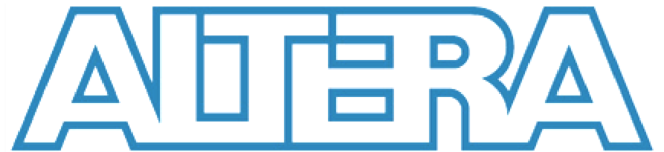


La sintesi in Verilog HDL

Il linguaggio Verilog HDL per la sintesi di sistemi digitali

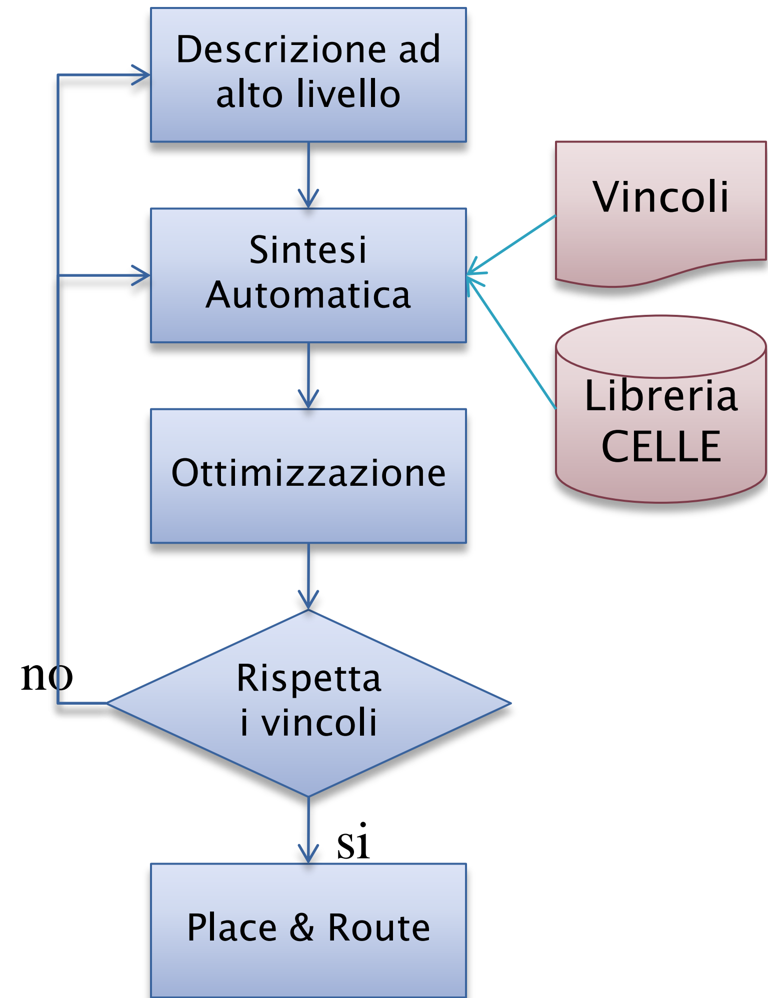
TESTI Consigliati

- J. Bhasker
  Verilog HDL Synthesis – A pratical Primer
  Star Galaxy Publisher

ALTERA

Il processo di Sintesi

- La sintesi
  - è un processo di conversione da un alto livello di astrazione ad una rappresentazione ottimizzata a livello di gate
- Sfrutta una libreria di Celle già definite
  - basic logic gates like and , or , not nand, nor,…
  - macro cells, such as adder, muxes, memory, and special flip-flops
- Il progettista dovrebbe
  - Prima interpretare e descrivere opportunamente l'architettura del sistema.
  - Poi considerare I vincoli quali timing, area, testability, and power



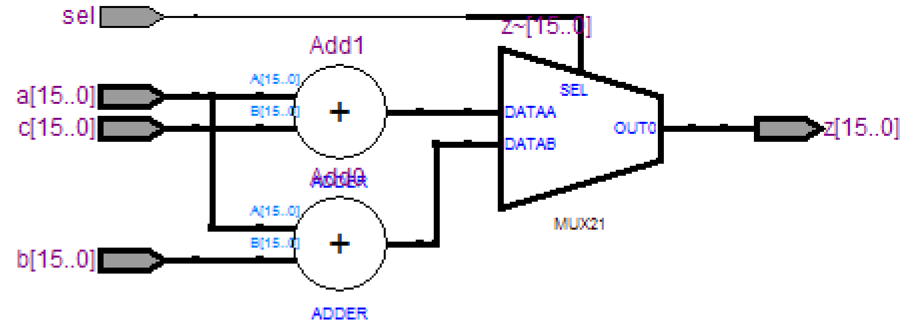
ALTERA

Contributo del "code Style"

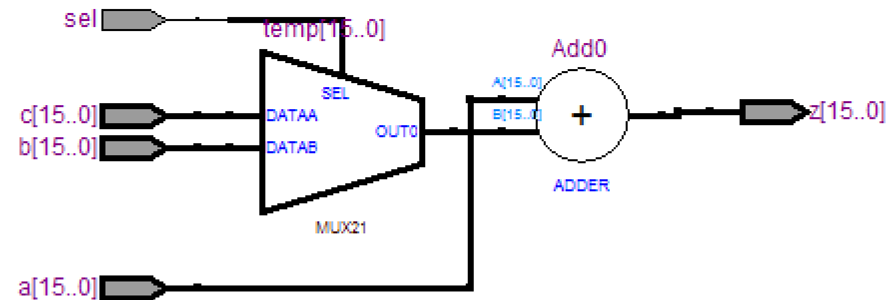
- Il processo di sintesi è guidato da: descrizione in HDL del sistema, dai vincoli, da eventuali processi di ottimizzazione.
- Nel raggiungimento di certi obiettivi il contributo al successo del progetto è da ricercarsi:
  - 80 % nella descrizione in HDL del sistema
  - 15% all'imposizione di vincoli e ad un processo di sintesi ben organizzato
  - 5% ad eventuali processi di ottimizzazione
- Es: si voglia progettare un sistema che funzioni a 100 MHz:
  - Una buona stesura del codice dovrebbe portare senza ottimizzazioni e senza imporre vincoli particolari a raggiungere una freq. di lavoro di almeno 80 MHz ... poi attraverso ottimizzazioni sw si può ottenere un miglioramento delle prestazioni di circa il 20%

Esempio

```
always @(sel or a or b or c)
    if (sel)
        z = a + b;
    else
        z = a + c;
```




```
always @(sel or b or c)
    if (sel)
        temp = b;
    else
        temp = c;

always @(temp or a)
    z= a+ temp;
```

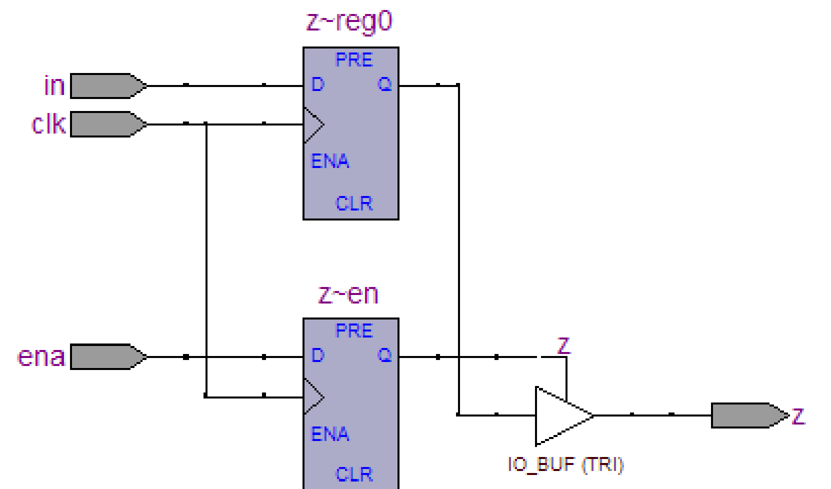


ALTERA

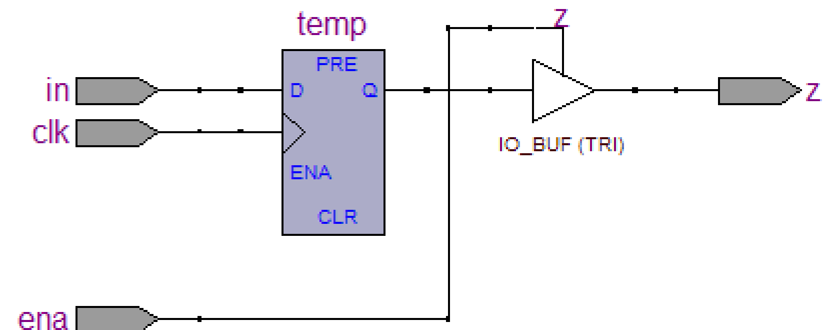
Logica Combinatoria e Sequenziale

- Tenere separata la logica combinatoria da quella sequenziale

```
always @(posedge clk)
begin
    if (ena)
        z=in;
    else
        z=1'bz;
    endendmodule
```




```
always @(posedge clk) begin
    temp=in;
end

always @(ena or temp)  begin
    if (ena)
        z=temp;
    else
        z=1'bz;
    endendmodule
```

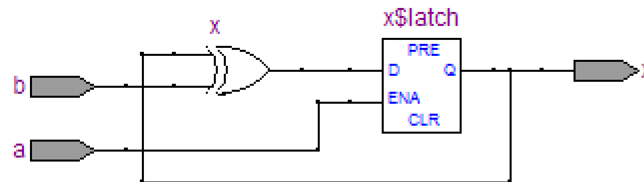


ALTERA

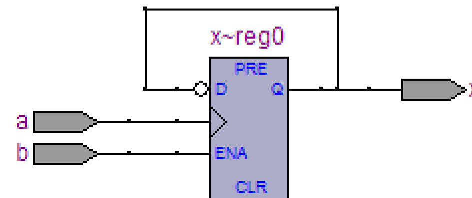
Evitare i latches

- L’impiego di latches può portare a “corse critiche” il cui risultato può essere aleatorio
- Molto meglio usare FF

```
always @(a or b)
begin
     if (a)
     x<=x^b;
end
```




```
always @(posedge a)
begin
     if (a)
     x<=x^b;
end
```

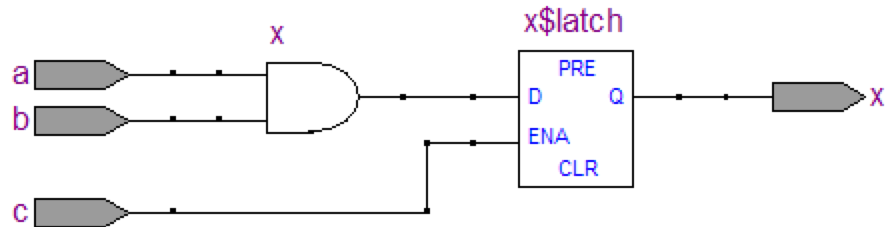


- Un codice scritto in modo poco accurato induce il sintetizzatore a introdurre (inutilmente) dei latches per memorizzare alcuni valori
  - Non bisogna lasciare assegnazioni implicite o sottointese

ALTERA

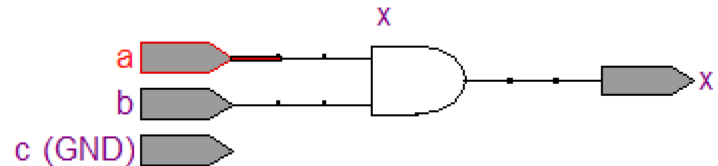
Esempio

```
always @(a or b or c)
begin
    if (c)
        x = a & b;
end
```



Viene implicato *("inferred")* un latch per memorizzare l'uscita

```
always @(a or b or c)
begin
    if (c)
        x = a & b;
    else
        x = 1'bx;
end
```



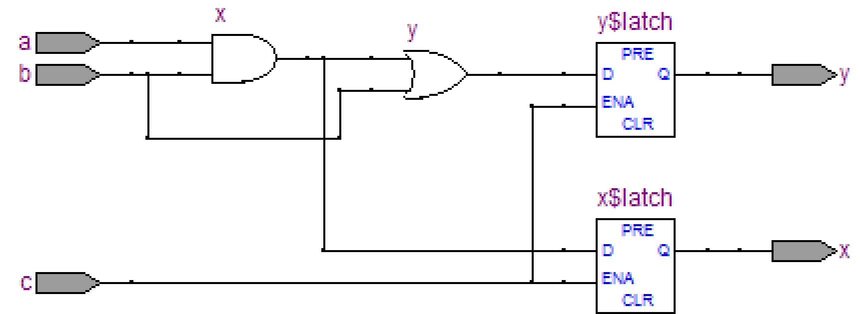
Soluzione puramente combinatoria

ALTERA

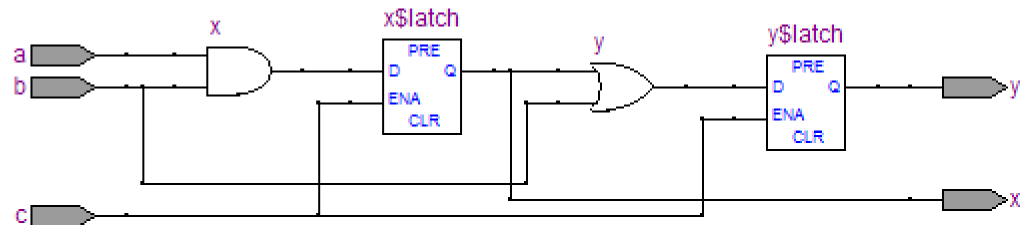
Esempio

Il rischio è inoltre quello di ottenere risultati imprevisti

```
always @(a or b or c)
begin
    if (c) begin
        x = a & b;
        y = x | b;
    end
end
```




```
always @(a or b or c)
begin
    if (c) begin
        x <= a & b;
        y <= x | b;
    end
end
```




```
always @(a or b or c)
begin
    if (c) begin
        y = x | b;
        x = a & b;
    end
end
```

ALTERA

Regola #1

If the procedure has several paths, every path must evaluate all outputs

- **Method1**:
  Set all outputs to some value at the start of the procedure.
  Later on different values can overwrite those values.

```
always @(...
    begin
        x=0;y=0;z=0;
        if (a) x=2; elseif (b) y=3; else z=4;
End
```

- **Method2**:
  Be sure every branch of every if and case generate every output

```
always @(...
    begin
        if (a) begin        x=2; y=0; z=0; end
        elseif (b) begin    x=0; y=3; z=0; end
        else begin          x=0; y=0; z=4; end
end
```

ALTERA

Regola #2

All inputs used in the procedure must appear in the trigger list

- **Right-hand side variables:**
Except variables both calculated and used in the procedure.

```
always @(a or b or c or x or y)
    begin
        x=a; y=b; z=c;
        w=x+y;
    end
```

(In the trigger list, "x or y" is crossed out.)

- **Branch controlling variables:**
Be sure every branch of every if and case generate every output

```
always @(a or b)
    begin
        if (a)        begin    x=2; y=0; z=0; end
        elseif (b)    begin    x=0; y=3; z=0; end
        else          begin    x=0; y=0; z=4; end
    end
```

ALTERA

Regola #3

All possible inputs used control statements must be covered

- End all case statements with the default case whether you need it or not.

```
case(state)
  ...
  default: next_state = reset;
Endcase
```

- Do not forget the self loops in your state graph

```
if(a|b&c) next_state=S1;
elseif(c&d) next_state=S2;
else next_state=reset;
```

ALTERA

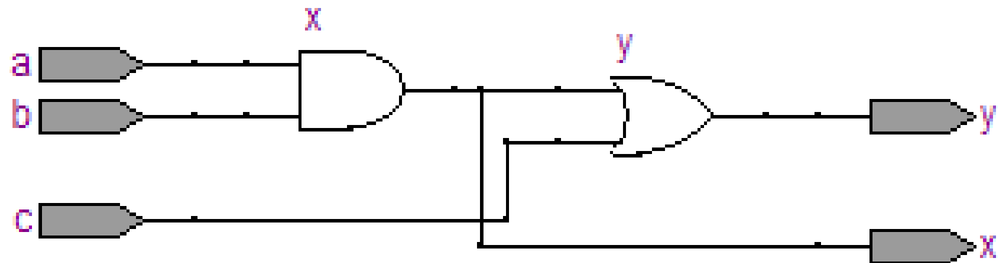
Block and NON Block assignments

<= NON blocking assignment

= Blocking assignment

In se non hanno significativi effetti sulla logica combinatoria

```
always @(a or b or c)
begin
    x = a & b;
    y = x | c;
end

always @(a or b or c)
begin
   y = x | c;
   x = a & b;
end
```



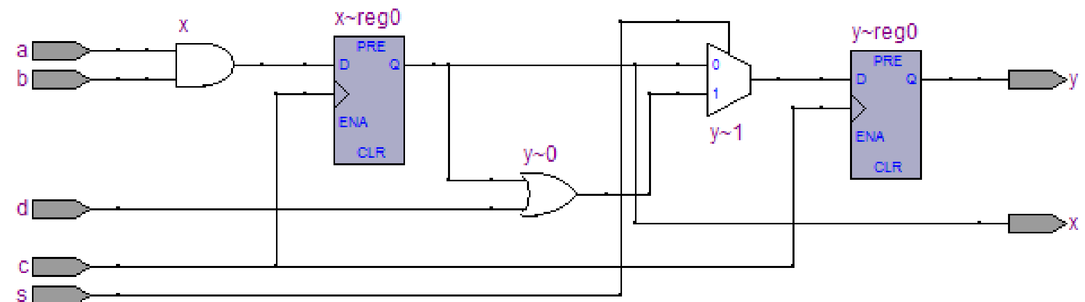
Raccomandazioni

- Blocking assignment per la logica combinatoria
- NON Blocking per la logica sequenziale
- Non usare contemporaneamente Blocking e NON Blocking assigmentents nella setssa procedura

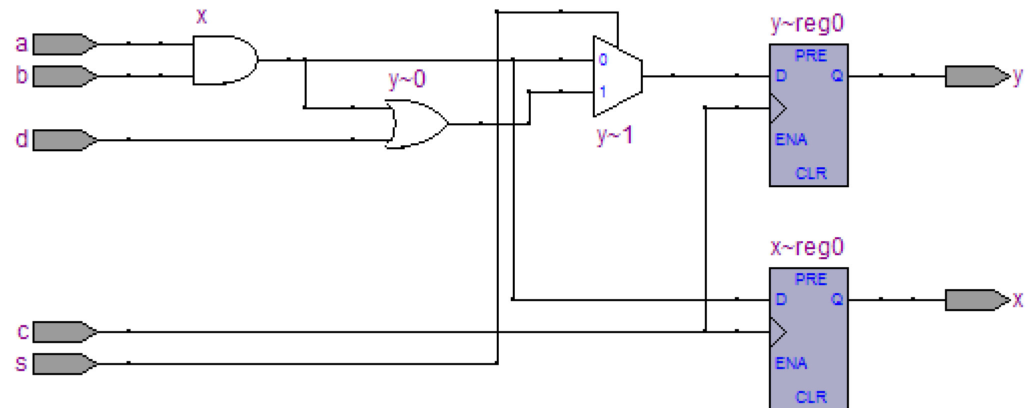
ALTERA

Block and NON Block assignments

```
always @(posedge c)
begin
                x<=a&b;
        if (s)
                y<=x|c;
        else
                y<=x;

end
```




```
always @(posedge c)
begin
                x=a&b;
        if (s)
                y=x|d;
        else
                y=x;

end
```



ALTERA

Ritardi

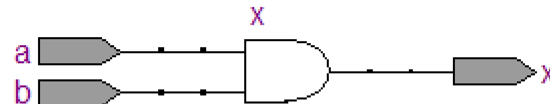
- I ritardi non vengono sintetizzati generando così probabili incongruenze fra la descrizione comportamentale ed il circuito sintetizzato
- In Quartus la simulazione funzionale è del circuito RTL e non del Verilog Sorgente … pertanto non è nemmeno possibile simulare eventuali ritardi.

ALTERA

Always List

- La lista associata al costrutto always (*event list*) DEVE contenere tutte le variabili in ingresso della logica **combinatoria**
  - Altrimenti ci sarebbe una discrepanza tra quanto descritto (e forse simulato) a livello comportamentale ed il circuito sintetizzato
    - Nota: In quartus NON si esegue la simulazione del sorgente comportamentale Verilog, ma della sua conversione in RTL

```
always @(b)
begin
    x=a&b;
end
```



A livello comportamentale le variazioni di ‘a’ non dovrebbero aver effetto sull’uscita, ma nel circuito sintetizzato, ovviamente si. In genere in fase di compilazione c’è un WARNING

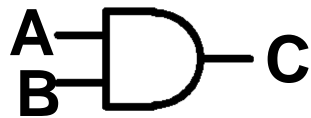
ALTERA

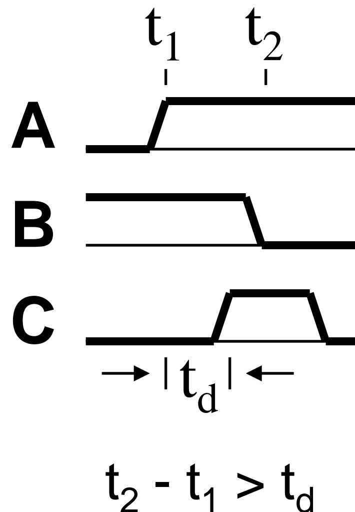
Sincroni o Asincroni

- Sincroni
  - Impiegano un clock (tempo campionato)
  - Disegno più semplice
  - Circuito più lento (worst case)
  - Elementi di memoria: FF
  - Facile esportabilità
- Asincroni
  - Impiegano un protocollo di handshaking
  - Problemi di hazards, corse critiche, glicthes
  - Circuiti più veloci
  - Memoria nei ritardi (difficili da prevedere)
  - Difficile esportabilità
  - Importantissima la stima dei ritardi e l'analisi temporale
    - Però i ritardi dipendono da molti parametri (temp, p&R, device …)

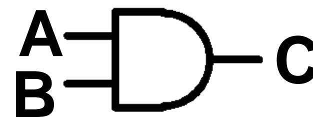
ALTERA

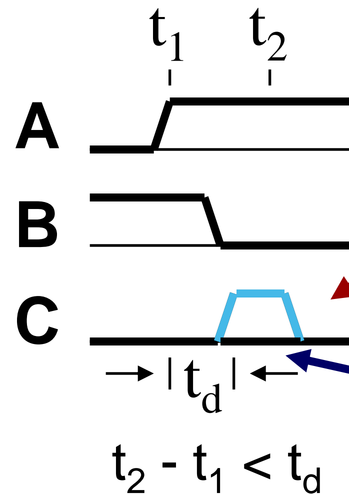
Glitches





$t_2 - t_1 > t_d$





$t_2 - t_1 < t_d$

Event-driven model:
Narrow Pulse

Real gate:
What really happen ?

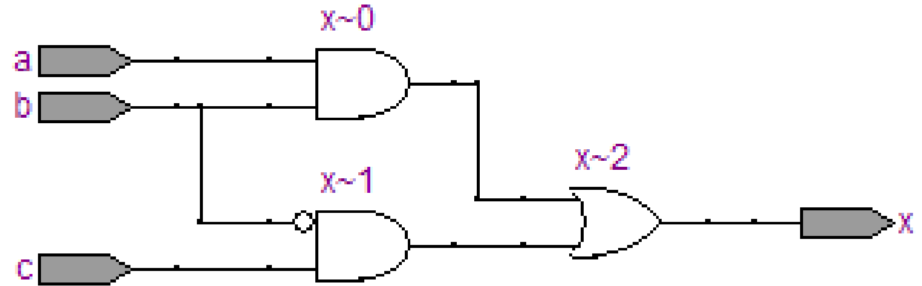
ALTERA

Alee

- Es:

  a=b=c=1 → x=1

  b=0 → x=1

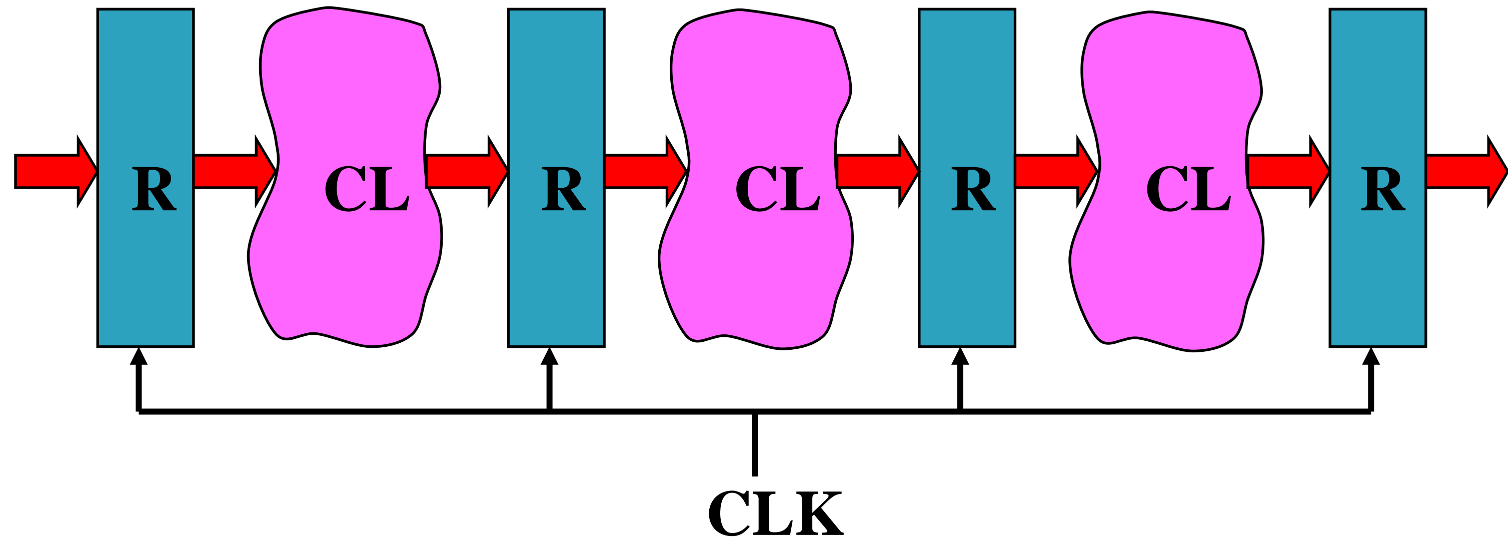


Si spegne il percorso alto e si accende il percorso basso, ma se i tempi di propagazione sono diversi tra i due rami si può aver un momento in cui l'uscita va a 0

Se sono presenti "loop" (memorie) il risultato può essere imprevisto

ALTERA

Circuiti Sincroni



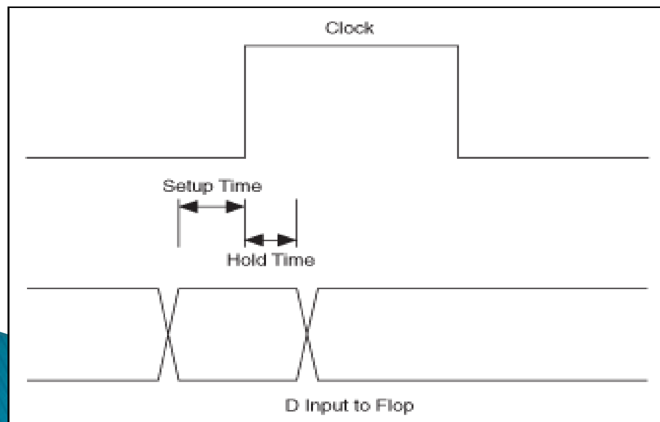
Implicit (global) synchronization between blocks

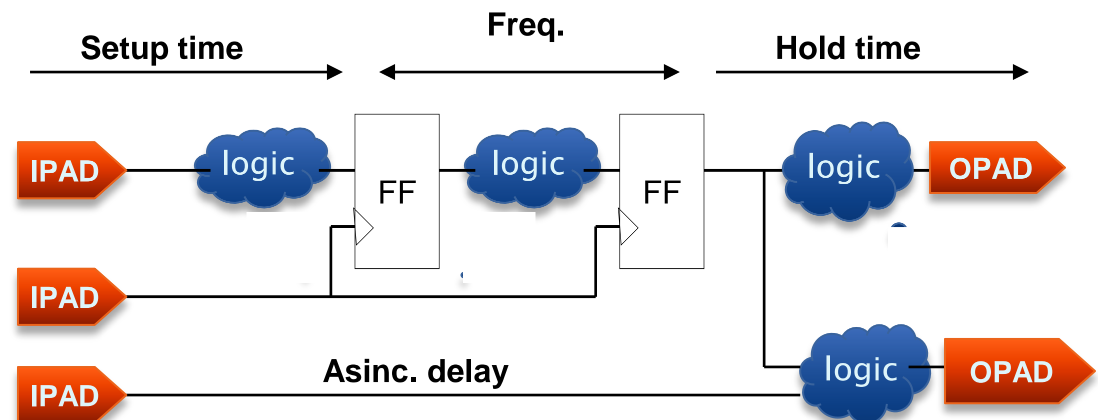
Clock period > Max Delay (CL + R)

ALTERA

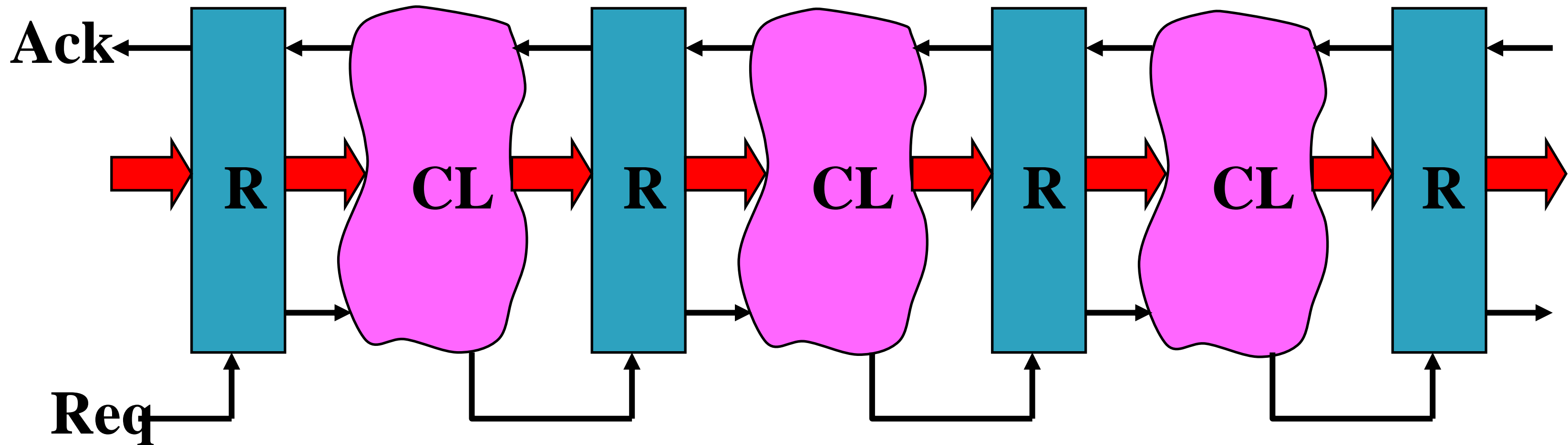
Circuiti Sincroni

- Un clock controlla i segnali (sul fronte)
  - Eventuali variazioni di ingressi/uscite non vengono rilevate o propagate fino al nuovo fronte
  - Ci sono vincoli sula temporizzazione dei dati (Tsu, Th, Freq.)
  - Se il vincolo viene violato il risultato è imprevedibile (stato metastabile)
    - Maggiore ritardo di propagazione
    - Uscita non corretta
    - Raramente si possono innescare oscillazioni





Circuiti Asincroni



Explicit (local) synchronization:

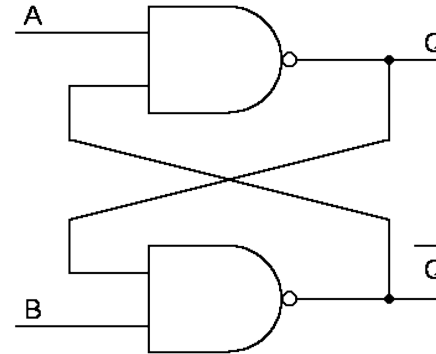
Req / Ack handshakes

Time = events + quantity

Time does not exist if nothing happens (Aristotle)

ALTERA

Circuiti asincroni



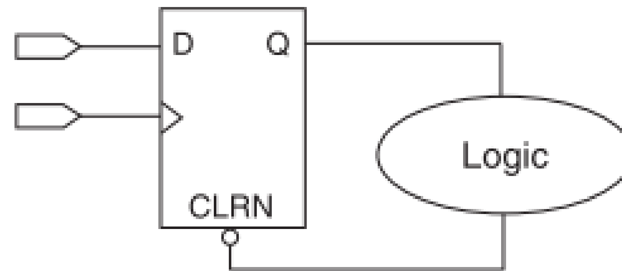
A B	Qn+1
1 1	Qn
1 0	0
0 1	1
0 0	?

- Impiego di Feedback
  - Per il protocollo di handshaking
  - Per ricordare lo stato del sistema
- Il tempo di ritardo dei vari rami del circuito è cruciale
  - Si possono generare glitches (che propagati a ritroso possono dare risultati imprevedibili)
  - Si possono innescare corse critiche e alee.
- I tempi di ritardo dipendono da molti parametri
  - Tecnologia adottata (o dal dispositivo FPGA)
  - Place and Route (o da diverse compilazioni del circuito)
  - Temperatura di esercizio,
  - ….
- Il progetto deve essere molto accurato ed i vincoli molto precisi

ALTERA

Loop Combinatorio

- E’ una delle principali cause di instabilità
- Nasce ad esempio se una variabile è presente tanto a sinistra che a destra dell’assegnazione
  (Es: a <= b+a;)
  - Il suo comportamento dipende dal ritardo di propagazione
  - Può degenerare un loop infinito.
- In un circuito sincrono deve includere dei registri (ma ciò non vale non per i suoi eventuali controlli asincroni)



- Può venir evidenziato dal *Time Quest Timing Analysis* – Recovery and removal Analysis

ALTERA

Latches

- Genericamente **sono da evitare**
- Possono nascere erroneamente da codici imprecisi
- A differenza di altre tecniche in un' FPGA un latch "costa" quanto un FF è non è più veloce.
- Quando il latch è trasparente può essere attraversato da glicthes ma il Time Quest analyzer (trattandolo come circuito sincrono) non evidenzia questo comportamento
- L'uso di latches inficia la possibilità di una corretta analisi temporale del circuito

ALTERA

Delay Chains

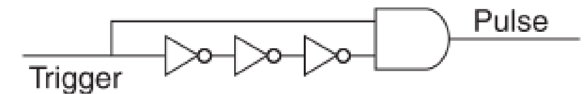
- Catene di elementi di ritardo (ad esempio invertitori) per creare ritardi di propagazione **sono da evitare**
- Negli ASIC a volte sono usati per bufferizzare i segnali; nelle FPGA tutti i segnali sono già bufferizzati opportunamente all’interno dei canali di trasmissione
- I ritardi possono cambiare in base alle diverse compilazioni (P&R) ed inoltre rendono il circuito non esportabile verso altre tecnologie
- Da evitare in particolare ritardi sulla linea del clock (disallineamento)

ALTERA

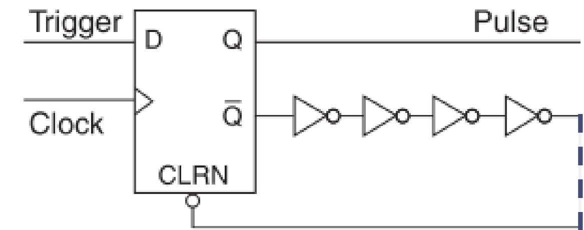
Pulse Generators & Multivibrators

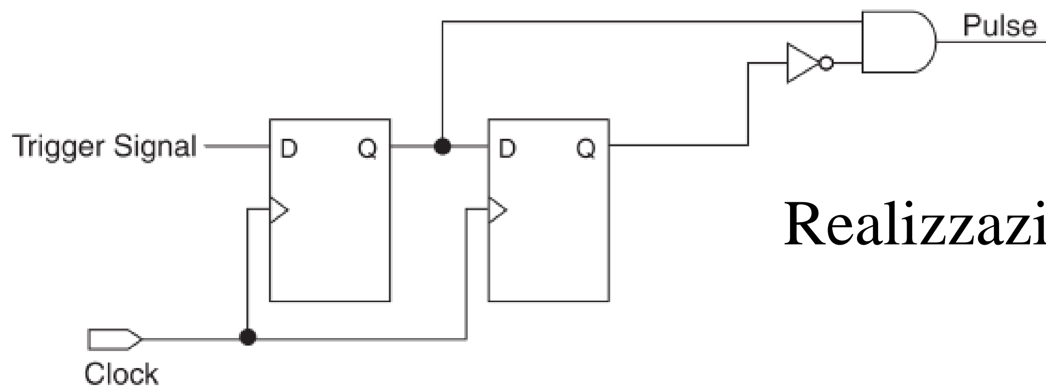
- Tramite catene di ritardi si potrebbero realizzare
  - Generatori di impulsi
  - Multivibratori (oscillatori digitali)
- Il loro funzionamento (asincrono) non è prevedibile e non può essere definito a priori

Using an AND Gate



Using a Register





Realizzazione corretta

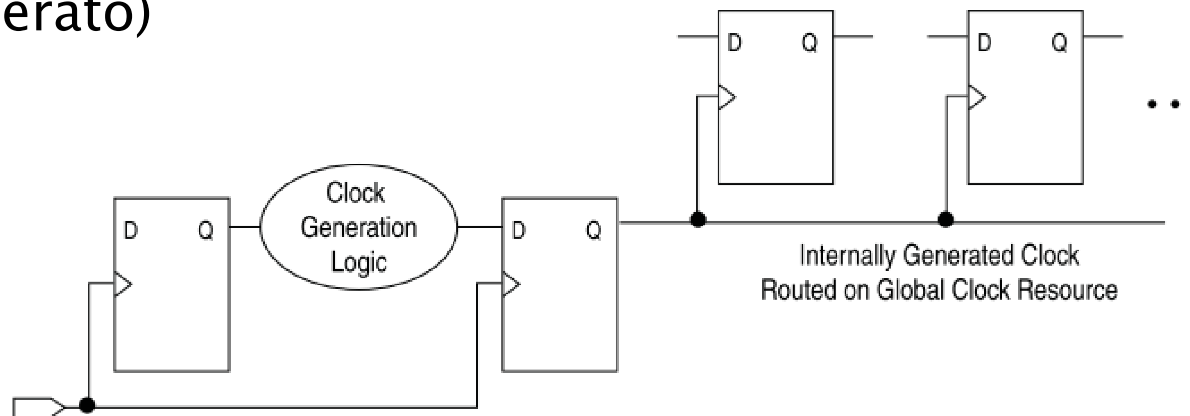
ALTERA

Clock schemes

- Evitare l'uso di clock generati internamente al circuito – ovvero segnali adibiti a clock (potrebbero presentare gliches)
- Evitare ove possibile il trasferimento di dati tra sistemi controllati da clock diversi (eventualmente usare una FIFO)

Generazione di Clock

- Usando un circuito logico per generare clock o reset asincroni comporterà facilmente la presenza di glitches che sono deleteri sul clock.
  - Possono portare a violazioni di Tsu e Th
  - Introducono "nuovi" cicli di elaborazione
- Si dovrebbe per lo meno usare un sistema che sincronizzi l'uscita della rete logica
- E' sconsigliato comunque realizzare sistemi che adottino contemporaneamente entrambi i clock (in ingresso e generato)



ALTERA

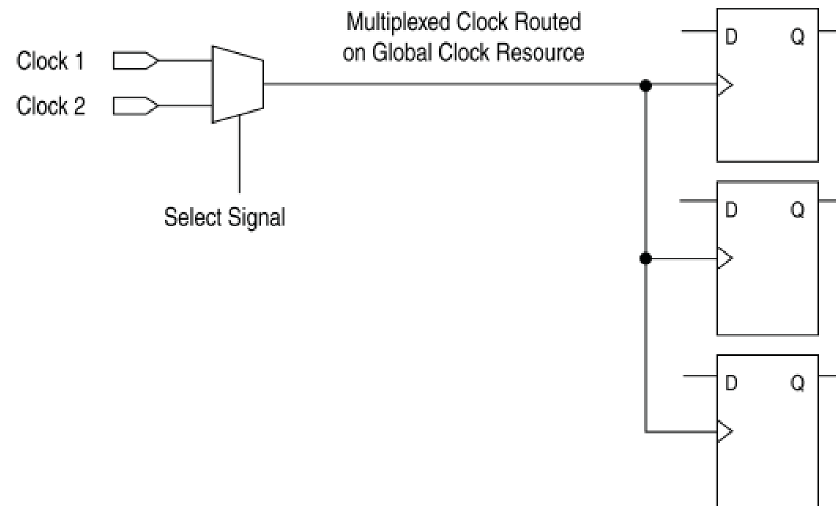
Divisori di clock

- Nelle FPGA ci sono PLL dedicate (consigliato)
- Se si usa una rete logica per lo meno sincronizzare le uscite col clock di ingresso
- NO a ripple counters (catene di FF-T ove l'uscita dell'uno è il clock del seguente)
- Evitare circuiti asincroni per la decodifica di contatori

ALTERA

Clock Multiplexing

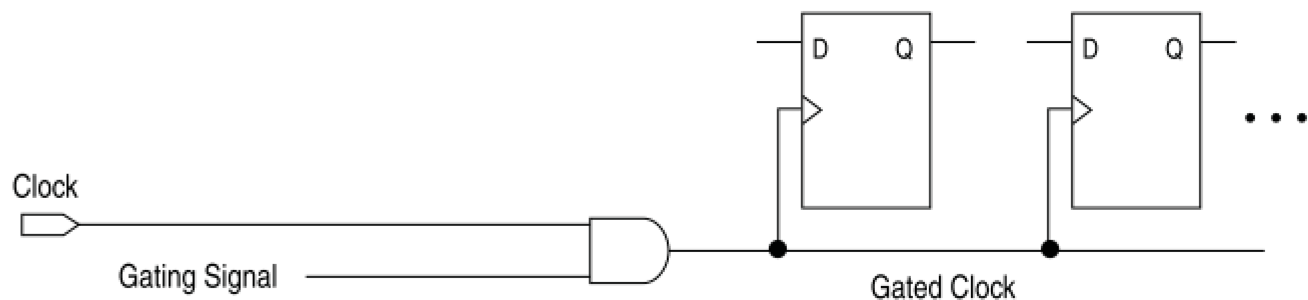
- Utile per far funzionare un circuito a diverse frequenze
- E' consigliato adottare le risorse interne dedicate dell'FPGA
- Al clock-switch si resettino tutti i registri
- Una eventuale errore nel transitorio da un clock all'altro non abbia conseguenze negative.



ALTERA

Gated Clock

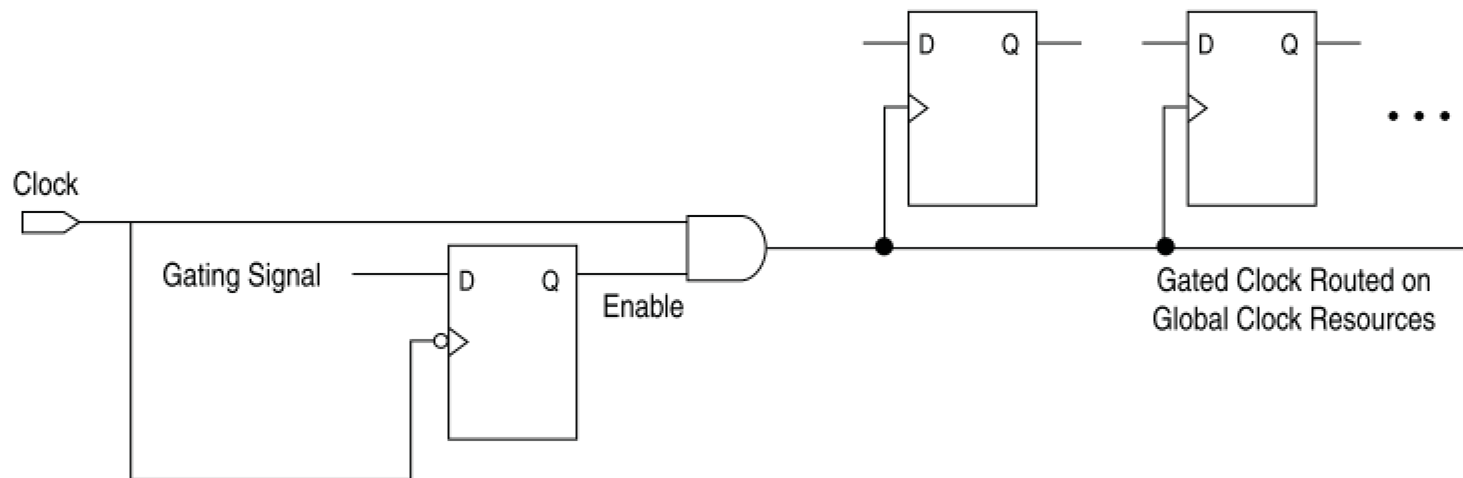
- Può essere utile per limitare la potenza dissipata
- E’ sconsigliato perché comporta un disallineamento del clock (clock skew)
- Può rendere il clock sensibile ai glitches
- Piuttosto che adottare porte logiche, nelle FPGA ci sono circuiti dedicati per spegnere il clock
- Si può usare il piedino di CE (però non limita la potenza)



ALTERA

Sincronous Gated Clock

- Se è richiesto il Gated Clock per limitare la potenza
- Usare comunque le risorse di routing dedicate
- Adottare lo schema riportato per evitare glitches
  - Il FF aggiunto campiona il gate signal sul fronte di discesa
  - La logica che genera il “Gating Signal” ha a disposizione ½ ciclo di clock per ottenere il risultato



ALTERA

Esempi

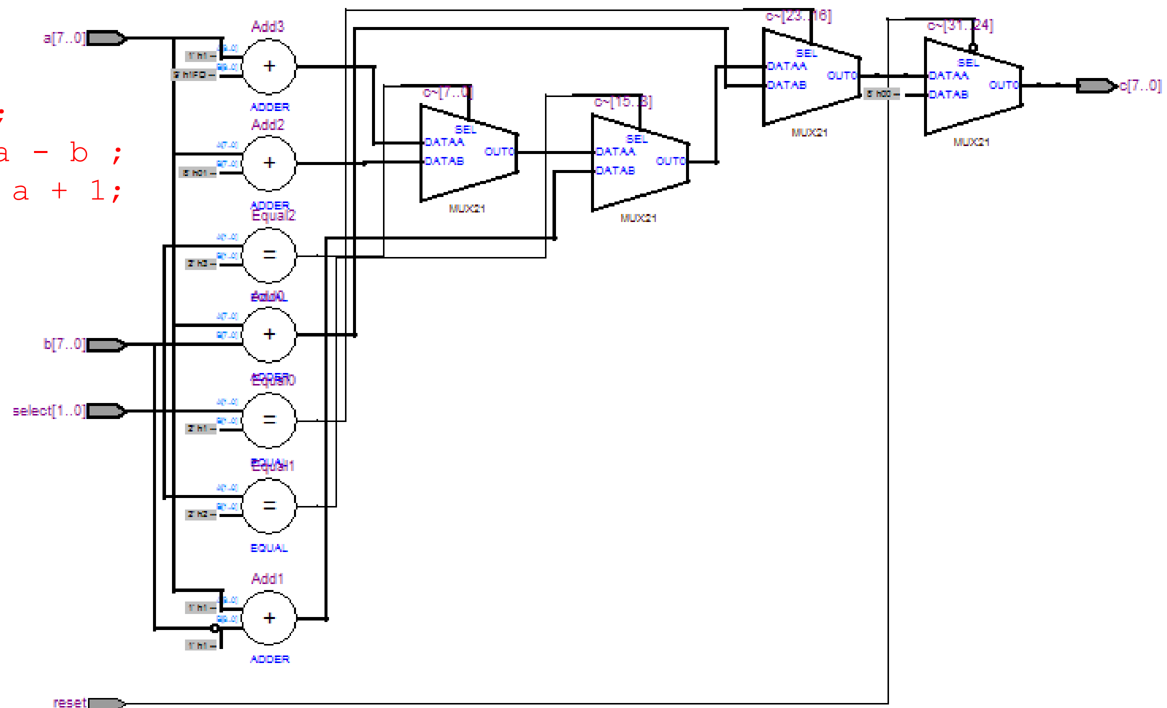
- Alcuni casi in cui in cui il codice influenza la sintesi e soprattutto i risultati
  - Utilizzo di “if” oppure di “case”
  - Pipelining della logica
  - Gestione di segnali ad alto Fan-Out
  - Gestione dei segnali di reset
  - Inferring (implicazione) di memorie

ALTERA

IF … else

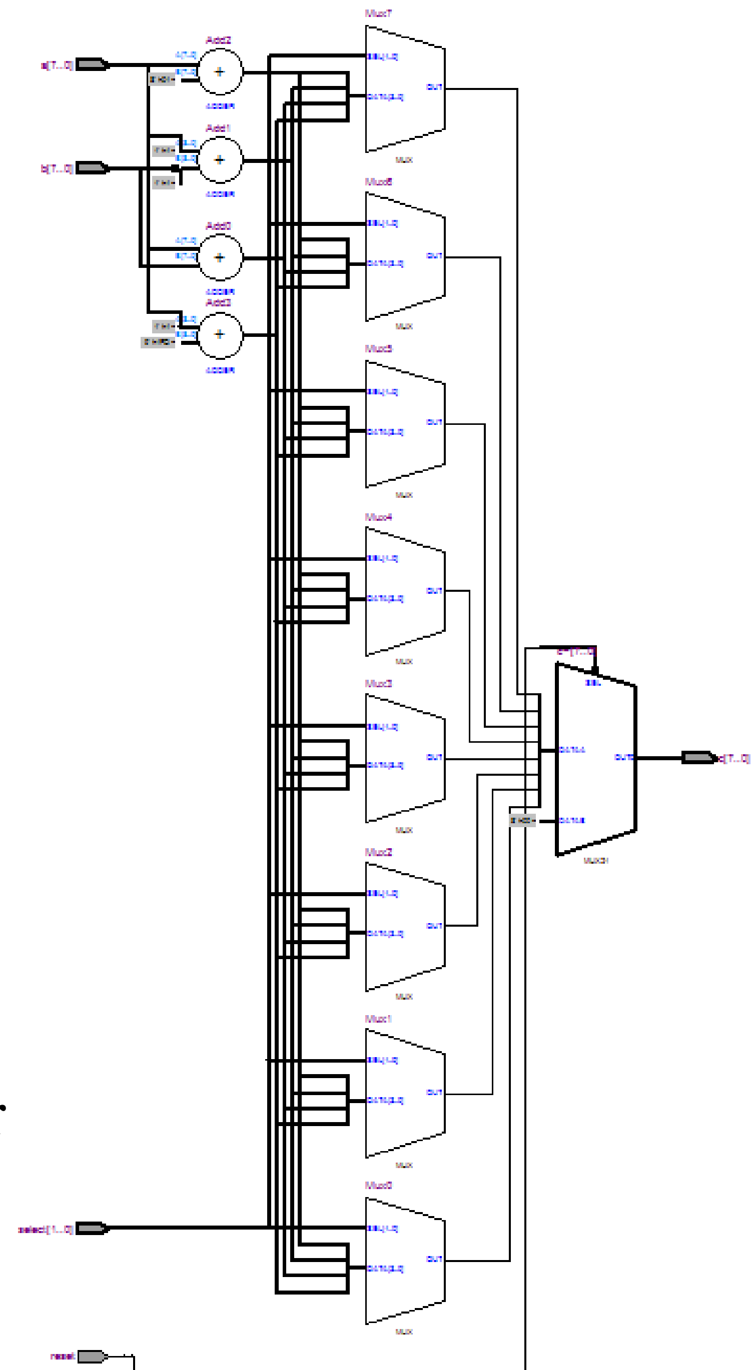
```
module test (reset, select, a,b,c);
input reset;
input [1:0] select;
input [7:0] a, b;
output [7:0] c;
reg  [7:0] c;

always@(select or reset or a or b or c) begin
if (!reset) begin
   c <=0;
end else begin

   if (select == 2'b01) c <= a + b;
   else if (select == 2'b10) c <= a - b ;
   else if (select == 2'b11) c <=  a + 1;
   else  c <= a -1;

 end
end

endmodule
```

Priority encoder



CASE

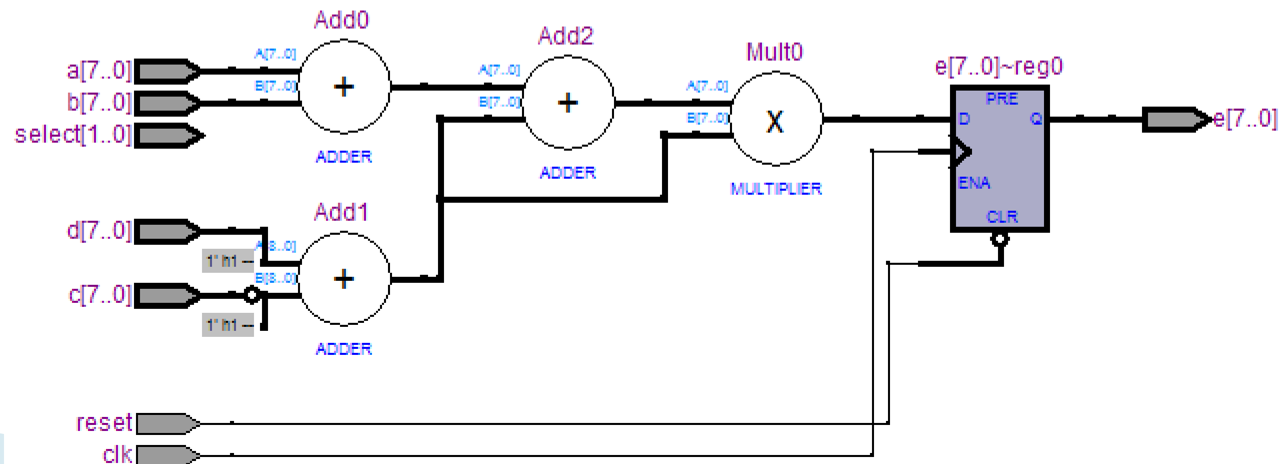
```
module test (reset, select, a,b,c);
input reset;
input [1:0] select;
input [7:0] a, b;
output [7:0] c;
reg  [7:0] c;

always@(select or reset or a or b or c) begin
if (!reset) begin
   c <=0;
end else begin

 case (select)
   2'b01 : c <= a + b;
   2'b10 : c <= a - b;
   2'b11 : c <= a + 1;
   2'b00 : c <= a - 1;
   endcase

 end
end

endmodule
```



NON priority encoder

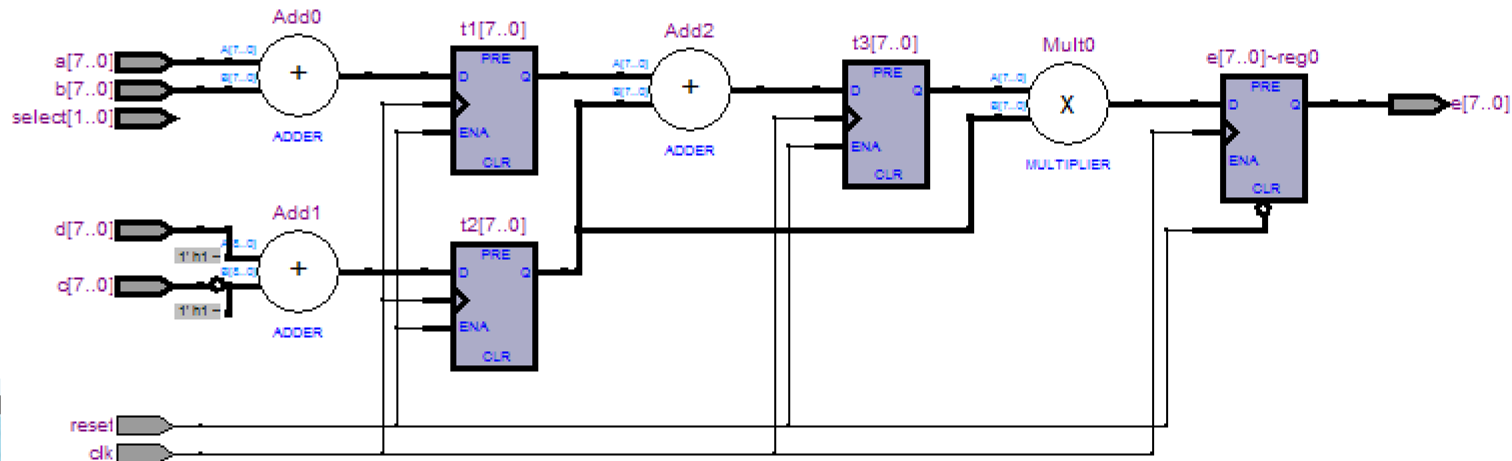
Pipelining

```
module test (clk, reset, select, a,b,c,d,e);
input reset;
input clk;
input [1:0] select;
input [7:0] a, b, c,d;
output [7:0] e;
reg  [7:0] e;
reg [7:0] t1,t2,t3,t4;

always@(posedge clk or negedge reset) begin
if (!reset) begin
   e <=0;
end else begin
t1 = a + b;
t2 = d - c;
t3 = t1 + t2;
e  = t3 *t2 ;
 end
end

endmodule
```



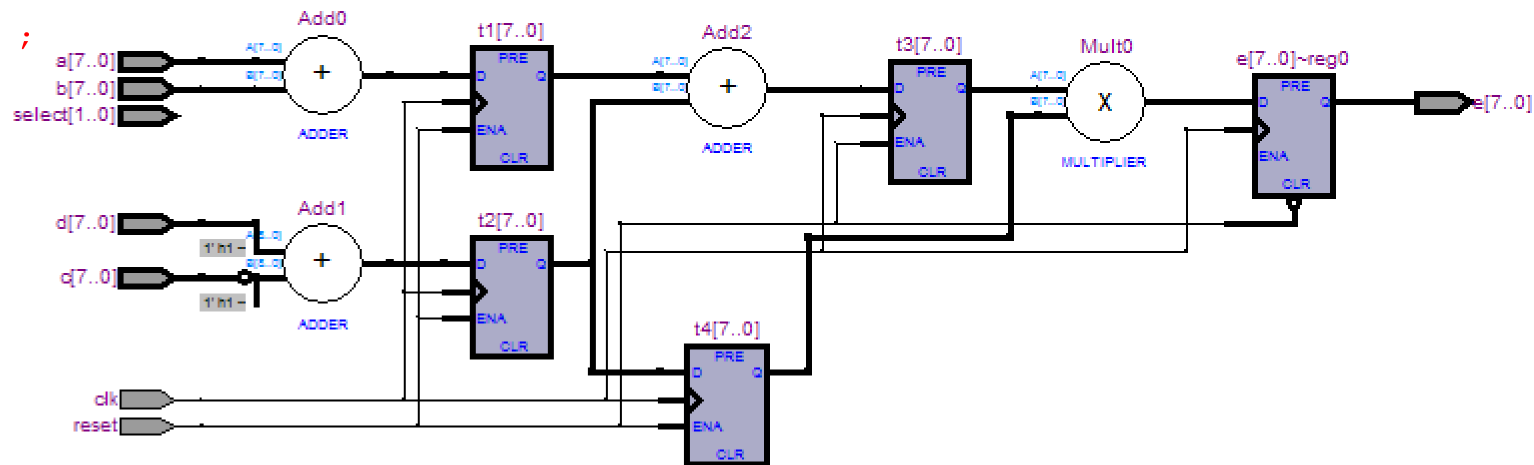
Pipelining

```
module test (clk, reset, select, a,b,c,d,e);
input reset;
input clk;
input [1:0] select;
input [7:0] a, b, c,d;
output [7:0] e;
reg  [7:0] e;
reg [7:0] t1,t2,t3,t4;

always@(posedge clk or negedge reset) begin
if (!reset) begin
   e <=0;
end else begin
t1 <= a + b;
t2 <= d - c;
t3 <= t1 + t2;
e  <= t3 *t2 ;
 end
end

endmodule
```



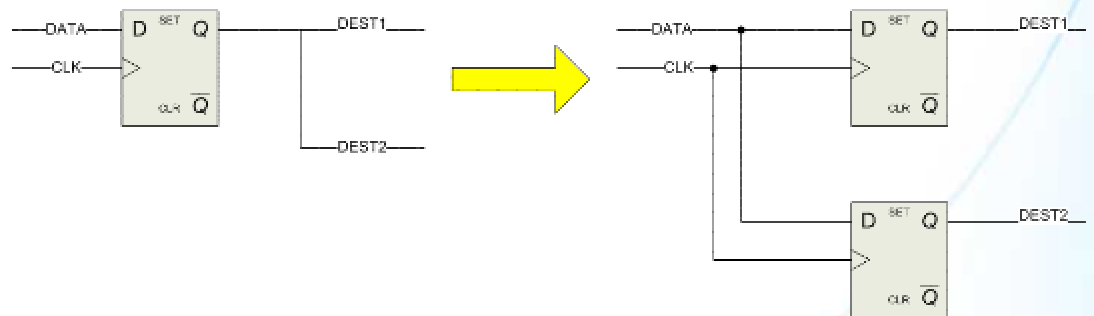
Pipelining

```
module test (clk, reset, select, a,b,c,d,e);
input reset;
input clk;
input [1:0] select;
input [7:0] a, b, c,d;
output [7:0] e;
reg  [7:0] e;
reg [7:0] t1,t2,t3,t4;

always@(posedge clk or negedge reset) begin
if (!reset) begin
   e <=0;
end else begin
t1 <= a + b;
t2 <= d - c;
t3 <= t1 + t2;
t4 <= t2 ;
e  <= t3 *t4 ;
end
end

endmodule
```



High fan-out signals

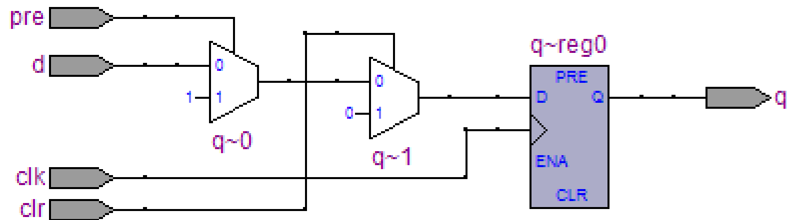
- Sono segnali condivisi da molti elementi
  - Devono arrivare in molti punti diversi della FPGA
  - Considerevoli ritardi nei collegamenti
  - Difficile mantenere il sincronismo
  - Sorgente (driver) e destinazione di un segnale è bene siano vicini
- Duplicazione della logica
  - Manuale o automatica (max_fanout)
  - “preserve attribute” per evitare che l’ottimizzazione elimini le ridondanze
- Utilizzo di reti dedicate (ad alto fanout – es: Reset)

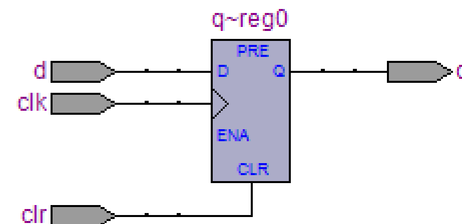


RESET sincrono ed asincrono

```
module sync (d,clk,clr,pre,q);
input d,clk,clr,pre;
output q;
reg q;

always @(posedge clk)
   begin
      if (clr)
          q <= 1'b0;
      else if (pre)
          q <= 1'b1;
      else
          q <= d;
   end
endmodule
```

```
module async (d,clk,clr,q);
input d,clk,clr;
output q;
reg q;

always @(posedge clk or posedge clr)
   begin
      if (clr)
          q <= 1'b0;
      else
          q <= d;
   end
endmodule
```

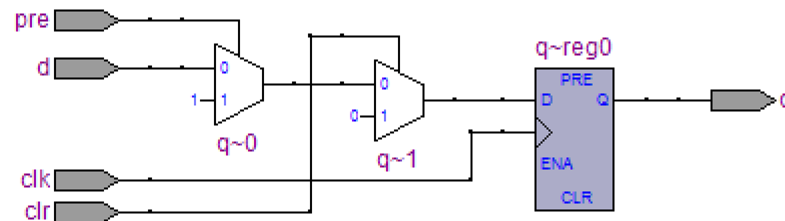




ALTERA

Gestione del Reset

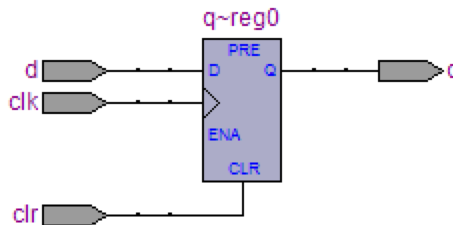
- Reset sincrono
  - Sconsigliato
  - Utilizza maggiori risorse
  - Non fa uso di risorse dedicate
  - E’ trattato alla stregua di qualunque segnale
  - Ha elevato fan-out



ALTERA

Gestione del Reset

- Reset asincrono
  - E’ più diffuso
  - Usa risorse già presenti su FPGA
  - Esistono anche linee globali di reset
  - Può presentare dei malfunzionamenti al momento di “ri-attivazione”
    - Essendo un seganle asincrono potrebbe arrivare nei pressi del fronte di clock:
    - Alcuni FF potrebbero riattivarsi subito, altri all’istante successivo

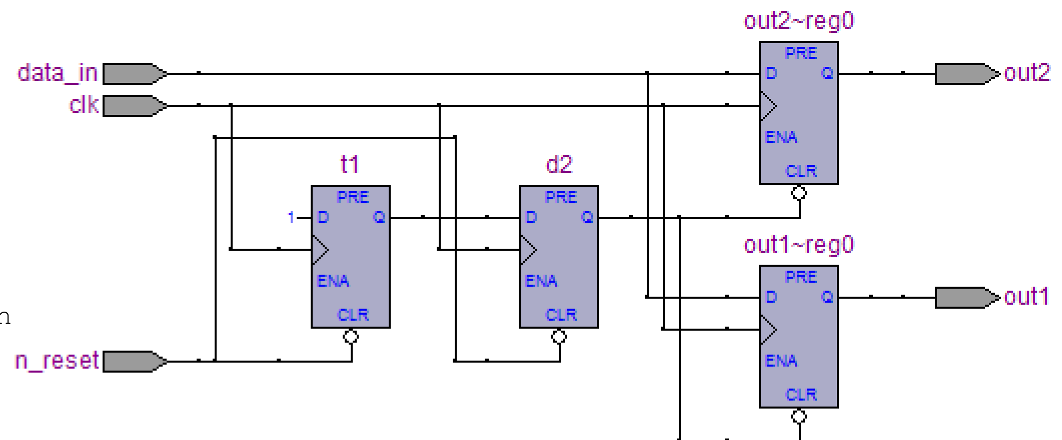


ALTERA

Gestione del Reset

- Reset asincrono / sincronizzato

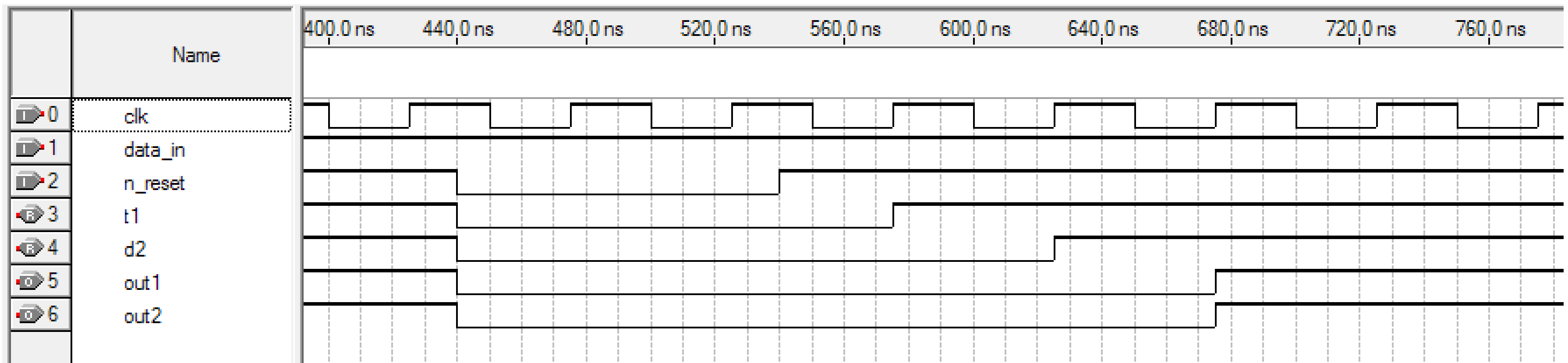
```
module reset_sinc_asinc(clk, n_reset, data_in, out1, out2);
input n_reset;
input clk;
input data_in;
output out1,out2;
reg out1,out2,t1,d2;

always@(posedge clk or negedge n_reset) begin
if (!n_reset) begin
   t1 <= 0;
   d2 <= 0;
end else begin
           t1 <= 1;
           d2 <= t1;
   end
end


always@(posedge clk or negedge d2) begin
if (!d2) begin
   out1 <= 0;
   out2 <= 0;
end else begin
           out1 <= data_in;
           out2 <= data_in;
   end
end

endmodule
```



ALTERA

Reset asincrono – sincronizzato

- Simulazione



- Conviene usare una struttura simile per pilotare i FF di ciascun "clock domain"

ALTERA

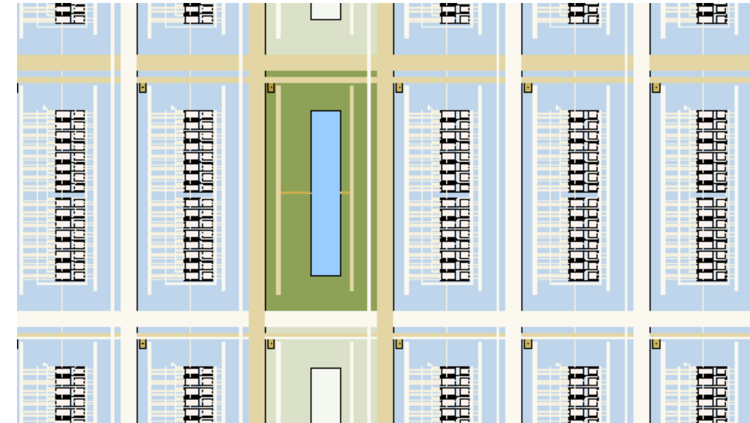
Inferring di memorie

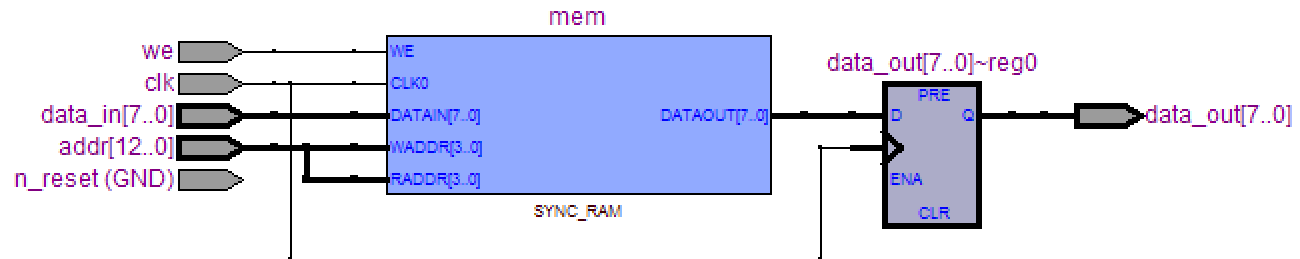
- Il sintetizzatore riconosce le memorie
- Se il codice coincide con la tipologia di memorie disponibili, questa viene “implicata”
- Altrimenti utilizza i singoli LE
  - Spreco di risorse
  - Elevato tempo di compilazione
  - Risultati scadenti (tempi, area, potenza)
- Esempi
  - le memorie sono sincrone
  - non prevedono linee di reset
  - Diverse metodologie di “read after writing”

ALTERA

Inferring di Memorie

```
module memory(
input n_reset,
input we,
input clk,
input [7:0] data_in,
input [12:0] addr,
output reg [7:0] data_out);

reg [7:0] mem [12:0];

always@(posedge clk) begin
if (we) mem[addr] <= data_in;
data_out <= mem [addr];
end


endmodule
```

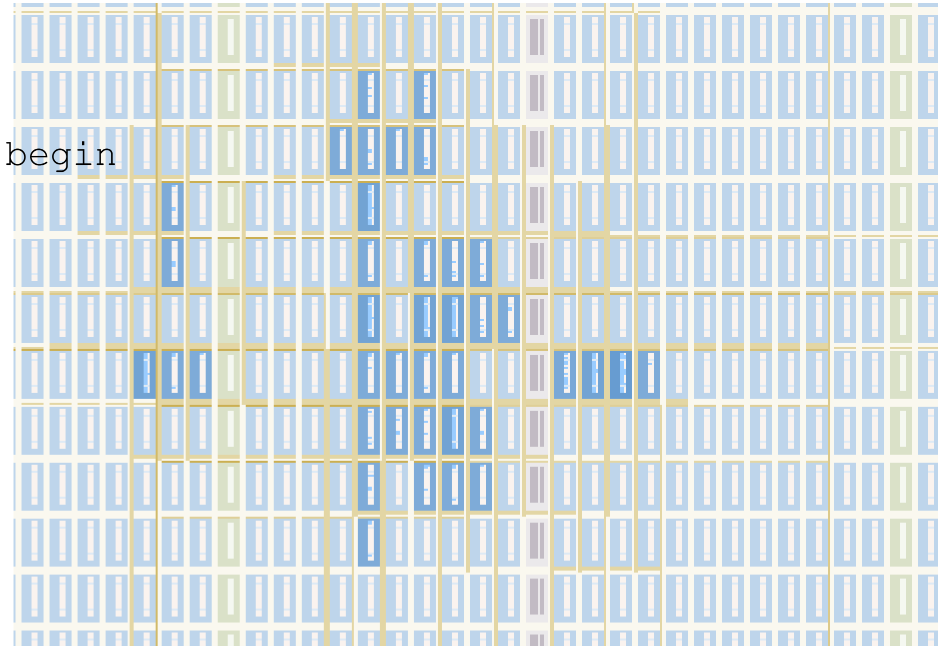


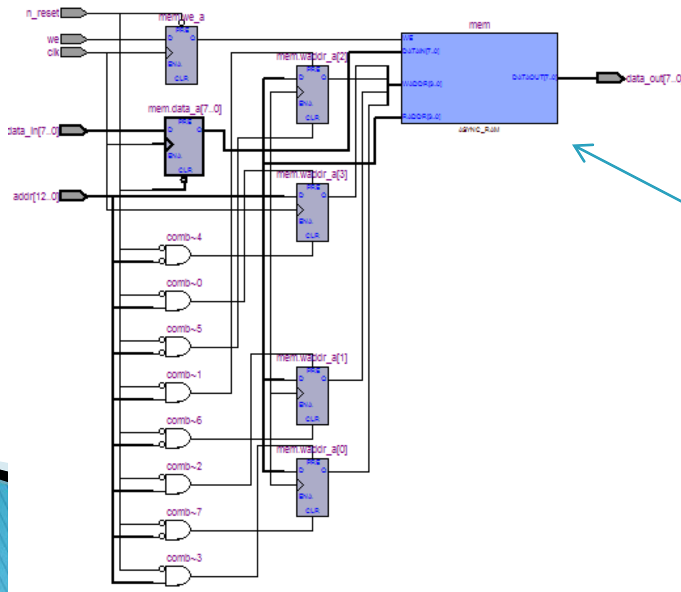


ALTERA

Inferring di Memorie

```
…
reg [7:0] mem [12:0];

always@(posedge clk or negedge n_reset) begin
if (!n_reset) mem[addr] <=0;
else if (we) mem[addr] <= data_in;
data_out <= mem [addr];
end


endmodule
```





Asinc. Mem

ALTERA

Risorse disponibili

- FPGA diverse possono essere dotate di risorse diverse
  - Le memorie tipicamente sono sincrone
  - Reset o altri controlli possono non essere supportati
  - La lettura contemporanea alla scrittura potrebbe essere di diverse tipologie (alcune supportate altre no)

ALTERA