Descrizione Strutturale

- In questo capitolo vedremo come la struttura di un sistema digitale è descritto *strutturalmente* in VHDL.
- **Outline:**
  - Entity & Architecture
  - Binding di Architectures and Entities
  - Istanziazione di componenti
  - Wiring dei blocchi;
  - Assegnazione concorrente;
  - I costrutti iterativi;
  - Design parametrization: i generics;
  - Generazione di testbench;
  - Esempi.

Convenzioni Tipografiche

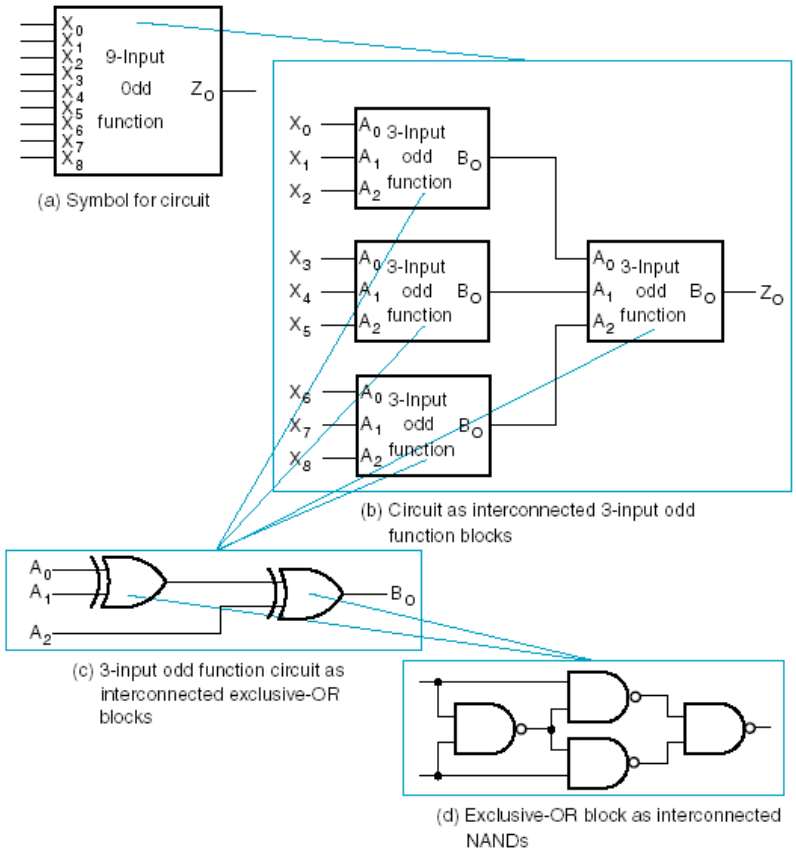
- In tutte le slide si useranno le seguenti convenzioni tipografiche:
- **`ENTITY`**, **`ARCHITECTURE`**, **`FOR`**, etc. sono comandi del linguaggio VHDL e sono parole riservate, ovvero non si possono usare come nomi assegnati dall'utente ad oggetti.
- **`Nome_oggetto`** è un oggetto del VHDL a cui l'utente ha assegna il nome nome_oggetto.
- *Entity*, *architecture*, etc. fa riferimento al concetto di entity in VHDL e non al comando entity che permette di instanziare una enity.
- Il VHDL non è case sensitive.
- Per introdurre dei commenti si usano due segni '-':

```
-- commento
```

Esempio di descrizione strutturale e gerarchica (1)

- Ad esempio consideriamo la funzione disparità (XOR) a 9 ingressi:

$$Z_0 = x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_8$$
$$= (x_0 \oplus x_1 \oplus x_2) \oplus (x_3 \oplus x_4 \oplus x_5) \oplus (x_6 \oplus x_7 \oplus x_8)$$



- La funzione disparità a 3 ingressi, si può esprimere in termini di XOR a 2 ingressi (XOR-2):

$$B_0 = x_0 \oplus x_1 \oplus x_2 = (x_0 \oplus x_1) \oplus x_2)$$

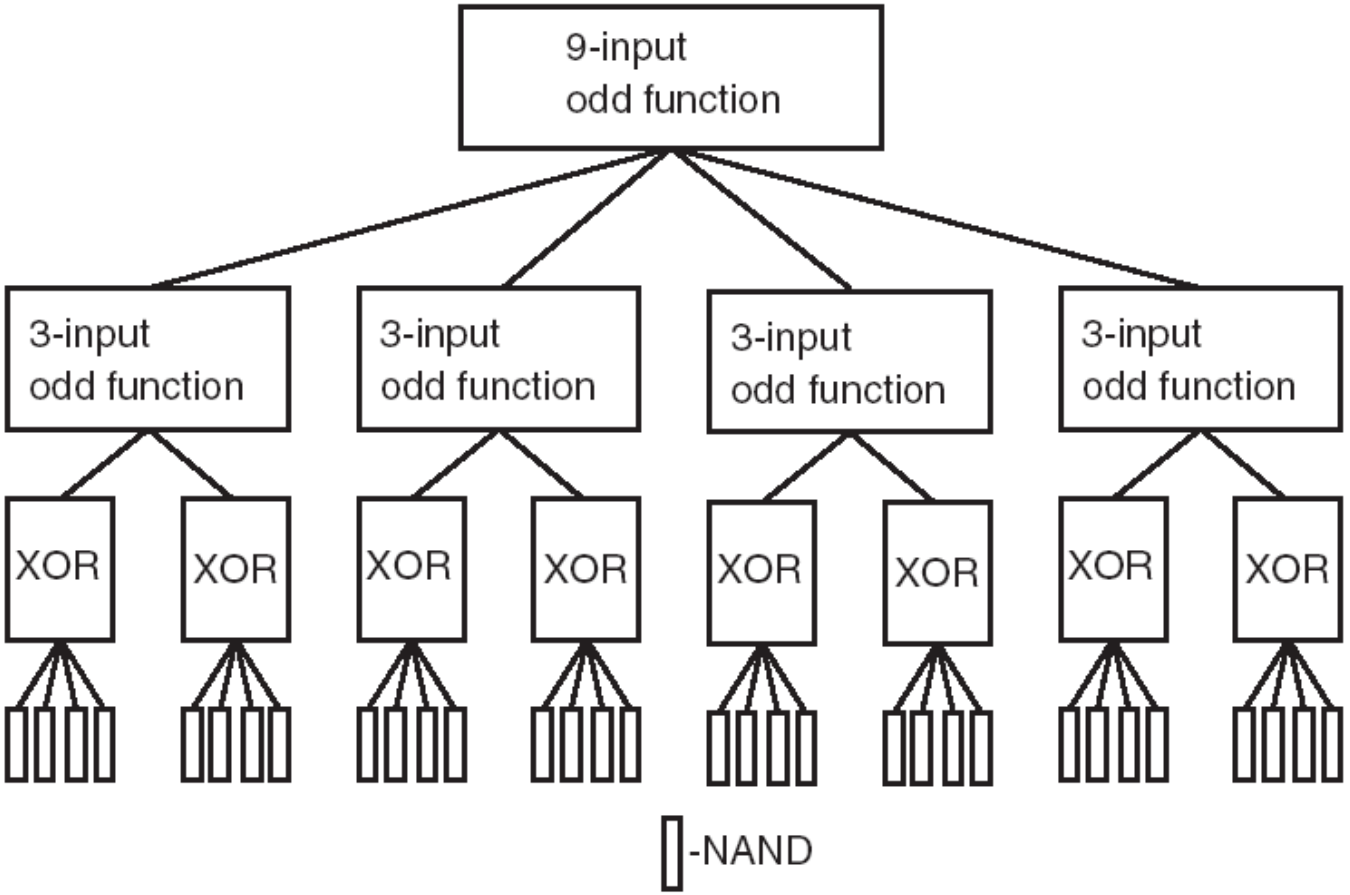
- Infine la XOR a 2 ingressi ha come espressione booleana:

$$a \oplus b = a \text{ and not}(b) + \text{not}(a) \text{ and } b$$

- Si noti che la XOR a tre ingressi (XOR-3) può anche essere espressa in termini di NAND a 2 ingressi (NAND-2), come in figura.

Esempio di descrizione strutturale e gerarchica (2)

- In pratica lo schema completo che rappresenta la scomposizione della funzione XOR-9 in termini di porte NAND è la seguente:



Entity/Architecture

- Un **sistema digitale** è tipicamente progettato come se fosse composto da un insieme di moduli organizzati gerarchicamente in un grafo aciclico, ovvero un albero:
  - c'è un componente più esterno top-level module (la radice dell'albero) che istanzia e collega altri moduli, che a loro volta istanziano altri componenti, … finché i componenti usati non corrispondono a moduli "elementari" (atomici);
- Ciascun modulo (*design entity*, *design module*) ha un insiemi di terminali di I/O (*ports*) che compongono la sua interfaccia con il mondo;
- In VHDL un modulo può essere usato come un componente o può essere usato come top level module del design.

- Ogni modulo in VHDL è descritto da 2 parti:
  - una **entity**: descrive l'interfaccia del modulo (i segnali di ingresso/uscita, oltre ad altri parametri, ovvero i *generics*);
  - una **architecture**: descrive il funzionamento del sistema ovvero il suo legame fra i pin di ingresso e uscita.

Entity (1)

- Una dichiarazione di **entity** può essere usata per dichiarare gli "oggetti" (segnali di I/O, generics) che potranno essere usati nell'implementazione dell'entity;
- Una dichiarazione di entity:
  - include la specifica del componente in termini dei suoi porti (*ports*), specificando il loro nome, il loro tipo e la loro direzione (*mode*);
  - può includere la specificazione di costanti che possono essere usate per controllare la struttura e il comportamento dell'entity (tramite i *generics*).

```
ENTITY component_name
    dichiarazione del nome, del tipo e del modo di tutti i porti
    dichiarazione (opzionale) del nome, del tipo e del valore dei generics
END component_name;
```

Entity (2)

```
ENTITY nome_entita IS
PORT (
    nome_segnale_1 : modo_segnale_1 tipo_segnale_1;
    nome_segnale_2 : modo_segnale_2 tipo_segnale_2;

    nome_segnale_N : modo_segnale_N tipo_segnale_N;
    nome_generic_1 : tipo_generic_1 [:= valore_generic_1];

    nome_generic_M : tipo_generic_M [:= valore_generic_M] );
END nome_entita;
```

- Il nome dell'entity dichiarata è **nome_entità**;
- Nella dichiarazione di entità riportata, per ogni dichiarazione di port 1…N:
  - **nome_segnale_*i*** rappresenta il nome assegnato all'i-esimo segnale;
  - **modo_*i*** è il modo di interazione (IN, OUT, INOUT, BUFFER) del segnale nome_segnale*i* con l'esterno;
  - **tipo_segnale_*i*** specifica il tipo del segnale ovvero l'insieme dei valori che il segnale può assumere in ogni istante (è analogo al tipo di una variabile sw).
- Analogamente per i generics (**nome_generic_*j***, **tipo_generic_*j***, **valore_generic_j**) con j che va da 1 a M;

Entity (3)

- La dichiarazione di una entity è la parte *formale*, ovvero specifica i tipi e i modi dei ports (e dei generics); i segnali che *concretamente* saranno collegati (e il valore attuale dei generics), viene specificato all'atto dell'istanziazione dell'entity;
- Quando il modulo sarà istanziato, verrà stabilita la corrispondenza tra ports e segnali, e quindi i segnali sono i parametri *effettivi*;
- All'interno all'architecture si usano i nomi dei ports per far riferimento ai segnali che verranno collegati;

- Significato hardware: l'operazione di istanziazione e specificazione dei collegamenti di un modulo VHDL corrisponde ad usare un componente (ad es. un integrato della serie 74) e a collegare i suoi piedini con dei fili (segnali) e tramite questi collegare diversi componenti assieme;

I ports (1)

- Ogni dichiarazione di *port* è caratterizzato da un nome, un tipo e da un modo;

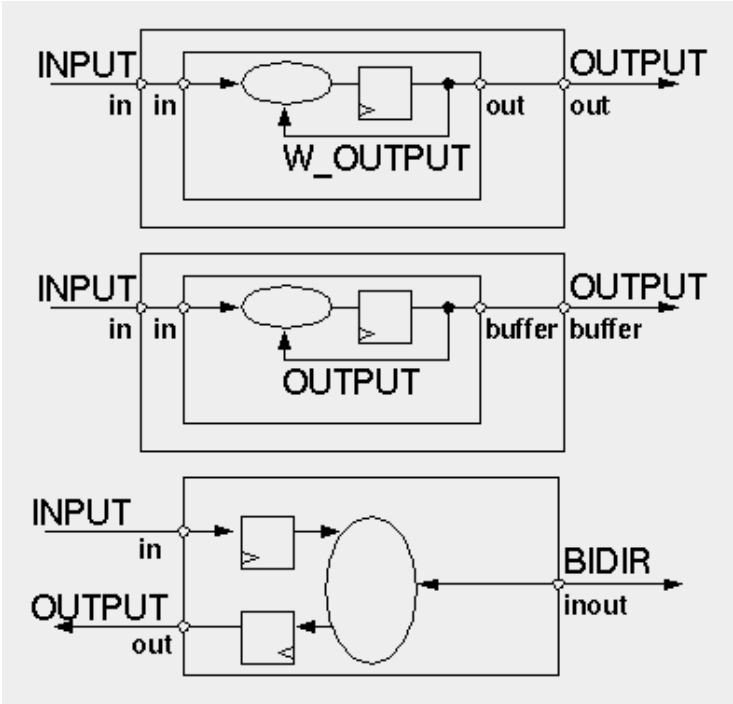
```
ENTITY nome_entita IS PORT (
     nome_segnale1 : modo1 tipo_segnale1;
```

- Il tipo specifica l'insieme di valori che qualunque segnale connesso al port considerato può assumere in ogni istante;
  - Esempio: se il port nome_port è di tipo *std_logic*, ovvero di un tipo che rappresenta a livello logico delle condizioni elettriche (il valore '0' → potenziale di massa, '1' → Vdd, 'X' valore nella regione ambigua, etc.), tutti i segnali che potranno essere connessi al port nome_port dovranno essere segnali di tipo std_logic.
- In generale gli oggetti che possono essere collegati ad un port sono solo segnali (*signals*) e devono essere di un tipo "compatibile" con il tipo del port a cui sono collegati (il VHDL è strong-typed);

I ports (2)

- Il modo di un port fissa la direzione in cui i dati possono essere trasferiti fra un modulo ed il mondo esterno attraverso i terminali di ingresso e di uscita.
- I modi previsti sono 4:
  - *IN* - Il port è solo un ingresso per l'entità. L'entity che pilota il terminale (il circuito driver) è esterno all'entità considerata;
  - *OUT* - Il port è una uscita per l'entità e quindi il driver è interno all'entità. Il valore di un pin OUT non può essere "letto" all'interno dell'entità;
  - *BUFFER* - Il port è di uscita ma il suo valore può anche essere "letto" all'interno dell'entità;
  - *INOUT* - Il segnale può essere utilizzato sia come ingresso che come uscita (ad es. un bus dati bidirezionale).

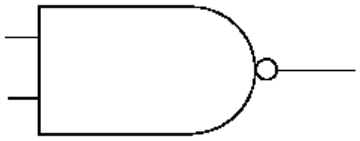
```
ENTITY nome_entita IS PORT (
     A : IN bit;
     B : IN bit_vector(0 to 3);

     Z : OUT bit );
```



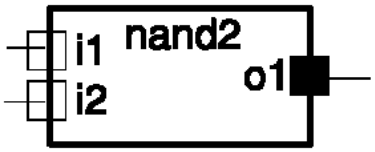
La prima descrizione completa VHDL

- Un esempio completo di descrizione VHDL (descrizione data-flow di una NAND a 2 ingressi, assieme al suo simbolo logico e al suo schematico) è riportata in seguito:

```
-- Descrizione Data-Flow di una nand a 2 ingressi
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--
ENTITY nand_2 IS PORT (
         i1 : IN std_logic;
         i2 : IN std_logic;
         o1 : OUT std_logic );
--
ARCHITECTURE dataflow OF nand_2 IS
BEGIN
         o1 <= NOT(i1 AND i2) AFTER 3ns;
END dataflow;
```



(a) Simbolo Logico

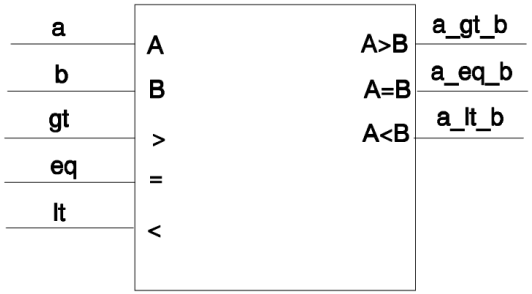


(b) Schematic VHDL

Esempio: un comparatore binario

- Un esempio è quello di un comparatore binario (cascadable), di cui è riportata una descrizione VHDL (con la sola definizione di architecture ma senza il body dell'architecture) e lo schematic.

```
-- Descrizione di un comparatore a 2 ingressi
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--
ENTITY bit_comparator IS
    PORT (a, b,           -- data inputs
    gt,                   -- previous greater than
    eq,                   -- previous equal
    lt : IN std_logic;    -- previous less than
    a_gt_b,               -- greater than
    a_eq_b,                  -- equal
    a_lt_b : OUT std_logic   -- less than
    );
END bit_comparator;

ARCHITECTURE gate_level OF bit_comparator IS
```



Confronto con il Sw (1)

- Volendo stabilire un paragone con un programma C:
  - la dichiarazione di una entity è analoga al prototipo di una funzione C (si definisce l'interfaccia ma non l'algoritmo);
  - la descrizione dell'architecture, invece, corrisponde al corpo della funzione C (si descrive l'algoritmo);

```
int somma (int a, int b);
--
int somma (int a, int b)
{
return a+b; }
```

```
ENTITY nand3 IS
        PORT (i1, i2: IN BIT; o1 : OUT BIT);
END nand3;
--
ARCHITECTURE single_delay OF nand3 IS
BEGIN
        o1 <= NOT(i1 AND i2) AFTER 3ns;
END single_delay;
```

Confronto con il Sw (2)

- I ports ed i generics sono i parametri di scambio (*parametri formali*) di una design entity VHDL;
- Anche nella definizione di una function C c'è una lista di parametri formali (a, b, il valore di ritorno), ciascuno con il suo tipo (es. int) ed il suo modo (uscita per il valore di ritorno, solo ingresso per le variabili a, b);

```
int somma (int a, int b) {
return a+b; }
```

- Solo quando si istanzia una design entity si decide quali segnali (*parametri concreti*) mappare sui vari porti;
- I parametri concreti nel caso del linguaggio C sono le variabili che vengono usate quando si richiama la function:

```
Spesa_totale = somma (spesa_gennaio, spesa_febbraio);
```

Confronto con il Sw (3)

- Una dichiarazione di una entity E non istanzia nessun componente, piuttosto definisce l'interfaccia di tutti i componenti che sono "esemplari" o "istanze" dell'entity E.
- Questo è analogo alla definizione di un tipo T di variabile in C: la dichiarazione del tipo non definisce nessuna variabile V, ma ogni variabile V del tipo T è un "esemplare" o "istanza" del tipo T.

Architecture (1)

- Abbiamo detto che la specificazione di una interfaccia definisce come il modulo interagisce con il mondo esterno, mentre la specificazione di una architecture descrive una particolare implementazione di un modulo;
- In generale, possono esistere diverse architetture per una stessa interfaccia:
  - ogni architecture body può descrivere lo stesso modulo (che ha quindi un'unica interfaccia) da diversi punti di vista (livelli di astrazione), livelli di dettaglio, …
  - In questo caso, all'atto della istanziazione di un modulo, è necessario scegliere il binding;

Architecture (2)

- Una architecture è composta da due parti separate:
  - una parte dichiarativa: in essa vengono definiti gli oggetti (tipi, segnali, costanti, component, …) che saranno usati per costruire la descrizione;
  - un corpo (*architecture body*): la vera e propria descrizione del componente, usando i segnali e i generics dell'interfaccia e gli oggetti definiti nella parte dichiarativa;

Parte dichiarativa

Architecture body

```
ARCHITECTURE identifier OF entity_name IS
    dichiarazioni iniziali
BEGIN
    Specifica del componente in  funzione degli
    ingressi, dei parametri fisici e degli altri
    parametri.
END identifier;
```

Parte dichiarativa (1)

```
ARCHITECTURE nome_architettura OF nome_entita IS
    dichiarazioni dei tipi usati
    definizione delle costanti
    dichiarazioni dei segnali interni al modulo
    definizione di funzioni o procedure
    dichiarazioni dei componenti
    specificazione della configurazione dei componenti (binding)
BEGIN
    statement concorrente

    statement concorrente
END nome_architettura;
```

Parte dichiarativa (2)

```
TYPE nome_tipo : definizione_tipo;
-- dichiarazione di tipo
CONSTANT nome_costante : tipo_costante := valore_costante;
-- dichiarazione di costante
SIGNAL nome_segnale : tipo_segnale [:= valore_iniziale];
-- dichiarazione di segnali
```

- Tutte le dichiarazioni sono sono locali al modulo;
- Le dichiarazioni di tipi e costanti servono a rendere più leggibile e manutenibile la descrizione;
- Le dichiarazioni dei segnali servono per "istanziare" dei fili per collegare fra loro i blocchi logici dell'architecture body (indipendentemente dal modo in cui sono descritti: assegnazioni, istanziazione di blocchi, process, …);

```
TYPE std_ulogic IS ( U , X , 0 , -- indefinito, più driver, 0 logico,
    1 , Z , W , L , H , - );     -- 1 logico, weak X, 0, 1, don t care
CONSTANT t_clock : time := 10ns;
SIGNAL in0 : std_ulogic :=  1 ;
SIGANL in1 : bit;
```

Parte dichiarativa (3)

```
COMPONENT nome_componente PORT (dichiarazione dei porti)
                GENERIC (dichiarazione dei generics);
-- dichiarazioni di componenti
FOR ALL | elenco di label : nome_componente
                USE ENTITY nome_libreria.nome_entity(nome_architecture);
-- specificazione della configurazione
```

- Le dichiarazioni di componenti servono a definire le interfacce delle design entity usate, mentre le specificazioni delle configurazioni servono a fissare per ogni istanza di un component quale architecture si debba usare;

```
COMPONENT n1 PORT (i1: IN BIT;
                   o1: OUT BIT);
END COMPONENT;
FOR ALL : n1 USE ENTITY WORK.inv (single_delay);
```

- Si vede come la dichiarazione di un component sia simile alla definizione di una entity, poiché si specificano le stesse informazioni per i segnali (nome, modo, tipo) e per i generic (nome, tipo, valore).

Architecture body

- Il corpo dell'architecture è composta da statement concorrenti (istanziazione di un componente, assegnazione di un segnale, definizione di un process, …)
- All'interno di un architecture body si possono usare i port come se fossero dei segnali, ma bisogna rispettare il loro modo:
  - segnali con modo IN possono essere solo letti e non scritti;
  - segnali con modo OUT possono essere solo scritti e non letti;
- un segnale è scritto se è a sinistra di una assegnazione.

Binding (1)

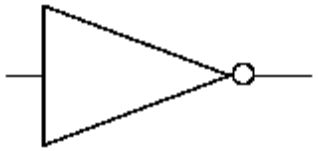
- Una architettura (*architecture body*) può usare entità descritte separatamente e disponibili nella libreria corrente (work) o in altre librerie (*design libraries*)
- Una architettura deve dichiarare un component, che è un "prototipo" del blocco che si vuole istanziare, è un blocco virtuale, perché prima di istanziarlo deve essere mappato in un blocco specifico per definirne il comportamento ingresso-uscita.
- La *component declaration* specifica i ports (ed eventualmente i generics) ma non qual è l'*architecture body* che verrà istanziato concretamente.

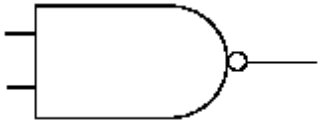
Binding (2)

- Nella parte dichiarativa di un architecture body (prima dello statement BEGIN) sono presenti:
  - una dichiarazione del componente (component declaration): si specifica l'interfaccia del componente che sarà istanziato;
  - una specificazione della configurazione (configuration specification): si specifica il binding cioè qual è l'architettura che viene usata nelle varie istanziazioni.
- Nella statement part (dopo il comando BEGIN) sono definite le istanziazioni dei componenti.

```
ARCHITECTURE nome_architettura OF nome_entita IS
     dichiarazioni di componenti
     binding dei component con le architectures
BEGIN
    statement concorrente
    instanziazione dei componenti
END nome_architettura;
```

Binding (3)





```
ARCHITECTURE nome_architettura OF nome_entita IS
     COMPONENT n1 PORT (i1: IN BIT; o1: OUT BIT); END COMPONENT;
     COMPONENT n2 PORT (i1, i2: IN BIT; o1: OUT BIT); END COMPONENT;
     FOR ALL : n1 USE ENTITY WORK.inv (single_delay);
     FOR ALL : n2 USE ENTITY WORK.nand2 (single_delay);
BEGIN
```

Istanziazione (1)

- L'istanziazione di un componente è uno statement concorrente, ovvero è sempre "attivo";
  - Quando si verifica un evento su uno dei suoi port di ingresso, viene "attivato" il componente che reagisce secondo il suo comportamento I/O, generando eventi sui segnali di uscita;
- E' possibile istanziare un componente sia assegnando ordinatamente i segnali e le costanti sui ports e sui generics ordinatamente secondo la definizione (assegnazione posizionale), che usando una associazione esplicita per nome (associazione nominale);

```
nome_label: nome_componente
    PORT MAP (segnale_1, segnale_2,  , segnale_N)
    GENERIC MAP (valore_1, valore_2,  , valore_M);
-- istanziazione con assegnazione posizionale
```

- Ogni istanziazione di un modulo richiede una label diversa;
- La label rappresenta il nome del blocco logico (altrimenti non sarebbe possibile distinguere diverse instanze della stessa entity);

Istanziazione (2)

- Nel caso della associazione nominale l'ordine con cui vengono associati i segnali con i ports, è evidentemente ininfluente, mentre nel caso di associazione posizionale è fondamentale rispettare l'ordine con cui i ports sono stati dichiarati;

```
nome_label: nome_componente
    PORT MAP (segnale_1, segnale_2,  , segnale_N)
    GENERIC MAP (valore_1, valore_2,  , valore_M);
-- istanziazione con assegnazione posizionale
```

```
nome_label: nome_componente
    PORT MAP (nome_segnale_1 => segnale_1,

        nome_segnale_N => segnale_N)
    GENERIC MAP (nome_generic_1 := valore_1,

        nome_generic_M := valore_M);
-- istanziazione con assegnazione nominale
```

Istanziazione (3)

```
COMPONENT n1 PORT (i1: IN BIT; o1: OUT BIT); END COMPONENT;
COMPONENT n2 PORT (i1, i2: IN BIT; o1: OUT BIT); END COMPONENT;
FOR ALL : n1 USE ENTITY WORK.inv(single_delay);
FOR ALL : n2 USE ENTITY WORK.nand2 (single_delay);
BEGIN
g0 : n1 PORT MAP (a, im1);
       -- associazione posizionale
g1 : n1 PORT MAP (i1 => b, o1 => im2);
       -- nominale nello stesso ordine
g2 : n2 PORT MAP (o1 => im3, i2 => im2, i1 => a);
       -- nominale in ordine diverso
g3 : n2 PORT MAP (a, gt, im4);
       -- associazione posizionale
```

Assegnazione di un segnale (1)

- Così come l'istanziazione di un componente, anche una assegnazione di un segnale è uno statement concorrente, ovvero è sempre "attivo";
- Quando si verifica un evento su uno dei segnali a destra dell'assegnazione:
  1. viene calcolato il valore del segnale a sinistra, secondo l'espressione logica dell'assegnazione,
  2. una transazione viene schedulata sul segnale di uscita secondo il timing dell'assegnazione;

```
-- Descrizione Data-Flow di una porta complessa AOI
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY AOI IS PORT (
    i1 : IN std_logic;
    i2 : IN std_logic;
    i3 : IN std_logic;
    O1 : OUT std_logic );
--
ARCHITECTURE dataflow OF AOI IS
BEGIN
    o1 <= NOT(i1 AND i2 OR i3) AFTER 3ns;
END dataflow;
```

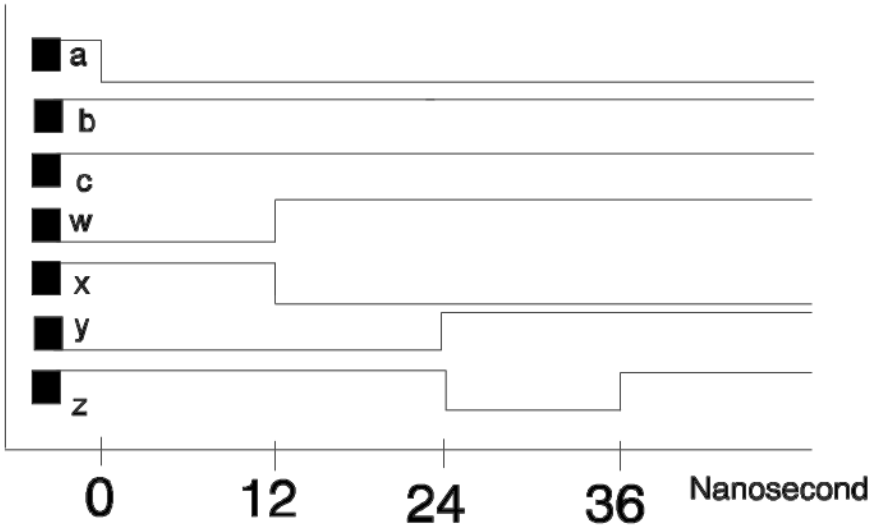
Assegnazione di un segnale (2)

- Quale è la differenza fra le due architetture?

```
-- Descrizione Data-Flow di una porta complessa AOI
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY porta_complessa IS
    PORT ( a, b, c : IN std_logic;
           z : OUT std_logic);
END porta_complessa ;
--
ARCHITECTURE concurrent OF porta_complessa IS
    SIGNAL w, x, y : std_logic ;
BEGIN
    w <= NOT a AFTER 12 NS;
    x <= a AND b AFTER 12 NS;
    y <= c AND w AFTER 12 NS;
    z <= x OR y AFTER 12 NS;
END concurrent;
---
ARCHITECTURE single_assignment OF porta_complessa IS
BEGIN
    z <= (a AND b) or (c AND (NOT a)) AFTER 36 NS;
END single_assignment;
```

Assegnazione di un segnale (3)

- Le 2 architetture hanno lo stesso comportamento funzionale (sono la stessa funzione booleana), ma hanno tempificazioni differenti perché la prima architettura ha 4 livelli di logica, mentre la seconda ha un solo livello;



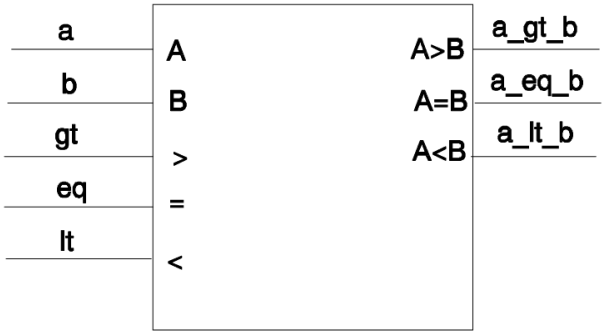

```
-- architettura concurrent
w <= NOT a AFTER 12 NS;
x <= a AND b AFTER 12 NS;
y <= c AND w AFTER 12 NS;
z <= x OR y AFTER 12 NS;
```

- Una variazione sul segnale c comporta una (eventuale) variazione di z dopo 24ns nella prima architettura, mentre nella assegnazione dell'altra architettura dopo 36ns;

```
-- architettura
-- singola_assegnazione
z <=    (a AND b) or
        (c AND (NOT a))
       AFTER 36 NS;
```

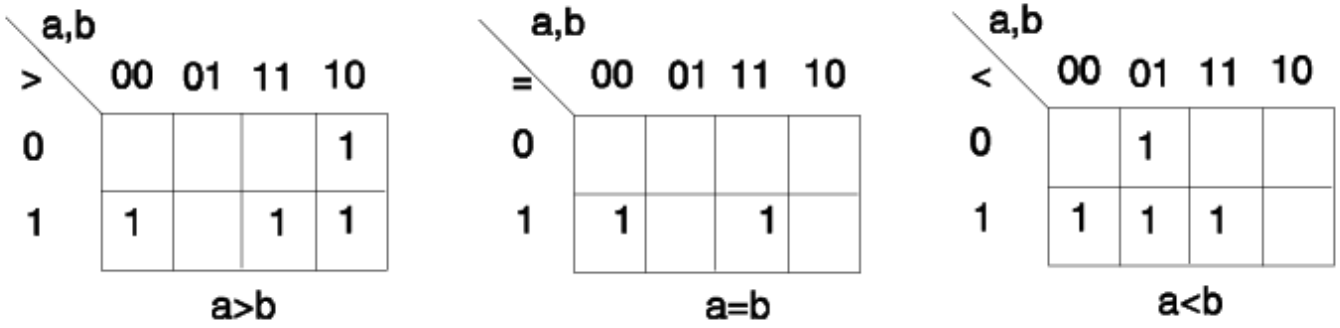
1-bit cascadable comparator (1)

- Il blocco 1-bit cascadable comparator permette di confrontare 2 bit e segnalare quale dei 2 è maggiore o se sono uguali, inoltre è progettato in modo da permettere di costruire un comparatore di word su un numero qualsiasi di bit, mettendo diversi blocchi in cascata (*cascadable*);
- Il blocco ha 2 ingressi primari (a, b), 3 ingressi di controllo (gt, eq, lt) e 3 uscite (a_gt_b, a_eq_b, a_lt_b);
- Il suo comportamento è riportato in seguito:
  - a_gt_b = 1 $\Leftrightarrow$ a > b oppure a = b e gt = 1;
  - a_eq_b = 1 $\Leftrightarrow$ a = b e eq = 1;
  - a_lt_b = 1 $\Leftrightarrow$ è il complemento di a_gt_b;
- Dato che a e b sono binari a > b $\Leftrightarrow$ (a, b) = (1, 0), analogamente per a < b;
- Le 3 uscite sono mutuamente esclusive,
  ed inoltre a_lt_b = not(a_gt_b);
- Per avere un comparatore di 2 word da 1 bit,
- basta mettere gli ingressi (a_gt_b, a_eq_b, a_lt_b) = (0, 1, 0);



1-bit cascadable comparator (2)

- Sulla base del comportamento richiesto dalle specifiche del comparatore si possono derivare facilmente le equazioni booleane, che poi possono essere minimizzate attraverso le mappe di Karnaugh;



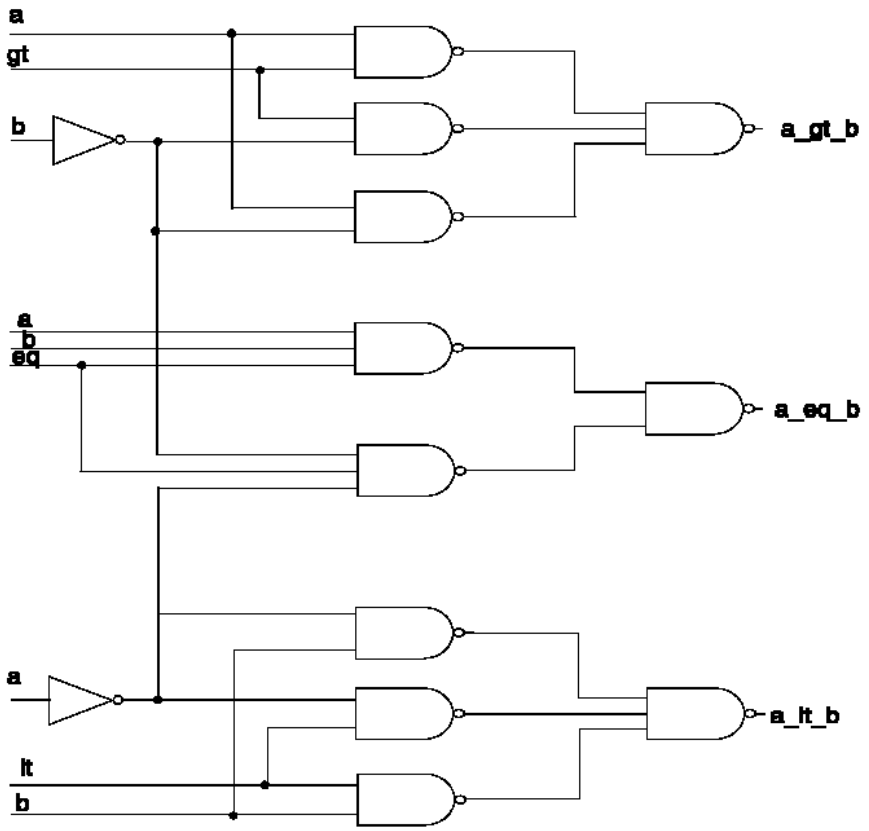
ottenendo:

  - a_gt_b = (a and gt) or (not(b) and gt) or (a and not(b))
  - a_eq_b = (a and b and eq) or (not(a) and not(b) and eq)
  - a_lt_b = (not(a) and lt) or (b and lt) or (not(a) and b))

- Usando il teorema di DeMorgan si possono ottenere le equazioni in termini di sole NAND (sia NAND-3 che NAND-2) e di NOT;

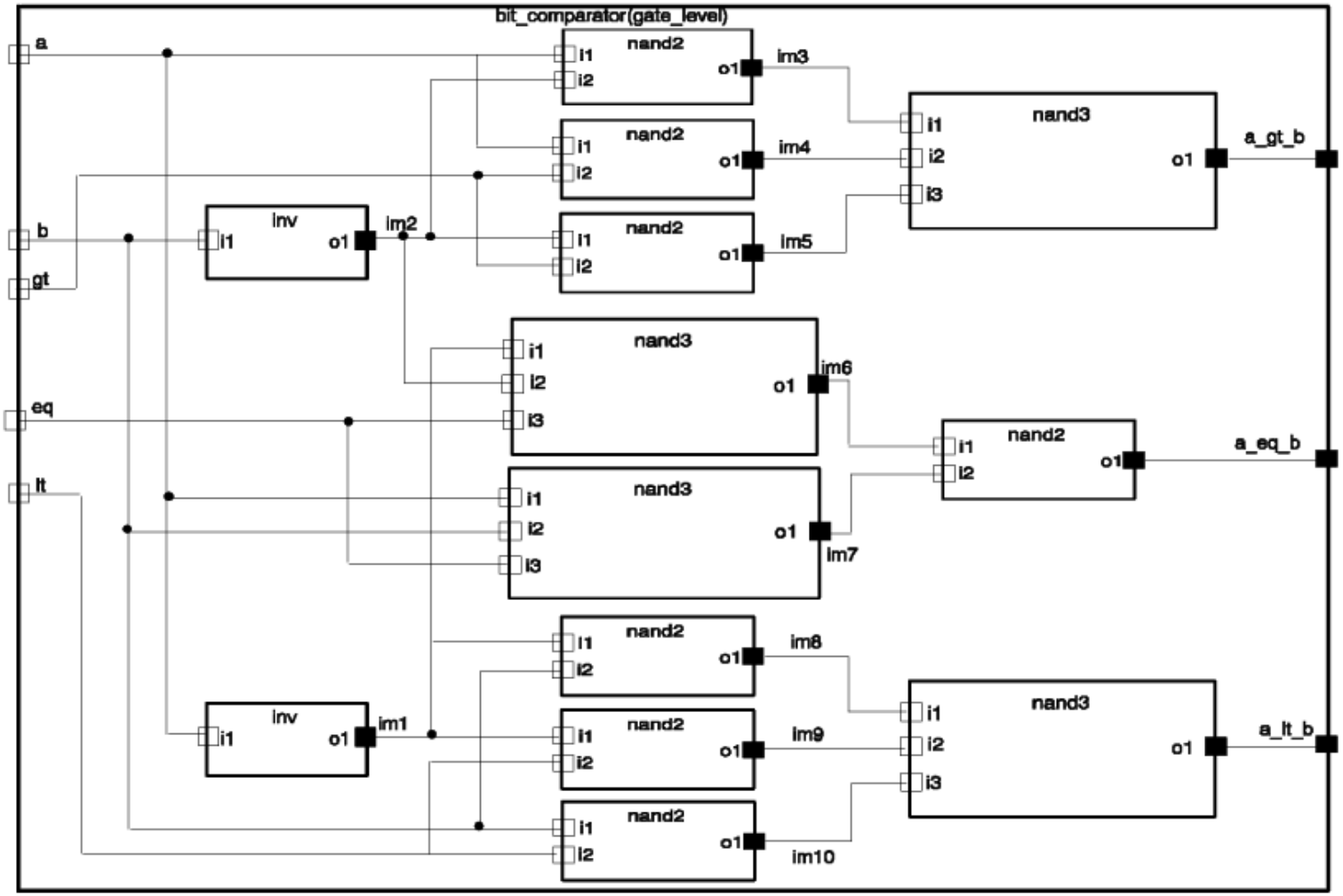
1-bit cascadable comparator (3)

- Il risultato è la rete logica riportata a fianco:
- Si vede come a_gt_b e a_lt_b usino la medesima logica;
- In generale si sarebbe anche potuto negare il risultato di a_gt_b, usando meno porte, ma aggiungendo un livel-lo di logica;



1-bit cascadable comparator (4)

- Quindi lo schematico da implementare in VHDL è il seguente:



1-bit cascadable comparator (5)

```
ENTITY bit_comparator IS
PORT (a, b,                          -- data inputs
      gt,                            -- previous greater than
      eq,                            -- previous equal
      lt : IN BIT;                   -- previous less than
      a_gt_b,                        -- greater
      a_eq_b,                        -- equal
      a_lt_b : OUT BIT);             -- less than
END bit_comparator;
--
ARCHITECTURE gate_level OF bit_comparator IS

COMPONENT n1 PORT (i1: IN BIT; o1: OUT BIT); END COMPONENT;
COMPONENT n2 PORT (i1, i2: IN BIT; o1: OUT BIT); END COMPONENT;
COMPONENT n3 PORT (i1, i2, i3: IN BIT; o1: OUT BIT); END COMPONENT;

FOR ALL : n1 USE ENTITY WORK.inv(single_delay);
FOR ALL : n2 USE ENTITY WORK.nand2 (single_delay);
FOR ALL : n3 USE ENTITY WORK.nand3 (single_delay);
-- Intermediate signals
SIGNAL im1,im2, im3, im4, im5, im6, im7, im8, im9, im10 : BIT;
```

1-bit cascadable comparator (6)

```
BEGIN
-- a_gt_b output
g0 : n1 PORT MAP (a, im1);
g1 : n1 PORT MAP (b, im2);
g2 : n2 PORT MAP (a, im2, im3);
g3 : n2 PORT MAP (a, gt, im4);
g4 : n2 PORT MAP (im2, gt, im5);
g5 : n3 PORT MAP (im3, im4, im5, a_gt_b);
-- a_eq_b output
g6 : n3 PORT MAP (im1, im2, eq, im6); g7 : n3 PORT MAP (a, b, eq, im7);
g8 : n2 PORT MAP (im6, im7, a_eq_b);
-- a_lt_b output
g9 : n2 PORT MAP (im1, b, im8);
g10 : n2 PORT MAP (im1, lt, im9);
g11 : n2 PORT MAP (b, lt, im10);
g12 : n3 PORT MAP (im8, im9, im10, a_lt_b);
END gate_level;
```

Generics (1)

- I **generics** sono specificati nell'interfaccia di una entity e permettono di parametrizzare la descrizione:
  - rispetto al numero di bit degli ingressi e dell'uscita del modulo;
  - esplicitando caratteristiche del timing (ad es. tempi di propagazione dei moduli, …);
  - definendo alcuni parametri fisici del modulo (ad es. la capacità dei nodi di ingresso o di uscita, la dipendenza del comportamento del modulo dalla temperatura, …);
- Il valore dei generics devono essere specificati in almeno uno dei seguenti modi:
  1. come valori di default (nella dichiarazione di entity);
  2. all'atto della dichiarazione del component;
  3. all'atto dell'instanziazione del component;
- In caso di uso di diversi modi di specificazione dei generics, il valore usato è quello che ha priorità maggiore (nella lista il caso 3 sovrascrive gli altri, …);
- Si possono usare come generics solo quantità costanti (appartenenti alla classe *constant*) o espressioni statiche, ovvero che coinvolgono solo quantità costanti;

Generics (2)

- Un esempio di come i generics possano essere usati per specificare una classe di entity che differiscono nei parametri della struttura è quello della descrizione di una ROM;
  - In questo caso 2 generics sono usati per specificare il numero dei bit del data-bus e dell'address bus;
  - La ROM descritta permette di memorizzare 2^{depth} word, ciascuna di *width* bit;

```
ENTITY rom IS
        GENERIC( width : positive := 32;
                 depth : positive := 4);
        PORT(    enable : IN std_logic;
                 address : IN std_logic_vector(depth 1 downto 0);
                 data : OUT std_logic_vector(width 1 downto 0) );
END rom;
```

- E' importante notare che se non è dato nessun valore di default per i generics, quando l'entity è istanziata, i valori effettivi devono essere specificati nella dichiarazione o nella istanziazione: alla fine i generics devono essere risolti;

Generics (3)

```
COMPONENT rom
          GENERIC (width : positive := 8;
                   depth : positive := 4);
          PORT (   enable : IN std_logic;
                   address : IN  std_logic_vector(depth 1 downto 0);
                   data : OUT std_logic_vector(width 1 downto 0) );
END COMPONENT;

rom_0 : rom GENERIC MAP (16, 4) PORT MAP ( );
-- rom di 16 word di 16 bit
rom_1 : rom PORT MAP ( );
-- rom di 16 word di 8 bit
rom_2 : rom GENERIC MAP (depth => 5, width => 32) PORT MAP ( );
-- rom di 32 word di 32 bit
```

- Nel primo caso di istanziazione (rom_0) il valore 16 sovrascrive il valore di width 8 assegnato nella dichiarazione del component (che a sua volta ha sovrascritto il valore di width 32 del generic di default);

Generics (4)

- I generics possono anche essere usati per passare dei parametri che rappresentano costanti fisiche che possono variare in diverse istanzazioni dello stesso modulo.

```
ENTITY processor IS
    GENERIC (max_clock_freq : frequency := 30 MHz);
    PORT (clock : IN std_logic;
        address : OUT integer;
        data : INOUT word_32;
        control : OUT proc_control;
        ready : IN std_logic);
END processor;
```

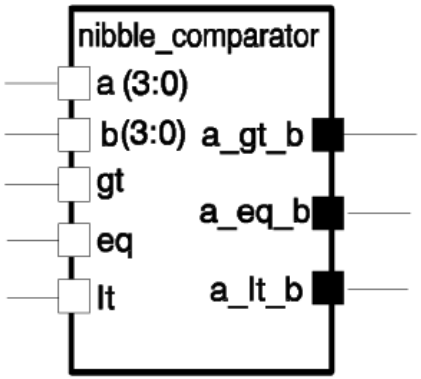
I costrutti iterativi

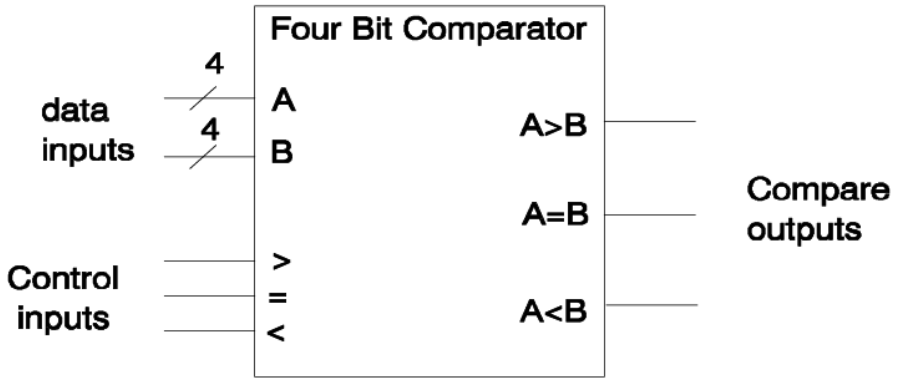
- I blocchi logici dei sistemi digitali spesso sono composti da un solo blocco più semplice istanziato molte volte (ad es. un addizionatore carry-ripple, una porta logica che operi su word, …);
- Per descrivere efficacemente questo tipo di blocchi il VHDL mette a disposizione il costrutto **generate** che permette di istanziare iterativamente un insieme di blocchi;
- Lo statement generate può essere usato sia con il costrutto FOR che con un costrutto IF;

Nibble comparator (1)

- Per come è stato progettato il comparatore ad 1 bit è semplice ottenere un nibble comparator, ovvero un blocco che segnala se il numero rappresentato da a (una word di 4 bit) è maggiore, minore o uguale ad un altro numero rappresentato da b;

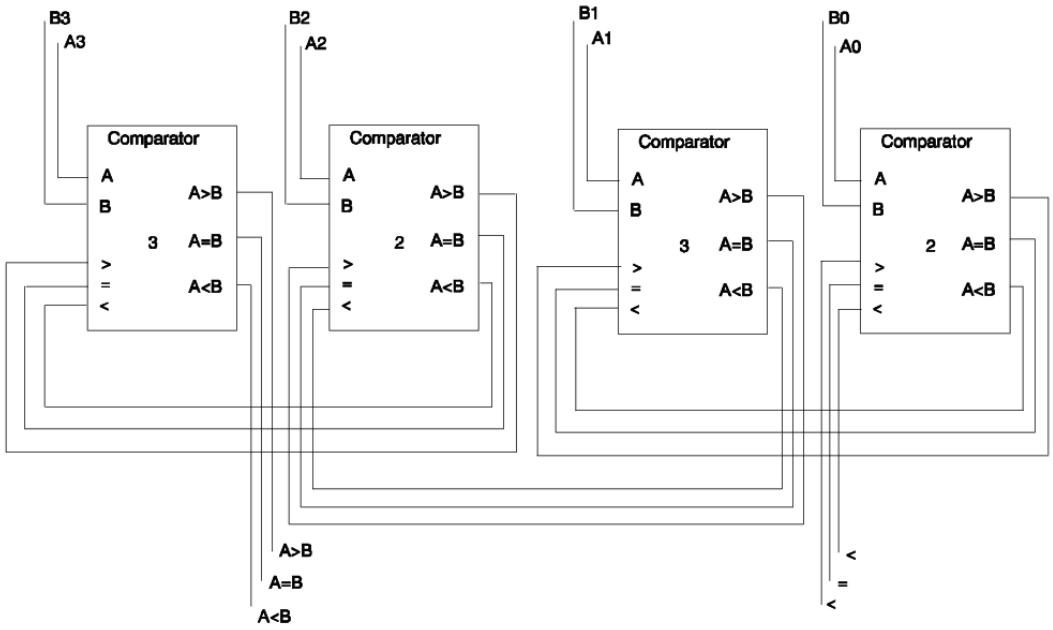
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY nibble_comparator IS
PORT ( a, b : IN std_logic_vector(3 DOWNTO 0);    -- a and b data inputs
       gt,  eq, lt : IN std_logic;  -- previous greater, equal, less than
       a_gt_b,                      -- a > b
       a_eq_b,                      -- a = b
       a_lt_b : OUT std_logic);     -- a < b
END nibble_comparator;
```





Nibble comparator (2)

- E' facile capire come mettendo opportunamente in cascata i comparatori ad 1 bit, se ne possa ottenere uno su 4 bit;
  - Infatti 2 numeri a e b sono uguali se hanno ordinatamente uguali tutte le cifre;
  - Un numero a è maggiore di b se la cifra di maggior peso di a è maggiore della omologa di b, oppure, se queste sono uguali, la cifra di peso minore di a è maggiore della omologa di b, oppure, se le 2 più significative sono uguali etc. …
- Il risultato è che collegando i comparatori ad 1 bit (con una specie di "riporto") nel modo di figura si ha il comportamento desiderato;



Nibble comparator (3)

- Si vede come i segnali si propaghino dai blocchi di peso inferiore a quelli di peso superiore;

```
      3  2  1  0
_____________________________________________________
A:    0  1  0  0  result propagates from bit 1
B:    0  1  1  0
_____________________________________________________

A:    1  0  1  1  bit 3 produces the result immediately
B:    0  0  1  1
_____________________________________________________
```

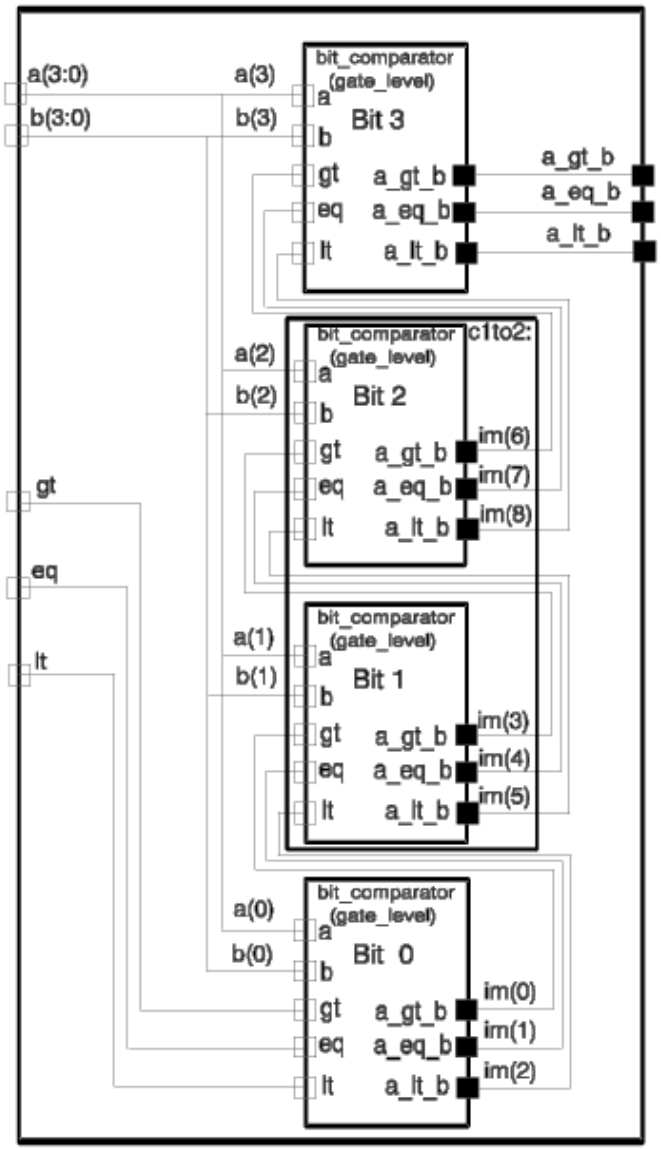
Nibble comparator (4)

- Una descrizione VHDL strutturale che usi il costrutto FOR … GENERATE è la seguente:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--
ENTITY nibble_comparator IS
PORT (   a, b : IN std_logic_vector(3 DOWNTO 0);    -- a and b data inputs
         gt, eq, lt : IN std_logic;          -- previous greater, equal, less than
         a_gt_b,                             -- a > b
         a_eq_b,                             -- a = b
         a_lt_b : OUT std_logic);            -- a < b
END nibble_comparator;
--
ARCHITECTURE iterative OF nibble_comparator IS

COMPONENT comp1 PORT (a, b, gt, eq, lt : IN std_logic;
         a_gt_b, a_eq_b, a_lt_b : OUT std_logic);
END COMPONENT;
FOR ALL : comp1 USE ENTITY WORK.bit_comparator(gate_level);
SIGNAL im : std_logic_vector(0 TO 8);
```

Nibble comparator (5)

```
BEGIN
-- primo blocco
c0: comp1 PORT MAP (
     a(0), b(0),
     gt, eq, lt,
     im(0), im(1), im(2));

-- ciclo sugli intermedi
c1to2: FOR i IN 1 TO 2 GENERATE
     c: comp1 PORT MAP (
          a(i), b(i),
          im(i*3-3), im(i*3-2), im(i*3-1),
          im(i*3+0), im(i*3+1), im(i*3+2) );
END GENERATE;

-- ultimo blocco
c3: comp1 PORT MAP (
     a(3), b(3),
     im(6), im(7), im(8),
     a_gt_b, a_eq_b, a_lt_b);
END iterative;
```



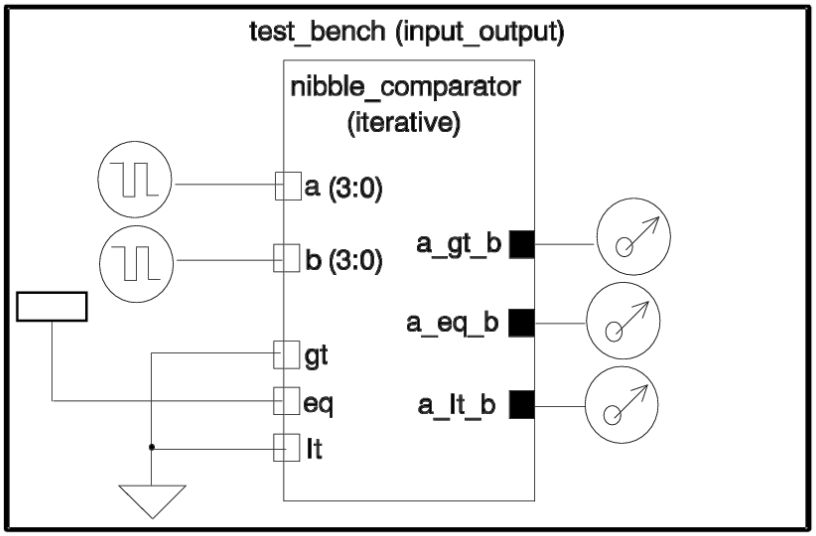
Nibble comparator (6)

```
ARCHITECTURE iterative OF nibble_comparator IS
COMPONENT comp1 PORT (a, b, gt, eq, lt : IN std_logic;
          a_gt_b, a_eq_b, a_lt_b : OUT std_logic);
END COMPONENT;
FOR ALL : comp1 USE ENTITY WORK.bit_comparator(gate_level);
CONSTANT n : INTEGER := 4;
SIGNAL im : std_logic_vector(0 TO (n-1)*3-1);
BEGIN
c_all: FOR i IN 0 TO n-1 GENERATE
          l: IF i = 0 GENERATE
                    least: comp1 PORT MAP (
                              a(i), b(i), gt, eq, lt, im(0), im(1), im(2) );
          END GENERATE;

          m: IF i = n-1 GENERATE
                    most: comp1 PORT MAP (a(i), b(i),
                              im(i*3-3), im(i*3-2), im(i*3-1), a_gt_b, a_eq_b, a_lt_b);
          END GENERATE;

          r: IF i > 0 AND i < n-1 GENERATE
                    rest: comp1 PORT MAP (a(i), b(i),
                              im(i*3-3), im(i*3-2), im(i*3-1),
                              im(i*3+0), im(i*3+1), im(i*3+2) );
          END GENERATE;
END GENERATE;
END iterative;
```

Testbench (1)

- Per verificare il funzionamento e la tempificazione di un progetto VHDL, si usano i **testbench**;
- Un testbench è una design unit VHDL che:
  - dichiara e istanzia il blocco da testare (design under test, DUT);
  - applica una opportuna sequenza di stimoli;
  - verifica che le uscite siano quelle corrette secondo le specifiche;

- In pratica un testbench è una entità VHDL che non ha segnali di ingresso né di uscita, poiché in pratica non può essere istanziato da nessun altro blocco;
- Il più semplice dei testbench applica le sequenza degli ingressi, senza verificarne l'esattezza; sarà l'utente a verificare se le uscite sono quelle corrette;



Testbench (2)

- Un testbench per il nibble comparator è il seguente:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
--
ENTITY nibble_comparator_tb IS
END nibble_comparator;
--
ARCHITECTURE input_output OF nibble_comparator_test_bench IS

COMPONENT comp4 PORT (a, b : IN std_logic_vector (3 DOWNTO 0);
          a_gt_b, a_eq_b, a_lt_b : IN std_logic;
          a_gt_b_out, a_eq_b_out, a_lt_b_out : OUT std_logic);
END COMPONENT;
FOR a1 : comp4 USE ENTITY WORK.nibble_comparator(iterative);

SIGNAL a, b : std_logic_vector (3 DOWNTO 0);
SIGNAL eql, lss, gtr, gnd : std_logic;
SIGNAL vdd : std_logic := '1';

BEGIN

-- istanziazione del DUT
a1: comp4 PORT MAP (a, b, gnd, vdd, gnd, gtr, eql, lss);
```

Testbench (3)

```
-- applicazione dei segnali su a2
a2: a <=  "0000",                -- a = b (steady state)
          "1111" AFTER 0500 NS, -- a > b (worst case)
          "1110" AFTER 1500 NS, -- a < b (worst case)
          "1110" AFTER 2500 NS, -- a > b (need bit 1 info)
          "1010" AFTER 3500 NS, -- a < b (need bit 2 info)
          "0000" AFTER 4000 NS, -- a < b (steady state, prepare for next)
          "1111" AFTER 4500 NS, -- a = b (worst case)
          "0000" AFTER 5000 NS, -- a < b (need bit 3 only, best case)
          "0000" AFTER 5500 NS, -- a = b (worst case)
          "1111" AFTER 6000 NS; -- a > b (need bit 3 only, best case)

-- applicazione dei segnali su a3
a3 : b <= "0000",                -- a = b (steady state)
          "1110" AFTER 0500 NS, -- a > b (worst case)
          "1111" AFTER 1500 NS, -- a < b (worst case)
          "1100" AFTER 2500 NS, -- a > b (need bit 1 info)
          "1100" AFTER 3500 NS, -- a < b (need bit 2 info)
          "1111" AFTER 4000 NS, -- a < b (steady state, prepare for next)
          "1111" AFTER 4500 NS, -- a = b (worst case)
          "1111" AFTER 5000 NS, -- a < b (need bit 3 only, best case)
           0000" AFTER 5500 NS, -- a = b (worst case)
           0000" AFTER 6000 NS; -- a > b (need bit 3 only, best case)
END input_output;
```

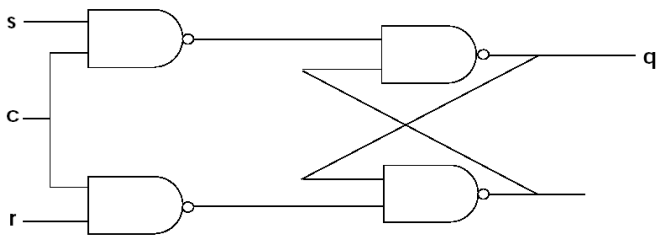
Testbench (4)

- Nella tabella i puntini rappresentano il fatto che un segnale ha lo stesso valore dell'istante precedente;
- Dall'istante 0 fino a 5ns non c'è attività e tutti gli ingressi sono bassi;
- Dall'istante 5ns, ogni 10ns sono applicati dei nuovi test patterns;
- Agli istanti 5ns e 15ns ci sono i worst cases (propagazione attraverso tutti gli stadi) → 48 ns;
- Agli istanti 50ns e 60ns ci sono i best cases (l'ultimo stadio decide) → 15 ns;
- Si testano le varie condizioni di funzionamento scambiando l'ingresso a con quello b, per sollecitare tutto il modulo;

TIME (NS)	SIGNAL NAMES				
	a(3:0)	b(3:0)	gtr	eql	lss
0	"0000"	"0000"	'0'	'0'	'0'
5	.....	.....	...	'1'	...
500	"1111"	"1110"	...	...	...
544	.....	.....	'1'	...	...
548	.....	.....	...	'0'	...
1500	"1110"	"1111"	...	...	...
1544	.....	.....	'0'	...	...
1548	.....	.....	...	...	'1'
2500	.....	"1100"	...	...	...
2533	.....	.....	...	...	'0'
2537	.....	.....	'1'	...	...
3500	**"1010"**	.....	...	...	...
3522	.....	.....	'0'	...	...
3526	.....	.....	...	...	'1'
4000	"0000"	"1111"	...	...	...
4500	"1111"	.....	...	...	...
4544	.....	.....	...	'1'	...
4548	.....	.....	...	...	'0'
5000	"0000"	.....	...	...	...
5011	.....	.....	...	'0'	...
5015	.....	.....	...	...	'1'
5500	.....	"0000"	...	...	...
5544	.....	.....	...	...	'0'
5548	.....	.....	...	'1'	...
6000	"1111"	.....	...	...	...

SR Latch (1)

- Un latch SR può essere realizzato usando 4 NAND incrociate, come in figura;

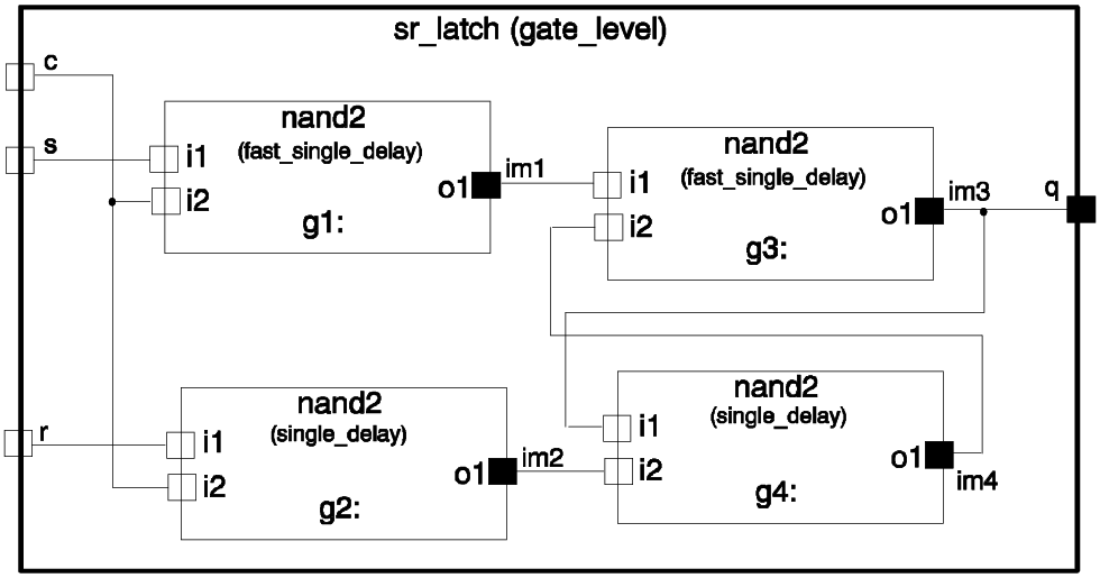



```
ENTITY nand2 IS PORT (
     i1, i2: IN std_logic; o1: OUT std_logic );
END nand2;
--
ARCHITECTURE single_delay OF nand2 IS
BEGIN
     o1 <= i1 NAND i2 AFTER 4 NS;
END single_delay;
```

```
ENTITY sr_latch IS PORT (
          s, r, c: IN std_logic; q : OUT std_logic );
--
ARCHITECTURE gate_level OF sr_flipflop IS
COMPONENT n2 PORT (i1, i2: IN std_logic;  o1: OUT std_logic); END COMPONENT;
FOR ALL : n2 USE ENTITY WORK.nand2 (single_delay);
SIGNAL im1, im2, im3, im4 : std_logic;
BEGIN
g1 : n2 PORT MAP (s, c, im1);
g3 : n2 PORT MAP (im1, im4, im3);
g2 : n2 PORT MAP (r, c, im2);
g4 : n2 PORT MAP (im3, im2, im4);
q <= im3;
END gate_level;
```

SR Latch (2)

- Il latch fatto nel modo precedente non funziona perché nel caso in cui i segnali im3 e im4 assumano lo stesso valore nello stesso istante, il circuito oscilla indefinitamente (come nel caso di 2 NOT con lo stesso ritardo);
- Questo, in realtà, non è un limite per il simulatore VHDL perché è chiaro che questo comportamento deriva da troppe assunzioni ideali:
  - le interconnessioni non hanno ritardo;
  - si sono usate 4 porte perfettamente identiche;
- La soluzione sta nell'usare 2 NAND più veloci in uno dei 2 rami;



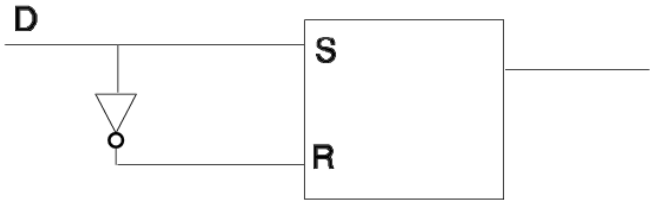
SR Latch (3)

```
ENTITY nand2 IS PORT (
     i1, i2: IN std_logic; o1: OUT std_logic );
END nand2;
--
ARCHITECTURE single_delay OF nand2 IS
BEGIN o1 <= i1 NAND i2 AFTER 4 NS;
END single_delay;
--
ARCHITECTURE fast_single_delay OF nand2 IS
BEGIN o1 <= i1 NAND i2 AFTER 3 NS;
END fast_single_delay;
```

```
ARCHITECTURE gate_level OF sr_flipflop IS
COMPONENT n2 PORT (i1, i2: IN std_logic;  o1: OUT std_logic);
END COMPONENT;
FOR g1, g3 : n2 USE ENTITY WORK.nand2(fast_single_delay);
FOR g2, g4 : n2 USE ENTITY WORK.nand2(single_delay);
SIGNAL im1, im2, im3, im4 : std_logic;
BEGIN
g1 : n2 PORT MAP (s, c, im1);
g3 : n2 PORT MAP (im1, im4, im3);
g2 : n2 PORT MAP (r, c, im2);
g4 : n2 PORT MAP (im3, im2, im4);
q <= im3;
END gate_level;
```

D Latch

- Adesso possiamo usare il nostro SR latch per realizzare un D latch secondo il collegamento riportato in figura:




```
ENTITY d_latch IS PORT(
          d, c : IN std_logic; q: OUT std_logic);
END d_latch;
--
ARCHITECTURE sr_based OF d_latch IS
COMPONENT sr_latch PORT (s, r, clk : IN std_logic; q : OUT std_logic);
END COMPONENT;
FOR ALL: sr_latch USE ENTITY WORK.sr_latch(gate_level);
COMPONENT inv_t PORT (i1 : IN std_logic; o1 : OUT std_logic); END COMPONENT;
FOR ALL: inv_t USE ENTITY WORK.inv_t(dataflow);
SIGNAL not_d: std_logic;
BEGIN
c1 : sr_latch PORT MAP (d, not_d, c, q);
c2 : inv_t PORT MAP (d, not_d);
END sr_based;
```