Lez. 1 – Introduzione ad Assembly

Laboratorio di Architettura degli Elaboratori

Michele Lora

14-15 marzo 2024

Outline

Outline

Vantaggi & svantaggi

Rispetto ad un linguaggio a più alto livello come Java o C, Assembly presenta i seguenti vantaggi:

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

Viceversa, i principali svantaggi sono:

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

A cosa serve conoscere Assembly?

Didattica permette di toccare con mano il funzionamento della CPU

A cosa serve conoscere Assembly?

Didattica permette di toccare con mano il funzionamento della CPU

Efficienza i programmi Assembly, una volta compilati, sono tipicamente più veloci e più piccoli dei programmi scritti in linguaggi ad alto livello

A cosa serve conoscere Assembly?

Didattica permette di toccare con mano il funzionamento della CPU

Efficienza i programmi Assembly, una volta compilati, sono tipicamente più veloci e più piccoli dei programmi scritti in linguaggi ad alto livello

Hardware è indispensabile per la scrittura di driver per hardware specifici

Outline

Registri

Tutti i processori della famiglia Intel x86 possiedono 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 pari a 16 bit. A partire dal 80386, la loro dimensione è stata portata a 32 bit e al loro nome è stata aggiunta la lettera E (per indicare extended) in prima posizione. Per ragioni di compatibilità, gli assemblatori accettano l'uso dei nomi originali considerando i corrispondenti registri a 16 bits.

Registri generici

EAX, EBX, ECX, ed EDX sono registri generici (*general purpose registers*), 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

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

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). E' possibile modificarlo manualmente ma occorre cautela!
- **EBP** (*base pointer*) punta alla base della porzione di stack gestita in quel punto del codice. E' possibile modificarlo manualmente ma occorre cautela!
- **EIP** (*instruction pointer*) punta alla prossima istruzione da eseguire. Non può essere modificato!

Registri indice

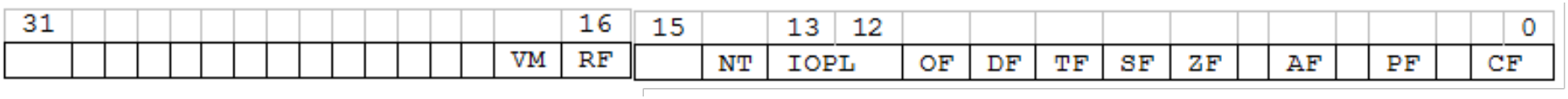
ESI e EDI sono i registri indice (*index registers*) e vengono utilizzati per operazioni con stringhe e vettori:

- **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

Composizione dei registri

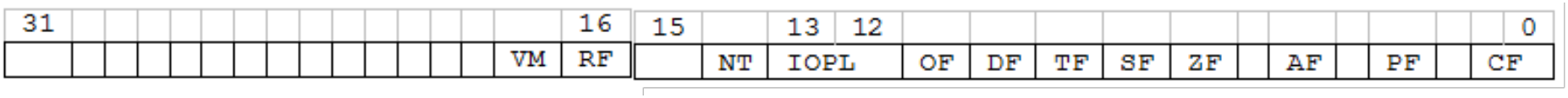
31 16	15 8	7 0	
	AH	AL	EAX
	BH	BL	EBX
	CH	CL	ECX
	DH	DL	EDX
	SP		ESP
	BP		EBP
	SI		ESI
	DI		EDI

Composizione del registro EFLAGS



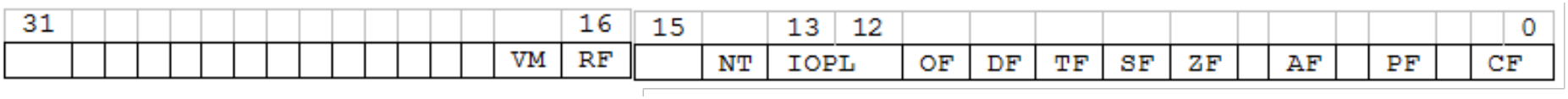
- **CF** (*carry flag*): impostato a 1 se un'operazione aritmetica chiede un prestito dalla cifra più significativa o esegue un riporto oltre la cifra più significativa
- **PF** (*parità flag*): impostato a 1 se il numero di 1 presenti nel risultato di un'operazione è dispari, a 0 se è pari.
- **AF** (*auxiliary flag*): utilizzato nell'aritmetica BCD (Binary Coded Decimal) per verificare se si è verificato un riporto o un prestito.

Composizione del registro EFLAGS



- **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.

Composizione del registro EFLAGS



- **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)

Modalità di indirizzamento

Si riferisce al modo in cui l'operando di un'istruzione viene specificato.
Esistono 7 modalità:

1. **Indirizzamento a registro**: l'operando è contenuto in un registro ed il nome del registro è specificato nell'istruzione (es. `%Ri`)
2. **Indirizzamento diretto** (o assoluto): l'operando è contenuto in una locazione di memoria, e l'indirizzo della locazione viene specificato nell'istruzione (es. `(IND)`)
3. **Indirizzamento immediato** (o di costante): l'operando è un valore costante ed è definito esplicitamente nell'istruzione (es. `$VAL`)

Modalità di indirizzamento

4. **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 (es. `(%Ri)` o `($VAL)`)
5. **Indirizzamento indicizzato** (base e spiazzamento): l'indirizzo effettivo dell'operando è calcolato sommando un valore costante al contenuto di un registro (es. `SPI(%Ri)`)
6. **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.
7. **Indirizzamento con autodecremento**: il contenuto di un registro specificato nell'istruzione viene decrementato. Il nuovo contenuto viene usato come indirizzo effettivo dell'operando.

Outline

Sintassi

La sintassi classica di un'istruzione è:

```
istruzione operando1, operando2, ...
```

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 `src`

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 `src`

`lea src, dst` **Load Effective Address offset**: trasferisce l'indirizzo di memoria dell'operando `src` nell'operando `dst`

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

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

Istruzioni aritmetiche e logiche

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

Istruzioni aritmetiche e logiche

`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

Istruzioni aritmetiche e logiche

`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`

Istruzioni aritmetiche e logiche

`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 a `dst` il valore di `src` e memorizza il risultato in `dst`

Istruzioni aritmetiche e logiche

`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).

Istruzioni aritmetiche e logiche

`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

Istruzioni aritmetiche e logiche

`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.

Istruzioni aritmetiche e logiche

`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`)

Istruzioni aritmetiche e logiche

`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 bit a bit dei due operandi e lo memorizza nell'operando `dst`

Istruzioni aritmetiche e logiche

`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 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`

Istruzioni aritmetiche e logiche

`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 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`

Outline

AT&T vs. Intel

Le principali differenze sono:

- in AT&T i nomi dei registri hanno il carattere % come prefisso (es. `%eax` invece di `eax`)
- 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 (parola) e `l` per double word (parola doppia)

AT&T vs. Intel

- gli operandi immediati sono indicati con il prefisso `$` (es. `addl $5, %eax`)
- la presenza di prefisso in un operando indica che si tratta di un indirizzo di memoria (es. `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 (es. `movl $5, 17 (%ebp)`)