



POLITECNICO
MILANO 1863

Reti Logiche

VHDL

Introduzione

Progettazione assistita dal calcolatore

- ➡ Al giorno d'oggi la progettazione dei sistemi digitali è svolta al calcolatore mediante appositi strumenti software
 - Si parla di **progettazione assistita dal calcolatore**
 - In inglese, **Computer-Aided Design** o **CAD**
- ➡ Il compito del progettista è quello di descrivere il sistema e «guidare» lo strumento CAD nell'analisi e nella sintesi del circuito finale
- ➡ Per descrivere la specifica del sistema si utilizza in genere un **linguaggio di descrizione dello hardware**
 - In inglese, **hardware description language** o **HDL**

Perché c'è bisogno di un HDL?

- ➔ I linguaggi di programmazione non supportano pienamente la specifica di diverse caratteristiche fondamentali dello hardware:
 - Interfacce input/output
 - Tipi di dati e specifica dell'ampiezza dei dati
 - Temporizzazione
 - Concorrenza
 - Sincronizzazione

- ➔ Lo HDL offre tutti i meccanismi necessari per descrivere tutti questi aspetti specifici dei circuiti digitali

Potenzialità di un HDL

- ➔ Lo HDL permette di specificare sistemi a diversi livelli di astrazione:
 - Livello logico
 - Porte AND, OR, NOT
 - Bistabili e flip-flop
 - Livello register-transfer
 - Addizionatori, multiplexer, registri, ...

- ➔ Lo HDL generalmente supporta diversi tipi di rappresentazione
 - **Dataflow**: descrizione mediante equazioni logiche
 - **Strutturale**: descrizione basata su uno schema a blocchi
 - **Comportamentale**: descrizione algoritmica

Utilizzi di un HDL

- ➔ Nel corso della progettazione di un sistema hardware, lo HDL è utilizzato per diversi scopi:
 - Descrizione e documentazione formale del comportamento del componente
 - Simulazione del sistema
 - Sintesi circuitale del sistema

Descrizione e documentazione

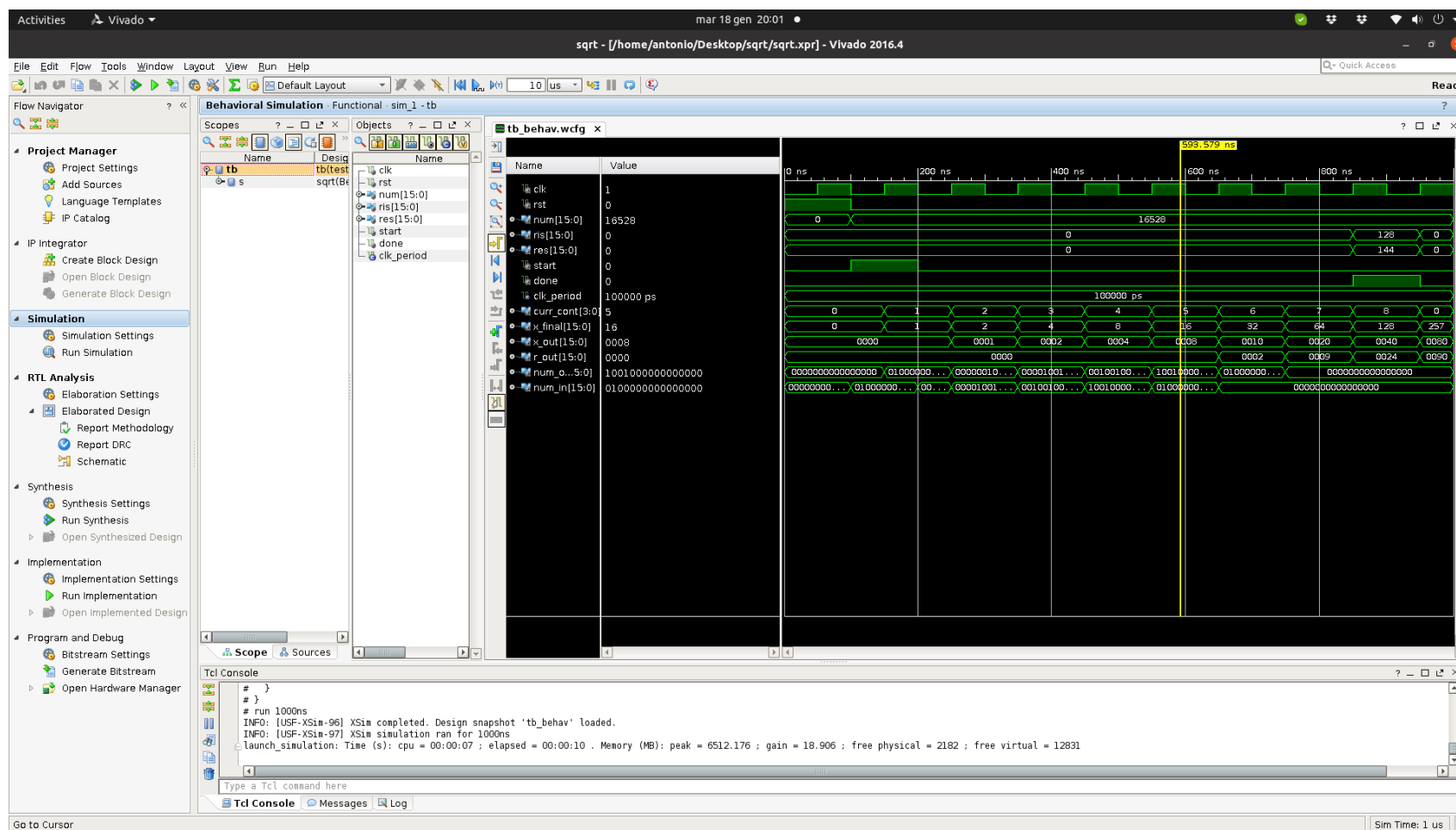
- ➡ Una delle funzioni del HDL è quella di **descrivere e documentare** il funzionamento di un sistema in modo chiaro, formale ed inequivocabile
- ➡ Potrebbe essere la descrizione di un sistema già in funzione
- ➡ Non è detto che questo sistema debba essere realizzato
- ➡ Alle volte è **IMPOSSIBILE** la realizzazione fisica della specifica
- ➡ Potrebbe essere un modo per descrivere gli stimoli da impiegare per testare un circuito

Simulazione

- ➡ Un sistema descritto in HDL viene solitamente simulato per analizzarne in comportamento (**simulazione comportamentale**)
- ➡ È necessario
 - Fornire degli stimoli (cioè gli input)
 - Avere un sistema capace di
 - Osservare l'evoluzione del modello durante la simulazione
 - Registrarne le variazioni delle uscite e dello stato interno per un'eventuale ispezione di funzionamento

Simulazione

➔ Simulazione di una specifica HDL con Xilinx Vivado

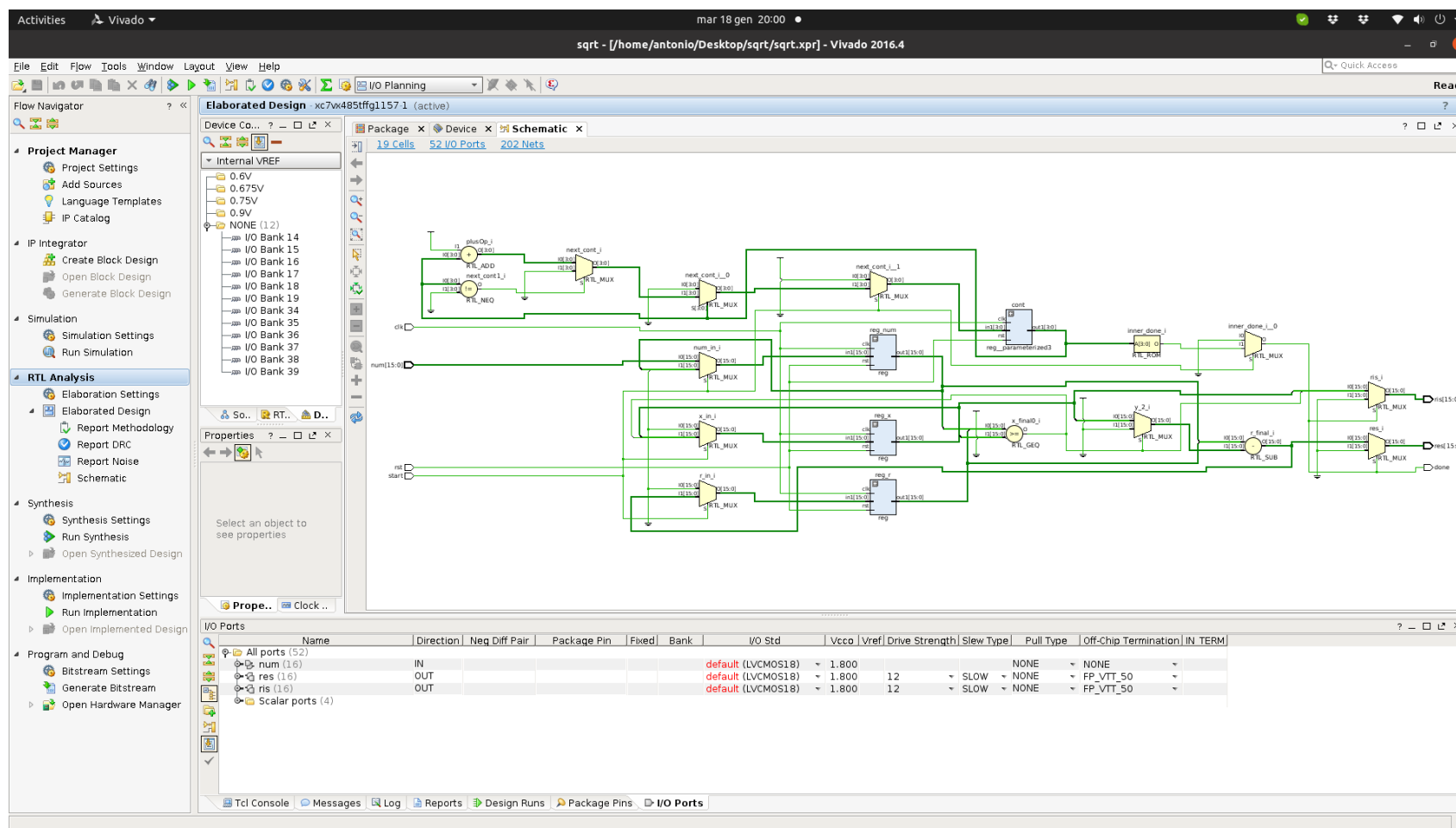


Sintesi circuitale

- ➡ La **sintesi circuitale** è il passaggio dalla descrizione comportamentale alla specifica della realizzazione finale sul dispositivo
- ➡ La sintesi avviene tramite appositi programmi che si appoggiano su librerie dove sono descritte le tecnologie realizzative da impiegare (fornite dal produttore)
- ➡ La sintesi è un processo delicato che deve essere opportunamente “guidato ed ottimizzato”
- ➡ **Non tutto ciò che è scritto in un HDL è sintetizzabile**
 - Solo un ristretto sottoinsieme del HDL si presta ad essere sintetizzato automaticamente
 - La restante parte è da impiegarsi per la descrizione e per la simulazione

Sintesi circuitale

➔ Sintesi di sistemi su dispositivo FPGA di una specifica HDL con Xilinx Vivado



VHDL

- ➡ Il **VHDL** è un linguaggio di descrizione dello hardware
- ➡ VHDL sta per VHSIC-HDL cioè
Very High Speed Integrated Circuit – Hardware Description Language
- ➡ Il VHDL è stato definito negli anni '80 dal dipartimento della difesa USA
- ➡ L'ultima versione pubblica risale al 1993 (IEEE STD 1076-1993)

Scrittura del codice sorgente

- ➡ Il codice sorgente di un modello VHDL è salvato in un file di semplice testo
- ➡ L'estensione deve essere *.vhd
- ➡ Il VHDL è case insensitive
- ➡ La sequenza -- indica l'inizio di un commento al codice che si estende per la restante parte della riga



POLITECNICO
MILANO 1863

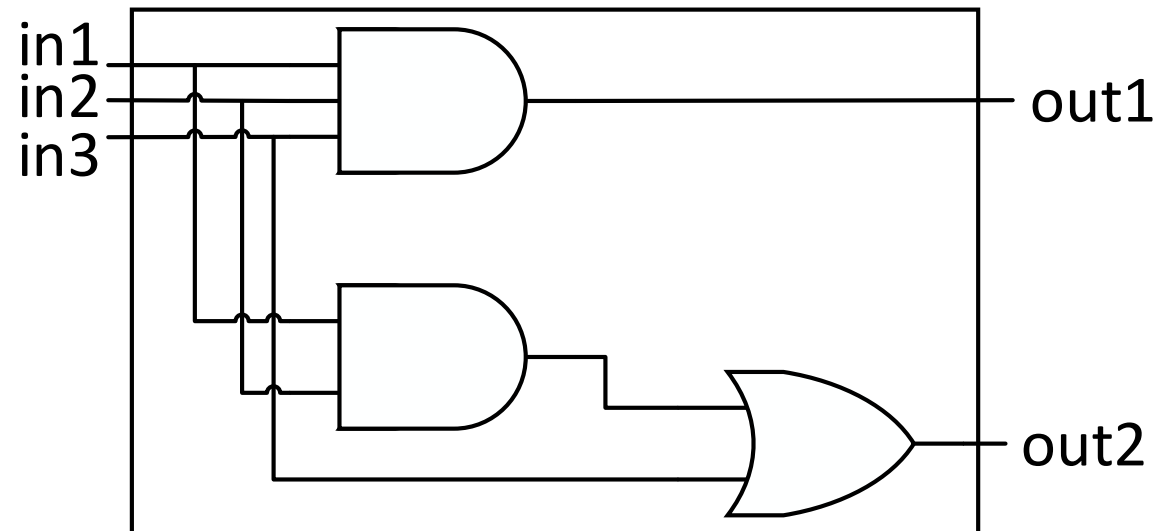
Reti Logiche

VHDL

Descrizione dataflow

Esempio di circuito 1

➡ Vogliamo specificare il seguente circuito combinatorio multi-uscita



Specifica dell'interfaccia del componente

- ➡ La **entity** è l'unità di progetto che descrive l'interfaccia di un componente in termini di input e output
- ➡ La **entity** non fornisce alcuna informazione sulla funzionalità svolta dal componente o sull'implementazione interna
- ➡ La **entity** può rappresentare
 - Una singola porta logica
 - Un componente
 - Un intero sistema complesso

Specifica dell'interfaccia del componente

➡ La entity del circuito di esempio 1:

```
entity esempio1 is
  port(
    in1, in2, in3: in std_logic;
    out1 : out std_logic;
    out2 : out std_logic
  );
end esempio1;
```

Specifica dell'interfaccia del componente

➡ La entity del circuito di esempio 1:

```
entity esempio1 is
  port(
    in1, in2, in3: in std_logic;
    out1 : out std_logic;
    out2 : out std_logic
  );
end esempio1;
```

Nome della entity

Elenco delle porte



Specifica dell'interfaccia del componente

- ➡ La entity del circuito di esempio 1:

```
entity esempio1 is
  port(
    in1, in2, in3: in std_logic;
    out1 : out std_logic;
    out2 : out std_logic
  );
end esempio1;
```

Per ogni porta bisogna specificare:

- Il nome
- La direzione
- Il tipo di dato

- ➡ La direzione può essere in o out
- ➡ La specifica dei nomi (di porte, entity, ...) segue regole simili alla dichiarazione delle variabili in un linguaggio di programmazione
- ➡ Il tipo di dato `std_logic` serve per rappresentare i valori e le operazioni su valori logici ad un bit

Specifica dell'interfaccia del componente

➡ La entity del circuito di esempio 1:

```
entity esempio1 is
  port(
    in1, in2, in3: in std_logic;
    out1 : out std_logic;
    out2 : out std_logic
  );
end esempio1;
```

È possibile specificare più porte con le stesse caratteristiche in una singola dichiarazione

Std_logic

- ➔ Il tipo di dato `Std_logic` è utilizzato per rappresentare valori logici a un bit
- ➔ `Std_logic` definisce 9 valori:
 - `'0'`
 - `'1'`
 - `'-'`: don't care, indifferenza
 - `'Z'`: alta impedenza

➔ Valori sintetizzabili

 - `'U'`: uninitialized
 - `'X'`: unknown
 - `'W'`: weak unknown
 - `'L'`: weak 0
 - `'H'`: weak 1

➔ Valori utilizzati in simulazione per avere un comportamento deterministico
- ➔ I valori `std_logic` vengono indicati tra singoli apici
- ➔ Noi considereremo soltanto i primi tre valori nell'elenco (`'0'`, `'1'`, `'-'`) e vedremo marginalmente alcuni degli altri (`'U'`)

Std_logic

- ➔ Per utilizzare il tipo `std_logic` va inclusa la seguente dichiarazione prima della specifica della `entity`:

```
LIBRARY ieee;  
USE ieee.std_logic_1164.ALL;
```

Tipi di dato:

- ➔ Il VHDL definisce diversi tipi di dato, non tutti sintetizzabili
- `Bit`, `real`, `integer`, `time`, `boolean`, `file`, `character`, ...
- ➔ Il VHDL è un linguaggio fortemente tipizzato
- Le operazioni possono essere eseguiti solo tra tipi compatibili
 - Non è possibile eseguire cast impliciti
 - Il VHDL fornisce funzioni specifiche per eseguire le conversioni

Specifica dell'architettura del componente

- ➡ La **architecture** è l'unità di progetto che descrive l'implementazione interna di un componente data la **entity**
- ➡ La **architecture** può essere descritta tramite tre approcci diversi:
 - **Dataflow**
 - **Strutturale**
 - **Comportamentale (o behavioral)**
- ➡ È possibile usare anche un mix dei tre approcci

Specifica dell'architettura del componente

➡ La architecture del circuito di esempio 1:

architecture dataflow of esempio1 is

begin

 out1 <= in1 and in2 and in3;

 out2 <= (in1 and in2) or in3;

end dataflow;

Specifica dell'architettura del componente

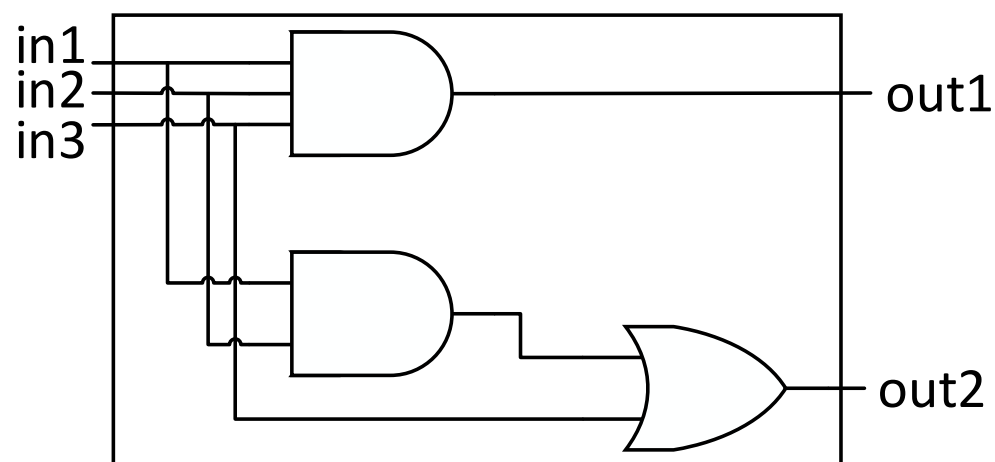
➔ La architecture del circuito di esempio 1:

```
architecture dataflow of esempio1 is
```

```
begin
```

```
    out1 <= in1 and in2 and in3;  
    out2 <= (in1 and in2) or in3;
```

```
end dataflow;
```



Nome della entity implementata

Nome della architecture

Implementazione tramite equazioni logiche (dataflow)

Specifica dataflow

➡ La architecture del circuito di esempio 1:

architecture dataflow of esempio1 is

begin

```
out1 <= in1 and in2 and in3;  
out2 <= (in1 and in2) or in3;
```

end dataflow;

➡ Istruzioni concorrenti

➡ Le equazioni logiche **NON** sono un blocco di codice sequenziale

➡ Le equazioni logiche

- Descrivono una serie di reti logiche
- Descrivono un comportamento concorrente
- Il loro ordinamento nel codice è quindi ininfluente

Specifica dataflow

➔ La architecture del circuito di esempio 1:

architecture dataflow of esempio1 is

begin

```
out1 <= in1 and in2 and in3;  
out2 <= (in1 and in2) or in3;
```

end dataflow;

→ Istruzioni concorrenti

Comportamento durante la simulazione:

- ➔ Ciascuna equazione è rivalutata ogni qualvolta c'è una variazione in una delle porte nell'espressione sulla destra dell'assegnamento
- ➔ L'aggiornamento del valore della porta alla sinistra dell'assegnamento è istantaneo

Specifica dataflow

➔ La architecture del circuito di esempio 1:

architecture dataflow of esempio1 is

begin

out1 <= in1 and in2 and in3;

out2 <= (in1 and in2) or in3;

end dataflow;

Assegnamento di un
valore ad una porta

Espressione logica

➔ Std_logic supporta gli operatori logici: and, or, not, xor, nand, nor, ...

➔ Attenzione:

- Non è possibile eseguire assegnamenti su porte di input
- Non è possibile utilizzare una porta di output all'interno di espressioni logiche alla destra di un assegnamento

Codice completo per l'esempio 1

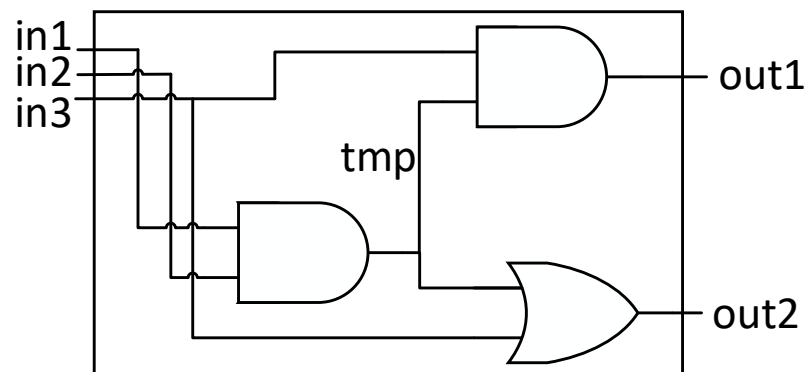
Commento	{	-- esempio1
Inclusioni delle librerie	{	LIBRARY ieee; USE ieee.std_logic_1164.ALL;
Entity	{	entity esempio1 is port(in1, in2, in3: in std_logic; out1 : out std_logic; out2 : out std_logic); end esempio1;
Architecture	{	architecture dataflow of esempio1 is begin out1 <= in1 and in2 and in3; out2 <= (in1 and in2) or in3; end dataflow;

Utilizzo di segnali

- ➔ Un'altra implementazione della architecture dell'esempio 1:

```
architecture dataflow of esempio1 is
    signal tmp : std_logic;
begin
    tmp <= in1 and in2;
    out1 <= tmp and in3;
    out2 <= tmp or in3;
end dataflow;
```

- ➔ Il segnale rappresenta un collegamento interno al componente



Utilizzo di segnali

- ➔ Un'altra implementazione della architecture dell'esempio 1:

architecture dataflow of esempio1 is

`signal tmp : std_logic;`

`begin`

`tmp <= in1 and in2;`

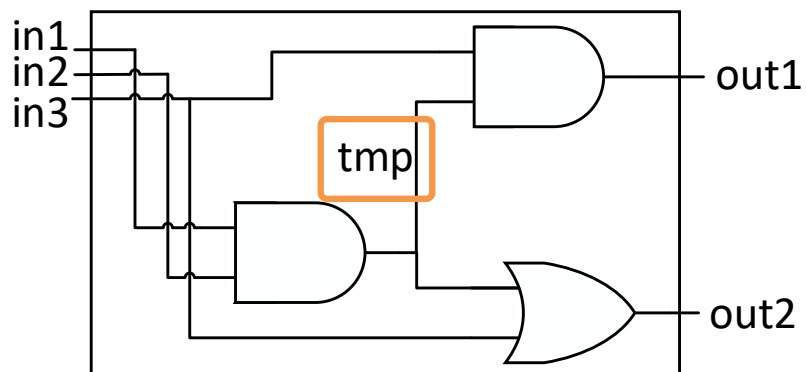
`out1 <= tmp and in3;`

`out2 <= tmp or in3;`

`end dataflow;`

➔ Dichiarazione di un segnale
(da specificare prima di `begin`)

- ➔ Il segnale rappresenta un collegamento interno al componente



Utilizzo di segnali

- ➔ Un'altra implementazione della *architecture* dell'esempio 1:

```
architecture dataflow of esempio1 is
    signal tmp : std_logic;
begin
    tmp <= in1 and in2;
    out1 <= tmp and in3;
    out2 <= tmp or in3;
end dataflow;
```

- ➔ Lo stesso segnale **NON** può comparire sia alla sinistra che alla destra dell'assegnamento nella stessa equazione logica
- ➔ Attenzione anche ad evitare eventuali cicli combinatori indiretti, cioè generati da più equazioni che formano un «anello»



POLITECNICO
MILANO 1863

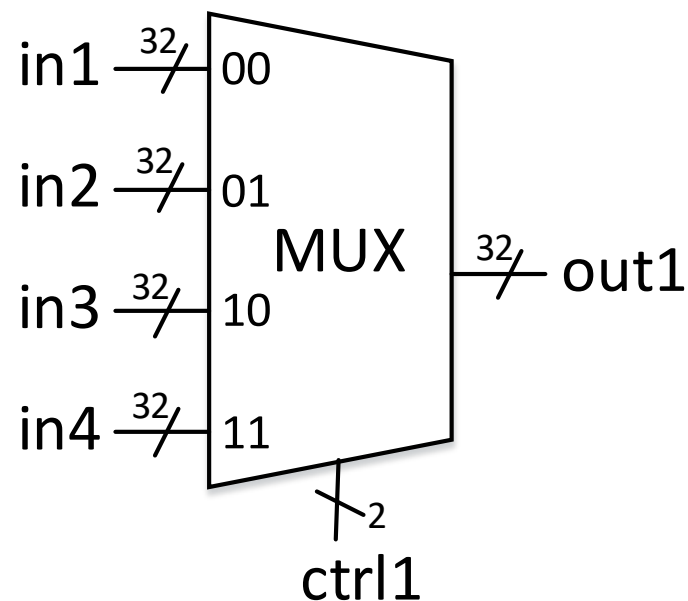
Reti Logiche

VHDL

Descrizione dataflow a livello register-transfer

Esempio di circuito 2

- ➔ Vogliamo specificare un multiplexer con 4 ingressi dati, dove ciascun ingresso dati (e l'uscita) è rappresentato da un bus a 32 bit



Std_logic_vector

➡ La entity del circuito di esempio 2:

```
entity esempio2 is
  port(
    in1, in2, in3, in4: in std_logic_vector(31 downto 0);
    ctrl1: in std_logic_vector(1 downto 0);
    out1 : out std_logic_vector(31 downto 0)
  );
end esempio2;
```

Std_logic_vector

- ➡ La entity del circuito di esempio 2:

```
entity esempio2 is
  port(
    in1, in2, in3, in4: in std_logic_vector(31 downto 0);
    ctrl1: in std_logic_vector(1 downto 0);
    out1 : out std_logic_vector(31 downto 0)
  );
end esempio2;
```

- ➡ Il tipo di dato `std_logic_vector` serve per rappresentare un vettore (un bus) di valori `std_logic`
- ➡ L'intervallo degli indici (estremi inclusi) viene specificato tramite le parole chiave `downto` (intervallo decrescente) o `to` (intervallo crescente)

Assegnamento condizionale

➡ La architecture del circuito di esempio 2:

```
architecture dataflow of esempio2 is
begin
    out1 <= in1 when ctrl1 = "00" else
              in2 when ctrl1 = "01" else
              in3 when ctrl1 = "10" else
              in4;
end dataflow;
```

Assegnamento condizionale

- ➡ La architecture del circuito di esempio 2:

architecture dataflow of esempio2 is
begin

```
    out1 <= in1 when ctrl1 = "00" else  
        in2 when ctrl1 = "01" else  
        in3 when ctrl1 = "10" else  
        in4;
```

end dataflow;

- ➡ Istruzione di assegnamento condizionale
- ➡ La condizione restituisce `true` o `false` (tipo `boolean`)
- ➡ Viene assegnato il valore corrispondente alla prima condizione vera

Assegnamento condizionale

- ➡ La architecture del circuito di esempio 2:

architecture dataflow of esempio2 is
begin

```
    out1 <= in1 when ctrl1 = "00" else  
        in2 when ctrl1 = "01" else  
        in3 when ctrl1 = "10" else  
        in4;
```

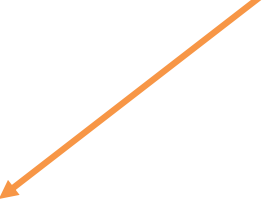
end dataflow;

- ➡ I tipi `std_logic` e `std_logic_vector` supportano le operazioni relazionali `=`, `/=`, `<`, `>`, `<=` e `>=`
- ➡ L'operazione viene eseguita bit per bit da sinistra verso destra (come se i vettori rappresentassero dei valori binari naturali)
- ➡ I vettori operandi devono avere la stessa dimensione
 - Tralasciamo il caso di operandi di dimensione diversa

Assegnamento condizionale

- ➡ La architecture del circuito di esempio 2:

```
architecture dataflow of esempio2 is
begin
    out1 <= in1 when ctrl1 = "00" else
        in2 when ctrl1 = "01" else
        in3 when ctrl1 = "10" else
        in4;
end dataflow;
```



- ➡ I valori `std_logic_vector` sono indicati tra doppi apici
- ➡ È possibile esprimere condizioni più complesse mediante gli operatori logic and, or, not , ...
- ➡ **Attenzione:** un valore `boolean` non può essere assegnato ad un segnale `std_logic` (VHDL è un linguaggio fortemente tipizzato!)

Assegnamento condizionale

➡ La architecture del circuito di esempio 2:

architecture dataflow of esempio2 is
begin

```
    out1 <= in1 when ctrl1 = "00" else  
           in2 when ctrl1 = "01" else  
           in3 when ctrl1 = "10" else  
           in4;
```

end dataflow;

➡ È necessario che le condizioni coprano tutti i casi possibili:

- Per un circuito combinatorio ad ogni combinazione di valori di ingressi è definita l'uscita
- Nel caso l'uscita sia don't care va esplicitato il corrispondente valore '-'
- Quindi è necessario specificare l'ultimo **else**

Assegnamento selettivo

➡ Altra architecture del circuito di esempio 2:

```
architecture dataflow of esempio2 is
begin
    with ctrl1 select
        out1 <= in1 when "00",
                in2 when "01",
                in3 when "10",
                in4 when others;
end dataflow;
```

Assegnamento selettivo

➡ Altra architecture del circuito di esempio 2:

architecture dataflow of esempio2 is
begin

```
with ctrl1 select  
    out1 <= in1 when "00",  
            in2 when "01",  
            in3 when "10",  
            in4 when others;
```

end dataflow;

➡ Istruzione di assegnamento selettivo

➡ **Attenzione:**

- I vari casi nelle clausole **when** devono essere mutuamente esclusivi
- È possibile elencare più valori in una clausola **when** separati dal simbolo **|**
- La clausola **Others** rappresenta tutti gli altri casi non elencati
 - Stessa regola dello **else** dell'assegnamento condizionale

Assegnamento selettivo

- ➔ Altra architecture del circuito di esempio 2:

architecture dataflow of esempio2 is
begin

```
  with ctrl1 select
    out1 <= in1 when "00",
             in2 when "01",
             in3 when "10",
             in4 when others;
```

end dataflow;

- ➔ Permette di rappresentare agevolmente le tabelle delle verità:

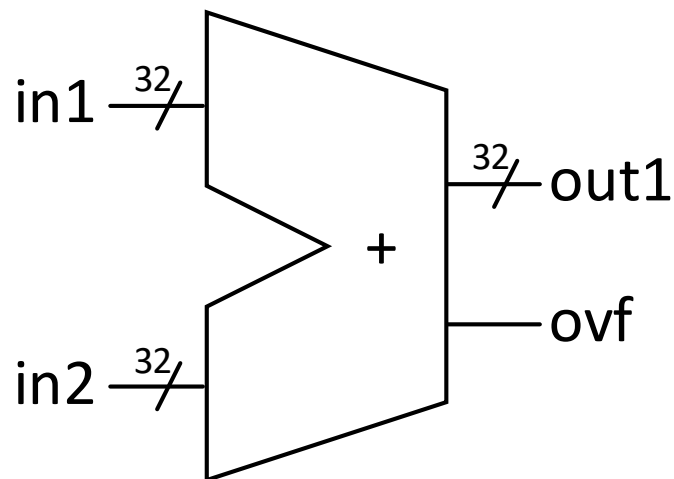
a(1 downto 0)		u
0	0	0
0	1	1
1	0	0
1	1	1



```
with a select
  u <= '1' when "01" | "11",
       '0' when others;
```

Esempio di circuito 3

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



Specifica dell'interfaccia dell'esempio

➡ La entity del circuito di esempio 3:

```
entity esempio3 is
  port(
    in1, in2: in std_logic_vector(31 downto 0);
    out1 : out std_logic_vector(31 downto 0);
    ovf : out std_logic
  );
end esempio3;
```

Numeric

- ➔ Il tipo `std_logic_vector` non supporta le operazioni aritmetiche
- ➔ Il VHDL ha altri due tipi di dato
 - `unsigned` per rappresentare numeri naturali in codifica binaria
 - `signed` per rappresentare numeri relativi in codifica binaria (in complemento a due)
- ➔ I tipi `signed` ed `unsigned` definiscono le operazioni aritmetiche e relazionali sui due insiemi rappresentati
- ➔ Per poter utilizzare i tipi `signed` ed `unsigned` è necessario includere la libreria `use ieee.numeric_std.all;`

Numeric

➡ La architecture del circuito di esempio 3:

```
architecture dataflow of esempio3 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;
```

Numeric

➡ La architecture del circuito di esempio 3:

architecture dataflow of esempio3 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;

➡ Dichiarazione di un segnale di tipo signed

- È necessario specificare l'intervallo degli indici come nel caso del tipo `std_logic_vector`

Numeric

- ➔ La architecture del circuito di esempio 3:

architecture dataflow of esempio3 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;
```

- ➔ I tipi `std_logic_vector` e `signed/unsigned` non sono tipi compatibili
- ➔ È necessario eseguire un cast esplicito in modo tale da eseguire le operazioni tra tipi compatibili

Numeric

➔ La architecture del circuito di esempio 3:

architecture dataflow of esempio3 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 da signed a
std_logic_vector
(con stessa dimensione)

Cast da std_logic_vector a
signed (con stessa dimensione)

Numeric

- ➔ La architecture del circuito di esempio 3:

architecture dataflow of esempio3 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);
```

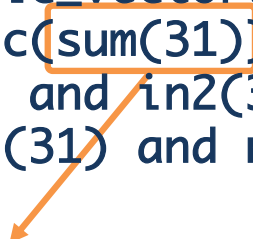
```
end dataflow;
```

- ➔ I tipi di dato numeric definiscono le operazioni aritmetiche e relazionali sui domini rappresentati
- Non tutte le operazioni (per esempio la divisione) sono sintetizzabili;
alle volte dipende dallo strumento di sintesi

Numeric

- ➡ La architecture del circuito di esempio 3:

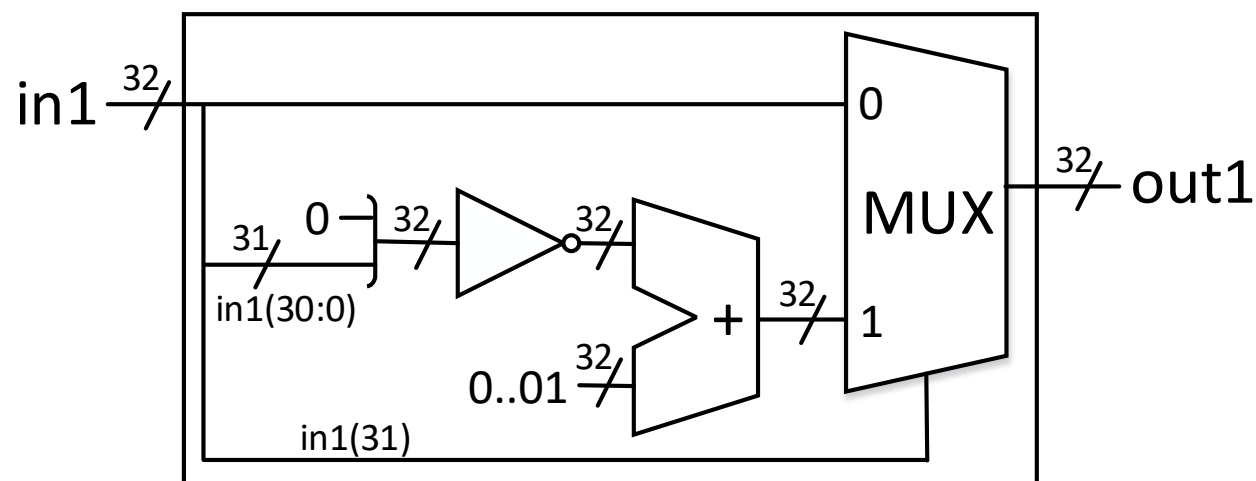
```
architecture dataflow of esempio3 is
    signal sum : SIGNED(31 downto 0);
    signal msb : std_logic;
begin
    sum <= SIGNED(in1) + SIGNED(in2);
    out1 <= std_logic_vector(sum);
    msb <= std_logic(sum(31));
    ovf <= (in1(31) and in2(31) and not msb) or
           (not in1(31) and not in2(31) and msb);
end dataflow;
```



- ➡ L'indirizzamento di un sottoinsieme dei valori in un segnale `std_logic_vector` o numeric si esegue con le parentesi tonde
- ➡ È possibile specificare un intervallo di valori mediante le parole chiave `to` e `downto`

Esempio di circuito 4

- ➔ Vogliamo specificare un circuito combinatorio che converte un valore a 32 bit che rappresenta un numero intero espresso in codifica modulo e segno nell'equivalente rappresentazione in complemento a 2



Specifica dell'interfaccia dell'esempio

➡ La entity del circuito di esempio 4:

```
entity esempio4 is
  port(
    in1: in std_logic_vector(31 downto 0);
    out1 : out std_logic_vector(31 downto 0)
  );
end esempio4;
```

Operazioni su segnali std_logic_vector

➡ La architecture del circuito di esempio 4:

```
architecture dataflow of esempio4 is
    signal cpl1, inv1 : std_logic_vector(31 downto 0);
begin
    cpl1 <= not ('0' & in1(30 downto 0));
    inv1 <= std_logic_vector(signed(cpl1) + 1);
    out1 <= in1 when in1(31) = '0' else
            inv1;
end dataflow;
```

Operazioni su segnali `std_logic_vector`

➡ La architecture del circuito di esempio 4:

```
architecture dataflow of esempio4 is
  signal cpl1, inv1 : std_logic_vector(31 downto 0);
begin
  cpl1 <= not ('0' & in1(30 downto 0));
  inv1 <= std_logic_vector(signed(cpl1) + 1);
  out1 <= in1 when in1(31) = '0' else
    inv1;
end dataflow;
```

Operazione logica su
segnale `std_logic_vector`
(eseguita bit per bit)

Concatenazione
di due segnali

È possibile esprimere
valori numeric in
formato decimale (cioè
in base 10)

Costanti

➡ Un'altra implementazione *architecture* del circuito di esempio 4:

```
architecture dataflow of esempio4 is
    signal cpl1, inv1 : std_logic_vector(31 downto 0);
    constant vdd32 :
        std_logic_vector(31 downto 0) := (others => '1');
begin
    cpl1 <= vdd32 xor ('0' & in1(30 downto 0));
    inv1 <= std_logic_vector(signed(cpl1) + 1);
    out1 <= in1 when in1(31) = '0' else
        inv1;
end dataflow;
```

Costanti

- ➡ Un'altra implementazione *architecture* del circuito di esempio 4:

```
architecture dataflow of esempio4 is
  signal cpl1, inv1 : std_logic_vector(31 downto 0);
  constant vdd32 :
    std_logic_vector(31 downto 0) := (others => '1');
begin
  cpl1 <= vdd32 xor ('0' & in1(30 downto 0));
  inv1 <= std_logic_vector(signed(cpl1) + 1);
  out1 <= in1 when in1(31) = '0' else
    inv1;
end dataflow;
```

- ➡ Definizione di un segnale costante

Costanti

- ➡ Un'altra implementazione *architecture* del circuito di esempio 4:

```
architecture dataflow of esempio4 is
  signal cpl1, inv1 : std_logic_vector(31 downto 0);
  constant vdd32 :
    std_logic_vector(31 downto 0) := (others => '1');
begin
  cpl1 <= vdd32 xor ('0' & in1(30 downto 0));
  inv1 <= std_logic_vector(signed(cpl1) + 1);
  out1 <= in1 when in1(31) = '0' else
    inv1;
end dataflow;
```

- ➡ Inizializzazione del segnale costante

Costanti

- ➡ Un'altra implementazione *architecture* del circuito di esempio 4:

```
architecture dataflow of esempio4 is
  signal cpl1, inv1 : std_logic_vector(31 downto 0);
  constant vdd32 :
    std_logic_vector(31 downto 0) := (others => '1');
begin
  cpl1 <= vdd32 xor ('0' & in1(30 downto 0));
  inv1 <= std_logic_vector(signed(cpl1) + 1);
  out1 <= in1 when in1(31) = '0' else
    inv1;
end dataflow;
```

- ➡ In questo modo viene assegnato a tutti i bit il valore '1'

Inizializzazione delle costanti e dei segnali

➡ Sintassi per l'inizializzazione di una costante (e più in generale di un segnale):

- Si specificano gli indici di ciascun sotto-assegnamento
- È possibile specificare intervalli con le parole chiave **to** e **downto** per assegnare lo stesso valore ad un bit a più indici
- È possibile utilizzare la parola chiave **others** per specificare tutti gli altri bit

➡ Esempio:

```
constant foo : std_logic_vector(31 downto 0) :=  
    (1 downto 0 => '1', 4 => '1', others => '0');
```



POLITECNICO
MILANO 1863

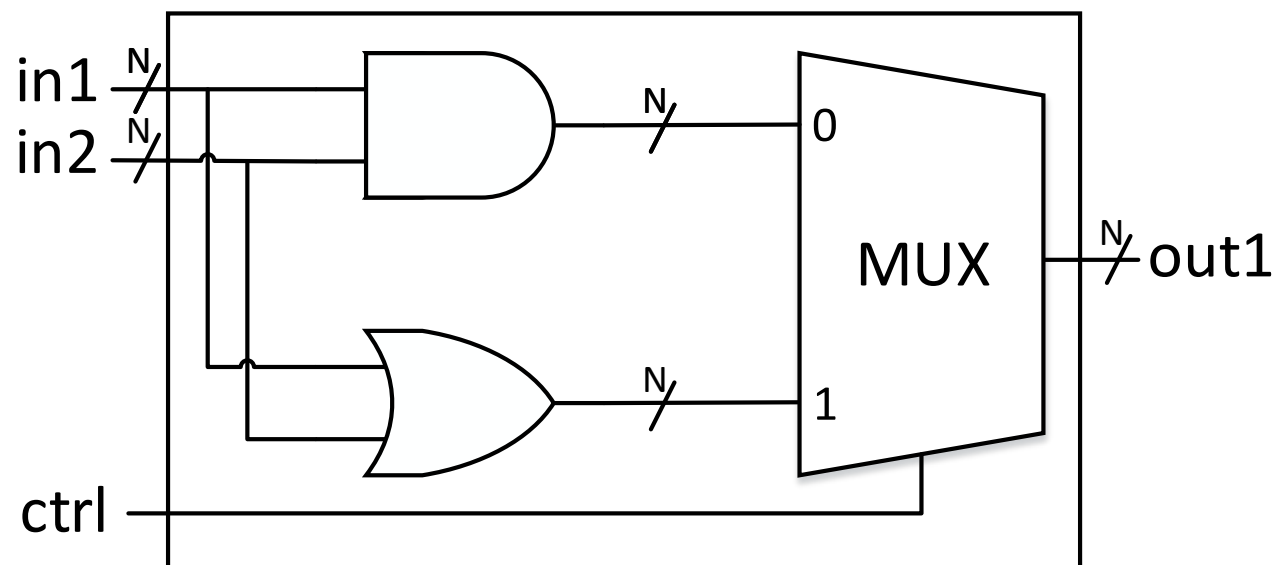
Reti Logiche

VHDL

Descrizione strutturale

Esempio di circuito 5

- ➔ Vogliamo specificare un modulo che esegue le operazioni logiche AND e OR in base ad un segnale di controllo ad un bit su due operandi a N bit. Si vuole lasciare parametrico il valore N



Generic

➡ La entity del circuito di esempio 5:

```
entity esempio5 is
  generic (
    N : integer := 5
  );
  port(
    in1, in2 : in std_logic_vector(N-1 downto 0);
    ctrl : in std_logic;
    out1 : out std_logic_vector(N-1 downto 0)
  );
end esempio5;
```

Generic

➡ La entity del circuito di esempio 5:

```
entity esempio5 is
```

```
  generic (
```

```
    N : integer := 5
```

```
  );
```

```
  port(
```

```
    in1, in2 : in std_logic_vector(N-1 downto 0);
```

```
    ctrl : in std_logic;
```

```
    out1 : out std_logic_vector(N-1 downto 0)
```

```
  );
```

```
end esempio5;
```

Specifica di un parametro
generic di tipo intero

Utilizzo del **generic** per rendere
parametrica l'interfaccia

Generic

➡ La `architecture` del circuito di esempio 5:

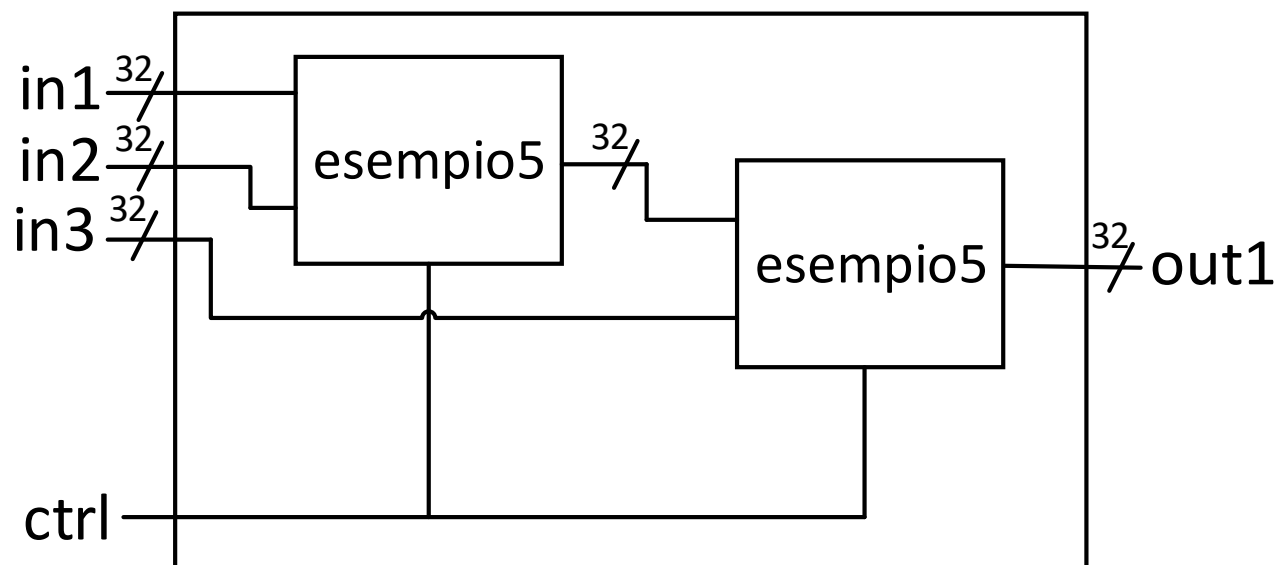
```
architecture dataflow of esempio5 is
begin
    out1 <= in1 and in2 when ctrl = '0' else
           in1 or in2;
end dataflow;
```

➡ La `architecture` non subisce variazioni in questo caso rispetto all'equivalente specifica non parametrica

- In altri casi potrebbe essere necessario l'utilizzo del parametro `N`, per esempio per dichiarare segnali interni della stessa dimensione delle porte

Esempio di circuito 6

- ➔ Vogliamo specificare un modulo che esegue le operazioni logiche AND e OR in base ad un segnale di controllo ad un bit su tre operandi a 32 bit. Si vuole riutilizzare il componente specificato nell'esempio 5



Descrizione strutturale

➡ La entity del circuito di esempio 6:

```
entity esempio6 is
  port(
    in1, in2, in3 : in std_logic_vector(31 downto 0);
    ctrl : in std_logic;
    out1 : out std_logic_vector(31 downto 0)
  );
end esempio6;
```

Descrizione strutturale

➡ La architecture del circuito di esempio 6:

```
architecture structural of esempio6 is
    signal tmp : std_logic_vector(31 downto 0);

    component esempio5 is
        generic (N : integer := 5);
        port(
            in1, in2 : in std_logic_vector(N-1 downto 0);
            ctrl : in std_logic;
            out1 : out std_logic_vector(N-1 downto 0)
        );
    end component;

begin
    --...
```

Descrizione strutturale

➡ La architecture del circuito di esempio 6:

```
architecture structural of esempio6 is  
    signal tmp : std_logic_vector(31 downto 0);
```

```
    component esempio5 is  
        generic (N : integer := 5);  
        port(  
            in1, in2 : in std_logic_vector(N-1 downto 0);  
            ctrl : in std_logic;  
            out1 : out std_logic_vector(N-1 downto 0)  
        );  
    end component;
```

```
begin  
--...
```

- I component che si vogliono utilizzare vanno elencati prima del **begin**
- Il component non è altro che la **entity** del modulo

Descrizione strutturale

➡ La architecture del circuito di esempio 6 (seconda parte):

```
architecture structural of esempio6 is
--...
begin
    es1 : esempio5
        generic map(32)
        port map(in1, in2, ctrl, tmp);

    es2 : esempio5
        generic map(N => 32)
        port map(in1 => tmp, in2 => in3,
                ctrl => ctrl, out1 => out1);
end structural;
```

Descrizione strutturale

- ➔ La architecture del circuito di esempio 6 (seconda parte):

```
architecture structural of esempio6 is
```

```
--...
```

```
begin
```

```
es1 : esempio5  
  generic map(32)  
  port map(in1, in2, ctrl, tmp);
```

```
es2 : esempio5  
  generic map(N => 32)  
  port map(in1 => tmp, in2 => in3,  
          ctrl => ctrl, out1 => out1);
```

```
end structural;
```

Istanziamenti
del componente

- ➔ Durante l'istanziamento del componente bisogna specificare i valori dei parametri **generic** ed i collegamenti alle sue porte

Descrizione strutturale

- ➔ La architecture del circuito di esempio 6 (seconda parte):

architecture structural of esempio6 is

--...

begin

es1 : esempio5

generic map(32)

port map(in1, in2, ctrl, tmp);

es2 : esempio5

generic map(N => 32)

port map(in1 => tmp, in2 => in3,
ctrl => ctrl, out1 => out1);

end structural;

Nome dell'istanza

Nome del componente

Descrizione strutturale

- ➡ La architecture del circuito di esempio 6 (seconda parte):

```
architecture structural of esempio6 is
```

```
--...
```

```
begin
```

```
  es1 : esempio5
```

```
    generic map(32)
```

```
      port map(in1, in2, ctrl, tmp);
```

```
  es2 : esempio5
```

```
    generic map(N => 32)
```

```
      port map(in1 => tmp, in2 => in3,  
              ctrl => ctrl, out1 => out1);
```

```
end structural;
```

Assegnamento
posizionale del valore
al parametro

Assegnamento nominale
del valore al parametro

- ➡ Il `generic map` non va specificato se il componente non ha nessun parametro `generic`
- ➡ Può essere omesso se si vuole usare il valore di default specificato

Descrizione strutturale

- ➡ La architecture del circuito di esempio 6 (seconda parte):

```
architecture structural of esempio6 is
```

```
--...
```

```
begin
```

```
    es1 : esempio5
```

```
        generic map(32)
```

```
        port map(in1, in2, ctrl, tmp);
```

Assegnamento
posizionale dei
segnali (o costanti)
alle porte

```
    es2 : esempio5
```

```
        generic map(N => 32)
```

```
        port map(in1 => tmp, in2 => in3,  
                  ctrl => ctrl, out1 => out1);
```

Assegnamento
nominale dei segnali (o
costanti) alle porte

```
end structural;
```

- ➡ La parola chiave `open` permette di lasciare non connessa una porta

Descrizione strutturale

➔ La architecture del circuito di esempio 6 (seconda parte):

```
architecture structural of esempio6 is
--...
begin
  es1 : esempio5
    generic map(32)
    port map(in1, in2, ctrl, tmp);
  es2 : esempio5
    generic map(N => 32)
    port map(in1 => tmp, in2 => in3,
             ctrl => ctrl, out1 => out1);
end structural;
```

Istruzioni concorrenti

➔ Ogni istanza funziona come un modulo concorrente a tutti gli altri (come succedeva con le equazioni logiche in una specifica dataflow)

Descrizione strutturale

- ➔ Un'implementazione alternativa del circuito di esempio 6:

```
architecture structural of esempio6 is
  --...
begin
  es1 : esempio5
    generic map(32)
    port map(in1, in2, ctrl, tmp);
  out1 <= in3 and tmp when ctrl = '0' else
    in3 or tmp;
end structural;
```

Istruzioni concorrenti

Strutturale

Dataflow

- ➔ È possibile specificare una descrizione mista dataflow, strutturale (e comportamentale)



POLITECNICO
MILANO 1863

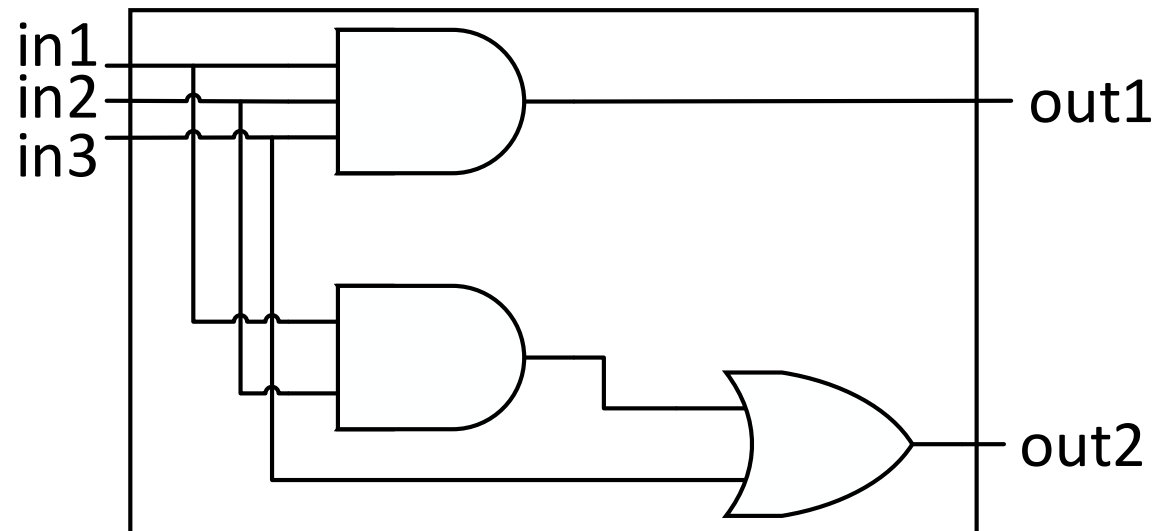
Reti Logiche

VHDL

Descrizione comportamentale

Esempio di circuito 7

- ➔ Riprendiamo in considerazione il circuito di esempio 1: vogliamo scrivere una specifica comportamentale del circuito combinatorio multi-uscita in esame



Specifica dell'interfaccia del componente

➡ La entity del circuito di esempio 7:

```
entity esempio7 is
  port(
    in1, in2, in3: in std_logic;
    out1 : out std_logic;
    out2 : out std_logic
  );
end esempio7;
```

Descrizione comportamentale

- ➔ Un'implementazione del circuito di esempio 7:
architecture behavioral of esempio7 is

```
begin
  process(in1, in2, in3)
    variable tmp1, tmp2 : std_logic;
  begin
    tmp1 := in3 and in2;
    out1 <= tmp1 and in1;
    tmp2 := in1 and in2;
    out2 <= tmp2 or in3;
  end process;
end behavioral;
```

Descrizione comportamentale

- ➡ Un'implementazione del circuito di esempio 7:
architecture behavioral of esempio7 is

begin

```
process(in1, in2, in3)
  variable tmp1, tmp2 : std_logic;
begin
  tmp1 := in3 and in2;
  out1 <= tmp1 and in1;
  tmp2 := in1 and in2;
  out2 <= tmp2 or in3;
end process;
```

end behavioral;

- ➡ Il costrutto process permette di descrivere il circuito in forma algoritmica

Descrizione comportamentale

- ➔ Un'implementazione del circuito di esempio 7:
architecture behavioral of esempio7 is

begin

```
process(in1, in2, in3)
  variable tmp1, tmp2 : std_logic;
begin
  tmp1 := in3 and in2;
  out1 <= tmp1 and in1;
  tmp2 := in1 and in2;
  out2 <= tmp2 or in3;
end process;
```

end behavioral;

- ➔ La descrizione comportamentale è ad un livello di astrazione maggiore rispetto a quelle dataflow e strutturale
- ➔ Non c'è una corrispondenza diretta tra la descrizione algoritmica e la struttura interna della futura realizzazione del circuito

Processo

- ➔ Un'implementazione del circuito di esempio 7:
architecture behavioral of esempio7 is

```
begin
  process(in1, in2, in3)
    variable tmp1, tmp2 : std_logic;
  begin
    tmp1 := in3 and in2;
    out1 <= tmp1 and in1;
    tmp2 := in1 and in2;
    out2 <= tmp2 or in3;
  end process;
end behavioral;
```

Lista di sensibilità

Dichiarazione delle variabili

Descrizione algoritmica della funzionalità implementata

Facoltativamente si può assegnare un nome al processo:
prova: process(in1, in2, in3)

Esecuzione di un processo

1. Il processo viene risvegliato quando si verifica una variazione in uno dei segnali specificati nella lista di sensibilità
2. Il corpo del processo viene eseguito in modo strettamente sequenziale ed atomicamente
3. Tutti i segnali che hanno subito un assegnamento vengono aggiornati **SOLO** al termine dell'esecuzione del corpo del processo

➡ ATTENZIONE:

- Assegnamenti a segnali durante l'esecuzione di un processo non sono immediati
- Se nella stessa esecuzione del processo si legge un segnale dopo che ha subito un assegnamento, verrà letto il valore iniziale
- Se sono stati eseguiti più assegnamenti allo stesso segnale, il segnale viene aggiornato solo con l'ultimo valore assegnato

Esecuzione di un processo

- ➔ Un'implementazione del circuito di esempio 7:
architecture behavioral of esempio7 is

```
begin
  process(in1, in2, in3)
    variable tmp1, tmp2 : std_logic;
  begin
    tmp1 := in3 and in2;
    out1 <= tmp1 and in1;
    tmp2 := in1 and in2;
    out2 <= tmp2 or in3;
  end process;
end behavioral;
```

1. Variazione in uno
dei segnali specificati

2. Esecuzione del corpo del
processo (le istruzioni sono
eseguire in sequenza)

3. Aggiornamento dei segnali (out1 e
out2 sono aggiornati effettivamente
solo dopo l'esecuzione)

Esecuzione di un processo

- ➔ Un'implementazione del circuito di esempio 7:
architecture behavioral of esempio7 is

```
begin
  process(in1, in2, in3)
    variable tmp1, tmp2 : std_logic;
  begin
    tmp1 := in3 and in2;
    out1 <= tmp1 and in1;
    tmp2 := in1 and in2;
    out2 <= tmp2 or in3;
  end process;
end behavioral;
```

Variabili – lette e scritte

Segnale (o porta) di ingresso – solo letto

Segnale (o porta) di uscita – solo scritto

- ➔ Il processi comunicano con l'esterno tramite i segnali
- ➔ Lavorano all'interno mediante variabili

Variabili

- ➔ Un'implementazione del circuito di esempio 7:
architecture behavioral of esempio7 is

```
begin
  process(in1, in2, in3)
    variable tmp1, tmp2 : std_logic;
  begin
    tmp1 := in3 and in2;
    out1 <= tmp1 and in1;
    tmp2 := in1 and in2;
    out2 <= tmp2 or in3;
  end process;
end behavioral;
```

- Dichiarazione di una variabile
- La visibilità è limitata al processo

- Assegnamento a variabili
- L'assegnamento è immediato

- ➔ Le variabili rappresentano un concetto più astratto rispetto ai segnali
- Non rappresentano necessariamente un collegamento
 - Non sono un elemento di memoria!

Variabili

- ➔ Un'implementazione del circuito di esempio 7:
architecture behavioral of esempio7 is

```
begin
  process(in1, in2, in3)
    variable tmp1 : std_logic;
  begin
    tmp1 := in3 and in2;
    out1 <= tmp1 and in1;
    tmp1 := in1 and in2;
    out2 <= tmp1 or in3;
  end process;
end behavioral;
```

tmp2 non è necessaria;
si può riutilizzare tmp1!

- ➔ Le variabili rappresentano un concetto più astratto rispetto ai segnali
- Non rappresentano necessariamente un collegamento
 - Non sono un elemento di memoria!

Segnali e variabili

	Segnali	Variabili
Dichiarazione	Parte dichiarativa di una architecture	Parte dichiarativa di un processo
Assegnamento	$\leftarrow$	$:=$
Inizializzazione	$:=$	
Natura dell'assegnamento	Concorrente	Sequenziale
Utilizzo	In architecture e processi	Solo in processi
Effetto dell'assegnamento	Non immediato nei processi	Immediato

Descrizione mista

- ➔ Un'implementazione del circuito di esempio 7:

```
architecture behavioral of esempio7 is  
  signal tmp2 : std_logic;  
begin
```

```
  process(in1, in2, in3)  
    variable tmp1 : std_logic;  
  begin  
    tmp1 := in1 and in2;  
    out1 <= tmp1 and in3;  
    tmp2 <= tmp1;  
  end process;
```

➔ Comportamentale

```
  out2 <= tmp2 or in3;  
end behavioral;
```

➔ Dataflow

- ➔ È possibile specificare una descrizione mista dataflow, strutturale e comportamentale

Descrizione mista

➡ Un'implementazione del circuito di esempio 7:

```
architecture behavioral of esempio7 is
  signal tmp2 : std_logic;
begin
  process(in1, in2, in3)
    variable tmp1 : std_logic;
  begin
    tmp1 := in1 and in2;
    out1 <= tmp1 and in3;
    tmp2 <= tmp1;
  end process;
  out2 <= tmp2 or in3;
end behavioral;
```

Istruzioni concorrenti

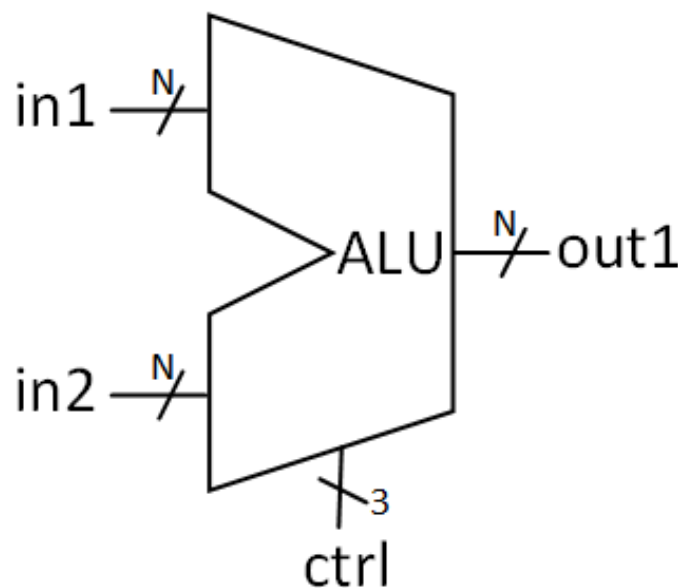
Istruzioni sequenziali

➡ Le istruzioni sono eseguite sequenzialmente all'interno di un processo

➡ Ogni processo è visto dall'esterno come una sola istruzione concorrente

Esempio di circuito 8

- ➔ Vogliamo specificare un'unità aritmetico-logica in grado di eseguire le operazioni AND, OR, +, −, = su due valori in ingresso ad N bit. Per le operazioni aritmetiche assumere che i valori in ingresso siano codificati in complemento a due. Un ingresso di controllo a 3 bit permette di selezionare l'operazione (AND: 000, OR: 001, +: 010, −: 011, =: 100); nel caso venga specificato il codice di un'operazione non supportata il modulo presenta in uscita il valore 0



Specifica dell'interfaccia del componente

➡ La entity del circuito di esempio 8:

```
entity esempio8 is
  generic (
    N : integer := 5
  );
  port(
    in1, in2: in std_logic_vector(N-1 downto 0);
    ctrl: in std_logic_vector(2 downto 0);
    out1: out std_logic_vector(N-1 downto 0)
  );
end esempio8;
```

Costrutto if

➡ La architecture del circuito di esempio 8:

```
architecture behavioral of esempio8 is
begin
  process(in1, in2, ctrl)
    constant GROUND : std_logic_vector(N-1 downto 0)
                        := (others => '0');

    begin
      if ctrl = "000" then
        out1 <= in1 and in2;
      elsif ctrl = "001" then
        out1 <= in1 or in2;
      elsif ctrl = "010" then
        out1 <= std_logic_vector(SIGNED(in1) + SIGNED(in2));
      --...
```

Costrutto if

➡ La architecture del circuito di esempio 8:

architecture behavioral of esempio8 is

begin

process(in1, in2, ctrl)

constant GROUND : std_logic_vector(N-1 downto 0)
:= (others => '0');

begin

if ctrl = "000" then

out1 <= in1 and in2;

elsif ctrl = "001" then

out1 <= in1 or in2;

elsif ctrl = "010" then

out1 <= std_logic_vector(SIGNED(in1) + SIGNED(in2));

--...

➡ Il costrutto if funziona come in qualsiasi linguaggio di programmazione

Costrutto if

➔ La architecture del circuito di esempio 8 (seconda parte):

```
--...
    elsif ctrl = "011" then
        out1 <= std_logic_vector(SIGNED(in1) - SIGNED(in2));
    elsif ctrl = "100" then
        if in1 = in2 then
            out1 <= (0 => '1', others => '0');
        else
            out1 <= GROUND;
        end if;
    else
        out1 <= GROUND;
    end if;
end process;
end behavioral;
```

Costrutto if

➔ La architecture del circuito di esempio 8 (seconda parte):

```
--...
    elsif ctrl = "011" then
        out1 <= std_logic_vector(SIGNED(in1) - SIGNED(in2));
    elsif ctrl = "100" then
        if in1 = in2 then
            out1 <= (0 => '1', others => '0');
        else
            out1 <= GROUND;
        end if;
    else
        out1 <= GROUND;
    end if;
end process;
end behavioral;
```

Possiamo annidare istruzioni if

Costrutto if

➡ La architecture del circuito di esempio 8 (seconda parte):

```
--...
    elsif ctrl = "011" then
        out1 <= std_logic_vector(SIGNED(in1) - SIGNED(in2));
    elsif ctrl = "100" then
        if in1 = in2 then
            out1 <= (0 => '1', others => '0');
        else
            out1 <= GROUND;
        end if;
    else
        out1 <= GROUND;
    end if;
end process;
end behavioral;
```

- Come nel caso degli assegnamenti condizionali del dataflow, è necessario specificare l'ultimo **else**
- Inoltre, se un segnale è assegnato in un ramo di un **if**, **DEVE** esserci un assegnamento sullo stesso segnale in **TUTTI** gli altri rami dell'**if**

Costrutto case

➔ Un'altra architecture del circuito di esempio 8:

architecture behavioral of esempio8 is

begin

 process(in1, in2, ctrl)

 constant GROUND : std_logic_vector(N-1 downto 0)
 := (others => '0');

 begin

 case ctrl is

 when "000" =>

 out1 <= in1 and in2;

 when "001" =>

 out1 <= in1 or in2;

 when "010" =>

 out1 <= std_logic_vector(SIGNED(in1) + SIGNED(in2));

--...

Costrutto case

➔ Un'altra architecture del circuito di esempio 8:

architecture behavioral of esempio8 is

begin

 process(in1, in2, ctrl)

 constant GROUND : std_logic_vector(N-1 downto 0)
 := (others => '0');

 begin

 case ctrl is

 when "000" =>

 out1 <= in1 and in2;

 when "001" =>

 out1 <= in1 or in2;

 when "010" =>

 out1 <= std_logic_vector(SIGNED(in1) + SIGNED(in2));

--...

Costrutto case

Costrutto case

➡ Un'altra architecture del circuito di esempio 8 (seconda parte):

```
--...
    when "011" =>
        out1 <= std_logic_vector(SIGNED(in1) - SIGNED(in2));
    when "100" =>
        if in1 = in2 then
            out1 <= (0 => '1', others => '0');
        else
            out1 <= GROUND;
        end if;
    when others =>
        out1 <= GROUND;
    end case;
end process;
end behavioral;
```

Costrutto case

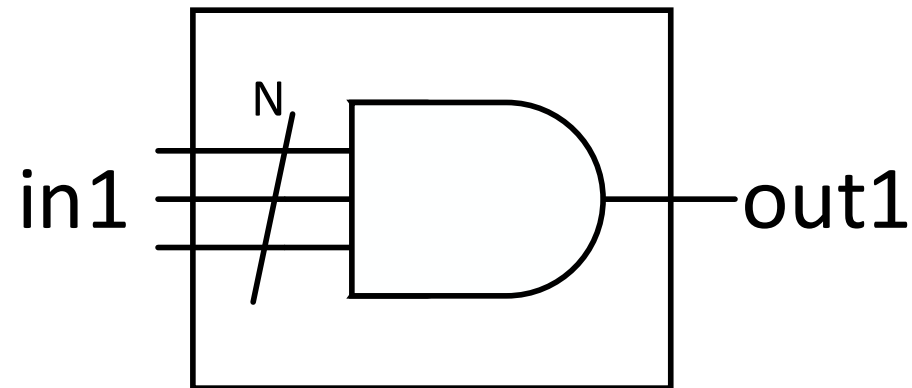
➡ Un'altra architecture del circuito di esempio 8 (seconda parte):

```
--...
    when "011" =>
        out1 <= std_logic_vector(SIGNED(in1) - SIGNED(in2));
    when "100" =>
        if in1 = in2 then
            out1 <= (0 => '1', others => '0');
        else
            out1 <= GROUND;
        end if;
    when others =>
        out1 <= GROUND;
    end case;
end process;
end behavioral;
```

- Come nel caso degli assegnamenti selettivi del dataflow, è necessario specificare il caso **others**
- Inoltre, se un segnale è assegnato in un ramo di un **case**, **DEVE** esserci un assegnamento sullo stesso segnale in **TUTTI** gli altri rami del **case**

Esempio di circuito 9

- ➔ Vogliamo realizzare un modulo che riceve in ingresso un segnale a N bit e produce in uscita l'and di tutti i singoli bit ricevuti



Specifica dell'interfaccia del componente

➡ La entity del circuito di esempio 9:

```
entity esempio9 is
  generic (
    N : integer := 5
  );
  port(
    in1: in std_logic_vector(N-1 downto 0);
    out1: out std_logic
  );
end esempio9;
```

Costrutto for

➡ La architecture del circuito di esempio 9:

```
architecture behavioral of esempio9 is
begin
  process(in1)
    variable tmp : std_logic;
  begin
    tmp := in1(0);
    for i in 1 to N-1 loop
      tmp := tmp and in1(i);
    end loop;
    out1 <= tmp;
  end process;
end behavioral;
```

Costrutto for

➡ La architecture del circuito di esempio 9:

architecture behavioral of esempio9 is
begin

 process(in1)
 variable tmp : std_logic;
 begin

 tmp := in1(0);
 for i in 1 to N-1 loop
 tmp := tmp and in1(i);

 end loop;
 out1 <= tmp;
 end process;
end behavioral;

➡ Affinché il circuito sia realizzabile, il ciclo for deve iterare su un intervallo noto durante la specifica del circuito



POLITECNICO
MILANO 1863

Reti Logiche

VHDL

Specifica di circuiti sequenziali

Esempio 10

- ➔ Cosa succede se non specifichiamo il ramo `else` di un'istruzione `if`? O più in generale se un segnale è assegnato solo in alcuni rami di un `if` ma non in tutti?

```
entity esempio10 is
  port(
    in1 : in std_logic;
    enable : in std_logic;
    out1 : out std_logic
  );
end esempio10;
```

```
architecture behavioral
  of esempio10 is
begin
  process(in1, enable)
  begin
    if enable = '1' then
      out1 <= in1;
    end if;
  end process;
end behavioral;
```

Inferenza di un latch

- ➡ Lo valore di out1
 - È aggiornato soltanto quando enable è uguale a 1
 - Negli altri casi rimane invariato
- ➡ Stiamo specificando un latch con segnale di enable!

architecture behavioral of esempio10 is

begin

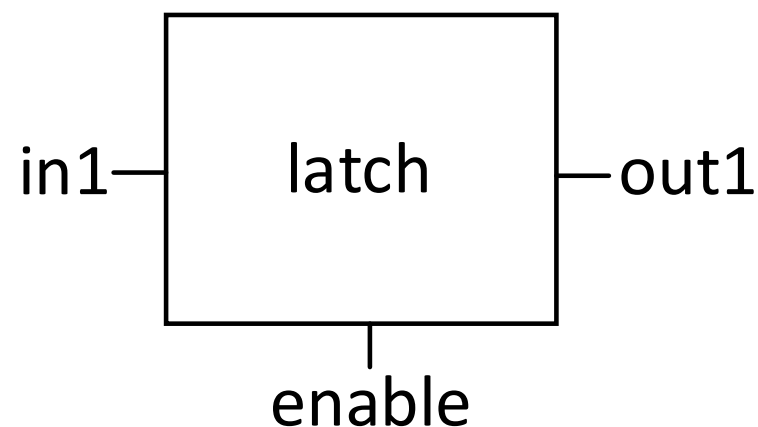
 process(in1, enable)

 begin

 if enable = '1' then
 out1 <= in1;
 end if;

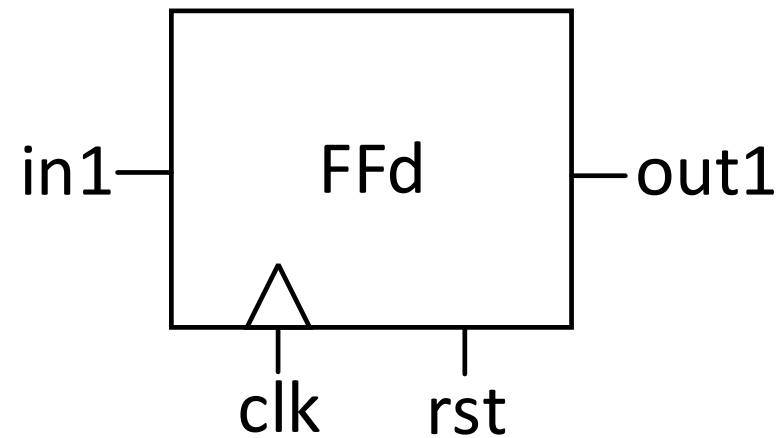
 end process;

end behavioral;



Esempio di circuito 11

➡ Vogliamo realizzare un flip-flop di tipo D con reset asincrono



Flip-flop D

➡ Specifica di un flip-flop di tipo D con reset asincrono:

```
entity esempio11 is
  port(
    in1 : in std_logic;
    clk, rst : in std_logic;
    out1 : out std_logic
  );
end esempio11;
```

```
architecture behavioral
  of esempio11 is
begin
  process(clk, rst)
  begin
    if rst = '1' then
      out1 <= '0';
    elsif clk'event and
          clk = '1' then
      out1 <= in1;
    end if;
  end process;
end behavioral;
```

Flip-flop D

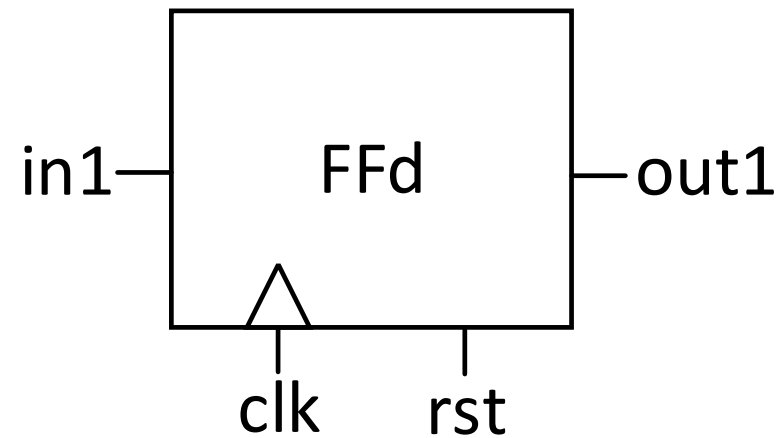
➔ Specifica di un flip-flop di tipo D con reset asincrono:

```
architecture behavioral
    of esempio11 is
begin
    process(clk, rst)
    begin
        if rst = '1' then
            out1 <= '0';
        elsif clk'event and
            clk = '1' then
            out1 <= in1;
        end if;
    end process;
end behavioral;
```

- Non è necessario specificare in1 (lo stato si aggiorna soltanto quando cambiano clk e rst)
- Specifica di un reset asincrono
- Il termine event indica se c'è stato un cambiamento nel segnale
- L'intera condizione indica se si è verificato un fronte di salita
- La condizione può essere sostituita con l'equivalente rising_edge(clk)

Esempio di circuito 12

➡ Vogliamo realizzare un flip-flop di tipo D con reset sincrono



Flip-flop D

➔ Specifica di un flip-flop di tipo D con reset sincrono:

```
entity esempio12 is
  port(
    in1 : in std_logic;
    clk, rst : in std_logic;
    out1 : out std_logic
  );
end esempio12;
```

```
architecture behavioral
  of esempio12 is
begin
  process(clk, rst)
  begin
    if rising_edge(clk) then
      if rst = '1' then
        out1 <= '0';
      else
        out1 <= in1;
      end if;
    end if;
  end process;
end behavioral;
```

Flip-flop D

➔ Specifica di un flip-flop di tipo D con reset sincrono:

```
entity esempio12 is
  port(
    in1 : in std_logic;
    clk, rst : in std_logic;
    out1 : out std_logic
  );
end esempio12;
```

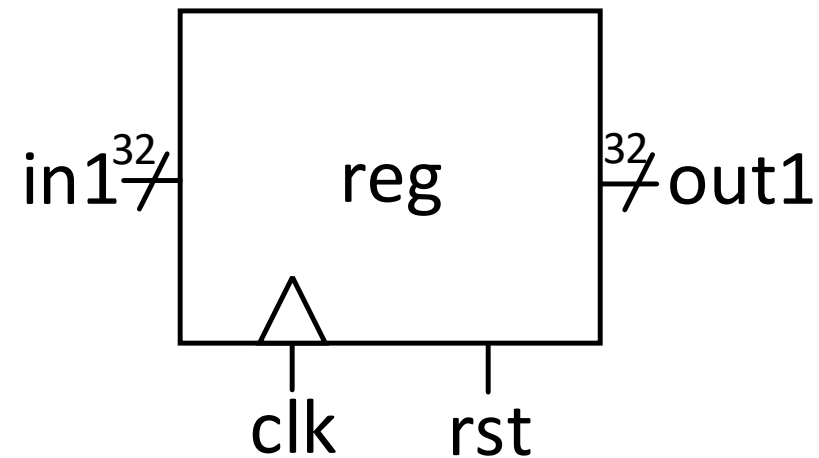
```
architecture behavioral
  of esempio12 is
  begin
    process(clk, rst)
    begin
      if rising_edge(clk) then
        if rst = '1' then
          out1 <= '0';
        else
          out1 <= in1;
        end if;
      end if;
    end process;
  end behavioral;
```

Specifica del reset
sincrono



Esempio di circuito 13

- ➔ Vogliamo realizzare un un registro parallelo-parallelo ad 32 bit con reset asincrono



Registro

➔ Specifica del registro:

```
entity esempio13 is
  port(in1 : in std_logic_vector(31 downto 0);
        clk, rst : in std_logic;
        out1 : out std_logic_vector(31 downto 0)
  );
end esempio13;
```

```
architecture behavioral of esempio13 is
begin
  process(clk, rst)
  begin
    if rst = '1' then
      out1 <= (others => '0');
    elsif clk = '1' and clk'event then
      out1 <= in1;
    end if;
  end process;
end behavioral;
```

Registro

➔ Specifica del registro:

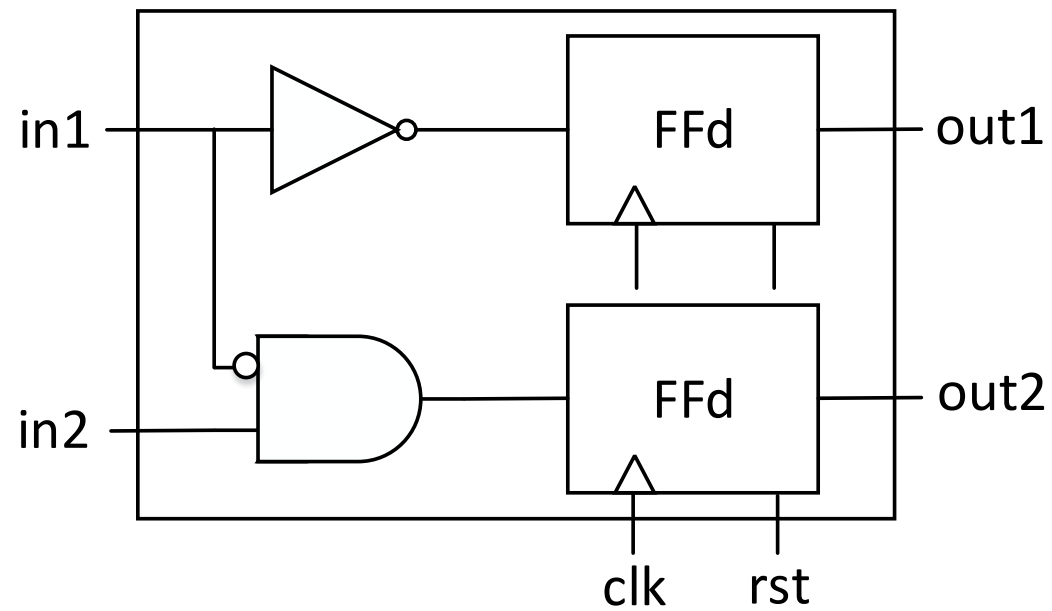
```
entity esempio13 is
  port(in1 : in std_logic_vector(31 downto 0);
        clk, rst : in std_logic;
        out1 : out std_logic_vector(31 downto 0)
  );
end esempio13;
```

```
architecture behavioral of esempio13 is
begin
  process(clk, rst)
  begin
    if rst = '1' then
      out1 <= (others => '0');
    elsif clk = '1' and clk'event then
      out1 <= in1;
    end if;
  end process;
end behavioral;
```

L'unica differenza rispetto al flip-flop è la dimensione delle porte e dei segnali

Esempio di circuito 14

➡ Vogliamo realizzare il seguente circuito sequenziale



Specifica di un circuito sequenziale

➔ Realizzazione del circuito di esempio 14:

```
entity esempio14 is
  port(
    in1, in2 : in std_logic;
    clk, rst : in std_logic;
    out1, out2 : out std_logic
  );
end esempio14;
```

```
architecture behavioral of
  esempio14 is
begin
  process(clk, rst)
  begin
    if rst = '1' then
      out1 <= '0';
      out2 <= '0';
    elsif rising_edge(clk) then
      out1 <= not in1;
      out2 <= not in1 and in2;
    end if;
  end process;
end behavioral;
```

Specifica di un circuito sequenziale

➡ Realizzazione del circuito di esempio 14:

```
entity esempio14 is
  port(
    in1, in2 : in std_logic;
    clk, rst : in std_logic;
    out1, out2 : out std_logic
  );
end esempio14;
```

Espressioni combinatorie
specificate all'interno della
descrizione dei flip-flop

```
architecture behavioral of
  esempio14 is
begin
  process(clk, rst)
  begin
    if rst = '1' then
      out1 <= '0';
      out2 <= '0';
    elsif rising_edge(clk) then
      out1 <= not in1;
      out2 <= not in1 and in2;
    end if;
  end process;
end behavioral;
```

Specifica di un circuito sequenziale

➡ Realizzazione alternativa del circuito di esempio 14:

```
architecture behavioral of esempio14 is
    signal tmp1, tmp2 : std_logic;
begin
    process(clk, rst)
    begin
        if rst = '1' then
            out1 <= '0';
            out2 <= '0';
        elsif rising_edge(clk) then
            out1 <= tmp1;
            out2 <= tmp2;
        end if;
    end process;

    tmp1 <= not in1;
    tmp2 <= not in1 and in2;
end behavioral;
```

Specifica di un circuito sequenziale

➔ Realizzazione alternativa del circuito di esempio 14:

```
architecture behavioral of esempio14 is
```

```
    signal tmp1, tmp2 : std_logic;
```

```
begin
```

```
    process(clk, rst)
```

```
    begin
```

```
        if rst = '1' then
```

```
            out1 <= '0';
```

```
            out2 <= '0';
```

```
        elsif rising_edge(clk) then
```

```
            out1 <= tmp1;
```

```
            out2 <= tmp2;
```

```
        end if;
```

```
    end process;
```

```
    tmp1 <= not in1;
```

```
    tmp2 <= not in1 and in2;
```

```
end behavioral;
```

- Notare bene che le due istruzioni dataflow ed il processo sono istruzioni concorrenti
- L'ordine delle istruzioni è influente

Esempio di circuito 15

- ➡ Vogliamo specificare in VHDL una macchina a stati finiti (finite state machine - FSM) con un ingresso ed un'uscita ad un bit in grado di riconoscere la sequenza in ingresso 001. La macchina presenta un 1 in uscita per un ciclo di clock quando riconosce la sequenza, altrimenti 0
- ➡ Tabella degli stati della macchina:

	0	1	U
S0	S1	S0	0
S1	S2	S0	0
S2	S2	S3	0
s3	S1	S0	1

Specifica dell'interfaccia della FSM

➡ La entity del circuito dell'esempio 15:

```
entity esempio15 is
  port(
    i:  in std_logic;
    clk: in std_logic;
    rst: in std_logic;
    o:  out std_logic
  );
end esempio15;
```

Specifica interna della FSM

➡ La architecture del circuito di esempio 15:

```
architecture FSM of esempio15 is
  type state_type is (S0, S1, S2, S3);
  signal next_state, current_state: state_type;
begin
  state_reg: process(clk, rst)
  begin
    if rst = '1' then
      current_state <= S0;
    elsif rising_edge(clk) then
      current_state <= next_state;
    end if;
  end process;
  --...
```

Specifica interna della FSM

➔ La architecture del circuito di esempio 15:

```
architecture FSM of esempio15 is
```

```
  type state_type is (S0, S1, S2, S3);
```

```
  signal next_state, current_state: state_type;
```

```
begin
```

```
  state_reg: process(clk, rst)
```

```
  begin
```

```
    if rst = '1' then
```

```
      current_state <= S0;
```

```
    elsif rising_edge(clk) then
```

```
      current_state <= next_state;
```

```
    end if;
```

```
  end process;
```

```
--...
```

Definizione di un tipo
enumerativo per
rappresentare l'elenco
degli stati della macchina

Segnali che
rappresentano
lo stato presente
e lo stato prossimo

Specifica interna della FSM

➡ La architecture del circuito di esempio 15:

```
architecture FSM of esempio15 is
  type state_type is (S0, S1, S2, S3);
  signal next_state, current_state: state_type;
begin
  state_reg: process(clk, rst)
  begin
    if rst = '1' then
      current_state <= S0;
    elsif rising_edge(clk) then
      current_state <= next_state;
    end if;
  end process;
  -- ...
```

Elementi di memoria
della macchina a stati

Specifica interna della FSM

➔ La architecture del circuito di esempio 15 (seconda parte):

```
--...
lambda: process(current_state, i)
begin
  case current_state is
    when S0 =>
      if i='0' then
        next_state <= S1;
      else
        next_state <= S0;
      end if;
    when S1 =>
      if i='0' then
        next_state <= S2;
      else
        next_state <= S0;
      end if;
  end case;
end process;
```

Specifica della funzione
di stato prossimo

```
when S2 =>
  if i='0' then
    next_state <= S2;
  else
    next_state <= S3;
  end if;
when S3 =>
  if i='1' then
    next_state <= S1;
  else
    next_state <= S0;
  end if;
end case;
end process;
```

--...

Specifica interna della FSM

➔ La architecture del circuito di esempio 15 (seconda parte):

```
--...
delta: process(current_state)
begin
  case current_state is
    when S0 =>
      o <= '0';
    when S1 =>
      o <= '0';
    when S2 =>
      o <= '0';
    when S3 =>
      o <= '1';
  end case;
end process;

end FSM;
```

Specifica della funzione
di uscita

Specifica interna della FSM

- ➔ È possibile anche specificare le funzioni di stato prossimo e di uscita tramite un singolo processo

```
--...
lambda_delta: process(current_state, i)
begin
  case current_state is
    when S0 =>
      o <= '0';
      if i='0' then
        next_state <= S1;
      else
        next_state <= S0;
      end if;
    when S1 =>
      o <= '0';
      if i='0' then
        next_state <= S2;
      else
        next_state <= S0;
      end if;
  end case;
end process;
--...
```

Specifica interna della FSM

- ➔ Per descrivere FSM non completamente specificate
 - Possiamo assegnare il valore ' - ' (don't care) all'uscita
 - Per poter assegnare il valore ' - ' allo stato prossimo dobbiamo definire i segnali che rappresentano lo stato presente e lo stato prossimo di tipo `std_logic` (e quindi definire una codifica)



POLITECNICO
MILANO 1863

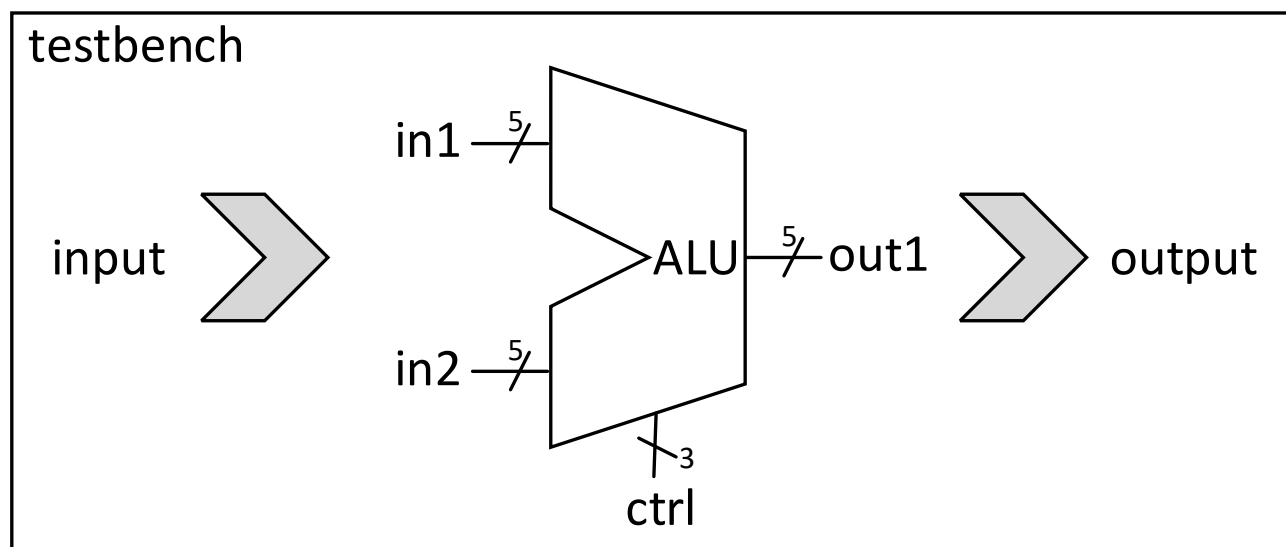
Reti Logiche

VHDL

Testbench

Esempio di circuito 16

- ➔ Vogliamo specificare in VHDL un testbench per l'esempio 8 (utilizziamo il valore di default per il parametro N)
- ➔ Il testbench è un banco di prova da usare durante una simulazione come ambiente che genera gli stimoli per il circuito e raccoglie/analizza gli output



Entity del testbench

➡ La entity del circuito di esempio 16:

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_TEXTIO.ALL;  
use STD.TEXTIO.ALL;
```

```
ENTITY esempio16 IS  
END esempio16;
```

Entity del testbench

➡ La entity del circuito di esempio 16:

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_TEXTIO.ALL;  
use STD.TEXTIO.ALL;
```

```
ENTITY esempio16 IS  
END esempio16;
```

La entity non
contiene alcuna porta

Nel testbench possiamo utilizzare
altri tipi di dato non sintetizzabili
(file, string, ...)

Architecture del testbench

➡ La architecture del circuito di esempio 16:

```

ARCHITECTURE testbench_arch OF esempio16 IS
  COMPONENT esempio8
    generic (
      N : integer := 5
    );
    port(
      in1, in2: in std_logic_vector(N-1 downto 0);
      ctrl: in std_logic_vector(2 downto 0);
      out1: out std_logic_vector(N-1 downto 0)
    );
  END COMPONENT;

  SIGNAL in1 : std_logic_vector (4 DownTo 0) := "00000";
  SIGNAL in2 : std_logic_vector (4 DownTo 0) := "00000";
  SIGNAL ctrl : std_logic_vector (2 DownTo 0) := "000";
  SIGNAL out1 : std_logic_vector (4 DownTo 0) := "00000";
  -- ...

```

Architecture del testbench

➔ La architecture del circuito di esempio 16:

ARCHITECTURE testbench_arch OF esempio16 IS

```
COMPONENT esempio8
  generic (
    N : integer := 5
  );
  port(
    in1, in2: in std_logic_vector(N-1 downto 0);
    ctrl: in std_logic_vector(2 downto 0);
    out1: out std_logic_vector(N-1 downto 0)
  );
END COMPONENT;
```

Componente da testare

Segnali da connettere alle porte dell'istanza

```
SIGNAL in1 : std_logic_vector (4 DownTo 0) := "00000";
SIGNAL in2 : std_logic_vector (4 DownTo 0) := "00000";
SIGNAL ctrl : std_logic_vector (2 DownTo 0) := "000";
SIGNAL out1 : std_logic_vector (4 DownTo 0) := "00000";
```

--...

Architecture del testbench

➡ La architecture del circuito di esempio 16:

```
ARCHITECTURE testbench_arch OF esempio16 IS
```

```
  COMPONENT esempio8
```

```
    generic (
```

```
      N : integer := 5
```

```
    );
```

```
    port(
```

```
      in1, in2: in std_logic_vector(N-1 downto 0);
```

```
      ctrl: in std_logic_vector(2 downto 0);
```

```
      out1: out std_logic_vector(N-1 downto 0)
```

```
    );
```

```
  END COMPONENT;
```

```
  SIGNAL in1 : std_logic_vector (4 DownTo 0) := "00000";
```

```
  SIGNAL in2 : std_logic_vector (4 DownTo 0) := "00000";
```

```
  SIGNAL ctrl : std_logic_vector (2 DownTo 0) := "000";
```

```
  SIGNAL out1 : std_logic_vector (4 DownTo 0) := "00000";
```

```
-- ...
```

Inizializzazione
dei segnali.
Funziona
soltanto in
simulazione!

Architecture **del testbench**

➔ La architecture del circuito di esempio 16 (seconda parte):

```
--...  
BEGIN  
  
    UUT : esempio8  
    PORT MAP (  
        in1 => in1,  
        in2 => in2,  
        ctrl => ctrl,  
        out1 => out1  
    );  
  
--...
```

Architecture del testbench

➔ La architecture del circuito di esempio 16 (seconda parte):

```
--...  
BEGIN
```

```
    UUT : esempio8  
    PORT MAP (  
        in1 => in1,  
        in2 => in2,  
        ctrl => ctrl,  
        out1 => out1  
    );
```

```
--...
```

- Istanziamento del componente da testare
- Connessione dei segnali che verranno stimolati e letti

Architecture del testbench

➔ La architecture del circuito di esempio 16 (terza parte):

```
--...
PROCESS
BEGIN
    WAIT FOR 100 ns;
    -- ----- Current Time: 100ns
    in1 <= "00001";
    in2 <= "00100";
    ctrl <= "001";
    WAIT FOR 200 ns;
    -- ----- Current Time: 300ns
    ctrl <= "010";
    WAIT FOR 200 ns;
    -- ----- Current Time: 500ns
    ctrl <= "011";
    WAIT FOR 1500 ns;
    -- ----- Current Time: 2000ns
    ASSERT(FALSE) REPORT "Simulation OK." SEVERITY FAILURE;
END PROCESS;

END testbench_arch;
```

Architecture del testbench

➔ La architecture del circuito di esempio 16 (terza parte):

```
-- ...
PROCESS
BEGIN
    WAIT FOR 100 ns;
    -- ----- Current Time: 100ns
    in1 <= "00001";
    in2 <= "00100";
    ctrl <= "001";
    WAIT FOR 200 ns;
    -- ----- Current Time: 300ns
    ctrl <= "010";
    WAIT FOR 200 ns;
    -- ----- Current Time: 500ns
    ctrl <= "011";
    WAIT FOR 1500 ns;
    -- ----- Current Time: 2000ns
    ASSERT(FALSE) REPORT "Simulation OK." SEVERITY FAILURE;
END PROCESS;

END testbench_arch;
```

Si utilizza un processo per specificare le forme d'onda da imporre come stimoli in ingresso al componente da testare

Architecture del testbench

➔ La architecture del circuito di esempio 16 (terza parte):

```
-- ...
PROCESS
BEGIN
    WAIT FOR 100 ns;
    -- ----- Current Time: 100ns
    in1 <= "00001";
    in2 <= "00100";
    ctrl <= "001";
    WAIT FOR 200 ns;
    -- ----- Current Time: 300ns
    ctrl <= "010";
    WAIT FOR 200 ns;
    -- ----- Current Time: 500ns
    ctrl <= "011";
    WAIT FOR 1500 ns;
    -- ----- Current Time: 2000ns
    ASSERT(FALSE) REPORT "Simulation OK." SEVERITY FAILURE;
END PROCESS;

END testbench_arch;
```

- Nessuna lista di sensibilità
- Il processo è avviato una volta a tempo 0 (e una volta terminato immediatamente riavviato)

Architecture del testbench

➔ La architecture del circuito di esempio 16 (terza parte):

```
-- ...
PROCESS
BEGIN
    WAIT FOR 100 ns;
    ----- Current Time: 100ns
    in1 <= "00001";
    in2 <= "00100";
    ctrl <= "001";
    WAIT FOR 200 ns;
    ----- Current Time: 300ns
    ctrl <= "010";
    WAIT FOR 200 ns;
    ----- Current Time: 500ns
    ctrl <= "011";
    WAIT FOR 1500 ns;
    ----- Current Time: 2000ns
    ASSERT(FALSE) REPORT "Simulation OK." SEVERITY FAILURE;
END PROCESS;

END testbench_arch;
```

- I segnali di ingresso vengono forzati ad assumere un dato valore
- L'istruzione **wait** forza l'aggiornamento dei segnali scritti e sospende il processo per un intervallo di tempo specificato

Architecture del testbench

➔ La architecture del circuito di esempio 16 (terza parte):

```
--...  
PROCESS  
BEGIN  
    WAIT FOR 100 ns;  
    -- ----- Current Time: 100ns  
    in1 <= "00001";  
    in2 <= "00100";  
    ctrl <= "001";  
    WAIT FOR 200 ns;  
    -- ----- Current Time: 300ns  
    ctrl <= "010";  
    WAIT FOR 200 ns;  
    -- ----- Current Time: 500ns  
    ctrl <= "011";  
    WAIT FOR 1500 ns;  
    -- ----- Current Time: 2000ns  
    ASSERT(FALSE) REPORT "Simulation OK." SEVERITY FAILURE;  
END PROCESS;  
  
END testbench_arch;
```



Ferma la simulazione

Architecture del testbench

➔ La architecture del circuito di esempio 16 (terza parte):

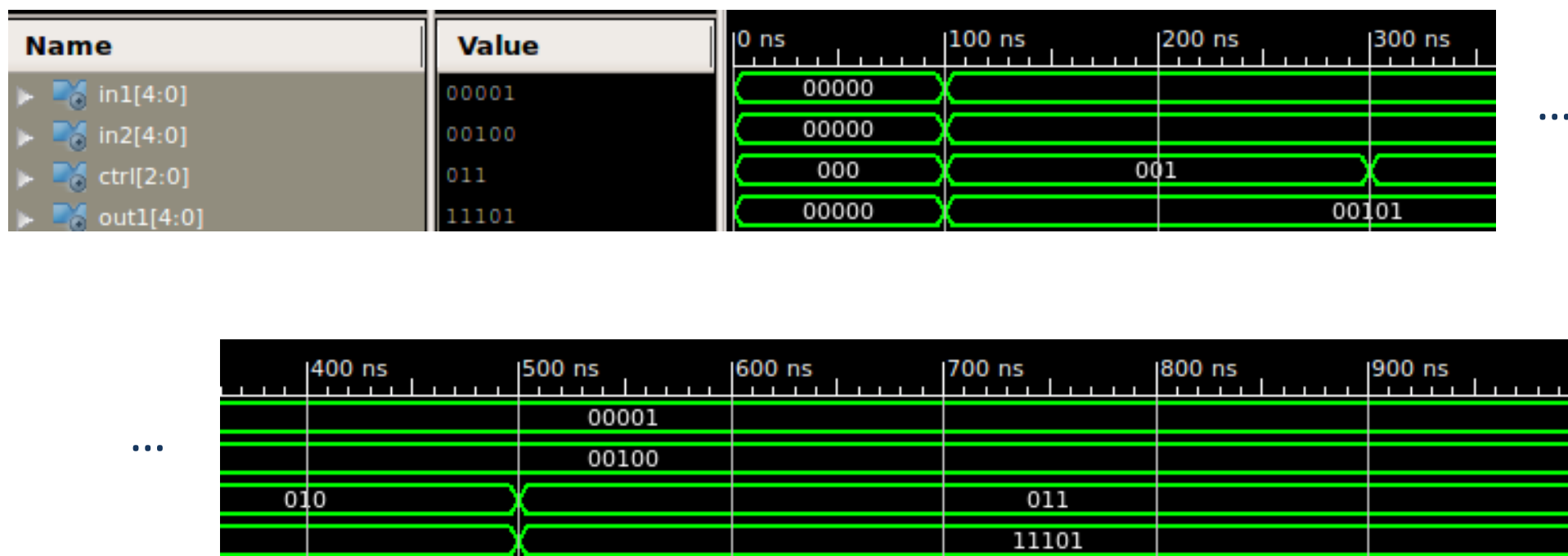
```
--...
PROCESS
BEGIN
    WAIT FOR 100 ns;
    -- ----- Current Time: 100ns
    in1 <= "00001";
    in2 <= "00100";
    ctrl <= "001";
    WAIT FOR 200 ns;
    -- ----- Current Time: 300ns
    ctrl <= "010";
    WAIT FOR 200 ns;
    -- ----- Current Time: 500ns
    ctrl <= "011";
    WAIT FOR 1500 ns;
    -- ----- Current Time: 2000ns
    ASSERT(FALSE) REPORT "Simulation OK." SEVERITY FAILURE;
END PROCESS;

END testbench_arch;
```

- In questo testbench non vengono collezionati gli output
- Si usa direttamente il simulatore per disegnare le forme d'onda

Esecuzione del testbench

➡ Output a video del simulatore:



Architecture del testbench

➔ La architecture del circuito di esempio 16 (terza parte):

```
--...
PROCESS
BEGIN
    WAIT FOR 100 ns;
    -- ----- Current Time: 100ns
    in1 <= "00001";
    in2 <= "00100";
    ctrl <= "001";
    WAIT FOR 200 ns;
    -- ----- Current Time: 300ns
    ctrl <= "010";
    WAIT FOR 200 ns;
    -- ----- Current Time: 500ns
    ctrl <= "011";
    WAIT FOR 1500 ns;
    -- ----- Current Time: 2000ns
    ASSERT(FALSE) REPORT "Simulation OK." SEVERITY FAILURE;
END PROCESS;

END testbench_arch;
```

ATTENZIONE: questa
descrizione non può essere
sintetizzata ma solo simulata!

Architecture del testbench

- ➔ Implementazione alternativa con lettura ed analisi automatizzata dei risultati nella architecture del circuito di esempio 16:

```
--...  
PROCESS  
BEGIN  
    WAIT FOR 100 ns;  
    -- ----- Current Time: 100ns  
    in1 <= "00001";  
    in2 <= "00100";  
    ctrl <= "001";  
    WAIT FOR 0 ns;  
    -- ----- Current Time: 100ns  
    ASSERT (out1="00101") REPORT "Simulation Failure." SEVERITY FAILURE;  
    WAIT FOR 200 ns;  
    --...  
END PROCESS;
```

Architecture del testbench

- ➔ Implementazione alternativa con lettura ed analisi automatizzata dei risultati nella architecture del circuito di esempio 16:

```
-- ...  
PROCESS  
BEGIN  
    WAIT FOR 100 ns;  
    -- ----- Current Time: 100ns  
    in1 <= "00001";  
    in2 <= "00100";  
    ctrl <= "001";  
    WAIT FOR 0 ns;  
    -- ----- Current Time: 100ns  
    ASSERT (out1="00101") REPORT "Simulation Failure." SEVERITY FAILURE;  
    WAIT FOR 200 ns;  
    -- ...  
END PROCESS;
```

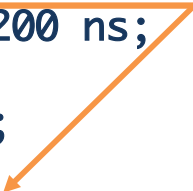


- ➔ Il processo è sospeso per permettere l'aggiornamento dei segnali
 - La simulazione è sospesa per 0 secondi perché il componente testato non presenta ritardi nella generazione degli output

Architecture del testbench

- ➔ Implementazione alternativa con lettura ed analisi automatizzata dei risultati nella architecture del circuito di esempio 16:

```
-- ...  
PROCESS  
BEGIN  
    WAIT FOR 100 ns;  
    -- ----- Current Time: 100ns  
    in1 <= "00001";  
    in2 <= "00100";  
    ctrl <= "001";  
    WAIT FOR 0 ns;  
    -- ----- Current Time: 100ns  
    ASSERT (out1="00101") REPORT "Simulation Failure." SEVERITY FAILURE;  
    WAIT FOR 200 ns;  
    -- ...  
END PROCESS;
```



- ➔ Lettura ed analisi dei risultati
- ➔ La assert blocca l'esecuzione nel caso di valore differente da quello atteso

Architecture del testbench

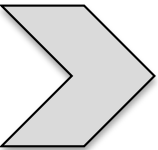
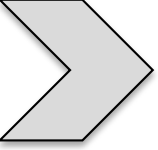
- ➔ Implementazione alternativa con lettura ed analisi automatizzata dei risultati nella `architecture` del circuito di esempio 16:

```
-- ...
PROCESS
BEGIN
    WAIT FOR 100 ns;
    -- ----- Current Time: 100ns
    in1 <= "00001";
    in2 <= "00100";
    ctrl <= "001";
    WAIT FOR 0 ns;
    -- ----- Current Time: 100ns
    ASSERT (out1="00101") REPORT "Simulation Failure." SEVERITY FAILURE;
    WAIT FOR 200 ns;
    -- ...
END PROCESS;
```

- ➔ I dati di input possono anche essere letti da file ed i risultati scritti su file

Esempio di circuito 17

- ➔ Vogliamo specificare in VHDL un testbench per la macchina sequenziale dell'esempio 15

		0	1	U		
Input		S0	S1	S0	0	 Output
		S1	S2	S0	0	
		S2	S2	S3	0	
		s3	S1	S0	1	

Entity del testbench

➡ La entity del circuito di esempio 17:

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
use IEEE.STD_LOGIC_TEXTIO.ALL;  
use STD.TEXTIO.ALL;
```

```
ENTITY esempio17 IS  
END esempio17;
```

Architecture del testbench

➔ La architecture del circuito di esempio 17:

```
ARCHITECTURE testbench_arch OF esempio17 IS
  COMPONENT esempio15 is
    port(
      i:  in std_logic;
      clk: in std_logic;
      rst: in std_logic;
      o:  out std_logic
    );
  END COMPONENT;

  CONSTANT clk_period : time := 10 ns;

  SIGNAL clk : std_logic := '0';
  SIGNAL rst : std_logic := '1';
  SIGNAL i : std_logic := '0';
  SIGNAL o : std_logic := '0';
  -- ...
```

Architecture del testbench

➔ La architecture del circuito di esempio 17:

```
ARCHITECTURE testbench_arch OF esempio17 IS  
  COMPONENT esempio15 is
```

```
    port(  
      i:  in std_logic;  
      clk: in std_logic;  
      rst: in std_logic;  
      o:  out std_logic  
    );
```

```
END COMPONENT;
```

```
CONSTANT clk_period : time := 10 ns;
```

Costante che rappresenta
il periodo di clock

```
SIGNAL clk : std_logic := '0';
```

```
SIGNAL rst : std_logic := '1';
```

```
SIGNAL i : std_logic := '0';
```

```
SIGNAL o : std_logic := '0';
```

```
-- ...
```

Architecture **del testbench**

➔ La architecture del circuito di esempio 17 (seconda parte):

```
--...  
BEGIN  
  
    es15: esempio15  
        PORT MAP(i, clk, rst, o);  
  
    clk_process: PROCESS  
    BEGIN  
        clk <= '0';  
        WAIT FOR clk_period/2;  
        clk <= '1';  
        WAIT FOR clk_period/2;  
    END PROCESS;  
  
--...
```

Architecture **del testbench**

➔ La architecture del circuito di esempio 17 (seconda parte):

```
--...  
BEGIN
```

```
    es15: esempio15  
        PORT MAP(i, clk, rst, o);
```

```
    clk_process: PROCESS  
    BEGIN  
        clk <= '0';  
        WAIT FOR clk_period/2;  
        clk <= '1';  
        WAIT FOR clk_period/2;  
    END PROCESS;
```

- Generazione dell'onda quadra di clock
- Il processo una volta terminato viene immediatamente riavviato

```
--...
```

Architecture del testbench

➔ La architecture del circuito di esempio 17 (terza parte):

```
--...
stimula_process: PROCESS
BEGIN
    WAIT FOR clk_period;
    rst <= '0';
    i <= '0';
    WAIT FOR clk_period;
    i <= '0';
    WAIT FOR clk_period;
    i <= '0';
    WAIT FOR clk_period;
    i <= '1';
    WAIT FOR clk_period;
    i <= '0';
    WAIT FOR clk_period;
    i <= '1';
    WAIT FOR clk_period;
    i <= '1';
    WAIT FOR clk_period;
    ASSERT(FALSE) REPORT "Simulation OK." SEVERITY FAILURE;
END PROCESS;

END testbench_arch;
```