Politecnico di Milano
Facoltà di Ingegneria dell'Informazione
Dipartimento di Elettronica e Informazione
Corso di Laurea in Ingegneria Informatica

VHDL
Introduzione

Ott./Nov.2018
Gen. 2012
Feb. 2007

Politecnico di Milano

Sommario

- Generalità
- Interfaccia
- Funzionalità
  - Livelli di descrizione
- Segnali
- Reti Combinatorie
- Macchine a stati
  - Process, Variabili e segnali, Istruzioni sequenziali
- Gerarchia
- Strutture parametriche
  - Costrutti `generic` e `generate`
- VHDL e sintesi

Generalità

- VHDL è un linguaggio concepito per scrivere modelli di sistemi digitali
  - **V**ery (High Speed Integrated Circuit) **H**ardware **D**escription **L**anguage
    - Nefinito negli anni ’80, l’ultima versione pubblica risale al 1993 (IEEE std 1076-1993)
- Ragioni per realizzare un modello
  - Specifica dei requisiti
  - Documentazione
  - Verifica mediante simulazione
  - **Sintesi**
    - Concetto di base: raggiungere la massima affidabilità del processo di progetto riducendo sai il costo sia il tempo che gli errori di progetto

Generalità

- VHDL non è un linguaggio eseguibile
  - Non descrive quali operazioni devono essere eseguite da un esecutore generico
- VHDL è un linguaggio che consente di produrre una specifica che può essere simulata
  - Due aspetti:
    - Specificare un progetto
    - Verificarne il comportamento che la specifica descrive
- VHDL è un linguaggio reattivo e parallelo
  - Il parallelismo è il paradigma di esecuzione dell'HW

Generalità

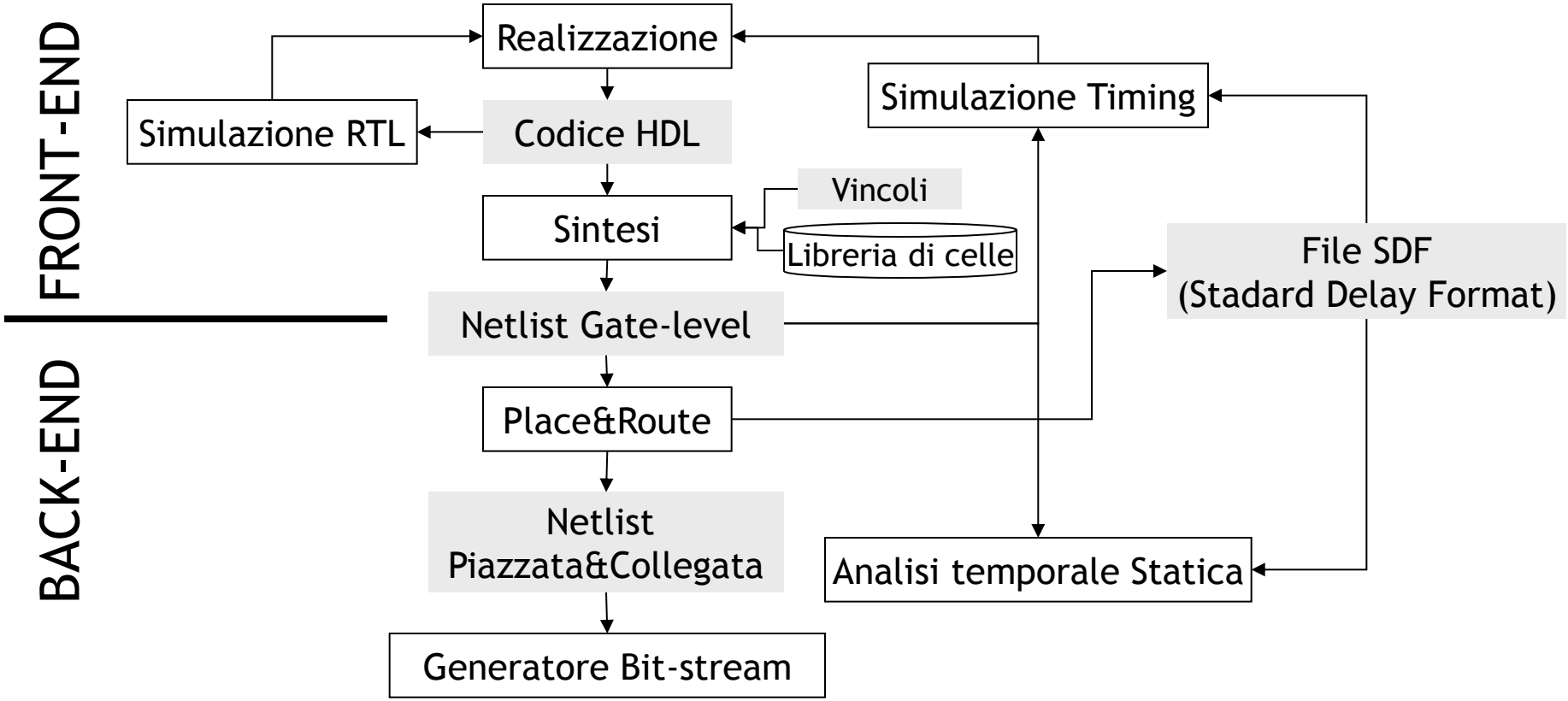
- Esigenze di progetto
  - Struttura di un progetto (specifica)
    - In generale, un sistema hardware è composto da una gerarchia di sottosistemi attivi in parallelo
    - Ogni sottosistema è caratterizzato da
      - Interfaccia
        - » Specifica quello che entra ed esce dal sottosistema
      - Funzionalità
        - » Specifica la relazione tra i dati in ingresso e i dati in uscita al sottosistema
  - Verifica del progetto
    - Un sistema descritto deve poter essere verificato
      - Sia funzionalmente sia temporalmente

Generalità

- Passi per realizzare un progetto in HDL
  - Definizione dei requisiti
    - Funzionalità, domini di clock, frequenze operative, formato dei dati,
  - Descrizione del progetto in codice HDL
  - Simulazione del codice sorgente per la verifica funzionale
    - La simulazione, a questi livello, considera solo l'aspetto logico del progetto; gli aspetti relativi alla implementazioni fisica, che considera anche l'aspetto temporale, sono affrontati in altri momenti
  - Sintesi e ottimizzazione
    - Il codice sorgente è sintetizzato e mappato sulla architettura target minimizzandone l'area o il tempo di esecuzione (prestazione)
  - Simulazione per la verifica dei vincoli di tempo
    - Si considera la fisica implementazione e si simulano sia gli aspetti logici che quelli temporali;
  - Implementazione finale

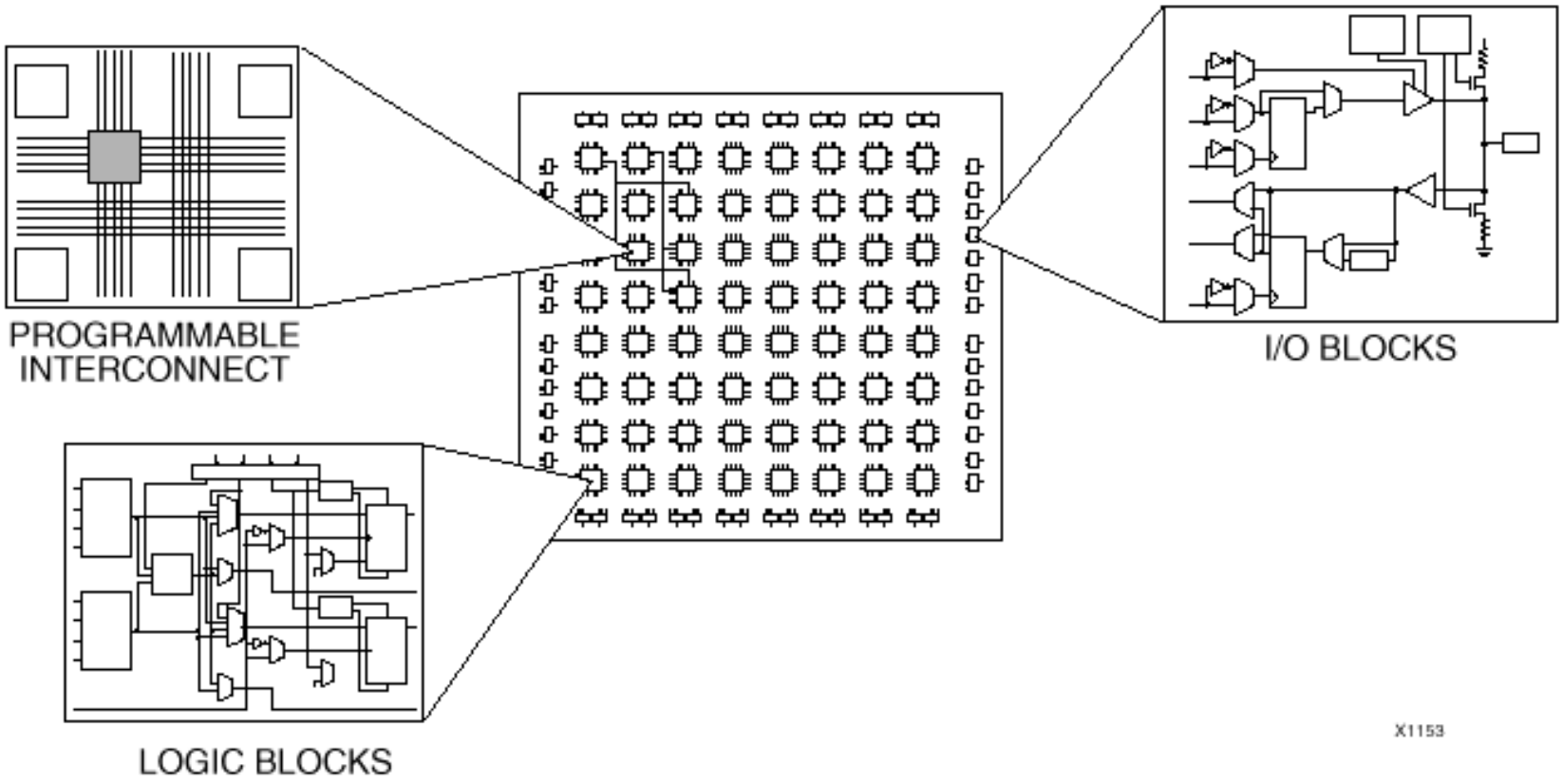
Metodologia e Flusso di Progetto

- Flusso di progetto per FPGA



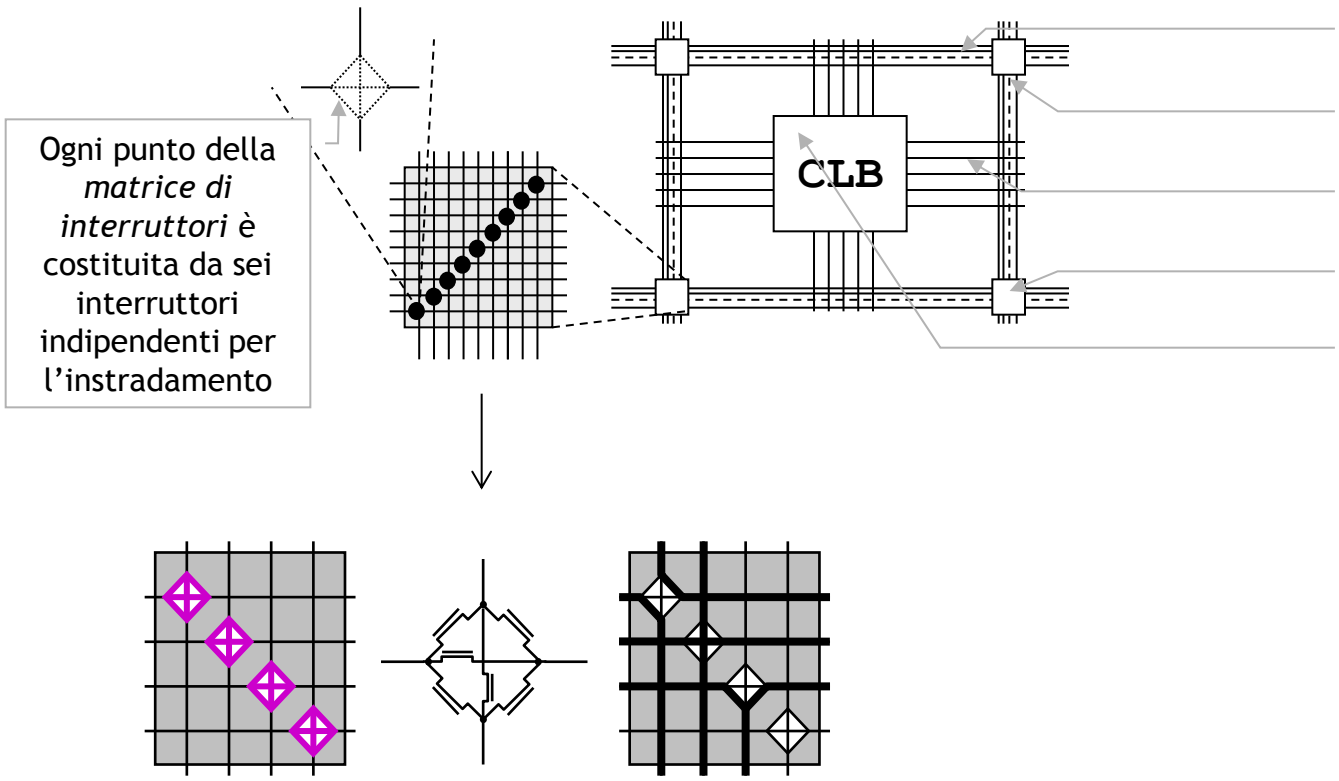
Field Programmable Gate Array

- Schema semplificato di una FPGA



Field Programmable Gate Array

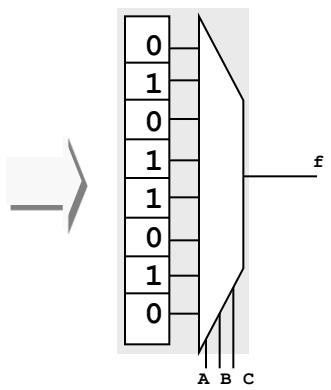
- Matrice di interconnessione
  - Programmable Interconnect



Field Programmable Gate Array

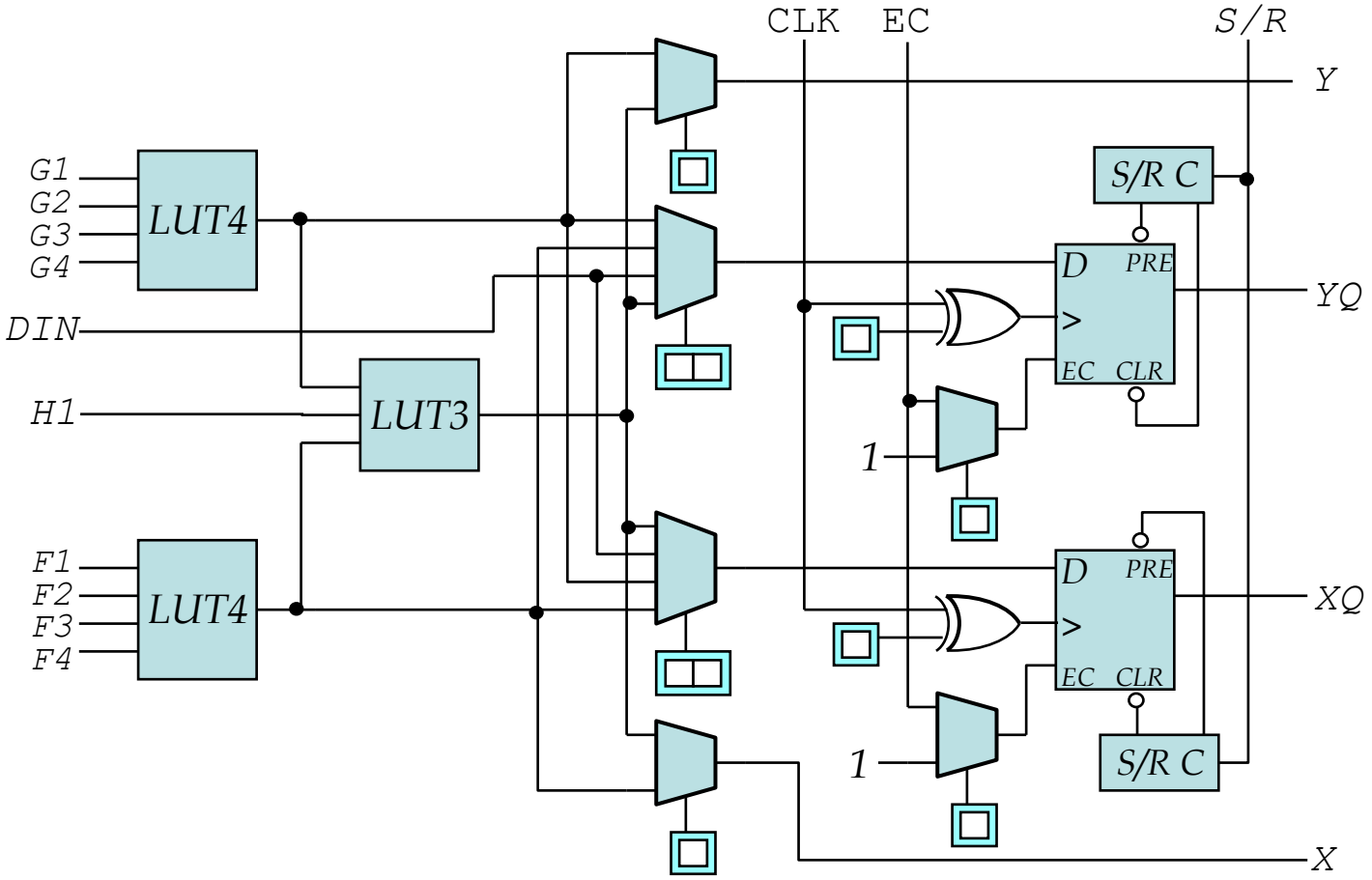
- LUT (look-up table): implementa una generica rete combinatoria a k ingressi
  - Combinazione di 2^k registri ed un multiplexer a k ingressi di selezione.
  - I valori caricati nei 2^k registri sono i valori assunti dalla funzione combinatoria. Il caricamento avviene in fase di configurazione della FPGA.
  - I k ingressi di selezione sono controllati dalla variabili di ingresso della funzione da realizzare.

A	B	C	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

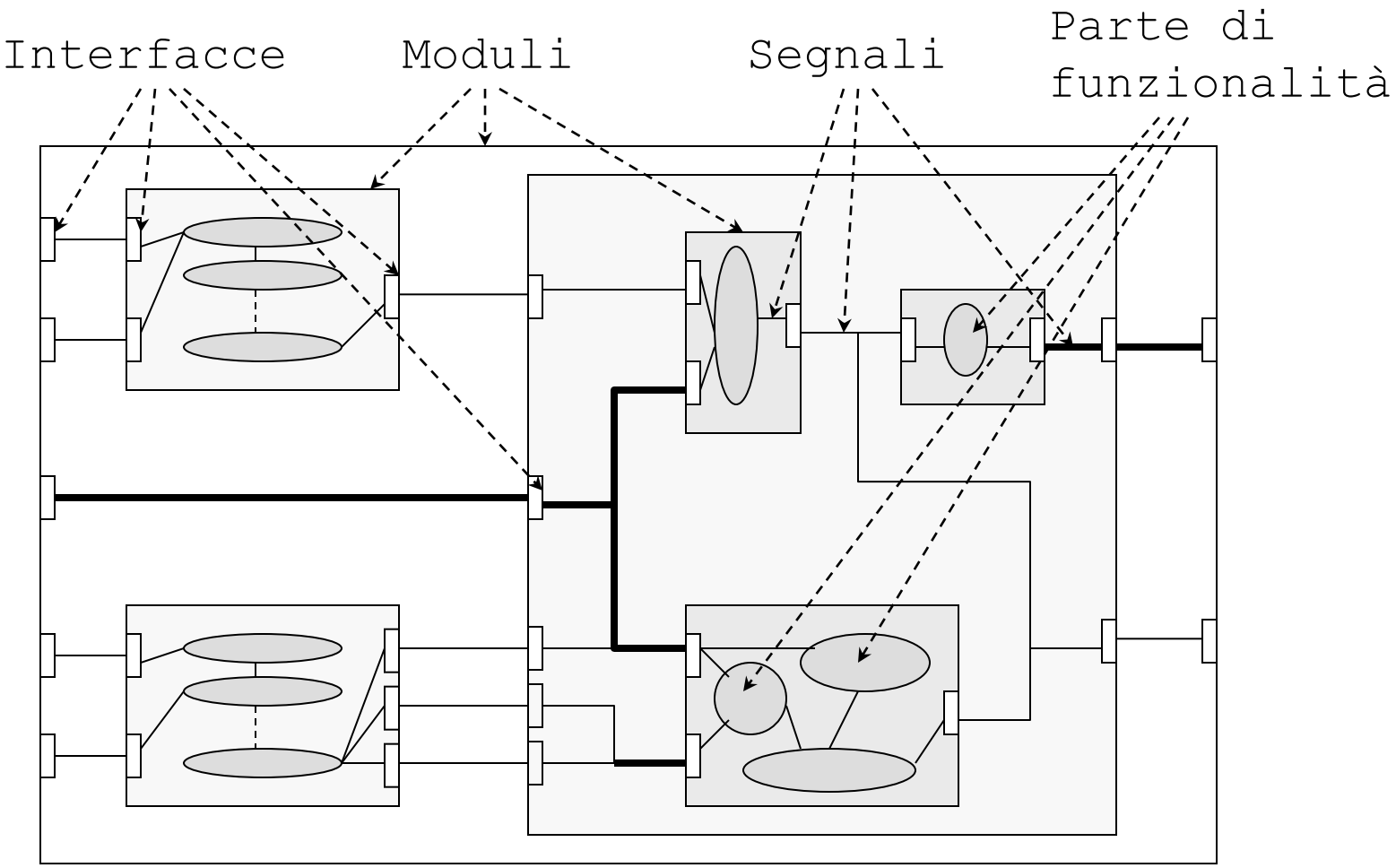


FPGA: Esempio di CLB

- CLB - Configurable Logic Block (schema semplificato)

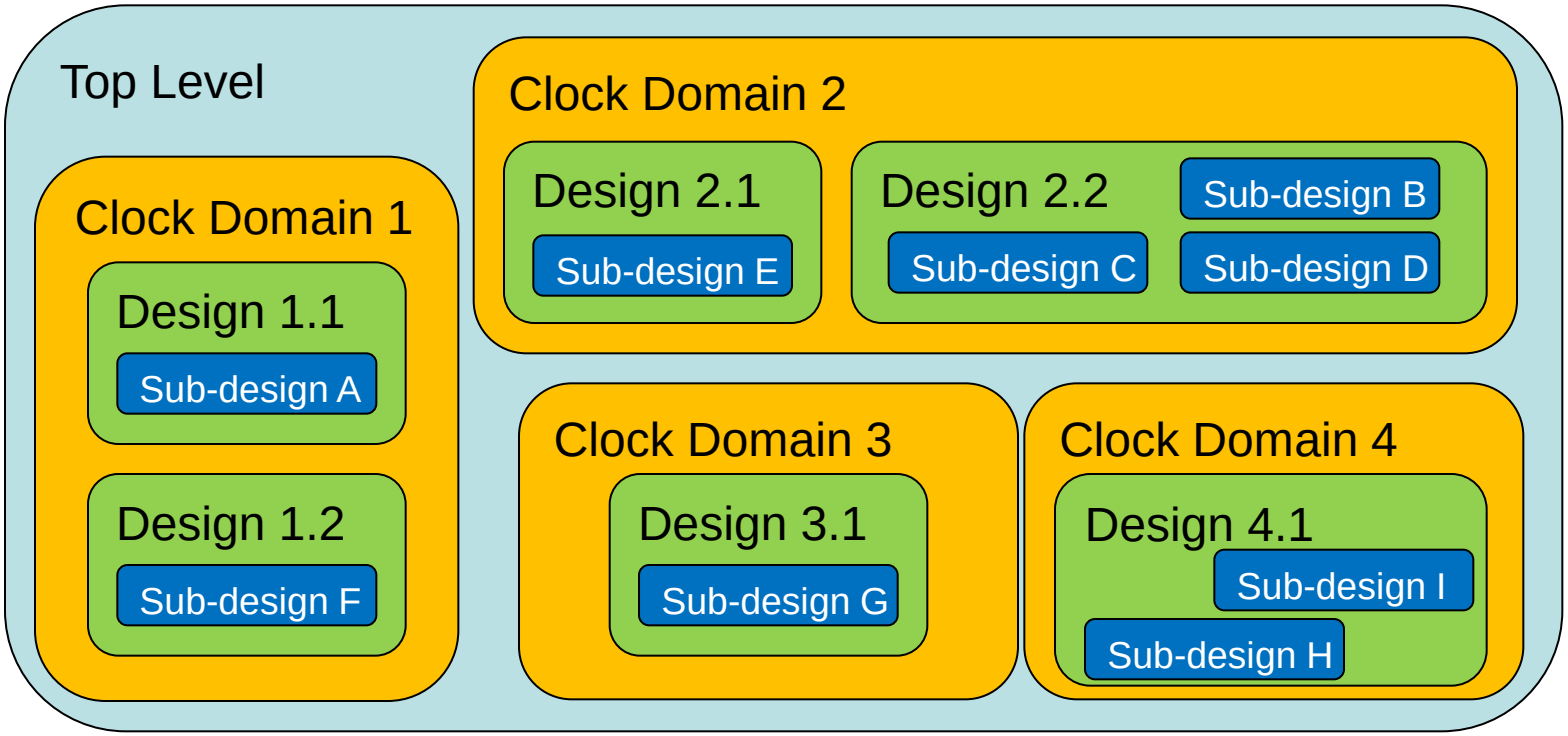


Struttura di un progetto



Struttura di un progetto

- In genere, la realizzazione di un progetto complesso attraversa fasi successive di raffinamento. L'approccio è gerarchico.

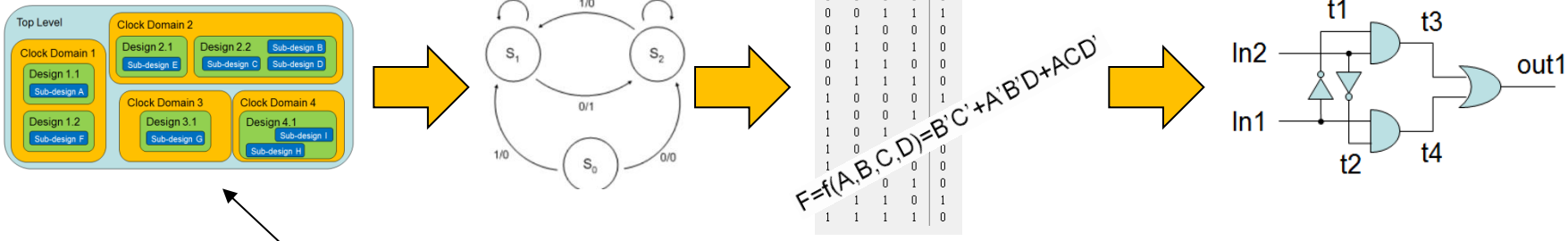


Struttura del codice sorgente

- Il codice sorgente di un modello VHDL è un file di testo
- In genere si usa un nome uguale al nome del modulo e l'estensione deve essere **`.vhd`**
- Il VHDL è case insensitive
- "`--`" indica l'inizio di una riga di commento al codice

Elementi Base

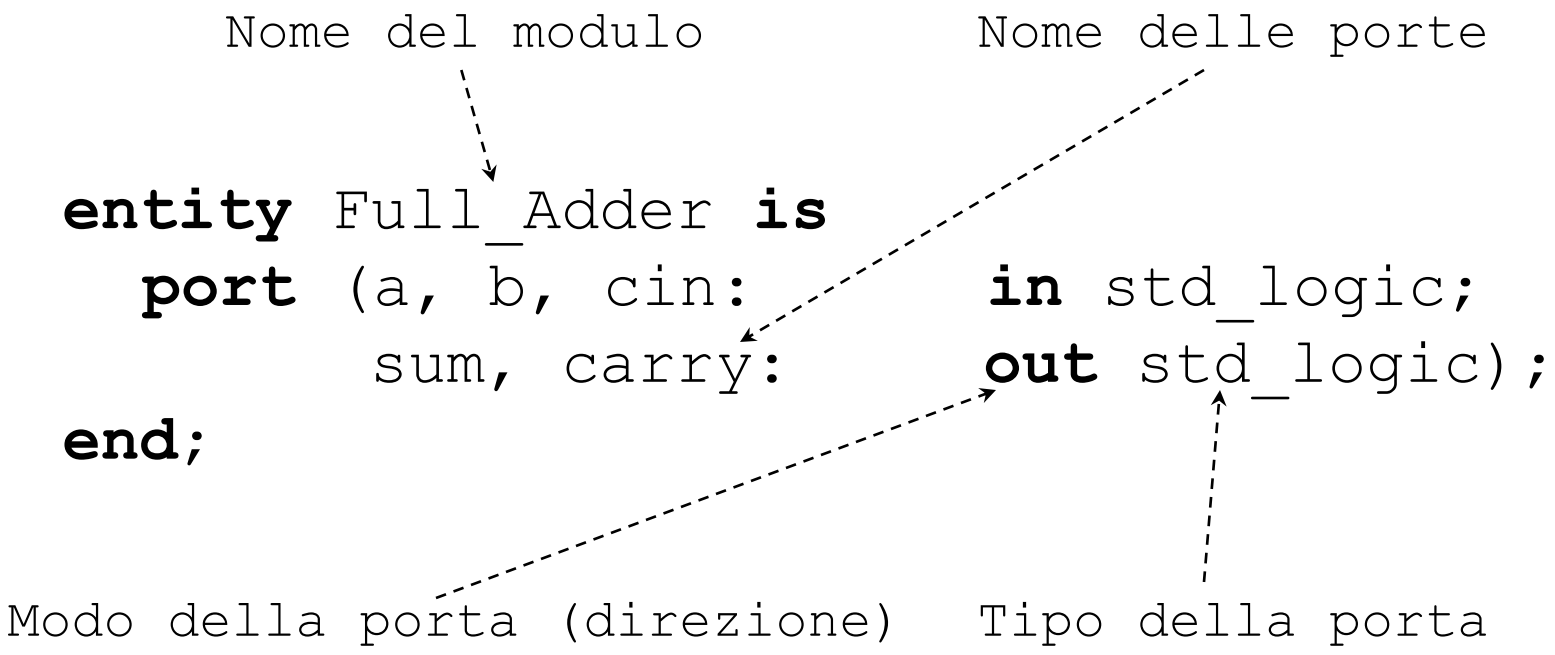
- Modulo
  - Interfaccia (*Entity*)
  - Funzionalità (*Architecture*)
    - 3 Livelli di astrazione possibili
      - Strutturale o Gate-Level
      - RTL o Data-Flow
      - Algoritmico o Behavioural



- Gerarchia dei moduli (Strutturale)

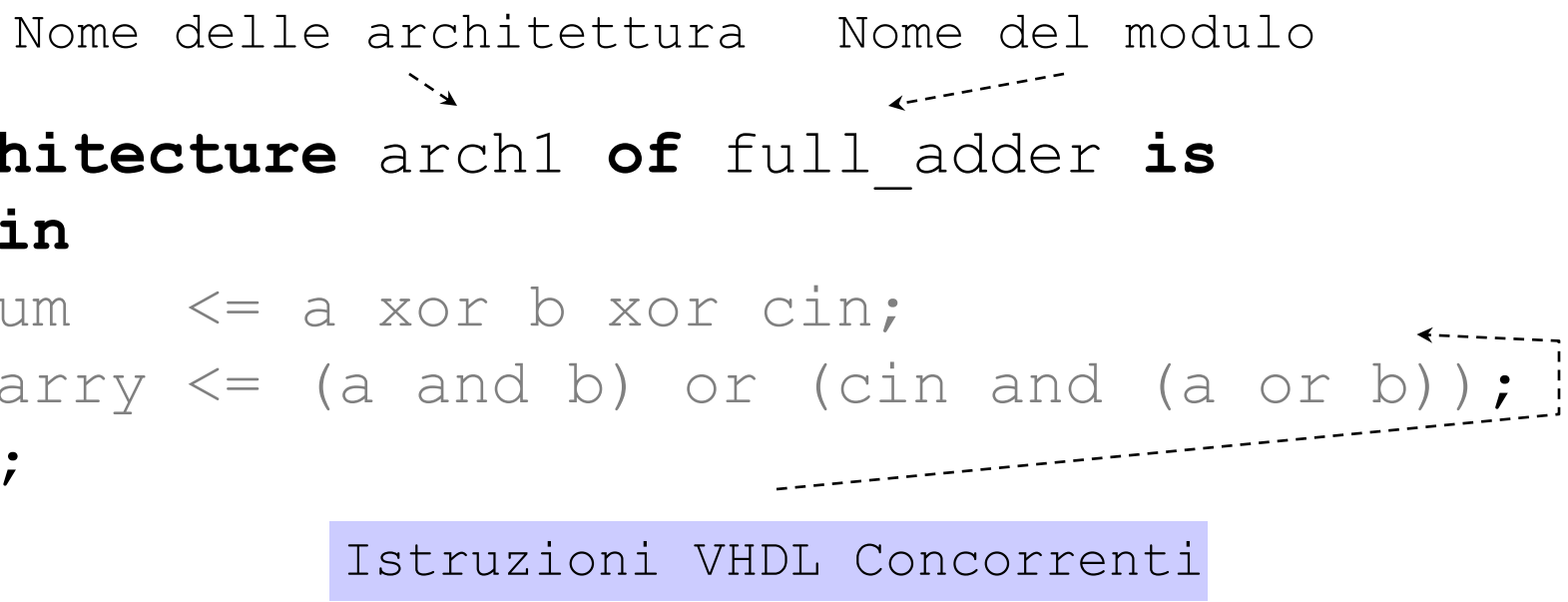
Interfaccia

- Dichiarazione della entità (*Entity Declaration*)
  - Descrive l'interfaccia di ingresso/uscita di un modulo



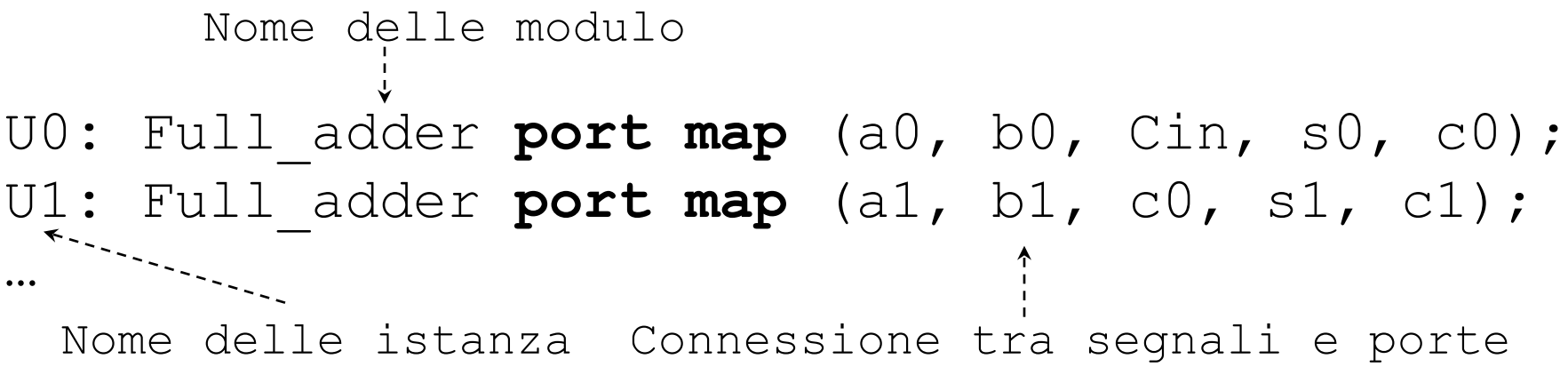
Funzionalità

- Dichiarazione della architettura (*Architecture Declaration*)
  - Descrive come il opera modulo
  - Sono possibili più architetture per ogni entità



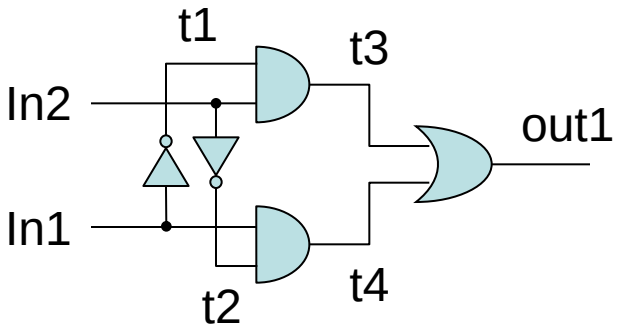
Funzionalità: Livello Strutturale

- Livello di astrazione più basso
- La funzionalità è idealmente espressa mediante un grafo
  - **Nodi**: rappresentano degli elementi logici
    - Funzionalità semplici o complesse
  - **Archi**: le connessioni tra gli elementi
- La descrizione è attraverso una *netlist*



Funzionalità: Livello Strutturale

- Esempio: XOR



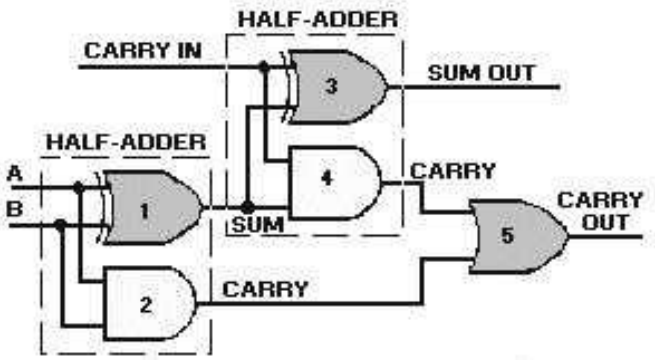

```
architecture structural of x_or is
    -- signal declarations
    signal t1, t2, t3, t4 : std_logic;
    -- local component declarations
    component and_gate
       port (a, b : in std_logic; c : out std_logic);
    end component;
     …
Begin
    -- component instantiation statements
    u0: and_gate port map ( a => t1, b => in2, c => t3);
    u1: and_gate port map ( a => in1, b => t2, c => t4);
    u2: inverter port map ( g => in1, h => t1);
    u3: inverter port map ( g => in2, h => t2);
    u4: or_gate  port map ( d => t3, e => t4, f => out1);
end structural;
```

Connections between signals and ports(by name)

Funzionalità: Livello RTL

- Livello di astrazione intermedio
  - RTL: Register Transfer Level
- La funzionalità è idealmente espressa da un flusso di dati che, propagandosi nel circuito da un registro ad un altro, subisce delle trasformazioni
  - Il trasferimento tra i registri ne giustifica il nome
  - Un sotto circuito può anche non contenere registri

A	B	CARRY IN	SUM OUT	CARRY OUT
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1




```
sum   <= a xor b xor cin;
carry <= (a and b) or (cin and (a or b));
```

Funzionalità: Livello RTL

- F=f(A,B,C,D)=B'C'+A'B'D+ACD'

A	B	C	D	F
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

```
…
entity F is
 port (
    A,B,C,D  : in  std_logic;
    F: out std_logic);
end F;

architecture rtl of F is
begin
 F <= (not B and not C) or (not A and not B and D)
       or (A and C not D);
end rtl;
```

Funzionalità: Livello Algoritmico

- Livello di astrazione più elevato
- La funzionalità è espressa mediante uno o più algoritmi
  - Non deve necessariamente riflettere gli aspetti implementativi del progetto
- Le descrizioni a livello algoritmico sono supportate dalla istruzione **process**

```
…
begin
 process(in1, in2)
 begin
   if in1 = in2 then out1 <= '0';
                else out1 <= '1';
   end if;
 end process;
```

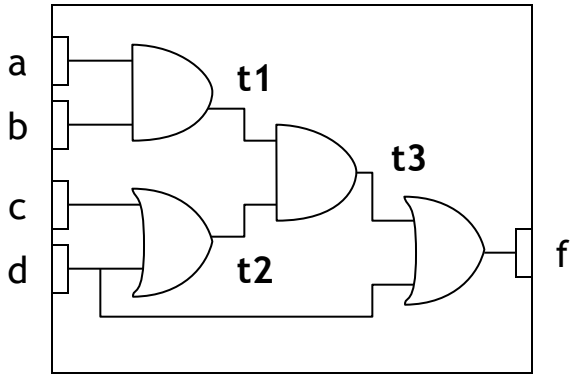
Funzionalità: Livelli Misti

- Una specifica può essere realizzata utilizzando contemporaneamente tutti i livelli di astrazione
  - Tipicamente il livello strutturale è utilizzato solo per la definizione delle gerarchie di progetto
- Direttive di progetto:
  - I moduli elementari (soluzioni effettive) sono descritti a livello RTL e/o Algoritmico (comportamentale)
  - I moduli che definiscono la gerarchia sono descritti a livello strutturale
    - Nota: il modulo a livello gerarchico più elevato è denominato *top-entity.*

Segnali

- I segnali rappresentano i valori dei dati su linee fisiche di un circuito
- I segnali possono essere
  - Segnali di interfaccia
    - utilizzati dai moduli far comunicare il loro contenuto con l'esterno
  - Segnali interni
    - Utilizzati nel modulo per realizzare la funzionalità desiderata

Segnali



- I segnali utilizzati in un modulo sono definiti a livello di architettura e non sono visibili al di fuori del modulo

```
architecture uno of esempio is
  signal t1, t2, t3: std_logic;
begin
```

Std_Logic

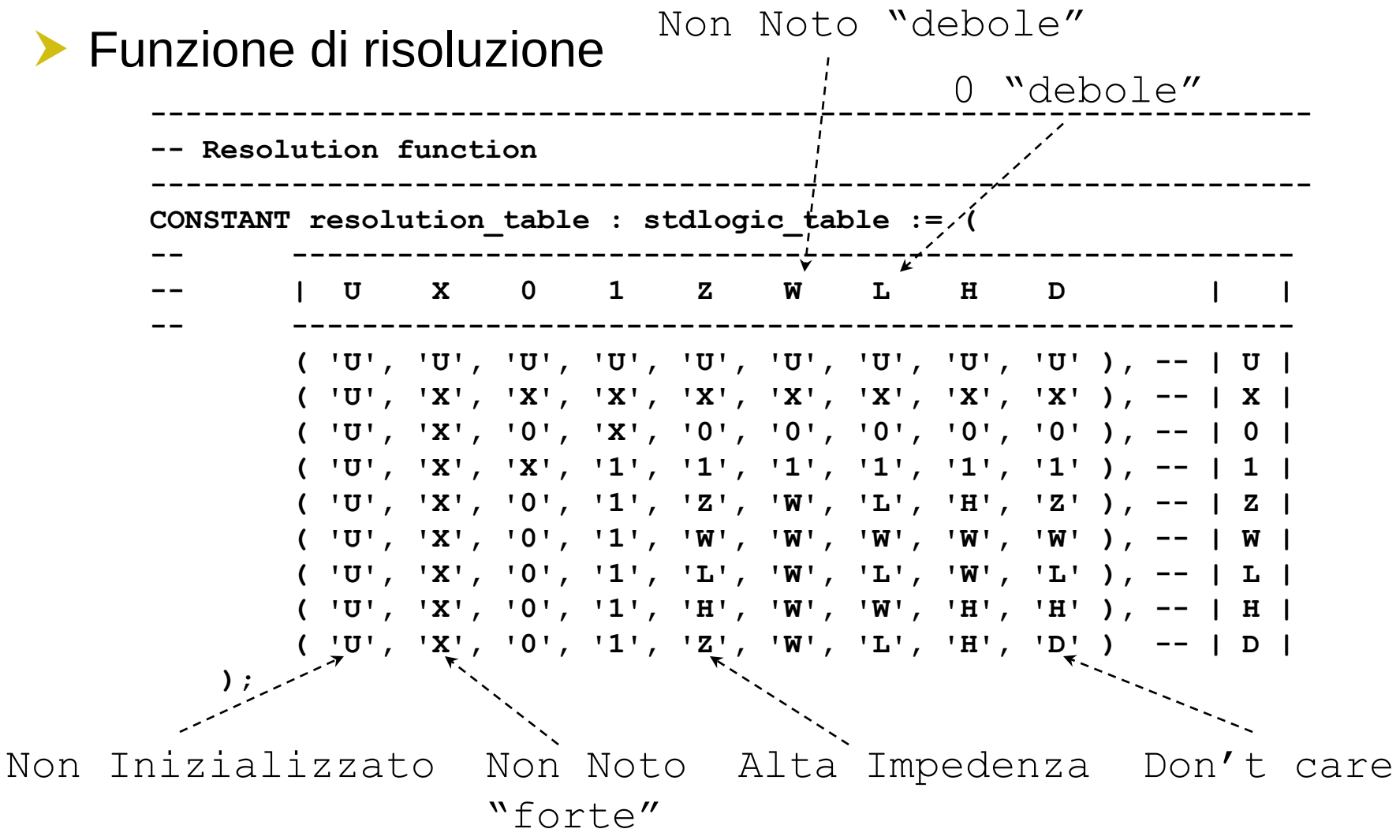
- Il VHDL dispone molti tipi di dato: solo alcuni sono sintetizzabili
  - `Bit, real, integer, time, boolean, file, character, ...`
- Per la sintesi e la simulazione si preferisce utilizzare i tipi definiti nella libreria `std_logic_1164` (IEEE)
  - Deve essere sempre dichiarata

```
library IEEE;
use IEEE.std_logic_1164.ALL;
```

- I tipi definiti nella libreria std_logic_1164 utilizzano un sistema logico a 9 valori
  - Consente di contemplare situazioni come, l'alta impedenza e l'indeterminazione (**NOTA: in simulazione**)
  - Consente di definire l'interazione tra valori differenti
    - Funzione di risoluzione

Std_Logic

- Funzione di risoluzione



Tipi di dato

- Note:
  - Il VHDL è un linguaggio fortemente tipizzato
    - Le operazioni possono essere eseguite solo tra tipi compatibili
    - Non è possibile eseguire cast impliciti
    - Il VHDL fornisce funzioni specifiche per eseguire le conversioni

Segnali

- Scalari
  - **std_logic**
    - `signal a,b: std_logic;`
  - Costanti utilizzano un singolo apice (`'0'`)
    - `signal a,b: std_logic := '0'`
- Vettori (BUS)
  - **std_logic_vector**
    - `signal a: std_logic_vector(0 to 7);`
      - 0 1 2 3 4 5 6 7 (notazione *big endian*)
    - `signal b: std_logic_vector(7 downto 0);`
      - 7 6 5 4 3 2 1 0 (notazione *little endian*)
    - `signal c: std_logic_vector(15 to 22);`
      - 15 16 17 18 19 20 21 22
  - Costanti utilizzano i doppi apici (`"00000000"`)
    - `signal a: std_logic_vector(0 to 7) := "00000000";`
    - `signal b: std_logic_vector(7 downto 0) := others => '0';`
    - `signal a: std_logic_vector(0 to 7) := 0 to 3 => '1', 4 to 7 => '0')`

NOTA: differenza tra il tipo bit e il tipo std_logic è che bit ammette solo i valori 0 e 1

Segnali

- Operazioni sui segnali
  - **Logiche**: `and, or, not, xor, (nxor` – VHDL93)
    - Operazioni bit a bit
  - **Estrazione**: *`nome`*`(`*`indice1`* `to/downto` *`indice2`*`)`
    - VHDL87: L'ordine di estrazione è quello imposto dalla dichiarazione del vettore
      - Un errore genera una stringa nulla
    - VHDL93: L'ordine di estrazione è generico
    - Sia `a` uno `std_logic_vector(0 to 7)`:
      - `a(2 to 3)` seleziona i bit 2 e 3
      - `a(4)` seleziona il bit 4
      - `a(3 downto 0)` non è ammesso
  - **Concatenamento**: `&`

Segnali

- Operazioni sui segnali (continua)
  - **Assegnamento**: `<=`
    - Gli operandi devono essere dello stesso tipo
    - Per assegnare un valore costante e non si conosce la dimensione del dato, si può utilizzare come operando sinistro il costrutto `(others => costante)`
      - Es: `Prova <= (others => 'x')`
  - **Aritmetiche**: `+`
    - Operazione di somma
      - Attenzione: due operandi da `n` bit producono un risultato su `n+1` bit
      - Nota: **`Std_logic_vector` non supporta le operazioni aritmetiche**. Utililizzare **`use ieee.numeric_std.all;`**
      - La libreria **`ieee.numeric_std.all`** definisce i tipi
        - **`unsigned`** per rappresentare numeri interi binari naturali
        - **`signed`** per rappresentare numeri interi binari relativi (codificati in complemento a due)

Segnali

- **Relazionali**: `< <= = /= >= >`
  - Nota: l'operatore di confronto **<=** e quello di assegnamento ad un segnale sono identici
    - Il significato dipende dal contesto

- Esempio di estrazione e concatenamento
  - `signal BUS: std_logic_vector(0 to 13);`
  - `BUS <= "11110000101010";`
  - `(BUS(13)&BUS(12))&BUS(0 to 11)) and BUS(0 to 13)`
    - `BUS(0 to 11): "111100001010"`
    - `BUS(13)&BUS(12): "01"`
    - `(BUS(13)&BUS(12))&BUS(0 to 11)): "011111100001010"`
    - Quindi, infine, `(BUS(13)&BUS(12))&BUS(0 to 11)) and BUS(0 to 13)` produce la configurazione `"011100000001010"`

Segnali

- Esempio di +
  - Siano `Somma`, `A` e `B` dei `std_logic_vector(3 downto 0)` e `Cin` e `Cout` degli `std_logic`
  - Utilizziamo un vettore temporaneo `Temp` per svolgere l'operazione di somma
    - Sia `Temp` un `std_logic_vector(4 downto 0)`
  - Allora:

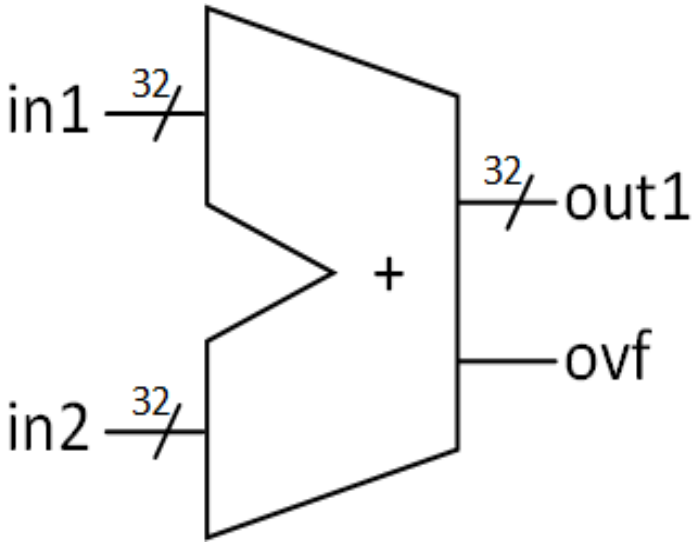
```
Temp <= STD_LOGIC_VECTOR( UNSIGNED( (A(3)&A)
)+UNSIGNED( B(3)&B) + UNSIGNED("0000"&Cin)));
Somma<= Temp(3 downto 0);
Cout <= Temp(4);
```

  - Librerie
    -

```
library IEEE;
Use IEEE.std_logic_1164.ALL;
Use IEEE.NUMERIC_STD.ALL;;
```

Esempio

- Vogliamo specificare un circuito combinatorio esegue la somma di due valori in ingresso a 32 bit codificati in complemento a 2 e produce in uscita il risultato dell'operazione a 32 bit ed un segnale di overflow



Esempio da slide prof. Antonio Miele

Esempio

- *entity* e *architecture* di esempio:

```
entity esempio is
  port(
    in1, in2: in std_logic_vector(31 downto 0);
    out1 : out std_logic_vector(31 downto 0);
    ovf : out std_logic
  );
end esempio;

architecture dataflow of esempio is
  signal sum : SIGNED(31 downto 0);
  signal msb : std_logic;
begin
  sum <=  SIGNED(in1) + SIGNED(in2);
  out1 <= std_logic_vector(sum);
  msb <= std_logic(sum(31));
  ovf <= (in1(31) and in2(31) and not msb) or (not in1(31) and not in2(31) and  msb);
end dataflow;
```

cast esplicito

Esempio da slide prof. Antonio Miele

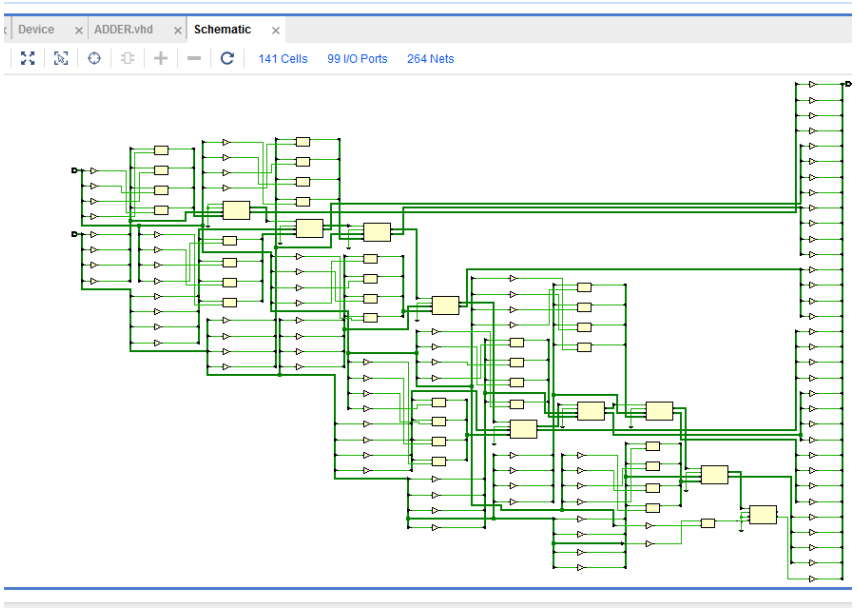
Esempio

- Altro esempio (senza OVF)

```
20
21
22  library IEEE;
23  Use IEEE.std_logic_1164.ALL;
24  Use IEEE.NUMERIC_STD.ALL;
25
26  entity ADDER is
27      Port ( A : in STD_LOGIC_VECTOR (32 downto 0);
28             B : in STD_LOGIC_VECTOR (32 downto 0);
29             O : out STD_LOGIC_VECTOR (32 downto 0));
30  end ADDER;
31
32  architecture Behavioral of ADDER is
33
34  begin
35     O <= STD_LOGIC_VECTOR(UNSIGNED(A) + UNSIGNED(B));
36  end Behavioral;
37
```

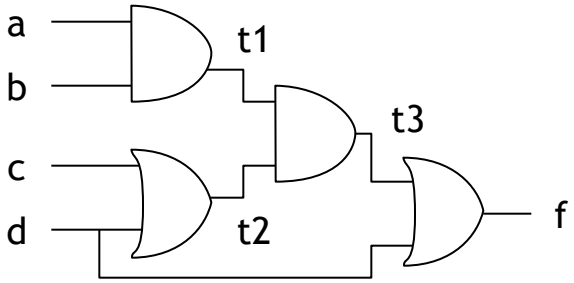
Runs | Power | Metric Results | DRC | Timing

Warning (1) | Info (13) | Status (19) | Show All



Reti Combinatorie

- Una rete combinatoria è descritta da
  - Una espressione logica
    - Nota: una rappresentazione circuitale è un modo speciale per rappresentare le espressioni



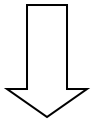
```
t1 = a and b
t2 = d and c
t3 = t1 and t2
f  = t3 and d
```

  - Una funzione booleana
    - Tabella della verità o delle implicazioni anche non completamente specificata

Reti Combinatorie

- Espressione logica
  - Una espressione logica è tradotta in una corrispondente istruzione VHDL sostituendo:
    - Alle variabili i segnali
    - Alle costanti i valori costanti tra singoli apici (o doppi apici se vettori)
    - Agli operatori logici gli operatori logici VHDL

```
f = a'(b(c+d)+ c')
```



```
f<= not(a) and (b and (c or d) or not(c))
```

  - Nota: `a`, `b`, `c`, `d` e `f` possono essere anche vettori

Reti Combinatorie

- Funzione Booleana
  - Una funzione booleana è tradotta nella corrispondente istruzione VHDL utilizzando un **assegnamento condizionale concorrente** **`with – select – when`**

Quando `a`=1 le uscite valgono `z1=z2`=0 mentre `z3`=1. Se `a`=0, `b`=1 e `c`=0 allora `z3`=0. Quando `a`=0 e `c`=1 allora `z1=z2=z3`=1.

```
with in select                    -- in è a&b&c
     out <=     "001" when "1--",
                "--0" when "010",
                "111" when "0-1",
                "---" when others;
```

Reti Combinatorie

- Funzione Booleana
  - **assegnamento condizionale concorrente `when - else`**

Quando a=1 le uscite valgono z1=z2=0 mentre z3=1. Se a=0, b=1 e c=0 allora z3=0. Quando a=0 e c=1 allora z1=z2=z3=1.

```
                    -- in è a&b&c
out <="001" when in="1--" else
      "--0" when in="010" else
      "111" when in="0-1" else
      "---" when others;
```

Reti Combinatorie

- Funzione Booleana (continua)
  - **assegnamento condizionale concorrente `when – else`**
  - La funzione può essere espressa in termini di tabella della verità o delle implicazioni, con uscite sia costanti che variabili (segnali)

a b c	g
000	0
001	1
010	0
011	0
100	d
101	d
110	e
111	f

a b c	g
000	0
001	1
01-	0
10-	d
110	e
111	f

```
                    -- in è a&b&c
g <=    '0' when in="000" else
        '1' when in="001" else
        '0' when in="01-" else
         d  when in="10-" else
         e  when in="110" else
         f  when in="111" else
        "xxx" when others;
```

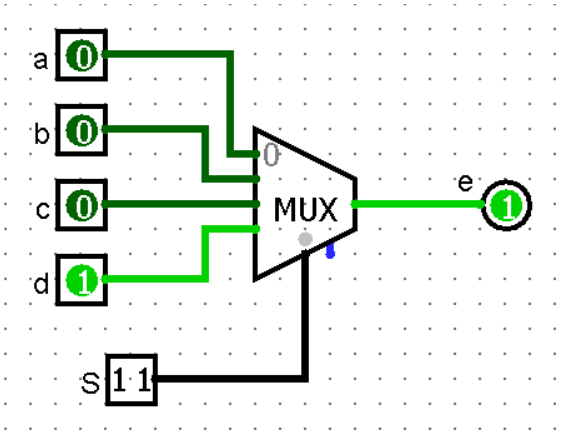
Reti Combinatorie

- Considerazioni sul costrutto **`select`**
  - Anche se sono state elencate tutte le configurazioni di ingresso la clausola **`when others`** deve essere esplicitata
    - In realtà non si esplicitano mai tutte le possibili configurazioni poiché si utilizza una logica a 9 valori (considerazione valida in simulazione)
      - Quindi, considerando per esempio un singolo segnale, indicando cosa fare per 0 e per 1 non si dice come agire per `U`, `X`, `Z`, `H`, `L`, `W`, `D`.
      - Si utilizzi '`X`' o '`D`'
  - I valori da assegnare possono essere sia costanti sia derivati da altri segnali
    - ```
      with A select
       Z<= B when '0', "01"&c when '1',
           "xxx" when others;
      ```
    - ```
      Z<= B       when A='0' else
          "01"&c  when A='1' else
          "xxx"   when others;
      ```

Reti Combina

- Esempio: Multiplexer 4

```
entity mux4 is
port(
 a, b, c, d : in std_logic;
 s : in std_logic-vector (1 downto 0);
 e : out std_logic);
end mux4;

architecture A-di-mux4 of mux4 is
-- declarative part: empty
begin
 e <= a when (s="00") else
      b when (s="01") else
      c when (s="10") else
      d when (s="11") else
      'x' when others;
end mux4_a;
```



Reti Combinatorie

- Esempio

```
entity mux is
        generic ( NumBit: integer);
        port(s0, s1          : in std_logic;
             i0, i1, i2, i3: in std_logic_vector(0 to NumBit);
             O               : out std_logic_vector(0 to NumBit));
end;

architecture prova of mux is
         signal sel: std_logic_vector(0 to 1);
begin
        sel <= s0&s1;
        with sel select
                O <=    i0 when "00",
                        i1 when "01",
                        i2 when "10",
                        i3 when "11",
                        (others => 'x') when others;
end;
```

Reti Combinatorie

- Tutte le istruzioni contenute in una architettura sono eseguite parallelamente
  - Paradigma hardware.
- Le istruzioni sono eseguite in “tempo zero”: la simulazione è solo funzionale
- La simulazione segue il modello di esecuzione “event driven”

```
Q <= R nor nQ;
nQ <= S nor Q;
```

```
Start:  R='0', S='0', Q='1', nQ='0'
passo1: R='1', S='0', Q='1', nQ='0'  '0' è schedulato per Q
passo2: R='1', S='0', Q='0', nQ='0'  '1' è schedulato per nQ
passo3: R='1', S='0', Q='0', nQ='1' nessun evento schedulato
passo2: R='0', S='0', Q='0', nQ='0' nessun evento schedulato
```

Macchine a stati

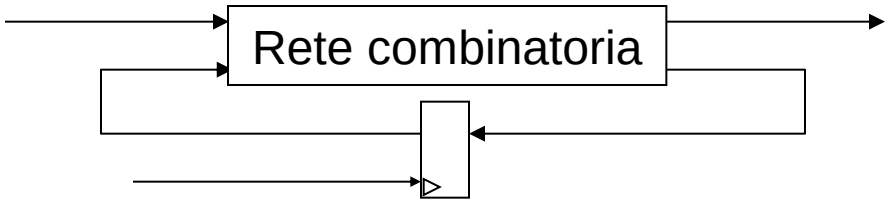
- I sistemi sequenziali sono
  - Sincroni
    - Lo stato è aggiornato quando il segnale di clock cambia
  - Asincroni
    - Lo stato è aggiornato non appena quest'ultimo cambia
      - Non è presente il clock
- Nella progettazione digitale ci si riferisce spesso a comportamenti sincroni
  - Si evitano problemi dovuti alla differenza di ritardo nel percorsi combinatori
  - Le alee dinamiche e statiche non hanno effetto
  - Più semplici i problemi di interfacciamento tra moduli
  - Maggiore indipendenza dalla temperatura di funzionamento, dalla tensione e dal processo
  - I tool di sintesi non gestiscono bene l'asincronicità

Macchine a stati

- Nella progettazione digitale ci si riferisce a comportamenti asincroni solo in casi particolari
  - Domini di clock asincroni
    - Problema molto comune
- Progettare sistemi asincroni è più complesso ma produce strutture con prestazioni migliori e consumi di potenza minori
  - "*solo gli idioti o gli esperti progettano riferendosi alla asincronicità*"

Macchine a stati

- Macchina di *Moore*
  - L'uscita è funzione del solo stato corrente
    - A volte, l'uscita è direttamente lo stato corrente
      - Es: contatori privi di rete di transcodifica
- Macchina di *Mealy*
  - L'uscita è funzione dello stato corrente e dell'ingresso
- Struttura generale di una macchina a stati
  - La rete combinatoria è scomposta in due entità: funzione stato prossimo e funzione d'uscita



Macchine a stati

- Descrivere una macchina a stati significa descrivere l'attività, parallela, delle reti combinatorie e dei registri
- A questo scopo si utilizza l'istruzione `process`

Nome del process

Sensitivity list

```
count : process (data)
     variable cnt: std_logic_vector(2 downto 0):="000";
begin
     cnt := std_logic_vector(unsigned(cnt) + 1);
     value <= cnt;
end process;
```

Dichiarazioni

Corpo del process

Macchine a stati: `process`

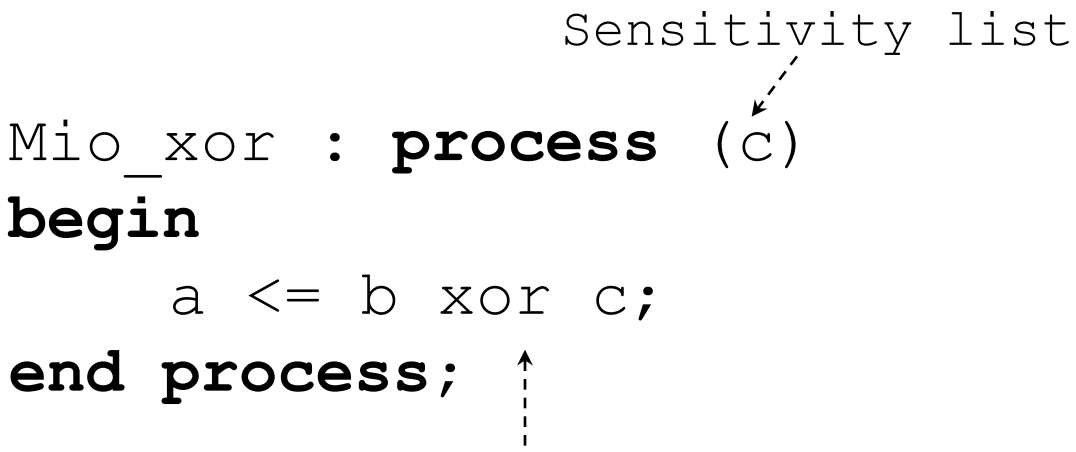
- Caratteristiche di un `process`
  - Una istruzione `process` può solo comparire nel corpo di una architettura di un modulo
  - Il `process` è eseguito parallelamente a qualunque altra istruzione presente nella architettura di un modulo
  - La istruzioni contenute in un `process` sono eseguite in sequenza (dalla prima all'ultima) e solo in corrispondenza di una variazione (evento) di un segnale specificato nella `sensitivity list`.
    - Il processo si sospende dopo l'esecuzione dell'ultima istruzione.
    - Il processo risvegliato da una variazione di un *signal* nella *sensitivity list* viene eseguito nuovamente dalla prima all'ultima istruzione
  - Il `process` può contenere istruzioni sequenziali come quelle tipicamente software
    - Le istruzioni sequenziali sono spesso più capaci di quelle parallele ma, di sovente, non hanno una corrispondente implementazione hardware

Macchine a stati: `process`

- Caratteristiche di un `process` (continua)
  - Segnali
    - I `signal` sono dichiarati solo a livello di architettura
      - assegnamento: <=
    - I segnali dichiarati in un modulo, sia di interfaccia sia interni, sono visibili all'interno di un `process`
    - I segnali devono essere usati per passare i dati da e verso un `process`
    - **Tutti** i segnali assegnati in un `process` vengono aggiornati **solo** alla terminazione del `process`
  - Variabili
    - Le `variable` sono dichiarate **solo** a livello di processo
      - assegnamento: `:=`
    - Un assegnamento ad una variabile cambia istantaneamente il suo contenuto

Macchine a stati: process

- Caratteristiche di un process (continua)
  - Attenzione: Contrariamente agli assegnamenti esterni al process, quelli interni vengono valutati solo in corrispondenza di una variazione nella sensitivity list.



L'assegnamento viene aggiornato solo quando cambia c;

ATTENZIONE: la funzionalità non rispecchia quanto previsto dallo schema di calcolo per a (non è un semplice XOR!).

Istruzioni Sequenziali

- Costrutto `if-then-else`
  - Attenzione: l'ordine di valutazione delle condizioni costituisce una direttiva di implementazione
    - Introduce priorità e possibili sbilanciamenti nei ritardi di propagazione

```
if condizione_1 then
  istruzioni_1
[elsif condizione_2
  istruzioni_2]
…
[else
  istruzioni_n]
end if;
```

Istruzioni Sequenziali

- Costrutto `case`
  - Confronto di un segnale con valori costanti

```
case segnale is
   when costante_1 =>
      istruzioni_1
   …
   when costante_n =>
      istruzioni_n
   when others =>
      istruzioni_altro
end case;
```

Istruzioni Sequenziali

- Costrutto `for`

```
for identificatore in intervallo loop
   corpo del clico
end loop;
```

Intervallo:

```
primo to ultimo
ultimo downto primo
array'range
```

- `Identificatore` è una variabile di controllo
- `a'range`, se `a` è un vettore `std_logic_vector(1 to 8)`, restituisce l'intervallo `1 to 8`
  - Consente di accedere a tutti gli elementi di un array senza conoscerne le dimensioni

Istruzioni Sequenziali

- Esempio costrutto `for`
  - Inversione dei bit nei byte (8 byte)

```
process(clock, reset)
   begin
      for i in 7 downto 0 loop
           temp(56+i) <= data(63-i);
           temp(48+i) <= data(55-i);
           temp(40+i) <= data(47-i);
           temp(32+i) <= data(39-i);
           temp(24+i) <= data(31-i);
           temp(16+i) <= data(23-i);
           temp(8+i)  <= data(15-i);
           temp(0+i)  <= data(7-i);
      end loop;
   end process;
```

Macchine a stati: esempi

- Esempio FF tipo D che commuta sul fronte di salita

```
entity D_FF is
   port( D : in std_logic;
         Clk: in std_logic;
         Q  : out std_logic);
end;
architecture FF_salita of DD_F is
begin
   ff: process(clk)
   begin
      if(clk'event and clk=1) then
         Q <= D;
      end if;
   end process;
end;
```

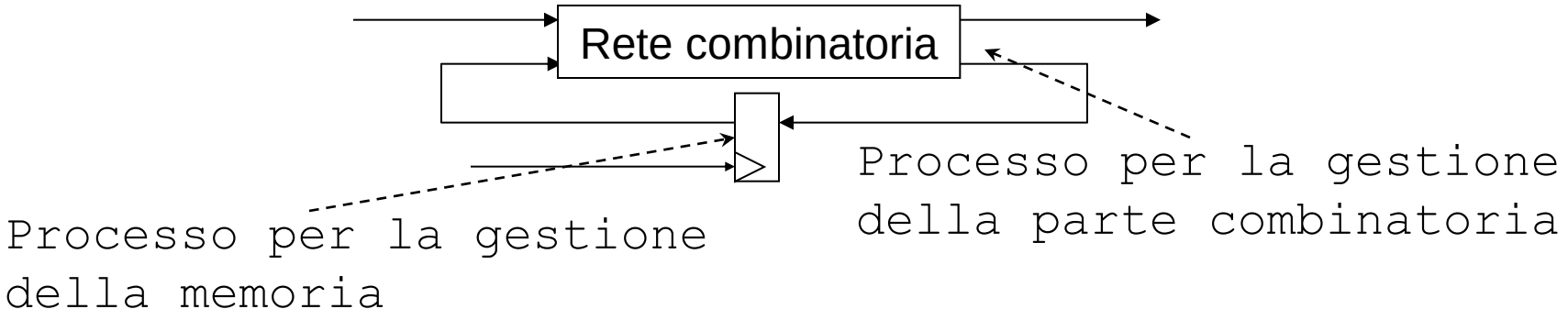
Macchine a stati: esempi

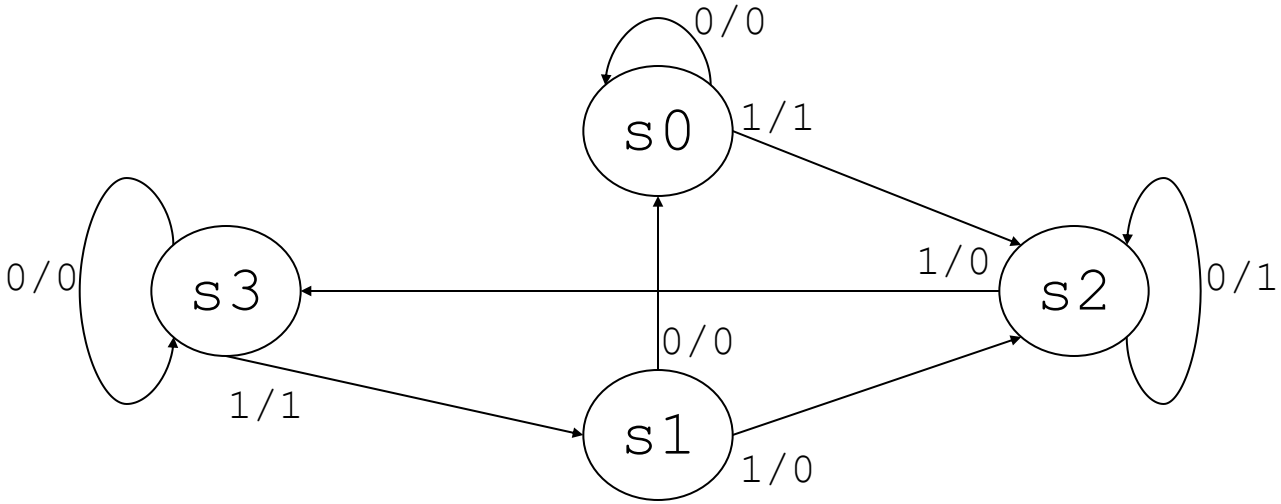
- Esempio di contatore binario naturale
  - Istanzia un contatore (non efficiente)

```
entity Counter is
   generic (N : integer := 10)
   port(clk, rst: in std_logic;
        cnt  : out std_logic_vector(0 to N));
end;
architecture count of counter is
begin
   ff: process(clk)
   begin
      if(clk'event and clk=1) then
         if (rst = 0)
           cnt <= (others =>'0');
         else
           cnt <= std_logic_vector(SIGNED(cnt) + 1);
      end if;
   end process;
end;
```

Macchine a stati: esempi

- Macchina di Mealy





Macchine a stati: esempi

- Macchina di Mealy

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

entity MEALY is
  port(X, Clock, Reset: in std_logic;
       Z: out std_logic);
end;

architecture BEHAVIOR of MEALY is
  type state_type is (s0, s1, s2, s3);
  signal CURRENT_STATE, NEXT_STATE: state_type;
begin
  -- Process per gestire la parte combinatoria
  …
  -- Process per gestire gli elementi di memoria
  …
end;
```

Macchine a stati: esempi

```
-- Process per gestire la parte combinatoria
-- Mealy
  combin: process (CURRENT_STATE, X)
  begin
    case CURRENT_STATE is
      when s0 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= s0;
        else
          Z <= '1';
          NEXT_STATE <= s2;
        end if;
      when s1 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= s0;
        else
          Z <= '0';
          NEXT_STATE <= s2;
        end if;
      when s2 =>
        if X = '0' then
          Z <= '1';
          NEXT_STATE <= s2;
        else
          Z <= '0';
          NEXT_STATE <= s3;
        end if;
      when s3 =>
        if X = '0' then
          Z <= '0';
          NEXT_STATE <= s3;
        else
          Z <= '1';
          NEXT_STATE <= s1;
        end if;
    end case;
  end process;
```

Macchine a stati: esempi

```
-- Process to hold synchronous elements (flip-flops)
 memory: process (Clock, Reset) -- reset sincrono
 begin
    if(Clock'EVENT and Clock='1') then
       if(Reset = '0') then -- active low
           CURRENT_STATE <= S0;
       else
           CURRENT_STATE <= NEXT_STATE;
       end if;
    end if;
 end process;


-- Process to hold synchronous elements (flip-flops)
 memory: process (Clock, Reset) -- reset asincrono
 begin
    if(Reset = '0') then -- active low
            CURRENT_STATE <= S0;
    else
      if(Clock'EVENT and Clock='1') then
           CURRENT_STATE <= NEXT_STATE;
       end if;
   end if
 end process;
```

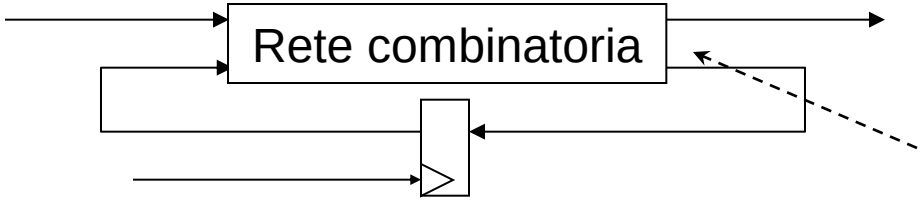
Macchine a stati: esempi

- Osservazione: La codifica dello stato viene fatta dallo strumento di sintesi
  - Codifica binaria a partire dal primo stato della lista
- È possibile forzare una codifica

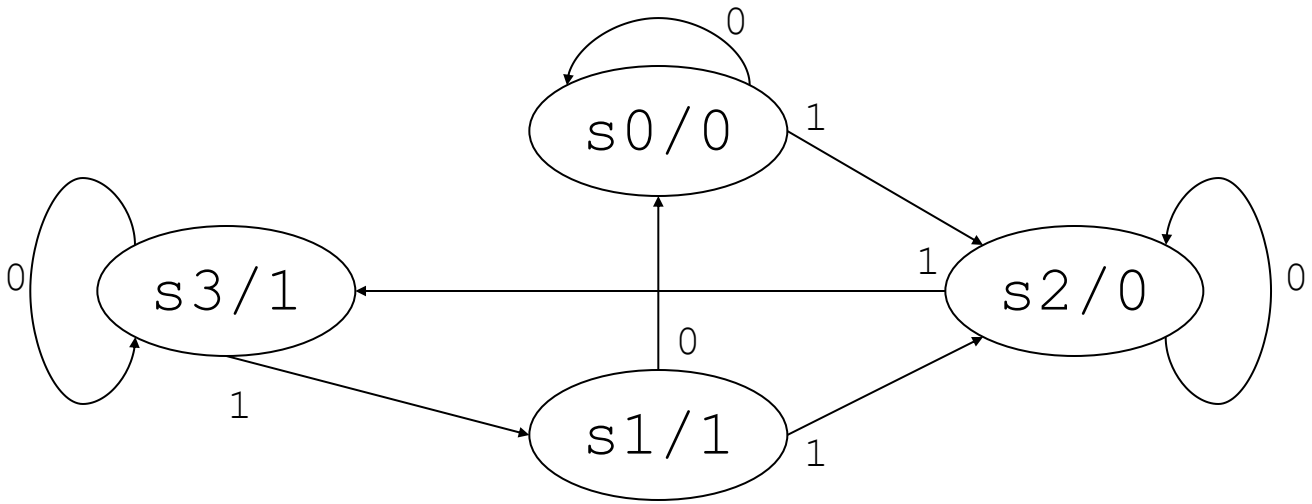
```
architecture BEHAVIOR of MEALY is
  type state_type is (s0, s1, s2, s3);
  constant s0: std_logic_vector(0 to 3):="0001";
  constant s1: std_logic_vector(0 to 3):="0010";
  constant s2: std_logic_vector(0 to 3):="0100";
  constant s3: std_logic_vector(0 to 3):="1000";
  signal CURRENT_STATE, NEXT_STATE: state_type;
begin
…
…
end BEHAVIOR;
```

Macchine a stati: esempi

- Macchina di Moore



Rispetto al caso precedente cambia solo il processo per la gestione della parte combinatoria



Macchine a stati: esempi

```
-- Process to hold combinational logic
  combin: process (CURRENT_STATE, X)
  begin
    case CURRENT_STATE is
      when s0 =>
         Z <= '0';
         if X = '0' then
            NEXT_STATE <= s0;
         else
            NEXT_STATE <= s2;
         end if;
      when s1 =>
         Z <= '1';
         if X = '0' then
            NEXT_STATE <= s0;
         else
            NEXT_STATE <= s2;
         end if;
      when s2 =>
         Z <= '0';
         if X = '0' then
            NEXT_STATE <= s2;
         else
            NEXT_STATE <= s3;
         end if;
      when s3 =>
         Z <= '0';
         if X = '0' then
            NEXT_STATE <= s3;
         else
            NEXT_STATE <= s1;
         end if;
    end case;
end process;
```

Macchine a stati

- Consigli per una buona macchina a stati
  - Reset sincrono
  - Non avere stati non raggiungibili
    - Evitare che si creino dei *dead-loop*
      - vengono raggiunti per guasti transitori
  - Ingressi registrati
    - Per evitare che si presentino dei cicli combinatori difficili da individuare
  - Usare nomi simbolici per gli stati
    - Ottima politica per strumenti di sintesi che consentono di applicare, mediante comando, delle politiche di codifica dello stato

Macchine a stati

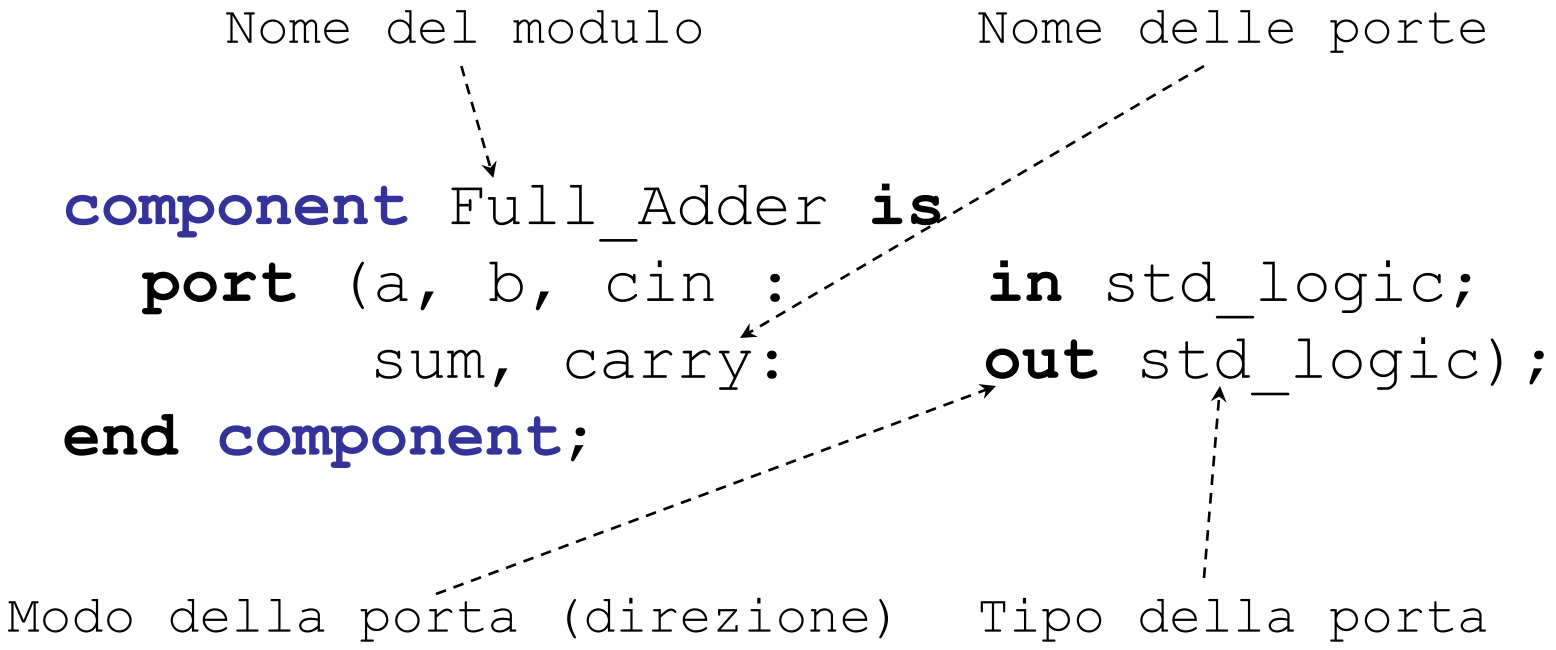
- Consigli per una buona codifica dello stato
  - Minimizzare il numero dei flip-flop può non essere una buona politica
    - Un numero ridotto di FF può avere un impatto negativo sulla funzione d'uscita
  - La codifica one-hot permette di ottenere una funzione stato-prossimo veloce e semplice
    - Utilizzare la codifica one-hot nelle FPGA poiché ne esistono molti e, se non utilizzati, sono persi.
      - Conduce ad un utilizzo efficiente delle CLB e aumenta le prestazioni dell'intero sistema.
  - La codifica one-hot garantisce distanza di hamming due tra gli stati
  - Per ridurre la potenza si potrebbe utilizzare una codifica gray tra gli stati adiacenti
    - Di difficile applicazione se non per i contatori

Gerarchia

- La gerarchia viene descritta mediante una o più rappresentazioni funzionali a livello strutturale
  - Alcuni/tutti dei moduli realizzati sono raggruppati in un unico modulo; quest'ultimo individua il modulo ad essi gerarchicamente superiore
  - I moduli utilizzati sono chiamati componenti
- Tre aspetti relativi ai componenti utilizzati
  - Dichiarazione
    - Definisce i componenti che dovranno essere collegati fra loro
  - Configurazione
    - Associa alla interfaccia l'architettura a cui si intende riferirsi
      - Solo nel caso che un modulo abbia più di una architettura
  - Istanziazione
    - Creazione di una copia del modulo

Gerarchia

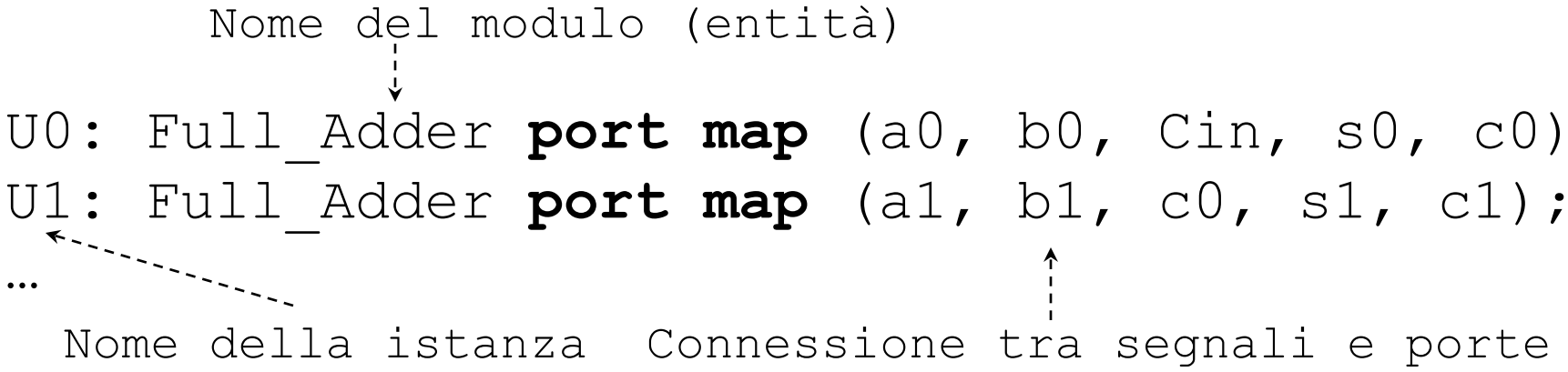
- Dichiarazione
  - Locata nella parte dichiarativa della architettura



In blue le differenze ripetto a *entity*

Gerarchia

- Istanziazione
  - Locata nel corpo della architettura
- *Port map* posizionale

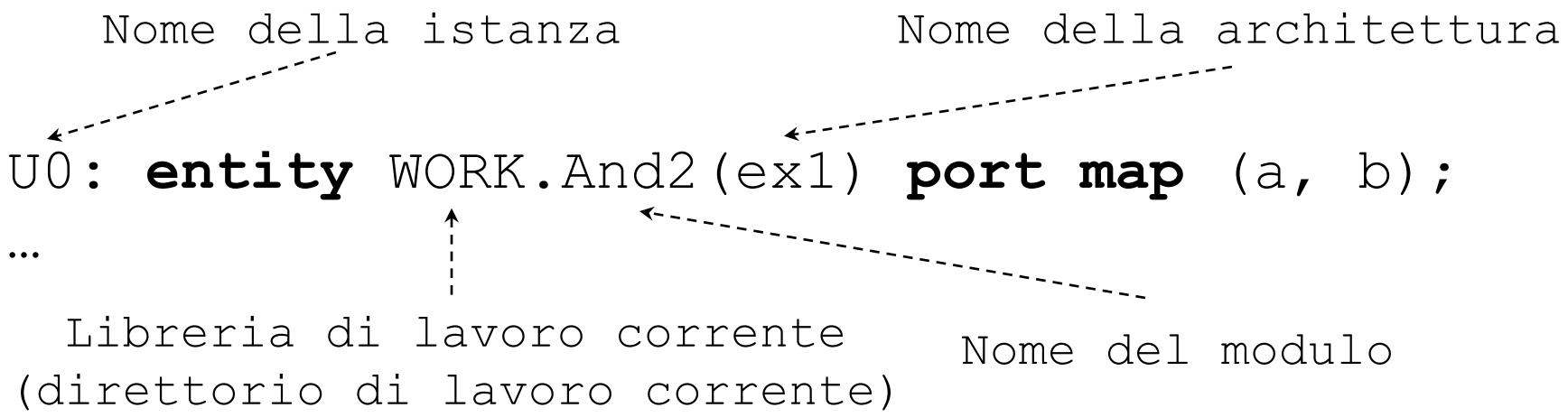


- *Port map* nominale

```
U0: Full_Adder port map
    (a=>a0,b=>b0,cin=>Cin,sum=>s0,carry=>c0);
```

Gerarchia

- Istanziazione diretta (VHDL ’93)
  - Dichiarazione e istanziazione sono fatti contemporaneamente
    - Codice più semplice e chiaro ma non compatibile con tutte le versioni degli strumenti di sintesi
    - Preferito per net-list semplici



Gerarchia

- Se non tutti gli ingressi e/o le uscite di un componente sono utilizzate:
  - Associare agli ingressi un valore costante
  - Utilizzare la parola riserva **open**
- Utile per la realizzazione di funzionalità parametriche

Gerarchia

- Esempio

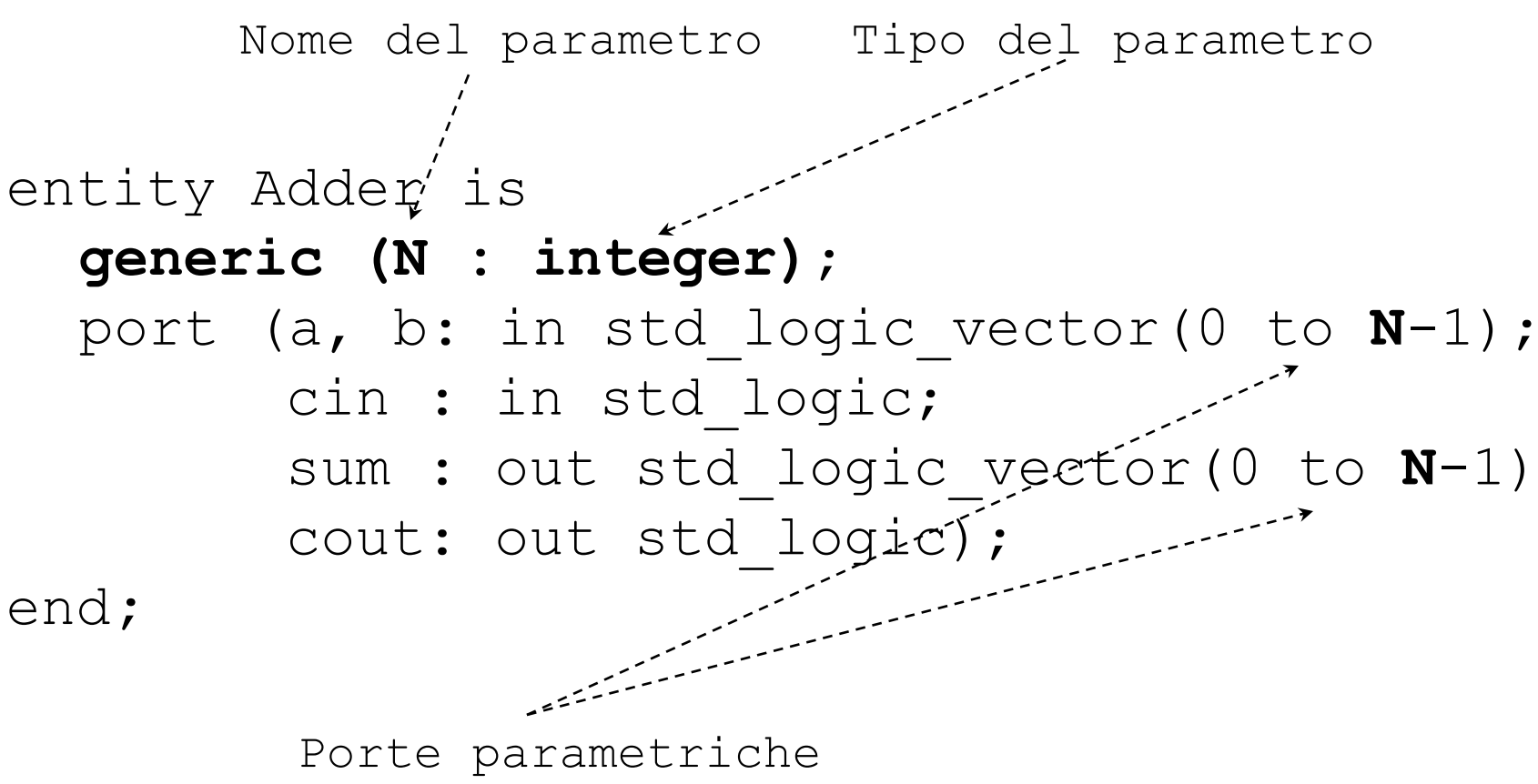
```
entity esempio is
   port(   x,y : in std_logic; inv: in std_logic :='0';
           a, o; out std_logic);
end entity esempio

architecture ex di esempio is
begin
   a <= (y and (x xor inv)) or (inv and not y);
   o <= (not x and (y xor inv)) or (x and not inv);
end architecture ex;
…

Uo: entity WORK.esempio(ex) port map (x,y,open,a,open);
```

Strutture Parametriche

- La parametrizzazione consente di realizzare modelli generali
- Due aspetti
  - Generalizzare l'interfaccia del modulo; tale informazione si propaga all'interno del modulo
    - Costrutto `generic`
  - Generalizzare l'architettura del modulo
    - Costrutto `generate`
      - Il costrutto generate ha l'obiettivo meno ambizioso di replicare moduli dello stesso tipo; tale proprietà può essere sfruttata vantaggiosamente per la realizzazione di architetture parametriche

Strutture Parametriche

- Costrutto `generic`



Strutture Parametriche

- Esempio

```
library IEEE;
use IEEE.std_logic_1164.all;

entity shifter_left_N_Dimop is
   generic ( N      : integer;
             Dim_sh: integer );
   port (in_sh  : in std_logic_vector (Dim_sh - 1 downto 0);
   out_sh : out std_logic_vector((Dim_sh - 1 + N) downto 0));
end shifter_left_N_Dimop;

architecture rtl of shifter_left_N_Dimop is
signal zeros: std_logic_vector (N-1 downto 0);
signal zero: std_logic;
begin
    zero   <='0';
    zeros  <=(others=>zero);
    out_sh <= in_sh & zeros(N-1 downto 0);
end rtl;
```

Strutture Parametriche

- Dichiarazione di un componente con `generic`
  - Equivalente alla dichiarazione priva di `generic`
- Istanziazione di un componente con `generic`
  - Oltre al `port map` si svolge una `generic map` che ha lo scopo di associare al parametro del componente un valore costante o un ulteriore parametro derivato dall'architettura in cui è istanziato

```
U1: Adder
    generic map (10 => N)
    port map (INa, INb, cin, OUT, cout);
```

Strutture Parametriche

- Esempio di dichiarazione

```
library IEEE;
use IEEE.std_logic_1164.all;

entity PROVA is
  generic ( A, B, M : integer);
  port (in_P : in  std_logic_vector (0 to A-1);
        out_P: out std_logic_vector (0 to B-1) );
end PROVA;

architecture rtl of PROVA is
component COMPONENTE_PARAMETRICO is
  generic ( N : integer;
            C : integer );
  port (in_CP  : in std_logic_vector (0 to C - 1);
        out_CP : out std_logic_vector(0 to C - 1 + N));
end component COMPONENTE_PARAMETRICO;
…
end rtl;
```

Strutture Parametriche

- Esempio di istanziazione

```
library IEEE;
use IEEE.std_logic_1164.all;

entity PROVA is
  generic ( A, B, M : integer);
  port (in_P : in  std_logic_vector (0 to A-1);
        out_P: out std_logic_vector (0 to B-1) );
end PROVA;

architecture rtl of PROVA is
 signal X: std_logic_vector (0 to A-1+M);
 …
begin
 SH1: COMPONENTE_PARAMETRICO
        generic map (C => A, N => M )
        port map (in_P,X);
 …
end rtl;
```

Strutture Parametriche

- Costrutto `generate`
  - Utilizzato durante l'istanziazione di moduli per realizzare architetture con una struttura ripetitiva
    - Es: architetture bit-wise
  - Generazione condizionale

```
Label: if condizione generate
          [regione dichiarativa (VHDL'93)]
       begin
          {istruzioni concorrenti}
       end generate [label];
```

Strutture Parametriche

- Costrutto `generate` (cont.)
  - Generazione iterativa

```
Label: for indice in intervallo generate
          [regione dichiarativa (VHDL'93)]
       begin
          {istruzioni concorrenti}
       end generate [label];
```

Intervallo:

```
primo to ultimo
ultimo downto primo
array'range
```

Strutture Parametriche

- Esempio di registro a scorrimento SIPO a 8 bit

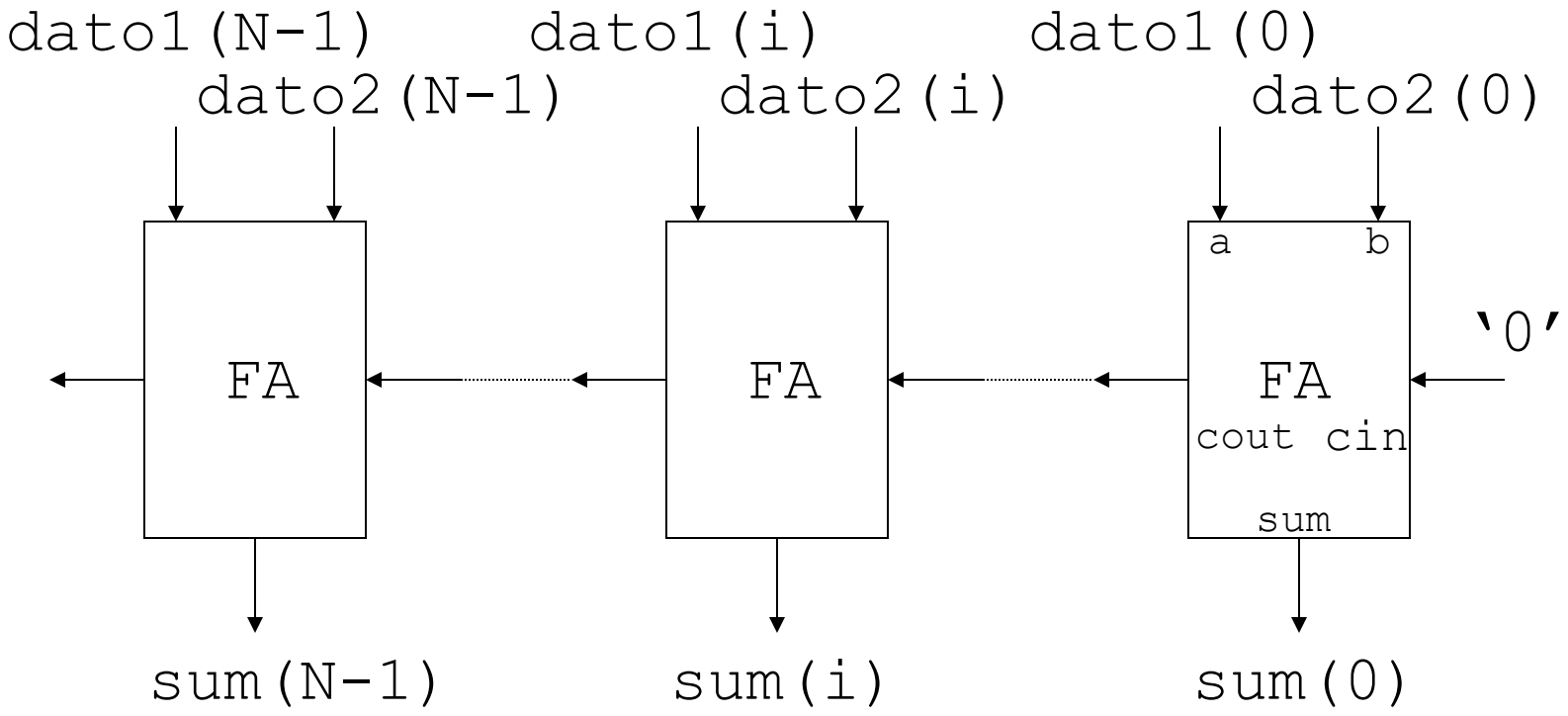
```
L : for i in 0 to 7 generate
    begin
    R0: if (i=0) generate
        begin
          U1:component DFF
             port map(CLK=>ck,D=>din, Q=>dout(i));
        end generate;
    RI: if (i/=0) generate
        begin
          U1:component DFF
             port map(CLK=>ck,D=>dout(i-1), Q=>dout(i));
        end generate;
    end generate;
```

Strutture Parametriche

- Funzioni e operatori per il dimensionamento parametrico (e non solo)
  - `abs(…)`
    - Valore assoluto – `abs(A-B)`; applicabile a qualunque tipo numerico
  - `mod`
    - `A= B*N +(A mod B)`; applicabile al tipo integer
  - `rem`
    - `A=(A/B)*B + (A rem B)` ; applicabile al tipo integer
  - `* ; /`
  - `**`
    - Esponenziale - `N**2` ; l'operando a sinistra deve essere integer o virgola mobile, l'operando a destra (esponente) deve essere integer

Esempio

- Sommatore Parametrico (Rypple Carry)
  - Schema di principio (il bit N-1 è MSB)



Esempio

- Sommatore Parametrico (Rypple Carry)
  - Full-Adder

```
library IEEE;
use IEEE.std_logic_1164.all;
entity FA is
    port (a, b, cin    : in std_logic;
          sum, cout    : out std_logic);
end FA;

-- description of full adder
architecture rtl of FA is
begin
    sum <= (a xor b) xor cin;
    cout <= (a and b) or (cin and a) or (cin and b);
end rtl;
```

Esempio

- Sommatore Parametrico (Rypple Carry)

```
library IEEE;
use IEEE.std_logic_1164.all;
entity Adder_ripple is
    generic (N :integer); -- N bit number
    port (dato1  : in  std_logic_vector (N-1 downto 0);
          dato2  : in  std_logic_vector (N-1 downto 0);
          somma  : out std_logic_vector (N-1 downto 0));
end Adder_ripple;

-- description of full adder
architecture rtl of Adder_ripple is
   component FA is
       port (a, b, cin    : in std_logic;
             sum, cout    : out std_logic);
   end component FA;
   …
```

Esempio

- Sommatore Parametrico (Rypple Carry)

```
signal carry: std_logic_vector (N downto 0);
signal sum: std_logic_vector (N-1 downto 0);
signal zero: std_logic;

begin
 zero<='0';
 FA_f: FA port map
      (dato1(0),dato2(0),zero, sum(0), carry(0));
 adder_ripple:
   for i in 1 to N-1 generate
     FA_i: FA port map
           (dato1(i),dato2(i),carry(i-1), sum(i), carry(i));
   end generate;
 somma<=sum;
end rtl;
```

VHDL e sintesi

- I tool di sintesi sono privi di intelligenza e non esistono termini riservati per specificare se un modello è sequenziale o combinatorio
- Il problema fondamentale nella sintesi modelli VHDL è di assicurarsi che quanto prodotto corrisponda alle reali aspettative
- Uno degli errori più probabili è quello relativo alla istanziazione non desiderata di Flip-Flop e Latch
  - In particolare, un FF o un latch è istanziato se una variabile o un segnale non cambia valore per qualche periodo di tempo

VHDL e sintesi

- In linea di principio, i processi possono produrre elementi di memoria indesiderati quando:
  - È attivato da una *sensitivity list* che non considera tutte le variabili o i segnali che ne aggiornano altri e non sono prodotti nel *process*
  - Alcuni percorsi assegnano valori a variabili o segnali mentre altri no
- In pratica, i tool di sintesi riconoscono un insieme abbastanza ridotto di strutture.
  - Ci si riferisce a quest'ultime

Istanza di Latch

```
U0: process (ctrl, A) is
begin
   if (Ctrl = '1') then
      Z <= A;
   end if;
end process u0;
```

- Il processo è attivato su `A` e `Ctrl`;
- Il valore di `Z`
  - È aggiornato quando il segnale `Ctrl` assume valore `1`
  - È inalterato quando `Ctrl` assume valore `0`
- Deve essere inferito un *latch* per conservare il valore di `Z`

Istanza di Latch

```
U0: process (Ctrl, A) is
Begin
   if (Ctrl = '1') then
      W <= A;
   else
      W <= W;
   end if;
end process u0;
Z <= W; -- Z è out
```

- Il processo è attivato su `A` e `Ctrl`;
- Il valore di `W`
  - È aggiornato con `A` quando il segnale `Ctrl` assume valore `1`
  - È inalterato quando `Ctrl` assume valore `0`
- Deve essere inferito un *latch* per conservare il valore di `W`

Istanza di Latch

```
process (ctrl, a, b) is
begin
  case ctrl is
     when "00" => z <= a;
     when "01" => z <= b;
     when others => null;
  end case;
end process;
```

- Il processo è attivato su `a`, `b` e `ctrl`;
- Il valore di `z`
  - È aggiornato con quando il segnale `Ctrl` assume valore “`00`” oppure “`01`”
  - È inalterato quando `Ctrl` assume le altre configurazioni
- Deve essere inferito un *latch* per conservare il valore di `Z`

Istanza di FF

```
process (clk) is
begin
  if (clk='1' and clk'event) then
     z <= a;
  end if;
end process;
```

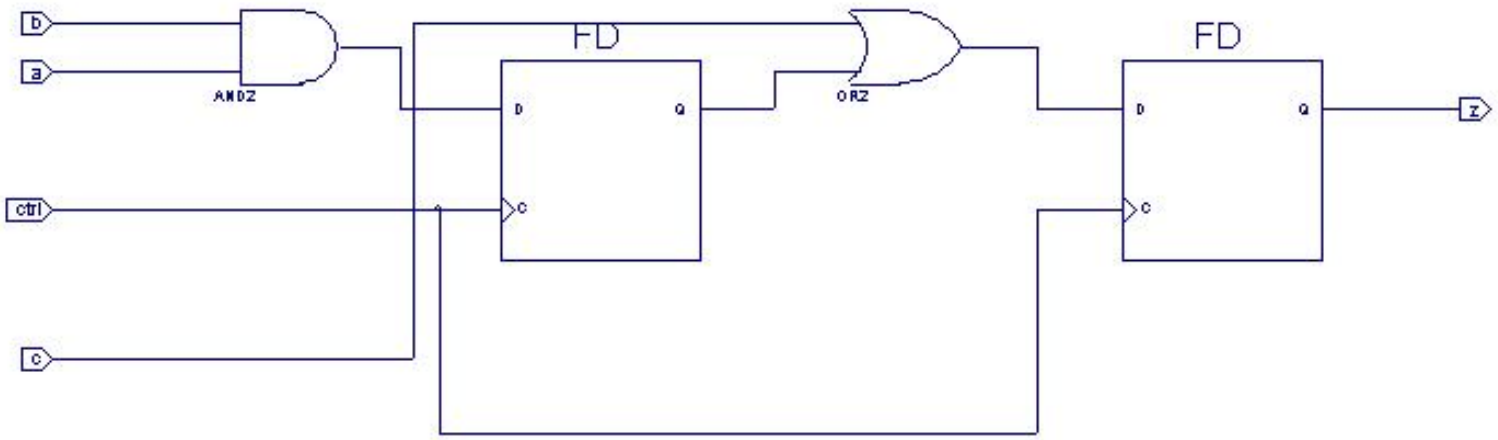
- Il processo è attivato su `clk`;
- Il valore di `z`
  - È aggiornato con `a` quando il segnale `clk` assume valore `'1'` ed è rilevato un fronte (fronte di salita)
  - È inalterato quando non si rileva un fronte di salita su `clk`
- Deve essere inferito un *FF* per conservare il valore di `Z`

Istanza di FF

```
process (clk) is
begin
  if (clk='1' and clk'event) then
     w <= a and b;
     z <= w or c;
  end if;
end process;
```

- Il processo è attivato su `clk`
- Sul fronte di salita di `clk`, i valori di `w` e `z` vengono riaggiornati con `a and b` e w_{old} `or c`
  - Si ricordi che i segnali vengono aggiornati alla fine del *process*
- Devono essere inferiti due *FF* (pipeline)

Istanza di FF

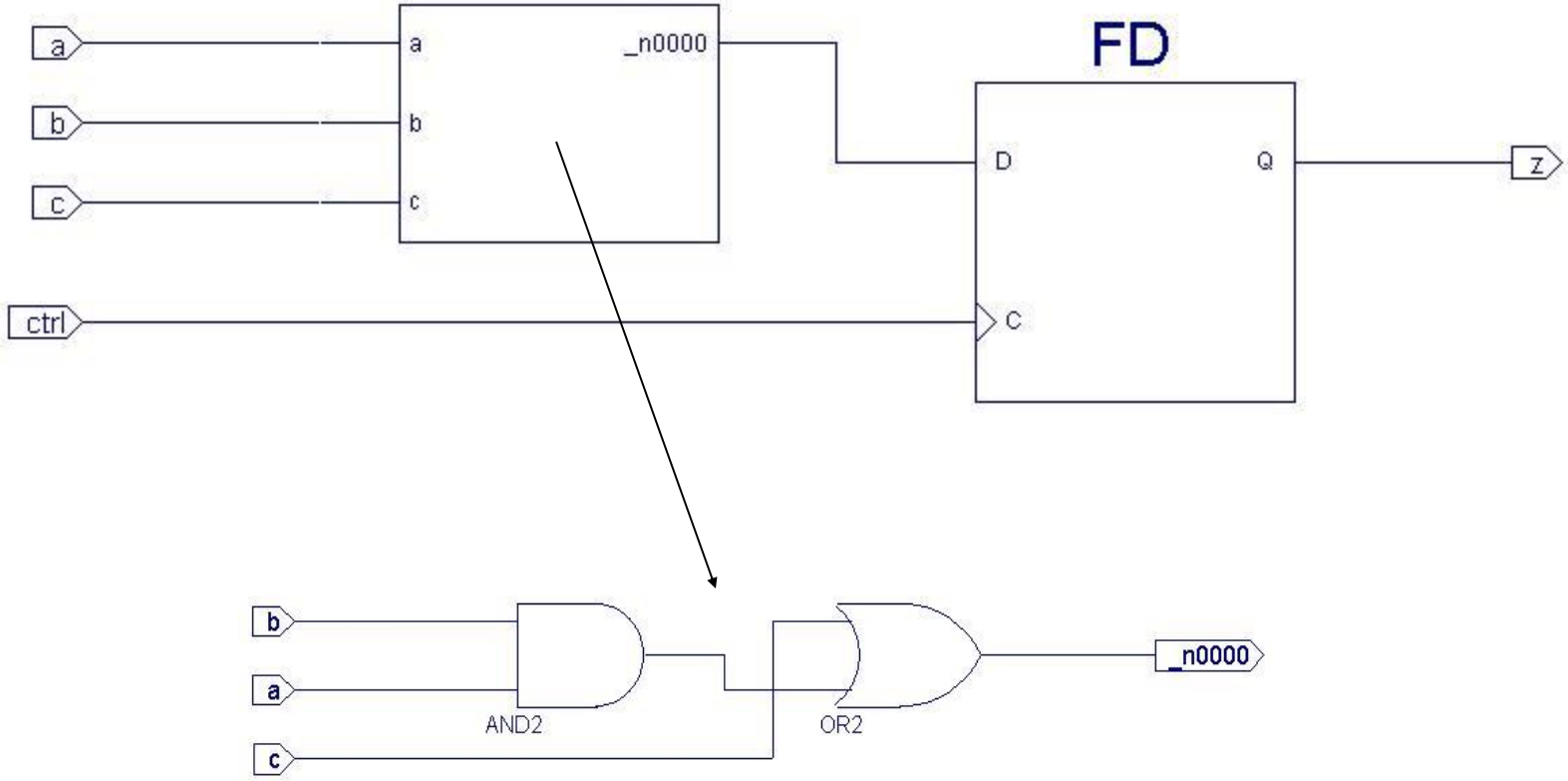


Istanza di FF

```
process (clk) is
  variable w : std_logic;
begin
  if (clk='1' and clk'event) then
     w := a and b;
     z <= w or c;
  end if;
end process;
```

- Il processo è attivato su `clk`
- Sul fronte di salita di `clk`, il valore di `z` viene riaggiornato con w_{new} `or c` dove w_{new}`= a and b`
  - Si ricordi che **solo** i segnali vengono aggiornati alla fine del *process*
- Deve essere inferito un *FF*

Istanza di FF



Reti combinatorie (Errato)

```
u0: process (a) is
begin
    z := a or b;
end process u0;
```

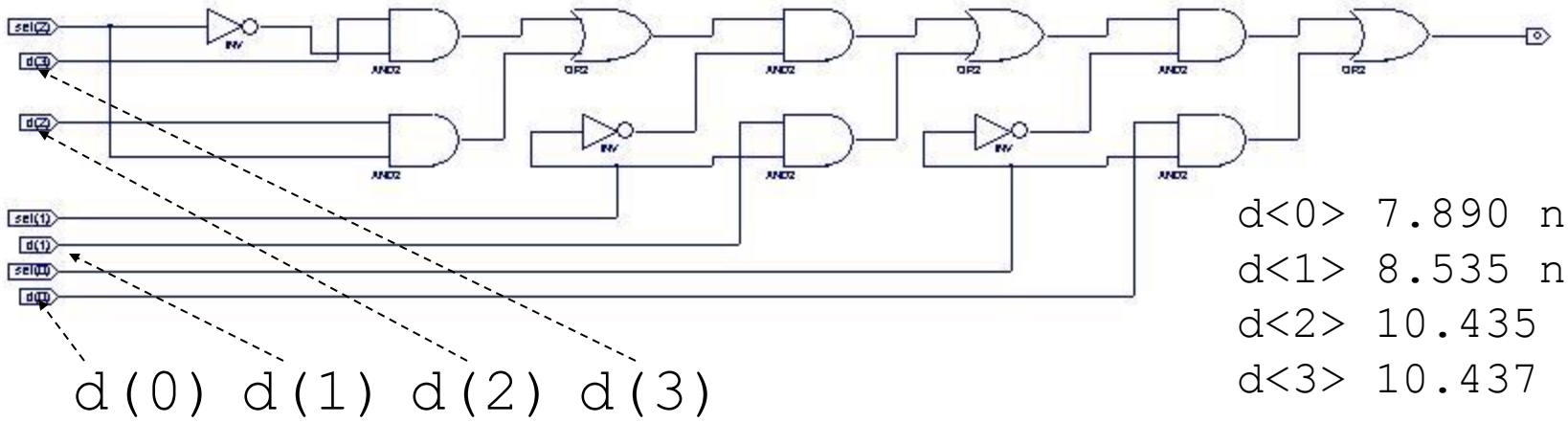
- Il processo è sensibile ad `a` ma non a `b`
- Il risultato dipende dallo strumento
  - O viene istanziata una coppia di FF: uno sensibile al fronte di salita ed uno sensibile al fronte di discesa di `a`.
    - Oltre ad una complessa rete combinatoria che genera `z`
  - Oppure viene sia segnalato un **warning/error** in fase di analisi HDL del modulo sia corretto aggiungendo implicitamente b nella sensitivity list.

Reti combinatorie

- Costrutti IF e CASE
  - Il costrutto IF crea una logica con priorità sui segnali coinvolti
    - L'IF può contenere un insieme di differenti predicati
  - Il costrutto CASE implementa una logica parallela
    - Il CASE è agisce sulla base di una espressione comune
  - Nel caso l'IF agisca su una basa di valutazione riconoscibile come comune, il risultato della sintesi può coincidere con quello prodotto dal CASE

Reti combinatorie

```
process (sel, d)
begin
  if    (sel(0) = '1') then  o <= d(0);
  elsif (sel(1) = '1') then  o <= d(1);
  elsif (sel(2) = '1') then  o <= d(2);
  else                       o <= d(3);
  end if;
end process;
```

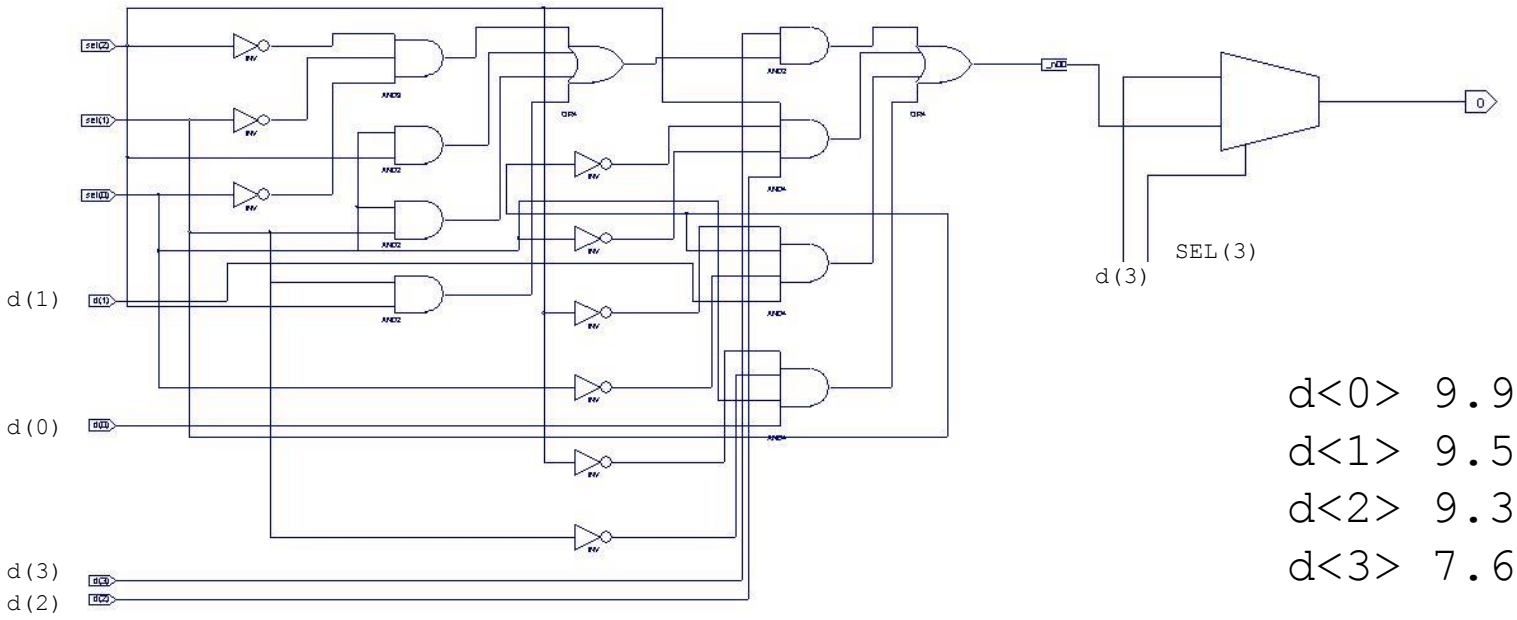



```
d<0> 7.890 ns
d<1> 8.535 ns
d<2> 10.435 ns
d<3> 10.437 ns
```

Reti combinatorie

```
process (sel, d)
begin
   if    (sel="0001") then  o <= d(0);
   elsif (sel="0010") then  o <= d(1);
   elsif (sel="0100") then  o <= d(2);
   else                     o <= d(3);
   end if;
end process;
```

```
process (sel, d)
begin
   case sel is
    when "0001" => o <= d(0);
    when "0010" => o <= d(1);
    when "0100" => o <= d(2);
    when others => o <= d(3);
   end case;
end process;
```




```
d<0> 9.957 ns
d<1> 9.537 ns
d<2> 9.347 ns
d<3> 7.653 ns
```

Regole di sintesi

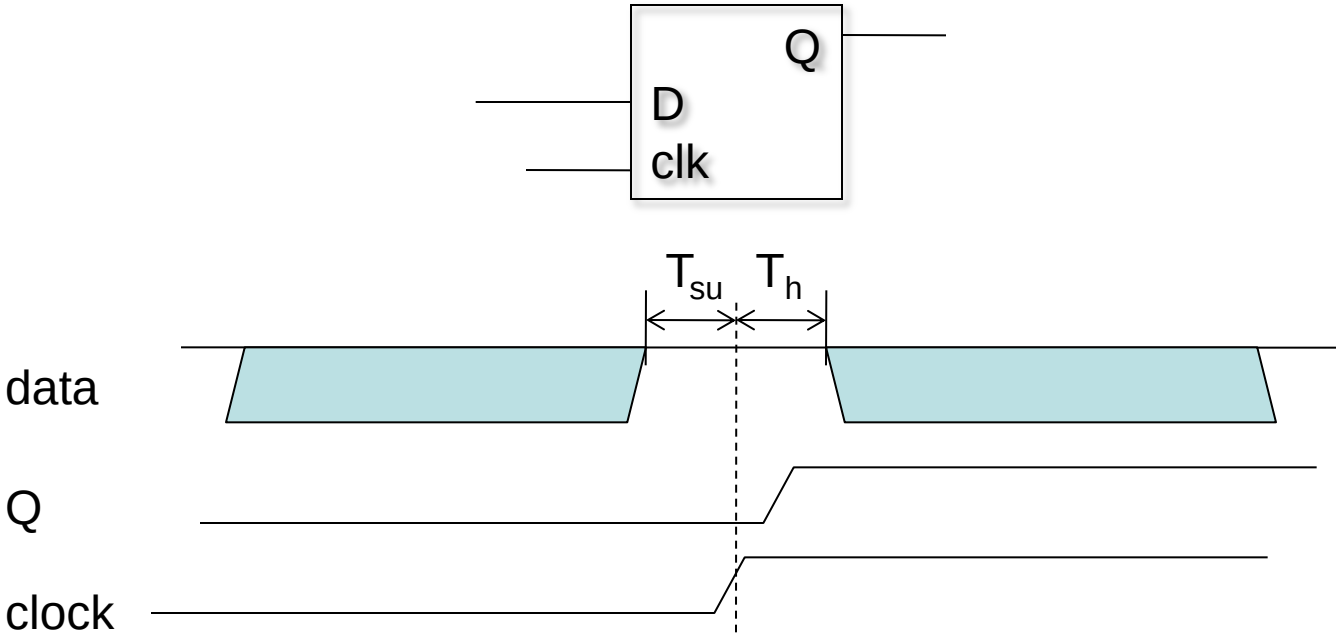
- Reti combinatorie
  - Sensitivity list: tutti i segnali primari nella parte destra degli assegnamenti e quelli utilizzati nei predicati (IF e CASE)
  - Branch: devono essere completi
- Latch
  - Sensitivity list: tutti i segnali primari nella parte destra degli assegnamenti e quelli utilizzati nei predicati (IF e CASE)
  - Branch: non completi
- Flip-flop
  - Sensitivity list: Clock ed i segnali asincroni di set e reset
    - Si utilizzano costrutti per rilevare il fronte
  - Branch: non completi

Domini di Clock

- Domini di clock
  - Un sistema è spesso realizzato in sottosistemi comunicanti
  - Tali sottosistemi utilizzano clock differenti sia di frequenze differenti che della stessa frequenza nominale
    - Attenzione: clock nominalmente dello stesso valore sono da considerarsi differenti
      - Per esempio una differenza costante del 10^{-6}% sulla frequenza porta ad avere una differenza di fase di 180 gradi (contro fase) in $50*10^{+5}$ cicli di clock che a 1MHz significa $50*10^{+5} *10^{-6}$ sec. = 5 sec.
  - Domini di clock diversi comunicano attraverso strutture specifiche per evitare questioni di **metastabilità**
  - Un clock, il suo complemento e ogni suo derivato (per esempio, il diviso-per-2) sono da considerarsi un dominio di clock.

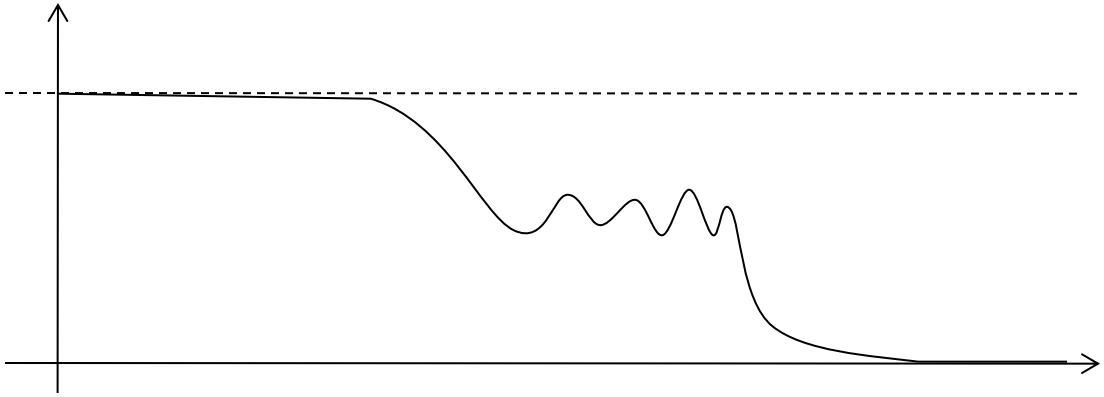
Metastabilità

- Metastabilità
  - La metastabilità è uno stato di equilibrio instabile dell'elemento di memoria provocato dalla violazione del tempo di set-up (T_{su}) e/o di hold (T_h)



Metastabilità

- La metastabilità si risolve in uno 0 o in un 1 ma in modo impredicibile.
  - Durante il periodo di metastabilità, in genere, l'uscita oscilla nell'intorno di un punto tra 0 e 1 (circa a metà) per stabilizzarsi in ad un valore definito in seguito
  - Una perturbazione porta l'elemento di memoria un uno stato di equilibrio stabile, in uno 0 o in un 1, non noto a priori



Note (VHDL 93)

- Operatore `rol`
  - Rotazione sinistra (es: `Sreg <= Sreg rol 2;`)
- Operatore `ror`
  - Rotazione destra (es: `Sreg <= Sreg ror 2;`)
- Operatore `sla`
  - Traslazione aritmetica a sinistra (es: `D <= D sla 8;`)
- Operatore `sll`
  - Traslazione logica a sinistra (es: `D <= D sll 8;`)
- Operator `sra`
  - Traslazione aritmetica a destra (es: `D <= D sra 8;`)
- Operator: `srl`
  - Traslazione logica a destra (es: `D <= D srl 8;`)

Block Ram (Xilinx) 1/3

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity memory is
   port (
      addra:  IN std_logic_vector(4 downto 0);
      addrb:  IN std_logic_vector(4 downto 0);
      clka:   IN std_logic;
      clkb:   IN std_logic;
      reset:  IN std_logic;
      dinb:   IN std_logic_vector(7 downto 0);
      douta:  OUT std_logic_vector(7 downto 0);
      web:    IN std_logic);
end memory;
```

Block Ram (Xilinx) 2/3

```
architecture Behavioral of memory is
   subtype RAM_WORD is std_logic_vector (7 downto 0);
   type RAM_TYPE is array (31 downto 0) of RAM_WORD;
   signal ram_block : RAM_TYPE;
begin
   …
   …
end Behavioral;
```

Block Ram (Xilinx) 3/3

```
architecture Behavioral of memory is
 …
begin
  -- write permitted only if reset is low
  write : process (clkb)
  begin
     if (clkb'event and clkb = '1') then
          if (web = '1') then
             ram_block(conv_integer(addrb)) <=
             RAM_WORD'(dinb);
          end if;
   end if;
  end process write;
 …
end Behavioral;
```

Block Ram (Xilinx)

```
architecture Behavioral of memory is
 …
begin
  -- write permitted only if reset is low
  …
 -- if reset is high, the first word is available on
output
  read : process (clka, reset)
  begin
   if (clka'event and clka = '1') then
            if (reset = '0') then
               douta <= "00000000";
            else
               douta <= ram_block(conv_integer(addra));
            end if;
      end if;
  end process read;
end Behavioral;
```

Bibliografia

- Materiale Generico
  - www.amontec.com/fix/vhdl_memo/
  - www.electronicsweekly.com/ → Tool Kit
    - The VHDL Tool Kit
    - The Jitter Timing Analysis Toolkit → Jitter Issues in Digital Design
      - Interessante per argomenti che riguardano la metastabilità, clock skew, ecc.
- VHDL: Tutorial e mini-reference
  - www.eng.auburn.edu/department/ee/mgc
    - VHDL tutorial
  - www.eng.auburn.edu/department/ee/mgc/vhdl.html
    - VHDL mini-reference
  - www.engineering.uiowa.edu/~digital/s2004/digital.htm
  - www.seas.upenn.edu/~ee201/vhdl/vhdl_primer.html
    - VHDL tutorial

Bibliografia

- V. Štuikys, G. Ziberkas, “Domain specific reuse with VHDL”, 1999, www.soften.ktu.lt/~stuik/knyga/
- http://www.acc-eda.com/vhdlref/index.html
- http://www.eda.org/rassp/vhdl/guidelines/1164qrc.pdf
  - Quick reference card VHDL’87

- Sintassi VHDL

- www.fys.uio.no/studier/kurs/fys329/doc/notes/VHDL87-93.pdf
- http://opensource.ethz.ch/emacs/vhdl87_syntax.html
- http://opensource.ethz.ch/emacs/vhdl93_syntax.html

Bibliografia

- Libri e Appunti
  - **M. Zwolinski, “Digital System Design with VHDL”, second edition, Pearson Education, 2004**
    ISBN 0 130 39985 X
  - C. Brandolese, “Introduzione al linguaggio VHDL”, web.cefriel.it/~brandole/courses/co-rla/course.htm
  - J.Bhasker, “VHDL primer”, Third Ed., Prentice-Hall
  - A. Rushton, “VHDL for Logic Synthesis”

Politecnico di Milano
Facoltà di Ingegneria dell'Informazione
Dipartimento di Elettronica e Informazione
Corso di Laurea in Ingegneria Informatica

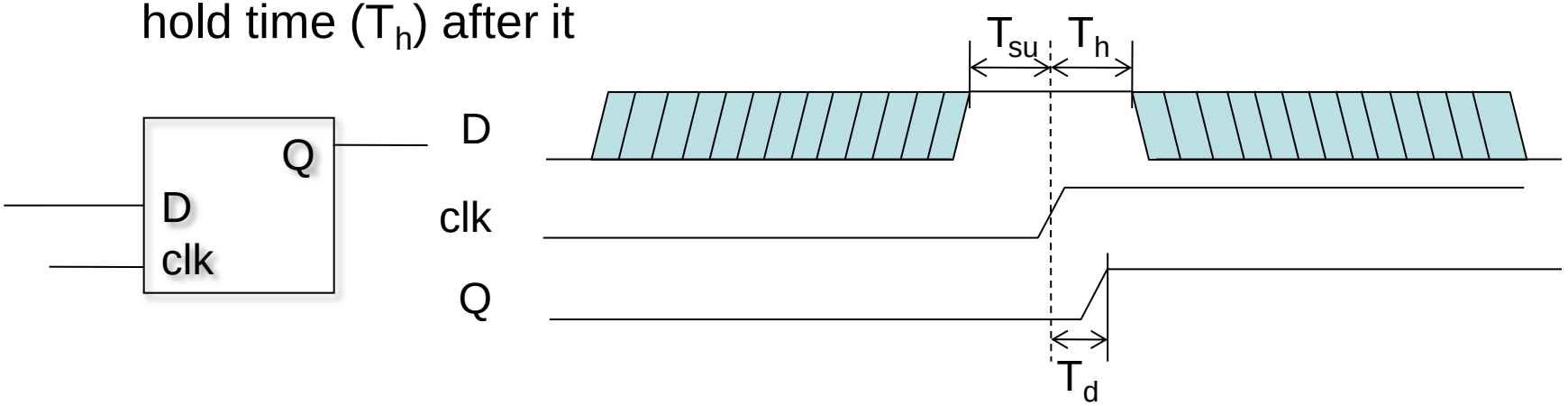
Approfondimenti e dettagli

Clock domains

- Clock domains
  - Often a system is made of intercommunicating subsystems
  - These subsystems might use different clock signals, which might have explicitly different frequencies, or the same nominal frequency but coming from independent sources
    - Warning: clock signals with the same nominal value, but coming from uncorrelated sources must be considered as with different values
      - As an example, a constant difference of 10^{-6}% (10 parts in a billion) on the actual frequency results in a relative phase inversion (180° phase shift) in just ½*100'000'000 = $50*10^{+6}$ clock cycles, which, at 1MHz would be only $50*10^{+6} *10^{-6}$ s = 50 seconds
  - A clock, its complement and any of its derivates (e.g. the divided-by-2 version) belong to the same clock domain
  - Independent clock domains must exchange data through specific structures to avoid **metastability**-related issues

Metastability

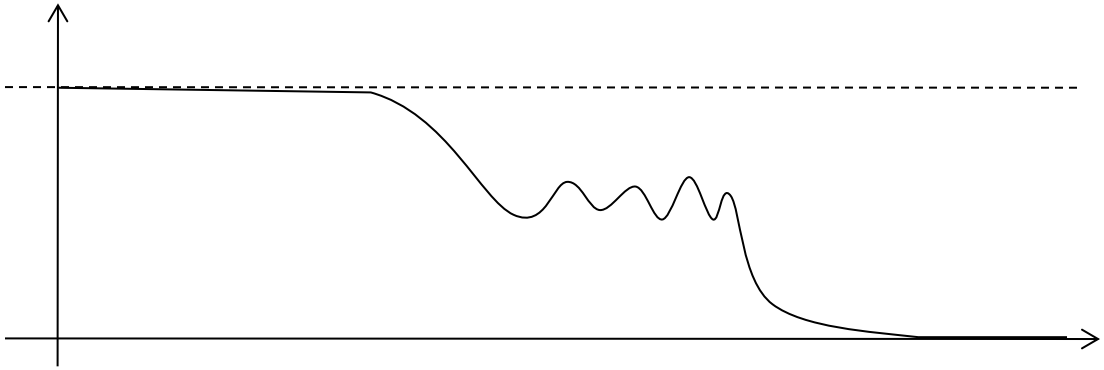
- Metastability
  - To correctly operate, FF input signal (D) must be stable for at least a set-up time (T_{su}) before the clock edge, and at least a hold time (T_h) after it



  - Under normal conditions, the updated value will be available at the Q output pin after a T_d delay after the clock edge
  - The metastability is a continuing unstable-equilibrium state of a memory element caused by the violation of its set-up or hold times (the output value will be invalid even after T_d)

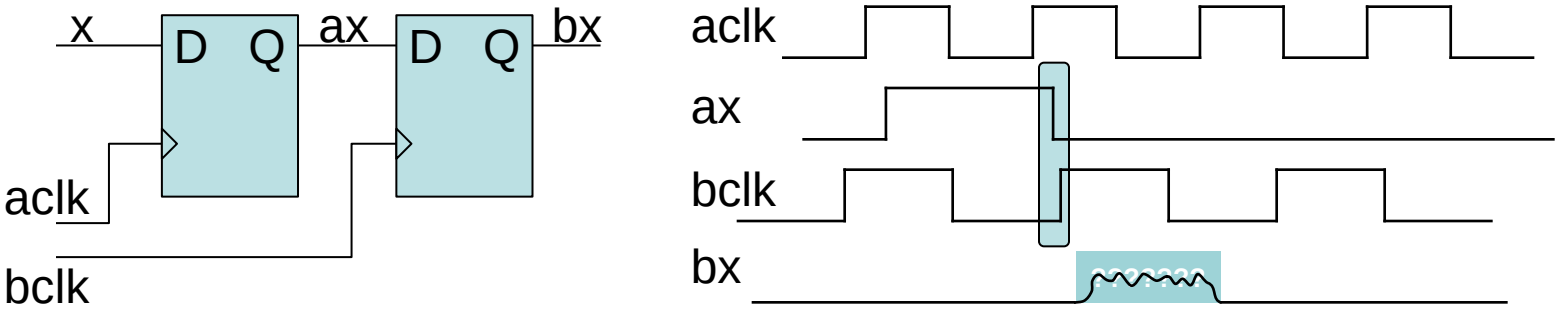
Metastability

- The metastability will resolve to either a logic 1 or a logic 0, but in an unpredictable way and in an unpredictable span of time.
  - During the metastability period, the output will usually oscillate around a point placed between 0 and 1 (about halfway), finally stabilizing to a valid value (which could be the last or the next)
  - It's the small perturbations (such as electronic noise) that will eventually force the ouptut to a valid stable output (be it 0 or 1), which can't be predicted in any way



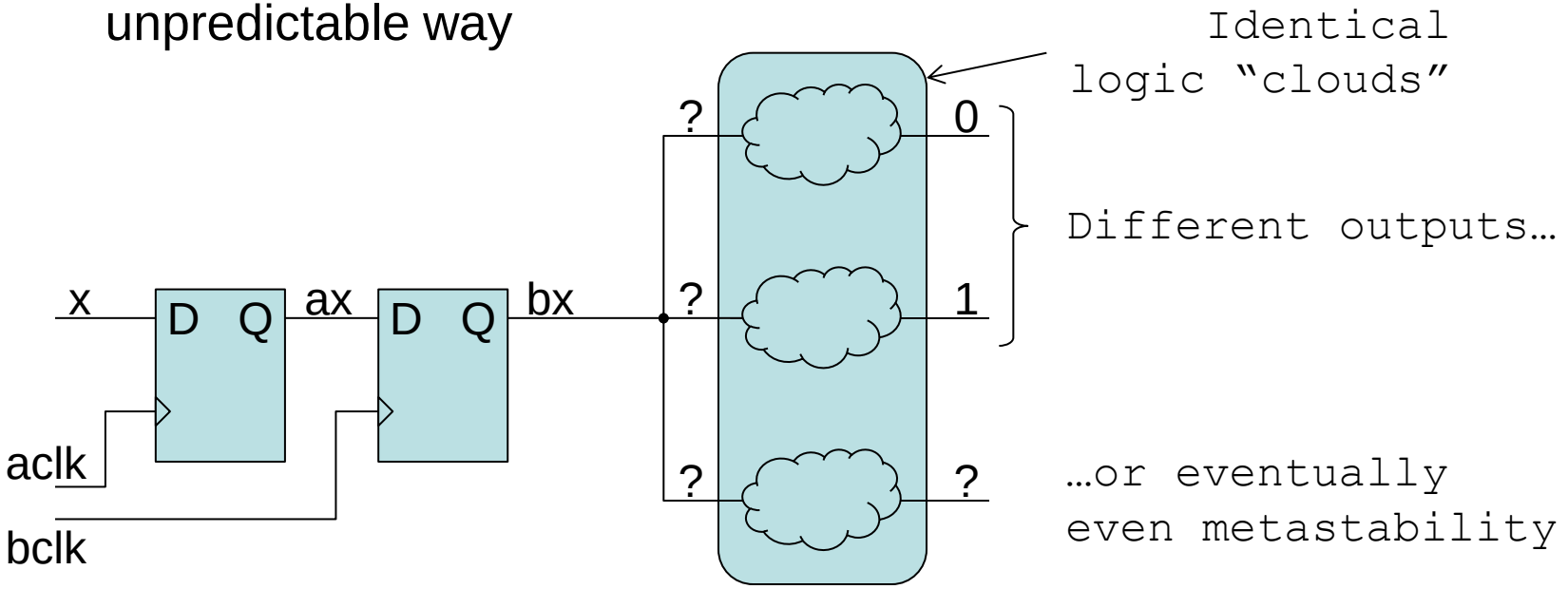
Metastability

- There is a risk of incurring into metastability whenever the change in the signal to be sampled is too close to the edge of the sampling clock
- The typical case is when you need to sample a changing data signal with a clock that does not belong to the same domain from which the data originated



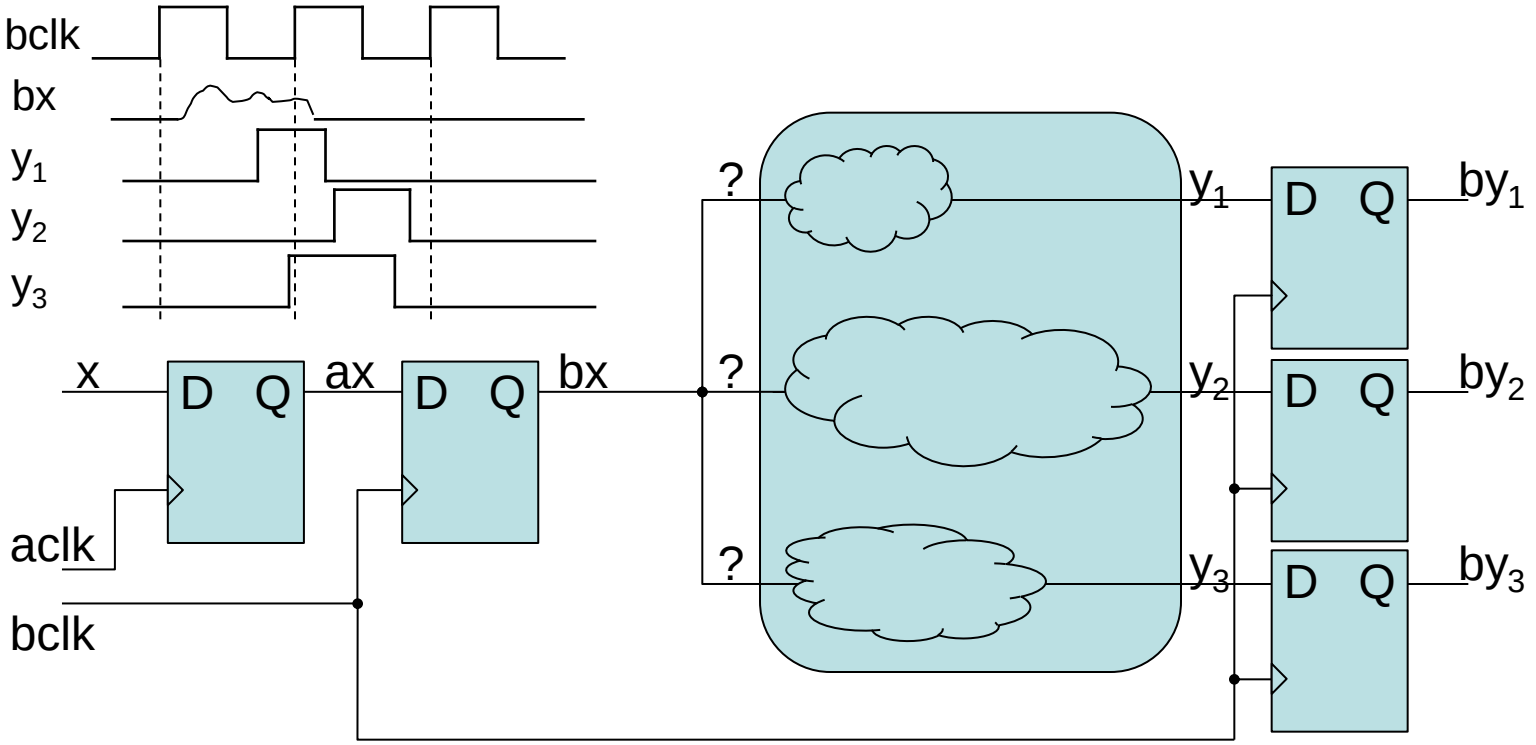
Metastability

- The problem is further aggravated if the output signal of a FF that could incur into metastability issues is fed to more than one logic stage
  - The following stages could interpret the metastable signal in different ways, therefore making the system behave in an unpredictable way



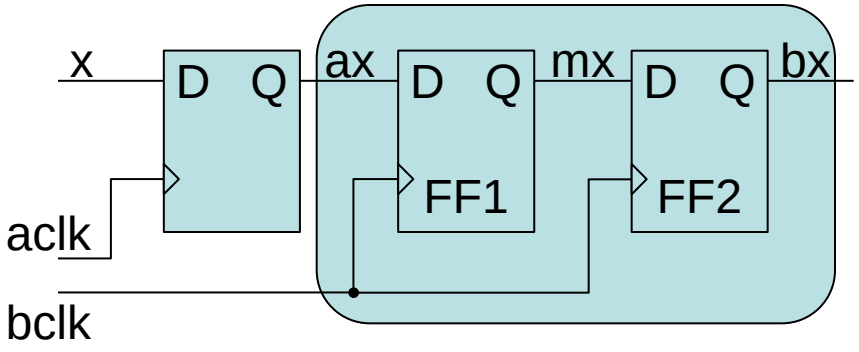
Metastability

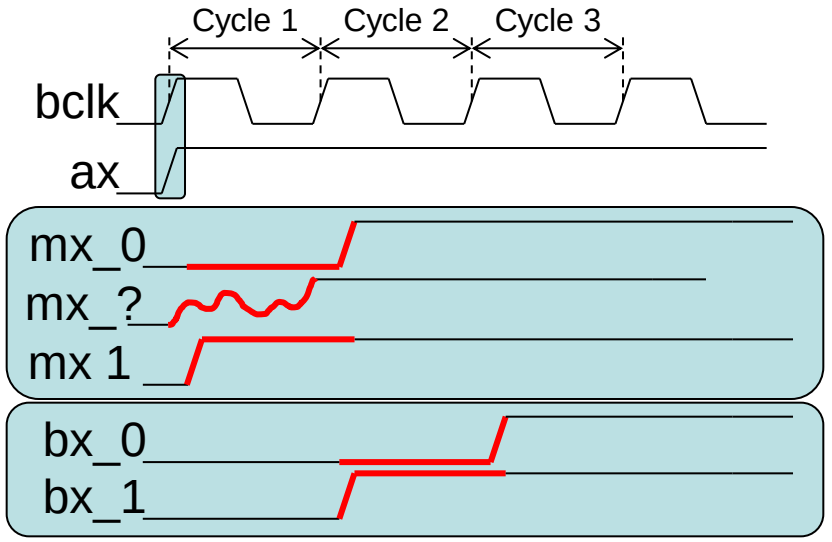
- A further worsening factor could arise if the different logic paths have also different delays before the next FF
  - This could propagate the results at different clock cycles, generate missed sampling, or even new metastabilities in the following FFs.



Metastability

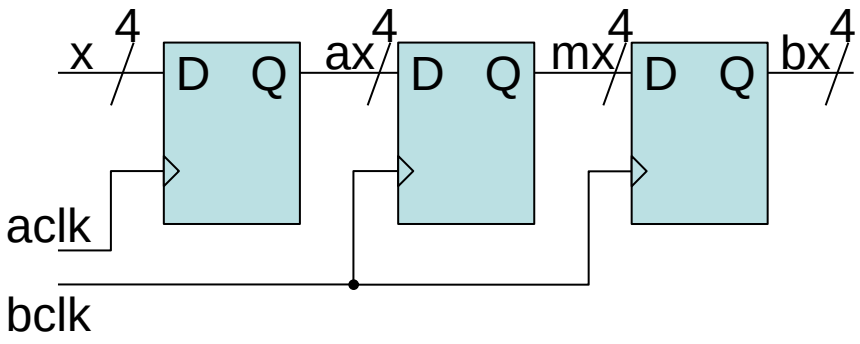
- The right way to cope with signals that cross clock domains is to use a two-FF synchronizer
  - This approach lies on the assumption that a metastable signal will settle to either a valid logic 1 or 0 **within** a single clock cycle
  - FF1 should be metastable on a change on `ax` but its output affects only FF2, and only at **the next rising edge** of `bclk`; therefore `bx` will always be valid, either with the old or with the new value.

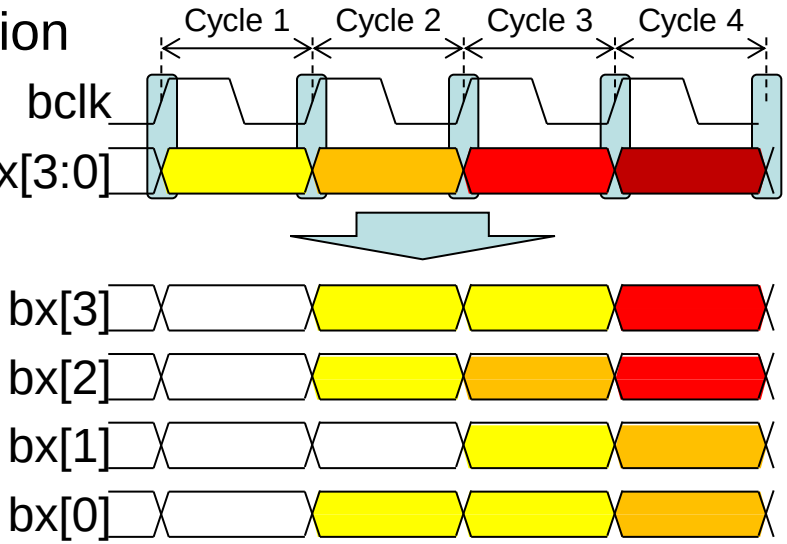




Metastability

- Since there is no way to tell if the updated value will be available after 1 or 2 clock cycles, you should pay attention to how you exchange data busses (or control busses) across clock domains
  - Different lines could be sampled at different clock cycles, therefore corrupting data/commands
  - The higher the number of bits in the bus, the higher the probability to corrupt the information





Metastability

- Failures due to metastabilities across two clock domains is estimated by a MTBF expressed as:

$$MTBF = \frac{e^{T/\tau}}{T_W \cdot f_{clk} \cdot f_{data}}$$

- where:
  - T is the metastability settling time, i.e. the time slack to allow an incoming potentially metastable signal to settle to a valid value
  - τ is the settling time constant of the FF (which is a technology-dependent constant, in the order of 100 ps = 10^{-10} s)
  - T_w is a parameter related to the window of susceptibility of the FF (usually in the order of 1 ps = 10^{-12} s)
  - f_{clk} and f_{data} are respectively the frequency of the clock signal at the synchronizer side, and the data rate of the input signal.

Metastability

- Increasing T is the most effective way to increase the MTBF, as it makes the MTBF grow exponentially

$$MTBF = \frac{e^{T/\tau}}{T_W \cdot f_{clk} \cdot f_{data}}$$

- This is the reason why we use the dual flop synchronizer:
  - A single FF fed with an asynchronous signal will repeatedly have a zero time-slack T, therefore reducing the numerator to $e^0 = 1$
  - Adding the second FF will increase T of about the same duration of the sampling clock period, so with τ = 100 ps, T_W = 1 ps, f_{clk} = 250 MHz and f_{data} = 25 MHz (and T = 4 ns), the MTBF would change from something as low as 160 µs, to over 10^{+6} years
  - Under the conditions listed above, further adding FFs would increase the final MTBF by e^{+40} (which is about 17 orders of magnitude), for each added FF

Metastability

- Whenever n bits change their values within the same clock cycle, the cumulative MTBF is given by:

$$MTBF_{cum} = \left(\sum_{i=1}^{n} \frac{1}{MTBF_i} \right)^{-1}$$

- This expression has two important implications:
  - The cumulative MTBF is always smaller than the smallest $MTBF_i$, so to increase the systems' global MTBF, we have to look for the the sections with the worst performance and enhance them
  - The global MTBF can be increased also reducing the number of simultaneously changing bits, using specific techniques, e.g.:
    - Consolidating different signals into a single signal prior to sending it
    - Using Gray codes to code information
    - Using FIFOs
    - Exchanging data with specific protocols, …

Metastability

- Consolidating different signals into a single signal prior to sending it
  - This approach can be applied only in a reduced set of situations, but it gives good results when the distinct signals carry redundant information, such as:
    - An "enable" and a "load" signal (which could be condensated into a single "load" signal)
    - Several simultaneous "enable" signals for different stages at the receiver (which could be replaced by a single signal across the clock domains, and then replicated after having safely sampled it at the receiver's side)
    - …
  - The goal is to change a single bit at time

Metastability

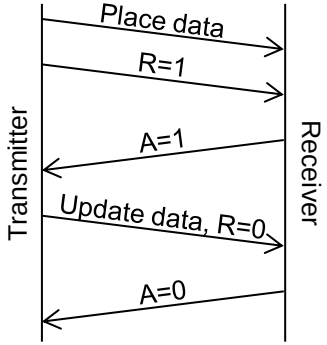
- Using Gray codes to code information
  - This approach can be applied only when the evolution of data to be exchanged is known in advance, as in:
    - Transmitting the state of a Finite State Machine, as there is a fixed set of states that can be reached from a given one
    - Transmitting the value of counters, as two adjacent values will always be consecutive to one another
    - …
  - The goal is to change a single bit at time

Metastability

- Using FIFOs
  - This approach is very versatile:
    - A FIFO (First In First Out) is a means to organize information (by queuing it), so that the order is preserved between immission and extraction
    - The two sides (the writing and the reading one) can belong to two different clock domains
    - It ensures data integrity for any data width (as long as data is not written to the FIFO when it’s already full, nor read when it’s already empty)
    - The only precaution is to correctly generate the "empty" and "full" flags
      - Both flags will be asynchronous in both clock domains, since they are the effect of both the number of reads and the number of writes

Metastability

- Exchanging data with specific protocols
  - One of these is the fully interlocked handshake
    - The data is enveloped within the protocol, which fully acknowledges each request, both for the assertion and for the negation:

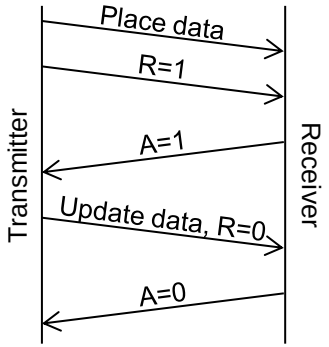


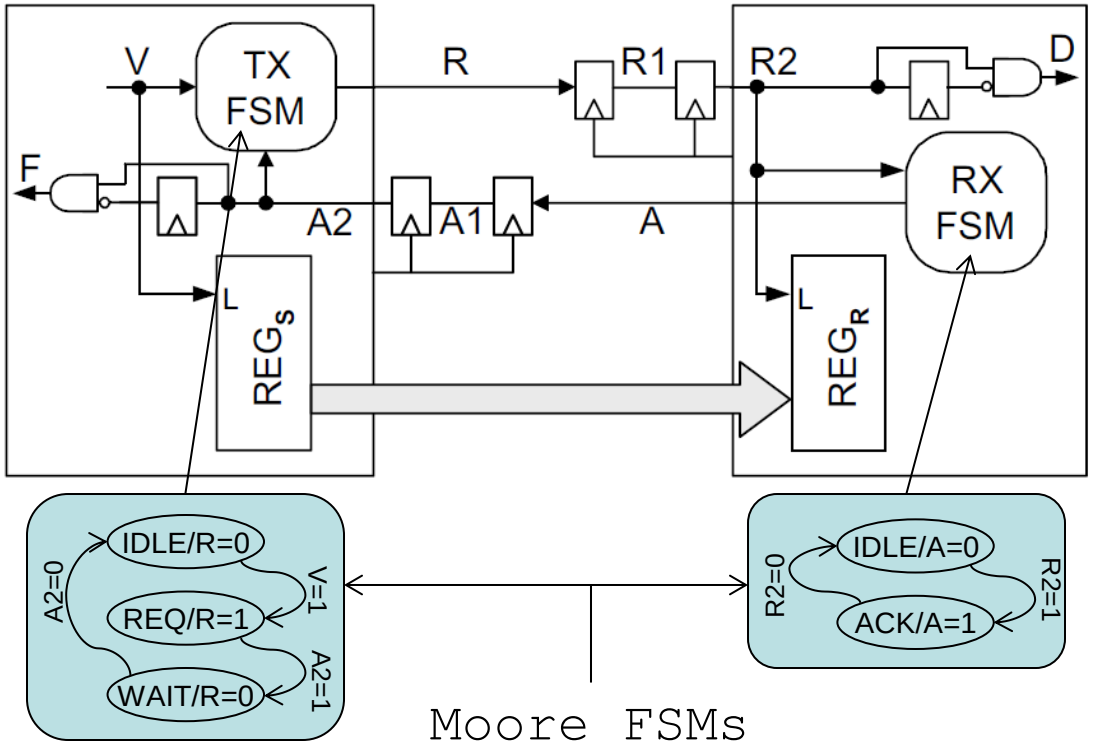
1. The sender places the data on the bus
2. The sender asserts the sending request line (R=1)
3. The receiver samples the data and acknowledges the request (A=1)
4. The sender updates the data and removes the sending request (R=0)
5. The receiver acknowledges the completion of the transfert (A=0)

- With this protocol, data is acquired only after R has been sampled twice by the two flop synchronizer at the reciver's end, so it is certainly stable
- On the other hand this protocol is highly time-inefficient, as it takes no less than 4 clock cicles on each side to complete the transaction

Metastability

- The fully interlocked handshake can be performed in hardware by implementing the following state machines:





Metastability

```
-- TRANSMITTER (inputs V, A, output R)
if rising_edge (tx_clock) then
   A2 <= A1; A1 <= A;              -- 2 flop
   A3 <= A2; F <= not A3 and A2; -- pulse
   case (tx_fsm_state) is
      when idle =>
         if (V = '1') then
            tx_fsm_state <= req;
            R <= '1';
         end if;
      when req =>
         if (A2 = '1') then
            tx_fsm_state <= waiting;
            R <= '0';
         end if;
      when waiting =>
         if (A2 = '0') then
            tx_fsm_state <= idle;
         end if;
      when others =>
         tx_fsm_state <= idle;
         R <= '0';
   end case;
end if;
```

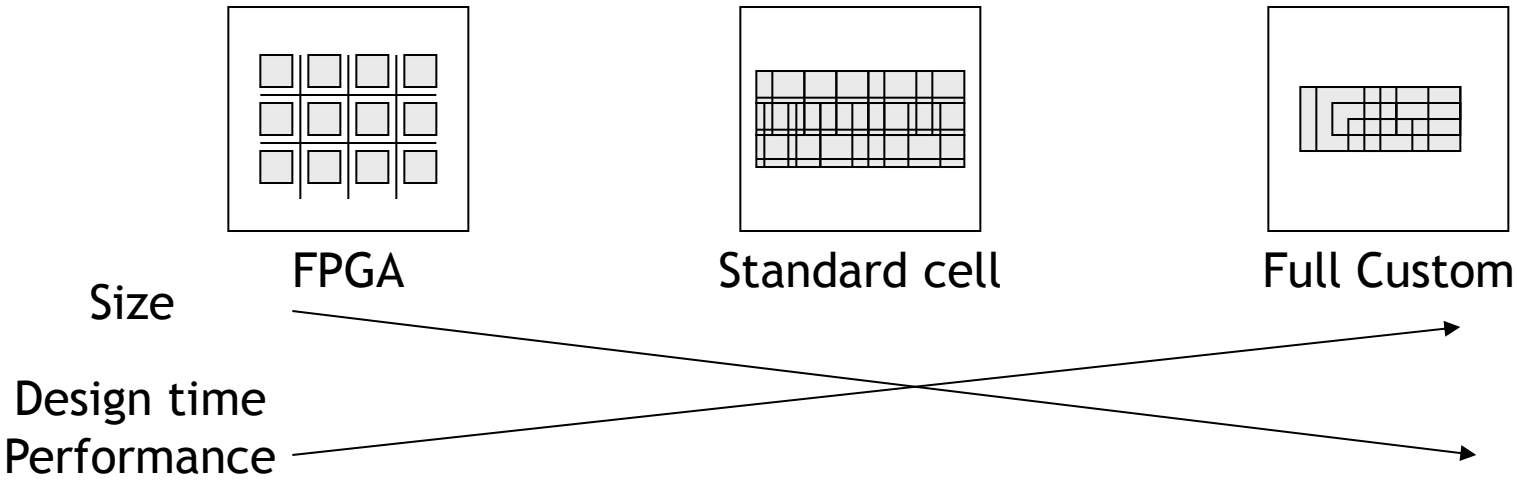
```
-- RECEIVER (input R, output A)
if rising_edge (rx_clock) then
   R2 <= R1; R1 <= R;              -- 2 flop
   R3 <= R2; D <= not R3 and R2; -- pulse
   case (rx_fsm_state) is
      when idle =>
         if (R2 = '1') then
            rx_fsm_state <= ack;
            A <= '1';
         end if;
      when ack =>
         if (R2 = '0') then
            rx_fsm_state <= idle;
            A <= '0';
         end if;
      when others =>
         rx_fsm_state <= idle;
         A <= '0';
   end case;
end if;
```

Hardware Technologies

introduction

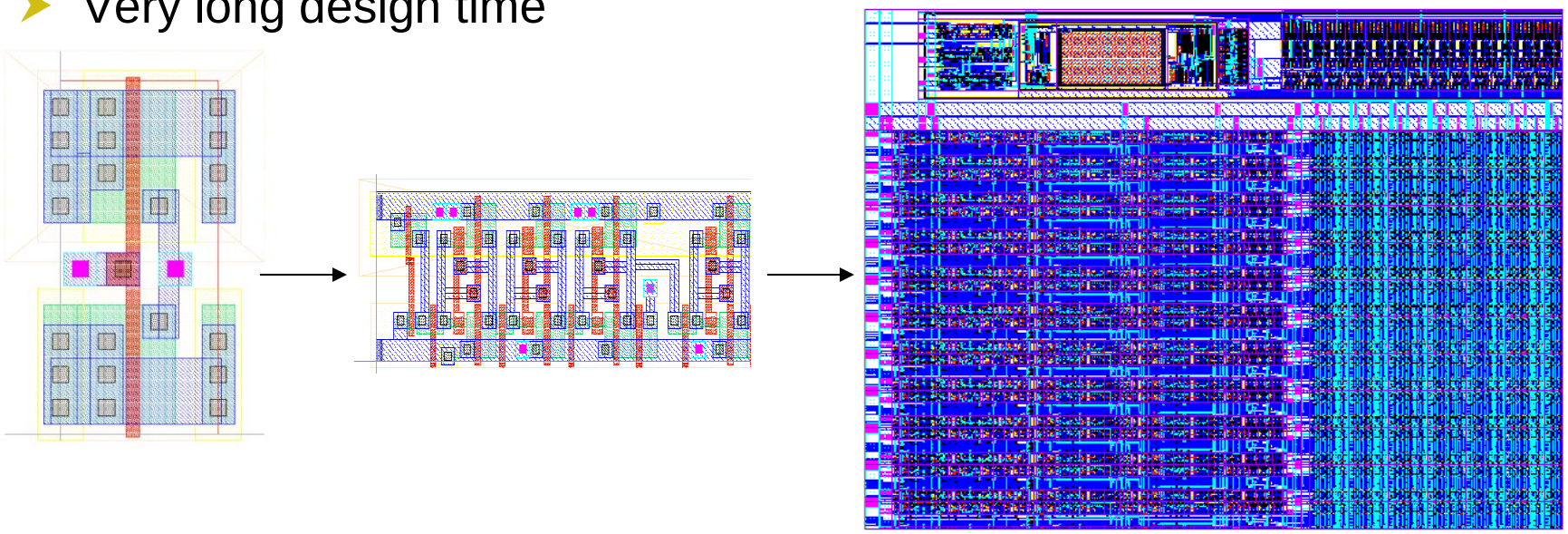
Technologies

- Full custom
- Standard cell
- Gate-array
- FPGA



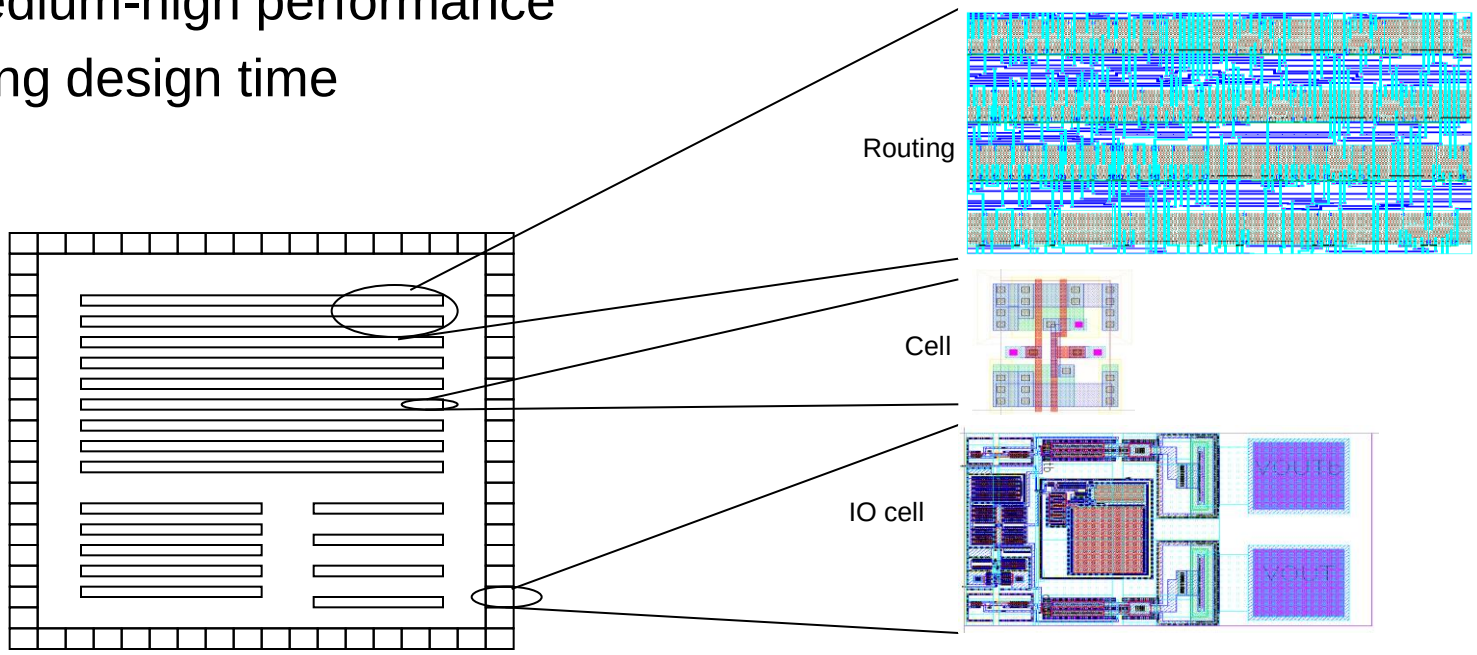
Full custom

- Hand made geometry
- Digital and analog
- Transistor level simulation
- High density
- High performance
- Very long design time



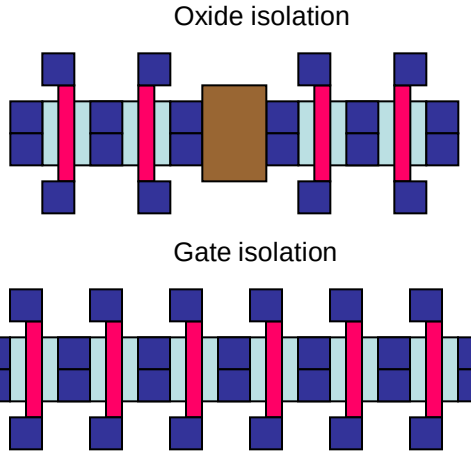
Standard cells

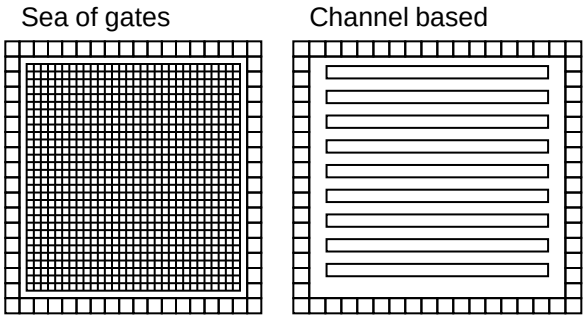
- Based on **Standard cells**
  - AN2, A2O, OR2, … Dlatch
- Mainly digital with some analog feature
- Gate level simulation (digital)
- Medium-high density
- Medium-high performance
- Long design time

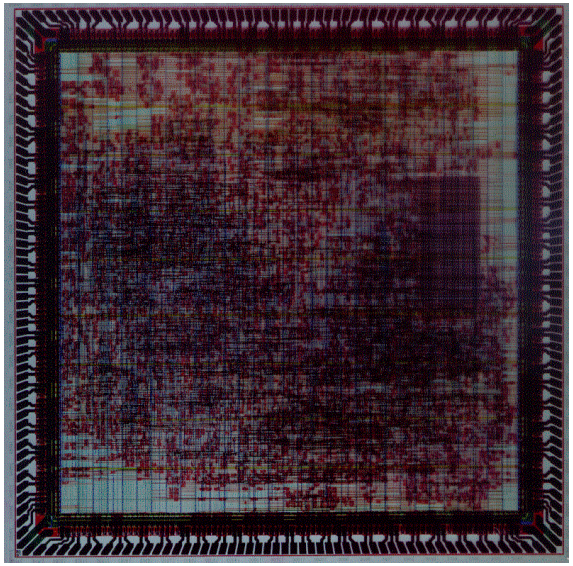


Gate-array

- Based on pre-designed Transistor
- Two types: channel based and *Sea of gates*
- Implementation of the metal layers
- Digital cell library (AN, OR, NOT, FFD,etc.)
- Gate level simulation (digital)
- Medium density
- Average performance
- Average design time

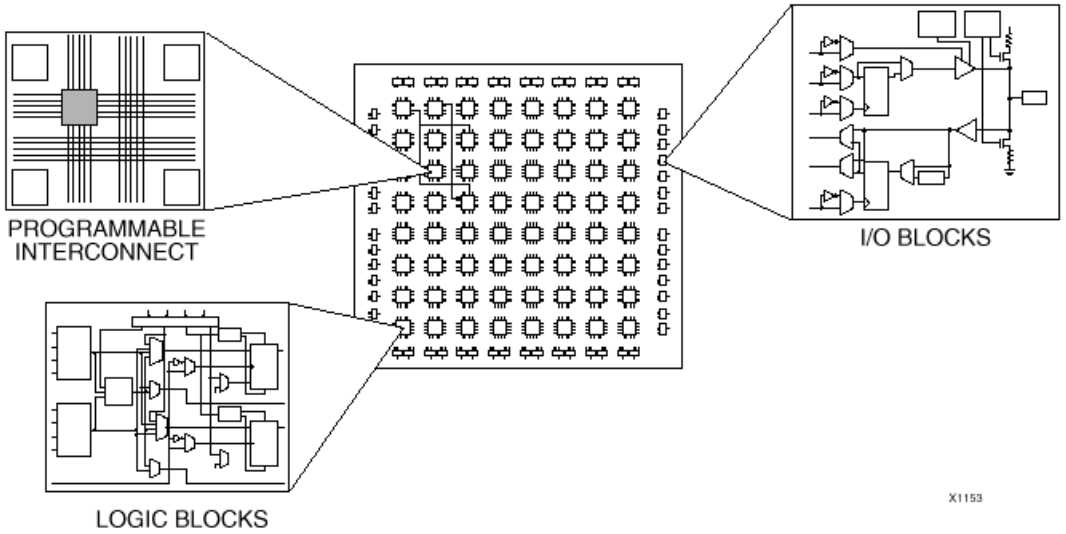






FPGA

- Programmable logic blocks
- Programmable connections among blocks
  - Technologies: SRAM, EEROM, Flash, Anti-fuse, etc
- Mainly digital
- Average- Low performance (frequancy 100MHz – 400MHz)
- Low density
- Fast reconfiguration
- Low design time
- High cost of the device



Comparison

	FPGA	Gate array	Standard cell	Full custom
Density	L	M	M	H
Flexibility	L-M-H	M	M	H
Analog	no	no	no	yes
Performance	L	M	H	VH
Design time	L	M	M	H
Design Cost	L	M/H	H	VH
Tools	simple	complex	complex	very complex
Volume	L	M	H	H
Device cost	H	M	L	L