Introducción a VHDL
Circuitos Combinacionales

Introducción a la Microfabricación y las FPGA

Instituto Balseiro

4 de Agosto 2014

Hoy veremos

- Nuestros primeros diseños. Usando VHDL en nivel de gates (compuertas lógicas básicas).
- Nuestros primeros testbenches y simulaciones en nivel de comportamiento (no timing accurate).
- Todo el flujo desde el diseño, codificación, simulación, síntesis y configuración de la FPGA.
- Primeros diseños en nivel RTL. Circuitos combinacionales. Mas tipos de datos. Mas operadores. Ruteo.
- Procesos.

Igualdad de 1 bit

input		**output**
i0	*i1*	*eq*
0	0	1
0	1	0
1	0	0
1	1	1

- A nivel de gates, o sea, expresado utilizando sólo compuertas lógicas básicas (*and*, *not*, *or*, *xor*).
- Usamos suma de productos: *eq* = *i0.i1* + *i0'.i1'*

Primer código

```
 1  library IEEE;
 2  use IEEE.STD_LOGIC_1164.ALL;
 3
 4  entity eq1bit is
 5
 6     port (
 7        i0,i1: in std_logic;
 8        eq1: out std_logic
 9     );
10  end eq1bit;
11
12  architecture Behavioral of eq1bit is
13     signal p0,p1:std_logic;
14
15  begin
16     -- suma de dos productos
17     eq1 <= p0 or p1;
18     -- terminos producto
19     p0 <= (not i0) and (not i1);
20     p1 <= i0 and i1;
21  end Behavioral;
```

- VHDL es case insensitive.
- – indica comentarios.

Library y package

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity eq1bit is
5
6     port (
7        i0,i1: in std_logic;
8        eq1: out std_logic
9     );
10 end eq1bit;
11
12 architecture Behavioral of eq1bit is
13    signal p0,p1:std_logic;
14
15 begin
16    -- suma de dos productos
17    eq1 <= p0 or p1;
18    -- terminos producto
19    p0 <= (not i0) and (not i1);
20    p1 <= i0 and i1;
21 end Behavioral;
```

- Nos permiten agregar tipos, operadores, funciones, etc.
- VHDL es *fuertemente tipado*.
- Tiene muchos tipos, pero los *sintetizables* no son tantos!

Entity

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity eq1bit is
5
6     port (
7        i0,i1: in std_logic;
8        eq1: out std_logic
9     );
10 end eq1bit;
11
12 architecture Behavioral of eq1bit is
13    signal p0,p1:std_logic;
14
15 begin
16    -- suma de dos productos
17    eq1 <= p0 or p1;
18    -- terminos producto
19    p0 <= (not i0) and (not i1);
20    p1 <= i0 and i1;
21 end Behavioral;
```

- Declaración de las señales I/O del circuito: in, out, inout.
- `std_logic`. Sintetizables '0', '1', 'Z', Simulación 'U', 'X'.
- Operaciones lógicas.

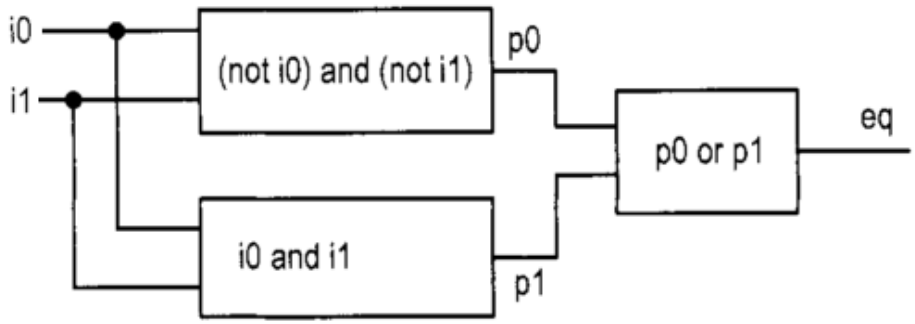
Arquitectura

```
 1  library IEEE;
 2  use IEEE.STD_LOGIC_1164.ALL;
 3
 4  entity eq1bit is
 5
 6     port (
 7        i0,i1: in std_logic;
 8        eq1: out std_logic
 9     );
10  end eq1bit;
11
12  architecture Behavioral of eq1bit is
13     signal p0,p1:std_logic;
14
15  begin
16     -- suma de dos productos
17     eq1 <= p0 or p1;
18     -- terminos producto
19     p0 <= (not i0) and (not i1);
20     p1 <= i0 and i1;
21  end Behavioral;
```

- Define la operación del circuito. Puedo tener más de una arquitectura por entidad.
- Sección de declaración (signal, constant, etc)
- Descripción principal: tres **sentencias concurrentes**

Representación gráfica

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity eq1bit is
5
6     port (
7        i0,i1: in std_logic;
8        eq1: out std_logic
9     );
10 end eq1bit;
11
12 architecture Behavioral of eq1bit is
13    signal p0,p1:std_logic;
14
15 begin
16    -- suma de dos productos
17    eq1 <= p0 or p1;
18    -- terminos producto
19    p0 <= (not i0) and (not i1);
20    p1 <= i0 and i1;
21 end Behavioral;
```



Igualdad de 2 bits

input				**output**
a0	*a1*	*b0*	*b1*	*eqab*
0	**0**	**0**	**0**	**1**
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	**1**	**0**	**1**	**1**
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	**0**	**1**	**0**	**1**
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	**1**	**1**	**1**	**1**

Igual que antes, usamos suma de productos:
$eqab = a0'.a1'.b0'.b1' + a0.a1'.b0.b1' + a0'.a1.b0'.b1 + a0.a1.b0.b1'$

Codigo VHDL - suma de productos

```
 1  library IEEE;
 2  use IEEE.STD_LOGIC_1164.ALL;
 3
 4  entity eq2bit is
 5     port(
 6        a,b: in std_logic_vector(1 downto 0);
 7        aeqb: out std_logic
 8     );
 9  end eq2bit;
10
11  architecture Behavioral of eq2bit is
12     signal p0,p1,p2,p3 : std_logic;
13  begin
14     -- suma de productos
15     aeqb <= p0 or p1 or p2 or p3;
16     -- productos
17     p0 <= (not a(0)) and (not a(1)) and
18           (not b(0)) and (not b(1));
19     p1 <= (not a(0)) and a(1) and
20           (not b(0)) and b(1);
21     p2 <=  a(0) and (not a(1)) and
22           b(0) and (not b(1));
23     p3 <=  a(0) and a(1) and b(0) and b(1);
24  end Behavioral;
```

Lo único nuevo es el tipo de datos `std_logic_vector`. Como su nombre lo indica, un vector de `std_logic`.

Descripción estructural

- Una misma entidad puede tener muchas arquitecturas diferentes.
- Cada arquitectura pueden sintetizar a una implementación diferente, o puede ser que dos arquitecturas sean diferentes maneras de expresar la misma implementación.
- Arquitectura estructural:
  - Descripción basada en dividir la entidad en diferentes componentes.
  - *Instanciamos* módulos ya existentes, asignando sus puertos a señales del diseño.

Descripción estructural - VHDL

```
26  architecture struct_arch of eq2bit is
27     signal e0, e1 : std_logic;
28  begin
29     -- instanciamos dos comparadores de 1 bit
30     eq_bit0_unit: entity work.eq1bit(Behavioral)
31        port map(i0 => a(0), i1 => b(0), eq1 => e0);
32
33     eq_bit1_unit: entity work.eq1bit(Behavioral)
34        port map(i0 => a(1), i1 => b(1), eq1 => e1);
35
36     -- a y b son iguales si sus bits individuales
37     -- son iguales
38     aeqb <= e0 and e1;
39
40  end struct_arch;
```

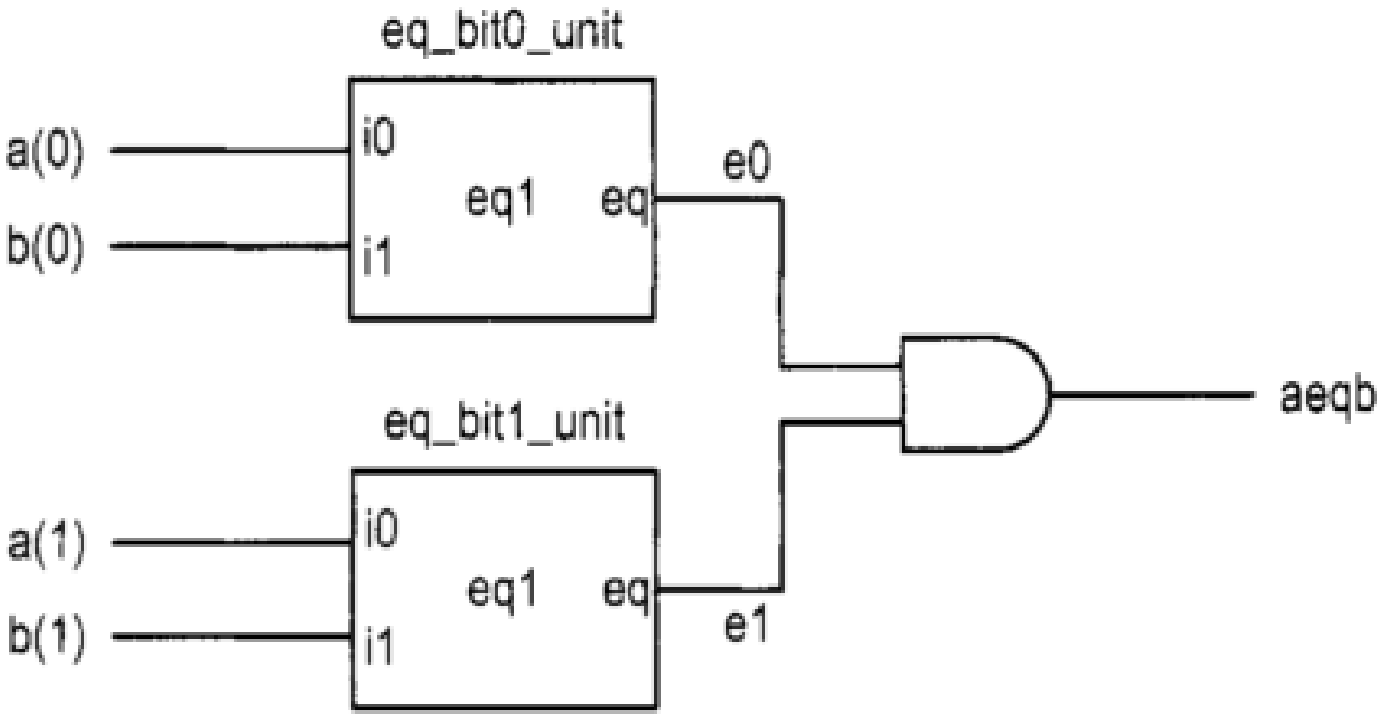
- El módulo `eq1bit` tiene que estar definido en el proyecto.
- `work`: librería donde están definidos todos los módulos del proyecto.
- Entre paréntesis va la arquitectura que queremos usar (opcional si el componente tiene una sóla arquitectura)

Descripción estructural VHDL 87

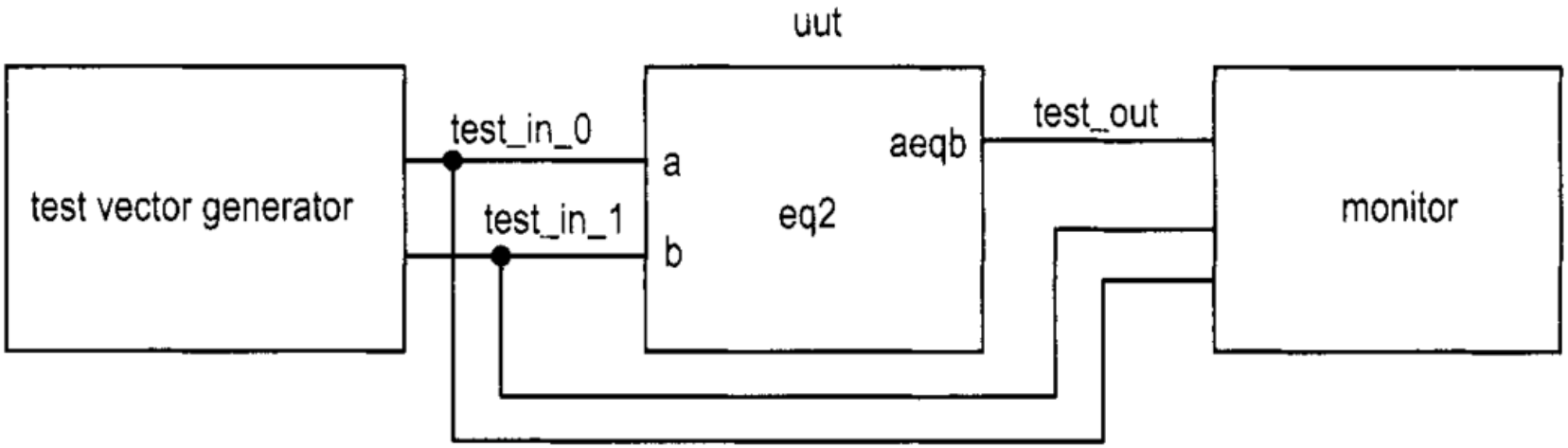
```
architecture struct_arch_vhdl87 of eq2bit is

   -- declaramos los componentes que vamos a
   -- usar
   component eq1bit is
      port(
         i0,i1: in std_logic;
         eq1:out std_logic
      );
   end component;

   signal e0, e1 : std_logic;
begin
   -- instanciamos dos comparadores de 1 bit
   eq_bit0_unit: eq1bit
      port map(i0 => a(0), i1 => b(0), eq1 => e0);

   eq_bit1_unit: eq1bit
      port map(i0 => a(1), i1 => b(1), eq1 => e1);

   -- a y b son iguales si sus bits individuales
   -- son iguales
   aeqb <= e0 and e1;

end struct_arch_vhdl87;
```

Declarar *explícitamente* los componentes a utilizar en la parte de declaraciones de la arquitectura.

Descripción gráfica



Testbench


uut
test vector generator
test_in_0
test_in_1
a
b
eq2
aeqb
test_out
monitor

Testbench en VHDL

```
1  LIBRARY ieee;
2  USE ieee.std_logic_1164.ALL;
3
4  ENTITY test_eq1bit IS
5  END test_eq1bit;
6
7  ARCHITECTURE behavior OF test_eq1bit IS
8      -- Component Declaration for the
9      -- Unit Under Test (UUT)
10     COMPONENT eq1bit
11     PORT(
12          i0 : IN  std_logic;
13          i1 : IN  std_logic;
14          eq1 : OUT  std_logic
15         );
16     END COMPONENT;
17
18    --Inputs
19    signal i0 : std_logic := '0';
20    signal i1 : std_logic := '0';
21
22    --Outputs
23    signal eq1 : std_logic;
24
25 BEGIN
```

- Simulación de Comportamiento (NO es timing accurate).
- Inclusión de librerias y definición de entidad igual que antes.
- Arquitectura: instanciación del componente, y definición de señales para estimular el diseño y chequear los resultados.

Testbench en VHDL - cont

```
27    -- Instantiate the Unit Under Test (UUT)
28    uut: eq1bit PORT MAP (
29           i0 => i0,
30           i1 => i1,
31           eq1 => eq1
32         );
33
34    -- Stimulus process
35    stim_proc: process
36    begin
37
38      -- PRIMER CASO : vector de test (0,0)
39      -- 1) genero vector de test y estimulo el diseño
40      i0 <= '0';
41      i1 <= '0';
42      -- 2) espero el tiempo de propagación
43      wait for 1 ns;
44      -- 3) El monitor chequea si la salida es correcta
45      assert eq1 <= '1' report "Igualdad falló en caso i0 =0, i1=0"
46             severity failure;
                                  ...
65      -- FINAL: si llegué hasta acá, está todo bien.
66      -- Hago assert para terminar simulacion
67      assert false report "Simulación terminó OK" severity failure;
68    end process;
69  END;
```

Manos a la obra

¡ESO!

Hasta ahora vimos

- Introducción a FPGA: qué son, para qué se usan, arquitecturas, cómo se programan.
- Introducción a VHDL.
  - VHDL para síntesis: nuestro primer programa a nivel de Gates.
  - VHDL para simulación: nuestro primer testbench.
- Herramientas:
  - *ISE*: Crear proyecto, fuente, ucf (user constraint file) y todo el proceso de síntesis hasta generar el archivo de programa (bitstream).
  - *ISim*: Usando el ISE, crear un testbench y simularlo con el ISE Simulator (ISim).
  - *Digilent Adept*: Configurar la FPGA.

Diseño y codificación

- Diseño a nivel de gate.
  - Hacemos tabla de verdad y utilizamos suma de productos.
  - Si podemos dividir en bloques mas pequeños (o ya tenemos bloques mas pequeños): instanciación.
- Library y package. Entidad. Arquitectura.
- Todas las sentencias que vimos hasta ahora son **concurrentes**

Ahora veremos: Register Transfer Level inicial

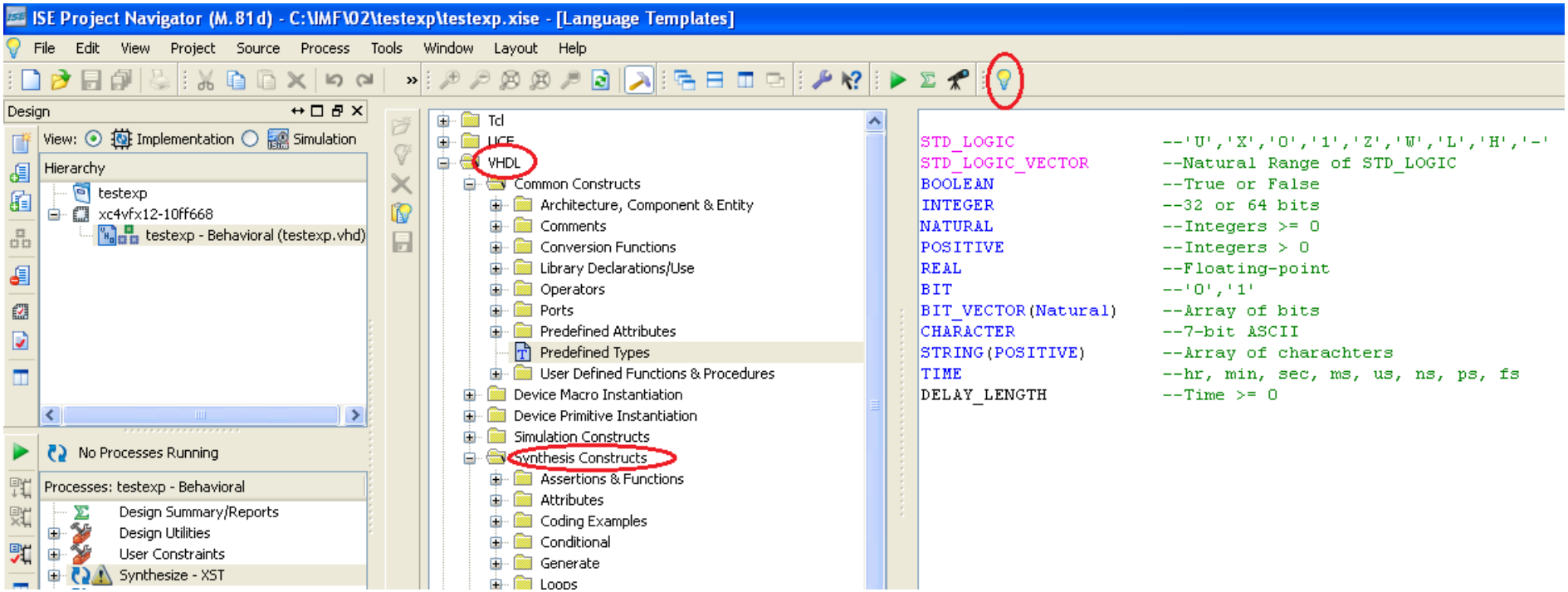
- RTL: Metodologia de diseño pensando la manipulación de datos como transferencia entre registros.
- Vamos a ver las componentes de la Logica Combinacional a este nivel de abstracción (mayor que el de gate).
- Circuitos combinacionales: las salidas sólo dependen de las entradas (no tienen estado/memoria).
- Nuevos tipos de datos. Nuevos operadores. Ruteo de señales.

Tipos nativos

```
STD_LOGIC               --'U','X','0','1','Z','W','L','H','-'
STD_LOGIC_VECTOR        --Natural Range of STD_LOGIC
BOOLEAN                 --True or False
INTEGER                 --32 or 64 bits
NATURAL                 --Integers >= 0
POSITIVE                --Integers > 0
REAL                    --Floating-point
BIT                     --'0','1'
BIT_VECTOR(Natural)     --Array of bits
CHARACTER               --7-bit ASCII
STRING(POSITIVE)        --Array of charachters
TIME                    --hr, min, sec, ms, us, ns, ps, fs
DELAY_LENGTH            --Time >= 0
```

- `real`, `time`, `delay_length` NO sintetizables.
- No todas las operaciones sobre estos tipos son sintetizables.

Referencia del lenguaje en ISE



Capitulo 21 del Ashenden.

Operadores nativos

Operator	Description	Data type of operands	Data type of result
`a ** b`	exponentiation	`integer`	`integer`
`a * b`	multiplication		
`a / b`	division	*integer type for constants and array boundaries, not synthesis*	
`a + b`	addition		
`a - b`	subtraction		
`a & b`	concatenation	1-D array, element	1-D array
`a = b`	equal to	any	`boolean`
`a /= b`	not equal to		
`a < b`	less than	scalar or 1-D array	`boolean`
`a <= b`	less than or equal to		
`a > b`	greater than		
`a >= b`	greater than or equal to		
not `a`	negation	`boolean`, `std_logic`, `std_logic_vector`	same as operand
`a` **and** `b`	and		
`a` **or** `b`	or		
`a` **xor** `b`	xor		

- Tipos nativos `integer`, `natural`: 32 bits.
- En general queremos mayor manejo de el numero de bits en cada señal: usar `signed` o `unsigned` de `IEEE.numeric_std`

Operadores overloaded signed y unsigned

Overloaded operator	Description	Data type of operands	Data type of result
a * b a + b a - b	arithmetic operation	unsigned, natural signed, integer	unsigned signed
a = b a /= b a < b a <= b a > b a >= b	relational operation	unsigned, natural signed, integer	boolean boolean

```
library ieee;
use ieee.numeric.std.all;
signal a : unsigned(3 downto 0);
```

Casting

Data type of a	To data type	Conversion function/type casting
unsigned, signed	std_logic_vector	std_logic_vector(a)
signed, std_logic_vector	unsigned	unsigned(a)
unsigned, std_logic_vector	signed	signed(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

Ruteo

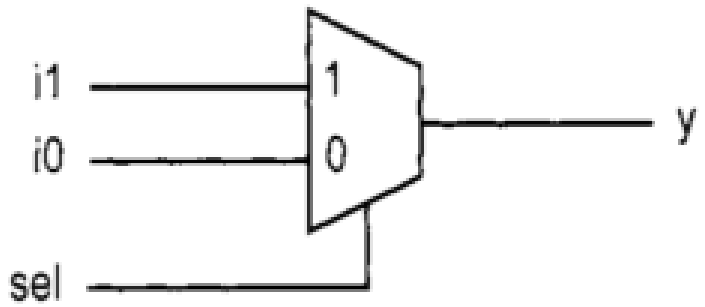
- Hasta ahora, sabemos trabajar con algunos tipos sintetizables y realizar operaciones lógicas, artiméticas y relacionales.
- Nos falta ver cómo seleccionar de diferentes posibles resultados, similar a las sentencias **if** y **case**.
- En hardware, esto se hace *evaluando* todos los posibles resultados en paralelo y *ruteando* el seleccionado a la salida. Esta selección se implementa con *multiplexores*.
- Vamos a ver dos maneras de escribir en VHDL sentencias concurrentes que generan dos modos distintos de ruteo: priority network y selected signal assignment.

Conditional Signal Assignment - Priority Network

Implica una prioridad en la asignación. Parecido al if. Sintaxis:

```
signal_name <= value_expr_1 when boolean_expr_1 else
               value_expr_2 when boolean_expr_2 else
                  . . .
               value_expr_n;
```

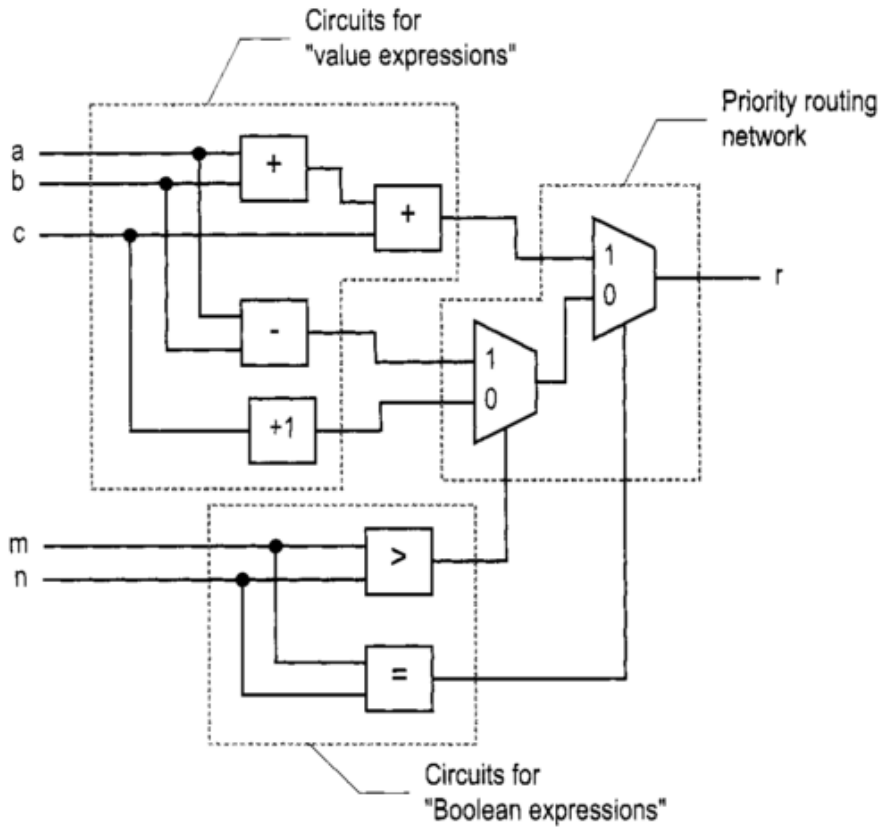
Multiplexor:



sel	y
0 (false)	i0
1 (true)	i1

Conditional Signal Assignment - Priority Network

```
r <= a + b + c when m = n else
     a - b     when m > n else
     c + 1;
```



- Expresiones de valor y booleanas se evalúan *concurrentemente*.
- Las expresiones booleanas setean las señales de selección de los multiplexores.
- Cada clausula `when-else` introduce un Mux2-1: cuántas mas cláusulas, más larga la cadena y por lo tanto mayores delays.

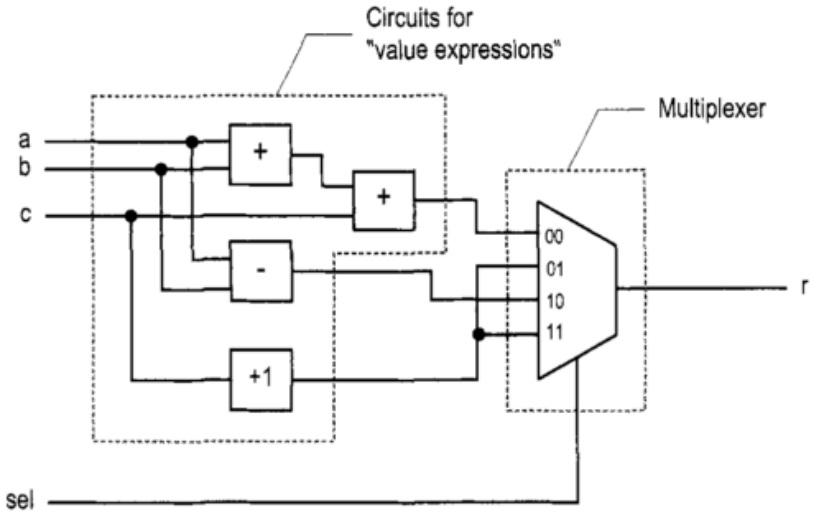
Selected signal assignment

```
with sel select
   sig <= value_expr_1 when choice_1,
          value_expr_2 when choice_2,
          value_expr_3 when choice_3,
          . . .
          value_expr_n when others;
```

- Parecido al case.
- Un `choice` tiene que ser un valor possible de `sel`.
- Tienen que estar todos los valores y ser mutuamente excluyentes. **others**.
- En general son `std_logic_vector` o tipos enumerados.

Selected signal assignment

```
signal sel: std_logic_vector(1 downto 0);
. . .
with sel select
   r <= a + b + c    when "00",
        a - b        when "10",
        c + 1        when others;
```



- Infiere un multiplexor de 2^2 en 1, con `sel` como señal de selección.
- El tamaño del multiplexor crece geométricamente con la cant de bits de `sel`.
- Nuevamente, las expresiones de valor se evalúan concurrentemente.

Comparación: Priority Encoder

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity prio_encoder is
   port(
      r: in std_logic_vector(4 downto 1);
      pcode: out std_logic_vector(2 downto 0)
   );
end prio_encoder;
```

```vhdl
architecture cond_arch of prio_encoder is
begin
   pcode <= "100" when (r(4)='1') else
            "011" when (r(3)='1') else
            "010" when (r(2)='1') else
            "001" when (r(1)='1') else
            "000";
end cond_arch;
```

```vhdl
architecture sel_arch of prio_encoder is
begin
   with r select
      pcode <= "100" when "1000"|"1001"|"1010"|"1011"|
                          "1100"|"1101"|"1110"|"1111",
               "011" when "0100"|"0101"|"0110"|"0111",
               "010" when "0010"|"0011",
               "001" when "0001",
               "000" when others;   -- r="0000"
end sel_arch;
```

Procesos

- Hasta ahora, todas las sentencias son concurrentes.
- Procesos: es una sentencia concurrente, pero dentro tiene una cantidad de sentencias secuenciales que describen su comportamiento.
- Sirven para simplificar la tarea de diseño. Pero hay que tener cuidado: nunca hay que perder de vista que aún estas "sentencias secuenciales" están sintetizando hardware concurrente.
- Siempre pensar en qué se está sintetizando.

```
process(sensitivity_list)
begin
   sequential statement;
   sequential statement;
   . . .
end process;
```

Ruteo con *if*

```
if boolean_expr_1 then
   sequential_statements;
elsif boolean_expr_2 then
   sequential_statements;
elsif boolean_expr_3 then
   sequential_statements;
. . .
else
   sequential_statements;
end if;
```

- Genera cascada de Mux2-1 como el conditional signal assignment.
- Son equivalentes si cada branch del if tiene una sola asignación a la misma señal.
- El *if* es mas potente: permite *cualquier número* y *cualquier tipo* de sentencias secuenciales en cada branch.
- Por ejemplo, permite nested ifs.

Ruteo con *case*

```
case sel is
   when choice_1 =>
      sequential statements;
   when choice_2 =>
      sequential statements;
   . . .
   when others =>
      sequential statements;
end case;
```

- Genera multiplexor similar al selected signal assignment.
- Son equivalentes si cada branch del case tiene una sola asignación a la misma señal.
- El *case* es mas potente: permite *cualquier número* y *cualquier tipo* de sentencias secuenciales en cada branch.
- Por ejemplo, permite nested cases.

Comparación: Priority Encoder

```
architecture case_arch of prio_encoder is
begin
   process(r)
   begin
      case r is
         when "1000"|"1001"|"1010"|"1011"|
              "1100"|"1101"|"1110"|"1111" =>
            pcode <= "100";
         when "0100"|"0101"|"0110"|"0111" =>
            pcode <= "011";
         when "0010"|"0011" =>
            pcode <= "010";
         when "0001" =>
            pcode <= "001";
         when others =>
            pcode <= "000";
      end case;
   end process;
end case_arch;
```

```
architecture if_arch of prio_encoder is
begin
   process(r)
   begin
      if (r(4)='1') then
         pcode <= "100";
      elsif (r(3)='1')then
         pcode <= "011";
      elsif (r(2)='1')then
         pcode <= "010";
      elsif (r(1)='1')then
         pcode <= "001";
      else
         pcode <= "000";
      end if;
   end process;
end if_arch;
```