VHDL

Indice

- Lenguajes descriptores de hardware
- VHDL
  - Introducción
  - Entidad de diseño (declaración de entidad y cuerpo de arquitectura)
  - Tipos
  - Objetos
  - Operadores
  - Expresiones concurrentes y secuenciales
  - Subprogramas
  - Funciones de resolución
  - Instanciación de un componente
- Bancos de prueba
- Funciones de conversión
- Máquinas de estado

Lenguajes descriptores de hardware

- **Introducción**

Un lenguaje descriptor de hardware es un lenguaje para la descripción formal y el diseño de circuitos electrónicos. Puede ser utilizado para describir el funcionamiento de un circuito, su diseño y organización, y para realizar las pruebas necesarias para verificar el correcto funcionamiento.

Ejemplos de este tipo de lenguaje son VHDL, Verilog, ABEL, AHDL, SystemC, SystemVerilog).

Lenguajes descriptores de hardware

- **Objetivos**

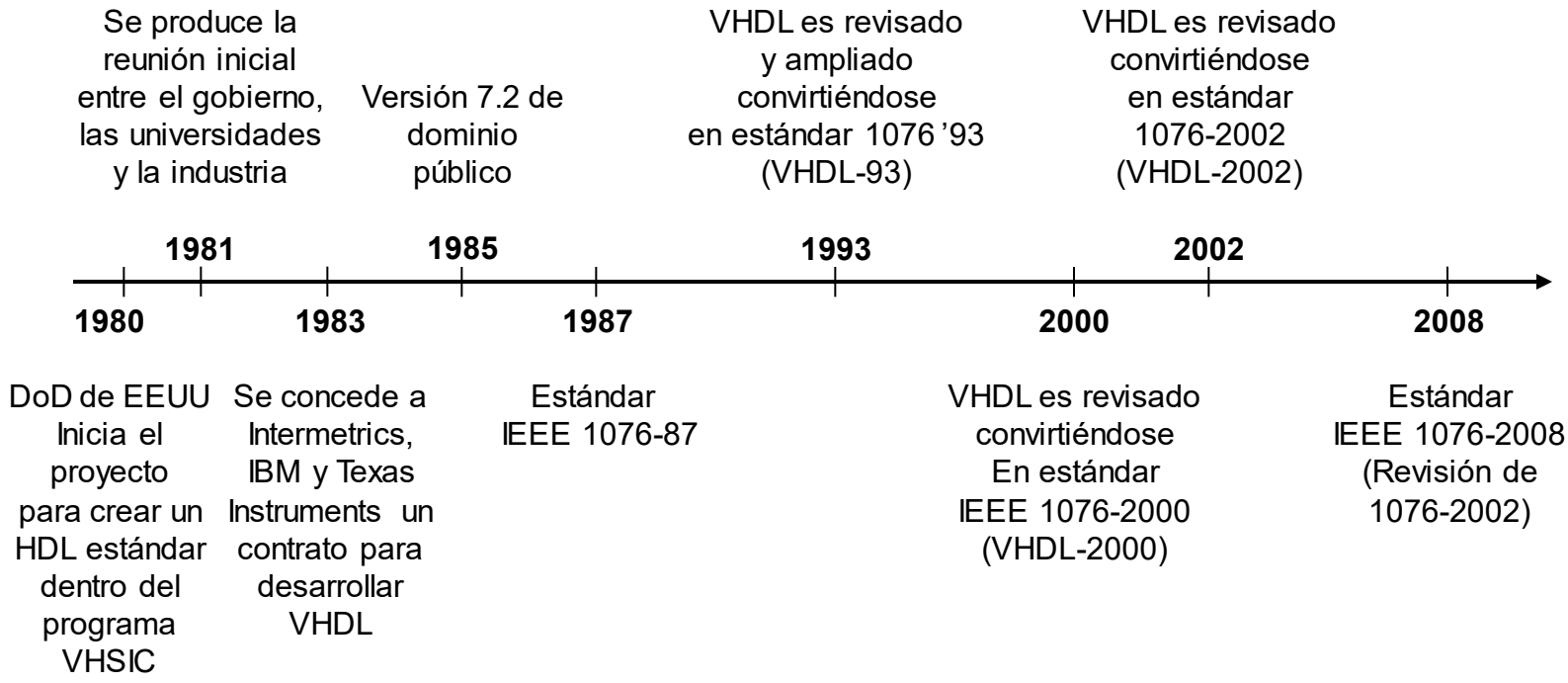
  - Especificación de circuitos electrónicos
  - Documentación
  - Simulación/Verificación de los circuitos antes de ser implementados

VHDL

- **Introducción**
  - VHDL significa **V**ery High Speed Integrated Circuit (VHSIC) **H**ardware **D**escription **L**anguage.
  - Fue desarrollado originalmente por pedido del departamento de defensa de Estados Unidos.
  - El lenguaje está especificado en la norma IEEE-1076
  - Es utilizado para modelar circuitos digitales, especificando su comportamiento, y para simulación.
  - La descripción de los circuitos puede hacerse en base a tres modelos: Comportamiento, Estructural y Flujo de datos

VHDL

- **Historia**



VHDL

- **Características**

  - Existen cuatro tipo de objetos en VHDL: constantes, variables, señales y archivos.
  - Es un lenguaje "***strong typing***", lo que implica que no pueden asignarse valores a señales o variables que no sean del tipo declarado para esa señal o variable.
  - Para realizar lo antes mencionado se debe llamar a una función de conversión de tipo.
  - Permite dividir un diseño de hardware en módulos más chicos.

VHDL

- **Características**

  - Permite la definición de nuevos tipos
  - El código suele más extenso que el desarrollado en Verilog (se busca que el mismo sirva como documentación)
  - Posee librerías estándar
  - No todo lo descripto en VHDL podrá ser utilizado para configurar una FPGA (**¡No todo VHDL es sintetizable!**).
  - No es “case sensitive”. Es lo mismo entity que ENTITY.
  - Los comentarios se expresan con -- (doble guión).
  - Las instrucciones finalizan con el carácter ; (punto y coma).

Entidad de diseño

La entidad de diseño es la **principal abstracción de hardware en VHDL**. Representa una porción del diseño de un hardware que tiene entradas y salidas definidas y realiza una determinada función. Puede representar un sistema completo, un subsistema, una plaqueta, un chip, una macrocelda, una compuerta lógica o cualquier nivel de abstracción que se encuentre entre los antes mencionados.(*)

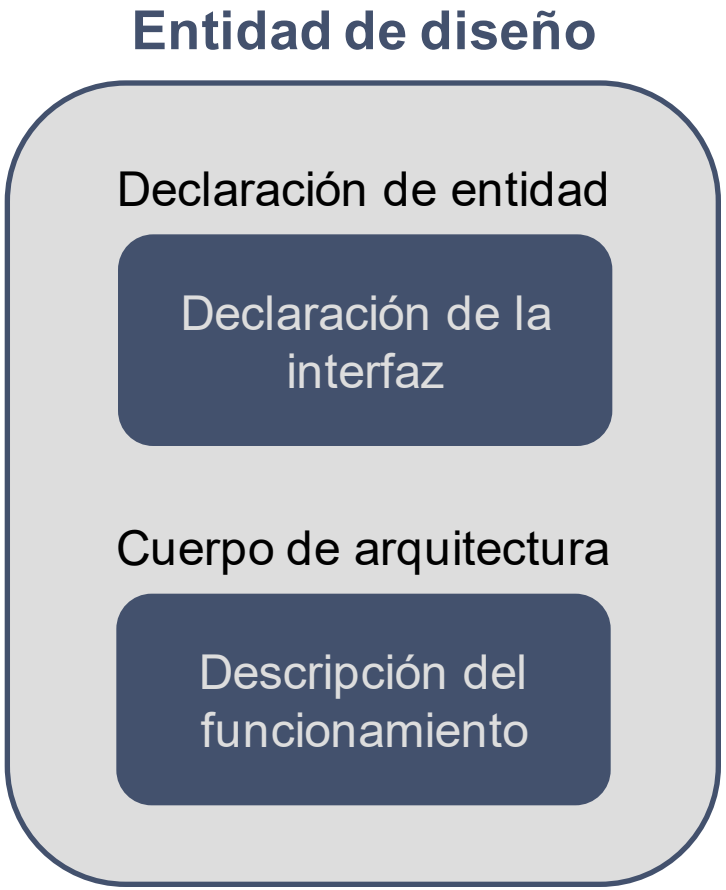
(*) IEEE Std 1076™-2008 - IEEE Standard VHDL Language

Entidad de diseño

Una entidad de diseño puede ser descripta en términos de una **jerarquía de bloques**, cada uno de los cuales representa una porción del diseño completo. El bloque superior (*top-level*) en esta jerarquía es la propia entidad de diseño; tal bloque es un bloque externo y puede ser utilizado como componente de otros diseños. Los bloques anidados en la jerarquía son bloques internos. (*)

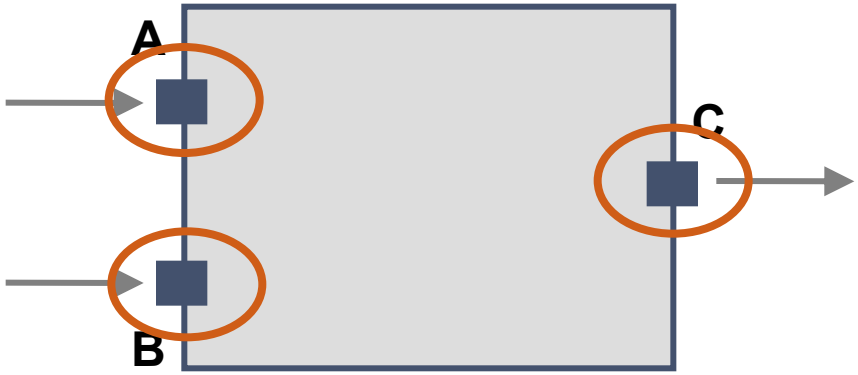
(*) IEEE Std 1076™-2008 - IEEE Standard VHDL Language

Entidad de diseño



Declaración de entidad

- Una declaración de entidad define la interfaz entre una entidad de diseño dada y el entorno en el cual es usada. (*)
- Define cuáles son los puertos y los modos de los mismos



(*) IEEE Std 1076™-2008 - IEEE Standard VHDL Language

Declaración de entidad

- **Sintaxis**

entity *nombre_entidad* ***is***

*[**generic** (generic_variable_declarations) ;]*

*[**port** (lista_de_interfaz) ;]*

[declaraciones]

[begin

[sentencias]]

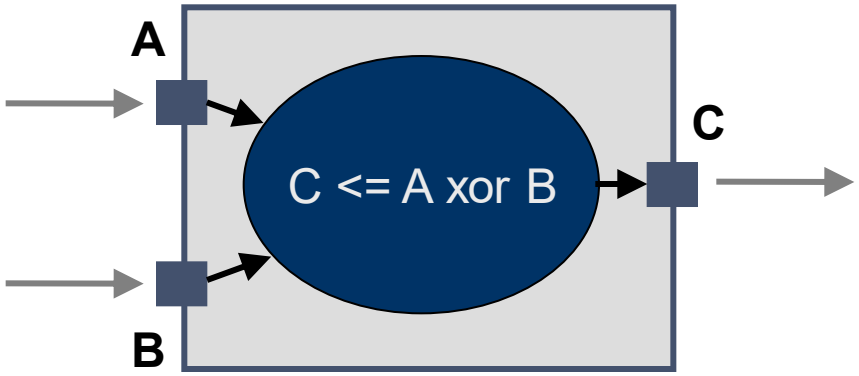
end [entity] *[nombre_entidad];* (*)

(*) IEEE Std 1076™-2008 - IEEE Standard VHDL Language

Cuerpo de arquitectura

- **Introducción**

  - Un cuerpo de arquitectura define el cuerpo de una entidad de diseño. Especifica las relaciones entre las entradas y las salidas de esta, y puede estar expresada en términos de estructura, flujo de datos o comportamiento. (*)



(*) IEEE Std 1076™-2008 - IEEE Standard VHDL Language

Cuerpo de arquitectura

- **Sintaxis**

architecture *nombre_arquitectura* ***of*** *nombre_entidad* ***is***

[declaraciones]

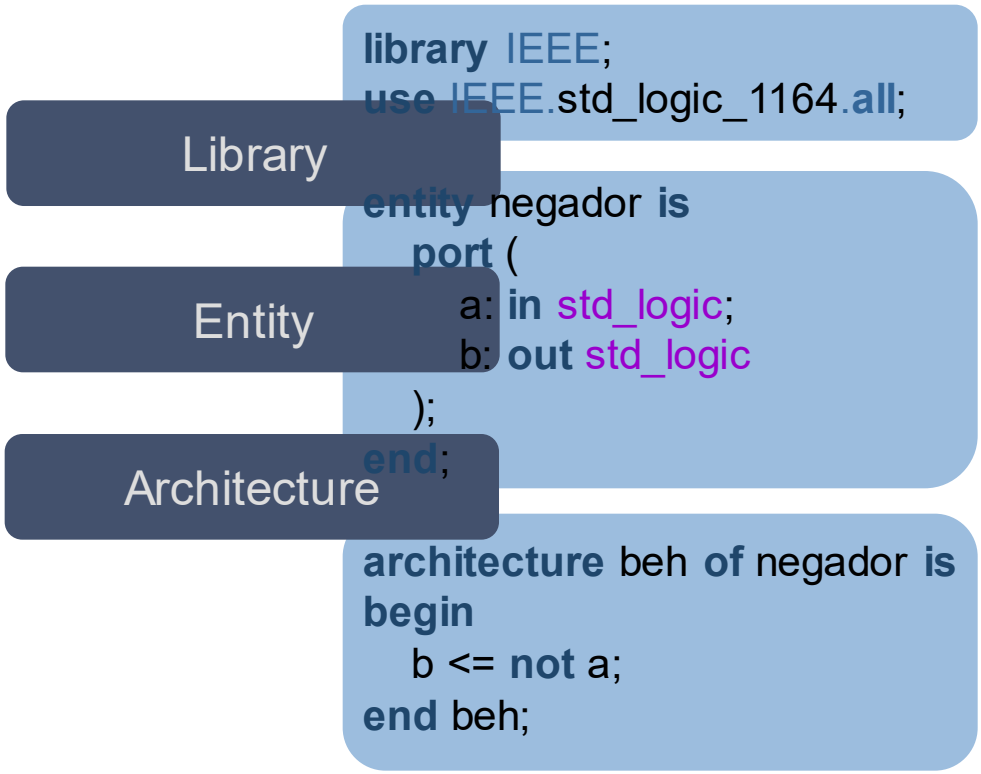
begin

[sentencias]

end *[**architecture**] [nombre_arquitectura];* (*)

(*) IEEE Std 1076™-2008 - IEEE Standard VHDL Language

Estructura del código VHDL

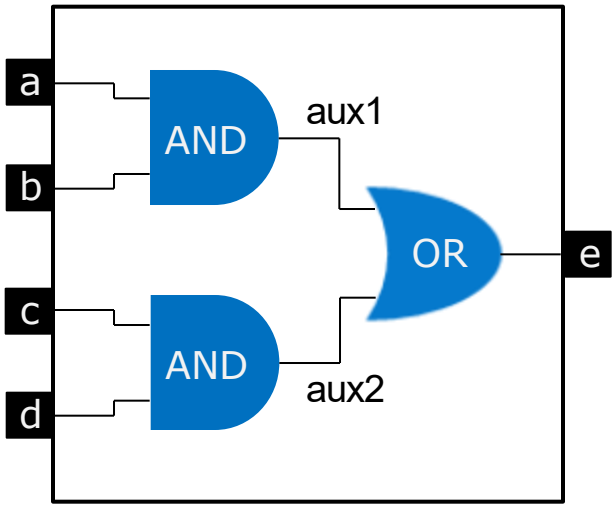


Ejemplo en VHDL

```
entity circuito is
    port (
        a  : in  bit;
        b  : in  bit;
        c  : in  bit;
        d  : in  bit;
        e  : out bit);
end circuito;
```

```
architecture arq of circuito is
    signal aux1: bit;
    signal aux2: bit;
begin
    aux1 <= a and b;
    aux2 <= c and d;
    e <= aux1 or aux2;
end arq;
```

e <= (a and b) or (c and d);



Tipos

- La declaración de los tipos predefinidos se encuentra en el paquete STANDARD
- Existen 5 clases de tipos:
    - **Scalar (entero, enumerado, físico, punto flotante)**
    - **Composite (array, record)**
    - **File**
    - **Access**
    - **Protected**

Tipos escalares (ejemplos)

- **Entero**

```
type WORD_INDEX is range 31 downto 0;
type BYTE_LENGTH_INTEGER is range 0 to 255;
```

* El único tipo entero predefinido es **INTEGER**

- **Enumerado**

```
type BIT is ('0', '1');
type estados is (Espera, Inicializacion, Procesamiento, Envio_Datos);
```

* Los tipos enumerados predefinidos son: CHARACTER, BIT, BOOLEAN, SEVERITY_LEVEL, FILE_OPEN_KIND, Y FILE_OPEN_STATUS.

Tipos escalares (ejemplos)

- **Físico**

```
type DURATION is range -1E18 to 1E18
        units
                fs;                 --femtosecond
                ps = 1000 fs;       --picosecond
                ns = 1000 ps;       --nanosecond
                us = 1000 ns;       --microsecond
                ms = 1000 us;       --millisecond
                sec = 1000 ms;      --second
                min = 60 sec;       --minute
        end units;
```

* El único tipo físico predefinido es **TIME**

Tipos escalares (ejemplos)

- **Punto flotante**

```
constant PI: REAL := 3.141592;
```

* El único tipo punto flotante predefinido es **REAL**

Tipos compuestos (ejemplos)

- **Array**

```
type MY_WORD is array(0 to 31) of bit;

type MEMORY is array(integer range <>) of MY_WORD;
```

Estos tipos así declarados se pueden usar para describir una memoria de 2N palabras de 32 bits:

```
signal MY_MEMORY: MEMORY (0 to 2**N-1);
```

* Los tipos array predefinidos son: STRING, BOOLEAN_VECTOR, BIT_VECTOR, INTEGER_VECTOR, REAL_VECTOR, and TIME_VECTOR (definidos en el paquete **STANDARD**)

Tipos compuestos (ejemplos)

- **Record**

```
type DATE is
        record
                DAY : integer range 1 to 31;
                MONTH : MONTH_NAME;
                YEAR : integer range 0 to 5000;
        end record;
```

* No existe ningún tipo record predefinido.

Tipo file (ejemplos)

- **File**

```
file of STRING          -- Define un tipo de archivo que puede
                        -- contener un número indefinido de
                        -- strings de largo arbitrario.

file of NATURAL         -- Define un tipo de archivo que puede
                        -- contener sólo valores enteros positivos
```

Objetos

- Un objeto es un elemento que guarda un valor de un tipo de dato específico. Sólo puede ser parte de operaciones coherentes con su tipo (lenguaje strong typing).

- En VHDL existen 4 clases de objetos:
  - **Constantes**
  - **Señales**
  - **Variables**
  - **Archivos**

Objetos

- **Constantes**
  - Son utilizadas para el manejo de **valores que se mantienen constantes durante el diseño**. De esta manera se evita el uso de números que a simple vista parecen mágicos.
  - Es una práctica muy recomendable el uso de nombres claros y que representen algo dentro del diseño.
  - Para la asignación de un valor a una variable se utiliza el símbolo ":="
  - Se declaran del siguiente modo:

    constant identificador: tipo [:= expresión];

Objetos

- **Constantes**

  - Se declaran del siguiente modo:

    constant identificador: tipo [:= expresión];

  Ejemplo:

    constant *N:* ***integer*** *:= 8;*

Objetos

- **Señales**
  - Representan las conexiones del hardware. En otras palabras, representan los cables.
  - En las simulaciones toman el valor asignado un delta de tiempo después (salvo que se indique un retardo por medio de la sentencia **after**).
  - Para la asignación de un valor a una señal se utiliza el símbolo “<=“
  - Se declaran del siguiente modo:

    signal identificador: tipo [:= expresión];

Objetos

- **Señales: Ejemplos**

Nota: Se supone definidas las siguientes variables

```
signal word: std_logic_vector(3 downto 0);
signal single_bit: std_logic;
```

Se podría tener las siguientes asignaciones:

- *word <= "1110";*
- *single_bit <= '1';*
- *word <= (others => '0');*
- *word <= single_bit & word(3 downto 1);*

Objetos

- **Variables**
  - Son utilizadas para almacenar valores que pueden sufrir cambios.
  - No tienen significado físico
  - Cuando un valor es asignado a una variable esta lo toma inmediatamente.
  - Para la asignación de un valor a una variable se utiliza el símbolo ":="
  - Se declaran del siguiente modo:

    variable identificador: tipo [:= expresión];

Objetos

- **Variables: Ejemplos**

```
variable aux: integer := 3;  -- declaración de variable con
                             -- asignación inicial de un valor

aux := aux + 1;  -- asignación de un valor a una variable
```

Objetos

- **Archivos**
  - Estos objetos no son sintetizables.
  - Se utilizan en simulaciones.
  - Se puede acceder a ellos tanto para lectura como para escritura
  - Se declaran del siguiente modo:

    file identificador **:** subtipo [info_apertura];

    info_apertura = [**open** modo_de_apertura] **is** nombre_archivo

    nombre_archivo = string

Objetos

- **Archivos: Ejemplos** (*)

```
type IntegerFile is file of INTEGER;
file F1: IntegerFile;
-- No existe llamado implícito a FILE_OPEN.
```

```
file F2: IntegerFile is "test.dat";
-- Existe un llamado implícito a:
-- FILE_OPEN (F2, "test.dat"). El modo de apertura por defecto
-- es READ_MODE.
```

```
file F3: IntegerFile open WRITE_MODE is "test.dat";
-- Existe un llamado implícito a:
-- FILE_OPEN (F3, "test.dat", WRITE_MODE);
```

(*) IEEE Std 1076™-2008 - IEEE Standard VHDL Language

Operadores

- **Lógicos**

  and, or, nor, xor, not

- **De relación**

  = (igual) , < (menor) , > (mayor) , /= (distinto) ,
  <= (menor o igual) , >= (mayor o igual)

- **Aritméticos**

  + (suma) , - (resta)

- **Concatenación**

  & Ejemplo: *a <= b(N-1) & b(N-1 **downto** 1);*

Expresiones concurrentes y secuenciales

- **Expresiones concurrentes**

  Su evaluación se efectúa al mismo tiempo.

  - Instrucciones concurrentes.
  - Procesos. Se utilizan para implementar una expresión concurrente utilizando una serie de expresiones secuenciales

- **Expresiones secuenciales**

  Se evalúan en orden, una a continuación de la otra.

  - Se sitúan dentro de los procesos

Sentencias secuenciales

- **Sentencia IF**

  La sentencia if selecciona una o ninguna secuencia de sentencias.

  Sintaxis:

```
[ if_label : ]
   if condition then
      sequence_of_statements
   { elsif condition then
      sequence_of_statements }
   [ else
      sequence_of_statements ]
   end if [ if_label ] ;
```

Sentencias secuenciales

- **Sentencia IF (ejemplo)**

```
if x1= '0' and X2 = '1' then
    sal <= '1' ;
elsif x3= '1' then
    sal <= '1';
else
    sal <= '0';
end if;
```

Sentencias secuenciales

- **Sentencia CASE**

La sentencia case selecciona una entre varias alternativas de secuencia de sentencias.

Sintaxis:

[label:] **case** expression **is**
when choices => sequential_statements
when choices => sequential_statements
...
end case [label];
choices = choice | ...
choice = constant_expression | range | others

Sentencias secuenciales

- **Sentencia CASE (ejemplo)**

```
C1: case sel is
        when 1 => A <= '0';
                  B <= '0';
        when 2 => A <= '1';
                  B <= '1';
        when 3 => A <= '0';
                  B <= '1';
        when others => A <= '1';
                       B <= '0';
end case C1;
```

Sentencias secuenciales

- **Sentencia LOOP**

La sentencia Loop incluye una secuencia de sentencias que se ejecutarán reiteradamente, ninguna o varias veces.

Sintaxis:

```
[ loop _label : ]
    [ iteration_scheme ] loop
        sequence_of_statements
    end loop  [ loop _label ] ;

iteration_scheme ::=
    while  condition
    | for loop_parameter_specification

parameter_specification ::=
    identifier in  discrete_range
```

Sentencias secuenciales

- ## Sentencia FOR LOOP

La sentencia For Loop contiene una secuencia de sentencias que se repetirán mientras el parametro_loop se mantenga dentro del rango mencionado (*range*) en el encabezado de la sentencia .

Sintaxis:

[etiqueta:] **for** parametro_loop **in** range loop
    sequential_statements
end loop [etiqueta];

- El parámetro parametro_loop no necesita ser declarado. La declaración del loop lo hace implícitamente.

- El parámetro parametro_loop es una constante dentro de un loop lo que implica que no puede asignársele ningún valor dentro del mismo.

Sentencias secuenciales

- **Sentencia FOR LOOP (ejemplo)**

```
A1: for i in vect'range loop
        valor := valor and vect(i);
    end loop;
```

Sentencias secuenciales

- **Sentencia WHILE LOOP**

La sentencia While Loop contiene una secuencia de sentencias que se repetirán mientras la condición se mantenga verdadera.

Sintaxis:

[etiqueta:] **while** condition **loop**
    sentencias_secuenciales
end loop [etiqueta];

Ejemplo:

```
A1: while i < 8  loop
    valor := valor and vect(i);
    i := i + 1;
end loop;
```

Sentencias secuenciales

- **Sentencia NEXT**

La sentencia Next es utilizada para dar por completa la ejecución de una de las iteraciones de un loop, pasando a la siguiente.
Permite saltear parte de la iteración de un loop. Si la condición especificada después de la palabra **when** es verdadera o no existe condición alguna entonces la sentencia es ejecutada. Con esto se logra saltear todas las sentencias que se encuentran por debajo hasta el final del loop y pasar el control a la primera sentencia de la próxima iteración.

Sintaxis:

[label:] **next** [loop_label] [**when** condition];

Sentencias secuenciales

- **Sentencia NEXT (ejemplo)**

```
L1: loop
    k:= 0;
    L2: for CountValue in 1 to 8 loop
        next when CountValue = N;
        if A(k) = 'U' then
            next L1;
        end if;
        k:= k + 1;
    end loop L2;
end loop L1;
```

Salta a la próxima iteración del loop L2

Salta al inicio del loop L1

Sentencias secuenciales

- **Sentencia EXIT**

  Sentencia secuencial que es usada para terminar la ejecución de una sentencia loop. Si existe una condición la sentencia exit será ejecutada cuando esta sea verdadera y el control pasará a la primera sentencia luego del final del loop.

  Sintaxis:

  [label:] **exit** [loop_label] [**when** condition];

Sentencias secuenciales

- **Sentencia EXIT (ejemplo)**

```
L1: loop
    L2: for i in aux'range loop
            if aux(i) = '0' then
                exit L1;
            end if;
            exit when i = N;
    end loop L2;
    ...
end loop L1;
```

Sale del loop L1

Sale del loop L2

Sentencias secuenciales

- **Sentencia WAIT**

  La sentencia wait provoca la suspensión de un process o un procedimiento.

  Sintaxis:

  [label :] **wait** [sensitivity_clause] [condition_clause] [timeout_clause] ;

  sensitivity_clause ::= **on** sensitivity_list
  sensitivity_list ::= signal_name { , signal_name }
  condition_clause ::= **until** condition
  condition ::= boolean_expression
  timeout_clause ::= **for** time_expression

Sentencias secuenciales

- **Sentencia WAIT (ejemplos)**

```
wait for 25 ns;

wait on signal_A;

wait on signal_A  for 100 ns;

wait until ena = '1' ;
```

Sentencias concurrentes

- **Sentencia WHEN-ELSE**

  Es una asignación condicional de señal.

  Sintaxis:

  target <= options conditional_waveforms ;

  conditional_waveforms ::=
  { waveform **when** condition **else** }
  waveform [**when** condition]

Sentencias concurrentes

- **Sentencia WHEN-ELSE (ejemplo)**

```
entity mux is
    port (
        ent: in bit_vector(3 downto 0);
        sel: in bit_vector(1 downto 0);
        o: out bit);
end;
architecture m of mux is
begin
    o <= ent(0) when sel = "00" else
         ent(1) when sel = "01" else
         ent(2) when sel = "10" else
         ent(3) when sel = "11" else
         '0';
end;
```

Sentencias concurrentes

- **Sentencia WITH-SELECT**

  Es una asignación de señal seleccionada.

  Sintaxis:

```
with expression select
    target <= options selected_waveforms ;

selected_waveforms ::=
    { waveform when choices , }
    waveform when choices
```

Sentencias concurrentes

- **Sentencia WITH-SELECT (Ejemplo)**

```
entity mux is
    port (
        ent: in bit_vector(3 downto 0);
        sel: in bit_vector(1 downto 0);
        o: out bit);
end;
architecture m of mux is
begin
    with sel select
        o <= ent(0) when sel = "00",
             ent(1) when sel = "01",
             ent(2) when sel = "10",
             ent(3) when sel = "11";
end;
```

Sentencias concurrentes

- **Sentencia PROCESS**

  Define un proceso secuencial independiente que representa el comportamiento de alguna parte del diseño.

  Sintaxis:

```
[ process_label : ]
    process [ ( process_sensitivity_list ) ] [ is ]
        process_declarative_part
    begin
        process_statement_part
    end [ postponed ] process [ process_label ] ;
```

Sentencias concurrentes

- **Sentencia PROCESS: Lista de sensibilidad**

Sintaxis:

```
[ process_label : ]
    process [ ( process_sensitivity_list ) ] [ is ]
        process_declarative_part
    begin
        process_statement_part
    end [ postponed ] process [ process_label ] ;
```

*Nota: si existe una lista de sensibilidad se asume que el process contiene una sentencia **wait** implícita como última sentencia en su parte descriptiva.*

Sentencias concurrentes

- **Sentencia PROCESS: Parte declarativa**

Sintaxis:

```
[ process_label : ]
    process [ ( process_sensitivity_list ) ] [ is ]
      process_declarative_part
    begin
       process_statement_part
    end [ postponed ] process [ process_label ] ;
```

```
process_declarative_part ::=
    { process_declarative_item }
```

```
process_declarative_item ::=
    subprogram_declaration
    | subprogram_body
    | type_declaration
    | subtype_declaration …
```

Sentencias concurrentes

- **Sentencia PROCESS: Parte descriptiva**

Sintaxis:

```
[ process_label : ]
    process [ ( process_sensitivity_list ) ] [ is ]
        process_declarative_part
    begin
        process_statement_part
    end [ postponed ] process [ process_label ] ;
```

```
process_statement_part ::=
    { sequential_statement }
```

Sentencias concurrentes

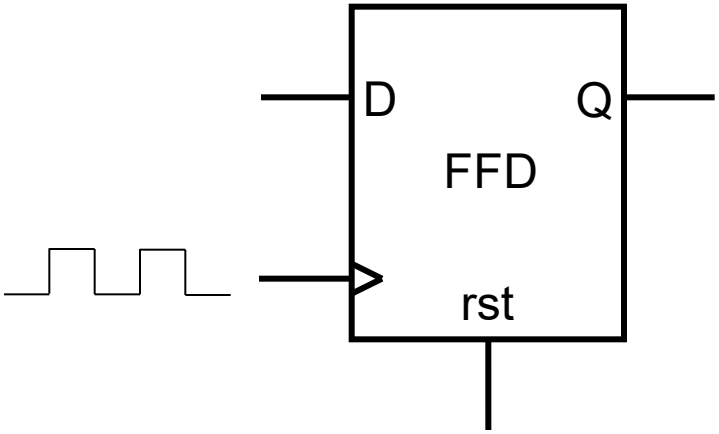
- **Sentencia PROCESS (Ejemplo: Multiplexor)**

```
entity mux is
    port (
        a, b: in bit;
        sel: in bit;
        o: out bit
    );
end;

architecture m of mux is
begin
    process(a,  b, sel)
    begin
        if (sel = '0') then
            o <= a;
        else
            o <= b;
        end if;
    end process;
end;
```

Sentencias concurrentes

- **Sentencia PROCESS (Ejemplo: FFD)**

  Se implementa un flip-flop D utilizando un process activado por el flanco ascendente del reloj y por un reset asincrónico.



Sentencias concurrentes

- **Sentencia PROCESS (Ejemplo: FFD)**

  Se implementa un flip-flop D utilizando un process activado por el flanco ascendente del reloj y por un reset asincrónico.

```
entity ffd is
    port(
        clk: in std_logic;
        rst: in std_logic;
        ena: in std_logic;
        D: in std_logic;
        Q: out std_logic
    );
end ffd;
```

```
architecture ffd_arq of ffd is
begin
    process(clk, rst)
    begin
        if rst = '1' then
            Q <= '0';
        elsif rising_edge(clk) then
            Q <= D;
        end if;
    end process;
end ffd_arq;
```

Sentencias concurrentes

- **Sentencia GENERATE**

A generate statement provides a mechanism for iterative or conditional elaboration of a portion of a description.

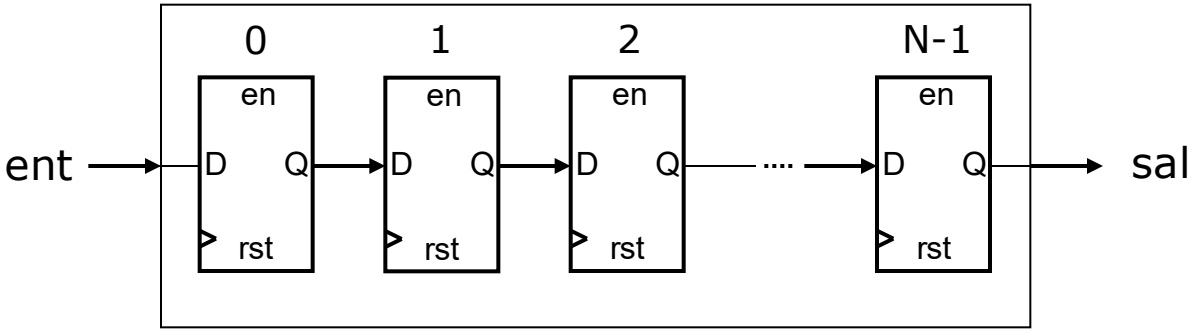
Sintaxis:

```
generate_label :
    generation_scheme generate
        [ { block_declarative_item }
    begin ]
        { concurrent_statement }
    end generate [ generate_label ] ;

generation_scheme ::=
    for generate_parameter_specification
    | if condition
```

Sentencias concurrentes

- **Sentencia GENERATE (Ejemplo)**

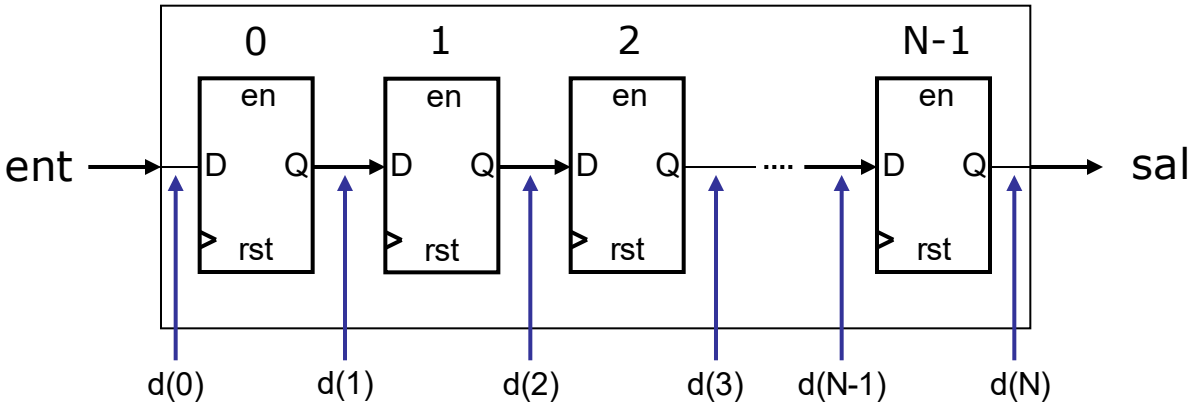
Ejemplo: Implementación de un registro de desplazamiento de N bits



Sentencias concurrentes

- **Sentencia GENERATE (Ejemplo)**

Ejemplo: Implementación de un registro de desplazamiento de N bits



Sentencias concurrentes

- **Sentencia GENERATE (Ejemplo)**

Ejemplo: Implementación de un registro de desplazamiento de N bits

```
entity shift_reg is
    generic (N: natural := 4);
    port ( clk: in bit; rst: in bit; ent: in bit;
           sal: out bit);
end;
architecture estruc of shift_reg is
    component ffd
        port ( rst: in bit := '0';
               ck: in bit;
               d: in bit;
               q: out bit);
    end component;
    signal d: bit_vector(0 to N);
begin
    shift_reg_i : for i in 0 to N-1 generate
        ff_inst : ffd port map(ck => clk , d => d(i), rst => rst; q => d (i +1));
    end generate;
    d(0) <= ent;
    sal <= d(N) ;
end;
```

Sentencias concurrentes

- **Sentencia GENERATE (Ejemplo)**

Ejemplo: Implementación de un registro de desplazamiento de N bits

```
architecture estruc of shift_reg is
    component ffd
        port ( rst: in bit := '0';
               ck: in bit;
               d: in bit;
               q: out bit
        );
    end component;
    signal d: bit_vector(0 to N);
begin
    shift_reg_i : for i in 0 to N-1 generate
        ff_inst : ffd port map(ck => clk , d => d(i), rst => rst; q => d (i +1));
    end generate;
    d(0) <= ent;
    sal <= d(N) ;
end;
```

Subprogramas

- Los subprogramas pueden existir como sólo cuerpos de procedimiento o cuerpos de función.
- También puede existir una declaración de procedimiento y una declaración de función.
- Cuando un subprograma se incluye en un paquete (package) la declaración del mismo se sitúa en la parte declarativa de dicho paquete y el cuerpo del subprograma en el cuerpo del mismo.

Subprogramas: Declaración

- Una declaración de subprograma declara un procedimiento o una función (indicado por la correspondiente palabra reservada, ***procedure*** o ***function***.
- Es opcional. En caso de no existir el propio cuerpo del procedimiento se utiliza como declaración.

Subprogramas: Declaración

- **Declaración de procedure**
  - Se utiliza para declarar la interfaz de llamada a un procedimiento

- **Declaración de función**
  - Se utiliza para declarar la interfaz de llamada y retorno de una función

Subprogramas: Declaración

- **Sintaxis**

procedure *nombre_procedure [(lista_parametros)]*

function *nombre_funcion [(lista_parametros)]* ***return*** *tipo*

Subprogramas: Declaración

- **Procedure: Parámetros formales**
  - Los parámetros formales se detallan en la *lista_parametros* separados por “;”
  - Cada parámetro formal es básicamente una declaración de un objeto que es local al procedure.
  - Las definiciones de tipo usados en los parámetros formales deben ser visibles en el lugar donde el procedure es declarado.

Subprogramas: Declaración

- **Procedure: Parámetros formales**
  - Constantes
  - Variables
  - Señales
  - Archivos

Si el modo es **IN** y no se especificó ninguna clase de objeto se asume **constante**.

Si el modo es **INOUT** o **OUT** se asume **variable**

- **Procedure: Modos de los parámetros formales**
  - IN
  - INOUT
  - OUT

Por defecto el mod**o es IN.**

El tipo **Archivo** no tiene modo.

Subprogramas: Declaración

- **Función: Parámetros formales**
  - Los parámetros formales se detallan en la *lista_parametros* separados por “;”
  - Cada parámetro formal es básicamente una declaración de un objeto que es local a la función.
  - Las definiciones de tipo usados en los parámetros formales deben ser visibles en el lugar donde la función es declarada.

Subprogramas: Declaración

- **Función: Parámetros formales**
  - Constantes
  - Señales
  - Archivos

Si no se especificó ninguna clase de objeto se asume **constante**.

- **Función: Modos de los parámetros formales**
  - IN

Por defecto el mod**o es IN.**

El tipo **Archivo** no tiene modo.

Subprogramas: Cuerpo

- Un cuerpo de subprograma especifica la ejecución de un subprograma
- Los items declarados luego de la especificación del subprograma declaran objetos que serán utilizados localmente dentro del cuerpo del subprograma.
  Los nombres de estos objetos **no son visibles** fuera del subprograma.

Subprogramas: Cuerpo

- **Sintaxis**

```
procedure nombre_procedure [(lista_parametros)] is
        parte_declarativa
begin
        parte_descriptiva
end;
```

```
function nombre_funcion [(lista_parametros)] return tipo is
        parte_declarativa
begin
        parte_descriptiva
end;
```

Subprogramas: Cuerpo

- **Ejemplo: Conversión a entero (1)**

```
function word_to_int(word: std_logic_vector) return integer is
    variable result : integer := 0;
begin
    for index in word'RANGE loop
        if (word(index) = '1') then
            result := result + 2**index;
        end if;
    end loop;
    return result;
end word_to_int;
```

Subprogramas: Cuerpo

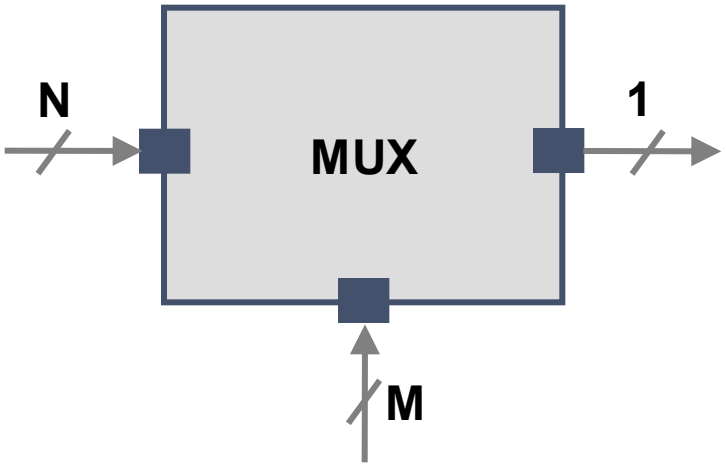
- **Ejemplo: Conversión a entero (2)**

```
function word_to_int(word: std_logic_vector) return integer is
    variable result : integer := 0;
begin
    for index in word'RANGE loop
        result := result * 2;
        if (word(index) = '1') then
            result := result + 1;
        end if;
    end loop;
    return result;
end word_to_int;
```

Subprogramas: Cuerpo

- **Ejemplo: Multiplexor**

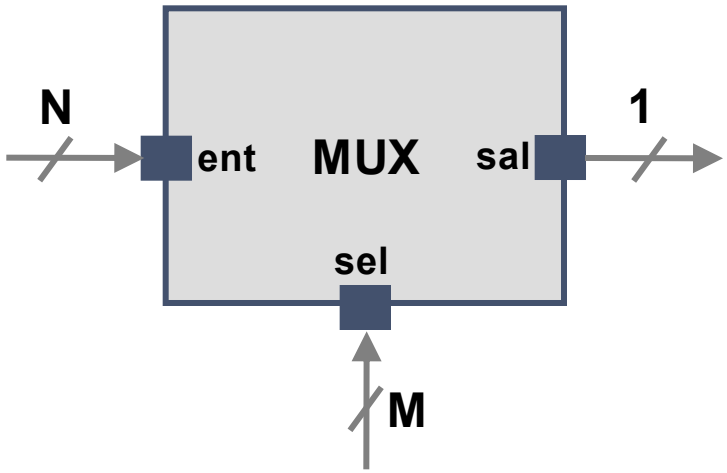
  Utilizando la función de conversión a entero antes mostrada se describe el funcionamiento en VHDL de un multiplexor de N bits de entrada con selección de M bits ($N = 2^M$) y salida de un bit



Subprogramas: Cuerpo

- **Ejemplo: Multiplexor**

```
entity mux is
    generic(N, M: integer);
    port(
        ent: in std_logic_vector(N-1 downto 0);
        sel: in std_logic_vector(M-1 downto 0);
        sal: out std_logic
    );
end;
```



Subprogramas: Cuerpo

- **Ejemplo: Multiplexor**

```
architecture mux_arq of mux is
    function word_to_int(word: std_logic_vector) return integer is
        variable result : integer := 0;
    begin
        for index in word'RANGE loop
            if (word(index) = '1') then
                result := result + 2**index;
            end if;
        end loop;
        return result;
    end word_to_int;
begin
    sal <= ent(word_to_int(sel));
end mux_arq;
```

Subprogramas: Cuerpo

- **Sobrecarga**
  - VHDL permite que dos subprogramas posean el mismo nombre, con la condición de que el número o el tipo de los parámetros sea distinto.
  - Para determinar de qué procedimiento se trata en el caso de una llamada a un procedimiento sobrecargado es usada la cantidad de parámetros reales, su orden, y los tipos, y los nombres de los correspondientes parámetros formales
  - Para el caso de funciones también se utiliza el tipo del resultado

Función de resolución

- Es una función que define cómo los valores de múltiples fuentes de una señal dada se resuelven en un valor único para esa señal. (*)
- Una señal con una función de resolución asociada se denomina señal resuelta.

(*) IEEE Std 1076™-2008 - IEEE Standard VHDL Language

Función de resolución

- **Ejemplo de un tipo no resuelto (BIT)**

```
entity testbench is
end testbench;

architecture behavior of testbench is
    signal a: bit;
begin
    a <= '1';
    a <= '0';
end;
```

```
Running ISim simulation engine ...
This is a Lite version of ISE Simulator(ISim).
ERROR: Resolution function required for the Net /testbench/a driven by:
/testbench//pru_tipo_no_resuelto.vhd:7
/testbench//pru_tipo_no_resuelto.vhd:8
```

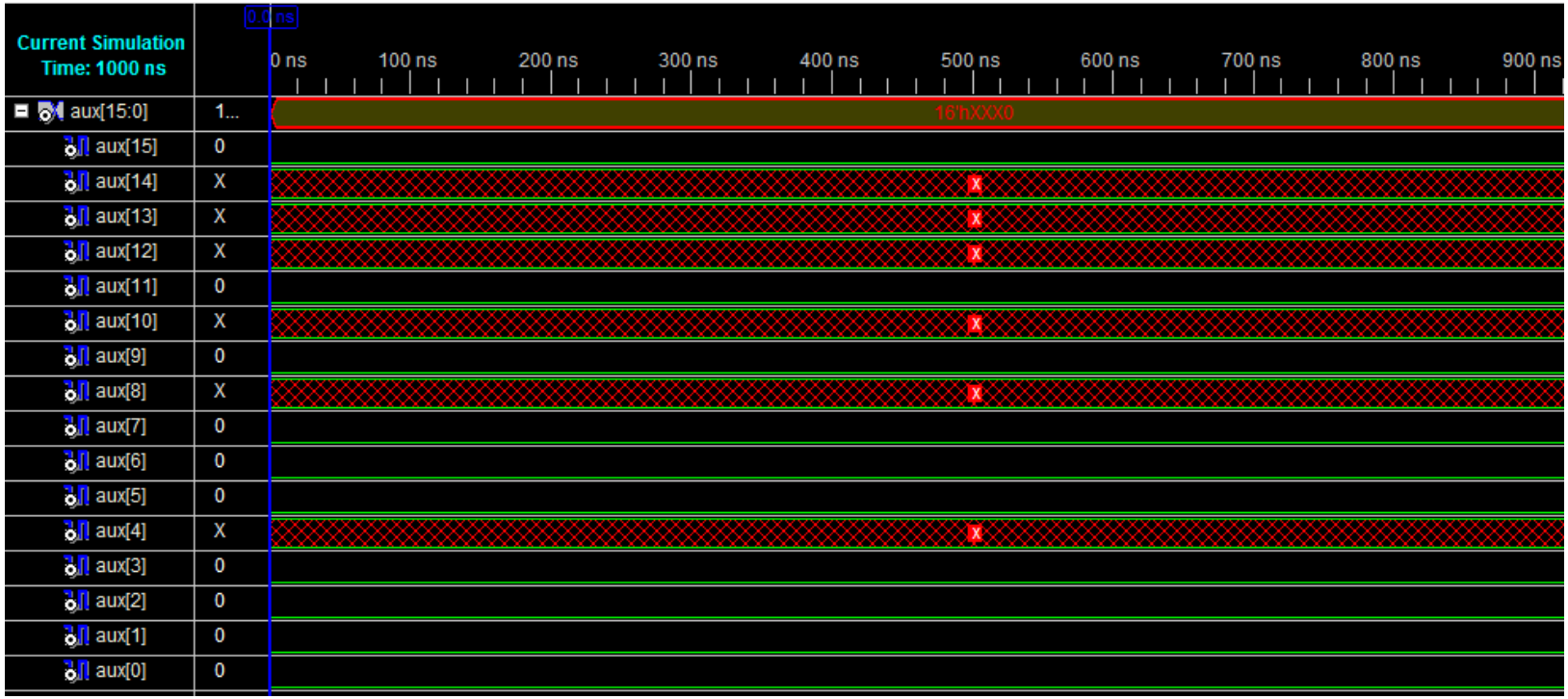
Función de resolución

- **Ejemplo de un tipo resuelto (STD_LOGIC)**

```
library IEEE;
use IEEE.std_logic_1164.all;

entity pru is
end;

architecture p of pru is
   signal aux: std_logic_vector(15 downto 0);
begin

    aux <= (others => '0');
    aux <= b"0111_0101_0001_0000";

end;
```

Función de resolución

- Ejemplo de un tipo resuelto (STD_LOGIC)



Instanciación de un componente

Los módulos se pueden conectar especificando sus conexiones de dos modos:

- **Asociación posicional**: las conexiones a los puertos del módulo deben respetar el orden en que fueron definidos estos últimos.

sum1b_inst: sum1b (a, b, ci, s, co); ← No se indica el puerto

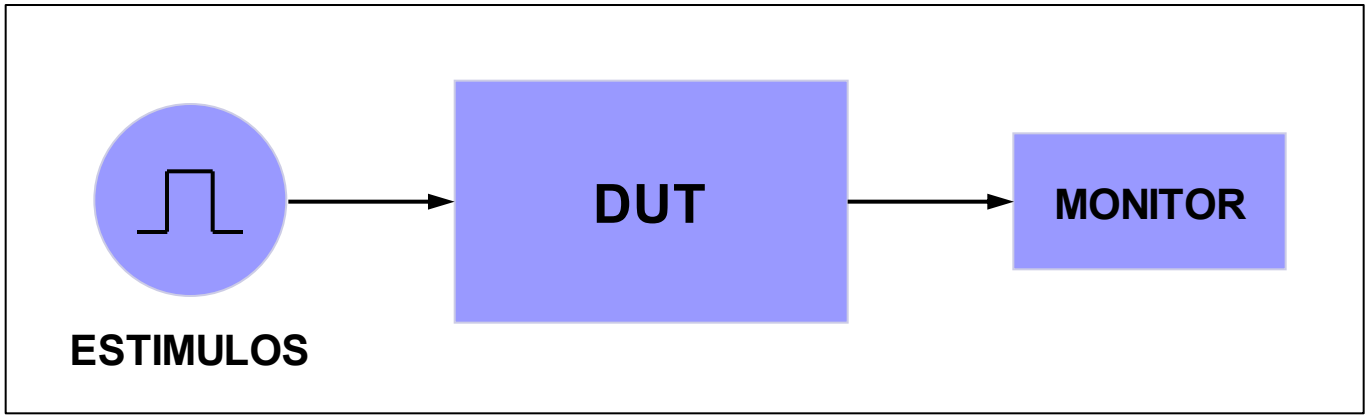
- **Asociación explícita**: las conexiones a los puertos del módulo pueden ser realizadas en cualquier orden.

sum1b_inst: sum1b (b => b, a => a, ci => ci, s =>s, co => co);

Puerto — Señal conectada

Bancos de prueba

¿Cómo realizar la prueba de un dispositivo descripto en VHDL?



Bancos de prueba

- Lograr un Testbench de calidad es imprescindible para la verificación de los diseños
- No están regidos por las normas que se aplican en la síntesis de circuitos
- Se utiliza un generador de señales de prueba y se analiza la respuesta del circuito mediante simulación
- No eliminan por completo la necesidad de probar el circuito una vez sintetizado

Bancos de prueba

- **Aplicación explícita del estímulo utilizando un archivo**
  - VHDL provee un tipo de datos llamado **file** que permite el manejo de archivos tanto de entrada como de salida
  - Los archivos pueden abrirse de tres maneras: **read_mode** para lectura, **write_mode** para escritura y **append_mode** para agregar datos al final de un archivo existente
  - Pueden declararse en la parte declarativa de una arquitectura, un proceso o un subprograma

Bancos de prueba

- Aplicación explícita del estímulo utilizando un archivo

```
entity mux_tb is
end;

architecture beh of mx_tb is
   -- declaración del componente a probar
   -- declaración de constantes y señales IDEM ejemplo anterior
begin
   pp: mux generic map(N_test) port map(mux_in0, mux_in1, sel, mux_out);
   estimulo: process
      variable sel_aux: std_logic_vector(0 downto 0);
      file arch_stim: text open READ_MODE is "datos_simulacion.txt";
      variable linea: line; variable dato1, dato2, dato3: integer;
      variable ch: character:= ' ';
   begin
      while not (endfile(arch_stim)) loop
         readline(arch_stim, linea);
         read(linea, dato1); read(linea, ch); read(linea, dato2); read(linea, ch); read(linea, dato3);
         sel_aux := std_logic_vector(to_unsigned(dato1, 1));
         sel <= sel_aux(0);
         mux_in0 <= std_logic_vector(to_unsigned(dato2, N_test));
         mux_in1 <= std_logic_vector(to_unsigned(dato3, N_test));
         wait for 30 ns;
      end loop;
      wait;
   end process;
end;
```

datos_simulacion.txt

```
0 0 8
0 1 9
0 2 10
0 3 11
0 4 12
0 5 13
0 6 14
0 7 15
1 0 8
1 1 9
1 2 10
1 3 11
1 4 12
1 5 13
1 6 14
1 7 15
```

Bancos de prueba

- **Instrucciones Assert y Report**

  - Instrucciones que se utilizan para verificar una condición y emitir un mensaje durante la simulación

  - Pueden usarse en cualquier parte de un process

assert condition
[**report** expression] [**severity** expression];

[label :] **report** expression [**severity** expression] ;

Bancos de prueba

- **Instrucciones Assert y Report**

```
assert MemoriaLibre >= MEM_LIMITE_MINIMO
report "Memoria baja, se sobrescribirán primeros valores"
severity note;
```

```
assert NumeroBytes /= 0
report "Se recibió paquete sin datos"
severity warning;
```

```
assert AnchoDePulso >= 2 ns
report "Pulso demasiado chico. No generará interrupción."
severity error;
```

```
report "Inicio de simulación."
```

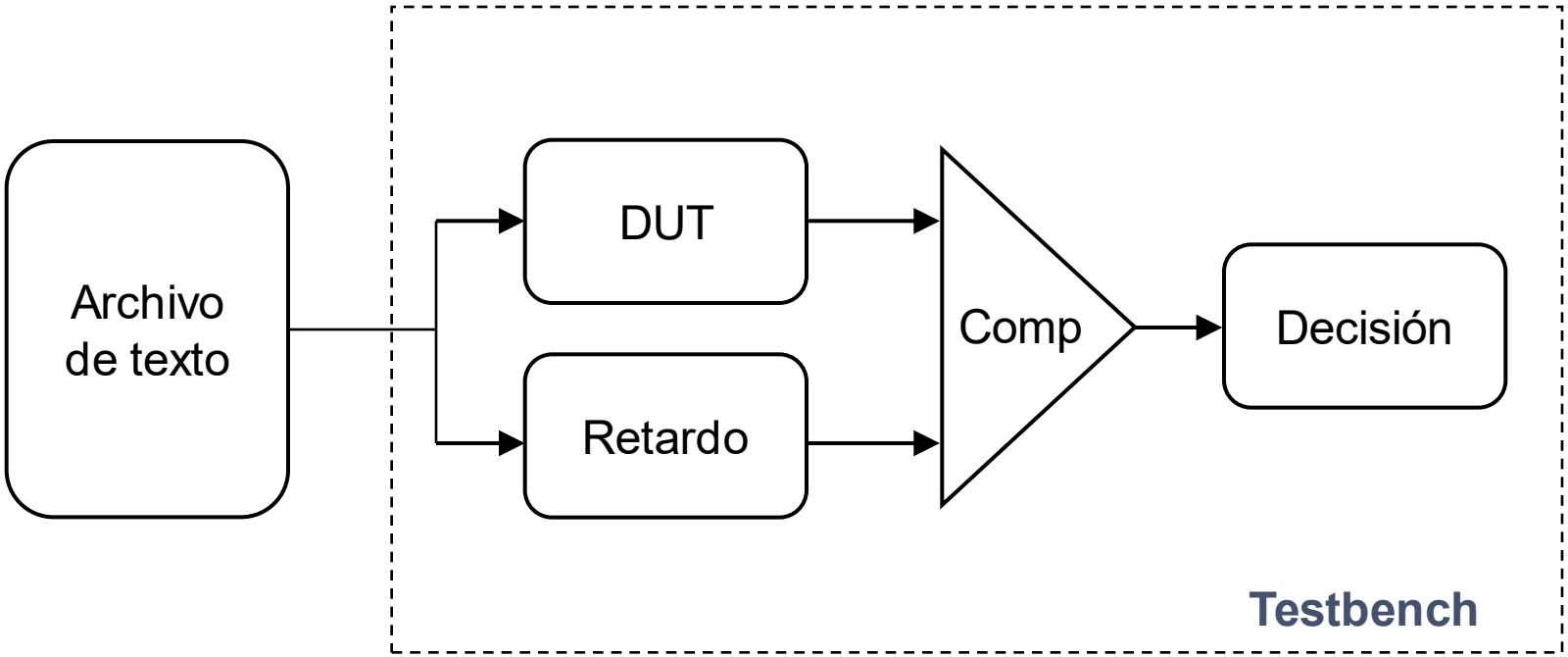
Bancos de prueba

- **Se pueden presentar dos casos**

  - **Caso 1: el DUT no posee clock**

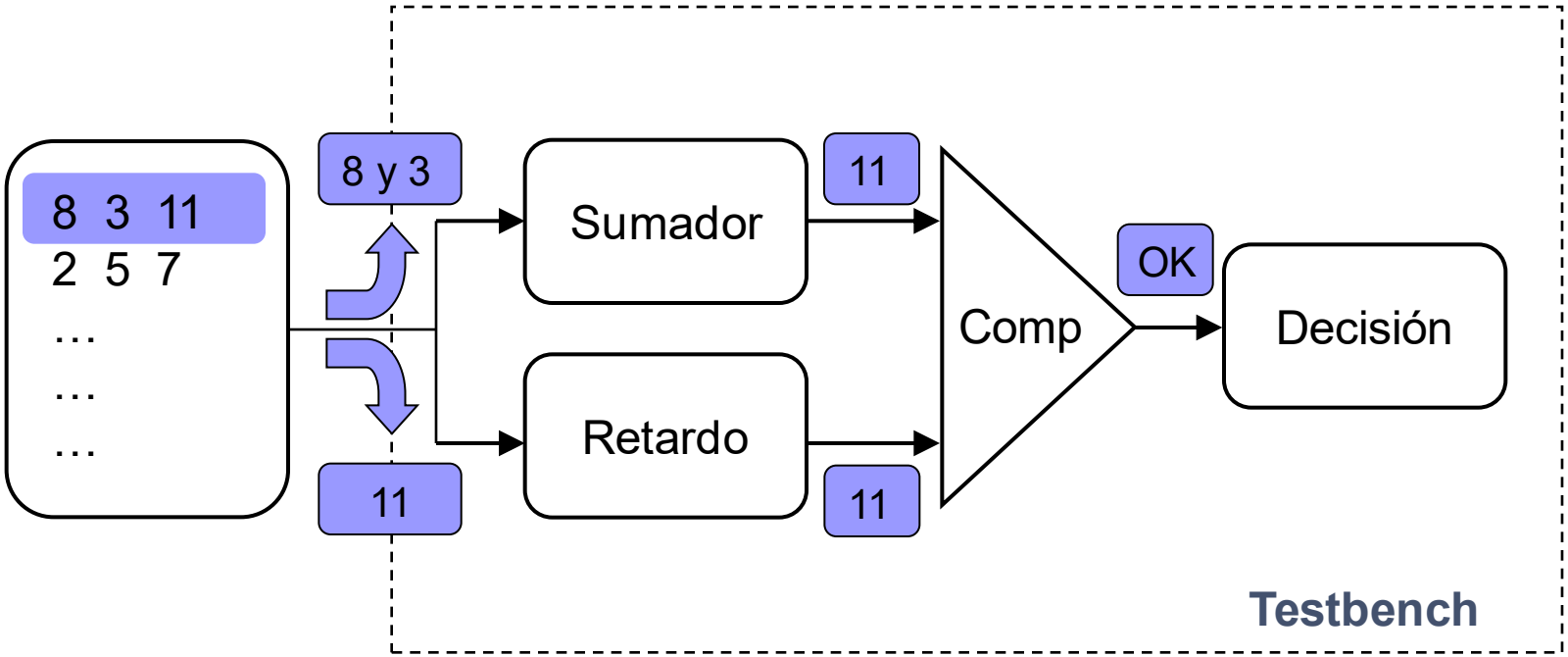
  - **Caso 2: el DUT posee clock**
    En este caso el DUT necesita N ciclos de reloj para obtener un resultado válido a su salida por lo que se debe retardar el valor “resultado” obtenido del archivo de patrones de prueba.

Bancos de prueba



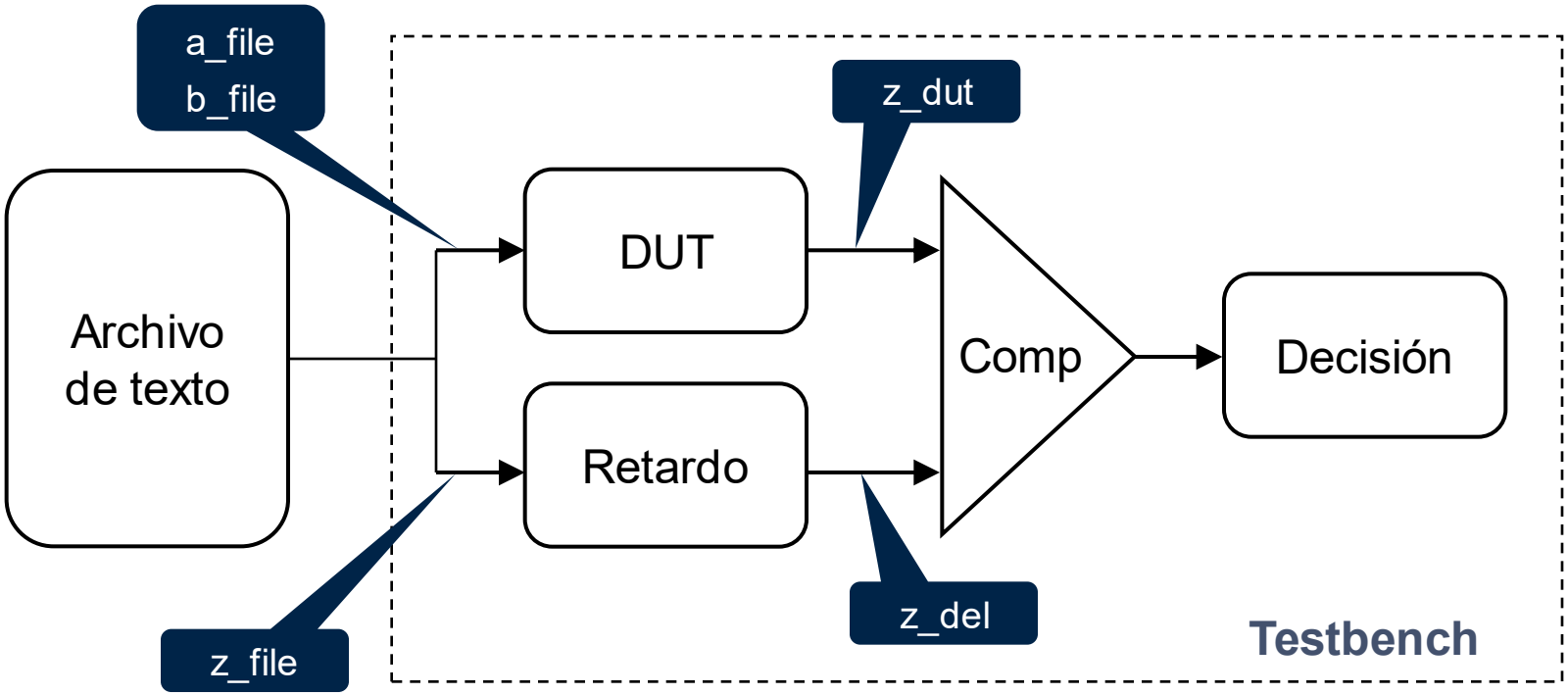
Bancos de prueba

- **Banco de pruebas para un sumador**



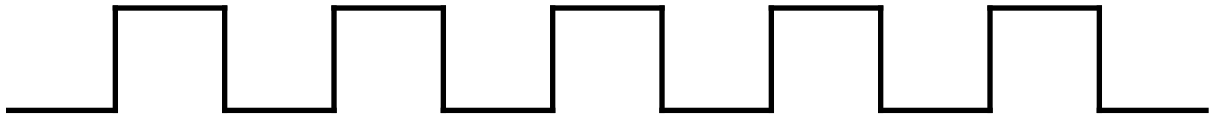
Bancos de prueba

- **Banco de pruebas para un DUT, con o sin retardo**



Bancos de prueba

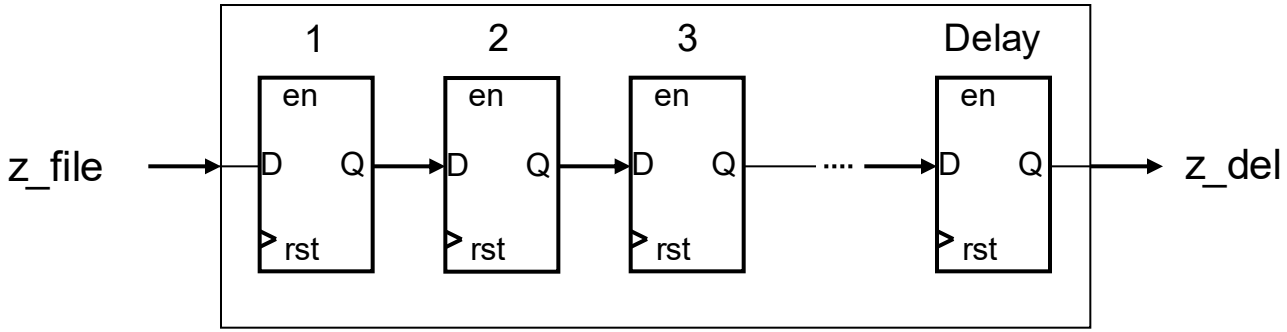
- Diagrama temporal para un DUT con retardo de 2 ciclos



a_file	8	2	1		
b_file	3	5	2		
z_file	11	7	3		
z_dut			11	7	3

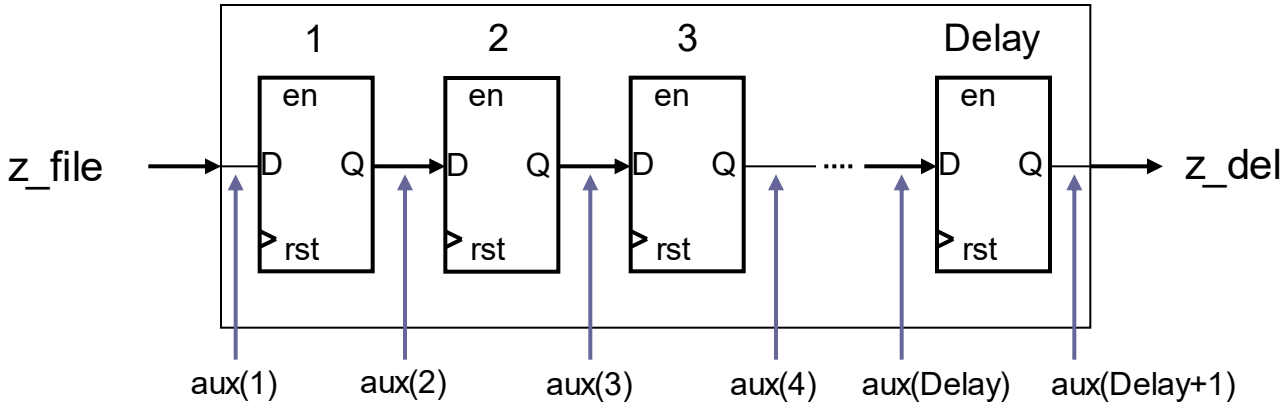
Bancos de prueba

- Cómo generar el retardo?



Bancos de prueba

- Cómo generar el retardo?



Ayuda: utilizar una señal aux de DELAY+2 elementos

Bancos de prueba

- Cómo generar el retardo?

```
aux(0) <= A;
gen_retardo: for i in 0 to DELAY generate
    sin_retardo: if i = 0 generate
        aux(1) <= aux(0);
    end generate;
    con_retardo: if i > 0 generate
        aa: ffd port map(clk => clk, rst => '0', D => aux(i), Q => aux(i+1));
    end generate;
end generate;
B <= aux(DELAY+1);
```

Funciones de conversión

- **Paquetes existentes para operar aritméticamente**

ieee.numeric_std:

Es un paquete de la librería estándar de la IEEE

ieee.std_logic_arith, ieee.std_logic_unsigned:

Son paquetes de Synopsys. Son utilizados casi por defecto por ser desarrollados por una de las empresas cuyo software es uno de los más usado

Funciones de conversión

- **Paquetes existentes para operar aritméticamente**

ieee.numeric_std:

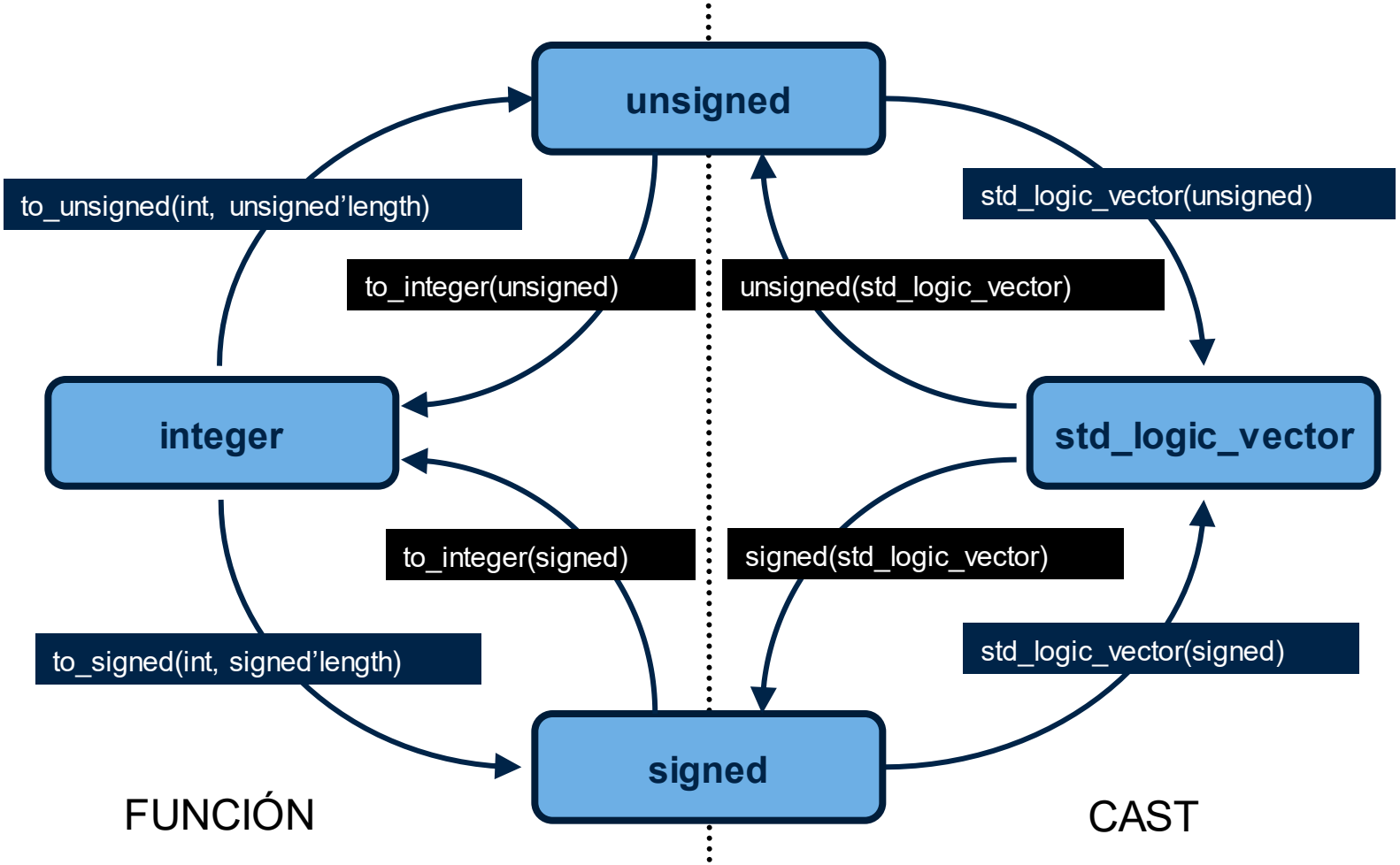
- No tiene definidas las operaciones matemáticas para std_logic, std_logic_vector
- Tiene definidas las operaciones matemáticas para los tipos signed, unsigned e integer

ieee.std_logic_arith, ieee.std_logic_unsigned:

- Tienen definidas las operaciones matemáticas para los tipos std_logic, std_logic_vector e integer

Funciones de conversión

- Diagrama de conversiones y casteos



Funciones de conversión: Ejemplo 1

```
-- Declaración de librerías (incluir numeric_std)

entity contador is
    port (
        clk, rst, ld: in std_logic;
        initial_value: in std_logic_vector(3 downto 0);
        count: out std_logic_vector(3 downto 0)
    );
end contador;

architecture beh of contador is
begin
    count_proc: process(clk, rst)
        variable count_i: integer range 0 to 16;
    begin
        if (rst='1') then
            count_i := 0;
        elsif (rising_edge(clk)) then
            if (ld = '1') then
                count_i := to_integer(unsigned(initial_value));
            else
                count_i := count_i + 1;
                if count_i = 16 then
                    count_i := 0;
                end if;
            end if;
        end if;
        count <= std_logic_vector(to_unsigned(count_i,4));
    end process count_proc;
end architecture;
```

Funciones de conversión: Ejemplo 2

```
-- Declaración de librerías (incluir numeric_std)

entity contador is
    port (
        clk, rst, ld: in std_logic;
        initial_value: in std_logic_vector(3 downto 0);
        count: out std_logic_vector(3 downto 0)
    );
end contador;

architecture beh of contador is
begin
    count_proc: process(clk, reset)
        variable count_i: unsigned(3 downto 0);
    begin
        if (rst='1') then
            count_i <= (others => '0');
        elsif (rising_edge(clk)) then
            if (ld = '1') then
                count_i <= unsigned(initial_value);
            else
                count_i <= count_i + 1;
            end if;
        end if;
        count <= std_logic_vector(count_i);
    end process count_proc;
end architecture;
```

FIN