Introduzione al linguaggio VHDL

Linguaggi di descrizione dell'hardware

M. Favalli

DE: Department of Engineering Ferrara

Motivazioni

- Illustrare i vantaggi dell'utilizzo dei linguaggi di descrizione dell'hardware (HDL) nel progetto di sistemi digitali
- Presentare il linguaggio VHDL
- Introdurne alcuni costrutti base per esempi
- Introdurre il ciclo di simulazione del VHDL e il suo modello timing

Sommario

1. Introduzione
2. Esempi
3. Modelli dei componenti
4. Modello timing del VHDL

Introduzione

Ragioni per l'introduzione di HDL

- Torniamo ai primi anni 80, in cui i componenti LSI avevano al massimo poche migliaia di transistori
- Come venivano progettati questi oggetti?
  - schematici a livello gate
  - pochissimi strumenti di EDA
- Con l'aumentare della densitá di integrazione, gli schematici iniziavano a occupare stanzoni in cui si disegnava con passerelle
- Esigenze molto sentite erano la verifica e la documentazione
- Alcune ditte di software iniziarono a sviluppare strumenti per la simulazione al livello logico con HDL proprietari
  - questo approccio metteva in difficoltá i produttori di circuiti integrati

Gli HDL come strumento per gestire la complessitá

- Seguendo la legge di Moore la complessitá dei circuiti integrati continuava ad aumentare
- Per gestirla sono stati sviluppati diversi strumenti e algoritmi
- Fra gli strumenti piú importanti ci sono gli HDL che devono supportare
  - **diversi livelli di astrazione**
  - **progetti di tipo gerarchico**
  - **strumenti di sintesi in aggiunta a quelli di verifica**
- Questa esigenza é tuttora molto rilevante perché la complessitá continua da aumentare

Realizzazione del VHDL - I

- Necessitá di svincolare la progettazione di IC da HDL proprietari
- Sviluppato nell'ambito del programma DARPA Very High Speed Integrated Circuits (VHSIC)
  - 1980
  - sforzo per migliorare la tecnologia VLSI
  - bisogno di un linguaggio di descrizione dell'hardware comune
  - $17M per lo sviluppo diretto del **VHDL** (Very High Speed Integrated Circuit Hardware Description Language)
  - $16M per la realizzazione di strumenti di progettazione basati sul VHDL
- Basato sul linguaggio ADA

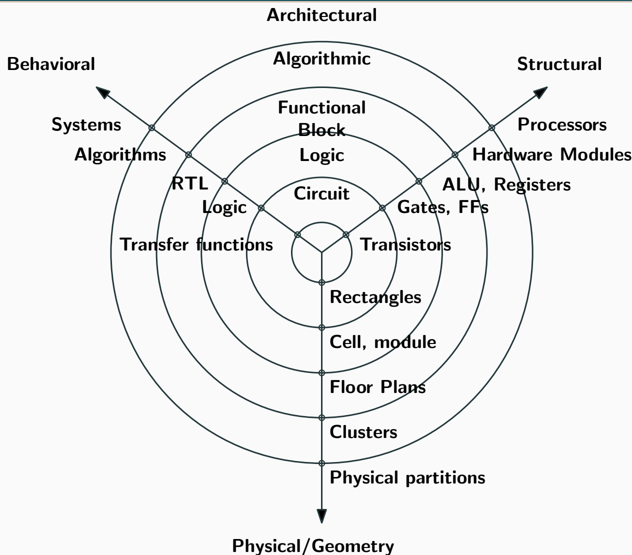
Realizzazione del VHDL - II

- Nel 1983, Intermetrics, IBM and Texas Instruments furono incaricate di sviluppare il VHDL
- Nel 1985, rilascio della versione finale del linguaggio: VHDL Version 7.2
- Nel 1987, il VHDL divenne IEEE Standard 1076-1987 e nel 1988 ANSI standard
- Nel 1993, il VHDL fu ristandardizzato per semplificarne l'utilizzo e per potenziarlo
- Nuovi standard nel 2002 e 2008

Ragioni per l'utilizzo del VHDL

- Il VHDL consente di modellare l' hardware a diversi **livelli di astrazione** da quello gate a quello di sistema
- Il VHDL puó supportare **strumenti automatici per la sintesi e per la verifica**
- Il VHDL mette a disposizione meccanismi per il progetto digitale e la sua documentazione
- Alternative: Verilog, SystemVerilog, SystemC

Gajski and Kuhn chart: rappresentazioni di un sistema digitale



Vantaggi del VHDL (e degli HDL)

- É **indipendente dalla tecnologia**
- Permette di descrivere una grande varietá di hardware digitale a **diversi livelli di astrazione**
- Permette di usare diverse metodologie di progetto
- Facilita la comunicazione attraverso un linguaggio standard
- Permette una migliore gestione dei progetti (design reuse)
- É particolarmente versatile (secondo me il suo vantaggio rispetto ad altri HDL)
- Ha dato luogo a standard : WAVES, VITAL, Analog VHDL

Linguaggi di descrizione dell'hardware - Linguaggi di descrizione del software (imperativi)

Hardware

Si descrive una rete logica come insieme di componenti interconnessi o mediante il suo comportamento

Sintesi: realizzazione del circuito mediante una specifica tecnologia

Simulazione: realizzazione di un modello utilizzabile per predire il comportamento del sistema per un dato insieme di stimoli

Software

Si descrive un algoritmo come sequenza di operazioni eseguite da una specifica architettura hardware (Von-Neumann - o sistemi distribuiti)

Compilazione: traduzione del programma nel linguaggio macchina di una specifica CPU

Esecuzione: da parte di una CPU

Sintesi logica (I)

HDL behavioral

analysis

intermediate
data structures

behavior

RTL

logic

technology
mapping

optimization

technology

area, power and
delay estimates

component library

physical
design

Sintesi logica (II)

- **La sintesi logica puó essere vista come l'insieme di operazioni svolte per passare dalle specifiche funzionali a una descrizione al livello gate di una rete che rispetta tali specifiche**
- Tale processo puó passare attraverso diversi stadi corrispondenti a diversi livelli di specifica e implementazione
- Alle specifiche funzionali si aggiungono anche obbiettivi che riguardano costo (area), prestazioni (ritardo, banda), affidabilitá e consumo di potenza

Lo spazio di progetto

- Ciascun passo della sintesi é accompagnato da un processo di **ottimizzazione**, ovvero dalla ricerca di una soluzione soddisfacente in uno spazio le cui coordinate sono
  - **costo (area)**
  - **prestazioni (ritardo, banda)**
  - affidabilitá
  - consumo di potenza
- La complesitá dei sistemi attuali richiede l'uso di sistemi di Electronic Design Automation (EDA) per esplorare tale spazio
- Proseguendo con il parallelo col software si puó notare che nella sintesi si ottimizza il progetto rispetto a una data tecnologia digitale, nel software si ottimizza il codice macchina rispetto a un architettura obbiettivo

Verifica

- La verifica di progetto procede in senso inverso alla sintesi logica e puó essere fatta in due modi diversi

  Simulazione: in generale produce risultati parziali

  Verifica formale: quando e' fattibile, produce risultati completi

- Entrambi gli approcci hanno una complessitá computazionale esponenziale
- La verifica viene fatta ad ogni passo di progetto
- La verifica non si limita ad aspetti funzionali, ma consente di valutare anche le prestazioni o l'affidabilitá

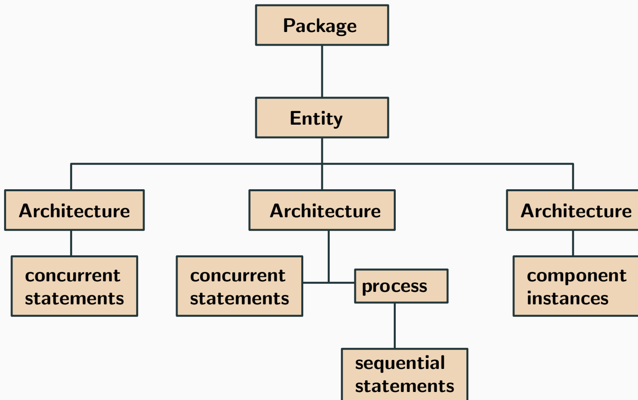
Simulazione vs verifica formale

- Sia la simulazione che la verifica formale utilizzano un modello del componente da verificare
- **La simulazione calcola la risposta di tale modello a un insieme di stimoli e la confronta con quella attesa**
  - per componenti di dimensioni realistiche la simulazione non é mai completa
  - per simulare in maniera esaustiva un sommatore a 64 bit sarebbero necessari $\mathbf{2^{129}}$ vettori di ingresso con un tempo di calcolo superiore all'etá di questo universo
- **Nella verifica formale si utilizza un modello del componente e uno del suo comportamento corretto (specifiche) provando la loro equivalenza o meno**
  - i circuiti di grandi dimensioni possono non essere trattabili, in questo caso, diversamente dalla simulazione, non si hanno risultati parziali

Struttura di un progetto VHDL

- Un progetto complesso descritto in VHDL ha dei punti in comune con la descrizione di un algoritmo in un linguaggi ad alto livello
- Approccio strutturato alla programmazione
- La descrizione puó passare attraverso diversi stadi nei quali alcune parti del progetto vengono specificate con maggiori dettagli
- **Possono essere possibili differenti viste dello stesso progetto (livello di astrazione, libreria di componenti)**

Visione di insieme



Esempi

Conoscenze

- In questi esempi si assume una conoscenza di base di
  - algebra di commutazione
  - sintesi di reti combinatorie
  - componenti sequenziali

Esempio: half-adder

- L' half-adder (HA) é un componente combinatorio alla base di molte reti aritmetiche (sommatori, moltiplicatori)
- Il componente ha 2 ingressi (x e y) e 2 uscite ($result$ e $carry$)
- Le uscite codificano in binario la somma (aritmetica) dei due bit in ingresso (si interpreti $carry$ come bit di maggior peso)
- Tabella di veritá

x	y	$carry$	$result$
0	0	0	0
0	1	0	1
1	1	1	0
1	0	0	1

Esempio: half-adder

- Espressione aritmetica delle uscite:

$$result = (x + y)_{mod2} \qquad carry = x \cdot y$$

- Mediante metodi di sintesi si ottiene l'espressione delle uscite con gli operatori dell'algebra di commutazione

$$result = (x \wedge \neg y) \vee (\neg x \wedge y) = xy' + x'y \qquad carry = x \wedge y = xy$$

- Espressioni descritte mediante l'operatore di somma modulo 2 (aritmetica) o exclusive-OR (XOR, logica)

$$result = x \oplus y \qquad carry = xy$$

- Il VHDL in teoria supporta tutte queste descrizioni, qui focalizzeremo su quelle logiche che sono tipicamente utilizzate nella descrizione di sistemi digitali

Esempio: half-adder con enable

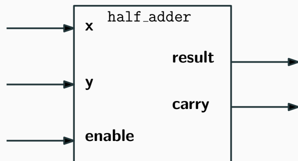
- Le specifiche del nostro esempio prevedono che sia presente anche un ingresso di $\boldsymbol{enable}$
- Quando $\boldsymbol{enable = 1}$, il comportamento é quello di un half-adder
- Quando $\boldsymbol{enable = 0}$, entrambe le uscite vanno a 0
- L'algebra di commutazione consente di scrivere le espressioni delle uscite anche in questo caso
- Il VHDL consente comunque di descrivere il componente anche senza tali espressioni

Progetto VHDL di un HA: dichiarazione di `entity`

Il primo passo é la **`entity`** declaration che definisce l'interfaccia del componente

- porte di input e output
- si noti che il VHDL é case unsensitive

```vhdl
entity half_adder is
  port(x,y,enable : in bit;
       carry,result : out bit);
end entity half_adder;
```



Progetto VHDL di un HA: dichiarazione di `architecture`

Descrizione comportamentale simulabile e sintetizzabile

```
architecture behav_a of half_adder is
  begin
    process (x, y, enable)
      begin
        if (enable = '1') then
          if (x/=y) then
            result <= '1';
          else
            result <= '0';
          end if;
          if (x='1' and y='1') then
            carry <= '1';
          else
            carry <= '0';
          end if;
        else
            carry <= '0';
            result <= '0';
          end if;
      end process;
    end architecture behav;
```

Progetto VHDL di un HA: architettura alternativa

Gli operatori logici possono essere (overloaded) applicati anche al tipo di dato **bit** $\Rightarrow$ descrizione behavioral piú compatta

```
architecture behav_b of half_adder is
  begin
    process (x, y, enable)
      begin
        if (enable = '1') then
          result <= x xor y;
          carry <= x and y;
        else
          carry <= '0';
          result <= '0';
        end if;
      end process;
  end architecture behav;
```

viene eseguito ogni volta che uno del segnali nella **sensitivity list** cambia

Progetto VHDL di un HA: descrizione di tipo dataflow

- Si possono anche utilizzare soltanto espressioni logiche
- Anche questa descrizione é simulabile e sintetizzabile ed é riferita alla stessa entity delle precedenti

```
architecture dataflow of half_adder is
begin
  carry <= enable and (x and y);
  result <= enable and (x xor y);
end architecture dataflow;
```

Note

- Si noti che le 3 espressioni sono funzionalmente equivalenti
- Fino al VHDL-2008 non veniva considerata la proprietá associativa di congiunzione e disgiunzione
  - Per cui l'espressione abc andava descritta come:

```
a and (b and c)
```

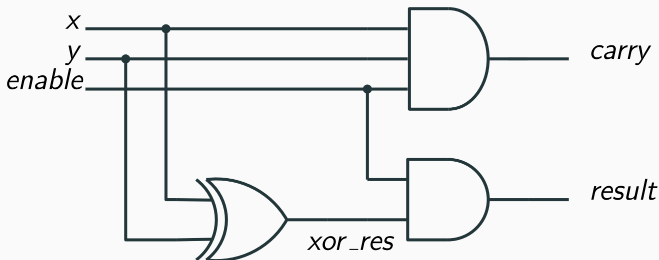
- Inoltre venivano ignorate le regole di prioritá degli operatori forzando l'uso di parentesi
  - Per cui l'espressione $a + bc$ andava descritta come:

```
a or (b and c)
```

- Si puó osservare la flessibilitá del linguaggio che supporta diversi stili descrittivi equivalenti

Progetto VHDL di un HA: descrizione strutturale

- Si supponga sia disponibile una **libreria di componenti elementari** al livello logico e che tali componenti corrispondano a celle fisiche
- In questo caso il componente puó essere descritto al livello strutturale come una rete logica



VHDL strutturale - I

```
architecture struct of half_adder is
  component and2
    port (in0, in1 : in bit;
          out0 : out bit);
  end component;
  component and3
    port (in0, in1, in2 : in bit;
          out0 : out bit);
  end component;
  component xor2
    port (in0, in1 : in bit;
          out0 : out bit);
  end component;
  for all : and2 use entity work.and2_cmos(and2_a);
  for all : and3 use entity work.and3_cmos(and3_a);
  for all : xor2 use entity work.xor2_cmos(xor2_a);
  signal xor_res : bit; -- internal signal
```

riferimento alla stessa **entity** delle architetture nei lucidi precedenti

si suppone che nello spazio di lavoro corrente **work** sia stato definita un architettura **and2_a** per la entity **and2_cmos**

VHDL strutturale - II

```
begin

  a0 : and2 port map (enable, xor_res, result);
  a1 : and3 port map (x, y, enable, carry);
  x0 : xor2 port map (x, y, xor_res);

end architecture struct;
```

Esempio di componente

```
entity and2_cmos is
  port (in0,in1: in bit;
        out0: out bit);
end entity and2_cmos;

architecture and2_a of and2_cmos is
begin
  out0<=in0 and in1;
end architecture and2_cmos;
```

Note

- Perché vengono supportati stili diversi di progettazione anche solo nell'ambito comportamentale?
  - supporto a diversi stili e fasi di progetto
  - efficienza nello scrivere il codice
- **Differenza fra operatori logici** (es. `and`) e **componenti** (es. `and2`)
  - gli operatori logici operano sul tipo di dato booleano e sono stati estesi ad altri tipi di dato che rappresentano i segnali (es. `bit`)
  - i componenti fanno riferimento a celle di librerie che sono state descritte tramite un modello `entity/architecture`

Decoder

- Componente con n ingressi $\{a_{n-1}, a_{n-2}, ..., a_0\}$ e 2^n uscite $\{o_{2^n-1}, o_{2^n-2}, ..., o_0\}$
- L'uscita il cui indice corrisponde al numero naturale codificato in binario dagli ingressi vale 1 e le altre valgono 0
- $o_k = 1$ se $k = \sum_{i=0}^{n-1} a_i 2^i$ e 0 altrimenti
- Viene prodotto in uscita un codice del tipo 1-out-of-n
- Componente utilizzato ad esempio per selezionare una cella di memoria dato il suo indirizzo

Configurazioni binarie, variabili binarie e numeri

- **Configurazione binaria** di n bit: $b = \{0, 1\}^n$
  - esempio $n = 4$, $b = 1110$
- Insieme di n variabili binarie $X = \{x_{n-1}, \ldots, x_1, x_0\}$ che puó assumere un valore in $\{0, 1\}^n$
  - esempio $n = 4$, $X = \{x_3, x_2, x_1, x_0\}$
- **Mintermine**: prodotto logico di tutte le variabili in X prese in forma vera o negata
  - a ogni mintermine é associata una configurazione binaria: il mintermine vale 1 se e solo se le variabili hanno il valore 0 (se in forma negata) o 1 (se in forma vera)
  - esempio $n = 4$, $m = x_3 x_2 x_1 x_0' \leftrightarrow 1110$
- **Numero naturale** v in base 2 associato a una configurazione binaria b: $v = \sum_{i=0}^{n-1} 2^i b_i$
  - esempio $n = 4$, $b = 1110 \leftrightarrow 2^3 + 2^2 + 2^1 = 14$

Esempi di decoder

Tabella di veritá per $n = 1$

a_0	o_1	o_0
0	0	1
1	1	0

Espressione delle uscite

$o_0 = a_0'$

$o_1 = a_0$

Tabella di veritá per $n = 2$

a_1, a_0	o_3	o_2	o_1	o_0
00	0	0	0	1
01	0	0	1	0
10	0	1	0	0
11	1	0	0	0

Espressione delle uscite

$o_0 = a_1'a_0'$

$o_1 = a_1'a_0$

$o_2 = a_1a_0'$

$o_3 = a_1a_0$

Modello di decoder VHDL con enable - entity

Si tratta di un decoder a 2 ingressi e 4 uscite cui é stato aggiunto un segnale di enable

- quando tale segnale é a 1 il comportamento é quello di un decoder (descritto nel lucido precedente)
- quando é a 0 tutte le uscite vanno a 0.

```
entity decoder_2_to_4 is
  port (en, a0, a1: in bit;
        o0, o1, o2, o3: out bit);
end decoder_2_to_4;
```

Modello di decoder VHDL con enable - behavioral architecture

```
architecture behavioral of decoder_2_to_4 is
begin
  process (en, a0, a1)
    begin
      if (en='1') then
        o0 <= (not a0) and (not a1);
        o1 <= a0 and (not a1);
        o2 <= (not a0) and a1;
        o3 <= a0 and a1;
      else
        o0 <= '0';
        o1 <= '0';
        o2 <= '0';
        o3 <= '0';
      end if;
     end process;
end architecture decoder_2_to_4;
```

Modello di decoder VHDL con enable - dataflow architecture

In maniera simile al caso dell'HA, il segnale di enable puó essere inserito nelle espressioni delle uscite.

```
architecture dataflow of decoder_2_to_4 is
signal na0,na1: bit;
begin
    na0 <= not a0;
    na1 <= not a1;
    o0 <= (na0 and na1) and en;
    o1 <= (a0 and na1) and en;
    o2 <= (na0 and a1) and en;
    o3 <= (a0 and a1) and en;

end architecture dataflow;
```

Note

- Il tipo di dato **`bit`** non é sufficiente a descrivere i possibili comportamenti di un segnale in un sistema digitale
- Ad esempio, le incertezze sul valore di uscita dei flip-flop all'accensione si descrivono con un valore denotato come X (unknown)
- Vedremo che il VHDL mette a disposizione la possibilitá di costruire tipi di dato enumerati
- Tale possibilitá é stata sfruttata per costruire il tipo di dato **`std_logic`** che é uno standard IEEE ampiamente utilizzato al livello industriale

Modello VHDL strutturale del decoder

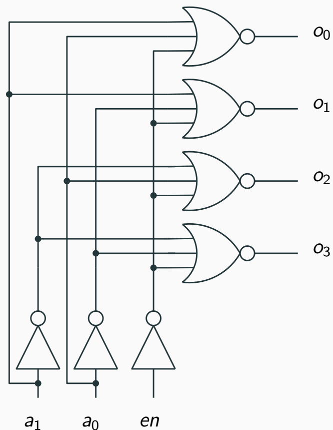
- Utilizzeremo il tipo di dato `std_logic` per costruire un decoder al livello strutturale
- Per utilizzare tali segnali standard nei progetti industriali, dobbiamo definire una nuova entity
- Supporremo che i componenti disponibili si trovino in una libreria `cmos_simple` che mette a disposizione le porte logiche:

tipo di gate	nome	espressione
inverter	**not1_cmos**	$out_1 = in_1'$
nand a 2 ingressi	**nand2_cmos**	$out_1 = (in_1 \, in_2)'$
nand a 3 ingressi	**nand3_cmos**	$out_1 = (in_1 \, in_2 \, in_3)'$
nor a 2 ingressi	**nor2_cmos**	$out_1 = (in_1 + in_2)'$
nor a 3 ingressi	**nor3_cmos**	$out_1 = (in_1 + in_2 + in_3)'$

Modello VHDL strutturale del decoder

- Mancano porte che realizzino direttamente un l'espressione dell'AND, per cui si manipolano le espressioni delle uscite per mapparle sui componenti disponibili
- Esempio:
  $o_1 = a_1' a_0 en = (a_1' \, a_0 \, en)'' = (a_1'' + a_0' + en')' = (a_1 + a_0' + en')'$
- Questa espressione é il NOR di a_1, a_0' e en'
- Quindi la rete puó essere realizzata con i gate **not1_cmos** e **nor3_cmos**
- Nella sintesi queste operazioni vengono fatte nel technology mapping

Schema logico



Modello del decoder - I

```
library ieee, cmos_simple;
use ieee.std_logic_1164.all, cmos_simple.all;
entity decoder is
  port(a0, a1, en: in std_logic;
       o0, o1, o2, o3: out std_logic);
end entity decoder;

architecture struct of decoder is
component not1
   port(in1: in std_logic;
        out1: out std_logic);
end component;
component nor3
   port(in1, in2, in3: in std_logic;
        out1: out std_logic);
end component;
for all not1: use entity cmos_simple.not1_cmos(behav);
for all nor3: use entity cmos_simple.nor3_cmos(behav);
```

Modello del decoder - II

```
signal not_en, not_a0, not_a1: std_logic;
begin
   g0: not1 port map (in1 => a0, out1 => not_a0);
   g1: not1 port map (in1 => a1, out1 => not_a1);
   gen: not1 port map (in1 => en, out1 => not_en);
   g2: nor3 port map (in1 => a0, in2 => a1,
                           in3 => not_en, out1 => o0);
   g3: nor3 port map (in1 => not_a0, in2 => a1,
                           in3 => not_en, out1 => o1);
   g4: nor3 port map (in1 => a0, in2 => not_a1,
                           in3 => not_en, out1 => o2);
   g5: nor3 port map (in1 => not_a0, in2 => not_a1,
                           in3 => not_en, out1 => o3);
end architecture struct;
```

Note

- Le descrizioni comportamentali e dataflow possono essere considerate al livello di design entry
- La descrizione strutturale puó essere considerata come il risultato di un processo di sintesi (manuale o automatico) a partire dalle specifiche
- Tale descrizione puó essere fornita in ingresso a tool che relizzano la parte fisica del processo di sintesi per circuiti integrati o FPGA
- Svantaggi della descrizioni viste fino ad ora:
  - cosa succede se il numero di ingressi aumenta?

Multiplexer

- Il multiplexer é un componente che ha $2^n + n$ ingressi partizionati fra:
  - ingressi dati $\{x_0, x_1, ..., x_{2^n-1}\}$
  - ingressi di selezione $\{s_0, s_1, ..., s_{n-1}\}$
- L'uscita é data da $y = x_i \mid i = \sum_{j=0}^{n-1} s_j 2^j$ (ove la sommatoria é aritmetica)
- Tradotta in un espressione booleana $y = \sum_{i=0}^{2^n-1} p_i x_i$ (ove la sommatoria é logica e p_i é il termine prodotto corrispondente alla configurazione i)
- Il multiplexer puó essere visto come un componente che riporta in uscita il valore dell'ingresso dati selezionato dagli ingressi di selezione

Espressione dell'uscita per MPX con $n = 1$, $n = 2$ e $n = 3$

- Espressioni booleane
- $n = 1$, un bit di selezione (s_0), due ingressi dati (x_0, x_1)

  $y = x_0 s_0' + x_1 s_0$
- $n = 2$

  $y = x_0 s_1' s_0' + x_1 s_1' s_0 + x_2 s_1 s_0' + x_3 s_1 s_0$
- $n = 3$

  $y = x_0 s_2' s_1' s_0' + x_1 s_2' s_1' s_0 + x_2 s_2' s_1 s_0' + x_3 s_2' s_1 s_0 + x_4 s_2 s_1' s_0' +$
  $x_5 s_2 s_1' s_0 + x_6 s_2 s_1 s_0' + x_7 s_2 s_1 s_0$

Modello VHDL dataflow di un MPX a 2 bit di selezione

Costrutto **when-else**

```
library ieee;
use ieee.std_logic_1164.all;
entity mpx_4_to_1 is
     PORT (s : in std_logic_vector(1 downto 0);
           x : in std_logic_vector(3 downto 0);
           y : out std_logic);
end entity mpx_4_to_1;

architecture function_table_we OF mpx_4_to_1 is
begin
  y <=x(0) when s = "00" else
      x(1) when s = "01" else
      x(2) when s = "10" else
      x(3) when s = "11" else
      'X';
end architecture function_table_we;
```

array di segnali

boolean condition

costrutto **when-else**

tiene conto dei casi rimanenti (ingressi a '1')

Modello VHDL dataflow di un MPX a 2 bit di selezione

Costrutto **with-select**

costrutto **with-select**

```
architecture function_table_ws OF mpx_4_to_1 is
begin
  with s select
  y <=x(0) when "00",
      x(1) when "01",
      x(2) when "10",
      x(3) when "11",
      'X' when others;
end architecture function_table_ws;
```

value of **s**

Confronto fra i due costrutti (`when-else` e `with-select`)

- Sembrano piuttosto simili, il **`when-else`** é piú generale
- Supponiamo di aggiungere al componente un segnale di *en* che manda a 0 l'uscita quando vale 0

```
architecture dataflow OF mpx_4_to_1_en is
begin
  y <=x(0) when s = "00" and en='1' else
      x(1) when s = "01" and en='1' else
      x(2) when s = "10" and en='1' else
      x(3) when s = "11" and en='1' else
      '0' when en = '0' else
      'X';
end architecture dataflow;
```

- Con il **`with-select`** bisognerebbe riportare tutta la tabella di veritá del nuovo multiplexer

Modelli dei componenti

Modelli dei componenti VHDL

- Una descrizione completa di un componente VHDL richiede una **`entity`** e un **`architecture`**
- La **`entity`** definisce l'interfaccia dei componenti
- L'**`architettura`** definisce la funzione dei componenti
- Diverse possibili architetture possono essere utilizzate per la stessa **`entity`**
- Tre diversi tipi di descrizione di un componente in VHDL:
  - strutturale
  - comportamentale (behavioral)
  - timing e delay

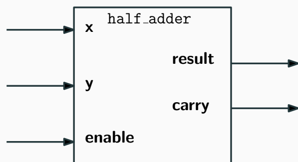
Modelli di componenti VHDL

- L'unitá fondamentale per la descrizione del behavior di un componente é il **processo**
- I processi possono essere definiti esplicitamente o implicitamente e sono contenuti nelle architetture
- **Il meccanismo primario di comunicazione tra processi é il segnale**
- Modo in cui viene simulato un componente VHDL
  - l'esecuzione di un processo risulta in nuovi valori assegnati a segnali che sono poi accessibili ad altri processi
  - un segnale puó essere reso accessibile a processi in altre architetture connettendo il segnale a porte nelle entities associate alle due architetture

Dichiarazione di `entity`

- Lo scopo primario di una **`entity`** é dichiarare i segnali nell'interfaccia del componente
  - tali segnali sono elencati nella dichiarazione di **`port`**
- Con la dichiarazione di **`generic`** si possono fornire ulteriori parametri al componente

```
entity half_adder is
  generic(prop_delay: time:=200 ps);
  port(x,y,enable : in bit;
       carry,result : out bit);
end entity half_adder;
```



Dichiarazione di `entity`: `port`

- `port` dichiara i segnali di interfaccia verso il mondo esterno

```
port (signal_name : mode data_type);
```

- Nella `port` sono presenti tre campi
  - **nome**
  - **modo**
  - **tipo di dato**
- Esempio di clausola `port`:

```
port ( input : in bit_vector(3 downto 0);
       ready, output : out bit );
```

Segnali di interfaccia (i.e. 'ports') dello stesso modo e tipo o sottotipo si possono dichiarare nella stessa linea

Dichiarazione di `entity`: modo delle `port`

- Il modo descrive la direzione cui viaggia il segnale rispetto al componente
- Ci sono cinque modi disponibili per una porta:
  - **`in`** - il dato entra nella porta e si puó solo leggere
  - **`out`** - il dato esce dalla porta e si puó solo scrivere
  - **`buffer`** - il dato puó muoversi in entrambe le direzioni, ma solamente un driver per volta puó essere attivo (da evitare)
  - **`inout`** - il dato si puó muovere in entrambe le direzioni, puó essere scritto da un numero qualsiasi di drivers e letto, ma richiede una Bus Resolution Function
  - **`linkage`** - la direzione del dato é sconosciuta (in teoria servirebbero per connettersi a un simulatore analogico)

Dichiarazione di `entity`: `generic`

- Le dichiarazioni di parametri **`generic`** si possono utilizzare per leggibilitá, manutenzione e configurazione
- Sintassi:

```
generic (generic_name : type [:= default_value]);
```

  - il valore di default é opzionale, se manca deve essere fornito quando il componente é instanziato
- Esempio:

```
generic (my_id : integer :=37);
```

  - il generic **`my_id`** ha un valore di default pari a 37 e puó essere referenziato in qualsiasi architettura di questa entity
  - il valore di default puó essere sovrascritto quando il componente veien instanziato

Architetture

- Descrive le operazioni del componente
- Consiste di due parti:
  - **parte dichiarativa**
    - tipo
    - segnale
    - componente
    - sottoprogramma
  - **parte istruzioni**
    - assegnamento concorrente di segnali
    - istruzioni di processo
    - istanze di componenti

Esempio - I

```
-- entity in slide 21
-- used only to provide an instance
architecture mix of half_adder is

-- begin declarative part
component and2
   port(a,b: in bit;
        output: out bit);
end component;
signal tmp0, tmp1 : bit;
for all and2: use entity work.and2(behav);
-- end declarative part
```

Esempio - II

```
begin  -- architecture body
  a0: and2 port map (a=>x, b=>y, output=>tmp0);

  tmp1 <= x xor y;

  process(tmp0,tmp1,enable)
    begin
      if (enable = '0') then
        result <= '0';
        carry <= '0';
      else
        result <= tmp1;
        carry <= tmp0;
      end if;
  end process;

end architecture;  -- end architecture body
```

component instance

concurrent assignment

process

implicit and explicit processes

Architetture con descrizioni di tipo strutturale

- I componenti VHDL definiti in librerie (il VHDL non ha componenti predefiniti) sono instanziati e connessi insieme
- Le descrizioni strutturali possono connettere semplici gate o componenti piú complessi a loro volta descritti in maniera strutturale o comportamentale
- **Il VHDL supporta descrizioni gerarchiche**
- Il VHDL consente di descrivere facilmente strutture altamente ripetitive

Architetture con descrizioni di tipo comportamentale (bheavioral o dataflow)

- Il VHDL mette a dispsizione due stili per descrivere il behavior di un componente
  - **dataflow**: istruzioni concorrenti di assegnamento dei segnali
  - **behavioral**: processi utilizzati per descrivere comportamenti complessi mediante costrutti di linguaggi ad alto livello e.g. variabili, cicli, if-then-else
- Un modello behavioral model puó’ essere largamente indipendente dall’implementazione del sistema

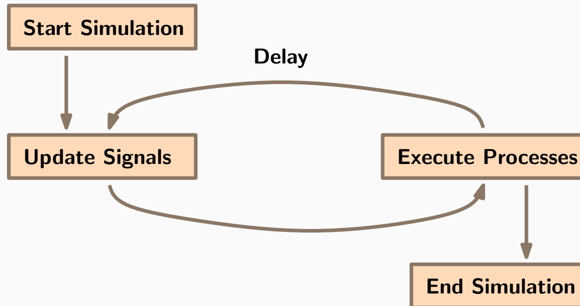
Utilizzo del VHDL

- Modellazione dei sistemi digitali a diversi livelli di astrazione nell'ambito della loro verifica e sintesi
- Consideriamo la verifica tramite **simulazione logica** il cui scopo é riprodurre nel modo piú accurato possibile il comportamento (funzionale e timing) dei sistemi digitali
- Il modello VHDL del componente e il simulatore devono corrispondere a questa esigenza
- Vedremo quindi quella che puó essere considerata la macchina virtuale del VHDL
- Rispetto a un linguaggio software sequenziale come il C dobbiamo tenere in conto del comportamento concorrente dei sistemi digitali
  - se in un circuito half-adder cambia un ingresso le due uscite vengono calcolate in parallelo

Modello timing del VHDL

Modello timing

Il VHDL utilizza il seguente ciclo per modellare il comportamento dell'hardware



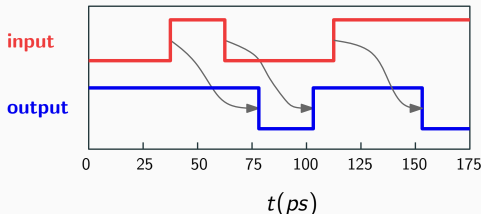
Tipi di ritardo

- Tutti gli assegnamenti di segnali del VHDL devono avere un ritardo non nullo prima che il segnale possa assumere il suo nuovo valore
- Il ritardo puó essere in una di queste tre forme:
  - **trasporto** - definisce un ritardo di propagazione finito
  - **inerziale** - definisce sia il ritardo di propagazione sia la minima durata di un impulso in ingresso tale da dare luogo a una risposta sull'uscita
  - **delta** - valore di default se non viene specificato alcun ritardo

Ritardo di tipo trasporto

- Il ritardo di tipo trasporto deve essere specificato con la keyword **`transport`**
- Il segnale assume il suo nuovo valore dopo il ritardo specificato

```
output <= transport not input after 40 ps;
```



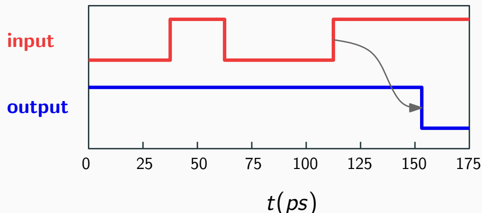
Ritardo di tipo inerziale

- Oltre a specificare il ritardo di propagazione, specifica un valore di soglia per la dimensione degli impulsi di ingresso al di sotto del quale non sono propagati:

```
target <= [reject time_expression] inertial waveform;
```

- Il ritardo inerziale é di default mentre `reject` é opzionale

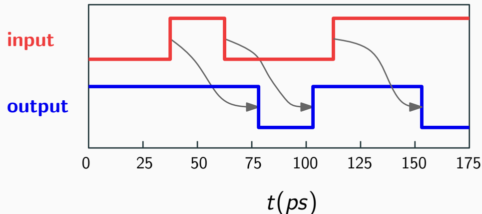
```
output <= not input after 40 ps;
-- Propagation delay and minimum pulse width
-- are 40 ps
```



Ritardo di tipo inerziale

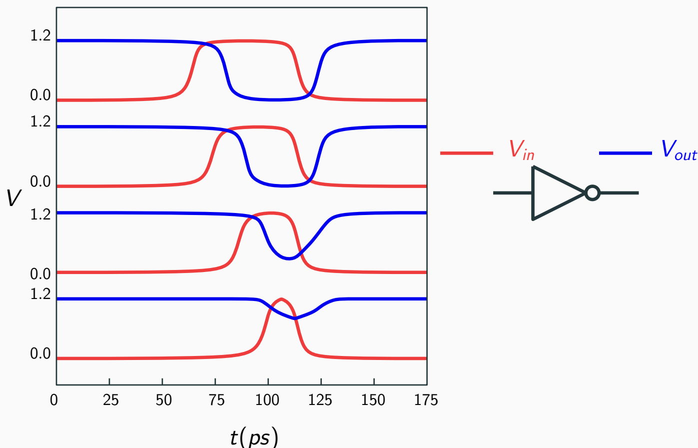
- Esempio di gate con 'inerzia' piú piccola del ritardo di propagazione
- Invertitore con ritardo di propagazione di 40 ps che elimina impulsi di durata inferiore a 20 ps

```
output <= reject 20 ps not input after 40 ps;
```



Confronto fra ritardo di tipo trasporto e inerziale

- Perché il VHDL mette a disposizione i due tipi di ritardo ?
- Simulazione a livello circuitale di un invertitore con applicato in ingresso un impulso di durata decrescente



Confronto fra ritardo inerziale e ritardo di tipo trasporto

- Nessuno dei due modelli é esatto perché al livello logico si perdono dettagli rispetto al livello circuitale
- Quale usare ?
  - il modello di tipo inerziale é piú efficiente perché elimina diversi eventi che non devono essere calcolati dal simulatore
  - il modello di tipo trasporto é conservativo e puó essere utilizzato in reti (come quelle asincrone) in cui un impulso puó dare portare a malfunzionamenti
- Applicazione: consumo di potenza in presenza di hazard

Ritardo Delta

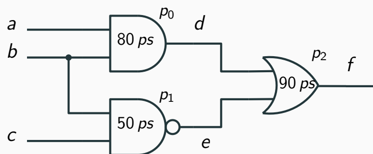
- Default se non si specifica niente
  - nel VHDL l'assegnamento dei segnali non avviene immediate per evitare problemi al simulatore
  - Delta é l'unitá minima di tempo (discreta ma non quantificabile)

```
output <= not input;
-- output assumes a new value in a delta cycle
```

- Consente di supportare un modello concorrente per l'esecuzione dei processi VHDL
  - **L'ordine in cui i processi sono eseguiti dal simulatore non cambia il risultato della simulazione**

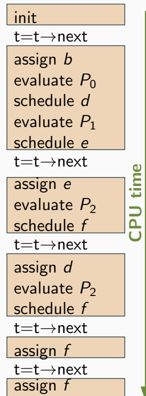
Esempio di simulazione VHDL

- Simulazione su una singola CPU
- **Tempo di simulazione** indipendente dal tempo di CPU
- **Evento**: segnale, nuovo valore e istante dell'assegnamento



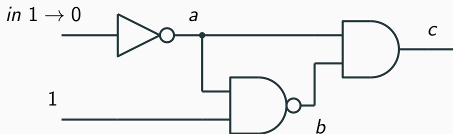
time (ps)	signal values	evaluated processes
0	$a=1,\ b=0,\ c=1,\ d=0,\ e=1,\ f=1$	init
20	$b=1$	P_0 P_1
70	$e=0$	P_2
100	$d=1$	P_2
160	$f=0$	
190	$f=1$	

simulation time



Ritardo Delta

- Si vuole conoscere il comportamento di c supponendo che non ci sia il ritardo Delta
- Dipendentemente dall'ordine in cui la CPU processa i segnali ci sono due possibili casi

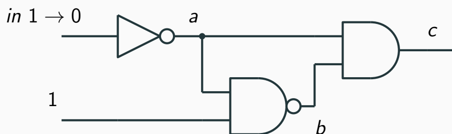


NAND gate valutato prima
in $1 \to 0$
a $0 \to 1$
b $1 \to 0$
c $0 \to 0$

AND gate valutato prima
in $1 \to 0$
a $0 \to 1$
c $0 \to 1$
b $1 \to 0$
c $1 \to 0$

Ritardo Delta

- Si vuole conoscere il comportamento di c utilizzando il ritardo Delta
- Si noti che i cicli Delta si sommano senza mai raggiungere un unitá di tempo



Utilizzo del ritardo Delta

time	Delta	events	eval
0 ns	1	$in\ 1 \rightarrow 0$	INV
	2	$a\ \ 0 \rightarrow 1$	NAND, AND
	3	$b\ \ 1 \rightarrow 0,\ c\ \ 0 \rightarrow 1$	AND
	4	$c\ \ 1 \rightarrow 0$	
1 ns			

Conclusioni

- Si sono visti alcuni esempi di descrizione di semplici componenti nel linguaggio VHDL
- Si é introdotto il modello timing del VHDL dal punto di vista della simulazione logica
- Rimane da vedere la sintassi del linguaggio e da esplorare meglio le sue possibilitá
- Guarderemo anche brevemente alla struttura di un simulatore logico