Introducción a VHDL
Parte II

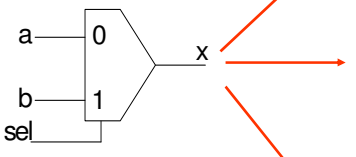
Martín Vázquez - Lucas Leiva
Mar 2013

INSTITUTO DE INVESTIGACIÓN EN TECNOLOGÍA INFORMÁTICA AVANZADA

Universidad Nacional del Centro de la Provincia de Buenos Aires

Circuitos básicos: Multiplexores

Multiplexores por asignación concurrente: simple, condicional y por selección



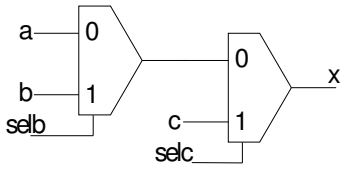

```
x <= (a and not sel) or (b and sel) ;
```

```
x <= a when sel='0' else
     b;
```

```
with sel select
     x <= a when sel='0',
          b when sel='1',
          '0' when others;
```

Circuitos básicos: Multiplexores

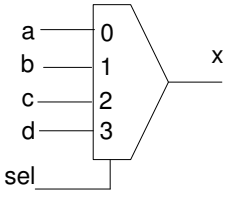
Multiplexores usando sentencias secuenciales: if-then-else.
Con prioridad




```
process (selc, selb, a, b, c)
begin
        if (selc = '1') then
                x <= c;
        elsif (selb = '1') then
                x <= b;
        else
                x <= a;
        end if;
end process;
```


Circuitos básicos: Multiplexores

Multiplexores usando sentencias secuenciales: case.
Sin prioridad

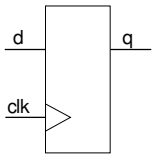


Mux 4-1

```
process (sel, a, b, c, d)
begin
        case sel is
                when "00" =>
                        x <= a;
                when "01" =>
                        x <= b;
                when "10" =>
                        x <= c;
                when others =>
                        x <= d;
        end case;
end process;
```

Registros

- Existen dos métodos para crear flip-flops
  - Instanciación del componente flip-flop desde una librería
  - Utilizar un proceso sensible al flanco de reloj



Registro: Instanciación de componentes

```
libary ieee;
use ieee.std_logic_1164.all;
entity ffd is
            port (d, clk : in std_logic;
                        q: out std_logic);
end ffd;

use ieee.rtlpkg.all;
architecture instffd of ffd is
begin
            flipflop:  dff port map (d,clk,q);
            -- componente dff definido en  rtlpkg
end instffd;
```

Ejemplo flip-flop tipo D

Registro: Descripción basada en proceso

```
libary ieee;
use ieee.std_logic_1164.all;
entity ffd is
        port (d, clk : in std_logic;
                q: out std_logic);
end ffd;

architecture compffd of ffd is
begin
        flipflop: process (clk)
        begin
                if (clk´event and clk='1') then
                        d <= q;
                end if;
        end process;
end compffd;
```

Ejemplo flip-flop tipo D

Registro: Descripción basada en proceso

El sintetizador infiere que debe crear registro para la señal *q* basándose en:

- El elemento es únicamente sensible a la señal de reloj clock
- La expresión ***clock´event and clock='1'*** implica que la asignación de la señal se realiza en el flanco de subida de reloj
- Se sintetiza un elemento síncrono
- La especificación incompleta de la sentencia ***IF-THEN*** por faltar la cláusula ***ELSE*** implica que si la condición ***clock´event and clock='1'*** no se satisface (no hay flanco de subida), *q* debe mantener su valor hasta la siguiente asignación (memoria implícita)

Ejemplos básicos de codificación

Tipos de registro

```
ff: process (clk)
begin
        if (clk´event and clk='1') then
                output <= input;
        end if;
end process;
```

Flip Flop D

```
ff_SA: process (clk, st)
begin
        if (st='1') then
                output <= others => '1';
        elsif (clk´event and clk='1') then
                output <= input;
        end if;
end process;
```

Flip Flop D con set asíncrono

Ejemplos básicos de codificación

Tipos de registro

```
ff_RS: process (clk)
begin
    if (clk´event and clk='1') then
        if (rst= '1') then
            output <= others => '0';
        else
            output <= input;
        end if;
    end if;
end process;
```

Flip Flop D con reset síncrono

```
ff_SA: process (clk, rst)
begin
    if (rst='1') then
        output <= others => '0';
    elsif (clk´event and clk='1') then
        output <= input;
    end if;
end process;
```

Flip Flop D con reset asíncrono

Ejemplos básicos de codificación

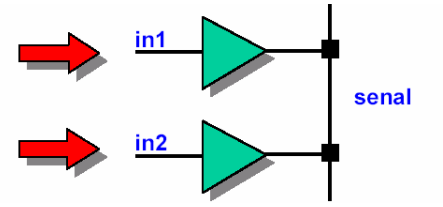
Contadores

```
PCont: process (clk, rst)
begin
        if (rst='1') then
                cont <= (others => '0');
        elsif (clk´event and clk='1') then
                cont <= cont + 1;
        end if;
end process;
```


Concepto de *driver* de una señal

- El *driver* es el elemento que da valores a una señal
- Para cada señal que se le asigna un valor dentro de un proceso se crea un *driver* para esa señal
  - Independientemente de cuanto veces se le asigne valor a una señal, se crea un único *driver* por proceso
  - Tanto para procesos explícitos como implícitos
  - Cuando hay múltiples *drivers* se usa una función de resolución

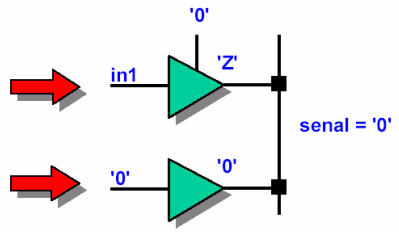
```
process (in1)
begin
        senal <= in1;
end process;

senal <= in2;
```



Inferencia triestado

- Cuando se quiere que un *driver* de la señal se quede en alta impedancia, se le asigna a la señal el valor 'Z'
  - Es válido para el tipo std_logic
- Igual que ocurre en la realidad, el estado de la señal lo fijará el *driver* que no esté en alta impedancia

```
senal <= in1 when ena='0'
         else 'Z';
senal <= '0';
```



Ejemplos de inferencia de buffer triestado

- Con asignación condicional

```
a_out <= a when enable_a='1' else 'Z';
b_out <= b when enable_b='1' else 'Z';
```

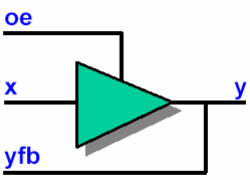
- Con un proceso

```
process (enable_a, a_out)
begin
        if (enable_a = '0') then
                a_out <= a;
        else a_out <= 'Z';
end process;
```

Soluciones bidireccionales

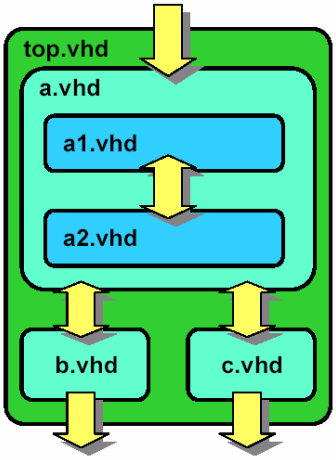
- En este caso la señal tiene *drivers* externos, fuera de la entidad

```
entity bufoe is port  (
            x, oe : in std_logic;
            y: inout std_logic;
            yfb : out std_logic);
end bufoe;

architecture simple of bufoe is
begin
            y <= x when oe='1'  else 'Z';
            yfb <= y;
end simple;
```



Diseño jerárquico

- Componentes pequeños son utilizados por otros más grandes
- Es fundamental para la reutilización de código
- Permite mezclar componentes creados con distintos métodos de diseño: VHDL, Verilog, esquemáticos
- Genera diseños más legibles y más portables
- Necesario para estrategias de diseño *top-down* o *botton-up*



Utilización de componentes

- Antes de poder usar un componente, se debe declarar
  - Especificar sus puertos (PORT)
  - Especificar parámetros (GENERIC)
- Una vez instanciado el componente, los puertos de la instancia se conectan a las señales del circuito usando PORT MAP
- Los parámetros se especifican usando GENERIC MAP
- La declaración de los componentes se puede hacer en un PACKAGE
  - Para declarar el componente, sólo habrá que importar el PACKAGE
  - La declaración de los componentes no aporta nada al lector

Ejemplo de diseño jerárquico: Top-Level

```
entity mux4to1 is port ( a,b,c,d : in std_logic;
                         sel : in std_logic_vector(1 downto 0);
                         o : out std_logic);
end mux4to1;

use  work.mymuxpkg.all;

architecture  InstComp of mux4to1 is
          signal r: std_logic_vector(1 downto 0);
begin
          mux0: mux2to1 port map  (i0 => a, i1 => b, s => sel(0), o => r(0));
          mux1: mux2to1 port map  (i0 => c, i1 => d, s => sel(0), o => r(1));
          mux2: mux2to1 port map  (r(0), r (1), sel(1), o);
end  InstComp;
```

Declaración del componente en un package

Asociación por nombre

Asociación por posición

Ejemplo de diseño jerárquico: componente inferior

```
entity mux2to1 is port ( i0, i1, s : in std_logic;
                         o : out std_logic);
end mux2to1;

architecture  archmux2to1 of mux2to1 is
begin
            o <= (i0 and not s) or (i1 and s);
end  archmux2to1;
```

Descripción del componente de nivel inferior

```
package mymuxpkg is
   component mux2to1 port (
                              i0,i1,s : in std_logic;
                               o : out std_logic);
   end component;
end mymuxpkg;
```

Creación del paquete

Instanciación repetitiva y condicional:GENERATE

- La instrucción GENERATE permite generar código iterativamente o condicionalmente
- Generación iterativa: Ideal para circuitos repetitivos como arreglos de componentes

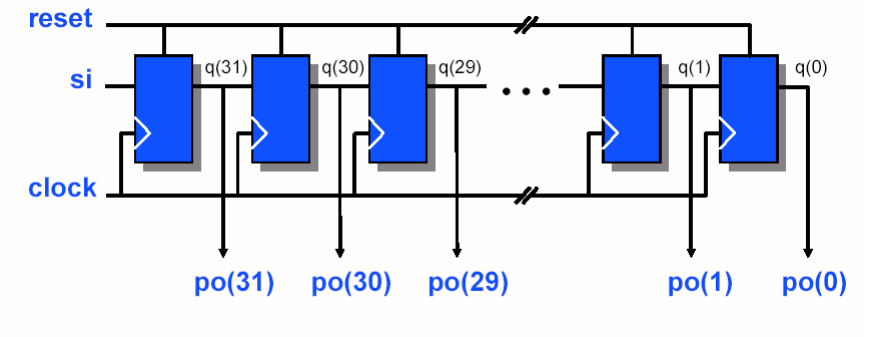
```
etiqueta: for <parámetro> in <rango> generate
            sentencias concurrentes
end generate;
```

- Generación condicional: Menos usada, por ejemplo para el primer elemento de un arreglo

```
etiqueta: if <condición> generate
            sentencias concurrentes
end generate;
```

Ejemplo de estructura repetitiva

Conversor serie paralelo de 32 bits



Ejemplo: Solución con GENERATE

```
entity sipo is port ( clk, rst, si : in std_logic;
                      po : out std_logic_vector(31 downto 0));
end sipo;

use  work.mypkg.all;

architecture  archsipo of sipo is
        signal q: std_logic_vector(31 downto 0);
begin
        gen: for i in 0 to 30 generate
                nxt: dsrff port map (clk, rst, q(i+1), q(i));
        end generate;
        beg: dsrff port map (clk, rst, si, q(31));
        po <= q;
end archsipo;
```

Configuración

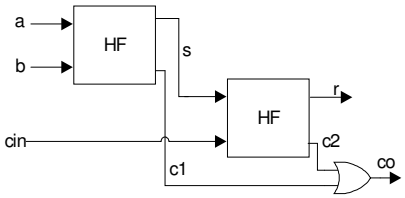
- Una configuración es una unidad de diseño
- Se usan para realizar asociaciones dentro de los modelos:
  - Asociar entidad con arquitectura
  - En la instanciación de un componente asociarlo a una entidad y arquitectura
- Muy utilizados en entornos de simulación: Una manera rápida y flexible de probar distintas alternativas de diseño
- Limitada o no soportada en entorno de síntesis

```
configuration <identificador> of <nombre_entidad> is
        for <nombre_arquitectura>
        end for
end configuration;
```

Aplicándolo a un componente en particular: for *nombre_instancia*: *nom_comp* use ...

Aplicándolo a todas las instancias: for all: *nom_comp* use entity *work.nom_entidad* (*nom_arq*)

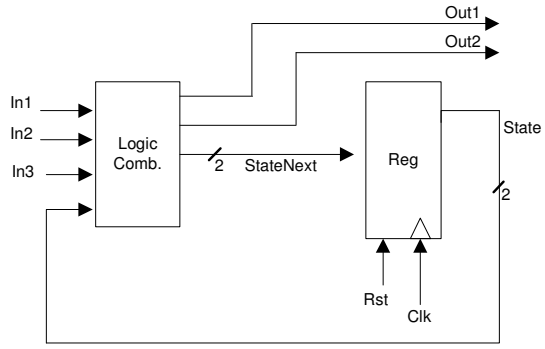
Configuración: Ejemplo




```
entity FullAdder is port (a,b,cin: in std_logic;
                          r,co: out std_logic);
end FullAdder;
architecture structural of FullAdder is
        signal s,c1,c2: std_logic;
begin
        hf0: HalfAdder port map (a,b,s,c1);
        hf1: HalfAdder port map (cin,s,r,c2);
        u2:  or_gate port map (c1,c2,co);
end structural;
```

```
configuration a_config of FullAdder is
    for structural
        for all: HalfAdder use entity work.HalfAdder (algorithmic);
        end for;
        for u2: or_gate  use entity work.or_gate (behavioral);
        end for;
    end for;
end a_config;
```

Metodología: Diseño de circuitos secuenciales



Metodología: Diseño de circuitos secuenciales

- Utilización de Subtipos
  - Definición de estados
- Tres bloques funcionales
  - Lógica combinacional: Decisión de cambio de estado
  - Registros: mantienen el estado
  - Lógica combinacional: Definición de salidas

```
architecture rtl of fsm is
  type TypeState is (St0, St1, St2, St3);
  signal State, StateNext :TypeState;
  signal in1, in2, in3: bit;
  signal out1, out2: bit;
  signal rst, clk: bit;

                . . .
```

Metodología: Diseño de circuitos secuenciales

- Utilización de Subtipos
  - Definición de estados
- Tres bloques funcionales
  - Lógica combinacional: Decisión de cambio de estado
  - Registros: mantienen el estado
  - Lógica combinacional: Definición de salidas

```
begin
LogComp: process (State, in1, in2, in3)
        begin
          case State is
              when St0 => out1 <= '0';
                          out2 <= '0';
                          StateNext <= St1;
              when St1 => out1 <= '1';
                          if (in1='1') then
                            StateNext <= St2;
                          else
                            StateNext <= St3;
                          end if;
              when St2 =>
                          . . .
              when St3 =>
                          . . .
          end case;
        end process;
```

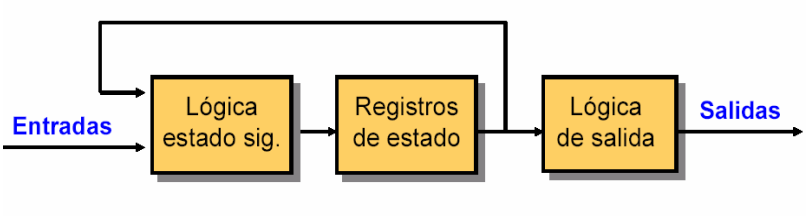

Metodología: Diseño de circuitos secuenciales

- Utilización de Subtipos
  - Definición de estados
- Tres bloques funcionales
  - Lógica combinacional: Decisión de cambio de estado
  - Registros: mantienen el estado
  - Lógica combinacional: Definición de salidas

```
PReg: process (clk, rst)
begin
      if (rst='1') then
            State <= St0;
      elsif (clk´event and clk='1') then
            State <= StateNext;
      end if;
end process;
```

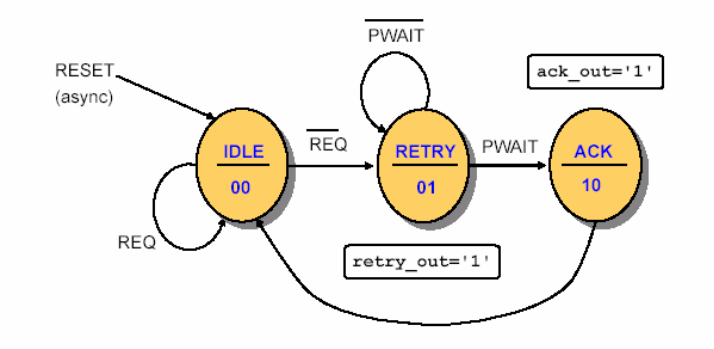
FSM: Máquinas de Moore

- FSM MOORE: Una máquina de estados en la que las salidas cambian solo cuando cambia el estado
- Las salidas son decodificadas a partir del valor de los estados



Ejemplo: Generador “wait states”

Diagrama de estados



Ejemplo: Declaración de la entidad

```
libary ieee;
use ieee.std_logic_1164.all;

entity Moore is port (
            rst, clk : in std_logic;
            req, pwait : in std_logic;
            retry_out, ack_out: out std_logic);
end Moore;
```

La declaración de entidad es la misma para todas las mplementaciones

Ejemplo: Solución 1

Salidas combinacionales decodificadas a partir de los estados

```
architecture  archMoore of Moore is
        type TypeStates is (idle, retry, ack);
        signal wait_gen: TypeStates;
begin
        fsm: process (clk, rst)
        begin
                if (rst='1') then
                        wait_gen <= idle;
                elsif (clk´event and clk='1') then
                        case wait_gen is
                                when idle =>
                                        if  req='0'  then wait_gen <= retry;
                                        else wait_gen <= idle;
                                        end if;
```

Ejemplo: Solución 1

```
                                        when retry =>
                                                if  pwait='1'  then wait_gen <= ack;
                                                else wait_gen <= retry;
                                                end if;

                                        when ack =>  wait_gen <= idle;

                                        when others => wait_gen <= idle;
                                end case;
                        end if;
                end process;

                retry_out <= '1' when (wait_gen=retry) else '0';
                ack_out <= '1' when (wait_gen=ack) else '0';

end archMoore;
```


Ejemplo: Solución 2

Salidas registradas decodificadas desde el valor de los estados

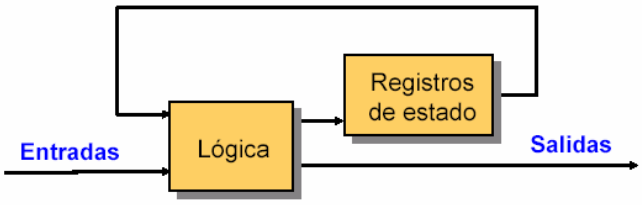
```
architecture  archMoore of Moore is
        type TypeStates is (idle, retry, ack);
        signal wait_gen: TypeStates;
begin
        fsm: process (clk, rst)
        begin
                if (rst='1') then
                        wait_gen <= idle;
                        retry_out <= '0';
                        ack_out <= '0';
                elsif (clk´event and clk='1') then
                        retry_out <= '0';
```

Ejemplo: Solución 2

```
case wait_gen is
        when  idle => if  req='0'  then  wait_gen <= retry;
                                ack_out <= '1';
                                ack_out <= '0';
                        else
                                wait_gen <= idle;
                                ack_out <= '0';
                        end if;
        when  retry =>  if  pwait='1'  then wait_gen <= ack_out;
                                ack_out <= '1';
                        else
                                wait_gen <= retry_out;
                                retry_out <= '1';
                                ack_out <= '0';
                        end if;
        when ack =>  wait_gen <= idle;
                        ack_out <= '0';
        when others => wait_gen <= idle;
                        ack_out <= '0';
    end case;
end if;
```

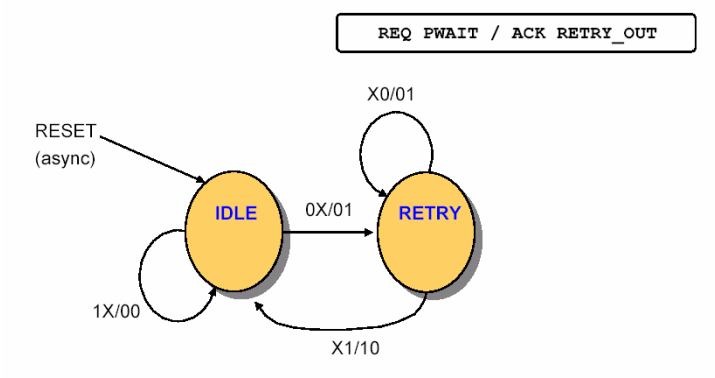
FSM de Mealy

- Las salidas cambian por un cambio de estado o por un cambio en el valor de las entradas
  - Hay que tener mucho cuidado con las entradas asíncronas



Ejemplo: Generador “wait states”

Diagrama de estados



Ejemplo: Solución

```
architecture  archMealy of Mealy is
        type TypeStates is (idle, retry);
        signal wait_gen: TypeStates;
begin
        fsm: process (clk, rst)
        begin
                if (rst='1') then
                        wait_gen <= idle;
                elsif (clk´event and clk='1') then
                        case wait_gen is
                                when idle => if  req='0'  then wait_gen <= retry;
                                             else wait_gen <= idle;
                                             end if;

                                when retry => if  pwait='1'  then wait_gen <= idle;
                                              else wait_gen <= retry;
                                              end if;
```

Ejemplo: Solución

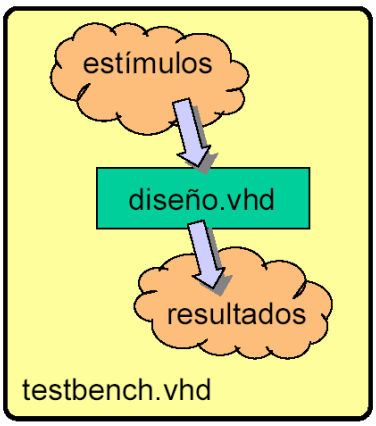
```
                                    when others => wait_gen <= idle;
                            end case;
                    end if;
            end process;

            retry_out <= '1' when (wait_gen=retry and pwait='0') or
                                  (wait_gen=idle and req='0')  else '0';

            ack_out <= '1' when (wait_gen=retry and pwait='1') else '0';

end archMealy;
```


Verificación con TestBenches

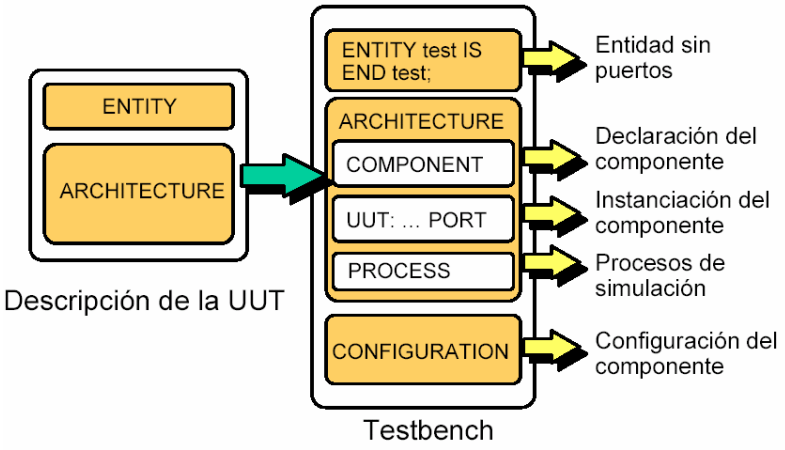
- Un diseño sin verificación no está completo:
  - Existen muchas maneras de verificar, la más utilizada es el banco de pruebas: TestBench
- Simular básicamente es:
  - Generar estímulos
  - Observar resultados
- Un TestBench es un código VHDL que automatiza estas dos operaciones
- Los TestBenches no se sintetizan
  - Se puede utilizar un VHDL algorítmico
  - Usualmente con algoritmos secuenciales

¿Cómo hacer un TestBench?

- Instanciar el diseño que vamos a verificar
  - El TestBench será el nuevo top-level
  - Será una entidad sin *ports*
- Escribir el código que:
  - Genera los estímulos
  - Observa los resultados
  - Informa al usuario



Instanciando la unidad bajo test (UUT)



Generando estímulos

- Dar valores a las señales que van hacia las entradas de la UUT
- En síntesis no tiene sentido el tiempo
  - En los testbench el tiempo es el concepto principal
- Asignación concurrente

```
s <= '1',
     '0' after 20 ns,
     '1' after 30 ns;
```

- Asignación secuencial

```
s <= '1';
wait for 20 ns,
s <= '0';
wait for 30 ns,
s <= '1';
```


Observando señales con Assert

- assert se usa para comprobar si se cumple una condición: equivalente a *IF (not condicion)*

```
assert  <condicion> report  string severity nivel;
```

- Tras report se añade una cadena de texto que se muestra si no se cumple la condición
- severity puede tener cuatro niveles
  - NOTE
  - WARNING
  - ERROR (nivel por defecto sino se incluye severity)
  - FAILURE

Algoritmo básico para los Testbenches

Algoritmo elemental de verificación:

- Dar valores a las señales de entrada a la UUT
- Esperar con wait for
- Comprobar los resultados con assert
- Volver a dar valores a las señales de entrada a la UUT
- y repetir ...

Ejemplo de código

```
entity dff_tb is
end dff_tb;

architecture  tb_arch of dff_tb is
        component dff is port (...) end component;
        signal d,c,q: std_logic;
begin
        uut: dff port map (d =>d, clk => c, q=>q);

        prcess
        begin
                c <= '0'; d <= '0';
                wait for 10 ns;
                c <= '1';
                wait for 10 ns;
                assert q=d report "falla" severity failure;
        end process;
end tb_arch;
```

Procedimientos

- VHDL permite definir procedimientos (subrutinas)

```
procedure  nombre (clase parámetro: dir tipo, ... ) is
            declaraciones
begin
            instrucciones secuenciales
end nombre;
```

- La clase de los parámetros pueden ser: variable, constant, signal
- La dirección: in, inout, out
- Los procedimientos se pueden declarar en la arquitectura o en un proceso, y se llaman desde un proceso o concurrentemente

Interesante para encapsular tareas repetitivas en la simulación

Ejemplo de código

```
architecture  tb_arch of dff_tb is
            (...)
            procedure send_clock_edge(signal c: out std_logic) is
            begin
                        c <= '0'; wait for 10 ns;
                        c <= '1'; wait for 10 ns;
            end send_clock_edge;
begin
            uut: dff port map (d =>d, clk => c, q=>q);
            prcess
            begin
                        d <= '0';
                        send_clock_edge(c);
                        assert q=d report "falla" severity failure;
            end process;
end tb_arch;
```

Acceso a archivos

- Las simulaciones más potentes trabajan sobre archivos
  - Simulación de un multiplicador que escribe los resultados en un archivo de texto
  - Testbench para un microprocesador que lee un programa ensamblador de un archivo, lo ensambla y lo ejecuta
- Acceso básico: **std.textio**
  - Archivos de texto
  - Acceso línea a línea: readline y writeline
  - Dentro de una línea los campos se procesan con: read y write
- Acceso específico para std_logic: **ieee.std_logic_textio**

Instrucciones para acceder a archivos

```
file archivo_estimulos: text is in "Stim.txt";
```

- Especificar el archivo

```
variable  linea: line;
          . . .
readline (archivos_estimulos, linea);
```

- Leer una línea

```
variable  opcode: string (2 downto 0);
          . . .
read (linea, opcode);
```

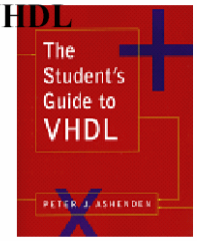
- Leer un campo de una línea

```
write (linea, resultado);
writeline (archivo_resultados, linea);
```

- Escribir

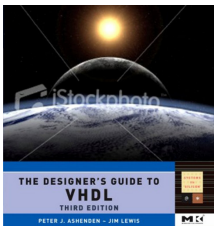
VHDL: Bibliografía

The Student`s Guide to VHDL



Peter J. Ashenden
Ed: Morgan Kaufmann
ISBN 1558605207

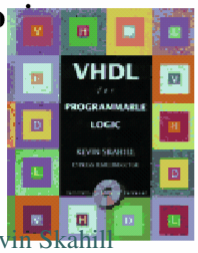
The Designer's Guide to VHDL



Peter Ashenden – Jim Lewis
Ed:Morgan Kaufmann
ISBN: 978-0120887859

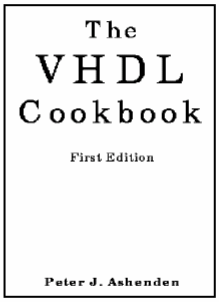
VHDL: Bibliografía

VHDL for Programmable Lo



Kevin Skahill
Ed: AddisonWesley
ISBN 0 201- 89573-0

VHDL Cookbook



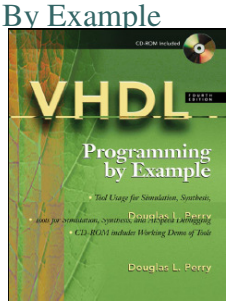
infaut.et.uni-magdeburg.de/~ks/VHDL/

VHDL: Bibliografía

VHDL 2008 Just the New Stuff

Peter Allenden – Jim Lewis
Ed:Morgan Kaufmann
ISBN 978-0-12-374249-0

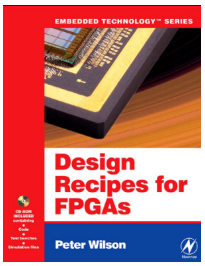
VHDL Programming By Example



Douglas L. Perry
Ed: McGraw-Hill
ISBN: 978-0-07-1409544

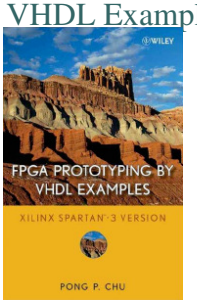
VHDL: Bibliografía Avanzada

Design Recipes for FPGAs



Peter Wilson
Ed: Elsiever
ISBN: 978-0-7506-6845-3

FPGA Prototyping By VHDL Examples



Pong P. Chu
Ed: Wiley-Interscience
ISBN 978-0-470-18531-5

VHDL: Enlaces de interés

http://fpgalibre.sourceforge.net/
Portal argentino de herramientas y recursos libres para desarrollar con FPGAs

http://www.opencores.org/
Repositorio de cores gratuitos sintetizables.

http://www.xilinx.com/
Uno de los principales fabricantes de FPGAs.

http://www.vhdl-online.de/tutorial/
http://esd.cs.ucr.edu/labs/tutorial/
Tutoriales online de VHDL