Reti logiche

Gabriele Frassi[1] (**g.frassi2@studenti.unipi.it**)

A.A 2020-2021 - Primo semestre

[1]Se questi appunti sono stati utili e vuoi ringraziarmi in qualche modo: `https://www.paypal.com/paypalme/GabrieleFrassi`

Origini del motto di Ingegneria UniPI, ispirato dal fondatore del corso di Reti logiche prof. Paolo Corsini.

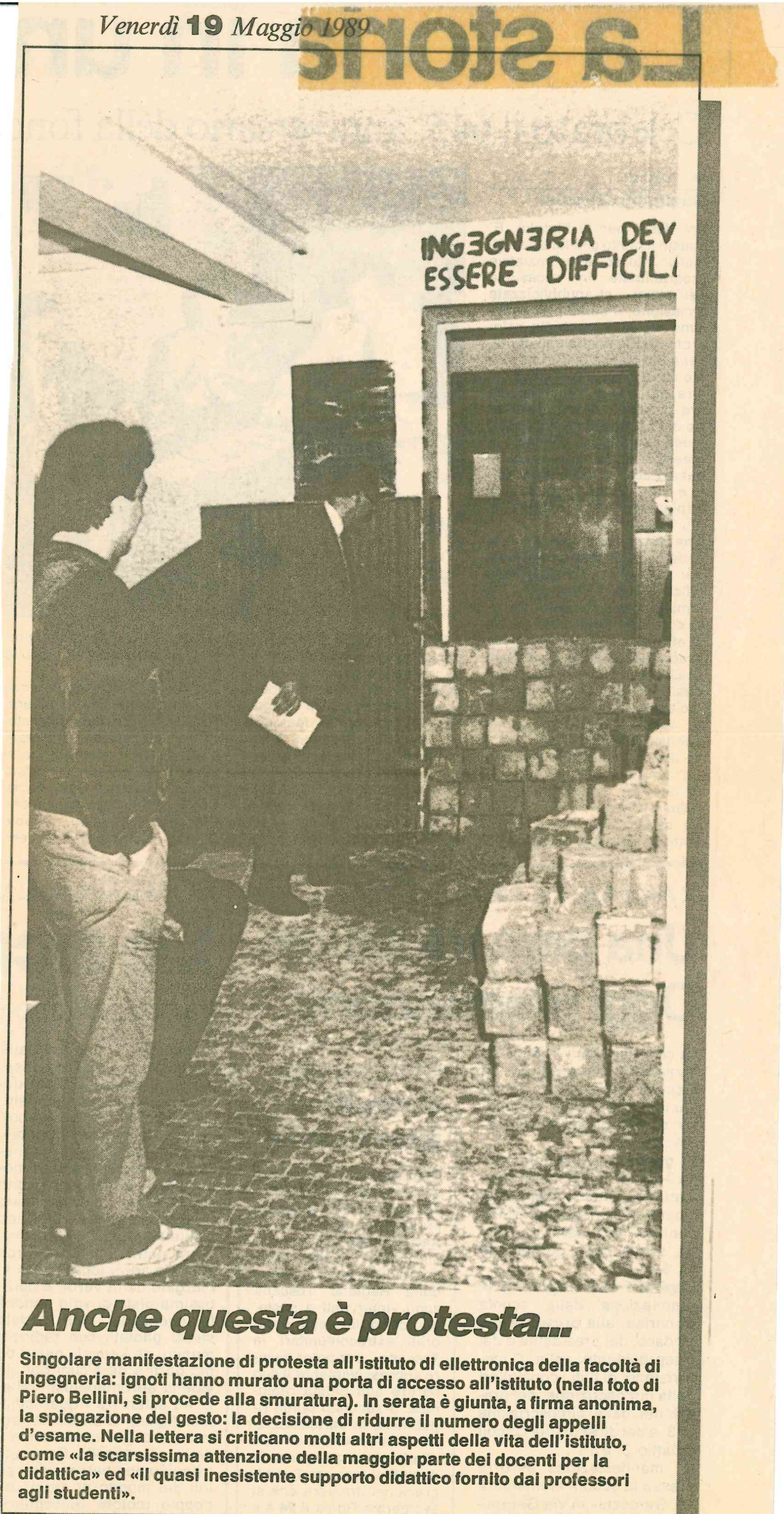
> *Il secondo anno diventa difficile, con Reti logiche al primo semestre e Calcolatori elettronici al secondo. D'altronde avete avuto la strada facile, dovete abituarvi! Quando ero studente ci fu una delle solite riunioni [...] che gli studenti fanno per aumentare il numero di esami, che è già record mondiale adesso (ndr. discorso del 2005-2006). Quando ero studente volevano l'appello mensile, o qualcosa del genere: i docenti erano ovviamente contrari, ci sono tre appelli e bastano [...]. Gli studenti dicevano no: per alcuni esami la probabilità di accesso è molto bassa, e quindi aumentando il numero di prove aumentano le possibilità di accesso. Si alzò un professore che voi conoscete bene, ma di cui non vi dico il nome (ndr. Paolo Corsini), che dice: "è inutile stare a discutere e a perder tempo, INGEGNERIA NON PUO' NON ESSERE DIFFICILE". Dopo qualche giorno alcuni ragazzi si fecero chiudere dentro l'edificio del triennio (ndr. il polo A) e murarono la porta d'ingresso del nostro dipartimento, usando il [...] di quelli di scienze delle costruzioni, che solitamente usano per fare le prove di stress. Impastarono la calce [...], portarono su i provini e con questi fecero un muro chiudendo la porta del dipartimento: sopra la porta del dipartimento scrissero "**INGEGNERIA DEVE ESSERE DIFFICILE**". Da allora questa frase è il motto di Ingegneria, anche se la frase originale era stata "Ingegneria non può non essere difficile". [...] Ingegneria deve essere difficile, non può non essere difficile altrimenti non sarebbe più Ingegneria! Dovete abituarvi a soffrire!!*

- prof. Giuseppe Anastasi, docente del corso di Reti informatiche (terzo anno)

L'episodio è avvenuto attorno al 1990, quando il prof. Anastasi era uno studente come noi. La frase oggi non è più presente, negli anni è stata cancellata e riscritta diverse volte (era ancora presente circa 15 anni fa).

Seguono due reperti storici, per cui si ringrazia il presidente di corso prof. Avvenuti e i rappresentanti in consiglio di Scuola.

Venerdì **19** *Maggio 1989*



Anche questa è protesta...

Singolare manifestazione di protesta all'istituto di elettronica della facoltà di ingegneria: ignoti hanno murato una porta di accesso all'istituto (nella foto di Piero Bellini, si procede alla smuratura). In serata è giunta, a firma anonima, la spiegazione del gesto: la decisione di ridurre il numero degli appelli d'esame. Nella lettera si criticano molti altri aspetti della vita dell'istituto, come «la scarsissima attenzione della maggior parte dei docenti per la didattica» ed «il quasi inesistente supporto didattico fornito dai professori agli studenti».

INGEGNERIA E' FACILE !

Dopo le recenti affermazioni di un noto e stimato docente del ramo calcolatori, sulla difficoltà di Ingegneria, che hanno spinto alcuni ad esprimere il proprio dissenso in maniera molto solida, il Comitato Inter Universitario Coordinamento ed Organizzazione, sempre in prima linea quando si tratta di fare chiarezza su questi argomenti, ha messo al lavoro i propri esperti, per dare una risposta definitiva alla questione.

E la risposta è giunta in breve tempo; utilizzando un programma per la ricerca di tautologie contraddittorie, che girava su di una PROLOG machine dotata di coprocessore filosofico, dopo una intera notte di lavoro è stato ricavato il risultato qua sotto riportato:

Consideriamo le seguenti proposizioni:

1) Le proposizioni scritte entro questo riquadro sono entrambe false.

2) Ingegneria è facile.

I valori di verità possibili per le due proposizioni sono dati dalla seguente tabella:

	propos.1	propos.2
caso A	vera	falsa
caso B	vera	vera
caso C	falsa	falsa
caso D	falsa	vera

I casi A e B, in cui la proposizione 1 è vera, sono evidentemente impossibili, dato che se la proposizione 1 fosse vera, essa sarebbe falsa (contraddizione); il caso C è analogamente impossibile, dato che se le due proposizioni fossero entrambe false, ciò implicherebbe che la proposizione 1 è vera (contraddizione). Quindi l'unico caso possibile è il D; da cui si dimostra come la proposizione 2 sia vera. Resta quindi provato che:

i) Ingegneria è facile.

ii) Anche i professori possono sbagliarsi.

Speriamo che questa inconfutabile dimostrazione possa servire a riportare la tranquillità tra i docenti e soprattutto tra gli studenti.

Il direttore della sezione ricerca del C.I.U.C.O.

Martino Giardiniere

Indice degli appunti

1 Programma del corso **14**

2 Unimap **15**

I Assembler **19**

3 Martedì 29/09/2020 **20**
- 3.1 Linguaggio assembler . . . 20
- 3.2 Struttura di un calcolatore . . . 21
- 3.3 Ripasso: rappresentazione dell'informazione . . . 22
  - 3.3.1 Numeri naturali . . . 22
  - 3.3.2 Numeri interi . . . 22
    - 3.3.2.1 Diagramma della farfalla . . . 23
    - 3.3.2.2 Tabella di conversione da -8 a $+7$ in $p = 4$ . . . 25
    - 3.3.2.3 Rappresentazione esadecimale . . . 25
- 3.4 Osservazione sul calcolo di MB, KB o GB occupati . . . 25
- 3.5 Struttura del calcolatore vista da un programmatore ASS . . . 26

4 Mercoledì 30/09/2020 **29**
- 4.1 Codifica macchina e codifica mnemonica . . . 29
- 4.2 Struttura di un'istruzione . . . 29
- 4.3 Esempio di programma . . . 29
- 4.4 Memoria occupata da una certa istruzione . . . 30
- 4.5 Indirizzamento degli operandi . . . 30
  - 4.5.1 Indirizzamento di registro . . . 30
  - 4.5.2 Indirizzamento immediato . . . 31
  - 4.5.3 Indirizzamento di memoria . . . 31
    - 4.5.3.1 Indirizzamento di tipo diretto . . . 31
    - 4.5.3.2 Indirizzamento di tipo indiretto . . . 31
    - 4.5.3.3 Indirizzamento con displacement e registro di modifica . . . 32
    - 4.5.3.4 Indirizzamento bimodificato senza displacement . . . 32
    - 4.5.3.5 Indirizzamento bimodificato con displacement . . . 32
  - 4.5.4 Indirizzamento delle porte di I/O . . . 32
- 4.6 Principali istruzioni . . . 33
  - 4.6.1 Istruzioni di trasferimento . . . 33
    - 4.6.1.1 MOVE . . . 33
    - 4.6.1.2 LOAD EFFECTIVE ADDRESS . . . 34
    - 4.6.1.3 EXCHANGE . . . 34

5 Giovedì 01/10/2020 **35**
5.1 Concludiamo con le istruzioni di trasferimento . . . 35
5.1.1 INPUT e OUTPUT . . . 35
5.1.1.1 ASSEMBLER non è linguaggio ortogonale . . . 36
5.1.1.2 Uscita da registro a porta . . . 36
5.1.2 Pila . . . 36
5.1.2.1 Pila come memoria temporanea . . . 37
5.1.3 PUSHAD e POPAD . . . 38
5.2 Istruzioni aritmetiche . . . 38
5.2.1 ADD e SUBTRACT . . . 38
5.2.2 INCREMENT e DECREMENT . . . 39
5.2.3 ADD WITH CARRY e SUBTRACT WITH BORROW . . . 40
5.2.4 NEGATE . . . 42
5.2.5 COMPARE . . . 43
5.2.6 INTEGER e INTEGER MULTIPLY . . . 43
5.2.7 DIVIDE e INTEGER DIVIDE . . . 45
5.2.8 Conclusioni . . . 47

6 Venerdì 02/10/2020 **48**
6.1 Continuiamo con le istruzioni aritmetiche . . . 48
6.1.1 Estensione di campo . . . 48
6.1.2 Naturali . . . 48
6.1.3 Interi . . . 48
6.1.4 CONVERT WORD TO DOUBLEWORD in EAX . . . 48
6.2 Istruzioni di traslazione . . . 49
6.2.1 SHIFT LOGICAL LEFT . . . 49
6.2.2 SHIFT ARITHMETIC LEFT . . . 50
6.2.3 SHIFT LOGICAL RIGHT . . . 50
6.2.4 SHIFT ARITHMETIC RIGHT . . . 50
6.3 Istruzioni di rotazione . . . 51
6.3.1 ROTATE LEFT . . . 51
6.4 Istruzioni logiche . . . 52
6.4.1 NOT . . . 52
6.4.2 AND . . . 52
6.4.3 OR . . . 52
6.4.4 XOR (OR esclusivo) . . . 53
6.4.5 Utilizzi di questi operatori . . . 53
6.5 Istruzioni di controllo . . . 54
6.5.1 JUMP . . . 54
6.5.2 JUMP if condition met . . . 54
6.5.3 Sottoprogrammi . . . 55
6.5.3.1 CALL . . . 55
6.5.3.2 RET . . . 55
6.5.3.3 NO OPERATION . . . 55
6.5.3.4 HALT . . . 56
6.6 Protezione ed istruzioni privilegiate . . . 56
6.7 Flag controllati dalle istruzioni condizionate . . . 57

7 Martedì 06/10/2020 **58**
7.1 Assemblatore . . . 58
7.2 Struttura di un programma assembler . . . 58
7.2.1 Assembler case-sensitive o case-insensitive? . . . 59
7.2.2 Cosa cambia rispetto agli esempi di programmi visti in passato? . . . 59
7.2.2.1 Attenzione al jump : confronto col C++ . . . 59
7.2.3 Riga di codice . . . 60
7.3 Direttive . . . 60
7.3.1 Dichiarazione di variabili . . . 60
7.3.1.1 Alternativa per dichiarare vettori (comando FILL) . . . 61
7.3.1.2 Codifica ASCII e caratteri speciali . . . 61
7.3.2 INCLUDE . . . 62
7.3.3 SET . . . 62
7.3.3.1 Calcolare memoria occupata . . . 62
7.3.3.2 Costanti numeriche . . . 63
7.4 Controllo di flusso . . . 63
7.4.1 if. . . then . . . else . . . 64
7.4.2 for. 64
7.4.3 do. . . while . . . 65
7.4.4 *Spaghetti-like* coding . . . 65
7.4.5 LOOP . . . 66
7.4.5.1 For discendente e for ascendente con LOOP . . . 66
7.4.6 LOOP condizionato . . . 66
7.4.7 Utilità dei LOOP e manipolazione stringhe . . . 67
7.4.8 Attenzione alla lunghezza dell'iterazione . . . 67
7.5 Sottoprogrammi e passaggio dei parametri . . . 67

8 Giovedì 08/10/2020 **68**
8.1 Uso dei registri ed effetti collaterali . . . 68
8.2 Sottoprogramma principale . . . 69
8.3 Dichiarazione e allocazione di spazio per la pila . . . 69
8.4 Ingresso/uscita e sottoprogrammi di utilità . . . 70
8.5 Osservazioni sulla tabella ASCII . . . 70
8.5.1 I/0 da tastiera e video . . . 71
8.5.2 File utility . . . 71
8.5.2.1 Sottoprogrammi di I/0 (su cui si basano i sottoprogrammi della sezione successiva) . . . 71
8.5.2.2 Sottoprogrammi a livello più alto . . . 72
8.5.2.3 Sottoprogrammi per l'ingresso/uscita di numeri esadecimali . . . 73
8.5.2.4 Sottoprogrammi per l'ingresso/uscita di numeri decimali . . . 74
8.6 Istruzioni che manipolano le stringhe . . . 75
8.6.1 Istruzioni *Direction Flag* (STD e CLD) . . . 76
8.6.2 MOVE DATA FROM STRING TO STRING . . . 76
8.6.3 LODSsuf (*load string*) e STOSsuf (*store string*) . . . 77
8.6.4 [Privilegiate] Istruzioni stringa per l'I/O - INSssuf, OUTSsuf . . . 78
8.6.5 COMPARE STRINGS (CMPSsuf) - confronto memoria-memoria . . . 78
8.6.6 SCAN STRING (SCASsuf) - confronto registro-memoria . . . 79
8.6.7 Prefissi di ripetizione . . . 80
8.6.8 Utilità delle due direzioni . . . 80
8.7 Ricapitoliamo . . . 82

9 Venerdì 09/10/2020 83
9.1 Conclusione su Assembler . . . 83
9.1.1 Differenze tra compilatore e assemblatore . . . 83
9.1.2 Tempo di esecuzione di un programma . . . 83
9.1.3 Lunghezza delle istruzioni e tempo di fetch . . . 84
9.1.4 Tempo di esecuzione delle istruzioni . . . 84
9.1.4.1 Come si evitano moltiplicazioni e divisioni? . . . 84

II Esercitazioni di Zippo 86

10 Martedì 06/10/2020 87
10.1 Assemblaggio . . . 87
10.1.1 File listato . . . 87
10.2 Debugging . . . 89
10.2.1 Comandi . . . 89

11 Venerdì 16/10/2020 95

12 Venerdì 23/10/2020 101
12.1 Esercizio sul fattoriale . . . 101
12.2 Calcolo del binomiale . . . 106

III Esercizi di Assembler 110

13 Simulazione di esame 111

14 Primo appello invernale 2021 113

15 Secondo appello invernale 2021 115

16 Terzo appello invernale 2021 (IL MIO) 118

17 Min e max con modulo e segno 121

18 Fattorizzazione 124

IV Reti combinatorie 126

19 Venerdì 09/10/2020 127
19.1 Introduzione alle reti logiche . . . 127
19.2 Reti combinatorie . . . 130
19.3 Algebra di Boole . . . 136

20 Martedì 13/10/2020 137
20.1 Continuiamo con l'Algebra di Boole . . . 137
20.1.1 Proprietà degli operatori di somma e prodotto . . . 138
20.1.2 Porte XOR e XNOR . . . 139
20.1.3 Teoremi di De Morgan con N variabili logiche . . . 140
20.1.3.1 Dimostrazione della prima tesi . . . 140
20.1.3.2 Dimostrazione della seconda tesi . . . 141
20.1.4 Equivalenza tra espressioni dell'Algebra di Boole e reti combinatorie . . . 141
20.1.5 Qualche osservazione sulla risoluzione di esercizi . . . 142

20.1.5.1 Dimostrazione di una identità 142
20.1.5.2 Semplificazione di espressioni 143
20.2 Reti combinatorie significative 145
20.2.1 Decoder 145
20.2.1.1 Decoder 2 to 4 145
20.2.1.2 Decoder 1 to 2 146
20.2.1.3 Decoder N to 2^N 147
20.2.2 Decoder con enabler (espandibile) 147
20.2.3 Costruzione di un decoder 4 to 16 da decoder 2 to 4 148
20.2.4 Demultiplexer 150
20.2.5 Multiplexer 151
20.2.6 Multiplexer come rete combinatoria universale 152

21 Mercoledì 14/10/2020 e Giovedì 15/10/2020 153
21.1 Modello strutturale universale 153
21.2 Sintesi di reti in forma SP a costo minimo 156
21.2.1 Metodo algoritmico per la lista degli implicanti principali 157
21.2.2 Metodo di Quine-McCluskey per la lista degli implicanti principali 159
21.2.3 Mappe di Karnaugh 160
21.2.4 Algoritmo di ricerca dei sottocubi principali (implicanti principali) partendo dalle mappe di Karnaugh 162
21.2.5 Ricerca di liste di copertura non ridondanti 163
21.2.5.1 Sintesi di leggi non completamente specificate 165
21.3 Sintesi di porte XOR, XNOR, NAND e NOR in forma SP 167

22 Martedì 20/10/2020 171
22.1 Sintesi di reti in formato PS 171
22.1.1 Sintesi duale 173
22.1.2 Sintesi meno costosa 173
22.2 Porte logiche universali 174
22.2.1 Costo a porte e costo a diodi 178

23 Mercoledì 21/10/2020 179
23.1 Porte tri-state 179
23.2 Circuiti di ritardo 182
23.2.1 Circuito di ritardo sul fronte di salita 182
23.2.2 Circuito di ritardo sul fronte di discesa 183
23.3 Formatore di impulsi 184
23.3.1 Formatore di impulsi sul fronte di salita 184
23.3.2 Formatore di impulsi sul fronte di discesa 185

V Aritmetica di un calcolatore 186

24 Mercoledì 21/10/2020 187
24.1 Rappresentazione dei numeri naturali 187
24.2 Teorema della divisione con resto 188
24.3 Notazione per indicare quoziente e resto 190
24.4 Proprietà dell'operatore modulo 190
24.4.1 Prima proprietà 190
24.4.2 Seconda proprietà 191
24.4.3 Terza proprietà 191
24.4.4 Quarta proprietà (dal libro di Corsini) 191

24.5 Pippe personali sui segni . . . 192
24.6 Rappresentazione dei numeri in qualunque base β . . . 194
24.7 Rappresentazione di naturali su un numero di cifre finito . . . 194
24.7.1 Quanti numeri possiamo rappresentare? . . . 194
24.7.2 Numero più grande rappresentabile? . . . 195
24.7.3 Numero di cifre richieste per rappresentare un numero . . . 195
24.8 Osservazione sugli esercizi della dispensa . . . 195
24.9 Elaborazione di numeri naturali tramite reti combinatorie . . . 196

25 Giovedì 22/10/2020 198
25.1 Complemento . . . 198
25.1.1 Circuito logico per l'operazione . . . 199
25.1.2 Complemento di numeri in base 10 con codifica *eccesso 3* . . . 200
25.2 Moltiplicazione e divisione per una potenza della base . . . 200
25.2.1 Conseguenza: operazioni di concatenamento e scomposizione . . . 201
25.3 Estensione di campo sui numeri naturali . . . 201
25.4 Addizione di numeri naturali . . . 202
25.4.1 Full adder in base 2 . . . 203
25.4.2 Incrementatore (*half adder*) . . . 206
25.5 Sottrazione . . . 208
25.5.1 Osservazione sulla complementazione nei pretest . . . 211
25.5.2 Comparatore di numeri naturali . . . 212
25.6 Moltiplicazione . . . 212

26 Martedì 27/10/2020 215
26.1 Moltiplicatore con addizionatore $n \times 1$ in base 2 . . . 215
26.2 Divisione . . . 216
26.2.1 Divisore elementare in base 2 . . . 219

27 Mercoledì 28/10/2020 220
27.1 Rappresentazione dei numeri interi . . . 220
27.1.1 Leggi di rappresentazione dei numeri interi . . . 221
27.1.1.1 Modulo e segno . . . 222
27.1.1.2 Traslazione . . . 222
27.1.1.3 Complemento alla radice . . . 223
27.1.2 Proprietà del complemento alla radice . . . 226
27.1.2.1 Determinazione del segno . . . 226
27.1.2.2 Legge inversa . . . 226
27.1.2.3 Forma alternativa della legge . . . 227
27.2 Operazioni su interi in complemento alla radice . . . 228
27.2.1 Valore assoluto . . . 228
27.2.2 Circuito di conversione da CR a MS . . . 229
27.2.3 Calcolo dell'opposto . . . 230
27.2.4 Estensione di campo . . . 232

28 Giovedì 29/10/2020 235
28.1 Riduzione di campo . . . 235
28.2 Moltiplicazione per potenza della base . . . 237
28.3 Divisione per potenza della base . . . 237
28.4 Assembler: Shift logico e aritmetico . . . 238
28.5 Somma . . . 239
28.6 Sottrazione . . . 241
28.6.1 Comparazione di numeri interi . . . 241

28.7 Confronto tra comparazioni 242
28.8 Moltiplicazione e divisione 242
28.8.1 Approccio 242
28.8.2 Circuito di conversione da MS a CR 243
28.8.3 Moltiplicazione 244
28.8.4 Divisione 245
28.8.4.1 Condizioni di fattibilità 247
28.8.4.2 Circuito 248

29 Riferimenti ad altri argomenti **249**
29.1 Assembler 249
29.2 Verilog (da *Circuiti logici per le operazioni sui numeri naturali e sui numeri interi* di Paolo Corsini) 251

VI Reti sequenziali **257**

30 Martedì 03/11/2020 **258**
30.1 Reti con memoria 258
30.2 Latch SR 259
30.2.1 Pilotaggio di un Lach SR 260
30.3 Il problema dello stato iniziale 261
30.3.1 Riprendiamo la sintetizzazione del Latch-SR 264
30.4 Tabelle di flusso e grafi di flusso 265
30.4.1 Tabella di flusso 265
30.4.2 Grafi di flusso 266
30.4.3 Diagramma di temporizzazione 266

31 Mercoledì 04/11/2020 **267**
31.1 D-latch trasparente 267
31.2 Reti trasparenti 269
31.3 D flip-flop 269
31.4 Memoria RAM statiche 272
31.5 Montaggio in parallelo di memorie RAM statiche 275
31.6 Montaggio in serie di memorie RAM statiche 275
31.7 Collegamento al bus e maschere 276

32 Giovedì 05/11/2020 **278**
32.1 Memorie ROM (*read-only*) 278
32.1.1 ROM programmabili 280
32.1.1.1 PROM 280
32.1.1.2 EPROM 280
32.1.1.3 EEPROM 281
32.2 Verilog 282
32.3 Reti sequenziali sincronizzate 287

33 Martedì 10/11/2020 **289**

- Descrizione in Verilog di registri.
- Regole di pilotaggio di una RSS, ritardi caratteristici.
- Primo esempio di RSS: Contatore. Descrizione e sintesi in Verilog (con mappe di Karnaugh).
- Contatori e divisione in frequenza.

34 Mercoledì 11/11/2020 **300**

- Registri multifunzionali: descrizione e sintesi in Verilog.
- Esempio di registro multifunzionale.
- Modello di Moore.
- Esempio di rete di Moore: Flip-flop JK. Descrizione e sintesi in Verilog (con mappe di Karnaugh).
- Riconoscitore di sequenze 11, 01, 10. Flip-flop JK come meccanismo di marcatura alternativo al D flip-flop.

35 Giovedì 12/11/2020 **310**

- Modello di Mealy.
- Contatore espandibile in base 3 realizzato come rete di Mealy. Descrizione e sintesi in Verilog.
- Differenze tra reti di Mealy e reti di Moore.

36 Martedì 17/11/2020 **317**

- Modello di Mealy ritardato.
- Osservazioni **vitali** sugli assegnamenti procedurali in Verilog (differenza rispetto al C++).
- Riconoscitore di sequenze come rete di Mealy ritardato.
- RSS complesse: modello generale.
- Primo esempio di RSS complessa: contatore di sequenze corrette 00, 01, 10.
- Linguaggio di trasferimento tra registri: nomenclatura e osservazioni.

37 Mercoledì 18/11/2020 **326**

- Esercizio: contatore di sequenze alternate 00,01,10-11,01,10
- Handshake /dav-rfd: regole e formatore di impulsi come esempio.
- Gestione dei cicli in microprogrammazione.

38 Giovedì 19/11/2020 **333**

- Handshake soc-eoc: regole e formatore di impulsi come esempio.
- Sintesi di RSS complesse: metodo della scomposizione in PO/PC.

39 Martedì 24/11/2020 **338**
39.1 Riepilogo sulla sintesi di una RSS compelssa in PO/PC 338
39.2 Tecniche ulteriori per la sintesi della parte controllo 340
39.2.1 Tecniche $\mu-$address e $\mu-$instruction based 340
39.3 Re-introduzione dei $\mu-$salti a più vie: registro MJR 342
39.3.1 Aggiornamento della sintesi PO/PC 343
39.4 Sottoliste utilizzando il registro MJR 344

40 [Stea] Riepilogo sulla descrizione e sintesi di reti logiche **345**

VII Struttura di un calcolatore **351**

41 Giovedì 26/11/2020 **352**

- Recap sul calcolatore di Von Neumann: sottosistema di I/O, memoria principale e processore (sEP8).
- Calcolatore visto dal programmatore,
- Linguaggio Assembler per il processore sEP8: opcode, formati.
- Architettura del calcolatore: processore e sottosistema di I/O visti come RSS, memoria principale vista come una RSA.

41.1 Ricapitoliamo sui formati 359

42 Martedì 01/12/2020 **360**

- Spazio di memoria.
- Spazio di I/O.
- Processore.
- Fasi del processore: reset iniziale, fase di fetch, fase di esecuzione, stato di blocco.
- Letture e scritture in memoria: temporizzazione ed esempio Verilog.
- Lettura e scritture nel sottosistema di I/O: temporizzazione ed esempio Verilog.
- Sottoprogrammi per operazioni di lettura o scrittura in memoria (su 1, 2, 3, 4 byte).
- Inizio della descrizione in Verilog del processore: reset, functions di supporto (valid_fetch, first_execution_state, alu_result, alu_flag, jmp_condition), fase di fetch.

42.1 Ricapitoliamo sulle operazioni di lettura e scrittura... 374
42.1.1 Memoria principale 374
42.1.2 Sottosistema di I/O 375
42.2 Ricapitoliamo sui sottoprogrammi per la lettura/scrittura 376
42.3 Corsini - Registri processore sEP8 a seguito della fase di fetch 378
42.4 Ricapitoliamo sugli stati della fase di fetch 379

43 Mercoledì 02/12/2020 **380**

- Fase di esecuzione.
- Introduzione alle interfacce: descrizione di un'interfaccia a livello funzionale, tipi di interfacce, necessità di introdurre meccanismi di handshake.

44 Giovedì 03/12/2020 **388**

- Interfaccia parallela di ingresso senza handshake.
- Interfaccia parallela di uscita senza handshake.
- Montaggio di interfacce parallele di ingresso e uscita.
- Interfaccia parallela di ingresso con handshake: descrizione Verilog della RSS che gestisce l'handshake.
- Interfaccia parallela di uscita con handshake: descrizione Verilog della RSS che gestisce l'handshake.
- Interfaccia parallela di ingresso-uscita con handshake.
- Interfacce seriali: nozioni di comunicazione seriale, descrizione Verilog di Trasmettitore e Ricevitore.

45 Martedì 09/12/2020 **401**
45.1 Conversione analogico/digitale e digitale/analogica 401
45.1.1 Convertitore D/A 403
45.1.1.1 Interfaccia per la conversione D/A 406
45.1.2 Convertitore A/D 407
45.1.2.1 Interfaccia di conversione A/D 410

Premessa

La dispensa vuole porsi come contributo agli studenti del corso di laurea triennale in Ingegneria informatica, presso la Scuola di Ingegneria dell'Università di Pisa.

Osservazione Gli appunti sono stati scritti durante le lezioni, approfondendo con il materiale ufficiale del corso.

Fonti Si segnala l'uso delle seguenti fonti:

- Dispense di Giovanni Stea caricate sul suo sito.
- Paolo Corsini, "Dalle porte AND, OR, NOT al Sistema calcolatore", edizioni ETS
- Paolo Corsini, "Circuiti logici per le operazioni sui numeri naturali e sui numeri interi", edizioni ETS
- Paolo Corsini, "Il Calcolatore Didattico C86.32", edizioni ETS (nuova edizione)

Dai libri del Corsini non sono riportati interi spezzoni, salvo un capitolo dove la cosa è segnalata. Le influenze maggiori stanno nella parte su Assembler (per quelle cose che il docente potrebbe non aver espresso con completa chiarezza) e nella parte su Struttura del calcolatore (per le immagini assenti dalla relativa dispensa).

Capitolo 1

Programma del corso

Il programma del corso prevede:

- Linguaggio Assembler (20 ore): quanto serve per scrivere un programma semplice, capire come programmi scritti in linguaggio ad alto livello vengano tradotti in linguaggio macchine
- La parte vera e propria di reti logiche:
  - Reti logiche: reti combinatorie, reti combinatorie per l'aritmetica, reti sequenziali asincrone e sincronizzate.
  - Microprogrammazione (*micro* sta per hardware) di reti sequenziali sincronizzate
  - Il calcolatore come esempio di rete sequenziale sincronizzata (cosa unica nel panorama universitario italiano).

Osservazione sugli esami Scritto e orale devono essere svolti nello stesso appello. L'unica eccezione riguarda gli appelli straordinari: persone che ne hanno diritto possono fare lo scritto all'ultimo appello della sessione e rimandare l'orale all'appello straordinario.

Perchè si parla di Reti logiche? In questo corso analizzeremo i circuiti da un punto di vista funzionale: non ci interessa come vengono realizzati (questo è compito dell'ingegnere elettronico), ma il loro comportamento. Il termine *logiche* è un richiamo all'hardware.

Difficoltà del corso Il corso non va sottovalutato, necessario seguire gli argomenti passo dopo passo: è molto facile che l'assenza di un argomento possa impedirci la comprensione di quelli successivi. Presente una complessità stratificata:

$$\text{Porte logiche } \longrightarrow \text{Reti logiche } \longrightarrow \text{Calcolatore}$$

Capitolo 2

Unimap

1. **Mar 29/09/2020 11:45-13:45 (2:0 h)** lezione: Lezione: Introduzione al corso ed informazioni pratiche. Richiami sulla rappresentazione dei numeri naturali ed interi in base 2. Schema a blocchi del calcolatore: memoria, spazio di I/O, processore. I registri del processore, condizioni al reset. (GIOVANNI STEA)

2. **Mer 30/09/2020 10:45-12:45 (2:0 h)** lezione: Codifica macchina e codifica mnemonica delle istruzioni e primo esempio di programma in Assembler. Indirizzamento degli operandi nelle istruzioni operative. Istruzioni di trasferimento (inizio). (GIOVANNI STEA)

3. **Gio 01/10/2020 10:45-12:45 (2:0 h)** lezione: Istruzioni di trasferimento (fine). Istruzioni aritmetiche (inizio). (GIOVANNI STEA)

4. **Ven 02/10/2020 15:00-17:00 (2:0 h)** lezione: istruzioni aritmetiche (fine). Istruzioni di traslazione e rotazione. Istruzioni logiche. Istruzioni di controllo. (GIOVANNI STEA)

5. **Mar 06/10/2020 11:45-13:45 (2:0 h)** lezione: Programmare in linguaggio Assembler: struttura sintattica di un programma. Direttive in Assembler: dichiarazione di variabile e di costante. Strutture di controllo di flusso tipiche dei linguaggi ad alto livello e loro traduzione in linguaggio Assembler. (GIOVANNI STEA)

6. **Mer 07/10/2020 10:45-12:45 (2:0 h)** esercitazione: (in copresenza con Ing. R. Zippo) Esercitazione Assembler. Presentazione ambiente di sviluppo. Uso del debugger per verifica del programma e controllo degli errori. (GIOVANNI STEA)

7. **Gio 08/10/2020 10:45-12:45 (2:0 h)** lezione: Sottoprogrammi e passaggio dei parametri. Gestione della pila. I/O e sottoprogrammi di ingresso/uscita. Istruzioni stringa. (GIOVANNI STEA)

8. **Ven 09/10/2020 15:00-17:00 (2:0 h)** lezione: Note sulla programmazione Assembler. Generalità sulle reti logiche. Modello, limiti del modello e non contemporaneità. descrizione mediante tabelle di verita'. Esempi di reti combinatorie semplici. AND e OR a piu' di due ingressi. Algebra di Boole (GIOVANNI STEA)

9. **Mar 13/10/2020 11:45-13:45 (2:0 h)** lezione: Algebra di Boole. Operatori e loro proprieta'. Equivalenza tra espressioni algebriche e sintesi di reti combinatorie. Reti combinatorie: decoder e decoder con enabler. Demultiplexer e Multiplexer. Il multiplexer come rete combinatoria universale. (GIOVANNI STEA)

10. **Mer 14/10/2020 10:45-12:45 (2:0 h)** lezione: Metodo strutturale universale per la sintesi di reti combinatorie: esempio con rete combinatoria. Sintesi a costo minimo in forma SP. Metodo algebrico: espansione di Shannon, forma canonica SP, mintermini, implicanti, implicanti principali. (GIOVANNI STEA)

11. **Gio 15/10/2020 10:45-12:45 (2:0 h)** lezione: Mappe di Karnaugh. Sintesi in forma SP a costo minimo: algoritmo per la ricerca dei sottocubi principali, classificazione, liste non ridondanti. Sintesi di leggi non completamente specificate. (GIOVANNI STEA)

12. **Ven 16/10/2020 15:00-17:00 (2:0 h)** esercitazione: Esercitazione: (in copresenza con Ing. R. Zippo) Esercitazione Assembler. Svolgimento di esercizi di programmazione. (GIOVANNI STEA)

13. **Mar 20/10/2020 11:45-12:45 (1:0 h)** lezione: Sintesi in forma PS. Sintesi a porte NAND e NOR. (GIOVANNI STEA)

14. **Mar 20/10/2020 12:45-13:45 (1:0 h)** esercitazione: Svolgimento di esercizi di sintesi in forma PS, a porte NAND, a porte NOR. (GIOVANNI STEA)

15. **Mer 21/10/2020 10:45-12:45 (2:0 h)** lezione: porte tri state. Circuiti di ritardo e formatori di impulsi. Introduzione all'aritmetica del calcolatore: rappresentazione dei numeri naturali. Teorema della divisione con resto. Reti combinatorie per i numeri naturali. (GIOVANNI STEA)

16. **Gio 22/10/2020 10:45-12:45 (2:0 h)** lezione: Aritmetica del calcolatore: circuito di complemento, moltiplicazione e divisione per potenze della base, concatenamento, estensione di campo. Somma: scomposizione ripple-carry e full adder. Incrementatore. Sottrazione e comparazione. Moltiplicazione (inizio). (GIOVANNI STEA)

17. **Ven 23/10/2020 15:00-17:00 (2:0 h)** esercitazione: (in copresenza con Ing. R. Zippo) Esercitazione Assembler. Svolgimento di esercizi di programmazione. (GIOVANNI STEA)

18. **Mar 27/10/2020 11:45-12:45 (1:0 h)** lezione: Aritmetica del calcolatore: moltiplicazione e divisione di numeri naturali. (GIOVANNI STEA)

19. **Mar 27/10/2020 12:45-13:45 (1:0 h)** esercitazione: Svolgimento di esercizi di aritmetica (GIOVANNI STEA)

20. **Mer 28/10/2020 10:45-12:45 (2:0 h)** lezione: Rapprsentazione dei numeri interi: leggi in traslazione, complemento alla radice, modulo e segno. Proprietà del complemento alla radice. Circuiti di calcolo del valore assoluto, dell'opposto, di estensione di campo. (GIOVANNI STEA)

21. **Gio 29/10/2020 10:45-12:45 (2:0 h)** lezione: Aritmetica dei numeri interi: riduzione di campo, moltiplicazione e divisione per potenza della base. Somma, sottrazione, moltiplicazione e divisione intera. Circuito per la conversione da modulo e segno a complemento alla radice. (GIOVANNI STEA)

22. **Ven 30/10/2020 15:00-17:00 (2:0 h)** esercitazione: Svolgimento di esercizi d'esame sull'aritmetica del calcolatore (GIOVANNI STEA)

23. **Mar 03/11/2020 11:45-13:45 (2:0 h)** lezione: Introduzione alle reti sequenziali: la funzione di memoria. Il Latch SR: struttura e pilotaggio. Il problema dello stato iniziale: circuito di reset e piedini per l'inizializzazione. Tabelle di flusso e grafi di flusso per la descrizione di reti sequenziali sincronizzate. Diagrammi di temporizzazione per la verifica delle descrizioni. (GIOVANNI STEA)

24. **Mer 04/11/2020 10:45-12:45 (2:0 h)** lezione: Elementi di memoria: D-latch e D-Flip-flop. Trasparenza e non trasparenza. Memorie RAM statiche: struttura e temporizzazione dei cicli di lettura e scrittura. Montaggi in serie/parallelo, aggancio al bus e maschere. (GIOVANNI STEA)

25. **Gio 05/11/2020 10:45-12:45 (2:0 h)** lezione: Memorie ROM, PROM, EPROM, EEPROM. Introduzione al linguaggio Verilog: sintassi per il collegamento di moduli, descrizione e sintesi di reti combinatorie. Introduzione alle reti sequenziali sincronizzate: i registri. (GIOVANNI STEA)

26. **Ven 06/11/2020 15:00-17:00 (2:0 h)** esercitazione: (in copresenza con ing. Raffaele Zippo): Presentazione ambiente Icarus Verilog e GTKWave. Esempio temporizzazione con ripple carry adder. (GIOVANNI STEA)

27. **Mar 10/11/2020 11:45-12:45 (1:0 h)** lezione: Reti sequenziali sincronizzate: temporizzazione dei registri. Generalità e disuguaglianze di temporizzazione. (GIOVANNI STEA)
28. **Mar 10/11/2020 12:45-13:45 (1:0 h)** lezione: Esempi di reti sequenziali sincronizzate: i contatori. Descrizione e sintesi di vari tipi di contatore in Verilog. (GIOVANNI STEA)
29. **Mer 11/11/2020 10:45-11:45 (1:0 h)** lezione: Reti di Moore: modello, descrizione e temporizzazione. (GIOVANNI STEA)
30. **Mer 11/11/2020 11:45-12:45 (1:0 h)** lezione: Reti di Moore: Sintesi del FF JK. Descrizione e sintesi del riconoscitore di sequenze. (GIOVANNI STEA)
31. **Gio 12/11/2020 10:45-11:45 (1:0 h)** lezione: Reti di Mealy: generalità e temporizzazione. (GIOVANNI STEA)
32. **Gio 12/11/2020 11:45-12:45 (1:0 h)** esercitazione: Descrizione e sintesi del riconoscitore di sequenze come rete di Mealy. Svolgimento di esercizi sulle reti di Mealy. (GIOVANNI STEA)
33. **Ven 13/11/2020 15:00-17:00 (2:0 h)** esercitazione: (in copresenza con ing. Raffaele Zippo): svolgimento di esercizi su reti combinatorie per l'aritmetica in ambiente Verilog. (GIOVANNI STEA)
34. **Mar 17/11/2020 11:45-13:45 (2:0 h)** lezione: Modello di Mealy ritardato: descrizione, temporizzazione, proprietà. Linguaggio di trasferimento tra registri: generalità e primi esempi. (GIOVANNI STEA)
35. **Mer 18/11/2020 10:45-12:45 (2:0 h)** esercitazione: Svolgimento di esercizi di descrizione di reti sequenziali sincronizzate contenenti handshake e temporizzazioni (GIOVANNI STEA)
36. **Gio 19/11/2020 10:45-11:45 (1:0 h)** lezione: Tecnica di scomposizione in parte operativa e parte controllo di reti sequenziali sincronizzate complesse (GIOVANNI STEA)
37. **Gio 19/11/2020 11:45-12:45 (1:0 h)** lezione: Esempi di scomposizione in parte operativa e parte controllo di reti sequenziali sincronizzate complesse (GIOVANNI STEA)
38. **Ven 20/11/2020 15:00-17:00 (2:0 h)** esercitazione: (in copresenza con l'ing. Raffaele Zippo). Svolgimento di esercizi di descrizione di reti sequenziali sincronizzate al calcolatore (GIOVANNI STEA)
39. **Mar 24/11/2020 11:45-12:45 (1:0 h)** lezione: Sintesi euristica della parte controllo: modelli basati su microindirizzi e microistruzioni. Il Multiway Jump Register e le sottoliste. (GIOVANNI STEA)
40. **Mar 24/11/2020 12:45-13:45 (1:0 h)** esercitazione: Svolgimento di esercizi di esame (GIOVANNI STEA)
41. **Mer 25/11/2020 10:45-12:00 (1:30 h)** esercitazione: Esercizi di descrizione e sintesi di reti sequenziali sincronizzate. (GIOVANNI STEA)
42. **Gio 26/11/2020 10:45-12:45 (2:0 h)** lezione: Il calcolatore visto come un insieme di moduli interconnessi: processore, spazio di memoria e spazio di I/O. Visione funzionale del processore didattico sEP8 (8 bit simple Educational Processor): i registri, le istruzioni, inizializzazione al reset. Modalità di indirizzamento degli operandi. Linguaggio Assembler e linguaggio macchina. Formato delle istruzioni. (GIOVANNI STEA)
43. **Ven 27/11/2020 15:00-17:00 (2:0 h)** esercitazione: (in copresenza con ing. Raffaele Zippo): svolgimento di esercizi di descrizione e sintesi di reti sequenziali e combinatorie in ambiente Verilog. (GIOVANNI STEA)
44. **Mar 01/12/2020 11:45-13:45 (2:0 h)** lezione: descrizione del calcolatore: memoria, spazio di I/O, processore. Descrizione dei registri del processore. Letture e scritture in memoria e nello spazio di I/O. Descrizione della fase di reset e di fetch in Verilog. (GIOVANNI STEA)

45. **Mer 02/12/2020 10:45-12:45 (2:0 h)** lezione: Descrizione del processore: fase di esecuzione. Interfacce: visione funzionale. Interfacce con e senza handshake. Accesso a controllo di programma. (GIOVANNI STEA)

46. **Gio 03/12/2020 10:45-12:45 (2:0 h)** lezione: Interfacce parallele con e senza handshake, ingresso ed uscita. Trasmissione seriale start/stop. Interfaccia seriale. (GIOVANNI STEA)

47. **Ven 04/12/2020 15:00-17:00 (2:0 h)** esercitazione: (in copresenza con ing. Raffaele Zippo). Svolgimento di esercizi di descrizione di reti complesse al calcolatore. (GIOVANNI STEA)

48. **Mer 09/12/2020 10:45-12:45 (2:0 h)** lezione: Descrizione del trasmettitore e ricevitore seriale. Conversione analogico/digitale e digitale/analogica. Convertitore D/A ed interfaccia di conversione. (GIOVANNI STEA)

49. **Gio 10/12/2020 10:45-12:45 (2:0 h)** lezione: Convertitore analogico/digitale e relativa interfaccia di conversione. Chiusura del corso. (GIOVANNI STEA)

50. **Ven 11/12/2020 15:00-17:00 (2:0 h)** esercitazione: (in copresenza con l'ing. Raffaele Zippo): svolgimento di esercizi di descrizione e sintesi al calcolatore. (GIOVANNI STEA)

51. **Mar 15/12/2020 11:45-13:45 (2:0 h)** esercitazione: (in copresenza con ing. Raffaele Zippo): svolgimento di esercizi al calcolatore. (GIOVANNI STEA)

52. **Mer 16/12/2020 10:45-12:45 (2:0 h)** lezione: simulazione esame scritto (GIOVANNI STEA)

53. **Gio 17/12/2020 10:45-11:15 (0:30 h)** esercitazione: ripasso su argomenti del programma (GIOVANNI STEA)

Parte I

Assembler

Capitolo 3

Martedì 29/09/2020

3.1 Linguaggio assembler

Nome preciso Il nome preciso è *Assembly*, ma lo chiameremo *Assembler* per ragioni storiche (a Pisa è sempre stato chiamato così)

Differenza da altri linguaggi Il linguaggio Assembler è di basso livello, in contrapposizione al C++ che è di alto livello. Con questo linguaggio saremo in grado di scrivere **istruzioni macchina** eseguite dal processore.

- Con linguaggio ad alto livello intendiamo un linguaggio i cui costrutti sono più vicini al ragionamento dell'essere umano
- Con linguaggio a basso livello intendiamo un linguaggio i cui costrutti sono più vicini al ragionamento della macchina.

Scrivere in assembler significa imparare a ragionare come la macchina!

Sintassi simbolica Con Assembler non scriveremo in linguaggio macchina (sequenze di zeri e uno incomprensibili per l'uomo), ma ricorreremo a una sintassi simbolica. Qual è la differenza?

- In C++ il compilatore effettua una traduzione vera e propria. Basti pensare che dietro l'istruzione

  ```
  a=b;
  ```

  possono celarsi centinaia di istruzioni macchina.

- In Assembler abbiamo l'**assemblaggio**, svolto dall'assemblatore. Ponendo quanto segue

  ```
  MOV %AX, %BX
  ```

  rappresentiamo un numero binario. Questa è una traduzione 1 : 1!

Ulteriori differenze rispetto al C++

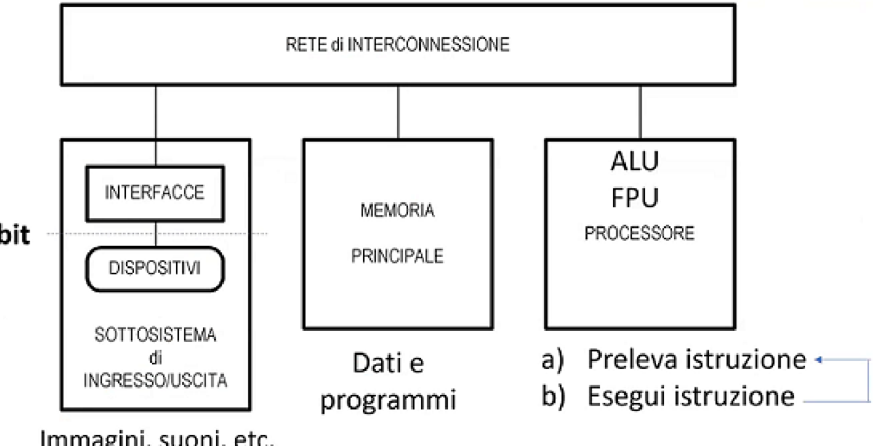
- Non sono presenti i cosiddetti *costrutti di flusso strutturato*: for, while, dowhile, ifthenelse, case. Saranno presenti istruzioni di salto *jump*.
- Le variabili non hanno un tipo. Avremo stringhe di bit. Cosa rappresenti una variabile è pensiero esclusivamente nostro, non della macchina

Assembler processor-specific L'assembler è specifico per ogni processore: i comandi possono cambiare. Segue che non sia un linguaggio, portabile, a differenza del C++ o dei linguaggi che impareremo simultaneamente a progettazione web. Utilizzeremo, in questo corso, assembler per *Intel x86*, visto la validità dei suoi principi per quasi tutte le versioni di Assembler. Non lo studieremo tutto, considerando la vastità del manuale (20 ore per 2000 pagine? impensabile)

Utilità di Assembler Assembler è utilizzato per *sistemi Embedded*. Al di là di questo un ingegnere informatico deve sapere come si comporta un processore!

3.2 Struttura di un calcolatore

Un calcolatore può essere immaginato come una serie di blocchi connessi da una *rete di interconnessione* detta **bus**. In particolare individuiamo nello schema funzionale:



1. **Sottosistema di ingresso/uscita**: area che si occupa di compiere la traduzione dal mondo esterno al calcolatore e viceversa. Immagini, suoni, qualunque contenuto vengono codificate in stringhe di bit. Nella stessa area abbiamo il dispositivo (che non è detto faccia sia input che output, pensiamo alla tastiera e al mouse): questo non è collegato direttamente al bus, ma passa per *interfacce*, reti logiche che adattano il dispositivo al bus (attraverso protocolli il calcolatore può interfacciarsi con i dispositivi senza conoscerne la struttura fisica)
2. **Memoria principale**: ospita i programmi (le istruzioni da eseguire) e i dati (non tutti, alcuni di questi potrebbero trovarsi nel sottosistema introdotto prima)
3. **Processore**: contiene al suo interno almeno le seguenti unità:

- la ALU (*Arithmetic Logic Unit*): unità che si occupa di eseguire istruzioni logiche (AND, OR e NOT) e aritmetiche (relativamente a numeri naturali e interi)
- la FPU (*Floating Point Unit*): unità che svolge istruzioni aritmetiche relative ai numeri reali

Cosa fa il processore Il processore, ciclicamente:

- preleva un'istruzione macchina dalla memoria
- la esegue

3.3 Ripasso: rappresentazione dell'informazione

3.3.1 Numeri naturali

Con N bit possiamo rappresentare 2^N numeri naturali, precisamente quelli compresi nell'intervallo $[0, 2^N - 1]$. Come nella base 10 anche qua abbiamo un sistema basato sulla presenza di cifre più significative e cifre meno significative (rispettivamente MSB, *Most significant bit*, e LSB, *Least Significant Bit*). Un numero in base 2 avente la seguente forma

$$b_{N-1}, b_{N-2}, \ldots, b_1, b_0$$

può essere rappresentato in base 10 mediante la seguente formula

$$X = \sum_{i=0}^{N-1} b_i \, 2^i$$

ricordiamo, inoltre, l'algoritmo **DIV&MOD** per convertire numeri dalla base 10 alla base 2 (vedere dispensa di Rappresentazione dell'informazione, FdP).

3.3.2 Numeri interi

Con N bit possiamo rappresentare 2^N numeri interi, precisamente quelli compresi nell'intervallo $[-2^{N-1}, 2^{N-1} - 1]$. Osserviamo che la cosa non è simmetrica:

- Con $N = 8$ abbiamo $[-128, 127]$
- Con $N = 16$ abbiamo $[-32768, 32767]$

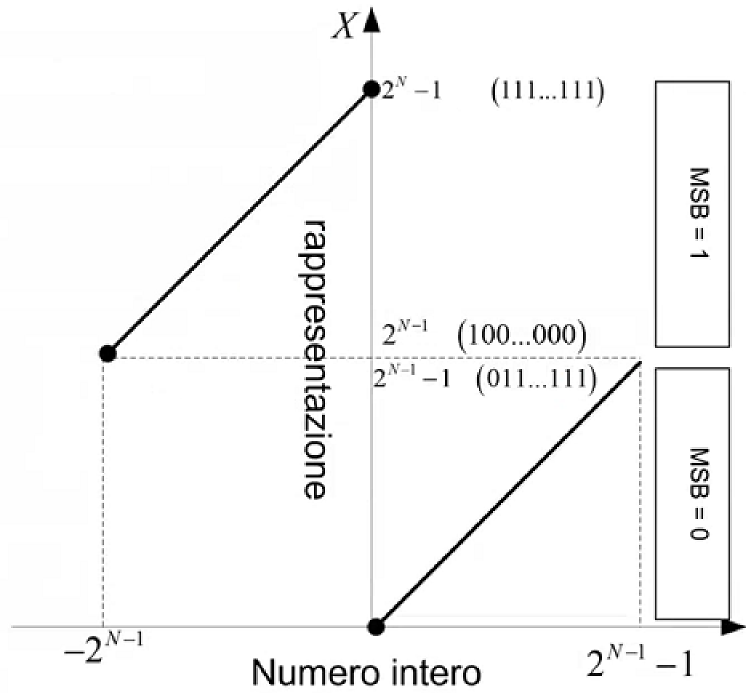
La rappresentazione che ci interessa è quella **in complemento a 2**. Per ottenere un numero da base 10 a base 2 procederemo identificando la seguente stringa di bit X (numeri naturali per rappresentare numeri interi)

$$X = \begin{cases} x & x \geq 0 \\ 2^N + x & x < 0 \end{cases} \quad \text{purchè si abbia } x \in [-2^{N-1}, 2^{N-1} - 1]$$

ricordiamo che la cosa deve essere verificata, il metodo ci porta ad ottenere una sequenza binaria in ogni caso. Se la condizione non è soddisfatta quella sequenza non rappresenta il numero da cui siamo partiti. Tutto questo può essere rappresentato anche così: $X = |x|_{2^N}$. Vedremo questa notazione nella parte di Aritmetica del corso.

3.3.2.1 Diagramma della farfalla

Quanto detto prima può essere tradotto sfruttando la seguente immagine



Lungo l'asse delle ascisse abbiamo gli interi x, lungo l'asse delle ordinate le stringhe di bit X (naturali, capiremo per bene nella parte di Aritmetica). Osserviamo che

- Nel semiasse positivo abbiamo una retta con coefficiente $m = 1$ dove $X = x$. Quest'area consiste nei numeri positivi ed è quella con MSB = 0
- Nel semiasse negativo abbiamo una retta parallela alla precedente traslata di 2^N ($X = x + 2^N$). Quest'area consiste nei numeri negativi ed è quella con MSB = 1

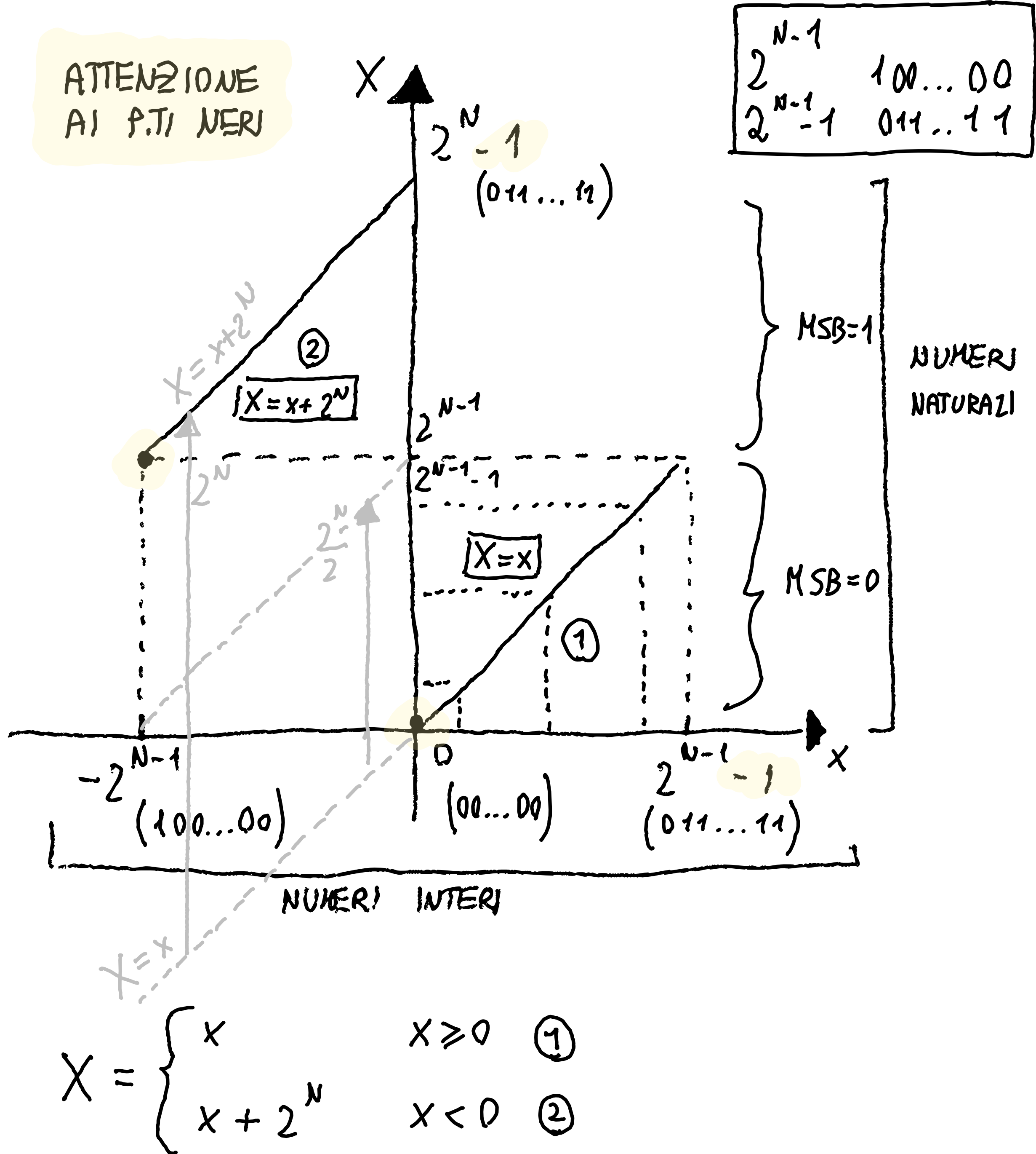
Ricordiamo che l'ultimo numero con $MSB = 0$ $(2^{N-1} - 1)$ sarà del tipo $011\dots11$, mentre l'ultimo numero con $MSB = 0$ $(2^N - 1)$ sarà del tipo $111\dots11$. Dallo stesso disegno possiamo capire il processo inverso per passare da base 2 a base 10

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

ottengo il naturale *complementando* i bit della rappresentazione (cioè invertendo gli zeri e gli uno). Spiegheremo questa cosa nella parte di Aritmetia del corso! Prendiamo il seguente esempio:

$$\begin{aligned} &10110101 \\ &-(01001010 + 1) \\ &-(01001011) = -(64 + 8 + 2 + 1) = -75 \end{aligned}$$

DISEGNO DELLA FARFALLA (IN C2)



$$X = \begin{cases} x & x \geq 0 \quad ① \\ x + 2^{N} & x < 0 \quad ② \end{cases}$$

① CHIARAMENTE UNA RETTA CON $m = 1$

$[0, 2^{N-1}-1]$

② RETTA TRASLATA

3.3.2.2 Tabella di conversione da -8 a $+7$ in $p = 4$

$\mathbb{N}$	**A**	**a**
0	0000	$+0$
1	0001	$+1$
2	0010	$+2$
3	0011	$+3$
4	0100	$+4$
5	0101	$+5$
6	0110	$+6$
7	0111	$+7$
8	1000	-8
9	1001	-7
10	1010	-6
11	1011	-5
12	1100	-4
13	1101	-3
14	1110	-2
15	1111	-1

3.3.2.3 Rappresentazione esadecimale

Molto spesso le sequenze binarie sono lunghe e difficili da comprendere. Potremo compattare convertendo in base 10, ma questo richiede dei calcoli. Proprio per questo andiamo ad adottare la **notazione esadecimale**. 4 bit consistono in un numero compreso tra 0 e 15, sfruttando una scorciatoia (descritta nella dispensa di rappresentazione dell'informazione, FdP) possiamo sostituire in modo rapido leggendo blocchi di cifre da destra verso sinistra.

Quadr.	Simb.	Quadr.	Simb.	Quadr.	Simb.	Quadr.	Simb.
0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F

Esempio 1 $\underbrace{1011}_{\text{B}}|\underbrace{1001}_{9} \Longrightarrow 0xB9$

Esempio 2 $\underbrace{1100}_{\text{C}}|\underbrace{0001}_{1} \Longrightarrow 0xC1$

3.4 Osservazione sul calcolo di MB, KB o GB occupati

Ricordarsi che

$$\boxed{2^{10} = 1\text{ KB}} \quad \boxed{2^{20} = 1\text{ MB}} \quad \boxed{2^{30} = 1\text{ GB}}$$

Esempio Definiamo lo spazio di memoria come l'insieme di 2^{32} locazioni.

$$2^{32} = 2^{30} \cdot 2^{2} = 4\text{ GB}$$

3.5 Struttura del calcolatore vista da un programmatore ASS

Cosa vede un programmatore quando programma un calcolatore? Essenzialmente tre cose: spazio di memoria, spazio di I/0 e processori.

Spazio di memoria Con spazio di memoria intendiamo una sequenza lineare e contigua di locazioni: ciascuna di esse ha capacità di un byte ed è identificata da un numero naturale a 32bit detto indirizzo. Se abbiamo 32 bit significa che potremo esprimere 2^{32} indirizzi diversi, quindi avremo 2^{32} locazioni possibili (si va da $0x000\ldots00$ a $0x111\ldots11$, se ho 2^{32}). Solitamente si accede alla memoria per le seguenti motivazioni:

- Prelevare istruzioni
- Prelevare gli operandi delle istruzioni (la maggior parte si trova in memoria, molto spesso nel processore - in particolare nella ALU)

può svolgere accessi di lettura o di scrittura, possibili su singola locazione (operando a 8 bit), doppia locazione (operando a 16 bit) e quadrupla locazione (32 bit).

Memoria del computer La memoria del computer si divide in

- **RAM**, *Random Access Memory.* Questa memoria è di tipo volatile, cioè le informazioni vengono mantenute finchè c'è tensione. C'è un problema: cosa fa il processore al momento dell'accensione se l'accesso è appunto casuale? Questo non può succedere e richiede l'introduzione della...
- **ROM**, *Read Only Memory.* Memoria di sola lettura, non volatile, che contiene il programma eseguito dal processore al momento dell'accensione.

Tipi di accessi Accessi di tipo diverso, dato lo stesso indirizzo, restituiscono un contenuto diverso. Negli accessi a 16/32bit si utilizza l'indirizzo più piccolo delle 2/4 locazioni. L'indirizzo più grande contiene i bit più significativi, segue quanto presente nell'immagine

	b7 … b0
0x3F65B432	0x1C
0x3F65B433	0x39
0x3F65B434	0xA2
0x3F65B435	0xC6

Lettura a	Indirizzo	Contenuto
8 bit (byte)	0x3F65B432	0x1C
16 bit (word)	0x3F65B432	0x391C
32 bit (double word)	0x3F65B432	0xC6A2391C

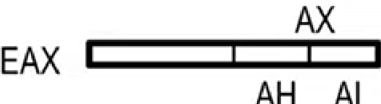
Dimensione dello spazio di memoria Che dimensione hanno 2^{32} bit? 4GB di memoria! Come faccio a sapere se posso riferire un indirizzo (in un calcolatore non sono presente solo io coi miei programmi, c'è anche la ROM, per esempio)? Non è un problema nostro: gli indirizzi in Assembler sono simbolici, ci pensa l'assemblatore a mappare la variabile su una cella utilizzabile.

Spazio di I/0 Lo spazio di Input/Output consiste in $2^{16} = 64$k locazioni (dette anche porte). Ciascuna porta ha una capacità pari ad un byte ed è indirizzabile mediante un indirizzo a 16bit. Possiamo accedere allo spazio di I/O usando due particolari istruzioni (IN e OUT). Contrariamente alle celle dello spazio di memoria le porte di I/0 non sono intercambiabili: a un certo indirizzo abbiamo la tastiera, a un altro il mouse. Segue la necessità di conoscere gli indirizzi! Generalmente

- si eseguono istruzioni IN per prelevare uno o più byte da un dispositivo: dati da elaborare o semplicemente informazioni sullo stato del dispositivo.
- si eseguono istruzioni OUT per trasmettere uno o più byte a un dispositivo: dati elaborati o informazioni per modificare lo stato del dispositivo.

Processore Il processore consiste in una collezione di registri. Un registro consiste in una piccola locazione di memoria, avente 32 bit. Si hanno due tipi di registri:

- **Registri generali**. Si osserva che tutti i registri sono introdotti dalla lettera E: questa sta per *Extended*. Precedentemente un registro occupava solo 16bit, successivamente sono stati estesi a 32 bit. La E risolve anche questioni non banali di compatibilità: è possibile richiamare l'intero registro con i suoi 32 bit col nome EXY, ma possiamo riferire la parte bassa dei registri col nome vecchio (XY)!



Risulta possibile richiamare la prima parte bassa e la seconda parte bassa con XH ed YL, dove H sta per *High* ed L sta per *Low*.

I nomi dei registri sono i seguenti:

- **EAX: Accumulator.** È il registro che serve per fare calcoli aritmetici. Alcune istruzioni aritmetiche lo usano per contenere operandi e risultati
- **EBX: Base.** Veniva spesso usato come indirizzo di base per l'accesso in memoria
- **ECX: Counter.** Viene usato come contatore nei cicli (la variabile "i" del for viene spesso mappata sul registro CX o parti di esso)
- **EDX: Data.** Anche questo viene usato come operando da istruzioni aritmetiche.
- **ESI: Source Index.** Veniva usato come registro indice per accessi in memoria
- **EDI: Destination Index.** Come sopra
- **EBP: Base pointer.** Veniva usato come registro "base" per accessi in memoria.

Alcuni di questi sono utilizzati per particolari funzioni:

- EAX è detto registro accumulatore ed è usato da alcune istruzioni aritmetiche (per esempio per variabili incremento)

- ESI, EDI, EBX ed EBP sono detti registri puntatore, dove B sta per *base* ed I sta per *indice*.
- ESP, utilizzato per indirizzare la pila. Solitamente si utilizza per gestire sottoprogrammi

Per comprendere l'utilità di alcuni registri leggere sull'*indirizzamento di memoria*.

- **Registri di stato**. I registri di stato sono due:
  - EIP, cioè *Instruction Pointer register* (detto anche *program counter*). Consiste nel registro contenente la locazione di memoria a partire dalla quale sarà prelevata la prossima istruzione. Il suo contenuto è fissato al momento dell'accensione del processore! Raffiniamo quanto detto all'inizio: il processore
    1. preleva dalla memoria, precisamente all'indirizzo EIP, la nuova istruzione
    2. incrementa EIP del numero di byte dell'istruzione che ha prelevato (si parla di istruzioni contigue, ovviamente la cosa cambia se si incontrano salti)
    3. esegue l'istruzione e ritorna allo step (1)

    EIP inizialmente vale $0xFFFF0000$: la prima istruzione si trova lì. Ovviamente le celle successive dovranno essere implementate in ROM.

  - EF, cioè *Extended Flag register*. Il registro consiste in 32 elementi detti *flag*. Ci interessano, in particolare, i seguenti flag:
    * OF (Overflow): elemento impostato ad 1 quando il risultato generato dall'ultima istruzione *trabocca*
    * SF (Sign): elemento impostato ad 1 quando il risultato generato dall'ultima istruzione restituisce un qualcosa con $MSB = 1$
    * ZF (Zero): elemento impostato ad 1 quando il risultato generato dall'ultima istruzione restituisce un qualcosa con tutti bit nulli
    * CF (Carry): elemento impostato ad 1 quando il risultato generato dall'ultima istruzione genera un riporto o un prestito.

    Per gli interi ci serviranno OF, SF, ZF (CF è inutile), per i naturali CF e ZF (OF ed SF sono inutili). Nel registro dei flag quelli che ci interessano valgono 0, inizialmente.

Capitolo 4

Mercoledì 30/09/2020

4.1 Codifica macchina e codifica mnemonica

Presente a pagina 7 della dispensa di Assembler.

4.2 Struttura di un'istruzione

Le istruzioni sono formate da tre campi:

- Il **codice operativo**, il nome dell'istruzione che vogliamo eseguire
- Il **suffisso di lunghezza** (lunghezza degli operandi): sono ammesse come lunghezze B (byte, 8it), W (word, 16bit), L (long, 32bit). Non dobbiamo porre il suffisso obbligatoriamente: ci pensa l'assemblatore se la lunghezza degli operandi può essere intuita (per esempio quando almeno uno degli operandi è un registro).
- **Operandi**: le istruzioni ammettono 0/2 operandi (sono poche le istruzioni con 0 operandi). Nelle istruzioni con due operandi si distingue l'operando **sorgente** dall'operando **destinatario**. Se si ha un solo operando quello è detto **source** (destinatario è implicito) o **destinatario** (se l'istruzione lavora con un solo operando e pone nell'operando stesso il risultato del suo lavoro). I due operandi (salvo rare eccezioni) hanno la stessa lunghezza! Gli operandi possono essere in registri o porte I/O, ma possono essere anche costanti!

Un esempio di istruzione è la seguente

```
ADD %BX, pippo
```

il contenuto presente all'indirizzo di una locazione di memoria (*pippo*) viene incrementato del valore contenuto nella libreria BX. Il linguaggio Assembler consente al programmatore di riferire le locazioni di memoria con nomi simbolici che l'assemblatore tradurrà in indirizzi. L'assemblatore si incaricherà di sostituire lo stesso indirizzo tutte le volte che trova scritto *pippo*.

4.3 Esempio di programma

Presente un esempio di programma, accompagnato da spiegazione, da pagina 8 a pagina 11 della dispensa di Assembler.

4.4 Memoria occupata da una certa istruzione

Sia sul libro di Corsini che sulla dispensa di Stea si definisce un'istruzione come una stringa che occupa da 1 a 14 byte. Tenendo conto delle cose appena introdotte osserviamo che

- Il numero di operandi è uno degli indicatori dello spazio occupato da una certa istruzione (è ovvio che un'istruzione senza operandi possa occupare meno di un'istruzione con un operando, stessa cosa se mettiamo a confronto istruzioni con uno e due operandi, rispettivamente)
- Lo spazio occupato dagli operandi dipende dall'indirizzamento adottato: nel caso di indirizzamento diretto abbiamo una costante; in tutti gli altri casi andiamo a salvare l'indirizzo dell'area di memoria in cui si trova un certo operando.
- Ricordiamo che un processore, nell'ordine:
  - estrae da EIP l'indirizzo dell'istruzione successiva
  - incrementa EIP in modo tale che questa punti alla nuova istruzione successiva
  - esegue l'istruzione estratta poco fa ed eventualmente preleva dalla memoria gli operandi necessari per eseguire l'istruzione (usando gli indirizzi salvati)
- Ricordarsi le dimensioni di un'istruzione: sono allocati al più 4byte per Displacement! Segue che non è possibile, in un'istruzione a due operandi, fare indirizzamenti di memoria in entrambi gli operandi.

4.5 Indirizzamento degli operandi

Ricordiamo la struttura di un'istruzione

```
OPCODEsize source, destination
```

4.5.1 Indirizzamento di registro

L'indirizzamento **NON** è possibile con i registri di stato. Possiamo scegliere tra i seguenti registri generali:

- 8 registri a 32bit (EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP)
- 8 registri a 16 bit (AX, BX, CX, DX, SI, DI, SP , BP)
- 8 registri ad 8 bit (AH, BH, CH, DH, AL, BL , CL, DL)

operandi di registro possono essere sia sorgente che destinatario

```
OPCODE %DI
OPCODE %EAX, %EBX
OPCODE %AH, %CL
```

la prima istruzione lavora su un operando a 16bit, la seconda su operandi a 32bit, la terza su operandi ad 8 bit. Noi non sappiamo quali sono le istruzioni (e quindi la dimensione che occuperanno queste istruzioni), ma sappiamo già che potremo omettere il suffisso di dimensione (grazie alla presenza di almeno un operando con indirizzamento di registro).

4.5.2 Indirizzamento immediato

Questo tipo di indirizzamento può essere usato solo nell'operando sorgente: scriviamo una costante direttamente nell'istruzione (per ovvie ragioni non ha senso porre una costante nel destinatario).

```
OPCODE $0x20, %AL
OPCODE $0x5683a20b, %ECX
```

nella prima lavoro su operandi ad 8 bit, nella seconda su operandi a 32bit.

4.5.3 Indirizzamento di memoria

L'indirizzamento di memoria è complicato, ma fondamentale: il processore, per la maggior parte del tempo, copia pezzi di memoria da una parte a un'altra. L'indirizzamento di memoria è possibile sia con l'operando sorgente che con quello destinatario, ma non è possibile farlo in entrambi in una stessa istruzione (l'assemblatore da errore se ci proviamo).

Caso più generale Il caso più generico di indirizzo è il seguente

$$\text{Indirizzo} = |\text{base} + \text{indice} \times \text{scala} \pm \text{displacement}|_{\text{modulo}_{2^{32}}}$$

cioè

```
OPCODEsfx +-disp(base,indice,scala)
```

- la base e l'indice consistono in registri generali a 32 bit. Precedentemente era obbligatorio porre un registro B in base e un registro I in indice: oggi si offre maggiore flessibilità e si possono utilizzare tutti i registri generali.
- scala è una costante che può avere per valore 1 (valore default se non indicato), $2, 4, 8$.
- displacement è una costante intera.

4.5.3.1 Indirizzamento di tipo diretto

Nelle cose viste fino ad ora abbiamo fatto indirizzamenti di memoria diretti, cioè indirizzamenti con solo il displacement.

```
OPCODEW 0x00002001
```

4.5.3.2 Indirizzamento di tipo indiretto

Specifico un solo registro, precisamente un registro puntatore.

Registro base Poniamo la cosa nella seguente forma

```
OPCODEL (%EBX)
```

dove EBX consiste nel registro contenente l'indirizzo. Attenzione: è necessario indicare il suffisso di lunghezza, non abbiamo un indirizzamento di registri.

Registro indice Se volessi indicare un registro indice pongo

```
OPCODEL (,%ESI, 4)
```

la scala moltiplica solo l'indice e non la base.

4.5.3.3 Indirizzamento con displacement e registro di modifica

```
OPCODEW 0x002A3A2B (%EDI)
```

Indirizzo un operando a 16bit, che si trova nella doppia locazione il cui indirizzo si ottiene sommando (modulo 2^{32}) il displacement e il contenuto di EDI. Questa cosa è molto versatile per i vettori

4.5.3.4 Indirizzamento bimodificato senza displacement

```
OPCODEW (%EBX, %EDI)
OPCODEW (%EBX, %EDI, 8)
```

utilizzo due registri puntatori ponendo, eventualmente, la scala.

4.5.3.5 Indirizzamento bimodificato con displacement

In questo caso utilizzeremo tutte le armi a nostra disposizione

```
OPCODEB 0x002F9000 (%EBX, %EDI)
OPCODEB -0x9000 (%EBX, %EDI)
```

4.5.4 Indirizzamento delle porte di I/O

L'indirizzo di I/O può avvenire sia con la sorgente che col destinatario, ma mai con entrambi. L'indirizzamento può essere diretto o indiretto con registro puntatore

- Possibili porre solo indirizzi su 8bit nell'indirizzamento diretto (perchè la macchina fornisce solo 8bit)
- L'indirizzamento indiretto è possibile solo usando il registro DX

Vediamo i seguenti esempi

```
IN 0x001A, %AL
IN (%DX), %AX
OUT %AL, 0x003A
OUT %AL, (%DX)
```

La prima istruzione preleva un operando a 8bit dalla porta di I/0 0x001A e pone il contenuto presente a quell'indirizzo nel registro AL, la terza pone il contenuto del registro Al nella porta di I/0 avente indirizzo 0x003A.

4.6 Principali istruzioni

Si distinguono

- Istruzioni operative
  - Istruzioni di trasferimento
  - Istruzioni aritmetiche
  - Istruzioni di traslazione/rotazione
  - Istruzioni logiche
- Istruzioni di controllo
  - Istruzioni di salto
  - Istruzioni per la gestione di sottoprogrammi

Per ogni istruzione diremo:

- il formato
- cosa fa
- se ci sono aggiornamenti di flag
- le modalità di indirizzamento ammesse per gli operandi

le istruzioni sono spiegate in modo dettagliato nella bibbia di Corsini.

4.6.1 Istruzioni di trasferimento

Ribadiamo che gli spostamenti possibili sono i seguenti

Memoria ⟷ Registro | Registro ⟶ Registro | I/0 ⟷ Registro

non è possibile fare trasferimenti da memoria a memoria. Le istruzione di trasferimento, inoltre, non modificano i flag.

4.6.1.1 MOVE

- **FORMATO**: MOV source, destination
- **AZIONE**: Sostituisce l'operando destinatario con una copia dell'operando sorgente
- **FLAG** di cui viene modificato il contenuto: Nessuno.

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

4.6.1.2 LOAD EFFECTIVE ADDRESS

- **FORMATO**: LEA source, destination
- **AZIONE**: Sostituisce l'operando destinatario con l'espressione indirizzo contenuta nell'operando sorgente.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

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

```
MOV 0x00213AB1(%EAX,%EBX,4),%ECX
LEA 0x00213AB1(%EAX,%EBX,4),%ECX
```

Col primo esempio intendiamo *copia l'indirizzo 0x00002000 nel registro EDX*. col secondo intendiamo *copia il risultato dell'operazione di indirizzamento di memoria nel registro ECX*.

Differenza tra MOV e LEA La MOV copia il contenuto posto a quell'indirizzo, la LEA copia l'indirizzo!

4.6.1.3 EXCHANGE

- **FORMATO**: XCHG source, destination
- **AZIONE**: Sostituisce all'operando destinatario una copia dell'operando sorgente e viceversa.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

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

- La XCHG **modifica il sorgente**. È l'unica istruzione che lo fa.
- Fatto curioso (per adesso): in Assembler si possano scambiare due operandi con una sola istruzione (**non trasparenza** dei registri)

```
.GLOBAL _main

.TEXT
_main:  MOV $1, %EAX
        MOV $2, %ECX
        XCHG %EAX, %ECX
        RET

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

```
7       .TEXT
8       _main:  MOV $1, %EAX
9               MOV $2, %ECX
10              XCHG %EAX, %ECX
11              RET
12
13      .INCLUDE "C:/amb_GAS/utility"
(gdb) i r
eax            0x2      2
ecx            0x1      1
edx            0x33f    831
ebx            0x102e   4142
esp            0x8e144  0x8e144
ebp            0x8e158  0x8e158
esi            0x54     84
```

Capitolo 5

Giovedì 01/10/2020

5.1 Concludiamo con le istruzioni di trasferimento

5.1.1 INPUT e OUTPUT

- **FORMATO**:

```
IN indirizzo, %AL
IN indirizzo, %AX
IN (%DX),%AL
IN (%DX),%AX
```

- **AZIONE**: Sostituisce il contenuto del registro destinatario (AL, AX) con il contenuto di un adeguato numero di porte consecutive. L'indirizzo della prima (ed eventualmente unica) porta è specificato direttamente nell'istruzione (primi due formati) o è contenuto nel registro DX (ultimi due formati); i primi **due** formati possono essere utilizzati solo per individuare porte con indirizzo inferiore a 256.
- **FLAG** di cui viene modificato il contenuto: Nessuno.
- **FORMATO**:

```
OUT     %AL, indirizzo
OUT     %AX, indirizzo
OUT     %AL,(%DX)
OUT     %AX,(%DX)
```

- **AZIONE**: Copia il contenuto del registro sorgente (AL, AX) in un adeguato numero di porte consecutive. L'indirizzo della prima (ed eventualmente unica porta) è specificato direttamente nell'istruzione (primi due formati) o è contenuto nel registro DX (ultimi due formati); i primi tre formati possono essere utilizzati solo per individuare porte con indirizzo inferiore a 256.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Gli operandi nell'I/O si possono trasferire solo nei/dai registri generali. Non si fanno operazioni sulle porte! Inoltre, gli unici registri utilizzabili sono AL, AX (come sorgente o destinatario) e DX (come registro puntatore).

5.1.1.1 ASSEMBLER non è linguaggio ortogonale

Contrariamente al C++ Assembler non può essere definito un linguaggio ortogonale (in C++ posso mettere qualunque variabile se la sintassi mi consente di scrivere una variabile). Cose che sono fatte con un certo registro generale non è detto possano essere fatte con altri registri.

5.1.1.2 Uscita da registro a porta

Supponiamo di voler passare il contenuto del registro BX (si possono effettuare trasferimenti solo dai registri AL e AX) alla porta con indirizzo 0x3142. Scriveremo le seguenti istruzioni

```
MOV %BX, %AX
MOV $0x3142, %DX
MOV %AX, (%DX)
```

ricordiamo che non è possibile fare trasferimenti da memoria a memoria. Inoltre, nei formati dove la sorgente (input) o il destinatario (output) sono indirizzi è necessario che l'indirizzo sia inferiore a 256.

5.1.2 Pila

Abbiamo già visto la pila a Fondamenti di programmazione: una struttura dati dove il contenuto è gestito secondo la regola LIFO (*Last in First Out*, cioè l'ultimo elemento inserito è il primo ad andarsene). Al di là della questione C++, la pila è essenziale per il funzionamento del calcolatore, precisament per annidare sottoprogrammi (L'assembler stesso è organizzato per sottoprogrammi)

- Quando avvio un sottoprogramma salvo l'indirizzo di ritorno, cioè quello dell'istruzione successiva e lo pongo nella pila (*push*)
- Quando termino l'esecuzione del sottoprogramma faccio *pop*, cioè estraggo dalla pila l'ultimo indirizzo inserito (quello della prossima istruzione da eseguire).

La pila permette di chiamare sottoprogrammi all'interno di altri sottoprogrammi.

Puntatore al top della pila Il registro ESP (*Extended stackpointer*) mi permette di puntare all'elemento più alto della pila.

***push* value** Decremento ESP in modo tale che punti alla zona di memoria immediatamente superiore, copio un certo valore in %ESP

***push* dest** Copio il contenuto di %ESP nel destinatario e incremento ESP in modo tale da puntare alla zonna di memoria immediatamente inferiore.

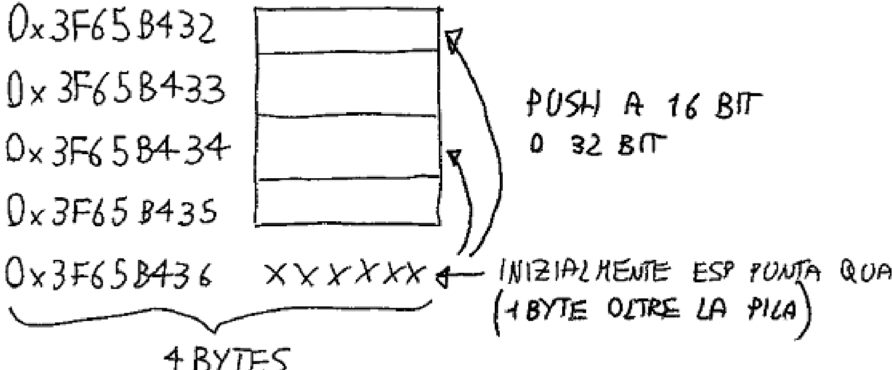
ESP deve essere inizializzato Ogni programma che utilizza la pila deve inizializzare ESP assegnando un valore sensato.

- **FORMATO**: `PUSH source`
- **AZIONE**: Salva nella pila corrente una copia dell'operando sorgente (che <u>deve</u> essere a **16 o a 32 bit**). Più in dettaglio, compie le seguenti azioni: i) decrementa l'indirizzo contenuto nel registro ESP di due o di quattro; ii) memorizza una copia dell'operando sorgente nella doppia o quadrupla locazione il cui indirizzo è contenuto in ESP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

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

- **FORMATO**: `POP destination`
- **AZIONE**: Rimuove dalla pila corrente una word o un long e la sostituisce all'operando destinatario. Più in dettaglio, compie le seguenti azioni: i) sostituisce all'operando destinatario una copia del contenuto della doppia o della quadrupla locazione il cui indirizzo è contenuto in ESP, ii) incrementa di due o di quattro l'indirizzo contenuto in ESP, rimuovendo in tal modo dalla pila la word o il long copiato.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Operandi	Esempi
Memoria	POPW 0x02AB2000
Registro Generale	POP %BX

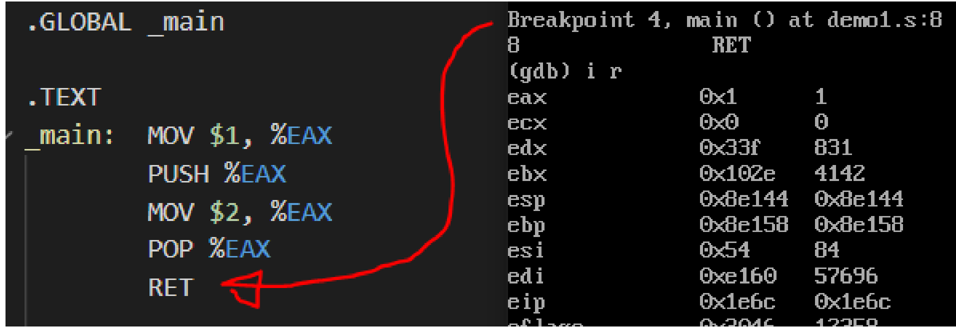


Gli operandi salvati ed ottenuti dalla pila devono essere a 16 o a 32 bit. Se non si ha un registro generale come operando è necessario indicare il suffisso di lunghezza (per forza W o L, come già detto).

5.1.2.1 Pila come memoria temporanea

La pila può essere usata come *parcheggio di dati*. I registri generali sono pochi e in alcune istruzioni sono gli unici operandi possibili. In alcune circostanze, addirittura, potrei essere costretto a usare lo stesso registro per due scopi diversi in momenti diversi (e se volessi recuperare il dato trovato nel primo momento?).

Esempio banalissimo della POP Osserviamo il valore di EAX dopo aver eseguito la POP:



Il valore di EAX, dopo la PUSH, è 2. Con la POP recupero il valore impostato precedentemente.

5.1.3 PUSHAD e POPAD

Significato Per il prof A dovrebbe stare per ALL e D per double. Non è sicuro.

- **FORMATO**: `PUSHAD`
- **AZIONE**: Salva nella pila corrente una copia del contenuto degli 8 registri generali a 32 bit, rispettando il seguente ordine: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

- **FORMATO**: `POPAD`
- **AZIONE**: Rimuove dalla pila 8 long e con essi rinnova il contenuto degli 8 registri generali a 32 bit, rispettando il seguente ordine: EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX.
- **FLAG** di cui viene modificato il contenuto: Nessuno
  - Si noti che ESP **non viene sovrascritto** (sarebbe un problema se lo fosse).

Chiaramente l'ESP verrà modificato non perchè poniamo noi un valore, ma perchè eseguiamo l'operazione POP (quindi L'ESP sarà incrementato quanto necessario).

5.2 Istruzioni aritmetiche

Introduzione a pagina 13 sulle dispensa di Assembler

5.2.1 ADD e SUBTRACT

Spiegazioni con algoritmo e flag coinvolte da pagina 13 a pagina 17 della dispensa di Assembler.

$$\boxed{\text{dest} += \text{src}} \quad \boxed{\text{dest} -= \text{src}}$$

Osservazioni

- Gli algoritmi, sia per l'addizione che per la sottrazione, sono gli stessi sia nei naturali che negli interi (ovviamente negli interi solo con rappresentazione in complemento a 2.
- Le proprietà tipiche della rappresentazione in C2 (spiegate nella parte relativa all'aritmetica) chiariscono perchè si possa utilizzare la stessa circuiteria, quindi svolgere le operazioni su naturali e interi utilizzando la stessa istruzione.
- La differenza tra lo svolgere operazioni su interi o naturali sta nei flag da osservare: la circuiteria modifica, in qualunque caso, tutti i flag relativi. Solo NOI sappiamo se quel numero è naturale o intero, segue che siamo noi (con le operazioni successive) a scegliere quali flag leggere (e quindi se trattare il risultato come un intero o un naturale).
- Precisamente:
  - Nei naturali controlliamo i flag CF, SF e ZF. In particolare se CF = 1 significa che l'ultimo riporto da una cifra in più al risultato finale, quindi siamo usciti dall'intervallo di rappresentazione.
  - Negli interi controlliamo i flag OF, SF e ZF. Il CF può avere valore uguale ad 1 come prima, ma non è più indicatore della validità del risultato. Andremo a vedere la Overflow flag, che
    * è uguale ad 1 se nell'addizione si è avuto riporto con operandi di segno concorde.
    * è uguale a 0 se gli operandi dell'addizione sono di segno discorde (il riporto si ignora)
    * è uguale ad 1 se nella sottrazione si è avuto prestito con operandi di segno discorde
    * è uguale a 0 se gli operandi della sottrazione sono di segno concorde (il prestito si ignora)
- In caso di Overflow negli interi il MSB indicherà un segno sbagliato.

Relativamente alla sottrazione se scriviamo l'operazione possiamo ricondurci ai casi dell'addizione (somma di elementi concordi, non sempre rappresentabile, o somma di elementi discordi, sempre rappresentabile).

5.2.2 INCREMENT e DECREMENT

Queste due istruzioni esistono per ragioni soprattutto storiche: molti anni fa la circuiteria più lenta era quella che svolgeva somme e sottrazioni. Avere una funzione di incremento e decremento permetteva di rendere più veloci certe operazioni. Oggi non è più la stessa cosa parlare di costi di operazione: i processori, in molte situazioni, eseguono operazioni in parallelo.

- **FORMATO**: `INC destination`
- **AZIONE**: Equivale all'istruzione `ADD $1,destination`, con la sola differenza che il contenuto del flag CF non viene modificato.
- **FLAG** di cui viene modificato il contenuto: OF, SF e ZF.

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

- **Più compatta**, in quanto nella ADD andrebbe comunque specificata una costante sul numero di bit del destinatario (8, 16, 32)
- Tanti (tanti) anni fa, era anche più veloce (vedremo perché)
- **FORMATO**: `DEC destination`
- **AZIONE**: Equivale all'istruzione `SUB $1,`*`destination`* con la sola differenza che il contenuto del flag CF non viene modificato.
- **FLAG** di cui viene modificato il contenuto: OF, SF e ZF.

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

Osservazione La increment e la decrement non modificano il CF. Questa cosa è evidente nell'esercizio a pagg.32-33 della dispensa di Assembler (programma che permette di verificare se la stringa di bit è palindroma). Nel ciclo viene utilizzata la decrement: se questa modificasse il CF ci sarebbero problemi.

5.2.3 ADD WITH CARRY e SUBTRACT WITH BORROW

Informazioni presenti a pagina 65 e 66 della dispensa di Assembler.

- HO OPERANDI A 64 BIT
- LA ADD LAVORA FINO A 32 BIT
- DIVIDO GLI OPERANDI IN 2 PARTI

1° OPERANDO	5 6 A 9 C 2 D 4	6 7 A 4 3 B 5 F +
2° OPERANDO	4 4 B 9 A 5 A 4	A 6 B 4 C 5 5 A =
	9 B 6 3 6 8 7 9	0 E 5 9 0 0 B 9
	POI QUESTE DUE	PRIMA SOMMO QUESTE DUE PARTI

AFFINCHÉ LA COSA ABBIA SENSO DEVO SOMMARE, NELLA SECONDA SOMMA, ANCHE IL RIPORTO. QUINDI:

- NELLA PRIMA SOMMA USO IL SOLITO ADD
- NELLA SECONDA USO ADC: SOMMA COME IN ADD, PIÙ IL RIPORTO FINALE DELLA PRECEDENTE SOMMA

5.2.4 NEGATE

- **FORMATO:** `NEG destination`
- **AZIONE:** Interpreta l'operando destinatario come un numero intero e lo sostituisce con il suo opposto. Qualora l'operazione non sia possibile, mette ad 1 il contenuto del flag OF. Mette ad 1 il contenuto del flag CF eccetto quando l'operando è zero: in tal caso lo mette a 0.
- **FLAG** di cui viene modificato il contenuto: Tutti.

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

Algoritmo Con la NEG si ottiene l'opposto (ovviamente la cosa va fatta con un destinatario concepito come intero)

- Complementando bit a bit il destinatario e
- aggiungendo uno

```
1   .GLOBAL _main
2
3   .TEXT
4   _main:  MOV $255, %EAX
5           NEG %EAX
6           RET
7
8   .INCLUDE "C:/amb_GAS/utility"
9
```

```
Continuing.

Breakpoint 4, main () at DEMO1.S:6
6               RET
(gdb) i r
eax            0xffffff01       -255
ecx            0x0      0
edx            0x33f    831
ebx            0x102e   4142
esp            0x8e144  0x8e144
ebp            0x8e158  0x8e158
esi            0x54     84
edi            0xe160   57696
```

Attenzione all'overflow Abbiamo che l'intervallo di rappresentabilità in C2, dato un numero N di bit, è il seguente

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

non possiamo calcolare l'opposto dell'estremo negativo (2^{N-1} non appartiene all'intervallo)! In questo caso l'OF sarà uguale ad uno.

5.2.5 COMPARE

- **FORMATO**: `CMP  source, destination`
- **AZIONE**: Verifica se l'operando destinatario è maggiore, uguale o minore dell'operando sorgente, sia interpretando gli operandi come numeri naturali che come numeri interi. Aggiorna poi il contenuto dei flag tenendo conto del risultato della verifica (i due operandi rimangono inalterati). L'esatto algoritmo di aggiornamento del contenuto dei flag è noioso da descriversi: l'aggiornamento è comunque consistente con l'interpretazione che del contenuto dei flag sarà data dall'istruzione di salto condizionato (vedi avanti), che in un programma sensato segue sempre l'istruzione CMP.
- **FLAG** di cui viene modificato il contenuto: Tutti.

Operandi	Esempi
Memoria, Registro Generale	CMP 0x00002000,%EDX
Registro Generale, Memoria	CMP %CL,0x12AB1024
Registro Generale, Registro Generale	CMP %AX,%DX
Immediato, Memoria	CMPB $0x5B,(%EDI)
Immediato, Registro Generale	CMP $0x45AB54A3,%EAX

75

Presente a pagina 17 della dispensa di Assembler l'algoritmo. Si considera che

- Con ZF = 1 si ha destination = source
- Con SF = 1 si ha risultato della sottrazione negativa, quindi destination $<$ source
- Con SF = 0 si ha risultato della sottrazione positiva, quindi destination $>$ source

Noi non controlleremo direttamente i flag ma utilizzeremo delle JUMP CONDIZIONATE. Nella dispensa e nel libro di Corsini sono presenti tutti i JUMP possibili.

5.2.6 INTEGER e INTEGER MULTIPLY

Questa istruzione è concepita diversamente dalle altre operazioni:

- Il risultato di una somma sta su N o $N + 1$ cifre
- Il prodotto di due numeri a N cifre sta su $2N$ cifre. Questo dettaglio pone un problema non secondario: contrariamente alla somma non posso comparare fattori e risultato. Segue che un operando non potrà essere utilizzato sia come fattore che come risultato.

Potrei pensare a un'istruzione a tre operandi, ma cose del genere in assembler non esistono. Si risolve specificando un solo operando (l'operando sorgente): l'altro operando e la sede del risultato sono impliciti. Si osserva che i risultati delle moltiplicazioni, in 16 e 32bit, vengono divisi tra due registri. In 32bit la cosa è necessaria, ma in 16bit a cosa serve? Non potrei fare...?

```
EAX=AX* source
```

Anche in questo caso le motivazioni sono storiche: i registri precedentemente erano a 16bit, e il ragionamento che si faceva era lo stesso fatto con gli operandi a 32bit. Con l'estensione dei registri si è mantenuta la cosa per compatibilità. Un trucco per porre il risultato in un registro a 32bit è il seguente

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

Pongo nella pila prima la parte alta (cifre più significative), poi la parte bassa (cifre meno significative), poi le estraggo dalla pila insieme (nei comandi push abbiamo indicato due registri a 16bit, col comando pop indichiamo un registro a 32bit).

- **FORMATO**: `MUL source`
- **AZIONE**: Considera l'operando sorgente come un moltiplicando e l'operando destinatario (implicito) come un moltiplicatore ed effettua l'operazione di moltiplicazione, interpretando gli operandi come numeri naturali.

  Se l'operando sorgente (moltiplicando) è:
  1) ad 8 bit, allora carica in AX il prodotto di AL e source
  2) a 16 bit, allora carica in DX_AX il prodotto di AX e source
  3) a 32 bit, allora carica in EDX_EAX il prodotto di EAX e source
- **FLAG** di cui viene modificato il contenuto: CF, OF: vengono messi a 1 se il risultato non sta sul numero di bit di source, e a zero altrimenti. SF e ZF sono **indefiniti.**

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

- **FORMATO**: `IMUL source`
- **AZIONE**: Considera l'operando sorgente come un moltiplicando e l'operando destinatario (implicito) come un moltiplicatore ed effettua l'operazione di moltiplicazione, interpretando gli operandi come numeri interi.

  Se l'operando sorgente (moltiplicando) è:
  1) ad 8 bit, allora carica in AX il prodotto di AL e source
  2) a 16 bit, allora carica in DX_AX il prodotto di AX e source
  3) a 32 bit, allora carica in EDX_EAX il prodotto di EAX e source
- **FLAG** di cui viene modificato il contenuto: CF, OF, SF (non attendibile) e ZF (non attendibile).

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

Quindi

- Uno dei due operandi è un registro, l'altro è indicato nell'istruzione (source)
- L'operando source determina il numero di bit che andremo ad utilizzare per il risultato, quindi i registri dove saranno salvati i risultati.
- Se source è a 16 o 32 bit i risultati saranno divisi tra due registri: con 32bit è necessario, in 16bit si ha questa cosa per ragioni storiche.

Non attendibile? Molto difficile da capire cosa succeda coi flag. Inutile guardare, non ci dovrebbero servire (cit.Stea)

5.2.7 DIVIDE e INTEGER DIVIDE

La divisione presenta qualche problemino ulteriore rispetto alla moltiplicazione:

- I risultati sono due: quoziente e resto
- l'operazione non è fattibile se il divisore vale zero.

Dato un dividendo X e un divisore Y individuiamo che $0 \leq R \leq Y - 1$ (può essere grande quanto il divisore) e $0 \leq Q \leq X$ (può essere grande quanto il dividendo).

Istruzione Il risultato è diviso tra due registri in tutte le versioni possibili. Ricordiamo che dobbiamo salvare quoziente e divisore.

Osservazione Risulta ovvio che la dimensione del divisore (source) risulta inferiore al dividendo.

- Se il divisore è di 8bit il dividendo sarà di 16bit
- Se il divisore è di 16bit il dividendo sarà di 32bit
- Se il divisore è di 32bit il dividendo sarà di 64bit

negli ultimi due casi il dividendo sarà preso dai due registri divisi.

Wait Ma non si era detto che il quoziente può stare al massimo nel numero di bit del dividendo? Cosa succede se il quoziente non è rappresentabile sul registro designato? Se il quoziente della divisione non sta sul numero di bit previso dal formato viene sollevata un'eccezione (la stessa eccezione che partirebbe con una divisione per 0 - il programma si inchioda). Per evitare problemi del genere dobbiamo scegliere un'adeguata versione della divisione

Esempio di divisione Supponiamo di voler fare 15.000 per 3 (abbiamo un dividendo di 16bit e un divisore di 8bit). Se noi dividiamo otteniamo come risultato 5000. Questo è problematico se adottiamo la divisione con source ad 8bit, segue che le seguenti istruzioni non vanno bene!

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

La cosa può essere risolta così:

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

Abbiamo modificato il contenuto dei registri AX e DX: questo è sufficiente per scegliere la divisione con source a 16bit. La terza istruzione è vitale, e molto spesso viene dimenticata (cit.)

• **FORMATO**: `DIV source`

• **AZIONE**: Considera l'operando sorgente come un divisore e l'operando destinatario (implicito) come un dividendo ed effettua l'operazione di divisione, interpretando gli operandi come numeri naturali. Più in dettaglio, compie le azioni che seguono.

i) Se l'operando sorgente (divisore) è:

1) ad 8 bit, allora divide AX per il sorgente, mettendo il quoziente in AL ed il resto in AH

2) a 16 bit, allora divide DX_AX per il sorgente, mettendo il quoziente in AX ed il resto in DX

3) a 32 bit, allora divide EDX_EAX per il sorgente, mettendo il quoziente in EAX ed il resto in EDX

ii) Se il quoziente non è esprimibile su un numero di bit pari a quello del divisore, allora genera un'*interruzione interna* in conseguenza della quale viene messo in esecuzione un opportuno sottoprogramma. Il contenuto dei flag e dei registri destinati a contenere il quoziente ed il resto è, in tal caso, non significativo.

• **FLAG** di cui viene modificato il contenuto: Tutti, ma in modo non attendibile.

Operandi	Esempi
Memoria	DIVB (%ESI)
Registro Generale	DIV %ECX

• **FORMATO**: `IDIV source`

• **AZIONE**: Considera l'operando sorgente come un divisore e l'operando destinatario (implicito) come un dividendo ed effettua l'operazione di divisione, interpretando gli operandi come numeri interi [...]

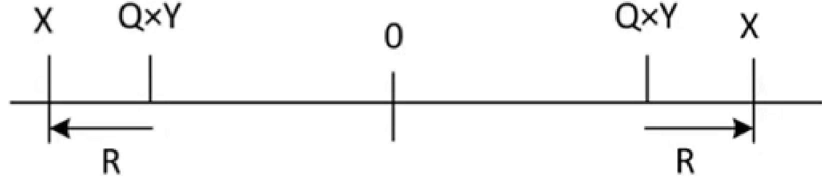
• **FLAG** di cui viene modificato il contenuto: Tutti, ma in modo non attendibile.

Operandi	Esempi
Memoria	IDIVB (%ESI)
Registro Generale	IDIV %ECX

• Speculare alla precedente, lavora su operandi **interi**

Divisione intera

- Nella divisione intera il resto ha sempre il segno del dividendo e in modulo è minore rispetto a quello del divisore.




```
-7 idiv 3 : quoziente -2, resto -1
7 idiv -3: quoziente -2, resto +1
```

- Ciò significa che il quoziente viene approssimato per troncamento, cioè viene sempre approssimato all'intero più vicino allo zero. In entrambi gli esempi ottengo che $Q \times Y$, cioè il prodotto tra quoziente e divisore, è più vicino allo zero rispetto al dividendo.

- La cosa è inconsistente con le nozioni di algebra che conosciamo: per capire la differenza pensiamo alla definizione di resto.

$$|-4|_3 = |3 \cdot (-2) + 2|_3 = |2|_3 = 2 \qquad |4|_3 = 1$$

5.2.8 Conclusioni

- Scegliere con cura la versione della divisione (basandoci sulle ipotesi in mano nostra)
- I registri devono essere azzerati prima del calcolo, altrimenti i risultati potrebbero essere inconsistenti (chi ce lo dice che il registro con le cifre più significative presenti bit nulli?)
- Ricordarsi che il contenuto di DX o EDX viene modificato dalle operazioni

Per la moltiplicazione:

- Se l'unico operando esplicito (un fattore) è a 8bit allora il risultato della moltiplicazione (l'altro fattore è un registro a 8bit) sarà posto in un registro a 16bit.
- Se l'unico operando esplicito (un fattore) è a 16bit allora il risultato della moltiplicazione (l'altro fattore è un registro a 16bit) sarà diviso in due registri a 16 bit (abbiamo quindi un risultato a 32bit).
- Se l'unico operando esplicito (un fattore) è a 32bit allora il risultato della moltiplicazione (l'altro fattore è un registro a 32bit) sarà diviso in due registri a 32 bit (abbiamo quindi un risultato a 64bit).

Per la divisione:

- Se l'unico operando esplicito (divisore) è a 8bit allora quoziente e resto saranno memorizzati su registri a 8bit (il dividendo è a 16bit).
- Se l'unico operando esplicito (divisore) è a 16bit allora quoziente e resto saranno memorizzati su registri a 16bit (il dividendo è a 32 bit, diviso su due registri).
- Se l'unico operando esplicito (divisore) è a 32bit allora quoziente e resto saranno memorizzati su registri a 32bit (il dividendo è a 64bit, diviso su due registri).

Capitolo 6

Venerdì 02/10/2020

6.1 Continuiamo con le istruzioni aritmetiche

6.1.1 Estensione di campo

Con estensione di campo intendiamo un'operazione con cui si rappresenta un numero usando più cifre.

6.1.2 Naturali

Nei numeri naturali l'operazione è banale, mi basta aggiungere gli zeri a sinistra. Segue una cosa del seguente tipo

$$100110 \longrightarrow \boxed{0}100110$$

6.1.3 Interi

Nei numeri interi non possiamo fare la stessa cosa: il MSB indica il segno, quindi non posso aggiungere zeri (andrei ad alterare il segno del numero). Il metodo adottato consiste nel ripetere il bit più significativo

$$100110 \longrightarrow \boxed{1}100110$$

Il motivo di questo metodo è chiaro quando passiamo da rappresentazione in base 2 di un intero negativo alla base 10.

6.1.4 CONVERT WORD TO DOUBLEWORD in EAX

- **FORMATO**: `CWDE`
- **AZIONE**: Interpreta il contenuto di AX come un numero intero a 16 bit, rappresenta tale numero su 32 bit e quindi lo memorizza in EAX.
- **FLAG** di cui viene modificato il contenuto: Nessuno

- Esempio: voglio sommare due interi, uno si trova in AX ed uno in EBX.

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

6.2 Istruzioni di traslazione

Le istruzioni di traslazione permettono la variazione dell'ordine di bit in un operando destinatario. Solitamente si hanno due formati

```
OPCODE src, dest
OPCODE dest
```

col primo formato indichiamo, attraverso src (immediato o registro CL, di cui si considerano solo i 5bit più bassi), quante volte vogliamo ripetere l'operazione; col secondo ci limitiamo ad eseguire l'operazione una sola volta (è come se ponessi src = 1). Ovviamente src deve valere al più 31: porre un src > 31 ha poco senso.

6.2.1 SHIFT LOGICAL LEFT

- **FORMATO**: `SHL source, destination`
  `SHL destination`
- **AZIONE**: Interpreta l'operando sorgente come un numero naturale *n* e, per *n* volte, sostituisce il bit contenuto in CF e ciascun bit dell'operando destinatario con il bit che gli è immediatamente a destra, considerando il bit più significativo dell'operando destinatario immediatamente a destra del bit contenuto in CF e sostituendo con 0 il bit meno significativo dell'operando destinatario. Pone infine a 0 il contenuto del flag OF.
- **FLAG** di cui viene modificato il contenuto: Tutti.

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

- Lo shift sinistro *n* volte equivale a **moltiplicare il destinatario per 2^n**
  - Vale in base 2 come in base 10
- Non è segno di grande furbizia usare una MUL quando si può usare una SHL
  - SHL è più veloce, e per un programmatore è estremamente più semplice
- CF viene sovrascritto *n* volte (difficile capire se il risultato è corretto)

Attenzione In alcuni casi lo shift può essere utile sul piano dell'efficienza (occupo meno spazio con l'operazione ed eseguo un'operazione più veloce). In altri casi può essere una condanna a morte: quando si utilizza questa istruzione ci si sbarazza ogni volta di un bit, quindi si perde informazione. Segue che una moltiplicazione per $2^3 = 8$ sia possibile con lo shift left solo se il risultato non esce dal numero di bit a disposizione.

6.2.2 SHIFT ARITHMETIC LEFT

- **FORMATO**: `SAL source, destination`
  `SAL destination`

AZIONE: la stessa cosa della SHL

- Sulla dispensa sono riportate alcune differenze minori, ma è un errore (SHL e SAL hanno lo stesso OPCODE sul manuale Intel)

- La SAL moltiplica per 2^{src} un operando **intero**
  - Se MSB cambia valore almeno una volta, OF=1 (risultato non attendibile)

6.2.3 SHIFT LOGICAL RIGHT

- **FORMATO**: `SHR source, destination`
  `SHR destination`
- **AZIONE**: Interpreta l'operando sorgente come un numero naturale *n* e, per *n* volte, sostituisce il bit contenuto in CF e ciascun bit dell'operando destinatario con il bit che gli è immediatamente a sinistra, considerando il bit meno significativo dell'operando destinatario immediatamente a sinistra del bit contenuto in CF e sostituendo con 0 il bit più significativo dell'operando destinatario. Pone infine a 0 il contenuto del flag OF. Il numero naturale *n* non deve superare 31. Se l'operando sorgente non è presente, compie l'operazione una sola volta.
- **FLAG** di cui viene modificato il contenuto: Tutti.

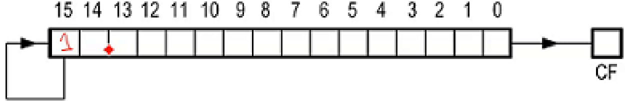
Operandi	Esempi
Immediato, Registro Generale	SHR $1,%EAX
Immediato, Memoria	SHRB $7,0x00002000
Registro CL, Registro Generale	SHR %CL,%BX
Registro CL, Memoria	SHRL %CL,(%EDI)
Memoria	SHRL (%EDI)
Registro Generale	SHR %AX

- `SHR src, %EAX`
  - Corrisponde a dividere EAX per 2^{src}, approssimando il quoziente per difetto
  - Purché EAX sia interpretato come **operando naturale** (vengono inseriti zeri in testa)

6.2.4 SHIFT ARITHMETIC RIGHT

Non è possibile applicare lo stesso algoritmo agli interi: abbiamo già detto che negli interi il MSB rappresenta il segno del numero, segue che non posso fare entrare zeri a sinistra come in SHIFT LOGICAL RIGHT

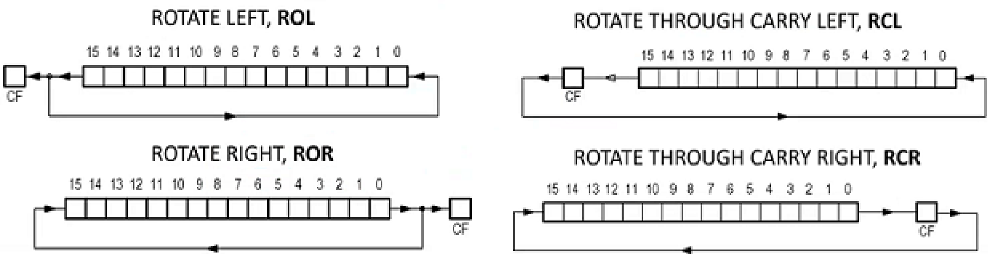
- **FORMATO:** `SAR source, destination`
- `SAR destination`
- **AZIONE:** [...].
- Guardando all'operando destinatario come ad un numero intero, questa istruzione divide tale numero per 2^n e lo sostituisce con il quoziente (approssimato per difetto).



Attenzione al quoziente SAR e IDIV restituiscono lo stesso quoziente quando il dividendo è positivo oppure quando la divisione da resto nullo. Casi diversi sono dovuti al fatto che la SAR approssima sempre a sinistra, mentre la IDIV approssima per troncamento (verso lo 0).

6.3 Istruzioni di rotazione

Le istruzioni di rotazione ruotano i bit (verso destra o verso sinistra) coinvolgendo il CF. Hanno tutte lo stesso formato (per questo segue l'introduzione della sola ROTATE LEFT). Si osserva che nelle ROTATE THROUGH CARRY sposteremo la cifra più significativa (o quella meno significativa) nel CF, ponendo come cifra meno significativa (o più significativa) il contenuto del CF presente prima della modifica.



6.3.1 ROTATE LEFT

- **FORMATO:** `ROL source, destination`
  `ROL destination`
- **AZIONE:** Interpreta l'operando sorgente come un numero naturale *n* e, per *n* volte, [...]. Il numero naturale *n* non deve superare 31. Se l'operando sorgente non è presente, compie l'operazione una sola volta.
- **FLAG** di cui viene modificato il contenuto: CF, OF.

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

6.4 Istruzioni logiche

Le istruzioni logiche permettono l'applicazione degli operatori dell'*Algebra di Boole*. Bisogna tenere conto di eventuali flag modificati (generalmente vengono modificati). Le seguenti operazioni applicano BIT a BIT le operazioni appena dette:

6.4.1 NOT

- **FORMATO**: `NOT destination`
- **AZIONE**: Modifica l'operando destinatario applicando a ciascuno dei suoi bit l'operazione logica *not*.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

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

6.4.2 AND

- **FORMATO**: `AND source, destination`
- **AZIONE**: Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica *and* tra il bit stesso ed il corrispondente bit dell'operando sorgente; mette a 0 il contenuto dei flag CF ed OF.
- **FLAG** di cui viene modificato il contenuto: Tutti.

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

6.4.3 OR

- **FORMATO**: `OR source, destination`
- **AZIONE**: Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica *or* tra il bit stesso ed il corrispondente bit dell'operando sorgente; mette a 0 il contenuto dei flag CF ed OF.
- **FLAG** di cui viene modificato il contenuto: Tutti.

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

6.4.4 XOR (OR esclusivo)

- **FORMATO:** `XOR source, destination`
- **AZIONE:** Sostituisce ciascun bit dell'operando destinatario con il risultato dell'operazione logica *xor* tra il bit stesso ed il corrispondente bit dell'operando sorgente; mette a 0 il contenuto dei flag CF ed OF.
- **FLAG** di cui viene modificato il contenuto: Tutti.

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

6.4.5 Utilizzi di questi operatori

- Singoli bit di un operando possono essere resettati usando l'operatore AND (usando una maschera formata da soli 1 tranne nel bit che si vuole resettare).
- Singoli bit di un operando possono essere settati usando l'operatore OR (usando una maschera formata da soli 0 tranne nel bit che si vuole settare)
- Singoli bit di un operando possono essere invertiti rispetto all'attuale valore usando l'operatore XOR (usando una maschera formata da soli 0 tranne nel bit che si vuole invertire). Ricordiamo che

$$\alpha \text{ XOR } 0 = \alpha \quad \alpha \text{ XOR } 1 = \overline{\alpha}$$

- **Applicazioni pratiche**:
  - L'operatore AND può essere usato per estendere operandi naturali. Supponiamo di voler sommare due numeri naturali: uno in AL e un altro in EBX.

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

  - XOR può essere usato per resettare un registro: uso la XOR (ponendo EAX sia come source che come dest) invece della MOV.

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

    Nel confronto bit a bit ho sempre due bit uguali, quindi ottengo 0 ogni volta! Ulteriore nota: la MOV, come istruzione, occupa maggiore spazio.

6.5 Istruzioni di controllo

Il flusso del programma normalmente è sequenziale: le istruzioni stanno in memoria consecutivamente e il processore, normalmente, incrementa EIP puntano l'area di memoria immediatamente successiva. Questa regola non vale più quando intervengono istruzioni di controllo: queste scrivono un nuovo valore in EIP alterando il normale flusso. Abbiamo due tipi di istruzioni:

- Istruzioni di salto (Jmp, Jcon). Nel salto non si memorizzano informazioni relative al punto da cui si è compiuto il salto (neanche implicitamente).
- Istruzioni per la gestione di sottoprogrammi (CALL, RET). Si memorizzano informazioni relative al salto (attraverso la pila)

6.5.1 JUMP

- **FORMATO:** `JMP %EIP ± displacement`
  `JMP *extended_register`
  `JMP *memory`
- **AZIONE:** Calcola un indirizzo di salto e lo immette nel registro EIP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

- In Assembler si usa **un nome simbolico** per indicare l'istruzione dove si deve saltare, quindi non è necessario aver presente il formato di indirizzamento (ci pensa l'assemblatore a tradurre in uno dei formati di sopra, in genere il primo)

6.5.2 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, allora si comporta come l'istruzione `JMP %EIP ± displacement`; in caso contrario termina senza compiere alcuna azione.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

Tabella completa con le condizioni a pagina 22 del libro di Corsini.

Cose di cui tenere conto

- Le JUMP condizionate si basano sui valori di certi flag, quindi vengono eseguite a seguito di CMP o operazioni aritmetiche.
- Il soggetto è sempre l'operando destinatario: se io dico JA significa che voglio verificare se l'operando destinatario è maggior dell'operando sorgente (naturali).
- Si distinguono:

- JUMP condizionate utilizzabili in qualunque caso (CMP, operazioni aritmetiche...);
- JUMP condizionate utilizzabili solo a seguito di operazioni CMP, precisamente...
  * JUMP per confronti tra numeri naturali (*Above* e *Below*), e
  * JUMP per confronti tra numeri interi (*Greater* e *Less*).

6.5.3 Sottoprogrammi

Le istruzioni coinvolte sono due:

- CALL, salto ad un sottoprogramma
- RET, ritorno al programma chiamante

entrambe fanno riferimento, come già detto, alla pila.

6.5.3.1 CALL

- **FORMATO**: `CALL %EIP ± $displacement`
  `CALL *extended_register`
  `CALL *`*`memory`*
- **AZIONE**: Effettua la *chiamata* di un *sottoprogramma*. Più precisamente, salva nella pila corrente il contenuto del registro EIP e poi modifica il contenuto di tale registro in modo del tutto simile a come fa l'istruzione JMP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

6.5.3.2 RET

- **FORMATO**: `RET`
- **AZIONE**: Effettua il *ritorno* da un sottoprogramma. Più precisamente, rimuove un long dalla pila e con esso rinnova il contenuto di EIP.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

6.5.3.3 NO OPERATION

- **FORMATO**: `NOP`
- **AZIONE**: Termina senza compiere alcuna azione.
- **FLAG** di cui viene modificato il contenuto: Nessuno.
- Serve a perdere tempo

6.5.3.4 HALT

- **FORMATO:** `HLT`
- **AZIONE:** Provoca la cessazione di ogni attività del processore.

Il processore alterna la fase di fetch (in cui preleva un'istruzione) con quella di esecuzione dell'istruzione che ha prelevato. Quando preleva un'istruzione HLT, smette di fare qualunque cosa, e si pone in una condizione dalla quale si può uscire solo resettando la macchina.

6.6 Protezione ed istruzioni privilegiate

- Il processore può funzionare in due modalità: utente e sistema.
- La modalità sistema permette l'esecuzione di tutte le istruzioni, la modalità utente permette l'esecuzione di una sola parte di queste istruzioni.
- Delle operazioni viste le seguenti sono istruzioni privilegiate non eseguibili in modalità utente: HLT, IN, OUT. Se chiamate va in esecuzione un'eccezione di protezione (comportamento diverso da sistema a sistema).

Relativamente all'I/0 Faremo IN e OUT attraverso sottoprogrammi di servizio (li vedremo più avanti). IN e OUT sono istruzioni privilegiate poichè è facile portarle in stato inconsistente.

6.7 Flag controllati dalle istruzioni condizionate

Si tenga conto che il soggetto delle condizioni è sempre l'operando destinatario.

- **Equal**: ZF = 1
- **Not equal**: ZF = 0
- Naturali:
  - **Above**: CF = 0, ZF = 0
  - **Above or equal**: CF = 0
  - **Below**: CF = 1
  - **Below or equal**: CF = 1 o ZF = 1
- Interi:
  - **Greater**: ZF = 0, SF = OF
  - **Greater or equal**: SF = OF
  - **Less**: SF $\neq$ OF
  - **Less or equal**: SF $\neq$ OF o ZF = 1
- **Zero**: ZF = 1. Il risultato dell'operazione precedente è stato uguale a zero.
- **Not zero**: ZF = 0. Il risultato dell'operazione precedente è stato diverso a zero.
- **Carry**: CF = 1
- **No Carry**: CF = 0
- **Overflow**: OF = 1
- **No Overflow**: OF = 0
- **Sign**: ZF = 1
- **No Sign**: ZF = 0

Perchè si guarda l'overflow per le condizioni riguardo gli interi? Bisogna considerare sia il caso senza overflow che quello senza. Si consideri che il segno del risultato della sottrazione sarà sbagliato in caso di overflow, quindi dobbiamo verificare che il suo valore sia l'opposto di quello previsto.

- in **Greater or equal**: mi aspetto una differenza con segno positivo. Se non ho l'overflow avrò SF = OF = 0. Se ho l'overflow avrò OF = 1. Se il valore del segno deve essere l'opposto di quello previsto dovrò avere SF = 1, quindi SF = OF = 1.
- in **Less or equal**: mi aspetto una differenza con segno negativo. Se non ho l'overflow avrò SF = 1 e OF = 0. Se ho l'overflow avrò OF = 1. Se il valore del segno deve essere l'opposto di quello previsto dovrò avere SF = 0, quindi SF = 0, OF = 1.

Capitolo 7

Martedì 06/10/2020

7.1 Assemblatore

Utilizzeremo l'assemblatore **GAS** (*GNU Assembler*): le informazioni per scaricarlo sono presenti sul sito del docente. Si raccomanda massimo rispetto della procedura di installazione.

7.2 Struttura di un programma assembler

Un programma Assembler si articolo in due sezioni:

- Sezione dati. Contiene le dichiarazioni delle variabili, cioè nomi simbolici per indirizzi di memoria.
- Sezione codice. Istruzioni del programma

Direttive e istruzioni In un programma sono presenti **direttive** (per esempio le dichiarazioni di variabili o di costanti) e **istruzioni** (ciò di cui abbiamo parlato fin dalla prima lezione). Ciascuna istruzione e direttiva è contenuta in una riga: alla fine è sempre presente il ritorno carrello *CR* (anche nell'ultima riga di tutto il codice). In un codice ASS individueremo, in conclusione:

- Una prima parte contenente le direttive

  ```
  .GLOBAL _main

  .DATA
  ....
  ```

  la prima istruzione ci permette di stabilire quale sottoprogramma eseguire quando viene avviato l'applicativo. Nell'area .DATA indicheremo, come già detto, le variabili da utilizzare.

- Una parte contenente le istruzioni

  ```
  .TEXT
  _main : NOP
  ```

```
...
RET
```

la RET deve essere posta in fondo ad ogni sottoprogramma, incluso quello principale. Relativamente alla NOP capiremo più avanti l'utilità nel porla all'inizio del sottoprogramma.

Ordine degli elementi L'ordine degli elementi descritto non è l'unico possibile: quanto detto è consigliato per ottenere un programma leggibile.

7.2.1 Assembler case-sensitive o case-insensitive?

Assembler è:

- **case-insensitive** relativamente ai caratteri delle keyword. Si consiglia di porle in stampatello maiuscolo
- **case-sensitive** relativamente alle etichette (nomi di variabili e nomi di sottoprogrammi). Significa che

$$_main \neq _Main$$

7.2.2 Cosa cambia rispetto agli esempi di programmi visti in passato?

```
# conteggio bit a 1 in un long

.GLOBAL _main                          0x00000100
.DATA                                  0x00000101
dato:       .LONG 0x0F0F0101           0x00000102
conteggio:  .BYTE 0x00                 0x00000103
                                       0x00000104
.TEXT
_main:      NOP                        ...
            MOVB $0x00,%CL             0x00000200  MOVB $0x00, %CL
            MOVL dato, %EAX            0x00000202  MOVL 0x00000100, %EAX
comp:       CMPL $0x00,%EAX            0x00000207  CMPL $0x0000000, %EAX
            JE fine                    0x0000020A  JE %EIP+$0x07
            SHRL %EAX                  0x0000020C  SHRL %EAX
            ADCB $0x00, %CL            0x0000020E  ADCB $0x00, %CL
            JMP comp                   0x00000211  JMP %EIP-$0x0C
fine:       MOVB %CL, conteggio        0x00000213  MOVB %CL, 0x00000104
            RET                        0x00000218  ...
```

Il codice risulta decisamente più ordinato ma soprattutto andrò a richiamare aree di memoria usando nomi simbolici (e non più solo displacement). Queste etichette, che poniamo nella parte DATA, rappresentano simbolicamente l'indirizzo di quell'area di memoria. La cosa ci semplifica la vita non poco: le cose viste nelle lezioni precedenti saranno gestione dell'assemblatore.

7.2.2.1 Attenzione al jump : confronto col C++

Si individua nel codice richiami ad etichette dichiarate più avanti: questa cosa è impensabile in C++ ma possibile in Assembler. Questo perchè l'assemblatore

1. Controlla tutti i nomi
2. traduce

l'approccio è necessario per poter svolgere salti in avanti, ma può essere anche mortale. Assembler, a differenze del C++, offre maggiori spazi di libertà: dobbiamo usare questi spazi con estrema cautela. Il suggerimento, in generale, è di continuare a rispettare la prassi del C++ (unica eccezione i salti in avanti).

Altro esempio di porcheria La libertà fornita da Assembler potrebbe portarci a scrivere quanto segue

```
nome1:
nome2: CMP
...
...
JMP nome1
JMP nome2
```

queste cose, pur essendo valide (di fatto abbiamo creato due alias per un sottoprogramma), sono porcherie da evitare.

7.2.3 Riga di codice

Le righe di codice presentano la seguente struttura

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

l'unico elemento obbligatorio in una riga è il carattere di ritorno carrello.

7.3 Direttive

Attenzione Ricordarselo come l'ave maria

$$\boxed{\text{Direttive} \neq \text{Istruzioni}}$$

Le direttive presentano la seguente struttura

```
.KEYWORD operandi [\CR]
```

7.3.1 Dichiarazione di variabili

Abbiamo già detto che dichiareremo variabili all'inizio del nostro programma. La cosa è utile non tanto per dire ad assembler su cosa stiamo lavorando ma per stabilire in modo agile quanta memoria ci serve. I tipi di variabili possibili sono:

- .WORD (2byte, 16bit)
- .BYTE (1byte, 8bit)
- .LONG (4byte, 32bit)

```
.WORD ________        # scalare, 2 byte, valore 0x0000
.BYTE 0x30            # scalare, 1 byte, valore 0x30
.BYTE 0x30, 0x31      # vett., 2 componenti da 1 byte
.WORD 0x1020, 0x32AB# vett., 2 componenti da 2 byte
.LONG var3+2          # scalare, 4 byte
```

Individuiamo che

- è ammesso dichiarare variabili senza inizializzarle. Formalmente il valore inizializzato dovrebbe essere quello indicato in var0, ma la cosa non è certa e il comportamento della macchina imprevedibile. Si consiglia di inizializzare in qualunque caso
- possiamo dichiarare variabili scalari, ma anche variabili vettoriali (attraverso una lista)
- Possiamo fare riferimento ad elementi di un vettore attraverso la seguente sintassi

  ```
  nome_variabile_vettoriale+K
  ```

  dove K consiste nel numero di byte. Se abbiamo un vettore di tipo WORD porremo $K = 2$ per passare dal primo al secondo elemento.

7.3.1.1 Alternativa per dichiarare vettori (comando FILL)

Assembler offre la direttiva FILL per dichiarare vettori. Abbiamo la seguente sintassi

```
.FILL numero, dim, espressione
```

- **numero** si utilizza per indicare il numero di elementi del vettore
- **dim** si utilizza per indicare il tipo di vettore. Può assumere solo i valori $1, 2, 4$ (per indicare, rispettivamente, byte, word e long)
- **espression**e si utilizza per indicare il valore con cui inizializzare ogni elemento del vettore. Se omesso è uguale a zero (in questo caso possiamo fidarci senza problemi). Segue che FILL è utile quando non vogliamo inizializzare una variabile.

Abbiamo utilizzato questa cosa per salvare stringhe da porre nel buffer: avremo come numero 80, come dim 1 (codifica ASCII, con 7bit posso codificare tutte le lettere necessarie per una stringa) e come espressione un valore iniziale. Per porre una stringa all'interno di questa area di memoria si utilizza la LEA e la funzione inline!

7.3.1.2 Codifica ASCII e caratteri speciali

Anche in Assembler possiamo immaginare il salvataggio di stringhe come array, precisamente un array di byte. Possiamo porre, tra singoli apici, i caratteri, o direttamente la codifica ASCII

```
var5: .BYTE ‘C’,’i’,’a’,’o’
var5: .BYTE 0x43, 0x69, 0x61, 0x6F
```

Possibile inizializzare vettori di BYTE anche attraverso l'istruzione ASCII e ASCIZ

```
var6: .ASCII ‘‘messaggio’’
var7: .ASCIZ ‘‘messaggio’’
```

con la prima istruzione creiamo un vettore composto da 9 elementi (ciascuno 1byte), mentre con la seconda creiamo un vettore composto da 9 elementi più uno aggiuntivo (backslash 0, NUL).

Caratteri speciali Caratteri speciali come ritorno carrello, tabulazione, etc... possono essere indicati attraverso le stesse sequenze di escape viste in C++.

7.3.2 INCLUDE

La direttiva include permette di includere file sorgente:

```
.INCLUDE ‘‘./path’’
```

essa può essere posta in cima o in fondo (si consiglia in cima). Il percorso del file sorgente incluso, inoltre, deve essere per forza incluso tra doppi apici.

7.3.3 SET

Con la direttiva SET possiamo creare costanti simboliche, richiamabili all’interno di istruzioni (ovviamente precedute dal dollaro)

```
.SET nome, espressione
```

- **nome**, cioè l’identificativo della costante
- **espressione**, cioè il contenuto della costante

Esempio

```
.SET dimensione, 4
.SET n_iter, (100*dimensione)
...
MOV $n_iter, %CX #op. immediato, ci vuole ‘$’
```

è il discorso di *pippo* fatto all’inizio.

7.3.3.1 Calcolare memoria occupata

Il seguente codice permette di calcolare il numero di byte occupati in memoria

```
dato1: .FILL 1024, 4
dato2: .FILL 100, 2
...
datoN : .FILL 350, 2
foo: .BYTE 1
.SET occupazione, (foo-dato1)
...
MOV $occupazione, %ECX
```

dichiaro una variabile foo che non serve a niente, ma permette di fare la differenza tra indirizzi. Questo mi permette di individuare il numero di byte occupati!

7.3.3.2 Costanti numeriche

Differenza tra naturali e interi

- I naturali non sono preceduti da segno, vengono convertiti nella loro rappresentazione in base 2.
- Gli interi sono preceduti da un segno (positivo o negativo), e vengono convertiti nella loro rappresentazione in complemento a bit sul numero di bit opportuno.

La presenza del segno permette all'assemblatore di capire se stiamo parlando di naturali o interi.

Basi numeriche I numeri possono essere scritti in base $2, 8, 10, 16$. Abbiamo già visto che si identifica la base delle costanti numeriche mediante prefissi:

- I numeri in base 2 iniziano con $0b$
- I numeri in base 8 iniziano con 0
- I numeri in base 10 non iniziano per 0
- I numeri in base 16 iniziano per $0x$

Attenzione all'intervallo di rappresentazione Supponiamo di voler porre 128 come contenuto del registro AL

```
MOV $128, %AL
MOV $+128, %AL
```

scrivere la seconda istruzione invece della prima potrebbe dare problemi. Ricordiamo che gli estremi dell'intervallo di rappresentazione degli interi sono asimmetrici $[-128, +127]$. Il numero 128, in 8bit, è rappresentabile come naturale ma non come intero!

Come si comporta l'assemblatore? Quando le rappresentazioni non sono della dimensione giusta vengono

- troncate se troppo lunghe (solitamente l'assemblatore avverte)
- estese se di dimensione minore (solitamente l'assemblatore non ce lo dice)

7.4 Controllo di flusso

Abbiamo già detto che in assembler non sono presenti i costrutti legati al controllo di flusso, precisamente

- if...then...else
- for...
- while...
- do...while

tuttavia con le istruzioni di salto viste nelle scorse lezioni e le notazioni simboliche introdotte oggi possiamo ottenere la stessa azione di quei costrutti. L'approccio adottato, cioè imparare prima il C++ con questi costrutti e poi l'assembler, aiuta moltissimo.

7.4.1 if... then ... else

```
if (%AX<variabile)       // naturali
    {ist1; …; istN;}
else
    {istN+1; …; istN+M;}
ist_nuova;
```

• Invertire ramo **then** e ramo **else**

```
            CMP variabile, %AX
            JB ramothen
ramoelse:   istN+1
            …
            istN+M
            JMP segue
ramothen:   ist1
            …
            istN
segue:      ist_nuova
```

• Invertire **la condizione**

```
CMP variabile, %AX
      JAE ramoelse
ramothen:   ist1
            …
            istN
            JMP segue
ramoelse:   istN+1
            …
            istN+M
segue:      ist_nuova
```

Se vogliamo lasciare inalterata la condizione nel codice Assembler dobbiamo invertire di ordine le righe relative al *then-statement* con le righe relative all'*else-statement*. Se non vogliamo fare ciò dobbiamo alterare la condizione ponendo l'esatto opposto rispetto a quella iniziale. Supponiamo di non fare ciò:

- inizializzo le variabili necessarie;
- eseguo un confronto, comunque sia l'istruzione che altera i flag;
- utilizzo una jump condizionata per andare alle righe del *then-statement* nel caso in cui la condizione sia soddisfatta (salto le righe dell'*else-statement*);
- se la condizione non è soddisfatta non salto ed eseguo le righe dell'*else-statement*, incluso il jump non condizionato finale che mi permette di saltare le righe del *then-statement* e proseguire nel codice.

7.4.2 for...

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

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

Teniamo conto delle fasi di un ciclo for:

- inizializzo le variabili necessarie prima di entrare nel ciclo;

- verifico la condizione che determina l'uscita dal ciclo (l'opposto della condizione che scriviamo nel C++), se questa è verificata salto le istruzioni del corpo del for;
- eseguo le istruzioni del corpo del for, inclusa quella finale di jump non condizionato;
- la jump non condizionata mi riporta al punto di verifica della condizione di uscita dal for.

7.4.3 do. . . while

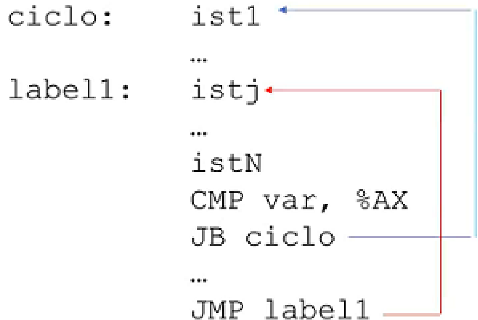
```
do
   {ist1; ...; istN;}
while (AX<var);
```

```
ciclo:    ist1
          …
          istN
          CMP var, %AX
          JB ciclo
```

Ricordiamoci che nel do-while il corpo viene eseguito almeno una volta. Quindi:

- inizializzo le variabili necessarie;
- eseguo le istruzioni del corpo del while;
- verifico che la condizione di permanenza nel ciclo sia rispettata, se lo è salto ritornando alla prima istruzione del corpo del while.

7.4.4 *Spaghetti-like* coding



In Assembler, contrariamente a C++, C e Pascal, è possibile saltare nel mezzo di un ciclo. La cosa può causare non pochi problemi: si parla di *spaghetti-like* coding poichè il programma, se simile a un piatto di spaghetti, può avere risultati imprevedibili. I linguaggi strutturati (quelli citati poco fa) sono stati pensati apposta per evitare questo stile di programmazione

Unico punto di ingresso, unico punto di uscita

Nel C++ esiste il costrutto *goto* (rammentato sul libro di Domenici e Frosini, ma mai affrontato a FdP): quel costrutto, dice Stea, serve solo per far danni e non conviene usarlo assolutamente!

7.4.5 LOOP

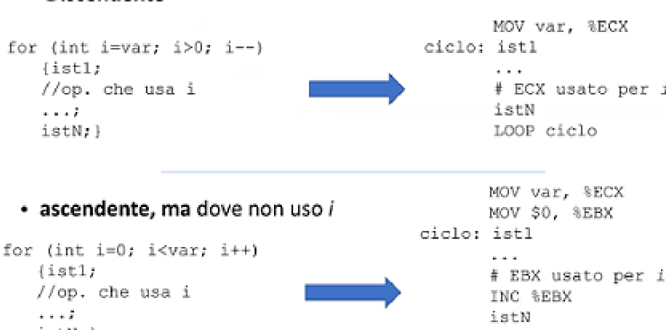
Il comando LOOP decrementa ogni volta il contenuto di ECX, e salta finchè il valore di ECX non sarà uguale a 0 (chiaramente ECX non deve essere toccato nel ciclo).

```
        MOV $5, %ECX
ciclo:  ist1
        ...
        istN;
        LOOP ciclo
```

La cosa è utilissima per riprodurre le stesse azioni di un for.

Attenzione La LOOP non modifica alcun flag nel decrementare ECX: questo per permettere istruzioni di tipo CMP all'interno del ciclo (essenziale nei LOOPcond).

7.4.5.1 For discendente e for ascendente con LOOP



7.4.6 LOOP condizionato

Le LOOP condizionali, precisamente le istruzioni LOOPE e LOOPNE, permettono di determinare l'uscita da un LOOP attraverso una istruzione CMP, posta all'interno del ciclo.

- Poniamo in ECX il numero massimo di iterazioni. Il ciclo si conclude col raggiungimento del numero massimo di iterazioni, o se la condizione della CMP non risulta più vera. **Condizioni presenti**: *Loop if equal, loop if zero, loop if not equal, loop if not zero.*

  - LOOPE, LOOPNE (LOOPZ, LOOPNZ)
  - Sensate soltanto **dopo una CMP**
  - Il salto avviene se:
    1. La condizione e' vera
    2. Dopo il decremento, ECX!=0
  - Si scrive in ECX il numero **massimo** di iterazioni
    - Il ciclo puo' terminare prima, se la condizione diventa falsa

```
        MOV $10, %ECX
ciclo:  ist1
        ...
        istN
        CMP <src>,<dest>
        LOOPcond ciclo
```

7.4.7 Utilità dei LOOP e manipolazione stringhe

I LOOP sono comodi ma non indispensabili. Possono essere posti in modo alternativo, per esempio attraverso una istruzione compare e una jump (con condizione). Andando avanti vedremo istruzioni per la manipolazione delle stringhe: sono presenti prefissi di ripetizione che permettono l'implementazione di cicli, quindi accessi sequenziali in memoria.

7.4.8 Attenzione alla lunghezza dell'iterazione

La LOOP prevede salti relativi del tipo

`EIP + - displacement`

sappiamo che il displacement è a 8 bit, segue che non possiamo fare salti troppo ampi. La LOOP è ottima in caso di cicli brevi: se necessitiamo di cicli più lunghi basta utilizzare le classiche istruzioni introdotte prima della LOOP.

7.5 Sottoprogrammi e passaggio dei parametri

Abbiamo già detto che le istruzioni per i sottoprogrammi sono due: la CALL e la RET. Per la prima abbiamo un unico operando (l'indirizzo del sottoprogramma), mentre la RET non prevede operandi. Per passare parametri ai sottoprogrammi (e viceversa) ci si affida a delle convenzioni:

- si utilizzano locazioni di memoria condivise tra il programma chiamante e il programma chiamato (il programma chiamante mette da qualche parte certi valori, il programma chiamato lo sa e li va a pescare lì)
- si utilizzano i registri (il programma chiamante pone qualcosa in un registro, il programma chiamato lo sa e va a leggere lì; la cosa vale anche al contrario)
- usare la pila (non affronteremo questo metodo, è un casino e ce ne occuperemo a Calcolatori elettronici - questo metodo è utilizzato proprio dai compilatori).

Osservazione In Assembler non esiste il concetto di variabile locale: la memoria principale è indirizzabile da qualunque sottoprogramma.

Regola Convenzione buona e giusta è specificare i parametri di ingresso e di uscita del sottoprogramma attraverso commenti.

```
              MOV …, %AX          # preparazione dei parametri
              MOV …, %EBX         # per la chiamata di sottoprogramma
              CALL sottoprg
              MOV %CX, var        # utilizzo del valore di ritorno

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

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

ogni parametro deve essere accompagnato da descrizione (contenuto, scopo...)

Capitolo 8

Giovedì 08/10/2020

8.1 Uso dei registri ed effetti collaterali

I registri che non contengono valori di ritorno non devono essere sporcati da un sottoprogramma. Precisamente:

- I registri usati da un sottoprogramma devono essere salvati in pila
- Attenzione ai registri toccati, non è detto che si vedano nel codice (dobbiamo controllare la documentazione)
- La pila deve essere tenuta in ordine: per ogni push dovrà esserci una POP. La cosa avviene nel seguente modo

  ```
  sottoprog: PUSH x1
  PUSH x2
  PUSH x3
  ...
  POP x3
  POP x2
  POP x1
  RET
  ```

  Se la RET, quando pesca l'indirizzo di ritorno dalla pila, trova valori casuali il programma si inchioda.

8.2 Sottoprogramma principale

Il _main è il sottoprogramma principale.

- Deve terminare con una RET.
- Solitamente ci si aspetta un valore nel registro EAX:
  - 0 se tutto è ok
  - un valore $\neq 0$ se ci sono errori (e il valore consiste nel codice errore)

  quanto detto consiste in una convenzione che non è obbligatoria all'esame.

Prassi usata spesso Se vogliamo rispettare la condizione ci basta mettere la seguente istruzione subito prima della RET

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

8.3 Dichiarazione e allocazione di spazio per la pila

Abbiamo lasciato in sospeso un discorso relativo alla pila: essa esiste perchè qualcuno la dichiara e la inizializza (allocando spazio). Dobbiamo:

- dichiarare lo stack
- inizializzare il registro ESP affinchè esso punti alla cella successiva al fondo dello stack (non in fondo, ma alla prima locazione successiva all'area riservata alla pila).

La dichiarazione avviene nella parte DATA

```
.DATA
mystack: .FILL 1024, 4
.SET initial_esp, (mystack + 1024* 4)

.TEXT
_main: NOP
MOV $initial_esp, %ESP
```

La dimensione dello stack è problema del programmatore (sappiamo noi cosa andiamo a fare e quindi quanto spazio ci serve).

Tuttavia Nel nostro ambiente, ma non in Assembler in generale, possiamo omettere la dichiarazione dello stack. Ci pensa l'ambiente secondo regole sue.

8.4 Ingresso/uscita e sottoprogrammi di utilità

Ricordiamo Abbiamo già detto che le istruzioni IN e OUT sono istruzioni privilegiate, normalmente non utilizzabili. L'assemblatore assembla ma al momento dell'esecuzione viene lanciata un'eccezione di protezione.

Come facciamo? Ricorriamo a sottoprogrammi offerti dal sistema DOS.

- DOS offre sottoprogrammi che girano in modalità sistema e che possono usare le IN/OUT
- Tuttavia questi sottoprogrammi sono molto primitivi: si ha l'ingresso o l'uscita di un solo carattere.
- A partire da questi sottoprogrammi i docenti Stea e Corsini hanno costruito altri sottoprogrammi che permettono ingresso e uscita a più alto livello (non troppo). Si specifica che
  - Gli ingressi avvengono da tastiera
  - Le uscite avvengono su video, precisamente su una finestra DOS di 80x25 caratteri.

8.5 Osservazioni sulla tabella ASCII

Prendiamo una tabella riepilogativa della codifica ASCII (che ricordiamo, da FdP, essere su 7bit). Ogni combinazione consiste in un particolare carattere stampabile. Osserviamo che:

- I caratteri numerici sono consecutivi, stessa cosa i caratteri maiuscoli e quelli minuscoli (nella tabella individuiamo prima le maiuscole delle minuscole).
- Le sequenze binarie associate ai caratteri numerici presentano la seguente struttura

  `0011XXXX`

  La parte rimanente della sequenza consiste nella rappresentazione binaria del corrispondente numero naturale. Questo ci permette di capire, in modo agile, se una certa sequenza rappresenta un certo carattere numerico[1].
- Le sequenze binarie associate alle lettere presentano la seguente struttura

  `01XYYYYY`

  dove X può assumere come valore 0 o 1. Precisamente, se si ha 0 il carattere è maiuscolo, se si ha 1 il carattere è minuscolo. Segue che dato un carattere maiuscolo (o viceversa) possiamo ottenere il corrispondente carattere minuscolo (o viceversa) attraverso la modifica del bit X (con una maschera).
- Ci interessano tre caratteri speciali in particolari
  - il backspace. Si ritorna indietro di una posizione nella riga
  - il line feed, o avanzamento di riga. Si scorre di uno le righe
  - il carrage return, o ritorno carrello. Si riporta il cursore all'inizio di una riga.

[1] Se dobbiamo convertire la codifica ASCII di un numero nel numero corrispondente ci basta utilizzare l'istruzione AND e una maschera che si sbarazza dei quattro bit più significativi.

8.5.1 I/0 da tastiera e video

Si hanno, sia in ingresso che in uscita, codifiche ASCII di singoli caratteri. Non abbiamo l'I/O tipica del C++. Prendiamo ad esempio il numero 32:

- stampo il primo carattere ASCII 0x33 (3) e poi il secondo, 0x32 (2)

Esempio Supponiamo di voler fare ingresso da tastiera di un numero naturale a 2 cifre, $\beta = 10$

- Memorizzo due codifiche ASCII: c_1, c_0
- Calcolo le singole cifre decimali: a_1, a_0 (a partire dalle codifiche c_1, c_0)
- Ricostruisco il numero digitato: $10 \cdot a_1 + a_0$ (tengo conto della posizione delle cifre)

8.5.2 File utility

```
.INCLUDE ‘‘C:/amb_GAS/utility’’
```

attraverso la direttiva andiamo a includere un pack di sottoprogrammi creato da Stea e Corsini.

8.5.2.1 Sottoprogrammi di I/0 (su cui si basano i sottoprogrammi della sezione successiva)

- **inchar**: metto nel registro AL la codifica ASCII del tasto premuto (non stampo)
- **outchar**: metto sul video la codifica ASCII contenuta in AL
- **newline**: per andare a capo (stampo due caratteri: ritorno carrello e line feed)
- **pauseN**: metto in pausa il programma stampando il seguente messaggio (N cifra decimale)

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

```
demo1.s
1   .GLOBAL _main
2
3   .TEXT
4   _main:  CALL inchar
5           # ...
6           CALL outchar
7
8           CALL newline
9
10          CALL outchar
11          CALL outchar
12
13          CALL pause1
14          CALL pause2
15
16          CALL newline
17
18          CALL outchar
19
20          XOR %EAX, %EAX
21
22          RET
23
24  .INCLUDE "C:/amb_GAS/utility"
25
```

```
DOSBox 0.73, Cpu Cycles: max, Frameskip 0, Program: GDB
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public Lice
welcome to change it and/or distribute copies of it under ce
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warrant
This GDB was configured as "--host=i386-pc-msdosdjgpp --targ
Breakpoint 1 at 0x1e60: file DEMO1.S, line 4.

Breakpoint 1, main () at DEMO1.S:4
warning: Source file is more recent than executable.

4       _main:  CALL inchar
Current language:  auto; currently asm
(gdb) CONTINUE
Continuing.
A
AA
Checkpoint number 1. Press any key to continue

Checkpoint number 2. Press any key to continue

A
Program exited normally.
(gdb)
```

8.5.2.2 Sottoprogrammi a livello più alto

- **inline**: consente di portare una stringa di massimo 80 caratteri in un buffer da memoria (digitando da tastiera con eco su video).
  - il registro EBX consiste nell'indirizzo di memoria del buffer
  - il registro CX contiene il numero di caratteri da leggere (massimo 80, una riga di DOS)

  La lettura da tastiera termina dopo 78 caratteri (78 + line feed + carrage return) o se premiamo invio. Cosa molto comoda è l'interpretazione del backspace come ordine di cancellare dal buffer e dal video l'ultimo carattere digitato.

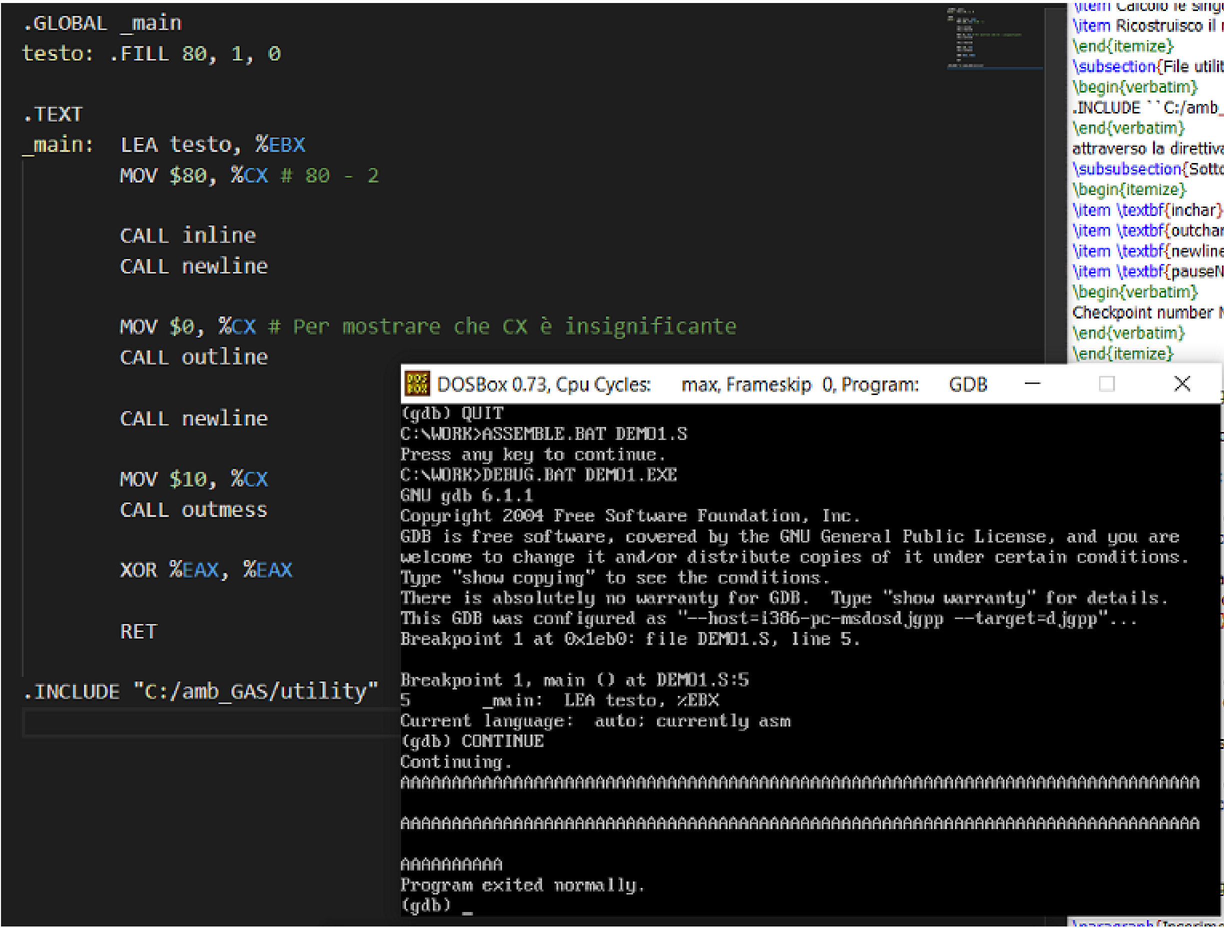
- **outline**: stampa a video massimo 80 caratteri. Si ferma prima se trova un carattere di ritorno carrello ed eventualmente stampa i caratteri per andare a capo
  - il registro EBX consiste nell'indirizzo di memoria del buffer. Se vediamo simboli esoterici significa che il registro contiene sequenze di zeri e uno che non hanno a che vedere con la codifica ASCII.

- **outmess**: stessa funzione del sottoprogramma precedente
  - il registro EBX consiste nell'indirizzo di memoria del buffer.
  - il registro CX contiene il numero di caratteri da stampare a video

```
.GLOBAL _main
testo: .FILL 80, 1, 0

.TEXT
_main:  LEA testo, %EBX
        MOV $80, %CX # 80 - 2

        CALL inline

        MOV $0, %CX # Per mostrare che CX è insignificante
        CALL outline

        MOV $10, %CX
        CALL outmess

        XOR %EAX, %EAX

        RET

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

```
DOSBox 0.73, Cpu Cycles:
Breakpoint 1 at 0x1eb0: fil

Breakpoint 1, main () at DE
5       _main:  LEA testo,
Current language:  auto; cu
(gdb) step
6               MOV $80, %C
(gdb) step
8               CALL inline
(gdb) step
inline () at utility:221
221     inline:  CMP     $2
(gdb) step
222             JB      L4
(gdb) step
223             CMP     $8
(gdb) step
224             JA      L4
(gdb) continue
Continuing.
Ciao Marco, come stai?
Ciao Marco, come stai?
Ciao Marco
Program exited normally.
(gdb)
```

Inserimento di una stringa di 80 caratteri Non appena si supera il limite il contenuto viene inviato. Attenzione: se non si inserisce una newline non si andrà a capo.



8.5.2.3 Sottoprogrammi per l'ingresso/uscita di numeri esadecimali

- **inbyte, inword, inlong**: prelevano da tastiera (facendo eco su video) 2, 4 o 8 caratteri (non fino a x caratteri). La sequenza ottenuta in ingresso viene interpretata come un numero esadecimale. Ignorano qualunque altro carattere premuto.
  - il numero esadecimale viene posto in AL, AX, EAX (in base al sottoprogramma scelto).
- **outbyte, outword, outlong**: stampano sul video, rispettivamente, 2, 4 o 8 caratteri corrispondenti a cifre esadecimale.
  - Vengono estratte interpretando il contenuto di AL, AX, EAX (in base al sottoprogramma scelto) come un numero naturale.

```
.GLOBAL _main
testo: .FILL 80, 1, 0

.TEXT
_main:  CALL inbyte
        CALL newline

        CALL outbyte
        CALL newline

        CALL outdecimal_byte

        XOR %EAX, %EAX

        RET

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

```
Program exited normally.
(gdb) QUIT
C:\WORK>ASSEMBLE.BAT DEMO1.S
Press any key to continue.
C:\WORK>DEBUG.BAT DEMO1.EXE
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, I
GDB is free software, covered by the GNU G
welcome to change it and/or distribute cop
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.
This GDB was configured as "--host=i386-pc
Breakpoint 1 at 0x1eb0: file DEMO1.S, line

Breakpoint 1, main () at DEMO1.S:5
5       _main:  CALL inbyte
Current language:  auto; currently asm
(gdb) continue
Continuing.
FF
FF
255
Program exited normally.
(gdb) _
```

8.5.2.4 Sottoprogrammi per l'ingresso/uscita di numeri decimali

- **indecimal_byte, indecimal_word, indecimal_long**: prelevano da tastiera fino a 3,5 o 10 cifre decimali. La sequenza è interpretata come un numero decimale. Tutti gli altri caratteri vengono ignorati.
  - il numero decimale viene posto in AL, AX, EAX (in base al sottoprogramma scelto).

  Se il numero è troppo grande allora viene troncato. Si ricordi che
  $2^8 - 1 = 255, 2^{16} - 1 = 65535, 2^{32} - 1 = 4294967295$.

- **outdecimal_byte, outdecimal_word, outdecimal_long**: stampano sul video il contenuto di AL, AX, EAX (in base al sottoprogramma scelto) interpretato come un numero naturale in base 10 sul numero di cifre strettamente necessario.

```
.GLOBAL _main
testo: .FILL 80, 1, 0

.TEXT
_main:  CALL indecimal_byte
        CALL newline

        CALL outdecimal_byte
        CALL newline

        CALL outbyte

        XOR %EAX, %EAX

        RET

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

```
Program exited normally.
(gdb) quit
C:\WORK>ASSEMBLE.BAT demo1.s
Press any key to continue.
C:\WORK>DEBUG.BAT demo1.exe
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, Inc
GDB is free software, covered by the GNU Ger
welcome to change it and/or distribute copie
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Ty
This GDB was configured as "--host=i386-pc-m
Breakpoint 1 at 0x1eb0: file demo1.s, line 5

Breakpoint 1, main () at demo1.s:5
5       _main:  CALL indecimal_byte
Current language:  auto; currently asm
(gdb) continue
Continuing.
255
255
FF
Program exited normally.
(gdb) _
```

8.6 Istruzioni che manipolano le stringhe

In Assembler non esistono tipi di dati nè strutture dati. Esistono soltanto byte, word e long. L'unica cosa simile a una struttura dati è quella dei vettori:

- possiamo dichiarare vettori di variabili di una certa direzione
- possiamo fare indirizzamento con displacement + registri base/indice

introduciamo le cosiddette istruzioni stringa, che permettono di copiare interi buffer di memoria (quindi interi blocchi di memoria). Queste istruzioni, oltre ad essere comodee, sono efficienti!

- il registro ESI è puntatore a sorgente
- il registro EDI è puntatore a destinazione

Copia di un vettore Supponiamo di voler copiare un vettore. Ho un ciclo con molte istruzioni

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

usiamo le istruzioni stringa

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

come vediamo la cosa è estremamente semplificata (una sola istruzione è sintomo di maggiore efficienza). REP consiste in un prefisso di ripetizione, mentre MOVSL rappresenta un'istruzione stringa.

8.6.1 Istruzioni *Direction Flag* (STD e CLD)

Abbiamo due istruzioni per gestire questo flag:

- STD (*Set direction flag*), si imposta DF = 1
- CLD (*Clear direction flag*), si imposta DF = 0

queste istruzioni dovranno essere poste prima delle istruzioni relative.

8.6.2 MOVE DATA FROM STRING TO STRING

Istruzione che permette di copiare vettori

- il suf è obbligatorio (NON ABBIAMO OPERANDI ESPLICITI) e stabilisce di quanti bit dobbiamo copiare (B, W, o L).
- Si utilizzano i puntatori di memoria ESI (puntatore sorgente) ed EDI (puntatore destinatario), che vengono incrementati o decrementati.
- L'istruzione, senza REP, copia il numero di byte specificato dal suffisso dall'indirizzo di memoria puntato da ESI all'indirizzo di memoria puntato da EDI.
- Se è incluso il prefisso REP le azioni appena dette vengono replicate per il numero di volte specificato in ECX. ECX viene decrementato finchè non avrà come valore 0.

la copia, attenzione, può essere sia in avanti che indietro: utilizzeremo un nuovo registro non ancora visto, il *Direction Flag*.

- Se DF = 0 si copia in avanti. ESI ed EDI vengono incrementati dopo aver copiato il numero di byte indicato.
- Se DF = 1 si copia all'indietro. ESI ed EDI vengono decrementi dopo aver copiato il numero di byte indicato.

Ovviamente ESI ed EDI, con DF = 1, saranno inizializzati puntando all'ultimo byte, piuttosto che al primo.

Attenzione DF non vale 0 di default.

- **FORMATO:** `MOVSsuf` `REP MOVSsuf`
- **AZIONE:** Copia il numero di byte specificato dal suffisso *suf* dall'indirizzo di memoria puntato da ESI all'indirizzo di memoria puntato da EDI. Se DF=0, somma ad ESI e ad EDI il numero di byte specificato dal suffisso. Se DF=1, sottrae da ESI e da EDI il numero di byte specificato dal suffisso.
- Se viene premesso il prefisso REP, allora le azioni indicate sopra vengono ripetute per il numero di volte specificato in ECX, che viene decrementato fino a zero.
- **FLAG** di cui viene modificato il contenuto: Nessuno.

8.6.3 LODSsuf (*load string*) e STOSsuf (*store string*)

- **LODSsuf**: Copia in AL, AX, oppure EAX (a seconda del suffisso) il contenuto della memoria all'indirizzo puntato da ESI. A seconda del valore del flag DF, incrementa o decrementa di 1, 2, 4 ESI.
- **STOSsuf**: Copia il registro AL, AX, oppure EAX (a seconda del suffisso) in memoria all'indirizzo puntato da EDI. A seconda del valore del flag DF, incrementa o decrementa di 1, 2, 4 EDI.

Osservazione Non ha senso utilizzare REP con LODS: se io sposto il contenuto nei registri significa che voglio utilizzarlo per fare qualcosa. Letture ripetute sono consecutive.

- Copiare un vettore da una parte ad un'altra della memoria, eseguendo la stessa operazione su tutti i suoi elementi

```
       MOV $1000, %CX
       LEA buffer_src, %ESI
       LEA buffer_dst, %EDI
       CLD
ciclo: LODSL          +4 %ESI
       # modifica %EAX
       STOSL          +4 %EDI
       LOOP ciclo
```

- Riempire un buffer di memoria con lo stesso valore (e.g., zero)

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

EAX → (%EDI)

```
.GLOBAL _main

.DATA
.SET num_elementi, 10
array_numeri: .FILL num_elementi, 1, 0

.TEXT
_main:  NOP
        MOV $num_elementi, %ECX
        LEA array_numeri, %ESI
ciclo1: CALL indecimal_byte
        CALL newline
        MOV %AL, (%ESI)
        INC %ESI
        LOOP ciclo1
poi:    LEA array_numeri, %ESI
        MOV $num_elementi, %ECX
        CLD
        CALL newline
        CALL newline
ciclo2: LODSB
        CALL outdecimal_byte
        CALL newline
        LOOP ciclo2
fine:   XOR %EAX, %EAX
        RET

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

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

.DATA
vettore: .BYTE 1, 2, 3, 4, 5, 6, 7, 8, 9
variabile_stupida: .BYTE 0

.TEXT
_main:          NOP
                LEA variabile_stupida, %ESI
                DEC %ESI
                MOV $9, %CL
                STD
ciclo:          LODSB
                CALL outdecimal_byte
                CALL newline
                DEC %CL
                JNZ ciclo
fine:           XOR %EAX, %EAX
                RET
```

```
DOSBox 0.73, Cpu Cycles:
Copyright 2004 Free Software
GDB is free software, covere
welcome to change it and/or
Type "show copying" to see t
There is absolutely no warra
This GDB was configured as "
Breakpoint 1 at 0x21cc: file

Breakpoint 1, main () at es2
9         _main:          NOP
Current language:  auto; cur
(gdb) continue
Continuing.
9
8
7
6
5
4
3
2
1

Program exited normally.
(gdb) _
```

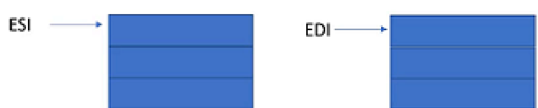
8.6.4 [Privilegiate] Istruzioni stringa per l'I/O - INSssuf, OUTSsuf

Attenzione Sono istruzioni privilegiate. Ne riparleremo a *Calcolatori elettronici.*

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

8.6.5 COMPARE STRINGS (CMPSsuf) - confronto memoria-memoria

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

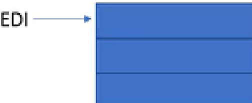


Osservazione Il soggetto del confronto è la source, e non destination come con la CMP.

8.6.6 SCAN STRING (SCASsuf) - confronto registro-memoria

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

Serve a trovare un elemento di valore noto dentro a un vettore
- DF = 0: cerca la prima occorrenza
- DF = 1: cerca l'ultima occorrenza



Osservazione Il soggetto del confronto è la source, e non destination come con la CMP. A pagina 49 della dispensa si ha una spiegazione corretta, mentre quella a pagina 69 (nell'appendice utilizzabile all'esame) è sbagliata. Cosa confermata via mail dall'ing.Zippo.

Esempio Nel salto devo considerare il cambio di soggetto rispetto alla CMP.

```
# numero di elementi maggiori rispetto al numero indicato
.GLOBAL _main

.DATA
.SET num_elementi, 11
array_numeri: .BYTE 1, 2, 3, 4, 8, 9, 10, 11, 12, 13, 14
numero: .BYTE 8

.TEXT
_main:  NOP
        MOV $0x00, %EAX
        MOV $0x00, %EDX
        MOV numero, %AL # numero nel confronto
        MOV $0x00, %DL # contatore per le occorrenze maggiori rispetto a numero
        MOV $num_elementi, %ECX # numero di volte che eseguo la loop
        LEA array_numeri, %EDI
        CLD

ciclo:  SCASB
        JAE poi
        INC %DL
poi:    LOOP ciclo

fine:   MOV %DL, %AL
        CALL outdecimal_byte
        XOR %EAX, %EAX
        RET

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

```
DOSBox 0.7
Z:\>MOUNT C .
Drive C is mo

Z:\>C:

C:\>CD WORK

C:\WORK>DEBUG
GNU gdb 6.1.1
Copyright 200
GDB is free s
welcome to ch
Type "show co
There is abso
This GDB was
Breakpoint 1

Breakpoint 1,
10        _main
Current langu
(gdb) continu
Continuing.
6
Program exite
(gdb)
```

8.6.7 Prefissi di ripetizione

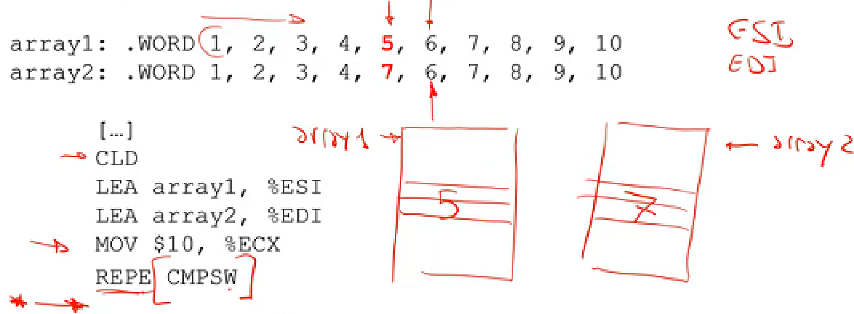
- REP
  - Puo' essere usato con MOVS, LODS, STOS, INS, OUTS.
  - Utilizzo con LODS è del tutto privo di senso, in quanto si finisce a sovrascrivere *N* volte lo stesso registro.
  - Si applica ad **una istruzione** (non ad un blocco di codice)
- REPE / REPNE
  - Puo' essere usato con CMPS, SCAS.
  - Si applica ad **una istruzione** (non ad un blocco di codice)
  - Si fanno **al massimo ECX ripetizioni**, finche' la condizione specificata e' vera

- Il prefisso non condizionato si utilizza con tutte le istruzioni di manipolazione tranne quelle di confronto. Si può utilizzare in LODS ma è privo di senso.
- Il prefisso condizionato si può utilizzare solo con le istruzioni di manipolazione per confronto.
- Necessario inizializzare il registro ECX per usare questi prefissi.

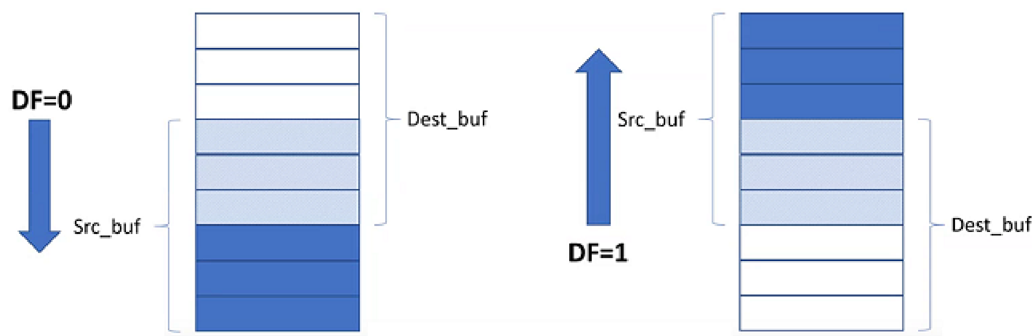
8.6.8 Utilità delle due direzioni

- **Trovare prima o ultima occorrenza di un dato in un vettore**.

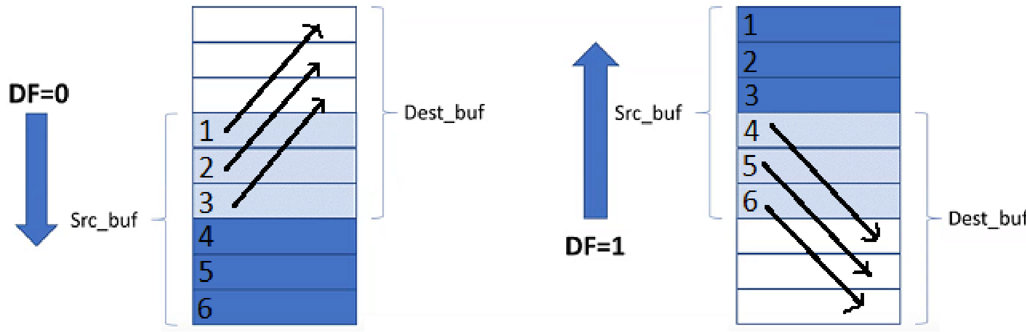
- Trovare il primo elemento differente tra due vettori



- **Copia di buffer parzialmente sovrapposti**. Attraverso un po' di algebra, e conoscendo le dimensioni dei buffer, possiamo notare se questi sono sovrapposti o meno. Se sono disgiunti possiamo copiare nel modo che preferiamo, se invece sono sovrapposti dobbiamo analizzare il tipo di sovrapposizione e procedere di conseguenza. Le situazioni possibili sono le seguenti:
  - prima il buffer destinatario e poi quello sorgente;
  - prima il buffer sorgente e poi quello destinatario



Dalle immagini deduciamo qual è la direzione di copia da assumere



in entrambi i casi le prime cose copiate sono quelle presenti in entrambi i buffer.

8.7 Ricapitoliamo

Abbiamo visto le seguenti istruzioni di manipolazione delle stringhe:

- **MOVE DATA FROM STRING TO STRING** (MOVSsuf): indico l'indirizzo della source in ESI e quello della destination in EDI. Copio da source a destination il numero di byte indicati nel suffisso. Incremento o decremento ESI ed EDI in base al direction flag e al numero di byte indicati nel suffisso.

- **LOAD STRING** (LODSsuf): indico l'indirizzo della source in ESI. Copio il contenuto dell'indirizzo puntato da ESI nel registro AL, AX o EAX in base al suffisso indicato. Incremento o decremento ESI in base al direction flag e al suffisso indicato.

- **STORE STRING** (STOSsuf): indico l'indirizzo della destination in EDI. Copio il contenuto presente nel registro AL, AX o EAX (in base al suffisso indicato) nell'indirizzo puntato da EDI. Incremento o decremento EDI in base al direction flag e al suffisso indicato.

- **COMPARE STRINGS** (CMPSsuf): pongo indirizzi in ESI e EDI. Eseguo un confronto (CMP) tra il contenuto puntato da ESI e quello puntato da EDI. Incremento o decremento i due registri in base al direction flag e al suffisso indicato. Attenzione: nella comparazione il soggetto è la source, e non la destination come normalmente avviene.

- **SCAN STRINGS** (SCASsuf): pongo un indirizzo in EDI e un valore nel registro AL, AX o EAX (in base al suffisso indicato). Svolgo un confronto (CMP) tra il il contenuto puntato da EDI e il valore presente nel registro. Incremento o decremento EDI in base al direction flag e al suffisso indicato. Attenzione: nella comparazione il soggetto è la source, e non la destination come normalmente avviene.

- **INSsuf**: pongo in DX l'indirizzo della porta I/O, in EDI un indirizzo di memoria. Memorizzo i byte ottenuti dalla porta a partire dall'indirizzo puntato da EDI. Incremento o decremento EDI in base al direction flag e al suffisso indicato.

- **OUTSsuf**: pongo in DX l'indirizzo della porta I/O, in ESI un indirizzo di memoria. Pongo il contenuto (attenzione al suffiso) puntato dall'indirizzo ESI nella porta I/O. Incremento o decremento ESI in base al direction flag e al suffisso indicato.

Osservazione 1 Si parla di stringhe, ma in realtà queste funzioni sono validissime per lavorare con vettori aventi elementi "di qualunque tipo".

Osservazione 2 Risulta necessario indicare il suffisso in tutte queste istruzioni: gli operandi sono impliciti.

Osservazione 3 Necessario indicare, prima di eseguire una di queste istruzioni, la direzione. Questa cosa si fa con le istruzioni STD o CLD, che permettono di settare o resettare il direction flag. Se il direction flag è uguale a 0 incremento i relativi registri, altrimenti li decremento (quanto incrementiamo o decrementiamo dipende dai suffissi indicati).

Capitolo 9

Venerdì 09/10/2020

9.1 Conclusione su Assembler

9.1.1 Differenze tra compilatore e assemblatore

Abbiamo già detto che...

- un compilatore C++ ottimizza il codice per il sistema su cui gira, mentre...
- un assemblatore traduce le istruzioni una per una.

Il compilatore ottimizza il nostro codice, mentre l'assemblatore traduce 1 : 1. Questo significa che l'efficienza del programma dipenderà esclusivamente da quanto scritto da noi.

9.1.2 Tempo di esecuzione di un programma

Il tempo di esecuzione non può essere determinato a partire dalle istruzioni: ciò che si misura non è il programma ma un processo[1]. Il tempo dipende

- dai dati
- dallo stato del sistema
- dipende da chi altri sta usando il processore, e da cosa ci fa.

Segue che se misuro n volte l'esecuzione dello stesso programma otterrò tempi molto diversi. La velocità di esecuzione è assolutamente imprevedebile. Inoltre

- il clock non va a velocità costante (dipende dal carico)
- il processo non gira necessariamente su un solo core
- la presenza di mille altri meccanismi che da una parte rendono i processi più veloci, ma da un'altra rendono i tempi più efficienti:
  - Memoria cache. Area posta tra CPU e RAM: molto piccola, ma in grado di ridurre i passaggi dalla RAM aumentando la velocità. Nel 98% dei casi i dati che mi servono stanno nella cache. Chiaramente se i dati non sono in cache dovremo passare dalla memoria RAM.

[1] Noi non assaggiamo la ricetta, ma la torta, cioè il prodotto della ricetta (cit.Stea)

- Code di prefetch.
- Esecuzione in pipeline
- Esecuzione non sequenziale
- Branch prediction.

Sono questioni che saranno affrontate alla magistrale o in corsi come *Calcolatori elettronici*.

9.1.3 Lunghezza delle istruzioni e tempo di fetch

Le istruzioni occupano un certo spazio in memoria. Ciò dipende:

- dall'OPCODE (quindi dall'istruzione)
- da dove sono gli operandi (in larga parte)

Se gli operandi sono registri le istruzioni stanno su un byte (normalmente), in caso di operandi immediati questi dovranno essere codificati nell'istruzione (e occuperanno 1,2, 4 byte).

Esempio

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

La MOV non è sbagliata, ma lo XOR occupa minore memoria e il fetch è più veloce.

9.1.4 Tempo di esecuzione delle istruzioni

Il tempo di esecuzione dipende molto dall'architettura del processore (anche all'interno della stessa famiglia, per esempio la Intel). In generale:

- le istruzioni operative della ALU (tranne divisione e moltiplicazione) costano poche: O(1) clock
- divisione e moltiplicazione costano molto: O(10) clock
- Istruzioni trascendenti della FPU (sin, cos) costano addirittura di più: O(100) clock
- Le istruzioni di controllo hanno un costo alto, ma per altri motivi.

9.1.4.1 Come si evitano moltiplicazioni e divisioni?

Moltiplicazioni e divisioni possono essere scomposte in catene di shift. Tenendo conto della dimensione degli operandi possiamo utilizzare le operazioni di shift al posto di quelle della divisione. Si osservi, inoltre, che la LEA può essere utilizzata per fare conti (ricordiamo la struttura generica di un indirizzo)

Compilatore C++ Il compilatore utilizza trucchi di questo tipo. Dire $a = b \cdot c$ non significa automaticamente tradurre con (I)MUL. Attraverso delle tabelle di corrispondenza il compilatore sceglie la traduzione più efficiente.

All'esame La cosa non è vitale: i programmi eseguiti sono "banali", segue che adottare un metodo o un altro non sia significativo nel determinare la velocità.

Es: $z = 16 \cdot x + 3 \cdot y - 13500$

OPCODE +-displacement (base, indice, scala)

$$\text{Indirizzo} = \left| \text{base} + \text{indice} \times \text{scala} \pm \text{displacement} \right|_{2^{32}}$$

```
MOV Y, %EAX
LEA -13500(%EAX,%EAX,2), %EAX
MOV X, %EBX
SHL $4, %EBX
ADD %EBX, %EAX
MOV %EAX, Z
```

$$\left| \underbrace{y + y \cdot 2}_{y \cdot 3} - 13500 \right|_{2^{32}}$$

$$16 \cdot x = 2^4 \cdot x$$

SOMMO LE DUE PARTI E OTTENGO Z

Parte II

Esercitazioni di Zippo

Capitolo 10

Martedì 06/10/2020

10.1 Assemblaggio

Dalla finestra di DOSBox poniamo

`ASSEMBLE.BAT PERCORSO\FILE.S`

saranno generati due file:

- il file eseguibile assemblato (nella stessa cartella dove si trova il file .s)
- il file listato.txt (che si trova nella cartella WORK)

10.1.1 File listato

La parte più interessante del file listato è quella relativa alla traduzione del codice. Possiamo vedere

- la riga di codice
- l'indirizzo di memoria (locazione di memoria dove è presente il dato o l'istruzione). Si capisce, da certi indirizzi, la dimensione delle variabili (si veda riga 5 e riga 6 come esempio)
- la relativa traduzione in bit

```
3                .DATA
4                # Abbiamo bisogno di un long per contare i dati e uno per svolgere il contegg
5 0000 01010F0F  dato:  .LONG 0x0F0F0101
6 0004 00        conteggio:  .BYTE 0x00 # Inizializzato a 0
7
8 0005 00000000  .TEXT
8      00000000
8      000000
9                # Inizializziamo i registri
10               # Definiamo un ciclo poichè dovremo analizzare tutta la variabile.
11               # Si fa il confronto per verificare se EAX è nullo. A un certo punto avremo
12               # Incrementiamo di 1 con ADD WITH CARRY (con la shift spostiamo sempre il bi
```

```
13 0000 90        _main : NOP
14 0001 B100      MOVB $0x00, %CL
15 0003 A1000000  MOVL dato, %EAX
15      00
16
17 0008 83F800    comp:  CMPL $0x00, %EAX
18 000b 7407      JE fine
19
20 000d D1E8      SHRL %EAX
21 000f 80D100    ADCB $0x00, %CL
22 0012 EBF4      JMP comp
23
24 0014 880D0400  fine: MOVB %CL, conteggio
24      0000
25 001a C3909090  RET
25      9090
```

Altra cosa utile, forse la più utile per noi che debuggiamo, è la lista dei simboli definiti e non definiti

```
DEFINED SYMBOLS
[...]
ESERCIZI\PRIMO.S:13     .text:00000000 _main
ESERCIZI\PRIMO.S:5      .data:00000000 dato
ESERCIZI\PRIMO.S:6      .data:00000004 conteggio
ESERCIZI\PRIMO.S:17     .text:00000008 comp
ESERCIZI\PRIMO.S:24     .text:00000014 fine

NO UNDEFINED SYMBOLS
```

in fondo avremo la lista di simboli non definiti in caso di errore.

10.2 Debugging

Come debugger utilizziamo GDB. Eseguiamo DEBUG.BAT con il percorso del file eseguibile

```
C:\WORK>DEBUG.BAT ESERCIZI\PROVA2.EXE
GNU gdb 6.1.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i386-pc-msdosdjgpp --target=djgpp"...
Breakpoint 1 at 0x1e60: file ESERCIZI/PROVA2.S, line 10.

Breakpoint 1, main () at ESERCIZI/PROVA2.S:10
warning: Source file is more recent than executable.

10      _main : NOP
Current language:  auto; currently asm
(gdb)
```

come vedremo GDB è molto scomodo da utilizzare. Non appena viene aperto il debugger si pone un breakpoint all'inizio del programma in modo da poter decidere come muoverci.

10.2.1 Comandi

Si distinguono due tipi di comandi: comandi per controllare l'esecuzione e comandi per controllare lo stato del programma. Essi sono:

- **list source**:

  `[list|l] X`

  mostra uno spezzone del file sorgente. X, facoltativo, consiste in un numero identificativo di una riga. Se omesso mi mostra la parte di file sorgente eseguita fino ad ora, se non omesso mi mostra la parte di file sorgente fino al rigo X.

```
Current language:  auto; currently asm
(gdb) l
5       .DATA
6       dato: .LONG 0b1010101 # Ho inizializzato ponendo come valore 5 (notazion
e decimale)
7       risultato: .BYTE 0x00
8
9       .TEXT
10      _main : NOP
11              MOVB dato, %EAX
12              MOVL $0x00, %CL
13
14      comp :  CMPL $0x00, %EAX
(gdb)
```

- **insert breakpoint**:

  `[break|b] nome_istruzione`

  inserisco un breakpoint all'istruzione individuata dal nome indicato. Si può indicare il numero di riga al posto del nome dell'istruzione

```
10      _main : NOP
Current language:  auto; currently asm
(gdb) break comp
Breakpoint 2 at 0x1e68: file ESERCIZI/PROVA2.S, line 14.
(gdb) break fine
Breakpoint 3 at 0x1e74: file ESERCIZI/PROVA2.S, line 20.
(gdb) _
```

- **info breakpoints**:

  `[info breakpoints|i b]`

  mostra i breakpoint inseriti. Per ogni breakpoint abbiamo un numero identificativo e l'istruzione a cui è associato

```
Breakpoint 3 at 0x1e74: file ESERCIZI/PROVA2.S, line 20.
(gdb) i b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00001e60 ESERCIZI/PROVA2.S:10
        breakpoint already hit 1 time
2   breakpoint     keep y   0x00001e68 ESERCIZI/PROVA2.S:14
3   breakpoint     keep y   0x00001e74 ESERCIZI/PROVA2.S:20
(gdb)
```

- **delete breakpoint**:

  `[delete breakpoint|d b] NUMBREAKPOINT`

  il numero del breakpoint è facoltativo: se omesso comporta l'eliminazione di tutti i breakpoint inseriti.

```
(gdb) d b 2
(gdb) i b
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00001e60 ESERCIZI/PROVA2.S:10
        breakpoint already hit 1 time
3   breakpoint     keep y   0x00001e74 ESERCIZI/PROVA2.S:20
(gdb)
```

- **step**:

  `[step | s] n`

  permette di procedere avanti di un certo numero di istruzioni. n può essere omesso, in quel caso è uguale ad 1

```
(gdb) step
11              MOVB dato, %EAX
(gdb) step
12              MOVL $0x00, %CL
(gdb) step 2
15              JE fine
(gdb) step
16              SHRL %EAX
(gdb)
```

- **continue**

  `[continue| c]`

  mette in esecuzione il programma dall'istruzione successiva fino al prossimo breakpoint

```
(gdb) continue
Continuing.

Breakpoint 3, fine () at ESERCIZI/PROVA2.S:20
20      fine :  MOVB %CL, risultato
(gdb) _
```

- **info registers**:

  `[info registers| i r]`

  mostra il contenuto di tutti i registri

```
Breakpoint 3, fine () at ESERCIZI/PROVA2.S:20
20      fine :  MOVB %CL, risultato
(gdb) i r
eax            0x0      0
ecx            0x4      4
edx            0x33f    831
ebx            0x102e   4142
esp            0x8dd44  0x8dd44
ebp            0x8dd58  0x8dd58
esi            0x54     84
edi            0xdd60   56672
eip            0x1e74   0x1e74
eflags         0x3046   12358
cs             0xe7     231
ss             0xef     239
ds             0xef     239
es             0xef     239
fs             0xdf     223
gs             0xff     255
(gdb) _
```

- **print**:

  `[print| p]/k $reg`

  mostra il contenuto del registro *reg*. K può essere omesso e permette di ottenere una rappresentazione in particolare del valore.

  - $k = x$, rappresentazione esadecimale
  - $k = t$, rappresentazione binaria
  - $k = d$, stampa come decimale interpretato come naturale
  - $k = u$, stampa come decimale interpretato come intero.

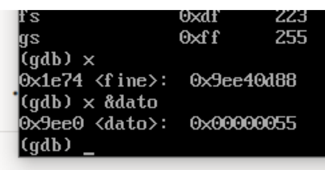
- **examine memory**:

  `x /numerotipo nome_di_una_variabile`

  visualizza il valore della variabile riferita.

  - numero consiste nel numero di componenti della variabile da esaminare. Se omesso numero equivale a 1
  - tipo consiste nel tipo della variabile. Gli unici valori possibili sono b, w ed l. Se omesso tipo sarà l.

  Ovviamente non è possibile omettere entrambi i parametri. Con le etichette simboliche è necessario porre la & commerciale poco prima. Se lascio vuoto la locazione implicita consiste in quella dell'istruzione appena eseguita.



- **quit**: per uscire dal debugger ed eseguire nuovi comandi sul DOS.

```
 1 #conteggio dei bit a 1 in un long
 2
 3 .GLOBAL _main
 4
 5 .DATA
 6 dato:        .LONG 0x0F0F0101
 7 conteggio:   .BYTE 0x00
 8
 9
10 .TEXT
11 _main:  NOP
12         MOVB $0x00, %CL
13         MOVL dato, %EAX
14
15 comp:   CMPL $0x00, %EAX    # while(eax != 0) {
16         JE fine
17
18         SHRL %EAX           # bit = eax[0]
19         ADCB $0x00, %CL     # if(bit == 1) cl++;
20
21         JMP comp            # }
22
23 fine:   MOVB %CL, conteggio
24         RET
25
```

```
1  # conteggio del numero di occorrenze di una lettera in una stringa
2
3  .GLOBAL _main
4
5  .DATA
6  stringa:    .ASCIZ "Questa e' la stringa di caratteri ASCII che usiamo come esempio"
7  lettera:    .BYTE 'e'
8  conteggio:  .BYTE 0x00
9
10 .TEXT
11 _main:      NOP
12             MOV $0x00, %CL
13             LEA stringa, %ESI
14             MOV lettera, %AL
15
16 comp:       CMPB $0x00, (%ESI)  # while (c != '\0') {
17             JE fine
18
19             CMP (%ESI), %AL
20             JNE poi             # if(c == 'e') {
21             INC %CL             #    CL++
22                                 # }
23
24 poi:        INC %ESI            # c = stringa[i++]
25             JMP comp            # }
26
27
28 fine:       MOV %CL, conteggio
29             RET
30
```

```
1  #conteggio dei bit a 0 in un long
2
3  .GLOBAL _main
4
5  .DATA
6  # Abbiamo bisogno di un long per contare i dati e uno per svolgere il conteggio
7  dato:       .LONG 0x0F0F0101
8  conteggio:  .BYTE 0x00
9
10 .TEXT
11 # Inizializziamo i registri
12 # Definiamo un ciclo poichè dovremo analizzare tutta la variabile.
13 # Si fa il confronto per verificare se EAX è nullo. A un certo punto avremo solo bit
   uguali a 0
14 # Utilizzo il registro DL per salvare tutte le volte il valore del CF (con la shift
   spostiamo sempre il bit "espulso" nel carry flag): resetto DL e incremento con ADC
   (ovviamente mi interessa solo il CF).
15 # Se il valore del flag è diverso da 0 (e quindi uguale ad 1) salto l'incremento
16
17 _main:  NOP
18         MOVB $0x00, %CL
19         MOVL dato, %EAX
20
21 comp:   CMPL $0x00, %EAX    # while(eax != 0) {
22         JE fine
23
24         SHRL %EAX           # bit = eax[0]
25         MOVB $0x00, %DL     # DL = 0
26         ADCB $0x00, %DL     # DL = DL + bit
27         CMPB $0x00, %DL     # if(DL == 0) cl++;
28         JNE step
29         INC %CL
30
31 step:   JMP comp             # }
32
33 fine:   MOVB %CL, conteggio
34         RET
35
```

Capitolo 11

Venerdì 16/10/2020

```
1  # Conteggio del numero di occorrenze di un numero in un array
2
3  .GLOBAL _main
4  .INCLUDE "C:/amb_GAS/utility"
5
6  .DATA
7  # Inizializzo l'array attraverso una lista di numeri. In una seconda variabile indico
   il numero di elementi presenti nell'array
8  # Inizializzo ulteriori variabili per indicare il numero di cui voglio trovare le
   occorrenze e salvare il risultato finale del conteggio.
9  array:      .WORD 1, 256, 256, 512, 42, 2048, 1024, 1, 0
10 array_len:  .LONG 9
11
12 numero:     .WORD 1
13 conteggio:  .BYTE 0x00
14
15 .TEXT
16 # Inizializzo CL, che sarà il mio contatore.
17 # Inizializzo AX, dove pongo il numero di cui voglio trovare le occorrenze.
18 # Inizializzo ESI, che mi servirà per scorrere il vettore (Registro source index) e
   verificare se ho visto tutti gli elementi
19 _main:      NOP
20             MOV $0, %CL
21             MOV numero, %AX
22             MOV $0, %ESI        # esi = 0
23
24 # Confronto ESI con la lunghezza dell'array, se le lunghezze coincidono ho finito
25 # Confronto AX, dove è presente il numero di cui vogliamo le occorrenze, con
   l'elemento dell'array in posizione ESI
26 # Se non c'è uguaglianza salto l'incremento di CL
27 comp:       CMP array_len, %ESI      # while (esi != array_len ) {
28             JE fine
29
30             CMPW array(, %ESI, 2), %AX   # if( array[esi] == numero )
31             JNE poi
32
33             INC %CL
34
35 poi:        INC %ESI            # esi++
36             JMP comp            # }
37
38 # Sposto il risultato in AL per poter effettuare la stampa del risultato con il
   sottoprogramma
39 fine:       MOV %CL, %AL
40             CALL outdecimal_byte
41             RET
42
```

```
1 # Conteggio del numero di occorrenze di un numero in un array
2 # Differenze rispetto al programma fatto con Zippo:
3 # - Indico in ingresso gli elementi dell'array
4 # - Indico in ingresso il numero di cui voglio verificare le occorrenze
5 # Il numero di elementi da controllare è indicato nella costante numero_elementi
6
7 .GLOBAL _main
8 .INCLUDE "C:/amb_GAS/utility"
9
10 .DATA
11 .SET numero_elementi, 9
12 array:      .FILL numero_elementi, 2 # Allocazione array: elementi da 2byte
13 conteggio:  .BYTE 0x00 # Conteggio delle corrispondenze
14
15 .TEXT
16 # Inizializzo CL, che sarà il mio contatore.
17 # AX è il registro che conterrà i numeri decimali inseriti in ingresso con la
   indecimal_word
18 # ESI lo utilizzo per scorrere l'array e individuare se ho visitato tutte le
   posizioni
19
20 _main:          NOP
21                 MOV $0, %CL
22                 MOV $0, %ESI        # esi = 0
23
24 popolamento:    CMP $numero_elementi, %ESI
25                 JE numdacontrollare
26
27                 CALL indecimal_word
28                 # Caldamente consigliata, situa orrenda senza
29                 CALL newline
30
31                 MOV %AX, array(, %ESI, 2)
32
33                 INC %ESI
34                 JMP popolamento
35
36 # Si estrae il numero di cui si vogliono trovare le corrispondenze. Anche in questo
   caso si guarda AX
37 # Confronto AX, dove è presente il numero di cui vogliamo le occorrenze, con
   l'elemento dell'array in posizione ESI
38 # Se non c'è uguaglianza salto l'incremento di CL
39 # NB: La ESI è adeguatamente resettata dopo l'uso precedente, idem la AX che viene
   sovrascritta con la indecimal_word
40
41 numdacontrollare:   CALL indecimal_word
42                     CALL newline
43 reset:              MOV $0, %ESI
44 comp:
45                     CMP $numero_elementi, %ESI
46                     JE fine
47
48                     CMPW array(, %ESI, 2), %AX
49                     JNE poi
50
51                     INC %CL
52
53 poi:                INC %ESI
54                     JMP comp
55
```

```
56 # Sposto il risultato in AL per poter effettuare la stampa del risultato con il
   sottoprogramma
57 fine:           MOV %CL, %AL
58                 CALL outdecimal_byte
59
60                 # Pongo per evitare la stampa di "Program exited with code X"
61                 XOR %EAX, %EAX
62
63                 RET
64
```

```
1 # Leggere messaggio da terminale, di sole lettere
2 # Convertire in minuscolo
3 # Stampare messaggio modificato
4 #
5 # Bonus: usare istruzioni stringa
6
7 .GLOBAL _main
8 .INCLUDE "C:/amb_GAS/utility"
9
10 .DATA
11 msg_in:     .FILL 80, 1, 0  # numero, dimensione, valore
12 msg_out:    .FILL 80, 1, 0
13
14 .TEXT
15 _main:      NOP
16
17             # lettura da terminale
18             MOV $80, %CX
19             LEA msg_in, %EBX
20             CALL inline
21
22             CLD
23             LEA msg_in, %ESI
24             LEA msg_out, %EDI
25
26 inizio:   LODSB                   # do {
27                                      # al = *esi
28
29             # elaborazione
30             CMP $0x41, %AL          # if ( al >= 'A' && al <= 'Z')
31             JB poi
32             CMP $0x5A, %AL
33             JA poi
34
35             OR $0x20, %AL
36
37 poi:        STOSB
38
39             CMP $0x0D , %AL         # while ( al != '\r' )
40             JNE inizio
41
42 fine:       LEA msg_out, %EBX
43             CALL outline
44             RET
45
46
```

```
 1 # Leggere messaggio da terminale, di sole lettere
 2 # Convertire in maiuscolo
 3 # Stampare messaggio modificato
 4 #
 5 # Bonus: usare istruzioni stringa
 6
 7 .GLOBAL _main
 8 .INCLUDE "C:/amb_GAS/utility"
 9
10 .DATA
11 msg_in:     .FILL 80, 1, 0  # numero, dimensione, valore
12 msg_out:    .FILL 80, 1, 0
13
14 .TEXT
15 _main:      NOP
16
17             # lettura da terminale
18             MOV $80, %CX
19             LEA msg_in, %EBX
20             CALL inline
21
22             CLD
23             LEA msg_in, %ESI
24             LEA msg_out, %EDI
25
26 inizio:   LODSB                   # do {
27                                      # al = *esi
28
29             # elaborazione
30             CMP $0x61, %AL          # if ( al >= 'a' && al <= 'z')
31             JB poi
32             CMP $0x7A, %AL
33             JA poi
34
35             AND $0xDF, %AL
36
37 poi:        STOSB
38
39             CMP $0x0D , %AL         # while ( al != '\r' )
40             JNE inizio
41
42 fine:       LEA msg_out, %EBX
43             CALL outline
44             RET
45
46
```

Capitolo 12

Venerdì 23/10/2020

12.1 Esercizio sul fattoriale

L'esercizio richiede di inserire in ingresso un numero che sia tra 0 e 9. Dato un valore n in ingresso vogliamo calcolare $n!$ e stamparlo.

Realizzazione del programma Il programma può essere realizzato in due modi:

- Per incremento da 2 ad al più 9

  ```
  n = indecimal_byte();
  fattoriale = 1;
  for(int i = 2; i <= n; i++)
  fattoriale = fattoriale*i;
  ```

- Per decremento da al più 9 fino a 0.

  ```
  n = indecimal_byte();
  fattoriale = 1;
  for(int i = n; i <= 2; i--)
  fattoriale = fattoriale*i;
  ```

Istruzione MUL Risulta ovvio che dovremo utilizzare la MUL per numeri naturali. I dubbi che sorgono riguardano il dimensionamento: quanto spazio ci serve?

- Ricordiamo che nell'istruzione MUL il dimensionamento viene deciso dall'operando sorgente indicato: l'altro operando, implicito, avrà la stessa dimensione dell'operando sorgente esplicito.
- Ricordiamo che nella MUL a 8 bit il risultato viene salvato in unico registro da 16bit, mentre nella MUL a 16 bit e in quella a 32 bit il risultato viene posto in due registri separati (due registri, rispettivamente, da 16 e 32 bit).
- Ricordiamo che con 8 bit possiamo rappresentare i numeri da 0 a 255 (dovreste già saperlo, quanto avete preso a Fondamenti? cit. Corsini), mentre con 16 bit i numeri da 0 a 65535.

- Il risultato più alto che possiamo ottenere, con entrambi gli algoritmi, è 362880 (contenibile in un registro a 32 bit). Tuttavia si osserva che
  - nell'algoritmo con incremento ci serve la MUL a 16 bit;
  - nell'algoritmo con decremento ci serve la MUL a 32 bit!

  entrambe le strategie sono praticabili.

Spiegazione codice Lungo il testo del codice sono presenti annotazioni.

- Se adottiamo il fattoriale con decremento utilizziamo la MUL a 32 bit. Sapendo che (ECX è l'operando esplicito)
  $$EDX_EAX = EAX * ECX$$
  posso porre l'istruzione

  ```
  MUL %ECX
  ```

  il risultato finirà tutte le volte in EAX. Non ho bisogno di interpellare EDX poichè il valore più alto possibile (362880) è rappresentabile in 32 bit.

- Se adottiamo il fattoriale con incremento utilizziamo la MUL a 16 bit. Sapendo che (CX è l'operando esplicito)
  $$DX_AX = AX * CX$$
  posso porre l'istruzione

  ```
  MUL %CX
  ```

  il risultato finirà tutte le volte in AX. Non ho bisogno di interpellare DX durante il ciclo: lo farò alla fine quando avrò l'unico risultato rappresentabile con più di 16bit. A quel punto attuo la strategia presente nel codice e spiegata attraverso le note.

```
 1 # Leggere numero naturale, sia n, da input
 2 # Controllare che sia tra 0 e 9
 3 # Calcolare e stampare in output il fattoriale n!
 4 #
 5 # Extra: organizzare il codice di calcolo del fattoriale come sottoprogramma
 6
 7 # n! = n * n - 1 * .... * 1
 8
 9 # n = indecimal_byte()
10 # # if( n > 9 )
11     # return;
12 #
13 # fattoriale = 1;
14 #
15 # for( int i = 2; i <= n; i++)
16      # fattoriale = fattoriale * i;
17 #
18 # 8 bit: da 0 a 255
19 # 16 bit: da 0 a 65535
20 #
21 # 2       x   1       =   2       (8 x 8 -> 16)
22 # 3       x   2       =   6       (8 x 8 -> 16)
23 # 4       x   6       =   24      (8 x 8 -> 16)
24 # 5       x   24      =   120     (8 x 8 -> 16)
25 # 6       x   120     =   720     (8 x 8 -> 16)
26 # 7       x   720     =   5040    (16 x 16 -> 32)
27 # 8       x   5040    =   40320   (16 x 16 -> 32)
28 # 9       x   40320   =   362880  (16 x 16 -> 32)
29 #
30 # for( int i = n; i >= 2; i--)
31     # fattoriale = fattoriale * i;
32 #
33 # 9       x   1       =   9       (8 x 8 -> 16)
34 # 8       x   9       =   72      (8 x 8 -> 16)
35 # 7       x   72      =   504     (8 x 8 -> 16)
36 # 6       x   504     =   3024    (16 x 16 -> 32)
37 # 5       x   3024    =   15120   (16 x 16 -> 32)
38 # 4       x   15120   =   60480   (16 x 16 -> 32)
39 # 3       x   60480   =   181440  (16 x 16 -> 32)
40 # 2       x   181440  =   362880  (32 x 32 -> 64)
41 #
42 # outdecimal(fattoriale);
43 #
44
45 .GLOBAL _main
46 .INCLUDE "C:/amb_GAS/utility"
47
48 .DATA
49
50 n:  .BYTE   0
51 risultato:  .LONG   1
52
53 msg_1:          .ASCII "Inserire naturale n da tra 0 e 9:\r"
54 msg_2:          .ASCII "Il fattoriale di n (n!) e':\r"
55
56 .TEXT
57
58 _main:          NOP
59
60                 LEA msg_1, %EBX
```

```
61                  MOV $80, %ECX
62                  CALL outline
63
64                  CALL indecimal_byte
65                  CALL newline
66                  MOV %AL, n
67
68                  MOV $0, %ECX
69                  MOVB n, %CL
70
71                  CALL factorial_inc
72                  MOV %EAX, risultato
73
74 fine:            LEA msg_2, %EBX
75                  MOV $80, %ECX
76                  CALL outline
77
78                  MOV risultato, %EAX
79                  CALL outdecimal_long
80                  RET
81
82 # sottoprogramma fattoriale, da n a 2
83 # input: ECX naturale da 0 a 9
84 # output: EAX fattoriale del numero (1 se invalido)
85 # sporca: EDX
86 factorial_dec:
87                  MOV $1, %EAX    # fara' da risultato e moltiplicando
88
89                  # controllo validita'
90                  CMP $2, %ECX
91                  JB  fine_factorial_dec
92                  CMP $9, %ECX
93                  JA  fine_factorial_dec
94
95 ciclo_factorial_dec:
96                  CMP $1, %CL     # while( cl > 1) {
97                  JE fine_factorial_dec
98
99                  MUL %ECX        # edx_eax = eax * ecx
100                 DEC %CL
101                 JMP ciclo_factorial_dec   # }
102
103 fine_factorial_dec:
104                 RET
105
106
107 # sottoprogramma fattoriale, da 2 a n
108 # input: ECX naturale da 0 a 9
109 # output: EAX fattoriale del numero (1 se invalido)
110 # sporca: EDX, BX
111 factorial_inc:
112                 MOV $1, %AX    # fara' da risultato e moltiplicando
113                 MOV $0, %DX
114
115                 # controllo validita'
116                 CMP $2, %ECX
117                 JB  fine_factorial_inc
118                 CMP $9, %ECX
119                 JA  fine_factorial_inc
120
```

Handwritten notes:

- (61–62) STAMPA MSG_1
- (64–66) RICHIEDO INPUT
- (68) PULISCO (VITALE PER QUANTO FAREMO DOPO)
- (69) PONGO L'INPUT E USO PIÙ AVANTI PER CONFRONTO
- (71) SOTTOPROGR.
- (74–76) STAMPO MSG_2
- (78–79) STAMPO IL RISULTATO
- (82–85) NON MI SERVE CALCOLARE IL FATTORIALE DI 1.
- (90–93) ESCO SE $ECX \notin [2,9]$
- (95–101)
  - RIMANGO NEL CICLO FINCHÉ CL NON SARÀ UGUALE AD 1.
  - OGNI VOLTA APPLICO LA MUL A 32 BIT. PERCHÉ?
    - OGNI VOLTA IL RISULTATO VA IN EAX
    - NON HO BISOGNO DI INTERPELLARE EDX
- (115–119) STESSI CONFRONTI DI PRIMA
- USO ECX CHE HA LO STESSO VALORE DI CL (ECX È STATO PULITO ALL'INIZIO)

```
121                 MOV $2, %BX
122
123 ciclo_factorial_inc:
124                 # do {
125
126                 MUL %BX      # dx_ax = ax * bx
127
128                 CMP %BX, %CX
129                 JE fine_factorial_inc
130
131                 INC %BX
132                 JMP ciclo_factorial_inc  # } while( cl > 1 )
133
134 fine_factorial_inc:
135                 # edx = ?_rh
136                 # eax = ?_rl
137                 SHL $16, %EDX    # edx = rh_0
138                 MOV %AX, %DX    # edx = rh_rl
139
140                 MOV %EDX, %EAX  # eax = rh_rl
141                 RET
142
```

INIZIALIZZO LA VARIABILE DA INCREMENTARE

APPLICO LA MUL A 16 BIT

CONFRONTO DOPO COME NEL DO-WHILE

- FINO ALL'ULTIMO STEP MI BASTA UN SOLO REGISTRO (AX) PER GESTIRE I RISULTATI PRECEDENTI.
- DX LO INTERPELLO SOLO ALLA FINE, COL RISULTATO FINALE

- EDX È OCCUPATO AL PIÙ NELLE PRIME 16 CIFRE MENO SIGNIFICATIVE
- EFFETTUO SHIFT E LIBERO LE CIFRE MENO SIGNIFICATIVE PER AX
- SPOSTO AX IN DX (DX EQUIVALENTE DELLE PRIME 16 CIFRE MENO SIGNIFICATIVE DI EDX)
- ADESSO HO TUTTO IL RISULTATO IN EDX
- SPOSTO IN EAX (ABBIAMO STABILITO DI PORRE IL RISULTATO LI)

12.2 Calcolo del binomiale

Il calcolo del binomiale

$$\frac{n!}{k!(n-k)!} \qquad n \geq k$$

può essere fatto sfruttando il codice scritto nel problema precedente. Nel codice che segue abbiamo utilizzato la versione del fattoriale con incremento (non per ragioni particolari). Le strategie adottate per il problema sul fattoriale relativamente ai registri (utilizzo del registro con le cifre meno significative) è valida anche per il binomio. Osserviamo che

- Il numeratore è rappresentabile in 32bit.
- Il denominatore è rappresentabile in 32bit

Useremo la DIV a 32 bit: abbiamo un dividendo a 64 bit diviso tra due registri e un divisore esplicito a 32 bit. Il risultato sarà a 32 bit, posto in un unico registro!

```
1  # Leggere due numeri naturali, a e b, da input
2  # Controllare che siano tra 0 e 9, e a >= b
3  # Calcolare e stampare in output il coeff. binomiale ( a  b ) = a! / ( b! * (a - b)!
   )
4
5  # a, b => 8 bit
6  # a!, b!, (a - b)! => 32 bit
7  # denom. =>
8          # (a-b) in N
9          # a! >= denom.
10         # => denom. su 32 bit
11
12 # divisore 32 bit => dividendo 64 bit
13
14 #   a_fatt = fattoriale(a)
15 #   b_fatt = fattoriale(b)
16 #   ab_fatt = fattoriale(a - b)
17 #   denom = b_fatt * ab_fatt
18 #   risultato = a_fatt / denom
19
20 .GLOBAL _main
21 .INCLUDE "C:/amb_GAS/utility"
22
23 .DATA
24 a:          .BYTE 0
25 b:          .BYTE 0
26 a_fatt:     .LONG 0
27 b_fatt:     .LONG 0
28 ab_fatt:    .LONG 0
29 denom:      .LONG 0
30 risultato:  .LONG 0
31
32 msg_1: .ASCII "Inserire i due naturali a e b, da 0 a 9:\r"
33 msg_2: .ASCII "Il coefficiente binomale (a b) e':\r"
34 msg_err: .ASCII "Input invalidi\r"
35
36 .TEXT
37
38 _main:
39             NOP
40
41             # lettura a e b
42             LEA msg_1, %EBX
43             MOV $80, %ECX
44             CALL outline
45
46             CALL indecimal_byte
47             CALL newline
48             MOV %AL, a
49
50             CALL indecimal_byte
51             CALL newline
52             MOV %AL, b
53
54             # controllo validita' a e b
55             CMPB $9, a
56             JA wrong_input
57             CMPB $9, b
58             JA wrong_input
59
```

- INPUT IN 8 BIT
- FATTORIALI IN 32BIT (QUINDI ANCHE NUMERATORE)
- DENOMINATORE IN 32BIT

STESSA STRATEGIA DEL PROBLEMA PRECEDENTE (GUARDO UN SOLO REGISTRO)

* - SO CHE IL NUM. STA IN 32 bit
- SO CHE IL NUM È MAGGIORE O UGUALE AL DENOMINATORE
- SEGUE CHE IL DENOMINATORE STA IN 32 BIT

INPUT (a, b)

SALVO I DATI CALCOLATI NEL TEMPO

OUTPUT (risultato)

STAMPO MSG_1

CHIEDO a

CHIEDO b

CONTROLLO CHE LE CONDIZIONI RICHIESTE SIANO RISPETTATE

```
60          MOVB a, %AL
61          MOVB b, %BL
62          CMP %AL, %BL
63          JA wrong_input
64
65          # calcolo dei fattoriali
66          MOV $0, %ECX
67          MOVB a, %CL
68          CALL factorial_inc
69          MOV %EAX, a_fatt
70
71          MOV $0, %ECX
72          MOVB b, %CL
73          CALL factorial_inc
74          MOV %EAX, b_fatt
75
76          MOV $0, %ECX
77          MOVB a, %CL
78          SUB b, %CL # cl -= b => cl = a - b
79          CALL factorial_inc
80          MOV %EAX, ab_fatt
81
82          # calcolo del denominatore
83          MOV b_fatt, %EAX
84          MOV ab_fatt, %EBX
85          MUL %EBX    # edx_eax = eax * ebx
86          MOV %EAX, denom
87
88          # divisione
89          MOV $0, %EDX
90          MOV a_fatt, %EAX
91          MOV denom, %EBX
92          DIV %EBX    # EAX = EDX_EAX / EBX
93          MOV %EAX, risultato
94
95          # stampa del risultato
96          LEA msg_2, %EBX
97          MOV $80, %ECX
98          CALL outline
99
100         MOV risultato, %EAX
101         CALL outdecimal_long
102         RET
103
104 wrong_input:
105         LEA msg_err, %EBX
106         MOV $80, %ECX
107         CALL outline
108         RET
109
110
111
112 # sottoprogramma fattoriale, da 2 a n
113 # input: ECX naturale da 0 a 9
114 # output: EAX fattoriale del numero (1 se invalido)
115 # sporca: EDX, BX
116 factorial_inc:
117                 MOV $1, %AX    # fara' da risultato e moltiplicando
118                 MOV $0, %DX
119
```

SE QUALCOSA NON É RISPETTATO FACCIO JUMP.

a! b! (a-b)!

CALCOLIAMO I FATTORIALI COL CODICE DEL PROBLEMA PRECEDENTE

NUMERATORE: a!

b!(a-b)!

MUL A 32 BIT

ISTRUZIONE DIV

$\frac{a!}{b!(a-b)!}$

DIV A 32 BIT (DIVISORE A 32 BIT)

STAMPO MSG_2

STAMPO IL RISULTATO

STAMPO MSG_ERR

CI SI FERMA

```
120                 # controllo validita'
121                 CMP $2, %ECX
122                 JB  fine_factorial_inc
123                 CMP $9, %ECX
124                 JA  fine_factorial_inc
125
126                 MOV $2, %BX
127
128 ciclo_factorial_inc:
129                 # do {
130
131                 MUL %BX    # dx_ax = ax * bx
132
133                 CMP %BX, %CX
134                 JE fine_factorial_inc
135
136                 INC %BX
137                 JMP ciclo_factorial_inc  # } while( cl > 1 )
138
139 fine_factorial_inc:
140                 # edx = ?_rh
141                 # eax = ?_rl
142                 SHL $16, %EDX    # edx = rh_0
143                 MOV %AX, %DX    # edx = rh_rl
144
145                 MOV %EDX, %EAX  # eax = rh_rl
146                 RET
147
```

VEDERE INDIETRO n! PER SPIEGAZIONE

Parte III

Esercizi di Assembler

Esercizio 1: Assembly

Scrivere un programma che si comporta come segue:

1. legge con eco da tastiera una stringa di lettere minuscole terminata da ritorno carrello;
2. legge con eco da tastiera una lettera minuscola;
3. stampa sulla riga il numero di occorrenze nella stringa della lettera;
4. se il numero di occorrenze è maggiore di uno torna al passo 2, altrimenti termina.

NB: si assuma che la stringa possa essere di lunghezza qualunque, ma il numero di occorrenze di ciascuna lettera stia su 8 bit.

Esempio:

```
aabcdddhbbc
a 2
d 3
f 0
```

```
 1 .GLOBAL _main
 2 .INCLUDE "C:/amb_GAS/utility"
 3 
 4 .DATA
 5 vettore:    .FILL 26, 1, 0
 6 
 7 .TEXT
 8 _main:                  NOP
 9                         XOR %EAX, %EAX
10                         CALL richiesta_lettere
11 
12 lettera_da_verificare:  CALL richiesta_lettera
13                         CALL spaziobianco
14                         SUB $'a', %AL
15 
16                         LEA vettore(%EAX), %ESI
17                         LODSB
18                         CALL outdecimal_byte
19                         CALL newline
20 
21                         CMP $1, %AL
22                         JA lettera_da_verificare
23 
24 fine_programma:         XOR %EAX, %EAX
25                         RET
26 
27 
28 /* Sottoprogramma per la richiesta della stringa di lettere minuscole */
29 richiesta_lettere:      NOP
30                         CALL richiesta_lettera
31                         CMP $0x0D, %AL
32                         JE fine_richiesta_lettere
33 
34                         SUB $'a', %AL
35                         INCB vettore(%EAX)
36 
37                         JMP richiesta_lettere
38 fine_richiesta_lettere: RET
39 
40 
41 /* Richiedo una singola lettera. Valore restituito in AL
42 e controlli sulla validita' della lettera */
43 richiesta_lettera:      NOP
44                         CALL inchar
45 
46                         CMPB $0x0D, %AL
47                         JE fine_richiesta_acapo
48 
49                         CMPB $'a', %AL
50                         JB richiesta_lettera
51 
52                         CMPB $'z', %AL
53                         JA richiesta_lettera
54 
55 tutto_ok:               CALL outchar
56                         JMP fine_richiesta_lettera
57 fine_richiesta_acapo:   CALL newline
58 fine_richiesta_lettera: RET
59 
60 /* Sottoprogramma per lo spazio bianco */
61 spaziobianco:           PUSH %AX
62                         MOV $' ', %AL
63                         CALL outchar
64                         POP %AX
65                         RET
66 
```

Esercizio 1: Assembly

Scrivere un programma che si comporta come segue:

1. Legge con eco da tastiera una cifra in base 10 N, effettuando gli opportuni controlli
2. Se $N = 0$, termina.
3. Altrimenti, va a capo e legge con eco da tastiera una cifra in base 10 k, effettuando gli opportuni controlli
4. Stampa N righe, ciascuna di 1, 2, ... N numeri, in modo che tutti i numeri formino un'unica sequenza crescente di passo k a partire da 1.
5. Lascia una riga bianca e ritorna al punto 1.

Nota: per opportuni controlli si intende che il programma ignora e non fa eco caratteri inattesi, rimanendo in attesa di una cifra.

Esempio (allegato in formato .txt):

```
5
1
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15

3
2
1
3 5
7 9 11

0[terminazione]
```

```
1 .GLOBAL _main
2
3 .DATA
4 N:              .BYTE 0x00
5 step_k:         .BYTE 0x00
6
7 .TEXT
8 _main:          NOP
9                 CALL chiedi_cifra
10                CMP $0, %AL
11                JZ stop
12
13                MOV %AL, %BL
14
15                CALL chiedi_cifra
16                MOV %AL, %DL
17
18                MOV $0x00, %ECX
19                MOV $0x01, %AL
20
21 ciclo1:        CMP %BL, %CL
22                JE fine
23
24                MOV $0x00, %AH
25 ciclo2:        CALL outdecimal_byte
26                CALL spazio_vuoto # spazio vuoto tra numeri
27                ADD %DL, %AL #incremento
28                INC %AH # numeri mostrati nella riga
29 fine_ciclo2:   CMP %CL, %AH
30                JBE ciclo2
31                CALL newline # vado a capo nell'output
32
33 fine_ciclo1:   INC %CL
34                JMP ciclo1
35
36 fine:          CALL newline
37                JMP _main
38
39 stop:          XOR %EAX, %EAX
40                RET
41
42
43 /* Richiedo una cifra e ne verifico la validita ' */
44 /* Non ho registri, input ma ho un output posto nel registro AL */
45 chiedi_cifra:  CALL inchar
46
47                CMP $'0', %AL
48                JB chiedi_cifra
49
50                CMP $'9', %AL
51                JA chiedi_cifra
52
53                CALL outchar
54                CALL newline
55
56            AND $0x0F, %AL
57
58                RET
59
60 /* Semplice sottoprogramma per ottenere uno spazio vuoto tra i vari numeri */
61 spazio_vuoto:  PUSH %AX
62                MOV $' ', %AL
63                CALL outchar
64                POP %AX
65                RET
66
67
68 .INCLUDE "C:/amb_GAS/utility"
```

Esercizio 1: Assembly

Scrivere un programma che si comporta come segue:

1. legge con eco da tastiera le rappresentazioni A di un intero a, e B di un intero b, in complemento alla radice su 2 cifre in base dieci, facendo i dovuti controlli. I due numeri devono essere stampati su righe separate.
2. se l'intero $c = a + b$ è rappresentabile in complemento alla radice su 2 cifre in base dieci: stampa, su una riga separata, la rappresentazione C di c, lascia una riga vuota e ritorna al passo 1. Altrimenti, termina.

Nota: per "dovuti controlli" si intende che il programma ignora e non fa eco di caratteri inattesi. Vengono accettate soltanto le codifiche di esattamente 2 cifre decimali.

Un esempio di output è allegato in formato .txt .
Si ponga attenzione alla formattazione di questo file, che fa parte delle specifiche.

```
 1 .GLOBAL _main
 2 .INCLUDE "C:/amb_GAS/utility"
 3
 4 .TEXT
 5 _main:              NOP
 6
 7                     XOR %ECX, %ECX
 8
 9                     # Primo numero
10                     CALL lettura_numero # Richiedo il numero
11                     CALL verifica_neg # Incremento CL se necessario
12                     MOV %AH, %BH
13                     MOV %AL, %BL
14
15                     # Vado a capo
16                     CALL newline
17
18                     # Secondo numero
19                     CALL lettura_numero # Richiedo il numero
20                     CALL verifica_neg # Incremento CL se necessario
21
22                     # Vado a capo
23                     CALL newline
24
25                     # A questo punto possiamo fare, in modo rudimentale, BH_BL +
   AH_AL
26                     ADD %BL, %AL
27                     CMP $9, %AL
28                     JBE no_carry1
29
30                     INC %CH # Setto il riporto uscente ad 1
31                     SUB $10, %AL # Risultato della somma decrementato in caso di
   riporto uscente settato
32
33 no_carry1:          ADD %BH, %AH
34                     ADD %CH, %AH
35
36                     XOR %CH, %CH # Resetto il riporto uscente dopo averlo usato
37
38                     CMP $9, %AH
39                     JBE no_carry2
40
41                     # Non mi serve il riporto uscente della seconda colonna, quindi
   non faccio incrementi in nessun registro
42                     SUB $10, %AH # Risultato della somma decrementato in caso di
   riporto uscente settato
43
44 no_carry2:          # A questo punto dobbiamo verificare che il risultato sia di
   segno positivo o negativo, e tirare le conclusioni
45                     # La somma si trova in AH_AL
46                     # Riprendiamo il registro %CL e verifichiamo quanto trovato
47
48                     # Verifico se i numeri sommati sono discordi, in quel caso non
   ho problemi e stampo
49                     CMP $1, %CL
50                     JE stampa_risultato
51
52                     # Verifico, se ho due numeri positivi sommati, che il risultato
   sia positivo. Se non lo e' termino l'esecuzione del programma
53                     CMP $0, %CL
```

```
54                      JNE verifica_concneg
55                      CMP $5, %AH
56                      JB stampa_risultato
57                      JMP fine
58
59  verifica_concneg:   # Non faccio confronti, sono certo che il valore di CL e' due a
    questo punto
60                      # Verifico, se ho due numeri negativi sommati, che il risultato
    sia negativi. Se non lo e' termino l'esecuzione del programma
61                      CMP $5, %AH
62                      JB fine
63                      JMP stampa_risultato
64
65  stampa_risultato:   NOP
66
67                      XCHG %AH, %AL
68                      CALL outdecimal_byte
69
70                      XCHG %AH, %AL
71                      CALL outdecimal_byte
72
73                      CALL newline
74                      CALL newline
75                      JMP _main
76
77  fine:               XOR %EAX, %EAX
78                      RET
79
80  /* Sottoprogramma per la lettura di un numero in base 10 a due cifre
81  con adeguati controlli. Pongo il numero letto in AH_AL */
82  lettura_numero:     CALL inchar
83                      CMP $'0', %AL
84                      JB lettura_numero
85                      CMP $'9', %AL
86                      JA lettura_numero
87                      AND $0x0F, %AL
88                      CALL outdecimal_byte
89                      MOV %AL, %AH
90
91  lettura_numero_2:   CALL inchar
92                      CMP $'0', %AL
93                      JB lettura_numero_2
94                      CMP $'9', %AL
95                      JA lettura_numero_2
96                      AND $0x0F, %AL
97                      CALL outdecimal_byte
98
99  fine_programma:     RET
100
101 /* Sottoprogramma che incrementa CL se il numero inserito e' negativo.
102 Guardo il registro AH, ricordandomi che a_{n-1} < \beta/2 se voglio avere un numero
    positivo */
103 # Conclusione: CL = 0 numeri positivi, CL = 1 numeri discordi , CL = 2 numeri
    negativi
104 verifica_neg:       NOP
105                     CMP $5, %AH
106                     JB fine_verifica
107                     INC %CL
108 fine_verifica:      RET
109
```

Esercizio 1: Assembly

Scrivere un programma in Assembler che svolge i seguenti compiti

1. richiede in ingresso due numeri naturali X ed Y in base 10, X a 6 cifre ed Y a 1 cifra, svolgendo gli opportuni controlli. In particolare:
   - Legge e fa eco di cifre in base 10, ignora e non fa eco di caratteri inattesi
   - Vengono accettate soltanto codifiche dal numero *esatto* di cifre
   - L'eco dei due input è fatto su due righe distinte
2. se $Y = 0$ termina, altrimenti
3. lascia una riga bianca ed esegue la *divisione in base 10* di X per Y, per passaggi successivi, stampando i risultati intermedi formattati come da esempio.
4. lascia una riga bianca e ritorna al punto 1.

Un esempio di output è allegato in formato .txt .
Si ponga attenzione alla formattazione di questo file, che fa parte delle specifiche.

```
 1 .GLOBAL _main
 2 .INCLUDE "C:/amb_GAS/utility"
 3
 4 .DATA
 5 vettore1: .FILL 6, 1, 0
 6 vettore2: .FILL 1, 1, 0
 7
 8 .TEXT
 9 _main:          MOV $6, %CL
10                 LEA vettore1, %EDI
11                 CALL richiedinumero
12
13                 # Richiedo Y
14                 MOV $1, %CL
15                 LEA vettore2, %EDI
16                 CALL richiedinumero
17
18                 # Verifico che Y \neq 0, se e' uguale termino
19                 CMP $0, vettore2
20                 JE fine_male
21
22                 CALL newline # Si chiede di lasciare una riga bianca
23
24 divisione:      MOV $0, %ESI # Dovro' scorrere il vettore
25                 MOV vettore2, %CL # Divisore, uguale in ogni caso
26                 MOV $6, %CH # Numero di divisioni da fare, variabile contatore
27                 MOV $0, %AL
28
29 ciclo_div:      CALL moltiplicatore
30                 # Ho in BX il numero moltiplicato per 10
31                 MOV vettore1(%ESI), %AL
32                 ADD %BL, %AL
33
34                 MOV $0, %AH
35
36                 PUSH %AX
37
38                 CALL outdecimal_byte
39
40                 MOV $'/', %AL
41                 CALL outchar
42
43                 MOV %CL, %AL
44                 CALL outdecimal_byte
45
46                 MOV $':', %AL
47                 CALL outchar
48
49                 MOV $' ', %AL
50                 CALL outchar
51
52                 POP %AX
53
54                 # A questo punto ho in AX il dividendo
55                 # Il divisore e' in CL
56                 DIV %CL
57                 # Dovrei avere quoziente in AL e resto in AH
58
59                 PUSH %AX
60
61                 MOV $'q', %AL
62                 CALL outchar
63
64                 MOV $'=', %AL
65                 CALL outchar
66
67                 POP %AX
68
69                 CALL outdecimal_byte
```

```
70
71                  PUSH %AX
72
73                  MOV $',', %AL
74                  CALL outchar
75
76                  MOV $' ', %AL
77                  CALL outchar
78
79                  MOV $'r', %AL
80                  CALL outchar
81
82                  MOV $'=', %AL
83                  CALL outchar
84
85                  MOV %AH, %AL
86                  CALL outdecimal_byte
87
88                  POP %AX
89
90                  INC %ESI
91
92                  MOV %AH, %AL # Metto in AL il resto
93
94                  DEC %CH
95                  CMP $0, %CH
96
97                  CALL newline
98                  JNE ciclo_div
99
100 fine_bene:      CALL newline
101                 JMP _main
102 fine_male:      XOR %EAX, %EAX
103                 RET
104
105 # Sottoprogramma con cui viene richiesto un numero naturale in base 10
106 # Input: CL (numero di cifre richiesto), EDI (puntatore a memoria)
107 # Output: contenuto del vettore puntato da EDI aggiornato
108 richiedinumero:     CLD
109 ciclo:              CALL inchar # Valore in AL
110                     CMP $'0', %AL
111                     JB ciclo
112                     CMP $'9', %AL
113                     JA ciclo
114                     CALL outchar
115                     AND $0x0F, %AL # Converto da codifica ASCII a numero
116                     STOSB # Aggiorno il vettore che contiene il numero
117 rn_poi:             DEC %CL
118                     CMP $0, %CL
119                     JNE richiedinumero # Esco solo dopo aver compiuto il numero di
    cicli indicato in CL
120 fine_rn:            CALL newline
121                     RET
122
123 # Sottoprogramma per la moltiplicazione
124 # Input: AL, cifra da moltiplicare
125 # Output: BX, cioe' AL moltiplicato per 10
126 moltiplicatore:     PUSH %AX
127                     PUSH %DX
128
129                     MOV $10, %DL
130                     MUL %DL
131
132                     MOV %AX, %BX
133
134 fine_moltiplic:     POP %DX
135                     POP %AX
136                     RET
137
```

Capitolo 17

Min e max con modulo e segno

Scrivere un programma in Assembler che svolge i seguenti compiti

1. richiede in ingresso un numero intero x in base 10 a 5 cifre, rappresentato in MODULO E SEGNO, svolgendo gli opportuni controlli, ed assumendo che ABS(x) stia su 16 bit.
2. se ABS(x)=0 termina, altrimenti
3. stampa l'intervallo in cui sono compresi i numeri digitati finora, sempre in modulo e segno.
4. ritorna al punto uno

Esempio

```
?+3
[+3 ; +3]
?-5
[-5 ; +3]
?+15305
[-5 ; +15305]
```

```
 1 .GLOBAL _main
 2 .INCLUDE "C:/amb_GAS/utility"
 3
 4 .DATA
 5 min: .LONG 0
 6 max: .LONG 0
 7
 8 .TEXT
 9 _main:              MOVL $+65535, min
10                 MOVL $-65535, max
11 inizio:             XOR %EAX, %EAX
12                     CALL richiedinum # segno in DL, modulo in AL
13                     CMP $0, %AL
14                     JE fine_male
15
16                     CALL conversionec2 # Prendo da EAX il numero convertito
17
18                     CALL ottieni_max_min
19
20                     CMP %EBX, %EAX
21                     JGE no_new_min
22
23                     MOV %EAX, min
24
25 no_new_min:         CMP %EDX, %EAX
26                     JL stampa_main
27
28                     MOV %EAX, max
29
30 stampa_main:        CALL ottieni_max_min
31                     CALL stampariga
32
33 fine_bene:          JMP inizio
34
35 fine_male:          XOR %EAX, %EAX
36                     RET
37
38 # Sottoprogramma con cui richiedo un numero in MS
39 # Uscite: registro CL (segno, 0 positivo, 1 negativo), registro AL (modulo)
40 richiedinum:        MOV $'?', %AL
41                     CALL outchar
42
43                     CALL inchar
44                     CMP $'+', %AL
45                     JE segno_p
46                     CMP $'-', %AL
47                     JE segno_n
48                     JMP richiedinum # segno indicato non valido
49
50 segno_p:            XOR %CL, %CL
51                     JMP modulo
52 segno_n:            MOV $1, %CL
53
54 modulo:             CALL outchar
55                     CALL indecimal_word # Modulo in %AX
56                     CALL newline
57                     RET
58
59 # Sottoprogramma per la conversione da MS a C2
60 # Input: CL e EAX
61 # Output: Registro EAX con la conversione
62 conversionec2:      CMP $0, %CL
63                     JE fineconversionec2
```

```
64
65                      NEG %EAX
66
67 fineconversionec2:    RET
68
69
70 # Sottoprogramma per la stampa di un numero in MS
71 # Input: EAX
72 # Output: stampa del numero in MS
73 stampanumero:   MOV %EAX, %EBX
74                 CMP $0, %EBX
75                 JL negativo
76
77                 MOV $'+', %AL
78                 CALL outchar
79                 MOV %EBX, %EAX
80                 CALL outdecimal_word
81                 JMP finestampanum
82
83 negativo:       MOV $'-', %AL
84                 CALL outchar
85                 MOV %EBX, %EAX
86                 NEG %EAX
87                 CALL outdecimal_word
88
89 finestampanum:  RET
90
91
92 # Sottoprogramma per la stampa della riga
93 # Input: EBX per il minimo ed EDX per il massimo
94 # Output: stampa della riga
95 stampariga:         PUSH %EAX
96                     MOV $'[', %AL
97                     CALL outchar
98
99                     MOV %EBX, %EAX
100                    CALL stampanumero
101
102                    MOV $' ', %AL
103                    CALL outchar
104
105                    MOV $0x3B, %AL
106                    CALL outchar
107
108                    MOV $' ', %AL
109                    CALL outchar
110
111                    MOV %EDX, %EAX
112                    CALL stampanumero
113
114                    MOV $']', %AL
115                    CALL outchar
116
117                    CALL newline
118
119                    POP %EAX
120                    RET
121
122 # Sottoprogramma per spostare nei registri massimo e minimo
123 # Output: EBX per il minimo, EDX per il massimo
124 ottieni_max_min:    MOV min, %EBX
125                     MOV max, %EDX
126                     RET
127
```

Capitolo 18

Fattorizzazione

Scrivere un programma che si comporta come segue:

1. stampa un punto interrogativo e legge con eco da tastiera un numero naturale A in base 10 (short)
2. se il numero è minore o uguale ad 1, termina
3. altrimenti, stampa la sua scomposizione in fattori primi, cioe' una lista di righe x^y, intendendo che il fattore primo x e' contenuto in A alla sua potenza y-esima, con $x > 1$, $y \geq 1$.
4. attende la pressione di un tasto e termina.

NB: si faccia l'ipotesi che il numero A sia rappresentabile su 16 bit.

Esempio

```
?338
2^1
13^2

?199
199^1

?1
[terminazione]
```

```
 1 .GLOBAL _main
 2 .INCLUDE "C:/amb_GAS/utility"
 3
 4 .TEXT
 5 _main:              CALL richiedi_numero
 6                     CMP $1, %AX
 7                     JBE fine
 8
 9 fattorizzazione:    MOV $2, %BX # Numero considerato, incremento nel tempo
10                     MOV %AX, %SI # Sposto perche' AX sara' sporcato dalla divisione
11
12 ciclo_numero:       MOV $0, %CL
13
14                     # Dividendo in AX, mentre DX deve stare vuoto.
15                     MOV %SI, %AX  # Lo rimetto uguale al dividendo ogni volta che
   provo un nuovo divisore
16
17 divisione:          XOR %DX, %DX # Pulisco DX per evitare problemi con la divisione
18                     DIV %BX
19                     # Il quoziente sta in AX, il resto in DX
20
21                     CMP $0, %DX
22                     JNE fine_numero # Se il resto e' diverso da 0 ho finito col
   numero che sta in BX
23                     INC %CL
24                     JMP divisione
25
26 fine_numero:        CMP $0, %CL
27                     JE poi
28                     CALL stampariga
29 poi:                INC %BX
30
31                     CMP $0, %AX
32                     JE fine
33                     JMP ciclo_numero
34
35 fine:               CALL pause
36                     XOR %EAX, %EAX
37                     RET
38
39 # Sottoprogramma con cui richiedo un numero naturale in base 10
40 # Output: AX
41 richiedi_numero:    MOV $'?', %AL
42                     CALL outchar
43
44                     CALL indecimal_word
45                     CALL newline
46                     RET
47
48 # Sottoprogramma con cui stampo uno step della fattorizzazione
49 # Input: BX per il numero, CL per l'esponente
50 stampariga:         PUSH %AX
51
52                     MOV %BX, %AX
53                     CALL outdecimal_word
54
55                     MOV $'^', %AL
56                     CALL outchar
57
58                     MOV %CL, %AL
59                     CALL outdecimal_byte
60
61                     CALL newline
62                     POP %AX
63 finestampa:         RET
64
```

Parte IV

Reti combinatorie

Capitolo 19

Venerdì 09/10/2020

19.1 Introduzione alle reti logiche

Una **rete logica** consiste in un modello astratto di un sistema fisico dove si hanno dispositivi tra loro interconnessi: questi dispositivi si scambiano informazioni codificate attraverso fenomeni fisici. Alcuni esempi di fenomeni sono:

- L'intensità di corrente
- La tensione
- Le perforazioni di un foglio di carta (pensare a quelle lavatrici usate in ambito professionale, precisamente in alberghi e ospedali, dove l'impiegato utilizza una tessera perforata per indicare alla macchina come svolgere un certo lavaggio)

le possibilità sono innumerevoli. Abbiamo sottolineato come questa rete logica sia un modello semplificato della realtà che ci permette di descrivere situazioni complesse. Nell'utilizzare questo modello noi siamo in grado di astrarre dalla realtà: ci occuperemo in particolare di circuiti elettronici limitandoci a descrivere il loro funzionamento (senza sapere la loro struttura o come si realizzano - competenza dell'ingegnere elettronico). Se teniamo conto dei limiti di questo modello (noi trascuriamo aspetti della realtà) potremo usarlo senza grossi problemi.

Cosa troviamo in una rete logica?

In una rete logica individuiamo:

- N variabili di ingresso identificate con valori che vanno da 0 ad $N-1$. L'insieme delle variabili di ingresso costituisce lo **stato di ingresso**, cioè i valori assunti in un istante temporale t da un insieme di variabili logiche di ingresso. Il numero di stati di ingresso possibili consiste in 2^N.

- M variabili di uscita identificate con valori che vanno da 0 ad $M-1$. L'insieme delle variabili di uscita costituisce lo **stato di uscita**, cioè i valori assunti in un istante temporale t da un insieme di variabili logiche di uscita. Il numero di stati di uscita possibili consiste in 2^M.

- una **legge di evoluzione del tempo** che dice come le uscite evolvono in funzione degli ingressi.

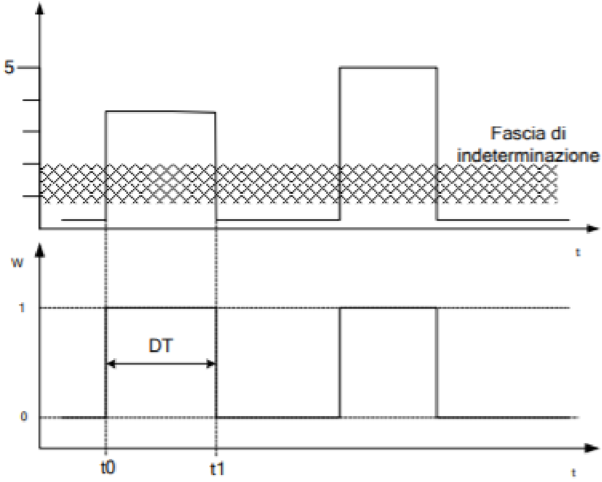
Abbiamo un contesto dove qualcuno stabilisce le variabili di ingresso e qualcun'altro legge le variabili di uscita. Queste variabili sono dette **variabili logiche** e possono assumere come valori 0 o 1 (si parla di bit, o *Binary Digit*).

Classificazione delle reti logiche

Possiamo classificare le reti secondo due criteri:

- **Assenza o presenza di memoria**
  - **Rete combinatoria**: rete dove lo stato di uscita dipende esclusivamente dallo stato di ingresso (ovviamente ricordiamo che siamo in un modello astratto, lo stato uscita in realtà dipenderebbe anche dagli elementi fisici presenti). Si dice rete senza memoria ed è associabile a una funzione matematica $F : X \to Y$ dove X consiste nell'insieme dei possibili stati di ingresso ed Y nell'insieme dei possibili stati di uscita.
  - **Rete sequenziale**: rete dove lo stato di uscita non dipende solo dall'ultimo stato di ingresso ma dallo storico degli stati di ingresso precedenti. Questa rete si dice rete con memoria.
- **Temporizzazione della legge di evoluzione del tempo**
  - **Rete asincrona**: rete dove lo stato di uscita è aggiornato costantemente
  - **Rete sincronizzata**: rete dove lo stato di uscita è aggiornato periodicamente, cioè in istanti separati nel tempo

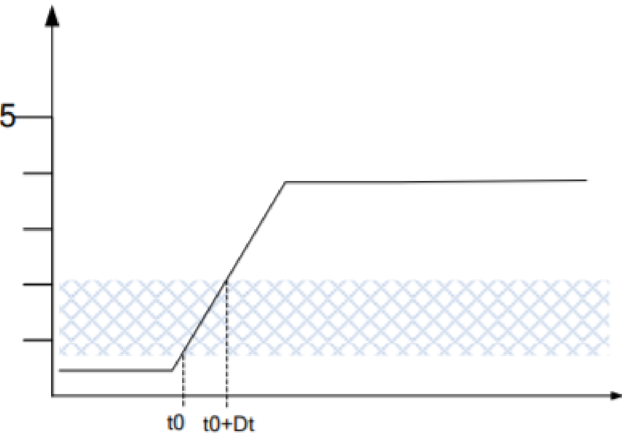
Transizione dei segnali



Supponiamo di avere una rete logica con un'unica variabili di ingresso (che da sola costituisce lo stato di ingresso). Sappiamo che questa andrà ad influire sullo stato di uscita (di cui non ci interessa il numero di variabili). Affermiamo che:

- All'istante temporale t_0 la variabile logica viene settata (si dice che *transisce*) a 1
- All'istante temporale t_1 la variabile logica viene resettata ponendo come valore 0.

In un modello astratto è facile dire che la variabile logica viene modificata in modo istantaneo. Tuttavia non possiamo dire la stessa cosa nella realtà: non possiamo avere un cambio di tensione o di corrente istantaneo (a meno che non abbia potenza infinita). Si individua che la grandezza fisica che determina la variabile logica si troverà, in un istante di tempo Δt che ha inizio nell'istante t_0, in una **fascia di indeterminazione**: non sappiamo se la rete logica, con certe grandezze fisiche, interpreterà la variabile logica come 0 o come 1.



Il modello è sempre valido? Il modello astratto adottato rimane ottimo per i calcoli e le descrizioni che dobbiamo fare se poniamo un'ulteriore ipotesi:

L'intervallo Δt è molto piccolo rispetto all'intervallo ΔT

dove ΔT consiste nell'intervallo di tempo in cui la variabile **resta a regime**. Se questo è vero stiamo guardando le variabili logiche su scale temporali molto più larghe rispetto a quelle in cui avvengono queste variazioni. Quindi, se $\Delta t << \Delta T$ potremo dire che all'istante t_0 la variabile si setta e all'istante t_1 la variabile logica si resetta.

Ma se avessi più variabili di ingresso?

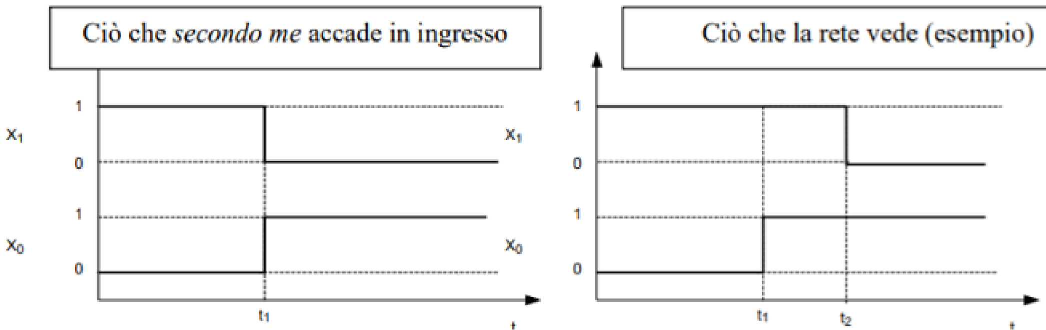
Con più variabili di ingresso la cosa diventa un po' più problematica. Abbiamo detto che ogni variabile logica dipende da una determinata grandezza fisica. Supponiamo, avendo due variabili di ingresso, che

- prima dell'istante t_1 lo stato di ingresso sia $(1, 0)$, e che
- dopo l'istante t_1 lo stato di ingresso diventi $(0, 1)$

Facile dire che entrambe le variabili logiche cambiano in modo istantaneo, impossibile garantire che le due grandezze fisiche varino contemporaneamente. Il problema grosso è che la rete si accorgerà in momenti diversi delle variazioni delle due variabili logiche. Segue che non dovremo occuparci solo dello stato di ingresso e dello stato di uscita, ma anche degli stati intermedi che vengono assunti nella fascia di indeterminazione.

Stati intermedi possibili In questo esempio gli stati intermedi possibili sono due

- $(1, 1)$, la rete percepisce la variazione della sola variabile x_1
- $(0, 0)$, la rete percepisce la variazione della sola variabile x_0



Tipi di reti

- Se la rete è di tipo sincronizzata non ci sono grossi problemi (ovviamente i tempi devono essere impostati in un certo modo)
- Se la rete è di tipo asincrono la cosa si fa più tosta: dobbiamo tenere conto degli stati di uscita derivanti dagli stati di ingresso intermedi. Gli stati di ingresso sono vincolati a cambiare un bit alla volta, segue che gli stati di ingresso consecutivi sono adiacenti (cioè differiscono per un solo bit).

Altro accorgimento che dobbiamo avere, in entrambe le reti, è il non variare gli ingressi più volte in modo molto ravvicinato.

19.2 Reti combinatorie

Le reti che descriveremo presentano le seguenti proprietà:

- lo stato di uscita è aggiornato continuamente (si ha una *rete asincrona*)
- lo stato di uscita dipende esclusivamente dallo stato di ingresso presente in quel momento (*rete combinatoria*)

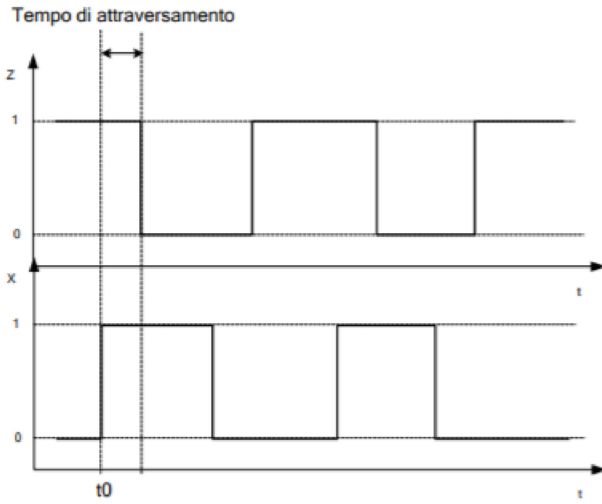
Una rete combinatoria presenta

- N variabili di ingresso
- M variabili di uscita
- una descrizione funzionale del tipo $F : X \to Z$ dove X consiste nell'insieme dei possibili stati di ingresso e Z nell'insieme dei possibili stati di uscita. Si spiega il comportamento della rete (cosa su cui perderemo molto tempo)
- una legge di evoluzione nel tempo che afferma di impostare continuamente, dato lo stato di ingresso X, lo stato di uscita $F(X)$.

Distinzione

Una legge di evoluzione nel tempo non descrive l'evoluzione delle uscite in funzione degli ingressi ma COME AVVIENE questa evoluzione. Del legame tra stati di ingresso e stati di uscita se ne occupa la descrizione funzionale.

Tempo di attraversamento (o tempo di accesso, o tempo di risposta)



Con tempo di attraversamento intendiamo il tempo necessario alla rete affinchè lo stato di uscita si adegui alle modifiche introdotte nello stato di ingresso. Si dice che una rete vada **a regime** dopo questo tempo di attraversamento. Dire che la rete va a regime significa dire che questa è pronta per recepire nuove modifiche allo stato di ingresso.

Rete pilotata in modo fondamentale La rete si dice pilotata in modo fondamentale quando si evita di variare lo stato di ingresso più velocemente del tempo di risposta della rete. Abbiamo già detto che fare questo può provocare effetti non previsti.

Descrizione di una rete combinatoria

Una rete combinatoria può essere descritta:

- a parole
- con una notazione testuale (precisamente il linguaggio Verilog, di cui parleremo più avanti)
- con le tavole di verità

Una tavola di verità consiste in una tabella avente a sinistra tutte i possibili stati di ingresso e a destra i corrispondenti stati di uscita.

- Se una variabile di uscita è uguale ad 1 si dice che riconosce certi stati di ingresso
- In alcuni casi, quando non ci interessa la variabile di uscita in corrispondenza di un particolare stato di ingresso, possiamo porre un trattino. Questo, che non rappresenta un valore

logico, significa **non specificato**. Sappiamo che assumerà come valore 0 o 1, ma non mi interessa sapere quale dei due valori assuma. La cosa può essere utile per semplificare la **sintetizzazione della rete**.

Descrizione e sintesi

- La descrizione di una rete è un modo formare per dire quale sia il suo comportamento osservabile (un esempio è la tavola di verità)
- La sintesi consiste nel progetto della realizzazione fisica di una rete.

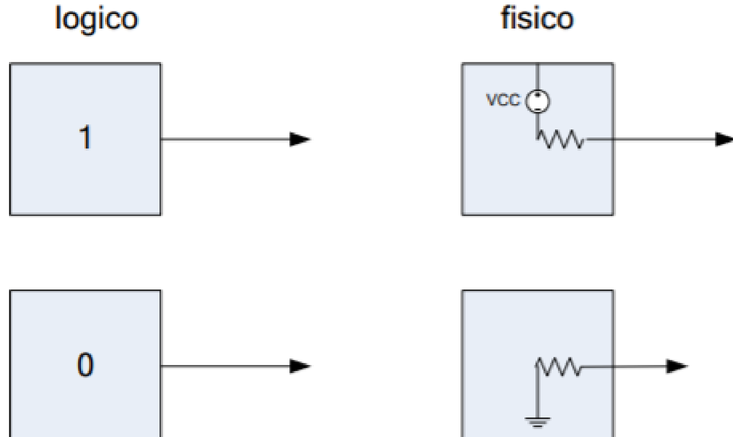
Reti combinatorie elementari

Proprietà base

Una rete combinatoria ad N ingressi ed M uscite può essere realizzata interconnettendo M reti combinatorie ad N ingressi ed un'uscita.

Questa proprietà ci permette di descrivere una rete combinatoria complessa come un insieme di reti combinatorie più semplici. Il grafico presente in dispensa non è detto sia il miglior modo per realizzare una rete ad N ingressi ed M uscite, però è uno dei metodi possibili.

Reti a zero ingressi



I generatori a zero ingressi sono **generatori di costante**, un caso degenere. Distinguiamo due reti possibili:

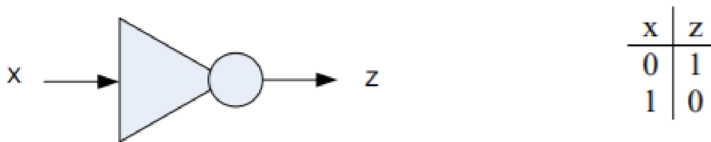
- una rete che restituisce 1 (abbiamo un filo connesso a una tensione di riferimento)
- una rete che restituisce 0 (abbiamo cun filo connesso ad una massa)

Reti ad un ingresso

Abbiamo due reti possibili:

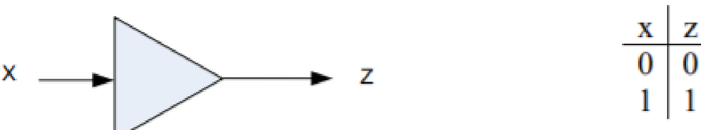
- l'invertitore: pongo come valore in ingresso 0 (1) e ottengo 1 (0)
- un elemento neutro (o *buffer*): pongo come valore in ingresso 0 (1) e ottengo 0 (1). A cosa serve un buffer?
  - A perdere tempo (si pongono queste reti in cascata)
  - Per rigenerare i segnali elettrici degradati. Ricordiamo che la porta non ha solo un ingresso e un'uscita ma è collegata a due poli (si ha tensione massa e tensione di riferimento Vcc). Rigenerare segnali degradati significa
    * ottenere uno 0 bello (vicino al fondo scala) dato uno 0 cattivo (uno 0 vicino alla fascia di indeterminazione)
    * ottenere un 1 bello (vicino al fondo scala) dato un 1 cattivo (un 1 vicino alla fascia di indeterminazione).

Invertitore



x	z
0	1
1	0

Elemento neutro



x	z
0	0
1	1

Reti a due ingressi e reti a più ingressi

Numero di possibili tavole di verità Quante tabelle di verità possiamo costruire?

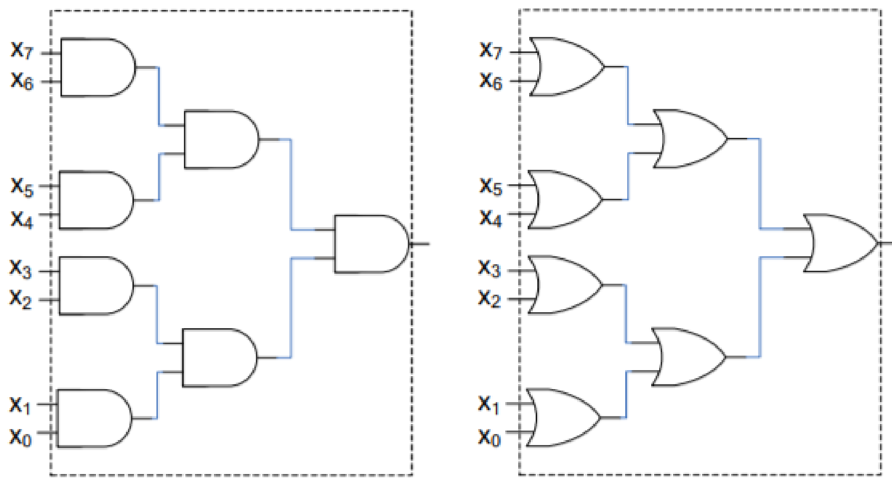
- Sappiamo che con N variabili è possibile ottenere 2^N possibili stati di ingresso
- Ogni tavola di verità presenta 2^N righe.
- Ogni stato di ingresso è associato a una singola variabile logica che ha due valori possibili.
- Il numero di righe di ogni tavola di verità possibile è lo stesso (abbiamo gli stessi stati di ingressi)
- Segue che il numero di tavole di verità possibili consiste nel numero di possibili insiemi di stati di uscita. Ciascun insieme presenta 2^N stati di uscita al suo interno, segue

$$\text{Numero di tavole di verità: } 2^{2^N}$$

Nella tabella della pagina successiva sono poste tutte le porte elementari. Inoltre abbiamo

- due generatori di costanti (casi degeneri)
- un caso semidegenere in cui esprimo come stato di uscita il valore della prima variabile
- un caso semidegenere in cui esprimo come stato di uscita il valore della seconda variabile
- un caso semidegenere in cui esprimo come stato di uscita il valore negato della prima variabile
- un caso semidegenere in cui esprimo come stato di uscita il valore negato della seconda variabile

AND e OR con più di due ingressi

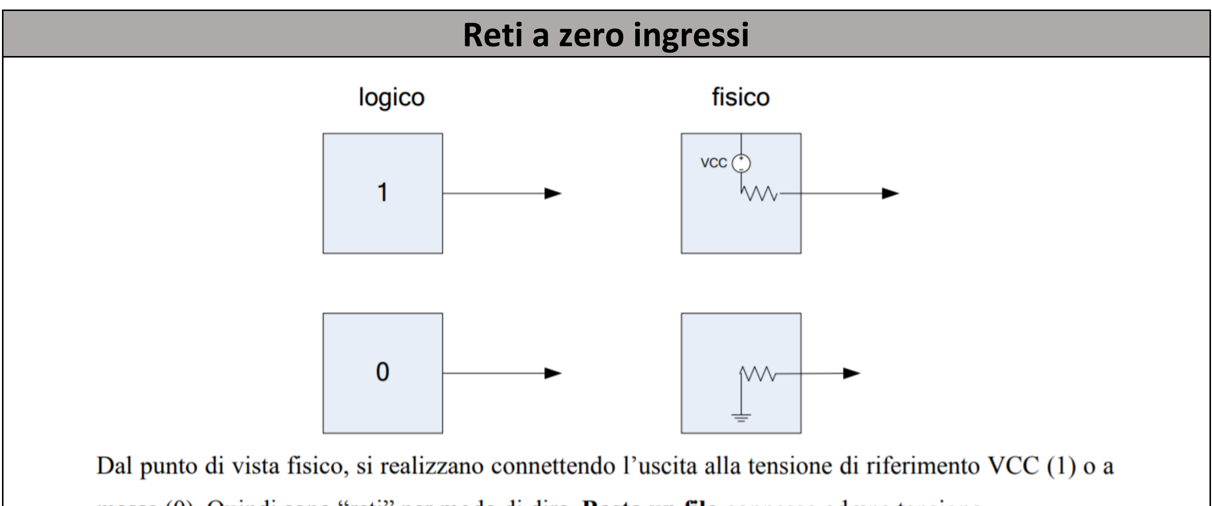


- La porta AND, con più ingressi, restituisce 1 se tutte le variabili di ingresso hanno come valore 1
- La porta OR, con più ingressi, restituisce 1 se almeno una delle variabili di ingresso ha come valore 1

Possiamo rappresentare una porta AND o una porta OR come un'interconnessione di porte AND o OR a due variabili di ingresso. Queste devono essere disposte in modo tale da ottenere una sorta di albero bilanciato: se si riducono i livelli logici si ottiene un tempo di attraversamento minore.

Reti elementari a due ingressi[1]			
Porta		**Valori**	**Descrizione**
	AND	$z = 1 \Leftrightarrow x_0 = x_1 = 1$	L’uscita vale 1 solo se entrambi gli ingressi valgono 1.
	NAND[2]	$z = 0 \Leftrightarrow x_0 = x_1 = 1$	L’uscita vale 0 solo se entrambi gli ingressi valgono 1.
	XOR[3]	$z = 1 \Leftrightarrow x_0 \neq x_1$	L’uscita vale 1 se gli ingressi sono diversi.
	*X**N**OR*[4]	$z = 1 \Leftrightarrow x_0 = x_1$	L’uscita vale 1 se gli ingressi sono uguali.
	OR	$z = 0 \Leftrightarrow x_0 = x_1 = 0$	L’uscita vale 1 se almeno un ingresso vale 1.
	NOR[5]	$z = 1 \Leftrightarrow x_0 = x_1 = 0$	L’uscita vale 0 se almeno un ingresso vale 1.

Reti a zero ingressi



Dal punto di vista fisico, si realizzano connettendo l’uscita alla tensione di riferimento VCC (1) o a massa (0). Quindi sono “reti” per modo di dire. **Basta un filo** connesso ad una tensione.

[1] Sono evidenziate le porte elementari per cui non è possibile ottenere una generalizzazione connettendo porte ad albero. La cosa è dimostrata negli esercizi relativi alle reti combinatorie. Attenzione alle N nei nomi: sono sinonimo di una cosa importante relativa alle leggi di corrispondenza.

[2] La legge di corrispondenza equivale a quella che si ottiene ponendo un invertitore in cascata ad una porta AND.

[3] Riconosce un numero dispari di 1

[4] La legge di corrispondenza è equivalente a quella che si ottiene inserendo un invertitore in cascata ad una porta XOR. Riconosce un numero pari di 1 (si restituisce 1 anche in loro assenza)

[5] La legge di corrispondenza equivale a quella che si ottiene ponendo un invertitore in cascata ad una porta OR.

19.3 Algebra di Boole

L'algebra di Boole è un sistema algebrico di cui abbiamo già sentito parlare in tutti i corsi specifici del primo anno (*Fondamenti di programmazione*, *Algoritmi e strutture dati*, *Basi di dati*). Abbiamo:

- Variabili logiche che assumono come valori 0 ed 1
- Operatori logici che si applicano alle variabili logiche e che restituiscono 0 ed 1

Complemento

Operatore unario, rappresentato come $\overline{x}$ (con codifica ASCII anche come $!x$ o $/x$). Otteniamo

$$\boxed{\overline{0} = 1} \qquad \boxed{\overline{1} = 0}$$

Prodotto logico

Operatore binario, rappresentato col simbolo tipico della moltiplicazione. Otteniamo

- $0 \cdot 0 = 0$
- $0 \cdot 1 = 0$
- $1 \cdot 0 = 0$
- $1 \cdot 1 = 1$

Somma logica

Operatore binario, rappresentato col simbolo tipico dell'addizione. Otteniamo

- $0 + 0 = 0$
- $0 + 1 = 1$
- $1 + 0 = 1$
- $1 + 1 = 1$

Confronto con le reti elementari (porte) appena introdotte

- La **porta NOT** realizza l'operazione di **complemento**
- La **porta OR** realizza l'operazione di **somma logica**
- La **porta AND** realizza l'operazione di **prodotto logico**

Capitolo 20

Martedì 13/10/2020

20.1 Continuiamo con l'Algebra di Boole

Le proprietà che elencheremo sono utili per la manipolazione di espressioni combinatorie[1]. Sono estremamente importanti in quanto alla base di molti futuri ragionamenti. Alcune proprietà sono immediate, altre necessitano di alcune riflessioni.

Perchè la pagina vuota? Per avere tutte le proprietà raccolte in un'unica pagina.

[1] Al docente non interessa il nome della proprietà, ma piuttosto la capacità di applicarla quando utile.

20.1.1 Proprietà degli operatori di somma e prodotto

- **Proprietà involutiva del complemento**: il complemento del complemento di x è x

$$\overline{\overline{x}} = x$$

- **Proprietà commutativa della somma e del prodotto**:

$$x + y = y + x \qquad x \cdot y = y \cdot x$$

anche questa è una proprietà ovvia. Ci permette di dire che le funzioni AND/OR sono cumulative (possiamo spostare ingressi liberamente).

- **Proprietà associativa della somma e del prodotto**:

$$x + y + z = (x + y) + z = x + (y + z) \qquad x \cdot y \cdot z = (x \cdot y) \cdot z = x \cdot (y \cdot z)$$

- **Proprietà distributiva della somma e del prodotto**:

$$x \cdot (y + z) = (x \cdot y) + (x \cdot z) \qquad \boxed{\boxed{x + (y \cdot z) = (x + y) \cdot (x + z)}}$$

la prima applicazione è ovvia, la seconda assolutamente no. Per avere le idee più chiare conviene disegnarci una tavola di verità: se le colonne coincideranno allora la proprietà risulterà verificata. Individuiamo che

x	y	z	y*z	x+(y*z)	(x+y)	(x+z)	(x+y)*(x+z)
0	0	0	0	**0**	0	0	**0**
0	0	1	0	**0**	0	1	**0**
0	1	0	0	**0**	1	0	**0**
0	1	1	1	**1**	1	1	**1**
1	0	0	0	**1**	1	1	**1**
1	0	1	0	**1**	1	1	**1**
1	1	0	0	**1**	1	1	**1**
1	1	1	1	**1**	1	1	**1**

le colonne sono uguali, quindi la proprietà introdotta risulta essere valida.

- **Somme**:
  - $x + 0 = x$, lo 0 è l'elemento neutro dell'addizione (anche nell'algbera tradizionale)
  - $x + 1 = 1$, 1 è l'elemento assorbente[2] dell'addizione (NON nell'algebra tradizionale)
  - $x + x = x$ (idempotenza)
  - $x + \overline{x} = 1$ (complementazione)
- **Prodotti**:
  - $x \cdot 1 = x$, 1 è l'elemento neutro della moltiplicazione (anche nell'algebra tradizionale)
  - $x \cdot 0 = 0$, lo 0 è l'elemento assorbente della moltiplicazione (anche nell'algebra tradizionale)
  - $x \cdot x = x$ (idempotenza)
  - $x \cdot \overline{x} = 0$ (complementazione)
- **Teoremi di De Morgan**:
  - $\overline{x \cdot y} = \overline{x} + \overline{y}$
  - $\overline{x + y} = \overline{x} \cdot \overline{y}$

  la cosa è verificabile usando le tavole di verità. Più avanti è presente la dimostrazione formale dei teoremi.

[2] Novità

20.1.2 Porte XOR e XNOR

- **Proprietà commutativa**: applicabile in ogni caso (porte AND, OR, XOR, XNOR)
- **Proprietà associativa**: applicabile in ogni caso (porte AND, OR, XOR, XNOR)
- **Proprietà distributiva del prodotto rispetto allo XOR**: attenzione, la distributiva contro-intuitiva vista prima non può essere applicata all'operator XOR

$$x \cdot (y \oplus z) = (x \cdot y) \oplus (x \cdot z) \qquad \boxed{\boxed{x \oplus (y \cdot z) \neq (x \oplus y) \cdot (x \oplus z)}}$$

- **Complementazione**:

$$x \oplus \overline{x} = 1 \qquad x \overline{\oplus} \overline{x} = 0$$

teniamo conto che una porta XOR a due ingressi restituisce 1 se gli ingressi sono diversi, mentre una porta XNOR restituisce 0. Applicare la proprietà di complementazione significa prendere due operandi: un valore e il suo opposto! Il risultato di OR e XOR è lo stesso.

- **Elemento neutro**: in entrambi i casi l'altro elemento determina il risultato finale

$$x \oplus 0 = x \qquad x \overline{\oplus} 1 = x$$

  - $1 \oplus 0 = 1$, $0 \oplus 0 = 0$ (ricordarsi che la porta XOR restituisce 1 se gli ingressi sono diversi)
  - $1 \overline{\oplus} 1 = 1$, $0 \overline{\oplus} 1 = 0$ (ricordarsi che la porta XNOR restituisce 1 solo se gli ingressi sono uguali)

si tenga conto che 0 è l'elemento neutro dell'operatore OR, e lo è anche dell'operator XOR. Lo XNOR, che è l'opposto dello XOR, ha come elemento neutro 1.

- **Elemento non neutro**:

$$x \oplus 1 = \overline{x} \qquad x \overline{\oplus} 0 = \overline{x}$$

gli elementi non neutri, negli operatori XOR e XNOR, restituiscono l'opposto dell'altro ingresso.

  - $0 \oplus 1 = 1$, $1 \oplus 1 = 0$ (ricordarsi che la porta XOR restituisce 1 se gli ingressi sono diversi)
  - $1 \overline{\oplus} 0 = 0$, $0 \overline{\oplus} 0 = 1$ (ricordarsi che la porta XNOR restituisce 1 solo se gli ingressi sono uguali)

Per questa proprietà basta prendere gli elementi non neutri.

- **Somma di operandi uguali (non ho idempotenza)**:

$$x \oplus x = 0 \quad x \overline{\oplus} x = 1$$

solita proprietà tipiche di porte XOR e XNOR. Non ho l'idempotenza, quindi porre due operandi uguali non mi restituisce lo stesso operando (come nelle porte AND, OR).

- **Autodualità**:

$$x \oplus x = \overline{x} \oplus \overline{x} \qquad x \overline{\oplus} x = \overline{x} \overline{\oplus} \overline{x}$$

sappiamo che la porta XOR restituisce 1 se gli operandi sono diversi, mentre la porta XNOR restituisce 1 se gli operandi sono uguali.

  - Se io complemento due operandi diversi otterrò sempre due operandi diversi.
  - Se io complemento due operandi uguali otterrò sempre due operandi uguali.

Segue che il risultato delle porte XOR e XNOR sarà sempre lo stesso.

20.1.3 Teoremi di De Morgan con N variabili logiche

Vogliamo dimostrare che i teoremi di De Morgan sono validi con un qualunque numero di variabili logiche, precisamente

1. $\overline{x_0 \cdot x_1 \cdot \dots \cdot x_{N-1}} = \overline{x_0} + \overline{x_1} + \dots + \overline{x_{N-1}}$
2. $\overline{x_0 + x_1 + \dots + x_{N-1}} = \overline{x_0} \cdot \overline{x_1} \cdot \dots \cdot \overline{x_{N-1}}$

dimostreremo entrambi i teoremi per via induttiva.

Recap molto veloce

- Prima dimostro la tesi relativa all'equivalenza tra prodotto negato e somma di negati.
- Faccio questo in modo induttivo partendo da un caso con solo due variabili logiche. Dopo la dimostrazione ho ottenuto la mia ipotesi induttiva da utilizzare nel passo induttivo.
- La dimostrazione della tesi relativa all'equivalenza tra somma negata e prodotto di negati si ottiene complementando entrambi i membri o con un procedimento simile a quello adottato per la prima tesi. In entrambi i casi ho bisogno della prima tesi del teorema di De Morgan.

20.1.3.1 Dimostrazione della prima tesi

- Abbiamo dimostrato con la tavola di verità che la prima tesi è valida con $N = 2$ variabili logiche.

x_1	x_0	$x_0 \cdot x_1$	$\overline{x_0 \cdot x_1}$	$\overline{x_0} + \overline{x_1}$
0	0	0	1	1
0	1	0	1	1
1	0	0	1	1
1	1	1	0	0

- Il nostro interesse, adesso, è di verificare la validità della tesi con $N + 1$ variabili logiche:
  - **Ipotesi induttiva**: $\overline{x_0 \cdot x_1 \cdot \dots \cdot x_{N-1}} = \overline{x_0} + \overline{x_1} + \dots + \overline{x_{N-1}}$
  - **Tesi**: $\overline{x_0 \cdot x_1 \cdot \dots \cdot x_N} = \overline{x_0} + \overline{x_1} + \dots + \overline{x_N}$
  - Poniamo $x_0 \cdot x_1 \cdot \dots \cdot x_{N-1} = \alpha$, segue

$$\overline{x_0 \cdot x_1 \cdot \dots \cdot x_N} = \overline{\alpha \cdot x_N} = \overline{\alpha} + \overline{x_N}$$

dall'ipotesi induttiva troviamo che

$$\overline{\alpha} + \overline{x_N} = \overline{x_0} + \overline{x_1} + \dots + \overline{x_{N-1}} + \overline{x_N}$$

cioè la tesi.

20.1.3.2 Dimostrazione della seconda tesi

- Abbiamo dimostrato con la tavola di verità che la seconda tesi è valida con $N = 2$ variabili logiche.
- Il nostro interesse, adesso, è di verificare la validità della tesi con $N + 1$ variabili logiche:
  - **Ipotesi induttiva**: $\overline{x_0 + x_1 + \cdots + x_{N-1}} = \overline{x_0} \cdot \overline{x_1} \cdot \cdots \cdot \overline{x_{N-1}}$
  - **Tesi**: $\overline{x_0 + x_1 + \cdots + x_N} = \overline{x_0} \cdot \overline{x_1} \cdot \cdots \cdot \overline{x_N}$
  - Poniamo $x_0 + x_1 + \cdots + x_{N-1} = \beta$, segue

    $$\overline{x_0 + x_1 + \cdots + x_N} = \overline{\beta + x_N} = \overline{\beta} \cdot \overline{x_N}$$

    dall'ipotesi induttiva troviamo che

    $$\overline{\beta} \cdot \overline{x_N} = \overline{x_0 + x_1 + \cdots + x_{N-1}} \cdot \overline{x_N} = \overline{x_0} \cdot \overline{x_1} \cdot \cdots \cdot \overline{x_{N-1}} \cdot \overline{x_N}$$

    cioè la tesi.
- Possibile dimostrare senza induzione. Si prende l'ipotesi induttiva, si applica la complementazione ad entrambi i membri. Dopo gli opportuni calcoli (dove utilizziamo la prima tesi già dimostrata) i membri coincideranno dimostrando la seconda tesi.

$$\begin{aligned}
\overline{x_0 + x_1 + \cdots + x_{N-1}} &= \overline{x_0} \cdot \overline{x_1} \cdot \cdots \cdot \overline{x_{N-1}} \\
\overline{\overline{x_0 + x_1 + \cdots + x_{N-1}}} &= \overline{\overline{x_0} \cdot \overline{x_1} \cdot \cdots \cdot \overline{x_{N-1}}} \\
x_0 + x_1 + \cdots + x_{N-1} &= \overline{\overline{x_0}} \cdot \overline{\overline{x_1}} \cdot \cdots \cdot \overline{\overline{x_{N-1}}} \\
x_0 + x_1 + \cdots + x_{N-1} &= x_0 + x_1 + \cdots + x_{N-1}
\end{aligned}$$

20.1.4 Equivalenza tra espressioni dell'Algebra di Boole e reti combinatorie

- Data una rete combinatoria (ovviamente complessa) posso trovare un'espressione booleana per ogni uscita
- Data un'espressione booleana posso sintetizzare una rete combinatoria la cui uscita consiste nel risultato dell'espressione.

Come nella matematica (con espressioni equivalenti) si hanno reti combinatorie logicamente equivalenti: la tavola di verità delle due reti risulta la stessa. Le uniche differenze, rilevanti, si hanno fisicamente: reti combinatorie diverse sono caratterizzate da porte diverse (anche nel numero) e disposizione degli elementi.

Attenzione alle porte Le porte logiche sono costose, introducono ritardi (quindi aumentano i tempi di risposta), disperdono potenza e si possono rompere. Il nostro obiettivo, in tutto il corso, sarà di sintetizzare (data una descrizione) la rete combinatoria di minor costo.

20.1.5 Qualche osservazione sulla risoluzione di esercizi

20.1.5.1 Dimostrazione di una identità

- L'idea iniziale che qualcuno di noi ha avuto è utilizzare le tavole di verità: è vero che dimostri l'identità, ma è anche vero che non apprendi l'utilizzo delle proprietà dell'algebra di Boole.

- Principio base: se ti stai complicando la vita probabilmente hai scelto la strada sbagliata. Non è assolutamente detto che la prima strada sia quella giusta: se non riusciamo ad andare avanti torniamo indietro e ripartiamo in un altro modo.

- Solitamente si prende il membro più complesso e si cerca di semplificarlo. In altri casi può essere necessario lavorare anche sul secondo membro (in particolare quando lavoriamo su prodotti di somme).

- Nelle somme di prodotti:

  - Prendiamo come esempio la prima identità del primo esercizio a pagina 24

    $$\boxed{x_1 \cdot x_2 + x_2 \cdot x_3 + \overline{x_1} \cdot x_3 = x_1 \cdot x_2 + \overline{x_1} \cdot x_3}$$

    * Semplifichiamo il primo membro

      $$x_1 \cdot x_2 + x_2 \cdot x_3 + \overline{x_1} \cdot x_3$$

      Il secondo termine è in parte sovrapponibile al primo, e in parte sovrapponibile al terzo. Il mio obiettivo è fare raccoglimento da entrambe le parti. Lo faccio modificando l'espressione così

      $$x_1 \cdot x_2 + (x_1 + \overline{x_1}) \cdot x_2 \cdot x_3 + \overline{x_1} \cdot x_3$$

      ho messo questa somma perchè mi serve x_1 per il primo membro ed $\overline{x_1}$ per il terzo. Inoltre $x_1 + \overline{x_1} = 1$.
    * Continuo

      $$x_1 \cdot x_2 + x_1 \cdot x_2 \cdot x_3 + \overline{x_1} \cdot x_2 \cdot x_3 + \overline{x_1} \cdot x_3$$

      raccogliamo

      $$x_1 \cdot x_2 \cdot (1 + x_3) + \overline{x_1} \cdot x_3 \cdot (x_2 + 1)$$

      poichè 1 è l'elemento assorbente otteniamo

      $$\boxed{x_1 \cdot x_2 + \overline{x_1} \cdot x_3}$$

      cioè il secondo membro.

- Nei prodotti di somme:

  - applico i teoremi di De Morgan in modo da ottenere somme di prodotti (su entrambi i membri);
  - applico le stesse cose dette prima.

- Prendiamo come esempio la seconda identità del primo esercizio a pagina 24

$$(x_1 + x_2) \cdot (x_2 + x_3) \cdot (\overline{x_1} + x_3) = (x_1 + x_2) \cdot (\overline{x_1} + x_3)$$

  * Applico i teoremi di De Morgan

$$\overline{(x_1 + x_2) \cdot (x_2 + x_3) \cdot (\overline{x_1} + x_3)} = \overline{(x_1 + x_2) \cdot (\overline{x_1} + x_3)}$$
$$\overline{x_1 + x_2} + \overline{x_2 + x_3} + \overline{\overline{x_1} + x_3} = \overline{x_1 + x_2} + \overline{\overline{x_1} + x_3}$$
$$\overline{x_1} \cdot \overline{x_2} + \overline{x_2} \cdot \overline{x_3} + x_1 \cdot \overline{x_3} = \boxed{\overline{x_1} \cdot \overline{x_2} + x_1 \cdot \overline{x_3}}$$

  * Continuo a sviluppare il primo membro

$$\overline{x_1} \cdot \overline{x_2} + \overline{x_2} \cdot \overline{x_3} + x_1 \cdot \overline{x_3}$$

Facciamo la stessa cosa fatta nel primo esercizio

$$\overline{x_1} \cdot \overline{x_2} + (x_1 + \overline{x_1}) \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot \overline{x_3}$$

ottengo

$$\overline{x_1} \cdot \overline{x_2} + x_1 \cdot \overline{x_2} \cdot \overline{x_3} + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} + x_1 \cdot \overline{x_3}$$

quindi

$$\overline{x_1} \cdot \overline{x_2} \cdot (1 + \overline{x_3}) + x_1 \cdot \overline{x_3} \cdot (1 + \overline{x_2})$$
$$\boxed{\overline{x_1} \cdot \overline{x_2} + x_1 \cdot \overline{x_3}}$$

cioè il secondo membro dell'identità dopo aver applicato i teoremi di De Morgan!

20.1.5.2 Semplificazione di espressioni

- Contrariamente all'esercizio precedente lavoriamo su un solo membro.
- Somme di prodotti:
  - cerco parti comuni nei termini della somma e raccolgo;
  - mi interessa ottenere, attraverso il raccoglimento, le espressioni $x+1$ o $x+\overline{x}$, entrambe uguali ad 1;
  - ricordarsi che $x \cdot \overline{x} = 0$ (se una cosa del genere è un termine o parte di un termine allora possiamo eliminarlo).

  - Prendiamo come esempio la prima espressione del secondo esercizio a pagina 24

$$\overline{a} \cdot \overline{b} \cdot \overline{c} \cdot \overline{d} + \overline{a} \cdot b \cdot \overline{c} \cdot \overline{d} + a \cdot \overline{b} \cdot \overline{c} \cdot \overline{d} + \overline{a} \cdot b \cdot d + b \cdot c \cdot d + a \cdot b \cdot d$$

    * Iniziamo raccogliendo

$$\overline{a} \cdot \overline{\mathbf{b}} \cdot \overline{\mathbf{c}} \cdot \overline{\mathbf{d}} + \overline{a} \cdot b \cdot \overline{c} \cdot \overline{d} + a \cdot \overline{\mathbf{b}} \cdot \overline{\mathbf{c}} \cdot \overline{\mathbf{d}} + \overline{a} \cdot b \cdot d + b \cdot c \cdot d + a \cdot b \cdot d$$
$$\boxed{\overline{a} \cdot \overline{b} \cdot \overline{c} \cdot \overline{d} + \overline{\mathbf{b}} \cdot \overline{\mathbf{c}} \cdot \overline{\mathbf{d}}} \cdot (\overline{a} + 1) + \overline{a} \cdot b \cdot \overline{c} \cdot \overline{d} + \overline{a} \cdot b \cdot d + b \cdot c \cdot d + a \cdot b \cdot d$$

L'idea intuitiva, legata all'algebra tradizionale, è rimuovere gli elementi che hanno generato il raccoglimento. Nell'algebra di Boole possiamo mantenerli, in modo tale da non perdere la possibilità di fare certi raccoglimenti. La cosa è resa possibile dalla proprietà $x + x = x$. Segue

$$\overline{a} \cdot \overline{b} \cdot \overline{c} \cdot \overline{d} + \overline{b} \cdot \overline{c} \cdot \overline{d} + \overline{a} \cdot b \cdot \overline{c} \cdot \overline{d} + \overline{a} \cdot b \cdot d + b \cdot c \cdot d + a \cdot b \cdot d$$

* Proseguiamo

$$\overline{a} \cdot \overline{c} \cdot \overline{d} \cdot (d + \overline{d}) + \overline{a} \cdot b \cdot d + \overline{b} \cdot \overline{c} \cdot \overline{d} + b \cdot c \cdot d + a \cdot b \cdot d$$

$$\overline{a} \cdot \overline{c} \cdot \overline{d} + \overline{a} \cdot \mathbf{b} \cdot \mathbf{d} + \overline{b} \cdot \overline{c} \cdot \overline{d} + b \cdot c \cdot d + a \cdot \mathbf{b} \cdot \mathbf{d}$$

$$\overline{a} \cdot \overline{c} \cdot \overline{d} + (\overline{a} + a) \cdot \mathbf{b} \cdot \mathbf{d} + \overline{b} \cdot \overline{c} \cdot \overline{d} + b \cdot c \cdot d$$

$$\overline{a} \cdot \overline{c} \cdot \overline{d} + \mathbf{b} \cdot \mathbf{d} + \overline{b} \cdot \overline{c} \cdot \overline{d} + c \cdot \mathbf{b} \cdot \mathbf{d}$$

$$\overline{a} \cdot \overline{c} \cdot \overline{d} + (c + 1) \cdot \mathbf{b} \cdot \mathbf{d} + \overline{b} \cdot \overline{c} \cdot \overline{d}$$

$$\boxed{\overline{a} \cdot \overline{c} \cdot \overline{d} + b \cdot d + \overline{b} \cdot \overline{c} \cdot \overline{d}}$$

A questo punto non possiamo più semplificare: è possibile un raccoglimento, ma non segue una semplificazione.

- Recap:
  1. Individuo il massimo numero di termini tra i vari prodotti.
  2. Osservo, nella somma, i prodotti che presentano quel numero di termini: individuo i raccoglimenti possibili.
  3. Per svolgere tutti i raccoglimenti utilizzo la proprietà $x + x = x$, cioè mantengo nella formula alcuni addendi che contribuiscono a fusioni (per farne altre)
  4. Quando ho fatto tutti i raccoglimenti cancello gli addendi mantenuti allo step precedente.
  5. Ripeto gli step precedenti con prodotti che presentano un numero inferiore di termini.

- Prodotti di somme:
  - La cosa principale da fare è semplificare utilizzando la proprietà distributiva

$$\boxed{\boxed{x + (y \cdot z) = (x + y) \cdot (x + z)}}$$

questo ci permetterà di ridurre i termini.

  - Prendiamo come esempio la seconda espressione del secondo esercizio a pagina 24

$$(a+\overline{b}+c)\cdot(\overline{a}+\overline{b}+\overline{c})\cdot(\overline{a}+\overline{b}+c)\cdot(a+\overline{b}+\overline{c})\cdot(\overline{a}+b)$$

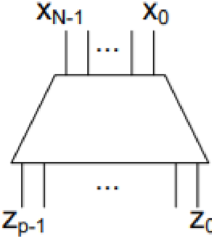
$$(a+\overline{b}+c)\cdot(\overline{a}+\overline{b}+\overline{c})\cdot(\overline{a}+\overline{b}+c)\cdot(a+\overline{b}+\overline{c})\cdot(\overline{a}+b)=$$
$$(\overline{a}+\overline{b})\cdot(a+\overline{b}+c)\cdot(a+\overline{b}+\overline{c})\cdot(\overline{a}+b)=$$
$$\overline{a}\cdot(a+\overline{b}+c)\cdot(a+\overline{b}+\overline{c})=$$
$$\overline{a}\cdot(a+\overline{b})=$$
$$\overline{a}\cdot\overline{b}$$

otteniamo molto spesso una cosa del tipo

$$x + (y \cdot \overline{y}) = (x + y) \cdot (x + \overline{y}) = x$$

20.2 Reti combinatorie significative

20.2.1 Decoder



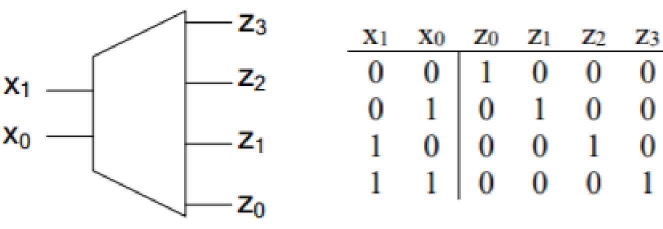
Un decoder consiste in una rete che ha

- N ingressi
- $p = 2^N$ uscite

Legge di corrispondenza Ogni uscita $j-$esima riconosce (ricordarsi quanto detto quando abbiamo parlato delle tavole di verità) un certo stato di ingresso quando i bit che costituiscono lo stato di ingresso consistono nella rappresentazione in base 2 dell'indice j.

Figura Il decoder è rappresentato da un trapezio che ha sulla base minore gli ingressi e sulla base maggiore le uscite.

20.2.1.1 Decoder 2 to 4



x1	x0	z0	z1	z2	z3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Descrizione Per la descrizione ci basta andare a vedere la tavola di verità presente a pagina 25 della dispensa sulle Reti combinatorie.

Sintetizzazione L'aspetto che ci interessa di più è la sintentizzazione: quali porte devo utilizzare per creare un decoder? Prendiamo le uscite:

$$z_3 = \begin{cases} 1 & x_1 x_0 = 11 \\ 0 & \text{Tutti gli altri casi} \end{cases} \Longrightarrow \text{Porta AND}$$

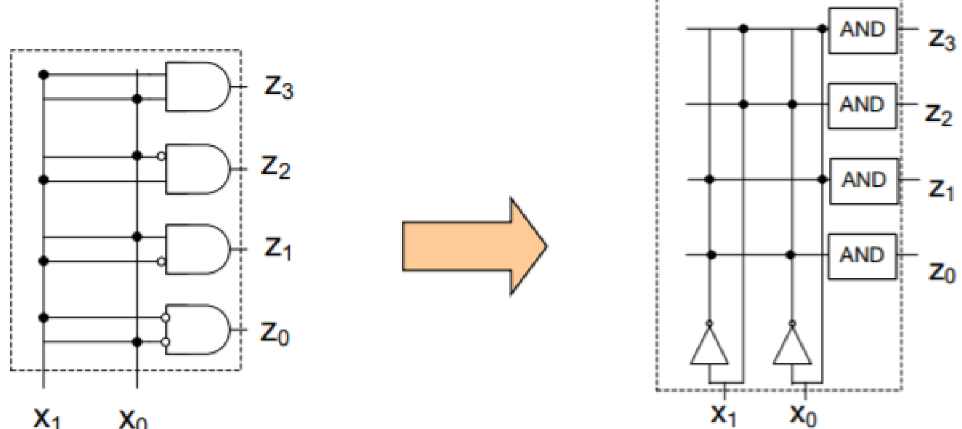
$$z_2 = \begin{cases} 1 & x_1 x_0 = 10 \\ 0 & \text{Tutti gli altri casi} \end{cases} \Longrightarrow z_2 = \begin{cases} 1 & x_1 \overline{x_0} = 11 \\ 0 & \text{Tutti gli altri casi} \end{cases} \Longrightarrow \text{Porta AND con invertitore su } x_0$$

$$z_1 = \begin{cases} 1 & x_1 x_0 = 01 \\ 0 & \text{Tutti gli altri casi} \end{cases} \Longrightarrow z_1 = \begin{cases} 1 & \overline{x_1} x_0 = 11 \\ 0 & \text{Tutti gli altri casi} \end{cases} \Longrightarrow \text{Porta AND con invertitore su } x_1$$

$$z_0 = \begin{cases} 1 & x_1 x_0 = 00 \\ 0 & \text{Tutti gli altri casi} \end{cases} \Longrightarrow z_0 = \begin{cases} 1 & \overline{x_1 x_0} = 1 \\ 0 & \text{Tutti gli altri casi} \end{cases} \Longrightarrow \text{Porta AND con invertitori su } x_0 \text{ e } x_1$$

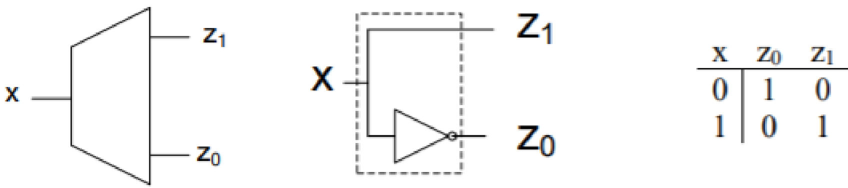
Problema Abbiamo detto che ridurre le porte è imperativo. Il numero di invertitori (uno per porta AND) è troppo. Possiamo creare una rete combinatoria dove:

- gli invertitori non sono agli ingressi delle porte AND, ma agli ingressi della rete
- per ogni variabile di ingresso si hanno due linee: una negata (dove è presente l'invertitore) e una diretta (senza invertitore).



otteniamo un numero di invertitori pari al numero di ingressi (due porte in meno).

20.2.1.2 Decoder 1 to 2



x	z_0	z_1
0	1	0
1	0	1

Questo decoder è un caso degenere di cui prendiamo semplicemente atto.

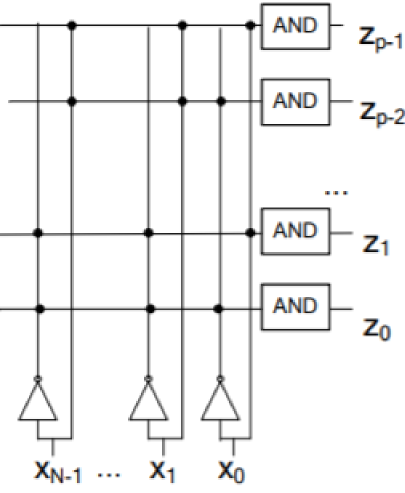
- Ho un valore in ingresso e due in uscita.
- I valori in ingresso possibili sono 0 e 1
- Abbiamo un invertitore posto sull'uscita z_0.

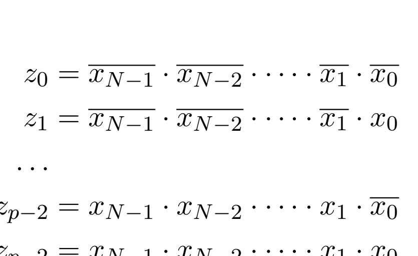
Quindi

- Se pongo 0 avrò in $z_1 = 0$ (passa diretto il valore in ingresso) e $z_0 = 1$ (presente invertitore). z_0 riconosce 0
- Se pongo 1 avrò in $z_1 = 1$ (passa diretto il valore in ingresso) e $z_0 = 0$ (presente invertitore). z_1 riconsoce 1.

20.2.1.3 Decoder N to 2^N

La legge di corrispondenza, scritta per ogni uscita sotto forma di espressione booleana, è la seguente:



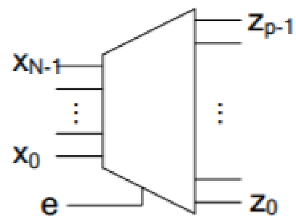


$$
\begin{aligned}
z_0 &= \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot \overline{x_0} \\
z_1 &= \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot x_0 \\
&\dots \\
z_{p-2} &= x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot \overline{x_0}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

La sintesi richiede tante porte AND quante sono le uscite (2^N) e tanti invertitori quanti sono gli ingressi (N).

20.2.2 Decoder con enabler (espandibile)

I decoder descritti fino ad ora non sono espandibili, cioè non è possibile costruire decoder grandi combinando decoder più piccoli. La cosa può essere risolta introducendo una nuova variabile logica in ingresso: l'enabler e.



Se l'enabler è 0 tutti i valori in uscita saranno uguale a 0, se 1 la rete si comporta come un decoder avente N ingressi e 2^N uscite. Quindi, un decoder con enabler ha

- $N + 1$ ingressi
- $p = 2^N$ uscite

Espressione booleana Possiamo immaginarci la seguente cosa

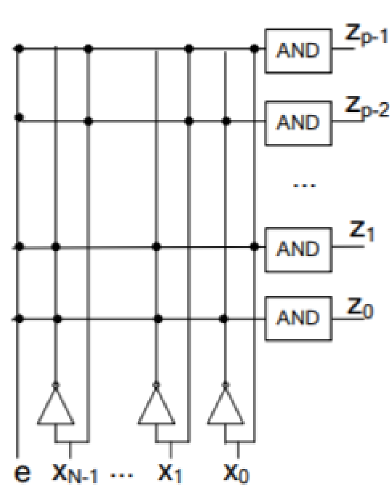
$$z_i = \begin{cases} y_i & e = 1 \\ 0 & e = 0 \end{cases}$$

cioè $z_i = e \cdot y_i$.

Sintetizzazione Possiamo ottenere quanto descritto attraverso una struttura simile a un decoder, aggiungendo una porta AND in prossimità di ogni uscita. I valori in ingresso, per ciascuna porta AND, sono uno dei valori in uscita (del decoder senza enabler) e l'enabler e.

Non stiamo aggiungendo troppe porte? Sappiamo dalle proprietà dell'algebra di Boole che porte AND/OR sono cumulabili: posso quindi togliere queste porte e porre la variabile enabler con le altre variabili in ingresso.

Legge di corrispondenza



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

20.2.3 Costruzione di un decoder 4 to 16 da decoder 2 to 4

Supponiamo di voler costruire un decoder avente 4 ingressi e $2^4 = 16$ uscite. Lo andremo a creare unendo decoder aventi 2 ingressi e $2^2 = 4$ uscite.

Tavola di verità

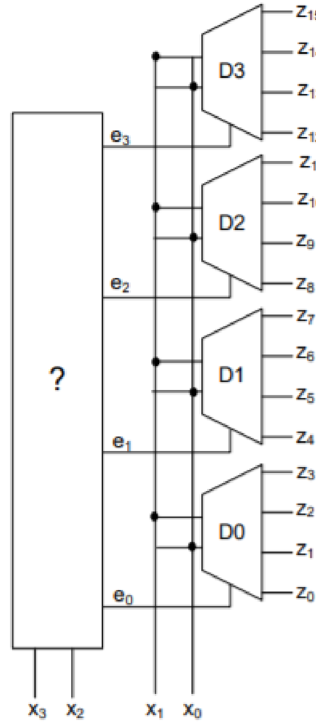
<table>
<tr><th>x3</th><th>x2</th><th>x1</th><th>x0</th><th>z0</th><th>z1</th><th>z2</th><th>z3</th><th>z4</th><th>z5</th><th>z6</th><th>z7</th><th>z8</th><th>z9</th><th>z10</th><th>z11</th><th>z12</th><th>z13</th><th>z14</th><th>z15</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td rowspan="4" colspan="4">0</td><td rowspan="4" colspan="4">0</td><td rowspan="4" colspan="4">0</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td rowspan="4" colspan="4">0</td><td>1</td><td>0</td><td>0</td><td>0</td><td rowspan="4" colspan="4">0</td><td rowspan="4" colspan="4">0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td rowspan="4" colspan="4">0</td><td rowspan="4" colspan="4">0</td><td>1</td><td>0</td><td>0</td><td>0</td><td rowspan="4" colspan="4">0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td rowspan="4" colspan="4">0</td><td rowspan="4" colspan="4">0</td><td rowspan="4" colspan="4">0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td></tr>
</table>

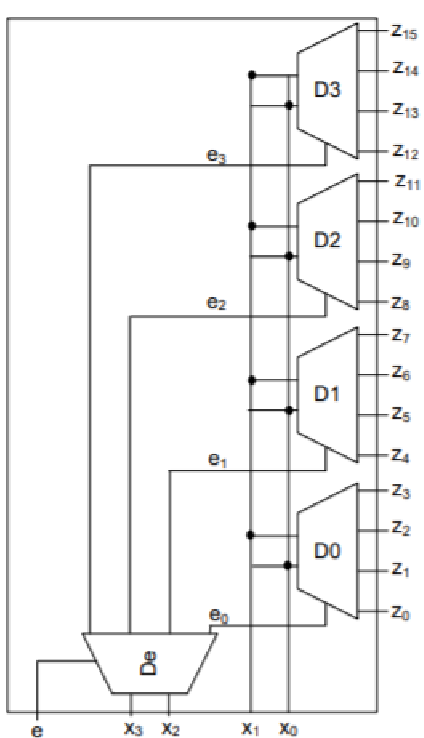
- Le colonne relative ai valori di uscita costituiscono una matrice identica. Segue che i blocchi dove non si ha parte della diagonale hanno tutti i valori uguali a 0.

- Osserviamo, all'interno di questa tabella, la tavola di verità per decoder a 2 ingressi ripetuta quattro volte (i valori in ingresso consistono nelle colonne x_1 e x_0)
- Quello che individuiamo è che per ogni caso avremo uno solo dei quattro decoder attivi (cioè il decoder $i-$esimo attivo avrà $e_i = 1$).
- Quale decoder sarà accesso dipenderà dai valori di ingresso x_2 e x_3

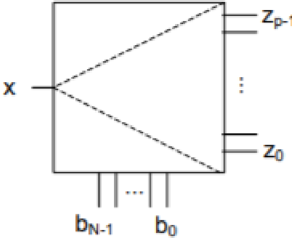
Sintetizzazione

- Ogni decoder $i-$esimo ha in ingresso x_1 e x_0, più l'enabler e_i.
- Per determinare i valori degli enabler $e_0, \ldots, e_3$ introduciamo un quinto decoder: esso avrà in ingresso x_1, x_0 ed e.
- Porre $e = 0$ significa spegnere questo quinto decoder e conseguentemente tutti gli altri $(e_0, \ldots, e_4 = 0)$: abbiamo effettivamente ottenuto un decoder con enabler a 4 ingressi.





20.2.4 Demultiplexer



Il demultiplexer è una rete avente

- N variabili di comando $b_{N-1}, \ldots, b_0$
- una variabile x detta variabile da commutare
- $p = 2^N$ uscite.

Le variabili di comando possono stare in 2^N possibili stati: selezionano una e una sola delle uscite (distribuisce il segnale in ingresso su una delle uscite, quella selezionata dalle variabili di comando) secondo la stessa descrizione funzionale dei decoder

$$(b_{N-1} \ldots b_0)_2 = j$$

il valore associato all'uscita di indice j sarà la variabile x. Prendiamo ad esempio la variabile z_0:

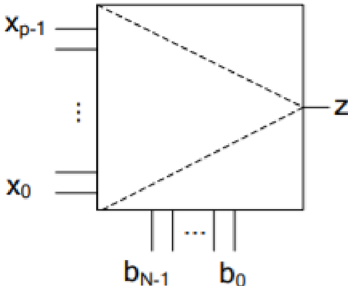
$$z_0 = \begin{cases} x & b_{N-1} \ldots b_1 b_0 = (0 \ldots 0) \\ 0 & \text{In tutti gli altri casi} \end{cases} \Longrightarrow z_0 = \begin{cases} x & \overline{b_{N-1}} \ldots \overline{b_1 b_0} = (1 \ldots 1) \\ 0 & \text{In tutti gli altri casi} \end{cases}$$

troveremo che $z_0 = x \cdot \overline{b_{N-1}} \cdot \cdots \cdot \overline{b_1} \cdot \overline{b_0}$. Otteniamo:

$$\begin{aligned} z_0 &= x \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot \cdots \cdot \overline{b_1} \cdot \overline{b_0} \\ z_1 &= x \cdot \overline{b_{N-1}} \cdot \overline{b_{N-2}} \cdot \cdots \cdot \overline{b_1} \cdot x_0 \\ &\ldots \\ z_{p-2} &= x \cdot b_{N-1} \cdot b_{N-2} \cdot \cdots \cdot b_1 \cdot \overline{b_0} \\ z_{p-2} &= x \cdot b_{N-1} \cdot b_{N-2} \cdot \cdots \cdot b_1 \cdot b_0 \end{aligned}$$

abbiamo ottenuto una cosa identica a quanto visto col decoder con enabler (abbiamo x al posto di e).

20.2.5 Multiplexer



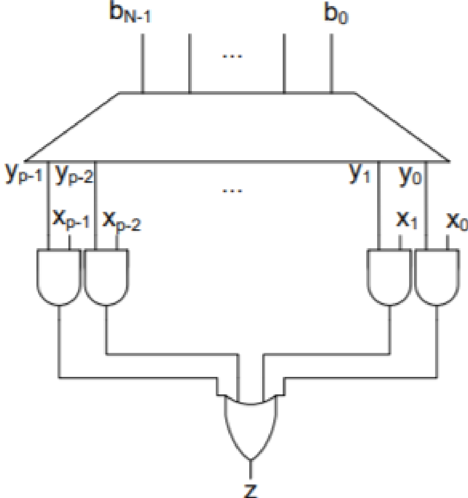
Il multiplexer è una rete avente

- 2^N ingressi $x_{p-1}, \ldots, x_0$
- N variabili di comando $b_0, \ldots, b_{N-1}$
- una sola uscita

le N variabili di ingresso possono stare in 2^N stati diversi: possiamo decidere quale dei 2^N ingressi è connesso all'uscita (cioè seleziono un singolo segnale elettrico tra quelli in ingresso in base al valore degli ingressi di controllo). Questo significa che

$$z = x_i \iff (b_{N-1} \ldots b_0)_2 = i$$

Sintetizzazione Partiamo dalla sintesi per comodità.

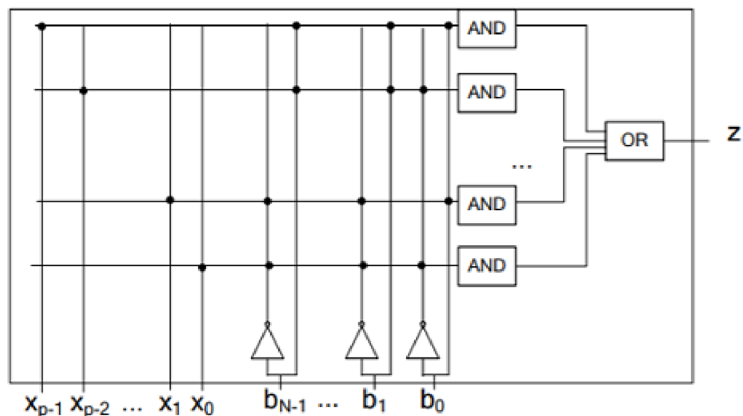


- Prendiamo un decoder con N ingressi (le variabili di pilotaggio) e $p = 2^N$ uscite (che mi permettono se un certo ingresso del multiplexer è stato selezionato). Ovviamente tutte le uscite, tranne una, risultano essere nulle.
- Ogni variabile di uscita viene messa in AND con il corrispondente ingresso del multiplexer.

- Una sola di queste porte potrà restituire un risultato diverso da zero: quella con la variabile x_j corrispondente allo stato delle variabili di comando. La porta restituirà come valore x_j.
- Le uscite ottenute dalle porte AND convergono in una porta OR. L'uscita della porta OR è l'uscita del multiplexer.
- Anche in questo caso possiamo ottenere una rete combinatoria con meno porte ponendo le variabili di ingresso $x_{p-1}, \dots, x_0$ con le variabili di comando $b_{N-1}, \dots, b_0$.

Il multiplexer è una rete a due livelli di logica: la strada più lunga tra ingresso e uscita passa attraverso due porte. Nel conto non si considerano gli invertitori sugli ingressi (motivazione sarà chiara più avanti).

Descrizione algebrica



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

20.2.6 Multiplexer come rete combinatoria universale

- I multiplexer con N variabili di comando possono realizzare qualunque legge combinatoria ad N ingressi ed un'uscita.
- Basta connettere ai 2^N ingressi dei generatori di costante.
- Una rete combinatoria a più uscite può essere scomposta in reti ad una uscita messe in parallelo (quindi più multiplexer in parallelo).

Conclusione Ogni rete combinatoria può essere costruita combinando AND, OR, NOT in al più due livelli di logica. La cosa è un'ottima notizia!

- I calcoli che dovremo fare saranno sempre semplici
- Si pone un limite al ritardo che un sistema combinatorio può avere.

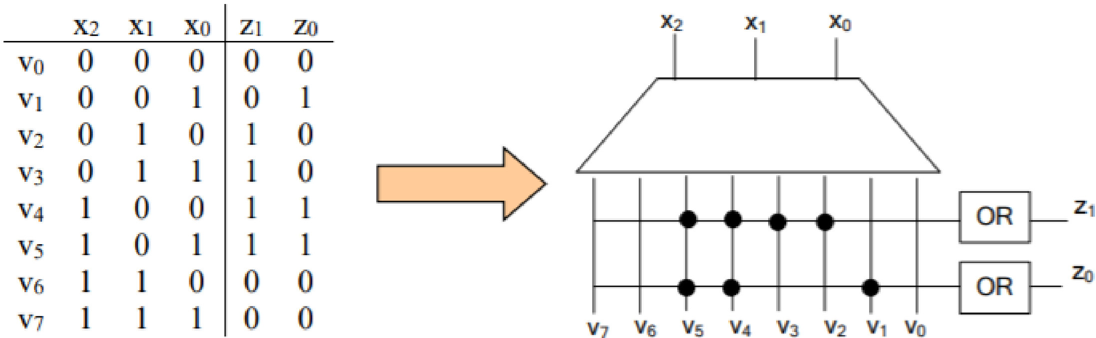
Capitolo 21

Mercoledì 14/10/2020 e Giovedì 15/10/2020

21.1 Modello strutturale universale

Abbiamo detto, attraverso i multiplexer, che ogni rete combinatoria può essere costruita combinando AND, OR, NOT in al più due livelli di logica. Possiamo sintetizzare una rete ad N ingressi ed M uscite nel seguente modo:

- un decoder con N ingressi e 2^N uscite
- M porte OR



	x_2	x_1	x_0	z_1	z_0
v_0	0	0	0	0	0
v_1	0	0	1	0	1
v_2	0	1	0	1	0
v_3	0	1	1	1	0
v_4	1	0	0	1	1
v_5	1	0	1	1	1
v_6	1	1	0	0	0
v_7	1	1	1	0	0

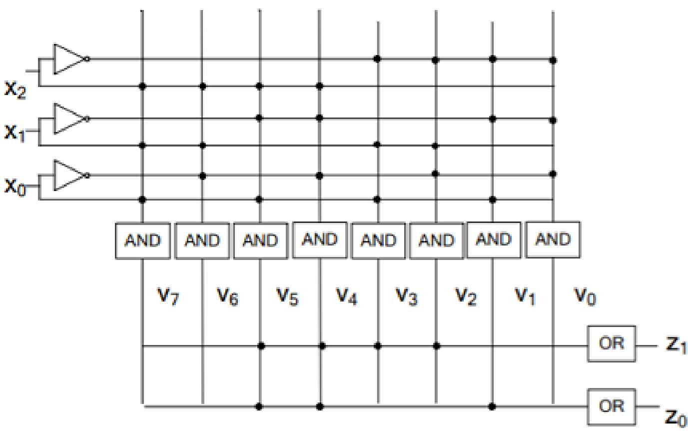
Quanto visibile nell'immagine consiste sostanzialmente in una rete con più multiplexer (uno per uscita, si faccia caso alla presenza di una porta OR per uscita). Sfruttando la tavola di verità decidiamo come collegare le 2^N uscite del decoder alle M porte OR. Per fare ciò consideriamo, per ogni variabile di uscita, quali stati di ingresso riconosce.

- L'uscita z_1 riconosce gli stati di ingresso v_2, v_3, v_4, v_5
- L'uscita z_0 riconosce gli stati di ingresso v_1, v_4, v_5

Il risultato è la rete in immagine (l'esempio a pagina 35 della dispensa). Si individua che

$$\begin{cases} z_1 = v_2 + v_3 + v_4 + v_5 \\ z_0 = v_1 + v_4 + v_5 \end{cases}$$

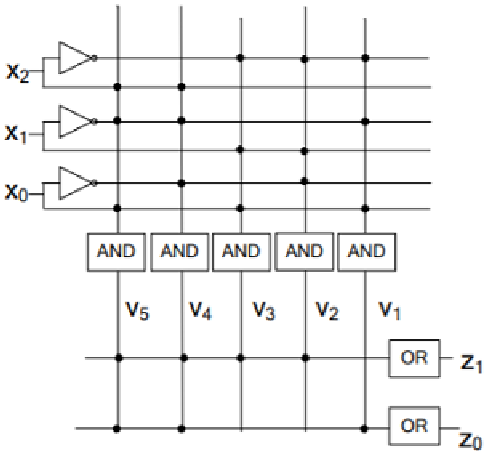
Proviamo a semplificare Spogliamo il decoder, e vediamo la sua struttura: abbiamo 2^N porte AND, una per uscita del decoder. Individuiamo che alcuni stati di ingresso non sono riconosciuti da alcuna uscita. Possiamo togliere quelle porte AND: nell'esempio siamo passati da 8 porte AND a 5 porte AND.



Semplifichiamo ulteriormente Sfruttiamo l'algebra di Boole ed effettuiamo delle sostituzioni. Sappiamo che ogni variabile v_i consiste nel risultato di una porta AND, segue (tenendo conto degli accessi diretti e negati)

$$\begin{cases} z_1 = \overline{x_2} \cdot x_1 \cdot \overline{x_0} + \overline{x_2} \cdot x_1 \cdot x_0 + x_2 \cdot \overline{x_1} \cdot \overline{x_0} + x_2 \cdot \overline{x_1} \cdot x_0 \\ z_0 = \overline{x_2} \cdot \overline{x_1} \cdot x_0 + x_2 \cdot \overline{x_1} \cdot \overline{x_0} + x_2 \cdot \overline{x_1} \cdot x_0 \end{cases}$$

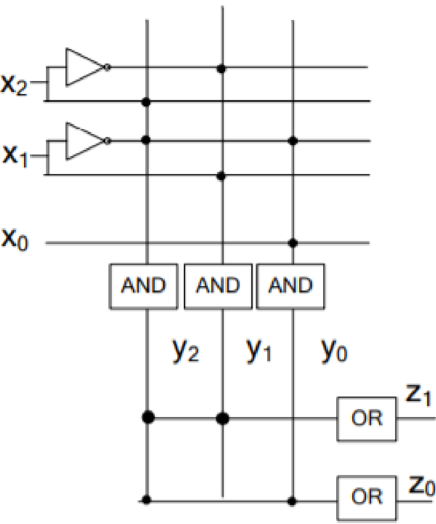
Le espressioni trovate si trovano nella cosiddetta *forma canonica SP* (dove SP sta per somma di prodotti): ogni variabile di uscira consiste nella somma del prodotto di tutte le variabili in ingresso dirette o negate.



Ancora Semplifichiamo ancora effettuando dei raccoglimenti nelle due espressioni. Sapendo che $y + \overline{y} = 1$ possiamo dire

$$\begin{cases} z_1 = \overline{x_2} \cdot x_1 + x_2 \cdot \overline{x_1} \\ z_0 = \overline{x_1} \cdot x_0 + x_2 \cdot \overline{x_1} \end{cases}$$

la forma ottenuta non è più canonica poichè si ha la somma del prodotto di qualche variabile di ingresso diretta o negata. Attraverso questi calcoli riduciamo ulteriormente il numero di porte AND (da 5 a 3).



Ancora? Si potrebbe semplificare ulteriormente, ma ciò ci porta fuori dal modello SP a due livelli di logica. Non è nostro interesse per il momento.

Metodo euristico Il metodo appena visto è euristico, cioè fatto ad occhio senza nessuna garanzia. Dedichiamoci adesso allo studio di un metodo formale.

21.2 Sintesi di reti in forma SP a costo minimo

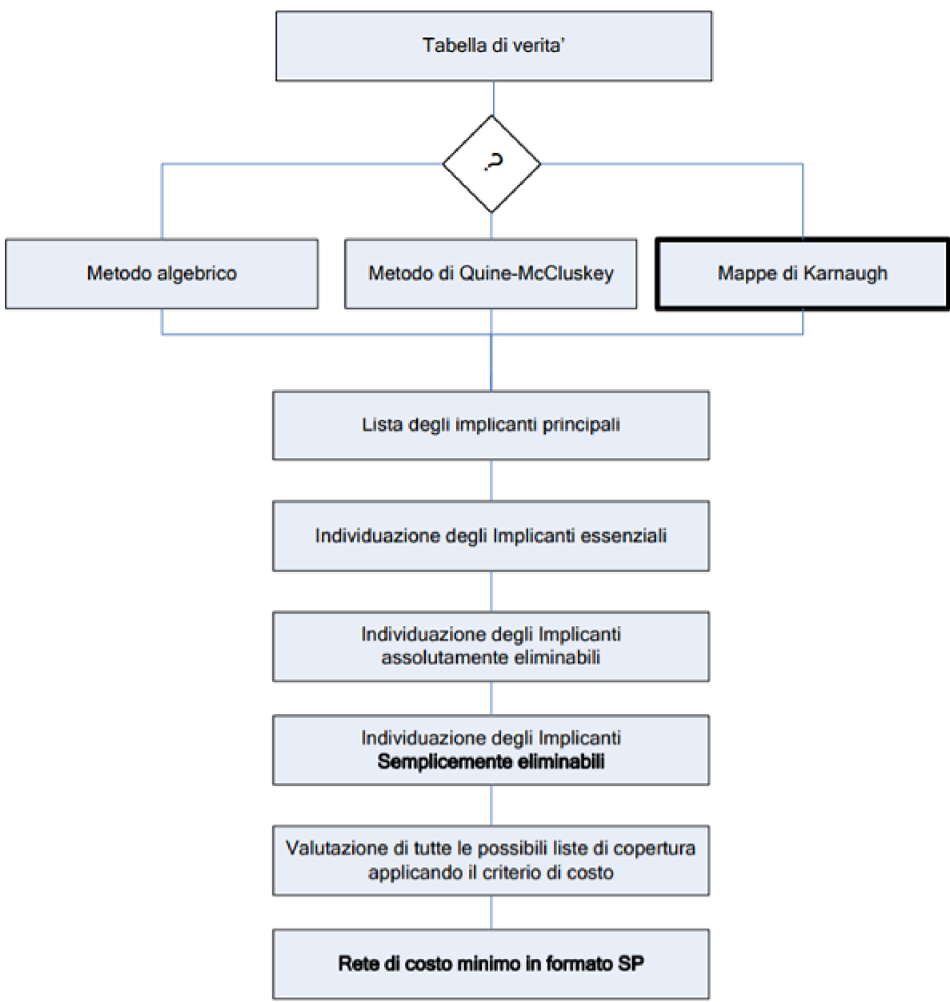
Esistono due criteri per determinare "il costo":

- criterio di costo a porte: una porta consiste in un'unità di costo
- criterio di costo a diodi: un ingresso consiste in un'unità di costo

Risulta facile capire che questi due criteri lasciano il tempo che trovano:

- se si considera il primo una porta AND a 2 ingressi e una a 10 ingressi valgono uguale
- se si considera il secondo una porta AND a 2 ingressi è meno costosa di una a 10 ingressi.

Premessa I metodi che studieremo a breve riguardano soltanto le reti in forma SP. Trovare la rete di costo minimo in assoluto è troppo complesso. In altri casi, addirittura, esistono reti a più di due livelli di logica meno costose (a discapito della velocità, ovviamente).



21.2.1 Metodo algoritmico per la lista degli implicanti principali

Il metodo che andremo a introdurre permette di ottimizzare reti **a un'uscita**. La cosa potrebbe fatta su reti a più uscite applicando il metodo ad ogni singola porta: questa cosa non è detto ci porti a una soluzione a costo minimo (l'unione di ottimi locali non è detto che restituisca un ottimo globale).

Cosa otteniamo? Una rete a 2 livelli di logica in forma SP (rete di costo minimo in questa forma)

- Sfruttiamo quanto detto da Shannon (*è sempre possibile scrivere qualunque legge F di una rete combinatoria come somma di prodotti degli ingressi diretti o negati*, cosa vista coi multiplexer) per scrivere l'omonima espansione

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

  dove i risultati di f consistono negli stati di uscita. L'espansione di Shannon è rappresentabile con un multiplexer: generatori di costante collegati agli ingressi.

- Consideriamo la nostra tabella di verità e scriviamo l'espansione di Shannon
- Dall'espansione di Shannon otteniamo la forma canonica SP considerando le seguenti regole:
  - se $f(x_{N-1},\dots,x_0) = 0$ tutta la riga vale 0 poichè $0\cdot\alpha = 0$. La riga sarà rimossa tenendo conto che $\alpha + 0 = \alpha$.
  - se $f(x_{N-1},\dots,x_0) = 1$ rimuoviamo il fattore poichè $1\cdot\alpha = \alpha$.

  nella forma canonica SP ogni termine è detto **mintermine**: esso consiste in uno stato di ingresso riconosciuto dalla rete (abbiamo rimosso tutte le righe dove $f(x_{N-1},\dots,x_0) = 0$).

- Semplifichiamo applicando sui mintermini le seguenti regole (in modo esaustivo, si va avanti finchè non avremo una situazione semplificata dove non è più possibile applicare queste regole):
  - $\alpha\mathbf{x} + \alpha\overline{\mathbf{x}} = \alpha\mathbf{x} + \alpha\overline{\mathbf{x}} + \alpha$
    con questa legge fondiamo i mintermini. Dati due termini che differiscono per una sola variabile, in un caso diretta e negata in un altro, produco un termine che contiene ciò che è comune tra i due mintermini fusi
  - $\alpha + \alpha = \alpha$
    con questa legge ribadiamo che non bisogna inserire duplicati.

  la cosa fastidiosa del metodo è che dobbiamo controllare tutte le coppie possibili: prendo il primo mintermine e lo confronto con tutti gli altri mintermini. Successivamente confronto il secondo con gli altri, e così via finchè non arriviamo all'ultimo. Nel corso del procedimento è importante non cancellare i mintermini fusi e limitarsi a segnarli (questo ci permette di

fare i confronti rimanenti). Il procedimento ci permette di passare dalla lista k_0 alla lista k_1: si applicano nuovamente le leggi se ci sono margini di semplificazione. Nell'esempio arriviamo alla lista k_2 dove sarà presente un solo mintermine.

- Scriviamo
$$z = k_0 + k_1 + k_2$$
la lista ottenuta (in forma SP non canonica) consiste nella **lista degli implicanti**. Ciascun termine è ovviamente detto implicante.

- Semplifichiamo la lista rimuovendo tutti gli implicanti che hanno prodotto qualcosa per fusione. Quanto ottenuto consiste nella **lista degli implicanti principali**.

- Introduciamo la **lista di copertura**. Essa è in forma SP non canonica e consiste in una lista di implicanti. Sono liste di copertura:
  - lista dei mintermini
  - lista degli implicanti
  - lista degli implicanti principali

  Si definsice **lista di copertura non ridondante** una lista che smette di essere lista di copertura rimuovendo un qualunque implicante.

21.2.2 Metodo di Quine-McCluskey per la lista degli implicanti principali

Questo metodo non è stato spiegato dal docente, ma può salvarti la vita.

Per utilizzare questo metodo dobbiamo avere la tavola di verità.

x_3	x_2	x_1	x_0	z_0
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

⇒

	x_3	x_2	x_1	x_0
x	1	0	0	0
x	0	0	0	1
x	0	0	1	0
x	0	0	1	1
x	1	0	0	1
x	1	0	1	0
x	0	1	1	0
x	0	1	1	1

⇒

	x_3	x_2	x_1	x_0
	1	0	0	-
	1	0	-	0
	0	0	-	1
	-	0	0	1
x	0	0	1	-
	-	0	1	0
x	0	-	1	0
x	0	-	1	1
x	0	1	1	-

⇒

x_3	x_2	x_1	x_0
0	-	1	-
~~0~~	~~-~~	~~1~~	~~-~~

- Si considerano solo gli stati riconosciuti dalla rete (cioè quelli per cui l'uscita è uguale a 1)
- Si dividono gli stati selezionati in partizioni: gli elementi di una partizione hanno in comune il numero di 1 presenti.
- All'interno di ogni partizione verifico se due coppie di implicanti possono generare fusioni: si ha una fusione se gli implicanti differiscono per una sola variabile. Scriviamo in un'ulteriore tavola di verità le fusioni ottenute (utilizzeremo il trattino $-$ per indicare l'unica variabile differente)
- Segnarsi gli implicanti che hanno generato fusione.
- Si applica il metodo esaustivamente escludendo i duplicati.
- Gli implicanti che nel procedimento non hanno generato fusioni sono quelli principali.

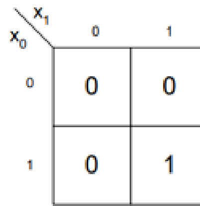
L'esercizio 6.2.8 nella dispensa di Stea sulle reti combinatorie è un esempio di applicazione di Quine-McCluskey.

Video in inglese `https://www.youtube.com/watch?v=l1jgq0R5EwQ`

21.2.3 Mappe di Karnaugh

Metodo alternativo alle tavole di verità per descrivere una rete combinatoria sono le cosiddette *mappe di karnaugh*. Come vedremo risultano vantaggiose per individuare liste di copertura non ridondanti.

Spiegazione La mappa consiste in una matrice di 2^N celle, date N variabili di ingresso. In un certo senso possiamo immaginarcele come le tabelle degli orari scolastici (quelle che dicono cosa si ha a una certa ora).



N=2. Mappa di Karnaugh per un AND.

- una cella è individuata da uno stato di ingresso, le cui variabili costituiscono le coordinate.
- una cella contiene il valore di uscita associato allo stato di ingresso definito dalle coordinate

Nelle nostre mappe adotteremo le seguenti convenzioni:

- si pongono coordinate adiacenti (in modo che queste costituiscano un'unica coordinata, non è detto si faccia sempre)
- si pone a sinistra le coordinate con indice più basso e in alto le coordinate con indice più alto.

Seguire questo standard nei compiti

x_1x_0 \ x_3x_2

00

- si dispongono i valori possibili delle coordinate in modo tale che questi siano a due a due adiacenti

x_0 \ x_2x_1	00	01	11	10
0	0	1	1	0
1	1	1	1	1

N=3. Attenzione a come scrivo le coordinate. Devo scriverle in modo che celle **contigue sulla mappa abbiano coordinate adiacenti (e viceversa).**

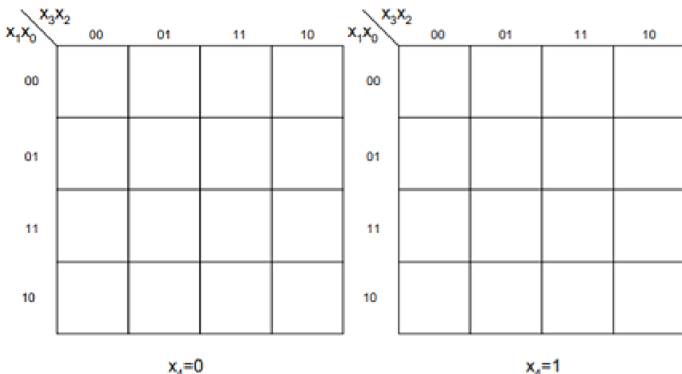
queste convenzioni sono necessarie affinchè **celle contigue sulla mappa abbiano coordinate adiacenti** (proprietà necessaria per quello che faremo più avanti). La regola è valida

- all'interno della mappa;
- sulla frontiera della mappa (*ciò che si trova all'estrema destra è contiguo a ciò che si trova all'estrema sinistra, ciò che si trova all'estremo superiore è contiguo a ciò che si trova all'estremo inferiore*, sia orizzontalmente che verticalmente). **EFFETTO PACMAN**

tutto quanto può essere immaginato come una superficie sferica.

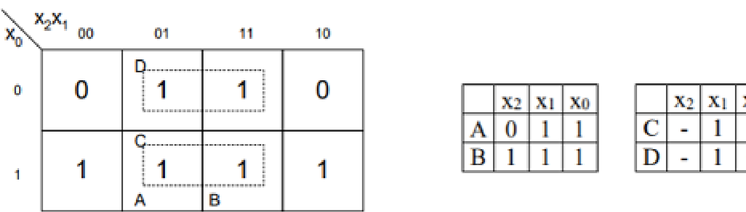
Ordine della mappa

- Generalmente le mappe di Karnaugh si utilizzano fino al quarto ordine (con ordine si intende il numero di variabili di ingresso).
- Nelle mappe di Karnaugh non è possibile mettere più di due variabili su un asse mantenendo le proprietà richieste. Risulta possibile, tuttavia, arrivare al quinto ordine introducendo la terza dimensione: oltre alle due coordinate composte abbiamo una terza variabile x_4 che può assumere come valore 0 o 1.
- Creiamo due mappe: in una abbiamo $x_4 = 0$ e nell'altra $x_4 = 1$. L'adiacenza si ha attraverso la sovrapposizione: tutte le celle presentano le stesse variabili logiche tranne x_4.



- Oltre il quinto ordine le condizioni di lavoro si complicano.

Definizioni



	x2	x1	x0
A	0	1	1
B	1	1	1

	x2	x1	x0
C	-	1	1
D	-	1	0

- Un **sottocubo di ordine 1** consiste in una cella avente per valore 1. Le coordinate di questi sottocubi sono stati di ingresso riconosciuti dalla rete (quindi un MINTERMINE).
- Due sottocubi di ordine 1 si dicono **adiacenti** se differiscono per una sola coordinata.
- Un **sottocubo di ordine 2** è costituito da due sottocubi adiacenti di ordine 1. Le coordinate si pongono nel seguente modo
  - non si specifica il valore delle variabili logiche che cambiano
  - si pongono i valori se questi sono costanti all'interno di tutto il sottocubo
- Due sottocubi di ordine 2 si dicono **adiacenti** se differiscono per una sola coordinata.

- ... *questi ragionamenti possono essere ripetuti per tutti gli ordini successivi*
- Un **sottocubo principale** è un cubo per cui non esiste un sottocubo più grande che lo copre completamente.
- Una **lista di copertura** consiste in un qualunque insieme di sottocubi tale che ogni sottocubo di ordine 1 sia coperto da almeno uno dei sottocubi appartenenti alla lista.
- Una lista di copertura è **non ridondante** se togliendo un sottocubo essa non è più una lista di copertura.

Chiusura del cerchio:

$$\boxed{\text{Sottocubo principale di ordine } p \equiv \text{Implicante principale di } N - (\log_2 p) \text{ variabili}}$$

Gli implicanti si indicano come prodotti di ingressi diretti o negati, mentre i sottocubi si indicano con coordinate. Il passaggio da l'una all'altra cosa è immediato: si guardano le coordinate e si mette una variabile diretta o negata a seconda che ci sia 1 o 0 nella coordinata.

Ricordarsi anche che (lo abbiamo già detto)...

$$\boxed{\text{Sottocubo di ordine } 1 \equiv \text{Mintermine}}$$

la cosa va ricordata in certi esercizi, in particolare in quello in cui si richiede se la sintesi trovata sia effettivamente quella a costo minimo assoluto. Ovviamente questo comporta uscire dal modello SP a costo minimo perchè si ha un qualcosa di ridondante (si lavora con implicanti non principali): in certi casi la ridondanza aiuta perchè ci porta ad avere parti del circuito da realizzare una sola volta. Se si analizza il costo mediante i criteri visti comprendiamo se quanto pensato può essere vantaggioso.

21.2.4 Algoritmo di ricerca dei sottocubi principali (implicanti principali) partendo dalle mappe di Karnaugh

- Per prima cosa considero i sottocubi di ordine più grande (qual è l'ordine in questione? lo vediamo ad occhio): li segno tutti! Troveremo sottocubi principali.
- A questo punto verifico se l'insieme dei sottocubi considerato basta a coprire tutti i sottocubi di ordine 1 della mappa.
  - Se ciò avviene abbiamo finito,
  - altrimenti dobbiamo continuare.
- Considero i sottocubi di ordine inferiore: segno tutti i sottocubi tranne quelli già interamente coperti da quelli considerati al primo step.
- Mi ripongo le solite domande di prima e valuto se ho finito o devo proseguire.
- Continuo finchè non avrò coperto tutta la mappa o quando avrò considerato i sottocubi di ordine 1 per coprire le parti rimanenti della mappa.

- A questo punto, ricordando che

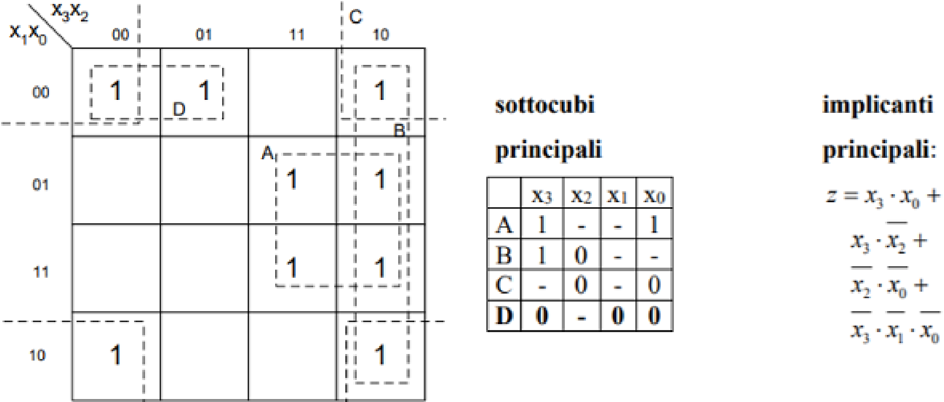
  $$\boxed{\text{Sottocubo principale di ordine } p \equiv \text{Implicante principale di } N - (\log_2 p) \text{ variabili}}$$

  effettuiamo il passaggio da mappa di Karnaugh a lista di implicanti principali:

  - Si guardano le coordinate del sottocubo
  - Si considerano esclusivamente le variabili specificate
  - Se la coordinata relativa a una certa variabile è 0 si pone la variabile negata, se è uguale ad 1 si pone la variabile diretta.

Attenzione Non dobbiamo fermarci non appena riusciamo a coprire tutti gli 1 segnando un certo sottocubo. Lo step richiede di individuare tutti i sottocubi di un certo ordine che non sono già coperti interamente da altri sottocubi di ordine maggiore.

Esempio



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

Esempio 2 Presente un altro esempio da pagina 48 della dispensa.

21.2.5 Ricerca di liste di copertura non ridondanti

Classifichiamo i vari sottocubi. Si definiscono

- **sottocubi essenziali**: costituiscono il cuore della mappa e contengono sottocubi di ordine 1. Ciò significa che questi sottocubi non possono essere eliminati: fare ciò comporterebbe avere stati di ingresso riconosciuti dalla rete scoperti.

- **sottocubi assolutamente eliminabili**: sottocubi che <u>devono</u> essere eliminati. Se dopo aver segnato tutti i sottocubi di ordine 1 (appartenenti a sottocubi essenziali) abbiamo coperto un altro sottocubo allora questo può essere rimosso.

- **sottocubi semplicemente eliminabili**: sottocubi che <u>potrebbero</u> essere eliminati. In questo gli stati di ingresso sono già stati riconosciuti da altri sottocubi, almeno uno di questi sottocubo non essenziale.

Quando si applica l'algoritmo si individuano sicuramente le prime due tipologie di sottocubi.

1. Si verifica l'esistenza di sottocubi che sono gli unici a coprire un dato sottocubo di ordine 1. Questi sono i **sottocubi essenziali**, che non possono essere tolti.

2. Dopo aver evidenziato i sottocubi essenziali verifichiamo se esistono sottocubi coperti interamente da uno o più sottocubi essenziali. Se sì abbiamo dei **sottocubi assolutamente eliminabili**, ridondanti.

Liste di copertura non ridondanti nei casi più semplici La lista di copertura non ridondante presenterà le seguenti proprietà:

- sono presenti tutti i sottocubi essenziali

- sono stati rimossi tutti i sottocubi assolutamente eliminabili

Liste di copertura non ridondanti nei casi più complessi La lista di copertura presenterà anche sottocubi semplicemente eliminabili: significa che potrò ottenere più liste di copertura non ridondanti (cioè scelgo se considerare un sottoinsieme di questi sottocubi essenziale o assolutamente eliminabile). Come proseguiamo in questi casi?

- Scelgo un un qualunque sottocubo semplicemente eliminabile e formulo due ipotesi:

  – una in cui il cubo è essenziale e quindi lo aggiungo alla lista di copertura. Altri sottocubi semplicemente eliminabili diventeranno assolutamente eliminabili,

  – una in cui il cubo è assolutamente eliminabile e quindi lo tolgo da qualunque lista di copertura. Altri sottocubi semplicemente eliminabili diventeranno essenziali.

  Il risultato finale non dipende dal primo sottocubo scelto: ovviamente scegliere un certo sottocubo può comportare semplificazione o complicazione esistenziale.

- Se l'onnipotente ce l'ha con noi sarà necessario formulare ulteriori ipotesi prendendo come riferimento altri sottocubi semplicemente eliminabili.

- Con tutte queste cose andiamo a creare un albero di decisioni binario: le foglie di questo albero consistono in tutte le possibili liste di copertura non ridondanti.

Come scelgo le liste di costo minore? Prendo le liste di copertura e confronto i loro costi secondo il criterio adottato (se non si specifica un criterio in particolare non guasta verificare i costi con entrambi i criteri. Data una rete ad N ingressi con N_s sottocubi, ciascuno di ordine $D_j, 1 \leq j \leq N_s$, le formule per i costi sono:

- **Costo a porte**: $\text{CP} = N_s + 1$[1]

- **Costo a diodi**: $\text{CD} = N_s + \sum_{j=1}^{N_s}(N - \log_2 D_j)$ [2]

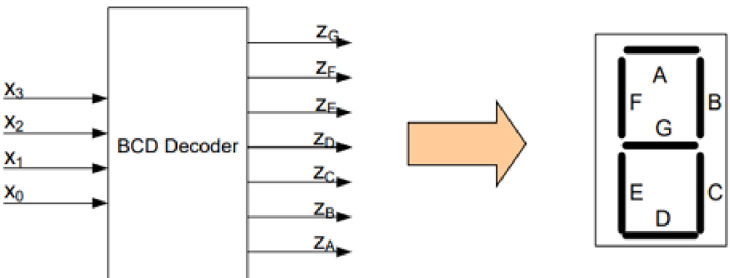
Attenzione alle liste ridondanti Si deve stare attenti che ad ogni passo non si generino liste non ridondanti. Vedere l'esempio a pagina 55.

[1]Numero di sottocubi più uno.

[2]Sommo il numero di sottocubi al numero di variabili relative a ciascun sottocubo

21.2.5.1 Sintesi di leggi non completamente specificate

Prendiamo l'esempio a pagina 47 della dispensa: quanto presente è un decodificatore BCD a 7 segmenti, una rete con quattro variabili di ingresso interpretate come la codifica in base 2 di una cifra decimale. Produce sette uscite: ciascuna contribuisce ad accendere un segmento del display.



Per ottenere la sintesi a costo minimo lavoreremo su ogni singola uscita. Nella dispensa si da un esempio con l'uscita E.

Perchè questa sintesi è diversa dalle altre? Abbiamo bisogno di quattro variabili di ingresso perchè solo con quattro bit è possibile rappresentare le cifre da 0 a 9. Sappiamo anche che con quattro bit si possono rappresentare numeri a due cifre: quei numeri non ci interessano, pertanto andremo a porre le uscite come non specificate.

Come si lavora?

- Metto i non specificati nelle celle della mappa di Karnaugh
- Ricerco gli implicanti principali considerando i valori non specificati come degli 1 (implicanti più grandi coprono più roba, cit.)
- Ricerco gli implicanti essenziali considerando i valori non specificati come zeri (se delle cose rimangono scoperte non ne farò un problema)

6.2.4 Riepilogo – definizione e classificazione di implicanti e sottocubi

Tipica domanda d'esame, alla quale bisogna saper rispondere.

Algebra	Definizione
Forma Canonica SP	Data una legge $z = f\left(x_{N-1},...,x_0\right)$, la FC SP è quella che si ottiene dall'espansione di Shannon della legge, cioè: $\begin{aligned} z = & f\left(0,...,0,0\right)\cdot\overline{x_{N-1}}\cdot\overline{x_{N-2}}\cdot...\cdot\overline{x_1}\cdot\overline{x_0} \\ & +f\left(0,...,0,1\right)\cdot\overline{x_{N-1}}\cdot\overline{x_{N-2}}\cdot...\cdot\overline{x_1}\cdot x_0 \\ & ... \\ & +f\left(1,...,1,0\right)\cdot x_{N-1}\cdot x_{N-2}\cdot...\cdot x_1\cdot\overline{x_0} \\ & +f\left(1,...,1,1\right)\cdot x_{N-1}\cdot x_{N-2}\cdot...\cdot x_1\cdot x_0 \end{aligned}$ applicando le identità $0+\alpha=\alpha$, $1\cdot\alpha=\alpha$, $0\cdot\alpha=0$
Mintermine	Prodotto di **tutte** le variabili di ingresso dirette o negate, che compare in una **forma canonica SP** e riconosce **uno stato di ingresso**.
Implicante	Prodotto di **alcune** variabili di ingresso dirette o negate, che compare in una **forma SP** di una legge di corrispondenza. Si ottiene a partire dalla forma canonica SP applicando esaustivamente le seguenti regole: $\begin{cases} \alpha x+\alpha\overline{x}=\alpha x+\alpha\overline{x}+\alpha \\ \alpha+\alpha=\alpha \end{cases}$ Riconosce **alcuni stati di ingresso**.
Implicante Principale	Implicante che si ottiene dalla lista degli implicanti applicando esaustivamente la legge $\alpha x+\alpha=\alpha$
I.P. Essenziale	Implicante che è **l'unico**, tra quelli principali, ad implicare un dato mintermine (a riconoscere uno stato di ingresso).
I.P. Assolut. Eliminabile	Implicante che riconosce solo stati di ingresso già riconosciuti da I.P. essenziali.
I.P. Semplic. Eliminabile	Implicante che riconosce solo stati di ingresso riconosciuti da altri I.P., almeno uno dei quali riconosciuto da un I.P. non essenziale.

SINTESI PORTA XOR IN FORMA SP (2 INGRESSI):

$x_0 \backslash x_1$	0	1
0	0	1
1	1	0

$$z = x_1 \cdot \overline{x_0} + \overline{x_1} \cdot x_0 = x_1 \oplus x_0$$

SINTESI PORTA XNOR IN FORMA SP (2 INGRESSI):

$x_0 \backslash x_1$	0	1
0	1	0
1	0	1

$$z = \overline{x_1} \cdot \overline{x_0} + x_0 \cdot x_1 = x_1 \overline{\oplus} x_0$$

SINTESI PORTA NAND IN FORMA SP (2 INGRESSI):

$x_0 \backslash x_1$	0	1
0	1	1
1	1	0

x_1	x_0
0	–
–	0

$$z = \overline{x_1} + \overline{x_0}$$

OTTENIBILE CON DE MORGAN

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

SINTESI PORTA NOR IN FORMA SP (2 INGRESSI):

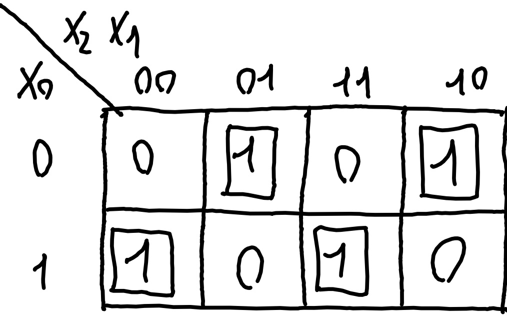
$x_0 \backslash x_1$	0	1
0	1	0
1	0	0

$$z = \overline{x_0} \cdot \overline{x_1}$$

OTTENIBILE CON DE MORGAN

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

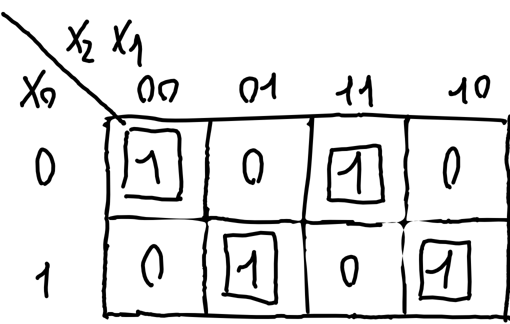
SINTESI PORTA XOR IN FORMA SP (3 INGRESSI):



- MINTERMINE CON VARIABILI TUTTE DIRETTE
- COMBINAZIONI POSSIBILI DI MINTERMINI CON 2 VARIABILI NEGATE

$$Z = \overline{X}_2 \cdot \overline{X}_1 \cdot X_0 + \overline{X}_2 X_1 \overline{X}_0 + X_2 \cdot X_1 \cdot X_0 + X_2 \cdot \overline{X}_1 \cdot \overline{X}_0$$
$$= X_2 \oplus X_1 \oplus X_0$$

SINTESI PORTA XNOR IN FORMA SP (3 INGRESSI):

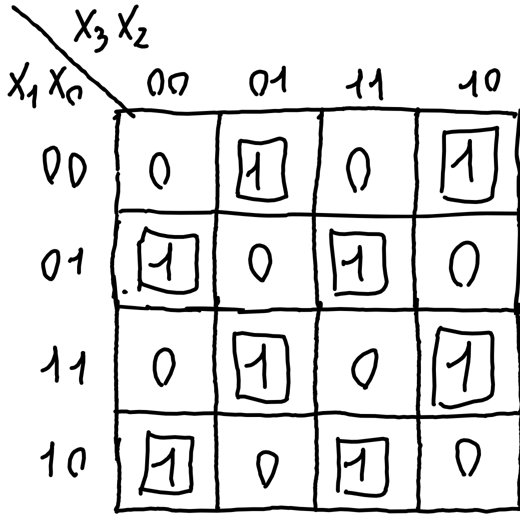


- MINTERMINE CON VARIABILI TUTTE NEGATE
- COMBINAZIONI POSSIBILI DI MINTERMINI CON UNA VARIABILE NEGATA

$$Z = \overline{X}_2 \cdot \overline{X}_1 \cdot \overline{X}_0 + \overline{X}_2 X_1 X_0 + X_2 \cdot X_1 \cdot \overline{X}_0 + X_2 \cdot \overline{X}_1 \cdot X_0$$
$$= X_2 \overline{\oplus} X_1 \overline{\oplus} X_0$$

UTILE, MA NEANCHE TANTO

SINTESI PORTA XOR IN FORMA SP (4 INGRESSI):



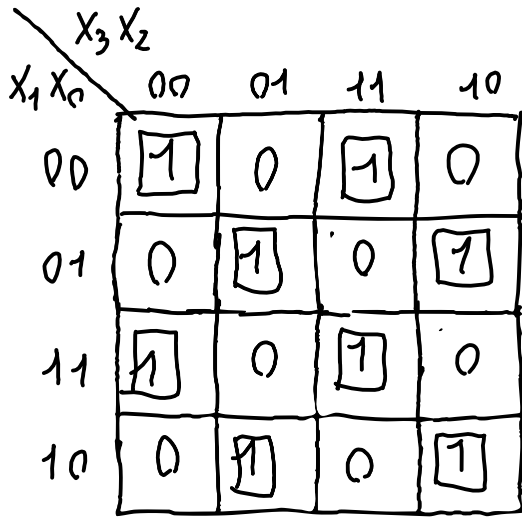
x_1x_0 \ x_3x_2	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

- COMBINAZIONI POSSIBILI DI MINTERMINI CON 3 VARIABILI NEGATE
- COMBINAZIONI POSSIBILI DI MINTERMINI CON UNA VARIABILE NEGATA

$$z = \bar{x}_3\bar{x}_2\bar{x}_1x_0 + \bar{x}_3\bar{x}_2x_1\bar{x}_0 + \bar{x}_3x_2\bar{x}_1\bar{x}_0 + \bar{x}_3x_2x_1x_0 +$$
$$x_3x_2\bar{x}_1x_0 + x_3x_2x_1\bar{x}_0 + x_3\bar{x}_2\bar{x}_1\bar{x}_0 + x_3\bar{x}_2x_1x_0 =$$
$$= x_3 \oplus x_2 \oplus x_1 \oplus x_0$$

PAZZIA RICORDARLO

SINTESI PORTA XNOR IN FORMA SP (4 INGRESSI):



- MINTERMINE CON VARIABILI TUTTE DIRETTE
- MINTERMINE CON VARIABILI TUTTE NEGATE
- COMBINAZIONI POSSIBILI DI MINTERMINI CON 2 VARIABILI NEGATE

$$Z = \bar{x}_3 \bar{x}_2 \bar{x}_1 \bar{x}_0 + \bar{x}_3 \bar{x}_2 x_1 x_0 + \bar{x}_3 x_2 \bar{x}_1 x_0 + \bar{x}_3 x_2 x_1 \bar{x}_0 +$$

$$x_3 x_2 \bar{x}_1 \bar{x}_0 + x_3 x_2 x_1 x_0 + x_3 \bar{x}_2 \bar{x}_1 x_0 + x_3 \bar{x}_2 x_1 \bar{x}_0 =$$

$$= x_3 \overline{\oplus} x_2 \overline{\oplus} x_1 \overline{\oplus} x_0$$

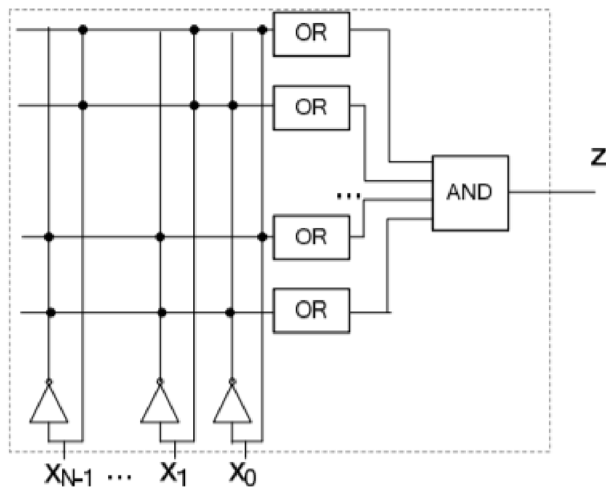
PAZZIA RICORDARLO

Capitolo 22

Martedì 20/10/2020

22.1 Sintesi di reti in formato PS

Un'alternativa alla sintesi in formato SP è la sintesi in formato PS, dove PS sta per prodotto di somme. Lo schema è molto simile a quello visto con le forme SP:



ma si sostituiscono le porte OR con le porte AND e viceversa.

Legge in forma PS Una legge in forma PS può essere scritta così

$$z = S_1 \cdot S_2 \cdot \dots \cdot S_n$$

dove $S_i = \sum x_j$, cioè la somma di variabili di ingresso dirette o negate. Si parla di **IMPLICATI**.

Teoria duale Otteniamo definizioni e concetti teorici simili a quelli visti con la forma SP (per esempio la forma canonica, la lista degli implicati e così via...): si parla di **teoria duale**.
Noi non introdurremo la teoria per le reti in forma PS, ma sfrutteremo quanto già visto per arrivare a sintetizzare una rete in formato PS di costo minimo.

Come si sintetizza una rete in formato PS a partire da una legge F?

- Abbiamo una legge di corrispondenza F (una tavola di verità): ricavo il corrispondente $\overline{F}$: date le uscite z di F, $\overline{F}$ avrà come uscite $\overline{z}$.
- Realizzo una sintesi in forma SP a costo minimo della legge $\overline{F}$
- Ottengo una sintesi della legge F inserendo un invertitore in uscita alla rete ottenuta allo step precedente ($\overline{\overline{F}} = F$).
- Applico, muovendomi da destra verso sinistra (all'indietro), i teoremi di De Morgan
  - Al posto della somma finale negata (porta OR con k ingressi e invertitore) pongo il prodotto dei suoi ingressi negati

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

  - Al posto di ciascun prodotto negato pongo le somme dei suoi ingressi negati

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

Quanto ottenuto è in forma PS. 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.

Segue

Se la sintesi SP di $\overline{F}$ è di costo minimo lo è anche la sintesi PS di F.

Questa cosa è dimostrabile per assurdo.

- Data una realizzazione SP di una legge $\overline{F}$ la realizzazione PS di F che si ottiene mettendo un invertitore in fondo e applicando De Morgan ha lo stesso costo, sia prendendo come riferimento le porte che gli ingressi.
- Segue che se esistesse una realizzazione PS di F di costo minore esisterebbe anche una realizzazione SP di $\overline{F}$ di costo minore.
- Ciò è contro l'ipotesi poichè abbiamo utilizzato il procedimento di sintesi SP a costo minimo.

■

22.1.1 Sintesi duale

- Data una legge F siamo in grado di ottenere la sintesi a costo minimo in forma SP e la sintesi a costo minimo in forma PS.
- Quest'ultima sintesi si ottiene a partire dalla sintesi a costo minimo in forma SP di $\overline{F}$
- Data una sintesi qualunque (SP o PS) per F siamo in grado di ricavare la sintesi duale di $\overline{F}$ applicando un invertitore in coda e De Morgan.
- Si dice che questa sintesi è in **forma duale** rispetto a quella di partenza (PS se quella di partenza era SP e viceversa).

Ribadiamo: data una legge F si definisce sintesi duale la sintesi di $\overline{F}$ ottenuta inserendo un invertitore in coda ed applicando due volte De Morgan.

22.1.2 Sintesi meno costosa

Data una legge F, quale sintesi è meno costosa? SP di F o PS di F? Ricordiamo che poco prima abbiamo detto che

$$\boxed{\text{SP(F)} = \text{PS}(\overline{F})}$$

e non SP(F) = PS(F).

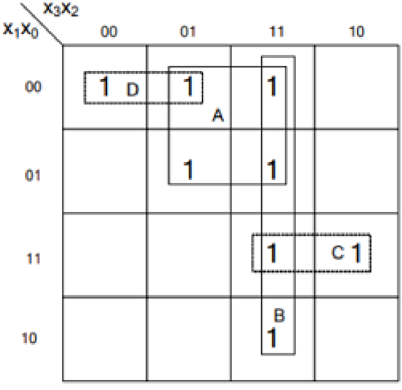
- Non è possibile dare una risposta valida in generale. Dipende dalla tavola di verità.
- Siamo certi che otterremo la sintesi a costo minimo rispetto a una certa sintesi, ma non possiamo essere certi di scegliere, tra le due sintesi (SP, PS), quella effettivamente a costo minore.

Mappa di Karnaugh corrispondente Data la mappa di Karnaugh di F, la mappa corrispondente di $\overline{F}$ si ottiene ponendo 1 dove si ha 0 e 1 dove si ha 0. Se si estrae l'espressione corrispondente dalla mappa di $\overline{F}$ otterremo prodotti di somme.

Mappa di Karnaugh per F

x_1x_0 \ x_3x_2	00	01	11	10
00				1
01	1			1
11	1	1		
10	1	1		1

Mappa di Karnaugh per $\overline{F}$



22.2 Porte logiche universali

Fino ad ora abbiamo sempre parlato di porte AND, OR e NOT, ma mai di porte NAND e NOR. Queste porte sono definite *universali* poichè risulta possibile sintetizzare una rete, data una legge combinatoria, usando esclusivamente porte NAND o porte NOR.

		Porta	realizz. a NAND	realizz. a NOR
NOT	$x = x \cdot x \Rightarrow \overline{x} = \overline{x \cdot x}$ $x = x + x \Rightarrow \overline{x} = \overline{x + x}$	x, z	x, z	x, z
AND	$x \cdot y = \overline{\left(\overline{x \cdot y}\right)}$ $x \cdot y = \overline{\overline{x} + \overline{y}}$	x, y, z	x, y, z	x, y, z
OR	$x + y = \overline{\overline{x} \cdot \overline{y}}$ $x + y = \overline{\left(\overline{x + y}\right)}$	x, y, z	x, y, z	x, y, z

- Nel ragionare ricordiamoci che:
  - la porta NAND consiste in una porta AND con invertitore a valle;
  - la porta NOR consiste in una porta OR con invertitore a valle.
- **Porte NOT**:
  - Sappiamo dall'algebra di Boole che $x + x = x$ ed $x \cdot x = x$
  - Seguono le seguenti uguaglianze:
    * $\overline{x} = \overline{x \cdot x}$, cioè una porta NAND che ha due ingressi uguali ad x.
    * $\overline{x} = \overline{x + x}$, cioè una porta NOR che ha due ingressi uguali ad x.
- **Porte AND**:
  - Sappiamo che $x \cdot y = \overline{\overline{x \cdot y}}$
  - Non applico De Morgan. Ottengo due porte NAND in cascata: la prima ha come ingressi x ed y, la seconda ha due ingressi uguali al risultato della porta precedente.
  - Applico De Morgan. Ottengo $x \cdot y = \overline{\overline{x} + \overline{y}}$, cioè tre porte NOR: una ha due ingressi uguali ad x, un'altra uguali ad y, mentre l'ultima presenta in ingresso i risultati delle prime due porte NOR.
- **Porte OR**:
  - Sappiamo che $x + y = \overline{\overline{x + y}}$
  - Non applico De Morgan. Ottengo due porte NOR in cascata: la prima ha come ingressi x ed y, la seconda ha due ingressi uguali al risultato della porta NOR precedente.
  - Applico De Morgan. Ottengo $x + y = \overline{\overline{x} \cdot \overline{y}}$, cioè tre porte NAND: una ha due ingressi uguali ad x, un'altra uguali ad y, mentre l'ultima presenta in ingresso i risultati delle prime due porte NAND.

- POSSIAMO COSTITUIRE RETI DI SOLE PORTE NAND o DI SOLE PORTE NOR. → PER QUESTO SI DICONO PORTE LOGICHE UNIVERSALI

SINTETIZZAZIONE: [SINTETIZZEREMO PORTE AND, OR, NOT USANDO LE PORTE NAND o LE NOR]

① PORTE NOT

$X = X \cdot X \Rightarrow \overline{X} = \overline{X \cdot X}$

ATTENZIONE AGLI INGRESSI: LA PORTA NAND RESTITUISCE UN PRODOTTO COMPLEMENTATO

PORTE NAND:

X_1	X_2	Z
0	0	1
0	1	1
1	0	1
1	1	0

$Z = 0 \Leftrightarrow X_1 = X_2 = 1$

X_1 X_2 → AND → NOT → Z

$X = X + X \Rightarrow \overline{X} = \overline{X + X}$

ATTENZIONE AGLI INGRESSI: LA PORTA NOR RESTITUISCE LA SOMMA COMPLEMENTATA

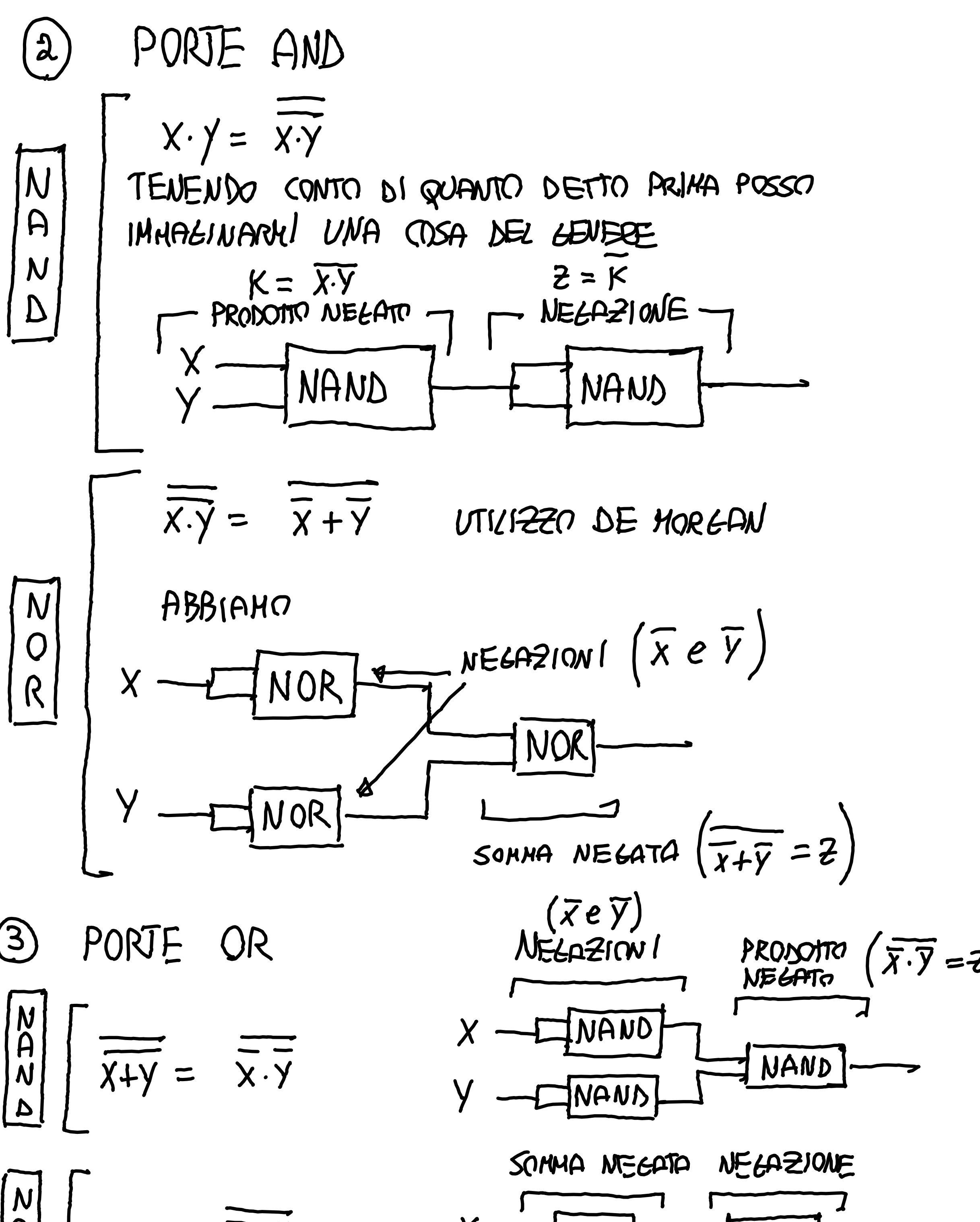
PORTE NOR:

X_1	X_2	Z
0	0	1
0	1	0
1	0	0
1	1	0

$Z = 1 \Leftrightarrow X_1 = X_2 = 0$

X_1 X_2 → OR → NOT → Z

RISULTATO ABBASTANZA EVIDENTE: SINTESI CON PORTE NAND o PORTE NOR CON INGRESSI UGUALI



2 PORTE AND
NAND
$X \cdot Y = \overline{\overline{X \cdot Y}}$
TENENDO CONTO DI QUANTO DETTO PRIMA POSSO IMMAGINARMI UNA COSA DEL GENERE
$K = \overline{X \cdot Y}$
$Z = \overline{K}$
PRODOTTO NEGATO
NEGAZIONE
X
Y
NAND
NAND
NOR
$\overline{\overline{X \cdot Y}} = \overline{\overline{X} + \overline{Y}}$
UTILIZZO DE MORGAN
ABBIAMO
NEGAZIONI $(\overline{X}$ e $\overline{Y})$
X
Y
NOR
NOR
NOR
SOMMA NEGATA $(\overline{\overline{X}+\overline{Y}} = Z)$
3 PORTE OR
NAND
$\overline{\overline{X+Y}} = \overline{\overline{X} \cdot \overline{Y}}$
$(\overline{X}$ e $\overline{Y})$
NEGAZIONI
PRODOTTO NEGATO $(\overline{\overline{X} \cdot \overline{Y}} = Z)$
X
Y
NAND
NAND
NAND
NOR
$X+Y = \overline{\overline{X+Y}}$
SOMMA NEGATA
NEGAZIONE
X
Y
NOR
NOR
$(K = \overline{X+Y})$
$(Z = \overline{K})$

- SINTESI IN PORTE NAND:
  - PARTO DA UN CIRCUITO IN FORMA SP
  - SOSTITUISCO LE PORTE CON GLI EQUIVALENTI NAND.
  - STESSA COSA PER LE PORTE NOT, MA QUELLE AGLI INGRESSI SI LASCIANO STARE.
  - ELIMINO LE COPPIE DI NAND IN CASCATA

FORMA SP → COMPLEMENTO DUE VOLTE → APPLICO DE MORGAN UNA VOLTA

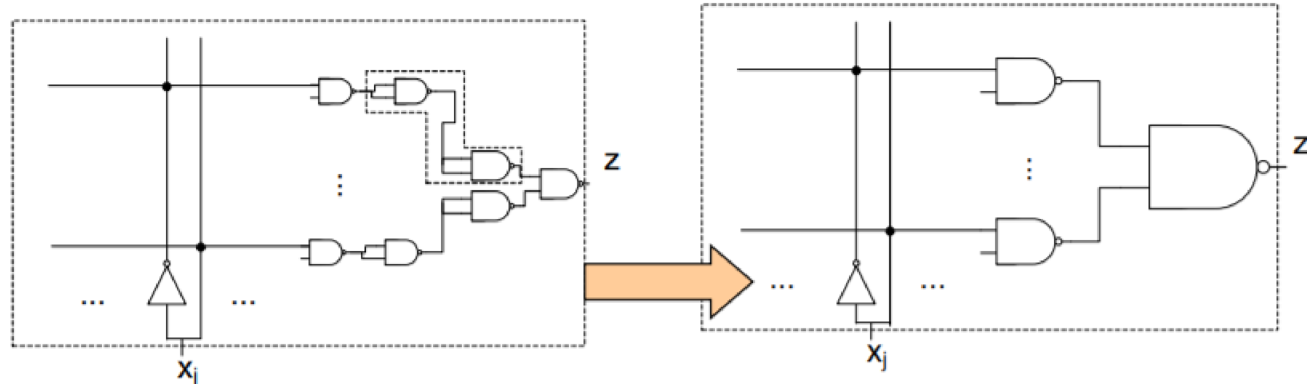
$$Z = P_1 + \dots + P_K = \overline{\overline{P_1 + \dots + P_K}}$$
$$= \overline{\overline{P_1} \cdot \dots \cdot \overline{P_K}}$$

- SINTESI IN PORTE NOR:
  - PARTO DA UN CIRCUITO IN FORMA PS
  - SOSTITUISCO LE PORTE CON GLI EQUIVALENTI NOR.
  - STESSA COSA PER LE PORTE NOT, MA QUELLE AGLI INGRESSI SI LASCIANO STARE.
  - ELIMINO LE COPPIE DI NOR IN CASCATA

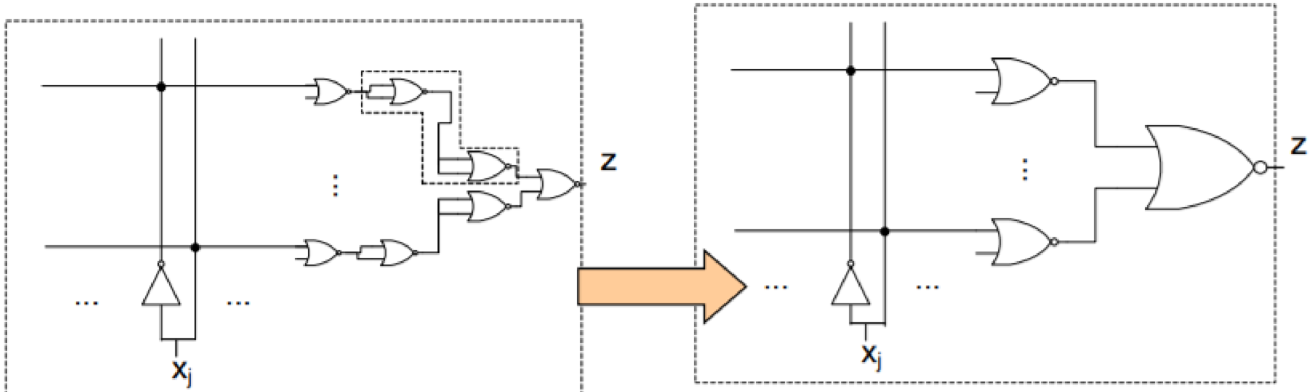
FORMA PS → COMPLEMENTO DUE VOLTE → APPLICO DE MORGAN UNA VOLTA

$$Z = S_1 \cdot \dots \cdot S_K = \overline{\overline{S_1 \cdot \dots \cdot S_K}}$$
$$= \overline{\overline{S_1} + \dots + \overline{S_K}}$$

Sintesi a porte NAND



Sintesi a porte NOR



22.2.1 Costo a porte e costo a diodi

Supponiamo di voler mettere a confronto una sintesi in porte NAND con una sintesi in porte NOR. Come dobbiamo comportarci?

- Si utilizzano le stesse formule introdotte precedentemente.
- Le applichiamo utilizzando non l'espressione finale, ma quella iniziale (quella che abbiamo prima di complementare due volte e applicare De Morgan una volta). Ricordarsi che

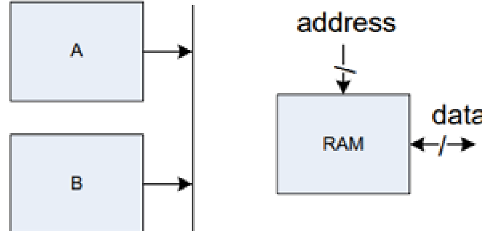
Se la sintesi SP di $\overline{F}$ costa c allora la sintesi PS di F costa c.

Esercizio Si veda, per chiarimenti, la soluzione dell'esercizio 2.4.4 a pagina 19 della dispensa di aritmetica (non presente in questi appunti).

Capitolo 23

Mercoledì 21/10/2020

23.1 Porte tri-state

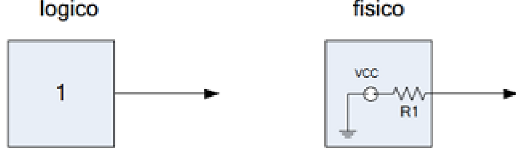


In molti casi fa comodo poter connettere insieme le uscite delle reti. Per esempio

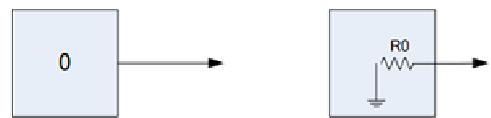
- ho un bus condiviso, una o più linee che reti diverse possono impostare ad un dato valore. Non lo faranno contemporaneamente, ma dovranno fare a turno.
- ho linee che servono, in tempi diversi, come ingressi ed uscite ad una data rete (pensiamo alla memoria RAM)

Ricordiamo

- **Generatore di costante 1**: rete la cui uscita è attaccata al polo positivo di un generatore di tensione tramite una resistenza di un certo valore



- **Generatore di costante 0**: rete la cui uscita è attacata al polo negativo di una batteria tramite una piccola resistenza



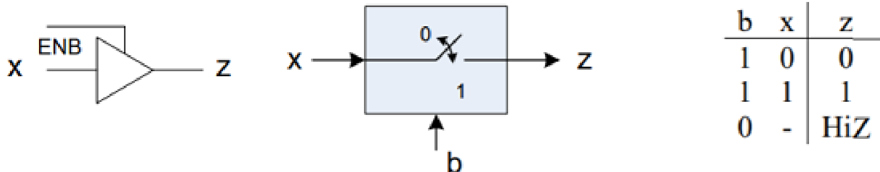
L'uscita della rete può essere immaginata come un interruttore: mi attaccato alla tensione V_{cc} o alla tensione di massa. Tutto questo è ragionevole finchè si parla di una sola uscita, ma cosa succede se collego più uscite?

Problemi elettrici e logici Supponiamo di collegare le uscite di due reti

- **Le due reti restituiscono 1**: dal punto di vista fisico entrambe le reti hanno la stessa struttura e la linea condivisa presenta la stessa tensione V_{cc}. Non scorre corrente.
- **Le due reti restituiscono 0**: anche qua le reti hanno la stessa struttura. Il potenziale di riferimento è lo stesso e la tensione sulla linea condivisa è 0. Non scorre corrente
- **Le due reti restituiscono valori discordi**: qua saltano fuori problemi! Ho una maglia chiusa: scorre tanta corrente e i circuiti si bruciano molto facilmente. Quale sarà il valore logico? Non lo possiamo sapere, dipende dalle resistenze (dettagli che noi solitamente non guardiamo).

La situazione peggiora se aggiungo una nuova rete.

Come risolviamo? Ci servono delle porte capaci di disconnettere fisicamente un'uscita da una linea condivisa (è l'unico modo per evitare sia i problemi elettrici che quelli logici). Queste porte sono dette **porte tri-state**.



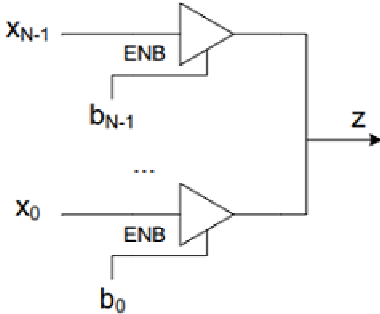
b	x	z
1	0	0
1	1	1
0	-	HiZ

- Sono porte con un ingresso e un'uscita.
- In aggiunta si ha una variabile di ingresso ENB, un enabler, che mi permette di pilotare la porta.
- Precisamente, se la variabile è uguale ad 1 la porta si comporta da elemento neutro (si ha in uscita il valore in ingresso). Se invece la variabile è uguale a 0 si dice che la porta sia in **ALTA IMPEDENZA**. La linea, ricordiamo, è fisicamente disconnessa.

Bus condivisi

Se abbiamo N uscite collegate insieme dovrà essere presente, su ogni uscita, una porta tri-state. In ogni istante almeno $N-1$ porte dovranno trovarsi in alta impedenza. Questo ci permette di evitare i problemi visti poco fa.

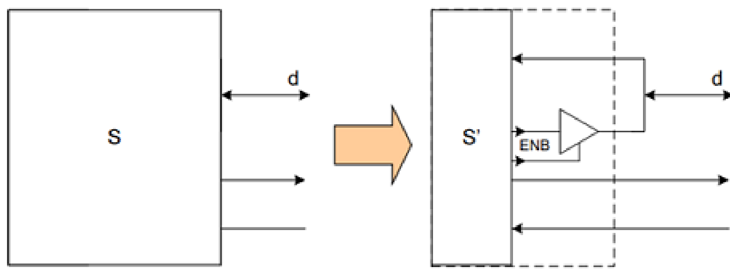
Esempio Un **multiplexer decodificato** consiste in una rete che come il multiplexer tradizionale restituisce una sola variabile logica in uscita. La differenza si ha nella struttura: invece di avere un decoder avrò tutte le variabili in ingresso passanti per una porta tri-state, e tante variabili di comando quante le porte-tristate. Si dice che se tutte le variabili di comando valgono 0 allora l'uscita è in alta impedenza. Affinchè tutto funzioni UNA SOLA variabile di comando dovrà essere ad 1.



Linee di ingresso/uscita

Si ha un montaggio a forchetta:

- Una linea funge da ingresso.
- L'altra, dedicata all'uscita, è forchettata da porte tri-state. Se l'enabler è uguale a 0 le linee fungono da ingresso, altrimenti fungono da uscite. L'idea è che la rete collegata alle linee abbia a sua volta delle porte tri-state: queste si dovranno comportare in modo opposto rispetto alle prime porte tri-state.

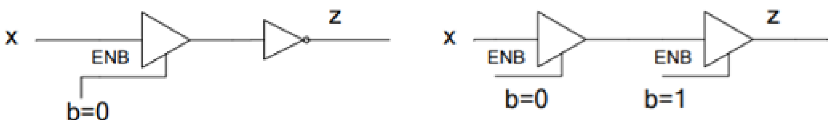


RICORDARE COME L'AVE MARIA

L'alta impedenza non è un valore logico.

L'alta impedenza è un filo staccato, non rappresenta qualcosa di particolare.

Esempi



1. Nella prima rete è presente una porta tri-state e subito dopo un invertitore. L'invertitore è un qualcosa di collegato o alla tensione V_{cc} o alla tensione di massa. Finchè questa porta non si rompe avrò come valore 0 o 1. L'interpretazione dipende dalla porta.
2. Ho due porte tri-state in sequenza. La prima ha $b = 0$, la seconda $b = 1$. Anche in questo caso l'uscita della rete sarà 0 o 1. L'uscita dipende dall'interpretazione che da la seconda porta tri-state al suo valore di ingresso.

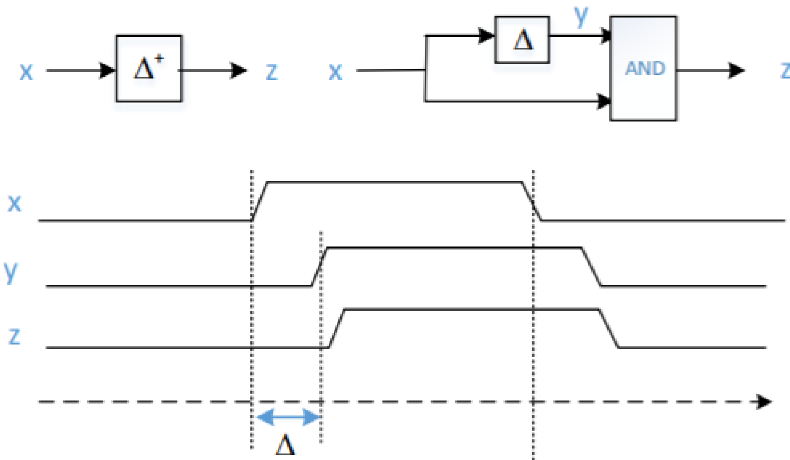
23.2 Circuiti di ritardo

Il ritardo viene generato attraverso dei buffer, degli elementi neutri. Solitamente questi elementi vengono realizzati attraverso un numero pari di porte NOT. Possiamo porre un numero di porte NOT tale da ottenere più o meno il ritardo voluto.

Ritardo simmetrico Il ritardo è simmetrico, cioè identico su transizioni 0-1 e transizioni 1-0 (ritardi simmetrici tra fronti di salita e fronti in discesa).

Non è quello che vogliamo Ci interessano circuiti con ritardo asimmetrico: grande sulle transizioni 0-1, piccolo sulle transizioni 1-0 (o viceversa).

23.2.1 Circuito di ritardo sul fronte di salita



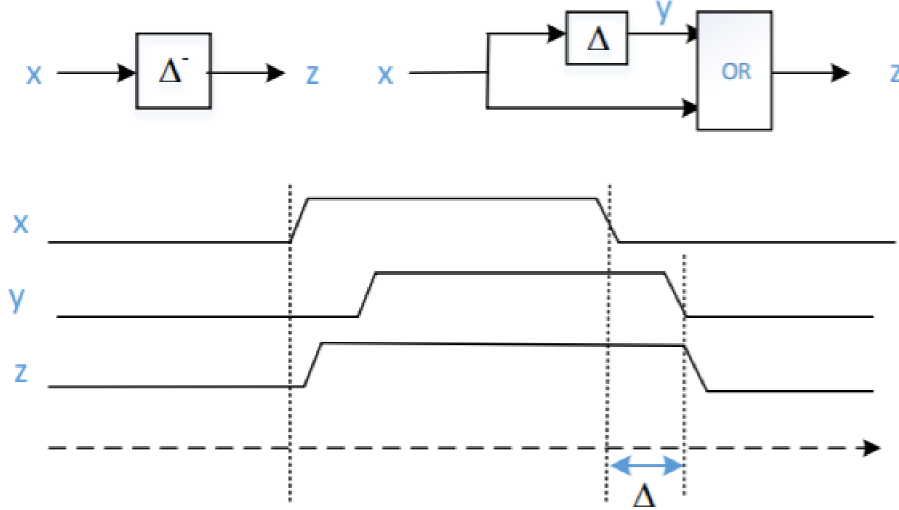
Questo circuito permette di ottenere ritardo sul fronte di salita, cioè quando passiamo da 0 ad 1. Si indica con Δ^+ e si realizza con una porta AND: questa avrà come ingressi

- x diretto, e
- x ritardato Δ.

Dal diagramma presente a pagina 78 si capisce cosa succede

- Abbiamo inizialmente $x = 0$, quindi $y = 0$
- A un certo punto otteniamo il fronte di salita, quindi $x = 1$
- Il fronte di salita di y viene ritardato di un tempo $\approx \Delta$.
- Per un certo periodo avrò in ingresso nella porta AND due valori discordi, che mi mantengono il valore in uscita uguale a 0
- Quando $y = 1$ la porta AND torna a ricevere valori concordi, quindi restituisce il valore x posto in ingresso al circuito con un ritardo $\approx \Delta$.
- Se vogliamo tornare ad $x = 0$, quindi ottenere un fronte di discesa, otterremo subito due valore discordi: la rete si adegua in modo immediato.

23.2.2 Circuito di ritardo sul fronte di discesa



Questo circuito permette di ottenere ritardo sul fronte di salita, cioè quando passiamo da 1 ad 0. Si indica con Δ^- e si realizza con una porta OR: questa avrà come ingressi

- x diretto, e
- x ritardato Δ.

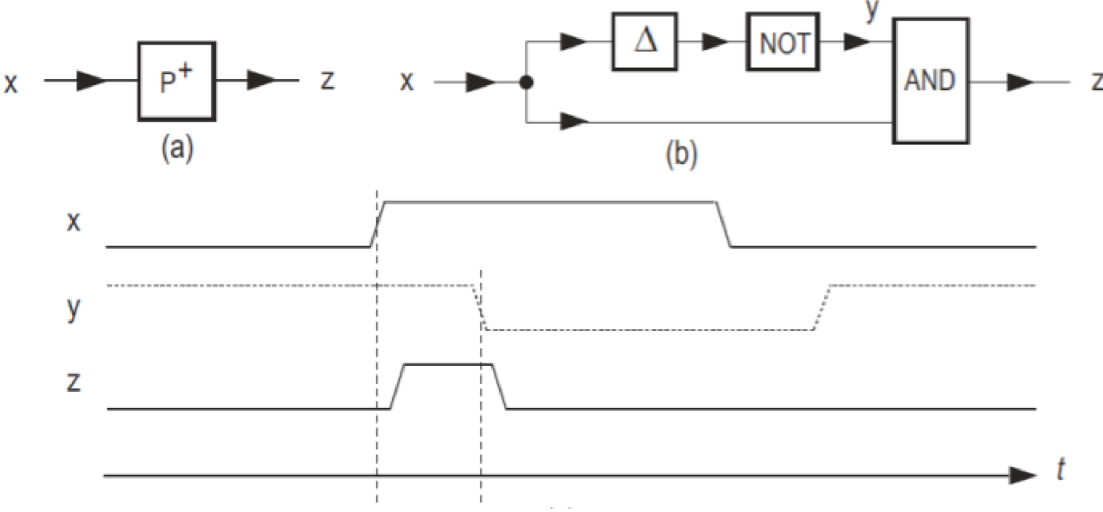
Dal diagramma presente a pagina 79 si capisce cosa succede

- Abbiamo inizialmente $x = 0$, quindi $y = 0$
- A un certo punto otteniamo il fronte di salita, quindi $x = 1$.
- Il fronte di salita di y viene ritardato di un tempo $\approx \Delta$.
- Ciò è ininfluente nel fronte di salita: una porta OR restituisce 1 se presenta almeno un valore di ingresso uguale ad 1. Segue che l'adeguamento dell'uscita è immediato.
- Quando $y = 1$ la porta OR continua a restituire la stessa uscita.
- Quando otteniamo il fronte di uscita, quindi $x = 0$, la porta OR riceverà, per un certo tempo $\approx \Delta$, valori discordi. La porta OR continua a restituire 1 in quell'intervallo.
- Quando l'uscita y si adeguerà al cambiamento di x otterremo due valori concordi e uguali a 0 in ingresso nella porta OR: a quel punto la porta OR si adegua ad x e restituisce un'uscita aggiornata.

23.3 Formatore di impulsi

Usando la stessa strategia adottata per i circuiti di ritardo posso realizzare dei formatori di impulsi, cioè dei circuiti che generano un impulso (uscita uguale ad 1) per una durata nota $\approx \Delta$. Posso realizzare due formatori di impulsi: uno che restituisce l'impulso in presenza di un fronte di salita, un altro che restituisce l'impulso in presenza di un fronte di discesa.

23.3.1 Formatore di impulsi sul fronte di salita



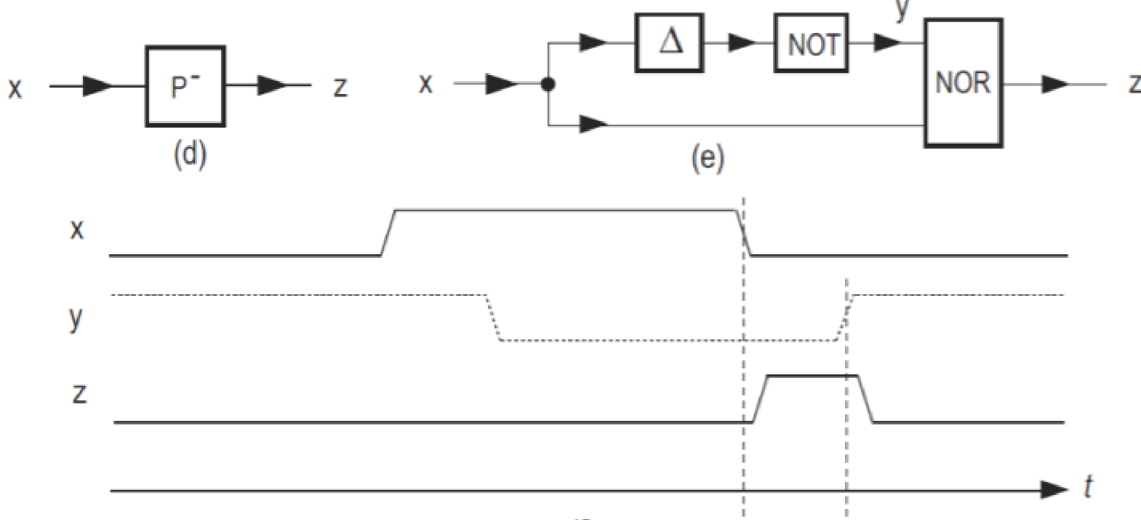
Questo circuito permette di ottenere un impulso in presenza di un fronte di salita. Si indica con P^+ e si realizza con una porta AND: questa avrà come ingressi

- x diretto, e
- x ritardato Δ e negato.

Dal diagramma vediamo che

- La porta AND ha in ingresso due valori solitamente discordi: questo fa sì che in situazioni non transitorie si abbia in uscita sempre 0.
- Quando otteniamo il fronte di salita, quindi $x = 1$, avrò un ritardo $\approx \Delta$ per il fronte di salita di y.
- Questo fa sì che la porta AND riceva, per un certo intervallo di tempo $\approx \Delta$, due valori concordi e uguali ad 1 in ingresso. Per questo intervallo di tempo la porta AND restituirà 1.
- Quando otteremo il fronte di salita di y la porta AND riceverà nuovamente valori discordi in ingresso, quindi tornerà a restituire 0.
- Con il fronte di discesa, cioè $x = 0$, la porta AND riceve subito due valori discordi in ingresso: segue che l'uscita rimarrà uguale a 0.

23.3.2 Formatore di impulsi sul fronte di discesa



Questo circuito permette di ottenere un impulso in presenza di un fronte di salita. Si indica con P^{-} e si realizza con una porta NOR: questa avrà come ingressi

- x diretto, e
- x ritardato Δ e negato.

Dal diagramma vediamo che

- La porta NOR restituisce 1 quando entrambi gli elementi sono uguali a 0.
- La porta NOR ha in ingresso due valori solitamente discordi: questo fa sì che in situazioni non transitorie si abbia in uscita sempre 0.
- Quando otteniamo il fronte di salita, quindi $x = 1$, avrò un ritardo $\approx \Delta$ per il fronte di salita di y.
- Questo fa sì che la porta NOR riceva, per un certo intervallo di tempo $\approx \Delta$, due valori concordi e uguali ad 1 in ingresso. Il valore rimane zero.
- Con il fronte di discesa, cioè $x = 0$, la porta NOR riceve inizialmente due valori concordi e uguali a 0: questo fa sì che si abbia in uscita 1 per un intervallo di tempo $\approx \Delta$.
- Dopo l'adeguamento di y ad x la porta NOR riceverà nuovamente valori discordi in ingresso, quindi tornerà a restituire 0.

Parte V

Aritmetica di un calcolatore

Capitolo 24

Mercoledì 21/10/2020

24.1 Rappresentazione dei numeri naturali

Con questa sezione iniziamo a parlare di Aritmetica sui calcolatori, ossia reti combinatorie utilizzate per manipolare numeri naturali e numeri interi (ci limiteremo a questi per ora, numeri reali in virgola mobile sono troppo complessi - cit). Sappiamo che l'informazione è un qualcosa di astratto e che debba essere adeguatamente rappresentata per essere usata in un calcolatore.

Sistema numerico di rappresentazione Un sistema numerico di rappresentazione è caratterizzato da

- una base di rappresentazione $\beta \geq 2$
- un insieme di simboli dette *cifre*, che consistono nei numeri naturali compresi tra 0 e $\beta - 1$.
- Una legge di rappresentazione che fa corrispondere ad ogni sequenza di cifre un numero naturale

Il sistema adottato è di tipo **posizionale**: questo significa che cifre uguali collocate in posizioni diverse non avranno lo stesso significato. La legge di rappresentazione adottata è quella vista fin dai tempi di *Fondamenti di programmazione*

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

Esistono altre notazioni? I numeri arabi con notazione posizionale sono stati introdotti in Europa da Leonardo Fibonacci. Fino ad allora abbiamo usato i numeri romani, basati su un sistema additivo.

Perchè è meglio la notazione posizionale? La notazione posizionale permette di utilizzare algoritmi molto semplici. Fare la divisione con i numeri romani, per esempio, è una pazzia!

Osservazione sulle rappresentazioni Difetto di tutti noi è non considerare la base dieci una rappresentazione, cioè dire che in base dieci si ha il numero e in tutte le altre basi una rappresentazione. La base dieci stessa è una rappresentazione del numero!

Esempi

Sistema numerico decimale

- **Base**: $\beta = 10$
- **Cifre**: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- **Esempio di numero**:

$$(2042)_{10} = 2 \cdot 10^0 + 4 \cdot 10^1 + 0 \cdot 10^2 + 2 \cdot 10^3$$

Sistema numerico esadecimale

- **Base**: $\beta = 16$
- **Cifre**: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$
- **Esempi di numeri**:

$$(A)_{16} = (10)_{10} \quad \dots \quad (F)_{16} = (15)_{10}$$

$$(1A2F)_{16} = (F)_{16} \cdot (16^0)_{10} + (2)_{16} \cdot (16^1)_{10} + (A)_{16} \cdot (16^2)_{10} + (1)_{16} \cdot (16^3)_{10}$$

Sistema numerico binario

- **Base**: $\beta = 2$
- **Cifre**: $\{0, 1\}$

24.2 Teorema della divisione con resto

Questo teorema permetterà di rispondere alle seguenti domande:

- Posso sempre rappresentare un certo numero in una certa base?
- Se sì, la rappresentazione è unica?
- Come faccio a trovare la rappresentazione?

Enunciato

$$\boxed{x/\beta}$$

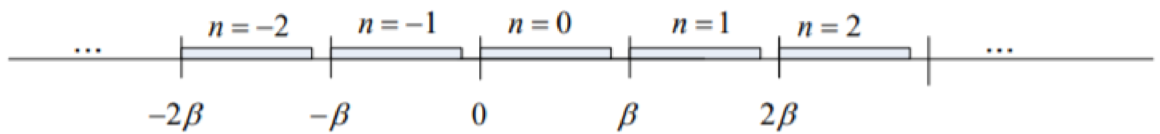
Dato un numero $x \in \mathbb{Z}$ (enunciato generico, noi utilizzeremo questo teorema in particolare coi naturali) e un numero $\beta \in \mathbb{N}$ (con $\beta > 0$), esiste ed è unica la coppia di numeri q, r tale che

$$\boxed{x = q \cdot \beta + r}$$

dove $q \in \mathbb{Z}$ è il quoziente, $r \in \mathbb{N}$ il resto ($0 \leq r < \beta$).

Dimostrazione

- Abbiamo la divisione x/β dove q è il quoziente ed r il resto.
- **Esistenza**:
  - Sappiamo che un qualunque sistema unidimensionale può essere rappresentato geometricamente attraverso una retta orizzontale. Rappresentiamo $\mathbb{Z}$ come una retta e dividiamola in intervalli β. Ogni intervallo sarà definito, dato un numero $n \in \mathbb{Z}$, nel seguente modo
  $$[n \cdot \beta, (n+1) \cdot \beta[$$
  l'unione di tutti questi intervalli mi restituisce l'insieme $\mathbb{Z}$.



  - Se nessun numero rimane escluso significa che x apparterrà ad uno di questi intervalli: supponiamo che questo sia il $q-$esimo
  $$q \cdot \beta \leq x < (q+1) \cdot \beta$$
  a questo punto siamo certi dell'esistenza del quoziente.
  - Vediamo per quanto riguarda il resto r: possiamo definirlo nel seguente modo
  $$r = x - q \cdot \beta$$
  cioè immaginarlo come la distanza del punto x da uno dei due estremi dell'intervallo (quale estremo dipende se stiamo considerando numeri positivi o numeri negativi). Affermando ciò sono sicuro che $0 \leq r < \beta$. Esiste sempre anche il resto r.
- **Unicità**:
  - Adesso vogliamo dimostrare l'unicità della coppia di numeri q, r. Supponiamo esistano due coppie (q_1, r_1 e q_2, r_2) diverse tali che
  $$x = q_1 \cdot \beta + r_1 \qquad x = q_2 \cdot \beta + r_2$$
  con $q_i \in \mathbb{Z}$ e $0 \leq r_i < \beta$.
  - Se $q_1 \cdot \beta + r_1 = q_2 \cdot \beta + r_2$ allora
  $$(q_1 - q_2) \cdot \beta = r_2 - r_1$$
  se per ipotesi $0 \leq r_i < \beta$ avremo (nei casi limite ho $r_2 = 0$ e $r_1 = \beta$, oppure $r_2 = \beta$ ed $r_1 = 0$)
  $$-\beta < (r_2 - r_1) < \beta$$
  Quindi
  $$-\beta < (q_1 - q_2) \cdot \beta < \beta$$
  semplificando (possiamo farlo, $\beta \neq 0$) otteniamo $-1 < (q_1 - q_2) < 1$. Segue $q_1 = q_2$
  - Se $q_1 = q_2$ avremo anche $r_1 = r_2$, contro l'ipotesi. Abbiamo dimostrato l'unicità.

■

P.S Questa unicità è garantita dal fatto che $0 \leq r \leq \beta - 1$

P.P.S Se io ho $x \in \mathbb{N}$ avrò $q \in \mathbb{N}$.

24.3 Notazione per indicare quoziente e resto

Quoziente

Per indicare il quoziente parleremo di *parte intera inferiore della frazione* x/β

$$q = \left\lfloor \frac{x}{\beta} \right\rfloor$$

Resto

Per indicare il resto parleremo di x *modulo* β

$$r = |x|_\beta$$

Le proprietà collegate sono importantissime. Prendiamo la definizione di Corsini: *Sia* x *un numero intero (o naturale) e* β *un numero naturale maggiore di 1 e sia* N_β *l'insieme dei numeri naturali seguente*

$$N_\beta = \{0, 1, \ldots, \beta - 1\}$$

Con x *modulo* β *si intende l'elemento di* N_β *il cui valore coincide con quello di* x *se* $0 \leq x < \beta$ *ovvero con quello del numero naturale ottenuto da* x *aggiungendovi (se negativo) o sottraendovi (se positivo)* β *o multipli di* β*.*

Esempi di moduli $|9|_7 = |9 - 7|_7 = |2|_7 = 2 \qquad |-5|_7 = |-5 + 7|_7 = |2|_7 = 2$

24.4 Proprietà dell'operatore modulo

L'operatore modulo presenta delle proprietà fondamentali da sapere per il seguito. Supponiamo di avere un numero $\alpha \in \mathbb{N}$ con $\alpha > 0$

24.4.1 Prima proprietà

$$\boxed{|x + k\alpha|_\alpha = |x|_\alpha \quad \text{con } k \in \mathbb{Z}}$$

Dim. Per dimostrarlo (ricordiamo che stiamo facendo la divisione x/α) ci basta porre

$$x = \left\lfloor \frac{x}{\alpha} \right\rfloor \alpha + |x|_\alpha$$

aggiungere $k\alpha$ in entrambi i membri ottenendo

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

sapendo che nel secondo membro $\left\lfloor \frac{x}{\alpha} \right\rfloor + k$ è un intero e $|x|_\alpha$ un numero compreso tra 0 e $\alpha - 1$. Si trova che per l'unicità del teorema visto prima gli elementi consistono, rispettivamente, nel quoziente e nel resto della divisione per α di $x + k\alpha$. ■

Conseguenza (dal libro di Corsini) Diciamo in modo esplicito che

$$\boxed{|k \cdot \alpha|_\alpha = 0}$$

24.4.2 Seconda proprietà

$$\boxed{|x \pm y|_\alpha = ||x|_\alpha \pm |y|_\alpha|_\alpha}$$

Dim. Si va a sostituire x ed y, ottenendo

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

quindi

$$= ||x|_\alpha + |y|_\alpha + \alpha \cdot (q_x + q_y)|_\alpha$$

sfruttando la prima proprietà possiamo ricondurci senza grossi problemi alla tesi. ■

24.4.3 Terza proprietà

$$\boxed{|x \cdot y|_\alpha = ||x|_\alpha \cdot |y|_\alpha|_\alpha}$$

Dim. Anche in questo caso sostituiamo x ed y per dimostrare e applicando nuovamente la prima proprietà ci riconduciamo alla tesi. ■

24.4.4 Quarta proprietà (dal libro di Corsini)

$$\boxed{|-x|_\beta = |(\beta - 1) \cdot |x|_\beta|_\beta = |-|x|_\beta|_\beta}$$

utilizzeremo questa proprietà nel calcolo dell'opposto di un numero intero.

Dim. Teniamo conto che

$$|-x|_\beta = |-1 \cdot x|_\beta = ||-1|_\beta \cdot |x|_\beta|_\beta$$

possiamo dire che

$$|-x|_\beta = |(\beta - 1) \cdot |x|_\beta|_\beta = |\beta \cdot |x|_\beta - |x|_\beta|_\beta = |-|x|_\beta|_\beta$$

■

24.5 Pippe personali sui segni

Attenzione ai segni! Si ricordi come l'ave maria che

$$\boxed{|-x|_\beta \neq |x|_\beta}$$

Attenzione La mia abitudine, non siete obbligati a seguirla, è di semplificare il contenuto del modulo utilizzando la prima proprietà del modulo. Dopo aver semplificato applico la definizione di Corsini del modulo.

Primo esempio

- $|13|_7 = |7 \cdot 1 + 6|_7 = \boxed{|6|_7 = 6}$
- $|-13|_7 = |7 \cdot (-1) - 6|_7 = \boxed{|-6|_7 = 1}$

Perchè abbiamo queste differenze? Ricordiamo che il modulo può consistere solo in uno dei valori appartenenti all'insieme N_β.

$$N_\beta = \{0, 1, \dots, \beta - 1\}$$

Ricordiamo anche che il resto consiste in un numero naturale ottenuto da x aggiungendo o togliendo β o multipli di β.

- In caso di numeri negativi si aggiunge β o multipli di β per ottenere un elemento dell'insieme N_β.

  $$|-6|_7 = -6 + 7 = 1$$

- In caso di numeri positivi si sottrae β o multipli di β per ottenere un elemento dell'insieme N_β. In questo caso non dobbiamo farlo perchè $x \in N_\beta$.

Secondo esempio

Prendiamo un altro esempio, un esercizio di un pretest passato

$$|31 - 44|_7 =?$$

1. $|31 + 26|_7$
2. $|31|_7 + |44|_7$
3. $|44 - 31|_7$
4. *Nessuna delle precedenti*

Possiamo escludere direttamente la terza visto che $|13|_7 \neq |-13|_7$. A questo punto quello che mi conviene fare è calcolare direttamente il risultato e confrontarlo con quello delle risposte. Se non c'è coincidenza nè con la prima nè con la seconda allora la risposta sarà la quarta.

- $|31 - 44|_7 = |-13|_7 = 1$
- $|31 + 26|_7 = |57|_7 = |8 \cdot 7 + 1|_7 = |1|_7 = 1$. Abbiamo già trovato la risposta!!!
- Per sfizio calcoliamo l'altra:

  $$|31|_7 + |44|_7 = |7 \cdot 4 + 3|_7 + |6 \cdot 7 + 2|_7 = |3|_7 + |2|_7 = 5$$

Terzo esempio

$$||31|_5 + |-12|_5|_5 = ?$$

1. 0
2. 4
3. *Non si può fare, perchè -12 non è naturale* (GRANDE TRANELLO)
4. *Nessuna delle precedenti*

Calcoliamo

$$\begin{aligned} ||31|_5 + |-12|_5|_5 &= ||5 \cdot 6 + 1|_5 + |5 \cdot (-2) - 2|_5|_5 = \\ &= ||1|_5 + |-2|_5|_5 = \\ &= |1 + 3|_5 = 4 \end{aligned}$$

La risposta è la 2.

Quarto esempio

Continuiamo le nostre pippe mentali con quest'altro quesito

$$-|X|_m \leq |-X|_m$$

1. *Vero*
2. *Falso*
3. *Non si può dire*

Questa è roba da sticker GRANDE TRANELLO di Pistolesi. Abbiamo detto che il resto è un naturale appartenente ad N_β. Se metto il segno negativo nel primo membro è chiaro che quello sarà sempre minore o uguale rispetto al secondo membro. Segue che la risposta giusta è la prima.

Quinto esempio con ulteriori riflessioni

$$|2 \cdot X|_m = 2|X|_m$$

1. *Vero*
2. *Falso*
3. *Nessuna delle precedenti*

Le proprietà non ci dicono nulla sul portare cose fuori dal modulo. Supponiamo $m = 2$:

- $|2 \cdot X|_2 = 0$ per la prima proprietà bis
- $2|X|_2$, non possiamo dire subito che il risultato è 0. Cosa succede se ho un numero X non multiplo di 2? Prendiamo 5:

  $$|2 \cdot 5|_2 = ||2|_2 \cdot |5|_2|_2 = 0 \qquad 2 \cdot |5|_2 = 2 \cdot 1 = 2$$

  chiaramente l'uguaglianza non è valida con un numero non multiplo di 2.

24.6 Rappresentazione dei numeri in qualunque base β

Il teorema della divisione con resto permette di individuare la rappresentazione di un qualunque numero naturale in una qualunque base β. Possiamo ottenere questa rappresentazione, se partiamo dalla base 10, mediante l'algoritmo **DIV&MOD** (o algoritmo della divisioni successive, o algoritmo MOD&DIV).

- Ho un numero A e voglio rappresentarlo in base β.
- Si effettuano una serie di divisioni: ogni volta segnamo quoziente e resto.
- Ogni divisione dopo la prima prende come quoziente l'ultimo quoziente calcolato.
- Si continua finchè non otterrò quoziente nullo.

La $n-$upla di resti ottenuta, letta dal basso verso l'alto (cioè dall'ultimo al primo resto trovato) costituisce l'insieme di cifre che rappresentano il numero A in base β.

Dimostrazione Ci basta sostituire svariate volte. Sappiamo che $A = a_0 + \beta q_1$, sostituiamo

$$A = a_0 + \beta(a_1 + \beta(a_2 + \beta(\dots)))$$

se riordiniamo gli elementi otteniamo la sommatoria $A = \sum_{i=0}^{n-1} \alpha_i \beta^i$ ■

Cosa c'entra il teorema della divisione con resto? L'algoritmo DIV&MOD si basa sullo svolgimento di divisioni. Grazie al teorema possiamo dire che la rappresentazione del numero A in base β è effettivamente unica!

Esempio $N \equiv (38)_{10},\ \beta = 6$

$$\begin{aligned} & q_0 = 38 & & \\ & q_1 = 38 \text{ DIV } 6 = 6 & & a_0 = 38 \text{ MOD } 6 = 2 \\ & q_2 = 6 \text{ DIV } 6 = 1 & & a_1 = 6 \text{ MOD } 6 = 0 \\ & q_3 = 1 \text{ DIV } 6 = 0 & & a_1 = 1 \text{ MOD } 6 = 1 \end{aligned}$$

$N \equiv (a_2 a_1 a_0)_6 = (102)_6$

24.7 Rappresentazione di naturali su un numero di cifre finito

24.7.1 Quanti numeri possiamo rappresentare?

Data una base β e un numero di cifre n individuiamo β^n numeri possibili. Lo abbiamo già visto a *Fondamenti di programmazione* parlando di intervalli di rappresentazione. Ottengo questo numero moltiplicando, per ogni cifra, il numero di cifre possibili (precisamente β)

$$\beta \cdot \beta \cdot \dots \cdot \beta = \beta^n$$

Attenzione Attenzione a non confondere il numero di numeri rappresentabili (β^n) dal numero più grande rappresentabile. Teniamo conto che tra i β^n numeri possibili abbiamo anche zero.

24.7.2 Numero più grande rappresentabile?

Il numero più grande rappresentabile in base β date n cifre consiste in $\beta^n - 1$. Possiamo verificarlo ponendo $a_i = \beta - 1, \forall i$.

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

con la differenza tra le due sommatorie annullo tutti i termini tranne β^n (per la prima sommatoria) e $\beta^0 = 1$ (per la seconda).

24.7.3 Numero di cifre richieste per rappresentare un numero

Il numero di cifre necessario per rappresentare il numero A è il minimo n per cui

$$\beta^n - 1 \geq A$$

segue $\beta^n \geq A + 1$: tale valore è $n = \lceil \log_\beta (A + 1) \rceil$ (arrotondo sempre per eccesso). Cosa intendiamo?

- Supponiamo di avere 9 e di volerlo rappresentare in base 2.
- Ricerco il minimo valore n per cui sia valida la disuguaglianza detta prima:
  - $2^1 - 1 = 1 < 9$
  - $2^2 - 1 = 3 < 9$
  - $2^3 - 1 = 7 < 9$
  - $2^4 - 1 = 15 > 9$. Ecco qua, servono $n = 4$ cifre!

24.8 Osservazione sugli esercizi della dispensa

Dagli esercizi alle pagine 8 e 9 della dispensa individuiamo che:

- Sia $A = (a_{n-1} \ldots a_0)_\beta$ un numero su n cifre in base β. Sia γ un sottomultiplo di β, quindi $\beta = k \cdot \gamma$. Possiamo dire, con queste condizioni, che

  $$|A|_\gamma = 0 \Longleftrightarrow |a_0|_\gamma = 0$$

  Esempio: se siamo in base $\beta = 10$ troviamo i criteri di divisibilità studiati alle elementari per i numeri naturali divisibili per 2 e per 5 (gli unici sottomultipli di 10).

- Sia $A = (a_{n-1} \ldots a_0)_\beta$ un numero su n cifre in base β. Sia γ sottomultiplo di $\beta - 1$. Possiamo dire, con queste condizioni, che

  $$|A|_\gamma = 0 \Longleftrightarrow \left| \sum_{i=0}^{n-1} a_i \right|_\gamma = 0$$

  Esempio: se $\beta = 10$ ottengo il criterio di divisibilità per 3 dei numeri naturali.

- Sia $A = (a_{n-1} \dots a_0)_\beta$ un numero su n cifre in base β. Sia $\gamma = \beta + 1$. Possiamo dire, con queste condizioni, che

$$|A|_{\beta+1} = 0 \Longrightarrow \left| \sum_{i=0}^{n-1} (-1)^i \cdot a_i \right|_{\beta+1} = 0$$

  la differenza tra la somma delle cifre di posizione pari e le cifre di posizione dispari è nulla.
  Esempio: con $\beta = 10$ ottengo il poco noto criterio di divisibilità per 11 dei numeri naturali.

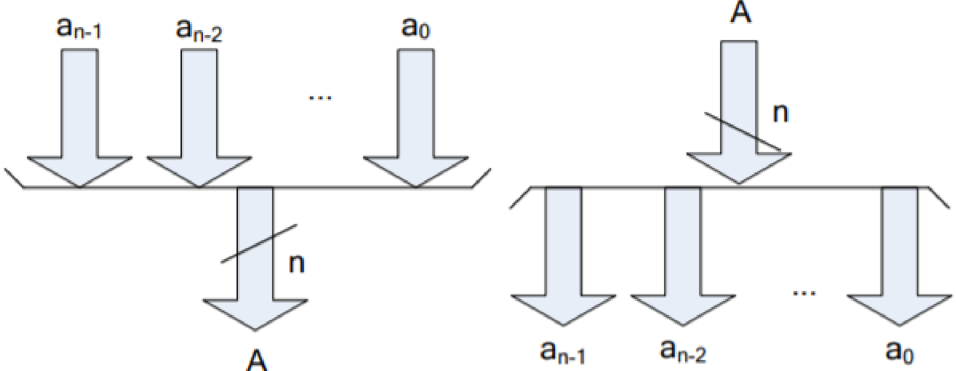
24.9 Elaborazione di numeri naturali tramite reti combinatorie

Il nostro obiettivo è costruire reti logiche che elaborino numeri naturali rappresentati in una data base β. Ovviamente utilizzeremo reti combinatorie: lo stato di uscita dipende esclusivamente dallo stato di ingresso, cioè dagli operandi!

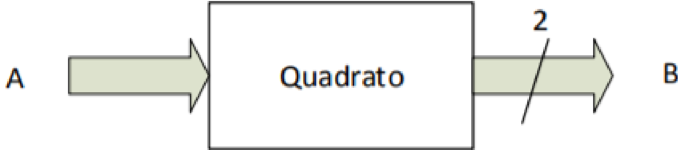
Cosa faremo

- Prenderemo in esame operazioni aritmetiche fornendo descrizioni indipendenti dalla base β. Ci chiederemo, in particolare, quante cifre ci servono per esprimere il risultato dell'operazione e come ottenere queste cifre.
- Sfruttando le proprietà della notazione posizionale scomporremo le operazioni in blocchi elementari
- Descriveremo le reti logiche che implementano i blocchi elementari in base 2.

Necessaria nuova notazione Per poter parlare di tutto questo è necessario adottare una notazione che svincoli i miei ragionamenti da una particolare base β. Presenti delle immagini a pagina 10 della dispensa di Aritmetica.



Esempio



Da pagina 10 della dispensa di aritmetica è presente un esempio di rete combinatoria che manipola numeri naturali. Precisamente la rete indicata restituisce il quadrato di un numero in base β indicato in ingresso. Le strade adottabili sono due, tenendo conto che in un qualunque caso faremo ricorso a variabili logiche:

- Adottare una codifica BCD (*Binary-coded Decimal*) dove un certo gruppo di variabili logiche in ingresso o in uscita rappresenta la codifica binaria di una delle cifre possibili in base β. Si dice, in questo caso, che la rete **opera in base** 10.

A	$b_1 b_0$
0	00
1	01
2	04
...	...
8	64
9	81

BCD

BCD(A)	BCD(b_1) BCD(b_0)
0000	0000 0000
0001	0000 0001
0010	0000 0100
...	...
1000	0110 0100
1001	1000 0001

- Lavorare in base $\beta = 2$ come abbiamo fatto fino ad ora. In questo caso una cifra equivale a una variabile logica.

Si noti che le sequenze binarie ottenute saranno diverse.

Capitolo 25

Giovedì 22/10/2020

25.1 Complemento

Il complemento è un'operazione che pur apparendo inutile è in realtà estremamente importante. Abbiamo un numero A

$$A = (a_{n-1} \dots a_0)_\beta$$

il tutto in n cifre. Segue $0 \leq A < \beta^n$. Si definisce il complemento di un numero A in base β su n cifre quanto segue

$$\overline{A} \triangleq \beta^n - 1 - A$$

dove $\beta^n - 1$ consiste nel massimo numero assumibile in base β su n cifre. Come vedremo il numero di cifre n è importante: con un numero di cifre diverse potrei ottenere un risultato diverso!

- $\overline{(1034)_{10}} = 9999 - 1034 = (8965)_{10}$
- $\overline{(1034)_5} = 4444 - 1034 = (3410)_5$
- $\overline{(001034)_{10}} = 999999 - 001034 = (998965)_{10}$

Vogliamo sintetizzare una rete dove, date in ingresso le cifre della rappresentazione di A, si ottengano in uscita le cifre della rappresentazione di $\overline{A}$. Poniamoci qualche domanda:

- **Su quante cifre sta il risultato?**
  Sappiamo che $0 \leq A < \beta^n$: segue che $0 \leq \overline{A} < \beta^n$.
  Il complemento è sicuramente rappresentabile su n cifre.

- **Come si trovano le cifre del risultato?**
  Ricavo, usando la definizione, che

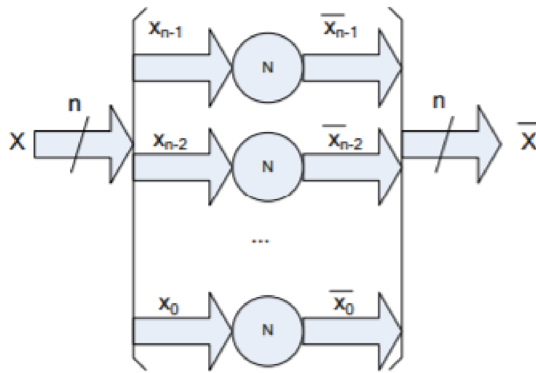
  $$\overline{A} = \sum_{0}^{n-1}(\beta - 1)\beta^i - \sum_{0}^{n-1} \alpha_i \beta^i = \sum_{0}^{n-1}(\beta - 1 - \alpha_i)\beta^i$$

  dove $\beta - 1 - \alpha_i$ è una cifra in base β compresa tra 0 e $\beta - 1$. Troviamo che

  $$\overline{\alpha_i} = \beta - 1 - \alpha_i \Longrightarrow \overline{A} \equiv (\overline{a_{n-1}}\, \overline{a_{n-2}} \dots \overline{a_0})_\beta$$

cioè otteniamo il complemento di un numero in base β su n cifre complementando individualmente ogni singola cifra $i-$esima. Non a caso, $\beta - 1 - \alpha_i$ consiste proprio nel complemento della singola cifra α_i in base β.

25.1.1 Circuito logico per l'operazione



Lo schema di circuito da utilizzare, come già detto, deve essere valido per qualunque base (ricordiamo che il numero di variabili logiche da utilizzare dipenderà sia dalla base adottata che dal numero di cifre massimo). Dall'immagine presente a pagina 13 della dispensa di Aritmetica rappresentiamo la seguente strategia:

- Pongo in ingresso nel circuito un numero X avente n cifre;
- All'interno del circuito il numero sarà scomposto in tutte le sue cifre;
- Applichiamo ad ogni cifra l'operazione di complemento su singola cifra;
- Unifichiamo i risultati di queste operazioni di complemento per restituire, in uscita, $\overline{X}$.

La cosa diventa ancora più semplice se lavoriamo in base $\beta = 2$: le variabili logiche coincidono con le cifre, che possono avere come valore 0 o 1. Questo significa che in base 2 l'operazione di complemento su singola cifra sarà svolta da una rete elementare dove

- dato 0 si ha in uscita 1;
- dato 1 si ha in uscita 0.

la rete elementare sarà ovviamente la porta NOT!

25.1.2 Complemento di numeri in base 10 con codifica *eccesso 3*

Il secondo punto dell'esercizio a pagina 13 della dispensa pone una codifica particolare per i numeri in base 10 ad n cifre.

- Rappresento ogni cifra a in base 10 come la cifra $a + 3$ in base 2. A situazione normale abbiamo bisogno di quattro cifre, segue che la cosa non sarà un problema.
- Osservo che il circuito di complemento di una singola cifra in base 10 è una barriera di 4 invertitori. Per questo si parla di *codifica autocomplementante*.

	x_3	x_2	x_1	x_0	*	z_3	z_2	z_1	z_0
(0+3)	0	0	1	1	(9-0=9)	1	1	0	0
(1+3)	0	1	0	0	(9-1=8)	1	0	1	1
(2+3)	0	1	0	1	(9-2=7)	1	0	1	0
(3+3)	0	1	1	0	(9-3=6)	1	0	0	1
(4+3)	0	1	1	1	(9-4=5)	1	0	0	0
(5+3)	1	0	0	0	(9-5=4)	0	1	1	1
(6+3)	1	0	0	1	(9-6=3)	0	1	1	0
(7+3)	1	0	1	0	(9-7=2)	0	1	0	1
(8+3)	1	0	1	1	(9-8=1)	0	1	0	0
(9+3)	1	1	0	0	(9-9=0)	0	0	1	1

$z_0 = \overline{x_0} \quad z_1 = \overline{x_1} \quad z_2 = \overline{x_2} \quad z_3 = \overline{x_3}$

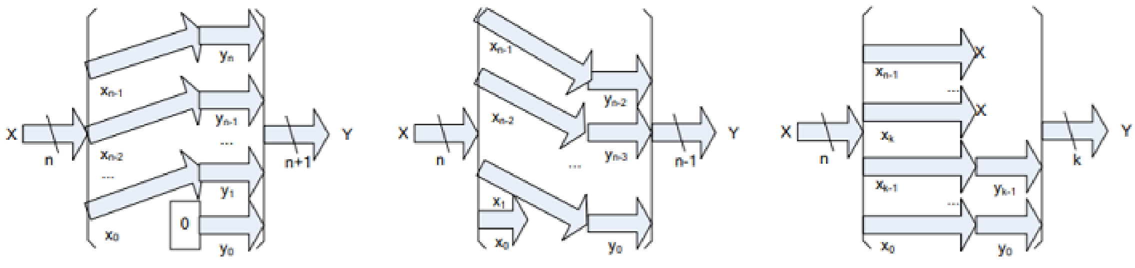
25.2 Moltiplicazione e divisione per una potenza della base

Sfruttando le proprietà della notazione posizionale possiamo effettuare questi calcoli con procedimenti aventi complessità nulla! Pensiamo ai seguenti esempi:

- 25×1000: svolgo 25×1 e aggiungo tre zeri in fondo
- $2563/100$: traslo una virgola di due posizioni da destra verso sinistra, ottengo che 25 è il quoziente e 63 il resto.

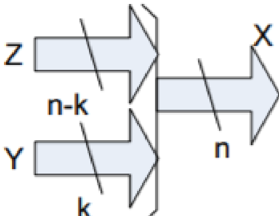
Le reti che permettono di implementare queste operazioni sono dette *reti di shift*.

- **Moltiplicazione** di un numero su n cifre per β^k: costruiamo un nuovo numero di $n + k$ cifre dove le k meno significative valgono 0. Se moltiplico per β^1 scarto tutte le volte la cifrà più significativa.
- **Divisione** di un numero su n cifre per β^k:
  - Quoziente: numero costituito dalle $n-k$ cifre più significative del numero di partenza. Se divido per β^1 scarto tutte le volte la cifra meno significativa.
  - Resto: numero costituito dalle k cifre meno significative del numero di partenza.



D'ora in avanti sarà considerato errore usare porte logiche per moltiplicare e dividere un numero per β^k.

25.2.1 Conseguenza: operazioni di concatenamento e scomposizione



Supponiamo di avere due numeri Y e Z, rispettivamente a k ed $n-k$ cifre, definiamo nel seguente modo l'operazione di concatenamento

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

che produce un numero su n cifre. Poniamo le $n-k$ cifre di Z accanto alle k cifre di Y. L'operazione è a costo nullo.

Scomposizione Risulta possibile fare a costo nullo anche l'operazione inversa di scomposizione di un numero su n cifre in due blocchi di k ed $n-k$ cifre.

25.3 Estensione di campo sui numeri naturali

L'estensione di campo è l'operazione con cui si intende rappresentare un numero naturale utilizzando un numero di cifre maggiore. In qualunque base si fa la stessa cosa: si mette il numero necessario di zeri in testa. Dato il seguente numero

$$X \equiv (x_{n-1} \dots x_0)_\beta$$

definisco X^{EST} come il numero che vale quanto X ma rappresentato su $n+1$ cifre

$$X^{\mathrm{EST}} \equiv (0\, x_{n-1} \dots x_0)_\beta$$

questa è l'unica rappresentazione possibile (visto che è l'unica dai teoremi imparati).

Attenzione agli interi Il procedimento nei numeri interi cambia!

25.4 Addizione di numeri naturali

L'algoritmo di addizione per i numeri naturali è lo stesso che conosciamo fin dalle elementari (basato sulle proprietà della notazione posizionale) e che è valido per una qualunque base β

- Sommare le cifre di pari posizione singolarmente, partendo dalla meno significativa, andando verso sinistra;
- se la somma di due cifre non è rappresentabile con una sola cifra si utilizza il riporto.
- **il riporto vale SEMPRE** 0 **o** 1.

Supponiamo di avere due numeri X, Y in base β su n cifre e un carry in ingresso C_{in} che sappiamo essere $0 \leq C_{in} \leq 1$. L'addizione consiste in

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

il fatto di avere un carry in ingresso mi permette di rendere l'operazione modulare (cioè divisibile in moduli, la cosa che abbiamo visto in Assembler per sommare operandi con più di 32 bit).

Su quante cifre sta il risultato? Sappiamo che

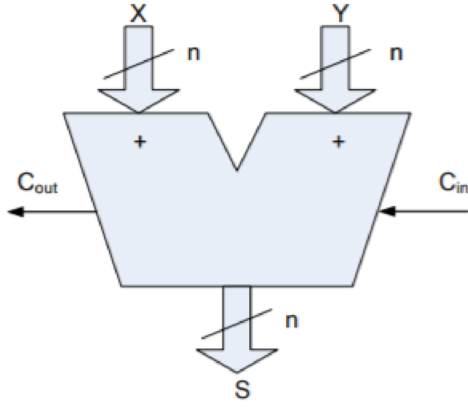
$$0 \leq X + Y + C_{in} \leq (\beta^n - 1) + (\beta^n - 1) + 1 \leq 2\beta^n - 1 \leq \beta^{n+1} - 1$$

possiamo dire l'ultima cosa poichè $\beta \geq 2$. Arriviamo alla seguente conclusione

- su $n+1$ bit la somma di naturali sarà sempre rappresentabile. La $n+1$-esima cifra consiste nel riporto dell'ultima somma (che abbiamo già detto è uguale a 0 o 1), detto **riporto uscente**.
- su n bit la somma di naturali potrebbe non essere rappresentabile.

Quindi *la somma di due numeri naturali espressi in base β su n cifre, più un eventuale riporto entrante che vale zero o uno, produce un numero naturale che è sempre rappresentabile con $n+1$ cifre in base β, l'$(n+1)^{ma}$ delle quali, detta riporto uscente, può essere soltanto zero o uno.*

Sommatore in base β a n cifre Il circuito che utilizzeremo è detto sommatore.



Abbiamo in ingresso:

- due operandi, con il numero di variabili logiche necessarie, entrambi ad n bit.
- il riporto in ingresso C_{in}

e in uscita

- il risultato in uscita, sempre ad n bit
- il riporto in uscita C_{out}.

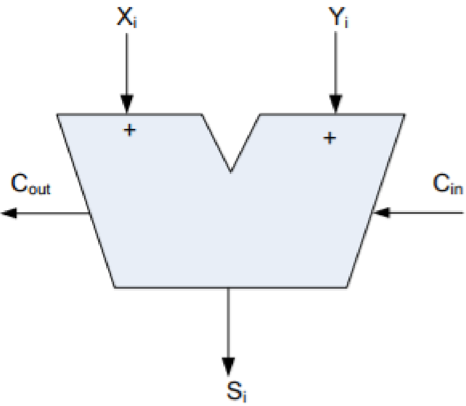
Osservazione relativa ai numeri naturali

- Se il riporto in uscita è zero il risultato della somma è rappresentabile su n bit.
- Se il riporto in uscita è uno il risultato della somma NON è rappresentabile su n bit.

Dimensionamento del sommatore Vizio di gioventù è immaginare il dimensionato degli operandi e del risultato all'interno del sommatore diversi. Ciò è sbagliato: i due operandi e il risultato hanno tutti lo stesso numero di n bit. Seguono alcune riflessioni:

- Se gli addendi hanno un numero di cifre differenti, per esempio n ed m con $n < m$, dovremo estendere l'operando con n bit in modo tale che abbia m cifre (si aggiungono $m - n$ zeri in testa).
- Se gli addendi sono su n cifre e vogliamo avere la certezza che la somma sia sempre rappresentabile dobbiamo estendere gli operandi utilizzando un sommatore ad $n + 1$ cifre.

25.4.1 Full adder in base 2



Il *full adder* consiste in un sommatore ad una cifra. Questi mi permettono di sintetizzare il circuito sommatore introdotto prima. Possiamo immaginarci una scomposizione della somma in base β su n cifre in somme in base β su una sola cifra: ciò è possibile sfruttando il carry di ingresso detto prima (ricordiamoci, abbiamo parlato di operazione modulare).

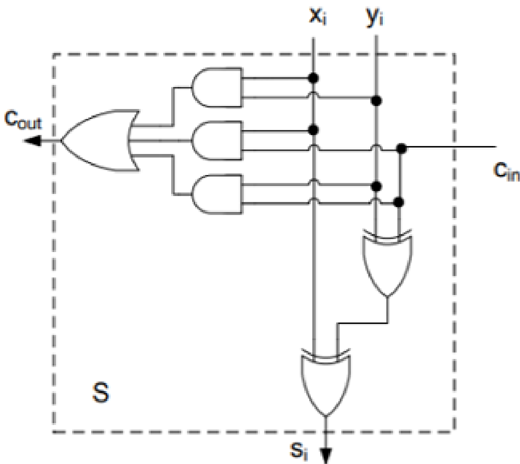
- Si eseguono le somme andando dalla cifra meno significativa alla più significativa
- Si propaga il riporto attraverso il riporto entrante C_{in} e il riporto uscente C_{out}.

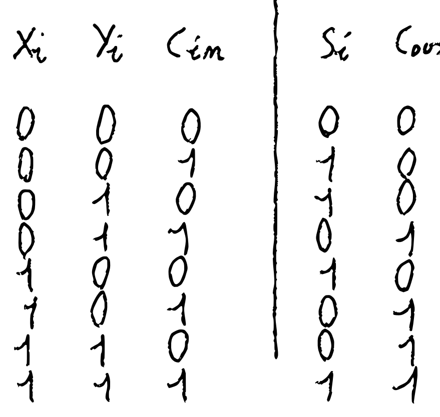
Questo montaggio di sommatori a una cifra costituisce il *ripple carry*. Osserviamo che questa rete ha 3 ingressi e 2 uscite: possiamo sintetizzarla coi metodi che già conosciamo. Attraverso le mappe di Karnaugh individuiamo

- Una facile sintetizzazione per C_{out} attraverso 3 porte AND a 2 ingressi e una porta OR a tre ingressi. Dalla mappa di Karnaugh troviamo

$$C_{out} = x_i y_i + y_i C_{in} + x_i C_{in}$$

- Una situazione un po' più complicata relativamente alla S_i: come dice Stea non ce lo prescrive il medico di dover usare la forma SP, pertanto ricorriamo a una porta XOR a 3 ingressi (due porte XOR a cascata). Scegliamo la porta XOR poichè permette di riconoscere un numero dispari di 1.





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

FULL ADDER

SINTESI A COSTO MINIMO IN FORMA SP:

$$S_i = \overline{X_i}\, Y_i\, \overline{C_{in}} + \overline{X_i}\, \overline{Y_i}\, C_{in} + X_i\, Y_i\, C_{in} + X_i\, \overline{Y_i}\, \overline{C_{in}}$$

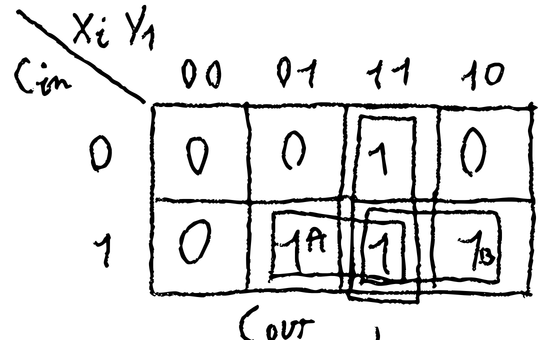
C_{in} \ $X_i Y_1$	00	01	11	10
0	0	1	0	1
1	1	0	1	0

S_i

ATTENZIONE AL NUMERO DI 1:
- NUMERO DISPARI : $S_i = 0$
- NUMERO PARI : $S_i = 1$

PORTA XOR

"NON CE LO DICE IL DOTTORE DI FARE UNA SINTESI IN FORMA SP"



C_{in} \ $X_i Y_1$	00	01	11	10
0	0	0	1	0
1	0	1 A	1	1 B

C_{out}

	X_i	Y_i	C_{in}
A	–	1	1
B	1	–	1
C	1	1	–

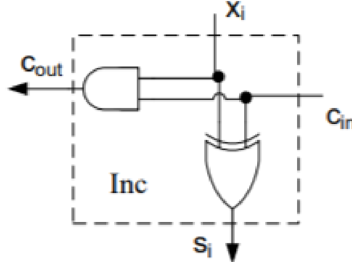
→ ATTENZIONE AL SOTTOCUBO MANCANTE STILE PACMAN: HA LE STESSE COORDINATE DI C. NE METTO UNO SOLO TRA I DUE

I 3 SOTTOCUBI SONO TUTTI ESSENZIALI

$$C_{out} = Y_i \cdot C_{in} + X_i \cdot C_{in} + X_i \cdot Y_i$$

PORTA OR A 3 INGRESSI, 3 PORTE AND A 2 INGRESSI

25.4.2 Incrementatore (*half adder*)

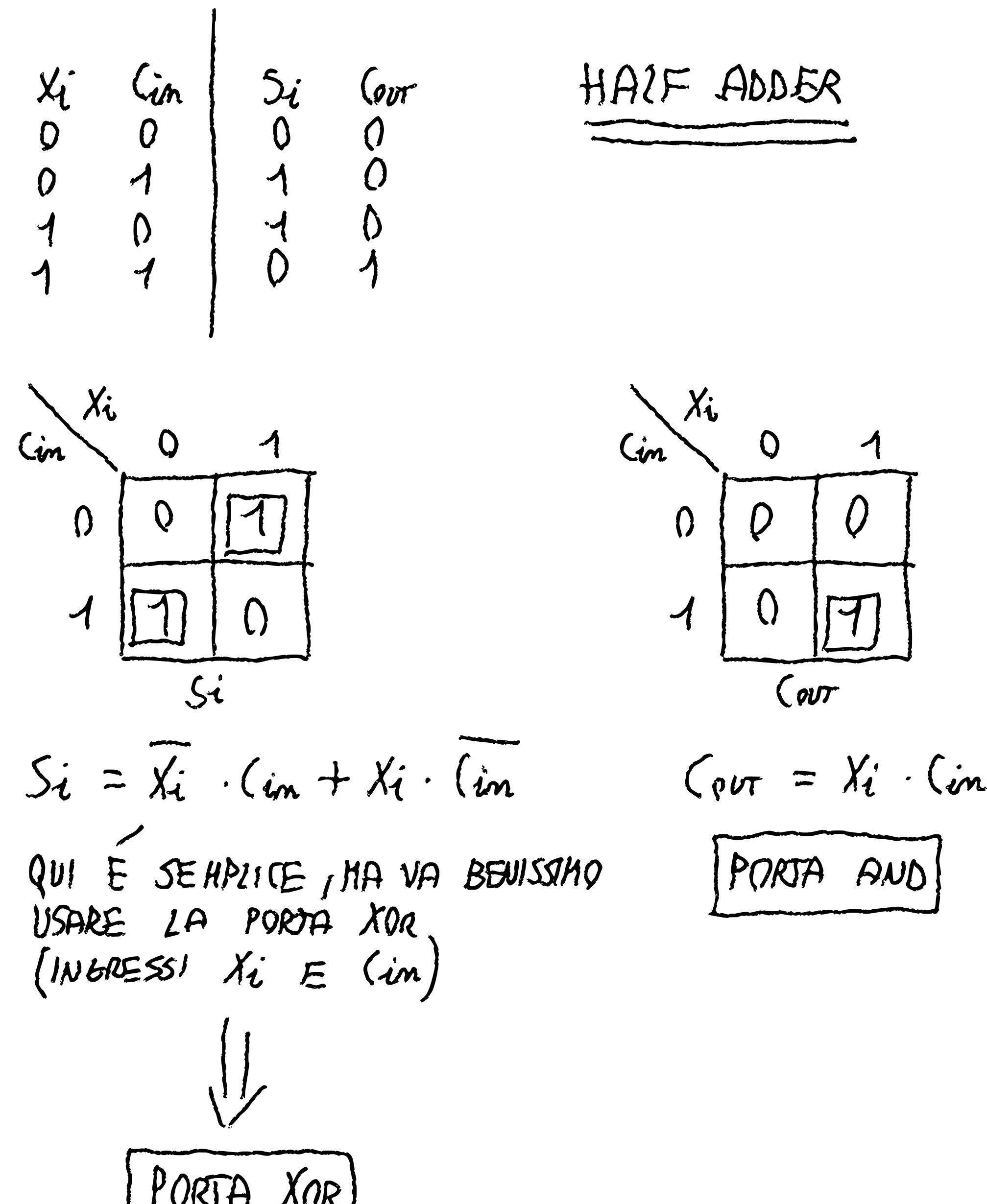


L'incrementatore consiste in un circuito che somma C_{in} ad un numero dato in ingresso. Questa cosa può essere immaginata come un caso particolare di somma di riporto tra addendi ad n cifre dove uno dei due addendi risulta essere nullo: come prima possiamo sintetizzare questa cosa mediante n moduli full adder. Tutti questi moduli presenteranno un ingresso sempre nullo: segue la possibilità di semplificarli.

Semplicità Questo circuito è più semplice del full adder: abbiamo un solo livello di logica.

Assembler Ricordiamo questi comandi visti in Assembler

```
ADD $1, %AL
INC %AL
```


HALF ADDER
Xi Cin Si Cout
0 0 0 0
0 1 1 0
1 0 1 0
1 1 0 1
Xi
Cin
0 1
0
1
Si
Cout
Si = X̄i · Cin + Xi · C̄in
Cout = Xi · Cin
QUI È SEMPLICE, MA VA BENISSIMO
USARE LA PORTA XOR
(INGRESSI Xi E Cin)
PORTA XOR
PORTA AND

25.5 Sottrazione

L'algoritmo di sottrazione per i numeri naturali è lo stesso che conosciamo fin dalle elementari (basato sulle proprietà della notazione posizionale) e che è valido per una qualunque base β

- Sottrarre le cifre di pari posizione singolarmente, partendo dalla meno significativa, andando verso sinistra;
- se la differenza di due cifre non è rappresentabile con una sola cifra si utilizza il prestito.
- **il prestito vale SEMPRE** 0 **o** 1.

Differenze mentali

- Personalmente ho sempre avuto l'abitudine di passare subito alla colonna successiva in caso di sottrazione non possibile, e ritornarci dopo aver prelevato.
- In questo caso "anticipiamo il resto", cioè poniamo come risultato della sottrazione nella colonna il modulo della differenza. A quel punto passiamo alla colonna successiva, svolgiamo la nuova sottrazione, sottraiamo al risultato appena ottenuto 1.

Cifre necessarie Supponiamo di avere due numeri naturali X, Y tali che $0 \leq X, Y \leq \beta^n - 1$. Abbiamo un prestito in ingresso b_{in} tale che $0 \leq b_{in} \leq 1$. Il risultato della differenza è

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

I naturali non sono un insieme chiuso rispetto alla sottrazion! Segue che il risultato Z potrebbe non essere un numero naturale. Osserviamo che

- con $X = \beta^n - 1$, $Y = b_{in} = 0$ si ha $X - Y - b_{in} \leq \beta^n - 1$
- con $X = 0$, $Y = \beta^n - 1$ e $b_{in} = 1$ si ha $X - Y - b_{in} \geq -\beta^n$

quindi

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

Riflessioni algebriche Recuperiamo quanto detto poco fa

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

immaginiamo Z, il risultato della sottrazione, come quoziente e resto della seguente divisione

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

l'idea è che il resto consista nel risultato della sottrazione (spostiamo la virgola n volte, cioè tante quante le cifre presenti nel numero), mentre il quoziente è uguale a 0. Si osserva, tuttavia, che il valore minimo del quoziente è -1: questo avviene quando l'operazione non è possibile nei numeri naturali.

... $q=-2$ $q=-1$ $q=0$ $q=1$ $q=2$ ...

$-2\beta^n$ $-\beta^n$ 0 β^n $2\beta^n$

Definiamo

- il prestito uscente $-b_{out} = \left\lfloor \frac{X-Y-b_{in}}{\beta^n} \right\rfloor$ dove $b_{out} \in [0,1]$
- il modulo $D = |X - Y - b_{in}|_{\beta^n}$ che consiste nella differenza.

segue

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

Quindi *La differenza D tra due numeri naturali in base β su n cifre, meno un eventuale prestito entrante, produce un numero che, se naturale, è sempre rappresentabile su n cifre in base β. Può inoltre produrre un numero non naturale, nel qual caso, c'è un prestito uscente. In ogni caso il prestito uscente può valere soltanto zero o uno.*

Calcolo della differenza Ricordiamo la definizione di complemento:

$$Y + \overline{Y} = \beta^n - 1$$

Spostando gli elementi tra i due membri posso dire

$$-Y = \overline{Y} - \beta^n + 1$$

Sostituiamo nella formula trovata prima

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

$$(-b_{out}) \cdot \beta^n + D = X + \overline{Y} - \boxed{\beta^n} + 1 - b_{in}$$

$$\boxed{(1 - b_{out})} \cdot \beta^n + D = X + \overline{Y} + \boxed{1 - b_{in}}$$

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

- La differenza può essere ottenuta sommando X ed Y complementato, più un eventuale riporto entrante complementando il prestito entrante.
- Se il riporto uscente della somma, $\overline{b_{out}}$ vale
  - 1, la differenza è un numero naturale pari a D, e il prestito uscente b_{out} vale 0
  - 0, la differenza non è un numero naturale, ed il prestito uscente b_{out} è 1

100 − 010 − 0 = 010
$X - Y - b_{in} = Z$

$X + \overline{Y} + \overline{b_{in}}$
100 + 101 + 1 = 1010

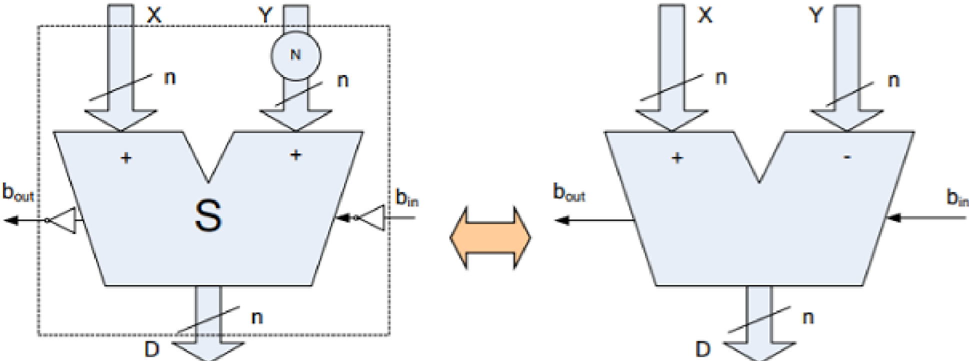
OVERFLOW, RIPORTO USCENTE UGUALE AD 1

010 − 100 − 0 (NO)
$X - Y - b_{in}$

$X + \overline{Y} + \overline{b_{in}}$
010 + 011 + 1 = 110 ??

NIENTE OVERFLOW, RIPORTO USCENTE UGUALE A ZERO

Sottrattore Possiamo costruire un sottrattore utilizzando i sommatori (abbiamo visto dall'espressione che possiamo risolvere l'operazione con addizioni di elementi).



Abbiamo in ingresso

- Il minuendo X
- il sottraendo Y, che sarà complementato con porta NOT ottenendo $\overline{Y}$
- il prestito in ingresso b_{in}, che sarà complementato con porta NOT ottenendo il riporto in ingresso $\overline{b_{in}}$

In uscita otterremo

- il risultato della sottrazione D
- il prestito uscente b_{out}, ottenuto complementando con porta NOT il riporto uscente $\overline{b_{out}}$

La strategia per sintetizzare un sottrattore è la stessa vista col sommatore: si utilizzano sottratori su una cifra e si costituisce un *ripple borrow*.

25.5.1 Osservazione sulla complementazione nei pretest

Errore tipico L'errore tipico di chi risponde a domande che hanno a che fare con la complementazione è il non stare attenti alla base β. Se $\beta \neq 2$ non va bene fare il complemento lavorando sui singoli bit. Dobbiamo utilizzare la formula della definizione

$$\overline{A} \triangleq \beta^n - 1 - A$$

che in caso di codifica BCD (si consideri un sommatore ad una cifra) significa

$$\overline{A} = 10^1 - 1 - A = 9 - A$$

Esempio di domanda sul sottratore Sia $X = 0000, Y = 0001, b_{in} = 0$. Si passi il tutto in un sottrattore in base 10 a una cifra (codifica BCD).

- Sappiamo che $X - Y = X + \overline{Y} + \overline{b_{in}}$.
- $\overline{Y} = \beta^n - 1 - Y = 10^1 - 1 - Y = 9 - Y = 9 - 1 = 8 \equiv (1000)_2$
- Quindi

  $$D = X + \overline{Y} + \overline{b_{in}} = 0000 + 1000 + 1 = 1001$$

  Il riporto uscente della somma è 0, segue $b_{out} = \overline{c_{out}} = 1$.
- Sapendo che $\beta = 10, n = 1$ otteniamo l'intervallo di rappresentabilità

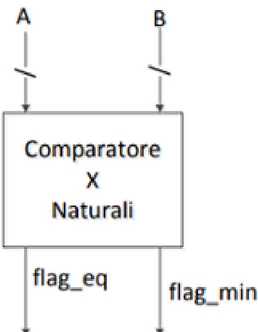
  $$\left[-\frac{\beta^n}{2}; +\frac{\beta^n}{2} - 1\right] \longrightarrow [-5; +4]$$

- Verifichiamo che non ci sia stato overflow e calcoliamo l'intero rappresentato dal numero naturale restituito dal sommatore

  $$d = -(\overline{D} + 1) = -((\beta^n - 1 - D) + 1) = -((10^1 - 1 - 1001) + 1) = -((9 - 9) + 1) = -1$$

  il risultato torna (ci aspettavamo questo numero dai valori iniziali), quindi non c'è overflow.

25.5.2 Comparatore di numeri naturali



Quanto detto vale ESCLUSIVAMENTE per i numeri naturali.

I sottrattori possono essere usati come comparatori. Dati due numeri naturali A e B svolgo la sottrazione tra i due $(A - B)$:

- se $A > B$ il prestito uscente b_{out} sarà uguale a 0
- se $A < B$ il prestito uscente b_{out} sarà uguale ad 1.

Per verificare se $A = B$, inoltre, devo controllare che il risultato della differenza sia uguale al numero $(00 \dots 00)$ con n bit. La scelta più conveniente pare analizzare il numero cifra per cifra.

- Facciamo questo passando l'uscita del sottrattore da una porta NOR con un opportuno numero di ingressi. Avrò

$$\boxed{z = 1 \iff x_0 = x_1 = \cdots = x_n = 0}$$

- **Attenzione**: noi quando vediamo due numeri non andiamo a fare la sottrazione per vedere se sono uguali! Scelta più veloce è fare lo XOR bit a bit delle codifiche di ciascuna cifra e far convergere tutte le uscite di queste porte in una porta NOR.
  - Dallo XOR passano sempre due variabili logiche. Con lo XOR ottengo 0 se i due elementi sono uguali, 1 se i due elementi sono diversi.
  - Se i due numeri sono uguali tutte le porte XOR (tante quante le variabili logiche usate) restituiranno 0.
  - Se passiamo tutti 0 in una porta NOR otteniamo in uscita 1.

25.6 Moltiplicazione

La moltiplicazione si svolge coi seguenti dati:

- un numero naturale X in base β su n cifre
- un numero naturale Y in base β su m cifre
- un numero naturale C in base β su n cifre (per la modularità, come vedremo)

Il risultato della moltiplicazione consiste nel seguente

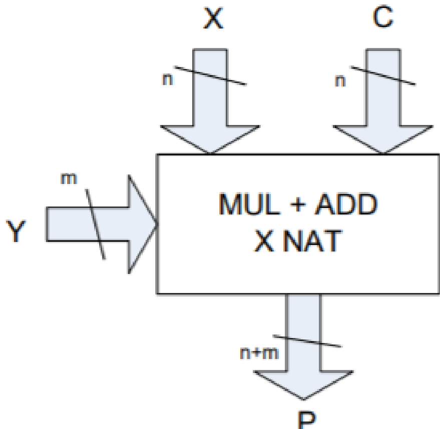
$$P = X \cdot Y + C$$

Quanti bit servono per il risultato? Sapendo che il valore massimo per un numero in base β su n cifre è $\beta^n - 1$ affermiamo quanto segue

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

L'ultima cosa la otteniamo considerando che abbiamo una sottrazione e che β^m sarà sicuramente maggiore di 1. Otteniamo che il risultato della moltiplicazione dovrà stare su $n + m$ cifre!

Rete La rete che ci permette di svolgere la moltiplicazione si chiama **moltiplicatore con addizionatore per naturali**.



Algoritmo delle elementari Anche in questo caso il nostro primo pensiero per la risoluzione di una moltiplicazione va agli algoritmi imparati alle elementari.

- Effettuo moltiplicazioni del primo fattore per ogni cifra del secondo fattore
- I risultati di ciascuno di questi prodotti parziali vengono scritti a partire dal posto occupato dalla cifra per la quale si sta moltiplicando
- tutti i risultati vengono sommati per ottenere il prodotto.

Svolgiamo moltiplicazioni tra un numero ad n cifre per un numero ad una cifra e sommiamo m addendi con eventuali operazioni di shift (che sono a costo nullo) per ottenere il risultato.

Algoritmo alternativo e scomposizione Per sintetizzare una rete dobbiamo pensare a un algoritmo facilmente scomponibile. L'algoritmo precedente è semplice, ma difficile da fare da un punto di vista circuitale (in particolare è complicato sommare m addendi con eventuali traslazioni in un colpo solo). Si osserva che

- la somma è associativa (quindi possiamo scomporla)
- ogni volta che sommiamo due addendi (nell'algoritmo precedente) siamo in grado di determinare una delle cifre del risultato del prodotto (nell'ordine si va dalla cifra meno significativa fino alle ultime cifre).

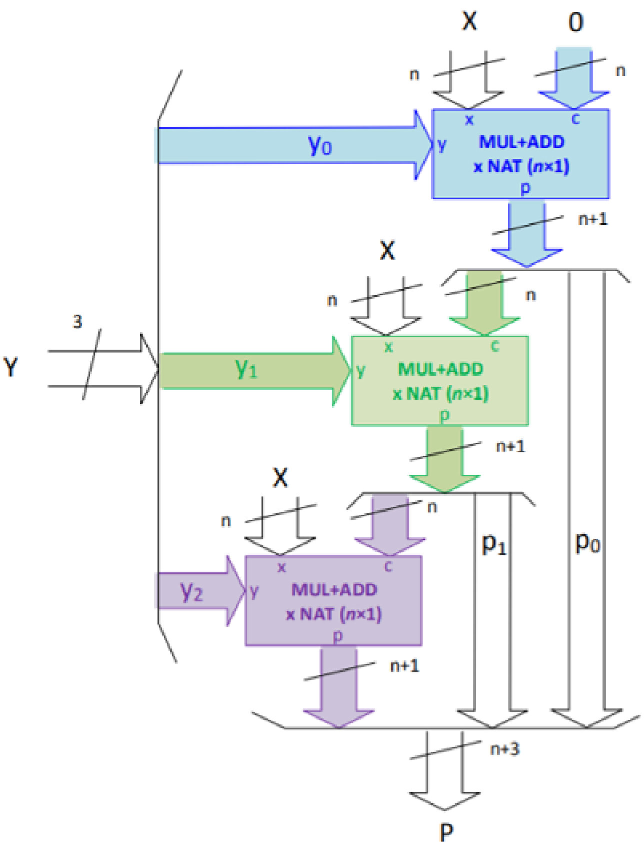
Adesso si capisce perchè abbiamo introdotto C all'inizio: si svolgono addizioni a due a due su $n+1$ cifre. Ovviamente si ottengono i vari elementi moltiplicando il primo fattore ad n cifre con un numero ad una sola cifra. Ogni volta la cifra meno significativa diventa la $i-$esima cifra del prodotto.

Esempio Prendiamo il prodotto 245×131

- Moltiplico 245×1.
- C è inizialmente nullo, quindi $245 + 0 = 245$. 5 è la prima cifra meno significativa del prodotto
- Moltiplico $245 \times 3 = 735$.
- C è uguale a 24, quindi $735+24 = 759$. 9 è la seconda cifra meno significativa del prodotto
- Moltiplico $245 \times 1 = 245$
- C è uguale a 75, quindi $245 + 75 = 320$. 320 consiste nella parte finale del mio numero
- Concatenando i risultati ottengo il numero 32095.

Disegno

- Abbiamo una concatenazione di moltiplicatori con addizionatori per naturali $(n \times 1)$.
- Ogni volta abbiamo in ingresso un numero X ad n cifre e una cifra y_i.
- Ogni volta si ottiene come risultato un prodotto ad $n+1$ cifre.
- Di queste $n+1$ cifre porto verso l'uscita, ogni volta, la cifra meno significativa.
- Le n rimanenti cifre diventano C, e vanno in ingresso in altri moltiplicatori. Ovviamente C nel primo moltiplicatore è uguale a 0.



Capitolo 26

Martedì 27/10/2020

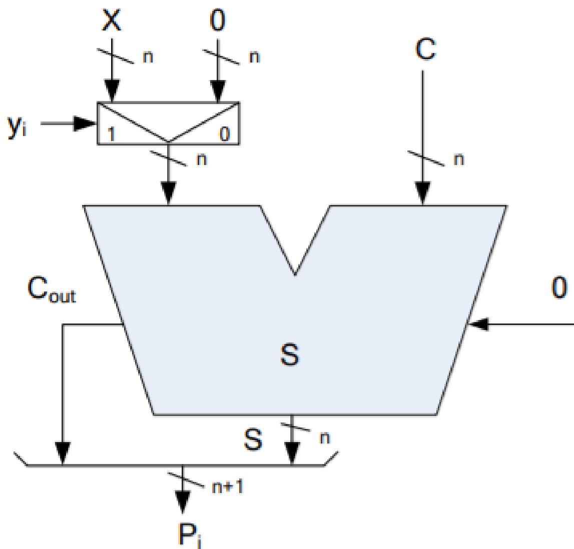
26.1 Moltiplicatore con addizionatore $n \times 1$ in base 2

Possiamo realizzare la moltiplicazione attraverso moltiplicatori ad $n \times 1$ cifra. In algebra abbiamo quanto segue...

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

dove P_i consiste nell'uscita del moltiplicatore. Attenzione al dimensionato del moltiplicatore: se abbiamo in ingresso robe di n ed m bit otterremo un risultato di $n + m$ bit. Nel caso di questo moltiplicatore abbiamo X, C ad n bit ed y_i di 1 bit: il risultato sarà di $n + 1$ bit!

Spogliamo il moltiplicatore



Osserviamo dall'algebra che abbiamo due comportamenti diversi in base al valore di y_i. Possiamo realizzare questo moltiplicatore attraverso un sommatore che ha in ingresso, oltre a C_{in},

- l'operando C, presente in ogni circostanza, e

- un secondo operando che può essere 0 o X.

L'idea iniziale è di usare un multiplexer dove y_i consiste nella variabile di comando. Tuttavia utilizzare un multiplexer è cosa poco intelligente in questo caso: l'alternativa al valore X è una costante. Metodo decisamente più conveniente è utilizzare n porte AND (tante porte quante le cifre) dove gli operandi sono una cifra di X e la variabili di comando y_i.

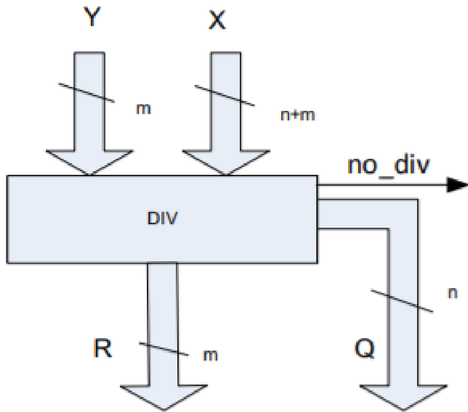
Ricordiamo Assemler L'istruzione MUL ha un solo operando: gli altri due sono impliciti e dipendono dalla dimensione dell'operando esplicito. Si osserva che il fattore esplicito è uguale al fattore implicito. Segue $n = m$: l'operazione calcola il risultato del prodotto su $2n$ bit partendo da operandi a n bit.

26.2 Divisione

Abbiamo:

- un numero naturale X in base β su $n + m$ cifre (dividendo) tale che $0 \leq X \leq \beta^{n+m} - 1$
- un numero naturale Y in base β su m cifre (divisore) tale che $0 \leq Y \leq \beta^m - 1$

Vogliamo ottenere due numeri Q, R (rispettivamente quoziente e resto). Sappiamo che la coppia (Q, R) è unica per il teorema della divisione (ricordiamoci che l'unicità è garantita dal fatto che il resto naturale è $0 \leq R \leq Y - 1$).



Uscita di non fattibilità In uscita avremo anche una no_div, cioè un'uscita di *non fattibilità*. Questa avrà valore uno se

- il divisore Y è zero
- il quoziente non sta sul numero di cifre prefissato

Dimensionamento

- Se il resto, per il teorema di unicità della divisione, è sicuramente minore del divisore allora mi basteranno m cifre.

- Relativamente al quoziente sappiamo che avremo il massimo con divisore $Y = 1$, cioè $X/1 = X$. Segue che il quoziente stia alla peggio in $n + m$ cifre.
- Vogliamo fare una cosa diversa: porre il quoziente su n cifre. Calcoliamo

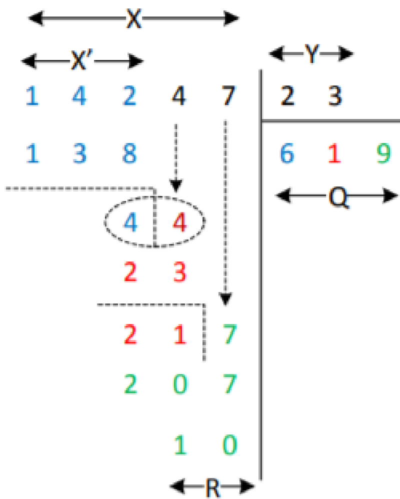
$$X = Q \cdot Y + R \leq \boxed{\boxed{(\beta^n - 1)} \cdot Y + \boxed{Y - 1} = \beta^n \cdot Y - Y + Y - 1 = \boxed{\beta^n \cdot Y - 1}}$$

  ponendo il massimo di Q $(\beta^n - 1)$ ed R $(Y - 1$, ricordiamo le proprietà del resto) troviamo che è possibile porre solo n cifre per il quoziente se $X < \beta^n \cdot Y$. Ovviamente non potremo fare tutte le divisioni.

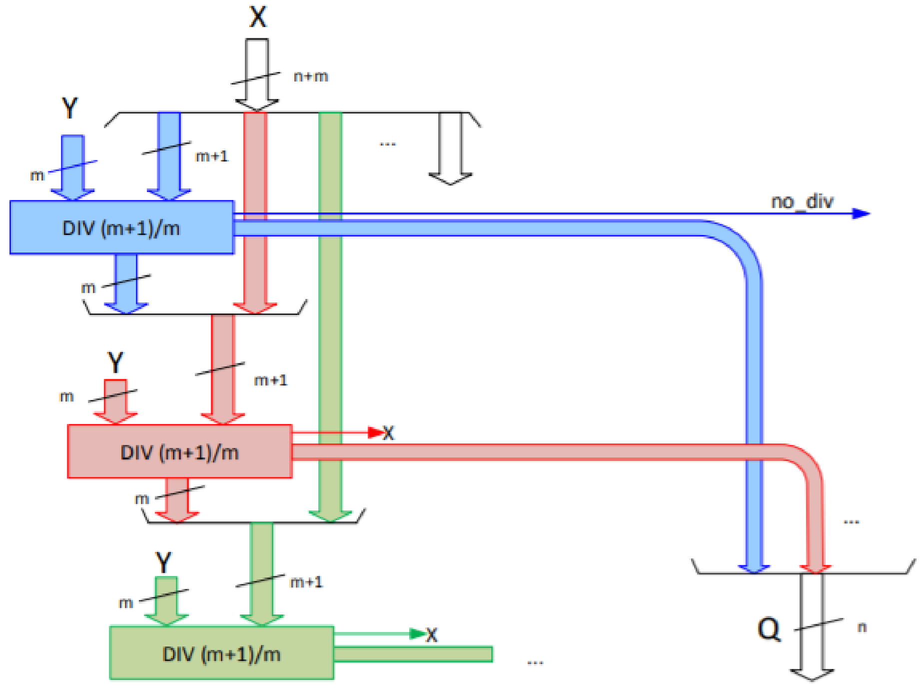
  - Se n, m non sono dati di un problema basterà porre un valore n tale che quella disuguaglianza sia vera per ciò che dobbiamo fare.
  - Se n, m sono dati del problema la cosa si fa più complicata. Purtroppo dovremo riflettere su questo caso, che è quello tipico nei calcolatori (si lavora sempre su campi finiti!)

Algoritmo L'algoritmo della divisione è il solito visto alle elementari. Supponiamo di svolgere la divisione X/Y:

- Prendo il minimo numero necessario delle cifre più significative di X. Questo numero sarà $X' \in [Y, \beta \cdot Y[$. Il numero di cifre necessarie è $m + 1$: m cifre possono essere insufficienti, $m + 1$ sicuramente bastano (ovviamente non con zeri in testa)
- Svolgo la divisione X'/Y ottenendo un quoziente q_i e un resto. Sappiamo dalle ipotesi che $X' < \beta \cdot Y$, quindi q_i sta su una sola cifra.
- Calcolo un nuovo dividendo X' concatenando il resto ottenuto con la prima cifra più significativa tra le cifre non ancora utilizzate del dividendo. L'ipotesi $X' < \beta \cdot Y$ rimane valida.
- Continuo così finchè non ho esaurito le cifre del dividendo



Il quoziente è ottenuto dal concatenamento dei quozienti parziali (tutti singole cifre), mentre il resto consiste nel resto dell'ultima divisione parziale. I due risultati sono stati ottenuti attraverso una serie di divisioni X/Y dove X ha $m+1$ cifre ed Y ha m cifre. Il risultato, tenendo conto delle proprietà dette prima, sarà di m cifre.



Condizioni di fattibilità Abbiamo già detto che

$$\boxed{X < \beta^n \cdot Y}$$

cioè X deve essere minore di Y traslato a sinistra n volte. Questa cosa può essere letta anche così: *le m cifre più significative del dividendo rappresentano un numero più piccolo del divisore.*

$$\begin{array}{lccccc|ccc} X: & x_{n+m-1} & x_{n+m-2} & \dots & x_{n+1} & x_n & x_{n-1} & \dots & x_0 \\ \beta^n \cdot Y: & y_{m-1} & y_{m-2} & \dots & y_1 & y_0 & 0 & \dots & 0 \end{array}$$

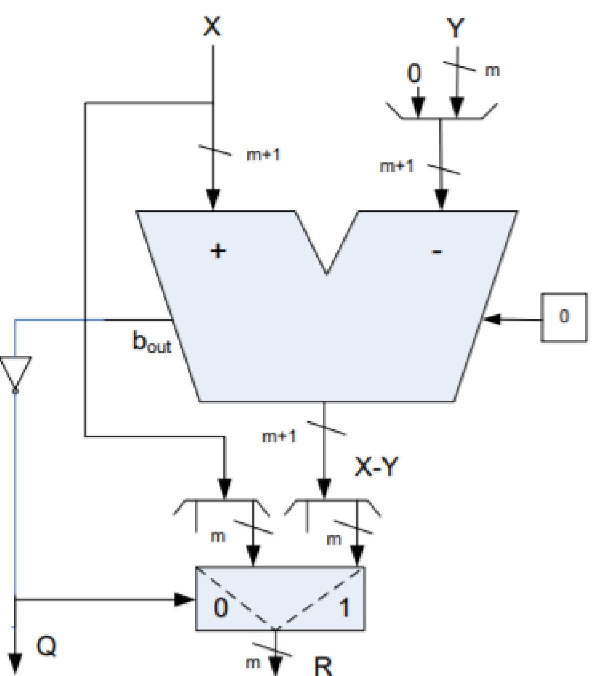
- Chiaramente il numero costituito dalle n cifre meno significative di X è maggiore rispetto a 0 (sequenza di n zeri in $\beta^n \cdot Y$)
- Segue che la validità o meno della condizione dipende esclusivamente dalle m cifre più significative di X e $\beta^n \cdot Y$.
- Se il numero costituito dalle m cifre più significative di X è minore rispetto al numero costitutito dalle m cifre più significative di $\beta^n \cdot Y$ avrò $X < \beta^n \cdot Y$.
- Si individua da questi ragionamenti che l'uscita no_div del circuito finale è data dall'uscita no_div dal primo modulo della scomposizione.

26.2.1 Divisore elementare in base 2

L'unità elementare esegue una divisione tra un dividendo a $m+1$ cifre e un divisore ad m cifre. Si ipotizza che $\boxed{X < 2 \cdot Y}$ ottenendo

- un quoziente su una cifra. Vale 0 se il divisore è maggiore del dividendo, 1 altrimenti.
- un resto su m cifre. Risulta uguale al dividendo se questo è minore del divisore, altrimenti è uguale alla differenza tra dividendo e divisore. Ripensare all'algoritmo delle elementari.

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



Abbiamo:

- un sottrattore, con ingresso due operandi ad $m+1$ cifre (quindi si espande Y di una cifra). Col sottrattore possiamo:
  1. stabilire se il dividendo X sia o meno minore del divisore Y;
  2. calcolare l'eventuale resto

  attenzione a dividendo e divisore: il primo è a $m+1$ bit, l'altro ad m bit. Segue la necessità di un sottrattore ad $m+1$ bit, quindi di estendere il divisore ad $m+1$ bit. Il risultato sarà una differenza ad $m+1$ bit, da cui togliamo la cifra più significativa (il resto sta su m bit). Otteniamo anche un prestito uscente, che ci indica se $X < Y$ o $X \geq Y$.
- Il quoziente, che ha valore 0 o 1 è determinato dal prestito uscente b_{out} del sottrattore (il quoziente si ottiene facendo passare il prestito uscente da una porta NOT).
- Q è variabile di comando del multiplexer: se $Q = 0$ passa X, altrimenti passa la differenza $X - Y$. Riduco X (stesso motivo di prima).

Capitolo 27

Mercoledì 28/10/2020

27.1 Rappresentazione dei numeri interi

Fino ad ora abbiamo parlato esclusivamente di numeri naturali: in modo molto semplice affermiamo che dati n bit possiamo rappresentare i numeri appartenenti all'intervallo $[0, \beta^n - 1]$ dove β consiste nella base adottata. Relativamente agli interi dobbiamo rappresentare il segno del numero: positivo o negativo (oltre al modulo).

- L'idea più intuitiva consiste nel rappresentare i numeri come modulo ($n-1$ bit) e segno (1 bit) all'interno di un calcolatore. La cosa può sembrare intuitiva ma in realtà complica i circuiti.
- Si è deciso di rappresentare i numeri interi in modo non intuitivo: abbiamo la cosiddetta **rappresentazione in complemento alla radice**. Questa rappresentazione viene ormai mantenuta per ragioni di compatibilità

Leggi biunivoche Parlando di naturali abbiamo individuato la seguente legge biunivoca

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

che mi permette di unire questi due insiemi

$$\boxed{\text{Numeri naturali da 0 a } \beta^n - 1} \iff \boxed{\text{Sequenze di } n \text{ cifre in base } \beta}$$

Con gli interi introdurremo un ulteriore legge biunivoca con cui uniamo i seguenti insiemi

$$\boxed{\text{Numeri naturali da 0 a } \beta^n - 1} \iff \boxed{\text{Insieme di } \beta^n \text{ numeri interi}}$$

Sappiamo di lavorare su n bit in base β: posso associare ad ogni elemento dell'insieme di β^n numeri interi un elemento dell'insieme dei numeri naturali $\in [0, \beta^n - 1]$, quindi posso associare l'insieme delle sequenze di n cifre in base β all'insieme dei β^n numeri interi.

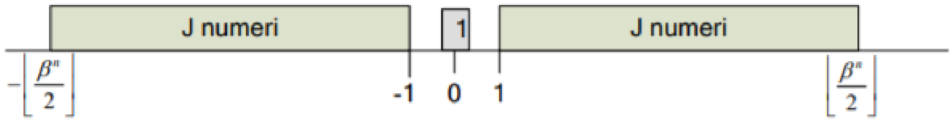
$$\boxed{\text{Insieme di } \beta^n \text{ numeri interi}} \iff \boxed{\text{Sequenze di } n \text{ cifre in base } \beta}$$

Questo significa che una sequenza binaria può rappresentarmi in modo corretto sia un numero naturale che un numero intero. Il numero intero che rappresenta dipenderà dalla **legge di rappresentazione** adottata.

27.1.1 Leggi di rappresentazione dei numeri interi

Immaginiamo queste legge come una funzione $L : \mathbb{Z} \to \mathbb{N}$. Di cosa abbiamo bisogno?

- **Dominio**. Il dominio consiste in un intervallo continuo (privo di buchi) e deve essere simmetrico il più possibile (abbiamo già visto che un intervallo non simmetrico può dare problemi se calcoliamo gli opposti in prossimità degli estremi dell'intervallo). L'idea di base per avere una corrispondenza biunivoca è immaginarci, escludendo lo 0, due parti identiche aventi J numeri

  Tuttavia si osserva che

  - con β dispari si avrà un numero β^n dispari, quindi togliendo lo 0 abbiamo un numero pari di elementi.
  - con β pari si avrà un numero β^n pari, quindi togliendo lo 0 abbiamo un numero dispari di elementi.
  - $n = 0$ non è un problema (motivazione ovvia)

  Segue che non è possibile avere una corrispondenza biunivoca con un insieme di β^n numeri se β è pari. **Relativamente ai numeri pari**: noi lavoreremo sempre con β pari, quindi dobbiamo accettare di avere un numero positivo o negativo in più. L'intervallo che ci interessa, il dominio, consiste nel seguente

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

  si ha un numero negativo in più![1] Con $\beta = 2$ ci riconduciamo all'intervallo di rappresentabilità visto a *FdP*.

- **Codominio**. Se la legge deve restituirci un naturale allora avremo il solito intervallo di rappresentabilità: $[0, \beta^n - 1]$

- **Legge di rappresentazione**. Leggi adeguate offrono vantaggi implementativi non banali e soprattutto permettono di utilizzare la stessa circuteria per maneggiare naturali e interi.

Nella discussione delle leggi utilizzeremo:

- La lettera maiuscola per indicare un numero naturale
- La lettera minuscola senza indici per indicare un numero intero
- La lettera minuscola con indici per indicare una cifra

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

[1] $\frac{\beta^n}{2}$ consiste nella metà dei numeri interi possibili. Ricordiamo che con β pari il risultato di β^n è pari! Prendo un numero pari di elementi (incluso lo zero), divido per due e sottraggo 1 da uno dei due estremi.

27.1.1.1 Modulo e segno

Un numero intero coincide con una coppia costituita da un numero naturale (modulo) e una variabile logica (segno).

$$(s, M) \longleftrightarrow a \qquad s = \begin{cases} 0 & a \geq 0 \\ 1 & a < 0 \end{cases}$$

Poniamo questa possibilità di legge a solo scopo informativo.

27.1.1.2 Traslazione

Dato $\frac{\beta^n}{2}$, detto **fattore di polarizzazione**, otteniamo

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

Mi immagino la cosa in modo molto simpatico: la conversione della temperatura da gradi Kelvin a gradi Celsius, e viceversa. La cosa è lampante se leggiamo la tabella a pagina 39 della dispensa di aritmetica. Supponiamo di avere $\beta = 2, n = 8$, quindi $\frac{\beta^n}{2} = 128$

Es: $\beta = 2,\ n = 8$

a	−128	−127	−1	0	+1	+126	+127
A	0	1	127	128	129	254	255
rapp.	00000000	00000001	01111111	10000000	10000001	11111110	11111111

- Il naturale 0 rappresenta l'intero -128

$$A = \boxed{-128} + 128 = 0$$

- Il naturale 128 rappresenta l'intero 0.

$$A = \boxed{0} + 128 = 128$$

- Il naturale 255 rappresenta l'intero +127.

$$A = \boxed{+127} + 128 = 255$$

Monotonia La legge è monotona, quindi possiamo dire

$$a < b \longleftrightarrow A < B$$

abbastanza scontato che se alteriamo allo stesso modo i due membri di una disuguaglianza, sommando o sottraendo (in questo caso il fattore di polarizzazione), l'esito della disuguaglianza risulti invariato.

Segno La cifra più significativa permette di ottenere il segno: la differenza rispetto alla rappresentazione in CR è il significato invertito delle cifre.

- Se MSD = 0 ho un numero negativo;
- se MSD = 1 ho un numero positivo.

Conversione da rappresentazione in traslazione a C2 Dato un numero rappresentato in traslazione si ottiene il suo equivalente in complemento a due complementando la cifra più significativa. Tenerne conto in esercizi Verilog in presenza di convertitori A/D bipolari.

Utilizzi

- Convertitori A/D e D/A
- Rappresentazione dell'esponente nei numeri reali.

27.1.1.3 Complemento alla radice

Questa legge (che abbiamo chiamato in passato anche *complemento a due*) è quella usata all'interno dei calcolatori:

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

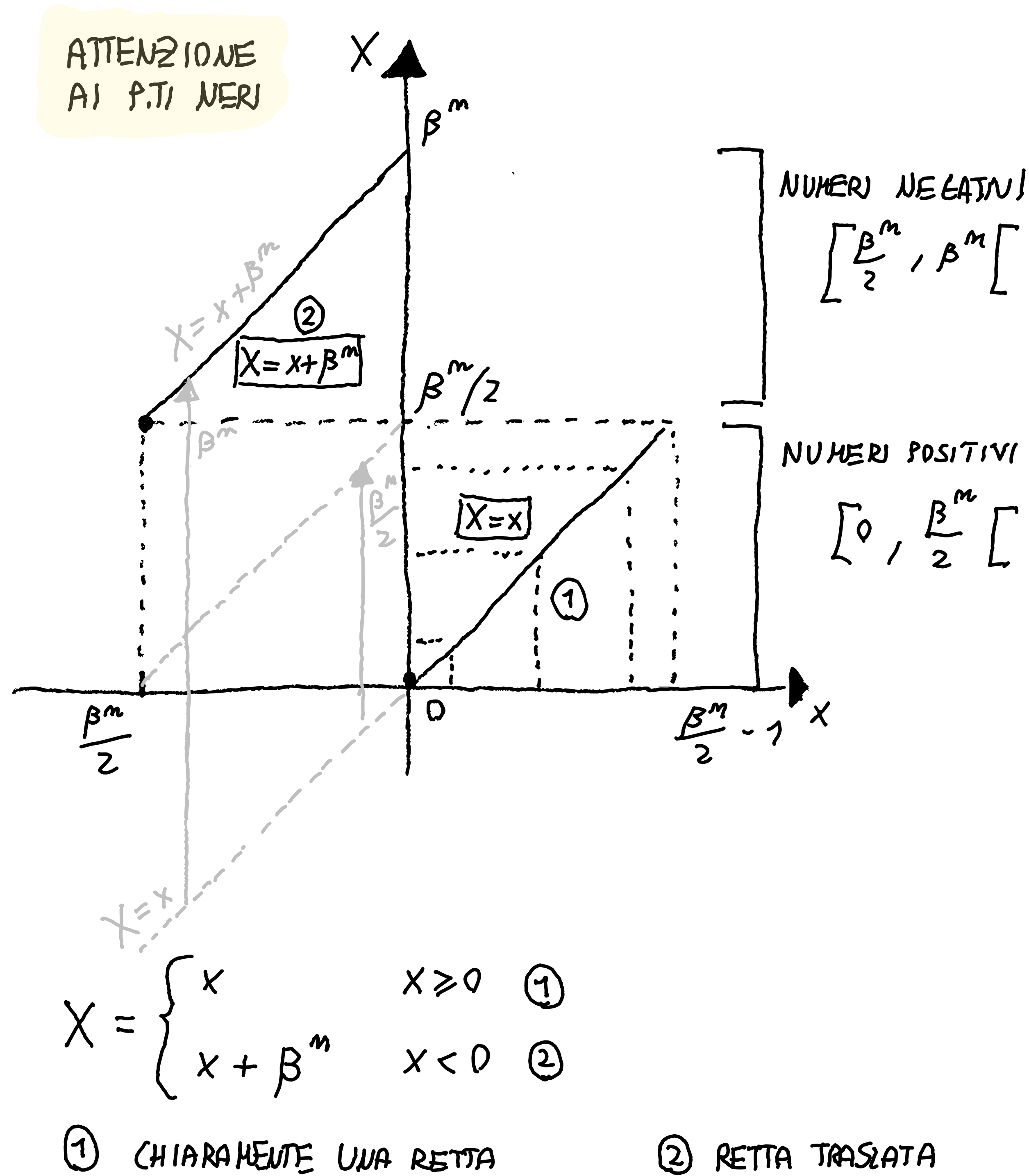
ricordarsi del disegno a farfalla visto all'inizio con il ripasso sulle Rappresentazione dell'informazione.

Es: $\beta = 2,\ n = 8$

a	-128	-127	-1	0	+1	+126	+127
A	128	129	255	0	1	126	127
rapp.	10000000	10000001	11111111	00000000	00000001	01111110	01111111

Monotonia La monotonia è valida SOLO tra numeri aventi stesso segno

DISEGNO DELLA FARFALLA (IN CR)



$$X = \begin{cases} x & x \geq 0 \quad ① \\ x + \beta^m & x < 0 \quad ② \end{cases}$$

① CHIARAMENTE UNA RETTA CON $m = 1$

② RETTA TRASLATA

$\beta = 2 \qquad n = 8 \qquad \left[-\frac{\beta^n}{2}, \frac{\beta^n}{2} - 1\right] \longrightarrow [-128, +127]$

$a = -128$

$A = \beta^n + a = 2^8 - 10000000 =$

$$
\begin{array}{r}
100000000 - \\
10000000 = \\
\hline
\not{0}10000000
\end{array}
$$

$a = -64$

$A = \beta^n + a = 2^8 - 1000000$

$$
\begin{array}{r}
100000000 - \\
1000000 = \\
\hline
\not{0}11000000
\end{array}
$$

RICORDARE L'ESTENSIONE DI CAMPO
-64 PUÒ ESSERE RAPPRESENTATO CON 7 CIFRE. $[-64, +63]$

$a = -1$

$A = \beta^n + a = 100000000 - 1 = \not{0}11111111$

SEMPRE VALIDO IN BASE 2, CAMBIA SOLO IL NUMERO DI CIFRE.

$a = 127$

$A = a = 01111111$

27.1.2 Proprietà del complemento alla radice

27.1.2.1 Determinazione del segno

Premessa Generalizzeremo quanto già sappiamo per la base $\beta = 2$

$$s = \begin{cases} a_{n-1} = 0 & a \geq 0 \\ a_{n-1} = 1 & a < 0 \end{cases}$$

Generalizziamo Dal disegno della farfalla individuiamo che

- $a \geq 0 \iff 0 \leq A < \frac{\beta^n}{2}$
- $a < 0 \iff \frac{\beta^n}{2} \leq A < \beta^n$

L'intero a più grande rappresentabile è, appunto, $\frac{\beta^n}{2} - 1$. Sapendo che il numero immediatamente successivo è $\frac{\beta^n}{2}$ poniamo

$$\frac{\beta^n}{2} = \beta^{n-1} \cdot \frac{\beta}{2} = (\beta/2 \cdot 00 \ldots 00)_\beta$$

cioè $\beta^n/2$ consiste in $\beta/2$ shiftato $n-1$ volte a sinistra. Segue il valore di $\beta^n/2 - 1$

$$\beta^n/2 - 1 \equiv \Big(\boxed{(\beta/2 - 1)} (\beta - 1)(\beta - 1) \ldots (\beta - 1) \Big)_\beta$$

si deduce che è possibile capire il segno di un numero controllando la sua cifra più significativa.

$$s = \begin{cases} a_{n-1} < \frac{\beta}{2} & a \geq 0 \\ a_{n-1} \geq \frac{\beta}{2} & a < 0 \end{cases}$$

Esempi

- $\beta = 10, n = 4$: $\beta^n/2 - 1 \equiv (4999)_{10}$
- $\beta = 2, n = 8$: $\beta^n/2 - 1 \equiv (01111111)_2$

27.1.2.2 Legge inversa

Data la legge già vista

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

troviamo

$$L^{-1} : a = \begin{cases} A & 0 \leq A < \frac{\beta^n}{2} \\ A - \beta^n & \frac{\beta^n}{2} \leq A < \beta^n \end{cases}$$

Semplificazione della legge inversa Possiamo scrivere la legge inversa in forma semplificata considerando la definizione di complemento e quanto detto sulla cifra più significativa

$$\overline{A} = \beta^n - 1 - A \longleftrightarrow A = \beta^n - 1 - \overline{A}$$

otteniamo

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

Esempi

- $\beta = 10, n = 3$: $A \equiv (852)_{10}$. Osserviamo che la cifra più significativa è maggiore di $\beta/2 - 1 = 4$, quindi il numero rappresentato è negativo. Calcolo $\overline{A} \equiv (147)_{10}$, sommo uno e cambio il segno. Risultato: $a = -148$
- $\beta = 10, n = 3$: $A \equiv (500)_{10}$. Il numero è negativo, quindi $a = -(499 + 1) = -500$
- $\beta = 2, n = 4$: $A \equiv (1011)_2$. La cifra più significativa è 1, il numero è negativo.

$$a = -(0100 + 1)_2 = -(101)_2 = -5$$

- $\beta = 2, n = 4$: $A \equiv (1111)_2$. La cifra più significativa è 1, il numero è negativo.

$$a = -(0000 + 1)_2 = -(1)_2 = -1$$

27.1.2.3 Forma alternativa della legge

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

La legge L può essere scritta in altro modo

$$\boxed{A = |a|_{\beta^n}} \quad \text{se} \quad -\frac{\beta^n}{2} \leq a < \frac{\beta^n}{2}$$

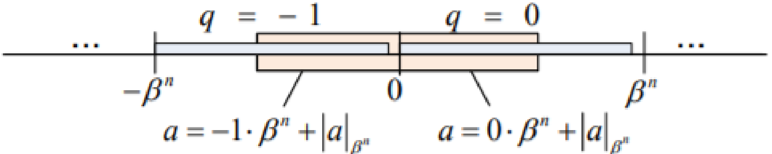
Perchè la cosa è valida?

- Con a positivo sappiamo che $a < \beta^n$, conseguentemente la divisione avrà quoziente $q = 0$.

$$A = 0 \cdot \beta^n + |a|_{\beta^n} = |a|_{\beta^n}$$

- Con a negativo sappiamo che $-\beta^n/2 \leq a < 0$. La divisione restituisce quoziente $q = -1$, quindi $a = -\beta^n + |a|_{\beta^n}$. Se sostituiamo si ottiene

$$A = \beta^n + (-\beta^n + |a|_{\beta^n}) = |a|_{\beta^n}$$



Attenzione alla rappresentabilità Prima di fare questi calcoli dobbiamo verificare che l'intero a sia effettivamente rappresentabile su n bit. Se il numero non è rappresentabile l'operatore modulo ci restituirà un numero che non ha nulla a che vedere con la rappresentazione di a. Vediamo un esempio

$$\beta = 10, n = 3, a = -953 \quad |a|_{\beta^n} = |-953|_{1000} = 47$$

abbiamo ottenuto un risultato ma sappiamo che su tre cifre possiamo rappresentare solo i numeri appartenenti all'intervallo $[-500, +499]$. Segue che il risultato non ha senso.

27.2 Operazioni su interi in complemento alla radice

I circuiti che progetteremo lavorano sulle rappresentazioni: ho in ingresso cifre in base β, otterrò in uscita cifre in base β. Attraverso la legge di rappresentazione che abbiamo in mente (precisamente quella in complemento alla radice) i circuiti eseguirano operazioni sui numeri interi. Ricordiamoci che i circuiti vedono esclusivamente cifre!

Conseguenza del discorso sulle leggi biunivoche (cit. Corsini) *Rappresentare le informazioni tramite numeri naturali e non direttamente tramite stringhe di bit, presenta un duplice vantaggio: si acquista in compattezza e leggibilità e si riduce l'elaborazione delle informazioni a familiari operazioni aritmetiche su numeri naturali.*

27.2.1 Valore assoluto

Vogliamo trovare il numero naturale $B = \text{ABS}(a)$. Sappiamo che $a \in [-\beta^n/2, \beta^n/2 - 1]$, segue che $B \in [0, \beta^n/2]$. B è un numero naturale rappresentabile su n cifre! Il circuito che vogliamo realizzare prende in ingresso

$$A \equiv (a_{n-1} \dots a_0)_\beta$$

e produce

$$B \equiv (b_{n-1} \dots b_0)_\beta$$

con $a \longleftrightarrow b$.

Definizione matematica Sappiamo che in matematica il valore assoluto consiste in un sistema a tratti

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

sfruttando la legge inversa possiamo scrivere quanto segue

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

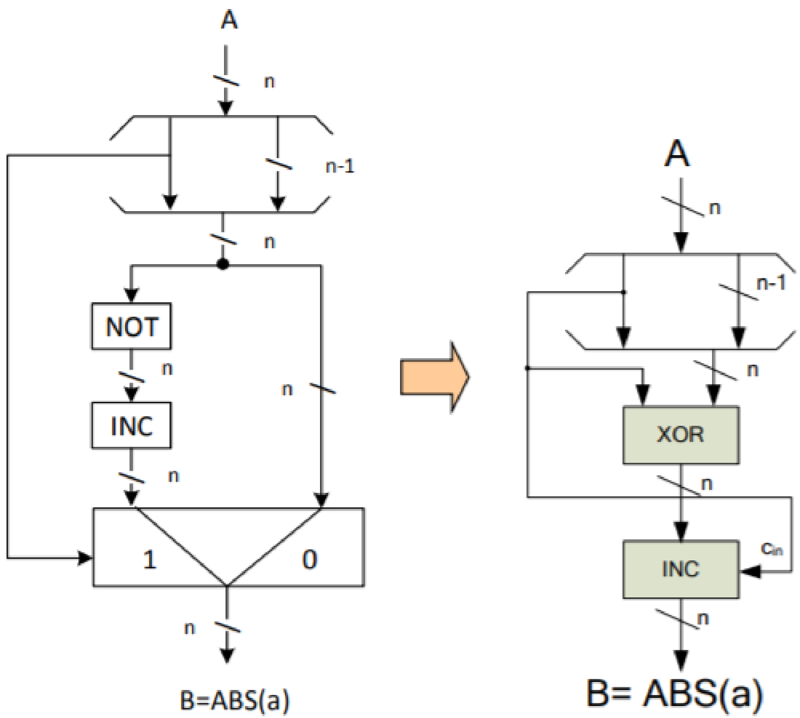
Osservazione Si ha il passaggio seguente (si capisce perchè l'operazione è sempre possibile)

$$[-\beta^n/2, \beta^n/2 - 1] \Longrightarrow [0, \beta^n - 1]$$

Sintetizzazione

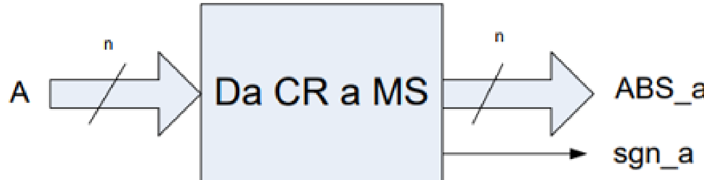
- Si può pensare a un multiplexer: dobbiamo scegliere se porre una cosa o un'altra.
- La variabile di comando del multiplexer sarà il risultato del confronto tra a_{n-1} e $\beta/2$. Poichè lavoriamo in base due in questo caso ci basta prendere la cifra più significativa e porla come variabile di comando.
- Un circuito che calcola complemento ed incremento, per il ramo inferiore.

Base 2 Il circuito può essere semplificato usando le porte XOR: la porta XOR $j-$esima calcolerà $a_{n-1} \oplus a_j$:



- Utilizzare le porte XOR permette di gestire entrambe le situazioni.
  - Se $a_{n-1} = 0$ non è nostro interesse fare il complemento: fare XOR tra 0 e un numero significa ottenere in uscita quel numero.
  - Se $a_{n-1} = 1$ dobbiamo fare il complemento: fare XOR tra 1 e un numero significa ottenere in uscita il complemento del numero.
- Ovviamente $a_{n-1} \oplus a_{n-1} = 0$: potrei omettere la porta XOR $n-1$ semplificando il circuito
- L'incrementatore presenta un riporto in ingresso: il riporto in ingresso consiste nella cifra più significativa. Il segno determina se ci sarà l'incremento o meno.

27.2.2 Circuito di conversione da CR a MS



Dato A in ingresso su n cifre, un circuito di conversione da CR a MS restituisce

- valore assoluto su n cifre;

- una variabile logica che indica il segno.

si osservi che abbiamo in uscita lo stesso numero di bit per il valore assoluto, non un bit in meno.

Rappresentabilità L'intervallo di rappresentabilità per MS è $[-(\beta^n - 1); +(\beta^n - 1)]$: in esso è contenuto l'intervallo di rappresentabilità in CR

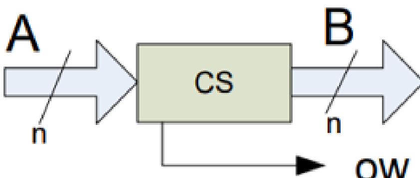
$$\left[-\frac{\beta^n}{2}; +\frac{\beta^n}{2} - 1\right] \subset [-(\beta^n - 1); +(\beta^n - 1)]$$

segue che la conversione è sempre possibile.

Sintesi Per il valore assoluto utilizziamo quanto visto prima. Per il flag che indica il segno abbiamo due strade:

- utilizzare il flag_min di un comparatore che confronta a_{n-1} e $\beta/2$;
- controllare direttamente a_{n-1} se siamo in base 2.

27.2.3 Calcolo dell'opposto



Il circuito pone in ingresso la rappresentazione A di un numero a e restituisce la rappresentazione B di un numero b. I numeri sono legati dalla seguente relazione

$$b = -a$$

Non bisogna dire che $B = -A$! Chi scrive questa cosa, dice Stea, non ha capito niente. La relazione è valida per i numeri interi a e b, non per le rappresentazioni A e B.

Non si può fare sempre Osserviamo l'intervallo di rappresentazione: sappiamo che

$$a \in [-\beta^n/2, \beta^n/2 - 1]$$

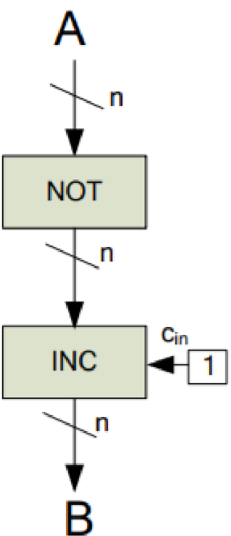
Esiste un intero negativo (l'estremo sinistro) che non ha opposti nell'intervallo. Segue che il circuito prevederà, oltre all'uscita B, anche una variabile logica OW.

Osservazione Perchè l'operazione del valore assoluto è sempre possibile e quella dell'opposto no? Nella prima si passa dall'insieme dei numeri interi all'insieme dei naturali, mentre in questa si rimane nell'insieme degli interi.

Sintetizziamo Per la sintetizzazione della rete, in particolare per la parte relativa all'uscita B, supponiamo inizialmente che $a \neq \beta^n/2$. Svolgiamo i calcoli (il simbolino sopra l'uguale - non presente qua - sottolinea che l'uguaglianza è valida con riserva, appunto quando non trattiamo l'estremo sinistro)

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

Utilizzando una delle proprietà introdotte da Corsini sul suo libro e la definizione di resto (vedere definizione di Corsini per avere le idee chiare) otteniamo il resto come A negato e incrementato. La cosa è facilmente sintetizzabile con la seguente rete:



Abbiamo

- n variabili in ingresso (rappresentazione di A)
- un operatore NOT (per la complementazione)
- un operatore INC con $C_{in} = 1$ (generatore di costante, l'incremento avviene sempre)
- n variabili in uscita (rappresentazione di B)

Come si rappresenta il modulo? Osserviamo che in uscita abbiamo solo n variabili: il riporto in uscita C_{out} non viene minimamente considerato.

Esempi

- Attenzione ad $A = 10000000$: l'uscita non è corretta

  $$B = 10000000$$

  guarda caso $A \longleftrightarrow -\beta^n/2$, l'opposto non è rappresentabile (dato un numero intero si ottiene un altro numero negativo).

- Attenzione ad $A = 00000000$. In questo caso gli affari tornano: se applico la formula trovo

$$B = 00000000$$

e il riporto uguale ad 1 (che non consideriamo). Risulta ovvio che con 0 in ingresso otterrò nuovamente zero (non abbiamo zero positivo e zero negativo come nella rappresentazione in modulo e segno).

Sintetizzazione della parte relativa a OW A questo punto possiamo sintetizzare la parte rimanente della rete. Abbiamo capito che ci sono problemi solo con un numero in particolare

$$A \equiv (10\dots00)$$

l'idea iniziale potrebbe essere verificare che il numero in ingresso sia uguale a quello, ma ciò richiederebbe una AND ad n ingressi. La cosa può essere semplificata: ciò che dobbiamo verificare sono i segni dell'input e dell'output

$$\text{sgn } a = \text{sgn } b = 1$$

cioè devo verificare se dato un numero negativo in ingresso ottengo un altro numero negativo in uscita (si capisce facilmente che qualcosa non va). Quindi:

- in base $\beta = 2$ prendo i bit più significativi e li pongo in una porta AND;
- in base $\beta \neq 2$ utilizzo due comparatori (uno per a e uno per b) e passo i loro risultati in una porta AND.

Osservazione Nel confronto dobbiamo vedere se i due bit sono uguali ad 1, non se sono uguali. Se ho i due bit uguali a zero significa che abbiamo cercato l'opposto di zero, e questo non è male.

Assembler Abbiamo visto l'istruzione NEG che interpreta una sequenza di bit come la rappresentazione di un numero intero, e produce la rappresentazione dell'opposto se questo esiste. Altrimenti setta il flag di overflow OF.

27.2.4 Estensione di campo

Abbiamo già visto l'estensione di campo per numeri naturali, dove ci basta aggiungere k zeri in testa. Negli interi, come già possiamo intuire da quanto visto con Assembler, non sarà così. Abbiamo il naturale $A \equiv (a_{n-1} \dots a_0)_\beta$ tale che $A \longleftrightarrow a$ in CR su n cifre, vogliamo ottenere

$$A^{EST} \equiv (a'_n a'_{n-1} \dots a'_0)_\beta$$

tale che $A^{EST} \longleftrightarrow a$ in CR su $n+1$ cifre.

Sempre possibile? Contrariamente all'opposto l'estensione di campo è sempre possibile: l'intervallo di rappresentabilità su $n+1$ cifre contiene quello su n cifre. Avremo n variabili di ingresso ed $n+1$ variabili di uscita.

Comprendiamo La formula per l'estensione di campo si ottiene a partire dalla formula per la legge inversa che abbiamo visto introducendo le rappresentazioni dei numeri interi.

$$L : A = \begin{cases} a & 0 \le a < \frac{\beta^n}{2} \\ \beta^n + a & -\frac{\beta^n}{2} \le a < 0 \end{cases} \Longleftrightarrow L^{-1} : a = \begin{cases} A & a_{n-1} < \frac{\beta}{2} \\ -(\overline{A}+1) & a_{n-1} \ge \frac{\beta}{2} \end{cases}$$

aumentiamo il numero di bit

$$L'_{n+1} : A^{\text{EST}} = \begin{cases} a & 0 \le a < \frac{\beta^n}{2} \\ \beta^{n+1} + a & -\frac{\beta^n}{2} \le a < 0 \end{cases} \Longleftrightarrow L_n^{-1} : a = \begin{cases} A & a_{n-1} < \frac{\beta}{2} \\ \boxed{-(\overline{A}+1)} & a_{n-1} \ge \frac{\beta}{2} \end{cases}$$

ritorniamo alla formula inversa introdotta inizialmente:

$$a = -(\overline{A}+1) = -(\beta^n - 1 - A + 1) = \boxed{-\beta^n + A}$$

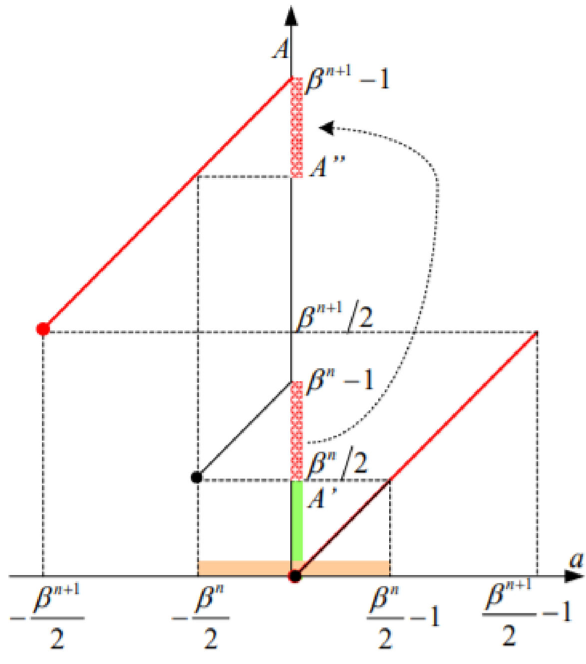
sostituiamo ottenendo

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

A^{EST} si ottiene attraverso concatenamenti di cifre. La relazione finale rende tutto più chiaro: la cifra aggiunta può avere come valore 0 o $\beta - 1$ in base al valore di a_{n-1}.

$$a'_i = a_i \quad 0 \le i \le n-1, a'_n = \begin{cases} 0 & a_{n-1} < \beta/2 \\ \beta - 1 & a_{n-1} \ge \beta/2 \end{cases}$$

Osservazioni grafiche



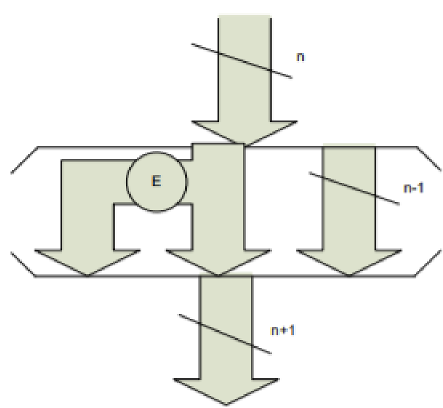
Abbiamo un disegno contenente sia la rappresentazione in CR su n bit che quella su $n+1$ bit. Voglio vedere come variano gli intervalli di ordinate.

- L'intervallo più grande contiene perfettamente quello più piccolo (come già detto)
- L'intervallo di ordinate che rappresenta gli interi positivi rimane invariato, mi limito ad aggiungere in testa una cifra più significativa di peso nullo.
- L'intervallo di ordinare che rappresenta gli intervalli negativi viene traslato verso l'alto di una quantità
$$(\beta^{n+1} - 1) - (\beta^n - 1) = \beta^n \cdot (\beta - 1)$$
aggiungo una cifra più significativa di peso $\beta - 1$ (ho fatto la differenza tra i valori di A più grandi possibili su $n+1$ bit ed n bit).

Concludiamo osservando che:

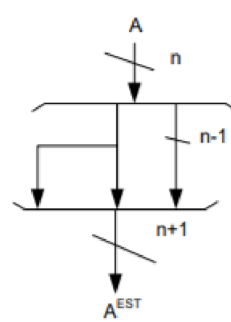
- rappresentazioni di numeri positivi estesi avranno MSD $= 0$ e la cifra successiva $< \beta/2$;
- rappresentazioni di numeri negativi estesi avranno MSD $= \beta - 1$ e la cifra successiva $\geq \beta/2$.

Sintetizzazione Si capisce che contrariamente ai numeri naturali dobbiamo utilizzare un minimo di logica. Il circuito estensore utilizzerà un sottrattore per capire se il numero da estendere è positivo o negativo.



Base 2 In base $\beta = 2$ la cosa si semplifica pesantemente
$$a'_n = a_{n-1}$$

L'unica cosa che dobbiamo fare è ripetere la cifra più significativa della rappresentazione!



Assembler Esistono delle istruzioni che permettono di interpretare una sequenza di bit come rappresentazione di un intero (su 8, 16 o 32 bit) e produrre la rappresentazione dello stesso numero estesa a 16, 32 o 64 bit. Non esistono operazioni simili per i naturali: in quel caso si aggiungono solo zeri in testa.

Capitolo 28

Giovedì 29/10/2020

28.1 Riduzione di campo

Supponiamo di avere un numero $A \equiv (a_n a_{n-1} \dots a_0)_\beta$ su $n+1$ cifre, tale che $A \longleftrightarrow a$ (cioè A è rappresentazione di un intero a). Noi vogliamo trovare

$$A^{RID} \equiv (a'_{n-1} \dots a'_0)_\beta$$

tale che $a \longleftrightarrow A^{RID}$. Contrariamente all'estensione non è un'operazione sempre fattibile, precisamente si può fare solo se

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

Recuperiamo quanto visto nel disegno delle farfalle:

- L'intervallo di ordinate $[0, A']$ che contiene rappresentazioni di numeri positivi avrà MSD = 0 e la cifra successiva minore di $\beta/2$
- L'intervallo di ordinate $[A'', \beta^{n+1} - 1]$ che contiene rappresentazioni di numeri negativi avrà MSD = 1 e la cifra successiva maggiore o uguale a $\beta/2$.

Segue la condizione di non overflow Avrò OW = 0 se

- $a_n = 0$ e $a_{n-1} < \beta/2$, oppure
- $a_n = \beta - 1$ e $a_{n-1} \geq \beta/2$

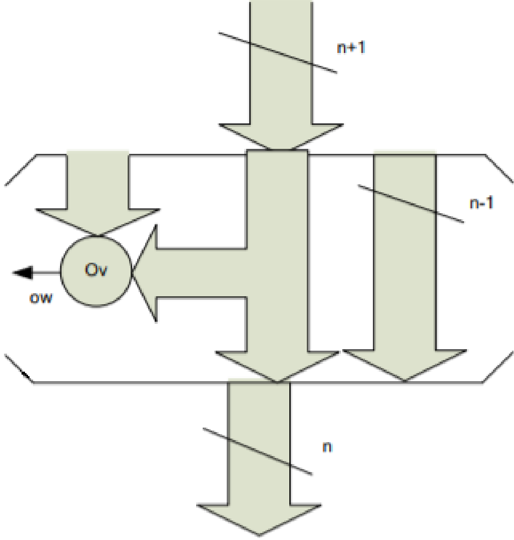
Da cosa si vede che stiamo sbagliando? Qualcosa non va se il risultato della riduzione di campo consiste in un qualcosa avente segno differente.

Come si ottiene la riduzione algebricamente? Osserviamo che $A = (A^{EST})^{RID}$, quindi possiamo dire

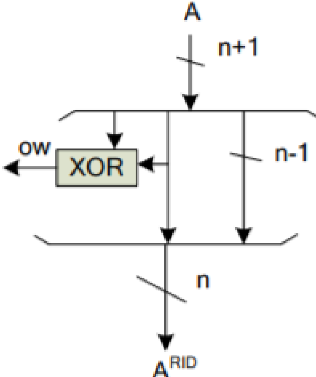
$$A^{RID} = |A|_{\beta^n}$$

dire questo significa sbarazzarsi della cifra più significativa (ricordiamo che partiamo da una rappresentazione su $n+1$ cifre).

Sintetizzazione Quello che ci rimane da sintetizzare è la parte relativa alla variabile logica OW: la non riducibilità sarà riconosciuta da un circuito detto **circuito di overflow**.



Ricordiamoci che in base 2 la condizione di non-overflow risulta semplificata notevolmente: le disuguaglianze diventano uguaglianze, ci basta controllare che le cifre più significative siano uguali.



Se in caso di uguaglianza abbiamo un non-overflow e quindi vogliamo restituire zero la soluzione migliore è uno XOR delle due cifre più significative.

28.2 Moltiplicazione per potenza della base

Supponiamo di avere un numero $A \equiv (a_n a_{n-1} \dots a_0)_\beta$ su n cifre, tale che $A \longleftrightarrow a$ (cioè A è rappresentazione di un intero a). Noi vogliamo trovare un numero B su $n+1$ cifre tale che $b \longleftrightarrow B$ e

$$b = \beta \cdot a$$

Posso rappresentare b su $n+1$ cifre Prendiamo la legge inversa che abbiamo visto introducendo la rappresentazione degli interi e sostituiamo

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

Sapendo che $A = \beta^n + a$ quando $a < 0$ si intuisce facilmente che la soluzione sarà

$$\boxed{B = \beta \cdot A}$$

quindi possiamo risolvere questa moltiplicazione nello stesso modo visto per i naturali!

28.3 Divisione per potenza della base

Supponiamo di avere un numero $A \equiv (a_n a_{n-1} \dots a_0)_\beta$ su $n+1$ cifre, tale che $A \longleftrightarrow a$ (cioè A è rappresentazione di un intero a). Noi vogliamo trovare un numero B su n cifre tale che $b \longleftrightarrow B$

$$b = \left\lfloor \frac{a}{\beta} \right\rfloor$$

Risoluzione algebrica Ricordiamo la legge semplificata vista non molto tempo fa

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

facciamo la stessa cosa

$$B = \left| \left\lfloor \frac{a}{\beta} \right\rfloor \right|_{\beta^n}$$

pongo $a = \left\lfloor \frac{a}{\beta^{n+1}} \right\rfloor \cdot \beta^{n+1} + |a|_{\beta^{n+1}}$, cioè mi immagino una divisione tra il dividendo a e β^{n+1} (il dividendo sta su $n+1$ cifre, ricordiamo)

$$B = \left| \left\lfloor \frac{\left\lfloor \frac{a}{\beta^{n+1}} \right\rfloor \cdot \beta^{n+1} + |a|_{\beta^{n+1}}}{\beta} \right\rfloor \right|_{\beta^n} = \left| \left\lfloor \left\lfloor \frac{a}{\beta^{n+1}} \right\rfloor \cdot \beta^n + \frac{|a|_{\beta^{n+1}}}{\beta} \right\rfloor \right|_{\beta^n}$$

per le proprietà del modulo posso togliere la prima parte e ottenere

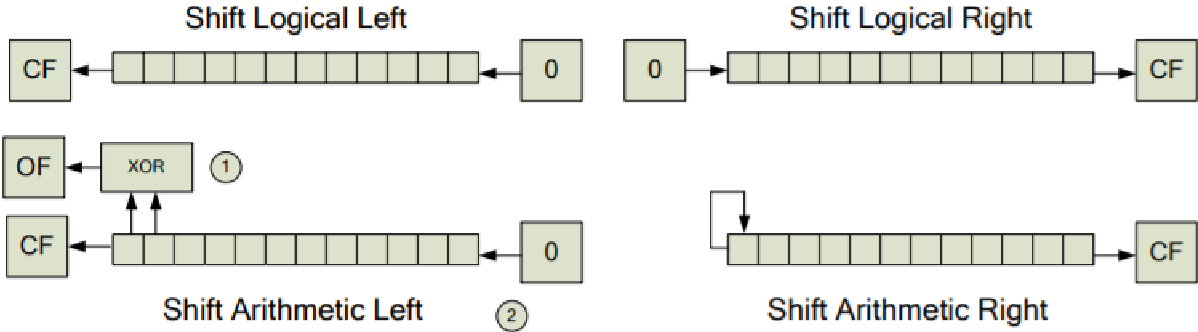
$$B = \left| \left\lfloor \frac{|a|_{\beta^{n+1}}}{\beta} \right\rfloor \right|_{\beta^n} =$$

sapendo che $A = |a|_{\beta^{n+1}}$ ottengo

$$= \left| \left\lfloor \frac{A}{\beta} \right\rfloor \right|_{\beta^n} = \left\lfloor \frac{A}{\beta} \right\rfloor$$

rimuovo il modulo poichè il risultato è sicuramente inferiore a β^n. Segue che la divisione può essere ottenuta con il metodo già visto per i numeri naturali.

28.4 Assembler: Shift logico e aritmetico



Fermi tutti! Abbiamo appena detto che il metodo di risoluzione è lo stesso: allora perchè abbiamo bisogno di due istruzioni distinte per ogni shift (cioè una per i naturali e una per gli interi)?

Shift sinistro

- Le due operazioni si svolgono nello stesso modo.
- Per la versioni con numeri interi è necessario considerare che lavoriamo su un numero fisso di bit. Segue che oltre al solito calcolo utilizzeremo anche un circuito di overflow (una XOR con ingresso le due cifre più significative).
- Non si hanno differenze procedurali. La distinzione tra i due shift è mantenuta soprattutto per ragioni di simmetria rispetto all'altro shift. Sta al programmatore discriminare quale flag guardare in base al contenuto del registro.

Shift destro

- Nello shift per naturali ci si sbarazza della cifra meno significativa e si aggiunge uno zero in testa
- Nello shift per interi ci si sbarazza della cifra meno significativa e si pone in testa una nuova cifra uguale a quella più significativa.
- Le differenze procedurali sono evidenti.

28.5 Somma

Supponiamo di avere due numeri naturali A, B tali che $a \longleftrightarrow A$ e $b \longleftrightarrow B$ (sono rappresentazioni di numeri interi). Vogliamo trovare il numero naturale S, rappresentazione dell'intero s, tale che

$$s = a + b$$

Rappresentazione Come nella somma coi naturali non è detto che il risultato sia rappresentabile su n cifre

$$-\frac{\beta^n}{2} - \frac{\beta^n}{2} \leq s \leq \frac{\beta^n}{2} - 1 + \frac{\beta^n}{2} - 1 \Longrightarrow \boxed{-\beta^n \leq s \leq \beta^n - 2}$$

per ottenere l'intervallo abbiamo considerato i casi limite ($a = b = \beta^n/2 - 1$ o $a = b = -\beta^n/2$). Sicuramente potremo rappresentare il risultato su $n + 1$ cifre poichè

$$\frac{\beta^{n+1}}{2} \geq \beta^n$$

segue che anche per questa operazione servirà un'uscita di overflow (noi lavoriamo su n bit).

Algebra Tenendo conto della condizionale posso dire che

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

si individua una cosa importantissima: la rappresentazione della somma consiste nella somma delle rappresentazioni modulo β^n. Quanto trovato giustifica l'uso della rappresentazione in complemento alla radice dei numeri interi (uso della stessa circuiteria)

Sintetizzazione Possiamo facilmente intuire che anche in questo caso utilizzeremo il sommatore visto per i naturali. Tuttavia:

- Possiamo utilizzare il C_{out} per controllare se siamo andati in overflow? **NO**! Il riporto uscente non è di nessun aiuto. Esistono situazioni in cui un eventuale variabile di overflow e il riporto uscente risultano essere diversi:
  - Prendiamo $A = \beta^n - 1, B = 1 \Longrightarrow S = 0, C_{\text{out}} = 1$. Andiamo a fare

    $$(111\ldots11)_2 + (00\ldots1)_2 = (\boxed{1}000\ldots00)_2$$

    evidenziato il riporto uscente. Il numero finale è 0. Il risultato è giusto, perchè abbiamo sommato -1 e 1. Tuttavia il riporto uscente è uguale ad 1!
  - Prendiamo $A = \beta^n/2 - 1, B = \beta^n/2 - 1 \Longrightarrow S = \beta^n - 2, C_{\text{out}} = 0$

    $$S = \beta^n/2 - 1 + \beta^n/2 - 1 = 2 \cdot \beta^n/2 - 2 = \beta^n/2 - 2$$

    cioè, se fossi in base 2,

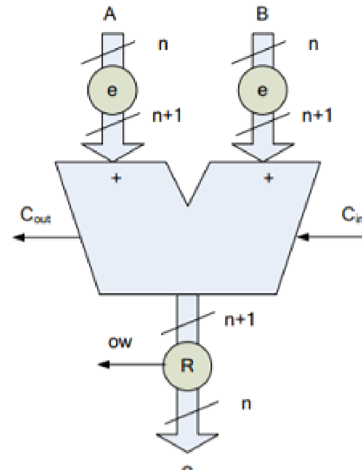
    $$S = (111\ldots101)_2$$

    il numero ottenuto è negativo, ma noi abbiamo sommato due numeri positivi. Lavoriamo su n cifre e questi numeri, nell'esempio immaginato in base 2, possono essere rappresentanti con $n - 1$ cifre: il riporto uscente è nullo ma abbiamo un overflow!

- L'idea è utilizzare un sommatore con $n+1$ cifre
  - Si estendono gli operandi da n ad $n+1$ cifre. Osserviamo che

$$\begin{aligned} S^{EST} &= |s|_{\beta^{n+1}} = |a+b|_{\beta^{n+1}} = \\ &= ||a|_{\beta^{n+1}} + |b|_{\beta^{n+1}}|_{\beta^{n+1}} \\ &= |A^{EST} + B^{EST}|_{\beta^{n+1}} \end{aligned}$$

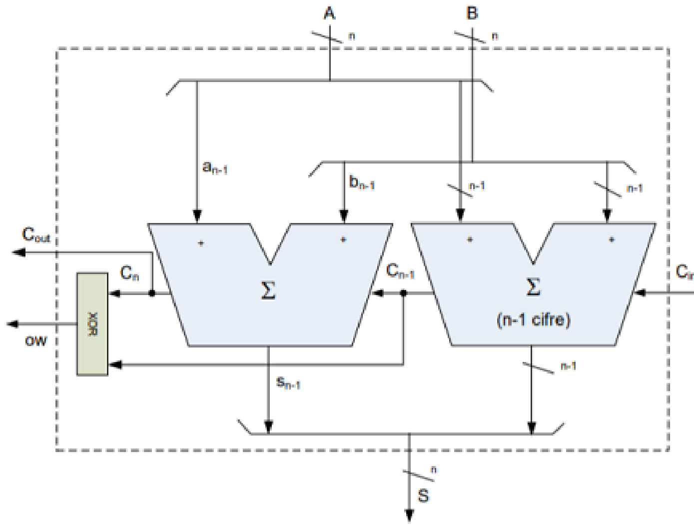
    la rappresentazione estesa è la somma delle rappresentazioni estese modulo β^n.
  - Dopo aver sommato effettuo una riduzione di campo del risultato da $n+1$ cifre ad n cifre.
  - La variabile OW ottenuta dalla riduzione di campo consiste nell'overflow della somma.



- **OF in base 2**: il circuito di overflow si costruisce attraverso una porta XOR avente in ingresso le cifre più significative (riduzione possibile solo con le ultime due cifre uguali).

$$\text{OW} = s_n \oplus s_{n-1} = (a_n \oplus b_n \oplus cin_n) \oplus (a_{n-1} \oplus b_{n-1} \oplus cin_{n-1}) = cin_n \oplus cin_{n-1}$$

  In entrambi i casi le estensioni sono state ottenute replicando le cifre più significative: a_n si semplifica con a_{n-1}, mentre b_n si semplifica con b_{n-1}. Rimangono solo i riporti entranti: segue che possiamo ottenere lo stesso risultato attraverso una porta XOR avente in ingresso i carry in ingresso negli ultimi sommatori. Si osservi che cin_n è il riporto uscente del penultimo sommatore (in un risultato esteso): segue che l'estensione in base $\beta = 2$ non è così necessaria, quindi $cin_n = c_{out}$



Morale della favola Non esiste un sommatore per naturali o un sommatore per interi: il sommatore è uguale in tutti i casi.

Assembler In Assembler esiste un'istruzione ADD valida sia con interi che con naturali. Il circuito utilizzato è lo stesso in entrambi i casi: cambia solo il flag da guardare (CF per i naturali, OF per gli interi). L'istruzione setta entrambi i parametri, sta a noi decidere quale guardare. Relativamente alle istruzioni JUMP con condizione legata all'overflow sta a noi scegliere quale guardare.

28.6 Sottrazione

Supponiamo di avere due numeri naturali A, B (su n cifre) tali che $a \longleftrightarrow A$ e $b \longleftrightarrow B$ (sono rappresentazioni di numeri interi). Vogliamo trovare il numero naturale D (su n cifre), rappresentazione dell'intero d, tale che

$$d = a - b$$

Possiamo dire, se d è rappresentabile su n cifre,

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

possiamo utilizzare lo stesso sottrattore per numeri naturali. Le cose aggiuntive sono praticamente le stesse viste per l'addizione

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

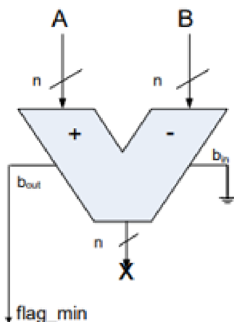
Flag OF Anche qua si individua il flag testando la riducibilità del risultato della differenza su $n+1$ cifre. Validissimo anche il metodo, già visto, per la base 2 (porte XOR).

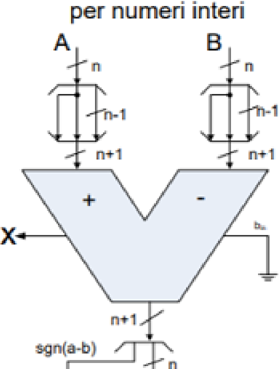
28.6.1 Comparazione di numeri interi

La comparazione dei numeri interi passa da un sottrattore esattamente come per i numeri naturali. Tuttavia:

- il prestito uscente non può essere guardato per determinare l'overflow.
- Si guarda il segno della differenza (sottrattore su $n+1$ cifre, si svolge l'estensione visto che la sottrazione non è sempre possibile su n cifre)

Per verificare l'uguaglianza, in base 2, utilizzo la stessa strategia vista con i naturali (Porte XOR da cui passo coppie di variabili logiche e porta NOR in cui convergono tutte le uscite delle porte XOR), mentre per il confronto devo verificare il segno: questo posso farlo attraverso un sottrattore ad $n+1$ cifre dove minuendo e sottraendo sono estesi. Non considero minimamente il prestito uscente





28.7 Confronto tra comparazioni

- **Comparazione di naturali**:
  - Per verificare se $A < B$ o $A > B$ utilizzo il prestito uscente.
  - Per verificare se $A = B$ o $A \neq B$ utilizzo una serie di porte XOR le cui uscite convergono in un'unica porta NOR.
- **Comparazione di interi**:
  - Per verificare se $a < b$ o $a > b$ con $A \leftrightarrow a$ e $B \leftrightarrow b$ non posso più utilizzare il prestito uscente: estendo minuendo e sottraendo (utilizzando logica) in modo da avere rappresentabili anche i risultati che non stanno su n cifre.
    A quel punto prendo dal risultato la cifra più significativa e scarto tutte le altre.
  - Per verificare se $a = b$ o $a \neq b$ con $A \leftrightarrow a$ e $B \leftrightarrow b$ utilizzo la stessa strategia dei naturali.

28.8 Moltiplicazione e divisione

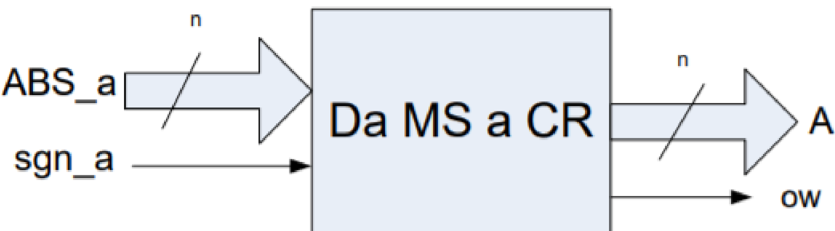
28.8.1 Approccio

- Vogliamo utilizzare la stessa circuiteria introdotta parlando dei numeri naturali.
- Tuttavia le proprietà viste per i numeri naturali (che permettono di applicare sommatore e sottrattore in modo quasi uguale) non sono valide per moltiplicazione e divisione.
  - potrei ottenere dalla moltiplicazione, dati due numeri positivi, un numero negativo
  - potrei ottenere dalla divisione, dati due numeri discordi, un numero positivo
- L'approccio adottato consiste nel considerare, in modo indipendente, il valore assoluto e il segno del numero intero.
- Per fare questa cosa ci servono dei convertitori da e per il complemento alla radice.
  - Il convertitore da complemento alla radice a modulo e segno lo abbiamo già introdotto

– Il convertitore da modulo e segno a complemento alla radice lo introdurremo a breve.

Il primo ci serve per gli ingressi della rete, il secondo per le uscite!

28.8.2 Circuito di conversione da MS a CR



Voglio un circuito che prende in ingresso il valore assoluto su n cifre ed il segno della rappresentazione di un numero intero.

Fattibilità L'operazione non è sempre fattibile, basta osservare gli intervalli di rappresentazione

- In ingresso posso porre solo i numeri interi appartenenti all'intervallo $[-(\beta^n - 1); +(\beta^n - 1)]$
- In uscita posso ottenere solo i numeri interi appartenenti all'intervallo $[-\beta^n/2; \beta^n/2 - 1]$

se per il primo convertitore abbiamo detto che il secondo insieme è contenuto nel primo chiaramente non possiamo dire il contrario. Se l'operazione è possibile possiamo dire (ragioniamo in modo distinto su positivi e su negativi)

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

cosa fattibilissima con un multiplexer a due vie:

- cortocircuito;
- circuito per il calcolo dell'opposto.

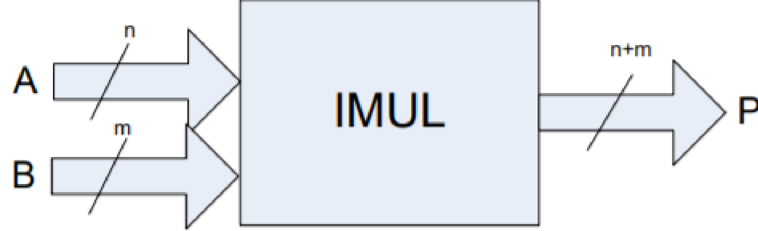
Per quanto riguarda l'overflow diciamo

$$\boxed{\text{ow} = 1 \Longleftrightarrow (\text{ABS_a} > \beta^n/2) \vee (\text{ABS_a} = \beta^n/2 \wedge \text{sgn_a} = 0)}$$

- Con la prima condizione verifico che il valore assoluto sia contenuto nell'intervallo $\left[0, \frac{\beta^n}{2}\right]$, quindi che $a \in \left[-\frac{\beta^n}{2}, \frac{\beta^n}{2}\right]$
- Con la seconda verifico se il valore assoluto è uguale a $\beta^n/2$ e che il segno sia positivo (se con questo modulo ho segno positivo allora ho l'estremo destro non rappresentabile). Ricordiamo che l'intervallo di rappresentabilità per gli interi è il seguente:

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

28.8.3 Moltiplicazione



Dati A su n cifre e B su m cifre tali che $a \longleftrightarrow A$ e $b \longleftrightarrow B$ voglio calcolare P su $n+m$ cifre tale che $p \longleftrightarrow P$ e

$$p = a \cdot b$$

Rappresentabilità Non abbiamo problemi di rappresentabilità

$$-\frac{\beta^n}{2} \cdot \frac{\beta^m}{2} \leq p \leq \frac{\beta^n}{2} \cdot \frac{\beta^m}{2} \implies \boxed{-\frac{\beta^{n+m}}{4} \leq p \leq \frac{\beta^{n+m}}{4}}$$

Funzione segno Ricordiamo che la funzione segno consiste nella seguente

$$\operatorname{sgn}(x) = \begin{cases} +1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

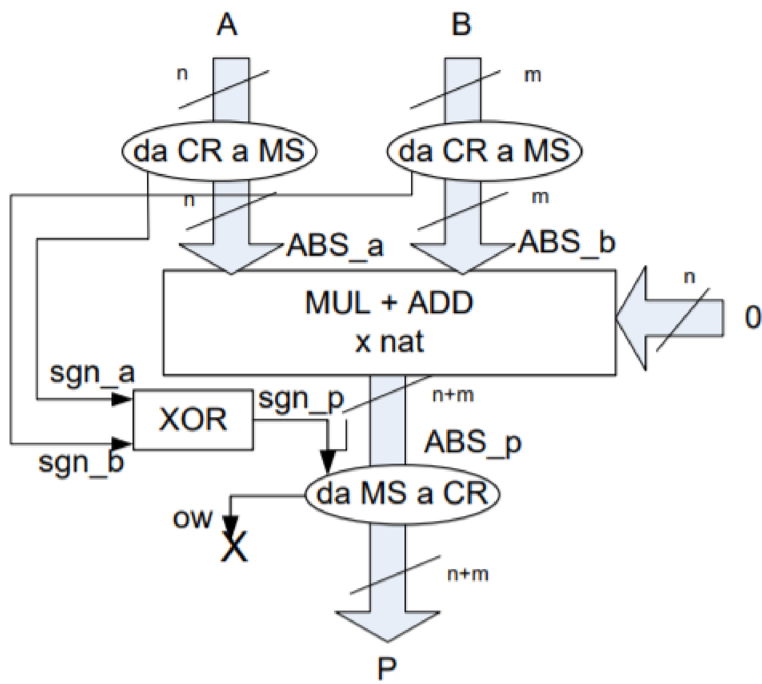
possiamo dire $x = \operatorname{sgn}(x) \cdot \text{ABS}(x)$. segue

$$p = \begin{cases} \text{ABS}(a) \cdot \text{ABS}(b) & a, b \text{ concordi} \\ -\text{ABS}(a) \cdot \text{ABS}(b) & a, b \text{ discordi} \end{cases}$$

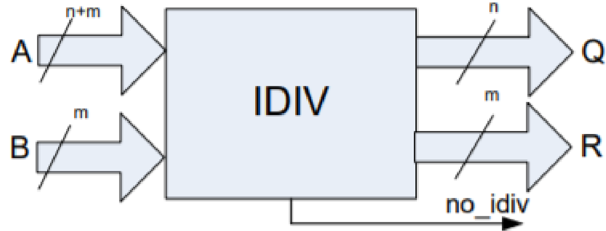
cioè valore assoluto come prodotto tra i valori assoluti e segno come prodotto tra le funzioni segno.

Sintetizzazione

- Utilizzeremo il moltiplicatore con addizionatore per naturali, prendendo come fattori i valori assoluti.
- Relativamente ai segni non si fa un prodotto ma si usa una porta XOR (che restituirà il segno)
- In ingresso si usano due convertitori da complemento alla radice a modulo e segno (che mi restituiscono i segni da passare nelle porte XOR e i valori assoluti da porre nel moltiplicatore con addizionatore per naturali).
- In uscita si usa il convertitore da modulo e segno a complemento alla radice: prendiamo come segno l'uscita della porta XOR e come valore assoluto l'uscita del moltiplicatore con addizionatore per naturali.



28.8.4 Divisione



Dati A su $n+m$ cifre e B su m cifre, con $A \leftrightarrow a, B \leftrightarrow b$ voglio ottenere Q, R, rispettivamente su n ed m cifre (quindi stessa allocazione vista coi naturali), tali che $Q \leftrightarrow q, r \leftrightarrow R$ e

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

Come nei naturali dobbiamo tener conto della presenza di divisioni non possibili, quindi esprimiamo un ulteriore uscita che ci segnala la fattibilità della divisione.

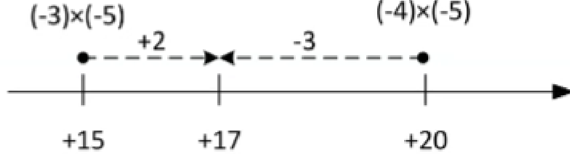
Attenzione all'unicità Il teorema della divisione con resto, che garantisce l'unicità, non è valido in questa situazione. Il divisore non è un naturale, quindi dire $0 \leq r \leq b-1$ non ci garantisce più quello che potevamo dire coi naturali. Devo trovare un vincolo che generalizzi il precedente.

Nuove condizioni per l'unicità Potrei dire che il valore assoluto del resto è minore del valore assoluto del divisore

$$\text{ABS}(r) < \text{ABS}(b)$$

ma questo non mi garantisce l'unicità. Prendiamo una divisione con $a = +17, b = -5$:

- Prima coppia: $q = -3, r = +2$
- Seconda coppia: $q = -4, r = -3$



l'unicità non è rispettata. Segue la necessità di introdurre un'altra condizione:

$$\text{sgn}(r) = \text{sgn}(a)$$

cioè che il segno del resto sia uguale al segno del dividendo.

$$\boxed{\begin{cases} \text{ABS}(r) < \text{ABS}(b) \\ \text{sgn}(r) = \text{sgn}(a) \end{cases}}$$

Approssimazione del quoziente Il quoziente, con queste ipotesi, è approssimato per troncamento, quindi $q \cdot b$ è sempre più vicino all'origine rispetto ad a (contrariamente alla divisione vista fino ad ora, che approssima a sinistra).

Formula Riscriviamo la formula per ottenere il dividendo così

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

con l'ipotesi di uguaglianza dei segni, appena fatta, sostituisco ottenendo

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

moltiplico per $\text{sgn}(a)$

$$\text{ABS}(a) = [q \cdot \text{sgn}(a) \cdot \text{sgn}(b)] \cdot \text{ABS}(b) + \cdot \text{ABS}(r)$$

ricordiamo che $\text{sgn}^2(a) = 1$, quindi possiamo semplificare. Teniamo conto che se nella formula $\text{ABS}(a), \text{ABS}(b), \text{ABS}(r)$ sono naturali lo sarà anche $q \cdot \text{sgn}(a) \cdot \text{sgn}(b)$. Pongo

$$\text{ABS}(q) = q \cdot \text{sgn}(a) \cdot \text{sgn}(b)$$

28.8.4.1 Condizioni di fattibilità

La fattibilità della divisione tra interi passa dalla fattibilità

- della divisione tra moduli, e
- dalla fattibilità della conversione da MS a CR del quoziente.

Fattibilità della divisione tra moduli La divisione può essere ottenuta utilizzando un divisore tra naturali, purchè ABS(q) sia un numero naturale rappresentabile su n cifre. Per verificare quanto detto riprendo la solita condizione tipica delle divisioni tra i naturali, e la adeguo alla nuova situazione

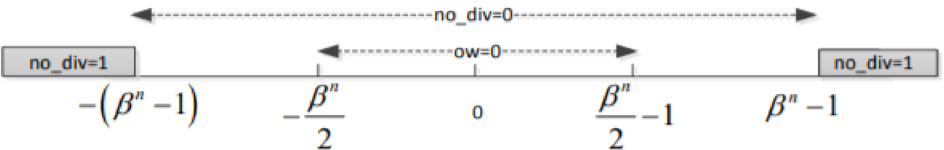
$$\mathrm{ABS}(a) < \beta^n \cdot \mathrm{ABS}(b)$$

la cosa si verifica con un sottrattore (il sottrattore è nel divisore per naturali). Se la disuguaglianza è falsa allora avrò ABS(q) $\geq \beta^n$, cioè

- $q \geq \beta^n$ se q è positivo
- $q \leq -\beta^n$ se q è negativo.

Segue (attenzione alla freccia)

ABS(q) non è un naturale rappresentabile su n cifre $\Longrightarrow$ q non è un intero rappresentabile su n cifre



Conversione da MS a CR del quoziente Quanto visto poco prima è una condizione necessaria, ma non sufficiente. Non devo verificare solo la fattibilità della divisione tra i moduli degli interi, ma anche la rappresentabilità dell'intero su n cifre, cioè

$$-\frac{\beta^n}{2} \leq q \leq \frac{\beta^n}{2} - 1$$

la cosa si verifica facilmente utilizzando l'overflow del convertitore da MS a CR per il quoziente. Il non overflow è condizione necessaria e sufficiente per la fattibilità della divisione tra numeri interi.

Recap

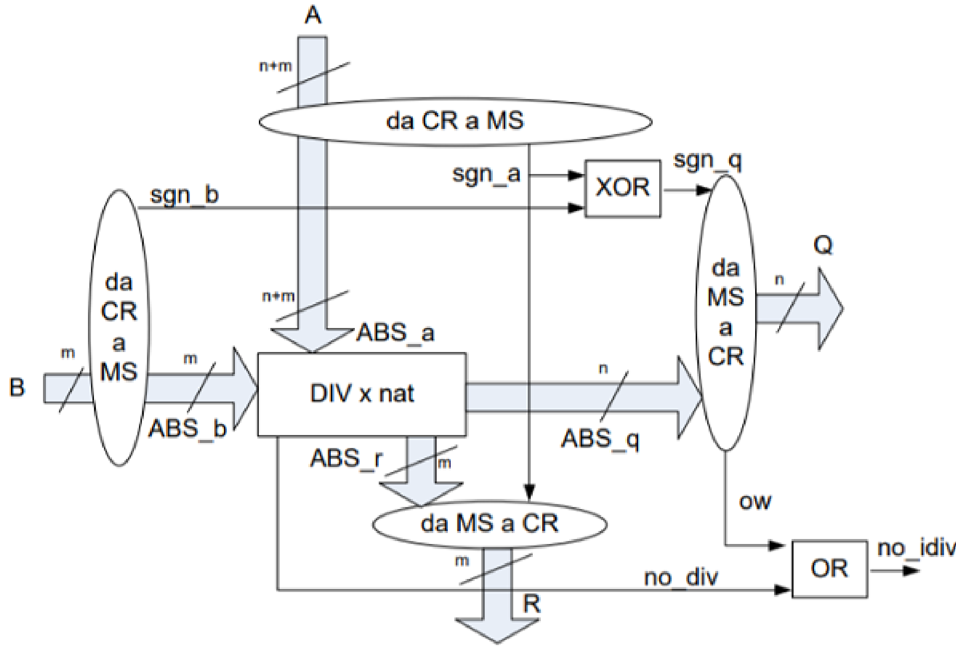
- La non fattibilità della divisione tra numeri naturali implica automaticamente l'overflow

$$\mathrm{no_div} = 1 \Longrightarrow \mathrm{ow} = 1$$

- Non possiamo dire il contrario, cioè ow = 1 non implica automaticamente no_div = 1
- Si gestiscono entrambe le condizioni con una porta OR: basta una condizione non soddisfatta per dire

$$\mathrm{no_idiv} = 1$$

28.8.4.2 Circuito



- Ho in ingresso A e B, con $A \leftrightarrow a, B \leftrightarrow b$
- Passo A e B nel convertitore da CR a MS, ottenendo valori assoluti e segni.
- Utilizzo il divisore per naturali per svolgere la divisione tra i valori assoluti degli interi.
- Passo il resto nel convertitore da CR a MS, prendendo come segno quello del dividendo (come abbiamo stabilito prima)
- Passo il quoziente nel convertitore da CR a MS, prendendo come segno il risultato di una porta XOR avente in ingresso il segno del dividendo e quello del divisore (solite regole imparate alle elementari)
- Per determinare la fattibilità della divisione prendo il no_div del divisore per naturali e l'ow del convertitore da MS a CR per il quoziente. Li passo in una porta OR. Ricordiamo che dobbiamo verificare:
  - la fattibilità della divisione tra numeri naturali;
  - la fattibilità della divisione tra numeri interi.

  Se una delle due cose non è possibile allora non è possibile svolgere la divisione tra interi.

Capitolo 29

Riferimenti ad altri argomenti

29.1 Assembler

Addizione nei numeri naturali

L'istruzione macchina ADD setta il CF quando il risultato non è un numero naturale rappresentabile sul numero di cifre degli operandi.

Moltiplicazione tra numeri naturali (non potenze)

L'istruzione Assembler MUL ha un solo operando. A seconda della dimensione dell'operando, seleziona l'altro operando (implicito: AL, AX, EAX), e mette il risultato in un destinatario implicito (AX, DX_AX, EDX_EAX). In pratica, abbiamo $n = m$, e quindi circuiti che calcolano il prodotto calcolano risultati su $2 \cdot n$ bit partendo da operandi a n bit.

Divisione tra numeri naturali (non potenze)

[1] La DIV ammette dividendo su $2 \cdot n$ bit e divisore su n bit, con $n = 8, 16, 32$, e richiede che il quoziente stia su n bit (genera un'interruzione in caso contrario). Il dividendo è selezionato implicitamente sulla base della lunghezza del divisore. In questo caso, è cura del programmatore assicurarsi che $X < \beta^n \cdot Y$, eventualmente estendendo la rappresentazione del dividendo (e del divisore) su un numero maggiore (doppio) di bit. Così facendo X ed Y rimangono identici. Poter disporre di $n = 32$ significa poter garantire che quella disuguaglianza può essere resa vera, eventualmente estendendo le rappresentazioni, per qualunque dividendo su 32 bit e qualunque divisore. Qualora il dividendo non stia su 32 bit, non è detto che la divisione si possa sempre fare, perchè non si possono estendere ulteriormente gli operandi.

Opposto

Esiste un'istruzione NEG, che interpreta una sequenza di bit come la rappresentazione di un numero intero, e produce la rappresentazione dell'opposto se questo esiste. Altrimenti setta il flag di overflow OF.

[1] Reminescenza dell'esempio 15000/3 con divisore su 8 bit. Si osservi che

$$15000 > 2^8 \cdot 3$$

Estensione di campo

Esistono istruzioni CBW, CWD, CWDE, CDQ, che interpretano una sequenza di bit come la rappresentazione di un numero intero (su 8,16,32 bit) e producono la rappresentazione dello stesso numero estesa su 16, 32, 64 bit. Per contro, non esiste nessuna istruzione dedicata allo stesso scopo per i naturali: per i naturali l'estensione si fa aggiungendo degli zeri in testa.

Addizione nei numeri interi

In Assembler esiste una sola istruzione ADD, che somma numeri naturali secondo l'algoritmo visto a suo tempo. Il risultato di tale somma è corretto anche se quei numeri sono la rappresentazione di numeri interi. La rappresentabilità del risultato si stabilisce:

- guardando se CF = 0, nel caso in cui i numeri sommati siano naturali
- guardando se OF = 0, nel caso in cui i numeri sommati siano rappresentazioni di interi

Visto che l'unico a saperlo è il programmatore, la ADD setta sia CF che OF, e sarà poi il programmatore a testare la condizione giusta, coerentemente con quella che sa essere l'interpretazione corretta. Tipicamente, l'istruzione che segue una ADD sarà una JC/JNC nel caso di somma di naturali, oppure una JO/JNO nel caso di somma di interi.

29.2 Verilog (da *Circuiti logici per le operazioni sui numeri naturali e sui numeri interi* di Paolo Corsini)

Concatenamento

Concatenamento di due numeri X ed Y (costrutto valido per qualunque dimensionamento):

```
assign z = {X,Y};
```

Scompattamento

Scompattamento di X in due numeri, rispettivamente ad $L - M$ bit ed M bit:

```
assign x_A = X[L-1:M];
assign x_B = X[M-1:0];
```

Estensione di campo per naturali

Estensione da n ad $n + 1$ bit in base $\beta = 2$:

```
assign X_EST = {1'B0, X};
```

Riduzione di campo per naturali

Riduzione da $n + 1$ ad n bit in base $\beta = 2$:

```
assign no_rid = (X[N] == 0) ? 0 : 1;
assign X_RID = X[N-1:0];
```

ricordiamo che la riduzione si può fare solo se la cifra più significativa è uguale a zero.

Circuito moltiplicatore per β^n, sia naturali che interi

Prodotto per β^n, dove $\beta = 2$ e $n = 1$:

```
assign P = {X, 1'B0};
```

Circuito divisore per β^M, sia naturali che interi

Divisione per β^M, dove $\beta = 2$ e $M = 2$

```
assign Q = X[N-1:2];
assign R = X[1:0];
```

N è la dimensione del numero X.

Circuito complementatore

```
assign X_COMPL = ~X;
```

Circuito sommatore per naturali

```
assign {c_out, S} = X + Y + c_in;
```

Circuito sottrattore per naturali

```
assign {b_out, D} = X - Y - b_in;
```

Circuito comparatore per naturali

```
assign flag_min = (X < Y) ? 1 : 0;
assign flag_eq = (X == Y) ? 1 : 0;
```

Circuito moltiplicatore con addizionatore

Modulo in base 2

```
assign P = {x[N-1]&y, X[N-2]&y, ..., X[0]&y} + C;
```

Ricordarsi

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

Non ci serve un multiplexer, ma una serie di porte AND: dobbiamo scegliere tra X e una costante.

Moltiplicatore

Moltiplicazione in $\beta = 2$:

```
assign P=X*Y;
```

Circuito divisore per naturali

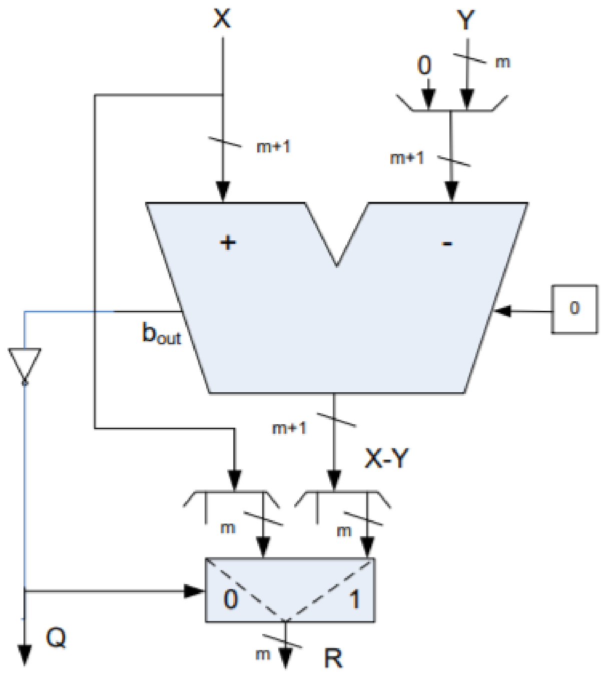
Modulo in base 2

```
assign no_div = (X < {Y, 1'B0}) ? 0: 1;
assign {b_out, D} = X - {1'B0, Y};
assign Q = ~b_out;
assign R = (Q == 0) ? X[M-1:0] : D[M-1:0];
```

Attenzione all'estensione di campo nella seconda istruzione, e alla successiva riduzione di campo nell'ultima istruzione. Ricordarsi che

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

la fattibilità è verificabile ricordandosi che dobbiamo avere $X < \beta \cdot Y$



Divisore

Il divisore in base $\beta = 2$ può essere espresso così:

```
assign Q = X/Y;
assign R = X%Y;
```

La fattibilità si verifica ricordando che $X < \beta^n \cdot Y$ equivale a dire: *le m cifre più significative del dividendo rappresentano un numero più piccolo del divisore.*

X:	x_{n+m-1}	x_{n+m-2}	...	x_{n+1}	x_n	x_{n-1}	...	x_0
$\beta^n \cdot Y$:	y_{m-1}	y_{m-2}	...	y_1	y_0	0	...	0

```
assign no_div = (X[L-1:N-1] < {Y, 1'B0}) ? 0 : 1;
```

dove $L = N + M$.

Circuito per la conversione dalla rappresentazione in complemento alla radice alla rappresentazione in modulo e segno e viceversa

Da CR a MS

Circuito in base $\beta = 2$:

```
assign segno_a = A[N-1];
assign ABS_a = (segno_a == 0) ? A : (~A +1);
```

Da MS a CR

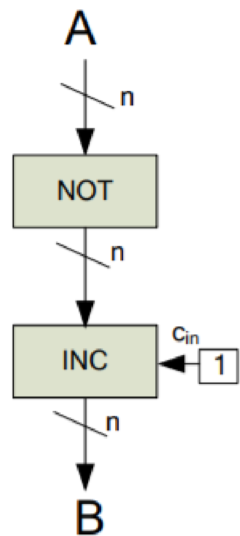
Circuito in base $\beta = 2$

```
wire [N-1:0] DUE_alla_N_mezzi = { 1'B1, (N-1){'B0} };
assign ow = (ABS_a > DUE_alla_N_mezzi) | (ABS_a == DUE_alla_N_mezzi & segno_a == 0);
assign A = (segno_a == 0) ? ABS_a : (~ABS_a + 1);
```

Ricordiamo la condizione di overflow:

$$\boxed{\text{ow} = 1 \iff (\text{ABS_a} > \beta^n/2) \vee (\text{ABS_a} = \beta^n/2 \wedge \text{sgn_a} = 0)}$$

e il grafico relativo al circuito



Circuito estensore di campo per interi

Estensione di un bit in base $\beta = 2$

```
assign A_EST = {A[N-1], A};
```

Circuito riduttore di campo per interi

Riduzione da $n + 1$ bit ad n bit in base $\beta = 2$

```
assign ow = A[N-1]^A[N-2];
assign A_RID = A[N-2:0];
```

Circuito dell'opposto

Circuito dell'opposto valido in base $\beta = 2$

```
wire [N-1:0] DUE_alla_N_mezzi = {1'B1, (N-1){'B0} };
assign ow = (A=DUE_alla_N_mezzi) ? 1 : 0;
assign A_opp = ~A + 1;
```

Ricordarsi che si va in overflow se $A = \frac{\beta^n}{2}$ ($A = 100\ldots0$ in base $\beta = 2$).

Circuito sommatore negli interi

Circuito sommatore in base $\beta = 2$ con uscita overflow:

```
assign S_est = {A[N-1], A} + {B[N-1], B};
assign ow = (S_est[N] == S_est[N-1]) ? 0 : 1;
assign S = S_est[N-1:0];
```

Estendo gli operandi, ottengo un risultato esteso, verifico mediante le due cifre più significative se si è andati in overflow o meno.

Circuito sottrattore negli interi

Circuito sottrattore in base $\beta = 2$ con uscita overflow:

```
assign D_est = {A[N-1], A} - {B[N-1], B};
assign ow = (D_est[N] == D_est[N-1]) ? 0 : 1;
assign D = D_est[N-1:0];
```

Estendo gli operandi, ottengo un risultato esteso, verifico mediante le due cifre più significative se si è andati in overflow o meno.

Circuito comparatore negli interi

Circuito comparatore in base $\beta = 2$ con uscita overflow:

```
assign D_est = {A[N-1], A} - {B[N-1], B};
assign flag_min = D_est[N];
assign flag_eq = ~( | D_est);
```

Estendo gli operandi, ottengo un risultato esteso, trovo chi è più grande guardando il segno (cifra più significativa), verifico se i due numeri sono uguali con porte NOR (non è proprio il metodo migliore per Stea, ma Corsini lo mette come esempio).

Circuito del valore assoluto

Circuito valido in base $\beta = 2$

```
ABS_a = (A[N-1] == 0) ? A : ~A + 1;
```

Circuito moltiplicatore negli interi

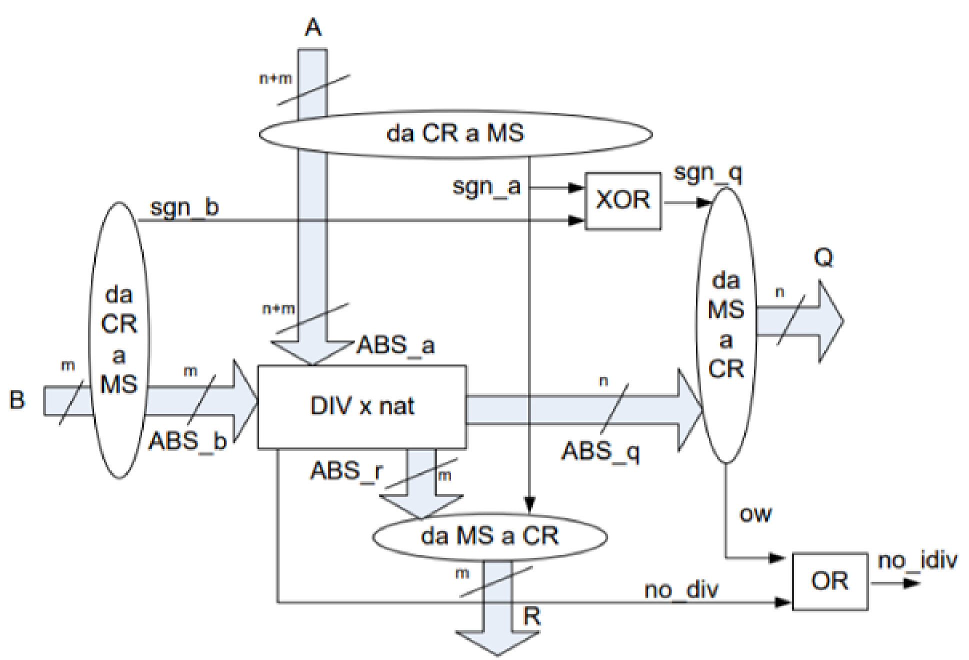
Circuito valido in base $\beta = 2$

```
assign segno_a = A[N-1];
assign ABS_a = (segno_a ==0) ? A : ~A + 1;

assign segno_b = B[N-1];
assign ABS_b = (segno_b ==0) ? B : ~B + 1;

assign segno_p = segno_a^segno_b;
assign ABS_p = ABS_a*ABS_b;

assign P = (segno_p == 0) ? ABS_p : ~ABS_p + 1;
```

Circuito divisore negli interi

Circuito valido in base $\beta = 2$

```
assign segno_a = A[N-1];
assign ABS_a = (segno_a ==0) ? A : ~A + 1;

assign segno_b = B[N-1];
assign ABS_b = (segno_b ==0) ? B : ~B + 1;

assign no_div = (ABS_a[L-1:N-1] < {ABS_b, 1'B0}) ? 0 : 1;
assign segno_q = segno_a^segno_b;
assign ABS_q = ABS_a/ABS_b;

assign segno_r = segno_a;
assign ABS_r = ABS_a%ABS_b;

wire [N-1:0] DUE_alla_N_mezzi = {1'B1, (N-1){'B0} };
assign ow_conv_rapp_q = (ABS_q > DUE_alla_N_mezzi) | (ABS_q == DUE_alla_N_mezzi & segno_q == 0);

assign no_idiv = no_div | ow_conv_rapp_q;
assign Q = (segno_q == 0) ? ABS_q : (~ABS_q + 1);
assign R = (segno_r == 0) ? ABS_r : (~ABS_r + 1);
```



Parte VI

Reti sequenziali

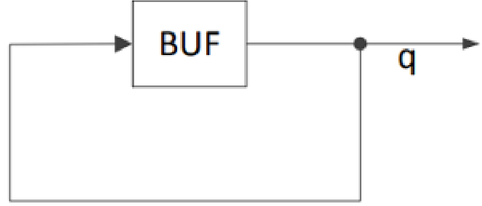
Capitolo 30

Martedì 03/11/2020

30.1 Reti con memoria

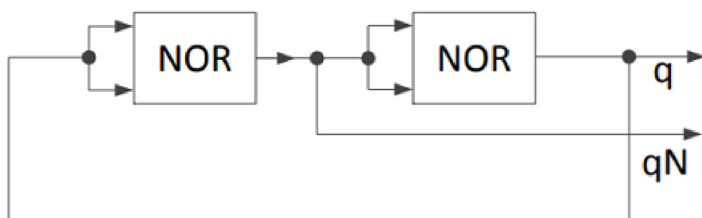
Fino ad ora abbiamo parlato di reti combinatorie *memoryless*, cioè senza memoria: in queste l'uscita dipende esclusivamente dallo stato in ingresso. Lavoreremo per oltre un mese sulle cosiddette **reti sequenziali**, cioè reti in cui lo stato di uscita dipende dallo storico degli ingressi.

Come ricordiamo queste sequenze di stati? Si utilizzano i cosiddetti **anelli di retroazione**: si prende l'uscita e la si riporta in ingresso. In un certo senso ho un valore logico che gira all'interno di un anello, pronto per essere utilizzato. Questa cosa, attenzione, può avvenire solo in presenza di un buffer, che mi permette di rigenerare quel valore): se io pongo il buffer in cortocircuito ho un filo staccato che non ha valore logico.



Stati interni dell'anello Si dice che l'anello può trovarsi in due stati: S_0 ed S_1. Questi sono detti **stati interni**.

Esempio delle dispense I primi esempi sono inutili sul piano pratico: lo stato dell'anello verrà posto in modo randomico quando colleghiamo il buffer alla corrente. Nel secondo esempio sostituiamo il buffer con due porte NOT implementate attraverso porte NOR.



Fondamentalmente:

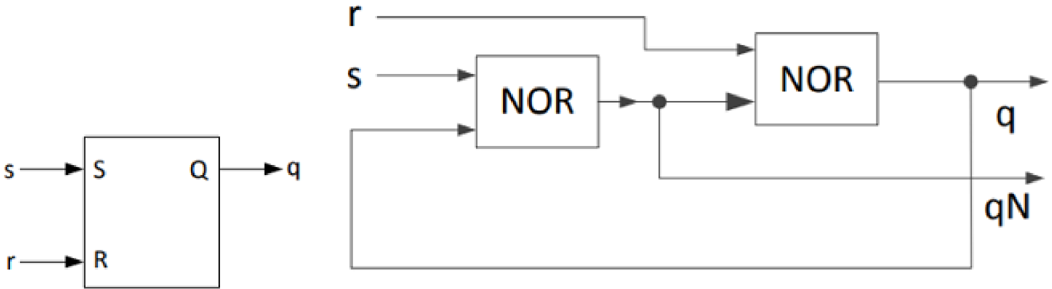
- Ho in ingresso $q = 0$, la prima porta NOR lo fa diventare 1, la seconda lo fa tornare 0
- Ho in ingresso $q = 1$, la prima porta NOR lo fa diventare 0, la seconda lo fa tornare 1

Sempre in questo esempio poniamo un'uscita detta q_N, cioè la negazione del valore di q. Questo significa che un anello si trova in uno stato stabile se q e q_N risultano discordi. Si dice che il circuito memorizza il valore di q.

Attenzione al valore delle porte NOR Quando do tensione alle due porte il valore in uscita di questo è assolutamente randomico. Questo significa che posso avere una situazione in cui q e q_N sono concordi e un'altra in cui sono discordi (ricordiamo che q_N è il valore tra le due porte NOR, quindi il valore uscente dalla prima; q è invece il valore uscente dalla seconda porta NOR)!

- Se q e q_N sono discordi la rete si trova già in uno dei due stati stabili, e li rimane per sempre (o perlomeno finchè c'è corrente)
- Se q e q_N sono concordi la situazione si fa più complessa:
  - In teoria dovrei avere una rete che oscilla all'infinito tra uno stato e un altro (ciò avviene se le porte hanno sempre lo stesso tempo di risposta, cosa impossibile)
  - In pratica abbiamo una rete che si stabilizza velocemente grazie al tempo di risposta diverso delle due porte

30.2 Latch SR



I primi esempi sono inutili sul piano pratico perchè non siamo in grado di pilotare lo stato interno dell'anello. Abbiamo bisogno di introdurre degli ingressi. Il Latch SR (che non si chiama *flip-flop SR*) consiste in una rete che oltre ad avere l'uscita q vista negli esempi presenta due ingressi:

- s, che sta per SET
- r, che sta per RESET.

Queste sono dette **attive alte** perchè un ingresso pari ad 1 significa eseguire una certa funzione (cioè settare o resettare la variabile logica salvata nell'anello).

Quali valori posso porre?

$$\boxed{\text{Ricordiamo la porta NOR: } z = 1 \iff x_1 = x_0 = 0}$$

- $s = 1$, $r = 0$. La prima porta NOR avrà per forza come valore 0, quindi $q_N = 0$. Segue $q = 1$.
- $s = 0$, $r = 1$. La seconda porta NOR avrà per forza come valore 0, quindi $q = 0$. Con 0 in ingresso nella prima porta avrò $q_N = 1$
- $s = 1$, $r = 1$. Questo stato di ingresso è insensato e contraddice le regole di pilotaggio: in un corrett pilotaggio questo stato non è permesso.

Studiamo il comportamento del Latch SR utilizzando una **tabella di applicazione** (non confondere con la tavola di verità)!

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

- Da una parte poniamo q e q': la prima consiste nel valore attuale, la seconda nel valore che vogliamo assumere
- Dall'altra abbiamo s ed r, le variabili in ingresso che dobbiamo porre per ottenere q'

Osserviamo che

- $q = 0$, $q' = 0$. Se voglio ottenere 0 basta non settare la variabile, quindi $s = 0$. A quel punto lo stato di r è ininfluente: se pongo 1 andrò ad impostare $q' = 0$, se pongo 0 lascerò lo stato inalterato.
- $q = 0$, $q' = 1$, Se voglio ottenere 1 devo settare la variabile, quindi $s = 1$. r è per forza uguale a 0
- $q = 1$, $q' = 0$. Se voglio ottenee 0 devo resettare la variabile, quindi $r = 1$. A quel punto $s = 0$ per forza.
- $q = 1$, $q' = 1$. Osserviamo che $r = 0$ per forza (altrimenti non potrei ottenere 1). Il valore di s è insignificante.

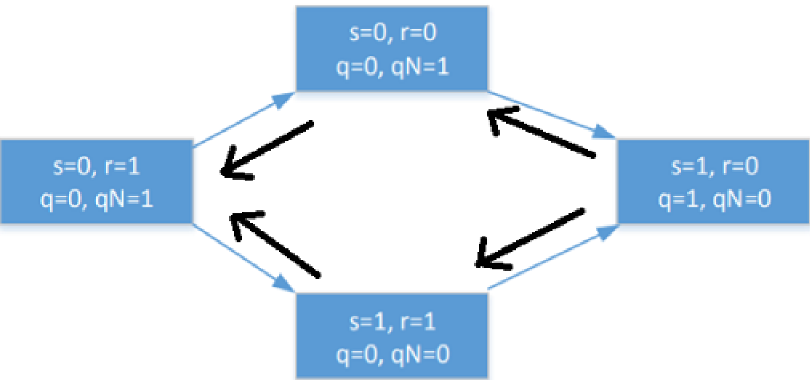
30.2.1 Pilotaggio di un Lach SR

Conosciamo già le regole di pilotaggio

- Piliotaggio in modo fondamentale: alterare gli ingressi quando la rete è a regime
- Stati di ingresso consecutivi adiacenti

La prima regola deve essere rispettata in ogni caso. Mentre la seconda regola può essere "ignorata" quando parliamo di Latch SR: questa rete è robusta rispetto a pilotaggi scorretti.

- Supponiamo di avere in ingresso lo stato $s = 1, r = 0$, e che si passi a $s = 0, r = 1$. Quindi passeremo da $q = 1, q_N = 0$ a $q = 0, q_N = 1$
- Sappiamo che il tempo di risposta per ogni singola variabile è diverso, quindi la rete può percepire due possibili stati intermedi: 00 e 11 (dipende da quale cambiamento di variabile sarà percepito per primo)
- Relativamente a 00 non ci sono problemi: avremo subito $q = 0, q_N = 1$
- Relativamente a 11 non ho problemi lo stesso: è inevitabile che entrambe le uscite, per un certo momento, saranno uguali a 0 (i tempi di risposta delle porte sono diversi, è normale succeda)



Come è evidente dal grafico la cosa è valida anche in altre situazioni: l'unico caso in cui ciò non avviene è quando vogliamo passare da $s = 1, r = 1$ a $s = 0, r = 0$. In questo caso uno dei due stati intermedi possibili può fare problemi (l'uscita sarà casuale). Resta il fatto che lo stato $s = 1, r = 1$ è inconcepibile per la logica del Latch SR e quindi è uno stato che la rete non deve impostare.

30.3 Il problema dello stato iniziale

Abbiamo detto che il Latch SR è alla base dei circuiti di memoria. Sappiamo già, da Assembler, che all'accensione del calcolatore alcuni elementi di memoria avranno un contenuto casuale (celle della memoria RAM, ad esempio), altre no (per esempio il contenuto dei registri EF ed EIP) perchè è necessario che non abbiano valore casuale.

Domanda Ma non basta pilotare la Latch SR? Dire questo significa porre delle variabili di ingresso a loro volta dipendenti da un'altra rete. Come usciamo da questo loop?

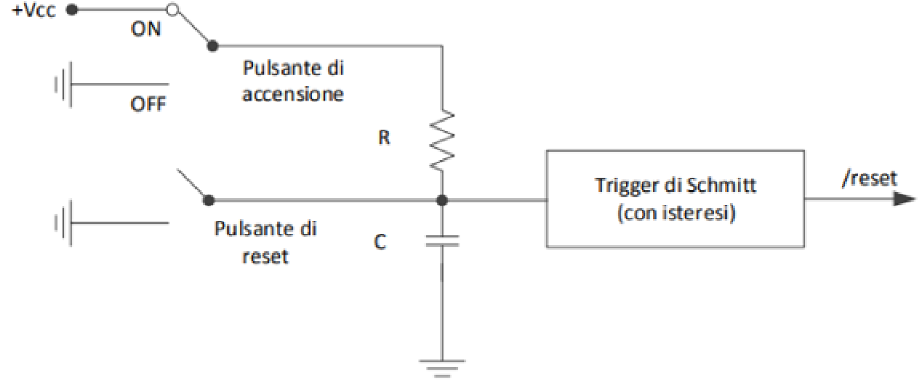
Inizializzazione Serve un metodo per inizializzare un elemento di memoria al valore voluto. Ciò avviene quando premiamo il pulsante di reset del calcolatore. La strategia che indicheremo ci permetterà di porre i valori iniziali desiderati. Distinguiamo due fasi di attività del calcolatore:

- la **fase di normale operatività**, quella che abbiamo analizzato fino ad ora

- la **fase di reset iniziale**, in cui inizializziamo certi contenuti.

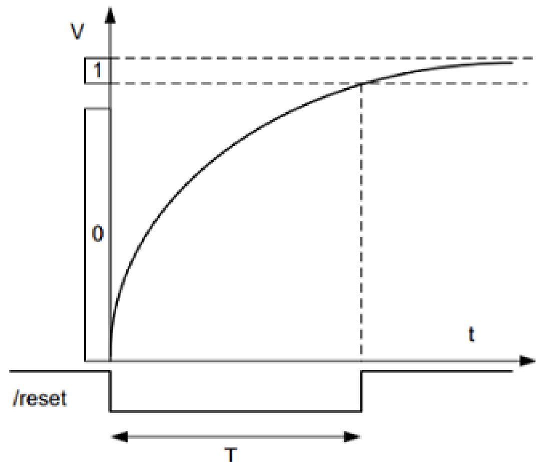
Osservazione per evitare confusione: con reset intendiamo sia la fase appena citata che una delle variabili di ingresso per pilotare il Latch SR.

Esempio di circuiteria



- Abbiamo un pulsante di accensione, che mi permette di far passare corrente all'interno della rete.
- Se il pulsante di accensione è su ON il condensatore del circuito RC si carica in tempi dell'ordine del microsecondo. A un certo punto avrò la tensione del condensatore prossima a V_{cc}. Il tempo è molto lungo per un circuito e dipende dalla resistenza presente.
- Premendo il pulsante di reset il condensatore si scarica a massa: ciò avviene solo se il contatto con il pulsante di reset dura abbastanza.
- Non avendo scollegato la rete dall'alimentazione segue che il condensatore sarà di nuovo ricaricato.

- Oltre al circuito RC è presente un'ulteriore parte detta **Trigger di Schmidt**. Il suo compito è restituire in uscita 0 o 1 in base al valore della tensione (0 se ho tensione bassa, 1 se ho tensione alta).



Questa rete, a normale operatività, restituisce 1. Dalla pressione del tasto di reset fino al raggiungimento di una certa soglia, la rete restituisce 0.

- La variabile in uscita del Trigger di Schmidt è detta variabile **attiva bassa**: questo significa che il trigger *fa il suo mestiere quando è uguale a* 0. Una variabile attiva bassa si riconsoce dallo slash prima del nome, o dall'underscore dopo il nome se stiamo scrivendo con Verilog.

Variabile logica /reset La variabile logica $/reset$ è utilizzata per inizializzare gli elementi di memoria:

- Se ho 1 l'elemento di memoria funziona normalmente
- Se ho 0 l'elemento di memoria si porta nello stato interno iniziale desiderato, indipendentemente dai valori di s, r.

Come stabiliamo lo stato iniziale? Lo stato iniziale potrà essere definito solo introducendo due nuovi ingressi:

- $/preset$, dove *pre* sta per prima e *set* per impostare 1
- $/preclear$, dove *pre* sta per prima e *clear* per impostare 0

Entrambi sono attivi bassi. La variabile $/reset$ non va diretta nel Latch SR ma influenza una delle due variabili di ingresso appena introdotte per indicare il valore iniziale.

Stati iniziali possibili Osserviamo che

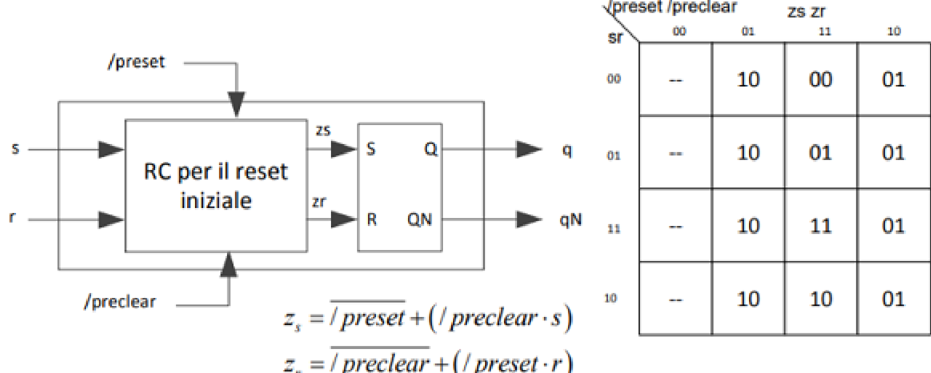
- Se $/preset = /preclear = 1$ la rete si comporta normalmente, come un Latch SR
- Lo stato $/preset = /preclear = 0$ non ha alcun significato logico: non posso porre la rete in entrambi gli stati.
- Se $/preset = 0$ la rete si porta nello stato $S1$
- Se $/preclear = 0$ la rete si porta nello stato $S0$

Sapendo che anche $/reset$ è un attivo basso affermo che

- Per avere lo stato $S1$ collego $/preset$ a $/reset$ e $/preclear$ a V_{cc}
- Per avere lo stato $S0$ collego $/preclear$ a $/reset$ e $/preset$ a V_{cc}

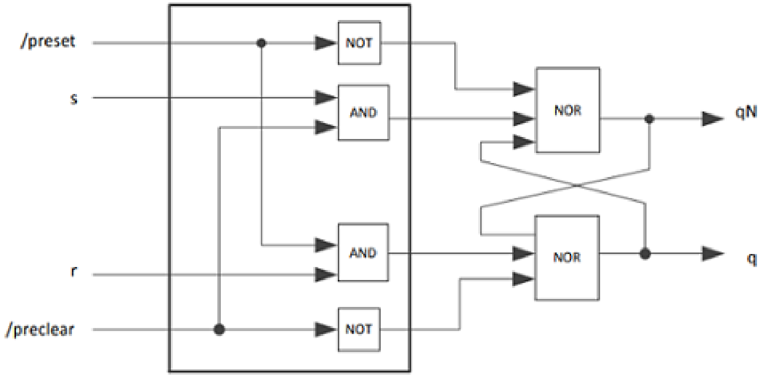
Inizialmente $/reset$ ha come valore 0, mentre $/preset$ e $/preclear$ hanno valore diverso. A un certo punto $/reset$ assume come valore 1, quindi sia $/preset$ che $/preclear$ avranno come valore 1 (una delle due variabili ha assunto questo valore fin da subito, poichè collegata alla tensione V_{cc}, l'altro è il valore ottenuto dal Trigger di Schmidt).

30.3.1 Riprendiamo la sintetizzazione del Latch-SR



- La prima colonna con stato 00 delle variabili preset e preclear è tutta segnata: abbiamo detto che non ha senso questo stato, e che non sarà mai impostato.
- La seconda colonna con stato 11 è caratterizzata da valori corrispondenti allo stato delle variabili s, r: abbiamo detto che con lo stato 11 la rete si comporta come un Latch SR, quindi ascolterà le variabili di ingresso s, r.
- Nelle colonne rimanenti le variabili s, r vengono ignorate:
  - Se ho $/preset = 0$ e $/preclear = 1$ avrò solo stati $s = 1, r = 0$. Imposto $q = 1$, quindi ho lo stato $S1$.
  - Se ho $/preset = 1$ e $/preclear = 0$ avrò solo stati $s = 0, r = 1$. Imposto $q = 0$, quindi ho lo stato $S0$.
- Interpretando i sottocubi della mappa di Karnaugh trovo le seguenti uscite:
  - $z_s = \overline{/preset} + (/preclear \cdot s)$. Porta OR avente in ingresso la preset negata e il risultato di una porta AND tra preclear ed s
  - $z_r = \overline{/preclear} + (/preset \cdot r)$. Porta OR avente in ingresso la preclear negata e il risultato di una porta AND tra preset ed r

Possiamo semplificare di più? Il Latch SR è caratterizzato da NOR, cioè da OR seguiti da una negazione. Se io unisco il Latch SR alla rete appena sintetizzata avrò due porte OR in cascata. Posso togliere queste porte ottenendo una rete con un livello di logica in meno.



30.4 Tabelle di flusso e grafi di flusso

Dobbiamo dotarci di una notazione per descrivere le reti sequenziali asincrone. Parleremo di tabelle di flusso e grafi di flusso, cioè due notazioni equivalenti e molto intuitive.

30.4.1 Tabella di flusso

Tabella in cui si descrive lo stato interno e l'uscita al variare degli stati di ingresso.

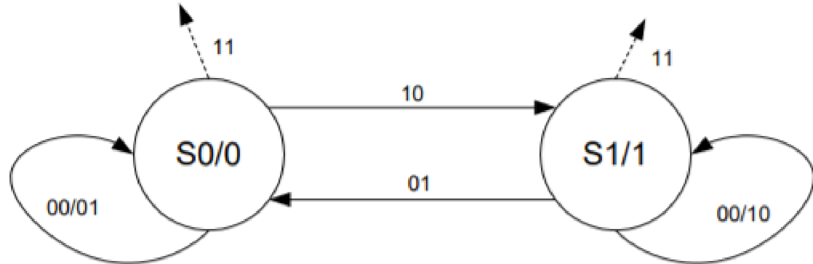
sr	00	01	11	10	q
S0	S0	S0	-	S1	0
S1	S1	S0	-	S1	1

- Ogni stato scritto a lato della matrice è detto **stato interno presente**. Nella parte superiore abbiamo gli stati di ingresso possibili (stesse regole delle mappe di Karnaugh)
- Nelle celle pongo lo **stato interno successivo**: ho lo stato iniziale (indicato nelle righe), e ho delle variabili di ingresso che pilotano la rete.
- Nella colonna di q indichiamo lo stato di uscita, che in questo caso dipende esclusivamente dallo stato interno presente.

Descriviamo l'esempio qua sopra.

- La terza colonna è tutta segnata: non avrò mai lo stato di ingresso $s = 1, r = 1$
- Sappiamo che una RSA si evolve a seguito di cambiamenti dello stato di ingresso: se mi trovo nello stato $S0$ con ingresso 00, per esempio, è ovvio che non avrò evoluzioni: le avrò solo inducendo un cambiamento nello stato di ingresso. In questa situazione di stabilità si dice che la rete è a regime, o che lo stato interno $S0$ è stabile con lo stato di ingresso 00. Quando si hanno situazioni di stabilità con certi stati questi sono cerchiati.
- Se l'elemento non è cerchiato allora significa che inducendo lo stato di ingresso s, r indicato avrò un cambiamento nello stato di ingresso.

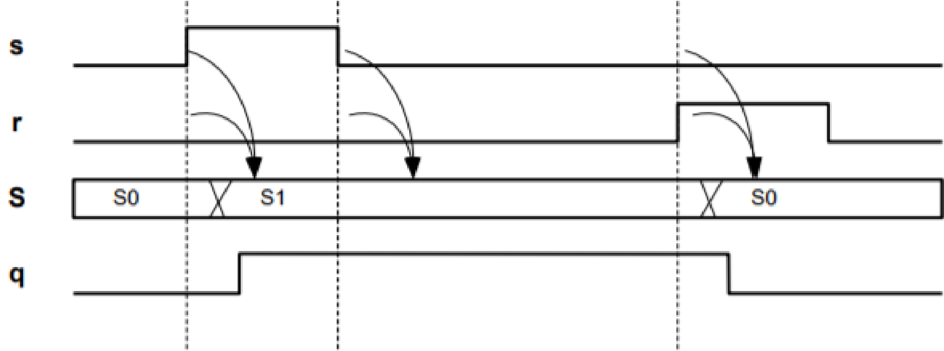
30.4.2 Grafi di flusso



Con grafi di flusso intendiamo insiemi di nodi e di archi.

- I nodi rappresentano gli stati interni.
- Gli archi sono etichettati con uno stato di ingresso.
- Se ho stabilita pongo un arco (etichettato con lo stato di ingresso con cui si ha stabilità) che va dal nodo a se stesso.
- Se con un certo stato non ho stabilità, ma un passaggio ad un altro stato, collego i due nodi con un arco (etichettato) che va dal nodo dello stato interno presente a quello dello stato interno successivi.
- Archi etichettati che vanno all'infinito sono relativi a stati di ingresso che non si devono verificare in corrispondenza di certi stati interni.

30.4.3 Diagramma di temporizzazione



Un diagramma di temporizzazione serve **per verificare se la rete si comporta come previsto** (ricordarselo più avanti, ogni volta che bisogna verificare la correttezza di quanto scritto in Verilog Stea dice di fare il diagramma di temporizzazione):

- decido uno stato iniziale;
- attribuisco valori agli ingressi nel tempo;
- osservo come si evolve la rete.

In un certo senso consiste nell'equivalente hardware del testing di un programma.

Capitolo 31

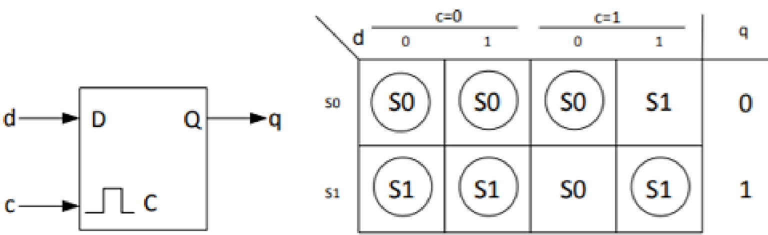
Mercoledì 04/11/2020

31.1 D-latch trasparente

Abbiamo visto che il latch SR può memorizzare 1 o 0 a seconda del comando che gli viene dato. Il D-Latch consiste in una rete sequenziale asincrona con due variabili di ingresso

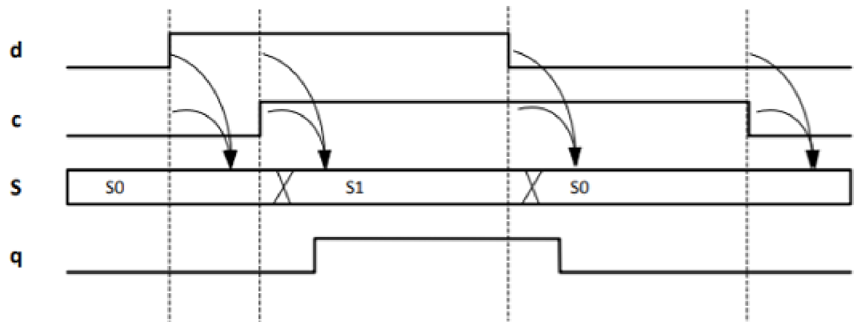
- d, data
- c, control

e una variabile in uscita q (presente anche la variabile q_N in uscita). Fondamentalmente un D-latch memorizza l'ingresso d quando c vale 1 (trasparenza), mentre quando c vale 0 è in conservazione, cioè mantiene in uscita l'ultimo valore che d ha assunto quando c valeva 1. Segue che la rete può trovarsi in due stati: uno in cui memorizza 0, un altro in cui memorizza 1.



	c=0, d=0	c=0, d=1	c=1, d=0	c=1, d=1	q
S0	(S0)	(S0)	(S0)	S1	0
S1	(S1)	(S1)	S0	(S1)	1

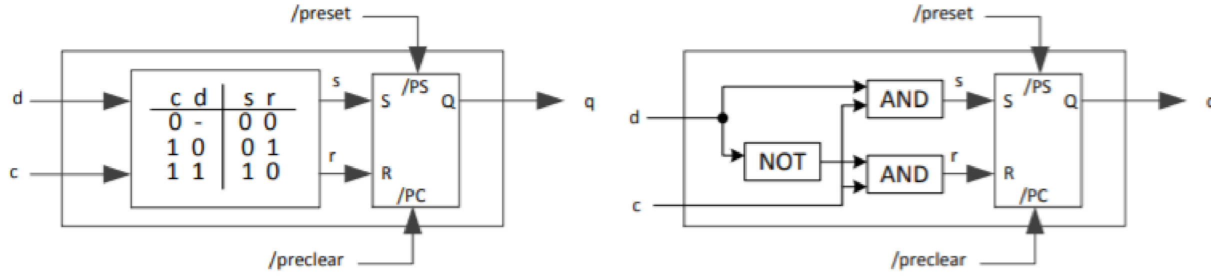
- la stabilità di ogni stato interno se $c = 0$, cioè lo stato di ingresso iniziale è anche quello successivo con qualunque valore d
- la non stabilità degli stati interni se abbiamo $c = 1$ e d che induce un cambio di valore in memoria



L'idea di base è

- Portare c ad 1
- Impostare d al valore da memorizzare
- Riportare c a 0

Sintetizzazione

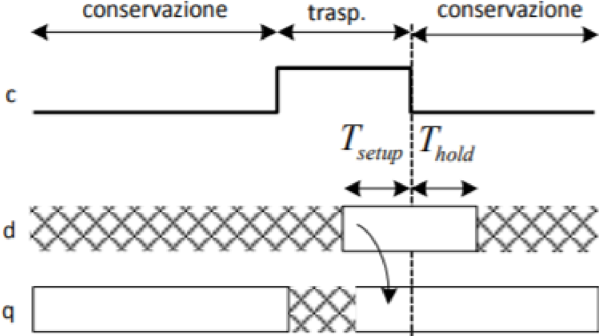


La sintetizzazione si ottiene molto facilmente osservando il comportamento della rete in base ai valori di d e c:

- Con $c = 0$ ho conservazione, quindi $s = r = 0$
- Con $c = 1$ e $d = 0$ si ha reset: $s = 0, r = 1$
- Con $c = 1$ e $d = 1$ si ha settaggio: $s = 1, r = 0$

Dalla tabella di verità individuo che $s = c \cdot d$ (il prodotto tra gli elementi mi restituisce sempre s) ed $r = c \cdot \overline{d}$. Il D-latch, visto che si basa sul Latch SR, avrà in uscita la q_N e si potranno utilizzare gli ingressi di */preset* e */preclear* per inizializzare il D-Latch al momento del reset.

Regole di pilotaggio



Le regole stabiliscono che d sia costante nella transizione di c da 1 a 0, precisamente d dovrà rimanere costante per un tempo T_{setup} prima del cambiamento di c e per un tempo T_{hold} dopo il cambiamento di c. Questa cosa è necessaria affinchè la rete non veda transizioni multiple di ingresso (che mi garantisce che la rete vedrà prima la variazione di d rispetto a quella di c?).

31.2 Reti trasparenti

Quando il D-Latch è in trasparenza si dice che l'ingresso è *direttamente connesso* all'uscita. Questa connessione è di senso esclusivamente logico (anche perchè ci saranno delle porte tra l'ingresso e l'uscita): pongo in ingresso un valore e quello sarà in uscita.

Problema Colleghiamo q e d in retroazione negativa (cioè collego l'uscita q all'ingresso d ponendo una porta NOT): individuiamo che se $c = 1$ il valore dell'uscita balla mentre con $c = 0$ il valore si stabilizzerà su un valore casuale. Emergono problemi di pilotaggio.

Definizione Quanto segue ha conseguenze su tutti gli argomenti del corso

Il D-Latch è una rete trasparente, la cui uscita cambia mentre la rete è sensibile alle variazioni di ingresso

In pratica non posso porre in ingresso di una rete qualcosa che dipenda dalla sua uscita, anche indirettamente! La cosa è problematica perchè collegamenti di questo tipo sono frequenti e non possono essere evitati. Le reti viste fino ad ora (reti combinatorie, latch SR, D-Latch) sono tutte trasparenti.

Istruzione Assembler Prendiamo ad esempio la seguente istruzione Assembler

```
INC %AX
```

Prendo l'uscita di un qualche elemento di memoria, la faccio passare in una rete combinatoria e la ripongo nel posto da cui l'ho presa. La relazione tra il vecchio valore e quello nuovo è combinatoria!

Soluzione Dobbiamo progettare **reti non trasparenti**.

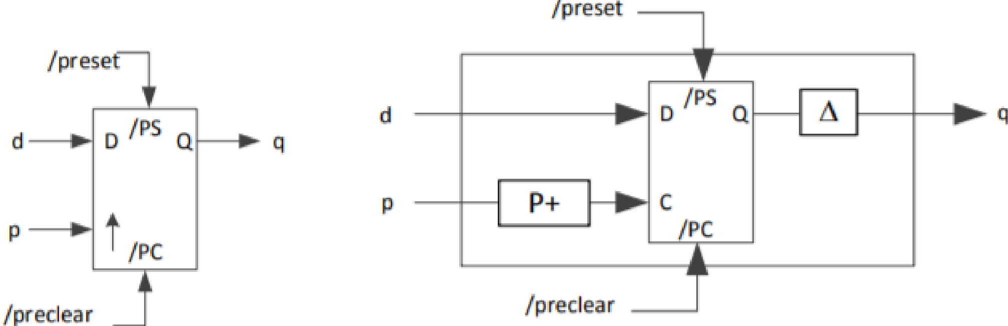
31.3 D flip-flop

Terminologia Si distinguono due reti

- le reti Latch, trasparenti
- le reti flip-flop, non trasparenti

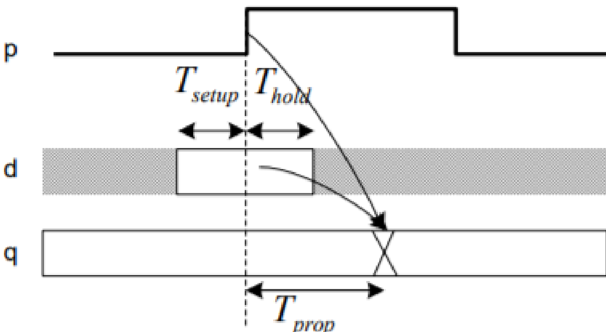
Rete flip-flop Una rete molto comune, quella che vedremo, è la *positive edge-triggered D flip-flop*: con *positive edge* intendiamo il fronte di salita di una certa variabile logica. Ho in ingresso due variabili: d e p. La rete si comporta così: *quando p ha un fronte in salita memorizzo d, attendo un po' (Δ) e adeguo l'uscita.*

Schema concettuale Quanto visto non consiste nella rete vera e propria ma in uno schema concettuale semplificativo.



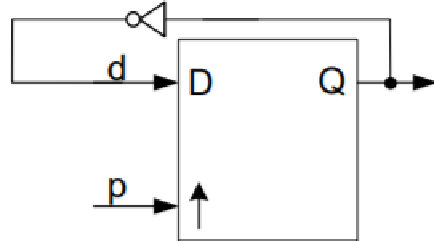
- Il nucleo della rete è il D-latch.
- Abbiamo un formatore di impulsi P^+ che riceve in ingresso p. A un fronte di salita corrisponde, con un ritardo a piacere, un impulso di durata nota. Il risultato di P^+ viene posto in C.
- Il valore d va nel D-latch e viene memorizzato (dopo aver posto $C = 1$, ovviamente)
- Si ritarda l'uscita di un ritardo Δ maggiore dell'intervallo del P^+. Questa cosa, vitale, fa si che la rete sia in conservazione dopo l'adeguamento di q a d. La rete non è più sensibile alle variazioni di ingresso evitando problemi di pilotaggio.

Regole di pilotaggio

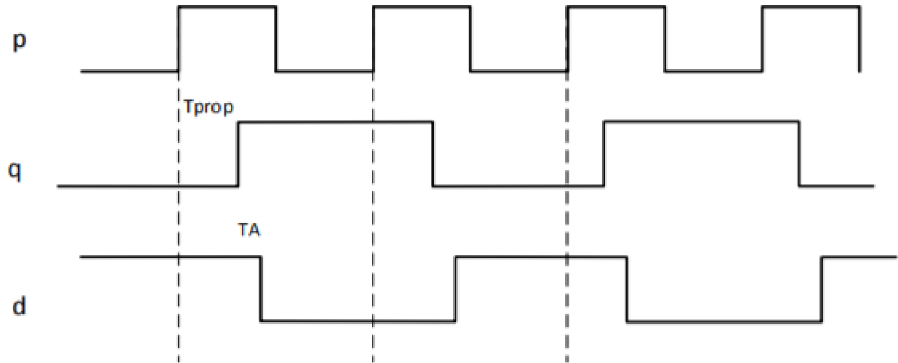


- A cavallo del fronte di salita di p la variabile d deve rimsanere costante. I tempi sono T_{setup} e T_{hold}. I nomi sono gli stessi visti prima ma rappresentano un qualcosa di diverso
- Tra due transizioni in salita della variabile p deve passare abbastanza tempo perchè l'uscita si adegui (solita regola già ribadita in passato). Il ritardo con cui la rete si adegua all'uscita (Δ) è detto T_{prop}. Con $T_{\text{prop}} > T_{\text{hold}}$ garantisco la non trasparenza della rete.

Diagramma di temporizzazione Prendiamo la seguente rete



e realizziamo un diagramma di temporizzazione.

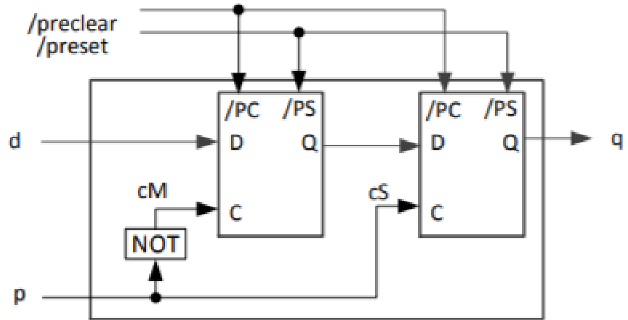


- La rete presenta q e d collegati in retroazione negativa. Il problema avuto col D-latch non si avrà in questo caso.

- q inizialmente è uguale a 0, stessa cosa p. d è invece uguale ad 1.
- Con $p = 0$ la rete è in conservazione: pur avendo $d = 1$ in ingresso continuerò a tenere $q = 0$.
- Ho un fronte di salita: p viene posto uguale ad 1.
- q è costante attorno al fronte di salita (T_{setup} e T_{hold})
- Osservo che si ha un certo margine tra il cambio di p e l'adeguamento di q (T_{prop})
- Poco dopo l'adeguamento di q si ha la variazione di d dovuta alla porta NOT. Il tempo di risposta della porta NOT fa si che il valore di d aggiornato arriverà alla rete quando questa si sarà nuovamente posta in conservazione.

Sintetizzazione con struttura master/slave Una rete D flip-flop.può essere sintetizzare attraverso due D-latch in cascata. Il primo è detto *master* e il secondo *slave*.

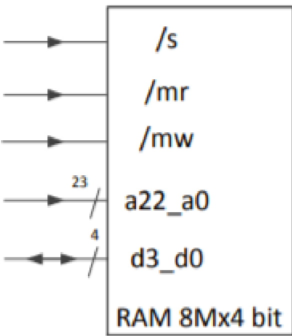
- Q in uscita del *master* va in ingresso nello *slave*.
- p del flip-flop va in ingresso nei C dei D-latch: prima dell'ingresso C del *master* si pone una porta NOT.



- L'idea è che questi due D-latch non si comporteranno mai allo stesso modo: uno è in conservazione, l'altro in trasparenza.
  - Se $p = 0$ il *master* campiona mentre lo *slave* conserva (non ascolta d)
  - Se $p = 1$ il *master* conserva mentre lo *slave* campiona (quindi insegue l'uscita del *master*)
- Potrebbe emergere un problema in situazioni transitorie: i due D-latch potrebbero essere entrambi in trasparenza. La cosa si risolve per via elettronica: si fa in modo che l'ingresso c dello *slave* riconosca come 0 un maggior range di tensioni rispetto al c del *master*.

31.4 Memoria RAM statiche

Le memoria RAM statiche, dette S-RAM (ci sono anche memoria RAM dinamiche, fatte in modo diverso). La caratteristica principale è la presenza di D-Latch montati a matrice. Una riga di D-Latch costituisce una locazione: il numero di D-latch per riga consiste nel numero di bit per locazione.

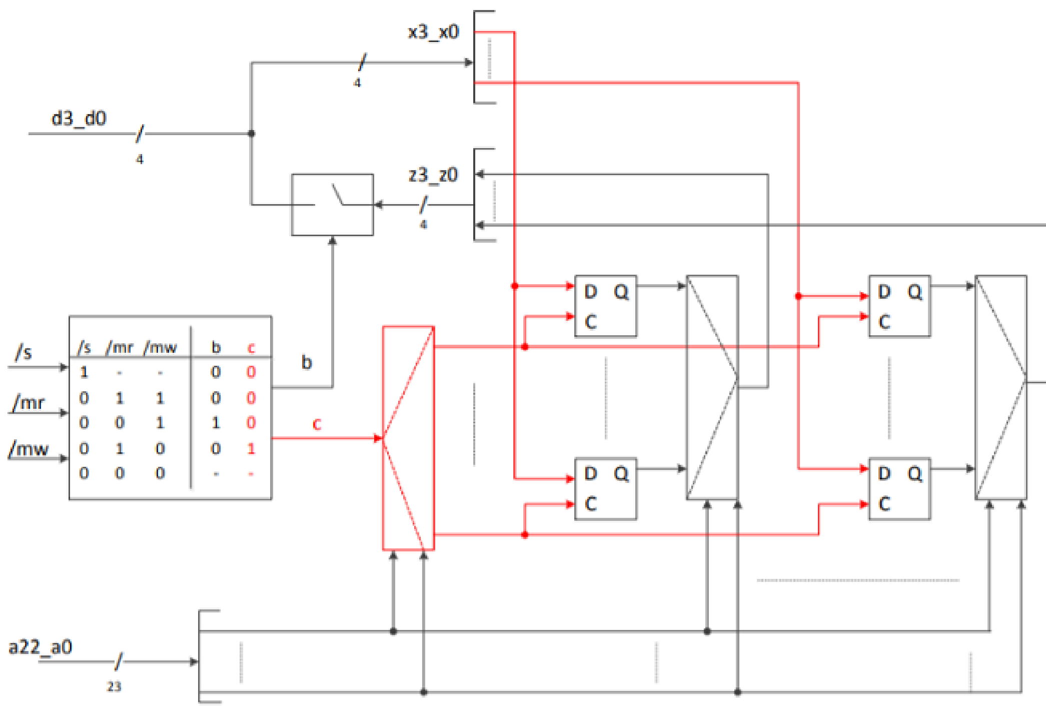


Ingressi La rete presenta i seguenti ingressi. Teniamo conto del numero di celle N e del numero di bit K.

- Un certo numero di fili di indirizzo, cioè una serie di ingressi che permettano di porre l'indirizzo di una cella di memoria. Per poter rappresentare tutti gli indirizzi porrò N fili di indirizzo (quindi potrò indicare 2^N indirizzi possibili, cioè $2^N = k$ dove k è il numero di celle presenti nella memoria RAM)

- Un certo numero di fili di dati (precisamente K fili, in modo tale da poter rappresentare tutti i bit della cella), che fungono sia da ingresso che da uscite (dipende dall'operazione che stiamo svolgendo, scrittura o lettura). Saranno gestiti mediante porte tristate.
- I seguenti comandi:
  - $/s$, attivo basso con cui indichiamo la selezione della memoria. Con $/s = 1$ la memoria è insensibile agli ingressi. La sua funzione è paragonabile a quella della variabile e in un decoder espandibile.
  - $/mr$, attivo basso con cui indichiamo comando di scrittura.
  - $/mw$, attivo basso con cui indichiamo comando di lettura

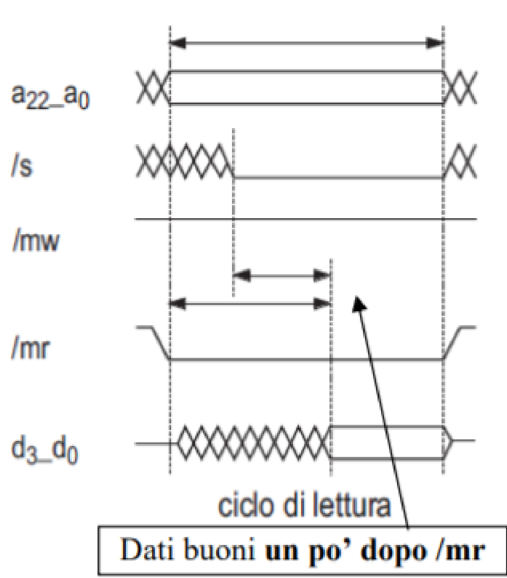
  ovviamente non potrò avere $/mr = /mw = 0$, non ha senso.



- Le porte tristate dipendono dalla variabile b: queste saranno abilitate solo con $b = 1$, cioè quando svolgiamo operazioni di lettura.
- I fili di indirizzo consistono in variabili di comando:
  - Nelle operazioni di lettura svolgono il ruolo di variabili di comando per una serie di multiplexer. Abbiamo un multiplexer per colonna: attraverso le variabili di comando indichiamo quale bit ci interessa leggere. L'unione dei bit indicati dai vari multiplexer costituiscono una riga, cioè il numero di bit che costituiscono una cella. I bit in uscita vanno alle porte tristate.
  - Nelle operazioni di scrittura svolgono il ruolo di variabili di comando per un unico demultiplexer: ho in ingresso c, che sarà uguale ad 1 quando voglio svolgere operazioni di scrittura. Con le variabili di comando pongo $c = 1$ solo nella riga dove voglio

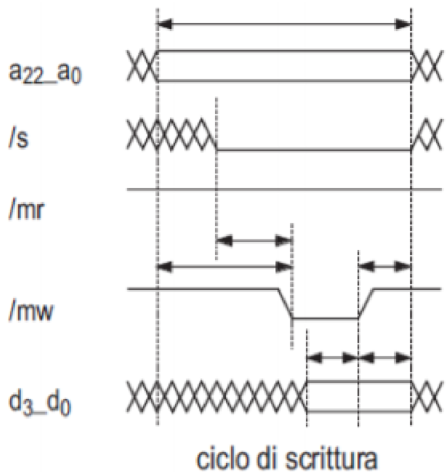
scrivere. I valori influiscono sulla variabile C dei vari D-latch. I D-latch di tutte le altre righe avranno $C = 0$ (quindi stanno in conservazione).

Temporizzazione - lettura



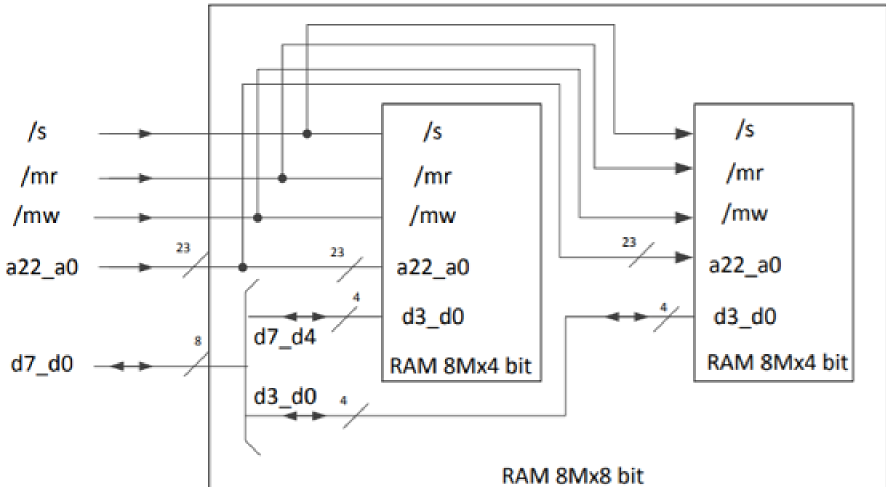
- Si stabilizzano gli indirizzi
- Si pone subito il comando di lettura quindi $/mr = 0$
- Il comando di select $/s$ si assesta con un po' di ritardo (il select può dipendere da una rete combinatoria detta *maschera*)
- Ovviamente $/mw = 1$ fisso
- Dopo un po' di tempo le tristate vanno in conduzione e i multiplexer vanno a regime.
- A quel punto i dati presenti nei fili di ingresso/uscita sono corretti e possono essere prelevati
- Ponendo $/mr = 0$ i dati tornano subito in alta impedenza.

Temporizzazione - scrittura



- La temporizzazione delle operazioni di scrittura non potrà essere simmetrica a quella delle operazioni di lettura: la scrittura è un'operazione distruttiva che influisce sui D-latch (che vanno in trasparenza).
- Si stabilizzano gli indirizzi
- Ovviamente $/mr = 1$ fisso
- Il comando di select $/s$ si assesta con un po' di ritardo (il select dipende da altri fili di indirizzo)
- Il comando $/mw$ non può andare subito a zero: devo attendere che si stabilizzino il comando select e gli indirizzi.
- I dati in ingresso possono ballare a piacimento anche con $/mw = 0$: i dati devono essere buoni a cavallo del fronte di salite di $/mw$ (un po' prima e un po' dopo il fronte di salita). Tale fronte corrisponde al fronte di discesa di c sui D-latch.

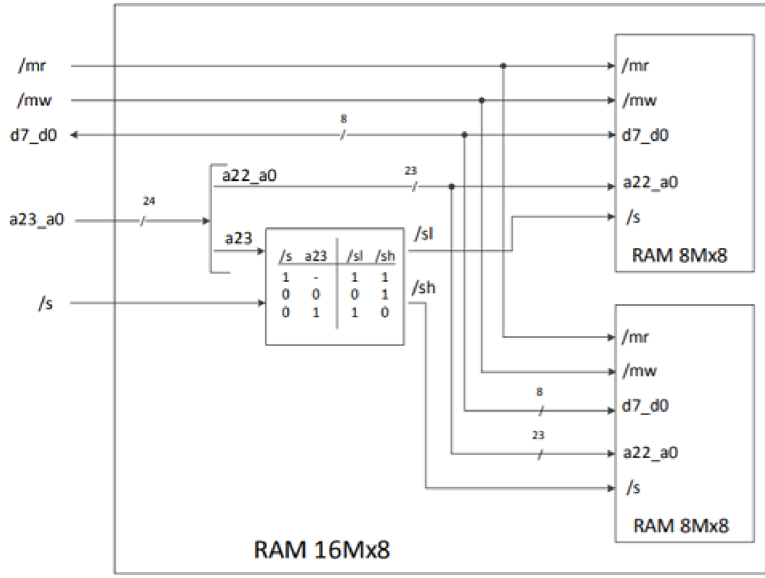
31.5 Montaggio in parallelo di memorie RAM statiche



Avevamo già intuito che il comando $/s$ permette di costruire memorie grandi a partire da memorie piccole.

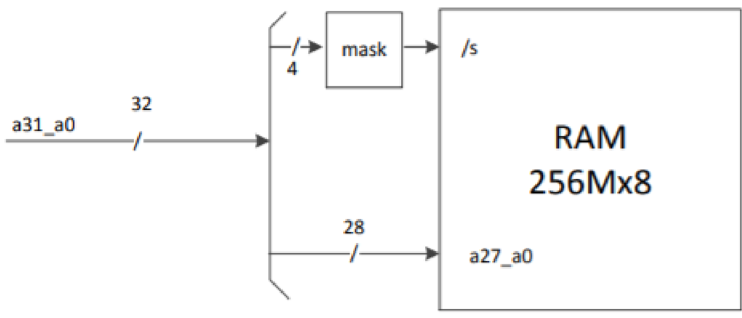
- Vogliamo costruire una memoria 8Mx8 usando banchi 8Mx4. Disporre le memorie in questo modo permette di raddoppiare la capacità delle celle (per ogni cella ho 4 bit in un banco e 4 bit in un altro).
- Connetto tutti i fili tranne i dati in parallelo (quindi al di là dei dati gli ingressi rimanenti sono uguali in tutti e due i banchi)
- Relativamente ai fili dei dati affastello i due gruppi con un montaggio che non richiede logica particolare.

31.6 Montaggio in serie di memorie RAM statiche



- Montare in serie significa aumentare il numero di celle/locazioni disponibili.
- Supponiamo di voler ottenere una memoria 16Mx8 usando banchi 8Mx8.
- Per indirizzare 16M ho bisogno di 24 fili di indirizzo: i moduli RAM 8Mx8 hanno ciascuno in ingresso 23 fili di indirizzo. A cosa serve il 24esimo a_{23}? Permette di distinguere in quale blocco si trovano i dati che voglio estrarre o modificare. Precisamente:
  - se $a_{23} = 1$ andrò nella parte alta (che consiste in un modulo)
  - se $a_{23} = 0$ andrò nella parte bassa (che consiste nell'altro modulo).
- $/mw$ e $/mr$ si collegano in parallelo ad entrambi i moduli (valori che saranno ignorati da uno dei due moduli in base al valore del suo $/s$). Stessa cosa per i fili di dati.
- Dei 24 fili di indirizzo i 23 più bassi vanno in parallelo ad entrambi i moduli. Il filo più significativo a_{23} va in ingresso ad una **maschera**.
- La maschera ha in ingresso anche il comando $/s$ e produce in uscita due variabili: select low ($/sl$) e select high ($/sh$). Dalla tavola di verità individuiamo che
  - Con $/s = 1$ avrò $/sl = /sr = 1$. Non ci interessa lavorare sui moduli
  - Con $/s = 0$ significa che ho selezionato la memoria RAM. Per capire quale modulo della memoria ci interessi prendiamo il filo più significativo a_{23}. In base al valore di a_{23} deciderò quale delle due uscite avrà valore 1 e quale 0. Ovviamente non posso avere $/sl = /sh = 0$

31.7 Collegamento al bus e maschere



a31	a30	a29	a28	/s
1	1	1	0	0
others				1

Da dove provengono i fili di indirizzo? Da un bus di indirizzi (pensare alla struttura del calcolatore di Von Neumann)

- Supponiamo di avere un bus indirizzi a 32 bit, che permette di indirizzare $2^2 \cdot 2^{30} = 4GB$ di memoria.
- Implementiamo la memoria RAM con un modulo di RAM 256Mx8 bit (28 fili di indirizzo, $2^8 \cdot 2^{20}$)
- Il modulo avrà un filo di select $/s$ e dovrà rispondere agli indirizzi da $0xE0000000$ a $0xEFFFFFFF$

- Dal BUS arrivano 32 fili di indirizzo
  - I 28 meno significativi forniscono un indirizzo $a_{27_}a_0$
  - I rimanenti 4 passano per una maschera che restituisce il valore che avrà $/s$
- Sappiamo che $(E)_{16} = (1110)_2$. Tutti gli indirizzi presentano quelle cifre, quindi mi basta averne almeno una diversa per dire di non selezionare la memoria RAM. Quindi:
  - se $a_{31} = 1, a_{30} = 1, a_{29} = 1, a_{28} = 0$ avrò $/s = 0$
  - se una sola delle uguaglianze dette prima non è uguale avrò $/s = 1$

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

Attenzione

$/s$ non sta sul bus, ma è prodotto da alcuni fili del bus. METTERSELO NEL CAPO.

Chi progetta la maschera? Colui che assembla il sistema e decide che un certo chip di memoria coincide con un certo intervallo di indirizzi.

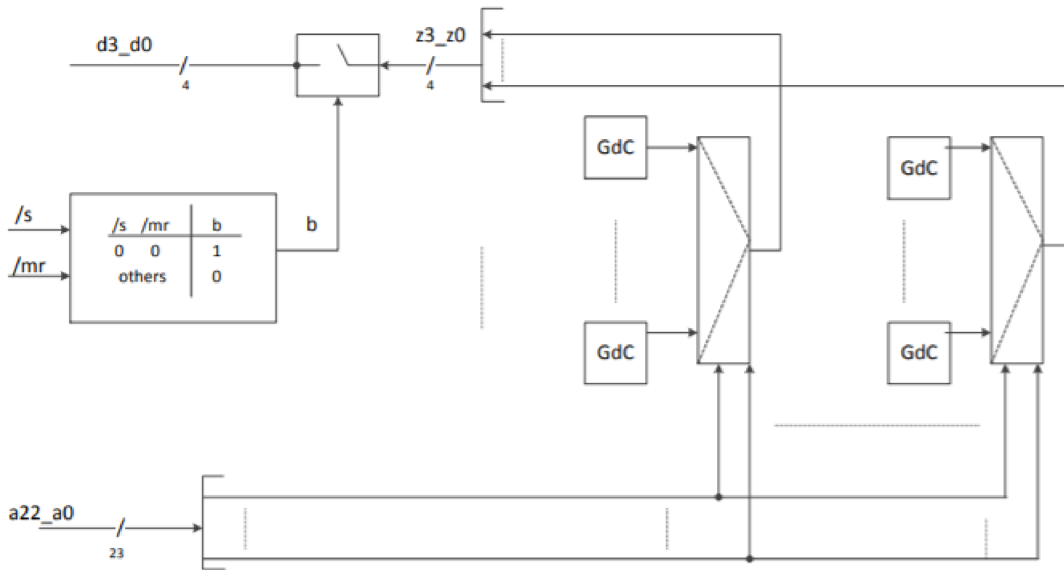
Capitolo 32

Giovedì 05/11/2020

32.1 Memorie ROM (*read-only*)

Le memorie ROM sono reti combinatorie: le celle contengono valori costanti (al posto dei D-latch ho dei generatori di costanti), lo stato di uscita dipende esclusivamente dallo stato di ingresso. Consistono nella parte non volatile dello spazio di memoria (cioè quella che mantiene informazione in assenza di tensione).

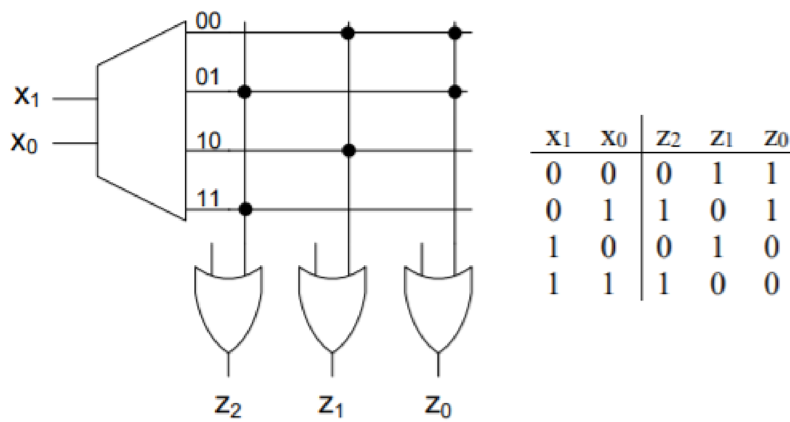
Descrizione Lo schema concettuale si ottiene semplificando quanto visto per le memorie RAM: si rimuovono tutte le componenti dedite alla scrittura e i D-latch. Rimangono le porte tri-state (abbiamo i fili di dati che possono restituire o accettare dati, esattamente come nelle RAM).



Programmazione della memoria La programmazione della memoria non può avvenire durante il funzionamento, altrimenti sarebbe una memoria RAM.

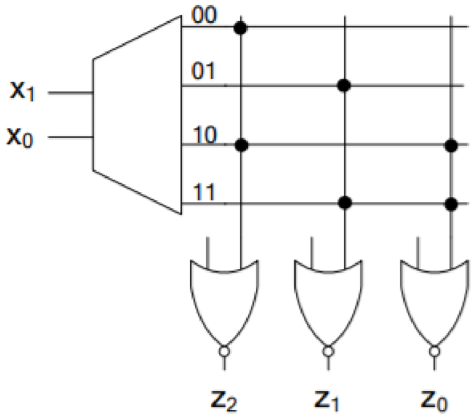
Sintesi

- Escludendo la parte relativa alle porte tri-state, il resto equivale a una rete combinatoria con N ingressi ed M uscite.
- La nostra memoria sarà caratterizzata da 2^N locazioni, ciascuna con M bit (le uscite)
- Parliamo di reti combinatorie: possiamo adottare un modello strutturale universale con un decoder N to 2^N e le uscite collegate attraverso una barriera ad M porte OR (ciascuna per uscita). Ovviamente i collegamenti fatti dipendono dai valori che vogliamo porre in memoria ROM:
  - Connetto l'uscita del decoder a una certa porta OR quando la cella che corrisponde a quella porta ha il corrispondente bit a 1.
  - Non connetto se il bit è a 0



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

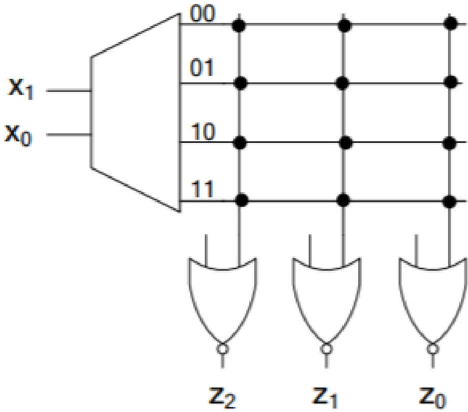
- In alcuni casi avremo delle porte NOR al posto delle porte OR. L'approccio con cui stabiliamo i collegamenti cambia:
  - Connetto l'uscita del decoder a una certa porta NOR quando la cella che corrisponde a quella porta ha il corrispondente bit a 0.
  - Non connetto se il bit è a 1



32.1.1 ROM programmabili

La memoria ROM è realizzata su un singolo chip di silicio. Le singole componenti non sono di per se costose: il problema è il supporto che le collega! Ricordiamo che queste memorie dovrebbero uscire dalla fabbrica già programmate. Vale la pena spendere così tanto se vogliamo realizzare circuiti con migliaia di pezzi. In tutti gli altri conviene adottare un'alternativa.

32.1.1.1 PROM



Immaginiamoci la solita struttura vista prima (modello strutturale universale con porte NOR al posto delel porte OR), ma con tutti i contatti possibili tra le uscite degli AND e gli ingressi dei NOR. Questo significa che il contenuto di ogni cella è zero.

- Il programmatore comprerà una memoria la cui matrice di connessione è fatta da fusibili.
- Il programmatore brucerà certi fusibili ponendo certi bit uguali ad 1.

La programmazione è di tipo distruttutiva: può essere fatta una volta soltanto (eventuali errori non possono essere corretti).

32.1.1.2 EPROM

EPROM è acronimo di *Erasable Programmable ROM*. Le connessioni sono fatte non con fusibili, ma con dispositivi elettronici programmabili per via elettrica e cancellabili tramite esposizione a raggi ultravioletti. La EPROM può essere riprogrammata più volte: viene scollegata dal circuito e collegata a un programmatore di EPROM sfruttando un foro (solitamente coperto da etichetta). La scarica di una EPROM non è selettiva: se si interviene si cancella tutto il contenuto.

- La *endurance*, cioè la capacità di sopportare riprogrammazioni, sta nell'ordine di 10K-100K volte.
- La *data retention*, cioè il periodo di affidabilità del contenuto della ROM, sta nell'ordine di 10-100 anni (se si vuole maggiore affidabilità conviene spendere di più).

32.1.1.3 EEPROM

EEPROM è acronimo di *Electrically Erasable Programmable ROM*. Sono programmate e cancellate sfruttando segnali elettrici appositi (diversi da quelli presenti normalmente quando la rete è a regime). Possono essere riprogrammate direttamente on chip, contrariamente alle EPROM.

Esempio di EEPROM Il bios, le impostazioni sono salvate in un dispositivo che mantiene i dati in assenza di tensione e ne permette la modifica.

Fermi Ma a questo punto ha senso parlare di memoria ROM? La EEPROM è un dispositivo programmabile.

- Il numero di volte in cui si può riprogrammare è limitato. Noi non ci accorgiamo di questo perchè le ROM sono utilizzate per salvare dati stabili.
- Le memorie continuano ad essere non volatili.
- Il tempo richiesto per scrivere sulla EPROM sta nell'ordine dei ms (teniamo conto che la lettura sta nei ns, e che $1ms = 10^6ns$).
- La tensione usata è diversa (più alta)

Verilog

Abbiamo detto che una rete può essere descritta in vari modi:
- A parole
- Con tavole di verità
- Attraverso linguaggi di descrizione.

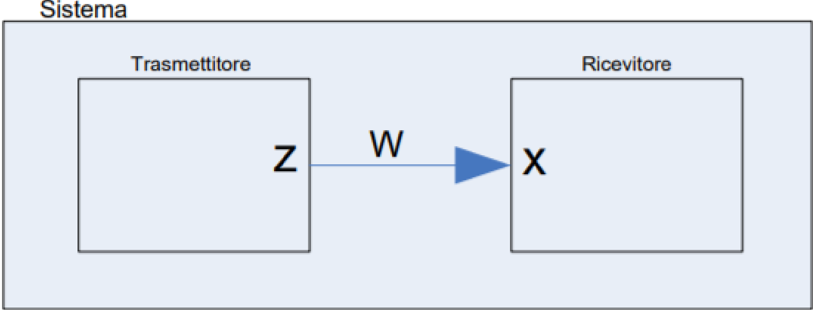
Il linguaggio che utilizzeremo noi è il **Verilog**, che permette di esprimere la descrizione o la sintesi di una rete
- in modo compatto, e
- interpretabile in modo automatico da una macchina.

Un linguaggio di descrizione è utile soprattutto con reti complesse il cui disegno si estenderebbe su una coperta matrimoniale.

Osservazione: di Verilog esistono numerose versioni. È caldamente sconsigliato cercare tutorial su internet. La notazione adottata è funzionale ai compiti che dobbiamo svolgere: non introdurremo in modo formale i costrutti e le funzionalità (come fatto a FdP con C++) ma ci limiteremo a introdurre solo le cose necessarie.

Esempio

Recuperiamo l'esempio delle due scatolette visto all'inizio del corso

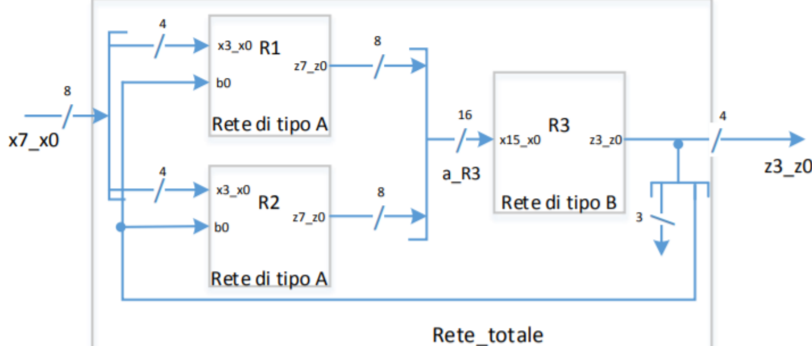


- Abbiamo un Sistema che contiene una rete Trasmettitore e una rete Ricevitore
- Descriviamo con **linguaggio Verilog**:

```
module Sistema;
    wire w;
    Trasmettitore T(w); // Alternativa: Trasmettitore T(.z(w))
    Ricevitore R(w); // Alternativa: Ricevitore R(.x(w))
endmodule

module Trasmettitore(z);   // Rete di tipo Trasmettitore
    output z;
    // descrizione della struttura interna
endmodule

module Ricevitore(x); // Rete di tipo Ricevitore
    input x;
    // descrizione della struttura interna
Endmodule
```

- Le scatolette corrispondono a moduli: i moduli hanno un nome (Verilog è case sensitive), presentano dei valori in ingresso e dei valori in uscita.
- Nella lista dei parametri poste tra parentesi si pongono sia gli ingressi che le uscite: sta a noi indicare con istruzioni all'interno del modulo quali sono i valori in ingresso (keyword *input*) e quali le uscite (keyword *output*).
- Per rappresentare il collegamento tra la prima e la seconda scatoletta serve un filo: questo filo, che nell'esempio è *w*, è introdotto nel modulo *Sistema* con la keyword *wire*.
- Nelle alternative all'interno del modulo *Sistema* si esprime in modo esplicito il collegamento della variabile di uscita o di ingresso al filo *w*.

Ulteriore esempio



- In questo esempio abbiamo una rete *Rete_totale* caratterizzata da tre sottoreti: due *Reti di tipo A* e una *Rete di tipo B*.
- Si noti che gli ingressi e le uscite non consistono in semplici variabile logiche.

- Sottoreti:

```
module Rete_di_tipo_A(x3_x_0, b0, z7_z0);
    input [3:0] x3_x0;
    input b0;
    output [7:0] z7_z_0;
    // descrizione della rete di tipo A
endmodule

module Rete_di_tipo_B(x15_x0, z3_z0);
    input [14:0] x15_x0;
    output [3:0] z3_z0;
    // descrizione della rete di tipo B
endmodule
```

  - Attraverso la notazione utilizzata per dichiarare le variabili a più bit specifichiamo prima l'indice più alto e poi l'indice più basso (come nell'array in C++).
  - Questo tipo di notazione permette di considerare, in certi casi, solo alcuni bit della variabile.
  - Se non si pone questa notazione si da per scontato che stiamo parlando di una variabile logica (quindi un solo bit)

- Rete principale:

```
module Rete_totale(x7_x0, z3_z0);
    input [7:0] x7_x0;
    output [3:0] z3_z0;
    wire [15:0] a_R3;

    Rete_di_tipo_A R1(x7_x0[7:4], z3_z0[0], a_R3[15:8]),
                   R2(x7_x0[3:0], z3_z0[0], a_R3[7:0]);
    Rete_di_tipo_B R3(a_R3[15:0], z3_z0[3:0]);
endmodule
```

  - Ho in ingresso una variabile a 8 bit: di questi i quattro più significativi vanno nella rete R1, quelli meno significativi nella rete R2.
  - La variabile b0 viene recuperata dall'output della rete R3: di quell'output si prende la cifra meno significativa e la si riporta indietro, in ingresso nelle reti R1 e R2.
  - L'output della rete R3 è anche l'output della *Rete_totale*.
  - Inoltre:
    - Le reti R1 e R2 restituiscono valori di 8 bit.
    - Questi valori vengono affastellati per formare a_R3, variabile a 16 bit.
    - a_R3 va in ingresso nella rete R3.

- Non è necessario indicare gli indici se vogliamo porre in ingresso o in uscita una variabile nella sua interezza. Non è sbagliato farlo, ma è il metodo migliore per generare confusione (si instilla l'idea che non stiamo considerando tutti i bit della variabile).

Osservazioni sulla sintassi

- Le keyword si pongono solitamente in minuscolo.
- Verilog, lo ribadiamo, è un linguaggio *case sensitive*.
- I nomi degli identificatori presentano le stesse regole del C++ (caratteri, numeri e/o underscore, ma non posso iniziare con un numero).

- **Costanti**: le costanti possono essere introdotte seguendo il seguente costrutto

```
n'{B,D,H}valore
```

  - *n* consiste nel numero di bit su cui va intesa la costante. Può essere omesso quando il numero di bit desiderato è deducibile dagli altri dati.
  - *{B, D, H}* consiste nella base in cui dobbiamo interpretare la costante
    - **B**: base binaria
    - **D**: base decimale
    - **H**: base esadecimale
  - *valore*

  - **Attenzione**: di default si interpreta in base 10.
    Bisogna stare attenti se si omette il secondo elemento.

  - **Esempio**: 8'D32 significa scrivere il numero $(32)_{10}$ su 8 bit.
  - **Esempio 2**: scrivere 0101 significa scrivere $(101)_{10}$ su 4 bit.

- Ciascun costrutto va terminato con punto e virgola.

Valori assegnabili a una variabile in Verilog

- **Uno**: 1'B1 che può essere scritto semplicemente come 1
- **Zero**: 1'B0 che può essere scritto semplicemente come 0

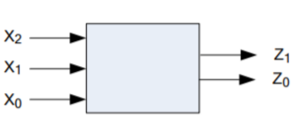
- Solo in uscita:
  - **Alta impedenza**: 1'BZ
  - **Non specificato**: 1'BX

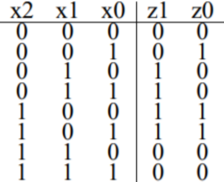
Commenti

I commenti si scrivono nello stesso modo visto in C++

Descrizioni di reti combinatorie in Verilog

- La descrizione di una rete combinatoria si ottiene a partire da una tavola di verità.
- Prendiamo la seguente rete





x2	x1	x0	z1	z0
0	0	0	0	0
0	0	1	0	1
0	1	0	1	0
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	0	0
1	1	1	0	0

  - Abbiamo tre variabili logiche di ingresso e due variabili logiche di uscita.
  - L'unica cosa nuova da introdurre è come esprimere i valori di *z1* e *z0* tenendo conto della tavola di verità. Effettueremo il cosiddetto **assegnamento continuo**: in ogni istante temporale ciò che sta a sinistra viene adeguato a ciò che sta a destra (una rete asincrona praticamente).

```
module RC(x2, x1, x0, z1, 0);
    input x2, x1, x0;
    output z1, z0;

    assign #T {z1, z0} = ({x2,x1,x0}=='B000) ? 'B00:
```

```
                              ({x2,x1,x0}=='B001) ? 'B01:
                              ({x2,x1,x0}=='B010) ? 'B10:
                              ({x2,x1,x0}=='B011) ? 'B10:
                              ({x2,x1,x0}=='B100) ? 'B11:
                              ({x2,x1,x0}=='B101) ? 'B11:
                              ({x2,x1,x0}=='B110) ? 'B00:
                              ({x2,x1,x0}=='B111) ? 'B00;
endmodule;
```

- Si utilizza un costrutto simile a quello dell'if aritmetico visto in C++

```
(condition) ? expression1 : expression2;
```

- L'assegnamento può essere scritto anche in altri modi:
  - Indico un valore default tenendo conto che ho tre stati di ingresso associati allo stesso stato di uscita (B00). Elimino le righe con questo stato di uscita e pongo alla fine

    ```
    /* default */ 'B00;
    ```

    (commento per rendere chiaro in cosa consiste quel valore, ripensare all'operatore del C++, nell'assegnamento continuo è come se applicassi quell'operatore un certo numero di volte.

    ```
    (condition1) ? expression1 : (condition2) ? expression2 : …;
    ```

  - **Utilizzo le *functions***: una function rappresenta un'espressione combinatoria.

```
module RC(x2, x1, x0, z1, 0);
    input x2, x1, x0;
    output z1, z0;

    assign #T {z1, z0} = F(x2, x1, x0);

    function [1:0] F;
        input x2, x1, x0;
        casex({x2,x1,x0})
          'B000 : F='B00;
          'B001 : F='B01;
          'B010 : F='B10;
          'B011 : F='B10;
          'B100 : F='B11;
          'B101 : F='B11;
          'B110 : F='B00;
          'B111 : F='B00;
        endcase
    endfunction
endmodule;
```

Possiamo semplificare la lista degli elementi in *casex*:

- Prendo il valore più assegnato in uscita, elimino le righe con questo valore e pongo un valore di default in fondo (questa volta default è una keyword, non un commento)

  ```
  'B001 : F='B01;
  'B011 : F='B10;
  'B010 : F='B10;
  'B100 : F='B11;
  'B101 : F='B11;
  default : F='B00;
  ```

- Alcuni valori presentano un bit in comune: sostituisco quattro righe con due righe dove utilizzo il ? per indicare un valore irrilevante relativamente a un certo bit

```
'B001 : F='B01;
'B01? : F='B10;
'B10? : F='B11;
default : F='B00;
```

Sintesi di reti combinatorie in Verilog

- **Chiarimento iniziale**: Verilog può essere utilizzato per esprimere la descrizione, ma anche per esprimere la sintesi di una rete combinatoria. È ovvio che le due cose sono distinte e non si possono fare in contemporanea: tenerle distinte nella testa per evitare confusione durante l'esame.

- La sintesi si scrive effettuando assegnamenti continui come prima: al posto di quella serie di controlli appena vista si scrivono espressioni logiche.
- Le espressioni logiche possono contenere gli operatori AND, OR, NOT, XOR. Ciascun operatore è identificato dai seguenti simboli:
    - `Cond1 AND Cond2 => Cond1 && Cond2`
    - `Cond1 OR Cond2 => Cond1 || Cond2`
    - `NOT Cond1 => ~Cond1`
    - `Cond1 XOR Cond2 => Cond1 ^ Cond2`

- Supponiamo di avere la seguente sintesi a costo minimo:
    - $z_1 = \overline{x_2} \cdot x_1 + x_2 \cdot \overline{x_1}$.
    - $z_0 = x_2 \cdot \overline{x_1} + x_0 \cdot \overline{x_1}$

  Otteniamo

```
module RC(z1, z0, x2, x1, x0);
    input x2, x1, x0;
    output z1, z0;
    assign #T z1=(~x2 & x1)|(x2 & ~x1);
    assign #T z0=(x2 & ~x1)|(x0 & ~x1);
endmodule
```

Reti sequenziale sincronizzate

Ricapitoliamo

Abbiamo visto fino ad ora le seguenti reti:

- **Reti combinatorie**, lo stato di uscita dipende esclusivamente dallo stato di ingresso. Equivalenti a funzioni matematiche, si dicono **reti senza memoria**.
- **Reti sequenziali asincrone**, lo stato di uscita non dipende esclusivamente dallo stato di ingresso (viene meno quella corrispondenza univoca tra ingresso e uscita tipica delle reti combinatorie), ma dallo storico degli ingressi. Si parla, non a caso, di **reti con memoria**.

Entrambe le reti sono **asincrone**, cioè l'aggiornamento delle uscite avviene continuamente nel tempo.

Introduciamo il nuovo tipo di rete

Introduciamo le cosiddette reti sequenziali sincronizzate.

- Sono reti con memoria esattamente come le RSA
- Contrariamente alle reti viste fino ad ora le RSS si evolvono soltanto in corrispondenza di istanti temporali ben precisi detti istanti di sincronizzazione.

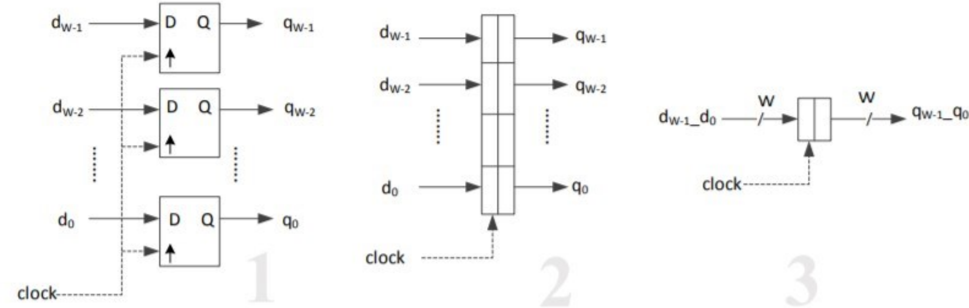
Cioè la rete si evolve con l'arrivo di un **segnale di sincronizzazione**: questo segnale viene portato dal **clock** ("oscillatore che genera impulsi elettrici ad alta frequenza con periodo fisso").

- Il clock scandisce gli istanti di sincronizzazione della rete.
- Consiste in un'onda periodica di periodo T e frequenza $1/T$
- Il *duty-cycle*, cioè la percentuale di tempo del periodo in cui si ha il valore espresso dal clock uguale ad 1, è normalmente del 50%. Nulla vieta di avere una percentuale diversa: questa, tuttavia, non deve essere né troppo piccola (5%) né troppo grande (95%)

- **Osservazione**: l'evento che sincronizza la rete (che riceve il segnale di sincronizzazione) è il fronte di salita del clock.

Registri (elemento base per la sintesi di Reti Sequenziali Sincronizzate)

- **Fermi tutti**: l'osservazione dovrebbe drizzarci le antenne.



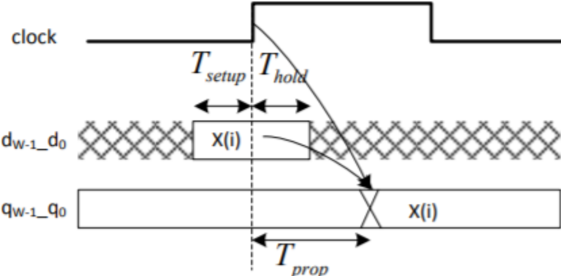
- Un registro a K bit consiste in una collezione di k *D-Flip-flop positive-edge-triggered*. Ciascuno di questi presenta:
    - Un ingresso d_i
    - Un'uscita q_i
    - Un ingresso p comune
- Sappiamo che i D-flip-flop consistono in reti sequenziali asincrone. Sappiamo che l'aggiornamento del contenuto del flip-flop è possibile ponendo l'ingresso $\mathrm{p} = 1$. Se consideriamo p come una variabile speciale, appunto quella che riceve in ingresso il segnale di sincronizzazione, allora possiamo considerare il tutto come una rete sequenziale sincronizzata.
- A questo punto p non indica un valore di ingresso, quindi smettono di considerarlo uno degli ingressi: d'ora in poi dirò che il registro a k bit presenta k ingressi e k uscite.
- Non è l'unico ingresso che diamo per scontato: abbiamo anche q_N,$/preset$,$/preclear$

- I k bit sono detti **capacità del registro**

- Lo stato di uscita consiste nel **contenuto del registro**.
- Utilizzare questi valori è **un'operazione di lettura** del registro.
- La memorizzazione di K bit a un certo istante di sincronizzazione è **un'operazione di scrittura** del registro.

- Ovviamente con $/\mathrm{preset}$ e $/\mathrm{preclear}$ adeguatamente collegate possiamo stabilire un valore iniziale per il registro.

- **Regole di pilotaggio**:
  - Gli ingressi d_i devono rimanere costanti a cavallo del fronte di salita del clock per un tempo T_{setup} Prima e un tempo T_{hold} dopo.
  - Ovviamente, parlando di un D-flip-flop, dovrò avere $T_{prop} > T_{hold}$. Ricordiamoci che il valore dello stato di uscita sarà aggiornato con un ritardo T_{prop}



- Tutto ciò che avviene al di fuori degli intervalli citati è irrilevante e non verrà memorizzato. Il fatto che due stati di ingresso presentati su istanti di clock consecutivi siano identici, adiacenti o non adiacenti non ha importanza (cioè, se devo aggiornare solo un bit contenuto nel registro posso farlo senza problemi, la prevedibilità della rete non viene scalfita).

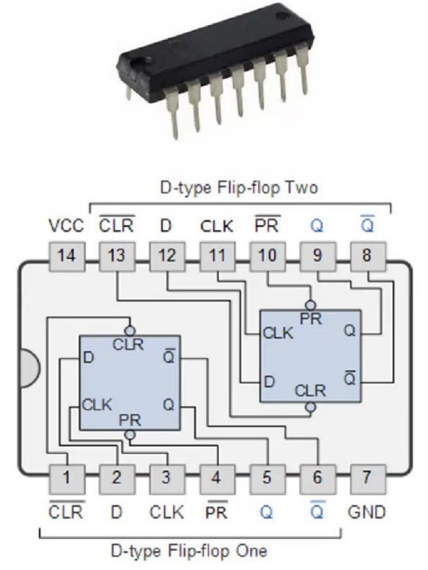
<u>**Esempio di Registro**</u>

- **Prendiamo un esempio di registro**:
  74LS74 DUAL D-TYPE FLIP-FLOP

- Il chip in questione può essere usato come due flip-flop autonomi, ma anche come un registro a due bit (ovviamente l'ingresso di clock dovrà essere collegato agli stessi generati di clock).

- I piedini, forse la cosa più riconoscibile di queste componenti, sono collegati a una rete e permettono lo scambio di valori di ingressi e valori di uscita (oltre a collegare il chip alla tensione e alla massa). Notiamo in particolare:
  - I clock dei singoli registri sui piedini 3 e 11
  - Gli ingressi di preclear sui piedini 13 e 1
  - Gli ingressi di preset sui piedini 1 e 4
  - L'uscita diretta sui piedini 5 e 9
  - L'uscita negata sui piedini 6 e 8

  - Collegamenti a tensione e a massa sui piedini 14 e 7.



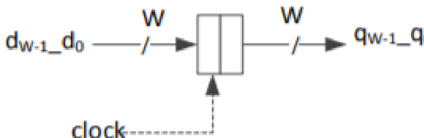
Capitolo 33

Martedì 10/11/2020

Descrizione in Verilog di registri

- Utilizzeremo il linguaggio Verilog per descrivere RSS. Facciamo ciò in parallelo ad altri formalismi già visti.
- Verilog sarà estremamente utile con RSS di notevole complessità: pensiamo a una rete con 50 stati e 20 ingressi. Impensabile disegnare tabelle.

- Descriviamo la seguente rete utilizzando Verilog



// Dichiarazione di un registro da W bit di tipo reg e nome REGISTRO.
// Dichiarazione delle variabili clock e reset_ da usarsi, rispettivamente, come clock e per l'impostazione dello stato interno iniziale.
// Dichiarazione di due variabili a W bit, dW-1_d0 e qW-1_q0 da usarsi, rispettivamente, come variabile di ingresso e come variabile di uscita e impostazione di quest'ultima come effettiva variabile di uscita.

```
reg [W-1:0] REGISTRO;
wire clock,reset_;
wire [W-1:0] dW-1_d0;
wire [W-1:0] qW-1_q0; assign qW-1_q0=REGISTRO;
```

// Immissione nel registro del contenuto_iniziale al reset_ e dello stato
// della variabile dW-1_d0 all'arrivo di ogni segnale di sincronizzazione

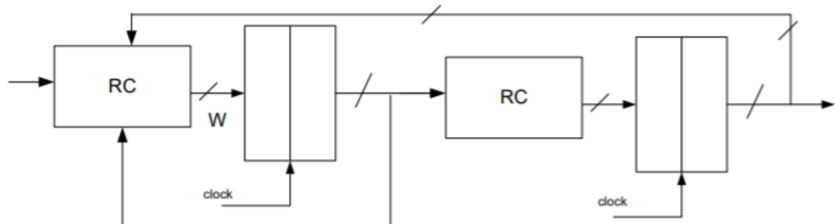
```
always @(reset_==0) #1 REGISTRO<=contenuto_iniziale;
always @(posedge clock) if (reset_==1) #Tpropagation REGISTRO<=dW-1_d0;
```

- Si dichiara un registro (keyword `reg`). Convenzione NOSTRA è scrivere i nomi dei registri in maiuscolo.
- Per motivi del linguaggio le uscite del registro devono essere dichiarate separatamente dal registro.
- Si dichiarano due fili: clock, per il collegamento al generatore di clock, e il filo per /reset
- Attenzione a `reset_` : si parla di un attivo basso. La convenzione per indicare gli attivi bassi è l'underscore subito dopo il nome. Con questa variabile indichiamo se vogliamo compiere il reset della rete.
- Effettuo un'operazione di assegnamento continuo ponendo i valori dei fili di uscita uguali ai valori contenuti nel registro

- **Parte nuova**:
  - Con always ribadiamo che quanto detto è valido sempre
  - La chiocciola è il costrutto di controllo degli eventi: stabiliamo che ogni volta che si ha un certo evento si verifica quanto segue.
  - In questo esempio gestiamo due eventi:
    - Se il valore di /reset è uguale a 0 porro il valore del registro uguale a quello iniziale da noi stabilito (contenuto_iniziale)
    - Se abbiamo un fronte di salita (*posedge*) del clock e non si ha reset (quindi *reset_* è uguale ad 1) si pone con ritardo $T_{\text{propagation}}$ il valore di REGISTRO e quindi delle uscite.

- **Attenzione**: nella parte nuova abbiamo utilizzato il cosiddetto assegnamento procedurale (rappresentato dal simbolo di minore o uguale). Qual è la differenza tra questo assegnamento e quello continuo?
  - **Assegnamento continuo**: detto bloccante, consiste in un assegnamento che è vero continuamente, ad ogni istante t.
  - **Assegnamento procedurale**: detto non bloccante, consiste in un assegnamento che avviene in un preciso istante temporale (cioè quando si ha un certo evento).

Riprendiamo il concetto di RSS



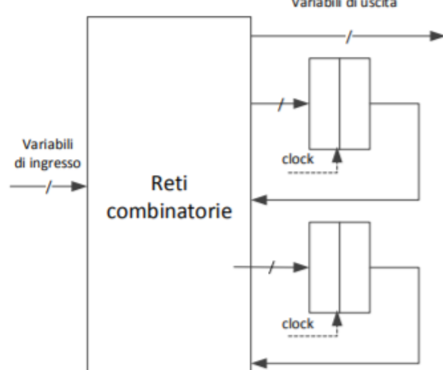
Esempio di RSS con due reti combinatorie.

- Definizione sempre approssimativa: una RSS è una collezione di registri e di reti combinatorie montati in modo arbitrario.
  - Non si hanno anelli di reti combinatorie (altrimenti avrei una rete sequenziale asincrona)
  - Tutti i registri hanno lo stesso clock
  - È possibile avere anelli a patto che sia presente al loro interno almeno un registro. Il registro sappiamo che è una rete non trasparente, quindi permette di evitare i problemi di pilotaggio che già abbiamo visto.

- **Regole di pilotaggio per una RSS**:
  - **Unica regola**: dato t_i l'i-esimo fronte di salita del clock, lo stato di ingresso ai registri deve essere stabile, per ogni i, nel seguente intervallo

$$[t_i - T_{\text{setup}}, t_i + T_{\text{hold}}]$$

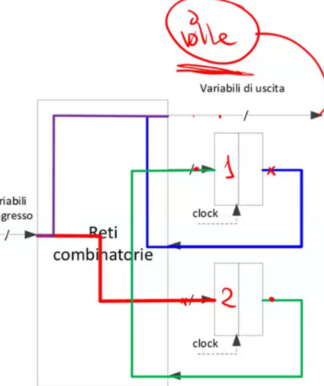
  - **Attenzione**: il clock non può essere fatto veloce quanto voglio.



Se io voglio che le variabili di ingresso arrivino a uno dei due registri in tempo per essere memorizzate allora il tempo di clock dovrà essere sufficientemente largo da permettere alle variabili di ingresso di attraversare la rete combinatoria e arrivare al registro.

Ritardi caratteristici di una RSS

- $T_{\text{in_to_reg}}$: il tempo di attraversamento della più lunga catena fatta di **sole** reti combinatorie che si trovi tra **un piedino di ingresso** fino **all'ingresso di un registro**
- $T_{\text{reg_to_reg}}$: (… … …) **l'uscita di un registro** e **l'ingresso di un registro**
- $T_{\text{in_to_out}}$: (… … …) **un piedino di ingresso** e **un piedino di uscita**
- $T_{\text{reg_to_out}}$: (… … …) **l'uscita di un registro** e **un piedino di uscita**



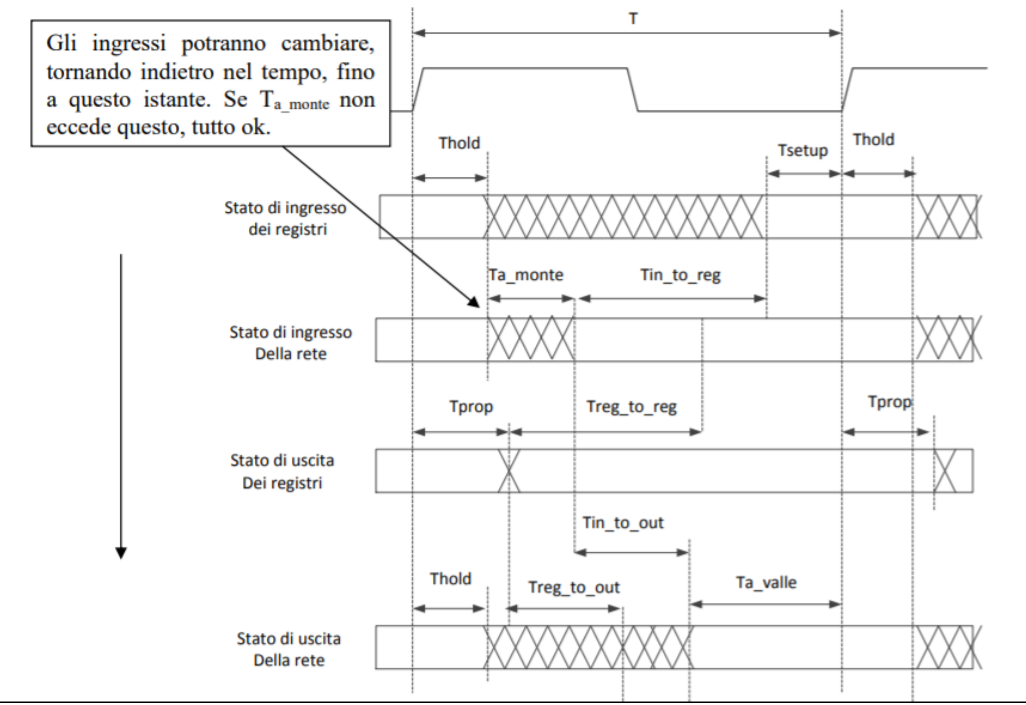
Quindi

- **Vincolo costruttivo dei registri**: Gli ingressi devono essere costanti nell'intervallo
$$[t_i - T_{\text{setup}}, t_i + T_{\text{hold}}]$$
- **Vincolo di pilotaggio in ingresso**: chi pilota gli ingressi (cioè chi sta a monte della RSS) deve avere almeno un tempo $T_{\text{a monte}}$ per poterli cambiare. Devo lasciare una finestra larga almeno $T_{\text{a monte}}$ in ogni periodo clock per il pilotaggio della rete.
- **Vincolo di pilotaggio in uscita**: chi pilota le uscite (cioè chi sta a valle della RSS) deve avere i valori stabili per almeno un tempo $T_{\text{a valle}}$ per poterci fare qualcosa. Devo lasciare una finestra larga almeno $T_{\text{a valle}}$ in ogni periodo di clock perché si possano usare le uscite.

Disuguaglianze di temporizzazione

- Prendiamo l'immagine alla pagina precedente. Si deduce che il tempo T che deve intercorrere tra due successivi segnali di sincronizzazione debba soddisfare le seguenti disuguaglianze:

    - **Percorso da ingresso a registro**: $T \geq T_{\text{hold}} + T_{\text{a monte}} + T_{\text{in_to_reg}} + T_{\text{setup}}$
        - T_{hold}: parte dopo il fronte
        - $T_{\text{a monte}}$: siamo all'ingresso (vincolo di pilotaggio di ingresso)
        - $T_{\text{in_to_reg}}$: tempo per raggiungere il registro
        - T_{setup}: parte prima del fronte
    - **Percorso da registro a registro**: $T \geq T_{\text{prop}} + T_{\text{reg_to_reg}} + T_{\text{setup}}$
        - T_{prop}: tempo necessario affinchè l'uscita si adegui al nuovo valore appena passato.
        - $T_{\text{reg_to_reg}}$: tempo per passare da questo registro a un altro registro
        - T_{setup}: parte prima del fronte
    - **Percorso da ingresso a uscita**: $T \geq T_{\text{hold}} + T_{\text{a monte}} + T_{\text{in_to_out}} + T_{\text{a valle}}$
        - T_{hold}: parte dopo il fronte
        - $T_{\text{a monte}}$: siamo all'ingresso (vincolo di pilotaggio di ingresso)
        - $T_{\text{in_to_out}}$: tempo per passare dall'ingresso all'uscita
        - $T_{\text{a valle}}$: siamo all'uscita (vincolo di pilotaggio di uscita)
    - **Percorso da registro a uscita**: $T \geq T_{\text{prop}} + T_{\text{reg_to_out}} + T_{\text{a valle}}$
        - T_{prop}: tempo necessario affinchè l'uscita si adegui al nuovo valore appena passato.
        - $T_{\text{reg_to_out}}$: tempo per passare dal registro all'uscita
        - $T_{\text{a valle}}$: siamo all'uscita (vincolo di pilotaggio di uscita)



- **Stato di ingresso dei registri**:
  - Abbiamo un tempo T_{hold} dopo il primo fronte di salita.
  - Passato questo periodo possiamo indicare qualunque dato in ingresso (possiamo indicare un valore in qualunque istante di quel periodo, anche modificarlo se necessario)
  - Quando entriamo nel periodo T_{setup} non possiamo più modificare il valore poiché in prossimità del successivo fronte di salita.

- **Stato di ingresso della rete**:
  - Abbiamo un tempo T_{hold} dopo il primo fronte di salita.
  - Abbiamo un tempo $T_{\text{a monte}}$ in cui possiamo pilotare liberamente la rete (quindi porre in ingresso un nuovo valore)
  - Non appena si entra nel periodo $T_{\text{in_to_reg}}$ non sarà più possibile modificare il valore in ingresso in tempo per il prossimo istante di sincronizzazione.

- **Stato di uscita dei registri**:
  - Passaggio da registro a registro: se $T_{\text{prop}} + T_{\text{reg_to_reg}} + T_{\text{setup}}$ è maggiore del clock allora il valore arriva in tempo per il prossimo istante di sincronizzazione nel registro successivo.

- **Stato di uscita della rete**:
  - Abbiamo un tempo T_{hold} dopo il primo fronte di salita.
  - Abbiamo un tempo $T_{\text{reg_to_out}}$ in cui il valore passa dal registro all'uscita.
  - Abbiamo un tempo $T_{\text{a valle}}$ in cui il valore non deve essere toccato e reso disponibile per l'utilizzo.
  - Attenzione: il valore in uscita non dipende solo dai registri, ma anche dagli ingressi. Segue che dobbiamo considerare anche $T_{\text{in_to_out}}$. Non devono arrivare in tempo solo valori provenienti dai registri, ma anche i valori provenienti dall'ingresso.

- **Quando potrà essere usato lo stato di uscita?**
  - Lo stato dei registri avrà attraverso la rete combinatoria per raggiungere l'uscita ($T_{\text{reg_to_out}}$)
  - Lo stato di ingresso della rete avrà attraverso la rete combinatoria per raggiungere l'uscita ($T_{\text{in_to_out}}$)

- **Sottigliezze di cui bisogna tenere conto**:
  - Definiamo T_{sfas} come il **massimo sfasamento tra due clock**.
    Posso avere uno sfasamento tra due clock. Ho lo stesso clock portato a elementi diversi: a qualche registro arriverà prima e qualche altro dopo.

  - Definiamo T_{reg} , cioè il tempo necessario affinchè tutti i flip D flip-flop che costituiscono un registro recepiscano il clock.
    Se abbiamo un registro formato da più di un bit, è impensabile che il cambiamento di tutti i bit avvenga in contemporanea. La cosa che conviene fare è attendere un ulteriore tempo T_{reg} per essere certi che lo stato di uscita di un registro sia cambiato per intero.
    Segue un nuovo valore di T_{prop}

$$T_{prop}' = T_{prop} + T_{\text{reg}}$$

- **Disuguaglianze definitive**:
  - **Percorso da ingresso a registro**: $T \geq T_{sfas} + T_{\text{hold}} + T_{\text{a monte}} + T_{\text{in_to_reg}} + T_{\text{setup}}$
  - **Percorso da registro a registro**: $T \geq T_{sfas} + T_{\text{prop}}' + T_{\text{reg_to_reg}} + T_{\text{setup}}$
  - **Percorso da ingresso a uscita**: $T \geq T_{sfas} + T_{\text{hold}} + T_{\text{a monte}} + T_{\text{in_to_out}} + T_{\text{a valle}}$
  - **Percorso da registro a uscita**: $T \geq T_{sfas} + T_{\text{prop}}' + T_{\text{reg_to_out}} + T_{\text{a valle}}$

  T_{sfas} è molto piccolo, quindi lo considereremo nullo d'ora in poi.

- **Attenzione alla penultima disuguaglianza.** supponiamo di voler ridurre il clock al minimo: la terza disuguaglianza è quella che rompe di più le scatole, considerando che nel secondo membro sono presenti i valori più rompiscatole: $T_{\text{a monte}}$ e $T_{\text{a valle}}$.
    - Possiamo risolvere vietando certi percorsi, per esempio proprio quello che va dall'ingresso all'uscita. Questo tipo di rete è detta **Rete su modello di Moore**.
    - Potrei prendere le uscite direttamente dai registri: questo significa far sparire $T_{\text{reg_to_out}}$ dalla quarta disuguaglianza. Questo tipo di rete è detta **Rete su modello di Mealy ritardato**.

Interazione fra reti

- Abbiamo detto che abbiamo una rete a monte e una rete a valle che interagiscono con la mia RSS.
- Per mettere in relazione due reti dobbiamo garantire il rispetto delle regole di temporizzazione, cosa assai difficile. Per farlo possiamo scegliere una delle seguenti strategie:
    - **Porre lo stesso clock** per le due reti
    - Fare in modo che le reti si mettano d'accordo attraverso **meccanismi di sincronizzazione** detti ***handshake*** (ne riparleremo più avanti).

Ulteriori osservazioni sul pilotaggio della rete

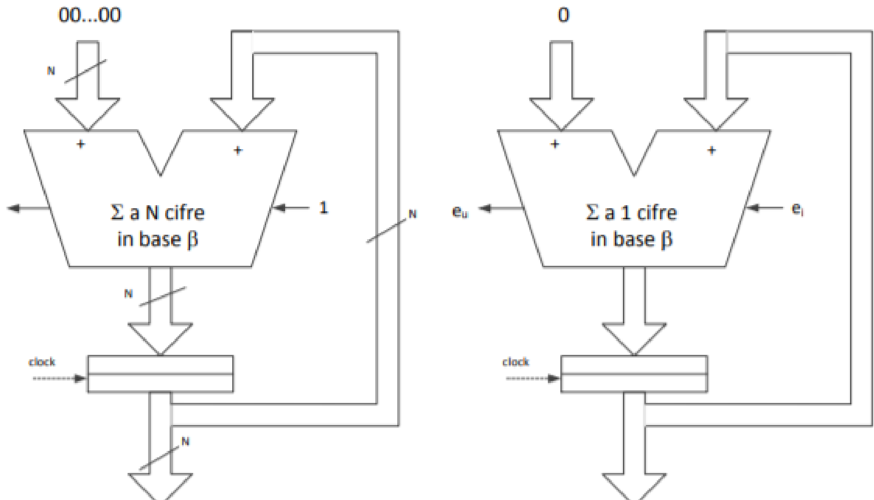
- Lo stato di ingresso di una RSS viene campionato dai registri all'arrivo del clock.

- **Non ci interessa**:
    - cosa faccia lo stato di ingresso tra due clock
    - di quanti bit varia un ingresso
    - se lo stato di ingresso effettivamente varia o no (in ogni caso i due stati saranno interpretati come stati di ingresso differenti)
- Ci interessa soltanto il rispetto dei tempi affinchè un valore eventualmente modificato arrivi in tempo al registro per l'istante di sincronizzazione.

Ribadiamo: la RSS si evolve all'arrivo del clock, non con la variazione degli ingressi.

Esempio di rodaggio: Contatori (ne riparleremo più avanti con le reti di Mealy)

- Introduciamo in modo euristico la seguente rete.
- Un contatore è una RSS il cui stato di uscita può essere visto come un numero naturale ad n cifre in base β, secondo una qualche codifica. Posso parlare, per esempio, di contatori a 2 cifre in base 10 BCD, ma anche di contatori a n cifre in base 2.



- Il contatore svolge una delle seguenti cose:
    - Incrementa di uno (modulo β^n) il valore in uscita (*contatore up*)
    - Decrementa di uno (modulo β^n) il valore in uscita (*contatore down*)
    - Incrementa o decrementa a seconda del valore di una variabile di comando (*contatore up/down*)

- **Come otteniamo una rete del genere**: si mettono insieme un modulo sommatore e un registro. Il primo lo abbiamo già conosciuto introducendo le reti combinatorie e permette di sommare in base β n cifre. Si

pone uno dei due addendi uguale a zero e il riporto in ingresso uguale ad uno.

- **Descrizione in Verilog di *contatori up* in base $\beta \neq 2$**:
```
module ContatoreUp_Ncifre_BaseBeta(numero,clock,reset_);
    input clock,reset_;
    output [W-1:0] numero;
    reg [W-1:0] OUTR; assign numero=OUTR;
    always @(reset_==0) #1 OUTR<=0;
    always @(posedge clock) if (reset_==1) #3
        OUTR<= Inc_N_cifre_beta(OUTR);

    function [W-1:0] Inc_N_cifre_beta;
        input [W-1:0] numero;
        casex(numero)
            <cod_0> : Inc_N_cifre_beta = <cod_1>;
            <cod_1> : Inc_N_cifre_beta = <cod_2>;
            …
            default : Inc_N_cifre_beta = 'BXXX…XXX;
        endcase
    endfunction
endmodule
```
  - Complessivamente non notiamo cose strane: tutte le cose, in particolare i costrutti per la gestione degli eventi, sono già state introdotte.
  - L'unica cosa che può suscitare perplessità è il ricorso a una function per rappresentare l'incremento. La cosa è obbligata (non possiamo fare diversamente, cit.) in quanto il simbolo + in Verilog rappresenta un sommatore ad N cifre in base due.
  - Nella funzione utilizzeremo l'equivalente dello switch in C++ per associare a tutti i numeri possibili su N cifre in base β il corrispondente numero successivo.
  - Osservazione sull'assegnamento continuo di *OUTR*: porre OUTR uguale a 0 significa porre tutti i W bit uguali a 0.

- **Descrizione in Verilog di *contatori up* in base 2**: la cosa si semplifica sicuramente….
```
module ContatoreUp_Ncifre_Base2(numero,clock,reset_);
    input clock,reset_;
    output [N-1:0] numero;
    reg [N-1:0] OUTR; assign numero=OUTR;
    always @(reset_==0) #1 OUTR<=0;
    always @(posedge clock) if (reset_==1) #3 OUTR <= numero+1;
endmodule
```

- **Descrizione in Verilog di *contatori down***: uguali ai precedenti, cambia solo la function (se siamo in una base $\boldsymbol{\beta} \neq \mathbf{2}$, si associa a ciascun numero il precedente e non il successivo) o l'operatore (se siamo in base 2, - al posto del +)

- **Contatori con ingresso di abilitazione :**
  - Noi non andiamo ad alterare la struttura del sommatore: semplicemente non utilizzeremo più un generatore di costante per rappresentare il riporto in ingresso e_i
  - Precisamente:
    - Se $e_i = 0$ all'arrivo del clock si ha conservazione
    - Se $e_i = 1$ all'arrivo del clock si ha incremento o decremento in base al contatore scelto.

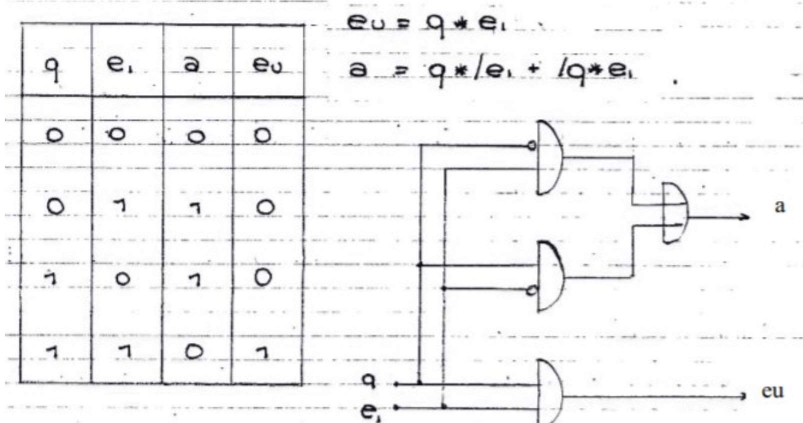
- **Descrizione in Verilog di contatori con ingresso di abilitazione in base 2**:
```
module ContatoreUp_Ncifre_Base2 (numero, clock, reset_, ei)
    input clock, reset_, ei;
    […]
    always @(posedge clock) if (reset==1) #3 OUTR<=OUTR+{'B00…00,ei};
endmodule
```

- **Relativamente all'assegnamento procedurale di OUTR**: sommo una stringa di N bit di cui l'ultimo vale e_i. Gli altri bit valgono zero.

- **Descrizione in Verilog di contatori con ingresso di abilitazione in base $\beta \neq 2$**:
```
module ContatoreUp_Ncifre_Base2 (numero, clock, reset_, ei)
    input clock, reset_, ei;
    […]
    always @(posedge clock) if (reset==1) #3
        casex(ei)
            0: OUTR<=OUTR;
            1: OUTR<=Inc_N_cifre_beta(OUTR)
        endcase
endmodule
```
  - Il *casex* viene utilizzato per distinguere i due casi possibili in base al valore di e_i. Mentre prima utilizzare casex era una complicazione esistenziale adesso è necessario poiché ci troviamo in una base β ≠ 2.

- **Scomposizione di contatori**:
  - Un contatore ad N cifre può essere scomposto in una serie di contatori ad una cifra.
  - Il collegamento è lo stesso visto per i somamtori: il ripple carry (catena di riporti)

  - **Elemento contatore in base 2**:
    - **Tavola di verità**: scriviamo la tavola di verità, otterremo che la sintesi è quella dell'incrementatore ad 1 cifra in base 2

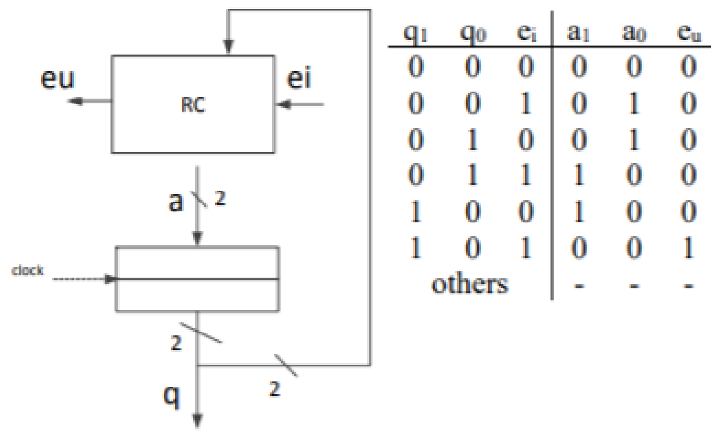


    - **Descrizione in Verilog**:
```
module Elemento_Contatore_Base_2(eu,q,ei,clock,reset_);
    input clock,reset_;
    input ei;
    output eu,q;
    reg OUTR; assign q=OUTR;
    wire a; // variabile di uscita dell'incrementatore
    assign {a,eu}= ({q,ei}=='B00) ?'B00:
                   ({q,ei}=='B10) ?'B10:
                   ({q,ei}=='B01) ?'B10:
                  /*({q,ei}=='B11)*/'B01;
    always @(reset_==0) #1 OUTR<=0;
    always @(posedge clock) if (reset_==1) #3 OUTR<=a;
endmodule
```

    - **Sintesi in Verilog**:
```
module Elemento_Contatore_Base_2(eu,q,ei,clock,reset_);
    input clock,reset_;
```

```
            input ei;
            output eu,q;
            reg OUTR; assign q=OUTR;
            wire a; // variabile di uscita dell'incrementatore
            assign a = q^ei;
            assign eu = (q&ei);

            always @(reset_==0) #1 OUTR<=0;
            always @(posedge clock) if (reset_==1) #3 OUTR<=a;
        endmodule
```

- **Contatore ad una cifra in base 3**:
    - Teniamo conto che ci vogliono 2 bit per codificare una cifra.
    - Possiamo dire: 0='B00, 1='B01, 2='B10.
    - Segue la necessità di
        - un registro a due bit
        - una rete che
            - Ha in ingresso i 2 bit che escono dal registro ed un riporto entrante
            - Ha in uscita i due bit che vanno in ingresso al registro ed il riporto uscente.
    - Scriviamo la tavola di verità relativa alla rete

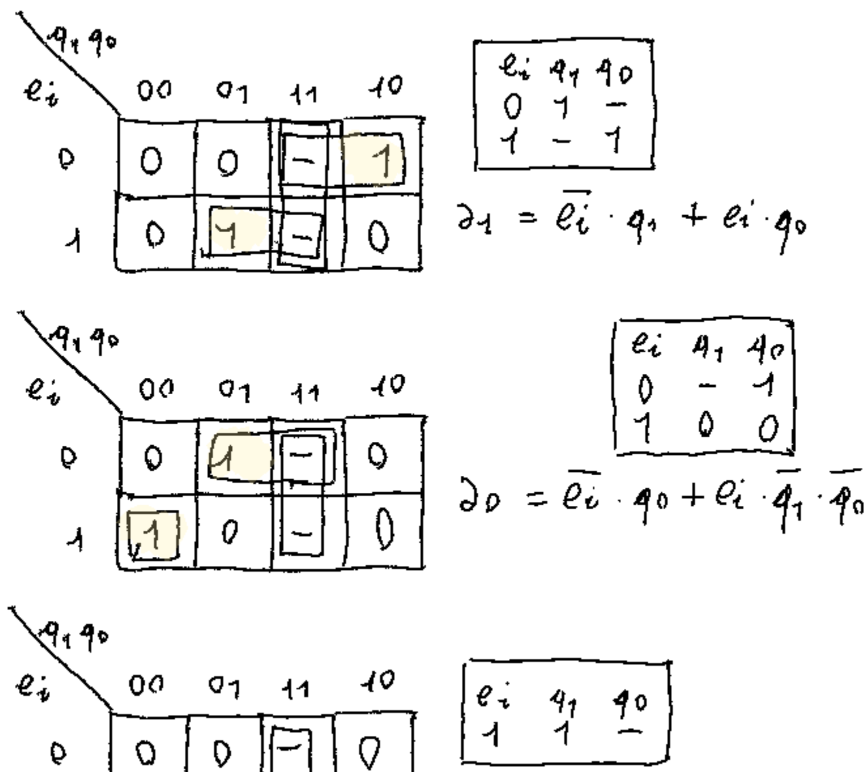


q_1	q_0	e_i	a_1	a_0	e_u
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	0	0	1
others			-	-	-

- **Descrizione in Verilog**:

```
module Elemento_Contatore_Base_3(eu,q1_q0,ei,clock,reset_);
    input clock,reset_;
    input ei;
    output eu;
    output [1:0] q1_q0;
    reg [1:0] OUTR; assign q1_q0=OUTR;
    wire [1:0] a1_a0; // variabile di uscita dell'incrementatore
    assign {a1_a0,eu}= ({q1_q0,ei}=='B000)?'B000:
                       ({q1_q0,ei}=='B010)?'B010:
                       ({q1_q0,ei}=='B100)?'B100:
                       ({q1_q0,ei}=='B001)?'B010:
                       ({q1_q0,ei}=='B011)?'B100:
                       ({q1_q0,ei}=='B101)?'B001:
                       /* default */ 'BXXX;
    always @(reset_==0) #1 OUTR<='B00;
    always @(posedge clock) if (reset_==1) #3 OUTR<=a1_a0;
endmodule
```

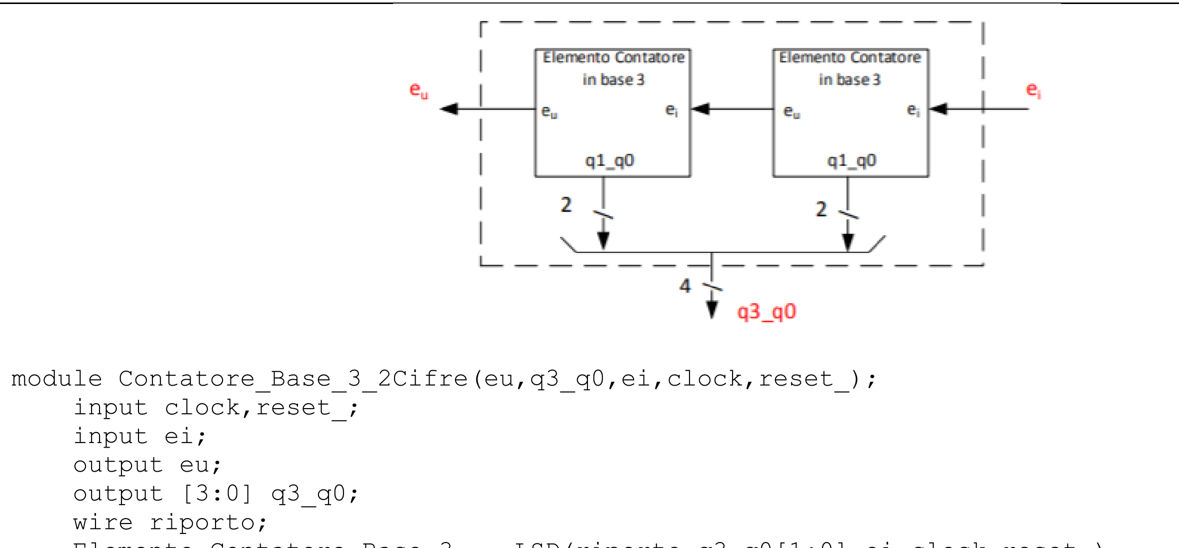
- **Sintesi in Verilog**:
    - Otteniamo la sintesi a costo minimo partendo dalle mappe di Karnaugh



    - Segue il seguente codice, da sostituire a una parte della descrizione

```
wire [1:0] a1_a0; // variabile di uscita dell'incrementatore
wire a1, a0; // variabili a un bit che compongono a1_a0
assign a1_a0={a1,a0};
assign eu=q1_q0[1]&ei;
assign a1=(q1_q0[0]&ei)|(q1_q0[1]&~ei);
assign a0=(~q1_q0[1]&~q1_q0[1]&ei)|(q1_q0[0]&~ei);
```

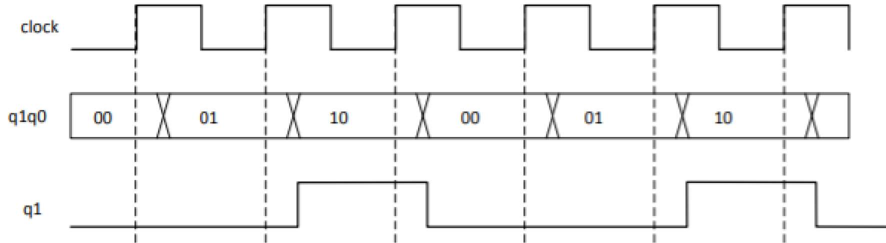
- **Contatore ad N cifre in base 3**:
    - Si collegano insieme gli N moduli necessari appena descritti e sintetizzati con Verilog. Otteniamo:




```
module Contatore_Base_3_2Cifre(eu,q3_q0,ei,clock,reset_);
    input clock,reset_;
    input ei;
    output eu;
    output [3:0] q3_q0;
    wire riporto;
    Elemento_Contatore_Base_3    LSD(riporto,q3_q0[1:0],ei,clock,reset_),
                                 MSD(eu,q3_qo[3:2],riporto,clock,reset_);
endmodule
```

Contatori e divisione in frequenza

- Si dice che i contatori dividono la frequenza di clock per un dato valore.
- Vediamo il MSB del contatore in base 3 per ottenere un clock che va tre volte più lento.



La cifra più significativa di un contatore ad N cifre in basse due che riceve clock a periodo T è a sua volta un clock a periodo $2^N \cdot T$. Si noti che per generare un clock in questo modo si possono usare solo uscite di registri, mai quelle di reti combinatorie (le reti combinatorie potrebbero ballare, il clock deve essere assolutamente stabile).

Capitolo 34

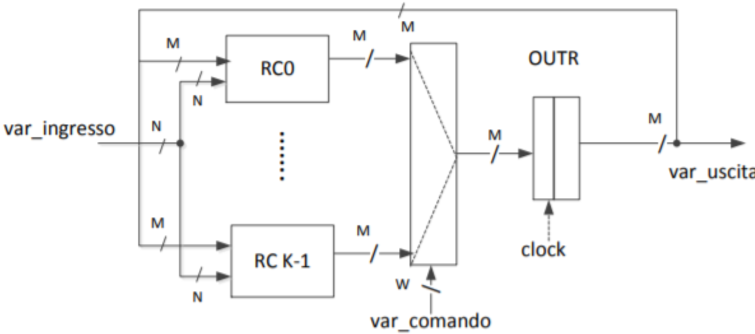
Mercoledì 11/11/2020

Registri multifunzionali

- Un registro multifunzionale è una rete che all'arrivo del clock memorizza nel registro stesso il valore di uscita di una delle possibili K funzioni combinatorie possibili. La scelta tra le K funzioni viene determinata da W variabili di comando ($2^W = K \Leftrightarrow W = \lceil \log_2 K \rceil$)

- Le reti combinatorie possono essere realizzate a nostro piacere, senza vincoli particolari.
- Queste reti possono avere in ingresso anche il valore di uscita del registro.

- **Sintetizzazione**:
  - La sintetizzazione si ottiene sfruttando un multiplexer con K ingressi, W variabili di comando e una singola uscita.
  - Gli ingressi consistono nelle uscite delle K reti combinatorie presenti.
  - L'uscita del multiplexer va in ingresso nel registro e viene salvata se arriva in tempo per il segnale di clock.
  - L'uscita del registro sarà l'uscita della rete. L'uscita può essere rimandata in ingresso nelle reti combinatorie.



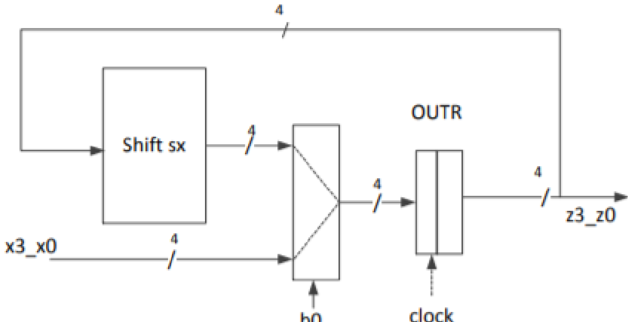
- **Descrizione in Verilog**:

```
module Registro_Multifunzionale(var_uscita,var_ingresso,
   var_comando,clock,reset_);
   input clock,reset_;
   input [N-1:0] var_ingresso;
   input [W-1:0] var_comando;
   output [M-1:0] var_uscita;

   reg [M-1:0] OUTR; assign var_uscita=OUTR;

   always @(reset_==0) #1 OUTR<=contenuto_iniziale;
   always @(posedge clock) if (reset_==1) #3
      casex(var_comando)
          0  : OUTR<=F0(var_ingresso,OUTR);
          ...
          ...
          K-1: OUTR<=FK-1(var_ingresso,OUTR);
      endcase
endmodule
```

Tenerne conto andando avanti, in particolare quando faremo la sintesi di RSS complesse con scomposizione in PO/PC

  - Imposto gli ingressi e le uscite.
  - Imposto il registro e svolgo assegnamento continuo (l'unica variabile di uscita della rete totale è quella del registro, OUTR)
  - I costrutti sono i soliti: l'unica differenza sostanziale si ha in *casex* con la chiamata di funzioni Fk in base alle variabile di comando posta in ingresso.

- **Esempio di registro multifunzionale**: registro bifunzionale di caricamento/traslazione sx. Supponiamo che il registro lavori su 4 bit. Abbiamo come funzioni:
    - Il cortocircuito degli ingressi
    - Un qualcosa che è poco più di un cortocircuito: lo shift a sx (mi limito solo a spostare elementi).



Vediamo la descrizione in Verilog:

```
module Registro_CaricaParallelo_TraslaSinistro(z3_z0,x3_x0,b0,
clock,reset_);
    input clock,reset_;
    input [3:0] x3_x0;
    input b0;
    output [3:0] z3_z0;

    reg [3:0] OUTR; assign z3_z0=OUTR;

    always @(reset_==0) #1 OUTR<='B0000;
    always @(posedge clock) if (reset_==1) #3
        casex(b0)
            'B0: OUTR<=x3_x0;
            'B1: OUTR<={OUTR[2:0],1'B0};
        endcase
endmodule
```

<u>Blocco che riproporremo tante volte</u>

- Attenzione al numero di bit per l'input x3_x0 e l'output z3_z0
- Facciamo le solite cose relativamente ai registri, ma attenzione a *casex*
    - Se indico 0 con la variabile di comando pongo come valore del registro l'input x3_x0
    - Se indico 1, invece, compio lo shift: prendo le tre cifre meno significative e aggiungo una costante 0 come nuova cifra meno significativa.
      Attenzione: non ha senso effettuare più assegnamenti trattando OUTR come un array. Il clock arriva contemporaneamente in tutti i bit del registro (fare più assegnamenti significa contraddire quanto detto)

<u>Dopo aver descritto in modo informale qualche esempio passiamo a descrivere modelli formali per la sintesi di reti RSS.</u>

Modello di Moore

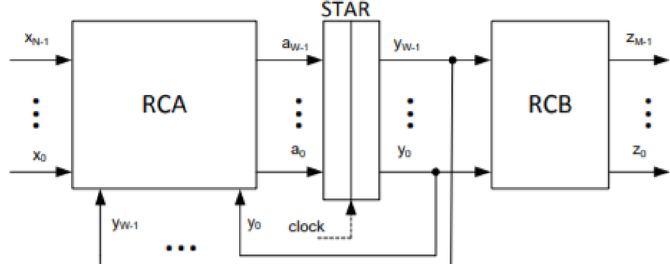
- Abbiamo già anticipato qualcosa parlando delle disuguaglianze di temporizzazione: nel modello di Moore lo stato di uscita dipende esclusivamente dallo stato interno e non dallo stato di ingresso.

- **Ingredienti**:
    - Un insieme di N variabili logiche di ingresso
    - Un insieme di M variabili logiche di uscita
    - Un meccanismo di marcatura che permetta ad ogni istante di marcare uno stato interno presente. Tutti gli stati interni appartengono a un insieme finito di K stati interni: $S \equiv \{S_0, \dots, S_{K-1}\}$. Parlare di meccanismo di marcatura significa, sostanzialmente, stabilire la strutture del registro STAR:

        - batteria di d flip-flop, oppure
        - batteria di jk flip-flop (che introdurremo più avanti).
    - Due leggi di evoluzione nel tempo:
        - $A: X \times S \rightarrow S$, metto in relazione gli stati di ingresso con gli stati interni
        - $B: S \rightarrow Z$, metto in relazione gli stati interni con gli stati di uscita.
            - Non posso avere una legge del tipo $B: X \times S \rightarrow Z$, altrimenti non avrei più una rete secondo modello di Moore.
    - Una legge di temporizzazione:
      Dato 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,
        - *Individuare il nuovo stato interno da marcare $S' = A(X, S)$*
        - *Attendere un tempo T_{prop} dopo il fronte di salita*
        - *Promuovere S' al rango di stato interno marcato.*

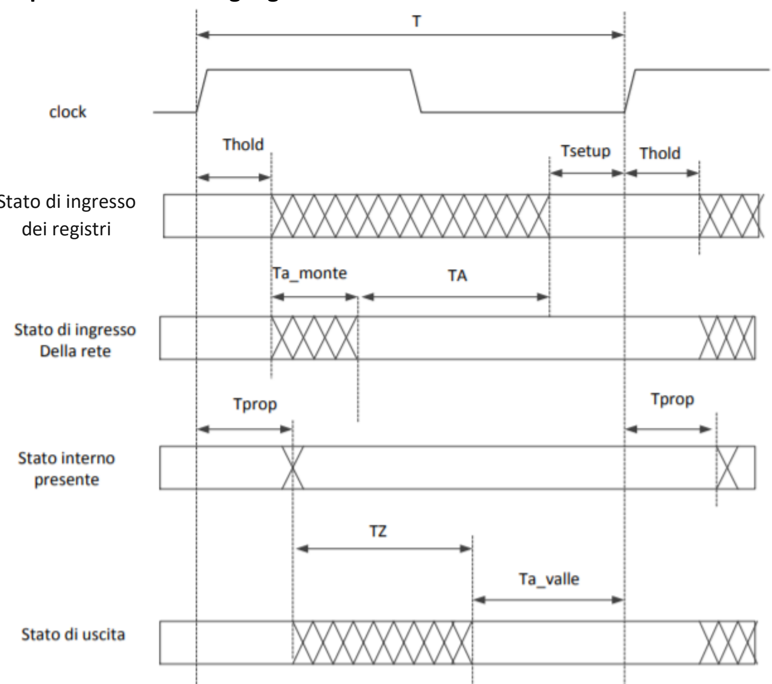
      E, inoltre, individuare continuamente $Z = B(S)$ e presentarlo in uscita.

- **Struttura di una RSS di Moore**:



    - All'interno di una rete di Moore abbiamo un registro detto STAR, cioè *status register*. Il compito di questo registro è memorizzare lo stato interno che determinerà quello di uscita.
    - Ribadiamo che il registro consiste in una collezione di D-flip-flop, e quindi non è trasparente. Il nuovo stato interno viene presentato alla rete dopo un tempo T_{prop} dal posedge del clock.
    - Sono presenti due reti combinatorie: RCA ed RCB. La prima implementa la legge di evoluzione nel tempo A, la seconda la legge di evoluzione B.

- **Diagramma di temporizzazione e disuguaglianze**:



- Stato di ingresso dei registri:
    - Si attende un tempo T_{hold} dopo il posedge del clock (lo stato rimane costante)
    - Posso modificare liberamente lo stato di ingresso del registro finchè non mi troverò a cavallo del *posedge* del clock, quindi nella finestra $[T_{setup}\ ;\ T_{hold}]$
    - Quando mi trovo nella finestra lo stato non può essere più modificato.
- Stato di ingresso della rete:
    - Si attende un tempo T_{hold} dopo il *posedge*
    - Chi sta a monte della rete ha a disposizione un tempo $T_{a\ monte}$ per poter pilotare la rete
    - Lo stato di ingresso immesso impiega un periodo T_A per raggiungere il registro STAR. Ovviamente dare meno tempo significa impedire allo stato di ingresso di arrivare in tempo per il clock al registro.
- Stato interno presente:
    - Ho un tempo T_{prop} prima che il nuovo stato interno venga presentato alla rete.
    - Lo stato interno rimane costante fino al prossimo fronte di salita, precisamente fino a un tempo T_{prop} dopo il successivo fronte di salita.
- Stato di uscita:
    - Attendo un tempo $T_{prop} + T_B$ per ottenere in uscita lo stato interno della rete.
    - Con T_B si intende il tempo necessario allo stato interno, dopo essere stato presentato alla rete, per raggiungere l'uscita della rete attraversando la rete combinatoria RCB.

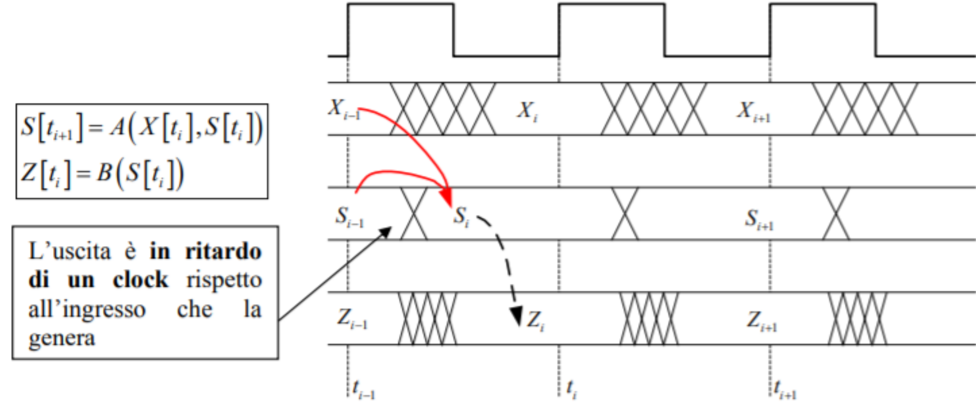
- Tenendo conto di tutte queste cose otteniamo le seguenti disuguaglianze di temporizzazione:
    - **Da ingresso a STAR**: $T \geq T_{\text{hold}} + T_{\text{a monte}} + T_A + T_{\text{setup}}$
    - **Da STAR a STAR**: $T \geq T_{prop} + T_A + T_{\text{setup}}$
    - **Da STAR a uscita**: $T \geq T_{prop} + T_B + T_{a\ valle}$

- Quale di queste disuguaglianze è la meno restrittiva? La seconda, poiché $T_{\text{prop}} \approx T_{\text{hold}}$ e $T_{\text{a monte}}$ **TBD** T_{prop}

- **Evoluzione nel tempo di una RSS di Moore**:
    - Bisogna tener conto di un aspetto solitamente sottovalutato e fonte di sbagli durante le pr4ove d'esame.
    - Supponiamo di avere degli istanti di sincronizzazione t_i con $i > 0$.
    - Analizziamo lo stato interno in un istante t_i e lo stato di uscita in un istante t_{i+1}. Troviamo che:
        - $S[t_i] = A(X[t_{i-1}], S[t_{i-1}])$
        - $Z[t_{i+1}] = B(S[t_i])$

      L'uscita è in ritardo di un clock rispetto all'ingresso che la genera. Troviamo che lo stato interno in un istante t_i dipende dagli ingressi e dallo stato interno relativi all'istante precedente t_{i-1}. Stessa cosa per le uscite: dipendono dallo stato interno relativo all'istante precedente.

      Quindi: z_i, se si fa un percorso a ritroso, dipende da x_0 (dal valore posto a reset), da tutta la storia temporale degli ingressi dal primo ingresso fino all'ingresso $i - 1$.



Descrizione di una rete di Moore in linguaggio Verilog

```
module Rete_di_Moore(zM-1,...,z0,xN-1,...,x0,clock,reset_);
    input clock,reset_;
    input xN-1,...,x0;
    output zM-1,...,z0;

    reg [W-1:0] STAR; parameter S0=codifica0,...,SK-1=codificaK-1;

    assign {zM-1,...,z0} = (STAR==S0) ?ZS0 :
                           (STAR==S1) ?ZS1 :
                             ...
                         /*(STAR==SK-1)*/ ZSK-1;

    always @(reset_==0) #1 STAR<=stato_interno_iniziale;
    always @(posedge clock) if (reset_==1) #3
        casex(STAR)
            S0 : STAR<=AS0(xN-1,...,x0);
            S1 : STAR<=AS1(xN-1,...,x0);
            ...
            SK-1: STAR<=ASK-1(xN-1,...,x0);
        endcase
endmodule
```

Con *parameter* vado a definire delle costanti. In Verilog è presente anche la keyword *localparam*. Si tenga conto che con la prima i valori dei parametri possono essere sovrascritti dall'esterno. Nelle prove pratiche conviene usare la seconda.

(ci sta bene, cit.) ---> #T

Rete B

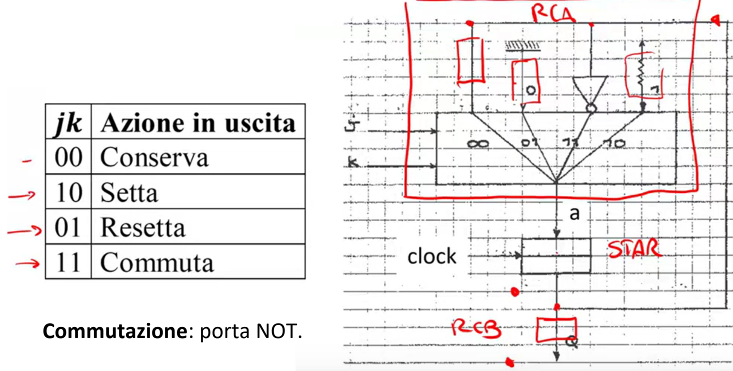
Rete A

Avrei potuto scrivere
`STAR <= legge_A(xN-1,...,x0, STAR);`

Abbiamo invece scritto K espressioni combinatorie diverse, ciascuna associata a un certo contenuto del registro STAR.

Esempio di rete di Moore: Flip-flop JK

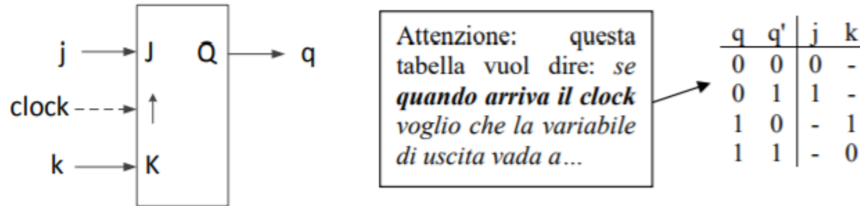
Il FF JK consiste in una rete sequenziale sincronizzata, con due ingressi (j, k) ed un'uscita, che all'arrivo del clock valuta gli ingressi e decide come comportarsi:



jk	Azione in uscita
00	Conserva
10	Setta
01	Resetta
11	Commuta

Commutazione: porta NOT.

- Questa rete può essere vista come un registro multifunzione ad un bit: abbiamo un multiplexer con quattro ingressi e due variabili di pilotaggio. Le operazioni che andiamo a eseguire non richiedono logica particolare….
- Nell'immagine, che consiste nella foto della dispensa con disegni del docente, si associano le varie parti alle componenti di una rete di Moore: RCB consiste in un cortocircuito mentre RCA in quello che abbiamo detto prima.
- La **tabella di applicazione** è simile a quella del Latch-SR.



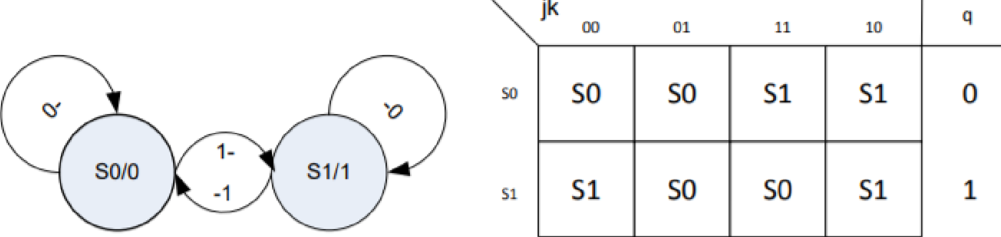
q	q'	j	k
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

Attenzione: perché mettiamo una variabile non specificata in ogni riga?

- Con $j = 0$ potremo avere come azione la conservazione o il reset. Poiché lo stato iniziale è 0 e quello successivo sempre 0 allora il valore di k non ha importanza
- Con $j = 1$ potremo avere come azione il settaggio o la commutazione. Poiché lo stato iniziale è 0 e quello successivo 1 allora il valore di k non ha importanza.
- Con $k = 1$ potremo avere il reset o la commutazione. Poiché lo stato iniziale è 1 e quello successivo è 0 allora il valore di j non ha importanza.
- Con $k = 0$ potremo avere la conservazione o il settaggio. Poiché lo stato iniziale è 1 e quello successivo sempre 1 allora il valore di j non ha importanza.

- Come abbiamo già intuito il FF JK serve a memorizzare un bit, e quindi posso associargli due stati: S0 ed S1
- Lo stato interno è memorizzato con una variabile di stato che varrà 0 e 1. RCB, segue, è un cortocircuito.

- **Tabella e grafo di flusso**:



jk	00	01	11	10	q
S0	S0	S0	S1	S1	0
S1	S1	S0	S0	S1	1

Nulla di particolare di cui tenere conto: quanto visibile si basa sulle spiegazioni relative alla tabella di applicazione. L'unica cosa da evidenziare è che non ha senso parlare di stati stabili e quindi cerchiare elementi nella tabella di flusso. Nella RSS gli elementi che memorizzano gli stati interni sono i registri. Gli stati sono stabili per un periodo di clock, se la rete è pilotata correttamente, e ad ogni clock ho una nuova transizione in cui potrei porre come stato interno quello posto al clock precedente (*marcatura del medesimo stato interno in cui la rete già si trova*).

- **Osservazione**: è sintomo di poca comprensione degli argomenti iniziare dalla sintesi della rete invece che dalla descrizione. È un po' come iniziare a realizzare qualcosa saltando la fase di progettazione.

- **Descrizione in Verilog**: la descrizione si ottiene semplificando quella generale vista prima

```
module FlipFlop_JK(q,j,k,clock,reset_);
    input clock,reset_;
    input j,k;
    output q;
    reg STAR; parameter S0='B0,S1='B1;
    assign q=(STAR==S0)?0:1;

    always @(reset_==0) #1 STAR<=S0;
    always @(posedge clock) if (reset_==1) #3
        casex(STAR)
            S0: STAR<=(j==0)?S0:S1;
            S1: STAR<=(k==0)?S1:S0;
        endcase
endmodule
```

- **Sintetizziamo RCA ed RCB.** Basta scegliere una codifica per gli stati interni
    - `S0='BO`
    - `S1='B1`

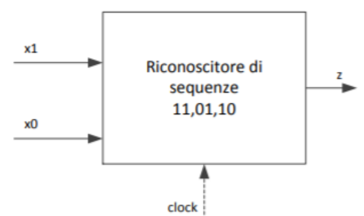
  e si ottiene ------>

jk	00	01	11	10	q
S0	S0	S0	S1	S1	0
S1	S1	S0	S0	S1	1

y0 \ jk	00	01	11	10
0	0	0	1	1
1	1	0	0	1

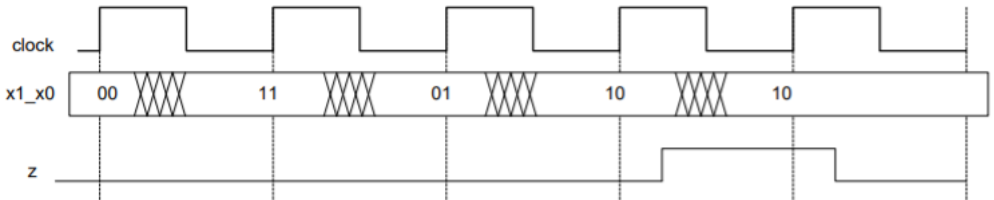
$a_0 = j \cdot \overline{y_0} + \overline{k} \cdot y_0$, $q = y_0$.

Riconoscitore di sequenze 11, 01, 10

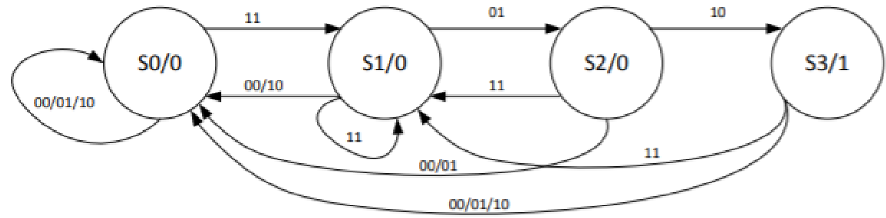


- Rete che sblocca, in un certo senso, la serratura di una cassaforte. La combinazione è data da una sequenza di parole di N bit che si devono presentare in tre clock consecutivi.
- Ovviamente se un valore permane per più di un clock non si riconosce la sequenza.
- Ovviamente parliamo di una rete con memoria, devo ricordare una sequenza di stati di ingresso.

- Dobbiamo ricordare il numero di passi di sequenza corretti consecutivi visti finora.
- Saranno richiesti K+1 stati interni per una sequenza di K stati di ingresso da riconoscere: tutti gli stati interni presentano un'uscita uguale a 0, tranne l'ultimo dove sarà uguale ad 1 se la sequenza dei K stati interni precedenti è quella giusta.



- **Partiamo da un grafo di flusso e descriviamolo**:



- S0:
  - Punto di partenza.
  - Passo allo stato S1 se pongo in ingresso il primo elemento della sequenza (11)
  - Se pongo tutti gli altri ingressi possibili (incluso lo stato di ingresso 00 non considerato nella sequenza) rimango in S0 perché non sto indicando la sequenza giusta.
- S1:
  - Abbiamo posto nel clock precedente il primo stato di ingresso della sequenza
  - Passo allo stato S2 se pongo in ingresso lo stato 01 (cioè il secondo elemento della sequenza)
  - Se pongo lo stato di ingresso 11 rimango in S1 (stiamo sbagliando la sequenza, è vero, ma porre 11 potrebbe essere l'inizio della sequenza che vogliamo riconoscere… non ha senso ritornare ad S0 perché ponendo 11 indico il primo stato della sequenza)
  - Ritorno ad S0 con tutti gli altri stati possibili in ingresso.
- S2:
  - Abbiamo posto nel clock precedente il secondo stato di ingresso della sequenza.
  - Passo allo stato S3 che presenta uscita 1 se pongo in ingresso lo stato 10, cioè l'ultimo stato di ingresso della sequenza. Passare allo stato S3 significa aver riconosciuto la sequenza e restituire 1 in uscita.
  - Ritorno allo stato S1 se lo stato di ingresso posto è 11
  - Ritorno allo stato S0 in tutti gli altri casi
- S3:
  - La sequenza è stata riconosciuta e l'uscita, conseguentemente, è uguale ad 1.
  - Se pongo come stato di ingresso 11 ritorno allo stato interno S1 (stesso discorso)
  - In tutti gli altri casi ritorno allo stato S0

PROMEMORIA: LA SEQUENZA DI N STATI DOVRA' PRESENTARSI IN n CLOCK, NON PIU' VELOCEMENTE.

- **Tabella di flusso risultante**:

	00	01	11	10	z
S_0	S0	S0	S1	S0	0
S_1	S0	S2	S1	S0	0
S_2	S0	S0	S1	S3	0
S_3	S0	S0	S1	S0	1

(colonne: x_1x_0)

- **Descrizione in Verilog**:

```
module Riconoscitore_di_Sequenza(z,x1_x0,clock,reset_);
  input clock,reset_;
  input [1:0] x1_x0;
  output z;

  reg [1:0] STAR; parameter S0='B00, S1='B01, S2='B10, S3='B11;

  assign z=(STAR==S3)?1:0;
  always @(reset_==0) #1 STAR<=S0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(x1_x0=='B11)?S1:S0;
      S1: STAR<=(x1_x0=='B01)?S2:(x1_x0=='B11)?S1:S0;
      S2: STAR<=(x1_x0=='B10)?S3:(x1_x0=='B11)?S1:S0;
      S3: STAR<=(x1_x0=='B11)?S1:S0;
    endcase
endmodule
```

Scelgo una codifica per gli stati interni

  - Si creano le costanti con i vari stati interni da riconoscere
  - Si crea un registro STAR che conterrà lo stato interno precedente.
  - Sappiamo che l'uscita della rete sarà uguale ad 1 solo se ci troviamo nello stato S3, altrimenti è 0.

  - Copiamo in *casex* in modo pedissequo il contenuto della tabella di flusso

```
STATO_INIZIALE: STAR<=(x_1x_0='ELEMENTO COMBINAZIONE) ? STATO_SUCCESSIVO : STATO_ERRORE
```

    Si tiene conto di quanto spiegato prima nell'indicare il nuovo stato da porre nel registro.

- **Sintesi**:
  - Per la sintesi abbiamo deciso di adottare il modello di Moore.
  - Adottare il modello di Moore significa dover sintetizzare due reti combinatorie: RCA ed RCB.
  - Per rappresentare gli stati interni dobbiamo scegliere una codifica: gli stati possibili sono 4, quindi avrò bisogno di almeno due bit per rappresentare tutti gli stati interni. La codifica viene scelta in modo tale da renderci la vita più semplice in RCB. Vedremo che RCB dipende esclusivamente dalla codifica adottata! Utilizzeremo la codifica già vista nella descrizione in Verilog.

  - **Passare alle mappa di Karnaugh**: abbiamo codificato gli stati interni, li sostituisco nella tabella…

x_1x_0	00	01	11	10	z
S_0	S0	S0	S1	S0	0
S_1	S0	S2	S1	S0	0
S_2	S0	S0	S1	S3	0
S_3	S0	S0	S1	S0	1

Sint	y_1y_0
S_0	00
S_1	01
S_2	10
S_3	11

y_1y_0 \ x_1x_0	00	01	11	10	z
00	00	00	01	00	0
01	00	10	01	00	0
11	00	00	01	00	1
10	00	00	01	11	0

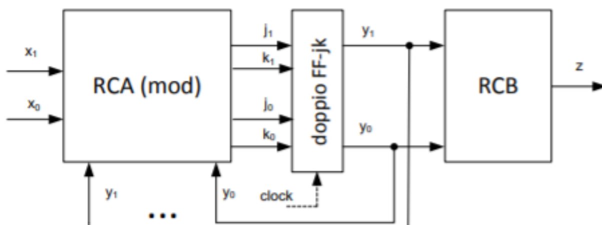
a_1a_0

Stiamo lavorando con una sola mappa per sintetizzare due uscite. Otteniamo

$a_1 = \overline{x_1}\cdot x_0 \cdot \overline{y_1}\cdot y_0 + x_1 \cdot \overline{x_0}\cdot y_1 \cdot \overline{y_0}$ $\qquad a_0 = x_1 \cdot x_0 + x_1 \cdot y_1 \cdot \overline{y_0}$ $\qquad z = y_1 \cdot y_0$

- Ricordare che l'ultimo stato interno è riconosciuto da 11, quindi basta una porta AND come rete combinatoria RCB. Tutte le codifiche, tranne quella relativa ad S3, presentano almeno un zero al loro interno: RCB darà in uscita, in questi casi, 0.

- **Sintesi in modo diverso**: invece di utilizzare un registro STAR si potrebbe porre un doppio FF-JK. Si tratta sempre di una rete di Moore. La cosa pare una complicazione, ma in realtà non lo è: le sintesi saranno piene di non specificati (quindi in alcuni si avrà una cosa addirittura più semplice).



Ricordiamo la tavola di verità relativa al Flip-flop JK e alle operazioni possibili nel pilotaggio

q	q'	j	k
0	0	0	-
0	1	1	-
1	0	-	1
1	1	-	0

jk	**Azione in uscita**
00	Conserva
10	Setta
01	Resetta
11	Commuta

Osserviamo le singole cifre. Se voglio mantenere la cifra piloterò per conservare, altrimenti piloterà per settare o resettare. Riscriviamo le mappe di Karnaugh e sintetizziamo la rete RCA

q q'	jk
0 0	0-
0 1	1-
1 0	-1
1 1	-0

j_1k_1

y_1y_0 \ x_1x_0	00	01	11	10
00	0-	0-	0-	0-
01	0-	1-	0-	0-
11	-1	-1	-1	-1
10	-1	-1	-1	-0

j_0k_0

y_1y_0 \ x_1x_0	00	01	11	10
00	0-	0-	1-	0-
01	-1	-1	-0	-1
11	-1	-1	-0	-1
10	0-	0-	1-	1-

y_1y_0	z
00	0
01	0
10	0
11	1

$j_1 = \overline{x_1}\cdot x_0 \cdot y_0$, $k_1 = \overline{x_1} + x_0 + y_0$, $j_0 = x_1 \cdot y_1 + x_1 \cdot x_0$, $k_1 = \overline{x_1} + \overline{x_0}$, $z = y_1 \cdot y_0$.

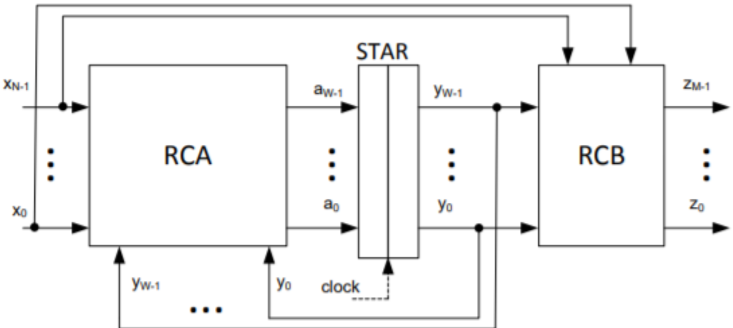
Capitolo 35

Giovedì 12/11/2020

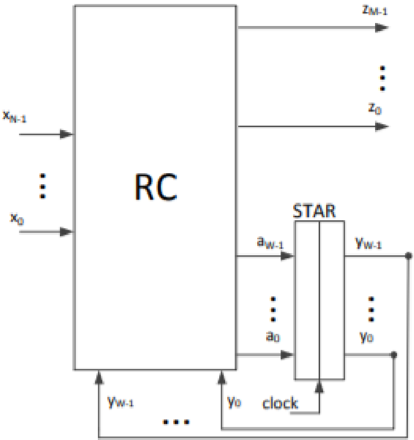
Modello di Mealy

- Abbiamo visto nelle scorse lezioni il Modello di Moore dove l'uscita è funzione soltanto dello stato interno presente: $B: S \rightarrow Z$

- Il **Modello di Mealy** adotta una legge B decisamente più flessibile: $B: S \times X \rightarrow Z$. Questa legge indica che l'uscita è funzione non solo dello stato interno presente ma anche degli ingressi presenti.



Osserviamo che gli ingressi delle reti **RCA** ed **RCB** sono gli stessi (sia relativamente alle variabili x_k che alle variabili y_j. Segue che un rete in modello di Mealy possa essere disegnata anche nel seguente modo:



- Leggi di temporizzazione: le leggi di temporizzazione sono identiche a quelle già viste. Ricapitoliamo le disuguaglianze viste fino ad oggi:

> - **Disuguaglianze definitive**:
>   - **Percorso da ingresso a registro**: $T \geq T_{sfas} + T_{\text{hold}} + T_{\text{a monte}} + T_{\text{in_to_reg}} + T_{\text{setup}}$
>   - **Percorso da registro a registro**: $T \geq T_{sfas} + T_{\text{prop}}' + T_{\text{reg_to_reg}} + T_{\text{setup}}$
>   - **Percorso da ingresso a uscita**: $T \geq T_{sfas} + T_{\text{hold}} + T_{\text{a monte}} + T_{\text{in_to_out}} + T_{\text{a valle}}$
>   - **Percorso da registro a uscita**: $T \geq T_{sfas} + T_{\text{prop}}' + T_{\text{reg_to_out}} + T_{\text{a valle}}$
>
> T_{sfas} è molto piccolo, quindi lo considereremo nullo d'ora in poi.

Disuguaglianze di temporizzazione generiche

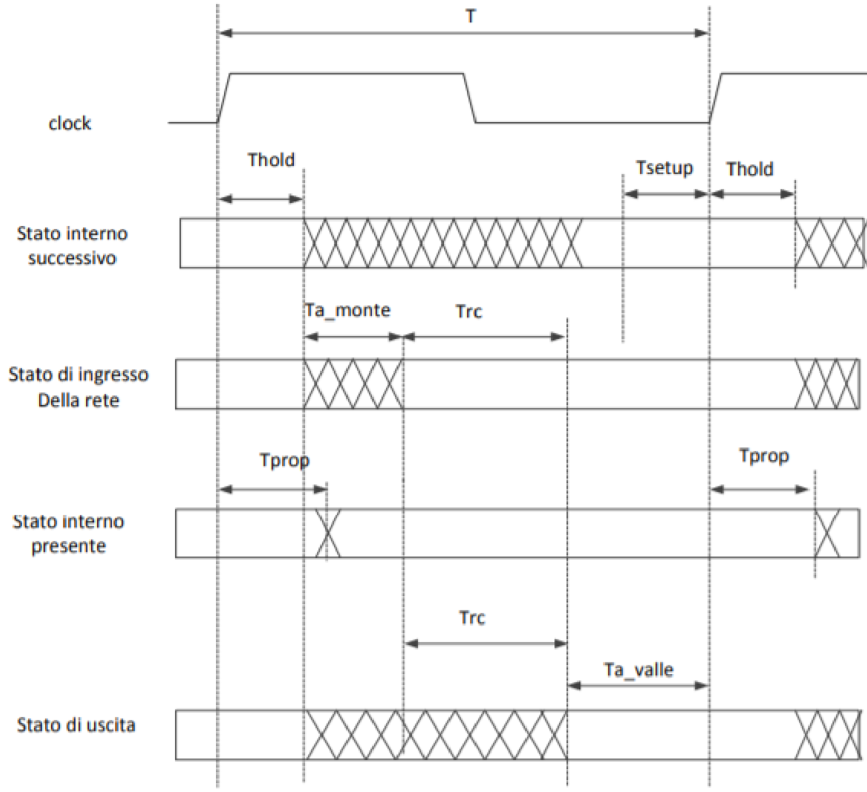
Scriviamo le leggi:
1. **Percorso da ingresso a registro**: $T \geq T_{\text{hold}} + T_{\text{a monte}} + T_{RC} + T_{\text{setup}}$
2. **Percorso da registro a registro**: $T \geq T_{\text{prop}} + T_{\text{RC}} + T_{\text{setup}}$
3. **Percorso da ingresso a uscita**: $T \geq T_{\text{hold}} + T_{\text{a monte}} + T_{\text{RC}} + T_{\text{a valle}}$
4. **Percorso da registro a uscita**: $T \geq T_{\text{prop}} + T_{\text{RC}} + T_{\text{a valle}}$

Abbiamo nuovamente una disuguaglianza relativa al passaggio da ingresso a uscita.

> - Tenendo conto di tutte queste cose otteniamo le seguenti disuguaglianze di temporizzazione:
>   - **Da ingresso a STAR**: $T \geq T_{\text{hold}} + T_{\text{a monte}} + T_A + T_{\text{setup}}$
>   - **Da STAR a STAR**: $T \geq T_{prop} + T_A + T_{\text{setup}}$
>   - **Da STAR a uscita**: $T \geq T_{prop} + T_Z + T_{a\,valle}$

Leggi viste con Moore

- La seconda disuguaglianza è implicata dalla prima: stessi elementi tranne $T_{\text{a monte}}$ mentre $T_{\text{hold}} \approx T_{\text{prop}}$.
- La quarta è implicata dalla terza per gli stessi motivi della precedente implicazione.
- La disequazione più vincolate è certamente la terza, dove abbiamo $T_{\text{a monte}}, T_{\text{RC}}, T_{\text{a valle}}$ insieme (cosa che non avviene nella rete di Moore).
- Se si confronta la disequazione più vincolante per Mealy con quella più vincolante per Moore individuiamo che il clock in Mealy debba andare più lentamente (ovviamente si dice tutto questo a parità di temporizzazione).
- Non si distinguono i tempi tra i diversi percorsi in RC: il tempo di attraversamento è sempre T_{RC}. Se la rete combinatoria è sintetizzata in modo ottimizzato i tempi di attraversamento dovrebbero essere più o meno uguali tra ogni coppia di morsetti.



- **Descrizione in Verilog**:

```
module Rete_di_Mealy(zM-1,...,z0,xN-1,...,x0,clock,reset_);
 input clock,reset_;
 input xN-1,...,x0;
 output zM-1,...,z0;

 reg [W-1:0] STAR; parameter S0=codifica0,...,SK-1=codificaK-1;

 assign {zM-1,...,z0} = (STAR==S0)? ZS0(xN-1,...,x0) :
                          ...
                          ...
                        /*(STAR==SK-1)*/ ZSK-1(xN-1,...,x0);
 always @(reset_==0) #1 STAR <= stato_interno_iniziale;
 always @(posedge clock) if (reset_==1) #3
   casex(STAR)
     S0 : STAR<=AS0(xN-1,...,x0);
     ...
     ...
     ...
     SK-1 : STAR<=ASK-1(xN-1,...,x0);
```

```
    endcase
endmodule
```

```
assign {zM-1,...,z0} =   (STAR==S0) ?zS0 :
                         (STAR==S1) ?zS1 :
                         ...
               /*(STAR==SK-1)*/ zSK-1;
```

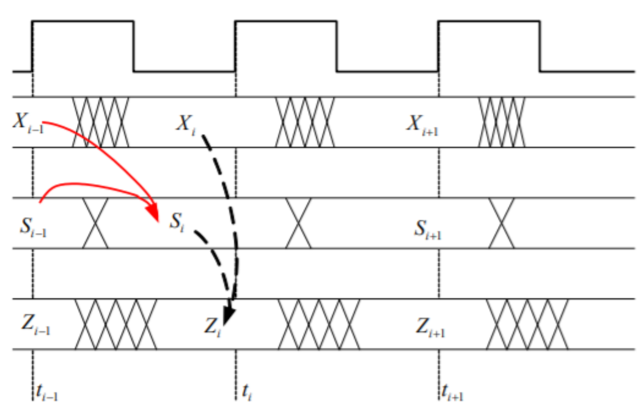
Legge B

Valore di uscita in una rete di Moore. Mealy si differenzia col fatto che i valori Z non dipendono solo dallo stato interno, ma anche dalle variabili di ingresso

- **Evoluzione nel tempo di una RSS di Mealy**:
  - Una rete di Mealy, differentemente da una rete con modello di Moore, produce un nuovo stato di uscita al variare dell'ingresso senza dover aspettare il successivo posedge del clock.
  - In Moore l'uscita varia quando arriva il clock (si parla in fatti di uscita è in ritardo di un clock rispetto all'ingresso che l'ha generata)
  - In Mealy l'uscita dipende anche dall'ultimo stato di ingresso, quello presente al clock attuale.

$$S[t_{i+1}] = A(X[t_i], S[t_i])$$
$$Z[t_i] = B(X[t_i], S[t_i])$$

Confrontare con la temporizzazione di una rete di Moore, dove è: $Z[t_i] = B(S[t_i])$.



Si dice che

- nelle reti di Moore, **l'uscita è un clock in ritardo rispetto all'ingresso che l'ha generata.** Dipende, infatti, soltanto dal *penultimo* stato di ingresso, quello al clock precedente. Infatti, è:

  $$Z[t_i] \propto X[t_{i-1}], X[t_{i-2}], ..., X[t_0], S[t_0]$$

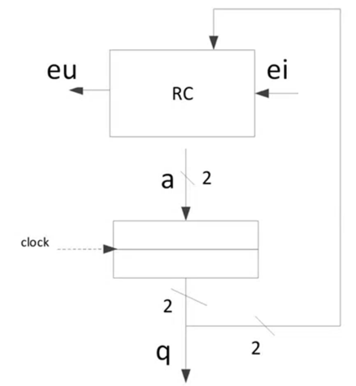
- nelle reti di Mealy, **l'uscita dipende anche dall'ultimo stato di ingresso, quello presente al clock attuale.**

  $$Z[t_i] \propto X[t_i], X[t_{i-1}], X[t_{i-2}], ..., X[t_0], S[t_0]$$

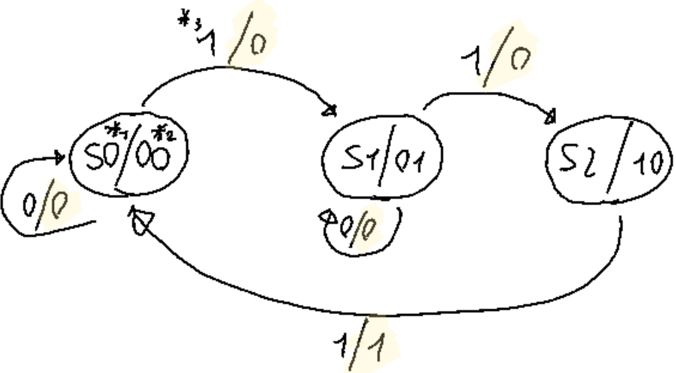
La flessibilità della legge B permette di risolvere gli stessi problemi con un numero minore di stati interni.

Esempio: contatore espandibile in base 3

- Un contatore espandibile in base 3, già incontrato precedentemente, è una rete di Mealy:
    - Abbiamo una legge $A: X \times S \rightarrow S$ con cui otteniamo un nuovo stato interno. Il nuovo stato interno si ottiene considerando quello precedente e i valori in ingresso (nel caso del contatore la variabile e_i).
    - Abbiamo una legge $B: X \times S \rightarrow Z$ che calcola l'uscita q, e_u. L'uscita dipende non solo dallo stato interno ma anche dagli ingressi. Contrariamente a Moore non si attende il clock successivo per esprimere una nuova uscita.
    - È una rete con tre stati interni: cifre 0, 1, 2 in base 3.
    - Non esistono vie combinatorie tra l'ingresso e_i e q



- **Grafo di flusso**: il grafo di flusso presenta delle differenze. Nei grafi di flusso relativi alle reti di Moore abbiamo sempre scritto l'uscita accanto allo stato (poiché l'uscita è funzione solo dello stato interno in quelle reti). In questo caso abbiamo la legge $B: X \times S \rightarrow Z$: questo significa che non possiamo scrivere più le uscite accanto agli stati poiché queste dipendono anche dagli ingressi. Scriveremo i valori delle uscite sugli archi! Nell'esempio del contatore l'uscita di Moore è direttamente lo stato del registro, quindi scriveremo accanto agli stati le codifiche adottate per gli stati del registro.



- EVIDENZIATE LE USCITE e_u AGGIORNATE

*1 STATO INTERNO

*2 CODIFICA CHE VA IN INGRESSO NELLA RETE COMBINATORIA

*3 INGRESSO e_i

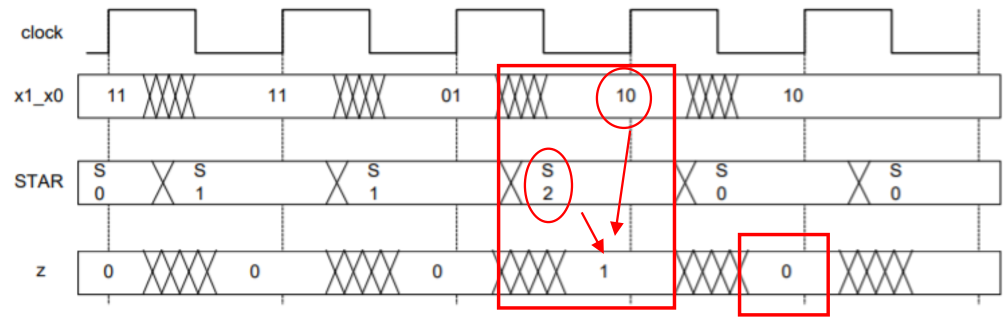
- **Tabelle di flusso**: la tabella di flusso può essere scritta così

	e_i \ q1q0 e_u	0	1
(00)	S0	S0 00 0	S1 00 0
(01)	S1	S1 01 0	S2 01 0
(10)	S2	S2 10 0	S0 10 1

e_i \ e_u	0	1	q1q0
S0	S0 0	S1 0	00
S1	S1 0	S2 0	01
S2	S2 0	S0 1	10

Esempio (di nuovo): riconoscitore di sequenza 11, 01, 10

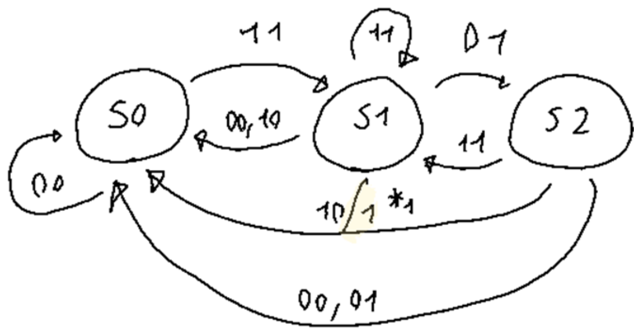
- Prima di passare alla sintetizzazione come rete di Mealy vediamo la temporizzazione



Non ho l'1 qua come in Moore

  - Siamo nello stato S0 e ho in ingresso 11: passo allo stato interno S1
  - Ovviamente se siamo in S1 e fornisco 11 rimango in S1: potrei iniziare una nuova sequenza
  - Se siamo in S1 e ho in ingresso 01 passo in S2
  - Se siamo in S2 e ho in ingresso 11 **non aspetto il clock successivo** come facevo in Moore per restituire z uguale ad 1: lo posso fare subito!

- Non ci servono K+1 stati, cioè quattro stati, ma solo 3! Vediamo il **grafo di flusso**:



*1 PER CHIAREZZA INDICO SOLO QUA, GLI ALTRI SI DA PER SCONTATO CHE L'USCITA SARÀ ZERO.

- **Ecco la tabella di flusso**:

x_1x_0	00	01	11	10
S0	S0/0	S0/0	S1/0	S0/0
S1	S0/0	S2/0	S1/0	S0/0
S2	S0/0	S0/0	S1/0	S0/1

Ovviamente almeno una riga dovrà presentare qualcosa di diverso, altrimenti stiamo facendo una sintesi in Moore

- **Attenzione**: questi due formalismi non mostrano le temporizzazioni (non mi dicono che la legge A è procedurale e di assegnamento al registro mentre la legge B è combinatoria, questa è una cosa che ci teniamo nel capo noi). La cosa sarà chiara, invece, quando andremo a scrivere la descrizione in Verilog (le leggi combinatorie si scrivono in un modo, le leggi procedurali in un altro.

- **Descrizione in Verilog**:

```
module Riconoscitore_di_Sequenze(z,x1_x0,clock,reset_);
  input clock,reset_;
  input [1:0] x1_x0;
  output z;

  reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;

  assign z=((STAR==S2) & (x1_x0=='B10))?1:0;

  always @(reset_==0) #1 STAR<=S0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(x1_x0=='B11)?S1:S0;
      S1: STAR<=(x1_x0=='B01)?S2:(x1_x0=='B11)?S1:S0;
      S2: STAR<=(x1_x0=='B11)?S1:S0;
    endcase
endmodule
```

Differenza principale: per determinare l'uscita z considero anche gli ingressi.

Come al solito si considera quanto detto in grafo e tabella di flusso.

Differenze tra reti di Mealy e Reti di Moore

Reti di Moore	*Reti di Mealy*

- Legge B meno flessibile
  - Più stati interni
	- Legge B più flessibile **(+)**
	  - Meno stati interni (registri più piccoli)
	  - Meno logica e complessità

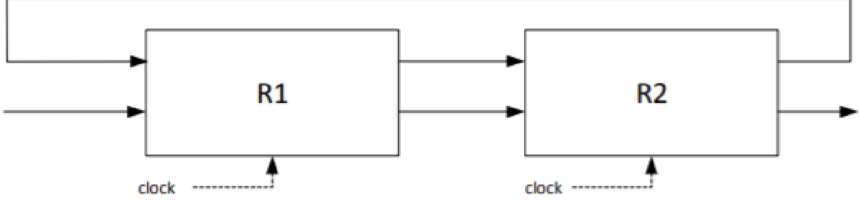
- La legge B è meno flessibile: possiamo fare con Moore tutte le cose che si possono fare con Mealy?	- Abbastanza scontato che con una legge B più flessibile Mealy permette di fare tutte le cose che si possono fare con Moore.

Stessa potenza descrittiva!
C'è un teorema che ci garantisce che dato un problema risolto con Moore si trova una rete di Mealy che lo risolve, e viceversa.

- Clock più veloce **(+)** - Uscita un clock in ritardo rispetto agli ingressi.	- Clock più lento - Uscita in pari con gli ingressi **(+ ???, wait)**

Attenzione all'elefante nella stanza (cit.Stea, ispirato dagli inglesi): le reti di Moore sono non trasparenti, le reti di Mealy SONO TRASPARENTI.

Supponiamo di avere la seguente rete



R1 ed R2 sono reti sequenziali sincronizzate generiche. Se entrambe sono di Mealy si va a creare un anello combinatorio, quindi una rete sequenziale asincrona. Se una delle due è una rete di Moore allora non ci sono problemi (questo garantisce che ci sia almeno un registro nella rete).

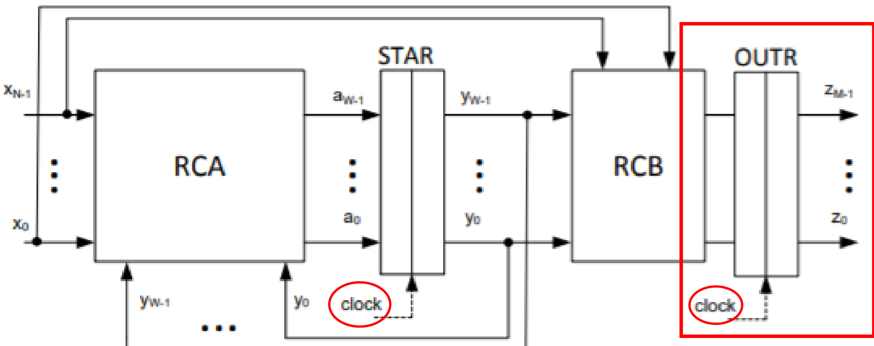
Presente alle pagg.69-70 della dispensa di Stea un esercizio sul modello di Mealy
(Sintesi di una RSS in cui si confronta lo stato attuale – gli ingressi – con lo stato precedente – STAR)

Capitolo 36

Martedì 17/11/2020

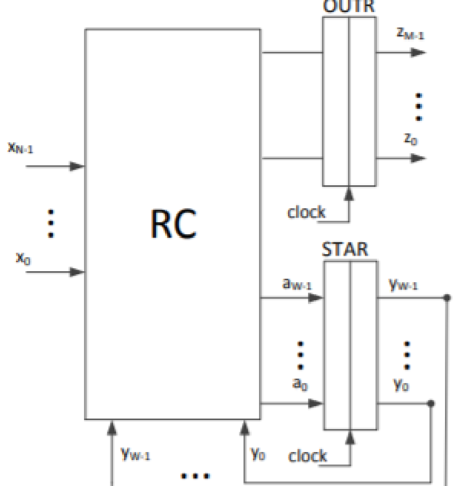
Modello di Mealy ritardato

- Il **modello di Mealy ritardato** si ottiene a partire da una rete di Mealy: si mette in uscita un altro registro OUTR con lo stesso clock di STAR.



  - Le uscite variano sempre all'arrivo del clock dopo il solito tempo T_{prop}
  - Le uscite variano in maniera netta senza oscillazioni (sparisce il problema dovuto alla trasparenza della classica rete di Mealy)
  - Le uscite rimangono stabili per l'intero ciclo di clock.
  - Le uscite sono non trasparenti.
  - I registri STAR e OUTR hanno lo stesso clock ma funzione completamente diversa:
    - STAR gestisce l'evoluzione dello stato interno della rete
    - OUTR temporizza le uscite, cioè le ritarda di un clock.

- Anche in una rete di Mealy ritardata possiamo collassare le reti RCA ed RCB (stessi ingressi):



- Le leggi di evoluzione nel tempo non cambiano in Mealy ritardato:
  - Una legge del tipo $A: X \times S \rightarrow S$ con cui si mappano coppie (stato di ingresso, stato interno) in un nuovo stato interno.
  - Una legge del tipo $B: X \times S \rightarrow Z$ con cui si mappano coppie (stato di ingresso, stato interno) in un nuovo stato di uscita.

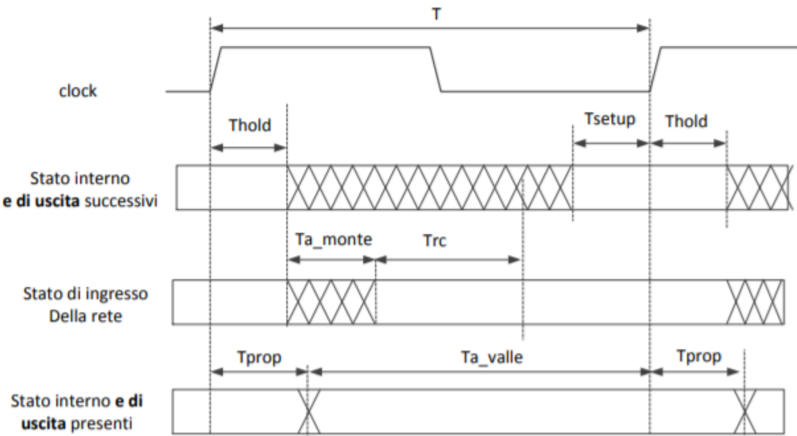
  Tuttavia in Mealy ritardato non ho la prima legge di tipo procedurale e la seconda di tipo combinatorio: sono entrambe procedurali (entrambe si occupano di aggiornare il contenuto di un registro)! Sia il nuovo stato interno, sia il nuovo stato di uscita, vengono marcati dopo l'arrivo del clock e dipendono dallo stato di ingresso e dallo stato interno PRIMA del clock.

> Lo stato di uscita cambia dopo il clock, ed il suo valore dipende dallo stato di ingresso e dallo stato interno marcato **precedenti all'arrivo del clock**.

- La differenza sostanziale tra reti di Mealy e reti di Mealy ritardate sta nella legge di temporizzazione: *Dato S, stato interno presente (marcato) ad un certo istante, e dato X stato di ingresso ad un certo istante precedente l'arrivo di un segnale di sincronizzazione*
    - *Individuare SIA il nuovo stato interno da marcare $S' = A(S, X)$, sia il nuovo stato di uscita $Z = B(S, X)$*
    - *Attendere T_{prop} dopo l'arrivo del segnale di sincronizzazione*
    - *Promuovere S' al rango di stato interno marcato, e promuovere Z al rango di nuovo stato di uscita.*

- **Leggi di temporizzazione**:
    1. **Percorso da ingresso a registro**: $T \geq T_{\text{hold}} + T_{\text{a monte}} + T_{\text{RC}} + T_{\text{setup}}$
    2. **Percorso da registro a registro**: $T \geq T_{\text{prop}} + T_{\text{RC}} + T_{\text{setup}}$
    3. **Percorso da registro a uscita**: $T \geq T_{\text{prop}} + T_{\text{a valle}}$

  La seconda disuguaglianza è implicata dalla prima per motivazioni simili a quelle incontrate in passato, la prima disuguaglianza è quella più vincolante: $T_{\text{a monte}}, T_{\text{RC}}$ sono insieme in un'unica disuguaglianza.



- **Evoluzione del tempo di una RSS di Mealy ritardata**:

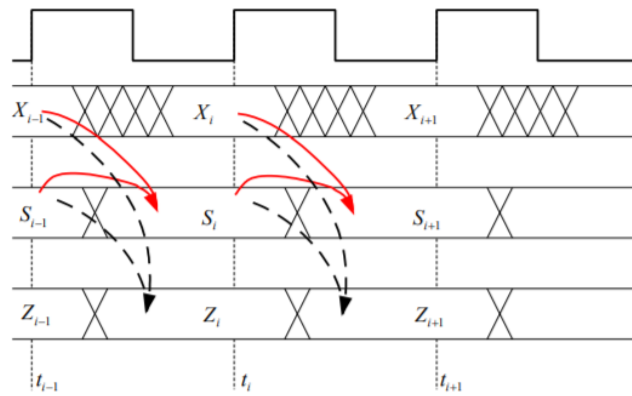
Moore: $Z[t_i] = B(S[t_i])$

Mealy: $Z[t_i] = B(X[t_i], S[t_i])$

$$S[t_{i+1}] = A(X[t_i], S[t_i])$$
$$Z[t_{i+1}] = B(X[t_i], S[t_i])$$

e tali nuovi stati interni e di uscita saranno resi disponibili **dopo** T_{prop}.

Confrontare con quelli per **reti di Moore e Mealy**



- Per lo stato interno valgono le stesse cose viste in Mealy non ritardato
- Per l'uscita abbiamo delle differenze: l'uscita non si adegua più in modo immediato ma si adegua con l'arrivo del clock. Segue che anche l'uscita, cioè il contenuto del registro OUTR aggiornato in istante di sincronizzazione t_i, dipenderà dallo stato interno e dagli ingressi legati all'istante di sincronizzazione precedente t_{i-1}.

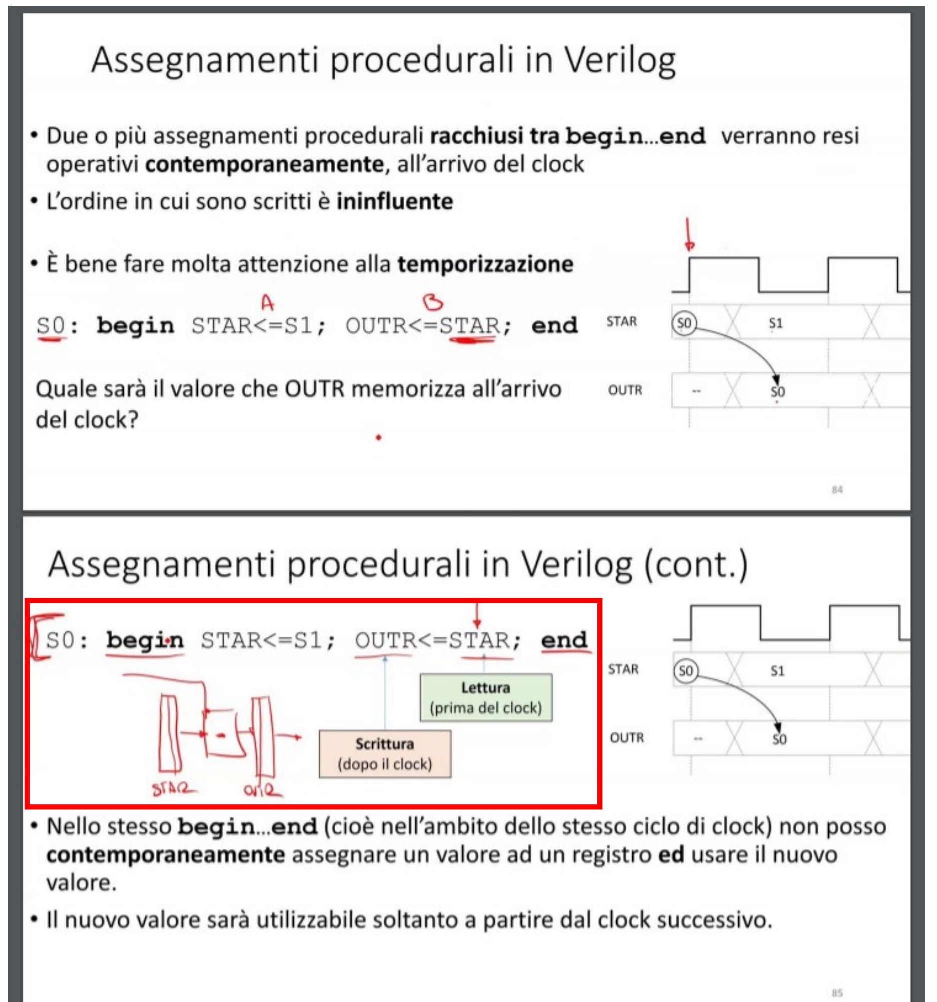
- **Descrizione in Verilog**: questa rete avrà più di un registro, e dobbiamo abituarci a questa cosa (andando avanti avremo sempre più registri all'interno delle nostre RSS)

```
module Rete_di_Mealy_Ritardato(zM-1,...,z0,xN-1,...,x0,clock,reset_);
  input clock,reset_;
  input xN-1,...,x0;
  output zM-1,...,z0;
```

```
    reg [W-1:0] STAR; parameter S0=codifica0,...,SK-1=codificaK-1;
    reg [M-1:0] OUTR; assign {zM-1,...,z0}=OUTR;
    always @(reset_==0) #1 begin OUTR<=...; STAR<=...; end
    always @(posedge clock) if (reset_==1) #3
      casex(STAR)
        S0  : begin OUTR<=ZS0(xN-1,...,x0); STAR<=AS0(xN-1,...,x0); end
        ...
        SK-1 : begin OUTR<=ZSK-1(xN-1,...,x0); STAR<=ASK-1(xN-1,...,x0); end
      endcase
endmodule
```

Le cose sono tutto sommato comprensibili, ma dobbiamo soffermarci sulle righe sottolineate: è ovvio che ad ogni istante di sincronizzazione dovremo aggiornare il contenuto sia del registro STAR che del registro OUTR.

- Nelle righe sottolineate abbiamo due assegnamenti procedurali: porre questi assegnamenti tra begin ed end significa stabilire che questi verranno resi operativi CONTEMPORANEAMENTE all'arrivo del clock.
- L'ordine di questi assegnamenti è insignificante: non stiamo lavorando in C++, non abbiamo una serie di statements eseguiti uno dopo l'altro.
- Dobbiamo inoltre tener conto del valore dei registri. Supponiamo di porre un secondo assegnamento dipendente dal primo all'interno della riga begin-end. Quando facciamo questo dobbiamo ricordarci che gli assegnamenti sono eseguiti in contemporanea: i valori presi sono quelli relativi all'istante di sincronizzazione precedente.
- **Per avere le idee più chiare vedere queste due diapositive**:



- **Tabelle e grafi di flusso**: grafi e tabelle di flusso, qualora sia possibile realizzarli in modo semplice, risulteranno uguali a quelli delle reti di Mealy. La cosa è valida sia per semanticamente che sintatticamente: la cosa non è casuale visto che la rete di Mealy ritardata si differenzia dalla rete di Mealy nella temporizzazione, non nelle leggi (cioè quello che cambia è solo il modo in cui rendo operativa una legge).

- **Differenze nelle descrizioni in Verilog**: le descrizioni in verilog, invece, cambiano. Supponiamo di prendere un riconoscitore di sequenze fatto secondo il modello di Mealy ritardato.

```
module Riconoscitore_di_Sequenze(z,x1_x0,clock,reset_);
  input clock,reset_;
  input [1:0] x1_x0;
  output z;

  reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;

  reg OUTR; assign z=OUTR;

  always @(reset_==0) #1 begin OUTR<=0; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: begin OUTR<=0; STAR<=(x1_x0=='B11)?S1:S0; end
      S1: begin OUTR<=0; STAR<=(x1_x0=='B01)?S2:(x1_x0=='B11)?S1:S0; end
      S2: begin OUTR<=(x1_x0=='B10)?1:0; STAR<=(x1_x0=='B11)?S1:S0; end
    endcase
endmodule
```

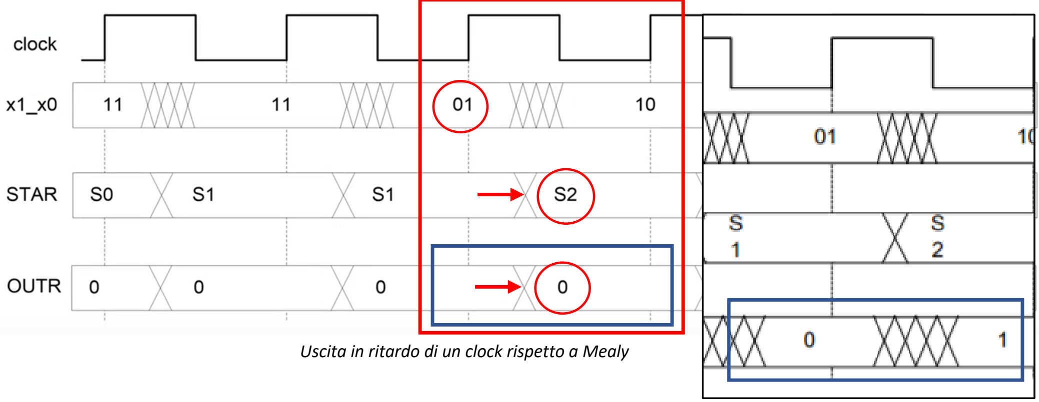
Parte relativa all'evoluzione del registro STAR identica.

Cambia la parte relativa all'evoluzione delle uscite: abbiamo assegnamenti procedurali al registro OUTR (invece che assegnamenti continui come in Mealy)

```
module Riconoscitore_di_Sequenze(z,x1_x0,clock,reset_);
  input clock,reset_;
  input [1:0] x1_x0;
  output z;
  reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;
  assign z=((STAR==S2) & (x1_x0=='B10))?1:0;

  always @(reset_==0) #1 STAR<=S0;
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: STAR<=(x1_x0=='B11)?S1:S0;
      S1: STAR<=(x1_x0=='B01)?S2:(x1_x0=='B11)?S1:S0;
      S2: STAR<=(x1_x0=='B11)?S1:S0;
    endcase
endmodule
```

Riconoscitore di sequenze in Mealy normale

- **Diagramma di temporizzazione relativo al riconoscitore di sequenze**:



Uscita in ritardo di un clock rispetto a Mealy

Frammento del diagramma di temporizzazione in Mealy normale.

Modello di Mealy ritardato (cont.)

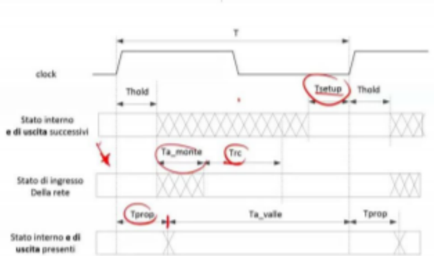
- Le reti di Mealy ritardato sono **non trasparenti**
  - Posso montarle in qualunque configurazione senza problemi
- Le uscite sono costanti per un intero periodo di clock
  - Posso montare **catene** di reti di Mealy ritardato con lo stesso clock **arbitrariamente lunghe**
  - Non avrò mai **problemi di temporizzazione**

89

Esempio

x1 N1 → RSS MR 1 → z1 M1 → RSS MR 2 → M2 z2 (clock su entrambe)

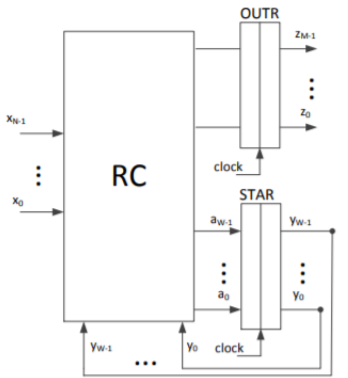
- Lo stato di ingresso di MR2 (stato di uscita di MR1) deve essere pronto $T_{RC_2} + T_{setup}$ prima del clock.
- Lo stato di uscita di MR1 è pronto **già** T_{prop} **dopo il fronte del clock.**
- Basta che $T \geq T_{prop} + T_{RC_2} + T_{setup}$ perché MR1 possa pilotare MR2
- Ma **questa disuguaglianza è vera**
  - è infatti quella che regola il percorso reg->reg in MR2



- **Ricapitoliamo, le reti di Mealy ritardato**:
  - Sono non trasparenti
  - Hanno una legge B flessibile che permette la risoluzione di problemi con meno stati interni
  - Hanno uscite stabili che cambiano in tempi certi
  - Non sono rallentate da percorsi combinatori troppo lunghi
  - Possono essere montate in cascata senza problemi di pilotaggio
  - Possono essere montate in reazione senza problemi di stabilità.

RSS complesse

- I tre modelli visti fino ad ora vanno bene solo per fare cose molto semplici.
- La pulizia concettuale adottata fino ad ora (cioè l'utilizzo di una struttura rigida con certe reti combinatorie, certi registri e certi ingressi) diventa un limite.
- Cerchiamo di superare questo problema partendo dal modello di Mealy ritardato, che ha molte caratteristiche positive.

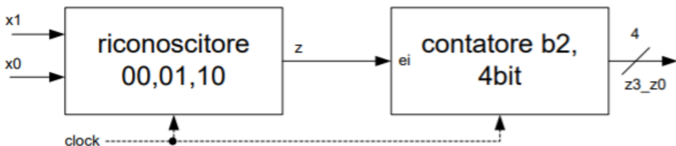


<u>Contatore di sequenze corrette 00, 01, 10</u>

- Supponiamo di voler descrivere una rete, usando il modello MR, che conta (modulo 16), il numero di sequenze corrette 00, 01, 10 ricevute in ingresso.

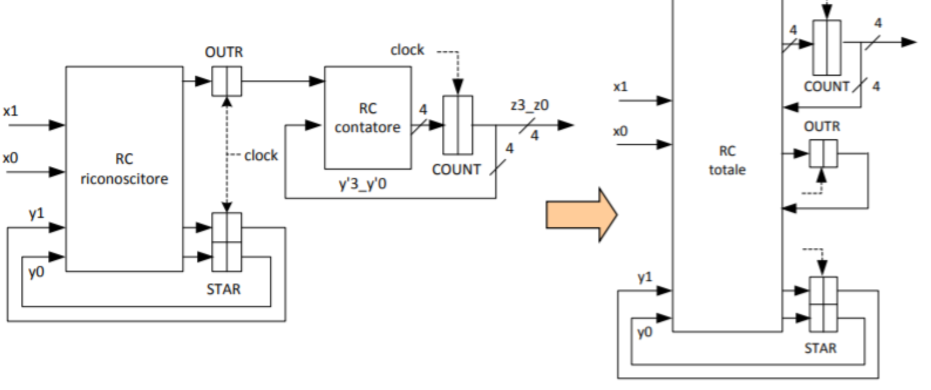
- Ogni volta che si vede una sequenza corretta si incrementa di 1 il valore in uscita. Poiché si parla di modulo 16 avrò bisogno di 4 bit in uscita.
- La rete presenta due ingressi (che mi permettono di inserire gli elementi della sequenza), quattro uscite (quelle che abbiamo detto prima) e un certo numero di stati interni. **<u>QUANTI?</u>**
  - Voglio cambiare il valore in uscita ogni tre stati interni.
  - Tenendo conto che vogliamo conteggiare al più 16 sequenze corrette, avrò bisogno di $16 \cdot 3 = 48$ stati interni. Ho bisogno di un registro STAR a 6 bit.
  - Non è difficile intuire che è ingestibile descrivere e sintetizzare una rete del genere (`casex…endcase` a quarantotto righe in Verilog)

- **Soluzione alternativa**: risulta molto più facile



  - Sintetizzare un riconoscitore di una sequenza come rete di MR, con un bit di uscita, e…
  - Sintetizzare un contatore a 4 in base 2, che prende come ingresso ei (riporto entrante) l'uscita del riconoscitore, e produce esso stesso un'uscita su 4 bit. Questo contatore, se non considero il riporto uscente, è una rete di Moore. Posso immaginarlo, volendo, anche come una rete di Mealy ritardato (l'uscita che rappresenta il numero in base 2 su 4 bit è supportata direttamente da un registro).

  Spogliamo la rete:



- Complessivamente non otteniamo una rete di Mealy ritardato:
  - **Differenza di lana caprina**: numero maggiore di registri
  - **Differenza vera**: contrariamente a Mealy ritardato non ho un solo registro con valore rientrante nella RC totale (STAR), ma anche altri due registri (OUTR e COUNT)
- Questa differenza è molto comoda: ho registri che supportano sia le uscite che computazioni intermedie: questo rende le descrizioni molto più semplici e nel nostro esempio permette di ridurre il numero di stati interni da 48 a 3.

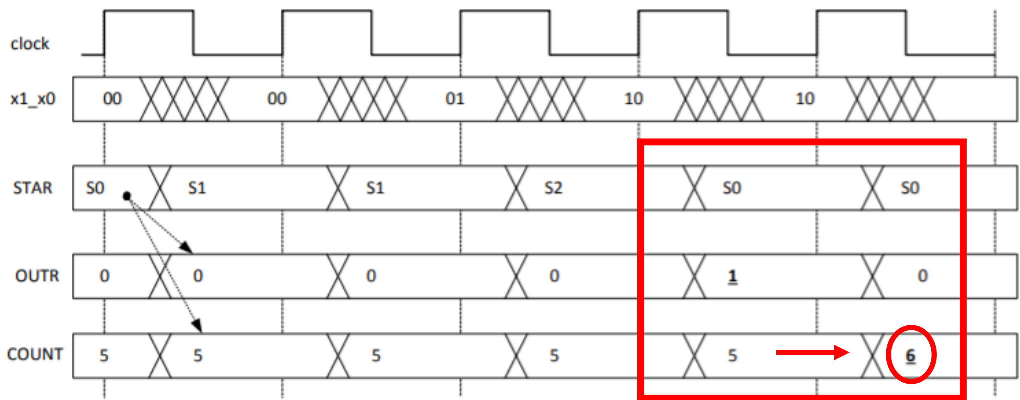
- **Descrizione in Verilog**:

```
module Contatore_Sequenze(z3_z0, x1_x0, clock, reset_)
  input clock, reset_;
  input [1:0] x1_x0;
  output [3:0] z3_z0;

  reg [3:0] COUNT; assign z3_z0=COUNT;

  reg [1:0] STAR; parameter S0='B00, S1='B01, S2='B10;
  reg OUTR;

  always @(reset_ ==0) #1 begin OUTR<=0; COUNT<=0; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0 : begin OUTR<=0; COUNT<=COUNT+OUTR; STAR<=(x1_x0=='B00)?S1:S0; end
      S1 : begin OUTR<=0; COUNT<=COUNT+OUTR;
      STAR<=(x1_x0=='B01)?S2:(x1_x0==’B00)?S1:S0; end
      S2 : begin OUTR<=(x1_x0=='B10)?1:0; COUNT<=COUNT+OUTR;
      STAR<=(x1_x0=='B00)?S1:S0; end
    endcase
endmodule
```

Attenzione al registro COUNT per la RC contatore.
Ricordarsi quanto detto sulle istruzioni poste dentro begin…end

Attenzione all’aggiornamento del registro COUNT

- **Fronte di temporizzazione**:
  Osserviamo cosa succede nell’evoluzione temporale della rete



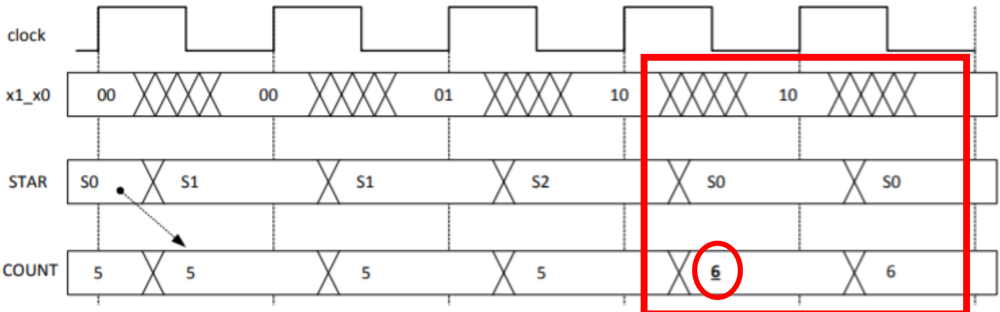
COUNT incrementa di un clock in ritardo rispetto alla sequenza degli ingressi riconosciuta

Questa cosa può essere dedotta dalla descrizione in Verilog: ricordiamo che negli assegnamenti si considerano sempre i valori dei registri memorizzati all’istante di sincronizzazione precedente.
Questa cosa non ci interessa: vogliamo che l’uscita si aggiorni un clock prima! Riscriviamo una parte della descrizione:

```
always @(reset_ ==0) #1 begin OUTR<=0; COUNT<=0; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0 : begin COUNT<=COUNT; STAR<=(x1_x0=='B00)?S1:S0; end
      S1 : begin COUNT<=COUNT;
      STAR<=(x1_x0=='B01)?S2:(x1_x0==’B00)?S1:S0; end
```

```
    S2 : begin COUNT<=(x1_x0 == 'B10)? COUNT+1:COUNT;
    STAR<=(x1_x0=='B00)?S1:S0; end
  endcase
```

Ci siamo sbarazzati del registro OUTR: segue un registro in meno e aggiornamento più veloce. Vediamo il nuovo grafico di temporizzazione



Modello più generale

- A partire dall'esempio precedente abbiamo elaborato un nuovo modello per realizzare reti.
- In questo modello sono presenti i seguenti registri:
  - **Registro STAR**, che svolge le solite funzioni (marcatura degli stati interni)
  - **Registri operativi**:
    - Tanti quanti me ne servono
    - Il loro contenuto può essere usato per le computazioni all'interno della RC totale (un grosso miglioramento)
- Le uscite sono tutte sostenute da registri operativi come nel modello di Mealy ritardato. Non per forza un registro operativo deve sostenere un'uscita.

Nomenclatura per la descrizione dell'ultima rete fatta

- Si parla di Linguaggio di trasferimento tra registri
- Osservazione: il Verilog non è un linguaggio di trasferimento tra registri, ma COMPRENDE questo linguaggio (tra tante cose). Il termine linguaggio non deve farci pensare al software: con linguaggio intendiamo descrizione di hardware, non un linguaggio di programmazione.

> Il prefisso μ- sta per hardware, non per piccolo!!!

- Ogni ramo del casex è detto statement (purtroppo, cit.) e comprende
  - Zero o più μ –istruzioni, cioè assegnamenti a registri operativi
  - Un μ –salto, cioè un assegnamento al registro STAR. Abbiamo vari tipi di μ –salto:
    - A tre vie: `STAR <= (x1_x0=='B01) ? S2 : (x1_x0=='BOO) ? S1 : S0;`
    - A due vie: `COUNT <= (x1_x0=='B10) ? COUNT+1 : COUNT;`
    - A una via (incondizionato): `STAR <= S2`
- Supponendo di avere Q registri operativi ogni statement avrà Q operazioni di assegnamento. Sono possibili deroghe?
  - Posso omettere assegnamenti relativi a registri operativi. Omettere il comportamento del registro significa sottintendere
    $$REGISTRO <= REGISTRO$$
    Ricordarsi che omettere il comportamento non significa non avere il clock: il clock c'è SEMPRE!
  - La stessa cosa non si può fare col registro STAR. È palese che sottintendere una cosa del genere
    $$STAR <= STAR$$
    significa mandare in deadlock la rete, visto che manterrà in eterno lo stesso stato interno.

Vincoli di temporizzazione

- Le disuguaglianze che devono essere vere sono le stesse di Mealy ritardato. Gli spostamenti sono i soliti:
  - da ingresso a registro;
  - da registro a registro;
  - da registro a uscita.
- Ribadiamo che nei diagrammi di temporizzazione lo stato di tutti i registri cambia in modo **SINCRONIZZATO** all'arrivo del clock.

Capitolo 37

Mercoledì 18/11/2020

Esercizio: contatore di sequenze alternate 00,01,10-11,01,10

- L'esercizio consiste in una complicazione di quanto già fatto per introdurre le RSS complesse.
- Consideriamo due sequenze di tre passi: 00,01,10 e 11,01,10.
- Vogliamo descrivere una rete che incrementa un contatore a 4 bit
  - Quando riconosce la prima delle due sequenze,
  - Poi quando vede la seconda,
  - Poi di nuovo la prima, e così via in modo alternato.

- Il dettaglio interessante è che le due sequenze differiscono per un solo elemento: la cosa può semplificarci il lavoro. Indipendentemente da questo l'esercizio può essere risolto in modo agile pure con sequenze arbitrarie ed elementi completamente diversi tra loro (FARE PER CASA).

- Ripensiamo a quanto fatto per l'esercizio precedente: abbiamo il registro COUNT, che contiene il numero di sequenze riconosciute! Individuiamo che:
  - Con COUNT pari la prima sequenza da riconoscere è $x_1x_0 = 00$
  - Con COUNT pari la prima sequenza da riconoscere è $x_1x_0 = 11$

  Possiamo creare una rete combinatoria che mi verifica se la sequenza è riconosciuta

COUNT[0], x_1, x_0	match
000	1
111	1
Others	0

- **Descrizione in Verilog**: otteniamo la descrizione in Verilog partendo da quella dell'esercizio precedente

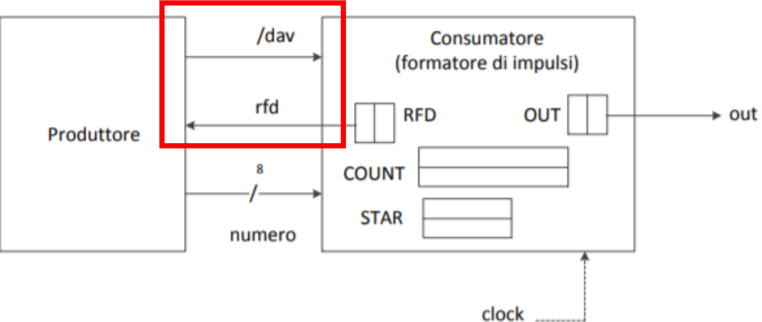
```
module Riconoscitore_e_Contatore(z3_z0,x1_x0,clock,reset_);
  input clock,reset_;
  input [1:0] x1_x0;
  output [3:0] z3_z0;

  reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;
  reg [3:0] COUNT; assign z3_z0=COUNT; // Registro operativo

  always @(reset_==0) #1 begin COUNT<='B0000; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
    casex(STAR)
      S0: begin COUNT<=COUNT; STAR<=(match(COUNT[0],x1_x0)==1)?S1:S0; end
      S1: begin COUNT<=COUNT;
            STAR<=(x1_x0=='B01)?S2:(match(COUNT[0],x1_x0)==1)?S1:S0; end
      S2: begin COUNT<=(x1_x0=='B10)? COUNT+1 : COUNT;
            STAR<=(match(COUNT[0],x1 x0)==1)?S1:S0; end
    endcase

  function match;
    input tipo sequenza;
    input [1:0] x1 x0;

    casex({tipo sequenza,x1 x0})
      'B000: match=1;
      'B111: match=1;
      default: match=0;
    endcase
  endfunction

endmodule
```

Modifichiamo le espressioni condizionali introducendo la function `match`. La utilizziamo:

- Nello stato S0 per capire quale elemento tra le due sequenze vogliamo riconoscere
- Negli stati rimanenti quando non poniamo elementi coincidenti con le sequenze. In base al `COUNT` e ai valori in ingresso decido se tornare/rimanere in S0 o in S1

Esempio: formatore di impulsi con handshake /dav-rfd

- Supponiamo di avere due RSS mutuamente asincrone, cioè due reti che
  - si evolvono l'una in modo indipendente dall'altra, e
  - non presentano alcuna forma di sincronizzazione comune (il clock può essere diverso)



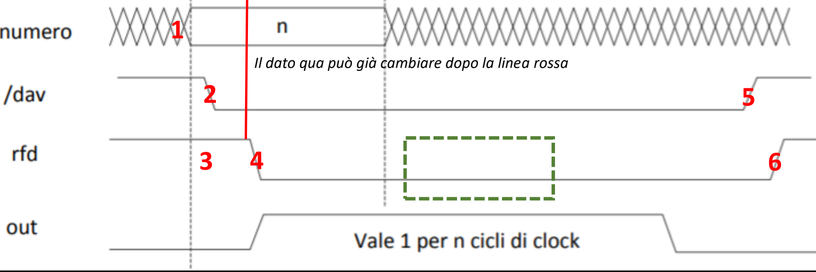
- **Funzionalità delle due RSS**:
  - Una è chiamata **produttore**: produce ogni tanto dei numeri naturali in base 2 su 8bit.
    Non è la rete che ci interessa in questo esercizio, daremo per scontato che sia già fatta.
  - Un'altra è detta **consumatore**: riceve in ingresso i numeri naturali restituiti in uscita dalla RSS produttore. Vogliamo sintetizzare questa rete!
    - La rete in questione è un formatore di impulsi: l'uscita è una singola variabile logica `out` che può assumere come valore zero o uno.
    - Dato un numero in ingresso, la rete tiene la linea di uscita a 1 per il numero di clock specificato nel numero.
    - **Caso particolare**: se il numero in ingresso vale 0 la RSS terrà l'uscita ad 1 per 256 cicli di clock.

- **Abbiamo dei problemi**, derivanti dalla mutua asincronicità delle RSS:
  - Come fanno le due reti a sincronizzarsi tra di loro?
  - Come fa il consumatore a sapere che c'è un dato nuovo in ingresso? Se il produttore volesse dare in ingresso due volte lo stesso numero, come fa il consumatore a distinguere che sono due numeri diversi?
  - Come fa il produttore a essere sicuro che il consumatore ha memorizzato il numero, visto che potrebbe in teoria anche essere molto più lento (NOI NON POSSIAMO FARE IPOTESI)?

- **Soluzioni a questi problemi**: introduzione delle linee di handshake.
  - L'**handshake** si ottiene attraverso due fili:
    - Uno dal produttore al consumatore detto `/dav` (*data valid*, un attivo basso).
      Il produttore segnala al consumatore che è in arrivo un nuovo dato.
    - Uno dal consumatore al produttore detto `rfd` (*ready for data*, un attivo alto)
      Il consumatore riferisce al produttore se è capace o incapace di accettare nuovi dati.

- **Transizioni alternate**: le transizioni dei due fili di *handshake* DEVONO essere alternate, altrimenti si creano situazioni di deadlock. Se non faccio le cose a modo potrei trovarmi in una situazione in cui il produttore attende il consumatore, e viceversa (situazione risolvibile solo facendo reset).

- Prendiamo il diagramma di temporizzazione:



- Supponiamo che tutti i dispositivi parte del nostro problema siano connessi allo stesso circuito di reset (con entrambe le linee di handshake poste uguali ad 1)
    - Il produttore non manda dati (`/dav = 1`).
      Lo stato dei fili numero non è da considerarsi un dato valido.
    - Il consumatore è pronto a ricevere dati (`rfd=1`)
- Il primo che deve fare la mossa è il produttore, il quale, nell'ordine
    1. prepara il nuovo dato mettendolo sull'uscita `numero`, e (DOPO)
    2. abbassa `/dav` (segnala che i dati forniti dal produttore sono validi).
- Successivamente entra in scena il consumatore, che, nell'ordine
    3. Preleva il nuovo dato memorizzandolo da qualche parte, e (DOPO)
    4. Abbassa `rfd` (segnala che non è più disponibile ad accogliere nuovi dati in ingresso, deve lavorare con i dati appena ricevuti). Il fronte di discesa è il segnale che non è più necessario mantenere il dato in uscita (posso già cambiarlo dopo la linea rossa).

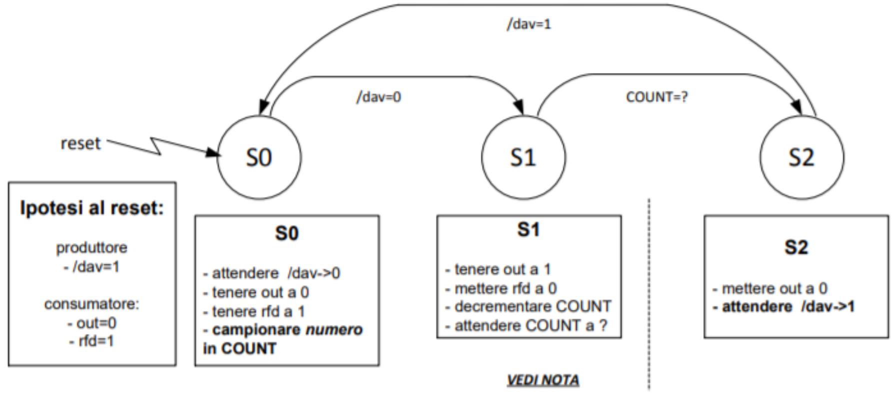
- L'handshake va avanti per **transizioni alternate**:
    5. Il produttore riporta su `/dav` quando vuole, in un qualunque istante successivo alla transizione 1-0 di `rfd`.
    6. Soltanto dopo aver riportato su `/dav` il consumatore può chiudere l'handshake riportando su `rfd`. L'handshake ritorna nella situazione di riposo in cui ci trovavamo all'inizio.

  Cosa succede se riportiamo su "`rfd`" nell'area tratteggiata in verde? Il problema è che non posso essere certo che il produttore si sia accorto del precedente fronte di discesa di `rfd`.
  Si va nella situazione di deadlock detta prima:
    - Il produttore potrebbe essere talmente lento da non aver visto i due fronti.
    - Agli occhi del produttore il consumatore non ha ancora prelevato i dati, quindi il produttore attende che il consumatore prelevi.
    - Il problema è che il consumatore ha già prelevato. Questo sta a sua volta attendendo che il produttore tiri su `/dav`.

- **Come fa l'uscita a rimanere ad 1 per un dato numero di cicli di clock?** Utilizzo
    - Un registro a 1 bit detto OUT che deve sostenere l'uscita. Gli assegniamo il valore 1 dopo aver letto il numero in ingresso.
    - Un registro contatore, COUNT, che inizializziamo ad un certo valore (parente di numero). Decrementiamo il valore ad ogni clock: quando arriviamo a 0 (o ad 1, vediamo), vorrà dire che sono passati un certo numero di cicli di clock. A quel punto metteremo nuovamente a 0 il valore di OUT. Questo registro si suppone sia ad 8 bit (ci ritorneremo su).

  Oltre a questo abbiamo
    - Un registro per ogni variabile di uscita, quindi al registro OUT aggiungiamo il registro RFD, sempre a 1 bit.
    - Il solito registro STAR che tiene traccia dell'evoluzione della rete. Anche sul numero di bit di STAR ritorneremo: la cosa dipende da come scrivo la descrizione.

- **Realizziamo un diagramma a stati:**



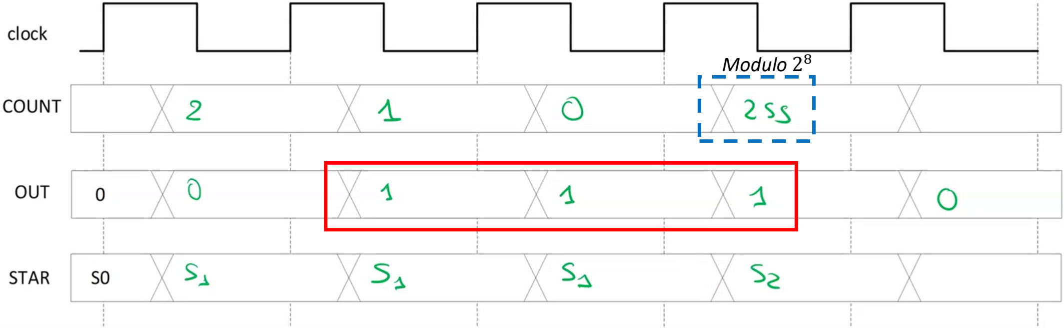
- La grafica è simile al diagramma di flusso, con la differenza che accompagniamo ogni stato interno con le sue caratteristiche. Oltre a questo indichiamo ipotesi al reset. Nell'immagine abbiamo riscritto tutte le proprietà spiegate. Inoltre:
    - Mi chiedo quando porre il nuovo valore del registro COUNT: chiaramente non posso fare assegnamento e decremento in contemporanea in S1. Segue che campiono numero in COUNT nello stato interno S0: il consumatore presenta $/rfd$ alzato, quindi non sta usando il valore salvato nel registro COUNT.
    - Ho bisogno per forza di uno stato interno S2: al termine del numero di clock indicati devo porre OUT uguale a 0, ma devo anche attendere che $/dav$ venga alzato. Fino ad allora non potrò tornare in S0.
    - Dopo aver definito il numero di stati interni posso dire il numero di bit necessari per il registro STAR: due bit!
    - Rimane il dubbio sul valore di COUNT necessario per passare dallo stato S1 allo stato S2.

- **Descrizione in Verilog**:

```
module Formatore_di_Impulsi(dav_,rfd,numero,out,clock,reset_);
 input clock,reset_;
 input dav_;
 output rfd;
 input [7:0] numero;
 output out;
 reg RFD; assign rfd=RFD;
 reg OUT; assign out=OUT;
 reg [7:0] COUNT;
 reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;

 always @(reset_==0) #1 begin RFD<=1; OUT<=0; STAR<=S0; end
 always @(posedge clock) if (reset_==1) #3
  casex(STAR)
   S0: begin RFD<=1; OUT<=0; COUNT<=numero; STAR<=(dav_==1)?S0:S1; end
   S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
   S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
  endcase
endmodule
```

Come arriviamo a dire che COUNT deve essere uguale ad 1 per passare da S1 a S2? Osserviamo cosa succede in un diagramma di temporizzazione (la temporizzazione delle uscite è cosa vitale in alcuni esercizi, sbagliare questa parte significa sbagliare completamente gli esercizi in questione).

Prima vediamo cosa succede ponendo come condizione COUNT == 0: attenzione ai valori posti, sono quelli restituiti dai registri prima del clock successivo (non i valori che andiamo a porre prima di un istante di sincronizzazione)

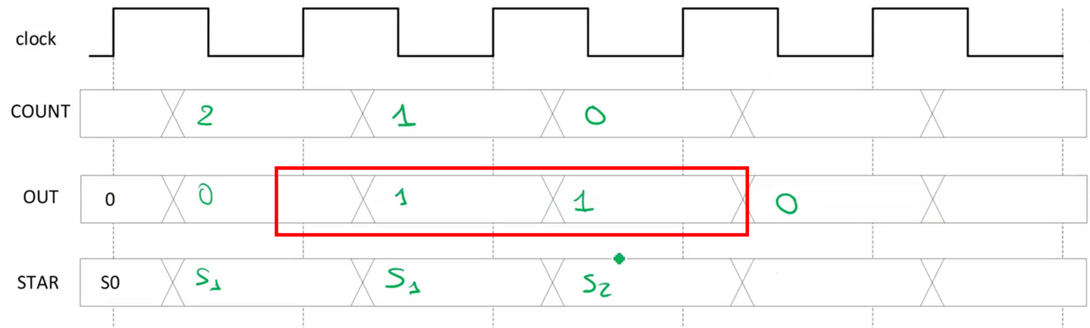


- Il valore di numero è 2. /dav è abbassato fin da subito.
- **Siamo in S0**: Poniamo OUT <= 0 e COUNT <= numero. Poiché /dav è abbassato passiamo ad S2.
- **Siamo in S1**: i valori di OUT e COUNT sono quelli determinati all'istante di sincronizzazione precedente. Rimaniamo in S1 poiché COUNT (quello dello step precedente) non è ancora uguale a 0, inoltre poniamo OUT = 1. Decrementiamo COUNT.

- **Siamo sempre in S1**: dopo il clock vediamo le modifiche ai registri fatte negli step precedenti (OUT e COUNT). Rimaniamo in S1 poiché COUNT (quello dello step precedente) non è ancora a 0.
- **Siamo sempre in S1**: stessi discorsi di prima. Tuttavia il COUNT ottenuto dallo step precedente è uguale a 0: decreto il passaggio da S1 ad S2. Decremento ancora, pur avendo COUNT = 0.
- **Siamo in S2**: vediamo il risultato del decremento all'istante precedente (in modulo 2^8), cioè 255. Solo quando saremo in S2 aggiorneremo OUT ponendolo uguale a 0: per il momento restituisce ancora 1. Dopo un ulteriore clock OUT restituirà 0.

  Numero di cicli in S1: 3. Non va bene, visto che abbiamo posto in ingresso 2.

- Cosa succede ponendo COUNT == 1?



Senza rispiegare gli step (il meccanismo è lo stesso) vediamo che il numero di cicli in S1 è 2, il numero da noi indicato. Concludiamo esprimendo la seguente regola generale:

Se:
- in uno stato S_{init} inizializzo un registro a k
- in uno stato S_{test} lo decremento e per uscire testo se il valore è pari a j ($\leq$ k),

il numero di cicli nello stato S_{test} è k-j + 1.

```
 S_init: begin … COUNT<=k; STAR<=S_test; end
→S_test: begin … COUNT<=COUNT-1; STAR<=(COUNT==j)?S_poi:S_test; end]  k-j+1
 S_poi : begin … end
```

```
always @(reset_==0) #1 begin RFD<=1; OUT<=0; STAR<=S0; end
 always @(posedge clock) if (reset_==1) #3
  casex(STAR)
   S0: begin RFD<=1; OUT<=0; COUNT<=numero; STAR<=(dav_==1)?S0:S1; end
   S1: begin RFD<=0; OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
   S2: begin RFD<=0; OUT<=0; STAR<=(dav_==1)?S0:S2; end
  endcase
```

Ulteriori osservazioni sulla descrizione:

- Possiamo omettere, come già detto, il comportamento di registri operativi. Per esempio non ho bisogno di specificare cosa faccia COUNT in S2 (non mi interessa il valore di COUNT). È come se avessi scritto `COUNT <= COUNT;`
- Potrei togliere, dal codice che abbiamo scritto, altre cose:
  - OUT <= 0 in S0 (valore nell'ipotesi di reset, e valore posto in S2)
  - RFD <= 0 in S2, visto che si arriva ad S2 solo passando da S1 e in S1 poniamo RFD <= 0;
- Potrei decidere di mettere RFD <= 0 in S2 invece che in S1. Fare ciò non violA le regole di handshake che ci siamo posti: semplicemente rimandoIAMO la gestione dell'handshake alla conclusione della temporizzazione. Fino ad allora il produttore rimarrà in attesa del consumatore mantenendo inalterato il valore di *numero*.

- Porre `rfd = 0` significa rinunciare alla protezione che l'handshake garantisce al valore di *numero*. Se la specifica dell'esercizio fosse stata "se numero=0, OUT non deve andare ad 1" allora dovrei porre

`RFD <= 1` per forza in S1, altrimenti non potrei utilizzare numero mentre sono nello stato S1. Vediamo il frammento di descrizione Verilog differente:

```
S1:
begin
 RFD<=1; OUT<=(numero==0)?0:1; COUNT<=COUNT-1;
 STAR<=((COUNT==1)|(numero==0))?S2:S1;
end
S2: begin RFD<=0; OUT=0;…
```

Se mi trovo in S1 e il numero in ingresso è uguale a 0 pongo come valore di OUT 0 e passo in S2. La condizione aggiunta in S1 mi garantisce che OUT non assumerà 1 come valore (senza di essa OUT verrebbe alzato per la durata di un clock). La seconda condizione mi fa passare subito ad S2 se il valore di *numero* è 0: non abbiamo da fare cicli, passiamo subito ad S2 e attendiamo che il produttore alzi /dav prima di tornare in S0.

Capitolo 38

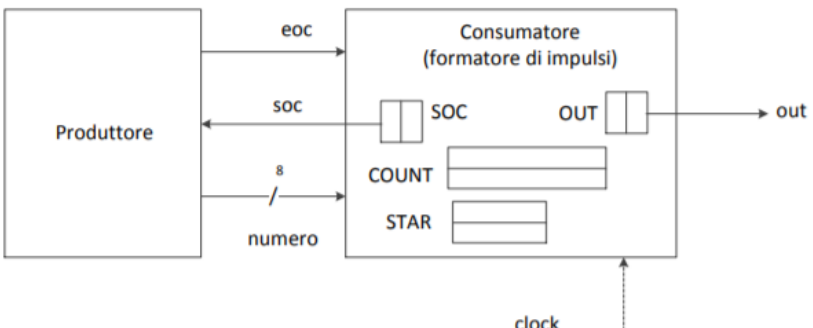
Giovedì 19/11/2020

Formatore di impulsi con handshake soc/eoc

- Modifichiamo le specifiche dell'handshake e introduciamo qualcosa di differente.

- L'*handshake soc-eoc* presenta due variabili logiche:
  - `soc`, variabile in uscita del consumatore (acronimo di *Start Of Computation*)
  - `eoc`, variabile in ingresso del consumatore (acronimo di *End of computation*)

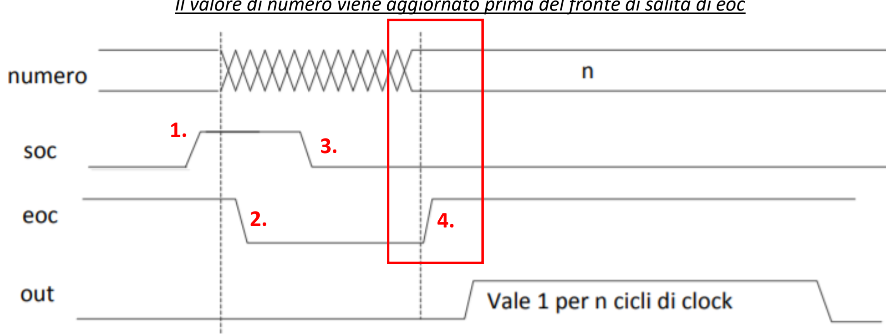
  entrambe sono attivi alti!



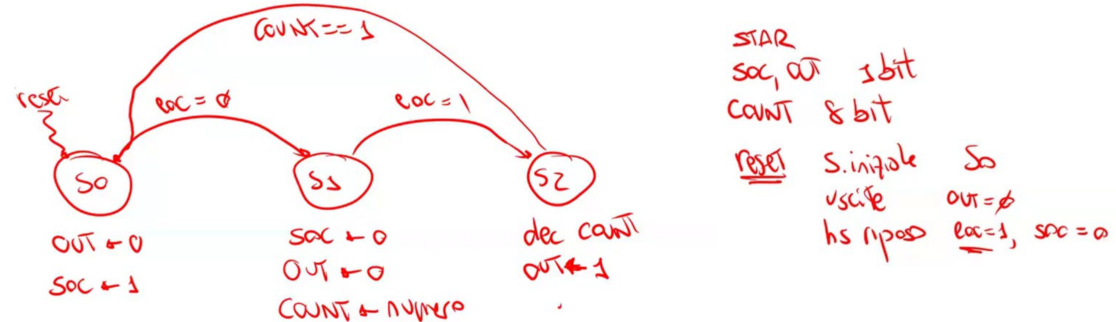
- I ruoli di consumatore e produttore sono invertiti: la prima mossa la fa il consumatore.
- I valori di riposo sono `eof = 1, soc = 0` (il consumatore ha chiesto dati e il produttore li ha forniti).

1. Il consumatore alza soc per informare il produttore che vuole nuovi dati.
2. Il produttore abbassa eoc perché inizia un nuovo processo di computazione. Chiaramente abbassare eoc significa privare di protezione i dati forniti dall'handshake (la stessa cosa che succede quando il consumatore di una rete con *handshake /dav-rfd* abbassa rfd). Il dato sarà nuovamente affidabile quando il produttore ri-alzerà eoc.
3. Il consumatore riporta soc a zero quando capisce che il produttore sta computando.
4. Il produttore computa, mette un nuovo dato sull'uscita, e infine rialza eoc.

Il valore di numero viene aggiornato prima del fronte di salita di eoc



Quali stati interni avrò per la mia rete?



- Siamo in S0: OUT sta a zero, facciamo il primo alzando soc.
- Rimaniamo in S0 finchè non ci accorgeremo che il produttore ha recepito la nostra richiesta. Segue che salteremo ad S1 solo dopo che il produttore avrà abbassato eoc.

- Osserviamo il diagramma di temporizzazione: il conteggio comincia quando il produttore fornisce il dato, cioè quando il produttore alza eof. Finchè rimarremo in S1 aggiorniamo ogni volta il contenuto del registro COUNT col numero fornito dal produttore. Quando il produttore alzerà eoc il valore del numero sarà valido, quindi potremo passare ad S2.
- In S2 avviene il decremento solito. Finchè COUNT sarà diverso da 1 rimango in S2. Dopo ritornerò in S0.

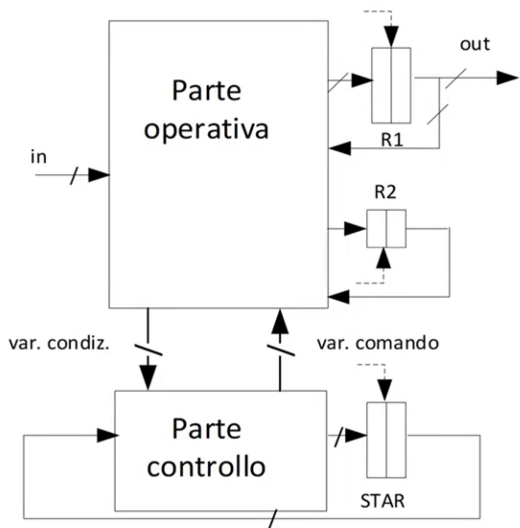
- **Descrizione in Verilog**:

```
module Formatore_di_Impulsi_Soc_Eoc(soc,eoc,numero,out,clock,reset_);
 input clock,reset_;
 output soc;
 input eoc;
 input [7:0] numero;
 output out;
 reg SOC; assign soc=SOC;
 reg OUT; assign out=OUT;
 reg [7:0] COUNT;
 reg [1:0] STAR; parameter S0='B00,S1='B01,S2='B10;

 always @(reset_==0) #1 begin SOC<=0; OUT<=0; STAR<=S0; end
 always @(posedge clock) if (reset_==1) #3
  casex(STAR)
   S0: begin OUT<=0; SOC<=1; STAR<=(eoc==1)?S0:S1; end
   S1: begin SOC<=0; COUNT<=numero; STAR<=(eoc==0)?S1:S2; end
   S2: begin OUT<=1; COUNT<=COUNT-1; STAR<=(COUNT==1)?S0:S2; end
  endcase
endmodule
```

Sintesi di RSS complesse

- Fino ad ora abbiamo parlato solo di descrizioni di RSS complesse usando il linguaggio di trasferimento tra registri del Verilog. Ci siamo limitati a simulare il comportamento della rete attraverso diagrammi di temporizzazione.
- Vogliamo fare la sintesi della rete descritta in termini di porte AND, OR, NOT e circuiti complessi (tutti quelli che abbiamo introdotto fino ad ora).

- Utilizziamo un procedimento semiautomatico che consiste nello scomporre la RSS in due sottoreti comunicanti:
  - **Parte operativa**, tutta la logica necessaria all'interfacciamento col mondo esterno e alla produzione degli stati di ingressi per i registri operativi.
  - **Parte controllo**, la logica necessaria all'aggiornamento dello stato interno



- L'obiettivo di questo procedimento **NON** è ottenere una sintesi ottima. Le due reti presentano lo stesso clock e comunicano tra di loro attraverso due gruppi di variabili:
  - **Variabili di comando**, dalla parte controllo alla parte operativa.
  - **Variabili di condizionamento**, dalla parte operativa alla parte controllo

Recuperiamo l'esercizio del riconoscitore e contatore di sequenze alternate.

1. Guardiamo ai registri operativi come a registri multifunzionali (ecco a cosa servono). Isolo ciascun registro operativo e individuo le $\mu - operazioni$ diverse. Il registro COUNT, in questo caso, è l'unico registro operativo:

```
S0, S1: COUNT<=COUNT;
S2: COUNT<=(x1_x0=='B10)?COUNT+1:COUNT;
```

Il registro multifunzionale è a due vie, ciascuna corrispondente a una $\mu - operazione$. Nella prima ho la conservazione, la seconda posso implementarla con un ulteriore multiplexer. Volendo potrei ridurre il numero di livelli ma non è nostro interesse.

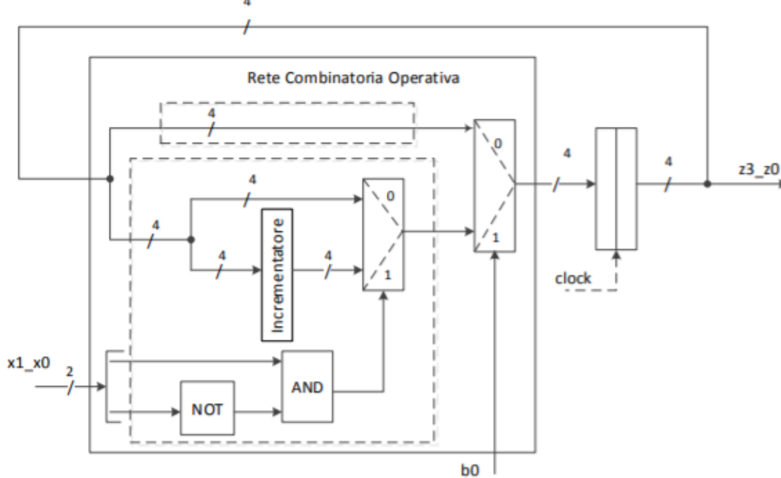
- Come si chiamano le variabili che pilotano il multiplexer? Variabili di comando! Dobbiamo generare la variabile di comando che guida il multiplexer a due vie del registro multifunzionale COUNT. Questa può essere prodotta da una rete combinatoria che abbia come ingresso lo stato interno marcato come segue:

```
assign b0 = (STAR==S2) ? 1 : 0;
```

ne riparleremo.

- Per ogni registro operativo andiamo a sintetizzare una **rete combinatoria operativa**, cioè quella rete che sta davanti al registro operativo. Ogni rete combinatoria operativa presenta in ingresso:
    - Le variabili di comando (che arrivano dalla *Parte controllo*)
    - Lo stato di uscita dei registri operativi

  e produce lo stato di ingresso dei registri operativi.



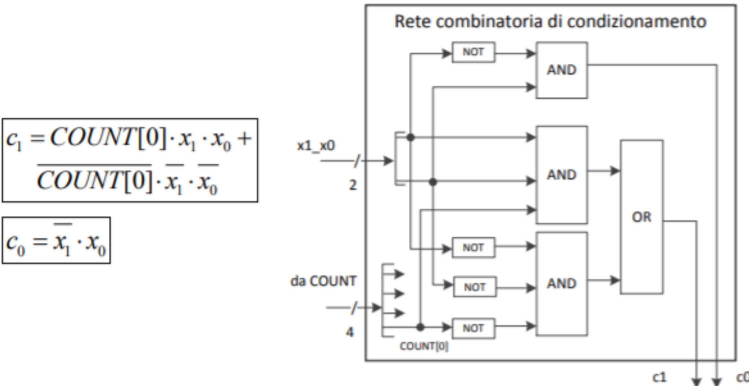
2. Step successivo consiste nel considerare le condizioni che guidano i $\mu - salti$. In questo caso le condizioni sono due

```
match(COUNT[0],x1_x0)==1
x1_x0=='B01
```

per ogni condizione dobbiamo generare una **variabile di condizionamento**, cioè una variabile uguale a 1 se la condizione è vera, o uguale a 0 se la condizione è falsa. Segue

```
assign c1=(match(COUNT[0],x1_x0)==1)?1:0,
c0=(x1_x0=='B01)?1:0
```

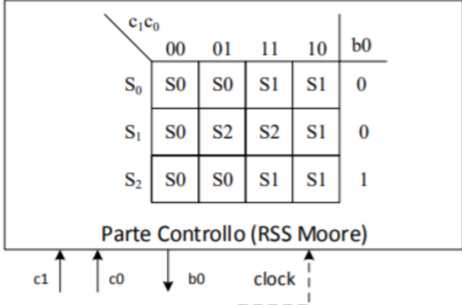
la rete che produce queste due variabili è detta **rete combinatoria di condizionamento**: ho in ingresso le variabili di ingresso, e in questo caso anche lo stato dei registri operativi.



- La parte operativa produce le uscite, presenta in ingresso le variabili di comando e le variabili di ingresso alla rete originale. Come uscita presenta le variabili di condizionamento e le variabili di uscita della rete originale. Chiaramente abbiamo una **RSS di Mealy**: alcune uscite sono funzioni combinatorie degli ingressi!

3. A questo punto dobbiamo sintetizzare la parte controllo, che abbiamo detto avere in ingresso le variabili di condizionamento (abbiamo appena fatto la rete che le genera) e in uscita le variabili di comando (in questo caso la variabile b0). Abbiamo già detto come deve essere generata l'uscita b0.

- Questa rete presenta ingressi e uscite, ma si limita a comunicare con un'altra rete. Non abbiamo contatti col mondo esterno: questa rete è una **RSS di Moore**! Le variabili di condizionamento determinano il passaggio da uno stato interno a un altro, le variabili di comando sono conseguenza dello stato interno presente in un certo istante temporale.
- Ripensiamo alle condizioni e alle variabili di condizionamento conseguenti. Otteniamo la seguente tabella di flusso:



c_1c_0	00	01	11	10	b0
S_0	S0	S0	S1	S1	0
S_1	S0	S2	S2	S1	0
S_2	S0	S0	S1	S1	1

Sintetizzabile con le strategie che abbiamo già introdotto.

- Non è un problema mettere insieme una RSS di Moore e una RSS di Mealy: avrei avuto un anello combinatorio in presenza di due RSS di Mealy.

- **Vedere da pagina 88 le varie descrizioni in Verilog**: prima si passa dalla descrizione originaria a una descrizione con le variabili di comando e le variabili di condizionamento appena definite, infine si ottiene una descrizione dove Parte operativa e Parte controllo sono divise in modo netto. Attraverso un'ulteriore descrizione top-level uniremo le due parti.

Capitolo 39

Martedì 24/11/2020

39.1 Riepilogo sulla sintesi di una RSS compelssa in PO/PC

Abbiamo visto che in modo algoritmico è possibile sintetizzare una RSS complessa distinguendo due moduli.

- La **Parte Operativa**, che si occupa dell'interfacciamento con l'esterno. Ha in ingresso lo stato di ingresso della RSS e in uscita lo stato di uscita della rete. Comunica con la Parte Controllo mediante variabili di condizionamento.

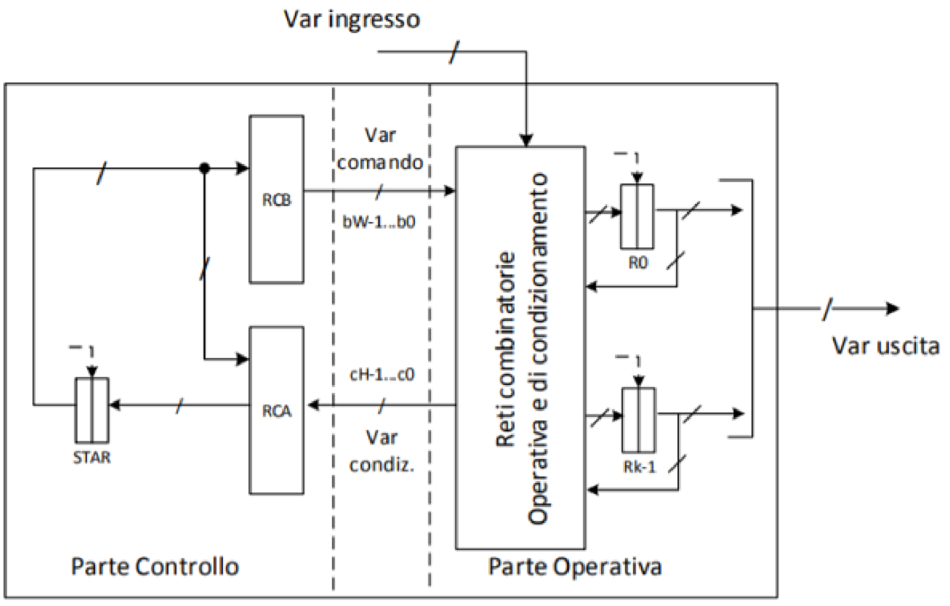
- La **Parte Controllo**, che si occupa di determinare lo stato interno della rete (quindi di aggiornare il registro STAR). Presenta in ingresso solo le variabili di condizionamento, provenienti dalla Parte operativa. Comunica con questa attraverso variabili di pilotaggio.

Come dobbiamo procedere?

- Per quanto riguarda la Parte Operativa:
  - Si consideri ogni singolo registro operativo: per ognuno sintetizzeremo una rete combinatoria operativa. Questa presenta in ingresso l'attuale stato del registro operativo, variabili di comando, ed eventuali variabili di ingresso (dipende dalle condizioni poste). Presenta in uscita il nuovo stato del registro operativo.
    - Immaginiamo ogni registro operativo come un registro multifunzionale (cioè un registro preceduto da multiplexer).
    - Nel codice Verilog cerchiamo tutte le funzioni che possono determinare il valore del registro operativo nell'assegnaqmento procedurale.
    - Quale funzione sarà considerata dipenderà dalle variabili di pilotaggio, generate dalla Parte Controllo. La cosa non è strana: è lo stato interno della rete a dirci cosa fare con un certo registro operativo.
  - Si sintetizza una rete combinatoria di condizionamento. Questa ha in ingresso le variabili di ingresso della rete (quelle necessarie), e in alcuni casi pure lo stato di registri operativi. Presenta in uscita le variabili di condizionamento necessarie per comunicare con la Parte Controllo.
    - Questa rete è solitamente la parte più facile. Si ottiene facendo ricorso all'Algebra di boole, con semplici espressioni.

* Si considerano tutte le condizioni indipendenti, cioè le condizioni che guidano i $\mu-$salti nella rete.
* Ciascuna di queste condizioni sarà assegnata a una variabile di condizionamento. Si capisce, a questo punto, lo scopo delle variabili di condizionamento: dirci se la condizione che determina il $\mu-$salto è vera o falsa.

- Per quanto riguarda la Parte Controllo:
  - Il modulo in questione è una RSS di Moore: isoliamo il registro STAR, abbiamo come ingressi le variabili di condizionamento e come uscite le variabili di comando. Non esiste alcun contatto col mondo esterno, neanche in modo indiretto: non si occupa di produrre le uscite, ma si dedica esclusivamente all'evoluzione dello stato interno della rete.
  - Le variabili di comando permettono di indicare ai multiplexer lo stato interno della rete. Possiamo ottenerle attraverso una semplice rete combinatoria che considera lo stato interno della rete (registro STAR).
  - Le variabili di condizionamento determinano il nuovo stato interno. Le gestiamo attraverso una semplice rete combinatoria che si limita a verificare la veridicità o meno delle condizioni. Noi non riceviamo i dati per verificare le condizioni, ma i risultati delle condizioni (la verifica è già avvenuta nella rete combinatoria di condizionamento). Quindi ci limitiamo a semplici constatazioni.
- Immagine riepilogativa:



39.2 Tecniche ulteriori per la sintesi della parte controllo

Abbiamo già visto che la parte controllo è una rete di Moore, quindi già si dovrebbe conoscere qualcosa per arrivare alla sintesi.

Perchè introduciamo altre tecniche? Potrei avere, in certi casi, un numero di ingressi e uscite elevati. La sintesi della rete coi metodi che già conosciamo rimane possibile, ma diventa complicata e noiosa.

39.2.1 Tecniche $\mu-$address e $\mu-$instruction based

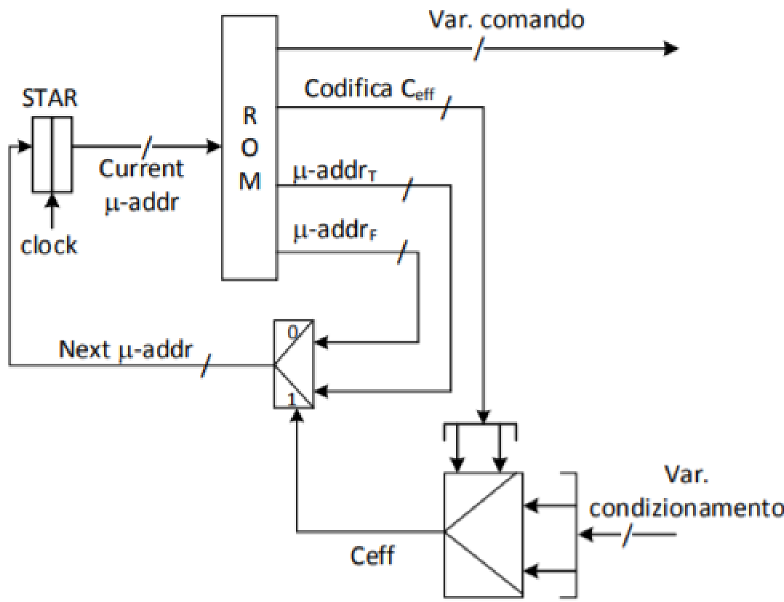
- Le due tecniche si basano su una procedura euristica (cioè fatta ad occhio).
- Non possono essere utilizzate in ogni caso: i μ-salti devono essere incondizionati o a due vie (non a più vie).
  - Si consideri che un $\mu-$salto incondizionato è un caso limite di un $\mu-$salto a due vie.

    ```
    STAR<=S2; <---------------> STAR <= (c1==1) ? S2 : S2;
    ```
- Cosa facciamo?
  - Definiamo:
    - $\mu-$indirizzo: codifica di uno stato interno
    - $\mu-$codice: stato delle variabili di comando
    - C_{eff}: variabile di condizionamento efficace, con cui rappresento la condizione analizzata (che è in ingresso dalla Parte Operativa).
      - Se ho due variabili di condizionamento avrò una codifica su un bit: $c_1 = 1$ e $c_0 = 0$.
      - Se ho quattro variabili di condizionamento avrò una codifica su tre bit: $c_3 = 11, \ldots, c_0 = 00$
      - ... e così via ...
    - $\mu-$indirizzo$_T$: indirizzo a cui si salta nel caso in cui la condizione sia vera.
    - $\mu-$indirizzo$_F$: indirizzo a cui si salta nel caso in cui la condizione sia falsa.
      - Si tenga conto che nel caso in cui si abbia un $\mu-$ salto incondizionato la codifica delle variabili di condizionamento è non specificata, e la codifica dei due $\mu-$indirizzi true e false identica.

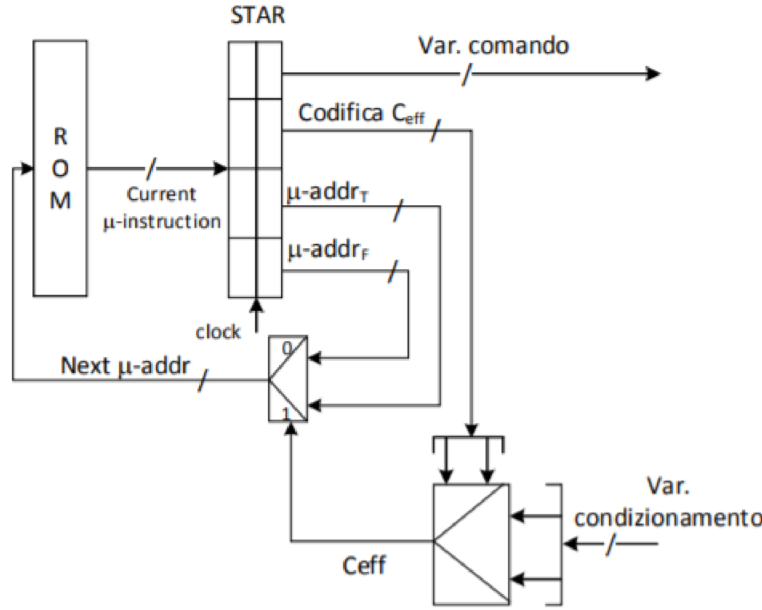
	μ-instruction			
μ-addr	b_1b_0	c_{eff}	**μ-addr$_T$**	**μ-addr$_F$**
00	00	1	00	01
01	11	0	10	01
10	01	1	00	10
11	--	-	--	--

  - Abbiamo ottenuto una memoria ROM con tre celle da 7 bit ciascuna. Il contenuto della cella è detto $\mu-$istruzione, ed è identificato in modo univoco da un $\mu-$ indirizzo.

* Le variabili di comando (il $\mu-$codice) va in uscita (sono le variabili di comando che vanno in ingresso nella PC)
* La codifica C_{eff} è variabile di comando di un multiplexer: indica quale variabile di condizionamento determinerà il $\mu-$salto.
* I $\mu-$indirizzi true e false vanno in ingresso in un multiplexer (true: 1, false: 0) che presenta come variabile di pilotaggio la variabile di condizionamento scelta al multiplexer precedente. Abbiamo già detto che l'unica cosa che facciamo con le variabili di condizionamento è constatare il risultato di una verifica già avvenuta.
* Il multiplexer restituisce il $\mu-$indirizzo della $\mu-$istruzione da considerare al clock successivo. Il $\mu-$indirizzo viene memorizzato nel registro STAR (quindi lo stato interno non sarà codificato nel solito modo).



- Quanto visto fino ad ora consiste in una sintesi $\mu-$address based, cioè una sintesi in cui il valore memorizzato nel registro STAR è il $\mu-$indirizzo.
- La differenza principale tra sintesi $\mu-$address based e quella $\mu-$instruction based sta nel contenuto memorizzato nel registro STAR: nell'ultimo tipo di sintesi memorizzeremo nel registro STAR il contenuto della $\mu-$istruzione, dopo averlo estratto dalla memoria ROM.
  - La cosa può risultare più conveniente in termini di clock: nel primo tipo di sintesi abbiamo ROM e RC operativa in sequenza in cascata. Dal punto di vista del tempo di attraversamento sono le reti più lente. Segue la necessità di un clock largo.
  - Nel secondo tipo di sintesi il registro è grande, però non abbiamo più ROM e RC operativa in cascata (quindi il clock non sarà necessariamente largo). Avremo in cascata, invece, RC di condizionamento e la ROM. Ciò non è un problema: la RC di condizionamento è una delle parti meno pesanti della sintesi.



39.3 Re-introduzione dei $\mu-$salti a più vie: registro MJR

Analizzando la struttura del calcolatore individueremo, nella maggior parte dei casi, $\mu-$salti con al più due vie. Abbiamo due eccezioni:

- L'inizio della fase di fetch, cioè quando devo decodificare il formato dell'istruzione e saltare ad un certo numero di blocchi di $\mu-$istruzioni differenti a seconda del formato. I formati delle istruzioni sono più di due.
- Alla fine di ogni fase di fetch relativa a ciascun formato, quando devo decidere quale istruzione (di quelle consentite in quel formato) eseguire.

Come possiamo gestire $\mu-$salti a più di due vie? Con più di due condizioni la ROM diventerebbe ingestibile!

Perchè? Immaginiamo di avere un $\mu-$salto a cinquanta vie. La verifica delle condizioni richiede normalmente un numero di clock pari al numero di condizione, quindi una condizione per clock. Richiedere 50 clock per la verifica delle condizioni è inaccettabile.

```
S0: begin /*elaborazioni;*/ STAR <= (condizione1)?S1:S0_1; end
S0_1: begin STAR<=(condizione2)?S2:S0_2; end
S0_2: begin STAR<=(condizione3)?S3:S0_3; end
[...]
S0_k-1: begin STAR<=(condizionek-1)?Sk-1:Sk; end
```

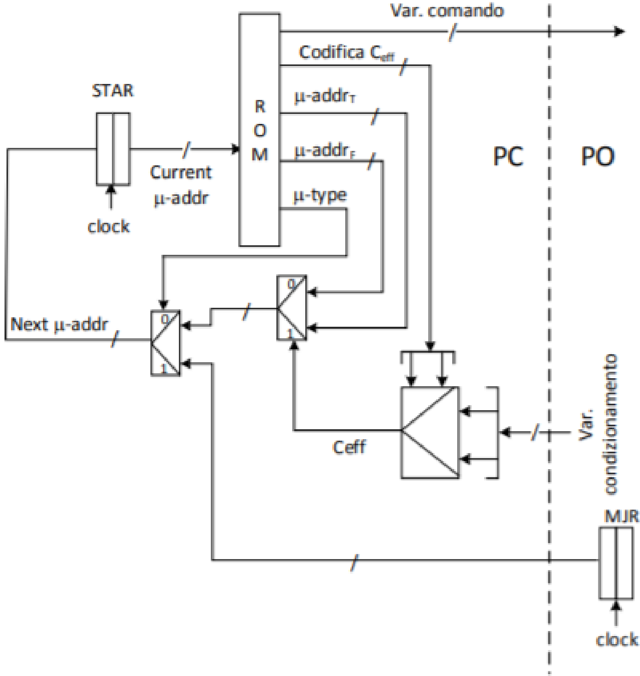
Soluzione La soluzione è l'introduzione del registro operativo MJR (*Multiway Jump Register*). Possiamo porre le condizioni nel seguente modo:

```
S0: begin /*elaborazioni;*/
       MJR<=(condizione1)?S1:
                 (condizione2)?S2:
                 [...]
                 (condizionek-1)?Sk-1:Sk;
                 STAR<=S0_1;
    end
S0_1: begin STAR<=MJR;
end
```

Gestiamo il tutto con soli due clock!

39.3.1 Aggiornamento della sintesi PO/PC

Nella sintesi vogliamo dare la possibilità di gestire i controlli sia con il metodo $\mu-$instruction based (o $\mu - address$ based), sia col registro MJR. Possiamo fare questa cosa introducendo il $\mu-$type, ossia un bit che indica quale metodo dobbiamo utilizzare con quella condizione.



Esempio di codice Verilog Per l'esempio completo andare a pagina 104

```
[...]
input x;
reg [K-1:0] STAR, MJR;
reg A,B,F;
[...]
S0: begin /*elaborazioni*/ STAR<=(x==0)?S0:S1; end
S1: begin /*elaborazioni*/ STAR<=S2; end
```

```
S2: begin /*elaborazioni*/ STAR<=(A!=B)?S5:S3; end
S3: begin /*elaborazioni*/ STAR<=(gamma(A)==0)?S1:S4; end
S4: begin /*elaborazioni*/ STAR<=S5; end
S5: begin MJR<=(x==1)?S4:(A!=F)?S1:S0; STAR<=S6; end
S6: begin /*elaborazioni*/ STAR<=MJR; end
```

39.4 Sottoliste utilizzando il registro MJR

Il registro operativo MJR può essere utilizzato per implementare sottoliste. Ciò equivale a realizzare sottoprogrammi con al più un livello di annidamento (se volessi più di un livello avrei bisogno di una $\mu-$pila, ma non è nostro interesse). Sostanzialmente:

- memorizzo in MJR lo stato interno a cui "saltare" dopo aver eseguito il sottoprogramma, in parallelo salto al primo stato interno relativo alla sottolista;
- alla fine del sottoprogramma utilizzo MJR per saltare allo stato interno memorizzato all'inizio.

```
S0:   begin  /*elaborazioni*/ MJR<=S1; STAR<=Ssub1; end        |
S1:   begin  /*elaborazioni*/; end                             |
                                                               |
[...]                                                          |  Lista princ.
                                                               |
Sx:     begin  /*elaborazioni*/ MJR<=Sx+1; STAR<=Ssub1; end    |
Sx+1:   begin  /*elaborazioni*/; end                           |

[...]

Ssub1:   begin  /*elaborazioni*/ end                |
[...]                                               |  Sottolista
SsubK:   begin  /*elaborazioni*/ STAR<=MJR; end     |
```

Nota finale: a meno che non sia assolutamente necessario o estremamente comodo (o esplicitamente richiesto nel testo di un esercizio), **evitare di usare il registro `MJR`**. In genere la maggior parte degli esercizi che dovrete svolgere può essere svolta usando salti a due vie.

4.5 Riflessione conclusiva su descrizione e sintesi delle reti logiche

Abbiamo visto vari tipi di reti logiche: quelle combinatorie, sia semplici (pochi ingressi ed uscite) sia complesse (e.g., quelle per l'aritmetica, caratterizzate da molti ingressi ed uscite); quelle sequenziali asincrone; quelle sequenziali sincronizzate, sia semplici (cioè con pochi ingressi, stati interni ed uscite) che complesse (le ultime che abbiamo visto). Siamo al punto giusto per riprendere in mano il progetto di una rete logica con maggior cognizione di causa.
Una rete logica deve essere **prima descritta**, e **poi sintetizzata**.
La **descrizione** altro non è che un **modo formale** di fornire le **specifiche del comportamento** della rete medesima. Infatti:

- Nel caso di una **rete combinatoria**, è un'associazione tra stati di ingresso e stati di uscita, per esempio scritta sotto forma di tabella di verità.
- Nel caso di una **rete sequenziale** (asincrona o sincronizzata), cioè di una rete con memoria, è un **diagramma a stati**, che può essere rappresentato:
  a) tramite tabella o grafo se la rete è abbastanza semplice, oppure
  b) tramite un formalismo più complesso, in cui ad ogni stato interno vengono associate delle azioni che la rete esegue (nel nostro caso, assegnamenti a registri).

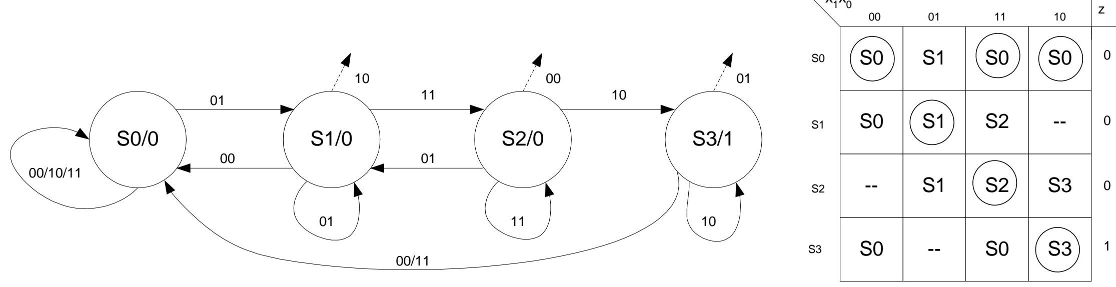
Per descrivere una rete esistono **formalismi differenti**, che si adattano meglio o peggio ad un particolare tipo di rete**.** Anche per uno stesso tipo di rete si possono usare più formalismi diversi. Ad esempio, per una rete combinatoria possiamo usare indifferentemente una **tabella di verità** o **poche righe di Verilog**.

x_2	x_1	x_0	z
0	0	0	0
0	0	1	1
0	1	0	0
…	…	…	…

```
module Rete(x2, x1, x0, z);
  input x2, x1, x0;
  output z;
  assign z=({x2,x1,x0}== 'B000)?    'B0:
           ({x2,x1,x0}=='B001)?    'B1:
           ({x2,x1,x0}=='B010)?    'B0:
            …
endmodule
```

È chiaro che la tabella di verità e la descrizione in Verilog scritta accanto hanno la medesima semantica. È altresì chiaro che entrambe dicono **cosa fa** la rete, **ma non come è realizzata,** cioè quali sono le porte logiche che la compongono.

Allo stesso modo, la descrizione di una RSA si può dare sotto forma di grafo di flusso, di tabella di flusso, o di linguaggio Verilog. Quest'ultimo non l'abbiamo mai usato, ma è chiaro che avremmo potuto farlo.



x_1x_0	00	01	11	10	z
S0	(S0)	S1	(S0)	(S0)	0
S1	S0	(S1)	S2	--	0
S2	--	S1	(S2)	S3	0
S3	S0	--	S0	(S3)	1

È indispensabile avere a disposizione un modo formale per descrivere una rete, perché una descrizione formale può essere **verificata**:

- verificare la descrizione di una RC significa controllare che gli stati di uscita siano quelli desiderati per ogni possibile stato di ingresso. È una verifica **statica**, che si fa per ispezione diretta.
- Verificare la descrizione di una RS (asincrona o sincronizzata) significa **simulare l'evoluzione della rete** a partire da una condizione iniziale di reset, e controllare che questa sia coerente con le specifiche (normalmente date a parole). Questa è una verifica **dinamica** (richiede un'evoluzione temporale della rete), concettualmente simile al processo di testing di un software. Così come è difficile, se non impossibile, testare il software in maniera esaustiva (i.e., per tutti i possibili input), è difficile verificare in maniera esaustiva il comportamento di reti sequenziali che non siano estremamente semplici. Nondimeno, è **sbagliato** non verificare la descrizione di una RS, tanto quanto è sbagliato non testare del software. Un **diagramma di temporizzazione** che mostra la simulazione di una RSS complessa con un certo input, come quelli che abbiamo fatto svariate volte finora, è un modo per verificare la descrizione di quella rete.

Ciò che rende un **formalismo di descrizione** preferibile rispetto ad un altro è **quanto è comodo da usare**. Per una RC semplice una tabella di verità o una mappa di Karnaugh sono (leggermente) più comodi di una descrizione in Verilog. La verifica statica di una tabella di verità è (leggermente) più agevole rispetto a quella di una descrizione in Verilog.

Allo stesso modo, per una RSA (semplice) una tabella di flusso è più comoda da scrivere di una descrizione in Verilog, e soprattutto è più facile – per un utente umano – simulare l'evoluzione della rete sulla tabella di flusso che sulla descrizione in Verilog. Se invece volessi avvalermi di un simulatore Verilog per simulare il comportamento di una RSA, mi converrebbe ovviamente scriverne la descrizione in Verilog.
Viceversa, per una RSS complessa, una descrizione in Verilog risulta ben leggibile, e può essere agevolmente fornita ad un simulatore Verilog per verificare il comportamento della rete.

Una descrizione non può fornire informazioni su **come è realizzata la rete**, cioè quali sono i suoi componenti elementari e come sono interconnessi. Per arrivare a questo livello è necessario procedere alla **sintesi**, che è **sempre** il punto di arrivo degli esercizi che svolgiamo. Fermarsi alla descrizione, infatti, comunica l'impressione di **non essere in grado di realizzare** ciò che si è pensato, impressione che un ingegnere non dovrebbe mai dare.

Esistono **due approcci** per la sintesi, e li abbiamo usati entrambi: quello **euristico**, che consiste nel "fare le cose ad occhio", e quello **formale**, che consiste nel seguire un procedimento algoritmico. L'approccio euristico viene usato tipicamente nel caso di **reti combinatorie per l'aritmetica**. La sintesi richiesta negli esercizi di aritmetica consiste, di fatto, nel prendere dei blocchi "atomici", e.g., sommatori, moltiplicatori, etc. – la cui struttura interna è data per assodata – ed assemblarli in modo che soddisfino delle specifiche date a parole (il testo dell'esercizio). Approcci euristici per la risoluzione di problemi sono tipici del know-how di un ingegnere (il processo di scrittura del software a partire dalle specifiche è, infatti, un procedimento euristico), e richiedono l'esperienza che si acquisisce solo con la pratica.
Abbiamo anche usato **approcci formali** per la sintesi delle reti logiche. Ad esempio:

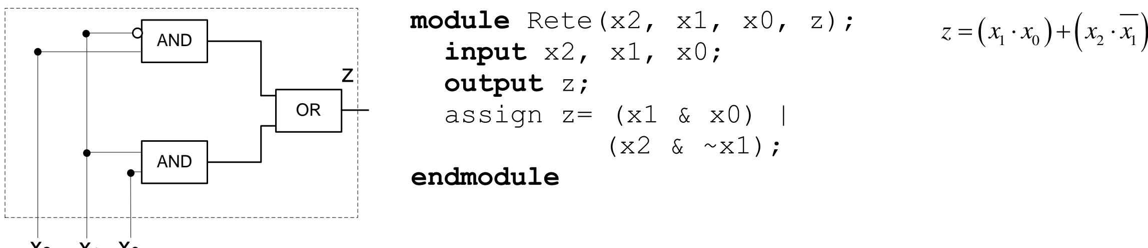
- La sintesi a costo minimo in forma SP (PS, NAND, NOR) di una RC;
- la sintesi di una RSS di Moore, Mealy, Mealy Ritardato, secondo il modello con registri FF-D o FF-JK;
- la sintesi di una RSS complessa secondo il modello con scomposizione in Parte Operativa e Parte Controllo.

In tutti questi casi, il processo seguito è di tipo algoritmico. Si parte dalla **descrizione** della rete, si adotta un **modello di sintesi**, e si procede secondo i passi dell'algoritmo. La scelta del modello di sintesi determina l'algoritmo. Ad esempio, l'algoritmo di sintesi di una RSS di Moore è diverso se

scelgo di usare il modello con registro di stato FF-D o quello con registro di stato FF-JK. L'algoritmo di sintesi di una RC è diverso a seconda che scelga un modello di sintesi SP o PS.

Sia come sia, alla fine una **sintesi** deve essere comunicata a qualcuno (e.g., il tecnico che realizzerà l'hardware) in forma intelligibile, e quindi deve essere **scritta secondo un qualche formalismo** pure lei. Come per le descrizioni, abbiamo usato diversi formalismi, scegliendo di volta in volta quello **più comodo perché più facile da leggere**.

Ad esempio, per le RC semplici abbiamo usato indifferentemente **diagrammi, espressioni di algebra di Boole, o il linguaggio Verilog**.



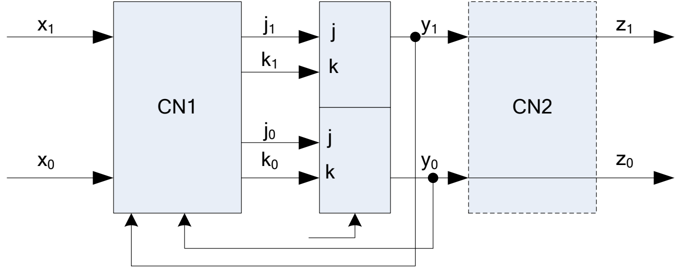

```
module Rete(x2, x1, x0, z);
  input x2, x1, x0;
  output z;
  assign z= (x1 & x0) |
            (x2 & ~x1);
endmodule
```

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

Tutti e tre sono equivalenti, ma noi preferiamo usare le espressioni algebriche perché sono più compatte. Si noti che il linguaggio **Verilog** può essere usato sia per rappresentare la **descrizione che la sintesi**. Ciò è spesso fonte di confusione, perché si confonde il **formalismo** (cioè il linguaggio Verilog) con il **contenuto** (cioè la descrizione o la sintesi, a seconda dei casi). È però chiaro che il pezzo di Verilog scritto sopra è una **sintesi**, in quanto è mappabile direttamente su porte logiche interconnesse. Non è una descrizione perché non dice come stati di uscita corrispondono a stati di ingresso.

Per la sintesi di RC complesse (quali quelle per l'aritmetica) usiamo normalmente **diagrammi**. Più raramente, scriviamo sintesi in Verilog (sono più difficili da leggere).

Per le RSS di Moore, Mealy, e Mealy Ritardato usiamo normalmente un **diagramma** che specifica quale **modello di sintesi** abbiamo scelto, ed **espressioni algebriche** che descrivono le relazioni ingresso-uscita delle reti combinatorie facenti parte del modello.



$$j_1 = k_1 = \overline{y_0} \cdot \overline{x_1} \cdot \overline{x_0} + \overline{y_0} \cdot x_1 \cdot x_0 + y_0 \cdot x_1 \cdot \overline{x_0} + y_0 \cdot \overline{x_1} \cdot x_0$$

$$j_0 = k_0 = 1$$

Per le RSS complesse, è di gran lunga più conveniente usare il **Verilog** per scrivere la sintesi, sempre stando attenti a non confondere il formalismo con il contenuto, visto che usiamo il Verilog **anche** per scrivere la descrizione. Scrivere la sintesi sotto forma di diagramma in questo caso richiederebbe fogli enormi e grossi intrighi di fili, e ne risulterebbe qualcosa di difficile da leggere. **Resta inteso**, in ogni caso, che ciò che si scrive quando si fa una sintesi in Verilog è **in corrispondenza biunivoca con un diagramma**, in cui:

- la parte operativa è composta da registri multifunzionali, che hanno le variabili di comando b_j come ingressi di comando dei propri multiplexer;
- ci sono reti combinatorie che generano le variabili di condizionamento;
- la parte di controllo consiste in un registro di stato ed una ROM (e poco altro), ed ha in ingresso le variabili di condizionamento e in uscita quelle di comando.

Non aver capito questo significa non aver capito cosa si sta facendo, ed è **grave**. Per rendere chiaro che si è capito, negli esercizi di esame è **sempre** richiesto di disegnare almeno i diagrammi delle reti combinatorie di condizionamento e di specificare il contenuto della ROM sotto forma di tabella di verità. Talvolta, può essere richiesto di disegnare anche alcune porzioni della parte operativa.

Fare una **sintesi** di una rete sequenziale senza averne fatto la descrizione (errore che capita di vedere talvolta durante la correzione dei compiti) è cosa completamente **assurda**: **è praticamente impossibile inferire** il comportamento della rete a partire da una sintesi, e quindi non si riesce a verificare la correttezza della medesima rispetto a delle specifiche date.
Come nota a margine, si osserva che per una **rete combinatoria** è invece relativamente semplice risalire alla descrizione (e.g., alla tabella di verità) a partire dalla sintesi – il che non è comunque un buon motivo per saltare la descrizione. Ciò è dovuto al fatto che le reti combinatorie non hanno memoria.

Una sintesi **deve quindi essere coerente con la descrizione che l'ha prodotta,** in modo tale che la correttezza del comportamento della rete (che si verifica sulla descrizione) sia mantenuta nella implementazione della medesima. I procedimenti formali per sintetizzare le reti (quelli elencati sopra) partono infatti da una descrizione e la realizzano secondo un modello. Tali procedimenti **garantiscono** che la rete così sintetizzata si comporti nel modo specificato dalla sua descrizione.

Ad esempio, una RSS di Moore sintetizzata a partire dalla tabella di flusso secondo il modello con FF-D come elementi di memorizzazione, con clock dimensionato opportunamente, si comporta come specificato nella tabella di flusso, se pilotata correttamente. Analogamente, una RSS complessa sintetizzata secondo il modello PO/PC a partire dalla descrizione, si comporterà come la descrizione, purché il clock sia dimensionato in modo corretto e gli input non vengano modificati a cavallo dei fronti di salita del clock.
Alcune ottimizzazioni in fase di sintesi sono talvolta possibili. Si deve tener presente, però, che le sintesi non si giudicano dal livello di ottimizzazione: se fosse necessario ottenere una sintesi "di costo minimo", qualunque cosa questo voglia dire, lo si farebbe fare ad un programma.

Ricapitolando: il progetto di una rete logica (qualunque) si affronta nel seguente modo:

1. **descrizione**, per stabilire in modo formale (e quindi verificabile) qual è il comportamento della rete;
2. **sintesi**, a partire dalla descrizione e seguendo un apposito modello, per realizzare una rete che si comporta come specificato nella descrizione.

Per rappresentare i risultati di entrambi i passi si usano dei **formalismi**, quelli che meglio si adattano al tipo di rete. Il fatto che lo stesso formalismo (in particolare, il Verilog) si possa usare per entrambi i passi non può essere fonte di confusione, se si è capito cosa si sta facendo. La scelta del formalismo da usare nella descrizione e nella sintesi risponde a criteri di facilità di utilizzo ed economicità di spazio. In particolare:

- per le RC (semplici), una descrizione come tabella di verità è più leggibile. Una descrizione come mappa di Karnaugh è equivalente, e facilita il procedimento di sintesi secondo uno qualunque dei modelli noti. La sintesi si dà sotto forma di espressioni algebriche.
- per le RSA e le RSS di Moore, Mealy e Mealy ritardato con pochi ingressi e pochi stati interni, una descrizione come tabella di flusso è facile da verificare, ed inoltre facilita il procedimento di sintesi secondo uno dei modelli noti. La sintesi si dà sotto forma di indicazione del modello da utilizzare, più le espressioni algebriche delle uscite delle reti combinatorie facenti parte del modello.
- Per le RSS complesse, una descrizione in Verilog è facile da verificare, ed inoltre facilita il procedimento di sintesi secondo il modello con scomposizione in PO/PC. La sintesi si dà parimenti in Verilog, con alcuni diagrammi e tabelle di verità a completamento.

Parte VII

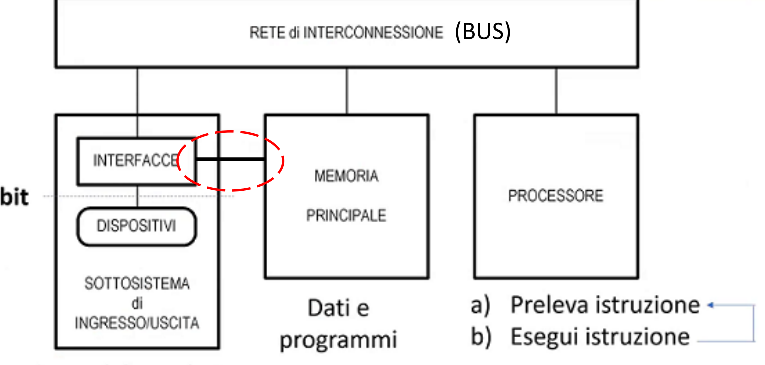
Struttura di un calcolatore

Capitolo 41

Giovedì 26/11/2020

Struttura del calcolatore

Il blocco finale consiste nel descrivere in Verilog un calcolatore, TUTTO, da cima a fondo. Riprendiamo la solita immagine del calcolatore di Von Neumann



Descriveremo processore, memoria, interfacce e dispositivi di I/O utilizzando quanto studiato fino ad ora! Chiaramente il calcolatore che descriveremo non sarà quello che normalmente utilizziamo per seguire le lezioni (quello è troppo complicato), ma un qualcosa che fino a trent'anni fa (quando Stea studiava) poteva essere considerato piuttosto potente.

<u>**Cosa possiamo dire sui vari moduli?**</u>

- **Sottosistema di I/O**. Si gestisce la codifica delle informazioni ed il loro scambio col mondo esterno (in entrambi i sensi). All'interno abbiamo:
  - **Dispositivi** (trasduttori), che effettuano la codifica vera e propria;
  - **Interfacce**, che gestiscono i vari dispositivi, cioè standardizzano il colloquio tra processore e trasduttore (necessario, il processore non deve conoscere i trasduttori per poter lavorare).

    Ciascuna interfaccia possiede un numero piccolo di registri dove il processore può svolgere operazioni di lettura o di scrittura (in casi rari entrambe): un registro in un interfaccia può contenere, ad esempio, l'ultimo tasto premuto della tastiera, oppure può indicare se deve essere accesa una spia del nostro dispositivo.

- **Memoria principale**. Contiene le istruzioni da eseguire e i dati elaborati dal processore (ricordiamo che alcuni dati possono essere ospitati nel sottosistema di I/O). Una parte di memoria è adibita a memoria video: non la faremo, ma dobbiamo tenere conto del collegamento diretto tra interfaccia e memoria principale (devo salvare in memoria la replica di ciò che stampo sullo schermo). La memoria è realizzata con tecnologia RAM, e in parte ROM (o EPROM, per eseguire certe istruzioni al reset).

- **Processore**. Il processore ciclicamente preleva istruzioni e le esegue. Normalmente abbiamo una sequenza di istruzioni eseguita nell'ordine posto, salvo utilizzo di istruzioni operative che alterano il normale flusso (si indica un nuovo indirizzo e il prelievo di istruzioni riparte da lì). Il processore si ferma quando incontra un'istruzione *HLT*.

  <u>Al momento del reset il processore deve essere avviato in modo consistente</u>.
  - Devo indicare la prima operazione da leggere (un'operazione ben precisa)
  - Chiaramente l'indirizzo di memoria ci porta a un'area realizzata con tecnologia ROM/EPROM, quindi una memoria non volatile.

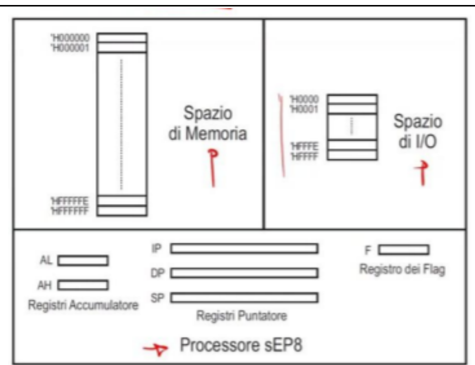
  Posso fare questo inizializzando l'IP (*Instruction pointer*) e altri registri. La memoria non volatile contiene al suo interno un programma bootstrap eseguito all'accensione del calcolatore.

  <u>Quale processore utilizzeremo?</u> Il sEP8 (acronimo di *8-bit simple Educational processor*), che presenta le seguenti proprietà:
  - elabora dati a 8 bit;
  - lavora in aritmetica in base 2 con interi rappresentati in C2;
  - indirizza memoria di 16Mbyte (quindi avrò 24 fili per rappresentare tutti gli indirizzi).

Il calcolatore consiste sostanzialmente in un insieme di RSS: tutti gli elementi che vedremo, tranne dispositivi (parti elettromagnetiche che non sono di nostro interesse) e parte della memoria (che è una RSA), sono RSS. Tutte le RSS presenteranno un piedino di reset e saranno collegate allo stesso circuito di reset (in modo tale da avere un avvio consistente).

In particolare, descriveremo il processore come una RSS sintetizzabile in Parte Controllo e Parte operativa. Ne daremo una specifica come con una qualunque RSS. Oltre a questo vedremo come il processore si interfaccia con le altre reti e qual è il suo comportamento osservabile (il suo linguaggio macchina, perderemo un po' di tempo sulla questione).

Calcolatore visto dal programmatore



- La memoria consiste in uno spazio lineare di 2^{24} byte (16Mbyte)
- Gli indirizzi sono a 24bit (combinazioni possibili di valori sono, non a caso, 2^{24})

- Lo spazio di I/O (registri di interfaccia) consiste in uno spazio lineare di 2^{16} locazioni o porte da un byte.
- Gli indirizzi sono a 16bit (tenerne conto quando vediamo il bus).
- Queste locazioni consistono nell'insieme dei registri di interfaccia che il processore può teoricamente indirizzare. Perché teoricamente? Non è detto che ad ogni locazione corrisponda un registro di interfaccia, anzi è molto probabile che la maggior parte di questo spazio non presenti implementazione fisica (le interfacce sono poche).
- Il processore, da queste porte, può leggere o scrivere un byte alla volta (quindi se dobbiamo leggere più byte dovremo svolgere più operazioni)

- **Registri del processore**:
  - Registri accumulatore (AH, AL, 8 bit), contengono operandi di elaborazioni.
  - Registro dei flag (8 bit), i 4 bit più interessanti per noi sono il CF, lo ZF, il SF e l'OF.
  - Registri puntatore (a 24 bit, visto che devono contenere indirizzi):
    - IP (Instruction pointer), indirizzo della prossima istruzione da eseguire;
    - SP (Stack pointer), contiene l'indirizzo del top della pila;
    - DP (Data Pointer), contiene l'indirizzo di operandi a seconda delle modalità di indirizzamento.

  Al reset dobbiamo inizializzare il registro dei flag e l'instruction pointer, nel seguente modo

```
F <= 'H00;
IP <= 'HFF0000;  // Chiaramente memoria ROM implementata a partire da questo indirizzo)
```

  Osservazione: abbiamo anche altri registri, noi abbiamo visto solo i registri utilizzabili nelle istruzioni.

Linguaggio Assembler e Linguaggio macchina

- All'inizio del corso abbiamo introdotto le peculiarità del linguaggio Assembler, che è legato al linguaggio macchina da un rapporto 1:1. Per descrivere il calcolatore dovremo parlare di linguaggio macchina, in modo da poter descrivere il comportamento osservabile della macchina.

- Il linguaggio Assembler, ricordiamo, è diverso da processore a processore. In questo caso utilizzeremo un linguaggio Assembler simile il più possibile a quello già visto per i processori Intel. Chiaramente l'obiettivo non è programmare in Assembler, ma disporre il linguaggio in modo tale che il calcolatore sia facilmente descrivibile.

Linguaggio Assembler

- **Formato delle istruzioni:**

```
OPCODE source, destination
```

  Dove OPCODE è il codice operativo dell'istruzione, mentre source e destination indicano, rispettivamente, l'operando sorgente e quello destinatario. In alcuni casi è assente l'operando source, in altri (rari) entrambi gli operandi (*NOP* e *HLT*).

- **Modalità di indirizzamento**:
    - Indirizzamento di registro, uno o entrambi gli operandi sono nomi di registro
      ```
      OPCODE AL, AH
      OPCODE DP
      ```
    - Indirizzamento immediato, l'operando sorgente è specificato direttamente nell'istruzione come costante (chiaramente possiamo fare questa cosa solo con l'operando sorgente)
      ```
      OPCODE $0x10, AL
      ```
    - Indirizzamento di memoria, valido per il sorgente o per il destinatario (mai contemporaneamente). L'indirizzamento di memoria può essere
        - Diretto, l'indirizzo è specificato direttamente nell'istruzione
          ```
          OPCODE  0x1010, AL
          ```
        - Indiretto, la locazione di memoria ha indirizzo contenuto nel registro DP
          ```
          OPCODE (DP), AL
          ```
    - Indirizzamento delle porte di I/0: le porte di I/O si indirizzano in modo diretto, specificando l'indirizzo della porta nell'istruzione stessa
      ```
      IN 0x1010, AL
      OUT AL, 0x9F10
      ```
- **Istruzioni di controllo**: istruzioni che alterano il flusso di esecuzione del programma. Permettono salti, condizionati e non, chiamate di sottoprogramma ed istruzioni di ritorno
  ```
  JMP indirizzo
  Jcon indirizzo
  CALL indirizzo
  RET
  ```
  Nelle prime tre istruzioni si specifica l'indirizzo a cui si salta (si sostituisce l'indirizzo di IP), le ultime due interagiscono con la pila. La CALL salva in pila il contenuto di IP (3 BYTE), cioè l'indirizzo dell'istruzione successiva alla CALL. La RET preleva dalla pila un indirizzo (3 byte) e lo sostituisce ad IP.

Linguaggio macchina e formati

- Un processore deve tradurre un'istruzione Assembler
  ```
  OPCODE source, destination
  ```
  in una sequenza di zeri e uni con una certa sintassi. La sintassi è il linguaggio macchina del processore, che deve essere compatta e facile da interpretare (per il compilatore, non per noi).
    - La prima cosa che guardano gli esseri umani, normalmente, è il tipo di operazione (l'Assembler, concepito per gli esseri umani, pone il tipo prima degli operandi).
    - I processori, invece, guardano per prima cosa gli operandi. Vediamo degli esempi
        - `MOV AH, AL`
          Gli operandi sono già posti all'interno di registri
        - `MOV $0x10, AL`
          Il processore deve leggere in memoria l'operando sorgente indicato nell'istruzione
        - `MOV (DP), AL`
          Il processore dovrà leggere in memoria per procurarsi l'operando sorgente. L'indirizzo non è il registro DP, ma il valore contenuto nel registro stesso.

    Dobbiamo distinguere, in queste operazioni, la fase di fetch dalla fase di esecuzione: la prima consiste nel procurarsi gli operandi (e può essere diversa in base all'indirizzamento scelto), la seconda è uguale per tutte queste istruzioni (cioè spostare valori)

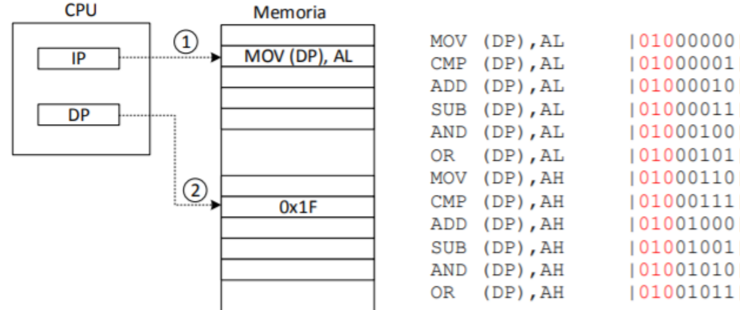
- Ciascuna istruzione macchina è lunga almeno un byte. Il primo byte di ogni istruzione codifica sia il **tipo di operazione** (su 5 bit, 32 opcode possibili) che il modo in cui si devono recuperare gli operandi (su 3 bit, 8 formati possibili). Quest'ultima cosa è detta **formato dell'istruzione**.

- **Formati possibili**:

```
HLT         |00000000|
NOP         |00000001|
MOV AL,AH   |00000010|
MOV AH,AL   |00000011|
INC DP      |00000100|
SHL AL      |00000101|
SHR AL      |00000110|
NOT AL      |00000111|
SHL AH      |00001000|
SHR AH      |00001001|
NOT AH      |00001010|
PUSH AL     |00001011|
POP AL      |00001100|
PUSH AH     |00001101|
POP AH      |00001110|
PUSH DP     |00001111|
POP DP      |00010000|
RET         |00010001|
```

  - Formato F0 (000)
    - Categoria in cui rientrano tutte le istruzioni per le quali non è necessario compiere nessuna azione per procurarsi gli operandi.
    - In questa categoria rientrano operazioni in cui gli operandi sono registri o dove non sono presenti operandi (HLT, NOP, RET).
    - La fase di fetch consiste esclusivamente nella lettura di un byte (quello dell'istruzione) visto che non c'è altro da fare.

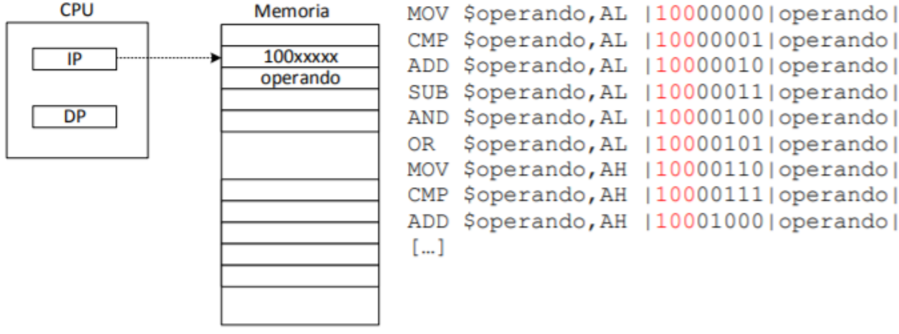
  - Formato F2 (010)
    - Categoria in cui rientrano le istruzioni dove l'operando sorgente si trova in memoria ed è indirizzato tramite DP.
    - Il sorgente deve essere ripescato in memoria. Dovrò fare una seconda lettura in memoria per portare l'operando sorgente dentro il processore.



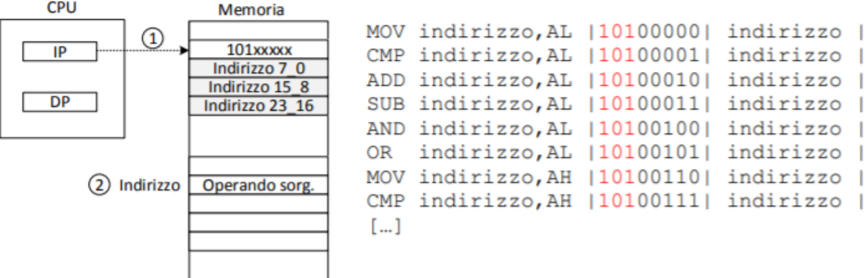
  - Formato F3 (011)
    - Categoria in cui rientrano le istruzioni dove l'operando destinatario si trova in memoria ed è indirizzato usando DP (solo le MOV)
    - Codifico su un unico byte l'istruzione. La fase di fetch consiste nel non fare niente: il contenuto da spostare è già presente nel processore, stessa cosa l'indirizzo da raggiungere.
    - La scrittura del destinatario avviene in fase di esecuzione.

```
MOV AL,(DP)  |01100000|
MOV AH,(DP)  |01100001|
```

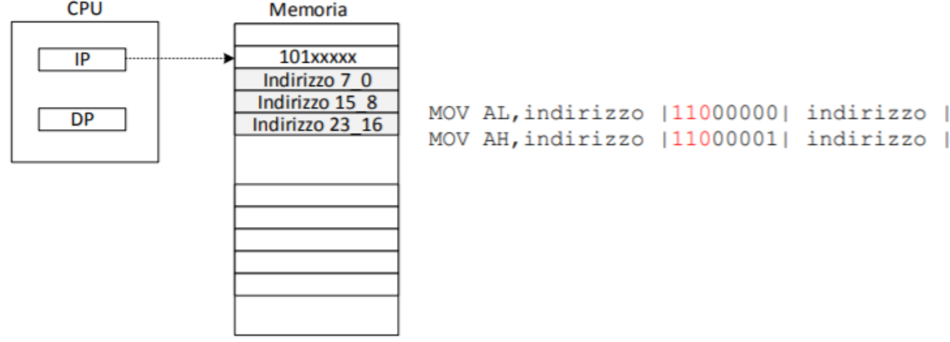
  - Formato F4 (100)
    - Categoria in cui rientrano le istruzioni dove l'operando sorgente è indirizzato in modo immediato, e sta su 8 bit.
    - L'istruzione è lunga due byte: il primo contiene l'istruzione, il secondo l'operando indirizzato in modo immediato. La fase di fetch consiste nel fare due letture consecutive.



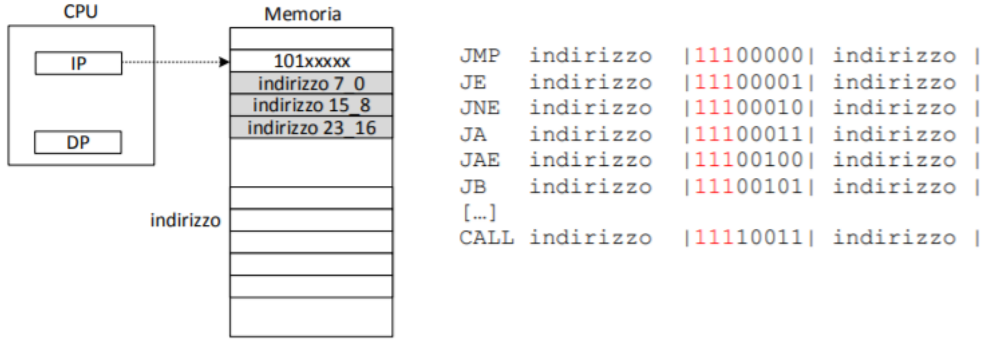
- Formato F5 (101)
  - Categoria in cui rientrano le istruzioni dove l'operando sorgente è indirizzato in modo diretto. Ciò pongo direttamente l'indirizzo del sorgente.
  - In fase di Fetch l'operando sorgente deve essere riportato nel processore.
  - L'operazione sarà lunga 4 byte: uno di opcode e tre di indirizzo di memoria (24 bit per poter rappresentare qualunque indirizzo). Seguono tre cicli di lettura consecutivi a partire da IP. Ciò non basta: devo fare un'altra lettura all'indirizzo trovato: a quel punto ho raggiunto l'operando sorgente e posso porlo nel processore.



- Formato F6 (110)
  - Categoria in cui rientrano le istruzioni dove l'operando destinatario è in memoria, indirizzato in modo diretto.
  - Il processore dovrà leggere 4 byte in memoria: uno per l'opcode, tre per l'indirizzo del destinatario.
  - La scrittura del destinatario avviene in fase di esecuzione.



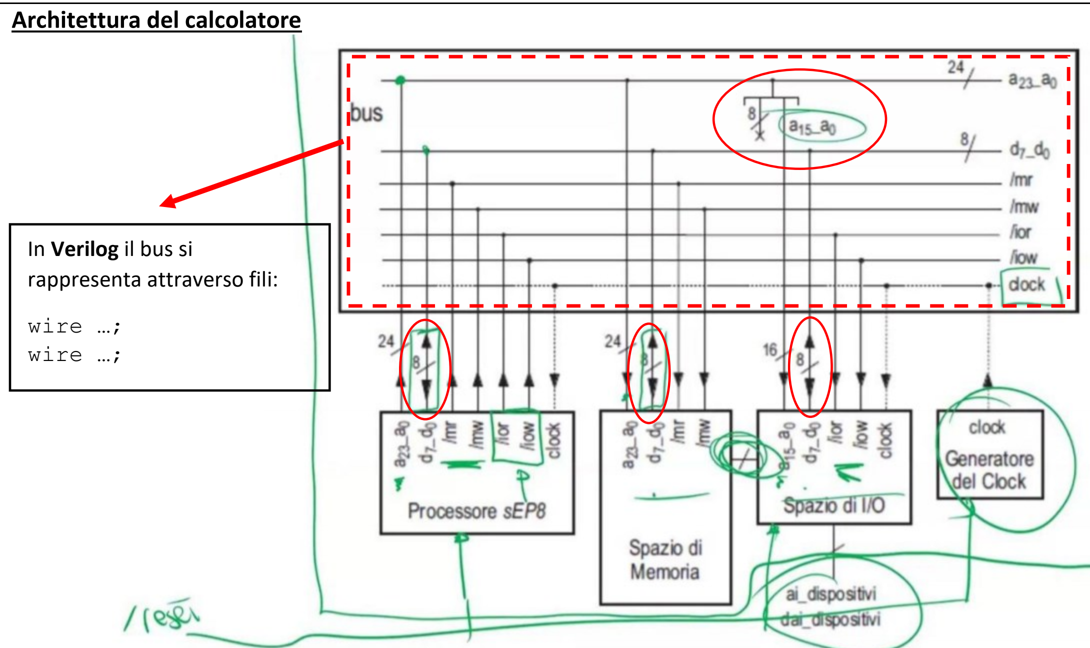
- Formato F7 (111)
  - Uguale al precedente, raggruppa le istruzioni di controllo (CALL, JMP, Jcon) in cui ho un indirizzo di salto.
  - Utilizzo un byte per l'opcode, altri tre per l'indirizzo. In fetch abbiamo la lettura di 4 byte consecutivi, a partire da IP.



- ○ Formato F1 (001) “formato delle varie ed eventuali”
  - Categoria in cui rientrano le istruzioni non classificabili nei formati precedenti: istruzioni I/O, MOV con uno dei registri a 24 bit.
  - Le azioni differiscono da un’istruzione a un’altra: si è preferito fare un unico formato dove ci si limita ad estrarre solo l’opcode. La gestione degli operandi viene rimandata alla fase di esecuzione.
  - La cosa è poco pulita, ma molto più semplice.

Lista completa delle istruzioni con formato corrispondente alle pagine 185-187 del libro di Corsini

Architettura del calcolatore



- Spogliamo il calcolatore e vediamo cosa è presente sulla rete di interconnessione

- Abbiamo:
  - ○ **24 fili di indirizzo** che partono dal processore sEP8 e indicano l’indirizzo delle locazioni di memoria o delle porte di I/O dove vuole leggere e scrivere. Entrano in ingresso in tutti gli altri moduli. Attenzione allo spazio di I/O: avendo solo 64K mi bastano le 16 cifre meno significative per rappresentare tutti gli indirizzi possibili.
  - ○ **Fili di dati**. Il processore legge e scrive singoli byte alla volta: ho bisogno di 8 fili che dovranno essere pilotati alternativamente dal processore e dagli altri dispositivi (porte tristate). Dobbiamo evitare cortocircuito sui fili di dati.
  - ○ **Fili di controllo**. Attivi bassi con cui possiamo pilotare alcuni moduli a partire dal processore: /mr e /mw per leggere e scrivere in memoria, /ior e /iow per leggere e scrivere nello spazio di I/O. Chiaramente l’uso di queste variabili terrà conto delle leggi di temporizzazione viste per i cicli di lettura in memoria RAM.
  - ○ **Generatore di clock**: tutti i moduli del calcolatore , tranne la memoria, sono collegati allo stesso generatore di clock.
  - ○ **Fili di interconnessione tra interfacce e dispositivi**.
  - ○ **Fili di comunicazione tra memoria video e adattatore grafico** (non vedremo)
  - ○ Tenere conto anche della presenza dei **piedini per il reset** (il reset arriva contemporaneamente a tutti i moduli che ne hanno necessità). Per semplicità grafica la parte relativa al reset viene solitamente omessa.

41.1 Ricapitoliamo sui formati

I formati non vanno imparati a memoria, ma ricordati mediante la logica.

Quanti sono gli operandi? Otto, lo capisco dal numero di bit dedicati al formato (3).

Quali sono i formati? Seguire la scaletta.

- **Primo formato** (F0): quello dove la vita è più semplice, non dobbiamo andare a recuperare nessun operando.
- **Terzo e quarto formato** (F2, F3): registri puntatore, rispettivamente in source e destination. L'altro operando è un registro.
- **Quinto formato** (F4): costante in source. L'altro operando è registro.
- **Sesto e settimo formato** (F5, F6): indirizzamento diretto, rispettivamente in source e dest. L'altro operando è registro.
- **Ottavo formato** (F7): istruzioni di salto (sia quello condizionato che non). L'operando è registro.
- **Secondo formato** (F1): varie ed eventuali, precisamente istruzioni IN/OUT o modifica di registri a 24bit (i registri puntatore). Lo si lascia in fondo nel ragionamento.

Capitolo 42

Martedì 01/12/2020

Spazio di memoria

RAM	'H000000 – 'H09FFFF
Mem Video (64Kb)	'H0A0000 – 'H0AFFFF
RAM	'H0B0000 – 'HFEFFFF
EPROM (64Kb)	'HFF0000 – 'HFFFFFF

- Lo spazio di memoria consiste in 16Mbyte di locazioni realizzate in larga parte con tecnologia RAM e in parte con tecnologia EPROM (questa parte conterrà il *programma bootstrap* da eseguire all'avvio).
- Una parte ulteriore della memoria è adibita a memoria video: questa è di tipo diverso dalle altre, visto che dobbiamo permette lo svolgimento di operazioni di lettura e scrittura in simultanea da più oggetti (l'interfaccia per stampare la schermata aggiornata, il processore per aggiornare il contenuto della schermata).

- Lo spazio di memoria è strutturato in più parti:
  - Due realizzate in tecnologia RAM
  - Una adibita a memoria video
  - Una realizzata in tecnologia EPROM.

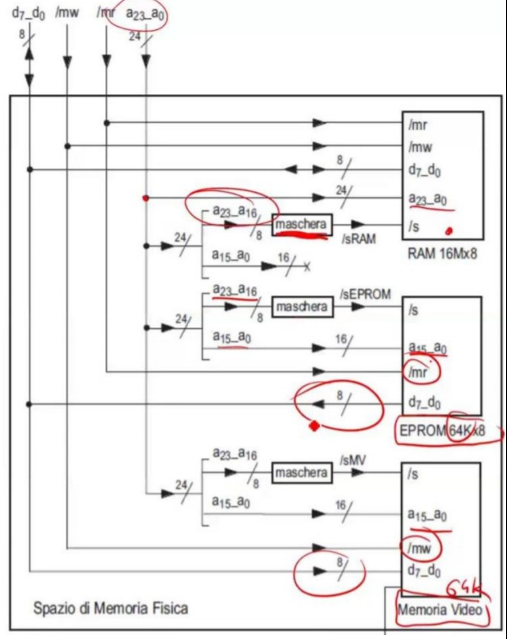
  Gli intervalli di indirizzo sono evidenti nell'immagine. Per ottenere una memoria del genere dobbiamo recuperare quanto detto sul montaggio in parallelo di memorie. Utilizzeremo delle maschere (rete combinatoria) per indicare quale chip di memoria, tra quelli introdotti, ci interessa visitare (genera il segnale di abilitazione).

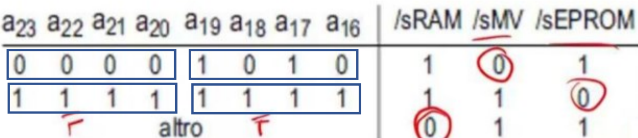
Nella rete avremo tre chip: uno di RAM 16Mx8, due a 64Kx8 (la EPROM e la memoria video).

- Tutti gli indirizzi dello spazio di memoria sono a 24bit: mentre nella RAM sono rilevanti anche le 8 cifre significative per determinare la posizione all'interno del CHIP, nelle altre sono rilevanti solo le cifre rimanenti. Le cifre più significative, in quei casi, servono solo a indicare quale tra i due chip vogliamo raggiungere.

- Pongo in ingresso nelle maschere le 8 cifre più significative dell'indirizzo. Le maschere determineranno i valori degli attivi alti /sRAM, /sEPROM, e /sMV.
- Chiaramente non abbiamo l'ingresso /mw nella memoria EPROM (possibili solo operazioni di lettura).

- **Come scriviamo la tavola di verità della maschera?**





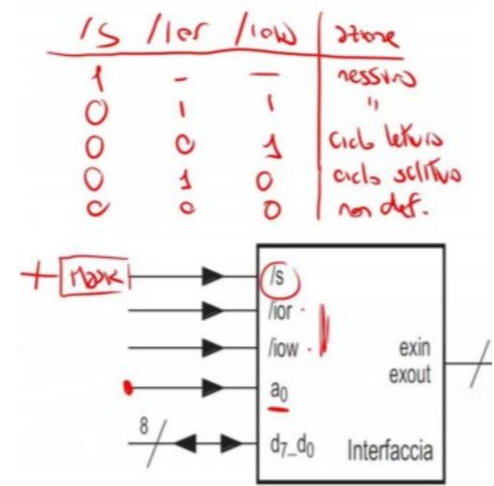
a_{23}	a_{22}	a_{21}	a_{20}	a_{19}	a_{18}	a_{17}	a_{16}	/sRAM	/sMV	/sEPROM
0	0	0	0	1	0	1	0	1	0	1
1	1	1	1	1	1	1	1	1	1	0
altro								0	1	1

- Consideriamo che la RAM è divisa in due parti: considero le cifre più significative (fisse) della memoria video e della EPROM, e la RAM la lascio alle varie ed eventuali.

- Le cifre più significative degli indirizzi della memoria video sono **0A**: 0000 e 1010
- Le cifre più significative degli indirizzi della EPROM sono **FF**: 1111 e 1111
- Con cifre significative diverse vado in memoria RAM.

- **Conclusione**: il segnale di select viene generato dall'indirizzo, non abbiamo un filo di select sul bus.
- **Ulteriore osservazione**: la RAM copre anche gli indirizzi coperti dalla EPROM e dalla memoria video. Se indico questi indirizzi, tuttavia, la RAM non risponde (la maschera non genera il segnale di selezione).

Spazio di I/O



Struttura di un'interfaccia con tavola di verità per ripassare

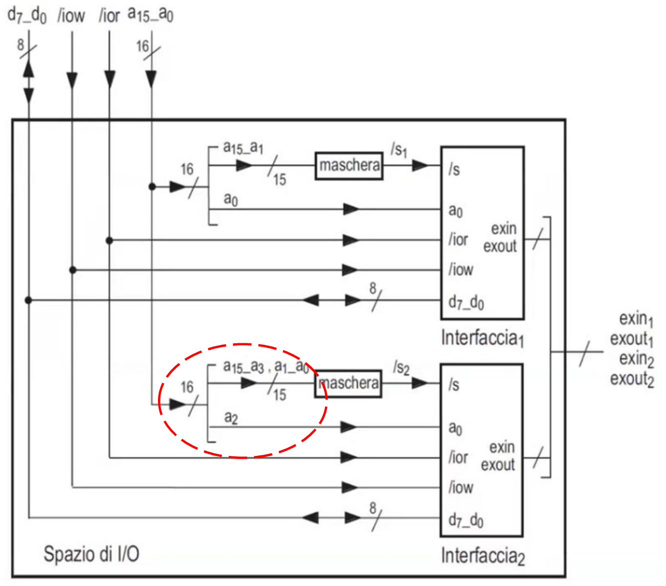
- Spazio di 64K celle (dette porte), non tutte implementate (l'implementazione è a cura delle interfacce montate sul calcolatore, poche).

- **Un'interfaccia** (qua a lato) si presenta come una memoria RAM: ho un piedino di select /s prodotto da una maschera.
- Le variabili /ior e /iow hanno la stessa funzione di /mr e /mw, rispettivamente. Sono identiche ma non uguali.
- Le variabili che non vanno nella maschera finiscono in ingresso come indirizzi interni se l'interfaccia presenta più di una porta. Nel nostro esempio ne ha due, quindi ci servirà una singola variabile logica.

- **Osservazioni sulle interfacce dal lato bus**:
  - In una RAM si può leggere e scrivere in una qualunque locazione. Un'interfaccia, invece, può supportare solo operazioni di lettura o solo operazioni di scrittura. Chiaramente uno dei due fili di comando, in questi casi, risulta superfluo. Nella maggior parte dei casi le interfacce presenteranno porte di entrambi i tipi.
  - Se un'interfaccia presenta una sola porta allora non ho bisogno delle variabili di indirizzo (mi basta solo il select)

- **Osservazioni sulle interfacce dal lato dispositivo**:
  - Gli ingressi e le uscite variano da interfaccia a interfaccia e verranno descritti più avanti.
  - **A cosa ci serve avere delle interfacce?**
    - I dispositivi hanno velocità molto diverse tra loro, e spesso sono più lenti del processore (visto che si interfacciano col mondo esterno). Se io collegassi i dispositivi direttamente al bus il processore dovrebbe conoscere le proprietà del dispositivo cosa insana realizzare i processori tenendo conto dei dispositivi esistenti, che cambiano nel tempo in modo rapido)
    - Le modalità di trasferimento dei dati sono molto diverse. Alcuni dispositivi trasferiscono un bit alla volta, altri gruppi di bit.

  L'interfaccia permette di avere temporizzazioni omogenee e trasferimento di dati omogeneo.



- Vediamo il nostro spazio di I/O, con due interfacce:
  - Entrambe hanno due porte (lo vediamo dalla variabile a_0)
  - Entrambe sono interfacce sia di ingresso che di uscita.
  - Abbiamo 16 fili (non 24) che entrano nello spazio di I/O. 15 fili vanno nella maschera per generare il select.

  - Nell'esempio supponiamo di avere le porte della prima interfaccia agli offset 'H03C8, 'H03C9, e le porte della seconda interfaccia agli offset 'H0060, 'H0064. La maschera ci permette di generare i valori di /s1 ed /s2

- **Attenzione**: non per forza la cifra meno significativa è quella utilizzata per distinguer le porte. Nella seconda interfaccia utilizziamo la terza cifra meno significativa per indicare la porta desiderata!

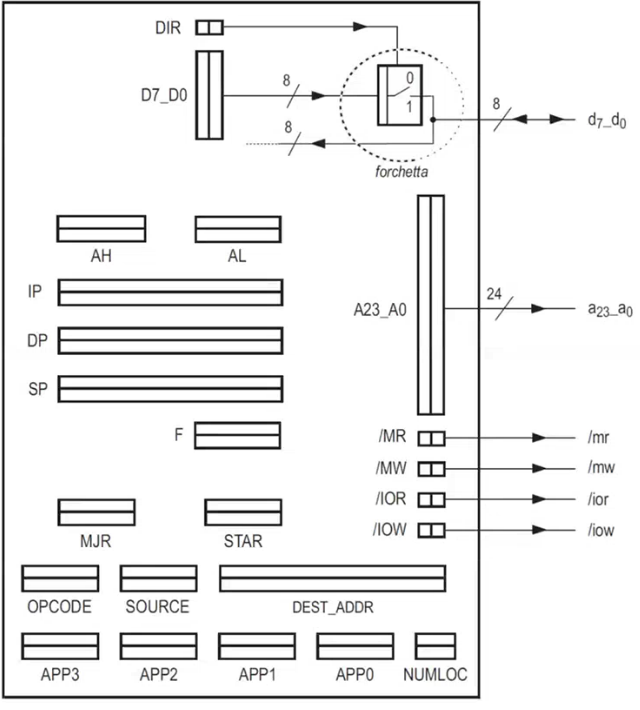
- ‘H03C8: 0000 0011 1100 1000
- ‘H03C9: 0000 0011 1100 1001

- ‘H0060: 0000 0000 0110 0000
- ‘H0064: 0000 0000 0110 0100

Cifre significative con cui identifico la porta dell’interfaccia

Processore

Il processore è una RSS sincronizzata che presenta un certo numero di registri. Abbiamo:



- Registri a supporto delle uscite, quindi:
  - D7_D0, per i fili di dati
  - A23_A0, per i fili di indirizzo
  - /MR e /MW, per interagire con la memoria principale
  - /IOR e /IOW, per interagire con il sottosistema di I/O
- Un registro DIR per gestire la porta tristate per i fili di dati. Se DIR è uguale a zero la porta tristate è in alta impedenza, altrimenti è in conduzione.

- Il registro STAR con cui indichiamo lo stato interno della rete.

- Un registro dei flag, che conterrà flag come CF, SF, ZF, OF.

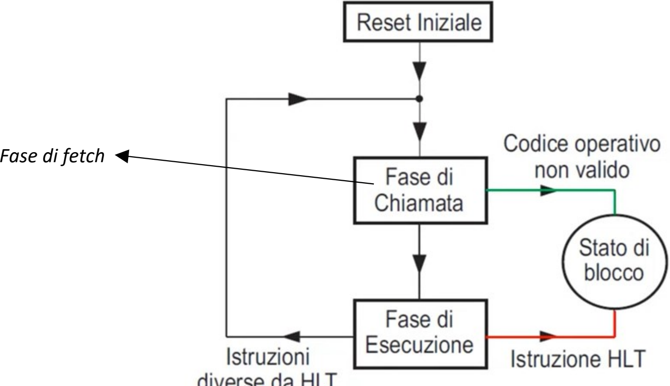
- I registri accumulatori AH e AL

- Il registro puntatore DP (indirizzamento di memoria indiretto)
- Il registro puntatore SP per la pila
- Il registro puntatore IP, per ricordarsi ogni volta l’istruzione successiva da eseguire. In alcuni casi potrebbe essere incrementato più volte (dipende dalla dimensione dell’istruzione, che può essere superiore al singolo byte).

- Cinque registri a supporto delle operazioni di lettura e scrittura:
  - APP0, APP1, APP2, APP3, che contengono il valore letto o da scrivere
  - NUMLOC, registro contatore utilizzato nelle operazioni di lettura e scrittura (per capire quando abbiamo finito).

- Registri a supporto dell’esecuzione delle istruzioni:
  - OPCODE, registro contenente gli 8 bit con formato e istruzione.
  - SOURCE e DEST_ADDR, registri contenenti dati sugli operandi. Non è detto che siano usati sempre. Ricordiamoci che del sorgente ci interessa il contenuto, mentre del destinatario l’indirizzo. Più avanti è presente un pdf di Corsini che spiega i vari casi relativamente ai formati.
  - MJR, registro per i salti (spiegato qualche lezione fa)

- **Fasi del processore**:



- **Fase di reset**: inizializzazione del contenuto dei registri (F, IP, ma anche altri).
  - `IP <- 'HFF0000`
  - `F <- 0`
  - Tutti i registri che reggono variabili di comando dovranno essere inizializzati in modo coerente: in particolare tutti gli attivi bassi dovranno assumere come valore 1.
  - Dobbiamo anche mettere in alta impedenza la porta tri-state. Segue DIR inizializzato a 0, cambiamo il suo valore solo quando dovremo scrivere sul bus (ricordiamoci che quando la porta tristate del processore si comporta in un modo quella presente dall'altra parte – per esempio quella in memoria principale – dovrà comportarsi in modo opposto).
  - STAR verrà inizializzato col primo stato della fase di fetch.

- **Fase di fetch**:
  - Preleva un byte dalla memoria, all'indirizzo IP
  - Incrementa IP (punta al byte successivo)
  - Controlla che il bit appena letto corrisponda all'OPCODE di una delle istruzioni note.
  - Il contenuto del byte letto deve essere copiato nel registro OPCODE. Inoltre dobbiamo valutare il formato dell'istruzione (tre bit più significativi). Valutare il formato significa decidere cosa fare (per ogni formato abbiamo una certa procedura)
    - Per alcuni formati dovremo incrementare IP opportunamente (in base al numero di bit da leggere)
    - In alcuni formati dobbiamo procurarci l'indirizzo dell'operando destinatario, e inserirlo in DEST_ADDR. Nel formato F3 l'indirizzo già sta in DP, mi basta copiarlo in DEST_ADDR. Nei formati F6 e F7 devo andarlo a leggere in memoria: incremento di 3 il valore di IP, leggo 3 bit e li sposto nel processore.
    - In altri formati, F0 ed F1, il processore non fa niente in fase di fetch.
  - Ultima cosa è la lettura del contenuto di OPCODE (i cinque bit meno significativi): devo capire qual è l'istruzione da seguire.

- **Fase di esecuzione**
  - Il processore esegue l'istruzione che ha decodificato.
  - Torna nella fase di fetch a meno che non stia eseguendo l'istruzione di HLT.

- **Stato di blocco**: raggiunto nei seguenti casi:
  - OPCODE non valido;
  - istruzione HLT.

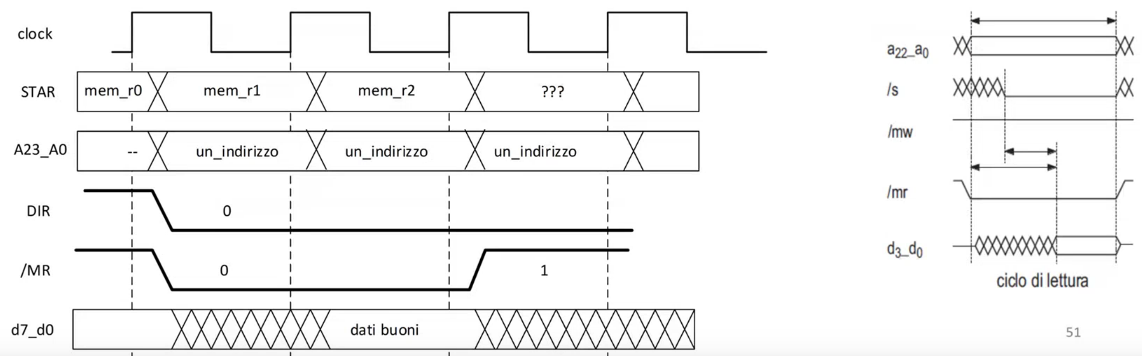
  Non è possibile uscire da questo stato: l'unico modo è fare reset.

- **Letture e scritture in memoria e spazio di I/O**:
  - Il processore legge in memoria durante la fase di fetch.
  - Durante la fase di esecuzione il processore dovrà leggere e scrivere in memoria (RET, MOV) o nello spazio di I/O (IN, OUT)

  - **Letture e scritture in memoria**:
    - Ricordarsi della struttura della RAM statica (multiplexer, demultiplexer, variabili di comando, RC per le variabili di pilotaggio).
    - Ricordarsi la temporizzazione delle RAM statiche in lettura e scrittura

    - **Cosa significano queste cose dal punto di vista del processore?**
      Vediamo il ciclo di lettura:

```
mem_r0: begin A23_A0<=un_indirizzo; DIR<=0; MR_<=0; STAR<=mem_r1; end
mem_r1: begin STAR<=mem_r2; end //stato di wait
mem_r2: begin QUALCHE_REGISTRO<=d7_d0; MR_<=1; ......... ; end
```



- Nel primo stato imposto l'indirizzo uguale a qualcosa, pongo DIR uguale a 0 (finchè non ho dati sensati rimane a 0). Pongo MR_ a 0 e indico il passaggio alla stato successivo (micro-istruzione a singolo step).
- Il secondo stato è detto stato di wait, utilizzato per dare alla memoria tempo per rispondere. D'ora in poi lo daremo per scontato, in modo tale da ottenere descrizioni più semplici.
- Il terzo stato è quello in cui abbiamo dati affidabili: li campioniamo in quale registro e riportiamo MR_ a 1.

- **Letture successive**: nell'ultimo stato non alzo MR_ e pongo un nuovo indirizzo da leggere.
- **Domanda**: posso alzare DIR a 1 nell'ultimo stato assieme a MR_? No, perché le tristate della memoria sono ancora in conduzione e ci mettono più tempo ad alzarsi rispetto alle tristate del processore. Se alzo insieme i due valori creo, per poco tempo, una situazione in cui entrambe le tristate sono attive.

Vediamo il ciclo di scrittura:

```
mem_w0: begin A23_A0<=un_indirizzo; D7_D0<=un_byte; DIR<=1; STAR<=mem_w1; end
mem_w1: begin MW_<=0; STAR<=mem_w2; end
mem_w2: begin MW_<=1; STAR<=mem_w3; end
mem_w3: begin DIR<=0; ....... ; end
```

- Operazione distruttiva. Sappiamo che dobbiamo attendere la stabilizzazione di /s e degli indirizzi prima di portare giù /mw.
- I dati, ricordiamo, devono essere corretti a cavallo del fronte di salita.

- Primo stato: setto l'indirizzo, pongo i dati che voglio salvare in ingresso, alzo DIR e indico il passaggio allo stato successivo.
- Secondo e terzo stato: stati di attesa.
- Terzo stato: abbasso DIR, abbiamo scritto.

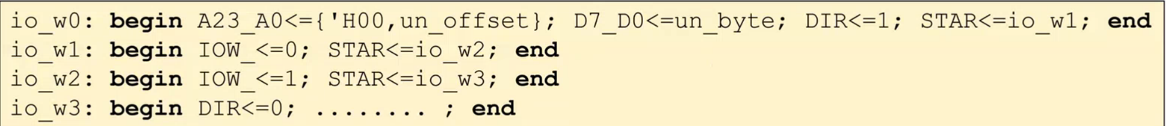
- **Posso scrivere in D7_D0 nel terzo stato?** No, altrimenti non sarebbe rispettata la regola di temporizzazione principale (dati costanti a cavallo del fronte di salita).
- **Posso portare DIR a 0 direttamente nel secondo stato?** No, altrimenti i dati non rimangono stabili.
- **Posso porre un nuovo indirizzo nel terzo stato?** No, devo aspettare il clock successivo.

  - **Letture nello spazio di I/O**:
    Vediamo la lettura

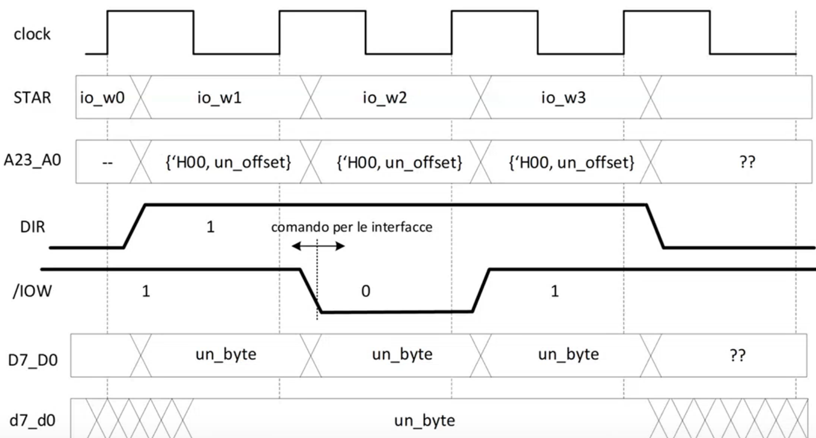
```
io_r0: begin A23_A0<={'H00,un_offset}; DIR<=0; STAR<=io_r1; end
io_r1: begin IOR_<=0; STAR<=io_r2; end
io_r2: begin STAR<=io_r3; end //stato di wait
io_r3: begin QUALCHE_REGISTRO<=d7_d0; IOR_<=1; .... ; end
```

- Le letture sono simili, MA NON UGUALI, alle lettura in memoria.
- Differenza: MR_ può essere portato a 0 nello stesso stato in cui setto gli indirizzi. La lettura in memoria è un'operazione non distruttiva, se gli indirizzi ballano non è un problema. Nello spazio di I/O anche la lettura può essere distruttiva: se io leggo da un'interfaccia un dato questa si regola col suo dispositivo per svolgere operazioni di scrittura. Quindi posso avere dati appena letti subito sovrascritti.
- Nell'ultimo stato posso assegnare il valore in uscita (d7_d0) a qualche registro. Alzo IOR_ e sono obbligato a farlo: in caso di nuove lettura siamo obbligati a finire un ciclo e aprirne un altro.

Vediamo la scrittura:



```
io_w0: begin A23_A0<={'H00,un_offset}; D7_D0<=un_byte; DIR<=1; STAR<=io_w1; end
io_w1: begin IOW_<=0; STAR<=io_w2; end
io_w2: begin IOW_<=1; STAR<=io_w3; end
io_w3: begin DIR<=0; ........ ; end
```



- Nel ciclo di scrittura in memoria l'assegnamento a D7_D0 poteva essere spostato nello stato successivo. Nella scrittura nello spazio di I/O dobbiamo avere i dati già pronti a cavallo del fronte di discesa di IOW_: questo perché alcune interfacce memorizzano sul fronte di discesa, e non su quello di salita. Segue che D7_D0 deve essere assegnato per forza nel primo stato e che non possa essere spostato.

- **Accessi per più di un byte alla memoria**:
  - Il processore deve poter leggere operandi a 2 e 3 byte.
  - Fa comodo strutturarsi di micro-sottoprogrammi di lettura/scrittura modulari. Questi sottoprogrammi possono essere riferiti ogni volta che dobbiamo leggere più di un byte.

  - **Registri utilizzati**:
    - Il registro MJR per salvare il micro-indirizzo di ritorno, cioè lo stato su cui devo tornare dopo aver eseguito il micro-sottoprogramma.
    - I registri APP0, APP1, APP2, APP3 per salvare i byte letti o da scrivere.
    - Il registro NUMLOC come contatore del numero di byte da leggere/scrivere.

  - **Lettura**
    - I micro-sottoprogrammi sono i seguenti:
      - readB (1 byte)
      - readW (2 byte)
      - readM (3 byte)
      - readL (4 byte, non lo utilizzeremo)
    - I parametri in ingresso sono A23_A0, che contiene il primo indirizzo in memoria (modificato dai micro-sottoprogrammi), DIR a zero e il valore di MJR (dobbiamo indicare

da dove ricominciare dopo aver terminato l'esecuzione del micro-sottoprogramma).

- Codice del microprogramma:

```
// PER L'ESECUZIONE
```

Dati in ingresso — *Chiamata di un micro-sottoprogramma*

```
Sx: begin … A23_A0<=un indirizzo; MJR<=Sx+1; STAR<=readB; end
Sx+1: begin … <utilizzo di APP0> end
```

Utilizzo dei dati letti dal micro-sottoprogramma. Questi dati sono recuperati dai registri di supporto

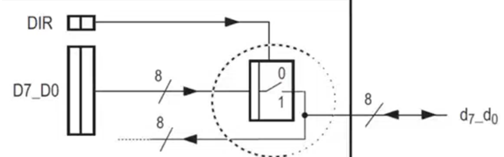
```
// MICROSOTTOPROGRAMMA PER LETTURE IN MEMORIA
readB: begin MR_<=0; DIR<=0; NUMLOC<=1; STAR<=read0; end
readW: begin MR_<=0; DIR<=0; NUMLOC<=2; STAR<=read0; end
readM: begin MR_<=0; DIR<=0; NUMLOC<=3; STAR<=read0; end
readL: begin MR_<=0; DIR<=0; NUMLOC<=4; STAR<=read0; end
```

Inizializzazione delle operazioni di lettura. Abbasso MR_ visto che voglio svolgere l'operazione, indico il numero di locazioni da leggere in NUMLOCK e passo alla prima lettura

```
read0: begin APP0<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
STAR<=(NUMLOC==1) ? read4 : read1; end
read1: begin APP1<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
STAR<=(NUMLOC==1) ? read4 : read2; end
read2: begin APP2<=d7_d0; A23_A0<=A23_A0+1; NUMLOC<=NUMLOC-1;
STAR<=(NUMLOC==1) ? read4 : read3; end
read3: begin APP3<=d7_d0; A23_A0<=A23_A0+1; STAR<=read4; end
read4: begin MR_<=1; STAR<=MJR; end
```

Operazioni di lettura. Salvo il byte letto nell'apposito registro APPx. Dopo aver letto e spostato nel processore mi chiedo se ho svolto il numero di letture necessarie. Se ho finito abbaso MR_ e indico come stato successivo quello posto all'inizio in MJR.

- **Scrittura**
  - I micro-sottoprogrammi sono i seguenti:
    - writeB (1 byte)
    - writeW (2 byte)
    - writeM (3 byte)
    - writeL (4 byte)
  - I parametri in ingresso sono A23_A0, che contiene il primo indirizzo in memoria su cui lavorare (verrà modificato dai micro-sottoprogrammi), DIR a 0 (ci pensa il sottoprogramma ad alzarlo e poi ad abbassarlo), APPj (j=0…3) con i byte da scrivere, MJR (come prima).



Ricordarsi a cosa serve DIR. Se la porta tristate ha 1 la "levetta" è alzata.

  - Codice del microsottoprogramma:

```
// PER L'ESECUZIONE
Sx: begin … APP1<=dato 16 bit[15:8]; APP0<=dato 16 bit[7:0];
A23_A0<=un indirizzo; MJR<=Sx+1; STAR<=writeW; end
Sx+1: begin … <prosecuzione del programma dopo la scrittura> end
```

Solite cose fatte prima, cambiano solo gli ingressi e i micro-sottoprogrammi chiamati

```
// MICROSOTTOPROGRAMMA PER SCRITTURE IN MEMORIA
// PIU' NOIOSO, PER SVOLGERE OGNI SINGOLA SCRITTURA SERVE UN CICLO IN PIU
// Metto il primo indirizzo su cui scrivo, alzo DIR perché devo svolgere operazioni
di lettura, indico il numero di scritture da fare, infine cambio stato per svolgere
la prima scrittura.
// In ogni ciclo di scrittura prima abbasso MW_, poi lo rialzo (ogni volta)

writeB: begin D7_D0<=APP0; DIR<=1; NUMLOC<=1; STAR<=write0; end
writeW: begin D7_D0<=APP0; DIR<=1; NUMLOC<=2; STAR<=write0; end
writeM: begin D7_D0<=APP0; DIR<=1; NUMLOC<=3; STAR<=write0; end
writeL: begin D7_D0<=APP0; DIR<=1; NUMLOC<=4; STAR<=write0; end
write0: begin MW_<=0; STAR<=write1; end
write1: begin MW_<=1; STAR<=(NUMLOC==1)?write11:write2;
end

write2: begin D7_D0<=APP1; A23_A0<=A23_A0+1;
NUMLOC<=NUMLOC-1; STAR<=write3; end
write3: begin MW_<=0; STAR<=write4; end
write4: begin MW_<=1; STAR<=(NUMLOC==1)?write11:write5;
end

write5: begin D7_D0<=APP2; A23_A0<=A23_A0+1;
NUMLOC<=NUMLOC-1; STAR<=write6; end
write6: begin MW_<=0; STAR<=write7; end
write7: begin MW_<=1; STAR<=(NUMLOC==1)?write11:write8; end

write8: begin D7_D0<=APP3; A23_A0<=A23_A0+1; STAR<= write9; end
write9: begin MW_<=0; STAR<= write10; end
write10: begin MW_<=1; STAR<= write11; end

write11: begin DIR<=0; STAR<=MJR; end
```

Ricordare l'ordine delle cifre nei vari bit.

Ogni volta:

- Indico l'indirizzo su cui voglio scrivere (nel caso della prima scrittura è indicato come parametro in ingresso) e i valori da porre sui fili di dati D7_D0.
- Dopo aver posto questi dati, E NON PRIMA, abbasso MW_ a 0.
- Al ciclo successivo rialzo e verifico se ho scritto il numero di byte richiesti.
- Se ho finito vado all'ultimo stato del micro-sottoprogramma: riabbasso la levetta della porta tristate e pongo come stato successivo quello indicato in MJR all'inizio.

- **Promemoria**: negli indirizzi più bassi si pongono le cifre meno significative.

Descrizione in Verilog del processore

Vediamo la descrizione in Verilog del processore. Nel corso della descrizione faremo uso di reti combinatorie in grado di semplificarci alcune operazioni. Non ci dedicheremo alla descrizione accurata di queste funzioni: sono estremamente noiose, ma non impossibili da scrivere per noi.

```
//------------------------------------------------------------------
// DESCRIZIONE COMPLETA DEL PROCESSORE
//------------------------------------------------------------------
module Processore(d7_d0,a23_a0,mr_,mw_,ior_,iow_,clock,reset_);
 input clock,reset_;
 inout [7:0] d7_d0;
 output [23:0] a23_a0;
 output mr_,mw_;
 output ior_,iow_;

// REGISTRI OPERATIVI DI SUPPORTO ALLE VARIABILI DI USCITA E ALLE
// VARIABILI BIDIREZIONALI E CONNESSIONE DELLE VARIABILI AI REGISTRI
 reg DIR;
 reg [7:0] D7_D0;
 reg [23:0] A23_A0;
 reg MR_,MW_,IOR_,IOW_;
 assign mr_ = MR_;
 assign mw_ = MW_;
 assign ior_ = IOR_;
 assign iow_ = IOW_;
 assign a23_a0 = A23_A0;
 assign d7_d0=(DIR==1)?D7_D0:'HZZ; //FORCHETTA
```

Definisco i registri di supporto alle variabili di uscita e definisco come valore di queste uscite quelli dei corrispondenti registri

Evidenziate in rosso due righe con sintassi nuova: con inout indichiamo fili che possono fungere sia come ingressi che come uscite, con 'HZZ indichiamo l'alta impedenza (in questo caso se DIR == 0)

```
// REGISTRI OPERATIVI INTERNI
 reg [2:0] NUMLOC;
 reg [7:0] AL,AH,F,OPCODE,SOURCE,APP3,APP2,APP1,APP0;
 reg [23:0] DP,IP,SP,DEST_ADDR;
```

Al più lavoriamo su 4 byte, quindi bastano 3 bit. (NUMLOC)

I registri a 24 bit contengono indirizzi

Definisco i registri a supporto delle operazioni di lettura e scrittura, oltre ai registri accumulatori che utilizziamo per gestire operazioni aritmetiche.

```
// REGISTRO DI STATO, REGISTRO MJR E CODIFICA DEGLI STATI INTERNI
 reg [6:0] STAR,MJR;
 parameter fetch0=0, .... write11=86;
```

Con 7 bit rappresentiamo 86 stati interni diversi!

```
// RETI COMBINATORIE NON STANDARD
 function valid_fetch;
 input [7:0] opcode;
 ...  ...
 endfunction
```

Prende in ingresso un byte e restituisce 1 se quel byte è l'OPCODE di un'istruzione nota, 0 altrimenti

```
 function [6:0] first_execution_state;
 input [7:0] opcode;
 ...  ...
 endfunction
```

Prende in ingresso 7 bit, interpretato come OPCODE valido, restituisce la codifica del primo stato interno dell'esecuzione dell'istruzione relativa.

```
 function [7:0] alu_result;
 input [7:0] opcode,operando1,operando2;
 ...  ...
 endfunction
```

Simulo lo ALU interna al processore. Interpreto i 3 byte passati come un OPCODE, un operando sorgente e un operando destinatario. Restituisco il risultato su 8 bit dell'elaborazione svolta. Posso porre come OPCODE add, sub, and, or …

```
 function [3:0] alu_flag;
 input [7:0] opcode,operando1,operando2;
 ...  ...
 endfunction
```

Prendo in ingresso gli stessi byte di alu_result e aggiorno in flag rendendoli consistenti. Restituisco 4 bit che consistono nei 4 flag significativi (quelli che ci interessano di più) del registro F.

```
function jmp_condition;
 input [7:0] opcode;
 input [7:0] flag;
 ...  ...
 endfunction
```

Prendo in ingresso il contenuto del registro OPCODE e quello del registro dei flag. Restituisco 1 se:

- *La condizione di JMP è incondizionata*
- *La condizione di JMP è condizionata e la condizione di salto risulta verificata (lettura dei flag per capire questo).*

```
// ALTRI MNEMONICI
 parameter [2:0] F0='B000,F1='B001,F2='B010,F3='B011,
F4='B100,F5='B101,F6='B110,F7='B111;
```

Formati possibili

```
//------------------------------------------------------------------
// AL RESET_ INIZIALE
 always @(reset_==0) #1 begin IP<='HFF0000; F<='H00; DIR<=0;
 MR_<=1; MW_<=1; IOR_<=1; IOW_<=1; STAR<=fetch0; end
```

Inizializzazione di registri importanti: primo indirizzo di istruzione, reset dei flag, porta tristate in alta impedenza, attivi bassi alzati, prima istruzione di fetch come stato successivo.

```
//------------------------------------------------------------------
// ALL'ARRIVO DEI SEGNALI DI SINCRONIZZAZIONE
 always @(posedge clock) if (reset_==1) #3
 casex(STAR)

//------------------------------------------------------------------
// FASE DI FETCH (CHIAMATA)
Ricordarsi le fasi:
   -  LEGGI GLI OPCODE
   -  PROCURATI GLI OPERANDI
   -  ESECUZIONE DELL'OPERAZIONE CON OPCODE VALIDO
// Pongo come indirizzo quello presente nell'IP e lo incremento (ricordare,
esecuzione in parallelo, non in sequenza)
fetch0: begin A23_A0<=IP; IP<=IP+1; MJR<=fetch1; STAR<=readB; end
fetch1: begin OPCODE<=APP0; STAR<=fetch2; end
fetch2: begin
MJR<=(OPCODE[7:5]==F0)? fetchEnd:
 (OPCODE[7:5]==F1)? fetchEnd:
 (OPCODE[7:5]==F2)? fetchF2_0:
 (OPCODE[7:5]==F3)? fetchF3_0:
 (OPCODE[7:5]==F4)? fetchF4_0:
 (OPCODE[7:5]==F5)? fetchF5_0:
 (OPCODE[7:5]==F6)? fetchF6_0:
/* default */ fetchF7_0;
STAR<=(valid_fetch(OPCODE)==1)? fetch3 : nvi;
end
```

Operazione di lettura di un byte.

Istruzione di ritorno. Passiamo a fetch1 dopo la lettura.

Porto il byte letto in OPCODE e passo allo stato successivo

Prendo le tre cifre più significative dell'OPCODE e verifico a quale formato corrispondono.

Se l'OPCODE è valido passo a fetch3, altrimenti ad nvi

Nessun problema a eseguirle nello stesso stato, valid_fetch non dipende da MJR.

```
// SALTO AL PRIMO PASSO SPECIFICO DELLA FASE DI FETCH (Lo abbiamo determinato prima)
fetch3: begin STAR<=MJR; end
```

F	Byte	OPCODE	SOURCE	DEST_ADDR
F0	1	readB @ IP	--	--
F1	*?*	readB @ IP	--	--
F2	1	readB @ IP	readB @ DP	--
F3	1	readB @ IP	--	DP
F4	2	readB @ IP	readB @ IP	--
F5	4	readB @ IP	readM @ IP, readB	--
F6	4	readB @ IP	--	readM @ IP
F7	4	readB @ IP	--	readM @ IP

```
POP DP      |00010000|
RET         |00010001|
```

- Formato F2 (010)
  - Categoria in cui rientrano le istruzioni dove l'operando sorgente si trova in memoria ed è indirizzato tramite DP.
  - Il sorgente deve essere ripescato in memoria. Dovrò fare una seconda lettura in memoria per portare l'operando sorgente dentro il processore.

```
fetchF2_0: begin A23_A0<=DP; MJR<=fetchF2_1; STAR<=readB; end
fetchF2_1: begin SOURCE<=APP0; STAR<=fetchEnd; end
```

- Formato F3 (011)
  - Categoria in cui rientrano le istruzioni dove l'operando destinatario si trova in memoria ed è indirizzato usando DP (solo le MOV)
  - Codifico su un unico byte l'istruzione. La fase di fetch consiste nel non fare niente: il contenuto da spostare è già presente nel processore, stessa cosa l'indirizzo da raggiungere.
  - La scrittura del destinatario avviene in fase di esecuzione.

```
fetchF3_0: begin DEST_ADDR<=DP; STAR<=fetchEnd; end
```

- Formato F4 (100)
  - Categoria in cui rientrano le istruzioni dove l'operando sorgente è indirizzato in modo immediato, e sta su 8 bit.
  - L'istruzione è lunga due byte: il primo contiene l'istruzione, il secondo l'operando indirizzato in modo immediato. La fase di fetch consiste nel fare due letture consecutive.

```
fetchF4_0: begin A23_A0<=IP; IP<=IP+1; MJR<=fetchF4_1; STAR<=readB; end
fetchF4_1: begin SOURCE<=APP0; STAR<=fetchEnd; end
```

- Formato F5 (101)
  - Categoria in cui rientrano le istruzioni dove l'operando sorgente è indirizzato in modo diretto. Ciò pongo direttamente l'indirizzo del sorgente.
  - In fase di Fetch l'operando sorgente deve essere riportato nel processore.
  - L'operazione sarà lunga 4 byte: uno di opcode e tre di indirizzo di memoria (24 bit per poter rappresentare qualunque indirizzo). Seguono tre cicli di lettura consecutivi a partire da IP. Ciò non basta: devo fare un'altra lettura all'indirizzo trovato: a quel punto ho raggiunto l'operando sorgente e posso porlo nel processore.

```
fetchF5_0: begin A23_A0<=IP; IP<=IP+3; MJR<=fetchF5_1; STAR<=readM; end
fetchF5_1: begin A23_A0<={APP2,APP1,APP0}; MJR<=fetchF5_2; STAR<=readB; end
fetchF5_2: begin SOURCE<=APP0; STAR<=fetchEnd; end
```

- Formato F6 (110)
  - Categoria in cui rientrano le istruzioni dove l'operando destinatario è in memoria, indirizzato in modo diretto.
  - Il processore dovrà leggere 4 byte in memoria: uno per l'opcode, tre per l'indirizzo del destinatario.
  - La scrittura del destinatario avviene in fase di esecuzione.

```
fetchF6_0: begin A23_A0<=IP; IP<=IP+3; MJR<=fetchF6_1; STAR<=readM; end
fetchF6_1: begin DEST_ADDR<={APP2,APP1,APP0}; STAR<=fetchEnd; end
```

- Formato F7 (111)
  - Uguale al precedente, raggruppa le istruzioni di controllo (CALL, JMP, Jcon) in cui ho un indirizzo di salto.
  - Utilizzo un byte per l'opcode, altri tre per l'indirizzo. In fetch abbiamo la lettura di 4 byte consecutivi, a partire da IP.

```
fetchF7_0: begin A23_A0<=IP; IP<=IP+3; MJR<=fetchF7_1; STAR<=readM; end
fetchF7_1: begin DEST_ADDR<={APP2,APP1,APP0}; STAR<=fetchEnd; end
```

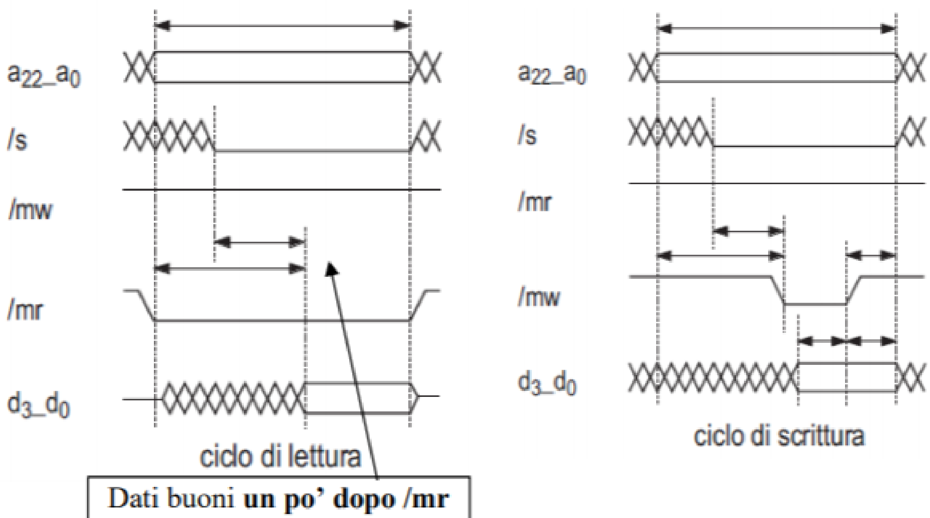
```
//--------------------------------------------------------------
// TERMINAZIONE DELLA FASE DI CHIAMATA
// TERMINAZIONE CON BLOCCO PER ISTRUZIONE NON VALIDA
// Eseguo se l'OPCODE non è valido (rivedere fetch2)
nvi: begin STAR<=nvi; end
// TERMINAZIONE REGOLARE CON PASSAGGIO ALLA FASE DI ESECUZIONE
// Individuo il primo passo da eseguire per la fase di esecuzione.
// Osservazione per bimbi spastici: perché il salto lo faccio un passo dopo?
Ricordiamo che le istruzioni sono eseguite in parallelo, non in sequenza!
fetchEnd: begin MJR<=first_execution_state(OPCODE); STAR<=fetchEnd1; end
fetchEnd1: begin STAR<=MJR; end
```

All'uscita dalla fase di fetch avrò:

- L'OPCODE che contiene il codice operativo dell'istruzione
- Se l'istruzione ha un operando sorgente immediato o in memoria questo è in SOURCE.
- Se l'istruzione ha un operando destinatario in memoria il suo indirizzo sta in DEST_ADDR.
- IP è stato incrementato e punta alla prossima istruzione da prelevare (ESECUZIONE IN PARALLELO, NON IN SEQUENZA).

42.1 Ricapitoliamo sulle operazioni di lettura e scrittura...

42.1.1 Memoria principale



- Quando scriviamo il codice Verilog dobbiamo tenere conto di quanto già visto nello spiegare la temporizzazione di scrittura e lettura in una memoria RAM statica:
  - **Lettura**:
    * Operazione non distruttiva. mr può essere abbassato fin da subito.
    * L'indirizzo può essere indicato fin da subito, ma anche dopo: bisogna tenere conto della necessità di aspettare un po' prima che i dati restituiti sui fili di dati si stabilizzino (considerare la struttura della memoria RAM statica). Ricordarsi che la select non si stabilizza immediatamente.
  - **Scrittura**:
    * Operazione distruttiva. mw non può essere abbassato subito.
    * Possiamo indicare fin da subito l'indirizzo, ma mw può essere abbassato solo dopo che si è stabilizzato l'indirizzo (se l'indirizzo non si è stabilizzato si corre il rischio di scrivere in posti sbagliati).
    * Non sono importanti i valori posti sui fili di dati all'inizio, ma quelli alla fine. Per evitare situazioni imprevedibili i dati devono rimanere costanti a cavallo del fronte di salita di mw.

- Quindi:
  - **Lettura**:

```
mem_r0: begin A23_A0<=un_indirizzo; DIR<=0; MR_<=0; STAR<=mem_r1; end
mem_r1: begin STAR<=mem_r2; end //stato di wait
mem_r2: begin QUALCHE_REGISTRO<=d7_d0; MR_<=1; end
```

    * **Primo stato**: si pone l'indirizzo dove leggere, si pone la porta tristate in alta impedenza (ricordiamoci che dobbiamo fare l'opposto di quanto avviene in memoria principale, se in memoria si ha un'operazione di lettura allora avrà la porta tristate in conduzione), si abbassa subito MR.

    * ...si attende quanto necessario affinchè si stabilizzino i dati. Abbiamo detto che supporemo che la rete è subito pronta per evitare di allungare le descrizioni
    * **Terzo stato**: si memorizza il contenuto in ingresso nel processore da qualche parte. A questo punto:
      · se abbiamo finito alziamo mr;
      · altrimenti poniamo subito un nuovo indirizzo.

  – **Scrittura**:

```
mem_w0: begin A23_A0<=un_indirizzo; D7_D0<=un_byte; DIR<=1;
STAR<=mem_w1; end
mem_w1: begin MW_<=0; STAR<=mem_w2; end
mem_w2: begin MW_<=1; STAR<=mem_w3; end
mem_w3: begin DIR<=0; ....... ; end
```

    * **Primo stato**: indichiamo l'indirizzo, i dati da scrivere nell'indirizzo, si pone la porta tristate in conduzione (stesso discorso di prima). Contrariamente a prima NON POSSIAMO abbassare subito mw.
    * ...si attende quanto necessario affinchè si stabilizzi l'indirizzo.
    * **Secondo stato**: adesso possiamo abbassare mw. L'indirizzo si è stabilizzato e non corriamo il rischio di scrivere in posti sbagliati
    * **Terzo stato**: alziamo mw. I dati a cavallo di questo fronte di salita sono quelli che saranno memorizzati.
    * **Quarto stato**: Riporto DIR a 0, quindi pongo la porta in alta impedenza. Si osservi che non è possibile fare questo prima, visto che dobbiamo mantenere i dati stabili sul fronte di salita. Dobbiamo rimettere la porta tristate ogni volta in alta impedenza, anche nel caso in cui si voglia svolgere più operazioni di scrittura: segue un numero di stati maggiori (vedere il sottoprogramma per operazioni di scrittura su più byte).

42.1.2 Sottosistema di I/O

- Presenti differenze non banali rispetto alle operazioni in memoria. Il numero di bit è minore (solo 16 bit), si utilizzano variabili di pilotaggio diverse (ior e iow, non mr e mw).

- **Lettura**:

```
io_r0: begin A23_A0<={'H00,un_offset}; DIR<=0; STAR<=io_r1; end
io_r1: begin IOR_<=0; STAR<=io_r2; end
io_r2: begin STAR<=io_r3; end //stato di wait
io_r3: begin QUALCHE_REGISTRO<=d7_d0; IOR_<=1; .... ; end
```

  – **Primo stato**: pongo l'indirizzo della locazione da leggere, pongo la porta tristate in alta impedenza (solito discorso di prima).
  – **Secondo stato**: abbasso ior. L'operazione di lettura è distruttiva, poichè può provocare nelle interfacce conseguenti operazioni di scrittura (potrei avere dati appena letti subito sovrascritti).
  – ...attendo

- **Quarto stato**: memorizzo i dati appena ricevuti in un registro e alzo ior. Contrariamente all'operazione di lettura non possiamo mettere subito indirizzi (ricordiamo che la lettura in I/O è distruttiva)

- **Scrittura**:

```
io_w0: begin A23_A0<={'H00,un_offset}; D7_D0<=un_byte; DIR<=1;
STAR<=io_w1; end
io_w1: begin IOW_<=0; STAR<=io_w2; end
io_w2: begin IOW_<=1; STAR<=io_w3; end
io_w3: begin DIR<=0; ........ ; end
```

  - **Primo stato**: indichiamo l'indirizzo, i dati da scrivere nell'indirizzo, poniamo la porta tristate in conduzione (soliti motivi di prima). Contrariamente a prima i dati devono essere già pronti sul fronte di discesa di iow: questo perchè molte interfacce memorizzano sul fronte di discesa di iow, e non sul fronte di salita.
  - **Secondo stato**: abbassiamo iow. Ricordiamo che la scrittura è distruttiva, quindi la cosa va fatta solo dopo la stabilizzazione dell'indirizzo dove scrivere.
  - **Terzo stato**: alzo iow. Valgono gli stessi discorsi di prima (soprattutto se l'interfaccia scrive in caso di fronte di salita).
  - **Quarto stato**: pongo la porta tristate in alta impedenza, visto che abbiamo finito. Non possiamo farlo prima.

42.2 Ricapitoliamo sui sottoprogrammi per la lettura/scrittura

- Si consideri che ogni volta dobbiamo indicare
  - l'indirizzo dove scrivere o leggere,
  - lo stato interno dove ritornare dopo aver eseguito il sottoprogramma (con il registro MJR), e
  - il numero di byte da leggere (che poniamo in modo implicito indicando il primo stato di partenza del sottoprogramma)
- **Lettura** (parametro di ingresso l'indirizzo):
  - **Primo stato**: pongo la porta tristate in alta impedenza (registro DIR), abbasso mr, indico il numero di locazioni da leggere (registro NUMLOC)
  - **Stati successivi**: ho, per ogni step possibile, uno stato interno. In ciascuno di essi
    * memorizzo il valore letto nel relativo registro di supporto,
    * incremento il registro contenente l'indirizzo,
    * decremento il registro NUMLOC (che funge da contatore), e
    * determino il passaggio allo step successivo o allo stato finale del sottoprogramma sulla base del valore non decrementato di NUMLOC.
  - **Stato finale**: alzo mr e indico come nuovo stato interno quello memorizzato nel registro MJR.

- **Scrittura** (parametro di ingresso l'indirizzo e i valori da memorizzare, posti nei registri APPx):
  - **Primo stato**: pongo la porta tristate in conduzione (registro DIR), indico il numero di locazioni da leggere (registro NUMLOC), indico come valore dei fili di dati quello del primo dei registri di supporto (APP0). Non posso abbassare subito mw, visto che devo avere valori stabili prima del fronte di discesa.
  - **Stati successivi**: ogni step di scrittura di un byte non può essere gestito con un solo stato interno (contrariamente alle operazioni di lettura).
    - **Primo stato step**: abbasso mw.
    - **Secondo stato step** alzo mw, e attraverso il numero di locazioni (non ancora decrementato) verifico se ho finito. Non posso modificare subito i fili di dati e i fili di indirizzo: i dati, RIBADIAMO, devono essere costanti a cavallo del fronte di salita.
    - **Terzo stato step**: se siamo a questo punto significa che dobbiamo scrivere un altro byte. Modifico i fili di dati indicando il successivo registro APPx, incremento il registro con l'indirizzo e decremento NUMLOC.
  - **Stato finale**: pongo la porta tristate in alta impedenza (registro DIR) e indico come nuovo stato interno quello memorizzato nel registro MJR.

Processore sEP8

Contenuto di alcuni registri alla fine della fase di chiamata

(F0, F1, F2, F3, F4, F5, F6, F7 sono i formati delle istruzioni di detto processore)

	OPCODE	SOURCE	DEST_ADDR	
F0,F1	codice operativo	non significativo	non significativo	
F2, F4, F5	codice operativo	operando sorgente	non significativo [1]	Indirizzamento con registro puntatore per l'operando sorgente Indirizzamento immediato per l'operando operando sorgente Indirizzamento diretto per l'operando sorgente
F3, F6	codice operativo	non significativo [2]	Indirizzo in memoria dell'operando destinatario	Indirizzamento con registro puntatore dell'operando destinatario Indirizzamento diretto per l'operando destinatario
F7	codice operativo	non significativo	Indirizzo in memoria dell'istruzione a cui saltare	Istruzioni di salto Istruzioni di chiamata dei sottoprogrammi

1 l'operando destinatario sarà immesso nel registro AL o nel registro AH

2 l'operando sorgente è nel registro AL o nel registro AH

42.4 Ricapitoliamo sugli stati della fase di fetch

Si tenga conto che al termine della fase di fetch avremo

- l'OPCODE nel registro OPCODE
- L'operando sorgente nel registro SOURCE, se il sorgente è immediato o in memoria (e non è un registro)
- L'operando destinatario nel registro DEST_ADDR, se il destinatario è in memoria (e non è un registro)

Step

- **fetch0**: Prendo l'indirizzo memorizzato nell'IP, incremento l'indirizzo memorizzato nell'IP (la cosa avviene in parallelo), indico col registro MJR che lo stato da richiamare più avanti è fetch1, avvio un'operazione di lettura del contenuto all'indirizzo appena estratto. Abbiamo ottenuto l'OPCODE.
- **fetch1**: Memorizzo nel registro OPCODE il risultato dell'operazione di lettura (un byte memorizzato nel registro di supporto APP0).
- **fetch2 e fetch3**: Utilizzo il registro MJR per gestire un salto a tante vie. Devo capire come ho gestito gli operandi, quindi quale formato ho adottato.
- Gestisco il formato adottato:
  - **F0**: si salta diretto a fetchEnd, abbiamo già gli operandi nel processore.
  - **F1**: si salta diretto a fetchEnd, rimandiamo la gestione degli operandi alla fase di esecuzione.
  - **F2**: eseguiamo un'operazione di lettura di un byte nell'indirizzo puntato dal registro DP. Si memorizza il risultato della lettura (posto in APP0) nel registro SOURCE.
  - **F3**: del destinatario non ci interessa il contenuto, ma solo l'indirizzo. Mi limito a copiare l'indirizzo puntato da DP nel registro DEST_ADDR.
  - **F4**: l'operando sorgente è posto nel byte immediatamente successivo a quello dell'OPCODE. Svolgo un'operazione di lettura di un byte nella locazione successiva, e pongo il risultato della lettura nel registro SOURCE. Ovviamente incrementiamo IP una volta in più (ricordarsi che le istruzioni sono eseguite in parallelo, non in sequenza).
  - **F5**: l'operando sorgente è un indirizzo memorizzato su 3 byte successivi. Svolgo un'operazione di lettura su 3 byte (readM) per recuperare questo indirizzo. Dopo aver recuperato l'indirizzo (memorizzato in APP2, APP1, APP0) svolgo un'ulteriore operazione di lettura (su un byte) per recuperare il contenuto della locazione puntata dall'indirizzo. Pongo il risultato di quest'ultima lettura nel registro SOURCE.
  - **F6**: svolgo un'operazione di lettura su 3 byte, come prima, per recuperare l'indirizzo posto in modo diretto. Il risultato della lettura viene posto in DEST_ADDR (ricordarsi che del destinatario ci interessa solo l'indirizzo, non il contenuto).
  - **F7**: le istruzioni di salto presentano solo l'operando destinatario. Svolgo un'operazione di lettura su 3 byte per recuperare l'indirizzo, come prima, e pongo come contenuto di DEST_ADDR l'indirizzo appena estratto.
- **fetchEnd e fetchEnd1**: conclusione della fase di fetch. Memorizzo nel registro MJR il risultato di una rete combinatoria. Questa restituisce il prossimo stato a cui saltare. Con questo salto concluso si passa alla fase di esecuzione dell'istruzione.

Capitolo 43

Mercoledì 02/12/2020

Continuiamo l'analisi della descrizione Verilog con la fase di esecuzione. La cosa sarà semplice:

- parte della complessità è stata già gestita in fase di fetch;
- la maggior parte delle istruzioni complesse sono implementate tramite reti combinatorie, quindi si riducono a un semplice assegnamento a registro.

```
//------------- istruzione NOP ----------------
// Istruzione che non fa niente. Ho un passo in più, ritorno subito al primo step di fetch.
nop: begin STAR<=fetch0; end

//------------- istruzione HLT ----------------
// Loop infinito da cui si esce solo premendo reset
hlt: begin STAR<=hlt; end

//------------- istruzione MOV AL,AH -----------
// Si assegna al registro AH il contenuto del registro AL
ALtoAH: begin AH<=AL; STAR<=fetch0; end

//------------- istruzione MOV AH,AL ----------------
// Si assegna al registro AL il contenuto del registro AH
AHtoAL: begin AL<=AH; STAR<=fetch0; end

//------------- istruzione INC DP ----------------
// Incremento il contenuto del registro DP
incDP: begin DP<=DP+1; STAR<=fetch0; end

//------------- istruzioni MOV (DP),AL       -----------
                          MOV $operando,AL
                          MOV indirizzo,AL
// Assegno al registro AL il contenuto del registro SOURCE (ricordiamo quanto detto nella scorsa lezione)
ldAL: begin AL<=SOURCE; STAR<=fetch0; end

//------------- istruzioni MOV (DP),AH       ------------
                          MOV $operando,AH
                          MOV indirizzo,AH
// Assegno al registro AH il contenuto del registro SOURCE (uguale a prima, registro destinatario diverso)
ldAH: begin AH<=SOURCE; STAR<=fetch0; end

//------------- istruzioni MOV AL,(DP)       ---------
                          MOV AL,indirizzo
// Indirizzo dell'operando destinatario nel registro DEST_ADDR. Contrariamente a prima dobbiamo svolgere
un'operazione di scrittura al di fuori del processore.
storeAL: begin A23_A0<=DEST_ADDR; APP0<=AL; MJR<=fetch0; STAR<=writeB; end

//------------- istruzioni MOV AH,(DP)      ----------
                          MOV AH,indirizzo
// Stessa cosa di prima, cambia solo il valore assegnato al registro APP0
storeAH: begin A23_A0<=DEST_ADDR; APP0<=AH; MJR<=fetch0; STAR<=writeB; end
```

- Raggruppa tutte le istruzioni che non possono essere classificate nei precedenti formati
  - Istruzioni di I/O
    - operando indirizzo a 16 bit, sorgente (IN) o destinatario (OUT)
  - MOV con uno dei registri a 24 bit (DP o SP)
    - Operando a 24 bit sorgente, sia immediato che diretto, o destinatario
- Le azioni per procurarsi gli operandi sono diverse da un'istruzione all'altra
  - Meglio fare una fase di fetch «scarna» in cui prelievo solo l'opcode
  - Gestiremo gli operandi successivamente in fase di esecuzione
  - Poco pulito dal punto di vista concettuale, ma molto più semplice

73

```
//---------- istruzione MOV $operando,SP ---------
// L'operando SP è a 24bit. IP è stato incrementato di 1 quando arriviamo qua. Incremento di tre andando al primo bit che non mi interessa. Svolgo un'operazione di lettura readM (3 byte). Come indirizzo da leggere ho l'Instruction Pointer, come MJR lo stato successivo in cui utilizzo i valori letti. L'unione dei tre registri modificati con l'operazione di lettura consiste nel valore dell'SP (In ordine decrescente, ricordiamo dove stanno le cifre più significative e quelle meno significative).
ldSP: begin A23_A0<=IP; IP<=IP+3; MJR<=ldSP1; STAR<=readM; end
ldSP1: begin SP<={APP2,APP1,APP0}; STAR<=fetch0; end

//---------- istruzione MOV $operando,DP --------
// Anche in questo caso abbiamo un operando (quello sorgente) a 24 bit. Facciamo le stesse cose di prima, cambia solo l'assegnamento a DP invece che ad SP.
ldimmDP: begin A23_A0<=IP; IP<=IP+3; MJR<=ldimmDP1; STAR<=readM; end
ldimmDP1: begin DP<={APP2,APP1,APP0}; STAR<=fetch0; end

//------------- istruzione MOV indirizzo,DP -------
// Devo leggere l'indirizzo, quindi svolgo una lettura di 3 byte. Dopo aver estratto l'indirizzo svolgo un'ulteriore operazione di lettura per leggere il contenuto dell'indirizzo. Pongo il contenuto dell'indirizzo come nuovo valore del registro DP.
lddirDP: begin A23_A0<=IP; IP<=IP+3; MJR<=lddirDP1; STAR<=readM; end
lddirDP1: begin A23_A0<={APP2,APP1,APP0}; MJR<=lddirDP2; STAR<=readM; end
lddirDP2: begin DP<={APP2,APP1,APP0}; STAR<=fetch0; end

//------------- istruzione MOV DP,indirizzo -----
// Mi devo procurare l'indirizzo (l'operando destinatario) con un'operazione di lettura su 3 byte. Successivamente svolgo un'operazione di scrittura sull'indirizzo trovato. Questa istruzione, suggeriva qualcuno, potrebbe essere accorpata ad F6: mi sbarazzo del primo step e nel secondo assegno DEST_ADDR invece di {APP2, APP1, APP0}
storeDP: begin A23_A0<=IP; IP<=IP+3; MJR<=storeDP1; STAR<=readM; end
storeDP1: begin A23_A0<={APP2,APP1,APP0}; {APP2,APP1,APP0}<=DP; MJR<=fetch0;
STAR<=writeM; end

//------------- istruzione IN offset,AL ---------
// Per prima cosa devo procurarmi l'offset: svolgiamo un'operazione di lettura su 2 byte a partire dall'IP. Dopo la lettura prendo i primi 16 bit (gli unici che mi interessano). A questo punto devo fare una lettura nello spazio di I/O
in: begin A23_A0<=IP; IP<=IP+2; MJR<=in1; STAR<=readW; end

// Preparo l'indirizzo, allo step successivo abbasso IOR_ (DOPO), nello step finale prendo i dati letti e li metto nel registro. Alzo IOR_ visto che ho finito l'operazione.
in1: begin A23_A0<={'H00,APP1,APP0}; STAR<=in2; end
in2: begin IOR_<=0; STAR<=in3; end
in3: begin AL<=d7_d0; IOR_<=1; STAR<=fetch0; end

//------------- istruzione OUT AL,offset ---------
// Leggo l'offset e svolgo operazione di lettura. Ricordarsi che non possiamo abbassare DIR e alzare IOW_ in contemporanea (altrimenti abbiamo problemi con la porta tristate). Il comando di scrittura nello spazio di I/O è il fronte di discesa.
out: begin A23_A0<=IP; IP<=IP+2; MJR<=out1; STAR<=readW; end
out1: begin A23_A0<={'H00,APP1,APP0}; D7_D0<=AL; DIR<=1; STAR<=out2; end
out2: begin IOW_<=0; STAR<=out3; end
out3: begin IOW_<=1; STAR<=out4; end
out4: begin DIR<=0; STAR<=fetch0; end

//------- istruzioni ADD (DP),AL -------
                      ADD $operando,AL
                      ADD indirizzo,AL
                      SUB (DP),AL
                      SUB $operando,AL
```

```
                    SUB indirizzo,AL
                    AND (DP),AL
                    AND $operando,AL
                    AND indirizzo,AL
                    OR (DP),AL
                    OR $operando,AL
                    OR indirizzo,AL
                    CMP (DP),AL
                    CMP $operando,AL
                    CMP indirizzo,AL
                    NOT AL
                    SHR AL
                    SHL AL
// Tutte le istruzioni elencate possono essere
collassate nel seguente codice. Si assegna ad AL il risultato di un'operazione
combinatoria dove indico OPCODE, SOURCE, AL. Dall'OPCODE capisco quale operazione
dovrà essere svolta, gli altri due consistono negli operandi. Le operazioni possibili
sono ADD, SUB, AND, OR, CMP, NOT, SHL, SHR: segue un multiplexer a otto vie con
variabili di comando generate dall'OPCODE (Rete combinatoria dipendente
dall'implementazione scelta dell'OPCODE).
Dobbiamo tener conto anche dell'aggiornamento dei flag: ricordiamo che il registro
dei flag consiste in un insieme di 8 bit di cui i primi 4 importanti. Attraverso
l'assegnamento non bloccante aggiorniamo i flag nelle posizioni 0,1,2,3
(OF,SF,ZF,CF): facciamo in questo modo visto che gli altri flag, in queste
operazioni, sono insignificanti.
aluAL: begin
AL <= alu_result(OPCODE,SOURCE,AL);
F<={F[7:4], alu_flag(OPCODE,SOURCE,AL)};
STAR<=fetch0;
end

//----- istruzioni ADD (DP),AH --
                  ADD $operando,AH
                  ADD indirizzo,AH
                  SUB (DP),AH
                  SUB $operando,AH
                  SUB indirizzo,AH
                  AND (DP),AH
                  AND $operando,AH
                  AND indirizzo,AH
                  OR (DP),AH
                  OR $operando,AH
                  OR indirizzo,AH
                  CMP (DP),AH
                  CMP $operando,AH
                  CMP indirizzo,AH
                  NOT AH
                  SHL AH
                  SHR AH
// In modo del tutto speculare facciamo le stesse cose di prima con AH operando
destinatario.
aluAH: begin
AH<=alu_result(OPCODE,SOURCE,AH);
F<={F[7:4],alu_flag(OPCODE,SOURCE,AH)};
STAR<=fetch0;
end

//----- istruzioni JMP indirizzo -----------
                  JA indirizzo
                  JAE indirizzo
                  JB indirizzo
                  JBE indirizzo
```

```
                    JC indirizzo
                    JE indirizzo
                    JG indirizzo
                    JGE indirizzo
                    JL indirizzo
                    JLE indirizzo
                    JNC indirizzo
                    JNE indirizzo
                    JNO indirizzo
                    JNS indirizzo
                    JNZ indirizzo
                    JS indirizzo
                    JO indirizzo
                    JZ indirizzo
```

// Tutte le istruzioni di salto consistono nell'impostare un nuovo valore dell'Instruction pointer. La fase di fetch mi ha portando l'indirizzo nel DEST_ADDR.
Attraverso l'OPCODE capisco se la condizione di JUMP è vera (ovviamente se ho la JMP la condizione sarà sempre vera), quindi se devo fare il salto.

```
jmp: begin
IP<=(jmp_condition(OPCODE,F)==1)?DEST_ADDR:IP;
STAR<=fetch0;
end

//------------- istruzione PUSH AL ----------------
```

// La push è una scrittura in memoria all'indirizzo del nuovo top della pila. Se SP punta al top della pila allora decremento per andare alla posizione successiva.

```
pushAL: begin
A23_A0<=SP-1;
SP<=SP-1;
APP0<=AL;
MJR<=fetch0;
STAR<=writeB;
end
```

Il decremento dipende dalla dimensione degli elementi di una pila (quanti byte?)

```
//------------- istruzione PUSH AH ----------------
```

// Ibidem

```
pushAH: begin
A23_A0<=SP-1;
SP<=SP-1;
APP0<=AH;
MJR<=fetch0;
STAR<=writeB;
end

//------------- istruzione PUSH DP ----------------
```

// Ibidem

```
pushDP: begin
A23_A0<=SP-3;
SP<=SP-3;
{APP2,APP1,APP0}<=DP;
MJR<=fetch0;
STAR<=writeM;
end

//------------- istruzione POP AL ----------------
```

// Letture in memoria all'indirizzo puntato da SP. Ricordiamoci che dobbiamo incrementare in modo da rendere consistenti le letture successive: questo incremento dipende dal numero di byte letti.

```
popAL: begin
A23_A0<=SP;
SP<=SP+1;
```

```
MJR<=popAL1;
STAR<=readB;
end
popAL1: begin AL<=APP0; STAR<=fetch0; end

//------------- istruzione POP AH ----------------
// Ibidem
popAH: begin
A23_A0<=SP;
SP<=SP+1;
MJR<=popAH1;
STAR<=readB;
end
popAH1: begin AH<=APP0; STAR<=fetch0; end

//------------- istruzione POP DP ----------------
// Ibidem
popDP: begin
A23_A0<=SP;
SP<=SP+3;
MJR<=popDP1;
STAR<=readM;
end
popDP1: begin DP<={APP2,APP1,APP0}; STAR<=fetch0; end

//------------- istruzione CALL indirizzo ---------
// La CALL è una JMP preceduta da una scrittura in memoria, se volete.
Nella pila devo salvare 3 byte di IP. Quindi scrivo 3 byte in memoria
all'indirizzo SP-3. Assegno ad IP il DESTINATION ADDRESS.
call: begin
A23_A0<=SP-3;
SP<=SP-3;
{APP2,APP1,APP0}<=IP;
MJR<=call1;
STAR<=writeM;
end
call1: begin IP<=DEST_ADDR; STAR<=fetch0; end
```

IP 7:0
IP 15:8
IP 23:16
SP → ...
...

```
//------------- istruzione RET ----------------
// Si ha una lettura in memoria della pila e la sostituzione di IP con
quanto letto.
ret: begin
A23_A0<=SP;
SP<=SP+3;
MJR<=ret1;
STAR<=readM;
end
ret1: begin IP<={APP2,APP1,APP0}; STAR<=fetch0;
end
```

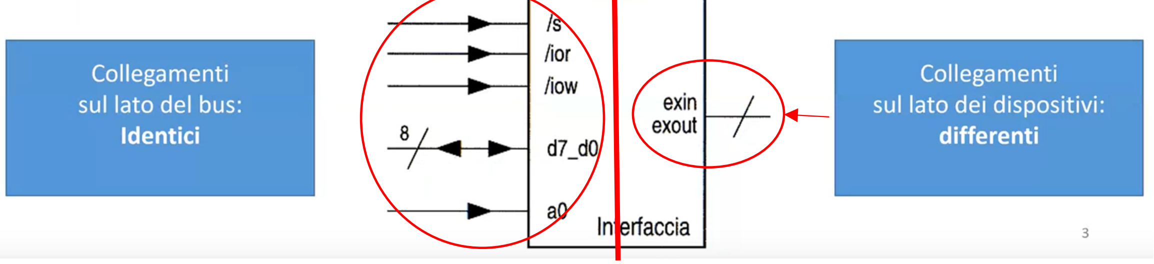
SP → IP 7:0
IP 15:8
IP 23:16
...
...

- **Conclusioni**:
  - In ogni stato della descrizione appena visto sono presenti al più micro-salti a due vie.
  - Se vogliamo essere precisi la maggior parte dei micro-salti sono a una solo via (micro-salti incondizionati). I micro-salti a due sono presenti quasi esclusivamente nelle sottoliste di lettura/scrittura in memoria.
  - Il processore è sintetizzabile con il metodo di scomposizione *Parte operativa – Parte controllo* visto a lezione. Le reti combinatorie contengono solo cose già viste a lezione. Non sono difficili da sintetizzare, il problema è solo la noia.
  - **Riflessione**: in soli due mesi di corso abbiamo acquisito la capacità di descrivere e sintetizzare dell'hardware capace di eseguire programmi software arbitrariamente complessi.

Interfacce

- A questo punto ci mancano le varie interfacce che completano il calcolatore.
- Vedremo interfacce di tre tipi:
  - interfacce parallele, quelle che colloquiano con dispositivi ai quali si invia un byte alla volta;
  - interfacce seriali, quelle che colloquiano con dispositivi con i quali si scambia un bit alla volta;
  - quelle per la conversione A-D e D-A, che trasformano gruppi di bit in tensioni e viceversa. Utilizzeremo la tensione come grandezza analogica.

- Le interfacce sono dispositivi collocati tra il bus e i dispositivi.



  - Per capire di quale interfaccia si parla dobbiamo guardare il lato dei dispositivi, diverso da interfaccia a interfaccia.
  - Dal lato del processore le interfacce sono tutte uguali e si presentano come piccole memorie.
  - Chiaramente se ho un solo filo di indirizzo avrò solo due porte: una di ingresso e una di uscita.
  - I fili di indirizzo rimanenti passano in una maschera che genera il /s.

<u>Descrizione dell'interfaccia a livello funzionale</u>:

- <u>Chi usa l'interfaccia (un sistemista per il montaggio, o un programmatore) come la deve usare?</u>

  - Nell'interfaccia sono presenti due registri:
    - RBR (*Receive Buffer Register*), dove salvo i dati scritti dal dispositivo esterno;
    - TBR (*Transmit Buffer Register*)., dove salvo i dati da mandare al dispositivo esterno.
  - L'indirizzo interno a_0 mi permette di indicare quale registro mi interessa (0 per *RBR* e 1 per *TBR*).

  - La lettura del registro RBR consiste nell'istruzione Assembler IN
    ```
    IN offset_RBR, %AL
    ```
  - La scrittura nel registro TBR consiste nell'istruzione Assembler OUT
    ```
    OUT %AL, offset_TBR
    ```
  - **Problema**: sincronizzazione tra processore e dispositivo. Il processore non ha modo di sincronizzarsi con i dispositivi (anche perché li vede solo come memorie)
    - Supponiamo che un programma contenga le seguenti istruzioni
      ```
      IN offset_RBR, %AL
      …
      IN offset_RBR, %AL
      ```
      Nessuno può garantirmi che tra le due IN il dispositivo sia stato in grado di produrre un dato nuovo. Il secondo dato potrebbe non essere significativo.
    - Dualmente…
      ```
      OUT %AL, offset_TBR
      …
      OUT %AL, offset_TBR
      ```

Nessuno può garantire che tra le due OUT il dispositivo abbia processato il dato.

Per risolvere questo problema dobbiamo dotarci di registri di stato per implementare un handshake tra processore e dispositivi. I due registri sono i seguenti:

- RSR (*Receive Status Register*),
- TSR (*Transmit Status Register*).

Molto spesso collassati in un unico registro *RTSR*. Di ciascun registro è significativo un solo bit, rispettivamente i seguenti:

- *Flag di buffer ingresso pieno* (FI)
- *Flag di buffer di uscita vuoto* (FO)

I flag sono gestiti dall'interfaccia che li setta e resetta quando capta certi eventi.

- **Registro FI**:
  - Il flag è inizialmente a 0.
  - L'interfaccia lo mette a 1 quando il dispositivo scrive un dato in RBR, segnalando la presenza di un nuovo dato
  - Quando il processore accede in lettura al registro RBR, l'interfaccia porta a 0 il flag FI.

Scrittura di un dato in RBR da parte del dispositivo

Lettura di RBR da parte del processore

FI

  - Un sottoprogramma Assembler che legge dati nuovi da un'interfaccia con handshake, mettendoli dentro AL, è il seguente

```
testFI: IN RSR_offset,%AL      # Copia in AL il contenuto di RSR
        AND $0x01,%AL          # Evidenzia in AL il contenuto di FI
        JZ testFI              # cicla finché FI vale 0
        IN RBR_offset,%AL      # Copia in AL il contenuto di RBR
        RET                    # Ritorna al chiamante
```

- **Registro FO**:
  - Il flag inizialmente è a 1.
  - L'interfaccia lo mette a 0 quando il processore scrive un dato in TBR (istruzione OUT), segnalando che il dispositivo non ha ancora processato.
  - Quando il dispositivo (con i suoi tempi) accede al registro TBR e legge il dato, l'interfaccia porta nuovamente a 1 il flag FO.

  - Un sottoprogramma Assembler che scrive il contenuto di AL dentro TBR in un'interfaccia con handshake è il seguente

```
        PUSH %AL              # Salva in pila il contenuto di AL
testFO: IN TSR_offset, %AL    # Copia in AL il contenuto di TSR
        AND $0x20,%AL         # Evidenzia in AL il contenuto FO
        JZ testFO             # Salta indietro se FO era a 0
        POP %AL               # Ripristina il contenuto di AL
        OUT %AL,TBR_offset    # Immette in TBR il contenuto di AL
        RET                   # Ritorna al chiamante
```

Ma tutto questo è efficiente? Per nulla: l'idea è che il processore rimanga in attesa, cioè che cicli finchè il dispositivo esterno non sarà pronto. Il processore perde tempo, considerando la lentezza delle reti che comunicano con lui.

Molto meglio se il processore può andare avanti per conto proprio, con l'interfaccia che segnala al processore quando è pronto. A quel punto il processore interrompe il lavoro che sta facendo.

Direct memory access (DMA): il processore demanda ad un'altra unita (il DMA controller) il compito di trasferire dati tra la memoria e le interfacce, mentre va avanti con le sue elaborazioni. Il processore riferisce al DMA l'area di memoria e l'interfaccia con cui lavorare.

Capitolo 44

Giovedì 03/12/2020

- Abbiamo già detto che nel ciclo di lettura dello spazio di I/O non è possibile settare l'indirizzo in simultanea ad /ior (che viene abbassato per indicare l'operazione di lettura).

lettura
A23_AØ <= _ _ ; IOR_ <= Ø ;

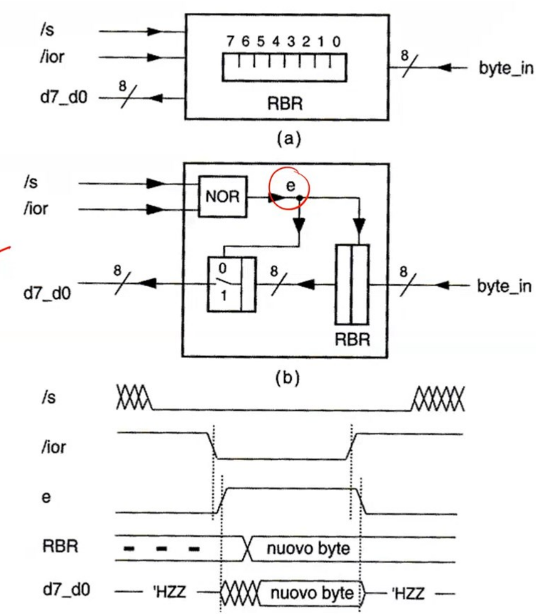
- **Perché non possiamo farlo?**
    - Sappiamo che /s è un'uscita combinatoria che può ballare.
    - Se questo piedino balla con /ior a zero può succedere che si indichi un comando di lettura del registro RBR. L'interfaccia si sincronizza col dispositivo a valle per indicare che il processore ha letto il registro e che quindi può essere sovrascritto. Leggere un registro, in alcuni casi, significa dire al relativo dispositivo di svolgere delle operazioni.
    - Segue che /ior deve essere abbassato solo ed esclusivamente dopo che gli indirizzi si sono stabilizzati.

Interfacce parallele

Interfaccia parallela di ingresso senza handshake

- L'interfaccia presenta un registro RBR, un /s, un /ior, i fili di dati d7_d0. Dal lato del dispositivo abbiamo otto fili con cui viene indicato il byte in ingresso.
- Non avendo operazioni in uscita non è necessario possedere un registro TBR e la variabile /iow.

- **Come è fatta l'interfaccia internamente?**
    - Abbiamo un registro RBR che salva i dati in ingresso.
    - Il clock deve essere fornito sulla base degli accessi fatti all'interfaccia. Quando il processore vuole accedere il registro campiona!
    - Il comando di memorizzazione non sarà dato col solito generatore di impulsi: esprimiamo un comando di memorizzazione quando il processore vuole accedere a questa interfaccia.
    - La cosa può essere ottenuta attraverso una porta NOR che ha in ingresso /s ed /ior: restituisco 1 soltanto se entrambe sono abbassate a zero. Se esprimo 1 le porta tristate viene abbassate e il dato in ingresso campionato.

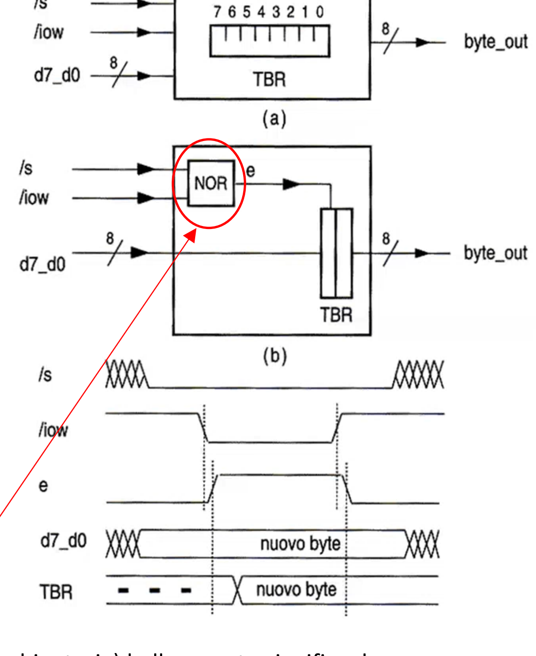


    - **Temporizzazione**:
        - Si stabilizzano gli indirizzi
        - Abbasso /ior a zero: con un leggero ritardo dovuto alla porta NOR le tri-state vanno in conduzione e il registro RBR campiona il nuovo byte.
        - Dopo un tempo di propagazione RBR presenterà il nuovo byte.
        - Non appena /ior ritorna ad 1 le tristate vanno in alta impedenza.

    - **Descrizione Verilog** (un po' sintesi-oriented, cit.):

```
module Interfaccia_Parallela_di_Ingresso(d7_d0,s_,ior_,byte_in);
 input s_,ior_;
 output[7:0] d7_d0;
 input[7:0] byte_in;
 reg[7:0] RBR;
 wire e; assign e=({s_,ior_}=='B00)?1:0; //e=~(s_|ior_)
assign d7_d0=(e==1)?RBR:'HZZ;
 always @(posedge e) #3 RBR<=byte_in;
endmodule
```

Interfaccia parallela di uscita senza handshake



- L'interfaccia presenta un solo registro TBR. Non ho bisogno di un indirizzo visto che la porta è 1, ho solo /s ed /iow. Dalla parte del dispositivo ho 8 fili in uscita.

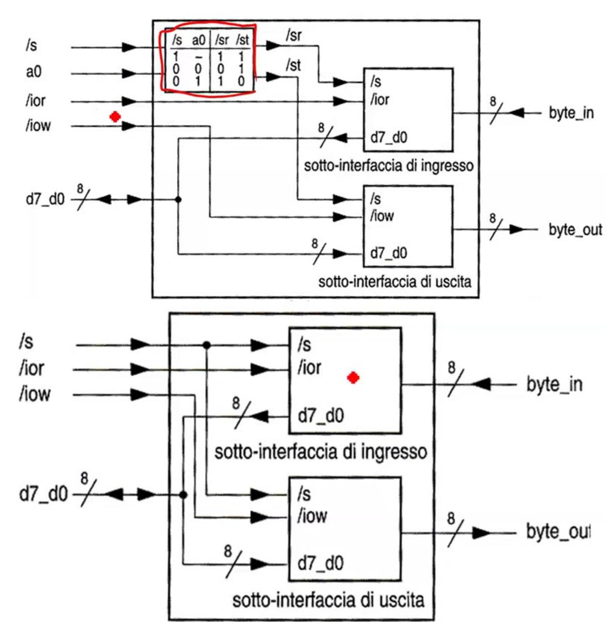
- All'interno dell'interfaccia ho una porta NOR che esprime il clock per il registro TBR.
- Non sono necessarie porte tri-state, ovviamente.
- Ricordiamo che il comando di memorizzazione è il fronte di discesa e non quello di salita.
- I dati devono essere pronti prima del fronte di discesa.
- All'arrivo del clock, con un po' di ritardo, i fili di uscita si adegueranno al nuovo byte.

- **Perché non posso fare un'interfaccia che memorizza sul fronte di salita?**
  - Per prima cosa avrei bisogno di una porta AND avente in ingresso /s negato ed /iow. Restituisco 1 solo con /s uguale a zero e /iow uguale ad 1 (cioè quando /iow viene alzato).
  - Questa cosa non funziona: l'uscita /s (uscita combinatoria) balla, questo significa dare continuamente comandi di memorizzazione al registro. Con la porta NOR siamo certi che daremo il comando di memorizzazione solo dopo la stabilizzazione di /s.

- **Descrizione in Verilog**:

```
module Interfaccia_Parallela_di_Uscita(d7_d0,s_,iow_,byte_out);
 input s_,iow_;
 input[7:0] d7_d0;
 output[7:0] byte_out;
 reg[7:0] TBR; assign byte_out=TBR;
 wire e; assign e=({s_,iow_}=='B00)?1:0; //e=~(s_|iow_)
 always @(posedge e) #3 TBR<=d7_d0;
endmodule
```

Montaggio di interfacce parallele di ingresso e uscita



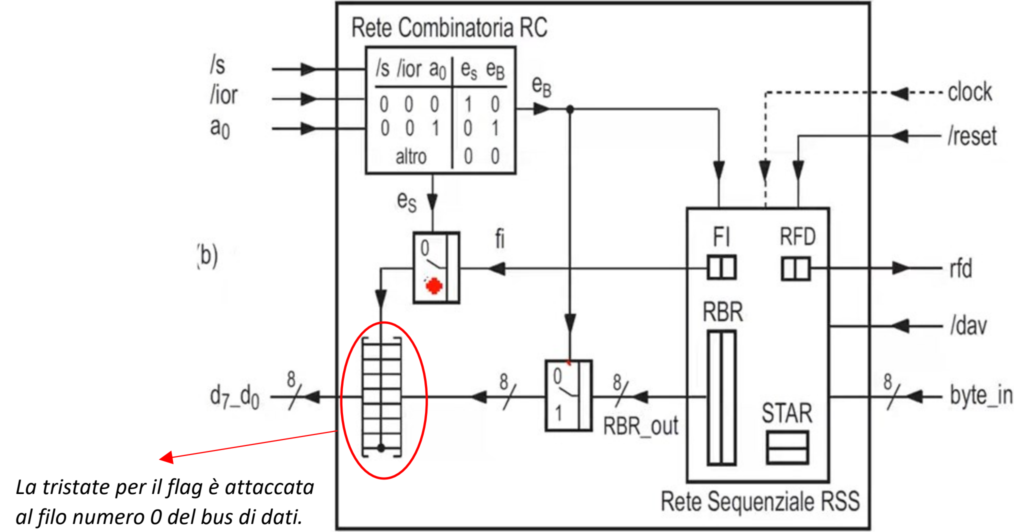
- Vogliamo realizzare un'interfaccia sia di ingresso che di uscita.

- **Primo metodo**:
  - Poniamo la porta di ingresso all'indirizzo pari (0) e quella di uscita all'indirizzo dispari (1). Serve un minimo di logica combinatoria per produrre due select. I meccanismi utilizzati sono i soliti.

- **Secondo metodo**:
  - Le due porte sono allo stesso offset.
  - Il programmatore le riferisce con lo stesso indirizzo nello spazio di I/O, e l'accesso dipende dal tipo di accesso: è immediato dalle regole di pilotaggio che solo una delle due sotto-interfacce sarà abilitata a fare qualcosa.

Interfaccia parallela di ingresso con handshake

- L'interfaccia presenta due registri: il registro RBR, come buffer per il contenuto posto in ingresso dal dispositivo, e il registro RSR, che presenta il flag di ingresso pieno FI.
- L'interfaccia da la possibilità di svolgere operazioni di lettura su entrambi i registri.
- Nella RSS viene gestito un handshake: il dispositivo è il produttore, l'interfaccia il consumatore.



- All'interno abbiamo una rete combinatoria che deve generare i segnali di abilitazione per le porte tri-state (e_B ed e_S), nel caso in cui il processore voglia svolgere operazioni di lettura.
- All'interno dell'interfaccia abbiamo anche una RSS che gestisce l'handshake col dispositivo e setta/resetta il flag FI. Per gestire l'handshake abbiamo bisogno del valore di e_B in ingresso nella RSS: quando e_b assume come valore 1 significa che il processore sta svolgendo un'operazione di lettura sul registro RBR.

Per quanto riguarda la disposizione delle porte tristate:

- Le tristate non sono mai in conduzione contemporaneamente:
  - Quando $e_s = 1, e_b = 0$ abbiamo la tristate relativa al flag FI in conduzione. Tutte le porte tristate relative al registro RBR sono in alta impedenza (vedere le cose a tre dimensioni…)
  - Quando $e_s = 0, e_b = 1$ le tristate relative al registro RBR sono in conduzione. La porta tristate relativa al flag FI si trova in alta impedenza.
- Quando $e_s = 0, e_b = 0$ non stiamo svolgendo operazioni di lettura, e tutte le porte tristate sono in alta impedenza.

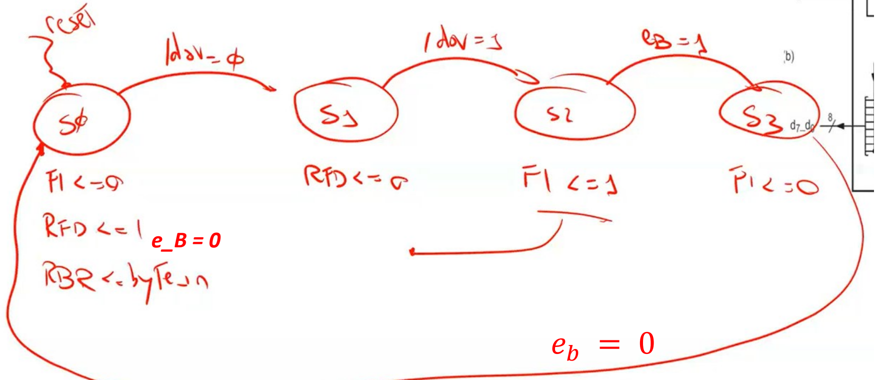
Sintetizziamo la RSS:

- Di quali registri ho bisogno? Quelli già detti, oltre al registro STAR per tenere conto della sequenza di stati.

- **Condizioni di reset**:
  - I valori per gestire l'handshake sono quelli già noti (rfd e /dav uguali ad 1)
  - Il flag di ingresso pieno va a 0 perché nessuno, al momento del reset, ha scritto qualcosa.
  - Il contenuto di RBR non è significativo e quindi non necessita di essere inizializzato.
  - Il registro STAR avrà come valore iniziale quello relativo al primo stato interno.

- **Descrizione delle operazioni negli stati** (immagine a pagina dopo):
  - **S0**: stato di riposo in cui attendiamo che il produttore (il dispositivo) fornisca un dato (cioè attendo che /dav venga abbassato). Ogni volta aggiorno il valore del registro RBR, in modo tale da averlo già pronto nello stato S1.
  - **S1**: Metto rfd a 0 per indicare che il consumatore (cioè l'interfaccia) ha ricevuto i dati e li sta elaborando. Attendo che /dav venga nuovamente alzato (gestione dell'handshake, necessario).
  - **S2**: metto il flag di ingresso pieno uguale ad 1. Rimango in questo stato finchè il processore non svolgerà un'operazione di lettura nel buffer RBR (cioè finchè non avrò $e_b = 1$).

- **S3**: il processore sta svolgendo un'operazione di lettura. Posso porre il flag di ingresso pieno a zero. Rimango in questo stato finchè l'operazione di lettura non sarà completata (il segnale è $e_b = 0$). A quel punto possiamo ritornare allo stato iniziale, ponendoci in attesa di nuovi input da parte del dispositivo esterno.



- **Descrizione in Verilog**:

```
module RSS(dav_,rfd,byte_in,fi,RBR_out,eB,clock,reset_);
 input clock,reset_; wire clock_RSS; assign #5 clock_RSS=clock;
 input dav_,eB;
 output rfd,fi;
 input[7:0] byte_in;
 output[7:0] RBR_out;

 reg RFD; assign rfd=RFD;
 reg FI; assign fi=FI;
 reg[7:0] RBR; assign RBR_out=RBR;

 reg[1:0] STAR; parameter S0=0,S1=1,S2=2,S3=3;
 always @(reset_==0) #1 begin RFD<=1; FI<=0; STAR<=S0; end
 always @(posedge clock_RSS) if (reset_==1) #3
   casex(STAR)
   //Handshake con il dispositivo esterno con immissione del
   //nuovo byte in RBR
   S0: begin RFD<=1; RBR<=byte_in; STAR<=(dav_==1)?S0:S1; end
   S1: begin RFD<=0; STAR<=(dav_==0)?S1:S2; end

   //Messa a 1 del contenuto di FI ed attesa che il processore
   //inizi la fase di esecuzione dell'istruzione IN RBR_offset,AL;
   //messa a 0 del contenuto di FI e passaggio allo stato interno
   //successivo non appena tale fase ha inizio
   S2: begin FI<=(eB==0)?1:0; STAR<=(eB==0)?S2:S3; end
```

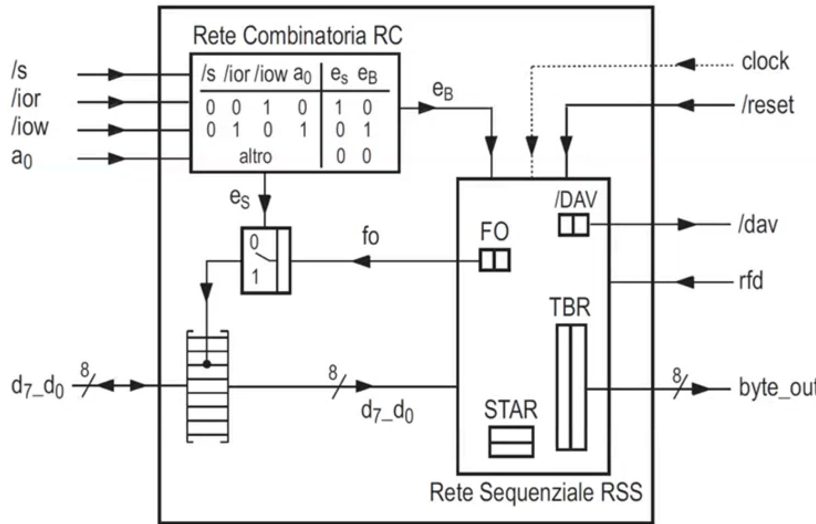
> Attenzione al clock. Stea nell'immagine aggiorna il flag con un clock di ritardo rispetto alla variazione di e_B. Se uno vuole fare il precisino può impostare l'aggiornamento del registro FI in questo modo e in S2.

```
   //Ritorno allo stato interno iniziale quando il processore
   //termina la fase di esecuzione dell'istruzione IN RBR_offset,AL
   S3: begin STAR<=(eB==1)?S3:S0; end
 endcase
endmodule
```

- Attenzione al filo del clock interno, che è il filo del clock del bus leggermente ritardato. Questa cosa non è rilevante, ma utile per capire se abbiamo lavorato correttamente.

Interfaccia parallela di uscita con handshake

- L'interfaccia presenta due registri: il registro TSR, che presenta il flag di uscita vuoto FO (in posizione 5), e il registro TBR, come buffer per il contenuto che sarà letto dal dispositivo.
- Contrariamente a prima abbiamo sia il comando di lettura /ior che quello di scrittura /iow. Con la rete vogliamo svolgere:
    - Operazioni di lettura sul flag FO, per verificare se il dispositivo ha indotto la modifica del flag leggendo il contenuto.
    - Operazioni di scrittura sul registro TBR, per trasmettere al dispositivo un nuovo byte.
- Abbiamo un meccanismo di handshake, ma con ruoli invertiti rispetto a prima: l'interfaccia è il produttore, il dispositivo il consumatore.



- Abbiamo una rete combinatoria che produce l'ingresso di abilitazione per una porta tristate, e l'uscita e_b. e_s è uguale a 1 se vogliamo svolgere una lettura del flag FO, zero altrimenti. e_b non è più utilizzato per una porta tristate, ma rimane per permettere alla RSS di capire se il processore ha eseguito un'operazione di lettura sul registro TBR.

Per quanto riguarda la disposizione delle porte tristate:

- A differenza di prima abbiamo una sola porta tristate: essa è in conduzione con $e_s = 1$, cioè se vogliamo svolgere un'operazione di lettura sul registro TSR.
    - Se la porta è in conduzione non avremo valori in ingresso coi fili di dati, ma avremo l'utilizzo del filo in posizione 5 come unico filo di uscita.

Descrizione:

```
module RSS(dav_,rfd,byte_out,fo,d7_d0,eB,clock,reset_);
 input clock,reset_; wire clock_RSS; assign #5 clock_RSS=clock;
 input rfd,eB;
 output dav_,fo;
 output[7:0] byte_out;
 input[7:0] d7_d0;

 reg DAV_; assign dav_=DAV_;
 reg FO; assign fo=FO;
 reg[7:0] TBR; assign byte_out=TBR;
 reg[1:0] STAR; parameter S0=0,S1=1,S2=2,S3=3;

 always @(reset_==0) #1 begin DAV_<=1; FO<=1; STAR<=S0; end
 always @(posedge clock_RSS) if (reset_==1) #3
   casex(STAR)
     //Messa a 1 del contenuto di FO ed attesa che il processore
     //inizi la fase di esecuzione dell'istruzione OUT AL,TBR_offset;
     //messa a 0 del contenuto di FO, immissione finale in TBR del byte
```

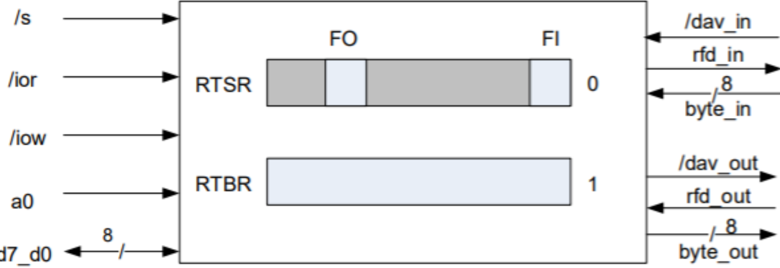
```
      //inviato dal processore tramite d7_d0 e passaggio allo stato
      //interno successivo, il tutto non appena tale fase ha inizio
      S0: begin FO<=(eB==0)?1:0; TBR<=d7_d0; STAR<=(eB==0)?S0:S1; end

      //Attesa che il processore termini la fase di esecuzione della
      //istruzione OUT AL,TBR_offset
      S1: begin STAR<=(eB==1)?S1:S2; end

      //Handshake con il dispositivo esterno con invio del byte contenuto
      //in TBR e conseguente ritorno allo stato interno iniziale
      S2: begin DAV_<=0; STAR<=(rfd==1)?S2:S3; end

      S3: begin DAV_<=1; STAR<=(rfd==0)?S3:S0; end
    endcase
endmodule
```

- **S0**: Sono in uno stato di attesa. Attendo che il processore esegua operazioni di scrittura sul TBR (me ne accorgo con $e_b = 1$). Aggiorno fin da subito il buffer per avere il contenuto già pronto in S1.
- **S1**: Attendo che l'operazione di scrittura venga conclusa (me ne accorgo con $e_b = 0$).
- **S2**: Pongo il flag di uscita vuota a zero poco prima di passare in S3. Abbasso /dav. Attendo che rfd venga posto a zero prima di passare in S3 (gestione dell'handshake, ricordare le regole).
- **S3**: Alzo /dav, e ritorno in S0 non appena avrò rfd a uno. A quel punto sono certo che il dispositivo esterno abbia concluso l'elaborazione del dato.

Interfaccia parallela di ingresso-uscita

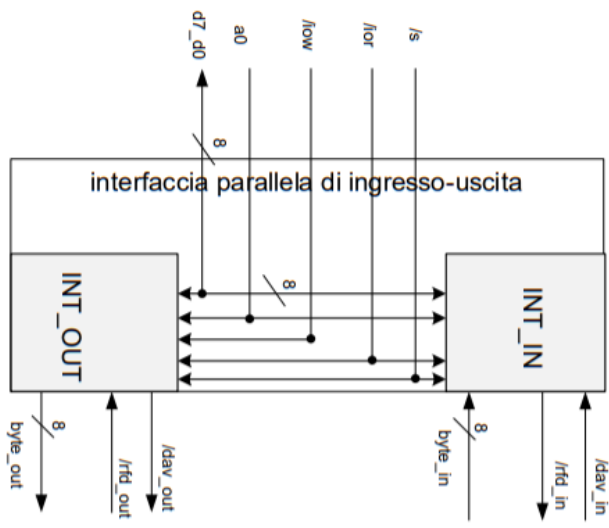
Le due interfacce con handshake appena introdotte possono essere connesse in un'unica interfaccia parallela con handshake d ingresso-uscita. Si consideri che:

- I registri RBR e TBR sono mappati su un unico indirizzo interno (1). Agiremo su un registro o su un altro in base all'operazione indicata con gli attivi bassi.



- Possiamo svolgere:
    - Operazioni di lettura sul registro RTSR (unione dei due registri già presentati coi flag). Il flag di uscita vuota è in posizione 5, mentre il flag di ingresso pieno in posizione 0.
    - Operazioni di lettura su RBR.
    - Operazioni di scrittura su TBR.

Osservazione: ci sono problemi se /s=0, /ior=0, a_0=0? No perché vanno in conduzione, in entrambi i moduli, le tristate relative ai flag, e questi vengono posizionati in posti differenti (posizione 5 e 0).



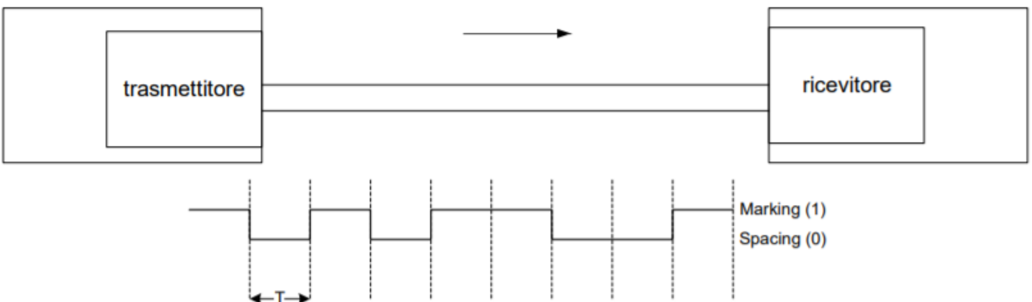
Interfacce seriali

- Un'interfaccia seriale è un'interfaccia dove la trasmissione dei singoli bit avviene in modo seriale. Con "in modo seriale" si intende la trasmissione di un bit alla volta, dal meno significativo, al più significativo. L'interfaccia:
    - riceve byte dal bus e trasmette all'esterno sequenze di bit (serializzazione);
    - riceve dall'esterno sequenze di bit e presenta byte al processore componendo quelle sequenze di bit in un registro (de-serializzazione).
- Le interfacce viste fino ad ora sono, teoricamente, seriali, in quanto permettono di trasmettere una serie di byte. La differenza sta nel fatto che qua serializziamo il singolo byte.

- Un PC, di norma (oggi sempre meno), presenta almeno una porta seriale. La cosa è utilizzata per modem e un tempo pure per i mouse (pensiamo ai mouse di una volta, quelli con la pallina, che presentano un connettore non-USB).
- Perché esistono le interfacce seriali: un tempo i calcolatori consistevano in grossi elaboratori (cervello) che venivano connessi a terminali stupidi (mani). La connessione tra terminale stupido ed elaboratore intelligente avveniva, appunto, mediante linee seriali.

Comunicazione seriale

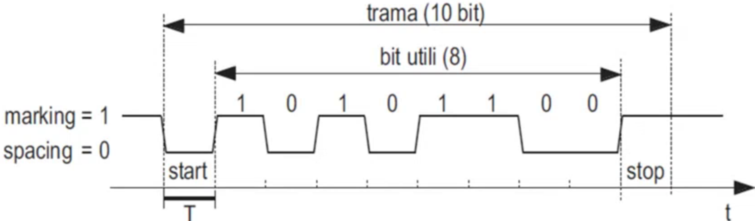
- Distinguiamo due tipi di comunicazione:
    - ***full-duplex***: il mezzo trasmissivo sostiene trasmissione in entrambe le direzioni contemporaneamente
    - ***half-duplex***: il mezzo trasmissivo indica la trasmissione da un solo lato.
- Nello spiegare la comunicazione seriale ci limiteremo, per questioni di semplificazione, a osservare la comunicazione *half-duplex*.



- Il mezzo trasmissivo consiste in un insieme di due linee:
    1. Una linea di riferimento, detta linea di massa
    2. Una linea che porta una tensione riferita a massa. Su questa linea sono leciti due valori:
        - *marking*, cioè 1 logico;
        - *spacing*, cioè 0 logico.
- La trasmissione di un bit consiste nel tenere la seconda linea in uno stato di *marking* o *spacing* per un determinato tempo T, detto **tempo di bit**.
- Un insieme di bit scambiati costituisce una trama, detta in inglese frame.

- **Problema fondamentale**: abbiamo detto che la comunicazione si ha mediante due fili, allora come possiamo sincronizzare trasmettitore e ricevitore? Chiaramente non potremo condividere un clock (significa avere un filo in più), né aggiungere delle linee dedicate all'*handshake* (due fili in più).
- **Soluzione al problema fondamentale**:
    - Le due parti devono essere concordi sul tempo di bit e sul numero di bit che caratterizza ogni trama (tipicamente si ha da 5 a 8 bit per trama);
    - Le trame devono essere rese riconoscibili, cioè abbiamo bisogno di elementi che permettono di capire quando una trama inizia e quando finisce. Abbiamo stabilito che:
        - La linea sta, normalmente, in uno stato di marking;
        - Per iniziare una nuova trama poniamo la linea in uno stato di spacing. Questo significa che ogni trama presenterà un bit iniziale, non informativo del contenuto della trama, detto bit di start.

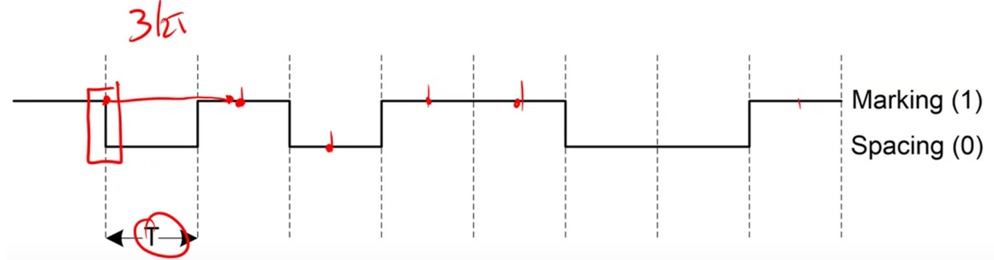
- Per concludere la trama poniamo la linea in uno stato di marking. Abbiamo quindi un bit finale, non informativo del contenuto della trama, detto bit di stop.



Esempio di trama. Il trasmettitore fa il primo passo ponendo il bit di start. Sia il trasmettitore che il ricevitore conoscono il tempo di bit.

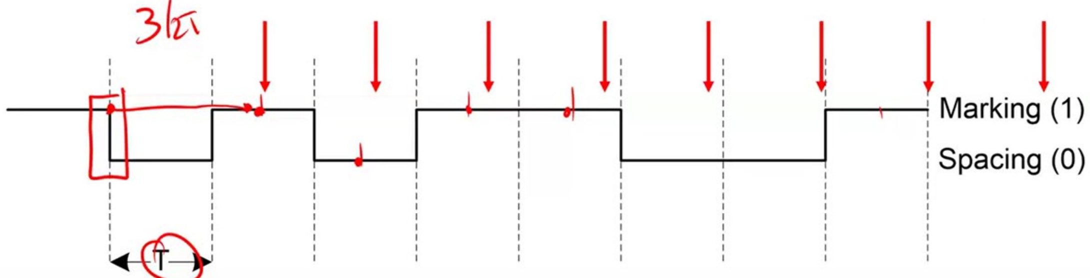
- Il bit di start ed il bit di stop, ribadiamo, sono bit di controllo:
  - Una trama di n bit utili è lunga almeno $n + 2$ bit (include il bit di start e il bit di stop);
  - La velocità di trasmissione netta è pari a $\frac{n}{n+2}$ della velocità della linea. La conseguenza per un ingegnere, che punta all'efficienza, è di fare trame con un numero di bit elevato.
- **Ma allora perché abbiamo detto che le trame sono costituite, in media, da 5/8 bit?** Il clock di trasmettitore e ricevitore non sono identici: possono essere simili, ma non uguali. Se le trame sono lunghe gli errori si accumulano e i bit della trama si disallineano.

- **Cosa avviene nella teoria?**



  - Devo individuare il fronte di discesa nel modo più preciso possibile.
  - Attendo 3/2 del tempo di bit.
  - Effettuo il primo campionamento e attendo, da adesso, un tempo di bit ogni volta.

- **Cosa avviene in realtà?**



  - Supponiamo che il clock del ricevitore sia un po' più lento: a causa di fenomeni fisici e della discrepanza dei clock si accumulano ritardi e nel giro di poco esco dal bit, quindi campiono in modo sbagliato.

- **Quale compromesso posso adottare?** Se il clock del ricevitore misura un tempo $T + \Delta T$ (sommo tempo di bit e ritardo), per poter decodificare n bit è necessario che
$$n \cdot \Delta T \leq \frac{T}{2}$$
cioè che la somma dei ritardi non superi la metà del tempo di bit. Se ciò avviene si esce dal bit e si campiona in modo sbagliato. Segue che

$$\frac{\Delta T}{T} \leq \frac{1}{2 \cdot n}$$

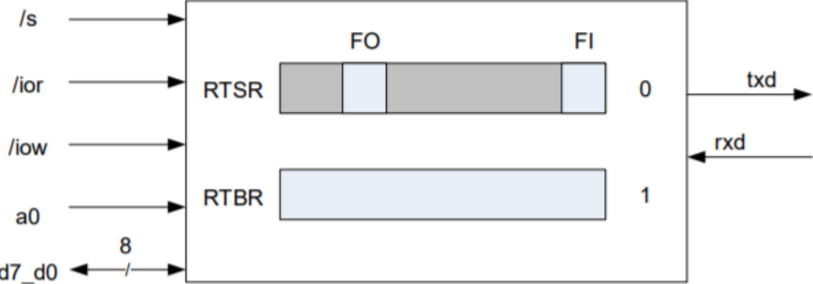
cioè l'errore relativo tollerabile è inversamente proporzionale al numero di bit che devono essere trasmessi tra due segnali di sincronizzazione.

- **Standard EIA-RS2E2C**: standard sviluppato negli anni 60 per gestire la comunicazione seriale. Si occupa di varie questioni: temporizzazione, funzione dei segnali, connettori, formato delle trame, protocollo di comunicazione, **voltaggi elettrici**…
- Quest'ultima cosa è quella che ci interessa maggiormente: si osservi che le tensioni indicate dallo standard sono diverse da quelle standard utilizzate nelle reti logiche.
  - 0 logico: tensione positiva [+3; +25]V
  - 1 logico: tensione negativa [-25; -3]V

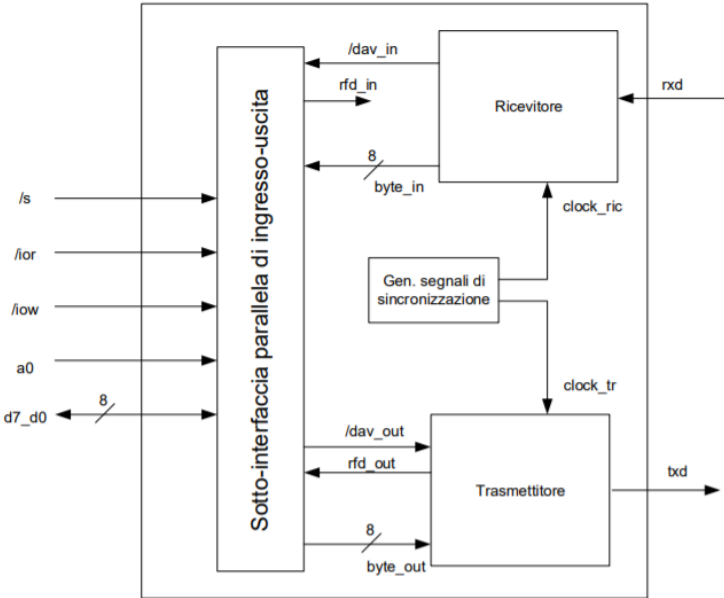
  Si è deciso di adottare queste tensioni per ragione di simmetria: si vuole fare in modo che il valor medio della tensione non sia significativamente diverso da zero (in caso contrario si accumula corrente).

Visione funzionale e struttura interna dell'interfaccia

- La struttura è molto simile all'interfaccia parallela con handshake ingresso/uscita:



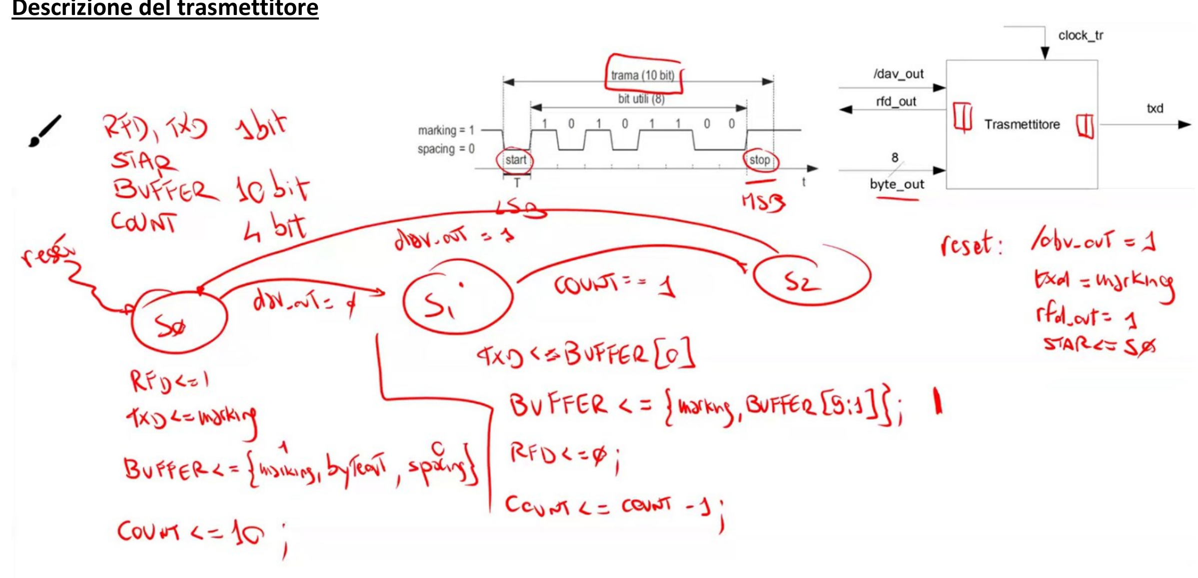
  Abbiamo un registro di stato RTSR, in cui il bit 5 ed il bit 0 sono rispettivamente il flag di uscita vuota FO e di ingresso pieno FI, e il registro RTBR ad 8 bit che serve per contenere i dati da trasmettere a quelli ricevuti.-
- Spogliamo ulteriormente l'interfaccia



  Individuiamo una RSS Ricevitore e una RSS Trasmettitore. Si osservi che:
  - Il singolo ricevitore o il singolo trasmettitore non costituiscono da soli l'interfaccia seriale. È necessaria una sotto-interfaccia parallela di ingresso-uscita.
  - Il ricevitore è un produttore (dal punto di vista della sotto-interfaccia): riceve dal dispositivo esterno, ma produce un byte per la sotto-interfaccia.
  - Il trasmettitore è un consumatore (dal punto di vista della sotto-interfaccia): riceve un byte dalla sotto-interfaccia, ma produce una trama per il dispositivo esterno.

- Inoltre:
  - Le due RSS presentano clock diverso (necessario, come capiremo poco più avanti).
  - La variabile rfd_in che non viene letta dal ricevitore. Questo perché il ricevitore non ha modi per comunicare col dispositivo esterno e indicare che il processore ha utilizzato i dati ed è pronto per riceverne altri.

Descrizione del trasmettitore



- L'interfaccia:
  - Accetta un nuovo byte dalla sotto-interfaccia parallela di uscita, con la quale ha un *handshake*;
  - Trasmette tutti i bit di quel byte sul mezzo trasmissivo tramite il filo *txd*.

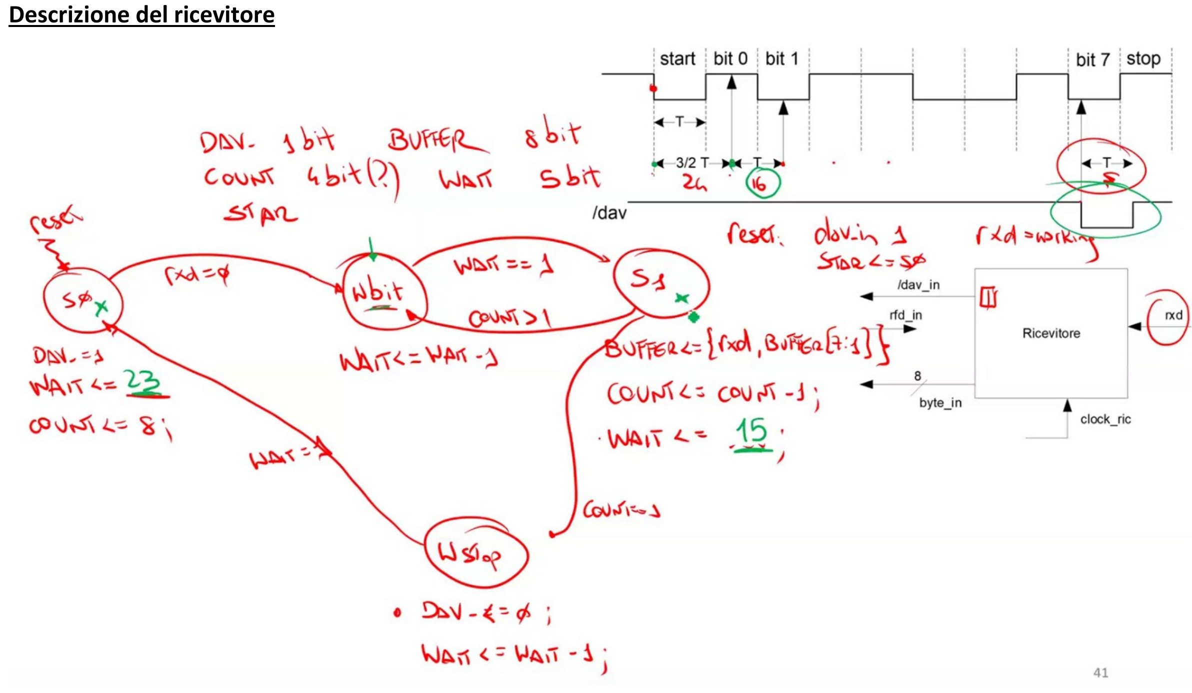
- **Di quali registri abbiamo bisogno?**
  - Registro di stato STAR
  - Registri a supporto delle uscite: RFD, TXD (ciascuno di 1bit)
  - Registro BUFFER dove memorizzare l'intera sequenza di bit (10 bit, incluso il bit di start e il bit di fine)
  - Registro COUNT, quattro bit (10) necessari per contare il numero di bit da trasmettere.
- **Ipotesi al reset**:
  - /dav_out e rfd_out presentano i valori tipici dell'handshake al momento del reset (entrambi 1)
  - txd è uguale ad uno, quindi è in uno stato di marking
  - STAR è uguale ad S0, come al solito si pone come valore iniziale quello del primo valore interno.
- **Ipotesi sul clock**: un clock equivale al tempo di bit T.

```
module Trasmettitore (dav_out_, rfd_out, byte_out, txd, clock, reset_);
  input clock, reset_;
  input dav_out_;
  input [7:0] byte_out;
  output rfd_out, txd;

  reg [3:0] COUNT;
  reg [9:0] BUFFER;
  reg RFD, TXD; assign rfd_out=RFD; assign txd=TXD;

  reg [1:0] STAR; parameter S0=0, S1=1, S2=2;
  parameter mark=1'B1, start_bit=1'B0, stop_bit=1'B1;

  always @(reset_==0) #1 begin RFD<=1; TXD<=mark; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
    casex (STAR)
      S0: begin RFD<=1; COUNT<=10; TXD<=mark;
                BUFFER<={stop_bit,byte_out,start_bit};
```

```
                 STAR<=(dav_out==1)?S0:S1; end
       S1: begin RFD<=0; TXD<=BUFFER[0]; BUFFER<={mark, BUFFER[9:1]};
                 COUNT<=COUNT-1; STAR<=(COUNT==1)?S2:S1; end
       S2: begin STAR<=(dav_out==0)?S2:S0; end
     endcase
endmodule
```

- **Spiegazione degli stati**:
  - **S0**: pongo RFD uguale ad 1, così come TXD in marking. Per ragioni di semplificazione e riduzione di stati (abbiamo già visto questo in passato) aggiorniamo il valore dei registri BUFFER e COUNT direttamente in S0. Attenzione al valore di BUFFER (includo nei bit anche il bit di star e quello di fine). Passo allo stato successivo quando /dav_out viene abbassato (cioè quando si hanno nuovi dati da trasmettere)
  - **S1**: gestisco la trasmissione dei bit mediante una serie di shift destri. Purtroppo la lettura in array del tipo BUFFER[COUNT] non siamo in grado di sintetizzarla. Esco quando avrò COUNT uguale ad 1, cioè quando avrò restituito tutti i bit attraverso il filo txd (a quel punto il buffer avrà tutti bit uguali ad 1).
  - **S2**: non posso tornare subito in S0, visto che devo finire di gestire la temporizzazione dell'handshake. Attendo che /dav_out ritorni ad 1 e solo a quel punto abbiamo finito.

Descrizione del ricevitore



- Il ricevitore:
  - Non è collegato al filo rfd_in. Come già anticipato il ricevitore non è in grado di controllare il flusso dei dati in ingresso: segue che non ha nessun senso gestire un handshake completo. Se la sottointerfaccia parallela non è in grado di accettare un nuovo dato è problema suo: il dato verrà sovrascritto da una nuova trama di bit.
  - Il ricevitore capta l'inizio di una trama col fronte di discesa da marking a spacing. Campiona il primo bit dopo 3/2 del tempo di bit da quando vede l'inizio della trama. Successivamente campiona ciascun successivo bit dopo un tempo di bit.
- **Ipotesi sul clock**:
  - Il clock del ricevitore deve essere diverso da quello del trasmettitore: questo perché dobbiamo riuscire a campionare i bit (per quanto possibile) a metà del tempo di bit. Non è possibile fare ciò con lo stesso clock del trasmettitore.
  - **Come imposto il clock?** Il periodo di clock è la minima finestra temporale su cui guardo gli eventi, il ricevitore deve individuare un certo evento nella maniera più veloce possibile: segue che più il clock è veloce, più saremo precisi. Supporremo che il clock abbia frequenza **16x**, cioè che sia sedici volte più veloce del clock del trasmettitore.

- **Di quali registri abbiamo bisogno?**
  - Registro DAV_ a 1 bit, che supporta l'uscita /dav_in
  - Il registro BUFFER dove la parte di trama ricevuta fino a questo momento. Mi bastano solo 8 bit.
  - Il registro COUNT, con cui tengo conto del numero di bit letti.
  - Il registro WAIT, per contare i clock. Utilizzo questo registro per attendere:
    - 16 cicli di clock tra un bit utile e il successivo, e alla fine (quando finisco nel bit di stop). In questo ultimo caso potrei attendere solo 8 clock, ma considerando le discrepanze fisiche nei due clock non mi conviene.
    - 24 cicli di clock dopo il fronte di discesa del bit di start

- **Ipotesi al reset**:
  - /dav_in è a 1 (come nell'handshake)
  - byte_in non è significativo poiché è protetto dall'handshake
  - rxd è in marking
- **Spiegazione degli stati**:
  - **S0**: in questo stato ho /dav a 1. Passo allo stato successivo quando rxd va in spacing, e rimango in quello stato per un numero di clock pari a 23 (cioè 3/2 del tempo di bit). Pongo WAIT uguale a 23 e non a 24 poiché perdo già un clock in S0 prima di passare allo stato Wbit.
  - **Wbit**: se mi trovo in questo stato significa che devo attendere prima di fare il prossimo campionamento. Abbiamo definito, in S0 e in S1, un valore di WAIT. WAIT viene decrementato e mi sposto (o ritorno) in S1 quando ho WAIT uguale ad 1.
  - **S1**: imposto il valore di BUFFER. Devo fare in modo che i primi valori inseriti si trovino sui posti meno significativi: faccio ciò inserendo ogni nuovo bit (preso dal filo rxd) come MSD (shifto sbarazzandomi della LSD). Decremento COUNT, ed ogni volta ritorno in Wbit (tranne quando avrò COUNT == 1). Memorizzo in WAIT 15, e non 16, per la stessa motivazione vista in S0. Quando COUNT sarà uguale ad 1 mi sposto in Wstop.
  - **Wstop**: porto /dav a 0, abbiamo finito. Prima di ritornare in S0 devo attendere 16 clock (il solito tempo di bit). A tale scopo recupero il valore del registro WAIT, impostato in S1. Decremento WAIT, e ritorno in S0 quando avrò WAIT == 1.

```
module Ricevitore (dav_in_, clock, rxd, reset_, byte_in);
  input clock, rxd, reset_;
  output dav_in_;
  output [7:0] byte_in;

  reg DAV_; assign dav_in_=DAV_;
  reg [3:0] COUNT;
  reg [4:0] WAIT;
  reg [7:0] BUFFER; assign byte_in=BUFFER;
  reg [1:0] STAR; parameter S0=0, S1=1, Wbit=2, Wstop=3;
  parameter start_bit=1'B0;

  always @(reset_==0) #1 begin DAV_<=1; STAR<=S0; end
  always @(posedge clock) if (reset_==1) #3
  casex (STAR)
   S0: begin DAV_<=1; COUNT<=8; WAIT<=23;
             STAR<=(rxd==start_bit)?Wbit:S0; end
   Wbit: begin WAIT<=WAIT-1; STAR<=(WAIT==1)?S1:Wbit; end
   S1: begin COUNT<=COUNT-1; WAIT<=15; BUFFER<={rxd,BUFFER[7:1]};
            STAR<=(COUNT==1)?WStop:Wbit; end
   WStop: begin DAV_<=0; WAIT<=WAIT-1; STAR<=(WAIT==1)?S0:WStop; end
  endcase
endmodule
```

Capitolo 45

Martedì 09/12/2020

45.1 Conversione analogico/digitale e digitale/analogica

Finora ci siamo limitati a osservare interfacce che consentono a due calcolatori di dialogare tra loro. Non dobbiamo farci ingannare dall'apparenza che i bit ci siano per magia: nel mondo esterno al calcolatore l'informazione è associata a grandezze analogiche, cioè grandezze che cambiano continuamente nel tempo (in contrapposizione alle stringhe di bit presenti nei calcolatori, che variano in modo discreto). Abbiamo la necessità di svolgere le seguenti conversioni:

- **Conversione A/D**: conversione di grandezze analogiche in stringhe di bit (comunicazione del mondo esterno col calcolatore)
- **Conversione D/A**: conversione di stringhe di bit in grandezze analogiche (comunicazione del calcolatore col mondo esterno).

La grandezza analogica da noi considerata è la tensione.

Obiettivo

Convertire una tensione v in un numero x (naturale o intero), in base 2 e rappresentato su N bit, e viceversa

Dove N è solitamente 8 o 16. La tensione sta su una scala di FSR volts (*Full-scale Range*): tipicamente abbiamo $\text{FSR} \in [5; 30]$.

Interpretazione del numero o della tensione Distinguiamo due tipi di conversioni:

- **Conversione unipolare**: $v \in [0; \text{FSR}]$ (tensioni positive), $x \in [0; 2^N - 1]$ (numeri naturali)
- **Conversione bipolare**: $v \in \left[-\frac{\text{FSR}}{2}; +\frac{\text{FSR}}{2}\right]$ (tensioni negative e positive), $x \in [-2^{N-1}; 2^{N-1} - 1]$ (numeri naturali)

Mondo ideale Definiamo il rapporto tra le due scale (tensione e numero)

$$K = \frac{\text{FSR}}{2^N}$$

in una conversione ideale dovrei ottenere

$$v = K \cdot x$$

Mondo non ideale Nella realtà questa cosa non è possibile poichè abbiamo un errore di conversione

$$|v - k \cdot x| \leq \text{err}$$

Questo errore è dovuto ai seguenti fattori:

1. **Imprecisioni circuitali**, cioè componenti di un circuito che non si comportano in maniera ideale. Questo errore, difficilmente eliminabile, è presente sia nella conversione D/A che in quella A/D. Si parla di **errore di non linearità**.

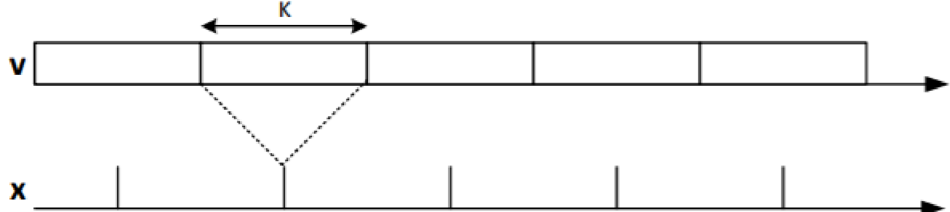
2. **Arrotondamento** (o **Errore di quantizzazione**). Si tenga conto nella formula $v = K \cdot x$ che $v \in \mathbb{R}$ e $x \in \mathbb{Z}$: questo significa che nella conversione A/D, cioè nel passaggio da tensione a numero, si avrà perdita di informazione (conversione da numero reale a numero intero).

Riflettiamo sull'errore di linearità L'errore di linearità deve essere più piccolo di $\frac{K}{2}$. La presenza dell'errore ci porta a dire che la tensione appartiene a questo intervallo

$$v \in [K \cdot x - \text{err}; K \cdot x + \text{err}]$$

avere un errore chec non rispetta la condizione detta significa avere intervalli parzialmente sovrapposti, quindi convertire numeri più grandi in tensioni più piccole e viceversa.

Riflettiamo sull'errore di quantizzazione Supponiamo di dividere la scala FSR in 2^N intervalli uguali a K, e di convertire tutto l'intervallo nello stesso numero.



La conversione da v a x sarà esatta per la tensione al centro dell'intervallo, ed errata di $\pm\frac{K}{2}$ per le tensioni agli estremi.

Conclusioni

- Nella conversione D/A dobbiamo avere err $< \frac{K}{2}$ (errore di non linearità)
- Nella conversione A/D dobbiamo avere err $< \frac{K}{2} + \frac{K}{2} = K$ (errore di non linearità ed errore di quantizzazione)

Esempi di conversioni Presenti degli esempi di conversione a pagina 53 della dispensa sulla struttura del calcolatore.

Tempi di risposta delle conversioni

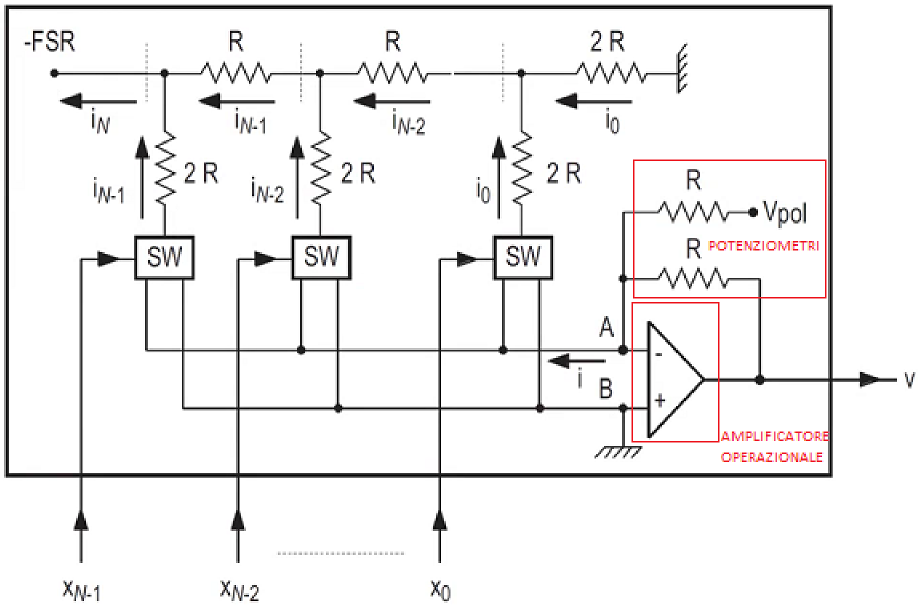
- I convertitori D/A sono circuiti estremamente semplici. Sono velocissimi (pochi ns).
- I convertitori A/D hanno tempi di risposta variabili: sono circuiti sequenziali dove posso incontrare architetture diverse. I convertitori da noi considerati hanno tempi di risposta di qualche centinaio di ns.

Osservazione sulla conversione bipolare I convertitori bipolari rappresentano i numeri interi con rappresentazione in traslazione. Il numero intero x è rappresentato dal naturale

$$X = x + 2^{N-1}$$

la tensione negativa di fondo scala, che corrisponde al numero intero negativo più piccolo, verrà convertita nel naturale 0. Ricordarsi che la conversione da traslazione a complemento a 2, e viceversa, si ottiene complementando il MSB.

45.1.1 Convertitore D/A



La tensione v in uscita dovrà essere direttamente proporzionale al numero posto in ingresso (già da questo si capisce che la rappresentazione avviene per traslazione). Facciamo l'assunzione che tutte le linee verticali siano connesse a massa: provengono tutte da switch, che presentano in ingresso la linea A e la linea B. La linea B è esplicitamente connessa a massa, la linea A lo verificheremo prossimamente.

- Ogni ramo verticale presenta la stessa corrente che scorre nel ramo orizzontale alla sua destra.
- La resistenza vista a destra di ogni tratteggio è pari ad R.
- Ottengo due resistenze in serie che sommate mi danno $R + R = 2R$.

- La linea verticale immediatamente vicina al ramo orizzontale considerato presenta sempre come resistenza $2R$, quindi otteniamo in ogni filo verticale lo stesso risultato di prima.
- Per quanto riguarda la corrente abbiamo, nel tratteggio vicino alla linea verticale più a destra
$$i_p = i_0 + i_0 = 2 \cdot i_k$$
- Sommando le varie correnti otteniamo, alla fine,
$$\boxed{i_N = 2^N \cdot i_0}$$

Osservando che $i_N = \frac{FSR}{R}$ troviamo

$$2^N \cdot i_0 = \frac{FSR}{R}$$

quindi

$$i_0 = \frac{\text{FSR}}{2^N} \cdot \frac{1}{R} = \frac{K}{R}$$

dobbiamo ancora verificare che i rami verticali siano collegati a massa.

Vediamo gli switch Gli switch sono guidati dai bit della rappresentazione del numero da convertire. Abbiamo tanti switch quante le cifre del numero.

- Se $x_j = 0$ l'interruttore è connessa alla linea B, quindi a massa.
- Se $x_j = 1$ 'interruttore è connesso alla linea A, che non sappiamo quanto vale.

Amplificatore operazionale Componente connessa all'alimentazione, la corrente non proviene dagli ingressi.

$$V^{\text{out}} = \alpha \cdot (V^+ - V^-)$$

con $\alpha >> 1$, la tensione dipende dalle differenze di tensione agli ingressi dell'amplificatore. tenendo conto che il filo B è connesso a massa possiamo dire

$$V^{\text{out}} = -\alpha \cdot V^-$$

Consideriamo anche l'altro ramo, quello con resistenza R e corrente i_a

$$V^{\text{out}} - R \cdot i_a = V^-$$

unisco il tutto e ottengo

$$V^{\text{out}} = -\alpha \cdot V^{\text{out}} + \alpha \cdot R \cdot i_a$$

quindi

$$V^{\text{out}} = \frac{\alpha}{1+\alpha} \cdot R \cdot i_a \approx R \cdot i_a$$

questo mi permette di dire $V^- \approx 0$. ■

Quanto vale la corrente che esce da A e va verso sinistra? Tenendo conto di quanto detto sullo switch affermiamo che

$$\begin{aligned} i &= x_0 \cdot i_0 + x_1 \cdot i_1 + \cdots + x_{N-1} \cdot i_{N-1} \\ &= i_0 \cdot x_0 + (2 \cdot i_0) \cdot x_1 + \cdots + (2^{N-1} \cdot i_0) \cdot x_{N-1} = i_0 \cdot \sum_{i=0}^{N-1} 2^i \cdot x_i \end{aligned}$$

ci siamo ricordati che $i_1 = 2 \cdot i_0$, e così via fino ad arrivare a $i_{N-1} = 2^{N-1} \cdot i_0$. La sommatoria ottenuta è la rappresentazione posizionale in base 2 di un numero naturale X. Ricordandomi che $i_0 = \frac{K}{R}$ ottengo

$$i = \frac{K}{R} \cdot \sum_{i=0}^{N-1} 2^i \cdot x_i = \frac{K}{R} \cdot X$$

segue che gli switch alterano la corrente che scorre da A verso sinistra.

Come faccio uscire la tensione giusta dal convertitore? Scriviamo le equazioni di bilancio della corrente al nodo A

$$\frac{K}{R} \cdot X = \frac{V_{pol} + V}{R}$$

Mi sbarazzo di R trovando

$$K \cdot X = V_{pol} + V$$

quindi (raccolgo rispetto a K)

$$V = K \cdot \left(X - V_{pol} \cdot \frac{2^N}{FSR} \right)$$

otteniamo che V è proporzionale rispetto ad X. Segue che con

- $V_{pol} = 0$ avrò una conversione unipolare $V = K \cdot X$, mentre con
- $V_{pol} = \frac{FSR}{2}$ avrò una conversione bipolare

  $$V = K \cdot (X - 2^{N-1})$$

  cioè il numero rappresentato in traslazione in base 2.

Il circuito è in sostanza combinatorio.

Problema 1: Situazione precedente nella realtà Quanto spiegato poco fa è valido solo in un modello ideale. Nella realtà dobbiamo tener conto di errori. Riprendiamo l'equazione di bilancio e teniamo conto della cosa

$$\frac{K}{R} \cdot X = \frac{V_{pol} \cdot \gamma_1 + V \cdot \gamma_2}{R}$$

otteniamo

$$V = \frac{K}{\gamma_2} \cdot \left(X - V_{pol} \cdot \gamma_1 \cdot \frac{2^N}{FSR} \right)$$

osserviamo il rapporto linea tra tensione e ingresso X (sappiamo che esiste una proporzionalità diretta tra tensione in uscita e stringa di bit X posta in ingresso)

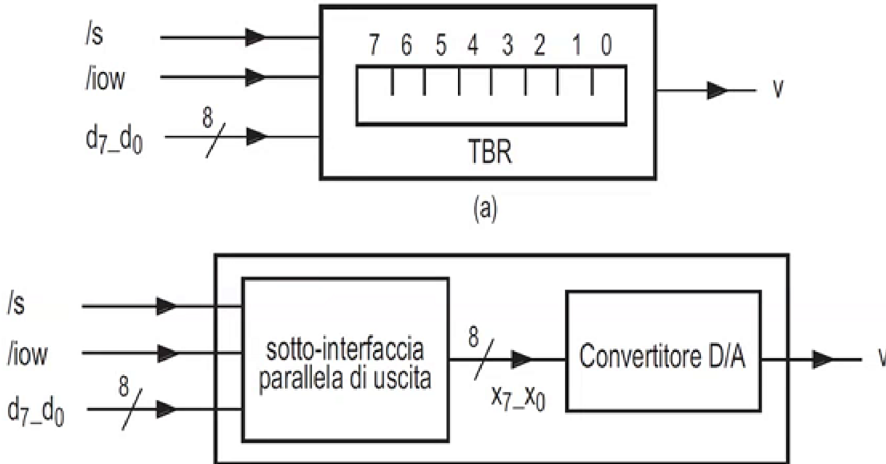
- Se altero γ_2 altero la pendenza della retta.
- Se altero γ_1 traslo la retta.

A tutto questo segue la necessità di dover tarare il convertitore prima di svolgere questa cosa. Le due resistenze vicine all'amplificatore operazionale sono dette *potenziometri*, cioè resistenze variabili che possono essere tarate.

Problema 2: Variazioni a frequenza elevata Supponiamo di voler passare da $X = 01111111$ alla sua complementata $X = 10000000$. Sappiamo fin dall'inizio che la rete non percepirà mai, in modo istantaneo e parallelo, la variazione di tutti i bit. Seguono stati intermedi. Questo comporta variazioni ad alta frequenza, un qualcosa che deve essere evitato. La soluzione è porre un filtro passa-basso, che taglia le variazioni a frequenza troppo elevata.

45.1.1.1 Interfaccia per la conversione D/A

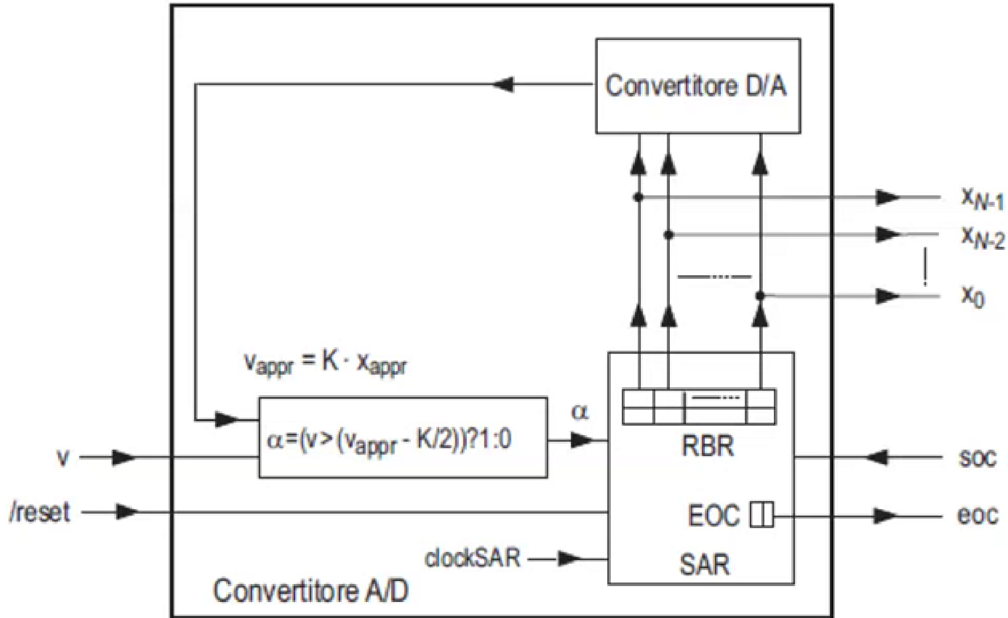
Attenzione Non confondere il convertitore con l'interfaccia!



Abbiamo un'interfaccia parallela senza handshake, con un unico registro TBR (posto in una sotto interfaccia parallela di uscita) e un convertitore D/A.

Velocità Il convertitore D/A è molto più veloce del processore, quindi non servono ulteriori cose nell'interfaccia.

45.1.2 Convertitore A/D



Il convertitore che andiamo a descrivere è detto **convertitore A/D ad approssimazioni successive** ad 8 bit. Il cuore del convertitore è la RSS detta SAR (*Successive Approximation Register*).

- All'interno del convertitore A/D è presente un convertitore D/A (chiaramente dovrò utilizzare un convertitore bipolare o unipolare in base alle circostanze) che utilizzerò per generare una tensione da confrontare, mediante comparatore analogico, con la tensione in ingresso.
- Utilizzeremo un'handshake soc/eoc, considerando la mole di lavoro.
- Il convertitore esegue una **ricerca logaritmica** (detta bisezione, o ricerca binaria).
  - Ho una tensione v analogica in ingresso, mantenuta costante nel corso dell'esecuzione dell'algoritmo mediante latch analogico (non avrebbe senso farla ballare durante i confronti, osservazione).
  - Quando il convertiitore percepisce soc = 1 inizia a "sparare" numeri nel convertitore D/A. Si pone come primo numero quello al centro dell'intervallo di rappresentabilità ($100\dots00$). Questo numero è valido sia con conversione unipolare che con conversione bipolare:
  - Ogni tensione ottenuta viene confrontata con la tensione in ingresso (resa costante, ricordiamo) con un comparatore di tensione.
  - Se la tensione generata è più grande della tensione in ingresso allora avremo $\alpha = 1$, quindi la cifra considerata in una certa iterazione uguale ad 1.
  - Continuo a ripetere le operazioni precedenti, quindi a svolgere confronti tra tensioni.
  - Il numero di passi da svolgere è pari al numero di bit: ogni confronto, in base 2, mi permette di individuare un bit. Li determiniamo dalla cifra più significativa a quella meno significativa.

Esempio

Vediamo l'esempio qua sopra. Ogni volta:

- Eseguo il confronto fra tensioni
- Determino se settare o resettare la cifra. La cifra meno significativa immediatamente successiva viene impostata uguale ad 1.

Descrizione Verilog Descriviamo la RSS che permette di gestire l'handshake e il contenuto del buffer che ogni volta sarà "sparato" verso il convertitore D/A.

```
module SAR(eoc,x7_x0,soc,alpha,clockSAR,reset_);
input clockSAR,reset_;
input soc,alpha;
output eoc;
output[7:0] x7_x0;
reg EOC; assign eoc=EOC;
reg[7:0] RBR; assign x7_x0=RBR;
reg[3:0] STAR;
parameter S0=0,S1=1,S2=2,S3=3,S4=4,S5=5,S6=6,S7=7,S8=8,S9=9,S10=10;

always @(reset_==0) #1 begin EOC<=1; STAR<=S0; end
always @(posedge clockSAR) if (reset_==1) #3
casex(STAR)
S0: begin EOC<=1; STAR<=(soc==0)?S0:S1; end
S1: begin RBR<='B10000000; EOC<=0; STAR<=S2; end
S2: begin RBR<={ alpha,'B1000000}; STAR<=S3; end
S3: begin RBR<={RBR[7], alpha,'B100000}; STAR<=S4; end
S4: begin RBR<={RBR[7:6],alpha,'B10000}; STAR<=S5; end
S5: begin RBR<={RBR[7:5],alpha,'B1000}; STAR<=S6; end
S6: begin RBR<={RBR[7:4],alpha,'B100}; STAR<=S7; end
S7: begin RBR<={RBR[7:3],alpha,'B10}; STAR<=S8; end
S8: begin RBR<={RBR[7:2],alpha,'B1}; STAR<=S9; end
S9: begin RBR<={RBR[7:1],alpha }; STAR<=S10; end
S10: begin EOC<=(soc==1)?0:1; STAR<=(soc==1)?S10:S0; end
endcase
endmodule
```

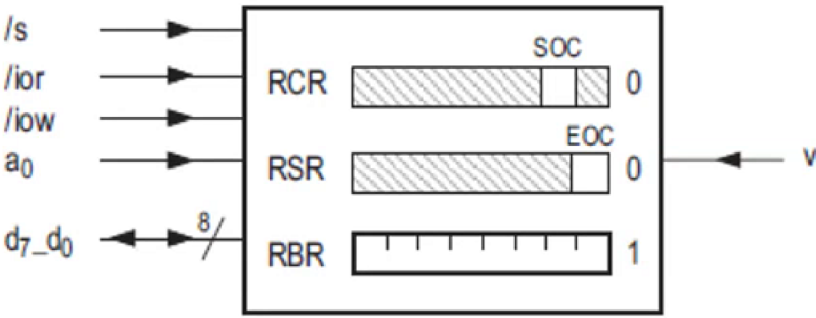
Si consideri anche la seguente versione, semplificata nel codice e nel numero di stati interni grazie all'introduzione di una function:

```
module SAR(eoc,x7_x0,soc,alpha,clockSAR,reset_);
input clockSAR,reset_;
input soc,alpha;
output eoc;
output[7:0] x7_x0;
reg EOC; assign eoc=EOC;
reg[7:0] RBR; assign x7_x0=RBR;
reg[3:0] STAR;
reg[2:0] COUNT;
parameter S0=0,S1=1,S2=2,S3=3;

always @(reset_==0) #1 begin EOC<=1; COUNT<=7; STAR<=S0; end
always @(posedge clockSAR) if (reset_==1) #3
casex(STAR)
S0: begin EOC<=1; STAR<=(soc==0)?S0:S1; end
S1: begin RBR<='B10000000; EOC<=0; STAR<=S2; end
S2: begin RBR<=nuovobyte(RBR, alpha, COUNT); COUNT<=COUNT-1;
STAR<=(COUNT==0)?S3:S2; end
S3: begin EOC<=(soc==1)?0:1; STAR<=(soc==1)?S3:S0; end
endcase

function [7:0] nuovobyte;
input [7:0] vecchiobyte;
input alpha;
input [2:0] posizione;
casex (posizione)
7: nuovobyte={ alpha,'B1000000};
6: nuovobyte={vecchiobyte[7], alpha,'B100000};
5: nuovobyte={vecchiobyte[7:6],alpha,'B10000};
4: nuovobyte={vecchiobyte[7:5],alpha,'B1000};
3: nuovobyte={vecchiobyte[7:4],alpha,'B100};
2: nuovobyte={vecchiobyte[7:3],alpha,'B10};
1: nuovobyte={vecchiobyte[7:2],alpha,'B1};
0: nuovobyte={vecchiobyte[7:1],alpha };
endcase
endfunction
endmodule
```

- Ogni volta, sfruttando l'ingresso alpha generato dal comparatore, si determina un bit della sequenza di cifre.
- La sequenza memorizzata inizialmente nel RBR è provvisoria, quando alzeremo eoc il processore chiederà una lettura del buffer trovando la sequenza definitiva.

45.1.2.1 Interfaccia di conversione A/D

Ricordiamo di nuovo L'interfaccia non è il convertitore.

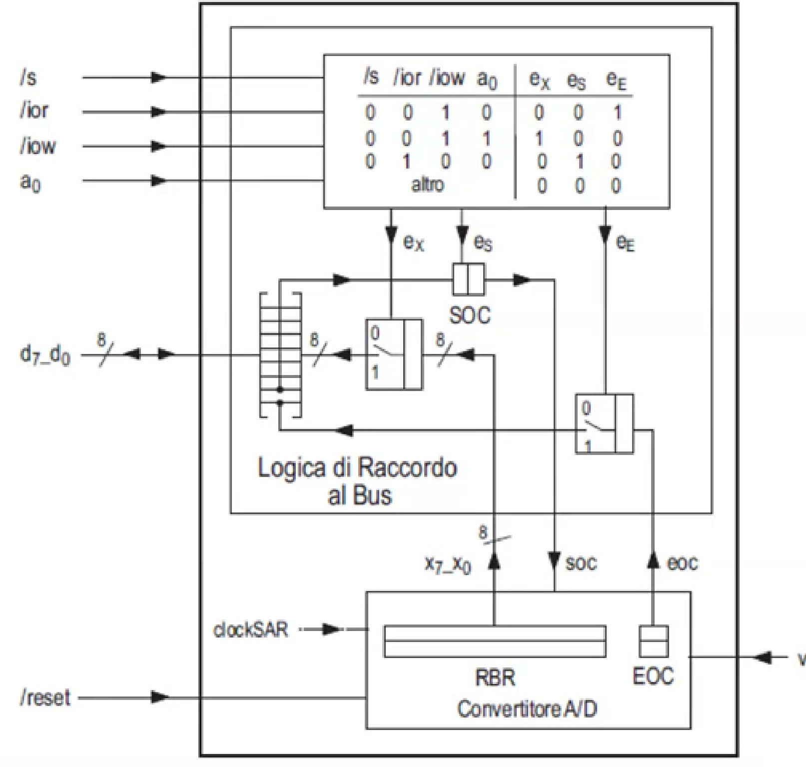


Abbiamo un'interfaccia parallela con handshake soc/eoc. Presenta i seguenti registri:

- Il registro di Buffer dove finisce il risultato (RBR, *Receive Buffer Registry*)
- Un registro di stato per leggere il valore di eoc (RSR, *Receive Status Registry*)
- Un registro di controllo per poter aggiornare il valore di soc (RCR, *Receive Control Registry*).

Gli ultimi due indirizzi sono mappati allo stesso indirizzo logico (si ottiene il registro RSCR, *Receive Status and Control Registry*). Non è un problema perchè i bit non si sovrappongono: inoltre la rete sa cosa restituire in base all'operazione indicata (lettura o scrittura).

Spogliamo l'interfaccia



- Le operazioni possibili sono:
  - lettura di eoc
  - letture di RBR
  - scrittura di soc
- Abbiamo una rete combinatoria che permette di gestire le porte tristate (e il registro SOC). Osserviamola:
  - Quando le variabili di pilotaggio non indicano una particolare operazione tutte le porte sono in alta impedenza.
  - Quando voglio leggere eoc avrò $e_E = 1$, quindi la porta relativa in conduzione. L'utente troverà eoc sul filo in posizione 0 (nei fili di dati).
  - Quando voglio leggere RBR avrò $e_x = 1$, enabler di tante porte tristate quante il numero di bit del registro (8). Tutte queste porte sono in conduzione, le rimanenti in alta impedenza. L'utente troverà in uscita, su tutti i fili di dati, il valore del registro RBR.
  - Quando voglio scrivere su soc avrò $e_S = 1$. Questa variabile è il clock per il registro SOC: se $e_s = 1$ il valore del registro impostato sarà quello posto in posizione 1 (nei fili di dati). Segue un nuovo ingresso nella SAR e quindi l'inizio di una nuova operazione di conversione, se necessario.