

## 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. **Microprogrammazione:** 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:  $A \text{ NOR } B = y$  significa  $y = 1 \Leftrightarrow (A == B) = 0$ .

### 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

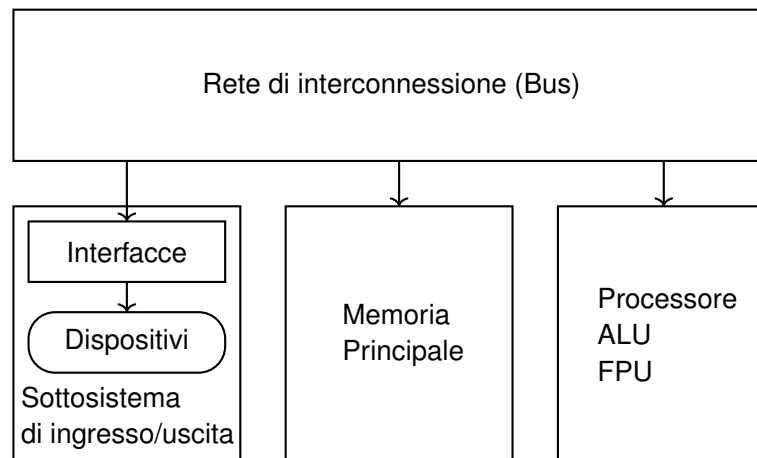
Si dice che assembler è un linguaggio a basso livello. Mancano i costrutti a cui siamo abituati da i linguaggi di alto livello:

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 quasi 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

I calcolatori lavorano con i numeri, e un numero si **rappresenta** su una certa base  $\beta$ . Nel caso dei calcolatori questa è la base 2. Da qui in poi  $x$  è il numero rappresentato e  $X$  la sequenza di bit rappresentante.

#### 1.4.1 Numeri naturali

Vediamo prima di tutto come rappresentare i 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  $b_{n-1}, b_{n-2}, \dots, b_1, b_0$  della rappresentazione  $X$ , essi rappresentano il naturale  $x$ :

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

Il bit più a sinistra è il Most Significant Bit (MSB), cioè  $b_{n-1}$ , 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. Questo generalizza a qualsiasi base  $\beta$  (per noi  $\beta = 2$ ) come:

**Algoritmo 1 DIV-MOD**

**Input:**  $x$  in base arbitraria

**Output:**  $X$  rappresentazione in base  $\beta$

Inizializza  $q \leftarrow x$ ,  $r \leftarrow 0$ , e  $i \leftarrow 0$

Crea un'array vuota  $R$  per i resti

**while**  $q \neq 0$  **do**

$r \leftarrow q \bmod \beta$

    Metti  $r$  in  $R[i]$

$q \leftarrow q/\beta$

$i \leftarrow i + 1$

**end while**

Gli  $R[n-1], R[n-2], \dots, R[0]$  rimasti (quindi gli  $R[i]$  letti al contrario) sono le cifre di  $X$ .

Notiamo come l'algoritmo permette di svolgere i calcoli *nella base di partenza*, a differenza della trasformazione diretta, che si svolge *nella base di arrivo*.

**1.4.2 Numeri interi in complemento a due**

Per rappresentare i numeri interi si usa, fra le altre, la rappresentazione in **complemento a due**. Tralasciamo le basi  $\beta$  arbitrarie (dove sarebbe *complemento a radice*), e soffermiamoci sulla base due: dettagli sulle varie altre rappresentazioni sia per i naturali che per gli interi, in basi qualsiasi, verranno riportati in un secondo momento.

**Intervallo di rappresentabilità**

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

**Trasformazione diretta**

Per portare un intero  $x$  in una base qualsiasi 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  $2^n - |x|$ , cioè bisogna ricordare che  $x$  è negativo.

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  $b_{n-1}, b_{n-2}, \dots, b_1, b_0$  della rappresentazione  $X$  con segno negativo:

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

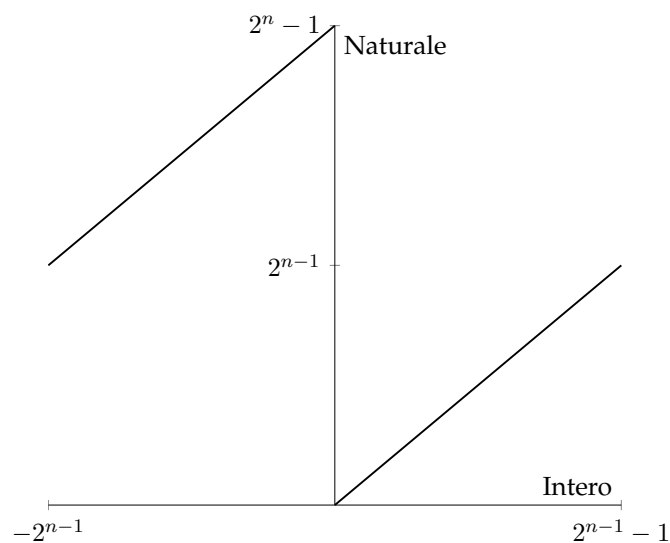
Alternativamente, si nota che il bit più significativo della rappresentazione sarà impostato a 0 per numeri positivi e 1 per numeri negativi. Ciò significa che avremo:

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

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  $n$  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 così 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 utilizzato 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 byte 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 `0x00000103` 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 alterare 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 sorgente.
- 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:

```
1 MOVB pippo, pluto
```

ma devo scrivere:

```
1 MOV pippo, %EAX # qua il suffisso di lunghezza e' implicito
2 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 registro.

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 verranno 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	<b>MOV</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>MOV</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>MOV</b> %AX, %DX
Immediato, Memoria	<b>MOVB</b> \$0x5B, (%EDI)
Immediato, Registro generale	<b>MOV</b> \$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	<b>LEA</b> 0x00002000, %EDX
	<b>LEA</b> 0x00213AB1 (%EAX,%EBX,4), %ECX

A differenza di MOV, LEA calcola l'indirizzo della locazione di memoria cercata come  $\text{base} + \text{index} \times \text{scala} \pm \text{displacement}$ , 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	<b>XCHG</b> 0x00002000, \%DX
Registro Generale, Memoria	<b>XCHG</b> \%AL, 0x000A2003
Registro Generale, Registro Generale	<b>XCHG</b> \%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), \%A1 (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)
  - **OUT** \%AX, indirizzo (16 bit)
  - **OUT** \%AX, (%DX) (8 bit)
  - **OUT** \%A1, (\%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 (LI-FO), 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	<b>PUSHW</b> 0x3214200A
Immediato	<b>PUSHL</b> \ \$0x4871A000
Registro Generale	<b>PUSH</b> %BX

### 3.2.2 POP

- **Formato:** **POP** destination
- **Azione:** copia una word o un long dall'indirizzo 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	<b>POPW</b> 0x02AB2000
Registro Generale	<b>POP</b> \%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:

```

1 # sto usando %EAX, mi serve un dato da una porta
2 PUSH %EAX
3 IN 0x001A, %AL
4 ...
5 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	<code>ADD 0x00002000, %EDX</code>
Registro Generale, Memoria	<code>ADD %CL, 0x12AB1024</code>
Registro Generale, Registro Generale	<code>ADD %AX, %DX</code>
Immediato, Memoria	<code>ADD \$0x5B, (%EDI)</code>
Immediato, Registro Generale	<code>ADD \$0x54A3, %AX</code>

#### 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 prossima 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:

$x_i$	$y_i$	$r_i$	$(x_i + y_i + r_i)_{10}$	$s_i$	$r_{i+1}$
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 valutata 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'è 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 due 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	<b>INCB</b> ( $\%ESI$ )
Registro Generale	<b>INC</b> $\%CX$

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 traboccamento.

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

### 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 prossima 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	<b>DEC</b> (%EDI)
Registro Generale	<b>DEC</b> %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	<b>ADC</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>ADC</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>ADC</b> %AX, %DX
Immediato, Memoria	<b>ADCB</b> \$0x5B, (%EDI)
Immediato, Registro Generale	<b>ADC</b> \$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	<b>SBB</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>SBB</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>SBB</b> %AX, %DX
Immediato, Memoria	<b>SBBB</b> \$0x255B, (%EDI)
Immediato, Registro Generale	<b>SBB</b> \$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	<b>NEGB</b> (%EDI)
Registro Generale	<b>NEG</b> %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	<b>CMP</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>CMP</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>CMP</b> %AX, %DX
Immediato, Memoria	<b>CMPB</b> \$0x255B, (%EDI)
Immediato, Registro Generale	<b>CMP</b> \$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, si differenziano 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	<code>MULB (%ESI)</code>
Registro Generale	<code>MUL %ECX</code>

#### 3.4.2 INTEGER MULTIPLY

- **Formato:** `IMUL 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	<code>IMULB (%ESI)</code>
Registro Generale	<code>IMUL %ECX</code>

#### 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  $AX = AL \times \text{source}$ ;
- Sorgente a 16 bit, si ha  $DX\_AX = AX \times \text{source}$ ;
- Sorgente a 32 bit, si ha  $EDX\_EAX = EAX \times \text{source}$ .

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:

```
1 PUSH \%DX
2 PUSH \%AX
3 POP  \%EAX
```

## 4 Lezione del 27-09-24

### 4.1 Divisioni

La divisione 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:

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

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  $AL = AX \div \text{source}$ , e  $AH = AX \bmod \text{source}$ ;
  - Sorgente a 16 bit, si ha  $AX = DX\_AX \div \text{source}$ , e  $DX = DX\_AX \bmod \text{source}$ ;
  - Sorgente a 32 bit, si ha  $EAX = EDX\_EAX \div \text{source}$ , e  $EDX = EDX\_EAX \bmod \text{source}$ ;

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 sotto-programma. Da lì in poi i risultati generati non sono più attendibili

- **Flag:** imposta tutti i bit, ma non è attendibile.

Operandi	Esempi
Memoria	<b>DIVB</b> (%ESI) \# AX destinazione implicita
Registro Generale	<b>DIV</b> \%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:** `IMUL 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  $AH = AX \bmod source$ ;
  - Sorgente a 16 bit, si ha  $AX = DX\_AX/source$ , e  $DX = DX\_AX \bmod source$ ;
  - Sorgente a 32 bit, si ha  $EAX = EDX\_EAX/source$ , e  $EDX = EDX\_EAX \bmod source$ ;
- **Flag:** li imposta tutti, ma non è attendibile.

Operandi	Esempi
Memoria	<code>IDIVB (%ESI) \# AX destinazione implicita</code>
Registro Generale	<code>IDIV %ECX \# EDX\_EAX destinazione implicita</code>

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,  $-7 \text{ idiv } 3 = -2, -1$  e  $7 \text{ idiv } -3 = -2, +1$ .

#### 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  $AH = AX \bmod source$ ;
- Sorgente a 16 bit, si ha  $AX = DX\_AX/source$ , e  $DX = DX\_AX \bmod source$ ;
- Sorgente a 32 bit, si ha  $EAX = EDX\_EAX/source$ , e  $EDX = EDX\_EAX \bmod source$ ;

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 così raro avere divisioni dove il quoziente non sta nella metà dei bit del dividendo, ad esempio:

```

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

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

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

## 4.2 Note conclusive su moltiplicazioni e divisioni

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

- Scegliere con cura la versione che usiamo (soprattutto 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:

```
1 MOV $-5, %AX
2 MOV $100000, %EBX
3 CWDE
4 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  $\leq 31$  (sarebbe inutile fare  $\geq 32$  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	<b>SHL</b> \$1, %EAX
Immediato, Memoria	<b>SHLB</b> \$7, 0x00002000
Registro CL, Registro Generale	<b>SHL</b> %CL, %EAX
Registro CL, Memoria	<b>SHLL</b> %CL, (%EDI)
Memoria	<b>SHLL</b> (%EDI)
Registro Generale	<b>SHL</b> %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  $2^{\text{source}}$ .
- **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	<b>SHR</b> \$1, %EAX
Immediato, Memoria	<b>SHRB</b> \$7, 0x00002000
Registro CL, Registro Generale	<b>SHR</b> %CL, %EAX
Registro CL, Memoria	<b>SHRL</b> %CL, (%EDI)
Memoria	<b>SHRL</b> (%EDI)
Registro Generale	<b>SHR</b> %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  $2^{\text{source}}$ .
- **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	<b>ROL</b> \$1, %EAX
Immediato, Memoria	<b>ROLB</b> \$7, 0x00002000
Registro CL, Registro Generale	<b>ROL</b> %CL, %EAX
Registro CL, Memoria	<b>ROLL</b> %CL, (%EDI)
Memoria	<b>ROLL</b> (%EDI)
Registro Generale	<b>ROL</b> %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	<b>ROR</b> \$1, %EAX
Immediato, Memoria	<b>RORB</b> \$7, 0x00002000
Registro CL, Registro Generale	<b>ROR</b> %CL, %EAX
Registro CL, Memoria	<b>RORL</b> %CL, (%EDI)
Memoria	<b>RORL</b> (%EDI)
Registro Generale	<b>ROR</b> %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	<b>RCL</b> \$1, %EAX
Immediato, Memoria	<b>RCLB</b> \$7, 0x00002000
Registro CL, Registro Generale	<b>RCL</b> %CL, %EAX
Registro CL, Memoria	<b>RCLL</b> %CL, (%EDI)
Memoria	<b>RCLL</b> (%EDI)
Registro Generale	<b>RCL</b> %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	<b>RCR</b> \$1, %EAX
Immediato, Memoria	<b>RCRB</b> \$7, 0x00002000
Registro CL, Registro Generale	<b>RCR</b> %CL, %EAX
Registro CL, Memoria	<b>RCRL</b> %CL, (%EDI)
Memoria	<b>RCRL</b> (%EDI)
Registro Generale	<b>RCR</b> %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	<b>NOTL</b> (%ESI)
Registro Generale	<b>NOT</b> %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	<b>AND</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>AND</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>AND</b> %AX, %DX
Immediato, Memoria	<b>AND</b> 5x5B, (%EDI)
Immediato, Registro Generale	<b>AND</b> \$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	<b>OR</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>OR</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>OR</b> %AX, %DX
Immediato, Memoria	<b>OR</b> 5x5B, (%EDI)
Immediato, Registro Generale	<b>OR</b> \$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	<b>XOR</b> 0x00002000, %EDX
Registro Generale, Memoria	<b>XOR</b> %CL, 0x12AB1024
Registro Generale, Registro Generale	<b>XOR</b> %AX, %DX
Immediato, Memoria	<b>XOR</b> 5x5B, (%EDI)
Immediato, Registro Generale	<b>XOR</b> \$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:

```

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

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

```

1 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:

```

1 MOV $5, %AL
2 MOV $100000, %EBX
3 AND $0x000000FF, %EAX
4 ADD %EAX, %EBX

```

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

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

- **XOR:**

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

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

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

```
1 XOR %EAX, %EAX # equivale a dire MOV $0, %EAX, ma occupa
2                # 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 sottoprogrammi:** CALL, RET.

### 4.7.1 JUMP

- **Formato:** JMP `\%EIP +/- displacement`, JMP `*extended\_register`, JMP `*memory`
- **Azione:** calcola un'indirizzo 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

```
1 ADD %AX, %BX
2 JC ...
3 # continua
```

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

```
1 ADD %AX, %BX
2 JO ...
3 # continua
```

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

```
1 SUB %AL, %BL
2 JS ...
3 # 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 0, 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.

```

1 CMP %AX, %BX
2 JAE ...
3 # continua

```

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

```

1 CMP %EDX, %ECX
2 JB ...
3 # 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 e 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.
JL	Jump if Less, la condizione è soddisfatta se SF è diverso da OF, cioè dopo CMP su un destinatario minore del sorgente.
JLE	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

```
1 CMP %AX, %BX
2 JGE ...
3 # continua
```

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

```
1 CMP %EDX, %ECX
2 JL ...
3 # 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 coinvolte 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	<b>CALL</b> 0x00400010
Registro	<b>CALL</b> *%EAX
Memoria	<b>CALL</b> *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 HCF

- **Formato:** **HCF**
- **Azione:** arresta l'esecuzione e causa l'autocombustione spontanea 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:

```

1 .GLOBAL _main
2
3 .DATA
4 ...
5
6 .TEXT
7 _main:  NOP
8 ...
9          RET

```

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

```
1 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:

```

1 .GLOBAL _main
2 .DATA
3 dato:      .LONG 0xF0F0101
4 conteggio: .BYTE 0x00
5
6 .TEXT
7 _main:     NOP
8           MOVB $0x00, %CL
9           MOVL dato, %EAX
10 comp:     CMPL $0x00, %EAX
11           JE fine
12           SHRL %EAX
13           ADCB $0x00, %CL
14           JMP comp
15 fine:     MOVB %CL, conteggio
16           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

```

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

```

```

5 var2: .BYTE 0x30,0x31      # vettore, 2 componenti da 1 byte,
6                             # valore 0x30 e 0x31
7 var3: .WORD 0x1020, 0x32AB  # vettore, 2 componenti da 2 byte,
8                             # valore 0x1020e 0x32AB
9 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 espressione (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:

```

1 var5: .BYTE 'S', 'o', 'n', 'n', 'o'      # vettore, 4 componenti
2                                           # da 1 byte
3 var6: .BYTE 0x53, 0x6F, 0x6E, 0x6E, 0x6F  # vettore, 4 componenti
4                                           # da 1 byte
5 var7: .ASCII "Stea"                      # vettore, 4 componenti
6                                           # da 1 byte
7 var8: .ASCIZ "Stea"                      # vettore, 5 componenti
8                                           # da 1 byte (include il
9                                           # terminatore)
10

```

#### • 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:

```

1 .SET dimensione, 4
2 .SET n_iter, (100 * dimensione)
3 ...
4 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 poi useremo una sintassi pseudo-C per indicare questi costrutti ad alto livello.

### 5.5.1 If-then-else

Prendiamo la sintassi:

```

1 if(%AX < variabile) {
2     //ramo if
3     ...
4 } else {
5     //ramo else
6     ...
7 }
8 //proseguì
9 ...

```

potremo tradurla in due modi:

- Invertendo i rami then e else:

```

1         CMP variabile, %AX
2         JB ramothen
3 ramoelse: ... # ramo else
4         JMP segue
5 ramothen: ... # ramo then
6 segue:   # proseguì
7

```

- Invertendo la condizione:

```

1         CMP variabile, %AX
2         JAE ramoelse
3 ramothen: ... # ramo then
4         JMP segue
5 ramoelse: ... # ramo else
6 segue:   ... # proseguì

```

### 5.5.2 Ciclo for

Prendiamo:

```

1 for(int i = 0; i < variabile; i++) {
2     //iter
3     ...
4 }
5 //proseguì
6 ...

```

si rende attraverso il registro CX, come:

```

1         MOV $0, %CX
2 ciclo:   CMP var, %CX
3         JE segue
4         ... # iter
5         INC %CX
6         JMP ciclo
7 segue:   ... # proseguì

```

### 5.5.3 Ciclo do-while

Prendiamo infine:

```

1 do {
2     //iter
3     ...
4 } while (AX < var)
5 //proseguì
6 ...

```

si rende come:

```

1 ciclo: ... # iter
2         CMP var, %AX
3         JB ciclo
4         ... # proseguì

```

#### 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:

```

1 ciclo: ... # inizio ciclo
2         ...
3 label1: ... # meta' ciclo
4         CMP var, %AX
5         JB ciclo
6         ...
7         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:

```

1 for(int i = var; i > 0; i--) {
2     //iter (usa i)
3 }

```

diventa:

```

1         MOV var, %ECX
2 ciclo: ... # iter
3         LOOP ciclo

```

```

1 for(int i = 0; i < var; i++) {
2     //iter (usa i)
3 }

```

diventa:

```

1         MOV $0, %EBX # usa EBX
2 ciclo: ... # iter
3         INC EBX
4         CMP var, %EBX
5         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:

```
1      MOV $10, %ECX
2 ciclo:  CMP src, dest
3          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 forniscono 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:

```
1 # sottoprogramma "sottoprogram", [descrizione]
2 # ingresso: %AX, [descrizione]
3 #           %EBX, [descrizione]
4 # uscita:   CAX, [descrizione]
5
6 sottoprogram: ...
7               MOV ..., %CX # preparo il ritorno
8               RET
```

adesso potremo usare il sottoprogramma come:

```
1 MOV ..., %AX # preparo i parametri
2 MOV ..., %EBX
3 CALL sottoprogram # chiamo
4 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:

```
1 sottoprogram: PUSH ... # fai push dei registri
2               PUSH ...
3               ... # esegui il sottoprogramma
4               MOV ..., %CX
5
```

```

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 implicitamente;
- 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 degli argomenti (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 codice 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 sul 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:

```
1 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 ( $\text{displacement} + \text{base} + \text{indice} \times \text{scala}$ ).

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:

```

1 vett_sorg:  .FILL 1000,4
2 vett_dest:  .FILL 1000,4
3
4             MOV $1000, %ECX
5             LEA vett_sorg, %ESI
6             LEA vett_dest, %EDI
7 ciclo:     MOV (%ESI), %EAX
8             MOV %EAX, (%EDI)
9             ADD $4, %ESI
10            ADD $4, %EDI
11            LOOP ciclo

```

ma abbiamo la possibilità di scrivere la stessa cosa come:

```

1 vett_sorg:  .FILL 1000,4
2 vett_dest:  .FILL 1000,4
3
4             MOV $1000, %ECX
5             LEA vett_sorg, %ESI
6             LEA vett_dest, %EDI
7             REP MOVSL

```

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

### 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**: `STOSsuf`
- **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:

```

1      MOV $1000, %CX
2      LEA buffer_src, %ESI
3      LEA buffer_dst, %EDI
4      CLD
5 ciclo: LODSL
6      ... #modifica %EAX
7      STOSL
8      LOOP ciclo

```

Riempi un buffer in memoria di zeri:

```

1 MOV $1000, %ECX
2 LEA buffer, %EDI
3 XOR %EAX, %EAX
4 CLD
5 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,  $DF = 0$  che cerca la prima occorrenza, e  $DF = 1$  che cerca l'ultima. Ad esempio, poniamo di voler trovare il primo elemento differente fra due vettori:

```
1 array1: .WORD 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
2 array2: .WORD 1, 2, 3, 4, 7, 6, 7, 8, 9, 10
3
4 CLD
5 LEA array1, %ESI
6 LEA array2, %EDI
7 MOV $10, %ECX
8 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**, **LDS**, **STOS**, **INS** e **OUTS**, anche se l'utilizzo con **LDS** 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 Perché due direzioni?

L'uso di due direzioni di scorrimento di stringhe attraverso il flag **DF** è utile, soprattutto nel caso si debbano fare traslazioni del vettore (copia di buffer **parzialmente sovrapposti**). Infatti, cercando di 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 assembler, 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 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 da 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 sbaglia, 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 tabelle 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 generali 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 48 (o 57) 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
<b>Regione alta</b>	0000 0000 0000 0000 0000 7fff ffff ffff	0000 0000 0000 0000 01ff ffff ffff ffff
<b>Regione bassa</b>	ffff 8000 0000 0000 ffff ffff ffff ffff	fe00 0000 0000 0000 ffff ffff ffff ffff

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

### 7.1.2 Istruzioni

Le operazioni 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 assembler 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  $X$ .  $X = \{x_{N-1}x_{N-2}\dots x_1x_0\}$ .
- 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  $Z$ .  $Z = \{x_{M-1}x_{M-2}\dots x_1x_0\}$ .
- 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, programmatore, 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  $\Delta T$ , 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  $\Delta t \ll \Delta T$ :



Decidiamo di ignorare questo problema, in quanto abbiamo visto che il  $\Delta t$  di transizione è molto più piccolo del  $\Delta t$  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 Il linguaggio Verilog

Per descrivere le reti logiche fa comodo adottare una **notazione testuale**. Per reti semplici useremo disegni o espressioni algebriche: per reti complesse introduciamo un **linguaggio di descrizione hardware**, il **Verilog**. Questo linguaggio è più **compatto**, e può essere **interpretato** automaticamente da una macchina, permettendoci di effettuare prove (e realizzare **diagrammi di temporizzazione**).

Non si riporteranno appunti riguardanti operatori e sintassi particolarmente specifiche del Verilog, in quanto esistono testi sicuramente più utili e approfonditi: procederemo principalmente per esempi, esplicitando quando si rende necessario particolarità del linguaggio.

### 7.4.1 Struttura di una sintesi Verilog

Il linguaggio Verilog descrive **moduli**. Un modulo è formato da un insieme di **input** e **output**, e da una **struttura interna** che descrive la legge di evoluzione degli output in funzione degli input. Ad esempio, una rete basilare potrebbe essere:

```
1 module rete(x, z);
2   input x;
3   output z;
4   wire y;
5   assign y = x;
6   assign z = y;
7 endmodule
```

Nell'esempio, si definisce un modulo *rete*, formato da un input  $x$  e un output  $z$ . La realizzazione interna della rete è formata da un filo  $y$  a cui sono connessi sia l'input che l'output. Il funzionamento della rete è quindi semplicemente quello di replicare il suo ingresso.

In particolare, diciamo che la parola chiave `assign` rappresenta un **assegnamento continuo**. Più avanti vedremo i diversi tipi di assegnamento e le differenze fra di loro.

## 7.5 Reti combinatorie

Il primo tipo di reti logiche che andiamo a studiare sono le **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  $F : X \rightarrow Z$  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.

### 7.5.1 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  $F$ , cioè 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.

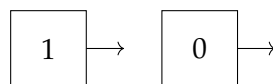
### 8.1.1 Descrizione e sintesi

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.

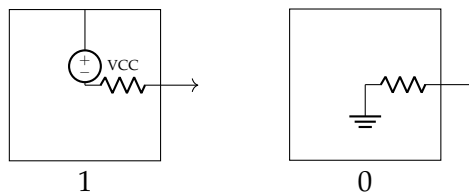
Notiamo una proprietà fondamentale: ogni rete combinatoria di  $N$  ingressi e  $M$  uscite può essere realizzata interconnettendo  $M$  reti combinatorie ad  $N$  ingressi ed una uscita. Questo ci permette di trattare tutte le reti con reti con una sola uscita.

## 8.2 Reti a 0 ingressi

Le reti a 0 ingresso di uscita si chiamano **generatori di costante**, e rappresentano un caso degenero. 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:



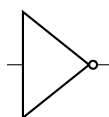
## 8.3 Reti a 1 ingresso

### 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:



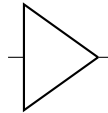
Essenzialmente nega il suo ingresso.

### 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

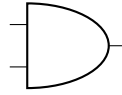
La prima domanda da porsi quando si parla di reti a 2 (come  $N$ ) ingressi, è quante reti possiamo creare in tutto. Su  $N$  ingressi, la tabella di verità avrà  $2^N$  righe. Le configurazioni possibili di 0 e 1 su  $2^N$  righe sono  $2^{2^N}$ . Ergo, nel caso  $N = 2$ , abbiamo  $2^{2^2} = 16$  possibili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

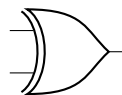


e ha tabella di verità:

$x_1$	$x_0$	$z$
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:

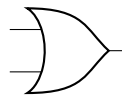


e ha tabella di verità:

$x_1$	$x_0$	$z$
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:

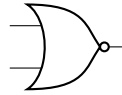


e ha tabella di verità:

$x_1$	$x_0$	$z$
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:

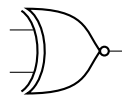


e ha tabella di verità:

$x_1$	$x_0$	$z$
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:

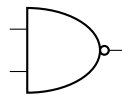


e ha tabella di verità:

$x_1$	$x_0$	$z$
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

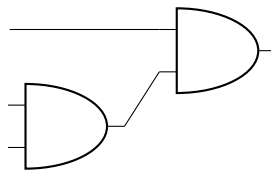
Alcuni casi speciali della tabella delle possibili reti a due porte sono degeneri: abbiamo due generatori di costante ( $z^0$  e  $z^{15}$ ), due elementi neutri, rispettivamente su  $x_1$  e  $x_0$  ( $z^3$  e  $z_5$ ), e due inversori sugli stessi ingressi ( $z_{10}$  e  $z_{12}$ ).

### 8.5 AND e OR a più ingressi

Posso pensare di estendere AND e OR ad  $N$  ingressi:

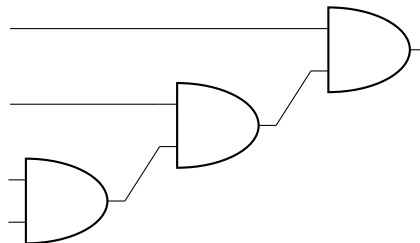
- **AND a  $N$  ingressi:** l'uscita vale 1 se tutti gli  $N$  ingressi valgono 1;
- **OR a  $N$  ingressi:** l'uscita vale 1 se almeno un'ingresso vale 1;

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

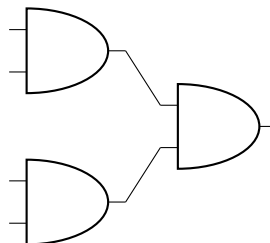


La dimostrazione è semplice dalla tabella di verità, o dalle proprietà degli operatori logici.

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



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  $\geq 2$  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  $\geq 2$  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  $\bar{x}$ , oppure  $!x$  o  $/x$ . Si definisce come:

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

- **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:**  $\bar{\bar{x}} = x$ ;
2. **Commutativa della somma e del prodotto:**  $x + y = y + x$ ,  $x \cdot y = y \cdot x$ ;
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:**  $x \cdot \bar{x} = 0, \quad x + \bar{x} = 1$ ;
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:**  $\overline{x \cdot y} = \bar{x} + \bar{y}$  e  $\overline{x + y} = \bar{x} \cdot \bar{y}$ .

### 8.6.1 Teoremi di De Morgan

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

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

#### 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:

$x$	$y$	$x \cdot y$	$\overline{x \cdot y}$	$\bar{x}$	$\bar{y}$	$\bar{x} + \bar{y}$
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:

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

e la tesi:

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

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

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

dove notiamo la variabile introdotta  $\alpha$ , se complementata, rispetta:

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

dall'ipotesi.

Possiamo quindi svolgere il passaggio:

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

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:

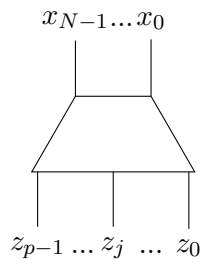
- **Data una rete combinatoria**, (comunque complessa), è sempre possibile trovare un'espressione booleana che mette in relazione ogni sua uscita con gli ingressi (in verità un'espressione per ogni uscita);
- **Data un'espressione booleana**, è sempre possibile sintetizzare una rete combinatoria (ad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

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

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

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).

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

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 uscita avrà 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:

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

Vediamo quindi le codifiche in Verilog di decoder a diversi valori di  $N$ . Si definisce innanzitutto il caso banale di  $N = 1$ , che finora non è stato trattato. Questo servirà a definire, in maniera gerarchica (ma come vedremo imperfetta), decoder più complessi:

```

1 // un decoder da 1 a 2 che prende @x0 come ingresso e attiva l'uscita
2 // @z1_z0 in codifica one-hot
3 module b1to2_decoder(x0, z1_z0);
4     input x0;
5     output [1:0] z1_z0;
6
7     assign z1_z0 = (x0 == 'B0) ? 'B01 :
8                   /*(x0 == 'B1)?*/'B10;
9 endmodule
10
11 // implementazione a porte logiche
12 module b1to2_decoder_p(x0, z1_z0);
13     input x0;
14     output [1:0] z1_z0;
15
16     assign z1_z0[0] = ~x0;
17     assign z1_z0[1] = x0;
18 endmodule

```

Possiamo quindi definire il decoder da 2 a 4 visto prima:

```

1 // un decoder da 2 a 4 che prende @x1_x0 come ingresso e attiva
2 // l'uscita @z4_z0 in codifica one-hot
3 module b2to4_decoder(x1_x0, z3_z0);
4     input [1:0] x1_x0;
5     output [3:0] z3_z0;
6
7     assign z3_z0 = (x1_x0 == 'B00) ? 'B0001:
8                   (x1_x0 == 'B01) ? 'B0010:
9                   (x1_x0 == 'B10) ? 'B0100:
10                  /*(x1_x0 == 'B11)?*/'B1000;
11 endmodule
12
13 // implementazione a porte logiche
14 module b2to4_decoder_p(x1_x0, z3_z0);
15     input [1:0] x1_x0;
16     output [3:0] z3_z0;
17
18     assign z3_z0[0] = ~x1_x0[1] & ~x1_x0[0];
19     assign z3_z0[1] = ~x1_x0[1] & x1_x0[0];
20     assign z3_z0[2] = x1_x0[1] & ~x1_x0[0];
21     assign z3_z0[3] = x1_x0[1] & x1_x0[0];
22 endmodule
23
24 // implementazione gerarchica
25 module b2to4_decoder_g(x1_x0, z3_z0);
26     input [1:0] x1_x0;
27     output [3:0] z3_z0;
28
29     wire [1:0] x1_d;
30     wire [1:0] x0_d;
31
32     b1to2_decoder b1to2_1 (x1_x0[1], x1_d);
33     b1to2_decoder b1to2_2 (x1_x0[0], x0_d);
34
35     assign z3_z0[0] = x1_d[0] & x0_d[0];
36     assign z3_z0[1] = x1_d[0] & x0_d[1];
37     assign z3_z0[2] = x1_d[1] & x0_d[0];
38     assign z3_z0[3] = x1_d[1] & x0_d[1];
39 endmodule

```

un decoder da 3 a 8:

```

1 // un decoder da 3 a 8 che prende @x2_x0 come ingresso e attiva
2 // l'uscita @z7_z0 in codifica one-hot
3 module b3to8_decoder(x2_x0, z7_z0);
4     input [2:0] x2_x0;
5     output [7:0] z7_z0;
6
7     assign z7_z0 = (x2_x0 == 'B000) ? 'B0000_0001:
8                   (x2_x0 == 'B001) ? 'B0000_0010:
9                   (x2_x0 == 'B010) ? 'B0000_0100:
10                  (x2_x0 == 'B011) ? 'B0000_1000:
11                  (x2_x0 == 'B100) ? 'B0001_0000:
12                  (x2_x0 == 'B101) ? 'B0010_0000:
13                  (x2_x0 == 'B110) ? 'B0100_0000:
14                  /*(x2_x0 == 'B111)?*/'B1000_0000;
15 endmodule
16
17 // implementazione a porte logiche
18 module b3to8_decoder_p(x2_x0, z7_z0);

```

```

19  input [2:0]  x2_x0;
20  output [7:0] z7_z0;
21
22  assign z7_z0[0] = ~x2_x0[2] & ~x2_x0[1] & ~x2_x0[0];
23  assign z7_z0[1] = ~x2_x0[2] & ~x2_x0[1] & x2_x0[0];
24  assign z7_z0[2] = ~x2_x0[2] & x2_x0[1] & ~x2_x0[0];
25  assign z7_z0[3] = ~x2_x0[2] & x2_x0[1] & x2_x0[0];
26  assign z7_z0[4] = x2_x0[2] & ~x2_x0[1] & ~x2_x0[0];
27  assign z7_z0[5] = x2_x0[2] & ~x2_x0[1] & x2_x0[0];
28  assign z7_z0[6] = x2_x0[2] & x2_x0[1] & ~x2_x0[0];
29  assign z7_z0[7] = x2_x0[2] & x2_x0[1] & x2_x0[0];
30  endmodule
31
32  // implementazione gerarchica
33  module b3to8_decoder_g(x2_x0, z7_z0);
34      input [2:0]  x2_x0;
35      output [7:0] z7_z0;
36
37      wire [1:0]  x2_d;
38      wire [3:0]  x1_x0_d;
39
40      b1to2_decoder b1to2 (x2_x0[2], x2_d);
41      b2to4_decoder b2to4 (x2_x0[1:0], x1_x0_d);
42
43      assign z7_z0[0] = x2_d[0] & x1_x0_d[0];
44      assign z7_z0[1] = x2_d[0] & x1_x0_d[1];
45      assign z7_z0[2] = x2_d[0] & x1_x0_d[2];
46      assign z7_z0[3] = x2_d[0] & x1_x0_d[3];
47      assign z7_z0[4] = x2_d[1] & x1_x0_d[0];
48      assign z7_z0[5] = x2_d[1] & x1_x0_d[1];
49      assign z7_z0[6] = x2_d[1] & x1_x0_d[2];
50      assign z7_z0[7] = x2_d[1] & x1_x0_d[3];
51  endmodule

```

e infine, ad evidenziare quanto velocemente esplode il numero di termini (cioè esponenzialmente), un decoder da 4 a 16:

```

1  // un decoder da 4 a 16 che prende @x3_x0 come ingresso e attiva
2  // l'uscita @z15_z0 in codifica one-hot
3  module b4to16_decoder(x3_x0, z15_z0);
4      input [3:0]  x3_x0;
5      output [15:0] z15_z0;
6
7      assign z15_z0 = (x3_x0 == 'B0000) ? 'B0000_0000_0000_0001:
8                      (x3_x0 == 'B0001) ? 'B0000_0000_0000_0010:
9                      (x3_x0 == 'B0010) ? 'B0000_0000_0000_0100:
10                     (x3_x0 == 'B0011) ? 'B0000_0000_0000_1000:
11                     (x3_x0 == 'B0100) ? 'B0000_0000_0001_0000:
12                     (x3_x0 == 'B0101) ? 'B0000_0000_0010_0000:
13                     (x3_x0 == 'B0110) ? 'B0000_0000_0100_0000:
14                     (x3_x0 == 'B0111) ? 'B0000_0000_1000_0000:
15                     (x3_x0 == 'B1000) ? 'B0000_0001_0000_0000:
16                     (x3_x0 == 'B1001) ? 'B0000_0010_0000_0000:
17                     (x3_x0 == 'B1010) ? 'B0000_0100_0000_0000:
18                     (x3_x0 == 'B1011) ? 'B0000_1000_0000_0000:
19                     (x3_x0 == 'B1100) ? 'B0001_0000_0000_0000:
20                     (x3_x0 == 'B1101) ? 'B0010_0000_0000_0000:
21                     (x3_x0 == 'B1110) ? 'B0100_0000_0000_0000:
22                     /*(x3_x0 == 'B1111)?*/ 'B1000_0000_0000_0000;
23  endmodule
24

```

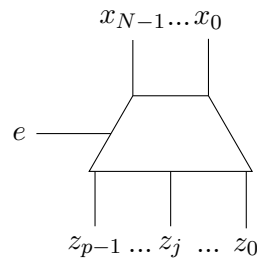
```

25 // implementazione a porte logiche
26 module b4to16_decoder_p(x3_x0, z15_z0);
27     input [3:0] x3_x0;
28     output [15:0] z15_z0;
29
30     assign z15_z0[0] = ~x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & ~x3_x0[0];
31     assign z15_z0[1] = ~x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & x3_x0[0];
32     assign z15_z0[2] = ~x3_x0[3] & ~x3_x0[2] & x3_x0[1] & ~x3_x0[0];
33     assign z15_z0[3] = ~x3_x0[3] & ~x3_x0[2] & x3_x0[1] & x3_x0[0];
34     assign z15_z0[4] = ~x3_x0[3] & x3_x0[2] & ~x3_x0[1] & ~x3_x0[0];
35     assign z15_z0[5] = ~x3_x0[3] & x3_x0[2] & ~x3_x0[1] & x3_x0[0];
36     assign z15_z0[6] = ~x3_x0[3] & x3_x0[2] & x3_x0[1] & ~x3_x0[0];
37     assign z15_z0[7] = ~x3_x0[3] & x3_x0[2] & x3_x0[1] & x3_x0[0];
38     assign z15_z0[8] = x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & ~x3_x0[0];
39     assign z15_z0[9] = x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & x3_x0[0];
40     assign z15_z0[10] = x3_x0[3] & ~x3_x0[2] & x3_x0[1] & ~x3_x0[0];
41     assign z15_z0[11] = x3_x0[3] & ~x3_x0[2] & x3_x0[1] & x3_x0[0];
42     assign z15_z0[12] = x3_x0[3] & x3_x0[2] & ~x3_x0[1] & ~x3_x0[0];
43     assign z15_z0[13] = x3_x0[3] & x3_x0[2] & ~x3_x0[1] & x3_x0[0];
44     assign z15_z0[14] = x3_x0[3] & x3_x0[2] & x3_x0[1] & ~x3_x0[0];
45     assign z15_z0[15] = x3_x0[3] & x3_x0[2] & x3_x0[1] & x3_x0[0];
46 endmodule
47
48 // implementazione gerarchica
49 module b4to16_decoder_g(x3_x0, z15_z0);
50     input [3:0] x3_x0;
51     output [15:0] z15_z0;
52
53     wire [3:0] x3_x2_d;
54     wire [3:0] x1_x0_d;
55
56     b2to4_decoder b2to4_1 (x3_x0[3:2], x3_x2_d);
57     b2to4_decoder b2to4_2 (x3_x0[1:0], x1_x0_d);
58
59     assign z15_z0[0] = x3_x2_d[0] & x1_x0_d[0];
60     assign z15_z0[1] = x3_x2_d[0] & x1_x0_d[1];
61     assign z15_z0[2] = x3_x2_d[0] & x1_x0_d[2];
62     assign z15_z0[3] = x3_x2_d[0] & x1_x0_d[3];
63     assign z15_z0[4] = x3_x2_d[1] & x1_x0_d[0];
64     assign z15_z0[5] = x3_x2_d[1] & x1_x0_d[1];
65     assign z15_z0[6] = x3_x2_d[1] & x1_x0_d[2];
66     assign z15_z0[7] = x3_x2_d[1] & x1_x0_d[3];
67     assign z15_z0[8] = x3_x2_d[2] & x1_x0_d[0];
68     assign z15_z0[9] = x3_x2_d[2] & x1_x0_d[1];
69     assign z15_z0[10] = x3_x2_d[2] & x1_x0_d[2];
70     assign z15_z0[11] = x3_x2_d[2] & x1_x0_d[3];
71     assign z15_z0[12] = x3_x2_d[3] & x1_x0_d[0];
72     assign z15_z0[13] = x3_x2_d[3] & x1_x0_d[1];
73     assign z15_z0[14] = x3_x2_d[3] & x1_x0_d[2];
74     assign z15_z0[15] = x3_x2_d[3] & x1_x0_d[3];
75 endmodule

```

### 9.1.1 Decoder con enabler

Il problema dei decoder come appena descritti è che sono poco agili nell'espansione: non si possono costruire, come avevamo visto per i gli AND o gli OR, reti di più decoder combinati, a meno di non ridursi a ritrovare quelli che sono effettivamente i mintermini della tabella di verità (come si nota dagli esempi). 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:

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

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  $x_3$ .

Vediamo un'esempio pratico, dato dalle implementazioni in Verilog degli stessi decoder visti prima, ma stavolta dotati di enabler (e posti in cascata, nelle sintesi gerarchiche, attraverso tali enabler). Si inizia col decoder 1 a 2:

```

1 // un decoder con enabler da 1 a 2 che prende @x0 come ingresso, @e
2 // come enabler e attiva l'uscita @z1_z0 in codifica one-hot
3 module b1to2_enb_decoder(x0, e, z1_z0);
4     input x0;
5     input e;
6     output [1:0] z1_z0;
7
8     assign z1_z0 = ({e, x0} == 'B10) ? 'B01:
9                   ({e, x0} == 'B11) ? 'B10:
10                  /* don't care */ 'B00;
11 endmodule
12
13 // implementazione a porte logiche
14 module b1to2_enb_decoder_p(x0, e, z1_z0);
15     input x0;
16     input e;
17     output [1:0] z1_z0;
18
19     assign z1_z0[0] = ~x0 & e;
20     assign z1_z0[1] = x0 & e;
21 endmodule

```



Possiamo quindi definire il decoder da 2 a 4:

```

1 // un decoder con enabler da 2 a 4 che prende @x1_x0 come ingresso,
2 // @e come enabler e attiva l'uscita @z4_z0 in codifica one-hot
3 module b2to4_enb_decoder(x1_x0, e, z3_z0);
4     input [1:0] x1_x0;
5     input e;
6     output [3:0] z3_z0;
7
8     assign z3_z0 = ({e, x1_x0} == 'B100) ? 'B0001:
9                   ({e, x1_x0} == 'B101) ? 'B0010:
10                  ({e, x1_x0} == 'B110) ? 'B0100:
11                  ({e, x1_x0} == 'B111) ? 'B1000:
12                  /*      don't care      */ 'B0000;
13 endmodule
14
15 // implementazione a porte logiche
16 module b2to4_enb_decoder_p(x1_x0, e, z3_z0);
17     input [1:0] x1_x0;
18     input e;
19     output [3:0] z3_z0;
20
21     assign z3_z0[0] = ~x1_x0[1] & ~x1_x0[0] & e;
22     assign z3_z0[1] = ~x1_x0[1] & x1_x0[0] & e;
23     assign z3_z0[2] = x1_x0[1] & ~x1_x0[0] & e;
24     assign z3_z0[3] = x1_x0[1] & x1_x0[0] & e;
25 endmodule
26
27 // implementazione gerarchica
28 module b2to4_enb_decoder_g(x1_x0, e, z3_z0);
29     input [1:0] x1_x0;
30     input e;
31     output [3:0] z3_z0;
32
33     wire [1:0] enb;
34
35     b1to2_enb_decoder b1to2_1 (x1_x0[0], enb[1], z3_z0[3:2]);
36     b1to2_enb_decoder b1to2_2 (x1_x0[0], enb[0], z3_z0[1:0]);
37     b1to2_enb_decoder b1to2_c (x1_x0[1], e, enb);
38 endmodule

```

un decoder da 3 a 8:

```

1 // un decoder con enabler da 3 a 8 che prende @x2_x0 come ingresso,
2 // @e come enabler e attiva l'uscita @z7_z0 in codifica one-hot
3 module b3to8_enb_decoder(x2_x0, e, z7_z0);
4     input [2:0] x2_x0;
5     input e;
6     output [7:0] z7_z0;
7
8     assign z7_z0 = ({e, x2_x0} == 'B1000) ? 'B0000_0001:
9                   ({e, x2_x0} == 'B1001) ? 'B0000_0010:
10                  ({e, x2_x0} == 'B1010) ? 'B0000_0100:
11                  ({e, x2_x0} == 'B1011) ? 'B0000_1000:
12                  ({e, x2_x0} == 'B1100) ? 'B0001_0000:
13                  ({e, x2_x0} == 'B1101) ? 'B0010_0000:
14                  ({e, x2_x0} == 'B1110) ? 'B0100_0000:
15                  ({e, x2_x0} == 'B1111) ? 'B1000_0000:
16                  /*      don't care      */ 'B0000_0000;
17 endmodule
18
19 // implementazione a porte logiche

```

```

20 module b3to8_enb_decoder_p(x2_x0, e, z7_z0);
21     input [2:0] x2_x0;
22     input e;
23     output [7:0] z7_z0;
24
25     assign z7_z0[0] = ~x2_x0[2] & ~x2_x0[1] & ~x2_x0[0] & e;
26     assign z7_z0[1] = ~x2_x0[2] & ~x2_x0[1] & x2_x0[0] & e;
27     assign z7_z0[2] = ~x2_x0[2] & x2_x0[1] & ~x2_x0[0] & e;
28     assign z7_z0[3] = ~x2_x0[2] & x2_x0[1] & x2_x0[0] & e;
29     assign z7_z0[4] = x2_x0[2] & ~x2_x0[1] & ~x2_x0[0] & e;
30     assign z7_z0[5] = x2_x0[2] & ~x2_x0[1] & x2_x0[0] & e;
31     assign z7_z0[6] = x2_x0[2] & x2_x0[1] & ~x2_x0[0] & e;
32     assign z7_z0[7] = x2_x0[2] & x2_x0[1] & x2_x0[0] & e;
33 endmodule
34
35 // implementazione gerarchica
36 module b3to8_enb_decoder_g(x2_x0, e, z7_z0);
37     input [2:0] x2_x0;
38     input e;
39     output [7:0] z7_z0;
40
41     wire [1:0] enb;
42
43     b2to4_decoder b2to4_1 (x2_x0[1:0], enb[1], z7_z0[7:4]);
44     b2to4_decoder b2to4_2 (x2_x0[1:0], enb[0], z7_z0[3:0]);
45     b1to2_decoder b1to2_c (x2_x0[2], e, enb);
46 endmodule

```

e infine, di cui notiamo la sintesi gerarchica molto più immediata rispetto al caso senza enabler, un decoder da 4 a 16:

```

1 // un decoder con enabler da 4 a 16 che prende @x3_x0 come ingresso,
2 // @e come enabler e attiva l'uscita @z15_z0 in codifica one-hot
3 module b4to16_enb_decoder(x3_x0, e, z15_z0);
4     input [3:0] x3_x0;
5     input e;
6     output [15:0] z15_z0;
7
8     assign z15_z0 = ({e, x3_x0} == 'B10000) ? 'B0000_0000_0000_0001:
9                     ({e, x3_x0} == 'B10001) ? 'B0000_0000_0000_0010:
10                    ({e, x3_x0} == 'B10010) ? 'B0000_0000_0000_0100:
11                    ({e, x3_x0} == 'B10011) ? 'B0000_0000_0000_1000:
12                    ({e, x3_x0} == 'B10100) ? 'B0000_0000_0001_0000:
13                    ({e, x3_x0} == 'B10101) ? 'B0000_0000_0010_0000:
14                    ({e, x3_x0} == 'B10110) ? 'B0000_0000_0100_0000:
15                    ({e, x3_x0} == 'B10111) ? 'B0000_0000_1000_0000:
16                    ({e, x3_x0} == 'B11000) ? 'B0000_0001_0000_0000:
17                    ({e, x3_x0} == 'B11001) ? 'B0000_0010_0000_0000:
18                    ({e, x3_x0} == 'B11010) ? 'B0000_0100_0000_0000:
19                    ({e, x3_x0} == 'B11011) ? 'B0000_1000_0000_0000:
20                    ({e, x3_x0} == 'B11100) ? 'B0001_0000_0000_0000:
21                    ({e, x3_x0} == 'B11101) ? 'B0010_0000_0000_0000:
22                    ({e, x3_x0} == 'B11110) ? 'B0100_0000_0000_0000:
23                    ({e, x3_x0} == 'B11111) ? 'B1000_0000_0000_0000:
24                    /* don't care */ 'B0000_0000_0000_0000;
25 endmodule
26
27 // implementazione a porte logiche
28 module b4to16_enb_decoder_p(x3_x0, e, z15_z0);
29     input [3:0] x3_x0;
30     input e;

```

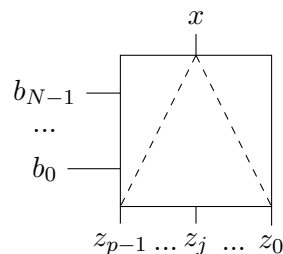
```

31 output [15:0] z15_z0;
32
33 assign z15_z0[0] = ~x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & ~x3_x0[0]
34                                     & e;
35 assign z15_z0[1] = ~x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & x3_x0[0]
36                                     & e;
37 assign z15_z0[2] = ~x3_x0[3] & ~x3_x0[2] & x3_x0[1] & ~x3_x0[0]
38                                     & e;
39 assign z15_z0[3] = ~x3_x0[3] & ~x3_x0[2] & x3_x0[1] & x3_x0[0] & e;
40 assign z15_z0[4] = ~x3_x0[3] & x3_x0[2] & ~x3_x0[1] & ~x3_x0[0]
41                                     & e;
42 assign z15_z0[5] = ~x3_x0[3] & x3_x0[2] & ~x3_x0[1] & x3_x0[0] & e;
43 assign z15_z0[6] = ~x3_x0[3] & x3_x0[2] & x3_x0[1] & ~x3_x0[0] & e;
44 assign z15_z0[7] = ~x3_x0[3] & x3_x0[2] & x3_x0[1] & x3_x0[0] & e;
45 assign z15_z0[8] = x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & ~x3_x0[0]
46                                     & e;
47 assign z15_z0[9] = x3_x0[3] & ~x3_x0[2] & ~x3_x0[1] & x3_x0[0] & e;
48 assign z15_z0[10] = x3_x0[3] & ~x3_x0[2] & x3_x0[1] & ~x3_x0[0]
49                                     & e;
50 assign z15_z0[11] = x3_x0[3] & ~x3_x0[2] & x3_x0[1] & x3_x0[0] & e;
51 assign z15_z0[12] = x3_x0[3] & x3_x0[2] & ~x3_x0[1] & ~x3_x0[0]
52                                     & e;
53 assign z15_z0[13] = x3_x0[3] & x3_x0[2] & ~x3_x0[1] & x3_x0[0] & e;
54 assign z15_z0[14] = x3_x0[3] & x3_x0[2] & x3_x0[1] & ~x3_x0[0] & e;
55 assign z15_z0[15] = x3_x0[3] & x3_x0[2] & x3_x0[1] & x3_x0[0] & e;
56 endmodule
57
58 // implementazione gerarchica
59 module b4to16_enb_decoder_g(x3_x0, e, z15_z0);
60     input [3:0] x3_x0;
61     input e;
62     output [15:0] z15_z0;
63
64     wire [3:0] enb;
65
66     b2to4_enb_decoder b2to4_1 (x3_x0[1:0], enb[3], z15_z0[15:12]);
67     b2to4_enb_decoder b2to4_2 (x3_x0[1:0], enb[2], z15_z0[11:8]);
68     b2to4_enb_decoder b2to4_3 (x3_x0[1:0], enb[1], z15_z0[7:4]);
69     b2to4_enb_decoder b2to4_4 (x3_x0[1:0], enb[0], z15_z0[3:0]);
70     b2to4_enb_decoder b2to4_c (x3_x0[3:2], e, enb);
71 endmodule

```

## 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:

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

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:

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

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

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

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

Vediamo infatti l'implementazione in Verilog di un demultiplexer da 1 a 2:

```

1 // un demultiplexer da 1 a 2 che prende @x come ingresso, @b0 come
2 // comando e @z1_z0 come uscita
3 module b1to2_demuxer(x0, b0, z1_z0);
4     input x0;
5     input b0;
6     output [1:0] z1_z0;
7
8     assign z1_z0 = ({x0, b0} == 'B10) ? 'B01:
9                   ({x0, b0} == 'B11) ? 'B10:
10                  /* don't care */ 'B00;
11 endmodule
12
13 // implementazione a porte logiche
14 module b1to2_demuxer(x0, b0, z1_z0);
15     input x0;
16     input b0;
17     output [1:0] z1_z0;
18
19     assign z1_z0[0] = ~x0 & b0;
20     assign z1_z0[1] = x0 & b0;
21 endmodule

```

e come se ne può ricavare uno da 1 a 4:

```

1 // un demultiplexer da 1 a 4 che prende @x0 come ingresso, @b1_b0
2 // come comando e @z4_z0 come uscita
3 module b1to4_demuxer(x0, b3_b0, z3_z0);
4     input x0;
5     input [1:0] b1_b0;
6     output [3:0] z3_z0;
7
8     assign z3_z0 = ({x0, b3_b0} == 'B100) ? 'B0001:
9                   ({x0, b3_b0} == 'B101) ? 'B0010:
10                  ({x0, b3_b0} == 'B110) ? 'B0100:
11                  ({x0, b3_b0} == 'B111) ? 'B1000:

```

```

12          /*      don't care      */ 'B0000;
13 endmodule
14
15 // implementazione a porte logiche
16 module b1to4_demuxer(x0, b1_b0, z3_z0);
17     input x0;
18     input [1:0] b1_b0;
19     output [3:0] z3_z0;
20
21     assign z3_z0[0] = ~b1_b0[1] & ~x1_x0[0] & x0;
22     assign z3_z0[1] = ~b1_b0[1] & x1_x0[0] & x0;
23     assign z3_z0[2] = b1_b0[1] & ~x1_x0[0] & x0;
24     assign z3_z0[3] = b1_b0[1] & x1_x0[0] & x0;
25 endmodule
26
27 // implementazione gerarchica
28 module b1to4_demuxer(x0, b1_b0, z3_z0);
29     input x0;
30     input [1:0] b1_b0;
31     output [3:0] z3_z0;
32
33     wire [1:0] enb;
34
35     b1to2_demuxer b1to2_1 (b1_b0[0], enb[1], z3_z0[3:2]);
36     b1to2_demuxer b1to2_2 (b1_b0[0], enb[0], z3_z0[1:0]);
37     b1to2_demuxer b1to2_c (b1_b0[1], x0, enb);
38 endmodule

```

da 1 a 8:

```

1 // un demultiplexer da 1 a 8 che prende @x0 come ingresso, @b2_b0
2 // come comando e @z7_z0 come uscita
3 module b1to8_demuxer(x0, b2_b0, z7_z0);
4     input x0;
5     input [2:0] b2_b0;
6     output [7:0] z7_z0;
7
8     assign z7_z0 = ({x0, b2_b0} == 'B1000) ? 'B0000_0001:
9                   ({x0, b2_b0} == 'B1001) ? 'B0000_0010:
10                  ({x0, b2_b0} == 'B1010) ? 'B0000_0100:
11                  ({x0, b2_b0} == 'B1011) ? 'B0000_1000:
12                  ({x0, b2_b0} == 'B1100) ? 'B0001_0000:
13                  ({x0, b2_b0} == 'B1101) ? 'B0010_0000:
14                  ({x0, b2_b0} == 'B1110) ? 'B0100_0000:
15                  ({x0, b2_b0} == 'B1111) ? 'B1000_0000:
16                  /*      don't care      */ 'B0000_0000;
17 endmodule
18
19 // implementazione a porte logiche
20 module b1to8_demuxer_p(x0, b2_b0, z7_z0);
21     input x0;
22     input [2:0] b2_b0;
23     output [7:0] z7_z0;
24
25     assign z7_z0[0] = ~b2_b0[2] & ~b2_b0[1] & ~b2_b0[0] & x0;
26     assign z7_z0[1] = ~b2_b0[2] & ~b2_b0[1] & b2_b0[0] & x0;
27     assign z7_z0[2] = ~b2_b0[2] & b2_b0[1] & ~b2_b0[0] & x0;
28     assign z7_z0[3] = ~b2_b0[2] & b2_b0[1] & b2_b0[0] & x0;
29     assign z7_z0[4] = b2_b0[2] & ~b2_b0[1] & ~b2_b0[0] & x0;
30     assign z7_z0[5] = b2_b0[2] & ~b2_b0[1] & b2_b0[0] & x0;
31     assign z7_z0[6] = b2_b0[2] & b2_b0[1] & ~b2_b0[0] & x0;

```

```

32  assign z7_z0[7] = b2_b0[2] & b2_b0[1] & b2_b0[0] & x0;
33  endmodule
34
35  // implementazione gerarchica
36  module b1to8_demuxer_g(x0, b2_b0, z7_z0);
37      input x0;
38      input [2:0] b2_b0;
39      output [7:0] z7_z0;
40
41      wire [1:0] enb;
42
43      b2to4_demuxer b2to4_1 (b2_b0[1:0], enb[1], z7_z0[7:4]);
44      b2to4_demuxer b2to4_2 (b2_b0[1:0], enb[0], z7_z0[3:0]);
45      b1to2_demuxer b1to2_c (b2_b0[2], x0, enb);
46  endmodule

```

e infine da 1 a 16:

```

1  // un demultiplexer da 1 a 16 che prende @0 come ingresso, @b3_b0
2  // come comando e @z15_z0 come uscita
3  module b1to16_demuxer(x0, b3_b0, z15_z0);
4      input x0;
5      input [3:0] b3_b0;
6      output [15:0] z15_z0;
7
8      assign z15_z0 = ({x0, b3_b0} == 'B10000) ? 'B0000_0000_0000_0001:
9                      ({x0, b3_b0} == 'B10001) ? 'B0000_0000_0000_0010:
10                     ({x0, b3_b0} == 'B10010) ? 'B0000_0000_0000_0100:
11                     ({x0, b3_b0} == 'B10011) ? 'B0000_0000_0000_1000:
12                     ({x0, b3_b0} == 'B10100) ? 'B0000_0000_0001_0000:
13                     ({x0, b3_b0} == 'B10101) ? 'B0000_0000_0010_0000:
14                     ({x0, b3_b0} == 'B10110) ? 'B0000_0000_0100_0000:
15                     ({x0, b3_b0} == 'B10111) ? 'B0000_0000_1000_0000:
16                     ({x0, b3_b0} == 'B11000) ? 'B0000_0001_0000_0000:
17                     ({x0, b3_b0} == 'B11001) ? 'B0000_0010_0000_0000:
18                     ({x0, b3_b0} == 'B11010) ? 'B0000_0100_0000_0000:
19                     ({x0, b3_b0} == 'B11011) ? 'B0000_1000_0000_0000:
20                     ({x0, b3_b0} == 'B11100) ? 'B0001_0000_0000_0000:
21                     ({x0, b3_b0} == 'B11101) ? 'B0010_0000_0000_0000:
22                     ({x0, b3_b0} == 'B11110) ? 'B0100_0000_0000_0000:
23                     ({x0, b3_b0} == 'B11111) ? 'B1000_0000_0000_0000:
24                      /* don't care */ 'B0000_0000_0000_0000;
25  endmodule
26
27  // implementazione a porte logiche
28  module b1to16_demuxer_p(x0, b3_b0, z15_z0);
29      input x0;
30      input b3_b0;
31      output [15:0] z15_z0;
32
33      assign z15_z0[0] = ~b3_b0[3] & ~b3_b0[2] & ~b3_b0[1] & ~b3_b0[0]
34                      & x0;
35      assign z15_z0[1] = ~b3_b0[3] & ~b3_b0[2] & ~b3_b0[1] & b3_b0[0]
36                      & x0;
37      assign z15_z0[2] = ~b3_b0[3] & ~b3_b0[2] & b3_b0[1] & ~b3_b0[0]
38                      & x0;
39      assign z15_z0[3] = ~b3_b0[3] & ~b3_b0[2] & b3_b0[1] & b3_b0[0]
40                      & x0;
41      assign z15_z0[4] = ~b3_b0[3] & b3_b0[2] & ~b3_b0[1] & ~b3_b0[0]
42                      & x0;
43      assign z15_z0[5] = ~b3_b0[3] & b3_b0[2] & ~b3_b0[1] & b3_b0[0]

```

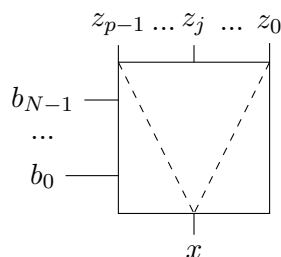
```

44                                     & x0;
45 assign z15_z0[6] = ~b3_b0[3] & b3_b0[2] & b3_b0[1] & ~b3_b0[0]
46                                     & x0;
47 assign z15_z0[7] = ~b3_b0[3] & b3_b0[2] & b3_b0[1] & b3_b0[0] & x0;
48 assign z15_z0[8] = b3_b0[3] & ~b3_b0[2] & ~b3_b0[1] & ~b3_b0[0]
49                                     & x0;
50 assign z15_z0[9] = b3_b0[3] & ~b3_b0[2] & ~b3_b0[1] & b3_b0[0]
51                                     & x0;
52 assign z15_z0[10] = b3_b0[3] & ~b3_b0[2] & b3_b0[1] & ~b3_b0[0]
53                                     & x0;
54 assign z15_z0[11] = b3_b0[3] & ~b3_b0[2] & b3_b0[1] & b3_b0[0]
55                                     & x0;
56 assign z15_z0[12] = b3_b0[3] & b3_b0[2] & ~b3_b0[1] & ~b3_b0[0]
57                                     & x0;
58 assign z15_z0[13] = b3_b0[3] & b3_b0[2] & ~b3_b0[1] & b3_b0[0]
59                                     & x0;
60 assign z15_z0[14] = b3_b0[3] & b3_b0[2] & b3_b0[1] & ~b3_b0[0]
61                                     & x0;
62 assign z15_z0[15] = b3_b0[3] & b3_b0[2] & b3_b0[1] & b3_b0[0] & x0;
63 endmodule
64
65 // implementazione gerarchica
66 module b1to16_enb_decoder_g(x0, b3_b0, z15_z0);
67     input x0;
68     input b3_b0;
69     output[15:0] z15_z0;
70
71     wire[3:0] enb;
72
73     b2to4_enb_decoder b2to4_1 (b3_b0[1:0], enb[3], z15_z0[15:12]);
74     b2to4_enb_decoder b2to4_2 (b3_b0[1:0], enb[2], z15_z0[11:8]);
75     b2to4_enb_decoder b2to4_3 (b3_b0[1:0], enb[1], z15_z0[7:4]);
76     b2to4_enb_decoder b2to4_4 (b3_b0[1:0], enb[0], z15_z0[3:0]);
77     b2to4_enb_decoder b2to4_c (b3_b0[3:2], x0, enb);
78 endmodule

```

### 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:

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

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 finle 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 → descrizione, mentre fino a questo punto avevamo fatto l'operazione inversa, descrizione → sintesi):

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

Notiamo che il multiplexer è una rete a 2 livelli di logica: il segnale passerà al massimo da una AND e una 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.

Vediamo quindi un'implementazione in Verilog di un multiplexer da 2 a 1:

```

1 // un multiplexer da 2 a 1 che prende @x1_x0 come ingresso, @b0 come
2 // comando e @z0 come uscita
3 module b2to1_muxer(x1_x0, b0, z0);
4     input [1:0] x1_x0;
5     input b0;
6     output z0;
7
8     assign z0 = (b0 == 'B0) ? x1_x0[0]:
9                 /*(b0 == 'B1)?*/x1_x0[1];
10 endmodule
11
12 // implementazione a porte logiche
13 module b2to1_muxer_p(x1_x0, b0, z0);
14     input [1:0] x1_x0;
15     input b0;
16     output z0;
17
18     assign z0 = b0 & x1_x0[1] | ~b0 & x1_x0[0];
19 endmodule
20
21 // implementazione via decoder
22 module b2to1_muxer_d(x1_x0, b0, z0);
23     input [1:0] x1_x0;
24     input b0;
25     output z0;
26
27     wire [1:0] s1_s0;
28
29     b1to2_decoder b1to2(b0, s1_s0);
30
31     assign z0 = s1_s0[1] & x1_x0[1] | s1_s0[0] & x1_x0[0];
32 endmodule

```



e come se ne può ricavare uno da 4 a 1:

```

1 // un multiplexer da 4 a 1 che prende @x3_x0 come ingresso, @b1_b0
2 // come comando e @z0 come uscita
3 module b4to1_muxer(x3_x0, b1_b0, z0);
4     input [3:0] x3_x0;
5     input [1:0] b1_b0;
6     output z0;
7
8     assign z0 = (b1_b0 == 'B00) ? x3_x0[0]:
9                 (b1_b0 == 'B01) ? x3_x0[1]:
10                (b1_b0 == 'B10) ? x3_x0[2]:
11                /*(b1_b0 == 'B11)?*/x3_x0[3];
12 endmodule
13
14 // implementazione a porte logiche
15 module b4to1_muxer_p(x3_x0, b1_b0, z0);
16     input [3:0] x3_x0;
17     input [1:0] b1_b0;
18     output z0;
19
20     assign z0 = b1_b0[1] & b1_b0[0] & x3_x0[3]
21                | b1_b0[1] & ~b1_b0[0] & x3_x0[2]
22                | ~b1_b0[1] & b1_b0[0] & x3_x0[1]
23                | ~b1_b0[1] & ~b1_b0[0] & x3_x0[0];
24 endmodule
25
26 // implementazione via decoder
27 module b4to1_muxer_d(x3_x0, b1_b0, z0);
28     input [3:0] x3_x0;
29     input [1:0] b1_b0;
30     output z0;
31
32     wire [3:0] s3_s0;
33
34     b2to4_decoder b2to4(b1_b0, s3_s0);
35
36     assign z0 = s3_s0[3] & x3_x0[3] | s3_s0[2] & x3_x0[2]
37                | s3_s0[1] & x3_x0[1] | s3_s0[0] & x3_x0[0];
38 endmodule
39
40 // implementazione gerarchica
41 module b4to1_muxer_g(x3_x0, b1_b0, z0);
42     input [3:0] x3_x0;
43     input [1:0] b1_b0;
44     output z0;
45
46     wire [1:0] s1_s0;
47
48     b2to1_muxer b2to1_1 (x3_x0[3:2], b1_b0[0], s1_s0[1]);
49     b2to1_muxer b2to1_2 (x3_x0[1:0], b1_b0[0], s1_s0[0]);
50     b2to1_muxer b2to1_f (s1_s0, b1_b0[1], z0);
51 endmodule

```

da 8 a 1:

```

1 // un multiplexer da 8 a 1 che prende @x7_x0 come ingresso, @b2_b0
2 // come comando e @z0 come uscita
3 module b8to1_muxer(x7_x0, b2_b0, z0);
4     input [7:0] x7_x0;
5     input [2:0] b2_b0;
6     output z0;

```

```

7
8  assign z0 = (b2_b0 == 'B000) ? x7_x0[0]:
9              (b2_b0 == 'B001) ? x7_x0[1]:
10             (b2_b0 == 'B010) ? x7_x0[2]:
11             (b2_b0 == 'B011) ? x7_x0[3]:
12             (b2_b0 == 'B100) ? x7_x0[4]:
13             (b2_b0 == 'B101) ? x7_x0[5]:
14             (b2_b0 == 'B110) ? x7_x0[6]:
15             /*(b2_b0 == 'B111)?*/x7_x0[7];
16 endmodule
17
18 // implementazione a porte logiche
19 module b8to1_muxer_p(x7_x0, b2_b0, z0);
20     input [7:0] x7_x0;
21     input [2:0] b2_b0;
22     output z0;
23
24     assign z0 = b2_b0[2] & b2_b0[1] & b2_b0[0] & x7_x0[7]
25                | b2_b0[2] & b2_b0[1] & ~b2_b0[0] & x7_x0[6]
26                | b2_b0[2] & ~b2_b0[1] & b2_b0[0] & x7_x0[5]
27                | b2_b0[2] & ~b2_b0[1] & ~b2_b0[0] & x7_x0[4]
28                | ~b2_b0[2] & b2_b0[1] & b2_b0[0] & x7_x0[3]
29                | ~b2_b0[2] & b2_b0[1] & ~b2_b0[0] & x7_x0[2]
30                | ~b2_b0[2] & ~b2_b0[1] & b2_b0[0] & x7_x0[1]
31                | ~b2_b0[2] & ~b2_b0[1] & ~b2_b0[0] & x7_x0[0];
32 endmodule
33
34 // implementazione via decoder
35 module b8to1_muxer_d(x7_x0, b2_b0, z0);
36     input [7:0] x7_x0;
37     input [2:0] b2_b0;
38     output z0;
39
40     wire [7:0] s7_s0;
41
42     b3to8_decoder b3to8(b2_b0, s7_s0);
43
44     assign z0 = s7_s0[7] & x7_x0[7] | s7_s0[6] & x7_x0[6]
45                | s7_s0[5] & x7_x0[5] | s7_s0[4] & x7_x0[4]
46                | s7_s0[3] & x7_x0[3] | s7_s0[2] & x7_x0[2]
47                | s7_s0[1] & x7_x0[1] | s7_s0[0] & x7_x0[0];
48 endmodule
49
50 // implementazione gerarchica
51 module b8to1_muxer_g(x7_x0, b2_b0, z0);
52     input [7:0] x7_x0;
53     input [2:0] b2_b0;
54     output z0;
55
56     wire [1:0] s1_s0;
57
58     b4to1_muxer b4to1_1 (x7_x0[7:4], b2_b0[1:0], s1_s0[1]);
59     b4to1_muxer b4to1_2 (x7_x0[3:0], b2_b0[1:0], s1_s0[0]);
60     b2to1_muxer b2to1_f (s1_s0, b2_b0[2], z0);
61 endmodule

```

e infine da 16 a 1:

```

1 // un multiplexer da 16 a 1 che prende @x15_x0 come ingresso, @b3_b0
2 // come comando e @z0 come uscita
3 module b16to1_muxer(x15_x0, b3_b0, z0);

```

```

4  input [15:0] x15_x0;
5  input [3:0] b3_b0;
6  output z0;
7
8  assign z0 = (b3_b0 == 'B0000) ? x15_x0[0]:
9              (b3_b0 == 'B0001) ? x15_x0[1]:
10             (b3_b0 == 'B0010) ? x15_x0[2]:
11             (b3_b0 == 'B0011) ? x15_x0[3]:
12             (b3_b0 == 'B0100) ? x15_x0[4]:
13             (b3_b0 == 'B0101) ? x15_x0[5]:
14             (b3_b0 == 'B0110) ? x15_x0[6]:
15             (b3_b0 == 'B0111) ? x15_x0[7]:
16             (b3_b0 == 'B1000) ? x15_x0[8]:
17             (b3_b0 == 'B1001) ? x15_x0[9]:
18             (b3_b0 == 'B1010) ? x15_x0[10]:
19             (b3_b0 == 'B1011) ? x15_x0[11]:
20             (b3_b0 == 'B1100) ? x15_x0[12]:
21             (b3_b0 == 'B1101) ? x15_x0[13]:
22             (b3_b0 == 'B1110) ? x15_x0[14]:
23             /*(b3_b0 == 'B1111)?*/x15_x0[15];
24 endmodule
25
26 // implementazione a porte logiche
27 module b16to1_muxer_p(x15_x0, b3_b0, z0);
28     input [15:0] x15_x0;
29     input [3:0] b3_b0;
30     output z0;
31
32     assign z0 = b3_b0[3] & b3_b0[2] & b3_b0[1] & b3_b0[0] & x15_x0[15]
33                | b3_b0[3] & b3_b0[2] & b3_b0[1] & ~b3_b0[0] & x15_x0[14]
34                | b3_b0[3] & b3_b0[2] & ~b3_b0[1] & b3_b0[0] & x15_x0[13]
35                | b3_b0[3] & b3_b0[2] & ~b3_b0[1] & ~b3_b0[0]
36                  & x15_x0[12]
37                | b3_b0[3] & ~b3_b0[2] & b3_b0[1] & b3_b0[0] & x15_x0[11]
38                | b3_b0[3] & ~b3_b0[2] & b3_b0[1] & ~b3_b0[0]
39                  & x15_x0[10]
40                | b3_b0[3] & ~b3_b0[2] & ~b3_b0[1] & b3_b0[0] & x15_x0[9]
41                | b3_b0[3] & ~b3_b0[2] & ~b3_b0[1] & ~b3_b0[0]
42                  & x15_x0[8]
43                | ~b3_b0[3] & b3_b0[2] & b3_b0[1] & b3_b0[0] & x15_x0[7]
44                | ~b3_b0[3] & b3_b0[2] & b3_b0[1] & ~b3_b0[0]
45                  & x15_x0[6]
46                | ~b3_b0[3] & b3_b0[2] & ~b3_b0[1] & b3_b0[0] & x15_x0[5]
47                | ~b3_b0[3] & b3_b0[2] & ~b3_b0[1] & ~b3_b0[0]
48                  & x15_x0[4]
49                | ~b3_b0[3] & ~b3_b0[2] & b3_b0[1] & b3_b0[0] & x15_x0[3]
50                | ~b3_b0[3] & ~b3_b0[2] & b3_b0[1] & ~b3_b0[0]
51                  & x15_x0[2]
52                | ~b3_b0[3] & ~b3_b0[2] & ~b3_b0[1] & b3_b0[0]
53                  & x15_x0[1]
54                | ~b3_b0[3] & ~b3_b0[2] & ~b3_b0[1] & ~b3_b0[0]
55                  & x15_x0[0];
56 endmodule
57
58 // implementazione via decoder
59 module b16to1_muxer_d(x15_x0, b3_b0, z0);
60     input [15:0] x15_x0;
61     input [3:0] b3_b0;
62     output z0;
63

```

```

64 wire [15:0] s15_s0;
65
66 b4to16_decoder b3to8(b3_b0, s15_s0);
67
68 assign z0 = s15_s0[15] & x15_x0[15] | s15_s0[14] & x15_x0[14]
69           | s15_s0[13] & x15_x0[13] | s15_s0[12] & x15_x0[12]
70           | s15_s0[11] & x15_x0[11] | s15_s0[10] & x15_x0[10]
71           | s15_s0[9] & x15_x0[9] | s15_s0[8] & x15_x0[8]
72           | s15_s0[7] & x15_x0[7] | s15_s0[6] & x15_x0[6]
73           | s15_s0[5] & x15_x0[5] | s15_s0[4] & x15_x0[4]
74           | s15_s0[3] & x15_x0[3] | s15_s0[2] & x15_x0[2]
75           | s15_s0[1] & x15_x0[1] | s15_s0[0] & x15_x0[0];
76 endmodule
77
78 // implementazione gerarchica
79 module b16to1_muxer_g(x15_x0, b3_b0, z0);
80     input [15:0] x15_x0;
81     input [3:0] b3_b0;
82     output z0;
83
84     wire [3:0] s3_s0;
85
86     b4to1_muxer b4to1_1 (x15_x0[15:12], b3_b0[1:0], s3_s0[3]);
87     b4to1_muxer b4to1_2 (x15_x0[11:8], b3_b0[1:0], s3_s0[2]);
88     b4to1_muxer b4to1_3 (x15_x0[7:4], b3_b0[1:0], s3_s0[1]);
89     b4to1_muxer b4to1_4 (x15_x0[3:0], b3_b0[1:0], s3_s0[0]);
90     b4to1_muxer b4to1_f (s3_s0, b3_b0[3:2], z0);
91 endmodule

```

### 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 parallelo".

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:

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

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::

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

#### 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  $z = f(x_{N-1}, \dots, X_0)$ , posso dire:

$$\begin{aligned} z = & f(0, \dots, 0, 0) \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot \overline{x_0} + \\ & f(0, \dots, 0, 1) \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot x_0 + \\ & \dots + \\ & f(1, \dots, 1, 0) \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot \overline{x_0} + \\ & f(1, \dots, 1, 1) \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot x_0 \end{aligned}$$

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 implicant 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 implicant 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 implicant 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 implicant di 1. Si ricava una forma SP, detta **lista di implicant**.

- **Rimuovere i duplicati** dalla lista degli implicanti, applicando un'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 modellizzazione 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 ridondanze. Vediamo, ad esempio, mappe con  $N = 2, 3$  e 4:

		$X_0$	
		0	1
$X_1$	0	0	1
	1	1	0

		$X_1 X_0$			
		00	01	11	10
$X_2$	0	0	0	1	-
	1	1	-	0	0

		$X_1 X_0$			
		00	01	11	10
$X_3 X_2$	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:

		$X_0$				
		0	1			
$X_1$	0	0	1		$X_1$	$X_0$
	1	1	0		A	0
					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  $N = 4$ :



		$X_1X_0$			
		00	01	11	10
$X_3X_2$	00	1	1	0	1
	01	0	0	1	1
	11	0	0	1	1
	10	1	0	0	1

	$X_3$	$X_2$	$X_1$	$X_0$
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 la 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:

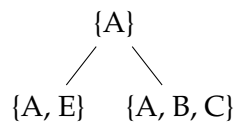
- **Essenziale**, in questo caso resta nella lista di copertura, e qualche altro sottocubo diventa assolutamente eliminabile;
- **Assolutamente eliminabile**, in questo caso si toglie dalla lista di copertura, e qualche altro sottocubo diventa essenziale.

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

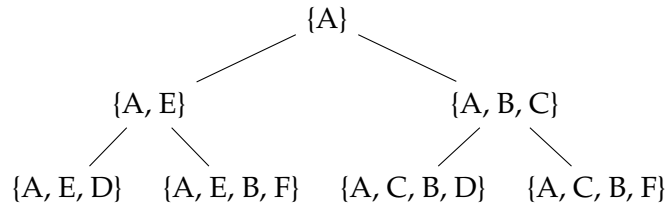
		$X_1X_0$			
		00	01	11	10
$X_3X_2$	00	0	0	0	1
	01	1	0	0	1
	11	1	1	0	0
	10	1	1	0	1

	$X_3$	$X_2$	$X_1$	$X_0$
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:



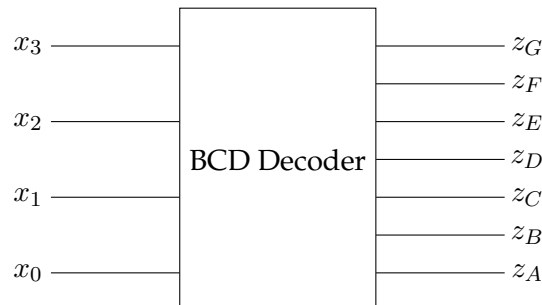
Scegliamo quindi {A, E, D}.

## 11 Lezione del 11-10-24

### 11.1 Sintesi di leggi non completamente specificate

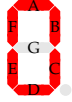
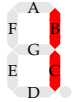

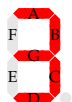
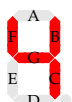
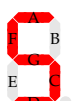
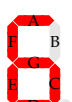



Applichiamo quanto abbiamo detto sulla sintesi di reti in forma SP a costo minimo, nel caso particolare in cui la legge non è completamente specificata (*don't care*).

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	—

Disegniamo 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:

		$X_1X_0$								
		00	01	11	10					
$X_3X_2$	00	1	0	0	1		$x_3$	$x_2$	$x_1$	$x_0$
	01	0	0	0	1					
	11	-	-	-	-					
	10	1	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 implicant 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 dell'uscita delle rete è allora:

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

Possiamo ripetere questa procedura per ogni uscita della rete, andando quindi a sintetizzare completamente il decodificatore BCD. Ad esempio, attraverso il pacchetto software Logisim-Evolution si è generato:

$$z_A = \overline{x_2} \cdot \overline{x_0} + x_1 + x_2 \cdot x_0 + x_3$$

$$z_B = \overline{x_2} + \overline{x_1} \cdot \overline{x_0} + x_1 \cdot x_0 + x_3$$

$$z_C = \overline{x_1} + x_0 + x_2 + x_3$$

$$z_D = \overline{x_2} \cdot \overline{x_0} + \overline{x_2} \cdot x_1 + x_2 \cdot \overline{x_1} \cdot x_0 + x_1 \cdot \overline{x_0} + x_3$$

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

$$z_F = \overline{x_1} \cdot \overline{x_0} + x_2 \cdot \overline{x_1} + x_2 \cdot \overline{x_0} + x_3$$

$$z_G = \overline{x_2} \cdot x_1 + x_2 \cdot \overline{x_1} + x_3 + x_1 \cdot \overline{x_0}$$

## 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 realizza 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:

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

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

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

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  $\bar{F}$  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:

		$X_1X_0$			
		00	01	11	10
$X_3X_2$	00	0	1	1	0
	01	1	1	1	0
	11	-	-	-	-
	10	0	1	-	-

	$x_3$	$x_2$	$x_1$	$x_0$
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  $\bar{F}$  è:

$$\bar{F} = \overline{z_E} = x_0 + x_2\bar{x}_1$$

che neghiamo per ottenere nuovamente  $F$ :

$$F = z_E = \overline{x_0 + x_2\bar{x}_1}$$

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

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

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.

## 12 Lezione del 15-10-24

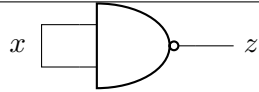
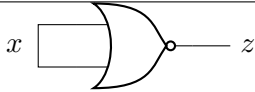
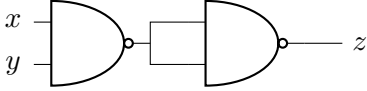
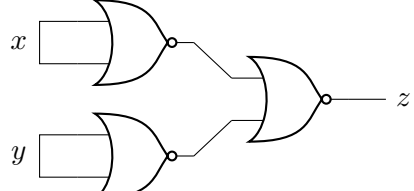
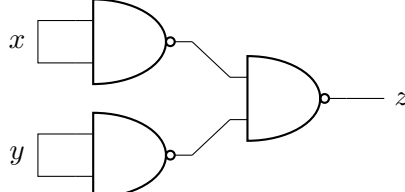
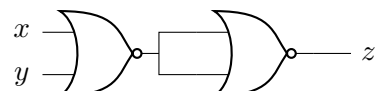
### 12.1 Porte logiche universali

Si dice che NAND e NOR sono **porte logiche universali**. Si possono realizzare AND, OR e NOT usando solo porte NAND o solo porte NOR.

Algebricamente, questo significa fare:

Porta	Realizzazione NAND	Realizzazione NOR
NOT	$x = x \cdot x \Rightarrow \overline{x} = \overline{x \cdot x}$	$x = x + x \Rightarrow \overline{x} = \overline{x + x}$
AND	$x \cdot y = \overline{(\overline{x \cdot y})}$	$x \cdot y = \overline{\overline{x} + \overline{y}}$ (de Morgan)
OR	$x + y = \overline{\overline{x} \cdot \overline{y}}$ (de Morgan)	$x + y = \overline{(\overline{x + y})}$

cioè collegare porte logiche fisiche nelle seguenti configurazioni:

Porta	Realizzazione NAND	Realizzazione NOR
NOT		
AND		
OR		

Potremmo sembrare che, se si poteva realizzare qualsiasi rete combinatoria con AND, OR e NOT su 2 livelli di logica, usando solo NAND o NOR dovremmo accontentarci di 4 livelli di logica (AND e OR richiedono di per sé una rete a 2 livelli di logica).

In verità, le porte NAND e NOR permettono di creare circuiti logici con gli stessi livelli di logica delle porte AND, OR e NOT.

#### 12.1.1 Sintesi a porte NAND

Vediamo quindi il seguente algoritmo per la sintesi di un circuito con sole porte NAND:



**Algoritmo 5** sintesi a porte NAND**Input:** un circuito in forma SP**Output:** una sintesi a porte NAND

Si sostituisce la porta OR con il suo equivalente a NAND

Si sostituisce ciascun AND con il suo equivalente a NAND

Si eliminano le coppie di NOT interne a cascata

Ignoriamo i NOT sull'ingresso, in quanto abbiamo visto sono effettivamente gratuiti. Abbiamo quindi che, rimuovendo le coppie NOT interni (creati da coppie di NAND con gli stessi input) ritorniamo in una forma a 2 livelli di logica.

Dal punto di vista algebrico si ha:

$$z = P_1 + P_2 + \dots + P_k = \overline{\overline{P_1 + P_2 + \dots + P_k}} = \overline{\overline{P_1} \cdot \overline{P_2} \cdot \dots \cdot \overline{P_k}}$$

dove il complemento superiore è l'ultima porta NAND (quella che sostituisce l'OR), e i singoli  $P_i$  complementati sono singole porte NAND ( $P_i$  è un prodotto, quindi  $\overline{P_i}$  è una porta NAND).

**12.1.2 Sintesi a porte NOR**

Vediamo poi l'algoritmo per la sintesi di un circuito con sole porte NOR:

**Algoritmo 6** sintesi a porte NOR**Input:** un circuito in forma PS**Output:** una sintesi a porte NOR

Si sostituisce la porta AND con il suo equivalente a NOR

Si sostituisce ciascun OR con il suo equivalente a NOR

Si eliminano le coppie di NOT interne a cascata

Anche qui ignoriamo i NOT sull'ingresso, per gli stessi motivi di prima, e rimuovendo le coppie NOT interni (creati da coppie di NAND con gli stessi input) ritorniamo nuovamente in una forma a 2 livelli di logica.

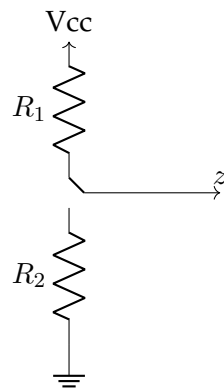
Dal punto di vista algebrico si ha:

$$z = S_1 \cdot S_2 \cdot \dots \cdot S_k = \overline{\overline{S_1 \cdot S_2 \cdot \dots \cdot S_k}} = \overline{\overline{S_1} + \overline{S_2} + \dots + \overline{S_k}}$$

dove il complemento superiore è l'ultima porta NOR (quella che sostituisce l'AND), e i singoli  $S_i$  complementati sono singole porte NOR ( $S_i$  è una somma, quindi  $\overline{S_i}$  è una porta NOR).

**12.2 Porte tri-state**

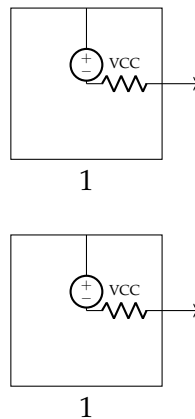
Fa comodo poter connettere insieme le uscite delle reti usando bus condivisi, cioè linee di ingresso-uscita. Abbiamo che l'uscita di una rete, dal punto di vista di una rete, corrisponde a un interruttore fra il Vcc (1 logico) o la massa (0 logico), cioè:



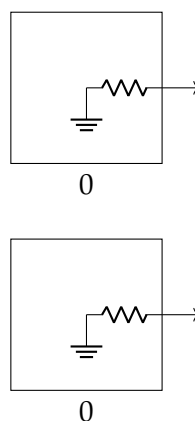
dove  $R_1$  e  $R_0$  sono incognite.

Quando vado a collegare più uscite sulla stessa linea possono crearsi più situazioni:

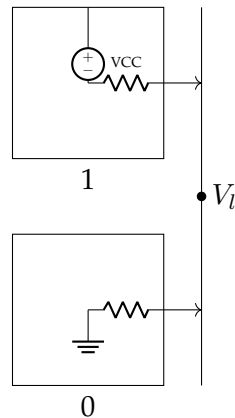
- **1 logici:** se ho due 1 logici, cioè due generatori di potenziale a  $V_{cc}$ , connessi sulla stessa linea, ho che la tensione sulla linea è sempre  $V_{cc}$ , quindi tutto ok:



- **0 logici:** allo stesso tempo, se ho due 0 logici, quindi due collegamenti a massa, sulla linea si avrà tensione nulla:



- **0 e 1 logici:** se collego un 1 logico e uno 0 logico alla stessa linea, ottengo effettivamente un partitore di tensione:

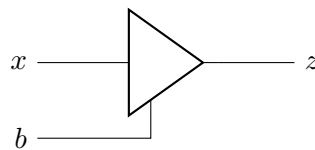


da cui ricavo:

$$V_l = \frac{R_0}{R_1 + R_0}$$

Notiamo soprattutto che se  $R_0$  e  $R_1$  sono molto piccoli, otteniamo correnti  $I$  molto grandi, che significa componenti bruciati.

Per risolvere il problema dato da 0 e 1 logici connessi sulla stessa linea, usiamo specifici apparecchi detti **porte tri-state**, che sono capaci di disconnettere fisicamente un'uscita da una linea condivisa. Si rappresentano come:



dove  $x$  è l'ingresso,  $z$  l'uscita, e  $b$  l'enabler. A  $b = 1$  la porta si comporta come un'elemento neutro, mentre a  $b = 0$  offre un'alta impedenza, effettivamente scollegando l'uscita. La tabella di verità corrispondente sarà:

$b$	$x$	$z$
1	0	0
1	1	1
0	-	Hi-Z

Notiamo che il valore Hi-Z (alta impedenza) non è un valore logico: ciò che esce da una porta in stato Hi-Z viene interpretato come un filo staccato dal resto della rete. Ogni porta logica gestisce poi questa situazione secondo le sue specifiche di realizzazione, restando comunque attaccata sia a Vcc che a massa, e quindi non in uno stato HiZ.

In Verilog, possiamo modellizzare una porta tristate come:

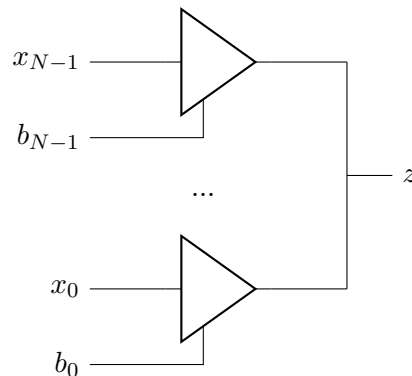
```

1 // una porta tri-state con @x variabile di ingresso, @z variabile di
2 // uscita e @b variabile di controllo attiva alta
3 module tristate(x, b, z);
4     input x, b;
5     output z;
6
7     assign z = b ? x : 'BZ;
8 endmodule

```

### 12.2.1 Multiplexer decodificati

Un componente realizzato attraverso le porte tri-state è il multiplexer decodificato:



Questo componente offre alta indipendenza a tutte le variabili  $x_i$  in ingresso tranne una, quella all'indice  $j$ , selezionata attraverso una variabile di comando  $b_j$ .

In Verilog, possiamo realizzare un multiplexer decodificato sfruttando la definizione già data di decodificatore. Vediamo quindi multiplexer decodificati da 2 a 1:

```

1 // un multiplexer decodificato da 2 a 1 che prende @x1_x0 come
2 // ingresso, @b0 come comando e @z0 come uscita
3 module b2to1_decoded_muxer(x1_x0, b0, z0);
4     inout [1:0] x1_x0;
5     input b0;
6     output z0;
7
8     wire [1:0] s1_s0;
9
10    b1to2_decoder b1to2(b0, s1_s0);
11
12    assign z0 = s1_s0[1] & x1_x0[1] | s1_s0[0] & x1_x0[0];
13    assign x1_x0[1] = s1_s0[1] ? x1_x0[1] : 1'BZ;
14    assign x1_x0[0] = s1_s0[0] ? x1_x0[0] : 1'BZ;
15 endmodule

```

da 4 a 1:

```

1 // un multiplexer decodificato da 4 a 1 che prende @x3_x0 come
2 // ingresso, @b1_b0 come comando e @z0 come uscita
3 module b4to1_decoded_muxer(x3_x0, b1_b0, z0);
4     input [3:0] x3_x0;
5     input [1:0] b1_b0;
6     output z0;
7
8     wire [3:0] s3_s0;
9
10    b2to4_decoder b2to4(b1_b0, s3_s0);
11
12    assign z0 = s3_s0[3] & x3_x0[3] | s3_s0[2] & x3_x0[2]
13              | s3_s0[1] & x3_x0[1] | s3_s0[0] & x3_x0[0];
14    assign x3_x0[3] = s3_s0[3] ? x3_x0[3] : 1'BZ;
15    assign x3_x0[2] = s3_s0[2] ? x3_x0[2] : 1'BZ;
16    assign x3_x0[1] = s3_s0[1] ? x3_x0[1] : 1'BZ;
17    assign x3_x0[0] = s3_s0[0] ? x3_x0[0] : 1'BZ;
18 endmodule

```

da 8 a 1:

```

1 // un multiplexer decodificato da 8 a 1 che prende @x7_x0 come
2 // ingresso, @b2_b0 come comando e @z0 come uscita
3 module b8to1_muxer_d(x7_x0, b2_b0, z0);
4     input [7:0] x7_x0;
5     input [2:0] b2_b0;
6     output z0;
7
8     wire [7:0] s7_s0;
9
10    b3to8_decoder b3to8(b2_b0, s7_s0);
11
12    assign z0 = s7_s0[7] & x7_x0[7] | s7_s0[6] & x7_x0[6]
13              | s7_s0[5] & x7_x0[5] | s7_s0[4] & x7_x0[4]
14              | s7_s0[3] & x7_x0[3] | s7_s0[2] & x7_x0[2]
15              | s7_s0[1] & x7_x0[1] | s7_s0[0] & x7_x0[0];
16    assign x7_x0[7] = s7_s0[7] ? x7_x0[7] : 1'BZ;
17    assign x7_x0[6] = s7_s0[6] ? x7_x0[6] : 1'BZ;
18    assign x7_x0[5] = s7_s0[5] ? x7_x0[5] : 1'BZ;
19    assign x7_x0[4] = s7_s0[4] ? x7_x0[4] : 1'BZ;
20    assign x7_x0[3] = s7_s0[3] ? x7_x0[3] : 1'BZ;
21    assign x7_x0[2] = s7_s0[2] ? x7_x0[2] : 1'BZ;
22    assign x7_x0[1] = s7_s0[1] ? x7_x0[1] : 1'BZ;
23    assign x7_x0[0] = s7_s0[0] ? x7_x0[0] : 1'BZ;
24 endmodule

```

e infine da 16 a 1:

```

1 // un multiplexer decodificato da 16 a 1 che prende @x15_x0 come
2 // ingresso, @b3_b0 come comando e @z0 come uscita
3 module b16to1_muxer_d(x15_x0, b3_b0, z0);
4     input [15:0] x15_x0;
5     input [3:0] b3_b0;
6     output z0;
7
8     wire [15:0] s15_s0;
9
10    b4to16_decoder b3to8(b3_b0, s15_s0);
11
12    assign z0 = s15_s0[15] & x15_x0[15] | s15_s0[14] & x15_x0[14]
13              | s15_s0[13] & x15_x0[13] | s15_s0[12] & x15_x0[12]
14              | s15_s0[11] & x15_x0[11] | s15_s0[10] & x15_x0[10]
15              | s15_s0[9] & x15_x0[9] | s15_s0[8] & x15_x0[8]
16              | s15_s0[7] & x15_x0[7] | s15_s0[6] & x15_x0[6]
17              | s15_s0[5] & x15_x0[5] | s15_s0[4] & x15_x0[4]
18              | s15_s0[3] & x15_x0[3] | s15_s0[2] & x15_x0[2]
19              | s15_s0[1] & x15_x0[1] | s15_s0[0] & x15_x0[0];
20    assign x15_x0[15] = s15_s0[15] ? x15_x0[15] : 1'BZ;
21    assign x15_x0[14] = s15_s0[14] ? x15_x0[14] : 1'BZ;
22    assign x15_x0[13] = s15_s0[13] ? x15_x0[13] : 1'BZ;
23    assign x15_x0[12] = s15_s0[12] ? x15_x0[12] : 1'BZ;
24    assign x15_x0[11] = s15_s0[11] ? x15_x0[11] : 1'BZ;
25    assign x15_x0[10] = s15_s0[10] ? x15_x0[10] : 1'BZ;
26    assign x15_x0[9] = s15_s0[9] ? x15_x0[0] : 1'BZ;
27    assign x15_x0[8] = s15_s0[8] ? x15_x0[0] : 1'BZ;
28    assign x15_x0[7] = s15_s0[7] ? x15_x0[1] : 1'BZ;
29    assign x15_x0[6] = s15_s0[6] ? x15_x0[1] : 1'BZ;
30    assign x15_x0[5] = s15_s0[5] ? x15_x0[1] : 1'BZ;
31    assign x15_x0[4] = s15_s0[4] ? x15_x0[0] : 1'BZ;
32    assign x15_x0[3] = s15_s0[3] ? x15_x0[0] : 1'BZ;
33    assign x15_x0[2] = s15_s0[2] ? x15_x0[1] : 1'BZ;
34    assign x15_x0[1] = s15_s0[1] ? x15_x0[0] : 1'BZ;

```

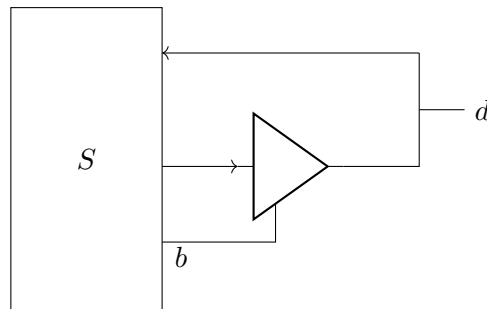
```

35 assign x15_x0[0] = s15_s0[0] ? x15_x0[0] : 1'BZ;
36 endmodule

```

### 12.2.2 Linea di ingresso/uscita

Si usano le porte tri-state per permettere a componenti di comunicare su linee di ingresso/uscita, ad esempio con la memoria. In questo caso, si biforca la linea, ammettendo la linea in entrata così com'è, e mettendo una porta tri-state nella linea in uscita.



Così, quando il componente  $S$  vuole comunicare con l'esterno, imposta l'enabler  $b$  a 1 e mette sulla linea  $d$  ciò che vuole comunicare. Altrimenti tiene  $b$  a 0 e ascolta ciò che arriva su  $d$ .

Se il componente  $S$  comunica con un altro componente  $T$ , questi dovranno impostare alternativamente i loro enabler  $b_S$  e  $b_T$  a 1 e 0, scambiandosi messaggi sulla linea  $d$ .

Una struttura di questo tipo, detta anche "forchetta", può essere realizzata in Verilog come segue:

```

1 // una forchetta sulla variabile bidirezionale @x controllata da @b,
2 // su cui il modulo scrive quanto rileva nel filo @z
3 module tristate(x);
4     inout x;
5     wire z;
6     wire b;
7
8     assign x = b ? z : 1'BZ;
9 endmodule

```

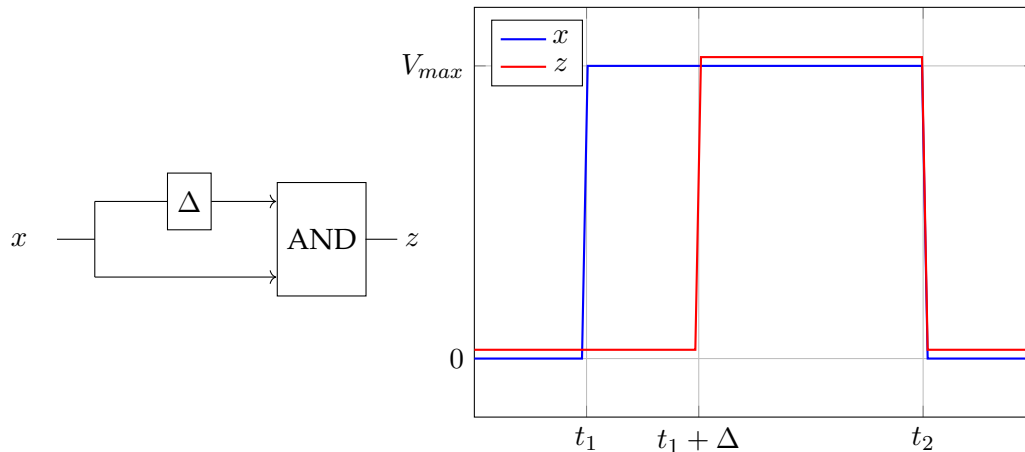
## 12.3 Circuiti di ritardo e formatori di impulso

A volte bisogna trattare di **segnali**. In questo caso si usano gli elementi neutri  $\Delta$  (realizzati spesso con numeri pari di invertitori), che sappiamo porre un **ritardo simmetrico** agli ingressi, dove simmetrico significa identico sulle transizioni  $0 \rightarrow 1$  e  $1 \rightarrow 0$ .

Potrebbe essere utile avere circuiti con ritardi **asimmetrici**, cioè variabili sulle transizioni  $0 \rightarrow 1$  e  $1 \rightarrow 0$ . Indichiamo questi componenti come  $\Delta^+$ .

### 12.3.1 Circuito di ritardo sul fronte di salita

Collegando un neutro  $\Delta$  assieme al segnale stesso ad una porta AND, si ottiene un'andamento del tipo:



Nello specifico, transizionando da  $1 \rightarrow 0$ , si ha che il primo ingresso che va a 0 porta a 0 l'uscita. C'è un ritardo piccolo da parte della porta AND. Quando invece si transiziona da  $0 \rightarrow 1$ , si ha che il secondo ingresso che va a 1 (quello che passa da  $\Delta$ ) porta a 1 l'uscita. C'è un ritardo grande da parte del  $\Delta$  e della porta AND.

La sintesi in Verilog di un circuito di questo tipo è la seguente:

```

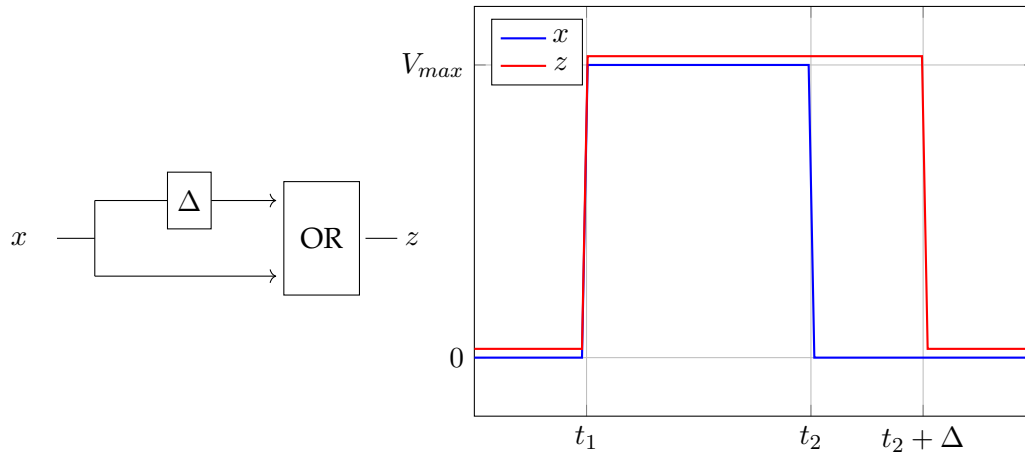
1 // un generatore di ritardo sul fronte di salita con variabile di
2 // ingresso @x e variabile di uscita ritardata @z
3 module posedge_delay_buffer(x, z);
4     input x;
5     output reg z;
6
7     always @(posedge x) begin
8         #5 z = x;
9     end
10    always @(negedge x) begin
11        z = x;
12    end
13 endmodule
14
15 // implementazione a porte logiche
16 module posedge_delay_buffer_p(x, z);
17     input x;
18     output z;
19
20     reg y;
21     initial y = 0;
22
23     always @(posedge x or negedge x) begin
24         #5 y = x;
25     end
26
27     assign z = y & x;
28 endmodule

```

Si nota che questa descrizione (e le seguenti) sono state realizzate in due modi: il primo prevede l'uso di blocchi `always` con statement bloccanti o non bloccanti, mentre il secondo usa la "sintesi a porte logiche" presentata sopra. Entrambe le descrizioni, però, hanno scopo dimostrativo in quanto gli statement di ritardo (`#5`) hanno valenza solo durante la simulazione della rete e non in fase di sintesi.

### 12.3.2 Circuito di ritardo sul fronte di discesa

Allo stesso modo, collegando un neutro  $\Delta$  assieme al segnale stesso ad una porta OR, si ottiene un'andamento del tipo:



Nello specifico, transizionando da  $0 \rightarrow 1$ , si ha che il primo ingresso che va a 1 porta a 1 l'uscita. C'è un ritardo piccolo da parte della porta OR. Quando invece si transiziona da  $1 \rightarrow 0$ , si ha che il secondo ingresso che va a 0 (quello che passa da  $\Delta$ ) porta a 0 l'uscita. C'è un ritardo grande da parte del  $\Delta$  e della porta OR.

La sintesi in Verilog di un circuito di questo tipo è la seguente:

```

1 // un generatore di ritardo sul fronte di discesa con variabile di
2 // ingresso @x e variabile di uscita ritardata @z
3 module negedge_delay_buffer(x, z);
4     input x;
5     output reg z;
6
7     always @(posedge x) begin
8         z <= x;
9     end
10    always @(negedge x) begin
11        #5 z <= x;
12    end
13 endmodule
14
15 // implementazione a porte logiche
16 module negedge_delay_buffer_p(x, z);
17     input x;
18     output z;
19
20     reg y;
21     initial y = 0;
22
23     always @(posedge x or negedge x) begin
24         #5 y = x;
25     end
26
27     assign z = y | x;
28 endmodule

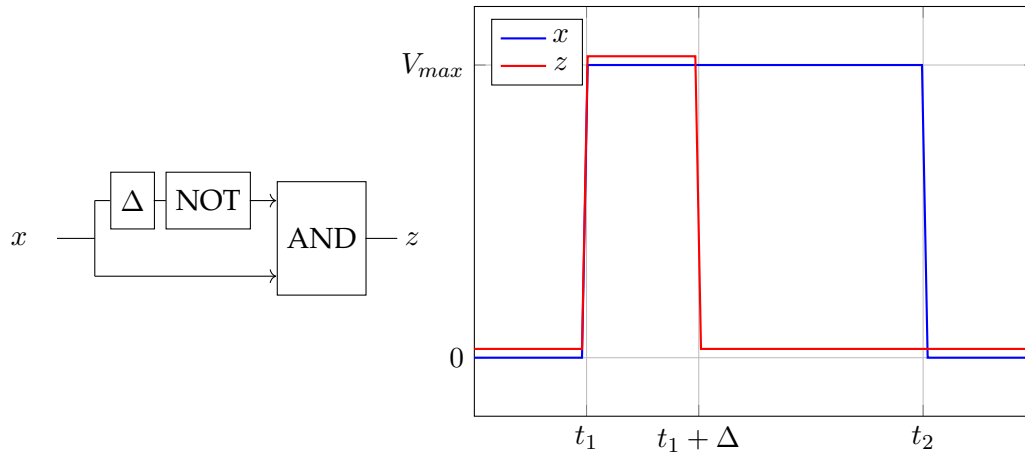
```



### 12.3.3 Formatore di impulso sul fronte di salita

I formatori di impulso sono reti combinatorie che generano in uscita un **impulso** di durata nota. Si indicano con  $P^+$ .

Si crea un formatore di impulso sul fronte di salita collegando la negazione di un  $\Delta$  e il segnale stesso ad una porta AND, cioè:



Nello specifico, transizionando da  $0 \rightarrow 1$ , si ha che il segnale va a 1, attivando la AND (l'ingresso dalla NOT era già attivo). Dopo il ritardo  $\Delta$ , NOT torna a 0, e quindi l'uscita della AND va a 0. Si ha quindi un'impulso di durata del ritardo  $\Delta$ . Transizionando da  $1 \rightarrow 0$ , invece, si ha che il segnale "ancora" istantaneamente l'uscita della AND a zero, ergo non si hanno altri artefatti.

La sintesi in Verilog di un circuito di questo tipo è la seguente:

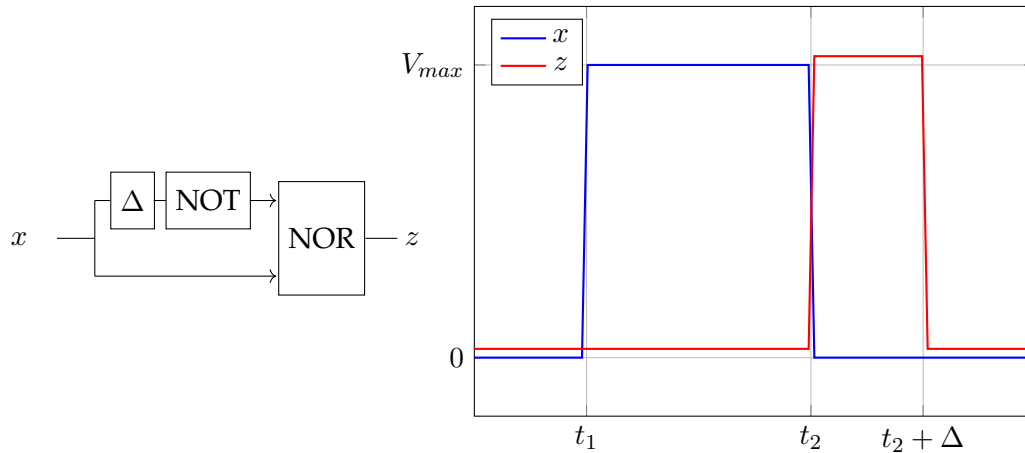
```

1 // generatore di impulso sul fronte di salita con variabile di
2 // ingresso @x e variabile d'impulso @z
3 module posedge_impulse_generator(x, z);
4     input x;
5     output reg z;
6
7     initial z = 0;
8
9     always @(posedge x) begin
10         z = 1;
11         #5;
12         z = 0;
13     end
14 endmodule
15
16 // implementazione a porte logiche
17 module posedge_impulse_generator_p(x, z);
18     input x;
19     output z;
20
21     reg y;
22     initial y = 0;
23
24     always @(posedge x or negedge x) begin
25         #5 y = x;
26     end
27
28     assign z = ~y & x;
29 endmodule

```

### 12.3.4 Formatore di impulso sul fronte di discesa

Si crea un formatore di impulso sul fronte di discesa collegando la negazione di un  $\Delta$  e il segnale stesso ad una porta NOR, cioè:



Nello specifico, transizionando da  $0 \rightarrow 1$ , si ha che il segnale va a 1, ergo la NOR resta a 0 (l'ingresso dalla NOT era già attivo, e due ingressi attivi sono sempre 0 della NOR). Dopo il ritardo  $\Delta$ , l'uscita della NOT torna a 0, così che quando si stacca il segnale, per una durata  $\Delta$  entrambe le linee in entrata alla NOR vanno a 0, e quindi questa va a 1. Dopo il ritardo  $\Delta$ , NOT torna a 0, e quindi l'uscita della AND va a 0. Si ha quindi, ancora una volta, un'impulso di durata del ritardo  $\Delta$ .

La sintesi in Verilog di un circuito di questo tipo è la seguente:

```

1 // generatore di impulso sul fronte di salita con variabile di
2 // ingresso @x e variabile d'impulso @z
3 module negedge_impulse_generator_x(x, z);
4     input x;
5     output reg z;
6
7     always @(negedge x) begin
8         z = 1;
9         #5;
10        z = 0;
11    end
12 endmodule
13
14 // implementazione a porte logiche
15 module negedge_impulse_generator(x, z);
16     input x;
17     output z;
18
19     reg y;
20     initial y = 0;
21
22     always @(posedge x or negedge x) begin
23         #5 y = x;
24     end
25
26     assign z = ~(~y | x);
27 endmodule

```

## 13 Lezione del 17-10-24

### 13.1 Rappresentazione dei numeri naturali

Riprendiamo l'argomento della rappresentazione dei numeri naturali (e in seguito anche degli interi) in modo da sintetizzare reti combinatorie che svolgano operazioni su tali insiemi numerici.

Noi esseri umani rappresentiamo i numeri naturali attraverso una notazione posizionale, ovvero come:

- Un numero  $\beta \geq 2$ , detto **base di rappresentazione**. Nel caso del sistema decimale,  $\beta = 10$ .
- Un'insieme di  $\beta$  simboli, detti **cifre**, a ciascuno dei quali è associato un numero naturale  $\in [0, \beta - 1]$ .
- Una **legge di rappresentazione** che fa corrispondere ad ogni sequenza di cifre un numero naturale.

#### 13.1.1 Notazione posizionale

Dato un numero  $A \in \mathbb{N}$ , lo possiamo rappresentare in base  $\beta$  attraverso una sequenza di cifre:

$$A \equiv (a_{n-1}a_{n-2}\dots a_1a_0)_\beta, \quad 0 \leq a_i \leq \beta - 1, \quad 0 \leq i \leq n - 1$$

dove la legge di rappresentazione è:

$$A = \sum_{i=0}^{n-1} a_i \cdot \beta^i$$

Nella rappresentazione di un numero naturale, una cifra contribuisce a determinare il numero in modo differente a seconda della propria posizione nella sequenza.

Normalmente usiamo il sistema decimale, con  $\beta = 10$ . Nell'informatica, ci interessiamo al sistema binario, con  $\beta = 2$ .

### 13.2 Teoremi della divisione con resto

Dimostriamo due teoremi che formulano effettivamente la divisione sugli insiemi dei naturali e degli interi, e che ci torneranno utili nella conversione fra basi diverse e in generale nell'aritmetica in base  $\beta$ .

#### 13.2.1 Numeri naturali

Nel caso naturale, il teorema è il seguente:

##### **Teorema 13.1: della divisione con resto sui numeri naturali**

Dato  $x \in \mathbb{N}, \beta \in \mathbb{N}, \beta > 0$ , esiste ed è unica la coppia di numeri  $q, r$  con:

- $q \in \mathbb{N}$ ;
- $r \in \mathbb{N}, 0 \leq r < \beta$ ;

tale che  $x = q \cdot \beta + r$ .

$q$  prende il nome di **quoziente** e  $r$  di **resto**.

### Dimostrazione

Per avere una definizione effettiva di  $q$  ed  $r$  pensiamo di partizionare  $\mathbb{N}$  in intervalli:

$$[n \cdot \beta, (n+1) \cdot \beta[, \quad n \in \mathbb{N}$$

Questi partiranno da  $[0, \beta]$ , e così via a coprire tutto  $\mathbb{N}$ :

$$\bigcup_{n \in \mathbb{N}} [n \cdot \beta, (n+1) \cdot \beta[ \equiv \mathbb{N}$$

Potremo quindi usare queste partizioni per definire:

$$\forall x \in \mathbb{N}, \beta \in \mathbb{N}, \quad \exists! q : x \in [q \cdot \beta, (q+1) \cdot \beta[, \quad \exists! r : r = x - q \cdot \beta$$

con le solite ipotesi  $q \in \mathbb{N}$  e  $r \in \mathbb{N}$  con  $0 < r < \beta$ .

Vogliamo dimostrare che (1) questo è sempre possibile e (2)  $q$  e  $r$  sono unici.

1. Questo viene direttamente dal fatto che l'unione di tutte le partizioni dà  $\mathbb{N}$  (come sopra), e per ogni partizione, visto che ciascuna di esse è di dimensione  $\beta$ ,  $r \in [0, \beta[$  basta a coprire tutti i naturali ivi compresi.
2. Assumiamo per assurdo che esistano due possibili rappresentazioni in quoziente e resto, cioè:

$$\exists q_1, r_1 \implies x = q_1 \cdot \beta + r_1$$

$$\exists q_2, r_2 \implies x = q_2 \cdot \beta + r_2$$

Sarà allora vero che:

$$x = x \implies q_1 \cdot \beta + r_1 = q_2 \cdot \beta + r_2, \quad (q_1 - q_2) \cdot \beta = r_2 - r_1$$

e dall'ipotesi  $0 < r < \beta$ :

$$-\beta < (r_2 - r_1) < \beta \implies -\beta < (q_1 - q_2) \cdot \beta < \beta$$

Cioè dividendo per  $\beta$ :

$$-1 < (q_1 - q_2) < 1$$

che in  $\mathbb{N}$  significa  $q_1 - q_2 = 0$ , cioè necessariamente  $q_1 = q_2$ .

□

### 13.2.2 Numeri interi

Il teorema nel caso naturale non è valido così com'è nell'insieme dei numeri interi, in quanto esistono infinite coppie  $q, r \in \mathbb{Z}$  tali per cui  $x = q \cdot \beta + r$ . Si decide di adottare una regola sui segni di  $q$  ed  $r$ , in particolare la seguente:

- $q$  è positivo se  $x$  e  $\beta$  sono concordi in segno, negativo altrimenti. Nel caso di  $\beta \geq 0$ , come sarà per le basi che prenderemo, questo significa che  $q$  è concorde a  $x$ ;
- $r$  è sempre  $\in [0, \beta[$ , e inoltre è concorde a  $x$ .

Queste restrizioni ci permetteranno di riformulare il teorema:

**Teorema 13.2: della divisione con resto sui numeri naturali**

Dato  $x \in \mathbb{Z}, \beta \in \mathbb{Z}, \beta > 0$ , esiste ed è unica la coppia di numeri  $q, r$  con:

- $q \in \mathbb{Z}, \begin{cases} q \geq 0, & \text{se } x \text{ e } \beta \text{ concordi} \\ q < 0, & \text{altrimenti} \end{cases}$
- $r \in \mathbb{N}, 0 \leq |r| < \beta, r \text{ concorde a } x;$
- $|x| = |q| \cdot |\beta| + |r|$

tale che  $x = q \cdot \beta + r$ .

**Dimostrazione**

La dimostrazione del teorema si basa sul mostrare, attraverso l'ausilio della funzione segno:

$$s(x) = \begin{cases} 1, & x \geq 0 \\ -1, & x < 0 \end{cases}$$

che il teorema con le condizioni riportate equivale al caso naturale già dimostrato.

Si ha quindi che:

$$x = q \cdot \beta + r \Rightarrow s(x)|x| = s(q)|q| \cdot s(\beta)|\beta| + s(r)|r|$$

Dalle ipotesi,  $s(q) = s(x) \cdot s(\beta)$  e  $s(r) = s(x)$ , e quindi:

$$s(x)|x| = s(x)s(\beta)|q| \cdot s(\beta)|\beta| + s(x)|r|$$

il quale, notando che  $s(\beta) \cdot s(\beta) = 1$  e dividendo per  $s(x)$ , diventa:

$$|x| = |q| \cdot |\beta| + |r|$$

Ergo, se le ipotesi sono soddisfatte, il teorema si riduce al caso naturale, e inoltre proprio per le ipotesi la rappresentazione quoziente resto è unica.  $\square$

**13.2.3 Interpretazione come divisione**

Notiamo che sia nel caso naturale che nel caso intero il teorema della divisione con resto ricalca la comune divisione naturale e intera con resto, cioè:

$$q = \left\lfloor \frac{x}{\beta} \right\rfloor, \quad r = |x|_{\beta}, \quad x = q \cdot \beta + r = \left\lfloor \frac{x}{\beta} \right\rfloor \cdot \beta + |x|_{\beta}$$

Inoltre se la divisione è tra naturali, anche  $q$  è naturale, cioè la divisione con resto è **chiusa** sui naturali:  $x \in \mathbb{N} \Rightarrow q \in \mathbb{N}$ .

**13.2.4 Proprietà dell'operatore modulo**

Abbiamo usato l'operatore modulo  $(|x|_{\beta})$ . Vediamone alcune proprietà, dato  $\alpha \in \mathbb{N}^+$ :

1.  $|x + k \cdot \alpha|_\alpha = |x|_\alpha, \quad k \in \mathbb{Z}$

Questo da:

$$x = \left\lfloor \frac{x}{\alpha} \right\rfloor \cdot \alpha + |x|_\alpha, \quad x + k \cdot \alpha = \left( \left\lfloor \frac{x}{\alpha} \right\rfloor + k \right) \cdot \alpha + |x|_\alpha$$

chiamiamo  $x' = x + k \cdot \alpha$ :

$$x' = \left\lfloor \frac{x'}{\alpha} \right\rfloor \cdot \alpha + |x'|_\alpha = \left\lfloor \frac{x}{\alpha} + k \right\rfloor \cdot \alpha + |x'|_\alpha = \left( \left\lfloor \frac{x}{\alpha} \right\rfloor + k \right) \cdot \alpha + |x'|_\alpha$$

Dove il passaggio  $\left\lfloor \frac{x}{\alpha} + k \right\rfloor = \left\lfloor \frac{x}{\alpha} \right\rfloor + k$  è concesso da  $k \in \mathbb{Z}$ . Notiamo che le ultime due espressioni ricavate si equivalgono, ergo dev'essere vero che  $|x'|_\alpha = |x|_\alpha$ , da cui la tesi.

2.  $|x + y|_\alpha = ||x|_\alpha + |y|_\alpha|_\alpha$

Questo da:

$$|x + y|_\alpha = \left| \left\lfloor \frac{x}{\alpha} \right\rfloor \cdot \alpha + |x|_\alpha + \left\lfloor \frac{y}{\alpha} \right\rfloor \cdot \alpha + |y|_\alpha \right|_\alpha = \left| \left( \left\lfloor \frac{x}{\alpha} \right\rfloor + \left\lfloor \frac{y}{\alpha} \right\rfloor \right) \cdot \alpha + |x|_\alpha + |y|_\alpha \right|_\alpha$$

e l'applicazione della proprietà (1), da cui la tesi.

3.  $|x \cdot y|_\alpha = ||x|_\alpha \cdot |y|_\alpha|_\alpha$

Questo da:

$$\begin{aligned} |x \cdot y|_\alpha &= \left| \left( \left\lfloor \frac{x}{\alpha} \right\rfloor \cdot \alpha + |x|_\alpha \right) \left( \left\lfloor \frac{y}{\alpha} \right\rfloor \cdot \alpha + |y|_\alpha \right) \right|_\alpha \\ &= \left| \left\lfloor \frac{x}{\alpha} \right\rfloor \left\lfloor \frac{y}{\alpha} \right\rfloor \alpha^2 + \left( \left\lfloor \frac{x}{\alpha} \right\rfloor |y|_\alpha + \left\lfloor \frac{y}{\alpha} \right\rfloor |x|_\alpha \right) \alpha + |x|_\alpha \cdot |y|_\alpha \right|_\alpha \end{aligned}$$

Chiamiamo allora:

$$\left\lfloor \frac{x}{\alpha} \right\rfloor \left\lfloor \frac{y}{\alpha} \right\rfloor \alpha + \left( \left\lfloor \frac{x}{\alpha} \right\rfloor |y|_\alpha + \left\lfloor \frac{y}{\alpha} \right\rfloor |x|_\alpha \right) = k$$

da cui:

$$|x \cdot y|_\alpha = |k \cdot \alpha + |x|_\alpha \cdot |y|_\alpha|_\alpha$$

che ancora applicando la proprietà (1) dà la tesi.

### 13.2.5 Algoritmo delle divisioni successive

Possiamo usare il teorema della divisione con resto iterativamente per trovare la sequenza di cifre che rappresentano  $A$  in base  $\beta$ :

---

#### Algoritmo 7 delle divisioni successive

---

**Input:** un naturale  $A$  e una base  $\beta$

**Output:** la rappresentazione di  $A$  in base  $\beta$  (in ordine inverso)

$i \leftarrow 1$

$q_0 \leftarrow A$

**while**  $q_i \neq 0$  **do**

$q_{i-1} \leftarrow \alpha_{i-1} + \beta \cdot q_i$

$i \leftarrow i + 1$

**end while**

---

Dimostriamo la correttezza dell'algoritmo: si ha che eseguendo i passaggi ricaviamo una forma:

$$A = a_0 + \beta \cdot q_1 = a_0 + \beta \cdot (a_1 + \beta(a_2 + \beta \cdot (...))) = a_0 + a_1 \cdot \beta + a_2 \cdot \beta^2 + \dots$$

e quindi:

$$A = \sum_{i=0}^{n-1} a_i \cdot \beta^i$$

che è per definizione la rappresentazione di  $A$  in base  $\beta$ . Inoltre, il teorema della divisione con resto garantisce che la  $n$ -upla di cifre trovata è **unica**.

Questo algoritmo non è altro che la formalizzazione del DIV-MOD visto in precedenza.

### 13.2.6 Rappresentazione su un numero finito di cifre

Con  $n$  cifre in base  $\beta$ , sappiamo che potremo formulare  $\beta^n$  sequenze differenti quindi rappresentare al massimo il numero  $\beta^n - 1$ , cioè quello dove tutte le cifre hanno valore massimo,  $\beta - 1$ .

Ciò si dimostra da:

$$A = \sum_{i=0}^{n-1} (\beta - 1) \cdot \beta^i = \sum_{i=0}^{n-1} \beta^{i+1} - \sum_{i=0}^{n-1} \beta^i = \sum_{i=1}^n \beta^i - \sum_{i=0}^{n-1} \beta^i = \beta^n - 1$$

Il numero di cifre necessario per rappresentare  $A$  è il numero minimo  $n$  per cui  $\beta^n - 1 \geq A$ , ergo:

$$n = \log_{\beta}(\beta^n) \geq \log_{\beta}(A + 1) \rightarrow n = \lceil \log_{\beta}(A + 1) \rceil$$

## 13.3 Reti combinatorie per i numeri naturali

Vogliamo cosotruire reti logiche che elaborino numeri naturali rappresentati in una data base  $\beta$ , generalmente  $\beta = 2$ . Si useranno **reti combinatorie**, dove lo *stato di uscita* è il **risultato** e lo *stato di ingresso* sono gli **operandi**.

Per ogni operazione aritmetica di base daremo una descrizione **indipendente dalla base**, usando le proprietà della notazione posizionale per scomporre l'operazione in blocchi elementari. In seguito, dettaglieremo le reti logiche che implementano questi blocchi elementari in base 2, attraverso le porte logiche già studiate.

Notiamo che spesso ci concentreremo più sulle **cifre** che sulle codifiche, indipendentemente dalla base.

### 13.3.1 Complemento

Dato  $A = (a_{n-1}a_{n-2}\dots a_1a_0)_{\beta}$ , in base  $\beta$  su  $n$  cifre,  $0 \leq A \leq \beta^n$ , definisco complemento di  $a$  in base  $\beta$  il numero:

$$\bar{A} = \beta^n - 1 - A$$

Si ha che il complemento di un numero a  $n$  cifre sta su  $n$  cifre, e che:

$$\bar{A} = \beta^n - 1 - A = \sum_{i=0}^{n-1} (\beta - 1)\beta^i - \sum_{i=0}^{n-1} \alpha_i \beta^i = \sum_{i=0}^{n-1} (\beta - 1 - \alpha_i) \beta^i$$

$\beta - 1 - \alpha_i$  è una cifra in base  $\beta$  in quanto compresa fra 0 e  $\beta - 1$ . Quindi,  $\bar{A} = (\bar{a}_{n-1}\bar{a}_{n-2}\dots\bar{a}_1\bar{a}_0)_{\beta}$ .

Questo significa che basta saper fare il complemento di una singola cifra per fare il complemento di un numero. In base 2, questo significa usare una porta NOT:

```

1 // un complementatore a una cifra in base 2 con ingresso @x e uscita
2 // @z
3 module b2_complementer(x, z);
4     input x;
5     output z;
6
7     assign z = ~x;
8 endmodule

```

In altre basi diventa più complicato, ad esempio in base 10 con codifica BCD avrò un circuito con 4 ingressi e 4 uscite, con tabella di verità:

$x_3$	$x_2$	$x_1$	$x_0$	$z_3$	$z_2$	$z_1$	$z_0$
0	0	0	0	1	0	0	1
0	0	0	1	1	0	0	0
0	0	1	0	0	1	1	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	0	0
0	1	1	0	0	0	1	1
0	1	1	1	0	0	1	0
1	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	—	—	—	—
1	0	1	1	—	—	—	—
1	1	0	0	—	—	—	—
1	1	0	1	—	—	—	—
1	1	1	0	—	—	—	—
1	1	1	1	—	—	—	—

che dopo una sintesi dà:

$$\begin{cases} z_3 = \overline{x_3 x_2 x_1} \\ z_2 = \overline{x_3 x_2} x_1 + x_2 \overline{x_1} \\ z_1 = x_1 \\ z_0 = \overline{x_0} \end{cases}$$

e il seguente codice Verilog:

```

1 // un complementatore a una cifra in base 10 con ingresso @x e uscita
2 // @z
3 module b10_complementer(x3_x0, z3_z0);
4     input [3:0] x3_x0;
5     output [3:0] z3_z0;
6
7     assign z3_z0 = (x3_x0 == 'B0000) ? 'B1001:
8                   (x3_x0 == 'B0001) ? 'B1000:
9                   (x3_x0 == 'B0010) ? 'B0111:
10                  (x3_x0 == 'B0011) ? 'B0110:
11                  (x3_x0 == 'B0100) ? 'B0101:
12                  (x3_x0 == 'B0101) ? 'B0100:
13                  (x3_x0 == 'B0110) ? 'B0011:

```



```

14         (x3_x0 == 'B0111') ? 'B0010:
15         (x3_x0 == 'B1000') ? 'B0001:
16         (x3_x0 == 'B1001') ? 'B0000:
17         /* don't care */ 'BXXXX;
18 endmodule
19
20 // implementazione a porte logiche
21 module b10_complementer_b(x3_x0, z3_z0);
22     input [3:0] x3_x0;
23     output [3:0] z3_z0;
24
25     assign z3_z0[3] = ~x3_x0[3] & ~x3_x0[2] & ~x3_x0[1];
26     assign z3_z0[2] = ~x3_x0[3] & ~x3_x0[2] & x3_x0[1] | x3_x0[2] & ~x3_x0
27         [1];
28     assign z3_z0[1] = x3_x0[1];
29     assign z3_z0[0] = ~x3_x0[0];
30 endmodule

```

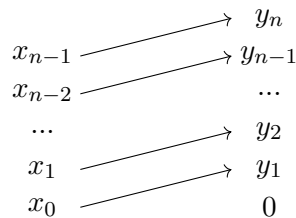
## 14 Lezione del 22-10-24

### 14.1 Operazioni a costo nullo

#### 14.1.1 Moltiplicazioni e divisioni per potenze di base

Moltiplicare e dividere per potenze della base  $\beta$  significa semplicemente aggiungere o togliere zeri, ergo si tratta di operazioni a **costo nullo**. Se le operazioni sono a costo nullo, è molto probabile che le reti che le implementano siano **prive di logica**.

- **Moltiplicazione:** effettivamente, la rete che implementa una moltiplicazione per  $\beta$  sposta gli input  $x_{n-1}, \dots, x_0$  "su", attraverso una mappa:

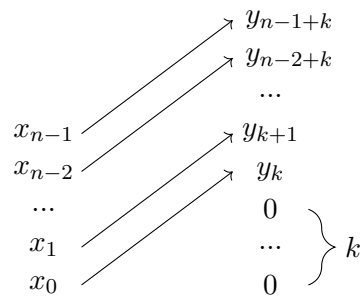


assegnando  $y_0$  ad un generatore di zero.

Per reti che moltiplicano per multipli  $\beta \cdot k$ , generalizzeremo la stessa cosa come:

$$\begin{cases} y_j = x_{j-k}, & k \leq j \leq n-1+k \\ y_j = 0, & 0 \leq j \leq k-1 \end{cases}$$

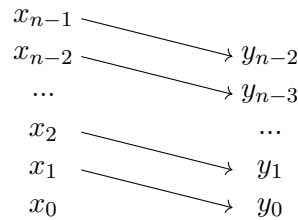
ottenendo quindi la mappa:



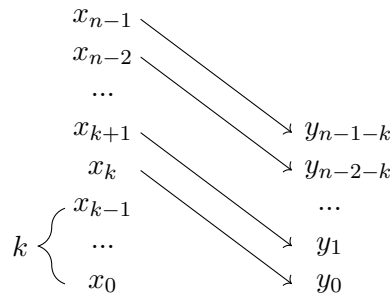
- **Quoziente:** allo stesso modo, si ha che si vogliono spostare gli input "giù", ovvero applicare:

$$\{y_j = x_{j+k}, \quad k \leq j \leq n-1-k\}$$

che per  $k = 1$  è:



e per  $k$  arbitrari è:

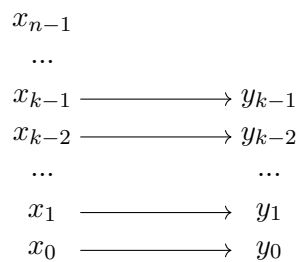


dove i primi  $k$  elementi di  $x$  vengono trascurati (**troncamento**).

- **Resto:** il resto significa semplicemente "tagliare" tutti gli ingressi prima di  $x_k$ , ergo:

$$\{y_j = x_j, \quad 0 \leq j \leq k-1\}$$

secondo la mappa:



### 14.1.2 Concatenamento

Concatenare in  $X$  due numeri  $Y$  e  $Z$  a  $k$  e  $n - k$  cifre significa dire:

$$X = Z \cdot \beta^k + Y$$

Anche questa è un'operazione a complessità nulla, in quanto significa prendere le cifre di  $Y$  e  $Z$ :

$$\begin{cases} x_j = y_j, & 0 \leq j \leq k-1 \\ x_j = z_j, & k \leq j \leq n-1 \end{cases}$$

secondo la mappa:

$$\begin{array}{ccc} z & \left\{ \begin{array}{l} z_{n-k-1} \longrightarrow y_n \\ \dots \\ z_0 \longrightarrow y_k \end{array} \right. & \\ y & \left\{ \begin{array}{l} y_{k-1} \longrightarrow y_{k-1} \\ \dots \\ y_0 \longrightarrow y_0 \end{array} \right. & \end{array}$$

### 14.1.3 Estensione di campo

L'estensione di campo è l'operazione con cui rappresentiamo un naturale su  $n$  cifre su un numero maggiore di cifre. Per i naturali dobbiamo trivialmente aggiungere zero a sinistra della MSD, mentre vedremo che per l'aritmetica intera dovremmo replicare la MSD sulle cifre aggiunte per mantenere il segno corretto.

Abbiamo quindi che per un numero  $x = (x_{n-1}, \dots, x_0)$  su  $n$  cifre vogliamo trovare l'esteso  $x' = (x_{n-1+k}, \dots, x_0)$  su  $n + k$  cifre, cioè il numero tale per cui:

$$\begin{cases} x'_j = x_j, & 0 \leq j \leq n-1 \\ x'_j = 0, & n \leq j \leq n-1+k \end{cases}$$

cioè che rispetta la mappa:

$$\begin{array}{ccc} & \left. \begin{array}{c} 0_{n-1+k} \\ \dots \\ 0_n \end{array} \right\} k & \\ x_{n-1} & \longrightarrow & y_{n-1} \\ x_{n-2} & \longrightarrow & y_{n-2} \\ \dots & & \dots \\ x_1 & \longrightarrow & y_1 \\ x_0 & \longrightarrow & y_0 \end{array}$$

## 14.2 Addizione

La somma, sostanzialmente, consiste nel:

1. Sommare le coppie di cifre di pari posizione, singolarmente, dalla LSD alla MSD e tenendo conto dell'eventuale **riporto entrante**;
2. Se la somma di cifre non è rappresentabile su una singola cifra, usare il **riporto uscente** per la coppia di cifre successive.

Abbiamo che il riporto è sempre  $\in \{0, 1\}$ , e che per la prima coppia di cifre possiamo assumerlo  $= 0$ . Ad ogni passaggio, quindi, applichiamo una funzione:

$$(a_i, b_i, c_{in}) \rightarrow (s_i, c_{out})$$

Inoltre, si ha che l'algoritmo non dipende dalla base  $\beta$ , ma solamente dalla **notazione posizionale**.

#### 14.2.1 Dimensioni di somme

Avevamo quindi che, dati  $X, Y$  in base  $\beta$  su  $n$  cifre, cioè  $X, Y \in [0, \beta^n - 1]$ , con  $C_{in} \in [0, 1]$ , volevamo calcolare:

$$Z = X + Y + C_{in}$$

ovvero trovare il cosiddetto **full adder**. Possiamo dimostrare che il numero di cifre su cui sta il risultato è:

$$0 \leq X + Y + C_{in} \leq 2\beta^n - 1 \leq \beta^{n+1} - 1$$

dove la cifra  $n + 1$  è compresa in  $Z_{n+1} \in [0, 1]$ , cioè rappresenta il riporto uscente di  $X + Y$ .

Possiamo quindi affermare con sicurezza che la somma fra due naturali espressi in base  $\beta$  su  $n$  cifre più un'eventuale riporto entrante  $C_{in}$  produce un naturale che è sempre rappresentabile su  $n + 1$  cifre in base  $\beta$ , delle quali la  $n + 1$ -esima cifra è il riporto uscente, e può valere soltanto 0 o 1.

Quello che vogliamo è un circuito sommatore in base  $\beta$  a  $n$  cifre che prenda le cifre di due naturali  $X$  e  $Y$  su  $n$  cifre e un riporto entrante  $C_{in}$  (un bit), e restituisca un'altro naturale  $Z$ , sempre su  $n$  cifre e un riporto uscente  $C_{out}$  (sempre un bit).

Nel caso uno dei numeri abbia  $m > n$  cifre, si estende il numero su  $n$  cifre fino a  $m$  (aggiungendo  $n - m$  zeri in testa), e poi si somma. Se si vuole poi che la somma sia *sempre* rappresentabile, bisogna usare un sommatore ad  $n + 1$  cifre, ed estendere gli ingressi su  $n + 1$  cifre. In questo caso l'ultimo riporto sarà sempre zero.

#### 14.2.2 Ripple carry e full adder

Creare circuiti per  $2n + 1$  ingressi può essere complicato, quindi si preferisce adottare un approccio **modulare**, dove si scompone ogni somma su una singola coppia di cifre, purchè:

- Le somme vengano eseguite dalla LSD alla MSD;
- Il riporto si **propaghi** (in inglese *ripple*) da una cifra alla successiva.

Chiamiamo quindi ogni sommatore su due cifre (una di  $X$  e una di  $Y$ ) **full adder**, e il montaggio in cui li disponiamo a **ripple carry** (*propagazione dei resti*).

#### 14.2.3 Full adder in base 2

In base 2, un full adder è un circuito con 3 ingressi ( $x_i, y_i$  e  $c_{in}$ ) e 2 uscite ( $s_i$  e  $c_{out}$ ). Abbiamo che la rete dovrebbe avere tabella di verità:

$x_i$	$y_i$	$c_{in}$	$s_i$	$c_{out}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Potremmo adesso applicare Karnaugh, ma notiamo che  $s_i$  vale 1 quando la somma degli ingressi è dispari, cioè si può dire che è lo XOR in cascata di  $x_i$ ,  $y_i$  e  $c_{in}$ . Allo stesso modo, il  $c_{out}$  non sarà altro che un circuito SP standard, che prende gli AND di ogni coppia di ingressi e li passa attraverso un OR. Abbiamo che il full adder è una rete a 2 livelli di logica.

Si ha allora l'implementazione in Verilog del full adder in base 2:

```

1 // un adder in base 2 che calcola @x + @y, mettendo il risultato in
2 // @s e il riporto in @cout
3 module b2_adder(x, y, cin, s, cout);
4     input x, y, cin;
5     output s, cout;
6
7     assign {s, cout} = ({x, y, cin} == 'B000) ? 'B00:
8                       ({x, y, cin} == 'B001) ? 'B10:
9                       ({x, y, cin} == 'B010) ? 'B10:
10                      ({x, y, cin} == 'B011) ? 'B01:
11                      ({x, y, cin} == 'B100) ? 'B10:
12                      ({x, y, cin} == 'B101) ? 'B01:
13                      ({x, y, cin} == 'B110) ? 'B01:
14                      /*({x, y, cin} == 'B111)?*/'B11;
15 endmodule
16
17 // implementazione a porte logiche
18 module b2_adder_p(x, y, cin, s, cout);
19     input x, y, cin;
20     output s, cout;
21
22     wire x_xor_y;
23     assign x_xor_y = x ^ y;
24
25     assign s = (x_xor_y) ^ cin;
26     assign cout = x & y | x_xor_y & cin;
27 endmodule

```

A questo punto si potranno concatenare, ad esempio, 4 full adder per creare un adder a 4 bit:

```

1 // un adder a 4 cifre in base 2 che calcola @x3_x0 + @y3_y0,
2 // mettendo il risultato in @s3_s0 e il riporto in @cout
3 module n4_b2_adder(x3_x0, y3_y0, cin, s3_s0, cout);
4     input [3:0] x3_x0, y3_y0;
5     input cin;
6     output [3:0] s3_s0;
7     output cout;
8
9     wire [2:0] carry;
10

```

```

11  b2_adder add_0 (
12      .x(x3_x0[0]), .y(y3_y0[0]),
13      .cin(cin),
14      .s(s3_s0[0]),
15      .cout(carry[0])
16  );
17
18  b2_adder add_1 (
19      .x(x3_x0[1]), .y(y3_y0[1]),
20      .cin(carry[0]),
21      .s(s3_s0[1]),
22      .cout(carry[1])
23  );
24
25  b2_adder add_2 (
26      .x(x3_x0[2]), .y(y3_y0[2]),
27      .cin(carry[1]),
28      .s(s3_s0[2]),
29      .cout(carry[2])
30  );
31
32  b2_adder add_3 (
33      .x(x3_x0[3]), .y(y3_y0[3]),
34      .cin(carry[2]),
35      .s(s3_s0[3]),
36      .cout(cout)
37  );
38  endmodule

```

#### 14.2.4 Incrementatore

Abbiamo che in assembler potevamo distinguere fra le operazioni **ADD** \$1, %a1 e **INC** %a1. Possiamo fare l'assunzione che almeno uno degli ingressi di un full adder sia sempre zero per realizzare un **half adder** o *incrementatore*: ad esempio, prendiamo  $y_i = 0$ . In questo caso,  $c_{out}$  potrà essere prodotto con un solo livello di logica, cioè attraverso l'AND fra  $x_i$  e  $c_{in}$ . Allo stesso modo, potremo ridurre  $s_i$  ad un solo XOR fra  $x_i$  e  $c_{in}$ . Riportiamo una sintesi a tabella di verità e a porte logiche in Verilog:

```

1  // un half adder in base 2 che calcola @x + @cin, mettendo il
2  // risultato in @s e il riporto in @cout
3  module b2_halfadder(x, cin, s, cout);
4      input x, cin;
5      output s, cout;
6
7      assign {s, cout} = ({x, cin} == 'B00) ? 'B00:
8                      ({x, cin} == 'B01) ? 'B10:
9                      ({x, cin} == 'B10) ? 'B10:
10                     /*({x, cin} == 'B11)?*/'B01;
11  endmodule
12
13  // implementazione a porte logiche
14  module b2_halfadder_p(x, cin, s, cout);
15      input x, cin;
16      output s, cout;
17
18      assign s = x ^ cin;
19      assign cout = x & cin;
20  endmodule

```

Analogamente, potremmo pensare di sintetizzare **incrementatori** (stavolta nel vero senso della parola, e non *half adder*, con cui corrispondevano in base 2 (detto questo, si sono comunque chiamati impropriamente i moduli Verilog `b3_halfadder` e `b10_halfadder` per congruenza con le altre definizioni)) in base 3:

```

1 // un half adder in base 3 che calcola @x1_x0 + @cin, mettendo il
2 // risultato in @s1_s0 e il riporto in @cout
3 module b3_halfadder(x1_x0, cin, s1_s0, cout);
4     input [1:0] x1_x0;
5     input cin;
6     output [1:0] s1_s0;
7     output cout;
8
9     assign {s1_s0, cout} = ({x1_x0, cin} == 'B000) ? 'B000:
10                          ({x1_x0, cin} == 'B001) ? 'B010:
11                          ({x1_x0, cin} == 'B010) ? 'B010:
12                          ({x1_x0, cin} == 'B011) ? 'B100:
13                          ({x1_x0, cin} == 'B100) ? 'B100:
14                          ({x1_x0, cin} == 'B101) ? 'B001:
15                          /*      don't care      */ 'BXXX;
16 endmodule
17
18 // implementazione a porte logiche
19 module b3_halfadder_p(x1_x0, cin, s1_s0, cout);
20     input [1:0] x1_x0;
21     wire x1 = x1_x0[1];
22     wire x0 = x1_x0[0];
23
24     input cin;
25
26     output [1:0] s1_s0;
27     output cout;
28
29     assign s1_s0[1] = (x1 & ~cin) | (x0 & cin);
30     assign s1_s0[0] = (x0 & ~cin) | ~x1 & ~x0 & cin;
31     assign cout = x1 & cin;
32 endmodule

```

e in base 10:

```

1 // un half adder in base 10 che calcola @x3_x0 + @cin, mettendo il
2 // risultato in @s3_s0 e il riporto in @cout
3 module b10_halfadder(x3_x0, cin, s3_s0, cout);
4     input [3:0] x3_x0;
5     input cin;
6     output [3:0] s3_s0;
7     output cout;
8
9     assign {s3_s0, cout} = ({x3_x0, cin} == 'B00000) ? 'B00000:
10                          ({x3_x0, cin} == 'B00001) ? 'B00010:
11                          ({x3_x0, cin} == 'B00010) ? 'B00010:
12                          ({x3_x0, cin} == 'B00011) ? 'B00100:
13                          ({x3_x0, cin} == 'B00100) ? 'B00100:
14                          ({x3_x0, cin} == 'B00101) ? 'B00110:
15                          ({x3_x0, cin} == 'B00110) ? 'B00110:
16                          ({x3_x0, cin} == 'B00111) ? 'B01000:
17                          ({x3_x0, cin} == 'B01000) ? 'B01000:
18                          ({x3_x0, cin} == 'B01001) ? 'B01010:
19                          ({x3_x0, cin} == 'B01010) ? 'B01010:
20                          ({x3_x0, cin} == 'B01011) ? 'B01100:
21                          ({x3_x0, cin} == 'B01100) ? 'B01100:

```

```

22         ({x3_x0, cin} == 'B01101) ? 'B01110:
23         ({x3_x0, cin} == 'B01110) ? 'B01110:
24         ({x3_x0, cin} == 'B01111) ? 'B10000:
25         ({x3_x0, cin} == 'B10000) ? 'B10000:
26         ({x3_x0, cin} == 'B10001) ? 'B10010:
27         ({x3_x0, cin} == 'B10010) ? 'B10010:
28         ({x3_x0, cin} == 'B10011) ? 'B00001:
29         /*      don't care      */ 'BXXXXX;
30 endmodule
31
32 // implementazione a porte logiche
33 module b10_halfadder_p(x3_x0, cin, s3_s0, cout);
34     input [3:0] x3_x0;
35     wire x3 = x3_x0[3];
36     wire x2 = x3_x0[2];
37     wire x1 = x3_x0[1];
38     wire x0 = x3_x0[0];
39
40     input cin;
41
42     output [3:0] s3_s0;
43     output cout;
44
45     assign s3_s0[3] = ~x3 & x2 & x1 & x0 & cin | x3 & ~x0 | x3 & ~cin;
46     assign s3_s0[2] = ~x2 & x1 & x0 & cin | x2 & ~x1 | x2 & ~x0 |
47                     x2 & ~cin;
48     assign s3_s0[1] = ~x3 & ~x1 & x0 & cin | x1 & ~x0 | x1 & ~cin;
49     assign s3_s0[0] = x0 ^ cin;
50     assign cout = x3 & x0 & cin;
51 endmodule

```

Le sintesi SP (con anche gate XOR aggiunti a discrezione) si ricavano direttamente dagli statement `assign` delle descrizioni riportate.

#### 14.2.5 Parallelizzazione della somma

Facciamo delle considerazioni sulle prestazioni: se in un full adder ogni input arriva in tempo  $t$ , dopo 2 livelli di logica il  $c_{in}$  del prossimo full adder arriverà a  $t + 2$ . Quindi il risultato di quel full adder uscirà a  $t + 4$  e così via. Si ha che per  $n$  full adder concatenati, ergo  $n$  cifre, l' $n - 1$ -esima cifra viene computata in tempo  $t + 2n$ . Questo ci dice che la somma è **scomponibile**, ma non **parallelizzabile**.

In verità, negli anni, sono state sviluppate architetture che implementano il **carry lookahead**, cioè implementano su due livelli di logica, con 5 ingressi, un "precalcolo" del carry a qualche  $t + 4$ , cioè ogni due full adder (i 5 ingressi sono il primo carry e i 2 + 2 ingressi dei 2 full adder). Questo pressapoco raddoppia la velocità di calcolo delle somme su  $n$  cifre, passando quindi da  $t + 2n$  a  $\approx t + n$ .

Possiamo ricavare le formule di precalcolo prendendo l'implementazione di un adder data prima. Avevamo che il bit di riporto di un full adder che calcolava  $x + y$  era dato da:

$$C_i = G_i + P_i \cdot C_{in}$$

dove  $G_i = x_i \cdot y_i$  e  $P_i = x_i \oplus y_i$  vengono detti **generazione** e **propagazione** di carry alla cifra  $i$ . Intuitivamente, infatti, questi rappresentano se un riporto viene *generato* o *propagato* alla cifra  $i$ .

Avremo allora le formule estese, facendo le dovute sostituzioni:

$$C_0 = G_0 + P_0 \cdot C_{in}$$



$$C_1 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_{in}$$

$$C_2 = G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot C_{in}$$

$$C_3 = G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot C_{in}$$

Questi riporti verranno propagati ai corrispettivi full adder, che non dovranno più aspettare la propagazione sul lato sinistro da parte dei full adder precedenti. Un esempio di un adder a 4 bit dotato di CLA implementato in Verilog è il seguente:

```

1 // un adder CLA a 4 cifre in base 2 che calcola @x3_x0 + @y3_y0,
2 // mettendo il risultato in @s3_s0 e il riporto in @cout
3 module n4_b2_cla_adder(x3_x0, y3_y0, cin, s3_s0, cout);
4     input [3:0] x3_x0, y3_y0;
5     input cin;
6     output [3:0] s3_s0;
7     output cout;
8
9     wire [2:0] carry;
10
11     n4_b2_cla cla(x3_x0, y3_y0, cin, {cout, carry});
12
13     b2_adder add_0 (
14         .x(x3_x0[0]), .y(y3_y0[0]),
15         .cin(cin),
16         .s(s3_s0[0])
17     );
18
19     b2_adder add_1 (
20         .x(x3_x0[1]), .y(y3_y0[1]),
21         .cin(carry[0]),
22         .s(s3_s0[1])
23     );
24
25     b2_adder add_2 (
26         .x(x3_x0[2]), .y(y3_y0[2]),
27         .cin(carry[1]),
28         .s(s3_s0[2])
29     );
30
31     b2_adder add_3 (
32         .x(x3_x0[3]), .y(y3_y0[3]),
33         .cin(carry[2]),
34         .s(s3_s0[3])
35     );
36 endmodule
37
38 // il modulo che calcola i riporti entranti in @carry
39 module n4_b2_cla(x3_x0, y3_y0, cin, carry);
40     input [3:0] x3_x0, y3_y0;
41     input cin;
42     output [3:0] carry;
43
44     wire [3:0] gen;
45     wire [3:0] pro;
46
47     assign gen = x3_x0 & y3_y0;
48     assign pro = x3_x0 ^ y3_y0;
49
50     assign carry[0] = gen[0]
51                     | pro[0] & cin;
52     assign carry[1] = gen[1]
53                     | pro[1] & gen[0]

```

```

54         | pro[1] & pro[0] & cin;
55     assign carry[2] = gen[2]
56         | pro[2] & gen[1]
57         | pro[2] & pro[1] & gen[0] | pro[2] & pro[1]
58                                     & pro[0] & cin;
59     assign carry[3] = gen[3]
60         | pro[3] & gen[2]
61         | pro[3] & pro[2] & gen[1]
62         | pro[3] & pro[2] & pro[1] & gen[0] | pro[3]
63                                     & pro[2] & pro[1] & pro[0] & cin;
64 endmodule
65
66 // implementazione alternativa dell'adder (non abbiamo piu' bisogno
67 // di cout)
68 module b2_adder(x, y, cin, s);
69     input x, y, cin;
70     output s;
71
72     assign s = (x ^ y) ^ cin;
73 endmodule

```

### 14.3 Sottrazione

L'algoritmo di sottrazione consiste nell'applicare un'algoritmo analogo alla somma ma con prestiti al contrario, cioè nel:

1. Sottrarre le coppie di cifre di pari posizione, singolarmente, dalla LSD alla MSD;
2. Se la somma di cifre non è rappresentabile su una singola cifra, generare un **prestito** (*borrow*) per la coppia di cifre successive.

Si ha anche qui che il prestito è sempre  $\in [0, 1]$ . Inoltre, anche questo algoritmo non dipende dalla base  $\beta$ , ma solo dalla notazione posizionale.

#### 14.3.1 Dimensioni di sottrazioni

Abbiamo quindi due naturali  $X$  e  $Y$  in base  $\beta$  su  $n$  cifre, quindi tali che  $X, Y \in [0, \beta^n - 1]$ , e un bit  $B_{in}$  con  $0 \leq b_{in} \leq 1$ . Voglio calcolare il naturale:

$$Z = X - Y - b_{in}$$

ammesso che questo naturale *esista!*. Questo perché i naturali non sono chiusi rispetto alla sottrazione, cioè:

$$-\beta^n \leq X - Y - b_{in} \leq \beta^n - 1$$

potrei avere  $Z \in \mathbb{Z}$ .

#### 14.3.2 Rappresentabilità

Dico quindi che, dal teorema della divisione con resto, posso scrivere  $Z$  come quoziente e resto di una divisione per  $\beta^n$ :

$$Z = -b_{out} \cdot \beta^n + D = X - Y - b_{in}$$

definito:

$$-b_{out} = \left\lfloor \frac{X - Y - b_{in}}{\beta^n} \right\rfloor, \quad D = |X - Y - b_{in}|_{\beta^n}$$

dove noto che  $b_{out} \in \{0, 1\}$  indipendentemente da  $\beta$  (si comporta come il *carry* della somma).

Posso quindi scrivere  $Y$  come il suo complemento, noto che:

$$Y + \bar{Y} = \beta^n - 1, \quad Y = \beta^n - 1 - \bar{Y}$$

da cui sostituendo:

$$(1 - b_{out}) \cdot \beta^n + D = X + \bar{Y} + (1 - b_{in}) \equiv \bar{b}_{out} \cdot \beta^n + D = X + \bar{Y} + \bar{b}_{in}$$

dove si complementano i bit  $b_{in}$  e  $b_{out}$ . Chiamiamo:

$$\begin{cases} \bar{b}_{out} = c_{out} \\ \bar{b}_{in} = c_{in} \end{cases}$$

Otteniamo che l'equazione finale è sostanzialmente quella di un sommatore:

$$\bar{b}_{out} \cdot \beta^n + D = X + \bar{Y} + \bar{b}_{in} \equiv c_{out} \cdot \beta^n + D = X + \bar{Y} + c_{in}$$

dove la differenza fra  $X$  e  $Y$  meno un prestito entrante, se naturale, può essere ottenuta se sommo  $X$  ad  $\bar{Y}$ , più un'eventuale riporto entrante ottenuto complementando il prestito entrante. Se a questo punto il riporto uscente di  $\bar{b}_{out}$  vale 1, si ha che la differenza è un naturale pari a  $D$ , altrimenti non è rappresentabile.

Vediamo quindi come implementare in Verilog un sottrattore a una cifra, racchiudendo semplicemente un adder a una cifra fra due NOT ai prestiti e un complementatore al secondo ingresso:

```

1 // un sottrattore in base 2 che calcola @x - @y, mettendo il
2 // risultato in @d e il prestito uscente in @bout
3 module b2_subtractor(x, y, bin, d, bout);
4     input x, y, bin;
5     output d, bout;
6
7     wire y_;
8     assign y_ = ~y;
9
10    wire cin;
11    assign cin = ~bin;
12
13    wire cout;
14    assign bout = ~cout;
15
16    b2_adder addr(x, y_, cin, d, cout);
17 endmodule

```

A questo punto avremo più possibilità per l'implementazione di un sottrattore a più cifre. Potremo collegare più sottrattori a una cifra come quello riportato sopra fra di loro, oppure, più efficientemente, sfruttare un adder a 4 cifre (che ci permetterà ad esempio di non riscrivere le formule di precalcolo dei carry se si volesse usare un meccanismo di CLA). Un esempio in Verilog di entrambi i casi è il seguente:

```

1 // un sottrattore a 4 cifre in base 2 che calcola @x3_x0 - @y3_y0,
2 // mettendo il risultato in @d3_d0 e il riporto in @bout
3 module n4_b2_subtractor(x3_x0, y3_y0, bin, d3_d0, bout);
4     input [3:0] x3_x0, y3_y0;
5     input bin;
6     output [3:0] d3_d0;

```

```
7  output bout;
8
9  wire[2:0] borrow;
10
11  b2_subtractor sub_0 (
12    .x(x3_x0[0]), .y(y3_y0[0]),
13    .bin(bin),
14    .d(d3_d0[0]),
15    .bout(borrow[0])
16  );
17
18  b2_subtractor sub_1 (
19    .x(x3_x0[1]), .y(y3_y0[1]),
20    .bin(borrow[0]),
21    .d(d3_d0[1]),
22    .bout(borrow[1])
23  );
24
25  b2_subtractor sub_2 (
26    .x(x3_x0[2]), .y(y3_y0[2]),
27    .bin(borrow[1]),
28    .d(d3_d0[2]),
29    .bout(borrow[2])
30  );
31
32  b2_subtractor sub_3 (
33    .x(x3_x0[3]), .y(y3_y0[3]),
34    .bin(borrow[2]),
35    .d(d3_d0[3]),
36    .bout(bout)
37  );
38  endmodule
39
40  // implementazione con adder a 4 bit
41  module n4_b2_subtractor_a(x3_x0, y3_y0, bin, d3_d0, bout);
42    input[3:0] x3_x0, y3_y0;
43    input bin;
44    output[3:0] d3_d0;
45    output bout;
46
47    wire[3:0] y3_y0_neg;
48    assign y3_y0_neg = ~y3_y0;
49
50    wire cin;
51    assign cin = ~bin;
52
53    wire cout;
54    assign bout = ~cout;
55
56    n4_b2_adder addr (
57      .x3_x0(x3_x0), .y3_y0(y3_y0_neg),
58      .cin(cin),
59      .s3_s0(d3_d0),
60      .cout(cout)
61    );
62  endmodule
```

### 14.3.3 Comparazione di numeri naturali

Dati due **naturali**  $X$  e  $Y$ , si possono usare i sottrattori per comparare i loro valori, cioè per ottenere  $x < Y$ . Per fare ciò, si calcola  $X - Y$  e si guarda il prestito uscente: se  $b_{out} = 1$ , allora  $X < Y$ , altrimenti viceversa.

Per controllare l'uguaglianza, invece, si prende  $b_{out}$  e  $D$ : se  $b_{out} = 1$  (differenza rappresentabile) e  $D = 0$ , allora  $X = Y$ , altrimenti viceversa.

Vediamo un implementazione in Verilog, dove si calcolano i flag `flag_eq`, `flag_lr` (minoranza) e `flag_gr` (maggioranza):

```

1 // un comparatore a 4 bit che confronta @x3_x0 e @y3_y0 e produce i
2 // flag:
3 // - flag_eq: @x3_x0 == @y3_y0
4 // - flag_gr: @x3_x0 > @y3_y0
5 // - flag_lr: @x3_x0 < @y3_y0
6 module n4_b2_comparator(x3_x0, y3_y0, flag_eq, flag_gr, flag_lr);
7     input [3:0] x3_x0, y3_y0;
8     output flag_eq, flag_gr, flag_lr;
9
10    wire [3:0] d3_d0;
11    wire bout;
12
13    n4_b2_subtractor subr(
14        .x3_x0(x3_x0), .y3_y0(y3_y0),
15        .bin('B0),
16        .d3_d0(d3_d0),
17        .bout(bout)
18    );
19
20    assign flag_eq = (d3_d0 == 'B0000) & ~bout;
21    assign flag_gr = ~(d3_d0 == 'B0000) & ~bout;
22    assign flag_lr = bout;
23 endmodule

```

## 14.4 Moltiplicazione

Dati  $X$  e  $C$  naturali in base  $\beta$  su  $n$  cifre, cioè  $X, C \in [0, \beta^n - 1]$ , e  $Y$  naturale in base  $\beta$  su  $m$  cifre, cioè  $Y \in [0, \beta^m - 1]$ , vogliamo calcolare:

$$P = X \cdot Y + C$$

### 14.4.1 Dimensioni di prodotti

Si ha che, da quanto detto prima:

$$P = X \cdot Y + C \leq (\beta^n - 1) \cdot (\beta^m - 1) + (\beta^n - 1) = \beta^m \cdot (\beta^n - 1) < \beta^{n+m} - 1$$

cioè il risultato sta su  $n + m$  cifre.

### 14.4.2 Algoritmo di moltiplicazione

La moltiplicazione fra naturali si effettua come segue:

1. Si moltiplica  $X$  per tutte le cifre di  $Y$ , iterativamente;
2. Moltiplicando, si generano **risultati parziali**, che vengono disposti a partire dalla cifra per cui stiamo moltiplicando, per quanto ci riguarda si tratta di una moltiplicazione per  $\beta^k$ ;

3. I risultati parziali vengono sommati fra di loro con riporto.

Diverse architetture implementano diversi algoritmi di moltiplicazione, ma l'idea fondamentale è quella di creare risultati parziali e sommarli fra di loro. Un modo particolarmente efficiente di fare moltiplicazioni è quello di:

1. Moltiplicare un numero ad  $n$  cifre per un numero ad una sola cifra;
2. Sommare gli  $m$  addendi, opportunamente traslati, per ottenere il risultato finale.

Possiamo sfruttare il fatto che la somma è **associativa**, e che la cifra  $i$ -esima del prodotto, con  $0 \leq i \leq n - 1$ , è determinata univocamente dai prodotti parziali  $j \leq i$ , ergo possiamo sommare i risultati parziali mentre si svolgono le moltiplicazioni. Quest'ultima differenza è la più sostanziale dalla classica moltiplicazione "in colonna" insegnata a scuola.

Si va quindi a definire una rete detta **moltiplicatore con addizionatore**, che:

1. Moltiplica  $X$  per una cifra di  $Y$ , sommando un termine  $C$  inizialmente nullo, che viene poi impostato alle cifre più significative del risultato parziale trovato. La LSD, invece, viene assegnata direttamente alla posizione corrispondente nel risultato finale.
2. Infine, concatena tutti le cifre ottenute come LSD nel risultato finale.

In questo modo possiamo fare solo moltiplicazioni su  $n \times 1$  cifra e somme su due addendi su  $n + 1$  cifre.

#### 14.4.3 Moltiplicatore con addizionatore in base 2

Vediamo quindi come realizzare un moltiplicatore con addizionatore  $n \times 1$  in base 2, cioè un moltiplicatore con addizionatore ad una cifra.

Vorremmo il risultato, piuttosto triviale in  $\beta = 2$ :

$$P_i = y_i \cdot X + C = \begin{cases} (0+) C, & y_i = 0 \\ X + C, & y_i = 1 \end{cases}$$

Possiamo effettuare la selezione su  $y_i$  attraverso quello che è effettivamente un **multiplexer**. Quello che facciamo quindi è collegare un multiplexer fra  $X$  e 0 con variabile di controllo  $y_i$  a un ingresso di un full adder, e  $C$  all'altro ingresso. L'ingresso  $C_{in}$  del full adder varrà 0, mentre l'uscita  $C_{out}$  verrà concatenata alla somma  $S$ .

Abbiamo che in base 2 un multiplexer a due ingressi, con uno di questi negato, è effettivamente una porta AND fra l'ingresso non nullo e la variabile di controllo. Sostituiamo quindi il multiplexer con un AND a  $n$  fra  $X$  e  $y_i$ .

Vediamo allora l'implementazione in Verilog del modulo moltiplicatore con addizionatore in base 2, con ingressi a 4 bit:

```

1 // un moltiplicatore a 4 * 1 cifre che calcola @x3_x0 * @y + @c3_c0
2 // e lo mette in @p4_p0
3 module n4by1_b2_multiplier(x3_x0, y, c3_c0, p4_p0);
4   input [3:0] x3_x0, c3_c0;
5   input y;
6   output [4:0] p4_p0;
7
8   wire [3:0] lhs;
```

```

9
10 assign lhs = x3_x0 & {4{y}};
11
12 n4_b2_adder addr (
13     .x3_x0(lhs), .y3_y0(c3_c0),
14     .cin('B0'),
15     .s3_s0(p4_p0[3:0]),
16     .cout(p4_p0[4])
17 );
18 endmodule

```

Si potrebbe pensare di implementare tale modulo, anziché con la struttura a multiplexer presentata, attraverso singoli moltiplicatori a una cifra concatenati fra di loro (attraverso altri multiplexer, sommatore o logica simile). Si ha però che una struttura del genere diventa velocemente più complessa di usare risultati intermedi a un numero prefissato (da  $n/2$ ,  $n/4$  e così via) di bit.

Combiniamo quindi 4 circuiti moltiplicatori con addizionatori per creare un moltiplicatore completo fra due numeri a 4 bit, con un ingresso di riporto a 4 bit:

```

1 // un moltiplicatore a 4 * 4 cifre che calcola
2 // @x3_x0 * @y3_y0 + @c3_c0 e lo mette in @p7_p0
3 module n4by4_b2_multiplier(x3_x0, y3_y0, c3_c0, p7_p0);
4     input [3:0] x3_x0, y3_y0, c3_c0;
5     output [7:0] p7_p0;
6
7     wire [3:0] par_1;
8     wire [3:0] par_2;
9     wire [3:0] par_3;
10
11     n4by1_b2_multiplier mul_0 (
12         .x3_x0(x3_x0), .y(y3_y0[0]), .c3_c0(c3_c0),
13         .p4_p0({par_1, p7_p0[0]})
14     );
15
16     n4by1_b2_multiplier mul_1 (
17         .x3_x0(x3_x0), .y(y3_y0[1]), .c3_c0(par_1),
18         .p4_p0({par_2, p7_p0[1]})
19     );
20
21     n4by1_b2_multiplier mul_2 (
22         .x3_x0(x3_x0), .y(y3_y0[2]), .c3_c0(par_2),
23         .p4_p0({par_3, p7_p0[2]})
24     );
25
26     n4by1_b2_multiplier mul_3 (
27         .x3_x0(x3_x0), .y(y3_y0[3]), .c3_c0(par_3),
28         .p4_p0(p7_p0[7:3])
29     );
30 endmodule

```

#### 14.4.4 Richiamo all'assembler

Avevamo visto che in assembler la moltiplicazione aveva un solo operando esplicito, mentre l'altro era implicito su AL, AX o EAX. Il risultato veniva poi concatenato in AX, DX\_AX o EDX\_EAX. Questo rispetta la logica vista finora: partendo da fattori su  $n$  e  $m$  bit, con  $n = m$ , si arriva ad un risultato rappresentabile su  $n + m = 2n$  bit, cioè  $8 + 8 = 16$  bit (AL  $\rightarrow$  AX),  $16 + 16 = 32$  bit (AX  $\rightarrow$  DX\_AX) e  $16 + 16 = 32$  bit (EAX  $\rightarrow$  EDX\_EAX).

#### 14.4.5 Convertitori di base

Vediamo come realizzare un convertitore da 2 cifre,  $x_1$  e  $x_0$  in codifica BCD, alla codifica binaria. Due cifre rappresentano al massimo 99, che in binario sta su 7 bit. Ergo vogliamo un circuito con 8 bit di ingresso (4 bit + 4 bit degli ingressi BCD) e 7 bit di uscita. Abbiamo che, banalmente, la conversione si effettua come:

$$y = 10 \cdot x_1 + x_0$$

Questo si può realizzare con un moltiplicatore con addizionatore con  $X = x_1$ ,  $Y = 10$  e  $C = x_0$ . Abbiamo che il risultato è su 8 bit, di cui sappiamo però possiamo ridurre il campo a 7.

Un circuito più efficiente può essere realizzato usando solo somme e shift, infatti abbiamo che:

$$y = 10 \cdot x_1 + x_0 = 8 \cdot x_1 + 2 \cdot x_1 + x_0$$

che appare migliore dal punto di vista della realizzazione in aritmetica binaria (8 e 2 sono  $2^3$  e  $2^1$ ). Abbiamo quindi che possiamo usare i moltiplicatori per  $b^k$ , e ottenere un circuito con lo stesso comportamento.

Per la precisione, prendiamo  $8 \cdot x_1$  e troviamo che si estende fino a 7 bit. Prendiamo poi  $2 \cdot x_1 + x_0$  e vediamo che la somma si rappresenta su 5 bit. Sommando i 7 bit di  $8 \cdot x_1$  ai 5 di  $2 \cdot x_1 + x_0$  abbiamo un risultato sempre su 7 bit.

### 15 Lezione del 23-10-24

#### 15.1 Divisione

Siano dati  $X$ , un naturale in base  $\beta$  su  $n + m$  cifre, detto **dividendo**, con  $0 \leq X \leq \beta^{n+m} - 1$ , e  $Y$ , un naturale in base  $\beta$  su  $m$  cifre, detto **divisore**, con  $0 \leq Y \leq \beta^m - 1$ . Vogliamo calcolare i due numeri  $Q$  ed  $R$  tali che:

$$X = Q \cdot Y + R$$

Abbiamo che, con  $|Y = 0|$ , la divisione non è fattibile, quindi avremo bisogno di un uscita di **non fattibilità** `no_div`.

##### 15.1.1 Dimensioni di resti e quozienti

Assumendo  $Y > 0$ , si ha che  $Q$  sta su  $n + m$  cifre (caso peggiore  $Y = 1$ ), mentre  $R$  sta su  $m$  cifre, in quanto  $0 \leq R \leq Y$  dalle proprietà della divisione. Scelgo, per ragioni tecniche, che il quoziente dovrà stare su  $n$  cifre, quindi impongo  $Q \leq \beta^n - 1$ . Nel caso non si possa rappresentare  $Q$ , quindi, userò sempre la stessa uscita `no_div` di prima.

La decisione fatta riguardo a  $Q$  implica che:

$$X = Q \cdot Y + R \leq (\beta^n - 1) \cdot Y + (Y - 1) = \beta^n \cdot Y - 1 \Rightarrow X < \beta^n \cdot Y$$

L'ipotesi potrebbe sembrare limitante, ma visto che si può ricavare  $n$  che soddisfi la disuguaglianza, possiamo eseguire qualsiasi divisione poste **estensioni** del dividendo e **riserve** di cifre (cioè più delle strettamente necessarie) per il quoziente.

Solo nel caso il numero di cifre  $n$ ,  $m$  sia dato dal problema, cioè quando si lavora su **campi finiti**, l'ipotesi è restrittiva.



L'obiettivo è quello di progettare circuiti che eseguano questa divisione su campi di dimensioni prestabilite: dovremmo ricordare questa proprietà nello sviluppo e del circuito (e noteremo ha un significato specifico), e quando andiamo ad utilizzarlo, cioè quando si scrivono istruzioni assembly che ordinano divisioni fra numeri su registri di dimensione diversa.

### 15.1.2 Modulo divisore

Vogliamo quindi realizzare un circuito che:

1. Verifichi la **fattibilità** della divisione nelle ipotesi date;
2. Se il quoziente sta su  $n$  cifre, lo restituisca, altrimenti restituisca `no_div`.

La divisione viene svolta, tradizionalmente, prendendo un sottoinsieme delle  $n$  cifre più significative del dividendo, tali per cui possiamo trovare quante volta il divisore sta nel sottoinsieme. Visto che non possiamo riscaldare il numero di cifre prese dal divisore una volta assemblato il circuito, abbiamo bisogno di un numero minimo di cifre da prendere ogni volta, per essere sicuri di poter eseguire la divisione. Formalmente, quindi, prendo il minimo numero di cifre più significative di  $X$  per ottenere un  $X' \in [Y, \beta \cdot Y[$ . In questo  $m$  cifre possono non bastare (potremmo avere che le  $m$  cifre più significative di  $X$  sono  $< Y$ ), mentre  $m + 1$  bastano sempre (purché  $X$  non abbia zeri in testa).

Si calcolano quindi i quozienti e i resti **parziali**,  $q$  e  $R'$ , dalla divisione di  $X'$  e  $Y$ . Si ha che  $q$  sta su una sola cifra, perché  $X' < \beta \cdot Y$  dall'ipotesi.

Calcolo quindi il nuovo dividendo  $X'$  concatenando  $R'$  con la cifra più significativa non ancora utilizzata di  $X$ . Il nuovo dividendo, date le ipotesi, è ancora  $< \beta \cdot Y$ :

$$R' \leq Y - 1, \quad \beta \cdot R' + (\beta - 1) \leq \beta \cdot Y - \beta + \beta + 1 = \beta \cdot Y$$

Si itera fino ad esaurimento delle cifre del dividendo. A questo punto il **quoziente** è ottenuto dal concatenamento dei quozienti parziali, e il resto è l'ultimo resto parziale.

Abbiamo che l'unica divisione effettiva è quella di  $m + 1$  per  $m$  cifre, mentre tutte le altre sono effettivamente scomposizioni, quindi circuiti di logica a costo nullo. Resta quindi da calcolare solo il flag di non fattibilità `no_div`: questo deriva naturalmente da quanto avevamo detto riguardo alle dimensioni del dividendo:

$$X < \beta^n \cdot Y$$

Inoltre, vogliamo impostare `no_div` anche nel caso  $Y$  sia nullo, per ovvi motivi.

### 15.1.3 Divisione nei processori Intel x86

Abbiamo visto come nei processori Intel x86, abbiamo a disposizione tre versioni della divisione:

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

Si ha che la **div** ammette dividendo su  $2n$  bit e divisore su  $n$  bit, con  $n = 8, 16, 32$ , e richiede che il quoziente stia su  $n$  bit (altrimenti genera un'eccezione). Questo è quello che si otterrebbe ponendo  $n = m$ .

### 15.1.4 Divisione elementare in base 2

Resta quindi da capire come effettuare la divisione elementare fra un numero a  $m + 1$  cifre e un altro a  $m$  cifre, sotto l'ipotesi  $X \leq 2Y = 2^1 \cdot Y$  (siamo in  $\beta = 2$ ).

Abbiamo che  $Q$  può valere 0 o 1. Vale 0 se il divisore  $Y$  è maggiore del dividendo  $X$ , 1 altrimenti.  $R$ , invece, è uguale al dividendo  $X$  se questo è minore del divisore  $Y$ , altrimenti è uguale a  $X - Y$ :

$$Q = \begin{cases} 0, & X < Y \\ 1, & X \geq Y \end{cases}, \quad R = \begin{cases} X, & X < Y \\ X - Y, & X \geq Y \end{cases}$$

Per rappresentare questo sistema ci serve un comparatore fra  $X$  e  $Y$ . Lo realizziamo con un sottrattore (di cui bisognavamo comunque per il calcolo di  $X - Y$ ), quindi mandando  $Y$  complementato (ed opportunamente esteso) al secondo input di un sommatore, ed  $X$  al primo. Il sommatore ha  $C_{in} = 0$ .

Fuori dal sommatore, avremo  $X - Y$  come risultato, e  $b_{out}$  come discriminante per  $X < Y$ . Mandiamo quindi  $X$  e  $X - Y$  agli ingressi di un multiplexer con variabile di controllo  $C_{out}$  dal sommatore, cioè discriminiamo fra  $X$  e  $X - Y$  sulla base di quanto restituito dal comparatore.

A questo punto si ha che  $b_{out}$  rappresenta  $Q$ , mentre l'uscita del multiplexer è  $R$ .

Vediamo quindi l'implementazione del singolo modulo divisore:

```

1 // un divisore a 3 / 2 cifre che calcola @x2_x0 / @y1_y0, mette il
2 // quoziente in @q e il resto in @r1_r0
3 module n3by2_b2_divider(x2_x0, y1_y0, q, r1_r0);
4     input [2:0] x2_x0;
5     input [1:0] y1_y0;
6     output q;
7     output [1:0] r1_r0;
8
9     wire [2:0] y2_y0;
10    assign y2_y0 = {1'B0, y1_y0};
11
12    wire [2:0] d2_d0;
13    wire bout;
14
15    n3_b2_subtractor cmp (
16        .x2_x0(x2_x0), .y2_y0(y2_y0),
17        .bin('B0),
18        .d2_d0(d2_d0),
19        .bout(bout)
20    );
21
22    // dal n3_b2_comparator (tecnicamente)
23    wire flag_geq;
24    assign flag_geq = ~bout;
25
26    assign q = flag_geq;
27
28    // un multiplexer a due vie
29    assign r1_r0 = flag_geq ? d2_d0[1:0] : x2_x0[1:0];
30 endmodule

```

e come questi si possono combinare a formare un modulo divisore completo a con dividendo a 4 e divisore a 2 cifre binarie:

```

1 // un divisore a 4 / 2 cifre che calcola @x3_x0 / @y1_y0, mette il
2 // quoziente in @q1_q0 e il resto in @r1_r0. no_div rappresenta la

```

```

3 // non fattibilita'
4 module n4by2_b2_divider(x3_x0, y1_y0, q1_q0, r1_r0, no_div);
5     input [3:0] x3_x0;
6     input [1:0] y1_y0;
7     output [1:0] q1_q0;
8     output [1:0] r1_r0;
9     output no_div;
10
11     wire [1:0] res;
12
13     feasibility_checker fes (
14         .x3_x0(x3_x0), .y1_y0(y1_y0),
15         .no_div(no_div)
16     );
17
18     n3by2_b2_divider div_0 (
19         .x2_x0(x3_x0[3:1]), .y1_y0(y1_y0),
20         .q(q1_q0[1]), .r1_r0(res)
21     );
22
23     n3by2_b2_divider div_1 (
24         .x2_x0({res, x3_x0[0]}), .y1_y0(y1_y0),
25         .q(q1_q0[0]), .r1_r0(r1_r0)
26     );
27 endmodule
28
29 // modulo che controlla la fattibilita' della divisione
30 module feasibility_checker(x3_x0, y1_y0, no_div);
31     input [3:0] x3_x0;
32     input [1:0] y1_y0;
33     output no_div;
34
35     wire [3:0] y1_y0_by4;
36     assign y1_y0_by4 = {y1_y0, 2'b1};
37
38     wire flag_lr;
39
40     n4_b2_comparator cmp (
41         .x3_x0(x3_x0), .y3_y0(y1_y0_by4),
42         .flag_lr(flag_lr)
43     );
44
45     assign no_div = ~(flag_lr & |y1_y0);
46 endmodule

```

dove si nota che il modulo `n3_b2_subtractor` è un sottrattore a 3 cifre realizzato analogamente a quanto mostrato alla lezione precedente (in ogni caso, un'implementazione è reperibile nella cartella `/verilog` della repository contenente gli appunti).

## 16 Lezione del 24-10-24

### 16.0.1 Rappresentazione dei numeri interi

Vogliamo rappresentare numeri interi su  $n$  cifre. Finora avevamo definito la legge di rappresentazione:

$$A = \sum_{i=0}^{n-1} a_i \cdot \beta^i$$

per i numeri naturali.

Nel sistema decimale, usiamo solutamente la rappresentazione **modulo e segno**, cioè usiamo un segno prefisso (+ o -), e poi indichiamo il modulo come un naturale.

In base 2, invece, decidiamo di sfruttare la seguente proprietà: preso un insieme di  $\beta^n$  numeri interi, si può sempre trovare una legge biunivoca che gli fa corrispondere un insieme di  $\beta^n$  numeri naturali.

Definiamo quindi una legge  $L : \mathbb{Z} \rightarrow \mathbb{N}$ , e chiamiamo  $A$  un numero naturale (che indicheremo in Maiuscolo da qui in avanti) e  $a$  un numero intero (che indicheremo in minuscolo da qui in avanti). Si ha che:

$$A = L(a) \Leftrightarrow a = L^{-1}(A)$$

e cioè:

$$a \leftrightarrow^L A \equiv (a_{n-1}a_{n-2}\dots a_1a_0)_\beta$$

che significa che con la stessa sequenza di cifre indichiamo sia un naturale che il corrispondente intero. Notiamo che la  $a$  minuscola qui significa *cifra*, che è indifferentemente di  $A$  o  $a$  (numeri naturali e interi).

Scegliendo leggi  $L$  valide possiamo ottenere dei significativi vantaggi implementativi: ad esempio potremmo definire una legge che permette di usare la stessa circuiteria per le operazioni aritmetiche sia sui naturali che sugli interi.

Il **dominio** di  $L$  dovrà essere contiguo, cioè un'intervallo, magari il più simmetrico possibile rispetto allo zero. Questo è possibile solo se  $\beta$  è dispari. Nel caso di  $\beta$  pari, come sarà nel nostro caso di interesse  $\beta = 2$ , dovremmo prendere un numero "in più" a destra o a sinistra. Nel sistema adottato (sarà il complemento a 2) prendiamo il numero a sinistra, cioè quello negativo, ergo avremo l'**intervallo di rappresentabilità**:

$$\left[ -\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1 \right]$$

Notiamo che da qui in poi assumeremo di lavorare in  $\beta$  pari, in quanto in caso contrario dovremmo usare la seguente notazione:

$$\left\lfloor -\frac{\beta^n}{2} \right\rfloor, \left\lceil -\frac{\beta^n}{2} \right\rceil, \dots$$

che appesantirebbe la trattazione, cosa inutile in quanto abbiamo stabilito che il nostro interesse finale è trovare metodi che si applichino a  $\beta = 2$ .

Abbiamo infatti che, posto  $\beta = 2$ , l'intervallo riportato precedentemente rappresenta quello a cui siamo abituati per la rappresentabilità dei numeri interi in complemento a 2:

$$\left[ -2^{n-1}, 2^{n-1} - 1 \right]$$

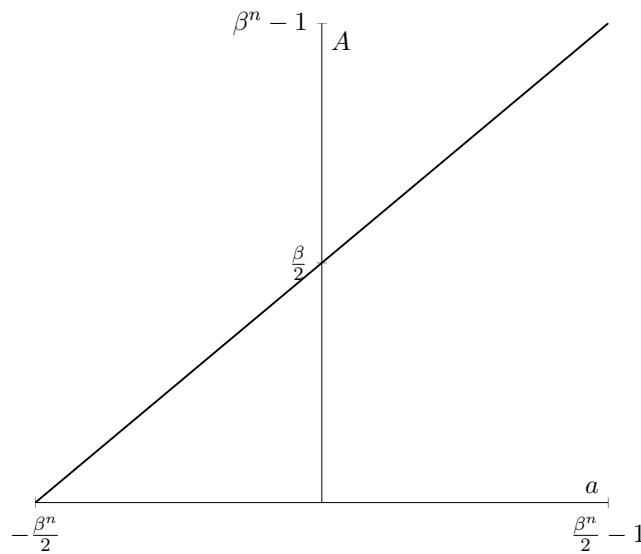
## 16.0.2 Rappresentazione in traslazione

Una possibile legge di rappresentazione è data da:

$$L(a) = A = a + \frac{\beta^n}{2}$$

chiamiamo  $\frac{\beta^n}{2}$  **fattore di polarizzazione**. Questa rappresentazione è utile, **monotona** ( $a < b \Leftrightarrow A < B$ ), e viene usata nei convertitori analogico/digitale e digitale/analogico, dove viene chiamata *binario bipolare*. Inoltre, ricordiamo che rappresenta l'esponente nei numeri reali in virgola mobile secondo lo standard IEEE 754.

Su un grafico dove le ordinate rappresentano  $A$  e le ascisse  $a$ , abbiamo la mappa:



### 16.0.3 Modulo e segno

Possiamo sempre usare la legge di rappresentazione in modulo e segno, cioè  $(s, M) \leftrightarrow a$ :

$$s = \begin{cases} 0, & a \geq 0 \\ 1, & a < 0 \end{cases}$$

$$M = \text{abs}(a)$$

Notiamo che noi intendiamo, per modulo e segno, una rappresentazione dove  $n$  bit rappresentano il modulo, e l' $n$ -esimo bit rappresenta il segno, per un totale di  $n + 1$  bit. Non stiamo quindi mettendo in relazione intervalli di naturali con intervalli di interi, ma intervalli di naturali complementati da un bit in più di segno, con intervalli di interi. Abbiamo quindi che questo tipo di rappresentazione non ricade nella categoria definita prima. Ricordiamo comunque che viene applicata per rappresentare il segno dei numeri reali in virgola mobile secondo lo standard IEEE 754.

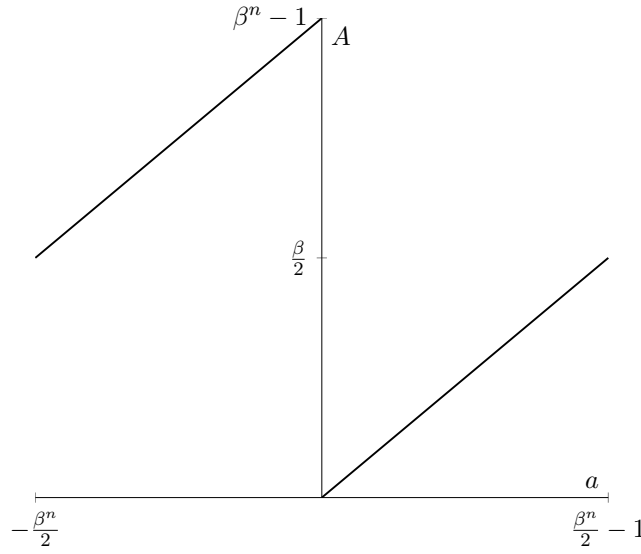
## 16.1 Complemento alla radice

Definiamo la legge, che abbiamo solitamente chiamato **complemento a 2** nel caso  $\beta = 2$ , e che prende il nome di **complemento alla radice** nel caso  $\beta$  arbitrario:

$$L(a) = A = \begin{cases} a, & 0 \leq a < \frac{\beta^n}{2} \\ \beta^n + a, & -\frac{\beta^n}{2} \leq a < 0 \end{cases}$$

Con questa legge perdiamo la **monotoneità**. Ciò nonostante, è la legge usata di norma dal processore, in quanto comporta semplificazioni significative alla circuiteria predisposta all'aritmetica del calcolatore (in particolare permette di usare la stessa circuiteria per somme e sottrazioni naturali e intere).

Ricordiamo che graficamente si può rendere la legge di rappresentazione con il cosiddetto *diagramma a farfalla*:



### 16.1.1 Determinazione del segno

Possiamo determinare il segno di un numero intero  $a$  dalle cifre della sua rappresentazione  $A$ :

$$\begin{cases} a \geq 0 \Leftrightarrow 0 \leq A < \frac{\beta^n}{2} \\ a < 0 \Leftrightarrow \frac{\beta^n}{2} \leq A < \beta^n \end{cases}$$

Facciamo le solite considerazioni:

- Il **massimo numero rappresentabile** è  $\frac{\beta^n}{2} - 1$ , che in CR ha rappresentazione  $(\frac{\beta}{2} - 1, \beta, \dots, \beta)_\beta$ ;
- Il **minimo numero rappresentabile** è  $\frac{\beta^n}{2}$ , che in CR ha rappresentazione  $(\frac{\beta}{2}, 0, \dots, 0)_\beta$ ;
- Lo **0** coincide in  $A$  e  $a$ , ergo vale 0 e ha rappresentazione in CR  $(0, \dots, 0)_\beta$ ;
- Il **-1** è  $\beta^n - 1$ , che in CR ha rappresentazione  $(\beta, \dots, \beta)_\beta$ .

Quindi, per capire se la rappresentazione  $A$  è un numero naturale maggiore o minore di  $\frac{\beta^n}{2}$ , che equivale a capire se l'intero che rappresenta è maggiore o minore di zero, basta guardare la cifra più significativa:

$$\begin{cases} a_{n-1} < \frac{\beta}{2} \Leftrightarrow 0 \leq A < \frac{\beta^n}{2} \\ a_{n-1} \geq \frac{\beta}{2} \Leftrightarrow \frac{\beta^n}{2} \leq A < \beta^n \end{cases}$$

### 16.1.2 Legge inversa del CR

Possiamo ottenere per sostituzione la legge inversa della legge di rappresentazione CR:

$$L(a) = A = \begin{cases} a, & 0 \leq a < \frac{\beta^n}{2} \\ \beta^n + a, & -\frac{\beta^n}{2} \leq a < 0 \end{cases}, \quad L^{-1}(A) = a = \begin{cases} A, & 0 \leq A < \frac{\beta^n}{2} \\ A - \beta^n, & \frac{\beta^n}{2} \leq A < \beta^n \end{cases}$$

Che possiamo riscrivere nel modo più elegante:

$$L^{-1}(A) = a = \begin{cases} A, & a_{n-1} < \frac{\beta}{2} \\ -(\bar{A} + 1), & a_{n-1} \geq \frac{\beta}{2} \end{cases}$$

usando quanto detto sulla MSD e quanto conoscevamo sui complementi (ancora, è sostanzialmente un complemento a 2).

In particolare, il cambio delle disequazioni viene fatto note le proprietà sulla MSD. La trasformazione  $A - \beta^n = -(\bar{A} + 1)$  si ricava invece dalla proprietà fondamentale:

$$A + \bar{A} = \beta^n - 1$$

con semplici passaggi algebrici.

### 16.1.3 Forma alternativa del CR

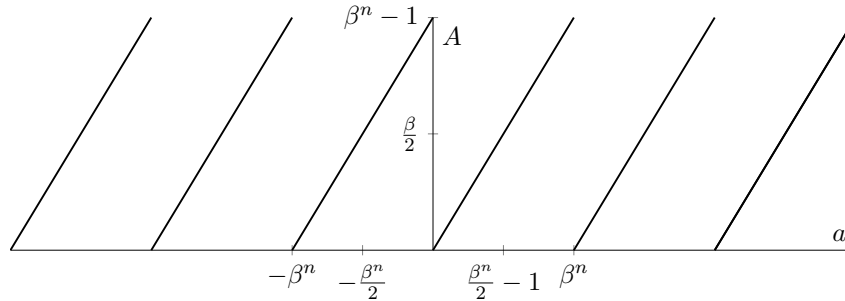
Possiamo usare la forma più concisa (ma anche più pericolosa) delle legge di rappresentazione CR:

$$L(a) = A = |a|_{\beta^n}, \quad \text{se } -\frac{\beta^n}{2} \leq a < \frac{\beta^n}{2} - 1$$

Abbiamo che:

1. Se  $a \geq 0$ , allora è anche  $< \beta^n$ , quindi  $A = a = |a|_{\beta^n}$ ;
2. Se  $a < 0$ , allora è compreso in  $[-\frac{\beta^n}{2}, 0]$ , quindi diviso  $\beta^n$  dà quoziente  $-1$ , cioè dal teorema della divisione con resto,  $a = -1 \cdot \beta^n + |a|_{\beta^n}$ . Avevamo dalla legge di rappresentazione in complemento a radice che, sotto questa ipotesi, volevamo esattamente  $\beta^n + a$ , ergo si ottiene ugualmente  $A = |a|_{\beta^n}$ .

Possiamo quindi interpretare il complemento a radice come una rappresentazione modulare:



Occorre fare attenzione in quanto questo è vero solo nel caso  $a$  sia **rappresentabile**, cioè se rispetta:  $-\frac{\beta^n}{2} \leq a < \frac{\beta^n}{2} - 1$ . In caso contrario, come è chiaro dal grafico, si potrebbero avere le stesse rappresentazioni per interi diversi fra di loro (cioè essenzialmente un overflow).

Vediamo quindi come sintetizzare reti che lavorano sui numeri interi. Vogliamo progettare circuiti che lavorano sulle rappresentazioni, come avevamo fatto per i naturali. Ricordiamo che la rappresentazione vale per un naturale  $A$  o un intero  $a$  a seconda di quanto deciso dal programmatore, e nient'altro.

## 16.2 Valore assoluto

Vogliamo trovare il valore assoluto di un numero intero  $B = \text{abs}(a)$ , con:

$$a \in \left[-\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1\right] \Rightarrow B \in \left[0, \frac{\beta^n}{2}\right]$$

Si ha che  $B$  è un numero naturale rappresentabile su  $n$  cifre. Sappiamo che:

$$\text{abs}(a) = \begin{cases} a, & a \geq 0 \\ -a, & a < 0 \end{cases}$$

Posso quindi ottenere  $\text{abs}(a)$  complementando la rappresentazione nel range di valori che so essere negativo (cioè quando  $a_{n-1} \geq \frac{\beta}{2}$ ):

$$B = \text{abs}(a) = \begin{cases} A, & a_{n-1} < \frac{\beta}{2} \\ \overline{A} + 1, & a_{n-1} \geq \frac{\beta}{2} \end{cases}$$

Logicamente, questo sarà rappresentato da un multiplexer che discrimina fra  $A$  e  $\overline{A} + 1$ . La variabile di comando sarà data da  $b_{out}$  di un comparatore fra  $a_{n-1}$  e  $\frac{\beta}{2}$ .

In base 2 questo è notevolmente più semplice: avrò che basta prendere  $a_{n-1}$  come variabile di comando. Si potrà quindi complementare con uno XOR fra le cifre di  $A$  e  $a_{n-1}$ , e usare  $a_{n-1}$  anche come ingresso di un incrementatore, con le cifre di  $A$  all'altro ingresso.

Vediamo quindi entrambe le implementazioni in base 2, dove il comparatore si è ridotto a un controllo sulla LSD:

```

1 // un modulo che calcola il valore assoluto di un numero intero
2 // @x3_x0 e lo mette in @z3_z0
3 module n4_c2_abs(x3_x0, z3_z0);
4     input [3:0] x3_x0;
5     output [3:0] z3_z0;
6
7     wire [3:0] x3_x0_neg;
8     assign x3_x0_neg = ~x3_x0;
9
10    wire [3:0] s3_s0;
11
12    n4_b2_incrementer inc (
13        .x3_x0(x3_x0_neg), .cin('B1),
14        .s3_s0(s3_s0)
15    );
16
17    assign z3_z0 = (x3_x0[3] == 'B1) ? s3_s0 : x3_x0;
18 endmodule
19
20 // implementazione a porte XOR
21 module n4_c2_abs_x(x3_x0, z3_z0);
22     input [3:0] x3_x0;
23     output [3:0] z3_z0;
24
25     wire [3:0] x3_x0_xor;
26     assign x3_x0_xor = x3_x0 ^ {4{x3_x0[3]}};
27
28     wire [3:0] z3_z0;
29
30    n4_b2_incrementer inc (
31        .x3_x0(x3_x0_xor), .cin(x3_x0[3]),
32        .s3_s0(z3_z0)
33    );
34 endmodule

```



### 16.3 Conversione da CR a MS

Vediamo come convertire un numero in CR nella rappresentazione modulo e segno. Prima di tutto notiamo una discrepanza nell'intervallo di rappresentabilità:

$$i_{MS} = [-\beta^n + 1, \beta^n - 1] \not\supset i_{CR} = \left[-\frac{\beta^n}{2}, \frac{\beta^n}{2}\right]$$

Abbiamo però che  $i_{CR} \subset i_{MS}$ , tolto il bit di segno, quindi l'operazione è sempre fattibile, calcolando l'assoluto e stabilendo:

$$\text{sgn}(a) = \begin{cases} 0, & a_{n-1} < \frac{\beta}{2} \Leftrightarrow a_{n-1} = 0 \\ 1, & a_{n-1} \geq \frac{\beta}{2} \Leftrightarrow a_{n-1} = 1 \end{cases}$$

dove si è riportato il valore di  $a_{n-1}$  in base  $\beta = 2$ .

In Verilog, quindi, si usa il modulo valore assoluto sintetizzato prima:

```

1 // un convertitore da complemento a 2 a modulo e segno di un intero
2 // su 4 bit @x3_x0 che mette la rappresentazione in @z3_z0 e @sgn
3 module n4_c2_ms_converter(x3_x0, z3_z0, sgn);
4     input [3:0] x3_x0;
5     output [3:0] z3_z0;
6     output sgn;
7
8     n4_c2_abs abs (
9         .x3_x0(x3_x0), .z3_z0(z3_z0)
10    );
11
12    assign sgn = x3_x0[3];
13 endmodule

```

### 16.4 Calcolo dell'opposto

Vediamo come trovare l'opposto di un numero in CR, quindi dato  $A \leftrightarrow a$ ,  $B \leftrightarrow b$  tale che  $b = -a$ . Questo' operazione non è sempre possibile, a causa dell'asimmetria dell'intervallo di rappresentabilità in CR  $\left[-\frac{\beta^n}{2}, \frac{\beta^n}{2}\right]$ : avremo che il numero in  $-\frac{\beta^n}{2}$ , negativo, non ha opposto positivo rappresentabile.

Avremo quindi bisogno di un flag di overflow, diciamo  $ow$ . Le due uscite, l'opposto e  $ow$ , andranno quindi calcolate separatamente. Assumendo  $ow = 0$ , si ha:

$$\begin{aligned} B &= |-a|_{\beta^n} = |-1|_{\beta^n} \cdot |a|_{\beta^n}|_{\beta^n} = |(\beta^n - 1) \cdot A|_{\beta^n} \\ &= |\beta^n \cdot A - A|_{\beta^n} = |-A|_{\beta^n} = |-\beta^n + 1 + \bar{A}|_{\beta^n} = |1 + \bar{A}|_{\beta^n} \end{aligned}$$

cioè si ritrova sostanzialmente la legge di rappresentazione inversa  $L^{-1}$ .

Sappiamo di poter implementare questa legge con una negazione di tutte le cifre, seguita da un incremento di 1. Nel caso precedente, avevamo usato lo XOR in quanto volevamo che la negazione fosse condizionale (pilotata dal bit di segno). In questo caso vogliamo negare sempre, quindi basta una porta NOT.

L' $ow$  viene invece impostato sulla base di un AND fra le cifre più significative  $a_{n-1}$  del numero non negato e  $b_{n-1}$  del numero negato. In Verilog:

```

1 // un negatore a 4 cifre in complemento a 2 @x3_x0 che mette il
2 // risultato in @z3_z0. ow il flag di overflow
3 module n4_c2_negator(x3_x0, z3_z0, ow);
4     input [3:0] x3_x0;

```

```

5  output [3:0] z3_z0;
6  output ow;
7
8  wire [3:0] x3_x0_neg;
9  assign x3_x0_neg = ~x3_x0;
10
11 wire [3:0] s3_s0;
12
13 n4_b2_incrementer inc (
14     .x3_x0(x3_x0_neg), .cin('B1),
15     .s3_s0(s3_s0)
16 );
17
18 assign z3_z0 = s3_s0;
19 assign ow = x3_x0[3] & s3_s0[3];
20 endmodule

```

### 16.4.1 Richiamo all'assembly

Si ricorda che in assembly avevamo l'istruzione **NEG**, che interpretava una sequenza di bit come un numero intero, e ne calcolava l'opposto se possibile, impostando il flag OF altrimenti.

## 16.5 Estensione di campo per gli interi

Avevamo detto che l'estensione di campo per gli interi richiedeva logica. Possiamo infatti ricavare, per via algebrica il valore dell'intero con l' $n$ -esima cifra aggiunta,  $A_{EST}$ :

$$A_{EST} = \begin{cases} a, & 0 \leq a < \frac{\beta^n}{2} \\ \beta^{n+1} + a, & -\frac{\beta^n}{2} \leq a < 0 \end{cases} = \begin{cases} A, & 0 \leq a < \frac{\beta^n}{2} \\ \beta^n \cdot (\beta - 1) + a, & -\frac{\beta^n}{2} \leq a < 0 \end{cases}$$

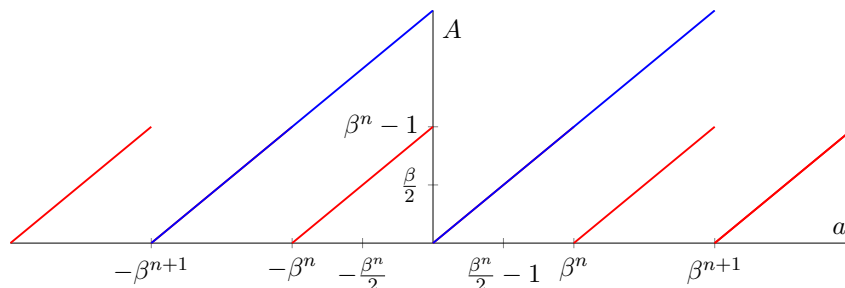
da cui troviamo:

$$= \begin{cases} 0 \cdot \beta^n + A, & 0 \leq a < \frac{\beta^n}{2} \\ \beta^n \cdot (\beta - 1) + a, & -\frac{\beta^n}{2} \leq a < 0 \end{cases}$$

Notiamo che i termini che moltiplicano  $\beta^n$  sono quelli della cifra che vogliamo aggiungere. Possiamo quindi definire la cifra aggiunta:

$$a_n = \begin{cases} 0, & a_{n-1} < \frac{\beta^n}{2} \\ \beta - 1, & a_{n-1} \geq \frac{\beta^n}{2} \end{cases}$$

Graficamente, possiamo pensare all'estensione di campo come una traslazione verso l'alto del lato sinistro dell'intervallo di ordinate, cioè quello che rappresenta gli interi negativi. Più propriamente, se avevamo dato la rappresentazione  $|a|_{\beta^n}$  per  $A$ , adesso dobbiamo prendere  $|a|_{\beta^{n+1}}$  che graficamente dà:



da cui si nota ancora meglio che il lato per interi positivi resta tale, mentre il lato negativo trasla in alto (in rosso si ha  $|a|_{\beta^n}$ , e in blu  $|a|_{\beta^{n+1}}$ ).

Dal punto di vista della base 2, questo tipo di estensore può essere realizzato semplicemente replicando la  $n - 1$ -esima cifra.

In Verilog, si ha:

```

1 // un estensore di campo per interi in codifica binaria con LSD @x,
2 // che mette in @x_est la nuova LSD
3 module b2_field_extensor(x, x_est);
4     input x;
5     output x_est;
6
7     assign x = x_est;
8 endmodule

```

Possiamo vedere anche l'implementazione per la base 10 in codifica BCD:

```

1 // un estensore di campo per interi in codifica BCD con LSD @x3_x0,
2 // che mette in @x3_x0_est la nuova LSD
3 module b10_field_extensor(x3_x0, x3_x0_est);
4     input [3:0] x3_x0;
5     output [3:0] x3_x0_est;
6
7     assign x3_x0_est = (x3_x0 == 'B0000) ? 'B0000:
8                       (x3_x0 == 'B0001) ? 'B0000:
9                       (x3_x0 == 'B0010) ? 'B0000:
10                      (x3_x0 == 'B0011) ? 'B0000:
11                      (x3_x0 == 'B0100) ? 'B0000:
12                      (x3_x0 == 'B0101) ? 'B1001:
13                      (x3_x0 == 'B0110) ? 'B1001:
14                      (x3_x0 == 'B0111) ? 'B1001:
15                      (x3_x0 == 'B1000) ? 'B1001:
16                      (x3_x0 == 'B1001) ? 'B1001:
17                      /* don't care */ 'BXXXX:
18 endmodule
19
20 // sintesi a porte NOR
21 module b10_field_extensor_n(x3_x0, x3_x0_est);
22     input [3:0] x3_x0;
23     output [3:0] x3_x0_est;
24
25     wire out;
26     assign out = ~(~(x3_x0[3] | x3_x0[2]) | ~(x3_x0[3] | x3_x0[1] | x3_x0[0]));
27
28     assign x3_x0_est[3] = out;
29     assign x3_x0_est[2] = 'B0;
30     assign x3_x0_est[1] = 'B0;
31     assign x3_x0_est[0] = out;
32 endmodule

```

## 17 Lezione del 29-10-24

### 17.1 Riduzione di campo di interi

Vogliamo creare un circuito che passa dalla rappresentazione  $A$  su  $n + 1$  cifre di un numero intero  $a$ , ad un  $A^{RID}$  su  $n$  cifre, che rappresenta sempre  $a$ . Chiaramente questo

non è sempre possibile, e vale soltanto se:

$$a \in \left[ -\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1 \right] \subset \left[ -\frac{\beta^{n+1}}{2}, \frac{\beta^{n+1}}{2} - 1 \right]$$

Avremo quindi bisogno di un flag di overflow  $ow$ , per indicare la non rappresentabilità. Verifichiamo da quanto visto sulle estensioni di campo, che i numeri che rispettano tale proprietà sono i tali per cui  $MSD = 0$  e la cifra successiva  $a_{n-1} < \frac{\beta}{2}$ , e i tali per cui  $MSD = \beta - 1$  e la cifra successiva  $a_{n-1} \geq \frac{\beta}{2}$ . Quindi:

$$ow = 0 \Leftrightarrow \left( a_n = 0 \wedge a_{n-1} < \frac{\beta}{2} \right) \vee \left( a_n = \beta - 1 \wedge a_{n-1} \geq \frac{\beta}{2} \right)$$

Abbiamo sul grafico a farfalla adattato all'estensione su  $n + 1$  bit, che queste regole isolano le due sezioni del campo di numeri estesi  $\left( \left[ -\frac{\beta^{n+1}}{2}, \frac{\beta^{n+1}}{2} - 1 \right] \right)$  che hanno riscontro nel campo ridotto  $\left( \left[ -\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1 \right] \right)$ .

Quindi, in questo caso, il numero  $a$  è rappresentabile su  $n$  cifre, e si può calcolare il ridotto  $A^{RID}$  semplicemente rimuovendo l'ultima cifra, cioè calcolando:

$$A^{RID} = |A|_{\beta^n}$$

Chiamiamo il circuito che riconosce la non riducibilità **circuito di overflow**. In base 2, si ha rispetto alle cifre che  $a_{n-1} < \frac{\beta}{2}$  vale se  $a_{n-1} = 0$ , e viceversa  $a_{n-1} \geq \frac{\beta}{2}$  vale se  $a_{n-1} = 1$ , cioè un numero non è rappresentabile su  $n - 1$  bit se le sue due cifre più significative sono uguali. In questo modo il circuito si traduce in un confronto fra le due cifre più significative  $a_n$  e  $a_{n-1}$ , che si fa con uno XOR.

In Verilog, questo si traduce come:

```
1 // un riduttore di campo per interi in codifica binaria con LSD
2 // @x1_x0, che mette in @ow la fattibilità (attiva bassa) della
3 // riduzione
4 module b2_field_reducer(x1_x0, ow);
5     input [1:0] x1_x0;
6     output ow;
7
8     assign ow = ^x1_x0;
9 endmodule
```

Per completezza, vediamo il circuito equivalente per la base 10 in codifica BCD, realizzato con un comparatore a 4 cifre binarie:

```
1 // un riduttore di campo per interi in codifica BCD con LSD @a3_x0,
2 // @b3_b0, che mette in @ow la fattibilità (attiva bassa) della
3 // riduzione
4 module b10_field_reducer(a3_a0, b3_b0, ow);
5     input [3:0] a3_a0, b3_b0;
6     output ow;
7
8     wire flag_lr;
9
10    n4_b2_comparator cmp (
11        .x3_x0(b3_b0), .y3_y0('B0101),
12        .flag_lr(flag_lr)
13    );
14
15    assign ow = ~((a3_a0 == 'B1001) & ~flag_lr |
16        (a3_a0 == 'B0000) & flag_lr);
17 endmodule
```

### 17.1.1 Moltiplicazione di interi per potenza della base

Vediamo come si realizza un moltiplicatore per  $b = \beta \cdot a$ , dato  $A = (a_n - 1 a_{n-2} \dots a_0)$  rappresentante  $a$  su  $n$  cifre,  $B$  rappresentante  $b$  su  $n + 1$  cifre.

Vogliamo chiederci prima di tutto se  $b$  è sempre rappresentabile da  $B$  su  $n + 1$  cifre. Questo è vero, in quanto si può dimostrare che:

$$B = \beta \cdot A$$

Questo viene da:

$$L: B = \begin{cases} b = \beta \cdot a, & 0 \leq a < \frac{\beta^n}{2} \\ \beta^{n+1} + b = \beta^{n+1} + \beta \cdot a = \beta \cdot (\beta^n + a), & -\frac{\beta^n}{2} \leq a < 0 \end{cases}$$

dove si nota che  $a$  e  $\beta^n + a$  valgono  $A$  nei rispettivi campi di esistenza. Si applica quindi quanto conosceamo sulle moltiplicazioni per potenze di base su naturali, e il prodotto sta su  $n + 1$  cifre.

Per prodotti con potenze ulteriori della base, diciamo  $\beta^k$ , si ha che:

$$b = \beta^k \cdot a \equiv B = \beta^k \cdot A$$

e quindi il risultato starà su  $n + k$  cifre.

### 17.1.2 Divisione per potenza della base

Vogliamo fare l'operazione equivalente per le divisioni, cioè dato  $A$  rappresentante  $a$  su  $n + 1$  cifre, trovare  $B$  rappresentante  $b$  su  $n$  cifre tale per cui  $b = \left\lfloor \frac{a}{\beta} \right\rfloor$ .

Possiamo dimostrare, come prima, che:

$$B = \left\lfloor \frac{A}{\beta} \right\rfloor$$

Per fare ciò, approfittiamo della proprietà vista sul complemento a radice che ci permette di rappresentare  $B$  come  $|b|_{\beta^n}$ :

$$\begin{aligned} B = \left\lfloor \frac{a}{\beta} \right\rfloor_{\beta^n} &= \left\lfloor \frac{\lfloor a/\beta^{n+1} \rfloor \cdot \beta^{n+1} + |a|_{\beta^{n+1}}}{\beta} \right\rfloor_{\beta^n} = \left\lfloor \lfloor a/\beta^{n+1} \rfloor \cdot \beta^n + \frac{|a|_{\beta^{n+1}}}{\beta} \right\rfloor_{\beta^n} \\ &= \left\lfloor \frac{|a|_{\beta^{n+1}}}{\beta} \right\rfloor_{\beta^n} = \left\lfloor \frac{A}{\beta} \right\rfloor = \left\lfloor \frac{A}{\beta} \right\rfloor \end{aligned}$$

Abbiamo quindi che possiamo sfruttare quanto avevamo detto sulla divisione per potenze di basi su naturali, e il quoziente sta su  $n$  cifre.

Per divisioni con potenze ulteriori della base, diciamo  $\beta^k$ , si ha che:

$$b = \left\lfloor \frac{a}{\beta^k} \right\rfloor \equiv B = \left\lfloor \frac{A}{\beta^k} \right\rfloor$$

e quindi il risultato starà su  $n - k$  cifre (o  $A$  dovrà stare su  $n + k$  cifre rispetto a  $B$ , solita cosa).

### 17.1.3 Note sugli shift logico e aritmetico

Abbiamo visto come sono state definite operazioni diverse per lo shift logico (SH) e aritmetico (SA) in linguaggio assembly. Abbiamo visto adesso, però, che moltiplicazione e divisione per la base si fanno allo stesso modo sia su interi che su naturali.

Possiamo dire che, nel caso dello shift a sinistra, effettivamente le operazioni eseguite dal calcolatore sono uguali sia nel caso di SHL che SAR. Per quanto riguarda lo shift a destra, invece, dobbiamo renderci conto che la  $n - 1$ -esima cifra (quella che avevamo escluso dicendo che  $A$  su  $n + 1$  cifre va in  $B$  su  $n$  cifre) resterà comunque nella locazione di memoria, cioè non si possono ridimensionare le locazioni. C'è quindi una differenza sul modo in cui si popola l' $n - 1$ -esimo bit entrante: lo shift aritmetico SAL ripete il MSD (cioè estende l'intero su  $n$  cifre) e lo shift logico SHL introduce sempre zeri (ergo perde i segni nel caso di  $a$  negativi).

## 17.2 Somma di interi

Dati  $A$  e  $B$  in base  $\beta$  su  $n$  cifre, rappresentanti rispettivamente gli interi  $a$  e  $b$ , vogliamo calcolare  $S$  su  $n$  cifre tale  $S$  rappresenta  $s$  e  $s = a + b$ . Abbiamo, che la somma potrebbe uscire dall'intervallo di rappresentabilità su  $n$  cifre, in quanto sta su:

$$-\beta^n \leq s \leq \beta^n - 2$$

e starebbe al massimo su  $n + 1$  cifre:

$$[-\beta^n, \beta^n - 2] \subset \left[ -\frac{\beta^{n+1}}{2}, \frac{\beta^n + 1}{2} - 1 \right]$$

Abbiamo quindi bisogno di flag di overflow,  $ow$ .

Quando  $s$  è invece rappresentabile su  $n$  cifre, si ha che:

$$S = |s|_{\beta^n} = |a + b|_{\beta^n} = ||a|_{\beta^n} + |b|_{\beta^n}|_{\beta^n} = |A + B|_{\beta^n}$$

Questa è la proprietà fondamentale per cui si usa il complemento alla radice, e lo passiamo dimostrare da quanto già dimostrato sulle proprietà dell'operatore modulo.

Si può quindi usare un sommatore (che è quindi indifferente per naturali e interi), e l'unico problema resta determinare il flag  $ow$ .

L'unica cosa che dovremo aggiungere è un modo per calcolare il flag  $ow$ . Per adesso abbiamo dal sommatore l'uscita  $C_{out}$ , cioè il riporto della somma: questa non basta da sola a verificare la rappresentabilità del risultato. Notiamo che la somma è sempre rappresentabile estendendo gli ingressi a  $n + 1$  bit e riducendo in uscita. Se la riduzione è possibile, ergo le ultime due cifre più significative sono diverse, allora la somma è rappresentabile.

In una base arbitraria, per fare ciò devo effettivamente fare lo XOR delle due cifre, mentre in binario posso sfruttare le proprietà dello XOR, ricordando che internamente al full adder, la cifra in uscita  $s$  non è altro che  $a \oplus b \oplus c$ . Si ha quindi:

$$ow = s_n \oplus s_{n-1} = (a_n \oplus b_n \oplus c_n) \oplus (a_{n-1} \oplus b_{n-1} \oplus c_{n-1}) = 0 \oplus c_n \oplus c_{n-1} = c_n \oplus c_{n-1}$$

Possiamo quindi ricavare il flag  $ow$  confrontando il  $C_{out}^n$  con l' $n - 1$ -esimo  $C_{out}^{n-1}$ : a bit uguali sia  $ow$  falso, e viceversa, cioè si usa un singolo XOR

Si ha quindi che la stessa circuiteria esegue somme sia fra interi che fra naturali. In assembly, avevamo visto che la ADD esegue le stesse operazioni, ed è compito del

programmatore controllare i flag di carry o di overflow a seconda di ciò che era andato a sommare (interi  $\rightarrow$  overflow, naturali  $\rightarrow$  carry).

Possiamo quindi aggiornare l'implementazione Verilog del sommatore a 4 cifre binarie per tenere conto di un flag di overflow, come segue:

```

1 // un adder a 4 cifre in base 2 che calcola @x3_x0 + @y3_y0,
2 // mettendo il risultato in @s3_s0 e il riporto in @cout, aggiornato
3 // con un flag di overflow @ow
4 module n4_b2_adder_i(x3_x0, y3_y0, cin, s3_s0, cout, ow);
5     input [3:0] x3_x0, y3_y0;
6     input cin;
7     output [3:0] s3_s0;
8     output cout, ow;
9
10    wire [3:0] carry;
11    assign cout = carry[3];
12
13    assign ow = carry[3] ^ carry[2];
14
15    b2_adder add_0 (
16        .x(x3_x0[0]), .y(y3_y0[0]),
17        .cin(cin),
18        .s(s3_s0[0]),
19        .cout(carry[0])
20    );
21
22    b2_adder add_1 (
23        .x(x3_x0[1]), .y(y3_y0[1]),
24        .cin(carry[0]),
25        .s(s3_s0[1]),
26        .cout(carry[1])
27    );
28
29    b2_adder add_2 (
30        .x(x3_x0[2]), .y(y3_y0[2]),
31        .cin(carry[1]),
32        .s(s3_s0[2]),
33        .cout(carry[2])
34    );
35
36    b2_adder add_3 (
37        .x(x3_x0[3]), .y(y3_y0[3]),
38        .cin(carry[2]),
39        .s(s3_s0[3]),
40        .cout(carry[3])
41    );
42 endmodule

```

### 17.3 Sottrazione di interi

La sottrazione fra interi è analoga alla somma: abbiamo sempre due  $A$  e  $B$  in base  $\beta$  su  $n$  cifre, e vogliamo trovare  $D$  sempre su  $n$  cifre tale per cui fra  $a$ ,  $b$  e  $d$  rappresentati vale  $a - b = d$ . Si ha, prendendo il complemento a radice:

$$D = |d|_{\beta^n} = |a - b|_{\beta^n} = |a - b|_{\beta^n} = ||a|_{\beta^n} - |b|_{\beta^n}| = |A - B|_{\beta^n} = |A + \overline{B} + 1|_{\beta^n}$$

Come prima, abbiamo che il flag `ow` è dato dallo XOR degli ultimi due prestiti (prima erano riporti). Questo si dimostra analogamente a prima, prendendo il bit esteso:

$$\begin{aligned} D^{EST} &= |d|_{\beta^{n+1}} = |a - b|_{\beta^{n+1}} = \left| |a|_{\beta^{n+1}} - |b|_{\beta^{n+1}} \right|_{\beta^{n+1}} \\ &= |A^{EST} - B^{EST}|_{\beta^{n+1}} = |A^{EST} + \overline{B^{EST}} - 1|_{\beta^{n+1}} \end{aligned}$$

cioè si ha che sull' $n + 1$ -esimo bit la differenza è uguale prendendo le estensioni degli ingressi su  $n + 1$  bit, ergo la rappresentabilità è data dalla riducibilità del risultato su  $n$  bit, e quindi come prima dallo XOR sugli ultimi due prestiti.

In Verilog, posso aggiornare il sottrattore a 4 cifre binarie come segue:

```

1 // un sottrattore a 4 cifre in base 2 che calcola @x3_x0 - @y3_y0,
2 // mettendo il risultato in @d3_d0 e il riporto in @bout, aggiornato
3 // con un flag di overflow @ow
4 module n4_b2_subtractor_i(x3_x0, y3_y0, bin, d3_d0, bout, ow);
5     input [3:0] x3_x0, y3_y0;
6     input bin;
7     output [3:0] d3_d0;
8     output bout, ow;
9
10    wire [3:0] borrow;
11    assign bout = borrow[3];
12
13    assign ow = borrow[3] ^ borrow[2];
14
15    b2_subtractor sub_0 (
16        .x(x3_x0[0]), .y(y3_y0[0]),
17        .bin(bin),
18        .d(d3_d0[0]),
19        .bout(borrow[0])
20    );
21
22    b2_subtractor sub_1 (
23        .x(x3_x0[1]), .y(y3_y0[1]),
24        .bin(borrow[0]),
25        .d(d3_d0[1]),
26        .bout(borrow[1])
27    );
28
29    b2_subtractor sub_2 (
30        .x(x3_x0[2]), .y(y3_y0[2]),
31        .bin(borrow[1]),
32        .d(d3_d0[2]),
33        .bout(borrow[2])
34    );
35
36    b2_subtractor sub_3 (
37        .x(x3_x0[3]), .y(y3_y0[3]),
38        .bin(borrow[2]),
39        .d(d3_d0[3]),
40        .bout(borrow[3])
41    );
42 endmodule
43
44 // implementazione con adder a 4 bit
45 module n4_b2_subtractor_i_a(x3_x0, y3_y0, bin, d3_d0, bout, ow);
46     input [3:0] x3_x0, y3_y0;
47     input bin;

```



```

48 output [3:0] d3_d0;
49 output bout, ow;
50
51 wire [3:0] y3_y0_neg;
52 assign y3_y0_neg = ~y3_y0;
53
54 wire cin;
55 assign cin = ~bin;
56
57 wire cout;
58 assign bout = ~cout;
59
60 n4_b2_adder addr (
61   .x3_x0(x3_x0), .y3_y0(y3_y0_neg),
62   .cin(cin),
63   .s3_s0(d3_d0),
64   .cout(cout),
65   .ow(ow)
66 );
67 endmodule

```

## 17.4 Comparazione di numeri interi

Notiamo che c'è una differenza fra la comparazione fra interi e quella fra naturali. Per quanto riguarda l'uguaglianza  $a = b$ , abbiamo effettivamente la stessa cosa dei naturali.

Invece, per la minoranza  $a < b$ , non possiamo più controllare i prestiti uscenti. Dobbiamo quindi guardare il segno del risultato della sottrazione, che deve quindi poter essere svolta: si estende su  $n + 1$  cifre e si controlla la  $n$  esima cifra del risultato: questa varrà da  $\text{sgn}(a - b)$ , e quindi da flag di minoranza per  $a < b$ . Se non si fosse esteso su  $n + 1$  cifre, non avremmo potuto essere sicuri di non aver scartato eventuali valori negativi (negli intervalli di non rappresentabilità).

Possiamo quindi definire un comparatore per interi in Verilog:

```

1 // un comparatore a 4 bit per interi che confronta @x3_x0 e @y3_y0 e
2 // produce i flag:
3 // - flag_eq: @x3_x0 = @y3_y0
4 // - flag_gr: @x3_x0 > @y3_y0
5 // - flag_lr: @x3_x0 < @y3_y0
6 module n4_b2_integer_comparator(x3_x0, y3_y0,
7                                 flag_eq, flag_gr, flag_lr);
8   input [3:0] x3_x0, y3_y0;
9   output flag_eq, flag_gr, flag_lr;
10
11   wire [4:0] x4_x0 = {x3_x0[3], x3_x0};
12   wire [4:0] y4_y0 = {y3_y0[3], y3_y0};
13
14   wire [4:0] d4_d0;
15   wire bout;
16
17   n5_b2_subtractor sub (
18     .x4_x0(x4_x0), .y4_y0(y4_y0),
19     .bin('B0),
20     .d4_d0(d4_d0),
21     .bout(bout),
22   );
23
24   assign flag_eq = (d4_d0 == 'B00000) & ~bout;
25   assign flag_gr = ~d4_d0[4] & ~(d4_d0 == 'B00000) & ~bout;

```

```
26 assign flag_lr = d4_d0[4];
27 endmodule
```

notando che il modulo `n5_b2_subtractor` non è altro che un sottrattore a 5 cifre binarie, che come sempre è implementato nel codice Verilog annesso alla lezione (directory / verilog).

## 17.5 Moltiplicazione e divisione di interi

Moltiplicazioni e divisioni di interi riescono più facili se prima si converte in rappresentazione modulo e segno: i moduli si moltiplicano o dividono come naturali, e il segno viene determinato dai segni degli operandi attraverso la comune algebra alternante ( $+\cdot+ = +$ ,  $+\cdot- = -$ ,  $- \cdot - = +$ ).

Ricordiamo di aver già visto un circuito di conversione da CR a MS. Ci manca quindi il circuito di conversione opposto:

### 17.5.1 Conversone da MS a CR

Vogliamo una rete che prende in ingresso il valore assoluto su  $n$  cifre ed il segno della rappresentazione di un numero intero, e produce un uscita la sua rappresentazione in complemento alla radice su  $n$  cifre. Quest'operazione non è sempre possibile: abbiamo  $-(\beta^n - 1) \leq a \leq \beta^n - 1$  in ingresso e  $-\frac{\beta^n}{2} \leq a \leq \frac{\beta^n}{2} - 1$  in uscita.

Se l'operazione è fattibile, avremo che:

$$A = |a|_{\beta^n} = \begin{cases} |ABS_a|_{\beta^n}, & a \geq 0 \\ | - ABS_a|_{\beta^n}, & a < 0 \end{cases} = \begin{cases} |ABS_a|_{\beta^n}, & a \geq 0 \\ |\overline{ABS_a} + 1|_{\beta^n}, & a < 0 \end{cases}$$

Quindi si usa un multiplexer, con il segno della rappresentazione MS a variabile di controllo, che distingue fra la rappresentazione stessa  $ABS_a$  e il suo complemento (calcolato con un circuito di inversione e incremento).

Per quanto riguarda l'overflow, abbiamo invece che  $ow$  è impostato in due casi:

- Siamo fuori dal campo di rappresentabilità: questo si verifica quando  $\text{abs}(a) > \frac{\beta^n}{2}$ , cioè si è passati oltre agli  $n - 1$  bit su cui dobbiamo ridurre il modulo;
- Si è sull'unico valore positivo che non ha rappresentazione con  $a = \frac{\beta^n}{2}$  e il bit di segno vale 0, cioè  $a$  è uguale al massimo rappresentabile  $\frac{\beta^n}{2} - 1 + 1$ .

Questo si sintetizza nella regola, espressa attraverso l'operatore ternario:

$$ow = \left( \left( \text{abs}(a) > \frac{\beta^n}{2} \right) \vee \left( \left( \text{abs}(a) = \frac{\beta^n}{2} \right) \wedge (\text{sgn}(a) = 1) \right) \right) ? 1 : 0$$

Occorre stare attenti fra la funzione segno e il valore del bit di segno, in quanto vale, come per la convenzione solita sul bit di segno:

$$\begin{cases} \text{sgn}(a) = 1 \Rightarrow \text{sgn} = 0 \\ \text{sgn}(a) = -1 \Rightarrow \text{sgn} = 1 \end{cases}$$

In Verilog questa rete si presenta come la duale del `n4_c2_ms_converter`, e si usa il modulo negatore sintetizzato prima:

```

1 // un convertitore da modulo e segno a complemento a 2 di un intero
2 // su 4 bit @x3_x0_abs e @sgn che mette la rappresentazione in @z3_z0
3 module n4_ms_c2_converter(x3_x0_abs, sgn, z3_z0, ow);
4     input [3:0] x3_x0_abs;
5     input sgn;
6     output [3:0] z3_z0;
7     output ow;
8
9     wire [3:0] x3_x0_neg;
10
11     n4_c2_negator neg (
12         .x3_x0(x3_x0_abs), .z3_z0(z3_z0(x3_x0_neg)
13     );
14
15     assign ow = ~((x3_x0_abs == 'B1000) & sgn | ~x3_x0_abs[3]);
16
17     assign z3_z0 = (sgn) ? x3_x0_neg : x3_x0_abs;
18 endmodule

```

Riassumiamo brevemente come si svolge la conversione fra complemento a radice e rappresentazione modulo e segno. Abbiamo sostanzialmente che, salvo il caso della conversione da MS a CR dove si può incappare in non rappresentabilità, la conversione da CR a MS e viceversa si fa sempre con un multiplexer che distingue fra la rappresentazione  $A$  presa così com'è e il suo complemento calcolato con inversione incremento. In particolare:

- Nel caso **CR a MS**, si prende  $A$  se per la cifra più significativa  $a_{n-1}$  vale  $a_{n-1} \geq \frac{\beta}{2}$ , altrimenti il complemento  $\bar{A}$ : questo significa che il multiplexer è pilotato dalla MSD;
- Nel caso **MS a CR**, si prende  $A$  se il bit di segno non è impostato, e il complemento  $\bar{A}$  altrimenti: questo significa che il multiplexer è pilotato dal bit di segno.

### 17.5.2 Moltiplicazione

Per svolgere la moltiplicazione vogliamo quindi trasformare due ingressi  $A$  e  $B$ , rappresentanti gli interi  $a$  e  $b$  su  $n$  e  $m$  bit, nella loro rappresentazione MS come:

$$a \Rightarrow \text{sgn}(a), \text{abs}(a), \quad a \Rightarrow \text{sgn}(a), \text{abs}(a),$$

dove i segni stanno su un bit e i moduli su  $n$  e  $m$  bit.

Abbiamo che la moltiplicazione dei moduli di  $A$  e  $B$  è sempre rappresentabile, in quanto:

$$\text{sgn}(A) \cdot \text{sgn}(B) \leq \frac{\beta^n}{2} \cdot \frac{\beta^m}{2} = \frac{\beta^{n+m}}{2}$$

Osserviamo quindi che il prodotto intero  $p$  è:

$$p = \begin{cases} \text{abs}(a) \cdot \text{abs}(b), & \text{sgn}(a) = \text{sgn}(b) \\ -\text{abs}(a) \cdot \text{abs}(b), & \text{sgn}(a) \neq \text{sgn}(b) \end{cases}$$

ergo:

$$\begin{cases} \text{abs}(p) = \text{abs}(a) \cdot \text{abs}(b) \\ \text{sgn}(p) = \text{sgn}(a) \cdot \text{sgn}(b) \end{cases}$$

cioè quanto avevamo detto sulla rappresentazione modulo e segno per i prodotti.

Si ha quindi che il moltiplicatore fra interi si realizza convertendo gli ingressi da CR a MS, mandando i valori assoluti ad un moltiplicatore per naturali, e ricavando il segno del successivo convertitore da MS a CR (che ci darà il risultato) da uno XOR fra i segni degli MS in ingresso. L'overflow non è considerato in quanto non potrà mai verificarsi (da sopra).

In Verilog, quindi, usiamo le definizioni date precedentemente di convertitori fra modulo e segno e complemento a 2:

```

1 // un moltiplicatore per interi a 4 * 4 cifre che calcola
2 // @x3_x0 * @y3_y0 e lo mette in @p7_p0
3 module n4by4_b2_integer_multiplier(x3_x0, y3_y0, p7_p0);
4     input [3:0] x3_x0, y3_y0;
5     output [7:0] p7_p0;
6
7     wire [3:0] x3_x0_abs, y3_y0_abs;
8     wire x_sgn, y_sgn;
9
10    n4_c2_ms_converter x_conv (
11        .x3_x0(x3_x0),
12        .z3_z0(x3_x0_abs), .sgn(x_sgn)
13    );
14
15    n4_c2_ms_converter y_conv (
16        .x3_x0(y3_y0),
17        .z3_z0(y3_y0_abs), .sgn(y_sgn)
18    );
19
20    wire [7:0] p7_p0_abs;
21    wire p_sgn;
22
23    n4by4_b2_multiplier mul (
24        .x3_x0(x3_x0_abs), .y3_y0(y3_y0_abs), .c3_c0('B0000),
25        .p7_p0(p7_p0_abs)
26    );
27
28    assign p_sgn = x_sgn ^ y_sgn;
29
30    n8_ms_c2_converter p_conv (
31        .x7_x0_abs(p7_p0_abs), .sgn(p_sgn),
32        .z7_z0(p7_p0)
33    );
34 endmodule

```

Il modulo `n8_ms_c2_converter`, in particolare, è un convertitore da modulo e segno a complemento a 2 su 8 cifre binarie (si è data, nella cartella `/verilog`, sia un implementazione a 4 che a 8 cifre).

### 17.5.3 Divisione

Vogliamo calcolare, dati due naturali  $A$  e  $B$  rappresentanti gli interi  $a$  e  $b$  su  $n$  e  $m$  cifre, il quoziente  $Q$  e il resto  $R$ , rispettivamente su  $n$  e  $m$  cifre, tali che:

$$a = q \cdot b + r$$

Per svolgere questa divisione abbiamo bisogno di una riformulazione del teorema della divisione con resto che funzioni sull'anello  $\mathbb{Z}$ , in quanto adesso la semplice  $a = q \cdot b + r$  con  $r < b$  ammette infiniti valori di  $r$  (che può essere negativo). Decidiamo quindi di imporre:

- Il quoziente  $q$  è positivo se i segni di  $a$  e  $b$  sono concordi, e negativo viceversa;
- $r$  e  $b$  sono uguali in segno.

Da qui si ha la proprietà più importante, cioè:

$$\begin{cases} \text{abs}(r) < \text{abs}(b) \\ \text{sgn}(r) = \text{sgn}(b) \end{cases}$$

Si verifica che questo significa che vogliamo i risultati che ci aspettiamo dalla comune divisione fra interi.

Abbiamo quindi, pensando in modulo e segno, che:

$$a = q \cdot b + r$$

diventa:

$$\text{sgn}(a) \cdot \text{abs}(a) = q \cdot \text{sgn}(b) \cdot \text{abs}(b) + \text{sgn}(r) \cdot \text{abs}(r)$$

Ma se avevamo  $\text{sgn}(a) = \text{sgn}(r)$ , allora:

$$\text{abs}(a) = (q \cdot \text{sgn}(b) \cdot \text{sgn}(a)) \cdot \text{abs}(b) + \text{abs}(r)$$

Si nota quindi che  $q \cdot \text{sgn}(b) \cdot \text{sgn}(a)$  è semplicemente  $\text{abs}(q)$ , ergo si può rendere la divisione fra interi come la divisione fra i moduli di quegli interi, prendendo il segno separatamente (che è quello che avevamo fatto per la moltiplicazione).

Resta da trovare il valore del flag di non fattibilità `no_div`. Abbiamo, dal divisore fra naturali, che la rappresentabilità è data da:

$$\text{abs}(a) < \beta^n \cdot \text{abs}(q)$$

Questa condizione non basta, in quanto non si è ancora assicurato che l'intero in uscita sia rappresentabile su  $n$  cifre. Si prende quindi anche il flag di overflow `ow`, ricavato dalla conversione finale da MS a CR, e si mette a OR con il flag `no_div` che abbiamo ricavato dal circuito divisore.

Tutto questo si traduce nell'ultima rete puramente combinatoria che vedremo, ovvero:

```

1 // un divisore per interi a 4 / 2 cifre che calcola @x3_x0 / @y1_y0,
2 // mette il quoziente in @q1_q0 e il resto in @r1_r0. no_idiv
3 // rappresenta la non fattibilit 
4 module n4by4_b2_integer_multiplier(x3_x0, y1_y0, q1_q0, r1_r0,
5                                     no_idiv);
6     input [3:0] x3_x0;
7     input [1:0] y1_y0;
8     output [1:0] q1_q0;
9     output [1:0] r1_r0;
10    output no_idiv;
11
12    wire [3:0] x3_x0_abs;
13    wire [1:0] y1_y0_abs;
14    wire x_sgn, y_sgn;
15
16    n4_c2_ms_converter x_conv (
17        .x3_x0(x3_x0),
18        .z3_z0(x3_x0_abs), .sgn(x_sgn)
19    );

```

```

20
21   n2_c2_ms_converter y_conv (
22     .x1_x0(y1_y0),
23     .z1_z0(y1_y0_abs), .sgn(y_sgn)
24   );
25
26   wire [1:0] q1_q0_abs;
27   wire q_sgn;
28
29   wire [1:0] r1_r0_abs;
30   wire r_sgn;
31
32   wire no_div;
33
34   n4by2_b2_divider div (
35     .x3_x0(x3_x0_abs), .y1_y0(y1_y0_abs),
36     .q1_q0(q1_q0_abs), .r1_r0(r1_r0_abs),
37     .no_div(no_div)
38   );
39
40   assign q_sgn = x_sgn ^ y_sgn;
41   assign r_sgn = x_sgn;
42
43   wire ow;
44
45   n2_ms_c2_converter q_conv (
46     .x1_x0_abs(q1_q0_abs), .sgn(q_sgn),
47     .z1_z0(q1_q0), .ow(ow)
48   );
49
50   n2_ms_c2_converter r_conv (
51     .x1_x0_abs(r1_r0_abs), .sgn(r_sgn),
52     .z1_z0(r1_r0)
53   );
54
55   assign no_idiv = ow | no_div;
56 endmodule

```

## 18 Lezione del 30-10-24

### 18.1 La funzione di memoria

Finora abbiamo visto reti combinatorie, cioè **reti prive di memoria**, dove lo stato di uscita ad un istante dipende solo dallo stato di ingresso corrente. Nelle **reti sequenziali**, invece, l'uscita dipende dalla sequenza degli stati di ingresso visti dalla rete fino a quel momento. Questa memoria si implementa attraverso **anelli di retroazione**.

Prendiamo ad esempio un buffer con un anello di retrazione, cioè una linea che porta la sua uscita al suo ingresso, e che estrae in uscita  $q$ .

Questo potrà quindi esistere in due situazioni di stabilità:

- L'uscita vale 0, e va in ingresso al buffer, dove si **rigenera** (o si *autosostiene*);
- L'uscita vale 1, e va in ingresso al buffer, dove ancora una volta si rigenera e mantiene il suo valore.

La presenza del buffer è fondamentale: mantiene l'uscita  $q$  a 0 e 1, e soprattutto ci assicura di poter associare a quel punto della rete uno stato logico.

Il problema di una rete di questo genere è che è fundamentalmente inutile: non si può controllare lo stato di stabilità del buffer, e quindi non si possono immagazzinare bit diversi a tempi diversi.

### 18.1.1 Uscita negata

Realizziamo allora il nostro buffer, sostituendolo con due porte NOR disposte come invertitori (quindi un doppio invertitore, che equivale al buffer). Si ha che fra le due porte NOR abbiamo il valore complementato del buffer, cioè 1 a 0 e 0 a 1. Possiamo quindi dotare la rete di un'ulteriore uscita  $q_N$ , che equivale appunto alla negazione di  $q$ . Per questo motivo, avevamo detto, nella valutazione dei livelli di logica si ignorano le porte NOT: solitamente abbiamo già un valore negato a disposizione dai registri.

### 18.1.2 Stato all'accensione

Ora, se all'accensione  $q$  e  $q_N$  sono discordi, la rete si troverà già in uno degli stati stabili, e lì resterà. In caso contrario, se sono concordi, teoricamente ciascuna delle due uscite dovrebbe oscillare all'infinito, con un periodo pari al doppio del tempo di risposta delle porte (in due passaggi si completa un ciclo, cioè l'ingresso della prima porta torna al neutro). Nella pratica, la rete si stabilizza, in quanto il tempo delle porte sarà necessariamente diverso e quindi si creerà prima o poi una condizione analoga a prima, dove le uscite sono discordi e la rete resta stabile.

### 18.1.3 Latch SR

Vediamo quindi come rendere pilotabile lo stato del circuito. Introduciamo due ingressi negli input (finora duplicati) delle porte NOR: in entrata alla prima porta avremo il comando  $S$ , per SET, e in entrata alla seconda porta avremo il comando  $R$ , per RESET. Questi ingressi sono *attivi alti*, cioè i comandi  $S$  e  $R$  vengono dati quando le rispettive entrate sono in tensione. Chiamiamo questa rete **latch SR**, a volte impropriamente detta *flip-flop SR*.

Vediamo il funzionamento della rete nei diversi casi di attivazione degli ingressi:

- $S = 1, R = 0$ : si ha che la prima NOR ha un ingresso 1, ergo mette l'uscita a 0. Quindi, la seconda NOR ha un ingresso 0, ergo mette l'uscita a 1. Ci troviamo nella configurazione stabile  $q = 1, q_N = 0$ , cioè abbiamo memorizzato un bit.
- $S = 0, R = 1$ : si ha che la seconda NOR ha un ingresso 1, ergo mette l'uscita a 0. Quindi, la prima NOR ha un ingresso 0, quindi mette l'uscita a 1. Ci troviamo nella configurazione stabile  $q = 0, q_N = 1$ , cioè abbiamo resettato un bit.
- $S = 0, R = 0$ : l'uscita della prima NOR vale 0 se  $q = 1$ , e 1 se  $q = 0$ , quindi  $q_N$  dà semplicemente  $\bar{q}$  e viceversa, e la rete conserva il valore che aveva precedentemente. Questo comportamento rende la rete **sequenziale**: nello stato di **conservazione**, cioè quello a ingressi disattivati, si ha che la rete rimane nello stato stabile  $S_0$  o  $S_1$  nel quale si era portata in un momento precedente nella sequenza di stati. Si può anche dire che la rete **ricorda** l'ultimo SET o RESET ricevuto. Comunque, è una rete **asincrona**, in quanto l'uscita si aggiorna subito rispetto agli ingressi (e non in sincronia ad un clock).
- $S = 1, R = 1$ : semanticamente, questa istruzione non ha molto significato. In uno stato di pilotaggio corretto, diciamo che questo stato **non è permesso**. Se si venisse

a verificare, avremmo che alla prima porta un'entrata è 1, e quindi l'uscita è 0. Alla seconda porta, quindi, un'uscita sarà 1, e avremo di nuovo uscita 0. Forzeremmo quindi la rete in uno stato  $q = 0$ ,  $q_N = 0$ , che non significa nulla dal punto di vista della rappresentazione in bit della memoria.

#### 18.1.4 Tabella di applicazione

Per descrivere il comportamento delle reti con memoria usiamo le **tabelle di applicazione**. Queste rappresentano, a sinistra, il valore attuale della variabile e il valore successivo che si vuole questa assuma, e a destra il comando necessario perché l'uscita passi dal valore attuale a quello successivo. Nel caso del latch SR, si ha che questa è:

$q$	$q'$	$s$	$r$
0	0	0	-
0	1	1	0
1	0	0	1
1	1	-	0

#### 18.1.5 Regole di pilotaggio

Avevamo visto le regole per le reti combinatorie:

- Siamo in **pilotaggio in modo fondamentale**: si cambiano gli ingressi solo quando la rete è a regime;
- Gli stati di ingresso consecutivi devono essere adiacenti (per evitare *race condition*).

Vogliamo definire una serie di regole simili per le reti sequenziali. Abbiamo che la regola di **pilotaggio in modo fondamentale** va rispettata comunque: la rete avrà un certo **tempo di attraversamento** di cui tenere conto. Anche la seconda regola, degli **stati di ingressi consecutivi adiacenti**, è fondamentale: se non viene rispettata, si possono presentare in ingresso stati transitori spuri, e l'evoluzione delle uscite diventa imprevedibile.

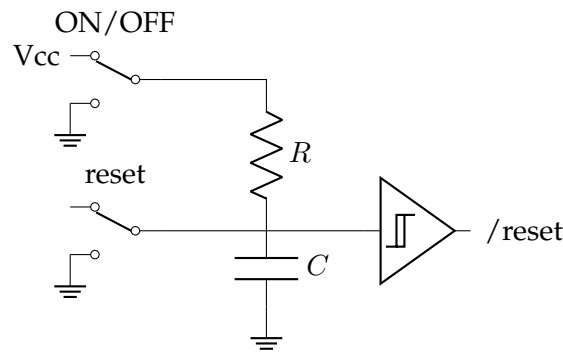
Nel latch SR, però, vale che questa legge può essere violata: cioè il latch SR è robusto nei confronti di pilotaggi scorretti. Questo è il punto di forza che lo rende la rete alla base dei registri e di tutti gli elementi di memoria.

#### 18.1.6 Lo stato iniziale

Abbiamo detto che l'SR è l'elemento alla base dei circuiti di memoria. Un SR può contenere informazioni, che corrispondono allo stato  $S_0$  o  $S_1$  in cui si trova. Si ha, però, che all'accensione il bit contenuto nell'SR è **casuale** (da quello che avevamo visto dalle modalità di pilotaggio). All'accensione di un calcolatore, si ha che alcuni elementi possono avere un valore casuale (ad esempio la RAM), altri no (ad esempio l'istruzione pointer). Si definisce quindi una **fase di reset**, distinta dalla **fase operativa**, cioè quella di operatività standard. Nella fase di reset si inizializzano gli elementi di memoria: notiamo che questo reset non corrisponde al comando R, di RESET, che diamo ai latch. In generale, quindi, non è vero che gli elementi di memoria contengono tutti zero all'accensione del calcolatore.

Vediamo quindi il circuito:





Abbiamo che la circuiteria di trigger è realizzata attraverso un circuito RC, fra l'interruttore ON/OFF e il pulsante di reset, con  $\tau = R \cdot C \approx 0 \mu s$ , dove si collega il nodo fra R e C ad un trigger di Schmitt. Il trigger di Schmitt effettivamente "quantizza" la tensione, cioè scatta ad un valore 1 di tensione solamente quando la tensione in entrata è maggiore di una certa soglia. Abbiamo quindi che, spostando l'interruttore nella posizione ON, il circuito raggiunge il regime in un tempo  $\approx \tau$ , e quindi il trigger va a 1 in un tempo  $\approx \tau$ . Lo stesso quando si preme il pulsante di reset, il capacitore C si scarica e dobbiamo riportare nuovamente il circuito a regime, per cui abbiamo un istante  $\approx \tau$  dove il trigger è a 0.

Abbiamo che l'uscita di questa rete va ad un ingresso detto /reset (che ricordiamo è distinto dai singoli reset dei latch SR), che è *attivo basso*: cioè nella fase iniziale dell'accensione, e ad ogni pressione successiva del pulsante reset, si ha che dal trigger esce per un tempo  $\approx \tau$  il comando di /reset.

Per implementare effettivamente il meccanismo di reset si dota il latch SR di due ingressi aggiuntivi: /preset e /preclear, entrambi attivi bassi. Si distinguono quindi i seguenti casi:

- /preset = /preclear = 1: la rete si comporta come un latch SR normale;
- /preset = 0: la rete si trova nello stato  $S_1$  (indipendentemente dallo stato di  $s$  e  $r$ );
- /preclear = 0: la rete si trova nello stato  $S_0$  (indipendentemente dallo stato di  $s$  e  $r$ );
- /preset = /preclear = 0: abbiamo, come nel caso già visto dei semplici ingressi Set e Reset, che questo stato non è permesso, e quindi non è interessante conoscere il funzionamento della rete in tale stato.

Abbiamo quindi che per inizializzare un latch SR a 1 si porta /preset a /reset, e /preclear al Vcc. Viceversa, per inizializzare il latch a 0 si porta /preset al Vcc e /preclear a /reset.

Vogliamo quindi modificare la sintesi del latch SR: conviene unirlo ad una rete combinatoria, che ha per ingresso S, R, /preset e /preclear, e in uscita  $z_s$  e  $z_r$  (che andranno in ingresso al latch vero e proprio). L'obiettivo di questa rete è di impostare i corrispondenti comandi di SET e RESET se uno fra /preset e /preclear è attivo basso, o di restituire S e R così come sono in caso entrambi siano alti.

Abbiamo, dalla sintesi con le mappe di Karnaugh, riportando i valori in coppie  $(z_s, z_r)$ :

		/preset/preclear			
		00	01	11	10
SR	00	–	10	00	01
	01	–	10	01	01
	11	–	10	10	01
	10	–	10	10	01

Si visualizzano i sottocubi nelle mappe presi separatamente  $z_s$  e  $z_r$ :

- $z_s$ :

		/preset/preclear			
		00	01	11	10
SR	00	-	1	0	0
	01	-	1	0	0
	11	-	1	1	0
	10	-	1	1	0

	S	R	/preset	/preclear
A	-	-	0	-
B	1	-	-	1

- $s_r$ :

		/preset/preclear			
		00	01	11	10
SR	00	-	0	0	1
	01	-	0	1	1
	11	-	0	1	1
	10	-	0	0	1

	S	R	/preset	/preclear
A	-	-	-	0
B	-	1	1	-

Da cui si ricavano le due sintesi di  $z_s$  e  $z_r$ :

$$\begin{cases} z_s = \overline{\text{preset}} + (\text{preclear} \cdot s) \\ z_r = \overline{\text{preclear}} + (\text{preset} \cdot r) \end{cases}$$

A questo punto, visto che il latch SR è realizzato a porte NOR, possiamo semplificare gli OR e i NOR in cascata: se assumiamo una NOR come una OR in serie ad una NOT, si ha che due OR equivalgono a una singola OR, ergo si possono mandare le uscite delle reti combinatorie appena sintetizzate direttamente ai NOR del latch SR, rimuovendo le OR che avremmo normalmente introdotto in una sintesi SP. Questo processo viene a volte detto *compenetrazione*.

## 18.2 Tabelle e grafi di flusso

Le reti sequenziali, più spesso che con la tabella di applicazione, si descrivono usando **tabelle di flusso** e **grafi di flusso**.

### 18.2.1 Tabelle di flusso

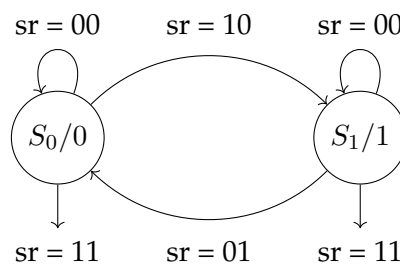
Una tabella di flusso è una tabella che descrive come si evolvono lo stato interno e l'uscita al variare degli stati di ingresso. Ad esempio, per un latch SR, ignorando  $\text{preclear}$  e  $\text{preset}$ :

	00	01	11	10	q
$S_0$	$\textcircled{S_0}$	$\textcircled{S_0}$	-	$s_1$	0
$S_1$	$\textcircled{S_1}$	$s_0$	-	$\textcircled{S_1}$	1

Si ha che nella tabella, le righe rappresentano gli **stati interni presenti** (SIP) e le colonne i possibili ingressi in entrata: all'intersezione fra uno stato e un ingresso si ha lo **stato interno successivo** (SIS). Si indicano con la barra (–) gli stati non definiti (in questo caso *non permessi*). Inoltre, l'ultima colonna indica il valore effettivo delle uscite in ogni stato (qui si è riportato solo  $q$ , e non  $q_N$ ). Gli stati interni successivi cerchiati sono quelli che restano invariati dagli stati interni presenti precedenti: cioè, le coppie di stati interni presenti e ingressi che individuano uno stato interno successivo cerchiato sono coppie **stabili**.

### 18.2.2 Grafi di flusso

Un formalismo del tutto identico è quello del grafo di flusso: si prendono gli stati come nodi, e si disegnano archi (orientati) etichettati con gli stati di ingresso. Gli archi uscenti da un nodo simboleggiano quindi i possibili ingressi di quello stato, e entrano nei nodi che rappresentano gli stati interni successivi. Ad esempio, il grafo corrispondente alla tabella di flusso precedente è:



Notiamo come ad ogni nodo si può associare, separato dalla barra (/), l'uscita corrispondente a un dato stato interno presente. Inoltre, gli stati non definiti vengono indicati con frecce non dirette verso alcun nodo.

Questi strumenti sono utili per la descrizione e la verifica (in questo caso, sotto **ispezione, statica**) delle reti logiche. Nel caso di reti sequenziali, poi, la verifica **dinamica** si fa attraverso un **diagramma di temporizzazione**, cioè un grafico temporale del valore logico di ogni variabile di interesse, creato seguendo i passaggi:

1. Si decide uno stato iniziale;
2. Si attribuiscono valori agli ingressi nel tempo;
3. Si osserva l'evoluzione temporale della rete.

## 19 Lezione del 05-11-24

### 19.1 D-Latch trasparente

Introduciamo una nuova rete sequenziale dotata di due ingressi,  $d$  (data) e  $c$  (control), e un'uscita  $q$ . Il D-latch memorizza il bit in  $d$  quando  $c$  (**trasparenza**) vale 1. Quando  $c$  vale 0, invece, si dice che è in **conservazione**, ergo memorizza l'ultimo valore che  $d$  ha assunto quando  $c$  valeva 1.

La tabella di flusso di questa rete è la seguente, assunti in quest'ordine  $c$  e  $d$ :

	00	01	10	11	q
$S_0$	$S_0$	$S_0$	$S_0$	$S_1$	0
$S_1$	$S_1$	$S_1$	$S_0$	$S_1$	1

cioè quando si è in conservazione, qualsiasi valore di  $d$  viene ignorato e si memorizza il valore passato. Quando si è in trasparenza, invece,  $q$  si adegua a  $d$ .

Si può realizzare un D-latch attraverso un latch SR, con in ingresso una certa rete combinatoria. Quello che vogliamo fare è portare  $d$  e  $c$  in  $s$  e  $r$ , attraverso la tabella di verità:

$c$	$d$	$s$	$r$
0	0	0	0
0	1	0	0
1	0	0	1
1	1	1	0

Questo si sintetizza in  $s = c \cdot d$  e  $r = c \cdot \bar{d}$ . Si ha che le porte AND che rappresentano le congiunzioni in questa rete combinatoria possono collapsare con le porte AND che formavano la rete combinatoria del latch SR che permetteva preset e preclear.

#### 19.1.1 Pilotaggio del D-Latch

Nel pilotaggio del D-latch, dobbiamo assicurarci che  $d$  sia costante a cavallo della transizione di  $c$  da 1 a 0, in quanto potremmo finire per memorizzare dati ignoti (l'ultima cosa che il D-latch ha "visto" prima del reset di  $c$ ). I tempi per cui  $d$  deve essere costante, rispettivamente **prima** e **dopo** della transizione di  $c$ , si dicono  $T_{setup}$  e  $T_{hold}$ , e sono dati di progetto.

### 19.1.2 Trasparenza

Quando il D-latch è in **trasparenza**, il suo ingresso è direttamente connesso, in **senso logico** (ci sono comunque ritardi nella logica delle reti), all'uscita. Per questo motivo, se  $q$  e  $d$  sono collegati in **retroazione negativa** (un feedback loop negato), si ha che con  $c = 1$  abbiamo oscillazioni incontrollate, e che con  $c = 0$  in  $q$  (cioè lo stato interno) resta un valore casuale (l'ultimo rilevato durante le oscillazioni casuali prima che  $c$  sia transitato a 0).

Questo significa che il D-latch è una rete **trasparente**, cioè *la sua uscita cambia mentre la rete è sensibile alle variazioni di ingresso*. Questo significa che non possiamo memorizzare niente che sia funzione dell'uscita (saremmo nel caso della retroazione negativa di prima).

Poniamo di voler eseguire un'istruzione semplice come **INC %AX**. A livello hardware, questo significa connettere un registro (quindi una serie di D-latch) ad una rete combinatoria per l'incremento (probabilmente un half adder), e quindi l'uscita di questa rete di nuovo al D-latch. Quello che abbiamo essenzialmente creato è un ciclo di retroazione: il sistema devolverà velocemente in uno stato di oscillazione incontrollata.

## 19.2 D flip-flop

Il **D flip-flop** è una rete sequenziale **non trasparente** che si pone di risolvere i problemi di trasparenza del D-latch. Quello che vedremo nel dettaglio è il **positive edge-triggered D flip-flop**, che è una rete che si comporta come segue, sulla base degli ingressi  $d$  (data) e  $p$ , e l'uscita  $q$ : quando  $p$  ha un fronte di salita, memorizza  $d$ , *attendi* un determinato istante temporale e adegua l'uscita.

Possiamo concettualizzare il D flip-flop come composto, alla base, da un D-latch. Mettiamo a  $c$ , invece dell'ingresso  $p$ , il **generatore di impulso**  $P^+$  sul fronte di salita di  $p$ . In uscita a  $q$ , poi, abbiamo un buffer  $\Delta$ , che introduce ritardo. La proprietà fondamentale che desideriamo è:

$$\Delta > P^+$$

Questo significa che  $q$  si adegua al valore campionato di  $d$  soltanto *dopo* che la rete ha smesso di essere sensibile a  $d$ . È questa proprietà a rendere il D flip-flop una rete non trasparente.

### 19.2.1 Pilotaggio del D flip-flop

Innanzitutto, a cavallo del fronte di salita di  $p$  l'ingresso  $d$  deve rimanere costante, ergo si hanno gli stessi  $setup$  e  $T_{hold}$  del D-latch. Inoltre, si ha il ritardo di adeguamento dell'uscita, che denominiamo  $T_{prop}$  (dall'inglese *propagation*). Qui la disuguaglianza di prima si traduce come:

$$T_{prop} > T_{hold}$$

Si che l'uscita di un D-FF non oscilla mai, a differenza di quella del D-Latch: l'adeguamento avviene in modo "secco", sul fronte di salita, e di lì in poi fino a reset e successivo set di  $p$ , l'uscita  $q$  è in conservazione e ignora il comportamento di  $d$ .

### 19.2.2 Sintesi Master-Slave di un D flip-flop

Storicamente, un D flip-flop è stato realizzato attraverso un montaggio master/slave, attraverso due D-latch in cascata (di cui uno master, e l'altro chiaramente slave). Si invia quindi l'ingresso  $p$  allo slave, e il suo negato al master, e si fa passare la linea  $d$  prima

dal master, poi dalla sua uscita all'ingresso del slave, e poi al  $q$  del D flip-flop. Si ha che negli stati:

- $p = 0$ : **master** e in *trasparenza*, **slave** in *conservazione*;
- $p = 1$ : **master** in *conservazione*, **slave** in *trasparenza*.

Quando  $p$  è a 0, lo slave è in conservazione, quindi la rete memorizza. Nel frattempo il master è in trasparenza, quindi reagisce al valore in entrata di  $d$ . Quando  $p$  transisce a 1, lo slavean automa in automaton theory passa in trasparenza, e quindi risponde a quello che esce dal master, che invece si trova in conservazione del valore che aveva un'attimo prima della transizione. Il risultato è un comportamento effettivamente analogo a quello della struttura a generatore di impulso e buffer vista prima.

Si possono avere problemi nel funzionamento transitorio dei due D-latch: per questo si agisce elettronicamente, sviluppando questi per commutare  $c$  a valori di tensione diversi. In particolare, vogliamo che in transizione di  $p$  da 1 a 0 lo slave conservi il valore prima che il master passi a trasparenza, quindi che  $c$  dello slave commuti prima di  $c$  del master.

Nella pratica, infine, si ha che la sintesi reale di un D flip-flop è fatta a partire da un latch SR, prima del quale si dispone una rete sequenziale asincrona la cui sintesi è fuori dagli scopi del corso.

### 19.3 Memorie RAM statiche

Esistono due tipi principali di memoria:

- **S-RAM**, costituite da matrici di D latch;
- **D-RAM**, realizzate in modo diverso, che per adesso ignoreremo.

Una riga di D latch rappresenta quindi una **locazione di memoria**, che può essere **letta** o **scritta** con apposite operazioni, strettamente **non simultanee**.

Una SRAM è presenta gli ingressi e le uscite:

- **Ingressi di indirizzo**: in numero sufficiente per indirizzare tutte le celle di memoria. Ad esempio, con  $2^{23}$  celle di 4 bit, 23 ingressi;
- **Ingressi/uscite di dati**: che andranno forchettati con porte **tri-state**;
- **Memory read** e **memory write**, segnali attivi bassi;
- **Select**, segnale attivo basso che fa da **enabler**, in modo simile a quanto avevamo visto nei decoder.

Il comportamento che vogliamo dalla SRAM è il seguente:

/s	/mr	/mw	Azione
1	-	-	Nulla
0	1	1	Nulla
0	0	1	Lettura in corso
1	1	0	Scrittura in corso

### 19.3.1 Temporizzazione delle RAM statiche

Facciamo innanzitutto la divisione lettura/scrittura:

- **Lettura:** per fare una lettura bisogna dare il comando (attivo basso) di memory read ( $/\text{mr}$ ), e impostare l'indirizzo di lettura. Il comando di select ( $/\text{s}$ ) arriva in ritardo, e a quel punto, quando sia  $/\text{s}$  che  $/\text{mr}$  sono in conduzione, i multiplexer vanno a regime e si può fare una lettura sull'uscita dei dati. Infine, quando  $/\text{mr}$  torna a 1, i dati tornano ad alta impedenza, e l'indirizzo di lettura e la select possono assumere valori arbitrari.
- **Scrittura:** si ha che la scrittura è **distruttiva** (manda i D-latch in trasparenza). Bisogna quindi attendere che il select  $/\text{s}$  e gli indirizzi siano stabili prima di abbassare  $\text{mw}$  per dare il comando di scrittura (l'opposto di quanto avevamo fatto in lettura, qui vogliamo scrivere solo quando siamo sicuri di poterlo fare, ergo i multiplexer sono a regime). A questo punto, abbiamo che quando  $\text{mw}$  torna alto dobbiamo assicurarci che i dati in scrittura siano fermi, in quanto i multiplexer riportano gli ingressi di controllo dei D-latch a 0 e l'indirizzo di lettura e la select possono, nuovamente, assumere valori arbitrari.

### 19.3.2 Montaggio di banchi di memoria

Vediamo come combinare più banchi di memoria per aumentare lo spazio di memoria indirizzabile.

- **Montaggio in parallelo:** prendiamo in considerazione due banchi di memoria da  $8\text{M} \times 4$  bit, e vediamo come collegarli per formare un singolo banco di memoria da  $8\text{M} \times 8$  bit, quindi raddoppiando la dimensione delle locazioni.
  - Per quanto riguarda gli **indirizzi di lettura**, basta inviare l'indirizzo ad entrambi i banchi, da cui preleveremo la parte *alta* e *bassa* della locazione;
  - Per quanto riguarda gli **ingressi/uscite di dati**, avremo che la combinazione delle linee sui due banchi, da 4 bit ciascuna, formano un singolo byte da 8 bit, ergo la locazione di memoria completa.
- **Montaggio in serie:** prendiamo in considerazione due banchi di memoria da  $8\text{M} \times 8$  bit, e vediamo come collegarli per formare un singolo banco di memoria da  $16\text{M} \times 8$  bit, quindi raddoppiando il numero di locazioni.
  - Per quanto riguarda gli **indirizzi di lettura**, discrimina dal MSB dell'indirizzo se selezionare dal primo o dal secondo banco, che faranno quindi da parte *alta* e *bassa* dello spazio di memoria indirizzabile. Facciamo questo attraverso l'ingresso di select  $/\text{s}$ , che useremo per determinare altri due segnali di select  $/\text{s}_1$  e  $\text{s}_h$  (*select low* e *select high*), che a loro volta ci permettono di discriminare sulla base del MSD quale banco andiamo a selezionare (effettivamente rendere attivo);
  - Per quanto riguarda gli **ingressi/uscite di dati**, avremo che il banco attivo in un dato momento determina completamente le uscite. Potremmo pensare di dover inserire porte tri-state in uscita ai singoli banchi di memoria sulla linea di ingresso/uscita, ma questo non è necessario: le  $\text{s}_1$  e  $\text{s}_h$  sono mutualmente esclusive, e quindi non si verificherà mai il caso in cui le linee di uscita di entrambi i banchi sono in conduzione contemporaneamente.

## 20 Lezione del 06-11-24

### 20.1 Collegamento al bus e maschere

Resta da capire da dove vengono gli indirizzi di lettura usati nell'accesso alla SRAM. Questi si trovano su un **bus indirizzi**, il cui valore è impostato dal processore ogni volta che vuole effettuare un accesso. Ad esempio, per un modulo di RAM da  $256\text{M} \times 8$  bit, quindi con 28 ingressi, si avrà che bisogna di montare banchi di memoria a partire da  $0xE0000000$  fino a  $0xFFFFFFFF$ .

L'ingresso di select verrà quindi generato a partire dalla parte alta dell'indirizzo, fatto passare attraverso una certa maschera, facendo quindi corrispondere una certa impostazione dei bit più significativi (in questo caso 4) al segnale di select. Visto che avevamo detto si parte da  $0xE0000000$ , vogliamo  $0xE = B1110$ , quindi serve la rete combinatoria (maschera):

$$/s = \overline{a_{31}} + \overline{a_{30}} + \overline{a_{29}} + a_{28}$$

Con un montaggio di questo tipo possiamo usare il select per scegliere quale banco di memoria RAM è associato a quale parte dello spazio indirizzabile. Questo tipo di configurazione giustifica inoltre il fatto che il bit di select viene impostato con ritardo rispetto agli indirizzi: visto che deve essere calcolato sulla base di quest'ultimi, risentirà del ritardo della rete combinatoria che lo genera.

### 20.2 Memorie ROM a sola lettura

Le memorie a sola lettura, dette ROM, sono effettivamente reti combinatorie: l'uscita è costante qualunque siano gli stati passati. Vengono montate nello spazio di memoria assieme alla RAM, e rappresentano la parte **non volatile** (persistente) dello spazio di memoria stesso (abbiamo visto come contengono le prime istruzioni eseguite dal processore). La loro struttura interna ricalca quella della memoria RAM, privata della circuiteria necessaria alla lettura, e che usa generatori di costante al posto dei D-latch.

Una ROM può essere realizzata attraverso il MSU (*Modello Strutturale Universale*), collegando in maniera ortogonale file di OR alle uscite di un decoder (come ci è concesso dall'espansione di Shannon), nei punti in cui vogliamo che un'uscita corrisponda a un certo ingresso. Nella pratica, si usano più spesso porte NOR per ragioni elettroniche, e quindi si collegano le linee nel caso un'uscita corrisponde alla negazione dell'ingresso. Questo circuito viene solitamente stampato su un singolo chip di silicio, il cui costo fisso di progettazione è giustificato solo nel caso di produzione su larga scala. Conviene quindi realizzare delle alternative, le **ROM programmabili**.

#### 20.2.1 ROM programmabili

In una memoria ROM programmabile, le porte NOR sono tutte attaccate alle linee degli AND del decoder. Si possono disabilitare selettivamente alcune di queste porte NOR per effettivamente **programmare** la memoria contenuta nella ROM.

Possiamo individuare delle categorie per queste reti:

- Le **OTPROM** (*One Time Programmable ROM*) vengono realizzate attraverso questa tecnologia, è la loro programmazione risulta quindi **distruittiva** (una volta programmato un bit non si può più rimuovere).



- Una tecnologia più sofisticata è rappresentata dalle **EPROM** (*Erasable programmable ROM*). Queste vengono realizzate attraverso transistor a field-effect. La scrittura della EPROM può essere ripetuta sottoponendola a una luce ultravioletta, e quindi cancellando tutti i dati, per poi riscrivere altri.

Di una EPROM ci interessano:

- **Endurance:** quante riscritture successive può supportare (solitamente dalle 10K alle 100K volte);
- **Data retention:** il periodo per cui si può fare affidamento sui dati contenuti in una EPROM (solitamente dai 10 ai 100 anni).
- Infine, le **EEPROM** (*Electrically Erasable Programmable ROM*) permettono la riprogrammazione direttamente attraverso segnali elettrici, sul chip già montato nello spazio di memoria. Potremmo pensare che EEPROM e RAM sono effettivamente equivalente. Ci sono invece alcune differenze, che sono:
  - L'EEPROM è persistente, mentre la RAM è volatile;
  - Il numero di volte in cui si può riprogrammare una EEPROM è comunque limitato;
  - Il tempo di riprogrammazione di una EEPROM è maggiore del tempo di lettura della RAM;
  - Le tensioni che si usano nella programmazione di una EEPROM (12V-18V) sono maggiori dei 5V (o 3.3V) che richiede la RAM.

## 20.3 Verilog per reti sequenziali

Estendiamo ciò che abbiamo visto finora sul Verilog per coprire le reti sequenziali (in particolare temporizzazioni e registri). Ad esempio, il seguente frammento implementa un contatore in base 2 (che vedremo più avanti):

```

1 module b2_counter(eu, q, ei, clock, reset_);
2   input clock, reset_;
3   input ei;
4   output eu, q;
5   reg OUTR;
6   assign q = OUTR;
7   wire a;
8   assign {a, eu} = ({q, ei} == 'B00) ? 'B00:
9                  ({q, ei} == 'B10) ? 'B10:
10                 ({q, ei} == 'B01) ? 'B10:
11                 /*({q, ei} == 'B11)*/ 'B01;
12   always @(reset_ == 0) #1 OUTR <= 0;
13   always @(posedge clock) if (reset_==1) #2 OUTR <= a;
14 endmodule

```

Si parte con la dichiarazione di un modulo `b2_counter`, dove la sintassi "ad argomenti" indica le variabili che potremo usare come input o come output. Definiamo poi input e output esplicitamente, notando che si può usare (e anzi è consigliata) una separazione logica delle variabili su più righe. La parola chiave `reg` definisce poi un registro.

### 20.3.1 Assegnamenti

Una parte a noi particolarmente interessante del linguaggio è rappresentata dagli **assegnamenti**. Abbiamo fatto 3 assegnamenti nelle ultime 3 istruzioni dell'esempio prece-

dente, prima per implementare un contatore (con `assign`), e in seguito specificare il comportamento al reset e l'aggiornamento del registro (con `always`). Vediamo nel dettaglio tutte gli assegnamenti possibili e le loro differenze:

- **Assegnamenti procedurali:** vengono dichiarati con `initial` (per riferirsi a stati iniziali), o con `always` (per riferirsi a stati qualsiasi durante la simulazione). In blocchi `always`, si può specificare una condizione con il carattere `@`, fra cui ad esempio il `posedge` che ci permette di ricavare il positive edge di un segnale (il clock nell'esempio precedente). All'interno di assegnamenti procedurali abbiamo a disposizione più operatori:
  - **Assegnamenti bloccanti** (`=`): vengono eseguiti strettamente nell'ordine in cui vengono incontrati (cosa a noi solitamente indesiderata, in quanto vogliamo modellizzare le inaccuratezze delle temporizzazioni, che non sono mai simultanee e non hanno, se non specificato, un ordine preciso);
  - **Assegnamenti non bloccanti** (`<=`): vengono eseguiti in **parallelo**, cioè il valore non varia fino alla fine del blocco di assegnamento procedurale, e viene aggiornato con simultaneità al suo termine.
- **Assegnamenti continui:** dichiarati con `assign`, legano **strutturalmente** una variabile ad un'altra variabile, o a un costrutto `case`, o ancora ad una sintassi ricavata dagli operatori ternari come quella nell'esempio precedente. Il valore legato viene aggiornato ad ogni aggiornamento del valore a cui è legato: vanno effettivamente intesi come fisicamente connessi.

Notiamo come negli assegnamenti in Verilog vale un concetto simile a quello di `lvalue` e `rvalue` nel C: si assegna sempre a *sinistra*, sulla **LHS** (LeftHand Side) un valore a *destra*, sulla **RHS** (RightHand Side). Per fare un esempio, notiamo che non si può usare un `wire` come LHS (infatti con un `wire` ci aspetteremmo di usare una notazione strutturale, come quella data dagli assegnamenti continui). Di contro, non si può avere aggiornamento continuo di registri (anche perché non avrebbe particolare significato logico).

Infine, notiamo che si può inserire, negli assegnamenti procedurali, un **ritardo**, indicato con il simbolo `#` e misurato solitamente in secondi.

## 21 Lezione del 07-11-24

### 21.1 Reti sequenziali sincronizzate

Le reti sequenziali sincronizzate (RSS), a differenza della asincrone (RSA), non si aggiornano per la sola variazione degli ingressi, ma per l'arrivo di un determinato segnale periodico, che chiamiamo **clock**.

Il clock è un segnale con forma d'onda periodica, di frequenza  $\frac{1}{T}$  periodo, e *duty cycle* (ciclo di lavoro)  $\frac{\tau}{T}$  intorno al 50%. Solitamente l'evento di sincronia delle reti sequenziali sincronizzate è il **fronte di salita** del clock.

### 21.2 Registri

Un registro a  $W$  bit è una collezione di  $W$  D flip-flop positive edge-triggered, che hanno:

- $W$  ingressi  $d_i$  e  $W$  uscite  $q_i$  separate (in verità ricordiamo che troviamo sempre  $q$  e  $\bar{q}$  negata, noi riporteremo solo la prima per semplicità);

- Un ingresso  $p$  in parallelo a tutti gli ingressi  $p_i$  dei singoli D flip-flop.

Si ha che  $p$  funge da **segnale di sincronizzazione** (effettivamente il nostro *clock*). Consideriamo quindi le variabili di entrata e di uscita di un registro come due singole variabili a più bit,  $d_{W-1}d_0$  e  $q_{W-1}q_0$ .

### 21.2.1 Pilotaggio di registri

Per il corretto pilotaggio di un registro gli ingressi  $d_i$  devono essere stabili intorno al fronte di salita del clock, per un tempo  $T_{setup}$  prima e  $T_{hold}$  dopo il fronte stesso. L'uscita cambia dopo, come avevamo visto per i D flip-flop, un tempo  $T_{prop} > T_{hold}$ .

Tutto ciò che accade in ingresso fra due istanti di sincronizzazione è irrilevante e non viene memorizzato.

Il registro *memorizza* lo stato di ingresso al **fronte di salita**. Gli stati di ingresso fra due fronti di salita adiacenti possono essere identici, adiacenti o non adiacenti: è irrilevante in quanto, come abbiamo detto, l'aggiornamento accade soltanto nelle condizioni di stabilità intorno al fronte di salita del clock.

Dopo il fronte di salita, le uscite cambiano il loro valore dopo  $T_{prop}$ .

Possiamo quindi aggiornare la nostra definizione di RSS come *collezione di registri e reti combinatorie*, montati arbitrariamente, purchè non ci siano anelli di retroazione di reti combinatorie (costituirebbero reti sequenziali asincrone). I registri hanno tutti lo stesso clock in comune, e possono formare anelli, in quanto abbiamo visto dal loro pilotaggio, questo non genera problemi.

### 21.2.2 Regole di pilotaggio per RSS

Dato l' $i$ -esimo fronte di salita del clock al tempo  $t_i$ , lo stato di ingresso ai registri dovrà essere stabile, dalle loro regole di pilotaggio, nell'intervallo  $[t_i - T_{setup}, t_i + T_{hold}]$ . Non potrò quindi scegliere periodi  $T$  del clock piccoli a piacere: dovrò lasciare tempo ai registri di produrre nuovi valori (in tempo  $T_{prop}$ ), e alle reti combinatorie di elaborare tali valori coi loro tempi di ritardo interni, e quindi di propagarsi nuovamente fino ai registri.

Definiamo, nello specifico, i ritardi:

- $T_{in\_to\_reg}$ : il tempo di attraversamento massimo della catena di sole reti combinatorie che da uno degli ingressi della rete all'ingresso di un registro;
- $T_{reg\_to\_reg}$ : il tempo di attraversamento massimo fra l'uscita e l'ingresso di un registro;
- $T_{in\_to\_out}$ : il tempo di attraversamento massimo fra un ingresso e un uscita dell'intera rete;
- $T_{reg\_to\_out}$ : il tempo di attraversamento massimo fra l'uscita di un registro e un uscita della rete.

Dobbiamo introdurre poi i tempi  $T_{a\_monte}$  e  $T_{a\_valle}$ , cioè i tempi necessari all'utente della rete per, rispettivamente, **modificare** gli ingressi e **leggere** le uscite. Questi formano due ulteriori vincoli di pilotaggio in ingresso e in uscita.

Queste variabili di temporizzazione daranno vita ad un sistema di 4 disequazioni. Vediamole nel dettaglio:

- $T \geq T_{hold} + T_{a\_monte} + T_{in\_to\_reg} + T_{setup}$   
Questa disequaglianza assicura che un registro abbia tempo  $T_{hold}$  di immagazzinare il valore dello scorso ciclo, l'utente esterno abbia tempo  $T_{a\_monte}$  di modificare l'ingresso della rete, e che questo ingresso abbia tempo di arrivare ai registri  $T_{in\_to\_reg}$  prima del tempo di setup  $T_{setup}$  degli stessi, che sappiamo essere necessario perchè al clock seguente i registri memorizzino effettivamente il valore (dopo  $T_{hold}$ , e lo replichino dopo  $T_{prop}$ );
- $T \geq T_{prop} + T_{reg\_to\_reg} + T_{setup}$   
Questa disequaglianza assicura che il valore generato dai registri possa propagarsi dopo  $T_{prop}$ , arrivare ai registri stessi in tempo  $T_{reg\_to\_reg}$  per il  $T_{setup}$  necessario perche lo memorizzino. In sostanza, è come la precedente ma riferita alle uscite dei registri anziché dell'utente;
- $T \geq T_{hold} + T_{a\_monte} + T_{in\_to\_out} + T_{a\_valle}$   
Questa disequaglianza assicura che la rete abbia tempo di aggiornarsi dopo un ingresso dell'utente ( $T_{a\_monte}$ ), e restituire il risultato per un tempo che basti all'utente per leggere l'uscita ( $T_{a\_valle}$ ). Nello specifico, sappiamo che l'utente non proverà a modificare gli ingressi della rete prima del  $T_{hold}$  necessario ad aggiornare i registri al positive edge del clock, e quindi impiegherà un tempo  $T_{a\_monte}$  per farlo. A questo punto, il segnale di uscita dovrà viaggiare almeno dall'ingresso all'uscita, quindi si dovrà aspettare un tempo  $T_{in\_to\_out}$ , e infine resterà il tempo  $T_{a\_valle}$  perchè l'utente abbia modo di effettuare la lettura. Notiamo che questa legge si rende necessaria in quanto un aggiornamento degli ingressi può comportare un aggiornamento delle uscite *prima* che i registri ne rispondano. In altre parole, reti di questo tipo non sono automaticamente **trasparenti**;
- $T \geq T_{prop} + T_{reg\_to\_out} + T_{a\_valle}$   
Quest'ultima disequaglianza assicura che la rete abbia tempo di aggiornare le sue uscite, e quindi farle leggere all'utente ( $T_{a\_valle}$ ), a memorizzazione effettuata dei registri. Nello specifico, i registri otterranno il valore al ciclo corrente nel tempo compreso fra  $T_{setup}$  e  $T_{hold}$  centrato sul positive edge dello scorso clock, e quindi si adegueranno dopo un tempo  $T_{prop}$  rispetto al positive edge stesso. Di qui in poi dovremo aspettare un tempo  $T_{reg\_to\_out}$  perchè questo valore attraversi la rete fino alle uscite, e infine il tempo  $T_{a\_valle}$  perchè l'utente abbia modo di effettuare la lettura. Questa legge si rende necessaria, al contrario della precedente, sia per reti *trasparenti* che per reti **non trasparenti**, e anzi vedremo che reti non trasparenti saranno proprio i registri a fornire le uscite.

Possiamo quindi porre il sistema completo:

$$\begin{cases} T \geq T_{hold} + T_{a\_monte} + T_{in\_to\_reg} + T_{setup} \\ T \geq T_{prop} + T_{reg\_to\_reg} + T_{setup} \\ T \geq T_{hold} + T_{a\_monte} + T_{in\_to\_out} + T_{a\_valle} \\ T \geq T_{prop} + T_{reg\_to\_out} + T_{a\_valle} \end{cases}$$

Dove, riassumendo, le prime due condizioni garantiscono che lo stato delle variabili di ingresso resti stabile negli intervalli  $(-T_{setup}, T_{hold})$  centrati sui positive edge di ogni clock; la prima e la terza tengono conto del mondo esterno *a monte*, quindi in fase di scrittura; la seconda e la quarta tengono conto del mondo esterno *a valle*, quindi in fase di lettura.

In verità, avremo altri due ritardi di cui tenere conto:

- $T_{sfas}$ : il **massimo sfasamento** fra due clock. Visto che questo viene portato a elementi diversi, a qualche registro arriverà prima e a qualche registro arriverà dopo;
- $T_{reg}$ : se un registro è formato da  $W > 1$  bit, questi non cambieranno tutti contemporaneamente: dovremmo aggiungere  $T_{prop} + T_{reg} = T'_{prop}$ . A questo punto, però, possiamo considerare solo  $T_{prop} \leftarrow T'_{prop}$  e ignorare  $T_{reg}$ .

### 21.2.3 Anticipazioni sui modelli di Moore e di Mealy ritardato

Potremmo voler determinare qual'è la più vincolante fra le disuguaglianze riportate prima. Questa, chiaramente, è quella che copre il percorso più lungo, cioè la terza. Se decidiamo di vietare il percorso che copre, cioè quello diretto fra ingressi e uscite, otteniamo il cosiddetto **modello di Moore**: cioè, un modello di RSS dove non si ammettono reti combinatorie che collegano gli ingressi direttamente alle uscite.

Un'altro vincolo che potremmo voler rilassare è il quarto, nel cosiddetto **modello di Mealy ritardato**. Questo equivale a prelevare le uscite direttamente dalle uscite dei registri, cioè a eliminare il tempo  $T_{reg\_to\_out}$ .

## 21.3 Contatori

Un contatore è una RSS il cui stato di uscita può essere visto come un **numero naturale** ad  $n$  cifre in base  $\beta$ . Ad ogni clock, il contatore **incrementa o decrementa**.

Abbiamo che si può realizzare un contatore collegando un modulo sommatore a  $n$  cifre a un registro a  $n$  cifre. L'uscita del registro viene collegata in anello di retroazione a uno degli ingressi del sommatore. Impostando il  $C_{in}$  del sommatore a 1, e il suo secondo ingresso ad un'array di  $n$  generatori di costante 0, si ha un contatore **incrementatore**, cioè che incrementa il suo valore ad ogni ciclo di clock. L'equivalente **decrementatore** si può creare usando un sottrattore a  $n$  cifre invece di un sommatore. veri

Si può creare un contatore con ingresso di abilitazione (sostanzialmente una **variabile di controllo**), cioè che incrementa o decrementa solo se è alto un certo bit di controllo, collegando tale bit al carry (o al borrow) del sommatore (sottrattore).

## 21.4 Contatore a una cifra in base 2

Vediamo quindi come realizzare un contatore a una cifra in base  $\beta = 2$ . Se l'intenzione è di creare un contatore per  $N$  cifre in codifica binaria, questo rappresenterà l'elemento fondamentale (che andremo a combinare nei prossimi paragrafi, attraverso catene di **ripple carry**).

Avremo bisogno di un input,  $e_i$  oltre al clock e al reset, che rappresenterà il riporto entrante dell'incrementatore (che come abbiamo visto può fungere da variabile di controllo, in quanto lasciare  $e_i$  a 0 significa sommare 0 a un numero, quindi lasciarlo invariato), Prenderemo poi due uscite:  $q$ , cioè l'uscita vera e propria dal registro del contatore (che chiameremo  $OUTR$ ), e  $e_u$ , il riporto uscente. Nel caso di un contatore a una cifra in base 2, il riporto uscente si riduce al valore dell'AND fra  $q$  ed  $e_i$ , ergo se  $q$  è alto e introduciamo un riporto entrante  $e_i$ , andremo al di fuori della rappresentazione possibile su un bit e dovremo passare il riporto (che possiamo anche qui intendere come segnale di controllo) al prossimo contatore della catena. L'uscita dell'incrementatore (chiamiamola  $a$ ) andrà messa in  $OUTR$  ad ogni aggiornamento, e verrà aggiornata dai valori di  $q$ , cioè

l'uscita stessa di  $OUTR$  (ciclo di retroazione) e  $e_i$ , attraverso la logica dell'incrementatore che riportiamo sotto forma di tabella di verità:

$q$	$e_i$	$a$	$e_u$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Notiamo che non abbiamo mai l'uscita 1,1 su  $a, e_u$ , in quanto uno degli ingressi del sommatore che forma l'incrementatore è "fissato" a 0. Possiamo quindi dare la sintesi in Verilog:

```

1 // un contatore up a una cifra in base 2 che prende @ei come enabler
2 // e mette la sua uscita in @q, con eventuale riporto in @eu
3 module b2_up_counter(eu, q, ei, clock, reset_);
4     input clock, reset_;
5     input ei;
6     output eu, q;
7
8     reg OUTR;
9     assign q = OUTR;
10
11     wire a; // l'uscita dell'incrementatore
12     b2_halfadder inc (
13         .x(q), .cin(ei),
14         .s(a), .cout(eu)
15     );
16
17     always @(reset_ == 0) #1 OUTR <= 0;
18     always @(posedge clock) if (reset_==1) #2 OUTR <= a;
19 endmodule

```

sfruttando la definizione già data di incrementatore (cioè un *half adder*).

### 21.4.1 Contatore a una cifra in base 3

Con procedimenti simili a quelli dell'esempio precedente si può ricavare un contatore a una cifra in base  $\beta = 3$ . In questo caso la tabella di verità dell'incrementatore sarà:

$q_1$	$q_0$	$e_i$	$a_1$	$a_0$	$e_u$
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	0	0	1
1	1	0	-	-	-
1	1	1	-	-	-

e la sintesi sarà:

```

1 // un contatore up a una cifra in base 3 che prende @ei come enabler
2 // e mette la sua uscita in @q1_q0, con eventuale riporto in @eu
3 module b3_up_counter(eu, q1_q0, ei, clock, reset_);
4     input clock, reset_;

```

```

5  input ei;
6  output eu;
7  output [1:0] q1_q0;
8
9  reg [1:0] OUTR;
10 assign q1_q0 = OUTR;
11
12 wire [1:0] a1_a0; // l'uscita dell'incrementatore
13 b3_halfadder inc (
14     .x1_x0(q1_q0), .cin(ei),
15     .s1_s0(a1_a0), .cout(eu)
16 );
17
18 always @(reset_ == 0) #1 OUTR <= 0;
19 always @(posedge clock) if (reset_==1) #2 OUTR <= a1_a0;
20 endmodule

```

sfruttando la definizione già data di incrementatore in base 3.

Contatore a una cifra in base 10 Infine, vediamo come si può ricavare un contatore a una cifra in base  $\beta = 10$ . In questo caso la tabella di verità dell'incrementatore sarà:

$x_3$	$x_2$	$x_1$	$x_0$	$e_i$	$z_3$	$z_2$	$z_1$	$z_0$	$e_u$
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	1	0
0	0	0	1	0	0	0	0	1	0
0	0	0	1	1	0	0	1	0	0
0	0	1	0	0	0	0	1	0	0
0	0	1	0	1	0	0	1	1	0
0	0	1	1	0	0	0	1	1	0
0	0	1	1	1	0	1	0	0	0
0	1	0	0	0	0	1	0	0	0
0	1	0	0	1	0	1	0	1	0
0	1	0	1	0	0	1	0	1	0
0	1	0	1	1	0	1	1	0	0
0	1	1	0	0	0	1	1	0	0
0	1	1	0	1	0	1	1	1	0
0	1	1	1	0	0	1	1	1	0
0	1	1	1	1	1	0	0	0	0
1	0	0	0	0	1	0	0	0	0
1	0	0	0	1	1	0	0	1	0
1	0	0	1	0	1	0	0	1	0
1	0	0	1	1	0	0	0	0	1
1	0	1	0	0	-	-	-	-	-

ecc...

e la sintesi sarà:

```

1 // un contatore up a una cifra in base 10 che prende @ei come enabler
2 // e mette la sua uscita in @q3_q0, con eventuale riporto in @eu
3 module b10_up_counter(eu, q3_q0, ei, clock, reset_);
4     input clock, reset_;
5     input ei;
6     output eu;
7     output [3:0] q3_q0;
8

```

```

9  reg[3:0] OUTR;
10 assign q3_q0 = OUTR;
11
12 wire[3:0] a3_a0; // l'uscita dell'incrementatore
13 b10_halfadder inc (
14     .x3_x0(q3_q0), .cin(ei),
15     .s3_s0(a3_a0), .cout(eu)
16 );
17
18 always @(reset_ == 0) #1 OUTR <= 0;
19 always @(posedge clock) if (reset_==1) #2 OUTR <= a3_a0;
20 endmodule

```

sfruttando la definizione già data di incrementatore in base 10.

### 21.4.2 Scomposizione in moduli di contatori

Un contatore può essere scomposto, in qualsiasi base, in una serie di contatori ad una cifra collegati a **catena di riporti** (*ripple carry*). In questo caso il registro è dato dalla combinazione di  $n$  registri, uno per ogni cifra (e quindi per ogni contatore), tutti sincronizzati sullo stesso clock. All'attivarsi del riporto di uno dei contatori, il successivo nella catena si attiva, e così via in un processo del tutto identico al propagarsi delle somme nei sommatore a più cifre. Si riportano le descrizioni in Verilog di sommatore a quattro cifre, in base  $\beta = 2$ :

```

1 // un contatore up a 4 cifre in base 2 che prende @ei come enabler e
2 // mette la sua uscita in @q3_q0, con eventuale riporto in @eu
3 module n4_b2_up_counter(eu, q3_q0, m_ei, m_clock, m_reset_);
4     input m_clock, m_reset_;
5     input m_ei;
6
7     wire[3:0] eu3_eu0;
8
9     output eu;
10    assign eu = eu3_eu0[3];
11    output[3:0] q3_q0;
12
13    b2_up_counter count0 (
14        .clock(m_clock), .reset_(m_reset_),
15        .ei(m_ei),
16        .eu(eu3_eu0[0]), .q(q3_q0[0])
17    );
18
19    b2_up_counter count1 (
20        .clock(m_clock), .reset_(m_reset_),
21        .ei(eu3_eu0[0]),
22        .eu(eu3_eu0[1]), .q(q3_q0[1])
23    );
24
25    b2_up_counter count2 (
26        .clock(m_clock), .reset_(m_reset_),
27        .ei(eu3_eu0[1]),
28        .eu(eu3_eu0[2]), .q(q3_q0[2])
29    );
30
31    b2_up_counter count3 (
32        .clock(m_clock), .reset_(m_reset_),
33        .ei(eu3_eu0[2]),
34        .eu(eu3_eu0[3]), .q(q3_q0[3])

```



```

35 );
36 endmodule

    in base  $\beta = 3$ :

1 // un contatore up a 4 cifre in base 3 che prende @ei come enabler e
2 // mette la sua uscita in @q31_q30, @q21_q20, @q11_q10 e @q01_q00,
3 // con eventuale riporto in @eu
4 module n4_b3_up_counter(eu, q31_q30, q21_q20, q11_q10, q01_q00, m_ei,
5                               m_clock, m_reset_);
6     input m_clock, m_reset_;
7     input m_ei;
8
9     wire [3:0] eu3_eu0;
10
11     output eu;
12     assign eu = eu3_eu0[3];
13
14     output [1:0] q01_q00;
15     output [1:0] q11_q10;
16     output [1:0] q21_q20;
17     output [1:0] q31_q30;
18
19     b3_up_counter count0 (
20         .clock(m_clock), .reset_(m_reset_),
21         .ei(m_ei),
22         .eu(eu3_eu0[0]), .q1_q0(q01_q00)
23     );
24
25     b3_up_counter count1 (
26         .clock(m_clock), .reset_(m_reset_),
27         .ei(eu3_eu0[0]),
28         .eu(eu3_eu0[1]), .q1_q0(q11_q10)
29     );
30
31     b3_up_counter count2 (
32         .clock(m_clock), .reset_(m_reset_),
33         .ei(eu3_eu0[1]),
34         .eu(eu3_eu0[2]), .q1_q0(q21_q20)
35     );
36
37     b3_up_counter count3 (
38         .clock(m_clock), .reset_(m_reset_),
39         .ei(eu3_eu0[2]),
40         .eu(eu3_eu0[3]), .q1_q0(q31_q30)
41     );
42 endmodule

```

e infine in base  $\beta = 10$ :

```

1 // un contatore up a 4 cifre in base 10 che prende @ei come enabler
2 // e mette la sua uscita in @q33_q30, @q23_q20, @q13_q10 e @q03_q00,
3 // con eventuale riporto in @eu
4 module n4_b10_up_counter(eu, q33_q30, q23_q20, q13_q10, q03_q00,
5                               m_ei, m_clock, m_reset_);
6     input m_clock, m_reset_;
7     input m_ei;
8
9     wire [3:0] eu3_eu0;
10
11     output eu;
12     assign eu = eu3_eu0[3];

```

```

13
14 output[3:0] q03_q00;
15 output[3:0] q13_q10;
16 output[3:0] q23_q20;
17 output[3:0] q33_q30;
18
19 b10_up_counter count0 (
20     .clock(m_clock), .reset_(m_reset_),
21     .ei(m_ei),
22     .eu(eu3_eu0[0]), .q3_q0(q03_q00)
23 );
24
25 b10_up_counter count1 (
26     .clock(m_clock), .reset_(m_reset_),
27     .ei(eu3_eu0[0]),
28     .eu(eu3_eu0[1]), .q3_q0(q13_q10)
29 );
30
31 b10_up_counter count2 (
32     .clock(m_clock), .reset_(m_reset_),
33     .ei(eu3_eu0[1]),
34     .eu(eu3_eu0[2]), .q3_q0(q23_q20)
35 );
36
37 b10_up_counter count3 (
38     .clock(m_clock), .reset_(m_reset_),
39     .ei(eu3_eu0[2]),
40     .eu(eu3_eu0[3]), .q3_q0(q33_q30)
41 );
42 endmodule

```

## 22 Lezione del 08-11-24

### 22.1 Contatori e divisione in frequenza

Si ha che i contatori *contano* i cicli di clock a cui sono sottoposti (cioè incrementano per ogni ciclo). Possiamo usare un contatore per **dividere** la frequenza del clock per un dato valore. Ad esempio, il MSB di un contatore in base 3 va 3 volte più lento del clock che lo pilota. In generale, si ha che per un contatore a  $N$  cifre in base 2 che riceve clock a periodo  $T$ , l'MSB è a periodo  $2^N \cdot T$ .

Si potrebbe pensare di usare l'uscita di riporto del contatore in MSB come uscita divisa del clock: questo non è raccomandabile in quanto l'uscita di riporto è un **uscita combinatoria**, che non è né **stabile** né a **temporizzazione certa** (a differenza dell'uscita di un registro).

### 22.2 Registro multifunzionale

Un registro multifunzionale è una rete che, all'arrivo del clock, memorizza nello stesso registro una tra  $K$  funzioni combinatorie possibili, scelte impostando un certo numero di variabili di comando  $W = \log_2 K$ .

L'implementazione effettiva del registro è data da un multiplexer da 0 a  $K - 1$  reti combinatorie, dove  $W$  è la variabile di comando, la cui uscita viene inviata a un certo registro (che spedisce poi la sua uscita in retroazione alle reti combinatorie funzionali, e così via).

## 22.3 Modello di reti sequenziali sincronizzate

Definiamo tre modelli per le RSS (che abbiamo già introdotto parlando delle regole di pilotaggio):

### 22.3.1 Modello di Moore

Una RSS di Moore è definita a partire da:

- Un insieme di  $N$  variabili logiche di ingressi;
- Un insieme di  $M$  variabili logiche di uscita;
- Un **meccanismo di marcatura**, che a ogni istante marca uno **stato interno presente**, scelto fra  $K$  finito stati interni  $S \equiv \{S_0, \dots, S_{K-1}\}$ ;
- Una legge di evoluzione nel tempo  $A : X \times S \rightarrow S$ , che mappa una coppia, data da un  $X$  stato di ingresso e un elemento  $s \in S$  stato interno, ad un nuovo stato interno (diciamo  $s' \in S$ );
- Una legge di evoluzione nel tempo  $B : S \rightarrow Z$ , che mappa uno stato interno  $s \in S$  a uno stato di uscita  $Z$ .

La rete riceve **segnali di sincronizzazione**, come ad esempio le transizioni da 0 a 1 del segnale di clock (avevamo detto il *leading edge*). La legge di temporizzazione di una RSS di Moore è quindi la seguente: dato un elemento  $s \in S$ , stato interno marcato ad un certo istante, e dato  $X$  ingresso ad un certo istante immediatamente precedente l'arrivo di un segnale di sincronizzazione:

1. Si individua il nuovo stato interno da marcare  $s' = A(X, s)$ ;
2. Si aspetta  $T_{prop}$  dopo l'arrivo del segnale di sincronizzazione;
3. Si promuove  $s'$  al rango di **stato interno marcato**.

Lo stato interno marcato viene memorizzato in un apposito registro, detto **STAR** (da *STatus Register*). Questo viene implementato con una batteria di D flip-flop non trasparenti.

In una RSS di Moore si hanno quindi i seguenti vincoli di pilotaggio:

$$\begin{cases} T \geq T_{hold} + T_{a\_monte} + T_A + T_{setup} \\ T \geq T_{prop} + T_A + T_{setup} \\ T \geq T_{prop} + T_Z + T_{a\_valle} \end{cases}$$

che riguardano rispettivamente i tempi ingresso-STAR, STAR-STAR e STAR-uscita.

### 22.3.2 Flip-flop JK

Un'esempio di RSS di Moore è il flip-flop JK, che valuta due ingressi  $j$  e  $k$  e si comporta come segue:

Un modo di vedere questa rete è come un registro multifunzionale ad un bit, con tabella di applicazione:

quando puoi fai un ripassone

$j$	$k$	Azione
0	0	Conserva
1	0	Setta
0	1	Resetta
1	1	Commuta

$q$	$q'$	$j$	$k$
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

Vediamone la sintesi: visto che conosciamo soltanto la sintesi di reti combinatorie attraverso le mappe di Karnaugh, basterà sintetizzare le due reti combinatorie, dalla definizione di rete di Moore. **RCA** e **RCB** che implementano le funzioni  $A$  e  $B$ . Il registro STAR conterrà a questo punto lo stato interno, che nel caso del flip-flop JK ridurrà RCB a un cortocircuito per ogni uscita del registro.

Per la sintesi di RCA, invece, dovremo consultare la tabella di flusso: riportala e applica Karnaugh

## 23 Lezione del 12-11-24

### 23.1 Riconoscitore di sequenze

Un riconoscitore di sequenze è una rete sequenziale sincronizzata a  $N$  ingressi ed un uscita. Questa rete si evolve secondo la legge seguente: se si presenta, in sequenza, una sequenza di stati di ingresso voluta, l'uscita vale 1, 0 altrimenti.

Ogni stato di ingresso deve essere presentato prima del prossimo ciclo di clock, e per  $n$  stati di ingresso avremo bisogno di  $n$  cicli di clock per leggerli tutti. Inoltre, se un valore permane per più di un ciclo di clock, si considera questa una ripetizione.

Si ha che questa è una **rete con memoria**: deve ricordare ad ogni stato di ingresso la sequenza degli stati di ingresso **corretti** e **consecutivi** visti finora, cioè  $K + 1$  stati (compreso lo stato finale con l'uscita a 1) per sequenze di  $K$  stati.

la sintesi è un esercizio per casa

### 23.2 Modello di Mealy

Nel modello di Moore avevamo detto che l'uscita è funzione soltanto dello stato interno precedente:  $B : S \rightarrow Z$ . Nelle reti di mealy, la legge  $B$  è più generale, e dipende anche dagli ingressi:  $B : X \times S \rightarrow Z$ .

Vediamo che le reti RCA e RCB, a questo punto, possono essere espresse come un'unica grande rete RC con cicli di retroazione dal registro STAR. Possiamo quindi riformulare le disuguaglianze di temporizzazione come:

$$\begin{cases} T \geq T_{hold} + T_{a\_monte} + T_{RC} + T_{setup} \\ T \geq T_{prop} + T_{RC} + T_{setup} \\ T \geq T_{hold} + T_{a\_monte} + T_{RC} + T_{a\_valle} \\ T \geq T_{prop} + T_{RC} + T_{a\_valle} \end{cases}$$

dove i tempi di attraversamento da ingresso a registro, da registro a registro e da registro a uscita sono sostituiti dal tempo di attraversamento di RC come mai?

Notiamo che al variare dell'ingresso, una rete di Mealy può produrre una nuova uscita *prima dell'aggiornamento del clock*. Questo rende le reti di Mealy **non trasparenti**: gli ingressi sono connessi direttamente alle uscite (in senso logico), ergo cicli di retroazione possono creare oscillazioni incontrollate. Prendiamo ad esempio il contatore visto precedentemente: è effettivamente una **rete di Mealy** rispetto alle uscite di riporto *eu* e *ei*. L'uscita *q*, invece, che è collegata direttamente al registro, non è trasparente (la chiamiamo *uscita di Moore*). Notiamo quindi che basta un'uscita di Mealy a rendere una rete una rete di Mealy.

## 24 Lezione del 14-11-24

### 24.1 Riconoscitore di sequenze

Vediamo come sintetizzare un circuito **riconoscitore di sequenze**, sia come rete di Moore che come rete di Mealy:

#### 24.1.1 Sintesi in rete di Moore

fallo

#### 24.1.2 Sintesi in rete di Mealy

fallo

### 24.2 Confronto fra Moore e Mealy

Abbiamo che le reti di Moore hanno leggi  $B$  meno flessibili delle reti di Mealy, e quindi in una rete di Mealy si hanno meno stati interni che in una rete di Moore. Si potrebbe quindi pensare che una rete di Mealy può esprimere funzioni che una rete di Moore non può rappresentare. Questo è falso, in quanto si può dimostrare che Moore e Mealy hanno la **stessa potenza descrittiva**: per una rete di Moore, si può ricavare l'equivalente di Mealy, e viceversa.

Tra le altre differenze che possiamo notare, si ha che il clock di una rete di Moore al pari di una rete di Mealy deve essere più veloce, e soprattutto che una rete di Mealy si aggiorna *al pari* con gli ingressi, cioè è una rete **trasparente**.

Un anello di retroazione fra due reti di Mealy può infatti creare un **anello combinatorio**, che sappiamo essere suscettibile a oscillazioni incontrollate. Di contro, fra due reti di Moore incontreremo sempre un registro, ergo non avremo problemi di formazione di anelli combinatori.

### 24.3 Modello di Mealy ritardato

Creiamo una cosiddetta rete di **Mealy ritardato** prendendo una rete di Mealy e introducendo un ulteriore registro, **OUTR**, in uscita. Le uscite, come nelle reti di Moore, non sono più trasparenti e variano all'arrivo del clock dopo un tempo  $T_{prop}$ .

rimetti in ordine da Mealy

### 24.3.1 Temporizzazione del modello di Mealy ritardato

riporta

## 24.4 Assegnamenti procedurali

leggi max chiedi foto simone

In Verilog possiamo descrivere il comportamento di una rete di Mealy attraverso i cosiddetti **assegnamenti procedurali**. Notiamo la temporizzazione di una forma del tipo:

```
1 s0: begin STAR<=S1; OUTR<=STAR; end
```

Le istruzioni contenute nel blocco `begin (...)` `end` accadono **contemporaneamente**, e il fatto che STAR sia a sinistra nel primo assegnamento e OUTR sia sinistra nel secondo indica che questi avvengono *dopo il clock*, cioè *prima del clock* si legge il valore (S1 o STAR nell'esempio) e soltanto dopo si scrive effettivamente sul registro.

finisci slide

## 24.5 Reti sequenziali complesse

I modelli concettuali che abbiamo visto finora (Moore, Mealy e Mealy ritardato) riescono a sintetizzare solo reti molto semplici. Prendiamo ad esempio il modello di Mealy ritardato.

Vogliamo creare una rete che conta, modulo 16, il numero di sequenze corrette 00, 01, 10 ricevute in ingresso. Quindi, ogni volta che viene registrata una sequenza corretta, la rete incrementa di 1 l'uscita, rappresentata su 4 bit. Abbiamo quindi **2 ingressi, 4 uscite**, e 3 stati interni per 16 stati di OUTR, cioè 48 stati interni totali.

Modellizzare questa rete con un tale numero di ingressi risulta chiaramente molto laborioso: un approccio migliore sarebbe creare una rete di Mealy ritardato che riconosce una sola sequenza, e mandarla in input a un contatore a 4 bit. potresti fare disegnini però è laborioso pure quello

*Esplodendo* una rete siffatta troviamo un modello formato da una rete di Mealy ritardato (cioè da una rete combinatoria di *riconoscimento* nel caso del riconoscitore di sequenza, annessa ai registri STAR e OUTR), e da un contatore (formato a sua volta da una rete combinatoria che implementa la logica del contatore e un altro registro, che chiameremo **COUNT**). L'uscita di questa rete sarà formata dal registro COUNT, e potremo inoltre racchiudere le due reti combinatorie in un'unica RC totale.

La rete così ottenuta non rispetta il modello di Mealy ritardato: ha più di due registri, e soprattutto fa rientrare nella RC totale più di un registro uscente (finora era stato STAR). Troviamo che questo è molto comodo: introducendo **registri operativi** abbiamo a disposizione locazioni di memoria che supportano sia uscite che computazioni intermedie.

Possiamo quindi distinguere i registri in due categorie:

- **Registri di stato:** simili a quelli che abbiamo già visto, cioè che rappresentano lo stato interno della rete;
- **Registri operativi:** che contengono sia *valori di uscita* che *valori intermedi* (o *computazioni intermedie*, insomma risultati utili al ricavo dell'uscita della rete).

## 25 Lezione del 19-11-24

### 25.0.1 Microprogrammazione

Avevamo visto il concetto basilare di **rete sequenziale sincronizzata complessa**. La sintesi di reti di questo tipo prende il nome di **microprogrammazione**. Bisogna notare che la parola *programmazione* qui è piuttosto fuorviante: l'idea è comunque quella di dare **descrizioni** di hardware.

Il Verilog comprende un sottoinsieme di linguaggio adibito esattamente agli scopi della microprogrammazione, cioè un **linguaggio di trasmissione tra registri**. In ogni statement includiamo:

- $\mu$ -istruzioni, cioè assegnamenti a registri operativi;
- $\mu$ -salti, cioè assegnamenti al registro STAR, che possono essere a una o più vie.

Notiamo che qui il  $\mu$  significa semplicemente "hardware". Si possono omettere le  $\mu$ -istruzioni relative a *registri operativi*, che in questo caso implicano conservazione. Per quanto riguarda il registro STAR, invece, l'omissione del  $\mu$ -salto implicherebbe un salto incondizionato (altrimenti avremmo  $STAR \leftarrow STAR$ , che porterebbe a un **deadlock**). Nella pratica, non ometteremo mai l'aggiornamento di STAR, in quanto porta facilmente a errori, o comunque a codice poco chiaro.

### 25.0.2 Temporizzazione di reti complesse

Le temporizzazioni di una rete complessa sono le stesse delle reti di Mealy ritardato: i percorsi sono gli stessi ( $T_{in\_to\_reg}$ , ecc...) e preleviamo le uscite direttamente dai registri.

## 25.1 Handshake e temporizzazione delle uscite

Solitamente le reti sequenziali sincronizzate comunicano con altre reti sequenziali sincronizzate, idealmente con cicli di clock mutualmente sincronizzati (così da doverci preoccupare solo dei tempi di lettura e scrittura  $T_{a\_monte}$  e  $T_{a\_valle}$ ). Nel caso le reti che consideriamo siano invece mutuamente asincrone fra di loro, cioè abbiano clock discordi che non si allineano mai (necessariamente), dovremmo adottare soluzioni diverse.

Poniamo una situazione di esempio: una rete, detta **produttore**, mette su una linea un numero rappresentato su 8 bit. Chiamiamo questa linea *linea di trasmissione*. Un'altra rete, detta **consumatore**, riceve questo numero e tiene la sua uscita alta per il numero di cicli indicato dal numero ricevuto. Visto che le due reti vedono solamente i rispettivi input e output, come facciamo in modo che il produttore sappia quando il consumatore ha letto il numero con successo, e viceversa che il consumatore sappia quando c'è un nuovo numero da leggere? Il problema si risolve introducendo **linee di handshake** (dall'inglese per *stretta di mano*).

Doteremo quindi il produttore di una linea di uscita  $/dav$  (*data valid*), e il consumatore di una linea di uscita  $/rfd$  (*ready for data*). La linea  $/dav$  segnala che ci sono nuovi dati sulla linea di uscita del produttore, mentre la linea  $/rfd$  segnala che il consumatore ha letto ciò che era sulla linea di trasmissione ed è pronto a ricevere nuovi dati. Entrambe le variabili sono attive basse.

Facciamo una nota sulla circuiteria di **reset**: la linea  $/reset$  sarà infatti presente e comune alle due reti. A  $/reset$  bassa, quindi, possiamo mettere  $/dav$  e  $/rfd$  a 1 (per segnalare che linea di trasmissione non è pronta e il consumatore non è pronto a leggerla).

In seguito, vorremo effettuare una trasmissione vera e propria di dati. Innanzitutto, il produttore metterà un numero sulla linea di trasmissione. In seguito, metterà  $/dav$  bassa per segnalare che i dati sulla linea di trasmissione sono pronti. A questo punto, il consumatore dovrà rilevare il  $/dav$ , leggere i dati sulla linea di trasmissione e mettere il suo  $/rfd$  basso. Questo segnerà, per la rete produttore, che il consumatore **ha letto** i dati con successo ed è pronto ad un nuovo ciclo di trasmissione. Da qui in poi, il consumatore non potrà più aspettarsi che i dati sulla linea di trasmissione siano validi: in qualsiasi momento il produttore potrebbe aggiornarli e rialzare  $/dav$  (o viceversa, rialzare  $/dav$  e poi scrivere sulla linea, ciò che importa è che il consumatore non ha più nulla da leggere fino a un nuovo ciclo di abbassamento del  $/dav$ ). Quando  $/dav$  si alza, quindi, anche il consumatore riporterà il suo  $/rfd$  alto, e ci troveremo nella situazione di partenza (cioè  $/dav$  e  $/rfd$  alti). Dobbiamo stare attenti al fatto che  $/rfd$  torna alto **dopo** che lo fa  $/dav$ : altrimenti il produttore potrebbe perdersi la doppia transizione di  $/rfd$ , e finiremmo in uno stato di deadlock. Possiamo riassumere quest'ultima affermazione come segue: una corretta sincronizzazione delle reti avviene **solamente** se si **alternano** le transizioni delle linee di handshake (a ogni aggiornamento del produttore segue un aggiornamento del consumatore, e così via, senza che nessuno faccia doppi aggiornamenti "di testa propria").

Veniamo quindi all'implementazione pratica della rete consumatore. Vogliamo che questa mantenga un uscita bassa per un numero di cicli fornito sulla linea di trasmissione. Questo si realizza con un **contatore**.

## 26 Lezione del 21-11-24

### 26.1 Handshake soc-eoc

Abbiamo visto l'handshake  $dav-rfd$ . Vediamo adesso un altro tipo di handshake, detto **handshake soc-eoc** (*Start Of Computation, End Of Computation*). Nell'handshake soc-eoc, è il consumatore a fare la prima mossa, cioè a segnalare al produttore che bisogna di un nuovo dato.

La situazione di riposo è quella dove  $soc$  è 0 e  $eoc$  è 1. In questo caso, l'ultimo dato è già sulla linea di trasmissione, quindi possiamo leggerlo liberamente. Quando il consumatore richiede un nuovo dato, mette  $soc$  a 1. A questo punto il produttore metterà  $eoc$  a 0: questo significa che la computazione è in corso e non possiamo leggere dalla linea di trasmissione. Una volta rilevato  $eoc$  a 0, il consumatore dovrà rimettere  $soc$  a 0, per rispettare l'alternanza dei segnali di handshake. Al rialzarsi di  $eoc$ , quindi, il nuovo dato sarà pronto e potremo leggerlo dalla linea di trasmissione,  $soc$  sarà a 0 da prima e ci troveremo quindi nuovamente nella situazione di riposo.

### 26.2 Sintesi di RSS complesse

Finora abbiamo visto (se uno ce le mettesse) descrizioni scritte attraverso il **linguaggio di trasferimento fra registri** del Verilog. Adesso dobbiamo vedere di come *sintetizzare* effettivamente una rete complessa, attraverso **circuiti elementari** noti. Notiamo che non parleremo di *sintesi ottime*, in quanto questo non è un problema che adesso ci poniamo.

Vediamo quindi come operiamo nella pratica: dovremo operare scomposizione in **parte operativa** e **parte di controllo**:

- **Parte operativa:** contiene la logica necessaria all'interfacciamento col mondo esterno e alla produzione di stati di ingresso per i **registri operativi**;



- **Parte di controllo:** si occupa di mantenere aggiornato lo stato interno.

Queste due reti hanno lo stesso clock, e comunicano fra di loro attraverso due gruppi di variabili: quelle di **comando** ( $PC \rightarrow PO$ ), e quelle di **condizionamento** ( $PO \rightarrow PC$ ).

Per realizzare la **parte operativa** dobbiamo pensare ai registri operativi come **registri multifunzionali**. Per ogni registro isoliamo le  $\mu$ -operazioni diverse, una per ogni stato, che dovremo effettuare in modo da ricavare il suo ingresso, e le inseriamo in una rete combinatoria con in coda multiplexer guidati da variabili di comando (che sono sia variabili di controllo dei multiplexer e variabili di comando della parte operativa).

Dalla parte operativa genereremo le variabili di condizionamento, attraverso un suo sottoinsieme detto **rete combinatoria di condizionamento**, che useremo per sintetizzare la parte di controllo. La rete combinatoria di condizionamento prende come ingressi le variabili di ingresso della RSS e lo stato dei registri operativi (che in generale sono visibili alla parte operativa). Notiamo che la parte operativa rappresenta effettivamente una rete di Mealy, in quanto alcune uscite escono da registri (registri multifunzionali), e altre vengono direttamente da reti combinatorie connesse agli ingressi (variabili di condizionamento).

La **parte di controllo** si occupa invece di gestire, sostanzialmente, il registro STAR. Prende in ingresso le variabili di condizionamento appena generate, e restituisce in uscita le variabili di comando dei multiplexer (credo) della parte operativa. Questo si può implementare effettivamente come un RSS di Moore, dove le uscite dei registri rappresentano le variabili di comando stesse.

Notiamo quindi di aver ricondotto l’RSS a due reti sequenziali, una di Moore e una di Mealy, in retroazione fra di loro. Questo, come abbiamo visto, si può fare in quanto una di loro è di Moore.

## 27 Lezione del 22-11-24

### 27.1 Microprogrammazione della parte di controllo

Le tecniche di microprogrammazione ci permettono di sintetizzare la parte di controllo di reti complesse. In particolare, associata una codifica ad ogni stato del registro STAR, e chiamata questa codifica per ogni STAR  **$\mu$ -indirizzo** dello stato, possiamo creare una tabella:

$\mu$ -indirizzo	$\mu$ -codice	$\mu$ -istruzione		
		$c_{eff}$	$\mu$ -indirizzo T	$\mu$ -indirizzo F
00	00	1	00	01
01	11	0	10	01
...				

dove si associa ad ogni  $\mu$ -indirizzo, quindi ad ogni stato, una  **$\mu$ -istruzione**: intendiamo una  $\mu$ -istruzione come un insieme di variabili di comando associate a quello stato (il  **$\mu$ -codice**), una **variabile efficiente**  $c_{eff}$  sulla base della quale si effettuano i  $\mu$ -salti, e due  $\mu$ -indirizzi,  $T$  e  $F$ , che determinano il salto successivo sulla base del valore, rispettivamente vero o falso, della  $c_{eff}$ .

A partire da una tabella del genere, possiamo sintetizzare la PC secondo due modalità:

- **Modello basato sui  $\mu$ -indirizzi:** mi sa non riesco ora

- **Modello basato sulle  $\mu$ -istruzioni:**

### 27.1.1 Reintrodurre i salti a più vie

Nel caso di salti a più di due vie, si dovranno considerare più condizioni in cicli di clock differenti. fai esempio Questo diventa poco efficiente quando i salti sono a un elevato numero di vie, in quanto per  $n$  possibili  $\mu$ -salti si perdono  $\sim n$  cicli di clock.

Nella pratica, i processori sono spesso progettati per compiere salti a due vie, tranne che in due casi particolari:

- All'**inizio** della fase di fetch, cioè quando si legge il **formato** dell'opcode, dove si dovrà saltare a un blocco  $\mu$ -codice diverso a seconda della posizione degli operandi fai esempio assembler. Si perderanno quindi  $\sim f$  cicli per  $f$  formati possibili delle istruzioni;
- Alla **fine** della fase di fetch, cioè quando si determina il salto al blocco di  $\mu$ -codice che gestisce la **fase di esecuzione** dell'istruzione. Si perderanno quindi  $\sim i$  cicli per  $i$  istruzioni possibili.

Una soluzione al problema dei salti a più vie è data quindi dal **Multiway Jump Register**.

### 27.1.2 Multiway Jump Register

L'MJR non è un gruppo punk americano ma un **registro operativo** destinato a contenere indirizzi di salto. Generiamo l'ingresso del MJR attraverso la parte operativa della sintesi, e lo utilizziamo nella parte di controllo.

Per codificare la presenza del MJR, nella ROM della sintesi della parte di controllo dovremmo introdurre una nuova uscita, il  **$\mu$ -tipo**. Il valore del  $\mu$ -tipo determina il tipo di salto che vorremo eseguire:  $\mu$ -tipo a 0 significherà salto standard a 2 vie, e  $\mu$ -tipo a 1 significherà salto basato sul MJR.

parla del b\_k di sovrascrittura del MJR

## 27.2 Sottoliste

Talvolta può convenire strutturare una descrizione di RSS con sottoliste simili a **sotto-programmi**. Porzioni di  $\mu$ -programma diverse potranno quindi essere raggiunte da stati di partenza diversi, che riporteranno allo stato di partenza stesso al termine della loro esecuzione attraverso un processo simile a quello delle CALL e RET viste sull'assembly ( integra). Questo può essere implementato nella pratica, inserendo il  $\mu$ -indirizzo successivo all'esecuzione della sottolista nel MJR, cioè impostando  $b_k$  a 0 per quello stato, e inserendo quindi il  $\mu$ -indirizzo dell'inizio della sottolista in STAR. A questo punto la rete di controllo "eseguirà" il  $\mu$ -codice ed effettuerà i  $\mu$ -salti specificati dalla sottolista fino al passo finale, che rimetterà MJR in STAR, e quindi riprenderà l'esecuzione dal  $\mu$ -indirizzo memorizzato prima della "chiamata" della sottolista.

Due limitazioni di questo approccio sono che MJR diventa inutilizzabile durante l'esecuzione della sottolista, e soprattutto che un singolo MJR ci permette un solo livello di annidamento di sottoliste. Per avere più livelli avremo bisogno di una **pila di MJR**, che però non è trattata in questo corso.

### 27.3 Struttura del calcolatore

Siamo arrivati ora a poter descrivere in Verilog un **sistema completo** di:

- Processore;
- Memoria;
- Interfacce;
- Dispositivi di I/O

collegati fra di loro attraverso una rete di interconnessione.

All'interno del **sottosistema di ingresso/uscita** distinguiamo **interfacce** e **dispositivi**. Gli ultimi si occupano effettivamente di ottenere codifiche di dati dal mondo esterno, mentre le prime gestiscono i dispositivi in modo che questi possano colloquiare col processore. Le interfacce contengono un piccolo numero di **registri di interfaccia** su cui il processore può leggere o scrivere.

La **memoria principale** sarà formata in larga parte da memoria RAM, e conterrà in ogni istante le **istruzioni** e i **dati** che questo elabora. Una parte della memoria principale dovrà essere implementata attraverso memoria ROM, in quanto c'è da risolvere il problema dello stato di avvio del processore introducendo dati predefiniti che vengono puntati per primi dall'Instruction pointer. (credo) Il modello che andremo a studiare poi sarà dotato di memoria video, che conterrà le immagini visualizzate sullo schermo, e sarà anch'essa in diretta comunicazione col processore.

Il **processore** eseguirà il ciclo **fetch-execute**, prelevando dalla memoria principale **istruzioni operative** e **istruzioni di controllo**. Dovrà partire in una determinata configurazione dei registri, ottenuta collegando opportunamente piedini di `/preset` e `/preclear` alla linea di `/reset`, in modo da inizializzare (come detto prima) l'Instruction pointer a puntare ad una locazione di memoria nota che lanci un determinato programma in memoria, detto **bootstrapper**.

Per quanto ci riguarda, il calcolatore sarà formato da una serie di RSS, e il processore potrà essere sintetizzato attraverso la separazione PO/PC.

#### 27.3.1 Memoria

La nostra memoria sarà formata da uno spazio lineare di  $2^{24}$  locazioni di memoria da un byte, per un totale di 16 MB indirizzati su 24 bit (3 byte). Lo spazio di I/O sarà formato da uno spazio lineare di  $2^{16}$  locazioni di memoria da un byte, per un totale di 64 B indirizzati su 16 bit (2 byte).

#### 27.3.2 Processore

Il processore sarà dotato di 3 tipi di registri:

- **Registri accumulatore:** AH e AL, da 8 bit ciascuno;
- **Registro dei flag:** 8 bit con 4 significativi: CF (0), ZF (1), SF (2), OF (3);
- **Registri puntatore:** da 24 bit (3 byte) ciascuno (devono contenere indirizzi di memoria), sono:
  - **IP:** l'Instruction pointer;

- **SP**: lo stack pointer;
- **DP**: il data pointer, che come vedremo contiene le locazioni degli operandi di istruzioni.

Come avevamo visto, non programmeremo il nostro processore attraverso il linguaggio macchina, ma con un linguaggio assembler che codifica le istruzioni macchina, nella forma già vista:

```
1 OPCODE source, destination
```

Questo linguaggio sarà simile a quello già studiato, cioè dei processori Intel x86. La differenza sarà che avremo come problema il dover effettivamente codificare ciò che scriviamo in assembler in istruzioni in linguaggio macchina da fornire al processore (adesso non stiamo solo *programmando*, ma anche *progettando* il processore).

### 27.3.3 Modalità di indirizzamento

Avevamo visto le seguenti modalità di indirizzamento per le istruzioni:

- Di registro;
- Immediato;
- Di memoria;
- Delle porte di I/O riempi

listone istruzioni

## 28 Lezione del 26-11-24

### 28.1 Dall'assembler al linguaggio macchina

Abbiamo visto come un processore si programma attraverso il linguaggio **assembler**, che è più distante dai dettagli di implementazione e più vicino al linguaggio umano, quindi al programmatore. Vediamo quindi il processo di conversione dall'assembler al linguaggio macchina "*assemblaggio*".

#### 28.1.1 Fetch degli operandi

Il problema principale del processore è individuare gli operandi in base al tipo di indirizzamento dell'istruzione. Ad esempio:

- **MOV** AH, AL contiene operandi che si trovano già nei registri;
- **MOV** \$0x10, AL contiene operandi che vanno letti in memoria assieme all'istruzione stessa;
- **MOV** 0x00FF, AL contiene operandi che vanno letti in memoria, in locazioni diverse da quella dell'istruzione stessa.

Durante la fase di fetch, quindi, il processore deve procurarsi gli operandi necessari ad eseguire la prossima istruzione, che questi siano nei registri o in memoria. Questa operazione sarà comune a ogni istruzione, cioè le modalità di indirizzamento degli operandi saranno identiche che si parli di una **ADD** come di una **MOV**, ecc... Dal punto di vista pratico, questo significa che la parte di **fetch** potrà essere messa *in comune*, mentre la fase di **esecuzione** sarà *specifica* ad ogni istruzione.

### 28.1.2 Formato delle istruzioni macchina

I primi 3 bit del codice operativo di un'istruzione rappresentano il formato dell'istruzione:

- **Formato F0 (000):** sono istruzioni per le quali il processore non deve compiere nessuna azione, in quanto accade alternativamente che:
  - Gli operandi sono registri;
  - Le istruzioni non hanno operandi.

In questo caso il processore deve solo raccogliere l'indirizzo della prossima istruzione dall'Instruction Pointer ??? ;

- **Formato F2 (010):** l'operando **sorgente** si trova in memoria, indirizzato attraverso DP. In questi casi tutta l'istruzione sta su un byte, ma l'operando sorgente va prelevato tramite un'ulteriore lettura in memoria di un byte;
- **Formato F3 (011):** l'operando **destinatario** si trova in memoria, indirizzato attraverso DP. Notiamo che in questo caso l'operando destinatario non va *prelevato* dal processore, ma deve essere *sovrascritto*, cosa che accade durante la fase di esecuzione. Non ci sono quindi letture ulteriori in fase di fetch;
- **Formato F4 (100):** l'operando **sorgente** è indirizzato in modo **immediato**, e sta su 8 bit. In questo caso l'istruzione è lunga 2 byte, e nella fase di fetch si dovranno leggere due byte in memoria consecutivi puntati dal registro IP (e non DP come nel caso precedente);
- **Formato F5 (101):** l'operando **sorgente** si trova in memoria con indirizzamento **diretto**. Visto che lo spazio di memoria è a 24 bit, un'istruzione sarà lunga 4 byte: 1 di OP-CODE e 3 di indirizzo. In fase di fetch si dovrà quindi:
  - Leggere i 4 byte consecutivi dell'istruzione puntati dal registro IP per ottenere istruzione e indirizzo dell'operando sorgente;
  - Leggere l'operando sorgente stesso, in un'altra locazione di memoria.
- **Formato F6 (110):** l'operando **destinatario** si trova in memoria con indirizzamento **diretto**. Di nuovo, il processore dovrà leggere 4 byte consecutivi puntati dall'IP, ma a questo punto non ci saranno *letture* in memoria, bensì *scritture* sull'indirizzo prelevato del destinatario, in fase di **esecuzione**;
- **Formato F7 (111):** raggruppa le istruzioni di controllo seguite da indirizzo (quindi, ad esempio, non la **RET**, ma le varie **CALL**, **JMP**, salti condizionali ecc..). In questo caso, l'indirizzo viene specificato come prima su 3 byte, e bisogna nuovamente leggere 4 byte consecutivi puntati dall'indirizzo in IP.
- **Formato F1 (001):** raggruppa le istruzioni non classificabili nei precedenti formati, fra cui:
  - Istruzioni I/O, con indirizzo a 16 bit sia sorgente (**IN**) che destinatario (**OUT**);
  - **MOV** con uno dei registri a 24 bit (DP o SP), sia sorgente che destinatario.

Le operazioni in formato F1 si limitano in fase di fetch a prelevare l'OP-CODE (fetch *scarna*). Gli operandi vengono gestiti successivamente in fase di esecuzione. Questa soluzione è sì poco elegante, ma risulta più agevole dal punto di vista dell'implementazione.

## 28.2 Architettura hardware del calcolatore

Vediamo quindi come viene implementato dal punto di vista fisico il calcolatore. Innanzitutto consideriamo un **bus**, formato da:

- **Linee di indirizzo**, nel nostro esempio 24 per 24 bit, di uscita per il processore e ingresso per gli spazi di memoria e I/O, notando che nel salto dal bus allo spazio di I/O si perdono gli 8 bit più significativi (si passa da 24 a 16). Non servono "forchette" con porte tri-state in quanto il processore è sempre l'unico a scrivere sulle linee;
- **Linee dati**, nel nostro esempio 8 per 8 bit, usate per leggere e scrivere byte di memoria. In questo caso il processore potrebbe leggere (dalla memoria o dallo spazio di I/O) o scrivere (sempre nella memoria o nello spazio di I/O), ergo potrebbero esserci conflitti di pilotaggio dei dati. Si adottano quindi le porte tri-state, disposte come abbiamo visto a forchetta.
- **Linee di controllo**, tutte attive basse, che sono nel nostro caso:
  - `/mr` e `mw`, cioè memory read e memory write per la memoria;
  - `ior` e `iow`, cioè I/O read e I/O write per lo spazio di I/O
- **Clock e reset**.

Notiamo come nel bus non figura la linea di selezione `/s` per gli spazi di memoria, in quanto questo viene generato attraverso una maschera dalla memoria stessa sulla base degli indirizzi di lettura, cioè avrà il solo scopo di selezionare diversi banchi di memoria, a *livello* di memoria.

### 28.2.1 Spazio di memoria

Abbiamo quindi che lo spazio di memoria è implementato, su 16 MB, in parte con tecnologia RAM, in parte con EPROM (che contiene il bootstrap), e in parte con memoria video dedicata. Diciamo di avere 64 KB di memoria EPROM e 64 KB di memoria video.

Possiamo combinare queste memorie attraverso, come abbiamo visto prima, linee di select generate attraverso opportune maschere. In particolare, finisci quando ci arrivi vai

### 28.2.2 Spazio di I/O

Lo spazio di I/O è realizzato fisicamente attraverso **interfacce**, che sono elementi di raccordo tra il bus e i dispositivi I/O. Dal lato dispositivo, queste sono implementate in una maniera che "risponde" al dispositivo. Dal lato bus, invece, sono tutte uguali, cioè presentano le entrate di selezione, I/O read e I/O write, eventuali **indirizzi interni**, che servono a discriminare più **porte**, e le linee di entrata/uscita di un byte di dati, cioè, tranne che per gli indirizzi interni, le stesse linee fornite da una RAM.

Notiamo che in un interfaccia una porta può operare o in **sola lettura**, o in **sola scrittura**. Ad esempio, non potremo scrivere nell'interfaccia di una tastiera, e non potremo leggere nell'interfaccia di un monitor.

Potremmo chiederci come mai implementare interfacce per ogni dispositivo, e non connetterli direttamente al bus. Ci sono principalmente due ragioni:

- Diversi dispositivi hanno **diverse velocità**: spesso di molti ordini di grandezza, e comunque molto più lente del processore;
- Diversi dispositivi hanno **diverse modalità di trasferimento**: a volta un bit alla volta (**seriale**), a volte in gruppi di bit (**parallelo**).

Implementare interfacce ci permette quindi di **standardizzare** l'input e l'output del calcolatore, rendendo le temporizzazioni e le modalità di trasferimento **omogenee**.

### 28.2.3 Processore

Possiamo quindi individuare, nel processore, tutti i registri effettivamente necessari:

- Registri visibili al programmatore, cioè:
  - i soliti
- Registri di supporto a uscite, fra cui:
  - i soliti

Dove notiamo la particolarità dei registri che supportano le linee dati, che saranno innanzitutto *forchettati*, cioè fatti passare attraverso una porta tri-state controllata da un enabler generato dal registro DIR (per *direzione*). DIR sarà a 0 di default, cioè scollegherà il registro dati di uscita dal bus, e lo porremo a 1 solo nel caso in cui dovremmo scrivere, attraverso tale registro, sul bus.

- I registri STAR e MJR;
- Registri di OPCODE, SOURCE e DEST\_ADDR necessari alla fase di fetch;
- Registri di appoggio per operazioni, che saranno APP3, APP2, APP1, APP0 e NUM-LOC.

### 28.2.4 Ciclo di fetch-execute

Vediamo quindi i dettagli del ciclo fetch-execute.

- La fase iniziale è quella di **reset**, dove si inizializzano i registri:
  - F verrà inizializzato a 0;
  - IP otterrà il valore del primo indirizzo dello spazio di memoria dove si trova il bootstrapper;
  - STAR sarà inizializzato al primo stato;
  - DIR verrà inizializzato a 0;
  - /MR, /MW, /IOR e /IOW verranno inizializzati a 1 (attivi bassi);
- Poi si passa alla fase di **fetch**, dove si prelevano istruzioni e operandi. In ordine:
  - Il processore preleva un byte dalla memoria, all'indirizzo indicato in IP;
  - Incrementa IP, modulo  $2^{24}$ ;
  - Controlla che il byte prelevato corrisponda all'OPCODE di una delle istruzioni che conosce. In caso contrario, si va in stato di blocco;

- Carica il byte prelevato in OP CODE;
- Controlla il formato dell'OP CODE in modo da definire le modalità di indirizzamento. A questo punto si ramifica, effettuando le letture necessarie in memoria come specificato qualche paragrafo fa. Nello specifico, si dovranno eseguire le seguenti operazioni:
  - \* F2, F4, F5: operando sorgente da 8 bit in SOURCE
    - F2: sorgente in memoria con indirizzamento indiretto: si fa accesso in memoria all'indirizzo contenuto in DP;
    - F4: sorgente immediato: si fa un accesso all'indirizzo contenuto in IP. Bisogna incrementare IP di 1;
    - F6: sorgente in mem. con indirizzamento diretto: si fanno due accessi in memoria, i 3 byte dell'indirizzo in IP e il byte all'indirizzo appena trovato. Bisogna incrementare IP di 3.
  - \* F4, F6, F7: indirizzo dell'operando destinatario da 24 bit in DEST\_ADDR
    - F3: indirizzo destinatario in memoria: si copia l'indirizzo da DP;
    - F6, F7: indirizzo destinatario in memoria: si fa accesso in memoria ai 3 byte puntati da IP. Bisogna incrementare IP di 3;
  - \* F0: non si fa nulla;
  - \* F1: gli operandi verranno raccolti in fase execute (per ora non si fa nulla).
- Come ultima cosa, si guarda al contenuto di OP CODE per iniziare l'esecuzione dell'istruzione desiderata.
- Dopo la fase di fetch, viene la fase **execute**, dove si eseguono effettivamente operazioni sugli operandi;
- Nel caso di un errore in fase di fetch, o dell'incontro dell'istruzione HLT in fase execute, si dovrà andare in **stato di blocco**, cioè il processore dovrà smettere di rispondere agli ingressi e mantenere ferme le sue uscite.

## 29 Lezione del 27-11-24

### 29.1 Letture e scritture nello spazio di memoria

Durante la fase di fetch, abbiamo eseguito solo **letture** in memoria. Vediamo adesso che nella fase **execute** abbiamo bisogno di effettuare sia letture che scritture. Vediamo come si effettuano a livello di  $\mu$ -istruzioni queste letture e scritture, in maniera compatibile con le temporizzazioni già definite sulle memorie.

#### 29.1.1 Lettura

Innanzitutto ripassiamo le temporizzazioni nel caso della lettura:

- A indirizzi stabili arriva il comando  $/m_r$ ;
- $/s$  arriva con ritardo;
- A  $/m_r$  e  $/s$  bassi si effettua la lettura, cioè le tri-state vanno in conduzione;
- I multiplexer alle uscite vanno a regime dopo gli indirizzi, da qui in poi i dati sono buoni e si prelevano;



- Quando `/mr` torna a 1 i dati tornano ad alta impedenza, da lì in poi `/s` e gli indirizzi possono tornare instabili.

Definiamo allora un  $\mu$ -programma per la lettura in memoria:

```
1 mem_r0: begin A23_A0 <= address; DIR <= 0; MR <= 0; STAR <= mem_r1; end
2 mem_r1: begin STAR <= mem_r2; end // stato di wait, da qui in poi omissso
3 mem_r2: begin cpu_register <= d7_d0; MR <= 1; ...; end
```

Notiamo che allo stato R2 si possono cambiare tutte le linee (indirizzo, ecc...) tranne che la DIR, in quanto impostando solo a quel punto MR non si è sicuri che la RAM risponda in tempo.

### 29.1.2 Scrittura

Ricordiamo che la scrittura è distruttiva. Ricordiamo quindi le temporizzazioni:

- Si abbassa `/s` e ci si assicura che gli indirizzi siano stabili;
- Si abbassa `/mr`;
- I dati dovranno essere corretti fino al fronte di salita di `/mr`.

Definiamo un'altro  $\mu$ -programma, stavolta per la scrittura in memoria:

```
1 mem_w0: begin A23_A0 <= address; D7_D0 <= new_byte; DIR <= 1; STAR <= mem_w1; end
2 mem_w1: begin MW <= 0; STAR <= mem_w2; end
3 mem_w2: begin MW <= 1; STAR <= mem_w3; end
4 mem_w3: begin DIR <= 0; ...; end
```

Notiamo di non poter abbassare DIR o gli indirizzi fino allo stato W3, in quanto non si può essere sicuri che a quel punto la RAM abbia finito di scrivere.

## 29.2 Letture e scritture nello spazio di I/O

Le letture e le scritture nello spazio di I/O sono diverse, in quanto qui la **lettura è distruttiva**. Inoltre, dobbiamo ricordarci di operare sui registri IOR e IOW anzichè MR e MW.

### 29.2.1 Lettura

Scriviamo quindi un  $\mu$ -programma per la lettura nello spazio di I/O dove teniamo conto di dover abbassare IOR **dopo** che gli indirizzi sono stabili, in maniera simile alla lettura:

```
1 io_r0: begin A23_A0 <= address; DIR <= 0; STAR <= io_r1; end
2 io_r1: begin IOR <= 0; STAR <= io_r2; end
3 io_r2: begin STAR <= io_r3; end
4 io_r3: begin cpu_register <= d7_d0; IOR <= 1; ...; end
```

### 29.2.2 Scrittura

Ridefiniamo quindi il  $\mu$ -programma di scrittura:

```
1 io_w0: begin A23_A0 <= address; D7_D0 <= new_byte; DIR <= 1; STAR <= io_w1; end
2 io_w1: begin IOW <= 0; STAR <= io_w2; end
3 io_w2: begin IOW <= 1; STAR <= io_w3; end
4 io_w3: begin DIR <= 0; ...; end
```

Notiamo che, in questo caso, la scrittura si fa sul fronte di discesa anziché di salita, e quindi l'assegnamento di D7\_D0 al nuovo byte va fatto esclusivamente in W0, e non in W1 com'era possibile nella scrittura nello spazio di memoria.

metti gli address giusti, qui sarebbe 'H00 + offset

### 29.3 Accessi multipli in memoria

Il processore potrebbe fare accessi non solo ad un byte, ma 2 byte (per operandi su 16 bit) o 3 byte (per indirizzi). Occasionalmente dovremo leggere anche 4 byte, ma questo non è considerato nel corso.

Per fare letture su locazioni multiple, si usano  $\mu$ -sottoprogrammi di lettura/scrittura modulari. Utilizzeremo:

- Il registro MJR per contenere il  $\mu$ -indirizzo di ritorno;
- Il registro NUMLOC come contatore del numero di byte da leggere/scrivere;
- Il registro A23\_A0 per contenere l'indirizzo del primo byte da leggere/scrivere;
- I registri APP0, ..., APP3 per contenere i byte letti/da scrivere.

#### 29.3.1 Lettura

Vediamo quindi il  $\mu$ -programma principale di lettura:

```
1 s_x: begin ... A23_A0 <= address; MJR <= s_x+1; STAR <= subprogram; end
2 s_x+1: begin ... /* qui si usa APP0 */ end
```

Definiamo 4  $\mu$ -sottoprogrammi:

- readB: legge 1 byte;
- readW: legge 2 byte;
- readM: legge 3 byte;
- readL: legge 4 byte.

I parametri di ingresso saranno l'indirizzo in memoria della prima locazione e la DIR impostata a 0, i parametri di uscita i registri APP da 0 a 3 (che conterranno i byte letti).

Vediamo quindi i  $\mu$ -sottoprogrammi di lettura:

```
1 readB: begin MR <= 0; NUMLOC <= 1; STAR <= read0; end;
2 readW: begin MR <= 0; NUMLOC <= 2; STAR <= read0; end;
3 readM: begin MR <= 0; NUMLOC <= 3; STAR <= read0; end;
4 readL: begin MR <= 0; NUMLOC <= 4; STAR <= read0; end;
5
6 read0: begin APP0 <= d7_d0; A23_A0 <= A23_A0 + 1; NUMLOC <= NUMLOC - 1;
    STAR <= ( NUMLOC == 1 ) ? read4 : read1; end
7 read1: begin APP1 <= d7_d0; A23_A0 <= A23_A0 + 1; NUMLOC <= NUMLOC - 1;
    STAR <= ( NUMLOC == 1 ) ? read4 : read2; end
8 read2: begin APP2 <= d7_d0; A23_A0 <= A23_A0 + 1; NUMLOC <= NUMLOC - 1;
    STAR <= ( NUMLOC == 1 ) ? read4 : read3; end
9 read3: begin APP3 <= d7_d0; A23_A0 <= A23_A0 + 1; STAR <= read4; end
10 read4: begin MR <= 1; STAR <= MJR; end
```

non sono sicuro, comunque commentali

### 29.3.2 Scrittura

Vediamo il  $\mu$ -programma principale di scrittura:

```
1 s: begin ... APP1 <= datum(15:8); APP0 <= datum(7:0); A23_A = <= address;
    MJR = s_x+1; STAR <= subprogram; end
```

E definiamo i 4  $\mu$ -sottoprogrammi:

- writeB: scrive 1 byte;
- writeW: scrive 2 byte;
- writeM: scrive 3 byte;
- writeL: scrive 4 byte.

I parametri di ingresso saranno l'indirizzo in memoria della prima locazione, la DIR impostata a 0, e i registri APP da 0 a 3 (che conterranno i byte da scrivere).

Implementiamo i  $\mu$ -sottoprogrammi come:

```
1 writeB: begin D7_D0 <= APP0; DIR <= 1; NUMLOC <= 1; STAR <= write0; end
2 writeW: begin D7_D0 <= APP0; DIR <= 2; NUMLOC <= 1; STAR <= write0; end
3 writeM: begin D7_D0 <= APP0; DIR <= 3; NUMLOC <= 1; STAR <= write0; end
4 writeL: begin D7_D0 <= APP0; DIR <= 4; NUMLOC <= 1; STAR <= write0; end
5
6 write0: begin MW <= 0; STAR <= write1; end;
7 write1: begin MW <= 1; STAR <= ( NUMLOC == 1 ) ? write11 : write2; end
```

finisci qua sopra

## 29.4 Descrizione in Verilog del processore

Iniziamo quindi a definire il nostro processore col linguaggio Verilog. qui per l'amor di cristo riscrivi per benino e metti nella cartella /verilog che così è drammatico

## 30 Lezione del 28-11-24

### 30.1 Descrizione in Verilog del ciclo fetch/execute

#### 30.1.1 Fase di fetch

tutto verilog

Alla fine della fase di fetch saremo riusciti con successo a mettere:

- Il codice operativo dell'istruzione in OPCODE;
- L'operando immediato o in memoria dell'istruzione in SOURCE;
- L'operando destinatario in DEST\_ADDR;
- IP sulla prossima istruzione da prelevare.

#### 30.1.2 Fase di esecuzione

Nella fase di esecuzione, avremo quindi tutti gli operandi già inizializzati, e dovremo solo farli passare attraverso apposite reti combinatorie.

altro verilog

### 30.1.3 Formato F1

Abbiamo visto come un'eccezione tra le istruzioni è rappresentata da quelle in formato F1, in quanto il "fetch" effettivo dei loro operandi sorgenti e destinatari va fatto in fase di esecuzione. Queste, ricordiamo, sono le istruzioni di I/O (operandi su indirizzi a 16 bit) e MOV sui registri da 24 bit.

indovina? altro verilog

## 30.2 Interfacce

Veniamo adesso alla descrizione di interfacce che completano il calcolatore, cioè gli permettono di comunicare col mondo esterno. Le interfacce possono essere di due tipi principali:

- **Parallele:** un byte alla volta (quindi più bit *in parallelo*);
- **Seriali:** un bit alla volta.

Vedremo poi anche le interfacce per la conversione da **analogico a digitale** e viceversa, che trasformano da tensioni a gruppi di bit.

I collegamenti lato bus delle interfacce, come avevamo anticipato sono sempre uguali, mentre cambiano sul lato dispositivo.

### 30.2.1 Visione funzionale di un interfaccia

La visione funzionale di un interfaccia è quella dal punto di vista di chi deve interagirci, cioè come un insieme di registri su cui opererà il **processore**:

- **Receive Buffer Register (RBR):** registro dove si vanno a *leggere* informazioni **dall'interfaccia**;
- **Transmit Buffer Register (TBR):** registro dove si vanno a *scrivere* informazioni **all'interfaccia**.

### 30.2.2 Sincronizzazione processore-dispositivi

Eseguendo un programma che contiene sequenze di istruzioni **IN** o **OUT**, il processore non può sapere se fra una **IN** e l'altra (o fra una **OUT** e l'altra) il dispositivo ha prodotto nuovi dati (o se ha processato quelli inviati). Dovremo quindi implementare un doppio handshake, sia sul lato processore (*handshake "software"*) che sul lato hardware (*handshake "hardware"*).

Dotiamo quindi le interfacce di registri di stato:

- **Receive Status Register (RSR):** contiene un bit di interesse, il flag **FI** di **ingresso pieno**;
- **Transmit Status Register (TSR):** contiene un altro bit di interesse, il flag **FO** di **uscita vuota**.

I due flag FI e FO vengono controllati dall'interfaccia, e quindi impostati a 1 o a 0 quando questa rileva le condizioni opportune.

### 30.2.3 Ingresso dati a controllo di programma

Vediamo quindi un ciclo di ingresso dati. Si parte con FI a 0. Quando il dispositivo gestito dall'interfaccia scrive in RBR, l'interfaccia mette FI a 1. Questo segnala al processore che c'è un nuovo dato. A questo punto, quando il processore accede in lettura al registro RBR, l'interfaccia riporta FI a 0.

Notiamo che su due letture consecutive il processore è in **attesa attiva** finché non FI non si alza nuovamente. Esistono altri metodi di accesso in memoria che non richiedono l'attesa attiva da parte del processore, fra cui il meccanismo degli *interrupt* e il *DMA* (*Direct Memory Access*).

### 30.2.4 Uscita dati a controllo di programma

Vediamo adesso un ciclo di uscita dati. Il flag FO parte a 0. L'interfaccia lo mette a 0 quando il processore scrive in TBR, per segnalare che il dispositivo non ha ancora elaborato. Quando il dispositivo accede a TBR per la lettura, FO torna a 0.

## 31 Lezione del 03-12-24

### 31.1 Interfacce parallele

#### 31.1.1 Interfacce di ingresso senza handshake

Iniziamo a vedere le interfacce parallele di **ingresso** senza handshake. Queste scambiano interi byte col processore, attraverso un solo registro. Non c'è quindi nessuna linea di indirizzo, l'intero registro va direttamente al processore su una linea dati. Abbiamo poi la linea di select  $/s$  e la linea di I/O read  $/ior$ .

Il registro in uscita è forchettato da una tri-state in modo da mantenere ad alta impedenza l'uscita del registro RBR quando il processore non sta leggendo dall'interfaccia. Inoltre, non vorremo nemmeno che il registro RBR si trovi a leggere dati quando il processore non sta leggendo.

Possiamo quindi ricavare, dal select e l'I/O read attraverso una porta NOR, un segnale di enable sia per la porta tri state che per il registro: quando entrambi sono bassi, si legge dal lato dispositivo dell'interfaccia all'interno del registro, e si restituisce l'uscita del registro al processore.

Notiamo che l'aggiornamento dell'RBR avviene al *fronte di salita* dell'enabler (quindi di  $/ior$ ), quindi una volta sola per ogni lettura.

#### 31.1.2 Interfacce di uscita senza handshake

L'interfaccia parallela di **uscita** senza handshake è simile: si ha sempre un solo registro, TBR, che memorizza le linee dati in entrata quando sono entrambi bassi il select e l'I/O write ( $/iow$ ), cosa ricavata attraverso un'altra porta NOR.

Notiamo che in questo caso TBR campiona sul *fronte di discesa* dell'enabler (quindi di  $/iow$ ), anziché di salita.

#### 31.1.3 Interfacce di ingresso/uscita senza handshake

Le interfacce di ingresso/uscita senza handshake si costruiscono unendo due interfacce, una di ingresso e una di uscita (e appunto dette **sottointerfacce** di ingresso e uscita),

e selezionando l'interfaccia giusta attraverso un bit di indirizzo ( $a_0$ ). Il select per la selezione viene generato da una semplice rete combinatoria che prende in ingresso il  $/s$  globale e il bit di indirizzo, con tabella di verità: Dove  $/s_i$  è il select della sottointerfaccia

$/s$	$/a_0$	$/s_i$	$/s_o$
1	-	1	1
0	0	0	1
0	1	1	0

di ingresso, e  $/s_o$  il select della sottointerfaccia di uscita. Come vediamo dall'esempio, solitamente si mette a indirizzo **pari** la porta di *ingresso*, e a indirizzo **dispari** la porta di *uscita*.

Notiamo che un montaggio alternativo si ottiene ignorando il bit di indirizzo e accedendo direttamente alle due sottointerfacce con un unico select. A questo punto sarà compito del processore discriminare fra  $/ior$  e  $/iow$  per selezionare la sottointerfaccia desiderata.

#### 31.1.4 Interfacce di ingresso con handshake

Ricordiamo la visione funzionale delle interfacce di ingresso con handshake: dovremo avere i registri RBR e RSR, da cui ricaviamo il flag FI lato processore, mentre lato dispositivo dovremo avere sia le linee di ingresso dati che le linee di handshake, che assumiamo essere in forma  $/dav$  e  $rfd$ .

L'interfaccia si implementa quindi come una combinazione di una rete combinatoria, come avevamo visto per le interfacce senza handshake, per la generazione degli enable, e una rete sequenziale per la gestione degli handshake.

Il processore potrà accedere in lettura sia al RBR che al RSR: questo si discrimina attraverso un bit di indirizzo ( $a_0$ ). Si ha quindi la rete combinatoria per la generazione degli enable, dove  $eb$  è l'enable del buffer e  $es$  è l'enable del registro di stato:

$/s$	$/ior$	$/a_0$	$/es$	$/eb$
0	0	0	1	0
0	0	1	0	1
...			0	0

sintesi rete sequenziale

Notiamo che, nella sintesi, si ritarda il clock dell'interfaccia rispetto a quello del processore per evitare condizioni di *corsa* all'interfaccia.

#### 31.1.5 Interfacce di uscita con handshake

L'interfaccia di uscita è realizzata in modo analogo: includiamo un registro TSR che contiene il flag FO lato processore, il registro TBR, e lato dispositivo le linee  $/dav$  e  $rfd$ , dove però è l'interfaccia, e non più il processore, a fare da produttore.

La struttura interna sarà del tutto simile a l'interfaccia di ingresso, con una parte combinatoria che si occupa degli enable e una parte sequenziale che si occupa degli handshake. La parte combinatoria obbedirà alla tabella di verità:

Notiamo che compare comunque la linea di uscita in quanto vogliamo leggere lo stato del TSR.

sintesi parte sequenziale

/s	/ior	/iow	/a0	/es	/eb
0	0	1	0	1	0
0	1	0	1	0	1
...				0	0

### 31.1.6 Interfacce di ingresso/uscita con handshake

Possiamo combinare le interfacce con handshake viste finora in un'unica interfaccia di ingresso/uscita. Combineremo il registro di stato in un unico registro RTSR dotato dei flag FO e FI, e useremo un bit di indirizzo (a0) per distinguere fra le porte di ingresso e uscita.

## 31.2 Interfacce seriali

Le interfacce seriali trasmettono informazioni un bit a volta. Noi ne consideriamo una versione semplificata, l'interfaccia seriale start/stop.

Dal punto di vista fisico, un'interfaccia seriale trasporta informazioni su due linee:

- La linea di **massa**, che funge da riferimento;
- La linea **segnale**, che porta appunto il segnale riferito a massa.

Dal punto di vista logico, invece, a interessarci sarà solo la linea segnale. Questa trasporta informazioni *half duplex*, cioè da un trasmettitore a un ricevitore (la comunicazione nelle due direzioni richiede quindi due linee).

Ora, se il segnale è rappresentato da una sequenza di **marking** (1, linea in tensione) e **spacing** (0, linea a massa) trasmessi con un periodo  $T$ , il problema diventerà sincronizzare trasmettitore e ricevitore in modo che possano comunicare efficientemente.

Quello che ci interesserà stabilire sarà quindi il **tempo di bit**  $T$  e la **trama** del segnale. Definiamo trama una sequenza di bit che rappresenta un frammento di comunicazione: iniziamo la trama con uno spacing (**bit di start**), e seguiamo poi con un numero che va da 5 a 8, stabilito in precedenza, di **bit utili** (solitamente LSB). Infine, trasmettiamo un marking per segnalare la fine della trama (**bit di stop**).

I bit di start e di stop rappresentano un **overhead**: una trama di  $n$  bit utili è lunga almeno  $n + 2$  per segnalare inizio e fine della trama. Avremo quindi che la velocità netta della linea è  $\frac{n}{n+2}$ , che è comunque più efficiente di usare un clock secondario o effettuare un handshake per ogni bit. se vuoi riguarda

Converrebbe quindi usare  $n$  arbitrariamente lunghi: purtroppo questo è fattibile fino ad un certo limite superiore, in quanto i clock di trasmettitore e ricevitore saranno necessariamente leggermente differenti in frequenza, ergo sul lungo termine si andrebbe ad accumulare un'errore troppo grande.

Infatti, l'impedenza dalla linea di trasmissione determinerà uno "smussamento" del segnale, motivo per cui preferiremo campionare ogni bit trasmesso nella posizione più centrale possibile rispetto alla sua forma d'onda. Questo significherà che, posto  $T$  il periodo del clock del trasmettitore, e  $T + \Delta T$  il periodo del clock del ricevitore, vorremo:

$$n \cdot \Delta T \leq \frac{T}{2} \implies \frac{\Delta T}{T} \leq \frac{1}{2n}$$

Uno standard di *interoperabilità* per le trasmissioni seriali è l'EIA-RS232C, che fissa lo 0 logico a tensioni da 3V a 25V, e l'1 logico a tensione negativa da -25V a -3V.

## 32 Lezione del 05-12-24

### 32.1 Vista funzionale delle interfacce seriali

L'interfaccia seriale è effettivamente un'interfaccia di ingresso uscita con handshake, dove ingressi e uscite sono i bit  $t_{xa}$  di trasmissione e  $r_{xa}$  di ricezione. In particolare, questi sono gestiti rispettivamente da due sottointerfacce dette **ricevitore** e **trasmettitore**, sincronizzati da un **generatore di segnali di sincronizzazione** e interfacciati con una sottointerfaccia di gestione attraverso handshake  $_{dav} - /_{rfd}$ . Nell'handshake lato sottointerfaccia di gestione, il **ricevitore** è *produttore* e il **trasmettitore** è il *ricevitore*, cioè il ricevitore *restituisce* i dati ricevuti e il trasmettitore *prende* i dati da trasmettere. Lato processore si hanno comunque 8 bit di uscita, cioè un byte, e il bit di indirizzo che discrimina fra porte di ingresso / uscita (cioè fra ricevitore e trasmettitore).

descrizione di entrambi

## 33 Lezione del 10-12-24

### 33.1 Conversione digitale/analogico e analogico/digitale

Finora abbiamo assunto che le interfacce lavorino solo e soltanto con segnali digitali. In verità nel mondo esterno al computer le grandezze variano su una scala continua. Occorrono appositi convertitori, detti convertitori digitale/analogico e analogico/digitale.

La grandezza analogica che consideriamo è un voltaggio appartenente alla scala FSR (Full Scale Range)  $[5, 30]V$ . Questa verrà convertita in un intero  $x$  rappresentato su  $N$  bit con  $N \in \{8, 16\}$ . A seconda dell'interpolazione scelta, potremo distinguere fra:

- **Conversione ubipolare:**  $v \in [0, FSR], x \in [0, 2^N - 1]$ ;
- **Conversione bipolare:**  $v \in \left[-\frac{FSR}{2}, \frac{FSR}{2}\right], x \in [-2^{N-1}, +2^{N-1} - 1]$

#### 33.1.1 Errori di conversione

Dato  $K = \frac{FSR}{2^N}$ , nel caso ideale vorremmo  $v = Kx$ . In realtà, avremo che  $|v - Kx| \leq \varepsilon$ , con un  $\varepsilon$  dovuto a errori di conversione:

- **Errore di non linearità;**
- **Errore di quantizzazione.** oggi è allegro

#### 33.1.2 Tempi di risposta

I convertitori D/A sono praticamente "combinatori", e quindi estremamente veloci (pochi nanosecondi). I convertitori A/D, di contro, hanno tempi di risposta variabili in base alle architetture. Noi vedremo i convertitori ad **approssimazioni successive (SAR)**, che hanno tempi di risposta di qualche centinaio di nanosecondi.

#### 33.1.3 Convertitori bipolari

I convertitori bipolari lavorano con rappresentazioni degli interi in traslazione (detta appunto anche *binaria bipolare*). Il numero intero  $x$  viene quindi rappresentato dal naturale  $X = x + 2^{N-1}$ . In ogni caso, per riportare in complemento a 2 basterà complementare il MSB.



### 33.2 Convertitore D/A

Un convertitore D/A può essere realizzato come segue:

- circuito
- descrizione

Anche se non si considerano resistori e amplificatori operazionali come componenti combinatori, il circuito è effettivamente "combinatorio" nel senso che ha tempi di risposta estremamente veloci. Il problema è però quello delle transizioni multiple dello stato di uscita: questo si risolve attraverso un filtro *passa-basso* in uscita.

#### 33.2.1 Interfaccia per la conversione D/A

Vediamo un'interfaccia parallela per l'operazione di un convertitore D/A. Sul lato di uscita non si avranno handshake (sola uscita) ma il convertitore D/A stesso.

- rete

### 33.3 Convertitore A/D

Descriviamo un particolare tipo di convertitori A/D detto convertitore ad **approssimazioni successive**:

- circuito
- descrizione

Il cuore di un convertitore di questo tipo è una rete sequenziale detta **SAR** (Successive Approximation Register). L'uscita del SAR viene fatta passare attraverso un convertitore D/A dello stesso tipo dell'A/D, e confrontata attraverso un **comparatore** con l'ingresso corrente in modo da migliorare la previsione, in quella che è effettivamente una **ricerca logaritmica** (o *binaria* o *dicotomica*). In particolare, ad ogni iterazione della ricerca si ricava il valore di un singolo bit, per cui  $n$  bit richiedono  $n$  iterazioni. Lato processore, il SAR dovrà implementare inoltre un handshake *soc/eoc*.

Una descrizione in verilog della SAR potrebbe essere:

- descrizione verilog corsini

Il problema di questa descrizione, supponendo questa sia perfettamente chiara, è che abbiamo bisogno di un nuovo stato per ogni iterazione di aggiornamento di RBR. Una soluzione alternativa potrebbe essere allora:

- descrizione verilog stea

dove si introduce un registro COUNT e una rete combinatoria per il calcolo di RBR.

#### 33.3.1 Interfaccia per la conversione A/D

Vediamo un'interfaccia parallela per l'operazione di un convertitore A/D. Lato processore si implementerà un handshake *soc/eoc*.

- rivedi