64.1 Terminologia impropria

È bene ricordare che nella documentazione standard sui microprocessori x86 si usa una terminologia coerente, ma impropria, riferita alla dimensione delle unità di dati. In particolare, il problema nasce dal fatto che originariamente questi microprocessori avevano parole da 16 bit, così è stato associato il termine *word* a 16 bit ed è rimasta tale l'associazione anche con la trasformazione successiva a 32 bit. Pertanto, generalmente valgono le convenzioni riportate nella tabella successiva.

Tabella 64.1. Terminologia associata alla dimensione dei dati nei microprocessori x86.

Definizione o abbreviazione	Dimensione dei dati	Definizione o abbreviazione	Dimensione dei dati
«b», byte	8 bit	«w», *word*	16 bit
«d», «dw», *dword*, *double word*	32 bit	«q», «qw», *qword*, *quad word*	64 bit

64.2 Registri principali fino ai 32 bit

I registri dei microprocessori x86 sono stati inizialmente da 16 bit; successivamente, quelli principali sono stati estesi a 32 bit. Alcuni registri hanno una funzione ben precisa; gli altri sono utilizzabili per scopi generali, ma in pratica ognuno ha un compito preferenziale.

Tutti i registri che sono stati estesi da 16 bit; a 32 bit; hanno due nomi: uno riferito alla porzione dei 16 bit meno significativi, l'altro che riguarda il registro nel suo complesso. Inoltre, per quattro registri in particolare, è possibile individuare anche i due byte che compongono la parte meno significativa. Per esempio, il registro ***EAX*** ha una dimensione di 32 bit, di cui è possibile individuare i 16 bit meno significativi con il nome ***AX***, ma in più, il nome ***AL*** individua il byte meno significativo di ***AX*** mentre ***AH*** ne individua quello più significativo.

Il registro ***EIP*** (o ***IP*** nei microprocessori a 16 bit) viene gestito automaticamente e serve a contenere l'indirizzo dell'istruzione successiva da eseguire. In effetti, la gestione manuale di tale registro non sarebbe conveniente, dal momento che la dimensione delle istruzioni in linguaggio macchina varia in base al tipo e agli operandi.

Il registro ***ESP*** (o ***SP*** nei microprocessori a 16 bit) individua l'ultimo elemento della pila dei dati e può essere gestito manualmente, sapendo che questo indice si deve spostare a gruppi di quattro byte (o due byte nella versione a 16 bit) e che la pila cresce diminuendo l'indice.

Il registro ***EBP*** (o ***BP*** nei microprocessori a 16 bit) si affianca all'indice della pila, per tenere conto della posizione raggiunta all'inizio di una funzione.

Il registro ***EFLAGS*** (o ***FLAGS*** nei microprocessori a 16 bit) raccoglie i vari indicatori che descrivono l'esito delle operazioni svolte. In particolare sono importanti gli indicatori che appaiono nella tabella 64.3.

Figura 64.2. Registri principali.

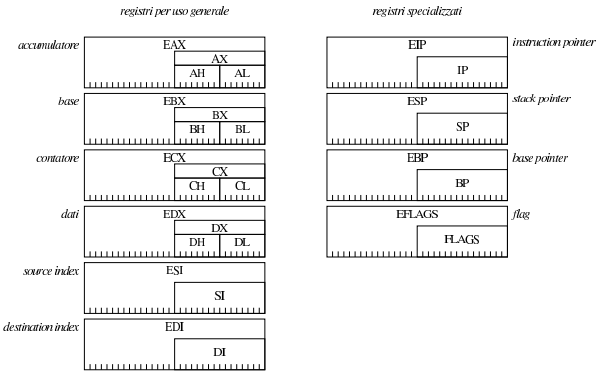


Tabella 64.3. Gli indicatori principali, contenuti nel registro ***FLAGS***.

Indicatore (*flag*)	Descrizione
C *carry*	È l'indicatore del riporto per le operazioni con valori senza segno. In particolare si attiva dopo una somma che genera un riporto e dopo una sottrazione che richiede il prestito di una cifra (in tal caso si chiama anche *borrow*).
O *overflow*	È l'indicatore di traboccamento per le operazioni che riguardano valori con segno.
Z *zero*	Viene impostato dopo un'operazione che dà come risultato il valore zero.
S *sign*	Riproduce il bit più significativo di un valore, dopo un'operazione. Se il valore è da intendersi con segno, l'indicatore serve a riprodurre il segno stesso.
P *parity*	Si attiva quando l'ultima operazione produce un risultato i cui otto bit meno significativi contengono una quantità pari di cifre a uno.

64.3 Sintesi delle istruzioni principali

Nelle tabelle successive vengono annotate le istruzioni più semplici che possono essere utilizzate con i microprocessori x86, raggruppate secondo il contesto a cui appartengono. In modo particolare sono assenti le istruzioni per i calcoli in virgola mobile e quelle per la gestione delle stringhe.

L'ordine in cui sono specificati gli operandi è quello «Intel», ovvero appare prima la destinazione e poi l'origine. Le sigle usate per definire i tipi di operandi sono: ***reg*** per «registro»; ***mem*** per «memoria»; ***imm*** per «immediato» (costante numerica).

Nella colonna degli indicatori appare: il simbolo «#» per annotare che l'indicatore relativo può essere modificato dall'istruzione; il simbolo «t» per annotare che lo stato precedente dell'indicatore viene considerato dall'istruzione; zero o uno se l'indicatore viene impostato in un certo modo; il simbolo «?» se l'effetto dell'istruzione sull'indicatore è indefinito.

Tabella 64.4. Assegnamenti, scambi, conversioni e istruzione nulla.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`NOP`		Istruzione nulla.	`c p z s o` `. . . . .`
`MOV`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Copia il valore dell'origine nella destinazione. Origine e destinazione devono avere la stessa quantità di bit. ***dst*** := ***org***	`c p z s o` `. . . . .`
`LEA`	***reg32***, ***mem***	*load effective address* Mette nel registro l'indirizzo della memoria. ***dst*** := indirizzo(***org***)	`c p z s o` `. . . . .`
`MOVSX`	***reg16***, ***reg8*** ***reg16***, ***mem8*** ***reg32***, ***reg8*** ***reg32***, ***mem8*** ***reg32***, ***reg16*** ***reg32***, ***mem16***	Tratta il valore nell'origine come un numero con segno e lo estende in modo da occupare tutto lo spazio della destinazione.	`c p z s o` `. . . . .`
`MOVZX`	***reg16***, ***reg8*** ***reg16***, ***mem8*** ***reg32***, ***reg8*** ***reg32***, ***mem8*** ***reg32***, ***reg16*** ***reg32***, ***mem16***	Tratta il valore nell'origine come un numero senza segno e lo estende in modo da occupare tutto lo spazio della destinazione.	`c p z s o` `. . . . .`
`XCHG`	***reg***, ***reg*** ***reg***, ***mem*** ***mem***, ***reg***	Scambia i valori. ***dst*** :==: ***org***	`c p z s o` `. . . . .`
`CBW`		Converte un intero con segno, della dimensione di 8 bit, contenuto in ***AL***, in modo da occupare tutto ***AX*** (da 8 bit a 16 bit). L'espansione tiene conto del segno. ***AX*** := ***AL***	`c p z s o` `. . . . .`
`CWDE`		Converte un intero con segno, della dimensione di 16 bit, contenuto in ***AX***, in modo da occupare ***EAX*** (da 16 bit a 32 bit). L'espansione tiene conto del segno. ***EAX*** := ***AX***	`c p z s o` `. . . . .`
`CDQ`		Converte un intero con segno, della dimensione di 32 bit, contenuto in ***EAX***, in modo da occupare la somma di ***EDX***:***EAX*** (da 32 bit a 64 bit). L'espansione tiene conto del segno. ***EDX***:***EAX*** := ***EAX***	`c p z s o` `. . . . .`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`BSWAP`	***reg32***	Inverte l'ordine dei byte contenuti nel registro: quello meno significativo diventa il più significativo; la coppia interna si scambia.	`c p z s o` `. . . . .`

Tabella 64.5. Operazioni aritmetiche.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`NEG`	***reg*** ***mem***	Inverte il segno di un numero, attraverso il complemento a due. ***dst*** := −***dst***	`c p z s o` `# # # # #`
`ADD`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Somma di interi, con o senza segno, ignorando il riporto precedente. Se i valori si intendono con segno, è importante l'esito dell'indicatore di traboccamento (*overflow*), se invece i valori sono da intendersi senza segno, è importante l'esito dell'indicatore di riporto (*carry*). ***dst*** := ***org*** + ***dst***	`c p z s o` `# # # # #`
`SUB`	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Sottrazione di interi con o senza segno, ignorando il riporto precedente. ***dst*** := ***org*** − ***dst***	`c p z s o` `# # # # #`

`ADC`	***reg***, ***reg***
	reg, ***mem***
	reg, ***imm***
	mem, ***reg***
	mem, ***imm***
		Somma di interi, con o senza segno, aggiungendo anche il riporto precedente (l'indicatore *carry*).
		dst := ***org*** + ***dst*** + ***c***
			`c p z s o`
			`t . . . .`
			`# # # # #`

`SBB`	***reg***, ***reg***
	reg, ***mem***
	reg, ***imm***
	mem, ***reg***
	mem, ***imm***
		Sottrazione di interi, con o senza segno, tenendo conto del «prestito» precedente (l'indicatore *carry*).
		dst := ***org*** + ***dst*** − ***c***
			`c p z s o`
			`t . . . .`
			`# # # # #`

`INC`	***reg*** ***mem***	Incrementa di una unità un intero. ***dst***++	`c p z s o` `. # # # #`
`DEC`	***reg*** ***mem***	Decrementa di una unità un valore intero. ***dst***−−	`c p z s o` `. # # # #`
`MUL`	***reg*** ***mem***	Moltiplicazione intera senza segno. L'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. ***AX*** := ***AL*src*** ***DX:AX*** := ***AX*src*** ***EDX:EAX*** := ***EAX*src***	`c p z s o` `# ? ? ? #`
`DIV`	***reg*** ***mem***	Divisione intera senza segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. ***AL*** := ***AX***/***src*** ***AH*** := ***resto*** ***AX*** := ***DX:AX***/***src*** ***DX*** := ***resto*** ***EAX*** := ***EDX:EAX***/***src*** ***EDX*** := ***resto***	`c p z s o` `? ? ? ? ?`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
IMUL	***reg*** ***mem***	Moltiplicazione intera con segno. In questo caso l'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. ***AX*** := ***AL*src*** ***DX:AX*** := ***AX*src*** ***EDX:EAX*** := ***EAX*src***	c p z s o # ? ? ? #
IDIV	***reg*** ***mem***	Divisione intera con segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. ***AL*** := ***AX***/***src*** ***AH*** := ***resto*** ***AX*** := ***DX:AX***/***src*** ***DX*** := ***resto*** ***EAX*** := ***EDX:EAX***/***src*** ***EDX*** := ***resto***	c p z s o ? ? ? ? ?

Tabella 64.6. Operazioni logiche.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
NOT	***reg*** ***mem***	NOT di tutti i bit dell'operando. ***dst*** := NOT ***dst***	c p z s o . . . . .
AND OR XOR	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	AND, OR, o XOR, tra tutti i bit dei due operandi. ***dst*** := ***org*** AND ***dst*** ***dst*** := ***org*** OR ***dst*** ***dst*** := ***org*** XOR ***dst***	c p z s o 0 # # # 0

Tabella 64.7. Scorrimenti e rotazioni.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
SHL SHR	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Fa scorrere i bit, rispettivamente verso sinistra o verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto). Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # . . . #
SAL	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Funziona esattamente come '**SHL**', ma esiste in quanto è la controparte di '**SAR**', riferendosi a uno scorrimento aritmetico.	c p z s o # . . . #

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
SAR	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Fa scorrere i bit verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto), mantenendo il segno originale. Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # . . . .
RCL RCR	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Ruota i bit, rispettivamente verso sinistra o verso destra, utilizzando anche l'indicatore di riporto (*carry*). Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o t . . . . # . . . #
ROL ROR	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Ruota i bit, rispettivamente verso sinistra o verso destra. Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # . . . #

Tabella 64.8. Chiamate e gestione della pila.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
INT	***imm8***	Esegue una chiamata attraverso un'interruzione.	c p z s o . . . . .
CALL	***reg*** ***mem*** ***imm***	Inserisce nella pila l'indirizzo dell'istruzione successiva e salta all'indirizzo indicato.	c p z s o . . . . .
RET		Estrae dalla pila l'indirizzo dell'istruzione da raggiungere e salta a quella (serve a concludere una chiamata eseguita con '**CALL**').	c p z s o . . . . .
PUSH	***reg*** ***mem***	Inserisce nella pila il valore (della dimensione di un registro comune).	c p z s o . . . . .
POP	***reg*** ***mem***	Estrae dalla pila l'ultimo valore inserito (della dimensione di un registro comune).	c p z s o . . . . .
PUSHF		Inserisce nella pila l'insieme del registro degli indicatori (***FLAGS*** o ***EFLAGS***).	c p z s o . . . . .
POPF		Estrae dalla pila l'insieme del registro degli indicatori (***FLAGS*** o ***EFLAGS***), aggiornando di conseguenza il registro stesso.	c p z s o . . . . .

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
PUSHA PUSHAD		Inserisce nella pila i registri principali: '**PUSHA**' inserisce nella pila i registri da 16 bit, mentre '**PUSHAD**' li inserisce a 32 bit. In sequenza vengono inseriti: ***AX***, ***CX***, ***DX***, ***BX***, ***SP***, ***BP***, ***SI***, ***DI***; ovvero: ***EAX***, ***ECX***, ***EDX***, ***EBX***, ***ESP***, ***EBP***, ***ESI***, ***EDI***. Si osservi che di solito si usa solo '**PUSHA**', lasciando al compilatore la responsabilità di scegliere quale istruzione è appropriata per il contesto.	c p z s o . . . . .
POPA POPAD		Ripristina i registri principali, estraendo i contenuti dalla pila: '**POPA**' ripristina i registri da 16 bit, mentre '**POPAD**' riguarda quelli da 32 bit. In sequenza vengono estratti: ***DI***, ***SI***, ***BP***, ***SP*** viene eliminato senza aggiornare il registro, ***BX***, ***DX***, ***CX***, ***AX***; ovvero: ***EDI***, ***ESI***, ***EBP***, ***ESP*** viene eliminato senza aggiornare il registro, ***EBX***, ***EDX***, ***ECX***, ***EAX***. Come si vede, anche se '**PUSHA**' e '**PUSHAD**' salvano l'indice della pila, in pratica questo indice non viene ripristinato. Si osservi che di solito si usa solo '**POPA**', lasciando al compilatore la responsabilità di scegliere quale istruzione è appropriata per il contesto.	c p z s o . . . . .

Tabella 64.9. Indicatori e confronti.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
CLC		Azzera l'indicatore del riporto (*carry*), senza intervenire negli altri indicatori.	c p z s o 0 . . . .
CMC		Inverte il valore dell'indicatore del riporto (*carry*).	c p z s o # . . . .
CMP	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Confronta due valori interi. La comparazione avviene simulando la sottrazione dell'origine dalla destinazione, senza però modificare gli operandi, ma aggiornando gli indicatori, come se fosse avvenuta una sottrazione vera e propria. ***dst - org***	c p z s o # # # # #
TEST	***reg***, ***reg*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	AND dei due valori senza conservare il risultato. Serve solo a ottenere l'aggiornamento degli indicatori. ***dst*** AND ***org***	c p z s o 0 # # # 0

Tabella 64.10. Salti.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
JMP	***reg*** ***mem*** ***imm***	Salto incondizionato all'indirizzo indicato.	c p z s o . . . . .
JA JNBE	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst*** > ***org*** THEN go to ***imm***	c p z s o t . t . .
JAE JNB	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** >= ***org*** THEN go to ***imm***	c p z s o t . t . .
JB JNAE	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst*** < ***org*** THEN go to ***imm***	c p z s o t . t . .
JBE JNA	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** <= ***org*** THEN go to ***imm***	c p z s o t . t . .
JE	***imm***	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** == ***org*** THEN go to ***imm***	c p z s o . . t . .
JNE	***imm***	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era diversa dall'origine. CMP ***dst***, ***org*** IF ***dst*** =/= ***org*** THEN go to ***imm***	c p z s o . . t . .
JG JNLE	***imm***	Dopo un confronto con segno, salta se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst*** > ***org*** THEN go to ***imm***	c p z s o . . t t t
JGE JNL	***imm***	Dopo un confronto con segno, salta se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** >= ***org*** THEN go to ***imm***	c p z s o . . t t t
JL JNGE	***imm***	Dopo un confronto con segno, salta se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst*** < ***org*** THEN go to ***imm***	c p z s o . . t t t

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`JLE` `JNG`	***imm***	Dopo un confronto con segno, salta se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** <= ***org*** THEN go to ***imm***	`c p z s o` `· · t t t`
`JC` `JNC`	***imm***	Salta se l'indicatore del riporto (*carry*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `t · · · ·`
`JO` `JNO`	***imm***	Salta se l'indicatore di traboccamento (*overflow*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `· · · · t`
`JS` `JNS`	***imm***	Salta se l'indicatore di segno (*sign*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `· · · t ·`
`JZ` `JNZ`	***imm***	Salta se l'indicatore di zero, rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `· · t · ·`
`JP` `JNP`	***imm***	Salta se l'indicatore di parità, rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `· t · · ·`
`JCXZ`	***imm***	Salta se il valore contenuto nel registro ***CX*** è pari a zero.	`c p z s o` `· · · · ·`

Tabella 64.11. Impostazione del valore in base all'esito di un confronto.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SETA` `SETNBE`	***reg8*** ***mem8***	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst*** > ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `· · · · ·`
`SETAE` `SETNB`	***reg8*** ***mem8***	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** >= ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `· · · · ·`
`SETB` `SETNAE`	***reg8*** ***mem8***	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst*** < ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `· · · · ·`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SETBE` `SETNA`	***reg8*** ***mem8***	Dopo un confronto di valori senza segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** <= ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `· · · · ·`
`SETE`	***reg8*** ***mem8***	Dopo un confronto, indipendentemente dal segno, imposta a uno il registro o la memoria indicati, se la destinazione era uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** == ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `· · · · ·`
`SETNE`	***reg8*** ***mem8***	Dopo un confronto, indipendentemente dal segno, imposta a uno il registro o la memoria indicati, se la destinazione era diversa dall'origine. CMP ***dst***, ***org*** IF ***dst*** =/= ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `· · · · ·`
`SETG` `SETNLE`	***reg8*** ***mem8***	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst*** > ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `· · · · ·`
`SETGE` `SETNL`	***reg8*** ***mem8***	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** >= ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `· · · · ·`
`SETL` `SETNGE`	***reg8*** ***mem8***	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst*** < ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `· · · · ·`
`SETLE` `SETNG`	***reg8*** ***mem8***	Dopo un confronto con segno, imposta a uno il registro o la memoria indicati, se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** <= ***org*** THEN ***x***:=1 ELSE ***x***:=0	`c p z s o` `· · · · ·`
`SETC` `SETNC`	***reg8*** ***mem8***	Imposta a uno il registro o la memoria indicati, se l'indicatore del riporto (*carry*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `t · · · ·`

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`SETO` `SETNO`	***reg8*** ***mem8***	Imposta a uno il registro o la memoria indicati, se l'indicatore di traboccamento (*overflow*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `. . . . t`
`SETS` `SETNS`	***reg8*** ***mem8***	Imposta a uno il registro o la memoria indicati, se l'indicatore di segno (*sign*), rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `. . . t .`
`SETZ` `SETNZ`	***reg8*** ***mem8***	Imposta a uno il registro o la memoria indicati, se l'indicatore di zero, rispettivamente, è attivo, oppure non è attivo.	`c p z s o` `. . t . .`

Tabella 64.12. Iterazioni

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`LOOP`	***imm8***	Senza alterare gli indicatori, decrementa di una unità il registro '**`ECX`**' (o solo '**`CX`**', se viene richiesta una dimensione più piccola), quindi, se il registro è ancora diverso da zero, salta all'indirizzo cui fa riferimento l'operando.	`c p z s o` `. . . . .`
`LOOPE` `LOOPZ`	***imm8***	Senza alterare gli indicatori, decrementa di una unità il registro '**`ECX`**' (o solo '**`CX`**', se viene richiesta una dimensione più piccola), quindi, se il registro è ancora diverso da zero e l'indicatore «zero» è attivo, salta all'indirizzo cui fa riferimento l'operando.	`c p z s o` `. . t . .`
`LOOPNE` `LOOPNZ`	***imm8***	Senza alterare gli indicatori, decrementa di una unità il registro '**`ECX`**' (o solo '**`CX`**', se viene richiesta una dimensione più piccola), quindi, se il registro è ancora diverso da zero e l'indicatore «zero» non è attivo, salta all'indirizzo cui fa riferimento l'operando.	`c p z s o` `. . t . .`

64.4 Primo approccio al linguaggio assemblatore per x86

Per scrivere programmi con il linguaggio assemblatore in un sistema GNU/Linux, è necessario disporre del compilatore: serve GNU AS[1] (GAS) per la sintassi AT&T oppure NASM[2] per quella Intel.

64.4.1 Il primo programma

Viene mostrato il listato di un programma molto semplice, il cui scopo è unicamente quello di concludere il proprio funzionamento restituendo un valore, attraverso una funzione del sistema operativo. Ne vengono preparate due versioni: una adatta a GNU AS e l'altra per NASM.

Listato 64.13. File '`fine-gas.s`', adatto per GNU AS.

```
1  #
2  # fine-gas.s
3  #
4  .section .data
5  #
6  .section .text
7  .globl _start
8  #
9  _start:
10     mov  $1, %eax
11     mov  $7, %ebx
12     int  $0x80
```

Listato 64.14. File '`fine-nasm.s`', adatto per NASM.

```
1  ;
2  ; fine-nasm.s
3  ;
4  section .data
5  ;
6  section .text
7  global _start
8  ;
9  _start:
10     mov eax, 1
11     mov ebx, 7
12     int 0x80
```

Per compilare il file si genera prima un file oggetto, quindi si passa per il *linker* (il «collegatore»), ovvero il programma che collega i file oggetto che devono comporre il file eseguibile finale. Si comincia con il sorgente per GNU AS:

```
$ as -o fine-gas.o fine-gas.s [Invio]

$ ld -o fine-gas fine-gas.o [Invio]
```

Per quanto riguarda il sorgente per NASM:

```
$ nasm -f elf -o fine-nasm.o fine-nasm.s [Invio]

$ ld -o fine-nasm fine-nasm.o [Invio]
```

In entrambi gli esempi, viene generato un file oggetto in formato ELF (*Executable and linkable format*), cosa che deve essere richiesta esplicitamente nel secondo caso, mentre nel primo è implicita. Pertanto, come si vede, il programma '**`ld`**' viene usato sempre nello stesso modo.

Il programma generato ('**`fine-gas`**' o '**`fine-nasm`**') si limita a chiamare una funzione del sistema operativo, con la quale conclude il suo lavoro restituendo il valore numerico sette. Lo si può verificare ispezionando il parametro '**`$?`**' della shell:

```
$ ./fine-gas [Invio]

$ echo $? [Invio]

7
```

I due programmi sono perfettamente uguali nel modo di disporsi nelle righe del file sorgente, pertanto vengono descritti senza fare distinzioni.

Le prime tre righe sono commenti, ignorati dal compilatore; la quarta riga contiene una direttiva per il compilatore che lo avverte dell'inizio della zona usata per descrivere l'uso della memoria (che in questo caso non viene usata affatto); la quinta riga è un altro commento; la sesta riga avverte il compilatore dell'inizio del codice del programma.

La settima riga serve ad avvisare il compilatore che il simbolo rappresentato dall'etichetta denominata '**`_start`**' individua l'indirizzo dell'istruzione iniziale da rendere pubblica, pertanto deve rimanere rintracciabile nel file oggetto generato dalla compilazione. L'ottava riga è un altro commento e la nona riga dichiara il simbolo '**`_start`**' già nominato.

Per il compilatore, l'etichetta '**_start**' indica convenzionalmente il punto di inizio del programma e il simbolo corrispondente deve essere reso pubblico, perché '**ld**' deve sapere da che parte si comincia (soprattutto quando più file oggetto si collegano assieme in un solo eseguibile).

Nella decima riga si assegna il valore uno al registro ***EAX*** e nell'undicesima si assegna il valore sette al registro ***EBX***; infine, nell'ultima riga, si esegue un'interruzione all'indirizzo 80_{16}. In pratica, le ultime tre righe servono a eseguire la chiamata di una funzione del sistema operativo. La funzione è individuata dal numero uno che deve essere collocato nel registro ***EAX*** e il parametro, rappresentato dal valore di uscita, da collocare nel registro ***EBX***. La chiamata effettiva avviene con l'interruzione all'indirizzo 80_{16}.

Dopo aver compilato il programma e ottenuto il file eseguibile, si può dare un'occhiata al suo contenuto con l'aiuto di Objdump:[3]

```
$ objdump --disassemble fine-gas [Invio]

fine-gas:     file format elf32-i386

Disassembly of section .text:

08048074 <_start>:
 8048074:       b8 01 00 00 00          mov    $0x1,%eax
 8048079:       bb 07 00 00 00          mov    $0x7,%ebx
 804807e:       cd 80                   int    $0x80
```

Usato in questo modo, Objdump disassembla il programma e mostra anche le istruzioni nel linguaggio macchina vero e proprio, con gli indirizzi di memoria virtuale che verrebbero utilizzati durante il funzionamento. Eventualmente si può richiedere espressamente di disassemblare utilizzando una notazione Intel:

```
$ objdump --disassemble -M intel fine-gas [Invio]

fine-gas:     file format elf32-i386

Disassembly of section .text:

08048074 <_start>:
 8048074:       b8 01 00 00 00          mov    eax,0x1
 8048079:       bb 07 00 00 00          mov    ebx,0x7
 804807e:       cd 80                   int    0x80
```

64.4.2 Utilizzo di GDB

Per poter iniziare lo studio di un linguaggio assemblatore è praticamente indispensabile saper utilizzare un *debugger*, ovvero uno strumento che permetta di eseguire passo per passo il proprio programma, consentendo di verificare lo stato dei registri ed eventualmente della memoria. Infatti, con un linguaggio assemblatore, operazioni «semplici» come l'emissione di informazioni attraverso lo schermo diventano invece molto complicate.

Nei sistemi GNU è disponibile GDB (GNU debugger)[4]. Per capire come utilizzarlo, si prenda nuovamente l'esempio di programma introduttivo, aggiungendo qualche piccola modifica:

```
1  #
2  # fine-gas.s
3  #
4  .section .data
5  #
6  .section .text
7  .globl _start
8  #
9  _start:
10     mov  $1, %eax
11 bp1:
12     mov  $7, %ebx
13 bp2:
14     int  $0x80
```



```
1  ;
2  ; fine-nasm.s
3  ;
4  segment .data
5  ;
6  segment .text
7  global _start
8  ;
9  _start:
10     mov eax, 1
11 bp1:
12     mov ebx, 7
13 bp2:
14     int 0x80
```

Rispetto al file originale sono state aggiunge due etichette, '**bp1**' e '**bp2**' (il cui nome è stato scelto arbitrariamente, ma in questo caso ricorda il termine *breakpoint*), collocate tra le istruzioni che si traducono in codici del linguaggio macchina. Naturalmente, il file va ricompilato nel modo già descritto; poi, una volta ottenuto il file eseguibile, lo si avvia all'interno di GDB:

```
$ gdb fine-gas [Invio]

GNU gdb 6.5-debian
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute
copies of it under certain conditions. Type "show copying"
to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details. This GDB was configured as
"i486-linux-gnu"...(no debugging symbols found)
Using host libthread_db library
"/lib/tls/libthread_db.so.1".
```

A questo punto GDB presenta un invito, dal quale è possibile inserire dei comandi in modo interattivo.

```
(gdb)
```

Di solito, la prima cosa da fare consiste nel definire degli «stop» (*breakpoint*), dove il programma deve essere fermato automaticamente. È per questa ragione che sono state aggiunte delle etichette nel sorgente: per poter associare a quei simboli dei punti di sospensione dell'esecuzione.

```
(gdb) break bp1 [Invio]

Breakpoint 1 at 0x8048079

(gdb) break bp2 [Invio]

Breakpoint 2 at 0x804807e
```

Una volta fissati gli stop, si può «avviare» il programma, che viene così sospeso in corrispondenza del primo di questi punti:

```
(gdb) run [Invio]

Starting program: /home/tizio/fine-gas
(no debugging symbols found)

Breakpoint 1, 0x08048079 in bp1 ()
```

Si ispezionano i registri:

```
(gdb) info registers [Invio]
```

```
eax            0x1      1
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbff50080       0xbff50080
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048079        0x8048079 <bp1>
eflags         0x292    [ AF SF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

Si può verificare che il registro ***EAX*** contiene il valore uno, come dovrebbe effettivamente in questa posizione. Per far proseguire il programma fino al prossimo stop si usa il comando '**`continue`**':

```
(gdb) continue [Invio]

 Breakpoint 2, 0x0804807e in bp2 ()
```

Si ispezionano nuovamente i registri:

```
(gdb) info registers [Invio]

 eax            0x1      1
 ecx            0x0      0
 edx            0x0      0
 ebx            0x7      7
 esp            0xbfcefe20       0xbfcefe20
 ebp            0x0      0x0
 esi            0x0      0
 edi            0x0      0
 eip            0x804807e        0x804807e <bp2>
 eflags         0x292    [ AF SF IF ]
 cs             0x73     115
 ss             0x7b     123
 ds             0x7b     123
 es             0x7b     123
 fs             0x0      0
 gs             0x0      0
```

Si può vedere che a questo punto il registro ***EBX*** risulta impostato con il valore previsto. Si lascia concludere il programma e si termina l'attività con GDB:

```
(gdb) continue [Invio]

 Continuing.

 Program exited with code 07.

(gdb) quit [Invio]
```

Il procedimento descritto vale per il programma compilato nel modo «normale», sia attraverso GNU AS, sia attraverso NASM. Ma per avere una visione più chiara di ciò che si fa, occorre abbinare il sorgente originale. Questo può essere fatto con GNU AS, utilizzando l'opzione '**`--gstabs`**', oppure con NASM mettendo l'opzione '**`-g`**'. Qui si mostra solo il caso di GNU AS:

```
$ as --gstabs -o fine-gas.o fine-gas.s [Invio]

$ ld -o fine-gas fine-gas.o [Invio]

$ gdb fine-gas [Invio]

 GNU gdb 6.5-debian
 Copyright (C) 2006 Free Software Foundation, Inc.
 GDB is free software, covered by the GNU General Public
 License, and you are welcome to change it and/or distribute
 copies of it under certain conditions. Type "show copying"
 to see the conditions.
 There is absolutely no warranty for GDB.  Type "show
 warranty" for details. This GDB was configured as
 "i486-linux-gnu"...(no debugging symbols found)
 Using host libthread_db library
 "/lib/tls/libthread_db.so.1".

(gdb) break bp1 [Invio]

 Breakpoint 1 at 0x8048079: file fine-gas.s, line 12.

(gdb) break bp2 [Invio]

 Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.

(gdb) run [Invio]

 Starting program: /home/tizio/fine-gas

 Breakpoint 1, bp1 () at fine-gas.s:12
 12          mov  $7, %ebx
 Current language:  auto; currently asm
```


Come si vede dall'esempio, si ottengono più informazioni collegate al sorgente originale; in particolare si può sapere qual è la prossima istruzione che verrebbe eseguita. Questa volta si decide di procedere eseguendo un'istruzione alla volta, con il comando '**`stepi`**':

```
(gdb) stepi [Invio]

 bp2 () at fine-gas.s:14
 14          int  $0x80
```

Dato che il programma è molto breve, con la prossima istruzione si va a concluderne il funzionamento:

```
(gdb) stepi [Invio]

 Program exited with code 07.

(gdb) quit [Invio]
```

Tabella 64.34. Alcuni comandi interattivi per GDB.

Comando	Descrizione
`set args` ***argomento*`...`**	Definisce gli argomenti della riga di comando del programma.
`r` [***argomento***]`...` `run` [***argomento***]`...`	Avvia l'esecuzione del programma, eventualmente con gli argomenti indicati.
`b` ***funzione*** `break` ***funzione***	Definisce uno stop all'esecuzione del programma in corrispondenza del simbolo indicato. Questo simbolo viene definito precisamente «funzione» in quanto riguarda la zona che descrive le istruzioni del programma e non quelle dei dati.
`b` ***riga*** `break` ***riga***	Definisce uno stop all'esecuzione del programma in corrispondenza della riga indicata, riferita al file sorgente. Per poter usare questo comando occorre compilare il programma con i riferimenti al file sorgente.
`cl` [***funzione***] `clear` [***funzione***]	Elimina uno stop associato a un certo simbolo, oppure, se c'è, elimina quello della posizione corrente.
`cl` ***riga*** `clear` ***riga***	Elimina uno stop associato alla riga indicata.
`d` [***n***]`...` `disable` [***n***]`...`	Disabilita gli stop indicati per numero, o tutti se non ne viene indicato alcuno.
`enable` [***n***]`...`	Riabilita gli stop indicati per numero, o tutti se non ne viene indicato alcuno.
`d` [***n***]`...` `delete` [***n***]`...`	Elimina gli stop indicati per numero, o tutti se non ne viene indicato alcuno.
`si` `stepi`	Esegue la prossima istruzione-macchina e poi si ferma nuovamente. Se l'istruzione è una chiamata di funzione, passa all'inizio della stessa.

Comando	Descrizione
`ni` `nexti`	Esegue la prossima istruzione-macchina e poi si ferma nuovamente. **Se l'istruzione è una chiamata di funzione, questa viene saltata.**
`c` `continue`	Riprende l'esecuzione di un programma dopo uno stop (che può poi fermarsi nuovamente allo stop successivo, se c'è).
`kill`	Interrompe definitivamente l'esecuzione del programma.
`i r` `info registers`	Mostra lo stato dei registri.
`bt` `backtrace`	Mostra lo stato della pila, precisamente mostra gli elementi che compongono lo *stack frame*.
`p` [`/`*f*] [(*tipo*)]*variabile* `print` [`/`*f*] [(*tipo*)]*variabile*	Mostra il valore contenuto in memoria, in corrispondenza del simbolo specificato (***variabile***). Se non si tratta di una variabile scalare di dimensione «standard», occorre specificarne il formato, tra parentesi tonde, esattamente prima del nome; se si vuole visualizzare il valore contenuto nella variabile in modo diverso da quello predefinito, occorre aggiungere l'opzione '***/f***', dove la lettera ***f*** specifica il tipo di rappresentazione.
`p /f` (*tipo*[*n*])*variabile* `print /f` (*tipo*[*n*])*variabile*	Se nella definizione del formato si mette un valore numerico tra parentesi quadre, si intende visualizzare ***n*** valori di quel tipo a partire dalla posizione indicata dal nome della variabile. Ciò consente di visualizzare il contenuto di un array.

64.4.3 Modalità «TUI»

GDB, se è stato compilato per includere tale funzionalità, può essere usato con l'opzione '**`-tui`**', con la quale si ottiene una suddivisione dello schermo in finestre:

`$` **`gdb -tui fine-gas`** [*Invio*]

```
   +--fine-gas.s-----------------------------------------+
   |10          mov  $1, %eax                           |
   |11      bp1:                                        |
   |12          mov  $7, %ebx                           |
   |13      bp2:                                        |
   |14          int  $0x80                              |
   |15                                                  |
   |16                                                  |
   |17                                                  |
   |18                                                  |
   |19                                                  |
   |20                                                  |
   |21                                                  |
   |22                                                  |
   |23                                                  |
   +----------------------------------------------------+
exec No process In:                        Line: ??   PC: 0x0
GDB is free software, covered by the GNU General Public
License, and you are welcome to change it and/or distribute
copies of it under certain conditions. Type "show copying"
to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details. This GDB was configured as
"i486-linux-gnu"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".
```

(gdb) **`break bp1`** [*Invio*]

(gdb) **`break bp2`** [*Invio*]


```
   +--fine-gas.s-----------------------------------------+
   |10          mov  $1, %eax                           |
   |11      bp1:                                        |
b+ |12          mov  $7, %ebx                           |
   |13      bp2:                                        |
b+ |14          int  $0x80                              |
   |15                                                  |
   |16                                                  |
   |17                                                  |
   |18                                                  |
   |19                                                  |
   |20                                                  |
   |21                                                  |
   |22                                                  |
   |23                                                  |
   +----------------------------------------------------+
exec No process In:                        Line: ??   PC: 0x0
This GDB was configured as "i486-linux-gnu"...Using host
libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) break bp1
Breakpoint 1 at 0x8048079: file fine-gas.s, line 12.
(gdb) break bp2
Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.
```

(gdb) **`run`** [*Invio*]

```
   +--fine-gas.s-----------------------------------------+
   |10          mov  $1, %eax                           |
   |11      bp1:                                        |
B+>|12          mov  $7, %ebx                           |
   |13      bp2:                                        |
b+ |14          int  $0x80                              |
   |15                                                  |
   |16                                                  |
   |17                                                  |
   |18                                                  |
   |19                                                  |
   |20                                                  |
   |21                                                  |
   |22                                                  |
   |23                                                  |
   +----------------------------------------------------+
child process 9699 In: bp1          Line: 12   PC: 0x8048079
(gdb) break bp2
Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.
(gdb) run
Starting program: /home/tizio/fine-gas

Breakpoint 1, bp1 () at fine-gas.s:12
Current language:  auto; currently asm
```

Durante il funzionamento si può anche attivare una finestra con lo stato dei registri, attraverso il comando '**`layout reg`**':

(gdb) **`layout reg`** [*Invio*]

```
+--Register group: general---------------------------------+
|eax            0x1      1                                |
|ecx            0x0      0                                |
|edx            0x0      0                                |
|ebx            0x0      0                                |
|esp            0xbfa77b90       0xbfa77b90               |
|ebp            0x0      0x0                              |
+----------------------------------------------------------+
   |10          mov  $1, %eax                           |
   |11      bp1:                                        |
B+>|12          mov  $7, %ebx                           |
   |13      bp2:                                        |
b+ |14          int  $0x80                              |
   |15                                                  |
   |16                                                  |
   +----------------------------------------------------+
child process 9699 In: bp1          Line: 12   PC: 0x8048079
Breakpoint 2 at 0x804807e: file fine-gas.s, line 14.
(gdb) run
Starting program: /home/tizio/fine-gas

Breakpoint 1, bp1 () at fine-gas.s:12
Current language:  auto; currently asm
(gdb) layout reg
```

(gdb) **`quit`** [*Invio*]

Durante il funzionamento normale di GDB, se è prevista tale mo-

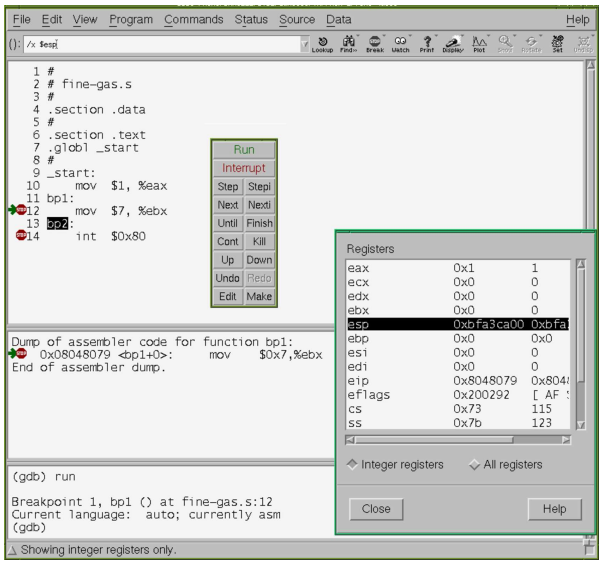
dalità di funzionamento, questa può essere attivata anche con la combinazione di tasti [*Ctrl x*][*a*].

64.4.4 Utilizzo di DDD

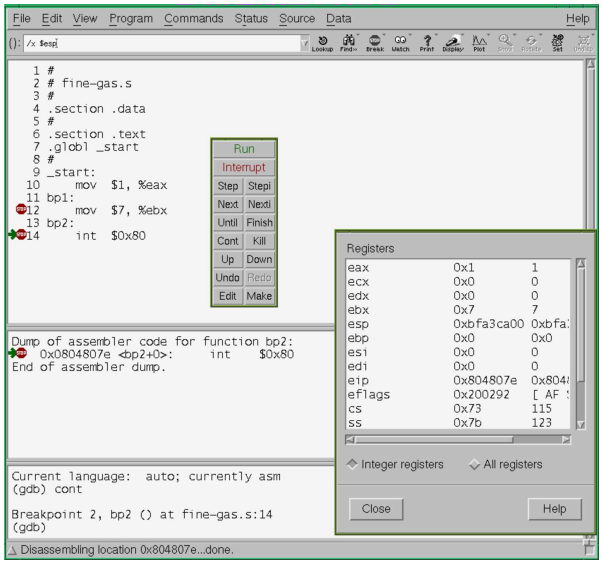
GDB è lo strumento fondamentale per il controllo del funzionamento di un programma, ma al suo fianco se ne possono aggiungere altri che consentono di avere una visione più «semplice». Per esempio, DDD,[5] ovvero *Data display debugger* è un programma frontale che si avvale di GDB, ma lo fa attraverso un'interfaccia grafica che consente di tenere sotto controllo più cose, simultaneamente.

```
$ ddd fine-gas [ Invio ]
```

Prendendo lo stesso esempio di programma già usato nella sezione precedente, compilato in modo da avere i riferimenti al sorgente, ecco come si può presentare DDD dopo che sono stati fissati gli stop e dopo che il programma è stato avviato, in modo che si arresti al primo di questi:



Ecco la situazione al livello del secondo stop:



64.4.5 Alcune istruzioni comuni

Le istruzioni del linguaggio assemblatore che si traducono direttamente in istruzioni del linguaggio macchina hanno una forma uniforme: un nome mnemonico seguito dagli operandi.

mnemonico operando [**,** ***operando***]...

Per esempio, l'istruzione seguente sottrae il valore contenuto nel registro ***EAX*** da quello di ***EBX***, mettendo il risultato nel registro ***EBX*** (***EBX***=***EBX***-***EAX***):

GNU AS
```
sub %eax, %ebx
```

NASM
```
sub ebx, eax
```

In questo caso il nome mnemonico è '**SUB**', mentre i nomi dei registri sono gli operandi (eventualmente, il codice corrispondente in linguaggio macchina sarebbe $29C3_{16}$). Nelle sezioni successive si descrivono alcune istruzioni comuni.

64.4.5.1 MOV

L'istruzione '**MOV**' serve a copiare il contenuto dell'origine nella destinazione. Gli operandi possono essere registri, aree di memoria e costanti numeriche, tenendo conto che le costanti numeriche possono figurare solo nell'origine e che si può fare riferimento ad aree di memoria in una sola posizione (nell'origine o nella destinazione).

GNU AS
```
mov origine, destinazione
```

NASM
```
mov destinazione, origine
```

L'esempio seguente copia il contenuto del registro ***EAX*** all'interno di ***EBX***:

GNU AS
```
mov %eax, %ebx
```

NASM
```
mov ebx, eax
```

64.4.5.2 ADD

L'istruzione '**ADD**' serve a eseguire la somma di valori interi, con o senza segno. Gli operandi possono essere registri, aree di memoria e costanti numeriche, tenendo conto che le costanti numeriche possono figurare solo nell'origine e che si può fare riferimento ad aree di memoria in una sola posizione: nell'origine o nella destinazione.

GNU AS
```
add origine, destinazione
```

NASM
```
add destinazione, origine
```

L'esempio seguente aggiunge al registro ***EAX*** una unità:

GNU AS
```
add $1, %eax
```

NASM
```
add eax, 1
```

64.4.5.3 SUB

L'istruzione '**SUB**' serve a eseguire la sottrazione di valori interi, con o senza segno. Gli operandi possono essere registri, aree di memoria e costanti numeriche, tenendo conto che le costanti numeriche possono figurare solo nell'origine e che si può fare riferimento ad aree di memoria in una sola posizione: nell'origine o nella destinazione.

GNU AS
```
sub origine, destinazione
```

NASM
```
sub destinazione, origine
```

L'esempio seguente sottrae il valore del registro ***EAX*** da quello di ***EBX***, mettendo il risultato in ***EBX***:

GNU AS
```
sub %eax, %ebx
```

NASM
```
sub ebx, eax
```

64.4.5.4 INC e DEC

Le istruzioni '**INC**' e '**DEC**' servono, rispettivamente, a incrementare e a decrementare di una unità il valore dell'operando, che può essere un registro o un'area di memoria. Dal momento che c'è un solo operando, non c'è differenza tra la sintassi di GNU AS e di NASM per quanto riguarda l'ordine degli stessi:

```
inc destinazione
```

dec ***destinazione***

I due esempi seguenti, rispettivamente, incrementano e decrementano di una unità il valore del registro ***EAX***:

- GNU AS `inc %eax`

  NASM `inc eax`

- GNU AS `dec %eax`

  NASM `dec eax`

64.4.6 Dimensione dei dati nelle istruzioni

Di norma, nelle istruzioni che elaborano dati, come quelle descritte nelle sezioni precedenti, se un registro si trova a essere origine o destinazione di qualcosa, implicitamente si intende che i dati debbano essere della sua dimensione. Per esempio, scrivendo '**`mov $1, %eax`**' si intende che la costante numerica sia precisamente $0000\ 0001_{16}$ (ovvero $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2$), perché la destinazione è da 32 bit.

Quando il contesto non è sufficiente a stabilire di quanti bit deve essere fatto il valore che si elabora, l'istruzione va precisata. Nel caso di GNU AS si aggiunge un suffisso al nome mnemonico che distingue l'operazione, suffisso che è composto da una sola lettera. Per esempio, '**`mov`**' diventa '**`movl`**' per chiarire che si tratta di dati da 32 bit. Nel caso di NASM si aggiunge una parola chiave dopo il nome mnemonico.

Tabella 64.41. Suffissi per precisare la quantità di bit coinvolti nelle istruzioni, secondo la sintassi di GNU AS. Negli esempi, '**`mem`**' è il nome di un simbolo (un'etichetta) che rappresenta un indirizzo di memoria.

Suffisso	Significato	Dimensione in bit	Esempi

`b`	byte	8 bit	`mov $1 %al`
			`movb $1 mem`
			`pushb mem`

`w`	*word*	16 bit	`mov $1 %ax`
			`movw $1 mem`
			`pushw mem`

`l`	*long*	32 bit	`mov $1 %eax`
			`movl $1 mem`
			`pushl mem`

Tabella 64.42. Parole chiave per precisare la quantità di bit coinvolti nelle istruzioni, secondo la sintassi di NASM. Negli esempi, '**`mem`**' è un simbolo che rappresenta un indirizzo di memoria.

Parola chiave	Dimensione in bit	Esempi

`byte`	8 bit	`mov al, 1`
		`mov byte [mem], 1`
		`push byte [mem]`

`word`	16 bit	`mov ax, 1`
		`mov word ax, 1`
		`mov word [mem], 1`
		`push word [mem]`

`long`	32 bit	`mov eax, 1`
		`mov long [mem], 1`
		`push long [mem]`

64.4.7 Direttive per il compilatore

Nel sorgente in linguaggio assemblatore è possibile inserire delle direttive (o pseudoistruzioni) che il compilatore può interpretare per la costruzione corretta del file oggetto. Si usano queste direttive in particolare per definire delle costanti e delle aree di memoria.

64.4.7.1 Commenti

Le righe vuote e quelle bianche vengono ignorate dal compilatore; inoltre, è possibile inserire dei commenti preceduti da un carattere che viene riconosciuto come prefisso, con il quale si annulla il significato di ciò che appare da quel punto fino alla fine della riga:

GNU AS ***#commento***

NASM ***;commento***

64.4.7.2 Direttiva «equ»

La direttiva '**equ**' viene usata per definire delle costanti simboliche (attraverso un nome, ovvero un'etichetta) da utilizzare poi nelle istruzioni, al posto del dato corrispondente. In linea di principio non dovrebbe essere possibile ridefinire le costanti.

GNU AS `.equ` ***nome, valore***

NASM ***nome*** `equ` ***valore***

Nell'esempio seguente si associa al simbolo '**`ADDRESS`**' il numero otto:

GNU AS `.equ ADDRESS, 8`

NASM `ADDRESS equ 8`

64.4.7.3 Direttive di dichiarazione dei dati

Attraverso delle direttive del compilatore si definiscono delle aree di memoria, a cui si fa riferimento nelle istruzioni attraverso dei nomi simbolici (etichette). In questo modo il compilatore attribuisce il loro indirizzo e lo sostituisce nella fase di assemblaggio. Si distingue tra dati che devono essere inizializzati preventivamente con un certo valore e dati il cui valore iniziale è indifferente. Quando si tratta di dati privi di valore iniziale, le informazioni necessarie consistono solo nel nome e nella dimensione dell'area di memoria:

GNU AS `.lcomm` ***nome, dimensione_in_byte***

NASM ***nome*** `resb` ***dimensione_in_byte***

NASM ***nome*** `resw` ***dimensione_in_multipli_di_16_bit***

NASM ***nome*** `resd` ***dimensione_in_multipli_di_32_bit***

Segue la descrizione di alcuni esempi.

- GNU AS `.lcomm BUFFER, 500`

  NASM `BUFFER resb 500`

  Dichiara localmente un'area di memoria da 500 byte, nominata provvisoriamente '**`BUFFER`**'.

- GNU AS `.lcomm NUMERO, 4`

  NASM `NUMERO resd 1`

  Dichiara localmente un'area di memoria da 4 byte (32 bit), nominata provvisoriamente '**`NUMERO`**'.

Quando si tratta di dati da inizializzare, le informazioni necessarie alla dichiarazione consistono nel nome e nel contenuto, mentre la dimensione deriva dalla pseudoistruzione scelta:

GNU AS ***nome*** `: .ascii` ***stringa*** [`,` ***stringa***]...

GNU AS ***nome*** `: .byte` ***byte*** [`,` ***byte***]...

GNU AS ***nome*** `: .int` ***intero*** [`,` ***intero***]...

NASM ***nome*** `db` ***byte*** [`,` ***byte***]

NASM ***nome*** `dd` ***intero*** [`,` ***intero***]

Segue la descrizione di alcuni esempi.

- GNU AS `X1: .byte 65`

  NASM `X1 db 65`

Inizializza l'area di memoria identificata temporaneamente con il nome '**X1**', con il valore 65_{10}.

- GNU/AS `X2: .byte 'A'`

  NASM `X2 db 'A'`

  Inizializza l'area di memoria identificata temporaneamente con il nome '**X2**', inserendo il codice corrispondente alla lettera «A».

- GNU/AS `X3: .int 12345`

  NASM `X3 dd 12345`

  Inizializza l'area di memoria identificata temporaneamente con il nome '**X3**', inserendo il valore 12345.

- GNU/AS `X4: .ascii 'Ciao!', 0`

  NASM `X4 db 'Ciao', 0`

  Inizializza l'area di memoria a partire dall'indirizzo identificato temporaneamente con il nome '**X3**', inserendo una stringa, terminata con il valore zero.

64.4.8 Sezioni del sorgente

Il sorgente si suddivide in sezioni, le quali descrivono la struttura del file-oggetto che si deve andare a produrre. Per quanto riguarda il formato ELF di un sistema GNU/Linux, si distingue l'area del codice, da quella dei dati; inoltre, nell'ambito dei dati si distingue la parte di quelli preinizializzati e di quelli che non lo sono.[6]

```
.section .data
    dichiarazione_dati_inizializzati
.section .bss
    dichiarazione_dati_non_definiti
.section .text
    istruzioni_del_programma
```

```
section .data
    dichiarazione_dati_inizializzati
section .bss
    dichiarazione_dati_non_definiti
section .text
    istruzioni_del_programma
```

I due modelli sintattici si riferiscono rispettivamente a GNU AS e a NASM.

64.4.9 Usare GNU AS con la notazione Intel

È possibile chiedere al compilatore GNU AS di interpretare il sorgente secondo la sintassi «Intel». Per questo si usa la direttiva '**.intel_syntax noprefix**' (l'opzione '**noprefix**' serve a consentire di annotare i nomi dei registri senza il prefisso '**%**'). L'esempio visto nella sezione 64.4.1 potrebbe essere modificato nel modo seguente:

```
#
.section .data
#
.section .text
.globl _start
#
.intel_syntax noprefix
_start:
    mov  eax, 1
    mov  ebx, 7
    int  0x80
```

La direttiva fa sì che il sorgente venga interpretato secondo la sintassi Intel a partire dal punto in cui si trova. Dal momento che l'effetto riguarda solo l'interpretazione delle istruzioni di codice che si traduce in linguaggio macchina (pertanto il modo di dare le altre direttive continua a essere quello normale di GNU AS), è conveniente piazzare la direttiva '**.intel_syntax noprefix**' all'inizio della sezione '**.text**', come si vede nell'esempio.

Naturalmente è possibile tornare alla sintassi AT&T con una direttiva analoga: '**.att_syntax**'. Lo si può vedere nell'esempio successivo:

```
#
.section .data
#
.section .text
.globl _start
#
.intel_syntax noprefix
_start:
    mov  eax, 1
.att_syntax
    mov  $7, %ebx
    int  $0x80
```

64.5 Esempi con le «quattro operazioni»

Vengono mostrati esempi di programmi estremamente banali, per dimostrare il funzionamento delle istruzioni con cui si eseguono le «quattro operazioni» su valori interi, attraverso l'aiuto di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**--gstabs**', mentre con NASM è bene aggiungere l'opzione '**-g**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

```
$ as --gstabs -o nome.o nome.s [Invio]
```

```
$ nasm -g -f elf -o nome.o nome.s [Invio]
```

Tabella 64.5. Operazioni aritmetiche.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
NEG	***reg*** ***mem***	Inverte il segno di un numero, attraverso il complemento a due. ***dst*** := −***dst***	c p z s o # # # # #
ADD	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Somma di interi, con o senza segno, ignorando il riporto precedente. Se i valori si intendono con segno, è importante l'esito dell'indicatore di traboccamento (*overflow*), se invece i valori sono da intendersi senza segno, è importante l'esito dell'indicatore di riporto (*carry*). ***dst*** := ***org*** + ***dst***	c p z s o # # # # #
SUB	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Sottrazione di interi con o senza segno, ignorando il riporto precedente. ***dst*** := ***org*** − ***dst***	c p z s o # # # # #
ADC	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Somma di interi, con o senza segno, aggiungendo anche il riporto precedente (l'indicatore *carry*). ***dst*** := ***org*** + ***dst*** + ***c***	c p z s o t . . . . # # # # #
SBB	***reg***, ***reg*** ***reg***, ***mem*** ***reg***, ***imm*** ***mem***, ***reg*** ***mem***, ***imm***	Sottrazione di interi, con o senza segno, tenendo conto del «prestito» precedente (l'indicatore *carry*). ***dst*** := ***org*** + ***dst*** − ***c***	c p z s o t . . . . # # # # #
INC	***reg*** ***mem***	Incrementa di una unità un intero. ***dst***++	c p z s o . # # # #
DEC	***reg*** ***mem***	Decrementa di una unità un valore intero. ***dst***−−	c p z s o . # # # #

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
`MUL`	***reg*** ***mem***	Moltiplicazione intera senza segno. L'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. ***AX*** := ***AL*src*** ***DX:AX*** := ***AX*src*** ***EDX:EAX*** := ***EAX*src***	`c p z s o` `# ? ? ? #`
`DIV`	***reg*** ***mem***	Divisione intera senza segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. ***AL*** := ***AX***/***src*** ***AH*** := ***resto*** ***AX*** := ***DX:AX***/***src*** ***DX*** := ***resto*** ***EAX*** := ***EDX:EAX***/***src*** ***EDX*** := ***resto***	`c p z s o` `? ? ? ? ?`
`IMUL`	***reg*** ***mem***	Moltiplicazione intera con segno. In questo caso l'operando è il moltiplicatore, mentre il moltiplicando è costituito da registri prestabiliti. ***AX*** := ***AL*src*** ***DX:AX*** := ***AX*src*** ***EDX:EAX*** := ***EAX*src***	`c p z s o` `# ? ? ? #`
`IDIV`	***reg*** ***mem***	Divisione intera con segno. L'operando è il divisore, mentre il dividendo è costituito da registri prestabiliti. ***AL*** := ***AX***/***src*** ***AH*** := ***resto*** ***AX*** := ***DX:AX***/***src*** ***DX*** := ***resto*** ***EAX*** := ***EDX:EAX***/***src*** ***EDX*** := ***resto***	`c p z s o` `? ? ? ? ?`

64.5.1 Somma

«

Viene proposto un programma che si limita a sommare due numeri (interi positivi) definiti in memoria e a restituire il risultato (ammesso che non sia troppo grande) attraverso il valore di uscita. Il programma viene mostrato sia nella forma adatta a GNU AS, sia in quella conforme a NASM. Le righe dei due listati coincidono.

```
1  # op1 + op2
2  #
3  .section .data
4  op1:    .int    15
5  op2:    .int     5
6  #
7  .section .text
8  .globl _start
9  #
10 _start:
11     mov op1, %edx   # Accumula il primo addendo in EDX.
12     add op2, %edx   # Somma il secondo addendo in EDX.
13 bp1:
14     mov $1,   %eax  # Restituisce il valore contenuto
15     mov %edx, %ebx  # in EDX come valore di uscita,
16     int $0x80       # attraverso la chiamata di sistema
17                     # 1 (exit).
```

```
1  ; op1 + op2
2  ;
3  section .data
4  op1:    dd 15
5  op2:    dd  5
6  ;
7  section .text
8  global _start
9  ;
10 _start:
11     mov edx, [op1]  ; Accumula il primo addendo in EDX.
12     add edx, [op2]  ; Somma il secondo addendo in EDX.
13 bp1:
14     mov eax, 1      ; Restituisce il valore contenuto
15     mov ebx, edx    ; in EDX come valore di uscita,
16     int 80h         ; attraverso la chiamata di sistema
17                     ; 1 (exit).
```


Nelle righe 4 e 5 vengono dichiarate due aree di memoria della dimensione di un registro (32 bit), associando rispettivamente i nomi '**`op1`**' e '**`op2`**', a indicare il primo e il secondo operando della somma. Nella riga 11 il contenuto della memoria che rappresenta il primo operando della somma, viene inserito nel registro ***EDX***, mentre nella riga 12 si somma quanto è rappresentato dal secondo operando, nello stesso registro.

Si osservi che per indicare l'indirizzo di memoria è stata usata la modalità di indirizzamento diretta. In pratica, il compilatore sostituisce i nomi '**`op1`**' e '**`op2`**' con l'indirizzo di memoria a cui fanno riferimento.

Al termine, nelle righe da 14 a 16, si prepara la chiamata di sistema '**`exit`**', passando il risultato in modo che venga usato come valore di uscita. Se il risultato della somma è inferiore o uguale a 255, può essere letto.

I programmi sono uguali, a parte qualche piccola differenza nell'allocazione della memoria. Si può controllare con Objdump. Si suppone che il programma sia stato compilato con il nome '`add`':

```
$ objdump --disassemble add [Invio]

 add:    file format elf32-i386

 Disassembly of section .text:

 08048074 <_start>:
  8048074:       8b 15 8c 90 04 08       mov   0x804908c,%edx
  804807a:       03 15 90 90 04 08       add   0x8049090,%edx

 08048080 <bp1>:
  8048080:       b8 01 00 00 00          mov   $0x1,%eax
  8048085:       89 d3                   mov   %edx,%ebx
  8048087:       cd 80                   int   $0x80
```

Si può controllare il funzionamento del programma, avviandolo e verificando poi il valore di uscita:

```
$ ./add ; echo $? [Invio]

 20
```

Si analizza il funzionamento del programma con GDB:

```
$ gdb add [Invio]

(gdb) break bp1 [Invio]

 Breakpoint 1 at 0x8048080: file add.s, line 14.

(gdb) run [Invio]

 Starting program: /home/tizio/add

 Breakpoint 1, bp1 () at add.s:14
 14          mov $1,   %eax  # Restituisce il valore contenuto in EDX
 Current language:  auto; currently asm


(gdb) info registers [Invio]

 eax            0x0      0
 ecx            0x0      0
 edx            0x14     20
 ebx            0x0      0
 esp            0xbf9dcb10       0xbf9dcb10
 ebp            0x0      0x0
 esi            0x0      0
 edi            0x0      0
 eip            0x8048080        0x8048080 <bp1>
```

```
eflags         0x216    [ PF AF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

```
(gdb) stepi [Invio]

bp1 () at add.s:15
15          mov %edx, %ebx  # come valore di uscita, attraverso
(gdb) stepi [Invio]

bp1 () at add.s:16
16          int $0x80  # chiamata di sistema 1 (exit).
(gdb) info registers [Invio]

eax            0x1      1
ecx            0x0      0
edx            0x14     20
ebx            0x14     20
esp            0xbfba64d0       0xbfba64d0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048087        0x8048087 <bp1+7>
eflags         0x216    [ PF AF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

```
(gdb) stepi [Invio][7]

Program exited with code 024.
(gdb) quit [Invio]
```

64.5.1.1 Traboccamento

«

Si può modificare leggermente il programma proposto, allo scopo di causare un traboccamento, in modo da vedere cosa accade nei registri:

```
3  .section .data
4  op1: .int 0x7FFFFFFF # 0b01111111111111111111111111111111
5  op2: .int 0x00000001 # 0b00000000000000000000000000000001
```

```
3  section .data
4  op1: dd 0x7FFFFFFF ; 01111111111111111111111111111111b
5  op2: dd 0x00000001 ; 00000000000000000000000000000001b
```

Compilando il programma ed eseguendolo con l'ausilio di GDB si può verificare che la somma di quei due valori trasforma il risultato in un valore apparentemente negativo, cosa che è indice di un traboccamento:

```
eax            0x0      0
ecx            0x0      0
edx            0x80000000       -2147483648
ebx            0x0      0
esp            0xbfa6f390       0xbfa6f390
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048080        0x8048080 <bp1>
eflags         0xa96    [ PF AF SF IF OF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

A ogni modo il fatto viene sottolineato dall'indicatore di traboccamento (*overflow*). In questo caso non interviene l'indicatore di riporto (*carry*), perché se il risultato fosse da intendersi senza segno, sarebbe ancora corretto.

64.5.1.2 Riporto

«

Si può modificare leggermente il programma proposto, allo scopo di causare un riporto, in modo da vedere cosa accade nei registri:

```
3  .section .data
4  op1: .int 0xFFFFFFFE # 0b11111111111111111111111111111110
5  op2: .int 0x00000002 # 0b00000000000000000000000000000010
```

```
3  section .data
4  op1: dd 0xFFFFFFFE ; 11111111111111111111111111111110b
5  op2: dd 0x00000002 ; 00000000000000000000000000000001b
```

In questo caso, la somma dei due valori supera proprio di una unità la capienza del registro, che alla fine risulta a zero, ma l'indicatore di riporto (*carry*) segnala l'accaduto:

```
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x0      0
esp            0xbfbb44e0       0xbfbb44e0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048080        0x8048080 <bp1>
eflags         0x257    [ CF PF AF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

Si può osservare che è assente l'indicatore di traboccamento (*overflow*), perché se la somma fosse avvenuta tra due numeri con segno, il risultato sarebbe corretto.

64.5.1.3 Somma di valori molti grandi

«

Viene mostrato un altro esempio, dove la somma riguarda valori molto grandi, divisi tra due registri. I due listati sono equivalenti e compatibili, riga per riga:

```
1  # op1 + op2
2  #
3  .section .data
4  op1:    .quad   0x00FFFFFFFFFFFFFF
5  op2:    .quad   0x0000000000000001
6  #
7  .section .text
8  .globl _start
9  #
10 _start:
11     mov op1  , %eax # Accumula metà del primo addendo
12                     # in EAX.
13     mov op1+4, %edx # Accumula il resto del primo
14                     # addendo in EDX.
15     add op2,   %eax # Somma metà del secondo addendo
16                     # in EAX.
17 bp1:
18     adc op2+4, %edx # Somma il resto del secondo addendo
19                     # in EDX. Il risultato atteso in
20                     # EDX:EAX è: 0x01000000:0x00000000
21 bp2:
22     mov $1,    %eax # Conclude il funzionamento del
23     mov $0,    %ebx # programma restituendo zero in
24     int $0x80       # ogni caso.
```

```
1  ; op1 + op2
2  ;
3  section .data
4  op1:    dd  0xFFFFFFFF, 0x00FFFFFF ; 0x00FFFFFFFFFFFFFF
5  op2:    dd  0x00000001, 0x00000000 ; 0x0000000000000001
6  ;
7  section .text
8  global _start
9  ;
10 _start:
11     mov eax, [op1]   ; Accumula metà del primo addendo
```

```
12                      ; in EAX.
13      mov edx, [op1+4] ; Accumula il resto del primo
14                      ; addendo in EDX.
15      add eax, [op2]   ; Somma metà del secondo addendo
16                      ; in EAX.
17  bp1:
18      adc edx, [op2+4] ; Somma il resto del secondo
19                      ; addendo in EDX. Il risultato
20                      ; atteso in EDX:EAX è:
21                      ; 0x01000000:0x00000000
22  bp2:
23      mov eax, 1       ; Conclude il funzionamento
24      mov edx, 0       ; del programma restituendo
25      int 80h          ; zero in ogni caso.
```

In questo programma ci sono delle complicazioni che vanno descritte, cominciando preferibilmente dalla versione per GNU AS (il primo listato). Nelle righe 4 e 5 vengono dichiarati due numeri molto grandi, da 64 bit. Successivamente, nelle righe da 11 a 15, si fa riferimento a questi due numeri, prendendoli a pezzi. Per la precisione, nella riga 11 si copiano i primi 32 bit a partire dall'indirizzo a cui fa riferimento l'etichetta '**op1**' (sono solo 32 bit perché l'istruzione copia il valore in un registro di tale dimensione); nella riga 12 si copiano gli altri 32 bit, indicando che dall'indirizzo dell'etichetta '**op1**' occorre spostarsi in avanti di quattro byte. Lo stesso ragionamento si fa per il secondo operando.

L'ordine in cui sono prelevati i dati è importante e occorre riflettere su questo fatto. Il registro ***EAX*** viene caricato con la porzione meno significativa del numero, che in memoria si trova nei «primi» 32 bit. Ciò avviene perché si intende che il microprocessore operi ordinando i byte in memoria secondo la modalità *little endian*.

Nel secondo listato, quello per NASM, non essendo possibile indicare un numero completo da 64 bit, si è reso necessario spezzarlo in due. In questo caso, per mantenere la stessa struttura dell'altro listato, i due tronconi sono stati messi in fila, apparentemente in ordine inverso, per riprodurre la stessa sequenza *little endian* complessiva.

Una volta compreso il modo in cui i dati sono prelevati dalla memoria, ci si può soffermare sulle istruzioni di somma: il troncone meno significativo viene sommato con l'istruzione '**ADD**', mentre per quello più significativo si usa '**ADC**' che aggiunge anche il riporto, se c'è.

Alla fine, una volta compilato il programma, con GDB è possibile eseguirlo fino al simbolo evidenziato dall'etichetta '**bp1**' per verificare l'effetto della prima addizione, quindi lo si può fare proseguire fino al simbolo '**bp2**', per verificare che la coppia di registri ***EDX:EAX*** contenga il risultato corretto:

```
(gdb) break bp1 [Invio]

(gdb) break bp2 [Invio]

(gdb) run [Invio]

(gdb) info registers [Invio]

eax            0x0      0
ecx            0x0      0
edx            0xffffff 16777215
ebx            0x0      0
esp            0xbfa89bb0       0xbfa89bb0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x8048085        0x8048085 <bp1>
eflags         0x257    [ CF PF AF ZF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

La prima somma ha prodotto uno zero nel registro ***EAX***, con riporto.

```
(gdb) continue [Invio]
```



```
(gdb) info registers [Invio]

eax            0x0      0
ecx            0x0      0
edx            0x1000000        16777216
ebx            0x0      0
esp            0xbfa89bb0       0xbfa89bb0
ebp            0x0      0x0
esi            0x0      0
edi            0x0      0
eip            0x804808b        0x804808b <bp2>
eflags         0x216    [ PF AF IF ]
cs             0x73     115
ss             0x7b     123
ds             0x7b     123
es             0x7b     123
fs             0x0      0
gs             0x0      0
```

```
(gdb) kill [Invio]

(gdb) quit [Invio]
```

64.5.2 Sottrazione

Viene proposto un programma che esegue una sottrazione, emettendo il risultato attraverso il valore di uscita. Il programma viene mostrato sia nella forma adatta a GNU AS, sia in quella conforme a NASM. Le righe dei due listati coincidono.

```
1   # op1 - op2
2   #
3   .section .data
4   op1: .int 0x00000001 # 0b00000000000000000000000000000001
5   op2: .int 0x00000002 # 0b00000000000000000000000000000010
6   #
7   .section .text
8   .globl _start
9   #
10  _start:
11      mov op1, %edx  # Accumula il minuendo in EDX.
12      sub op2, %edx  # Riduce EDX del sottraendo.
13  bp:
14      mov $1,   %eax # Restituisce il valore contenuto in
15      mov %edx, %ebx # EDX come valore di uscita,
16      int $0x80      # attraverso la chiamata di sistema
17                     # 1 (exit).
```

```
1   ; op1 - op2
2   ;
3   section .data
4   op1: dd 0x00000001 ; 0000000000000000000000000000001b
5   op2: dd 0x00000002 ; 0000000000000000000000000000010b
6   ;
7   section .text
8   global _start
9   ;
10  _start:
11      mov edx, [op1] ; Accumula il minuendo in EDX.
12      sub edx, [op2] ; Riduce EDX del sottraendo.
13  bp1:
14      mov eax, 1     ; Restituisce il valore contenuto in EDX
15      mov ebx, edx   ; come valore di uscita, attraverso la
16      int 80h        ; chiamata di sistema 1 (exit).
```

Rispetto agli esempi che riguardano la somma, qui, nella riga 12 si utilizza l'istruzione '**SUB**'. Come si può comprendere, dal momento che si sottrae una grandezza maggiore di quella contenuta nel minuendo, si ottiene un valore negativo, oppure, se i valori sono da intendersi senza segno, si ottiene un «riporto», come richiesta del prestito di una cifra oltre quella più significativa. Con GDB, in corrispondenza del simbolo '**bp1**' si possono vedere i registri e si può verificare che è scattato il riporto (*carry*) e il segno:

```
…
edx            0xffffffff       -1
…
eflags         0x297    [ CF PF AF SF IF ]
…
```

Se si lascia concludere il programma, il valore di uscita che si ottiene è 255_{10}, pari a FF_{16}, ovvero 377_8, ovvero 11111111_2, dal momento

che si possono ottenere solo otto bit.

64.5.2.1 Inversione del segno

Si può vedere cosa accade se, invece di usare l'istruzione '**SUB**', si cambia il segno al sottraendo e lo si somma semplicemente all'altro operando:

```
1   # op1 - op2
2   #
3   .section .data
4   op1:    .int    0x00000001   # 0b00000000000000000000000000000001
5   op2:    .int    0x00000002   # 0b00000000000000000000000000000010
6   #
7   .section .text
8   .globl _start
9   #
10  _start:
11      mov op2, %edx  # Accumula il sottraendo in EDX.
12      neg %edx       # Ne inverte il segno con il complemento a due.
13  bp1:
14      add op1, %edx  # Somma il minuendo a EDX.
15  bp2:
16      mov $1,   %eax # Restituisce il valore contenuto in EDX
17      mov %edx, %ebx # come valore di uscita, attraverso la
18      int $0x80      # chiamata di sistema 1 (exit).
```

```
1   ; op1 - op2
2   ;
3   section .data
4   op1:    dd      0x00000001   ; 0b00000000000000000000000000000001
5   op2:    dd      0x00000002   ; 0b00000000000000000000000000000010
6   ;
7   section .text
8   global _start
9   ;
10  _start:
11      mov edx, [op2] ; Accumula il sottraendo in EDX.
12      neg edx        ; Ne inverte il segno con il complemento a due.
13  bp1:
14      add edx, [op1] ; Somma il minuendo a EDX.
15  bp2:
16      mov eax, 1     ; Restituisce il valore contenuto in EDX
17      mov ebx, edx   ; come valore di uscita, attraverso la
18      int 80h        ; chiamata di sistema 1 (exit).
```

Rispetto a quanto fatto nel caso precedente, qui, nella riga 11 viene sommato il valore del sottraendo nel registro ***EDX***, di cui viene invertito il segno nella riga 12. Successivamente, nella riga 14 viene sommato il valore del minuendo. Il risultato è lo stesso, ma gli indicatori si comportano in modo differente durante il procedimento. In corrispondenza del simbolo '**bp1**' si può vedere quanto segue:

```
…
edx             0xfffffffe       -2
…
eflags          0x293    [ CF AF SF IF ]
…
```

Si può osservare che l'inversione del segno ha prodotto un riporto, oltre alla segnalazione del segno, ammesso che questo vada considerato.

In corrispondenza del simbolo '**bp2**' è appena stata eseguita la somma del minuendo. Si può osservare che questa volta non si ottiene alcun riporto:

```
…
edx             0xffffffff       -1
…
eflags          0x293    [ PF SF IF ]
…
```

64.5.2.2 Sottrazione per fasi successive

Viene proposto un esempio di sottrazione da svolgere in due fasi, perché il valore non è contenibile in un solo registro. Si vuole eseguire: $1000000000000000_{16} - 0FFFFFFFFFFFFFFF_{16}$.

```
1   # op1 + op2
2   #
3   .section .data
4   op1:    .quad   0x1000000000000000
5   op2:    .quad   0x0FFFFFFFFFFFFFFF
6   #
7   .section .text
```



```
8   .globl _start
9   #
10  _start:
11      mov op1  , %eax # Accumula metà del primo valore
12                      # in EAX.
13      mov op1+4, %edx # Accumula il resto del primo
14                      # valore in EDX.
15  bp1:
16      sub op2,   %eax # Sottrae metà del secondo valore
17                      # in EAX.
18  bp2:
19      sbb op2+4, %edx # Sottrae il resto del secondo
20                      # valore in EDX.
21  bp3:
22      mov $1,    %eax # Conclude il funzionamento del
23      mov $0,    %ebx # programma restituendo zero
24      int $0x80       # in ogni caso.
```

```
1   ; op1 + op2
2   ;
3   section .data
4   op1:    dd  0x00000000, 0x10000000 ; 0x1000000000000000
5   op2:    dd  0xFFFFFFFF, 0x0FFFFFFF ; 0x0FFFFFFFFFFFFFFF
6   ;
7   section .text
8   global _start
9   ;
10  _start:
11      mov eax, [op1]   ; Accumula metà del primo valore
12                       ; in EAX.
13      mov edx, [op1+4] ; Accumula il resto del primo
14                       ; valore in EDX.
15  bp1:
16      sub eax, [op2]   ; Sottrae metà del secondo valore
17                       ; in EAX.
18  bp2:
19      sbb edx, [op2+4] ; Sottrae il resto del secondo
20                       ; valore in EDX.
21  bp3:
22      mov eax, 1       ; Conclude il funzionamento del
23      mov edx, 0       ; programma restituendo zero
24      int 0x80         ; in ogni caso.
```

Il valore del minuendo viene copiato, in due pezzi, nei registri ***EDX:EAX***; quindi si sottraggono i 32 bit inferiori del sottraendo al registro ***EAX*** e infine si sottraggono i 32 bit più significativi del sottraendo dal registro ***EDX***, tenendo conto del riporto precedente.

In corrispondenza del simbolo '**bp1**', il minuendo è stato copiato nei registri ***EDX:EAX***:

```
…
eax             0x0      0
…
edx             0x10000000       268435456
…
eflags          0x292    [ AF SF IF ]
…
```

Al punto di '**bp2**' è stata eseguita la sottrazione dei 32 bit inferiori, causando un riporto, da intendersi come richiesta di un prestito:

```
…
eax             0x1      1
…
edx             0x10000000       268435456
…
eflags          0x213    [ CF AF IF ]
…
```

Al punto di '**bp3**' è stata completata la sottrazione:

```
…
eax             0x1      1
…
edx             0x0      0
…
eflags          0x256    [ PF AF ZF IF ]
…
```

64.5.3 Moltiplicazione senza segno

Nella moltiplicazione si distingue il fatto che si consideri il segno o meno. Quando si esegue una moltiplicazione senza segno si usa l'istruzione '**MUL**' con l'indicazione di un solo operando, perché gli altri sono impliciti. Nella moltiplicazione il contenitore del risultato deve essere più capiente di ciò che è stato usato per produrlo. Si distinguono questi casi:

AX* := *AL*src

DX:AX* := *AX*src

EDX:EAX* := *EAX*src

In pratica, l'origine deve essere di pari dimensioni del moltiplicando, costituito, rispettivamente da: ***AL***, ***AX*** o ***EAX***.

```
1  # op1 * op2
2  #
3  .section .data
4  op1:    .short  0x8008
5  op2:    .short  0x2002
6  #
7  .section .bss
8  .lcomm prodotto, 4
9  #
10 .section .text
11 .globl _start
12 #
13 _start:
14     mov  op1, %ax       # Accumula il moltiplicando
15                         # in AX.
16     mulw op2            # Moltiplica il secondo valore
17                         # per AX (implicito).
18 bp1:
19     mov %ax, prodotto   # Copia in memoria la prima
20                         # parte del risultato.
21     mov %dx, prodotto+2 # Copia in memoria la seconda
22                         # parte del risultato.
23     mov prodotto, %eax  # Copia il risultato dalla
24                         # memoria a EAX.
25 bp2:
26     mov $1,   %eax      # Restituisce il valore
27                         # contenuto in EAX
28     mov %eax, %ebx      # come valore di uscita,
29                         # attraverso la
30     int $0x80           # chiamata di sistema 1 (exit).
```

```
1  ; op1 * op2
2  ;
3  section .data
4  op1:    dw      0x8008
5  op2:    dw      0x2002
6  ;
7  section .bss
8  prodotto resb 4
9  ;
10 section .text
11 global _start
12 ;
13 _start:
14     mov  ax, [op1]      ; Accumula il moltiplicando
15                         ; in AX.
16     mul word [op2]      ; Moltiplica il secondo
17                         ; valore per AX (implicito).
18 bp1:
19     mov  [prodotto], ax   ; Copia in memoria la prima
20                           ; parte del risultato.
21     mov  [prodotto+2], dx ; Copia in memoria la
22                           ; seconda parte del
23                           ; risultato.
24     mov  eax, [prodotto]  ; Copia il risultato
25                           ; dalla memoria a EAX.
26 bp2:
27     mov  eax, 1           ; Restituisce il valore
28                           ; contenuto in EAX
29     mov  ebx, eax         ; come valore di uscita,
30                           ; attraverso la
31     int  0x80             ; chiamata di sistema
32                           ; 1 (exit).
```


In questo esempio i valori da moltiplicare sono della dimensione di 16 bit e sono, rispettivamente, 8008_{16} e 2002_{16}. Moltiplicando questi due valori si deve ottenere 10020010_{16}. In pratica si deve eseguire una moltiplicazione del tipo ***DX:AX*** := ***AX*src***.

Rispetto a esempi già visti nelle sezioni precedenti, in questo si dichiara un'area di memoria non inizializzata, nella riga numero 8, per contenere almeno quattro byte (32 bit), con il nome simbolico '**prodotto**'. All'interno di questa area di memoria si vuole ricostruire il risultato della moltiplicazione in modo che occupi un gruppo continuo di 32 bit.

Nella riga 15 si esegue la moltiplicazione, utilizzando come operando direttamente la memoria. Tuttavia, per farlo, occorre specificare la dimensione di questo operando, altrimenti verrebbe presa in considerazione un'area più grande del voluto.

In corrispondenza del punto '**bp1**' si può vedere il risultato della moltiplicazione diviso tra ***DX*** e ***AX***:

```
…
eax            0x10     16
…
edx            0x1002   4098
…
eflags         0xa93    [ CF AF SF IF OF ]
…
```

Nelle righe 17 e 18 viene copiato il risultato in memoria, ricomponendolo nell'ordine corretto, osservando che si usa una rappresentazione dei valori numerici in modalità *little endian*, quindi la parte meno significativa viene copiata prima. In corrispondenza del punto '**bp2**' il risultato della moltiplicazione è tutto contenuto nel registro ***EAX***:

```
…
eax            0x10020010       268566544
…
edx            0x1002   4098
…
eflags         0xa93    [ CF AF SF IF OF ]
…
```

64.5.4 Moltiplicazione con segno

La moltiplicazione con segno si ottiene con un'istruzione differente, '**IMUL**', che però va usata con la stessa modalità di quella senza segno. Sarebbe possibile usare più di un operando con questa istruzione, senza bisogno di operandi impliciti, ma in generale non è conveniente perché c'è il rischio di creare confusione sulla dimensione di questi operandi.

```
1  # op1 * op2
2  #
3  .section .data
4  op1:    .short   0x0007
5  op2:    .short  -0x0001
6  #
7  .section .bss
8  .lcomm prodotto, 4
9  #
10 .section .text
11 .globl _start
12 #
13 _start:
14     mov   op1, %ax      # Accumula il moltiplicando
15                         # in AX.
16     imulw op2           # Moltiplica il secondo valore
17                         # per AX (implicito).
18 bp1:
19     mov %ax, prodotto   # Copia in memoria la prima
20                         # parte del risultato.
21     mov %dx, prodotto+2 # Copia in memoria la seconda
22                         # parte del risultato.
23     mov prodotto, %eax  # Copia il risultato dalla
24                         # memoria a EAX.
```

```
25  bp2:
26      mov $1,   %eax      # Restituisce il valore
27                          # contenuto in EAX
28      mov %eax, %ebx      # come valore di uscita,
29                          # attraverso la
30      int $0x80           # chiamata di sistema 1 (exit).
```

```
1   ; op1 * op2
2   ;
3   section .data
4   op1:    dw       0x0007
5   op2:    dw      -0x0001
6   ;
7   section .bss
8   prodotto resb 4
9   ;
10  section .text
11  global _start
12  ;
13  _start:
14      mov       ax, [op1]        ; Accumula il
15                                 ; moltiplicando in AX.
16      imul word [op2]            ; Moltiplica il secondo
17                                 ; valore per AX
18                                 ; (implicito).
19  bp1:
20      mov       [prodotto], ax   ; Copia in memoria la
21                                 ; prima parte del
22                                 ; risultato.
23      mov       [prodotto+2], dx ; Copia in memoria la
24                                 ; seconda parte del
25                                 ; risultato.
26      mov       eax, [prodotto]  ; Copia il risultato
27                                 ; dalla memoria a EAX.
28  bp2:
29      mov       eax, 1           ; Restituisce il valore
30                                 ; contenuto in EAX
31      mov       ebx, eax         ; come valore di uscita,
32                                 ; attraverso la
33      int       0x80             ; chiamata di sistema
34                                 ; 1 (exit).
```

Rispetto a esempi già visti, questo utilizza una costante numerica negativa (riga 5); in pratica è il compilatore che la trasforma nel complemento a due, in modo automatico. Per il resto, tutto procede come nell'esempio della moltiplicazione intera, a parte l'uso dell'istruzione '**IMUL**'.

In corrispondenza del punto '**bp1**' si può vedere il risultato della moltiplicazione, distribuito tra ***DX:AX***:

```
…
eax             0xfff9   65529
…
edx             0xffff   65535
…
eflags          0x292    [ AF SF IF ]
…
```

In '**bp2**' il risultato è completo nel registro ***EAX***:

```
…
eax             0xfffffff9       -7
…
eflags          0x292    [ AF SF IF ]
…
```

64.5.5 Divisione

Per la divisione si usa un meccanismo simile a quello della moltiplicazione, ma opposto:

AL := AX/src AH := resto

AX := DX:AX/src DX := resto

EAX := EDX:EAX/src EDX := resto

In questo esempio si parte da valori che occupano 32 bit, azzerando inizialmente ***EDX*** perché il dividendo non è così grande da richiederne l'utilizzo.

```
1   # op1 / op2
2   #
3   .section .data
4   op1:    .int    0x00010001
5   op2:    .int    0x00000002
6   #
7   .section .text
8   .globl _start
9   #
10  _start:
11      mov  op1, %eax # Accumula il dividendo in EAX.
12      mov  $0,  %edx # Azzera EDX.
13      divl op2       # Divide EDX:EAX per il divisore.
14  bp1:
15      mov $1,   %eax # Restituisce il valore contenuto
16      mov %edx, %ebx # in EDX come valore di uscita,
17      int $0x80      # attraverso la chiamata di sistema
18                     # 1 (exit).
```

```
1   ; op1 * op2
2   ;
3   section .data
4   op1:    dd      0x00010001
5   op2:    dd      0x00000002
6   ;
7   section .text
8   global _start
9   ;
10  _start:
11      mov  eax, [op1] ; Accumula il dividendo in EAX.
12      mov  edx, 0     ; Azzera EDX.
13      div long [op2]  ; Divide EDX:EAX per il divisore.
14  bp1:
15      mov  eax, 1     ; Restituisce il valore contenuto
16      mov  edx, eax   ; in EDX come valore di uscita,
17      int  0x80       ; attraverso la chiamata di sistema
18                      ; 1 (exit).
```

In corrispondenza del punto '**bp1**' si può leggere il risultato della divisione in ***EAX*** e il resto in ***EDX***:

```
…
eax             0x8000   32768
…
edx             0x1      1
…
eflags          0x212    [ AF IF ]
…
```

Per quanto riguarda la divisione con segno, tutto procede nello stesso modo, a parte il fatto che si utilizza l'istruzione '**IDIV**':

```
10  _start:
11      mov   op1, %eax # Accumula il dividendo in EAX.
12      mov   $0,  %edx # Azzera EDX.
13      idivl op2       # Divide EDX:EAX per il divisore.
```

```
10  _start:
11      mov       eax, [op1] ; Accumula il dividendo in EAX.
12      mov       edx, 0     ; Azzera EDX.
13      idiv long [op2]      ; Divide EDX:EAX per il divisore.
```

64.6 Esempi con gli «spostamenti»

Vengono mostrati esempi di programmi estremamente banali, per dimostrare il funzionamento delle istruzioni con cui si eseguono gli spostamenti di bit (scorrimenti e rotazioni), attraverso l'aiuto di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**--gstabs**', mentre con NASM è bene aggiungere l'opzione '**-g**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

```
$ as --gstabs -o nome.o nome.s [Invio]

$ nasm -g -f elf -o nome.o nome.s [Invio]
```

Tabella 64.7. Scorrimenti e rotazioni.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione	Indicatori principali: *carry*, *parity*, *zero*, *sign*, *overflow*
SHL SHR	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Fa scorrere i bit, rispettivamente verso sinistra o verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto). Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # · · · #
SAL	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Funziona esattamente come '**SHL**', ma esiste in quanto è la controparte di '**SAR**', riferendosi a uno scorrimento aritmetico.	c p z s o # · · · #
SAR	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Fa scorrere i bit verso destra (l'ultima cifra perduta finisce nell'indicatore del riporto), mantenendo il segno originale. Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # · · · ·
RCL RCR	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Ruota i bit, rispettivamente verso sinistra o verso destra, utilizzando anche l'indicatore di riporto (*carry*). Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o t · · · · # · · · #
ROL ROR	***reg***, 1 ***mem***, 1 ***reg*** ***mem***	Ruota i bit, rispettivamente verso sinistra o verso destra. Se appare un solo operando, la rotazione viene eseguita ***CL*** volte. Se il valore immediato è maggiore di uno, è il compilatore che ripete l'istruzione per più volte.	c p z s o # · · · #

64.6.1 Scorrimento logico

Viene proposto un esempio in cui si vede l'uso delle istruzioni '**SHL**' e '**SHR**'. Si parte da un valore che viene fatto scorrere a sinistra, poi viene ripristinato e quindi viene fatto scorrere a destra.

```
# Scorrimento logico (logic shift)
#
.section .data
op1:    .byte   0b01001100      # 0x4C
#
.section .text
.globl _start
#
_start:
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    shl  $1,  %al  # Sposta a sinistra le cifre di una
                   # posizione.
bp1:               # AL = 0b10011000 0x98 overflow
    shl  $1,  %al  # Sposta a sinistra le cifre di una
                   # posizione.
bp2:               # AL = 0b00110000 0x30 carry, overflow
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    shr  $1,  %al  # Sposta a destra le cifre di una
                   # posizione.
bp3:               # AL = 0b00100110 0x26
    shr  $2,  %al  # Sposta a destra le cifre di due
                   # posizioni.
bp4:               # AL = 0b00001001 0x09 carry
    mov $0,   %ebx # Restituisce il valore contenuto in AL
    mov %al,  %bl  # copiandolo nella parte inferiore del
    mov $1,   %eax # registro EBX ed eseguendo la chiamata
    int $0x80      # di sistema 1 (exit).
```

```
; Scorrimento logico (logic shift)
;
section .data
op1:    dd      01001100b       ; 0x4C
;
section .text
global _start
;
_start:
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    shl  al, 1     ; Sposta a sinistra le cifre di una
                   ; posizione.
bp1:               ; AL = 10011000b 0x98 overflow
    shl  al,  1    ; Sposta a sinistra le cifre di una
                   ; posizione.
bp2:               ; AL = 00110000b 0x30 carry, overflow
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    shr  al, 1     ; Sposta a destra le cifre di una
                   ; posizione.
bp3:               ; AL = 00100110b 0x26
    shr  al, 2     ; Sposta a destra le cifre di due
                   ; posizioni.
bp4:               ; AL = 00001001b 0x09 carry
    mov ebx, 0     ; Restituisce il valore contenuto in AL
    mov bl, al     ; copiandolo nella parte inferiore del
    mov eax, 1     ; registro EBX ed eseguendo la chiamata
    int 0x80       ; di sistema 1 (exit).
```

Viene mostrato lo stato del registro ***EAX*** e degli indicatori, nei vari punti di sospensione previsti:

bp1

```
eax             0x98    152
eflags          0xa92   [ AF SF IF OF ]
```

Bisogna tenere presente che l'ultima istruzione eseguita riguarda solo la porzione ***AL*** del registro ***EAX***; pertanto, dal momento che il bit più significativo è a uno, si attiva l'indicatore del segno e anche quello del traboccamento, dato che, se lo scorrimento riguardasse un valore con segno, il valore ottenuto non sarebbe più valido (perché a questo punto si sarebbe trasformato in un numero negativo).

bp2

```
eax             0x30    48
eflags          0xa17   [ CF PF AF IF OF ]
```

Sempre tenendo conto che le istruzioni eseguite riguardano solo ***AL***, si può osservare che la perdita della cifra più significativa a uno si traduce nell'attivazione del riporto; inoltre, se si trattasse di un valore con segno, anche questa volta si sarebbe verificata un'inversione (da negativo a positivo) e per questo si attiva ancora l'indicatore di traboccamento.

bp3

```
eax             0x26    38
eflags          0x212   [ AF IF ]
```

Questo scorrimento a destra fa perdere solo delle cifre a zero, pertanto il segno non cambia e non c'è alcun riporto (resto).

bp4

```
eax             0x9     9
eflags          0x217   [ CF PF AF IF ]
```

Questo ulteriore scorrimento, di due posizioni, comporta la fuoriuscita di una cifra a uno che implica l'attivazione dell'indicatore del riporto (resto).

64.6.2 Scorrimento aritmetico

Viene proposto un esempio analogo a quello della sezione precedente, in cui si vede l'uso delle istruzioni '**SAL**' e '**SAR**'. Si parte da un valore negativo che viene fatto scorrere a sinistra, poi viene ripristinato e quindi viene fatto scorrere a destra.

```
# Scorrimento aritmetico
#
.section .data
op1:    .byte   0b11001100      # 0xCC
#
.section .text
.globl _start
#
_start:
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    sal  $1,  %al  # Sposta a sinistra le cifre di una
                   # posizione.
bp1:               # AL = 0b10011000 0x98 carry
    sal  $1,  %al  # Sposta a sinistra le cifre di una
                   # posizione.
bp2:               # AL = 0b00110000 0x30 carry, overflow
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    sar  $1,  %al  # Sposta a destra le cifre di una
                   # posizione.
bp3:               # AL = 0b11100110 0xE6
    sar  $2,  %al  # Sposta a destra le cifre di due
                   # posizioni.
bp4:               # AL = 0b11111001 0xF9 carry
    mov $0,   %ebx # Restituisce il valore contenuto in AL
    mov %al,  %bl  # copiandolo nella parte inferiore del
    mov $1,   %eax # registro EBX ed eseguendo la chiamata
    int $0x80      # di sistema 1 (exit).
```

```
; Scorrimento aritmetico
;
section .data
op1:    dd      11001100b       ; 0xCC
;
section .text
global _start
;
_start:
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    sal  al, 1     ; Sposta a sinistra le cifre di una
                   ; posizione.
bp1:               ; AL = 10011000b 0x98 carry
    sal  al,  1    ; Sposta a sinistra le cifre di una
                   ; posizione.
bp2:               ; AL = 00110000b 0x30 carry, overflow
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    sar  al, 1     ; Sposta a destra le cifre di una
                   ; posizione.
bp3:               ; AL = 11100110b 0xE6
    sar  al, 2     ; Sposta a destra le cifre di due
                   ; posizioni.
bp4:               ; AL = 11111001b 0xF9 carry
    mov ebx, 0     ; Restituisce il valore contenuto in AL
    mov bl, al     ; copiandolo nella parte inferiore del
    mov eax, 1     ; registro EBX ed eseguendo la chiamata
    int 0x80       ; di sistema 1 (exit).
```

Viene mostrato lo stato del registro ***EAX*** e degli indicatori, nei vari punti di sospensione previsti:

bp1

```
eax             0x98    152
eflags          0x293   [ CF AF SF IF ]
```

Bisogna tenere presente che l'ultima istruzione eseguita riguarda solo la porzione ***AL*** del registro ***EAX***; pertanto, dal momento che il bit più significativo è a uno, è attivo l'indicatore del segno e anche quello del riporto, avendo perduto una cifra a uno. Il numero non ha cambiato di segno e quindi l'indicatore di traboccamento non è attivo.

bp2

```
eax             0x30    48
eflags          0xa17   [ CF PF AF IF OF ]
```

Sempre tenendo conto che le istruzioni eseguite riguardano solo ***AL***, si può osservare che anche questa volta è stata persa una cifra più significativa a uno, essendo attivo l'indicatore del riporto, ma la cosa più importante è che adesso il valore ha cambiato di segno e così si vede attivo anche l'indicatore di traboccamento.

bp3

```
eax             0xe6    230
eflags          0x292   [ AF SF IF ]
```

Questo scorrimento a destra fa perdere solo delle cifre a zero, pertanto non c'è alcun riporto (resto).

bp4

```
eax             0xf9    249
eflags          0x297   [ CF PF AF SF IF ]
```

Questo ulteriore scorrimento, di due posizioni, comporta la fuoriuscita di una cifra a uno che implica l'attivazione dell'indicatore del riporto (resto).

64.6.3 Rotazione

Viene proposto un esempio analogo a quello delle sezioni precedenti, in cui si vede l'uso delle istruzioni '**ROL**' e '**ROR**'. Si parte da un valore che viene fatto ruotare a sinistra, poi viene ripristinato e quindi viene fatto ruotare a destra.

```
# Rotazione
#
.section .data
op1:    .byte   0b11001100      # 0xCC
#
.section .text
.globl _start
#
_start:
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    rol  $1,  %al  # Ruota a sinistra le cifre di una
                   # posizione.
bp1:               # AL = 0b10011001 0x99 carry
    rol  $1,  %al  # Ruota a sinistra le cifre di una
                   # posizione.
bp2:               # AL = 0b00110011 0x33 carry, overflow
    mov  op1, %al  # Inizializza AL con il valore di
                   # partenza.
    ror  $1,  %al  # Ruota a destra le cifre di una
                   # posizione.
bp3:               # AL = 0b01100110 0x66 overflow
    ror  $2,  %al  # Ruota a destra le cifre di due
                   # posizioni.
bp4:               # AL = 0b10011001 0x99 carry, overflow
    mov $0,   %ebx # Restituisce il valore contenuto in AL
    mov %al,  %bl  # copiandolo nella parte inferiore del
    mov $1,   %eax # registro EBX ed eseguendo la chiamata
    int $0x80      # di sistema 1 (exit).
```

```
; Rotazione
;
section .data
op1:    dd      11001100b       ; 0xCC
;
section .text
global _start
;
_start:
    mov  al, [op1] ; Inizializza AL con il valore di
                   ; partenza.
    rol  al, 1     ; Ruota a sinistra le cifre di una
                   ; posizione.
```

```
bp1:              ; AL = 10011001b 0x99 carry
    rol  al,  1   ; Ruota a sinistra le cifre di una
                  ; posizione.
bp2:              ; AL = 00110011b 0x33 carry, overflow
    mov  al, [op1] ; Inizializza AL con il valore di
                  ; partenza.
    ror  al, 1    ; Ruota a destra le cifre di una
                  ; posizione.
bp3:              ; AL = 01100110b 0x66 overflow
    ror  al, 2    ; Ruota a destra le cifre di due
                  ; posizioni.
bp4:              ; AL = 10011001b 0x99 carry, overflow
    mov ebx, 0    ; Restituisce il valore contenuto in AL
    mov bl, al    ; copiandolo nella parte inferiore del
    mov eax, 1    ; registro EBX ed eseguendo la chiamata
    int 0x80      ; di sistema 1 (exit).
```

Viene mostrato lo stato del registro ***EAX*** e degli indicatori, nei vari punti di sospensione previsti:

bp1

```
eax             0x99     153
eflags          0x293    [ CF AF SF IF ]
```

Si deve tenere conto che l'ultima istruzione eseguita riguarda solo la porzione ***AL*** del registro ***EAX***. La rotazione ha fatto uscire a sinistra una cifra a uno che rientra nella parte destra. La cifra spostata la si ritrova nell'indicatore di riporto.

bp2

```
eax             0x33     51
eflags          0xa93    [ CF AF SF IF OF ]
```

Questa ulteriore rotazione a sinistra fa uscire un'altra cifra a uno, che viene segnalata nel riporto, ma il bit più significativo passa a zero e quindi l'indicatore di traboccamento viene attivato.

bp3

```
eax             0x66     102
eflags          0xa92    [ AF SF IF OF ]
```

Questa rotazione a destra fa perdere solo una cifra a zero, pertanto non c'è alcun riporto (resto), ma dal momento che rientra a sinistra, l'indicatore di traboccamento manifesta l'ipotesi di cambiamento del segno.

bp4

```
eax             0x99     153
eflags          0xa93    [ CF AF SF IF OF ]
```

Questa ulteriore rotazione, di due posizioni, comporta la fuoriuscita di una cifra a uno che implica l'attivazione dell'indicatore del riporto (resto). Dal momento che la cifra rientra a sinistra, l'indicatore di traboccamento segnala l'ipotesi di cambiamento del segno.

64.6.4 Rotazione con riporto

Viene proposto un esempio analogo a quello della sezione precedente, in cui si vede l'uso delle istruzioni '**RCL**' e '**RCR**'. Si parte da un valore che viene fatto ruotare a sinistra, poi viene ripristinato e quindi viene fatto ruotare a destra. Qui viene usata anche l'istruzione '**CLC**' per azzerare il riporto.

```
# Rotazione con riporto
#
.section .data
op1:    .byte   0b11001101      # 0xCD
#
.section .text
.globl _start
#
_start:
    clc             # Azzera il riporto.
    mov  op1, %al   # Inizializza AL con il valore di
                    # partenza.
    rcl  $1,  %al   # Ruota a sinistra le cifre di una
                    # posizione.
bp1:                # AL = 0b10011010 0x9A carry
```



```
    rcl  $1,  %al   # Ruota a sinistra le cifre di una
                    # posizione.
bp2:                # AL = 0b00110101 0x35 carry, overflow
    clc             # Azzera il riporto.
    mov  op1, %al   # Inizializza AL con il valore di
                    # partenza.
    rcr  $1,  %al   # Ruota a destra le cifre di una
                    # posizione.
bp3:                # AL = 0b01100110 0x66 carry, overflow
    rcr  $2,  %al   # Ruota a destra le cifre di due
                    # posizioni.
bp4:                # AL = 0b01011001 0x59 carry, overflow
    mov $0,   %ebx # Restituisce il valore contenuto in AL
    mov %al,  %bl  # copiandolo nella parte inferiore del
    mov $1,   %eax # registro EBX ed eseguendo la chiamata
    int $0x80      # di sistema 1 (exit).
```

```
; Rotazione con riporto
;
section .data
op1:    dd      11001101b       ; 0xCD
;
section .text
global _start
;
_start:
    clc             ; Azzera il riporto.
    mov  al, [op1] ; Inizializza AL con il valore di
                    ; partenza.
    rcl  al, 1     ; Ruota a sinistra le cifre di una
                    ; posizione.
bp1:                ; AL = 10011010b 0x9A carry
    rcl  al,  1    ; Ruota a sinistra le cifre di una
                    ; posizione.
bp2:                ; AL = 00110101b 0x35 carry, overflow
    clc             ; Azzera il riporto.
    mov  al, [op1] ; Inizializza AL con il valore di
                    ; partenza.
    rcr  al, 1     ; Ruota a destra le cifre di una
                    ; posizione.
bp3:                ; AL = 01100110b 0x66 carry, overflow
    rcr  al, 2     ; Ruota a destra le cifre di due
                    ; posizioni.
bp4:                ; AL = 01011001b 0x59 carry, overflow
    mov ebx, 0     ; Restituisce il valore contenuto in AL
    mov bl, al     ; copiandolo nella parte inferiore del
    mov eax, 1     ; registro EBX ed eseguendo la chiamata
    int 0x80       ; di sistema 1 (exit).
```

Viene mostrato lo stato del registro ***EAX*** e degli indicatori, nei vari punti di sospensione previsti:

bp1

```
eax             0x9a     154
eflags          0x293    [ CF AF SF IF ]
```

Si deve tenere conto che l'ultima istruzione eseguita riguarda solo la porzione ***AL*** del registro ***EAX***. La rotazione ha fatto uscire a sinistra una cifra a uno che passa nel riporto, mentre a destra entra il valore precedente del riporto (zero).

bp2

```
eax             0x35     53
eflags          0xa93    [ CF AF SF IF OF ]
```

Questa ulteriore rotazione a sinistra fa uscire un'altra cifra a uno che passa nel, mentre entra a destra la cifra a uno del riporto precedente. Dopo la rotazione il bit più significativo passa a zero e quindi l'indicatore di traboccamento viene attivato.

bp3

```
eax             0x66     102
eflags          0xa93    [ CF AF SF IF OF ]
```

Questa rotazione a destra fa perdere una cifra a uno che si trasferisce del riporto; inoltre, dato che il riporto precedente era zero, a sinistra si inserisce una cifra a zero, la quale fa scattare l'indicatore di traboccamento (nell'ipotesi di un valore con segno).

bp4

```
eax             0x59     89
eflags          0xa93    [ CF AF SF IF OF ]
```

Questa ulteriore rotazione, di due posizioni, comporta la fuoriuscita di una cifra a zero e poi a uno che implica l'attivazione dell'indicatore del riporto (resto). In precedenza c'era stato un riporto che attualmente risulta inserito subito dopo la cifra più significativa. Dal momento l'ultimo scorrimento (dei due eseguiti qui) fa cambiare la cifra più significativa, si attiva l'indicatore di traboccamento.

64.7 Esempi con i confronti

Viene mostrato un esempio di programma, con l'unico scopo di dimostrare il funzionamento dell'istruzione di confronto, attraverso l'aiuto di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**`--gstabs`**', mentre con NASM è bene aggiungere l'opzione '**`-g`**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

$ **as --gstabs -o** ***nome*****.o** ***nome*****.s** [*Invio*]

$ **nasm -g -f elf -o** ***nome*****.o** ***nome*****.s** [*Invio*]

```
# Confronto
#
.section .text
.globl _start
#
_start:
    mov $0b01000000, %al        # A =  64       A =   64
    mov $0b10000000, %bl        # B = 128       B = -128
    cmp %bl, %al                # carry, segno, overflow
bp1:
    mov $0b01000000, %al        # A =  64       A =   64
    mov $0b11000000, %bl        # B = 192       B =  -64
    cmp %bl, %al                # carry, segno, overflow
bp2:
    mov $0b01000000, %al        # A =  64       A =   64
    mov $0b00000000, %bl        # B =   0       B =    0
    cmp %bl, %al                #
bp3:
    mov $0b01000000, %al        # A =  64       A =   64
    mov $0b01000000, %bl        # B =  64       B =   64
    cmp %bl, %al                # zero
bp4:
    mov $0b00000000, %al        # A =   0       A =    0
    mov $0b10000000, %bl        # B = 128       B = -128
    cmp %bl, %al                # carry, segno, overflow
bp5:
    mov $0b00000000, %al        # A =   0       A =    0
    mov $0b11000000, %bl        # B = 192       B =  -64
    cmp %bl, %al                # carry
bp6:
    mov $0b00000000, %al        # A =   0       A =    0
    mov $0b00000000, %bl        # B =   0       B =    0
    cmp %bl, %al                # zero
bp7:
    mov $0b00000000, %al        # A =   0       A =    0
    mov $0b01000000, %bl        # B =  64       B =   64
    cmp %bl, %al                # carry, segno
bp8:
    mov $0b11000000, %al        # A = 192       A =  -64
    mov $0b10000000, %bl        # B = 128       B = -128
    cmp %bl, %al                #
bp9:
    mov $0b11000000, %al        # A = 192       A =  -64
    mov $0b11000000, %bl        # B = 192       B =  -64
    cmp %bl, %al                # zero
bp10:
    mov $0b11000000, %al        # A = 192       A =  -64
    mov $0b00000000, %bl        # B =   0       B =    0
    cmp %bl, %al                # segno
bp11:
    mov $0b11000000, %al        # A = 192       A =  -64
    mov $0b01000000, %bl        # B =  64       B =   64
    cmp %bl, %al                # segno
bp12:
    mov $0b10000000, %al        # A = 128       A = -128
    mov $0b10000000, %bl        # B = 128       B = -128
    cmp %bl, %al                # zero
bp13:
    mov $0b10000000, %al        # A = 128       A = -128
    mov $0b11000000, %bl        # B = 192       B =  -64
    cmp %bl, %al                # carry, segno
bp14:
    mov $0b10000000, %al        # A = 128       A = -128
    mov $0b00000000, %bl        # B =   0       B =    0
    cmp %bl, %al                # segno
bp15:
    mov $0b10000000, %al        # A = 128       A = -128
    mov $0b01000000, %bl        # B =  64       B =   64
    cmp %bl, %al                # overflow
bp16:
    mov $0,   %ebx      # Conclude il funzionamento con la
    mov $1,   %eax      # chiamata di sistema 1 (exit).
    int $0x80           #
```

```
; Confronto
;
section .text
global _start
;
_start:
    mov al, 01000000b           ; A =  64       A =   64
    mov bl, 10000000b           ; B = 128       B = -128
    cmp al, bl                  ; carry, segno, overflow
bp1:
    mov al, 01000000b           ; A =  64       A =   64
    mov bl, 11000000b           ; B = 192       B =  -64
    cmp al, bl                  ; carry, segno, overflow
bp2:
    mov al, 01000000b           ; A =  64       A =   64
    mov bl, 00000000b           ; B =   0       B =    0
    cmp al, bl                  ;
bp3:
    mov al, 01000000b           ; A =  64       A =   64
    mov bl, 01000000b           ; B =  64       B =   64
    cmp al, bl                  ; zero
bp4:
    mov al, 00000000b           ; A =   0       A =    0
    mov bl, 10000000b           ; B = 128       B = -128
    cmp al, bl                  ; carry, segno, overflow
bp5:
    mov al, 00000000b           ; A =   0       A =    0
    mov bl, 11000000b           ; B = 192       B =  -64
    cmp al, bl                  ; carry
bp6:
    mov al, 00000000b           ; A =   0       A =    0
    mov bl, 00000000b           ; B =   0       B =    0
    cmp al, bl                  ; zero
bp7:
    mov al, 00000000b           ; A =   0       A =    0
    mov bl, 01000000b           ; B =  64       B =   64
    cmp al, bl                  ; carry, segno
bp8:
    mov al, 11000000b           ; A = 192       A =  -64
    mov bl, 10000000b           ; B = 128       B = -128
    cmp al, bl                  ;
bp9:
    mov al, 11000000b           ; A = 192       A =  -64
    mov bl, 11000000b           ; B = 192       B =  -64
    cmp al, bl                  ; zero
bp10:
    mov al, 11000000b           ; A = 192       A =  -64
    mov bl, 00000000b           ; B =   0       B =    0
    cmp al, bl                  ; segno
bp11:
    mov al, 11000000b           ; A = 192       A =  -64
    mov bl, 01000000b           ; B =  64       B =   64
    cmp al, bl                  ; segno
bp12:
    mov al, 10000000b           ; A = 128       A = -128
    mov bl, 10000000b           ; B = 128       B = -128
    cmp al, bl                  ; zero
bp13:
    mov al, 10000000b           ; A = 128       A = -128
    mov bl, 11000000b           ; B = 192       B =  -64
    cmp al, bl                  ; carry, segno
bp14:
    mov al, 10000000b           ; A = 128       A = -128
```

```
    mov bl, 00000000b           ; B =   0       B =    0
    cmp al, bl                  ; segno
bp15:
    mov al, 10000000b           ; A = 128       A = -128
    mov bl, 01000000b           ; B =  64       B =   64
    cmp al, bl                  ; overflow
bp16:
    mov ebx, 0  ; Conclude il funzionamento con la
    mov eax, 1  ; chiamata di sistema 1 (exit).
    int 0x80    ;
```

Tra i commenti si possono osservare i valori confrontati, interpretandoli sia con segno, sia senza segno, assieme all'effetto atteso sugli indicatori. Viene mostrato lo stato degli indicatori nei vari punti di sospensione previsti, con l'ausilio di GDB:

bp1

```
eax             0x40    64
ebx             0x80    128
eflags          0xa87   [ CF PF SF IF OF ]
```

bp2

```
eax             0x40    64
ebx             0xc0    192
eflags          0xa83   [ CF SF IF OF ]
```

bp3

```
eax             0x40    64
ebx             0x0     0
eflags          0x202   [ IF ]
```

bp4

```
eax             0x40    64
ebx             0x40    64
eflags          0x246   [ PF ZF IF ]
```

bp5

```
eax             0x0     0
ebx             0x80    128
eflags          0xa83   [ CF SF IF OF ]
```

bp6

```
eax             0x0     0
ebx             0xc0    192
eflags          0x203   [ CF IF ]
```

bp7

```
eax             0x0     0
ebx             0x0     0
eflags          0x246   [ PF ZF IF ]
```

bp8

```
eax             0x0     0
ebx             0x40    64
eflags          0x287   [ CF PF SF IF ]
```

bp9

```
eax             0xc0    192
ebx             0x80    128
eflags          0x202   [ IF ]
```

bp10

```
eax             0xc0    192
ebx             0xc0    192
eflags          0x246   [ PF ZF IF ]
```

bp11

```
eax             0xc0    192
ebx             0x0     0
eflags          0x286   [ PF SF IF ]
```

bp12

```
eax             0xc0    192
ebx             0x40    64
eflags          0x282   [ SF IF ]
```


bp13

```
eax             0x80    128
ebx             0x80    128
eflags          0x246   [ PF ZF IF ]
```

bp14

```
eax             0x80    128
ebx             0xc0    192
eflags          0x287   [ CF PF SF IF ]
```

bp15

```
eax             0x80    128
ebx             0x0     0
eflags          0x282   [ SF IF ]
```

bp16

```
eax             0x80    128
ebx             0x40    64
eflags          0xa02   [ IF OF ]
```

64.8 Le istruzioni di salto

Con il linguaggio macchina, le strutture di controllo si realizzano solo attraverso le istruzioni di salto. Una di queste istruzioni è incondizionata, mentre le altre sono sottoposte al verificarsi di una condizione. A loro volta, le istruzioni di salto condizionato si dividono in due gruppi: uno riferito al controllo dello stato di un certo indicatore, l'altro riferito virtualmente a un confronto di valori.

64.8.1 Portata del salto

In generale, le istruzioni di salto hanno un solo operando, costituito dal riferimento all'indirizzo di memoria da raggiungere. Il compilatore traduce il riferimento all'indirizzo di memoria in un «dislocamento», ovvero nella quantità di byte da percorrere, in avanti o indietro. A questo proposito, ciò che rappresenta il riferimento alla memoria può avere dimensioni diverse e questo significa che l'istruzione può richiedere di precisare la dimensione del numero che rappresenta tale dislocamento. In modo predefinito, la dimensione del numero usato per indicare il dislocamento è quella di un registro comune.

Si osservi che la possibilità di dichiarare esplicitamente l'entità del dislocamento dipende dal compilatore; d'altro canto, sarebbe compito del compilatore determinarlo automaticamente.

64.8.2 Salto incondizionato

Il salto incondizionato si ottiene con l'istruzione '**JMP**':

```
JMP imm
```

Con un linguaggio assemblatore, il valore immediato richiesto come operando si ottiene con un simbolo, ovvero indicando il nome di un'etichetta già dichiarata altrove nel sorgente:

```
# Salto incondizionato
#
.section .text
.globl _start
#
_start:
    mov $1, %ebx
bp1:
    jmp bp3
bp2:
    mov $2, %ebx     # Questa istruzione non viene eseguita.
bp3:
    mov $1, %eax     # Conclude il funzionamento con la
    int $0x80        # chiamata di sistema 1 (exit).
```

```
; Salto incondizionato
;
section .text
```

```
global _start
;
_start:
    mov ebx, 1
bp1:
    jmp bp3
bp2:
    mov ebx, 2       ; Questa istruzione non viene eseguita.
bp3:
    mov eax, 1       ; Conclude il funzionamento con la
    int 0x80         ; chiamata di sistema 1 (exit).
```

In questo esempio si vede che l'istruzione contenuta tra i punti '**bp2**' e '**bp3**' non viene mai eseguita.

64.8.3 Salto condizionato dallo stato di un indicatore

Un gruppo di istruzioni di salto condizionato dipende dallo stato di un certo indicatore. Queste istruzioni hanno la forma seguente, dove la lettera ***x*** identifica l'indicatore da controllare:

```
Jx  imm
```

```
JNx  imm
```

Per esempio, l'istruzione '**JZ**' salta se l'indicatore zero è attivo, mentre '**JNZ**' salta se l'indicatore zero non è attivo.

Tabella 64.135. Salti condizionati dallo stato di un indicatore particolare.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione
JC JNC	***imm***	Salta se l'indicatore del riporto (*carry*), rispettivamente, è attivo, oppure non è attivo.
JO JNO	***imm***	Salta se l'indicatore di traboccamento (*overflow*), rispettivamente, è attivo, oppure non è attivo.
JS JNS	***imm***	Salta se l'indicatore di segno (*sign*), rispettivamente, è attivo, oppure non è attivo.
JZ JNZ	***imm***	Salta se l'indicatore di zero, rispettivamente, è attivo, oppure non è attivo.
JP JNP	***imm***	Salta se l'indicatore di parità, rispettivamente, è attivo, oppure non è attivo.

64.8.4 Salto condizionato da un confronto

Il gruppo più importante di istruzioni di salto condizionato dipende da un confronto, che di solito si realizza con l'istruzione '**CMP**'. In pratica, l'istruzione '**CMP**' simula una sottrazione, aggiornando gli indicatori come se si trattasse di una sottrazione vera e propria; successivamente, l'istruzione di salto condizionato verifica gli indicatori e si comporta di conseguenza.

Da quanto descritto si deve comprendere che, anche se le istruzioni di questo tipo richiamano l'idea del confronto tra due valori, in pratica dipendono da indicatori che possono essere stati modificati da istruzioni di tipo differente.

Dal momento che, ai fini del confronto tra due valori, la valutazione degli indicatori va fatta diversamente a seconda che si tratti di interi senza segno o con segno, queste istruzioni sono suddivise in sottogruppi.

Tabella 64.136. Salti condizionati dall'esito di un confronto con valori interi senza segno.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione
JA JNBE	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst*** > ***org*** THEN go to ***imm***
JAE JNB	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** >= ***org*** THEN go to ***imm***
JB JNAE	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst*** < ***org*** THEN go to ***imm***
JBE JNA	***imm***	Dopo un confronto di valori senza segno, salta se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** <= ***org*** THEN go to ***imm***

Tabella 64.137. Salti condizionati dall'esito di un confronto con valori interi, indipendentemente dal segno.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione
JE	***imm***	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** == ***org*** THEN go to ***imm***
JNE	***imm***	Dopo un confronto, indipendentemente dal segno, salta se la destinazione era diversa dall'origine. CMP ***dst***, ***org*** IF ***dst*** =/= ***org*** THEN go to ***imm***

Tabella 64.138. Salti condizionati dall'esito di un confronto con valori interi con segno.

Nome	Operandi: ***dst***, ***org1***, ***org2***	Descrizione
JG JNLE	***imm***	Dopo un confronto con segno, salta se la destinazione era maggiore dell'origine. CMP ***dst***, ***org*** IF ***dst*** > ***org*** THEN go to ***imm***
JGE JNL	***imm***	Dopo un confronto con segno, salta se la destinazione era maggiore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** >= ***org*** THEN go to ***imm***
JL JNGE	***imm***	Dopo un confronto con segno, salta se la destinazione era minore dell'origine. CMP ***dst***, ***org*** IF ***dst*** < ***org*** THEN go to ***imm***
JLE JNG	***imm***	Dopo un confronto con segno, salta se la destinazione era minore o uguale all'origine. CMP ***dst***, ***org*** IF ***dst*** <= ***org*** THEN go to ***imm***

64.8.5 Cicli

È possibile realizzare dei cicli enumerativi attraverso delle istruzioni simili a quelle di salto condizionato, dove in pratica, dopo il blocco di istruzioni da reiterare, viene verificata la condizione e quindi, eventualmente, si ripete il ciclo. Si distinguono tre casi:

```
LOOP imm
```

```
LOOPE imm | LOOPZ imm
```

```
LOOPNE imm | LOOPNZ imm
```

In tutte le situazioni, il valore immediato che viene fornito come operando si riferisce al dislocamento dell'indirizzo di memoria da raggiungere (cosa che viene tradotta dal compilatore, sostituendo il simbolo con il numero appropriato). Il dislocamento consentito è breve ($\pm$128 byte), quindi non si possono realizzare cicli contenenti tante istruzioni.

In tutte le situazioni, l'istruzione decrementa prima il registro ***ECX*** (oppure solo ***CX***, se viene dichiarato l'uso di una dimensione «breve», ovvero a soli 16 bit[8]) senza alterare gli indicatori, quindi verifica se tale registro è diverso da zero. Nel caso dell'istruzione '**LOOP**', il fatto che il registro, dopo il decremento di una unità, contenga ancora un valore diverso da zero, è sufficiente per far scattare la ripetizione del ciclo; nel caso di '**LOOPE**' o di '**LOOPZ**', è necessario anche che l'indicatore zero sia attivo; per converso, con '**LOOPNE**' o '**LOOPNZ**', è necessario anche che l'indicatore zero non sia attivo.

64.9 Esempi di programmi con strutture di controllo

Vengono mostrati esempi di programmi estremamente banali, per dimostrare il funzionamento delle strutture di controllo, basate sostanzialmente su istruzioni di salto condizionato, attraverso l'aiuto di GDB. Per la compilazione, se si utilizza GNU AS è bene ricordare di inserire l'opzione '**--gstabs**', mentre con NASM è bene aggiungere l'opzione '**-g**', in modo da poter gestire più facilmente GDB, disponendo dei riferimenti al sorgente:

```
$ as --gstabs -o nome.o nome.s [Invio]
```

```
$ nasm -g -f elf -o nome.o nome.s [Invio]
```

64.9.1 Somma attraverso l'incremento unitario

Viene mostrato un programma che esegue la somma di due valori interi senza segno, incrementando progressivamente il primo addendo, di una unità, corrispondentemente alla riduzione di una unità del secondo. Quando il secondo addendo è stato ridotto a zero, il primo contiene il risultato della somma. Il ciclo con cui si incrementa il primo addendo è controllato dall'istruzione '**LOOP**':

```
# op1 + op2
#
.section .data
op1:    .int    0x00000007      # Intero senza segno.
op2:    .int    0x00000002      # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1, %eax   # Primo addendo.
    mov op2, %ecx   # Secondo addendo.
bp1:
    cmp $0, %ecx    # Se il secondo addendo è zero, non
    je end_do_somma # esegue il ciclo di incrementi.
do_somma:
    inc %eax        # Aggiunge una unità a EAX.
    loop do_somma   # Ripete il ciclo se nel frattempo
                    # ECX non si è azzerato.
end_do_somma:
    mov %eax, %ebx  # Restituisce il valore
    mov $1,   %eax  # ottenuto dalla somma.
    int $0x80       #
```

```
; op1 + op2
;
section .data
op1:    dd      0x00000007      ; Intero senza segno.
op2:    dd      0x00000002      ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov eax, [op1]  ; Primo addendo.
    mov ecx, [op2]  ; Secondo addendo.
bp1:
    cmp ecx, 0      ; Se il secondo addendo è zero, non
    je end_do_somma ; esegue il ciclo di incrementi.
do_somma:
    inc eax         ; Aggiunge una unità a EAX.
    loop do_somma   ; Ripete il ciclo se nel frattempo
                    ; ECX non si è azzerato.
end_do_somma:
    mov ebx, eax    ; Restituisce il valore ottenuto
    mov eax, 1      ; dalla somma.
    int 0x80        ;
```

Come si può vedere, il ciclo di incremento è racchiuso tra i simboli '**do_somma**' e '**end_do_somma**'; inoltre, prima di entrare nel ciclo di somma, viene verificato che il secondo addendo sia diverso da zero, perché se è pari a zero, viene saltato il ciclo di somma, dal momento che ***EAX*** contiene già il valore corretto.

Viene mostrata una seconda versione del ciclo, dove si sostituisce l'istruzione '**LOOP**' con altre istruzioni di salto condizionato:

```
    cmp $0, %ecx    # Se il secondo addendo è zero, non
    je end_do_somma # esegue il ciclo di incrementi.
do_somma:
    inc %eax        # Aggiunge una unità a EAX.
    dec %ecx        # Riduce ECX di una unità.
    cmp $0, %ecx    # Se il secondo addendo è ancora diverso
    jnz do_somma    # da zero, allora ripete il ciclo.
end_do_somma:
```

```
    cmp ecx, 0      ; Se il secondo addendo è zero, non
    je end_do_somma ; esegue il ciclo di incrementi.
do_somma:
    inc eax         ; Aggiunge una unità a EAX.
    dec ecx         ; Riduce ECX di una unità.
    cmp ecx, 0      ; Se il secondo addendo è ancora diverso
    jnz do_somma    ; da zero, allora ripete il ciclo.
end_do_somma:
```

Viene mostrata una terza versione del ciclo, dove il controllo di uscita avviene solo all'inizio:

```
do_somma:
    cmp $0, %ecx    # Se il secondo addendo è zero, esce dal
    je end_do_somma # ciclo di incrementi.
    dec %ecx        # Riduce di una unità ECX.
    inc %eax        # Aggiunge una unità a EAX.
    jmp do_somma    # Ritorna all'inizio del ciclo.
end_do_somma:
```

```
do_somma:
    cmp ecx, 0      ; Se il secondo addendo è zero, esce dal
    je end_do_somma ; ciclo di incrementi.
    dec ecx         ; Riduce di una unità ECX.
    inc eax         ; Aggiunge una unità a EAX.
    jmp do_somma    ; Ritorna all'inizio del ciclo.
end_do_somma:
```

64.9.2 Moltiplicazione attraverso la somma

Viene mostrato un programma che esegue la moltiplicazione di due valori interi senza segno, sommando progressivamente il moltiplicando a un registro che inizialmente è pari a zero, corrispondentemente alla riduzione di una unità del moltiplicatore. Quando il moltiplicatore è stato ridotto a zero, il registro che viene incrementa-

to contiene il risultato della moltiplicazione. Il ciclo è controllato dall'istruzione '**LOOP**':

```
# op1 * op2
#
.section .data
op1:    .int    0x00000007      # Intero senza segno.
op2:    .int    0x00000003      # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1, %eax        # Moltiplicando.
    mov op2, %ecx        # Moltiplicatore.
    mov $0, %ebx         # Risultato.
bp1:
    cmp $0, %ecx         # Se il moltiplicatore è zero,
    je end_do_moltiplica # non esegue il ciclo di somme.
do_moltiplica:
    add %eax, %ebx       # Aggiunge il moltiplicando al
                         # risultato.
    loop do_moltiplica   # Ripete il ciclo se nel frattempo
                         # ECX non si è azzerato.
end_do_moltiplica:
    mov $1,   %eax       # Restituisce il valore ottenuto
    int $0x80            # dalla moltiplicazione.
```

```
; op1 * op2
;
section .data
op1:    dd      0x00000007      ; Intero senza segno.
op2:    dd      0x00000003      ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov eax, [op1]       ; Moltiplicando.
    mov ecx, [op2]       ; Moltiplicatore.
    mov ebx, 0           ; Risultato.
bp1:
    cmp ecx, 0           ; Se il moltiplicatore è zero,
    je end_do_moltiplica ; non esegue il ciclo di somme.
do_moltiplica:
    add ebx, eax         ; Aggiunge il moltiplicando al
                         ; risultato.
    loop do_moltiplica   ; Ripete il ciclo se nel frattempo
                         ; ECX non si è azzerato.
end_do_moltiplica:
    mov eax, 1           ; Restituisce il valore ottenuto
    int 0x80             ; dalla moltiplicazione.
```

Viene mostrata una seconda versione del ciclo, dove si sostituisce l'istruzione '**LOOP**' con altre istruzioni di salto condizionato:

```
    cmp $0, %ecx         # Se il moltiplicatore è zero,
    je end_do_moltiplica # non esegue il ciclo di somme.
do_moltiplica:
    add %eax, %ebx       # Aggiunge il moltiplicando al
                         # risultato.
    cmp $0, %ecx         # Se il moltiplicatore è ancora
    jnz do_moltiplica    # diverso da zero, allora ripete
                         # il ciclo.
end_do_moltiplica:
```

```
    cmp ecx, 0           ; Se il moltiplicatore è zero,
    je end_do_moltiplica ; non esegue il ciclo di somme.
do_moltiplica:
    add ebx, eax         ; Aggiunge il moltiplicando al
                         ; risultato.
    cmp ecx, 0           ; Se il moltiplicatore è ancora
    jnz do_moltiplica    ; diverso da zero, allora ripete
                         ; il ciclo.
end_do_moltiplica:
```

Viene mostrata una terza versione del ciclo, dove il controllo di uscita avviene solo all'inizio:

```
do_moltiplica:
    cmp $0, %ecx         # Se il moltiplicatore è zero,
    je end_do_moltiplica # esce dal ciclo di somme.
    dec %ecx             # Riduce di una unità il
                         # moltiplicatore.
    add %eax, %ebx       # Aggiunge il moltiplicando al
                         # risultato.
    jmp do_moltiplica    # Ritorna all'inizio del ciclo.
end_do_moltiplica:
```

```
do_moltiplica:
    cmp ecx, 0           ; Se il moltiplicatore è zero,
    je end_do_moltiplica ; esce dal ciclo di somme.
    dec ecx              ; Riduce di una unità il
                         ; moltiplicatore.
    add ebx, eax         ; Aggiunge il moltiplicando al
                         ; risultato.
    jmp do_moltiplica    ; Ritorna all'inizio del ciclo.
end_do_moltiplica:
```


64.9.3 Divisione attraverso la sottrazione

Viene mostrato un programma che esegue la divisione di due valori interi senza segno, sottraendo progressivamente il divisore a un registro che inizialmente è pari al valore del dividendo, corrispondentemente all'incremento di una unità del risultato (a partire da zero). Quando il registro che viene ridotto, di volta in volta, del valore del divisore, diventa minore del divisore, la divisione termina. Viene proposta una sola versione, con un ciclo controllato da una condizione iniziale:

```
# op1 / op2
#
.section .data
op1:    .int    33      # Intero senza segno.
op2:    .int    12      # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1,  %eax       # Dividendo.
    mov op2,  %ecx       # Divisore.
    mov $0,   %ebx       # Risultato.
    mov %eax, %edx       # Resto.
do_dividi:
    cmp %ecx, %edx       # Se il resto è minore del
    jb end_do_dividi     # divisore, conclude il ciclo di
                         # sottrazioni.
    sub %ecx, %edx       # Sottrae al resto il divisore.
    inc %ebx             # Incrementa il risultato di una
                         # unità.
    jmp do_dividi        # Torna all'inizio del ciclo.
end_do_dividi:
    mov $1,   %eax       # Restituisce il valore ottenuto
    int $0x80            # dalla moltiplicazione.
```

```
; op1 / op2
;
section .data
op1:    dd      33      ; Intero senza segno.
op2:    dd      12      ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov eax, [op1]       ; Dividendo.
    mov ecx, [op2]       ; Divisore.
    mov ebx, 0           ; Risultato.
    mov edx, eax         ; Resto.
do_dividi:
    cmp edx, ecx         ; Se il resto è minore del
    jb end_do_dividi     ; divisore, conclude il ciclo di
                         ; sottrazioni.
    sub edx, ecx         ; Sottrae al resto il divisore.
    inc ebx              ; Incrementa il risultato di una
                         ; unità.
    jmp do_dividi        ; Torna all'inizio del ciclo.
end_do_dividi:
    mov eax, 1           ; Restituisce il valore ottenuto
    int 0x80             ; dalla moltiplicazione.
```

64.9.4 Elevamento a potenza

Viene mostrato un programma che calcola la potenza di due numeri interi senza segno, moltiplicando progressivamente la base, corrispondentemente al decremento di un contatore che parte dal valore dell'esponente. Quando il contatore raggiunge lo zero, il ciclo di moltiplicazioni termina e il risultato della potenza viene emesso come valore di uscita, ammesso che sia abbastanza piccolo da poter essere rappresentato:

```
# op1 / op2
#
.section .data
op1:    .int    5     # Intero senza segno.
op2:    .int    3     # Intero senza segno.
#
.section .text
.globl _start
#
_start:
    mov op1,  %esi    # Base.
    mov op2,  %edi    # Esponente.
    mov $0,   %edx    # Risultato: EDX:EAX
    mov $1,   %eax    #
    mov %edi, %ecx    # Contatore.
do_potenza:
    cmp $0, %ecx      # Se il contatore ha raggiunto lo
    jz end_do_potenza # zero, conclude il ciclo di
                      # moltiplicazioni.
    mul %esi          # EDX:EAX := EAX*ESI.
    dec %ecx          # Riduce di una unità il contatore.
    jmp do_potenza    # Torna all'inizio del ciclo.
end_do_potenza:
    mov %eax, %ebx    # Restituisce il valore della potenza,
    mov $1,   %eax    # ammesso che sia abbastanza piccolo
    int $0x80         # da poter essere rappresentato come
                      # valore di uscita.
```

```
; op1 / op2
;
section .data
op1:    dd      5     ; Intero senza segno.
op2:    dd      3     ; Intero senza segno.
;
section .text
global _start
;
_start:
    mov esi, [op1]    ; Base.
    mov edi, [op2]    ; Esponente.
    mov edx, 0        ; Risultato: EDX:EAX
    mov eax, 1        ;
    mov ecx, edi      ; Contatore.
do_potenza:
    cmp ecx, 0        ; Se il contatore ha raggiunto lo
    jz end_do_potenza ; zero, conclude il ciclo di
                      ; moltiplicazioni.
    mul esi           ; EDX:EAX := EAX*ESI.
    dec ecx           ; Riduce di una unità il contatore.
    jmp do_potenza    ; Torna all'inizio del ciclo.
end_do_potenza:
    mov ebx, eax      ; Restituisce il valore della potenza,
    mov eax, 1        ; ammesso che sia abbastanza piccolo
    int 0x80          ; da poter essere rappresentato come
                      ; valore di uscita.
```

64.9.5 Moltiplicazione attraverso lo scorrimento e la somma

Viene mostrato un programma che calcola il prodotto di due numeri interi senza segno, sommando progressivamente il moltiplicando che viene fatto scorrere verso sinistra. Per comprendere il procedimento occorre fare mente locale al modo in cui la moltiplicazione verrebbe eseguita a mano: il moltiplicando viene sommato al risultato (che inizialmente è pari a zero) se la cifra meno significativa del moltiplicatore è pari a uno; successivamente il moltiplicando viene fatto scorrere verso sinistra di una posizione e lo si somma nuovamente al risultato se la cifra successiva del moltiplicatore è pari a uno; quindi si continua fino a che si esauriscono le cifre del moltiplicatore.

Nel programma mostrato, durante il ciclo di somme, il moltiplicatore viene fatto scorrere verso destra, in modo da poter verificare il valore della cifra espulsa attraverso l'indicatore di riporto (*carry*). Così facendo, quando il moltiplicatore è pari a zero, il ciclo di somme può terminare.

```
# op1 * op2
#
.section .data
op1:    .byte   0x07, 0x00 # little endian = 0x0007
                           # intero senza segno.
op2:    .byte   0x03, 0x00 # little endian = 0x0003
                           # intero senza segno.
#
.section .text
.globl _start
#
_start:
    movzx op1,  %edx       # Moltiplicando.
    movzx op2,  %ecx       # Moltiplicatore.
    mov   $0,   %eax       # Risultato.
do_mol:
    cmp   $0,   %ecx       # Se il moltiplicatore è pari a
    jz    end_do_mol       # zero, il ciclo deve terminare.
    shr   $1,   %ecx       # Fa scorrere a destra il
    jnc   end_do_mol_somma # moltiplicatore e se
                           # l'indicatore di riporto non è
                           # attivo, salta la somma.
do_mol_somma:
    add   %edx, %eax       # Aggiunge il moltiplicando al
                           # risultato.
end_do_mol_somma:
    shl   $1,   %edx       # Fa scorrere il moltiplicando
                           # a sinistra.
    jmp   do_mol           # Torna all'inizio del ciclo.
end_do_mol:
    mov   %eax, %ebx       # Restituisce il valore del
    mov   $1,   %eax       # prodotto, ammesso che sia
    int   $0x80            # abbastanza piccolo da poter
                           # essere rappresentato come
                           # valore di uscita.
```

```
; op1 * op2
;
section .data
op1:    dw      0x0007     ; Intero senza segno.
op2:    dw      0x0003     ; Intero senza segno.
;
section .text
global _start
;
_start:
    movzx edx, word [op1]  ; Moltiplicando.
    movzx ecx, word [op2]  ; Moltiplicatore.
    mov   eax,      0      ; Risultato.
do_mol:
    cmp   ecx,      0      ; Se il moltiplicatore è pari a
    jz    end_do_mol       ; zero, il ciclo deve terminare.
    shr   ecx,      1      ; Fa scorrere a destra il
    jnc   end_do_mol_somma ; moltiplicatore e se
                           ; l'indicatore di riporto non è
                           ; attivo, salta la somma.
do_mol_somma:
    add   eax,      edx    ; Aggiunge il moltiplicando al
                           ; risultato.
end_do_mol_somma:
    shl   edx,      1      ; Fa scorrere il moltiplicando a
                           ; sinistra.
    jmp   do_mol           ; Torna all'inizio del ciclo.
end_do_mol:
    mov   ebx,      eax    ; Restituisce il valore del
    mov   eax,      1      ; prodotto, ammesso che sia
    int   0x80             ; abbastanza piccolo da poter
                           ; essere rappresentato come
                           ; valore di uscita.
```

Come si può vedere, moltiplicando e moltiplicatore sono variabili da 16 bit, in modo da avere la certezza che il risultato del prodotto sia contenibile in un registro. Nel caso del primo listato, fatto per GNU AS, lo spazio in memoria viene dichiarato come sequenza di due byte, perché manca la possibilità di definire esplicitamente un

intero «corto».

64.9.6 Conteggio dei bit a uno

Viene proposto un esempio di programma che conta quanti bit a uno compongono un certo valore numerico. Per farlo, si usa lo scorrimento (in questo caso è a destra, ma non farebbe differenza) e quindi viene contato il riporto (l'indicatore di riporto è attivo se fuoriesce una cifra a uno).

```
#
.section .data
op1:      .int 44
#
.section .text
.globl _start
#
_start:
    mov   op1, %eax    # EAX contiene il numero di cui
                       # contare i bit a 1.
    mov   $0, %ecx     # ECX è il contatore dei bit a uno.
do_conta:
    cmp   $0, %eax     # Se EAX è a zero, il conteggio dei
    jz    end_do_conta # bit si conclude.
    shr   $1, %eax     # Fa scorrere a destra EAX.
    adc   $0, %ecx     # ECX = ECX + 0 + carry.
    jmp   do_conta     # Riprende il ciclo.
end_do_conta:
    mov   %ecx, %ebx   # Restituisce la quantità di bit.
    mov   $1, %eax     #
    int   $0x80        #
```

```
;
section .data
op1:      dd 44
;
section .text
global _start
;
_start:
    mov   eax, [op1]   ; EAX contiene il numero di cui
                       ; contare i bit a 1.
    mov   ecx, 0       ; ECX è il contatore dei bit a uno.
do_conta:
    cmp   eax, 0       ; Se EAX è a zero, il conteggio dei
    jz    end_do_conta ; bit si conclude.
    shr   eax, 1       ; Fa scorrere a destra EAX.
    adc   ecx, 0       ; ECX = ECX + 0 + carry.
    jmp   do_conta     ; Riprende il ciclo.
end_do_conta:
    mov   ebx, ecx     ; Restituisce la quantità di bit.
    mov   eax, 1       ;
    int   0x80         ;
```

Viene proposto un metodo alternativo che utilizza la sottrazione e l'abbinamento con l'operatore logico AND (è descritto nella sezione 63.5.10):

```
#
.section .data
op1:      .int 44
#
.section .text
.globl _start
#
_start:
    mov   op1, %eax     # EAX contiene il numero di cui
                        # contare i bit a 1.
    mov   $0, %ecx      # ECX è il contatore dei bit a uno.
do_conta:
    cmp   $0, %eax      # Se EAX è a zero, il conteggio
    jz    end_do_conta  # dei bit si conclude.
    mov   %eax, %edx    # Fa una copia in EDX.
    dec   %eax          # Decrementa EAX di una unità.
    and   %edx, %eax    # EAX := EAX AND EDX.
    inc   %ecx          # ECX++
    jmp   do_conta      # Riprende il ciclo.
end_do_conta:
    mov   %ecx, %ebx    # Restituisce la quantità di bit.
    mov   $1, %eax      #
    int   $0x80         #
```

```
;
section .data
op1:      dd 44
;
section .text
global _start
;
_start:
    mov   eax, [op1]    ; EAX contiene il numero di cui
                        ; contare i bit a 1.
    mov   ecx, 0        ; ECX è il contatore dei bit a uno.
do_conta:
    cmp   eax, 0        ; Se EAX è a zero, il conteggio dei
    jz    end_do_conta  ; bit si conclude.
    mov   edx, eax      ; Fa una copia in EDX.
    dec   eax           ; Decrementa EAX di una unità.
    and   eax, edx      ; EAX := EAX AND EDX.
    inc   ecx           ; ECX++
    jmp   do_conta      ; Riprende il ciclo.
end_do_conta:
    mov   ebx, ecx      ; Restituisce la quantità di bit.
    mov   eax, 1        ;
    int   0x80          ;
```


64.10 Funzioni

Attraverso l'uso delle istruzioni '**CALL**' e '**RET**' è possibile realizzare delle subroutine, ovvero qualcosa che assomigli alle funzioni dei linguaggi di programmazione più evoluti.

```
CALL indirizzo
```

```
RET
```

L'istruzione '**CALL**', prima di passare il controllo all'indirizzo di memoria indicato,[9] salva l'indirizzo dell'istruzione successiva alla chiamata nella pila dei dati (*stack*). Per converso, l'istruzione '**RET**' recupera dalla pila l'ultimo elemento e passa il controllo all'istruzione che si trova all'indirizzo contenuto in tale elemento estratto.

64.10.1 Esempio banale di chiamata

Per comprendere il meccanismo con cui si realizzano le subroutine, si prende in considerazione un esempio, già descritto, che calcola il prodotto tra due numeri attraverso lo scorrimento dei valori. Come sempre, vengono mostrati due listati equivalenti, fatti rispettivamente per GNU AS e NASM.

```
1  # op1 * op2
2  #
3  .section .data
4  op1:    .byte   0x07, 0x00    # little endian = 0x0005
5                                # intero senza segno.
6  op2:    .byte   0x03, 0x00    # little endian = 0x0003
7                                # intero senza segno.
8  #
9  .section .text
10 .globl _start
11 #-----------------------------------------------------
12 _start:
13     movzx op1,  %edx       # Moltiplicando.
14     movzx op2,  %ecx       # Moltiplicatore.
15     mov   $0,   %eax       # Risultato.
16 bp1:
17     call  f_mol            # Esegue la moltiplicazione:
18                            # EAX = EDX * ECX
19 bp2:
20     mov   %eax, %ebx       # Restituisce il valore del
21     mov   $1,   %eax       # prodotto, ammesso che sia
22     int   $0x80            # abbastanza piccolo da poter
23                            # essere rappresentato come
24                            # valore di uscita.
25 #-----------------------------------------------------
26 # Moltiplicazione di due numeri interi.
27 # EAX = EDX * ECX
28 # I registri EDX e ECX vengono alterati durante il
29 # procedimento.
```

```
#
f_mol:
    cmp   $0,   %ecx       # Se il moltiplicatore è
    jz    f_end_mol        # pari a zero, il ciclo
                           # deve terminare.
    shr   $1,   %ecx       # Fa scorrere a destra il
    jnc   end_do_somma     # moltiplicatore e se
                           # l'indicatore di riporto
                           # non è attivo, salta
                           # la somma.
do_somma:
    add   %edx, %eax       # Aggiunge il moltiplicando
                           # al risultato.
end_do_somma:
    shl   $1,   %edx       # Fa scorrere il
                           # moltiplicando a sinistra.
    jmp   f_mol            # Torna all'inizio della
                           # funzione.
f_end_mol:
    ret                    # Torna all'istruzione
                           # successiva alla chiamata.
```

```
1  ; op1 * op2
2  ;
3  section .data
4  op1:    dw      0x0007    ; Intero senza segno.
5  op2:    dw      0x0003    ; Intero senza segno.
6  ;
7  section .text
8  global _start
9  ;-------------------------------------------------------
10 _start:
11     movzx edx, word [op1]  ; Moltiplicando.
12     movzx ecx, word [op2]  ; Moltiplicatore.
13     mov   eax,      0      ; Risultato.
14 bp1:
15     call  f_mol            ; Esegue la moltiplicazione:
16                            ; EAX = EDX * ECX
17 bp2:
18     mov   ebx,      eax    ; Restituisce il valore
19     mov   eax,      1      ; del prodotto, ammesso che
20     int   0x80             ; sia abbastanza piccolo da
21                            ; poter essere rappresentato
22                            ; come valore di uscita.
23 ;-------------------------------------------------------
24 ; Moltiplicazione di due numeri interi.
25 ; EAX = EDX * ECX
26 ; I registri EDX e ECX vengono alterati durante il
27 ; procedimento.
28 ;
29 f_mol:
30     cmp   ecx,      0      ; Se il moltiplicatore è
31     jz    f_end_mol        ; pari a zero, il ciclo deve
32                            ; terminare.
33     shr   ecx,      1      ; Fa scorrere a destra il
34     jnc   end_do_somma     ; moltiplicatore e se
35                            ; l'indicatore di riporto
36                            ; non è attivo, salta
37                            ; la somma.
38 do_somma:
39     add   eax,      edx    ; Aggiunge il moltiplicando
40                            ; al risultato.
41 end_do_somma:
42     shl   edx,      1      ; Fa scorrere il
43                            ; moltiplicando a sinistra.
44     jmp   f_mol            ; Torna all'inizio della
45                            ; funzione.
46 f_end_mol:
47     ret                    ; Torna all'istruzione
48                            ; successiva alla chiamata.
```

Si osservi che il programma viene eseguito a partire dalla riga 11 e che si conclude alla riga 19. La subroutine, ovvero la funzione che esegue la moltiplicazione, si trova in un gruppo di istruzioni successive, il cui inizio è segnalato dal simbolo '**f_mol**'.

La funzione riceve il moltiplicando e il moltiplicatore attraverso due registri, utilizzando a sua volta un altro registro per restituire il risultato.

64.10.2 Salvataggio dei registri prima della chiamata

Ogni funzione ha la necessità di elaborare dati senza interferire con il resto del programma; in pratica, ogni funzione deve poter utilizzare i registri con una certa libertà. Nell'esempio precedente vengono utilizzati dei registri per passare alla funzione i valori da moltiplicare, ma all'interno della funzione il contenuto dei registri viene modificato. Per lo scopo dell'esempio, il fatto che i registri ***ECX*** e ***EDX*** vengano modificati, non produce effetti collaterali, ma in un programma più complesso potrebbe essere il caso di salvaguardare il contenuto originale dei registri prima della chiamata di una funzione. L'esempio successivo mostra una variante del codice contenuto tra i simboli '**bp1**' e '**bp2**', allo scopo di conservare una copia dei registri che contengono il moltiplicando e il moltiplicatore, per ripristinarla dopo la chiamata:

```
bp1:
    push  %ecx  # Salva il moltiplicatore nella pila.
    push  %edx  # Salva il moltiplicando nella pila.
    call  f_mol # Esegue la moltiplicazione: EAX = EDX * ECX
    pop   %edx  # Recupera il moltiplicando dalla pila.
    pop   %ecx  # Recupera il moltiplicatore dalla pila.
bp2:
```

```
bp1:
    push  ecx   ; Salva il moltiplicatore nella pila.
    push  edx   ; Salva il moltiplicando nella pila.
    call  f_mol ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop   edx   ; Recupera il moltiplicando dalla pila.
    pop   ecx   ; Recupera il moltiplicatore dalla pila.
bp2:
```

Come si può intendere, il recupero dei valori dalla pila deve avvenire in senso inverso.

64.10.3 Passaggio di parametri attraverso la pila

Per rendere più libera la funzione dal programma chiamante, conviene utilizzare la stessa pila per il passaggio dei parametri. In pratica, dopo avere salvato i registri che contengono dati importanti (ammesso che ciò vada fatto), occorre accumulare nella pila gli argomenti della chiamata della funzione, secondo un ordine convenuto per la funzione stessa. All'interno della funzione, poi, si vanno a pescare questi valori per usarli nell'elaborazione.

```
# op1 * op2
#
.section .data
op1:    .byte   0x07, 0x00 # little endian = 0x0005
                           # intero senza segno.
op2:    .byte   0x03, 0x00 # little endian = 0x0003
                           # intero senza segno.
#
.section .text
.globl _start
#-------------------------------------------------------
_start:
    movzx op1,  %edx # Moltiplicando.
    movzx op2,  %ecx # Moltiplicatore.
bp1:
    push  %ecx       # Salva il moltiplicatore nella pila.
    push  %edx       # Salva il moltiplicando nella pila.
    #
    push  %ecx       # Inserisce il secondo parametro nella
                     # pila.
    push  %edx       # Inserisce il primo parametro nella
                     # pila.
    call  f_mol      # Esegue la moltiplicazione:
                     # EAX = EDX * ECX
    add   $4,   %esp # Espelle il primo parametro della
                     # chiamata.
    add   $4,   %esp # Espelle il secondo parametro della
                     # chiamata.
    #
    pop   %edx       # Recupera il moltiplicando dalla pila.
    pop   %ecx       # Recupera il moltiplicatore dalla
                     # pila.
bp2:
```

```
    mov   %eax, %ebx # Restituisce il valore del prodotto,
    mov   $1,   %eax # ammesso che sia abbastanza piccolo
    int   $0x80      # da poter essere rappresentato come
                     # valore di uscita.
#-------------------------------------------------------------
# Moltiplicazione di due numeri interi.
# f_mol (a, b) => EAX
# EAX = a * b
#
f_mol:
    mov   4(%esp), %edx # Copia il primo parametro in EDX.
    mov   8(%esp), %ecx # Copia il secondo parametro in ECX.
    mov   $0,      %eax # Azzera EAX per sicurezza.
do_mol:
    cmp   $0,      %ecx # Se il moltiplicatore è pari a
    jz    end_do_mol    # zero, il ciclo deve terminare.
    shr   $1,      %ecx # Fa scorrere a destra il
    jnc   end_do_somma  # moltiplicatore e se l'indicatore
                        # di riporto non è attivo, salta
                        # la somma.
do_somma:
    add   %edx,    %eax # Aggiunge il moltiplicando al
                        # risultato.
end_do_somma:
    shl   $1,      %edx # Fa scorrere il moltiplicando a
                        # sinistra.
    jmp   do_mol        # Torna all'inizio del ciclo di
                        # moltiplicazione.
end_do_mol:
    ret                 # Torna all'istruzione successiva
                        # alla chiamata.
```

```
; op1 * op2
;
section .data
op1:    dw      0x0007      ; Intero senza segno.
op2:    dw      0x0003      ; Intero senza segno.
;
section .text
global _start
;-------------------------------------------------------------
_start:
    movzx edx, word [op1]   ; Moltiplicando.
    movzx ecx, word [op2]   ; Moltiplicatore.
bp1:
    push  ecx       ; Salva il moltiplicatore nella pila.
    push  edx       ; Salva il moltiplicando nella pila.
    ;
    push  ecx       ; Inserisce il secondo parametro nella
                    ; pila.
    push  edx       ; Inserisce il primo parametro nella
                    ; pila.
    call  f_mol     ; Esegue la moltiplicazione:
                    ; EAX = EDX * ECX
    add   esp,  4   ; Espelle il primo parametro della
                    ; chiamata.
    add   esp,  4   ; Espelle il secondo parametro della
                    ; chiamata.
    ;
    pop   edx       ; Recupera il moltiplicando dalla pila.
    pop   ecx       ; Recupera il moltiplicatore dalla pila.
bp2:
    mov   ebx,  eax ; Restituisce il valore del prodotto,
    mov   eax,  1   ; ammesso che sia abbastanza piccolo
    int   0x80      ; da poter essere rappresentato come
                    ; valore di uscita.
;-------------------------------------------------------------
; Moltiplicazione di due numeri interi.
; f_mol (a, b) => EAX
; EAX = a * b
;
f_mol:
    mov   edx,  [esp+4] ; Copia il primo parametro in EDX.
    mov   ecx,  [esp+8] ; Copia il secondo parametro in ECX.
    mov   eax,  0       ; Azzera EAX per sicurezza.
    ;
do_mol:
    cmp   ecx,  0      ; Se il moltiplicatore è pari a zero,
    jz    end_do_mol   ; il ciclo deve terminare.
    shr   ecx,  1      ; Fa scorrere a destra il
    jnc   end_do_somma ; moltiplicatore e se l'indicatore di
```



```
                       ; riporto non è attivo, salta
                       ; la somma.
do_somma:
    add   eax,  edx    ; Aggiunge il moltiplicando al
                       ; risultato.
end_do_somma:
    shl   edx,  1      ; Fa scorrere il moltiplicando a
                       ; sinistra.
    jmp   do_mol       ; Torna all'inizio del ciclo di
                       ; moltiplicazione.
end_do_mol:
    ret                ; Torna all'istruzione successiva
                       ; alla chiamata.
```

Come si vede, rispetto alla versione precedente dello stesso programma, il risultato del prodotto, calcolato all'interno della funzione, continua a essere restituito attraverso il registro ***EAX***, ma sarebbe stato possibile accumulare nella pila, prima della chiamata, un valore in più, da considerare poi come il risultato generato dalla funzione.

All'inizio della funzione vengono recuperati i valori che costituiscono gli argomenti della chiamata. Tale operazione viene eseguita attraverso queste istruzioni:

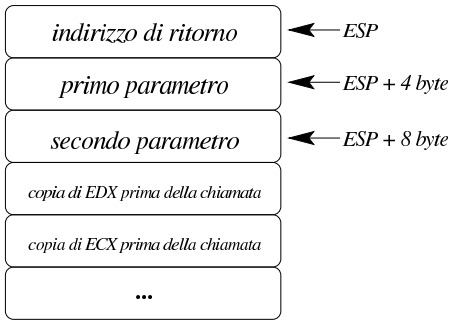
```
    mov 4(%esp), %edx # Copia il primo parametro in EDX.
    mov 8(%esp), %ecx # Copia il secondo parametro in ECX.
```

```
    mov edx, [esp+4] ; Copia il primo parametro in EDX.
    mov ecx, [esp+8] ; Copia il secondo parametro in ECX.
```

L'operando '`4(%esp)`', ovvero '`[esp+4]`', individua l'indirizzo di memoria corrispondente al valore del registro ***ESP***, più quattro byte.

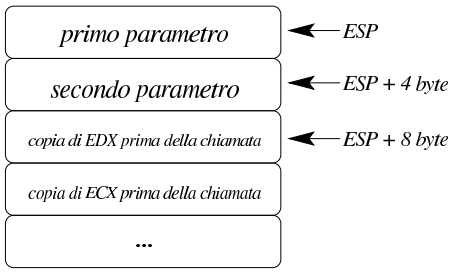
Il registro ***ESP*** è l'indice della pila (*stack pointer*) che punta all'ultimo elemento presente (quello in cima alla pila). Nei sistemi Unix (compresi i sistemi GNU) la pila parte da una posizione elevata della memoria e «cresce» utilizzando indirizzi che invece decrescono. Pertanto, considerato che l'ultimo elemento della pila è l'indirizzo di ritorno, l'elemento immediatamente precedente lo si raggiunge quattro byte dopo (32 bit) e quello ancora precedente si trova otto byte dopo la posizione finale.

Figura 64.169. Situazione della pila in corrispondenza del simbolo '`f_mol`'.



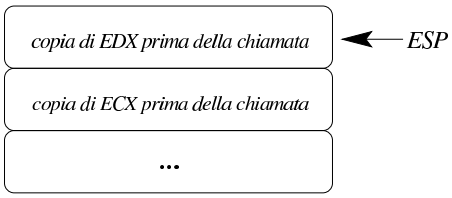
La funzione elabora il prodotto dei valori forniti come argomento e ne lascia il risultato nel registro ***EAX***. Al ritorno, la pila si presenta come si vede nella figura successiva:

Figura 64.170. Situazione della pila immediatamente dopo la chiamata della funzione.



Come si vede, occorre espellere dalla pila i parametri usati per la chiamata. Dal momento che non c'è bisogno di rileggere il loro valore, ci si limita a decrementare l'indice della pila, ovvero si incrementa il valore del registro ***ESP*** a gruppi di quattro byte alla volta.

Figura 64.171. Situazione della pila dopo l'espulsione dei parametri della chiamata.



Naturalmente, considerato che la funzione non altera i valori accumulati nella pila, la chiamata potrebbe essere semplificata un po':

```
bp1:
    push  %ecx  # Salva ECX, inserendolo come secondo
                # parametro nella pila.
    push  %edx  # Salva EDX, inserendolo come primo
                # parametro nella pila.
    call  f_mol # Esegue la moltiplicazione: EAX = EDX * ECX
    pop   %edx  # Recupera EDX dalla pila.
    pop   %ecx  # Recupera ECX dalla pila.
bp2:
```

```
bp1:
    push  ecx   ; Salva ECX, inserendolo come secondo
                ; parametro nella pila.
    push  edx   ; Salva EDX, inserendolo come primo
                ; parametro nella pila.
    call  f_mol ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop   edx   ; Recupera EDX dalla pila.
    pop   ecx   ; Recupera ECX dalla pila.
bp2:
```

64.10.4 Utilizzo del registro «EBP»

Perché una funzione possa gestire delle variabili «locali», ovvero tali da avere un campo di azione limitato alla funzione stessa, senza lasciare tracce al ritorno dalla chiamata, si deve usare la pila aggiungendovi altri elementi. Questo fatto complica l'accesso ai parametri della chiamata, perché durante l'esecuzione delle istruzioni della funzione, l'indice della pila può spostarsi. A questo proposito, all'inizio di una funzione, conviene conservare una copia del registro ***ESP*** in un altro registro apposito: ***EBP*** (*base pointer*). In pratica, l'indice contenuto in ***EBP*** dovrebbe essere sempre usato per rappresentare la posizione in cui si trova l'indirizzo di ritorno della funzione in cui ci si trova.

Viene riproposto il programma già presentato nella sezione precedente, con le semplificazioni già descritte a proposito della chiamata e con le modifiche relative all'uso del registro ***EBP***.

```
# op1 * op2
#
.section .data
op1:    .byte   0x07, 0x00 # little endian = 0x0005
                           # intero senza segno.
op2:    .byte   0x03, 0x00 # little endian = 0x0003
                           # intero senza segno.
#
.section .text
.globl _start
#-------------------------------------------------------------
_start:
    movzx op1,  %edx # Moltiplicando.
    movzx op2,  %ecx # Moltiplicatore.
    mov   $0,   %eax # Risultato.
bp1:
    push  %ebp       # Salva il registro EBP
                     # prima della chiamata.
    push  %ecx       # Inserisce il moltiplicando nella
```



```
                     # pila.
    push  %edx       # Inserisce il moltiplicatore nella
                     # pila.
    call  f_mol      # Esegue la moltiplicazione:
                     # EAX = EDX * ECX
    pop   %edx       # Recupera il moltiplicando dalla pila.
    pop   %ecx       # Recupera il moltiplicatore dalla
                     # pila.
    pop   %ebp       # Recupera il registro EBP
                     # dopo la chiamata.
bp2:
    mov   %eax, %ebx # Restituisce il valore del prodotto,
    mov   $1,   %eax # ammesso che sia abbastanza piccolo
    int   $0x80      # da poter essere rappresentato come
                     # valore di uscita.
#-------------------------------------------------------------
# Moltiplicazione di due numeri interi.
# f_mol (a, b) => EAX
# EAX = a * b
#
f_mol:
    mov   %esp,  %ebp   # Copia ESP in EBP.
    mov   4(%ebp), %edx # Copia il primo parametro in EDX.
    mov   8(%ebp), %ecx # Copia il secondo parametro in ECX.
    mov   $0,    %eax   # Azzera EAX per sicurezza.
    #
do_mol:
    cmp   $0,    %ecx   # Se il moltiplicatore è pari a
    jz    end_do_mol    # zero, il ciclo deve terminare.
    shr   $1,    %ecx   # Fa scorrere a destra il
    jnc   end_do_somma  # moltiplicatore e se l'indicatore
                        # di riporto non è attivo, salta la
                        # somma.
do_somma:
    add   %edx,  %eax   # Aggiunge il moltiplicando al
                        # risultato.
end_do_somma:
    shl   $1,    %edx   # Fa scorrere il moltiplicando a
                        # sinistra.
    jmp   do_mol        # Torna all'inizio del ciclo di
                        # moltiplicazione.
end_do_mol:
    mov %ebp,    %esp   # Ripristina ESP, espellendo
                        # le variabili locali.
    ret                 # Torna all'istruzione successiva
                        # alla chiamata.
```

```
; op1 * op2
;
section .data
op1:    dw      0x0007      ; Intero senza segno.
op2:    dw      0x0003      ; Intero senza segno.
;
section .text
global _start
;-------------------------------------------------------------
_start:
    movzx edx, word [op1]   ; Moltiplicando.
    movzx ecx, word [op2]   ; Moltiplicatore.
bp1:
    push  ebp     ; Salva il registro EBP prima
                  ; della chiamata.
    push  ecx     ; Inserisce il moltiplicando nella pila.
    push  edx     ; Inserisce il moltiplicatore nella pila.
    call  f_mol   ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop   edx     ; Recupera il moltiplicando dalla pila.
    pop   ecx     ; Recupera il moltiplicatore dalla pila.
    pop   ebp     ; Recupera il registro EBP dopo
                  ; la chiamata.
bp2:
    mov   ebx,   eax ; Restituisce il valore del prodotto,
    mov   eax,   1   ; ammesso che sia abbastanza piccolo
    int   0x80       ; da poter essere rappresentato come
                     ; valore di uscita.
;-------------------------------------------------------------
; Moltiplicazione di due numeri interi.
; f_mol (a, b) => EAX
; EAX = a * b
;
f_mol:
    mov   ebp,   esp     ; Copia ESP in EBP.
```

```
    mov   edx,   [ebp+4] ; Copia il primo parametro in EDX.
    mov   ecx,   [ebp+8] ; Copia il secondo parametro
                         ; in ECX.
    mov   eax,   0       ; Azzera EAX per sicurezza.
    ;
do_mol:
    cmp   ecx,   0       ; Se il moltiplicatore è pari a
    jz    end_do_mol     ; zero, il ciclo deve terminare.
    shr   ecx,   1       ; Fa scorrere a destra il
    jnc   end_do_somma   ; moltiplicatore e se l'indicatore
                         ; di riporto non è attivo, salta
                         ; la somma.
do_somma:
    add   eax,   edx     ; Aggiunge il moltiplicando al
                         ; risultato.
end_do_somma:
    shl   edx,   1       ; Fa scorrere il moltiplicando a
                         ; sinistra.
    jmp   do_mol         ; Torna all'inizio del ciclo di
                         ; moltiplicazione.
end_do_mol:
    mov   esp,   ebp     ; Ripristina ESP, espellendo
                         ; le variabili locali.
    ret                  ; Torna all'istruzione successiva
                         ; alla chiamata.
```

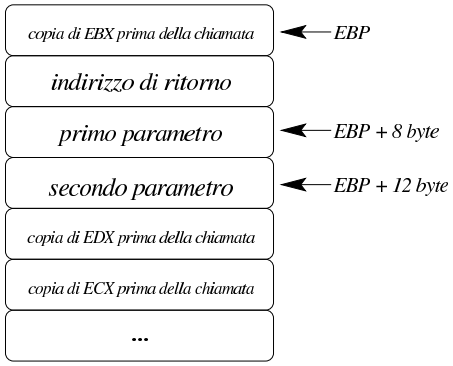
Nei listati appena mostrati si vede che ***EBP*** viene salvato prima della chiamata e ripristinato successivamente. Esiste però un altro modo, più diffuso, per cui il registro ***EBP*** va salvato nella pila all'inizio della funzione, con le conseguenze che ciò comporta:

```
…
bp1:
    push  %ecx  # Inserisce il moltiplicando nella pila.
    push  %edx  # Inserisce il moltiplicatore nella pila.
    call  f_mol # Esegue la moltiplicazione: EAX = EDX * ECX
    pop   %edx  # Recupera il moltiplicando dalla pila.
    pop   %ecx  # Recupera il moltiplicatore dalla pila.
bp2:
…
f_mol:
    push   %ebp           # Salva il registro EBP.
bp3:
    mov   %esp,     %ebp  # Copia ESP in EBP.
    mov   8(%ebp),  %edx  # Copia il primo parametro in EDX.
    mov   12(%ebp), %ecx  # Copia il secondo parametro
                          # in ECX.
    mov   $0,       %eax  # Azzera EAX per sicurezza.
…
end_do_mol:
    mov   %ebp,     %esp  # Ripristina ESP, espellendo le
                          # variabili locali.
    pop   %ebp            # Riporta il registro EBP allo
                          # stato precedente.
    ret                   # Torna all'istruzione successiva
                          # alla chiamata.
```

```
…
bp1:
    push  ecx   ; Inserisce il moltiplicando nella pila.
    push  edx   ; Inserisce il moltiplicatore nella pila.
    call  f_mol ; Esegue la moltiplicazione: EAX = EDX * ECX
    pop   edx   ; Recupera il moltiplicando dalla pila.
    pop   ecx   ; Recupera il moltiplicatore dalla pila.
bp2:
…
f_mol:
    push  ebp               ; Salva il registro EBP.
bp3:
    mov   ebp,      esp     ; Copia ESP in EBP.
    mov   edx,      [ebp+8] ; Copia il primo parametro
                            ; in EDX.
    mov   ecx,      [ebp+12] ; Copia il secondo parametro
                            ; in ECX.
    mov   eax,      0       ; Azzera EAX per sicurezza.
…
end_do_mol:
    mov   esp,      ebp     ; Ripristina ESP, espellendo le
                            ; variabili locali.
    pop   ebp               ; Ripristina il registro EBP.
    ret                     ; Torna all'istruzione successiva
                            ; alla chiamata.
```

In tal caso, in corrispondenza del simbolo '**bp3**', la pila ha i contenuti che sono schematizzati nella figura successiva.

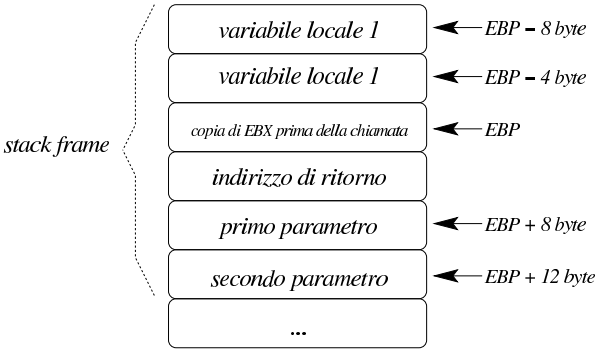
Figura 64.178. Situazione della pila in corrispondenza del simbolo '**bp3**'.



64.10.5 Allocazione dello spazio per le variabili locali e preservazione dei registri

Come accennato nella sezione precedente, una volta salvato il valore di ***EBP*** nella pila e assegnatovi il valore di ***ESP***, le variabili locali possono essere accumulate nella pila quando servono. Tuttavia, di solito si preferisce definire subito lo spazio utilizzato dalle variabili locali. Per esempio, supponendo di averne due, la pila potrebbe mostrarsi come nella figura successiva.

Figura 64.179. Situazione della pila, all'interno di una funzione con due variabili locali.



L'insieme degli elementi della pila, costituito dai parametri della funzione fino alle variabili locali, è noto come *stack frame*. Spesso, le «convenzioni di chiamata» prescrivono che siano le funzioni stesse a preservare lo stato precedente dei registri, pertanto, di solito, dopo la definizione dello spazio usato dalle variabili locali, si salvano nella pila anche tutti i registri principali, eventualmente con l'aiuto dell'istruzione '**PUSHA**'. I listati successivi mostrano una modifica ulteriore del programma già utilizzato nelle sezioni precedenti.

```
# op1 * op2
#
.section .data
op1:    .byte   0x07, 0x00    # little endian = 0x0005
                              # intero senza segno.
op2:    .byte   0x03, 0x00    # little endian = 0x0003
                              # intero senza segno.
#
.section .text
.globl _start
#-------------------------------------------------------------
_start:
    movzx op1,  %edx # Moltiplicando.
    movzx op2,  %ecx # Moltiplicatore.
    mov   $0,   %eax # Risultato.
```

```
bp1:
    push  %ecx       # Inserisce il moltiplicando nella
                     # pila.
    push  %edx       # Inserisce il moltiplicatore nella
                     # pila.
    call  f_mol      # Esegue la moltiplicazione:
                     # EAX = EDX * ECX
    add   $8, %esp   # Espelle i parametri
                     # di chiamata.
bp2:
    mov   %eax, %ebx # Restituisce il valore del prodotto,
    mov   $1,   %eax # ammesso che sia abbastanza piccolo
    int   $0x80      # da poter essere rappresentato come
                     # valore di uscita.
#------------------------------------------------------------
# Moltiplicazione di due numeri interi.
# f_mol (a, b) => EAX
# EAX = a * b
#
f_mol:
    push %ebp           # Salva il registro EBP.
    mov  %esp,    %ebp  # Copia ESP in EBP.
    sub  $4,       %esp # Crea lo spazio per
                        # una variabile locale.
    pusha               # Salva i registri
                        # principali.
    #
    mov  8(%ebp), %edx  # Copia il primo parametro in EDX.
    mov  12(%ebp), %ecx # Copia il secondo parametro in ECX.
    mov  $0,      %eax  # Azzera EAX per sicurezza.
    #
do_mol:
    cmp  $0,      %ecx  # Se il moltiplicatore è pari a
    jz   end_do_mol     # zero, il ciclo deve terminare.
    shr  $1,      %ecx  # Fa scorrere a destra il
    jnc  end_do_somma   # moltiplicatore e se l'indicatore
                        # di riporto non è attivo, salta
                        # la somma.
do_somma:
    add  %edx,    %eax  # Aggiunge il moltiplicando al
                        # risultato.
end_do_somma:
    shl  $1,      %edx  # Fa scorrere il moltiplicando a
                        # sinistra.
    jmp  do_mol         # Torna all'inizio del ciclo di
                        # moltiplicazione.
end_do_mol:
    mov  %eax, -4(%ebp) # Copia EAX nella variabile
                        # locale prevista.
    popa                # Ripristina i registri principali.
    mov -4(%ebp), %eax  # Rimette a posto il valore di EAX
                        # che deve essere restituito.
    mov   %ebp,    %esp # Ripristina ESP, espellendo le
                        # variabili locali.
    pop   %ebp          # Riporta il registro EBP allo
                        # stato precedente.
    ret                 # Torna all'istruzione successiva
                        # alla chiamata.
```

```
; op1 * op2
;
section .data
op1:    dw      0x0007      ; Intero senza segno.
op2:    dw      0x0003      ; Intero senza segno.
;
section .text
global _start
;------------------------------------------------------------
_start:
    movzx edx, word [op1]   ; Moltiplicando.
    movzx ecx, word [op2]   ; Moltiplicatore.
bp1:
    push  ecx               ; Inserisce il moltiplicando
                            ; nella pila.
    push  edx               ; Inserisce il moltiplicatore
                            ; nella pila.
    call  f_mol             ; Esegue la moltiplicazione:
                            ; EAX = EDX * ECX
    add esp,        8       ; Espelle i parametri
                            ; della chiamata.
bp2:
    mov   ebx,      eax     ; Restituisce il valore del
    mov   eax,      1       ; prodotto, ammesso che sia
    int   0x80              ; abbastanza piccolo da poter
                            ; essere rappresentato come
                            ; valore di uscita.
;------------------------------------------------------------
; Moltiplicazione di due numeri interi.
; f_mol (a, b) => EAX
; EAX = a * b
;
f_mol:
    push  ebp               ; Salva il registro EBP.
    mov   ebp,      esp     ; Copia ESP in EBP.
    sub   esp,      4       ; Crea lo spazio per una
                            ; variabile locale.
    pusha                   ; Salva i registri principali.
    ;
    mov   edx,      [ebp+8]  ; Copia il primo parametro
                             ; in EDX.
    mov   ecx,      [ebp+12] ; Copia il secondo parametro in
                             ; ECX.
    mov   eax,      0       ; Azzera EAX per sicurezza.
    ;
do_mol:
    cmp   ecx,      0       ; Se il moltiplicatore è pari
    jz    end_do_mol        ; a zero, il ciclo deve
                            ; terminare.
    shr   ecx,      1       ; Fa scorrere a destra il
    jnc   end_do_somma      ; moltiplicatore e se
                            ; l'indicatore di riporto non
                            ; è attivo, salta la somma.
do_somma:
    add   eax,      edx     ; Aggiunge il moltiplicando al
                            ; risultato.
end_do_somma:
    shl   edx,      1       ; Fa scorrere il moltiplicando
                            ; a sinistra.
    jmp   do_mol            ; Torna all'inizio del ciclo
                            ; di moltiplicazione.
end_do_mol:
    mov   [ebp-4],  eax     ; Copia EAX nella variabile
                            ; locale prevista.
    popa                    ; Ripristina i registri
                            ; principali.
    mov   eax,      [ebp-4] ; Rimette a posto il valore
                            ; di EAX che deve essere
                            ; restituito.
    mov   esp,      ebp     ; Ripristina ESP, espellendo
                            ; le variabili locali.
    pop   ebp               ; Ripristina il registro EBP.
    ret                     ; Torna all'istruzione
                            ; successiva alla chiamata.
```


Come si vede, avendo usato la coppia di istruzioni '**PUSHA**', '**POPA**', alla fine occorre prendersi cura del risultato che è già disponibile nel registro ***EAX***: infatti viene salvato prima nello spazio riservato per la variabile locale, quindi vengono ripristinati tutti i registri (tutti eccetto '**ESP**') e ancora viene ripristinato ***EAX***, che deve trasmettere il valore alla chiamata.

A questo punto occorre sapere che le istruzioni seguenti possono essere sostituite dall'istruzione '**ENTER**', dove ***n*** rappresenta una quantità di byte:

```
    push  %ebp
    mov   %esp, %ebp
    sub   $n, %esp
```

```
    push  ebp
    mov   ebp, esp
    sub   esp, n
```

Per la precisione, il rimpiazzo avviene come nei due brani seguenti: si osservi che in questo caso, gli attributi di '**ENTER**' non vengono invertiti nelle due sintassi.

```
    enter $n, $0
```

```
    enter n, 0
```

Le istruzioni che invece va a rimpiazzare '**LEAVE**' sono quelle seguenti:

```
    mov   %ebp, %esp
    pop   %ebp
```

```
    mov   esp, ebp
    pop   ebp
```

Logicamente, '**LEAVE**' non richiede operandi, quindi si usa nello stesso modo nelle due sintassi:

```
    leave
```

64.10.6 Convenzioni di chiamata

Da quanto descritto si comprende che si possono usare diversi modi per chiamare una funzione, ma anche se esistono delle modalità equivalenti, occorre definire una convenzione. In generale, per chi scrive programmi in un sistema compatibile con la tradizione Unix, la cosa migliore è uniformarsi alle convenzioni di chiamata del linguaggio C (precisamente servono quelle usate dal proprio compilatore), in modo da poter mettere assieme programmi scritti in parte in linguaggio assemblatore e in parte anche in C. Di solito, le regole per chi scrive funzioni in linguaggio assemblatore sono sostanzialmente quelle dell'ultimo esempio mostrato nella sezione precedente:

- i parametri vanno messi sulla pila in ordine inverso, in modo tale che prima della chiamata appaia in cima il primo parametro;
- all'inizio della funzione, occorre accumulare nella pila il contenuto di ***EBP***, che deve essere ripristinato immediatamente prima del ritorno;
- all'interno della funzione si accede ai parametri contenuti nella pila, senza estrarli dalla stessa e senza modificarli, perché le modifiche non verrebbero considerate;
- i registri principali devono essere preservati (ogni compilatore ha la sua politica e non si può dire, in generale, quali siano);
- la funzione deve restituire il risultato della sua elaborazione attraverso ***EAX***, oppure, se si richiede una dimensione più grande, deve essere usata la coppia ***EDX***:***EAX***.[10]

Nel caso si vogliano utilizzare funzioni scritte in linguaggio C, all'interno di un programma scritto in linguaggio assemblatore, occorre verificare quali registri le funzioni scritte in C non preservano (oltre alla coppia ***EDX***:***EAX***, usata per restituire il risultato della chiamata). In generale, si può considerare che le funzioni scritte in linguaggio C potrebbero alterare i registri ***EAX***, ***ECX*** e ***EDX***. Se però si vuole avere la certezza assoluta sul contenuto dei registri dopo la chiamata di una funzione realizzata con un altro linguaggio, conviene organizzarsi salvando tutti quelli che si stanno utilizzando prima della chiamata e ripristinandoli subito dopo, come in parte è stato mostrato.

64.10.7 Nota sugli array «locali»

Generalmente, un array viene gestito attraverso uno spazio di memoria condiviso da tutto il programma, dove le funzioni che devono manipolarlo ricevono l'indirizzo di questo, tra i parametri della chiamata. Tuttavia, nel caso si volesse gestire, all'interno di una funzione, un array locale, il cui contenuto viene abbandonato alla conclusione della stessa, l'unico modo per ottenere ciò è attraverso la pila dei dati. In pratica, come per le variabili locali scalari, andrebbe riservato un certo spazio aumentando la dimensione della pila in modo adeguato, per poi scandire tale spazio con indici appropriati.

64.11 Esempi di funzioni ricorsive

Vengono mostrati alcuni esempi molto semplici in cui si applica la ricorsione, adatti alla compilazione con GNU AS e NASM.

64.11.1 Elevamento a potenza

Viene proposta una soluzione ricorsiva del problema dell'elevamento a potenza. In pratica, x^y è pari a $x \cdot x^{(y-1)}$, ma in particolare: se x è pari a zero, il risultato è zero; altrimenti, se y è pari a zero, il risultato è uno.

All'interno della funzione '**f_pwr**' viene riservato lo spazio per una sola variabile locale, che serve a conservare il valore di ***EAX***, per poi recuperarlo quando si ripristinano tutti i registri, prima della conclusione della funzione stessa.

```
# op1 ^ op2
#
.section .data
op1:    .int    3
op2:    .int    4
#
.section .text
.globl _start
#
# Main.
#
_start:
    mov   op1, %esi   # Base.
    mov   op2, %edi   # Esponente.
bp1:
    push  %edi        # f_pwr (ESI, EDI) ==> EAX
    push  %esi        #
    call  f_pwr       #
    add   $8, %esp    #
bp2:
    mov   %eax, %ebx  # Restituisce il valore della potenza,
    mov   $1, %eax    # ammesso che sia abbastanza piccolo
    int   $0x80       # da poter essere rappresentato come
                      # valore di uscita.
#
# Potenza di due numeri interi senza segno.
# f_pwr (a, b) ==> EAX
# EAX = a ^ b
#
f_pwr:
    enter $4, $0
    pusha
    #
    mov   8(%ebp), %esi  # Base.
    mov   12(%ebp), %edi # Esponente.
    #
    cmp   $0, %esi       # Se la base è pari a 0,
    jz    f_pwr_end_0    # restituisce 0.
    #
    cmp   $0, %edi       # Se l'esponente è pari a 0,
    jz    f_pwr_end_1    # restituisce 1.
    #
    dec   %edi           # Riduce l'esponente di una unità.
    push  %edi           # f_pwr (ESI, EDI) ==> EAX
    push  %esi           #
    call  f_pwr          #
    add   $8, %esp       #
    mul   %esi           # EDX:EAX = EAX*ESI
    mov   %eax, -4(%ebp) # Salva il risultato.
    jmp   f_pwr_end_X    # Conclude la funzione.
    #
f_pwr_end_0:
    popa                 # Conclude la funzione con EAX = 0.
    mov $0, %eax         #
    leave                #
    ret                  #
f_pwr_end_1:
    popa                 # Conclude la funzione con EAX = 1.
    mov $1, %eax         #
    leave                #
    ret                  #
f_pwr_end_X:
    popa                 # Conclude la funzione con EAX pari
    mov -4(%ebp), %eax   # al valore salvato nella variabile
    leave                # locale.
    ret                  #
```

```
; op1 ^ op2
;
section .data
```

```
op1:    dd      3
op2:    dd      4
;
section .text
global _start
;
; Main.
;
_start:
    mov   esi, [op1] ; Base.
    mov   edi, [op2] ; Esponente.
bp1:
    push  edi        ; f_pwr (ESI, EDI) ==> EAX
    push  esi        ;
    call  f_pwr      ;
    add   esp, 8     ;
bp2:
    mov   ebx, eax   ; Restituisce il valore della potenza,
    mov   eax, 1     ; ammesso che sia abbastanza piccolo
    int   0x80       ; da poter essere rappresentato come
                     ; valore di uscita.
;
; Potenza di due numeri interi senza segno.
; f_pwr (a, b) ==> EAX
; EAX = a ^ b
;
f_pwr:
    enter 4,0
    pusha
    ;
    mov   esi, [ebp+8]  ; Base.
    mov   edi, [ebp+12] ; Esponente.
    ;
    cmp   esi, 0        ; Se la base è pari a 0,
    jz    f_pwr_end_0   ; restituisce 0.
    ;
    cmp   edi, 0        ; Se l'esponente è pari a 0,
    jz    f_pwr_end_1   ; restituisce 1.
    ;
    dec   edi           ; Riduce l'esponente di una unità.
    push  edi           ; f_pwr (ESI, EDI) ==> EAX
    push  esi           ;
    call  f_pwr         ;
    add   esp, 8        ;
    mul   esi           ; EDX:EAX = EAX*ESI
    mov   [ebp-4], eax  ; Salva il risultato.
    jmp   f_pwr_end_X   ; Conclude la funzione.
    ;
f_pwr_end_0:
    popa                ; Conclude la funzione con EAX = 0.
    mov eax, 0          ;
    leave               ;
    ret                 ;
f_pwr_end_1:
    popa                ; Conclude la funzione con EAX = 1.
    mov eax, 1          ;
    leave               ;
    ret                 ;
f_pwr_end_X:
    popa                ; Conclude la funzione con EAX pari
    mov eax, [ebp-4]    ; al valore salvato nella variabile
    leave               ; locale.
    ret                 ;
```

64.11.2 Fattoriale

Viene proposta una soluzione ricorsiva del problema del fattoriale. In pratica, $x!$ è pari a $x \cdot (x-1)!$, ma in particolare: se x è pari a 1, il risultato è uno.

All'interno della funzione '**`f_fact`**' viene riservato lo spazio per una sola variabile locale, che serve a conservare il valore di ***EAX***, per poi recuperarlo quando si ripristinano tutti i registri, prima della conclusione della funzione stessa.

```
# op1!
#
.section .data
op1:    .int    5
#
.section .text
```



```
.globl _start
#
# Main.
#
_start:
    mov   op1, %esi  # ESI contiene il valore di cui si
                     # vuole calcolare il fattoriale.
bp1:
    push  %esi       # f_fact (ESI) ==> EAX
    call  f_fact     #
    add   $4, %esp   #
bp2:
    mov   %eax, %ebx # Restituisce il valore del fattoriale,
    mov   $1, %eax   # ammesso che sia abbastanza piccolo
    int   $0x80      # da poter essere rappresentato come
                     # valore di uscita.
#
# Fattoriale di un numero senza segno.
# f_fatt (a) ==> EAX
# EAX = a!
#
f_fact:
    enter $4, $0
    pusha
    #
    mov   8(%ebp), %edi  # Valore di cui calcolare il
                         # fattoriale.
    cmp   $1, %edi       # Il fattoriale di 1 è 1.
    jz    f_fact_end_1   #
    #
    mov   %edi, %esi     # ESI contiene il valore di cui si
    dec   %esi           # vuole il fattoriale, ridotto di
                         # una unità.
    push  %esi           # f_fact (ESI) ==> EAX
    call  f_fact         #
    add   $4, %esp       #
    mul   %edi           # EDX:EAX = EAX*EDI
    mov   %eax, -4(%ebp) # Salva il risultato.
    jmp   f_fact_end_X   # Conclude la funzione.
    #
f_fact_end_1:
    popa                 # Conclude la funzione con EAX = 1.
    mov $1, %eax         #
    leave                #
    ret                  #
f_fact_end_X:
    popa                 # Conclude la funzione con EAX pari
    mov -4(%ebp), %eax   # al valore salvato nella variabile
    leave                # locale.
    ret                  #
```

```
; op1!
;
section .data
op1:    dd      5
;
section .text
global _start
;
; Main.
;
_start:
    mov   esi, [op1] ; ESI contiene il valore di cui si
                     ; vuole calcolare il fattoriale.
bp1:
    push  esi        ; f_fact (ESI) ==> EAX
    call  f_fact     ;
    add   esp, 4     ;
bp2:
    mov   ebx, eax   ; Restituisce il valore del fattoriale,
    mov   eax, 1     ; ammesso che sia abbastanza piccolo
    int   0x80       ; da poter essere rappresentato come
                     ; valore di uscita.
;
; Fattoriale di un numero senza segno.
; f_fatt (a) ==> EAX
; EAX = a!
;
f_fact:
    enter 4,0
    pusha
    ;
```

```
    mov   edi, [ebp+8] ; Valore di cui calcolare il
                       ; fattoriale.
    cmp   edi, 1       ; Il fattoriale di 1 è 1.
    jz    f_fact_end_1 ;
    ;
    mov   esi, edi     ; ESI contiene il valore di cui si
    dec   esi          ; vuole il fattoriale, ridotto di
                       ; una unità.
    push  esi          ; f_fact (ESI) ==> EAX
    call  f_fact       ;
    add   esp, 4       ;
    mul   edi          ; EDX:EAX = EAX*EDI
    mov   [ebp-4], eax ; Salva il risultato.
    jmp   f_fact_end_X ; Conclude la funzione.
    ;
f_fact_end_1:
    popa               ; Conclude la funzione con EAX = 1.
    mov eax, 1         ;
    leave              ;
    ret                ;
f_fact_end_X:
    popa               ; Conclude la funzione con EAX pari
    mov eax, [ebp-4]   ; al valore salvato nella variabile
    leave              ; locale.
    ret                ;
```

64.12 Indirizzamento dei dati

In generale, con l'architettura x86 ci si preoccupa di definire il modo in cui fare riferimento ai dati nel sorgente in linguaggio assemblatore (incluso ciò che riguarda la pila), mentre per il riferimento alle istruzioni si usano simboli che il compilatore traduce normalmente in indirizzi relativi (il «dislocamento»). Viene mostrata una tabella che riepiloga i vari modi con cui è possibile fare riferimento ai dati, secondo le due sintassi più comuni.

Tabella 64.193. Indirizzamento dei dati.

Esempio AT&T	Esempio Intel	Descrizione

`$21`
`$0x15`
`$025`
`$0b10101`
	`21`
	`0x15`
	`025`
	`10101b`
		Si sta facendo riferimento al numero 21_{10} in modo letterale.

`$`***nome_simbolo***	***nome_simbolo***	Si fa riferimento all'indirizzo corrispondente al simbolo, ovvero al numero che costituisce tale indirizzo.
nome_simbolo	`[`***nome_simbolo***`]`	Si fa riferimento all'area di memoria che inizia in corrispondenza del simbolo.
nome_simbolo $\pm$ ***n***	`[`***nome_simbolo*** $\pm$ ***n***`]`	Si fa riferimento all'area di memoria che inizia in corrispondenza del simbolo, $\pm$***n*** byte.
`%eax`	`eax`	Si sta facendo riferimento al registro ***EAX***, in qualità di variabile.
`(%eax)`	`[eax]`	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo contenuto nel registro ***EAX***
$\pm$***n***`(%eax)`	`[eax]` $\pm$ ***n***	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo contenuto nel registro ***EAX***, $\pm$***n*** byte.
nome_simbolo`(%eax)`	`[eax+`***nome_simbolo***`]`	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo a cui fa riferimento il simbolo, più il contenuto del registro ***EAX***.

Esempio AT&T	Esempio Intel	Descrizione
(***nome_simbolo*** $\pm$ ***n***)`(%eax)`	`[eax+`***nome_simbolo*** $\pm$ ***n***`]`	Si sta facendo riferimento all'area di memoria che inizia a partire dall'indirizzo a cui fa riferimento il simbolo, più il contenuto del registro ***EAX***, $\pm$***n*** byte.
`(%eax,%ebx,`***n***`)`	`[eax+%ebx*`***n***`]`	Si sta facendo riferimento all'area di memoria che inizia a partire da ***EAX***+(***EBX***$\cdot$***n***).
$\pm$***m***`(%eax,%ebx,`***n***`)`	`[eax+%ebx*`***n*** $\pm$ ***m***`]`	Si sta facendo riferimento all'area di memoria che inizia a partire da (***EAX***+(***EBX***$\cdot$***n***))$\pm$***m***.

64.12.1 Gestione di array

Per comprendere l'uso della forma più complessa di indirizzamento, si prenda l'esempio seguente, in cui appare un simbolo, denominato '**`record`**', che descrive una sequenza di valori, contenenti anche ciò che si vuole considerare una matrice di numeri.

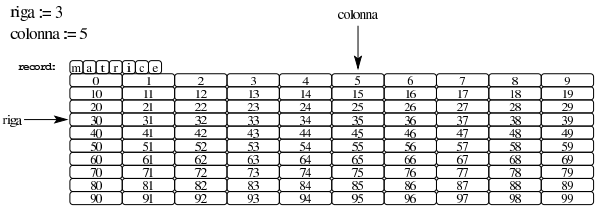
```
#
.section .data
record:          .ascii "matrice"
                 .int    0,  1,  2,  3,  4,  5,  6,  7,  8,  9
                 .int   10, 11, 12, 13, 14, 15, 16, 17, 18, 19
                 .int   20, 21, 22, 23, 24, 25, 26, 27, 28, 29
                 .int   30, 31, 32, 33, 34, 35, 36, 37, 38, 39
                 .int   40, 41, 42, 43, 44, 45, 46, 47, 48, 49
                 .int   50, 51, 52, 53, 54, 55, 56, 57, 58, 59
                 .int   60, 61, 62, 63, 64, 65, 66, 67, 68, 69
                 .int   70, 71, 72, 73, 74, 75, 76, 77, 78, 79
                 .int   80, 81, 82, 83, 84, 85, 86, 87, 88, 89
                 .int   90, 91, 92, 93, 94, 95, 96, 97, 98, 99
dimensione_int:  .int    4      # 32 bit
dimensione_riga: .int   40      # 4 * 10 byte
riga:            .int    3      # da 0 a 9
colonna:         .int    5      # da 0 a 9
#
.section .text
.globl _start
#
_start:
    mov   riga, %eax       # Legge le coordinate di riga e
    mov   colonna, %ecx    # colonna, copiandole in EAX e ECX.
    mull  dimensione_riga  # EDX:EAX := EAX * 40.
    mov   (record+7)(%eax,%ecx,4), %ebx
                           # EBX := array[riga,colonna].
bp1:
    mov   $1, %eax         # Restituisce il valore trovato
    int   $0x80            # nella matrice.
```

```
;
section .data
record:          db     "matrice"
                 dd      0,  1,  2,  3,  4,  5,  6,  7,  8,  9
                 dd     10, 11, 12, 13, 14, 15, 16, 17, 18, 19
                 dd     20, 21, 22, 23, 24, 25, 26, 27, 28, 29
                 dd     30, 31, 32, 33, 34, 35, 36, 37, 38, 39
                 dd     40, 41, 42, 43, 44, 45, 46, 47, 48, 49
                 dd     50, 51, 52, 53, 54, 55, 56, 57, 58, 59
                 dd     60, 61, 62, 63, 64, 65, 66, 67, 68, 69
                 dd     70, 71, 72, 73, 74, 75, 76, 77, 78, 79
                 dd     80, 81, 82, 83, 84, 85, 86, 87, 88, 89
                 dd     90, 91, 92, 93, 94, 95, 96, 97, 98, 99
dimensione_int:  dd      4      ; 32 bit
dimensione_riga: dd     40      ; 4 * 10 byte
riga:            dd      3      ; da 0 a 9
colonna:         dd      5      ; da 0 a 9
;
section .text
global _start
;
_start:
    mov   eax, [riga]            ; Legge le coordinate di riga
    mov   ecx, [colonna]         ; e colonna, copiandole in
                                 ; EAX e ECX.
    mul   long [dimensione_riga] ; EDX:EAX := EAX * 40.
```

```
    mov   ebx, [record+7+eax+ecx*4]
                                ; EBX := array[riga,colonna].
bp1:
    mov   eax, 1                ; Restituisce il valore
    int   0x80                  ; trovato nella matrice.
```

In pratica, '**record**' individua l'inizio di una struttura composta da una stringa di sette byte, seguita da un centinaio di interi da 32 bit, suddivisi idealmente in righe da 10 unità. In pratica, una «riga» di questi numeri occupa 40 byte.

Per accedere a un certo elemento della matrice contenuta nel «record», occorre considerare uno scostamento iniziale di 7 byte, quindi occorre calcolare la posizione dell'elemento, secondo lo schema che si vede nella figura successiva.



0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

$$7 + (\text{riga} \times 40) + (\text{colonna} \times 4)$$

Viene proposto un altro esempio equivalente, ma realizzato gestendo in modo diverso l'indirizzamento. La dichiarazione dei dati è la stessa:

```
…
_start:
    mov   riga, %eax          # Legge le coordinate di riga.
    mull  dimensione_riga     # EDX:EAX := EAX * 40.
    mov   %eax, %ecx          # ECX := riga * 40.
    #
    mov   colonna, %eax       # Legge le coordinate di colonna.
    mull  dimensione_int      # EDX:EAX := EAX * 4.
    add   %eax, %ecx          # ECX := (riga*40)+(colonna*4)
    #
    add   $7, %ecx            # ECX := 7+(riga*40)+(colonna*4)
    mov   record(%ecx), %ebx  # EBX := array[riga,colonna].
bp1:
    mov   $1, %eax            # Restituisce il valore trovato
    int   $0x80               # nella matrice.
```

```
…
_start:
    mov eax, [riga]             ; Legge le coordinate di riga.
    mul long [dimensione_riga]  ; EDX:EAX := EAX * 40.
    mov ecx, eax                ; ECX := riga * 40.
    ;
    mov eax, [colonna]          ; Legge le coord. di colonna.
    mul long [dimensione_int]   ; EDX:EAX := EAX * 4.
    add ecx, eax                ; ECX := (riga*40)+(colonna*4)
    ;
    add ecx, 7                  ; ECX := 7+(riga*40)+(colonna*4)
    mov ebx, [record+ecx]       ; EBX := array[riga,colonna].
bp1:
    mov eax, 1                  ; Restituisce il valore trovato
    int 0x80                    ; nella matrice.
```

Come si vede, si arriva a mettere nel registro ***ECX*** il risultato di 7+(***riga***×40)+(***colonna***×4); pertanto, alla fine si aggiunge solo l'indirizzo iniziale della struttura e si ottiene, complessivamente, l'indirizzo del valore cercato nella matrice.

Un altro esempio, dove si vede un altro modo di usare l'indirizzamento indiretto fornito dal linguaggio macchina:

```
…
_start:
    mov   riga, %eax        # Legge le coordinate di riga.
    mull  dimensione_riga   # EDX:EAX := EAX * 40.
    mov   %eax, %ecx        # ECX := riga * 40.
    #
    mov   colonna, %eax     # Legge le coordinate di colonna.
    mull  dimensione_int    # EDX:EAX := EAX * 4.
```



```
    add   %eax, %ecx        # ECX := (riga*40)+(colonna*4)
    #
    add   $7, %ecx          # ECX := 7+(riga*40)+(colonna*4)
    mov   $record, %edx     # EDX := indirizzo iniziale di
                            # "record".
    mov   (%edx,%ecx), %ebx # EBX := array[riga,colonna].
bp1:
    mov   $1, %eax          # Restituisce il valore trovato
    int   $0x80             # nella matrice.
```

```
…
_start:
    mov eax, [riga]             ; Legge le coordinate di riga.
    mul long [dimensione_riga]  ; EDX:EAX := EAX * 40.
    mov ecx, eax                ; ECX := riga * 40.
    ;
    mov eax, [colonna]          ; Legge le coordinate di colonna.
    mul long [dimensione_int]   ; EDX:EAX := EAX * 4.
    add ecx, eax                ; ECX := (riga*40)+(colonna*4)
    ;
    add ecx, 7          ; ECX := 7+(riga*40)+(colonna*4)
    mov edx, record     ; EDX := indirizzo iniziale di "record".
    mov ebx, [edx+ecx]  ; EBX := array[riga,colonna].
bp1:
    mov eax, 1                  ; Restituisce il valore trovato
    int 0x80                    ; nella matrice.
```

Infine, un esempio dove si fa in modo che ***ECX*** contenga l'indirizzo completo dell'elemento cercato:

```
…
_start:
    mov   riga, %eax      # Legge le coordinate di riga.
    mull  dimensione_riga # EDX:EAX := EAX * 40.
    mov   %eax, %ecx      # ECX := riga * 40.
    #
    mov   colonna, %eax   # Legge le coordinate di colonna.
    mull  dimensione_int  # EDX:EAX := EAX * 4.
    add   %eax, %ecx      # ECX := (riga*40)+(colonna*4)
    #
    add   $7, %ecx        # ECX := 7+(riga*40)+(colonna*4)
    add   $record, %ecx   # ECX := indirizzo completo.
    mov   (%ecx), %ebx    # EBX := array[riga,colonna].
bp1:
    mov   $1, %eax        # Restituisce il valore trovato
    int   $0x80           # nella matrice.
```

```
…
_start:
    mov eax, [riga]             ; Legge le coordinate di riga.
    mul long [dimensione_riga]  ; EDX:EAX := EAX * 40.
    mov ecx, eax                ; ECX := riga * 40.
    ;
    mov eax, [colonna]          ; Legge le coord. di colonna.
    mul long [dimensione_int]   ; EDX:EAX := EAX * 4.
    add ecx, eax                ; ECX := (riga*40)+(colonna*4)
    ;
    add ecx, 7          ; ECX := 7+(riga*40)+(colonna*4)
    add ecx, record     ; ECX := indirizzo completo.
    mov ebx, [ecx]      ; EBX := array[riga,colonna].
bp1:
    mov eax, 1                  ; Restituisce il valore trovato
    int 0x80                    ; nella matrice.
```

64.12.2 Istruzione «LEA»

L'istruzione '**LEA**' consente di calcolare un indirizzo di memoria e di salvarlo in un registro. Per esempio, le due istruzioni successive, una con '**MOV**' e l'altra con '**LEA**', fanno la stessa cosa:

```
    mov   (%ecx), %ebx
    lea   %ecx, %ebx
```

Ovvero:

```
    mov   ebx, [ecx]
    lea   ebx, ecx
```

Pertanto, con l'istruzione '**LEA**', ciò che appare nell'operando che svolge il ruolo di «origine» viene considerato solo per il suo indirizzo e copiato nella destinazione. La differenza sostanziale, rispetto a '**MOV**', sta nel poter usare le espressioni di indirizzamento indiretto.

Per riprendere l'esempio usato precedentemente per mostrare come accedere a una matrice di numeri, si potrebbe fare una cosa come si vede nel listato successivo, anche se sarebbe poco utile in tale circostanza particolare:

```
…
_start:
    mov  riga, %eax      # Legge le coordinate di riga e
    mov  colonna, %ecx   # colonna, copiandole in EAX e ECX.
    #
    mull dimensione_riga # EDX:EAX := EAX * 40.
    #
    lea  (record+7)(%eax,%ecx,4), %edx # EDX contiene
                                       # l'indirizzo
                                       # dell'elemento.
    mov  (%edx), %ebx        # EBX := array[riga,colonna].
bp1:
    mov  $1, %eax            # Restituisce il valore trovato
    int  $0x80               # nella matrice.
```

```
…
_start:
    mov   eax, [riga]    ; Legge le coordinate di riga e
    mov   ecx, [colonna] ; colonna, copiandole in EAX e ECX.
    ;
    mul   long [dimensione_riga]  ; EDX:EAX := EAX * 40.
    ;
    lea   edx, [record+7+eax+ecx*4] ; EDX contiene
                                    ; l'indirizzo
                                    ; dell'elemento.
    mov   ebx, [edx]         ; EBX := array[riga,colonna].
bp1:
    mov   eax, 1             ; Restituisce il valore trovato
    int   0x80               ; nella matrice.
```

64.13 Rappresentazione dei dati in memoria attraverso un esempio

Viene proposto un esempio in cui si utilizzano i tipi principali di variabili inizializzate in memoria, con due listati equivalenti, il primo adatto a GNU AS mentre il secondo è per NASM. In particolare, nel secondo listato, non potendo dichiarare variabili da 64 bit, sono state usate coppie di variabili a 32 bit, inizializzate tenendo conto dell'inversione *little endian*.

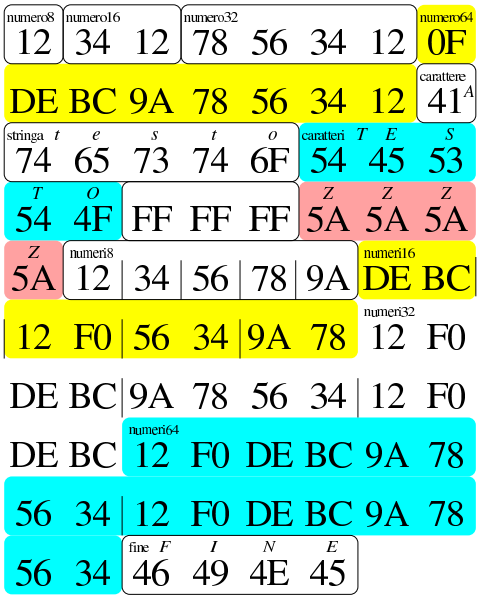
```
#
.section .data
numero8:    .byte   0b00010010           # 0x12       # 18
numero16:   .short  0b0001001000110100   # 0x1234     # 4660
numero32:   .int    0b00010010001101000101011001111000
                                         # 0x12345678
                                         # 305419896
numero64:   .quad   0x123456789ABCDEF0
carattere:  .byte   'A'
stringa:    .ascii  "testo"
caratteri:  .byte   'T', 'E', 'S', 'T', 'O'
            .skip 3, 0xFF
            .rept 4
            .byte   'Z'
            .endr
numeri8:    .byte   0x12, 0x34, 0x56, 0x78, 0x9A
numeri16:   .short  0xBCDE, 0xF012, 0x3456, 0x789A
numeri32:   .int    0xBCDEF012, 0x3456789A, 0xBCDEF012
numeri64:   .quad   0x3456789ABCDEF012, 0x3456789ABCDEF012
fine:       .byte   'F', 'I', 'N', 'E'
#
.section .text
.globl _start
#
_start:
    mov   $0, %ebx
    mov   $1, %eax
    int   $0x80
```

```
;
section .data
numero8:    db   00010010b              ; 0x12       = 18
numero16:   dw   0001001000110100b      ; 0x1234     = 4660
numero32:   dd   00010010001101000101011001111000b
```



```
                                        ; 0x12345678
                                        ;  = 305419896
numero64:   dd   0x9ABCDEF0, 0x12345678 ; 0x123456789ABCDEF0
carattere:  db   'A'
stringa:    db   "testo"
caratteri:  db   'T', 'E', 'S', 'T', 'O'
times 3     db   0xFF
times 4     db   'Z'
;
;
numeri8:    db   0x12, 0x34, 0x56, 0x78, 0x9A
numeri16:   dw   0xBCDE, 0xF012, 0x3456, 0x789A
numeri32:   dd   0xBCDEF012, 0x3456789A, 0xBCDEF012
numeri64:   dd   0xBCDEF012, 0x3456789A, 0xBCDEF012,
                 0x3456789A
fine:       db   'F', 'I', 'N', 'E'
;
section .text
global _start
;
_start:
    mov   ebx, 0
    mov   eax, 1
    int   0x80
```

Il programma in questione non fa alcunché e serve solo per verificare con GDB come sono rappresentati i dati in memoria e come questi si possono leggere. Complessivamente, le variabili occupano 78 byte, secondo lo schema che si vede nella figura successiva, dove vengono riprodotte anche le inversioni dovute alla rappresentazione *little endian*:



Per leggere la memoria già inizializzata con GDB non è necessario avviare il programma, in quanto, appena aperto, questa è subito accessibile:

```
$ gdb nome_programma [Invio]

(gdb) print /x (char[78])numero8 [Invio]

$1 = {0x12, 0x34, 0x12, 0x78, 0x56, 0x34, 0x12, 0xf0, 0xde,
  0xbc, 0x9a, 0x78, 0x56, 0x34, 0x12, 0x41, 0x74, 0x65,
  0x73, 0x74, 0x6f, 0x54, 0x45, 0x53, 0x54, 0x4f, 0xff,
  0xff, 0xff, 0x5a, 0x5a, 0x5a, 0x5a, 0x12, 0x34, 0x56,
  0x78, 0x9a, 0xde, 0xbc, 0x12, 0xf0, 0x56, 0x34, 0x9a,
  0x78, 0x12, 0xf0, 0xde, 0xbc, 0x9a, 0x78, 0x56, 0x34,
  0x12, 0xf0, 0xde, 0xbc, 0x12, 0xf0, 0xde, 0xbc, 0x9a,
  0x78, 0x56, 0x34, 0x12, 0xf0, 0xde, 0xbc, 0x9a, 0x78,
  0x56, 0x34, 0x46, 0x49, 0x4e, 0x45}
```

Quello che viene richiesto con il comando appena mostrato è di mostrare la memoria a partire dall'indirizzo corrispondente al simbolo

‘**numero8**’ (pertanto dall’inizio), in forma di array di byte, composto da 78 elementi. Si può verificare la corrispondenza tra quanto ottenuto e la figura mostrata in precedenza.

Con i comandi successivi si ispezionano le variabili, singolarmente. È sempre necessario dichiarare la dimensione a cui si è interessati, quando questa è diversa da quella predefinita (32 bit). Non è possibile ispezionare le variabili da 64 bit in modo complessivo, pertanto occorre arrangiarsi, come se fossero array di due numeri a 32 bit. Nei comandi mostrati appare spesso l’opzione ‘**/f**’, con la quale si dichiara il tipo di rappresentazione che si vuole ottenere.

```
(gdb) print /x (char)numero8 [Invio]

  $2 = 0x12
(gdb) print (char)numero8 [Invio]

  $3 = 18 '\022'
(gdb) print /x (short)numero16 [Invio]

  $4 = 0x1234
(gdb) print (short)numero16 [Invio]

  $5 = 4660
(gdb) print /x (int)numero32 [Invio]

  $6 = 0x12345678
(gdb) print (int)numero32 [Invio]

  $7 = 305419896
(gdb) print /x (int[2])numero64 [Invio]

  $8 = {0x9abcdef0, 0x12345678}
(gdb) print (char)carattere [Invio]

  $9 = 65 'A'
(gdb) print (char[11])carattere [Invio]

  $10 = "AtestoTESTO"
(gdb) print (char[5])stringa [Invio]

  $11 = "testo"
(gdb) print (char[5])caratteri [Invio]

  $12 = "TESTO"
(gdb) print /x (char[12])caratteri [Invio]

  $13 = {0x54, 0x45, 0x53, 0x54, 0x4f, 0xff, 0xff, 0xff, ↩
 ↪0x5a, 0x5a, 0x5a, 0x5a}
(gdb) print /x (char[5])numeri8 [Invio]

  $14 = {0x12, 0x34, 0x56, 0x78, 0x9a}
(gdb) print /d (char[5])numeri8 [Invio]

  $15 = {18, 52, 86, 120, -102}
(gdb) print /x (short[4])numeri16 [Invio]

  $16 = {0xbcde, 0xf012, 0x3456, 0x789a}
(gdb) print /d (short[4])numeri16 [Invio]

  $17 = {-17186, -4078, 13398, 30874}
(gdb) print /d (unsigned short[4])numeri16 [Invio]

  $18 = {48350, 61458, 13398, 30874}
(gdb) print /x (int[3])numeri32 [Invio]

  $19 = {0xbcdef012, 0x3456789a, 0xbcdef012}
(gdb) print /d (int[3])numeri32 [Invio]

  $20 = {-1126240238, 878082202, -1126240238}
```



```
(gdb) print /d (unsigned int[3])numeri32 [Invio]

  $21 = {3168727058, 878082202, 3168727058}
(gdb) print /x (int[4])numeri64 [Invio]

  $22 = {0xbcdef012, 0x3456789a, 0xbcdef012, 0x3456789a}
(gdb) print (char[4])fine [Invio]

  $23 = "FINE"
```

64.14 Esempi con gli array

Vengono mostrati alcuni esempi elementari di scansione di array, adatti alla compilazione con GNU AS e NASM. Si dà per scontato che si sappia utilizzare GDB per ispezionare la memoria e leggere, in particolare, il contenuto degli array stessi.

Negli esempi viene usata la direttiva ‘**.equ**’, o ‘**equ**’, per associare una sigla al livello in cui si trovano i dati nella pila (più precisamente nello *stack frame*).

Tutti gli esempi sono mostrati con listati a coppie: uno valido per GNU AS e l’altro per NASM.

64.14.1 Ricerca sequenziale

Viene mostrato un esempio di programma contenente una funzione che esegue una ricerca sequenziale all’interno di un array di interi, senza segno. Il metodo utilizzato si rifà a quanto descritto in pseudocodifica nella sezione 62.4.1. Il risultato della scansione viene emesso attraverso il valore restituito dal programma; ciò che si ottiene è precisamente l’indice dell’elemento trovato, oppure `-1` se nessun elemento corrisponde.

```
# Ricerca sequenziale.
#
.section .data
lista: .int 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 # Interi senza
                                               # segno.
a:     .int 0           # Indice minimo.
z:     .int 9           # Indice massimo.
#
.section .text
.globl _start
#
# Main.
#
_start:
    push  z          # f_rs ($lista, $7, a, z) ==> EAX
    push  a          # Si cerca il valore 7 nell'array
    push  $7         # «lista», tra gli indici «a» e «z».
    push  $lista     # Viene restituito l'indice
    call  f_rs       # dell'elemento trovato, oppure
    add   $16, %esp  # -1 se non è presente.
bp1:
    mov   %eax, %ebx # Restituisce l'indice trovato,
    mov   $1, %eax   # ammesso che sia abbastanza piccolo
    int   $0x80      # da poter essere rappresentato come
                     # valore di uscita.
#
# Ricerca sequenziale all'interno di una lista di valori.
# f_rs (lista, x, a, z) ==> EAX
# Al termine EAX contiene l'indice del valore trovato,
# oppure -1 se questo non c'è.
#
f_rs:
    enter $4, $0
    pusha
    .equ  rs_i,     -4          # Gli si associa EAX.
    .equ  rs_lista,  8          # Gli si associa ESI.
    .equ  rs_x,     12          # Gli si associa EDX.
    .equ  rs_a,     16
    .equ  rs_z,     20
    #
    mov   rs_lista(%ebp), %esi  # ESI contiene l'indirizzo
                                # dell'array.
    mov   rs_x(%ebp),     %edx  # EDX contiene il valore
                                # cercato.
    #
    mov   rs_a(%ebp),     %eax  # EAX viene usato come
```

```
                                # indice di scansione.
f_rs_loop:
    cmp   rs_z(%ebp),     %eax  # Se EAX è maggiore
                                # dell'indice massimo,
    ja    f_rs_non_trovato      # l'elemento cercato non
                                # c'è.
    #
    cmp   (%esi,%eax,4),  %edx  # Se il valore cercato
    je    f_rs_trovato          # corrisponde a quello
                                # dell'indice corrente,
                                # termina la scansione.
    #
    inc   %eax                  # Incrementa l'indice
    jmp   f_rs_loop             # di scansione e salta
                                # all'inizio del ciclo.
    #
f_rs_non_trovato:
    popa                  # Conclude la funzione con
    mov $-1, %eax         # EAX = -1.
    leave                 #
    ret                   #
f_rs_trovato:
    mov %eax, rs_i(%ebp)  # Salva EAX nella variabile
                          # locale prevista.
    popa                  # Conclude la funzione con EAX
    mov rs_i(%ebp), %eax  # pari al valore salvato nella
    leave                 # variabile locale.
    ret                   #
```

```
; Ricerca sequenziale.
;
section .data
lista: dd 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 ; Interi senza
                                             ; segno.
a:     dd 0             ; Indice minimo.
z:     dd 9             ; Indice massimo.
;
section .text
global _start
;
; Main.
;
_start:
    push  long [z] ; f_rs ($lista, $7, a, z) ==> EAX
    push  long [a] ; Si cerca il valore 7 nell'array
    push  long 7   ; «lista», tra gli indici «a» e «z».
    push  lista    ; Viene restituito l'indice dell'elemento
    call  f_rs     ; trovato, oppure -1 se non è presente.
    add   esp, 16  ;
bp1:
    mov   ebx, eax ; Restituisce l'indice trovato,
    mov   eax, 1   ; ammesso che sia abbastanza piccolo
    int   0x80     ; da poter essere rappresentato come
                   ; valore di uscita.
;
; Ricerca sequenziale all'interno di una lista di valori.
; f_rs (lista, x, a, z) ==> EAX
; Al termine EAX contiene l'indice del valore trovato,
; oppure -1 se questo non c'è.
;
f_rs:
    enter 4, 0
    pusha
    rs_i     equ -4             ; Gli si associa EAX.
    rs_lista equ  8             ; Gli si associa ESI.
    rs_x     equ 12             ; Gli si associa EDX.
    rs_a     equ 16
    rs_z     equ 20
    ;
    mov   esi, [rs_lista+ebp] ; ESI contiene l'indirizzo
                              ; dell'array.
    mov   edx, [rs_x+ebp]     ; EDX contiene il valore
                              ; cercato.
    ;
    mov   eax, [rs_a+ebp]     ; EAX viene usato come indice
                              ; di scansione.
f_rs_loop:
    cmp   eax, [rs_z+ebp]     ; Se EAX è maggiore
    ja    f_rs_non_trovato    ; dell'indice massimo,
                              ; l'elemento cercato non c'è.
    ;
```



```
    cmp   edx, [esi+eax*4]    ; Se il valore cercato
    je    f_rs_trovato        ; corrisponde a quello
                              ; dell'indice corrente,
                              ; termina la scansione.
    ;
    inc   eax                 ; Incrementa l'indice di
    jmp   f_rs_loop           ; scansione e salta
                              ; all'inizio del ciclo.
    ;
f_rs_non_trovato:
    popa            ; Conclude la funzione con EAX = -1.
    mov eax, -1     ;
    leave           ;
    ret             ;
f_rs_trovato:
    mov [rs_i+ebp], eax ; Salva EAX nella variabile locale
                        ; prevista.
    popa                ; Conclude la funzione con EAX pari
    mov eax, [rs_i+ebp] ; al valore salvato nella variabile
    leave               ; locale.
    ret                 ;
```

64.14.2 Ricerca binaria

Viene mostrato un esempio di programma contenente una funzione che esegue una ricerca binaria all'interno di un array ordinato, di interi con segno. Il metodo utilizzato si rifà a quanto descritto in pseudocodifica nella sezione 62.4.2. Il risultato della scansione viene emesso attraverso il valore restituito dal programma; ciò che si ottiene è precisamente l'indice dell'elemento trovato, oppure -1 se nessun elemento corrisponde.

```
# Ricerca binaria.
#
.section .data
lista: .int 1, 3, 4, 7, 9, 10, 11, 22, 23, 44 # Interi con
                                               # segno.
a:     .int 0             # Indice minimo.
z:     .int 9             # Indice massimo.
#
.section .text
.globl _start
#
# Main.
#
_start:
    push z          # f_rb ($lista, $7, a, z) ==> EAX
    push a          # Si cerca il valore 7 nell'array
    push $7         # ordinato «lista», tra gli indici
    push $lista     # «a» e «z». Viene restituito l'indice
    call f_rb       # dell'elemento trovato, oppure -1 se
    add  $16, %esp  # non è presente.
bp1:
    mov  %eax, %ebx # Restituisce l'indice trovato,
    mov  $1, %eax   # ammesso che sia abbastanza piccolo
    int  $0x80      # da poter essere rappresentato come
                    # valore di uscita.
#
# Ricerca binaria all'interno di una lista ordinata di
# valori.
# f_rb (lista, x, a, z) ==> EAX
# Al termine EAX contiene l'indice del valore trovato,
# oppure -1 se questo non c'è.
#
f_rb:
    enter $4, $0
    pusha
    .equ rb_m,     -4           # Gli si associa EAX.
    .equ rb_lista,  8           # Gli si associa ESI.
    .equ rb_x,     12           # Gli si associa EDX.
    .equ rb_a,     16
    .equ rb_z,     20
    #
    mov  rb_lista(%ebp), %esi # ESI contiene l'indirizzo
                              # dell'array.
    mov  rb_x(%ebp),     %edx # EDX contiene il valore
                              # cercato.
                              # EAX viene usato come
                              # «elemento centrale».
    #
```

```
    mov  rb_a(%ebp), %eax     # Calcola l'indice
    add  rb_z(%ebp), %eax     # dell'elemento centrale e lo
    sar  $1, %eax             # mette in EAX. Lo scorrimento
                              # a destra serve a dividere
                              # per due EAX.
    #
bp2:
    cmp  rb_a(%ebp), %eax     # Se EAX, ovvero l'indice,
    jb   f_rb_non_trovato     # è minore dell'indice minimo,
                              # l'elemento non c'è.
    #
    cmp  (%esi,%eax,4), %edx  # Se il valore cercato è
    jl   f_rb_minore          # minore di quello trovato,
    jg   f_rb_maggiore        # cerca nella parte inferiore;
    je   f_rb_fine            # se è maggiore cerca in
                              # quella superiore; se è
                              # uguale, l'elemento è stato
                              # trovato.
    #
f_rb_minore:
    dec   %eax
    push  %eax       # f_rb (lista, x, a, z) ==> EAX
    push  rb_a(%ebp) # Si cerca il valore nell'array
    push  %edx       # ordinato «lista», tra gli indici
    push  %esi       # «a» e «z». Viene restituito l'indice
    call  f_rb       # dell'elemento trovato, oppure -1 se
    add   $16, %esp  # non è presente.
    jmp   f_rb_fine
    #
f_rb_maggiore:
    inc   %eax
    push  rb_z(%ebp) # f_rb (lista, x, a, z) ==> EAX
    push  %eax       # Si cerca il valore nell'array
    push  %edx       # ordinato «lista», tra gli indici
    push  %esi       # «a» e «z». Viene restituito l'indice
    call  f_rb       # dell'elemento trovato, oppure -1 se
    add   $16, %esp  # non è presente.
    jmp   f_rb_fine
    #
f_rb_non_trovato:
    mov $-1, %eax    # Conclude la funzione con EAX = -1.
    jmp f_rb_fine
f_rb_fine:
    mov %eax, rb_m(%ebp) # Salva EAX nella variabile locale
    popa                 # prevista. Conclude la funzione
    mov rb_m(%ebp), %eax # con EAX pari al valore salvato
    leave                # nella variabile locale.
    ret                  #
```

```
; Ricerca binaria.
;
section .data
lista:  dd 1, 3, 4, 7, 9, 10, 11, 22, 23, 44 ; Interi con
                                             ; segno.
a:      dd 0              ; Indice minimo.
z:      dd 9              ; Indice massimo.
;
section .text
global _start
;
; Main.
;
_start:
    push long [z] ; f_rb ($lista, $7, a, z) ==> EAX
    push long [a] ; Si cerca il valore 7 nell'array ordinato
    push long 7   ; «lista», tra gli indici «a» e «z».
    push lista    ; Viene restituito l'indice dell'elemento
    call f_rb     ; trovato, oppure -1 se non è presente.
    add  esp, 16  ;
bp1:
    mov  ebx, eax ; Restituisce l'indice trovato,
    mov  eax, 1   ; ammesso che sia abbastanza piccolo
    int  0x80     ; da poter essere rappresentato come
                  ; valore di uscita.
;
; Ricerca binaria all'interno di una lista ordinata di
; valori.
; f_rb (lista, x, a, z) ==> EAX
; Al termine EAX contiene l'indice del valore trovato,
; oppure -1 se questo non c'è.
;
```



```
f_rb:
    enter 4, 0
    pusha
    rb_m     equ -4              ; Gli si associa EAX.
    rb_lista equ  8              ; Gli si associa ESI.
    rb_x     equ 12              ; Gli si associa EDX.
    rb_a     equ 16
    rb_z     equ 20
    ;
    mov esi, [rb_lista+ebp] ; ESI contiene l'indirizzo
                            ; dell'array.
    mov edx, [rb_x+ebp]     ; EDX contiene il valore
                            ; cercato.
                            ; EAX viene usato come
                            ; «elemento centrale».
    ;
    mov eax, [rb_a+ebp]     ; Calcola l'indice dell'elemento
    add eax, [rb_z+ebp]     ; centrale e lo mette in EAX. Lo
    sar eax, 1              ; scorrimento a destra serve a
                            ; dividere per due EAX.
    ;
bp2:
    cmp eax, [rb_a+ebp]     ; Se EAX, ovvero l'indice, è
    jb  f_rb_non_trovato    ; minore dell'indice minimo,
                            ; l'elemento non c'è.
    ;
    cmp edx, [esi+eax*4]    ; Se il valore cercato è minore
    jl  f_rb_minore         ; di quello trovato, cerca nella
    jg  f_rb_maggiore       ; parte inferiore; se è maggiore
    je  f_rb_fine           ; cerca in quella superiore; se
                            ; è uguale, l'elemento è stato
                            ; trovato.
    ;
f_rb_minore:
    dec   eax
    push  eax              ; f_rb (lista, x, a, z) ==> EAX
    push  long [rb_a+ebp]  ; Si cerca il valore nell'array
    push  edx              ; ordinato «lista», tra gli
    push  esi              ; indici «a» e «z». Viene
    call  f_rb             ; restituito l'indice
    add   esp, 16          ; dell'elemento trovato, oppure
    jmp   f_rb_fine        ; -1 se non è presente.
    ;
f_rb_maggiore:
    inc   eax
    push  long [rb_z+ebp]  ; f_rb (lista, x, a, z) ==> EAX
    push  eax              ; Si cerca il valore nell'array
    push  edx              ; ordinato «lista», tra gli
    push  esi              ; indici «a» e «z». Viene
    call  f_rb             ; restituito l'indice
    add   esp, 16          ; dell'elemento trovato, oppure
                           ; -1 se non è presente.
    jmp   f_rb_fine
    ;
f_rb_non_trovato:
    mov eax, -1            ; Conclude la funzione con
    jmp f_rb_fine          ; EAX = -1.
f_rb_fine:
    mov [rb_m+ebp], eax    ; Salva EAX nella variabile
    popa                   ; locale prevista. Conclude la
    mov eax, [rb_m+ebp]    ; funzione con EAX pari al
    leave                  ; valore salvato nella variabile
    ret                    ; locale.
```

64.14.3 Bubblesort

Viene mostrato un esempio di programma che esegue il riordino di array attraverso l'algoritmo Bubblesort. L'array in question contiene numeri interi con segno. L'algoritmo è descritto in pseudocodifica nella sezione 62.5.1.

```
# Bubblesort
#
.section .data
lista: .int 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 # Interi con
                                               # segno.
a:     .int 0                  # Indice minimo.
z:     .int 9                  # Indice massimo.
#
.section .text
.globl _start
```

```
#
# Main.
#
_start:
    push  z         # f_bs ($lista, a, z)
    push  a         # Riordina l'array «lista» senza
    push  $lista    # restituire alcun valore.
    call  f_bs      #
    add   $12, %esp #
bp1:
    mov   $0, %ebx  # Restituisce sempre zero.
    mov   $1, %eax  #
    int   $0x80     #
#
# Riordino con l'algoritmo «bubblesort».
# f_bs (lista, a, z)
#
f_bs:
    enter $0, $0
    pusha
    .equ  bs_lista,  8        # EDX
    .equ  bs_a,     12        #
    .equ  bs_z,     16        #
                              #
    mov bs_lista(%ebp), %edx  # EDX contiene il riferimento
                              # alla lista.
                              # ESI viene usato come indice
                              # di scansione.
                              # EDI viene usato come indice
                              # di scansione.
                              # EAX viene usato per
                              # scambiare i valori.
                              # EBX viene usato per
                              # scambiare i valori.
    mov   bs_a(%ebp), %esi    # ESI parte dall'indice
                              # iniziale.
f_bs_loop_1:
    cmp   bs_z(%ebp), %esi    # Se ESI >= z, termina.
    jae   f_bs_end_loop_1     #
    #
    mov   %esi, %edi          # EDI := ESI - 1
    inc   %edi                #
f_bs_loop_2:
    cmp   bs_z(%ebp), %edi    # Se EDI > z, termina.
    ja    f_bs_end_loop_2     #
    #
    mov   (%edx,%esi,4), %eax # Se EBX < EAX scambia
    mov   (%edx,%edi,4), %ebx # i valori
    cmp   %eax, %ebx          #
    jl    f_bs_scambio        #
f_bs_loop_2_inc_edi:
    inc   %edi                # EDI++
    jmp   f_bs_loop_2         #
f_bs_scambio:
    mov   %eax, (%edx,%edi,4) # lista[ESI] :==: lista[EDI]
    mov   %ebx, (%edx,%esi,4) #
    jmp   f_bs_loop_2         #
f_bs_end_loop_2:
    inc   %esi                # ESI++
    jmp   f_bs_loop_1         #
f_bs_end_loop_1:
    popa
    leave
    ret
```

```
; Bubblesort
;
section .data
lista: dd 1, 4, 3, 7, 9, 10, 22, 44, 11, 23 ; Interi con
                                            ; segno.
a:     dd 0                     ; Indice minimo.
z:     dd 9                     ; Indice massimo.
;
section .text
global _start
;
; Main.
;
_start:
    push  long [z]   ; f_bs ($lista, a, z)
    push  long [a]   ; Riordina l'array «lista» senza
    push  long lista ; restituire alcun valore.
    call  f_bs       ;
    add   esp, 12    ;
bp1:
    mov   ebx, 0     ; Restituisce sempre zero.
    mov   eax, 1     ;
    int   0x80       ;
;
; Riordino con l'algoritmo «bubblesort».
; f_bs (lista, a, z)
;
f_bs:
    enter 0, 0
    pusha
    bs_lista equ  8           ; EDX
    bs_a     equ 12           ;
    bs_z     equ 16           ;
                              ;
    mov   edx, [bs_lista+ebp] ; EDX contiene il riferimento
                              ; alla lista.
                              ; ESI viene usato come indice
                              ; di scansione.
                              ; EDI viene usato come indice
                              ; di scansione.
                              ; EAX viene usato per
                              ; scambiare i valori.
                              ; EBX viene usato per
                              ; scambiare i valori.
    mov   esi, [bs_a+ebp]     ; ESI parte dall'indice
                              ; iniziale.
f_bs_loop_1:
    cmp   esi, [bs_z+ebp]     ; Se ESI >= z, termina.
    jae   f_bs_end_loop_1     ;
    ;
    mov   edi, esi            ; EDI := ESI - 1
    inc   edi                 ;
f_bs_loop_2:
    cmp   edi, [bs_z+ebp]     ; Se EDI > z, termina.
    ja    f_bs_end_loop_2     ;
    ;
    mov   eax, [edx+esi*4]    ; Se EBX < EAX scambia
    mov   ebx, [edx+edi*4]    ; i valori.
    cmp   ebx, eax            ;
    jl    f_bs_scambio        ;
f_bs_loop_2_inc_edi:
    inc   edi                 ; EDI++
    jmp   f_bs_loop_2         ;
f_bs_scambio:
    mov   [edx+edi*4], eax    ; lista[ESI] :==: lista[EDI]
    mov   [edx+esi*4], ebx    ;
    jmp   f_bs_loop_2         ;
f_bs_end_loop_2:
    inc   esi                 ; ESI++
    jmp   f_bs_loop_1         ;
f_bs_end_loop_1:
    popa
    leave
    ret
```


Per verificare il funzionamento del programma si deve usare necessariamente GDB. Inizialmente, prima di mettere in esecuzione il programma, si vede l'array nel suo stato originale:

```
(gdb) print (int[10])lista[Invio]

  $1 = {1, 4, 3, 7, 9, 10, 22, 44, 11, 23}
```

Si fissa quindi uno stop e si avvia il programma:

```
(gdb) break bp1[Invio]

(gdb) run[Invio]
```

Quando il programma viene sospeso in corrispondenza di '**bp1**', l'array è ordinato:

```
(gdb) print (int[10])lista[Invio]

  $1 = {1, 3, 4, 7, 9, 10, 11, 22, 23, 44}
```

64.15 Calcoli con gli indirizzi in fase di compilazione

Attraverso le funzionalità del compilatore è possibile calcolare la distanza tra due indirizzi, espressa in byte. È anche possibile fare riferimento all'indirizzo attuale, attraverso un simbolo predefinito. Nelle sezioni successive vengono mostrati alcuni esempi per dimostrare l'uso di queste funzionalità.

64.15.1 Distanza tra due indirizzi

L'esempio seguente serve a dimostrare come il compilatore possa calcolare la distanza tra due indirizzi, contrassegnati da dei simboli, inizializzando con tale valore calcolato una variabile globale. In pratica, viene calcolata la grandezza complessiva in byte di un array di numeri interi; grandezza che viene poi emessa come valore di uscita.

```
#
.section .data
lista:  .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      .int    (z - lista)
#
.section .text
.globl _start
#
_start:
    mov   z,  %ebx
    mov   $1, %eax
    int   $0x80
```

```
;
section .data
lista:  dd      1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      dd      (z - lista)
;
section .text
global _start
;
_start:
    mov   ebx, [z]
    mov   eax, 1
    int   0x80
```

Il significato dell'istruzione è intuitivo: alla variabile '**z**' si assegna la differenza tra gli indirizzi utilizzati da '**lista**' a '**z**'. In questo caso, il risultato che si ottiene è 40, dal momento che si contano 10 valori da 4 byte ciascuno. Eventualmente, si può fare riferimento alla posizione attuale in modo differente, in questo caso significa sostituire il riferimento esplicito alla variabile '**z**' con un riferimento implicito:

```
…
lista:  .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      .int    (. - lista)
…
```

```
…
lista:  dd      1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      dd      ($ - lista)
…
```

Si possono anche fare dei calcoli più complessi, come nel caso dell'esempio seguente in cui si determina l'indice superiore dell'array. Il risultato che si ottiene è nove, dal momento che l'indice del primo elemento deve essere zero.

```
…
lista:  .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      .int    (z - lista) / 4 - 1
…
```

```
…
lista:  dd      1, 4, 3, 7, 9, 10, 22, 44, 11, 23
z:      dd      (z - lista) / 4 - 1
…
```


64.15.2 Riempimento di spazio inutilizzato

In certe situazioni è necessario riempire una certa area di memoria (o di codice) in modo che complessivamente occupi una dimensione data. Questo procedimento si usa specialmente quando si genera un file binario, privo di formato (come nel caso di un settore di avvio), che deve avere una dimensione stabilita e che in una certa posizione deve contenere un'impronta determinata. Qui viene dimostrato il concetto intervenendo solo nell'area della memoria che viene inizializzata.

```
#
.section .data
lista:  .int    1, 4, 3, 7, 9, 10, 22, 44, 11, 23
        .skip (0x30 - (. - lista)), 0xFF
#
.section .text
.globl _start
#
_start:
    mov   $0, %ebx
    mov   $1, %eax
    int   $0x80
```

```
;
section .data
lista:  dd      1, 4, 3, 7, 9, 10, 22, 44, 11, 23
times   (30h - ($ - lista))  db  0xFF
;
section .text
global _start
;
_start:
    mov   ebx, 0
    mov   eax, 1
    int   0x80
```

In questo caso, dopo la definizione dell'array, si richiede al compilatore di allocare altro spazio di memoria, in modo da occupare complessivamente 48 byte (30_{16}), riempiendo lo spazio ulteriore con caratteri FF_{16}. Naturalmente, il valore complessivo dello spazio da utilizzare può essere espresso in qualunque base di numerazione; in questo caso, la scelta di rappresentare in base sedici è motivata dal fatto che con NASM non si può usare la forma consueta (non si può scrivere '**0x30**'), perché si otterrebbe un risultato differente, a causa di un errore di interpretazione da parte del compilatore.

In quasi tutti gli esempi di questo capitolo, realizzati per il compilatore NASM, si usa la notazione '**0x*nn*…**' per esprimere un numero in base sedici. Generalmente l'interpretazione da parte di NASM è corretta, ma nella situazione particolare mostrata, il compilatore si confonde e riconosce solo la forma '***nn*…h**'.

Dal momento che l'array occupa già 40 byte, vengono aggiunti semplicemente 8 byte, pari a due gruppi da 32 bit:

```
(gdb) print /x (int[12])lista [Invio]

$1 = {0x1, 0x4, 0x3, 0x7, 0x9, 0xa, 0x16, 0x2c, 0xb, 0x17,
  0xffffffff, 0xffffffff}
```

64.16 Interazione con il sistema operativo

È possibile gestire un certo grado di comunicazione tra il programma in linguaggio assemblatore e il sistema GNU/Linux. In particolare si possono ottenere i parametri della chiamata del programma (gli argomenti della riga di comando) ed è possibile chiamare delle funzioni di sistema attraverso delle «interruzioni».

64.16.1 Parametri di chiamata del programma

All'avvio del programma, questo riceve una pila contenente il numero degli argomenti della riga di comando (nome del programma incluso), il nome del programma che è stato avviato e quindi gli argomenti stessi. Si può realizzare un sorgente molto semplice per l'indagine con GDB:

```
.section .text
.globl _start
_start:
    mov   %esp, %ebp
bp1:
    mov   $0, %ebx    # Restituisce zero.
    mov   $1, %eax    #
    int   $0x80       #
```

```
section .text
global _start
_start:
    mov   ebp, esp
bp1:
    mov   ebx, 0    ; Restituisce zero.
    mov   eax, 1    ;
    int   0x80      ;
```

Supponendo che il programma compilato si chiami '**argomenti**', lo si avvia sotto il controllo di GDB nello stesso modo di sempre:

```
$ gdb argomenti [Invio]
```

Gli argomenti del programma vanno passati necessariamente attraverso un comando di GDB:

```
(gdb) set args 1 2 3 4 "ciao amore" [Invio]
```

Si fissa il punto di sospensione del programma e quindi si avvia:

```
(gdb) break bp1 [Invio]

(gdb) run [Invio]
```

Il programma viene sospeso in corrispondenza del simbolo '**bp1**' e si può consultare la pila, o più precisamente lo *stack frame* della pila:

```
(gdb) backtrace [Invio]

#0  bp1 () at argomenti.s:6
#1  0x00000006 in ?? ()
#2  0xbfeb1a89 in ?? ()
#3  0xbfeb1a98 in ?? ()
#4  0xbfeb1a9a in ?? ()
#5  0xbfeb1a9c in ?? ()
#6  0xbfeb1a9e in ?? ()
#7  0xbfeb1aa0 in ?? ()
#8  0x00000000 in ?? ()
```

In questa situazione non sono presenti variabili locali; quindi, nella pila, dopo l'indirizzo corrispondente al simbolo '**bp1**' (che si trova lì solo perché si sta usando GDB ed è stato sospeso il corso del programma), appare la quantità di argomenti (sei); gli elementi successivi contengono dei puntatori alle stringhe che rappresentano i vari argomenti ricevuti (si osservi che gli argomenti sono rappresentati tutti in forma di stringa), stringhe che sono tutte terminate con un byte a zero. Per leggere gli argomenti con GDB si devono fare dei tentativi; qui vengono indicate le dimensioni esatte, ma se si usano dimensioni maggiori si possono vedere delle porzioni degli argomenti successivi:

```
(gdb) print (char[33])*0xbfeb1a89 [Invio]

  $1 = "/tmp/argomenti\0001\0002\0003\0004\000ciao amore"
```

In questo caso si legge il primo argomento, ma usando una dimensione eccessiva, si vedono di seguito anche gli altri, separati dai vari byte a zero, rappresentati con la sequenza '**\000**'. Il primo argomento da solo sarebbe:

```
(gdb) print (char[14])*0xbfeb1a89 [Invio]

  $2 = "/tmp/argomenti"
```

Gli altri argomenti:

```
(gdb) print (char[1])*0xbfeb1a98 [Invio]

  $3 = "1"

(gdb) print (char[1])*0xbfeb1a9a [Invio]

  $4 = "2"
```



```
(gdb) print (char[1])*0xbfeb1a9c [Invio]

  $5 = "3"

(gdb) print (char[1])*0xbfeb1a9e [Invio]

  $6 = "4"

(gdb) print (char[9])*0xbfeb1aa0 [Invio]

  $7 = "ciao amore"

(gdb) quit [Invio]
```

64.16.2 Funzioni del sistema operativo

Si accede alle funzioni offerte dal sistema operativo attraverso quella che è nota come «interruzione software» (*interrupt*) e si genera con l'istruzione '**INT**'. Per la precisione, in un sistema GNU/Linux occorre l'interruzione 80_{16} (128_{10}), come è stato mostrato in tutti gli esempi apparsi fino a questo punto. Per selezionare il tipo di funzione e per passarle degli argomenti si usano i registri in questo modo:

Registro	Utilizzo
EAX	Contiene il numero che rappresenta la funzione di sistema.
EBX	Primo parametro della funzione.
ECX	Secondo parametro della funzione.
EDX	Terzo parametro della funzione.
ESI	Quarto parametro della funzione.
EDI	Quinto parametro della funzione.

Se la funzione deve restituire un valore, questo viene ottenuto attraverso il registro ***EAX***.

Per una mappa completa delle chiamate di sistema si può consultare *http://wayback.archive.org/web/*/www.lxhp.in-berlin.de/lhpsysc0.html*, come annotato nei riferimenti alla fine del capitolo. Qui vengono mostrate delle tabelle riepilogative di alcune funzioni importanti.

Pagina di manuale	Descrizione	***EAX***	***EBX***	***ECX***	***EDX***
exit(2)	Conclude il funzionamento del programma restituendo un valore.	1	Valore da restituire: numero intero compreso tra zero e 255.		
read(2)	Legge da un descrittore di file e mette i dati letti in una memoria tampone. Attraverso ***EAX*** restituisce la quantità di byte letta effettivamente e aggiorna il puntatore del file per la lettura successiva.	3	Descrittore del file da leggere.	Indirizzo iniziale della memoria tampone.	Dimensione in byte della memoria tampone.
write(2)	Scrive il contenuto di una memoria tampone in un descrittore di file. Attraverso ***EAX*** restituisce la quantità di byte scritta effettivamente.	4	Descrittore del file da scrivere.	Indirizzo iniziale della memoria tampone.	Dimensione in byte della memoria tampone.

64.16.3 Esempi di lettura e scrittura con i flussi standard

Di solito, il primo programma che si scrive è quello che visualizza un messaggio e termina, ma in questo caso, un'operazione così semplice sul piano teorico, in pratica è già abbastanza complicata. Quello che segue è un programma che, avvalendosi di una chiamata di sistema, visualizza un messaggio attraverso lo standard output. Come si può osservare, si utilizza anche una tecnica per far calcolare al compilatore la lunghezza della stringa da visualizzare.

```
#
.equ SYS_EXIT,  1  # exit(2)
.equ SYS_WRITE, 4  # write(2)
.equ STDOUT,    1  # Descrittore di standard output.
#
```

```
.section .data                  # Qui si dichiara la stringa
msg:  .ascii "Ciao a tutti!\n"  # da visualizzare,
size = . - msg                  # calcolandone la
                                # dimensione.
#
.section .text
.globl _start
_start:
    mov   $SYS_WRITE, %eax      # Scrive nello standard
    mov   $STDOUT,    %ebx      # output.
    mov   $msg,       %ecx      #
    mov   $size,      %edx      #
    int   $0x80
exit:
    mov   $SYS_EXIT,  %eax      # Conclude il funzionamento.
    mov   $0,         %ebx      #
    int   $0x80
```

```
;
SYS_EXIT   equ     1  ; exit(2)
SYS_WRITE  equ     4  ; write(2)
STDOUT     equ     1  ; Descrittore di standard output.
;
section .data                   ; Qui si dichiara la stringa
msg:  db  "Ciao a tutti!", 0x0A ; da visualizzare,
size  equ $ - msg               ; calcolandone la
                                ; dimensione.
;
section text
global _start
_start:
    mov   eax, SYS_WRITE        ; Scrive nello standard
    mov   ebx, STDOUT           ; output.
    mov   ecx, msg              ;
    mov   edx, size             ;
    int   0x80
exit:
    mov   eax, SYS_EXIT         ; Conclude il funzionamento.
    mov   ebx, 0                ;
    int   0x80
```

Segue un esempio di programma che legge dallo standard input e scrive ciò che ha letto attraverso lo standard output. Come già nell'esempio precedente, vengono dichiarate inizialmente delle costanti per semplificare la lettura del codice; inoltre vengono usate aree di memoria non inizializzate e delle funzioni banali senza parametri, per le quali non si utilizzano variabili locali. Si mostrano due listati, uno adatto per GNU AS e l'altro per NASM.

```
#
.equ SYS_EXIT,  1    # exit(2)
.equ SYS_READ,  3    # read(2)
.equ SYS_WRITE, 4    # write(2)
.equ STDIN,     0    # Descrittore di standard input.
.equ STDOUT,    1    # Descrittore di standard output.
.equ STDERR,    2    # Descrittore di standard error.
.equ MAX_SIZE,  1000 # Dimensione massima dei dati da
                     # leggere.
#
.section .data       # Non ci sono variabili già
                     # inizializzate.
#
.section .bss
.lcomm record, MAX_SIZE # Memoria tampone per la lettura dei
                        # dati.
.lcomm size, 4          # Quantità di byte letti
                        # effettivamente.
#
.section .text
.globl _start
_start:
read_write_begin:
    call  read             # Legge dallo standard input.
    cmp   $0, %eax         # Se sono stati letti zero byte,
    jz    read_write_end   # il ciclo termina.
    call  write            # Scrive i byte letti nello
                           # standard output.
    jmp   read_write_begin # Ripete il ciclo.
read_write_end:
    jmp   exit
```



```
read:
    mov   $SYS_READ,  %eax # Legge dallo standard input.
    mov   $STDIN,     %ebx #
    mov   $record,    %ecx #
    mov   $MAX_SIZE,  %edx #
    int   $0x80
    mov   %eax,       size # Salva la dimensione letta
                           # effettivamente.
    ret
write:
    mov   $SYS_WRITE, %eax # Scrive nello standard output.
    mov   $STDOUT,    %ebx #
    mov   $record,    %ecx #
    mov   size,       %edx #
    int   $0x80
    ret
exit:
    mov   $SYS_EXIT,  %eax # Conclude il funzionamento.
    mov   $0,         %ebx #
    int   $0x80
```

```
;
SYS_EXIT   equ     1 ; exit(2)
SYS_READ   equ     3 ; read(2)
SYS_WRITE  equ     4 ; write(2)
STDIN      equ     0 ; Descrittore di standard input.
STDOUT     equ     1 ; Descrittore di standard output.
STDERR     equ     2 ; Descrittore di standard error.
MAX_SIZE   equ  1000 ; Dimensione massima dei dati da
                     ; leggere.
;
section .data        ; Non ci sono variabili già
                     ; inizializzate.
;
section .bss
record resb MAX_SIZE ; Memoria tampone per la lettura dei
                     ; dati.
size   resd 1        ; Quantità di byte letti
                     ; effettivamente.
;
section .text
global _start
_start:
read_write_begin:
    call  read             ; Legge dallo standard input.
    cmp   eax, 0           ; Se sono stati letti zero byte,
    jz    read_write_end   ; il ciclo termina.
    call  write            ; Scrive i byte letti nello
                           ; standard output.
    jmp   read_write_begin ; Ripete il ciclo.
read_write_end:
    jmp   exit
read:
    mov   eax, SYS_READ    ; Legge dallo standard input.
    mov   ebx, STDIN       ;
    mov   ecx, record      ;
    mov   edx, MAX_SIZE    ;
    int   0x80
    mov   [size], eax      ; Salva la dimensione letta
                           ; effettivamente.
    ret
write:
    mov   eax, SYS_WRITE   ; Scrive nello standard output.
    mov   ebx, STDOUT      ;
    mov   ecx, record      ;
    mov   edx, [size]      ;
    int   0x80
    ret
exit:
    mov   eax, SYS_EXIT    ; Conclude il funzionamento.
    mov   ebx, 0           ;
    int   0x80
```

64.17 Riferimenti

- *Intel 80386 Reference Programmer's Manual*, http://pdos.csail.mit.edu/6.828/2006/readings/i386/toc.htm
  - *80386 Instruction Set*, http://pdos.csail.mit.edu/6.828/2006/readings/i386/c17.htm

- *Using Assembly Language in Linux*, *http://asm.sourceforge.net/articles/linasm.html*
- Norman Matloff, *Introduction to Linux Intel Assembly Language*, *http://heather.cs.ucdavis.edu/~matloff/50/LinuxAssembly.html*
- H.-Peter Recktenwald, *i386-PC-Linux System Calls*, 2000, *http://wayback.archive.org/web/*/www.lxhp.in-berlin.de/lhpsysc0.html*

[1] **GNU Binutils** GNU GPL

[2] **NASM** GNU LGPL

[3] **GNU Binutils** GNU GPL

[4] **GDB** GNU GPL

[5] **DDD** GNU GPL

[6] Il formato ELF prevede altri tipi di sezione, ma quelle di uso più frequente sono rappresentate nel modello sintattico.

[7] Evidentemente, il valore di uscita viene espresso in base otto: 24_8 è uguale a 20_{10}.

[8] Dipende dal compilatore se è possibile limitare effettivamente l'uso al solo registro ***CX***.

[9] L'indirizzo di memoria da raggiungere con l'istruzione '**`CALL`**', può essere fornito in modo «immediato», attraverso l'indicazione di un simbolo, oppure con un registro o con un indirizzo di memoria. Nell'ipotesi di un registro o di un indirizzo di memoria, si intende che il contenuto del registro o della variabile in memoria vadano considerati come l'indirizzo di destinazione della chiamata.

[10] Nel caso di un valore in virgola mobile, il risultato potrebbe essere atteso dal registro ***ST0***, ma la gestione della virgola mobile non viene affrontata in questo capitolo.