Fondamenti di Architettura degli Elaboratori

Francesco Giacomini
INFN – CNAF

Bologna, 21 dicembre 2015

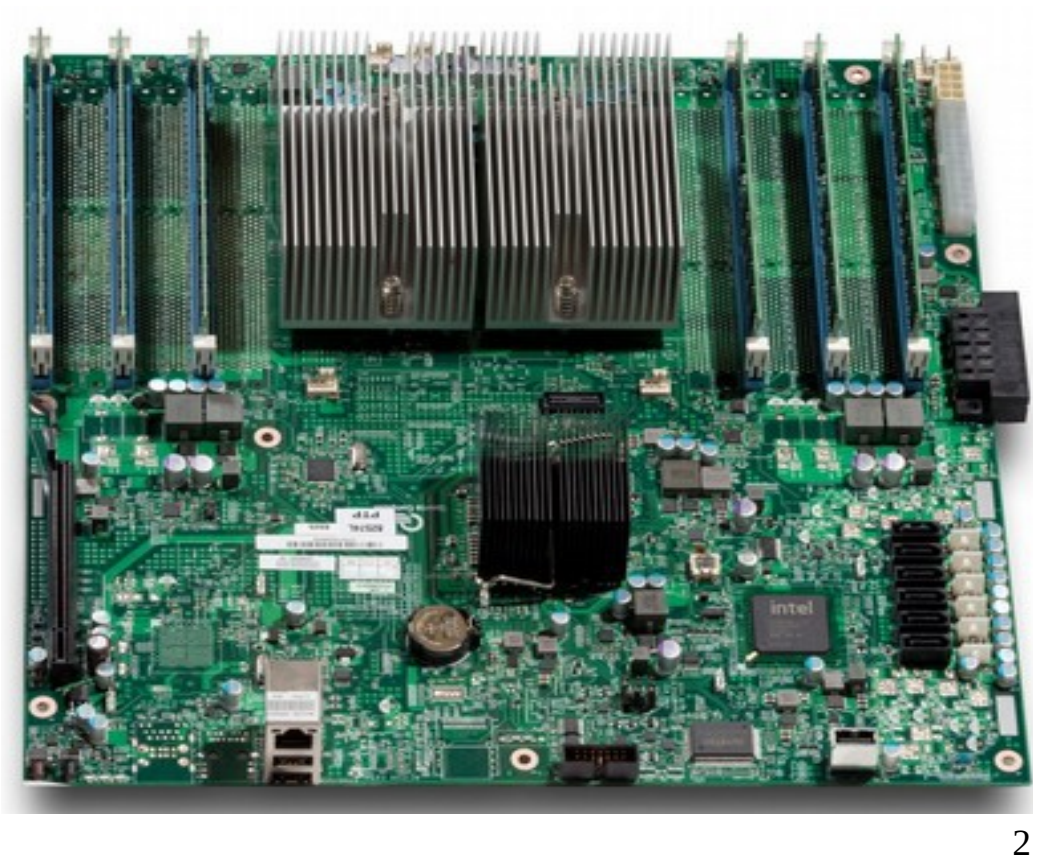
tratto da: http://www.unife.it/scienze/informatica/insegnamenti/architettura-elaboratori

CC BY

Obiettivo della lezione

- Comprendere gli elementi essenziali dell'interfaccia offerta da un sistema di calcolo per l'esecuzione, corretta ed efficiente, di un programma scritto in un linguaggio di alto livello

```
void swap(int* l, int* r)
{
  int t = *l;
  *l = *r;
  *r = t;
}
```



Argomenti trattati

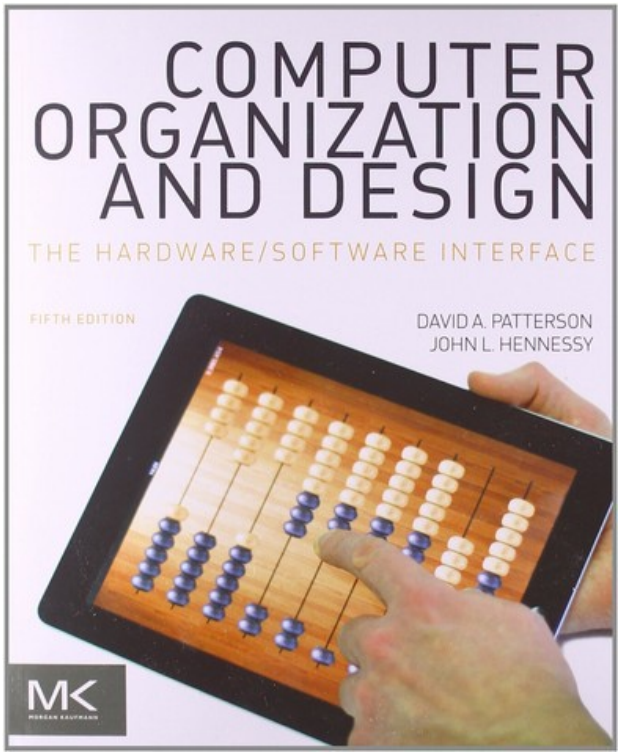
- Rappresentazione binaria dell'informazione
- Il processore
  - Clock
  - Registri
  - Datapath
  - Pipeline
- La memoria
  - Spazio indirizzabile di un processo
  - Gerarchia di memoria, cache

Per approfondire

Computer Organization and Design
The Hardware/Software Interface

D. Patterson, J. Hennessy

5th Edition, Elsevier, 2013

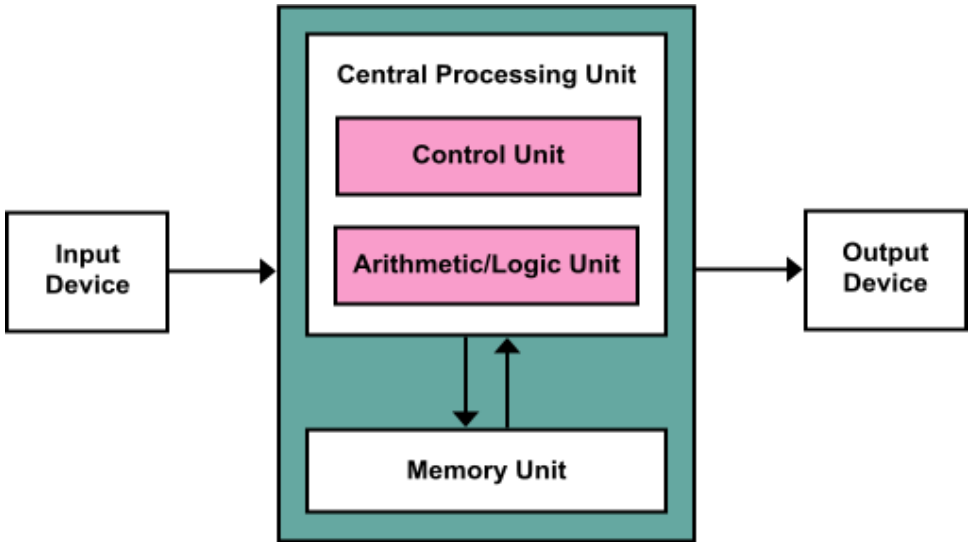


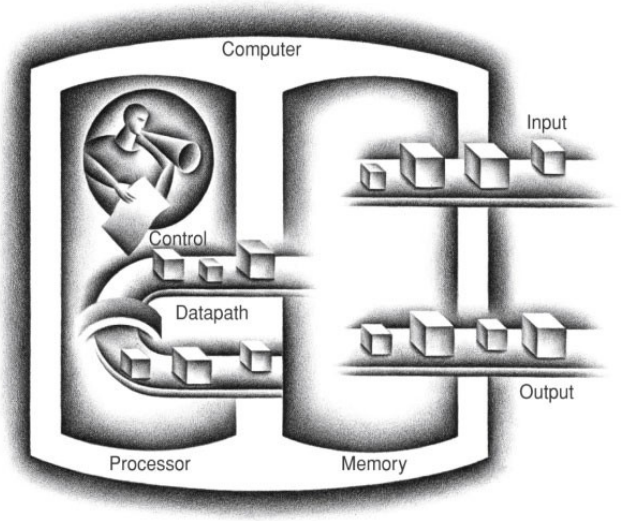
Struttura e Progetto dei Calcolatori

4ª edizione italiana, condotta sulla 5ª edizione americana

Zanichelli, 2015

Architettura di von Neumann





- Concetto di *stored program*
  - le istruzioni di un programma sono mantenute in memoria, così come i dati

Rappresentazione binaria

- In un sistema di calcolo digitale qualsiasi informazione viene mantenuta e manipolata in forma binaria
  - la rappresentazione consiste in una sequenza di `0` e `1`
  - ogni elemento della sequenza viene chiamato bit
- Alcune sequenze hanno nomi specifici
  - un byte è una sequenza di 8 bit
  - una parola (word) indica una sequenza di alcuni byte, ad es. 4 byte (32 bit)

Operazioni logiche

- Su una sequenza di bit sono definite alcune operazioni logiche di base
  - shift left

    ```
    00111010 << 3 =
    11010000
    ```

  - shift right

    ```
    00111010 >> 3 =
    00000111
    ```

  - bit-by-bit AND

    ```
    0011 1000 &
    0101 0101 =

    0001 0000
    ```

  - bit-by-bit OR

    ```
    0011 1000 |
    0101 0101 =

    0111 1101
    ```

  - bit-by-bit NOT

    ```
    ~0011 1000 =

     1100 0111
    ```

Sistema numerico posizionale

- Un sistema di numerazione si dice posizionale se i simboli (cifre) usati per scrivere i numeri assumono valori diversi a seconda della posizione che occupano nella notazione

  `3564 = 3 migliaia, 5 centinaia, 6 decine, 4 unità`

  $3564 = 3 \cdot 1000 + 5 \cdot 100 + 6 \cdot 10 + 4 \cdot 1 =$

  $= 3 \cdot 10^3 + 5 \cdot 10^2 + 6 \cdot 10^1 + 4 \cdot 10^0$

- Nell'esempio il sistema è anche decimale
  - nella notazione si usano dieci cifre, da `0` a `9`
  - potenze di dieci
  - la base dieci non è speciale, ci è semplicemente più familiare
    - ovvero, le tabelline le conosciamo in base dieci

Sistema numerico posizionale

- In generale (per numeri interi):
  - $(c_n c_{n-1} \ldots c_1 c_0)_p = c_n \cdot P^n + c_{n-1} \cdot P^{n-1} + \ldots + c_1 \cdot P^1 + c_0 \cdot P^0$
- Esempi
  - $(456)_{otto} = (4 \cdot 8^2 + 5 \cdot 8^1 + 6 \cdot 8^0)_{dieci} = (302)_{dieci}$
  - $(302)_{dieci} = (3 \cdot 12^2 + 0 \cdot 12^1 + 2 \cdot 12^0)_{otto}$
    $= (3 \cdot 144 + 2)_{otto} = (454 + 2)_{otto} = (456)_{otto}$
  - $(1011)_{due} = (1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{dieci} = (11)_{dieci}$
  - $(1011)_{cinque} = (1 \cdot 5^3 + 0 \cdot 5^2 + 1 \cdot 5^1 + 1 \cdot 5^0)_{dieci} = (131)_{dieci}$
  - $(FA9)_{sedici} = (15 \cdot 16^2 + 10 \cdot 16^1 + 9 \cdot 16^0)_{dieci} = (4009)_{dieci}$

    $A \rightarrow (10)_{dieci}$ $B \rightarrow (11)_{dieci}$
    $C \rightarrow (12)_{dieci}$ $D \rightarrow (13)_{dieci}$
    $E \rightarrow (14)_{dieci}$ $F \rightarrow (15)_{dieci}$

Da base P a base B

- $(c)_P = (c_n c_{n-1} \ldots c_1 c_0)_P = (b_m b_{m-1} \ldots b_1 b_0)_B$

  $= b_m \cdot B^m + b_{m-1} \cdot B^{m-1} + \ldots + b_1 \cdot B + b_0$

- Se divido $(c)_P$ per B e prendo il resto ottengo b_0

  $= B \cdot \underbrace{(b_m \cdot B^{m-1} + b_{m-1} \cdot B^{m-2} + \ldots + b_1)}_{q_0} + b_0$

- Se divido q_0 per B e prendo il resto ottengo b_1 e q_1
- Continuo fino a quando il quoziente non è zero
- Esempi (con P = dieci)

```
131 / 5 = 26 resto 1 ↑        131 / 16 = 8 resto 3 ↑
 26 / 5 =  5 resto 1 |         8 / 16 = 0 resto 8 |
  5 / 5 =  1 resto 0 |
  1 / 5 =  0 resto 1 |
```

$(131)_{dieci} = (1011)_{cinque} = (83)_{sedici}$

Numerazione binaria

- Base due, due simboli (cifre) disponibili: `0` e `1`
- Esempi

```
11 / 2 = 5 resto 1
 5 / 2 = 2 resto 1
 2 / 2 = 1 resto 0
 1 / 2 = 0 resto 1
```

```
32 / 2 = 16 resto 0
16 / 2 =  8 resto 0
 8 / 2 =  4 resto 0
 4 / 2 =  2 resto 0
 2 / 2 =  1 resto 0
 1 / 2 =  0 resto 1
```

$(11)_{dieci} = (1011)_{due}$ $(32)_{dieci} = (2^5)_{dieci} = (100000)_{due}$

- Spesso è comodo rappresentare i numeri in base 2^p, con p tipicamente 3 (base 8) o 4 (base 16)
  - si raggruppano le cifre binarie in gruppi di p

$(111\ 011)_{due} = (73)_{otto} = 073\ (0110\ 1111)_{due} = (6F)_{sedici} = 0x6F$

Rappresentazione dei numeri interi

- In un sistema di calcolo un numero ha una rappresentazione finita
- I numeri interi sono tipicamente rappresentati con un numero di cifre binarie (bit) che è una potenza di 2
  - in C/C++ corrispondono ai tipi `char`, `short`, `int`, ... con lunghezze che vanno tipicamente da 8 a 32 o 64 bit
  - con segno (`signed`) e senza segno (`unsigned`)
  - anche un puntatore T* si può entro certi limiti considerare come un intero, di lunghezza 32 o 64 bit

Numeri senza segno

- Per i numeri naturali (interi **senza segno**) la rappresentazione tipica corrisponde a una sequenza di bit corrispondente alla notazione in base 2

```
0000 0000 =   0
0000 0001 =   1
0000 0010 =   2
0000 0011 =   3
...
1111 1110 = 254
1111 1111 = 255 = 2⁸-1
```

Most Significant Bit (MSB)

Least Significant Bit (LSB)

- Con n bit si possono rappresentare 2^n numeri, da 0 a 2^n-1

Numeri con segno

- Per i numeri interi **con segno** si sono inventate diverse rappresenzioni. Quella universalmente usata ora è "**complemento a 2**"
  - complemento a potenza di 2
  - il bit più significativo (MSB) rappresenta il segno
    - 0 positivo, 1 negativo
  - gli altri bit rappresentano il modulo

  secondo il seguente schema (per n = 4)

```
0111 =  7 = 2³-1
0110 =  6
0101 =  5
0100 =  4
0011 =  3
0010 =  2
0001 =  1
0000 =  0

1111 = -1       = -8 + 7
1110 = -2       = -8 + 6
1101 = -3       = -8 + 5
1100 = -4       = -8 + 4
1011 = -5       = -8 + 3
1010 = -6       = -8 + 2
1001 = -7       = -8 + 1
1000 = -8 = -2³ = -8 + 0
```

Opposto di un numero in C2

- $x + \sim x + 1 = 0 \pmod{2^n} \rightarrow -x = \sim x + 1$
- $\sim x$ significa il valore negato (bit-by-bit NOT)
- per calcolare l'opposto di un numero in C2, si nega la sequenza di bit corrispondente al numero e si aggiunge 1

  `0011 (+3) + 1100 + 1 = 10000 → 1101 (-3) = 1100 + 1`

  `1101 (-3) + 0010 + 1 = 10000 → 0011 (+3) = 0010 + 1`

Range dei numeri con segno

- Con n bit si possono rappresentare 2^n numeri interi, da -2^{n-1} a $2^{n-1}-1$
  - n = 8 → $[-2^7, 2^7-1]$ = [-128, 127]
  - n = 16 → $[-2^{15}, 2^{15}-1]$ = [-32768, 32767]
  - n = 32 → $[-2^{31}, 2^{31}-1]$ = [-2147483648, 2147483647]
- Attenzione ad eseguire operazioni che coinvolgano contemporaneamente numeri con segno e numeri senza segno

```
int i = -1;
unsigned u = 0;
std::cout << std::boolalpha << (i < u) << '\n'; // false
std::cout << i + u << '\n';   // 4294967295 (0xFFFFFFFF)
```

Estensione del segno in C2

- Un numero in C2 a `n` bit si può trasformare in un numero in C2 a `m` bit (con `m > n`) estendendo il bit del segno negli `m - n` bit più significativi

  `0011 (+3 a 4 bit) → 0000 0011 (+3 a 8 bit)`

  `1101 (-3 a 4 bit) → 1111 1101 (-3 a 8 bit)`

Rappresentazione di caratteri

- Oltre ai numeri è necessario rappresentare anche caratteri
- Esistono molte codifiche, tra cui la più diffusa è l'ASCII
  - 127 caratteri
  - ogni carattere è codificato con 7 bit
    → sta in 8 bit (byte)

(sono mostrati solo i caratteri stampabili)

```
   2 3 4 5 6 7
 ------------
0:   0 @ P ` p
1: ! 1 A Q a q
2: " 2 B R b r
3: # 3 C S c s
4: $ 4 D T d t
5: % 5 E U e u
6: & 6 F V f v
7: ´ 7 G W g w
8: ( 8 H X h x
9: ) 9 I Y i y
A: * : J Z j z
B: + ; K [ k {
C: , < L \ l |
D: - = M ] m }
E: . > N ^ n ~
F: / ? O _ o DEL
```

Floating point

- Rappresentazione di numeri non interi
- Consideriamo solo la parte “decimale”
  - sappiamo già gestire la parte intera

$$\ldots . c_{-1} c_{-2} \ldots c_{-n} \ldots = \ldots + c_{-1} \cdot B^{-1} + c_{-2} \cdot B^{-2} + \ldots + c_{-n} \cdot B^{-n} + \ldots$$

Conversione di base

- Esempi

$(0.456)_{otto} = (4 \cdot 8^{-1} + 5 \cdot 8^{-2} + 6 \cdot 8^{-3})_{dieci} = (0.58984375)_{dieci}$

$(0.456)_{otto} = (4 \cdot 8^{-1} + 5 \cdot 8^{-2} + 6 \cdot 8^{-3})_{sedici} = (0.97)_{sedici}$

$(0.1101)_{due} = (1 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-4})_{dieci} = (0.8125)_{dieci}$

$(0.8125)_{dieci} = (8 \cdot A^{-1} + 1 \cdot A^{-2} + 2 \cdot A^{-3} + 5 \cdot A^{-4})_{sedici} = (0.D)_{sedici}$

$(0.75)_{dieci} = (111 \cdot 1010^{-1} + 101 \cdot 1010^{-2})_{due} = (0.11)_{due}$

Conversione di base

- $.f = (0.b_{-1}\ldots b_{-m}\ldots)_B$

  $= b_{-1}\cdot B^{-1} + b_{-2}\cdot B^{-2} + b_{-3}\cdot B^{-3} + \ldots$

- Se moltiplico $.f$ per B e prendo la parte intera ottengo b_{-1}

  $.f\cdot B = b_{-1} + b_{-2}\cdot B^{-1} + b_{-3}\cdot B^{-2} + \ldots = (b_{-1}.b_{-2}b_{-3}\ldots)_B$

- Se moltiplico per B la parte decimale di $.f\cdot B$ e prendo la parte intera ottengo b_{-2} e così via

- In generale la procedura non termina e ci si ferma quando si raggiunge il numero di cifre voluto

Conversione di base

- Esempi

$$0.8125 \cdot 2 = 1.625 \rightarrow 1$$
$$0.625 \cdot 2 = 1.25 \rightarrow 1$$
$$0.25 \cdot 2 = 0.5 \rightarrow 0$$
$$0.5 \cdot 2 = 1 \rightarrow 1$$

$$0.8125 \cdot 16 = 13 \rightarrow D$$

$$(0.8125)_{10} = (0.1101)_2 = (0.D)_{16}$$

$$0.2 \cdot 2 = 0.4 \rightarrow 0$$
$$0.4 \cdot 2 = 0.8 \rightarrow 0$$
$$0.8 \cdot 2 = 1.6 \rightarrow 1$$
$$0.6 \cdot 2 = 1.2 \rightarrow 1$$
$$0.2 \cdot 2 = 0.4 \ldots$$

$$0.2 \cdot 5 = 1 \rightarrow 1$$

$$(0.2)_{10} = (0.\overline{0011})_2 = (0.1)_5$$

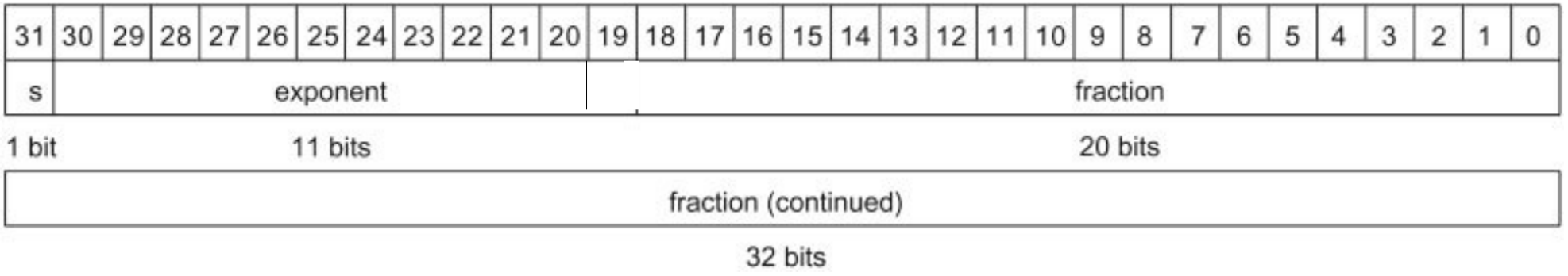
Rappresentazione floating point

- Necessariamente un'approssimazione dei numeri reali
  - tipi `float` e `double` in C/C++
- Notazione scientifica
  - $-4.8744 \cdot 10^{-9}$
  - $-1343.8383 \cdot 10^{5}$
  - $+0.0002$
- Notazione scientifica normalizzata
  - $\pm d.dddd \cdot 10^{e}$ (decimale)
  - $\pm \mathbf{1}.bbbb \cdot 2^{e}$ (binaria)
- Definita nello standard IEEE 754
  - singola (32 bit) e doppia (64 bit) precisione

Rappresentazione floating point

32 bit

64 bit



Rappresentazione floating point

$$v = (-1)^s \cdot (1 + \text{fraction}) \cdot 2^{(\text{exponent} - \text{bias})}$$

- s è il sign bit (`1` → numero negativo, `0` → non negativo)
- Il bit prima della virgola è implicito a 1
  - 1 + fraction = *significand*
  - `1.0 ≤ |`significand`| < 2.0`
- L'esponente è in notazione *biased*: valore + bias
  - `00...00` è il valore più piccolo, *bias* rappresenta lo `0`, `11...11` è il valore più grande
  - SP: bias = `127 (0111 1111)`; DP: bias = `1023 (011 1111 1111)`
- La rappresentazione consente di confrontare i floating point come se fossero interi in complemento a 2

Floating point: esempio

- Rappresentare il numero $(-0.75)_{10} = (-0.11)_2$

  $(-0.11)_2 = (-1)^1 \cdot 1.1 \cdot 2^{-1}$
- S = 1
- Fraction = 10000...0000
- Exponent = -1 + bias
  - SP: $-1 + 127 = 126 = (01111110)_2$
  - DP: $-1 + 1023 = 1022 = (01111111110)_2$
- SP: 1 01111110 100000...00000000000
- DP: 1 01111111110 100000...00000000000

Floating point: esempio

- Che numero è rappresentato dal SP float?

  `1 10000001 01000...00`
- `S = 1`
- `Fraction = 01000...00`
- `Exponent = 10000001 = (129)`$_{10}$

$$v = (-1)^s \cdot (1 + \text{fraction}) \cdot 2^{(\text{exponent} - \text{bias})}$$
$$= (-1)^1 \cdot (1 + 0.01)_2 \cdot 2^{(129 - 127)}$$
$$= -1.25 \cdot 2^2$$
$$= -5 \text{ (come FP)}$$

Riepilogo codifica floating point

Single precision		Double precision		Object represented
Exponent	Fraction	Exponent	Fraction	
0	0	0	0	0
0	Nonzero	0	Nonzero	± denormalized number
1–254	Anything	1–2046	Anything	± floating-point number
255	0	2047	0	± infinity
255	Nonzero	2047	Nonzero	NaN (Not a Number)

Intervallo singola precisione

- Esponenti `00000000` e `11111111` sono riservati
- Valore più piccolo (in valore assoluto)
  - Esponente: `00000001`

    → esponente reale = `1 - 127 = -126`
  - Frazione: `000…00` → significand = `1.0`
  - $\pm 1.0 \cdot 2^{-126} \approx \pm 1.2 \cdot 10^{-38}$ $\quad (38 \approx 126 \cdot \log_{10}(2) \approx 126 \cdot 0.3)$
  - Sotto è underflow (denormal a parte)
- Valore più grande
  - Esponente: `11111110`

    → esponente reale = `254 - 127 = +127`
  - Frazione: `111…11` → significand ≈ `2.0`
  - $\pm 2.0 \cdot 2^{+127} \approx \pm 3.4 \cdot 10^{+38}$
  - Sopra è overflow

Intervallo doppia precisione

- Esponenti 00000000000 e 11111111111 sono riservati
- Valore più piccolo
  - Esponente: 00000000001

    → esponente reale = 1 – 1023 = –1022
  - Frazione: 000…00 → significand = 1.0
  - $\pm 1.0 \cdot 2^{-1022} \approx \pm 2.2 \cdot 10^{-308}$
  - Sotto è underflow (denormal a parte)
- Valore più grande
  - Esponente: 11111111110

    → esponente reale = 2046 – 1023 = +1023
  - Frazione: 111…11 → significand ≈ 2.0
  - $\pm 2.0 \cdot 2^{+1023} \approx \pm 1.8 \cdot 10^{+308}$
  - Sopra è overflow

Rappresentazione istruzioni

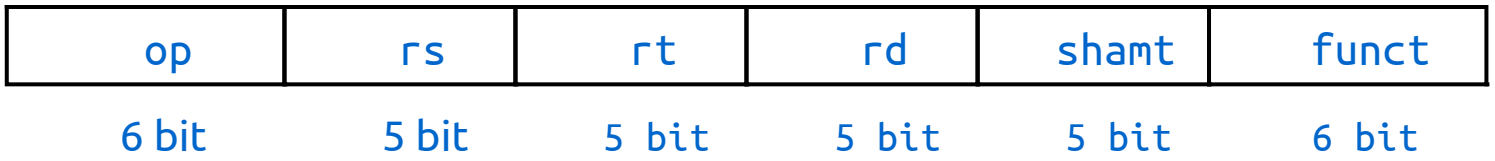
- Anche il programma sta in memoria e le istruzioni che lo compongono sono rappresentate in forma binaria

```
(gdb) x/4 0x02000000
0x2000000:  0xa0100000  0x29008004  0x81c52000  0x01000000
(gdb) disassemble 0x02000000 0x2000010
Dump of assembler code from 0x2000000 to 0x2000010:
0x2000000:  mov  %g0, %l0
0x2000004:  sethi  %hi(0x2001000), %l4
0x2000008:  jmp  %l4
0x200000c:  nop
(gdb) x/4 0x02009ab8
0x2009ab8:  0x00000001  0x00000000  0x40340000  0x00000000
(gdb) disass 0x02009ab8 0x02009ac8
Dump of assembler code from 0x2009ab8 to 0x2009ac8:
0x2009ab8:  unimp  0x1
0x2009abc:  unimp  0
0x2009ac0:  call  0x2d09ac0
0x2009ac4:  unimp  0
End of assembler dump.
(gdb) x 0x2d09ac0
0x2d09ac0:  Cannot access memory at address 0x2d09ac0.
(gdb)
```

Instruction Set Architecture

- Nel corso degli anni sono state definite molte architetture, molte delle quali ancora in uso
  - i386, x86_64, SPARC, MIPS, ARM, VAX, Alpha, PowerPC, …
- Un'ISA definisce l'interfaccia principale tra l'HW e il SW
  - istruzioni e loro significato
  - registri disponibili
  - convenzioni per la chiamata di funzioni
  - interrupt
  - layout in memoria
  - …

MIPS - Formato R



- Campi
  - op: codice dell'operazione (opcode) – sempre a 0 per R
  - rs: numero del primo registro sorgente
  - rt: numero del secondo registro sorgente
  - rd: numero del registro destinazione
  - shamt: numero di bit dello shift
  - funct: codice della funzione (estende l'opcode)

R[rd] = R[rs] op/funct R[rt]

R[rd] = R[rt] << shamt

Formato R - esempio

op	rs	rt	rd	shamt	funct
6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

```
add $t0, $s0, $s1
```

R-format	16	17	8	0	add

000000	10000	10001	01000	00000	100000

0000 0010 0001 0001 0100 0000 0010 0000 = 0x02114020

Indicatori di performance

```
$ perf stat -d ./a.out

 Performance counter stats for './a.out':

      1992.419970 task-clock (msec)         #    0.999 CPUs utilized
              215 context-switches          #    0.108 K/sec
                5 cpu-migrations            #    0.003 K/sec
              368 page-faults               #    0.185 K/sec
    4,139,391,573 cycles                    #    2.078 GHz                      [44.54%]
    2,207,522,429 stalled-cycles-frontend   #   53.33% frontend cycles idle     [44.67%]
    2,182,353,973 stalled-cycles-backend    #   52.72% backend  cycles idle     [44.79%]
    2,302,995,011 instructions              #    0.56  insns per cycle
                                            #    0.96  stalled cycles per insn [55.85%]
      657,124,588 branches                  #  329.812 M/sec                    [55.85%]
      155,694,678 branch-misses             #   23.69% of all branches          [55.67%]
      329,767,307 L1-dcache-loads           #  165.511 M/sec                    [55.45%]
       20,723,322 L1-dcache-load-misses     #    6.28% of all L1-dcache hits   [55.25%]
          200,896 LLC-loads                 #    0.101 M/sec                    [44.15%]
   <not supported> LLC-load-misses:HG

      1.994174760 seconds time elapsed
```

Performance

- Come si definisce il concetto di “performance”?
  - Tempo di esecuzione di un programma
    - Wall-clock time
      - tiene conto anche di I/O e delle attese dovute alla condivisione delle risorse con altri programmi
    - **CPU time**
      - tiene conto solo del tempo in cui il programma usa la CPU
      - user time + system time
  - *throughput*
    - numero di task/transazioni eseguiti per unità di tempo
  - …

performance = 1 / CPU time

Cosa influenza il CPU time

- Algoritmo e qualità di implementazione
- Linguaggio di programmazione
- Compilatore
- Instruction Set Architecture
- Processore
- Memoria

CPU time

$$\text{CPU time} = \#\text{istruzioni} \times \text{CPI} \times \text{Clock cycle}$$

- #istruzioni: numero di istruzioni nel programma
- CPI: numero medio di cicli di clock per istruzione
- Clock cycle: durata di un ciclo di clock

- Per ridurre il tempo di CPU si può intervenire su ognuno dei fattori
  - I fattori non sono indipendenti!

Logic design

- L'elettronica all'interno dei calcolatori moderni è digitale
  - basata su due livelli validi di tensione elettrica
    - livelli intermedi sono transitori, bisogna attendere la stabilizzazione della tensione prima di leggere un valore valido
  - motivo per cui si usa la rappresentazione binaria per le entità da manipolare: numeri, lettere, istruzioni, ecc.
    - alto – basso
    - vero – falso
    - `1` – `0`
- La manipolazione di tali valori binari (bit) è basata sull'algebra di Boole

Algebra booleana

- Algebra definita su un insieme di due valori {0, 1}, con le seguenti operazioni:
  - “addizione” – OR

    A + B = 1 se almeno uno tra A e B è uguale a 1
  - “moltiplicazione” – AND

    $A \cdot B = 1$ solo se entrambi A e B sono uguali a 1
  - complemento – NOT

    $\overline{A} = 1$ se A = 0; $\overline{A} = 0$ se A = 1
- Corrispondenza con operazioni tra insiemi (unione, intersezione, complemento)

Algebra booleana

- Proprietà dell'algebra booleana
  - identità

    $A + 0 = A$

    $A \cdot 1 = A$
  - 0 e 1 elementi assorbenti

    $A + 1 = 1$

    $A \cdot 0 = 0$
  - inversione

    $A + \overline{A} = 1$

    $A \cdot \overline{A} = 0$
  - commutativa

    $A + B = B + A$

    $A \cdot B = B \cdot A$
  - associativa

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

    $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
  - distributiva

    $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

    $A + (B \cdot C) = (A + B) \cdot (A + C)$
  - De Morgan

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

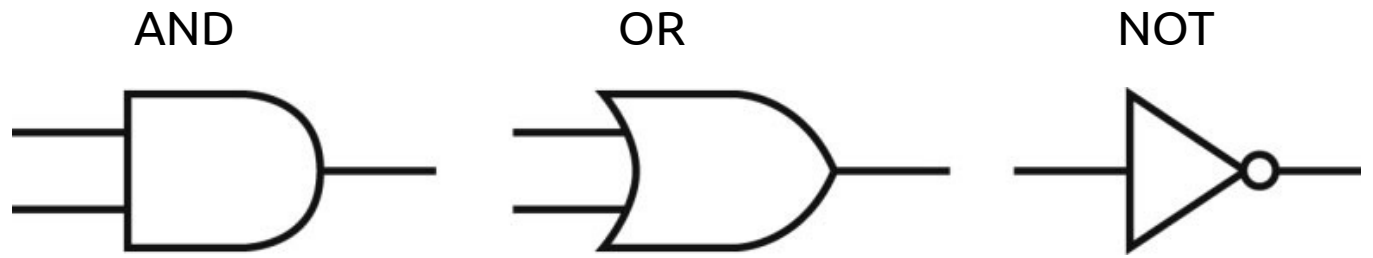
    $\overline{A \cdot B} = \overline{A} + \overline{B}$

Espressioni logiche

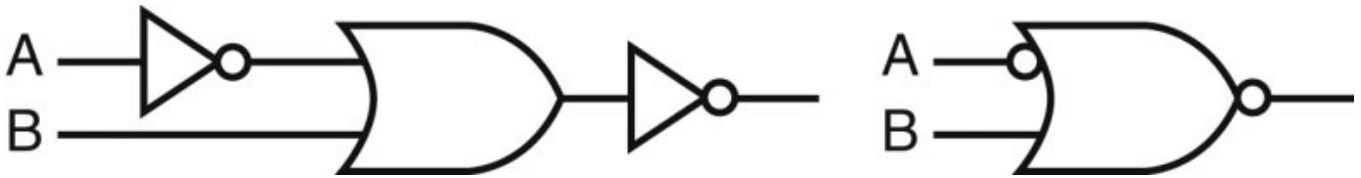
- Un'espressione che coinvolge variabili logiche e operatori logici
  - $D = A + B + C$
  - $E = ((A \cdot B) + (A \cdot C) + (B \cdot C)) \cdot (\overline{A \cdot B \cdot C})$
  - $F = A \cdot B \cdot C$
  - D, E e F sono funzioni logiche di A, B e C.

Porte logiche

- Le operazioni fondamentali dell'algebra di Boole sono implementate con dei circuiti logici elementari (porte logiche), che sono astrazioni di dispositivi elettronici veri e propri basati tipicamente su transistor



- Una funzione logica si può esprimere con un blocco logico fatto di AND, OR e NOT
  - $\overline{\overline{A} + B}$



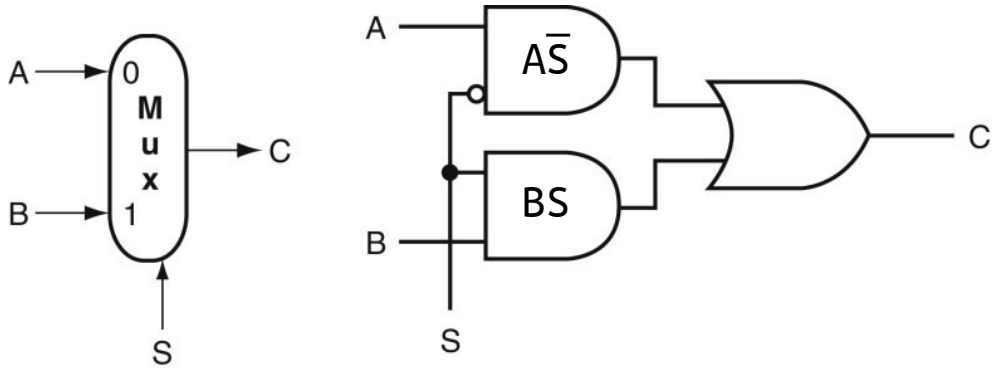
Versione semplificata

Blocchi logici

- A partire dalle porte logiche elementari si possono costruire circuiti (blocchi) logici arbitrariamente complessi
- Due tipi di blocchi logici:
  - logica combinatoria: l'output del circuito dipende solo dall'input
  - logica sequenziale: l'output del circuito dipende dall'input e dallo stato
    - il blocco ha una memoria interna

Multiplexor (o selector)

- Funzione logica di base che replica in output una delle variabili di input in base al valore di una o più variabili di controllo



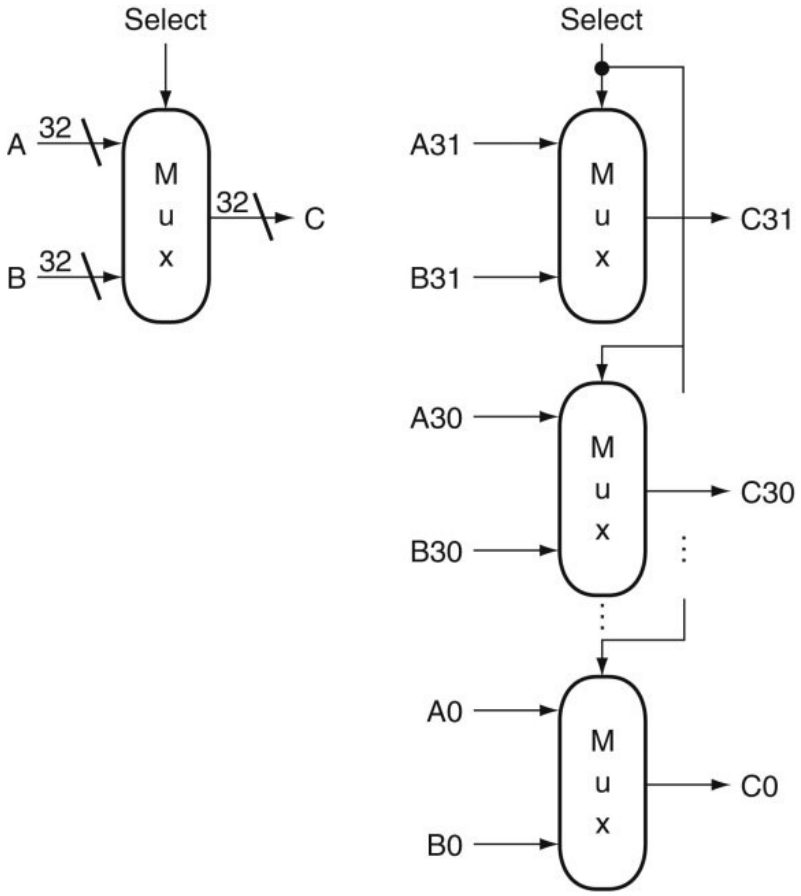
- $S = 0 \rightarrow C = A$, $S = 1 \rightarrow C = B$

  $C = A\overline{S} + BS$

- N controlli possono selezionare uno di 2^N input

Array di blocchi logici

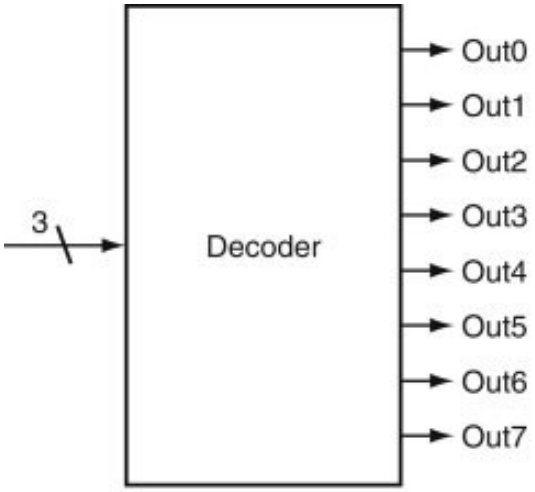
- In molti casi la stessa operazione logica deve essere applicata non a un singolo bit (per operando) ma, ad esempio, a un'intera parola di 32 bit



Esempio per un multiplexor, implementato come array di 32 multiplexor, uno per ogni coppia di bit, controllati tutti dallo stesso selettore

Decoder

- Un circuito logico con n segnali di input e 2^n segnali di output, dove solo un segnale è alto per ogni combinazione di input



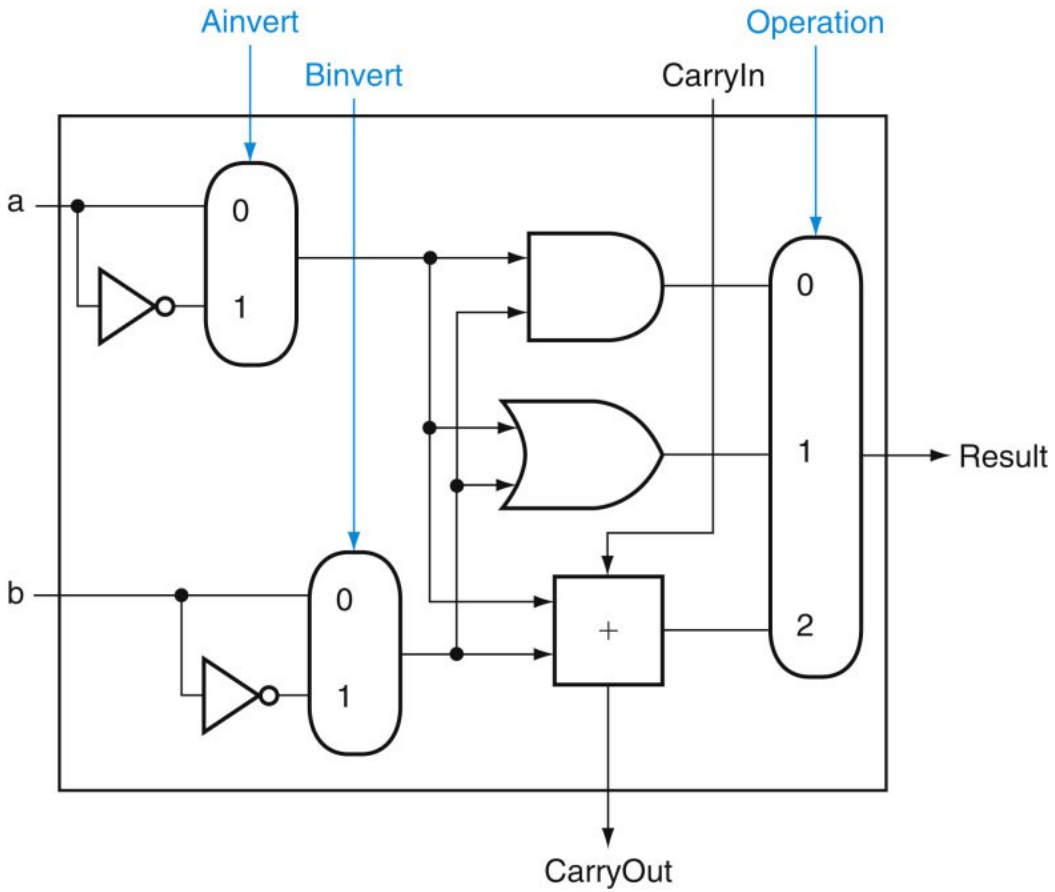
a. A 3-bit decoder

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

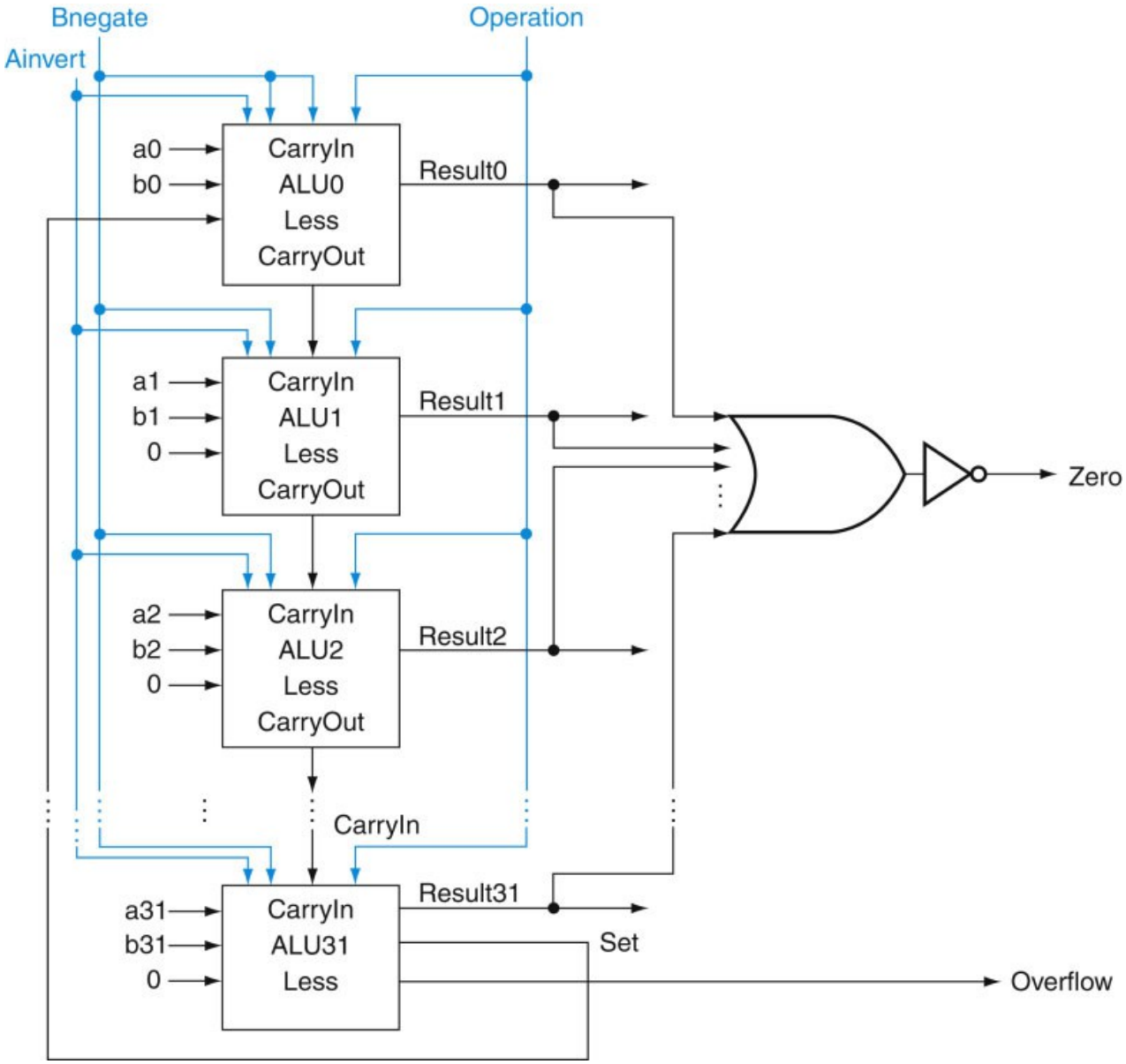
b. The truth table for a 3-bit decoder

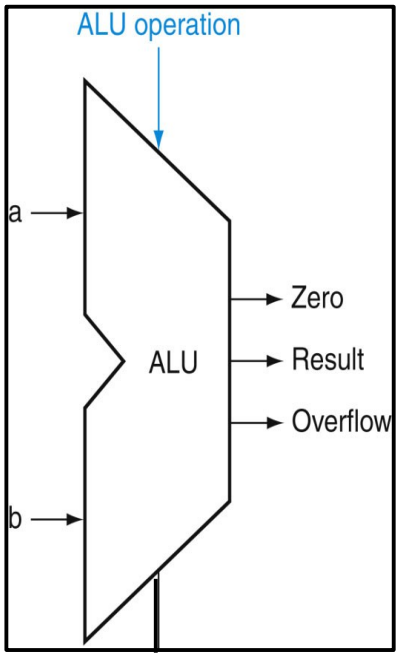
1-bit ALU

- Arithmetic Logic Unit
  - dispositivo che all'interno di un processore compie le operazioni logiche (AND, OR) e aritmetiche (ADD, SUB, …)



32-bit ALU


Bnegate
Ainvert
Operation
a0
b0
CarryIn
ALU0
Less
CarryOut
Result0
a1
b1
0
CarryIn
ALU1
Less
CarryOut
Result1
Zero
a2
b2
0
CarryIn
ALU2
Less
CarryOut
Result2
CarryIn
a31
b31
0
CarryIn
ALU31
Less
Result31
Set
Overflow


ALU operation
a
b
ALU
Zero
Result
Overflow

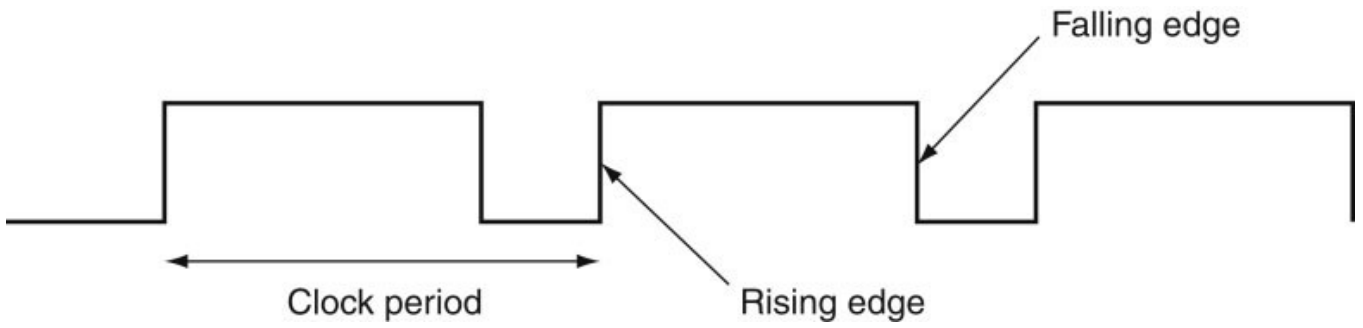
Logica sequenziale

- L'output del circuito dipende dall'input **e dallo stato**
  - il blocco ha una memoria interna
- Vogliamo combinare logica combinatoria e sequenziale in questo modo:

- Come si implementa un circuito che ricorda un valore?
- Quando è ora di aggiornare l'elemento di stato?

Clock

- Segnale che oscilla periodicament tra due valori, uno alto e uno basso



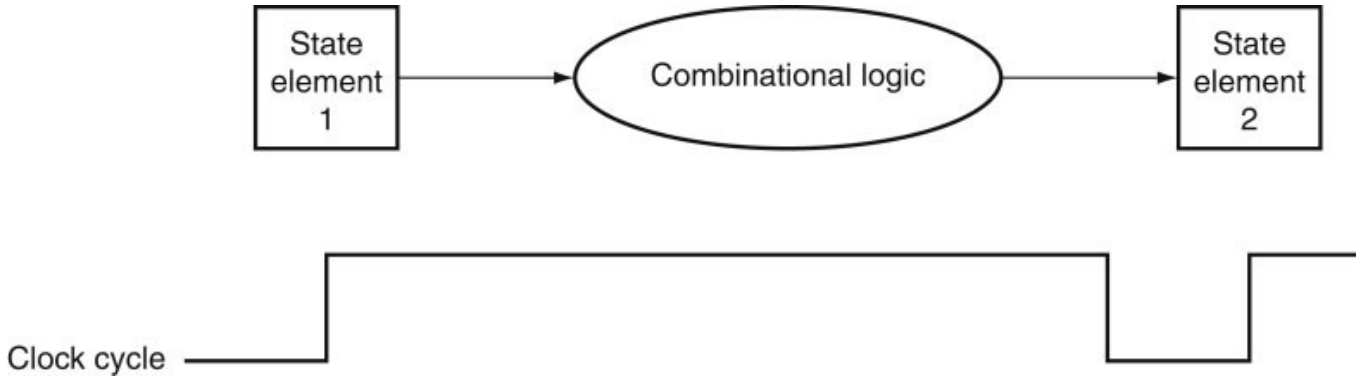
$$periodo = \frac{1}{frequenza}$$

$$frequenza = \frac{1}{periodo}$$

$$frequenza = 1\text{GHz} \Rightarrow periodo = 1\text{ns}$$

- Il clock è usato come segnale di campionamento

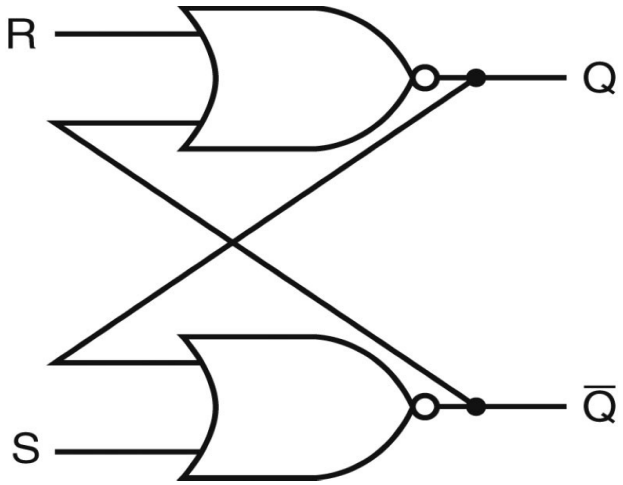
Clock



- Il primo elemento di stato fornisce gli input al circuito logico combinatorio, il cui output è usato per modificare il secondo elemento di stato
- La durata di un ciclo di clock deve essere sufficientemente lungo affinché i segnali in input riescano a propagarsi all'interno del circuito logico (**ogni** circuito logico) e generare un segnale di output stabile

S-R latch

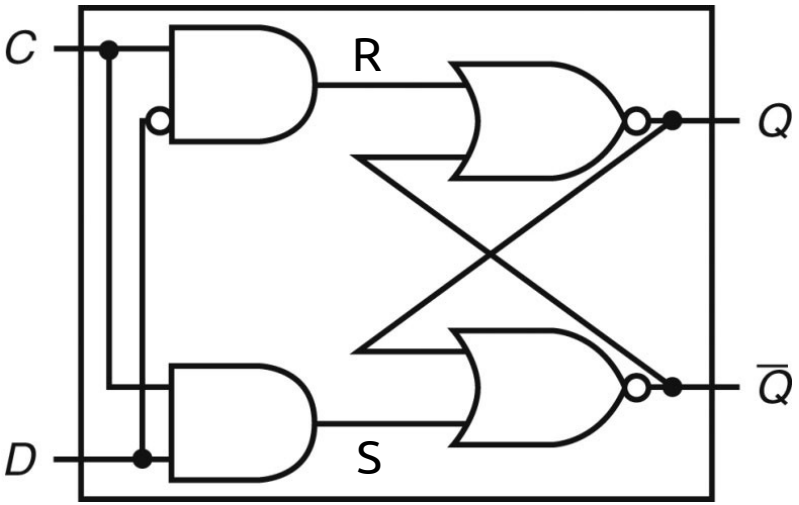
- Set-Reset latch
- E' la base per elementi di memoria più complessi
- Costituito da due NOR incrociati

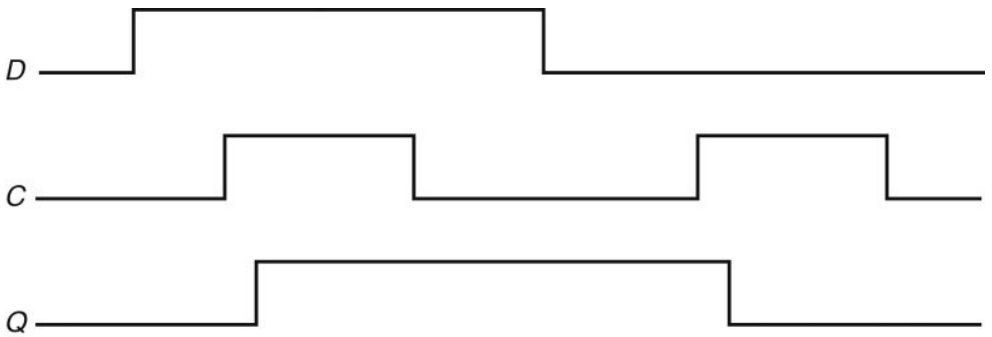


- S = 1, R = 0 → Q = 1
- S = 0, R = 1 → Q = 0
- S = 0, R = 0 → Q mantiene il proprio valore

D latch

- Clocked latch
  - C clock, D input, Q output
- C = 0 → S = R = 0
  - Q mantiene il valore precedente
- C = 1 → S = D, R = $\overline{D}$
  - Q = D
- Il latch è *trasparente*
  - se C è alto, Q assume “immediatamente” il valore di D



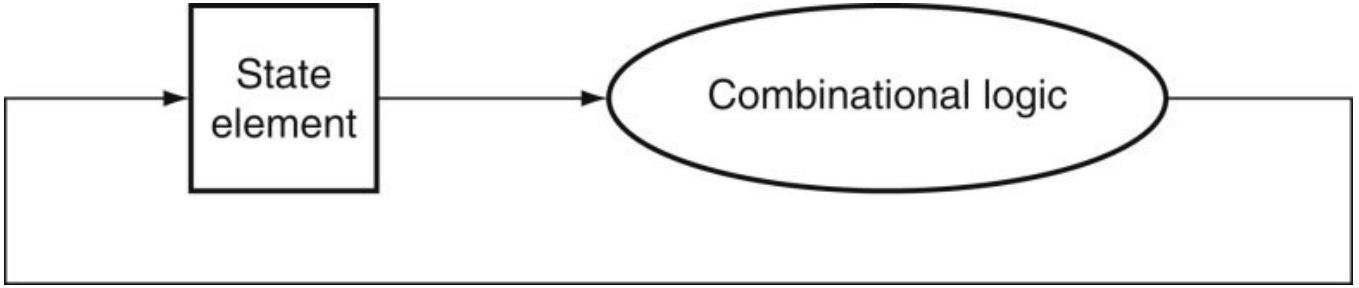


supponiamo Q inizialmente a 0

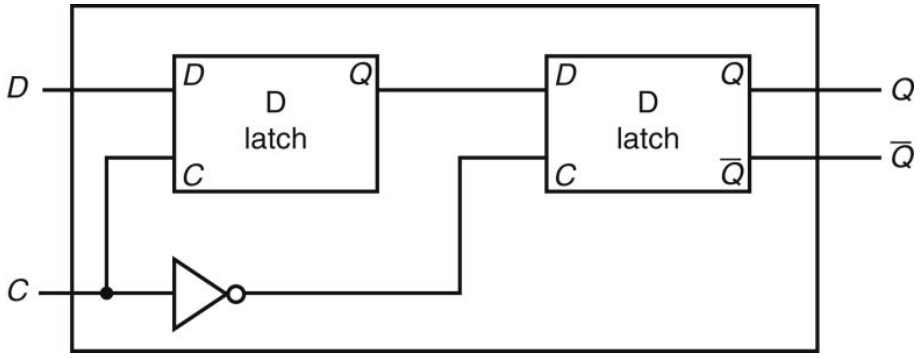
D flip-flop

- Elemento di stato che può fornire sia i segnali di input sia essere usato per salvare i segnali di output di un circuito

```
add $s0, $s0, $s1
```

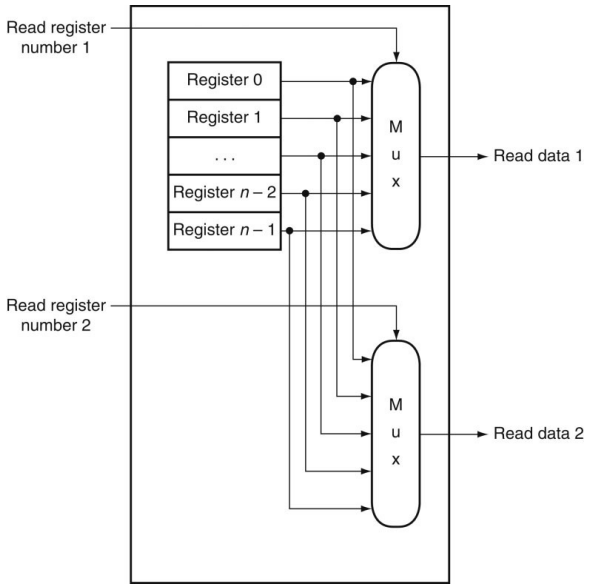


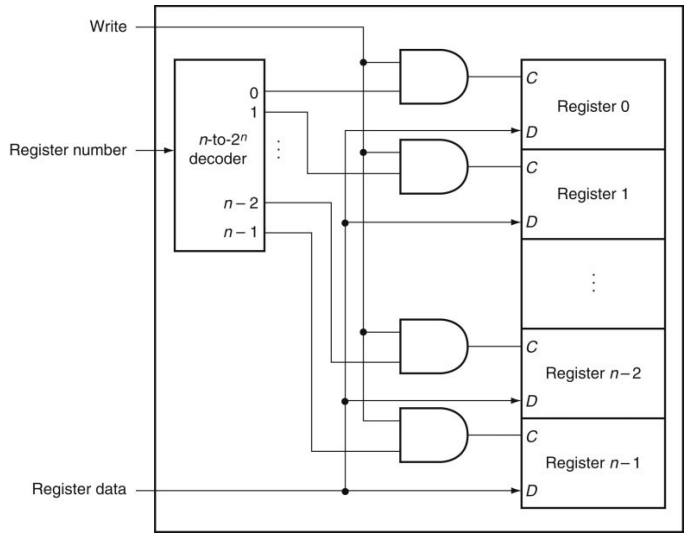
- Costituito a partire da due D latch

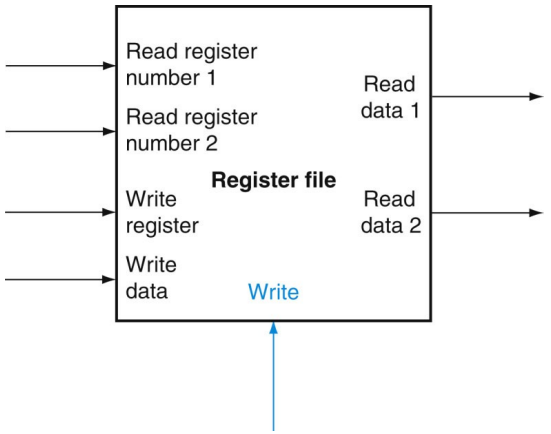


Registri e file di registri

- Un flip-flop memorizza 1 bit
- Un array di 32 flip-flop memorizza una parola di 32 bit
  - registro
- Un insieme di registri costituisce un *register file*







Altre memorie

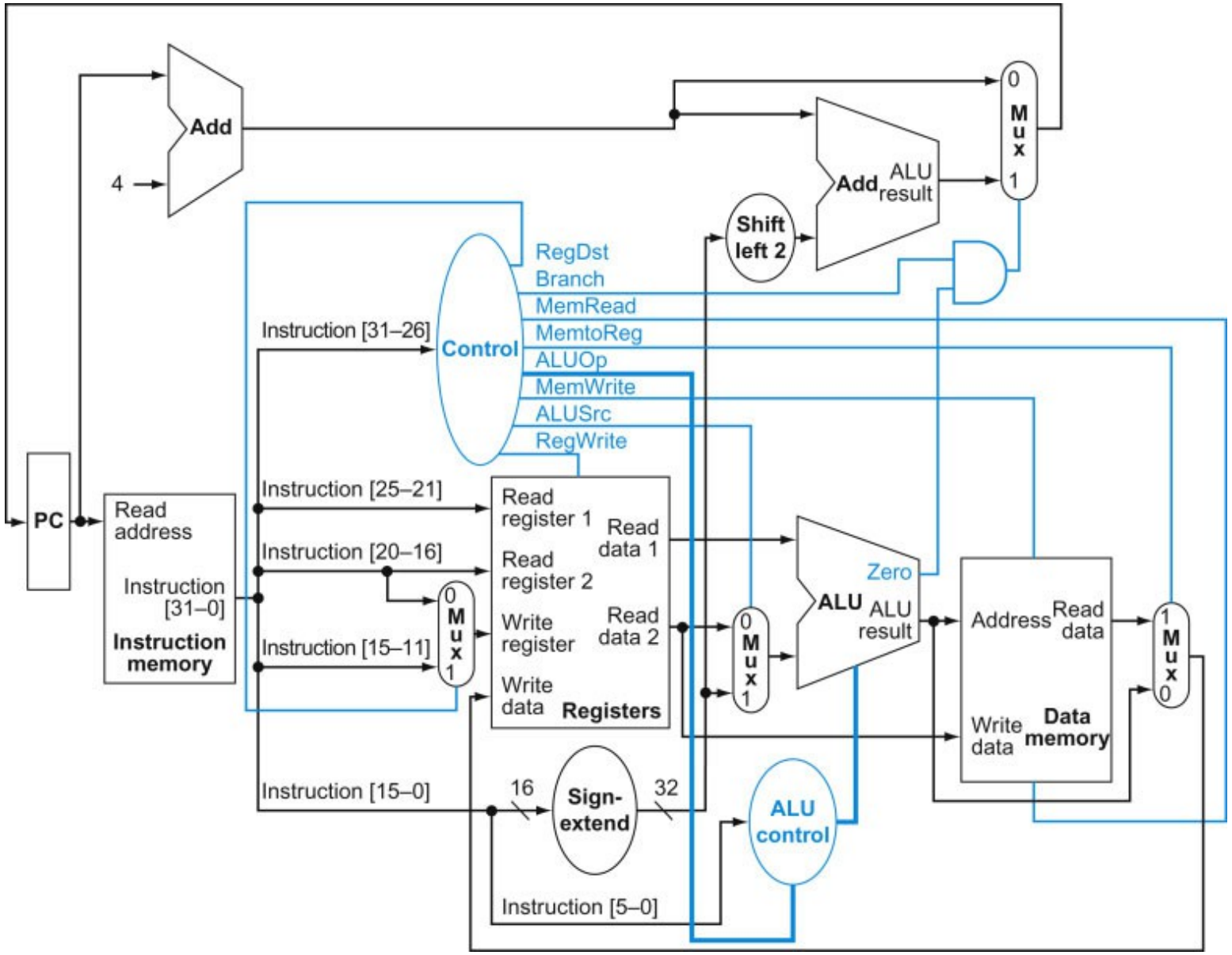
- Non è economicamente sostenibile avere memorie di dimensioni molto superiori a un register file implementate usando flip-flop
- Esistono tecnologie alternative che permettono di avere
  - \+ densità e capacità maggiori
  - \+ costi inferiori
  - – prestazioni inferiori

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

Come lavora un processore

- (Discussione semplificata basata su MIPS)
- Per ogni istruzione:
  - legge (*fetch*) dalla memoria l'istruzione il cui indirizzo è nel Program Counter
  - legge dei registri in base a quanto specificato nell'istruzione
- A seconda dell'istruzione:
  - usa la ALU per eseguire calcoli
  - accede alla memoria per load/store
  - aggiorna il Program Counter con l'indirizzo della prossima istruzione da eseguire

Datapath

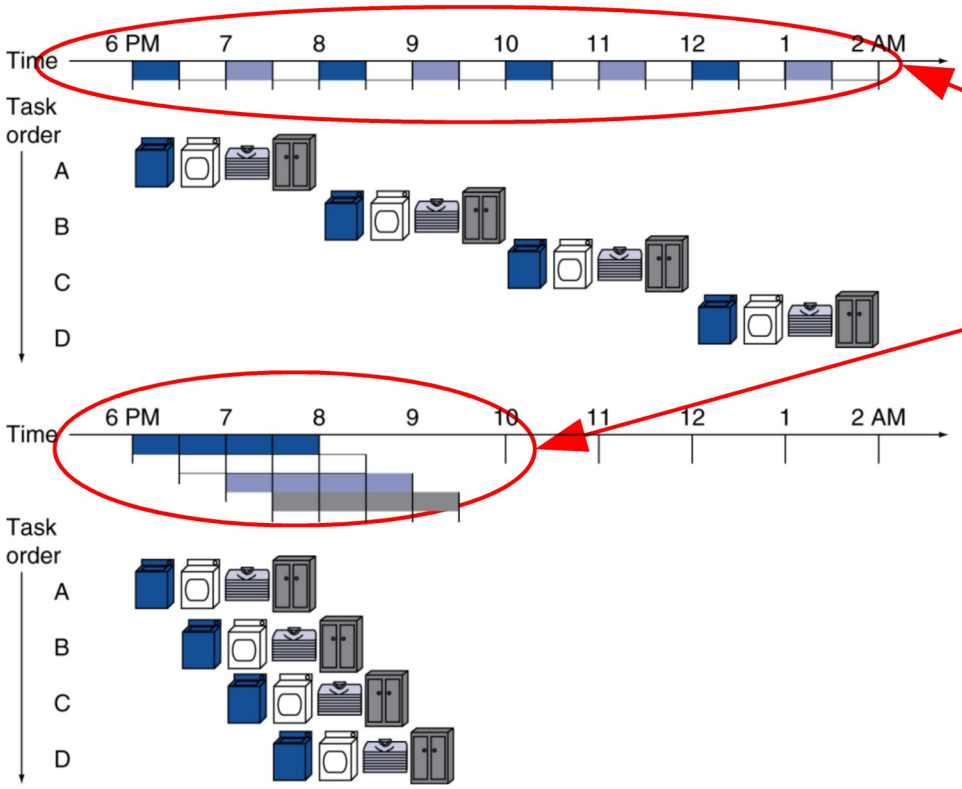

Add
4
0
M
u
x
1
Add
ALU result
Shift left 2
RegDst
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Control
Instruction [31–26]
PC
Read address
Instruction [31–0]
Instruction memory
Instruction [25–21]
Instruction [20–16]
Instruction [15–11]
Instruction [15–0]
Instruction [5–0]
Read register 1
Read register 2
Write register
Write data
Read data 1
Read data 2
Registers
Zero
ALU
ALU result
Address
Read data
Write data
Data memory
16
Sign-extend
32
ALU control

Pipelining

- L'implementazione di un datapath basata su un singolo ciclo di clock è inefficiente
  - il ciclo di clock deve essere sufficiente ad eseguire l'intera istruzione più lenta
    - tipicamente accesso in memoria
  - CPI = 1, ma ciclo di clock molto alto
  - può andare bene solo per instruction set molto semplici
- Il **pipelining** è una tecnica implementativa in cui le esecuzioni di più istruzioni sono sovrapposte nel tempo

Esempio

- Basato su una lavanderia
  - lava, asciuga, piega, metti via
  - in un dato momento ogni carico usa una risorsa diversa



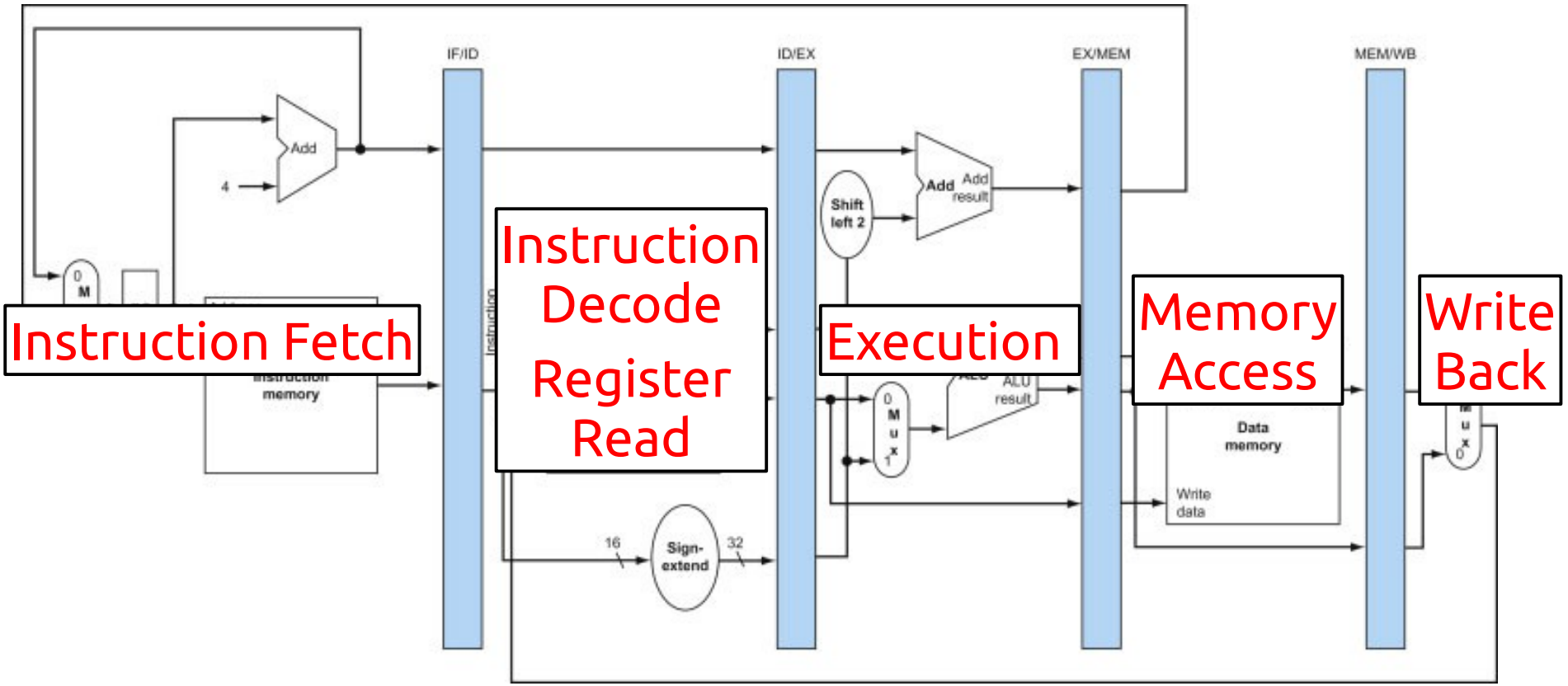
- Speed-up
  - 4 carichi

    8 / 3.5 = 2.3
  - N carichi

    2N / (0,5N + 1,5) → 4

    = numero di passi (stadi)
- Aumenta il numero di carichi eseguiti per unità di tempo (throughput), non la durata di un singolo carico (latenza)

Pipelined datapath



- Il ciclo di clock ora corrisponde allo stadio più lungo
- Se gli stadi non sono perfettamente bilanciati
  - lo speed-up massimo non è dato dal numero di stadi
  - la latenza della singola istruzione può aumentare

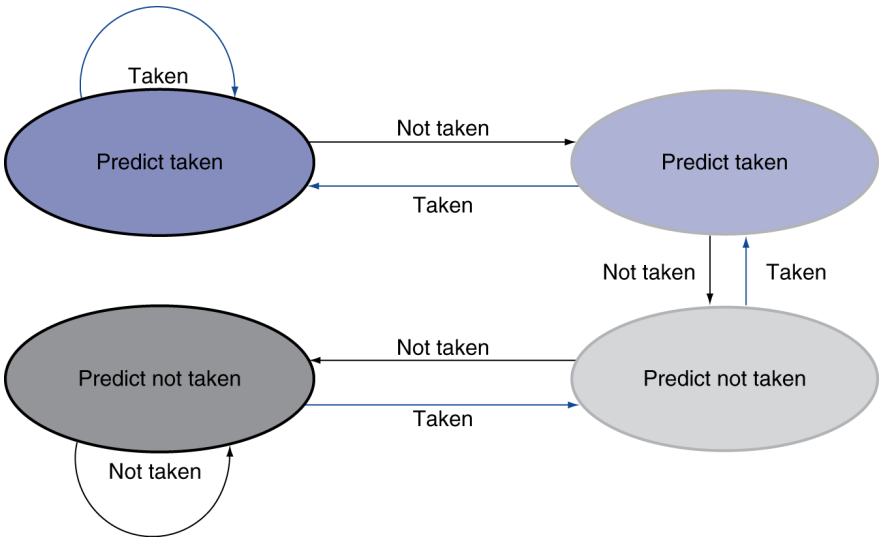
Criticità

- Situazioni che impediscono la partenza dell'istruzione successiva nel ciclo successivo
- Criticità strutturali: una risorsa necessaria all'esecuzione di una istruzione è occupata
- Criticità sui dati: un'istruzione dipende da un valore non ancora disponibile
- Criticità sul controllo: una decisione di controllo dipende dal risultato di un'istruzione non ancora conclusa

- La soluzione generale in caso di criticità è **aspettare**
  - spreco di cicli di clock → stalli

Branch prediction

- Per mitigare la perdita di tempo dovuta all'attesa del risultato di un branch si può provare a indovinare
- Dynamic branch prediction
  - L'hardware misura il comportamento recente effettivo
- Se la predizione è sbagliata
  - **branch miss**
  - i risultati parziali vanno scartati
  - si carica l'istruzione corretta
  - si aggiorna la storia



Indicatori di performance

```
$ perf stat -d ./a.out

 Performance counter stats for './a.out':

      1992.419970 task-clock (msec)         #    0.999 CPUs utilized
              215 context-switches          #    0.108 K/sec
                5 cpu-migrations            #    0.003 K/sec
              368 page-faults               #    0.185 K/sec
    4,139,391,573 cycles                    #    2.078 GHz                      [44.54%]
    2,207,522,429 stalled-cycles-frontend   #   53.33% frontend cycles idle     [44.67%]
    2,182,353,973 stalled-cycles-backend    #   52.72% backend  cycles idle     [44.79%]
    2,302,995,011 instructions              #    0.56  insns per cycle
                                            #    0.96  stalled cycles per insn [55.85%]
      657,124,588 branches                  #  329.812 M/sec                    [55.85%]
      155,694,678 branch-misses             #   23.69% of all branches          [55.67%]
      329,767,307 L1-dcache-loads           #  165.511 M/sec                    [55.45%]
       20,723,322 L1-dcache-load-misses     #    6.28% of all L1-dcache hits   [55.25%]
          200,896 LLC-loads                 #    0.101 M/sec                    [44.15%]
   <not supported> LLC-load-misses:HG

      1.994174760 seconds time elapsed
```

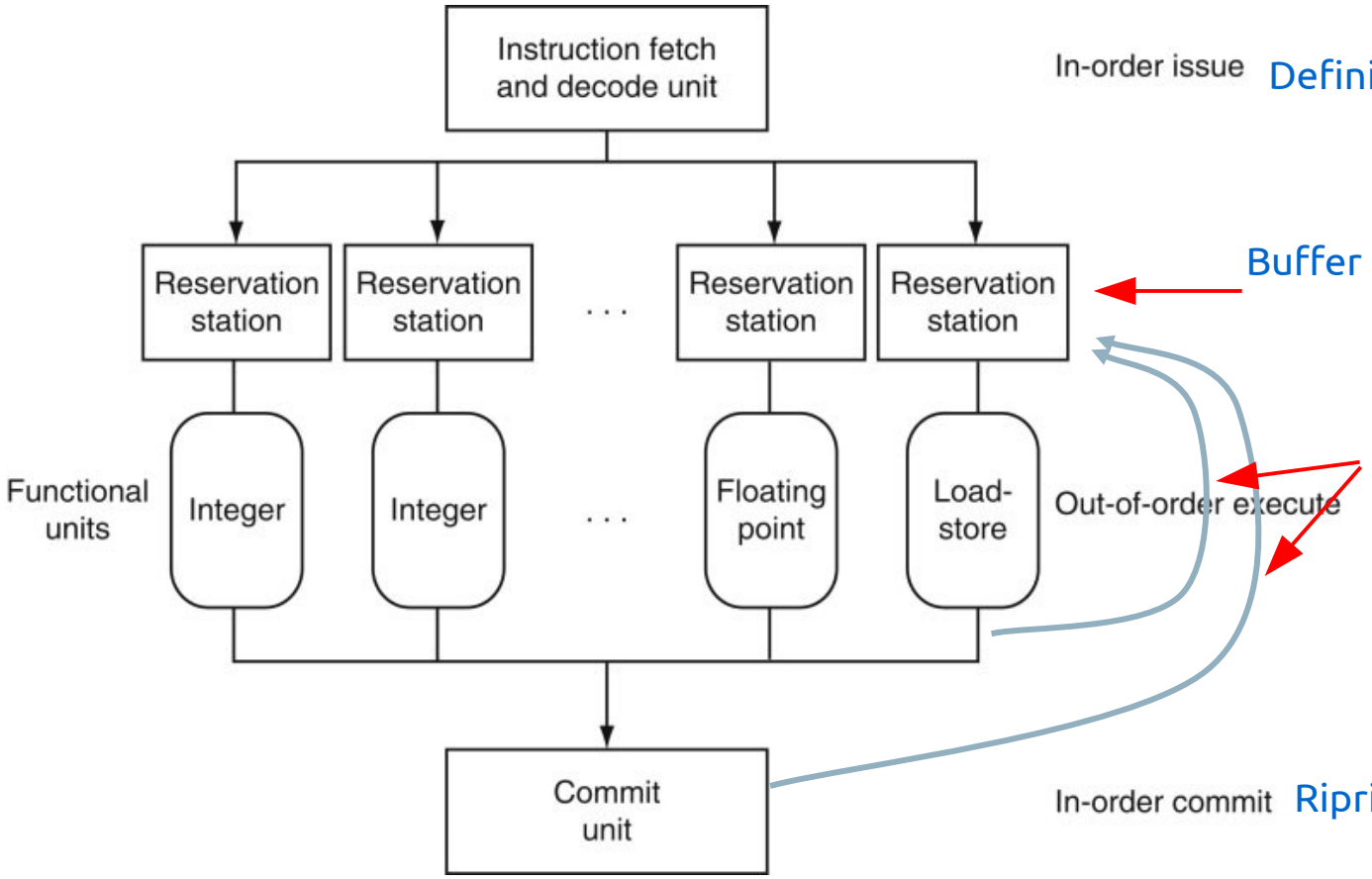
Instruction-level parallelism (ILP)

- Per aumentare il grado di ILP
  - pipeline con più stadi (più profonde)
    - singoli stadi più semplici → **riduzione ciclo di clock**
  - multiple issue
    - più istruzioni sono iniziate contemporaneamente
    - replicare la pipeline o alcuni stadi della pipeline
    - processore “superscalare”
    - in questo modo si può ridurre il CPI a un valore < 1
      - uso di **Instructions Per Cycle** IPC = 1 / CPI
    - efficacia limitata dalle criticità
      - dipendenze e condivisione di risorse

Indicatori di performance

```
$ perf stat -d ./a.out

 Performance counter stats for './a.out':

      1992.419970 task-clock (msec)         #    0.999 CPUs utilized
              215 context-switches          #    0.108 K/sec
                5 cpu-migrations            #    0.003 K/sec
              368 page-faults               #    0.185 K/sec
    4,139,391,573 cycles                    #    2.078 GHz                      [44.54%]
    2,207,522,429 stalled-cycles-frontend   #   53.33% frontend cycles idle     [44.67%]
    2,182,353,973 stalled-cycles-backend    #   52.72% backend  cycles idle     [44.79%]
    2,302,995,011 instructions              #    0.56  insns per cycle
                                            #    0.96  stalled cycles per insn [55.85%]
      657,124,588 branches                  #  329.812 M/sec                    [55.85%]
      155,694,678 branch-misses             #   23.69% of all branches          [55.67%]
      329,767,307 L1-dcache-loads           #  165.511 M/sec                    [55.45%]
       20,723,322 L1-dcache-load-misses     #    6.28% of all L1-dcache hits   [55.25%]
          200,896 LLC-loads                 #    0.101 M/sec                    [44.15%]
   <not supported> LLC-load-misses:HG

      1.994174760 seconds time elapsed
```

Dynamic pipeline scheduling



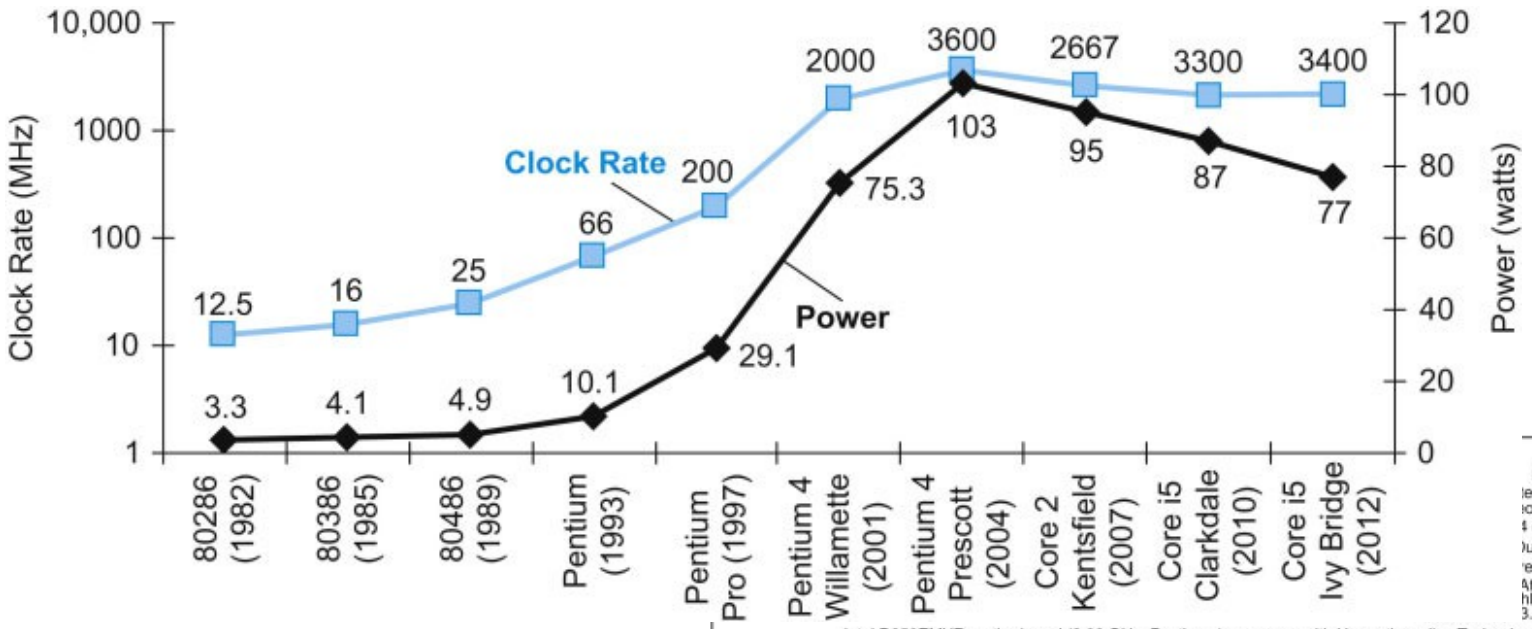
In-order issue — Definisce le dipendenze tra istruzioni

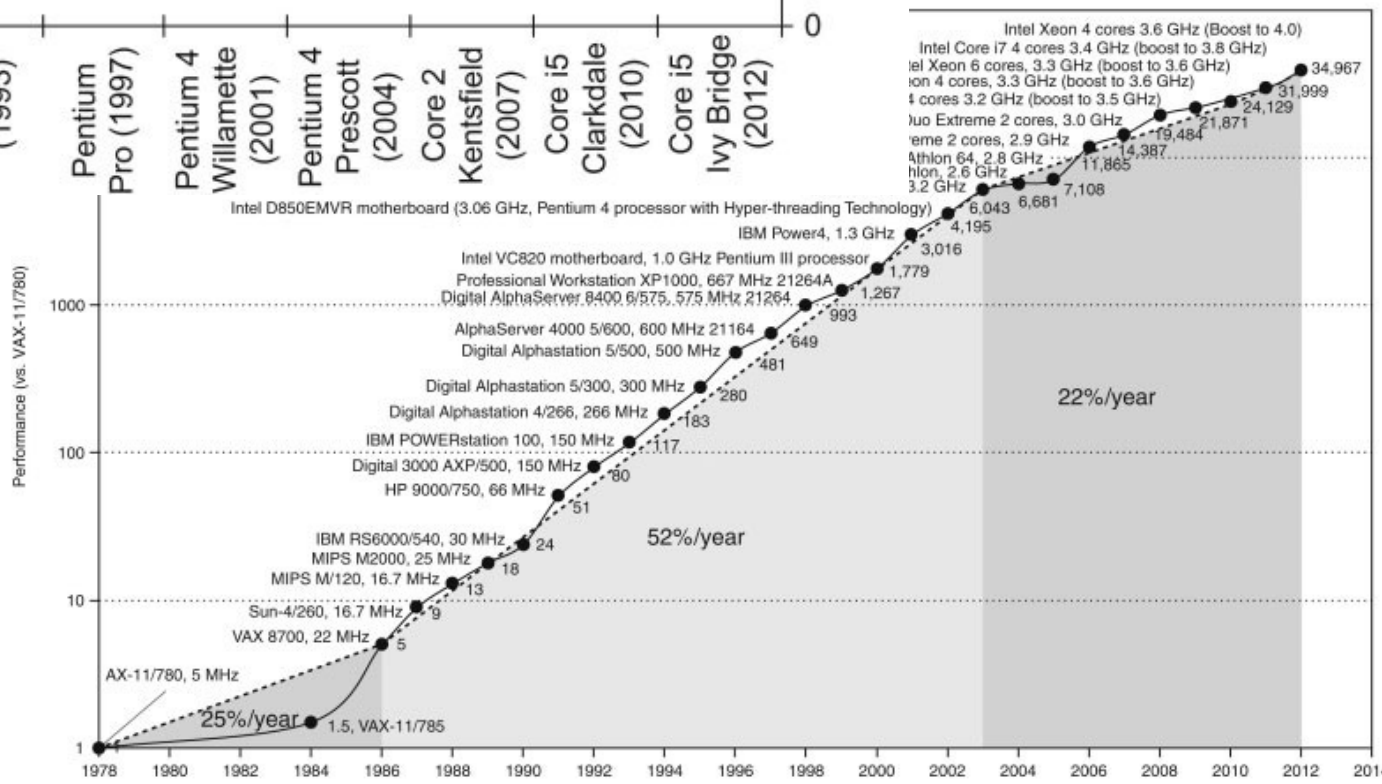
Buffer di operazioni (e relativi operandi) ancora da eseguire

I risultati inizializzano gli operandi per operazioni in attesa

In-order commit — Ripristina le dipendenze tra istruzioni

A proposito di clock cycle



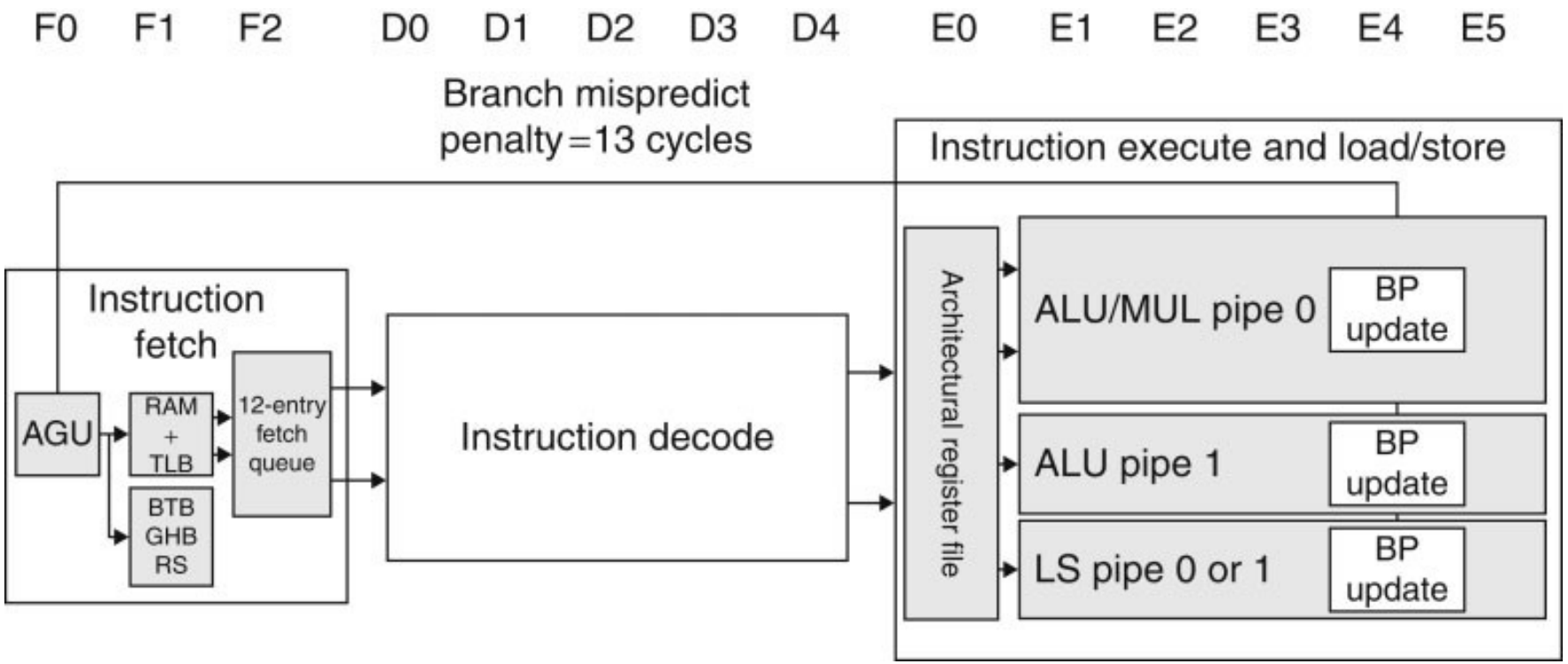


Power μ V^2 f

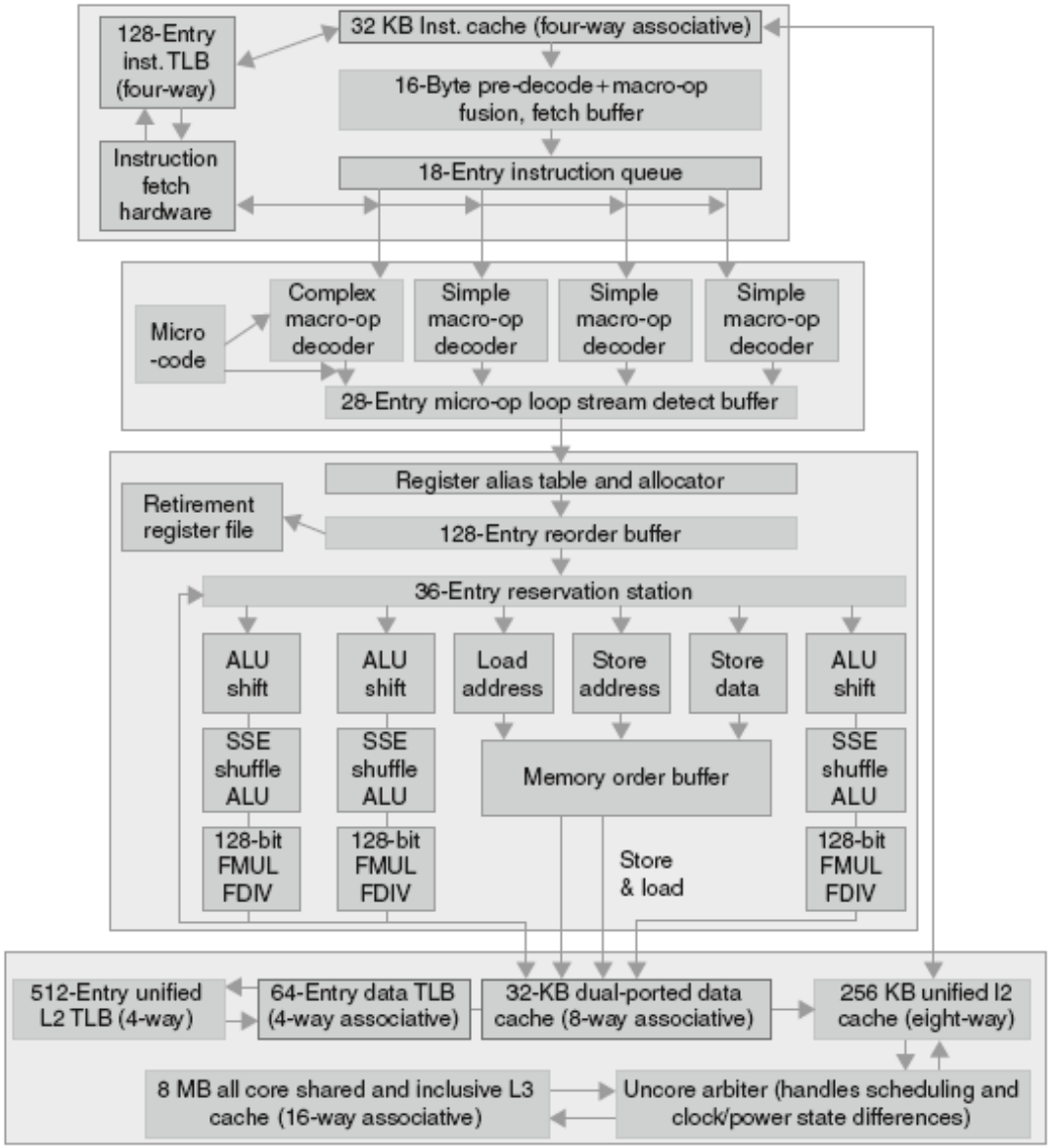
ARM A8 e Intel Core i7

Processor	ARM A8	Intel Core i7 920
Market	Personal Mobile Device	Server, Cloud
Thermal design power	2 Watts	130 Watts
Clock rate	1 GHz	2.66 GHz
Cores/Chip	1	4
Floating point?	No	Yes
Multiple Issue?	Dynamic	Dynamic
Peak instructions/clock cycle	2	4
Pipeline Stages	14	14
Pipeline schedule	Static In-order	Dynamic Out-of-order with Speculation
Branch prediction	2-level	2-level
1st level caches / core	32 KiB I, 32 KiB D	32 KiB I, 32 KiB D
2nd level cache / core	128–1024 KiB	256 KiB
3rd level cache (shared)	–	2–8 MiB

A8 pipeline


F0
F1
F2
D0
D1
D2
D3
D4
E0
E1
E2
E3
E4
E5
Branch mispredict
penalty = 13 cycles
Instruction execute and load/store
Instruction
fetch
AGU
RAM
+
TLB
BTB
GHB
RS
12-entry
fetch
queue
Instruction decode
Architectural register file
ALU/MUL pipe 0
BP
update
ALU pipe 1
BP
update
LS pipe 0 or 1
BP
update

Intel i7 pipeline

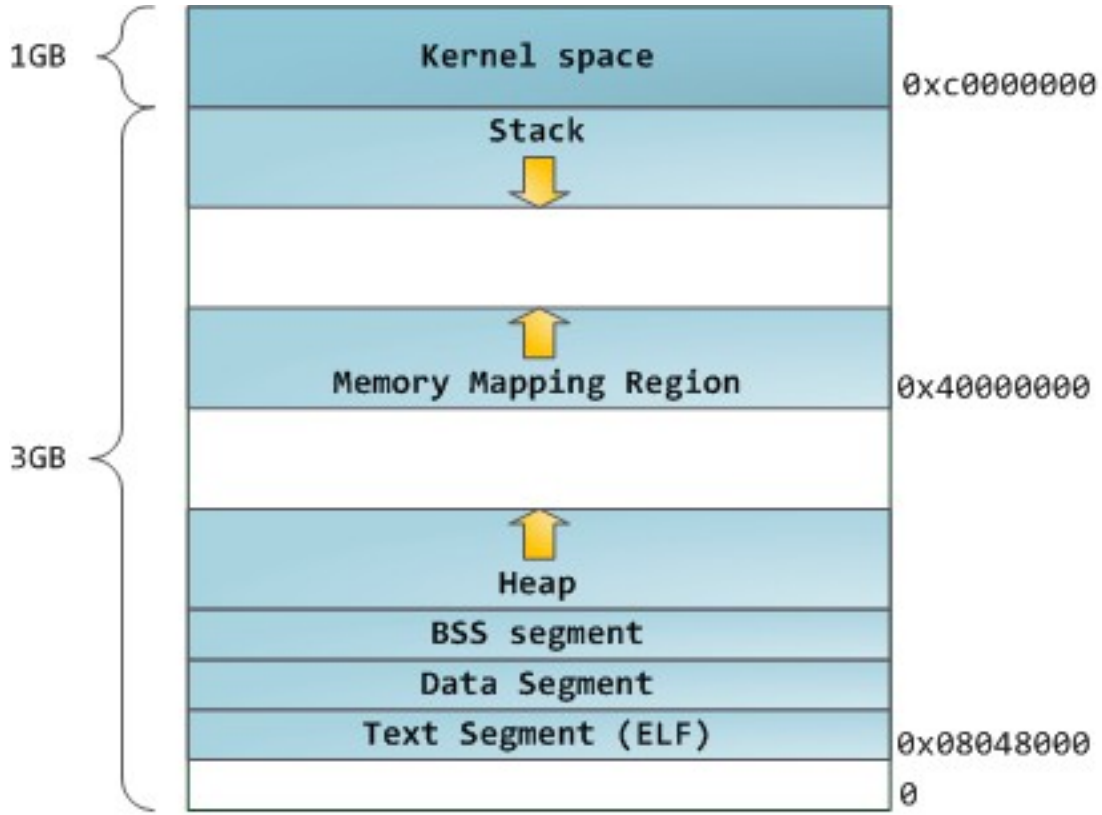


Memoria

- La memoria può essere vista come una sequenza mono-dimensionale di *locazioni*, ognuna associata a un *indirizzo*
  - Di solito la minima unità indirizzabile è il byte
- Ad ogni processo viene allocata un'area di memoria dedicata
  - la memoria è virtuale
    - ad es. se l'area è di 4 GB, ogni processo può usare indirizzi nell'intervallo [`0x00000000`, `0xFFFFFFFF`], ma questi non corrispondono necessariamente agli stessi indirizzi in memoria fisica
  - la memoria fisica viene allocata quando necessario
  - esiste un meccanismo di mapping tra indirizzi virtuali e fisici

Memory layout di un processo

- Text
  - istruzioni del programma
- Data e BSS Segment
  - variabili globali e *static* inizializzate e non (= `0`)
- Heap
  - allocazioni con malloc e new
- Stack
  - es. variabili locali di una funzione



http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory/

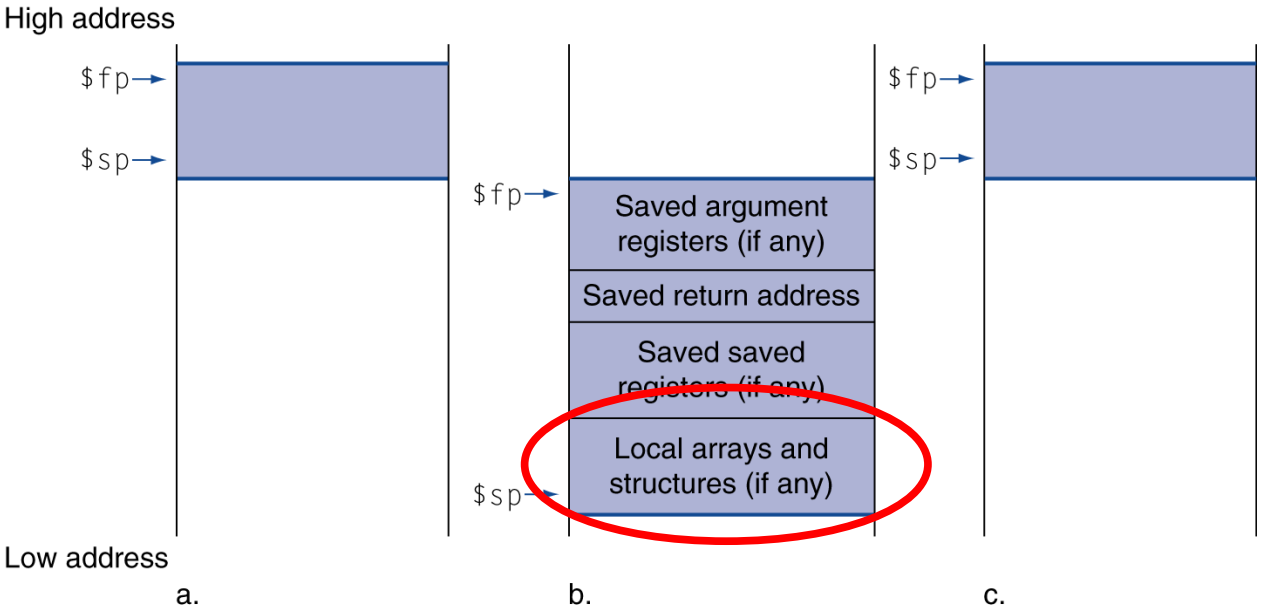
Chiamata di funzione

```
void swap(int* i, int* j) { … }
int max(int* A, int N) { … }
int increment(int i) { … }

int main()
{
  int const N = …;
  int A[N] = { … };
  int i = …;

  int o = increment(i);
  swap(&o, &i);
  int m = max(A, N);
}
```

Chiamata di funzione

- Passi necessari
  - Mettere i parametri dove la funzione chiamata li possa accedere
  - Trasferire il controllo alla funzione
  - Acquisire spazio di memoria per l'esecuzione della funzione
  - Eseguire le operazioni della funzione
  - Mettere il risultato dove la funzione chiamante lo possa accedere
  - Ritornare il controllo al chiamante

Stack



NB: questa disposizione è un esempio

- Stack: coda LIFO (Last-In, First-Out)
- Lo stack di solito cresce verso il basso
  - da indirizzi alti a indirizzi bassi
- Per ogni chiamata di funzione viene allocata sullo stack un'area dedicata (*frame* o *activation record*)
  - il frame viene deallocato alla fine della chiamata

Cosa ci va in uno stack frame?

- Dipende dall'hardware, dal compilatore, …
- Dati necessari per la gestione della catena delle chiamate
- Argomenti passati alla funzione
  - ma alcuni di solito sono passati attraverso i registri per evitare accessi in memoria
- Registri da preservare durante una chiamata di funzione
  - il numero di registri è limitato
- Variabili locali (automatiche)

Tutte queste informazioni sono note al compile time

⇒ l'allocazione di un frame consiste nell'incremento dello stack pointer

Stack vs Heap

- Runtime overhead

sorgente

```
void allocazione_stack()
{
  int m{123};
}

void allocazione_heap()
{
  int* m = new int{123};
  delete m;
}
```

assembler

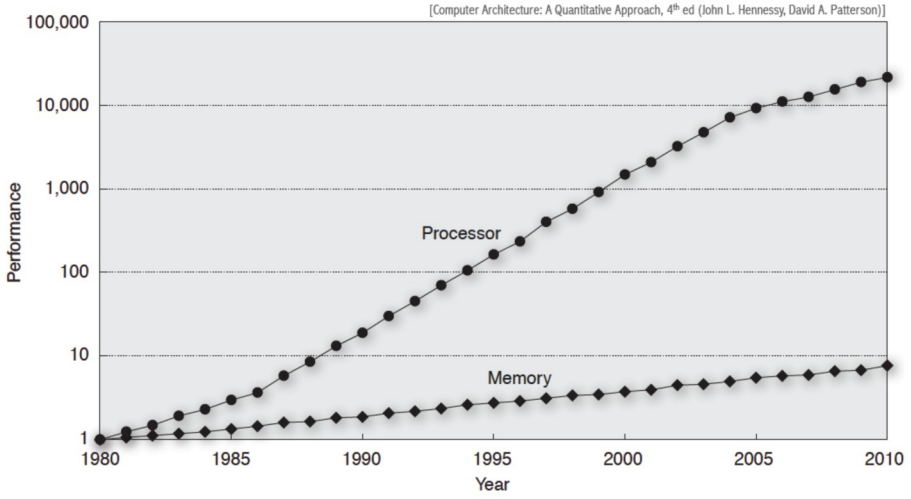
```
allocazione_stack:
     subq %4, %rsp
     movl $123, (%rsp)
     addq $4, %rsp
     ret

allocazione_heap:
     subq $8, %rsp
     movl $4, %edi
     call operator new(unsigned long)
     movl $123, (%rax)
     movl $4, %esi
     movq %rax, %rdi
     call operator delete(void*, unsigned long)
     addq $8, %rsp
     ret
```

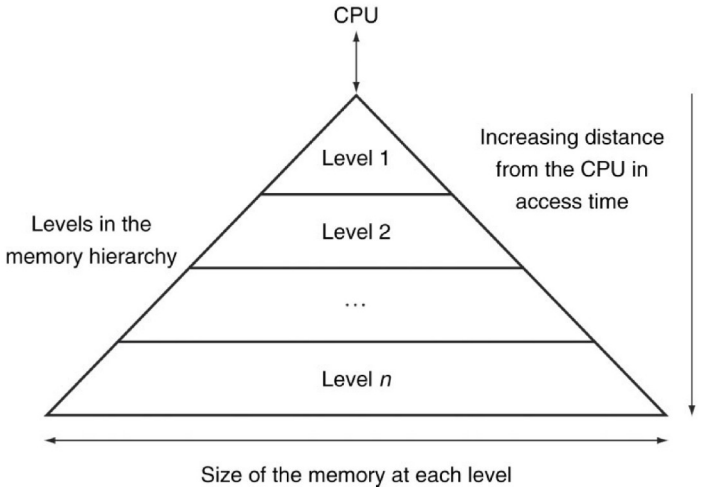
- Frammentazione heap

Qual è la memoria ideale?

Memory technology	Typical access time	$ per GiB in 2012
SRAM semiconductor memory	0.5–2.5 ns	$500–$1000
DRAM semiconductor memory	50–70 ns	$10–$20
Flash semiconductor memory	5,000–50,000 ns	$0.75–$1.00
Magnetic disk	5,000,000–20,000,000 ns	$0.05–$0.10

- La memoria ideale dovrebbe avere le prestazioni della SRAM e la capacità e il costo del disco
- Possiamo approssimare la situazione ideale
  - usando una gerarchia di memorie
  - sfruttando il principio di località





Principio di località

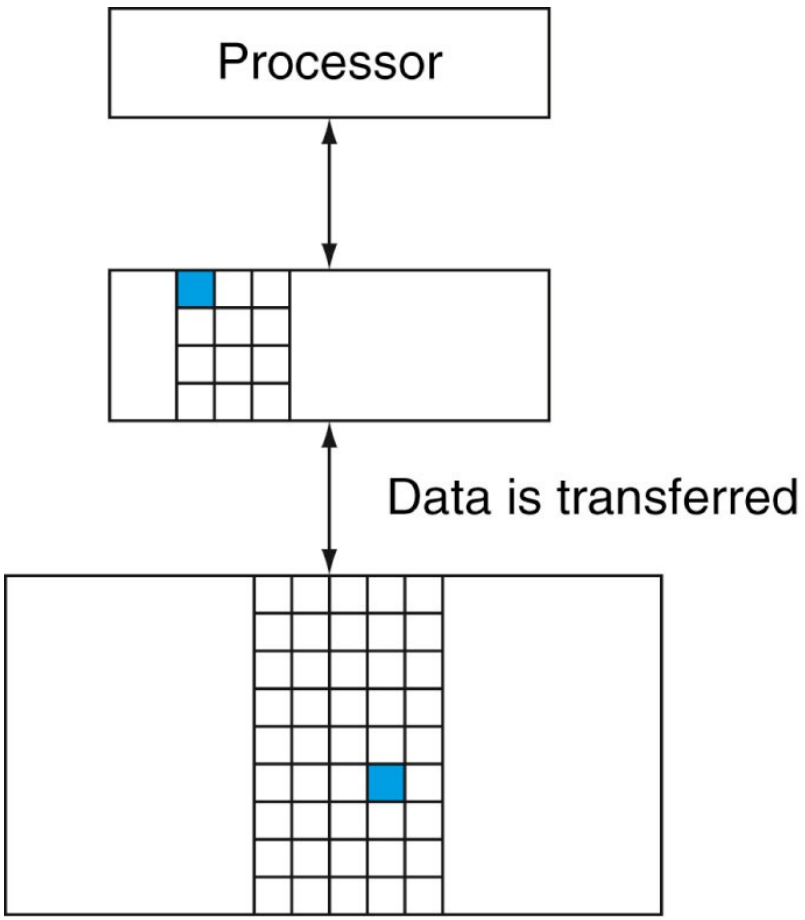
```
size_t strlen(char const* s)
{
  size_t l = 0;
  while (*s++) ++l;
  return l;
}
```

- Dati
  - Accessi ripetuti alla variabile `l`
  - Scansione dell'array `s`
- Istruzioni
  - Ripetizione delle espressioni/istruzioni `*s++` e `++l`
  - Esecuzione di istruzioni consecutive

Principio di località

- In un dato intervallo limitato di tempo, un programma accede a una piccola parte del proprio spazio indirizzabile
- Località temporale
  - Elementi acceduti recentemente tendono ad essere acceduti nuovamente nel prossimo futuro
  - Esempi: istruzioni e variabili contatore in un loop
- Località spaziale
  - Elementi vicini a quelli acceduti recentemente tendono ad essere acceduti nel prossimo futuro
  - Esempi: accesso sequenziale a istruzioni, dati in un array

Livelli di gerarchia



- Il dato cercato **è presente** nel livello superiore
  - *hit*

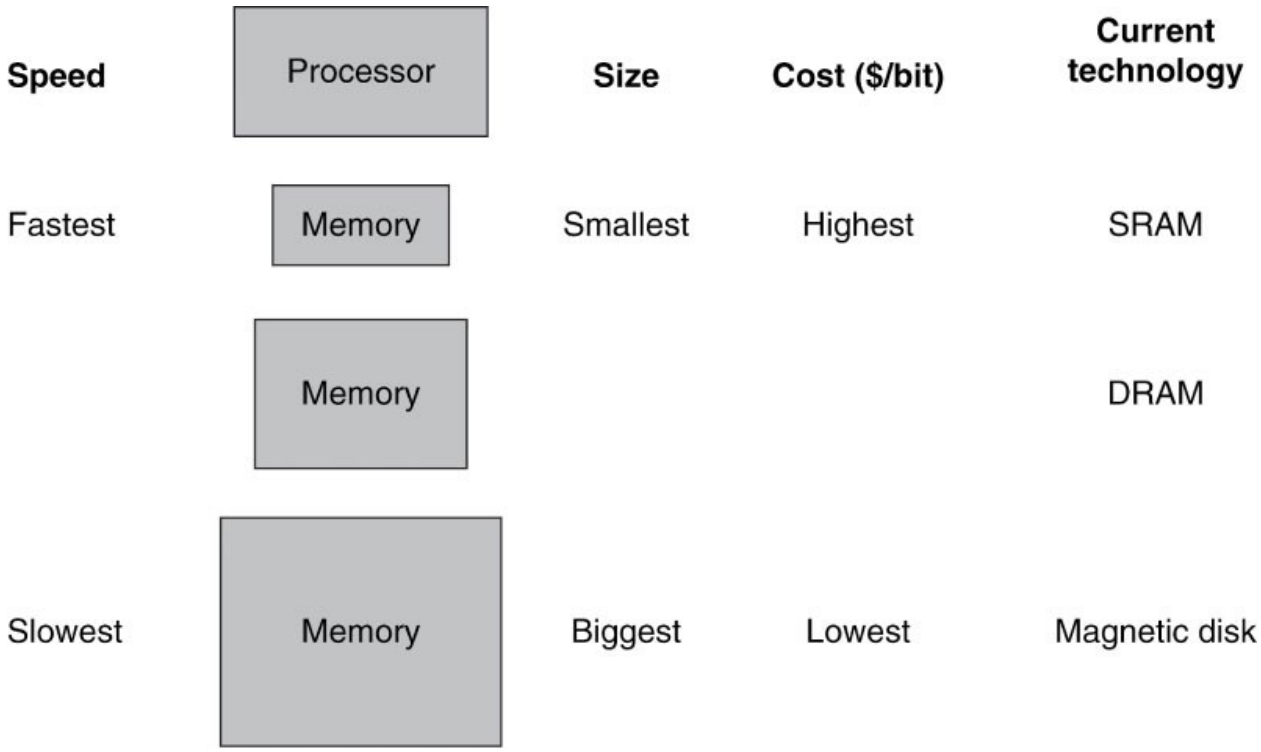
    hit rate = hits / accessi
- Il dato cercato **non è presente** nel livello superiore
  - *miss*: il dato è cercato nel livello inferiore e da lì copiato

    miss rate = misses / accessi = 1 – hit rate
  - miss penalty: tempo impiegato per ricerca e copia
  - una volta copiato nel livello superiore, il dato è acceduto come se fosse un hit
  - un miss causa stall nella pipeline (mitigabile con multiple issue, speculation, out-of-order, ...)
- Di solito non viene copiato il singolo dato ma un blocco (*cache line*) di più byte (es. 64 byte)

Memory-stall time = Memory accesses × Miss rate × Miss penalty

Livelli di gerarchia

- Configurazione tipica

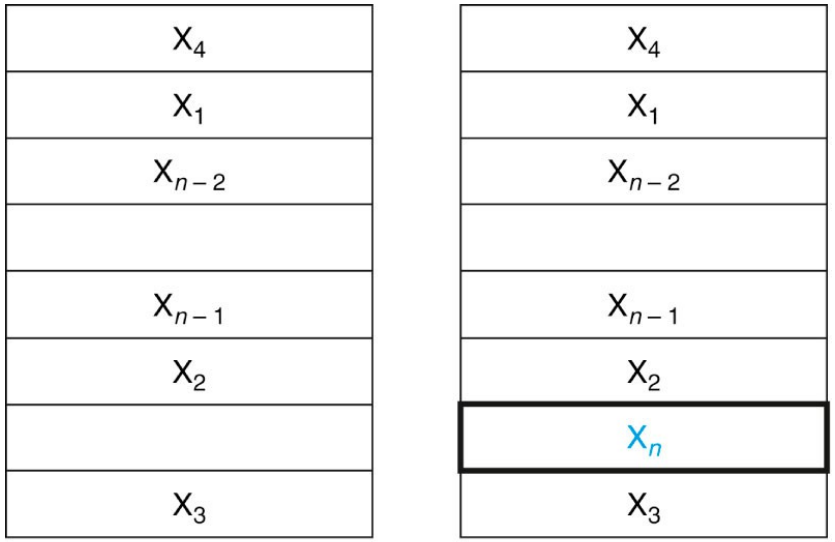


- Illusione di avere memoria contemporaneamente capace, veloce e a costi accettabili

Memoria cache

- Livello della gerarchia di memoria più vicino al processore
- Capacità limitata ma accesso veloce

- Dati gli accessi $X_1, X_2, \ldots, X_n$



a. Before the reference to X_n b. After the reference to X_n

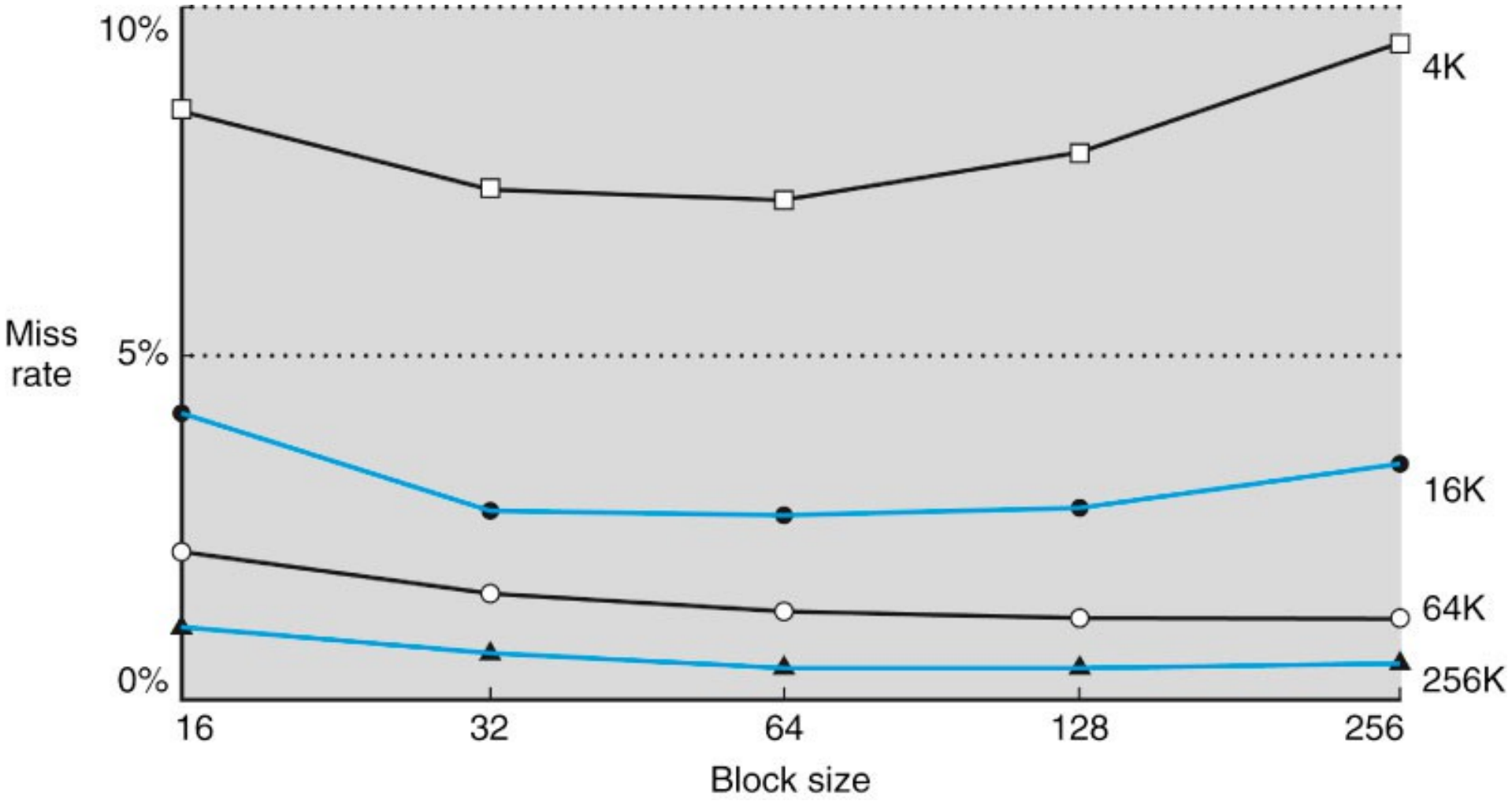
- Come sappiamo se un dato è già nella cache?
- Se è nella cache, dove sta?
- Diverse soluzioni possibili, ma in generale la posizione dipende dall'indirizzo

Il primo accesso a X_n causa un cache miss

Dimensione del blocco

- Qual è la dimensione ideale del blocco?
- Blocchi più grandi sfruttano la località spaziale → maggiore hit rate
- Ma...
  - blocco più grande → meno blocchi → più competizione → minore hit rate
    - a parità di dimensione della cache
  - blocco più grande → più dati da trasferire per ogni miss → maggiore miss penalty

Miss rate vs block size



Prestazioni della cache

Average memory access time = Hit time + Miss rate × Miss penalty

- Come si può migliorare la prestazione della cache?
- Si può aumentare l'hit rate/ridurre il miss rate
  - introduzione di meccanismi più flessibili nell'allocazione dei blocchi in una cache → *associatività*
- Si può ridurre la penalizzazione dovuta a un miss
  - introduzione di cache a più livelli

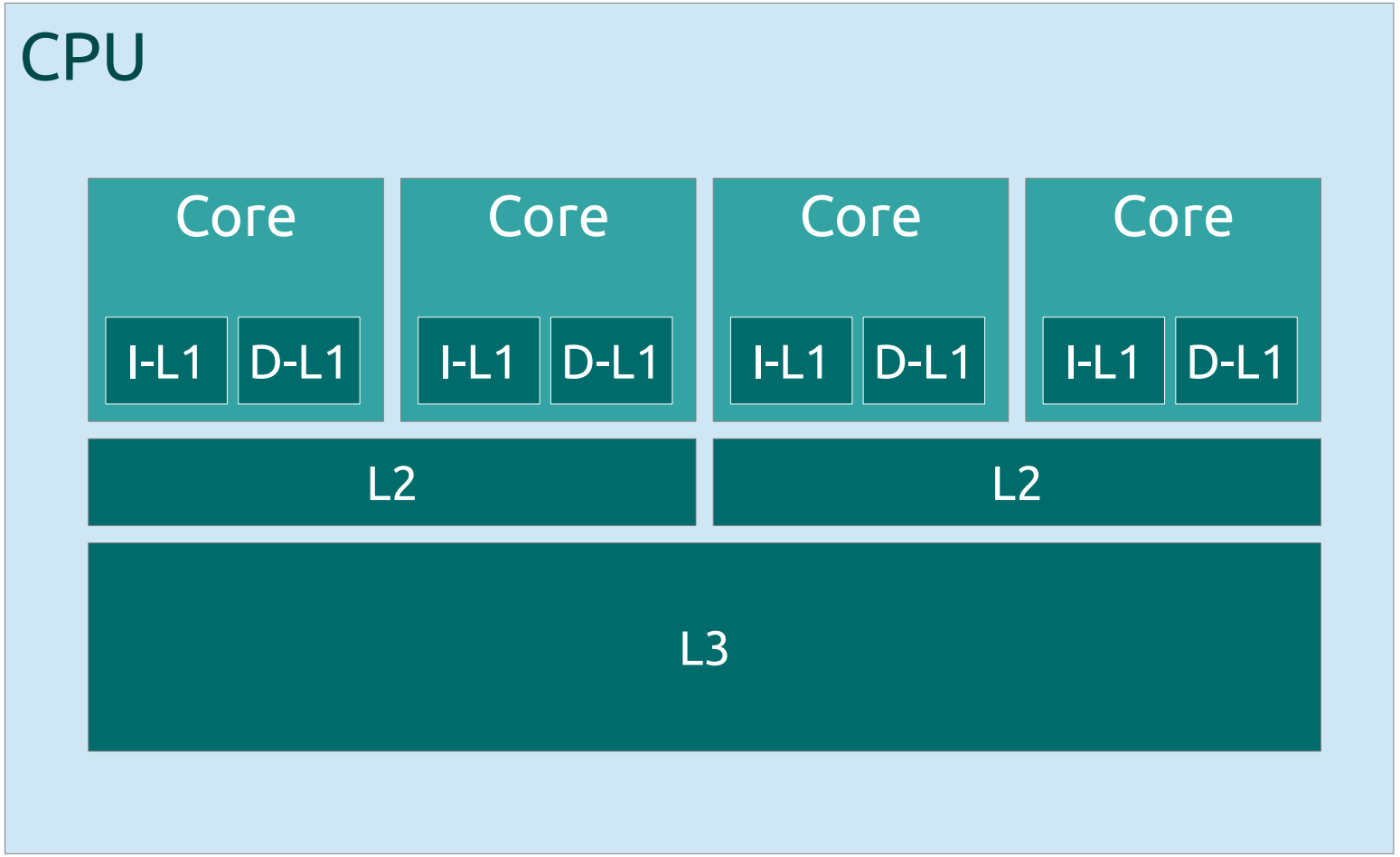
Indicatori di performance

```
$ perf stat -d ./a.out

 Performance counter stats for './a.out':

      1992.419970 task-clock (msec)         #    0.999 CPUs utilized
              215 context-switches          #    0.108 K/sec
                5 cpu-migrations            #    0.003 K/sec
              368 page-faults               #    0.185 K/sec
    4,139,391,573 cycles                    #    2.078 GHz                      [44.54%]
    2,207,522,429 stalled-cycles-frontend   #   53.33% frontend cycles idle     [44.67%]
    2,182,353,973 stalled-cycles-backend    #   52.72% backend  cycles idle     [44.79%]
    2,302,995,011 instructions              #    0.56  insns per cycle
                                            #    0.96  stalled cycles per insn [55.85%]
      657,124,588 branches                  #  329.812 M/sec                    [55.85%]
      155,694,678 branch-misses             #   23.69% of all branches          [55.67%]
      329,767,307 L1-dcache-loads           #  165.511 M/sec                    [55.45%]
       20,723,322 L1-dcache-load-misses     #    6.28% of all L1-dcache hits    [55.25%]
          200,896 LLC-loads                 #    0.101 M/sec                    [44.15%]
   <not supported> LLC-load-misses:HG

      1.994174760 seconds time elapsed
```

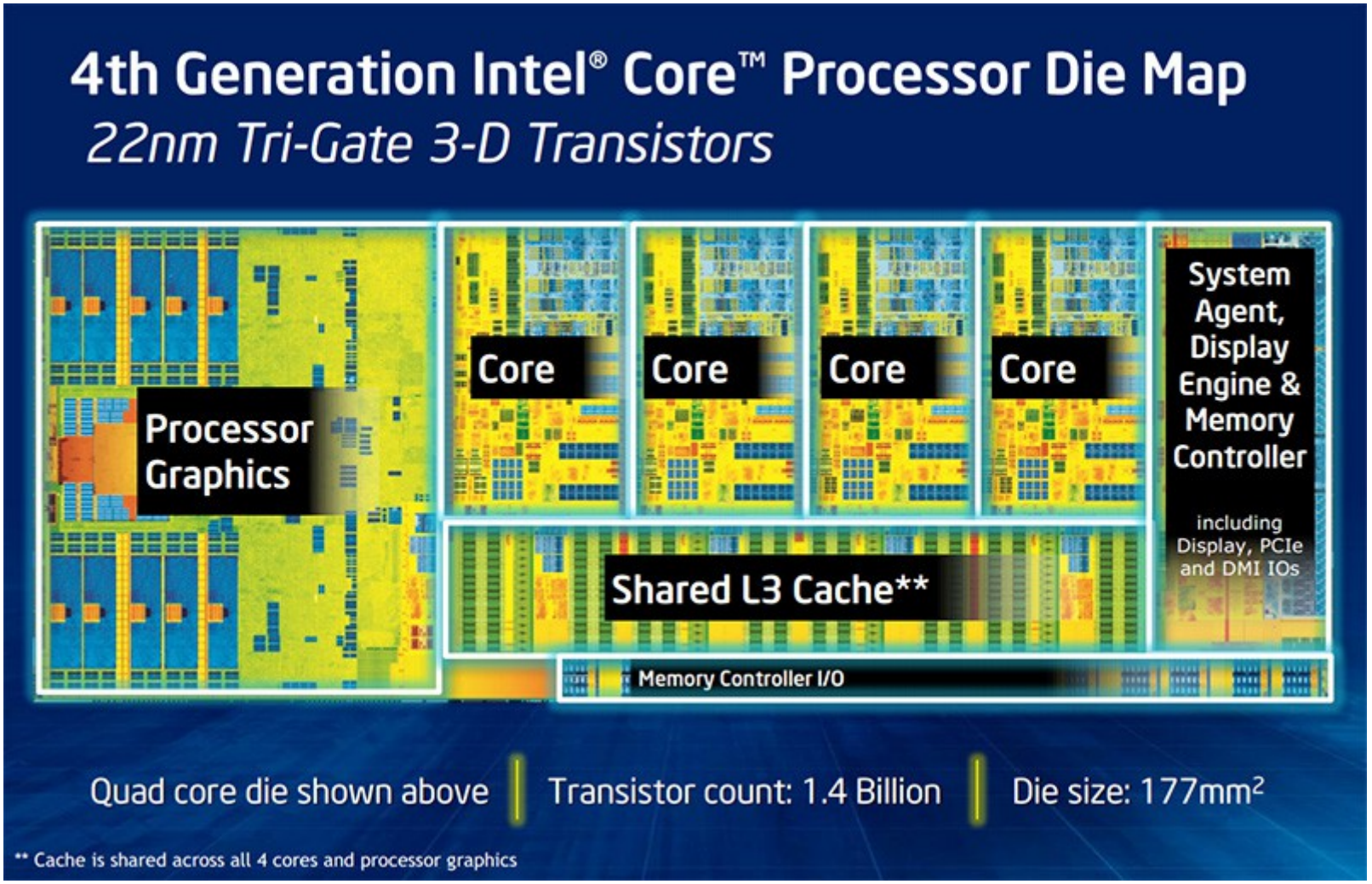
Cache a più livelli

- Per ridurre miss rate e miss penalty si può usare una cache a più livelli (2 o anche 3)
- Il principio è lo stesso: se un dato è disponibile nel livello di cache superiore (hit) lo si usa, altrimenti (miss) si cerca nel livello inferiore
  - il dato esiste sicuramente nel livello più basso
- Man mano che ci si allontana dalla CPU la cache diventa più capiente ma più lenta a rispondere
- Scopi diversi:
  - cache L1 progettata per minimizzare il tempo di accesso in caso di hit
  - cache L2 progettata per minimizzare il miss rate e quindi gli accessi in memoria

Cache a più livelli

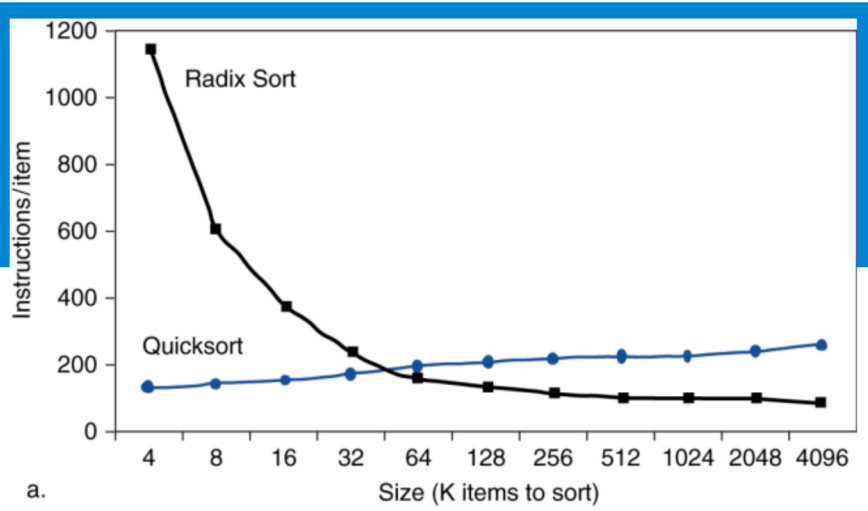


Cache a più livelli

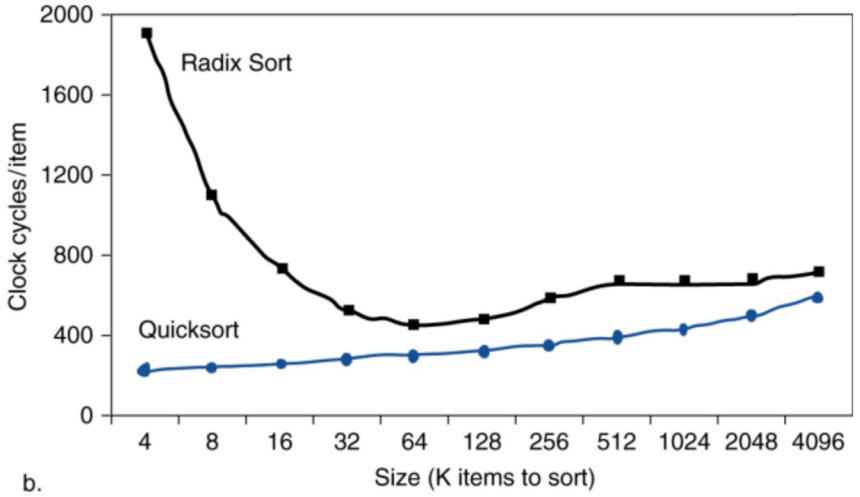


Effetto cache

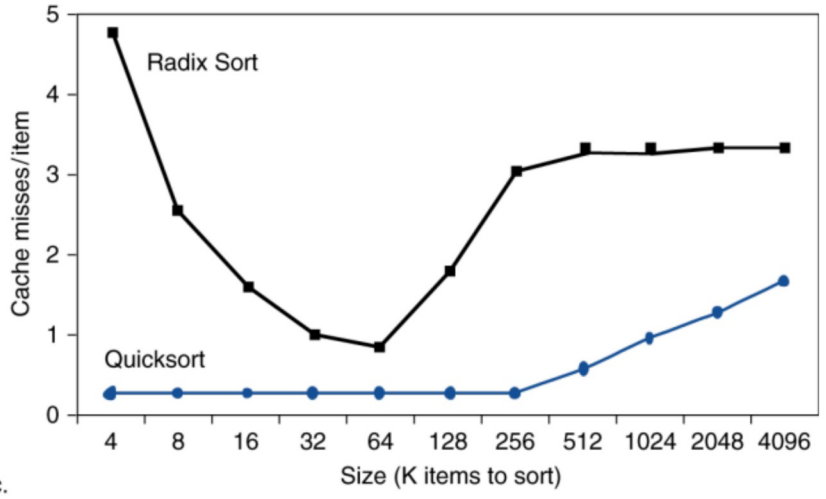
- L'efficienza di un programma non dipende solo dalla complessità computazionale dell'algoritmo...



a.



b.



c.

ARM A8 vs Intel Nehalem

Characteristic	ARM Cortex-A8	Intel Nehalem
L1 cache organization	Split instruction and data caches	Split instruction and data caches
L1 cache size	32 KiB each for instructions/data	32 KiB each for instructions/data per core
L1 cache associativity	4-way (I), 4-way (D) set associative	4-way (I), 8-way (D) set associative
L1 replacement	Random	Approximated LRU
L1 block size	64 bytes	64 bytes
L1 write policy	Write-back, Write-allocate(?)	Write-back, No-write-allocate
L1 hit time (load-use)	1 clock cycle	4 clock cycles, pipelined
L2 cache organization	Unified (instruction and data)	Unified (instruction and data) per core
L2 cache size	128 KiB to 1 MiB	256 KiB (0.25 MiB)
L2 cache associativity	8-way set associative	8-way set associative
L2 replacement	Random(?)	Approximated LRU
L2 block size	64 bytes	64 bytes
L2 write policy	Write-back, Write-allocate (?)	Write-back, Write-allocate
L2 hit time	11 clock cycles	10 clock cycles
L3 cache organization	–	Unified (instruction and data)
L3 cache size	–	8 MiB, shared
L3 cache associativity	–	16-way set associative
L3 replacement	–	Approximated LRU
L3 block size	–	64 bytes
L3 write policy	–	Write-back, Write-allocate
L3 hit time	–	35 clock cycles

Non-uniform memory access