Appunti Reti Logiche

Luca Seggiani
2024

1 Lezione del 24-09-24

1.1 Introduzione

Il corso di reti logiche tratta di:

1. **Linguaggio assembler:** come scrivere programmi semplici, come avviene la compilazione in linguaggio macchina;
2. **Reti logiche:** reti combinatorie, reti combinatorie per l'aritmetica, reti sequenziali asincrone e sincronizzate;
3. **Microprogammazione:** reti sequenziali sincronizzate, come realizzare una rete logica da specifiche. "Micro" qui sta per *hardware*;
4. **Il calcolatore:** processore, interfacce comuni e convertitori.

1.1.1 Introduzione alle reti logiche

Si parla di reti *logiche* in quanto si guarda all'hardware da una prospettiva funzionale, indipendente dalla sua tecnologia. Ad esempio, una porta NOR sarà implementata con determinati circuiti, ma tutto ciò che interessa a questo corso è come si comporta logicamente: $y = 1 \Leftrightarrow A = B = 0$.

1.2 Programmazione assembly

Il nome corretto del linguaggio sarebbe Assembly, ma noi lo chiameremo Assembler per ragioni storiche. L'assembler è il linguaggio con cui si scrivono le istruzioni eseguite dal processore. Il processore implementa effettivamente un ciclo fetch-execute dove preleva la prossima istruzione macchina (in assembler) dalla memoria e la esegue.

1.2.1 Linguaggio macchina

Il linguaggio macchina (LM) è dato dal contenuto effettivo della memoria che contiene le istruzioni, ergo una sequenza di zero e uno. Il linguaggio assembler adotta una sintassi simbolica per il linguaggio macchina: ad esempio, `MOV %AX, %BX`.

Il processo di trasformazione dall'assembler all'LM si chiama **assemblaggio**, mentre il processo di traasformazione da un linguaggio ad alto livello all'assembler si chiama **compilazione**.

1.2.2 Generalità sull'assembler

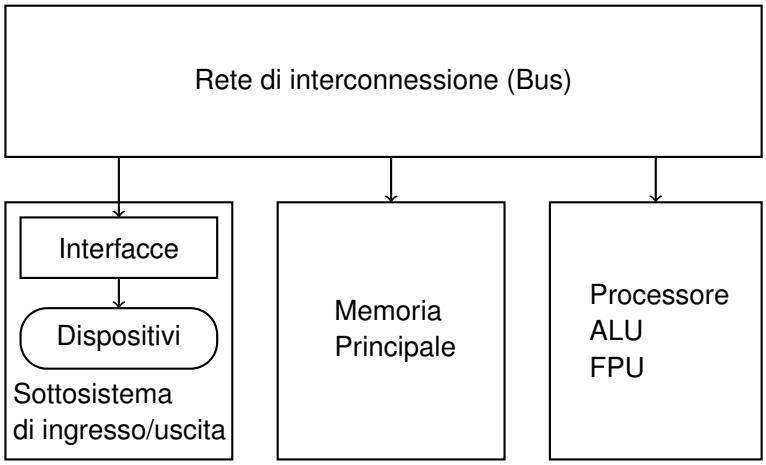
Si dice che assembler è un linguaggio a basso livello. Mancano i costrutti a cui siamo abituati da i linguaggi di alto livello:

1. Non esistono costrutti di flow control (for, if-else, ecc...), tutto si fa con istruzioni di salto.
2. Non esistono tipi variabile: gli operandi sono stringhe di bit che si riferiscono a locazioni in memoria.

Inoltre, l'assembler è strettamente legato all'hardware, ed è specifico per ogni processore. Noi vedremo l'assembler dei processori della famiglia Intel x86, che non è uguale all'assembler dei processori Arm Cortex, ecc... Questo rende il codice in assembler mai portatile. Fatta questa precisazione, possiamo dire che i principi generali restano comunque validi fra famiglie di processori diverse.

Esiste ancora oggi una nicchia di utilizzo del linguaggio assembler: quello dello sviluppo di sistemi embedded. Inoltre, il linguaggio ha un importante significato didattico e culturale.

1.3 Schema a blocchi del calcolatore



Un calcolatore è formato, in linea generale, da una rete di interconnessione (bus) che collega fra di loro:

- Interfacce che comunicano con dispositivi;
- La memoria principale che contiene dati e programmi;
- Il processore, che esegue il ciclo fetch-execute. Possiamo aggiungere che ogni processore, oggi, contiene almeno due blocchi:
  - L'**ALU**, Arithmetic Logic Unit, che si occupa di calcoli aritmetici su numeri interi (interpretando le stringhe di bit come numeri naturali o interi in complemento a 2) e operazioni logiche;
  - L'**FPU**, Floating Point Unit, che si occupa dei numeri a virgola mobile.

1.4 Riassunto di rappresentazione dell'informazione

Da qui in poi x è il numero rappresentato e X la sequenza di bit rappresentante.

1.4.1 Numeri naturali

Intervallo di rappresentabilità

n bit rappresentano 2^n naturali sull'intervallo $[0, 2^n - 1]$.

Trasformazione diretta

Per portare un'intero in rappresentazione binaria nel suo corrispondente in base 10, si

sa che presi n bit $b_{n-1}, b_{n-2}, ..., b_1, b_0$ della rappresentazione X, essi rappresentano il naturale x:

$$x = b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + ... + b_1 \cdot 2 + b_0 = \sum_{i=0}^{n-1} b_n \cdot 2^i$$

Il bit più a sinistra è il Most Significant Bit (MSB), cioé b_{n-1}, quello più a destra il Least Significant Bit (LSD) cioè b_0.

Trasformazione inversa

Per portare un'intero in base 10 nella sua rappresentazione binaria, si usa l'algoritmo DIV-MOD:

Algoritmo 1 DIV-MOD

Input: x in base 10
Output: X rappresentazione in base 2
Inizializza $q \leftarrow x$, $r \leftarrow 0$, and $i \leftarrow 0$
Crea un'array vuota R per i resti
while $q \neq 0$ **do**
  $r \leftarrow q \mod 2$
  Metti r in $R[i]$
  $q \leftarrow q/2$
  $i \leftarrow i + 1$
end while
Gli $R[n-1], R[n-2], ..., R[0]$ rimasti (letti al contrario) sono le cifre di X.

1.4.2 Numeri interi in complemento a due

Intervallo di rappresentabilità

n bit rappresentano 2^n interi sull'intervallo $[-2^{n-1}, 2^{n-1} - 1]$.

Trasformazione diretta

Per portare un intero x in base 10 nella sua rappresentazione in complemento a due X su n bit, si decide alternativamente rispetto al segno di x di rappresentare il naturale N in X:

$$N = \begin{cases} x & x \geq 0 \\ 2^n + x & x < 0 \end{cases}, \quad X = N_2$$

dove si nota che nella seconda espressione $2^n + x$ equivale a $2^n - |x|$, dalla negatività di x.

Alternativamente, sui soli numeri negativi:

- Si converte x in rappresentazione binaria.
- Si trova il complemento, ovvero la rappresentazione che inverte tutti i bit (che equivale alla rappresentazione in complemento a 2 dell'opposto -1).
- A questo punto si aggiunge 1, ignorando qualsiasi overflow.

La rappresentazione X trovata è il complemento a 2 di x. Simbolicamente:

$$X = \begin{cases} x_2 & x \geq 0 \\ (\bar{x} + 1)_2 & x < 0 \end{cases}$$

Trasformazione inversa

Per portare la rappresentazione in complemento a due X su n bit di un intero x all'intero stesso, ci si comporta come per le rappresentazioni di naturali, ma prendendo il bit più significativo dagli n bit $b_{n-1}, b_{n-2}, ..., b_1, b_0$ della rappresentazione X con valenza negativa:

$$x = -b_{n-1} \cdot 2^{n-1} + b_{n-2} \cdot 2^{n-2} + ... + b_1 \cdot 2 + b_0 = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_n \cdot 2^i$$

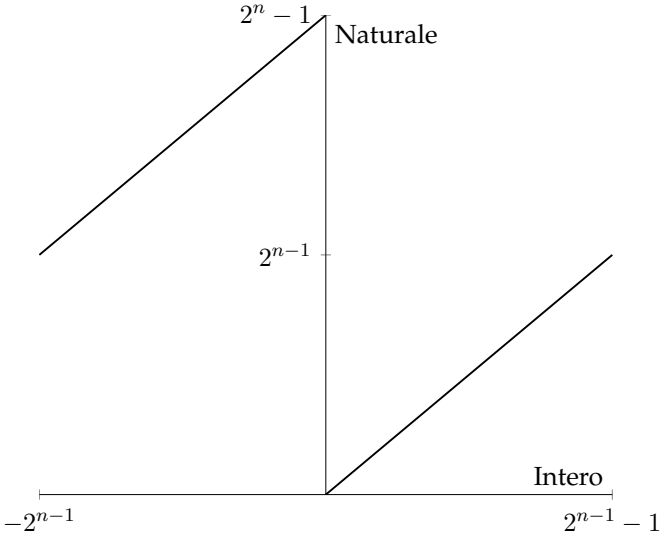
Alternativamente, si nota che il bit più significativo della rappresentazione sarà impostato a 0 per numeri positivi e 1 per numeri negativi. Ciò significa che avremo:

$$x = \begin{cases} X_{10} & X_{n-1} = 0 \\ -(\bar{X} + 1)_{10} & X_{n-1} = 1 \end{cases}$$

dove la barra rappresenta l'operazione complemento.

1.4.3 Rappresentazioni di interi e naturali, diagramma a farfalla

La rappresentazione in complemento 2 su n bit è effettivamente una funzione dal dominio $[-2^{n_1}, 2^{n-1} - 1]$ degli interi al codominio $[0, 2^n - 1]$ dei naturali. Tale funzione prende il nome di *diagramma a farfalla*:



da cui notiamo la relazione fra un'intero e il naturale che lo rappresenta in complemento a 2.

1.4.4 Valori notevoli del complemento a 2

Vale la pena notare alcuni valori notevoli del complemento a 2 su n bit.

- Innanzitutto, 0 rimane 0, ergo una fila di di n zeri.
- Uno zero seguito da $n-1$ uni è il numero più positivo positivo, ergo $2^{n-1}-1$.
- Aggiungendo uno, si arriva ad un uno seguito da $n-1$ zeri, che è il numero negativo possibile, ergo -2^{n-1}. Notare che questo combacia col prendere il numero più positivo $2^{n-1}-1$, e ricavare uno meno del suo opposto -2^{n-1}, che abbiamo appurato essere ciò che accade quando si complementa (e infatti i due numeri sono l'uno il complemento dell'altro).
- Infine, una sequenza di n uno rappresenta il più piccolo numero negativo, ergo -1.

Si nota che, al pari dei naturali, la rappresentazione dei numeri interi in complemento a 2 è effettivamente ciclica.

1.4.5 Notazione esadecimale

Scrivere lunghe stringhe binarie diventa velocemente complicato. Per questo si adotta una notazione esadecimale per stringhe di 4 bit ($[0, 15]$):

Decimale	Binario	Esadecimale
0	0000	`0x0`
1	0001	`0x1`
2	0010	`0x2`
3	0011	`0x3`
4	0100	`0x4`
5	0101	`0x5`
6	0110	`0x6`
7	0111	`0x7`
8	1000	`0x8`
9	1001	`0x9`
10	1010	`0xA`
11	1011	`0xB`
12	1100	`0xC`
13	1101	`0xD`
14	1110	`0xE`
15	1111	`0xF`

A questo punto, possiamo denotare qualsiasi stringa binaria come una lista di numeri esadecimali prefissi da `0x` (che serve ad indicare la rappresentazione esadecimale stessa), ad esempio `0xC1` (11000001).

1.4.6 Nota sulle potenze di 2

Conviene ricordare le prime potenze di 2:

Esponente	Valore
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
9	512
10	$1024 \approx 1000$
11	2048
12	4096
13	8192

e inoltre ricordare che, visto $2^10 = 1024 \approx 1000$, le unità di misura usuali diventano:

Unità	Potenza
2^{10}	1 KB
2^{20}	1 MB
2^{30}	1 GB

e cosi via.

1.5 Struttura del calcolatore

1.5.1 Spazio di memoria

La memoria del calcolatore, vista dal programmatore assembler, è uno spazio lineare di 2^{32} (su calcolatori a 32 bit) locazioni (celle) di memoria, dalla capacità di un byte ciascuna. Ogni cella è quindi identificata da un numero di 32 bit, detto **indirizzo**.

Lo spazio di memoria è in larga parte implementato attraverso Random Access Memory (RAM), ovvero memoria volatile. Solo una piccola parte dello spazio è implementata attraverso Read Only Memory (ROM), ovvero memoria permanente, che contiene le istruzioni da eseguire al reset.

Accesso allo spazio di memoria

Il processore può accedere (leggere/scrivere) a:

- Singole locazioni (byte) da 8 bit;
- Doppie locazioni (word) da 16 bit;
- Quadruple locazioni (double word) da 32 bit.

Per gli accessi 16/32 bit si usa l'indirizzo più piccolo delle 2/4 locazioni. Si ricorda che l'indirizzo più grande contiene i bit più significativi.

Gli indirizzi di memoria assembler sono solo simbolici, e vengono tradotti dall'assemblatore, e in parte runtime. Questo significa che non si può accedere a memoria appartenente al sistema operativo, o memoria fuori dai limiti fisici del sistema, ecc...

1.5.2 Spazio di Input/Output

Lo spazio di Input/Output è formato da 2^{16}, ovvero 64k, locazioni o **porte**. Ogni porta ha una capacità di un byte ed è indirizzata da un numero di 16 bit.

Il processore accede alle porte attraverso operazioni particolari di lettura o scrittura (in o out). Spesso le porte sono configurate per un solo tipo di operazione: sola lettura o sola scrittura.

Le locazioni di memoria sono solitamente identiche fra di loro, le porte di I/O no. Indirizzi diversi significano dispositivi diversi, e si rende quindi necessario conoscere fisicamente gli indirizzi.

1.5.3 Processore

Il processore è dotato di una memoria interna formata da locazioni di memoria da 32 bit (**registri**). Questi si dividono in registri **generali**, riservati alle elaborazioni, e **di stato**, riservati a compiti speciali.

Registri generali

I registri iniziano generalmente con la lettera **E**, che sta per *Extended*. Questo perché storicamente i registri erano da 16 bit, e successivamente sono stati estesi a 32 bit. Possiamo quindi riferirci a più sezioni dello stesso registro:

- **EAX:** tutti i 32 bit del registro esteso;
- **AL:** la parte *bassa* del registro **AX**, ergo quella meno significativa, da 8 bit;
- **AH:** la parte *alta* del registro **AX**, ergo quella più significativa, da 8 bit;
- **AX:** il registro AX legacy, che combina **AL** e **AH**, da 16 bit.

Alcuni registri vengono storicamente utilizzati per particolari funzioni:

- **EAX** è utilizzato da alcune istruzioni aritmetiche per contenere operandi e risultati. Viene detto **accumulatore**.
- **ESI**, **EDI**, **EBX**, **EBP** vengono detti registri puntatore, dove **B** sta per base e **I** per indice. In particolare:
  - **ESI**, **EDI** vengono utilizzati come registri indice per accessi in memoria.
  - **EBX** è utilizzato come indirizzo di base per l'accesso in memoria. Viene solitamente detto **base**.
  - **EBP** è utlizzato sempre come indirizzo di base per l'accesso in memoria.
- **ECX** è utilizzato come contatore nei cicli. Viene detto **contatore**.
- **EDX** è utilizzato come operando di operazioni aritmetiche. Viene detto **data**.
- **ESP** è utilizzato per indirizzare la **pila** o **stack**, ovvero una parte di memoria con disciplina LIFO che serve a gestire sottoprogrammi.

Registri di stato

Ricordiamo due registri di stato:

- L'**EIP** viene detto **instruction pointer**, o **program counter**. Viene usato per contenere l'indirizzo della locazione dalla quale sarà prelevata la prossima istruzione da eseguire. Il contenuto dell'EIP è fissato al reset iniziale, e impostato sulla prima istruzione da eseguire (in memoria ROM) all'indirizzo `0xFFFF0000`. Un po' di celle in memoria centrale da questo indirizzo in poi sono implementate in ROM.

  Possiamo quindi dire che il ciclo fetch-loop si svolge come segue:

  - Il processore preleva dalla memoria, all'indirizzo EIP, una nuova istruzione;
  - Incrementa EIP del numero di byte dell'istruzione prelevata;
  - Esegue l'istruzione e ripete.

  Da questo si ha che le istruzioni in memoria vengono eseguite sequenzialmente nell'ordine in cui sono incontrate, a meno che non si definiscano salti attraverso altre determinate istruzioni.

- L'**EF** viene detto **extended flag**. Consiste di 32 elementi detti **flag**, fra cui ricordiamo:

  - **OF:** flag di overflow (traboccamento) delle operazioni aritmetiche;
  - **SF:** flag di segno, impostato quando l'ultima operazione restituisce un complemento a 2 con $MSB = 1$ (ergo negativo);
  - **ZF:** flag zero, che viene impostato quando l'ultima operazione restituisce qualcosa di nullo;
  - **CF:** flag di carry (riporto), che viene impostato quando l'ultima operazione richiede un riporto o un prestito.

I flag **OF** e **SF** sono significativi per operazioni su interi. Il flag **CF** è significativo per operazioni su naturali. Il flag **ZF** è significativo per entrambi i tipi di operazione.

Al reset i flag visti finora sono impostati a 0.

2 Lezione del 25-09-24

2.1 Introduzione all'Assembler

2.1.1 Codifica macchina e codifica mnemonica

Possiamo adottare 2 metodi per codificare le istruzioni eseguite dal processore:

- **Codifica macchina:** la serie di zeri e di uni che codificano, nel linguaggio del processore, le operazioni che esegue. Il formato macchina è, nell'architettura che ci interessa, il seguente:

- **Codifica mnemonica:** un modo **simbolico** per riferirsi alle istruzioni. Un'istruzione può quindi essere semplicemente: `MOV \%EAX, 0x01F4E39`.

Il linguaggio assembler usa la codifica mnemonica delle istruzioni, e dispone di sovrastrutture sintattiche che lo rendono più comprensibile agli umani. Ad esempio, permette l'uso di nomi simbolici per locazioni di memoria: `MOV \%EAX, pippo`.

Segmento	Byte	Funzione
I Prefix (Instruction Prefix)	0/1 byte	Modifica l'istruzione.
O Prefix (Operand-size prefix)	0/1 byte	Modifica la dimensione degli operandi.
Opcode	1/2 byte	Specifica l'operazione.
Mode (ModR/M Byte)	0/1 byte	Specifica la modalità d'indirizzamento e i registri operandi.
SIB Byte	0/1 byte	Viene usato in congiunzione con il Mod/RM btye quando si usa l'indirizzamento complesso (scale-index-base).
Displacement	0/1/2/4 byte	Specifica un'offset in memoria, sempre nell'indirizzamento complesso.
Immediate	0/1/2/4 byte	Specifica le costanti ad indirizzamento immediato.

2.1.2 Istruzioni in codifica mnemonica

Un'istruzione ha 3 campi:

- **Codice operativo:** stabilisce quale operazione eseguire;
- **Suffisso di lunghezza:** stabilisce la lunghezza (che può variare) degli operandi;
- **Operandi:** gli operandi su cui si applica l'operazione. Possono essere contenuti in registri, in celle di memoria, nelle porte I/O o direttamente nell'istruzione (**costanti**).

Il suffisso di lunghezza può essere omesso quando è chiaro (essenzialmente quando si usa un registro).

Sintatticamente la struttura è `OPCODEsuffix source, dest`, che diventa qualcosa come `ADD \%BX, pluto`. Questa istruzione effettua l'operazione `ADD` (aggiungi), aggiungendo al registro `BX` ciò che è contenuto nel simbolo `pluto`.

Operandi di istruzioni

Le istruzioni ammettono 0, 1 o 2 operandi. Quando sono 2, il primo operando si chiama **sorgente** e il secondo **destinatario**, e solitamente hanno la stessa lunghezza. Quando è 1, l'operando può essere sia sorgente che destinatario a seconda dell'istruzione.

2.1.3 Primo esempio di programma

Si presenta un programma per contare il numero di uno trovati dalla locazione `0x00000100` a `0x0000010i3`e scriverlo nella locazione `0x00000104`.

```
1  MOVB $0x00, %CL          # sposta $0x00 in %CL
2  MOVL 0x00000100, %EAX    # sposta 32 bit da 0x00000100 a %EAX
3  CMPL $0x00000000, %EAX   # confronta 32 bit di 0 con il registro %EAX
4  JE   %EIP+$0x07          # salta se uguale a %EIP+$0x07,
5                           # ergo 0x0000020C + 0x07 = 0x00000213
6  SHRL %EAX                # trasla a destra %EAX
7  ADCB $0x00, %CL          # aggiungi a %CL 0 + carry
8  JMP  %EIP-$0x0C          # salta incondizionato a %EIP-$0x0C,
9                           # ergo 0x00000213 - 0x0C = 0x00000207
10 MOVB %CL, 0x00000104     # sposta byte da %CL a 0x00000104
```

Il programma svolge i seguenti passi:

Algoritmo 2 Conta 0

Inizializza il registro CL (Counter Low) a 0
Sposta i 32 bit da `0x00000000` a `0x00000103` in EAX
while true **do**
  if EAX è vuoto (tutti zeri) **then**
    Salta all'ultima istruzione
  end if
  Sposta EAX a destra
  Aggiungi il flag carry (che prende il valore rimosso da EAX) al registro CL
end while
Sposta il byte in CL nella locazione `0x00000104`

2.1.4 Istruzioni assembler

Le istruzioni assembler si dividono in:

- **Operative:** ovvero quelle che svolgono qualche operazione (ADD, SHR, MOV, CMP, ...);
- **Di controllo**: cioè che si occupano di altreare il flusso del programma (JMP, JE, ecc...).

Indirizzamento delle istruzioni operative

Le istruzioni operative si indirizzano attraverso l'**OPCODE** (codice operazione, ADD, MOV, ecc...), seguito da un suffisso (**B**, *byte* da 8 bit, **W**, *word* da 16 bit o **L**, *long* da 32 bit) che può essere omesso, e gli indirizzi sorgente e destinazione.

- Si possono **indirizzare i registri** sia come sorgenti che come destinatari, ovvero gli 8 registri generali da 32 bit, gli 8 registri generali da 16 bit, e gli 8 registri generali da 8 bit (disponibili solo sui registri A, B, C e D). Bisogna precedere i nomi dei registri con %.
- Si può avere **indirizzamento immediato**, ovvero di costanti preceduti da **$**, solo sull'operando sogente.
- Si può **indirizzare la memoria**, ma solo da sorgente o solo da destinatario, specificando un'indirizzo di memoria da 32 bit. Ergo non posso scrivere:

```
MOVB pippo, pluto
```

  ma devo scrivere:

```
MOV pippo, %EAX % qua il suffisso di lunghezza e' implicito
MOVL %EAX, pluto
```

  L'indirizzamento della memoria, nel caso più generale, è dato da:

$$\text{indirizzo} = \text{base} + \text{indice} \times \text{scala} \pm \text{displacement}$$

  dove base e indice sono due registri generali da 32 bit, scala una costante dal valore 1 (default), 2, 4, 8, e displacement una costante intera.

  La sintassi è `OPCODEsfx +/- disp(base,indice,scala)`.

  Si distingue poi l'indirizzamento di tipo:

- **Diretto**, dove si indica soltanto il displacement, che coincide con l'indirizzo. `OPCODEW 0x00002001` significa prendi la word a partire da `0x00002001`.
- **Indiretto**, o con registro puntatore, dove si sfrutta un registro: `OPCODEL (\%EBX)` significa indirizzare il valore indirizzato da EBX. Si può specificare una scala: `OPCODEL (,\%EBX,4)` significa il valore nel registro EBX moltiplicato per 4. Si noti come a essere moltiplicato è l'indice e non la base.
- **Displacement e registro di modifica**, ad esempio da `OPCODEW 0x002A3A2B (\%EDI )` si ottiene l'operando a 16 bit ottenuto sommando al displacement `0x002A3A2B` il contenuto di EDI, modulo 2^32.
- **Bimodificato senza displacement**, ad esempio `OPCODEW (\%EBX, \%EDI)`, che dipende sia da EBX che da EDI. Si può anche includere una scala: `OPCODEW (\% EBX, \%EDI, 8)`.
- **Bimodificato con displacement**, come prima ma con displacement: `OPCODEB 0x002F9000 (\%EBX, \%EDI)`, ovvero l'indirizzo dato da base in EBX + indice in EDI + l'offset modulo 2^32. Si può avere anche negativo: `OPCODEB -0x9000 (\%EBX , \%EDI)`, dove si sottrae l'offset invece di sommarlo.

Notare che senza il $ i numeri in formato esadecimale sono interpretati automaticamente come indirizzi.

- Si possono **indirizzare le porte I/O**, come prima in uno solo dei due operandi. Questo si fa con le istruzioni specifiche IN e OUT. In particolare si ha indirizzamento di tipo:
  - **Diretto**, solo per indirizzi < 256, in quanto nel formato macchina ci sono 8 bit. Ad esempio `IN 0x001A, \%AL` o `OUT \%AL, 0x003A`.
  - **Indiretto con registro puntatore**, usando come registro puntatore soltanto DX. Ad esempio `IN (\%DX), \%AX` o `OUT \%AL, (\%DX)`.

2.2 Panoramica sulle istruzioni

Abbiamo diviso le istruzioni in **operative** e **di controllo**. Possiamo fare ulteriori suddivisioni:

- **Operative:**
  - Di trasferimento;
  - Aritmetiche;
  - Di traslazione/rotazione:
  - Logiche.
- **Di controllo:**
  - Di salto;
  - Di gestione di sottoprogrammi.

Conviene definire formato, funzionamento, comportamento sui flag e modalità di indirizzamento ammesse per gli operandi di ogni operazione, in quanto l'assembler non è **ortogonale**, ergo ci sono particolari restrizioni su *quali* operandi e modalità di indirizzamento possono essere combinate.

3 Lezione del 26-09-24

3.1 Istruzioni di trasferimento

Le istruzioni di trasferimento spostano memoria:

- Dalla memoria a un registro;
- Da un registro a un registro;
- Dallo spazio I/O a un regsitro.

Non esistono altre possibilità, ergo non si può (per quanto interessa a noi) spostare da memoria a memoria. In verità esistono alcune istruzioni nei processori di nuova generazione che ottimizzano operazioni di questo tipo, che verrano viste in seguito. Sfruttando i registri, il trasferimento da memoria a memoria si fa attraverso un registro, in due istruzioni.

Nessuna istruzione di trasferimento modifica i flag.

3.1.1 MOVE

- **Formato:** `MOV source, destination`
- **Azione:** sostituisce l'operando destinatario con una copia dell'operando sorgente.
- **Flag:** nessuno.

Operandi	Esempi
Memoria, Registro Generale	`MOV 0x00002000, \%EDX`
Registro Generale, Memoria	`MOV \%CL, 0x12AB1024`
Registro Generale, Registro Generale	`MOV \%AX, \%DX`
Immediato, Memoria	`MOVB \$0x5B, (\%EDI)`
Immediato, Registro generale	`MOV \$0x54A3, \%AX`

3.1.2 LOAD EFFECTIVE ADDRESS

- **Formato:** `LEA source, destination`
- **Azione:** sostituisce l'operando destinatario con l'espressione indirizzo contenuta nell'operando sorgente.
- **Flag:** nessuno.

Operandi	Esempi
Memoria, Registro Generale a 32 bit	`LEA 0x00002000, \%EDX`
	`LEA 0x00213AB1 (\%EAX,\%EBX,4), \%ECX`

A differenza di MOV, LEA calcola l'indirizzo della locazione di memoria cercata come base + index $\times$ scala $\pm$ displacement, e carica quell'indirizzo nella destinazione, non il valore contenuto in esso. Nel primo esempio, questo equivale alla MOV con indirizzamento immediato. In altri casi permette di ricavare esplicitamente il valore ottenuto dall'indirizzamento complesso.

3.1.3 EXCHANGE

- **Formato:** `XCHG source, destination`
- **Azione:** sostituisce l'operando destinatario con l'operando sorgente e viceversa. Questa operazione è l'unica che modifica il sorgente.
- **Flag:** nessuno.

Operandi	Esempi
Memoria, Registro Generale	`XCHG 0x00002000, \%DX`
Registro Generale, Memoria	`XCHG \%AL, 0x000A2003`
Registro Generale, Registro Generale	`XCHG \%EAX, \%EDX`

Grazie a quest'istruzione in assembler si possono scambiare due operandi con una sola istruzione (**non trasparenza** dei registri) **atomica**. Questo è particolarmente utile nel caso di esecuzione concorrente.

3.1.4 INPUT

- **Formato:**
  - `IN indirizzo, \%AL` (8 bit)
  - `IN indirizzo, \%AX` (16 bit)
  - `IN (\%DX), \%AX` (8 bit)
  - `IN (\%DX), \%Al` (16 bit)
- **Azione:** sostituisce il contenuto del registro destinatario (AL 8 bit, AX 16 bit) con il contenuto di un adeguato numero di porte consecutive. L'indirizzo è specificato direttamente (per porte con indirizzo < 256), o indirettamente usando il registro DX.
- **Flag:** nessuno.

3.1.5 OUTPUT

- **Formato:**
  - `OUT \%AL, indirizzo` (8 bit)
  - `IN \%AX, indirizzo` (16 bit)
  - `IN \%AX,` (%DX) (8 bit)
  - `IN \%Al, (\%DX)` (16 bit)
- **Azione:** copia il contenuto del registro sorgente (AL 8 bit, AX 16 bit) su un adeguato numero di porte consecutive. L'indirizzo è specificato direttamente (per porte con indirizzo < 256), o indirettamente usando il registro DX.
- **Flag:** nessuno.

3.1.6 Non ortogonalità INPUT/OUTPUT

Le uniche due operazioni che gestiscono l'input e l'output possono trasferire solo dai o nei registri AL e AX, e indirizzare indirettamente la memoria puntando col registro DX. Questo rende le operazioni non ortogonali: non si possono usare altri registri, ed eventuali operazioni vanno fatte nel processore,

3.2 Pila

La pila, o **stack**, è una regione di memoria gestita con politica Last In First Out (LIFO), essenziale al funzionamento del calcolatore. Permette di annidare sottoprogrammi, funzionalità per cui l'assembler è organizzato.

Generalmente, la pila viene usata come segue per eseguire i sottoprogrammi:

- Prima di saltare al sottoprogramma, si fa **PUSH** sulla pila dell'indirizzo di ritorno (e.g. l'indirizzo della prossima istruzione);
- Si esegue il sottoprogramma;
- Al termine del sottoprogramma, si fa **POP** dalla pila del prossimo indirizzo.

Più sottoprogrammi possono chiamarsi a vicenda (annidarsi), ponendosi su livelli via via superiori della pila. Al termine della sua esecuzione, ogni sottoprogramma tornerà all'indirizzo di ripresa del sottoprogramma precedente, finché tutti i sottoprogrammi non termineranno l'esecuzione.

Il registro **ESP** punta al top della pila, ergo non va usato per altri scopi. Va però inizializzato prima che parta il programma. Si deve inoltre notare che la pila in assembler si estende *verso il basso*: aggiungere alla pila significa decrementare ESP, e rimuovere dalla pila significa incrementare ESP. I frame successivi della pila si vanno a disporre via via sotto (o "a sinistra") del frame corrente.

Per lavorare sulla pila si usano le istruzioni:

3.2.1 PUSH

- **Formato:** `PUSH source`
- **Azione:** decrementa ESP e copia il sorgente nell'indirizzo puntato da ESP. Il sorgente deve essere a 16 bit o a 32 bit. Nello specifico, compie le seguenti azioni:
  - Decrementa l'indirizzo contenuto nel registro ESP di 2 o 4;
  - Memorizza una copia dell'operando sorgente nella word o long il cui indirizzo è contenuto in ESP.
- **Flag:** nessuno.

Operandi	Esempi
Memoria	`PUSHW 0x3214200A`
Immediato	`PUSHL \$0x4871A000`
Registro Generale	`PUSH \%BX`

3.2.2 POP

- **Formato:** `POP destination`
- **Azione:** copia una word o un long dall'indirzzo puntato dall'ESP nel destinatario e incrementa ESP. Nello specifico compie le seguenti azioni:
  - Sostituisce all'operando destinatario una copia del contenuto nella word o long il cui indirizzo è contenuto in ESP;
  - Incrementa di due o quattro l'indirizzo contenuto in ESP, rimuovendo la word o il long copiato.
- **Flag:** nessuno.

Operandi	Esempi
Memoria	`POPW 0x02AB2000`
Registro Generale	`POP \%BX`

Dati temporanei nella pila

Solitamente la pila viene usata per memorizzare dati temporanei, visto che i registri sono pochi e spesso hanno scopi diversi in momenti diversi. Ad esempio:

```
# sto usando %EAX, mi serve un dato da una porta
PUSH %EAX
IN 0x001A, %AL
...
POP %EAX # ritorno da dove ero
```

3.2.3 PUSHAD

- **Formato:** `PUSHAD`
- **Azione:**: salva nella pila corrente una copia degli 8 registri generali a 32 bit, nell'ordine: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.
- **Flag:** nessuno.

3.2.4 POPAD

- **Formato:** `POPAD`
- **Azione:**: copia dalla pila corrente gli 8 registri generali a 32 bit, nell'ordine: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.
- **Flag:** nessuno.

3.3 Istruzioni aritmetiche

Molte operazioni aritmetiche di base non distinguono numeri naturali e numeri interi, distinzione che viene fatta solo per moltiplicazioni e divisioni.

Le operazioni possono modificare i flag, e in questo caso i flag da controllare dipenderanno dal tipo di numeri su cui si è fatta l'operazione (informazione nota soltanto al programmatore).

Abbiamo quindi che un'operazione aritmetica si svolge seguendo i passi:

- Si esegue l'operazione;
- Si controllano i flag interessati (OF, SF e ZF sugli interi, CF e ZF sui naturali) per verificarne l'esito.

Vediamo quindi le operazioni aritmetiche:

3.3.1 ADD

- **Formato:** `ADD source, destination`
- **Azione:** modifica l'operando destinatario sommandovi l'operando sorgente. Il risultato è consistente sia che si interpretino i numeri come naturali, che come interi.
- **Flag:** attiva CF se, interpretando i numeri come naturali, si è verificato un riporto; attiva OF se, interpretando gli operandi come interi, si è verificato un trabocco-mento. Inoltre attiva opportunamente ZF e SF se il numero è rispettivamente zero o negativo (in complemento a 2).

Operandi	Esempi
Memoria, Registro Generale	`ADD 0x00002000, \%EDX`
Registro Generale, Memoria	`ADD \%CL, 0x12AB1024`
Registro Generale, Registro Generale	`ADD \%AX, \%DX`
Immediato, Memoria	`ADDB \$0x5B, (\%EDI)`
Immediato, Registro Generale	`ADD \$0x54A3, \%AX`

Funzionamento della ADD

Il passo elementare di una somma consiste nel sommare due addendi (propriamente due cifre degli addendi) e un riporto entrante per produrre:

- Una cifra;
- Un riporto uscente (cioè il riporto entrante per il prossimo passo).

L'ultimo riporto, se non entra in memoria, attiva il carry flag (CF).

L'operazione di somma ha lo stesso effetto sia su naturali che su interi in complemento a 2: la differenza sta nel controllo dell'attivazione dei flag. Il carry flag non ha infatti alcun significato nella somma fra interi: dobbiamo controllare l'OF.

In generale, si ha overflow (OF) quando il risultato esce dall'intervallo di rappresentabilità. Si può capire se si è verificato un overflow controllando i segni degli operandi:

- **Segni discordi:** non c'é overflow;
- **Segni concordi:** il risultato è concorde se è concorde con gli operandi.

La ADD imposta quindi OF secondo queste regole. Il ZF viene poi impostato se il risultato è fatto da tutti zeri, e il SF viene impostato se il MSB è uno.

3.3.2 INCREMENT

- **Formato:** `INC destination`
- **Azione:** equivale all'istruzione `ADD \$1, destination`.
- **Flag:** modifica tutti i flag di ADD tranne CF (il riporto).

Operandi	Esempi
Memoria	`INCB (\%ESI)`
Registro Generale	`INC \%CX`

Quest'istruzione è più compatta di ADD, e storicamente era anche più veloce. Questo deriva dal fatto che la circuiteria che implementava l'incremento era più efficiente di quella che implementa le somme.

3.3.3 SUBTRACT

- **Formato:** `SUB source, destination`
- **Azione:** modifica l'operando destinatario sottraendovi l'operando sorgente. Il risultato è consistente sia che si interpretino i numeri come naturali, che come interi.
- **Flag:** attiva CF se, interpretando i numeri come naturali, si è verificato un riporto; attiva OF se, interpretando gli operandi come interi, si è verificato un trabocamento.

Operandi	Esempi
Memoria, Registro Generale	`SUB 0x00002000, \%EDX`
Registro Generale, Memoria	`SUB \%CL, 0x12AB1024`
Registro Generale, Registro Generale	`SUB \%AX, \%DX`
Immediato, Memoria	`SUBB \$0x5B, (\%EDI)`
Immediato, Registro Generale	`SUB \$0x54A3, \%AX`

Funzionamento della SUBTRACT

Il passo elementare della sottrazione è effettivamente il contrario di quello della somma: si sottraggono il sottraendo e un prestito entrante al minuendo, producendo:

- Una cifra;
- Un prestito uscente.

Il carry flag (CF) memorizza il prestito. Se alla fine dell'operazione il CF è impostato, significa che il risultato è un numero intero.

Questo funziona anche sugli interi: in questo caso, come prima, non si controlla il CF, ma l'OF, che conterrà la seguente informazione:

- La differenza di numeri concordi è sempre rappresentabile;
- La differenza di numeri discordi è rappresentabile solo se il risultato ha il segno del minuendo.

Il ZF e il SF vengono attivati secondo le regole già note.

3.3.4 DECREMENT

- **Formato:** `DEC destination`
- **Azione:** equivale all'istruzione `SUB \$1, destination`.
- **Flag:** modifica tutti i flag di SUBTRACT tranne CF (il prestito).

Operandi	Esempi
Memoria	`DECB (\%EDI)`
Registro Generale	`DEC \%CX`

3.3.5 ADD WITH CARRY

- **Formato:** `ADC source, destination`
- **Azione:** modifica l'operando destinatario sommandovi sia l'operando sorgente sia il contenuto del flag CF.
- **Flag:** modifica tutti i flag come ADD.

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

Quest'istruzione è utile per effettuare somme di numeri più grandi di 32 bit. In questo caso si:

- Effettua la somma dei 32 bit meno significativi con ADD;
- Sommano i successivi 32 bit con ADC portandosi quindi dietro il carry.

3.3.6 SUBTRACT WITH BORROW

- **Formato:** `SBB source, destination`
- **Azione:** modifica l'operando destinatario sottraendovi sia l'operando sorgente sia il contenuto del flag CF.
- **Flag:** modifica tutti i flag come SUBTRACT.

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

Come ormai dovrebbe essere chiaro, è la duale dell'ADC, e si usa per effettuare sottrazioni di numeri più grandi di 32 bit.

3.3.7 NEGATE

- **Formato:** `NEG destination`
- **Azione:** interpreta l'operando destinatario come un numero intero e lo sostituisce con il suo opposto in complemento a 2.
- **Flag:** quando l'operazione non è possibile (l'intervallo di rappresentabilità degli interi in complemento a 2 non è simmetrico) imposta il flag OF. Imposta inoltre il flag CF quando l'operando è diverso da zero, e tutti gli altri flag in base a nullità e segno del risultato.

Operandi	Esempi
Memoria	`NEGB (\%EDI)`
Registro Generale	`NEG \%CX`

Funzionamento della NEGATE

L'opposto di un numero X in complemento a due è:

$$-X = \bar{X} + 1$$

Si ricordi che questo ha senso *solamente* se il numero è rappresentato in complemento a due.

3.3.8 COMPARE

- **Formato:** `CMP source, destination`
- **Azione:** verifica se l'operando destinatario è maggiore, uguale o minore dell'operando sorgente, sia interpretando gli operandi come naturali che come interi, e aggiorna i flag di conseguenza. Più propriamente, la compare si comporta come la SUB, ma senza sovrascrivere nessuno degli operandi.
- **Flag:** come la SUB.

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

3.3.9 Funzionamento della COMPARE

Solitamente la CMP si usa nei salti condizionati come:

```
1 CMP %AX, %BX
2 JCOND # salto condizionato
```

Ciò che fa la CMP è effettivamente creare un'oggetto temporaneo:

$$\text{tmp} = \text{dest} - \text{source}$$

che viene poi rimosso.

I flag restano però aggiornati, e questo valore può essere interpretato correttamente dalla JE per effettuare un salto condizionale.

3.4 Moltiplicazioni

Le moltiplicazioni, a differenza delle somme e delle differenze, sono diverse fra naturali ed interi. Bisogna inoltre notare che le dimensioni il risultato della somma di un numero a n cifre sta su n o $n+1$ cifre, mentre il prodotto di due numeri a n cifre sta su $2n$ cifre. In altre parole, il numero di bit necessari a memorizzare il risultato non è più confrontabile con quello degli operatori.

3.4.1 MULTIPLY

- **Formato:** `MUL source`
- **Azione:** considera l'operando sorgente come un moltiplicando, l'operando destinatario (implicito) come un moltiplicatore, e effettua la moltiplicazione assumendo i numeri naturali. Nello specifico:
  - Sorgente a 8 bit, si ha AX $=$ AL $\times$ source;
  - Sorgente a 16 bit, si ha DX_AX $=$ AX $\times$ source;
  - Sorgente a 32 bit, si ha EDX_EAX $=$ EAX $\times$ source.
- **Flag:** imposta CF e OF se il risultato non sta nel numero di bit di source. SF e ZF sono indefiniti.

Operandi	Esempi
Memoria	`MULB (\%ESI)`
Registro Generale	`MUL \%ECX`

3.4.2 INTEGER MULTIPLY

- **Formato:** `MUL source`
- **Azione:** considera l'operando sorgente come un moltiplicando, l'operando destinatario (implicito) come un moltiplicatore, e effettua la moltiplicazione assumendo i numeri interi. Nello specifico:
  - Sorgente a 8 bit, si ha AX $=$ AL $\times$ source;
  - Sorgente a 16 bit, si ha DX_AX $=$ AX $\times$ source;
  - Sorgente a 32 bit, si ha EDX_EAX $=$ EAX $\times$ source.
- **Flag:** li imposta tutti, ma non è attendibile.

Operandi	Esempi
Memoria	`IMULB (\%ESI)`
Registro Generale	`IMUL \%ECX`

Funzionamento delle MULTIPLY e INTEGER MULTIPLY

Queste operazioni hanno sia un operando che il destinatario impliciti, in base al tipo dell'operando fornito. Questo deriva dal fatto che il risultato di una moltiplicazione raramente sta nello stesso numero di bit dei fattori. Di preciso, abbiamo visto i 3 tipi di moltiplicazione concessi:

- Sorgente a 8 bit, si ha AX $=$ AL $\times$ source;
- Sorgente a 16 bit, si ha DX_AX $=$ AX $\times$ source;
- Sorgente a 32 bit, si ha EDX_EAX $=$ EAX $\times$ source.

La differenza fra le prime due operazioni e l'ultima, in particolare con sorgente a 16 bit, che usa una due registri da 16 bit separati, ha principalmente motivi storici (il registro EAX è stato introdotto dopo).

Si può rimettere il valore dai due registri a 16 bit in un registro a 32 bit attraverso la pila:

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

4 Lezione del 27-09-24

4.1 Divisioni

La divisone è l'operazione più complessa fra le 4 operazioni aritmetiche fondamentali. I risultati, di base, sono due: **quoziente** e **resto**. Inoltre, l'operazione non è ben definita quando il divisore vale 0.

Facciamo innanzitutto delle considerazioni di dimensione dei risultati:

$$X/Y \to (Q, R), \quad 0 \leq R \leq Y - 1, \quad 0 \leq Q \leq X$$

In assembler, si assume il quoziente e il resto stiano sulla metà dei bit che rappresentano il dividendo. Bisogna fare attenzione in quanto questo non è sempre il caso.

4.1.1 DIVIDE

- **Formato:** `DIV source`
- **Azione:** considera l'operando sorgente come un divisore, l'operando destinatario (implicito) come un dividendo, e effettua la divisione assumendo i numeri naturali. Nello specifico:
  - Sorgente a 8 bit, si ha AL $=$ AX $\div$ source, e AH $=$ AX $\mod$ source;
  - Sorgente a 16 bit, si ha AX $=$ DX_AX $\div$ source, e DX $=$ DX_AX $\mod$ source;
  - Sorgente a 32 bit, si ha EAX $=$ EDX_EAX $\div$ source, e EDX $=$ EDX_EAX $\mod$ source;

  Nel caso il quoziente non sia esprimibile su un numero di bit pari a quello del divisore, allora si genera un'eccezione interna, che mette in esecuzione un sottoprogramma. Da lì in poi i risultati generati non sono più attendibili
- **Flag:** imposta tutti i bit, ma non è attendibile.

Operandi	Esempi
Memoria	`DIVB (\%ESI) \# AX destinazione implicita`
Registro Generale	`DIV \%ECX \# EDX\_EAX destinazione implicita`

Attenzione: la destinazione implicita non è quella che va a contenere il risultato, ma quella che contiene il dividendo. Negli esempi, le destinazioni quoziente resto sono rispettivamente AL e AH, EAX e EDX.

4.1.2 INTEGER DIVIDE

- **Formato:** `MUL source`
- **Azione:** considera l'operando sorgente come un divisore, l'operando destinatario (implicito) come un dividendo, e effettua la divisione assumendo i numeri interi. Nello specifico:
  - Sorgente a 8 bit, si ha AL $=$ AX/source, e AH $=$ AX $\mod$ source;
  - Sorgente a 16 bit, si ha AX $=$ DX_AX/source, e DX $=$ DX_AX $\mod$ source;
  - Sorgente a 32 bit, si ha EAX $=$ EDX_EAX/source, e EDX $=$ EDX_EAX $\mod$ source;
- **Flag:** li imposta tutti, ma non è attendibile.

Operandi	Esempi
Memoria	`IDIVB (\%ESI) \# AX destinazione implicita`
Registro Generale	`IDIV \%ECX \# EDX\_EAX destinazione implicita`

Bisogna stare attenti ai segni della divisione intera. Nella divisione intera il resto ha sempre il segno del dividendo, ed è minore in modulo del divisore. Ciò significa che il quoziente si approssima sempre all'intero più vicino allo zero (*per troncamento*). Ad esempio, -7 idiv $3 = -2, -1$ e 7 idiv $-3 = -2, +1$.

Funzionamento delle DIVIDE e INTEGER DIVIDE

Esistono quindi, come per le moltiplicazioni, tre tipi di divisione, con operando e destinatario impliciti:

- Sorgente a 8 bit, si ha AL $=$ AX/source, e AH $=$ AX $\mod$ source;
- Sorgente a 16 bit, si ha AX $=$ DX_AX/source, e DX $=$ DX_AX $\mod$ source;
- Sorgente a 32 bit, si ha EAX $=$ EDX_EAX/source, e EDX $=$ EDX_EAX $\mod$ source;

In tabella questo significa:

Dim. sorgente (divisore)	Dim. dividendo	Dividendo	Quoziente	Resto
8 bit	16 bit	AX	AL	AH
16 bit	32 bit	DX_AX	AX	DX
32 bit	64 bit	EDX_EAX	EAX	EDX

Se il quoziente non sta nel numero di bit previsto, viene sollevata un'eccezione, e il programma va in HALT. Bisogna quindi decidere quali versioni usare tenendo conto delle dimensioni dei possibili quoziente. Questo è importante in quanto non è cosi raro avere divisioni dove il quoziente non sta nella metà dei bit del dividendo, ad esempio:

```
1 MOV $3, %CL
2 MOV $15000, %AX
3 DIV %CL # come metto 5000 su una locazione da 8 bit?
```

per risolvere il problema, dobbiamo costringere il processore ad usare un altro tipo di divisione, quindi:

```
MOV $3, %CX
MOV $15000, %AX
MOV $0, %DX # devo ripulire DX, verra' usato il dividendo DX_AX
DIV %CX # il risultato va in AX, tutto bene
```

4.2 Note conclusive su moltiplicazioni e divisioni

Dobbiamo quindi ricordarci, riguardo a moltiplicazioni e divisioni, di:

- Scegliere con cura la versione che usiamo (sopratutto nel caso di divisioni dove il quoziente potrebbe non stare nella metà del numero di bit del dividendo);
- Azzerare di azzerare i registri DX o EDX prima della divisione, se è a più di 8 bit;
- Ricordare che il contenuto di DX o EDX viene modificato per operazioni su più di 8 bit.

4.3 Estensione di campo

Attraverso l'estensione di campo si rappresenta lo stesso numero su più cifre. Questo è banale sui naturali (si aggiunge uno zero), ma più complicato per gli interi. In questo caso si estende con il bit più significativo (quello di segno).

4.3.1 CONVERT BYTE TO WORD

- **Formato:** `CBX`
- **Azione:** interpreta il contenuto di AL come un numero intero a 8 bit, la rappresenta su 16 bit e quindi lo memorizza in AX.
- **Flag:** nessuno.

4.3.2 CONVERT WORD TO DOUBLEWORD

- **Formato:** `CWDE`
- **Azione:** interpreta il contenuto di AX come un numero intero a 16 bit, la rappresenta su 32 bit e quindi lo memorizza in EAX.
- **Flag:** nessuno.

Poniamo ad esempio di voler sommare due interi, uno in AX e l'altro in EBX:

```
MOV $-5, %AX
MOV $100000, $EBX
CWDE
ADD %EAX, %EBX
```

4.4 Istruzioni di traslazione e rotazione

Queste istruzioni variano l'ordine dei bit in un operando destinatario. Hanno due formati: `OCPODE source, destination` o `OPCODE destination`.

Quando si specifica un sorgente, esso rappresenta il numero di iterazioni per cui si ripete l'operazione. Il sorgente può essere ad indirizzamento immediato o essere il registro CL. Inoltre, deve essere ≤ 31 (sarebbe inutile fare ≥ 32 trasformazioni di 32 bit). Quando è omesso, il sorgente vale di default 1.

4.4.1 SHIFT LOGICAL LEFT

- **Formato:** `SHL source, destination`
- **Azione:** interpreta l'operando sorgente come un naturale n, e per n iterazioni:
  - Sostituisce il bit in CF con il MSB;
  - Sostituisce ogni bit (tranne il LSB) con il bit immediatamente a destra (il meno significativo);
  - Sostituisce il LSB con 0.
- **Flag:** nessuno.

Operandi	Esempi
Immediato, Registro Generale	`SHL \$1, \%EAX`
Immediato, Memoria	`SHLB \$7, 0x00002000`
Registro CL, Registro Generale	`SHL \%CL, \%EAX`
Registro CL, Memoria	`SHLL \%CL, (\%EDI)`
Memoria	`SHLL (\%EDI)`
Registro Generale	`SHL \%AX`

La SHL è utile per effettuare moltiplicazioni per 2 (shift a sinistra in binario significa $\times 2$), tranne nei casi in cui il prodotto non sta sul numero di bit del destinatario.

Per questo si controlla il CF, facendo però attenzione che per n iterazioni (date dal sorgente) vengono effettuati n sovrascrizioni del CF. Ergo, se la moltiplicazione fallisce, non sappiamo *quando* fallisce.

4.4.2 SHIFT ARITHMETIC LEFT

- **Formato:** `SAL source, destination`
- **Azione:** è identica alla SHL. Quindi equivale a moltiplicare per 2^{source}.
- **Flag:** nessuno.

Esiste come duale della SAR, ma in questo caso non deve fare nulla di diverso dalla SHL.

4.4.3 SHIFT LOGICAL RIGHT

- **Formato:** `SHR source, destination`
- **Azione:** interpreta l'operando sorgente come un naturale n, e per n iterazioni:
  - Sostituisce il bit in CF con il LSB;
  - Sostituisce ogni bit (tranne il MSB) con il bit immediatamente a sinistra (il più significativo);
  - Sostituisce il MSB con 0.
- **Flag:** nessuno.

Operandi	Esempi
Immediato, Registro Generale	`SHR \$1, \%EAX`
Immediato, Memoria	`SHRB \$7, 0x00002000`
Registro CL, Registro Generale	`SHR \%CL, \%EAX`
Registro CL, Memoria	`SHRL \%CL, (\%EDI)`
Memoria	`SHRL (\%EDI)`
Registro Generale	`SHR \%AX`

La SHR, come la SHL, è utile per effettuare divisioni per 2 (shift a destra in binario significa $\div 2$), concessa approssimazione del bit perso, tranne nei casi in cui il numero è un intero (lo 0 al MSB corrompe il segno). Per questo motivo si definisce la:

4.4.4 SHIFT ARITHMETIC RIGHT

- **Formato:** `SAR source, destination`
- **Azione:** è identica alla SHR, ma non sostituisce il MSB con 0, lasciandolo tale. Questo equivale a dividere per 2^{source}.
- **Flag:** nessuno.

La SAR ci permette di dividere velocemente interi per 2, come avremmo fatto sui naturali con la SHR.

4.4.5 Divisioni intere

Le IDIV e SAR approssimano diversamente: la IDIV approssima per troncamento, mentre la SAR approssima sempre a sinistra. Quindi, IDIV e SAR danno lo stesso quoziente solo quando il dividendo è positivo, o il resto nullo.

4.5 Istruzioni di rotazione

Le istruzioni di rotazione ruotano i bit, cioè effettuano uno shift con rientro dei bit in uscita dal lato opposto, con la possibilità di includere o meno CF nella rotazione.

4.5.1 ROTATE LEFT

- **Formato:** `ROL source, destination`
- **Azione:** interpreta l'operando sorgente come un naturale n, e per n iterazioni ruota verso sinistra senza usare il carry.
- **Flag:** nessuno.

Operandi	Esempi
Immediato, Registro Generale	`ROL \$1, \%EAX`
Immediato, Memoria	`ROLB \$7, 0x00002000`
Registro CL, Registro Generale	`ROL \%CL, \%EAX`
Registro CL, Memoria	`ROLL \%CL, (\%EDI)`
Memoria	`ROLL (\%EDI)`
Registro Generale	`ROL \%AX`

4.5.2 ROTATE RIGHT

- **Formato:** `ROR source, destination`
- **Azione:** interpreta l'operando sorgente come un naturale n, e per n iterazioni ruota verso destra senza usare il carry.
- **Flag:** nessuno.

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

4.5.3 ROTATE CARRY LEFT

- **Formato:** `RCL source, destination`
- **Azione:** interpreta l'operando sorgente come un naturale n, e per n iterazioni ruota verso sinistra usando il carry.
- **Flag:** imposta il carry assumendolo a sinistra del MSB.

Operandi	Esempi
Immediato, Registro Generale	`RCL \$1, \%EAX`
Immediato, Memoria	`RCLB \$7, 0x00002000`
Registro CL, Registro Generale	`RCL \%CL, \%EAX`
Registro CL, Memoria	`RCLL \%CL, (\%EDI)`
Memoria	`RCLL (\%EDI)`
Registro Generale	`RCL \%AX`

4.5.4 ROTATE CARRY RIGHT

- **Formato:** `RCR source, destination`
- **Azione:** interpreta l'operando sorgente come un naturale n, e per n iterazioni ruota verso destra usando il carry.
- **Flag:** imposta il carry assumendolo a destra del LSB.

Operandi	Esempi
Immediato, Registro Generale	`RCR \$1, \%EAX`
Immediato, Memoria	`RCRB \$7, 0x00002000`
Registro CL, Registro Generale	`RCR \%CL, \%EAX`
Registro CL, Memoria	`RCRL \%CL, (\%EDI)`
Memoria	`RCRL (\%EDI)`
Registro Generale	`RCR \%AX`

4.6 Istruzioni logiche

Queste istruzioni applicano gli operatori dell'algebra di Boole, e solitamente modificano flag.

4.6.1 NOT

- **Formato:** `NOT destination`
- **Azione:** modifica il destinatario applicandogli il NOT bit a bit.
- **Flag:** nessuno.

Operandi	Esempi
Memoria	`NOTL (\%ESI)`
Registro Generale	`NOT \%CX`

4.6.2 AND

- **Formato:** `AND source, destination`
- **Azione:** modifica il destinatario applicando l'AND bit a bit degli operandi.
- **Flag:** modifica tutti i flag (annulla CF e OF).

Operandi	Esempi
Memoria, Registro Generale	`AND 0x00002000, \%EDX`
Registro Generale, Memoria	`AND \%CL, 0x12AB1024`
Registro Generale, Registro Generale	`AND \%AX, \%DX`
Immediato, Memoria	`AND 5x5B, (\%EDI)`
Immediato, Registro Generale	`AND \$0x45AB54A3, \%EAX`

4.6.3 OR

- **Formato:** `OR source, destination`
- **Azione:** modifica il destinatario applicando l'OR bit a bit degli operandi.
- **Flag:** modifica tutti i flag (annulla CF e OF).

Operandi	Esempi
Memoria, Registro Generale	`OR 0x00002000, \%EDX`
Registro Generale, Memoria	`OR \%CL, 0x12AB1024`
Registro Generale, Registro Generale	`OR \%AX, \%DX`
Immediato, Memoria	`OR 5x5B, (\%EDI)`
Immediato, Registro Generale	`OR \$0x45AB54A3, \%EAX`

4.6.4 XOR

- **Formato:** `XOR source, destination`
- **Azione:** modifica il destinatario applicando l'OR bit a bit degli operandi.
- **Flag:** modifica tutti i flag (annulla CF e OF).

Operandi	Esempi
Memoria, Registro Generale	`XOR 0x00002000, \%EDX`
Registro Generale, Memoria	`XOR \%CL, 0x12AB1024`
Registro Generale, Registro Generale	`XOR \%AX, \%DX`
Immediato, Memoria	`XOR 5x5B, (\%EDI)`
Immediato, Registro Generale	`XOR \$0x45AB54A3, \%EAX`

4.6.5 Uso delle istruzioni logiche

Le istruzioni logiche vengono usate per operare su singoli bit degli operandi, usando uno specifico operatore sorgente immediato detto maschera (**bitmask**). Nello specifico:

- **AND:**
  - si usa per testare singoli bit di un operando. Ad esempio, si può implementare un salto condizionale se il quinto bit di AL vale zero:

```
AND $0x20, %AL  # 0x20 = 00100000
JZ # vale zero
```

  - si usa per resettare singoli bit di un operando. Ad esempio, si può resettare il sesto bit di BH:

```
AND $0xBF, $BH  # 0xBF = 10111111
```

  - si usa per l'estensione di operandi *naturali*. Ad esempio, si possono sommare due numeri naturali, di cui uno in AL e l'altro in EBX:

```
MOV $5, $AL
MOV $100000, %EBX
AND $0x000000FF, $EAX
ADD %EAX, %EBX
```

- **OR:** si usa per settare singoli bit di un operando. Ad esempio, si può settare il quarto bit di CL:

```
OR $0x10, %CL # =x10 = 00010000
```

- **XOR:**
  - si usa per invertire singoli bit. Ad esempio, si può invertire il quinto bit del registro AH:

```
XOR $0x20, %AH  # 0x20 = 00100000
```

  - si usa per resettare registri. Ad esempio, si può resettare EAX come:

```
XOR %EAX, %EAX  # equivale a dire MOV $0, %EAX, ma occupa
                # 1 byte invece di 5
```

4.7 Istruzioni di controllo

Le istruzioni di controllo permettono di alterare il flusso del programma, che altrimenti scorrerebbe normalmente in sequenza (le istruzioni vengono eseguite come vengono lette in memoria).

Conosciamo il ciclo fetch-execute: il processore carica un'istruzione, incrementa EIP, e la esegue. Alcune istruzioni alterano il valore di EIP, implementando quindi alterazioni del flusso di esecuzione:

- **Istruzioni di salto:** JMP, Jcon;
- **Istruzioni di gestione sottoprgrammi**: CALL, RET.

4.7.1 JUMP

- **Formato:** `JMP \%EIP +/- displacement`, `JMP *extended\_register`, `JMP *memory`
- **Azione:** calcola un'indrizzo di salto e lo immette nel registro EIP.
- **Flag:** nessuno.

Solitamente le istruzioni di salto si riferiscono ad un nome simbolico, ed è quindi compito dell'assemblatore ricondurre la sintassi ad una delle forme sopra riportate.

4.7.2 JUMP if CONDITION MET

- **Formato:** `Jcon \%EIP +/- displacement`
- **Azione:** esamina il contenuto dei flag. Se da questo esame risulta che la condizione *con* è soddisfatta, si comporta come `JMP \%EIP +/- displacement`, altrimenti non fa nulla.
- **Flag:** nessuno.

I prossimi paragrafi riguardano tutti i di condizione supportati.

Condizione	Funzionamento
JZ	Jump If Zero, la condizione è soddisfatta se ZF è impostato, ergo se il risultato dell'istruzione precedente è stato 0.
JNZ	Jump If Not Zero, la condizione è soddisfatta se ZF non è impostato, ergo se il risultato dell'istruzione precedente non è stato 0.
JC	Jump if Carry, la condizione è soddisfatta se CF è impostato.
JNC	Jump if No Carry, la condizione è soddisfatta se CF non è impostato.
JO	Jump if Overflow, la condizione è soddisfatta se OF è impostato.
JNO	Jump if No Overflow, la condizione è soddisfatta se OF non è impostato.
JS	Jump if Sign, la condizione è soddisfatta se SF è impostato.
JNS	Jump if No Sign, la condizione è soddisfatta se SF non è impostato.

4.7.3 Condizioni sui flag

Esistono le seguenti condizioni sui singoli flag:

Esempi

```
ADD %AX, %BX
JC ...
# continua
```

Se la somma dei contenuti di AX e BX presi come naturali non è rappresentabile su 16 bit, salta.

```
ADD %AX, %BX
JO ...
# continua
```

Se la somma dei contenuti di AX e BX presi come interi non è rappresentabile su 16 bit, salta.

```
SUB %AL, %BL
JS ...
# continua
```

Se la somma differenza dei contenuti di BL ed AL (in quest'ordine) presi come interi è negativa, salta.

4.7.4 Condizioni sui naturali

Esistono le seguenti condizioni sui confronti fra naturali:

Tutte queste condizioni seguono sempre una CMP, che aggiorna i flag in modo da permettere il confronto. I risultati dei confronti possono sempre evincersi dai flag.

Esempi

Condizione	Funzionamento
JE	Jump if Equal, la condizione è soddisfatta se ZF contiene 1, cioè dopo CMP su due numeri uguali.
JNE	Jump if Not Equal, la condizione è soddisfatta se ZF contiene 0, cioè dopo CMP su due numeri non uguali.
JA	Jump if Above, la condizione è soddisfatta se CF contiene 0 e ZF contiene 1, cioè dopo CMP su un destinatario maggiore del sorgente.
JAE	Jump if Above or Equal, la condizione è soddisfatta se CF contiene 0, cioè dopo CMP su un destinatario maggiore o uguale del sorgente.
JB	Jump if Below, la condizione è soddisfatta se CF contiene 1, cioè dopo CMP su un destinatario minore del sorgente.
JBE	Jump if Below or Equal, la condizione è soddisfatta se CF contiene 1 o ZF contiene 1, cioè dopo CMP su un destinatario minore o uguale del sorgente.

```
CMP %AX, %BX
JAE ...
# continua
```

Se BX è maggiore o uguale di AX, presi come naturali, salta.

```
CMP %EDX, %ECX
JB ...
# continua
```

Se ECX è minore stretto di EDX, presi come naturali, salta.

4.7.5 Condizioni sugli interi

Esistono le seguenti condizioni sui confronti fra interi:

Condizione	Funzionamento
JE	Jump if Equal, la condizione è soddisfatta se ZF contiene 1, cioè dopo CMP su due numeri uguali.
JNE	Jump if Not Equal, la condizione è soddisfatta se ZF contiene 0, cioè dopo CMP su due numeri non uguali.
JG	Jump if Greater, la condizione è soddisfatta se ZF contiene 0 o se SF è uguale a OF, cioè dopo CMP su un destinatario maggiore del sorgente.
JGE	Jump if Greater or Equal, la condizione è soddisfatta se SF è uguale a OF, cioè dopo CMP su un destinatario maggiore o uguale del sorgente.
JB	Jump if Less, la condizione è soddisfatta se SF è diverso da OF, cioè dopo CMP su un destinatario minore del sorgente.
JBE	Jump if Less or Equal, la condizione è soddisfatta se ZF contiene 1 o se Sf è diverso da OF, cioè dopo CMP su un destinatario minore o uguale del sorgente.

Come prima, queste operazioni seguono sempre una CMP ed evincono il risultato del confronto dai flag.

Esempi

- ```
  CMP %AX, %BX
  JGE ...
  # continua
  ```

  Se BX è maggiore o uguale di AX, presi come interi, salta.

- ```
  CMP %EDX, %ECX
  JL ...
  # continua
  ```

  Se ECX è minore stretto di EDX, presi come interi, salta.

5 Lezione del 01-10-24

5.1 Istruzioni per sottoprogrammi

Nei sottoprogrammi vengono coninvolte due istruzioni `CALL`, e `RET`. Entrambe si riferiscono alla pila.

5.1.1 CALL

- **Formato:** `CALL %EIP +/- $displacement`, `CALL *extended_register`, `CALL *memory`
- **Azione:** effettua la chiamata di un sottoprogramma, ovvero:
  - Salva il valore corrente di EIP nella pila;
  - Modifica EIP come farebbe JMP.
- **Flag:** nessuno.

Operandi	Esempi
Displacement	`CALL 0x00400010`
Registro	`CALL *%EAX`
Memoria	`CALL *0x00400010`

5.1.2 RET

- **Formato:** `RET`
- **Azione:** ritorna da un sottoprogramma, ovvero:
  - Rimuove un long dalla pila;
  - Lo inserisce in EIP.
- **Flag:** nessuno.

Esistono poi altre istruzioni di controllo, ovvero:

5.1.3 NOP

- **Formato:** `NOP`
- **Azione:** è l'istruzione nulla.
- **Flag:** nessuno.

5.1.4 HLT

- **Formato:** `HLT`
- **Azione:** arresta l'esecuzione fino al prossimo interrupt.
- **Flag:** nessuno.

5.1.5 HLF

- **Formato:** `HLF`
- **Azione:** arresta l'esecuzione e causa l'autocombustione del processore.
- **Flag:** nessuno.

5.2 Istruzioni privilegiate

Il codice in assembler può girare secondo due modalità sul sistema:

- **Sistema:** con accesso totale a tutte le istruzioni;
- **Utente:** senza l'accesso ad alcune istruzioni dette privilegiate.

Tra le istruzioni privilegiate ci sono `HLT`, `IN` e `OUT`. La `HLT` non è un grande problema, ma lo sono `IN` e `OUT`. Per ottenere input e output dal sistema, adoperiamo quindi determinati sottoprogrammi di servizio atti a fornire esattamente queste informazioni.

L'uso di sottoprogrammi di servizio per l'input/output è dovuto al fatto che le interfacce sono sistemi complessi, facili da portare in stato inconsistente, mentre i sottoprogrammi si assicurano di farne un corretto uso.

5.3 Struttura di un programma assembler

Vediamo adesso come strutturare un programma assembler scritto nell'ambiente GAS (Gnu Assembler). Un programma assembler è diviso in due sezioni

- **Sezione dati:** qui si dichiarano le variabili, ergo nomi simbolici per indirizzi di memoria che contengono i dati del programma;
- **Sezione codice:** istruzioni.

  In un programma abbiamo bisogno di:

- **Istruzioni**, viste finora;
- **Direttive**, necessarie all'assemblaggio e alla dichiarazione di variabili.

Ad esempio, potremo avere:

```
.GLOBAL _main

.DATA
...

.TEXT
_main:  NOP
...
        RET
```

Le linee che iniziano col punto sono direttive, le altre istruzioni. Una riga qualsiasi del codice è fatta come:

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

dove abbiamo una label, l'istruzione e un commento.

Tutto qui può mancare, tranne il ritorno carrello. Tutte le righe, inclusa l'ultima, vanno terminate. Inoltre, l'ultima riga dovrebbe essere una RET, che restituisce l'esecuzione al chiamante (qui l'ambiente).

Conviene iniziare il programma con una NOP, per assicurarsi che in fase di inizializzazione esso non faccia effettivamente nulla.

Vediamo ad esempio il programma visto prima per il conteggio degli uni, reso in questa struttura:

```
.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 $0x00, %CL
            JMP comp
fine:       MOVB %CL, conteggio
            RET
```

5.3.1 Direttive

Tutte le direttive iniziano con il carattere punto. Esse sono:

- **Dichiarazione di variabili:** Variabili dichiarate di seguito sono sempre consecutive in memoria. Si ha, di base:
  - `.BYTE`: riserva 1 byte;
  - `.WORD`: riserva 2 byte;
  - `.LONG`: riserva 4 byte.

  Esempi

```
var0: .WORD                 # scalare, 2 byte, valore 0x0000
                            # (considerato brutto, non inizializzare
                            #  si fa con .FILL)
var1: .BYTE 0x30            # scalare, 1 byte, valore 0x30
```

```
var2: .BYTE 0x30,0x31          # vettore, 2 componenti da 1 byte,
                               # valore 0x30 e 0x31
var3: .WORD 0x1020, 0x32AB     # vettore, 2 componenti da 2 byte,
                               # valore 0x1020e 0x32AB
var4: .LONG var3+2             # scalare, 4 byte, valore 0xAB
```

Esistono altri modi di inizializzare variabili particolari:

- `.FILL numero, dim, espressione`: dichiara `numero` variabili di lunghezza `dim` e le inizializza ad espression (0 di default). Dim può essere 1, 2 o 4.
- **ASCII**: si può usare la codifica ASCII fra single tick ', coi caratteri speciali dopo sequenze di escape, per indicare singoli byte. Ad esempio:

```
var5: .BYTE 'S', 'o', 'n', 'n', 'o'        # vettore, 4 componenti
                                           # da 1 byte
var6: .BYTE 0x53, 0x6F, 0x6E, 0x6E, 0x6F   # vettore, 4 componenti
                                           # da 1 byte
var7: .ASCII "Stea"                        # vettore, 4 componenti
                                           # da 1 byte
var8: .ASCIZ "Stea"                        # vettore, 5 componenti
                                           # da 1 byte (include il
                                           # terminatore)

```

- **Altre direttive:**
  - `.INCLUDE "path"`: include un sorgente nel presente file, prima dell'assemblamento;
  - `.SET nome, espressione`: serve a creare **costanti simboliche**. Tali costanti hanno nome `nome` e valore `espressione`. Ad esempio:

```
.SET dimensione, 4
.SET n_iter, (100 * dimensione)
...
MOV $n_iter, %CX  # e' accesso immediato
```

5.4 Costanti numeriche

Possiamo indicare costanti numeriche attraverso le seguenti convenzioni:

- **Naturali:** non hanno segno, e vengono convertite nella loro rappresentazione in base 2;
- **Intere:** hanno un segno + o - davanti, e vengono convertite nella loro rappresentazione in complemento a 2.

Inoltre possiamo scrivere costanti in base 2, 8, 10 e 16 attraverso i prefissi `0b`, `0`, nessun prefisso e `0x`.

Le variabili, quando non sono della dimensione giusta, vengono solitamente troncate (con avviso dall'assemblatore) o estese (senza avvisi dall'assemblatore).

5.5 Controllo di flusso

I costrutti di flusso a cui siamo abituati vengono implementati attraverso istruzioni di salto. Conviene comunque ragionare in costrutti ad alto livello, e limitarsi a tradurli in assembler. Da qui in puoi useremo una sintassi pseudo-C per indicare questi costrutti ad alto livello.

5.5.1 If-then-else

Prendiamo la sintassi:

```
if(%AX < variabile) {
  //ramo if
  ...
} else {
  //ramo else
  ...
}
//prosegui
...
```

potremo tradurla in due modi:

- Invertendo i rami then e else:

```
          CMP variabile, %AX
          JB ramothen
ramoelse: ... # ramo else
          JMP segue
ramothen: ... # ramo then
segue:    # prosegui
          ...
```

- Invertendo la condizione:

```
          CMP variabile, %AX
          JAE ramoelse
ramothen: ... # ramo then
          JMP segue
ramoelse: ... # ramo else
segue:    ... # prosegui
```

5.5.2 Ciclo for

Prendiamo:

```
for(int i = 0; i < variabile; i++) {
  //iter
  ...
}
//prosegui
...
```

si rende attraverso il registro CX, come:

```
        MOV $0, %CX
ciclo:  CMP var, %CX
        JE segue
        ... # iter
        INC %CX
        JMP ciclo
segue:  ... # prosegui
```

5.5.3 Ciclo do-while

Prendiamo infine:

```
do {
  //iter
  ...
} while(AX < var)
//prosegui
...
```

si rende come:

```
ciclo:  ... # iter
        CMP var, %AX
        JB ciclo
        ... # prosegui
```

5.5.4 Un piatto di spaghetti

In assembler ci è concesso fare ciò che non è permesso da linguaggi strutturati come il C o il Pascal. In questi linguaggi, un costrutto ha un solo punto di ingresso e un solo punto di uscita.

In assembler, invece, possiamo saltare fuori e dentro cicli e costrutti quando e dove vogliamo, ed è il programmatore che deve pensare a cosa il programma sta effettivamente facendo. Ad esempio, nessuno ci vieta di dire:

```
ciclo:  ... # inizio ciclo
        ...
label1: ... # meta' ciclo
        CMP var, %AX
        JB ciclo
        ...
        JMP label1 # salto dentro un ciclo a meta' esecuzione?
```

In assembler abbiamo a disposizione un'istruzione dedicata per i loop, che è:

5.5.5 LOOP

- **Formato:** `LOOP destination`
- **Azione:** decrementa ECX e salta alla destinazione se ECX $\neq 0$. ECX va inizializzato al numero di iterazioni desiderate, e non va toccato durante il ciclo.
- **Flag:** nessuno.

Si nota che la LOOP decrementa sempre ECX, quindi si applica difficilmente a cicli FOR dove vogliamo che la variabile di controllo incrementi, e ci serve che il suo valore nel corpo del ciclo. Si noti la differenza nei due esempi:

```
for(int i = var; i > 0; i--) {
  //iter (usa i)
}
```

diventa:

```
        MOV var, %ECX
ciclo:  ... # iter
        LOOP ciclo
```

```
for(int i = 0; i < var; i++) {
  //iter (usa i)
}
```

diventa:

```
        MOV $0, %EBX # usa EBX
ciclo:  ... # iter
        INC EBX
        CMP var, %EBX
        JE ciclo
```

5.5.6 LOOP condizionali

Esistono versioni condizionali della LOOP, che sono `LOOPE` e `LOOPNE`, simili alle Jump condizionali. In questo caso, oltre al registro ECX, si verifica la condizione e nel caso si salta. Ad esempio:

```
        MOV $10, %ECX
ciclo:  CMP src, dest
        LOOPcond ciclo
```

Queste istruzioni non sono indispensabili, in quanto possono essere rimpiazzate facilmente dalla `CMP` unita ad un Jump condizionale.

5.6 Passaggio di argomenti a sottoprogrammi

Le `CALL` e `RET` prima definite non fornisicono modi per passare parametri ai sottoprogrammi, o restituire valori ai chiamanti.

Dobbiamo quindi stabilire delle convenzioni, scegliendo se:

- Usare locazioni di memoria condivise;
- Usare registri;
- Usare la pila (che non verrà visto nel corso).

In assembler non esiste il concetto di visibilità o variabili locali, tutta la memoria è indirizzabile a qualsiasi livello. Comunque, quando si scrive un sottoprogramma, bisogna specificare i parametri di ingresso e di uscita con un'opportuno commento, come:

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

sottoprog:  ...
            MOV ..., %CX # preparo il ritorno
            RET
```

adesso potremo usare il sottoprogramma come:

```
MOV ..., %AX # preparo i parametri
MOV ..., %EBX
CALL sottoprog # chiamo
MOV %CX, var # var contiene il ritorno
```