# Architettura degli elaboratori

UniVR - Dipartimento di Informatica

**Fabio Irimie**

2° Semestre 2023/2024

# Indice

# 1 Laboratorio

## 1.1 Vantaggi e svantaggi di assembly

### 1.1.1 Vantaggi

Siccome assembly è un linguaggio di basso livello, è molto vicino all'hardware e quindi è possibile:

- accedere direttamente ai **registri** della CPU
- scrivere **codice ottimizzato** per una specifica architettura di CPU
- ottimizzare le **sezioni "critiche"** dei programmi

### 1.1.2 Svantaggi

I principali svantaggi sono:

- possono essere richieste **molte più righe** di codice
- è facile introdurre dei **bug** perchè la programmazione è più complessa
- il **debugging** è complesso
- **non è garantita la compatibilità** del codice per altri hardware

## 1.2 Utilità

Assembly permette di gestire direttamente il funzionamento della CPU, di conseguenza, i programmi Assembly, una volta compilati sono tipicamente più veloci e più piccoli dei programmi scritti in linguaggi di alto livello. Per questo motivo, l'assembly è utilizzato per scrivere codice che deve essere il più veloce possibile, come ad esempio i driver di hardware specifici.

## 1.3 Registri

Tutti i processori della famiglia x86 hanno i seguenti registri: AX, BX, CX, DX, CS, DS, ES, SS, SP, BP, SI, DI, IP, FLAGS.

Originariamente i registri AX, BX, CX, DX, SP, BP, SI, DI, IP e FLAGS avevano una dimensione di 16 bit. A partire dal 80386, la loro dimensione è stata estesa a 32 bit e al loro nome è stato aggiunto il prefisso E (Extended). Per ragioni di retrocompatibilità, i registri di 16 bit possono essere utilizzati anche nei processori a 32 bit utilizzando il loro nome originale.

### 1.3.1 Registri general purpose

I seguenti registri sono generici, pertanto è possibile assegnargli qualunque valore. Tuttavia, durante l'esecuzione di alcune istruzioni i registri generici vengono utilizzati per memorizzare valori ben determinati:

- **EAX** (Accumulator register): è usato come accumulatore per operazioni aritmetiche e contiene il risultato dell'operazione

- **EBX** (Base register): è usato per operazioni di indirizzamento della memoria
- **ECX** (Counter register): è usato per "contare", ad esempio nelle operazioni di loop
- **EDX** (Data register): è usato nelle operazioni di input/output, nelle divisioni e nelle moltiplicazioni

### 1.3.2 Registri di segmento

CS, DS, ES e SS sono i **registri di segmento** (segment registers) e devono essere utilizzati con cautela:

- **CS** (Code Segment): punta alla zona di memoria che contiene il codice. Durante l'esecuzione del programma, assieme al registro IP, serve per accedere alla prossima istruzione da eseguire (attenzione: non può essere modificato!)
- **DS** (Data Segment): punta alla zona di memoria che contiene i dati
- **ES** (Extra Segment): può essere usato come registro di segmento ausiliario
- **SS** (Stack Segment): punta alla zona di memoria in cui risiede lo stack

### 1.3.3 Registri puntatore

ESP, EBP, EIP sono i registri puntatore (pointer registers):

- **ESP** (Stack Pointer): punta alla cima dello stack. Viene modificato dalle operazioni di PUSH (inserimento di un dato nello stack) e POP (estrazioni di un dato dallo stack). Si ricordi che lo stack è una struttura di tipo LIFO (Last In First Out - l'ultimo che entra è il primo che esce). È possibile modificarlo manualmente ma occorre cautela!
- **EBP** (Base Pointer): punta alla base della porzione di stack gestita in quel punto del codice. È possibile modificarlo manualmente ma occorre cautela!
- **EIP** (Instruction Pointer): punta alla prossima istruzione da eseguire. Non può essere modificato!

### 1.3.4 Registri indice

ESI e EDI sono i registri indice (index registers) e vengono utilizzati per operazioni con stringhe e vettori:

- **ESI** (Source Index): punta alla stringa/vettore sorgente
- **EDI** (Destination Index): punta alla stringa/vettore destinazione
- **EFLAGS**: è utilizzato per memorizzare lo stato corrente del processore. Ciascuna flag (bit) del registro fornisce una particolare informazione. Ad esempio, la flag in prima posizione (carry flag) viene posta a 1 quando c;è stato un riporto o un prestito durante un'operazione aritmetica; la flag

in seconda posizione (parity flag) viene usata come bit di parità e viene posta a 1 quando il risultato dell'ultima operazione ha un numero pari di 1

### 1.3.5 Composizione dei registri

I registri sono composti da 32 bit e possono essere divisi in registri più piccoli:

- **EAX**: AX, AH, AL
- **EBX**: BX, BH, BL
- **ECX**: CX, CH, CL
- **EDX**: DX, DH, DL
- **ESP**: SP
- **EBP**: BP
- **ESI**: SI
- **EDI**: DI

### 1.3.6 Composizione del registro EFLAGS

Il registro EFLAGS è composto da 32 bit e ogni bit corrisponde ad un flag:

- **ZF** (Zero flag): impostato a 1 se il risultato dell'operazione è 0
- **SF** (Sign flag): impostato a 1 se il risultato dell'operazione è un numero negativo, a 0 se è positivo (rappresentazione in complemento a 2)
- **OF** (Overflow flag): impostato a 1 nel caso di overflow di un'operazione
- **TF** (Trap flag): impostato a 1 genera un'interruzione ad ogni istruzione. Utilizzato per l'esecuzione passo-passo dei programmi
- **IF** (Interrupt flag): impostato a 1 abilita gli interrupt esterni, con 0 li disabilita
- **DF** (Direction flag): impostato a 1 indica che nelle operazioni di spostamento di stringhe i registri DI e SI si autodecrementano (con 0 tali registri si auto incrementano)

## 1.4 Modalità di indirizzamento

Si rifersce al modo in cui un'istruzione assembly accede ai dati in memoria e può essere:

- **Indirizzamento a registro**: l'operando è contenuto in un registro ed il nome del registro è specificato nell'istruzione. Ad esempio:

```
%Ri
```

- **Indirizzamento diretto** (o assoluto): l'operando è contenuto in una locazione di memoria, e l'indirizzo della locazione viene specificato nell'istruzione. Ad esempio:

  `(IND)`

- **Indirizzamento immediato** (o di costante): l'operando è un valore costante ed è definito esplicitamente nell'istruzione. Ad esempio:

  `$VAL`

- **Indirizzamento indiretto**: l'indirizzo di un operando è contenuto in un registro o in una locazione di memoria. L'indirizzo della locazione o il registro viene specificato nell'istruzione. Ad esempio:

  `(%Ri)` o `($VAL)`

- **Indirizzamento indicizzato** (Base e spiazzamento): l'indirizzo effettivo dell'operando è calcolato sommando un valore costante al contenuto di un registro. Ad esempio:

  `SPI(%Ri)`

- **Indirizzamento con autoincremento**: l'indirizzo effettivo dell'operando è il contenuto di un registro specificato nell'istruzione. Dopo l'accesso, il contenuto del registro viene incrementato per puntare all'elemento successivo

- **Indirizzamento con autodecremento**: il contenuto di un registro specificato nell'istruzione viene decrementato. Il nuovo contenuto viene usato come indirizzo effettivo dell'operando

## 1.5 Istruzioni

### 1.5.1 Istruzioni di inizializzazione

- `mov src, dst` **Move**: consente l'inizializzazione di un registro o di un'area di memoria. Accetta i modificatori `l`, `w` e `b` per indicare la dimensione dell'operando `s rc`

- `lea src, dst` **Load Effective Address**: trasferisce l'indirizzo effettivo dell'operando `src` nel registro `dst`

### 1.5.2 Istruzioni aritmetiche e logiche

- `sar op1, op2` **Shift Arithmetic Right**: esegue lo shift a destra sul registro `op2` di tanti bit quanti specificati in `op1`. Il bit più significativo viene replicato (così da funzionare anche con numeri negativi in complemento 2) e il bit scartato viene messo nel **Carry Flag**. `op1` può essere un registro o un valore immediato, `op2` deve essere un registro

- `sal op1, op2` **Shift Arithmetic Left**: esegue lo shift a sinistra sul registro `op2` di tanti bit quanti specificati in `op1`. Il bit meno significativo viene messo a 0 e il bit scartato viene messo nel **Carry Flag**. `op1` può essere un registro o un valore immediato, `op2` deve essere un registro

- `inc op` **Increment**: incrementa di 1 il valore memorizzato in `op`. `op` può essere un registro o una locazione di memoria

- `dec op` **Decrement**: decrementa di 1 il valore memorizzato in `op`. `op` può essere un registro o una locazione di memoria

- `add src, dst` **Add**: somma a `dst` il valore di `src` e memorizza il risultato in `dst`

- `sub src, dst` **Subtract**: sottrae da `dst` il valore di `src` e memorizza il risultato in `dst`

- `mul moltipl` **Unsigned Multiplication**: esegue la moltiplicazione senza segno. `moltipl` deve essere un registro o una variabile. Se `moltipl` è un byte il registro `AL` viene moltiplicato per l'operando e il risultato viene memorizzato in `AX`. Se `moltipl` è una word il contenuto del registro `AX` viene moltiplicato per l'operando e il risultato viene memorizzato nella coppia di registri `DX:AX` (`DX` conterrà i 16 bit più significativi del risultato). Se `moltipl` è un long il contenuto del registro `EAX` viene moltiplicato per l'operando e il risultato viene memorizzato nella coppia di registri `EDX:EAX` (`EDX` conterrà i 32 bit più significativi del risultato).

- `imul moltipl` moltiplicazione con segno

- `div divisore` **Unsigned Division**: esegue la divisione senza segno. `divisore` deve essere un registro o una variabile. Se `divisore` è un byte il registro `AX` viene diviso per l'operando, il quoziente viene memorizzato in `AL`, e il resto in `AH`. Se `divisore` è una word, il valore ottenuto concatenando il contenuto di `DX` e `AX` viene diviso per l'operando (i 16 bit più significativi del dividendo devono essere memorizzati nel registro `DX`), il quoziente viene memorizzato nel registro `AX` e il resto in `DX`. Se `divisore` è un long, il valore ottenuto concatenando il contenuto di `EDX` e `EAX` viene diviso per l'operando (i 32 bit più significativi del dividendo sono nel registro `EDX`), il quoziente viene memorizzato nel registro `EAX` e il resto in `EDX`

- `xor src, dst` **Logical Exclusive OR**: calcola l'OR esclusivo bit a bit dei due operandi e lo memorizza nell'operando `dst`. Spesso si utilizza per azzerare un registro, utilizzandolo sia come `src` che come `dst`)

- `or src, dst` **Logical OR**: calcola l'OR logico bit a bit dei due operandi e lo memorizza nell'operando `dst`

- `and src, dst` **Logical AND**: calcola l'AND bit a bit dei due operandi e lo memorizza nell'operando `dst`

- `not op` **Logical NOT**: inverte ogni singolo bit dell'operando op

## 1.6 AT&T vs Intel

Le principali differenze tra la sintassi AT&T e Intel sono:

- In AT&T i nomi dei registri hanno il carattere `%` come prefisso
- In AT&T l'ordine degli operandi è `<sorgente>, <destinazione>`, opposto rispetto alla sintassi Intel
- In AT&T la lunghezza dell'operando è specificata tramite un suffisso al nome dell'istruzione. `b` per **byte** (8 bit), `w` per **word** (16 bit), `l` per **double word** (32 bit)
- Gli operandi immediati in AT&T sono preceduti dal simbolo `$`
- la presenza di prefisso in un operando indica che si tratta di un indirizzo di memoria. Ad esempio:

  `movl $pippo, %eax` è diverso da `movl pippo, %eax`

- l'indicizzazione o l'indirezione è ottenuta racchiudendo tra parentesi l'indirizzo di base espresso tramite un registro o un valore immediato. Ad esempio:

  ```
  movl 5, 17(%ebp)
  ```

## 1.7 Assemblare, verificare ed eseguire un programma Assembly

Il processo di creazione di un programma Assembly passa attraverso le seguenti fasi:

1. Scrittura di uno o più file ASCII (con estensione `.s`) contenenti il programma sorgente, tramite un normale editor di testo.
2. Assemblaggio dei file sorgenti, e generazione dei file **oggetto** (con estensione `.o`), tramite un **assemblatore**.
3. Creazione del file **eseguibile**, tramite un **linker**
4. Verifica del funzionamento e correzione degli eventuali errori tramite un **debugger**

### 1.7.1 L'assemblatore

L'Assemblatore trasforma i file contenenti il programma sorgente in altrettanti file oggetto contenenti il codice in linguaggio macchina. Durante questo corso verrà usato l'assemblatore **gas** della GNU.
Per assemblare un file è necessario eseguire il seguente comando:

```
$ as -o <nomefile>.o <nomefile>.s
```

### 1.7.2 Il linker

Il linker combina i moduli oggetto e produce un unico file eseguibile. In particolare unisce i moduli oggetto, risolvendo i riferimenti a simboli esterni; ricerca i file di libreria contenenti le procedure esterne utilizzate dai vari moduli e produce un file eseguibile. L'operazione di linking deve essere effettuata anche se il programma è composto da un solo modulo oggetto.
Durante il corso verrà usato il linker **ld** della GNU.
Per creare l'eseguibile a partire da un file oggetto è necessario eseguire il seguente comando:

```
$  ld -o <nomefile>.x <nomefile1>.o <nomefile2>.o ...
```

### 1.7.3 Assembly 32bit su macchine 64bit

La gran parte del codice ASM32 è compatibile con macchine a 64bit, tuttavia alcune estensioni in particolari istruzioni non sono riconosciute dai compilatori. È possibile utilizzare codice ASM32 su architetture a 64bit utilizzando dei flag di compilazione che permettono di simulare il comportamento di una architettura a 32bit.

```
$  as --32 -o <nomefile>.o <nomefile>.s
$  ld -m elf_i386 -o <nomefile> <nomefile>.o
```

## 1.8 Stampa di numeri

I numeri sono memorizzati nei registri e nelle variabili come interi in complemento a 2 su 32 bit. Affinchè essi possano essere stampati a video occorre trasformarli in stringhe di caratteri cioè vettori di byte dove ciascun byte rappresenta un carattere secondo la codifica ASCII.
Per trasformare un numero intero in una stringa occorre scomporlo nelle sue cifre mediante divisioni successive per 10. Per la particolare conformazione della tabella ASCII il codice del carattere corrispondente alla cifra $n$ si ottiene come $n + 48$.

### 1.8.1 Tabella dei caratteri ASCII

La tabella seguente è relativa al codice US ASCII, ANSI X3.4-1986 (ISO International Reference Version). I codici decimali da 0 a 31 e il 127 sono caratteri non stampabili (codici di controllo). Il 32 corrispondente al carattere "spazio". I codici dal 32 al 126 sono caratteri stampabili.

| Char | Dec | Nome | Descrizione |
|---|---|---|---|
| | 0 | NUL (Ctrl-@) | NULL |
| | 1 | SOH (Ctrl-A) | Start of Heading |
| | 2 | STX (Ctrl-B) | Start of Text |
| | 3 | ETX (Ctrl-C) | End of Text |
| | 4 | EOT (Ctrl-D) | End of Transmission |
| | 5 | ENQ (Ctrl-E) | Enquiry |

| | | | |
|---|---|---|---|
| | 6 | ACK (Ctrl-F) | Acknowledge |
| | 7 | BEL (Ctrl-G) | Bell (Beep) |
| | 8 | BS (Ctrl-H) | Backspace |
| | 9 | HT (Ctrl-I) | Horizontal Tab |
| | 10 | LF (Ctrl-J) | Line Feed |
| | 11 | VT (Ctrl-K) | Vertical Tab |
| | 12 | FF (Ctrl-L) | Form Feed |
| | 13 | CR (Ctrl-M) | Carriage Return |
| | 14 | SO (Ctrl-N) | Shift Out |
| | 15 | SI (Ctrl-O) | Shift In |
| | 16 | DLE (Ctrl-P) | Data Link Escape |
| | 17 | DC1 (Ctrl-Q) | Device Control 1 (XON) |
| | 18 | DC2 (Ctrl-R) | Device Control 2 |
| | 19 | DC3 (Ctrl-S) | Device Control 3 (XOFF) |
| | 20 | DC4 (Ctrl-T) | Device Control 4 |
| | 21 | NAK (Ctrl-U) | Negative Acknowledge |
| | 22 | SYN (Ctrl-V) | Synchronous Idle |
| | 23 | ETB (Ctrl-W) | End of Transmission Block |
| | 24 | CAN (Ctrl-X) | Cancel |
| | 25 | EM (Ctrl-Y) | End of Medium |
| | 26 | SUB (Ctrl-Z) | Substitute |
| | 27 | ESC (Ctrl-[) | Escape |
| | 28 | FS (Ctrl-\) | File Separator |
| | 29 | GS (Ctrl-]) | Group Separator |
| | 30 | RS (Ctrl-^) | Record Separator |
| | 31 | US (Ctrl-_) | Unit Separator |
| | 32 | | Space |
| ! | 33 | | Exclamation mark |
| " | 34 | | Quotation mark |
| # | 35 | | Number sign |
| $ | 36 | | Dollar sign |
| % | 37 | | Percent sign |
| & | 38 | | Ampersand |
| ’ | 39 | | Apostrophe |
| ( | 40 | | Left parenthesis |
| ) | 41 | | Right parenthesis |
| * | 42 | | Asterisk |
| + | 43 | | Plus sign |
| , | 44 | | Comma |
| - | 45 | | Hyphen |
| . | 46 | | Period, dot |
| / | 47 | | Slash |
| 0 | 48 | | Zero |
| 1 | 49 | | One |
| 2 | 50 | | Two |
| 3 | 51 | | Three |
| 4 | 52 | | Four |

| | | | |
|---|---|---|---|
| 5 | 53 | | Five |
| 6 | 54 | | Six |
| 7 | 55 | | Seven |
| 8 | 56 | | Eight |
| 9 | 57 | | Nine |
| : | 58 | | Colon |
| ; | 59 | | Semicolon |
| < | 60 | | Less-than sign |
| = | 61 | | Equal sign |
| > | 62 | | Greater-than sign |
| ? | 63 | | Question mark |
| @ | 64 | | At sign |
| A | 65 | | Uppercase A |
| B | 66 | | Uppercase B |
| C | 67 | | Uppercase C |
| D | 68 | | Uppercase D |
| E | 69 | | Uppercase E |
| F | 70 | | Uppercase F |
| G | 71 | | Uppercase G |
| H | 72 | | Uppercase H |
| I | 73 | | Uppercase I |
| J | 74 | | Uppercase J |
| K | 75 | | Uppercase K |
| L | 76 | | Uppercase L |
| M | 77 | | Uppercase M |
| N | 78 | | Uppercase N |
| O | 79 | | Uppercase O |
| P | 80 | | Uppercase P |
| Q | 81 | | Uppercase Q |
| R | 82 | | Uppercase R |
| S | 83 | | Uppercase S |
| T | 84 | | Uppercase T |
| U | 85 | | Uppercase U |
| V | 86 | | Uppercase V |
| W | 87 | | Uppercase W |
| X | 88 | | Uppercase X |
| Y | 89 | | Uppercase Y |
| Z | 90 | | Uppercase Z |
| [ | 91 | | Left square bracket |
| \ | 92 | | Backslash |
| ] | 93 | | Right square bracket |
| ^ | 94 | | Circumflex accent |
| _ | 95 | | Underscore |
| ‘ | 96 | | Grave accent |
| a | 97 | | Lowercase a |
| b | 98 | | Lowercase b |
| c | 99 | | Lowercase c |

| | | | |
|---|---|---|---|
| d | 100 | | Lowercase d |
| e | 101 | | Lowercase e |
| f | 102 | | Lowercase f |
| g | 103 | | Lowercase g |
| h | 104 | | Lowercase h |
| i | 105 | | Lowercase i |
| j | 106 | | Lowercase j |
| k | 107 | | Lowercase k |
| l | 108 | | Lowercase l |
| m | 109 | | Lowercase m |
| n | 110 | | Lowercase n |
| o | 111 | | Lowercase o |
| p | 112 | | Lowercase p |
| q | 113 | | Lowercase q |
| r | 114 | | Lowercase r |
| s | 115 | | Lowercase s |
| t | 116 | | Lowercase t |
| u | 117 | | Lowercase u |
| v | 118 | | Lowercase v |
| w | 119 | | Lowercase w |
| x | 120 | | Lowercase x |
| y | 121 | | Lowercase y |
| z | 122 | | Lowercase z |
| { | 123 | | Left curly brace |
| \| | 124 | | Vertical bar |
| } | 125 | | Right curly brace |
| ~ | 126 | | Tilde |
| | 127 | DEL (Ctrl-?) | Delete |

## 1.9 Etichette ed istruzioni di salto

In Assembly non esiste il costrutto `if` ... `then` ... `else` e quindi le istruzioni di salto servono per far saltare l'esecuzione del programma ad una certa istruzione in funzione del valore di una condizione. Le uniche condizioni che si possono valutare sono $<, =, >$ tra due valori numerici e la presenza di zero nel registro ECX. In particolare, la valutazione di una condizione di $<, =, >$ consiste di due istruzioni: la prima sottrae tra loro i due valori numerici e imposta i bit SF e ZF del registro EFLAGS, la seconda effettua il salto in base al valore di tali flags. Le etichette sono essenziali per le istruzioni di salto in quanto indicano a quale punto della sequenza di istruzioni bisogna saltare. Occorre inserire prima dell'istruzione a cui si vuole saltare un nome simbolico seguito dal carattere "`:`". Ad esempio:

```
etichetta:
  istruzioni
  ...
```

È importante che il nome dell'etichetta sia unico in tutto il programma. Anche in questo caso, come per i nomi delle variabili, l'assemblatore trasforma i nomi delle etichette in numeri binari (che in questo caso indicano l'indirizzo dell'istruzione che segue) a meno che non si voglia conservarli per il debug (con l'opzione `-gstabs`).
In Assembly non esistono istruzioni ad alto livello per realizzare i cicli come `for` ..., `while` ...; essi si devono costruire manualmente a partire dalle istruzioni di salto condizionato. Se si vuole eseguire un ciclo per un certo numero di volte occorre utilizzare ECX come contatore.

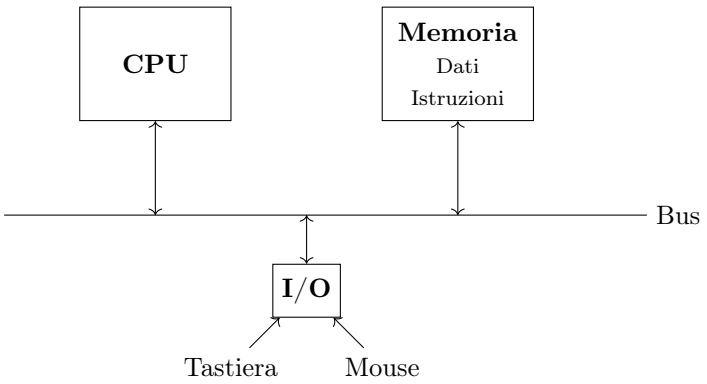
# 2 Architettura di Von Neumann

L'esigenza era quella di avere un'architettura che permettesse di eseguire programmi in modo automatico, senza dover cambiare il cablaggio del calcolatore, quindi il circuito deve essere abbastanza generale per poter eseguire programmi diversi.

## 2.1 Struttura

L'architettura di Von Neumann è composta da 5 parti principali:

- **Unità aritmetico-logica**: si occupa di eseguire le operazioni aritmetiche e logiche
- **Unità di controllo**: si occupa di controllare il flusso delle istruzioni
- **Memoria**: contiene i dati e le istruzioni
- **Input/Output**: permette di comunicare con l'esterno
- **Bus**: permette di trasferire i dati tra la memoria e l'unità aritmetico-logica (generalmente in oro)




## 2.2 Caratteristiche

Le istruzioni hanno bisogno di un'operazione che permetta di effettuare dei salti, in modo da poter implementare i cicli e le strutture di controllo. Inoltre le istruzioni devono essere eseugite in sequenza (un'istruzione alla volta).

## 2.3 CPU

Ogni processore ha un set di istruzioni diverso in base all'architettura, questo set di istruzioni è chiamato **ISA** (Instruction Set Architecture). **Assembly** è un linguaggio di programmazione che permette di scrivere programmi in base all'ISA del processore e questo linguaggio viene tradotto in linguaggio binario attraverso un **assembler**. In questo corso viene usata l'architettura x86 (80x86).

### 2.3.1 Modello semplificato

Per rappresentare il funzionamento di un processore si può usare un modello semplificato rappresentato ad alto livello. Questo modello è composto da:

- **Central Processing Unit (CPU)**: esegue le istruzioni
- **Control Unit (CU)**: controlla il flusso delle istruzioni
- **Bus Dati (BD)**: trasferisce i dati alla CPU
- **Bus Istruzioni (BI)**: trasferisce le istruzioni alla CPU
- **Bus di Controllo (BC)**: trasferisce i segnali di controllo
- **Memory Address Register (MAR)**: contiene l'indirizzo di memoria da leggere
- **Memory Data Register (MDR)**: contiene i dati letti dalla memoria
- **Program Counter (PC)**: tiene conto dell'indirizzo dell'istruzione da eseguire
- **Instruction Register (IR)**: contiene l'istruzione corrente
- **Program Status Word (PSW)**: contiene i flag del processore (es. zero, carry, overflow). È come se fosse un array in cui ad ogni indice corrisponde un flag per ogni operazione.
- **Register File**: contiene i registri del processore (es. EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP)
- **Arithmetic Logic Unit (ALU)**: esegue le operazioni aritmetiche e logiche

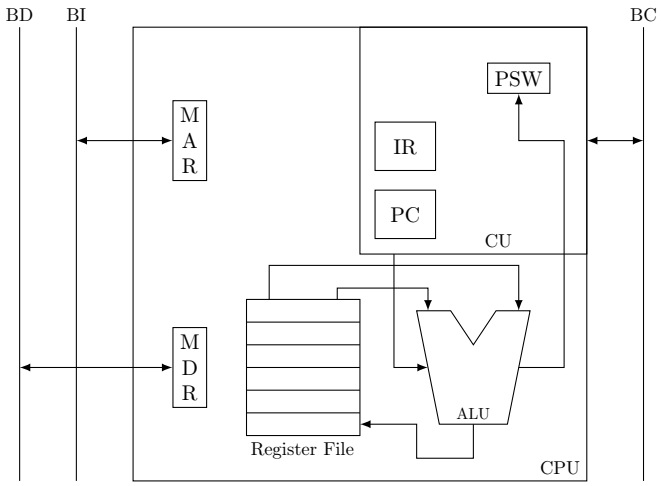



Figura 1: Struttura di un processore

Il flusso di esecuzione delle istruzioni è il seguente:

- **Fetch**: CU legge l'istruzione dalla memoria
- **Decode**: CU decodifica l'istruzione
- **Execute**: ALU esegue l'istruzione

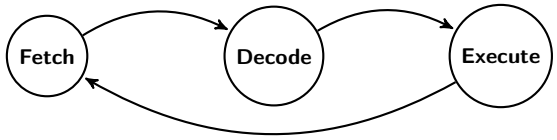



Figura 2: Flusso di esecuzione delle istruzioni

## 2.4 Modello concreto (LC-3)

L'LC-3 è un processore Turing complete, ovvero può eseguire qualsiasi programma. È un processore che nasce per motivi didattici, molto semplice e permette di capire come funziona un processore. L'LC-3 è composto da:

- **Memoria**:
  - **MAR**: Memory Address Register, contiene l'indirizzo di memoria da leggere
  - **MDR**: Memory Data Register, contiene i dati letti dalla memoria
- **Processing Unit**:
  - **ALU**
  - **Registri**: 8 registri da 16 bit

- **Input**:
  - **Tastiera**
  - **Mouse**
  - **Scanner**
  - **Disco**
- **Output**:
  - **Monitor**
  - **Stampante**
  - **LED**
  - **Disco**

  L'imput e l'output non vengono gestiti direttamente dai dispositivi, ma vengono gestiti dalla memoria.
- **Control Unit**:
  - **PC**: Program Counter, contiene l'indirizzo dell'istruzione da eseguire
  - **IR**: Instruction Register, contiene l'istruzione corrente

### 2.4.1 Memoria

$2^16 \times 16$ celle di memoria. Per interfacciarsi con la memoria si usa la `LOAD` e la `STORE`.

I registri sono 8 e vanno da $R0, \ldots, R7$ ognuno da 16 bit. La dimensione della parola del processore è di 16 bit. Anche le istruzioni sono di 16 bit.

### 2.4.2 Processing Unit

Questa unità può eseguire 3 operazioni:

- **ADD**
- **AND**
- **NOT**

### 2.4.3 Input e Output

Ci sono 2 tipi di periferiche:

- **A caratteri**: tastiera, mouse, scanner. Sono periferiche che comunicano con il processore inviando un carattere alla volta.
- **A blocchi**: monitor, stampante, disco. Sono periferiche che comunicano con il processore inviando un blocco di dati alla volta.

L'input e l'output viene gestito direttamente dalla CPU, quindi non c'è nessun componente esterno che lo fa.

- Per gestire la tastiera ci sono 2 registri:
  - **KBDR**: Keyboard Data Register, contiene il carattere letto dalla tastiera
  - **KBSR**: Keyboard Status Register, contiene lo stato della tastiera
- Per gestire il monitor ci sono 2 registri:
  - **DDR**: Display Data Register, contiene il carattere da scrivere sul monitor
  - **DSR**: Display Status Register, contiene lo stato del monitor

### 2.4.4 Processazione delle istruzioni

Il loop di esecuzione delle istruzioni è il seguente:

- **Fetch**: legge l'istruzione dalla memoria
- **Decode**: decodifica l'istruzione
- **Evaluate address**: calcola l'indirizzo dell'operando
- **Fetch operands**: legge gli operandi dalla memoria
- **Execute**: esegue l'istruzione
- **Store result**: scrive il risultato nella memoria

### 2.4.5 Istruzioni

Le istruzioni sono di 16 bit e sono specificate nel seguente modo:

- **opcode**: specifica l'operazione da eseguire
- **operands**: dati o indirizzi su cui operare

Ogni istruzione è codificata come una sequenza di bit ed è il componente più piccolo e non interrompibile del sistema.

La specifica di come sono fatte le istruzioni è chiamata **ISA** (Instruction Set Architecture).

### 2.4.6 Operazioni

- **Operazioni di calcolo**:
  - **ADD**: somma due numeri
  - **AND**: effettua l'AND bit a bit tra due numeri
  - **NOT**: effettua il NOT bit a bit di un numero
- **Operazioni di movimento dei dati**:
  - **LD**
  - **LDI**

  - **LDR**
  - **LEA**
  - **ST**
  - **STR**
  - **STI**

- **Operazioni di controllo**:

  Sono gestite da 3 flag:

  - **N**: negativo
  - **Z**: zero
  - **P**: positivo

  Le operazioni sono:

  - **BR**
  - **JMP**
  - **JSR**
  - **JSRR**
  - **RTI**
  - **TRAP**

I tipi di dato sono gestiti come 16 bit codificati in complemento a 2.

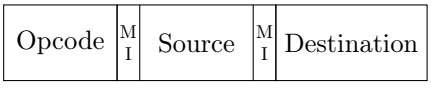
#### 2.4.7 Metodi di indirizzamento

- **Immediate**: l'operando è un valore costante
- **Register**: l'operando è un registro
- **PC-relative**: l'operando è un offset rispetto al PC
- **Indirect**: l'operando è un indirizzo in memoria
- **Base+offset**: l'operando è un indirizzo in memoria calcolato come somma di un registro e un offset

# 3 Assembly (Intel x86)

## 3.1 Codifica

Ogni istruzione è codificata nel seguente modo:




dove **MI** indica il metodo di indirizzamento.

## 3.2 Istruzioni

### 3.2.1 Istruzioni di inizializzazione

- `MOVL <source> <destination>`: copia il contenuto di un registro (o costante) in un altro. Questa istruzione di solito viene utilizzata per spostare i valori dalla memoria ai registri e viceversa, in modo da poter effettuare operazioni solo su dati presenti nei registri e non direttamente in memoria, questo rende l'esecuzione più efficiente.
- `NOP`: non fa nulla e occupa solo un byte. La sua utilità è quella di "riempire i buchi", cioè delle zone di memoria non occupate da nessuna istruzione.

### 3.2.2 Istruzioni aritmetiche

- `ADDL <source> <destination>`: somma il contenuto di due registri (o costante). Siccome sono disponibili solo 2 parametri, il risultato viene salvato nel secondo parametro perchè viene visto sia come sorgente che destinazione per evitare di aggiungerne un terzo.
- `SUBL <source> <destination>`: sottrae il contenuto di due registri (o costante)
- `MULL <source> <destination>`: moltiplica il contenuto di due registri (o costante)
- `INC <source>`: incrementa il contenuto di un registro (o costante) di 1
- `DEC <source>`: decrementa il contenuto di un registro (o costante) di 1

### 3.2.3 Istruzioni logiche

- `CMPL <source> <destination>`: confronta il contenuto di due registri (o costanti) e modifica il flag `PSW` in base al risultato del confronto.

### 3.2.4 Istruzioni di salto

Se il salto è **assoluto** l'indirizzo fa riferimento alla memoria diretta, mentre se il salto è **relativo** l'indirizzo è relativo al Program Counter.

- `JMP <etichetta>`: salta all'istruzione con etichetta
- `JE <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che i due operandi siano uguali
- `JNE <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che i due operandi siano diversi
- `JG <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che il primo operando sia maggiore del secondo
- `JGE <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che il primo operando sia maggiore o uguale del secondo
- `JL <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che il primo operando sia minore del secondo

- `JLE <etichetta>`: salta all'istruzione con etichetta se i flag sono settati in modo che il primo operando sia minore o uguale del secondo

Comparando e poi utilizzando i salti si possono implementare le strutture di controllo come i cicli e le condizioni. Nell'etichetta si può inserire un'indirizzo di memoria assoluto che permette di saltare a quell'indirizzo, questo però non è molto utile perchè il programma potrebbe essere caricato in un'area diversa della memoria.

### 3.2.5 Istruzioni di gestione dello Stack

- `PUSHL <source>`: inserisce il contenuto di un registro (o costante) nello stack
- `POPL <destination>`: estrae il contenuto dello stack e lo mette in un registro (o costante)
- `CALL <etichetta>`: salva l'indirizzo successivo al Program Counter nello stack e salta all'istruzione con etichetta

### 3.2.6 Metodi di indirizzamento

I metodi di indirizzamento (MI) sono diversi modi per accedere ai dati in memoria, i più comuni sono:

- **Registro**: Un'istruzione può accedere direttamente ai registri ad esempio: `MOVL %EAX, %EBX`
- **Immediato**: Un'istruzione può accedere direttamente ai dati ad esempio: `MOVL $10, %EBX`
- **Assoluto**: Un'istruzione può accedere direttamente ai dati ad esempio: `MOVL DATO, %EBX`
  dove `DATO` è un'etichetta che punta ad un'indirizzo di memoria
- **Indiretto Registro**: Un'istruzione può contenere un registro che punta ad un altro registro ad esempio: `MOVL (%EAX), %EBX`
- **Indiretto Registro con Spiazzamento**: Un'istruzione può mettere un offset rispetto al registro contenuto nell'istruzione ad esempio: `MOVL $8(%EAX), %EBX`

Non tutte le istruzioni ammettono tutti i metodi di indirizzamento e alcuni metodi di indirizzamento possono essere usati solo con alcune istruzioni.

## 3.3 Esempi

Un esempio di codice in C è il seguente:

```
...
int a; // INDA (etichetta che punta ad un indirizzo di memoria con
  valore intero)
int b; // INDB
int c; // INDB
```

```
...
a = 5; // %EAX
b = 10; // %EBX

if (a > b) {
  c = a - b; // %ECX
} else { // ELSE
  c = a + b;
}
```

La traduzione in assembly è la seguente:

```
MOVL INDA, %EAX // Ridondante
MOVL $5, %EAX
MOVL INDB, %EBX // Ridondante
MOVL $10, %EBX
MOVL %EAX, %ECX
COMPL %EAX, %EBX
JLE ELSE
SUBL %ECX, %EAX
JMP ENDIF
ELSE:
ADDL %EBX, %ECX
ENDIF:
```

Un altro esempio di un for loop in C:

```
for (int i = 0; i < 10; i++) { // int i; %EDX
  ...
}
```

La traduzione in assembly è la seguente:

```
MOVL $0, %EDX
FOR:
COMPL $10, %EDX
JE ENDFOR
...
INC %EDX
JMP FOR
ENDFOR:
```

## 3.4 File assembly

Un file assembly ha estensione `.s` e può contenere diverse sezioni:

- **.section .data**: contiene le variabili globali e le costanti

```
.section .data
hello:
  .ascii "Hello, world!\n" ; Dichiarazione di una stringa
  costante
```

```
hello_len:
  .long . - hello ; Lunghezza della stringa (posizione corrente
  (.) - posizione iniziale)
```

- **.section .text**: contiene il codice assembly composto da istruzioni, etichette e sottoprogrammi

```
.section .text
.global _start ; Nome convenzionale del punto di inizio del
  programma
_start:
  movl ...
  ...
```

- **.section .bss**: contiene le variabili globali non inizializzate (spazio da riservare)

### 3.5 Compilazione

Per compilare un file assembly si compiono i seguenti passi:

1. **Compilazione**: si compila il file assembly con il comando `as` che crea un file binario (`.o`) contenente l'implementazione di ogni singolo file.
2. **Linking**: si uniscono i file binari con il comando `ld` che crea un file eseguibile a partire dai file binari.
3. **Esecuzione**: si rende eseguibile il file e si esegue con il comando `./<nomefile>`

## 4 Memoria

La memoria è una lista indicizzata di celle, a cui ognuna è associata un indirizzo. La memoria è composta da due parti principali:

- **Codice**: contiene le istruzioni
- **Dati statici**: contiene i dati

Non si può sapere a priori dove verrà caricato il programma in memoria, quindi è necessario utilizzare lo spostamento relativo per accedere ai dati e alle istruzioni.

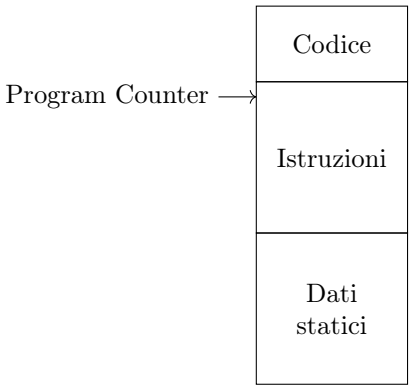



Figura 3: Struttura della memoria

***Definizioni utili 4.1***
***Footprint****: è l'area di memoria occupata da un programma:*

- ***L****: 32 bit*
- ***V****: 16 bit*
- ***B****: 8 bit*

## 4.1 Memoria dinamica

La parte dei dati è composta da due parti principali:

- **Heap**: contiene le variabili allocate dinamicamente e ha una dimensione variabile
- **Stack**: contiene le variabili locali e i parametri delle funzioni. Ha una dimensione fissa e limitata. Lo stack cresce con la modalità **LIFO** (Last In First Out), cioè l'ultimo elemento inserito è il primo ad essere estratto e nell'architettura x86 cresce verso l'alto. Lo stack è composto da 2 puntatori:
  - **ESP** (Extended Stack Pointer): punta all'ultimo elemento inserito nello stack
  - **EBP** (Extended Base Pointer): punta alla base dello stack

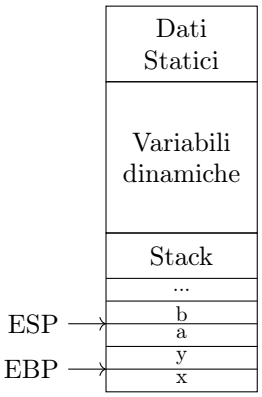



Figura 4: Struttura della memoria

Per gestire i dati nello stack si utilizzano le seguenti istruzioni:

- `PUSHL <source>`: inserisce il contenuto di un registro (o costante) nello stack
- `POPL <destination>`: estrae il contenuto dello stack e lo mette in un registro (o costante)

## 4.2 Richiamare una funzione

Per richiamare una funzione bisogna far saltare il Program Counter all'indirizzo della funzione e poi salvare l'indirizzo successivo nello stack. Per fare ciò si utilizza l'istruzione `CALL`.

- `CALL <etichetta>`: salva l'indirizzo successivo al Program Counter nello stack e salta all'istruzione con etichetta. Quando la funzione termina, per tornare al punto di chiamata si utilizza l'istruzione `RET`.
- `RET`: estrae l'indirizzo successivo al Program Counter dallo stack e salta a quell'indirizzo (torna al punto di chiamata).

Per recuperare i dati in memoria si utilizza l'istruzione `LEAL` (Load Effective Address):

- `LEAL <source>, <destination>`: prende l'indirizzo di memoria in cui è stato salvato qualcosa e lo mette in un registro

## 4.3 Struttura dettagliata della CPU

Di seguito è riportato uno schema più dettagliato dei componenti della CPU in modo da capire come vengono eseguite le istruzioni.

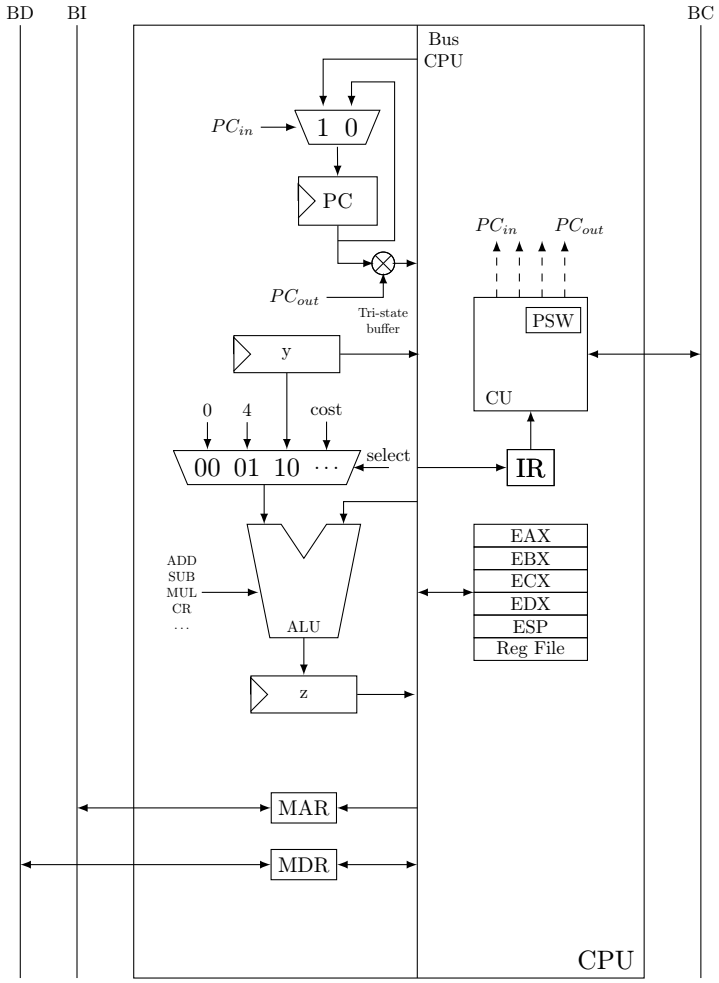



Figura 5: Schema della CPU

Da questo schema si possono notare le seguenti caratteristiche:

- Nella gestione del Program Counter il segnale passa attraverso un **Buffer Tri-State** che permette di disabilitare il segnale.
- Per mandare 2 valori alla ALU si utilizza un registro collegato ad un multiplexer che permette di selezionare quale valore mandare alla ALU. Nel multiplexer sono cablate anche delle costanti utili.
- Ogni indirizzo di memoria è gestito in **byte** indipendentemente. Quindi per accedere a parole da 32 bit bisogna andare avanti di 4 byte, mentre per accedere a parole da 64 bit bisogna andare avanti di 8 byte.

# 5 Micro operazioni

Le micro operazioni sono le operazioni elementari che la CPU esegue per eseguire un'istruzione.

***Definizioni utili 5.1***
***CPI** (Clock Per Instruction): è il numero di cicli di clock necessari per eseguire un'istruzione. L'obiettivo è avere un CPI il più basso possibile.*

## 5.1 Esempi

***Esempio 5.1***
*Andiamo ad analizzare la sequenza di micro operazioni (Fetch, Decode, Execute) per la seguente istruzione:*

```
MOVL %EAX, %EBX
```

F 1. $PC_{out}$, $MAR_{in}$, $READ$, $SELECT_4$, $ADD$, $Z_{in}$
*Il Program Counter manda l'indirizzo di memoria in cui si trova l'istruzione da eseguire al Memory Address Register e manda un segnale di lettura.*
*Il segnale di selezione 4 manda un segnale al multiplexer per selezionare il valore da mandare alla ALU e il segnale di addizione manda un segnale alla ALU per sommare 4 all'indirizzo di memoria. Ciò vuol dire che l'indirizzo di memoria successivo è l'indirizzo di memoria corrente + 1 word. Tutto ciò per incrementare il Program Counter in modo da accedere all'istruzione successiva.*

2. $WMFC$, $Z_{out}$, $PC_{in}$
***WMFC**(Wait for Memory Function to Complete): è un segnale che blocca la CPU finchè il dato viene messo nel bus dati.*
*Siccome in questo ciclo di clock il bus non viene utilizzato viene sfruttato per mandare il segnale di incremento al Program Counter.*
*Il segnale di lettura manda un segnale di attesa finchè il dato non viene messo nel bus dati.*

3. $MDR_{out}$, $IR_{in}$
*Il Memory Data Register manda il dato letto dall'indirizzo di memoria all'Instruction Register.*

DE 4. $EAX_{out}$, $EBX_{in}$, $END$
*Il contenuto del registro EAX viene mandato in uscita e viene messo in ingresso al registro EBX. Successivamente viene messo a 1 il segnale di fine che fa ripartire il ciclo di Fetch-Decode-Execute.*

***Esempio 5.2***
*Istruzione:*

`MOVL (%EAX), %EBX`

F 1. $PC_{out}$, $MAR_{in}$, $READ$, $SELECT_4$, $ADD$, $Z_{in}$
2. $WMFC$, $Z_{out}$, $PC_{in}$
3. $MDR_{out}$, $IR_{in}$

D 4. $EAX_{out}$, $MAR_{in}$, $READ$
5. $WMFC$

E 6. $MDR_{out}$, $EBX_{in}$, $END$

***Esempio 5.3***
*Istruzione:*

`ADDL $4, %ECX`

*La costante 4 è già presente nell'Instruction Register, quindi non c'è bisogno di andare a leggerla dalla memoria.*

F 1. $PC_{out}$, $MAR_{in}$, $READ$, $SELECT_4$, $ADD$, $Z_{in}$
2. $WMFC$, $Z_{out}$, $PC_{in}$
3. $MDR_{out}$, $IR_{in}$

DE 4. $OFFSET_{IR_{out}}$, $y_{in}$
*L'offset dell'Instruction Register serve per prendere il pezzo in cui è situata la costante 4*
5. $ECX_{out}$, $SELECT_y$, $ADD$, $Z_{in}$
6. $Z_{out}$, $ECX_{in}$, $END$

***Esempio 5.4***
`JZ END` *(salto relativo)*
*Il valore dell'etichetta* `END` *è già calcolato dall'assembler e viene memorizzato nell'Instruction Register*

F 1. $PC_{out}$, $MAR_{in}$, $READ$, $SELECT_4$, $ADD$, $Z_{in}$
2. $WMFC$, $Z_{out}$, $PC_{in}$
3. $MDR_{out}$, $IR_{in}$

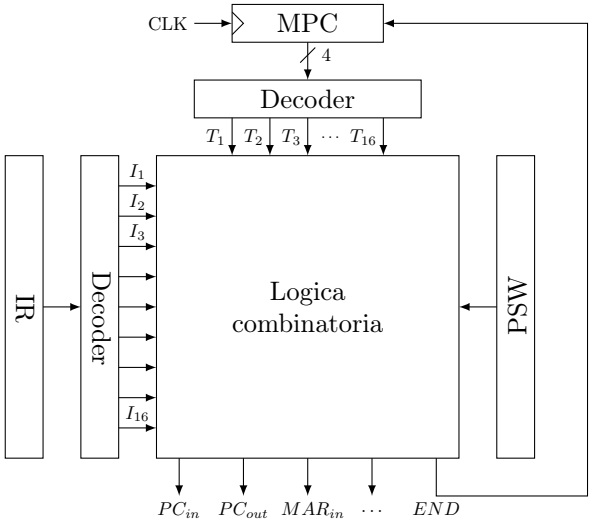
DE 4. *(if ZERO == 0 END)* , $OFFSET_{IR_{out}}$, $y_{in}$
*L'if e il resto vengono eseguiti insieme, sempre*
5. $PC_{out}$, $SELECT_y$, $ADD$, $Z_{in}$
6. $Z_{out}$, $PC_{in}$, $END$

## 5.2 Struttura della Control Unit

Per rappresentare i cicli di clock delle micro istruzioni si utilizza un registro chiamato **Micro Program Counter** che indica la prossima micro istruzione

da eseguire. È un contatore con un segnale di reset che permette di far ripartire l'esecuzione delle micro istruzioni. Questo registro viene poi collegato ad un decoder con 16 uscite che indica il tempo del ciclo di clock. Se il segnale $T_2 = 1$ allora il segnale $PC_{in} = 1$, quindi $PC_{in} = T_2$. È presente poi un decoder che parte dall'Istruction Register e ha in uscita tutte le istruzioni $I_n$. Si possono così realizzare tutte le equazioni in logica combinatoria, ad esempio:

$$END = (I_1 + I_2 + I_3 + \ldots) \cdot T_6 + I_3 \cdot T_4 \cdot \overline{ZERO}$$




Esiste una memoria che contiene tutte le micro istruzioni chiamata **Firmware**, ma **CPU cablate** in questo modo non si realizzano più.

## 5.3 Esercizi

***Esercizio 5.1***
*Descrivere le micro operazioni per l'istruzione:*

*Fetch*

*F* 1. $PC_{out}$, $MAR_{in}$, $READ$, $SELECT_4$, $ADD$, $Z_{in}$
*Il Program Counter manda l'indirizzo di memoria in cui si trova l'istruzione da eseguire al Memory Address Register e manda un segnale di lettura.*

*Il segnale di selezione 4 manda un segnale al multiplexer per selezionare il valore da mandare alla ALU e il segnale di addizione manda un segnale alla ALU per sommare 4 all'indirizzo di memoria. Ciò vuol dire che l'indirizzo di memoria successivo è l'indirizzo di memoria corrente + 1 word. Tutto ciò per incrementare il Program Counter in modo da accedere all'istruzione successiva.*

2. $Z_{out}$, $PC_{in}$, $WMFC$
   ***WMFC****(Wait for Memory Function to Complete): è un segnale che blocca la CPU finchè il dato viene messo nel bus dati.*

   *Siccome in questo ciclo di clock il bus non viene utilizzato viene sfruttato per mandare il segnale di incremento al Program Counter.*

   *Il segnale di lettura manda un segnale di attesa finchè il dato non viene messo nel bus dati.*
3. $MDR_{out}$, $IR_{in}$
   *Il Memory Data Register manda il dato letto dall'indirizzo di memoria all'Instruction Register.*

***Esercizio 5.2***
*Descrivere le micro operazioni per l'istruzione:*

```
INC %EAX
```

*F*
1. $PC_{out}$, $MAR_{in}$, $READ$, $SELECT_4$, $ADD$, $Z_{in}$
2. $Z_{out}$, $PC_{in}$, $WMFC$
3. $MDR_{out}$, $IR_{in}$

*DE*
1. $EAX_{out}$, $SELECT_0$, $CB$, $ADD$, $Z_{in}$
2. $Z_{out}$, $EAX_{in}$, $END$

***Esercizio 5.3***
*Descrivere le micro operazioni per l'istruzione:*

```
INC var
```

*Assumo la variabile come un indirizzo immediato nell'istruzione. Si fa riferimento ad essa con* $IR_{imm\_field}$

*F*
1. $PC_{out}$, $MAR_{in}$, $READ$, $SELECT_4$, $ADD$, $Z_{in}$
2. $Z_{out}$, $PC_{in}$, $WMFC$
3. $MDR_{out}$, $IR_{in}$

*DE*
1. $IR_{imm\_field\_out}$, $MAR_{in}$, $READ$, $WMFC$
2. $MDR_{out}$, $SELECT_0$, $CB$, $ADD$, $Z_{in}$
3. $Z_{out}$, $MDR_{in}$, $WRITE$, $WMFC$, $END$

***Esercizio 5.4***
*Descrivere le micro operazioni per l'istruzione:*

```
CALL etichetta
```

*Dove etichetta è un indirizzo immediato, ma relativo al program counter* $PC + IR_{imm\_field}$

*F*
1. $PC_{out}$, $MAR_{in}$, $READ$, $SELECT_4$, $ADD$, $Z_{in}$
2. $Z_{out}$, $PC_{in}$, $WMFC$
3. $MDR_{out}$, $IR_{in}$

*DE*
1. $ESP_{out}$, $SELECT_4$, $SUB$, $Z_{in}$
2. $Z_{out}$, $MAR_{in}$, $ESP_{in}$
3. $PC_{out}$, $MDR_{in}$, $WRITE$, $V_{in}$
4. $IR_{imm\_field}$, $SELECT_V, ADD$, $Z_{in}$, $WMFC$
5. $Z_{out}$, $PC_{in}$, $END$

***Esercizio 5.5***
*Descrivere le micro operazioni per l'istruzione:*

```
RETURN
```

*F*
1. $PC_{out}$, $MAR_{in}$, $READ$, $SELECT_4$, $ADD$, $Z_{in}$
2. $Z_{out}$, $PC_{in}$, $WMFC$
3. $MDR_{out}$, $IR_{in}$

*DE*
1. $ESP_{out}$, $SELECT_4$, $ADD$, $Z_{in}$, $MAR_{in}$, $READ$
3. $Z_{out}$, $ESP_{in}$, $EMFC$
3. $MDR_{out}, PC_{in}$

***Esercizio 5.6***
*Descrivere le micro operazioni per l'istruzione:*

```
CALL (%EAX, %EBX)
```

*Viene fatta la call all'indirizzo ottenuto sommando EBX a EAX*

*F*
1. $PC_{out}$, $MAR_{in}$, $READ$, $SELECT_4$, $ADD$, $Z_{in}$
2. $Z_{out}$, $PC_{in}$, $WMFC$
3. $MDR_{out}$, $IR_{in}$

*DE*
1. $ESP_{out}$, $SELECT_4$, $SUB$, $Z_{in}$
2. $Z_{out}$, $MAR_{in}$, $ESP_{in}$
6. $PC_{out}$, $MDR_{in}$, $WRITE$
3. $EAX_{out}$, $V_{in}$
4. $EBX_{out}$, $SELECT_V$, $ADD$, $Z_{in}, WMFC$
5. $Z_{out}$, $PC_{in}$, $END$

# 6 Dispositivi di input e output

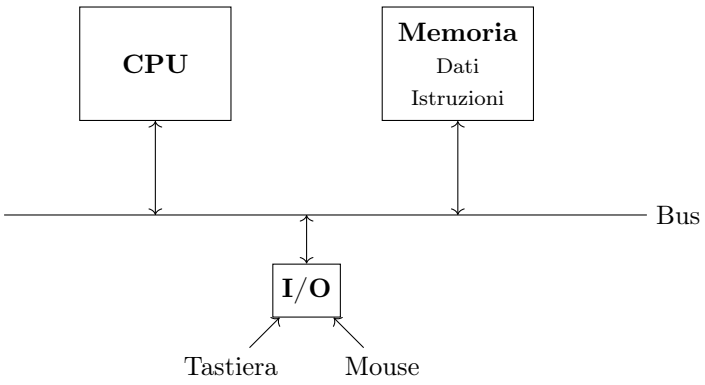



Figura 6: Schema di un sistema con dispositivi di input e output

Per poter ottenere un'interazione con l'utente è necessario avere dei dispositivi di input e output e a loro volta devono essere codificati per far corrispondere l'intenzione dell'utente con l'azione del computer. La strutture del microcontrollore input/output è composto da:

- **Dato**: contiene i dati da inviare o ricevere
- **Stato**: contiene lo stato del dispositivo
- **MC**: contiene il microcontrollore del dispositivo
- **IntA/D**: contiene l'interfaccia di analogico/digitale

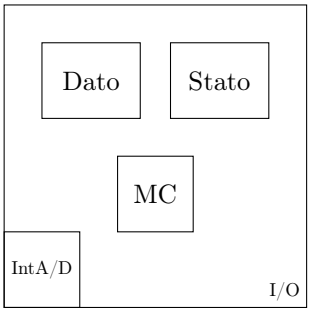



Figura 7: Struttura del microcontrollore input/output

La CPU accede ai valori dei registri di input/output tramite degli indirizzi che vengono riservati in un intervallo di memoria. Gli indirizzi di `Dato` e `Stato` sono:

- `IND DATA KEY` (Dato)
- `IND STATUS KEY` (Stato)

Questi indirizzi sono assegnati in fase di progettazione del sistema e sono fissi, ma si possono anche cambiare in certe architetture.

Ogni bit nel registro `status` ha un preciso significato, ad esempio se vale 0 significa che nessun tasto è stato premuto, se vale 1 significa che un tasto è stato premuto.

Un esempio in assembly è il seguente:

```
TEST_KEY:
 MOVL IND_STATUS_KEY, %EAX ; Carica lo stato della tastiera nel
   registro EAX
 CMPL $0, %EAX ; Compara il valore di EAX con 0
 JE TEST_KEY ; Se EAX = 0 allora salta a TEST_KEY
 MOV IND_DATA_KEY, %EBX ; Carica il tasto premuto nel registro EBX
 MOV %EBX, INDC ; Carica il tasto premuto nel registro C
```

La verifica di un dispositivo di input/output è un'operazione che viene detta **polling**.

Ogni volta che si vuole leggere un dato da un dispositivo di input/output si deve effettuare una **SVC** (Supervisor Call) che permette di richiamare del pezzo di codice al livello del sistema operativo che permette di effettuare diverse operazioni.

## 6.1 Ottimizzazione

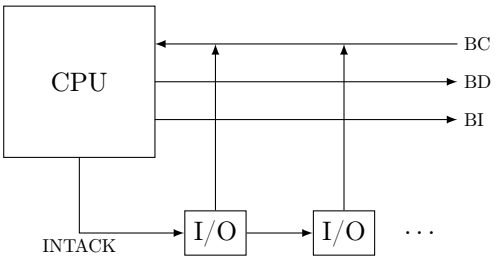
Ogni operazione di lettura effettuata con il bus richiede circa 10 cicli di clock.

```
TEST_KEY:
 MOVL IND_STATUS_KEY, %EAX ; 1 Read 1 Read Bus
 CMPL $0, %EAX ; 1 Read
 JE TEST_KEY ; 1 Read
 MOV IND_DATA_KEY, %EBX
 MOV %EBX, INDC
```

In totale si avranno $10 + 3$ cicli di clock $\approx 10$. Con una frequenza di $10GHz$ si avranno $10^9 clock/sec$. Visto che un umano può premere un tasto al massimo 10 volte al secondo, si può dire che la frequenza di lettura è troppo alta, quindi si sprecano cicli di clock e non può essere gestito in polling.

## 6.2 Interrupt

Al posto di fare polling, cioè la CPU che controlla continuamente lo stato del dispositivo, si può utilizzare un **interrupt** che è un segnale hardware che interrompe il normale flusso di esecuzione del programma solo quando il dispositivo è pronto. La CPU **prima di ogni fetch** controlla se c'è qualche richiesta di interrupt.




Esiste un bit $\overline{INTERRUPT}$ che vale 1 solo quando c'è una **interrupt request**. Ogni interrupt ha un valore che contiene un pezzo di codice chiamato **Interrupt Service Routine (ISR)** che viene passato alla CPU attraverso il `INTACK` (Interrupt Acknowledge). Le ISR sono gestite dal sistema operativo e tutto l'insieme viene chiamato **Device Driver**. Ogni dispositivo input/output ha interrupt con valori diversi.

Siccome l'esecuzione di un interrupt può modificare i registri della CPU può esserci qualche conflitto con dei programmi già in esecuzione, quindi c'è bisogno di un meccanismo per eseguire l'ISR senza che vengano modificati i registri della CPU. Questo viene effettuato dal dispositivo input/output che salva il valore dei registri `PSW` e `PC` ed eventuali altri registri salvaldoli nello stack prima di eseguire l'ISR.

## 6.3 DMA (Direct Memory Access)

Per trasferire grandi quantità di dati da un dispositivo di input/output alla memoria non conviene utilizzare degli interrupt perchè sarebbe uno spreco di risorse. Si utilizza il DMA che è un dispositivo che permette di trasferire dati dalla memoria al dispositivo di input/output senza passare per la CPU.

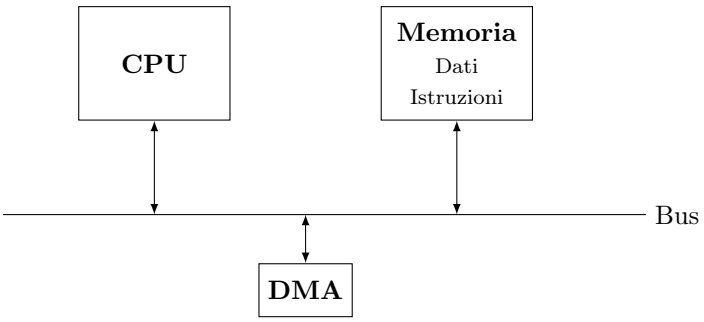



Figura 8: Schema di un sistema con DMA

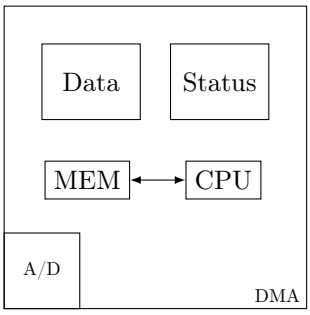



Figura 9: Struttura del DMA

## 6.4 Bus

Il bus è una linea di comunicazione che collegana i vari componenti di un sistema informatico. Vengono gestiti direttamente dalla CPU che quando non lo usa lo assegna ad altri componenti.

### 6.4.1 Bus Sincrono

Il segnale di clock è presente. Questo tipo di bus non viene mai realizzato perchè rallenta il sistema. Vengono invece utilizzati soltanto per collegare i componenti all'interno di un chip. Il segnale di clock può anche avere un ritardo, quindi non è detto che tutti i componenti ricevano il segnale di clock nello stesso momento.

### 6.4.2 Bus Asincrono

Il segnale di clock non è presente. Ciò permette di adattare dinamicamente la velocità di trasmissione dei dati in base alla velocità di trasmissione dei componenti.

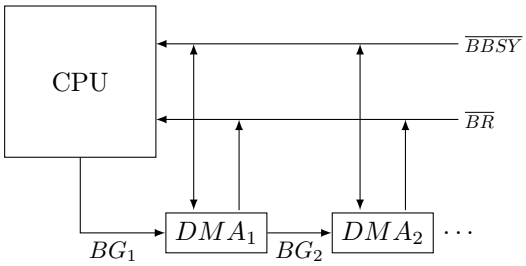



Figura 10: Schema di un sistema con bus

I segnali di controllo del bus sono:

- $\overline{BBSY}$ (Bus Busy): quando vale 0 indica che il bus è occupato
- $\overline{BR}$ (Bus Request): quando vale 0 indica che il dispositivo vuole utilizzare il bus
- $BG_1$ (Bus Grant): indica che il bus è assegnato al DMA1

- $BG_2$ (Bus Grant): indica che il bus è assegnato al DMA2

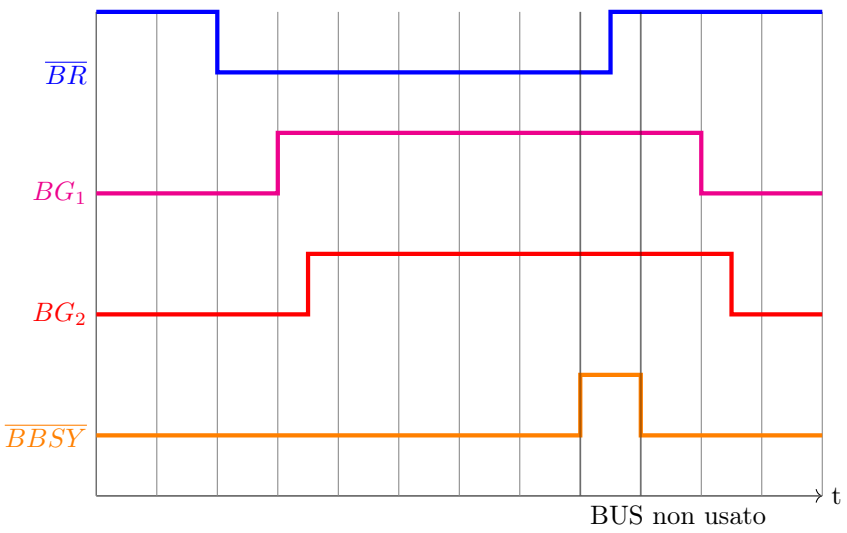



Figura 11: Segnali di controllo DMA

Il $DMA_2$ vuole utilizzare il bus, che però è già in uso dalla CPU, quindi mette a 0 il $\overline{BR}$ e aspetta che il bus venga liberato. Quando la CPU libera il BUS vede la request del $DMA_2$ e mette a 0 il $BG_1$ e il $BG_2$. Il $\overline{BBSY}$ viene messo a 0 quando il bus è occupato e quando viene liberato viene messo a 1 mettendo a 1 il $\overline{BR}$ e resettando a 0 il $BG_1$ e $BG_2$.

In questo modo si riduce al minimo il tempo in cui il BUS **non** è utilizzato.

# 7 Multitasking (multiprocesso)

Il multitasking è la capacità di un sistema operativo di eseguire più processi contemporaneamente e questa è una caratteristica che sta alla base di tutti i calcolatori moderni.

> ***Definizioni utili 7.1***
> *Il concetto di* ***programma*** *è diverso da un* ***processo****. Un programma è un file binario che contiene le istruzioni da eseguire, mentre un processo è un programma in esecuzione.*

Per permettere che più processi vengano eseguiti contemporaneamente si ricorre al concetto di **time sharing**

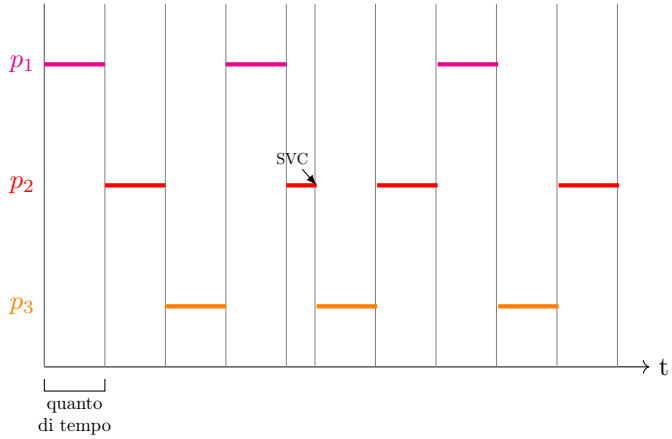



Figura 12: Time sharing tra i processi

Il quanto di tempo è stato messo a $1ms$ perchè è il tempo minimo per cui un umano non percepisce il cambio di processo. I processi vengono eseguiti per poco tempo, ma così frequentemente che sembra che vengano eseguiti contemporaneamente.

Se un processo in esecuzione esegue una supervisor call (SVC) il processo viene immediatamente interrotto e viene eseguita la SVC. Questo perchè finchè la SVC verrà eseguita si perderebbero migliaia di quanti di tempo.

## 7.1 Kernel

La parte del sistema operativo che gestisce i processi si chiama **kernel** e lo fa dialogando direttamente con la CPU.

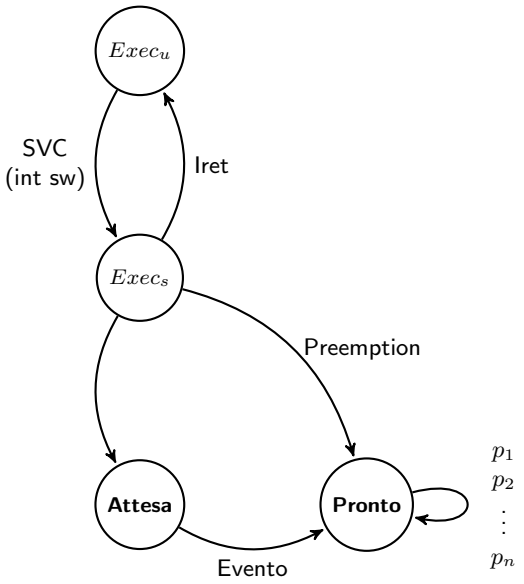



Figura 13: Flusso di esecuzione delle istruzioni

La chiamata di una SVC porta l'esecuzione dalla modalità utente ($E_u$ ) alla modalità supervisor ($E_s$ ). Questo perchè la SVC può accedere a tutte le parti del sistema operativo e quindi deve avere i permessi necessari. Nell'architettura x86 gli SVC sono degli interrupt chiamati **Software Interrupt**. L'istruzione assembly è la seguente:

```
INT $0x80 ; Numero dell'interrupt
```

La **preemption** è l'interruzione di un processo in esecuzione per eseguire un altro processo. Questo avviene quando un processo è stato eseguito per un tempo maggiore del quanto di tempo.

### 7.1.1 Scheduler

È una delle operazioni principali del sistema operativo. La CPU conta i cicli di clock che sono stati destinati ad un processo, in questo modo si può sapere che frazione del quanto di tempo è stata impiegata direttamente per il processo e non per eseguire gli interrupt. Per sapere quanto tempo è stato impiegato per eseguire un processo si usa il seguente comando bash:

```
$ time ./programma
```

L'output è diviso in:

- **User**: tempo impiegato per eseguire il programma
- **System**: tempo impiegato per eseguire gli interrupt
- **Real**: tempo reale impiegato per eseguire il programma

Lo scheduler controlla se il tempo dedicato ad un singolo processo è maggiore della metà di un quanto di tempo, in tal caso lo sospende e passa ad un altro processo, altrimenti lo fa continuare. Lo scheduler quindi decide quale processo della lista dei processi pronti deve essere eseguito. Ogni processo ha una certa priorità e lo scheduler deve decidere quale processo eseguire in base alla priorità e al tempo di esecuzione ( tempo reale). Per modificare la priorità dei processi si utilizza i seguenti comandi bash:

```
$ nice -n 10 ./programma
```

Il comando `nice` permette di modificare la priorità di un processo solo hardware, ma non software. Per modificare la priorità software si utilizza il comando:

```
$ renice -n 10 -p 1234
```

## 7.2 Realtime

Un processo può essere eseguito in tempo reale se il tempo di esecuzione è all'interno di un certo intervallo di tempo. Ci sono 2 tipi di realtime:

- **Soft Realtime**: il processo deve essere eseguito entro un certo intervallo di tempo ristretto ma non troppo.
- **Hard Realtime**: il processo deve essere eseguito entro un certo intervallo di tempo molto stretto.

## 7.3 Caratteristiche di un processo

Ogni processo ha le seguenti caratteristiche contenute in un **descrittore**:

- **PID**: identificatore del processo
- **Proprietà**: proprietario del processo
- **Stato**: indica lo stato della cpu
- **Cache**: contiene le informazioni più utilizzate dal processo
- **File ID**: contiene i file aperti dal processo

Questa struttura deve essere salvata dallo scheduler quando il processo viene sospeso e deve essere caricata quando il processo viene ripreso. Questa operazione viene chiamata **Context Switch**.

# 8 Stack

Prendiamo in considderazione il seguente codice in C:

```
int main() {
  int A;
  int B;
  int C;
```

```
  ...
  A = 5;
  B = 7;
  C = pippo(A, B);
  ...
  return;
}

int pippo(int x, int y) {
  int z;
  z = z * y;
  return z;
}
```

Questo codice in C chiede alla CPU di riservare 3 locazioni di memoria per le variabili `A`, `B` e `C`. Queste variabili verranno poi riempite da dei valori che siano costanti o funzioni.

Nel file binario è presente un **header** che indica come interpretare il file. Il file binario contiene un'immagine della memoria che verrà caricata in memoria prima di essere eseguito. Questo file binario contiene:

- **Codice**: contiene le istruzioni da eseguire
- **Dati statici**: contiene i dati che non cambiano durante l'esecuzione, come ad esempio i `#DEFINE` o le stringhe dei print, che sono tutti dati che rimangono costanti durante l'esecuzione.
- **Heap**: contiene i dati che vengono allocati dinamicamente durante l'esecuzione del programma, ad esempio la funzione `malloc()` in C.
- **Stack**: contiene i dati che vengono allocati durante l'esecuzione di una funzione e che vengono deallocati quando la funzione termina, ad esempio il `main` o le funzioni generiche. Lo stack pointer (ESP) punta alla cima dello stack e cresce verso l'alto.
- **Puntatori Limit e Base**: sono due registri che indicano la base e il limite dello stack. Questi servono per evitare che il programma salvato in memoria modifichi gli altri processi salvati in memoria nel caso in cui il programma vada in **Segmentation Fault**. Se il programma va in Segmentation Fault il sistema operativo lo termina tramite un interrupt.

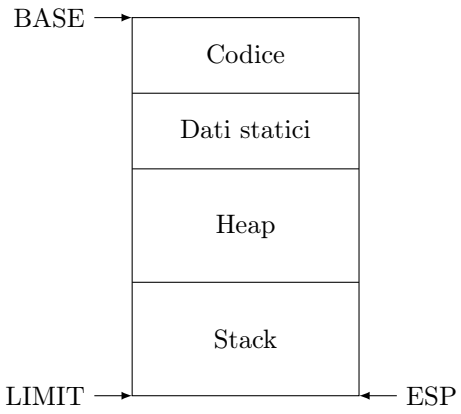



Figura 14: Struttura di un file binario

Lo stack è stato posizionato in basso perchè cresce verso l'alto, infatti se viene superato il limite dello stack la memoria rimanente viene posizionata al posto dello heap e di tutti i dati presenti al di sopra.

Prima di usare qualsiasi registro bisogna salvarlo all'interno dello stack, in modo da poterlo ripristinare alla fine dell'esecuzione. Questo viene fatto attraverso il comando asm:

```
PUSHL %EBP
```

Lo stack del programma in C scritto sopra è il seguente:

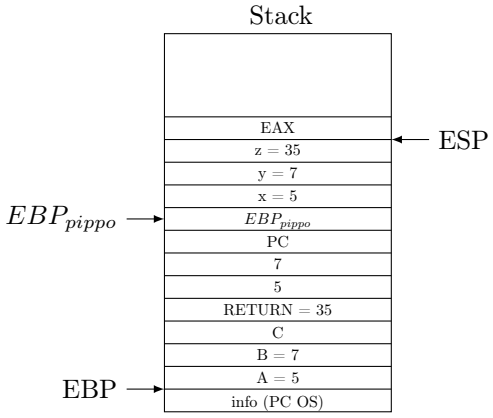



Figura 15: Stack con variabili

Di seguito il codice assembly per la gestione dello stack:

```
pippo:
PUSHL %EBP
MOVL %ESP, %EBP
SUBL $12, %ESP
```

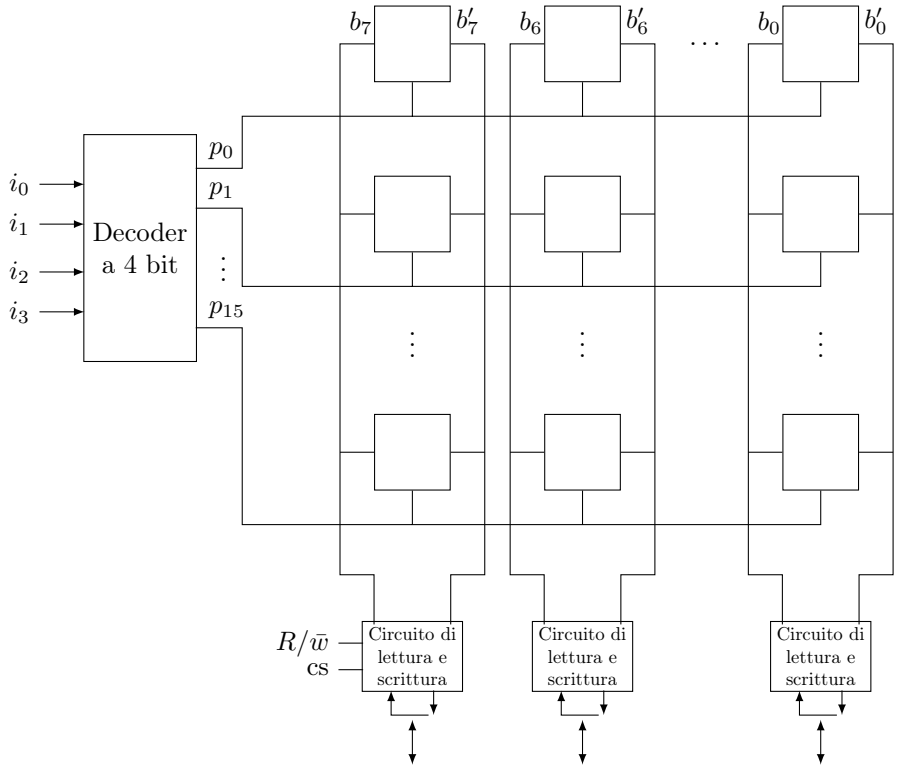
```
PUSHL %EAX
MOVL $12(%EBP), %EAX
MOVL %EAX, -4(%EBP)
...
MOVL %EAX, $16(%EBP)

POPL %EAX
ADDL $12, %ESP
POPL %EBP
RET ; Considera il valore in cima allo stack come PC


PUSHL %EBP
MOVL %ESP, %EBP
SUBL $12, %ESP
MOVL $5, $-4(%EBP)
MOVL $7, $-8(%EBP)
SUBL $4, %ESP
MOVL $-4(%EBP), %EAX
PUSHL %EAX
MOVL $-8(%EBP), %EAX
PUSHL %EAX
CALL pippo

ADDL $8, %ESP
POPL %EAX
MOVL %EAX, $-12(%EBP)
...
```

# 9 Struttura della memoria




Una memoria è compattata in un chip con diversi pin con compiti diversi:

- **8 Pin dati**: permettono di leggere o scrivere i dati
- **4 Pin indirizzi**: permettono di selezionare la cella di memoria
- **2 Pin controllo**: permettono di controllare la memoria
- **2 pin di alimentazione**: permettono di alimentare la memoria

## 9.1 Static RAM (SRAM)

La SRAM è una memoria statica, ovvero non ha bisogno di essere rinfrescata per mantenere i dati. Questo tipo di memoria è più veloce, ma occupa più spazio ed è più costosa.
Ogni singola cella è strutturata in questo modo:

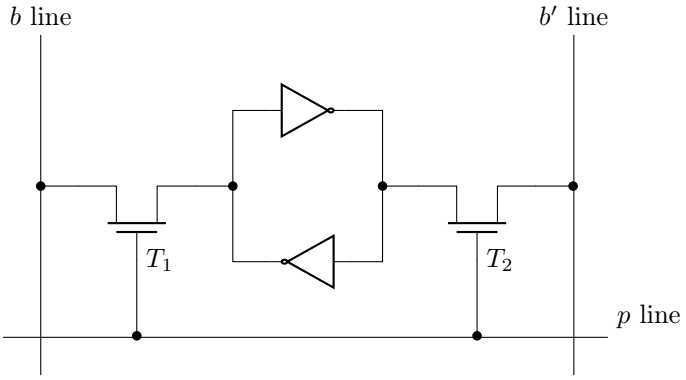



Figura 16: Struttura di una cella di memoria

Considerando che la porta NOT è costruita nel seguente modo:

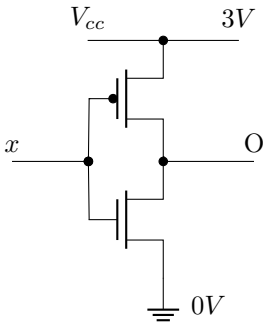



Figura 17: Circuito di negazione

La struttura della cella di memoria si può espandere ulteriormente in questo modo:

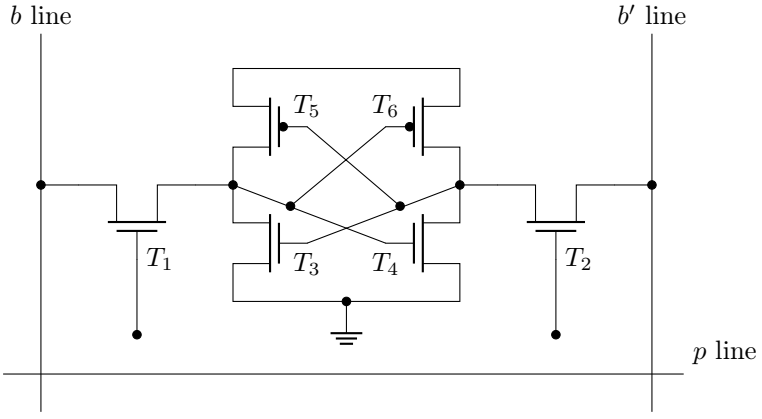



Figura 18: Struttura di una cella di memoria

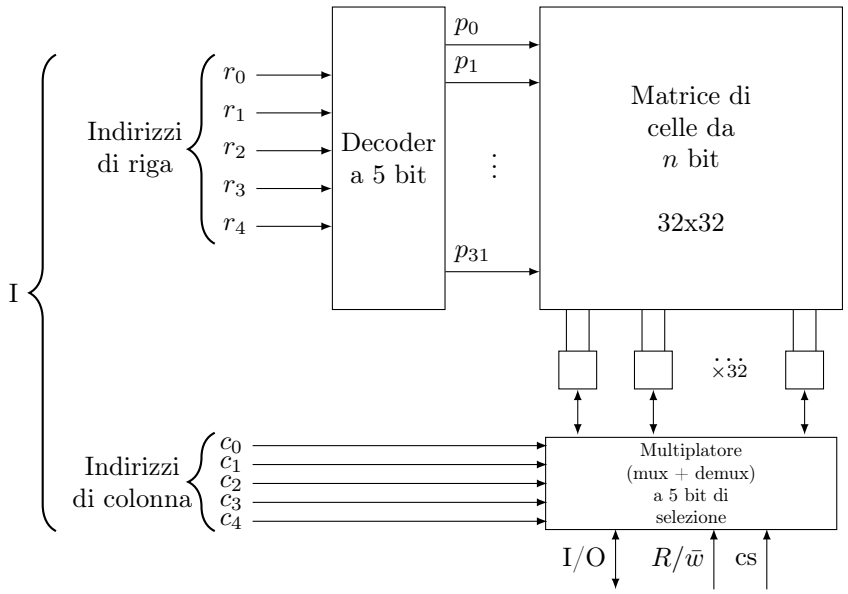
## 9.2 Dynamic RAM (DRAM)

I condensatori permettono di mantenere una carica elettrica per un certo periodo di tempo, questa carica però si degrada nel tempo. Per mantenere la carica si deve rinfrescare ogni condensatore per ripristinare la carica. Questo rinfresco però riduce la velocità della memoria. Visto che la DRAM occupa meno spazio rispetto alla SRAM, è più lenta, ma è più economica e può essere fatta più capiente.

***Esempio 9.1***

$$1Kbit = 32 \times 32 = 2^5 \times 2^5$$

*Si avrà una SRAM che sarà una matrice da m bit* $32 \times 32$




*Mediante gli indirizzi di riga e di colonna si può scegliere esattamente la cella desiderata.*

## 9.3 Syncronous DRAM (SDRAM)

La SDRAM è una DRAM sincrona, ovvero sincronizzata con il clock della CPU. Questo permette di avere una maggiore velocità di trasferimento dei dati.

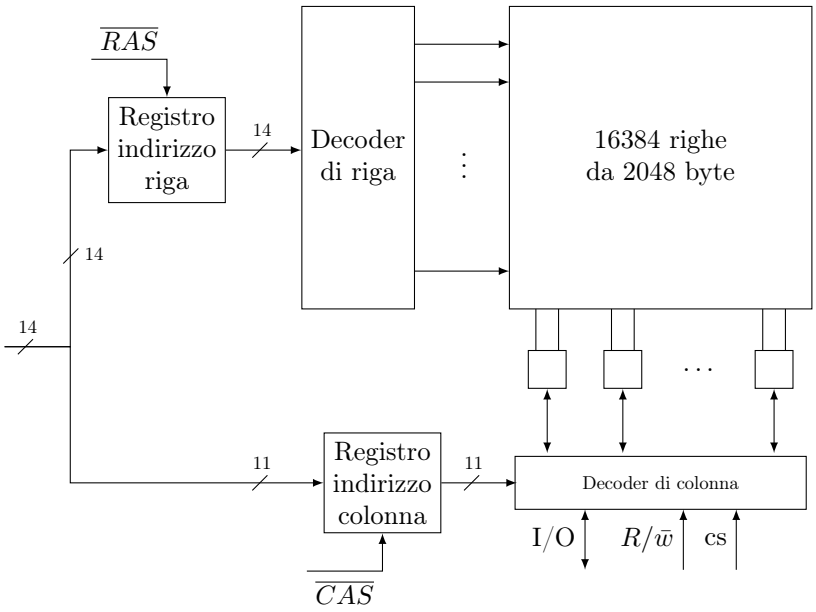
***Esempio 9.2***

$$256Mbit \;\; 32Mbit \times 8 \quad celle \; 16K \times 16K$$

*Ogni riga ha 16384 bit divisi in 2048 byte. Sono presenti 2 segnali:*

- $\overline{RAS}$ ***(Row Address Strobe)****: permette di selezionare la riga*

- $\overline{CAS}$ ***(Column Address Strobe)****: permette di selezionare la colonna*




*Al primo ciclo di clock viene attivato il segnale $\overline{RAS}$ selezionando la riga, al secondo ciclo di clock viene attivato il segnale $\overline{CAS}$ selezionando la colonna. In questo modo non servono $14 + 11$ bit, ma ne bastano 14; si sacrifica il ritardo per guadagnare area.*

La SDRAM è molto efficiente in lettura, perchè una volta impostati i segnali di riga e colonna, i dati vengono letti in sequenza senza dover cambiare i segnali, come mostrato nel grafico seguente:

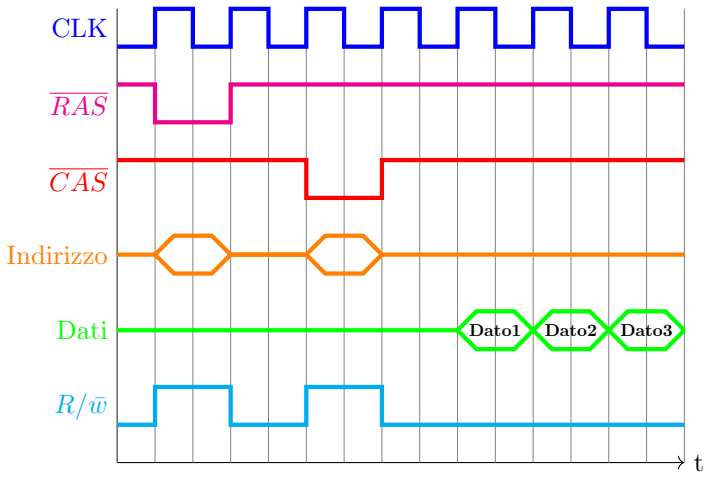



Figura 19: Segnali di controllo SDRAM

## 9.4 Caratteristiche

I dispositivi moderni hanno più di un banco di memoria, questo però rende più complesso il controllo della memoria. Di conseguenza la distanza dalla CPU alla memoria gioca un ruolo importante. La seguente tabella mostra le caratteristiche di alcune memorie ordinate in base alla distanza dalla CPU

| **CPU** | **Tecnologia** | **Accesso** | **Velocità** $[s]$ | **Dimensione** $[B]$ | **Costo** $[€/B]$ |
|---|---|---|---|---|---|
| **Registri** | CMOS<br>Elettronica<br>Volatile | Data path | $10^{-9}$ | $10^{4}$ | $10^{-3}$ |
| **Cache** | Statica<br>Elettronica<br>Volatile | Associativo | $10^{-8}$ | $10^{7}$ | $10^{-6}$ |
| **RAM** | Dinamica<br>Elettronica<br>Volatile | Casuale | $10^{-7}$ | $10^{12}$ | $10^{-8}$ |
| **SSD** | Elettronica<br>Elettronica Flash<br>Non volatile | Casuale<br>a blocchi | $10^{-5}$ | $10^{10}$ | $10^{-10}$ |
| **Hard Disk** | Magnetica meccanica<br>Non volatile | Diretto | $10^{-3}$ | $10^{13}$ | $10^{-11}$ |
| **CD/DVD** | Ottica | Diretto | $10^{-1}$ | $10^{10}$<br>fino a<br>$10^{11}$ | $10^{-9}$ |
| **Nastri DAT** | Magnetica | Sequenziale | $10^{0}$ | $10^{9}$<br>fino a<br>$10^{11}$ | $10^{-9}$ |

- **SSD**: Solid State Drive
- **Memoria Elettronica Flash**: È nata con le SIM card, è molto veloce e nel tempo è stata utilizzata per realizzare le USB. I droganti del silicio che

formano le celle di memoria non sono fissi, ma si possono muovere permettendo di modificare il silicio. Lo svantaggio è l'isteresi (si può tornare allo stato di origine, ma non sarà mai perfettamente uguale a ciò che si aveva in partenza), ovvero la memoria non può essere sovrascritta più di un certo numero di volte. All'inizio questo valore era di giorni, per aumentarlo si è pensato di distribuire le scritture uniformemente su tutta la memoria per far durare di più la memoria. La memoria flash è circolare e si scrive su un blocco (unità minima di lettura e scrittura). Il blocco letto viene copiato in un buffer, cioè una memoria RAM. In scrittura invece si scrive prima nel buffer e poi si scrive sul primo blocco libero.

MTBF (Mean Time Between Failure) è il tempo medio per cui un dispositivo si guasta. Quando l'MTBF viene superato, la probabilità di guasto aumenta esponenzialmente.

- **Nastri DAT**: (Digital Audio Tape) è un supporto di memorizzazione di dati digitali su nastro magnetico
- **RAM**: La memoria ram è "casuale", cioè la velocità di accesso è costante indipendentemente dalla posizione del dato.
- **Cache**: La cache è associativa, cioè il dato è associato ad un indirizzo e non ad una posizione, è più veloce, ma bisogna controllare se il dato è presente nella cache, se non lo è si verifica una **cache miss**.
- **Hard disk**: I dati vengono letti in blocchi da 512 byte. Il disco viene letto da una testina con una spira che ne legge la corrente inferita dalla carica magnetica del disco, la testina si muove su un braccio che si muove su un asse. Questo sistema però è molto fragile, perchè se la testina non è ad una distanza corretta dal disco, si può danneggiare il disco. Il tempo di lettura viene descritto dalla seguente formula:

$$T_{HD} = T_{seek} + T_{\omega} + T_B$$

Dove $T_{seek}$ è il tempo di ricerca, $T_{\omega}$ è il tempo di rotazione e $T_B$ è il tempo di trasferimento del blocco.

**FAT** (File Allocation Table) è una tabella (localizzata nel cerchio più esterno) che contiene le informazioni sui file presenti nel disco, come la dimensione, la posizione, le directory, ecc. Ogni sistema operativo ha la sua FAT, ma la più famosa è la FAT32.

**INODE** contiene un puntatore alla posizione del blocco successivo. Tutti i blocchi sono collegati da un'unica catena di blocchi liberi. Quando un blocco viene utilizzato viene tolto dalla catena di blocchi liberi e viene aggiunto ad un file. Quando un file viene cancellato vengono tolti i blocchi relativi a quel file e vengono aggiunti alla catena di blocchi liberi.

**Defrag** è un'operazione che serve a riordinare i blocchi in modo che siano tutti vicini tra loro, in modo da ridurre il tempo di accesso.

I dischi moderni sono più di uno e sono in una pila, inoltre vengono scritti su entrambi i lati (quindi 2 testine) e quando i dischi sono fermi le vengono posizionate in una posizione di sicurezza.

- **CD**: Sono nati per la musica con 700mb di spazio, ma poi sono stati utilizzati per i dati. La lettura viene effettuata da un laser che viene riflesso su uno specchietto che può ruotare in modo da riflettere il laser su tutte le parti possibili del disco. Per creare uno 0 si effettuavano dei veri e propri buchi nel dusco. Il CD è composto da una traccia unica che si sviluppa a spirale. All'inizio c'è il nome dei brani (massimo 13) e di seguito i brani. In seguito i CD sono anche stati utilizzati per distribuire i software.

  **WORM** (Write Once Read Many) è un CD che può essere scritto una sola volta e letto più volte. La scrittura viene fatta da un laser più potente che crea una cavità nel disco.

  Successivamente il CD si è evoluto per poter scrivere e cancellare i dati, nasce così il **DVD**. È presente un gel all'interno del disco che cambia il colore in base al colore del laser. Per scrivere si usa una luce rossa e una blu per l'1 e lo 0. Per leggere si usa una luce bianca che riflette il colore assunto dal gel all'interno del disco.

  Aumentando l'area dei DVD a 10GB si riesce a memorizzare interi film su di essi.

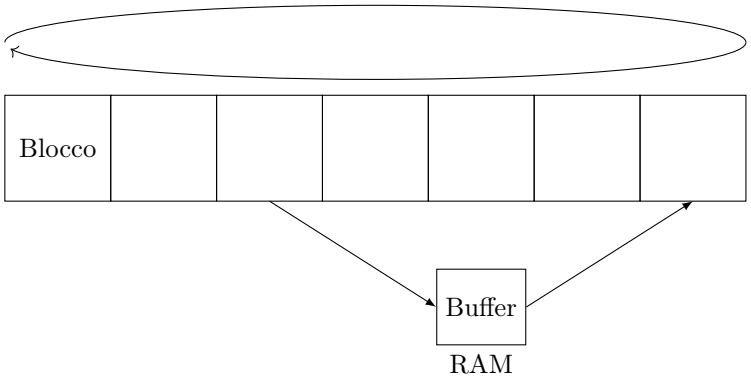



Figura 20: Struttura della memoria flash

## 9.5 Gerarchia della memoria

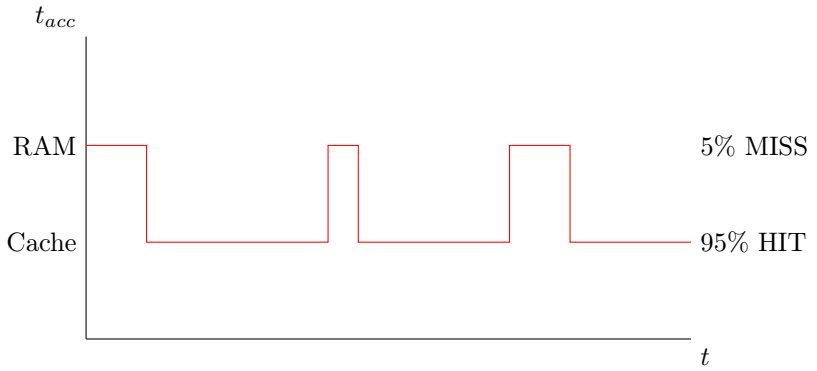
Il principio di località dice che se si accede ad un dato, è probabile che si acceda a dati vicini e si divide in:

- **Spaziale**: Se al tempo $t_j$ accedo alla cella di memoria $M_i$ succede che se considero un intervallo della memoria $\Delta M_i$ accederò a tutte le celle per un certo tempo $\Delta t$:

$$t_j \; M_i \rightarrow M_i + \Delta M \rightarrow t_j + \Delta t$$

- **Temporale**: Se accedo ad una certa cella $m_i$ al tempo $t_j$ succede che se considero un intervallo di tempo $\Delta t$ accederò ad un intorno di celle $\Delta M_i$:

$$M_i\ t_j \rightarrow t_j + \Delta t\ M_i + \Delta M$$




La probabilità di trovare un dato in cache (**cache hit**) è del $\approx 95\%$, la probabilità di non trovarlo (**cache miss**) è del $\approx 5\%$.

Più ci si allontana dalla CPU, più la memoria è grande, ma più è lenta, questa viene chiamato **gerarchia di memoria** per il principio di località.

# 10 Cache

Nella maggior parte delle architettura la cache fa parte del microprocessore stesso. Avere una cache è fondamentale per portare il CPI (Clock Per Instruction) a 1. Consideriamo una dimensione della cache ragionevole di $4MB$ e una dimensione della ram di $4GB$. Definiamo ora la dimensione di una "pagina" di $1KB$, la lettura si fa pagina per pagina perchè per il principio di località si è praticamente certi che se si accede ad una cella si accederà anche a celle vicine. La parola è la dimensione minima di lettura nella ram, in questo caso è di $1B$

$$\#dim \text{ cache} = 4MB$$

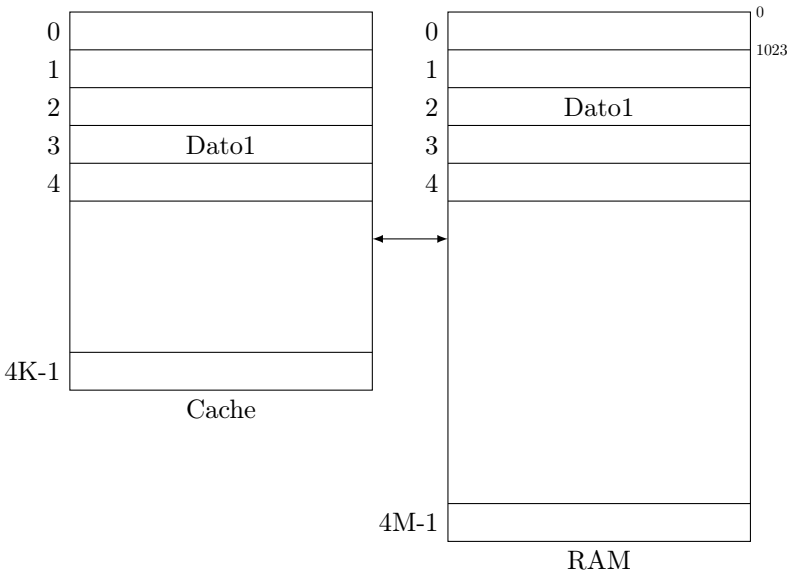
$$\#dim \text{ RAM} = 4GB$$

$$\#dim \text{ pagina} = 1KB$$

$$\#dim \text{ parola} = 1B$$

$$\#dim \text{ pagine RAM} = \frac{4GB}{1KB} = 4M$$

$$\#dim \text{ pagine Cache} = \frac{4MB}{1KB} = 4K$$




La CPU per cercare una parola di memoria deve cercare in cache, se la trova si avrà una cache hit e allora la ottiene molto in fretta, altrimenti si avrà una cache miss e si dovrà cercare in RAM.
Quando avviene una cache miss la CPU deve controllare se il dato in cache che stava cercando è stato modificato prima di sovrascrivaerlo.

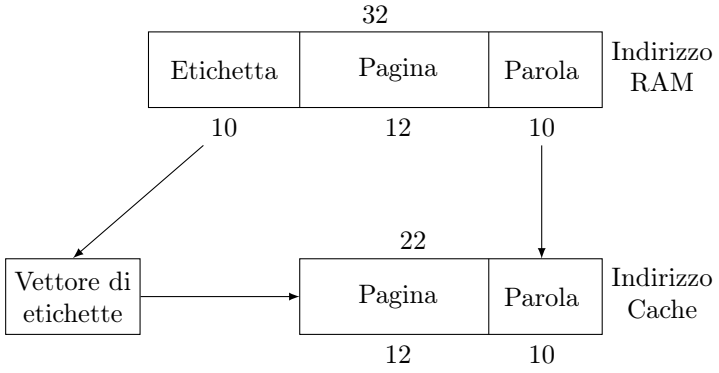
## 10.1 Indirizzamento della cache

Ci sono più modi per organizzare la cache e sono:

- Diretto
- Set-Associativo (tutte le cache reali sono set-associative)
- Associativo

### 10.1.1 Diretto

Nel caso di cache diretta si ha che ogni blocco di memoria è associato ad una sola cella di cache. Viene confrontato l'indirizzo della cache con una lista di etichette e se l'etichetta è uguale a quella in cui è stato salvato il dato si ha una cache hit. La lista di etichette contiene tutte le pagine in cache.
La cache diretta equivale ad avere una cache set-associativa con 4K set.

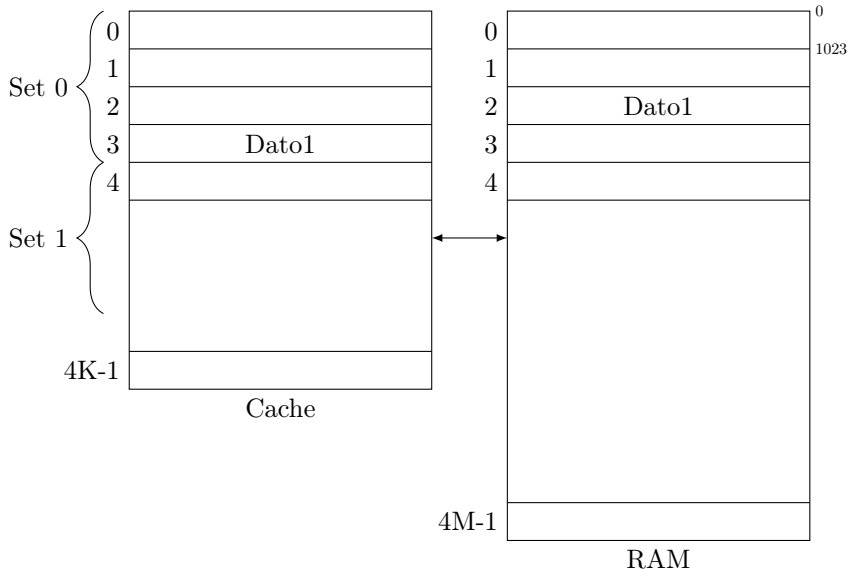



### 10.1.2 Set-Associativo

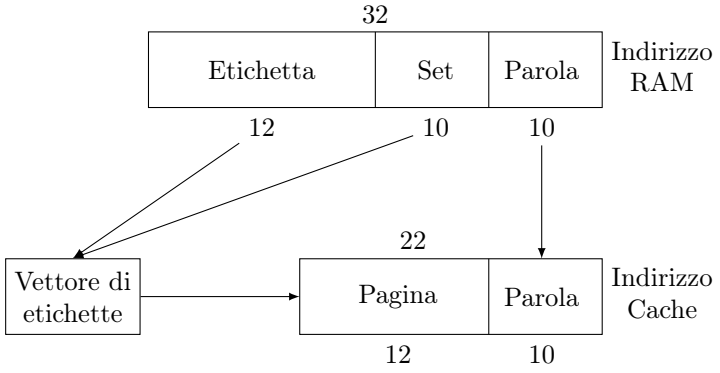
$$dim \text{ set} = 4$$

$$\#\text{ set cache} = \frac{\#\text{ pagine cache}}{\text{dim set}} = \frac{4K}{4} = 1K$$

In questo caso si ha che ogni blocco di memoria può essere associato ad una cella di cache appartenente ad un set di cache.

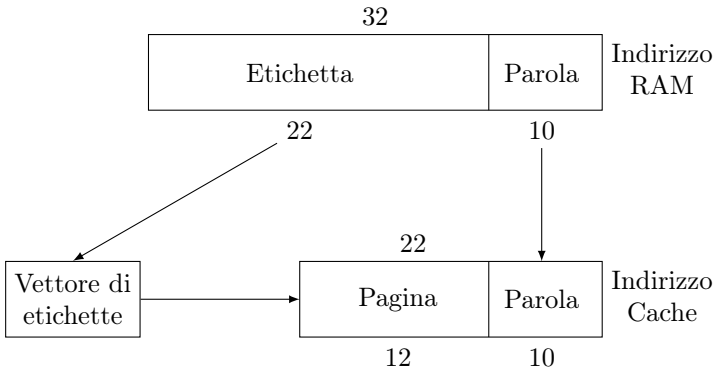



Il campo set indica in che set di cache si trova il dato, poi viene confrontata l'etichetta con le 4 etichette contenute nel set. Se l'etichetta è uguale a quella in cui è stato salvato il dato si ha una cache hit, altrimenti si ha una cache miss.




### 10.1.3 Associativo

Nel caso di cache associativa si ha che ogni blocco di memoria può essere associato ad una qualsiasi cella di cache. Il vettore di etichette è ordinato in modo che ogni cella di cache abbia un'etichetta associata.
Una cache associativa equivale ad una cache set-associativa con un solo set.

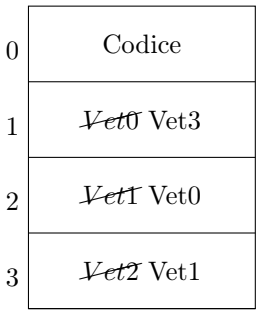



## 10.2 Rimpiazzamento

Quando si ha una cache miss si deve rimpiazzare un blocco di memoria con un altro, ma bisogna prima controllare se il dato da rimpiazzare serve ancora o no. Le due tecniche di dimpiazzamento più comuni sono:

- **LRU (Least Recently Used)**: Si rimpiazza il blocco più vecchio (tecnica più utilizzata)

  Non funziona nel caso in cui si ha un programma che ha il codice in una sola pagina (1024). Questo programma fa un ciclo 1024 volte e accede ad un vettore di caratteri (1B) che contiene 3098 caratteri. Considerando una cache con 4 pagine:




Con LRU viene cancellato il vettore necessario ad ogni ciclo e quindi si avranno $4 \cdot 1024$. Con MRU si avranno soltanto $4 + 1023$

- **MRU (Most Recently Used)**: Si rimpiazza il blocco più recente

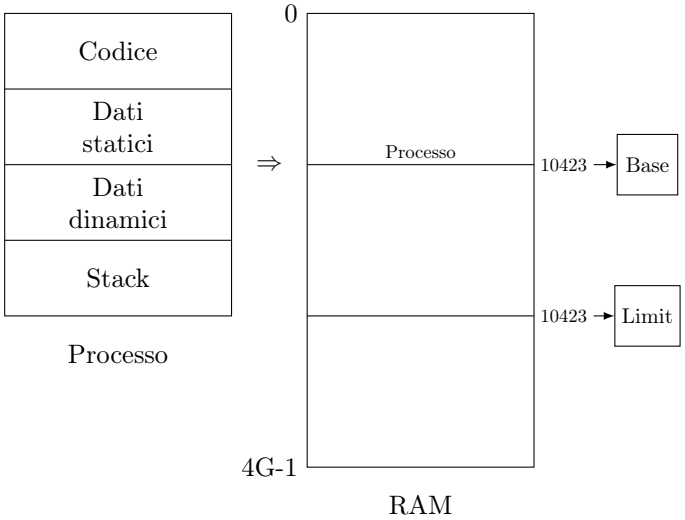
Per ogni pagina c'è un contatore che incrementa ogni volta che passa un ciclo di clock e viene azzerato quando si accede alla pagina.

## 10.3 Gestione

- Rilocazione
- Paginazione
- Memoria virtuale

Queste funzioni fanno si che la memoria appaia come uno spazio infinito ad un processo:

### 10.3.1 Rilocazione




c'è bisogno di un meccanismo per trasformare un indirizzo logico (di un processo) in un indirizzo fisico (della RAM). Questo meccanismo è la rilocazione e si divide in 2 tipi:

- **Rilocazione statica**: Il compilatore quando compila non può sapere in che zona della memoria viene caricato il processo, quindi l'unica cosa che può dire è l'inizio di ogni sezione del processo, ad esempio il codice inizia all'indirizzo 0, ecc...

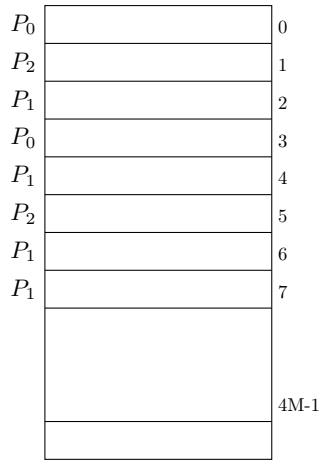
  Questa rilocazione prende l'indirizzo logico 0 e gli somma l'indirizzo fisico in cui inizia il codice, ad esempio 10423.

- **Rilocazione dinamica**: Prima di inserire l'indirizzo logico nel MAR si somma a quel valore ciò che si trova in un registro chiamato **base** che contiene l'indirizzo fisico in cui inizia il processo. È anche presente un registro chiamato **limit** che contiene la dimensione del processo e questo viene usato per controllare che **base** non sia maggiore di **limit**. Se questo controllo fallisce si ha un errore di segmentation fault. Se il processo vuole accedere ad un indirizzo fisico preciso lo deve fare tramite una syscall.

Ci sono però dei problemi, ad esempio quando un processo finisce viene liberato lo spazio che occupava e quando bisogna mettere in esecuzione un altro processo nello stesso spazio di quello vecchio, si ha un problema perchè il nuovo processo potrebbe non essere della stessa dimensione di quello vecchio, per cui potrebbe sovrapporsi ad un altro processo. Bisogna anche considerare i dati dinamici, cioè ad ogni malloc viene ingrandito lo spazio in memoria ingrandendo così il `limit`. Per risolvere questi problemi si usa la paginazione.

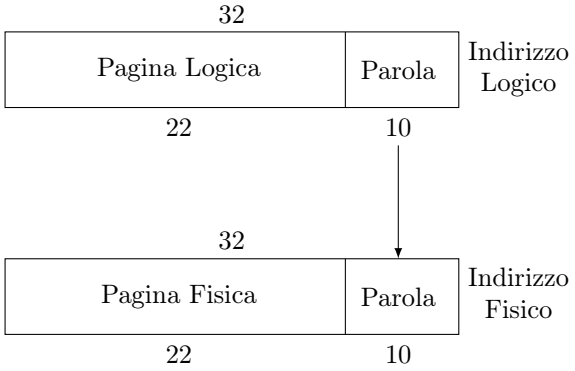
### 10.3.2 Paginazione

La memoria RAM è un insieme di pagine e dei processi possono essere caricati in pagine diverse.




RAM

Con un meccanismo di paginazione dopo aver sommato `base` e averlo controllato con `limit` si ha ancora un indirizzo logico che dovrà essere tradotto in un indirizzo fisico corrispondente alla pagina in cui si trova il processo.



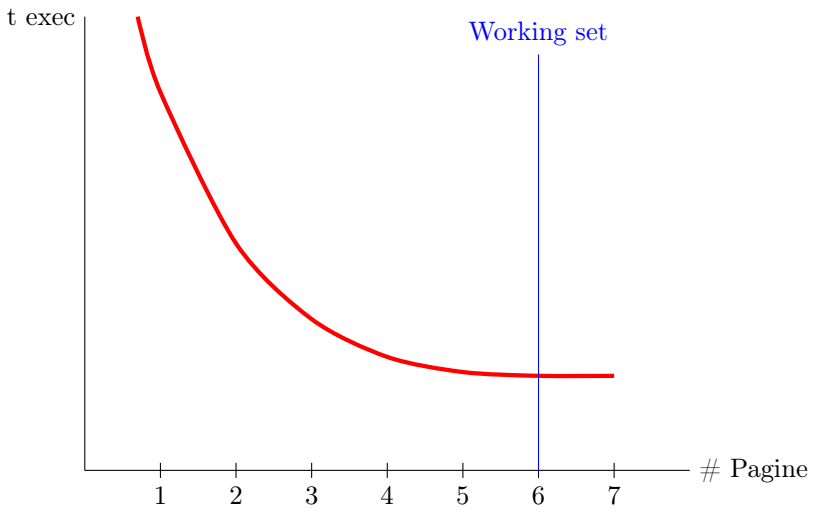

La pagina logica (quella in cui si trova il processo) viene mappata in una pagina fisica (quella in cui si trova il processo in RAM) attraverso una **tabella delle pagine**. Questa tabella è una tabella con lunghezza il numero di pagine logiche e ogni cella contiene l'indirizzo fisico della pagina corrispondente.

| Pagina logica | Pagina fisica |
|---|---|
| 0 | |
| 1 | 1042 |
| 2 | |
| 3 | 57 |
| 4 | 6 |
| 5 | |
| 6 | 4 |
| ... | |

È utile avere tutta la tabella soltanto nel caso in cui venga usata tutta la memoria, che per il principio di località non è possibile. Per questo motivo questa tabella si trova interamente nella RAM siccome è grande 12MB e ne viene caricata soltanto una parte nella CPU. Se la CPU non trova la pagina che sta cercando all' interno della tabella caricata in CPU la va a cercare in cache (o in RAM).

### 10.3.3 Memoria virtuale

Ogni processo ha un numero minimo di pagine necessario per poter essere eseguito senza interruzione, questo numero è detto **working set**.




Si avrà un numero di pagine di memoria per cui il tempo di esecuzione è minimo. Si dovrà quindi tenere in memoria soltanto il working set di ogni processo, il resto della memoria si troverà in un disco rigido in uno spazio riservato al sistema operativo chiamato **swap space** che consente di "salvare" le pagine di memoria che non sono nel working set per ripristinarle quando servono.

Se la CPU richiede una pagina che non si trova in cache e neanche in RAM, allora viene presa dal disco e messa in una delle pagine libere della RAM. In questo modo al processo viene permesso di usare (quasi) tutto lo spazio di memoria che vuole.

# 11 Pipeline

Possiamo dividere il processo di esecuzione in 4 fasi:

| **Fase** | **Memoria** | **Clock** |
|---|---|---|
| Fetch | 1 | 1 |
| Decode | $0-1$ | 1 |
| Execute | 0 | 1 |
| Write B. | $0-1$ | 1 |
| Totale | $1-3$ | 4 |

- **Fetch**: Ha un circuito di somma a parte, per permettere di sommare in un ciclo di clock è necessario avere dei registri flip-flop al posto di registri latch.
- **Decode**: Prende dalla memoria i parametri per l'operazione da fare. Se i dati al posto di essere nei registri sono in memoria bisogna accedere alla memoria e quindi si avranno 0 cicli di memoria se il dato non è presente in memoria, 1 se è presente.

- **Execute**: Esegue l'operazione richiesta. Sicuramente non si avranno accessi alla memoria. La fase exec impiega 1 ciclo di clock se e solo se si ha un'operazione aritmetica che prende dei dati da 2 registri e li mette in un terzo registro. Con un solo bus non si può raggiungere un ciclo di clock perchè visto che si può trasferire un solo dato alla volta bisogna fare più cicli di clock per trasferire più dati. Se invece avessimo 3 bus per fare i calcoli sarebbe possibile fare l'exec in un solo ciclo di clock.
- **Write Back**: Siccome l'accesso a memoria richiede circa 10 cicli di clock bisogna ridurre il numero di accessi alla memoria, questo si fa sfruttando la cache. Un altro modo per ridurre i cicli di memoria è di impedire che ci siano cicli di memoria in Write quando ci sono già stati in Decode e viceversa. Si dovranno avere istruzioni che facciano massimo 1 accesso alla memoria. Di conseguenza al posto di avere come totale $1-3$ cicli di memoria si avranno $1-2$:

| **Fase** | **Memoria** | **Clock** |
|---|---|---|
| Fetch | 1 | 1 |
| Decode | $0-1$ | 1 |
| Execute | 0 | 1 |
| Write B. | $0-1$ | 1 |
| Totale | $1-2$ | 4 |

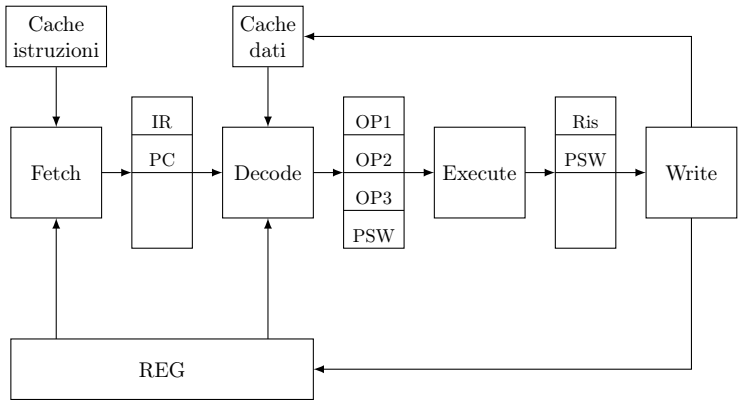
  Supponendo che la chache-hit sia al 100% si avrà un totale di $5-6$ cicli di clock.

Per ottimizzare ancora di più il CPI si sfrutta il concetto di **Pipeline**.

## 11.1 Concetto di pipeline

La pipeline può essere vista come un vero e proprio sistema di tubi, come ad esempio il trasporto del gas che ci impiega un po' per arrivare a destinazione, ma una volta arrivato il gas arriva in continuazione.

Se applichiamo questo concetto alla CPU possiamo vedere le fasi della cpu in più stati:



Ogni stato della pipeline lavorerà sulla copia dei registri messi a disposizione agli stati. Nel momento in cui una serie di istruzioni arriva alla pipeline si avrà che al primo ciclo di clock verrà eseguita la fase di Fetch della prima, al secondo la Fetch della seconda e la decode della prima e così via. In questo modo in un ciclo di clock si processano più istruzioni alla volta, di conseguenza alla fine della pipeline si avrà un CPI minore.

## 11.2 Stallo

Può succedere che durante l'esecuzione ci sia uno stato che va in stallo. Se ad esmpio si ha una compare e una jump di seguito, la jump non può essere eseguita finchè la compare non ha finito di eseguire. In questo caso si ha uno stallo che crea una **bolla** cioè uno stato che vuole lavorare non può e quindi si creano delle **dipendenze** tra le istruzioni.

## 11.3 Ottimizzazione

Per far tendere il CPI a 1 bisogna avere una pipeline efficiente.

Le dipendenze possono essere di più tipi e si risolvono nel seguente modo:

- **Istruzioni**: Si possono "allontanare" le istruzioni dipendenti riordinandole mantenendo però il senso del programma. I microprocessori moderni riordinano automaticamente le istruzioni per evitare stallo caricandole in blocco e riordinandole in modo da evitare stallo.
- **Salti condizionali**: Siccome non si sa quale ramo della condizione verrà eseguito si può fare **predizione di salto**. La predizione di salto viene fatta attraverso un circuito chiamato **crystal ball**. Finchè non si arriva alla write back i dati non vengono modificati, quindi per predirre il salto viene fatta partire la fetch mentre l'istruzione di salto aspetta la compare e se il compare è falso si butta via la fetch appena fatta e si fa ripartire la fetch non appena finisce la compare. Se il compare è vero si continua con la fetch fatta risparmiando così CPI.

  Ci sono metodi più efficienti per fare la predizione di salto, infatti alcuni salti hanno una probabilità maggiore di essere veri rispetto ad altri. Per questo motivo si considera che un salto sia vero. Se invece il salto è falso si usa l'output precedentemente salvato per decidere se fare la fetch o meno.

| Istruzione | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| movl %eax %eax | F | D | E | W | | | | | F | D | E | W |
| addl $4, %ecx | | F | D | E | W | | | | | F | D | E |
| subl $1, %ebx | | | F | D | D | E | W | | | | F | D |
| cmpl %eax, %ecx | | | | NOP | F | D | E | W | | | | |
| jz init | | | | | | F | D | D | D | E | W | |

Nell'istruzione subl al 4 ciclo di clock il registro %ebx non è ancora pronto, quindi si aspetta 1 ciclo di clock finchè non è pronto. Per capire quanto un

registro è pronto è presente un flag che segna se un certo registro è stabile o se è ancora in esecuzione.

$$CPI_{medio} = \frac{11}{5} \approx 2.2$$

$$\Downarrow$$

$$CPI_{medio} = \frac{11 - \overbrace{4}^{\text{ist. per riempire pipeline}}}{5} \approx 1.4$$

$$\Downarrow$$

$$CPI_{medio} = \frac{12 - 4}{6} \approx 1.35$$

L'ultimo CPI calcolato è dopo aver fatto tutte le branch dei jmp.

# 12 RISC (Reduced Instruction Set Computer)

Il tempo di esecuzione di un programma è dato da:

$$T_{cpu} = N_{istr} \cdot CPI_m \cdot \frac{1}{f_{clk}}$$

La cosa negativa è che i 3 fattori sono collegati, quindi non ci si può concentrare per minimizzarne uno alla volta.

- $f_{clk}$:
  - Non si può aumentare a piacere perchè nella CPU è presente un datapath con un cammino critico che fissa il tempo di clock.
  - La pipeline permette di aumentare $f_{clk}$ ma non è possibile aumentare all'infinito il numero di stadi della pipeline.
  - L'ISA (Instruction Set Architecture) è un altro fattore che influenza $f_{clk}$ perchè se si ha un'ISA complessa si avrà un $f_{clk}$ più basso. Il legame tra il set di istruzioni e frequenza di clock è estremamente stretto.
- $N_{istr}$:
  - In base all'ISA si avrà un numero diverso di istruzioni per uno stesso programma. Se si volesse ridurre il numero di istruzioni da un programma compilato bisognerebbe creare un ISA con istruzioni più complesse. Questo però portava il CPI a salire e le frequenze a scendere.
  - 2 Professori della Stanford University Hennessey e Patterson hanno raccolto diversi programmi in svariati linguaggi e hanno notato che il 90% delle istruzioni avevano 2 o 3 operandi, quindi espressioni semplici. I cicli erano singoli, quindi non c'erano cicli dentro cicli. Gli if erano al massimo a 2 innestazioni. Nonostante tutta la complessità il codice viene scritto in modo semplice e spesso quando le istruzioni sono semplici il codice è scritto meglio. I professori si sono accorti

che indipendentemente dai compilatori e dagli ISA la maggior parte dell'assembly aveva istruzioni semplici. Si aveva il 90% di istruzioni semplici e il 10% complicate. Hanno inventato il RISC (Reduced Instruction Set Computer) e si è iniziato a chiamare gli altri processori CISC (Complex Instruction Set Computer).

Siccome a Stanford c'era una fonderia di silicio i 2 professori hanno creato il primo processore RISC chiamato DLX. Con questo microprocessore mostrarono con i dati alla mano che la loro idea era giusta.

Successivamente il MIPS diventò il primo microprocessore RISC commerciale

## 12.1 Caratteristiche del RISC

- Istruzioni semplici
- Pochi metodi di indirizzamento
- Pipeline bilanciata dovuta alle istruzioni semplici
  - Abbatte il CPI facendolo tendere a 1
  - Aumenta la frequenza di clock

## 12.2 Alcuni processori RISC

- Intel
  - 80386: 60-80 MHz
  - Pentiun: 90 MHz
  - P6: 200 MHz
- MIPS: 100 MHz
- Motorola IBM
  - PowerPC: 200 MHz poi fino a 600 MHz
- DEC
  - Alpha: 600 MHz
- SUN
  - SPARC

Quando intel creò le istruzioni 80x86 per passarle al processore P6 si utilizza un **pre fetcher** che traduce le istruzioni 80x86 in istruzioni RISC. Questo sistema permetteva di avere un processore CISC ma con le prestazioni di un RISC. Dopo questa invenzione intel inventò il microprocessore Pentium.

# 13 Evoluzione dei calcolatori

- Anni '90:

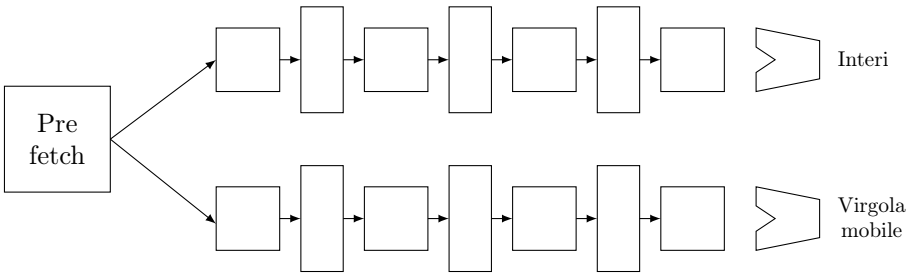
  La rivoluzione RISC ha portato ad un aumento delle prestazioni e ha raggiunto l'obiettivo di avere un CPI uguale a 1. Nonostante questo si è continuato a cercare di aumentare le prestazioni.

  - L'utente percepisce soltanto il tempo di elaborazione: $T_{cpu}$, quindi bisogna diminuire il tempo di esecuzione.
  - Il gestore del sistema (gestisce un pc per più utenti) percepisce le prestazioni in un altro modo. Esiste un livello minimo del tempo di esecuzione di un processo oltre il quale non cambia la percezione delle prestazioni. Il gestore però vuole che il tempo di esecuzione sia uguale per tutti gli utenti, quindi:

    $$\frac{\text{n° processi}}{\text{unità di tempo}}$$

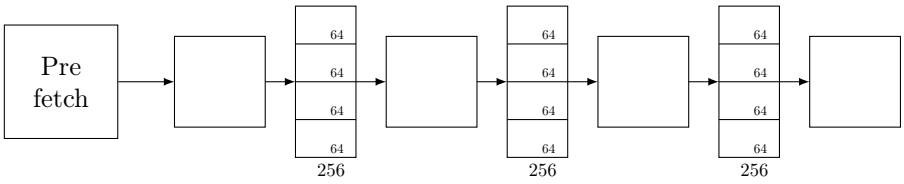
    il gestore vuole avere il maggior numero di processi per unità di tempo.

  Per aumentare le prestazioni sono nate le CPU **superscalari** che hanno un $CPI_m < 1$. PowerPC è stato il primo a scendere sotto l'1 dedicando una pipeline per l'esecuzione di interi e una per i float riuscendo così ad eseguire 2 operazioni in 1 ciclo di clock e quindi $CPI_m = 0.5$. Separando le pipeline si possono anche avere più segnali di clock e quindi il cammino critico sarà quello della ALU intera.

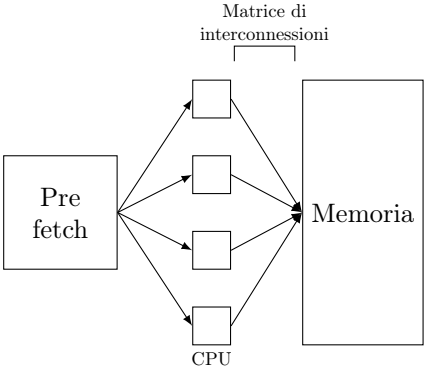



  Successivamente sono state aggiunte più pipeline all'unità di prefetch, il problema però è che le dipendenze possono accadere tra più pipeline diverse.

  Successivamente nacque il **VLIW** (Very Long Instruction Word) che permetteva di avere più istruzioni in un'unica istruzione. Questo è possibile utilizzando un registro da 256 bit che contiene più istruzioni (8 istruzioni) tutte svolte in parallelo. Aggiungere un registro troppo grande può creare problemi con le dipendenze tra le istruzioni.




Successivamente si tentò di implementare il parallelismo e sono nati i calcolatori vettoriali che permettevano di eseguire operazioni in parallelo grazie a più ALU. Il computer vettoriale più famoso era il Thinking Machine che aveva più CPU che lavoravano in parallelo scrivendo il risultato in un'unica memoria condivisa.




Questo sistema però scomparve per la legge di **Amdahl** che dice che la prestazione totale è data da:

$$P_{T_1} = U_1 \cdot P_1 + U_2 \cdot P_2 + \ldots + U_k \cdot P_k$$

Che sarebbe la somma delle moltiplicazioni del tempo di utilizzo del componente moltiplicato per la sua prestazione. Se si aumentano le prestazioni del secondo componente di 100 volote si ha:

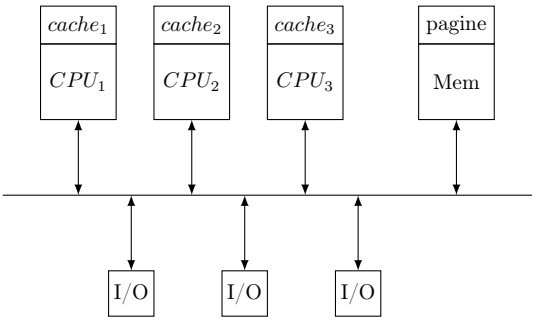
$$P_{T_2} = U_1 \cdot P_1 + 100 \cdot U_2 \cdot P_2 + \ldots + U_k \cdot P_k$$

e lo speedup è:

$$S = \frac{P_{T_2}}{P_{T_1}}$$

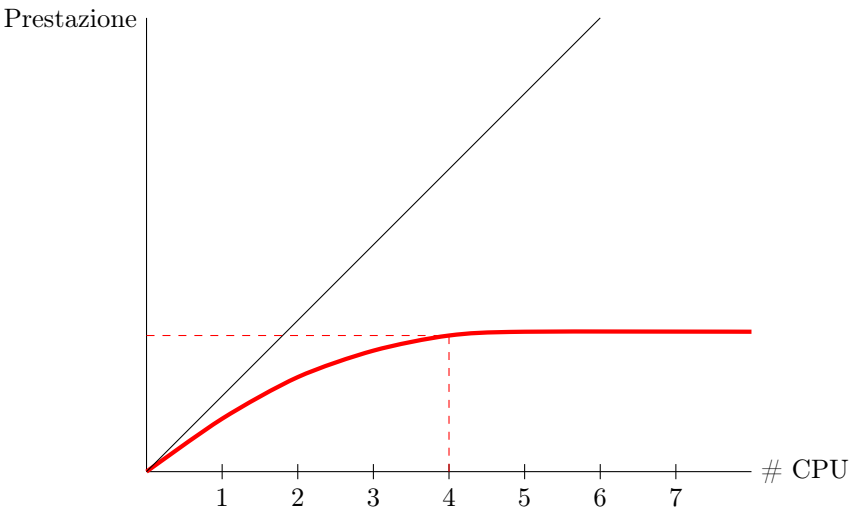
Il miglioramento dipende dalla percentuale di utilizzo di quel componente, quiundi se viene usato poco la prestazione non migliora molto. Quindi se la percentuale di paralelizzazione dell'intero sistema è più bassa del 90% non si ha un miglioramento significativo perchè rimane una parte sequenziale.

L'idea dei calcolatori vettoriali è fallita, quindi si è passati a parallelizzare i processi (task) riducendo il tempo totale di esecuzione dei processi. Sono nati quindi i **Multiprocessori**

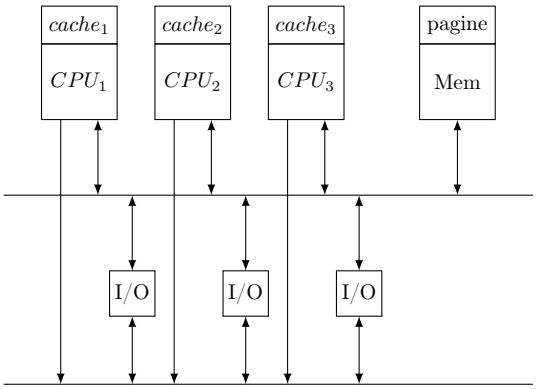



Siccome ogni CPU ha una cache diversa e la memoria è condivisa tra le CPU allora le pagine della memoria possono trovarsi in cache diverse e questo crea il problema della **coerenza della cache**. Si può risolvere in diversi modi, ma l'algoritmo migliore è quello di **Snoopy** (ficcanaso) che continua a monitorare il bus e tutte le volte che vede un indirizzo si salva questa informazione. Si avrà quindi una tabella aggiuntiva che contiene l'indirizzo e il processore che lo ha in cache. Se una CPU vuole una pagina che si trova nella cache di un'altra CPU deve chiedere di inserirla in memoria perchè potrebbe essere non aggiornata.

Avere più CPU però non sempre migliora le prestazioni, questo perchè si ha un solo bus e quindi si ha un collo di bottiglia. A 4 CPU si arriva alle prestazioni massime.

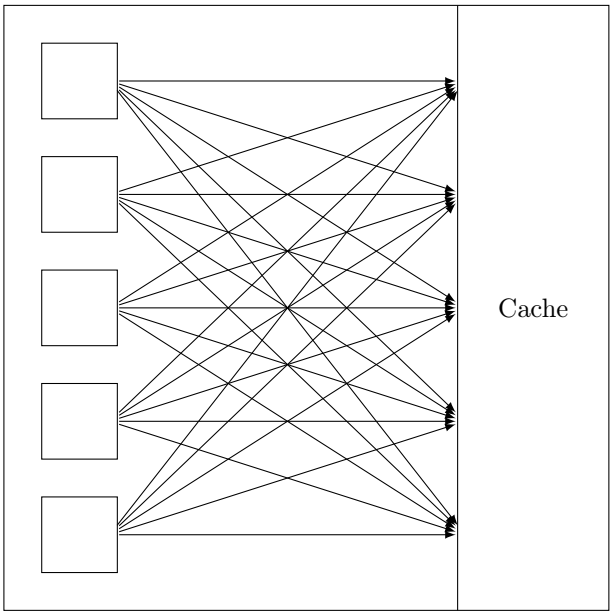



Si potrebbe aumentare il numero di bus, ma questo aumenterebbe i costi e la complessità del sistema.




- Anni 2000:

  Nacquero i Network Chip (N.C) con più chip (multicore) in cui ogni chip aveva una cache condivisa e ci si poteva accedere in parallelo.




CPU
Multicore

Con l'arrivo di applicazioni multimediali e real-time nacque la GPU (Graphics Processing Unit) che permetteva di fare calcoli molto semplici in parallelo. La GPU è composta da migliaia di piccoli processori che eseguono operazioni elementari e la memoria video viene salvata all'interno del chip (in memory computation) per eliminare la necessità del bus.

Per aumentare la complessità nacque la GPGPU (General Purpose GPU) che permette di fare calcoli più complessi con la GPU. Questo permette

di scrivere un programma in cui la parte sequenziale viene eseguita dalla CPU e la parte parallelizzabile viene eseguita in parallelo sulla GPU.

Con l'avvento dell'intelligenza artificiale basata sui dati si è passati ad usare la GPGPU per costruire le reti neurali.