenuto delegando alcuni compiti a processori ausiliari.

- DMA (Direct Memory Access): complesso controllore che si fa carico della gestione I/O.
- Scheda video: elaborazione di immagini in tempo reale.
- Co-processori multimediali: codifica/decodifica audio-video (MPEG, MP3), post-elaborazione del segnale.
- Scheda di rete: controllo degli errori, instradamento dei messaggi.
- Cripto-processori: criptazione e decrittazione delle informazioni.

Multiprocessori eterogenei

System-on-Chip (SoC), diverse CPU su singolo chip

- core principale di controllo
- co-processori specializzati
- memoria
- bus di interconnessione su diversi standard:
CoreConnect (IBM), ispirato a PCI
AMBA (per ARM)
VCI (Virtual Component Interconnect).

Chip progettati combinando preesistenti progetti di singole componenti (core).

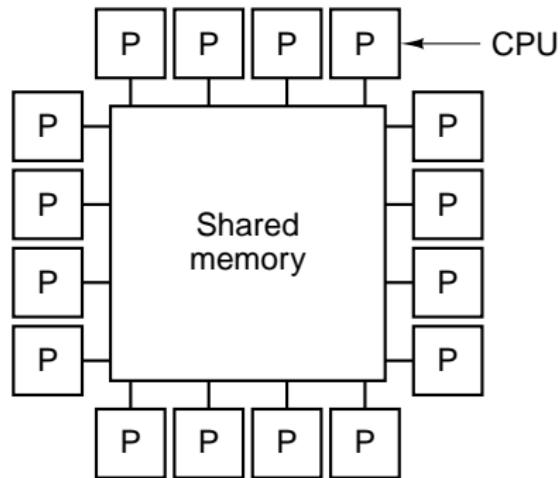
Parallelismo su più chip

Più processori su più circuiti integrati.

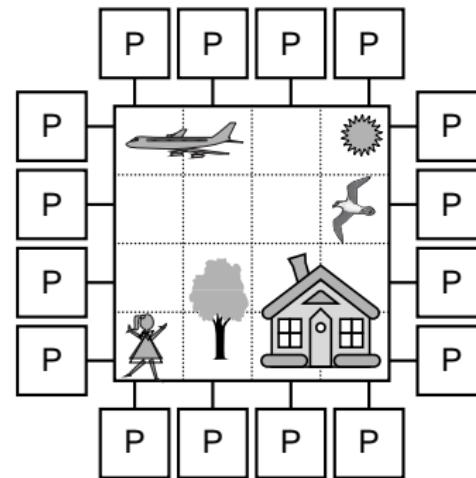
- Scalabilità dell'aumento di prestazioni.
- Necessità di programmi ad-hoc (o di un adatto S.O.).
- Approcci: multiprocessori e multicompiler.

Multiprocessori

Sistemi a memoria condivisa.



(a)



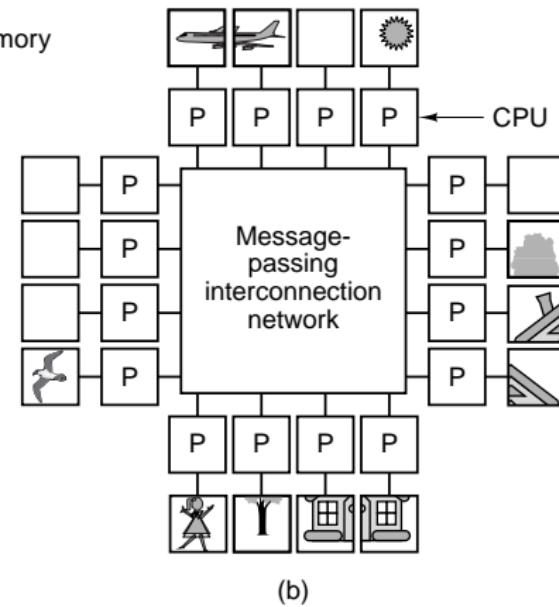
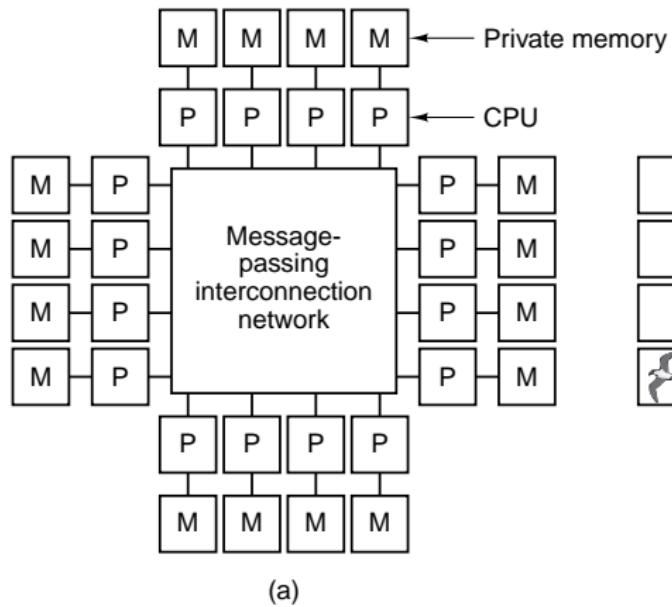
(b)

Multiprocessori: idea

- Le CPU condividono lo stesso spazio di memoria.
- Comunicazione tra CPU attraverso l'accesso alla memoria condivisa: LOAD, STORE.
- Non è necessario ripartire i dati tra le CPU.
- L'accesso alla memoria limita il grado di parallelismo.
- Es.: PC multi-processore; server.

Multicomputer

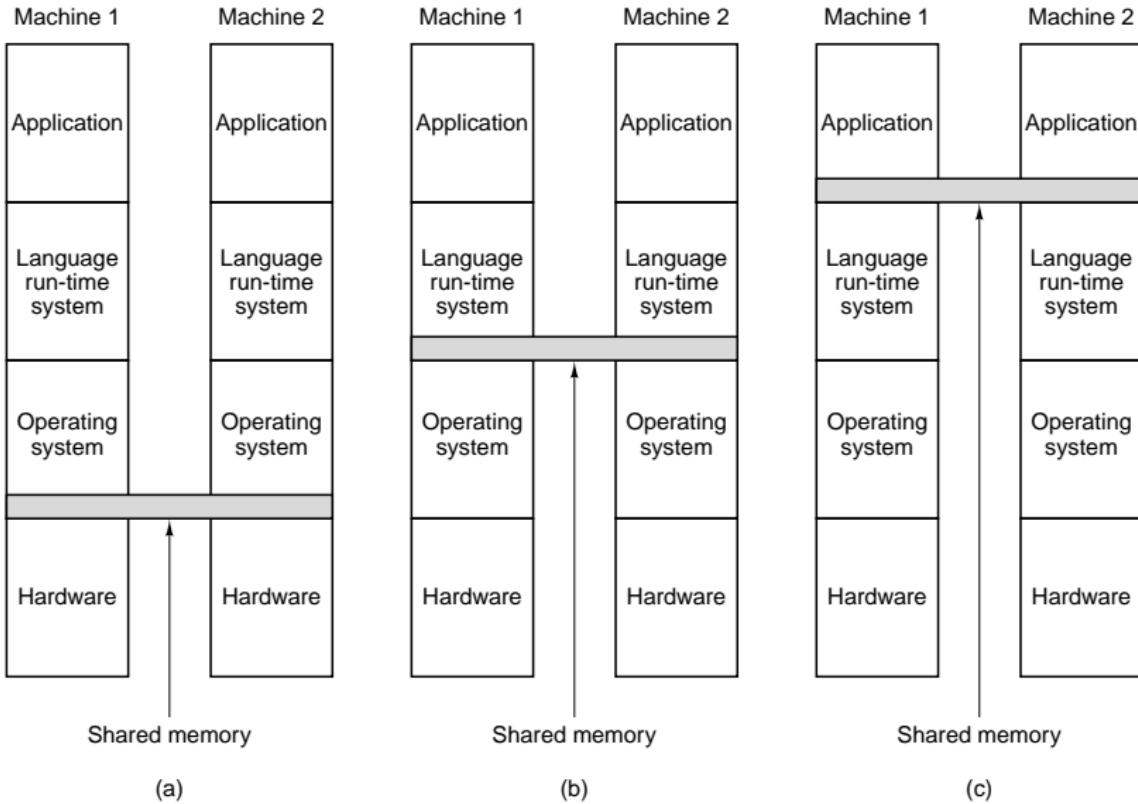
Sistemi a memoria distribuita:



Multicomputer: idea

- Ogni CPU ha un proprio spazio di memoria.
- Comunicazione attraverso **scambio di messaggi**: SEND, RECEIVE.
- Più semplice integrare le CPU.
- Numero dei processori limitato dalla capacità della rete di interconnessione.
- Paradigma a scambio di messaggi più difficile da programmare.

Rappresentazione memoria condivisa



Rappresentazione memoria condivisa

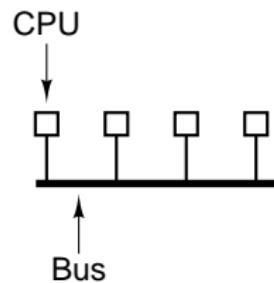
- 1 Memoria principale condivisa.
- 2 Memoria virtuale paginata condivisa.
Distributed Shared Memory (DSM). Una pagina può trovarsi:
 - nella memoria locale della CPU
 - nella memoria di massa
 - nella memoria locale di un'altra CPU.
- 3 Memoria condivisa a livello applicazione
 - il sistema operativo è sollevato dal rappresentare la memoria alle CPU
 - la responsabilità della memoria è delegata al programma utente.
 -

Es.: Linda, ORCA.

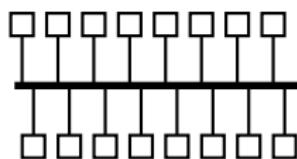
Scalabilità

Una stessa idea architetturale può essere implementata con un numero variabile di unità di calcolo.

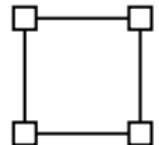
Un'architettura è scalabile se funziona correttamente con molte unità di calcolo.



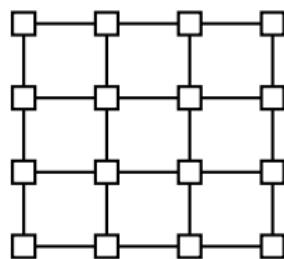
(a)



(b)



(c)



(d)

Scalabilità: prestazioni

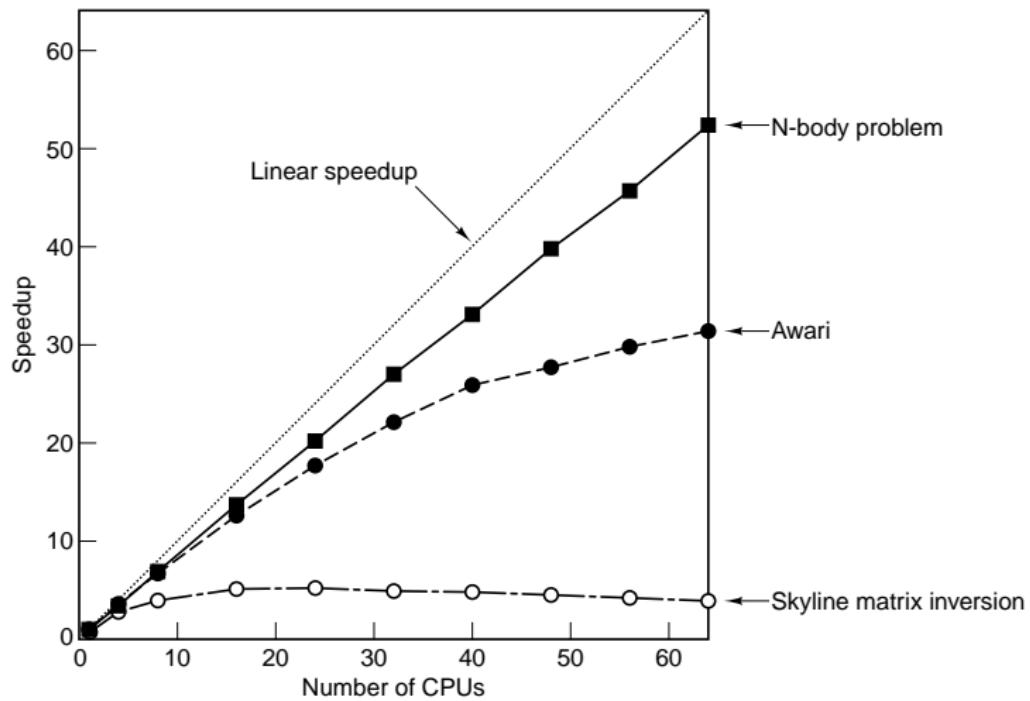
- Le unità di calcolo devono sincronizzarsi tra loro: **overhead** di attività.
- Memoria, bus e altre risorse condivise creano potenziali conflitti tra le unità di calcolo.

Conseguenze:

- è impossibile che n CPU svolgano un lavoro n volte più velocemente. Legge di **Amdahl**:
rapporto incremento di velocità = $\frac{n}{1+(n-1)(1-f)}$
con $0 \leq f \leq 1$ frazione di **tempo parallelizzabile**
- per certi algoritmi un aumento delle unità di calcolo porta a un peggioramento delle prestazioni.

Scalabilità: prestazioni

Incremento misurato su algoritmi di benchmark:



Multrprocessor/Multicomputer: componenti

- **Unità di calcolo:** CPU/processori standard.
- **Unità di memoria:** memorie cache locali alle CPU, memoria principale divisa in banchi per gestire più richieste contemporanee.
- **Rete di interconnessione**
 - nei multiprocessori: collegano processori e moduli di memoria
 - nei multicomputer: collegano le unità di calcolo.

Memoria

- Diverse unità operanti in parallelo.
- Complessità dovuta alla replicazione dei dati:
 - ogni core possiede una cache locale,
 - cache di alto livello comune a più core,
 - memoria principale comune a più processori, eventualmente divisa in banchi.
- Tempi di accesso dipendenti dalla distanza della memoria dal processore.
- Per motivi di efficienza sono possibili **semantiche della memoria** in cui sono ammesse letture e scritture fuori ordine.

Semantica della memoria

Compromesso tra efficienza e consistenza.

Consistenza

- **stretta**: rappresentazione ordinata della memoria
- **sequenziale**: rappresentazione ordinata della sequenza di scritture in memoria
- **di processore**: rappresentazione ordinata della sequenza di scritture in memoria di ogni CPU
- **debole**: rappresentazione ordinata solo dopo sincronizzazioni regolari della memoria
- **dopo il rilascio**: rappresentazione ordinata solo dopo che un evento di sincronizzazione è invocato da una CPU.

Reti di interconnessione

Equivalenza logica coi protocolli per i bus, ma a un livello di complessità maggiore.

Componente fondamentale dei calcolatori paralleli.

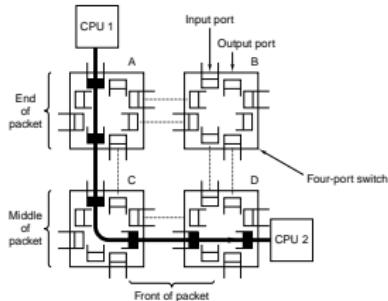
Composte da

- link (connessioni - cavi - bus) paralleli o seriali
- switch (instradatori - commutatori)
- interfacce (processori - rete - memorie).

Interconnessione: switch

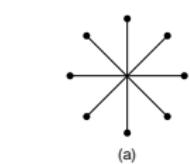
Tipi di istradamento:

- **commutazione di circuito** (circuit switching): messaggio instradato su un circuito cablato temporaneo
- **commutazione di pacchetto** (packet switching): messaggio suddiviso in pacchetti.

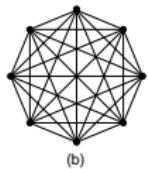


Routing: determinazione del percorso del circuito (statico) o dei pacchetti (dinamico).

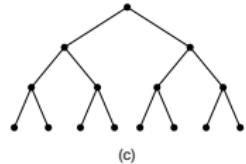
Topologie di rete



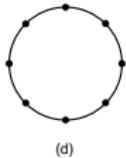
(a)



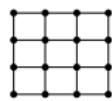
(b)



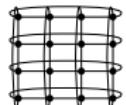
(c)



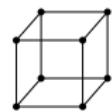
(d)



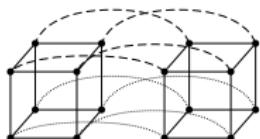
(e)



(f)



(g)



(h)

Topologie di rete

- La rete è dimensionata al fine di
 - massimizzare l'ampiezza di banda (bidirezionale)
 - evitare colli di bottiglia
 - minimizzare le distanze (in numero di archi): problema delle latenze.
- Configurazioni: **stella**, completamente interconnessa, albero, anello, **griglia**, doppio toroide, **iper cubo**.
- Geometria fissa o variabile.

Tassonomia dei computer paralleli

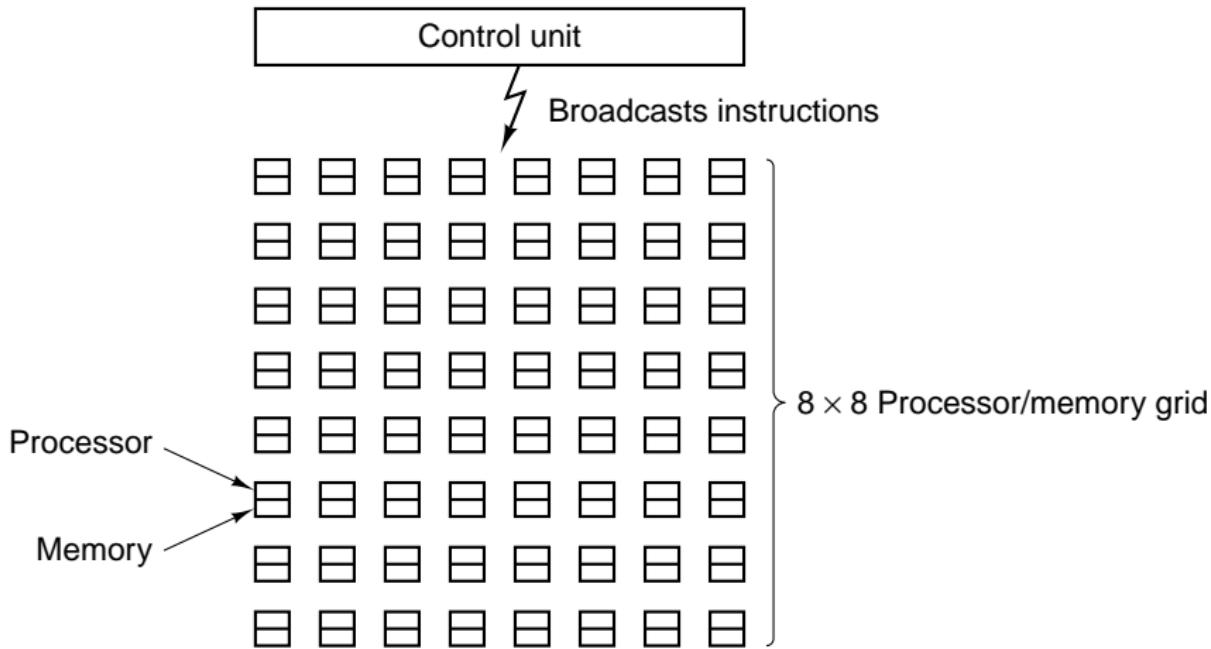
Classificazione di Flynn (1972)

- **SISD** Single Instruction Single Data (macchina di von Neumann)
- **SIMD** Single Instruction Multiple Data (computer vettoriali)
- **MIMD** Multiple Instruction Multiple Data (multiprocessori e multicompiler)
- **MISD** Multiple Instruction Single Data (nessun esempio).

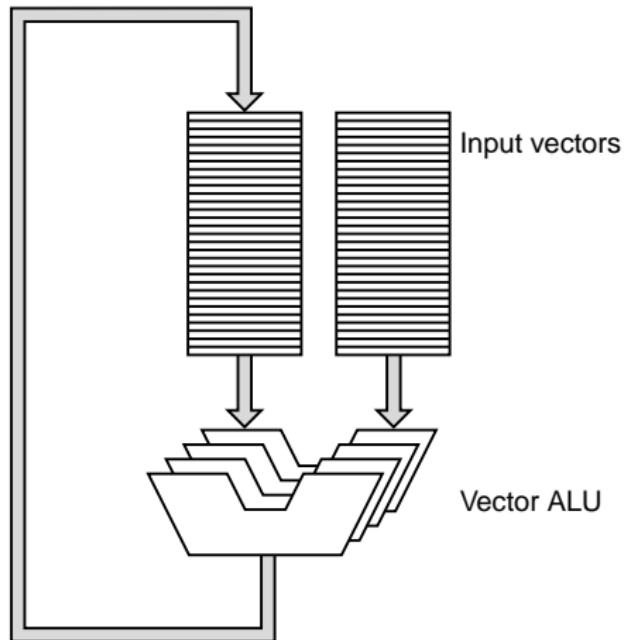
SIMD: array processor

Più processori controllati da una unità.

Ancestore: ILLIAC IV



SIMD: array processing



ALU e registri operano su vettori (array).

Supercomputer anni '70-'80: Cray-1 e successivi.

SIMD: istruzioni

I processori attuali ereditano l'esperienza SIMD potendo eseguire istruzioni vettoriali.

- Istruzioni multimediali: lunghezza fissa.
- Istruzioni vettoriali: insiemi di dati di lunghezza variabile, parametriche.

ISA x86:

- MMX (MultiMedia eXtension)
- SSE (Streaming SIMD Extension), ... , SSE4
- AVX (Advance Vector eXtension)

Utilizzano registri molto lunghi, centinaia di bit, contenenti vettori di dati (byte, half-word, word) su cui operare con istruzioni vettoriali.

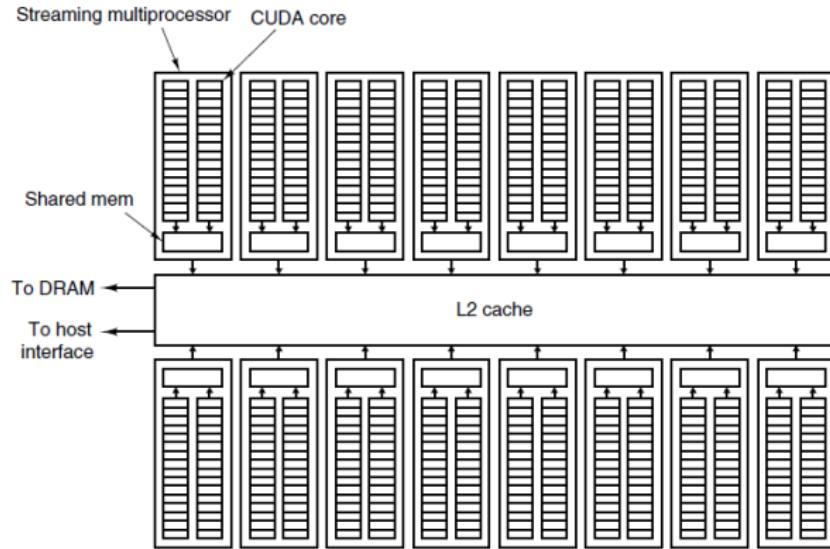
SIMD: Graphics Processing Unit

Alcune schede grafiche ereditano l'architettura dei computer vettoriali.

Nvidia Fermi:

- la stessa operazione è eseguita da identici core (CUDA) clusterizzati entro Streaming Multiprocessor (SM): $16 \text{ SM} \times 32 \text{ SIMD} \times 2 \text{ thread} = 16 \text{ SM} \times 64 \text{ CUDA} = 512 \text{ core CUDA}$
- ogni processore SIMD esegue 2 thread
- la struttura della memoria è complessa
 - ogni core ha una memoria privata
 - ogni SM ha una memoria locale
 - gli SM condividono una memoria cache (L2).

GPU: Nvidia Fermi



Gerarchia di memoria: locale al singolo core,
condivisa tra gruppi di core, memoria comune.

GPGPU (General Purpose GPU)

Si utilizzano le GPU per calcolare algoritmi generici (non grafici):

- si sfrutta l'elevato numero di core
- solo alcuni algoritmi altamente parallelizzabili sono adatti a questo tipo di architettura
- linguaggi di programmazione dedicati:
 - CUDA (Nvidia, proprietario)
 - OpenGL (non proprietario, eseguibile su più schede grafiche).

Calcolatori MIMD

Multiprocessor:

- **UMA** Uniform Memory Access
- **NUMA** Non Uniform Memory Access.

Multicomputer:

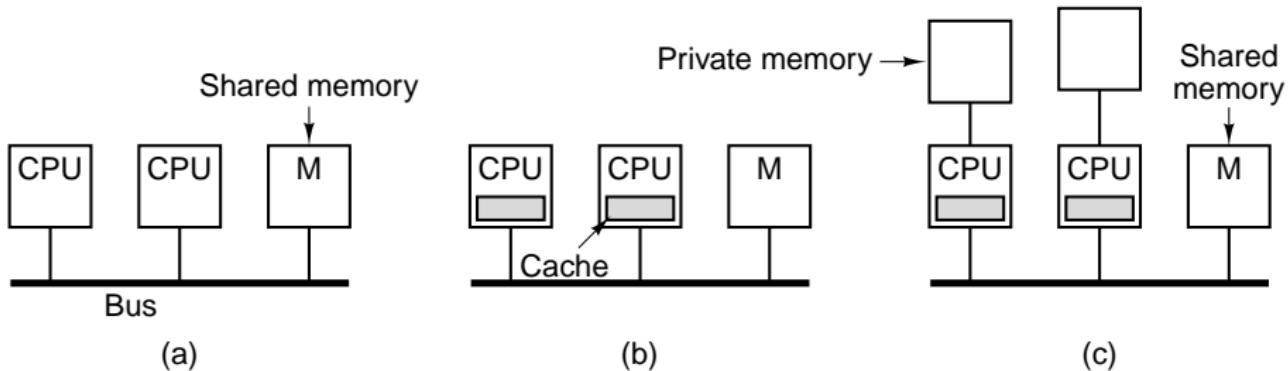
- **MPP** Massive Parallel Processor
- **COW** Cluster Of Workstation
- **NOW** Network Of Workstation
- **Grid Computing.**

Multiprocessori UMA: SMP

Tutti i processori hanno stessi tempi di accesso alla memoria.

Symmetric Multi Processor (SMP): tutti i processori hanno la stessa visione della memoria e dei dispositivi I/O.

SMP: singolo bus



- L'implementazione più semplice di un sistema multiprocessore.
- Bus limita il numero di CPU utilizzabili (16).
- Cache di grosse dimensioni per limitare l'accesso al bus.
- Necessità di mantenere la coerenza della cache.

SMP: cache snooping

Le cache “spia” il traffico sul bus per **garantire** la coerenza.

Protocollo **write-through**.

- **Read Hit**: lettura dalla cache, nessuna azione del protocollo.
- **Read Miss**: lettura dalla memoria condivisa, nessuna azione del protocollo.
- **Write Miss**: scrittura in memoria condivisa, nessuna azione del protocollo.
- **Write Hit**: scrittura in cache, il protocollo invalida la stessa linea delle cache locali alle altre CPU.

Semplice ma poco efficiente: troppe scritture in memoria condivisa.

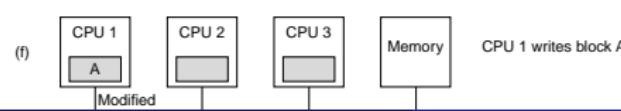
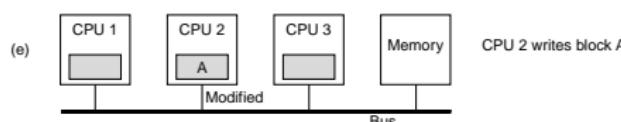
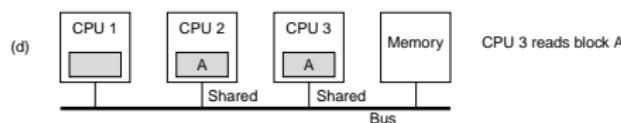
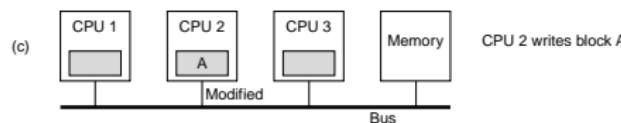
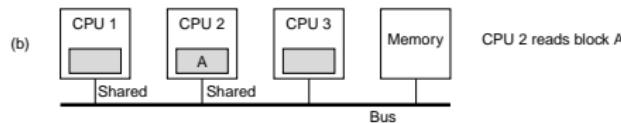
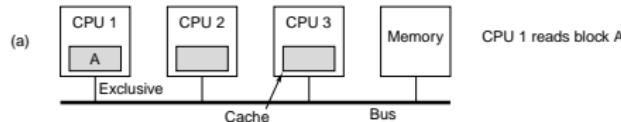
SMP: protocollo MESI

Protocollo **write-back**.

Ogni linea di cache viene marcata non in due, ma quattro possibili modi.

- **Shared**: linea condivisa, memoria aggiornata
- **Exclusive**: linea non condivisa, memoria aggiornata
- **Modified**: linea non condivisa, memoria non aggiornata
- **Invalid**: linea non valida.

MESI: esempio



Laboratorio di architettura degli elaboratori

ASSEMBLY PER ARM Lezione 9

- CONTATTI
- Prof. F. Fontana (federico.fontana@uniud.it)
- Y. De Pra (depra.yuri@spes.uniud.it)

Esercizio 9.1

Scrivere del codice assembly ARM che, nella sezione .data, definisca un vettore con 4 valori uguali a 0, e nella sezione .text contenga un programma che sostituisce il vettore con il vettore contenente i primi 4 numeri naturali.

Esercizio 9.2

Scrivere del codice assembly ARM che, nella sezione .data, definisca un vettore di 3 numeri interi, e nella sezione .text contenga un programma che sostituisce ogni valore nel vettore con il suo quadruplo.

Esercizio 9.3

Scrivere del codice assembly ARM che, nella sezione .data, definisca un vettore di 3 numeri interi, e nella sezione .text contenga un programma che sostituisce il vettore con una sua rotazione in cui ogni elemento viene spostato in avanti di una posizione.

Esercizio 9.4

Scrivere del codice assembly ARM che, nella sezione .data, inserisca in memoria due numeri interi positivi m ed n e nella sezione .text contenga un programma che, attraverso una serie di sottrazioni, calcoli quoziente e resto della divisione m/n. I valori quoziente e resto devono essere inseriti nei registri r0, r1.

Es.: manipolazione di stringhe

Sostituzione di caratteri in una stringa

```
.data
stringa:.asciiz "stringa da manipolare"
a:      .ascii "a"
o:      .ascii "o"
.text
main:   ldr r0, =stringa  @ inizializza registri
        ldr r1, =a
        ldrb r1, [r1]
        ldr r2, =o
        ldrb r2, [r2]
        bl scambia      @ chiamata procedura
        swi 0x11
.end
```

Es.: manipolazione di stringhe

Procedura per la sostituzione dei caratteri

scambia:

```
ldr r3, [r0], #1
    cmp r3, #0          @ test fine stringa
    beq exit
    cmp r3, r1          @ confronto caratteri
    streqb r2, [r0, #-1] @ modifica stringa
    cmp r3, r2          @ confronto caratteri
    streqb r1, [r0, #-1] @ modifica stringa
    b scambia           @ ciclo
exit:   mov pc, lr
```

Esercizi

- 1 Scrivere una procedura che determini se il processore funziona in modalità big-endian o little-endian.
- 2 Scrivere una procedura che determini se la stringa di lunghezza come da `r1`, contenuta in memoria all'indirizzo base `r0`, è palindroma.
- 3 Scrivere una procedura che determini se il contenuto del registro `r0` è una sequenza binaria palindroma.

Chiamate al sistema operativo

Software interrupt: `swi 0xCODE`. Es.: `swi 0x12`.

Chiama una procedura eseguita dal sistema operativo identificata dall'argomento `0xCODE`.

Diversa della chiamata di procedura standard per garantire i **meccanismi di protezione** propri degli interrupt.

Il processore che esegue un programma assembly prevede due modalità di funzionamento

- **system**: esegue qualunque operazione
- **user**: alcune operazioni sono vietate.

L'accesso alla modalità `system` è possibile solo alle procedure invocate mediante la `swi`.

ArmSim: software interrupt

ArmSim simula una serie di procedure di sistema.

Per fare questo i *plugin* devono essere abilitati:

File → Preferences → Plugins → SWIInstructions

operazione	cod.	argomento
print_char	0x00	r0 char
print_string	0x02	r0 string address
exit	0x11	
allocate	0x12	r0 size ⇒ address

ArmSim: accesso a file

operazione	cod.	argomento
open	0x66	r0 name \Rightarrow handle, r1 mode
close	0x68	r0 handle
write str	0x69	r0 handle, r1 string
read str	0x6a	r0 handle, r1 string, r2 size
write int	0x6b	r0 handle, r1 integer
read int	0x6c	r0 handle \Rightarrow integer

Operano su file di testo (carattere).

Software interrupt for files

File aperto in modalità

- `input` (mode 0), solo lettura
- `output` (mode 1), lettura e scrittura – sovrascrive il contenuto
- `append` (mode 2), lettura e scrittura – accoda il contenuto

`open` restituisce un handle, identifica il file.

Il file di uscita standard (`stdout: standard output`) ha handle = 1.

ArmSim non implementa lo `stdin: standard input`.

`read str`: legge al più `r2` caratteri di una riga di testo, e passa alla riga successiva.

Altre direttive assembly

Costanti definite con la direttiva `.equ`

```
.equ PrintInt, 0x6b  
...  
swi PrintInt
```

Direttiva da inserire nella parte `.data`.

Nomi simbolici per i registri definiti da `.req`:

```
lo .req r0  
hi .req r1  
adds lo, lo, r2  
adcs hi, r3, lo
```

I nomi simbolici migliorano la leggibilità.

Istruzione iniziale

Denotata con l'etichetta `_start`. Es.:

```
_start:    mov r0, ...
```

forza l'inizio dell'esecuzione da una qualsiasi istruzione.

Assembler dipendente, non universale.

Funziona con ArmSim, non necessariamente con altri simulatori.

Strutture dati: matrici

Le matrici vengono **linearizzate**, ovvero ricondotte a strutture di tipo **array**.

Es.:

$$\begin{pmatrix} 1 & 3 & 5 & 7 \\ 2 & 5 & 8 & 1 \\ 3 & 7 & 1 & 5 \end{pmatrix}$$

Per riga:

1 3 5 7 2 5 8 1 3 7 1 5

Per colonna:

1 2 3 3 5 7 5 8 1 7 1 5

Matrici

Data una matrice M con m righe ed n colonne, indichiamo gli elementi con $M[i, j]$, $i \in [0, m - 1]$, $j \in [0, n - 1]$.

Nella linearizzazione di M , l'elemento $M[i, j]$ occupa la posizione

- $i \times n + j$ (memorizzazione per righe)
- $j \times m + i$ (memorizzazione per colonne).

Nella memorizzazione per righe, se l'elemento $M[i, j]$ occupa l'indirizzo x allora

- l'elemento $M[i + d, j]$ occupa l'indirizzo $x + n \times d$
- l'elemento $M[i, j + d]$ occupa l'indirizzo $x + d$.

Esercizi

- 1) Scrivere una procedura che, ricevuti in r0 e r1 l'indirizzo base e lunghezza di un array di interi (word), calcoli in r0 la somma degli elementi.
- 2) Scrivere una procedura che ricevuti in r0 l'indirizzo base di una matrice, in r1 il numero di righe, in r2 il numero di colonne, e infine in r3 l'indirizzo base di un array, inserisca nello stesso array le somme degli elementi delle righe della matrice. Ripetere l'esercizio stavolta sommando le colonne.
- 3) Scrivere una procedura che calcola la somma degli elementi pari di tutte le diagonali di una matrice quadrata.