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.

1.10 Debugging

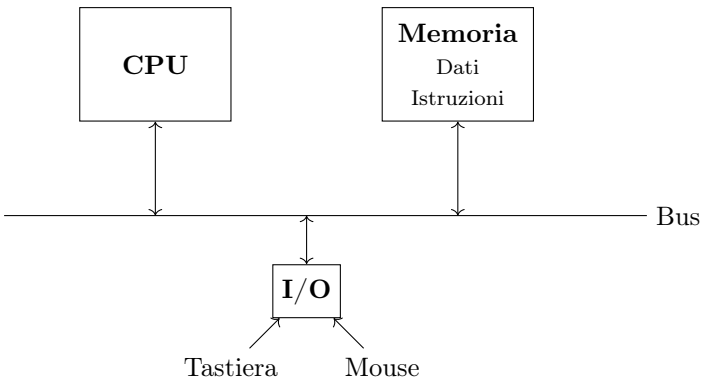
2 Architettura di Von Neumann

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

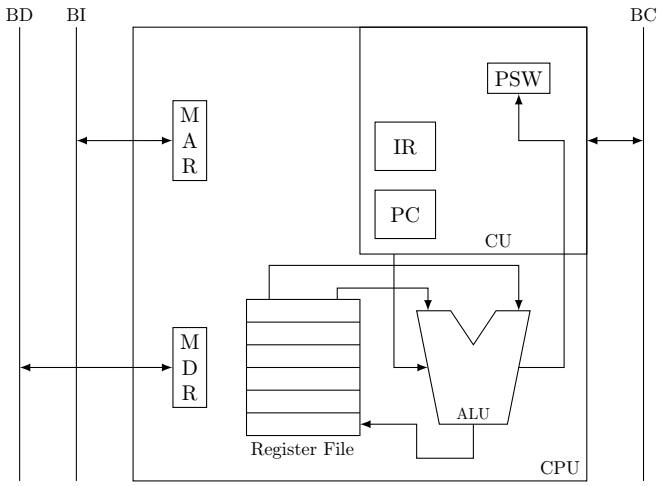


Figura 1: Struttura di un processore

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

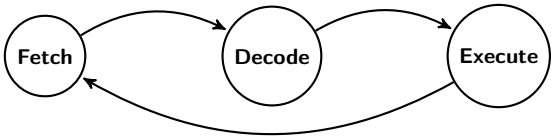


Figura 2: Flusso di esecuzione delle istruzioni

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`. Questo viene fatto:

- **LOAD**: per caricare una cella (A):
  1. Scrive l'indirizzo della cella da leggere nel MAR
  2. TODO

2.4.2 Processing Unit

Questa unità può eseguire 3 operazioni:

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

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.

2.4.3 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.

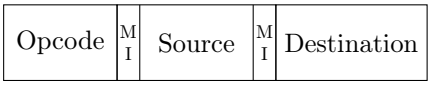
2.4.7 Metodi di indirizzamento

- **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:



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.

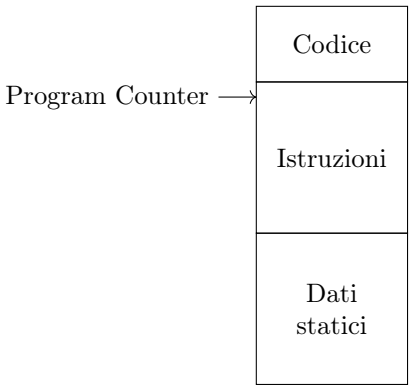


Figura 3: Struttura della memoria

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 memoria dinamica 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 composta anche da 2 puntatori:
  - **ESP** (Extended Stack Pointer): punta all'ultimo elemento inserito nello stack
  - **EBP** (Extended Base Pointer): punta alla base dello stack

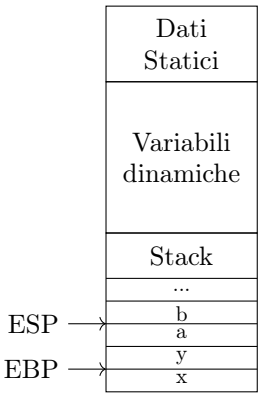


Figura 4: Struttura della memoria

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 uan 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.

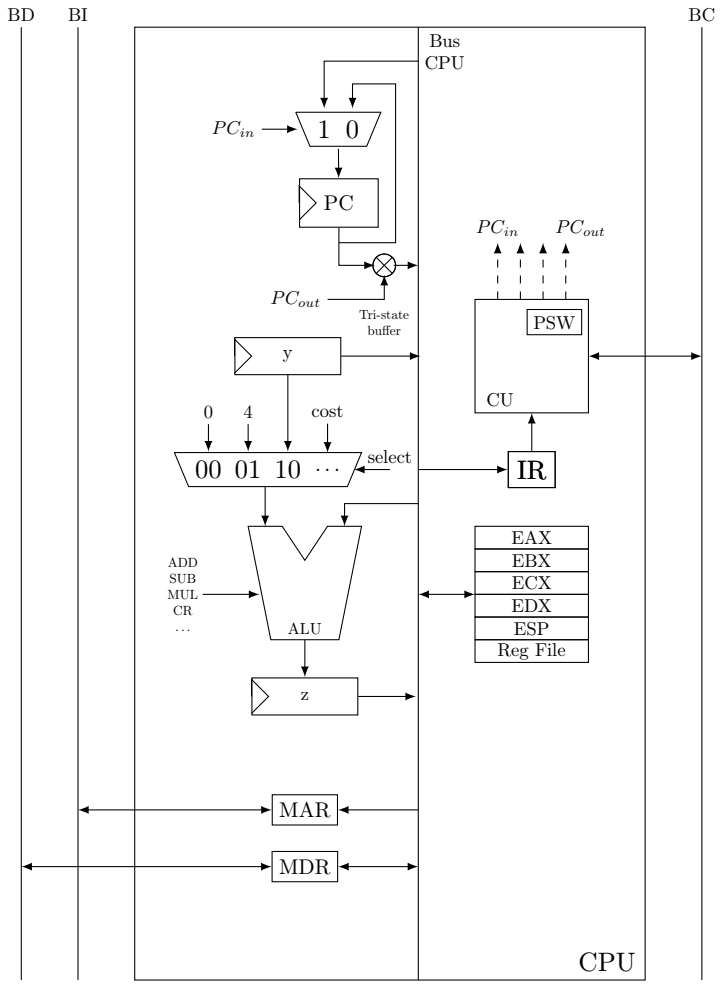


Figura 5: Schema della CPU

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 delle costanti utili da passare alla ALU.
- 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 il clock successivo della 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*

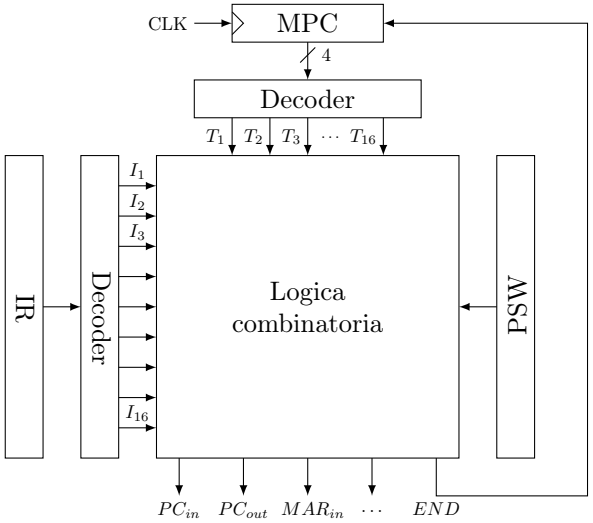
F 1. PC_{out}, MAR_{in}, $READ$, $SELECT_4$, ADD, Z_{in}
2. $WMFC$, Z_{out}, PC_{in}
3. MDR_{out}, IR_{in}

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 incre-

mentare 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 il clock successivo della 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

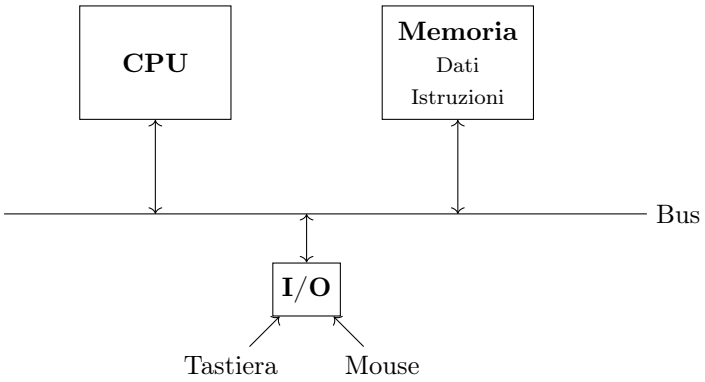


Figura 6: Schema di un sistema con dispositivi di input e output

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

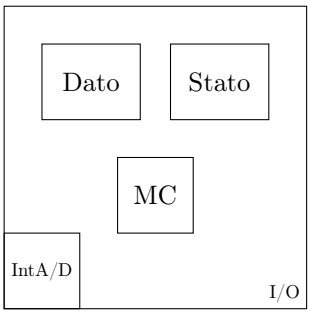


Figura 7: Struttura del microcontrollore input/output

La CPU accede ai valori dei registri di input/outpit 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

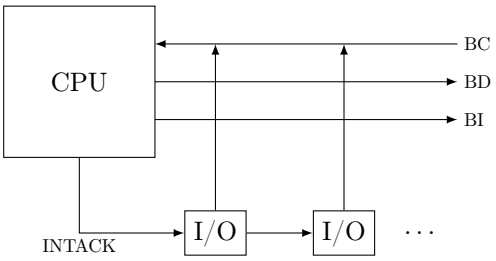
Ogni operazione di lettura effettuata con il bus richiede circa 10 cicli di clock.

```
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 ≈ 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 è sempre a 1 (negato a 0) 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.

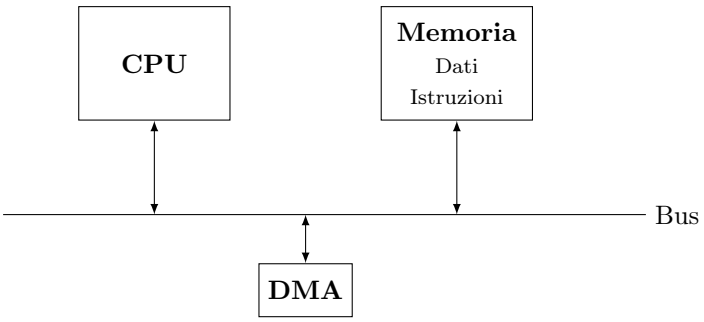


Figura 8: Schema di un sistema con DMA

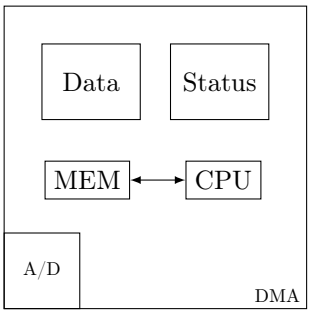


Figura 9: Struttura del DMA

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.

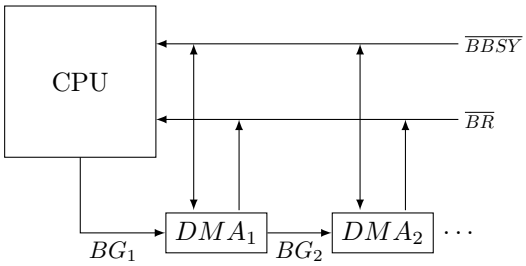


Figura 10: Schema di un sistema con bus

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

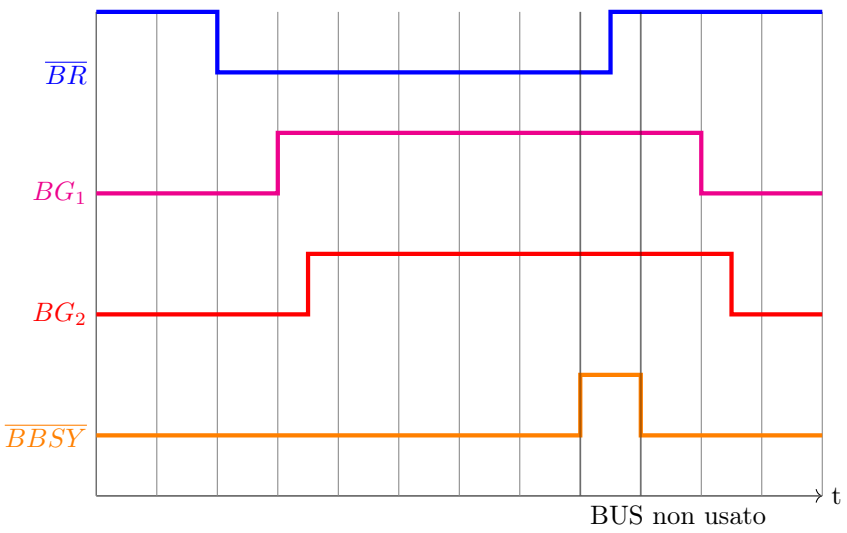


Figura 11: Segnali di controllo DMA

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**

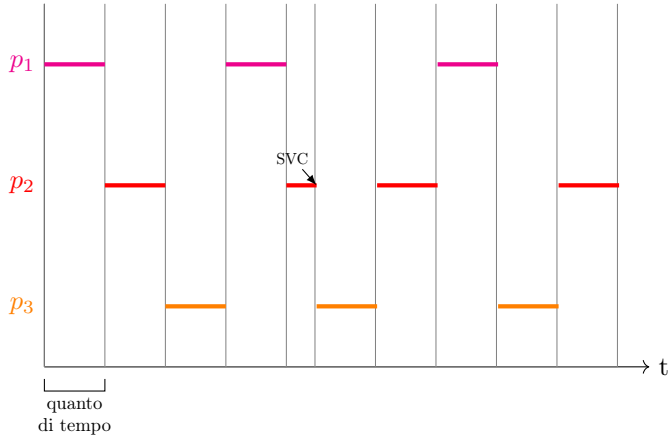


Figura 12: Time sharing tra i processi

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.

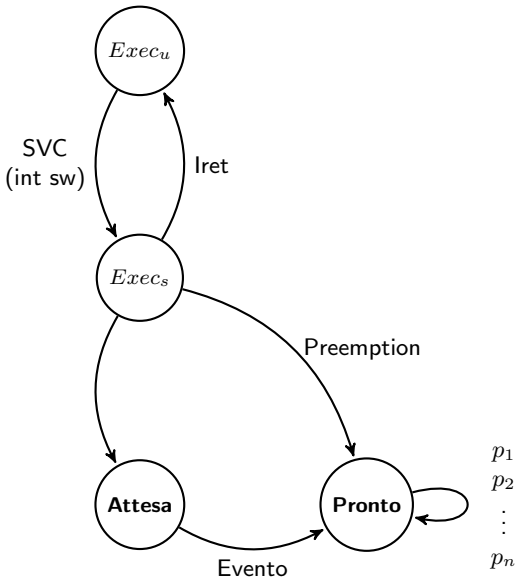


Figura 13: Flusso di esecuzione delle istruzioni

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
```

Il comando `renice` permette di modificare la priorità di un processo software.

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.

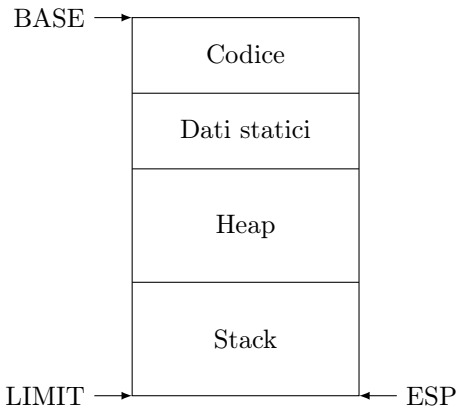


Figura 14: Struttura di un file binario

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:

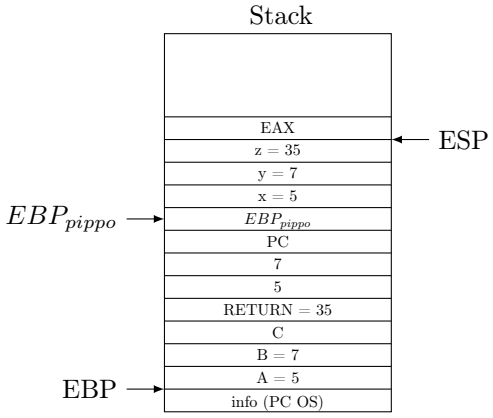


Figura 15: Stack con variabili

Di seguito il codice assembly per la gestione dello stack:

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

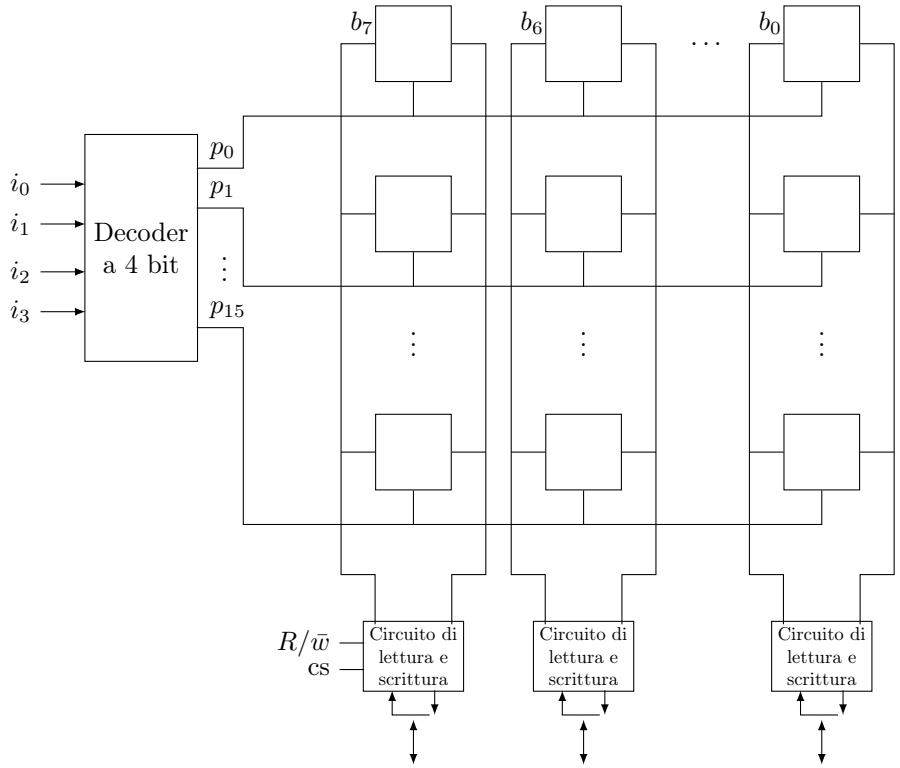
```
PUSHL %EAX
MOVL $12(%EBP), %EAX
MOVL %EAX, -4(%EBP)
...
MOVL %EAX, $16(%EBP)

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



9.1 Static RAM (SRAM)

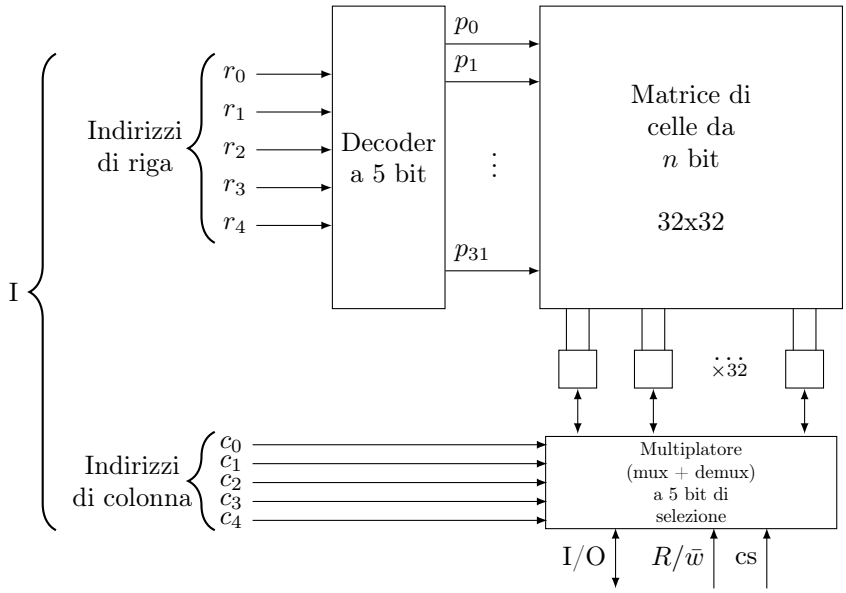
La SRAM è una memoria statica, ovvero non ha bisogno di essere rinfrescata per mantenere i dati. Questo tipo di memoria è più veloce, ma occupa più spazio ed è più costosa.

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

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

Si avrà una SRAM che sarà una matrice da m bit 32×32



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

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.

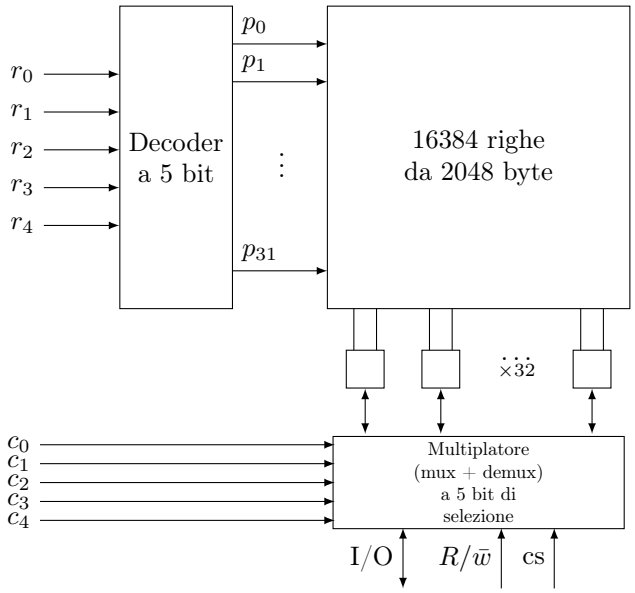
9.3 Syncronous DRAM (SDRAM)

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

$$256Mbit \;\; 32Mbit \times 8 = 16K \times 16K$$

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


r_0
r_1
r_2
r_3
r_4
Decoder
a 5 bit
p_0
p_1
p_{31}
16384 righe
da 2048 byte
×32
c_0
c_1
c_2
c_3
c_4
Multiplatore
(mux + demux)
a 5 bit di
selezione
I/O
$R/\bar{w}$
cs