

Il Processore z64

Introduzione all'Architettura Hardware e Programmazione Software



SAPIENZA
UNIVERSITÀ DI ROMA

Alessandro Pellegrini

Architettura dei Calcolatori Elettronici
Sapienza, Università di Roma

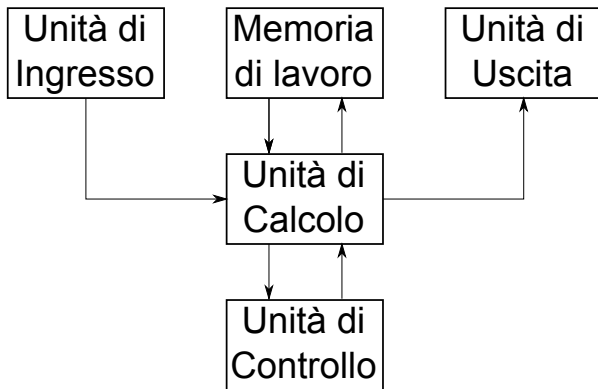
A.A. 2018/2019

Architetture Hardware

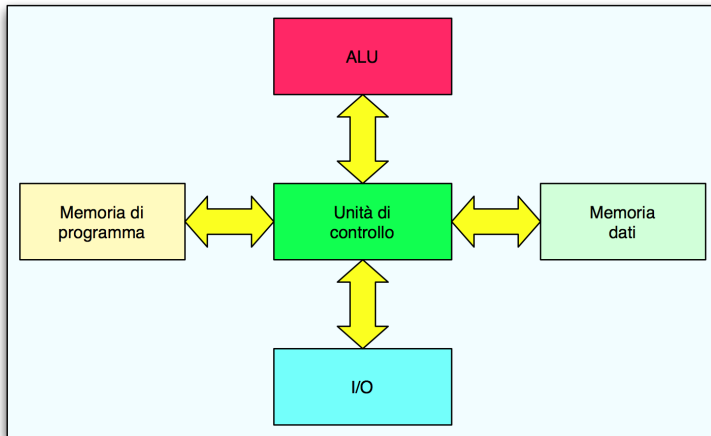
Un'architettura hardware descrive le relazioni tra le parti di un elaboratore elettronico.

- Ad esempio:
 - Come accede un processore alla memoria?
 - Come fa il processore a dialogare con le periferiche?
 - Come interagiscono dei processori tra loro?
- La prima “architettura” che compare nella storia si trova nella corrispondenza tra Charles Babbage ed Ada Lovelace, che descrive una macchina analitica
- Il più famoso modello di architettura è quello di John von Neumann, del 1945, per descrivere l'organizzazione logica degli elementi dell'EDVAC

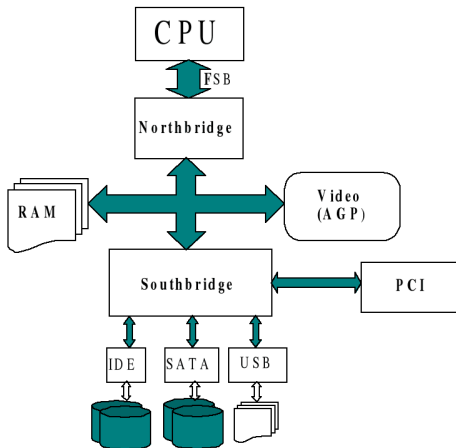
Architettura di von Neumann



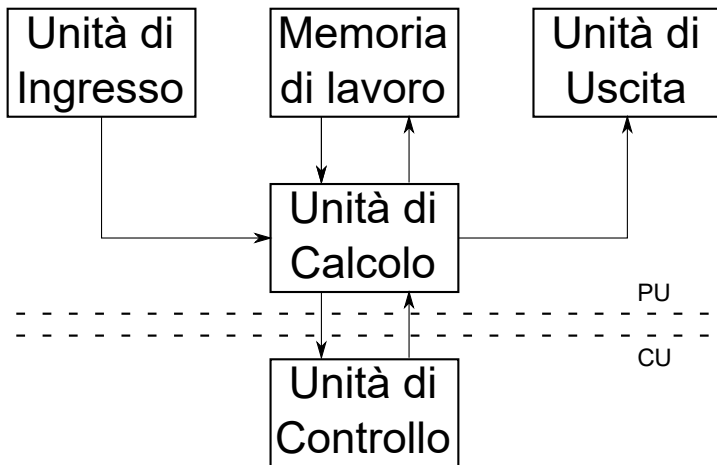
Architettura Harvard



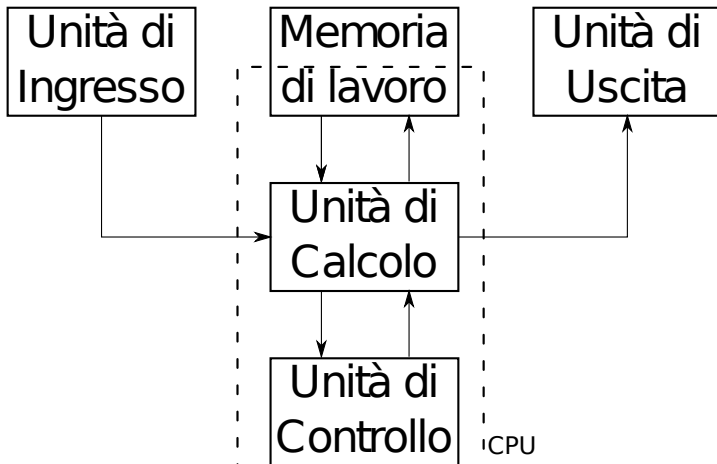
Architettura IBM



Architetture di von Neumann Rivisitata: PU e CU

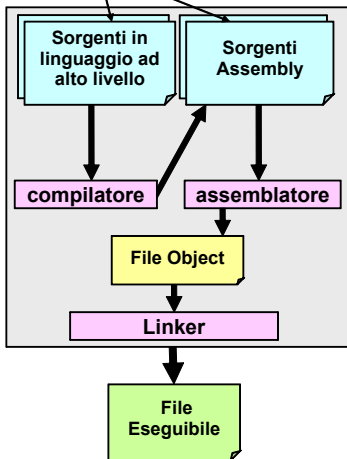


Architetture di von Neumann Rivisitata: CPU



Generazione di un programma

File creati dall'utente



$a = b + c$

```
movw b, %ax
movw c, %bx
addw %ax, %bx
movw %bx, a
```

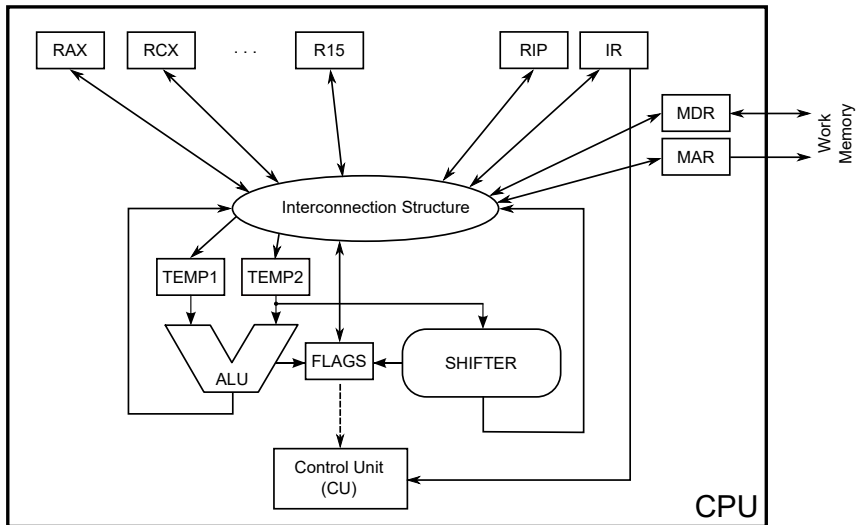
```
000101..0101001
1011101..010100
01011..11101010
010..1110101010
```

Generazione di un programma (2)

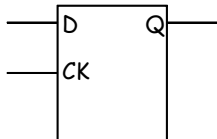
Il processo di generazione di un programma si snoda in tre fasi principali:

- **Compilazione:** Generazione automatica di codice Assembly da uno o più file scritti in linguaggi di alto livello
- **Assemblazione:** Generazione di uno o più file oggetto contenenti le rappresentazioni binarie delle istruzioni, a partire da codice Assembly
- **Collegamento:** Traduzione degli indirizzi e risoluzione di tutti i collegamenti tra *funzioni*, *variabili*, *etichette*.

II Processore z64

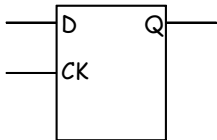


Memorizzare i dati nel processore

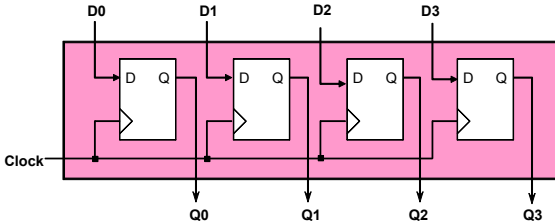


- Un flip-flop è un circuito basato su un bistabile
- Ha un ingresso dati, ed uno di abilitazione
- Usato come unità elementare di memorizzazione
- Combinandone più insieme, si possono costruire dei registri

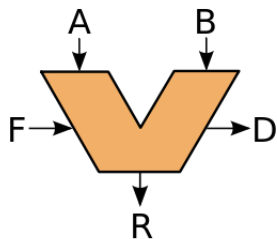
Memorizzare i dati nel processore



- Un flip-flop è un circuito basato su un bistabile
- Ha un ingresso dati, ed uno di abilitazione
- Usato come unità elementare di memorizzazione
- Combinandone più insieme, si possono costruire dei registri



Operazioni: la ALU

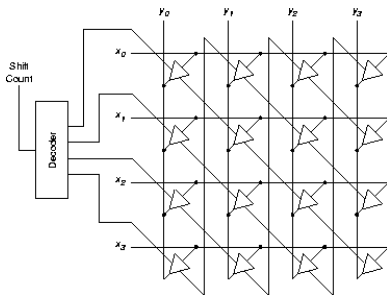


A, B: operandi
F: segnali dalla CU
D: segnali di stato
R: risultato

- L'Unità Logico-Aritmetica (ALU) è un componente fondamentale di ogni processore
- Essa implementa la maggior parte di operazioni aritmetiche (somma, sottrazione, moltiplicazione, ...) e logiche (and, or, xor, ...)
- L'ALU utilizza i dati provenienti dai registri del processore, li processa e mette a disposizione il risultato sulla linea di uscita (per essere memorizzato all'interno di un altro registro del processore)

Operazioni: lo Shifter

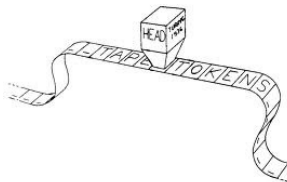
Uno shifter è un circuito digitale che può traslare i bit di una parola di un numero specificato di posizioni in un solo colpo di clock



z64: l'Unità di Processamento (PU)

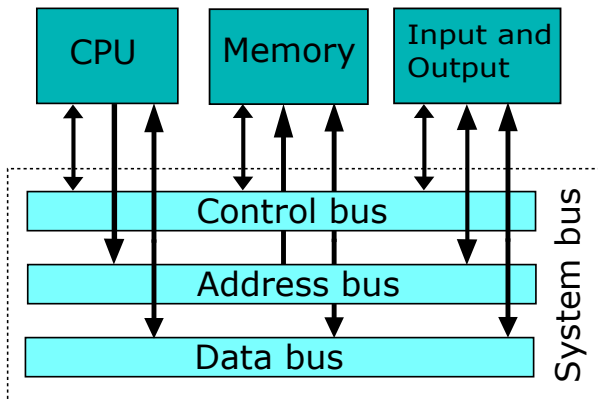
- Registri (basati su flip flop con segnali di enable)
 - Fondamentali
 - Di utilizzo generale
 - Nascosti
- Dispositivi di Calcolo
 - Shifter
 - ALU (somma e sottrazione)
- MUX
- Decodificatori
- Struttura di interconnessione: BUS

Modello di memoria



- La memoria può essere vista come un lunghissimo nastro, diviso in celle (*modello di memoria piatta*)
- Ogni cella di memoria viene identificata da un numero intero progressivo (*indirizzo*)
- Ciascuna cella ha dimensione pari ad un byte (8 bit)
- Virtualmente, una testina si muove sul nastro, per leggere o scrivere i dati nelle celle
- Una cella viene scritta necessariamente per intero

Trasferire dei dati: il BUS



Processamento delle istruzioni

Il processamento delle istruzioni (ciclo istruzione) prevede tre fasi:

- *Fetch*: l'istruzione viene prelevata dalla memoria
- *Decodifica*: viene identificato quale microprogramma dell'unità di controllo (CU) deve essere attivato
- *Esecuzione*: la *semantica* dell'istruzione viene effettivamente implementata. Può richiedere un numero differente di cicli macchina (se, ad es., si richiede l'interazione con la memoria o dispositivi di I/O).

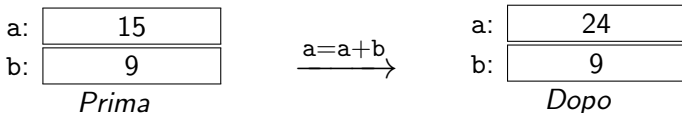
Un semplice esempio

Consideriamo l'istruzione $a=a+b$ espressa in linguaggio di alto livello:

Somma

Memorizza nella variabile di nome a la somma dei valori contenuti nelle variabili di nome a e b

Nota: le variabili sono individuate da un nome simbolico deciso precedentemente nel programma



Un semplice esempio (2)

- Per eseguire questa istruzione è necessario:
 - Stabilire dove sono memorizzati i valori da sommare
 - Stabilire dove va scritto il risultato dell'operazione
 - Decidere quale operazione svolgere
- Nello z64, gli operandi sono memorizzati nei registri interni alla CPU (registri di uso generale)
- Il formato dell'istruzione (nella sintassi AT&T) è:

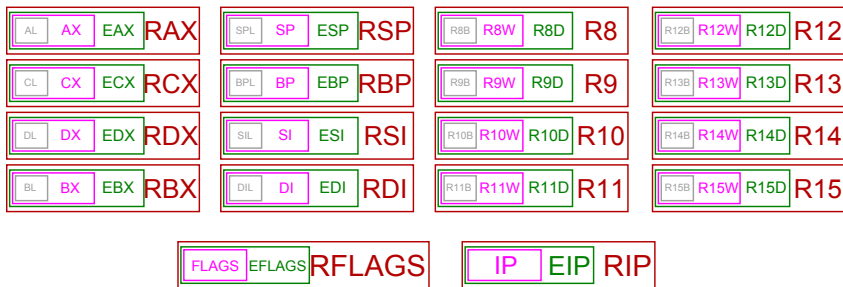
ADDs <sorgente>, <destinazione>

- s può essere B, W, L, Q—il nome dei registri varia di conseguenza
- Il campo destinazione è un registro che contiene il valore iniziale di un operando e sarà modificato:

ADDW %ax, %bx

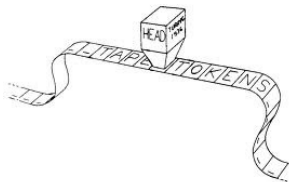
(somma %ax con %bx—a 16 bit—e pone il risultato in %bx)

Registri Fisici e Registri Virtuali



■ 64-bit Register ■ 32-bit Register ■ 16-bit Register ■ 8-bit Register

Tenere traccia dell'esecuzione: l'Instruction Pointer

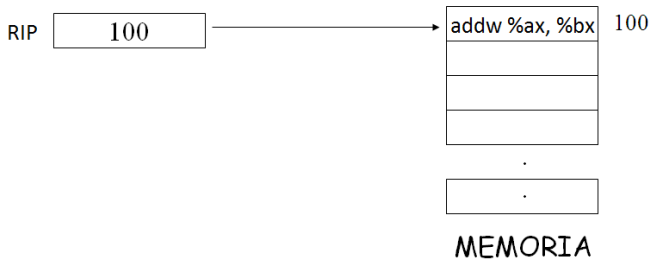


- Nell'architettura di von Neumann dati e programma si trovano nella stessa memoria
- Per conoscere l'indirizzo della prossima istruzione da eseguire, il processore utilizza un registro fondamentale: l'*Instruction Pointer* (RIP)
- Ogni volta che un'istruzione viene eseguita, RIP viene aggiornato automaticamente per puntare a quella successiva
- Le istruzioni devono quindi essere contigue in memoria
- Il flusso d'esecuzione può non seguire il loro ordine in memoria (salti)

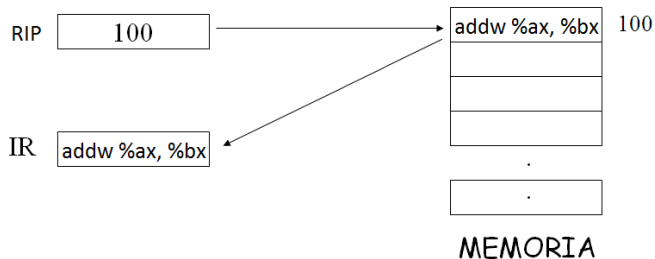
Esecuzione di un'istruzione: l'Instruction Register

- Accedere alla memoria è un'operazione costosa
 - I registri dei processori sono realizzati con tecnologia molto più veloce, ma più costosa
 - La memoria, avendo dimensione più grande, è realizzata con tecnologie più economiche ma meno veloci
- L'esecuzione di un'istruzione è divisa in due fasi (decodifica, esecuzione)
- La fase di esecuzione può richiedere un numero variabile di sottofasi, in cui può essere necessario accedere nuovamente in memoria
- Accedere ripetutamente in memoria è troppo costoso
- Prima di essere eseguita, l'istruzione viene copiata (fase di fetch) in un registro fondamentale: l'*Instruction Register* (IR)

Esecuzione di un'istruzione

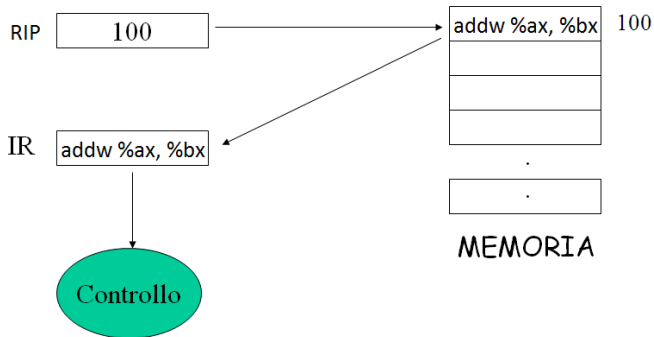


Esecuzione di un'istruzione: fase di fetch



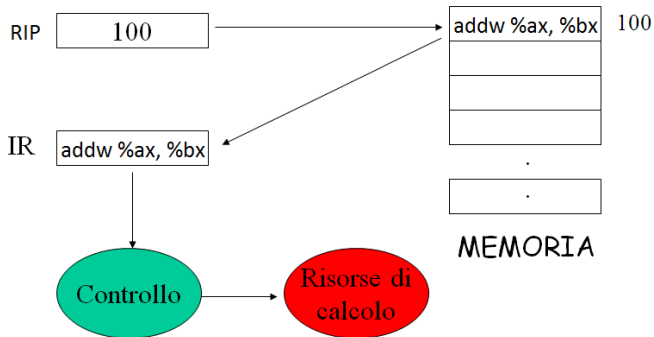
Durante la fase di fetch, l'istruzione viene copiata dalla memoria di lavoro nell'IR

Esecuzione di un'istruzione: fase di decodifica



Durante la fase di decodifica, l'istruzione viene interpretata per identificare il microprogramma che la possa eseguire

Esecuzione di un'istruzione: fase di esecuzione



La CU pilota la PU per implementare la semantica dell'istruzione

CPU come interprete

- L'esecuzione di un programma può essere vista come la ripetizione dei seguenti passi che interpretano ed eseguono le istruzioni del programma contenute in memoria.

Ciclo Istruzione

fetch: (RIP) \rightarrow IR

incrementa RIP

esegui istruzione in IR

vai al passo fetch

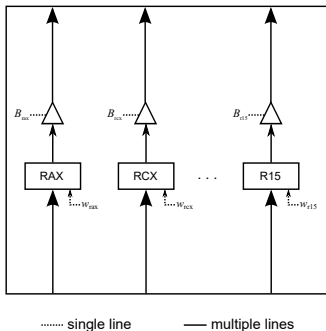
- La CPU interpreta le istruzioni che sono presenti nel suo IR
- L'esecuzione di un'istruzione può modificare il contenuto di RIP
- Tale schema è semplificato: per interagire con l'esterno, o per gestire situazioni anomale, tale ciclo deve poter essere interrotto

z64: BUS interno

- Usato per il collegamento dei registri interni
- Operazioni che caratterizzano il BUS:
 - *Ricezione dati*: i bit presenti sul bus sono memorizzati in un registro
 - *Trasmissione dati*: il contenuto di un registro è posto sul bus
- Al più un solo registro può scrivere sul bus
 - Utilizzo di segnali di controllo opportunamente generati:
 - *Write Enable*
 - *Buffer Three-state*

z64: BUS interno e segnali di controllo

Si può avere una sola scrittura per volta sul bus (controllo mediante B_i o W_i), pertanto con n registri, si avranno $2n$ segnali di controllo:



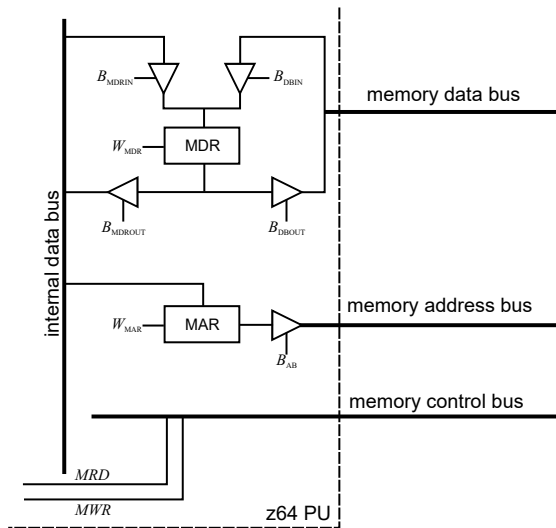
$W_i = 1$: si può leggere dal bus

$B_i = 1$: si può scrivere sul bus

z64: Interazione con la memoria

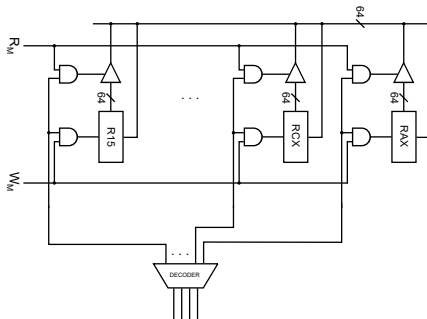
- La memoria contiene sia i dati che le istruzioni (architettura di von Neumann)
- Può essere sia scritta che letta
- È necessario quindi:
 - Prelevare istruzioni
 - Leggere dati
 - Scrivere dati
- È necessario inoltre instradare opportunamente i dati ricevuti dalla memoria verso i registri e viceversa

z64: Interazione con la memoria



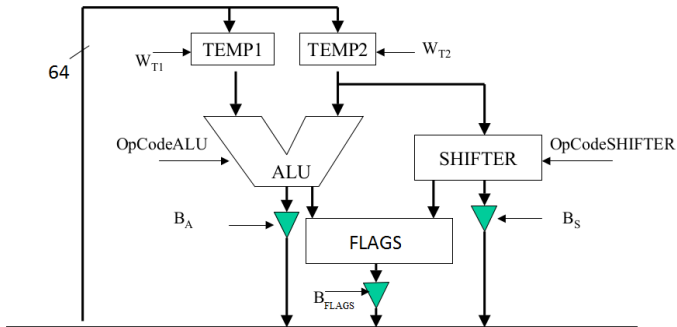
z64: Banco dei registri

- Insieme di 16 registri di uso generale
- Sono controllati mediante segnali di abilitazione per:
 - Scrittura del registro (W_i)
 - Lettura e consegna sul bus interno del contenuto del registro (R_i)






Il registro FLAGS

- Il registro FLAGS è un registro fondamentale
- Contiene informazioni sull'esito dell'ultima operazione (bit di stato)
- Contiene dei *flag* per variare il modo d'esecuzione (bit di controllo)
- Usato anche come ingresso per alcune operazioni (es: salti condizionati)



Il registro FLAGS

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	D	I		S	Z				P	1	C
				F	F	F		F	F				F		F

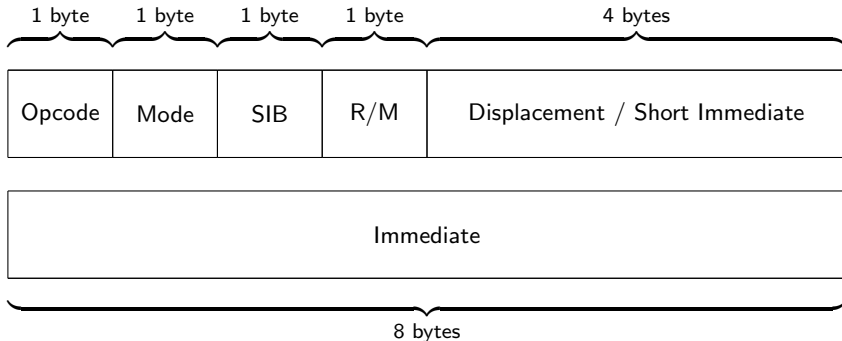
-  Riservato (da non modificare)
-  Control Flags
-  Status Flags

- **carry** (CF): vale 1 se l'ultima operazione ha prodotto un riporto
- **parity** (PF): vale 1 se nel risultato dell'ultima operazione c'è un numero pari di 1
- **zero** (ZF): vale 1 se l'ultima operazione ha come risultato 0
- **sign** (SF): vale 1 se l'ultima operazione ha prodotto un risultato negativo
- **overflow** (OF): vale 1 se il risultato dell'ultima operazione supera la capacità di rappresentazione
- **interrupt enable** (IF): indica se c'è la possibilità di interrompere l'esecuzione del programma in corso
- **direction** (DF): modifica il comportamento delle operazioni su stringhe

Rappresentare delle operazioni: le Istruzioni

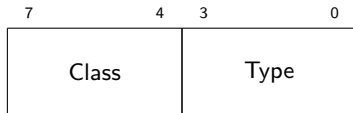
- Sono organizzate in otto classi:
 0. Istruzioni di controllo dell'hardware
 1. Spostamento di dati
 2. Aritmetiche (su interi) e logiche
 3. Rotazione e shift
 4. Operazioni sui bit di FLAGS
 5. Controllo del flusso d'esecuzione del programma
 6. Controllo condizionale del flusso d'esecuzione del programma
 7. Ingresso/Uscita di dati

Rappresentare le operazioni: le Istruzioni (2)



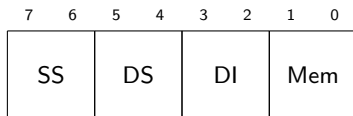
Nel formato dello z64, le istruzioni hanno un formato variabile

Il campo Opcode



- Class è un codice di 4 bit che identifica la famiglia di istruzioni cui appartiene quella corrente
- Type è un codice di quattro bit che identifica la precisa istruzione nella famiglia
- Questa differenziazione consente di realizzare microcodice più ottimizzato

Il campo Mode



- SS e DS contengono rispettivamente la codifica della dimensione dell'operando sorgente e destinazione
- DI è un campo di 2 bit che indica o meno la presenza di un displacement e di un immediato
- Mem indica quali degli operandi (sorgente o destinazione) sono da considerarsi operandi in memoria

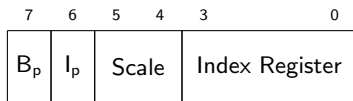
Il campo Mode (2)

Campo	Valore	Significato
SS	00	La sorgente è un byte
	01	La sorgente è una word
	10	La sorgente è una longword
	11	La sorgente è una quadword
DS	00	La destinazione è un byte
	01	La destinazione è una word
	10	La destinazione è una longword
	11	La destinazione è una quadword
DI	00	Spiazzamento non utilizzato, immediato non presente
	01	Immediato presente
	10	Spiazzamento utilizzato
	11	Spiazzamento utilizzato, immediato presente
Mem	00	Sia la sorgente che la destinazione sono registri
	01	La sorgente è un registro, la destinazione è in memoria
	10	La sorgente è in memoria, la destinazione è un registro
	11	Condizione impossibile (genera un'eccezione a runtime)

z64: Modalità di Indirizzamento in Memoria

$$\left[\left(\begin{array}{c} \text{AX} \\ \text{BX} \\ \text{CX} \\ \text{DX} \\ \text{SP} \\ \text{BP} \\ \text{SI} \\ \text{DI} \\ \text{R8} \\ \text{R9} \\ \text{R10} \\ \text{R11} \\ \text{R12} \\ \text{R13} \\ \text{R14} \\ \text{R15} \end{array} \right) \right] + \left[\left(\begin{array}{c} \text{AX} \\ \text{BX} \\ \text{CX} \\ \text{DX} \\ \text{SP} \\ \text{BP} \\ \text{SI} \\ \text{DI} \\ \text{R8} \\ \text{R9} \\ \text{R10} \\ \text{R11} \\ \text{R12} \\ \text{R13} \\ \text{R14} \\ \text{R15} \end{array} \right) * \left\{ \begin{array}{c} 1 \\ 2 \\ 4 \\ 8 \end{array} \right\} \right] + [\text{spiazzamento}]$$

Il campo SIB (Scala, Indice, Base)

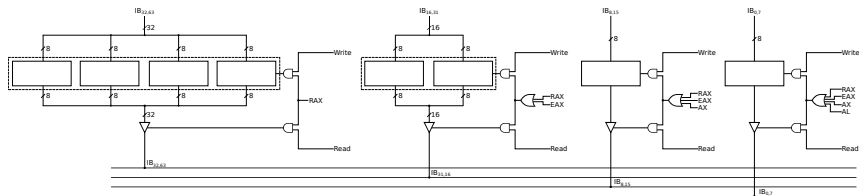


- B_p e I_p indicano se la modalità di indirizzamento in memoria dell'istruzione corrente utilizza una base e/o un indice
- Se I_p == 1, il campo Index mantiene la codifica binaria del registro indice
- Scale mantiene il valore della scala, i cui valori leciti sono 1 (codificato come 00), 2 (codificato come 01), 4 (codificato come 10), and 8 (codificato come 11)

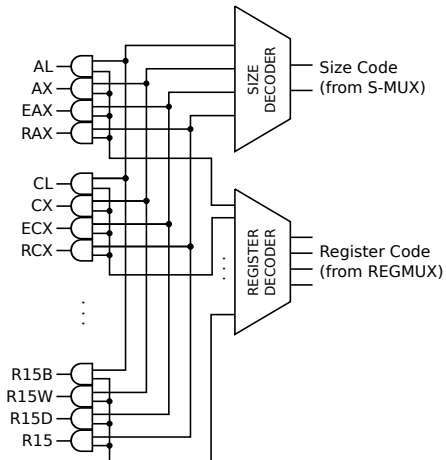
z64: i Registri Fisici

Nome	Codifica	Uso Comune
RAX	0000	Registro Accumulatore
RCX	0001	Registro Contatore
RDX	0010	Registro Dati
RBX	0011	Registro Base
RSP	0100	Stack Pointer
RBP	0101	Base Pointer
RSI	0110	Registro Sorgente
RDI	0111	Registro Destinazione
R8	1000	Registro di uso generale
R9	1001	Registro di uso generale
R10	1010	Registro di uso generale
R11	1011	Registro di uso generale
R12	1100	Registro di uso generale
R13	1101	Registro di uso generale
R14	1110	Registro di uso generale
R15	1111	Registro di uso generale

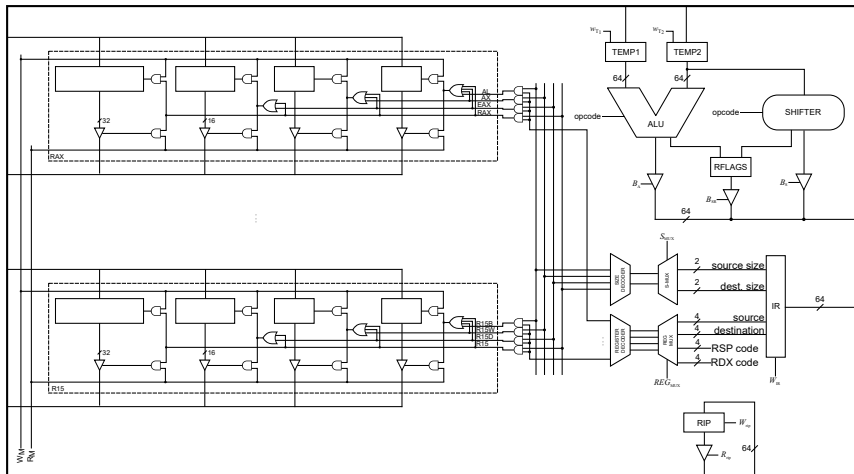
z64: i Registri Virtuali



z64: i Registri Virtuali



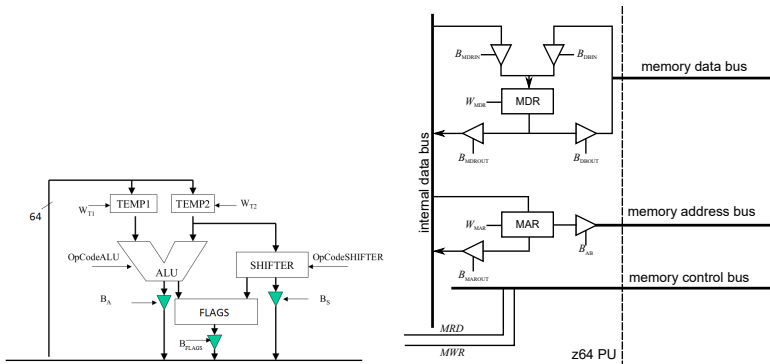
z64: i Registri Virtuali



Come si calcola un indirizzo?

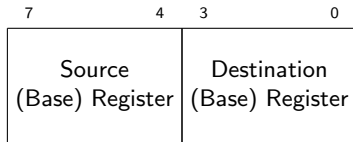
- La determinazione di un indirizzo di memoria è un'operazione complessa nel processore z64
- È necessario fare affidamento sulla PU per calcolare questo indirizzo
 - Introduzione di hardware dedicato
 - Utilizzo dell'hardware già presente
- Qual è la soluzione più conveniente?

Lettura del registro MAR



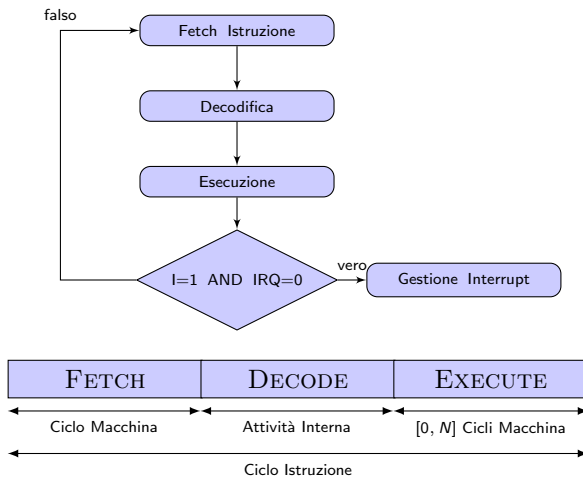
$$base + idx \cdot scale + offset$$

Il campo R/M



- Questo campo ha spazio per mantenere esattamente due codifiche di registri
- L'interpretazione di questi campi dipende dai valori dei sottocampi di Mem e di B_p di SIB
- I registri possono quindi essere interpretati come registri semplici oppure come registri base

Ciclo istruzione



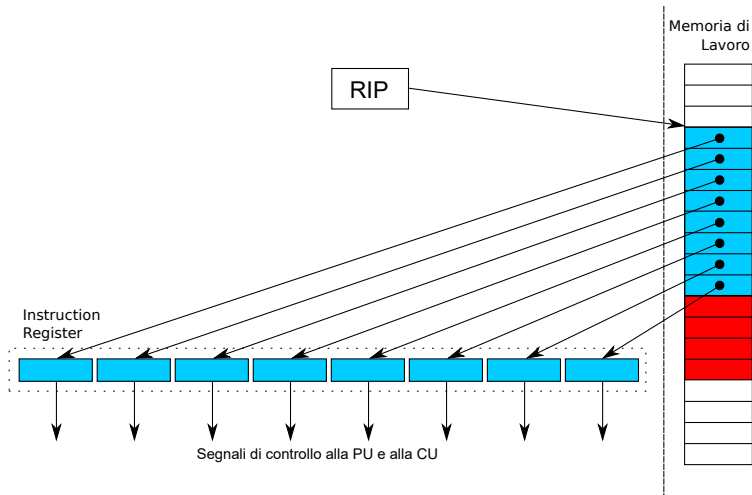
Ciclo istruzione (2)

Ciclo istruzione del microprocessore:

Passo	Operazione
1	$MAR \leftarrow RIP$
2	$MDR \leftarrow (MAR)$
3	$IR \leftarrow MDR; RIP \leftarrow RIP + 8$
4	Decodifica istruzione (<i>Decode</i>)
5	Esecuzione istruzione (<i>Execute</i>)
6	Torna al passo 1

- Il tempo (in *nsec*) necessario ad eseguire l'intera sequenza delle operazioni costituisce il *ciclo di istruzione della CPU*
- La CPU esegue tale ciclo continuamente, finché non carica nel registro IR (ed esegue) l'istruzione HLT
- Un'eccezione a questa regola (*interruzione*) si verifica quando un dispositivo esterno chiede di trasferire dati dalla/alla CPU

Ciclo istruzione: Fetch



Alcune istruzioni Assembly

- `movb %al, %bl`: copia il contenuto del byte meno significativo di RAX in RBX
- `movw %ax, (%rdi)`: copia il contenuto dei 2 byte meno significativi di RAX nei due byte di memoria il cui indirizzo iniziale è memorizzato in RDI
- `movl (%rsi), %eax`: copia nei 4 byte meno significativi di RAX il contenuto dei 4 byte di memoria il cui indirizzo è specificato in %rsi

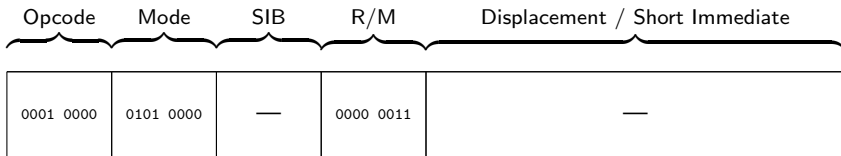
Alcune istruzioni Assembly (2)

- `movq (%rsi, %rcx, 8), %rax`: copia nel registro RAX il contenuto degli 8 byte di memoria il cui indirizzo iniziale è calcolato come $RSI + RCX \cdot 8$
- `subl %eax, %edx`: sottrai il contenuto dei 4 byte meno significativi di RAX dai 4 byte meno significativi di RDX e aggiorna il contenuto dei 4 byte meno significativi di RDX.
- `addb $d, %ax`: somma al byte meno significativo di RAX la quantità costante `d`

Traduzione istruzioni Assembly in linguaggio macchina

`movw %ax, %bx`

- Non si ha alcun accesso in memoria
- Sorgente e destinazione sono entrambi di 2 byte
- Non è coinvolta alcuna costante

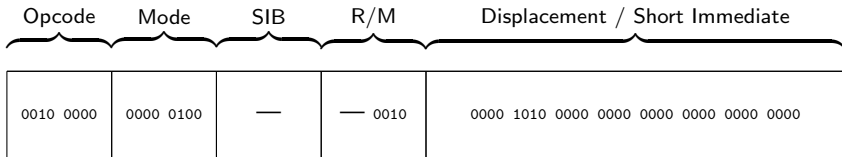


10 A0 00 03 00 00 00 00

Traduzione istruzioni Assembly in linguaggio macchina

addb \$0xa, %d1

- Si utilizza una costante numerica come operando
- Non è presente uno spiazzamento

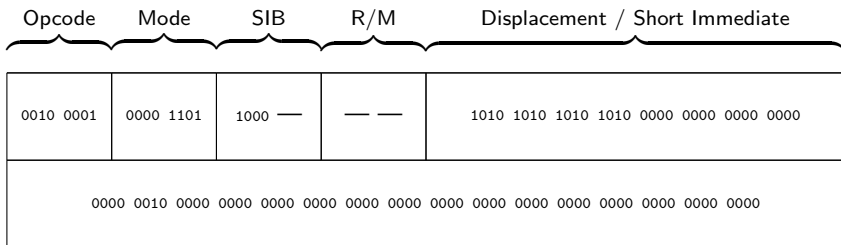


20 04 00 02 0A 00 00 00

Traduzione istruzioni Assembly in linguaggio macchina

subb \$0x2, 0xAAAA

- Si utilizza una costante numerica come operando
- È presente uno spiazzamento

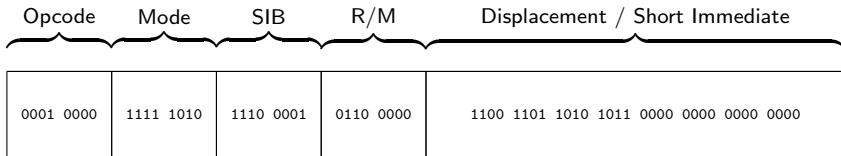


21 0D 80 02 AA AA 00 00 02 00 00 00 00 00 00 00

Traduzione istruzioni Assembly in linguaggio macchina

```
movq 0xabcd(%rsi, %rcx, 4), %rax
```

- La sorgente è un operando in memoria
- Per accedere in memoria vengono usati scala, indice, base e spiazzamento



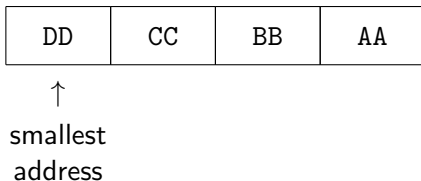
10 F8 E1 60 CD AB 00 00

Memory Endianness—Ordine dei Byte

- L'ordine dei byte descrive la modalità utilizzata dai calcolatori per immagazzinare in memoria dati di dimensione superiore al byte
- La differenza dei sistemi è data dall'ordine con il quale i byte costituenti il dato da immagazzinare vengono memorizzati:
 - **litte-endian**: memorizzazione che inizia dal byte meno significativo per finire col più significativo (usato dai processori Intel)
 - **big-endian**: memorizzazione che inizia dal byte più significativo per finire col meno significativo (Network-byte order)
 - **middle-endian**: ordine dei byte né crescente né decrescente (es: 3412, 2143)
- La differenza si rispecchia nel *network-byte order* vs *host order*

Effetti della Little-Endianness

- Il processore z64 accede in memoria secondo lo schema little-endian
- Valori multibyte sono memorizzati con il loro byte meno significativo all'indirizzo più basso
- Il valore 0xAABBCCDD è posto nel layout di memoria come segue:



Effetti della Little-Endianness (2)

- Cosa succede se memorizziamo due interi da 4 byte in modo consecutivo in memoria?
- Prendiamo ad esempio `0x00cf9200` e `0x0000ffff`

Effetti della Little-Endianness (2)

- Cosa succede se memorizziamo due interi da 4 byte in modo consecutivo in memoria?
- Prendiamo ad esempio 0x00cf9200 e 0x0000ffff

00	92	CF	00	FF	FF	00	00
----	----	----	----	----	----	----	----

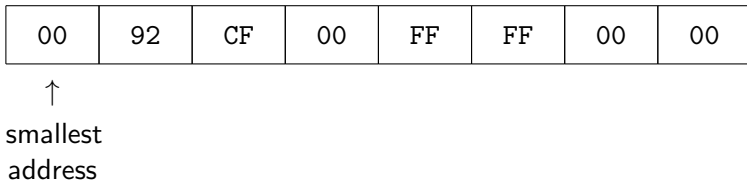
↑
smallest
address

Effetti della Little-Endianness (2)

- Cosa succede se memorizziamo due interi da 4 byte in modo consecutivo in memoria?
- Prendiamo ad esempio `0x00cf9200` e `0x0000ffff`

Effetti della Little-Endianness (2)

- Cosa succede se memorizziamo due interi da 4 byte in modo consecutivo in memoria?
- Prendiamo ad esempio 0x00cf9200 e 0x0000ffff



- I byte di ogni singolo intero sono scambiati, ma non l'ordine dei due interi!

Effetti della Little-Endianness (3)

- Cosa succede se memorizziamo due interi a 4 byte, seguiti da un intero a 8 byte?
- Prendiamo ad esempio 0x00cf9200, 0x0000ffff e 0x00cf92000000ffff

Effetti della Little-Endianness (3)

- Cosa succede se memorizziamo due interi a 4 byte, seguiti da un intero a 8 byte?
- Prendiamo ad esempio 0x00cf9200, 0x0000ffff e 0x00cf92000000ffff

smallest
address



00	92	CF	00	FF	FF	00	00
FF	FF	00	00	00	92	CF	00

La Little-Endianness aiuta in qualche modo?

- Apparentemente questa rappresentazione dei dati è una complicazione
- La CPU deve infatti “ribaltare” ogni volta i dati, o le componenti della PU devono lavorare a byte invertiti
- Cosa succede con i *cast*?

La Little-Endianness aiuta in qualche modo?

- Apparentemente questa rappresentazione dei dati è una complicazione
- La CPU deve infatti “ribaltare” ogni volta i dati, o le componenti della PU devono lavorare a byte invertiti
- Cosa succede con i *cast*?
- L'indirizzo di memoria non cambia, se voglio accedere ad una sottoporzione del dato!

Le istruzioni dello z64

Le seguenti convenzioni vengono utilizzate per rappresentare gli operandi delle istruzioni:

- B** — L'operando è un registro di uso generale, un indirizzo di memoria o un valore immediato. In caso di un indirizzo di memoria, qualsiasi combinazione delle modalità di indirizzamento è lecita. In caso di un immediato, la sua posizione dipende dalla possibile presenza dello spiazzamento e dalla sua dimensione.
- E** — L'operando è un registro di uso generale, o un indirizzo di memoria. In caso di un indirizzo di memoria, qualsiasi combinazione delle modalità di indirizzamento è lecita.
- G** — L'operando è un registro di uso generale.
- K** — L'operando è una costante numerica non segnata di valore fino a $2^{32} - 1$
- M** — L'operando è una locazione di memoria, codificata come uno spiazzamento a partire dal contenuto del registro RIP dopo l'esecuzione della fase di fetch

Classe 0: Istruzioni di controllo hardware

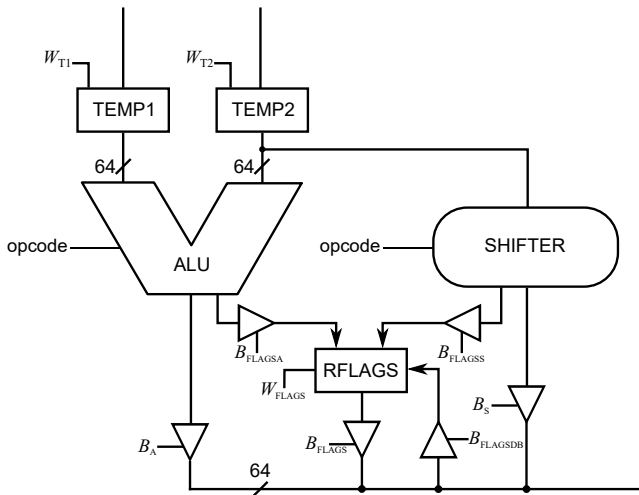
- Le istruzioni di controllo hardware consentono di modificare lo stato della CPU, oppure eseguono istruzioni particolari
- Non hanno bisogno di operandi particolari per la loro esecuzione

Tipo	Mnemonico	Operandi	O	S	Z	P	C	Descrizione
1	hlt	–	–	–	–	–	–	Mette la CPU in modalità di basso consumo energetico, finché non viene ricevuta l'interruzione successiva
2	nop	–	–	–	–	–	–	Nessuna operazione
3	int	–	–	–	–	–	–	Chiama esplicitamente un gestore di interruzioni

Classe 1: Istruzioni di movimento dei dati

Tipo	Mnemonico	Operandi	O	S	Z	P	C	Descrizione
0	mov	B, E	-	-	-	-	-	Fa una copia di B in E
1	movsX	E, G	-	-	-	-	-	Fa una copia di E in G con estensione del segno
2	movzX	E, G	-	-	-	-	-	Fa una copia di E in G con estensione dello zero
3	lea	E, G	-	-	-	-	-	Valuta la modalità di indirizzamento, salva il risultato in G
4	push	E	-	-	-	-	-	Copia il contenuto di E sulla cima dello stack
5	pop	E	-	-	-	-	-	Copia il contenuto della cima dello stack in E
6	pushf	-	-	-	-	-	-	Copia sulla cima dello stack il registro FLAGS
7	popf	-	-	-	-	-	-	Copia nel registro FLAGS il contenuto della cima dello stack
8	movs	-	-	-	-	-	-	Esegue una copia memoria-memoria
9	stos	-	-	-	-	-	-	Imposta una regione di memoria ad un dato valore

popf: come supportarne l'esecuzione?



Estensioni del segno

Istruzione	Tipo di conversione
<code>movsbw %al, %ax</code>	Estendi il segno da byte a word
<code>movsbl %al, %eax</code>	Estendi il segno da byte a longword
<code>movsbq %al, %rax</code>	Estendi il segno da byte a quadword
<code>movswl %ax, %eax</code>	Estendi il segno da word a longword
<code>movswq %ax, %rax</code>	Estendi il segno da word a quadword
<code>movslq %eax, %rax</code>	Estendi il segno da longword a quadword

L'istruzione `movzX` supporta le stesse combinazioni di suffissi
I nomi dei registri virtuali devono essere coerenti con i suffissi

Classe 2: Istruzioni logico/aritmetiche

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
0	add	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Memorizza in E il risultato di $E + B$
1	sub	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Memorizza in E il risultato di $E - B$
2	adc	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Memorizza in D il risultato di $E + B + CF$
3	sbb	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Memorizza in D il risultato di $E - (B + \text{neg}(CF))$
4	cmp	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Confronta i valori di B ed E calcolando $E - B$, il risultato viene poi scartato
4	test	B, E	⇕ ⇕ ⇕ ⇕ ⇕	Calcola l'and logico bit a bit di B ed E, il risultato viene poi scartato
5	neg	E	⇕ ⇕ ⇕ ⇕ ⇕	Rimpiazza il valore di E con il suo complemento a 2

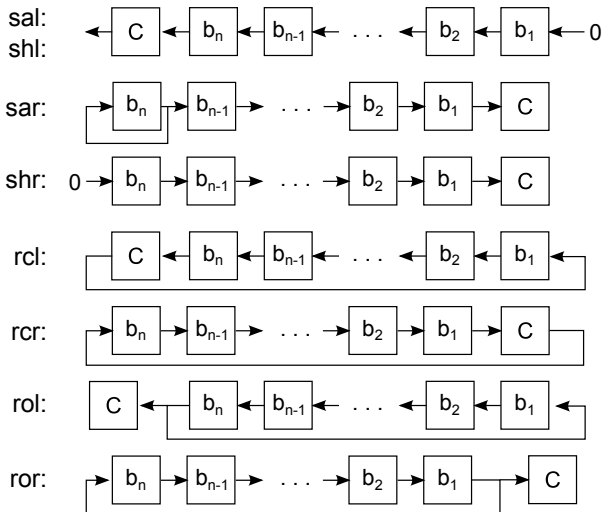
Classe 2: Istruzioni logico/aritmetiche (2)

Tipo	Mnemonico	Operandi	O S Z P C	Descrizione
6	and	B, E	0 ⇕ ⇕ ⇕ 0	Memorizza in E il risultato dell'and bit a bit tra B ed E
7	or	B, E	0 ⇕ ⇕ ⇕ 0	Memorizza in E il risultato dell'or bit a bit tra B ed E
8	xor	B, E	0 ⇕ ⇕ ⇕ 0	Memorizza in E il risultato dello xor bit a bit tra B ed E
9	not	E	0 ⇕ ⇕ ⇕ 0	Rimpiazza il valore di E con il suo complemento a uno
10	bt	K, E	- - - - ⇕	Imposta CF al valore del K-simo bit di E (bit testing)

Classe 3: Istruzioni di rotazione e shift

Tipo	Mnemonico	Operandi	O	S	Z	P	C	Descrizione
0	sal	K, G	⇕	⇕	⇕	⇕	⇕	Moltiplica per 2, K volte
1	sal	G	⇕	⇕	⇕	⇕	⇕	Moltiplica per 2, RCX volte
0	shl	K, G	⇕	⇕	⇕	⇕	⇕	Moltiplica per 2, K volte
1	shl	G	⇕	⇕	⇕	⇕	⇕	Moltiplica per 2, RCX volte
2	sar	K, G	⇕	⇕	⇕	⇕	⇕	Dividi (con segno) per 2, K volte
3	sar	G	⇕	⇕	⇕	⇕	⇕	Dividi (con segno) per 2, RCX volte
4	shr	K, G	⇕	⇕	⇕	⇕	⇕	Dividi (senza segno) per 2, K volte
5	shr	G	⇕	⇕	⇕	⇕	⇕	Dividi (senza segno) per 2, RCX volte
6	rcl	K, G	⇕	-	-	-	⇕	Ruota a sinistra, K volte
7	rcl	G	⇕	-	-	-	⇕	Ruota a sinistra, RCX volte
8	rcr	K, G	⇕	-	-	-	⇕	Ruota a destra, K volte
9	rcr	G	⇕	-	-	-	⇕	Ruota a destra, RCX volte
10	rol	K, G	⇕	-	-	-	⇕	Ruota a sinistra, K volte
11	rol	G	⇕	-	-	-	⇕	Ruota a sinistra, RCX volte
12	ror	K, G	⇕	-	-	-	⇕	Ruota a destra, K volte
13	ror	G	⇕	-	-	-	⇕	Ruota a destra, RCX volte

Operazioni di rotazione e shift



Classe 4: Manipolazione dei bit di FLAGS

Tipo	Mnemonico	Operandi	O	S	Z	P	C	Descrizione
0	clc	—	—	—	—	—	0	Resetta CF
1	clp [†]	—	—	—	—	0	—	Resetta PF
2	clz [†]	—	—	—	0	—	—	Resetta ZF
3	cls [†]	—	—	0	—	—	—	Resetta SF
4	cli	—	—	—	—	—	—	Resetta IF
5	cld	—	—	—	—	—	—	Resetta DF
6	clo [†]	—	0	—	—	—	—	Resetta OF
7	stc	—	—	—	—	—	1	Imposta CF
8	stp [†]	—	—	—	—	1	—	Imposta PF
9	stz [†]	—	—	—	1	—	—	Imposta ZF
10	sts [†]	—	—	1	—	—	—	Imposta SF
11	sti	—	—	—	—	—	—	Imposta IF
12	std	—	—	—	—	—	—	Imposta DF
13	sto [†]	—	1	—	—	—	—	Imposta OF

†: non esiste un'istruzione corrispondente nell'assembly x86

Classe 5: Controllo del flusso di programma

Tipo	Mnemonico	Operandi	O	S	Z	P	C	Descrizione
0	jmp	M	-	-	-	-	-	Esegue un salto relativo
1	jmp	*G	-	-	-	-	-	Esegui un salto assoluto
2	call	M	-	-	-	-	-	Esegue una chiamata a subroutine relativa
3	call	*G	-	-	-	-	-	Esegue una chiamata a subroutine assoluta
4	ret	-	-	-	-	-	-	Ritorna da una subroutine
5	iret	-	⇕	⇕	⇕	⇕	⇕	Ritorna dal gestore di una interruzione

Classe 6: Controllo condizionale del flusso

Tipo	Mnemonico	Operandi	O	S	Z	P	C	Descrizione
0	jc	M	-	-	-	-	-	Salta a M se CF è impostato
1	jp	M	-	-	-	-	-	Salta a M se PF è impostato
2	jz	M	-	-	-	-	-	Salta a M se ZF è impostato
3	js	M	-	-	-	-	-	Salta a M se SF è impostato
4	jo	M	-	-	-	-	-	Salta a M se OF è impostato
5	jnc	M	-	-	-	-	-	Salta a M se CF non è impostato
6	jnp	M	-	-	-	-	-	Salta a M se PF non è impostato
7	jnz	M	-	-	-	-	-	Salta a M se ZF non è impostato
8	jns	M	-	-	-	-	-	Salta a M se SF non è impostato
9	jno	M	-	-	-	-	-	Salta a M se OF non è impostato