DOMANDE ORALE

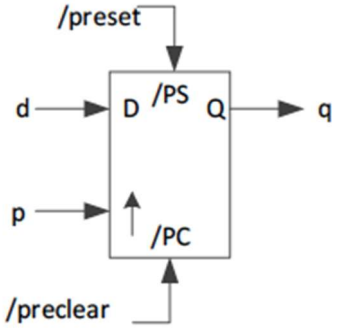
SOMMARIO

SESSIONE INVERNALE – A.A. 2020-2021

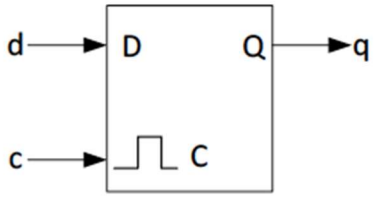
PRIMO APPELLO

STRUTTURA DI UN D FLIP-FLOP PARTENDO DA UN D-LATCH

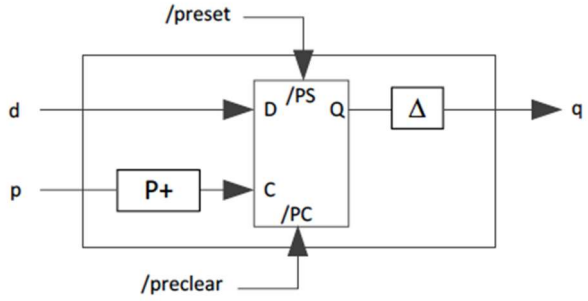
Innanzitutto, un D Flip-flop è una rete con due variabili di ingresso, *d* e *p*, che **quando *p* ha un fronte di salita, memorizza *d*, altrimenti attende un po' e adegua l'uscita.**



Per sintetizzarlo, si prende un D-Latch, fatto in questo modo:



E si premette alla variabile *c* un **formatore di impulsi**, in modo tale che, al fronte di salita di *p*, il D-Latch **vada brevemente in trasparenza** e memorizzi *d*. Inoltre, si **ritarda l'uscita** di un ritardo Δ **maggiore dell'intervallo del P+.**



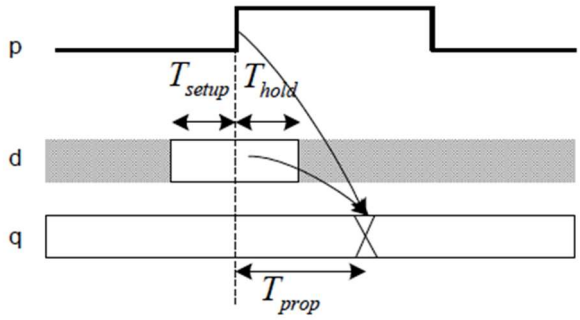
In questo modo, quando *q* cambia adattandosi a *d*, **la rete non è più in trasparenza, ma in conservazione.**

L'uscita *q* viene adeguata al valore campionato di *d* <u>dopo</u> che la rete ha smesso di essere sensibile al valore di *d*.

<u>TEMPORIZZAZIONI</u>

A cavallo del fronte di salita di *p*, la variabile *d* **deve rimanere costante**. I tempi per cui deve rimanere costante prima e dopo il fronte di salita si chiamano T_{setup} e T_{hold}. Questi due tempi, nonostante abbiano lo stesso nome, **non sono gli stessi del D-Latch**.

Il ritardo con cui si adegua l'uscita, misurato a partire dal fronte di salita di *p*, si chiama T_{prop} e vale la relazione $T_{prop}>T_{hold}$. Questa ultima disuguaglianza garantisce che la rete è **non trasparente**.

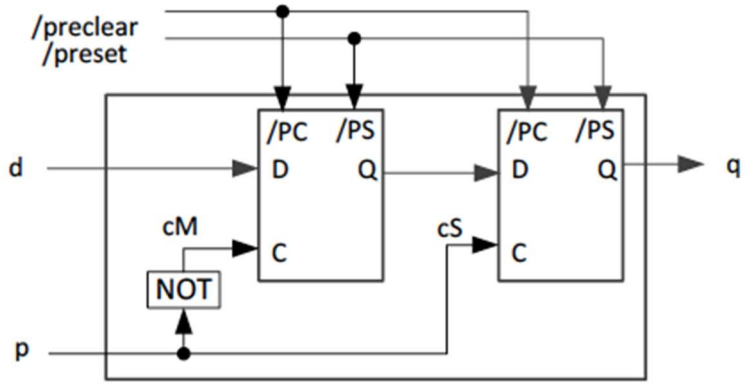


RETE NON TRASPARENTE

Una rete si dice non trasparente quando adegua l'uscita al valore in ingresso **dopo** che la rete ha smesso di essere sensibile al valore in ingresso.

MONTAGGIO MASTER-SLAVE

Il montaggio master-slave si realizza montando **due D-Latch in cascata.**

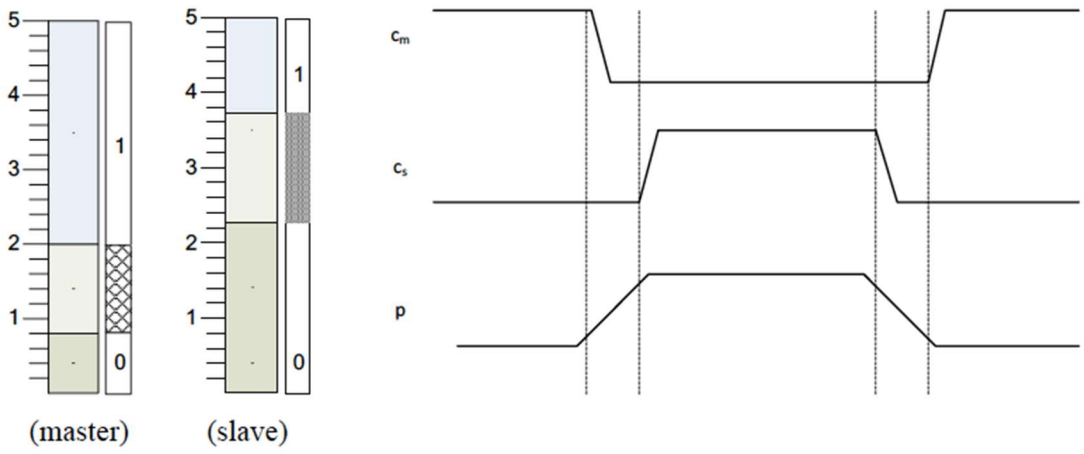


- *p=0*, **il master campiona e lo slave conserva** (quindi non ascolta il proprio ingresso *d*).
- *p=1*, **il master conserva e lo slave campiona** (quindi insegue l'uscita del master).

Sul fronte di salita di *p*, il master memorizza l'ultimo valore di *d* che ha letto, e lo slave presenta (con un certo ritardo T_{prop}) quell'ultimo valore come uscita della rete globale.

L'adeguamento dell'uscita avviene dopo il fronte di salita di *p*, ma più o meno contemporaneamente il master isola l'uscita dall'ingresso.

Ci possono essere problemi per quanto riguarda il funzionamento **transitorio**. In particolare, può succedere che il master e lo slave siano **contemporaneamente in trasparenza**, anche se per un transitorio. Per evitare questo, si agisce per via **elettronica**. Si fa in modo che l'ingresso *c* dello slave riconosca la tensione di ingresso come 1 soltanto quando questa è prossima al fondo scala, e riconosca come 0 un maggior range di tensioni.



FASE DI ESECUZIONE DELL'ISTRUZIONE Jcon

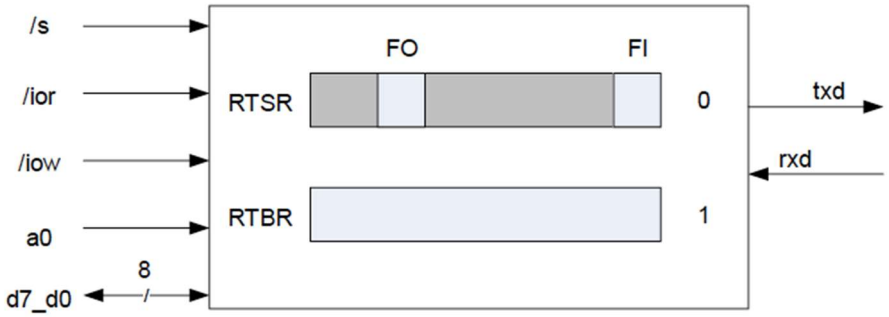
Per tutte le istruzioni di salto (condizionato o non), la decisione se saltare o meno è fornita dalla rete combinatoria `jmp_condition()`, che, sulla base dell'opcode (e quindi della condizione richiesta), stabilisce se la condizione sia vera o meno. Se si deve saltare, il nuovo valore di `IP` è il contenuto di `DEST_ADDR`, perché il formato F7 mette in `DEST_ADDR` l'indirizzo contenuto dell'istruzione stessa.

```
jmp: begin IP<=(jmp_condition(OPCODE, F)==1)?DEST_ADDR:IP;
     STAR<=fetch0; end
```

INTERFACCIA SERIALE

VISIONE FUNZIONALE

Dal punto di vista funzionale, un'interfaccia seriale di ingresso/uscita è simile ad un'interfaccia parallela di ingresso/uscita.



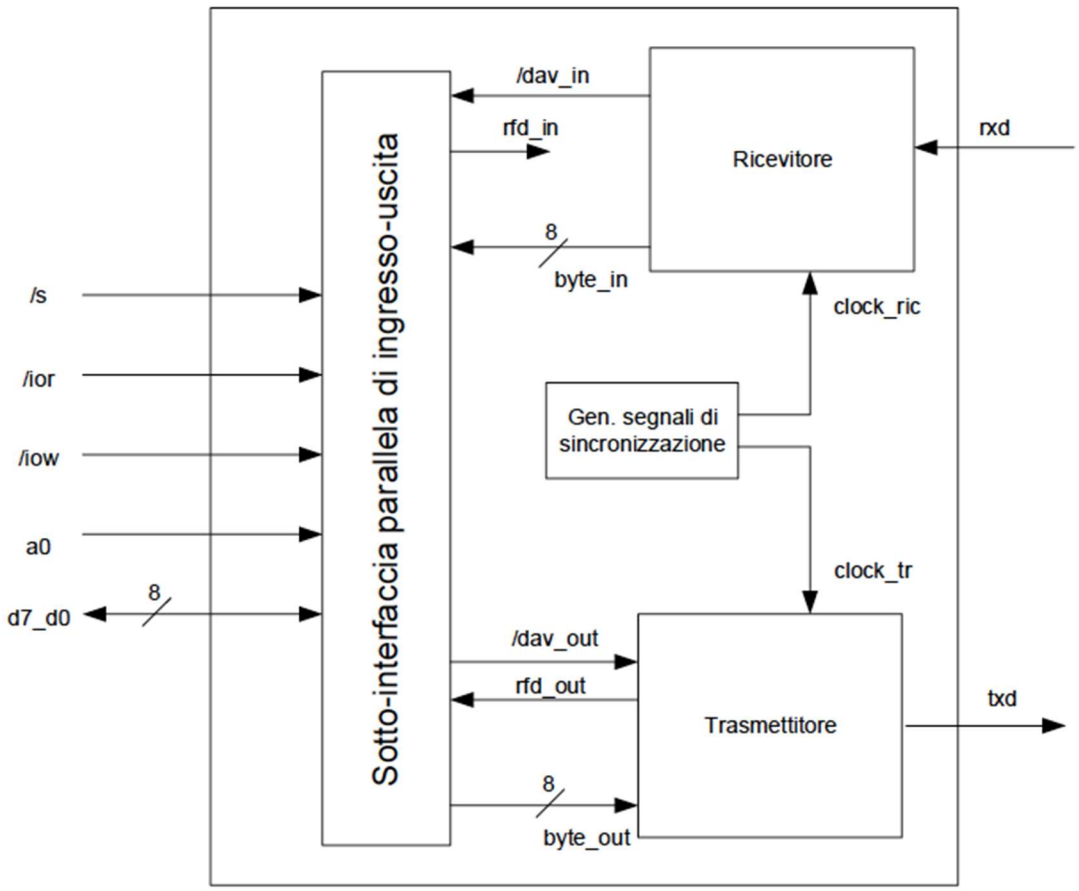
Abbiamo:

1. Un registro di stato `RTSR`, in cui il bit 5 ed il bit 0 sono rispettivamente il flag di **uscita vuota FO** e il flag di **ingresso pieno FI.**
2. Un registro `RTBR` ad 8 bit che serve per contenere i dati da trasmettere o quelli ricevuti.

STRUTTURA INTERNA

Dal punto di vista della struttura interna, è costituita da:

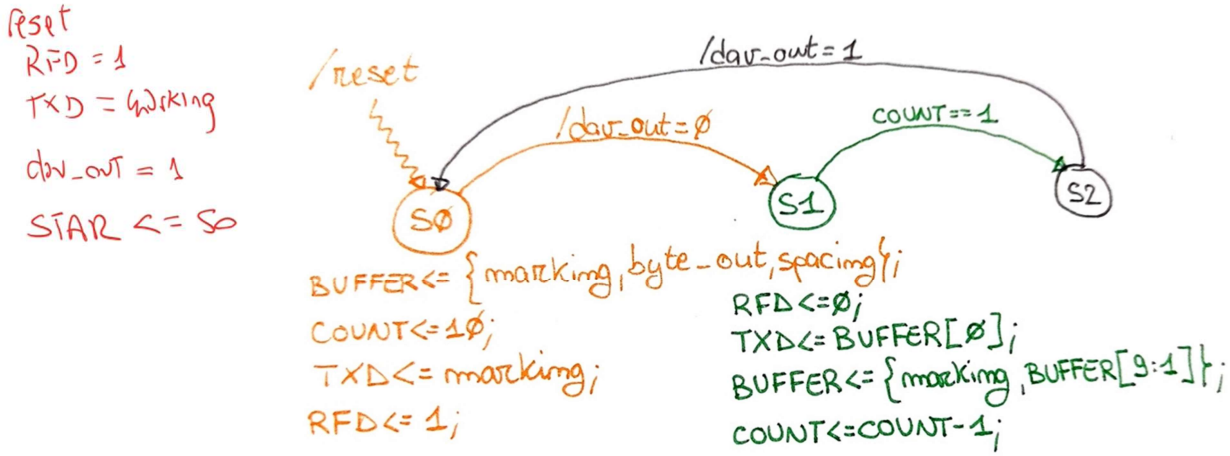
1. Una **sottointerfaccia parallela di ingresso/uscita con handshake**.
2. **Due reti sequenziali sincronizzate** chiamate *ricevitore* e *trasmettitore*.
3. Un **generatore di segnali di sincronizzazione**.



Il ricevitore riceve le trame dall'apparato esterno e presenta i byte utili contenuti in esse alla sezione di ingresso della sotto-interfaccia parallela.

Il trasmettitore riceve i byte utili dalla sezione di uscita della sotto-interfaccia parallela, costruisce le trame e le invia all'apparato esterno.

DESCRIZIONE DEL TRASMETTITORE

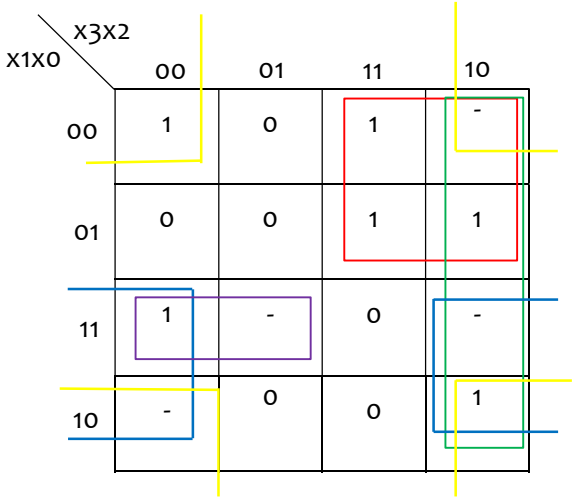



```
module Trasmettitore(
      clock, reset_, dav_out_, byte_out,
      rfd_out, txd
);
```

```
      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, txd=TXD;

      reg [1:0] STAR;        localparam S0=0, S1=1, S2=2;
      localparam 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
                        BUFFER<={stop_bit, byte_out, start_bit};
                        COUNT<=10;
                        TXD<=mark;
                        RFD<=1;
                        STAR<=(dav_out_==0)?S1:S0;
                  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_==1)?S0:S2;
                  end
            endcase
endmodule
```

SINTESI A PORTE NAND

Per prima cosa, si fa la sintesi SP della mappa di Karnaugh data.



	x3	x2	x1	x0	
A	-	0	-	0	essenziale
B	1	-	0	-	essenziale
C	1	0	-	-	ass. eliminabile
D	-	0	1	-	sempl. eliminabile
E	0	-	1	1	sempl. eliminabile

La sintesi SP è quindi:

$$z = \overline{x2} \cdot \overline{x0} + x3 \cdot \overline{x1} + \overline{x2} \cdot x1$$

Una volta ottenuta la sintesi SP, si complementa due volte e si applica DeMorgan, ottenendo:

$$z = \overline{\overline{(\overline{x2} \cdot \overline{x0}) + (x3 \cdot \overline{x1}) + (\overline{x2} \cdot x1)}} = \overline{\overline{(\overline{x2} \cdot \overline{x0})} \cdot \overline{(x3 \cdot \overline{x1})} \cdot \overline{(\overline{x2} \cdot x1)}}$$

DIVISIONE TRA NUMERI INTERI

IPOTESI PRELIMINARI

Nel caso della divisione tra naturali, abbiamo visto che il risultato è unico se

$$0 \leq r \leq b - 1$$

Tuttavia, nel caso dei numeri interi, questo vincolo potrebbe non avere senso, in quanto *b* potrebbe essere negativo. Si devono quindi aggiungere dei vincoli che generalizzino il caso naturale. Un primo vincolo è

$$\boldsymbol{ABS(r) < ABS(b)}$$

Che racchiude il vincolo sui naturali. Tuttavia, anche questo vincolo da solo non è sufficiente per rendere univoco il risultato della divisione. L'ulteriore condizione è che **il resto abbia lo stesso segno del dividendo.** Riassumendo, le ipotesi preliminari che rendono unico il risultato sono:

$$\begin{cases} \boldsymbol{ABS(r) < ABS(b)} \\ \boldsymbol{sgn(r) = sgn(a)} \end{cases}$$

Questa ipotesi vuol dire che nella divisione tra interi **il quoziente è approssimato per troncamento**; quindi, $q \cdot b$ è sempre più vicino all'origine rispetto ad *a*.

CONDIZIONI DI FATTIBILITA'

Sotto le ipotesi preliminari, possiamo scrivere la divisione nel seguente modo:

$$sgn(a) \cdot ABS(a) = q \cdot sgn(b) \cdot ABS(b) + sgn(r) \cdot ABS(r)$$

Dato che $sgn(a) = sgn(r)$ per tali ipotesi preliminari, moltiplicando entrambi i membri per $sgn(a)$ si ottiene:

$$\boldsymbol{sgn(a)} \cdot sgn(a) \cdot ABS(a) = [q \cdot sgn(b) \cdot \boldsymbol{sgn(a)}] \cdot ABS(b) + sgn(a) \cdot \boldsymbol{sgn(a)} \cdot ABS(r)$$

$$\Rightarrow ABS(a) = [q \cdot sgn(b) \cdot sgn(a)] \cdot ABS(b) + ABS(r)$$

Che è una **divisione tra naturali**. Infatti, se il dividendo e il divisore sono naturali, lo sarà anche il quoziente. Pertanto, *q* è negativo solo se *a* e *b* sono discordi. In ogni caso, l'espressione tra parentesi è positiva. Possiamo quindi definire $ABS(q) = q \cdot sgn(b) \cdot sgn(a)$.

Possiamo calcolare $ABS(r)$ e $ABS(q)$ utilizzando un modulo **divisore tra naturali**, purché $ABS(q)$ sia un **numero naturale rappresentabile su *n* cifre**. Per testare se è vero, dobbiamo controllare se

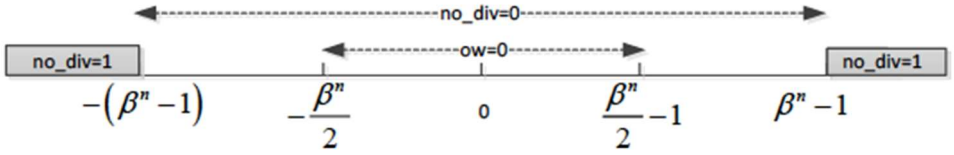
$$ABS(a) < \beta^n \cdot ABS(b)$$

Che si fa con un sottrattore ad *n+m* cifre, ovvero lo stesso che useremmo per testare la fattibilità di una divisione naturale. Se questa disuguaglianza è **falsa**, il quoziente della divisione naturale scritta sopra è **un numero naturale che sta su più di *n* cifre**, quindi $ABS(q) \geq \beta^n$.

In questo caso, il quoziente *q* della **divisione intera** è o $q \geq \beta^n$ (se *q* è positivo), oppure $q \leq -\beta^n$ (se *q* è negativo). Se $ABS(q)$ non è un naturale rappresentabile su *n* cifre, quindi, il quoziente *q* **non è un numero intero rappresentabile su *n* cifre.** Quindi,

La fattibilità della divisione naturale tra i valori assoluti è condizione <u>necessaria</u> per la fattibilità della divisione intera.

Come vediamo dalla seguente figura, si vede che questa **non è una condizione sufficiente**.



Quando *no_div=0*, non abbiamo garanzia che *q* sia un numero intero rappresentabile su *n* cifre.

Questo avviene perché **nella conversione della rappresentazione di *q* da MS a CR posso avere overflow.** Infatti, il fatto che la divisione **naturale** sia fattibile non implica che

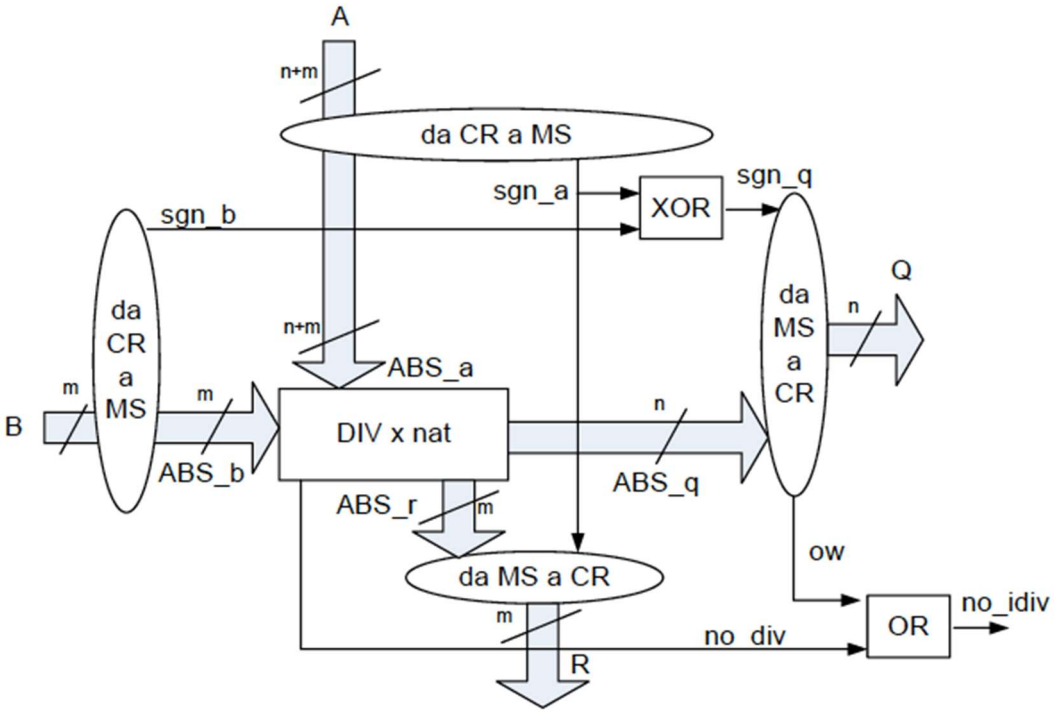
$$-\frac{\beta^n}{2} \leq q \leq \frac{\beta^n}{2} - 1$$

E quindi si può avere overflow. La seconda condizione per la fattibilità della divisione intera è quindi la seguente.

Il risultato della divisione naturale deve essere un valore assoluto minore di $\frac{\beta^n}{2}$, oppure uguale a $\frac{\beta^n}{2}$ con un segno negativo.

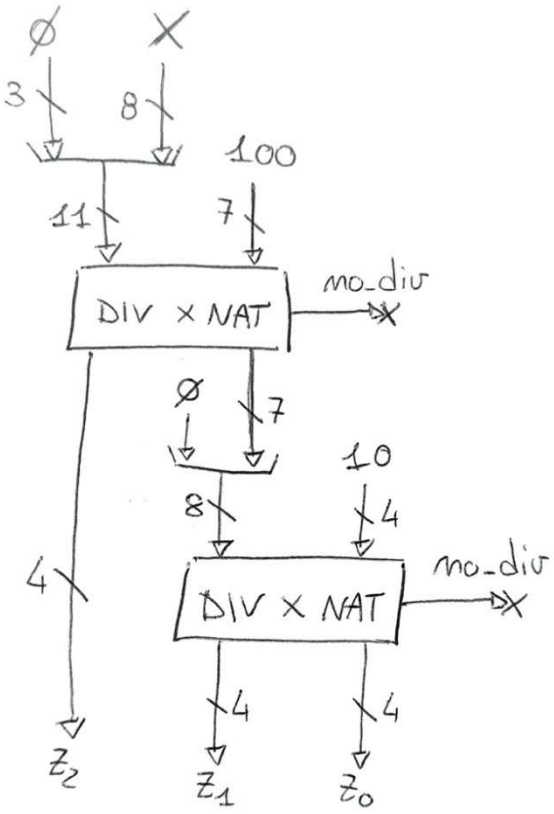
Questa condizione è esattamente ciò che ci dice il segnale di overflow in uscita al convertitore da MS a CR. Quindi, l'OR dei due mi dà il segnale *no_idiv*.

DISEGNO DEL CIRCUITO



SECONDO APPELLO

SINTETIZZARE LA RETE CHE CONVERTE UN NUMERO NATURALE DA BASE 2 A BASE 10. SI SUPPONGA CHE IL NUMERO DA CONVERTIRE SIA MINORE DI 256.



FASE DI FETCH DELL'ISTRUZIONE MOVB $operando, %AL

Devo leggere un byte di operando sorgente all'indirizzo puntato da `IP`, e metterlo in `SOURCE`.

```
fetchF4_0: begin A23_A0<=IP; IP<=IP+1; MJR<=fetchF4_1; STAR<=readB; end

fetchF4_1: begin SOURCE<=APP0; STAR<=fetchEnd; end
```

CONVERSIONE ANALOGICO-DIGITALE

ERRORI

Nella conversione analogico-digitale, la **tensione *v*,** ovvero la nostra grandezza analogica, da convertire sarà su una scala di *FSR* volts (**Full-Scale Range**). Il numero ***x* in base 2** nel quale sarà convertita è su N bit.

A seconda dell'interpretazione del numero della tensione, posso distinguere:

1. Conversione **unipolare:** $v \in [0, FSR],\ x \in [0, 2^N - 1]$
2. Conversione **bipolare:** $v \in \left[-\frac{FSR}{2}, +\frac{FSR}{2}\right],\ x \in [-2^{N-1}, 2^{N-1} - 1]$

Definiamo $K = \frac{FSR}{2}$ **costante di proporzionalità** fra i due intervalli. Una conversione *ideale* sarebbe

$$v = K \cdot x$$

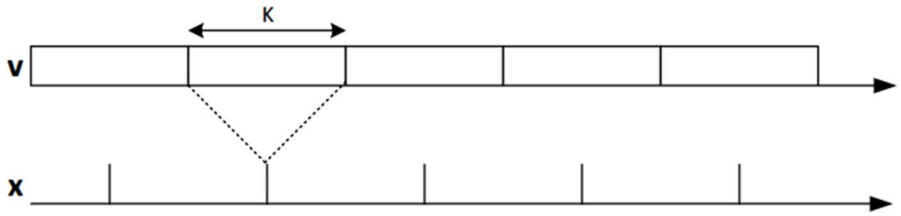
In realtà, dovremo accontentarci di $|v - K \cdot x| \leq err$, con *err* detto **errore di conversione.** Gli errori di conversione sono dovuti a:

1. Imprecisioni a livello **circuitale**: i convertitori sono circuiti con resistenze, fili, reattanze, che non si comportano in maniera **ideale**. Ci sarà un'imprecisione dovuta alla non idealità dei componenti. Viene chiamato **errore di non linearità**, ed è presente in entrambi i tipi di convertitori.
2. Quantizzazione. Nella conversione A/D (e soltanto in quella), devo convertire una grandezza **continua** in una **discreta**. Facendo questo si perde dell'informazione a causa dell'**arrotondamento**. Viene chiamato **errore di quantizzazione.**

L'errore di **non linearità** deve essere più piccolo di $\frac{K}{2}$.

L'errore massimo di **quantizzazione** non dipende dalla natura del convertitore. Data una costante K, è pari a $\frac{K}{2}$. Infatti, se divido il FSR in 2^N intervalli larghi K e converto tutto un intervallo nello stesso numero, la conversione sarà:

- **Esatta** per la tensione al **centro** dell'intervallo
- Errata di $\pm\frac{K}{2}$ per le tensioni agli **estremi**



Quindi, nella conversione analogico/digitale $err < \frac{K}{2} + \frac{K}{2} = K$.

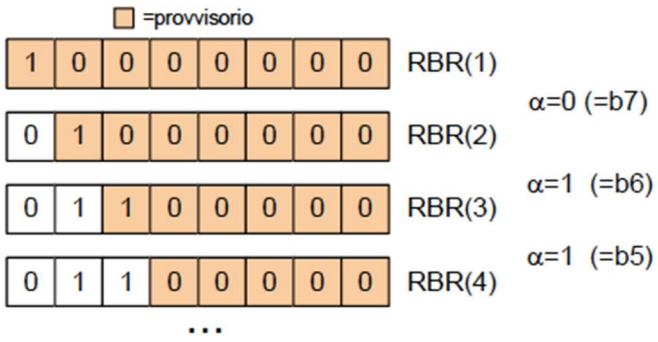
CONVERTITORE A/D

Descriviamo un convertitore A/D ad approssimazioni successive **ad 8 bit**.

La sua parte centrale è una RSS, con un suo clock, detta **SAR** (Successive Approssimation Register). Al suo interno ha, inoltre, un **convertitore D/A** (**dello stesso tipo** del convertitore A/D), con il quale fa la seguente cosa: quando riceve una tensione di ingresso, il SAR comincia una **ricerca logaritmica** per "indovinare" il byte corrispondente alla tensione fornita. Comincia producendo un byte intermedio, lo converte in analogico, confronta la tensione in ingresso con quella prodotta, e di conseguenza decide se produrre un nuovo byte più grande o più piccolo del precedente.

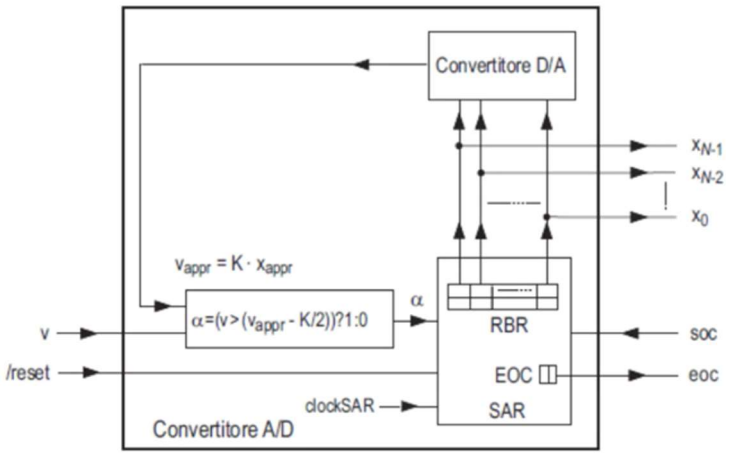
In base 2, la ricerca logaritmica diventa **mettere a posto un bit alla volta, partendo dal più significativo.** Quindi, si comincia presentando in uscita sul registro `RBR` il byte "al centro dell'intervallo di rappresentabilità". Sia che il convertitore lavori in modalità **unipolare**, sia che lavori in **bipolare**, il byte 10000000 (ovvero 2^{N-1}) è il centro dell'intervallo di rappresentabilità. Di fatto, **è il convertitore D/A a stabilire se il convertitore A/D lavorerà in unipolare o bipolare**: se il primo sarà bipolare, allora lo sarà anche il secondo e viceversa.

Questo byte va in ingresso al **convertitore D/A**, e da questo al **comparatore**. Se la **tensione esterna è maggiore**, allora il numero che la rappresenta sarà **più grande di 10000000.** Quindi, avrà di sicuro il primo bit a 1, altrimenti avrà il primo bit a 0. Quindi, α **rappresenta il bit più significativo del numero da convertire.**



Allora, al successivo clock, andrò a mettere a posto il **secondo bit** (da sinistra), che sarà dato ancora una volta dal valore di α, e così via.

Per terminare la ricerca logaritmica serve tempo. Quindi il convertitore A/D porta avanti, con l'interfaccia di conversione cui è connesso, un **handshake di tipo soc/eoc.** Inoltre, **la tensione analogica di ingresso deve rimanere costante per tutto il tempo di conversione.** Questa ipotesi viene resa vera inserendo, **prima del convertitore, un latch analogico**, che mantiene stabile la tensione.

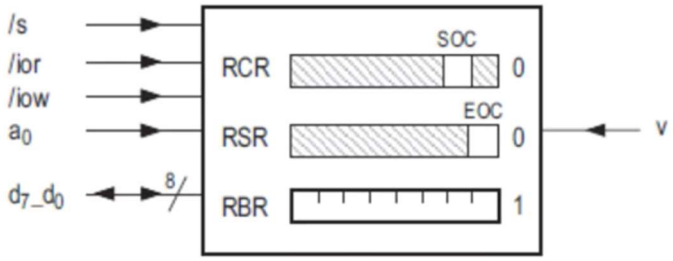


Perché è necessario il termine $\frac{K}{2}$ nella disequazione del comparatore? Serve principalmente a ridurre l'errore di quantizzazione: se infatti la disequazione fosse semplicemente v > v_appr avremmo come errore di quantizzazione Eq = LSB, con LSB pari al Less Significant Bit. Invece facendo ricorso a questa legge, apparentemente asimmetrica, l'errore risulterà pari a Eq = ½ * LSB, di fatto la metà del precedente.

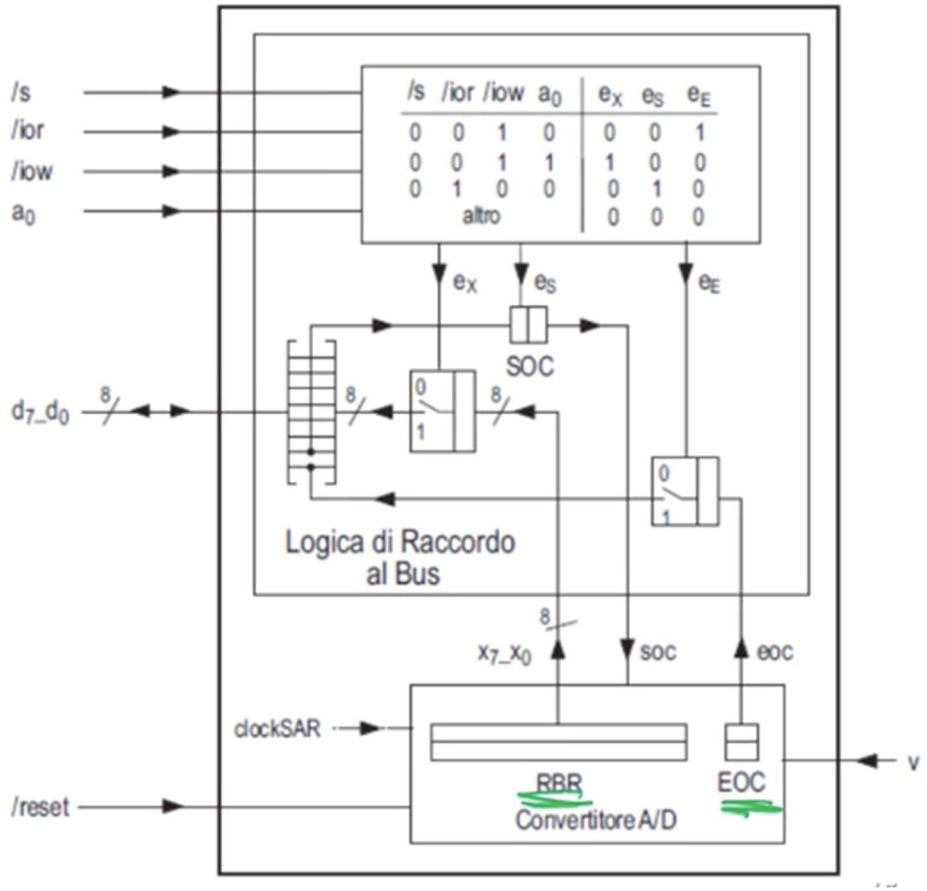
INTERFACCIA DI CONVERSIONE A/D

Dal punto di vista **funzionale**, sarà un'interfaccia di ingresso/uscita: infatti, la tensione convertita in numero è un ingresso per il processore, così come il segnale di `eoc`, ma il processore deve poter **scrivere** per poter iniziare una nuova conversione. Ci vorranno quindi **due registri:**

1. Un **Receive Status and Control Register** (`RSCR`), a 8 bit, con due bit significativi: `SOC` (bit 1), che può essere scritto, ed `EOC` (bit 0), che può essere letto.
2. Un **Receive Buffer Register** (RBR), a 8 bit, che (quando `EOC` vale 1) contiene il byte che converte l'ultima tensione di ingresso vista, secondo la legge del convertitore (unipolare o bipolare).



A livello di **struttura interna**, l'interfaccia di conversione dovrà abilitare delle tri-state per le proprie uscite (`eoc` e `x7_x0`, che vengono dal convertitore). Queste saranno abilitate quando il processore opera una **lettura**, discriminando la porta in base all'indirizzo. Inoltre, dovrà **inviare il segnale di memorizzazione** al registro `SOC`, quando il processore scrive nel `RSCR`. Quindi, c'è soltanto **logica combinatoria**. Si noti che `SOC` viene memorizzato sul fronte di discesa di `/iow` (per le scritture in memoria basta che siano buoni sul fronte di salita di `/mw`).



SOTTOPROGRAMMA PER L'INGRESSO DI UN VALORE DALL'INTERFACCIA DI CONVERSIONE A/D

In Assembler:

```
            MOV $0x01, %AL
            OUT %AL, RCR_offset         # SOC=1

test1:      IN RSR_offset, %AL
            AND $0x01, %AL
            JNZ test1                   # attendi EOC=0
            MOV $0x00, %AL
            OUT %AL, RCR_offset         # SOC=0
```

```
test2:      IN RSR_offset, %AL
            AND $0x01, %AL
            JZ test2                    # attendi EOC=1
            IN RBR_offset, %AL          # prelievo dato convertito
            RET
```

In C++:

```
byte acquisizione( ) {
      #define RCR_offset ...
      #define RSR_offset RCR_offset
      #define RBR_offset ...
      byte tmp;

      //Attiva la conversione immettendo 1 in SOC
      outport(RCR_offset,0x02);

      //Attende che il contenuto di EOC vada a 0 e quindi immette 0 in
      SOC
      do {tmp=inport(RSR_offset)&0x01;} while (tmp!=0x00);
      outport(RCR_offset,0x00);

      //Attende che il contenuto di EOC vada a 1
      do {tmp=inport(RSR_offset)&0x01;} while (tmp==0x00);

      //Ritorna il risultato della conversione
      return inport(RBR_offset);
}
```

Visto che il convertitore è **molto veloce ad iniziare la conversione**, possiamo evitare di attendere che EOC vada a zero, e togliere le due parti riquadrate. **Attenzione** a non confondere questa cosa con il fatto di omettere di testare eoc in un esercizio di descrizione. Se ho un esercizio con un convertitore A/D, e devo descrivere una rete che si interfaccia con esso per ottenere dei dati, **non posso certo scrivere:**

```
S1: begin SOC<=1; STAR<=S2; end
S2: begin SOC<=0; … end
```

Ma devo scrivere qualcosa di equivalente a

```
S1: begin SOC<=1; STAR<=S2; end
S2: begin SOC<=eoc; STAR<=(eoc==1)?S2:S3; end
```

PROCEDIMENTO PER INDIVIDUARE FORMA CANONICA SP, MINTERMINE, IMPLICANTE, IMPLICANTE PRINCIPALE, IP ESSENZIALE, IP ASSOLUTAMENTE ELIMINABILE

Come punto di partenza, consideriamo il seguente risultato (dovuto a Shannon):

"è sempre possibile scrivere **qualunque** legge F di una rete combinatoria come **somma di prodotti degli ingressi (diretti o negati)**".

Data una legge $z = f(x_{N-1}, \dots, x_0)$, possiamo scrivere l'**espansione di Shannon** come segue:

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

$$+f(0,0,\dots,1)\cdot\overline{x_{N-1}}\cdot\overline{x_{N-2}}\cdot\dots\cdot\overline{x_1}\cdot x_0$$

$$\dots$$

$$+f(1,1,\dots,0)\cdot x_{N-1}\cdot x_{N-2}\cdot\dots\cdot x_1\cdot\overline{x_0}$$

$$+f(1,1,\dots,1)\cdot x_{N-1}\cdot x_{N-2}\cdot\dots\cdot x_1\cdot x_0$$

A partire da questa espansione, possiamo ottenere la **forma canonica SP** (SP perché somma di prodotti, canonica perché ogni prodotto ha come fattori *tutti gli ingressi* diretti o negati) applicando le seguenti proprietà:

- $0\cdot\alpha=0$
- $0+\alpha=\alpha$
- $1\cdot\alpha=\alpha$

E osservando che, per le precedenti tre proprietà:

1. Se $f(x_{N-1},\dots,x_0)=0$, allora tutto il termine corrispondente sulla riga vale 0.
2. Se $f(x_{N-1},\dots,x_0)=1$, allora posso togliere uno dei fattori dal prodotto.

La forma canonica SP prende anche il nome di **lista dei mintermini.** Un mintermine è un prodotto di *tutte* le variabili di ingresso dirette o negate, che compare in una forma canonica SP e riconosce *uno stato di ingresso.*

Il prossimo passo è quello di ottenere la **lista degli implicanti** applicando esaustivamente le seguenti proprietà:

1. $\boldsymbol{\alpha x+\alpha\bar{x}=\alpha x+\alpha\bar{x}+\alpha}$, che consente di **fondere i mintermini**. Dati due termini che differiscono solo per **una sola variabile**, che è diretta in un caso e negata nell'altro, posso produrre un termine che contiene ciò che è a comune tra i due mintermini che fondono.
2. $\boldsymbol{\alpha+\alpha=\alpha}$, che consente di **eliminare i duplicati**.

Questa lista è ancora in forma SP, e ciascun termine di questa è detto **implicante**. Un implicante è il prodotto di **alcune** variabili di ingresso dirette o negate che riconosce alcuni stati di ingresso. Un mintermine è un caso particolare di implicante.

Posso semplificare ulteriormente la lista, applicando esaustivamente la legge $\alpha x+\alpha=\alpha$. Questo equivale a **togliere tutti gli implicanti che hanno prodotto qualcosa per fusione**. Il risultato è la **lista degli implicanti principali.** Un implicante è detto principale se non è in grado di fondere con nessun altro implicante.

IP ESSENZIALE: implicante che è **l'unico**, tra quelli principali, ad implicare un dato mintermine, ovvero a riconoscere uno stato di ingresso.

IP ASSOLUTAMENTE ELIMINABILE: implicante che riconosce solo stati di ingresso già riconosciuti da IP essenziali.

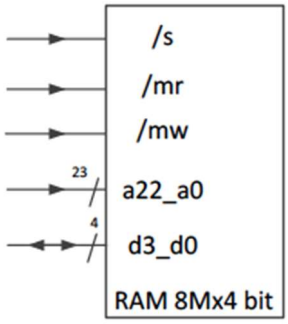
IP SEMPLICEMENTE ELIMINABILE: implicante che riconosce solo stati di ingresso riconosciuti da altri IP, almeno uno dei quali riconosciuto da un IP non essenziale.

MEMORIA RAM STATICA

SCHEMA

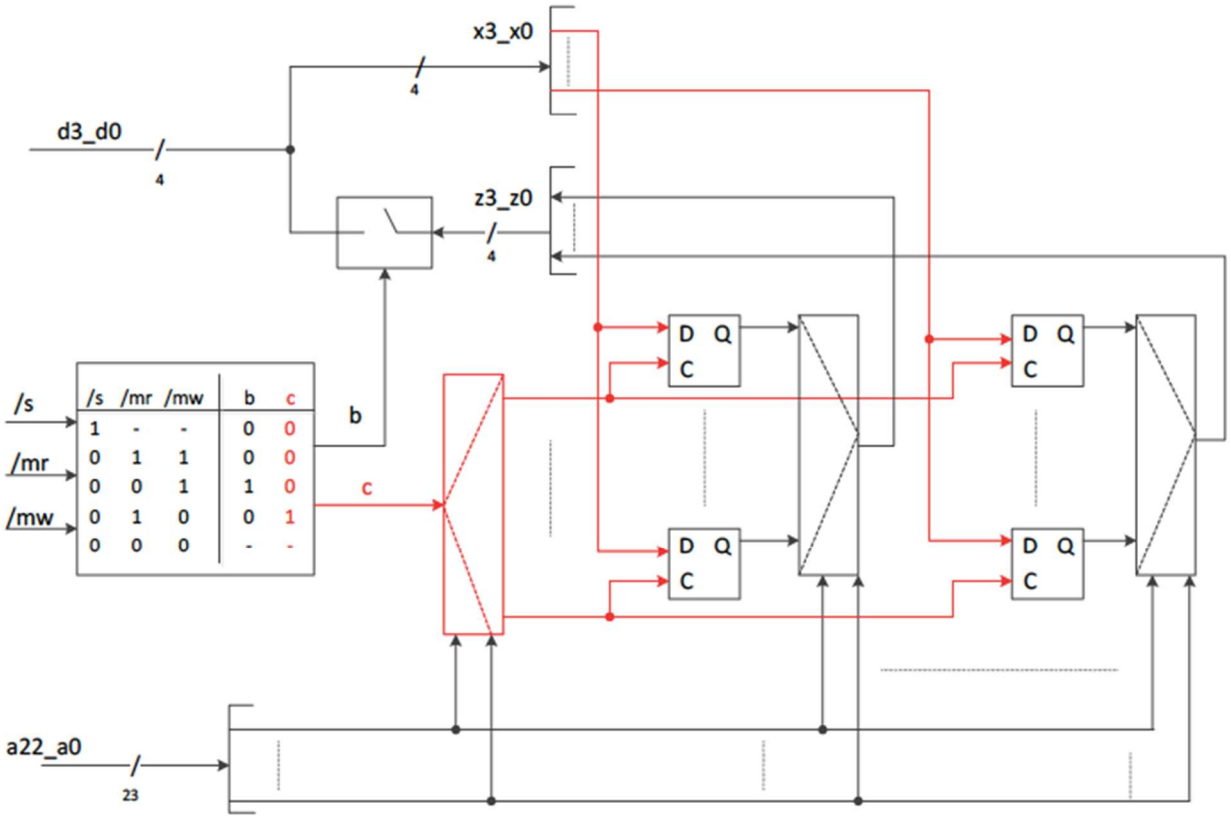
Le RAM statiche sono batterie di D-Latch montati a **matrice**. Una riga di D-Latch costituisce una **locazione di memoria**, che può essere **letta o scritta** con un'operazione di lettura o scrittura. Le operazioni di lettura e scrittura **non possono essere simultanee**.

Dal punto di vista dell'utente, una memoria è dotata dei seguenti collegamenti:



- Un certo numero di **fili di indirizzo**, che sono **ingressi**, in numero sufficiente ad indirizzare tutte le celle della memoria.
- Un certo numero di **fili di dati**, che sono fili di **ingresso/uscita**, e come tali andranno **forchettati con porte tri-state**.
- Due segnali **attivi bassi** di **memory read e memory write**. Non dovranno mai essere attivi contemporaneamente. Servono a dare il comando di lettura o scrittura della cella il cui indirizzo è trasportato sui fili di indirizzo.
- Un segnale attivo basso di **select**, che viene attivato quando la memoria è selezionata. Quando /s vale 1, la memoria è insensibile a tutti gli ingressi. Quando vale 0, la memoria è selezionata, e reagisce agli ingressi.

/s	*/mr*	*/mw*	**Azione**	***b***	***c***
1	-	-	Nessuna azione (memoria non selezionata)	0	0
0	1	1	Nessuna azione (memoria selezionata, nessun ciclo in corso)	0	0
0	0	1	Ciclo di lettura in corso	1	0
0	1	0	Ciclo di scrittura in corso	0	1
0	0	0	Non definito	-	-

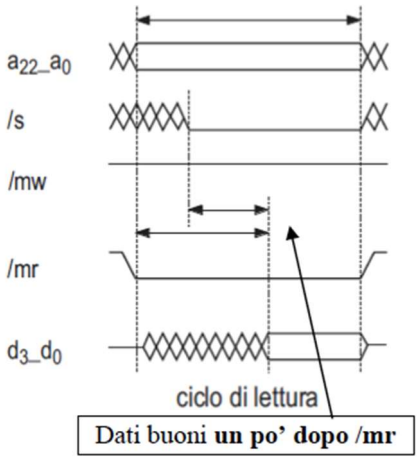


- Una riga di D-Latch è una **locazione**.

- Le uscite dei D-Latch dovranno essere selezionate **una riga alla volta**, per finire sui fili di dati in uscita. Ci vuole **un multiplexer per ogni bit**, in cui:
    - Gli ingressi sono le uscite dei D-Latch
    - Le variabili di comando sono i **fili di indirizzo**
- Le uscite di ciascuno dei multiplexer vanno **bloccate** con altrettante tri-state. Queste dovranno essere abilitate **quando sto leggendo dalla memoria**. Ci vuole una RC che mi produca l'enable **b** come funzione di /s, /mr, /mw.
- Per quanto riguarda gli ingressi: posso portare a ciascuna **colonna** di D-Latch i fili di dati sull'ingresso **d**. Basta che faccia in modo che, quando voglio **scrivere**, **soltanto una riga di D-Latch** sia abilitata, cioè abbia **c a 1**. Quindi, ciascuna riga di D-Latch avrà l'ingresso **c** prodotto da un **demultiplexer**, comandato dai fili di indirizzo. Questo demultiplexer **commuterà sulla riga giusta** il comando di scrittura, attivando solo una riga di **c** alla volta. In questo modo, anche se i fili di dati vanno in ingresso contemporaneamente a tutti i D-Latch, solo una riga li sentirà.

TEMPORIZZAZIONE DEL CICLO DI LETTURA

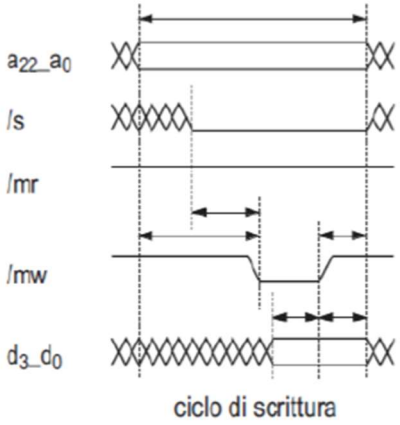
Ad un certo istante, gli indirizzi si stabilizzano al valore della cella che voglio leggere ed arriva il comando di **/mr**. Il comando di /s arriva con un po' di ritardo, e **balla** nel frattempo, essendo funzione combinatoria di altri bit di indirizzo.



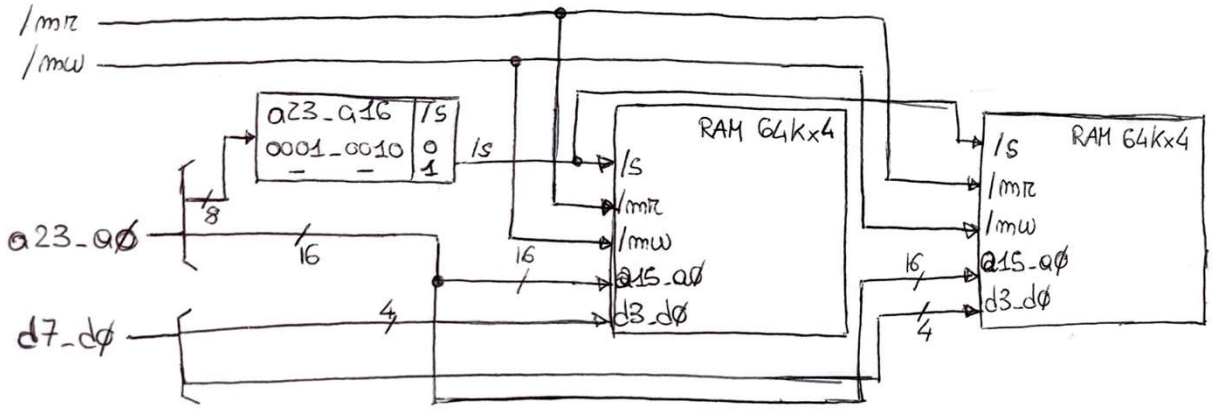
Quando sia /s che /mr sono a 0, dopo un po' le porte tri-state vanno in conduzione, e i multiplexer sulle uscite vanno a regime. Da quel punto in poi i dati sono buoni, e chi li ha chiesti li può prelevare. Quando /mr viene tirato nuovamente su (il che verrà fatto quando **chi voleva leggere i dati li ha già prelevati**), i dati tornano in alta impedenza. A quel punto gli indirizzi e /s possono ballare a piacere, tanto non succede niente.

TEMPORIZZAZIONE DEL CICLO DI SCRITTURA

Visto che la scrittura è **distruttiva** (quando scrivo i D-Latch sono in trasparenza), devo **attendere che /s e gli indirizzi siano stabili** prima di portare giù **/mw**. I dati, invece, possono ballare a piacimento (anche quando /mw vale 0), ma devono **essere buoni a cavallo del fronte di salita di /mw**. Tale fronte corrisponde, infatti, (con un minimo di ritardo dovuto alla rete C ed al multiplexer), al fronte di discesa di c sui D-Latch. Il tempo per cui devono essere tenuti buoni dopo il fronte di salita di /mw è maggiore di T_{hold}, in quanto c'è dell'altra logica davanti.



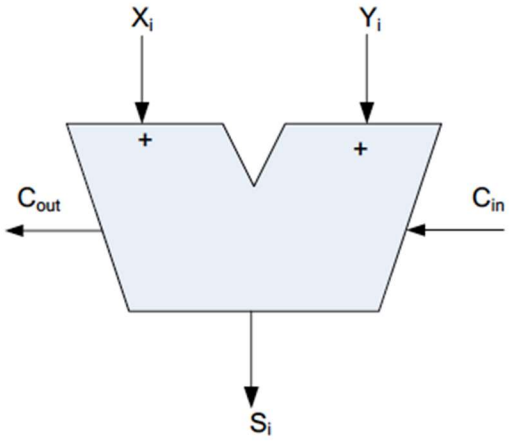
CONNESSIONE AL BUS DEL CALCOLATORE VISTO A LEZIONE DI DUE MODULI DI RAM 64Kx4 BIT, IN MODO CHE IMPLEMENTINO LE CELLE DI MEMORIA A PARTIRE DALL'INDIRIZZO 'H120000



TERZO APPELLO

FULL-ADDER

DISEGNARE LA STRUTTURA INTERNA DI UN FULL-ADDER



Xi	yi	Cin	Si	Cout
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

Si

Cin \ XiYi	00	01	11	10
0	0	1	0	1
1	1	0	1	0

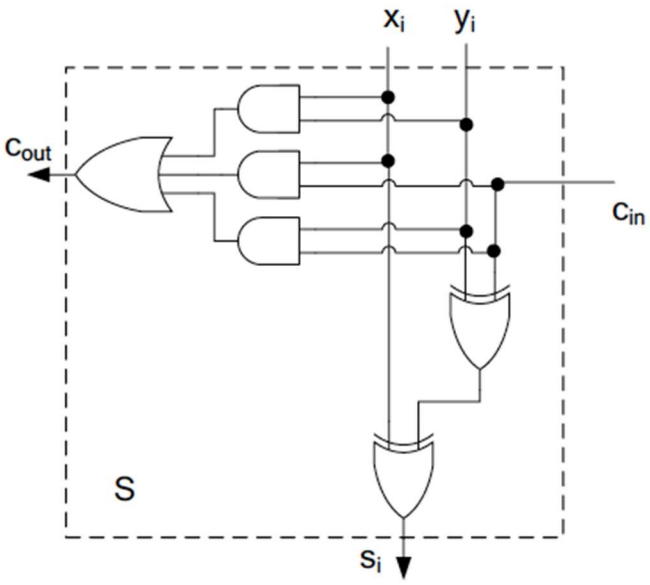
Cout

Cin \ XiYi	00	01	11	10
0	0	0	1 (A)	0
1	0	1 (B)	1	1 (C)

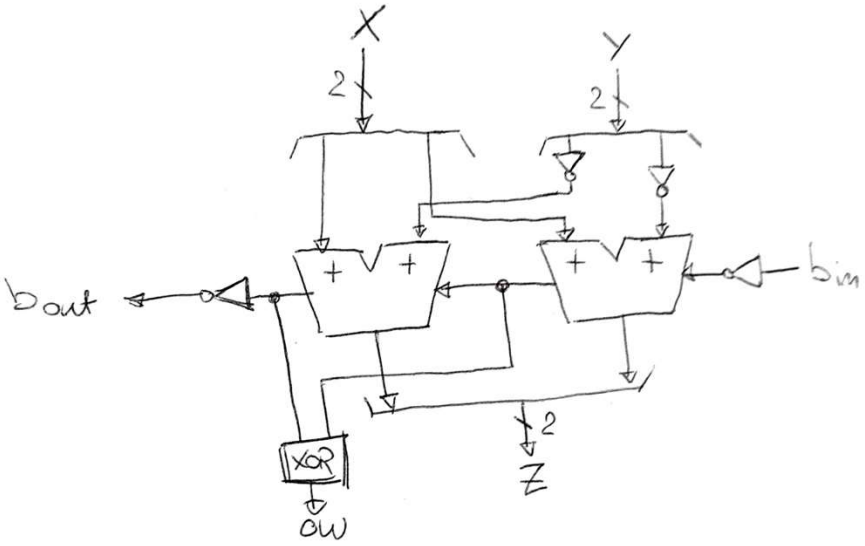
$$S_i = \overline{X_i}\,\overline{y_i}\,C_{in} + \overline{X_i}\,y_i\,\overline{C_{in}} + X_i\,y_i\,C_{in} + X_i\,\overline{y_i}\,\overline{C_{in}} = \overline{X_i}\left(\overline{y_i}\,C_{in} + y_i\,\overline{C_{in}}\right) + X_i\left(y_i\,C_{in} + \overline{y_i}\,\overline{C_{in}}\right) =$$

$$= \overline{X_i}\left(y_i \oplus C_{in}\right) + X_i\left(\overline{y_i \oplus C_{in}}\right) = X_i \oplus \left(y_i \oplus C_{in}\right)$$

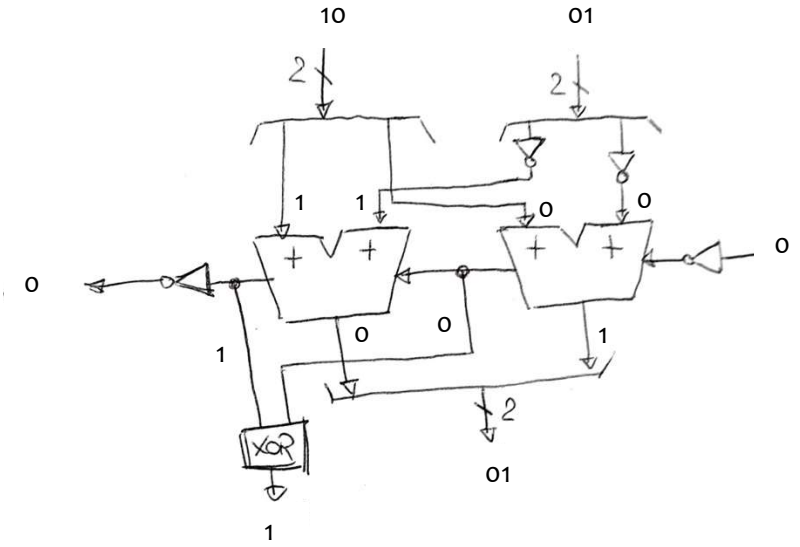
$$C_{out} = X_i y_i + y_i C_{in} + X_i C_{in}$$



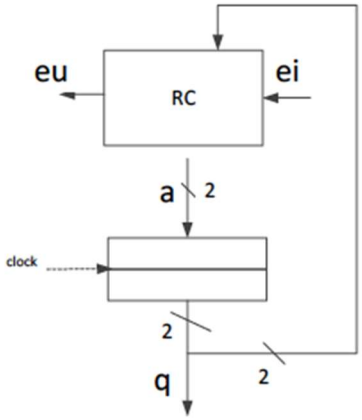
SINTETIZZARE UN CIRCUITO SOTTRATTORE A DUE CIFRE IN BASE 2 UTILIZZANDO FULL-ADDER A UNA CIFRA, COMPLETO DELL'USCITA DI OVERFLOW.



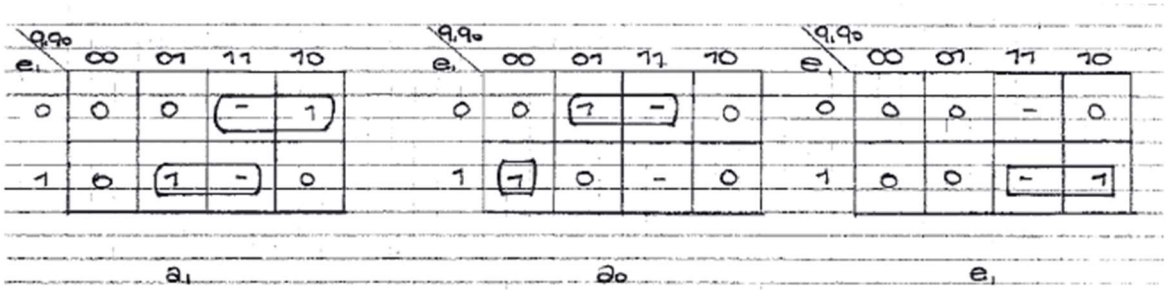
RAPPRESENTARE LE USCITE QUANDO GLI INGRESSI SONO b_{in}=0, X=10, Y=01



CONTATORE ESPANDIBILE AD UNA CIFRA IN BASE 3. SINTETIZZARE FINO ALLA OTTIMIZZAZIONE DELLA PARTE COMBINATORIA. COSA CONTA IL CONTATORE?



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			-	-	-



$$a_1 = q_1 \cdot \overline{e_i} + q_0 \cdot e_i \qquad a_0 = q_0 \cdot \overline{e_i} + \overline{q_1} \cdot \overline{q_0} \cdot e_i \qquad e_u = q_1 \cdot e_i$$

I contatori "**dividono in frequenza**". Possono quindi essere usati per **dividere la frequenza del clock** per un certo valore. Ad esempio, posso usare il **bit più significativo** dell'uscita del contatore in base 3 per ottenere un clock che va 3 volte più lento del clock del contatore.

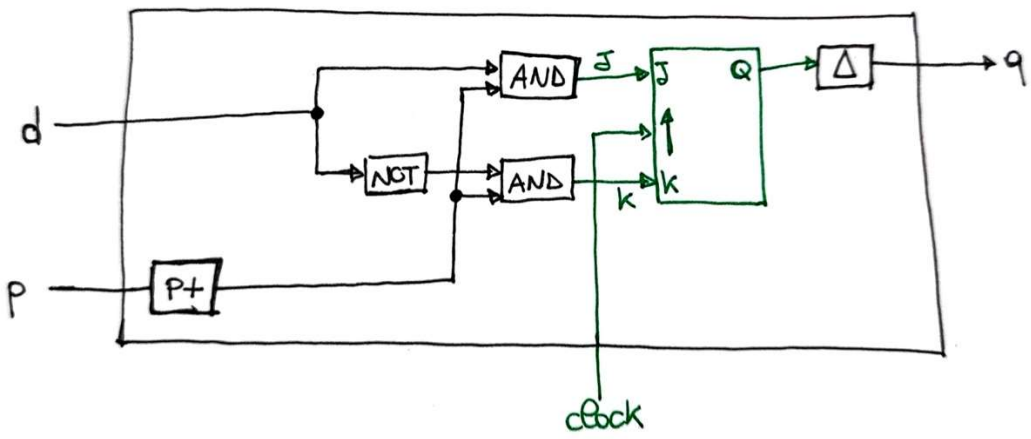
In generale, la **cifra più significativa** di un contatore ad N cifre in base β che riceve clock a periodo T è un clock a periodo $\beta^N \cdot T$.

Dato il processore visto a lezione, scrivere la fase di fetch di un nuovo formato di istruzioni (si assuma che esista una terna di bit utilizzabile per indicare un nuovo formato). Le istruzioni sono del tipo

OPCODE $byte, indirizzo

```
fetch0:    begin A23_A0<=IP; IP<=IP+1; MJR<=Fetch1; STAR<=readB; end

fetch1:    begin OPCODE<=APP0; STAR<=fetch2; end

fetch2:    begin STAR<=(valid_fetch(OPCODE)==1)?fetch3:nvi; end

nvi:       begin STAR<=nvi; end

fetch3:    begin A23_A0<=IP; IP<=IP+1; MJR<=fetch4; STAR<=readB; end

fetch4:    begin SOURCE<=APP0; STAR<=fetch5; end

fetch5:    begin A23_A0<=IP; IP<=IP+3; MJR<=fetch6; STAR<=readM; end

fetch6:    begin DEST_ADDR<={APP2,APP1,APP0}; STAR<=fetchEND; end

fetchEND:  begin MJR<=first_execution_statement(OPCODE);
           STAR<=fetchEND_1; end

fetchEND_1:     begin STAR<=MJR; end
```

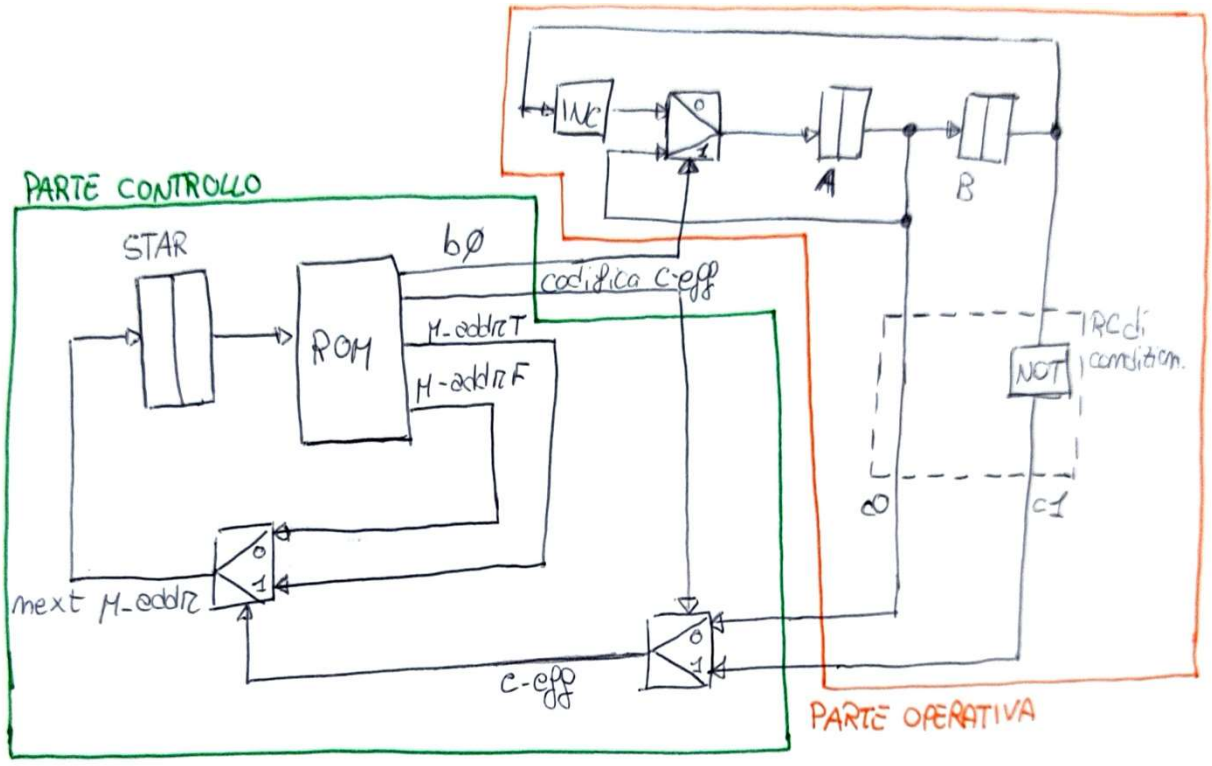
REALIZZARE UN FLIP-FLOP D-POSITIVE EDGE-TRIGGERED PARTENDO DA UN FLIP-FLOP JK



SINTETIZZARE L'UNITA' XXX DISEGNANDO I CIRCUITI DELLA PARTE OPERATIVA E DELLA PARTE CONTROLLO, E SCRIVENDO LA SINTESI ANCHE IN VERILOG.

```
module XXX(clock, reset_);
      input clock, reset_;
      reg A,B, STAR;
      parameter S0 = 0, S1 = 1;

      always @(reset_ == 0) begin A <= 0; B <= 0, STAR <= S0; end
      always @(posedge clock) if(reset_ == 1) #3
            casex(STAR)
                  S0: begin A <= B+1; B <= A; STAR <= (A==1)? S1: S0; end
                  S1: begin B <= A; STAR <= (B == 0) ? S1 : S0; end
      endcase
endmodule


module XXX(clock, reset_);
      input clock, reset_;

      wire b0, c0, c1;
      Parte_Operativa PO(clock, reset_, b0, c0, c1);
      Parte_Controllo PC(clock, reset_, b0, c0, c1)
endmodule

module Parte_Operativa(clock, reset_, b0, c0, c1);
      input clock, reset_;
      reg A, B;

      input b0;
      output c0, c1;
      assign c0=A;
      assign c1=~B;
      //registro A
      always @(reset_==0) #1 A<=0;
      always @(posedge clock) if(reset_==1) #3
            casex(b0)
                  0: A<=B+1;
                  1: A<=A;
            endcase
      //registro B
      always @(reset_==0) #1 B<=0;
      always @(posedge clock) if(reset_==1) #3 B<=A;
endmodule

module Parte_Controlo(clock, reset_, b0, c0, c1);
      input clock, reset_;
```

```
      reg STAR;  parameter S0=0, S1=1;

      input c0, c1;
      output b0;
      assign b0=(STAR==S0)?0:1;

      always @(reset_==0) #1 STAR<=S0;
      always @(posedge clock) if(reset_==1) #3
            casex(STAR)
                  S0: STAR<=(c0==1)?S1:S0;
                  S1: STAR<=(c1==1)?S1:S0;
            endcase
endmodule

/*          |          m-instructions
            |-------------------------------------
m-addr      |     b0    c-eff      m-addrT    m-addrF
0           |     0     0          1          1
1           |     1     1          0          0
*/
```

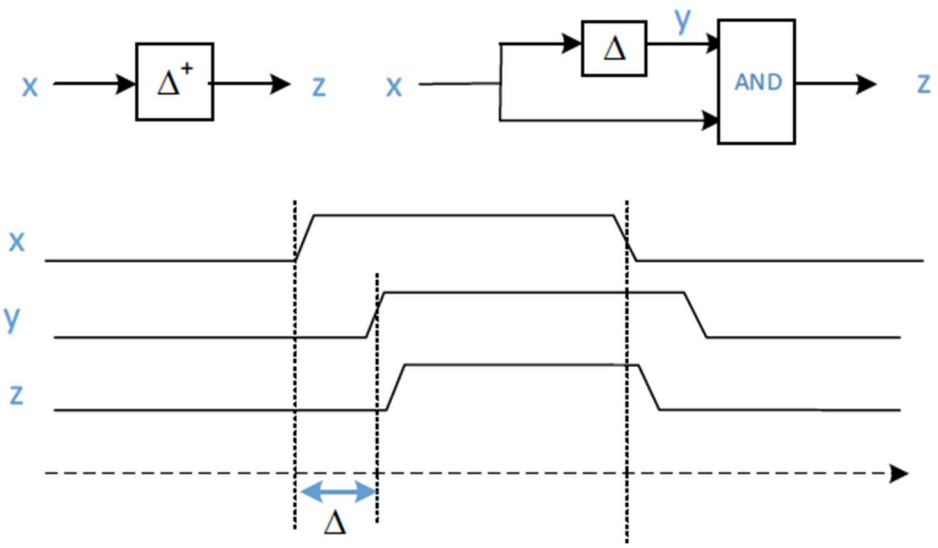


ORALE STRAORDINARIO DI MARZO – A.A 2020-2021

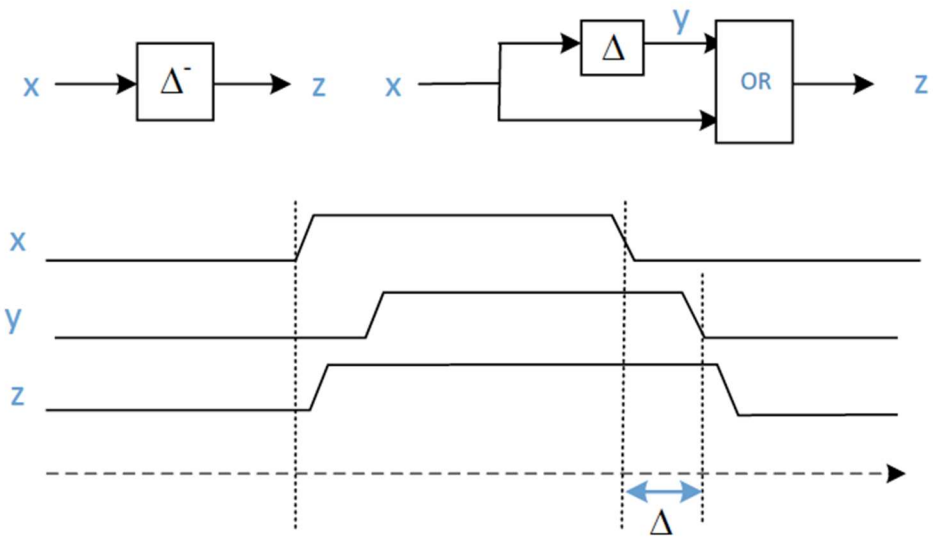
DESCRIVERE I FORMATORI DI IMPULSO E I CIRCUITI DI RITARDO

I **circuiti di ritardo** sono di due tipi:

1. Con ritardo **grande sulla transizione 0-1** e piccolo sulla transizione 1-0 (Δ^+)

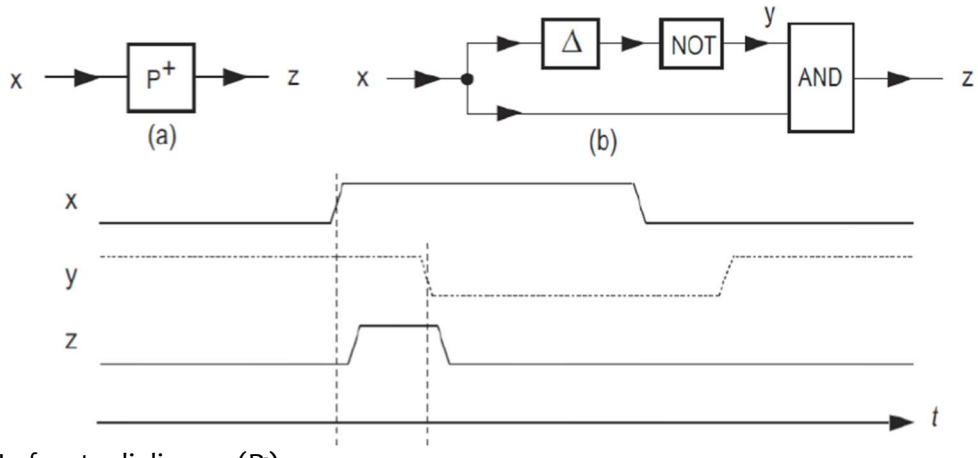


2. Con ritardo **grande sulla transizione 1-0** e piccolo sulla transizione 0-1 (Δ^-)

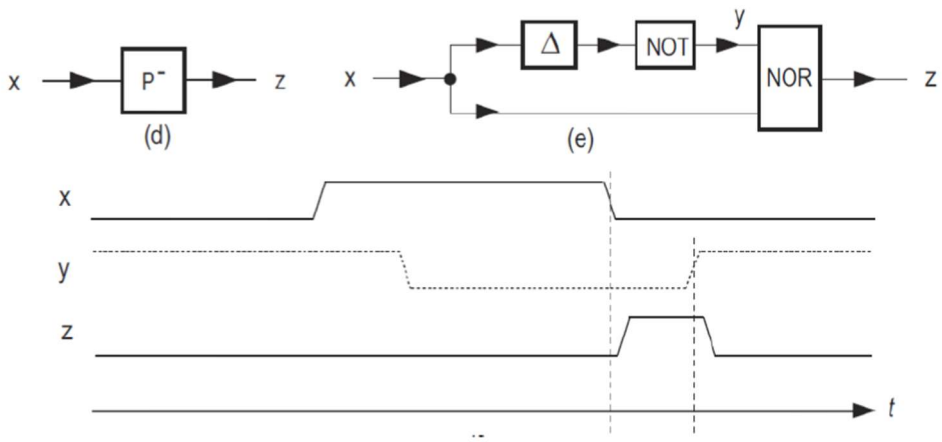


I **formatori di impulsi** servono invece a generare un **impulso di durata nota sull'uscita** quando l'ingresso ha:

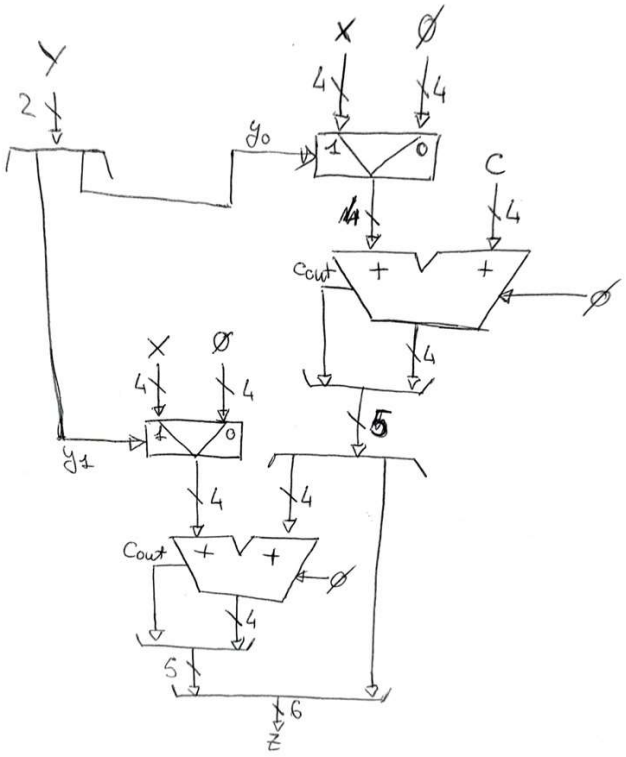
1. Un fronte di salita (P^+)



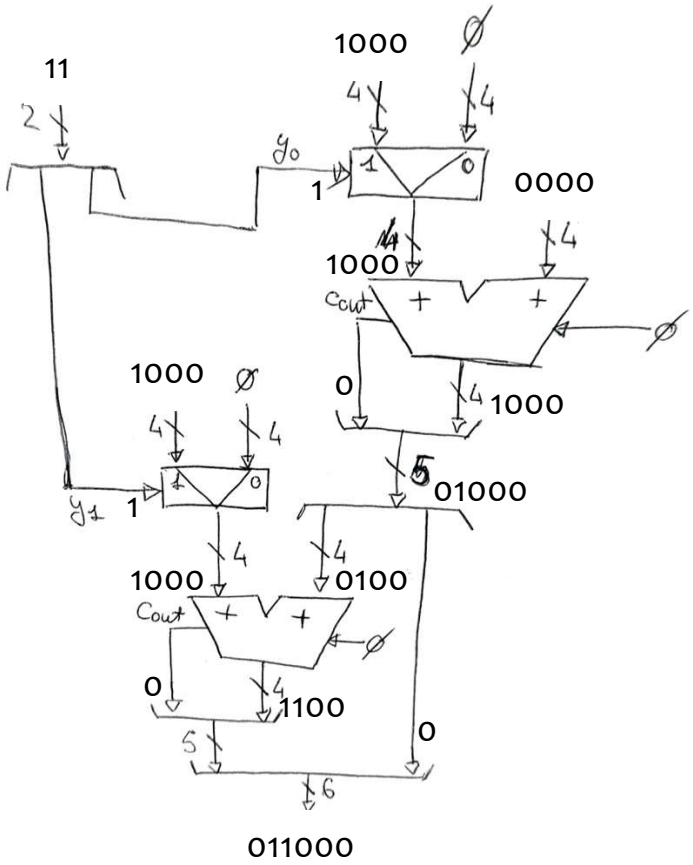
2. Un fronte di discesa (P^-)



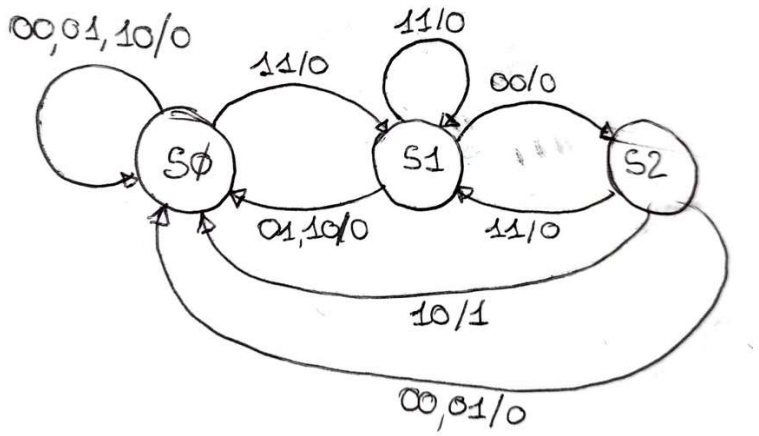
SINTETIZZARE UN MOLTIPLICATORE CON ADDIZIONE PER NATURALI 4x2 CIFRE IN BASE 2, SCOMPONENDOLO FINO AL LIVELLO DI SOMMATORI ED ALTRE RETI ELEMENTARI.



EVIDENZIARE I VALORI CHE COMPAIONO SU TUTTI I FILI QUANDO GLI INGRESSI SONO X=1000, Y=11, C=0000



DESCRIZIONE DEL RICONOSCITORE DI SEQUENZE 11,00,10 COME RETE DI MEALY



x1xØ	00	01	11	10
SØ	S0/0	S0/0	S1/0	S0/0
S1	S2/0	S0/0	S1/0	S0/0
S2	S0/0	S0/0	S1/0	S0/1

```
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=='B00)?S2:(x1_x0=='B11)?S1:S0;
                S2: STAR<=(x1_x0=='B11)?S1:S0;
          endcase
endmodule
```

FASE DI ESECUZIONE DELL'ISTRUZIONE IN offset, %AL

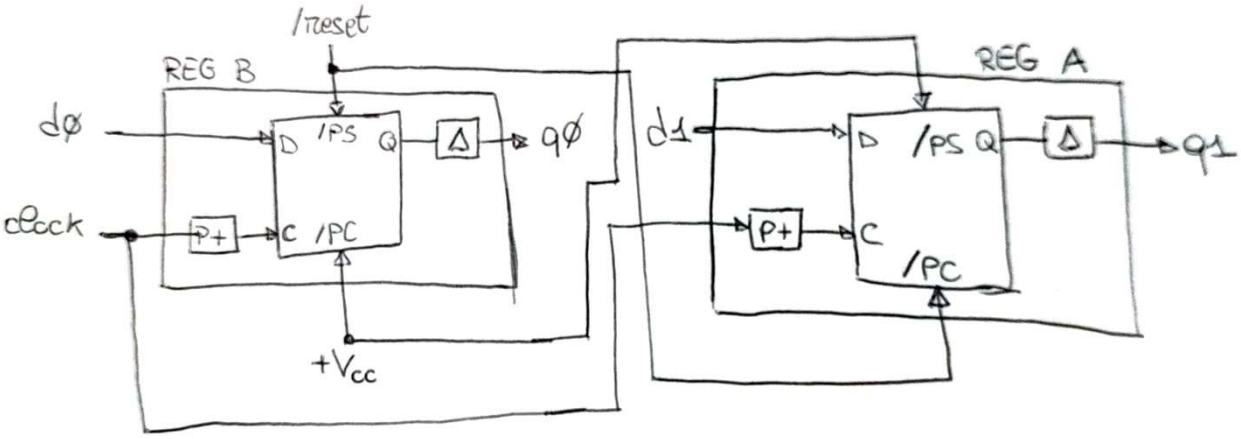
```
In1: begin A23_A0<=IP; IP<=IP+2; STAR<=readW; MJR<=In2; end

In2: begin A23_A0<={'H00, APP1, APP0}; STAR<=In3; end

In3: begin IOR_<=0; STAR<=In4; end

In4: begin IOR_<=1; AL<=d7_d0; STAR<=fetch0; end
```

SINTESIZZARE LA SEGUENTE PORZIONE DI RSS

```
reg A,B;

always @(reset_) begin A <= 0; B <= 1; end

always @(posedge clock) if(reset_ == 1) #3 ...
```

Questa RSS è un circuito che, al reset, setta il registro B e resetta il registro A. Dato che i registri non sono nient'altro che D-FlipFlop, il cui ingresso p è il clock, per:

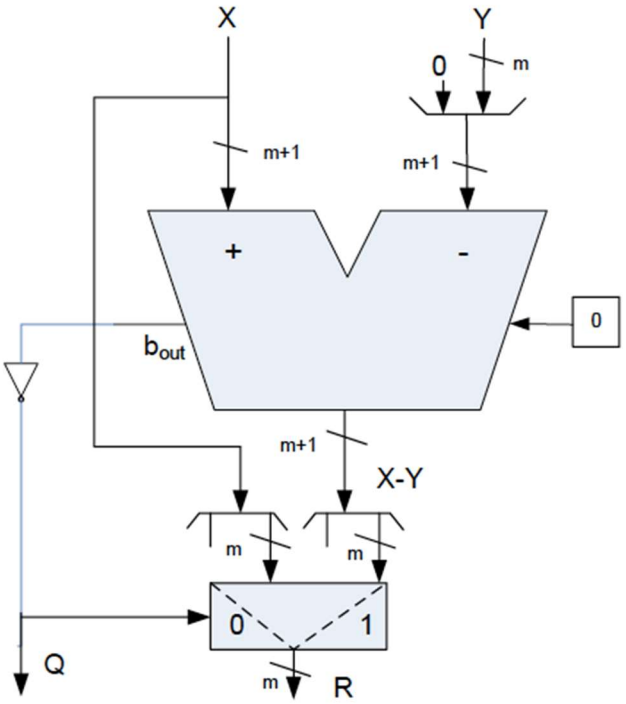
1. Settare il registro B, dovremo connettere il suo ingresso **/preset a /reset** e il suo ingresso **/preclear a Vcc.**
2. Resettare il registro A, dovremo connettere il suo ingresso **/preclear a /reset** e il suo ingresso **/preset a Vcc.**



SESSIONE ESTIVA – A.A. 2020-2021

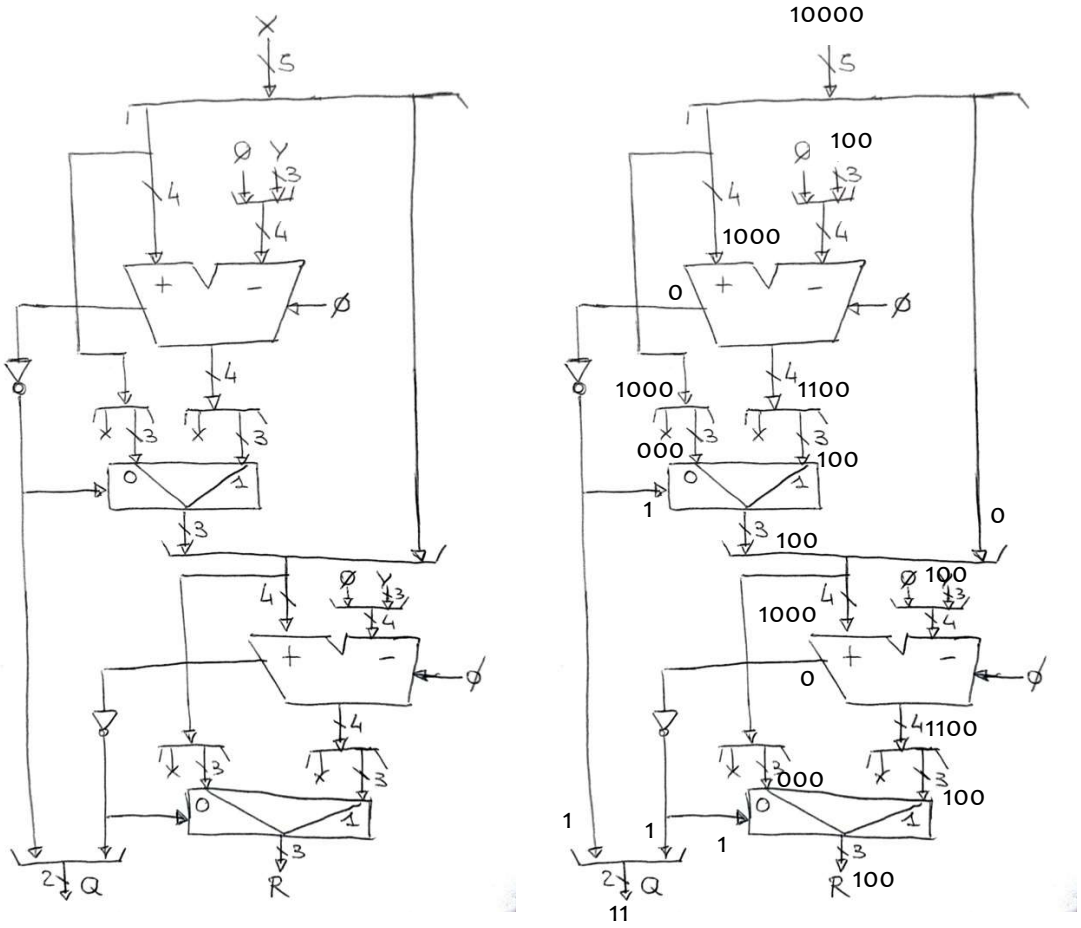
PRIMO APPELLO

DISEGNARE LA STRUTTURA INTERNA DI UN DIVISORE ELEMENTARE PER NATURALI IN BASE 2



Il rilevatore di fattibilità (la parte di rete che genera l'uscita *no_div)* è l'uscita *flag_min* di un *comparatore* tra X e $2Y$ (non disegnata per semplicità).

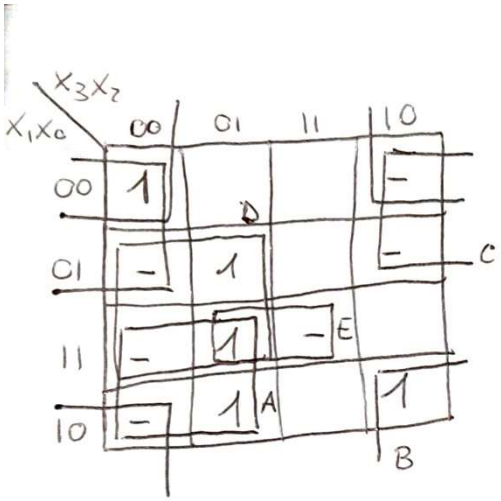
DISEGNARE IL CIRCUITO COMPLETO PER LA DIVISIONE DI UN NUMERO NATURALE A 5 CIFRE PER UN NUMERO A 3 CIFRE IN BASE 2, SCOMPONENDOLO IN BLOCCHI ELEMENTARI. RAPPRESENTARE INGRESSI E USCITE DI OGNI ELEMENTO DELLA RETE QUANDO X=10000, Y=100



SINTESI A COSTO MINIMO E A PORTE NOR

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

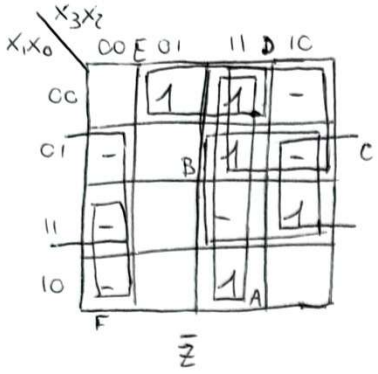
SINTESI A COSTO MINIMO



	X_3	X_2	X_1	X_0	
A	0	–	–	1	essenziale
B	–	0	–	0	essenziale
C	–	0	0	–	ass. eliminabile
D	0	–	–	1	essenziale
E	–	1	1	1	ass. eliminabile

$$z = \overline{X_3} \cdot X_0 + \overline{X_2} \cdot \overline{X_0} + \overline{X_3} \cdot X_0$$

SINTESI A PORTE NOR



	X_3	X_2	X_1	X_0	
A	1	1	–	–	essenziale
B	1	–	–	1	sempl. eliminabile
C	–	0	–	1	sempl. eliminabile
D	1	–	0	–	ass. eliminabile
E	–	1	0	0	essenziale
F	0	0	1	–	ass. eliminabile

$\{A, E\} \xrightarrow{B} \{A, E, B\}$ $\not\rightarrow \{A, E, C\}$

$$\bar{z} = X_3X_2 + X_3X_0 + X_2\overline{X_1}\,\overline{X_0} \Rightarrow z = \bar{\bar{z}} = \overline{X_3X_2 + X_3X_0 + X_2\overline{X_1}\,\overline{X_0}} =$$

$$= \left(\overline{X_3X_2}\right) \cdot \left(\overline{X_3X_0}\right) \cdot \left(\overline{X_2\overline{X_1}\,\overline{X_0}}\right) = \left(\overline{X_3} + \overline{X_2}\right) \cdot \left(\overline{X_3} + \overline{X_0}\right) \cdot \left(\overline{X_2} + X_1 + X_0\right)$$

Per ottenere la sintesi a porte NOR, devo complementare due volte e applicare De Morgan una volta

$$z = \bar{\bar{z}} = \overline{\left(\overline{X_3} + \overline{X_2}\right) \cdot \left(\overline{X_3} + \overline{X_0}\right) \cdot \left(\overline{X_2} + X_1 + X_0\right)} = \overline{\left(\overline{X_3} + \overline{X_2}\right)} + \overline{\left(\overline{X_3} + \overline{X_0}\right)} + \overline{\left(\overline{X_2} + X_1 + X_0\right)}$$

REGISTRO MJR

UTILIZZI NELLE RETI MICROPROGRAMMATE

Il registro MJR ha due principali utilizzi:

1. Gestire **salti a più di due vie.**
2. Gestire le **sottoliste.**

MICROSALTI A PIU' VIE

Nella descrizione del processore come RSS, ci sono due casi in cui serve gestire salti a più di due vie:

1. All'**inizio** della fase di fetch, quando devo **decodificare il formato** dell'istruzione, e quindi saltare ad un certo numero di blocchi di μ-istruzioni differenti a seconda del formato.
2. Alla **fine** di ogni fase di fetch relativa a ciascun formato di istruzione, quando devo decidere quale istruzione, di quelle consentite dal formato, devo andare ad eseguire.

In teoria, **potrei scrivere un μ-salto a N vie utilizzando soltanto μ-salti a due vie**, nel seguente modo:

```
S0: begin /*elaborazioni;*/
          STAR<=(condizione1)?S1:
                (condizione2)?S2:
                […]
                (condizionek-1)?Sk-1:Sk;
end
```

può essere tradotto come

```
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
```

Il problema è che mentre scorro questi stati **il tempo passa**. Se realizzo un processore come RSS, non posso certo pensare di perdere decine di cicli di clock per prelevare e decodificare ogni istru-zione del linguaggio macchina. Né si può prevedere spazio per 50 indirizzi alternativi nella ROM, 103 soltanto perché in **due casi** su chissà quante µ-istruzioni mi servono salti con 50 alternative. Il problema si risolve usando un **registro operativo** apposta, detto **Multiway Jump Register, MJR**. A questo punto, la parte di μ-programma scritta sopra la posso tradurre in

```
S0: begin /*elaborazioni;*/
          MJR<=(condizione1)?S1:
                (condizione2)?S2:
                […]
                (condizionek-1)?Sk-1:Sk;
          STAR<=S0_1;
end
S0_1: begin STAR<=MJR; end
```

Ed in **due stati interni posso gestire qualunque μ-salto a N vie**, qualunque sia il valore di N.

SOTTOLISTE

MJR può anche essere utilizzato per implementare le **sottoliste**. In alcuni casi fa comodo strutturare una descrizione di RSS con sottoliste equivalenti a **sottoprogrammi**, cioè a **pezzi di μ-programma che possono essere raggiunti a partire da stati di partenza diversi**, di modo che il controllo possa:

a. Passare temporaneamente al sottoprogramma.
b. Riprendere dallo stato successivo a quello in cui si era interrotta.

Questa cosa si realizza salvando, all'atto della "chiamata di sottoprogramma", la codifica dello stato di ritorno dentro MJR. Alla fine del sottoprogramma inserirò un salto guidato da MJR, che avrà il ruolo equivalente a quello di una istruzione *return* di un linguaggio di programmazione.

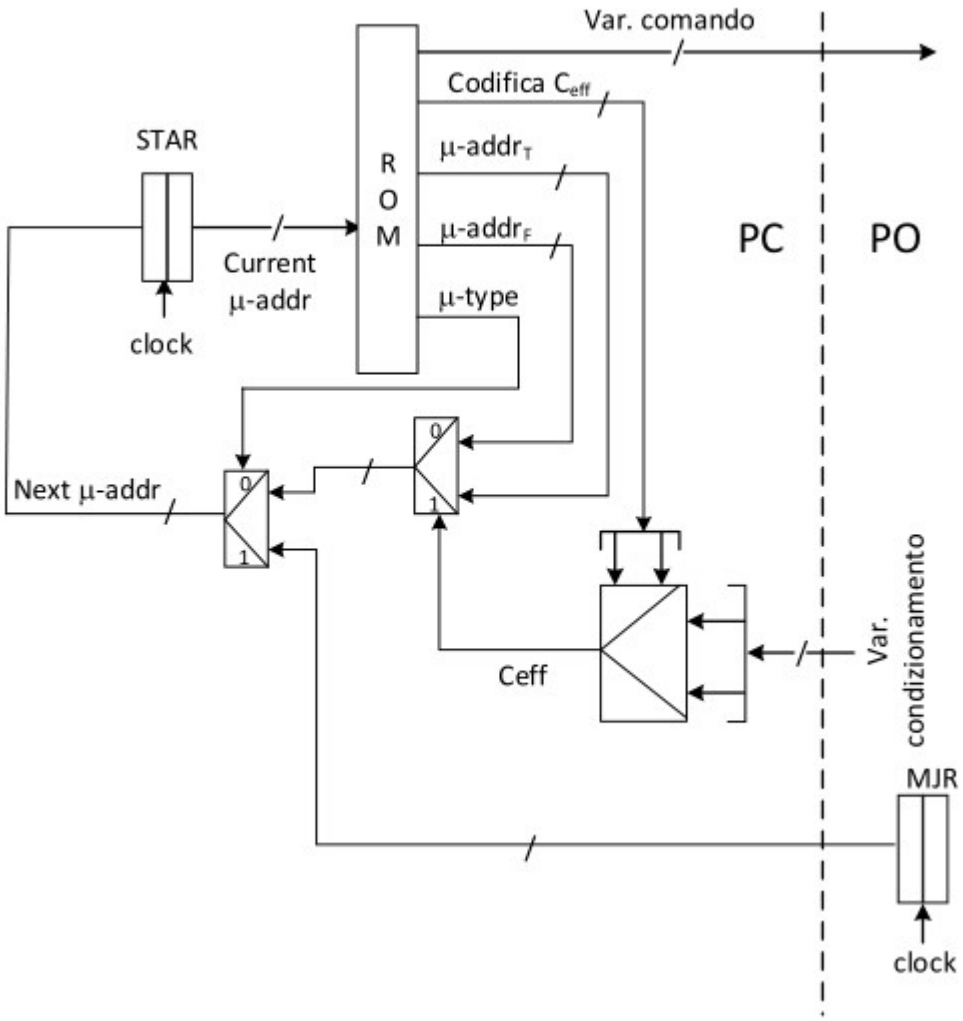
```
S0:   begin  /*elaborazioni*/ MJR<=S1; STAR<=Ssub1; end
S1:   begin  /*elaborazioni*/; end

[…]

Sx:    begin  /*elaborazioni*/ MJR<=Sx+1; STAR<=Ssub1; end
Sx+1:  begin  /*elaborazioni*/; end

[…]

Ssub1:  begin  /*elaborazioni*/ end
[…]
SsubK:  begin  /*elaborazioni*/ STAR<=MJR; end
```

Lista princ.

Sottolista

Il parallelo con i sottoprogrammi si limita **ad un livello di annidamento**. Non posso annidare più sottoprogrammi, se ho un solo registro MJR. Se lo volessi fare, mi servirebbe una ***μ*-pila di registri MJR**.

SCHEMA DI UNA PARTE CONTROLLO DI UNA RSS CHE UTILIZZA MJR



CALL subprog ← PUSH %EIP

JMP subprog

LA CALL PUO' ESSERE SOSTITUITA DALLA COPPIA DI ISTRUZIONI A DESTRA? SE SÌ', PERCHE'? SE NO, PERCHE'?

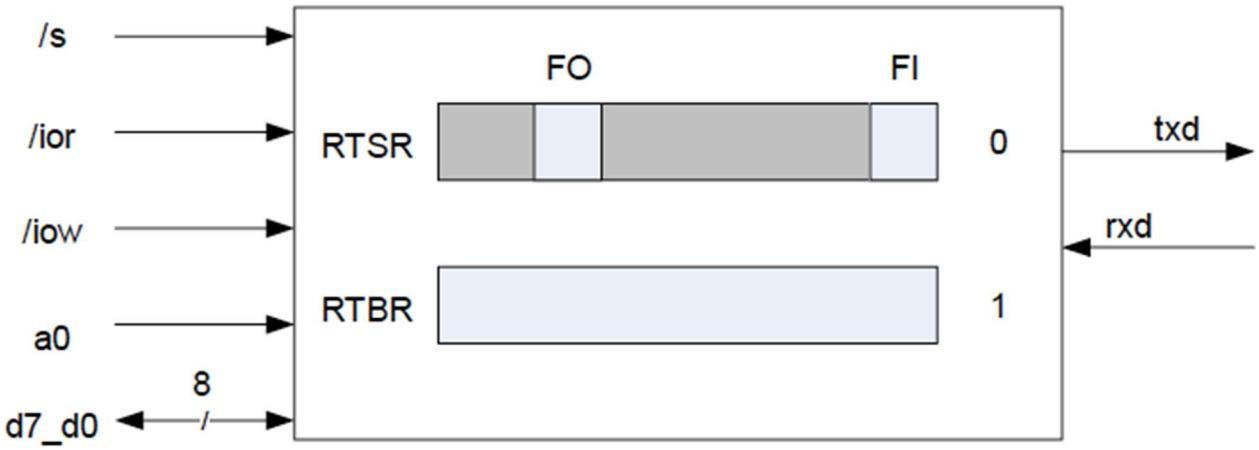
Sì, perché la `CALL` salva nella pila (tramite una `PUSH`) l'indirizzo contenuto in `%EIP` (ovvero l'indirizzo della prossima istruzione da eseguire dopo la `CALL`), e poi prosegue all'indirizzo `subprog`. Per poter proseguire all'indirizzo puntato da `EIP`, serve una `RET`, che al contrario fa una `POP` di `%EIP` e salta all'indirizzo immesso in `%EIP`.

INTERFACCIA SERIALE START/STOP *[DA FINIRE]*

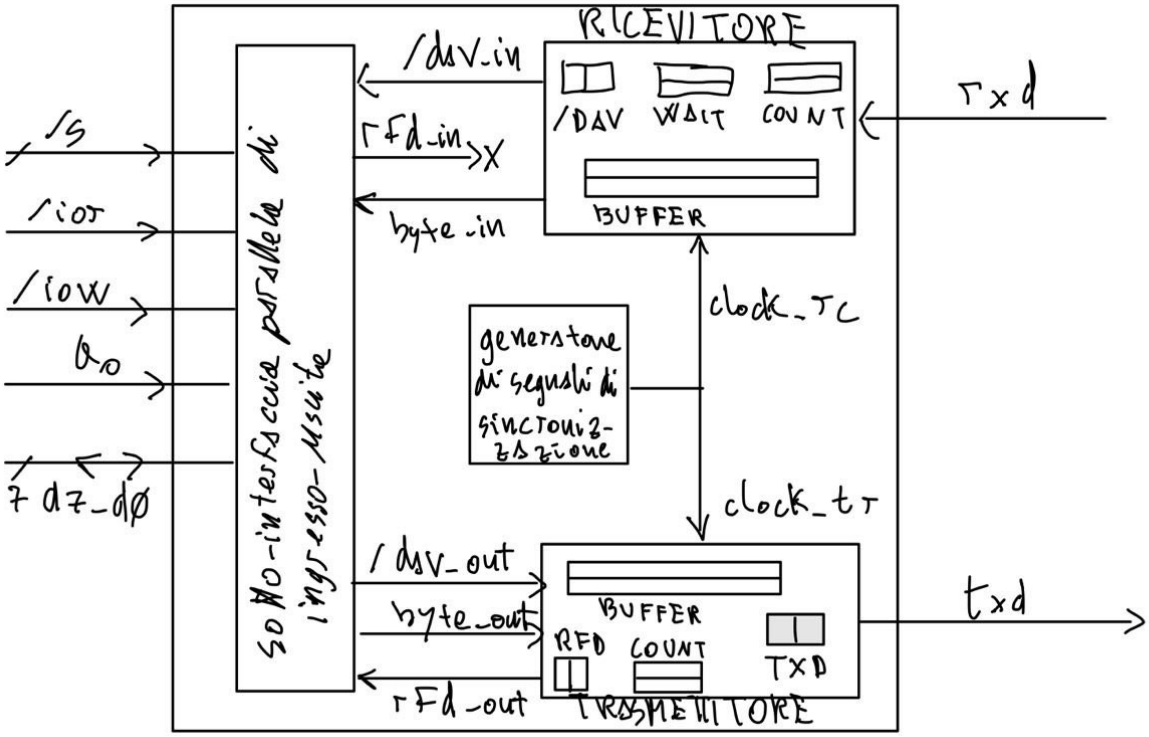
VISIONE FUNZIONALE

Da un punto di vista funzionale, un'interfaccia seriale di ingresso/uscita è simile ad un'interfaccia parallela di ingresso/uscita. Abbiamo infatti:

- 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**.
- Un registro RTBR ad 8 bit che serve per contenere i dati da trasmettere o quelli ricevuti.



STRUTTURA INTERNA A BLOCCHI



C'è una **sottointerfaccia parallela di I/O con handshake**, che colloquia con due **reti sequenziali sincronizzate** dette trasmettitore e ricevitore.

SCRIVERE UN PEZZO DI SOFTWARE CHE LEGGE UN DATO DALL'INTERFACCIA SERIALE, CHE COMPLEMENTA I BIT E GLIELO RIMANDA. SI ASSUMA CHE L'INDIRIZZO BASE DELL'INTERFACCIA SIA 0x1230.

```
check_in:
        IN 0x1230, %AL      # offset 0: RTSR
        AND $0x01, %AL      # controllo FI per l'handshake
        JZ check_in         # se 1, posso fare le operazioni
        IN 0x1231, %AL      # prelevo il bit dal buffer
        NOT %AL             # nego
check_out:
        IN 0x01230, %AH     # controllo FO per l'handshake
        AND $0x20, %AH      # se 1, posso fare le operazioni
        JZ check_out
        OUT %AL, 0x1231     # metto in uscita nel buffer all'offset 1
```

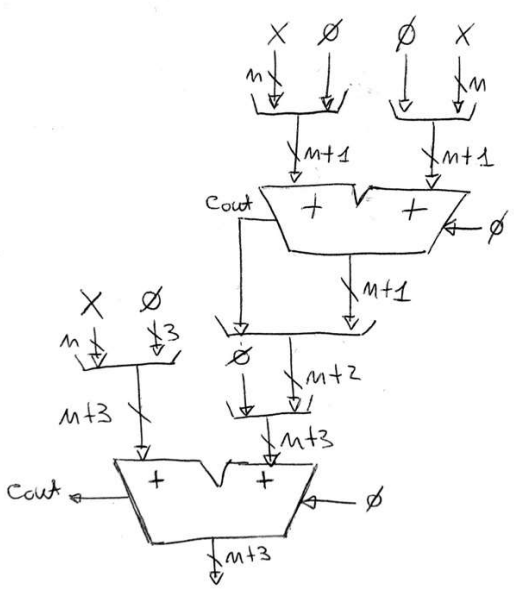
SECONDO APPELLO

DATO UN NATURALE X AD n CIFRE IN BASE 2, REALIZZARE UN CIRCUITO CHE LO MOLTIPLICA PER 11.

Quello che vogliamo ottenere è $Z = X \cdot 11$. Però il numero 11 lo possiamo scomporre nel seguente modo: $11 = 8 + 2 + 1$, e quindi possiamo riscrivere la moltiplicazione come

$$Z = X \cdot (8 + 2 + 1) = X \cdot (2^3 + 2 + 1) = X \cdot 2^3 + X \cdot 2 + X$$

Ci basterà solo sommare tra di loro delle moltiplicazioni per potenze della base, che si fanno con degli shift a sinistra.

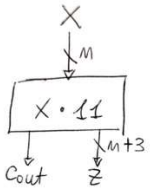


<u>USANDO IL CIRCUITO PRECEDENTE, REALIZZARE UN CIRCUITO CHE MOLTIPLICA X PER 120</u>

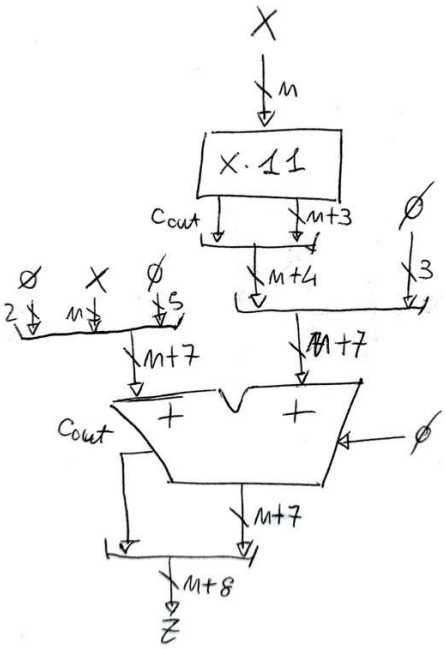
Possiamo vedere la moltiplicazione come

$$Z = X \cdot 120 = X \cdot (2^3 \cdot 15) = X \cdot 2^3 \cdot (8 + 4 + 2 + 1) = X \cdot 2^3 \cdot 2^2 + X \cdot 2^3 \cdot (8 + 2 + 1)$$
$$= X \cdot 2^5 + (X \cdot 11) \cdot 2^3$$

Indicando con il seguente schema il circuito precedente:



Otteniamo:



DESCRIVERE UNA RETE DI MOORE CON UN INGRESSO ED UN'USCITA CHE RICONOSCE SEQUENZE DI STATI DI INGRESSO (NON INTERLACCIATE – L'ULTIMO 1 DI UNA SEQUENZA NON PUO' ESSERE ANCHE IL PRIMO 1 DELLA SEQUENZA SUCCESSIVA) 1, 0, 0, 1, 1

```
module SequenzaMoore (
    numero, z,
    clock, reset_
);
input clock, reset_, numero;

output z; assign z = (STAR == S5)? 1 : 0;

reg [2:0] STAR; localparam S0 = 0, S1 = 1, S2 = 2, S3 = 3, S4 = 4, S5 = 5;

always @(reset_ == 0)#1 begin
    STAR <= S0; end
always @(posedge clock) if(reset_ == 1)#3 begin
    casex (STAR)
    S0: STAR <= (numero == 1)? S1 : S0;
    S1: STAR <= (numero == 0)? S2 : S0;
    S2: STAR <= (numero == 0)? S3 : S0;
    S3: STAR <= (numero == 1)? S4 : S0;
    S4: STAR <= (numero == 1)? S5 : S0;
    S5: STAR <= (numero == 1)? S1 : S0;
    endcase
end
endmodule
```

Sintetizzare, disegnando la porzione di circuito, la seguente parte di descrizione di una rete sequenziale sincronizzata:

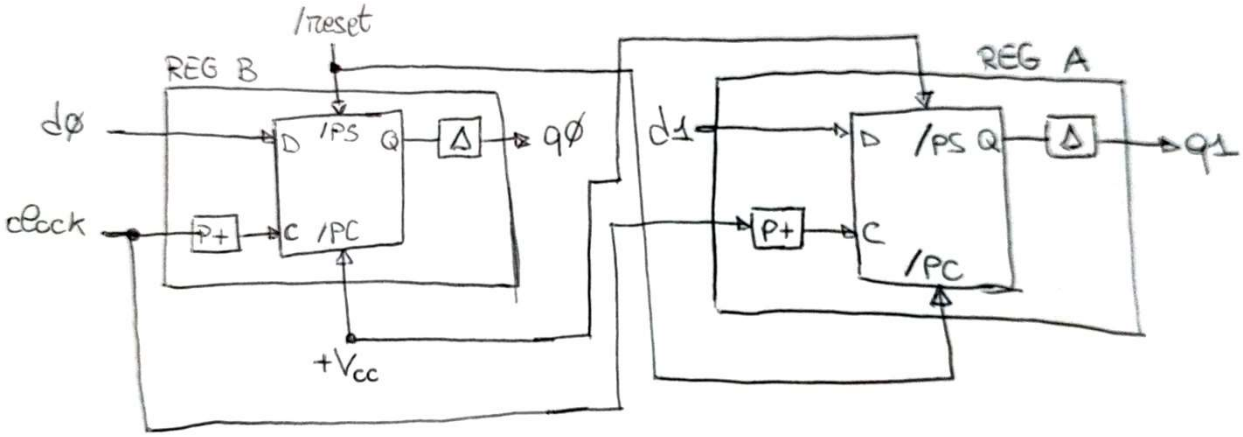
...

reg A, B;

...

always @(reset_ == 0) begin A <= 0; B <= 1; end

always @(posedge clock) if (reset_ == 1) #3 ...



FORMA CANONICA DI UNA FUNZIONE BOOLEANA AD N VARIABILI DI INGRESSO. DESCRIVERE COSA È E COME SI TROVA.

Stiamo parlando della **forma canonica SP**, ed il procedimento per trovarla è il seguente.

Come punto di partenza, consideriamo il seguente risultato (dovuto a Shannon):

"è sempre possibile scrivere **qualunque** legge F di una rete combinatoria come **somma di prodotti degli ingressi (diretti o negati)**".

Data una legge $z = f(x_{N-1}, \dots, x_0)$, possiamo scrivere l'**espansione di Shannon** come segue:

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

$$+ f(0,0,\dots,1) \cdot \overline{x_{N-1}} \cdot \overline{x_{N-2}} \cdot \dots \cdot \overline{x_1} \cdot x_0$$

$$\dots$$

$$+ f(1,1,\dots,0) \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot \overline{x_0}$$

$$+ f(1,1,\dots,1) \cdot x_{N-1} \cdot x_{N-2} \cdot \dots \cdot x_1 \cdot x_0$$

A partire da questa espansione, possiamo ottenere la **forma canonica SP** (SP perché somma di prodotti, canonica perché ogni prodotto ha come fattori *tutti gli ingressi* diretti o negati) applicando le seguenti proprietà:

- $0 \cdot \alpha = 0$
- $0 + \alpha = \alpha$
- $1 \cdot \alpha = \alpha$

E osservando che, per le precedenti tre proprietà:

3. Se $f(x_{N-1}, \dots, x_0) = 0$, allora tutto il termine corrispondente sulla riga vale 0.
4. Se $f(x_{N-1}, \dots, x_0) = 1$, allora posso togliere uno dei fattori dal prodotto.

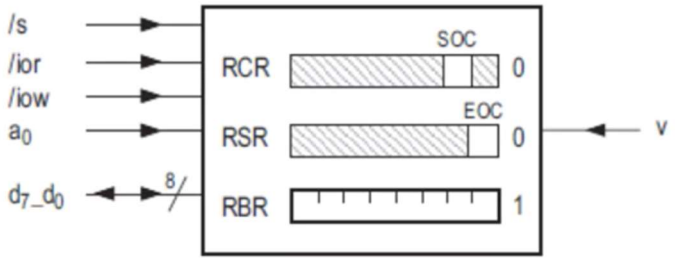
La forma canonica SP prende anche il nome di **lista dei mintermini.** Un mintermine è un prodotto di *tutte* le variabili di ingresso dirette o negate, che compare in una forma canonica SP e riconosce *uno stato di ingresso.*

INTERFACCIA DI CONVERSIONE A/D *[DA FINIRE]*

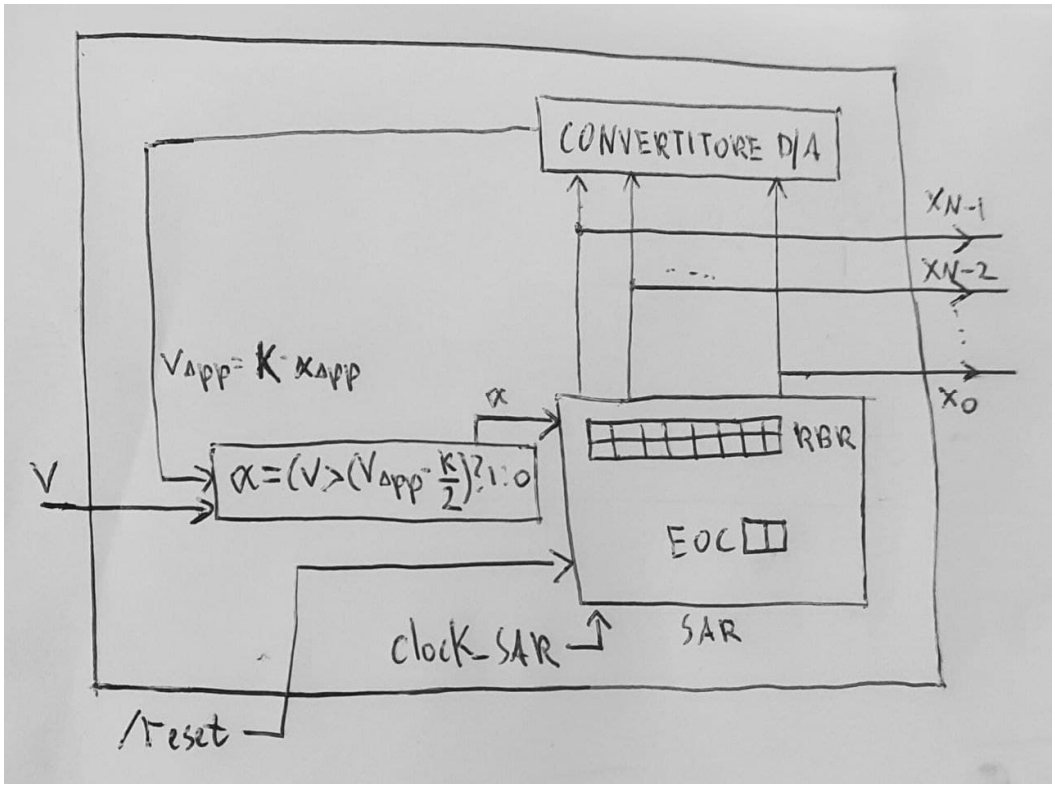
VISIONE FUNZIONALE CON I REGISTRI DELL'INTERFACCIA

Dal punto di vista **funzionale**, sarà un'interfaccia di ingresso/uscita: infatti, la tensione convertita in numero è un ingresso per il processore, così come il segnale di `eoc`, ma il processore deve poter **scrivere** per poter iniziare una nuova conversione. Ci vorranno quindi **due registri:**

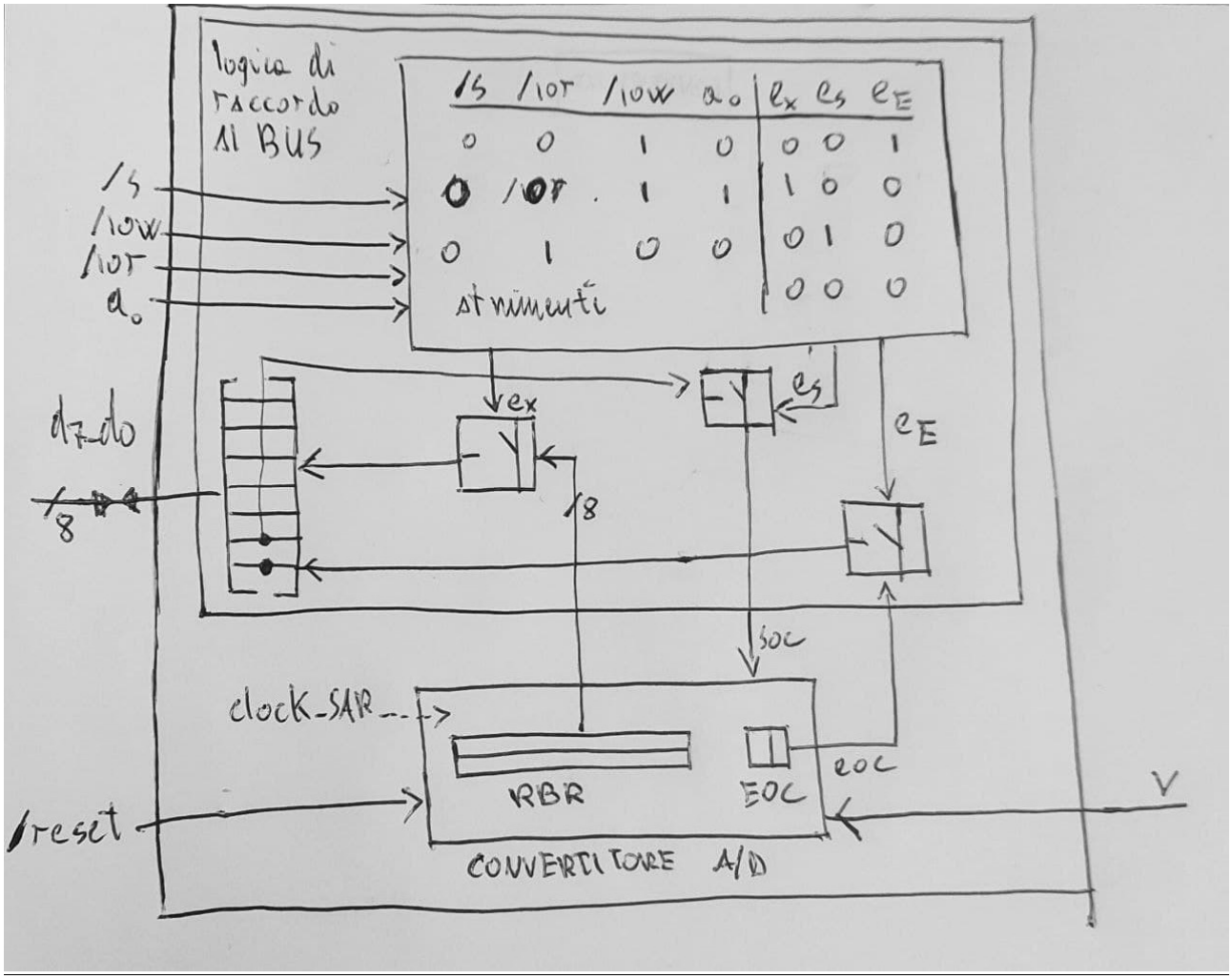
1. Un **Receive Status and Control Register** (`RSCR`), a 8 bit, con due bit significativi: `SOC` (bit 1), che può essere scritto, ed `EOC` (bit 0), che può essere letto.
2. Un **Receive Buffer Register** (RBR), a 8 bit, che (quando `EOC` vale 1) contiene il byte che converte l'ultima tensione di ingresso vista, secondo la legge del convertitore (unipolare o bipolare).



STRUTTURA INTERNA DEL CONVERTITORE



STRUTTURA INTERNA DELL'INTERFACCIA



GESTIONE SOFTWARE DELL'INTERFACCIA STESSA

In Assembler:

```
            MOV $0x01, %AL
            OUT %AL, RCR_offset         # SOC=1

test1:      IN RSR_offset, %AL
            AND $0x01, %AL
            JNZ test1                   # attendi EOC=0
            MOV $0x00, %AL
            OUT %AL, RCR_offset         # SOC=0

test2:      IN RSR_offset, %AL
            AND $0x01, %AL
            JZ test2                    # attendi EOC=1
            IN RBR_offset, %AL          # prelievo dato convertito
            RET
```

In C++:

```
byte acquisizione( ) {
      #define RCR_offset ...
      #define RSR_offset RCR_offset
      #define RBR_offset ...
      byte tmp;

      //Attiva la conversione immettendo 1 in SOC
      outport(RCR_offset,0x02);

      //Attende che il contenuto di EOC vada a 0 e quindi immette 0 in
      SOC
      do {tmp=inport(RSR_offset)&0x01;} while (tmp!=0x00);
      outport(RCR_offset,0x00);

      //Attende che il contenuto di EOC vada a 1
      do {tmp=inport(RSR_offset)&0x01;} while (tmp==0x00);

      //Ritorna il risultato della conversione
      return inport(RBR_offset);
}
```

Visto che il convertitore è **molto veloce ad iniziare la conversione**, possiamo evitare di attendere che EOC vada a zero, e togliere le due parti riquadrate. **Attenzione** a non confondere questa cosa con il fatto di omettere di testare eoc in un esercizio di descrizione. Se ho un esercizio con un convertitore A/D, e devo descrivere una rete che si interfaccia con esso per ottenere dei dati, **non posso certo scrivere:**

```
S1: begin SOC<=1; STAR<=S2; end
S2: begin SOC<=0; … end
```

Ma devo scrivere qualcosa di equivalente a

```
S1: begin SOC<=1; STAR<=S2; end
S2: begin SOC<=eoc; STAR<=(eoc==1)?S2:S3; end
```

RELAZIONE TRA LA TENSIONE IN INGRESSO V DEL CONVERTITORE A/D INTERNO ALL'INTERFACCIA E IL CAMPIONE DI USCITA x PRODOTTO DAL CONVERTITORE INDICANDO PER PUNTI LE CAUSE DEGLI ERRORI DI CONVERSIONE

La tensione è idealmente pari a v = K * X, dove x è il valore con il quale la tensione sarà convertita come valore discreto. Diamo alcune definizioni: K è la **costante di proporzionalità** ed è pari a $K = FSR / 2^N$, dove N è il numero di bit con il quale esprimiamo X, e FSR [Full Scale Range] è una scala di valori a cui la tensione appartiene. La scelta dell'intervallo è fondamentale, difatti se ipotizziamo $v \in [0, FSR]$ vorrà dire che stiamo effettuando una **conversione unipolare** [$X \in [0, 2^N - 1]$, quindi X naturale], altrimenti $v \in [-FSR/2, +FSR/2]$ che implica una codifica bipolare [$X \in [2^{N-1}, 2^{N-1} - 1]$, quindi X intero in traslazione]. Dobbiamo però rinunciare all'ipotesi di una conversione ideale, quindi accontentarci che risulti $| v - K*X | <= err$, con err detto **errore di conversione**. Quest'ultimo è dovuto da due tipologie di errori: errore di **non linearità**, dovuto alla non idealità degli elementi circuitali in gioco, ed **errore di quantizzazione**, dovuto alla conversione di una grandezza continua in una discreta. Il primo è presente nel convertitore D/A, e quindi anche nell' A/D, mentre il secondo è tipico dei convertitori A/D, e possono essere quantificati: il primo deve essere minore di K/2, e così anche il secondo. Per il primo vale questo ragionamento: se avessimo un errore maggiore, vorrebbe dire che numeri consecutivi in una scala di conversione avrebbero intervalli parzialmente sovrapposti, una condizione che va comunque evitata perché perderemmo la monotonia. Ponendo invece l'attenzione sul secondo, notiamo come dipenda

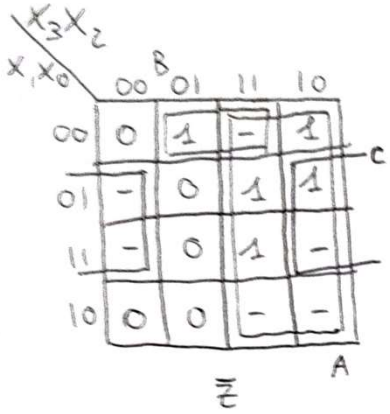
dalla costante K, e sia pari a K/2: perché? Dividendo FSR in 2^N intervalli larghi K e convertendo tutto un intervallo con lo stesso numero, avremo una conversione esatta al centro dell'intervallo, e una errata di |K/2| agli estremi, da cui l'errore. Riassumendo, avremo un errore massimo pari a K/2 nel convertitore D/A, e un errore massimo pari a K/2 + K/2 = K

SESSIONE INVERNALE – A.A. 2021-2022

PRIMO APPELLO

SINTESI A COSTO MINIMO IN FORMA PS DELLA LEGGE z DESCRITTA DALLA SEGUENTE MAPPA DI KARNAUGH

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



	X_3	X_2	X_1	X_0	
A	1	-	-	-	essenziale
B	-	1	0	0	essenziale
C	-	0	-	1	assolutamente eliminabile

$$\overline{z} = X_3 + X_2 \cdot \overline{X}_1 \cdot \overline{X}_0$$

$$z = \overline{\overline{z}} = \overline{X_3 + X_2 \cdot \overline{X}_1 \cdot \overline{X}_0} = \overline{X}_3 \cdot \left(\overline{X_2 \cdot \overline{X}_1 \cdot \overline{X}_0}\right) = \overline{X}_3 \cdot \left(\overline{X}_2 + X_1 + X_0\right)$$

DIMOSTRARE CHE $\forall k \geq 0, \left|10^k\right|_3 = 1$

Noto immediatamente che 10 = 9 + 1, e quindi $10^k = (9+1)^k$

Posso immaginare $(9+1)^k$ come una moltiplicazione a k termini, in modo da trattare singolarmente ognuno di essi per le proprietà dei moduli, infatti $|x * y|_a = ||x|_a * |y|_a|_a$; ricordo la regola $|x + y|_a = ||x|_a + |y|_a|_a$, quindi con semplici passaggi [non riportati per semplicità] dimostro che il singolo termine è pari a $|1|_3 = 1$. Avrò dunque un prodotto di K termini pari a 1: applicando dunque l'operazione di modulo troverò proprio 1, da cui la tesi.

UN FLIP FLOP T (*TOGGLE*) È UNA RSS CON INGRESSO T ED UN'USCITA Q CHE ALL'ARRIVO DEL CLOCK SI COMPORTA COME SEGUE:

SE T=0, CONSERVA

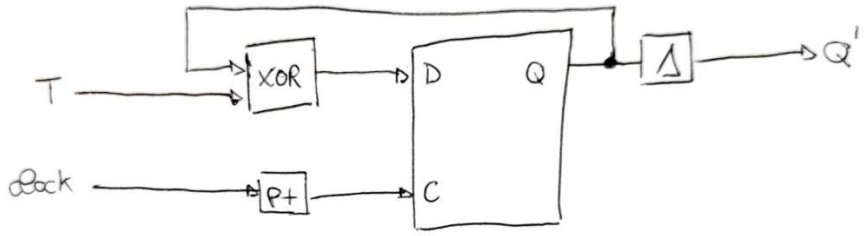
SE T=1, COMMUTA

SINTETIZZARE UN FFT PARTENDO DA UN FF D-POSITIVE EDGE TRIGGERED

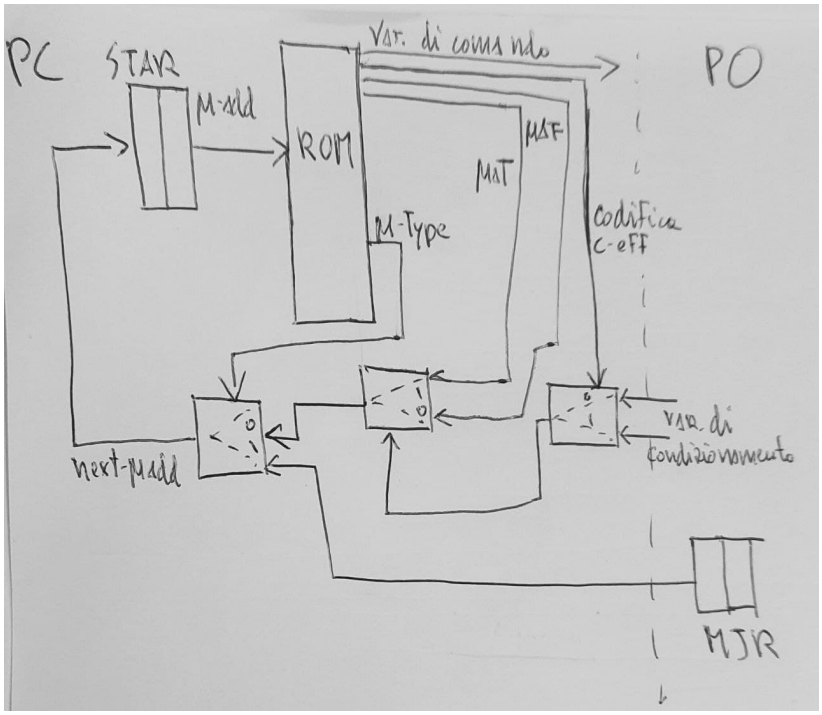
La tabella di applicazione del FF T è la seguente:

T	Q	Q'
0	0	0
0	1	1
1	0	1
1	1	0

Da cui segue che $Q' = T \oplus Q$.



SCHEMA DI UNA PARTE CONTROLLO DI UNA RETE CHE UTILIZZA MJR

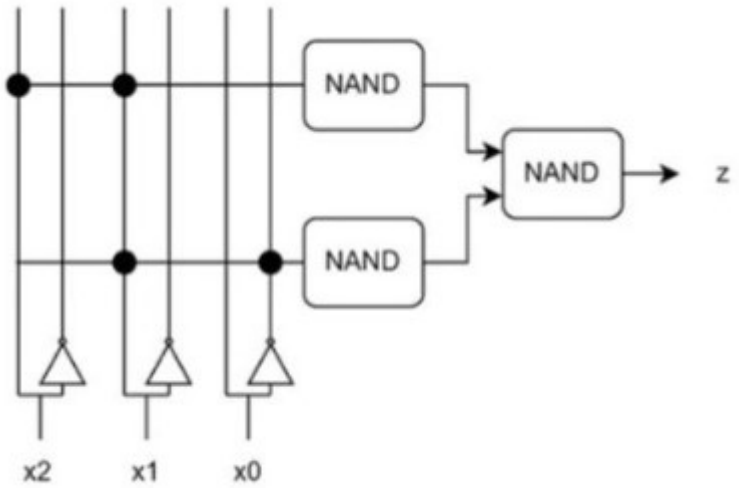


SI SUPPONGA DI ESTENDERE L'ASSEMBLER DEL PROCESSORE VISTO A LEZIONE AMMETTENDO UN'ISTRUZIONE DEL TIPO ADD $numero, indirizzo DOVE ENTRAMBI GLI OPERANDI OCCUPANO UN BYTE. SUPPONENDO CHE ALLA FINE DELLA FASE DI FETCH IL REGISTRO SOURCE CONTENGA $numero E DEST_ADDR indirizzo SCRIVERE LA FASE DI ESECUZIONE DELL'ISTRUZIONE

```
esec_1: begin
      A23_A0<=DEST_ADDR; STAR<=readB; MJR<=esec_2;
end
esec_2: begin
      A23_A0<=DEST_ADDR; APP0<=alu_result(OPCODE, SOURCE, APP0);
      F<={F[7:4], alu_flag(OPCODE, SOURCE, APP0)}; STAR<=writeB;
      MJR<=esec_3;
end
esec_3: begin
      STAR<=fetch0;
End
```

SECONDO APPELLO

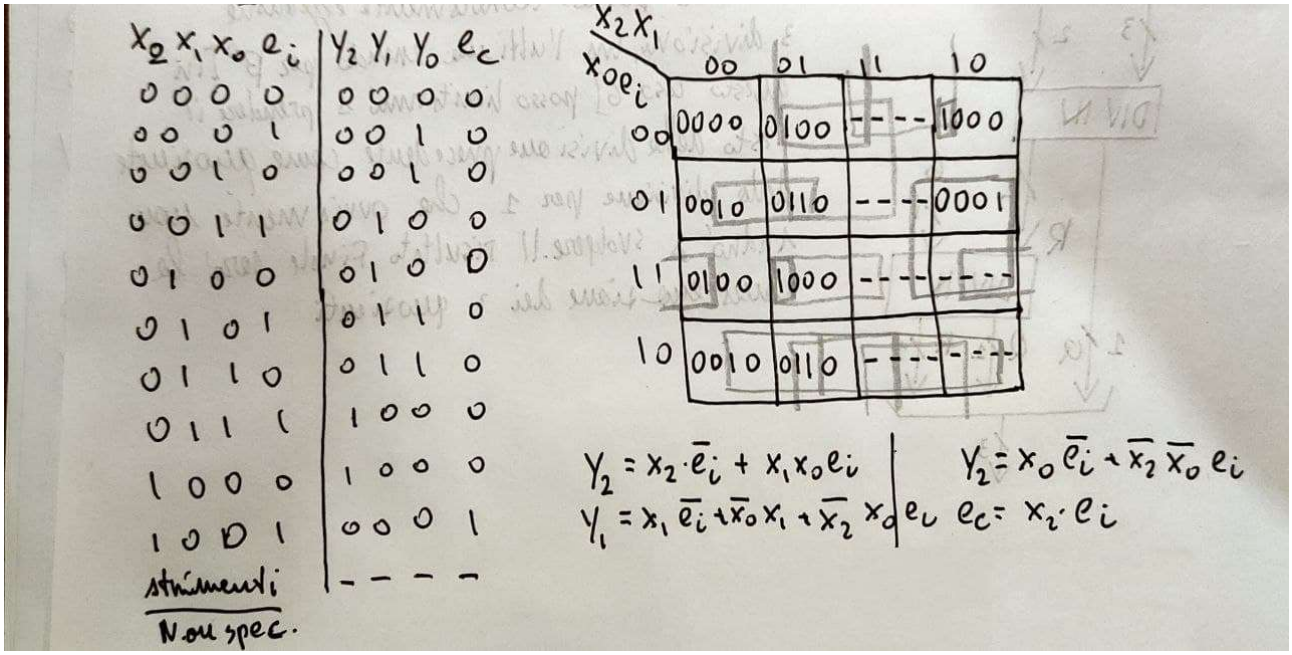
SINTETIZZARE A PORTE NOR LA RETE COMBINATORIA LA CUI SINTESI È IN FIGURA



SINTETIZZARE IL CIRCUITO CHE CONVERTE UN NUMERO NATURALE A 2 CIFRE IN BASE 10 IN UN NUMERO IN BASE 6 [DA FARE]

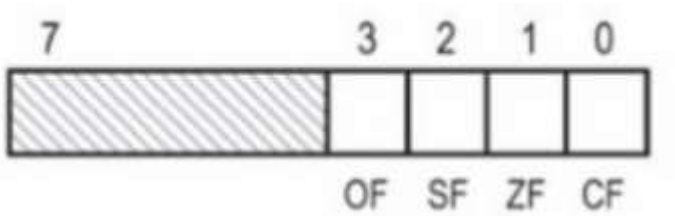
DESCRIVERE [IN MODO FORMALE, E.G., USANDO SCHEMI CIRCUITALI E TABELLE DI VERITÁ, OPPURE IN VERILOG] UN CONTATORE UP AD UNA CIFRA IN BASE 5

Un Contatore è una rete che permette di rallentare il clock, in base alla base in cui si sta lavorando. In generale, il clock verrà reso di periodo x*T, con x cifra più significativa del contatore. Può essere implementato come RSS con al suo interno delle reti combinatorie che, al clock, verificano gli ingressi e agiscono di conseguenza. Dato che la maggior parte della complessità della rete si trova nella RC, si procede alla sintesi della stessa, esaurendo dunque la descrizione della rete che si limita agli ingressi [4, per la cifra base 5 e 1 per verificare l'incremento] e alle uscite [4, 3 per la codifica, che torneranno in ingresso alla rete stessa, e 1 per verificare che la base è stata contata]



PROGETTARE UNO SPAZIO DI MEMORIA RAM DI 4k x 8bit UTILIZZANDO CHIP DI RAM DA 2k x 4bit [DA FARE]

SI SUPPONGA DI ESTENDERE L'ASSEMBLY DEL PROCESSORE STUDIATO A LEZIONE CON LA SEGUENTE ISTRUZIONE "LOOPE *indirizzo*", IL CUI EFFETTO È 1)DECREMENTARE AL (SENZA MODIFICARE I FLAG] E, 2) SE AL È DIVERSO DA 0 E L'ULTIMA ISTRUZIONE CHE SETTAVA I FLAG HA SETTATO LA CONDIZIONE DI UGUALIANZA, SALTARE A *indirizzo.* SI SCRIVA LA FASE DI ESECUZIONE DI QUESTA ISTRUZIONE, SUPPONENDOLA APPARTENENTE AL FORMATO F7 [QUELLO DELLE ISTRUZIONI DI SOLO SALTO]. IL REGISTRO DEI FLAG DEL PROCESSORE STUDIATO A LEZIONE È RIPORTATO SOTTO PER COMODITA[FORNITO DAL PROFESSORE NELLA TRACCIA].




```
ex1: begin
    AL <= AL - 1; STAR <= ex2;
end
ex2: begin
    IP <= (AL != 0 && F[1] == 1)? DEST_ADDR : IP;
    STAR <= fetch0;
end
```