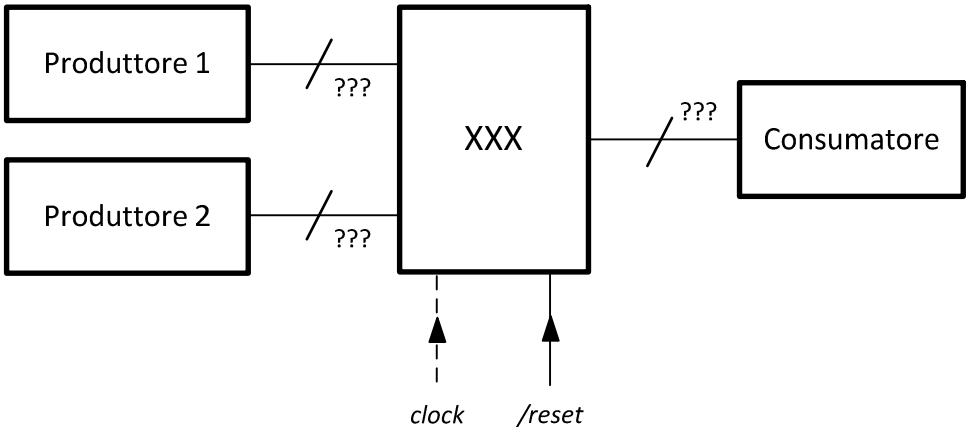
3.3 Esercizio – descrizione e sintesi di RSS complessa

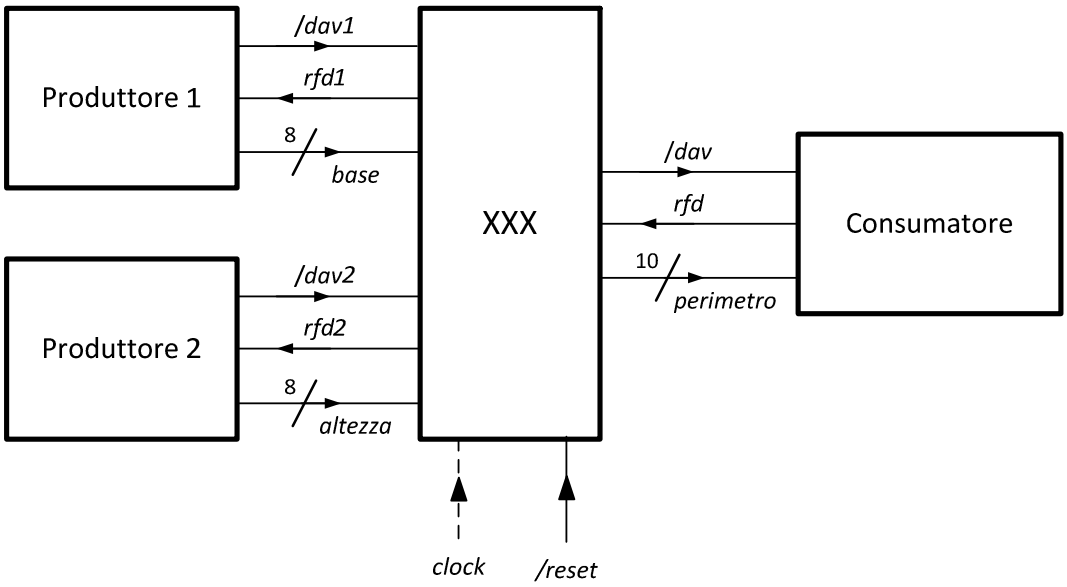


Descrivere la rete **XXX** che si evolve ciclicamente come segue: "preleva un byte dal Produttore 1 e un byte dal Produttore 2, elabora i byte ed invia il risultato della elaborazione al Consumatore." L'elaborazione viene fatta tramite una funzione *mia_rete*(*base*,*altezza*), che interpreta i byte ricevuti da XXX come numeri naturali in base 2 costituenti la base e l'altezza di un rettangolo e restituisce il perimetro del rettangolo. **Specificare in dettaglio** la struttura della rete combinatoria che implementa la funzione *mia_rete* di cui sopra.

NOTE: non è possibile fare nessuna ipotesi sulla velocità dei Produttori e del Consumatore

3.3.1 Descrizione

Le reti Produttore*i* e Consumatore sono asincrone rispetto alla rete XXX, quindi il colloquio deve essere protetto da **handshake** /dav-rfd. La somma dei due lati sta su 9 bit, quindi il perimetro sta su 10 bit. Detto questo, possiamo disegnare i collegamenti nel dettaglio.



Per quanto riguarda la descrizione, osserviamo quanto segue:

1) guardiamo i **registri** che servono, con un dimensionamento di massima.

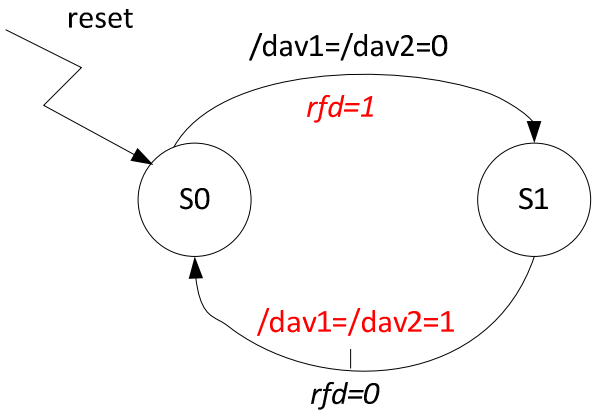
- Ne servono alcuni per sostenere le uscite: RFD1, RFD2, DAV_, a 1 bit ciascuno.
- PERIMETRO a 10 bit
- STAR a un po' di bit (quanti saranno lo vediamo alla fine della descrizione)
- I valori *base* e *altezza* si prendono treamite handshake. Potrei volerli memorizzare da qualche parte (nel qual caso userei due registri BASE e ALTEZZA, a 8 bit ciascuno), ma probabilmente non sarà necessario (posso scrivere il risultato direttamente in PERIMETRO usando una rete combinatoria *mia_rete* che prende in input i valori *base* e *altezza*).

3) condizioni al **reset**:

Posso dare per scontato che tutti gli handshake siano a riposo, quindi che gli input siano inizialmente: */dav*1=1, */dav*2=1, *rfd*=1. **Devo settare gli output** di conseguenza: */dav*=1, *rfd*1=1, *rfd*2=1.

Il valore iniziale di PERIMETRO non è significativo (tanto la sua validità è determinata dal fronte di discesa di /dav). Assumo che lo stato iniziale sia S0.

3) **diagramma a stati** (di massima) della rete:



Disegnare man mano che si descrive

In **S0** devo:

- **tenere *rfd*1, *rfd*2, */dav* a 1;**
- aspettare che **entrambi */dav*1 e */dav*2** siano andati a zero.

Aggiungere gli assegnamenti sotto gli stati nel diagramma

Infatti, non ha senso attendere **prima uno e poi l'altro**, passando da uno stato intermedio. Non ho nessun motivo per credere che uno sia più veloce dell'altro, né uno dei due dati che prelevo con l'handshake dipende dall'altro. Devo comunque aspettare entrambi.

In **S1** i dati *base* e *altezza* sono buoni. Posso quindi calcolare $P = 2(B + A)$ (il conto lo faccio fare a una rete combinatoria), e:

- devo portare a zero RFD1 e RFD2 per far progredire l'handshake con i produttori.
- Devo assegnare PERIMETRO:

```
PERIMETRO<=mia_rete(base, altezza);
```

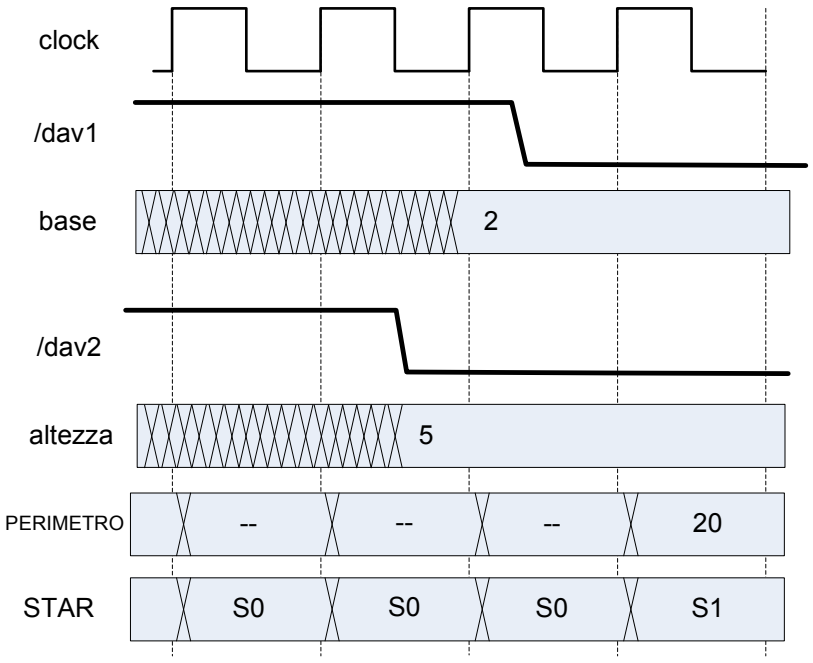
Posso contestualmente **portare DAV_ a 0**, perché il dato di uscita è pronto, in modo da iniziare l'handshake con il consumatore.

Potrei pensare di **ciclare** in S1 finché il consumatore non porta *rfd* a 0. Se le cose stanno così **non lo posso fare**. Infatti, in S1 ho un assegnamento a PERIMETRO che dipende da **dei fili di input**. Ma in S1 ho portato a 0 RFD1 e RFD2, segnalando ai due produttori che quei dati sono **già stati prelevati**. Quando un produttore vede la transizione 1/0 del proprio *rfd*, **non ha più l'obbligo di mantenere il dato in uscita**. Quindi, se ciclo in S1, ad ogni iterazione verrà rinnovato l'assegnamento, ma il valore degli ingressi è garantito essere corretto **soltanto alla prima iterazione**. Dalla seconda iterazione in poi gli input su cui calcolo il perimetro possono essere **non significativi**.

Visto che comunque ho bisogno di attendere che *rfd* del consumatore vada a 0, come risolvo la situazione? O uso un altro stato, oppure (meglio) **porto in S0 l'assegnamento:**

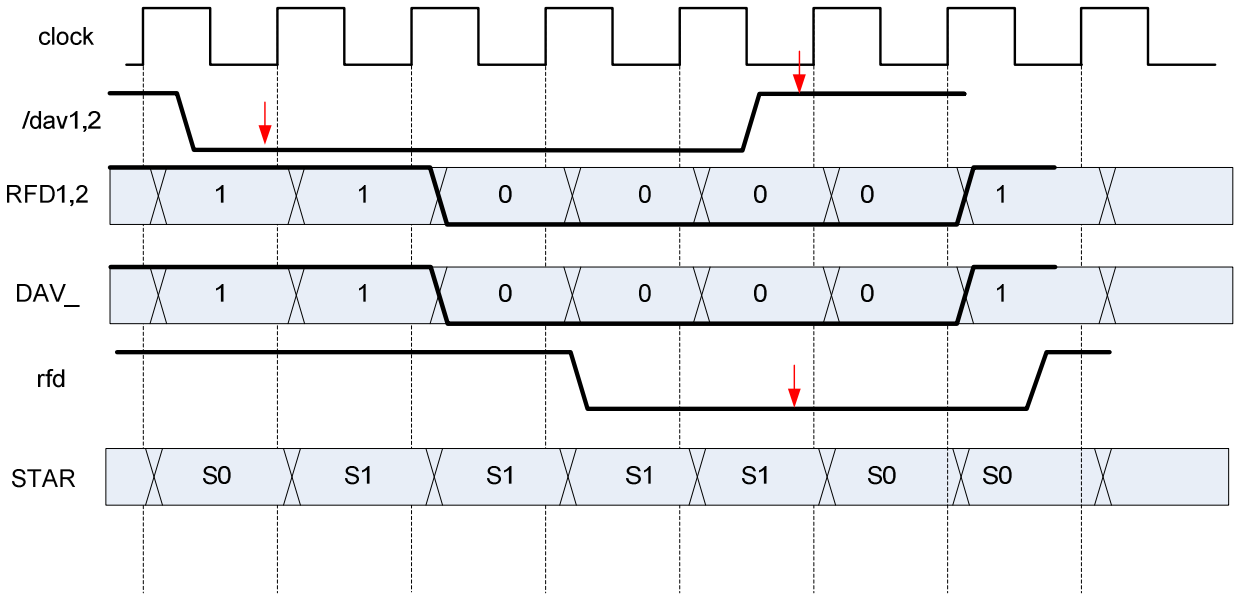
```
PERIMETRO<=mia_rete(base, altezza);
```

In questo modo in S0 ciclo assegnando a PERIMETRO valori a caso (finché almeno uno dei due */dav* vale 1). Però esco da S0 quando **entrambi i /dav** sono a 0, e quindi l'ultimo assegnamento è quello corretto. Il contenuto del registro PERIMETRO balla, ma non è un problema, perché l'output *perimetro* è soggetto all'handshake, e non verrà letto dal consumatore prima che dav_ vada a 0.



A questo punto in S1 posso gestire l'handshake con il consumatore. Tiro giù DAV_**,** attendo che *rfd* sia andato a 0 e vado dove? Non posso saltare indietro in S0, a meno che non mi sia assicurato **anche** che */dav1 /dav2* siano tornati a 1 (cioè che si sia chiuso l'handshake con i due produttori. Se tornassi in S0 senza testare, finirei per rischiare di violare l'handshake.

Guardiamo meglio i tre handshake:



Quindi la condizione per poter tornare in S0 è che:

- *rfd*=0
- */dav1* e */dav2* sono **entrambi a 1** (infatti in S0 metto *rfd1* e *rfd2* ad 1, chiudendo l'handshake)

Però si vede bene che, in questo modo, **non si controlla mai** che ***rfd* sia tornato a 1**. Pertanto, ad un nuovo ciclo di esecuzione, non potrei essere sicuro che quando metto */dav* a 0 in S1 l'handshake con il consumatore si è svolto correttamente.

Si rimedia testando che ***rfd* sia tornato a 1 in S0**. La condizione per uscire da S0 va modificata, aggiungendo che *rfd* deve essere uguale a 1.

La descrizione è quindi la seguente:

```
module XXX (base,dav1_,rfd1,altezza,dav2_,rfd2,perimetro,dav_,rfd,
            clock,reset_);
 input          clock,reset_;
 input          dav1_, dav2_, rfd;
 output         rfd1, rfd2, dav_;
 input   [7:0] base, altezza;
 output  [9:0] perimetro;

 reg            RFD1, RFD2, DAV_; // ne basterebbe uno solo.
 reg     [9:0] PERIMETRO;
 reg STAR;
 parameter S0=0,S1=1;

 assign   rfd1=RFD1;
 assign   rfd2=RFD2;
 assign   dav_=DAV_;
 assign   perimetro=PERIMETRO;
```

```
always @(reset_==0) #1 begin STAR<=S0; RFD1<=1; RFD2<=1; DAV_<=1; end
always @(posedge clock) if (reset_==1) #3
 casex(STAR)
 S0: begin RFD1<=1; RFD2<=1; DAV_<=1; PERIMETRO<=mia_rete(base,altezza);
           STAR<=({dav1_,dav2_,rfd}=='B001)?S1:S0; end

 S1: begin RFD1<=0; RFD2<=0; DAV_<=0;
           STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S1; end
 endcase

 function[9:0] mia_rete;
   input [7:0] base, altezza;
   mia_rete = {({1'B0,base}+{1'B0,altezza}),1'B0};
 endfunction

 endmodule
```

Guardando la descrizione ci si rende subito conto che RFD1, RFD2 e DAV_ sono lo stesso registro, e quindi ne basta uno solo, chiamiamolo HS (handshake). La descrizione ottimizzata è:

```
 […]
 reg           HS;
 […]

 assign   rfd1=HS;
 assign   rfd2=HS;
 assign   dav_=HS;

 […]

 always @(reset_==0) #1 begin STAR=S0; HS<=1; end

 casex(STAR)
 S0: begin HS<=1; PERIMETRO<=mia_rete(base,altezza);
           STAR<=({dav1_,dav2_,rfd}=='B001)?S1:S0; end

 S1: begin HS<=0; STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S1; end
 endcase
```

3.3.2 Sintesi

La sintesi è banale: andiamo per ordine:

Registro operativo HS

```
S0: HS<=1;
S1: HS<=0;
```

Una variabile di comando b0, che vale 1 in S0 e 0 in S1.

```
always @(posedge clock) if (reset_==1) #3
    HS<=b0;
```

Registro operativo PERIMETRO

```
S0: PERIMETRO<=mia_rete(base,altezza);
S1: PERIMETRO<=PERIMETRO;
```

Questo è un registro multifunzionale a due vie. Basta una variabile di comando b0, che vale 1 in S0 e 0 in S1.

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

```
 casex(b0)
  'B1: PERIMETRO<=mia_rete(base,altezza);
  'B0: PERIMETRO<=PERIMETRO;
 endcase
```

Registro di stato STAR

```
 S0: STAR<=({dav1_,dav2_,rfd}=='B001)?S1:S0;
 S1: STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S1;
```

Ci sono due condizioni indipendenti, quindi servono **due variabili di condizionamento** c0 e c1:

```
c0=({dav1_,dav2_,rfd}=='B001)?1:0;
c1=({dav1_,dav2_,rfd}=='B110)?1:0;
```

Abbiamo tutto per poter scrivere la sintesi secondo il paradigma parte operativa / parte controllo.

```
module XXX (base,dav1_,rfd1,altezza,dav2_,rfd2,perimetro,dav_,rfd,
            clock, reset_);

 input         clock,reset_;
 input         dav1_, dav2_, rfd;
 output        rfd1, rfd2, dav_;
 input   [7:0] base, altezza;
 output  [9:0] perimetro;

 wire c1,c0,b0;
 Parte_Operativa PO(base,dav1_,rfd1,altezza,dav2_,rfd2,
                    perimetro,dav_,rfd, c1,c0,b0,clock,reset_);
 Parte_Controllo PC(b0,c1,c0,clock,reset_);
endmodule

//-----------------------------------------------------------------
module Parte_Operativa(base,dav1_,rfd1,altezza,dav2_,rfd2,
                       perimetro,dav_,rfd, c1,c0,b0,clock,reset_);
 input         clock,reset_;
 input         dav1_, dav2_, rfd;
 output        rfd1, rfd2, dav_;
 input   [7:0] base, altezza;
 output  [9:0] perimetro;

 input b0;
 output c1,c0;

 assign c0=({dav1_,dav2_,rfd}=='B001)?1:0;
 assign c1=({dav1_,dav2_,rfd}=='B110)?1:0;

//Registro HS
 always @(reset_==0) #1 HS<=1;
 always @(posedge clock) if (reset_==1) #3 HS<=b0;

//Registro PERIMETRO
 always @(posedge clock) if (reset_==1) #3
  casex(b0)
   'B1: PERIMETRO<=mia_rete(base,altezza);
   'B0: PERIMETRO<=PERIMETRO;
  endcase
endmodule
```

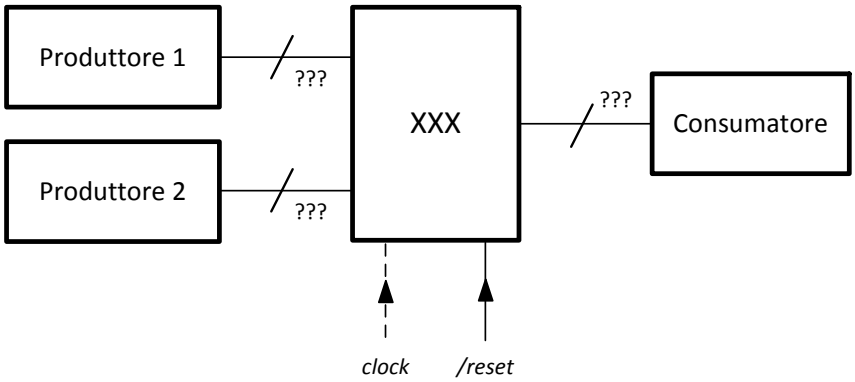
```
module Parte_Controllo(b0,c1,c0,clock,reset_);
 input clock,reset_;
 input c1,c0;
 output b0;
 reg STAR; parameter S0='B0,S1='B1;
 assign b0=(STAR==S0)?'B1:'B0;

 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)?S0:S1;
  endcase
endmodule

//----------------------------------------------------------------------
```

Qualora si voglia vedere la parte controllo come ROM-based (micro-address-based o micro-instruction based), abbiamo:

μ-addr	**μ-code** b_0	C_{eff}	**μ-addr T**	**μ-addr F**
0 (S0)	1	0	1 (S1)	0 (S0)
1 (S1)	0	1	0 (S0)	1 (S1)

Si osservi che la parte controllo è di fatto un FF-JK, con c0=j, c1=k, b0=~q.

3.4 Esercizio – Calcolo del prodotto con algoritmo di somma e shift



Descrivere il circuito **XXX** che si evolve ciclicamente come segue: "preleva un byte dal Produttore 1 e un byte dal Produttore 2, elabora i byte ed invia il risultato della elaborazione al Consumatore." L'elaborazione consiste nell'interpretare i byte ricevuti da XXX come numeri naturali in base 2 costituenti la base e l'altezza di un rettangolo e calcolare l'**area** del rettangolo, *usando l'algoritmo di somma e shift*:

Dati X, C *numeri naturali in base* β *su n cifre,* Y *numero naturale in base* β *su m cifre, l'algoritmo calcola* $P = X \cdot Y + C$ *come segue:*

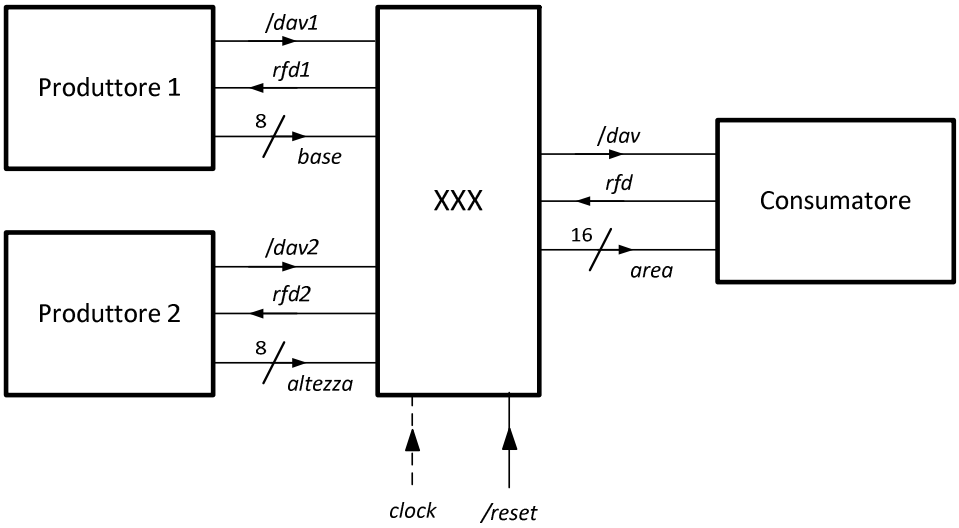
$$P_0 = C \cdot \beta^m$$

$$P_{i+1} = \left\lfloor \frac{y_i \cdot \beta^m \cdot X + P_i}{\beta} \right\rfloor$$

NOTE: non è possibile fare nessuna ipotesi sulla velocità dei Produttori e del Consumatore

3.4.1 Descrizione

I collegamenti della rete sono identici al caso precedente, salvo che adesso l'uscita si chiama *area* ed è su 16 bit.

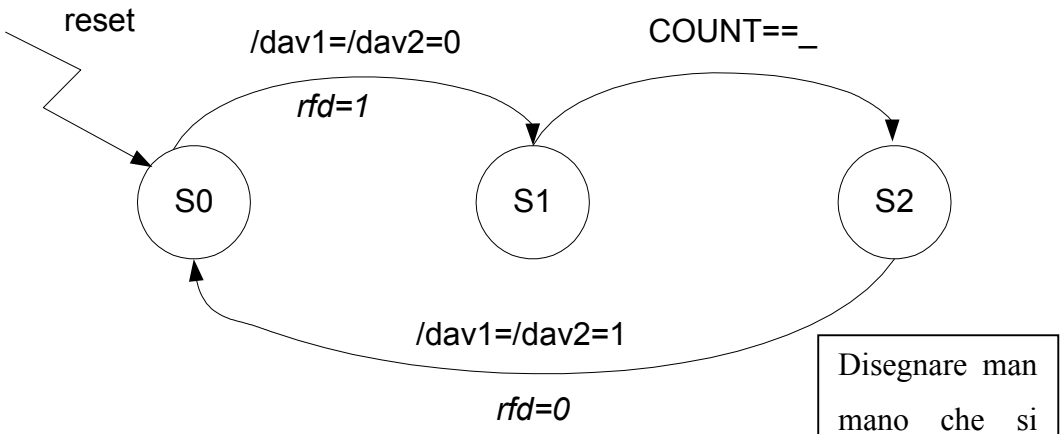


Per far girare l'algoritmo, ho $C = 0$, quindi $P_0 = 0$. Definirò una rete combinatoria `mia_rete` che sintetizza il passo iterativo dell'algoritmo.

1) guardiamo i **registri**, con un dimensionamento di massima.
   - Ne servono alcuni per sostenere le uscite: RFD1, RFD2, DAV_, a 1 bit ciascuno. Non è improbabile che possa compattarli come nel caso precedente.
   - AREA a 16 bit
   - STAR a un po' di bit (quanti saranno lo vediamo alla fine della descrizione)
   - Ne servirà qualcuno **per fare i conti**. Devo calcolare qualcosa attraverso un algoritmo **iterativo**, e quindi userò un registro COUNT come "variabile di conteggio" per tener traccia del numero di iterazioni. A quanti bit? Devo fare 8 iterazioni, quindi ci vorranno 3 o 4 bit.
   - I valori *base* e *altezza* li prendo con un handshake. Converrà memorizzarli da qualche parte. Uso due registri BASE e ALTEZZA, a 8 bit ciascuno.

4) condizioni al **reset**:
   Tutti gli handshake sono a riposo, quindi posso dare per scontato che */dav*1=1, */dav*2=1, *rfd*=1 (input), e **devo fare in modo che** gli output siano consistenti: */dav*=1, *rfd*1=1, *rfd*2=1.

3) **diagramma a stati** (di massima) della rete



Disegnare man mano che si descrive

In **S0** devo:
- campionare *base, altezza*
- tenere *rfd*1, *rfd*2, */dav* a 1
- aspettare che */dav*1 **e** */dav*2 siano andati a zero. Con l'esperienza dell'esercizio precedente, possiamo dire fin d'ora che ci vorrà una terza condizione, cioè *rfd*=1, per gestire la chiusura dell'handshake con il consumatore.

In **S1** comincio il **calcolo iterativo del prodotto**. Facciamo che AREA contiene, ad ogni clock, P_i. Devo quindi:

- inizializzare COUNT e AREA, e lo devo fare in S0, perché in S1 li sto usando. AREA lo inizializzo a zero, e COUNT, **in prima battuta**, lo inizializzo a 8.
- devo portare a zero RFD1 e RFD2 (**e non DAV_,** perché il dato non è ancora stato calcolato)
- devo implementare la formula scritta sopra. Mi serve una **rete combinatoria**, che abbia in ingresso: a) BASE, b) il vecchio valore di AREA, e c) **un bit di altezza**. Il bit che mi serve cambia da un ciclo all'altro. Tutte le volte che questo succede, la tecnica **standard** da usare è la seguente: metto il valore da usare in un registro, utilizzo il bit 0 del registro, e **shifto a destra il contenuto del registro ad ogni clock** (visto che mi servono, nell'ordine, i bit dal meno al più significativo. Se fosse stato il contrario avrei preso il bit 7 e shiftato a sinistra).

  Quindi, in S1, devo decrementare COUNT, shiftare ALTEZZA, e assegnare AREA:

```
AREA<=mia_rete(BASE, ALTEZZA[0], AREA);
dec COUNT;
shr ALTEZZA;
```

  Devo infine **saltare altrove, quando** ho finito le iterazioni del ciclo che dovevo fare. Scrivo la condizione in modo generico, come

```
STAR<=(COUNT==_)?S2:S1;
```

  poi la specifico meglio in fondo.

Vado in uno stato **S2**: il dato è pronto, e devo gestire l'handshake con il consumatore. Tiro giù DAV_**,** attendo che *rfd* sia andato a 0 e vado dove? Posso saltare indietro in S0, se mi assicuro **anche** che */dav1 /dav2* siano tornati a 1.

Ma a questo punto bisogna che, tra S0 e S2, testi anche che **rfd sia tornato a 1**. Lo posso fare soltanto in S0, e quindi la condizione per uscire da S0 va modificata, aggiungendo che rfd deve essere uguale a 1.

Mancano da chiarire la condizione per uscire da S1 e la rete combinatoria. Quando si hanno **cicli di decremento e test** (come in questo caso) la regola è semplice:

se inizializzo a *k* e testo a *j* (<=*k*), il numero di iterazioni è $k - j + 1$**.**

Quindi, se inizializzo COUNT a 8, lo devo testare ad 1 per avere 8 iterazioni. Allora conviene inizializzarlo a 7 e testarlo a 0, così tra l'altro posso dimensionare il registro su 3 bit invece che 4.

Per quanto riguarda la **rete combinatoria *mia_rete***, basta seguire la formula. Al numeratore:

- se $y_i = 0$, ho direttamente P_i
- se $y_i = 1$, ho una somma di due addendi, entrambi a 16 bit. La somma sta quindi su 17 bit.

In uscita, devo buttare il bit meno significativo (divisione per beta). Quindi serve un sommatore a 17 bit, un multiplexer e devo strigare un po' di fili.

In realtà la rete che fa la somma può essere semplificata, perché si vede subito che gli 8 bit più bassi sono $P_i[7:0]$, in quanto sono sommati a zeri. Quindi non c'è bisogno di un sommatore a 17 bit, ma di uno a 9.

La descrizione è la seguente:

```
module XXX (base,dav1_,rfd1,altezza,dav2_,rfd2,area,dav_,rfd,
                clock,reset_);
 input         clock,reset_;
 input         dav1_, dav2_, rfd;
 output        rfd1, rfd2, dav_;
 input   [7:0] base, altezza;
 output [15:0] area;

 reg           RFD1, RFD2, DAV_;
 reg    [15:0] AREA;
 reg     [7:0] BASE, ALTEZZA;
 reg     [3:0] COUNT;           // non posso sapere subito quanti bit
 reg     [1:0] STAR;            // non posso sapere subito quanti bit
 parameter S0='B00,S1='B01,S2='B10;

 assign   rfd1=RFD1;
 assign   rfd2=RFD2;
 assign   dav_=DAV_;
 assign   area=AREA;

always @(reset_==0) #1 begin STAR<=S0; RFD1<=1; RFD2<=1; DAV_<=1; end
always @(posedge clock) if (reset_==1) #3
 casex(STAR)
  S0: begin RFD1<=1; RFD2<=1; DAV_<=1; BASE<=base; ALTEZZA<=altezza;
            AREA<=0; COUNT<=7; STAR<=({dav1_,dav2_,rfd}=='B001)?S1:S0;
      end

  S1: begin RFD1<=0; RFD2<=0; COUNT<=COUNT-1;
            ALTEZZA<={1'B0, ALTEZZA[7:1]};
            AREA<=mia_rete(BASE, ALTEZZA[0], AREA);
            STAR<=(COUNT==0)?S2:S1; end

  S2: begin DAV_<=0; STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S2; end
 endcase

 function[15:0] mia_rete;
   input [7:0] x;
   input y_i;
   input [15:0] p_i;
   casex(y_i)
      'B0: mia_rete={1'B0, p_i[15:1]};
      'B1: mia_rete={1'B0, p_i[15:1]}+{1'B0, x, 7'B0000000};
   endcase
 endfunction

endmodule
```

Un registro non è un vettore. Non lo posso indicizzare con una *variabile* in Verilog. **Non** posso scrivere

```
ALTEZZA[COUNT]
```

Possibili ottimizzazioni rispetto alla descrizione di cui sopra:

- RFD1, RFD2 possono andare a 0 direttamente in S2. In tal caso, RFD1, RFD2 e DAV_ contengono sempre lo stesso bit in ogni stato, quindi posso usare un solo registro HS come

nel precedente esercizio. Peraltro, se faccio così, **non ho più bisogno del registro BASE**, e posso mandare in input a `mia_rete` direttamente gli ingressi *base*, che in S1 sono tenuti stabili dal produttore1 (dato che */dav1*=0, *rfd1*=1). L'ingresso *altezza* va invece salvato in un registro in ogni caso, perché devo usarne un bit alla volta.

- Volendo risparmiare qualcosa, posso inserire l'inizializzazione di COUNT direttamente nella fase di reset. Infatti, quando si esce dal ciclo in S1 COUNT vale 7, cioè il valore che gli sarebbe stato assegnato al successivo passaggio in S0. Pertanto non c'è bisogno di inizializzarlo in S0 esplicitamente.

La descrizione con le ottimizzazioni di cui sopra è la seguente:

```
[…]
 reg           HS;
 reg    [15:0] AREA;
 reg     [7:0] ALTEZZA;
 reg     [2:0] COUNT;
 reg     [1:0] STAR;

 assign   rfd1=HS;
 assign   rfd2=HS;
 assign   dav_=HS;
[…]
always @(reset_==0) #1 begin STAR<=S0; HS<=1; COUNT<=7; end
always @(posedge clock) if (reset_==1) #3
 casex(STAR)
  S0: begin HS<=1; ALTEZZA<=altezza;
            AREA<=0; STAR<=({dav1_,dav2_,rfd}=='B001)?S1:S0; end

  S1: begin COUNT<=COUNT-1;
            ALTEZZA<={1'B0, ALTEZZA[7:1]};
            AREA<=mia_rete(base, ALTEZZA[0], AREA);
            STAR<=(COUNT==0)?S2:S1; end

  S2: begin HS<=0;
            STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S2; end
 endcase
endmodule
```

3.4.2 Sintesi

Partiamo dai registri operativi:

Registro operativo HS

```
S0: HS<=1;
S1: HS<=HS;
S2: HS<=0;
```

Registro operativo ALTEZZA

```
S0: ALTEZZA<=altezza;
S1: ALTEZZA<={1'B0, ALTEZZA[7:1]};
S2: ALTEZZA<=ALTEZZA;
```

Registro operativo AREA

```
S0: AREA<=0;
S1: AREA<=mia_rete(base, ALTEZZA[0], AREA);
S2: AREA<=AREA;
```

Registro operativo COUNT

```
S0,S2: COUNT<=COUNT;
S1: COUNT<=COUNT-1;
```

Tre dei registri operativi sono registri multifunzionali a tre funzioni con un multiplexer a 3 vie, comandato da due variabili di comando. Pertanto, sono necessarie due variabili di comando, b0, b1, che posso assegnare in questo modo:

```
S0: b1b0=00;
S1: b1b0=01;
S2: b1b0=10;
```

In tal modo, posso usare solo b0 per comandare il multiplexer di COUNT, che è a due vie.

Registro di stato STAR

```
  S0: STAR<=({dav1_,dav2_,rfd}=='B001)?S1:S0; end
  S1: STAR<=(COUNT==0)?S2:S1; end
  S2: STAR<=({dav1_,dav2_,rfd}=='B110)?S0:S2; end
```

Ci sono tre condizioni indipendenti, quindi servono tre variabili di condizionamento:

```
c0=({dav1_,dav2_,rfd}=='B001)?1:0
c1=(COUNT==0)?1:0
c2=({dav1_,dav2_,rfd}=='B110)?1:0
```

Quindi avremo per la parte operativa (saltando un po' di sintassi):

```
[…]
 input b1,b0;
 output c2,c1,c0;

 assign c0=({dav1_,dav2_,rfd}=='B001)?1:0;
 assign c1=(COUNT==0)?1:0;
 assign c2=({dav1_,dav2_,rfd}=='B110)?1:0;

//Registro HS
 always @(reset_==0) #1 HS<=1;
 always @(posedge clock) if (reset_==1) #3
  casex({b1,b0})
   'h2'B00: HS<=1;
   'h2'B01: HS<=HS;
   'h2'B10: HS<=0;
  endcase

[…]

//Registro COUNT
 always @(reset_==0) #1 COUNT<=7;
 always @(posedge clock) if (reset_==1) #3
  casex(b0)
   'B0: COUNT<=COUNT;
   'B1: COUNT<=COUNT-1;
  endcase
```

Per la parte controllo abbiamo (sempre saltando un po' di sintassi):

```
module Parte_Controllo(b0,c1,c0,clock,reset_);

 […]
 input c2,c1,c0;
 output b1,b0;

 reg STAR; parameter S0='B00, S1='B01, S2'B10;

 assign {b1,b0}=  (STAR==S0)?'B00:
                  (STAR==S1)?'B01:
                /*(STAR==S2)*/'B10;

 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)?S2:S1;
   S2: STAR<=(c2==1)?S0:S2;
  endcase
endmodule
```

Qualora si voglia vedere la parte controllo come ROM-based (micro-address-based o micro-instruction based), abbiamo:

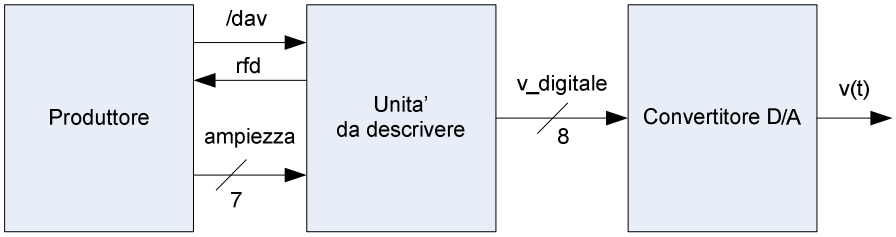
μ-addr	**μ-code** b_1b_0	C_{eff}	**μ-addr T**	**μ-addr F**
00 (S0)	00	00	01 (S1)	00 (S0)
01 (S1)	01	01	10 (S2)	01 (S1)
10 (S2)	10	10	00 (S0)	10 (S2)

3.5 Esercizio – tensioni analogiche

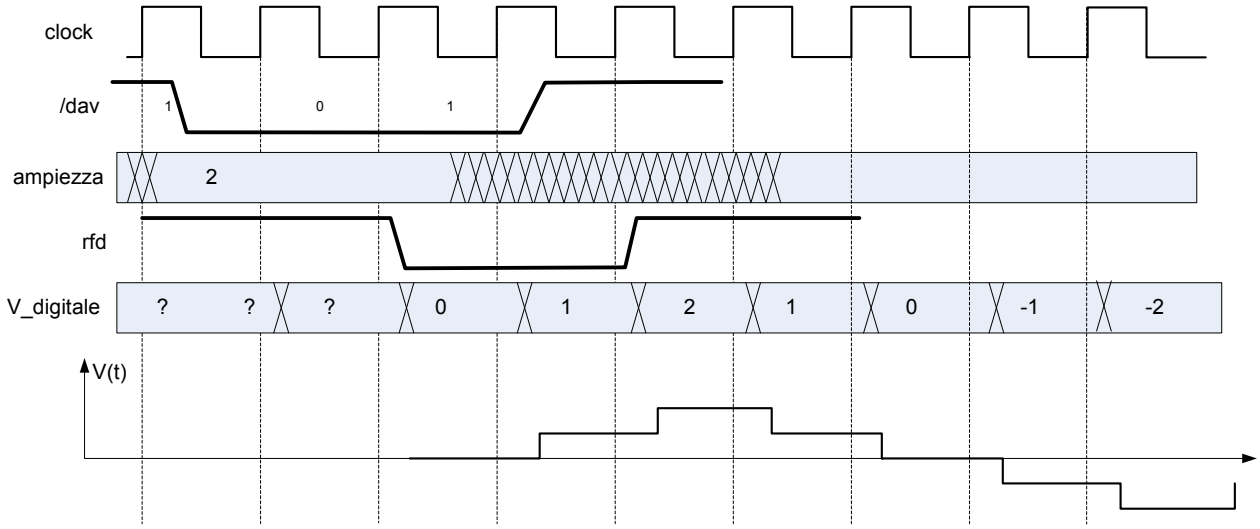
Descrivere l'unità di figura che opera ciclicamente nelle seguenti ipotesi:

- l'unità gestisce con il *produttore* un handshake classico, con passaggio di dati a **7 bit** (e non 8)
- l'unità interpreta il dato fornito dal produttore tramite la variabile *ampiezza* come un numero naturale N in base 2, e presenta la variabile di uscita *v_digitale* in ingresso ad un convertitore D/A. Tale variabile costituirà una sequenza di byte (al ritmo di uno per clock), che il convertitore interpreta in modo tale da produrre in uscita una tensione con forma d'onda triangolare di ampiezza pari ad N

Si faccia riferimento ad un convertitore che opera secondo la legge *binaria bipolare*. Si assuma che N sia diverso da zero.



Ad esempio:



3.5.1 Descrizione

L'esercizio parla di **binario bipolare**: i numeri da mandare al convertitore D/A sono **interi rappresentati in traslazione**. Devo poter mandare tutti i numeri compresi in $k \in [-N;+N]$, dove $0 < N \leq 2^7 - 1$. La rappresentazione in traslazione di un numero intero k su 8 bit è $K = k + 2^{8-1}$. Pertanto:

- $N = 1$: devo poter inviare i numeri $k = -1, 0, 1$, cioè $K = 2^7 + \{-1, 0, 1\}$;
- $N = 2$: devo poter inviare i numeri $k = +2, -1, 0, 1, +2$, cioè $K = 2^7 + \{-2 - 1, 0, 1, +2\}$;

- $N = 2^7 - 1$: devo poter inviare i numeri $k = -\left(2^7 - 1\right), ..., -1, 0, 1, ..., +\left(2^7 - 1\right)$, cioè $K = 2^7 + \left\{-\left(2^7 - 1\right), ..., -1, 0, 1, ..., +\left(2^7 - 1\right)\right\}$.

Quindi, nel caso peggiore devo inviare numeri naturali *K* compresi tra 1 e $2^8 - 1$, il che va bene perché ho otto bit a disposizione per l'uscita.

Per poter generare le tensioni digitali, dovrò scrivere **tre cicli**:

- Il primo ciclo, facendo uscire i numeri da 2^{8-1} a $2^{8-1} + N$ (a salire);
- Il secondo ciclo sarà a scendere da $2^{8-1} + N$ a $2^{8-1} - N$;
- Il terzo ciclo sarà a salire da $2^{8-1} - N$ a 2^{8-1}.

Il tutto facendo caso a contare gli estremi una volta sola.

Oltre a questo devo gestire l'handshake con il produttore. Dovrò campionare l'ampiezza sul fronte di discesa di */dav*, mettere *rfd* a zero, **e** dovrò attendere che */dav* sia tornato a uno prima di ricominciare. Si faccia caso al fatto che **non è scritto da nessuna parte** che una nuova iterazione del lavoro della rete deve iniziare **immediatamente** quando è finito il precedente. Peraltro, potrebbe non essere possibile se il produttore è lento a riportare */dav* ad 1.

Detto questo, cerchiamo di capire di quali **registri** posso aver bisogno:

- Un registro OUT per sostenere l'uscita *v_digitale*, che dovrà essere ad 8 bit.
- Un registro RFD per sostenere l'uscita omonima (ad 1 bit)
- Visto che devo confrontare il valore di OUT con delle costanti che dipendono dall'ampiezza campionata, potrebbe farmi comodo memorizzare queste costanti in due registri MAX e MIN, entrambi a 8 bit.
- Un registro STAR, da dimensionare opportunamente alla fine della descrizione.

Per quanto riguarda le **condizioni al reset**:

- Potrò dare per scontato che */dav*=1
- Dovrò fare in modo che RFD=1, e che *v_digitale*= 2^{8-1}, quindi OUT= 2^{8-1}.

Conviene definire una *costante*:

```
parameter zero='H80;
```

In modo da rendere la descrizione più leggibile.

Descriviamo ora cosa dovrebbe succedere nei vari stati:

S0: si arriva qui dal reset, e si dovrà campionare *ampiezza* finché /*dav* non va a zero. Teniamo RFD=1, OUT=*zero*, e possiamo assegnare MAX=*zero* + *ampiezza*, MIN=*zero* – *ampiezza*. Si va in S1 quando /*dav* va a zero.

S1: si deve portare RFD a zero per far proseguire l'handshake, e poi bisogna eseguire il primo ciclo: si incrementa OUT, e si testa se OUT è arrivato a MAX. Quando questo succede si va in S2.

S2: eseguire il secondo ciclo: si decrementa OUT, e si testa se OUT è arrivato a MIN. Quando questo succede si va in S3.

S3: eseguire il terzo ciclo: si incrementa OUT, e si testa OUT è arrivato a *zero*.

S4: si finisce di gestire l'handshake (si testa se /*dav*=1) e si torna in S0.

Quindi, a posteriori, posso dire che servono cinque stati interni, quindi STAR deve essere di 3 bit.

Come si fa a testare se OUT ha raggiunto il valore necessario (e.g., MAX)?

- Se lo sto **incrementando**, dovrò scrivere

```
Sx: begin … OUT<=OUT+1; STAR<=(OUT==MAX-1)?Sy:Sx; … end
```

- Se lo sto **decrementando**, dovrò scrivere

```
Sx: begin … OUT<=OUT-1; STAR<=(OUT==MIN+1)?Sy:Sx; … end
```

In quanto la condizione viene testata sul **vecchio** valore di OUT, quello **prima del** clock. All'arrivo del clock, OUT verrà incrementato o decrementato ancora una volta.

```
module XXX (out, dav_, rfd, ampiezza, clock, reset_);
 input          clock,reset_;
 input   [6:0] ampiezza;
 input          dav_;
 output         rfd;
 output  [7:0] out;
 reg            RFD;
 reg     [7:0] OUT, MAX, MIN;
 reg     [2:0] STAR;
 parameter S0=0, S1=1, S2=2, S3=3, S4=4, zero='H80; // zero=128
 assign   rfd=RFD;
 assign   out=OUT;
 always @(reset_==0) #1 begin RFD<=1; OUT<=zero; STAR<=S0; end
 always @(posedge clock) if (reset_==1) #3
  casex(STAR)
   S0:  begin RFD<=1; OUT<=zero; MAX<=zero+{'B0,ampiezza};
              MIN<=zero-{'B0,ampiezza}; STAR<=(dav_==0)?S1:S0; end
   S1:  begin RFD<=0; OUT<=OUT+1; STAR<=(OUT==MAX-1)?S2:S1; end
   S2:  begin OUT<=OUT-1; STAR<=(OUT==MIN+1)?S3:S2; end
   S3:  begin OUT<=OUT+1; STAR<=(OUT==zero-1)?S4:S3; end
   S4:  begin STAR<=(dav_==0)?S4:S0; end
  endcase
endmodule
```

Possibili ottimizzazioni:

se si evita di porre RFD a 0 in S1, e lo si lascia quindi a 1 fino a S3 (o S4), si può usare l'ingresso *ampiezza* per fare i conti. In questo caso, non c'è più bisogno dei registri MAX e MIN, in quanto si può usare direttamente il valore *ampiezza* in S1 ed S2. Posso riscrivere quindi le condizioni in S1 e S2 come:

```
STAR<=(OUT==zero+ampiezza-1)?...

STAR<=(OUT==zero-ampiezza+1)?...
```