Introduzione ad Assembly

Perché Assembly?

Da un punto di vista didattico, Assembly rappresenta il linguaggio di programmazione ideale durante lo studio dell'architettura dei calcolatori perché permette di toccare con mano il funzionamento della CPU.
Prescindendo dall'aspetto didattico, l'utilizzo del linguaggio Assembly, rispetto all'uso dei tradizionali linguaggi ad alto livello quali Java o C, è talvolta giustificato dalla maggiore efficienza del codice prodotto. I programmi Assembly, una volta compilati, sono tipicamente più veloci e più piccoli dei programmi scritti in linguaggi ad alto livello anche se il numero di righe di codice scritte dal programmatore è maggiore. Inoltre, la programmazione Assembly è talvolta indispensabile per la scrittura di driver per hardware specifici. Non va infine dimenticato che conoscere l'Assembly consente di scrivere codice ad alto livello di qualità migliore.
Confrontando le potenzialità del linguaggio Assembly con quelle di un linguaggio di programmazione ad alto livello, si possono individuare i seguenti vantaggi/svantaggi:

Vantaggi	Svantaggi
• è possibile accedere direttamente ai registri della CPU • è possibile scrivere codice ottimizzato per una specifica architettura di CPU • è possibile ottimizzare accuratamente le sezioni "critiche" dei programmi	• possono essere richieste molte righe di codice Assembly per riprodurre il comportamento di poche righe di Java o C • è facile introdurre dei bug perché la programmazione è più complessa • i bug sono difficili da trovare • non è garantita la compatibilità del codice per versioni successive dell'hardware

Richiami sui processori Intel x86

I 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, fino alla nascita del processore 80386, tutti i registri avevano dimensione pari a 16 bit. A partire dal processore 80386, con l'introduzione dell'architettura IA-32, la dimensione dei registri AX, BX, CX, DX, SP, BP, SI, DI, IP e FLAGS è stata portata a 32 bit e al loro nome è stato aggiunto il prefisso *E* per indicare *extended*. Con l'introduzione dei processori a 64 bit, i registri sono stati ulteriormente rinominati sostituendo il prefisso *E* con il prefisso *R*.

General purpose registers

Sono registri generici e 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.

Segment registers

Sono i registri di segmento 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.

Pointer registers

Sono i registri puntatore:

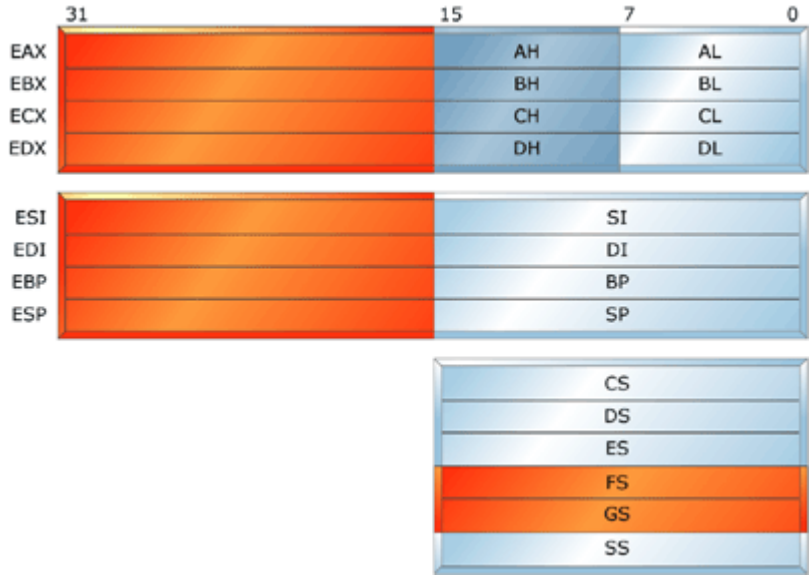
- **ESP** (*stack pointer*) punta alla cima dello stack. Viene modificato dalle operazioni di PUSH (inserimento di un dato nello stack) e POP (estrazioni di un dato dallo stack). Si ricordi che lo stack è una struttura di tipo LIFO (Last In First Out – l'ultimo che entra è il primo che esce). Può essere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

Index registers

Sono i registri indice e vengono utilizzati per operazioni con stringhe e vettori:

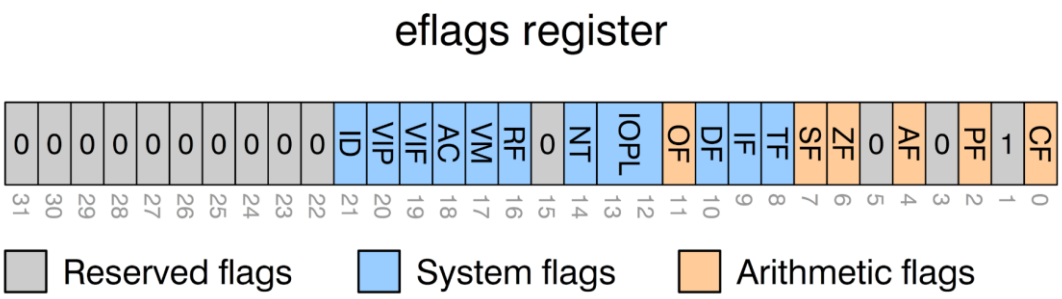
- **ESI** (*source index*) punta alla stringa/vettore sorgente.
- **EDI** (*destination index*) punta alla stringa/vettore destinazione.
- **EFLAGS** è utilizzato per memorizzare lo stato corrente del processore. Ciascuna *flag* (bit) del registro fornisce una particolare informazione. Ad esempio, il 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.

Come precedentemente accennato a partire dal microprocessore 80386, parte dei registri presenti nell'8086 sono stati estesi da 16 a 32 bit. La composizione dei registri general purpose e dei registri indice e puntatore è la seguente:



I 16 bit meno significativi dei registri general purpose (EAX, EBX, ECX, EDX) corrispondono ai registri AX, BX, CX, DX dei processori antecedenti all'80386.

La composizione del registro EFLAGS è la seguente:



Il significato dei flag più utilizzati è il seguente:

- **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** *(parity 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.
- **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).

Modalità di indirizzamento

Il termine *modalità di indirizzamento* si riferisce al modo in cui l'operando di un'istruzione viene specificato. Esistono 7 modalità di indirizzamento principali:

1. **Indirizzamento a registro:** l'operando è contenuto in un registro. Il nome del registro è specificato nell'istruzione. Es. %Ri.
2. **Indirizzamento diretto (o assoluto):** l'operando è contenuto in una locazione di memoria. 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. $VAL.
4. **Indirizzamento indiretto:** l'indirizzo di un operando è contenuto in un registro o in una locazione di memoria. L'indirizzo della locazione o il registro viene specificato nell'istruzione. Es. (%Ri) o ($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 all'operando, 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.

System calls

Le chiamate di sistema (in inglese *system calls*) sono il meccanismo, usato da un processo a livello utente o livello applicativo, per richiedere un servizio a livello kernel dal sistema operativo del computer in uso. Sono disponibili come funzioni in quei linguaggi di programmazione che supportano la programmazione di sistema (es. C). Per una lista completa delle chiamate di sistema del kernel Linux si può fare riferimento al manuale o al seguente link: http://asm.sourceforge.net/syscall.html.
In assembly, queste funzioni vengono chiamate ogni volta che si richiama l'interrupt 80 con il comando `int $0x80`.

Istruzioni

La sintassi classica di un'istruzione è la seguente:

```
istruzione operando1, operando2
```

Il numero di operandi dipende dal tipo di istruzione.

Sintassi AT&T vs. Intel

Esistono due principali tipi di sintassi per il linguaggio Assembly: la sintassi Intel e la sintassi AT&T. Il compilatore *gas* utilizza quest'ultima. Confrontando la sintassi AT&T con quella Intel, si possono evidenziare le seguenti differenze:

- In AT&T i nomi dei registri hanno il carattere `%` come prefisso, cosicché i registri sono `%eax`, `%ebx` e così via invece di solo `eax`, `ebx`, etc. Ciò fa sì che sia possibile includere nomi simbolici di variabili direttamente nel sorgente Assembly, senza alcun rischio di confusione (nella sintassi Intel esiste la consuetudine di premettere ai nomi di variabili l'underscore).
- In AT&T l'ordine degli operandi è `<sorgente>, <destinazione>` ed è opposto rispetto a quello della sintassi Intel. Quindi, l'istruzione che carica il contenuto del registro EDX nel registro EAX, in AT&T è `mov %edx, %eax`.
- In AT&T la lunghezza dell'operando è specificata tramite un suffisso al nome dell'istruzione. Il suffisso è `b` per *byte* (8 bit), `w` per *word* (parola) e `l` per *double word* (parola doppia). Ad esempio, la sintassi corretta per l'istruzione menzionata poco fa è `movl %edx, %eax`. Tuttavia, poiché *gas* non richiede una sintassi AT&T rigorosa, il suffisso è opzionale quando la lunghezza dell'operando può essere ricavata dal tipo registri usati nell'operazione. In caso contrario viene posta a 32 bit e viene emesso un Warning.
- Gli operandi immediati sono indicati con il prefisso `$`. Ad esempio `addl $5, %eax` (somma su 32 bit il valore 5 al registro EAX e metti il risultato in EAX).
- La presenza di prefisso in un operando indica che si tratta di un indirizzo di memoria. Pertanto, l'istruzione `movl $pippo, %eax` mette l'*indirizzo* della variabile pippo nel registro EAX, mentre `movl pippo, %eax` mette il contenuto della variabile pippo nel registro 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 l'istruzione `movl $5, 17(%ebp)` copia il valore 5 (a 32 bit) nella posizione 17 contando a partire dalla cella puntata dal valore contenuto in EBP.

Alcune istruzioni Assembly

Di seguito sono riportate e descritte alcune delle principali istruzioni Assembly. Il set completo di istruzioni dei processori della famiglia Intel x86 è facilmente reperibile in rete o nei manuali Assembly reperibili nella biblioteca del Dipartimento di Informatica.

Istruzioni generali

Sintassi	Descrizione
`mov src, dst`	**Move** – Effettua lo spostamento di valore tra due operandi. 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`.
`int op`	**Interrupt** – Attiva un interrupt software. Si usa per chiamare delle routine di sistema (System calls) e l'operando deve essere un valore numerico che identifica l'interrupt.

Istruzioni Aritmetiche

Sintassi	Descrizione
`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, mentre `op2` deve essere necessariamente 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, mentre `op2` deve essere necessariamente 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 contenuto in `dst`.
`sub src, dst`	**Subtract** – Sottrae a `dst` il valore di `src` e memorizza il contenuto in `dst`.
`mul op`	**Unsigned multiplication** – Esegue la moltiplicazione senza segno. `op` deve essere un registro o una variabile. Se `op` è un byte il registro AL viene moltiplicato per l'operando e il risultato viene memorizzato in AX. Se `op` 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 `op` 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).
`div op`	**Unsigned division** – Esegue la divisione senza segno. `op` deve essere un registro o una variabile. Se `op` è un byte il registro AX viene diviso per l'operando, il quoziente viene memorizzato in AL, e il resto in AH. Se `op` 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 `op` 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 devono essere memorizzati nel registro EDX), il quoziente viene memorizzato nel registro EAX e il resto in EDX.
`imul op`	Moltiplicazione con numeri con segno
`idiv op`	Divisione con numeri con segno

Istruzioni logiche

Sintassi	Descrizione
`xor src, dst`	**Logical exclusive OR** – Calcola l'OR esclusivo bit a bit dei due operandi e memorizza il risultato in `dst`. Si utilizza spesso per azzerare un registro (utilizzando il registro stesso sia come `src` che `dst`).
`or src, dst`	**Logical OR** – Calcola l'OR bit a bit dei due operandi e memorizza il risultato in `dst`.
`and src, dst`	**Logical AND** – Calcola l'AND bit a bit dei due operandi e memorizza il risultato in `dst`.
`not op`	**Logical NOT** – Inverte ogni singolo bit dell'operando `op`.

Struttura di un programma Assembly

I programmi Assembly sono solitamente composti da almeno tre sezioni: *text*, *data* e *bss*. Ognuna di queste sezioni può essere eventualmente vuota. Altre sezioni possono essere create mediante la direttiva `.section`.
La sezione `.data` viene utilizzata per dichiarare variabili globali inizializzate e costanti.

La sezione `.text` contiene il codice Assembly vero e proprio. Questa sezione deve iniziare con la dichiarazione `.global _start` che fornisce al sistema operativo la locazione di memoria in cui si trova la prima istruzione del programma (è simile alla funzione main di Java o di C).
La sezione `.bss` consente di riservare spazio in memoria quando il programma verrà caricato in memoria per essere eseguito; può servire per contenere variabili non inizializzate.

Esempio: Hello, World!

```
# Nome file
# ----------
# hello.s
#
# Istruzioni per la compilazione
# ------------------------------
# as -o hello.o hello.s
# ld -o hello hello.o
#
# Funzionalita'
# -------------
# Stampa a video la scritta "Hello, World!" e va a capo
#
# Commenti
# --------
# Il carattere # indica l'inizio di un commento.

.section .data              #sezione variabili globali

hello:                      #etichetta
     .ascii "Hello, World!\n"   #stringa costante

hello_len:
     .long . - hello        #lunghezza della stringa in byte

.section .text              #sezione istruzioni
     .global _start         #punto di inizio del programma

_start:
     movl $4, %eax          #Carica il codice della system call
                            #WRITE in eax per scrivere la stringa
                            #"Ciao Mondo!" a video.

     movl $1, %ebx          #Mette a 1 il contenuto di EBX
                            #Quindi ora EBX=1. 1 è il
                            #primo parametro per la write e
                            #serve per indicare che vogliamo
                            #scrivere nello standard output
```

```
leal hello, %ecx        #Secondo parametro del write
                        #Carica in ECX l'indirizzo di
                        #memoria associato all'etichetta
                        #hello, ovvero il puntatore alla
                        #stringa "Ciao Mondo!\n" da stampare.
movl hello_len, %edx    #Terzo parametro della write
                        #carica in EDX la lunghezza della
                        #stringa "Ciao Mondo!\n".

int $0x80               #esegue la system call write
                        #tramite l'interrupt 0x80

movl $1, %eax           #Mette a 1 il registro EAX
                        #1 è il codice della system call exit

xorl %ebx, %ebx         #azzera EBX. Contiene il codice di
                        #ritorno della exit

int $0x80               #esegue la system call exit
```

NOTA: è necessario che tutte le variabili siano inizializzate durante la dichiarazione nella sezione `.data`.

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 (estensione *.s*) contenenti il programma *sorgente*, tramite un normale *editor di testo*.
2. Assemblaggio dei file sorgenti, e generazione dei file *oggetto* (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*.

L'assemblatore

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

```
as –o miofile.o miofile.s
```

Si consulti la documentazione (`man as`) per l'elenco delle opzioni disponibili.

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. Notare che l'operazione di linking deve essere effettuata anche se il programma è composto da un solo modulo oggetto.
Durante il corso verrà utilizzato il linker ***ld*** della GNU.
Per creare l'eseguibile a partire da un file oggetto è necessario eseguire il seguente comando:

```
ld –o miofile.x miofile1.o miofile2.o miofile3.o
```

Si consulti la documentazione (`man ld`) per l'elenco delle opzioni disponibili.