**ARCHITETTURE DI SISTEMI DIGITALI – GUIDA VHDL/VIVADO – DE BENE**

Il VHDL è, insieme al Verilog, il linguaggio più usato per la progettazione di sistemi elettronici digitali: Un progetto VHDL può essere trasformato direttamente in un **formato sintetizzabile**, utilizzabile per configurare un dispositivo FPGA o ASIC mediante un apposito tool detto sintetizzatore.

Rispetto agli approcci alla progettazione che abbiamo richiamato, il **VHDL consente di descrivere il comportamento di una macchina sia in termini dell’espressione logica realizzata che della struttura interna** intesa come interconnessione di componenti, utilizzando livelli di astrazione diversi. Il livello di astrazione utilizzato nel progetto **influisce sulla sua sintetizzabilità**, e livelli di astrazione differenti utilizzati nella descrizione di un circuito potrebbero dare luogo a implementazioni diverse nella tecnologia target.

Costrutti fondamentali: entity e architecture

* **Entity**: consente di definire l’interfaccia di un componente, ogni porto ha un nome\, una direzione, un tipo e una dimensione
* **Architecture**: consente di definire il comportamento di un componente specificato dopo la clausola **begin**. La stessa entitý puó avere piú implementazioni differenti. Per questo motivo è importante scegliere un nome per l’architettura. In tal caso verrá presa l’ultima disponibile e sintetizzata.

Immagine che contiene testo, Carattere, schermata, design

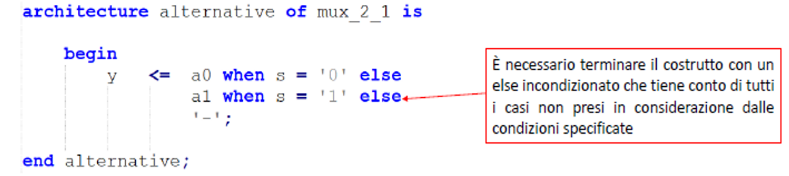
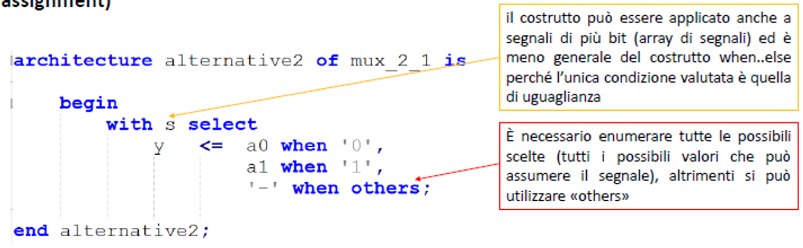
Descrizione generata automaticamente**std\_logic** è un **tipo (booleano)** che definisce i valori che possono assumere i segnali digitali secondo lo standard IEEE 1164:

* '**U**': **uninitialized**. This signal hasn't been set yet.
* '**X**': **unknown**. Impossible to determine this value/result.
* '**0**': logic 0
* '**1**': logic 1
* '**Z**': **High Impedance**
* '**W**': Weak signal, can't tell if it should be 0 or 1.
  + WEAK: non so dire se è 0 o 1
* '**L**': Weak signal that should probably go to 0
  + L: probabilmente va a 0
* '**H**': Weak signal that should probably go to 1
  + H: probabilmente va a 1
* '**-**': **Don't care**.

//Le direzioni dei ports possono essere IN/OUT/INOUT/BUFFER

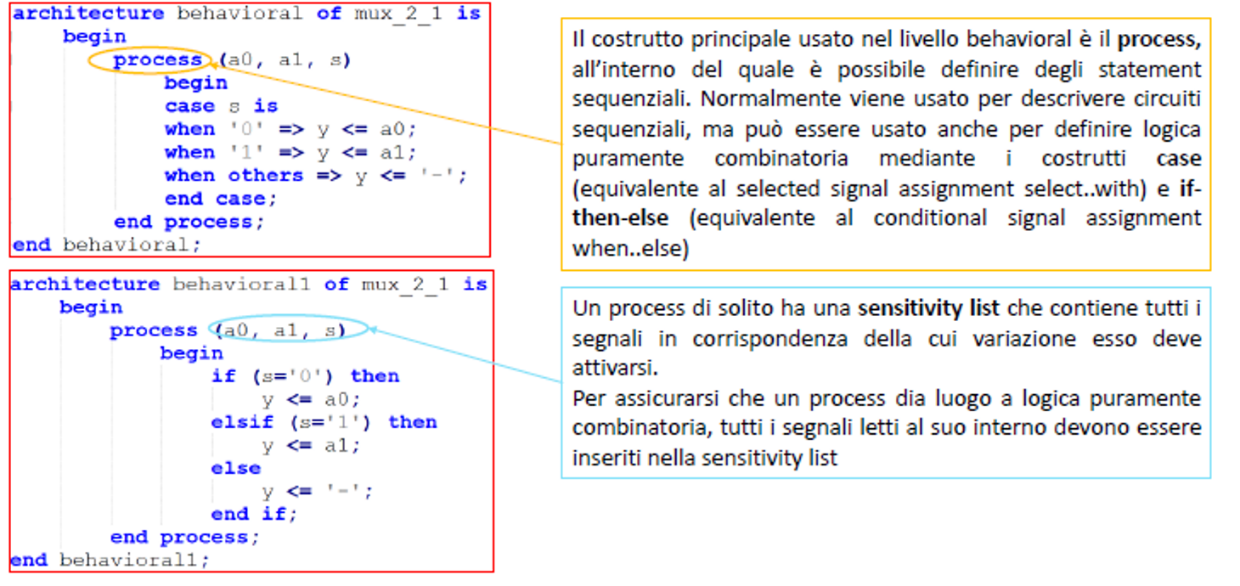
**Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamenteArchitecture: Dataflow**

 Il costrutto principale usato nel dataflow è **l’assegnazione concorrente (<=):** viene assegnato ad un segnale il valore risultante da una espressione logica contenente 0 o piú operatori logici.   
Tale costrutto puó essere implementato anche con una sintassi alternativa, ad esempio con il costrutto when-else. Attenzione in questo caso a considerare il caso **else ‘-‘** poiché vogliamo specificare cosa deve accadere per valori diversi da quelli attesi. Nel caso in cui non lo mettessimo si verrebbe a creare un latch che conserva il valore precedente, trasformando la nostra macchina in sequenziale. I casi **non sono mutuamente esclusivi** (come nel costrutto if dei linguaggi di programmazione): le condizioni vengono valutate **sequenzialmente** e la prima vera determina l’assegnazione.

**Architecture: Behavioral**

Rappresenta un livello di astrazione piú alto che consiste nel descrivere semplicemente cosa fa la macchina usando costrutti di alto livello. Il process è l’istruzione fondamentale delle descrizioni behavioral. Rappresenta una porzione di oggetto descritta dal punto di vista algoritmico.



Dentro al process vi sono **SOLO ISTRUZIONI SEQUENZIALI**. Per realizzare una macchina combinatoria TUTTI i segnali devono essere inseriti nella sensitivity list.

Quindi un process è uno statement concorrente che puó essere usato in un architecture body e reagire contemporaneamente agli altri statement concorrenti.

Un process è composto da parte dichiarativa, sensitivity list e process body. L’assegnazione di segnale <= è considerata sequenziale all’interno di un process.

VI SONO DUE TEMPLATES PER UN PROCESS:

1. Con sensitivity list -> non ci dovrá mai essere una wait alla fine
2. Senza sensitivity list -> ci vuole un istruzione wait prima di chiudere il process

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamente

nel caso di *sensitivity list* il **process viene attivato da un evento su un segnale che appartiene alla sensitivity list e sospeso quando raggiunge la fine del process;** nel caso di uso d*i istruzioni WAIT*, quando il **flusso di esecuzione incontra una WAIT**, il processo viene sospeso e la sua esecuzione è ripresa quando la condizione richiesta dall'istruzione WAIT è verificata;

I possibili tipi di uno statement WAIT sono: WAIT FOR waiting\_time WAIT ON waiting\_sensitivity\_list; WAIT UNTIL waiting\_condition; (usato ad esempio per protocolli asincroni) WAIT, WAIT ON sensitivity list;

* Un processo ha visibilità di tutti gli “oggetti” definiti nella sua architettura (sottotipi, costanti, segnali, procedure, functions,…): in altre parole **lo scope di un process è lo stesso della architettura che lo contiene;**

Le dichiarazioni della parte dichiarazione di un process sono invece locali al process; **L’unico modo che ha un processo per comunicare con l’esterno (e con altri process) è tramite i segnali** di cui ha visibilità e che assegna e legge;

All’interno del process è possibile usare un insieme completo di istruzioni sequenziali, in maniera del tutto simile a quanto è possibile fare con qualsiasi linguaggio di programmazione.

**TESTBENCH:**

Una volta che ho implementato il mio codice ho bisogno di fornire degli ingressi, degli stimoli in modo da effettuare correttamente la simulazione. A questo scopo ho bisogno di un file detto **testbench** in cui includiamo anche la libreria work in cui è finito il nostro progetto compilato: use work.all

Nel test bench è necessario inserire una entity vuota che rappresenta un po’la nostra scatola vuota vista dall’esterno e, nell’architecture andremo ad inserire i componenti da voler testare:

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamenteImmagine che contiene testo, Carattere, schermata, design

Descrizione generata automaticamenteCreo i segnali di input control e output in modo da simulare gli ingressi (Stimoleranno il componente).

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Dopo aver inserito una serie di valori per i miei segnali di input ed aver inserito istruzioni di wait for, inserisco un ultimo wait prima di terminare il process e quindi l’architecture. Da notare l’utilitá degli assert, una clausola che mi permette di fare una ipotesi sull’output e di mostrare un errore in caso in cui l’output sia differente da quello atteso.

Si inserisce l’entity work.nome\_entity(nome\_architettura). Dentro al port map faccio il wiring, a destra stiamo dicendo a quale filo lo stiamo collegando. Con stim\_proc definisco un process nel quale andró a definire quanto valgono i miei fili.

**GUARDIAMO NEL DETTAGLIO ENTITY E ARCHITECTURE:**

* **Entity**: Una dichiarazione di **entity** può essere usata per dichiarare gli “oggetti” (segnali di I/O ed eventualmente parametri generics) che potranno essere usati nell’implementazione dell’entity; una dichiarazione entity include la specifica del componente in termini dei suoi porti e puó includere la specificazione di costanti atte al controllo del comportamento dell’entity, cioè i GENERICS. Quando il modulo sará istanziato verrá stabilita la corrispondenza tra ports e segnali.

*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;

GENERICS: sono specificati nell’interfaccia di una entity e permettono di PARAMETRIZZARNE la descrizione. Il loro valore deve essere specificato in almeno uno dei modi:

1. come **valori di default nella dichiarazione di entity** (sempre raccomandato per assicurare che il componente sia sintetizzabile e simulabile);
2. all’atto della dichiarazione del component;
3. all’atto dell’istanziazione del component;

in caso di usi diversi il valore usato è quello a prioritá maggiore (3) si possono usare come generics solo quantitá costanti. 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;**

**PORTS:** ogni dichiarazione di ports è caratterizzato da un nome, tipo e modo. I modi previsti sono IN/OUT/INOUT/BUFFER. È consigliabile usare inout solo nella top level entity ma comunque evitiamo e NON usiamo il buffer ma preferiamo usare segnali di appoggio.

* **Architecture:** la specificazione di una architecture descrive una particolare implementazione di un modulo. Possono esserci diverse architecture per la stessa interfaccia. È composta da due parti separate:
  + una **parte dichiarativa**: in essa vengono **definiti gli oggetti** ( *segnali, costanti, component, …*)che saranno usati per costruire la descrizione;

Tutte **le dichiarazioni sono sono locali al modulo;** 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, …);

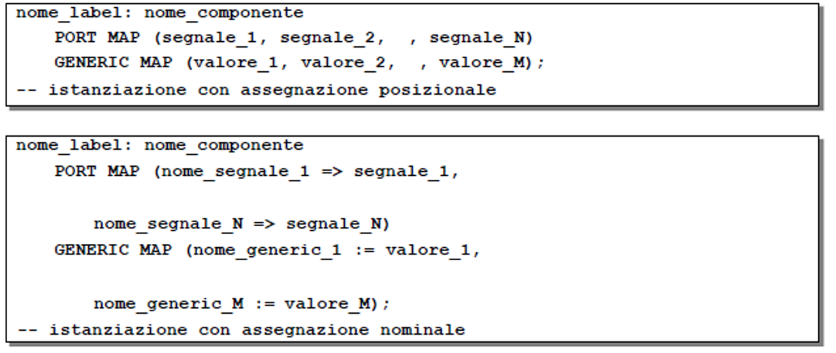
Dichiaro il componente con la parola chiave \*\*component\*\*

* Se il componente ha dei generic → utilizzo anche la parola chiave generic
* 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;

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

**BINDING:** 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.

* + 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.** É 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.

**Istanziazione:** L’istanziazione di un componente è uno statement concorrente, ovvero è sempre “**attivo**”. 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). 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);**

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;

Immagine che contiene testo, Carattere, schermata, bianco

Descrizione generata automaticamente**Assegnazione concorrente:** costrutto per eccellenza. È sempre attivo come l’istanziazione. Quando si verifica un evento su uno dei segnali a destra dell’assegnazione: viene calcolato il valore del segnale a sinitra + una transazione viene schedulata sul segnale di uscita secondo il timing dell’assegnazione;

Variazione del segnale a dx dell’= il circuito si attiva

Siano due architetture che realizzano la stessa funzione booleana ma con tempificazioni differenti, nella prima vi sará una propagazione delle modifiche agli ingressi attraverso vari livelli di logica provocando un ritardo accumulativo. Nella seconda qualsiasi cambiamento si rifletterá sull’output dopo un tempo unico (36ns).

Immagine che contiene testo, schermata

Descrizione generata automaticamente**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;

Nell’hardware il for può servirmi per lavorare nello spazio (non nel tempo come siamo abituati solitamente) → ad esempio se devo istanziare tante componenti uguali opportunamente connessi. Ho però un problema di wiring → **il wiring lo devo fare mentre istanzio i vari componenti.** Quando usi il costrutto generate per replicare un blocco logico, devi assicurarti che ciascuna istanza del blocco sia connessa correttamente agli altri blocchi o segnali nel tuo design.

**TIMING:**

**Inizializzazione di una simulazione VHDL:**

* Al tempo 0+0delta della simulazione viene eseguita una ***fase di inizializzazione*** in cui:
  + **ai segnali vengono assegnati i loro valori iniziali** (valori esplicitamente assegnati, oppure i valori di default per il loro tipo);
  + vengono **eseguiti** tutti i ***processi***, **finché tutti non raggiungono lo stato di sospensione;**
  + anche le **assegnazioni concorrenti <= sono eseguite**, essendo equivalenti a dei process, anche se poi **il loro risultato si avrà nei delta successivi;**
* Terminata la fase di inizializzazione, la simulazione è pilotata dagli eventi;

**Esecuzione di un process:**

* Ogni successiva esecuzione del processo è **innescata da eventi che vanno esplicitamente indicati**
* Un process esegue tutte le istruzioni sequenziali e poi le ripete ripartendo dall'inizio, il tutto ripetuto come in un loop infinito;

**VARIABILI:**

Le **variabili** possono essere usate per rappresentare:

* lo **stato** del sottosistema descritto dal process
* dei **risultati intermedi** dell’elaborazione all’interno di un process

Ogni variabile ha una sua **dichiarazione** (un tipo, un valore di inizializzazione), come altri oggetti in VHDL (segnali, costanti, …). É possibile **assegnare un valore** ad una variabile attraverso l’**operatore := (**è analogo a <= per i segnali); VARIABLE nome\_variabile : tipo;

**Un *signal* e una *variable* sono oggetti completamente diversi dal punto di vista della tempificazione** (anche se entrambi “portano” una informazione);

Una ***assegnazione di segnale <=*** comporta la **schedulazione** di un evento (al più di un delta dopo), **ma non ha mai l’effetto di una assegnazione immediata**

Una ***assegnazione di variabile :=* non comporta lo scheduling di nessun evento ed ha effetto di una assegnazione immediata**

**ASSEGNAZIONE DI UN SEGNALE (extra):**

**Immagine che contiene testo, schermata, software, Pagina Web

Descrizione generata automaticamente**

**APPENDICE: NOTE VHDL**

Il VHDL NON è case sentive

I commenti si effettuano con il ‘- -‘

Il file ha estensione .vhd

È buona norma scrivere in testa al file le righe:

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL; --di questa libreria utilizziamo la standard logic 1164 e tutte le entità

-- in questa libreria troviamo tutti i fili, e anche bus di fili (vettori di fili)

Una volta compilato un file, tutto finisce all’interno della libreria “work”.

Con STD\_LOGIC\_VECTOR (4 down to 0 ) intendo un vettore di fili 5 con standard std logic.

La label utt: è buona norma inserirla nella testbench ma serve soltanto a dare piú leggibilitá ed indentificare meglio il componente che stiamo testando.

**Numeri**: I numeri possono essere rappresentati come interi (integer) o come reali (real);

**Caratteri**: sono composti da caratteri ASCII racchiusi fra apici;

**Stringhe**: sono array di caratteri racchiusi fra doppi apici;

**Bit Strings**: per rappresentare le stringhe di bit si possono usare le notazioni binarie, ottali o esadecimali;

***Identificatori***: gli identificatori sono definiti dal programmatore per **identificare degli oggetti VHDL.** L’importante è non usare parole riservate, cioè le parole chiave tipo BEGIN, COMPONENT ecc.

Un **oggetto** è una entità VHDL che ha un suo ***nome***(un **identificatore** alfanumerico unico), un suo ***tipo***ed un suo ***valore***(che deve appartenere all'insieme dei valori leciti all’interno del tipo); Esistono tre classi di oggetti in VHDL **costanti (CONSTANT**), **variabili (VARIABLE**) e **segnali (SIGNAL**);

Una **costante** è un oggetto che è inizializzato con un certo valore all'atto della  sua definizione e che non può essere successivamente modificato

Immagine che contiene Carattere, testo, schermata, Elementi grafici

Descrizione generata automaticamente

Una **variabile** è un oggetto il cui valore può essere modificato dopo che stato  definito (**può** **essere usato solo** **nei corpi sequenziali**, *process*); Una variabile, appena viene definita, deve avere un suo valore. Questo valore iniziale può essere: assegnato esplicitamente alla varibile o implicitamente secondo regole fissate dallo standard

I **segnali** sono usati per **connettere dei moduli fra di loro** (sono leciti solo nei  corpi concorrenti, *architecture*);

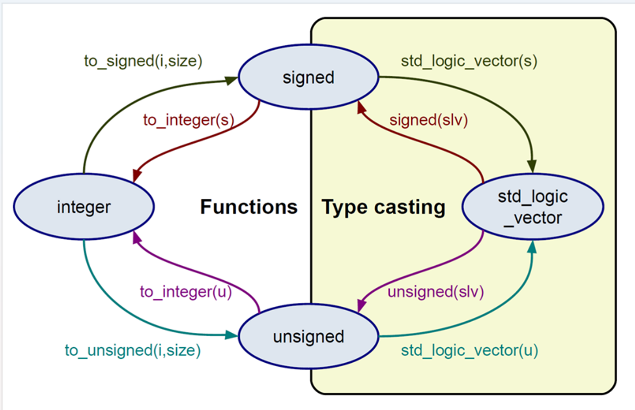
* Un tipo **intero** è un intervallo di numeri interi entro uno specifico range. In VHDL è predefinito un tipo intero chiamato ***integer***, definito nella libreria standard (std). L'intervallo di valori che può contenere dipende dall'implementazione, ma lo standard richiede che includa l'intervallo compreso fra –2147483647 e +2147483647 (codificabile con 32 bit). Tipo ***integer*** → **ci assegna 32 bit per la variabile,** dunque per ridurre il numero di bit con cui un integer viene implementato è possibile specificare il **range**:

Esempio:

SIGNAL myint : integer range -10 TO 15

a myint viene assegnato un numero di bit tale da poter rappresentare i numeri da -10 a 15 (5 bit)

I bit di un segnale di tipo *std\_logic\_vector* hanno tutti lo stesso “peso”; per dare un’interpretazione “numerica” ad un segnale fatto di diversi bit (in cui cioè ogni bit ha un peso in base alla sua posizione nel vettore) è possible usare i tipi ***signed*** e ***unsigned***. I tipi signed e unsigned sono definiti nel package ***numeric\_std*.** Per usarli bisogna **importare** non solo questo package, ma anche il package ***std\_logic\_1164*** (poichè *numeric\_std* usa *std\_logic\_1164*)

Il package **numeric\_std** definisce delle **funzioni di trasformazione** per convertire signed/unsigned in integer e viceversa:

* To\_integer
* To\_signed
* To\_unsigned

E’ inoltre possibile convertire signed/unsigned in std\_logic\_vector e viceversa tramite un **type casting.**

Un tipo enumerazione è un **insieme ordinato di identificatori di caratteri**, tutti **distinti** fra loro, anche se in tipi di enumerazione distinti lo stesso identificativo può essere usato più volte.

Gli **array** sono tipi di dati fondamentali nel VHDL. Un array in VHDL è una **collezione di elementi indicizzati tutti dello stesso tipo**. Inutile usare ARRAY quando servono tipi di STD\_LOGIC → ma conviene il STD\_LOGIC\_VECTOR. Un array può essere **constrained**, ovvero i limiti degli indici sono stabiliti, oppure **unconstrained** se i limiti sono stabiliti all'atto della dichiarazione di una variabile di quel tipo. Una **slice** di elementi contigui di un array monodimensionale può essere riferita usando un ***range* degli indici**: **a(8 TO 15)** è un vettore di 8 elementi parte dell'array a.

LA ROM:

Una ROM (Read-Only Memory) 16x4 è un tipo di memoria non volatile utilizzata per memorizzare dati che non cambiano, o cambiano raramente. Il termine "16x4" indica che questa ROM ha 16 posizioni di memoria, ognuna in grado di memorizzare 4 bit di dati. Vediamo come funziona:

***Struttura e Funzionamento***

1. **Dimensione**: La dimensione 16x4 significa che la ROM ha 16 indirizzi distinti (da 0 a 15) e ogni indirizzo contiene un dato di 4 bit.
2. **Indirizzi**: Per accedere a un dato specifico nella ROM, è necessario fornire un indirizzo. Poiché ci sono 16 indirizzi, è necessario un bus di indirizzi di 4 bit (perché 2^4 = 16) per selezionare uno di questi indirizzi.
3. **Dati**: Ogni indirizzo è associato a un dato di 4 bit. Questi dati sono pre-programmati e non possono essere cambiati (o possono essere cambiati solo mediante procedimenti speciali non previsti nell'uso normale del dispositivo).
4. **Lettura dei Dati**: Quando un indirizzo è fornito alla ROM, i 4 bit di dati corrispondenti a quell'indirizzo vengono letti e resi disponibili su un bus di uscita.

Una serie di locazioni

* Il singolo elementino è un bit → ma ad una granularità più alta ho la locazione
* La rom è un’array di locazioni
* Una locazione possiamo considerarla come uno std\_logic\_vector
* Devo avere un’indirizzo e lui mi deve dare il contenuto della locazione.
* Ogni locazione è grande 4 (3 downto 0)
* Se fornisci l'indirizzo 0011 (3 in decimale) alla tua ROM,
* Il valore recuperato e posto sull'uscita dout sarà 0011, che corrisponde al valore memorizzato all'indirizzo 3 nella tua definizione di ROM.

Una volta definito un tipo, è possibile assegnare un **nome alternativo al tipo** o ad una sua parte, usando la dichiarazione *alias*, in modo da potervi fare riferimento più comodamente. L’aliasing chiaramente è semplicemente un **renaming** e non definisce nessun oggetto.

L'uso di un *subtype* permette di usare un **sottoinsieme dei valori assunti da un  tipo:** Un subtype può essere anche usato per **rendere constrained una dichiarazione di  array unconstrained, specificando i limiti per gli indici.** Un **subtype** è **completamente compatibile con il suo tipo base.**

Un largo insieme di oggetti VHDL (es. tipi enumerativi e discreti, tipi array, segnali, etc.) dichiarati in una descrizione VHDL possono avere delle **informazioni addizionali, associando loro degli attributi.** In generale gli attributi sono **poco usati in VHDL** per descrivere dei sistemi. Alcuni attributi utili sono (‘event, ‘stable).

La sintassi è: nome segnale ‘nome attibuto

s1’EVENT -- fammi capire se si è verificato...

FUNZIONAMENTO DI VIVADO:

Prima della sintesi fisica, la ***simulazione logica*** viene utilizzata per verificare il comportamento del design. Dopo la simulazione si **assegnano i Constraint**, ovvero vincoli logici e fisici per guidare la sintesi e il place-and-route. Da un lato avremo dei ***Constraint fisici* →** l’FPGA vero dovrà avere dei pin connessi. Poi avremo constraint di tipo ***Timing*** *(****constraint temporali)***. Specificano i **requisiti di timing del design, come la frequenza del clock e la durata dell'impulso**. Il tool cercherà di ottimizzare la sintesi per rientrare nei vincoli che gli diamo noi. Se il tool riesce ad ottimizzare il design entro questi limiti, si raggiunge la Timing Closure. In caso contrario, il tool segnalerà che i constraint non possono essere soddisfatti. La parte di Logic Synthesis traduce il codice RTL in una "netlist", un insieme di oggetti logici che implementa le specifiche di alto livello (è un'elenco di porte logiche e le loro connessioni.). La netlist è composta da:

* **LUT (Look-Up Tables)**: Contengono le tabelle di verità e gestiscono la logica combinatoria del design.
* **Elementi Combinatori**: Come multiplexer e porte logiche standard (XOR, AND).
* **Flip-Flop**: Utilizzati per gestire la logica sequenziale del design.

Fasi:

Simulazione, sintesi (trasformazione di un design in una netlist), Implementation (ottimizzazione del design, scelta delle locazioni fisiche dell’FPGA dove piazzaregli oggetti fisici delle NetList, Instradamento cablando logicamente le connessioni tra questi componenti) . Dopodiché si genera il file bitstream il quale HA SEMPRE STESSA DIMENSIONE poiché dico che valori devono avere TUTTE le parti dell’FPGA, quindi le configuriamo tutte anche se vuote. Questi dati sono volatili, persi se viene meno l’alimentazione, per ovviare è possibile caricare il bitstream su una rom.

È preferibile non usare set o reset asincroni sui registri (posson portare ad un utilizzo subottimale delle risorse del dispositivo). È preferibile evitare di usare flipflop con set e reset simultanei. È preferibile descrivere sempre i segnali di clock, set e reset dei fliflop come attivi in alto. NON USARE MAI il reset asincrono: reset sincrono -> il segnale è allineato con il clock e non verrá attivato finché non arriva il segnal di clock. Reset asincrono-> il segnale di reset puó essere attivato in qualsiasi momento indipendentemente dal clock, i lche puó portare a imprevisti e problemi.

Vivado effettua l**'inferenza basandosi sulla descrizione fornita nel codice** HDL. Questo significa che sceglierà automaticamente il tipo di flip-flop in base a come viene descritto il comportamento desiderato nel codice. L'inferenza è una caratteristica potente dei moderni tool di sintesi che consente al progettista di **descrivere il comportamento desiderato a un alto livello di astrazione, lasciando al tool la scelta della migliore implementazione hardwar**e. Tuttavia, è importante seguire le best practice per garantire che l'inferenza funzioni come previsto e per evitare problemi nelle fasi di sintesi e di implementazione.

Per la creazione del progetto giá sai.

Attenzione a mettere la tb nelle simulation module.

Per vedere ancora piú a fondo le componenti, synthesis e poi open synthetized design.

**OVERFLOW:** **Nel caso di somma di due numeri in complemento a due:**

1. Se entrambi i numeri sono positivi (il bit più significativo, ovvero il bit di segno, è 0 per entrambi), e il risultato è negativo (il bit di segno del risultato è 1), allora si è verificato un overflow. Questo è perché il risultato ha superato il valore massimo rappresentabile per un numero positivo.
2. Se entrambi i numeri sono negativi (il bit più significativo è 1 per entrambi), e il risultato è positivo (il bit di segno del risultato è 0), allora si è verificato un overflow. Questo è perché il risultato è inferiore al valore minimo rappresentabile per un numero negativo.

Immagine che contiene testo, schermata, schermo, software

Descrizione generata automaticamenteImmagine che contiene testo, schermata, Carattere, software

Descrizione generata automaticamenteIn entrambi i casi, il cambiamento imprevisto nel bit di segno indica un overflow.