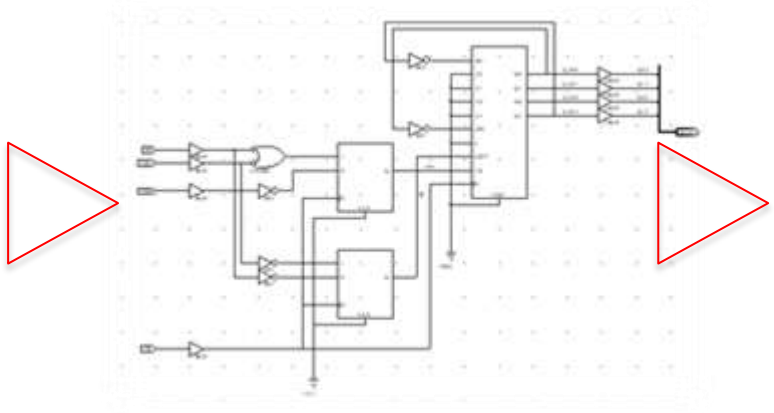
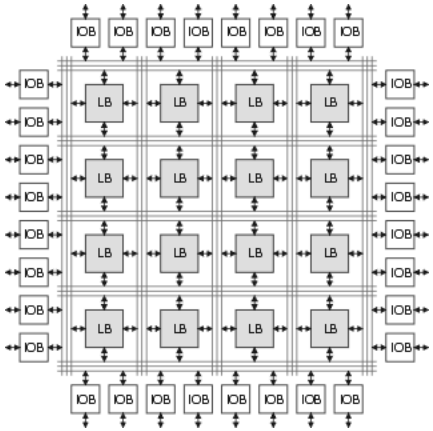
ALMA MATER STUDIORUM – UNIVERSITY OF BOLOGNA
DEIS - DEPARTMENT OF ELECTRONICS, COMPUTER ENGINEERING AND SYSTEMS

```
begin
  if (RESET_N = '0') then

    for col in 0 to BOARD_COLUMNS-1 loop
    for row in 0 to BOARD_ROWS-1 loop
    ...
   elsif (rising_edge(CLOCK)) then
    ...
```





Laboratorio di Sistemi Digitali M
A.A. 2011/12

1 – Introduzione a FPGA, principi di design e richiami di VHDL

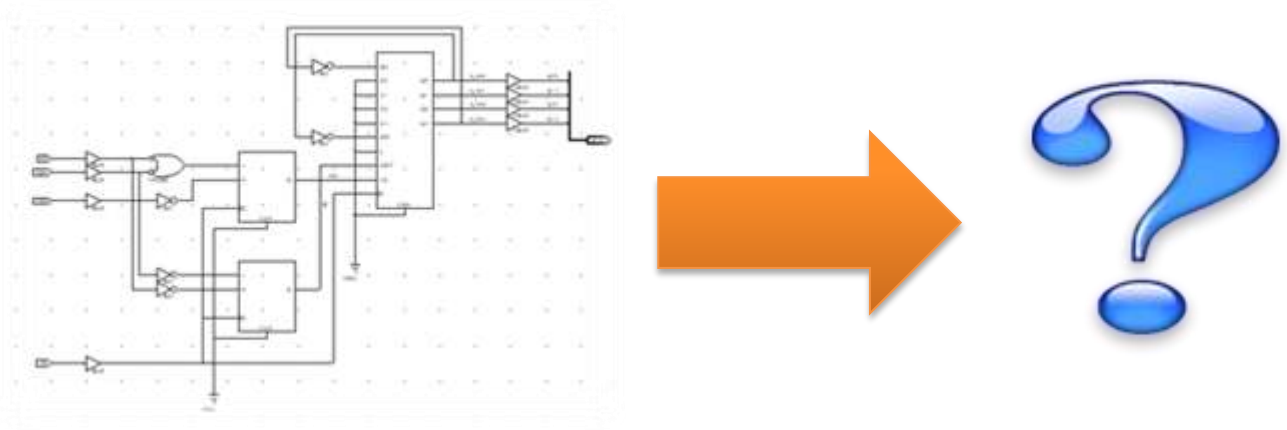
Primiano Tucci
primiano.tucci@unibo.it

www.primianotucci.com

Agenda

- **Introduzione a FPGA**
- Principi di design
- Progettazione in VHDL

Sistemi Digitali: Come, Dove e Perché?

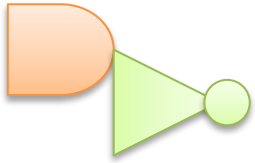


- Perché i Sistemi Digitali?
- Dove si usano?
- Come si realizzano?

Una panoramica del mondo embedded

Soluzioni PC-Based

Hardware G.P.
(e.g., PIC Microchip, STM ST6/ST7, Atmel, Arduino)



Sistemi Digitali

Costi			
Complessità realizzativa			
Prestazioni			
Consumi			

Complessità vs. Affidabilità

Non sempre la tecnologia più complessa è la più affidabile!

In quali contesti si usano?

Tipicamente integrano funzionalità specifiche per:

- Elaborazione di segnali digitali

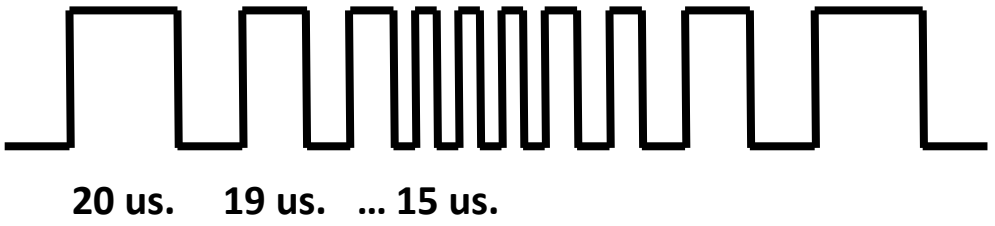
  Es: Moduli baseband per TLC, GPS, DSP audio / video

- Componenti digitali special-purpose

  Es: Design CPU/microcontrollori , controller ATA/USB, acceleratori crittografici

- Sistemi caratterizzati da vincoli temporali particolarmente stringenti

  Es: Tracking assi elettrici, generazione di ultrasuoni



Architetture di riferimento

Descrizione Hardware
(Schema a blocchi / VHDL / ...)

Verifica e simulazione

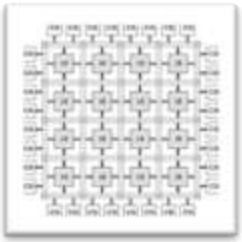
ASIC

Application Specific Integrated Circuit

- Prodotti tramite processi litografici
- Altissimo costo iniziale (NRE ≈ 500K $)
- Bassissimo costo per unità (≈ qualche $)
- Minori consumi e area
- Alto *time to market* (≈ 6 mesi)

Vie di mezzo
(e.g. ASIC semi-custom)

FPGA

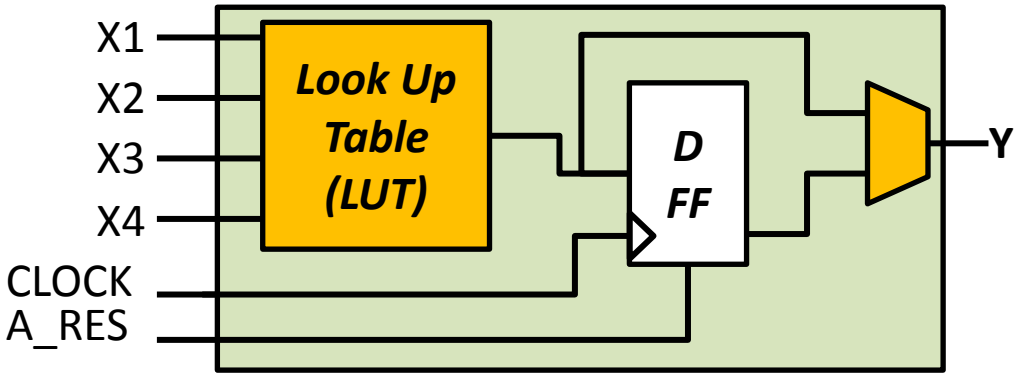


Field Programmable Gate Array

- Hardware riconfigurabile
- Investimento iniziale praticamente nullo
- Costo per unità (50-100$)
- Immediatamente operativi
- Maggiori consumi ed area
- f_{MAX} minore (50-400 Mhz)

Architettura di un FPGA

Logic cell



Modello generale di riferimento

I dettagli delle celle logiche in realtà variano in base al produttore ed al modello dell'FPGA.

Look-Up Table

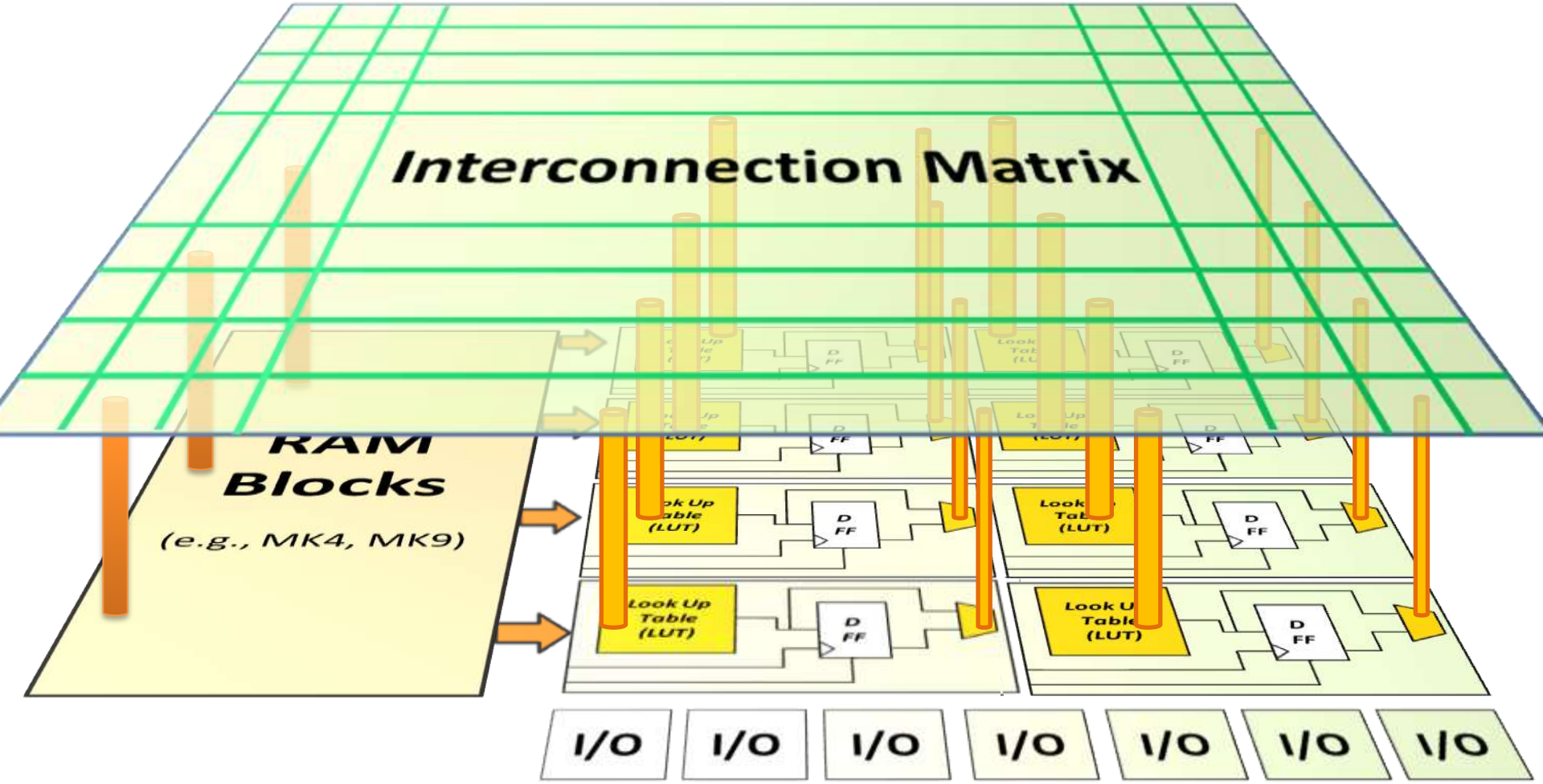
X1	X2	X3	X4	Y
0	0	0	1	?
0	0	1	0	?
…	…	…	…	?
1	1	1	1	?

Tipicamente ogni cella comprende:

- **Una look-up table:** che consente di mappare una qualsiasi funzione combinatoria 4 ingressi 1 uscita
- **Un FF di tipo D** (con set e clr asincroni)
- **Un mux 2 -> 1:** per bypassare il FF in caso di celle puramente combinatorie

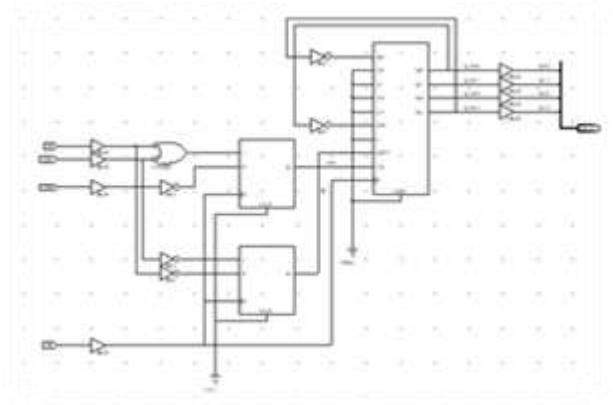
Elemento programmabile

Architettura di un FPGA



Come - Metodologie di progettazione

Schemi a blocchi



PRO

- Semplicità e rapidità di sviluppo
- Componenti legacy (e.g., 74xx)

CONTRO

- Formato file e comp. NON standard
- Scarsa manutenibilità
- Prettamente per sistemi semplici

Linguaggi di descriz. HW

VHDL, Verilog, SystemC

```
begin
  if (RESET_N = '0') then

    for col in 0 to BOARD_COLUMNS-1 loop
    ...
   elsif (rising_edge(CLOCK)) then
```

PRO

- Linguaggi standard (!)
- Flessibilità e manutenibilità
- Gestibilità di design complessi

CONTRO

- Modello computazionale

Modelli computazionali (1/2)

Modello sincrono bloccante

- Il codice consiste in una sequenza di istruzioni che possono durare indefinitamente.
- Esse possono sospendersi per un intervallo di tempo o in attesa di un evento esterno.
- Lo "stato" del programma è implicito, ovvero consiste nel trovarsi in una certa riga anziché un'altra.
- Tutto ciò è possibile solo se c'è una infrastruttura sottostante (es: il sistema operativo, una JVM, …) che gestisce per noi la "temporalità" delle operazioni.

Esempio:

```
uscita = 1;
while(ingresso == 0) {}
uscita = 0;
usleep(100);
...
```

Vantaggi

- La sequenza di operazioni da compiere è codificata e visibile direttamente nella sequenza di istruzioni.

Svantaggi

- Molto difficile gestire attività concorrenti (soprattutto se devono interagire tra loro)

Modelli computazionali (2/2)

Modello asincrono non bloccante

- Il programma consiste in blocchi di istruzioni "immediate".
- Lo stato deve essere mantenuto esplicitamente.

Esempio

```
switch(state) {
  case INIT:
    uscita = 1;
    state = WAIT_INGRESSO;

  case WAIT_INGRESSO:
    if(ingresso != 0) {
        uscita = 0;
        start_timer(100);
        state = ATTESA;
       }

  case ATTESA:
       if(timer_expired())
                  …
```

Vantaggi

- L'applicazione ha il pieno controllo dello stato, nessuna operazione "trattiene" il controllo.
- E' molto più agevole gestire operazioni concorrenti.

Questo modello computazionale è (quasi) sempre possibile su tutte le architetture, basta avere le API equivalenti non bloccanti (e.g., socket e I/O con O_NONBLOCK).

In alcuni contesti (**design hardware**, PLC, …) esso è **l'unico** modello ammesso.

VHDL per sintesi

- Il modello computazionale dei linguaggi di descrizione hardware (VHDL, Verilog, …) è profondamente diverso dai linguaggi di programmazione tradizionali.
- Nei linguaggi di programmazione (C, C#, Java …) gli statement del linguaggio definiscono istruzioni, che vengono eseguite sequenzialmente da una infrastruttura (dalla CPU nel caso di C, per mezzo di una virtual machine nel caso di Java …)
- Nei linguaggi di descrizione dell'hardware gli statement del linguaggio, invece, definiscono blocchi di hardware.
- Non c'è nessuna esecuzione sequenziale, nessuna infrastruttura sottostante, nessun run-time.

Sintesi logica

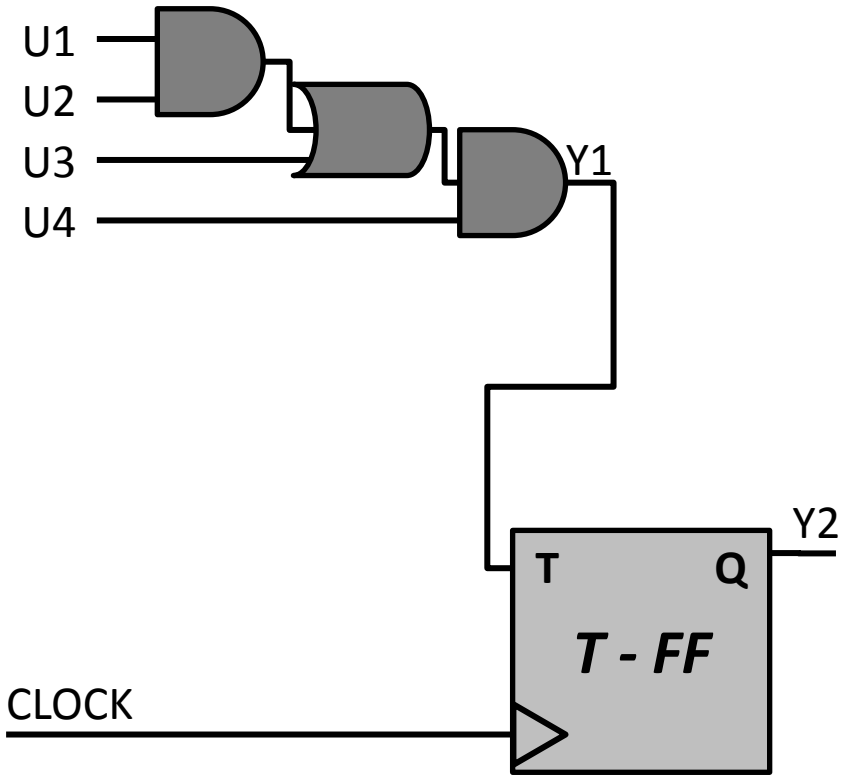
"The process of deriving **efficient results** from **clear specifications**"

- Il processo di sintesi trasforma una descrizione HDL in una *netlist* di gate elementari.
- La sintesi è applicabile ad un sub-set del linguaggio (VHDL sintetizzabile)
- La descrizione avviene attenendosi a *template* che vengono analizzati e riconosciuti dai sintetizzatori e danno luogo ai componenti logici corrispondenti.
- Il risultato della sintesi dipende dal sintetizzatore e dalle librerie di mapping adoperate (forniti dal produttore in caso di FPGA).

Flusso di sviluppo: Analisi e sintesi

Sorgenti VHDL

```
Y1 <=((U1 and U2) or U3) and U4;

ff1 : process (CLOCK)
Begin
  if (rising_edge(CLOCK)) then
   if(Y1 = '1') then
     Y2 <= not(Y2)
   end if;
  end if;
end process;
```



Flusso di sviluppo: Analisi e sintesi

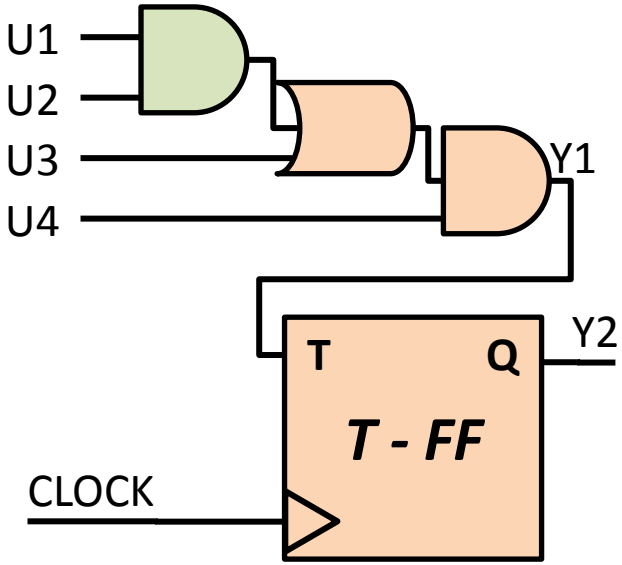
- Nella fase di Analisi e Sintesi il sintetizzatore analizza i costrutti del linguaggio (VHDL & c.), riconosce i template utilizzati e deriva i componenti di alto livello (contatori, multiplexer, decoder …).
  Sostanzialmente: trasforma il testo in uno schema a blocchi

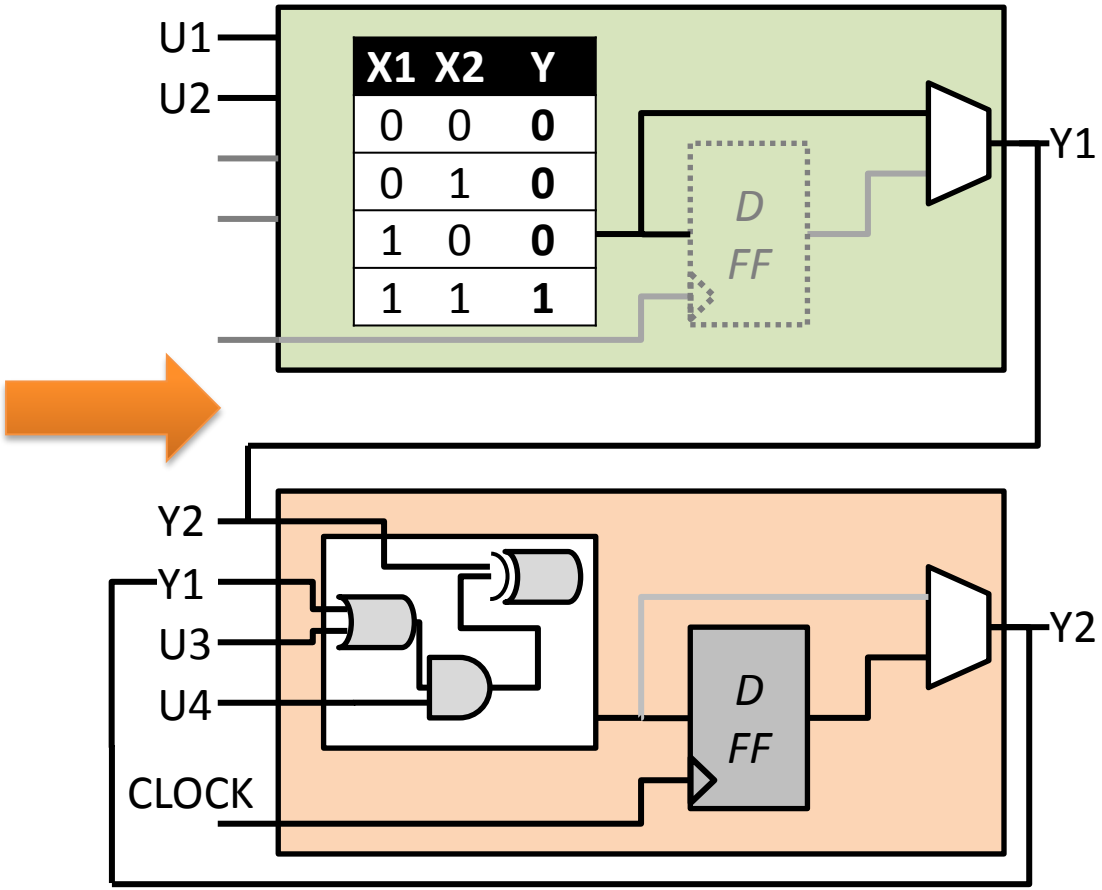
- Tale rappresentazione, però, non ha una corrispondenza diretta con l'hardware finale, ma solo "funzionale".

- L'hardware viene inferito solo nella fase di Mapping (o Fitting, Place and Route), in cui il sintetizzatore si avvale delle celle logiche dell'FPGA per mappare le funzionalità descritte su hardware reale.

- Il ruolo del progettista è di descrivere *cosa* va fatto, non *come.*
  Es: ~~*out <= not(not(not(not(a))));*~~ *(NON da luogo ad catena di invertitori)*

Flusso di sviluppo: Place and route (o Fitting, o Mapping)





I componenti di alto livello inferiti durante la A&S vengono mappati sull' hardware sfruttando le risorse dispnibili (celle logiche programmabili nel caso di FPGA).

Come funziona il processo di Mapping(1/3)

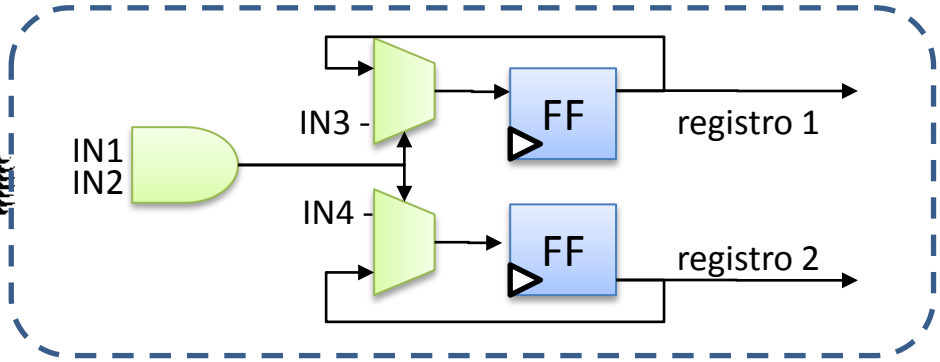
- Identificazione degli elementi di memoria: tutti i gli elementi di memoria di alto livello (registri/contatori/shift-register) vengono ricondotti ad elementari Flip-Flop.
- Identificazione delle funzioni di trasferimento (equazioni booleane) . Ovvero identificazione dei percorsi combinatori: (i) tra registri; (ii) tra input e registri; (iii) tra registri ed output; (iv) tra input ed output.
- Riduzione delle equazioni (ottimizzazione).
- Mapping delle equazioni corrispondenti sulle Look-Up Table delle celle logiche.
- Tutto questo vale non solo per VHDL ma anche per gli schemi a blocchi. Quello che si “disegna”, infatti, non riflette l’hardware finale, ma solo la funzionalità che si desidera modellare.

Come funziona il processo di Mapping (2/3)

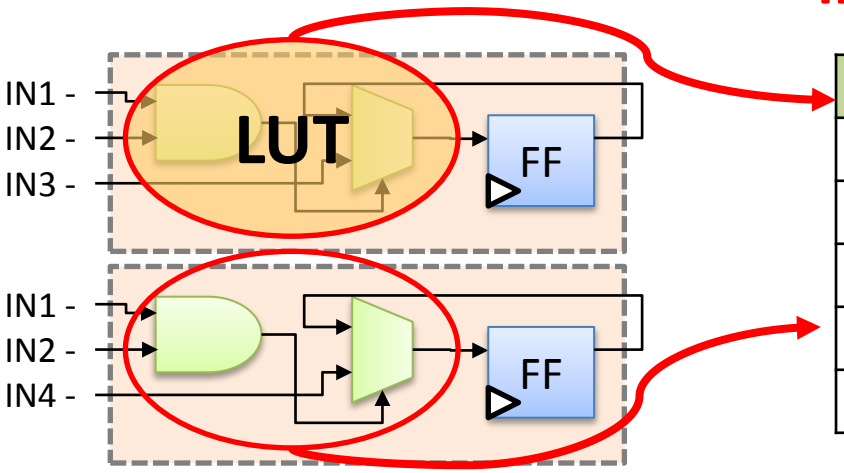
Nota: il processo di mapping può talora "duplicare" delle funzioni combinatorie, allo scopo di ridurre i tempi di elaborazione e favorire la F_{MAX}.

Vediamo un esempio:

```
if rising_edge(CLOCK) then
 if (IN1='1' and IN2='1') then
  registro1 <= IN3;
  registro2 <= IN4;
 end if;
end if;
```

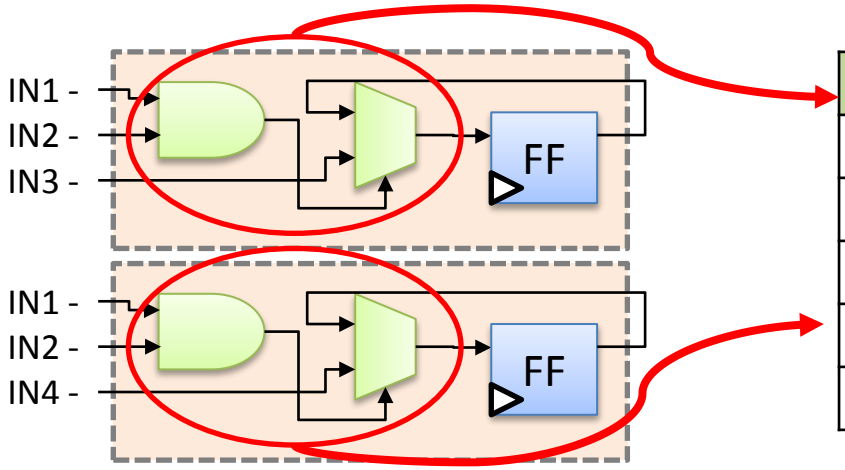


In realtà …



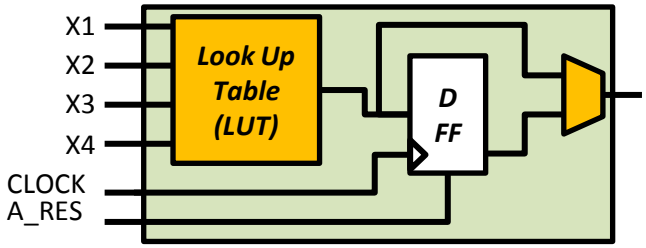
IN1	IN2	IN3	Y	Y_{FUT}
0	X	X	**1**	**1**
X	0	X	**1**	**1**
…	…	…	…	…
1	1	**1**	X	**1**
1	1	**0**	X	**0**

Come funziona il processo di Mapping (3/3)



IN1	IN2	IN3	reg	OUT
0	X	X	**1**	**1**
X	0	X	**1**	**1**
…	…	…	**…**	**…**
1	1	**1**	X	**1**
1	1	**0**	X	**0**

Come mai? Ogni LUT ha un costo fisso (sia in termini di area utilizzata che in termini di tempo di propagazione) a prescindere da "quanto" sia utilizzata. Una sequenza di due celle in cascata, se pure più intuitiva come soluzione, avrebbe rappresentato un maggiore costo, sia intermini di area (sarebbero servite 3 celle) che temporale (2xTP)



Morale: *non ha senso ottimizzare a mano la logica combinatoria, il sintetizzatore sa farlo meglio di noi. Tuttavia non è escluso che si possa realizzare uno stesso componente in due modi funzionalmente diversi, e con prestazioni profondamente diverse.*

Agenda

- Introduzione a FPGA

  Riassumendo:

  – Il progettista esprime attraverso il design (sia schema a blocchi che VHDL) le funzionalità desiderate, ma non ha controllo fine-grained sull'hardware finale (in particolare la logica combinatoria) che viene generato, per via del processo di mapping e delle ottimizzazioni

  – Alla fine delle fasi di Analisi&Sintesi e Mapping il sintetizzatore produce una netlist, "funzionalmente equivalente" a quanto che abbiamo progettato, contenente:

    - I registri che abbiamo previsto esplicitamente (disegnandoli con schema a blocchi) o implicitamente (adottando template sequenziali in VHDL)
    - Logica combinatoria funzionalmente equivalente a quella che abbiamo disegnato (con schema a blocchi) o determinato tramite statement VHDL.

  – Pertanto è necessario adottare uno "stile" di progettazione che non risenta di questo.

- **Principi di design**
- Progettazione in VHDL

Design sincrono

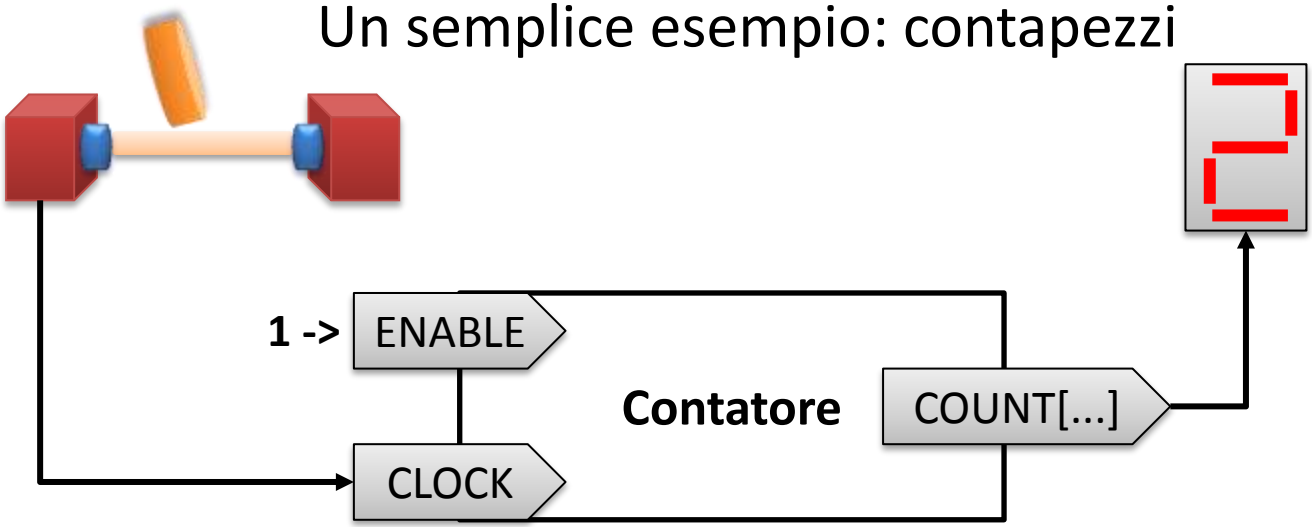
Poche, semplici ma rigorose, regole

- Tutta la logica è sincronizzata su un unico segnale di CLOCK (ovvero, tutti i template sincroni usano **SOLO** rising_edge(**CLOCK**))
- Di conseguenza, tutti i segnali (interni) vengono prodotti sui fronti di salita del clock, ed al più subiscono un ritardo per via della logica combinatoria presente tra un registro ed un altro.
- I segnali provenienti dall'esterno dell'FPGA (es: reset, pulsanti di ingresso, …) vanno sincronizzati e portati nel proprio *dominio di clock*. (vedremo come)

Perché tutto questo?

Svantaggi dei design asincroni (1/4)

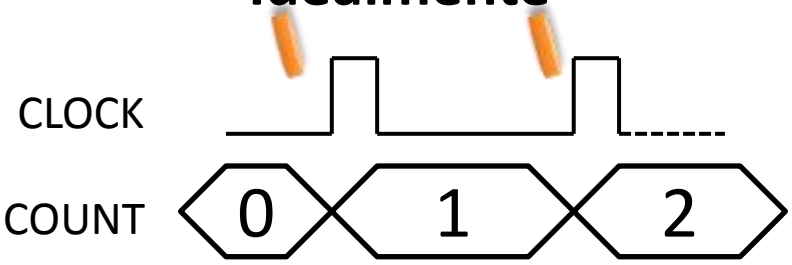
Un semplice esempio: contapezzi



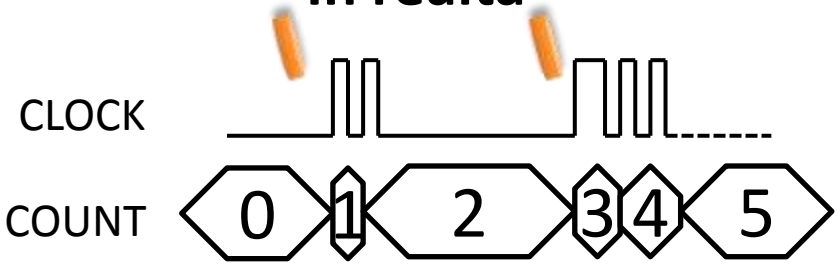
1° Problema:

Glitch di segnali non sincroni.

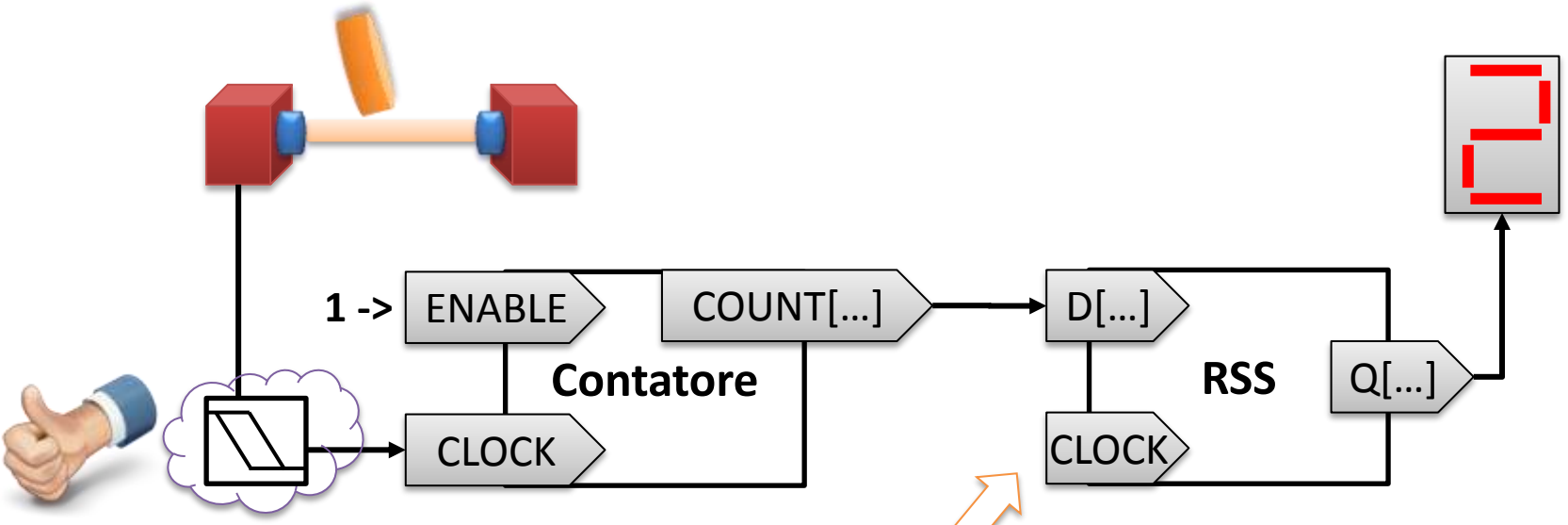
Idealmente

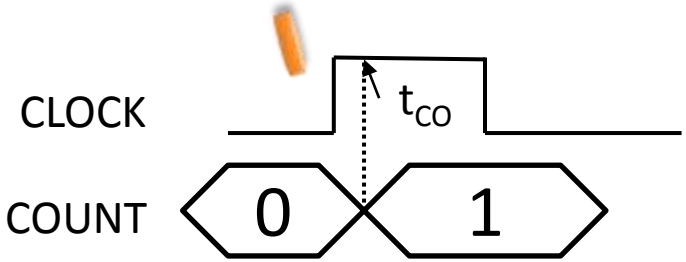


In realtà



Svantaggi dei design asincroni (2/4)

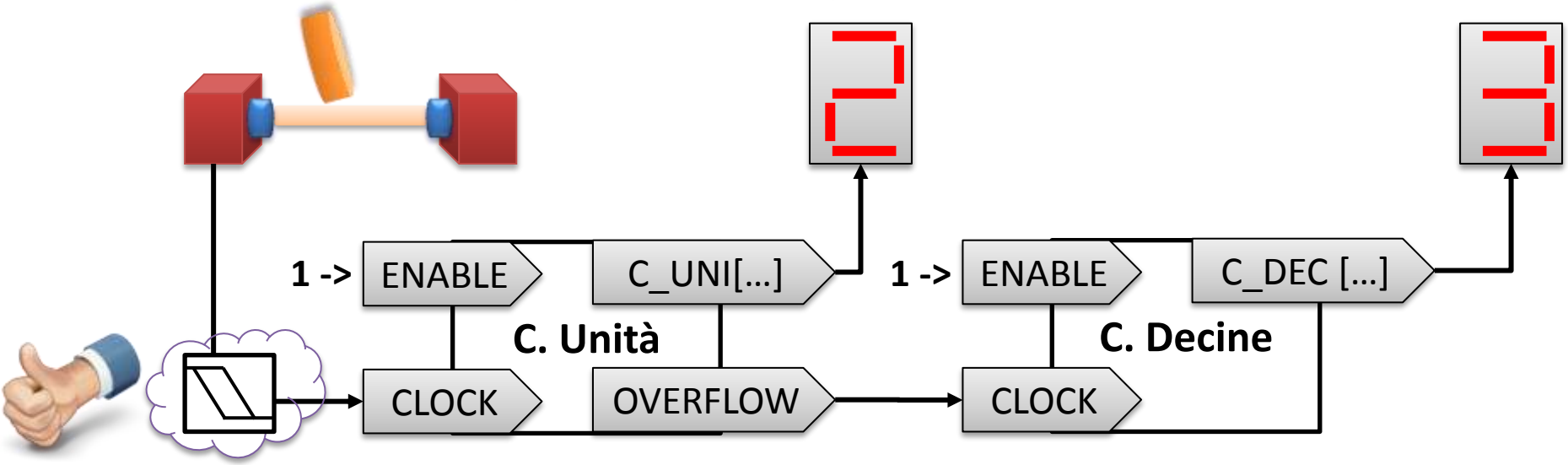


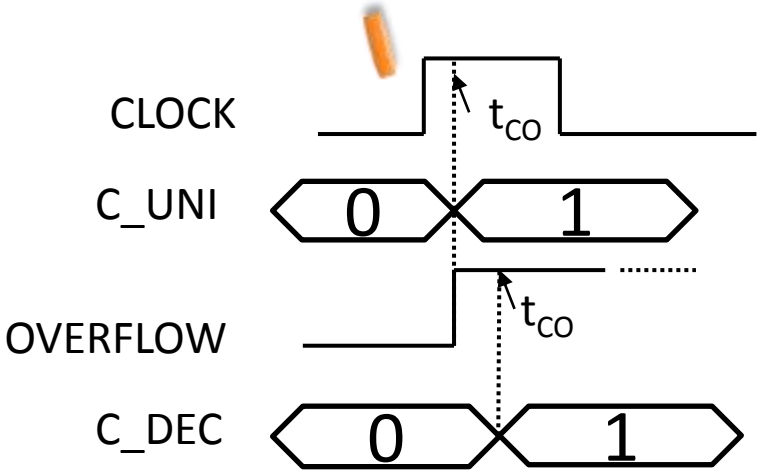


?

2° Problema:
Sincronizzazione con altre reti sequenziali.

Svantaggi dei design asincroni (3/4)





3° Problema:

E' estremamente difficile verificare il rispetto dei vincoli temporali

Svantaggi dei design asincroni (4/4)

4° Problema:

E' sempre possibile minimizzare una qualsiasi funzione combinatoria come una rete a due livelli.

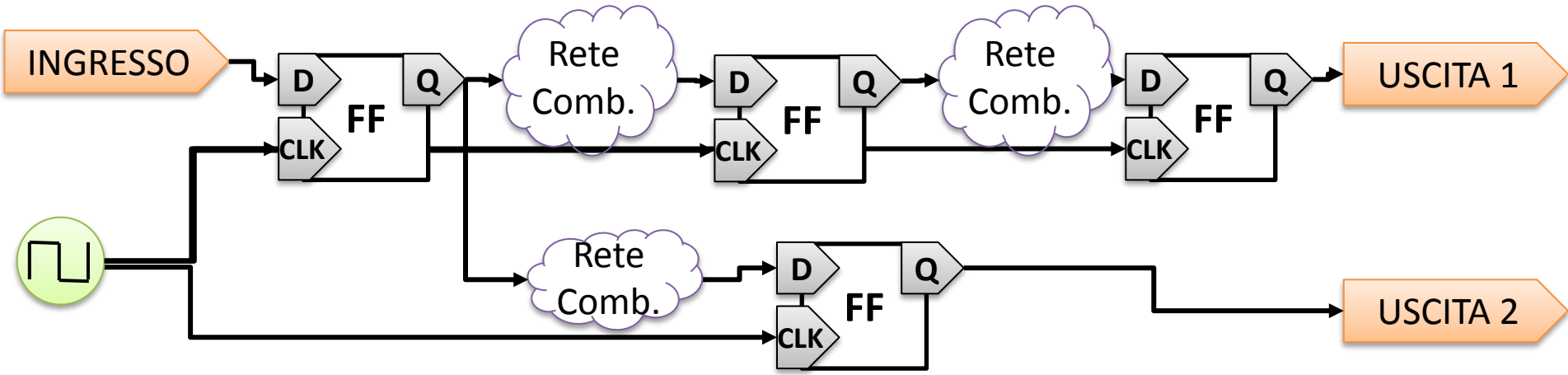
Il processo di fitting (place and route) fa proprio questo : la logica combinatoria viene "mappata" sulle look-up-table delle celle logiche.

Tuttavia la rete che ne deriva si comporta in modo identico solo "a regime", ma può comportarsi differentemente durante i transitori.

Se il design è **a**sincrono (ovvero i FF catturano proprio in occasione i transitori), hanno luogo funzionamenti inaspettati. E non c'è (quasi) niente che possiate fare a riguardo, almeno per quanto riguarda gli FPGA.

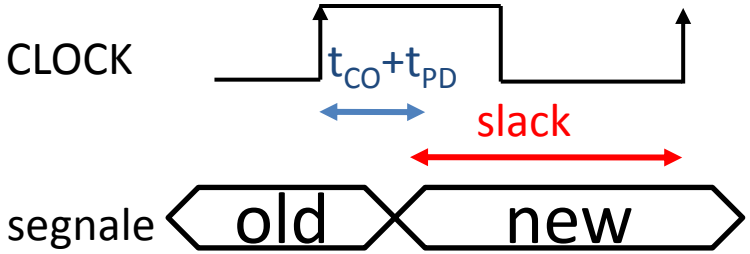
Questo è il motivo per cui spesso si hanno seri problemi sviluppando su FPGA tramite schemi a blocchi. Gli schemi a blocchi danno l'illusione di poter realizzare design asincroni utilizzando logica combinatoria arbitraria.

Design sincrono



Il design sincrono consiste, sostanzialmente, in due grandi promesse che si fanno a se stessi (ed agli altri):

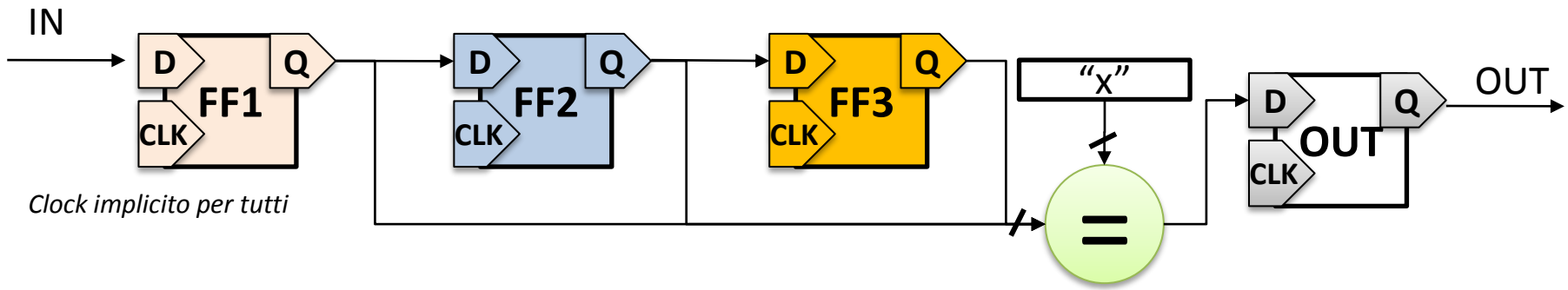
1. Lo stato del sistema evolve (leggi: i FF campionano) esclusivamente sul fronte di un unico clock.
2. Tutti i segnali evolvono esclusivamente nell'immediata successione del fronte del clock. Quindi, gli eventuali transitori imprevisti si esauriscono dopo il fronte del clock, ed i segnali hanno tempo per stabilizzarsi (slack) entro il fronte successivo. Al più l'evoluzione di un segnale può essere ulteriormente ritardata a causa di una rete combinatoria, più o meno complessa.

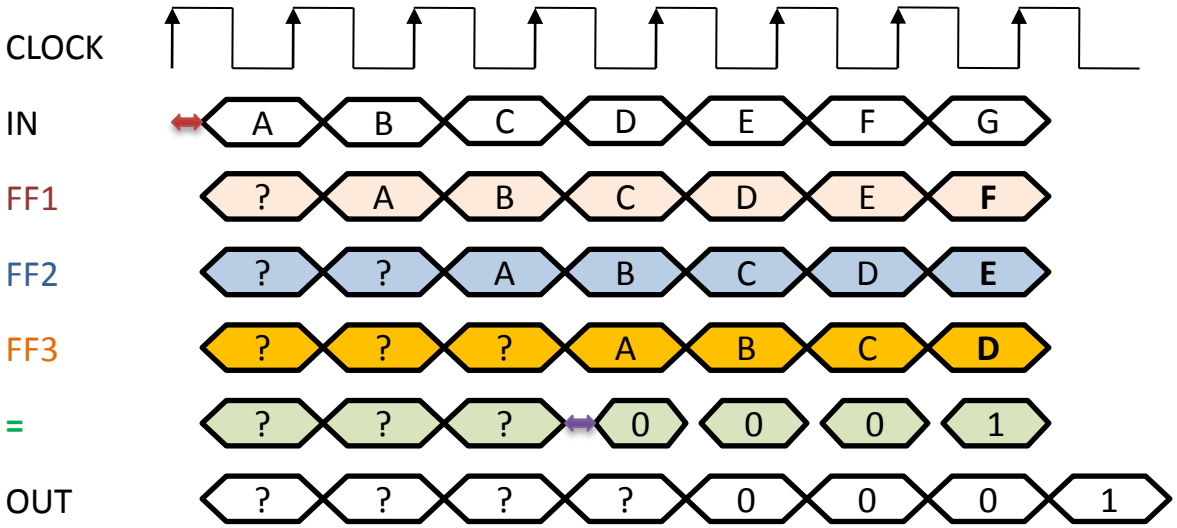


In questo modo è facile* determinare la frequenza massima di clock tollerabile dal sistema (f_{MAX}). Il clock può essere tollerato fino a che tutti i percorsi hanno slack > 0.

** In realtà subentrano altri fattori: e,g, anche il clock può subire ritardi.*

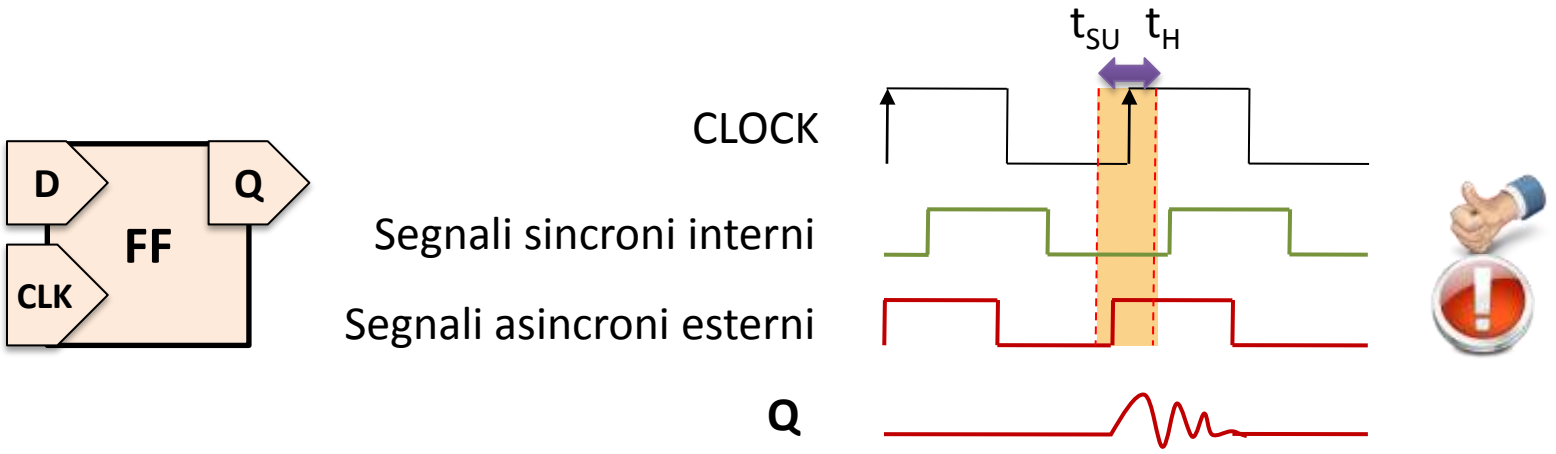
Un esempio: ricezione seriale con controllo word





- Tutti i segnali si producono, rispetto al fronte del clock, dopo (almeno) un tempo T_{CO} (T clock to out). Detto in altre parole, in un design sincrono, nessuno può "vedere" il nuovo dato prima del fronte successivo del clock.
- La logica combinatoria tra un registro ed un altro può ulteriormente ritardare la stabilità di un segnale. Ma in fondo c'è tempo fino al fronte successivo. Starà al tool di analisi temporale verificare che tali ritardi siano compatibili con la F. clock.

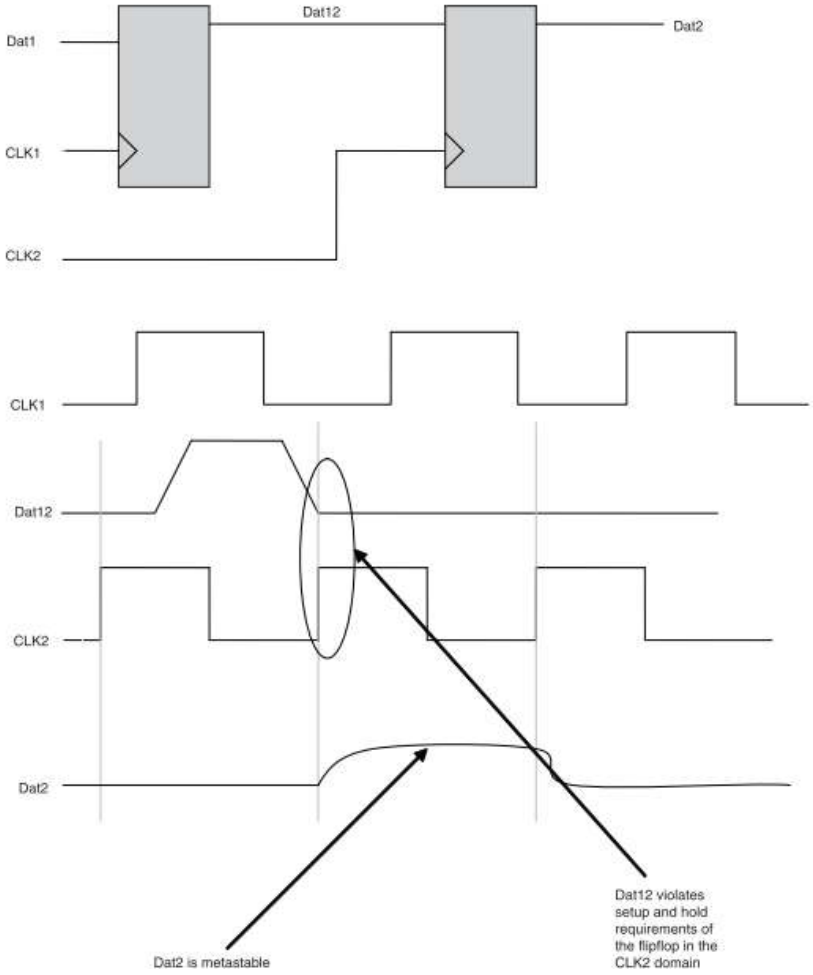
Meta-stabilità : il problema



- Anche nel caso degli FPGA gli elementi di memoria sincroni (flip-flop) hanno vincoli sulla stabilità degli ingressi nell'intorno del fronte del clock (tempi di setup e hold)
- Qualora questi vincoli non siano rispettati, il FF interessato può entrare in uno stato di meta-stabilità. Gli elementi a valle del FF possono, pertanto, vedere un livello instabile per tutta la durata della meta-stabilità (tipicamente 1-2 periodi di clock).
- La metodologia di progettazione sincrona ci evita questi problemi (se la f_{CLK} è compatibile) per i segnali interni all'FPGA (nell'ambito di un dominio di clock)
- **Ma i segnali provenienti dall'esterno?**

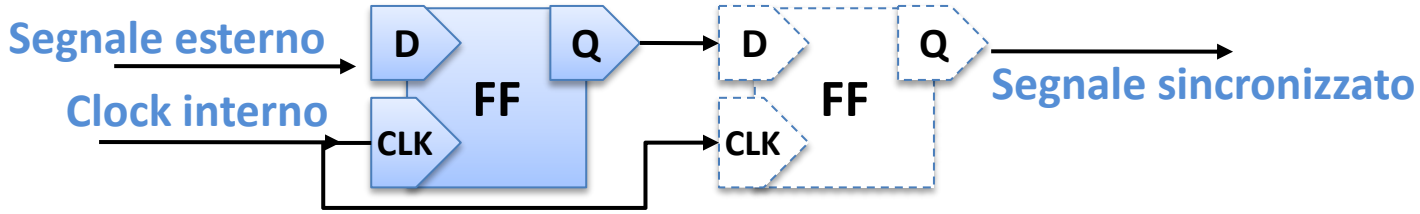
Metastabilità (Da Altera AN 545: Hardcopy Design Guidelines)

Figure 1. Synchronization Failure for a Signal Crossing Clock Domains



Meta-stabilità : come affrontarla

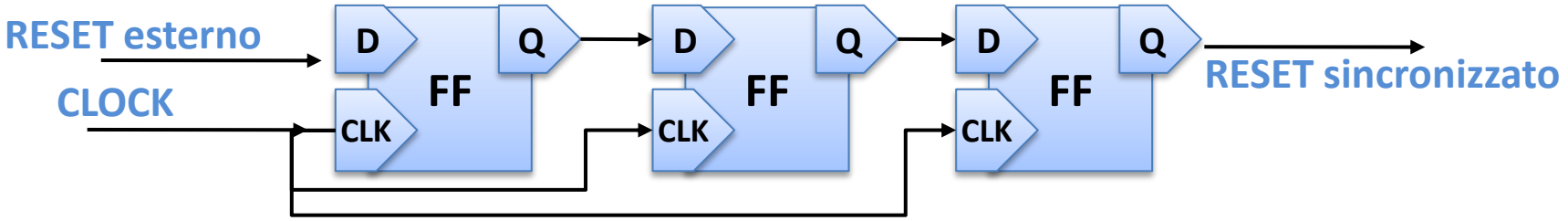
- I fenomeni di meta-stabilità vengono "arginati" introducendo sui segnali non correlati rispetto al clock una catena di FF di **sincronizzazione**.



- Il principio è semplice: al peggio la meta-stabilità viene percepita solo dagli *N* FF di sincronizzazione e non viene propagata (a meno che non duri più di *N* periodi di clock, e quindi si propaghi da un FF all'altro)
- Tipicamente 1 FF è sufficiente per la maggior parte dei segnali. Un numero maggiore di FF riduce la probabilità di meta-stabilità.
- Svantaggio: introduzione ritardo sugli ingressi.

Generazione e gestione dei segnali di RESET

- Anche il segnale di RESET_N deve essere sincronizzato (tipicamente consigliati 2-3 FF nei datasheet)
- In assenza di un segnale esterno esso può essere generato all'interno dell'FPGA sfruttando i power-up value dei FF.
- Nell'esempio sottostante si consideri i power-up value dei FF a '0', applicando sulla porta di reset esterno '1'.
- Il reset rimane asserito per 3 cicli al boot per poi rimanere de-asserito per tutto il funzionamento dell'FPGA



Agenda

- Introduzione a FPGA
- Principi di design

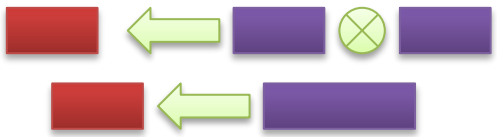
  Riassumendo:
  - I design sincroni rappresentano l'unica via agilmente percorribile sugli FPGA.
  - Regola molto semplice: tutti i registri campionano sullo stesso CLOCK.
  - Garantiscono l'insensibilità ai transitori.
  - Sono facilmente verificabili (temporalmente).
  - Sono più facili da progettare, una volta entrati nel giusto modo di pensare.
- Progettazione in VHDL

Livelli di astrazione nel design hardware

Come per quasi tutti i linguaggi, anche nel caso di VHDL le specifiche non dicono *come* va usato. Per quanto riguarda il design di hardware esistono, tuttavia, tre approcci di riferimento, contraddistinti da diversi livelli di astrazione:

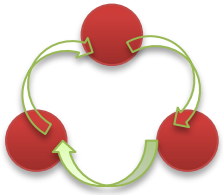
Dataflow

- Il sistema viene modellato analiticamente (attraverso assegnamenti concorrenti) enfatizzando il flusso dei dati.
- Si presta bene alla modellazione elementi statici (e.g. descrizione di un datapath)
- Si presta molto poco alla modellazione di componenti "interattivi" (e.g. macchine a stati)
- Poco ambiguo. Grande potere espressivo per descrivere *cosa* va fatto.



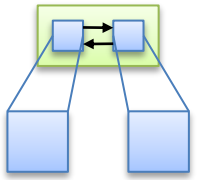
Behavioural

- Il sistema viene modellato sfruttando statement sequenziali (*process* VHDL) che enfatizzano il comportamento ed il flusso logico del sistema.
- Risulta in generale più "leggibile".
- Ricorda (purtroppo!) lo stile sequenziale con cui vengono generalmente pensati i software per PC.
- Fa molto leva sulla intelligenza del sintetizzatore . Spesso si perde il controllo (funziona ma non so cosa ha sintetizzato)



Structural

- Il sistema viene modellato come composizione di elementi primitivi.
- Il design finale risulta molto strutturato, ma a volte anche troppo (poco gestibile).
- NON portabile (molte librerie elementari sono proprietarie). Effetto lock-in!
- E' poco flessibile (un piccolo cambiamento dei requisiti richiede in generale cambiamenti sostanziali in molte zone del codice)
- Ricalca il modo di pensare tipico dell'hardware (praticamente si disegna uno schema a blocchi scrivendo codice)



Livelli di astrazione: un esempio

```
entity Full_Adder is
port(
     A, B, CARRY_IN :  in std_logic;
     SUM, CARRY_OUT :  out std_logic
     );
```

Dataflow

```
architecture Dataflow of Full_Adder is
begin

 SUM       <= A xor B xor CARRY_IN;


 CARRY_OUT <= (A and B)
           or ((A or B) and CARRY_IN));

end architecture;
```

Behavioural

```
architecture Behavioural of Full_Adder is
begin

 sum_gen: process(A,B,CARRY_IN) is
 begin

 if ((A='1' and B='1' and CARRY_IN='0')
     or ((A='1' or B='1') and
     CARRY_IN='1'))
  then
    SUM  <= '1';
  else
    SUM  <= '0';
  end if;
end process;
 -- omissis per CARRY_OUT
end architecture;
```

Structural

```
Architecture Structural  OF  Full_Adder is
 component  and2
   port (A,B : in std_logic;
         Z   : out std_logic);
 end component;
 component xor2 ...  -- omissis
 component or3  ...  -- omissis

 signal addt, c1, c2, c3 : std_logic;

begin
  G1: xor2 PORT MAP(A,B,addt);
  G2: xor2 PORT MAP(addt,CARRY_IN,SUM);
  G3: and2 PORT MAP(A,B,c1);
  G4: and2 PORT MAP(A,CARRY_IN,c2);
  G5: and2 PORT MAP(B,CARRY_IN,c3);
  G6: or3  PORT MAP(c1,c2,c3,CARRY_OUT);
end architecture;
```

Pertanto … come procedere?

Con consapevolezza e sensibilità.

Dataflow: Molto adatto per modellare datapath ed elementi combinatori semplici.

Behvioral: Praticamente obbligatorio per modellare Control Unit e logica sequenziale in generale. Molto adatto per modellare componenti combinatori "complessi".

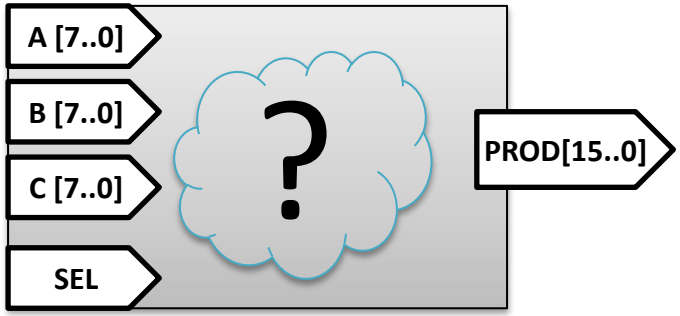
Structural: Molto adatto per descrizione top-level di un sistema come composizione di unità di alto livello.

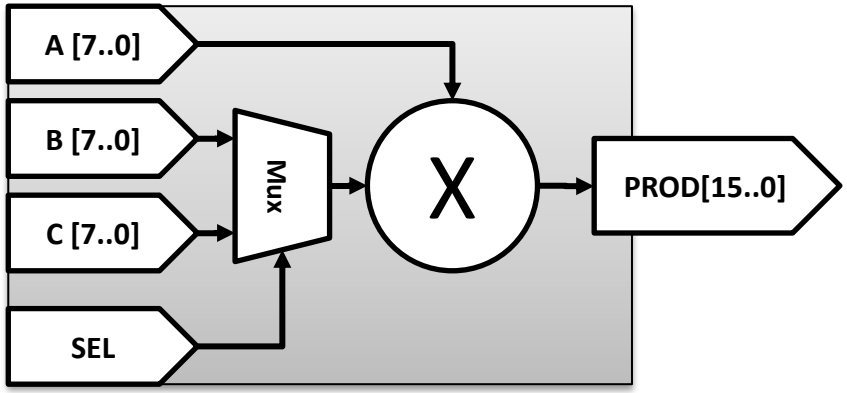
Linee guida

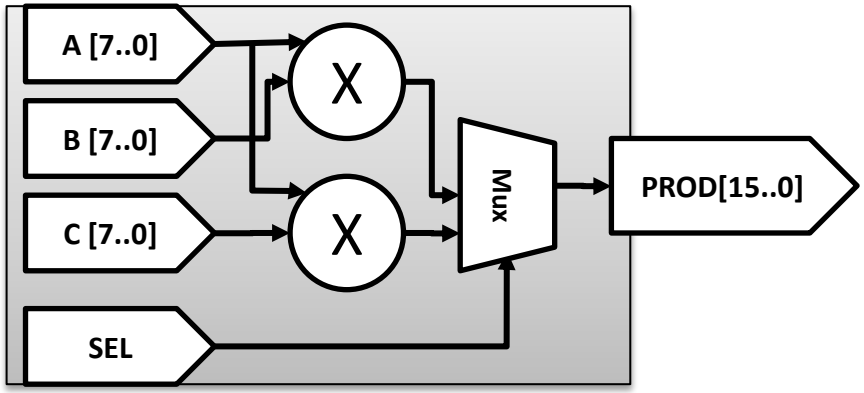
- **Divide et Impera**
  Decomporre (ma non frammentare) il sistema in unità elementari, facendo riferimento a modelli di interazione ben definiti ed evitando soluzioni "troppo creative".
- **Mantenere il controllo del design**
  Bisogna avere sempre ben chiaro in mente quello che sarà il risultato della sintesi (non scoprirlo a posteriori). Il codice è un mezzo per descrivere un componente nel modo più elegante possibile, non il risultato di una serie di tentativi.
- **Chiarezza e leggibilità**
  Ogni unità, ogni registro, ogni segnale devono avere un ruolo, e di conseguenza un nome, chiaro e ben preciso. Se un componente "funziona" ma non è ben chiaro come e perché, probabilmente c'è un modo più sensato per modellarlo!

Mantenere il controllo del design

```
if (SEL = '1') then
   PROD <= A * B;
else
   PROD <= A * C;
end if;
```



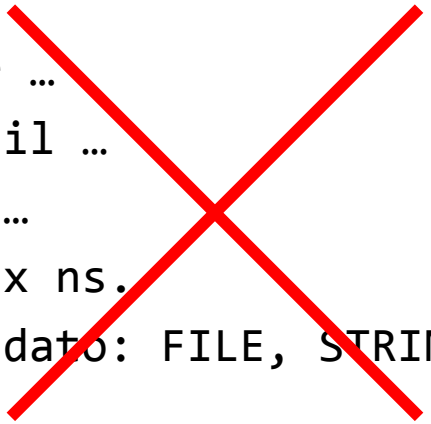




VHDL per sintesi / ~~VHDL per simulazione~~

In queste esercitazioni ci occuperemo esclusivamente di **descrizione hardware e di sintesi**, NON di simulazione. Se pur il linguaggio è sintatticamente identico, *VHDL per simulazione* assume un modello computazionale profondamente diverso.

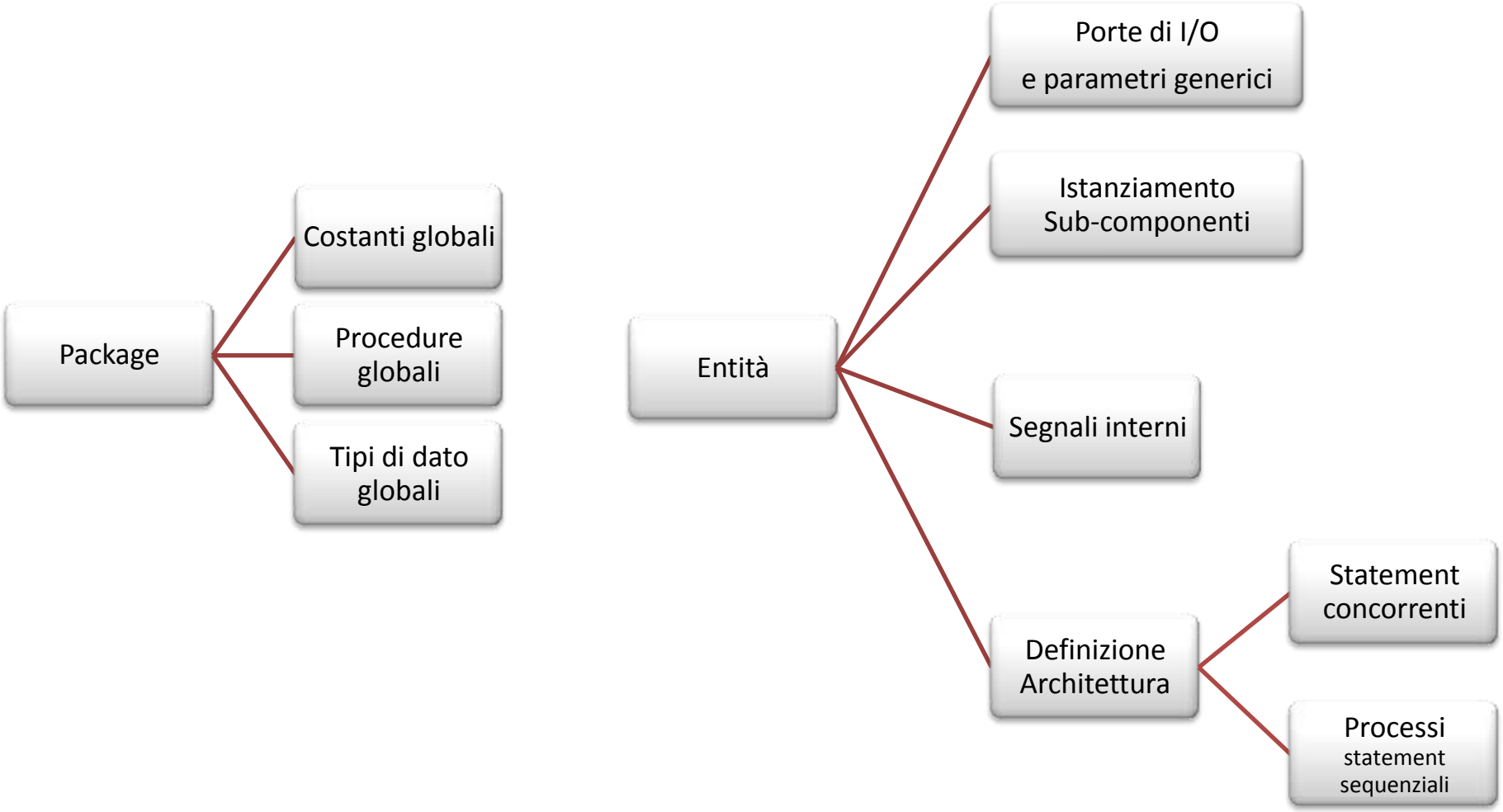
- `wait for …`
- `wait until …`
- `wait on …`
- `… after x ns.`
- `Tipi di dato: FILE, STRING,`



Un design orientato alla sintesi si può (quasi) sempre simulare, e la sua simulazione è molto attendibile.

Ma allora: perché esistono i linguaggi orientati alla simulazione?

Struttura di un progetto VHDL



Un po' di terminologia

- **Package:** una collezione, accessibile globalmente, di costanti, tipi di dati e funzioni. Pensatelo un po' come un namespace.
- **Entity:** unità elementare di design. Sostanzialmente, una *classe*.
- **Architecture:** ogni entity può avere differenti "implementazioni" (architecture). Es: una per la sintesi ed una per la simulazione. Chi istanzia la entity può scegliere a quale *architecture* fare riferimento (attraverso lo statement configuration). Noi non le utilizzeremo, ogni nostra entity avrà una sola architettura: RTL
- **Generic:** parametri di configurazione definibili durante l'istanziamento di una Entity. Concettualmente simili ai template di C++ o i generics di C#/Java
- **Driver:** sorgente di un segnale. Tipicamente un processo o uno statement concorrente.
- **Attribute:** informazioni addizionali per alcuni oggetti (segnali, porte, variabili). Es: PORT_X'length, signal'range
- **Concurrent statements:** istruzioni stand-alone, tipicamente adoperate per definire reti combinatorie semplici (e.g., signal_x <= PORT_A and not(signal_y))
- **Sequential statements:** istruzioni, utilizzabili solo all'interno di process e derivati (funzioni, procedure), che danno *l'illusione* di un programma sequenziale.

VHDL – Tipi di dato

- **Tipi base**

```
signal un_wire        : std_logic;
signal stringa_di_bit : std_logic_vector(31 downto 0);
signal counter        : integer range 0 to 10;
```

*Evitate bit e std_**U**logic*

- **Enumerativi**

```
type fsm_type is (IDLE, WAITING, BUSY);
type button_state_type is (PRESSED, RELEASED);
```

- **Subtype (typedef)**

```
Subtype counter_type is integer range 0 to 10;
```

- **Record (struct)**

```
type pixel_type is record
  x_coord     : integer range 0 to SCREEN_WIDTH-1;
  y_coord     : integer range 0 to SCREEN_HEIGHT-1;
  colour      : colour_type;
end record;
```

- **Array**

Dichirazione: `type pixel_array is array (natural range <>) of pixel_type;`
Uso: `signal pixels : pixel_array(0 to 100);`

VHDL è un linguaggio <u>strongly</u> typed. Rispettare i tipi ed effettuare casting espliciti ove necessario! (Vedi slide succ.)

std_logic_vector (What Your Mom never told you)

- **Attributi**

```
signal v : std_logic_vector(7 downto 0);
```

  - `v'range` : range del segale/porta (7 downto 0)
  - `v'reverse_range` : range specchiato del segale/porta (0 to 7)
  - `v'left` : parte sinistra del range (7)
  - `v'right` : parte destra del range (0)
  - `v'low` : parte bassa del range (0)
  - `v'high` : parte alta del range (7)
  - `v'length` : lunghezza (8)

- **Assegnamenti**
  - **Binario:** `v <= "11001010"; -- la dimensione deve essere esatta`
  - **Esadecimale:** `v <= X"CA"; -- equivale a "11001010"`
  - **Clear:** `v <= (others => '0'); -- equivale a "00000000"`
  - **Parziale:** `v <= "1101" & (others => '0'); -- equivale a "11010000"`

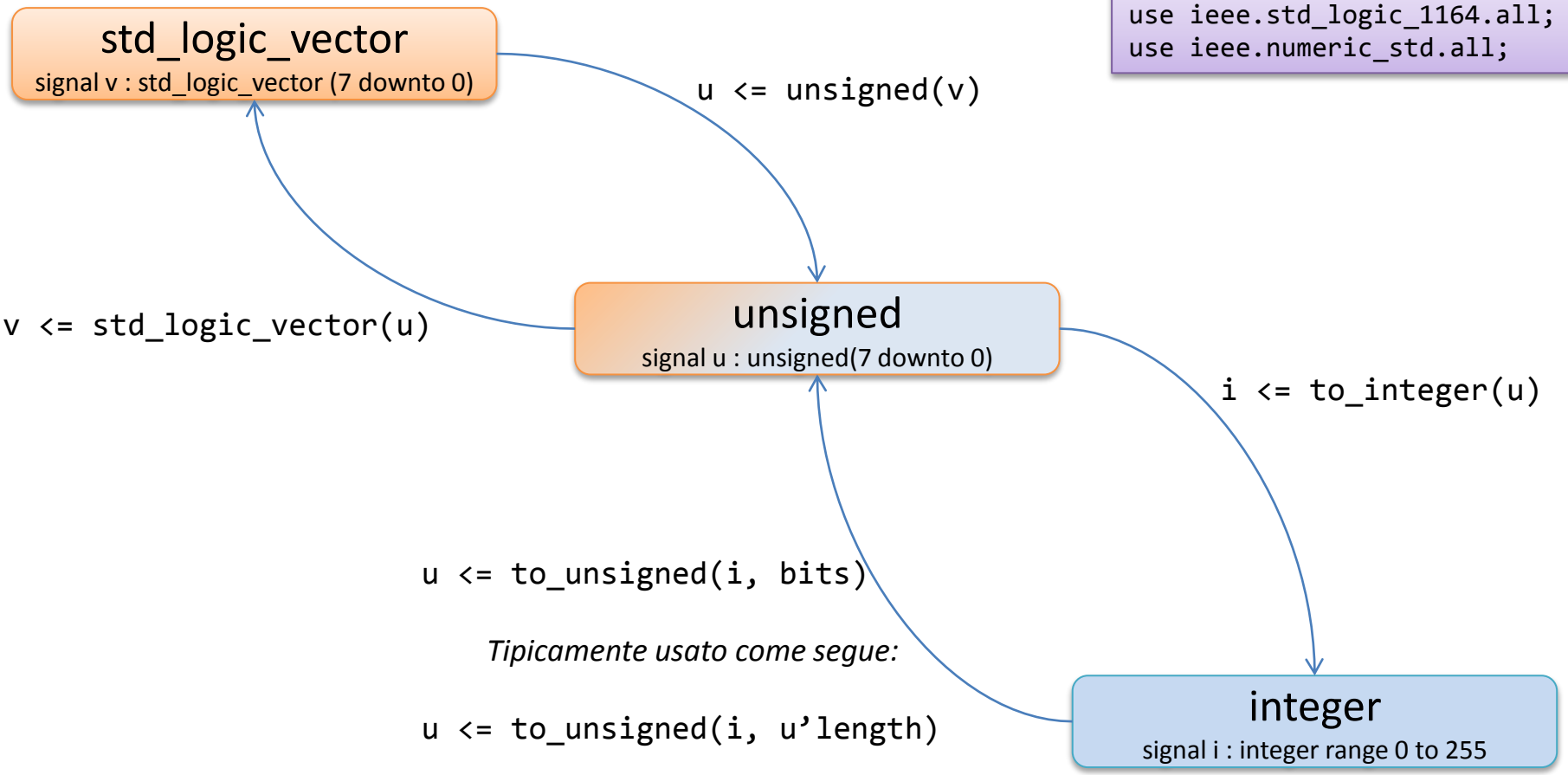
std_logic_vector o integer? (1/3)

- **std_logic_vector:** stringa di bit priva di rappresentazione numerica
  - OK, operazioni bitwise (AND, OR, rotate)
  - NO operazioni aritmetiche, non hanno una corrispondenza con i numeri.
- **integer (e derivati):** numeri interi
  - OK operazioni aritmetiche (perfetti per ALU, contatori …)
  - NO operazioni bitwise
  - Molto comodi da utilizzare: count <= 0, if (count= 1024) then, count<=count+1
  - Hanno una limitazione: possono rappresentare solo interi nel range ± 2^31
  - NON vanno utilizzati come tipi di dato per le porte per i componenti top-level.
- **signed e unsigned:** una via di mezzo tra std_logic_vector ed interi. Sostanzialmente una stringa di bit in cui viene esplicitata la rappresentazione numerica.
  - Supportano tutte le operazioni degli std_logic_vector + quelle aritmetiche
  - Sono più "scomodi" da utilizzare: count <= (others => '0'), if(count = X"ff00ac")…

Quindi? Personalmente solo std_logic_vector ed integer. Al limite cast per I/O

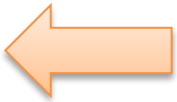
std_logic_vector o integer? (2/3)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```



std_logic_vector o integer? (3/3)

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```



Usare solo package standard

NO ~~std_logic_arith~~

NO ~~std_logic_unsigned~~

Conversioni dirette tra integer e std_logic_vector

Da std_logic_vector ad integer

to_integer(unsigned(nome_segnale_porta_o_variabile))

```
Es:  signal   s : std_logic_vector(3 downto 0);
     signal   i : integer;
     i <= to_integer(unsigned(s));
```

Occhio alla rappresentazione unsigned o signed (compl. a 2)

Da integer a std_logic_vector

std_logic_vector(to_unsigned(numero_o_variabile_int , dimensione))

```
Es:  signal   s : std_logic_vector(3 downto 0);
     signal   i : integer;
     s <= std_logic_vector(to_unsigned(i, s'length));
```

VHDL – Convenzioni e raccomandazioni

```
entity NomeEntita is
    port
    (
        CLOCK                : in  std_logic;
        RESET_N              : in  std_logic;

        INGRESSO_1           : in std_logic;
        INGRESSO_2           : in  std_logic_vector(DATA_SIZE-1 downto 0);

        USCITA_1             : out std_logic;
        USCITA_2             : out std_logic_vector(DATA_SIZE-1 downto 0)
    );
end;
```

Nome entità in CamelCasing

Nome porte in maiuscolo

Non inizializzare
le porte con := …
(al limite farlo al reset)

Definire le porte solo come in, out *(rarissimamente inout)* e solo con i seguenti tipi:

- std_logic, std_logic_vector()
- Tipi di dati strutturati (record) … ma NON nel top-level

Chiarezza e leggibilità

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity AvalonSlaveInterface is

      port
      (
            CLOCK                : in  std_logic;
            RESET_N              : in  std_logic;

            INGRESSO_1           : in std_logic;
            INGRESSO_2           : in  std_logic_vector(DATA_SIZE-1 downto 0);

            USCITA_1             : out std_logic;
            USCITA_2             : out std_logic_vector(DATA_SIZE-1 downto 0)
      );

end;
```

Uno statement per riga

Incolonnare il codice
(con spazi)

Rispettare i livelli di indent
(spazi, al limite tab)

Nomi chiari ed esplicativi che definiscano **chiaramente** il ruolo di ciascuna porta/segnale/entità

~~CLK~~	~~RST~~	~~add_tmp~~	~~urdy~~
CLOCK	RESET	partial_sum	unit_ready

Un identificativo breve non è più efficiente, ma solo più incomprensibile!

Estendibilità

- Evitare numeri e dimensioni (di vettori/porte) cablate nel codice
- Utilizzare un package per le costanti globali
- Adoperare i **generici** se la dimensione caratterizza una intera entità
- Sostanzialmente si tratta di una costante, che però può essere cambiata, istanza per istanza, dal richiedente.

```
entity ShiftRegister is
    generic
    (
        LENGTH              : positive
    )
    port
    (
        CLOCK               : in  std_logic;
        RESET_N             : in  std_logic;

        SER_IN              : in std_logic;
        PARALLEL_OUT        : out std_logic_vector(LENGTH-1 downto 0)
    )
```

Rappresentare in funzione dei generici
Range discendenti per le stringhe di bit

Hint: Se LENGTH è una dimensione,
I range vanno da 0 a LENGTH-1

Definizione entità

```
architecture RTL of NomeArchitettura is
    type BusServicesType              is array (natural range <>) of BusServiceType;
    type   StateType                  is (IDLE, WAIT_SERVICE_ACK);

    constant ACCESS_WRITE             : std_logic := '1';
    constant ACCESS_READ              : std_logic := '0';


    signal state                      : StateType;
    signal nextState                  : StateType;
    ...
begin

    tickGen    : entity work.TickGenerator
      port map(
         PORTA_DEL_COMPONENTE => sengale_locale
          ...
       );
```

Tipi (locali)

Costanti (locali)

Segnali
(alcuni daranno eventualmente luogo a registri)

Richiamare gli eventuali sub-componenti
NOTA: per le entità contenute nel progetto non è necessario ridefinire (copia/incolla) il *component* ma si può importare direttamente l'entità usando la sintassi
entity work.nome_entita

Generazione di hardware

Quando e come viene inferito dell'hardware dal codice VHDL?

- La chiave è negli assegnamenti: ogni qualvolta un segnale (o una porta di uscita) viene assegnato viene generato l'hardware corrispondente.
- Gli assegnamenti danno luogo a gate combinatori (and, or, mux, decoder) o a registri (flip-flop) in base al tipo di *template* utilizzato.

Esistono due tipi di statement in VHDL:

- Statement concorrenti, direttamente nella *architecture*, es:

```
USCITA <= INGRESSO and segnale and not(a or b);

segnale <= '1' when (condizione) else '0';
with (selettore) select (valore_da_assegnare) <= ….
```

- Statement sequenziali, all'interno dei processi: sequenziali o combinatori in base al *template* adottato.

Semantica dei segnali

- Ogni segnale/porta può essere assegnato da un solo processo (combinatorio o sequenziale) o da un solo statement concorrente.
- In un processo i segnali hanno una semantica atomica *stile PLC*: il loro valore è bloccato all'ingresso del processo ed il loro assegnamento ha effetto solo in fondo.
- Nell'ambito di un processo, è possibile assegnare più volte uno stesso segnale. In tal caso vale l'ultimo assegnamento fatto. (in molti casi risulta molto più leggibile). Es:

```
s <= '0';
if (...) then
 s <= '1';
end if;
```

Equivale a

```
if (...) then
 s <= '1';
else
 s <= '0';
end if;
```

```
v    <= (others => '0');
v(0) <= '1';
```

Equivale a

```
v    <= (v'high downto 1 => '0') & '1';
```

- Una porta di uscita non può essere letta ma solo assegnata

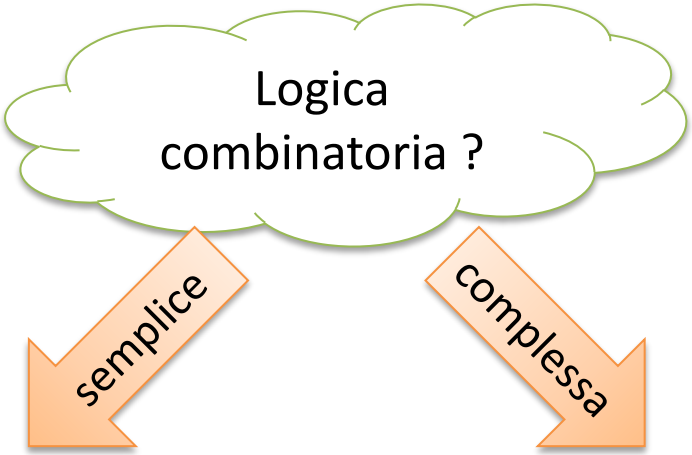
HINT: usare un segnale intermedio

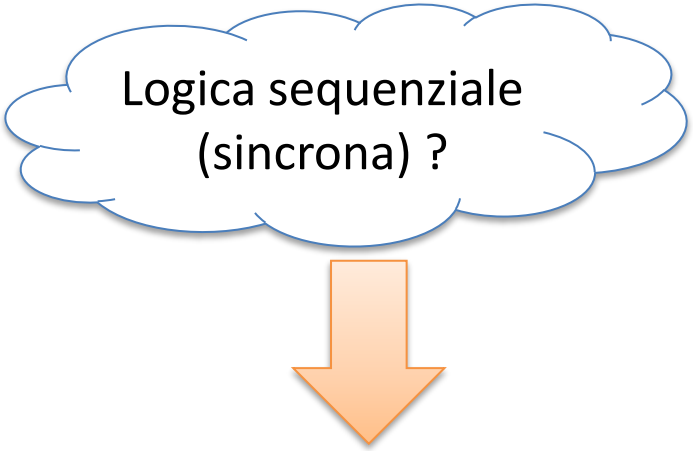
```
USCITA_1 <= a and b;
USCITA_2 <= USCITA_1 and c;
```

```
signal s : std_logic;
...
s <= a and b;
USCITA_1 <= s;
USCITA_2 <= s and c;
```

Template VHDL

Quale template adottare?





Statement concorrenti

```
a<= b and c;
d<='1' when (f='1' or g='1')
       else '0';

with x select y <=
   '1' when "0000",
   '0' when "0001",
    …  when others;
```

Template combinatorio

```
Nome : process(ingr1, ingr2…)
begin
   uscita <= '0';
   if (ingr1='1') then
      uscita <= '1';
   end if;
   …statement sequenziali…
end process;
```

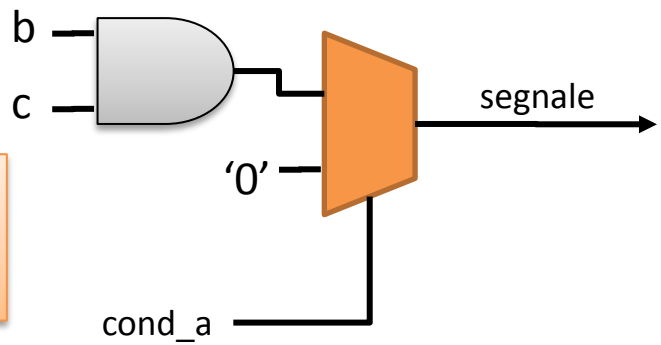
Template sequenziale

```
Nome : process(CLOCK, RESET_N)
begin
   if (RESET_N = '0') then
      …valori di reset…
   elsif rising_edge(CLOCK) then
      …statement sequenziali…
   end if;
end process;
```

Sintesi di template combinatori

```
process(a, b, c)
 if (cond_a = '1') then
      segnale  <= b and c;
  else
      segnale  <= '0';
  end if;
 end if;
end process;
```

Tutti i segnali/porte letti nella sensitivity list



Intuitivamente:

- Ogni assegnamento dà luogo a dei gate sul percorso del segnale
- NON ci devono essere assolutamente loop (a <= b; b <= a)
- L'assegnamento DEVE essere sempre completo (quindi sempre un ramo "else" o un valore predefinito in testa per ogni segnale assegnato).
  Altrimenti la sintesi da luogo a latch, che vanno assolutamente evitati

Hint: il sintetizzatore segnala tutte queste condizioni sotto forma di warning.

Tenete d'occhio i warning… il più delle volte sono dei veri e proprio errori (un po' come in C/C++)

Template per logica sequenziale

```
nome_processo : process (RESET_N, CLOCK)
begin

  if (RESET_N = '0') then
     segnale  <= '0';
     registro <= (others => '0');

  elsif rising_edge(CLOCK) then
    if (. . .) then
       segnale <= '1';
    elsif (. . .) then
       registro <= PORTA_INGRESSO;
       case (...)
    else
       . . .
 end if;
end process;
```

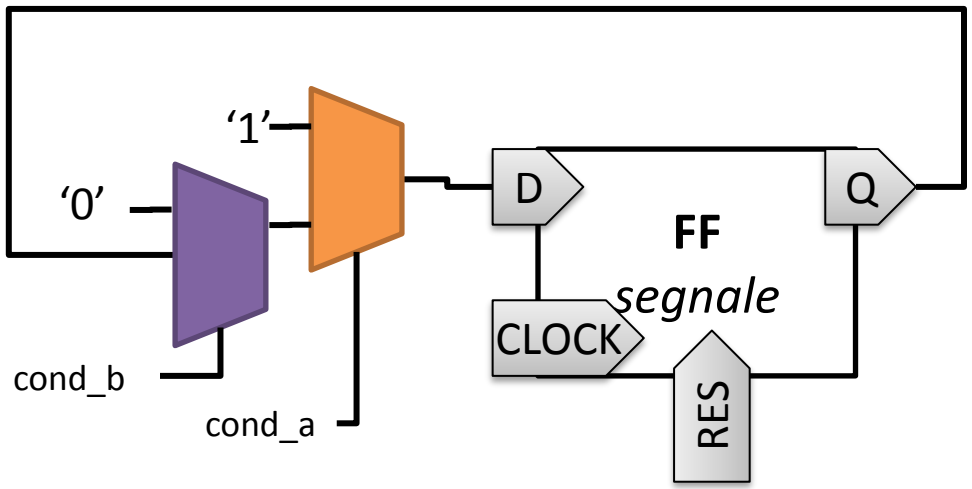
Solo RESET_N e CLOCK nella sensitivity list

Reset **asincrono** dei registri
(Solo a fini di inzializzazione del sistema NON usare per servizi sincroni)

Logica sincrona
Gli assegnamenti di segnali e porte fatti sotto rising_edge danno luogo a registri (FF). Tutti gli assegnamenti avranno luogo atomicamente nell'immediato seguire del fronte di salita del clock.

Sintesi di template sequenziali

```
process(CLOCK, RESET_N)
 if (RESET_N = '0') then
      segnale  <= '0';
  elsif rising_edge(CLOCK) then
   if (cond_a) then
      segnale <= '1';
   elsif (cond_b) then
      segnale <= '0';
    end if;
 end if;
end process;
```



- Ogni segnale assegnato sotto rising_edge dà luogo ad un FF (o registro se è un segnale composto)
- Le alternative nel flusso (if, elsif) danno luogo a mux che selezionano l'ingresso
- Se un segnale è assegnato più volte, vale alla fine l'ultimo assegnamento
- Se non tutti i rami del flusso danno luogo ad un assegnamento, il FF mantiene il suo valore precedente (a meno che non vi sia un assegnamento in testa di default)
- Il valore del registro sarà attuato solo dopo il fronte del clock (quindi gli altri potranno leggerlo solo al fronte successivo)

Cicli for e while

- Come regola generale è possibile sintetizzare un ciclo se le condizioni da cui dipende sono note a tempo di sintesi (costanti, dimensioni di una porta), Es:

```
for i in 0 to COSTANTE-1 loop
for i in PORTA'range loop
while (i < PORTA'length) loop
   …
   i := i + 2;
end loop;
```

(Hint: nei cicli for NON è necessario dichiarare la variabile i)

- Per rendersi conto di cosa venga sintetizzato basta sbrogliare a mano il loop. Es:

```
for i in 0 to 3 loop
  v_out(i) <= v_in_1(i) xor v_in_2(3-i);
end loop;
```

equivale a scrivere

```
v_out(0) <= v_in_1(0) xor v_in_2(3);
v_out(1) <= v_in_1(1) xor v_in_2(2);
v_out(2) <= v_in_1(2) xor v_in_2(1);
v_out(3) <= v_in_1(3) xor v_in_2(0);
```

- **Assolutamente da evitare** loop che dipendono da valori noti solo a runtime, es:
  - ~~for i in 0 to to_integer(INGRESSO) loop~~
  - ~~while (ingresso = '0') loop…~~
  - ~~break~~

Quando usare le *variable* (nei *process*)?

- Nei *process* è possibile definire delle variabili. Sembrano molto simili ai segnali? Quali e quando li utilizziamo?
- Non esiste una regola precisa, ma …
- Nell'ottica della sintesi si dovrebbero usare quasi sempre i segnali, soprattutto se state descrivendo registri, o funzioni combinatorie.
- Perché? E' difficile valutare l'hardware che verrà inferito usando le variabili. Spesso la rete che ne deriva non riflette le aspettative.
- L'uso di variabili nei process dovrebbe essere particolarmente contenuto:
  1. Come "alias": assegnate una sola volta ed in seguito solo lette (slide succ.)
  2. Per definire logica combinatoria "estendibile" (slide succ.)

Uso delle *variable* come alias

Assegnate una sola volta, in testa al processo, e poi solo lette

```
process(…)
begin

   if(signal_a.cell(2).id == 2)
    …
  elsif(signal_a.cell(2).id == 3)
    …
```

```
process(…)
  variable cell_id : integer;
begin
   cell_id := signal_a.cell(2).id;
   if(cell_id == 2)
    …
   elsif(cell_id == 3)
    …
```

Uso delle *variable* per generare logica combinatoria

Caso molto semplice: AND di tutti i bit di un std_logic_vector

```
signal vec        : std_logic_vector(3 downto 0);
signal and_result : std_logic;
…
and_result <= vec(3) and vec(2) and vec(1) and vec(0);
```

Ma se la dimensione del vettore non è nota a priori? Esempio
signal vec : std_logic_vector(WIDTH-1 downto 0);

```
process(vec)
  variable and_var : std_logic;
begin
 and_var := '1';
 for i in vec'range loop
    and_var := and_var and vec(i);
 end loop;
 and_result <= and_var;
end process
```

Troppo potere (espressivo) dà alla testa

```
USCITA_1 <= ING_1 and ING_2;
```

```
USCITA_1 <= '1' when (ING_1 = '1' and
    ING_2 = '1') else '0';
```

```
with (ING_1 & ING_2) select USCITA_1 <=
    '1' when "11",
    '0' when others;
```

E' spesso possibile codificare una stessa rete in modalità differenti. Quale preferire?

Quello più chiaro e leggibile!

```
nome_processo : process(ING_1,ING_2)
    begin
      USCITA_1    <= ING_1 and ING_2;
    end process;
```

```
nome_processo : process(ING_1,ING_2)
    begin
      if (ING_1 = '1' and ING_2 = '1')
    then
        USCITA_1    <= '1';
      else
        USCITA_1    <= '0';
      end if;
    end process;
```

```
nome_processo : process(ING_1,ING_2)
    begin
      USCITA_1    <= '0';
      if (ING_1 = '1' and ING_2 = '1')
         then
        USCITA_1    <= '1';
      end if;
    end process;
```

and (rosso) : and tra conidzioni booleane
and (azzurro): and tra std_logic

Qualche esempio in VHDL

Multiplexer

- Usando statement concorrenti:

```
with sel select uscita <=
   a when “00”,
   b when “01”,
   c when “10”,
   d when others;
```

- In un processo combinatorio

```
process(sel,a,b,c,d)
begin
     case sel is
       when "00" =>   uscita <= a;
       when "01" =>   uscita <= b;
       when "10" =>   uscita <= c;
       when others => uscita <= d;
     end case;
end process;
```

Decoder n -> 2^n

```
port(
    sel             : std_logic_vector(2 downto  0);
    uscita      : std_logic_vector(7 downto  0)
);
…..
process(sel)
     variable sel_int : integer;
begin
     sel_int := to_integer(unsigned(sel));
     uscita <= (others => ‘0’);
     uscita(sel_int) <= ‘1’;
end process;
```

Shift register serial in (MSB first), parallel out

```
architecture RTL of ShiftRegister is
  signal shift_register : std_logic_vector(WIDTH-1 downto 0);
begin

  process(CLOCK, RESET_N)
  begin
    if (RESET_N = '0') then
      shift_register <= (others => '0');
    elsif rising_edge(CLOCK) then
      if (SHIFT_ENABLE = '1') then
        shift_register <=
    shift_register(WIDTH-2 downto 0) & SERIAL_IN;
      end if;
    end if;
  end process;

  PARALLEL_OUT <= shift_register;
end architecture;
```

In caso di LSB first

```
SERIAL_IN & shift_register(WIDTH-1 downto 1);
```

Shift register parallel in, serial out (LSB first)

```
architecture RTL of ShiftRegister is
  signal shift_register : std_logic_vector(WIDTH-1 downto 0);
begin

  process(CLOCK, RESET_N)
  begin
    if (RESET_N = '0') then
      shift_register <= (others => '0');
    elsif rising_edge(CLOCK) then
      if (LOAD = '1') then
        shift_register <= PARALLEL_IN;
      else
        shift_register <= shift_register(0) & shift_register(WIDTH-1 downto 1);
      end if;
    end if;
  end process;

  SERIAL_OUT <= shift_register(0);
end architecture;
```

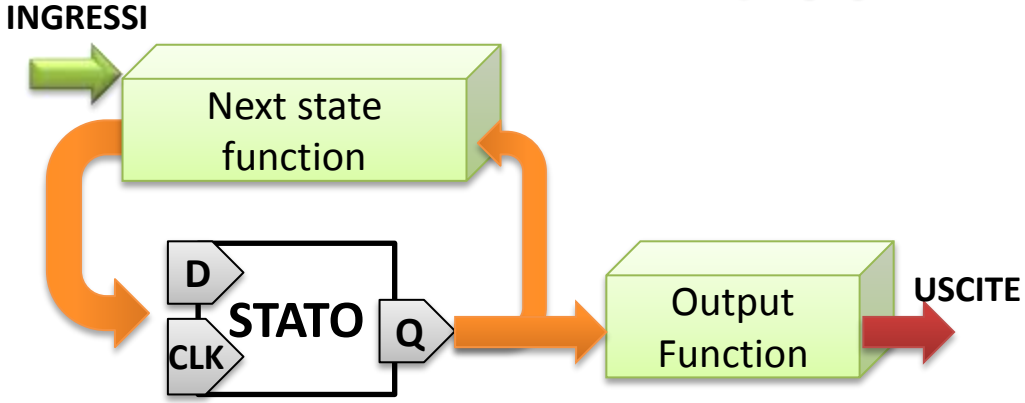
Macchine a stati

- Le macchine a stati rappresentano l'approccio migliore per modellare comportamenti reattivi che evolvono nel tempo.

- Come modellarle? Come implementarle?
  - Moore FSM / Mealy FSM
  - Uscite combinatorie / uscite registrate

- Avremo modo di vederle meglio in azione nel caso di studio.

Moore FSM (approccio tradizionale)



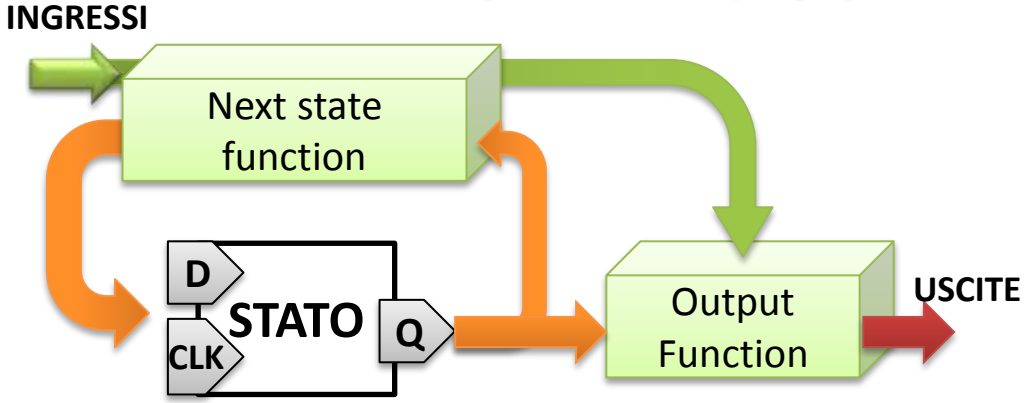
Come implementarla in VHDL?

- Un processo comb. per la next state function
- Un processo comb. per la output function
- Tipicamente i due processi soprastanti si accorpano per praticità e leggibilità.
- Un processo sincrono (elementare) per lo stato

```
architecture RTL of <Entita> is
  type state_type is (STATO1, STATO2,…);
  signal state_r    : state_type;
  signal next_state : state_type;
begin

  State_reg : process (CLOCK, RESET_N)
    begin
      if (RESET_N = '0') then
       state_r <= STATO_Al_RESET;
      elsif rising_edge(CLOCK) then
       state_r <= next_state;
      end if;
  end process;
```

```
OutputAndNextState : process (state_r, INGRESSI)
begin
    USCITA_1   <= '0';
    USCITA_2   <= '0';
    next_state <= state_r;
    case (state_r) is

      when STATO_1 =>
        USCITA_1 <= '1';
        if (IN1 = '1') then
           next_state <= STATO_2;
        end if;

      when STATO_2 =>
           …omissis
    end case;
end process;
```

Mealy FSM (approccio tradizionale)



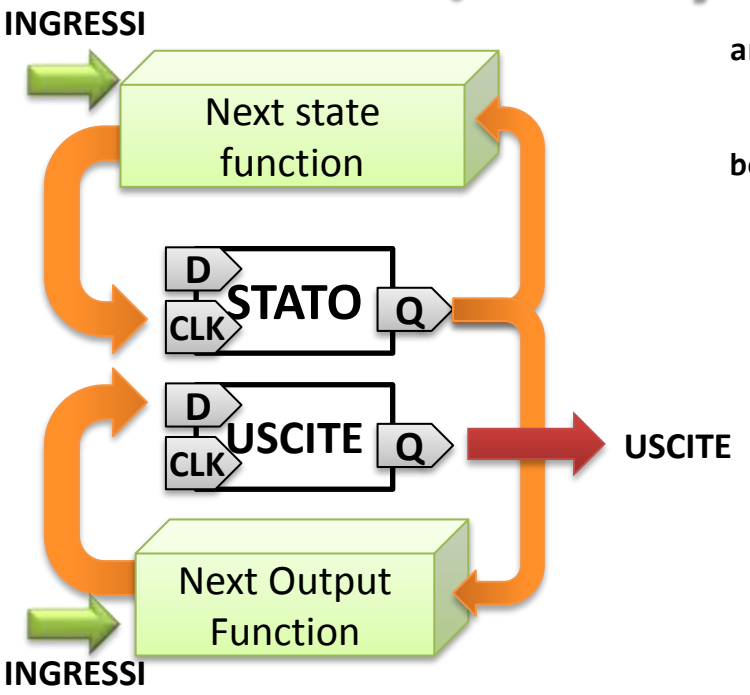
Analogamente a prima

- L'unica differenza è che l'uscita può essere condizionata anche dagli ingressi.
- Sconsigliata perché può dare luogo a comportamenti indesiderati e loop combinatori inaspettati.

```
architecture RTL of <Entita> is
  type state_type is (STATO1, STATO2,…);
  signal state_r    : state_type;
  signal next_state : state_type;
begin

  State_reg : process (CLOCK, RESET_N)
    begin
      if (RESET_N = '0') then
       state_r <= STATO_Al_RESET;
      elsif rising_edge(CLOCK) then
       state_r <= next_state;
      end if;
  end process;
```

```
OutputAndNextState : process (state_r, INGRESSI)
begin
    USCITA_1   <= '0';
    USCITA_2   <= '0';
    next_state <= state_r;
    case (state_r) is
      when STATO_1 =>
        USCITA_1 <= '1';
        if (IN1 = '1') then
           next_state <= STATO_2;
           USCITA_2 <= '1';
        end if;

      when STATO_2 =>
           …omissis
    end case;
end process;
```

Moore/Mealy FSM con uscite registrate




```
architecture RTL of <Entita> is
  type   state_type   is (STATO1, STATO2,…);
  signal state_r     : state_type;
begin

  StateMachine : process (CLOCK, RESET_N)
    begin
      if (RESET_N = '0') then
       state_r   <= STATO_Al_RESET;
       USCITA_1  <= '0';
      elsif rising_edge(CLOCK) then
       USCITA_1  <= '0'; --inizializzare se monoimpulsive

       case (state_r) is
         when STATO_1 =>
           USCITA_1 <= '1';
          if (IN1 = '1') then
           next_state <= STATO_2;
          end if;

          when STATO_2 =>
             …omissis
        end case;
      end if;
  end process;
```

- Molto più flessibile
- Facile da implementare (un solo processo sequenziale) e molto leggibile.

Uno svantaggio

- Le uscite si aggiornano con un clock di ritardo. Può portare problemi modellando comportamenti reattivi.

A differenza del caso precedente l'uscita viene aggiornata al clock successivo rispetto al quale si transita in STATO_1

Risorse e riferimenti

- OpenCores www.opencores.org
  Comunità open-source. Molti IP core open-source a disposizione in VHDL e Verilog.

- A Short Introduction to VHDL (185 slide)
  http://si2.epfl.ch/~zanini/class/win2010/_HW_schedule/VHDL_full.pdf

- VHDL Quick Reference Card http://www.vhdl.org/rassp/vhdl/guidelines/vhdlqrc.pdf
  Un raccolta in 2 pagine degli elementi chiave del VHDL. Ottimo da stampare e tenere sempre a portata di mano.

- Forum specializzati http://www.fpgarelated.com/

- Documentazione Altera: http://www.altera.com/literature/lit-index.html
  Dettagli FPGA e tool di sviluppo: veramente ben fatta