EL LENGUAJE VHDL
CONCEPTOS BÁSICOS

Introducción

Entidades y arquitecturas

Sentencias y procesos

Objetos

Tipos de datos y operadores

Autores: Luis Entrena Arrontes, Celia López, Mario García, Enrique San Millán, Marta Portela, Almudena Lindoso

Historia del VHDL

- **1980: Programa VHSIC (Very High Speed Integrated Circuit) del Departamento de Defensa de E.E.U.U.**
- **1983: Comienzan los trabajos para desarrollar el VHDL**
- **1987: Aparece el estándar IEEE 1076-1987**
- **1994: Nueva versión del estándar: IEEE 1076-1993**
- **1996: Aparecen las primeras herramientas que soportan la nueva versión del estándar**
  - **Nos centraremos en la versión de 1987 ya que es la versión más universalmente aceptada**
- **2002: Nueva versión con pequeñas modificaciones**

El lenguaje VHDL

- Es un estándar de IEEE
- Ampliamente usado, principalmente en Europa
- Gran ámbito de aplicación
  - Lenguaje muy amplio que se adapta bien a las necesidades del diseño de circuitos digitales desde el nivel de sistema hasta el nivel lógico
  - Modelado y simulación de circuitos digitales en múltiples niveles de abstracción
  - Síntesis lógica, RT y de alto nivel

Características generales

- Jerarquía
- Soporte para la utilización de bibliotecas de diseño
- Diseño genérico
- Concurrencia
- Estilos de descripción
  - Estructural
  - Comportamental (“Behavioral”)
- Soporte para simulación (modelado) y síntesis
  - VHDL sintetizable es un subconjunto del VHDL simulable

Sintaxis utilizada durante el curso

- Letras mayúsculas para las palabras reservadas del lenguaje
- Los corchetes [ ] indican claúsulas opcionales de los comandos
- La barra vertical | indica elementos alternativos
- Los comentarios se indicarán de la misma manera que en VHDL, con dos guiones --
- Los ejemplos con código VHDL se encierran siempre en un recuadro

EL LENGUAJE VHDL
CONCEPTOS BÁSICOS

Introducción

Entidades y arquitecturas

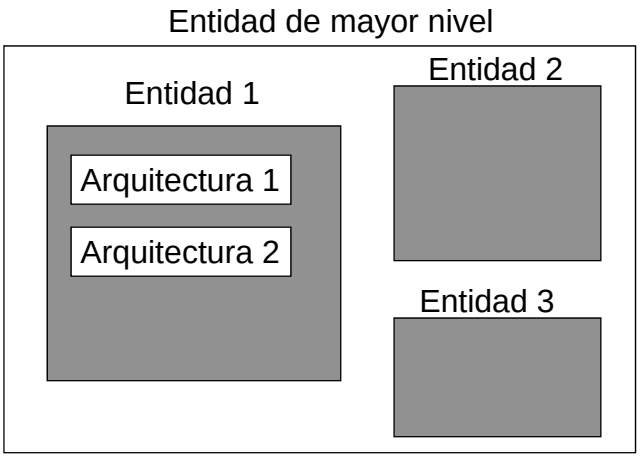
Sentencias y procesos

Objetos

Tipos de datos y operadores

Autor: Luis Entrena Arrontes

Entidades y arquitecturas



Entidades

- La entidad es el bloque básico de diseño
- La declaración de entidad contiene
  - Declaración de los parámetros genéricos de la entidad
  - La descripción del interfaz

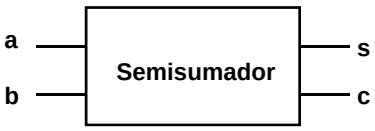
```
ENTITY inversor IS
        GENERIC (retraso: TIME := 5 NS);
        PORT (i1: IN BIT; o1: OUT BIT);
END inversor;
```

Arquitecturas

- La arquitectura define el comportamiento de la entidad
- Una entidad puede tener varias arquitecturas asociadas, que describen el comportamiento de la entidad de diferentes formas.
- Dentro de una arquitectura se pueden instanciar otras entidades, dando lugar a la jerarquía del diseño

```
ARCHITECTURE ejemplo OF inversor IS
        -- Declaraciones
BEGIN
        -- Operaciones
        o1 <= NOT  i1 AFTER retraso;
END ejemplo;
```


Ejemplo: un semisumador



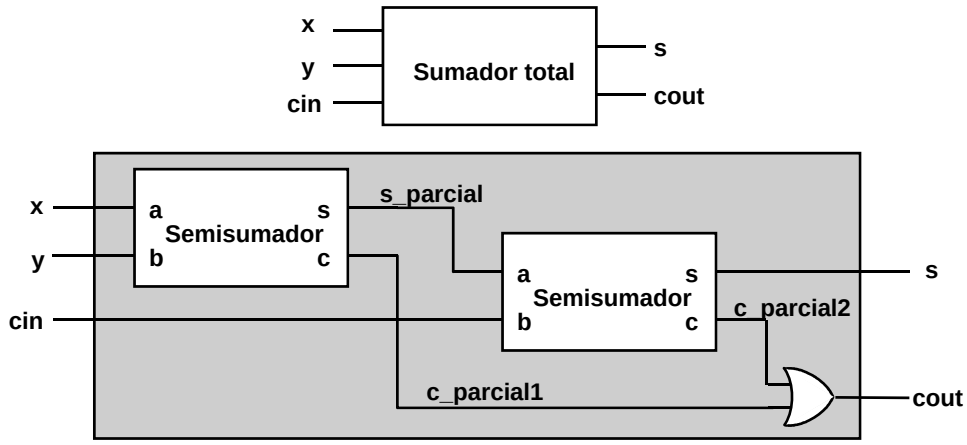
- Ecuaciones del semisumador
  - s = a xor b
  - c = a and b

El semisumador en VHDL

```
ENTITY semisumador IS
        PORT ( a: IN BIT;  b: IN BIT; s: OUT BIT; c: OUT BIT);
END semisumador;
```

```
ARCHITECTURE simple OF semisumador IS
BEGIN
        s <= a XOR b;
        c <= a AND b;
END simple;
```


Un sumador total



El sumador total en VHDL

```
ENTITY sumador_total IS
        PORT ( x: IN BIT; y: IN BIT; cin: IN BIT;
               s: OUT BIT; cout: OUT BIT);
END sumador_total;

ARCHITECTURE estructural OF sumador_total IS
        COMPONENT semisumador
                PORT ( a: IN BIT;  b: IN BIT; s: OUT BIT; c: OUT BIT);
        END COMPONENT;
        SIGNAL s_parcial: BIT;
        SIGNAL c_parcial1, c_parcial2: BIT;
BEGIN
        SS0: semisumador PORT MAP (x, y, s_parcial, c_parcial1);
        SS1: semisumador PORT MAP (s_parcial, cin, s, c_parcial2);
        cout <= c_parcial1 OR c_parcial2;
END estructural;
```


Asociación de puertos

☞ La asociación de puertos se ha realizado por posición

```
COMPONENT semisumador
                PORT ( a: IN BIT;  b: IN BIT; s: OUT BIT; c: OUT BIT);
END COMPONENT;

        .....

SS0: semisumador PORT MAP (x, y, s_parcial, c_parcial1);
```

Asociación de puertos explícita

- Alternativamente, la asociación también se puede indicar explícitamente

```
COMPONENT semisumador
                PORT ( a: IN BIT;  b: IN BIT; s: OUT BIT; c: OUT BIT);
END COMPONENT;

        ......

SS0: semisumador PORT MAP (a => x, b => y , s => s_parcial, c =>c_parcial1);
```


EL LENGUAJE VHDL
CONCEPTOS BÁSICOS

Introducción
Entidades y arquitecturas
Sentencias y procesos
Objetos
Tipos de datos y operadores

Autor: Luis Entrena Arrontes

Sentencias concurrentes

- Se ejecutan a la vez!
- Se pueden poner en cualquier orden
  - El simulador detecta los cambios en los valores de los objetos y determina cuando tiene que actualizarlos
- Todas las sentencias dentro de una arquitectura son concurrentes

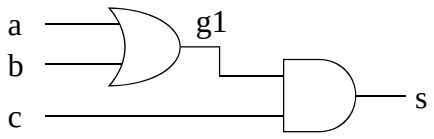
```
ARCHITECTURE estructural OF sumador_total IS
...
BEGIN
        SS0: semisumador PORT MAP (x, y, s_parcial, c_parcial1);
        SS1: semisumador PORT MAP (s_parcial, cin, s, c_parcial2);
        cout <= c_parcial1 OR c_parcial2;
END estructural;
```


Sentencias concurrentes

- Dos ejemplos equivalentes

```
ARCHITECTURE a OF circuito IS
BEGIN
        g1 <= a OR b;
        s <= g1 AND c;
END a;
```

```
ARCHITECTURE a OF circuito IS
BEGIN
        s <= g1 AND c;
        g1 <= a OR b;
END a;
```



Simulación de sentencias concurrentes

```
ARCHITECTURE a OF circuito IS
BEGIN
        g1 <= a OR b;
        s <= g1 AND c;
END a;
```

```
ARCHITECTURE a OF circuito IS
BEGIN
        s <= g1 AND c;
        g1 <= a OR b;
END a;
```

- Simulación:
  - Si hay un evento en a o b -> Calcular posible nuevo valor de g1
  - Si hay un evento en g1 o c -> Calcular posible nuevo valor de s
- Cada sentencia se ejecuta tantas veces como sea necesario
  - El orden en que se escriben las sentencias es irrelevante!

Sentencias secuenciales

- La concurrencia es difícil de manejar:
  - Podemos describir un diseño mediante sentencias secuenciales
- Sentencias secuenciales:
  - Se ejecutan una detrás de otra, como en los lenguajes de software
  - Las sentencias secuenciales van siempre dentro de procesos o de subprogramas, que determinan la sincronización con el resto del diseño
  - Entre la ejecución de una sentencia secuencial y la siguiente no transcurre el tiempo

Procesos

- Sirve para describir el hardware mediante sentencias secuenciales
- Contienen
  - Declaraciones
  - Sentencias secuenciales
- El proceso debe de tener una lista de sensibilidad explícita o al menos una sentencia WAIT
- Los procesos se ejecutan cuando
  - ocurre un evento en alguna de las señales de su lista de sensibilidad, o
  - se cumplen las condiciones de disparo de la sentencia WAIT en la que se paró
- Los procesos se ejecutan indefinidamente de forma cíclica

Ejemplo de proceso (I)

```
ARCHITECTURE una OF ejemplo IS
BEGIN
        PROCESS ( i1 )
                VARIABLE a: BIT;
        BEGIN
                a := NOT i1;      -- Sentencia secuencial
                o1 <=  a;         -- Sentencia secuencial
        END PROCESS;
END ejemplo;
```

Lista de sensibilidad

- El proceso se dispara cuando i1 cambia de valor
- Las sentencias dentro del proceso se ejecutan secuencialmente

Ejemplo de proceso (II)

```
ARCHITECTURE dos OF ejemplo IS
BEGIN
        PROCESS
                VARIABLE a: BIT;
        BEGIN
                a := NOT i1;    -- Sentencia secuencial
                o1 <= a;        -- Sentencia secuencial
                WAIT ON i1;     -- Sentencia secuencial
        END PROCESS;
END ejemplo;
```

- La ejecución del proceso se para en la sentencia WAIT
- La ejecución se reanuda cuando i1 cambia de valor
- Si se llega al final del proceso, se vuelve a comenzar por el principio

Algunas recomendaciones para la síntesis

- No utilizar claúsulas AFTER en descripciones que vayan a ser sintetizadas
  - Los sintetizadores suelen ignorar la claúsula AFTER en las asignaciones, ya que el retraso depende de la tecnología utilizada y no se puede fijar en el diseño
- Utilizar procesos con lista de sensibilidad
  - La sentencia WAIT sólo es sintetizable en casos muy concretos
  - La sentencia WAIT sólo es admitida por algunos sintetizadores

EL LENGUAJE VHDL
CONCEPTOS BÁSICOS

Introducción
Entidades y arquitecturas
Sentencias y procesos
Objetos
Tipos de datos y operadores

Autor: Luis Entrena Arrontes

Clases de objetos

- Constantes
- Variables
- Señales

Similares a las de los lenguajes de programación convencionales

Representan señales hardware cuyos valores evolucionan en el tiempo

Clases de objetos: Constantes

- Presentan un valor constante
- Se pueden declarar en cualquier ámbito

```
CONSTANT nombre1, nombre2, ..., nombren: tipo [ := valor ];
```

- Ejemplos

```
CONSTANT gnd : BIT := '0';
CONSTANT n, m : INTEGER := 3;
CONSTANT retraso : TIME := 10 NS;
```


Clases de objetos: Variables

- Pueden cambiar de valor
- El cambio de valor se produce inmediatamente tras la asignación
- Sólo se pueden declarar en ámbitos secuenciales, es decir, dentro de procesos o subprogramas
- Sólo son visibles dentro del proceso o subprograma en el que están declaradas. No existen variables globales

```
VARIABLE nombre1, nombre2, ..., nombren: tipo [ := valor ];
```

```
VARIABLE una_variable : BIT := '1';
VARIABLE i, j, k: INTEGER;
```

Clases de objetos: Señales

- Sus valores tienen siempre un componente temporal asociado
- Cuando se asigna un valor a una señal, el cambio de valor no se produce inmediatamente, sino después del tiempo especificado para que se produzca. El par *(valor, tiempo)* se denomina *transacción*
- Sólo se pueden declarar en ámbitos concurrentes, pero puede ser visibles en ámbitos secuenciales, es decir, dentro de un proceso o subprograma

```
SIGNAL nombre1, nombre2, ..., nombren: tipo [ := valor ];
```

```
SIGNAL una_señal : BIT := '1';
SIGNAL i, j, k: INTEGER;
```


Simulación de señales: Drivers

- En cada proceso donde se asigna un valor a una señal, existe un *driver* para dicha señal
- El driver de una señal almacena una cola de transacciones que representan los valores futuros de la señal.
- La primera transacción en un driver es el *valor actual del driver*
- El kernel se encarga de actualizar el valor de las señales a medida que avanza el tiempo: cuando la componente temporal de una transacción se iguala al tiempo actual, la señal se actualiza con el valor del driver

Asignaciones a señales

- Si dentro de un mismo proceso se asignan valores diferentes a una misma señal, solo se verá el último valor asignado

```
PROCESS (a)
BEGIN
        z <= a;
        z <= '1';
END PROCESS;
```

← *z siempre vale '1'*

Asignaciones a señales

- Si una misma señal se asigna en varias sentencias concurrentes a la vez, existirá un driver por cada proceso
- El valor de la señal se calcula mediante una *función de resolución*

```
PROCESS (a)
BEGIN
        z <= a;
END PROCESS;
```

→ z? ←

```
PROCESS (b)
BEGIN
        z <= b;
END PROCESS;
```

↓

Función de resolución

↓

Valor actual de la señal (resuelto)

Consideraciones acerca del uso de variables y señales

- Variables:
  - Son locales a los procesos. Sólo sirven para almacenar valores locales o intermedios
  - Ocupan menos espacio en memoria y son más eficientes en la simulación, ya que su actualización es inmediata
- Señales
  - Necesarias para expresar la concurrencia. Es la única manera de comunicar los procesos
- Asignaciones
  - De variable :=
  - De señal <=

EL LENGUAJE VHDL
CONCEPTOS BÁSICOS

Introducción
Entidades y arquitecturas
Sentencias y procesos
Objetos
Tipos de datos y operadores

Autor: Luis Entrena Arrontes

Índice

- Tipos de datos
- Operadores y funciones de conversión
- Atributos
- Interpretación de los tipos de datos en la síntesis

Tipos de datos

- VHDL es un lenguaje que posee un conjunto amplio de tipos de datos
- Todos los tipos de datos llevan asociada una restricción, que determina el rango de valores permitido
- Tipos
  - Escalares:
    - Enumerado
    - Entero
    - Real
    - Físico
  - Compuestos
    - Vector/Matriz
    - Registro
  - Fichero

Tipos escalares

- Enumerados
  - Los valores son identificadores o literales de un solo carácter
  - Pueden ser predefinidos o definidos por el usuario
- Entero
- Real
- Físicos
  - Un entero con unidades
  - Pueden ser predefinidos o definidos por el usuario
  - El tipo predefinido TIME es el único que tiene interés

Tipos enumerados definidos por el usuario

- Declaración de tipo

```
TYPE conjunto_de_letras IS ('A', 'B', 'C', 'D');
TYPE semaforo IS (verde, amarillo, rojo);
TYPE estados IS (s0, s1, s2, s3, s4, s5);
```

- Utilización

```
CONSTANT primera_letra: conjunto_de_letras := 'A';
SIGNAL semaforo1: semaforo;
SIGNAL estado_actual: estados;
...
estado_actual <= s0;
semaforo1 <= verde;
```

Tipos enumerados predefinidos

- BIT ('0', '1')
- BOOLEAN (FALSE, TRUE)
- CHARACTER (NUL, SOH, ..., 'A', 'B', ...)
- STD_LOGIC (definido en el estándar IEEE 1164)
  ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')

Tipo lógico estándar

- Definido por el estándar IEEE 1164. Es ampliamente utilizado
- Es un tipo enumerado para lógica de múltiples valores
  - 'U' - Sin inicializar. Es el valor por defecto
  - 'X' - Desconocido (fuerte)
  - '0' - Cero lógico (fuerte). Señal puesta a tierra
  - '1' - Uno lógico (fuerte). Señal puesta a alimentación
  - 'Z' - Alta impedancia
  - 'W' - Desconocido (débil)
  - 'L' - Cero (débil). Resistencias pull-down
  - 'H' - Uno (débil). Resistencias pull-up
  - '-' - Valor indiferente ("don't-care"). Usado en síntesis

Tipo lógico estándar

- Existen dos variantes del tipo lógico estándar
  - STD_ULOGIC: tipo no resuelto
  - STD_LOGIC: tipo resuelto
    - ✓ Una señal STD_LOGIC puede tener varios drivers
    - ✓ La función de resolución asigna un valor 'X' si los valores de los drivers no son compatibles
- Para usar los tipos lógicos estándar hay que añadir las siguientes líneas antes de la declaración de entidad o arquitectura donde se utilicen:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY...
```


Tipo entero

- Declaración y utilización

```
CONSTANT n: INTEGER := 8;
SIGNAL i, j, k: INTEGER;
...
i <= 3;
```

- Es conveniente indicar un rango
  - Delimita el tamaño de bits en la síntesis
  - La simulación da un error en caso de que el rango se exceda

```
SIGNAL valor_bcd: INTEGER RANGE 0 TO 9;
```

Tipo REAL

- Declaración y utilización

```
SIGNAL d: REAL;
...
d <= 3.1416;
d <= 10E5;
d <= 2.01E-3;
```

- No es sintetizable
  - No obstante, existen bibliotecas de componentes para operaciones con números reales

Tipos físicos

- Ejemplo de declaración y utilización

```
TYPE peso IS RANGE 0 TO 100.000.000
        UNITS
                gramo;
                kilo = 1000 gramo;
                tonelada = 1000 kilo;
        END UNITS;
SIGNAL p: peso;
```

- El tipo físico más importante es el tipo predefinido TIME

FS	femtosegundo = 10^{-15} seg.
PS	picosegundo = 10^{-12} seg.
NS	nanosegundo = 10^{-9} seg.
US	microsegundo = 10^{-6} seg.
MS	milisegundo = 10^{-3} seg.
SEC	segundo
MIN	minuto = 60 seg.
HR	hora = 60 minutos

```
SIGNAL t: TIME;
...
t  <= 10 NS;
```

Tipos compuestos: ARRAY

- Todos los elementos son del mismo tipo, que puede ser cualquier tipo VHDL

```
TYPE byte IS ARRAY ( 7 DOWNTO 0 ) OF STD_LOGIC;
SIGNAL s: byte;
```

- Se puede referenciar un elemento o un subconjunto de ellos

```
s <= "00000000";
s(2) <= '1';
s(4 downto 3) <= "00";
```

- El sentido puede ser ascendente (TO) o descendente (DOWNTO)
  - Los subconjuntos deben tener el mismo sentido que el vector del que proceden

Tipos compuestos: ARRAY

- Vectores de tamaño no especificado (el tamaño se especifica en una declaración posterior)

```
TYPE bit_vector IS ARRAY (NATURAL RANGE <>) OF BIT;
SIGNAL s: bit_vector (7 DOWNTO 0);
```

- Los vectores de BIT y STD_LOGIC están predefinidos

```
SIGNAL s: BIT_VECTOR (7 DOWNTO 0);
SIGNAL a: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

ARRAYS multidimensionales

- Matrices multidimensionales y vectores de vectores

```
TYPE matriz IS ARRAY(0 TO 479, 0 TO 479) OF STD_LOGIC;
SIGNAL imagen: matriz;
TYPE memoria1 IS ARRAY(0 TO 1023) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL ram1: memoria1;
TYPE memoria2 IS ARRAY(0 TO 1023) OF INTEGER RANGE 0 TO 255;
SIGNAL ram2: memoria2;
```

- Utilización

```
imagen(0,0) <= '1';
ram1(0)(0) <= '1';
ram1(0) <= "00000000";
imagen(0, 0 TO 7) <= "00000000";  -- ERROR
```


Tipos compuestos: RECORD

- Los elementos son de tipos diferentes

```
TYPE tipo_opcode IS (sta, lda, add, sub, and, nop, jmp, jsr);
TYPE tipo_operando IS BIT_VECTOR (15 DOWNTO 0);
TYPE formato_de_instruccion IS RECORD
        opcode   : tipo_opcode;
        operando: tipo_operando;
END RECORD;

SIGNAL ins1: formato_de_instruccón := (nop, "0000000000000000");

ins1.opcode <= lda;
ins1.operando <= "0011111100000000";
```

Literales

- Símbolos utilizados para representar valores constantes en VHDL
- Caracteres: siempre entre comillas simples

```
'0'  '1'  'Z'
```

- Cadenas de caracteres: entre dobles comillas

```
"Esto es un mensaje"
"00110010"
```

- Cadenas de bits: un prefijo indica el código utilizado

```
SIGNAL b: BIT_VECTOR (7 DOWNTO 0);
b <= B"11111111";          -- Binario
b <= X"FF";                -- Hexadecimal
b <= O"377";               -- Octal
```


Alias y subtipos

- Los alias permiten dar nombres alternativos a un objeto o a una parte de él

```
SIGNAL registro_de_estado: BIT_VECTOR (7 DOWNTO 0);
ALIAS bit_de_acarreo: BIT IS registro_de_estado(0);
ALIAS bit_z: BIT IS registro_de_estado(1);
```

- Los subtipos definen un subconjunto de un tipo definido previamente

```
SUBTYPE diez_valores IS INTEGER RANGE 0 TO 9;
```

Operadores predefinidos

Clase	Operadores	Tipo de operando	Tipo de resultado
Lógicos	NOT, AND, OR, NAND, NOR, XOR	BOOLEAN, BIT, STD_LOGIC	BOOLEAN, BIT, STD_LOGIC
Relacionales	=, /=, <, <=, >, >=	cualquier tipo	BOOLEAN
Aritméticos	+, -, *, /, **, MOD, REM, ABS	INTEGER, REAL, Físico, STD_LOGIC_VECTOR	INTEGER, REAL, Físico, STD_LOGIC_VECTOR
Concatenación	&	ARRAY & ARRAY ARRAY & elemento	ARRAY

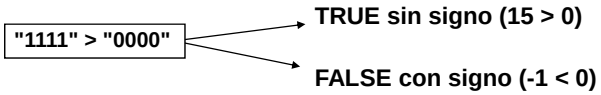
MOD y REM

- Resto de la división:
  - MOD toma el signo del divisor
  - REM toma el signo del dividendo

```
5 rem 3 = 2
5 mod 3 = 2
(–5) rem 3 = –2
(–5) mod 3 = 1
(–5) rem (–3) = –2
(–5) mod (–3) = –2
5 rem (–3) = 2
5 mod (–3) = –1
```

Operandos con signo y sin signo

- El resultado de algunas operaciones aritméticas sobre vectores de bits puede ser diferente dependiendo de si se consideran en una representación con signo o sin signo:



- Existen dos posibles soluciones:
  - Utilizar dos conjuntos de operadores diferenciados:
    - ✓ STD_LOGIC_UNSIGNED para operaciones sin signo
    - ✓ STD_LOGIC_SIGNED para operaciones con signo
  - Utilizar tipos de datos diferentes:
    - ✓ UNSIGNED para operaciones sin signo
    - ✓ SIGNED para operaciones con signo

Operandos con signo y sin signo

- Ejemplo de la primera solución

```
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
...
"1111" > "0000" -- sin signo
```

```
USE IEEE.STD_LOGIC_SIGNED.ALL;
...
"1111" > "0000" -- con signo
```

- Los paquetes STD_LOGIC_UNSIGNED y STD_LOGIC_SIGNED definen las mismas operaciones de forma diferente, por lo que no se pueden aplicar los dos a la vez sobre una misma unidad de diseño

Operandos con signo y sin signo

- Ejemplo de la segunda solución

modernamente:
USE IEEE.NUMERIC_STD.ALL;

```
USE IEEE.STD_LOGIC_ARITH.ALL;
...
SIGNAL u1, u2: UNSIGNED(3 DOWNTO 0);
SIGNAL s1, s2: SIGNED(3 DOWNTO 0);

-- sin signo
        u1 >u2
        UNSIGNED("1111") > UNSIGNED("0000")
-- con signo
        s1 > s2
        SIGNED("1111") > SIGNED("0000")
```


Funciones de conversión

- Para la conversión entre tipos de datos existen algunas funciones predefinidas
  - CONV_INTEGER
  - CONV_STD_LOGIC_VECTOR
- Estas funciones están definidas en STD_LOGIC_ARITH

```
USE IEEE.STD_LOGIC_ARITH.ALL;
...
SIGNAL s : STD_LOGIC_VECTOR ( 7 DOWNTO 0 );
SIGNAL i : INTEGER RANGE 0 TO 255;
...
i <= CONV_INTEGER(s);
s <= CONV_STD_LOGIC_VECTOR(i, 8)
```

Concatenación y agregación

- La concatenación permite concatenar vectores

```
SIGNAL s1, s2 : BIT_VECTOR ( 0 TO 3 );
SIGNAL x, z : BIT_VECTOR ( 0 TO 7 );
...
z <= s1 & s2;
z <= x(0 TO 6) & '0';
```

- La agregación permite conjuntar elementos para crear datos de un tipo compuesto (ARRAY o RECORD)

```
s1 <= ( '0', '1', '0', '0' );              -- Equivalente a s1 <= "0100";
s1 <= ( 2 => '1', OTHERS => '0');          -- Equivalente a s1 <= "0010";
s1 <= ( 0 to 2 => '1', 3 => s2(0) );
```


Atributos

- Denotan valores, funciones, tipos o rangos asociados con varios elementos del lenguaje
- Pueden ser definidos por el usuario o predefinidos
- Atributos predefinidos
  - Atributos de arrays: permiten acceder al rango, longitud o extremos de un array
  - Atributos de tipos: permiten acceder a elementos de un tipo
  - Atributos de señales: permiten modelar propiedades de las señales

Atributos de arrays

```
SIGNAL d: STD_LOGIC_VECTOR(15 DOWNTO 0)
```

Atributo	Descripción	Ejemplo	Resultado
‘LEFT	Límite izquierdo	d’LEFT	15
‘RIGHT	Límite derecho	d’RIGHT	0
‘HIGH	Límite superior	d’HIGH	15
‘LOW	Límite inferior	d’LOW	0
‘RANGE	Rango	d’RANGE	15 DOWNTO 0
‘REVERSE_RANGE	Rango inverso	d’REVERSE_RANGE	0 TO 15
‘LENGTH	Longitud	d’LENGTH	16

Atributos de tipos

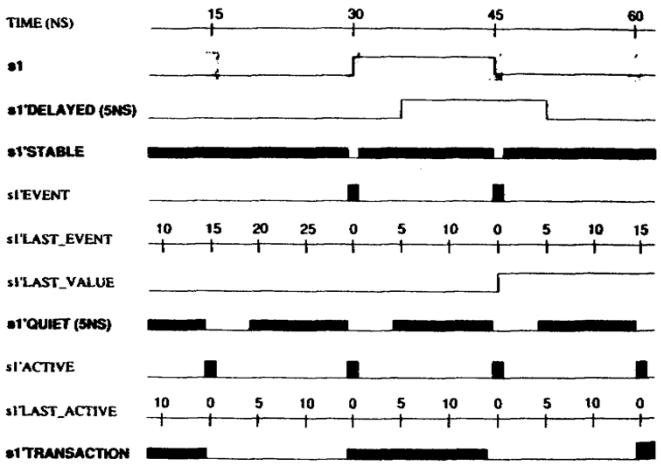
```
TYPE qit IS (‘0’, ‘1’, ‘Z’, ‘X’);
SUBTYPE tit IS qit RANGE ‘0’ TO ‘Z’;
```

Atributo	Descripción	Ejemplo	Resultado
‘BASE	Base del tipo	tit’BASE	qit
‘LEFT	Límite izquierdo (subtipo)	tit’LEFT	‘0’
‘RIGHT	Límite derecho (subtipo)	tit’RIGHT	‘Z’
‘HIGH	Límite superior (subtipo)	tit’HIGH	‘Z’
‘LOW	Límite inferior (subtipo)	tit’LOW	‘0’
‘POS(v)	Posición de v (tipo base)	tit’POS(‘X’)	3
‘VAL(p)	Valor en la posición p (t. base)	tit’VAL(3)	‘X’
‘SUCC(v)	Valor siguiente a v (tipo base)	tit’SUCC(‘Z’)	‘X’
‘PRED(v)	Valor anterior a v (tipo base)	tit’PRED(‘1’)	‘0’
‘LEFTOF(v)	Valor a la izquierda de v (t. base)	tit’LEFTOF(‘1’)	‘0’
‘RIGHTOF(v)	Valor a la derecha de v (t. base)	tit’RIGHTOF(‘1’)	‘Z’

Atributos de señales

Atributo	Tipo	Descripción
'DELAYED(t)	Señal	Genera una señal exactamente igual, pero retrasada t
'STABLE(t)	Señal BOOLEAN	Vale TRUE si la señal no ha cambiado de valor durante t
'EVENT	Valor BOOLEAN	Vale TRUE si la señal ha tenido un evento
'LAST_EVENT	Valor TIME	Tiempo transcurrido desde el último evento de la señal
'LAST_VALUE	Valor	El valor de la señal antes del último evento
'QUIET(t)	Señal BOOLEAN	Vale TRUE si la señal no ha recibido ninguna transacción durante t
'ACTIVE	Valor BOOLEAN	Vale TRUE si la señal ha tenido una transacción
'LAST_ACTIVE	Valor TIME	Tiempo transcurrido desde la último transacción en la señal
'TRANSACTION	Señal BIT	Cambia de valor cada vez que la señal recibe una transacción

Atributos de señales



Atributos definidos por el usuario

- Se pueden definir atributos de entidades, arquitecturas, tipos, objetos, etc...
- Primero hay que declarar el tipo del atributo

```
ATTRIBUTE <nombre_atributo> : <tipo>;
```

- Después hay que especificar su valor

```
ATTRIBUTE <nombre_atributo> OF <item> : <clase> IS <valor> ;
```

- Ejemplo

mi_diseño es una entidad

```
ATTRIBUTE tecnologia : STRING;
ATTRIBUTE tecnologia OF circuito: ENTITY IS “CMOS”;
...
        ... circuito’tecnologia ...
```


Interpretación de los tipos de datos para la síntesis

- INTEGER:
  - Se sintetiza como un número en binario natural.
  - Si el rango contiene valores negativos se sintetiza como un número en complemento a 2
  - Es importante indicar el rango para que la síntesis sea eficiente

```
SIGNAL a: INTEGER;                    -- 32 bits
SIGNAL b: INTEGER RANGE 0 TO 7;       -- 3 bits
```

- Enumerados:
  - Se sintetizan como un número en binario natural, asignando un código binario a cada valor en orden de aparición

Interpretación de los tipos de datos para la síntesis

- No son sintetizables
  - REAL
  - Físicos
  - Punteros
  - Ficheros
- Uso de atributos
  - Atributos de arrays: son útiles y sintetizables
  - Atributos de tipos: pueden ser sintetizables, aunque no se suelen utilizar
  - Atributos de señales: EVENT es el más usado. Los demás no son sintetizables, salvo STABLE

Valores iniciales

- Simulación:
  - Para poder simular un diseño, todas las señales y variables deben de tener un valor inicial. El valor inicial se puede añadir en la declaración del objeto
  - Si no se especifica un valor inicial, el lenguaje asume un valor por defecto. En tipos enumerados es el primero por la izquierda

```
TYPE estado IS (S0, S1, S2, S3);
SIGNAL s: estado := S3;          -- Valor inicial: S3
SIGNAL s: estado;                -- Valor inicial: S0
SIGNAL a: BIT;                   -- Valor inicial: '0'
SIGNAL b: STD_LOGIC;             -- Valor inicial: 'U'
```

Valores iniciales

- **Síntesis**
  - **Los sintetizadores no tienen en cuenta los valores iniciales**
  - **La inicialización debe de hacerse en el propio diseño (tal como ocurre en el hardware real)**

```
SIGNAL b: STD_LOGIC;
....
PROCESS (reset, ....)
BEGIN
        IF reset = '1' THEN
                b <= '0';
...
```