Diseño Automático de sistemas

VHDL Avanzado

Prof. Pablo Sarabia Ortiz

UNIVERSIDAD
NEBRIJA

Objetivos Clase

- Entender los genéricos, tipos de datos y su uso práctico.
- Comprender el uso de las variables.
- Simplificar el código mediante el uso de procedimientos y funciones.
- Realizar diseños modulares y reusables mediante el diseño jerárquico.
- Entender el diseño paramétrico usando genéricos, y la generación automática de componentes.

Anuncio

- Examen Parcial: 16 Marzo 15:30: El examen constará de unos ejercicios de respuesta múltiple y unas breves cuestiones teórico/prácticas.
- Cambio en el calendario de prácticas:
  - **Grupo A**: 8 de Febrero, 22 de Febrero, 15 de Marzo (Online), **5 de Abril y 3 de Mayo (11:00).**
  - **Grupo B:** 15 de Febrero, 1 de Marzo, 22 de Marzo (Online), **5 de Abril (11:00) y 3 de Mayo.**

Bibliografia

- **Capítulos 2 y 8**
  Digital Systems Design Using VHDL (Second Edition), Charles H. Roth, Jr and Lizy Kurian John.

- **Cápitulos 3, 13, 14 y 15**

  Pong P. Chu (2006), RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability, Willey, 1st Edition

Contenidos

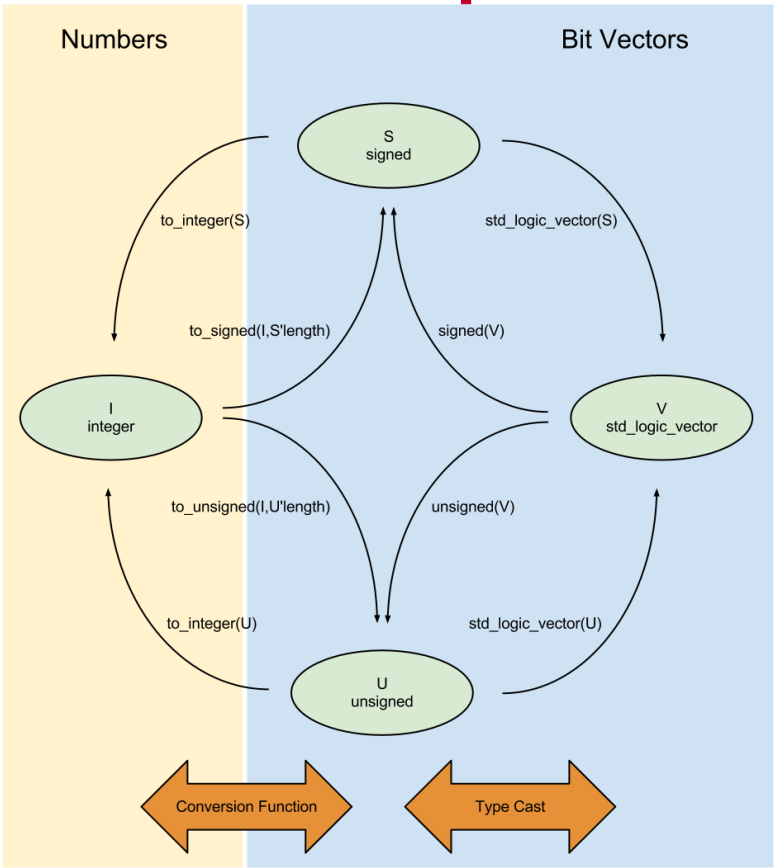
1. Tipos de datos y arrays
2. Variables y señales locales
3. Diseño jerárquico
4. Procedimientos y funciones
5. Creación de paquetes y librerías
6. Diseño paramétrico
7. Diseño a bajo nivel: Inferir e Instanciar

Introducción a las señales y los tipos de datos

Los datos en HW son agrupaciones de señales, no son lugares en la memoria (como en software normal).

Los tipos de datos que debemos utilizar (en elementos **sintetizables**) son: **signed, unsigned, std_logic** y **std_logic_vector**

Conversión entre los tipos de datos



Valores posibles en std_logic y std_logic_vector

Para reflejar mejor el carácter eléctrico de las señales en VHDL el standard IEEE std_logic_1164 (std_logic_vector y std_logic).

Los valores posibles son: U, X, 0, 1, Z, W, L, H, -.

- '0' y '1': señales validas (sintetizables).
- 'L' y 'H': señales débiles
- 'X' y 'W': Desconocido y desconocido débil, se da cuando hay señales contradictorias o señales indeterminadas.
- 'Z': Alta impedancia (sintetizable, se usa para las salidas de la FPGA en determinados buses).
- 'U': Sin inicializar.
- '-': Don't care

Atributos de un array

- Los arrays tienen atributos que pueden ser muy útiles en el diseño.
  - Los más usados son length, range, low y high.

```
32  signal refresh_counter : std_logic_vector(19 downto 0);
```

refresh_counter'left = 19
refresh_counter'low = 0
refresh_counter'range = 19 downto 0
refresh_counter'length = 20
refresh_counter'element = std_logic

refresh_counter'right = 0
refresh_counter'high = 19
refresh_counter'reverse_range = 0 downto 19
refresh_counter'ascending = false

Signed y Unsigned

- Son arrays de std_logic.
- Unsigned, el bit más significativo (MSB) es parte del número a representar ej: 1000 es 8 (siendo el 1 el MSB).
- Signed el bit más significativo es el que indica el signo ej: 1000 es -4 (siendo el 1 el MSB).

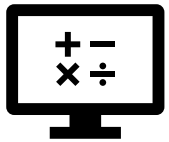
¿Qué valor tiene “1100”?

Operadores disponibles (un)signed

Los operadores disponibles son: ‘abs’, ‘*’, ‘/’, ‘mod’, ‘rem’, ‘+’ y ‘-’.

Los operadores relacionales son: ‘=‘, ‘/=‘, ‘<‘, ‘>’, ‘<=‘ y ‘>=’.

- Trata los array como números binarios ej: “0011” > “1000” es falsa para std_logic_vector, pero verdadera para un tipo signed.



¿Cuando usar variables?

- Es recomendable **evitar** su uso.
- Se declaran dentro de un proceso y son **locales** al mismo.
- Toman el valor de forma **inmediata**, pueden estar interconectadas.
- Se pueden utilizar para una variable temporal, nunca con un sentido físico.
- Pueden ser no sintetizables. Ej:

```
signal a, b, y: std_logic;
. . .
process(a,b)
  variable tmp: std_logic;
begin
   tmp := '0';
   tmp := tmp or a;
   tmp := tmp or b;
   y <= tmp;
end process;
```

Tipos propios

Podemos usar nuestros propios tipos de datos usando la siguiente estructura: type *name* is *(fsm_state1, fsm_state2, etc..);*

- Muy útil para las máquinas de estados, funcionan como un enum en c.

Uso y recomendaciones

- Siempre se debe definir el ancho de la señal.
- Definir el bit más significativo (MSB).
- Por convenio se utiliza MSB downto LSB.
- No usar tipos de datos propios salvo uso necesario (FSMs).
- No usar asignación inmediata (:=).
- Usar operandos con la misma longitud en los operadores relacionales.
- En general no usar variables.

```
37    signal rstcnt, r_rstcnt      : unsigned(3 downto 0);
```

Declaración MSB LSB

Diseño jerárquico

- Al aumentar el tamaño del diseño, la complejidad aumenta.
- El objetivo es dividir recursivamente el diseño en partes pequeñas, más fácilmente gestionables.
- Cada módulo debe ser **independiente**.
  - **Independiente**: Comprobable y funcional

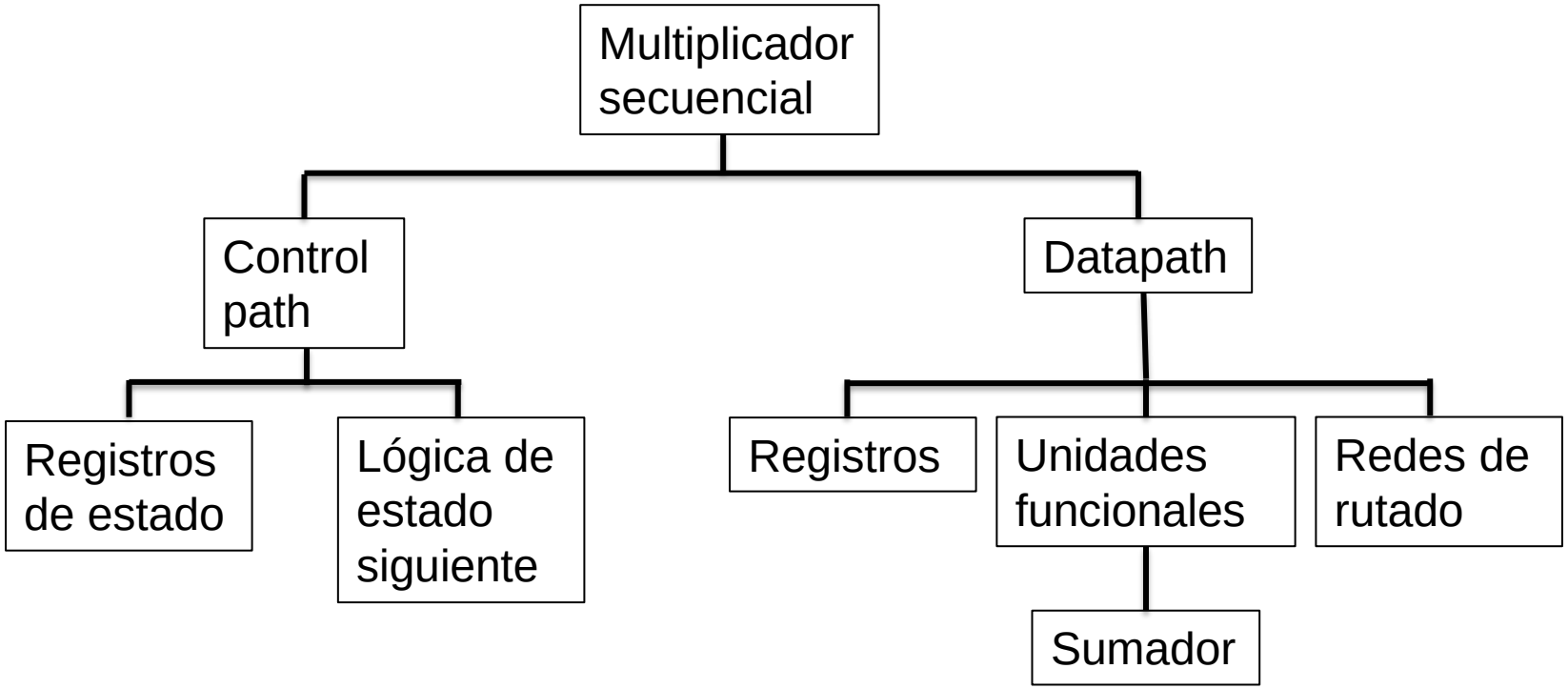
Diseño jerárquico: Ejemplo

- Multiplicador secuencial: Consiste en sumar tantas veces el primer término como indique el segundo termino.
- Usando un diseño jerárquico podemos reutilizar el mismo operador (suma o and) o instanciar tantas sumas como deseemos.

$$
\begin{array}{cccccccc}
 & & & a_3 & a_2 & a_1 & a_0 & \\
 & & \times & b_3 & b_2 & b_1 & b_0 & \\
\hline
 & & & p_{30} & p_{20} & p_{10} & p_{00} & \\
 & & p_{31} & p_{21} & p_{11} & p_{01} & \times & \\
 & p_{32} & p_{22} & p_{12} & p_{02} & \times & \times & \\
p_{33} & p_{23} & p_{13} & p_{03} & \times & \times & \times & \\
\hline
s_7 & s_6 & s_5 & s_4 & s_3 & s_2 & s_1 & s_0
\end{array}
$$

$$s_0 = p_{00} = a_0 \times b_0$$
$$p_{10} = a_1 \times b_0$$
$$p_{01} = a_0 \times b_1$$
$$s_1 = p_{10} + p_{01}$$

Diseño jerárquico: Ejemplo



Diseño jerárquico: Beneficios

- Más fácil, nos centramosos en un diseño más manejable. Diseñar, analizar y verificar; cada módulo por separado.
- Diseño colaborativo más sencillo.
- Menor tiempo de síntesis, síntesis incremental.
- Mayor facilidad de integrar IPs del fabricante.
- Reusabilidad de módulos en futuros diseños.
- Mejor portabilidad

Diseño jerárquico: Elementos

Elementos utilizados en un diseño jerárquico.

- Componentes
- Genéricos
- Configuraciones
- Librerías
- Paquetes
- Subprogramas (procedimientos y funciones)

Genéricos

- Mecanismo para pasar información a una entidad o componente.
- Funcionan como parámetros.
- Se toma el valor dado en el fichero superior, si no le asigna valor se le da el valor default (aparece en la declaración del genérico en el submódulo.).

```
entity entity_name is
    generic(
        generic_names: data_type;
        generic_names: data_type;
        . . .
    );
    port(
        port_names: mode data_type;
        ...
    );
end entity_name;
```

Configuraciones

- Sirve para utilizar diferentes arquitecturas en diferentes instancias.

```
configuration conf_name of entity_name is
  for archiecture_name
    for instance_label: component_name
      use entity lib_name.bound_entity_name(bound_arch_name);
    end for;
    for instance_label: component_name
      use entity lib_name.bound_entity_name(bound_arch_name);
    end for;
    . . .
  end for;
end;
```

Genéricos: Ejemplo

```
26  entity debouncer is
27      generic(
28          g_timeout           : integer   := 5;
29          g_clock_freq_KHZ    : integer   := 100_000
30      );
31      port (
32          rst_n       : in    std_logic;
33          clk         : in    std_logic;
34          ena         : in    std_logic;
35          sig_in      : in    std_logic;
36          debounced   : out   std_logic
37      );
38  end debouncer;
39
```

Genéricos: Ejemplo

- Instanciar

```
40  architecture behavioural of top_practica1 is
41    component debouncer is
42      generic(
43          g_timeout          : integer      := 5;
44          g_clock_freq_KHZ    : integer  := 100_000
45      );
46      port (
47          rst_n       : in    std_logic;
48          clk         : in    std_logic;
```

Procedimientos

- Un procedimiento en VHDL (procedure) es una construcción que nos permite ejecutar bloques de código desde distintas partes de un diseño.
- Evita repetir innecesariamente trozos de código.
- Puede contener llamadas a otros procedimientos o funciones.
- Puede tener ninguno, uno o más argumentos.
- Los argumentos pueden ser inputs, outputs o inouts.

Ejercicio

- ¿Cómo diseñamos un reloj que cuente segundos, minutos y horas?

Procedimientos: Ejemplo Reloj

```
20  procedure IncrementWrap(signal   Counter   : inout integer;
21                          constant WrapValue : in    integer;
22                          constant Enable    : in    boolean;
23                          variable Wrapped   : out   boolean) is
24  begin
25      Wrapped := false;
26      if Enable then
27          if Counter = WrapValue - 1 then
28              Wrapped := true;
29              Counter <= 0;
30          else
31              Counter <= Counter + 1;
32          end if;
33      end if;
34  end procedure;
```

Procedimientos: Ejemplo Reloj

```
38      process(Clk) is
39          variable Wrap : boolean;
40      begin
41          if rising_edge(Clk) then
42
43              -- If the negative reset signal is active
44              if nRst = '0' then
45                  Ticks   <= 0;
46                  Seconds <= 0;
47                  Minutes <= 0;
48                  Hours   <= 0;
49              else
50
51                  -- Cascade counters
52                  IncrementWrap(Ticks, ClockFrequencyHz, true, Wrap);
53                  IncrementWrap(Seconds,              60, Wrap, Wrap);
54                  IncrementWrap(Minutes,              60, Wrap, Wrap);
55                  IncrementWrap(Hours,                24, Wrap, Wrap);
56
57              end if;
58          end if;
59      end process;
```

Funciones

- Son bloques de código, habitualmente un comportamiento combinacional sencillo.
- Puede tener ninguno, uno o más argumentos.
- Los argumentos solo pueden ser inputs.
- Sólo devuelven un valor, a diferencia de los procedimientos.

Funciones: Conversión a ASCII

```
1  function f_ASCII_2_HEX (
2      r_ASCII_IN : in std_logic_vector(7 downto 0))
3      return std_logic_vector is
4      variable v_TEMP : std_logic_vector(3 downto 0);
5    begin
6      if (r_ASCII_IN = X"41" or r_ASCII_IN = X"61") then
7        v_TEMP := X"A";
8      elsif (r_ASCII_IN = X"42" or r_ASCII_IN = X"62") then
9        v_TEMP := X"B";
10     elsif (r_ASCII_IN = X"43" or r_ASCII_IN = X"63") then
11       v_TEMP := X"C";
12     elsif (r_ASCII_IN = X"44" or r_ASCII_IN = X"64") then
13       v_TEMP := X"D";
14     elsif (r_ASCII_IN = X"45" or r_ASCII_IN = X"65") then
15       v_TEMP := X"E";
16     elsif (r_ASCII_IN = X"46" or r_ASCII_IN = X"66") then
17       v_TEMP := X"F";
18     else
19       v_TEMP := r_ASCII_IN(3 downto 0);
20     end if;
21     return std_logic_vector(v_TEMP);
22 end;
```

Librerías

- Repositorio virtual que agrupa unidades de diseño.
- Librería por defecto work.
- Organizar los diferentes módulos.
- Evita duplicar los ficheros en cada carpeta.

Paquetes

- Paquete: Conjunto de declaraciones que sirven para un mismo propósito.
- Compuesto por declaraciones de señales, tipos, procedimientos, funciones, etc..
- Un paquete es sintetizable si todos los elementos lo son.

Paquetes: Ejemplo

```
1  package demo_pack is
2
3  constant c1 : std_logic_vector(7downto0):= "11110000";
4  type STATE is(RESET, IDLE, STOPPED);
5
6  component ADDER is
7      port(A, B : instd_logic;
8           SUM, Cout: outstd_logic
9      );
10
11 end component;
12 end demo_pack;
```

Paquetes: Ejemplo

Pasos a seguir para añadir el anterior paquete a un proyecto de Vivado:

1. Crear un código VHDL con el paquete y añadirlo a *designsources*
2. Pulsar botón derecho sobre el código VHDL > *Set Library* y elegir la librería por defecto(***work***)
3. Modificar el orden de compilación de modo que la librería se procese antes que el resto de los códigos VHDL del proyecto
4. Incluir la librería y el paquete en nuestros códigos VHDL añadiendo:

```
14  library work;
15  use work.demo_pack.all;
```

Diseño paramétrico

- Objetivo: Diseñar módulos comunes que pueden ser reutilizados en diferentes diseños digitales.
- Es necesario para el ancho de las palabras, el tamaño de las memorias, longitud de los operadores.
- Se hace habitualmente mediante parámetros (genéricos), parámetros de los array que se utilizan para adaptar el tamaño de los arrays y usar bucles para generar el numero de entidades necesarias.

For generate

- Sirve para generar un número determinado de instancias.
- Ejemplo sumador en el repo de git.

```
32      GEN_ADD: for I in 0 to 7 generate
33
34      LOWER_BIT: if I=0 generate
35        U0: halfAdder port map
36            (A(I),B(I),S(I),C(I));
37      end generate LOWER_BIT;
38
39      UPPER_BITS: if I>0 generate
40        UX: fullAdder port map
41            (A(I),B(I),C(I-1),S(I),C(I));
42      end generate UPPER_BITS;
43
44    end generate GEN_ADD;
```

Diseño a bajo nivel

Las FPGAs no sólo tienen CLBs también tienen otros elementos como DSPs, BRAMs, Registros para sicronizar, etc…

Existen dos formas de usarlos:

- Inferir: El sintetizador lo hace por nosotros.
- Instanciar: Obligamos al sintetizador a utilizar estos elementos.

Diseño a bajo nivel: Inferir

Inferir consiste en diseñar los módulos de tal manera que el sintetizador entienda que debe usar las estructuras primitivas subyacentes.
Problema: Debemos revisar que se ha inferido la estructura esperada.

Primero debemos conocer la arquitectura de nuestra FPGA. Para ello consultamos:

- 7 Series FPGAS Memory Resources
- Vivado Design Suite 7 Series FPGA Libraries Guide

Diseño a bajo nivel: Instanciar

Instanciar consiste en llamar a las primitivas directamente. Nos da control total sobre el Hardware, pero perdemos portabilidad.

Debemos conocer las primitivas disponibles:

- Vivado Design Suite 7 Series FPGA Libraries Guide

Diseño a bajo nivel: Ejemplo(I)

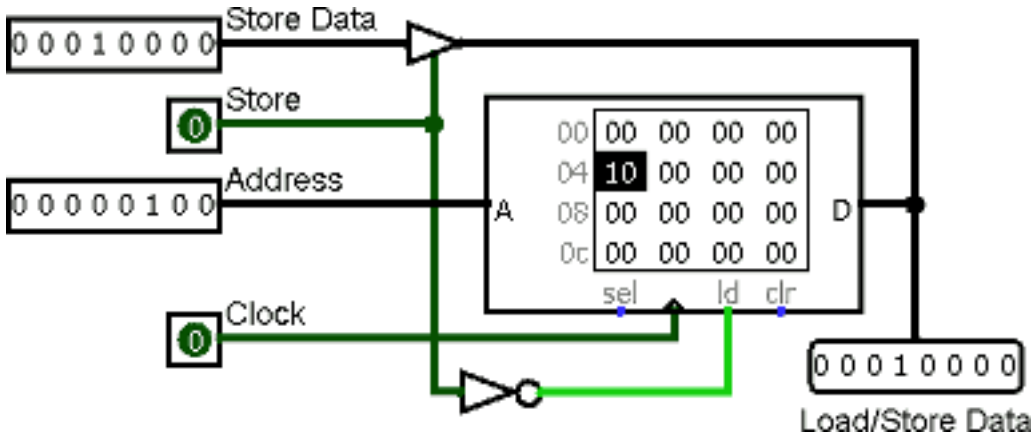
Queremos diseñar una memoria:

- Usar **LUT RAM** y **registros (Distributed RAM)**: Muy flexible, de cualquier tamaño, mucho área, no se puede leer y escribir a la vez, ideal para almacenar elementos pequeños.
- Usar **BRAM**: No ocupa registros ni LUTS, más rápido, menos flexible, bloques de tamaño fijo.

Diseño a bajo nivel: Ejemplo(II)

La implementación en LUT RAM o registros sería directa, en BRAM es donde debemos elegir como hacerlo:

- ¿Qué es una RAM? Consiste en un array de dos dimensiones en el que se almacena información.



Diseño a bajo nivel: Ejemplo(III)

Infiriendo BRAM

- Según la arquitectura de la FPGA (nuestro caso serie 7 de Xilinx) se definen una serie de puertos.

```
9   entity bram is
10      generic(
11          g_addr_width : integer := 8;       --Address bus width
12          g_data_size  : integer := 4        --Data word size
13      );
14      port(
15          clk        : in     std_logic;                                  -- System clock
16          data_write : in     std_logic_vector (g_data_size-1 downto 0);  -- Data write
17          read_add   : in     std_logic_vector (g_addr_width-1 downto 0); -- Read address
18          read_en    : in     std_logic;                                  -- Read enable
19          write_add  : in     std_logic_vector (g_addr_width-1 downto 0); -- Write address
20          write_en   : in     std_logic;                                  -- Write enable
21          data_read  : out    std_logic_vector (g_data_size-1 downto 0)   -- Data read
22      );
23
24  end entity bram ;
25
```

Diseño a bajo nivel: Ejemplo(IV)

```
27  -- Architecture declarations
28  architecture rtl of bram is
29
30
31      constant c_ram_capacity : integer := 2**g_addr_width;
32
33      -- Internal signal declarations
34      type t_mem is array (0 to c_ram_capacity -1 )
35          of std_logic_vector(g_data_size-1 downto 0); --RAM declaration
36      signal mem : t_mem; -- RAM instantion
37      attribute ram_style : string;
38      attribute ram_style of mem : signal is "block";
39
40
```

Tipo de RAM

Diseño a bajo nivel: Ejemplo(V)

Escritura

```
48      process(clk)
49      begin
50          if clk'event and clk = '1' then
51              if write_en = '1' then
52                  mem(to_integer(unsigned(write_add))) <= data_write;
53              end if;
54          end if;
55      end process;
```

Lectura

```
60      process(clk)
61      begin
62          if clk'event and clk = '1' then
63              if read_en = '1' then
64                  data_read <= mem(to_integer(unsigned(read_add)));
65              end if;
66          end if;
67      end process;
```

Resumen

- En VHDL el uso correcto de las señales y los tipos es muy importante.
- Se debe usar std_logic(_vector) y (un)signed.
- El diseño jerárquico y parametrizable facilita nuestra labor.
- Podemos usar instancias de elementos a bajo nivel para asegurarnos la implementación deseada o inferirlos y comprobar que el resultado es el esperado.

En la próxima hora

- Resolución de problemas.

¿Preguntas?

UNIVERSIDAD
NEBRIJA