Lezione n. 4 ADC 16/03/20

Approcci di sviluppo VHDL

Abbiamo già parlato dei vari livelli di astrazione che possiamo adottare nel corso della progettazione.

Behavioral Description

- La **descrizione comportamentale** (*behavioral*) permette una descrizione *algoritmica o procedurale* (simile ad un programma software) del comportamento di un modulo;
- Questo livello permette *descrizioni a livello più astratto*, perché
  - non contiene riferimenti a come il sistema sarà implementato;
  - non si esplicitano dettagli architetturali o circuitali;
- Questo tipo di descrizione è *utile* in molte situazioni:
  - per scopi di documentazione,
  - per esprimere in forma univoca le specifiche di comportamento di un sistema,
  - per modellare componenti (complessi) di una libreria,
  - per descrivere elementi che servono solo in fase di simulazione (testbench),
  - per evitare eccessivi tempi di simulazione.

La descrizione comportamentale serve solo a descrivere funzionalmente il progetto. Non contiene informazione in merito ai dettagli necessari all'implementazione del programma.

Magari questo approccio è utile quando voglio concentrarmi sulla realizzazione di un componente senza soffermarmi su uno meno importante (o perlomeno che non devo implementare io).

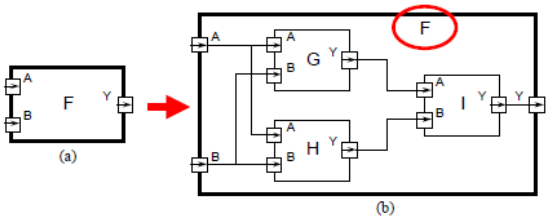
In altri casi, invece, si dà importanza alle interconnessioni tra componenti. Quando devo descrivere le connessioni con gli ingressi di un componente e le uscite di un altro, sto adottando un *approccio strutturale*. Gli strumenti necessari a collegare gli ingressi alle uscite sono detti segnali (connettori, fili).

Structural Description (1)

- La **descrizione strutturale** (*structural*) di un modulo specifica:
  - i componenti che costituiscono un modulo;
  - le interconnessioni tra i componenti.
- In altre parole, un modulo è descritto strutturalmente in termini dei moduli che lo compongono e di come i sotto-moduli sono collegati fra loro, con dei segnali (*signals*) che collegano i porti;
- In generale, ciascun modulo componente:
  - è una instanza di qualche entità precedentemente definita;
  - è caratterizzato dalla funzione logica svolta e dal suo timing (tempi di propagazione, di set-up, di hold, ...);
- Un modulo descritto strutturalmente ha una sua funzione logica e un suo timing, che sono funzioni delle caratteristiche corrispondenti dei moduli che lo compongono e del modo in cui sono interconnessi;
- A volte si fa riferimento ad una rappresentazione strutturale in forma grafica (e non testuale come in VHDL) con il termine di schematico (*schematic*).

Structural Description (2)

- Ad esempio, in figura (a) è riportata *l interfaccia del modulo F*;
  - Il modulo F ha 2 input A, B ed un output Y;
- In figura (b) c'è una *descrizione strutturale dell entità F*;
  - l'entità F è composta da instanze delle entità G, H ed I;
  - le entità G, H ed I in questo caso sono delle *black box* (possono avere una descrizione strutturale, comportamentale, … oppure possono essere componenti di libreria).
- In figura (b) sono distinguibili gli ingressi, le uscite e i nodi di interconnessione tra i moduli che compongono il modulo F.



L'interfaccia tra il componente F e gli altri è lo scatolotto. Come realizzo la connessione? Collegando i componenti più piccoli. I componenti più piccoli possono essere stati realizzati da me con un approccio strutturale o behavioural, oppure sono derivati da una libreria (e non li conosco bene)

Structural Description (3)

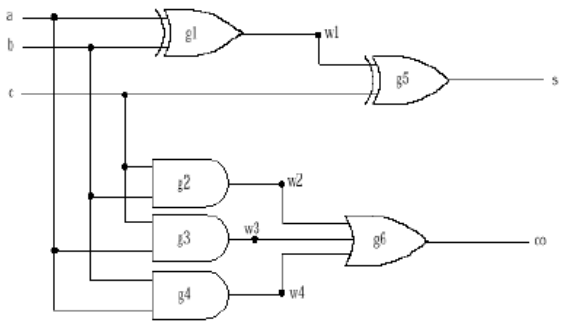
- Ogni modulo è una instanza di una entità (in un certo senso l'insieme entità-architettura è il "tipo" del modulo instanziato) e viene collegato facendo corrispondere i port di ciascuna instanza con dei segnali.
- In generale è difficile estrarre gli aspetti funzionali dalla lettura del codice di una descrizione strutturale;
- La simulazione è più onerosa in termini di tempo e memoria perché c'è la propagazione di molti eventi;

Il carico della simulazione è maggiore. Vedremo dopo perché in maniera più approfondita.

Descrizione Data-Flow

Data-flow Description (1)

- La descrizione di tipo **data-flow** rappresenta il sistema specificando come i segnali si propagano dai pin di ingresso a quelli di uscita;
- Una esempio di descrizione data-flow è la descrizione di una rete combinatoria in termini delle funzioni booleane delle sue uscite.
- Esempio:




```
s <= c NOR (a NOR b);
co <= (a AND b) OR (c AND a) OR (b AND a);
```

Un altro approccio alla progettazione è il *Data-flow*. Il livello di astrazione è contenuto. Vanno comunque definite le uscite come funzioni booleane degli ingressi (vedere figura).

"<=" è l'assegnazione che si usa per grandezze dette segnali (a, b, c, d). Segnali perché quando andremo a realizzare a

livello di hardware il componente, diventeranno qualcosa di fisico sul chip (silicio). Potrebbero diventare flip-flop, collegamenti ecc.

Data-flow Description (2)

- **Osservazione**: il circuito precedente potrebbe anche essere descritto a livello strutturale, usando come componenti elementari AND, OR, NOR.
- Ci sono importanti differenze fra una descrizione strutturale e una data-flow;
- La *descrizione strutturale*:
  - descrive il circuito come una netlist di porte fissate e disponibili in una libreria;
  - avendo disponibili una implementazione delle porte necessarie (and, or, nor) si potrebbe realizzare il circuito esattamente secondo lo schematico;
- La *descrizione data-flow*:
  - descrive il legame ingresso-uscita (la tabella di verità) senza far riferimento ad una sua implementazione concreta (e quindi non viene fatto alcun riferimento ad una libreria di porte);
  - si potrebbe realizzare con and-or-not, oppure con sole nand, o sole nor.
- In generale è necessario un passo di sintesi per passare da una descrizione data-flow ad una descrizione strutturale con componenti di libreria (descrizione gate-level) che possa essere implementata.

Se voglio realizzare nel dettaglio un componente, magari con quello uso un approccio strutturale mentre con tutti gli altri uso un approccio Data-flow.

Sono dei linguaggi supportati da una serie di costrutti che descriveremo successivamente.

Gate Level Description

- Una descrizione VHDL gate level contiene:
  - una lista di componenti (gates) che sono usati in un design,
  - una lista delle effettive instanziazioni dei componenti e delle loro interconnessioni.

Da un punto di vista logico, come progetto la macchina, indipendente dal VHDL?

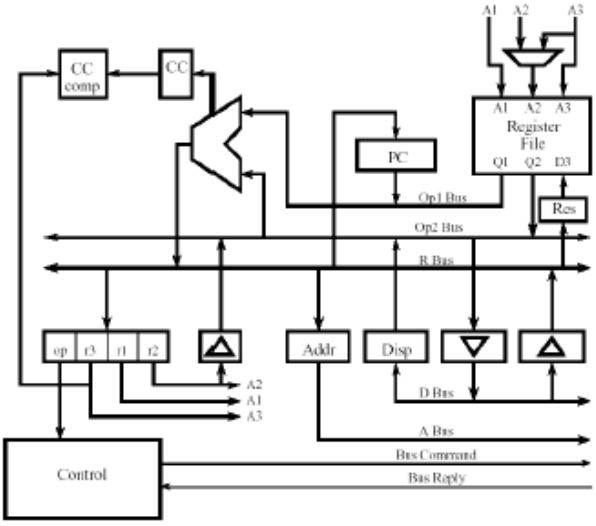
- Tutte le porte usate fanno riferimento ad una tecnologia *target* in cui sono accuratamente descritte ulteriori informazioni sulle porte (area, propagation delay, capacità di ingresso, capacità di pilotaggio, ...).

- Una descrizione gate level è la descrizione con il minimo livello di astrazione in VHDL, perché fornisce indicazioni estremamente dettagliate sul circuito (massima frequenza di clock, area occupata, potenza dissipata, …);
- Essendo una descrizione di tipo strutturale:
  - E' in generale difficile estrarre gli aspetti funzionali dalla lettura del codice;
  - La simulazione è più onerosa in termini di tempo e memoria perché c'è la propagazione di molti eventi;

Posso usare diversi livelli di astrazione. Posso immaginare la macchina aritmetica come un insieme di porte logiche connesse tra di loro. Questo livello è detto Gate level. È diciamo il livello relativamente più basso, perché sotto di esso ci sono solo i transistors.

RT Level Description

- **RTL = Register Transfer Level**
- A questo livello di astrazione il design è diviso in 2 parti:
  - una parte di *logica combinatoria* (che calcola il prossimo stato e le uscite a partire dallo stato corrente e dagli input)
  - *elementi di memoria* (gli elementi di memoria sono controllati dai clock di sistemi).
- E' una descrizione del sistema in termini di memorizzazione, di elaborazione e di spostamenti/ trasformazione dei dati fra i registri.
- In questo tipo di descrizione è contenuta l'informazione sulla struttura del sistema perché sono distinti i componenti che memorizzano i dati da quelli che li elaborano.
- Rispettando un insieme di regole è una descrizione sintetizzabile.



Se il progetto diventa ancora più complesso, mi avvicino ad un livello più astratto, immediatamente superiore a quello delle porte logiche, detto *RT level*. Vediamo le macchine più grandi (ALU, blocchi di memoria ecc.). Esempio: vado a progettare ad alto livello dei componenti. Se però uno dei componenti è troppo lento o troppo veloce lo devo manipolare. Se voglio farlo devo trascendere di qualche livello di progettazione. Se sono, dopo tutte le modifiche, soddisfatti i requisiti non funzionali, torno al livello di astrazione più alto.

Tecniche di simulazione dell Hw

- La simulazione dell'hardware è imprescindibile nel ciclo di sviluppo dei sistemi digitali;
- Per circuiti complessi è fondamentale che sia efficiente;

- **Simulazione a tempo quantizzato**:
  - Il tempo di simulazione procede per incrementi di ampiezza prefissata (passo di quantizzazione);
  - E' incapace di adattarsi alla dinamica del sistema;
- **Simulazione ad eventi**:
  - Il tempo di simulazione procede per incrementi di ampiezza variabile (solo quando ci sono variazioni in un nodo si calcola la propagazione di queste variazioni);
  - Permette di seguire la dinamica del sistema;
- **Varie tecniche di ottimizzazione**:
  - Ad es. determinazione dello scope di una variazione (per entrambe le tecniche);
  - Variazione del passo di quantizzazione per le tecniche a tempo quantizzato (ad es. Spice);

Come si simula la propagazione dei segnali? (il file .vcd). Esistono diverse tecniche di simulazione.

La prima è quella a tempo quantizzato. Consiste nel dividere il tempo in intervalli piccoli che devono soddisfare il *teorema di campionamento* (in modo da non perdere alcuna variazione del segnale).

L'estensione dell'intervallo deve dipendere quindi dalla frequenza dei segnali.

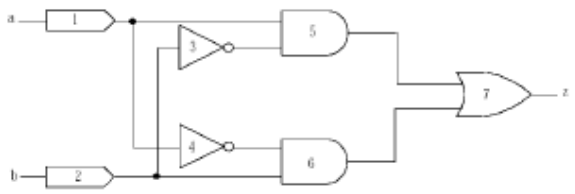
A questo punto si prende l'ingresso, si legge il valore ad un certo istante e si calcolano tutti i valori prodotti dalle porte logiche del progetto in quell'istante (es. istante t_0=0). Dopodichè si passa all'analisi di tutti i segnali di uscita e di ingresso all'istante $t_0 + \Delta$ (dove il Δ è l'intervallo di campionamento che abbiamo scelto). Ovviamente può capitare che un segnale vari la frequenza nel corso della sua vita e per questo i dati registrati potrebbero essere ridondanti (sistema inefficiente). La dinamica dei segnali quindi può cambiare.

Il secondo approccio è detto simulazione ad eventi: l'intervallo a questo punto è variabile e cambia a seconda delle variazioni del segnale (appena viene incontrata una variazione, quindi si verifica un evento, si registrano i valori di tutti i segnali, ossia i dati). Si dice che viene individuato il fronte del clock. Ovviamente mi baserò sul segnale di clock, dal momento che condiziona tutti gli altri.

Simulazione a tempo quantizzato

Simulazione a tempo quantizzato (1)

- Esempio nel caso di *0-delay* (non si tiene conto della tempificazione del circuito ma solo del suo comportamento) :



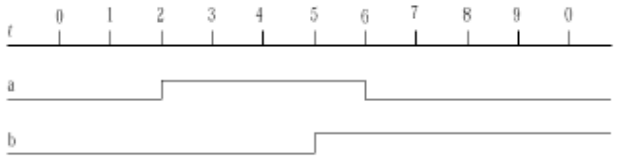
Gate	Function	Input1	Input2	Output Value
1	Buffer	A	-	0
2	Buffer	B	-	0
3	Not	2	-	1
4	Not	1	-	1
5	And	3	1	0
6	And	4	2	0
7	Or	5	6	0

- Lo *stato del circuito* è rappresentato dalla tensione ad ogni nodo del circuito;
- Il *circuito* è rappresentato come una netlist in forma tabulare;
  - Gli ingressi delle gate sono rappresentati dagli identificativi delle porte;
  - Lo stato è rappresentato da un'altra colonna (output value);
  - L'ordine delle righe rappresenta un modo di ispezionare il grafo del circuito in modo da rispettare la dipendenza funzionale delle porte;

Primo approccio: Viene scelto un quanto di tempo sufficientemente piccolo. Prende tutti i segnali della macchina e ne calcola i valori ad ogni quanto.

Simulazione a tempo quantizzato (2)

- Il simulatore si attiva ad intervalli di tempo fissati T_{SAMPLE} e aggiorna lo stato del circuito (= i valori della tabella);
- Dopo ogni T_{SAMPLE} → aggiornamento dello "stato" del circuito:
  - si campionano gli ingressi;
  - si calcola il valore del potenziale di ogni nodo secondo le gate e le loro interconnessioni, ispezionando la tabella;



Gate	Function	Input1	Input2	T = 0	T = 2	T = 5
1	Buffer	A	-	0	1	1
2	Buffer	B	-	0	0	1
3	Not	2	-	1	1	0
4	Not	1	-	1	0	0
5	And	3	1	0	1	0
6	And	4	2	0	0	0
7	Or	5	6	0	1	0

Simulazione a tempo quantizzato (3)

- **Principio di funzionamento:**
  - Il tempo di simulazione procede per incrementi di ampiezza prefissata (*passo di quantizzazione*);
- **Vantaggi:**
  - E' (abbastanza) semplice da implementare nel caso di assenza di ritardi (simulazione puramente funzionale);
  - Può simulare anche variazioni continue (purché discretizzate) dei potenziali (ad es. Spice);
- **Svantaggi:**
  - Questa tecnica è incapace di adattarsi alla dinamica del sistema;
  - E' difficile gestire i tempi di propagazione delle porte;
  - Il passo di quantizzazione dovrebbe essere minore delle costanti di tempo del circuito → cresce l'onere computazionale delle simulazioni;
- **Possibili ottimizzazioni:**
  - Determinazione dello scope di una variazione del potenziale di un nodo;
  - Il passo di quantizzazione può essere variabile e adattarsi all'evoluzione del circuito;

Il vantaggio è che è semplice da attuare, ma non si adatta alla dinamica del sistema.

NB. Per dinamica del sistema si intende la dinamica del segnale. Le porte hanno una propria dinamica (possono filtrare i segnali poiché non sono ideali, hanno un ritardo interno). Inoltre, c'è anche da considerare il ritardo di propagazione del segnale nelle porte, che però non determina la dinamica del segnale di uscita dalla porta. In ogni caso si tratta di un segnale dipendente dal clock (ingresso) e quindi l'unica dinamica rilevante è quella del clock. Le altre sono derivate in un certo senso.

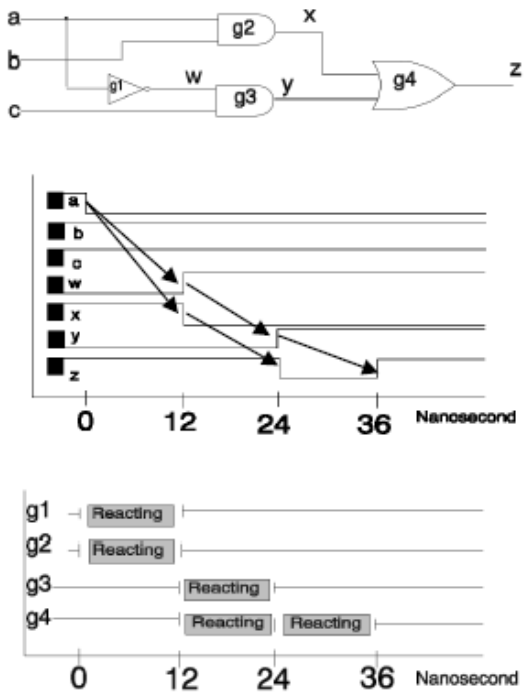
Simulazione ad eventi

Simulazione ad eventi (1)

- **Principio di funzionamento:**
  - Valuta il circuito solo quando si verifica un "evento" ovvero un cambiamento del potenziale di un nodo;
  - Se non ci sono eventi, la rete non evolve ma è stabile;
- **Vantaggi:**
  - Permette una simulazione veloce dei circuiti digitali → i simulatori VHDL sono ad eventi (*event driven*);
- **Svantaggi:**
  - È più difficile da implementare;

Simulazione ad eventi (2)

- Esempio del circuito riportato a fianco:
  - Si suppone che tutte le porte abbiano lo stesso ritardo (12ns);
  - Sono mostrate le *waveforms* (forme d'onda) ai nodi del circuito;
- Le variazioni di un nodo si propagano lungo il circuito;
- Ad es. la commutazione sull' ingresso a: 1→0 @ 0ns provoca, tramite la porta NOT g1, una commutazione su w: 0→1 @ 12ns, la quale provoca una commutazione 0→1 @ 24ns sul segnale y…

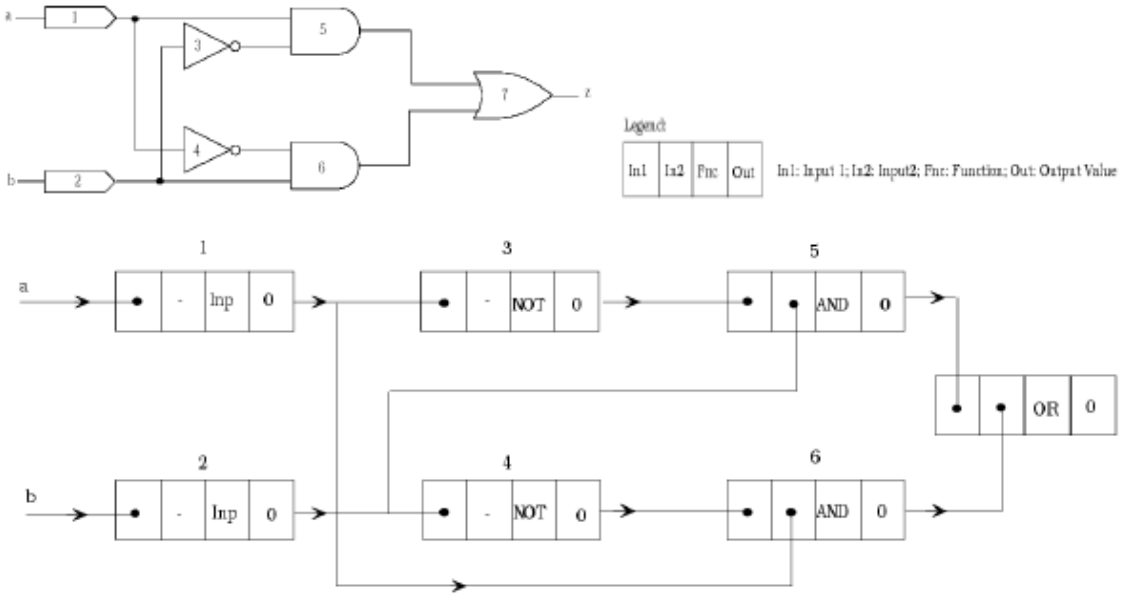


Qui analizzo quando cambiano gli ingressi. In questo caso si analizza un ingresso per volta: nell'immagine ci si concentra su **a**. Visto che **a** determina sia **x** che **y**, e le porte subiscono ritardi di propagazione, si attende il ritardo di propagazione della porta prima di registrare i valori. La stessa cosa si fa per **z**. Cioè, prima di registrarne il valore si aspetta il ritardo di propagazione della porta **g4** (considerando anche quello delle porte a monte) (Nell'esempio il ritardo di ciascuna porta è 12ms.). Non ci siamo proprio posti il problema del quanto di tempo. Traccio gli eventi e le dipendenze tra segnali. Quando si verifica una variazione su un ingresso, vado a calcolare solo dopo i ritardi ed i segnali che dipendono da quell'ingresso. Dopo la prima schedulazione, se ho altre variazioni di **a**, devo vedere il prossimo evento che si trova nella coda degli eventi.

Non spreco in registrazione dati, ma spreco nella definizione della dipendenza tra porte (e quindi tra segnali). Devo tener conto anche di code di eventi che nel tempo si succedono dinamicamente (la rete genera eventi nel tempo). La lista quindi va sempre aggiornata.

Simulazione ad eventi (3)

- La rappresentazione del circuito per la simulazione ad eventi si basa su una linked list;



IL MODELLO DI TEMPIFICAZIONE VHDL

Il modello di tempificazione VHDL (1)

- Abbiamo detto che il VHDL permette di rappresentare il fatto che diversi oggetti (moduli, assegnazioni di segnali, process…) evolvano contemporaneamente ("concorrentemente");
- Le assegnazioni e le instanziazioni dei moduli sono concorrenti ed è del tutto indifferente l'ordine con cui vengono specificate.

```
ENTITY example IS
        PORT (a, b, c : IN BIT ; z : OUT BIT);
END example;
--
ARCHITECTURE concurrent OF example IS
        SIGNAL w, x, y : BIT;
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;
```

Il VHDL usa una simulazione ad eventi. Nell'esempio è realizzata proprio la rete di due figure fa (Simulazione eventi (2)).

La **Entity** è l'interfaccia, il prototipo del metodo. Abbiamo degli ingressi di tipo bit e un'uscita sempre di tipo bit. La connessione tra gli ingressi e uscite è descritta nell'architettura ed usa dei segnali intermedi. W, x, y sono dei segnali che verranno realizzati in hardware. Sono anch'essi di tipo bit e poi tra il **begin** ed **end** c'è il rapporto ingresso-uscita. Dopo che viene specificata la porta cui è collegato il segnale, si specifica anche il ritardo (12 ns).

NB: il vhdl è *case insensitive* (scrivo maiuscolo o minuscolo, non cambia niente).

In questo schema le istruzioni sono *concorrenti* (non sequenziali, non è un ciclo Von Neumann). Quindi le assegnazioni di x e z possono essere poste anche all'inizio. **L'ordine non è rilevante.**

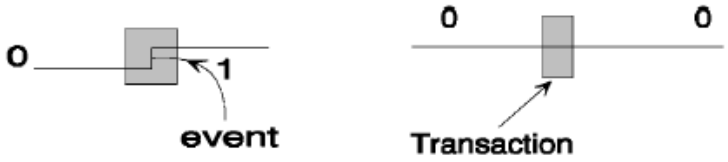
Il modello di tempificazione VHDL (2)

- Una *assegnazione* su un segnale comporta l'aggiunta di una *transazione* (transaction) sul segnale target:

```
w <= NOT a;
```

```
x <= a AND b AFTER 12 NS;
```

- Una transazione ha un valore e una componente di tempo (value, time)
  - *value* rappresenta il valore che verrà assegnato al segnale e viene calcolato nell'istante in cui si genera la transazione;
  - *time* rappresenta dopo quanto tempo il valore value sarà assegnato al segnale.
- Non sempre una assegnazione comporta una variazione del valore di un segnale. Si parla di *evento* quando questo avviene, altrimenti di semplice transazione.



NB: a, b c e **d** non sono definiti come *signal* perché già definirli **Input** e **output** basta (*gli ingressi e le uscite sono per forza segnali*).

All'interno della macchina i **signal** possono essere ricalcolati e quindi i loro valori possono cambiare o rimanere gli stessi. Se cambiano si dice che c'è stato un evento sul segnale. Se non cambia c'è stata una transazione.

Il modello di tempificazione VHDL (3)

- Ad esempio se a = 1 → 0 @ 0s, l'assegnazione riportata

```
w <= NOT a AFTER 12 NS;
```

genera una transazione di valore (1, 12ns).

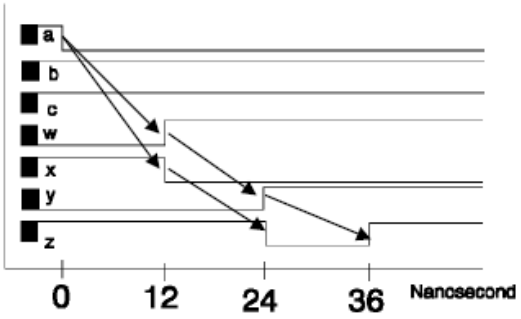
- Dopo 12ns dall'istante 0, l'evento 0 → 1 avverrà sul segnale w.

Il modello di tempificazione VHDL (4)

- Si consideri la descrizione data-flow:

```
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;
```

- La tempificazione dei segnali a causa di una commutazione sul segnale a 1 → 0 @ 0 (b è sempre alto) è riportata in figura:



Il calcolo dei segnali avviene sempre prima al tempo 0, poi si considera tutto il processo legato agli eventi.

DELTA CYCLES

Delta cycles (1)

- Se si assegna un valore ad un segnale senza specificare il ritardo si assume che il ritardo sia 0:

```
w <= NOT a;
```

è equivalente a:

```
w <= NOT a AFTER 0ns;
```

- La transazione è "schedulata" (*scheduled*) per lo stesso istante dell'assegnazione, ma concretamente il simulatore VHDL la eseguirà nel ciclo di simulazione successivo (→ il *delta cycle* successivo);
- Il tempo della simulazione è del tipo: $x_1T+x_2\delta$ dove T è un tempo (ns, ms, s) e delta (δ) è un ciclo di simulazione che non corrisponde a tempo reale;
- In altre parole una transazione con tempo zero scade un "delta" dopo che è stata posta sul segnale;

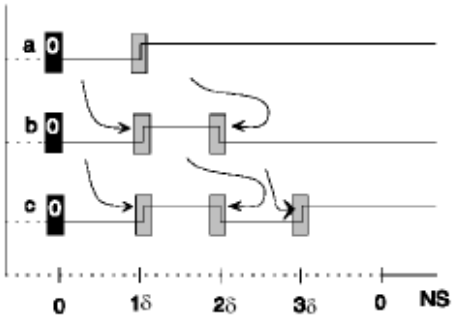
- **Osservazione:**
  - Delta è usato per lo scheduling: un miliardo di delta non compongono un femtosecondo!

Tuttavia, non è detto che in un'espressione diamo un ritardo. Possiamo anche indicare che tutte le variazioni avvengono senza ritardi (caso ideale). Tuttavia, essendo le istruzioni concorrenti, se programmo delle istruzioni senza ritardo e altre con ritardo, il vhdl introduce in ritardo fittizio (perché comunque se si tratta di un processore è impossibile che non ci sia ritardo) che viene detto **delta cycle**. Viene utilizzato per creare la forma d'onda ma alla fine viene posto pari a 0.

Delta cycles (2)

- Vediamo un esempio di assegnazione senza ritardo:

```
ARCHITECTURE concurrent OF es_delta IS
        SIGNAL a, b, c : BIT := '0';
BEGIN
        a <= '1';
        b <= NOT a;
        c <= NOT b;
END concurrent;
```



- All'inizio i segnali sono tutti 0 (*inizializzazione*);
- Ma (a,b,c) = (0,0,0) è uno stato "instabile" (incompatibile con il circuito) e la rete comincia ad *andare a regime* → (0,1,0);
- La transizione di a 0→1 @ 0ns+1δ provoca l'evento 0→1 dopo 1 delta e la variazione si propaga nella rete.

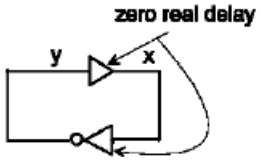
```
                ps delta a b c
=======================
                0     +0 0 0 0
                0     +1 1 1 1
                0     +2 1 0 0
                0     +3 1 0 1
```

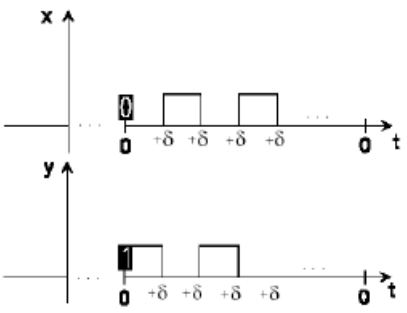
Nelle istruzioni sopra non ci sono riardi. Tutti i risultati dovrebbero essere istantanei. Il vhdl se vede che tutti i segnali sono inizializzati tutti a 0, già vede cosa dovranno diventare questi segnali, calcola il segnale "futuro" e schedula l'istruzione dopo un **delta-cycle**. Visto che **b** dipende da **a**, schedula dopo due delta-cycle l'istruzione legata a **b**. Visto che **c** dipende da **b**, vhdl schedula dopo 3 delta-cycle l'istruzione legata a **c**.

Delta cycles (3)

- Un "paradosso":

```
ARCHITECTURE concurrent OF timing_demo IS
        SIGNAL x, y : BIT := '0';
BEGIN
    x <= y;
    y <= NOT x;
END concurrent;
```





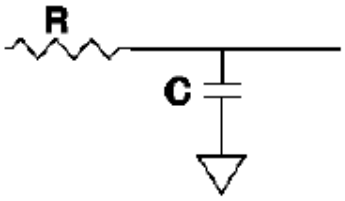
- I segnali x ed y oscillano indefinitamente, senza che il tempo reale della simulazione avanzi

Il simulatore crea al suo interno lo schema nella figura nella slide DC(2) a destra. In realtà però, appena apriremo il simulatore di forma d'onda, non vedremo nessun ritardo. Il vhdl ragiona tenendo conto delle dipendenze tra segnali e schedulando l'assegnazione dei valori in altri istanti nel futuro. Se noi abbiamo inserito un ritardo, verrà inserito + un delta cycle. Tutti i delta cycle vengono posti a zero prima di generare l'output, quindi tutte queste variazioni per noi rimangono invisibili.

Questo è un altro esempio. Anche qui non è stato inserito alcun ritardo ed abbiamo due segnali x,y.x diventa y dopo un *delta-cycle* e poi **y** diventa **not x** dopo due delta cycle e **x** ritorna uguale a **y** dopo 3 delta cycle. Nonostante vengano fatti tutti questi calcoli, il tempo reale non avanza mai (poiché delta cycle vale zero). Quindi avremo un'oscillazione continua del segnale, e se provassimo a simulare in un progetto una situazione del genere, il simulatore andrebbe in errore (*simulation stopped by delta*); il messaggio di errore sta ad indicare che nonostante faccia molti calcoli il tempo non cambia (ciclo infinito). Se, invece, inseriamo il ritardo reale, la simulazione continua all'infinto finché non indichiamo al simulatore di terminare la simulazione.

Inertial e transport delays (1)

- Spesso, a causa dei parametri parassiti, le interconnessioni dei circuiti reali possono essere schematizzate come delle reti RC;
- La presenza di queste reti filtra gli impulsi troppo brevi rispetto alla costante di tempo RC della rete;



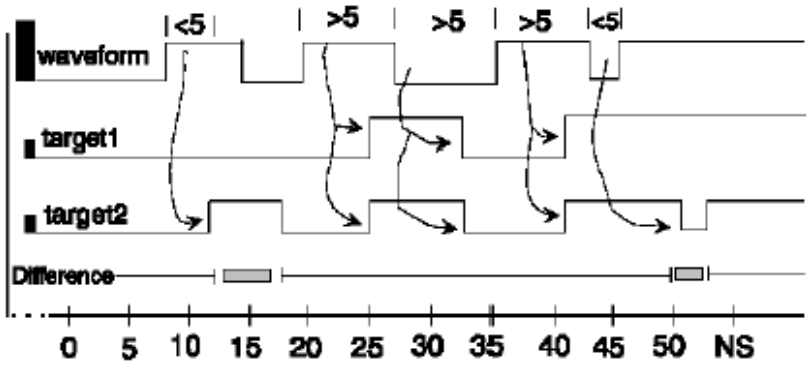
- Questo fenomeno può essere modellato in VHDL con dei ritardi "inerziali" (*inertial delays*):
  - I ritardi **inerziali** rappresentano un fenomeno di ritardo che però non propaga tutti gli impulsi (0→1→0, 1→0→1), ma solo quelli che durano più di un certa soglia;
  - I ritardi **transport** invece propagano sempre le transazioni indipendentemente dalla distanza temporale fra due transazioni successive;
- Tipicamente i ritardi inertial sono poco usati in VHDL;
- Un ritardo senza la parola chiave INERTIAL è di tipo TRANSPORT.

Una cosa è il ritardo, un'altra la dinamica del sistema della porta, ossia quella che un circuito è in grado di seguire. In ogni circuito ci sono dei parametri parassiti e quindi delle capacità e resistenze. Ciò comporta che il segnale, per propagarsi da un punto all'altro (lungo qualsiasi collegamento), impieghi molto tempo perché deve caricare il condensatore ecc. Se la capacità parassita è sufficientemente grande e il segnale dura poco, il segnale non ce la fa a caricare il condensatore e il segnale va a zero e la capacità si scarica di nuovo. Quindi non raggiungeremo mai il valore alto dall'altra parte del circuito. Tutti i componenti reali non è detto che riescano a seguire i segnali d'ingresso se variano con una frequenza troppo alta. Pertanto, quando andiamo a progettare un componente, dovremmo tener conto di questi parametri fisici. Il vhdl ci permette di inserire un ritardo nel componente che può essere o ideale (come quelli che abbiamo fatto fino ad ora, ossia che il segnale viene rilevato dopo un certo ritardo, indipendentemente dalla sua frequenza) e queste assegnazioni sono detti transport e sono di default di vhdl. Il ritardo è inerziale quando l'ingresso rimane costante per un tempo più piccolo del ritardo, l'assegnazione non viene proprio calcolata.

Inertial e transport delays (2)

- Esempio di ritardo inertial e transport:

```
ARCHITECTURE delay OF example IS
   SIGNAL target1, target2, waveform : BIT;
BEGIN
   waveform <=                                  -- forma d onda in figura
   target1 <= INERTIAL waveform AFTER 5 NS;  -- inertial delay
   target2 <= TRANSPORT waveform AFTER 5 NS; -- transport  delay
END delay;
```



Esempio: Nel primo caso il ritardo è inerziale e nel secondo è transport. Se la waveform varia e la sua durata è minore di 5 secondi, allora target 2 la vede con un ritardo di 5 sec e target 1 la perde (è come se fosse un filtro).

Interazioni fra transazioni

- In generale possono essere applicati, in successione, al medesimo segnale diverse transazioni;
- In questo caso la regola di risoluzione è la seguente:

	TRANSPORT	INERTIAL
New Transaction is BEFORE Already Existing	Overwrite existing transaction.	Overwrite existing transaction.
New Transaction is AFTER Already Existing	Append the new transaction to the driver.	Overwrite existing if different values, otherwise keep both.

- Sono fenomeni che si incontrano raramente nel VHDL;
- Navabi fa una trattazione estesa di questi casi di interazione fra transazioni.

Noi svilupperemo solo componenti di tipo transport.

DATAFLOW

Descrizioni VHDL Dataflow

- In questo capitolo vedremo come la struttura di un sistema digitale è descritto in VHDL utilizzando una descrizione di tipo *data-flow*.
- **Outline:**
  - espressioni ed operatori
  - assegnazione concorrente
  - assegnazione condizionata
  - assegnazione concorrente
  - esempi vari

Convenzioni Tipografiche

- In tutte le slide si useranno le seguenti convenzioni tipografiche:
- `ENTITY`, `ARCHITECTURE`, `FOR`, ... 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
```

Espressioni e operatori (1)

- Ad un **segnale,** ad una **variabile** o ad una **costante** può essere associato il risultato di una espressione che coinvolge diversi altri valori (non si possono mischiare segnali e variabili!)
- Le **espressioni** in VHDL sono del tutto simili alle espressioni in altri linguaggi di programmazione:
  - Una espressione combina degli operandi (valori di segnali, costanti, variabili), tramite degli operatori;
  - Di seguito sono riportati gli operatori predefiniti dal VHDL, in ordine di priorità descrescente (le parentesi permettono di alterare l'ordine di valutazione):

Highest precedence:	**	**abs**	**not**			
	*	/	**mod**	**rem**		
	+ (sign)	– (sign)				
	+	–	&			
	=	/=	<	<=	>	>=
Lowest precedence:	**and**	**or**	**nand**	**nor**	**xor**	

Espressioni e operatori (2)

- Gli **operatori relazionali** sono:

  =, /= (diverso), <, <=, > e >=

  e devono essere applicati ad elementi dello stesso tipo, producendo un boolean;
- Gli **operatori aritmetici**:

  ** (elevamento a potenza), abs, *, /, mod, rem (resto), +, -

  possono essere applicati solo ai tipi numerici (integer, floating point ed i tipi physical), dando un valore dello stesso tipo (oppure overflow e underflow) secondo il normale significato;
- Gli **operatori logici** (and, or, nand, nor, xor e not) operano su oggetti di tipo bit, std_logic e boolean, ma anche sugli oggetti di tipo bit_vector e std_logic_vector (word gate).
- L'**operatore &** permette di concatenare diversi oggetti ottenendo array di oggetti;

- In generale è possibile per l'utente anche definire nuovi operatori e fare overloading di altri operatori;

Assegnazione concorrente (1)

- Le assegnazioni di segnale **concorrenti** appaiono all'interno del corpo di una architecture, mentre le assegnazioni nel corpo di un process sono chiamate assegnazioni **sequenziali**;
- Le assegnazioni concorrenti sono attivate ogni volta che c'è una variazione in uno dei segnali da cui dipende il segnale assegnato;

```
S <= NOT ((A XOR B) AND C);
```

- In VHDL
  - tutte le assegnazioni di segnali (di qualsiasi tipo: incondizionate, condizionate, selezionate),
  - tutte le instanziazioni di blocchi,
  - tutti i process,

  presenti all'interno del corpo di una *architecture*, sono concorrenti fra loro;

Utilizzando un approccio dataflow, tra il begin e l'end troviamo una serie di istruzioni concorrenti che solitamente rappresentano una funzione logica. Le grandezze coinvolte sono segnali e l'operatore di assegnazione è <=.

Assegnazione concorrente (2)

- Il **segnale target** di una assegnazione di qualsiasi tipo può essere un segnale semplice, un segnale indicizzato, e.g. a(5), una slice a (5 downto 0), o un segnale aggregate a(5) & a(2);
- Il simbolo di assegnazione fra segnali è <=

```
S <= A XOR B;
S(0) <= A(0) XOR B(0);
S(3 downto 0) <= A(3 downto 0) XOR B(3 downto 0);
S(1) & S(0) <= A(1 downto 0) XOR B(1 downto 0);
```

- In VHDL ci sono **3 tipi di assegnazioni** concorrenti fra segnali:
  1. assegnazione incondizionata – la waveform che viene assegnata al segnale è fissa;
  2. assegnazione condizionata – la waveform assegnata è determinata sulla base di un costrutto *if-then-else*;
  3. assegnazione selezionata – la waveform assegnata è determinata sulla base di un costrutto di tipo *case*;

Tutte quelle che vediamo sono assegnazioni incondizionate. Nella seconda riga troviamo dei vettori di bit. Una differenza con il c è che quando definiamo un vettore in c, l'indice degli elementi va da 0 a n-1. Nel vhdl l'indice può variare tra 0 e n o n e 0, oppure può variare da qualsiasi indice all'altro (*down to*). Nella 3 riga troviamo un esempio. Troviamo un'assegnazione su 3 bit. Dobbiamo immaginare questi array come dei cavi, connettori. Invece **l'operatore di & è di concatenazione** (non è and). Indica che debbano essere connessi come un vettore di due elementi e assegna il risultato sul vettore di due elementi. (due fili di due cavi diversi ed assegnarlo a due pin di un terzo cavo). Potremmo anche usare delle assegnazioni condizionate e selezionate.

Assegnazione incondizionata

- Un'espressione che coinvolge diverse forme d'onda è semplicemente assegnata ad un segnale:

```
ARCHITECTURE dataflow OF HalfAdder IS
BEGIN
  Sum <= A XOR B;
  Carry <= A AND B;
END ARCHITECTURE dataflow;
```

Assegnazione condizionata (1)

- Una **assegnazione condizionata** è del tipo:

```
segnale <=      waveform_1 WHEN condizione_1 ELSE
                waveform_2 WHEN condizione_2 ELSE

                waveform_(n-1) WHEN condizione_(n-2) ELSE
                waveform_n;
```

- Una assegnazione condizionata permette di scegliere la waveform da assegnare (fra *waveform_1* … *waveform_n*) ad un segnale target (*segnale*), a seconda di quale condizione booleana (*condizione_1* … *condizione_n*) sia verificata;

- **Nota bene**:
  - In generale le condizioni *possono* sovrapporsi ed in questo caso l'ordine di valutazione delle condizioni (dal primo fino all'else finale) è decisivo;
  - Una assegnazione condizionata deve terminare con una ELSE senza condizione;

Questa è un'assegnazione condizionata (come **if else** o **switch case**). Potremmo o elencare tutte le condizioni o omettere qualcosa.

Assegnazione condizionata (2)

- Un esempio tipico è quello di un **buffer tristate**, con segnale di selezione *Enable*:

```
ARCHITECTURE dataflow OF TriStateBuffer IS
BEGIN
  BufOut <=       BufIn WHEN Enable = '1
                  ELSE 'Z';
END ARCHITECTURE dataflow;
```

  - Quando *Enable* è alto, al segnale *BufOut* viene assegnato *BufIn*, altrimenti viene assegnato il valore tristate ('Z');
- Un altro esempio è quella di un **multiplexer 2 a 1** su un bit, con abilitazione tristate:

```
  mux_out <=      'Z'  AFTER Tpd WHEN en = '0' ELSE
                  in_0 AFTER Tpd WHEN sel = '0' ELSE
                  in_1 AFTER Tpd;
```

  - In pratica è un mux il cui ingresso di selezione è *sel*, la cui uscita è l'ingresso primario di una tristate controllata dal segnale *en*;
  - In questo esempio sono presenti anche dei ritardi di propagazione;

Ricordando il buffer tristate: porta che lavora in questo modo. Ha in uscita l'ingresso se l'abilitazione è 1, altrimenti è alta impedenza (Z). (bufOut e buIn non sono bit, altrimenti non potrei assegnare Z, ma sono segnali più complessi, sono detti Standard Logic).

NB: il vhdl è standardizzato.

Assegnazione condizionata (3)

- In pratica una assegnazione condizionata è un abbreviazione per un processo che contiene assegnazioni di segnali, sottoposti a delle condizioni di tipo IF...THEN;
- Ad esempio l'assegnazione condizionata:

```
segnale <=      waveform_1 WHEN condition_1 ELSE
                waveform_2 WHEN condition_2 ELSE

                waveform_n;
```

è equivalente al process riportato:

```
PROCESS
        IF condition_1 THEN
                segnale <= waveform_1;
        ELSIF condition_2 THEN
                segnale <= waveform_2;
        ELSIF
        ELSE
                segnale <= waveform_n;
        WAIT [sensitivity_list];
END PROCESS;
```

dove [sensitivity list] è l'insieme dei segnali da cui dipendono le clausole.

Assegnazione condizionata (4)

- Il caso degenere di una assegnazione condizionata è una assegnazione che non contenga alcuna parte condizionale ed è quindi equivalente ad un processo con una sola assegnazione, ovvero:

```
segnale <= waveform;
```

- è equivalente a:

```
PROCESS
        segnale <= waveform;
        WAIT [ sensitivity_clause ];
END PROCESS;
```

ovvero a:

```
PROCESS (sensitivity_clause)
        segnale <= waveform;
END PROCESS;
```

L'esempio di sotto è il multiplexer. L'uscita varia a seconda del segnale di abilitazione: questo dispositivo è un multiplexer 2 a 1 con abilitazione. Il segnale in uscita è alta impedenza dopo un certo ritardo quando l'abilitazione è 0. Altrimenti, è uguale al primo ingresso dopo un certo ritardo, quando il segnale di selezione è 0, altrimenti è uguale all'ingresso 1. In questo caso è importante l'ordine delle condizioni. Nel valutare le condizioni, il compilatore, quando ne trova una vera esce dalla condizione (si ferma sempre alla prima che trova). Per questo è importante dare una priorità (spesso è l'abilitazione). Nell'esempio, se anche la selezione

Assegnazione selezionata (1)

- Una **assegnazione selezionata** è del tipo:

```
WITH segnale_di_selezione SELECT
        segnale <=      waveform_1 WHEN caso_1,
                        waveform_2 WHEN caso_2,

                        waveform_n WHEN caso_n;
```

- Una assegnazione selezionata permette di scegliere la waveform da assegnare (fra *waveform_1* … *waveform_n*) ad un segnale target (*segnale*), a seconda del valore (caso_1 … caso_n) assunto da un segnale (*segnale_di_selezione*);

- **Nota bene**:
  - I casi *devono essere esaustivi* (a meno che non si usi un caso OTHERS);
  - I casi non si possono sovrapporre, ovvero devono essere mutuamente esclusivi:

fosse zero, si ferma comunque alla prima. Se poi abbiamo un reset su flip-flop, quella condizione è la prima che dobbiamo andare a considerare.

L'altra assegnazione è quella selezionata che possiamo usare come la condizionata. Anch'essa viene valutata concorrentemente con tutte le altre.

Caso 1, 2, 3 sono condizioni logiche. Queste non possono essere sovrapposte, devono essere per forza disgiunte.

Assegnazione selezionata (2)

- Un blocco che esegue diverse operazioni logiche a seconda del segnale di controllo è riportata in seguito (**Universal Gate**):

```
ARCHITECTURE dataflow OF UniversalGate IS
BEGIN
  WITH Command SELECT
    DataOut <= InA AND InB          WHEN "000",
               InA OR InB           WHEN "001",
               InA NAND InB         WHEN "010",
               InA NOR InB          WHEN "011",
               InA XOR InB          WHEN "100",
               InA XNOR InB         WHEN "101",
               'Z' WHEN OTHERS;
END ARCHITECTURE dataflow;
```

- Si possono anche usare range o valori alternativi:

```
WITH IntCommand SELECT
  MuxOut <= InA WHEN 0 | 1,
            InB WHEN 2 to 5,
            InC WHEN 6,
            InD WHEN 7,
            'Z' WHEN OTHERS;
```

Esempio:

Questa è una porta che in uscita genera un bit risultato di un'operazione tra due bit. Notare che le uscite cambiano a seconda dei tre bit (quando sono tutti e 3 zero, usa la **and** ecc).

NB: notare che quando si indica un solo bit (ascii) si usa un solo apice. Quando si indica una stringa di bit si usano i doppi apici.

Tutte le istruzioni sono diverse e sono 6. Se non vogliamo indicarle tutte, basta usare **when others**

(quelle che specifichiamo però devono essere disgiunte).

Assegnazione selezionata (3)

- In pratica una assegnazione condizionata è un abbreviazione per un processo che contiene assegnazioni di segnali in un costrutto CASE;
- Il processo equivalente a

```
WITH segnale_di_selezione SELECT
        segnale <=      waveform_1 WHEN caso_1,
                        waveform_2 WHEN caso_2,

                        waveform_n WHEN caso_n;
```

è :

```
PROCESS
    CASE segnale_di_selezione IS
        WHEN caso_1 => s <= waveform_1;
        WHEN caso_2=> s <= waveform_2;

        WHEN caso_n=> s <= waveform_n;
    END CASE;
    WAIT [ sensitivity_list ];
END PROCESS;
```

- La sensitivity_list del wait contiene tutti i segnali riferiti nelle forme d'onda (waveform_1 … waveform_n) e nell'espressione del segnale di selezione.

Assegnazione selezionata (4)

- Un esempio di assegnazione selezionata si può trovare in una ALU:

```
WITH alu_function SELECT
        alu_result <=  op1 + op2 WHEN alu_add | alu_incr,
                       op1   op2 WHEN alu_subtract,
                       op1 AND op2 WHEN alu_and,
                       op1 OR op2 WHEN alu_or,
                       op1 AND NOT op2 WHEN alu_mask;
```

  - Il valore del segnale *alu_function* è usato per scegliere quale assegnazione associare al segnale *alu_result*;
  - La sensitivity list del processo equivalente a questa assegnazione è composta dai segnali *alu_function*, *op1* e *op2*: un evento su questi segnali provoca la valutazione dell'espressione;

Confronto fra assegnazioni (1)

- **Assegnazione condizionata**:

```
segnale <=      waveform_1 WHEN condizione_1 ELSE
                waveform_2 WHEN condizione_2 ELSE

                waveform_(n-1) WHEN condizione_(n-2) ELSE
                waveform_n;
```

  - i casi *non sono mutuamente esclusivi*;
  - vengono valutate sequenzialmente le condizioni e la prima che è vera determina l'assegnazione;

- **Assegnazione selezionata**:

```
WITH segnale_di_selezione SELECT
        segnale <=      waveform_1 WHEN caso_1,
                        waveform_2 WHEN caso_2,

                        waveform_n WHEN caso_n;
```

  - i casi *sono mutuamente esclusivi*;
  - non c'è un ordine di scansione dei confronti (i confronti sono in "parallelo");

- Se tutti i casi di una assegnazione condizionata sono mutuamente esclusivi, allora questo costrutto è equivalente ad una assegnazione selezionata.

NB: com'è realizzato il costrutto hardware (porte logiche) non ci interessa. Se ne occuperà il sintetizzatore del chip che abbiamo acquistato. Se vogliamo apportare delle modifiche, dobbiamo trascendere di un livello in giù.

Confronto fra assegnazioni (2)

- Qual è la differenza fra i 2 multiplexer riportati?

```
ENTITY mux_3to1 IS PORT (
          i0, i1, i2: IN BIT;
          s0, s1, s2 : IN BIT;
          y : OUT BIT );
END mux_3to1;

ARCHITECTURE dataflow1 OF mux_4to1 IS
   SIGNAL sel : BIT_VECTOR(2 DOWNTO 0);
BEGIN
   sel <= s2 & s1 & s0;

   WITH sel SELECT
          y <=      i0 WHEN  001 ,
                    i1 WHEN  010 ,
                    i2 WHEN OTHERS;
END dataflow1;
```

```
ENTITY mux_3to1 IS PORT (
          i0, i1, i2: IN BIT;
          s0, s1, s2 : IN BIT;
          y : OUT BIT );
END mux_3to1;

ARCHITECTURE dataflow2 OF mux_4to1 IS
BEGIN
   y <=   i0 WHEN s0 =  1  ELSE
          i1 WHEN s1 =  1  ELSE
          i2;
END dataflow2;
```

- Se s0 = 1, s1 = 1, s2 = 1 cosa succede nei due casi?

Esempio multiplexer: notare che qui l'interfaccia (entity) è la stessa, ma utilizza architetture diverse (dataflow1 e dataflow2). In una c'è un'implementazione con il costrutto di selezione, nell'altra con assegnazione condizionata.

NB: Negli if si possono concatenare le **and** e le **or.**