DEIS
Dipartimento di Elettronica Informatica e Sistemistica
Universita' di Bologna

Introduzione al linguaggio VHDL

PARTE I

Stefano Mattoccia
e-mail: smattoccia@deis.unibo.it
Telefono: +39 051 2093860

Bologna, 4 Marzo 2004

Il **VHDL** e’ un linguaggio per la **sintesi automatica** e la **simulazione** di circuiti digitali

- **VHDL**: VHSIC Hardware Description Language
- **VHSIC**: Very High Speed Integrated Circuit

Standardizzato nel 1993 (IEEE Standard 1076-1993)

Libri

- **Zainalabedin Navabi, *"VHDL: Analysis and modelling of digital systems (2nd Ed)"*, McGraw-Hill 1998**
- **Douglas Perry, *"VHDL - Third Edition"*, McGraw-Hill 1998**
- **Kevin Skahill, *"VHDL for Programmable Logic"*, Addison-Wesley**

Internet

- **Sito di "Reti Logiche L-A" Cesena, www.ingce.unibo.it**
- **http://www.fpga.com.cn/hdl/training/vhdl_intr.pdf**
- **http://www.ee.pdx.edu/~mperkows/CLASS_VHDL_99/005.pdf**
- …….

Dispense corso Reti Logiche

"Introduzione alle esercitazioni di laboratorio"

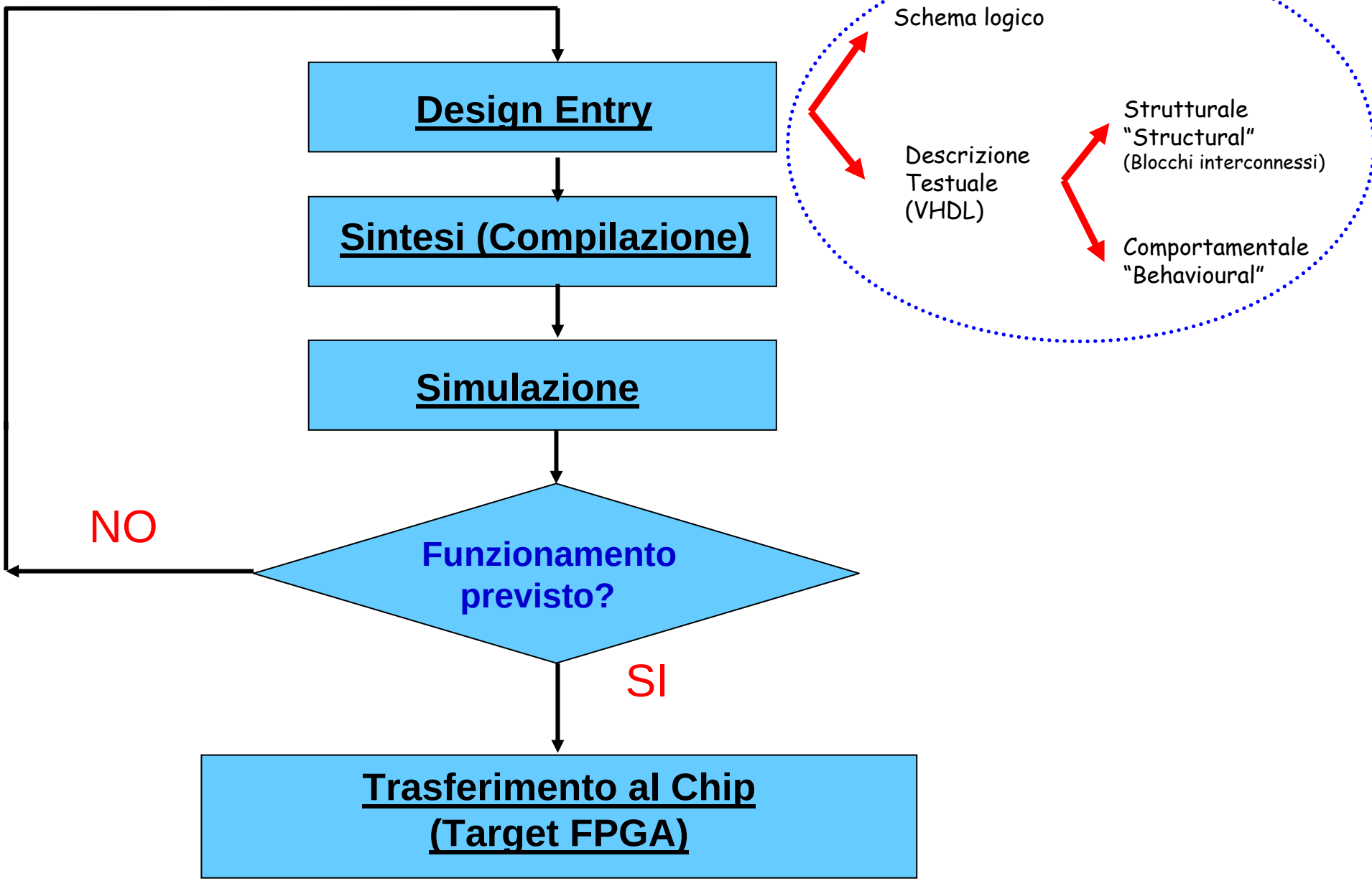
www.ingce.unibo.it/corsi_di_laurea/piano_studi/programma_didattico_01_02/reti_logiche/IndiceMateriale.htm

Software

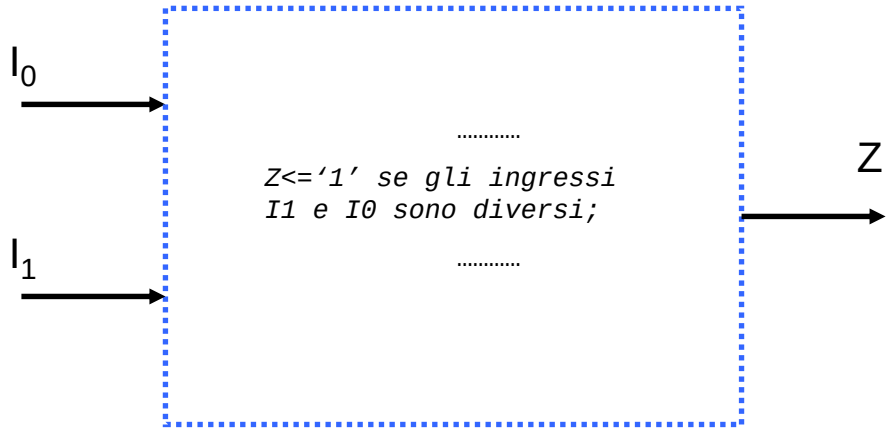
Altera Max Plus (versione 10)

www.altera.com

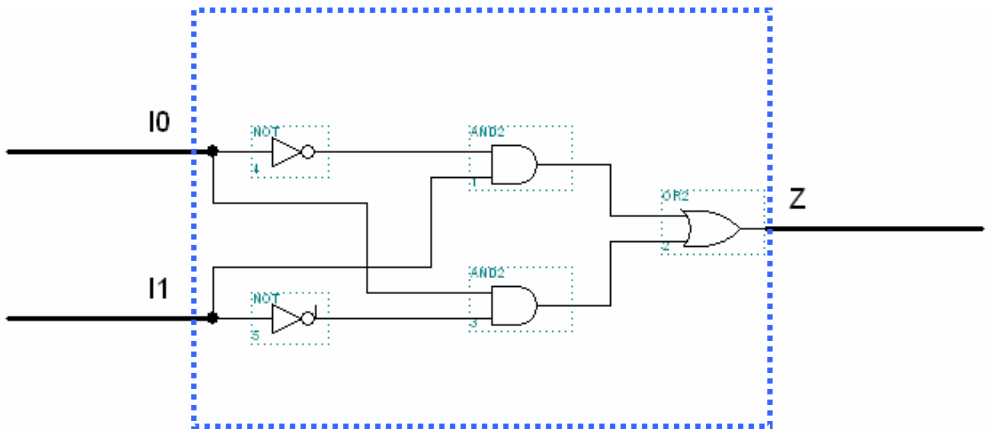
www.ingce.unibo.it/corsi_di_laurea/piano_studi/programma_didattico_01_02/reti_logiche/IndiceMateriale.htmd


Design Entry
Sintesi (Compilazione)
Simulazione
Funzionamento previsto?
NO
SI
Trasferimento al Chip (Target FPGA)
Schema logico
Descrizione Testuale (VHDL)
Strutturale "Structural" (Blocchi interconnessi)
Comportamentale "Behavioural"

Nella rappresentazione comportamentale (‘***Behavioural***’) un componente viene descritto mediante il suo comportamento ‘ingresso-uscita’.
Si descrive come dovra’ rispondere la rete ma non la sua struttura.

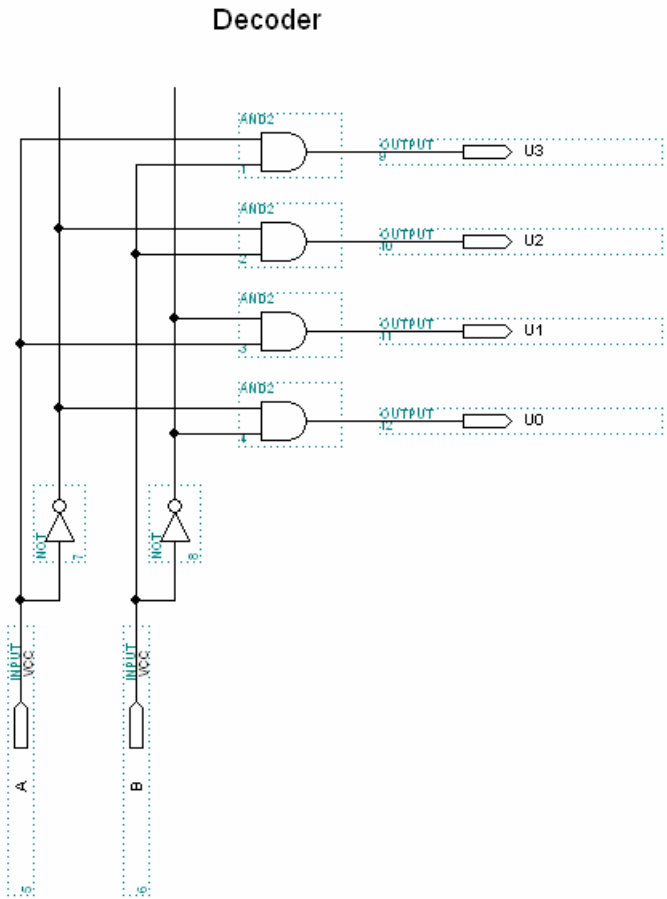


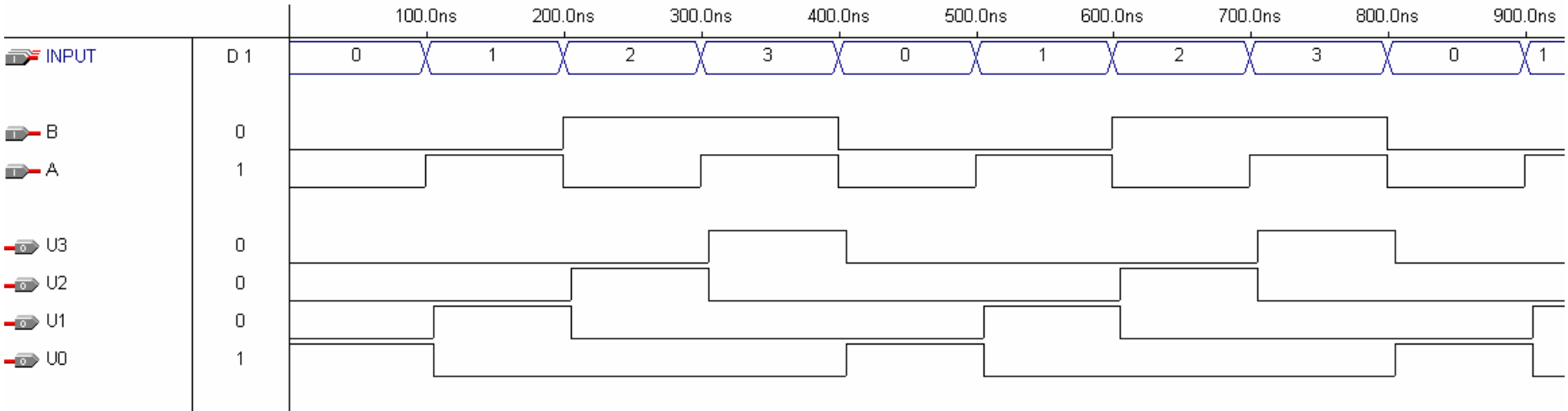
Nella rappresentazione strutturale (‘***Structural***’) un componente e’ descritto connettendo tra loro più blocchi. L’approccio e’ basato su un linguaggio testuale (VHDL) ma risulta concettualmente analogo al design-entry mediante schema logico.



Esempio: Decoder

Effettuando il *design entry* mediante **schema logico** ed effettuando la simulazione si perviene alle seguenti forme d'onda.



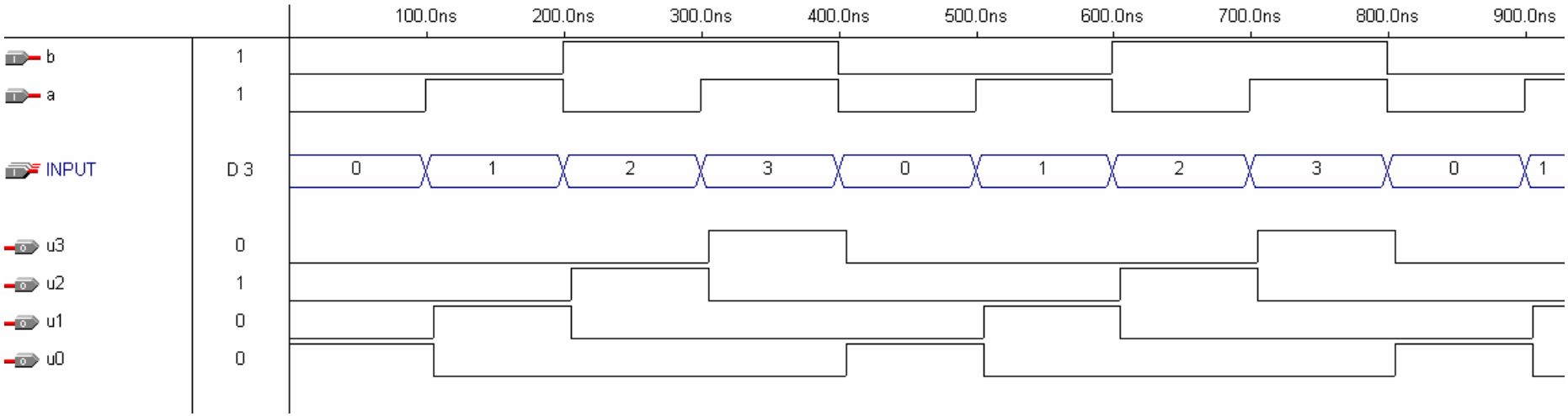


Una possibile codifica in VHDL della rete combinatoria dell'esempio potrebbe essere la seguente:

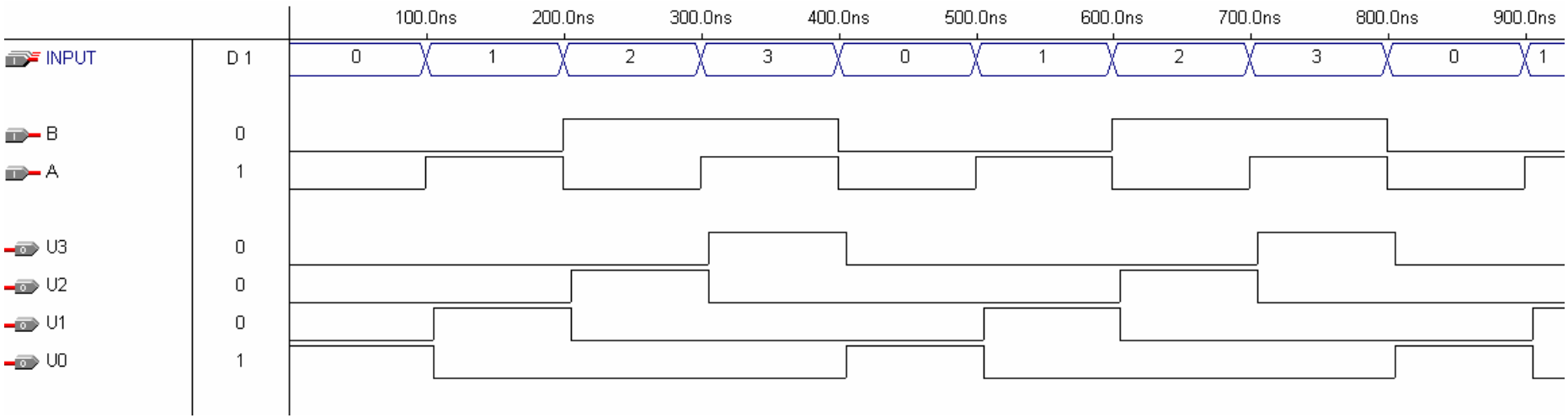
```
-- Possibile codifica VHDL del decoder

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY decoder_vhdl  IS
   PORT (a, b             : IN  BIT;
              u0,u1,u2,u3 : OUT BIT);
END decoder_vhdl;

ARCHITECTURE arch_decoder_vhdl OF decoder_vhdl IS

BEGIN

                -- attivazione (concorrente) dei segnali
                -- non e' importante l'ordine

                u3 <=  (a AND b) ;
                u2 <=  ((NOT a) AND b);
                u1 <=  (a AND (NOT b));
                u0  <=  ((NOT a) AND (NOT b));

END arch_ decoder_vhdl;
```

Come risulta dal confronto tra le due simulazioni, ottenute rispettivamente dal design entry mediante schema logico e mediante VHDL, i due approcci sono equivalenti,

Simulazione: codice VHDL



Simulazione: schema logico



Elementi che caratterizzano il linguaggio VHDL

Il VHDL essendo un linguaggio di programmazione per la descrizione dell'HARDWARE presenta delle sostanziali differenze rispetto ai linguaggi di programmazione standard (i.e. C, Java, Pascal).

In particolare vi sono due caratteristiche che differenziano il VHDL rispetto ai linguaggi di programmazione convenzionali:

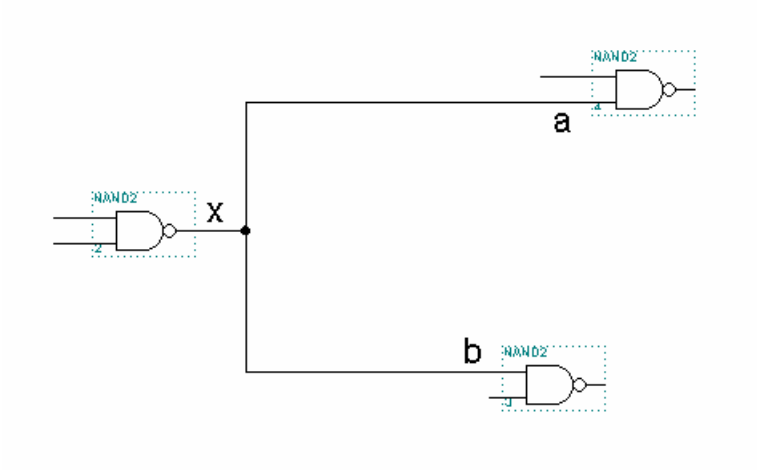
Timing → **La capacità di poter gestire i tempi di propagazione dei segnali all'interno dei circuiti digitali**

Concurrency → **La capacità di simulare lo svolgimento di più operazioni contemporanee, tipica dei dispositivi Hardware**

‘Timing’ & ‘Concurrency’

La propagazione dei segnali tra i vari componenti che compongono un un circuito digitale avviene attraverso fili o bus.

Nella realtà la propagazione dei segnali non avviene istantaneamente perché ritardata dalle caratteristiche fisiche delle connessioni (fenomeni parassiti). Si consideri ad esempio la seguente rete logica



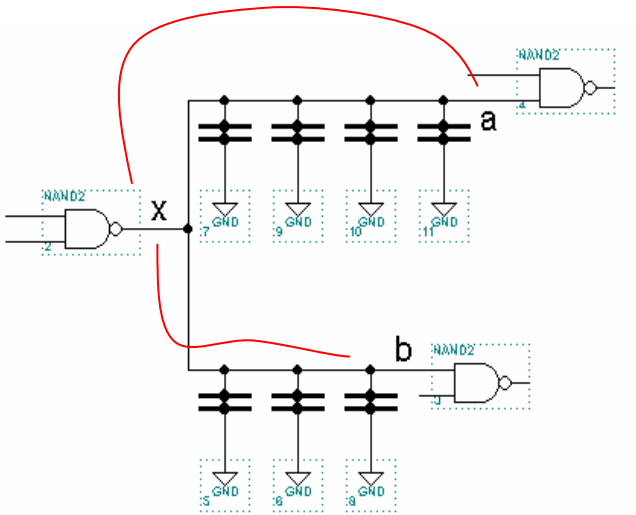
Rete ideale

Volendo descrivere come si propaga il segnale dal punto **x** verso **a** e **b** con un linguaggio di programmazione ad alto livello (ad esempio il ‘C’) si potrebbe scrivere:

```
a:=x;          /* assegna x ad a */
b:=x;          /* assegna x a b */
```

Prima osservazione (PROBLEMA DI TIMING):

La rappresentazione attraverso le due assegnazioni del lucido precedente non contempla i ritardi di propagazione del segnale che in realtà ci sono. La rete seguente evidenzia le capacità parassite associate ai due tratti della connessione tra il gate di ingresso e i due gate in uscita (lungo percorsi da **x** a **a** da **x** a **b)** Queste capacità determinano il ritardo di propagazione del segnale.



Rete reale con capacità parassite che impongono la variazione dei segnali **a** e **b** in ritardo rispetto alla variazione di **x**

Seconda osservazione (PROBLEMA DI CONCURRENCY):

Solitamente i linguaggi di programmazione prevedono che due istruzioni di assegnazione vengano eseguite nella sequenza in cui compaiono nel programma.Nella realtà il segnale elettrico parte da x e si propaga contemporaneamente verso a e b (non prima su a e poi su b come avverrebbe con il codice della pagina precedente scritto in ‘C’).

Il VHDL, essendo orientato all'hardware deve essere in grado di descrivere sia il **TIMING** sia la **CONCURRENCY** dei segnali:

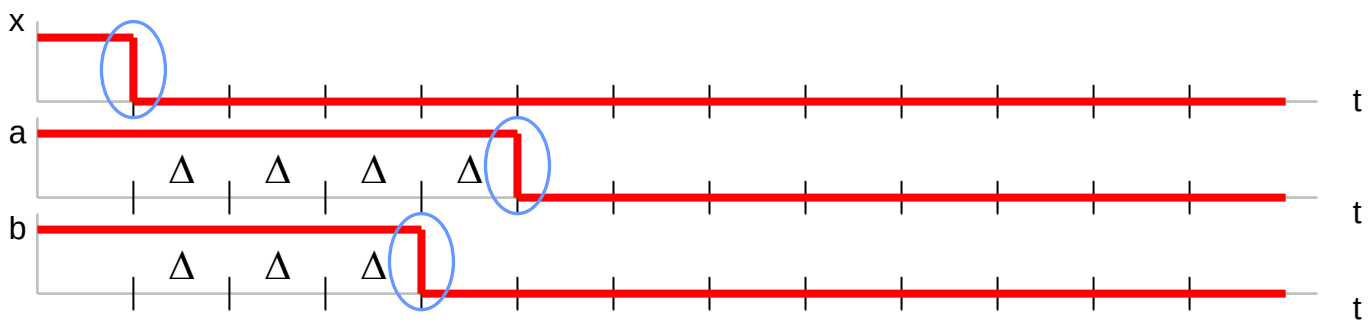
(TIMING) rendendo possibile la gestione dei ritardi con cui si propagano i segnali digitali all'interno della rete

(CONCURRENCY) Eseguendo più istruzioni (processi) contemporaneamente

Considerando la rete dell'esempio precedente. Se ogni capacità introduce un ritardo pari a 1 unit_delay (Δ) il codice VHDL che sintetizza il funzionamento reale della rete potrebbe essere il seguente:

```
a<=x AFTER 4*unit_delay
b<=x AFTER 3*unit_delay
```

Da un punto di vista logico le 2 istruzioni sono eseguite contemporaneamente in VHDL (esecuzione concorrente)

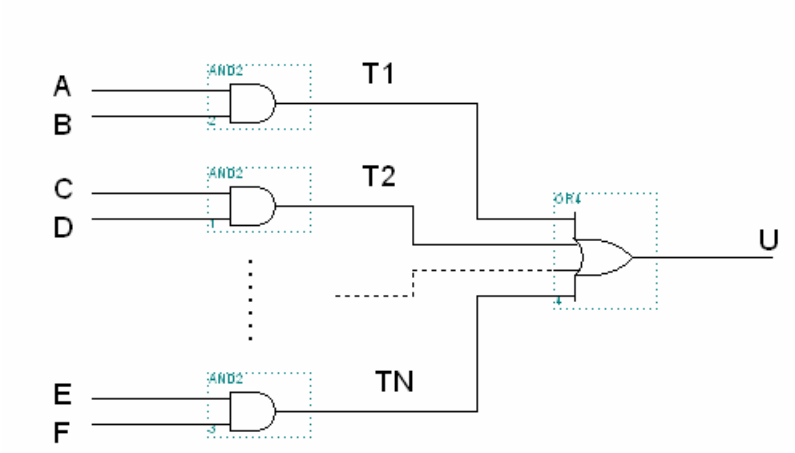


CONCURRENCY

Quando un **programmatore** sviluppa del codice con un linguaggio ad alto livello (i.e. C, Java, Pascal) immagina di scomporre il problema in una serie di istruzioni che saranno eseguite in modo sequenziale. **PARADIGMA DI PROGRAMMAZIONE SEQUENZIALE**.

Al contrario, un **progettista hardware** decompone il progetto in blocchi interconnessi che reagiscono ad eventi e generano eventi. Gli eventi sono le transizioni dei segnali: ogni transizione di un ingresso, di un'uscita o del clock è un evento. Ogni evento impone che tutti i blocchi "dipendenti" da tale evento vengano valutati. L'ordine in cui vengono valutati i blocchi deve essere ininfluente sul risultato finale(cioe', qualunque sia l'ordine di valutazione delle espressioni associate ai singoli blocchi, lo stato complessivo raggiunto dalla rete quando tutti gli eventi sono stati gestiti deve essere sempre il medesimo). Il tipo di programmazione che consente di modellare questo tipo di funzionamento e' il **PARADIGMA DI PROGRAMMAZIONE PARALLELA**: infatti visto che il risultato dell'elaborazione deve essere indipendente dalla sequenza in cui le istruzioni sono state eseguite, allora possiamo concludere che tutte le istruzioni possono anche essere eseguite in parallelo,senza che mai una istruzione, per essere eseguita, debba attendere il completamento di un'altra.

Ad esempio, si pensi ad una rete combinatoria di tipo SP.
Il programmma che descrive questa rete deve generare un risultato (uscita di ciascun gate) che dipende solo dal valore degli ingressi e non dall'ordine con il quale vengono valutati gli and e l'or.
Le istruzioni che descrivono i blocchi possono essere codificate secondo il paradigma della programmazione parallela




```
T1 <=  A AND B;
T2 <=  C AND D;
TN <=  E AND F;
U <= T1 OR T2 OR T3;
```

Equivalenti

```
U <= T1 OR T2 OR T3;
T1 <=  A AND B;
T2 <=  C AND D;
TN <=  E AND F;
```

Oggetti definiti in VHDL

Oggetto VHDL: entità alla quale e' associato un valore

Sono definite 4 classi di oggetti:

Segnali → Rappresentano connessioni hardware (input, output, etc). Ad ogni segnale è associata una evoluzione temporale. Le assegnazioni sono effettuate mediante l'operatore **<=**.

Variabili → Non hanno alcun significato hardware. Sono utilizzate per memorizzare valori temporanei. Le variabili possono essere dichiarate, o avere valori assegnati, solo all'interno di blocchi sequenziali del codice VHDL. Il simbolo di assegnamento è **:=**

Costanti → Rappresentano valori costanti di un determinato tipo. Il loro valore non può essere cambiato. Attraverso lo statement GENERIC il VHDL dispone di un meccanismo per passare parametri non hardware (es ritardi) di un componente.

Files → Oggetti che possono essere dichiarati ed utilizzati sia in blocchi concorrenti che in blocchi sequenziali.

Data types

Nel linguaggio VHDL (Standard Package) sono predefiniti i seguenti tipi di dato:

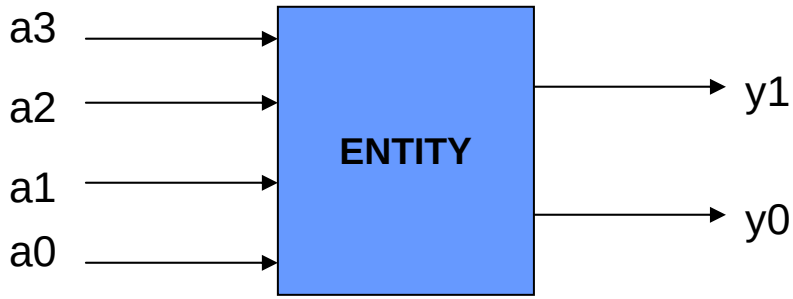
- **BIT ('0','1')**
- **BIT_VECTOR**
- **BOOLEAN ('TRUE','FALSE')**
- **INTEGER**
- **REAL**
- **TIME**

Oltre ai tipi predefiniti e' possibile utilizzarne altri disponendo di specifiche librerie (es LIBRARY IEEE).

E possibile definire dei vettori di bit utilizzando il tipo di dato BIT_VECTOR...

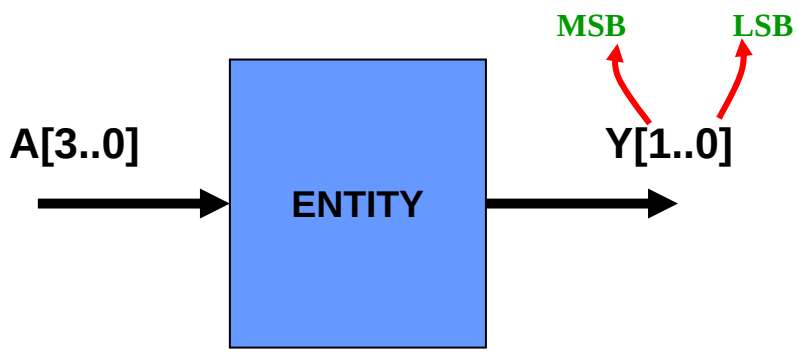
Formalismo scalare

```
PORT ( a3,a2,a1,a0 : IN  BIT;
       y1,y0 : OUT BIT);
```



Formalismo vettoriale (‘bus’)

```
PORT ( a : IN  BIT_VECTOR(3 DOWNTO 0);
       y : OUT BIT_VECTOR(1 DOWNTO 0));
```



In VHDL gli elementi dei vettori possono essere referenziati attraverso gli indici.
Ad esempio: **y(2) <= a(1)**

E’ inoltre possibile assegnare una sequenza di cifre.
Ad esempio **y <= “01”**;

Per costanti binarie di un singolo bit si utilizza il simbolo ‘ mentre per configurazioni binarie composte da 2 o più bit si utilizza il simbolo “.

Ad esempio:

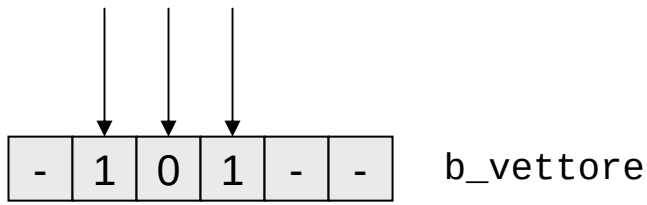
a <= ‘1’;

b_vettore <= “100111”;

E’ anche possibile effettuare assegnazioni ad un numero limitato di bit del vettore (*bit-slicing*).

Ad esempio:

b_vettore(4 downto 2) <= “101”



LIBRARY IEEE

Mediante l'utilizzo di librerie (keyword LIBRARY) e' possibile aggiungere al VHDL standard nuovi tipi di dato, nuove funzioni, etc. Il meccanismo e' analogo a quello utilizzato dai linguaggi di programmazione ad alto livello.

LIBRARY IEEE

Tale libreria contiene i seguenti packages:

- **std_logic_1164** (definisce la std_logic e relative funzioni)
- **std_logic_arith** (funzioni aritmetiche)
- **std_logic_signed** (funzioni aritmetiche su numeri con segno
- **std_logic_unsigned** (funzioni aritmetiche su numeri senza segno)

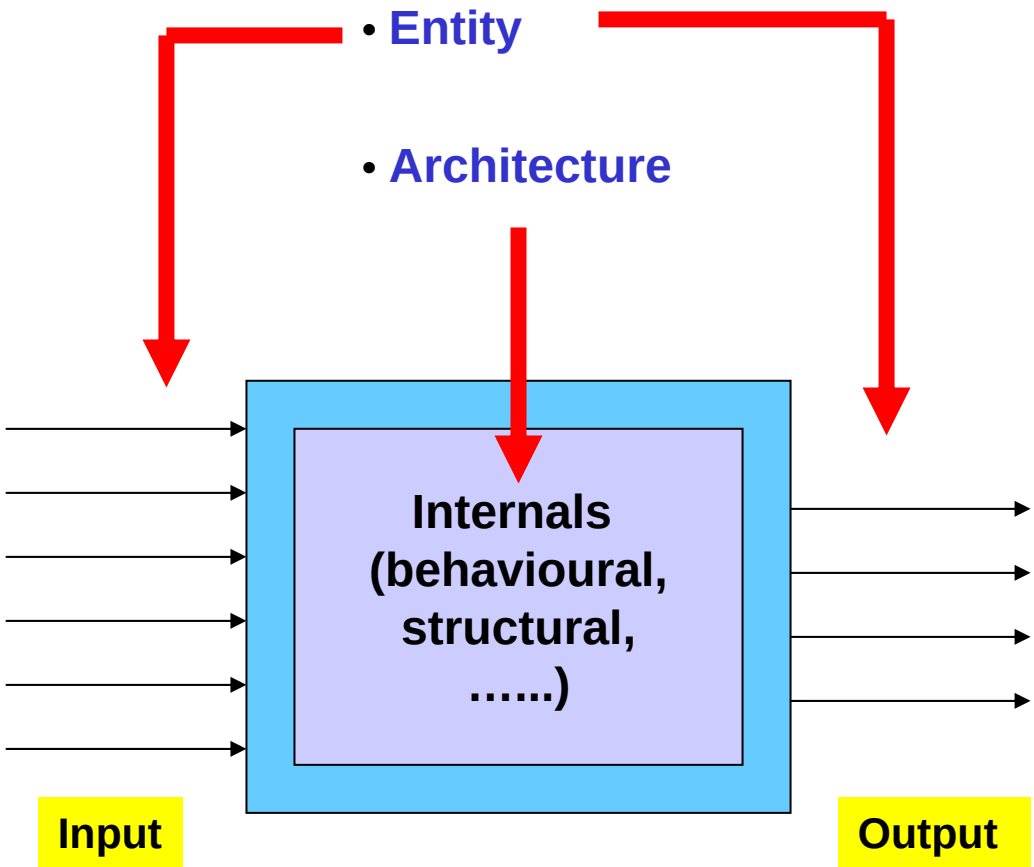
VHDL & Altera Max+ Plus II

Altera Max+ Plus II fornisce un compilatore per codice vhdl. Esistono alcune regole che e' necessario tenere in considerazione utilizzando l'ambiente Altera.

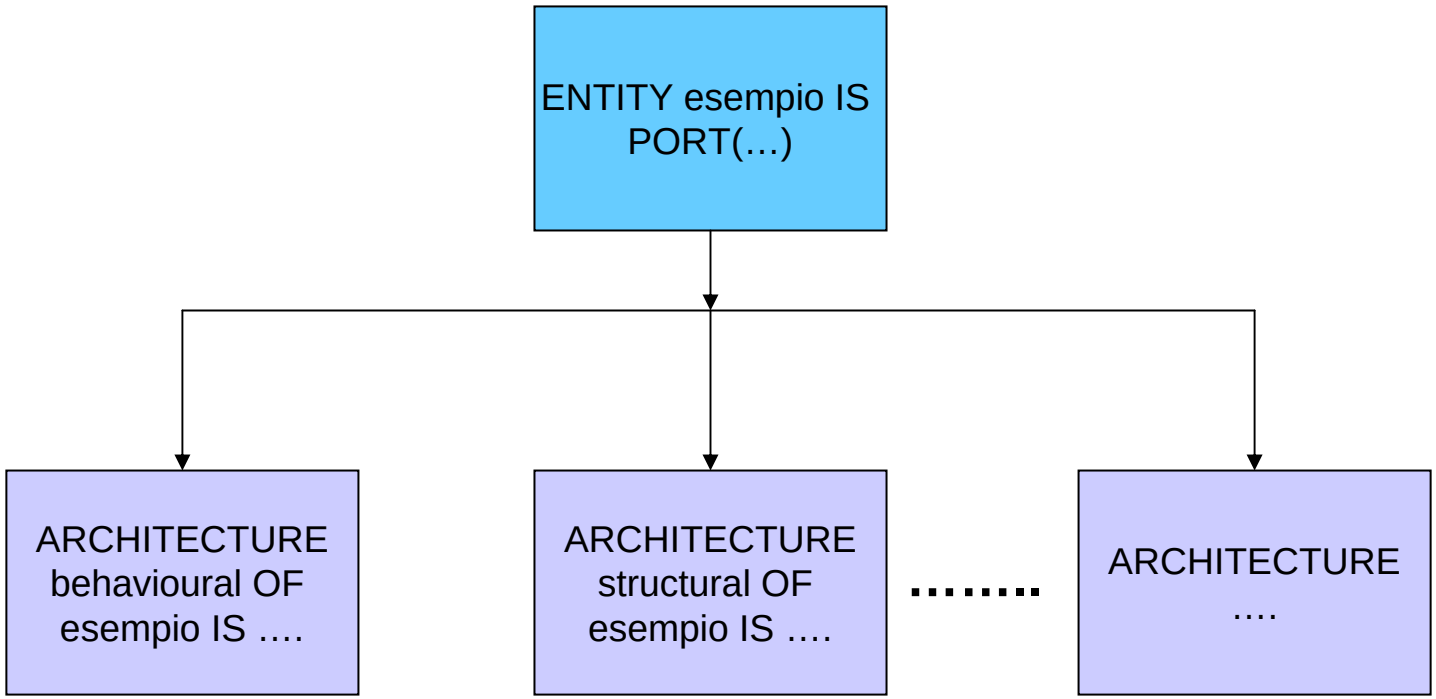
E' disponibile un editor di testo per codice VHDL all'interno dell'ambiente software ma e' possibile utilizzare anche editor esterni.

- Il linguaggio VHDL **NON E'** *case sensitive*. Per convenzione utilizzeremo le lettere maiuscole per le parole chiave del linguaggio
- I file di testo che contengono il codice VHDL **DEVONO** avere estensione .vhd
- Il nome del file VHDL (estensione .vhd) **DEVE** coincidere con il nome assegnato all' **ENTITY**.

Concetti fondamentali per la descrizione di circuiti digitali con il linguaggio VHDL



Data una *interface description* (**ENTITY**) sono possibili diverse architectural specification (**ARCHITECTURE**)



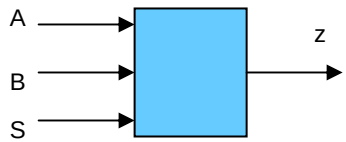
```
-- Sintassi di un blocco descritto mediante VHDL
```

```
LIBRRY IEEE;                    -- Standard IEEE library
USE IEEE.std_logic_1164.all; -- Standard logic package
```

Inclusione di librerie

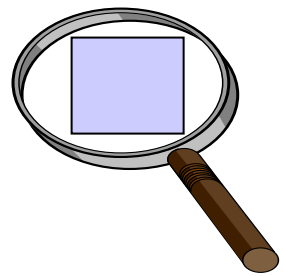
```
-- Entity declaration

ENTITY nome_entity  IS

          PORT(signal_name : direction type);
          GENERIC(generic_name: type := default_value);

END nome_entity;
```

Interface specification




```
-- Architecture description

ARCHITECTURE architettura_entity OF nome_entity IS

<architecture declarations>

BEGIN

<processing statements>

END architettura_entity ;
```

Architectural specification



Entity

```
-- Entity declaration

ENTITY nome_entity  IS

          PORT(signal_name: direction type);
          GENERIC(generic_name: type := default_value);


END nome_entity;
```

Mediante il costrutto **GENERIC** si definiscono tipicamente delle costanti. Queste possono essere utilizzate ad esempio per definire dei tempi di ritardo,

```
GENERIC(delay: time := 1 ns);
```

Mediante il costrutto **PORT** si specificano:

- quali sono i segnali della rete esaminata (*signal_names*)
- (**direction**) quali segnali sono di input (IN), di output (OUT), bidirezionali (INOUT)
- (**type**) di che tipo sono i segnali

Architecture

```
-- Architecture description

ARCHITECTURE architettura_entity OF nome_entity IS

<architecture declarations>

BEGIN

<processing statements>

END architettura_entity ;
```

All'interno della sezione **ARCHITECTURE** viene specificata la logica della rete.

All'interno della sezione **ARCHITECTURE** (in ***<architecture declarations>***) è possibile definire degli oggetti. Tali oggetti sono tipicamente dei segnali (vedi esempio 2) e possono essere utilizzati (*scope*) solo all'interno della *architecture description*. E' possibile utilizzare i ***data types*** definiti all'interno del VHDL o disponibili mediante l'utilizzo di librerie aggiuntive. Ad esempio:

```
SIGNAL T1,T0 : BIT;
```

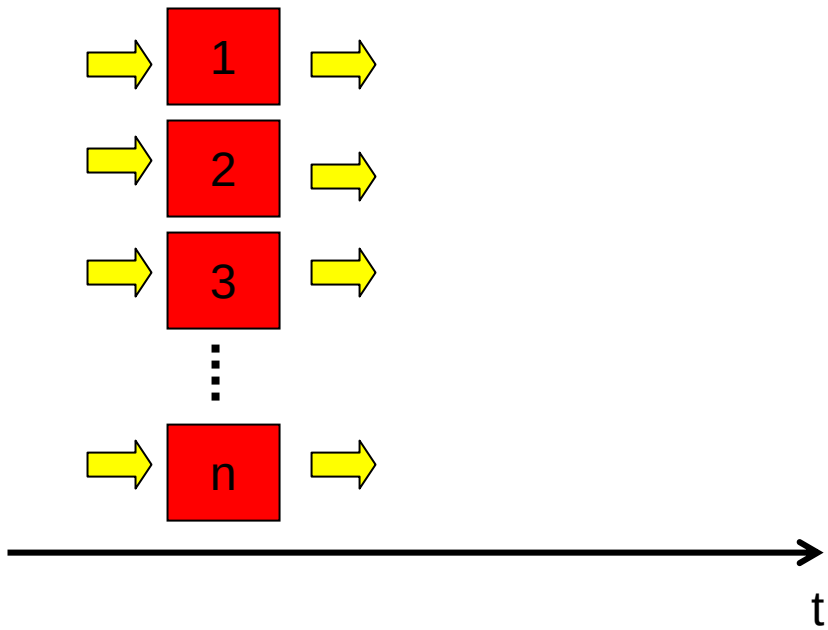
La parte nella quale viene specificata la logica vera e propria della rete è il ***<processing statements>***, compresa tra **BEGIN** ed **END**.

Per il <*processing statements*> esistono due distinti modelli di elaborazione…..

CONCURRENT PROCESSING

Le espressioni comprese tra **BEGIN** ed **END** vengono elaborate tutte contemporaneamente. Non ha alcun significato l'ordine con il quale appaiono nel codice VHDL.

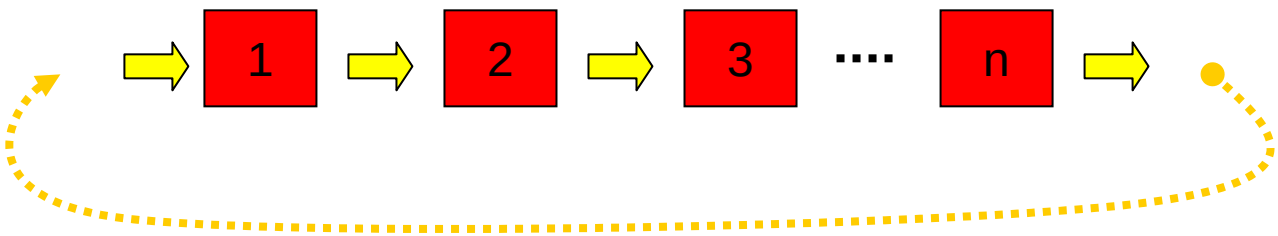
```
ARCHITECTURE …

BEGIN

<concurrent statement 1>
<concurrent statement 2>
<concurrent statement 3>
          .
        .
        .
<concurrent statement n>


END;
```



SEQUENTIAL PROCESSING

Mediante l'istruzione **PROCESS** e' possibile definire una sequenza di istruzioni che verranno elaborate secondo il paradigma di programmazione sequenziale. All'interno di un blocco **PROCESS** le istruzioni sono elaborate in modo sequenziale dall'alto verso il basso (nell'ordine in cui vengono scritte). Ogni processo e' eseguito in modo concorrente rispetto ad altri processi o istruzioni concorrenti definite nell'architecture.

```
ARCHITECTURE …

BEGIN

        PROCESS(sensitivity_list)

        <sequential statement 1>
        <sequential statement 2>
                .
                .
                .
        <sequential statement n>


        END PROCESS;
END
```



sensitivity_list

Lista degli segnali (eventi) ai quali le istruzioni del processo sono sensibili. Rappresentano quindi gli eventi che possono modificare le espressioni definite all’interno di un processo. Se nessuno di questi eventi si verifica il processo rimane inattivo.

Operatori logici definiti in VHDL

And:	**AND**
Or:	**OR**
Not:	**NOT**
Nand:	**NAND**
Nor:	**NOR**
Xor:	**XOR**
Xnor:	**XNOR**

Esempi:

```
Y<= NOT A;
Z<= a AND b;
```

Tali operatori agiscono su tipi predefiniti: BIT, BOOLEAN, BIT_VECTOR.

Se sono usati vettori il numero di bit dei due operandi deve essere lo stesso.

Operatori relazionali definiti in VHDL

Tali operatori agiscono su operandi dello stesso tipo e ritornano un valore BOOLEAN (*TRUE* o *FALSE*).

Uguale:	**=**
Diverso:	**/=**
Minore:	**<**
Minore Uguale:	**<=**
Maggiore:	**>**
Maggiore Uguale:	**>=**

Esempi:

```
a_boolean <= op1 > op2;
b_boolean <= op1 /= op2;
```

Operatori come +, -, <, >, <= ,>= sono definiti solo per il tipo INTEGER.

Altri operatori definiti in VHDL

In VHDL sono definiti altri operatori, quali:

- **SHIFT**
- **SOMMA**
- **SOTTRAZIONE**
- **MOLTIPLICAZIONE**

Consultare il manuale del VHDL per una descrizione dettagliata di tali operatori.

CONCURRENT STATEMENTS

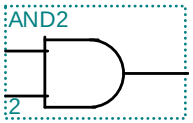
C*oncurrent statements* possono essere utilizzati solo al di fuori di un processo e da un punto di vista concettuale sono eseguiti in modo concorrente. Per questo motivo l'ordine con il quale vengono scritti non e' importante.

- Concurrent signal assignements
- Selective signal assignements (**WITH-SELECT-WHEN**)
- Conditional signal assignements (**WHEN-ELSE**)

WITH-SELECT-WHEN
Selected signal assignement

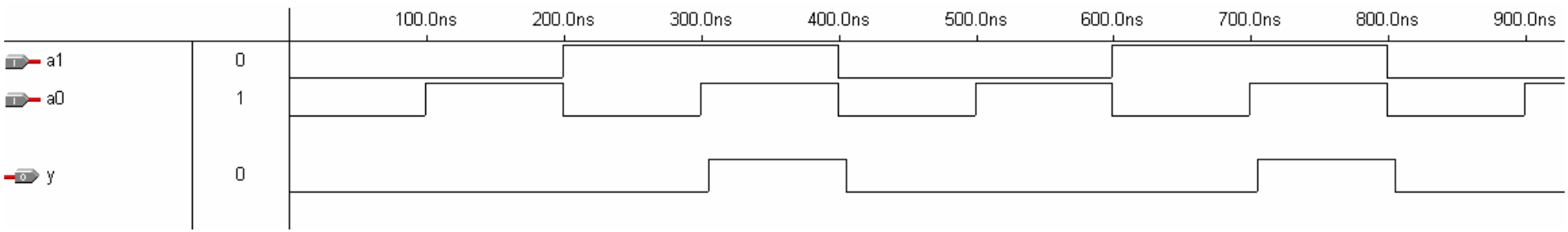
```
WITH <expression> SELECT
<signal_name>  <= <signal/value> WHEN <condition1>,
                  <signal/value> WHEN <condition2>,
                                 .
                                 .
                                 .
                  <signal/value> WHEN OTHERS;
```

Esempio (and2)



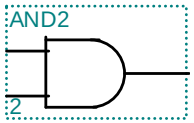

```
ENTITY and2with IS
   PORT (a : IN  BIT_VECTOR (1 DOWNTO 0);
         y : OUT BIT);
END and2with;

ARCHITECTURE arch_and2with OF and2with IS

BEGIN

WITH a SELECT
     y <= '1' WHEN "11",
          '0' WHEN "00",
          '0' WHEN "01",
          '0' WHEN "10";

END arch_and2with;
```

and2with.vhd



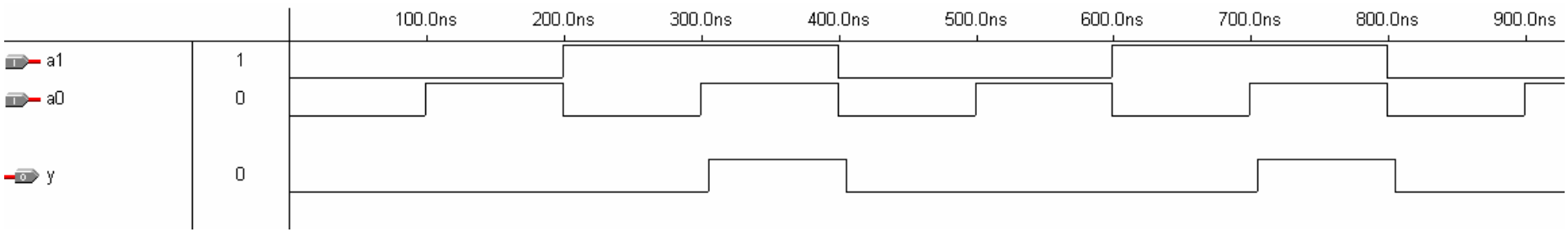
Una realizzazione più semplice potrebbe essere la seguente

Esempio (and2)




```
ENTITY and2withothers IS
   PORT (a : IN  BIT_VECTOR (1 DOWNTO 0);
         y : OUT BIT);
END and2withothers;

ARCHITECTURE arch_and2withothers OF and2withothers IS

BEGIN

WITH a SELECT
     y <= '1' WHEN "11",
          '0' WHEN OTHERS;

END arch_and2withothers;
```

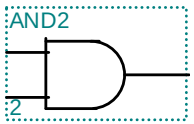
and2withothers.vhd



WHEN-ELSE
Conditional signal assignement

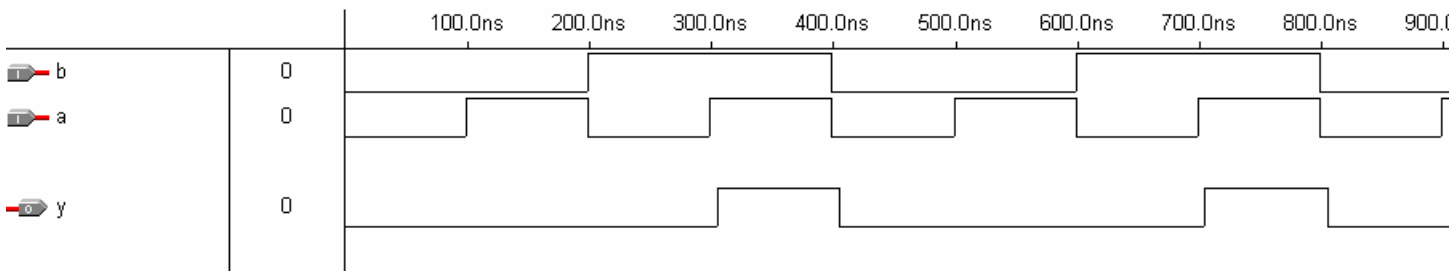
```
<signal_name> <= <signal/value> WHEN <condition1> ELSE
                 <signal/value> WHEN <condition2> ELSE
                                .
                                .
                                .
                 <signal/value> WHEN <conditionN> ELSE
                 <signal/value>;
```

Esempio (and2)




```
ENTITY and2 IS
   PORT (a, b : IN  BIT;
         y    : OUT BIT);
END and2;

ARCHITECTURE arch_and2 OF and2 IS

BEGIN

        y<='1' WHEN (A='1' AND B='1') ELSE
           '0';

END arch_and2;
```

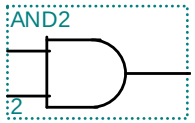
and2.vhd



Nel caso più condizioni siano verificate al segnale di uscita e' assegnato il primo valore che soddisfa la catena dei WHEN.

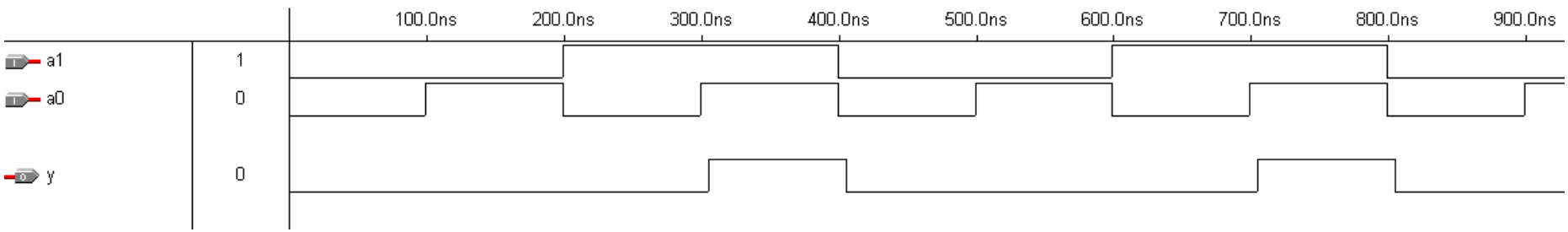
Codice VHDL che sintetizza l'AND a 2 ingressi utilizzando il costrutto WHEN-ELSE

Esempio (and2)



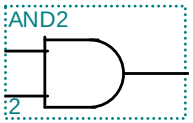

```
ENTITY and2whenelse1 IS
   PORT (a: IN  BIT_VECTOR (1 DOWNTO 0);
         y    : OUT BIT);
END and2whenelse1;

ARCHITECTURE arch_and2whenelse1 OF and2whenelse1 IS

BEGIN

     y <= '0' WHEN a(0)='0' ELSE
          '0' WHEN a(1)='0' ELSE
           '1';

END arch_and2whenelse1;
```

and2whenelse1.vhd



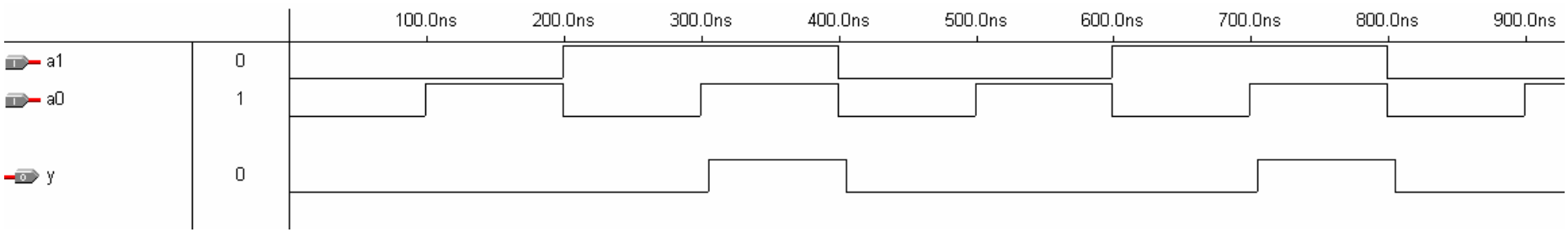
Un’altra soluzione, sempre utilizzando WHEN-ELSE, potrebbe essere la seguente..

Esempio (and2)



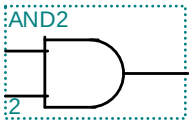

```
ENTITY and2whenelse2 IS
   PORT (a: IN  BIT_VECTOR (1 DOWNTO 0);
         y    : OUT BIT);
END and2whenelse2;

ARCHITECTURE arch_and2whenelse2 OF and2whenelse2 IS

BEGIN

     y <= ‘0' WHEN a(0)=‘0’ ELSE
          ‘1' WHEN a(1)=‘1’ ELSE
           ‘0’;

END arch_and2whenelse2;
```

and2whenelse2.vhd



Oppure...

Esempio (and2)




```
ENTITY and2whenelse3 IS
   PORT (a : IN  BIT_VECTOR (1 DOWNTO 0);
         y : OUT BIT);
END and2whenelse3;

ARCHITECTURE arch_and2whenelse3 OF and2whenelse3 IS

BEGIN

y <= '1' WHEN a="11" ELSE
          '0';

END arch_and2whenelse3;
```

and2whenelse3.vhd