# 1 Lezione del 27-09-24

## 1.1 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.

### 1.1.1 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.

### 1.1.2 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:

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

## 1.2 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.

## 1.3 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).

### 1.3.1 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.

### 1.3.2 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:

```
1 MOV $-5, %AX
2 MOV $100000, $EBX
3 CWDE
4 ADD %EAX, %EBX
```

## 1.4 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 $\leq 31$ (sarebbe inutile fare $\geq 32$ trasformazioni di 32 bit). Quando è omesso, il sorgente vale di default 1.

### 1.4.1 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.

### 1.4.2 SHIFT ARITHMETIC LEFT

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

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

### 1.4.3 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:

### 1.4.4 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 $2^{\text{source}}$.
- **Flag:** nessuno.

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

### 1.4.5 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.

## 1.5 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.

### 1.5.1 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` |

### 1.5.2 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` |

### 1.5.3 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` |

### 1.5.4 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` |

## 1.6 Istruzioni logiche

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

### 1.6.1 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` |

### 1.6.2 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` |

### 1.6.3 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` |

### 1.6.4 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` |

### 1.6.5 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
```

## 1.7 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.

### 1.7.1 JUMP

- **Formato:** `JMP \%EIP $\pm$ 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.

### 1.7.2 JUMP if CONDITION MET

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

I prossimi paragrafi riguardano tutti i di condizione supportati.

### 1.7.3 Condizioni sui flag

Esistono le seguenti condizioni sui singoli flag:

| 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. |

#### 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.

### 1.7.4 Condizioni sui naturali

Esistono le seguenti condizioni sui confronti fra naturali:

| 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. |

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

```
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.

### 1.7.5 Condizioni sui 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.