

Appunti di Programmazione Assembler

Versione per studenti

Giovanni Stea

a.a. 2020/21

Ultima modifica: 25/09/2020

Sommario

1	Rappresentazione di numeri naturali ed interi	5
1.1	Numeri naturali:	5
1.2	Numeri interi:	5
2	Codifica “macchina” e codifica “mnemonica” delle istruzioni	7
2.1	Primo esempio di programma in codifica mnemonica.....	8
2.1.1	Note sulla sintassi	11
2.1.2	Lunghezza degli operandi e suffissi.....	11
3	Note sulle principali istruzioni.....	13
3.1.1	MOVE.....	13
3.2	Istruzioni aritmetiche.....	13
3.2.1	ADD	13
3.2.2	SUBTRACT.....	16
3.2.3	COMPARE	17
3.3	Moltiplicazioni e divisioni	17
3.3.1	DIVIDE.....	18
3.3.2	INTEGER DIVIDE.....	19
3.4	Istruzioni di traslazione e rotazione	20
3.4.1	SHIFT ARITHMETIC RIGHT.....	20
3.4.2	ROTATE THROUGH CARRY LEFT/RIGHT.....	21
3.5	Istruzioni di controllo	22
3.5.1	JUMP if CONDITION MET	22
3.6	Meccanismi di protezione e istruzioni privilegiate	24
4	Programmare in Assembler.....	25
4.1.1	Esempio: conto dei bit a 1 in un long	25
4.1.2	Esercizio.....	27
4.2	Controllo di flusso.....	28
4.2.1	If...then...else.....	28
4.2.2	Ciclo for	29
4.2.3	Ciclo do...while	29
4.2.4	Istruzioni LOOP/LOOPcond	30
4.2.5	Esercizio – calcolo del fattoriale.....	32
4.2.6	Esercizio: test se una stringa di bit è palindroma.....	32
4.2.7	Esempio: test di primalità	33

4.3	Sottoprogrammi e passaggio dei parametri	35
4.3.1	Dichiarazione e inizializzazione dello stack nel programma.....	37
4.3.2	Esercizio: conteggio bit a 1 in un vettore (con sottoprogramma)	38
4.3.3	Esercizio: calcolo dei coefficienti binomiali.....	39
4.3.4	Esercizio: conversione in caratteri maiuscoli	41
4.3.5	Esercizio: algoritmo di Euclide per il MCD	41
4.3.6	Esercizio: calcoli con numeri naturali.....	43
5	Istruzioni che manipolano le stringhe	46
5.1.1	Prefissi di ripetizione	49
5.1.2	Esercizi:.....	51
5.1.3	Esercizio: calcolo della media di un vettore.	51
5.1.4	Esercizio: implementazione della memcpy	52
5.1.5	Esercizio: implementazione di funzioni per stringhe di caratteri	52
6	Appendice: programmare in Assembler in modo efficiente	53
6.1	Tempo di esecuzione	53
6.2	Buone pratiche per aumentare l'efficienza dei programmi	55
6.2.1	Allineare operandi e istruzioni	55
6.2.2	Evitare moltiplicazioni e divisioni	57
6.2.3	Usare le istruzioni stringa	59
6.2.4	Evitare i salti condizionati	59
6.2.5	SET if CONDITION MET	60
6.2.6	CONDITIONAL MOVE	62
7	Appendice: istruzioni Assembler aggiuntive	65
7.1.1	ADD WITH CARRY	65
7.1.2	SUBTRACT WITH BORROW	66
7.1.3	ROTATE THROUGH CARRY LEFT/RIGHT.....	67
7.1.4	LOOP / LOOPcon.....	67
7.1.5	MOVE DATA FROM STRING TO STRING (with REPEAT)	68
7.1.6	LOAD STRING	68
7.1.7	STORE STRING (with REPEAT).....	68
7.1.8	COMPARE STRING OPERANDS (with CONDITIONAL REPEAT)	69
7.1.9	SCAN STRING (with CONDITIONAL REPEAT)	69
7.1.10	INPUT STRING (with REPEAT).....	70
7.1.11	OUTPUT STRING (with REPEAT).....	70

7.1.12	SET /CLEAR DIRECTION FLAG.....	70
7.1.13	SET if CONDITION MET.....	71
7.1.14	CONDITIONAL MOVE.....	72

Version history

- 27/09/2018: prima versione online.
- 09/10/2019: correzione di alcune imprecisioni e modifiche cosmetiche minori.
- Giugno 2020: aggiunta LOOP, istruzioni stringa, appendici 6 e 7.
- Settembre 2020: modifiche cosmetiche

1 Rappresentazione di numeri naturali ed interi

La ALU è in grado di eseguire operazioni **logiche** (AND, OR, NOT, etc.) su stringhe di bit ed operazioni **aritmetiche**, interpretando le stringhe di bit che maneggia come **numeri naturali in base 2, o come numeri interi rappresentati in complemento a 2**. La rappresentazione dei numeri naturali ed interi in complemento a 2 è già stata affrontata durante corsi precedenti, e quindi viene richiamata brevemente. Sarà inoltre oggetto di trattazione approfondita più avanti nel corso.

1.1 Numeri naturali:

Su N bit sono rappresentabili i 2^N numeri naturali nell'intervallo $[0; 2^N - 1]$. Ciascuna configurazione di N bit $b_{N-1}, b_{N-2}, \dots, b_1, b_0$ può essere vista come un numero naturale rappresentato in base 2, il numero $X = \sum_{i=0}^{N-1} b_i \cdot 2^i$. Per tale motivo, in una stringa di N bit il bit b_0 è detto **Bit Meno Significativo (LSB)**, mentre il bit b_{N-1} è detto **Bit Più Significativo (MSB)**.

Dualmente, il numero naturale X può essere codificato su una stringa di bit se si trovano le cifre della sua rappresentazione in base 2. Tali cifre si trovano **dividendo per due** successivamente il numero X , e considerando tutti i resti.

1.2 Numeri interi:

Su N bit sono rappresentabili tutti i 2^N numeri interi x compresi nell'intervallo

$$[-2^{N-1}; +2^{N-1} - 1]$$

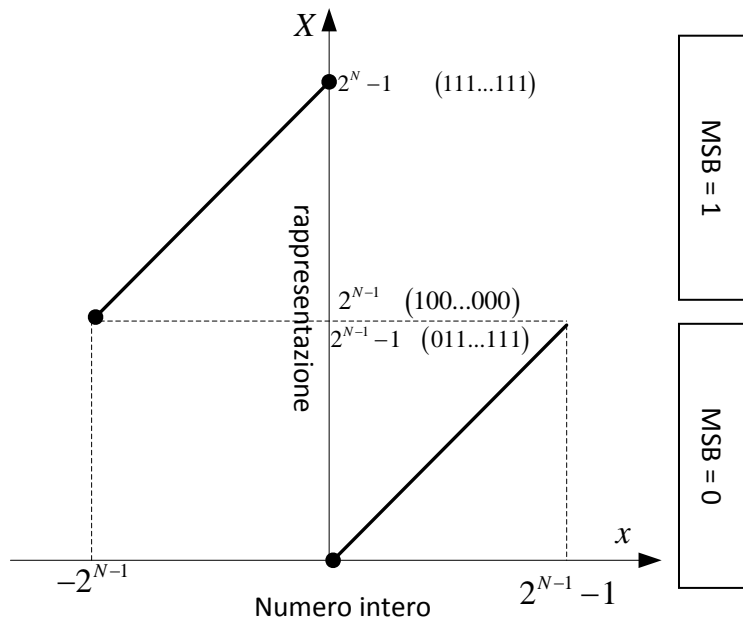
Quindi un numero intero x è rappresentabile su N bit o meno a seconda che entri o meno in questo intervallo. Su 8 e 16 bit l'intervallo è $[-128, +127]$, $[-32768, +32767]$, etc.

Il numero x viene rappresentato in C2 dalla stringa di bit che corrisponde al numero naturale X così calcolato:

$$X = \begin{cases} x & x \geq 0 \\ 2^N + x & x < 0 \end{cases}$$

Purché x sia un numero intero rappresentabile in C2 su N bit, cosa che deve essere verificata. In modo equivalente, si ha:

$$X = |x|_{2^N}$$



Quando si parla di numeri in C2, tenere **sempre** a mente questo disegno, che aiuta a ricordare come vanno le cose

Come si vede dalla figura, tutte le stringhe di bit il cui bit più significativo è “0” rappresentano numeri interi *positivi*, mentre tutte quelle che cominciano per “1” rappresentano numeri interi *negativi*. Quindi, data una stringa di bit X che rappresenta un numero intero, il numero intero x che le corrisponde è il seguente:

$$x = \begin{cases} X & X_{N-1} = 0 \\ -(\bar{X} + 1) & X_{N-1} = 1 \end{cases}$$

Il *complemento* è l’operazione che cambia gli “0” in “1” e viceversa.

Esercizio:

- rappresentare i seguenti numeri naturali su 8 bit: 32, 63, 130
- per ciascun numero x rappresentato, chiedersi quale sia la rappresentazione di $2x, 4x, \dots, 2^k x$, e quale quella del quoziente della divisione intera $x/2, x/4, \dots, x/2^k$. Ricavare una regola generale valida per i numeri naturali.
- rappresentare i numeri interi (su 8 bit in complemento a 2) -32, -1, -128, 127
- per ciascun numero x rappresentato, chiedersi quale sia la rappresentazione di $2x, 4x, \dots, 2^k x$, e quale quella del quoziente della divisione intera $x/2, x/4, \dots, x/2^k$. Ricavare una regola generale valida per i numeri interi.
- Rappresentare i numeri interi di cui alla precedente domanda su 16 bit in complemento a 2.
- Qual è la relazione tra la loro rappresentazione su 8 bit e la loro rappresentazione su 16 bit? Ricavare una regola generale per ottenere la seconda dalla prima.

2 Codifica “macchina” e codifica “mnemonica” delle istruzioni

Cominciamo a parlare di linguaggio macchina. Ci vuole una doverosa precisazione: userò **tre concetti diversi**, cercando di non creare confusione:

- codifica **macchina**: serie di zeri e di uni che codificano le istruzioni che il processore esegue
- codifica **mnemonica**: esiste un modo **simbolico** di scrivere quelle serie di zeri e di uni, che rende comprensibile a noi quello che stiamo facendo. Useremo principalmente quello
- linguaggio **Assembler** (in realtà sarebbe più corretto chiamarlo Assembly, ma ormai la consuetudine prevale sulla correttezza): è un linguaggio di programmazione, che usa una codifica mnemonica per le istruzioni, **completato da una serie di sovrastrutture sintattiche** che lo rendono più semplice da usare per un programmatore. Per esempio si possono riferire le locazioni di memoria con **nomi simbolici** nelle istruzioni.

Un’istruzione in codifica macchina per il processore x86 a 32 bit è una stringa di bit, lunga da 1 a 14 byte a seconda dei casi:

I prefix 0/1 byte	O prefix 0/1 byte	Opcode 1/2 byte	Mode 0/1 byte	SIB 0/1 byte	Displacement 0/1 byte	Immediate 0/1 byte
----------------------	----------------------	--------------------	------------------	-----------------	--------------------------	-----------------------

I campi sono sette, e possono non essere presenti tutti contemporaneamente. All’interno dei campi, i singoli bit hanno un significato ben preciso. Programmare in codifica macchina significherebbe descrivere come, per ogni istruzione del processore, questi campi devono essere riempiti. Ciò sarebbe **oltremodo noioso**, ed utile soltanto a chi dovesse implementare un programma assembler. Scriveremo in realtà qualcosa di intuitivamente più semplice (chiamato appunto **codifica mnemonica**), cioè:

```
MOV %EAX, 0x01F4E39A
```

Che significa che questa è l’istruzione con la quale spostiamo il contenuto del registro EAX nella quadrupla locazione il cui indirizzo più basso è 0x01F4E39A. Visto che il processore accetta istruzioni nella codifica macchina scritta sopra, ci vorrà qualcuno che **traduca** dalla codifica mnemonica che ho appena scritto in codifica macchina. Tale programma si chiama **assembler**. Non mi interessa descrivere **cosa faccia**, in quanto ho la garanzia che ad una istruzione scritta in codifica mnemonica corrisponderà un’istruzione in codifica macchina secondo una traduzione biunivoca. Pertanto, d’ora in poi scriveremo le istruzioni in codifica mnemonica.

Il linguaggio Assembler si spinge un passo oltre, consentendo al programmatore di riferire le locazioni di memoria con **nomi simbolici**, che l'assemblatore tradurrà in indirizzi. Ad esempio, in Assembler potrò scrivere questo:

```
MOV %EAX, pippo
```

Dove `pippo` è l'indirizzo di una locazione di memoria. L'assemblatore si incaricherà di sostituire lo stesso indirizzo tutte le volte che trova scritto `pippo`.

Visto che la codifica mnemonica è di fatto un'invenzione per rendere più semplice il compito di ricordarsi le istruzioni della macchina, esistono **diversi linguaggi Assembler** per lo stesso processore, con codifiche mnemoniche differenti. Tra questi, le **differenze sintattiche** possono essere anche importanti.

2.1 Primo esempio di programma in codifica mnemonica

Il seguente esempio descrive un programma che fa quanto segue:

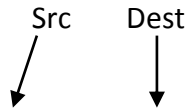
- preleva dalla memoria un operando a 32 bit, che si trova a partire dall'indirizzo `0x00000100`
- conta il numero di bit a 1 che ci sono in quell'operando
- scrive il risultato nella locazione di memoria di indirizzo `0x00000104`

Vediamo come si fa nel dettaglio:

- si porta il contenuto della memoria in `EAX` (i conti si fanno meglio sui registri)
- si azzerava un registro, ad esempio `CL`. Lo incrementeremo ogni volta che troviamo un 1 in `EAX`.
- Entriamo in un **ciclo**, nel quale facciamo scorrere verso (ad esempio) **destra** i bit di `EAX`, ed ogni volta che ne troviamo uno che vale uno incrementiamo `CL`. Il ciclo termina quando `EAX` vale zero (potremmo farlo terminare dopo 32 passi, ma così si fa prima).
- Ricopiamo il contenuto di `CL` nella cella di memoria richiesta.

Scriviamo il codice immaginandolo **contenuto in qualche locazione di memoria**, ad esempio a partire dall'indirizzo 0x00000200.

Indirizzo	Contenuto
0x00000100	
0x00000101	
0x00000102	
0x00000103	
0x00000104	
...	
0x00000200	MOVB \$0x00, %CL
0x00000202	MOVL 0x00000100, %EAX
0x00000207	CMPL \$0x00000000, %EAX
0x0000020A	JE %EIP+\$0x07
0x0000020C	SHRL %EAX
0x0000020E	ADCB \$0x00, %CL
0x00000211	JMP %EIP-\$0x0C
0x00000213	MOVB %CL, 0x00000104
0x00000218	...



Note sintattiche (valide per l'Assembler, alle quali ci uniformiamo già da ora):

- Quando figurano come operandi in qualche istruzione (e solo in questo caso) le **costanti** vengono scritte premettendo il simbolo \$. I numeri che non hanno il \$ davanti sono interpretati come **indirizzi**. La seconda istruzione riferisce **la cella di memoria di indirizzo** 0x00000100, la terza il valore costante \$0x00000000.
- i numeri sono interpretati in base 10, a meno che non siano preceduti dal prefisso 0x, nel qual caso sono esadecimali.
- i nomi dei **registri** devono essere preceduti dal simbolo di %.
- In alcune istruzioni, i letterali L e B sono i suffissi di lunghezza. La stessa istruzione (MOV) può lavorare con operandi a 8 (B), 16 (W), 32 (L) bit

Descriviamo **informalmente** il funzionamento del programma. Più avanti vedremo in modo formale le modalità di indirizzamento degli operandi e le istruzioni.

0x00000200 MOVB \$0x00, %CL
 MOV: "sposta". Istruzione equivalente all'**assegnamento** nei linguaggi ad alto livello. Assegna il byte 0x00 al registro CL. In quest'istruzione entrambi gli operandi sono ad 8 bit, donde il suffisso "B". Quest'istruzione **occupa 2 byte** (infatti, la successiva comincia a 0x00000202). Ne prendiamo atto, preannunciando che **non sarà necessario ricordarlo**.

0x00000202 MOVL 0x00000100, %EAX

Lo stesso che prima. Adesso, però, l'istruzione lavora su operandi a 32 bit (suffisso "L"). In questo caso, mette nel registro `EAX` il contenuto della (quadrupla) locazione di memoria di indirizzo `0x00000100`. Quest'istruzione **occupa 5 byte**.

```
0x00000207    CMPL $0x0000000, %EAX
```

CMP: "confronta". Confronta la costante 0 ed `EAX`. Nel farlo, scriverà qualcosa nel registro dei flag.

```
0x0000020A    JE EIP+$0x0B
```

JE: "Jump if equal". Se da un'analisi del registro dei flag (che è stato settato dal precedente confronto) è emerso che i due operandi erano uguali (cioè che il contenuto di `EAX` era zero), si prosegue ad eseguire il programma dalla cella di memoria che sta `0x0B` indirizzi più avanti. Se uno si fa il conto (che è noiosissimo), viene fuori **l'ultima** istruzione. Infatti, mentre sto eseguendo quest'istruzione, `EIP` ha **già** il valore della prossima locazione di memoria (in quanto viene incrementato subito dopo che ho prelevato un'istruzione), cioè `0x0000020C`. Se a questo valore sommo `0x0B`, ottengo `0x00000217`. Quest'istruzione e la precedente sono ciò che mi consente di **uscire dal ciclo**.

```
0x0000020C    SHRL %EAX
```

SHRL: "Shift right". Quest'istruzione trasla a destra il contenuto di `EAX`. Da sinistra (bit più significativo) viene inserito uno 0, ed il bit meno significativo finisce nel flag `CF`. A questo punto, ho due alternative: se il bit che è finito in `CF` è pari a zero, non devo sommare niente. Se il bit che è finito in `CF` è pari ad uno, devo sommare uno.

```
0x0000020E    ADCB $0x00, %CL
```

ADC: "Add with Carry". Quest'istruzione somma all'operando **destinatario** (`CL`) l'operando sorgente **ed il contenuto del flag** `CF`. (come se scrivessi: `CL += 0x00 + CF`). In questo caso, l'operando sorgente è zero, quindi somma il contenuto del flag solamente. Quindi, se il precedente shift ha messo in `CF` un bit pari ad uno, il registro `CL` viene incrementato, altrimenti no.

```
0x00000211    JMP %EIP-$0x0C
```

JMP: "Jump". Devo continuare il ciclo. Devo quindi saltare nuovamente alla prima istruzione del ciclo, quella cioè nella quale confronto `EAX` con zero. Fatti i debiti conti, e tenuto conto del fatto che `EIP` vale l'indirizzo della **successiva** istruzione, ottengo che devo tornare indietro di 12 locazioni.

```
0x00000213    MOVB %CL, 0x00000104
```

Posso arrivare qui soltanto se uscito dal ciclo. Quindi, a questo punto, CL contiene il numero di bit a 1 che erano originariamente nel registro EAX. Come da specifica, copio questo numero nella cella di indirizzo 0x00000104.

Per scrivere un programma in questo modo è necessario conoscere **la lunghezza delle istruzioni**, e farsi **i conti a mano** per i salti (noioso e pericoloso). Noi **non programmeremo in questo modo**, ma lo faremo utilizzando un linguaggio (Assembler) che è sostanzialmente **identico**, salvo che permette di specificare in modo **simbolico** gli indirizzi delle istruzioni e delle locazioni di memoria (si fa molto prima). Il programma che scriveremo dovrà essere **tradotto** (assemblato) in codifica macchina.

2.1.1 Note sulla sintassi

È bene tenere a mente quanto segue, perché è normalmente fonte di errori difficili da tracciare:

- gli operandi immediati (le costanti) **vanno preceduti dal dollaro \$**
- i registri **vanno preceduti dal simbolo di percentuale %**
- in un'istruzione, un numero non preceduto dal dollaro è **un indirizzo di memoria (indirizzamento diretto)**, non una costante.

Ad esempio:

```
MOV 0x00002000,%EDX
MOV $0x00002000,%EDX
```

Fanno due cose diverse, entrambe corrette e come tali accettate dall'assemblatore. La prima trasferisce il contenuto di una (quadrupla) locazione di memoria in EDX, la seconda mette un numero (una costante su 32 bit) in EDX.

2.1.2 Lunghezza degli operandi e suffissi

Nel formato delle istruzioni è presente (in maniera opzionale) un **suffisso di lunghezza** che specifica la lunghezza degli operandi (8,16,32). Tale suffisso è:

- B per byte (8 bit)
- W per word (16 bit)
- L per doubleword (32 bit)

Ad esempio:

```
MOVL 0x00002000,%EDX
MOVW %AX, %DX
```

Se ce lo mettete non succede niente di grave. Se non lo mettete va bene lo stesso (io non lo metto mai). Ci sono dei casi però in cui è **obbligatorio** inserire il suffisso, perché l'indirizzamento degli operandi non consente di capire di quale lunghezza si stia parlando. In particolare, quando **nessuno degli operandi è indirizzato con un indirizzamento di registro** è indispensabile mettere il suffisso (l'assemblatore segnala errore altrimenti).

Ad esempio:

```
MOV $0x2000, (%EDI)
MOV $0x10, 0x2000
```

In questo caso, visto che il destinatario è un **indirizzo di memoria** ed il sorgente è **immediato**, non sono in grado di definire in maniera univoca se si tratta di 8, 16, 32, byte. Il modo in cui esprimo l'operando immediato non serve (in quanto non determina univocamente la lunghezza in bit - potrei anche usare costanti in base 10). In questi casi il suffisso ci vuole.

Non contate sull'assemblatore per assistervi in casi del genere. Ogni tanto segnala l'errore, altre volte **no**, e il programma non funziona. L'unico modo per accorgersene è **guardare il codice disassemblato** (attivando il debugger, vedremo poi come si fa) e rendersi conto di cosa ha messo l'assemblatore, ma ci vuole una pazienza da certosino.

Attenzione a sbagliare suffisso. Ad esempio:

```
MOVB $0x2050, %AX
```

Viene tradotta dall'assemblatore dando priorità al suffisso. Molto probabilmente, verrà tradotta come se aveste scritto:

```
MOVB $0x50, %AL
```

L'assemblatore dà comunque un warning in questi casi.

In linea generale, un compilatore C++ sta molto più attento a quel che fate di un assemblatore. Non si può far conto sullo stesso livello di supporto quando si programma in Assembler.

3 Note sulle principali istruzioni

Le presenti note sono a complemento di quello che si trova sulla dispensa.

3.1.1 MOVE

Una nota da tenere **bene** a mente è che:

Non è possibile trasferire dati da una locazione di memoria in un'altra con una sola istruzione MOV.

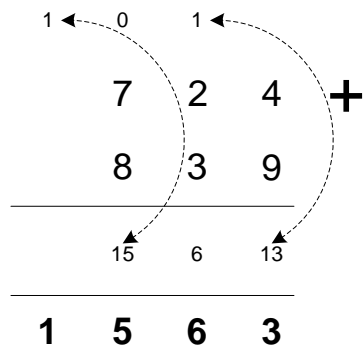
3.2 Istruzioni aritmetiche

Le istruzioni aritmetiche hanno alcune caratteristiche comuni:

- **alcune** di esse (**ma non tutte**) considerano gli operandi indifferentemente come **naturali** e come **numeri interi**, in quanto l'algoritmo che devono eseguire per modificare il destinatario è **identico** nei due casi (è una proprietà della rappresentazione in C2: **la somma delle rappresentazioni è pari alla rappresentazione della somma**, lo stesso vale per la differenza). L'istruzione di somma esegue quindi l'algoritmo giusto quale che sia l'interpretazione delle stringhe di bit che le vengono passate come operandi.
- Tali istruzioni modificano i flag (quelle di trasferimento non lo facevano). Nel modificare i flag, considerano due algoritmi: uno che va bene se gli operandi sono numeri **naturali**, ed uno che va bene se sono **interi**.
- Il programmatore è l'unico che sa cosa sta scrivendo, ed è quindi l'unico a conoscere l'interpretazione da dare ad una certa stringa di bit. Pertanto, il programmatore farà seguire una di queste istruzioni da qualche altra istruzione che va a guardare i **flag giusti**.

3.2.1 ADD

Per quanto riguarda il calcolo del risultato, è come se scrivessi `dest += src` in C++. Vediamo in dettaglio come si fa la somma. In base 2 funziona come in base 10, secondo l'algoritmo imparato alle elementari.

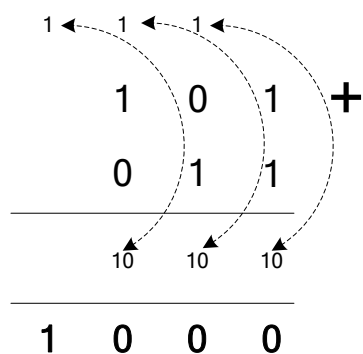


$\beta = 10$. L'algoritmo consiste in:

- sommare le cifre di pari posizione singolarmente, partendo dalle meno significative
- se la somma di due cifre non è rappresentabile con una sola cifra, usare il **riporto** per la coppia di cifre successive
- Il riporto **vale sempre 0 o 1**. Per la prima coppia di cifre (quelle meno significative), possiamo assumerlo **nullo**.

Si osservi che, visto che l'ultima somma ha generato un **riporto**, il risultato non è rappresentabile sul numero di cifre (3) degli operandi. In Inglese, riporto si dice **carry**.

Questo algoritmo **non dipende dalla base di rappresentazione**, e funziona anche in base due.



In questo caso sommo le due stringhe di bit 101 (5, se rappresentato come naturale) e 011 (3, se rappresentato come naturale) su tre bit, ed il risultato sta su 4 bit. Questo è testimoniato dalla presenza di un **carry**.

Quando faccio la somma su 8, 16, 32 bit, **l'ultimo riporto finisce in CF**. Se (nella testa del programmatore) i numeri sommati erano **naturali**, il fatto che $CF=1$ significa che il risultato che ho nel destinatario **non** è corretto, in quanto il risultato **non può** essere rappresentato sul numero di bit degli operandi.

Supponiamo adesso che le stringhe di tre bit che ho scritto siano la rappresentazione di **numeri interi in C2**. In questo caso,

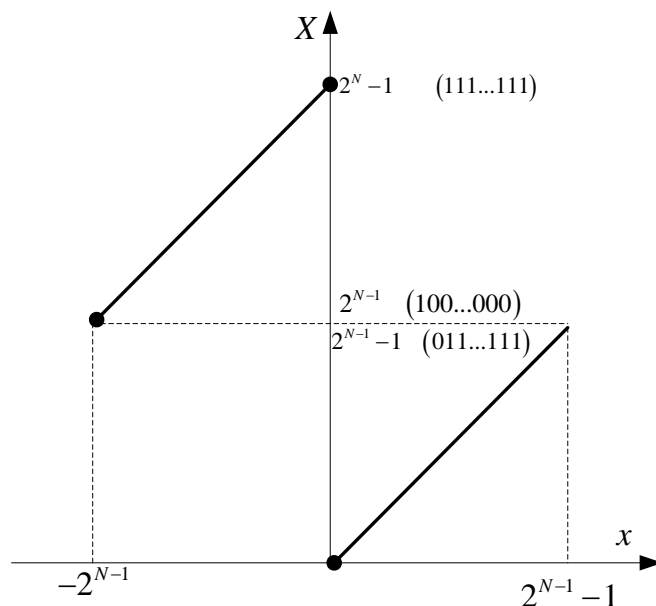
- il primo è la rappresentazione di $-(\overline{101} + 1) = -(010 + 1) = -011$, cioè di -3
- il secondo è la rappresentazione di +3

La somma dei due numeri fa **zero**. La **somma delle due rappresentazioni** su 3 bit fa anch'essa zero, come deve essere. Il fatto che ci sia un riporto non nullo è **irrilevante** ai fini della correttezza del risultato. Il risultato, rappresentato su 3 bit, è perfettamente corretto.

Infatti, C_F va guardato soltanto se i numeri sono da interpretare come **naturali**, e non come interi. Per i numeri interi, invece, il criterio per stabilire la rappresentabilità del risultato è un altro, ed è sostanzialmente basato sui **segni degli operandi**.

Sappiamo che

- in C2 il bit più significativo dà il segno: 0 se il numero è positivo, 1 se negativo.
- Un numero è rappresentabile in C2 su N bit se è nell'intervallo $[-2^{N-1}; 2^{N-1} - 1]$.

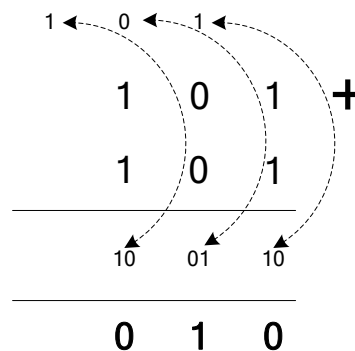
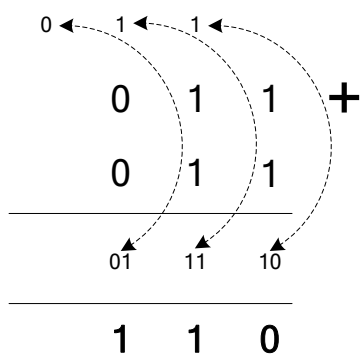


Quindi:

- se i segni degli operandi sono **discordi** (cioè uno positivo e uno negativo) e gli operandi stanno in quell'intervallo, la loro somma è sempre rappresentabile, in quanto starà per forza all'interno dell'intervallo.
- se i segni degli operandi sono **concordi**, la loro somma può non essere rappresentabile. In particolare, la somma sarà rappresentabile se e solo se la rappresentazione della somma è concorde con gli operandi.

Se sommo le rappresentazioni di due numeri positivi, posso ottenere la rappresentazione di un numero negativo. In questo caso il risultato non può essere corretto. Lo stesso, se sommo le rappresentazioni di due numeri negativi su N bit, posso ottenere la rappresentazione di un numero positivo. Nei due casi precedenti, il flag di overflow O_F viene messo a zero. Altrimenti viene messo ad 1, ad indicare che il risultato (interpretato come intero) non è corretto.

Esempio (su 3 bit). In entrambi i casi, la rappresentazione del risultato ha un segno diverso da quello degli operandi. In entrambi i casi c'è overflow, ed il bit O_F è settato.



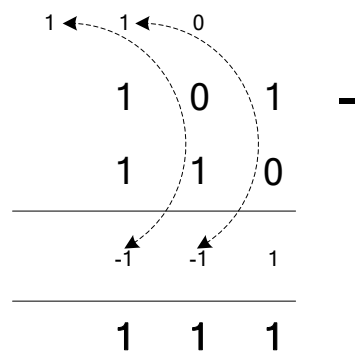
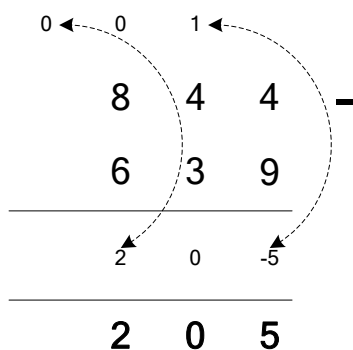
Esercizio per casa: dimostrare che le due seguenti condizioni sono equivalenti per interi rappresentati in C2:

condizione 1: gli ultimi due riporti di una somma sono identici

condizione 2: gli operandi sono discordi, oppure sono concordi ma il bit più significativo della somma è uguale a quello di entrambi gli operandi.

3.2.2 SUBTRACT

Per le sottrazioni tra numeri naturali si parla di **prestito (borrow)**, invece che di riporto. L'algoritmo è identico sia in base 10 che in base 2:



Ancora una volta, se le stringhe di bit contenute negli operandi rappresentano numeri **naturali**, per vedere se il risultato è corretto o meno bisogna andare a vedere il **carry flag** (0: il risultato è ancora un numero naturale, 1: il risultato non è un naturale). Se invece rappresentano numeri **interi**, il carry è irrilevante, e bisogna andare a vedere l'**overflow flag**. Nell'esempio appena fatto, se le due stringhe di bit rappresentano naturali il risultato non è corretto (c'è un **prestito uscente, che va nel flag CF**). Se, invece, rappresentano interi, il risultato è perfettamente corretto (e pari a -1).

Per quanto riguarda i numeri **interi**:

- la sottrazione di numeri concordi è sempre rappresentabile (non c'è mai overflow)

- la sottrazione di numeri discordi è rappresentabile soltanto se il risultato su N bit ha lo stesso segno del minuendo, altrimenti c'è overflow.

3.2.3 COMPARE

La CMP fa quanto segue:

- 1) `null=dest - source`
- 2) aggiorna i flag come farebbe la SUB

È un'istruzione fondamentale. Serve a testare due numeri, ed unita alle istruzioni di salto condizionato costituisce la coppia base per i costrutti di controllo di flusso. La CMP, di fatto, **sottrae** il sorgente dal destinatario, ma **non aggiorna il destinatario**. Si limita invece ad aggiornare il **registro dei flag** come se avesse eseguito la sottrazione. Le istruzioni di **salto condizionato**, che vedremo più in là, saltano o meno a seconda del contenuto del registro dei flag.

3.3 Moltiplicazioni e divisioni

Le istruzioni di moltiplicazione e divisione sono **diverse** per naturali ed interi, perché diversi sono gli algoritmi da seguire. Quindi c'è una MUL ed una IMUL. Sono istruzioni particolarmente laboriose per il processore (a differenza delle somme/sottrazioni). Inoltre, mentre per una somma/sottrazione il risultato sta o su N o su $N + 1$ cifre, (quindi, più o meno sullo stesso numero di cifre degli operandi), una moltiplicazione tra operandi ad N cifre ha un risultato che sta su un numero di cifre fino a $2N$ cifre. Quindi non ha molto senso fare come nelle somme/sottrazioni, cioè utilizzare (ad esempio) un registro per contenere un operando e il destinatario, in quanto i due hanno dimensione intrinsecamente diversa.

Per questo motivo, il destinatario della moltiplicazione è **implicito**, così come uno dei due **operandi**. Esistono tre tipi di moltiplicazione:

- a 8 bit: `AX = AL * source`
- a 16 bit: `DX_AX = AX * source`
- a 32 bit: `EDX_EAX = EAX * source`

Quale delle tre sia quella da svolgere viene deciso dal **numero di bit di source** (oltre che, quando serve, dal suffisso). Nei casi con 16 e 32 bit, vengono usati **due registri**, che il programmatore deve considerare come, rispettivamente, la parte alta e la parte bassa del risultato, nell'ordine con cui sono stati scritti. Per il caso a 32 bit è assolutamente necessario, perché non esistono registri a 64 bit in questo processore (e quindi l'unico modo è usarne 2 da 32 giustapposti). Per il caso a 16 bit sarebbe probabilmente stato più logico poter scrivere:

- `EAX = AX * source`

Il motivo per cui viene invece usato anche `DX` è **storico**. Funzionava così nei processori a 16 bit (prima che venissero introdotti i registri a 32 bit). Se si fosse cambiato il comportamento nei nuovi processori a 32 bit, i programmi non sarebbero più stati compatibili verso il basso (cioè tutti i programmi già scritti per processori a 16 bit non avrebbero più funzionato). Nel caso in cui serva portare il risultato di una moltiplicazione nel registro `%EAX`, basta far seguire la moltiplicazione da:

```
PUSH %DX
PUSH %AX
POP  %EAX
```

3.3.1 DIVIDE

Per questa valgono le **stesse considerazioni fatte per la MUL**, con qualche problema in più. Tanto per cominciare, una divisione non ha **un** risultato, ma **due: quoziente e resto**. Vediamo quali sono gli intervalli di rappresentabilità:

$X \text{ div } Y \rightarrow \text{quoziente } Q \text{ e resto } R$

- $0 \leq R < Y$: il resto sta per forza sullo stesso n. di bit del divisore
- $0 \leq Q \leq X$: (infatti, posso dividere anche per $Y = 1$). Quindi, in teoria, il n. di bit del quoziente dovrebbe essere pari a quello del dividendo.

In pratica, le tre versioni di divisione possibili sono queste:

- a 8 bit: `AL = quoziente (AX / source); AH = resto (AX / source);`
- a 16 bit: `AX = quoziente (DX_AX / source); DX = resto (DX_AX / source);`
- a 32 bit: `EAX = quoziente (EDX_EAX/source); EDX = resto (EDX_EAX/source);`

source (divisore)	dimensione dividendo	dividendo	quoziente	resto
8 bit	16	<code>%AX</code>	<code>%AL</code>	<code>%AH</code>
16 bit	32	<code>%DX %AX</code>	<code>%AX</code>	<code>%DX</code>
32 bit	64	<code>%EDX %EAX</code>	<code>%EAX</code>	<code>%EDX</code>

Come si concilia questo con il fatto che il quoziente può essere grande quanto il dividendo? Che succede se non c'entra? Se il quoziente non è rappresentabile sul numero di bit indicato, viene eseguita un'eccezione (la stessa che partirebbe se si fosse tentato di dividere per zero) ed il programma **si pianta**. Questa non è una condizione che può essere testata *a posteriori* guardando i flag (come per l'addizione/sottrazione), deve essere **evitato a priori**. È il **programmatore** che si deve **sincerare che il quoziente stia sul numero di bit dove la versione della DIV intende metterlo**.

Esempio:

```
MOV $3, %CL
MOV $15000, %AX
DIV %CL
```

Il quoziente è 5000, che non entra in %AL. Come fa il programmatore a sincerarsi che questo non accada? In questo caso si vede al volo, ma in un caso generale il dividendo ed il divisore potrebbero provenire dalla memoria, da registri di interfaccia, da altri conti svolti in precedenza.

Lo fa **selezionando la versione opportuna della DIV**. In questo caso, devo passare a 16 bit e scrivere:

```
MOV $3, %CX
MOV $15000, %AX
MOV $0, %DX
DIV %CX
```

Se ci si riflette un attimo, si vede che è **sempre** possibile trovare una versione della divisione che permette di dividere qualunque coppia di operandi che stia su 32 bit. Alla peggio, basta usare la divisione più grande.

Nota: Non ci sono mai problemi di rappresentabilità per il resto, visto che il numero di bit dove lo voglio mettere è sempre uguale al numero dei bit del divisore, ed il resto della divisione tra naturali è per definizione minore del divisore.

3.3.2 INTEGER DIVIDE

Fa le stesse cose della DIV ed ammette gli stessi formati. Va usata se gli operandi contengono rappresentazioni di interi invece che naturali.

Una nota sui **segni degli operandi e dei risultati**: nella divisione intera il **resto ha sempre il segno del dividendo**, ed è in modulo minore del divisore. Ciò vuol dire che il **quoziente viene sempre approssimato all'intero più vicino allo zero**. Questa cosa è difficile da accettare, perché inconsistente con le nozioni di algebra che possediamo.

Esempio:

- -7 div 3: quoziente -2; resto -1
- 7 div -3: quoziente -2; resto +1.

Note conclusive sulle istruzioni di moltiplicazione e divisione

- Va scelta con cura la versione (soprattutto per la divisione). Dipende dal problema e dalle ipotesi che posso fare.
- È **indispensabile** ricordare di azzerare DX o EDX prima della divisione, se questa è a più di 8 bit (scordarselo è un errore tipico)
- È **indispensabile** ricordare che il contenuto di DX o EDX viene **modificato** dall'operazione di moltiplicazione/divisione se questa è a più di 8 bit (altro errore tipico).

3.4 Istruzioni di traslazione e rotazione

Servono a muovere i bit dell'operando destinatario uno a uno.

Tutte hanno due formati:

OPCODE src, dest
OPCODE dest

	SX	DX
traslazione	SHL SAL	SHR SAR
rotazione	ROL RCL	ROR RCR

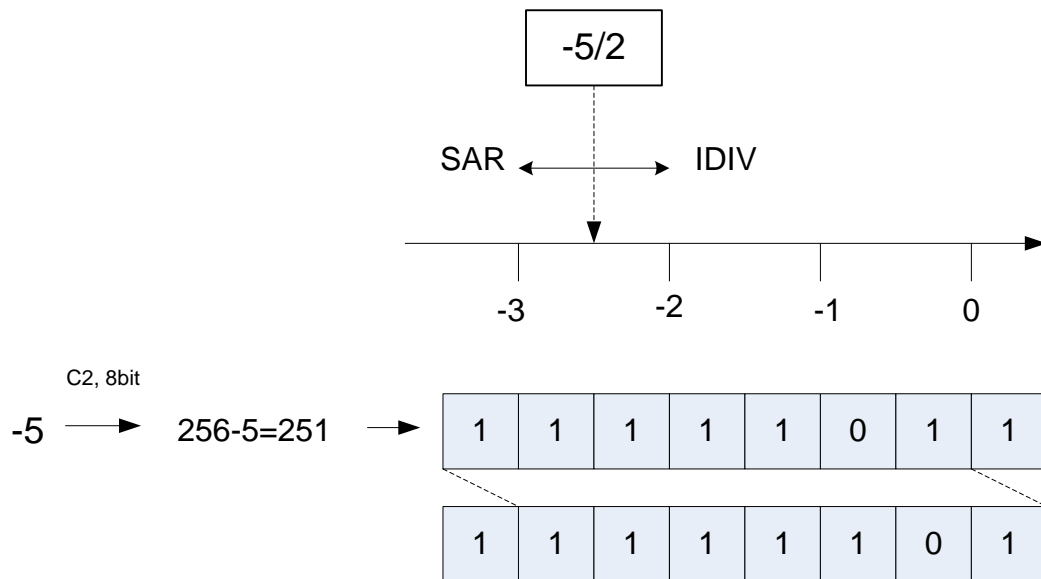
Dove `src` è il numero di volte che l'operazione `OPCODE` deve essere ripetuta sul destinatario. In più, `src` deve essere un operando immediato o il contenuto del registro `CL`, e deve valere **al massimo 31**. Nell'istruzione uno ci può scrivere cosa gli pare, ma tanto il processore considera **soltanto i 5 bit più bassi del `src`**, e niente altro, quindi scrivere 32 è come se scrivere **zero** (il che implica non fare niente di significativo). Nel formato ad un solo operando è come se si assumesse un operando sorgente implicito pari ad 1.

Si noti che in questo caso le istruzioni hanno **operandi di lunghezza diversa**. Non è un problema, visto che hanno un significato completamente diverso. Nel caso sia necessario inserire un suffisso, questo deve concordare con l'**operando destinatario**, ovviamente.

3.4.1 SHIFT ARITHMETIC RIGHT

Corrisponde al calcolo del quoziente di una **divisione per 2^{src} di un operando intero**. Infatti, invece di inserire **zeri in testa** come fa la `SHR`, **inserisce nuovamente il bit più significativo** (mantiene, cioè, il segno dell'operando). Attenzione, perché questo **non è lo stesso quoziente che calcolerebbe la `IDIV`**. Infatti, con la `SAR` abbiamo sempre approssimazione verso meno infinito del quoziente, qualora la divisione abbia resto. Nella `IDIV`, invece, l'approssimazione del quoziente è sempre verso lo zero.

Ad esempio, se uso la `IDIV` con dividendo -5 e divisore +2, ottengo $r = -1$, $q = -2$. Se faccio lo shift destro di -5 (supponendo che sia rappresentato su otto bit), invece, ottengo quanto segue:



E, per calcolare il numero intero rappresentato dal risultato, devo applicare la nota regola:

$$x = \begin{cases} X & X_{N-1} = 0 \\ -(\bar{X} + 1) & X_{N-1} = 1 \end{cases}$$

Quindi, $x = -(00000010 + 1)_{b2} = -(11)_{b2} = -3$. È necessario ricordarselo quando si fanno i conti.

3.4.2 ROTATE THROUGH CARRY LEFT/RIGHT

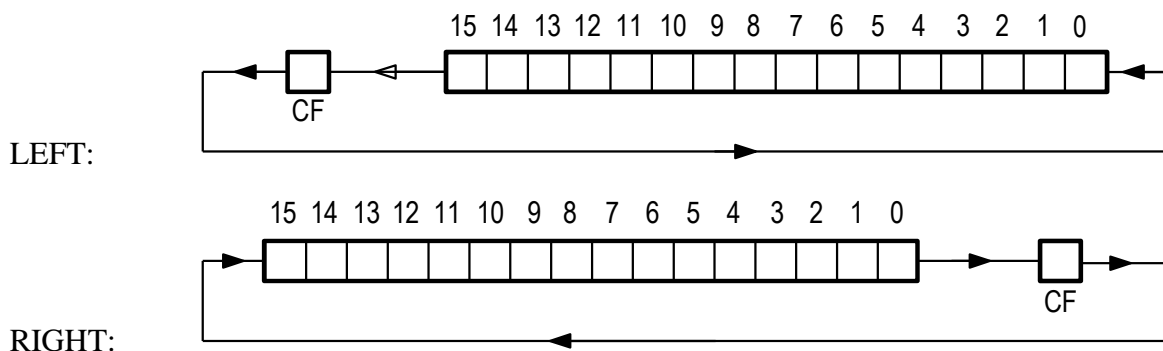
FORMATO: RCL source, destination

RCL destination

RCR source, destination

RCR destination

Fanno la stessa cosa delle corrispondenti istruzioni di rotazione ROL, ROR, coinvolgendo anche CF nella rotazione come da disegno sottostante. I formati di indirizzamento degli operandi sono identici a quelli della ROL, ROR.



3.5 Istruzioni di controllo

3.5.1 JUMP if CONDITION MET

Le istruzioni di salto condizionato **seguono sempre qualche istruzione che tocca i flag**, in quanto le condizioni che si possono specificare hanno tutte a che fare con i flag. Tipicamente (ma non esclusivamente) seguono una `CMP`. Più esattamente: dopo una `CMP` ci sta **sicuramente** un salto condizionato, in qualunque programma sensato. Visto che però anche diverse altre istruzioni toccano i flag (ad esempio, quelle aritmetiche e logiche), i salti condizionati possono stare anche dopo quelle.

I salti condizionati che vediamo sono di due tipi:

- flag a 1/0
- relazioni tra naturali / tra interi

1) **flag a 1 o a 0**: `JC/JNC`, `JS/JNS`, `JZ/JNZ`, `JO/JNO`.

Questo tipo di salti condizionati può essere usato dopo **qualunque istruzione che tocca i flag**.

Esempi di porzione di programma:

```
ADD %AX, %BX
JC ...
ist1
```

Se la somma del contenuto di `AX` e `BX` non è rappresentabile su 16 bit (interpretando i numeri come **naturali**), salta all'indirizzo specificato, altrimenti prosegue da `ist1`.

```
ADD %AX, %BX
JO ...
ist1
```

Se la somma del contenuto di `AX` e `BX` non è rappresentabile su 16 bit (interpretando i numeri come **interi**), salta all'indirizzo specificato, altrimenti prosegue da `ist1`.

2) **relazione tra naturali**: `JE`, `JNE`, `JA`, `JAE`, `JB`, `JBE`. È bene limitarsi ad usare queste **soltanto dopo una `CMP`**. In tal caso, le condizioni scritte nel codice operativo si riferiscono all'operando **destinatario** della precedente `CMP`, e sono valide **soltanto** se l'interpretazione degli operandi è quella di numeri **naturali**.

Esempio di porzione di programma:

```
CMP %AX, %BX
JAE ...
ist1
```

Se il numero **naturale** contenuto in `BX` è maggiore o uguale di quello in `AX`, salta all'indirizzo specificato, altrimenti prosegue da `ist1`.

L'esatto algoritmo che consente di vedere, dai flag, se un numero è maggiore/minore dell'altro (interpretando la sua rappresentazione o come naturale o come intero), è noioso a descriversi (e di scarsa utilità pratica). Facciamo soltanto un esempio per capire che è sempre possibile.

Jump if Below or Equal: testa se, nella precedente CMP, $dest \leq src$, interpretando le stringhe di bit come numeri naturali. Come si fa questa cosa? La CMP sottrae `src` da `dest`, e quindi:

- se il risultato è **nullo**, va bene. Ma se è nullo `ZF` contiene 1
- se la sottrazione ha generato **prestito**, va bene (perché vuol dire che il destinatario era minore, sempre stando all'interpretazione delle due stringhe di bit come naturali). Ma in questo caso `CF` contiene 1

La condizione è quindi: $CF=1 \text{ OR } ZF=1$.

3) **relazione tra interi:** `JE`, `JNE`, `JG`, `JGE`, `JL`, `JLE`. È bene limitarsi ad usare queste **soltanto dopo una CMP**. In tal caso, le condizioni scritte nel codice operativo si riferiscono all'operando **destinatario** della precedente CMP, e sono valide **soltanto** se l'interpretazione degli operandi è quella di numeri **interi**. Si noti che le due - `JE`, `JNE` - sono valide sia per interi che per naturali, visto che fanno il confronto bit a bit e che la rappresentazione è unica.

Esempio di porzione di programma:

```
CMP %AX, %BX
JLE ...
ist1
```

Se il numero **intero** contenuto in `BX` è minore o uguale di quello in `AX`, salta all'indirizzo specificato, altrimenti prosegue da `ist1`.

L'esatto algoritmo che consente di vedere, dai flag, se un numero è maggiore/minore dell'altro (interpretando la sua rappresentazione o come naturale o come intero), è noioso a descriversi (e di scarsa utilità pratica). Anche in questo caso riportiamo un esempio per vedere che è possibile.

Jump if Greater or Equal: testa se, nella precedente CMP, $dest \geq src$, interpretando le stringhe di bit come numeri **interi**. Come si fa questa cosa? La CMP sottrae `src` da `dest`, e quindi:

- se la differenza è **rappresentabile** $OF=0$. La condizione è vera se la differenza è positiva ($SF=0$)
- se la differenza **non è rappresentabile** $OF=1$, il bit del segno è certamente sbagliato. Quindi, la condizione è vera quando la differenza è negativa ($SF=1$)

La condizione è quindi: $SF=OF$.

Il buono è che non c'è bisogno di ricordarsi gli algoritmi, **basta conoscere i codici mnemonici delle istruzioni** (e quelli sono facili da ricordare).

3.6 Meccanismi di protezione e istruzioni privilegiate

Non tutte le istruzioni descritte possono essere usate **sempre**. Ciò avviene per ragioni che saranno chiare dopo aver seguito corsi di Calcolatori Elettronici o Sistemi Operativi, correlate alla **protezione**. Il processore può funzionare in due modalità: **utente** e **sistema**. I programmi del sistema operativo girano in modalità sistema, quelli scritti da un programmatore girano normalmente in modalità utente. In modalità sistema, i programmi possono utilizzare l'intero set di istruzioni (cioè tutte quelle che abbiamo visto, più altre che non vedremo per motivi di tempo). In modalità utente, **non tutte le istruzioni** possono essere usate. In particolare, non si possono usare:

- quelle di ingresso/uscita, **IN** e **OUT**.
- La **HLT**

Tali istruzioni sono dette **privilegiate**, perché possono essere eseguite solo in modalità sistema. Se si inseriscono istruzioni **IN** e **OUT** in un programma in modalità utente, parte un'**eccezione di protezione**, che passa il controllo al sistema operativo. Windows normalmente **ignora** quest'eccezione, e quindi il programma va avanti **come se non le aveste scritte**. Se fa qualcosa di sensato (dipende, spesso, da quale porta di I/O si sta indirizzando), è perché va in esecuzione un sottoprogramma che, in qualche modo, rimedia facendo altre cose e fa vedere al programmatore un risultato simile a quello che avrebbe ottenuto se avesse eseguito veramente quelle istruzioni. In ogni caso, si può star certi che non è stata eseguita una **IN** o una **OUT** (ad esempio, il numero di cicli di clock trascorsi sarà notevolmente superiore di quello che trovereste sui *data sheet*).

Se provate a scrivere una **HLT**, viene lanciata un'eccezione di protezione, che normalmente fa terminare il vostro programma. Tale istruzione, ovviamente, **non** inchioda il processore, ma ripassa il controllo al sistema operativo.

Il motivo per cui certe istruzioni vengono nascoste è che **potrebbero essere usate male**. In particolare, potrebbero bloccare il processore o rendere inconsistente lo stato delle interfacce, rendendole quindi inutilizzabili dagli altri programmi. Per accedere a queste, è bene che vi limitiate ad usare **i servizi che il sistema operativo** mette a disposizione degli utenti (e che poi vedremo).

La mancanza di istruzioni di ingresso/uscita, invece, rende il calcolatore non utilizzabile. In realtà, per l'ingresso/uscita si utilizzano altri meccanismi, cioè dei **servizi** (sottoprogrammi) che il sistema operativo mette a disposizione.

4 Programmare in Assembler

Per quanto riguarda le **parole chiave del linguaggio** (istruzioni, direttive), l'Assembler è **case-insensitive**. Anche i nomi dei registri possono essere scritti indifferentemente maiuscoli e minuscoli. I **nomi simbolici** definiti dall'utente (ad esempio `_main`) sono invece **case-sensitive**. Un buono stile di programmazione (al quale mi attengo) è quello di scrivere le keyword ed i registri con lettere **maiuscole** e tutto il resto in minuscolo.

Partiamo con un esempio ormai noto.

4.1.1 Esempio: conto dei bit a 1 in un long

Questo è esattamente lo stesso programma che ho scritto a suo tempo in codifica mnemonica. In Assembler, il codice è scritto in questo modo, ed è di gran lunga più semplice.

#conteggio dei bit a 1 in un long		
.GLOBAL _main		
.DATA		
dato:	.LONG 0x0F0F0101	0x00000100
conteggio:	.BYTE 0x00	0x00000101
		0x00000102
		0x00000103
		0x00000104
.TEXT		
_main:	NOP	...
	MOVB \$0x00,%CL	0x00000200 MOVB \$0x00, %CL
	MOVL dato, %EAX	0x00000202 MOVL 0x00000100, %EAX
comp:	CMPL \$0x00,%EAX	0x00000207 CMPL \$0x00000000, %EAX
	JE fine	0x0000020A JE %EIP+\$0x07
	SHRL %EAX	0x0000020C SHRL %EAX
	ADCB \$0x00, %CL	0x0000020E ADCB \$0x00, %CL
	JMP comp	0x00000211 JMP %EIP-\$0x0C
fine:	MOVB %CL, conteggio	0x00000213 MOVB %CL, 0x00000104
	RET	0x00000218 ...

Come struttura generale, **una riga di codice Assembler** è fatta in questo modo:

```
nome:            KEYWORD operandi    #commento [\CR]
```

Dove alcuni campi possono mancare. Il commento deve stare su una singola riga. Si possono avere

- righe di solo commento
- righe con solo KEYWORD operandi
- righe con solo nome

L'unica cosa che non va dimenticata è il ritorno carrello a fine riga.

Commento dettagliato:

```
dato:          .LONG 0x0F0F0101
conteggio:     .BYTE 0x00
```

Dichiarazioni di variabile (direttive).

- La prima è un LONG, cioè uno spazio da 4 locazioni contigue, che contengono (inizialmente) il numero 0x0F0F0101 secondo le convenzioni note. L'indirizzo della prima locazione è riferibile nel programma con il nome `dato`.
- La seconda è un BYTE, cioè uno spazio da 1 locazione, che contiene (inizialmente) il numero 0x00. L'indirizzo di tale locazione è riferibile nel programma con il nome `conteggio`.

Andando avanti nel programma, trovo:

```
MOVL dato, %EAX
```

Questo è un caso di **indirizzamento diretto**, in cui l'indirizzo della (prima) locazione è stato sostituito dal nome simbolico.

```
comp:          CMPL $0x00,%EAX      ...
               JMP comp
```

In questo caso assegno all'istruzione `CMPL` un nome simbolico, che posso riferire dentro la successiva `JMP`. Questo viene tradotto dall'assemblatore come se fosse un **salto relativo** (vedere codifica mnemonica a destra). L'aspetto positivo è, ovviamente, che non sono tenuto a farmi i conti per poterlo scrivere.

```
               JE fine
               ...
fine:          MOVB %CL, conteggio
```

Stessa cosa di prima. Attenzione ad una sottigliezza. Il nome `fine` **non è stato dichiarato al momento del suo utilizzo**. In Assembler, **i nomi possono essere usati prima di essere stati definiti**. Ci pensa l'assemblatore a strigare il tutto (fa due passate: nella prima controlla che i nomi riferiti ci siano tutti, nella seconda fa la traduzione vera e propria). Il motivo per cui ciò è necessario è ovvio: altrimenti non sarebbe possibile scrivere codice con **salti in avanti**, come quello che c'è in questo programma. Dall'altra parte, questo consente di scrivere programmi **di devastante incomprensibilità**. Ad esempio, nessuno mi obbliga a mettere tutte le dichiarazioni di variabili raggruppate in cima, né a mettere la definizione delle costanti prima del loro primo utilizzo, magari insieme alle dichiarazioni di variabile. Questi stili di programmazione **vanno evitati**. La regola è: **prima** le

dichiarazioni di costante/variabile, tutte insieme, **nell'ordine in cui le accetterebbe un compilatore C++, poi** la parte codice.

L'unico caso in cui è lecito (stilisticamente) usare un'etichetta non ancora definita è quello di **salto in avanti nel codice**. Non va fatto in nessun altro caso.

È inoltre perfettamente lecito scrivere:

```
nome1:      [\CR]
nome2:      KEYWORD operandi    # commento [\CR]
```

Con il che la stessa variabile (o la stessa istruzione) può essere riferita con entrambi i nomi `nome1`, `nome2`.

4.1.2 Esercizio

Scrivere un programma che legge una stringa di memoria lunga un numero arbitrario di caratteri (ma terminata da `\0`), inserita in un buffer di memoria di indirizzo noto, e conta le volte che appare il carattere specificato dentro un'altra locazione di memoria. Il risultato viene messo in una terza locazione di memoria.

```
.GLOBAL _main

.DATA
stringa:    .ASCIZ "Questa e' la stringa di caratteri ASCII che usiamo
             come esempio"

lettera:    .BYTE 'e'
conteggio:  .BYTE 0x00

.TEXT
_main:      NOP
             MOV $0x00, %CL
             LEA stringa, %ESI
             MOV lettera, %AL
comp:       CMPB $0x00, (%ESI) ←
             JE fine
             CMP (%ESI), %AL
             JNE poi
             INC %CL
poi:        INC %ESI
             JMP comp
fine:       MOV %CL, conteggio
             RET
```

Fondamentale: che succede se mi scordo la B nella **CMPB**?
Succede che l'**assemblatore non segnala niente, e ci mette una L** (vedere il disassemblato per rendersene conto). In questo modo il programma **non funziona** (infatti, prende sempre una lettera in più, perché straborda nella locazione successiva `lettera`).

Variazione sul tema per l'indirizzamento (uso di displacement + registro di modifica):

```
             MOV $0, %ESI
             MOV lettera, %AL
comp:       CMPB $0x00, stringa(%ESI)
             JE fine
             CMP stringa(%ESI), %AL
```

4.2 Controllo di flusso

Il linguaggio Assembler non ha costrutti di controllo di flusso di alto livello come il C++ (cicli, o `if...then...else`). Le uniche istruzioni di controllo che possono essere usate (a parte quelle per le chiamate di sottoprogramma, che non servono a questo scopo) sono salti e salti condizionati. Pertanto, sia i costrutti condizionali `if...then...else` che i cicli vanno scritti in termini di queste istruzioni.

4.2.1 If...then...else

Se voglio scrivere una porzione di codice come:

```
if (%AX<variabile)           // interpretati come contenitori di naturali
    {ist1; ...; istN;}
else
    {istN+1; ...; istN+M;}
ist_nuova;
```

Invece dell'`if`, in Assembler posso scrivere `CMP+JCond`:

```
CMP variabile, %AX
JB label           # JB consistente con i naturali
```

Il trucco (banale) è il seguente: **invertire il ramo** `else` **con quello** `then`, e scrivere il codice come segue:

```

    CMP variabile, %AX
ramoelse:    JB ramothen           # JB consistente con i naturali
             istN+1
             ...
             istN+M
             jmp segue
ramothen:    ist1
             ...
             istN
segue:       ist_nuova
```

Attenzione: in Assembler la `CMP` ha come oggetto il destinatario. Nelle `JCond` i termini `B`, `BE`, `A`, `AE`, etc. si riferiscono a questo.

In alternativa, se voglio mantenere l'ordine del codice (ramo `then` prima del ramo `else`), **devo invertire la condizione**:

```

    CMP variabile, %AX
ramothen:    JAE ramoelse          # JAE consistente con i naturali
             ist1
             ...
             istN
             JMP segue
ramoelse:    istN+1
             ...
             istN+M
segue:       ist_nuova
```

4.2.2 Ciclo for

Un qualunque ciclo può essere tradotto in un `IF` + salto a un'etichetta. Di norma, per scrivere cicli in Assembler si usa incrementare/decrementare il registro `CX` (`ECX`, `CL`) o i registri `EDI`, `ESI` (questi ultimi, infatti, vengono utilizzati per l'accesso in memoria con registro puntatore, quindi fa comodo poterli usare nei cicli per indirizzare vettori di variabili).

Supponiamo di avere il seguente spezzone di codice:

```
for (int i=0; i<var; i++)          // dove "var" è una variabile o costante
    {ist1; ...; istN}
```

In Assembler posso usare `CX` come contatore e scrivere:

```
ciclo:      MOV $0, %CX
            CMP var, %CX
            JE fuori
            ist1
            ...
            istN
            INC %CX          // e NON ADD $1, %CX (che sta su 3 byte)
            JMP ciclo
fuori:      ...
```

Se il ciclo è a decremento, si fa la stessa cosa cambiando poche righe.

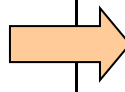
4.2.3 Ciclo do...while

In questo caso il test della condizione è in fondo al ciclo, quindi scriveremo:

```
do
    {ist1; ...; istN;}
while (AX<var);
```

In Assembler posso tradurre così:

```
ciclo:      ist1;
            ...
            istN;
            CMP var, %AX
            JB ciclo
```



Osservazione: niente mi vieta di scrivere uno spezzone di codice di questo genere:

```
ciclo:      ist1;
            ...
            istj;
            ...
            istN;
            CMP var, %AX
            JB ciclo
            ...
            JMP label1
```

Cioè di **saltare** nel mezzo di un ciclo. In C++ non lo si può fare, a meno di non usare l'istruzione **goto**, che credo non vi sia stata descritta (perché, in un linguaggio dove il controllo di flusso è strutturato, serve solo a far danni). Una prassi del genere **va evitata come la peste**, perché conduce a programmi incomprensibili ed inverificabili.

Avendo a che fare con `JMP` e `JCond` è **molto** facile farsi prendere la mano e scrivere programmi disordinati, nei quali si fanno salti e contro salti al solo scopo di risparmiare un'istruzione, con il

risultato che si scrivono programmi incomprensibili ed impossibili da testare e debuggare quando non funzionano (*spaghetti-like programming*).

I linguaggi **strutturati** (Pascal, C, etc.) sono stati inventati apposta alla fine degli anni '60 perché i linguaggi che c'erano allora (Fortran, Assembler, etc.) consentivano questo tipo di programmazione, che risultava **inverificabile** e non **debuggabile**.

La raccomandazione per un programmatore esperto di linguaggi ad alto livello è quindi la seguente: si ragioni pure in termini di costrutti di controllo di flusso C++ (if...then...else, for, while, etc.), e si traduca ciascuno di questi in Assembler nel modo sopra indicato, perché questo è uno stile di programmazione (programmazione **strutturata**) sicuro e corretto. Mentre i linguaggi ad alto livello obbligano il programmatore a tenere questo stile (forzandolo a scrivere blocchi di programma con un **unico punto di ingresso ed un unico punto di uscita**), in Assembler il programmatore deve limitarsi da solo.

4.2.4 Istruzioni LOOP/LOOPcond

Una variazione sul tema per il controllo del flusso è data dalle possibilità dell'istruzione `LOOP`. L'istruzione macchina ha come unico operando un indirizzo (che in Assembler può essere sostituito da un'etichetta specificata in modo simbolico). Vediamo un esempio:

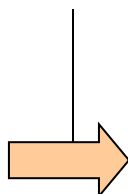
		E fa quanto segue:
ciclo:	MOV \$5, %ECX	- decrementa ECX
	ist1	- non tocca i flag (come farebbe una normale istruzione <code>DEC %ECX</code>)
	...	- se (dopo il decremento) <code>ECX != 0</code> , salta all'etichetta
	istN;	- altrimenti continua
	LOOP ciclo	

Quindi, `ECX` va inizializzato prima del ciclo al **numero di iterazioni desiderate**. Una prassi a cui attenersi rigorosamente è quella di **non toccare** `ECX` **durante il ciclo**, altrimenti si scrivono programmi che non si capisce cosa facciano. Un utilizzo classico della `LOOP` è per tradurre cicli `for`, quando:

- 1) si **conosce esattamente il numero di iterazioni da svolgere**, ed inoltre
- 2) non è necessario usare il contatore del `for` (che in questo caso sarebbe `ECX`), o lo si usa in modo discendente.

In questo caso va benissimo usare un `LOOP`:

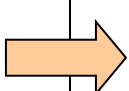
<pre>for (int i=var; i>0; i--) {ist1; //op. che usa i ...; }</pre>	<pre>istN;}</pre>
---	-------------------



ciclo:	MOV var, %ECX ist1 ...		# ECX usato per i istN LOOP ciclo
--------	------------------------------	--	---

In questo caso usare un LOOP è meno efficiente:

for (int i=0; i<var; i++) {ist1; //op. che usa i ...; istN;}		ciclo:	MOV var, %ECX MOV \$0, %EBX ist1 ... # EBX usato per i INC %EBX istN LOOP ciclo
--	--	--------	--



Ma, se devo usare un altro registro per fare il conto, allora posso usarlo anche per testare il ciclo e non ho più bisogno di usare LOOP e tenere impegnati due registri inutilmente.

Esistono anche versioni condizionali della LOOP, che	ciclo:	ist1 ... istN CMP <src>, <dest> LOOPcond ciclo
---	--------	--

si scrivono appendendo i suffissi E, Z, NE, NZ. In questo caso, il paradigma di programmazione è il seguente:

E il salto avviene se 1) la condizione `cond` è vera, e 2) se ECX non è nullo dopo il decremento. I loop condizionali funzionano **solo per il flag ZF** (infatti le condizioni E e Z sono equivalenti).

Questo è infatti il **motivo per cui la LOOP decrementa %ECX senza toccare i flag**: deve infatti preservare lo stato dei flag settati dall'istruzione precedente (CMP), altrimenti non è possibile usare la versione condizionale.

Attenzione: Il formato della LOOP prevede un salto relativo, cioè un operando `%EIP ± displacement`. In questo caso, però, il displacement può stare soltanto **su 8 bit**, e quindi non si possono fare cicli LOOP troppo lunghi (poche decine di istruzioni al massimo). Questa non è una limitazione pesante (in genere i loop che si scrivono son piccoli). Se dovessero servire dei cicli lunghi, basta usare una Jcond invece della LOOP.

Per gli scopi visti finora, le istruzioni LOOP non sono indispensabili: possono infatti essere sostituite da costrutti già visti (salti e decrementi). In realtà in Assembler esistono altre istruzioni (quelle **per la manipolazione delle stringhe**) che fanno qualcosa di simile ad un ciclo for, utilizzando ECX come contatore ed i registri EDI, ESI come registri puntatore a zone di memoria interpretate come vettori. Le vedremo più avanti.

4.2.5 Esercizio – calcolo del fattoriale

Si scriva un programma che calcola il fattoriale di un numero naturale (da 0 a 9) contenuto nella variabile `dato`, di tipo `byte`. Il risultato deve essere inserito in una variabile `risultato`, di dimensione opportuna. Si controlli che `dato` non ecceda 9. Prestare attenzione al dimensionamento della moltiplicazione.

```
.GLOBAL _main

.DATA
numero:      .BYTE 9
risultato:    .LONG 1

.TEXT
_main:       NOP

            MOV $0, %ECX
            MOV $1, %EAX
            MOV numero, %CL
            CMP $9, %CL
            JA fine
            CMP $1, %CL
            JBE fine

ciclo_f:     MUL %ECX
            DEC %CL                      # anche LOOP ciclo_f
            JNZ ciclo_f

fine:        MOV %EAX, risultato
            RET
```

Commento: devo fare una moltiplicazione, e quindi devo dimensionare il numero di bit. La moltiplicazione più grande che devo fare ha un operando a 32 bit ($9!/2$ sta su 32 bit). Quindi conviene stare su moltiplicazioni a 32 bit. Avrei potuto fare il ciclo diversamente, cominciando da 2 e andando verso l'alto. In tal modo, l'ultima moltiplicazione ($8! * 9$) ha operandi a 16 bit. Però avrei avuto il risultato in `DX_AX`, che è leggermente più scomodo da gestire (avrei dovuto spendere più tempo a riportarlo in memoria). Per portare in un registro a 32 bit un risultato che sta su due registri (ad esempio, `DX_AX`), posso fare così:

```
PUSH %DX
PUSH %AX
POP %EAX
```

4.2.6 Esercizio: test se una stringa di bit è palindroma

Scrivere un programma che si comporta come segue:

1. prende in ingresso un numero a 16 bit, contenuto in memoria nella variabile "numero".

- controlla se "numero" è o meno una stringa di 16 bit palindroma (cioé se la sequenza di 16 bit letta da sx a dx è uguale alla sequenza letta da dx a sx).
- Se X è (non è) palindromo, il programma inserisce 1 (0) nella variabile a 8 bit "palindromo", che si trova in memoria

```
.GLOBAL _main
.DATA
numero:      .WORD 0xF18F
palindromo:   .BYTE 1      # scommetto sul risultato positivo.

.TEXT

_main:        NOP
              MOV numero, %AX
              MOV $8, %CL
              MOV $0, %BL

ciclo:        RCL %AH      # metto i bit di AH in BL in ordine inverso
              RCR %BL      # usando il carry come appoggio
              DEC %CL      # anche LOOP ciclo
              JNZ ciclo    # (purché inizializzi %ECX invece di %CL)

              CMP %AL, %BL
              JE termina
              MOVB $0, palindromo

termina:      RET
```

4.2.7 Esempio: test di primalità

Scrivere un programma che si comporta come segue:

- prende in ingresso un numero a 16 bit, contenuto in memoria nella variabile numero.
- controlla se numero è o meno un numero primo. Se lo è, mette in primo il numero 1. Altrimenti mette in primo il numero 0.

```
.GLOBAL _main

.DATA
numero:      .WORD 39971
primo:       .BYTE 1

.TEXT
_main:        NOP
              MOV  numero, %AX
              CMP  $3,%AX
              JBE  termina

# AX contiene il numero N su 16 bit. BX contiene il divisore. BX va
# inizializzato a 2 e portato, al piu', fino a N-1.
```

```

        MOV    $2,%BX

ciclo:   MOV    $0,%DX        #[DX,AX] contiene il numero N su 32 bit
        PUSH  %AX            # salvo AX viene sporcato dalla divisione
        DIV   %BX
        POP   %AX            # si ripristina AX

        CMP   $0,%DX        # DX contiene il resto della divisione
        JE    nonprimo      # il numero ha un divisore

        INC   %BX
        CMP   %AX,%BX
        JAE   termina

# Una finezza: visto che il numero da dividere sta su 16 bit, se non
# è primo ha un divisore che sta su 8 bit (teorema di Gauss).
# Quindi, quando BH è diverso da 0, posso terminare il ciclo.

        CMP   $0, %BH
        JNE   termina
        JMP   ciclo

nonprimo: MOVB  $0,primo

termina:  RET

```

Altra versione dello stesso programma, più efficiente: si testa la divisione per due fuori dal ciclo (basta guardare il LSB di AX) , si parte testando 3 come primo divisore, e si saltano tutti i divisori pari sommando due al divisore.

```

# test di primalita' (2)

.GLOBAL _main

.DATA
numero:      .WORD 39971
primo:       .BYTE 1

.TEXT

_main:       NOP

        MOV   numero, %AX
        CMP   $3,%AX
        JBE   termina

# testo subito la divisibilità per due

        RCR   %AX
        JNC   nonprimo
        RCL   %AX

# AX contiene il numero N su 16 bit. BX contiene il divisore. BX va
# inizializzato a 3 e portato, al piu', fino a N-1.

```

```

MOV    $3,%BX

ciclo:  MOV    $0,%DX      #[DX,AX] contiene il numero N su 32 bit
        PUSH  %AX        # salvo AX (viene sporcato dalla DIV)
        DIV   %BX
        POP   %AX        # ripristino AX

        CMP   $0,%DX      # DX contiene il resto della divisione
        JE    nonprimo    # il numero ha un divisore

        ADD   $2,%BL      # sfrutto il teorema di Gauss: se il
        JC    termina     # divisore non sta su 8 bit, ho finito.
        CMP   %AX,%BX
        JAE   termina
        JMP   ciclo

nonprimo:  MOVB $0,primo

termina:   RET

```

4.3 Sottoprogrammi e passaggio dei parametri

Abbiamo visto le istruzioni per la gestione di sottoprogrammi, `CALL` e `RET`. Un sottoprogramma, generalmente, opera su dei parametri. Visto che la `CALL` e la `RET` non prevedono il passaggio di parametri alla chiamata, né il ritorno di un valore al ritorno da sottoprogramma, è necessario che **il programmatore** stabilisca una **convenzione** tra sottoprogramma chiamante e chiamato per gestire questi aspetti. Ci sono, sostanzialmente **due modi** di gestire i parametri (d'ora in avanti parlo genericamente di "parametri", includendo in essi il valore di ritorno del sottoprogramma):

- 1) usare variabili (i.e. locazioni di memoria) condivise
- 2) usare i registri

Ovviamente, le due modalità possono essere usate in concorso. Ricordiamo che sia i registri sia le locazioni di memoria possono contenere **indirizzi**, quindi è possibile usare sia locazioni di memoria che registri per passare *indirizzi* ad altre zone di memoria. In Assembler **non esiste il concetto di variabile locale ad un sottoprogramma**. Tutte le variabili (cioè la memoria indirizzabile) sono **globali**. Non esistono regole di scopo. La memoria è accessibile da qualunque sottoprogramma, in qualunque punto.

Chi scrive un sottoprogramma **deve** specificare (con dei commenti) quali sono i parametri che il sottoprogramma richiede, **dove** vuole trovarli, **come** passerà all'indietro un valore di ritorno e **cosa** questo significhi.

Esempio:

```

MOV ..., %AX      # preparazione dei parametri
MOV ..., %EBX     # per la chiamata di sottoprogramma

```

```

CALL sottoprg
MOV %CX, var          # utilizzo del valore di ritorno

# sottoprogramma "sottoprog", [descrizione]
# ingresso:  %AX,  [descrizione]
#           %EBX, [descrizione]
# uscita:    %CX,  [descrizione]

sottoprg:    ...
            ...
            MOV $. . ., %CX # preparazione del valore di ritorno
            RET

```

Fondamentale: un sottoprogramma dovrà fare dei conti. Per farli utilizzerà, in generale, dei **registri del processore** (difficilmente potrà farne a meno). A meno che questi registri non siano **dichiarati come contenitori di parametri di ritorno, non devono essere modificati dal sottoprogramma**.

Quindi, il sottoprogramma precedente va riscritto come segue:

```

sottoprg:    PUSH ...          # registro1 usato dal sottoprogramma
            PUSH ...          # registro2 usato dal sottoprogramma
            ...
            MOV $. . ., %CX # preparazione del valore di ritorno

            POP ...           # registro2 usato dal sottoprogramma
            POP ...           # registro1 usato dal sottoprogramma
            RET

```

Questo è **fondamentale**: chi scrive un programma si aspetta che **dopo** la `CALL` il contenuto dei registri **non** interessati da valori di ritorno del sottoprogramma **rimanga inalterato**. Ci deve pensare il programmatore, ed il modo per pensarci è **salvare i registri in pila** e ripristinarli. L'unico registro che fa eccezione è quello dei **flag**, che si assume che possa cambiare.

Attenti ai registri da salvare: non sempre si vedono scorrendo il sottoprogramma. Per esempio, se una `MUL` o una `DIV` possono sporcare anche `DX` o `EDX` (perché un risultato o un dividendo a 32/64 bit viene messo lì). Magari nel sottoprogramma non compare `DX` come destinatario di un'istruzione, ma ciò non implica che non sia stato sporcato.

Attenzione: per ogni `PUSH` ci deve essere una `POP` (e viceversa), altrimenti il programma va in crash. Infatti, alla `RET` l'indirizzo di ritorno viene ripescato dalla pila, e se si è messo qualcosa in pila senza toglierlo (o viceversa) l'indirizzo sarà casuale. Quindi **la pila va lasciata come è stata trovata**.

Un modo per essere sicuri di ricordarsene è scrivere sempre le seguenti tre righe **prima** di iniziare a scrivere il codice del sottoprogramma:

```
sottoprg:    PUSH ???  
            ...  
            POP  ???  
            RET
```

In questo modo, il programma non assembla finché non si è messo qualcosa di consistente al posto di “???", e quindi non è possibile dimenticarsi di salvare i registri in pila.

Il **sottoprogramma principale** (`_main`), come abbiamo già visto, è a tutti gli effetti un **sottoprogramma**. La convenzione con il programma che lo chiama è che il **registro %EAX**, alla fine dell'esecuzione, deve ritornare un codice che indichi se il programma è andato a buon fine: tale convenzione è

- 0: programma terminato in modo corretto
- Valore diverso da 0: programma terminato in modo errato.

Per questo motivo, prima della `RET` finale nel sottoprogramma principale, si è soliti mettere la seguente linea di codice:

```
XOR %EAX, %EAX
```

Che di fatto mette `%EAX` a zero (ed occupa un solo byte, invece che 5). Io non lo faccio quasi mai

4.3.1 Dichiarazione e inizializzazione dello stack nel programma

Se in un programma Assembler usiamo lo stack (o perché ci serve di salvare qualche registro, o perché lo abbiamo strutturato in sottoprogrammi), sarebbe opportuno **dichiararlo ed inizializzarlo**. Dichiararlo significa **riservare uno spazio di memoria sufficiente** nella sezione dati. Inizializzarlo significa mettere **l'indirizzo successivo all'ultima variabile nel registro `ESP`**, che punta al top dello stack. Questo va fatto all'inizio della sezione codice, prima di usare la pila. Vediamo come si fa:

```
.DATA  
...  
mystack:    .FILL 1024,4          #dichiarazione stack  
.SET        initial_esp, (mystack + 1024*4)  
  
.TEXT  
_main:      NOP  
            MOV $initial_esp, %ESP  #inizializzazione stack
```

Si noti che:

- `mystack` è un (nome simbolico di) indirizzo a 32 bit, quindi può essere usato in un'espressione
- `initial_esp` è l'indirizzo della locazione successiva all'ultimo byte di `mystack`.

Quanto deve essere grande lo stack? **Il corretto dimensionamento è a carico del programmatore.** In questo caso sono 1k doppie parole, che dovrebbero essere più che sufficienti per i nostri scopi. Se uno scrive codice altamente ricorsivo, con un uso massiccio dello stack, deve pensarci su. È opportuno osservare che, se uno vuole scrivere codice di quel genere, normalmente non lo fa in Assembler.

Nel nostro ambiente di programmazione (e **non** nell'Assembler in generale), la parte di codice relativa a dichiarazione ed inizializzazione può essere **omessa**, e viene riempita direttamente dall'assemblatore in modi da lui giudicati opportuni.

Nota finale: esiste un terzo modo di passare i parametri tra i sottoprogrammi, che è usare la **pila**. Questo verrà visto a Calcolatori (è ciò che fanno i compilatori quando traducono le chiamate di funzione C++), in quanto è piuttosto complesso. La complessità sta nel fatto che, se nella pila per ogni chiamata di sottoprogramma ci troviamo un numero **variabile** di locazioni occupate dai parametri (infatti, ogni sottoprogramma ha un numero e tipo qualunque di parametri di ingresso e di uscita) diventa **parecchio noioso e complesso fare i conti**. Se questi conti li fa il compilatore C++, poco male. Se dovete farli voi a mano, il bilancio complessità vs. utilità è decisamente sfavorevole.

4.3.2 Esercizio: conteggio bit a 1 in un vettore (con sottoprogramma)

Scrivere un programma che:

- definisce un vettore `numeri` di `enne` numeri naturali a 16 bit in memoria (`enne` sia una costante simbolica)
- definisce un sottoprogramma per contare il numero di bit a 1 di un numero a 16 bit. Tale sottoprogramma ha come parametro di ingresso il numero da analizzare (in `AX`), e restituisce il numero di bit a 1 in `CL`.
- utilizzando il sottoprogramma appena descritto, calcola il numero totale di bit a 1 nel vettore ed inserisce il risultato in una variabile `conteggio` di tipo `word`.

```
.GLOBAL _main

.DATA
.SET enne, 10
numeri:      .WORD 0,0,0,0,0,0,0,0,0,1
conteggio:   .WORD 0x00
```

```

.TEXT
_main:  NOP

        MOV $0, %ESI
        MOV $0, %CX
        MOV $0, %DX

ciclo:   MOV numeri(, %ESI, 2), %AX
        CALL conta
        INC %ESI
        ADD %CX, %DX
        CMP $enne, %ESI
        JB ciclo
        MOV %DX, conteggio
        XOR %EAX, %EAX
        RET

#-----
# sottoprogramma "conta"
# conta il n. di bit a 1 in una word
# par. ingresso: AX, word da analizzare
# par. uscita: CL, conto dei bit a 1

conta:   PUSH %AX
        MOVB $0x00,%CL
comp:    CMP $0x00,%AX
        JE fine
        SHR %AX
        ADCB $0x0, %CL
        JMP comp
fine:    POP %AX
        RET

#-----

```

4.3.3 Esercizio: calcolo dei coefficienti binomiali

Si scriva un programma che calcola e mette nella variabile di memoria `risultato` il coefficiente binomiale $\binom{A}{B}$, calcolato come $\frac{A!}{B!(A-B)!}$. Si assuma che A e B siano due numeri naturali minori di 10, con $A \geq B$, contenuti in memoria. Si ponga particolare attenzione nel dimensionare correttamente le variabili in memoria (a partire da `risultato`) moltiplicazioni e le divisioni. Si faccia uso di un sottoprogramma per il calcolo del fattoriale di un numero.

```

.GLOBAL _main

.DATA
A:      .BYTE 9
B:      .BYTE 5
A_fatt: .LONG 0
B_fatt: .LONG 0
AB_fatt: .LONG 0
den:    .LONG 0
risultato: .WORD 0

.TEXT

```

```

_main:      NOP
            MOV B, %AL
            CMP %AL, A                #2. Se A<B, termina.
            JB fine_prog

#3. Calcola il coefficiente binomiale (A B), pari a A!/(B!*(A-B)!)
#   (si tenga presente che 0!=1, e che 9!=362000).

            MOV B, %CL
            CALL fatt
            MOV %EAX, B_fatt
            MOV A, %CL
            CALL fatt
            MOV %EAX, A_fatt
            SUB B, %CL
            CALL fatt
            MOV %EAX, AB_fatt

            MOV B_fatt, %EDX        #   Calcola il denominatore B!*(A-B)!
            MUL %EDX
            MOV %EAX, den

            MOV $0,%EDX             #   Calcola A!/(B!*(A-B)!)
            MOV A_fatt, %EAX
            DIVL den

#   In EAX c'e' il quoziente della divisione, che e' il risultato che ci
#   interessa. Sta sicuramente su 16 bit, in quanto il max e' (9 4),
#   che fa 362880 / (24 * 120) = 126.

            MOV %AX, risultato
fine_prog:  XOR %EAX, %EAX
            RET

#-----
# sottoprogramma "fatt"
# calcola in EAX il fattoriale del numero passato in CL, ammesso
# che stia su 32 bit.

fatt:      MOV $1, %EAX
            CMP $1, %CL
            JBE fine_f
            PUSH %ECX                #possono essere spostate qui
            PUSH %EDX
            AND $0x000000FF, %ECX

ciclo_f:   MUL %ECX
            DEC %ECX
            JNZ ciclo_f

            POP %EDX
            POP %ECX
fine_f:    RET

```


4.3.4 Esercizio: conversione in caratteri maiuscoli

Scrivere un programma che accetta in ingresso una stringa di massimo 80 caratteri *esclusivamente* minuscoli terminata da ritorno carrello, stampa i singoli caratteri mentre vengono digitati, poi va a capo e stampa l'intera stringa a video in maiuscolo.

```
.DATA
messaggio:    .FILL 256,1,0

.TEXT
_main:        NOP
              MOV $80, %CX
              LEA messaggio, %EBX
ciclo:        CALL inchar
              CMP $0x0D, %AL
              JE dopo
              CMP '$a', %AL
              JB ciclo
              CMP '$z', %AL
              JA ciclo
              CALL outchar
              AND $0x5F, %AL           #converte in maiuscolo
              MOV %AL, (%EBX)
              INC %EBX
              DEC %CX
              JNZ ciclo
dopo:         MOVB $0x0A, (%EBX)
              MOVB $0x0D, 1(%EBX)
              CALL newline
              LEA messaggio, %EBX
              CALL outline
              CALL pause               # pausa per vedere cosa c'è sul video
              XOR %EAX, %EAX
              RET

.INCLUDE "C:/amb_GAS/utility"
```

4.3.5 Esercizio: algoritmo di Euclide per il MCD

Scrivere un programma Assembler che si comporta come segue:

1. legge da tastiera due numeri naturali A e B in base 10, sotto l'ipotesi che siano rappresentabili su 16 bit.
2. Se almeno uno dei due e' nullo, termina. Altrimenti,
3. Esegue l'algoritmo di Euclide per il calcolo del loro MCD, (riassunto di seguito), *stampando tutti i risultati intermedi*.
4. ritorna al punto 1.

L'algoritmo di Euclide per il calcolo dell'MCD tra due numeri A e B e':

```
passo 0: i=0; X(0)=A; Y(0)=B;
passo i: stampa i, X(i), Y(i).
         se X(i)=0, allora Y(i)=MCD e l'algoritmo e' terminato.
         altrimenti:
             X(i+1)=max( X(i), Y(i) ) mod min( X(i), Y(i) )
             Y(i+1)=min ( X(i), Y(i) )
             i=i+1
         ripeti
```

Esempio:

15	15120
10	4389
0) 15 10	0) 15120 4389
1) 5 10	1) 1953 4389
2) 0 5	2) 483 1953
	3) 21 483
	4) 0 21

```
.DATA
X:          .WORD 0x0000
Y:          .WORD 0x0000
```

```
.TEXT
_main:      NOP
```

#1. legge da tastiera un numero naturale a 5 cifre in base 10.

```
punto1:     MOV $'X', %AL
             CALL outchar
             MOV $':', %AL
             CALL outchar
             CALL indecimal_short
             MOV %AX, X
             CALL newline
             MOV $'Y', %AL
             CALL outchar
             MOV $':', %AL
             CALL outchar
             CALL indecimal_short
             MOV %AX, Y
```

#2. Se A=0 o B=0, termina.

```
punto2:     CMPW $0, X
             JE fine_prog
             CMPW $0, Y
             JE fine_prog
```

#3. Esegue l'algoritmo di Euclide per il calcolo del loro MCD,
in SI c'e' la variabile "i"

```
punto3:     MOV $0, %SI
ciclo:      MOV %SI, %AX
             CALL outdecimal_short
             MOV $')', %AL
             CALL outchar
```

```

        MOV $' ', %AL
        CALL outchar
        MOV X, %AX
        CALL outdecimal_short
        MOV $' ', %AL
        CALL outchar
        MOV Y, %AX
        CALL outdecimal_short
        CALL newline

punto4:  CMPW $0, X
        JE punto1

        MOV X, %AX
        MOV Y, %CX
        CMP %AX, %CX      #scambio AX e CX in modo che AX=max
        JBE dopo
        XCHG %AX, %CX

dopo:    MOV $0, %DX
        DIV %CX
        MOV %DX, X
        MOV %CX, Y

        INC %SI
        JMP ciclo

fine_prog: XOR %EAX, %EAX
        RET

.INCLUDE "C:/amb_GAS/utility"

```

4.3.6 Esercizio: calcoli con numeri naturali

Si implementi un programma Assembler che si comporta come segue:

- 1) legge con eco da tastiera due numeri naturali A e B di in base 10 (assumendo che siano rappresentabili su 16 bit) e un numero naturale N in base dieci (assumendo che sia rappresentabile su 8 bit)
- 2) se $A \geq B$ (maggiore o uguale), ovvero $N=0$ termina, altrimenti:
- 3) stampa a video, su una nuova riga la sequenza di N numeri:
 $B + (B-A), B + 2*(B-A), \dots, B + N*(B-A)$
 eventualmente terminando la sequenza in anticipo qualora il successivo numero da stampare non appartenga all'intervallo di rappresentabilità per numeri naturali su 16 bit.
- 4) ritorna al punto 1).

Esempio:

A:0013	A:0000
B:0025	B:9000
N:05	N:12
37 49 61 73 85	18000 27000 36000 45000 54000 63000

```

.DATA
A:      .WORD 0x0000
B:      .WORD 0x0000

.TEXT
_main:  NOP
inizio: CALL newline
        CALL newline

#punto 1
        MOV  '$A', %AL    #Legge A
        CALL outchar
        MOV  $':', %AL
        CALL outchar
        CALL indecimal_short
        MOV  %AX, A
        CALL newline
        MOV  '$B', %AL    #Legge B
        CALL outchar
        MOV  $':', %AL
        CALL outchar
        CALL indecimal_short
        CALL newline
        MOV  %AX, B
        MOV  '$N', %AL    #Legge N
        CALL outchar
        MOV  $':', %AL
        CALL outchar
        CALL indecimal_tiny
        CALL newline
        MOV  %AL, %CL      # CL contiene il numero di ite-
razioni
# punto 2
        CMP  $0,%CL
        JE   fine
        MOV  A,%AX
        MOV  B,%DX
        CMP  %DX,%AX
        JAE  fine

# punto 3
        SUB  %AX,%DX      # DX contiene B-A
        MOV  B, %AX
ciclo:  ADD  %DX, %AX
        JC   inizio

stampa: CALL outdecimal_short
        PUSH %AX
        MOV  $' ',%AL
        CALL outchar
        POP  %AX
        DEC  %CL
        JNZ  ciclo

```

```
# punto 4
        JMP  inizio
fine:    RET

.INCLUDE "C:/amb_GAS/utility"
```

5 Istruzioni che manipolano le stringhe

In Assembler **non esistono tipi di dati né strutture dati**. Esistono soltanto byte, word e double word. Qualunque struttura dati più complessa (ad esempio, una struttura con più campi) deve essere realizzata a mano, mettendo insieme gruppi di celle di memoria di dimensione opportuna. Il linguaggio Assembler supporta però il concetto di **vettore**. Infatti:

- 1) è possibile dichiarare vettori di variabili di una certa dimensione;
- 2) è possibile indirizzare la memoria con un indirizzamento indiretto che coinvolge fino a due registri + un displacement, con il che è abbastanza semplice utilizzare questi registri per un accesso di memoria consistente con l'indirizzamento di un vettore. Lo abbiamo fatto in uno degli esercizi precedenti.

Il tipo di dati “vettore” (di byte, word, dword) è in realtà ulteriormente supportato da un set di istruzioni, dette **istruzioni stringa**. Si faccia caso al fatto che il termine **stringa** è sinonimo di **vettore** (e non, come siamo abituati a pensare, di sequenza di caratteri ASCII). Esistono istruzioni che consentono di copiare operandi dalla memoria alla memoria, o anche **interi buffer** in modo sequenziale. Tali istruzioni utilizzano i registri indice %ESI e %EDI come puntatori in memoria, rispettivamente per l'operando sorgente (“S”) e destinatario (“D”).

Con le istruzioni di cui disponiamo adesso, per copiare un vettore di memoria da una parte all'altra dovremmo scrivere il codice in questo modo:

```
vett_sorg:    .FILL 1000,4
vett_dest:    .FILL 1000,4

[... ]
MOV $1000, %ECX
LEA vett_sorg, %ESI
LEA vett_dest, %EDI
ciclo:       MOV (%ESI), %EAX
              MOV %EAX, (%EDI)
              ADD $4, %ESI
              ADD $4, %EDI
              LOOP ciclo
```

Alternativamente:

```
MOV $1000, %ECX
ciclo: MOV vett_sorg(,%ECX,4), %EAX
       MOV %EAX, vett_dest(,%ECX,4)
       LOOP ciclo
```

che copia in ordine inverso, richiede qualche istruzione in meno ed è più efficiente. In realtà, mi interessa osservare che dobbiamo scrivere **comunque** un ciclo che comprende due MOV.

L'intero ciclo di sopra può essere invece espresso con **una sola istruzione stringa**, più un **prefisso di ripetizione** (che non è un'istruzione).

```
MOV $1000, %ECX
LEA vett_sorg, %ESI
LEA vett_dest, %EDI
REP MOVSL
```

In particolare **MOVSSuf** (e il suffisso va **sempre** specificato) è un'istruzione senza operandi, che fa due cose:

1. copia il sorgente, che assume essere indirizzato in modo indiretto usando `%ESI` come puntatore, nel destinatario, che assume essere indirizzato in modo indiretto usando `%EDI` come puntatore.
2. Modifica entrambi i registri sommando **il numero di byte specificato nel suffisso** (4, nell'esempio).

Il prefisso `REP`, a livello di linguaggio macchina, viene tradotto aggiungendo un byte all'istruzione, e serve a **decrementare `%ECX`** e ripetere se `%ECX` è diverso da zero (come farebbe la `LOOP`). Quindi, basta inizializzare correttamente `%ECX` prima, ed il ciclo viene gestito in maniera automatica.

Il guadagno di efficienza nell'usare una sola istruzione stringa con il prefisso di ripetizione è **notevole**. Il fetch viene fatto una volta sola, e viene ripetuta l'esecuzione. Peraltro questo costrutto consente di **copiare da memoria a memoria** (le `MOV` standard non possono avere due operandi in memoria). Nel caso (frequente) di copia memoria-memoria, l'efficienza è estremamente importante.

In realtà il quadro completo è **leggermente più complesso**. Le istruzioni-stringa (ce ne sono diverse, e le introduciamo un po' per volta), si basano su **un bit del registro dei flag, detto Direction Flag (DF)**. Tale bit segna la **direzione** in cui vengono copiate le stringhe: 0 vuol dire **in avanti** (cioè dall'indirizzo minore al maggiore) e 1 vuol dire **all'indietro**, cioè dall'indirizzo maggiore a quello minore. Tale flag deve essere impostato alla direzione desiderata prima di un'istruzione stringa. Infatti, a seconda della direzione, i registri puntatore `%ESI` ed `%EDI` verranno incrementati o decrementati. Le istruzioni per settare il flag di direzione sono:

- `STD` (set direction flag): imposta `DF` a 1 (quindi abilita la copia "all'indietro")
- `CLD` (clear direction flag): imposta `DF` a 0 (quindi abilita la copia "in avanti")

Nel codice scritto sopra, quindi, il loop va sostituito con:

```
CLD
REP MOVSL
```

Attenzione: un errore tipico è **scordarsi di impostare `DF`** (ad esempio dando per scontato che tale flag contenga 0). In questo caso, i programmi funzionano ogni tanto sì e ogni tanto no, e sono difficili da debuggare.

Ci sono diverse altre istruzioni che lavorano con le stringhe. Guardiamo prima cosa fanno singolarmente, e poi diamo uno sguardo al loro comportamento in presenza di prefissi di ripetizione.

`LODSsuf`: carica nel registro `%AL`, `%AX`, oppure `%EAX` (a seconda della lunghezza specificata come suffisso) il contenuto della locazione (doppia locazione, quadrupla locazione) di memoria indirizza-

ta dal contenuto del registro `%ESI`. A seconda del valore del flag `DF`, incrementa o decrementa di 1, 2, 4 il contenuto del registro `%ESI` medesimo.

STOSsuf: fa la cosa duale rispetto alla precedente. Copia il contenuto del registro `%AL`, `%AX`, oppure `%EAX` (a seconda della lunghezza specificata come suffisso) in memoria alla locazione (doppia locazione, quadrupla locazione) indirizzata dal contenuto del registro `%EDI`. A seconda del valore del flag `DF`, incrementa o decrementa di 1, 2, 4 il contenuto del registro `%EDI` medesimo.

Si faccia caso al fatto che **i registri indice impliciti usati per l'operando sorgente e destinatario sono differenti** (`%ESI` ed `%EDI`). Ciò consente di utilizzare i due registri in momenti differenti.

Ad esempio, il pezzo di codice scritto sopra poteva anche essere scritto come:

```
ciclo:      CLD
            LODSL
            # eventuale codice che modifica %EAX
            STOSL
            LOOP ciclo
```

Se scrivo così, quindi, posso inserire qui istruzioni che modificano il contenuto di `%EAX`, in modo da poter trasformare gli elementi del vettore durante la copia.

Un altro utilizzo tipico della `STOS` è quello di **ripulire una certa zona di memoria** (ad esempio, riempiendola di zeri):

```
MOV $1000, %ECX
LEA buffer, %EDI
XOR %EAX, %EAX
CLD
REP STOSL
```

Ci sono poi due utili istruzioni che servono **a confrontare stringhe in memoria** o a **trovare un elemento in una stringa** con una ricerca lineare.

CMPSsuf: confronta il contenuto delle locazioni (doppie locazioni, quadruple locazioni) indirizzate da `%ESI` (sorgente) ed `%EDI` (destinatario). A seconda del valore del flag `DF`, incrementa o decrementa di 1, 2, 4 il contenuto dei registri `%ESI`, `%EDI`.

SCASsuf: Confronta il contenuto del registro `%AL`, `%AX`, oppure `%EAX` (a seconda della lunghezza specificata come suffisso) con la locazione (doppia locazione, quadrupla locazione) di memoria indirizzata dal contenuto del registro `%EDI`. L'algoritmo usato nel confronto è identico a quello della `CMP`. A seconda del valore del flag `DF`, incrementa o decrementa di 1, 2, 4 il contenuto del registro `%EDI` medesimo.

La prima può essere usata, ad esempio, per confrontare due stringhe (ad esempio, per stabilire l'ordinamento alfabetico di stringhe di caratteri ASCII), o per controllare se due vettori sono uguali. La seconda può essere usata per cercare un particolare valore in una stringa di memoria.

Attenzione: contro ogni attesa, la `CMPS` e la `SCAS` considerano gli operandi in ordine **opposto** rispetto a quello che farebbe una `CMP`. Cioè, il test scritto nella condizione della `Jcon` si riferisce **al sorgente e non al destinatario**.

Esempi:

<code>CMPSB</code>		<code>SCASW</code>	
<code>JA</code>	<code># se (%ESI) > (%EDI)</code>	<code>JA</code>	<code># se (%EDI) > %AX</code>

Non è un gran problema, visto che normalmente si usano `JE/JNE`, per i quali il verso del confronto è irrilevante.

Ci sono infine le **versioni stringa delle istruzioni di ingresso/uscita** nello spazio di I/O.

- **INSsuf**: fa ingresso di uno, due, quattro byte dalla porta di I/O il cui offset è contenuto in `%DX`. L'operando viene inserito in memoria a partire dall'indirizzo di memoria contenuto in `%EDI`. A seconda del valore del flag `DF`, incrementa o decrementa di 1, 2, 4 il contenuto di `%EDI`.
- **OUTsuf**: copia uno, due, quattro byte, contenuti in memoria a partire dall'indirizzo di memoria contenuto in `%ESI`, alla porta di I/O il cui offset è contenuto in `%DX`. A seconda del valore del flag `DF`, incrementa o decrementa di 1, 2, 4 il contenuto di `%ESI`.

Entrambe sono protette, quindi non si possono usare in modalità utente.

5.1.1 Prefissi di ripetizione

Uno lo abbiamo già visto, ed è `REP`. Può essere usato con le seguenti istruzioni stringa: `MOVS`, `LODS`, `STOS`, `INS`, `OUTS`. Il suo utilizzo con `LODS` è del tutto privo di senso, in quanto si finisce a scrivere N volte nello stesso registro. È bene ripetere che il prefisso `REP` si applica ad **una istruzione**. Se è necessario ripetere più volte un pezzo di programma, va usata l'istruzione `LOOP`.

Con le istruzioni che – invece di spostare informazione – scandiscono stringhe alla ricerca di un matching con `EAX` o con l'elemento che ha la stessa posizione in un altro vettore, sarebbe sciocco usare il prefisso `REP`, in quanto vorrei poter condizionare il progresso del confronto o della scansione al risultato del confronto stesso. È comodo in questi casi poter usare un prefisso di ripetizione **condizionato**, cioè tale per cui mi fermo se ho trovato/non ho trovato matching.

I due prefissi `REPE` e `REPNE` continuano fintanto che è vera la condizione scritta, fino ad un massimo di `ECX` ripetizioni. Si applicano soltanto alla `CMPS` e alla `SCAS`. Più nel dettaglio, fanno quanto segue:

1. se `%ECX=0`, l'istruzione termina
2. viene decrementato `%ECX`, senza modificare i flag
3. viene eseguita l'istruzione stringa successiva al prefisso (**per intero!**)
4. se la condizione specificata nel prefisso è vera, si esegue una nuova ripetizione ripartendo dal punto 1, altrimenti si passa all'istruzione successiva.

L'istruzione che segue viene quindi ripetuta quindi **fino ad un massimo di `%ECX` volte**, ma la ripetizione continua solo **finché è vera la condizione** specificata.

Esempio: voglio trovare il primo elemento diverso nei due vettori sottostanti

```
array1: .WORD 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
array2: .WORD 1, 2, 3, 4, 7, 6, 7, 8, 9, 10
```

```
[...]
CLD
LEA array1, %ESI
LEA array2, %EDI
MOV $10, %ECX
REPE CMPSW
```

Cosa puntano `%ESI` ed `%EDI` all'uscita del ciclo? **Non** gli elementi diversi, ma quelli **successivi**. Infatti, **prima** viene eseguita la `CMPSW` (che incrementa di due `ESI` ed `%EDI`), **poi** viene testata la condizione, e si esce quando è falsa.

Il comportamento del prefisso `REP/REPcond` è **indeterminato** se tale prefisso è applicato davanti ad una istruzione diversa da quelle appena menzionate.

Nota: Perché esiste un meccanismo per mandare le ripetizioni **sia in avanti che all'indietro**? Quand'è che può far comodo lavorare in un senso o in un altro? Ci sono almeno due casi in cui l'assenza di un tale meccanismo non consente di lavorare correttamente:

- devo trovare l'**ultima occorrenza** di un certo valore in una stringa. In questo caso, devo necessariamente procedere all'indietro, partendo dal fondo.
- Devo **copiare buffer parzialmente sovrapposti**. In questo caso, a seconda di come sono sovrapposti (se il secondo sulla coda del primo, o il primo sulla coda del secondo), la copia è corretta soltanto se vado in una delle due direzioni.

5.1.2 Esercizi:

- Dati due array di 10 word in memoria, stampare il numero di elementi diversi che occupano la stessa posizione.

```
.DATA

array1: .WORD 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
array2: .WORD 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

.TEXT

_main:      NOP
            CLD
            XOR %AX, %AX
            LEA array1, %ESI
            LEA array2, %EDI
            MOV $10, %ECX

ciclo:      CMPSW
            SETNE %BL
            ADD %BL, %AL
            LOOP ciclo

            CALL outdecimal_short
            CALL newline
            CALL pause

            XOR %EAX, %EAX
            RET

.INCLUDE "C:/GAS/utility"
```

Si può anche utilizzare il prefisso REPE per rendere il tutto più efficiente, ma bisogna stare attenti alla condizione di uscita dal ciclo:

```
ciclo:      REPE CMPSW
            SETNE %BL
            ADD %BL, %AL
            CMP $0, %ECX
            JNE ciclo
```

Nel caso che i vettori siano molto grande e gli elementi diversi siano relativamente pochi, la seconda versione è più efficiente (in quanto svolge molte iterazioni direttamente dentro la REPE).

5.1.3 Esercizio: calcolo della media di un vettore.

Dato un array A di N numeri naturali su 16 bit, sia M il valor medio (approssimato per difetto) calcolato sugli elementi di A. Riempire l'array B, di identica dimensione, con numeri interi $B_i = A_i - M$. Stampare poi a video il valore *minimo* e *massimo* dell'array B. Utilizzare *esclusivamente* istruzioni stringa per accedere ai due array A e B.

5.1.4 Esercizio: implementazione della `memcpy`

Implementare un sottoprogramma che copia un buffer di memoria in un altro. Il sottoprogramma prende in ingresso `%ESI`, `%EDI`, ed `%ECX` come parametri, e questi contengono l'indirizzo sorgente, destinatario ed il numero di byte da copiare. Si presti particolare attenzione al caso di buffer parzialmente sovrapposti.

5.1.5 Esercizio: implementazione di funzioni per stringhe di caratteri

Si implementino come sottoprogrammi Assembler le seguenti funzioni della libreria `stdio.h` del C++:

- `strlen`: calcola la lunghezza di una stringa (escluso il carattere di terminazione). L'indirizzo di partenza della stringa è dato in `%EBX`, la lunghezza deve essere restituita in `%AX`.
- `strcpy`: copia una stringa in un'altra. L'indirizzo di partenza della stringa sorgente è dato in `%EBX`, l'indirizzo di partenza della stringa di destinazione è dato in `%EBP`.
- `strcmp`: confronta due stringhe secondo l'ordinamento della tabella ASCII. L'indirizzo di partenza delle due stringhe è in `%EBX` ed `%EBP` rispettivamente. Restituisce in `%AL` 0 se le stringhe sono uguali, 1 se la prima è successiva alla seconda, -1 altrimenti.
- `strchr`: trova la prima occorrenza di un carattere in una stringa. Ritorna (in `%EAX`) l'indirizzo della prima occorrenza del carattere. L'indirizzo di partenza della stringa si trova in `%EBX`. Se il carattere non viene trovato, `%EAX` deve valere 0.
- `strrchr`: trova l'ultima occorrenza di un carattere in una stringa. Ritorna (in `%EAX`) l'indirizzo dell'ultima occorrenza del carattere. L'indirizzo di partenza della stringa si trova in `%EBX`. Se il carattere non viene trovato, `%EAX` deve valere 0.

Si faccia l'ipotesi che ciascuna stringa in memoria sia terminata dal carattere nullo `\0`.

Si scriva del codice che accetta in ingresso delle stringhe da tastiera, terminate con il carattere di ritorno carrello, e testa i sottoprogrammi sopra scritti.

6 Appendice: programmare in Assembler in modo efficiente¹

Lo scopo principale dell'Assembler è quello di essere strumento di scrittura di **piccole porzioni di codice estremamente efficienti**. Pertanto è necessario occuparsi dell'efficienza di ciò che si scrive. Inoltre, quando si programma in linguaggi ad alto livello **si può contare su compilatori** che, se richiesti, si occupano di tradurre le istruzioni nel modo più efficiente possibile. Un assembler traduce in linguaggio macchina le istruzioni che scrivete voi, quindi **le ottimizzazioni vanno fatte a mano**. Vediamo quali sono i punti sui quali si può giocare per guadagnare in efficienza:

Lunghezza delle istruzioni e tempo di fetch

Le istruzioni non hanno tutte la stessa lunghezza. La lunghezza dipende **non solo dal codice operativo**, ma (più largamente) da **dove sono gli operandi**. Infatti:

Le istruzioni della ALU che hanno come operandi **soltanto registri** normalmente stanno su un byte, in quanto è possibile codificare sia il nome del codice operativo che il nome dei registri dentro un solo byte. Le istruzioni che usano operandi indirizzati in modo **immediato**, oppure **di memoria** (sia diretto che indiretto, usando un **displacement**), devono contenere tale informazione nell'istruzione, e quindi ci vogliono dei byte aggiuntivi (da 1 a 4) per codificarla. Esempio:

```
MOV $0, %EAX
XOR %EAX, %EAX
```

Fanno la stessa cosa. La prima è lunga 5 byte, la seconda 1 byte. Istruzioni più lunghe richiedono (in genere) più tempo per trasferire nel processore gli operandi. La fase di **fetch** non ha infatti un tempo nullo. Richiede una o più letture in memoria per leggere l'intera istruzione. Se un operando dell'istruzione si trova in memoria, richiede una o più **letture consecutive** per procurarselo. Anche questo tempo non è nullo.

Le informazioni sulla lunghezza delle istruzioni nei vari formati si trovano **sui manuali Intel**.

6.1 Tempo di esecuzione

Quanto costa un'istruzione in termini di tempo? Il tempo di esecuzione si misura in **cicli di clock** del processore. Sarebbe semplice rispondere se il calcolatore fosse fatto come quello che vedremo più avanti a Reti Logiche. La domanda è in realtà di **difficile risposta** in un calcolatore moderno, dove esistono una serie di meccanismi circuitali intermedi che rimescolano le carte in

¹ Si consiglia di leggere – o rileggere – questa appendice dopo aver studiato la descrizione in Verilog del processore (ultime settimane del corso di Reti Logiche).

modo tale da rendere impossibile dare una risposta definitiva. Vediamo comunque cosa si può dire sulla base delle conoscenze acquisite finora.

La fase di **esecuzione** prende un numero di clock variabile, in base al **tipo di operazione**.

Per quanto riguarda il **tipo di operazione**, esistono operazioni più o meno “difficili” per un calcolatore. La **maggior parte delle istruzioni operative della ALU** (comprese somme e sottrazioni) costa molto poco (1-3 cicli di clock). Le **moltiplicazioni e le divisioni costano molto** (10-50 cicli di clock). Per le operazioni della FPU sui **numeri reali** il discorso è analogo: le somme/sottrazioni costano poco, le **moltiplicazioni/divisioni** costano molto, le **istruzioni trascendenti costano moltissimo** (anche 100+ cicli di clock). Conviene evitare di usarle quando non sono indispensabili.

Per le **istruzioni di controllo** della ALU (`JMP`, `Jcon`, `CALL`, `RET`, `LOOP`), il discorso è diverso. Il costo in termini di esecuzione è **normalmente abbastanza alto**, ma **non necessariamente per via del numero di clock** (vediamo le ragioni esatte tra un attimo).

Detto questo, è fondamentale capire che:

non è possibile misurare la durata di un programma guardando la lista delle sue istruzioni

in quanto in un calcolatore intervengono mille meccanismi che scombinano le carte. Accenno ad alcuni dei più diffusi, parte dei quali vedrete nei prossimi insegnamenti.

- non si misura la durata di **un programma**: si misura la durata di **un processo**, cioè dell’esecuzione di un programma con certi dati di ingresso, all’interno di un certo ambiente (interfacce, memoria, sistema operativo, etc.) che si trova in un certo stato. Se **cambiate i dati** cambia il numero di iterazioni di un ciclo, la direzione presa a seguito di una `Jcon`, etc. e quindi cambia la durata del processo. Se **cambia lo stato delle interfacce**, cambia la durata del processo. Per usare una metafora, un programma è la ricetta di una torta, il processo è la torta in lavorazione. Così come non si può assaggiare una *ricetta*, non si può misurare un *programma*.
- Oltre al **vostro processo**, vanno in esecuzione mille altre cose (routine del sistema operativo, interruzioni, altri programmi). Se qualche altro processo ha **interrotto** il vostro, la durata percepita (o misurata) della vostra esecuzione cambia notevolmente.
- Nei processori moderni, **il clock non va a velocità costante** (cambia, e non di poco, a seconda del carico, per motivi di risparmio energetico), né il vostro processo esegue necessariamente sempre sullo stesso core;
- **memorie cache**: parte della memoria viene **replicata** in una memoria più piccola vicina al processore. Quando il processore può, preleva i dati e le istruzioni da lì, e ci mette molto meno tempo. Le prestazioni del vostro calcolatore dipendono in misura enorme dalla presenza (e dall’efficienza) di questo componente;

- **code di prefetch:** il processore non preleva un'istruzione per volta. In realtà preleva un blocco di N locazioni di memoria per volta. Tale prelievo avviene in tempi diversi rispetto al fetch delle istruzioni da parte del processore. Le istruzioni vengono tenute in una coda locale, detta *coda di prefetch*, dalle quali il processore le preleva.
- **esecuzione parallela:** un processore è organizzato a **catena di montaggio** (pipeline). Un'istruzione viene suddivisa **in diversi stadi di lavorazione**, e ad un dato momento sono "in lavorazione" più istruzioni contemporaneamente. Ad esempio, lo stadio che si procura gli operandi in memoria durante il fetch è diverso da quello che elabora l'istruzione, che è diverso da quello che riscrive il risultato in memoria. Ciò significa che il fetch della prossima istruzione può iniziare prima che la precedente sia terminata, **sotto opportune ipotesi**. Inoltre, la FPU e la ALU possono lavorare in parallelo.
- **esecuzione non sequenziale:** seguendo algoritmi che non conosciamo, alcuni processori **rior-
dinano le istruzioni** nella coda di prefetch per rendere l'esecuzione più efficiente (quando possono farlo), cioè per sfruttare al meglio il parallelismo. Quindi, di fatto, un'istruzione può durare un tempo o un altro a seconda dello stato della ALU/FPU in quel momento.

Per tutti questi motivi, è abbastanza assurdo pensare di ottimizzare i programmi relativamente al tempo di esecuzione. Nondimeno, alcune **regole valide in generale** per rendere i programmi Assembler più efficienti possono essere scritte. Sono relative a quattro aspetti:

- **allineamento** di operandi e istruzioni
- **evitare moltiplicazioni e divisioni** (quando si può)
- preferire **istruzioni stringa**
- **evitare salti condizionati** (quando si può), o ridurne l'occorrenza.

6.2 Buone pratiche per aumentare l'efficienza dei programmi

6.2.1 Allineare operandi e istruzioni

La ALU è in grado di leggere e scrivere in memoria operandi di 1, 2, 4 byte (la memoria RAM è organizzata in modo leggermente più complesso di come visto a Reti Logiche). Il che significa che sul bus è in grado di trasferire 8, 16, 32 bit con una sola operazione di lettura/scrittura. Il problema è che, per trasferire 16 o 32 bit con una sola lettura/scrittura in memoria, l'indirizzo di memoria deve essere **un multiplo di due o di quattro**. Altrimenti il processore ci mette **due cicli di accesso in memoria**, e perde più tempo.

Esempio:

0x12345670	
0x12345671	
0x12345672	01
0x12345673	B4
0x12345674	C6
0x12345675	75
0x12345676	
0x12345677	

```
MOVL $0x75C6B401, 0x12345672
```

Questa istruzione richiede **due accessi in memoria**: uno a 16 bit per scrivere la parte bassa in 0x12345672, uno a 16 bit per scrivere la parte alta in 0x12345674.

Per evitare questo problema (che **rallenta** l'esecuzione), gli operandi devono essere **allineati** sul numero di bit corretto. Quando dichiaro le variabili nella sezione `.DATA` di un programma, queste vengono inserite **in indirizzi di memoria consecutivi**, inizialmente **allineati a 16 byte** (per motivi di caching).

Ci sono due modi per allineare gli operandi:

- **a mano**: basta dichiarare le variabili **in ordine di lunghezza decrescente**. Non è il modo migliore di fare le cose per almeno due motivi:
 - a) l'ordine di dichiarazione delle variabili spesso **segue una logica**; se uno le scombina, poi non capisce più a cosa servono (si tenga presente che è comunque improbabile scrivere programmi in Assembler che necessitano di molte variabili);
 - b) l'accorgimento di cui sopra non serve **se scrivo programmi su più file**. Quando abbiamo programmi su più file non possiamo sapere a priori in che ordine l'assemblatore mette insieme le variabili tra più file diversi, e – anche se le mettesse in ordine di file – il fatto che ciascuna sezione `.DATA` nei file sia allineata correttamente non implica che lo sia la sezione risultante (che è l'unione delle sezioni di ciascun file).
- **Usando un'apposita direttiva**:


```
.BALIGN nbyte, fill_expr
```

Allinea ad un indirizzo multiplo di `nbyte` **la successiva variabile** (e **non** tutte le variabili da quel punto in poi), riempiendo eventuali spazi vuoti con `fill_expr`. Quest'ultima è opzionale, e vale 0 come default. Questo assicura che ciascuna variabile intera possa essere letta/scritta in un solo ciclo di accesso alla memoria.

Esempio:

```
.DATA
a:          .BYTE 5
x:          .LONG 100      # indirizzo di x = indirizzo di a + 1
y:          .LONG 40       # indirizzo di x = indirizzo di a + 5
z:          .LONG 1        # indirizzo di x = indirizzo di a + 9

.DATA
a:          .BYTE 5
.balign 4,0
x:          .LONG 100      # indirizzo di x = indirizzo di a + 4
y:          .LONG 40       # indirizzo di x = indirizzo di a + 8
z:          .LONG 1        # indirizzo di x = indirizzo di a + 12
```

Per default, l'inizio della parte `.DATA` e `.TEXT` è allineato a 16 byte, che si sposa bene con il caching dei dati.

Anche i **blocchi di istruzioni** possono essere allineati. L'allineamento delle istruzioni (ad esempio a 16 byte) è comodo perché fa sì che blocchi di codice stiano compatti in cache, riducendo il tempo di fetch.

Istruzioni scritte consecutivamente nella parte `.TEXT` stanno in memoria ad indirizzi consecutivi. Il problema è che **non posso inserire dei byte a caso tra un'istruzione e l'altra**, perché verrebbero prelevati e interpretati dal processore come istruzioni. Non posso neanche metterci dei `NOP`, perché farei perdere tempo al processore. Gli unici punti dove **ha senso (e conviene)** allineare il codice sono quelli in cui **si interrompe la sequenza**, cioè **dopo una `JMP` o dopo una `RET`**. In questo caso, ci possono essere dei byte “vuoti” nel mezzo del codice, tanto non verranno mai letti dal processore.

6.2.2 Evitare moltiplicazioni e divisioni

Le moltiplicazioni e le divisioni costano. Se si può, è bene evitarle. Un modo già visto è usare **shift** per surrogare moltiplicazioni e divisioni per potenze di due. Così come in base 10 noi:

- aggiungiamo uno zero in coda quando moltiplichiamo per 10
- buttiamo l'ultima cifra quando dividiamo per 10

In base 2 aggiungeremo uno zero in coda per moltiplicare per 2.

Un modo meno ovvio, spesso sfruttato dai programmatori, è far uso dell'istruzione **LEA** al posto di moltiplicazioni semplici, o gruppi di moltiplicazioni e somme. Di fatto, la `LEA` compie quest'operazione:

$$\text{LEA } \text{disp}(\text{base}, \text{indice}, \text{scala}), \text{dest}$$

$$\text{dest} = \text{disp} + \text{base} + \text{indice} \cdot \text{scala}$$

Dove *disp* è una costante intera, *base* e *indice* sono due registri (non necessariamente distinti), e *scala* è una costante che vale 1, 2, 4, 8. Visto che la `LEA` costa un ciclo di clock, molte operazioni che coinvolgono moltiplicazioni e somme possono essere riscritte in termini di `LEA`. Peraltro, la `LEA` opera su registri a 32 bit, e consente di selezionare **esplicitamente sorgente e destinatario**, il che ne aumenta particolarmente l'efficienza rispetto alla `MUL`.

Esempio: devo calcolare $z = 16 \cdot x + 3 \cdot y - 13500$. A suon di moltiplicazioni è abbastanza noioso.

Usando `LEA` si fa così:

```
.DATA
x:      .LONG 100
y:      .LONG 40
z:      .LONG 1

.TEXT
[...]
MOV y, %EAX
LEA -13500(%EAX, %EAX, 2), %EAX    # EAX <- 3y-13500
MOV x, %EBX
SHL $4, %EBX                      # EBX <- x*16
ADD %EBX, %EAX
MOV %EAX, z
```

Che è sicuramente più efficiente che scrivere:

```
PUSH %EDX                        # la MUL lo sporcherà'
MOV y, %EAX
MOV $3, %EBX
MUL %EBX                        # MUL non ha op. immediati
SUB $13500, %EAX
MOV %EAX, z
MOV x, %EAX
MOV $16, %EBX
MUL %EBX
ADD %EAX, z
POP %EDX
```

I compilatori dei linguaggi ad alto livello hanno delle **tabelle di corrispondenza per le moltiplicazioni intere/naturali**. Quando in C++ scrivete “ $A \cdot B$ ”, quest'espressione non necessariamente viene tradotta con una `MUL`. In genere, il compilatore cerca di usare trucchi di questo genere per guadagnare efficienza. Non è difficile osservare che tutte le moltiplicazioni per 3, 5, 9 possono essere tradotti con una singola `LEA`. Quindi, anche le moltiplicazioni per **prodotti di (pochi) fattori 2, 3, 5, 9** possono essere sostituite da sequenze di `SHL` e `LEA`. Molto spesso, moltiplicazioni per numeri $A \cdot B$ qualunque vengono scomposte in $A \cdot (B_1 + B_2 + \dots)$, dove ciascuna moltiplicazione $A \cdot B_i$ si fa velocemente con `SHL` e `LEA`. Se il numero di fattori in cui scompongo B è piccolo (2 o 3), ci guadagno comunque.

Anche i processori moderni hanno **tabelle di corrispondenza interne per le moltiplicazioni** (in genere più limitate rispetto a quelle dei compilatori). Di fatto, hanno nelle loro reti moltiplicatore dei **multiplexer** che selezionano risultati particolari (ad esempio, moltiplicazioni per zero, per uno, per due, etc.). Questo giustifica il fatto che **il numero di clock per una MUL è variabile**.

6.2.3 Usare le istruzioni stringa

Le istruzioni stringa vanno privilegiate quando si lavora con buffer di memoria, soprattutto se possono essere utilizzate con prefissi di ripetizione e **su vettori allineati** (importante!). Sono nettamente più veloci che scrivere un blocco di codice che le surroga, anche per il motivo che andiamo ad esaminare adesso.

6.2.4 Evitare i salti condizionati

I salti condizionati sono istruzioni in cui si biforca il flusso del codice. Il tempo di esecuzione di un'istruzione di salto condizionato è **diverso** a seconda del fatto che **si debba saltare o meno**. In particolare, non saltare costa quasi niente, mentre saltare costa diversi cicli di clock in più (si svuota la coda di prefetch). È pertanto una buona prassi di programmazione organizzare il codice in modo che si debba **saltare il meno possibile**, cioè che la condizione di salto sia generalmente **falsa**. Più si va in sequenza, meglio è.

Ciò detto, è bene **evitare di mettere salti condizionati** quando se ne può fare a meno. Il motivo è dovuto **all'esecuzione parallela**. Se un processore esegue non un'istruzione per volta, ma tante contemporaneamente in pipeline, quando c'è una biforcazione del flusso di controllo ha due alternative:

- **Smette di elaborare le istruzioni successive** finché la condizione non è stata valutata;
- **prende una strada, sperando che sia quella giusta** (*branch prediction*). Comincia ad eseguire uno dei due rami (in base a previsioni sue). Se poi si accorge di aver sbagliato, torna indietro e riparte per l'altro ramo.

La seconda strategia è quella che viene seguita più spesso, perché più efficiente (purché le previsioni ci indovino la maggior parte delle volte). Quale che sia la strategia adottata, un salto condizionato è qualcosa che rischia di mettere in crisi l'efficienza del processing delle istruzioni.

Se si può fare a meno di metterlo nel codice è meglio. Ovviamente, i salti condizionati **servono**, e non se ne può fare a meno totalmente. Se ne può però evitare una parte se si dispone di istruzioni che usano **i flag come operandi**. Alcune le abbiamo già viste, e sono la ADC, SBB.

Esempio: calcolo del n. dei bit a 1

<pre>#senza JC .GLOBAL _main .DATA dato: .LONG 0x0F0F0101 conteggio: .BYTE 0x00 .TEXT _main: NOP MOVB \$0x00,%CL MOVL dato,%EAX comp: CMPL \$0x00,%EAX JE fine SHRL %EAX ADCB \$0x0,%CL JMP comp fine: MOVB %CL,conteggio RET</pre>	<pre>#con JC .GLOBAL _main .DATA dato: .LONG 0x0F0F0101 conteggio: .BYTE 0x00 .TEXT _main: NOP MOVB \$0x00,%CL MOVL dato,%EAX comp: CMPL \$0x00,%EAX JE fine SHRL %EAX JNC comp INC %CL JMP comp fine: MOVB %CL,conteggio RET</pre>
---	---

Una buona parte di voi, se non avesse già visto questa parte di codice, l'avrebbe scritta come nella figura a destra. Tutto sommato è soltanto un'istruzione in più, ma scritta così impedisce di sfruttare la pipeline per l'esecuzione del programma.

Nell'esempio a sinistra, possiamo **evitare** il salto condizionato perché abbiamo un'istruzione (ADC) che usa come operando (implicito) il flag CF. Infatti, se ci pensate, i salti condizionati sono (a parte pochi altri esempi, tipo ADC, SBB, RCR, RCL) le uniche istruzioni che usano i flag come operandi (impliciti).

Altre istruzioni che fanno riferimento ai flag sono in grado di:

- **trasferire zero o uno in un registro generale** (o in un operando di memoria) a seconda della condizione (SET condizionali).
- **eseguire o meno una MOV in un registro generale** a seconda della condizione (MOV condizionali).

Con un po' di pratica, un buon numero di Jcon può essere sostituito da spezzoni di codice che contengono una di queste due istruzioni.

6.2.5 SET if CONDITION MET

FORMATO: SETcon %reg

SETcon mem

AZIONE: Setta a 1 o a 0 l'operando destinatario, a seconda se la condizione *con* specificata è vera o falsa. Le condizioni sono le stesse che possono essere scritte come salti condizionati. Il destinatario deve essere ad 8 bit.

FLAG di cui viene modificato il contenuto: Nessuno.

Di seguito sono riassunti i codici operativi **di alcune** delle istruzioni di salto condizionato e, per ciascuno di essi, è spiegato brevemente il significato della pertinente condizione *con*.

SETE	(Set if Equal) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era uguale all'operando sorgente.
SETNE	(Set if Not Equal) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizione dopo un'istruzione CMP indica che l'operando destinatario non era uguale all'operando sorgente.
SETA	(Set if Above) la condizione è soddisfatta se CF contiene 0 e ZF contiene 0; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come numeri naturali.
SETAE	(Set if Above or Equal) la condizione è soddisfatta se CF contiene 0; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri naturali.
SETB	(Set if Below) la condizione è soddisfatta se CF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come numeri naturali.
SETBE	(Set if Below or Equal) la condizione è soddisfatta se CF contiene 1 o ZF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore od uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri naturali.
SETG	(Set if Greater) la condizione è soddisfatta se ZF contiene 0 e se il contenuto di SF è uguale a quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come numeri interi.
SETGE	(Set if Greater or Equal) la condizione è soddisfatta se il contenuto di SF è uguale a quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri interi.
SETL	(Set if Less) la condizione è soddisfatta se il contenuto di SF è diverso da quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come numeri interi.
SETLE	(Set if Less or Equal) la condizione è soddisfatta se ZF contiene 1 oppure se il contenuto di SF è diverso da quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore od uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri interi.
SETZ	(Set if Zero) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione indica che il risultato dell'istruzione precedente è stato zero.
SETNZ	(Set if Not Zero) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizione indica che il risultato dell'istruzione precedente è stato diverso da zero.
SETC	(Set if Carry) la condizione è soddisfatta se CF contiene 1.
SETNC	(Set if No Carry) la condizione è soddisfatta se CF contiene 0.
SETO	(Set if Overflow) la condizione è soddisfatta se OF contiene 1.
SETNO	(Set if No Overflow) la condizione è soddisfatta se OF contiene 0.
SETS	(Set if Sign) la condizione è soddisfatta se SF contiene 1.
SETNS	(Set if No Sign) la condizione è soddisfatta se SF contiene 0.

6.2.6 CONDITIONAL MOVE

FORMATO: CMOVcon %reg, %reg
CMOVcon mem, %reg

AZIONE: esegue la corrispondente istruzione MOV se è vera la condizione specificata, altrimenti non fa niente. Sorgente e destinatario devono essere a 16 o 32 bit.

FLAG di cui viene modificato il contenuto: Nessuno.

Di seguito sono riassunti i codici operativi **di alcune** delle istruzioni di conditional move e, per ciascuno di essi, è spiegato brevemente il significato della pertinente condizione *con*.

CMOVE	(Move if Equal) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era uguale all'operando sorgente.
CMOVNE	(Move if Not Equal) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizione dopo un'istruzione CMP indica che l'operando destinatario non era uguale all'operando sorgente.
CMOVA	(Move if Above) la condizione è soddisfatta se CF contiene 0 e ZF contiene 0; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come numeri naturali.
CMOVAE	(Move if Above or Equal) la condizione è soddisfatta se CF contiene 0; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri naturali.
CMOVB	(Move if Below) la condizione è soddisfatta se CF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come numeri naturali.
CMOVBE	(Move if Below or Equal) la condizione è soddisfatta se CF contiene 1 o ZF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore od uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri naturali.
CMOVBG	(Move if Greater) la condizione è soddisfatta se ZF contiene 0 e se il contenuto di SF è uguale a quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come numeri interi.
CMOVGE	(Move if Greater or Equal) la condizione è soddisfatta se il contenuto di SF è uguale a quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri interi.
CMOVL	(Move if Less) la condizione è soddisfatta se il contenuto di SF è diverso da quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come numeri interi.
CMOVLE	(Move if Less or Equal) la condizione è soddisfatta se ZF contiene 1 oppure se il contenuto di SF è diverso da quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore od uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri interi.
CMOVZ	(Move if Zero) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione indica che il risultato dell'istruzione precedente è stato zero.
CMOVNZ	(Move if Not Zero) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizione indica che il risultato dell'istruzione precedente è stato diverso da zero.

CMOVC	(Move if Carry) la condizione è soddisfatta se CF contiene 1.
CMOVNC	(Move if No Carry) la condizione è soddisfatta se CF contiene 0.
CMOVO	(Move if Overflow) la condizione è soddisfatta se OF contiene 1.
CMOVNO	(Move if No Overflow) la condizione è soddisfatta se OF contiene 0.
CMOVS	(Move if Sign) la condizione è soddisfatta se SF contiene 1.
CMOVNS	(Move if No Sign) la condizione è soddisfatta se SF contiene 0.

Un esempio di utilizzo delle *conditional move* è la **traduzione dell'if aritmetico del C++**:

$a = (x > y) ? x : y$ # DX ← x, BX ← y, AX ← a

CMP %BX, %DX	CMP %BX, %DX
CMOVA %DX, %AX	JA dopo
CMOVBE %BX, %AX	MOV %BX, %AX
	JMP fine
dopo:	MOV %DX, %AX
fine:	[...]

Oltre che risparmiare un paio di istruzioni, si consente al sistema di tenere la pipeline di esecuzione piena, e quindi si guadagna notevolmente in efficienza.

7 Appendice: istruzioni Assembler aggiuntive

7.1.1 ADD WITH CARRY

FORMATO: `ADC source, destination`

AZIONE: Modifica l'operando destinatario sommandovi sia l'operando sorgente sia il contenuto del flag CF (il risultato dell'operazione è consistente sia se gli operandi sono interpretati come numeri naturali sia se gli operandi sono interpretati come numeri interi); mette ad 1 il contenuto del flag CF se, interpretando gli operandi come numeri naturali, si è verificato un riporto; mette ad 1 il contenuto del flag OF se, interpretando gli operandi come numeri interi, si è verificato un traboccamento. Poiché l'istruzione ADC prevede la gestione del riporto che può essere stato generato durante l'esecuzione di una precedente istruzione ADD o ADC, essa può essere usata per predisporre pacchetti di istruzioni per la somma di operandi multi_word.

Ad esempio, la somma di due operandi a 64 bit (16 cifre esadecimali) può essere effettuata utilizzando una istruzione ADD per elaborare i 32 bit meno significativi ed una istruzione ADC per elaborare i 32 bit più significativi.

```

      ADC      ADD
56A9C2D4 67A43B5F +
44B9A5A4 A6B4C55A =
9B636879 0E5900B9

```

FLAG di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	<code>ADC 0x00002000,%EDX</code>
Registro Generale, Memoria	<code>ADC %CL,0x12AB1024</code>
Registro Generale, Registro Generale	<code>ADC %AX,%DX</code>
Immediato, Memoria	<code>ADCB \$0x5B, (%EDI)</code>
Immediato, Registro Generale	<code>ADC \$0x54A3,%AX</code>

7.1.2 SUBTRACT WITH BORROW

FORMATO: SBB source, destination

AZIONE: Modifica l'operando destinatario sottraendovi sia l'operando sorgente sia il contenuto del flag CF (il risultato dell'operazione è consistente sia se gli operandi sono interpretati come numeri naturali sia se gli operandi sono interpretati come numeri interi); mette ad 1 il contenuto del flag CF se, interpretando gli operandi come numeri naturali, è stato richiesto un prestito; mette ad 1 il contenuto del flag OF se, interpretando gli operandi come numeri interi, si è verificato un traboccamento. Poiché l'istruzione SBB prevede la gestione del prestito che può essere stato richiesto durante l'esecuzione di una precedente istruzione SUB o SBB, essa può essere usata per predisporre pacchetti di istruzioni per la sottrazione di operandi multi_word.

Ad esempio, la somma di due operandi a 64 bit (16 cifre esadecimali) può essere effettuata utilizzando una SUB per elaborare i 32 bit meno significativi ed una SBB per elaborare i 32 bit più significativi.

```
      SBB      SUB
9B636879 0E5900B9 -
56A9C2D4 67A43B5F =
44B9A5A4 A6B4C55A
```

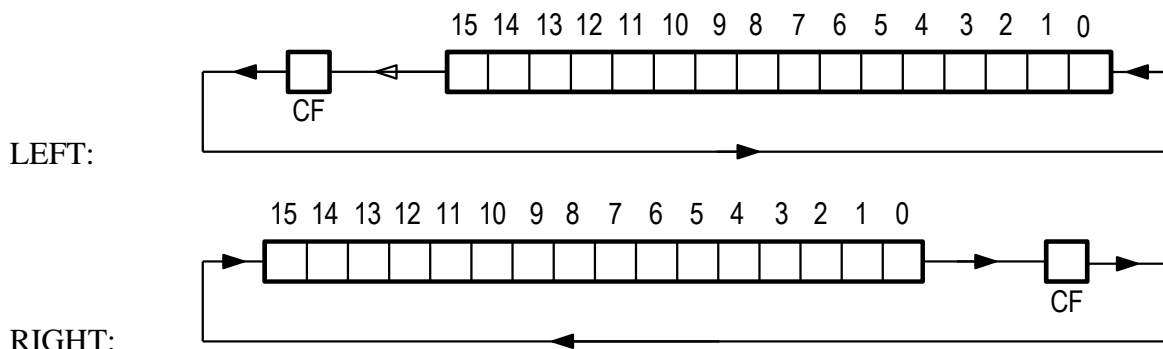
FLAG di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	SBB 0x00002000,%EDX
Registro Generale, Memoria	SBB %CL,0x12AB1024
Registro Generale, Registro Generale	SBB %AX,%DX
Immediato, Memoria	SBBW \$0x255B, (%EDI)
Immediato, Registro Generale	SBB \$0x54A3,%AX

7.1.3 ROTATE THROUGH CARRY LEFT/RIGHT

FORMATO: RCL source, destination
RCL destination
RCR source, destination
RCR destination

Fanno la stessa cosa delle corrispondenti istruzioni di rotazione, coinvolgendo anche CF. I formati di indirizzamento sono identici.



Operandi	Esempi
Immediato, Registro Generale	ROR \$1,%EAX
Immediato, Memoria	RORB \$7,0x00002000
Registro CL, Registro Generale	ROR %CL,%BX
Registro CL, Memoria	RORL %CL, (%EDI)
Memoria	RORL (%EDI)
Registro Generale	ROR %AX

7.1.4 LOOP / LOOPcon

FORMATO: LOOP %EIP ± displacement
LOOPcon %EIP ± displacement

AZIONE: Decrementa ECX. Se, dopo il decremento, $ECX \neq 0$, e se l'eventuale condizione *con* è vera, sostituisce EIP con il nuovo indirizzo.

FLAG di cui viene modificato il contenuto: Nessuno.

Di seguito sono riassunti i codici operativi delle istruzioni di LOOP condizionato e, per ciascuno di essi, è spiegato brevemente il significato della pertinente condizione *con*.

LOOPE LOOPZ	(Loop if Equal, Loop if Zero) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era uguale all'operando sorgente.
LOOPNE LOOPNZ	(Loop if Not Equal, Loop if Not Zero) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizione dopo un'istruzione CMP indica che l'operando destinatario non era uguale all'operando sorgente.

7.1.5 MOVE DATA FROM STRING TO STRING (with REPEAT)

FORMATO: MOVSSuf

REP MOVSSuf

AZIONE: Copia il numero di byte specificato dal suffisso *suf* dall'indirizzo di memoria puntato da ESI all'indirizzo di memoria puntato da EDI. Se DF=0, somma ad ESI e ad EDI il numero di byte specificato dal suffisso. Se DF=1, sottrae da ESI e da EDI il numero di byte specificato dal suffisso.

Se viene premesso il prefisso REP, allora le azioni indicate sopra vengono ripetute per il numero di volte specificato in ECX, che viene decrementato fino a zero.

FLAG di cui viene modificato il contenuto: Nessuno.

7.1.6 LOAD STRING

FORMATO: LODSSuf

AZIONE: Copia il numero di byte specificato dal suffisso *suf* dall'indirizzo di memoria puntato da ESI dentro AL, AX o EAX. Se DF=0, somma a ESI il numero di byte specificato dal suffisso. Se DF=1, sottrae da ESI il numero di byte specificato dal suffisso.

Se viene premesso il prefisso REP, allora le azioni indicate sopra vengono ripetute per il numero di volte specificato in ECX, che viene decrementato fino a zero.

FLAG di cui viene modificato il contenuto: Nessuno.

7.1.7 STORE STRING (with REPEAT)

FORMATO: STOSSuf

REP STOSSuf

AZIONE: Copia il contenuto di AL, AX o EAX (a seconda del suffisso *suf*) all'indirizzo di memoria puntato da EDI. Se DF=0, somma a EDI il numero di byte specificato dal suffisso. Se DF=1, sottrae da EDI il numero di byte specificato dal suffisso.

Se viene premesso il prefisso REP, allora le azioni indicate sopra vengono ripetute per il numero di volte specificato in ECX, che viene decrementato fino a zero.

FLAG di cui viene modificato il contenuto: Nessuno.

7.1.8 COMPARE STRING OPERANDS (with CONDITIONAL REPEAT)

FORMATO: CMPSsuf

REPE CMPSsuf

REPNE CMPSsuf

AZIONE: Confronta due operandi in memoria, lunghi il numero di byte specificato dal suffisso *suf*. L'operando sorgente si trova a partire dall'indirizzo di memoria puntato da ESI, l'operando destinatario si trova a partire dall'indirizzo di memoria puntato da EDI. Se DF=0, somma a ESI e a EDI il numero di byte specificato dal suffisso. Se DF=1, sottrae da ESI e da EDI il numero di byte specificato dal suffisso.

Se tale istruzione viene seguita da un'istruzione di salto condizionato Jcon, la condizione *con* si intende riferita all'operando *sorgente* (quello puntato da ESI), contrariamente a quanto avviene con l'istruzione CMP.

Se viene premesso il prefisso REPE (REPNE), allora le azioni indicate sopra vengono ripetute per il numero massimo di volte specificato in ECX, che viene decrementato fino a zero. Le ripetizioni proseguono finché gli operandi sono uguali (diversi).

FLAG di cui viene modificato il contenuto: Tutti.

7.1.9 SCAN STRING (with CONDITIONAL REPEAT)

FORMATO: SCASsuf

REPE SCASsuf

REPNE SCASsuf

AZIONE: Confronta il contenuto di AL, AX o EAX (a seconda del suffisso *suf*) con l'operando destinatario di pari lunghezza che si trova in memoria a partire dall'indirizzo puntato da EDI. Se DF=0, somma a EDI il numero di byte specificato dal suffisso. Se DF=1, sottrae da EDI il numero di byte specificato dal suffisso.

Se tale istruzione viene seguita da un'istruzione di salto condizionato Jcon, la condizione *con* si intende riferita all'operando *destinatario* (quello puntato da EDI).

Se viene premesso il prefisso REPE (REPNE), allora le azioni indicate sopra vengono ripetute per il numero massimo di volte specificato in ECX, che viene decrementato fino a zero. Le ripetizioni proseguono finché gli operandi sono uguali (diversi).

FLAG di cui viene modificato il contenuto: Tutti, secondo l'algoritmo della CMP, ad ogni ripetizione.

7.1.10 INPUT STRING (with REPEAT)

FORMATO: `INSsuf`
`REP INSsuf`

AZIONE: preleva uno, due, quattro byte (a seconda del suffisso *suf*) dalla porta, doppia porta, quadrupla porta di ingresso il cui offset è contenuto in `DX`. Il dato prelevato viene inserito in memoria a partire dall'indirizzo di memoria contenuto in `EDI`. Se `DF=0`, somma a `EDI` il numero di byte specificato dal suffisso. Se `DF=1`, sottrae da `EDI` il numero di byte specificato dal suffisso.

Se viene premesso il prefisso `REP`, allora le azioni indicate sopra vengono ripetute per il numero di volte specificato in `ECX`, che viene decrementato fino a zero.

FLAG di cui viene modificato il contenuto: Nessuno

7.1.11 OUTPUT STRING (with REPEAT)

FORMATO: `OUTSsuf`
`REP OUTSsuf`

AZIONE: fa uscita di uno, due, quattro byte (a seconda del suffisso *suf*) alla porta, doppia porta, quadrupla porta di uscita il cui offset è contenuto in `DX`. L'operando sorgente è prelevato a partire dall'indirizzo di memoria contenuto in `ESI`. Se `DF=0`, somma a `ESI` il numero di byte specificato dal suffisso. Se `DF=1`, sottrae da `ESI` il numero di byte specificato dal suffisso.

Se viene premesso il prefisso `REP`, allora le azioni indicate sopra vengono ripetute per il numero di volte specificato in `ECX`, che viene decrementato fino a zero.

FLAG di cui viene modificato il contenuto: Nessuno

7.1.12 SET /CLEAR DIRECTION FLAG

FORMATO: `STD`
`CLD`

AZIONE: setta (`STD`) o resetta (`CLD`) il flag `DF`

FLAG di cui viene modificato il contenuto: `DF`.

Quando `DF` vale 1, le istruzioni stringa decrementano i registri indice (`ESI` e/o `EDI`).

Quando `DF` vale 0, le istruzioni stringa incrementano i registri indice (`ESI` e/o `EDI`).

7.1.13 SET if CONDITION MET

FORMATO: SETcon %reg

SETcon mem

AZIONE: Setta a 1 o a 0 l'operando destinatario, a seconda se la condizione *con* specificata è vera o falsa. Le condizioni sono le stesse che possono essere scritte come salti condizionati. Il destinatario deve essere ad 8 bit.

FLAG di cui viene modificato il contenuto: Nessuno.

Di seguito sono riassunti i codici operativi di alcune delle istruzioni di salto condizionato e, per ciascuno di essi, è spiegato brevemente il significato della pertinente condizione *con*.

SETE	(Set if Equal) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era uguale all'operando sorgente.
SETNE	(Set if Not Equal) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizione dopo un'istruzione CMP indica che l'operando destinatario non era uguale all'operando sorgente.
SETA	(Set if Above) la condizione è soddisfatta se CF contiene 0 e ZF contiene 0; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come numeri naturali.
SETAE	(Set if Above or Equal) la condizione è soddisfatta se CF contiene 0; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri naturali.
SETB	(Set if Below) la condizione è soddisfatta se CF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come numeri naturali.
SETBE	(Set if Below or Equal) la condizione è soddisfatta se CF contiene 1 o ZF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore od uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri naturali.
SETG	(Set if Greater) la condizione è soddisfatta se ZF contiene 0 e se il contenuto di SF è uguale a quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come numeri interi.
SETGE	(Set if Greater or Equal) la condizione è soddisfatta se il contenuto di SF è uguale a quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri interi.
SETL	(Set if Less) la condizione è soddisfatta se il contenuto di SF è diverso da quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come numeri interi.
SETLE	(Set if Less or Equal) la condizione è soddisfatta se ZF contiene 1 oppure se il contenuto di SF è diverso da quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore od uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri interi.
SETZ	(Set if Zero) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione indica che il risultato dell'istruzione precedente è stato zero.
SETNZ	(Set if Not Zero) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizione indica che il risultato dell'istruzione precedente è stato diverso da zero.
SETC	(Set if Carry) la condizione è soddisfatta se CF contiene 1.
SETNC	(Set if No Carry) la condizione è soddisfatta se CF contiene 0.
SETO	(Set if Overflow) la condizione è soddisfatta se OF contiene 1.
SETNO	(Set if No Overflow) la condizione è soddisfatta se OF contiene 0.
SETS	(Set if Sign) la condizione è soddisfatta se SF contiene 1.
SETNS	(Set if No Sign) la condizione è soddisfatta se SF contiene 0.

7.1.14 CONDITIONAL MOVE

FORMATO: CMOVcon %reg, %reg
CMOVcon mem, %reg

AZIONE: esegue la corrispondente istruzione MOV se è vera la condizione specificata, altrimenti non fa niente. Sorgente e destinatario devono essere a 16 o 32 bit.

FLAG di cui viene modificato il contenuto: Nessuno.

Di seguito sono riassunti i codici operativi **di alcune** delle istruzioni di conditional move e, per ciascuno di essi, è spiegato brevemente il significato della pertinente condizione *con*.

CMOVE	(Move if Equal) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era uguale all'operando sorgente.
CMOVNE	(Move if Not Equal) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizione dopo un'istruzione CMP indica che l'operando destinatario non era uguale all'operando sorgente.
CMOVA	(Move if Above) la condizione è soddisfatta se CF contiene 0 e ZF contiene 0; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come numeri naturali.
CMOVAE	(Move if Above or Equal) la condizione è soddisfatta se CF contiene 0; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri naturali.
CMOVB	(Move if Below) la condizione è soddisfatta se CF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come numeri naturali.
CMOVBE	(Move if Below or Equal) la condizione è soddisfatta se CF contiene 1 o ZF contiene 1; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore od uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri naturali.
CMOVG	(Move if Greater) la condizione è soddisfatta se ZF contiene 0 e se il contenuto di SF è uguale a quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era maggiore dell'operando sorgente, essendo gli operandi interpretati come numeri interi.
CMOVGE	(Move if Greater or Equal) la condizione è soddisfatta se il contenuto di SF è uguale a quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario maggiore o uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri interi.
CMOVL	(Move if Less) la condizione è soddisfatta se il contenuto di SF è diverso da quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore dell'operando sorgente, essendo gli operandi interpretati come numeri interi.
CMOVLE	(Move if Less or Equal) la condizione è soddisfatta se ZF contiene 1 oppure se il contenuto di SF è diverso da quello di OF; il verificarsi di questa condizione dopo l'esecuzione di un'istruzione CMP indica che l'operando destinatario era minore od uguale rispetto all'operando sorgente, essendo gli operandi interpretati come numeri interi.
CMOVZ	(Move if Zero) la condizione è soddisfatta se ZF contiene 1; il verificarsi di questa condizione indica che il risultato dell'istruzione precedente è stato zero.
CMOVNZ	(Move if Not Zero) la condizione è soddisfatta se ZF contiene 0; il verificarsi di questa condizione indica che il risultato dell'istruzione precedente è stato diverso da zero.
CMOVC	(Move if Carry) la condizione è soddisfatta se CF contiene 1.
CMOVNC	(Move if No Carry) la condizione è soddisfatta se CF contiene 0.
CMOVO	(Move if Overflow) la condizione è soddisfatta se OF contiene 1.
CMOVNO	(Move if No Overflow) la condizione è soddisfatta se OF contiene 0.
CMOVS	(Move if Sign) la condizione è soddisfatta se SF contiene 1.
CMOVNS	(Move if No Sign) la condizione è soddisfatta se SF contiene 0.