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 trasformazione da un linguaggio ad alto livello all'assembler si chiama **compilazione**.

1.2.2 Generalità sull'assembler

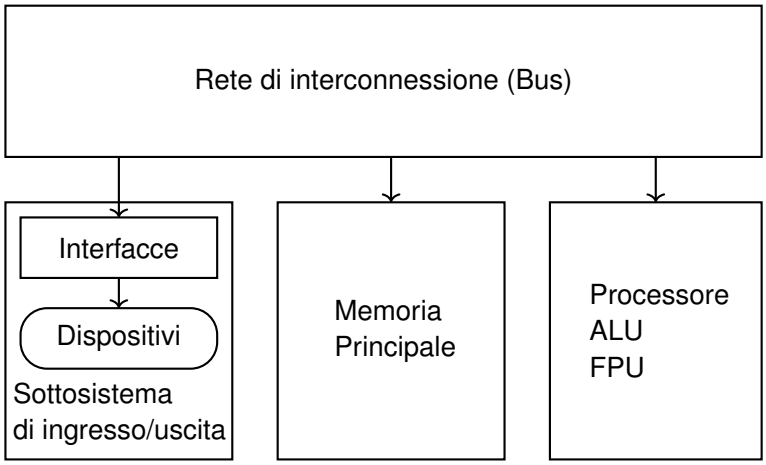
Si dice che assembler è un linguaggio a basso livello. Mancano i costrutti a cui siamo abituati da i linguaggi di alto livello:

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 $q \leftarrow x$, $r \leftarrow 0$, e $i \leftarrow 0$
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 (quindi 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:

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

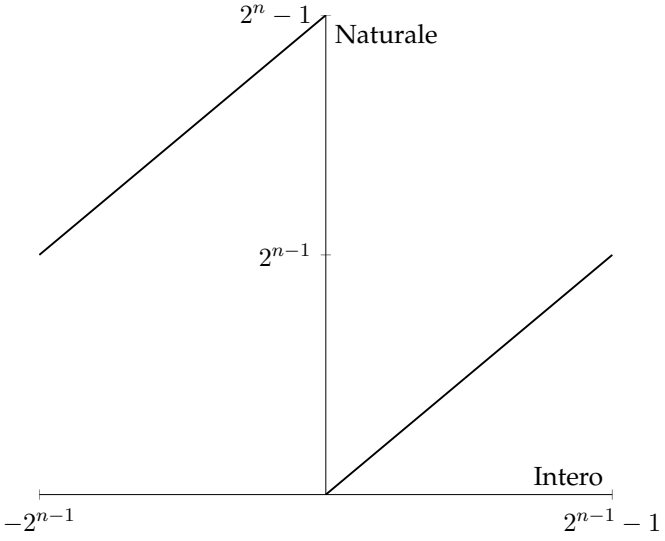
Alternativamente, si nota che il bit più significativo della rappresentazione sarà impostato a 0 per numeri positivi e 1 per numeri negativi. Ciò significa che avremo:

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

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 possibile, 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 (lo spazio di memoria è *little-endian*).

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, si imposta se l'ultima operazioni, presi gli operandi come interi, ha prodotto un risultato non rappresentabile su n bit;
  - **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, ergo presi gli operandi come naturali il risultato non è rappresentabile su n bit.

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 dalla somma di EBX e EDI. Si può anche includere una scala: `OPCODEW (%EBX, %EDI, 8)`, che va a moltiplicare solo `%EDI`.
  - **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. Inoltre, i suffissi di dimensione si riferiscono al numero di locazioni all'indirizzo *puntato* dai registri, non alla dimensione dei registri o altre cose ridicole.

- 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 traboccamento. 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 le cifre degli addendi, x_i e y_i e un riporto entrante r_i per produrre:

- La possima cifra s_i del risultato;
- Un riporto uscente r_{i+1} (cioè il riporto entrante per il prossimo passo).

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

Possiamo facilmente ricavare il risultato che dà ogni tripla di argomenti su un singolo bit del risultato, e il riporto entrante generato:

$\mathbf{x_i}$	$\mathbf{y_i}$	$\mathbf{r_i}$	$(\mathbf{x_i}+\mathbf{y_i}+\mathbf{r_i})_{10}$	$\mathbf{s_i}$	$\mathbf{r_{i+1}}$
0	0	0	0	0	0
1	0	0	1	1	0
0	1	0	1	1	0
0	0	1	1	1	0
1	1	0	2	0	1
1	0	1	2	0	1
0	1	1	2	0	1
1	1	1	3	1	1

Questa tabella viene evaluta su ogni tripla di cifre x_i, y_i e riporto r_i incontrata, generando r_n riporti consecutivi.

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. Si ha infatti che, se X

e Y sono le rappresentazioni su n bit di due interi x e y, allora la rappresentazione di $s = x + y$ (se esprimibile su n bit) è data da $S = X + Y$.

Tolto il CF, il processore attiva i flag OF, ZF e SF secondo le modalità:

- **OF:** rappresenta l'overflow, ergo la non rappresentabilità, nel caso di somme intere, e guarda ai segni (il MSB):
  - **Segni discordi:** non c'é overflow;
  - **Segni concordi:** il risultato è corretto se è concorde con gli operandi.

  Alternativamente, si può pensare che l'OF viene attivato sulla base degli ultimi dure riporti. Se sono discordi (cioè $r_n \neq r_{n-1}$), si attiva;
- **ZF:** si attiva se la rappresentazione S finale è uguale a 0 (si ricorda che lo 0 è tale sia in base che in complemento a 2);
- **SF:** si attiva se il MSB è uguale a 1, che in complemento a due significa segno negativo.

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 (forse solo storicamente) è anche più veloce. Questo deriva dal fatto che la circuiteria che implementa l'incremento è (in teoria) 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 potrebbe sembrare effettivamente il contrario di quello della somma: si sottraggono le cifre del sottraendo e del minuendo, x_i e y_i, e un riporto entrante r_i al minuendo, per produrre:

- La possima cifra d_i del risultato;
- Un riporto uscente r_{i+1} (cioè il riporto entrante per il prossimo passo).

In verità risulta più conveniente usare la stessa circuiteria della somma, e adottare quindi semplicemente il complemento a 2. Abbiamo che:

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

ergo possiamo sfruttare il carry bit (che prendiamo come già impostato), ed eseguire la somma fra X e il complemento di Y. Questo ci dà il risultato corretto sia che si parli di naturali che di interi, visto che equivale a $X + (-Y)$.

In generale, l'algoritmo esatto usato per la sottrazione (sottrazione manuale con prestiti, o addizione del complemento) non è poi così importante. Dobbiamo però fare attenzione ai flag, che vengono impostati in modo diverso rispetto alla somma, ovvero:

- **CF:** non viene impostato sul riporto effettivamente generato dalla somma (in quanto 1) sarebbe irrilevante; 2) non siamo nemmeno sicuri che il complemento a 2 sia il modo in cui viene effettivamente svolta la sottrazione a livello ALU), ma sul **prestito** che si sarebbe dovuto fare nel caso si avesse avuto $Y > X$. In altre parole, nel caso di numeri naturali, il CF rappresenta se la somma è un naturale (quando è 0) o se è un intero negativo (quando è 1).
- **OF:** rappresenta l'overflow, ergo la non rappresentabilità, nel caso di sottrazioni intere, e guarda come per le somme, ai segni (il MSB):
  - **Segni concordi:** non c'è overflow;
  - **Segni discordi:** il risultato è corretto se è concorde col minuendo.
- **ZF:** si attiva se la rappresentazione S finale è uguale a 0 (si ricorda che lo 0 è tale sia in base che in complemento a 2);
- **SF:** si attiva se il MSB è uguale a 1, che in complemento a due significa segno negativo.

Si ricorda un'ultima volta: la circuiteria per la somma (e per la sottrazione) non è diversa fra naturali ed interi: è controllando i flag giusti che si riesce ad ottenere informazioni riguardo all'esito della somma, e i flag giusti sono noti solo se lo è il tipo di rappresentazione degli operandi (nozione che conosce solo il programmatore).

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
```

6 Lezione del 02-10-24

6.1 Effetti collaterali

I sottoprogrammi non dovrebbero avere effetti collaterali, ergo dovrebbero lasciare i registri come li trovano. Per fare ciò, si sfrutta la pila per immagazzinare i loro valori precedenti:

```
sottoprog:  PUSH ... # fai push dei registri
            PUSH ...
            ... # esegui il sottoprogramma
            MOV ..., %CX

```

```
6             POP ... # riprendi i resisti
7             POP ...
8             RET
```

Sono fondamentali due linee guida:

- Bisogna stare attenti ad operazioni come `IDIV` e `IMUL`, che sporcano registri come EDX implictamente;
- Bisogna far corrispondere una `POP` ad ogni `PUSH`, altrimenti si lascia la pila in uno stato inconsistente per il prossimo `RET`.

6.2 Sottoprogramma principale

Il `_main` va in esecuzione come un sottoprogramma, ergo deve terminare con una `RET` e lasciare in EAX un valore di ritorno (0 significa tutto ok, $\neq 0$ significa codice di errore). Per quanto ci riguarda, basterà scrivere `XOR %EAX, %EAX`.

6.3 Dichiarazione dello stack

Lo stack esiste se viene:

1. Dichiarato con una direttiva;
2. Inizializzato con il registro ESP.

Dichiarare significa allocare abbastanza memoria, e inizializzare significa impostare ESP alla cella successiva al fondo dello stack (si ricorda che lo stack si evolve verso sinistra). Ad esempio, potremo avere:

```
1 .DATA
2 ...
3 mystack:  .FILL 1024, 4 #dichiarazione stack
4 .SET      initial_esp, (mystack + 1024*4)
5
6 .TEXT
7 _main:    NOP
8           MOV $initial_esp, %ESP  # inizializzazione stack
```

Lo stack può essere grande a piacere del programmatore. Nel nostro ambiente (ma non in generale) possiamo omettere la dichiarazione.

La pila può essere anche usata per il passaggio dei documanti (è il metodo che usano i compilatori). Questo risulta difficile da fare a mano, e quindi è sconsigliato per programmi più semplici.

6.4 Sottoprogrammi di Input/Output

In assembler non esistono istruzioni di ingresso e uscita (tranne le `IN` e `OUT`, che però sappiamo essere privilegiate). Si usano quindi i servizi del sistema (DOS), ovvero sottoprogrammi scritti da altri che girano in modalità sistema. Questi servizi sono molto primitivi: permettono l'uscita di singoli caratteri. Esistono quindi sottoprogrammi (leggermente) più sofisticati per l'output di numeri, ecc...

6.4.1 I/O tastiera e video

Le informazioni che entrano ed escono da interfacce sono solo codifice ASCII di singoli caratteri. Infatti in assembler non esiste il concetto di I/O tipato di variabili.

Ricevere il numero 32 significa ottenere i caratteri '3' e '2', mentre stamparlo significa inviare i caratteri '3' e '2'. Questo chiaramente sui decimale si traduce in moltiplicazioni per 10 (in entrata) e divisioni per 10 con resto (in uscita) atte ad ottenere queste cifre.

6.4.2 I/O di caratteri e stringhe

Nel corso si userà il file di utilità `.INCLUDE "./files/utility.s"`. Questo file mette a disposizione alcuni sottoprogrammi fra cui:

- **inchar:** mette in AL la codifica ASCII del tasto premuto;
- **outchar:** mette sul video la codifica ascii contenuta in AL;
- **newline:** stampa `0x0D` (Carriage Return) e `0x0A` (Line Feed), ergo va a capo;
- **pauseN:** mette in pausa il programma e stampa a video:

```
Checkpoint number N. Press any key to continue
```

  dove N deve essere una cifra decimale.

Sopra questi sottoprogrammi sono state scritte routine più complesse:

- **inline:**
  - **Descrizione:** porta una stringa di massimo 80 caratteri in un buffer di memoria, digitando con eco su video.
  - **Parametri di ingresso:**
    - EBX: indirizzo di memoria del buffer;
    - CX: numero di caratteri da leggere (massimo 80, una linea).

  Questo programma legge effettivamente 78 caratteri utili, in quanto gli ultimi 2 sono obbligatoriamente il nuova linea. Il programma inoltre gestisce la pressione dei tasti invio (finisci di ottenere caratteri) e backspace (cancella caratteri).

- **outline, outmess:**
  - **Descrizione:** stampa a video massimo 80 caratteri da un buffer di memoria. Si ferma prima se trova un carattere di ritorno carrello, andando anche a capo.
  - **Parametri di ingresso:**
    - EBX: indirizzo di memoria del buffer;

- **inbyte, inword, inlong:**
  - **Descrizione:** prelevano da tastiera (con eco sul video) 2, 4 o 8 caratteri. Interpretano tale sequenza di caratteri come un numero esadecimale a 2, 4 o 8 cifre. Ignorano tutti gli altri caratteri.
  - **Parametri di ingresso:**
    - AL, AX, o EAX: il numero esadecimale digitato.

- **outbyte, outword, outlong:**
  - **Descrizione:** stampano a video 2, 4 o 8 caratteri, corrispondenti a cifre esadecimali.
  - **Parametri di ingresso:**
    * AL, AX, o EAX: il numero esadecimale da stampare.
- **indecimal_byte, indecimal_word, indecimal_long:**
  - **Descrizione:** prelevano da tastiera (con eco sul video) fino a 3, 5 o 10 cifre decimali. Interpretano tale sequenza di caratteri come un numero decimale.
  - **Parametri di ingresso:**
    * AL, AX, o EAX: il numero decimale digitato.

  Se il numero decimale è troppo grande viene troncato. Inoltre si può usare invio per dare ingresso a meno cifre.
- **outdecimal_byte, outdecimal_word, outdecimal_long:**
  - **Descrizione:** stampano a video caratteri corrispondenti a cifre decimali.
  - **Parametri di ingresso:**
    * AL, AX, o EAX: il numero decimale da stampare.

6.5 Manipolazione di stringhe e vettori

In assembler non esistono tipi di dati né strutture dati. Si supporta però il concetto di vettore: si dichiarano vettori di variabili di una certa dimensione, e si indirizzano i loro elementi attraverso l'indirizzamento complesso (displacement + base + indice $\times$ scala).

In verità esistono istruzioni stringa, che servono a copiare interi buffer di memoria, che sfruttano i registri ESI e EDI. Ad esempio, copiare un vettore a mano significherebbe:

```
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
```

ma abbiamo la possibilità di scrivere la stessa cosa come:

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

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

dove l'istruzione `REP MOVSL` indica ripetizione (prefisso `REP`), di movimento da stringa a stringa su long (`MOVSL`) finché ECX $\neq 0$.

6.5.1 Direction Flag

Esiste un'altro bit utile nel registro dei flag: il Direction Flag, o DF. Si imposta con le istruzioni:

- **STD**: SET DIRECTION FLAG, la imposta ad 1;
- **CLD**: CLEAR DIRECTION FLAG, la imposta a 0;

Si usa questo flag per dare indicazioni alla prossima istruzione:

6.5.2 MOVE DATA FROM STRING TO STRING (with REPEAT)

- **Formato:** `MOVSsuf`, `REP MOVSsuf`
- **Azione:** copia il numero di byte indicato dal suffisso *suf* dall'indirizzo di memoria puntato da ESI all'indirizzo di memoria puntato da EDI. Successivamente, SE DF è 1, sottrae da ESI e EDI il numero di byte indicati da *suf*, altrimenti li somma.

  Se si include il prefisso, le operazioni vengono ripetute decrementando ECX (come per `LOOP`).
- **Flag:** nessuno.

Esistono poi altre istruzioni di stringa, fra cui:

6.5.3 LOAD DATA FROM STRING

- **Formato:** `LODSsuf`
- **Azione:** copia in AL, AX, oppure EAX, il contenuto della memoria all'indirizzo puntato da ESI. Successivamente incrementa o decrementa ESI di 1, 2 o 4 a seconda di DF.
- **Flag:** nessuno.

6.5.4 STORE DATA TO STRING

- **Formato:** `LODSsuf`
- **Azione:** copia il registro AL, AX, oppure EAX, in memoria all'indirizzo puntato da EDI. Successivamente incrementa o decrementa EDI di 1, 2 o 4 a seconda di DF.
- **Flag:** nessuno.

Si dovrebbe essere notato che ESI sta per sorgente, ed EDI per destinatario. Vediamo quindi degli esempi:

Copia un vettore da una parte all'altra, eseguendo un'operazione su tutti i suoi elementi:

```
        MOV $1000, %CX
        LEA buffer_src, %ESI
        LEA buffer_dst, %EDI
        CLD
ciclo:  LODSL
        ... #modifica %EAX
        STOSL
        LOOP ciclo
```

Riempi un buffer in memoria di zeri:

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

6.5.5 Istruzioni stringa per l'I/O

Esistono delle istruzioni stringa di ingresso e uscita:

6.5.6 INSERT STRING

- **Formato:** `INSsuf`
- **Azione:** fa ingresso di 1, 2 o 4 byte dalla porta di I/O il cui offset è contenuto in DX. L'operando viene inserito in memoria a partire dall'indirizzo contenuto in EDI. Successivamente incrementa o decrementa EDI di 1, 2, o 4 a seconda di DF.
- **Flag:** nessuno.

6.5.7 OUTPUT STRING

- **Formato:** `INSsuf`
- **Azione:** fa uscita di 1, 2 o 4 byte dall'indirizzo di memoria contenuto in EDI. L'operando viene inserito nella porta di I/O il cui offset è contenuto in DX. Successivamente incrementa o decrementa ESI di 1, 2, o 4 a seconda di DF.
- **Flag:** nessuno.

6.5.8 Istruzioni di confronto su stringhe

Vediamo infine alcune istruzioni per effettuare confronti su e fra stringhe:

6.5.9 COMPARE STRINGS

- **Formato:** `CMPSsuf`
- **Azione:** confronta il valore delle locazioni (singole, doppie o quadruple) indicate da ESI (sorgente) ed EDI (destinatario). Successivamente incrementa o decrementa ESI di 1, 2, o 4 a seconda di DF.
- **Flag:** nessuno.

6.5.10 SCAN STRING

- **Formato:** `SCASsuf`
- **Azione:** confronta il contenuto del registro AL, AX o EAX con la locazione (singola, doppia o quadrupla) di memoria indirizzata da EDI. L'algoritmo di confronto è lo stesso di CMP. Successivamente incrementa o decrementa ESI di 1, 2, o 4 a seconda di DF.
- **Flag:** nessuno.

Quest'espressione si usa per trovare valori noti dentro un vettore con, $\text{DF} = 0$ che cerca la prima occorrenza, e $\text{DF} = 1$ che cerca l'ultima. Ad esempio, poniamo di voler trovare il primo elemento differente fra due vettori:

```
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 array 1, %ESI
LEA array2, %EDI
MOV $10, %ECX
REPE CMPSW
```

dove si noti che alla fine del ciclo EDI e ESI puntano all'elemento successivo.

6.5.11 Prefissi di ripetizione

Vediamo nel dettaglio il prefisso `REP`, e le sue varianti `REPE` e `REPNE`. Bisogna ricordare che questi prefissi si applicano ad istruzioni, non a blocchi di codice.

- `REP`: si può usare con `MOVS`, `LODS`, `STOS`, `INS` e `OUTS`, anche se l'utilizzo con `LODS` è privo di senso (almeno che non si voglia ottenere l'ultimo elemento...).
- `REPE` e `REPNE`: si può usare con `CMPS` e `SCAS`, ed effettua al massimo ECX ripetizioni, finché la condizione specificata è vera.

6.5.12 Perchè due direzioni?

L'uso di due direzioni di scorrimento di stringhe attraverso il flag DF è utile, sopratutto nel caso si debbano fare traslazioni del vettore (copia di buffer **parzialmente sovrapposti**). Infatti, cercando si spostare il vettore a destra spostandoci verso destra, finiremo per copiare sempre gli stessi dati.

6.6 Note sull'efficienza

Un compilatore ottimizza il codice in alto livello per il sistema su cui quel codice dovrà girare. Un assemblatore, invece, traduce le istruzioni una per una.

6.6.1 Tempo di esecuzione di un processo

Un processo è un programma in esecuzione con dei dati. In questo, dipende dai dati, dallo stato del sistema, e da cosa sta facendo il processore (chi lo sta usando?). Questo rende il calcolatore una macchina poco prevedibile, e il tempo di esecuzione del processo difficile da calcolare a priori. Di base, infatti:

- Il clock non va a velocità costante;
- Il vostro processo non necessariamente gira su un solo core;
- Altri meccanismi introducono variabilità considerevoli:
  - Memorie cache;
  - Code di prefetch;
  - Esecuzione in pipeline: eseguire un'istruzione significa fare fetch dell'istruzione, recuperare l'OPCODE, il sorgente, scrivere sul destinatario, ecc... conviene eseguire queste operazioni in pipeline, cioè eseguendo in parallelo più istruzioni possibili contemporaneamente;

- Esecuzione non sequenziale: il processore non esegue necessariamente il codice nell'ordine in cui è scritto: se possibile, modifica l'ordine in modo dal caricare in modo più efficiente possibile la pipeline;
- Branch prediction: quando si esegue in pipeline, le istruzioni condizionali creano forti bottleneck di prestazioni. Per ovviare a questo problema, il processore cerca di predire il tipo della prossima istruzione, pagando un prezzo nel caso si sbagli, ma ottenendo un significativo incremento di velocità nel caso abbia successo.

6.6.2 Lunghezza delle istruzioni e tempo di fetch

Il numero di byte occupati da un'istruzione dipende dall'OPCODE e dal tipo di indirizzamento. Se gli operandi sono **registri**, le istruzioni stanno normalmente su 1 byte; gli operandi **immediati** devono essere codificati (in 1, 2 o 4 byte); i **displacement** occupano 4 byte.

La lunghezza delle istruzioni, oltre alle dimensioni dei file binari, influenza anche il tempo di fetch delle stesse, e va quindi tenuto in considerazione.

6.6.3 Tempo di esecuzione delle istruzioni

Il tempo di esecuzione delle istruzioni dipende molto dall'architettura specifica del processore (anche in processori della stessa famiglia).

Abbiamo che le istruzioni ALU (escluse MUL e DIV) costano poco, su O(1) cicli di clock. Le MUL e DIV costano sui O(10) cicli di clock, e per questo vengono tradotte in procedure alternative (attraverso LEA o le istruzioni di shift) dai compilatori attraverso apposite tablle di corrispondenza.

Le operazioni più costose sono quelle dellA FPU (Floating Point Unit), che richiedono sulle O(100) istruzioni.

Anche le istruzioni condizionali sono molto costose, ma per i motivi visti prima che rallentano le pipeline.

7 Lezione del 03-10-24

7.1 Assembler a 64 bit

Finora abbiamo studiato il linguaggio assembler a 32 bit (registri estesi EAX, EBX, ecc...). Vediamo adesso alcune caratteristiche dell'assembler a 64 bit.

Nei processori a 64 bit Intel-AMD x86 abbiamo 16 registri generali a 64 bit, con prefisso R, e che quindi si indicano come RAX, RBX, ecc... Di questi si può indirizzare la parte estesa dei 32 bit meno significativi (EAX), i 16 bit meno significativi (AX), e gli 8 bit meno significativi (AL). Per RAX, RBX, RCX e RDX si possono inoltre indirizzare gli 8 bit precedenti ad AL, BL, CL e DL usando AH, BH, CH e DH, ma questo è sconsigliato in quanto ci sono diverse limitazioni (non sono compatibili col prefisso REX).

Una lista completa dei registri genrali è la seguente, inclusi i nomi dei sottoregistri di dimensione minore:

Ricordiamo poi i registri RIP, l'instruction pointer, e RFLAGS che è il registro dei flag.

64 bit	32 bit	16 bit	8 bit	8 bit (legacy)
RAX	EAX	AX	AL	AH
RBX	EBX	BX	BL	BH
RCX	ECX	CX	CL	CH
RDX	EDX	DX	DL	DH
RSP	ESP	SP	SPL	
RBP	EBP	BP	BPL	
RSI	ESI	SI	SIL	
RDI	EDI	DI	DIL	
R8	R8D	R8W	R8B	
R9	R9D	R9W	R9B	
R10	R10D	R10W	R10B	
R11	R11D	R11W	R11B	
R12	R12D	R12W	R12B	
R13	R13D	R13W	R13B	
R14	R14D	R14W	R14B	
R15	R15D	R15W	R15B	

7.1.1 Spazio indirizzabile

Tecnicamente con architettura a 64 bit si potrebbero indirizzare 2^{64} byte distinti, ma i processori moderni permettono di indirizzarne solo $2^{48} = 256$ TiB, con alcuni modelli più recenti che arrivano a $2^{57} = 128$ PiB. I 47 (o 56) bit occupati sono i meno significativi, e i restanti 16 (o 7) devono avere il valore del bit più significativo utilizzato. Questo significa che sono indirizzabili effettivamente due porzioni contigue ma separate fra di loro di memoria:

	48 bit	57 bit
Regione alta	`0000 0000 0000 0000`	`0000 0000 0000 0000`
	`0000 7fff ffff ffff`	`01ff ffff ffff ffff`
Regione bassa	`ffff 8000 0000 0000`	`fe00 0000 0000 0000`
	`ffff ffff ffff ffff`	`ffff ffff ffff ffff`

Lo spazio I/O, infine, è di $2^{16} = 64$ KiB locazioni.

7.1.2 Istruzioni

Le operazioni possono possono usare 1, 2, 4 o 8 byte per un operando (rispettivamente Byte, Word, Long e Quad).

Notiamo che non possiamo usare displacement o operandi immediati a 64 bit: siamo limitati a 32 bit. Per ovviare a questo problema esiste una versione alternativa della `MOV`:

7.1.3 MOVABS

- **Formato:** `MOVABS $const, destination`
- **Azione:** porta una costante a 64 bit (che ci permette di scrivere) in un indirizzo generale.
- **Flag:** nessuno.

Operandi	Esempi
Immediato	`MOVABS $0xffff8105402300ef, %RBX`
Memoria	`CALL 0x00ef0b2a, %RAX`
Registro	`CALL %RAX, 0x00ef0b2a`

In generale, in assember a 64 bit si usano registri con valori base di 64 bit, e poi si indirizza con displacement a 32 bit, che in complemento a 2 concedono $\pm 2^{32}$, ergo $\pm 2\text{GB}$ di memoria indirizzabile rispetto alla base.

7.2 Reti logiche

Una rete logica è un modello astratto di un sistema fisico, costituito da dispositivi tra loro interconnessi. Le informazioni vengono codificate da questi dispositivi attraverso fenomeni fisici che si presentano in due aspetti distinti (corrente forte / corrente debole, tensione forte / tensione debole, magnetizzazione / non magnetizzazione, ecc...).

7.2.1 Caratterizzazione di rete logica

Una rete logica è caratterizzata da:

- Un'insieme di N variabili di ingresso. Il loro valore all'istante temporale t si chiama stato di ingresso. L'insieme di tutti i 2^N stati di ingresso si indicherà come $X.X = \{x_{N-1}x_{N-2}...x_1x_0\}$.
- Un'insieme di M variabili di uscita. Il loro valore all'istante temporale t si chiama stato di uscita. L'insieme di tutti i 2^M stati di uscita si indicherà come $Z.Z = \{x_{M-1}x_{M-2}...x_1x_0\}$.
- Una legge di evoluzione che determina come le uscite si evolvono in funzione degli ingressi.

Possiamo classificare le reti logiche in base a 2 criteri riguardanti l'evoluzione nel tempo:

- **Presenza/assenza di memoria**:
  - **Reti combinatorie:** analoghe a funzioni matematiche, le loro uscite dipendono solo dai loro ingressi in un qualsiasi istanti t;
  - **Reti sequenziali:** lo stato di uscita dipende dalla storia degli ingressi precedenti, ergo sono reti con memoria.
- **Temporizzazione della legge di evoluzione:**
  - **Reti asincrone:** l'aggiornamento delle uscite avviene costantemente nel tempo;
  - **Reti sincronizzate:** l'aggiornamento delle uscite avviene ad istanti di sincronizzazione discreti nel tempo.

I modelli sono ortogonali, ergo possiamo avere qualsiasi delle 4 combinazioni di queste caratteristiche:

- Reti combinatorie (si considerano le sincronizzate come caso particolare);

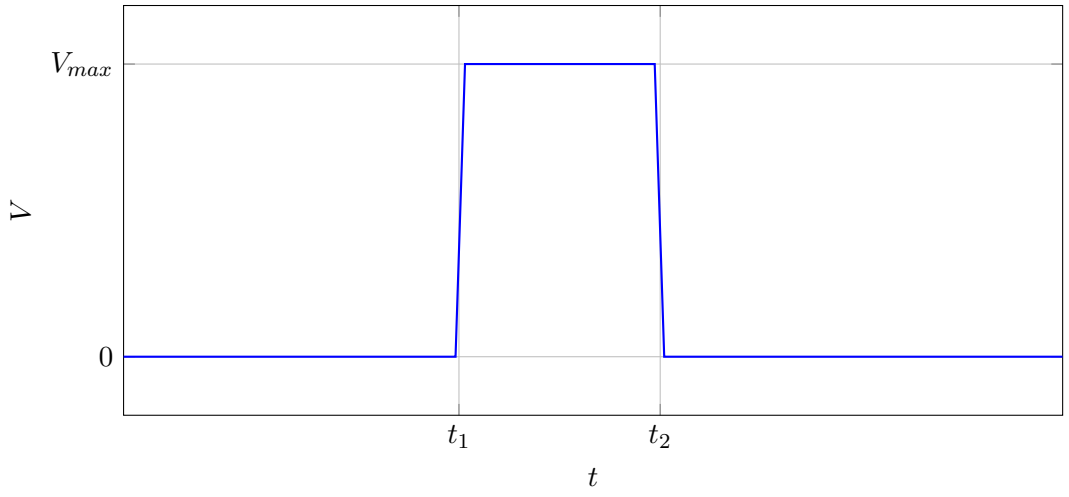
- Reti sequenziali asincrone;
- Reti sequenziali sincronizzate.

Quindi in sostanza una rete logica comunica con l'esterno attraverso variabili logiche (0 e 1). L'interpretazione di questi messaggi è una convenzione del progettista, programatore, ecc...

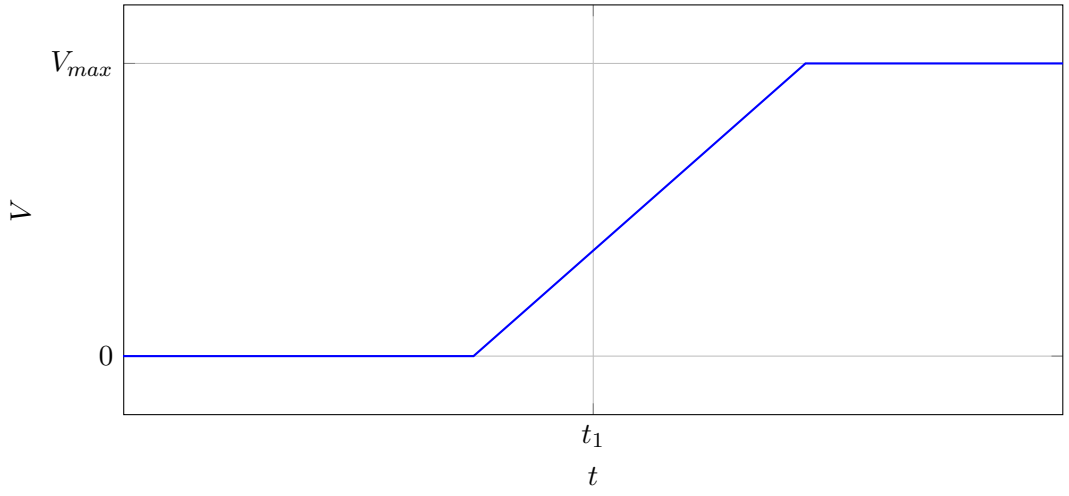
Usiamo le reti logiche per modellizzare circuiti elettronici all'interno del calcolatore, che codificano le informazioni in tensione. Notiamo quindi che una rete logica fisica ha, oltre agli ingressi e alle uscite, i collegamenti ai terminali positivi e negativi di un generatore di tensione, che noi ignoreremo.

7.3 Transizione dei segnali

Una variabile logica (per noi il voltaggio su un circuito) può settarsi (andare a 1), restare settato per tempi paragonabili a ΔT, e resettarsi (andare a 0) in un qualsiasi momento temporale t:

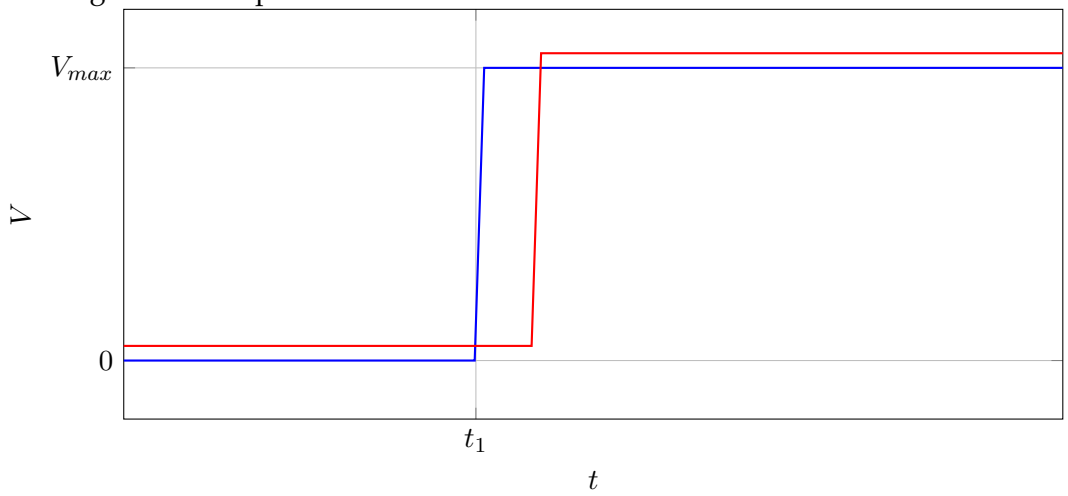


In un sistema fisico reale, durante la transizione c'è un periodo di indecisione in cui il voltaggio sale o scende fisicamente fino al valore necessario, sotto l'atto di una qualche potenza. Vediamo il grafico a $\Delta t << \Delta T$:



Decidiamo di ignorare questo problema, in quanto abbiamo visto che il Δt di transizione è molto più piccolo del Δt di stasi delle variabili.

Il problema si presenta nel caso si parli di **contemporaneità**. Supponiamo di avere una rete logica con due ingressi x_0 e x_1 e un'uscita z_0. Abbiamo che prima dell'istante t_1 lo stato di ingresso è $(1, 0)$, e che subito dopo lo stesso stato è $(0, 1)$. Nell'istante di transizione non abbiamo la sicurezza che le singole transizioni delle due variabili della rete avvengano contemporaneamente:



Questa considerazione è importante nel caso delle reti logiche asincrone, dove considerare le transizioni come contemporanee potrebbe portare alla comparsa di stati di uscita spuri, e nelle reti sequenziali, dove potrebbe portare ad evoluzioni imprevedibili del sistema.

7.4 Reti combinatorie

Una rete combinatoria è caratterizzata da:

- Un'insieme di N variabili logiche di ingresso;
- Un'insieme di M variabili logiche di uscita;
- Una descrizione funzionale $F : X \to Z$ che mappa stati di ingresso a stati di uscita;
- Una legge di evoluzione nel tempo che adegua $F(X)$ allo stato di ingresso X continuamente.

7.4.1 Tempo di attraversamento

Il tempo di attraversamento (o di accesso) è una caratteristica di tutte le reti logiche asincrone: è il tempo necessario perché la rete si "accorga" della variazione degli ingressi e aggiorni di conseguenza le sue uscite.

Questo tempo è solitamente non nullo, ed è quindi necessario attendere che la rete arrivi a **regime** prima di valutare le uscite. Questo vincolo prende il nome di **pilotaggio in modo fondamentale**: si dice che è una rete è pilotata in modo fondamentale quando chi la pilota aspetta sempre che essa arrivi a regime prima di valutare le sue uscite.

8 Lezione del 08-10-24

8.1 Descrizione funzionale

La caratteristica più importante di una rete combinatoria è la funzione F, cioé la descrizione funzionale. Esistono più modi per esprimere questa funzione:

- A parole;
- Usando notazioni testuali (e..g. il Verilog);
- Attraverso **tabelle di verità**. In una tabella di verità contiene due insiemi di colonne: gli ingressi e le uscite. Ogni riga mostra una configurazione di stati di ingresso e il corrispondente stato d'uscita. Ad esempio:

x_2	x_1	x_0	z_1	z_0
0	0	0	0	0
0	0	1	$-$	1
0	1	0	1	0
...				

  Si dice che la variabile di uscita **riconosce** particolari stati quando si attiva in presenza di essi. Inoltre, i trattini indicano stati **non specificati**, in inglese DC, *don't care*. Questi non equivalgono alla fascia di indeterminazione, ma a uno dei due stati accettati, anche se non è importante quale. I *don't care* vanno conservati, e non fissati a variabili come 0 o 1, in quanto è importante mantenere il funzionamento interno delle reti il più semplice possibile.

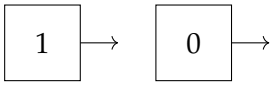
8.1.1 Descrizione e sintesi

Una **descrizione** di una rete deve essere formale, in modo che si possa capire esattamente cosa fa quella rete. La **sintesi** di una rete è il progetto stesso di realizzazione della rete, cioè quali componenti combinare in quale modo, ecc... Prima si fa la descrizione, e poi la sintesi.

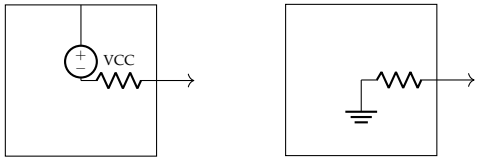
Notiamo una proprietà fondamentale: ogni rete combinatoria di N ingressi e M uscite può essere realizzata interconnettendo M reti combinatorie ad N ingressi ed una uscita. Questo ci permette di trattare tutte le reti con reti con una sola uscita.

8.2 Reti a 0 ingressi

Le reti a 0 ingresso di uscita si chiamano **generatori di costante**, e rappresentano un caso degenere. Si indicano come:



La loro uscita chiaramente vale 1 o 0 costante. Fisicamente, i generatori di costante si realizzano collegando resistori in serie al VCC (genera 1) o a massa (genera 0), ergo:



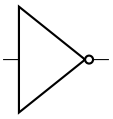
8.3 Reti a 1 ingresso

8.3.1 Invertitore

L'invertitore, detto anche porta **NOT** è una rete descritta dalla tabella di verità:

x	z
0	1
1	0

e indicata come:



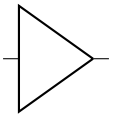
Essenzialmente nega il suo ingresso.

8.3.2 Elemento neutro

L'elemento neutro, detto anche *buffer*, è una rete descritta dalla tabella di verità:

x	z
0	0
1	1

e indicata come:



Lascia il suo ingresso invariato. Può avere un utilità come rete di rallentamento, in quanto, inevitabilmente, si perde tempo per attraversarla (pensa alla NOP). Questo è utile per le temporizzazioni delle reti.

Inoltre, dal punto di vista elettrico, l'elemento neutro ha anche un utilità per la **rigenerazione** dei segnali. Infatti, essendo collegato a massa e al VCC, può prendere segnali scadenti (vicini alla fascia di indeterminazione) e trasformarli in segnali di buona qualità (vicini al fondoscala). Questa proprietà, veramente, è comune a tutte le reti logiche, ma l'elemento neutro è l'unico che non ha altri effetti collaterali.

8.3.3 Reti costanti

Si possono interpretare i generatori di costante come reti ad un ingresso degeneri. Effettivamente, restano tali a se stesse, in quanto gli ingressi sono ignorati. Le loro tabelle di verità sono:

Generatore di 1:

x	z
0	1
1	1

Generatore di 0:

x	z
0	0
1	0

8.4 Reti a 2 ingressi

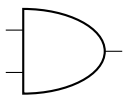
La prima domanda da porsi quando si parla di reti a 2 (come N) ingressi, è quante reti possiamo creare in tutto. Su N ingressi, la tabella di verità avrà 2^N righe. Le configurazioni possibili di 0 e 1 su 2^N righe sono 2^{2^N}. Ergo, nel caso $N = 2$, abbiamo $2^{2^2} = 16$ possibili combinazioni, che sono:

x_1	x_0	z^0	z^1	z^2	z^3	z^4	z^5	z^6	z^7	z^8	z^9	z^{10}	z^{11}	z^{12}	z^{13}	z^{14}	z^{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Ad alcune di queste corrispondono nomi speciali. Vediamole nel dettaglio:

8.4.1 Porta AND

La porta AND, indicata in z^1 , corrisponde al $\wedge$ logico, ergo $z = 1 \Leftrightarrow x_0 = x_1 = 1$. Si indica come:

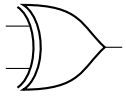


e ha tabella di verità:

x_1	x_0	z
0	0	0
0	1	0
1	0	0
1	1	1

8.4.2 Porta XOR

La porta XOR, indicata in z^6 , corrisponde all'*aut* logico, cioè esclusivo, ergo $z = 1 \Leftrightarrow x_0 \neq x_1$. Si indica come:

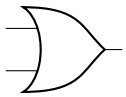


e ha tabella di verità:

x_1	x_0	z
0	0	0
0	1	1
1	0	1
1	1	0

8.4.3 Porta OR

La porta OR, indicata in z^7 , corrisponde al $\vee$ logico, ergo $z = 0 \Leftrightarrow x_0 = x_1 = 0$. Si indica come:

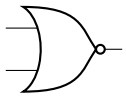


e ha tabella di verità:

x_1	x_0	z
0	0	0
0	1	1
1	0	1
1	1	1

8.4.4 Porta NOR

La porta NOR, indicata in z^8 , corrisponde alla negazione dell'$\vee$ logico, ergo $z = 1 \Leftrightarrow x_0 = x_1 = 0$. Si indica come:

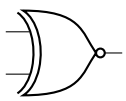


e ha tabella di verità:

x_1	x_0	z
0	0	1
0	1	0
1	0	0
1	1	0

8.4.5 Porta XNOR

La porta XNOR, indicata in z^9 , corrisponde alla negazione dell'*aut* logico, ergo $z = 1 \Leftrightarrow x_0 = x_1$. Si indica come:

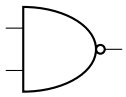


e ha tabella di verità:

x_1	x_0	z
0	0	1
0	1	0
1	0	0
1	1	1

8.4.6 Porta NAND

La porta NAND, indicata in z^14 , corrisponde alla negazione dell'$\wedge$ logico, ergo $z = 0 \Leftrightarrow x_0 = x_1 = 1$. Si indica come:



e ha tabella di verità:

x_1	x_0	z
0	0	1
0	1	1
1	0	1
1	1	0

Si dovrebbe essere notato che un pallino finale indica negazione. A volte si usa solo questa notazione, invece di tutta la porta NOT.

8.4.7 Casi degeneri

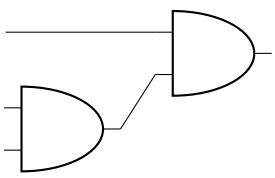
Alcuni casi speciali della tabella delle possibili reti a due porte sono degeneri: abbiamo due generatori di costante (z^0 e z^{15}), due elementi neutri, rispettivamente su x_1 e x_0 (z^3 e z_5), e due inversori sugli stessi ingressi (z_{10} e z_{12}).

8.5 AND e OR a più ingressi

Posso pensare di estendere AND e OR ad N ingressi:

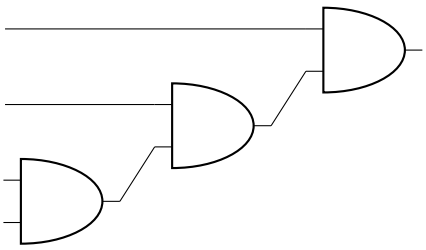
- **AND a N ingressi:** l'uscita vale 1 se tutti gli N ingressi valgono 1;
- **OR a N ingressi:** l'uscita vale 1 se almeno un'ingresso vale 1;

Questo può essere realizzato concatenando più porte logiche dello stesso tipo, come segue:

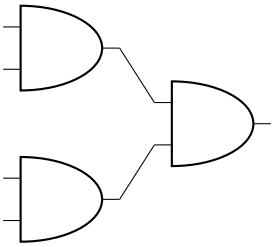


La dimostrazione è semplice dalla tabella di verità, o dalle proprietà degli operatori logici.

Una nota va fatta sulle combinazioni di più di 3 ingressi, infatti una rete del genere è sconveniente:



in quanto il segnale deve attraversare al massimo 3 livelli di logica, mentre disponendo le porte come:



il segnale dovrà attraversare al massimo 2 livelli di logica.

Conviene quindi disporre gli N ingressi e le relative porte come un'albero binario bilanciato, in modo da minimizzare gli attraversamenti di livelli di logica. Si noti che questo discorso vale per AND e OR: non per NAND, NOR, XOR o XNOR.

Possiamo osservare velocemente cosa accade se si collegano queste porte fra di loro:

- **NAND:** un singolo NAND può formare un NOT quando i suoi ingressi sono uniti insieme. Se si mettono 2 NAND in serie (a *cascata*) in questo modo, si ottiene di nuovo un AND;
- **NOR:** un singolo NOR può formare un NAND nello stesso modo del NAND. Se si mettono 2 NOR a cascata, si ottiene di nuovo un NOR;
- **XOR:** con ≥ 2 XOR, si crea effettivamente un controllore di parità, ergo una rete che si attiva quando un numero dispari dei suoi ingressi sono accesi;
- **XNOR:** con ≥ 2 XNOR, si ha l'opposto che con gli XOR: si crea una rete che si attiva quando un numero pari dei suoi ingressi sono accesi.

Queste porte si indicano solitamente come con gli input su unica orizzontale, che risulta più compatto.

8.6 Algebra di Boole

L'algebra di Boole adopera gli operatori logici conosciuti, applicati ad elementi del campo binario $GF(2) = \{0, 1\}$

Vediamo questi operatori:

- **Complemento logico:** si indica come $\overline{x}$, oppure $!x$ o $/x$. Si definisce come:

$$\overline{0} = 1, \quad \overline{1} = 0$$

- **Somma logica:** si indica con $x + y$, e ha tabella di verità:

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

  cioè equivale all'OR.

- **Prodotto logico:** si indica con $x \cdot y$, e ha tabella di verità:

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

  cioè equivale all'AND.

Su questi operatori valgono le proprietà:

1. **Involutiva del complemento:** $\overline{\overline{x}} = x$;
2. **Commutativa della somma e del prodotto:** $x + y = y + x, \quad x \cdot y = y \cdot x$;
3. **Associativa della somma:** $x + y + z = (x + y) + z = x + (y + z)$;
4. **Associativa del prodotto:** $x \cdot y \cdot z = (x \cdot y) \cdot z = x \cdot (y \cdot z)$;
5. **Distributiva della somma rispetto al prodotto:** $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$;
6. **Distributiva del prodotto rispetto alla somma:** $x+(y \cdot z) = (x+y) \cdot (x+z)$. Bisogna fare attenzione in quanto questa non vale in $\mathbb{R}$;
7. **Complementazione:** $x \cdot \overline{x} = 0, \quad x + \bar{x} = 1$;
8. **Unione e intersezione:** $x + 0 = x, \quad x + 1 = 1$, cioè 0 è l'elemento neutro e 1 l'elemento assorbente della somma (non lo è in $\mathbb{R}$);
   $x \cdot 0 = 0, \quad x \cdot 1 = x$, cioè 1 è l'elemento neutro e 0 l'elemento assorbente del prodotto;
9. **Idempotenza:** $x + x = x, \quad x \cdot x = x$, altra che non vale in $\mathbb{R}$;
10. **Leggi di De Morgan:** $\overline{x \cdot x} = \overline{x} + \bar{x}$ e $\overline{x + x} = \bar{x} \cdot \bar{x}$.

8.6.1 Teoremi di De Morgan

Le leggi di De Morgan comuni della logica si estendono ad N variabili come:

1. $\overline{x_0 \cdot x_1 \cdot ... \cdot x_n} = \overline{x}_0 + \bar{x}_1 + ... + \bar{x}_n$
2. $\overline{x_0 + x_1 + ... + x_n} = \overline{x}_0 \cdot \bar{x}_1 \cdot ... \cdot \bar{x}_n$

Dimostrazione per induzione
Richiamiamo le basi dell'induzione:

- Si dimostra che una proprietà vale per un certo numero n_0 (passo base);
- Si dimostra che se vale per un certo $n \geq n_0$, allora vale anche per $n + 1$.

Partiamo con le dimostrazioni classiche ottenute con le tabelle di verità:

x	y	$x \cdot y$	$\overline{x \cdot y}$	$\overline{x}$	$\bar{y}$	$\bar{x} + \bar{y}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	1	0

che ci portano a $n_0 = 2$. Posso quindi porre l'ipotesi:

$$\overline{x_0 \cdot ... \cdot x_{n-1}} = \overline{x}_0 + ... + \bar{x}_{n-1}$$

e la tesi:

$$\overline{x_0 \cdot ... \cdot x_{n-1} \cdot x_n} = \overline{x}_0 + ... + \bar{x}_{n-1} + \cdot x_n$$

A questo punto faccio il passo induttivo, sfruttando l'associatività del prodotto (o della somma), e quindi riscrivendo la tesi come:

$$\overline{\alpha \cdot x_n}, \quad \alpha = x_0 + ...x_{n-1}$$

dove notiamo la variabile introdotta α, se complementata, rispetta:

$$\overline{\alpha} = \overline{x_0 \cdot ... \cdot x_{n-1}} = \bar{x}_0 + ... + \bar{x}_{n-1}$$

dall'ipotesi.

Possiamo quindi svolgere il passaggio:

$$\overline{\alpha \cdot x_n} = \overline{\alpha} + \bar{x}_n = \bar{x}_0 + ... + \bar{x}_{n-1} + \bar{x}_n$$

che conferma la tesi.

8.6.2 Algebra di Boole e reti combinatorie

Esiste una corrispondenza fra l'algebra di Boole e le reti combinatorie. In particolare, si ha che:

- **Data una rete combinatoria**, (comunque complessa), è sempre possibile trovare un'espressione booleana che mette in relazione ogni sua uscita con gli ingressi (in verità un'espressione per ogni uscita);

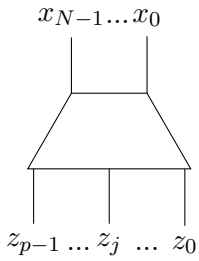
- **Data un'espressione booleana**, p sempre possibile sintetizzare una rete combinatoria (ad un'uscita) in cui la relazione tra ingresso ed uscita data è dall'espressione.

Si noti che, effettivamente, espressioni logiche equivalenti $\Leftrightarrow$ reti logiche che svolgono lo stesso compito, ma non per questo l'equivalenza è totale: ci conviene creare reti che usano meno componenti possibili, in quanto queste le rende più affidabili, più economiche e meno dispendiose di energia. Le proprietà dell'algebra di Boole possono quindi essere usate per ridurre il numero di porte logiche, attraverso un processo che chiameremo **minimizzazione**.

9 Lezione del 09-10-24

9.1 Decoder

Un decoder è una rete con N ingressi e p uscite con $p = 2^N$. Si indica come:



La sua legge di corrispondenza stabilisce che ogni uscita riconosce uno ed un solo stato di ingresso, in particolare l'uscita j-esima (z_j) riconosce lo stato di ingresso i cui bit sono la codifica di j in base 2, cioè:

$$(x_{n-1}, ..., x_0)_2 = j$$

Ad esempio, un decoder da 2 a 4 ha tabella di verità:

x_1	x_0	z_0	z_1	z_2	z_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

che equivale alla codifica *one-hot* del binario in ingresso (cioè ogni numero codificato da n bit viene mandato al j-esimo di p output che corrispondono uno ad uno ai numeri rappresentabili).

Vediamo di passare da questa descrizione ad una sintesi della rete. Abbiamo che:

$$\begin{cases} z_3 = x_1 \cdot x_0 \\ z_2 = x_1 \cdot \overline{x}_0 \\ z_1 = \overline{x_1} \cdot x_0 \\ z_0 = \overline{x_1} \cdot \overline{x_0} \end{cases}$$

cioè ogni "indice" del decoder corrisponde al prodotto dei due ingressi opportunamente negati: l'ultima uscità avra tutti i bit attivi (sarebbe $2^N - 1$ considerando numeri naturali),

ergo prende il prodotto di tutti gli ingressi. Di contro, la prima uscita (0) avrà tutti i bit disattivi, quindi prenderà il prodotto di tutti gli ingressi negati. Gli altri numeri vengono indirizzati prendendo il prodotto e complementando i bit che quel particolare numero si aspetterebbe come 0. Notiamo che, sebbene si abbiano 4 negazioni, nella rete fisica conviene negare gli input in entrata risparmiando 2 invertitori.

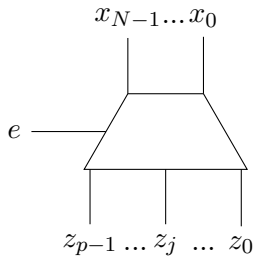
Per le figure, rimandiamo a `https://github.com/Guray00/IngegneriaInformatica/blob/master/SECONDO%20ANNO/I%20SEMESTRE/Reti%20Logiche/Diapositive%20OCR/Reti%20combinatorie%20ocr.pdf`.

Generalizziamo quindi questa struttura a decoder da N a 2^N, applicando quanto detto prima. Si avrà:

$$\begin{cases} z_0 = \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot \overline{x_0} \\ z_1 = \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot x_0 \\ ... \\ z_{p-2} = x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot \overline{x_0} \\ z_{p-1} = x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot x_0 \end{cases}$$

9.1.1 Decoder con enabler

Il problema dei decoder come appena descritti è che non sono espandibili: non si possono costruire, come avevamo visto per i gli AND o gli OR, reti di più decoder combinati. Introduciamo per questo motivo il decoder con **enabler**:



Questi decoder hanno $N + 1$ ingressi, cioè quelli normali più l'enabler, che ha il compito di "accendere" il decoder stesso. Fisicamente, potremmo semplicemente inserire il decoder e come ingresso aggiuntivo agli AND già predisposti, per avere che:

$$z_i = \begin{cases} y_i & e = 1 \\ 0 & e = 0 \end{cases}$$

e quindi:

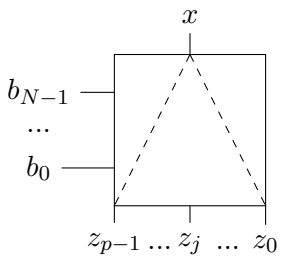
$$\begin{cases} z_0 = e \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot \overline{x_0} \\ z_1 = e \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot x_0 \\ ... \\ z_{p-2} = e \cdot x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot \overline{x_0} \\ z_{p-1} = e \cdot x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot x_0 \end{cases}$$

Adesso basta accorgersi che reti di decoder con $N > 2$ possono crearsi concatenando decoder a decoder, cioè usando un decoder con i bit più significativi in entrata per generare l'enabler di N nuovi decoder, i quali ricevono i bit meno significativi in entrata.

Ad esempio, se vogliamo creare un decoder `4to16` a partire da decoder `2to4`, useremo 4 decoder, con gli stessi input (x_0 e x_1), abilitati da un quinto decoder con input x_2 e x_3.

9.2 Demultiplexer

Il demultiplexer è una rete con $N+1$ ingressi e $p = 2^N$ uscite:



Chiamiamo x la **variabile da commutare**, e le altre **variabili di comando** (b). La j-esima uscita insegue la variabile da commutare se e solo se:

$$(b_{n-1}, ..., b_0)_2 = j$$

altrimenti vale 0. Questo significa che il demultiplexer invia il suo input, x, all'output z_j tale che i controlli $b_{N-1}...b_0$ sono la codifica binaria di j.

Il multiplexer, fisicamente, è identico ad un decoder con enabler: si fa la parte di decoding con il:

$$\begin{cases} z_0 = \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot \overline{x_0} \\ z_1 = \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot x_0 \\ ... \\ z_{p-2} = x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot \overline{x_0} \\ z_{p-1} = x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot x_0 \end{cases}$$

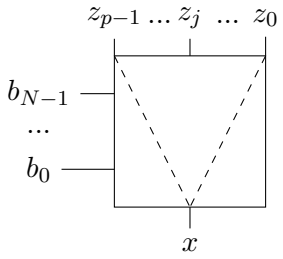
di prima, e si moltiplica per x per ottenere il comportamento desiderato:

$$\begin{cases} z_0 = x \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x}_1 \cdot \overline{x_0} \\ z_1 = x \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{x_1} \cdot x_0 \\ ... \\ z_{p-2} = x \cdot x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot \overline{x_0} \\ z_{p-1} = x \cdot x_{N-1} \cdot x_{N-2} \cdot ... \cdot x_1 \cdot x_0 \end{cases}$$

Con $x = e$ questo è un decoder con enabler x.

9.3 Multiplexer

Il multiplexer è il duale del demultiplexer: una rete con $N + 2^N$ ingressi e 1 uscita:



Gli ingressi b_i si chiamano variabili di comando, e selezionano l'ingresso connesso all'uscita come:

$$z = x_i \Leftrightarrow (b_{N-1}, ..., b_1, b_0) = i$$

Abbiamo detto che il multiplexer è il duale del demultiplexer: se quest'ultimo prendeva un segnale x e lo inviava al j-esimo output sulla base della codifica di j ottenuta alle variabili di controllo, il multiplexer prende il j-esimo ingresso, secondo gli stessi canoni, e lo invia alla linea x di uscita.

Alla base della sintesi di un multiplexer sta un decoder: infatti, abbiamo che quest'ultimo seleziona uno solo (*one-hot*) degli output, che possiamo moltiplicare (mettiamo una AND) per l'ingresso corrispondente. Visto che solo uno degli output in uscita dagli AND è attivo in un dato momento, possiamo ricombinare il segnale finare con un unico grande OR.

Come prima, possiamo eliminare gli AND in cascata dal decoder connettendoli agli AND già contenuti in esso.

Otteniamo quindi la descrizione algebrica (si noti che adesso abbiamo fatto sintesi $\rightarrow$ descrizione, mentre fino a questo punto avevamo fatto l'operazione inversa, descrizione $\rightarrow$ sintesi):

$$\begin{aligned} z = x_0 \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot ... \cdot \overline{b_1} \cdot \overline{b_0} + \\ x_1 \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot ... \cdot \overline{b_1} \cdot b_0 + \\ ... + \\ x_{p-2} \cdot b_{N-1} \cdot b_{N-2} \cdot ... \cdot b_1 \cdot \overline{b_0} + \\ x_{p-1} \cdot b_{N-1} \cdot b_{N-2} \cdot ... \cdot b_1 \cdot b_0 \end{aligned}$$

Notiamo che il multiplexer è una rete a 2 livelli di logica: il segnale passerà al massimo da un AND e un OR. Le NOT sugli ingressi non si contano, in quanto in una rete fisica le variabili di comando proverranno da registri, che forniscono già una versione negata del loro output senza bisogno di ulteriori inversori.

9.3.1 Multiplexer come rete combinatoria universale

Dimostriamo il seguente teorema:

Teorema 9.1: Multiplexer come rete combinatoria universale

Un multiplexer con N variabili di comando è in grado di realizzare qualunque legge combinatoria ad N ingressi ed un uscita, connettendo i 2^N ingressi a generatori di costante.

Abbiamo che:

- Un multiplexer si ricava con porte AND, OR e NOT a due livelli di logica;

- Un multiplexer realizza qualsiasi rete combinatoria ad un'uscita;
- una rete a più uscite può essere scomposta in più reti con le uscite messe "in parellelo".

Allora qualsiasi rete combinatoria può essere creata combinando AND, OR e NOT su due livelli di logica.

Inoltre, si può dimostrare che per qualsiasi tabella di verità ad N ingressi, si può trovare una rete che la implementa tramite un multiplexer a $N-1$ variabili di comando, e al più porte NOT.

9.4 Modello strutturale universale per reti combinatorie

Vediamo adesso un modo per sintetizzare una rete logica ad N ingressi ed M uscite a partire da una tabella di verità. Si prende prima di tutto un decoder con N ingressi, e si creano M linee parallele alle 2^N (che è anche il numero delle righe della tabella di verità) linee di uscita del decoder. Si combinano quindi queste linee di uscita attraverso OR su ogni intersezione che corrisponde ad una certa cella della tabella di verità.

9.4.1 Riduzione dei costi

Definiamo informalmente il costo come ridotto quando si usano meno porte logiche. Troviamo quindi un modo per ridurre il costo della rete creata. Avremo che, inizialmente, tutte le uscite si presentano in una forma canonica **SP**, che sta per Somma di Prodotti, del tipo:

$$z_j = x_{n-1} \cdot ... \cdot x_0 + ... + x_{n-1} \cdot ... \cdot x_0$$

con la possibilità di complementare qualsiasi x. Questa forma equivale effettivamente a una forma normale disgiuntiva.

Possiamo quindi usare le proprietà dell'algebra di Boole per raggruppare e semplificare i termini. Vogliamo un algoritmo che ci permetta di eseguire questi passaggi in modo ordinato, e ci porti sempre alla soluzione ottimale.

10 Lezione del 10-10-24

10.1 Sintesi di reti in forma SP a costo minimo

Esistono due criteri di costo per le reti:

- **A porte:** ogni **porta** conta per un'unità di costo;
- **A diodi:** ogni **ingresso** conta per un'unità di costo.

Presentiamo un metodo, applicabile a reti con'un uscita, che produce reti in forma SP a 2 livelli di logica in quanto, per una legge combinatoria F, si ha::

$$\text{Sintesi di F a 2 L.L. in forma SP} \subset \text{Sintesi di F a 2 L.L.} \subset \text{Sintesi di F}$$

10.1.1 Espansione di Shannon

Si può dimostrare il seguente risultato:

Teorema 10.1: Espansione di Shannon

Si può sempre scrivere qualunque legge combinatoria f come somma di prodotti degli ingressi (diretti o negati).

Questo significa che, se ho una legge combinatoria $z = f(x_{N-1}, ..., X_0)$, posso dire:

$$\begin{aligned} z = f(0,...,0,0) \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{X_1} \cdot \overline{X_0} + \\ f(0,...,0,1) \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot ... \cdot \overline{X_1} \cdot X_0 + \\ ... + \\ f(1,...,1,0) \cdot x_{N-1} \cdot x_{N-2} \cdot ... \cdot X_1 \cdot \overline{X_0} + \\ f(1,...,1,1) \cdot x_{N-1} \cdot x_{N-2} \cdot ... \cdot X_1 \cdot X_0 \end{aligned}$$

che equivale a quanto avevamo visto con la sintesi di reti combinatorie a N ingressi con multiplexer a N variabili di comando.

A questo punto possiamo ottenere la cosiddetta **forma canonica SP**, applicando le proprietà:

$$\begin{cases} 1 \cdot \alpha = \alpha \\ 0 \cdot \alpha = 0 \\ 0 + \beta = \beta \end{cases}$$

all'espansione di Shannon (sostanzialmente rimuoviamo tutti i termini a cui corrispondono uscite negate). Della forma canonica SP possiamo dire che è:

- **SP:** è fatta da somme e prodotti;
- **Canonica:** ogni prodotto ha come fattori tutti gli ingressi, diretti o negati;
- Ciascuno dei termini della somma si chiama **mintermine**;
- Ogni mintermine corrisponde ad uno stato riconosciuto dalla rete.

L'insieme dei termini (mintermini) sommati fra di loro che otteniamo dall'espansione di Shannon prende il nome di **lista di mintermini**.

10.2 Semplificazione della forma canonica SP

Definiamo quindi un metodo per la semplificazione della lista dei mintermini. Divideremo quest'operazione in due passaggi principali:

- **Identificazione degli implicanti principali**: si ricava una lista di termini ricavati da quelli di partenza, e di dimensioni più piccole, che rappresentano la stessa legge combinatoria;
- **Eliminazione delle ridondanze**: si rimuovono gli implicanti che non portano informazioni utili alla legge combinatoria.

10.2.1 Metodo di Quine-McCluskey

Si presenta il metodo di Quine-McCluskey per l'identificazione degli implicanti principali. Questo metodo prevede di:

- **Fondere i mintermini** applicando **esaustivamente** la regola:

  $$\alpha x + \alpha \bar{x} = \alpha$$

  che possiamo dimostrare come:

  $$\alpha x + \alpha \bar{x} = \alpha(x + \bar{x}) = \alpha, \quad x + \bar{x} = 1$$

  alla lista dei mintermini.

  Ripetiamo questo passaggio $N - 1$ volte per la dimensione N dei termini, riducendo ogni volta la dimensione degli implicanti di 1. Si ricava una forma SP, detta **lista di implicanti**.

- **Rimuovere i duplicati** dalla lista dei duplicanti, applicando l'altra regola:

  $$\alpha x + \alpha = \alpha$$

  sugli implicanti che hanno elementi in comune.

Troviamo quindi quella che è detta **lista degli implicanti principali**. Questa lista contiene meno elementi della forma canonica SP, ma non è ancora di costo minimo: potrebbe contenere ridondanze, cioè implicanti non necessari alla corretta modelizzazione della legge combinatoria.

10.2.2 Liste di copertura non ridondanti

Una **lista di copertura** è una lista di implicanti, la cui somma è una forma SP per la funzione f. La **lista di copertura non ridondante** è la lista che smette di essere una lista di copertura appena si toglie un elemento.

Un punto importante è che la lista dei mintermini è una lista non ridondante, mentre la lista degli implicanti principali può esserlo. Si introduce quindi uno strumento per la visualizzazione degli implicanti e le loro ridondanze.

10.3 Mappe di Karnaugh

Per una rete a N ingressi la corrispondente **mappa di Karnaugh** è una matrice di 2^N celle, dove le coordinate rappresentano gli ingressi, e gli elementi della matrice le uscite. Sono diagrammi che tornano utili per rappresentare graficamente gli implicanti, ed eliminarne le ridonanze. Vediamo, ad esempio, mappe con $N = 2$, 3 e 4:

		X_0	
		0	1
X_1	0	0	1
	1	1	0

X_2 \ X_1X_0	00	01	11	10
0	0	0	1	-
1	1	-	0	0

X_3X_2 \ X_1X_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	0	0	0	0

In una mappa di Karnaugh, celle **contigue** hanno coordinate **adiacenti**, e viceversa. Oltre le 4 coordinate, per le mappe non possiamo più rappresentare queste mappe senza la terza dimensione.

10.3.1 Ricerca dei sottocubi principali

Definiamo innanzitutto:

- **Sottocubo di ordine 1:** una casella che contiene un 1, corrispondente quindi ad uno stato di ingresso riconosciuto dalla rete, si indica come SO1;
- **Coordinate** di un SO1: stato di ingresso corrispondente al sottocubo;
- **Adiacenza** fra SO1: due SO1 sono adiacenti se differiscono fra loro di una sola coordinata.

Vediamo, ad esempio, una mappa di Karnaugh con $N = 2$, una serie di sottocubi di ordine 1 con la tabella associata:

X_1 \ X_0	0	1
0	0	1
1	1	0

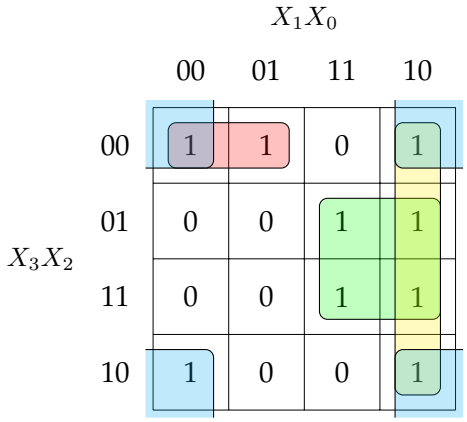
	X_1	X_0
A	0	1
B	1	0

Notiamo come A corrisponde all'implicante $\overline{X_1}X_0$, e B all'implicante $X_1\overline{X_0}$.
Possiamo continuare:

- **Sottocubo di ordine 2:** costituito da SO1 adiacenti, e si dice che **copre** i SO1 che lo formano. Si indica come SO2;
- **Sottocubo di ordine 4:** costituito da SO2 adiacenti, e si dice che **copre** i SO2 che lo formano. Si indica come SO4;

- **Sottocubo di ordine 8:** costituito da SO4 adiacenti, e si dice che **copre** i SO4 che lo formano. Si indica come SO8;

Vediamo un'ultimo esempio, con $N = 4$:



	X_3	X_2	X_1	X_0
A	0	0	0	-
B	-	1	1	-
C	-	-	1	0
D	-	0	-	0

Notiamo dall'esempio che le mappe di Karnaugh rispettano quello che potremmo chiamare *effetto pacman*: lo stesso implicante può esistere su lati opposti della mappa. Il bisogno di rappresentare le adiacenze dà origine a questa particolarità, come determina l'ordine particolare delle attivazioni degli ingressi. Inoltre, notiamo come i trattini nelle tabelle delle coordinate denotano che la variabile non influenza l'implicante, cioè rappresentano, in inglese, un *don't care*.

Si dice che un sottocubo è **principale** quando non esiste nessun sottocubo più grande che lo copre completamente. Si ha quindi che sottocubi e implicanti sono correlati: un sottocubo principale di ordine p rappresenta un implicante principale di $N - \log_2(p)$ variabili.

Si presenta quindi l'algoritmo per la ricerca dei sottocubi principali:

Algoritmo 3 per la ricerca dei sottocubi principali

Input: una mappa
Output: i sottocubi principali della mappa
ciclo:
Considera tutti i sottocubi di ordine p non interamente contenuti in sottocubi di ordine più grande, e segnali tutti: questi sono sicuramente principali
if l'insieme trovato finora basta a coprire tutta a mappa **then**
  L'algoritmo è terminato
else
  Poni $p \leftarrow \frac{p}{2}$ e vai a ciclo
end if

10.3.2 Ricerca delle liste di copertura non ridondanti

Una **lista di copertura** è l'insieme (qualunque) di sottocubi che coprono tutti gli SO1 della mappa. La lista dei sottocubi principali è una lista di copertura. Una **lista di copertura non ridondante** è una lista di copertura che smette di essere tale quando si toglie un sottocubo.

Come avevamo visto algebricamente, la lista degli SO1 (che corrisponderebbe ai mintermini) non è ridondante, ma la lista dei sottocubi principali (che corrisponderebbe agli implicanti) può esserlo.

Possiamo classificare i sottocubi principali come segue:

- Alcuni sottocubi sono gli unici a coprire un dato sottocubo di ordine 1. In questo caso, si chiamano sottocubi **essenziali**, e costituiscono il **cuore** (*core*) della mappa.
- Alcuni insiemi di sottocubi possono essere disposti, per cui rimuovere uno a caso fra i sottocubi compresi non risulta in variazioni della copertura. Questi si dicono **semplicemente eliminabili**.
- I sottocubi interamente contenuti all'interno di sottocubi essenziali sono sempre ridondanti, e si dicono **assolutamente eliminabili**.

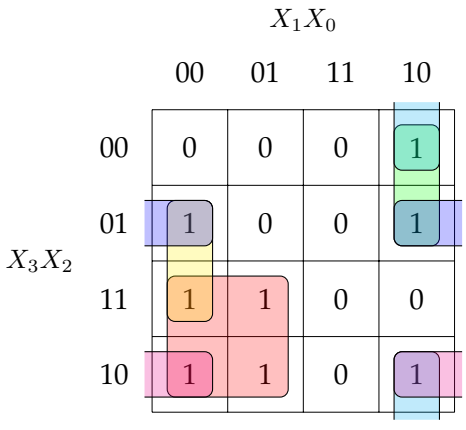
Rimuovendo i sottocubi assolutamente eliminabili, e mantenendo solo un sottoinsieme dei sottocubi semplicemente eliminabili, possiamo generare una qualsiasi lista non ridondante a partire dalla lista di copertura data. Vogliamo però selezionare la lista non ridondante che dà costo minimo.

10.3.3 Ricerca delle liste di copertura di costo minimo

Per trovare una **lista di copertura di costo minimo**, applichiamo un determinato criterio quando eliminiamo i sottocubi semplicemente eliminabili. In generale avremo che, per qualsiasi sottocubo semplicemente eliminabile, potremo considerarlo come:

- **Essenziale**, in questo caso resta nella lista di copertura. e qualche altro sottocubo diventa assolutamente eliminabile;
- **Assolutamente eliminabile**, in questo caso si toglie dalla lista di copertura, e qualche altro sottocubo diventa essenziale.

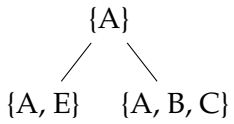
Il criterio adottato è quello di conservare il minor numero di sottocubi possibile. Prendiamo in esempio la mappa:



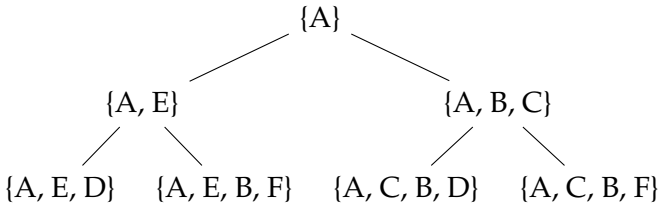
	X_3	X_2	X_1	X_0
A	1	-	0	-
B	0	-	1	0
C	-	1	0	0
D	-	0	1	0
E	0	1	-	0
F	1	0	-	0

Si ha che il sottocubo A è essenziale: altrimenti non potremmo mai coprire 1101 e 1001. Tutti gli altri cubi sono semplicemente eliminabili. Partiamo quindi dalla configurazione {A}, e decidiamo quali elementi includere successivamente. La prima zona che

osserviamo è quella di E: i sottocubi semplicemente eliminabili sono {A, B, C, E}. Eliminando C e B, resta E, e viceversa: chiaramente è meglio lasciare solo E. Possiamo rendere questo graficamente come:



La prossima zona di interesse è rappresentata da B, D, E e F. Si può continuare con la struttura ad albero:



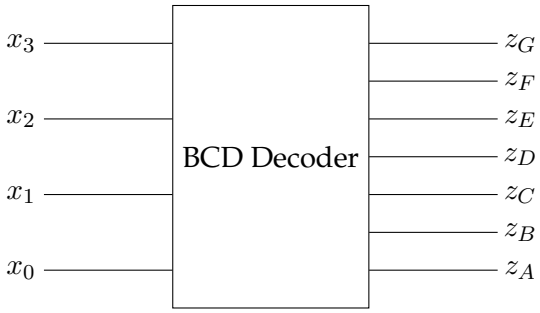
Scegliamo quindi {A, E, D}.

11 Lezione del 11-10-24

11.1 Sintesi di leggi non completamente specificate

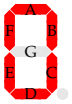
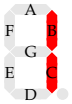
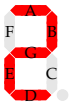
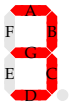
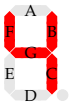
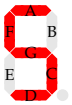
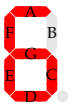
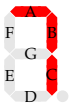
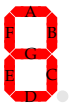
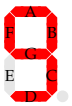
Applichiamo quanto abbiamo detto sulla sintesi di reti in forma SP a costo minimo, nel caso particolare in cui la legge non è completamente specificata (*don't care*).

Prendiamo in esempio un decodificatore BCD a 7 segmenti, simile a quello che si potrebbe trovare ad accendere le tracce di un display a cristalli liquidi.



Questo componente ha 4 variabili di ingresso, interpretate come una cifra in base 2 j, e 7 uscite, che specificano quali tracce accendere per visualizzare la cifra ottenuta in base 10. Si ha che con 4 variabili di ingresso si indirizzano 16 possibili configurazioni dei segmenti, quando ne abbiamo bisogno solo 10 (una per ogni cifra decimale). Le configurazioni di ingresso scartate si dicono quindi non specificate.

La tabella di verità della rete sarà quindi:

j	x_3	x_2	x_1	x_0	z_G	z_F	z_E	z_D	z_C	z_B	z_A	**Display**
0	0	0	0	0	0	1	1	1	1	1	1	
1	0	0	0	1	0	0	0	0	1	1	0	
2	0	0	1	0	1	0	1	1	0	1	1	
3	0	0	1	1	1	0	0	1	1	1	1	
4	0	1	0	0	1	1	0	0	1	1	0	
5	0	1	0	1	1	1	0	1	1	0	1	
6	0	1	1	0	1	1	1	1	1	0	1	
7	0	1	1	1	0	0	0	0	1	1	1	
8	1	0	0	0	1	1	1	1	1	1	1	
9	1	0	0	1	1	1	0	1	1	1	1	
10	1	0	1	0	–	–	–	–	–	–	–	
11	1	0	1	1	–	–	–	–	–	–	–	
12	1	1	0	0	–	–	–	–	–	–	–	
13	1	1	0	1	–	–	–	–	–	–	–	
14	1	1	1	0	–	–	–	–	–	–	–	
15	1	1	1	1	–	–	–	–	–	–	–	

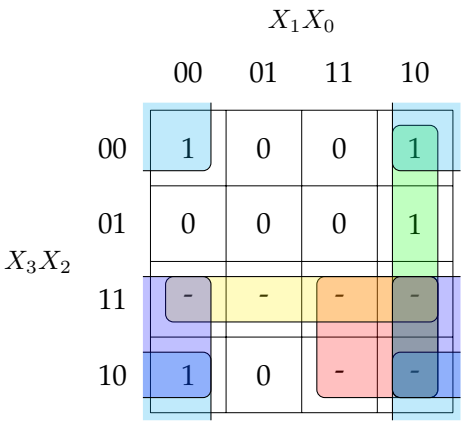
Visto che vogliamo sintetizzare reti su uscite singole, prendiamo la tabella di verità della rete sull'uscita z_E (le altre uscite richiederanno procedimenti simili):

j	x_3	x_2	x_1	x_0	z_E
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	1
3	0	0	1	1	0
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	1
7	0	1	1	1	0
8	1	0	0	0	1
9	1	0	0	1	0
10	1	0	1	0	–
11	1	0	1	1	–
12	1	1	0	0	–
13	1	1	0	1	–
14	1	1	1	0	–
15	1	1	1	1	–

Disegnamo quindi la mappa di Karnaugh. Quando si disegnano mappe di Karnaugh con elementi indeterminati, questi si interpretano diversamente a seconda che si stiano cercando i sottocubi principali, o che si stiano classificando:

- **Ricerca dei sottocubi principali:** si prendono come 1. Questo ci permette di prendere i sottocubi più grandi possibili nella ricerca dei primali (è irrilevante se si vanno a impostare uscite non specificate a 1).

  Si trova quindi:



	x_3	x_2	x_1	x_0
A	1	-	1	-
B	-	-	1	0
C	1	1	-	-
D	-	0	-	0
E	1	-	-	0

- **Classificazione dei sottocubi principali:** si prendono come 0. Così si evita di conservare implicanti che siano rilevanti su uscite non specificate (sarebbe inutile).

  Si ha quindi che i sottocubi A e C prendono solo SO1 indeterminati, ergo si scartano. Restano B e D essenziali, ed E a questo punto eliminabile, in quanto è già compreso in questi.

La sintesi completa dela rete è allora:

$$z_E = x_1\overline{x_0} + \overline{x_2x_0}$$

11.2 Sintesi in forma PS

Abbiamo usato finora la forma SP (somma di prodotti). Esiste la duale, ovvero la forma PS (prodotto di somme). Per trovare questa forma, esiste un metodo parallelo a quello studiato per la SP, dove si parte dal considerare i maxtermini invece che dei mintermini, cioè scegliendo sottocubi negli elementi che valgono 0 della mappa di Karnaugh.

Non considereremo questo metodo, ma un'alternativa più veloce:

Algoritmo 4 per la sintesi in forma PS

Input: una legge combinatoria F
Output: la sintesi in forma PS di F
Si ricava $\overline{F}$ complementando F
Si reallizza una sintesi SP della legge $\overline{F}$
Si ottiene una sintesi di F aggiungendo un invertitore in uscita alla rete SP che sintetizza $\overline{F}$
Si applicano i teoremi di de Morgan, da destra verso sinistra

Algebricamente, l'ultimo passaggio significa scrivere $\overline{F}$ in forma SP:

$$\overline{z} = P_1 + ... + P_k$$

dove P_i sono prodotti di variabili di ingresso, e applicare de Morgan come:

$$z = \overline{\overline{z}} = \overline{P_1 + ... + P_k} = \overline{P_1} \cdot ... \cdot \overline{P_k}$$

A questo punto si applica di nuovo de Morgan, come:

$$\overline{P_i} = \overline{\prod x_j} = \sum \overline{x_j}$$

11.2.1 Dualità fra forme SP e PS

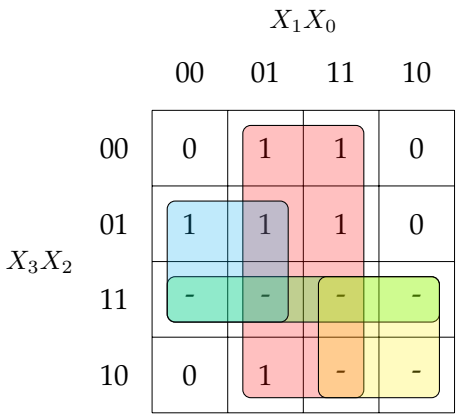
Con il procedimento presentato abbiamo che se $\overline{F}$ è in forma canonica SP, allora F è in forma canonica PS. Se la sintesi SP di $\overline{F}$ costa C, allora la sintesi PS di F costa C. Quindi se la sintesi SP di $\overline{F}$ è a costo minimo fra tutte le possibili sintesi SP, lo è anche la sintesi PS di F fra tutte le possibili sintesi PS. Se fosse il contrario, applicando de Morgan più volte avrei sintesi di costo sempre minore, violando la dualità.

A questo punto sappiamo effettuare la sintesi a costo minimo in forma SP di una qualsiasi legge F, e ponendo di sintetizzare prima $\overline{F}$ in forma SP, sappiamo anche trovare la sintesi a costo minimo in forma PS della stessa legge. Non possiamo determinare con sicurezza quale fra queste due sintesi ha costo minimo in generale, quindi bisogna controllare per forza la tabella della verità.

Troviamo ad esempio la sintesi in forma PS del BCD a 7 segmenti visto prima. Si ha che la negazine di F, su z_E, è:

j	x_3	x_2	x_1	x_0	z_E	$\overline{z_E}$
0	0	0	0	0	1	0
1	0	0	0	1	0	1
2	0	0	1	0	1	0
3	0	0	1	1	0	1
4	0	1	0	0	0	1
5	0	1	0	1	0	1
6	0	1	1	0	1	0
7	0	1	1	1	0	1
8	1	0	0	0	1	0
9	1	0	0	1	0	1
10	1	0	1	0	–	–
11	1	0	1	1	–	–
12	1	1	0	0	–	–
13	1	1	0	1	–	–
14	1	1	1	0	–	–
15	1	1	1	1	–	–

Ricaviamo quindi la mappa di Karnaugh:



	x_3	x_2	x_1	x_0
A	-	-	-	1
B	1	1	-	-
C	1	-	1	-
D	-	1	0	-

Si ha che B e C sono inutili, in quanto comprendono solo indeterminati. Restano allora A e D, entrambi essenziali, ergo la sintesi SP di $\overline{F}$ è:

$$\overline{F} = \overline{z_E} = x_0 + x_2\overline{x_1}$$

che neghiamo per ottenere nuovamente F:

$$F = z_E = \overline{x_0 + x_2\overline{x_1}}$$

A questo punto si può applicare de Morgan, prima sulla somma e poi sul prodotto a destra, per ottenere:

$$= \overline{x_0} \cdot \overline{x_2\overline{x1}} = \overline{x_0} \cdot (\overline{x_2} + x_1)$$

Cioè è la sintesi di z_E in forma PS, che notiamo essere meno costosa della sintesi in forma SP, di due porte logiche in meno.