Introduzione al VHDL

Corso introduttivo al VHDL e all'uso dei tools ISE e Modelsim

Vincenzo Izzo
Università Federico II e INFN
Napoli

Napoli, Novembre 2004

Sommario

- Perché usare il VHDL
- Le regole di base
- Semplici progetti VHDL
- Circuiti combinatori e circuiti sincroni
- Macchine a stati finiti

Lezione 1

VHDL

VHDL e’ un acronimo di **V**HSIC **H**ardware **D**escription **L**anguage

VHSIC e’ un acronimo di **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit

Usare il VHDL - Vantaggi

- Potenza e flessibilità

  E’ possibile descrivere circuiti complessi con relativa semplicità; oltre alla progettazione, il VHDL consente la simulazione del progetto

- Portabilità

  Il VHDL è un linguaggio standard e quindi consente di esportare il codice da un sintetizzatore (e/o un simulatore) all’altro

- Progettazione indipendente dal dispositivo

  La portabilità consente di valutare le prestazioni di un progetto su componenti diversi. Inoltre un progetto VHDL o parte di esso possono essere anche riutilizzati su differenti dispositivi

Usare il VHDL - Svantaggi

- Difficoltà di controllo dell'implementazione

  L'utilizzo di costrutti astratti (clausole if, case, when..) non consente di controllare l'implementazione di un progetto a livello di gate

- Possibile inefficienza delle implementazioni

  I compilatori VHDL non sempre producono l'implementazione ottimale per gli obiettivi dell'utente. Tuttavia diversi tools presentano opzioni per l'implementazione

- Differenti qualità di sintesi

  La qualità della sintesi varia da programma a programma, anche se i produttori di software stanno cominciando ad affrontare questo problema

Entity e architetture

Entity

La dichiarazione di Entity descrive l'I/O di un progetto. Essa contiene il nome del componente da instanziare e le porte di ingresso e di uscita

```
LIBRARY IEEE;
USE ieee.std_logic_1164.ALL;
-- or_a2 è una porta OR

ENTITY or_a2 IS
PORT (a, b: IN std_logic;
      c: OUT std_logic);
END or_a2;

ARCHITECTURE or_a2 OF or_a2 IS
BEGIN
c<=(a OR b);
END or_a2;
```

ieee.std_logic_1164.ALL è la libreria fondamentale. Si usa con l'istruzione USE

Si possono però anche definire librerie personali

Altre librerie spesso utilizzate sono la std_logic_unsigned, la std_logic_arith e textio

Notazioni

```
LIBRARY IEEE;
USE ieee.std_logic_1164.ALL;
-- or_a2 è una porta OR

ENTITY or_a2 IS
PORT (a, b: IN std_logic;
      c: OUT std_logic);
END or_a2;

ARCHITECTURE or_a2 OF or_a2 IS
BEGIN
c<=(a OR b);
END or_a2;
```

Ogni istruzione si conclude con un ;

-- è il simbolo che indica una riga di commento

<= è il simbolo utilizzato per indicare l'assegnazione di un segnale

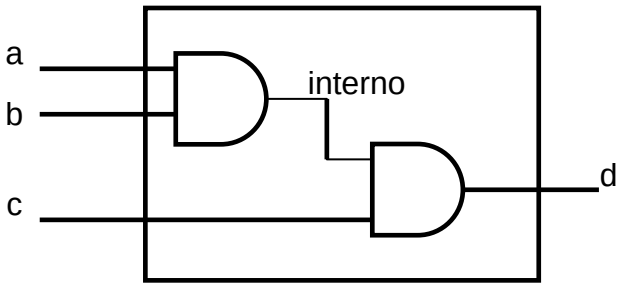
Il valore logico di un bit deve essere contenuto tra ' '

Il valore di un vettore deve essere contenuto tra " "

Segnali

Oltre ai bit di ingresso e uscita del nostro simbolo, in qualche caso è necessario avere a disposizione dei segnali interni, che non sono visti all'esterno del simbolo stesso

```
LIBRARY IEEE;
USE ieee.std_logic_1164.ALL;

ENTITY sig_ex IS
PORT (a, b, c: IN std_logic;
        d: OUT std_logic);
END sig_ex ;

ARCHITECTURE sig_ex OF sig_ex IS
SIGNAL interno: std_logic;
BEGIN
    interno<=(a AND b);
    d<=(interno AND c);
END sig_ex ;
```



Porte e modi

- Porte

  Ogni segnale di I/O in una dichiarazione di entity è chiamato **porta**, ed è analogo ad un pin nel simbolo di uno schematico. Ogni porta che viene dichiarata deve avere un nome, una direzione (**modo**) e un tipo.

- Modi

  I modi descrivono la direzione in cui un dato viene trasferito attraverso la porta.

  Il modo di una porta può essere IN, OUT, INOUT, BUFFER

  IN: i dati possono solo entrare nell'entità

  OUT: i dati possono solo uscire dall'entità

  INOUT: per segnali bidirezionali; permette anche la retroazione

  BUFFER: consente di utilizzare internamente la retroazione; la porta, però, non può essere pilotata dall'esterno dell'entità e può essere connessa solo ad un segnale interno o a una porta di tipo BUFFER di un'altra entità

Tipi

I tipi più utili e meglio supportati per la sintesi, forniti dal package IEEE std_logic_1164 sono i tipi STD_LOGIC e gli array derivati da questi. Per STD_LOGIC si intende il tipo standard per descrivere i circuiti logici

Il tipo STD_LOGIC è definito come segue:
‘U’, -- Uninitialized
‘X’, -- Forcing Unknown
‘0’, -- Forcing 0
‘1’, -- Forcing 1
‘Z’, -- High Impedance
‘W’, -- Weak Unknown
‘L’, -- Weak 0
‘H’, -- Weak 1
‘-’, -- Don’t care

Un oggetto di tipo array consiste di elementi multipli dello stesso tipo. Lo standard IEEE 1164 definisce array di STD_LOGIC come STD_LOGIC_VECTOR

Tipi (2)

Un tipo fondamentale per le macchine a stati è il tipo enumerativo, che è una lista di valori che un oggetto di quel tipo può assumere:

TYPE states IS (s0, s1, s2, s3);

Un segnale potrà poi essere definito del tipo enumerativo appena dichiarato

SIGNAL current_state: states;

TYPE states IS (s0, s1, s2, s3);

SIGNAL current_state: states;

Esempio

Comparatore di uguaglianza a 4 bit.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

--eqcomp4: comparatore di uguaglianza a 4 bit

ENTITY eqcomp4 IS
PORT( a,b: IN std_logic_vector(3 DOWNTO 0);
          equals: OUT std_logic );
END eqcomp4;

ARCHITECTURE dataflow OF eqcomp4 IS
BEGIN
equals<= '1' WHEN (a = b) ELSE '0';
END dataflow;
```

Stavolta gli ingressi non sono due bit, ma due vettori di 4 bit

"equals" va alto quando i due vettori sono uguali bit a bit.

L'istruzione di assegnamento condizionato WHEN-ELSE è abbastanza intuitiva, ma verrà ripresa in seguito

Vettori

Gli indici sono assegnati in modo crescente o decrescente attraverso le parole chiave DOWNTO e TO

```
Bus1: OUT std_logic_vector (7 DOWNTO 0);
Bus2 : OUT std_logic_vector (0 TO 7);
Bus3 : OUT std_logic_vector (0 DOWNTO 7);
……
Bus1 <=“10110010”;    -- IN_BUS1(7)= 1, IN_BUS1(0) = 0
Bus2(3) < =‘1’ ;          -- IN_BUS2 = (U,U,U,U,1,U,U,U)
Bus3 <=“10110010”;    -- IN_BUS3(7)= 0, IN_BUS3(0) = 1
```

E’ possibile invertire l’ordine degli elementi di un bus, compatibilmente con il proprio sintetizzatore

```
Bus1 : IN std_logic_vector ( 3 DOWNTO 0);
Bus2 : OUT std_logic_vector ( 0 TO 3);
……
Bus1 <= Bus2;
.....
Bus1(0) <= Bus2(3);
Bus1(1) <= Bus2(2);
Bus1(2) <= Bus2(1);
Bus1(3) <= Bus2(0);
```

Vettori (2)

E’ possibile riempire un vettore per parti, e non soltanto bit per bit

```
SIGNAL X_bus, Y_bus, Z_bus : std_logic_vector (3 DOWNTO 0);
SIGNAL Byte_bus : std_logic_vector (7 DOWNTO 0);

Byte_bus <= ( 7 => ‘1’ , 6 DOWNTO 4 => ‘0’, OTHERS => ‘1’); -- Byte_bus <=“10001111”;

-- OTHERS si riferisce agli altri valori dell’array non specificati
-- Es.
 Z_bus <= (OTHERS=>’0’);    -- Z_bus <=“00000000”;
```

E’ possibile unire (concatenare) più vettori per generare un vettore più grande

```
SIGNAL X_bus, Y_bus, Z_bus : std_logic_vector (3 DOWNTO 0);
SIGNAL Byte_bus : std_logic_vector (7 DOWNTO 0);
……

Byte_bus <= x_bus & y_bus;  -- Operatore di concatenazione &
```

Architecture

- Architettura

  L'architettura descrive le funzioni di un'entity. Tale descrizione può essere di tipo comportamentale (behavioral) o strutturale

- Behavioral

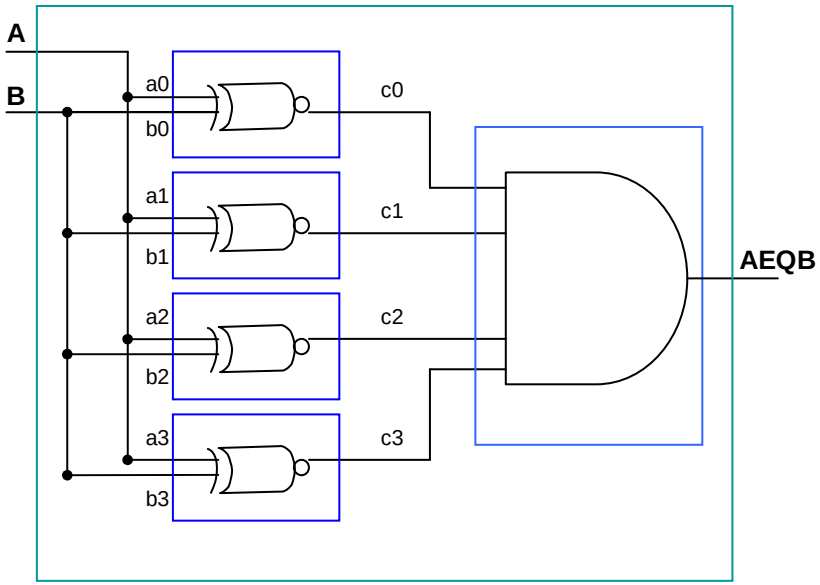
  La descrizione behavioral consente di specificare un insieme di istruzioni che, quando eseguite, descrivono il comportamento dell'entità

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY eqcomp4 IS
PORT( a,b: IN std_logic_vector(3 DOWNTO 0);
          equals: OUT std_logic );
END eqcomp4;

ARCHITECTURE behavioral OF eqcomp4 IS
BEGIN
comp: PROCESS (a,b)
    BEGIN
      IF a=b THEN
          equals <= '1';
      ELSE
          equals <='0';
      END IF;
    END PROCESS comp;
END behavioral;
```

Architecture (2)

- Strutturale

  La descrizione strutturale consiste di una netlist VHDL, che è molto simile ad una netlist di schematico: i componenti sono elencati e connessi insieme mediante segnali




```
……….
USE work.gatespkg.all;
ARCHITECTURE struct OF eqcomp4 IS
SIGNAL c : std_logic_vector (3 DOWNTO 0);
BEGIN
u0: xnor2 PORT MAP (a(0), b(0), c(0));
u1: xnor2 PORT MAP (a(1), b(1), c(1));
u2: xnor2 PORT MAP (a(2), b(2), c(2));
u3: xnor2 PORT MAP (a(3), b(3), c(3));
u4: and4 PORT MAP (c(0), c(1), c(2), c(3), aeqb);
END struct;
```

Process

Il concetto di processo proviene dal software e può essere paragonato ad un programma sequenziale. Un processo può essere combinatorio o clockato.

E' possibile usare più processi in una stessa architecture: bisogna tener presente, però, che tutti i processi saranno eseguiti parallelamente

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY eqcomp4 IS
PORT( a,b: IN std_logic_vector(3 DOWNTO 0);
          equals: OUT std_logic );
END eqcomp4;

ARCHITECTURE behavioral OF eqcomp4 IS
BEGIN
comp: PROCESS (a,b)
    BEGIN
       IF a=b THEN
            equals <= '1';
       ELSE
            equals <='0';
       END IF;
    END PROCESS comp;
END behavioral;
```

Sensitivity list

Elemento fondamentale è la **sensitivity list**, che contiene tutti i segnali ai quali il processo deve essere sensibile. Ogni volta che un segnale della **sensitivity list** cambia valore, il processo viene “attivato”

Come viene creata la sensitivity list?

•Se un cambiamento nel segnale di input causa un IMMEDIATO cambiamento in uno qualsiasi dei segnali assegnati in un processo, allora il segnale DEVE entrare nella sensitivity list

•Se un cambiamento nel segnale di input NON causa un IMMEDIATO cambiamento in uno qualsiasi dei segnali assegnati in un processo, allora il segnale NON DEVE entrare nella sensitivity list

Sensitivity list (2)

```
……
PROCESS (sel,a)
BEGIN
    IF sel='1' then
        out_signal<=a;
    ELSE
        out_signal<=not a;
    END IF;
END PROCESS;
```

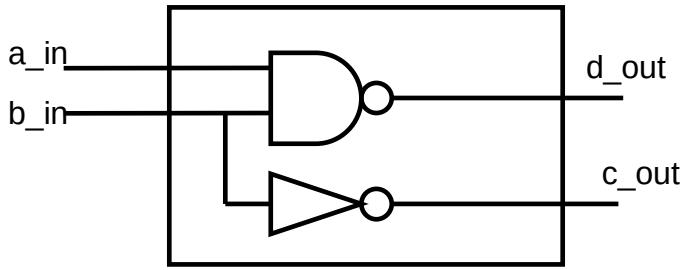
```
……
PROCESS (sel)
BEGIN
    IF sel='1' then
        out_signal<=a;
    ELSE
        out_signal<=not a;
    END IF;
END PROCESS;
```

Il secondo processo deve tenere memoria dell'ultimo valore di *a*, quando c'è stato il cambiamento di *sel*. Se il *tool* di sintesi accetta il codice, il sintetizzatore introdurrà un *latch* indesiderato

Processo combinatorio

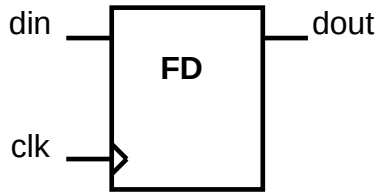
I processi combinatori si usano per descrivere logica puramente combinatoria.

```
……
comb_process: process (a_in, b_in)
begin
 c_out <= not (a_in and b_in);
 d_out <= not b_in;
end process comb_process;
```



Processo clockato

```
……
example: process
begin
  wait until clk='1';
  dout<=din;
end process example;
```



Il risultato di un processo clockato consiste in flip-flops e logica combinatoria (eventuale)

Tutti i segnali assegnati all'interno di un processo clockato sono uscite di flip-flop

E' possibile "unire" più processi clockati

Variabili e segnali

I segnali sono utilizzati per esecuzione parallela

Le variabili sono usate per l'esecuzione sequenziale, come un comune programma software

Un segnale può essere dichiarato soltanto nella parte parallela, ma può essere usato sia nella parte parallela che in quella sequenziale

Una variabile può essere soltanto dichiarata ed utilizzata nella parte parallela

L'assegnazione di un segnale richiede il simbolo "<="

L'assegnazione di una variabile richiede il simbolo ":="

E' possibile assegnare ad un segnale il valore di una variabile e viceversa, a patto che abbiano lo stesso tipo

```
……..
ARCHITECTURE rtl OF example IS
Concurrent declaration part
 BEGIN
   concurrent VHDL
   process(…)
   sequential declaration part
   begin
        sequential VHDL
    end process;
    concurrent VHDL;
 END;
```

Operatori logici

NOT
AND
OR
NAND
XOR
XNOR

Sono operatori definiti dallo standard IEEE1164 per il tipo STD_LOGIC e per i suoi vettori STD_LOGIC_VECTOR.

Sono definiti anche per altri tipi, utili per la sintesi, es. BIT (‘0’,’1’) e BOOLEAN (‘TRUE’,’FALSE’)

Questi operatori non hanno un ordine di precedenza, quindi **SONO RICHIESTE LE PARENTESI**

L’operatore **NOT** ha priorità più alta, per tutti gli altri la priorità va da sinistra a destra

Operatori relazionali

=	*Operatore di uguaglianza*
/=	*Operatore di disuguaglianza*
>, >=	*Maggiore e maggiore uguale*
<, <=	*Minore e minore uguale*

I tipi di operandi devono essere uguali

Il risultato è booleano (cioè è vero o falso)

Gli array sono uguali se le loro lunghezze coincidono e tutti gli elementi corrispondenti sono uguali

Introduzione a ISE 6.x

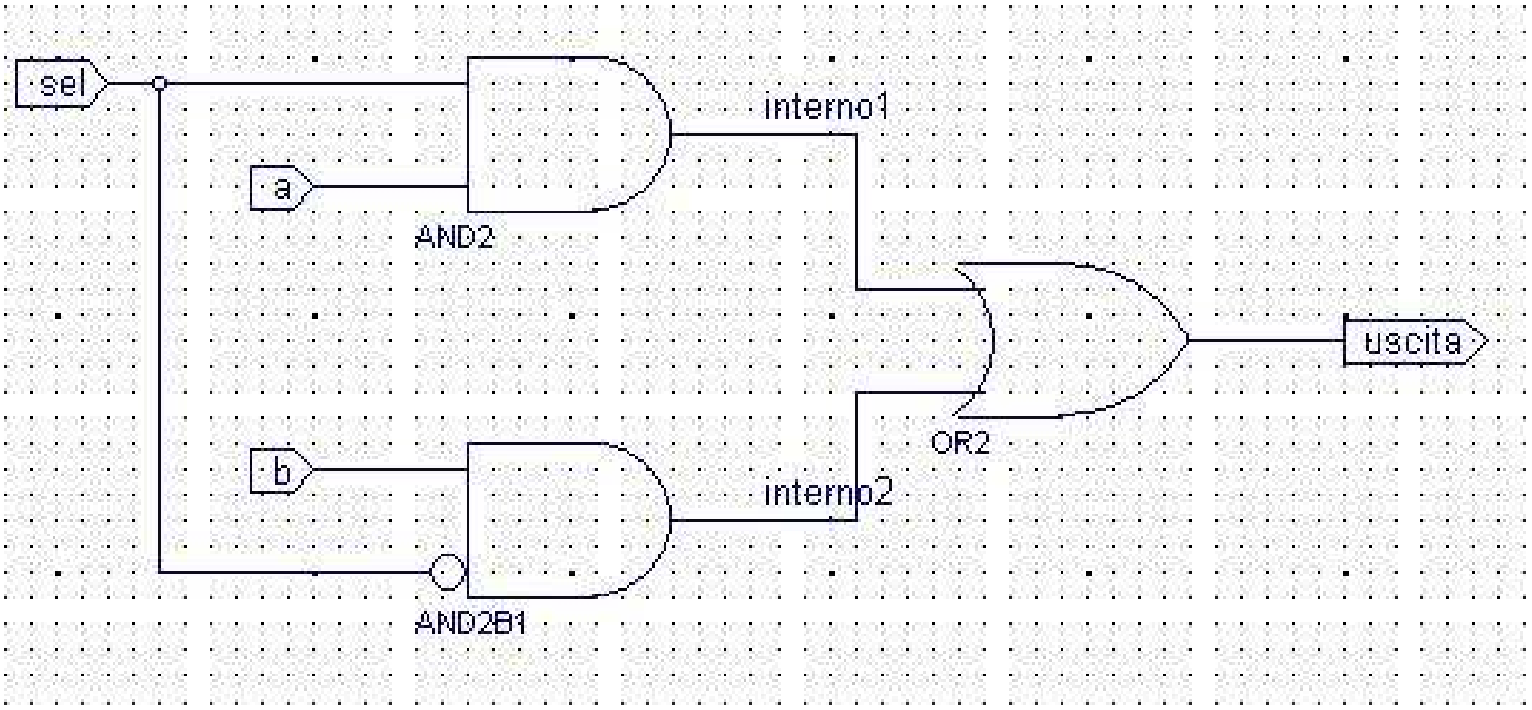
- Creazione di un nuovo progetto
- Scelta delle proprietà del progetto
- Creazione di files sorgenti, schematico e VHDL
- Sintesi e creazione di simboli
- Syntax check e individuazione degli errori

Esercizi

- Multiplexer
- Comparatore
- Decodificatore

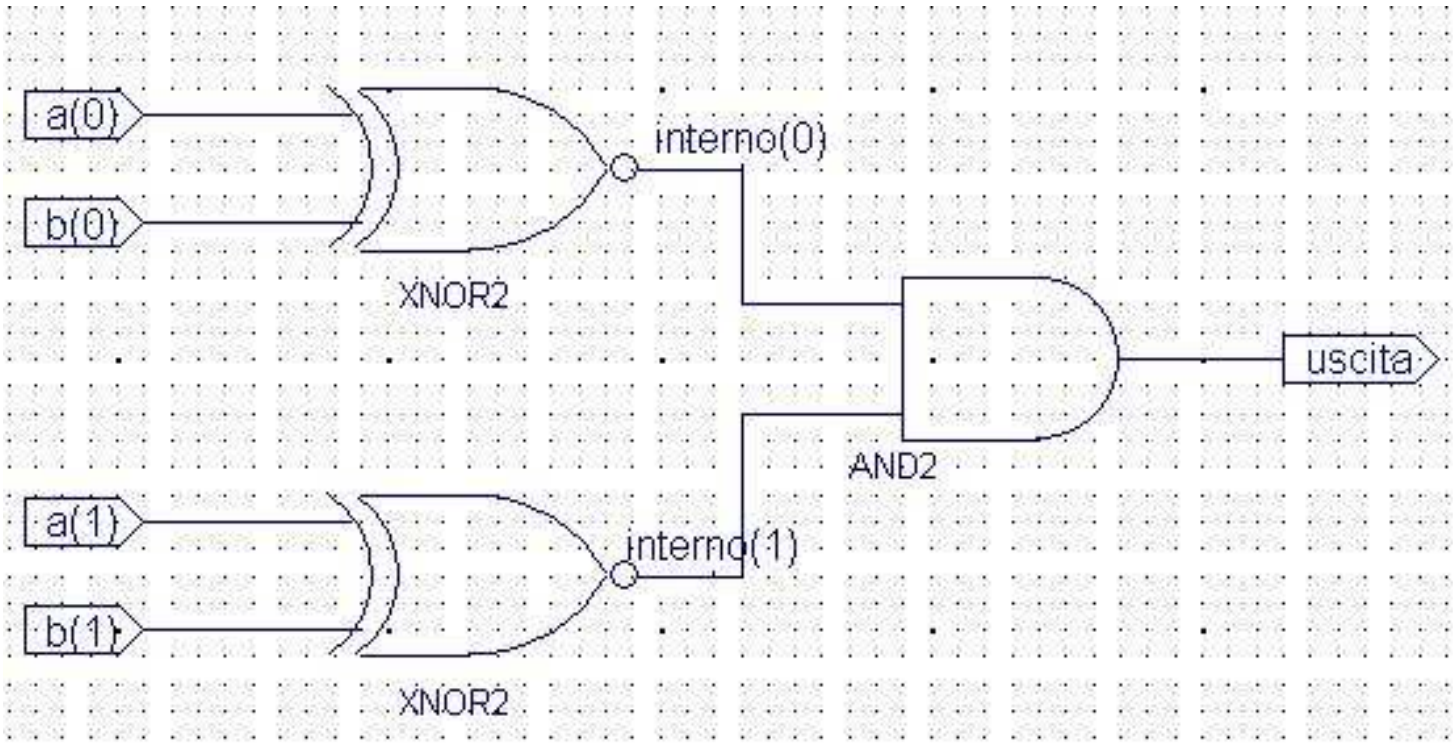
Esercizi

Multiplexer



Esercizi

Comparatore



Lezione 2

Istruzioni

- When – Else
- If – Then – Else
- Case – When
- Rising_edge e Clk'Event

When - Else

Ad un segnale viene assegnato un valore in base al verificarsi di una determinata condizione

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux IS
PORT( a,b, c, d: IN std_logic;
            s: IN std_logic_vector(1 DOWNTO 0);
            exit: OUT std_logic );
END mux;

ARCHITECTURE archmux OF mux IS
BEGIN
      exit <= a WHEN (s = “00”) ELSE
               b WHEN (s = “01”) ELSE
               c WHEN (s = “10”) ELSE
               d;
END archmux;
```

```
signal <= value_a WHEN condition1 ELSE
          value_b WHEN condition2 ELSE
            ……..
          value_n;
```

La condizione dopo il WHEN può contenere qualunque espressione e non deve specificare valori mutuamente esclusivi

Se le condizioni in un’istruzione WHEN – ELSE non sono mutuamente esclusive, la priorità più alta viene assegnata alla prima condizione WHEN elencata

If – Then - Else

L'istruzione IF-THEN-ELSE è utilizzata per selezionare un insieme di istruzioni da eseguire, in base alla valutazione di una condizione o un insieme di condizioni

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux IS
PORT( a, b, s: IN std_logic;
              exit: OUT std_logic );
END mux;

ARCHITECTURE archmux OF mux IS
BEGIN
  comb: PROCESS (a, b, s)
    BEGIN
        IF (s = '0') THEN
              exit<=a;
        ELSE
              exit<=b;
        END IF;
    END PROCESS;
END archmux;
```

```
IF (condition1) THEN
              do something1;
ELSIF (condition2) THEN
              do something2;
……..
ELSE
              do something else;
END IF;
```

L'insieme di istruzioni viene chiuso da END IF

Le istruzioni che seguono la parola chiave THEN sono eseguite in ordine di apparizione: sono dunque sequenziali e non concorrenti

Case - When

L'istruzione CASE-WHEN è utilizzata per specificare un insieme di istruzioni (complicate quanto si voglia) da eseguire in base al valore di un segnale selezione

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY mux IS
PORT( a, b, s: IN std_logic;
                exit: OUT std_logic );
END mux;

ARCHITECTURE archmux OF mux IS
BEGIN
  comb: PROCESS (a, b, s)
    BEGIN
        CASE s IS
                WHEN '0' =>
                    exit <=a;
                WHEN OTHERS =>
                    exit<=b;
        END CASE;
    END PROCESS;
END archmux;
```

```
CASE (selection_signal) IS
    WHEN value_1 =>
                do something1;
    WHEN value_2 =>
                do something2;
……..……
    WHEN OTHERS
                do something else;
END CASE;
```

L'insieme di istruzioni viene chiuso da END CASE

Memorie implicite

La parola chiave *ELSE*, quando usata con l'istruzione *IF- THEN* o con l'istruzione *WHEN* è fondamentale, in quanto permette al sintetizzatore di "capire" sempre cosa fare; in caso contrario viene introdotta nel circuito una cosiddetta **memoria implicita** (o *inferred latch*)

Analoga importanza assume la parola chiave *WHEN OTHERS*, quando usata all'interno dell'istruzione *CASE*

Tuttavia, se sono elencati tutti i possibili valori che il segnale di selezione può assumere (come nel caso di una macchina a stati finiti) l'istruzione *WHEN OTHERS* può creare problemi di sintesi

Rising_edge e Clk'Event

Il package STD_LOGIC_1164 definisce le funzioni RISING_EDGE e FALLING_EDGE per rilevare i fronti di salita e di discesa dei segnali

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY d_flipflop IS
PORT( d, clk: IN std_logic;
                q: OUT std_logic );
END d_flipflop;

ARCHITECTURE d_flipflop OF d_flipflop IS
BEGIN
  comb: PROCESS (clk)
    BEGIN
        IF (clk'EVENT AND clk='1') THEN
          q <= d;
        END IF;
    END PROCESS;
END d_flipflop;
```

Tali funzioni sono di fondamentale importanza per i circuiti sincroni, nei quali i segnali sono aggiornati sul fronte di salita o di discesa del segnale di cadenza

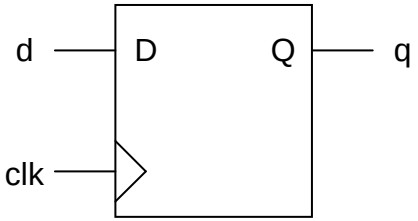
Nell'esempio è utilizzata la costruzione (clk'EVENT AND clk='1'), che e' equivalente alla funzione RISING_EDGE, se il segnale di clock e' di tipo STD_LOGIC

Ovviamente (clk'EVENT AND clk='0'), e' equivalente alla funzione FALLING_EDGE, per circuiti attivi su transizioni *High to Low*

Esempi

- Flip Flop di tipo D

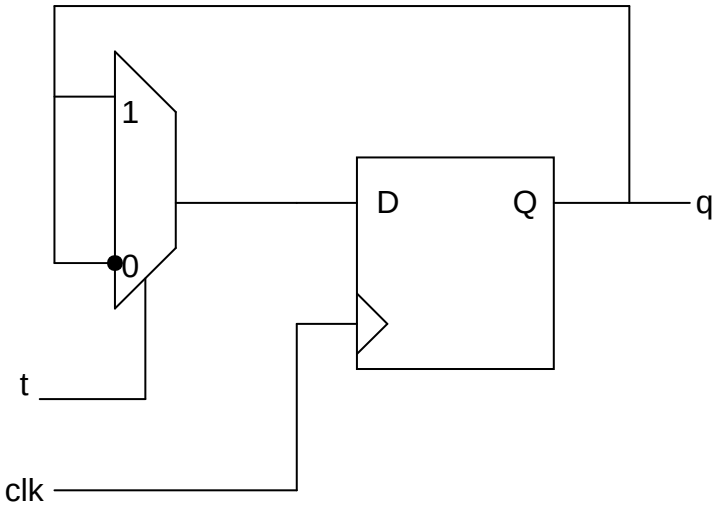
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY d_flipflop IS
PORT( d, clk: IN std_logic;
                q: OUT std_logic );
END d_flipflop;

ARCHITECTURE d_flipflop OF d_flipflop IS
BEGIN
  comb: PROCESS (clk)
     BEGIN
          IF (RISING_EDGE(clk)) THEN
            q <= d;
          END IF;
     END PROCESS;
END d_flipflop;
```



Esempi (2)

- Flip Flop di tipo T

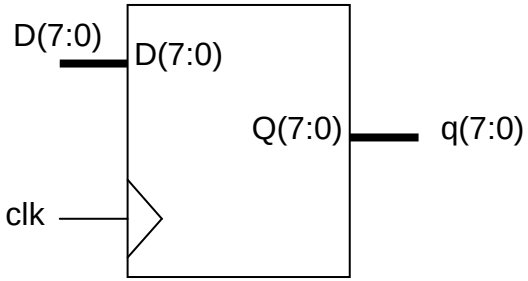
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY t_flipflop IS
PORT( t, clk: IN std_logic;
                q: INOUT std_logic );
END t_flipflop;

ARCHITECTURE t_flipflop OF t_flipflop IS
BEGIN
  comb: PROCESS (clk)
    BEGIN
        IF (clk'EVENT AND clk='1') THEN
          IF (t = '1') THEN
            q <=  not (q);
          ELSE
            q <= q;
          END IF;
        END IF;
    END PROCESS;
END t_flipflop;
```



Esempi (3)

- Registro ad 8 bit

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY reg_8bit IS
PORT(       d: IN std_logic_vector (7 DOWNTO 0);
            clk: IN std_logic;
            q: OUT std_logic_vector (7 DOWNTO 0));
END reg_8bit;

ARCHITECTURE reg_8bit OF reg_8bit IS
BEGIN
 comb: PROCESS (clk)
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
        q <= d;
      END IF;
   END PROCESS;
END reg_8bit;
```



Il Reset

- Reset e Preset Asincroni
- Reset e Preset Sincroni

Reset asincrono

Lo standard VHDL non richiede che un circuito venga inizializzato o resettato. In hardware tuttavia è necessario specificare lo stato di reset di ciascun dispositivo

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY d_flipflop IS
PORT( d, clk, reset: IN std_logic;
                q: OUT std_logic );
END d_flipflop;

ARCHITECTURE async_reset OF d_flipflop IS
BEGIN
  comb: PROCESS (clk, reset)
    BEGIN
        IF (reset='1' ) THEN
          q <= '0';
        ELSIF (clk'EVENT AND clk='1') THEN
          q <= d;
        END IF;
    END PROCESS;
END async_reset;
```

Lo standard specifica che per la simulazione un segnale, se non sia esplicitamente inizializzato, sia inizializzato ad un valore che è legato al suo tipo. Così, il tipo STD_LOGIC verrà inizializzato ad 'U', mentre uno di tipo bit verrà inizializzato a '0'.

Preset asincrono

Per descrivere un preset al posto di un reset basta apportare una piccola modifica al listato della pagina precedente

```
……..
BEGIN
IF (preset =‘1’) THEN
    q <= ‘1’;
ELSIF (clk’EVENT AND clk=‘1’) THEN …..
```

Il reset sincrono

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY d_flipflop IS
PORT( d, clk, reset: IN std_logic;
                q: OUT std_logic );
END d_flipflop;

ARCHITECTURE sync_reset OF d_flipflop IS
BEGIN
 comb: PROCESS (clk, reset)
   BEGIN
      IF (clk'EVENT AND clk='1') THEN
         IF (reset = '1') THEN
           q <= '0';
         ELSE
           q <= d:
         END IF;
      END IF
   END PROCESS;
END sync_reset;
```

E' possibile resettare (o presettare) un flip flop in modo sincrono ponendo la condizione di reset (o preset) all'interno del processo che descrive la logica sincrona con il clock

In VHDL è possibile anche descrivere una combinazione di reset e/o preset sincroni/asincroni. L'esempio successivo è un registro a 8 bit che può essere resettato a 0 in maniera asincrona, ogni volta che il segnale di reset va a '1', e può essere inizializzato in maniera sincrona con tutti '1' e caricato sul fronte di salita del clock

Il reset sincrono

ESEMPIO

Registro a 8 bit con reset asincrono e inizializzazione sincrona

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY reg_8bit IS
PORT( d: IN std_logic_vector (7 DOWNTO 0);
        clk, init, reset: IN std_logic;
        q: OUT std_logic_vector (7 DOWNTO 0));
END reg_8bit;

ARCHITECTURE example OF reg_8bit IS
BEGIN
  PROCESS (clk, reset)
    BEGIN
       IF (reset ='1') THEN
          q <= "00000000";
       ELSIF(clk'event AND clk='1') THEN
            IF (init='1') THEN
               q <= "11111111";
            ELSE
               q <= d:
            END IF;
       END IF
    END PROCESS;
END example;
```

Testbench

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
LIBRARY UNISIM;
USE UNISIM.Vcomponents.ALL;

ENTITY testbench IS
END testbench;

ARCHITECTURE behavioral OF testbench IS

  COMPONENT top_level
  PORT( clock, rst: IN STD_LOGIC;
        ingresso1: INSTD_LOGIC;
        ingresso2: IN STD_LOGIC;
        uscita: OUT STD_LOGIC);
  END COMPONENT;

 SIGNAL clock:STD_LOGIC;
 SIGNAL rst:STD_LOGIC;
 SIGNAL ingresso1: STD_LOGIC;
 SIGNAL ingresso2: STD_LOGIC;
 SIGNAL uscita: STD_LOGIC;
 ……..
```

Anche i file per la simulazione di un progetto sono scritti in VHDL.

Bisogna definire un componente, relativo al progetto che intendiamo simulare, che ha come ingressi e uscite le net che si riferiscono al nostro progetto

Bisognerà poi definire i segnali che vorremo impulsare. Può essere opportuno chiamarli nello stesso modo delle porte del componente che abbiamo definito

Testbench (2)

```
……..
SIGNAL uscita: STD_LOGIC;

BEGIN

  UUT: top_level PORT MAP(
            clock => clock,
            rst => rst,
            ingresso1 => ingresso1,
            ingresso2 => ingresso2,
            uscita => uscita
            );


reset: process
  begin

  rst<='0';
wait for 200 ns;
  rst<='1';
wait for 20000 ns;

end process reset;
 ……..
```

Il comando *PORT MAP* consente di connettere tra loro due net: in questo caso colleghiamo ciascuna porta del componente relativo al nostro progetto con il corrispondente segnale che abbiamo definito

A questo punto possiamo impulsare i segnali che abbiamo definito. Per farlo, dobbiamo definire uno o più processi, al cui interno vengono assegnati dei valori ai segnali che ci interessa impulsare

Occorre ricordare che, se si definiscono più processi, essi saranno eseguiti parallelamente, e quindi possono crearsi conflitti, se sullo stesso segnale agiscono due processi differenti

Testbench (3)

```
……..

clocked: process
begin
   clock<='0';
wait for 20 ns;
   clock<='1';
wait for 20 ns;
end process clocked;

dati: process
begin
     ingresso1<='0';
     Ingresso2<='1'
wait for 240 ns;
     ingresso1<='1';
wait for 40 ns;
     ingresso2<='0';
wait for 200 ns;
     ingresso2<= 'Z';
wait for 400 ns;
……..
end process dati;
```

Può essere utile utilizzare tre processi differenti: uno per il reset, uno per il clock e uno per impulsare gli altri segnali.

In tal modo si ha una migliore leggibilità del codice ma anche una migliore struttura logica: sarà più semplice intervenire per eventuali future correzioni e si evitano conflitti tra i vari processi

N.B. Quando un processo esaurisce le sue istruzioni viene rieseguito dall'inizio

Applicazioni con ISE 6.x

- Multiplexer
- Registri
- Contatore
- Testbench e simulazione
- Language templates

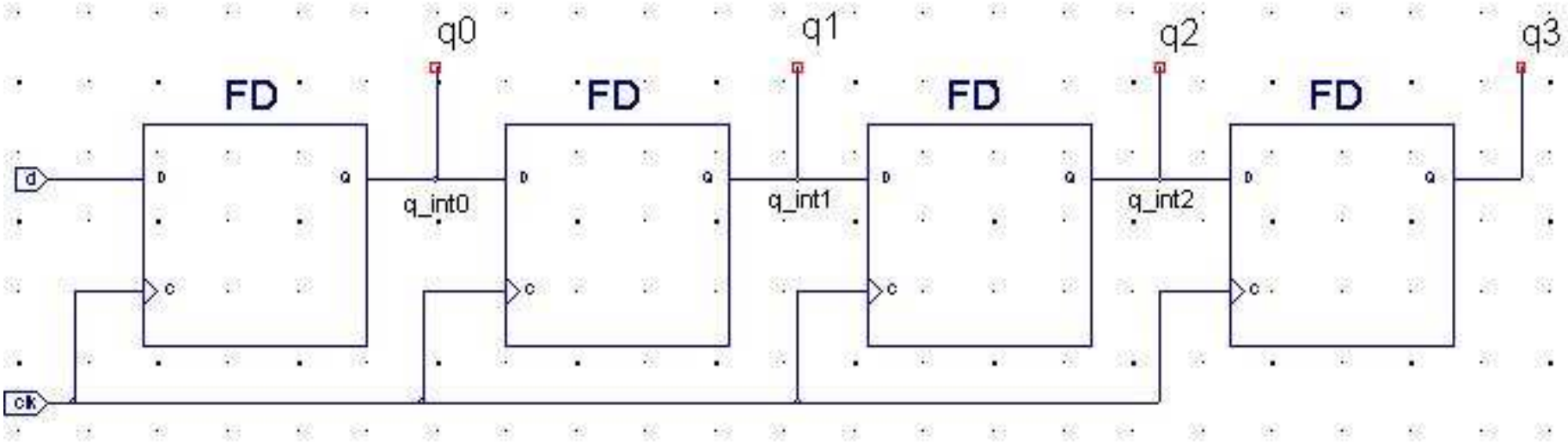
Esercizi

- Disegnare un multiplexer utilizzando i costrutti IF-THEN, WHEN-ELSE e CASE-WHEN
- Disegnare un flip-flop con clok enable ed un registro serie-parallelo
- Disegnare un contatore

Esercizi

REGISTRO SERIE-PARALLELO

Suggerimento

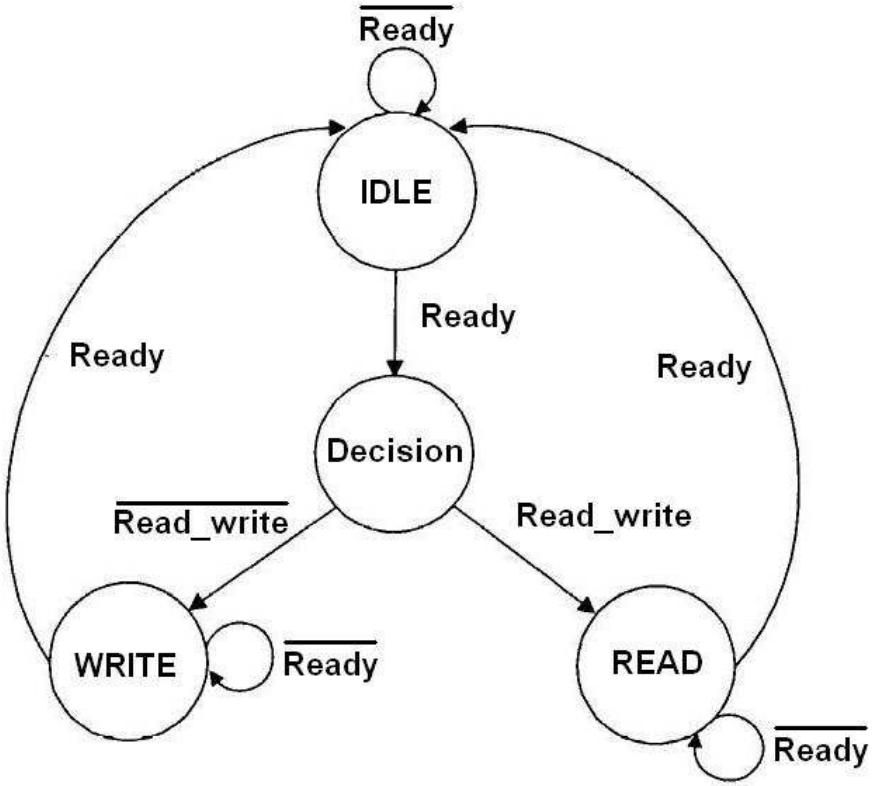


Lezione 3

Macchine a stati

Metodo tradizionale

Disegnare una macchina che controlla i segnali di *Output_Enable* e *Write_Enable* di una memoria



Primo passo: disegnare il diagramma

Secondo passo: assegnare una codifica a ciascuno stato e le uscite in corrispondenza di ciascuno degli stati

Terzo passo: creare una tabella di transizione degli stati, procedendo ad una eventuale minimizzazione degli stati

	Outputs	
State	***OE***	***WE***
Idle	***0***	***0***
Decision	***0***	***0***
Write	***0***	***1***
Read	***1***	***0***

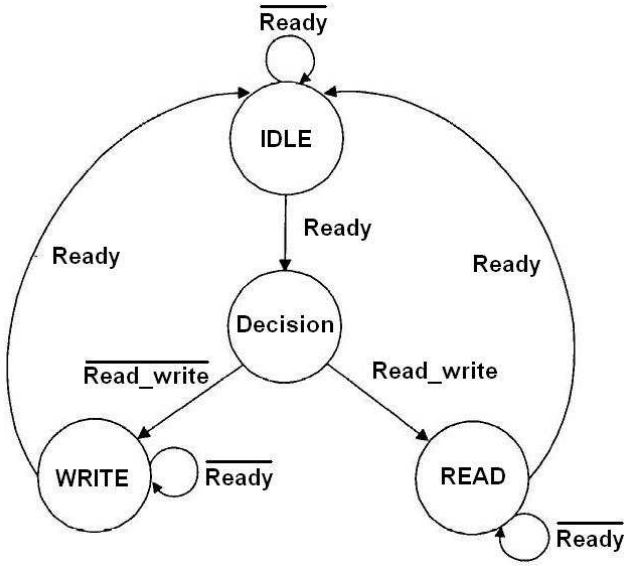
Metodo tradizionale (2)

Vogliamo utilizzare il minor numero di registri per i 4 stati, quindi 2 (qo e q1)

La codifica degli stati è riportata nella colonna *Present State*

La colonna *Next State* mostra la transizione dallo stato presente verso lo stato successivo in base al valore dei due ingressi *read_write* e *ready*

			(read_write, ready)					
	Present State		***00***	***01***	***11***	***10***	***Outputs***	
State	***qo***	***q1***	***Next State (Q0 Q1)***				***OE***	***WE***
Idle	***0***	***0***	***00***	***01***	***01***	***00***	***0***	***0***
Decision	***0***	***1***	***11***	***11***	***10***	***10***	***0***	***0***
Write	***1***	***1***	***11***	***00***	***00***	***11***	***0***	***1***
Read	***1***	***0***	***10***	***00***	***00***	***10***	***1***	***0***



Metodo tradizionale (3)

Con le mappe di Karnaugh è possibile determinare l'equazione dello stato successivo per ciascuno dei bit di stato

$Q_o = \overline{q_0} q_1 + q_0 \overline{ready}$

$Q_1 = \overline{q_0}\,\overline{q_1} ready + \overline{q_0} q_1 \overline{read\_write} + q_0 q_1 \overline{ready}$

Q0

qoq1 \ read_write, ready	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	1	0	0	1
10	1	0	0	1

Q1

qoq1 \ read_write, ready	00	01	11	10
00	0	1	1	0
01	1	1	0	0
11	1	0	0	1
10	0	0	0	0

Progettazione in VHDL

In VHDL ogni stato può essere tradotto in un “caso” mediante un’istruzione *CASE-WHEN*

La transizione degli stati può essere specificata mediante una serie di istruzioni *IF-THEN-ELSE*

Primo passo: disegnare il diagramma

```
TYPE stato IS (idle, decision, read, write);
SIGNAL present_state, next_state: stato;
```

Secondo passo: definire un tipo enumerativo, formato dagli stati della macchina, e due segnali di tale tipo

```
comb: PROCESS (present_state, read_write, ready)
BEGIN
        ……..
END PROCESS comb;
```

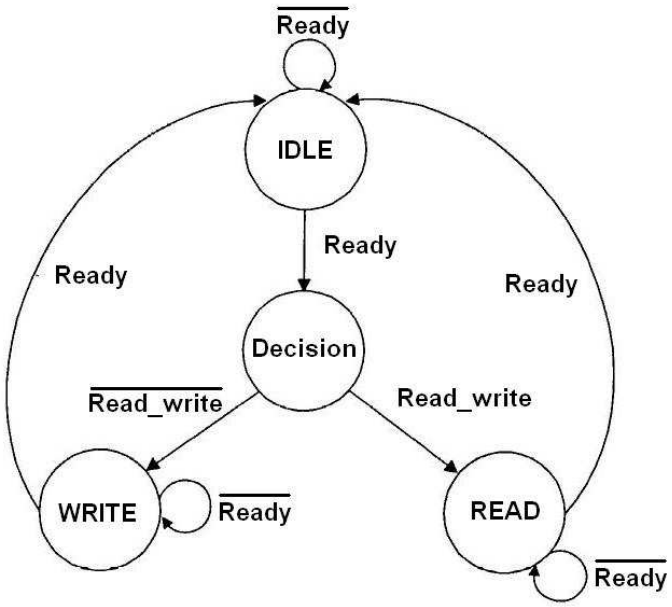
Terzo passo: creare un processo in cui sono descritte le transizioni della macchina.

Il processo che determina *next_state* deve essere sensibile a *present_state* e agli ingressi *ready* e *read_write*

Nel processo viene definita una istruzione *CASE-WHEN* e vengono specificati i diversi stati della macchina

Dichiarazione degli stati

Per ciascuno stato si specificano le uscite relative a quello stato e le transizioni che partono da esso



Il primo caso da specificare è lo stato di *idle*.

Ci sono due opzioni, nel caso in cui *present_state* è *idle:*

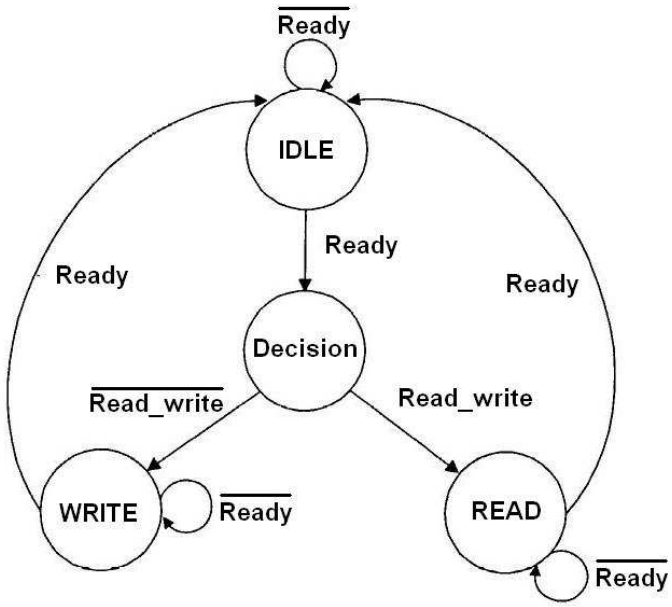
Se *ready* è ‘1’ si ha una transizione allo stato *decision*

Altrimenti si resta nello stato *idle*

```
CASE present_state IS
    WHEN idle =>
        oe <= ‘0’;
        we <= ‘0’;
      IF ready = ‘1’ THEN
          next_state <= decision;
      ELSE
          next_state <= idle;
      END IF;
```

Dichiarazione degli stati (2)

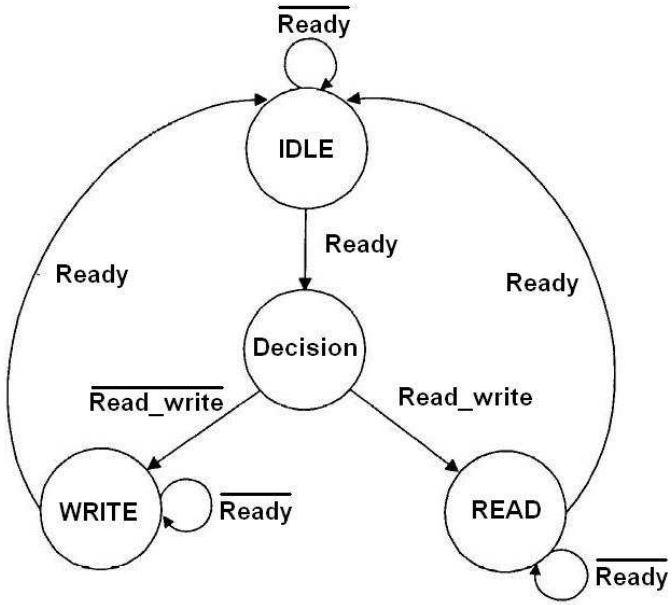
La codifica degli altri stati si effettua in modo analogo:




```
CASE present_state IS

    WHEN idle =>
        oe <= '0'; we <= '0';

      IF (ready = '1' ) THEN
          next_state <= decision;
      ELSE
           next_state <= idle;
      END IF;

    WHEN decision =>
        oe <='0'; we <='0';

       IF (read_write = '1') THEN
          next_state <= read;
       ELSE
          next_state <= write;
       END IF;

    WHEN read =>
         oe <='1'; we <='0';

       IF (ready = '1') THEN
          next_state <= idle;
       ELSE
          next_state <= read;
       END IF;
     ……..……

  END CASE;
END PROCESS comb;
```

Dichiarazione degli stati (3)

Un modo alternativo di descrivere i differenti casi della macchina a stati è quello di specificare volta per volta le uscite per ciascuno stato: il codice è più leggibile, non crea problemi al sintetizzatore, ma ci sono molte più righe ed è più complesso da debuggare

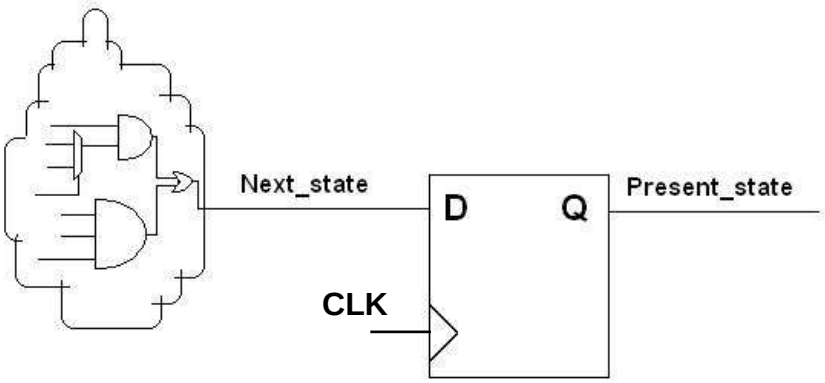



```
CASE present_state IS

    WHEN idle =>

       IF (ready = '1' ) THEN
           next_state <= decision;
           oe <= '0';
           we <= '0';
       ELSE
           next_state <= idle;
           oe <= '0';
           we <= '0';
       END IF;

    WHEN decision =>

        IF (read_write = '1') THEN
           next_state <= read;
           oe <='1';
           we <='0';
        ELSE
           next_state <= write;
           oe <='0';
           we <='1';
        END IF;


     ……..……..……..……..……..……..

  END CASE;
END PROCESS comb;
```

Processo clockato

Il processo descritto in precedenza indica soltanto che l'assegnazione di *next_state* è basata su *present_state* e sugli ingressi correnti, ma non indica quando *present_state* diventa *next_state*.

Questo avviene in modo sincrono, sul fronte di salita del clock, e viene specificato da un secondo processo:




```
clocked: PROCESS (clk)

BEGIN
    IF (clk'event AND clk ='1') THEN
        present_state <= next_state;
    END IF;
END PROCESS clocked;
```

Il reset asincrono

E’ possibile, con una piccola modifica al processo descritto in precedenza, specificare l’azione che la macchina a stati deve intraprendere in presenza di un reset asincrono (es. al POWER UP)

Poiché questo deve avvenire in modo asincrono, è necessario che il comando di reset non sia subordinato al verificarsi di un fronte di salita del clock

```
clocked: PROCESS (reset, clk)

BEGIN
    IF reset =‘1’ THEN
        present_state <= idle;
    ELSIF (clk’event AND clk =‘1’) THEN
        present_state <= next_state;
    END IF;
END PROCESS clocked;
```

Il reset sincrono

Oltre a specificare l'azione che la macchina a stati deve intraprendere in presenza di un reset asincrono (es. al POWER UP), puo' essere necessario introdurre un reset sincrono (es. generato da un'altra macchina a stati)

Poiché questo deve avvenire in modo sincrono, è necessario che il comando di reset sia subordinato al verificarsi di un fronte di salita del clock

```
clocked: PROCESS (reset, clk)

BEGIN
   IF (clk'event AND clk ='1') THEN
        IF ( reset ='1' ) THEN
            present_state <= idle;
        ELSIF
            present_state <= next_state;
       END IF;
   END IF;
END PROCESS clocked;
```

Esercizi

Macchina di Self-Test dei LED