Programmare in **Assembly**

-

`> Sintassi di riferimento: AT&T sotto Linux`

Autore: BlackLight
< blacklight@autistici.org >
Rilasciato sotto licenza GPL 3.0

Indice generale

Introduzione all'Assembly....4
  Uso odierno....4
  Note di sintassi....4
  Prerequisiti....5
Registri di una CPU Intel....6
  Definizione 6
  Tipi di registro e loro funzione....6
  Stack e accumulatore 8
  vantaggi e svantaggi 9
  ulteriori informazioni 9
  struttura logica interna di un registro 9
Indirizzamento della memoria....11
Quick start Assembly....13
  Commenti....13
  Data e text segment....13
  Tipi di dato....14
  Operatori e operandi....14
    Suffissi....14
  Syscalls....15
    Gestione degli interrupt....22
  Confronto con la sintassi Intel....22
Variabili in Assembly....24
  Costanti....24
  Dimensione di una variabile....24
  Passaggi per valore e per riferimento....25
  Esempio di read....25
  Valori di ritorno....26
Salti e controllo del flusso....27
Operazioni logico-aritmetiche....29
Esempio pratico: inversione di una stringa....31
Stack....32
  Istruzione call: Chiamate a funzioni in Assembly....35
  Passaggio di parametri a funzioni....35
  Valori di ritorno....37
I/O su periferiche....38
Gestione di file in Assembly....39
Inline Assembly....42

Introduzione all'Assembly

L'Assembly è il linguaggio a basso livello per eccellenza. Per basso livello intendiamo un linguaggio più vicino alla logica binaria del calcolatore che al modo di vedere l'algoritmo tipicamente umano, quindi più orientato alla risoluzione di un algoritmo attraverso gli step base all'interno della macchina (spostamenti di dati CPU<->memoria<->I/O) che alla sequenza *logica* di operazioni che il programmatore vuole realizzare. È infatti il più datato dei linguaggi, sviluppato già negli anni '50 come alternativa alle disumane sequenze binarie che fino ad allora venivano inserite manualmente nei calcolatori. L'Assembly di fatto non è altro che una rappresentazione simbolica del linguaggio macchina, dove ad ogni istruzione binaria elementare viene fatta corrispondere un'istruzione relativamente più semplice da ricordare. L'Assembly rimane comunque un'associazione uno-a-uno con il linguaggio macchina: ogni *statement* Assembly all'atto della creazione dell'eseguibile viene tradotto nella corrispondente stringa binaria.

Uso odierno

Nonostante sia da molti considerato un linguaggio datato e fortemente complesso, ci sono applicazioni al giorno d'oggi dove l'uso dell'Assembly è indispensabile. L'applicazione principale è quella della programmazione di dispositivi elettronici (PIC, EPROM...), dove le prestazioni sono importanti e dove è necessario controllare aspetti di basso livello (interrupt, segnali, trasferimento di dati...) non controllabili tramite un qualsiasi linguaggio di alto livello, se non in C. Altra applicazione tipica è generalmente quella della scrittura di driver per periferiche o di componenti del sistema operativo (in entrambi i casi generalmente si evita di scrivere interi listati in Assembly, cercando di ricorrere all'inline Assembly, ovvero all'incapsulamento di routine Assembly all'interno generalmente di codice scritto in C); anche in questi casi la scelta ricade sull'Assembly sia per un discorso di prestazioni (un listato scritto in Assembly, se scritto bene, è generalmente più ottimizzato rispetto allo stesso frammento di codice che fa la stessa cosa scritto in un linguaggio di alto livello) sia di controllo a basso livello dei dispositivi (controllo di interrupt, segnali, trasferimenti a basso livello ecc.). Altro uso tipico è quello per la scrittura di virus e malware in generale, dato che un linguaggio a basso livello può molto più facilmente manipolare le singole istruzioni che un calcolatore va ad eseguire, e infine il suo uso è pane quotidiano in tutte le pratiche connesse al *reverse engineering*, allo splendido mondo del reversing degli eseguibili.

Note di sintassi

A differenza di un linguaggio di alto livello, dove il compilatore offre un livello di astrazione che consente al programmatore di scrivere codice senza preoccuparsi della

macchina che ha sotto, l'Assembly è strettamente legato alla configurazione hardware della macchina sottostante. Esisteranno quindi molte sintassi Assembly, a seconda dell'architettura dove si va a programmare. La più documentata al giorno d'oggi è la sintassi *x86*, usata con qualche variante praticamente su tutte le macchine Intel-based (ma anche AMD) dall'8086 in su e implementata da un gran numero di assemblatori (MASM, Turbo ASM, NASM, FASM...). Un'altra sintassi spesso insegnata in ambito didattico per la sua scarsità di istruzioni è quella del *Motorola Z80*, processore con set di istruzioni ridotto sviluppato negli anni '80 e utilizzato ancora oggi in molte applicazioni embedded. C'è poi la sintassi PowerPC, sviluppata per le macchine IBM su cui fino a pochi anni fa venivano installati i Mac e ora caduta in disuso in seguito alla scelta di Apple di migrare a macchine Intel-based. In questa sede esamineremo invece la sintassi *AT&T* su sistemi Linux, sintassi sviluppata insieme agli stessi sistemi operativi Unix e nella quale è scritta buona parte del kernel Linux. Non ci sono grosse differenze logiche fra la sintassi AT&T in uso su Linux e BSD e la sintassi Intel usata da NASM o MASM, le differenze sono perlopiù a livello sintattico e verranno esaminate quando possibile una per una. Per la creazione di eseguibili a partire da listati Assembly useremo *gcc*, il compilatore di default sulla maggior parte dei sistemi Unix che ci farà anche da assemblatore e linker.

Prerequisiti

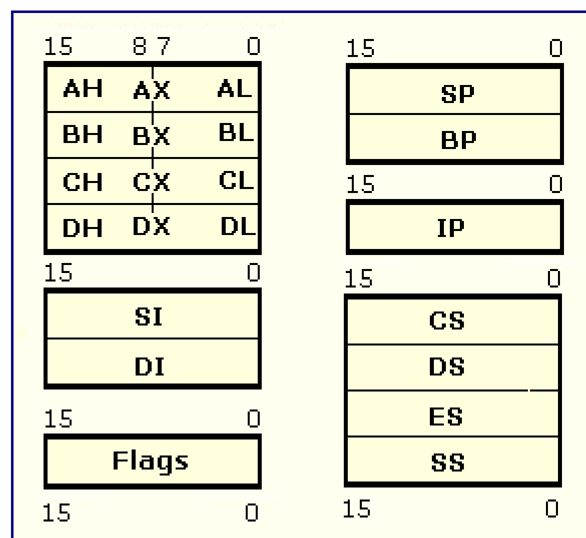
Innanzitutto è consigliabile avere già esperienza con qualche linguaggio di programmazione ad alto livello. Avvicinarsi alla programmazione direttamente con l'Assembly non è una strada consigliata. Inoltre, è necessario conoscere a fondo l'architettura interna dei computer, in particolare le funzioni e i principali tipi di registri della CPU (usati in modo massiccio in Assembly) e l'interfacciamento con la memoria e l'I/O.

Registri di una CPU Intel

In questa sezione ci occuperemo,di definire in modo completo la funzione dei registri nell'architettura di una CPU. In particolare faremo riferimento per gli esempi all'architettura Intel 8086.

Definizione

I registri sono riconducibili,come funzione,a piccole memorie di dimensione fissa,che l'ALU utilizza per memorizzare,gli operandi dei propri calcoli. I registri intel sono divisibili in varie categorie:



I registri dell'8086 sono tutti di ugal dimensione (16 bit),che è la stessa dimensione del parallelismo degli operandi,ovvero la dimensione dei dati che la cpu tratta. L'intel opera con un set specifico di registri non ortogonali,significa che non sono utilizzabili liberamente,ma che hanno uno specifico ruolo. Daremo ora uno sguardo al set di registri più da vicino,più avanti spiegheremo il modello stack e accumulatore.

Tipi di registro e loro funzione

-registri general purpose:I registri general purpoue AX,BX,CX,DX sono tutti divisi in due parti da 8 bit identificabili rispettivamente con AL-AH.BL-BH,CL-CH,DL-DH. Questi registri sono usati normalmente per i seguenti scopi:opreazioni aritmetiche,operazioni logiche,trasferimento di dati. Inoltre:

AX(accumulatore):può essere utilizzato in operazioni I/O,traslazioni e oprazioni BCD.

BX(base):può essere utilizzato come base per il calcolo di indirizzi in memoria,sommando a esso specifici offset.

CX(contatore):viene anche utilizzato come contatore per operazioni ripetute.

DX(dati):registro supplementare per dati;nello specifico può contenere operandi per divisioni,moltiplicazioni,e gli indirizzi delle porte per I/O.

-registri puntatori e indici: Questo set di registri si divide in due tipi:

i puntatori SP(stack pointer) e BP(base pointer) che puntano rispettivamente alla cima e a un punto interno dello stack;

SI (source index) e DI (destination index) usati come registri di indirizzamento sorgente e destinazione per movimenti di blocchi di memoria.

-registri di segmento:Sono registri che utilizzati in coppia con altri,vengono utilizzati per generare indirizzi a 20 bit,partendo da una dimensione di 16 bit.

CS(code segment):inizio della locazione di memoria che contiene il programma da eseguire.

DS(data segment):primo byte della zona di memoria che contiene i dati.

SS(stack segment):inizio della parte di memoria denominata come stack.

ES(extra segment):segmento dinamico,utilizzabile secondo le esigenze.

Dato che i primi tre contengono,gli indirizzi di base delle zone di memoria,sommando a essi, opportuni offset,possiamo accedere a tutte le celle di quella zona di memoria;infatti quando la cpu accede in memoria,non richiama direttamente un indirizzo,ma somma un offset alla base della zona di memoria interessata.

-registri speciali: IP(istruction pointer):contiene l'indirizzo della prossima istruzione da eseguire,cioè l'offset da sommare a CS dell'istruzione successiva,nel programma in esecuizione.

FLAG(registro(di stato): a 16 bit,composto da 16 celle da 1 bit,dove ogni bit ha un significato specifico.

L'intel ne utilizza solo 9. Questi bit,non controllabili dal programmatore,vengono modificati dalla cpu quando,all'interno del sistema,si avvera un particolare

evento.Queste celle possono assumere solo valore 1 o 0. Significato delle celle:

overflow(OF):l'operazione eseguita ha riportato un risultato troppo grande;

sign(SF):viene posto a 1 se il risultato dell'operazione eseguita è negativo;

zero(ZF):abilitato se il risultato di un operazione è zero.

auxiliary carry(AF):indica un riporto o un prestito.

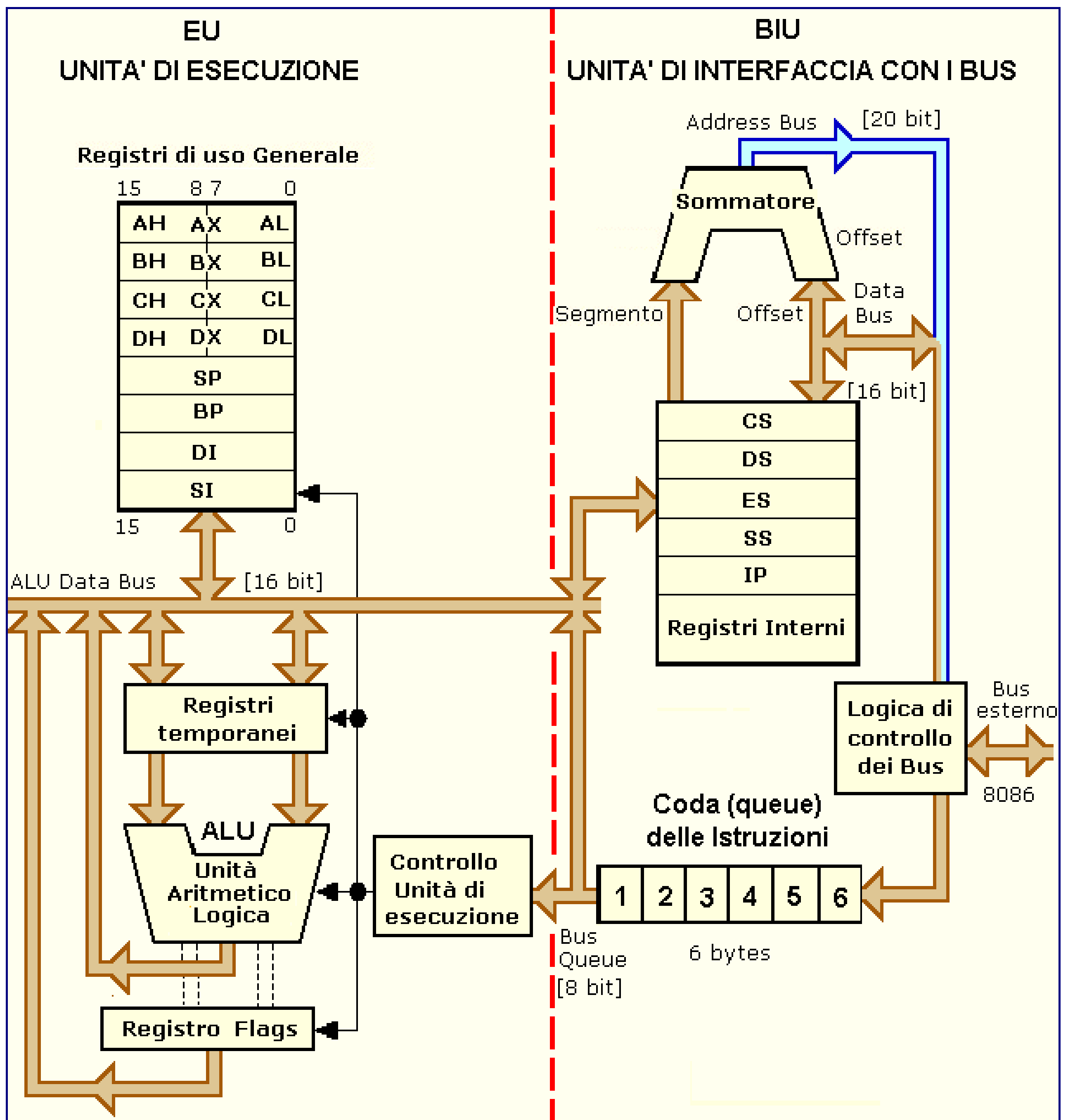
Parity flag(PF):posto a 1 quando c'è un numero pari di bit a 1 nel risultato dell'operazione.

Carry flag(CF):indica un riporto o un prestito nell'ultimo risulato.

Direction(DF):indica se incrementare o decrementare per le istruzioni con le stringhe.

interrupt enable(IF):indica se le interruzioni mascherate sono abilitate.

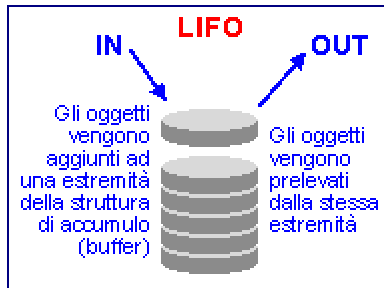
trap(TF):usato nei debugger per eseguire un passo alla volta.Genera un INT 3 dopo ogni istruzione.


EU
UNITA' DI ESECUZIONE
BIU
UNITA' DI INTERFACCIA CON I BUS
Registri di uso Generale
15
8 7
0
AH
AX
AL
BH
BX
BL
CH
CX
CL
DH
DX
DL
SP
BP
DI
SI
15
0
ALU Data Bus
[16 bit]
Registri temporanei
ALU
Unità Aritmetico Logica
Registro Flags
Controllo Unità di esecuzione
Address Bus
[20 bit]
Sommatore
Offset
Segmento
Offset
Data Bus
[16 bit]
CS
DS
ES
SS
IP
Registri Interni
Logica di controllo dei Bus
Bus esterno
8086
Coda (queue) delle Istruzioni
1
2
3
4
5
6
Bus Queue
[8 bit]
6 bytes

Stack e accumulatore

I modelli di memorie a stack e accumulatore,differiscono da quelli visi in precedenza,per le modalità di accesso,alle locazioni in essi contenute.

stackIl registro tack,utilizza un modello di accesso detto LIFO(last in first out);significa che qualsiasi cosa noi depositiamo all'interno di questo registro,essa verrà posta alla sommità di una sorta di pila;e nel caso di una lettura,potrà essere prelevata solo la cella che si trova più in alto(da qui LIFO). Infatti dal punto di vista delle istruzioni assembler di controllo,esistono solo due istruzioni,push e pop.La prima deposita sulla pila un dato,la seconda lo preleva.



accumulatore L'architettura ad accumulatore,è molto semplice;poichè prevede un solo registro AC, caricabile e scaricabile liberamente;questa però rimane,dal punto di vista delle prestazioni una soluzione molto limitativa.

Vantaggi e svantaggi

STACK:difficoltà di accesso,effetto collo di bottiglia.*Vanatggi*:indipendenza dal register set.

ACCUMULATORE:limitatezza,l'accumulatore è il collo di bottiglia.*Vantaggi*:gestione semplice e veloce.

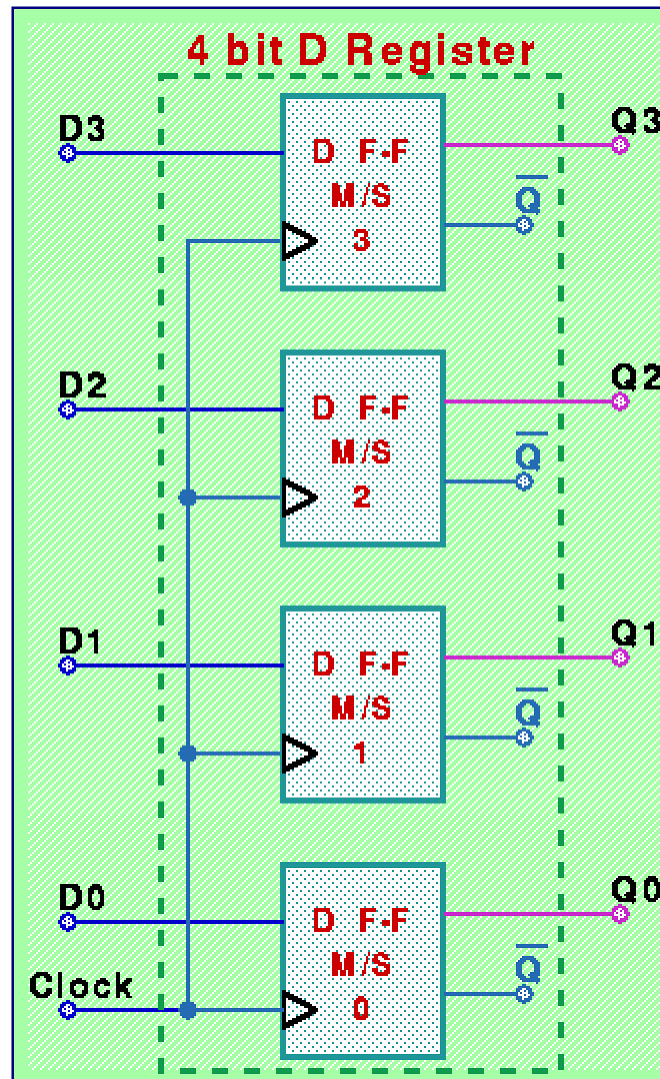
SET DI REGISTRI:codice puù lungo.*Vantaggi*:molto generale,opreandi espliciti. (in fase di sviluppo)

Ulteriori informazioni

I registri sono fondamentali nella realizzazione di qualsiasi istruzione da parte della cpu. E' importante conoscere bene il ruolo e le funzioni che possono avere. Per controllare i registri bisogna fare riferimento,al codice assembly,ed utilizzare apposite istruzioni. Di questo parleremo nella sezione dedicata a questo linguaggio. Può risultare utile in termini di completezza,illustrare la struttura interna e la rete logica che costituisce un registro.

Struttura logica interna di un registro

I registri sono formati da flip-flop sincronizzati sullo stesso clock. Significa che ogni flip-flop,in grado di memorizzare 1 bit,ha il clock che lo attiva,collegato agli altri dispositivi in parallelo. Quando il clock (generalmente associato anche a un altro segnale di abilitazione)si attiva,i flip-flop ad esso collegati si attivano anch'essi,memorizzando la parola in ingresso,dividendola bit per bit in ogni dispositivo. Quindi un registro di n flip-flop in parallelo è in grado di memorizzare parole di n bit.



Nella figura d'esempio D0-D3 sono gli n bit in ingresso,che vengono memorizzati;Q0-Q3 sono le uscite dei flip flop,che restituiscono la parola memorizzata,qualora richiesto.

CK è il clock di abilitazione che temporizza i flip-flop;si può anche trovare inserito

in una porta logica and con un altro segnale,di enable (EN),che abilita o meno il registro.

Indirizzamento della memoria

L'operazione con cui la cpu accede a dati e istruzioni è detta "indirizzamento in memoria". Essa infatti richiama non il dato direttamente (o l'istruzione),ma la cella di memoria che contiene questi bit. La memoria di fatto è divisa in celle,di uguale dimensione,allocate e organizzate a seconda del tipo di architettura e di ISA.

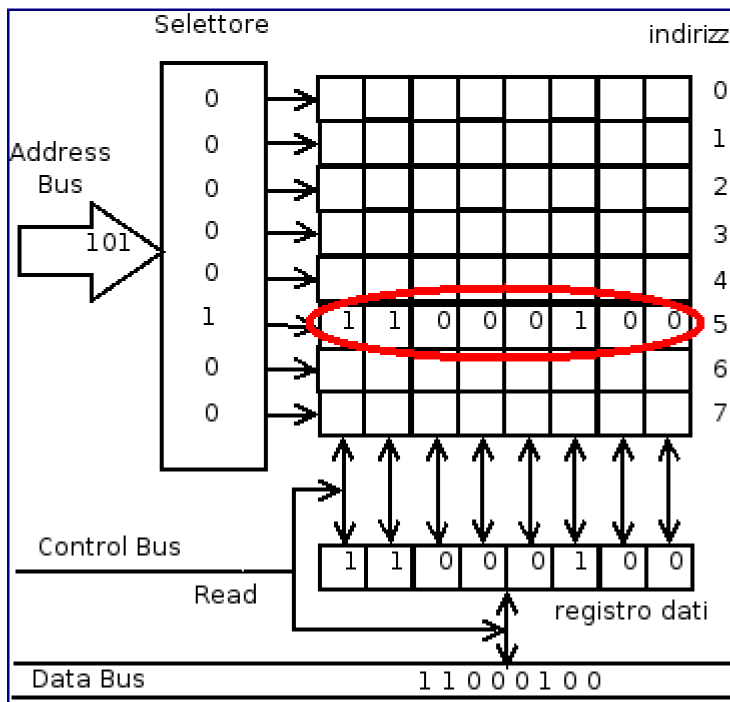
E' detto spazio di indirizzamento,la grandezza di memoria che la cpu ha a disposizione,e quindi,che può indirizzare. La cpu,di fatto,dispone di due bus,che transitano verso e dalla memoria,quello dei dati (dove fa transitare i dati veri e propri),e quello degli indirizzi (dove invece transitano gli indirizzi da cercare in memoria). Indicati con nd e na,essi permettono di ricavare la dimensione dei dati in transito,e la dimensione dello spazio di indirizzamento,infatti,per calcolare quest'ultimo,si può utilizzare la formula:

spazio di indirizzamento = 2na

Questo valore indica quanta memoria può essere indirizzata dal calcolatore. Se poi questo valore,viene diviso per la grandezza di parola,che è la dimensione massima di bit,memorizzabili in ogni cella(non sempre conincide con la dimensione del bus nd);si può conoscere il numero di indirizzi in cui è suddivisa la memoria(il numero di celle allocate).

$$\text{N max indirizzi} = \frac{spazio\ indirizzamento}{dim.parola}$$

Quindi tutte le volte che la cpu,deve accedere in memoria,prima invia l'indirizzo di interesse,e poi opera sulla cella selezionata.



Per dare un ultimo esempio: L'intel 8086 avente nd=16 e na=20 è in grado di indirizzare

$2^{na} = 1Mb$ di memoria.

Quick start Assembly

Cominciamo subito considerando un piccolo codice Assembly che stampa su standard output la stringa "Hello world":

```
// hello.S
// In AT&T Assembly possiamo tranquillamente usare i commenti stile
C/C++

.data

.text
        .global main

hello:
        .string         "Hello world!\n"
main:
        movl            $4,%eax
        movl            $1,%ebx
        movl            $hello,%ecx
        movl            $13,%edx
        int             $0x80

        movl            $1,%eax
        movl            $0,%ebx
        int             $0x80
```

Per la compilazione diamo il seguente comando:

```
gcc -o hello hello.S
```

Esaminiamolo nel dettaglio.

Commenti

Innanzitutto i commenti. Il listato, essendo compilato con gcc, accetta tranquillamente sia i commenti /* stile C puro */ sia i commenti // stile C++. Inoltre, accetta anche i commenti stile Bash e Perl, che iniziano con # e finiscono con il fine riga. L'Assembly puro (sintassi Intel, quale ad esempio quella supportata da NASM, MASM, TASM ecc.) supporta invece i commenti che cominciano con *;* e finiscono con il fine riga, ma tali commenti non sono invece supportati dalla sintassi AT&T.

Data e text segment

All'atto della creazione in memoria di un programma vengono allocati sulla memoria centrale due segmenti: uno per i dati (variabili globali e statiche) e uno per il codice (text segment). L'inizio di questi due segmenti viene indicato nel codice Assembly

rispettivamente dalle etichette speciali *.data* e *.text*. In questo caso il data segment è vuoto (non abbiamo variabili globali), quindi potremmo benissimo usare solo l'etichetta *.text*.

Con *.global main* dichiariamo un'etichetta globale che identifica l'inizio del programma vero e proprio. Le etichette nel codice in Assembly vanno specificate sempre allo stesso modo:

```
nome_etichetta:
  codice
  codice
  codice
```

Tipi di dato

Abbiamo quindi l'etichetta *hello*, che identifica la nostra stringa (da notare l'operatore *.string* per identificare una stringa). Altri tipi di dato comuni nell'Assembly AT&T sono

```
.string   // Stringa
.byte     // Variabile a 8 bit
.short    // Variabile a 16 bit
.long     // Variabile a 32 bit
.space <n_byte>    // Alloca n byte all'etichetta specificata
```

Operatori e operandi

Istanziati i segmenti e dichiarate le variabili, nell'etichetta *main* troviamo il codice vero e proprio. Notiamo subito l'uso di *movl*. *mov* è il principale operatore usato in Assembly, ed è usato per lo spostamento di dati fra i registri della CPU e fra i registri e la memoria centrale. A seconda dei dati che va a spostare può assumere diversi suffissi:

```
movb    // Spostamento di 1 byte
movw    // Spostamento di 2 byte
movl    // Spostamento di 4 byte
```

Questi suffissi come vedremo in seguito sono usati con qualsiasi operatore che manipoli dati fra registri e memoria per specificare il tipo di dato che si va a manipolare. Inoltre la MOV nella convenzione AT&T ha la seguente sintassi:

```
mov  SORGENTE,DESTINAZIONE
```

Quindi

```
// La seguente istruzione sposta il valore 4 nel registro EAX
movl  $4,%eax
```

Un'altra caratteristica della convenzione AT&T è l'indicare i registri con il prefisso *%* e gli scalari (costanti numeriche, variabili, indirizzi ecc.) con il prefisso *$*.

Suffissi

Si noti l'uso dei suffissi di MOV:

- Suffisso **b** -> Spostamento di 1 byte
- Suffisso **w** -> Spostamento di 2 byte (una word)
- Suffisso **l** -> Spostamento di 4 byte (un long)

L'uso di tali suffissi vale per ogni istruzione Assembly che implica la manipolazione di una certa quantità di memoria o un certo registro. Non è però richiesto, a meno che la sua omissione non causi ambiguità di interpretazione per l'assemblatore, quindi il codice può essere tranquillamente scritto come

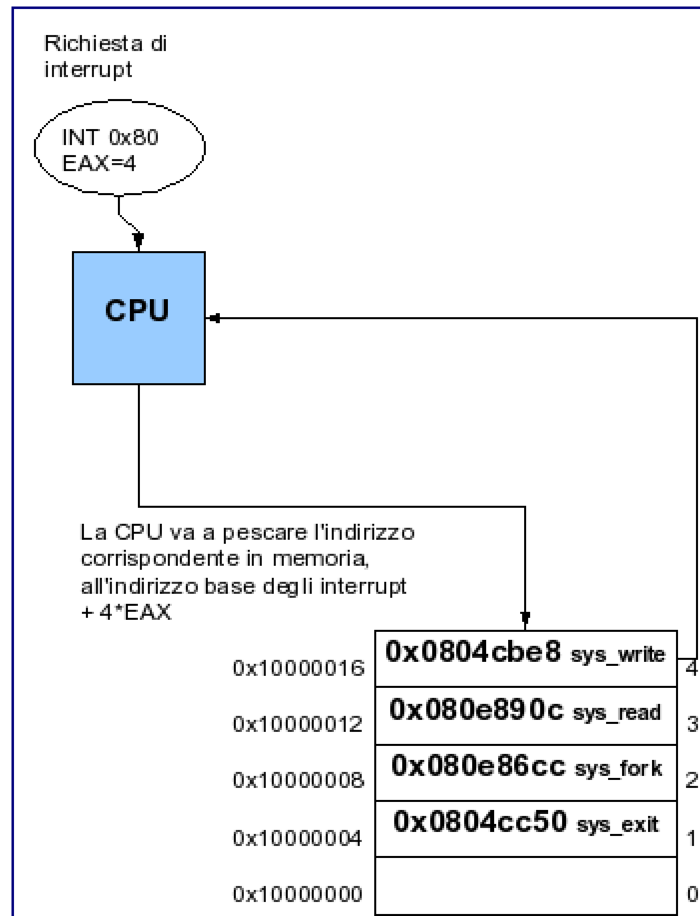
```
mov   $4,%eax
mov   $1,%ebx
; .....
```

Di seguito verrà usata indifferentemente la sintassi con o senza suffissi per gli operatori, a meno che l'uso dei suffissi non sia richiesto per evitare ambiguità nel codice.

Syscalls

Vediamo ora come viene fatta la stampa vera e propria. Per la stampa su stdout viene usata la *write syscall*, la chiamata di sistema corrispondente alla *write* che può essere richiamata anche in C. Una syscall si effettua richiamando un **interrupt** speciale, su sistemi Linux l'interrupt 0x80.

Prima di proseguire è doverosa una parentesi sugli interrupt. La CPU di un calcolatore generalmente fa una continua fase di **fetch-decode-execute**, ovvero prelievo dell'istruzione-decodifica dell'istruzione in termini di microistruzioni-esecuzione delle istruzioni. Tuttavia in alcuni casi il mondo esterno può aver bisogno di comunicare con la CPU. Il mondo esterno può essere una periferica di I/O (es. tastiera o scheda VGA) o il sistema operativo stesso. In quel caso manda alla CPU un *interrupt*, ovvero chiede alla CPU di interrompere quello che sta facendo ed eseguire una determinata operazione. Quando viene richiamato un interrupt la CPU salva tutti i valori dei suoi registri in memoria in modo da poter poi continuare il lavoro e va a vedere a che categoria appartiene (l'interrupt 0x80 sui sistemi Linux identifica gli interrupt del kernel, sui sistemi DOS c'è lo 0x21 per le funzioni di sistema DOS e lo 0x13 dedicato alle funzioni hardware, detti *interrupt del BIOS*), quindi va a vedere cosa c'è nel registro EAX per vedere che funzione deve richiamare. A questo punto accede alla memoria a un dato indirizzo (generalmente un indirizzo basso) in cui sono salvati gli indirizzi di memoria delle varie funzioni degli interrupt, usando il numero di funzione specificato in EAX come offset. Una volta prelevato l'indirizzo a cui si trova la routine da eseguire accede a quell'indirizzo ed esegue le istruzioni contenute al suo interno. Graficamente la situazione è quella che segue:



Ovvero

1. Il processo o la periferica richiede l'attenzione della CPU richiamando un interrupt
2. La CPU interrompe il suo lavoro, salva lo status dei registri e va a vedere il valore memorizzato in EAX e il tipo di interrupt richiesto
3. Il numero di funzione salvato in EAX viene usato come offset per accedere in memoria a partire di un certo indirizzo
4. All'indirizzo calcolato è salvato un altro indirizzo, al quale si trova il codice vero e proprio da eseguire
5. La CPU accede all'indirizzo specificato ed esegue il codice contenuto in esso
6. Terminata l'esecuzione del codice la CPU carica nuovamente lo stato dei registri dallo stack e riprende il suo lavoro prima della chiamata dell'interrupt

Questo è il meccanismo degli interrupt. Un interrupt in Assembly si lancia con la keyword *int*. Quando viene chiamato un interrupt come abbiamo visto la CPU va a vedere che tipo di interrupt è stato richiamato, e cosa c'è in EAX. L'interrupt numero 0x80 come abbiamo accennato in precedenza identifica le syscall, ovvero le chiamate interne del kernel. Per sapere che chiamata è stata richiesta si va a vedere in EAX. EAX contiene il valore 4: la funzione numero 4 dell'interrupt 0x80 è la *sys_write*, la cui sintassi in C dovrebbe essere già nota:

```
int write(int fd, const void *buf, size_t count);
```

Per conoscere l'elenco completo delle funzioni richiamabili dall'interrupt 0x80 su un sistema Linux basta dare un'occhiata generalmente al file */usr/include/asm/unistd.h*, al cui inizio sono definite tutte le syscall (ovvero che numero mettere in EAX quando si va a richiamare lo 0x80 per avere una data funzione).

```
#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5
#define __NR_close               6
#define __NR_waitpid             7
#define __NR_creat               8
#define __NR_link                9
#define __NR_unlink             10
#define __NR_execve             11
#define __NR_chdir              12
#define __NR_time               13
#define __NR_mknod              14
#define __NR_chmod              15
#define __NR_lchown             16
#define __NR_break              17
#define __NR_oldstat            18
#define __NR_lseek              19
#define __NR_getpid             20
#define __NR_mount              21
#define __NR_umount             22
#define __NR_setuid             23
#define __NR_getuid             24
#define __NR_stime              25
#define __NR_ptrace             26
#define __NR_alarm              27
#define __NR_oldfstat           28
#define __NR_pause              29
#define __NR_utime              30
#define __NR_stty               31
#define __NR_gtty               32
#define __NR_access             33
#define __NR_nice               34
#define __NR_ftime              35
#define __NR_sync               36
#define __NR_kill               37
#define __NR_rename             38
#define __NR_mkdir              39
#define __NR_rmdir              40
#define __NR_dup                41
#define __NR_pipe               42
#define __NR_times              43
#define __NR_prof               44
#define __NR_brk                45
#define __NR_setgid             46
#define __NR_getgid             47
#define __NR_signal             48
```

```
#define __NR_geteuid            49
#define __NR_getegid            50
#define __NR_acct               51
#define __NR_umount2            52
#define __NR_lock               53
#define __NR_ioctl              54
#define __NR_fcntl              55
#define __NR_mpx                56
#define __NR_setpgid            57
#define __NR_ulimit             58
#define __NR_oldolduname        59
#define __NR_umask              60
#define __NR_chroot             61
#define __NR_ustat              62
#define __NR_dup2               63
#define __NR_getppid            64
#define __NR_getpgrp            65
#define __NR_setsid             66
#define __NR_sigaction          67
#define __NR_sgetmask           68
#define __NR_ssetmask           69
#define __NR_setreuid           70
#define __NR_setregid           71
#define __NR_sigsuspend         72
#define __NR_sigpending         73
#define __NR_sethostname        74
#define __NR_setrlimit          75
#define __NR_getrlimit          76      /* Back compatible 2Gig
limited rlimit */
#define __NR_getrusage          77
#define __NR_gettimeofday       78
#define __NR_settimeofday       79
#define __NR_getgroups          80
#define __NR_setgroups          81
#define __NR_select             82
#define __NR_symlink            83
#define __NR_oldlstat           84
#define __NR_readlink           85
#define __NR_uselib             86
#define __NR_swapon             87
#define __NR_reboot             88
#define __NR_readdir            89
#define __NR_mmap               90
#define __NR_munmap             91
#define __NR_truncate           92
#define __NR_ftruncate          93
#define __NR_fchmod             94
#define __NR_fchown             95
#define __NR_getpriority        96
#define __NR_setpriority        97
#define __NR_profil             98
#define __NR_statfs             99
#define __NR_fstatfs           100
#define __NR_ioperm            101
#define __NR_socketcall        102
```

```
#define __NR_syslog           103
#define __NR_setitimer        104
#define __NR_getitimer        105
#define __NR_stat             106
#define __NR_lstat            107
#define __NR_fstat            108
#define __NR_olduname         109
#define __NR_iopl             110
#define __NR_vhangup          111
#define __NR_idle             112
#define __NR_vm86old          113
#define __NR_wait4            114
#define __NR_swapoff          115
#define __NR_sysinfo          116
#define __NR_ipc              117
#define __NR_fsync            118
#define __NR_sigreturn        119
#define __NR_clone            120
#define __NR_setdomainname    121
#define __NR_uname            122
#define __NR_modify_ldt       123
#define __NR_adjtimex         124
#define __NR_mprotect         125
#define __NR_sigprocmask      126
#define __NR_create_module    127
#define __NR_init_module      128
#define __NR_delete_module    129
#define __NR_get_kernel_syms  130
#define __NR_quotactl         131
#define __NR_getpgid          132
#define __NR_fchdir           133
#define __NR_bdflush          134
#define __NR_sysfs            135
#define __NR_personality      136
#define __NR_afs_syscall          137 /* Syscall for Andrew File
System */
#define __NR_setfsuid         138
#define __NR_setfsgid         139
#define __NR__llseek          140
#define __NR_getdents         141
#define __NR__newselect       142
#define __NR_flock            143
#define __NR_msync            144
#define __NR_readv            145
#define __NR_writev           146
#define __NR_getsid           147
#define __NR_fdatasync        148
#define __NR__sysctl          149
#define __NR_mlock            150
#define __NR_munlock          151
#define __NR_mlockall         152
#define __NR_munlockall       153
#define __NR_sched_setparam           154
#define __NR_sched_getparam           155
#define __NR_sched_setscheduler       156
```

```
#define __NR_sched_getscheduler         157
#define __NR_sched_yield                158
#define __NR_sched_get_priority_max     159
#define __NR_sched_get_priority_min     160
#define __NR_sched_rr_get_interval      161
#define __NR_nanosleep          162
#define __NR_mremap             163
#define __NR_setresuid          164
#define __NR_getresuid          165
#define __NR_vm86               166
#define __NR_query_module       167
#define __NR_poll               168
#define __NR_nfsservctl         169
#define __NR_setresgid          170
#define __NR_getresgid          171
#define __NR_prctl              172
#define __NR_rt_sigreturn       173
#define __NR_rt_sigaction       174
#define __NR_rt_sigprocmask     175
#define __NR_rt_sigpending      176
#define __NR_rt_sigtimedwait    177
#define __NR_rt_sigqueueinfo    178
#define __NR_rt_sigsuspend      179
#define __NR_pread              180
#define __NR_pwrite             181
#define __NR_chown              182
#define __NR_getcwd             183
#define __NR_capget             184
#define __NR_capset             185
#define __NR_sigaltstack        186
#define __NR_sendfile           187
#define __NR_getpmsg            188     /* some people actually want
streams */
#define __NR_putpmsg            189     /* some people actually want
streams */
#define __NR_vfork              190
#define __NR_ugetrlimit          191      /* SuS compliant getrlimit
*/
#define __NR_mmap2              192
#define __NR_truncate64         193
#define __NR_ftruncate64        194
#define __NR_stat64             195
#define __NR_lstat64            196
#define __NR_fstat64            197
#define __NR_lchown32           198
#define __NR_getuid32           199
#define __NR_getgid32           200
#define __NR_geteuid32          201
#define __NR_getegid32          202
#define __NR_setreuid32         203
#define __NR_setregid32         204
#define __NR_getgroups32        205
#define __NR_setgroups32        206
#define __NR_fchown32           207
#define __NR_setresuid32        208
```

```
#define __NR_getresuid32        209
#define __NR_setresgid32        210
#define __NR_getresgid32        211
#define __NR_chown32            212
#define __NR_setuid32           213
#define __NR_setgid32           214
#define __NR_setfsuid32         215
#define __NR_setfsgid32         216
#define __NR_pivot_root         217
#define __NR_mincore            218
#define __NR_madvise            219
#define __NR_madvise1            219     /* delete when C lib stub is
removed */
#define __NR_getdents64         220
#define __NR_fcntl64            221
#define __NR_security               223      /* syscall for security
modules */
#define __NR_gettid             224
#define __NR_readahead          225
#define __NR_setxattr           226
#define __NR_lsetxattr          227
#define __NR_fsetxattr          228
#define __NR_getxattr           229
#define __NR_lgetxattr          230
#define __NR_fgetxattr          231
#define __NR_listxattr          232
#define __NR_llistxattr         233
#define __NR_flistxattr         234
#define __NR_removexattr        235
#define __NR_lremovexattr       236
#define __NR_fremovexattr       237
#define __NR_tkill              238
#define __NR_sendfile64         239
#define __NR_futex              240
#define __NR_sched_setaffinity  241
#define __NR_sched_getaffinity  242
#define __NR_set_thread_area    243
#define __NR_get_thread_area    244
#define __NR_io_setup           245
#define __NR_io_destroy         246
#define __NR_io_getevents       247
#define __NR_io_submit          248
#define __NR_io_cancel          249
#define __NR_alloc_hugepages    250
#define __NR_free_hugepages     251
#define __NR_exit_group         252
```

Da notare che per richiamare la *write* bisogna effettivamente mettere 4 in EAX:

```
#define __NR_write                4
```

Ora bisogna tenere presente la sintassi della write:

```
int write(int fd, const void *buf, size_t count);
```

Quando si chiama una syscall a livello Assembly gli argomenti passati alla funzione vanno salvati sui registri in ordine, quindi primo argomento della funzione su EBX, secondo su ECX e terzo su EDX. Per stampare "Hello world\n" attraverso la write in C richiameremo qualcosa di questo tipo:

```
write (1,hello,sizeof(hello));
```

ovvero passeremo alla write come parametri

1. Descrittore di file (1, standard output)
2. Stringa
3. Dimensione della stringa

In Assembly facciamo esattamente la stessa cosa, ma scrivendo questi argomenti sui registri:

```
movl            $4,%eax
movl            $1,%ebx
movl            $hello,%ecx
movl            $13,%edx
int             $0x80
```

Su EBX scriviamo il nostro descrittore dello standard output, su ECX l'indirizzo della nostra stringa (indirizzo a cui si trova l'etichetta *hello* nel nostro programma) e lunghezza della stringa. A questo punto l'interrupt una volta richiamato eseguirà in maniera corretta una *write*.

(Piccola nota: nella sintassi AT&T $13 identifica 13 come numero decimale, $0x13 identificherà 13 come numero esadecimale, ovvero come $19 decimale, quindi attenzione: un numero senza alcun prefisso viene visto dall'assemblatore come decimale, se preceduto dal prefisso 0x come esadecimale).

Ora dobbiamo uscire dal programma. Per fare ciò richiamiamo un'altra syscall. La *exit* vedendo dalla lista è la funzione 1 dell'interrupt 0x80, quindi richiamiamo questa funzione scrivendo l'argomento (ovvero il codice di uscita) su EBX. In C scriveremmo

```
exit(0);
```

In Assembly:

```
movl            $1,%eax
movl            $0,%ebx
int             $0x80
```

Gestione degli interrupt

Le macchine Intel hanno una gestione degli interrupt molto versatile, e un'interruzione può essere mascherabile o non mascherabile (ovvero, giunta un'interruzione la CPU può potenzialmente anche ignorarla se richiesto nel codice). Ovviamente, le interruzioni non mascherabili sono quelle a priorità massima critiche per il funzionamento del sistema, ad esempio un critical error da un modulo del

kernel, un segnale KILL inviato dall'utente ecc.

È possibile gestire le interruzioni all'interno del codice Assembly manipolando il registro *FLAG*, in particolare il bit *interrupt flag* (*IF*). Se tale flag è settato a 1, la CPU può accettare interruzioni esterne al codice in esecuzione, altrimenti le ignorerà. Per manipolare questo bit, esistono rispettivamente le due istruzioni Assembly **STI** e **CLI**. Uso tipico:

```
cli    ; Ignoro le interruzioni
; Eseguo una sessione critica (sessione di I/O o altro tipo non interrompibile)
sti    ; Accetto nuovamente le interruzioni
```

Confronto con la sintassi Intel

Per completezza vediamo come avremmo scritto lo stesso codice usando la sintassi Intel pura e usando NASM come assemblatore invece di gcc:

```
section .text
        hello   db      "Hello world!",0xa
        global  _start
_start:
        mov             eax,4
        mov             ebx,1
        mov             ecx,hello
        mov             edx,13
        int             80h

        mov             eax,1
        mov             ebx,0
        int             80h
```

Per assemblare il codice sorgente:

```
nasm -f elf hello.asm
```

In questo modo creiamo un file oggetto. Per linkarlo e creare l'eseguibile:

```
ld -o hello hello.o
```

Vediamo ora le differenze a livello sintattico. Innanzitutto la dichiarazione del text segment va fatta precedere dalla keyword *section*. Inoltre le variabili non sono tipizzate come nella sintassi AT&T, ma basta specificare se sono dimensionate in byte (*db*), in word (*dw*) o in double word (*dd*). Il simbolo per l'inizio del programma, inoltre, non è il *main* come in *gcc* (convenzione legata al C), ma è generalmente _start. Le differenze che balzano maggiormente all'occhio in ogni caso sono nel codice vero e proprio. Innanzitutto la sintassi Intel non richiede i suffissi per le MOV ma fa i dimensionamenti automaticamente. Allo stesso modo non richiede nemmeno i prefissi per differenziare gli scalari dai registri, mentre invece i numeri esadecimali vengono identificati con un *h* finale invece che dal prefisso 0x. La differenza più importante però è nell'ordine degli operandi della MOV. Mentre la sintassi AT&T prevede una convenzione SORGENTE,DESTINAZIONE la sintassi Intel prevede la

convenzione DESTINAZIONE,SORGENTE. A parte queste differenze sintattiche, è indifferente usare l'una o l'altra sintassi (e quindi NASM o GAS/GCC).

Variabili in Assembly

In Assembly una variabile non è altro che un'allocazione di spazio identificata da un etichetta e un tipo, che può essere fatta in qualsiasi parte del programma. La sintassi della dichiarazione è questa:

```
nome:   .tipo   valore
```

Esempio:

```
hello:  .string  "Ciao\n"
```

Dichiara una stringa chiamata *hello* e contenente i valori ASCII che formano la parola "Ciao\n". Una dichiarazione può essere fatta in qualsiasi punto del codice, anche se per maggior pulizia del codice è consigliato dichiarare le variabili nel segmento dati dell'applicazione. Il tipo .string è stato già incontrato nell'esempio precedente, e identifica una normale stringa ASCII. Fra gli altri tipi:

```
.byte      # Singolo byte o carattere
.word      # short int, 2 byte
.long      # long int, 4 byte
.string    # stringhe
.space     # allocazione di spazio arbitrario
```

Il tipo .space è molto particolare. È più o meno corrispondente al tipo void di linguaggi come il C, identifica semplicemente uno spazio di dimensione arbitraria da allocare in memoria,e torna utile per definire tutti i tipi di dati derivati (strutture, enumerazioni ecc.). Esempio:

```
# Alloco una variabile grande 64 byte
var:   .space   64
```

Costanti

Le costanti in sintassi AT&T si dichiarano semplicemente nel seguente modo

```
nome = valore
```

Esempio:

```
var = 2
```

Dimensione di una variabile

La sintassi AT&T mette a disposizione un modo estremamente versatile e veloce per conoscere la dimensione di una variabile o la lunghezza di una stringa. Ecco la

sintassi:

```
str:   .string   "Ciao\n"
str_len = .-str
```

A questo punto la variabile *str_len* conterrà la lunghezza della variabile str. Si può ovviamente applicare a tutti i tipi di dato:

```
myspace:   .space   64
myspace_size = .-myspace    # myspace_size=64
```

Passaggi per valore e per riferimento

Facendo precedere al nome di una variabile il simbolo *$* si identifica il suo indirizzo, a meno che essa non sia una costante (in questo caso il simbolo $ è obbligatorio). L'abbiamo visto anche nell'esempio precedente per quanto riguarda la scrittura su stdout di una stringa. La syscall write prende come argomento l'indirizzo della zona di memoria da scrivere, e in quel caso facevamo precedere la nostra stringa dal simbolo $. Quando invece vogliamo identificare il valore contenuto in una variabile e non il suo indirizzo useremo semplicemente il nome della variabile senza prefissi, oppure il nome della variabile fra parentesi (le due notazioni sono equivalenti). Ovviamente questo ragionamento non è applicabile alle stringhe, che sono sempre viste come puntatori a zone di memoria terminanti con \0. Esempio:

```
.data
var:            .long   4

.text
        .global main
main:
        movl            $var,%eax
        movl            var,%ebx

......
```

Andando a debuggare vedremmo

```
10              movl            $var,%eax
Current language:  auto; currently asm
(gdb) p/x $eax
$1 = 0x80495d8    # Indirizzo di var
...
11              movl            var,%ebx
(gdb) p $ebx
$2 = 4      # Contenuto di var
```

Esempio di read

Con le nozioni che abbiamo ora, vediamo come poter leggere una stringa da stdin e stamparla nuovamente su stdout:

```
.data
# Alloco lo spazio per la stringa str
str:            .space  128

# Lunghezza della stringa
str_len =       .-str

.text
        .global main
main:
        # Chiamata a sys_read - funzione 3 dell interrupt 0x80
        movl            $3,%eax

        # Primo argomento della funzione 0 -> stdin
        movl            $0,%ebx

        # Passo l indirizzo della stringa e la relativa lunghezza
        movl            $str,%ecx
        movl            $str_len,%edx
        int             $0x80

        # Stampo la stringa letta
        movl            $4,%eax
        movl            $1,%ebx
        movl            $str,%ecx
        movl            $str_len,%edx
        int             $0x80

        movl            $1,%eax
        movl            $0,%ebx
        int             $0x80

        leave
        ret
```

Valori di ritorno

I valori di ritorno di una syscall vanno generalmente piazzati in EAX. Esempio, la syscall read ritorna il numero di byte letti dal descrittore, e tale valore è leggibile in EAX:

```
.data
# Alloco lo spazio per la stringa str
str:            .space  128

# Lunghezza della stringa
str_len =       .-str

# Numero di caratteri letti
N               .long   0

.text
        .global main
main:
```

```
        movl            $3,%eax
        movl            $0,%ebx
        movl            $str,%ecx
        movl            $str_len,%edx
        int             $0x80
         movl              %eax,N     # N ora conterrà il valore di
ritorno della chiamata, ovvero il numero di byte letti
```

Salti e controllo del flusso

L'Assembly non fornisce delle strutture per il controllo del flusso del codice versatili come i linguaggi ad alto livello (for, foreach, while, do while, switch...). Il controllo del flusso in un programma Assembly si fa nel modo più elementare possibile, nonché il primo inventato dagli informatici e oggi tanto deprecato nei linguaggi ad alto livello: i salti condizionati (equivalente al goto dei linguaggi ad alto livello se vogliamo). A basso livello fondamentalmente controllo la veridicità di una certa condizione esaminando il registro FLAG della CPU, e in caso affermativo salto ad una data etichetta nel codice. In questo modo posso sia creare degli if sia dei loop. Concettualmente, se voglio controllare che una variabile sia positiva ragionerò nel seguente modo:

```
 confronta  var,0
 salta_se_maggiore  etichetta_vero
etichetta_falso:
 // Qui ci va il codice da eseguire se var<=0
etichetta_vero:
 // Qui ci va il codice da eseguire se var>0
```

Oppure ecco come è possibile fare l'equivalente di un ciclo for o while che, ad esempio, svolge una certa azione 10 volte:

```
 poni  var=0
loop:
 // Azioni da compiere
 incrementa  var
 confronta  var,10
 salta_se_minore  loop  // Torno a loop finché var<10

 // Qui metto il codice da eseguire una volta uscito dal loop
```

In Assembly tutto ciò è possibile attraverso due semplici tipi di istruzioni:

- **cmp** (*compare*) - Confronta due tipi di dati, e setta nel registro FLAG i flag giusti ricavati dal confronto (ad esempio Zero Flag se i due dati sono uguali, GF o LF se il secondo è rispettivamente maggiore o minore dell'altro ecc.)
- **jmp** (*jump*) - Serie di istruzioni per eseguire salti incondizionati o condizionati (in questo caso vengono esaminati gli opportuni valori nel registro FLAG).

Ecco i principali tipi di jump:

- **jmp** - Salto incondizionato. L'esecuzione del codice passa all'etichetta specificata senza controllare ulteriori condizioni. Esempio:

```
// Istruzioni
jmp   end
```

```
// Il codice che c'è qui non verrà mai eseguito

end:
 // L'esecuzione del codice arriva direttamente qui
```

- **je/jz - jne/jnz** - Salta se è uguale/se è zero - Salta se non è uguale/se non è zero. Le prime due istruzioni fanno esattamente la stessa cosa, ovvero saltano ad un'etichetta se lo Zero Flag è attivo. Lo Zero Flag può essere attivo se il confronto precedente ha dato zero, e ciò è possibile nel caso in cui i valori confrontati precedentemente sono risultati uguali. jne/jnz ovviamente sono le istruzioni duali, ovvero saltano ad una certa etichetta se lo Zero Flag non è settato.

```
// Istruzioni
movl  $1,%eax   // Metto 1 in EAX
cmpl  $1,%eax   // Confronto 1 e il valore in EAX
je    label     // Salto se sono uguali a label

...

label:
 // Istruzioni
```

- **jg - jl** - Rispettivamente saltano ad un'etichetta se dal confronto precedente il secondo valore è risultato maggiore o minore del primo.
- **jge - jle** - Rispettivamente saltano ad un'etichetta se dal confronto precedente il secondo valore è risultato maggiore o uguale o minore o uguale al primo.

Questi sono i salti principali che ci serviranno nei nostri esempi.

Operazioni logico-aritmetiche

Vedremo qui gli opcode messi a disposizione dall'Assembly per le operazioni logico-aritmetiche.

- ***add* <term1>,<term2>**. Effettua la somma fra due valori, e copia il risultato nel secondo termine passato, che dovrà essere un registro. Esempio:

```
movl   $1,%eax
addl   $1,%eax
```

Questo codice scrive in EAX il valore 1, quindi gli somma 1. EAX conterrà quindi 2.

- ***sub* <term1>,<term2>**. Del tutto analoga alla somma come operazione, effettua la differenza. Esempio:

```
movl   $1,%eax
subl   $1,%eax
```

Questo codice scrive in EAX il valore 1, quindi gli sottrae 1. EAX conterrà quindi 0.

- ***mul* <term>**. Effettua il prodotto fra due numeri. Attenzione: questo opcode prende un solo parametro. Infatti, il moltiplicando è sempre contenuto in EAX, e anche il risultato verrà salvato in EAX (in EDX:EAX nel caso di numeri maggiori di 2^32). Quello che va invece specificato è il moltiplicatore, che deve essere un numero contenuto in un registro. Esempio:

```
movl   $3,%eax
movl   $2,%ebx
mull   %ebx
```

In questo caso metto in EAX il valore 3 e in EBX 2. Richiamo quindi l'operatore di moltiplicazione, specificando come moltiplicatore EBX. Il programma moltiplica dunque 2 per il valore contenuto in EAX e salva il risultato in EAX, che quindi conterrà 6.

- ***div* <term>**. Analogo a *mul*, effettua la divisione. La sintassi è praticamente identica. La differenza sta nel fatto che *div* piazza il quoziente nelle cifre meno significative di EAX, e il resto in EDX.

```
movl   $3,%eax
movl   $2,%ebx
divl   %ebx
```

In questo caso AL conterrà 1 (quoziente della divisione fra 3 e 2), e EDX conterrà 1 (resto della divisione).

- ***and* <term1>,<term2>**. Effettua l'AND logico fra due termini e piazza il risultato nel secondo termine, che deve essere un registro.

```
movl   $0xe4,%eax
andl   $0xf0,%eax
```

Dopo l'operazione, EAX conterrà 0xe0 (risultato dell'AND fra 0xe4 e 0xf0).

- ***or*** **<term1>,<term2>**. Completamente analogo all'AND, effettua l'OR logico fra due termini mettendo il risultato nel secondo termine.
- ***xor*** **<term1>,<term2>**. Completamente analogo all'AND, effettua lo XOR logico fra due termini mettendo il risultato nel secondo termine. È molto usato per svuotare il contenuto di un registro, dato che lo XOR di un valore logico con se stesso ritorna sempre 0:

```
movl   $1,%eax
xorl   %eax,%eax   # EAX conterrà 0
```

- ***not*** **<term>**. Calcola il complemento a 1 del valore contenuto in un registro e piazza il risultato nel registro stesso.

```
movl   $0x0000ffff,%eax
notl   %eax  # EAX=0xffff0000
```

- ***shll*** **<term1>,<term2>**. Effettua uno *shift* a sinistra di tanti bit quanti sono indicati nel primo termine del secondo termine (che deve essere un registro) e salva il risultato in esso. Esempio:

```
movl   $0x0000ffff,%eax
shll   $16,%eax
# Effettuo uno shift a sinistra di 2 byte del valore contenuto in
EAX,
# che alla fine conterrà quindi $0xffff0000
```

- ***shrl*** **<term1>,<term2>**. Analogo a *shl*, ma effettua lo shift a destra anziché a sinistra.

Esempio pratico: inversione di una stringa

Con le conoscenze che abbiamo maturato finora (syscall, variabili, salti condizionati, operazioni logico-aritmetiche) consideriamo un piccolo esempio che non fa altro che prendere una stringa da stdin, invertire fra loro i caratteri al loro interno e stampare quindi la stringa ribaltata. Il codice è abbastanza commentato e con le conoscenze che abbiamo ora non dovrebbe essere un problema capirlo:

```
.data
  // La mia stringa
  str:            .space  30

  // Lunghezza di str
  s_len = .-str

  // Stringa per il carattere a capo
  nline:  .string "\n"

.text
        .global main
main:
        # Leggo la stringa da stdin (sys_read)
        movl            $3,%eax
        movl            $0,%ebx
        movl            $str,%ecx
        movl            $s_len,%edx
        int             $0x80

        # In %edi metto la lunghezza della stringa
        movl            $s_len,%edi

loop:
        # Metto l indirizzo di $str[%edi] in %ecx
        movl            $str,%ecx
        addl            %edi,%ecx

        # Stampo $str[%edi] su stdout (sys_write)
        movl            $4,%eax
        movl            $1,%ebx
        movl            $1,%edx
        int             $0x80

        # Decremento %edi. Una volta arrivato a 0, esco dal ciclo
        decl            %edi
        cmpl            $-1,%edi
        jne             loop

        # sys_exit(0)
```

```
movl            $1,%eax
movl            $0,%ebx
int             $0x80
```

Stack

Al momento della creazione di ogni processo, il sistema operativo assegna a quest'ultimo un'area di memoria, chiamata *stack*, nella quale il processo potrà salvare le sue variabili locali, eventuali dati temporanei e chiamate a funzione. L'indirizzo base dello stack, su un sistema Linux, è all'indirizzo di memoria 0xc0000000, e da lì gli indirizzi vanno a decrescere. Lo stack ha inoltre una struttura **LIFO** (*Last in, first out*), ovvero l'ultimo dato immesso su di esso è in genere il primo a essere prelevato. Lo possiamo proprio concettualmente vedere come una pila, in cui di volta in volta infilo un nuovo oggetto spostandomi verso l'alto, e il primo oggetto che andrò ad estrarre dall'alto sarà proprio l'ultimo che ho inserito. Ovviamente questo esempio è solo da prendere a livello concettuale, dato che in realtà come abbiamo appena visto gli indirizzi sullo stack non vanno dal basso verso l'alto ma al contrario, partendo dall'indirizzo base 0xc0000000 e andando a decrescere man mano che vengono inseriti nuovi oggetti, ma questo non modifica molto l'esempio concettuale appena proposto. Al massimo possiamo vedere lo stack come una pila al contrario che 'sfida' le leggi fisiche di gravità.

A livello hardware, e quindi di codice macchina, possiamo gestire lo stack attraverso due registri della CPU:

- **ESP** - *Stack pointer*. È il registro che punta all'attuale cima dello stack, ovvero l'indirizzo corrente a cui si trova lo stack dell'applicazione. Supponiamo ad esempio di avere uno stack completamente vuoto e di cominciare dall'indirizzo 0xc0000000 (nella realtà non capiterà mai una situazione del genere, dato che i processi cominciano salvando automaticamente sullo stack informazioni sulle chiamate di funzioni principali). Se salviamo un int a 4 byte sullo stack in questa situazione, la nostra variabile verrà memorizzata, la cima dello stack attuale sarà all'indirizzo 0xc0000000 - 4 = 0xbffffffc, e quindi ESP dopo il salvataggio della variabile conterrà il valore 0xbffffffc. Abbiamo quindi imparato una cosa fondamentale nella gestione dello stack: per scrivere un valore sullo stack basta decrementare ESP di tante unità quanti sono i byte da scrivere sullo stack, quindi scrivere il valore da salvare sull'indirizzo puntato da ESP. Esempio:

```
subl   $4,%esp     ; Sottraggo 4 byte alla cima dello stack
movl   $1,(%esp)   ; Salvo il valore 0x00000001 sullo stack (4 byte)
```

L'ISA Intel mette a disposizione una sola istruzione per compiere questa operazione: **push**. Semplicemente, richiamo l'istruzione push passando come argomento il valore o il registro da salvare sullo stack, e automaticamente decrementa lo stack pointer di tante unità quanti sono i byte da salvare e scrive sul nuovo indirizzo puntato da ESP il valore. La scrittura di sopra si può tranquillamente condensare in un

```
pushl   $1
```

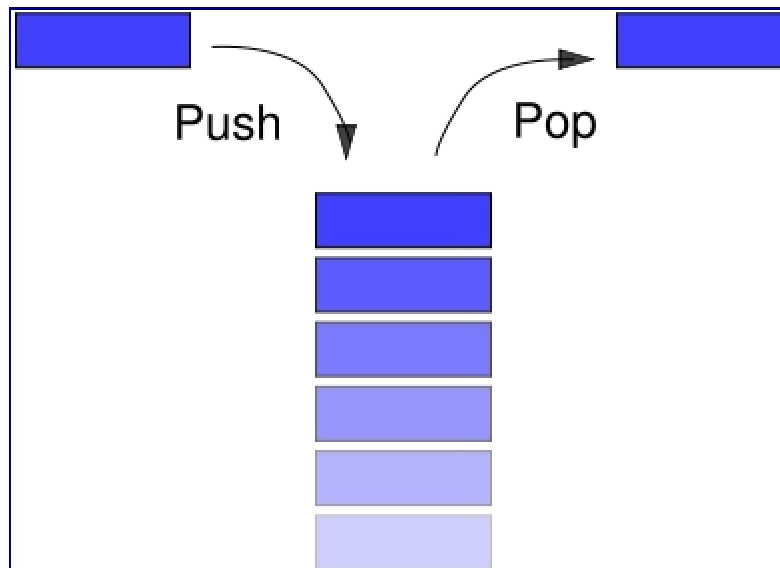
Analogalmente, per rimuovere a livello logico un dato dallo stack basta sommare alla cima dello stack tante unità quanti sono i byte che si vogliono rimuovere. Alla scrittura successiva sullo stack, verrà preso l'indirizzo puntato da ESP e i nuovi dati verranno scritti lì, sovrascrivendo quindi i dati precedenti. Possiamo anche salvare l'attuale cima dello stack su un registro e rimuovere dallo stack i dati appena letti in questo modo. Basta copiare il valore puntato da ESP in un registro e sommare ad ESP tante unità quanti sono i byte letti. Tornando all'esempio di prima, possiamo scrivere un int sullo stack e poi andare a leggere la cima dallo stack e salvare il valore lì puntato su un registro in questo modo:

```
movl    (%esp),%eax    ; Copio l'attuale valore presente in cima allo stack (4 byte) in EAX
addl    $4,%esp         ; Sommo 4 byte alla cima dello stack, dicendo al sistema che quello spazio è ora libero
```

Anche qui, la ISA Intel mette a disposizione una sola istruzione per effettuare quest'operazione: **pop**. La sintassi, semplicemente, prevede che alla pop si passi il registro in cui salvare la cima dello stack. Il codice di sopra è perfettamente equivalente ad una

```
popl   %eax
```

A livello concettuale quindi le due istruzioni rispettivamente salvano un elemento sulla cima dello stack e prelevano il valore attualmente presente in cima allo stack per salvarlo in un registro.



Attenzione però a ricordare sempre le caratteristiche LIFO dello stack. Se effettuo un salvataggio di dati in quest'ordine

```
pushl   $1
pushl   $2
pushl   $3
```

i dati verranno poi prelevati dallo stack in ordine inverso, ovvero prima 3, poi 2, poi 1, in quanto viene sempre prelevato per primo l'ultimo elemento inserito, in quanto rappresenta la cima dello stack. Inoltre, è in genere buona norma, quando i byte scritti sullo stack non servono più, deallocarli, o effettuando tante pop quante sono le push, oppure sommando a ESP tante unità quanti sono i byte scritti, in modo da minimizzare l'occupazione di questa zona di memoria.

- **EBP** - *base pointer*. Questo registro contiene l'indirizzo di base dello stack per il processo corrente. Inizialmente, all'avvio del processo viene scritto in EBP il suo indirizzo base dello stack, quindi tale valore viene copiato in ESP. A questo punto EBP rimane in genere non toccato, mentre invece ESP può essere incrementato o decrementato partendo dal valore base ogni volta che vengono salvati o prelevati valori sullo stack.

Ci sono inoltre altre due istruzioni che tornano molto utili quando si devono scrivere righe di codice Assembly da integrare in progetti già esistenti e in modo da ridurne l'impatto: **pusha** e **popa**. Queste due istruzioni rispettivamente salvano sullo stack la situazione attuale dei registri, e prelevano la situazione dei registri salvata precedentemente sullo stack ripristinandola. Esempio classico di utilizza:

```
; Frammento di codice ASM richiamato dall'esterno

pusha   ; Salvo sullo stack la situazione attuale dei registri

; Codice eseguito dalla procedura

popa        ;  Ripristino  la  situazione  iniziale  dei  registri
prelevandola dallo stack
```

Ora possiamo anche capire come vengono gestiti a basso livello gli array nei linguaggi di programmazione ad alto livello. Chi viene dal C saprà che in questo linguaggio un array non è altro che un puntatore tipizzato al primo elemento in esso contenuto. Questa caratteristica rispecchia proprio quello che accade a basso livello: un array non è altro che una lista di elementi dello stesso tipo. Quando un compilatore incontra la definizione di un array, salva tutti i suoi elementi, ovviamente in ordine inverso, sullo stack. Ad ogni elemento inserito sullo stack il registro ESP viene incrementato di tante unità quanti i byte scritti, e complessivamente, se ho un array di *n* elementi,

```
ESP = ESP + n*(dimensione singolo elemento)
```

Quello che interessa a me programmatore di alto livello è sapere a che indirizzo di memoria è salvato l'array, quindi, dopo la fase di inserimento, mi salvo da qualche parte la cima dello stack, che rappresenta l'indirizzo del primo elemento del mio array. Esempio: l'allocazione di un array di questo tipo in C

```
int v[] = { 0,1,2,3,4 };
```

viene riscritta in Assembly come

```
v:      .long   0   ; Variabile che conterrà l'indirizzo del primo
elemento del vettore
```

```
......

pushl   $4       ; Salvo gli elementi del vettore sullo stack
pushl   $3
pushl   $2
pushl   $1
pushl   $0
movl    %esp,v   ; Copio l'attuale cima dello stack in v
```

Ora *v* conterrà l'indirizzo del primo elemento del nostro vettore, e possiamo leggere gli elementi successivi semplicemente incrementando il suo valore. Se infatti ora andiamo a leggere 5 int a partire da *v* usando un debugger otterremo proprio

```
(gdb) x/5x v
0xbf92a378:        0x00000000          0x00000001          0x00000002
0x00000003
0xbf92a388:     0x00000004
```

Istruzione *call*: Chiamate a funzioni in Assembly

La *call* è l'istruzione a basso livello usata per richiamare una qualsiasi funzione. Una funzione, a basso livello, viene trattata come una semplice etichetta, ad esempio una printf sarà qualcosa del tipo

```
printf:
   istruzioni
   .......
```

e una call è molto simile concettualmente ad una semplice *jmp*. La differenza è che la call prima di saltare all'etichetta indicata salva sullo stack l'indirizzo dell'istruzione successiva da eseguire, contenuto nel registro EIP, in modo da sapere da dove riprendere l'esecuzione del codice quando la funzione richiamata ritorna, quindi effettua il salto vero e proprio. A livello concettuale, un

```
call   func
```

è equivalente a un

```
push   %eip   ; Salvo l'indirizzo da cui ripartire dopo la chiamata
della funzione
jmp     func    ; Salto all'etichetta contenente il codice della
funzione
```

Ovviamente questo codice è valido solo a livello concettuale...l'assemblatore non lo accetterà mai in quanto il registro EIP non è direttamente modificabile dal programmatore, anzi non è nemmeno visibile dal codice, ma è un pezzo di codice che serve per capire cosa succede a basso livello quando nel codice viene incontrata una *call*. Allo stesso modo, la funzione sarà strutturata nel seguente modo:

```
func:
   .......
```

```
   .......
   ret
```

Il *ret* finale dice di ritornare al chiamante. Semplicemente, riprende dallo stack l'indirizzo salvato in precedenza dal chiamante e salta lì.

Passaggio di parametri a funzioni

Ora è anche comprensibile come funziona a basso livello il passaggio di parametri a funzioni. Un parametro è semplicemente un valore che viene salvato sullo stack prima della chiamata della funzione, e può essere poi prelevato direttamente dallo stack all'interno del codice stesso della funzione. Esempio, se ad una mia funzione voglio passare il valore 1:

```
pushl   $1
call    func

.......

func:
movl    8(%esp),%eax
```

Semplicemente, salvo sullo stack il valore che voglio passare e richiamo la funzione. A questo punto prendo il valore salvato all'indirizzo [ESP+8] e lo copio in EAX: vedremo che in EAX sarà presente proprio il valore 1 che il chiamante ha passato alla funzione. Dovrebbe anche essere chiaro perché per prelevare il primo parametro passato alla funzione vado a leggere all'indirizzo [ESP+8]...il chiamante effettua una push del parametro da passare, ma subito dopo c'è una call, che a sua volta effettua un'altra push, salvando sullo stack il valore di EIP (indirizzo a cui riportare l'esecuzione del programma una volta terminata la funzione). Quando entro nel codice di *func* la situazione dello stack sarà quindi qualcosa del tipo

```
+-------------------------------+
| Indirizzo di ritorno (4 byte) |  <--- ESP+4
+-------------------------------+
|       Parametro passato       |  <--- ESP+8
+-------------------------------+
```

Poiché ESP punta alla cima dello stack, punterà all'indirizzo in cui è salvato l'indirizzo di ritorno. Se invece voglio leggere il parametro passato, devo andare a leggere il valore presente a ESP+8. Se alla funzione volessi passare invece di uno, due parametri interi (quindi a 8 byte), il primo sarebbe situato a [ESP+8], e il secondo a [ESP+12].

È proprio in questo modo che vengono passati i parametri alle funzioni che richiamiamo quotidianamente dai nostri codici scritti in linguaggio ad alto livello. Ad esempio, una

```
printf ("%d\n",n);
```

in Assembly si traduce come una

```
format:   .string   "%d\n"
n:        .long   (valore)

......

pushl   n
pushl   $format
call    printf
addl    $8,%esp
```

I parametri, come al solito, vengono salvati in ordine inverso. Si noti la buona norma di sommare alla fine, quando i dati salvati sullo stack non servono più, al registro ESP tante unità quanti sono i byte scritti in precedenza, per dire alla macchina che quella zona di memoria è ora libera.

Potete testare, volendo, il codice riportato sopra di persona, includendo in testa al sorgente stdio.h, come si farebbe con un normale codice C, e compilandolo normalmente con gcc. A sorpresa, avrete richiamato una printf in Assembly.

```
#include <stdio.h>

.text
format:   .string  "La variabile n=%d e' all'indirizzo 0x%x\n"
n:        .long    3

.global main
main:
        push    $n
        push     n          ; Si noti la differenza. Con n salvo il
valore di n, con $n il suo indirizzo
        push    $format
        call    printf
        add     $12,%esp  ; Sommo a ESP 4*3=12 byte

        mov     $1,%eax
        mov     $0,%ebx
        int     $0x80

        leave
        ret
```

Valori di ritorno

Possiamo già intuire come funzioni a basso livello il ritorno di valori di una funzione. Per convenzione, il valore di ritorno viene scritto dalla funzione in EAX. Il codice della funzione che abbiamo visto prima quindi

```
foo:
        mov    8(%esp),%eax
        ret
```

non fa altro che prendere il parametro passato alla funzione, salvarlo in EAX e ritornare. Quindi, ad alto livello potrebbe corrispondere al codice di una funzione del

tipo

```
int foo (int n)  {
  return n;
}
```

I/O su periferiche

L'Assembly consente di gestire direttamente la lettura e la scrittura di dati su una periferica noto l'indirizzo attraverso le istruzioni **IN** e **OUT**. Attenzione: ciò è consentito solo se il sistema operativo non si trova in *protected mode* (ad esempio, sui sistemi Windows della serie NT, dal 2000 in poi, è stato disabilitato l'accesso diretto all'I/O). In tal caso, è necessario ricorrere alle API del sistema operativo che effettueranno la richiesta in kernel space per noi. Altrimenti, è possibile gestire anche dallo user space l'input e l'output su periferiche attraverso queste istruzioni. Se il sistema operativo lo permette inoltre, a partire dal Pentium II sono state introdotte due istruzioni che consentono rispettivamente di entrare o uscire dal protected mode:

```
sysenter   ; Porta la CPU in protected mode
sysexit    ; Disabilita il protected mode ed entra in user mode
```

Qui invece la sintassi di IN e OUT:

```
in   src,dst
out  src,dst
```

Per convenzione, nella IN src è DL, DX o EDX, a seconda che si voglia leggere un dato da una porta il cui indirizzo è esprimibile a 8, 16 o 32 bit, e dst è generalmente AL, AX o EAX. Stessa convenzione per la OUT (stavolta src, che conterrà il dato da scrivere, sarà AL/AX/EAX, mentre dst sarà DL/DX/EDX). Esempio: un paio di righe di codice che leggono il valore attualmente presente sulla porta parallela (in genere all'indirizzo 0x378), lo incrementa di 1 e lo riscrive. Si noti che la porta parallela ha un parallelismo a 8 bit, quindi il codice leggerà e scriverà un byte per volta:

```
mov   $0x378,%dx
in    %dx,%ax
inc   %ax
out   %ax,%dx
```

Gestione di file in Assembly

I file in Assembly sono gestiti dalle consuete primitive per la gestione dell'I/O richiamabili anche dal C (*open*, *read*, *write*, *close*) attraverso le chiamate all'interrupt 0x80. Le modalità di richiamo delle primitive sono le consuete:

```
mov   $codice_syscall,%eax
mov   $primo_arg,%ebx
mov   $secondo_arg,%ecx
mov   $terzo_arg,%edx
int   $0x80
```

Procedendo per passi, sappiamo che un'apertura di file in C usando la *open* sarebbe qualcosa del tipo

```
open ("nome_file",modo);
```

dove *modo* è un codice numerico, richiamabile attraverso una delle macro definite in */usr/include/asm/fcntl.h*, */usr/include/asm-generic/fcntl.h*, */usr/include/sys/fcntl.h* o simili, a seconda della distribuzione (es., *O_RDONLY*, *O_WRONLY*, *O_TRUNC* ecc., e si possono mettere anche più modalità in serie facendo la somma logica fra loro).

Controllando in */usr/include/asm/unistd.h* o simili vedremo che alla *open* è associata la funzione 5 dell'interrupt 0x80:

```
#define __NR_open                 5
```

mentre invece, per vedere qual è il valore della modalità *O_RDONLY* (file aperto in lettura), basta dare un'occhiata a */usr/include/asm/fcntl.h* o simili:

```
#define O_RDONLY        00000000
```

Per aprire un certo file in lettura attraverso la *open* si ricorrerà quindi alle istruzioni

```
mov   $5,%eax
mov   $nome_file,%ebx
mov   $0,%ecx
int   $0x80
```

Effettuata la chiama alla primitiva, il valore di ritorno verrà messo in EAX. Sappiamo che il valore di ritorno sarà il descrittore del file in caso di apertura avvenuta con successo (sappiamo che il descrittore di un file aperto in questo modo sarà un numero >= 3, dato che i descrittori 0, 1 e 2 sono riservati, rispettivamente, a stdin, stdout ed stderr), e -1 in caso di insuccesso. Quindi, immediatamente dopo la syscall effettuiamo un controllo sull'avvenuta apertura:

```
cmp   $0,%eax
jl    etichetta_errore
; altrimenti continuiamo, salvando in una variabile
```

```
; il descrittore del file appena aperto
mov   %eax,fd
```

Le operazioni di lettura e scrittura avverranno attraverso le *read* e le *write* nel modo che abbiamo già visto finora, con la sola differenza che invece di scrivere o leggere su stdin ed stdout passeremo in EBX il descrittore del file appena aperto. Esempio di lettura:

```
mov   $3,%eax
mov   fd,%ebx
mov   $var,%ecx
mov   $len,%edx
int   $0x80
```

La chiusura della risorsa invece in C andrebbe fatta come

```
close(fd);
```

Guardando nuovamente all'elenco dei codici delle syscall, vediamo che alla *close* corrisponde la funzione 6 dell'interrupt 0x80:

```
#define __NR_close                6
```

quindi faremo semplicemente un

```
mov   $6,%eax
mov   fd,%ebx
int   $0x80
```

Vediamo ora un esempio completo: un sorgente in grado di leggere un file carattere per carattere e stamparne il contenuto in output, un po' un piccolo porting del comando per Unix *cat*:

```
.data
; Stringa contenente il nome del file
file:   .string   "nome_file"

; Variabile di appoggio che conterrà il carattere letto
buff:   .space    1

; Descrittore del file, intero inizializzato a 0
fd:     .long     0

.text
        .global main
main:
        ; fd = open(file,O_RDONLY);
        mov             $5,%eax
        mov             $file,%ebx
        mov             $0,%ecx
        int             $0x80
        mov             %eax,fd

        ; if (fd<0)
        ;    goto end;
```

```
        cmp             $0,%eax
        jl              end

        ; do  {
        ;   esi = read (fd,&buff,1);
        ;   write (1,&buff,1);
        ; } while (esi>0);
loop:
        ; read
        mov             $3,%eax
        mov             fd,%ebx
        mov             $buff,%ecx
        mov             $1,%edx
        int             $0x80
        mov             %eax,%esi

        ; write
        mov             $4,%eax
        mov             $1,%ebx
        mov             $buff,%ecx
        mov             $1,%edx
        int             $0x80

        ; controllo sul valore di ritorno di read
        cmp             $0,%esi
        jg              loop

        ; arrivo qui solo quando il ciclo è terminato
        ; oppure la lettura non è avvenuta con successo
end:
        ; exit(0);
        mov             $1,%eax
        mov             $0,%ebx
        int             $0x80

        leave
        ret
```

Inline Assembly

Raramente ci si ritrova a scrivere interi listati di grandi dimensioni in Assembly. Un uso comune dell'Assembly è piuttosto quello di essere integrato all'interno di listati scritti in un linguaggio di livello superiore (es. C o Pascal), in particolare per gestire routine critiche, passaggi in cui è richiesta un'elevata ottimizzazione o interfacciarsi a basso livello con routine non gestibili dal codice ad alto livello. Vedremo come operare quest'integrazione nei listati C usando *gcc*, attraverso la pratica nota come scrittura di codice **inline Assembly**.

L'ANSI C prevede l'uso della keyword **__asm__** per identificare blocchi di codice scritti in Assembly all'interno del listato in C. Tale codice va scritto fra parentesi tonde e come stringa. Esempio: questa routine inserita in un codice C sarà l'equivalente di un *__exit(0)*:

```
__asm__(
  "mov   $1,%eax\n"
  "mov   $0,%ebx\n"
  "int   $0x80"
);
```

La cosa più interessante però è passare al codice Assembly dei valori. È infatti poco utile una routine in inline Assembly che non può comunicare con il codice circostante. Per questo fine, entra in gioco il cosiddetto **extended Assembly**. La sintassi è la seguente:

```
__asm__(
  "codice ASM"
  : operatori_di_output
  : operatori_di_input
  : registri_clobbered
);
```

Vediamo subito un esempio pratico:

```
#include <stdio.h>

main()  {
        int a=10,b=2;

        printf ("b vale %d\n",b);

        __asm__(
                        "mov   %1,%%eax\n"
                        "mov   %%eax,%0\n"
                        :"=r"(b)
                        :"r"(a)
                        :"%eax"
                  );
```

```
        printf ("...ora b vale %d\n",b);
}
```

Si noti innanzitutto che nell'extended Assembly i registri vogliono il prefisso *%%* invece del consueto '%', per distinguerli dagli operandi.

Nel codice abbiamo un parametro di output, ovvero sul quale il codice Assembly andrà a scrivere, che è la variabile *b*:

```
:"=r"(b)
```

un parametro di input, che è quello che il codice potrà andare a leggere, che è la variabile *a*:

```
:"r"(a)
```

e un registro *clobbered*, che è EAX:

```
:"%eax"
```

Un registro indicato come *clobbered* è un registro che è usato all'interno del codice per memorizzare i dati, e quindi diciamo al compilatore di non utilizzarlo come registro di appoggio per memorizzare dati temporanei, e quindi modificarlo, finché è in esecuzione il codice Assembly (e infatti la lettura e la scrittura usano come registro di appoggio, nel codice preso in esame, proprio EAX, una modifica di questo registro da parte del compilatore porterebbe a dati in uscita falsati).

Con *%0* viene identificato invece il parametro di output (*b*), e con *%1* quello di input (*a*). Dati *m* parametri di output e *n* parametri di input, i parametri di output verranno identificati da *%0,%1,...,%(m-1)*, mentre i parametri di input verranno identificati da *%m,%(m+1),...,%(m+n-1)*. Quindi, a livello logico

```
"mov   %1,%%eax\n"
"mov   %%eax,%0\n"
```

equivale a

```
"mov   a,%%eax\n"
"mov   %%eax,b\n"
```

L'effetto sarà quello di copiare il valore di *a* all'interno della variabile *b*. L'output del programma sarà quindi

```
b vale 2
...ora b vale 10
```

Altro esempio:

```
#include <string.h>

main()  {
        char *s = "Hello world\n";
```

```
        __asm__(
                        "mov            $4,%%eax\n"
                        "mov            $1,%%ebx\n"
                        "mov            %0,%%ecx\n"
                        "mov            %1,%%edx\n"
                        "int            $0x80"
                        :
                        :"g"(s), "g"(strlen(s))
                        :"%eax","%ebx","%ecx","%edx"
                );
}
```

In questo caso non abbiamo parametri di output, in quanto vogliamo semplicemente fare la *write* di una stringa. Abbiamo però due parametri di input (la stringa s e la sua relativa lunghezza), identificati rispettivamente da *%0* e *%1*, e, poiché utilizziamo tutti i registri all-purpose, diciamo al compilatore di non toccare EAX, EBX, ECX ed EDX.

Altro esempio in cui possiamo vedere anche come vengono gestite a basso livello le funzioni:

```
#include <stdio.h>

int foo (int a)  {
        __asm__(
                        "mov            8(%esp),%eax\n"
                        "inc            %eax"
                );
}

main()  {
        int x=1;
        printf ("Prima di foo: x=%d\n",x);

        x=foo(x);
        printf ("Dopo foo: x=%d\n",x);
}
```

L'output sarà:

```
Prima di foo: x=1
Dopo foo: x=2
```

Quello che fa la funzione foo è quindi prendere il suo argomento, incrementarlo e ritornarlo, l'equivalente di un

```
int foo (int a)  {
  return a+1;
}
```

Quello che ci interessa però è vedere come preleva l'argomento passato. Sappiamo infatti che gli argomenti che vengono passati a una funzione vengono salvati sullo stack in ordine inverso. Bene, al momento della chiamata di foo la cima dello stack (ovvero il valore contenuto in ESP) conterrà un valore qualsiasi. Se sommiamo a ESP

un numero di unità pari al parallelismo della memoria (ovvero, nel caso di una macchina a 32 bit, ESP+4), otteniamo l'indirizzo del chiamante, in modo che il codice sappia dove ritornare quando la funzione è terminata. Se gli sommiamo ancora 4 (ESP+8), otteniamo il primo argomento della funzione, se sommiamo ancora 4 (ESP+12) conterrà il secondo argomento, e così via. Provare con gdb per credere.

```
(gdb) x $esp+4      <-- ESP+4: Indirizzo a cui tornare a funzione terminata
0xbff51b34:     0x08048396
(gdb) x 0x08048396
0x8048396 <main+53>:    0x8908c483  <-- E infatti si trova nel main
(gdb) x $esp+8     <-- ESP+8: Primo argomento della funzione
0xbff51b38:     0x00000001
```

Quindi, copiamo il primo argomento nel registro EAX e lo incrementiamo. Sappiamo che il valore di ritorno di una funzione viene passato al chiamante nel registro EAX. Bene, quindi al chiamante tornerà il valore di *a* incrementato. E i conti tornano.