DISPOSITIVOS LÓGICOS PROGRAMABLES

LENGUAJE VHDL

http://es.wikibooks.org/wiki/Programaci%C3%B3n_en_VHDL

2011

Tabla de contenidos

ELEMENTOS SINTÁCTICOS DEL LENGUAJE VHDL

Comentarios: Los comentarios van precedidos de dos guiones "--". En una línea se ignorará todo aquello que vaya después de dos guiones seguidos.

```
-- Esto es un comentario
```

Símbolos especiales: Aparte de ciertas palabras reservadas, hay símbolos especiales que no pueden ser usados como parte de los identificadores, estos son:

+ - / * ( ) . , : ; & ‘ “ < > = | # ** => := /= >= <= <>

Identificadores: Son cualquier cadena de caracteres que sirven para identificar variables, señales, procesos, etc. Puede ser cualquier nombre compuesto por letras y/o números, incluyendo el símbolo de subrayado "_". Las mayúsculas y minúsculas son consideradas iguales, por lo tanto los identificadores *SALIDA* y *salida* representan el mismo elemento. No debe contener los símbolos especiales mostrados anteriormente, no debe empezar con número o subrayado, no debe acabar en subrayado, ni puede contener dos subrayados seguidos. Ningún identificador debe coincidir con alguna palabra reservada del lenguaje.

Números: Por defecto los números se representan en base 10, se admite la notación científica convencional. Para expresar números en una base diferente se usa el símbolo "#", ejemplo: 2#1100# (binario), 16#CF5A# (hexadecimal)

Caracteres: Se representan entre comillas simples, ejemplo : ‘1’ , ‘3’, ‘a’

Cadenas: Se representan entre comillas, ejemplo: “esto es una cadena”

Cadena de bits: Para definir una cadena de bits, no necesariamente debe convertirse a binario, pues puede usar la siguiente notación: B”100101” , O”1407” , X”FE4”

OPERADORES Y EXPRESIONES

Lógicos:
Actúan sobre los tipos bit, bit_vector y boolean. En el caso de utilizar este tipo de operadores en un vector, la operación se realizará bit a bit. Son: AND, OR, NAND, NOR, XOR, XNOR y NOT.

Aritméticos:
+ suma o signo positivo.
- resta o signo negativo.
* multiplicación.
/ división.
** exponencial,la base puede ser entero y real, pero el exponente sólo puede ser real . ejemplo: 4**2 sería 4^2.
ABS() valor absoluto
MOD módulo. Los operandos solo pueden ser enteros. Cumple con la ecuación: a=b*N+(a MOD b); donde N es un entero. El resultado toma el signo de b
REM resto de la división entera. Los operandos solo pueden ser enteros. Cumple con la ecuación: a=(a/b)*b+(a REM b); El resultado toma el signo de a

Relacionales:
Siempre devuelven un valor booleano (true o false). Los tipos de datos que pueden manejar son siempre de tipo escalar o matrices
== igualdad

/= desigualdad
> mayor que
>= mayor o igual que
< meor que
<= menor o igual que

Desplazamientos
SLL n (Shift Left Logic) desplazamiento lógico a la izquierda n veces.
SRL n (Shift Right Logic) desplazamiento lógico a la derecha n veces.
SLA n (Shift Left Arithmetic) desplazamiento aritmético a la izquierda n veces.
SRA n (Shift Right Arithmetic) desplazamiento aritmético a la derecha n veces.
ROL n (ROtate Left) rotación a la circular a la izquierda n veces.
ROR n (ROtate Right) rotación a la circular a la derecha n veces.
Ejemplos:
-- Inicialmente a vale 1100 1101
a sll 4 -- El resultado es 1101 0000
a sla 4 -- El resultado es 1101 1111 (el último bit se replica)
a srl 4 -- El resultado es 0000 1100
a sra 4 -- El resultado es 1111 1100 (el bit más significativo se replica)
a rol 4 -- El resultado es 1101 1100 (los primeros 4 bits pasan a la última posición)
a ror 4 -- El resultado es 1101 1100(los últimos 4 bits pasan a las primeras posiciones)

Concatenación:
& (concatenación): Concatena vectores de manera que la dimensión de la matriz resultante es la suma de las dimensiones de las matrices con las que se opera. Ejemplo c<=a&b construye la matriz c con la matriz a en las primeras posiciones y la matriz b en las últimas.

Orden de precedencia(de mayor a menor)

**, ABS, NOT
*, /, MOD, REM
+, - (signo)
+, -, & (operaciones)
=, /=, <, <=, >, >=
AND, OR, NAND, NOR, XOR

TIPOS DE DATOS

El VHDL es estricto con respecto al tipo de datos, todo objeto debe tener un tipo declarado explícitamente. Hay dos grupos: escalares y compuestos.

Tipos escalares: Son tipos simples que contienen algún tipo de magnitud.

Enteros: Contienen un valor numérico entero. No se dice explícitamente que es un tipo entero, pues se definen con la palabra RANGE que indica el intervalo de valores que puede contener. Ejemplo;

```
TYPE byte IS RANGE 0 TO 255;
TYPE index IS RANGE 7 DOWNTO 1;
```

Reales: Llamados también “punto flotante”. Se definen igual que los enteros con la diferencia que los límites son números reales.Ejemplo:

```
TYPE nivel IS RANGE 0.0 TO 5.0
```

Físicos: Se trata de datos que corresponden con magnitudes físicas, que tienen un valor y unas unidades. Ejemplo:

```
TYPE longitud IS RANGE 0 TO 1.0e9
  UNITS
    um;
    mm=1000 um;
    m=1000 mm;
    in=25.4 mm;
  END UNITS;
```

Enumerados: Son datos que puede tomar cualquier valor especificado en una lista o conjunto finito. Este conjunto se indica mediante una lista encerrada entre paréntesis. Ejemplo:

```
TYPE dia IS (lunes, martes,miércoles,jueves,viernes,sábado,domingo);
```

Tipos compuestos

Matrices: Se trata de un conjunto de elementos del mismo tipo, accesibles mediante un índice. Los hay de dos tipos: monodimensionales o multidimensionales.

```
TYPE word IS ARRAY (31 DOWNTO 0) OF bit;
TYPE transformada IS ARRAY (1 TO 4, 1 TO 4) OF real;
```

A los elementos de una matriz se accede individualmente mediante índices: *word(3)* , o puede acceder a un rango de datos: *word(2 to 9)*

Registros: Es equivalente al tipo record de otros lenguajes.

```
TYPE trabajador IS
  RECORD
    nombre : string;
    edad : integer;
  END RECORD;
```

Para referirse a un elemento dentro del registro se usa un punto entre el nombre del registro y el nombre del campo: trabajador.nombre="Carlos"

Subtipos de datos
Son restricciones o subconjuntos de tipos existentes. Hay dos tipos:

Restriccion de un tipo escalar a un rango:

```
SUBTYPE raro IS integer RANGE 4 TO 7;
SUBTYPE digitos IS character RANGE '0' TO '9';
```

Restriccion del rango de una matriz:

```
SUBTYPE id IS string(1 TO 20);
SUBTYPE word IS bit_vector(31 DOWNTO 0);
```

La ventaja de utilizar un subtipo es que la operaciones que sirven para el tipo, también sirven para el subtipo.

Conversión de tipos
En ocasiones puede ser necesario convertir unos tipos a otros, esta operación es conocida como casting. Algunas de las conversiones son automáticas, como puede ser el paso de entero a real, otras conversiones deben realizarse de forma explícita, indicando el nombre del tipo al que se quiere pasar seguido del valor entre paréntesis.

```
real(15);
```

```
integer(3.5);
```

En muchos diseños es necesario realizar conversiones entre bits y enteros. A continuación se muestran varias funciones de conversión entre ambos tipos.

```
conv_integer(std_logic_vector); -- Conversión de vector a entero
conv_std_logic_vector(integer, numero_bits); -- Conversión de entero a vector de numero_bits de tamaño
```

Atributos

Los elementos en VHDL como señales, variables,etc pueden tener información adicional llamada atributos. Se manejan usando : el nombre, una comilla simple y el atributo.Se incluyen algunos ejemplos:

t'left límite izquierdo del tipo t
t´right límite derecho del tipo t
t'low límite inferior del tipo t
t´high límite superior del tipo t
t´pos(x) posición de x dentro del tipo t
t´val(N) Elemento N del tipo t

s'event devuelve true si se ha producido un cambio en la señal s
s'stable(tiempo) devuelve true si la señal ha estado estable durante tiempo

Atributos definidos por el usuario

Es posible asignar información adicional a los objetos que se definen en VHDL.

```
ATTRIBUTE nombre : tipo
ATTRIBUTE nombre OF id_elemento : clase_elemento IS valor
```

El id_elemento en un objeto declarado en alguna parte. La clase_elemento es el tipo de elemento al que se le va añadir dicho atributo.El valor será lo que devuelva al preguntar por dicho atributo.

DECLARACIÓN DE CONSTANTES, SEÑALES Y VARIABLES

Constantes

Es un elemento que se inicializa en un determinado valor que no puede ser cambiado:

```
CONSTANT e : real := 2.71828;
CONSTANT retraso : time := 10 ns;
```

Variables

Su valor que puede ser cambiado en cualquier instante, y es posible asignarles un valor inicial.

```
VARIABLE contador : natural := 0;
VARIABLE aux : bit_vector(31 DOWNTO 0);
```

A un elemento o a una parte del mismo es posible asignarle un identificador adicional:

```
VARIABLE instruccion : bit_vector(31 DOWNTO 0);
ALIAS cod_op : bit_vector(7 DOWNTO 0) IS instruccion(31 DOWNTO 24);
```

Señales

Las señales se declaran igual que las variables y pueden ser: normal (por defecto), register o bus.

```
SIGNAL sel : bit := '0';
SIGNAL datos : bit_vector(7 DOWNTO 0);
```

Una señal es diferente a una variable. Las variables sólo tienen sentido dentro de un bloque (PROCESS) o un subprograma, y tienen sentido en entornos donde las sentencia se ejecutan en serie. La señales pueden ser declaradas únicamente en las arquitecturas, paquetes (PACKAGE) , o en los bloques concurrentes (BLOCK). Las señales tienen un equivalente físico, y representan conexiones en el circuito, las variables no lo tienen. Las señales son visibles por todos los procesos, y pueden ser usadas en cualquier parte del programa.

DECLARACIÓN DE IDENTIDAD Y ARQUITECTURA

DECLARACIÓN DE ENTIDAD:

```
ENTITY nombre IS
  [GENERIC(lista de parámetros);]
  [PORT(lista de puertos);]
END [ENTITY] nombre;
```

La instrucción GENERIC define y declara propiedades o constantes del módulo. Las constantes declaradas en esta sección son como los parámetros en las funciones de cualquier otro lenguaje de programación, por lo que es posible introducir valores, en caso contrario tomará los valores por defecto. Para declarar una constante se indicará su nombre seguido de dos puntos y el tipo del que se trata, finalmente se indicará el valor al que es inicializado mediante el operador de asignación :=. En el caso que existan más constantes se terminará con un punto y coma, la última constante no lo llevará.

```
nombre_constante : tipo := inicializacion;
```

La instruccion PORT definen las entradas y salidas del módulo definido. Basicamente consiste en indicar el nombre de la señal seguido de dos puntos y la dirección del puerto (se verá más adelante), además del tipo de señal del que se trata. Al igual que antes, si existe más de una señal se finalizará con un punto y coma, exceptuando la última señal de la lista.

```
nombre_señal : dirección tipo;
```

A continuación se muestra un ejemplo de una entidad, con una serie de constantes y señales de entrada y salida.

```
ENTITY mux
  GENERIC(
    C_AWIDTH : integer := 32;
    C_DWIDTH : integer := 32
  );
  PORT(
    control  : IN bit;
    entrada1 : IN bit;
    entrada2 : IN bit;
    salida   : OUT bit
  );
END mux;
```

Direcciones de los puertos de una entidad
Las señales representarían la función que harían los cables en un diseño hardware tradicional, es decir, sirven para transportar información y establecer conexiones. Dentro de una entidad los puertos son considerados como señales, en donde se pueden diferenciar varios tipos.

IN: Son señales de entrada, las cuales sólo se pueden leer, pero no se le pueden asignar ningún valor, es decir, no se puede modificar el valor que poseen. Por lo tanto, su funcionalidad es similar a las constantes.
OUT: Corresponden a las señales de salida, en este caso su valor puede ser modificado, pero en este caso no pueden leerse, es decir no pueden ser utilizadas como argumentos en la asignación de cualquier elemento.
INOUT: Este tipo es una mezcla de los dos anteriores, pueden ser utilizados tanto como de lectura o de escritura.
BUFFER: Es idéntico al anterior, con la diferencia de que sólo una fuente puede modificar su valor.

DECLARACIÓN DE ARQUITECTURA:

```
ARCHITECTURE nombre OF nombre_entidad IS
    [declaraciones]
  BEGIN
    [sentencias concurrentes]
  END [ARCHITECTURE] [nombre];
```

Declaraciones: pueden aparecer varias instrucciones para indicar la declaración de señales, componentes, funciones, etc.. Estas señales son internas, es decir, a ellas no se puede acceder desde la entidad, por los que los circuitos de nivel superior no podrían acceder a ellas. En un símil con un microprocesador, estas señales podrían ser las líneas que comunican la unidad central con la ALU, a las que no se puede acceder directamente desde el exterior del microprocesador. Obsérvese que en este caso no se indica si son entradas o salidas, puesto que al ser internas pueden ser leídas o escritas sin ningún problema. En esta parte de la arquitectura también pueden aparecer otros elementos, como pueden ser las constantes. Lo siguiente es la palabra clave BEGIN, que da paso a la descripción del circuito, mediante una serie de sentencias

Un ejemplo de una arquitectura podría ser la siguiente.

```
  ARCHITECTURE mux_rtl OF mux IS
    SIGNAL int1, int2, int3 : BIT;
  BEGIN
    int1 <= NOT control;
    int2 <= entrada1 AND int1;
    int3 <= entrada2 AND S;
    salida <= int2 OR int3;
  END mul_rtl;
```

ESTILOS DE ARQUITECTURAS:

La descripción puede ser de tres tipos:

1.Descripción de flujo de datos
2.Descripción de comportamiento
3.Descripción estructural

Descripción de flujo de datos

A la hora de plantearse crear un programa en VHDL no hay que pensar como si fuera un programa típico para ordenador. No hay que olvidar que en VHDL hay que describir un hardware, algo que no se hace en un programa para ordenador. Un circuito electrónico puede tener muchos elementos que estén ejecutando acciones a la vez, por ejemplo en un circuito puede tener una entrada que se aplique a dos compuertas lógicas y de cada una obtener una salida, en este caso tendría dos caminos en los que se ejecutarían acciones (las compuertas lógicas) de forma paralela. Esto es lo que se llama concurrencia.

VHDL es un lenguaje concurrente, como consecuencia no se seguirá el orden en que están escritas las instrucciones a la hora de ejecutar el código. De hecho, si hay dos instrucciones, no tiene porqué ejecutarse una antes que otra, pueden ejecutarse a la vez.

Sentencias Concurrentes

La instrucción básica de la ejecución concurrente es la asignación entre señales a través del símbolo <=. Para facilitar la asignación de las señales VHDL incluye elementos de alto nivel como son instrucciones condicionales, de selección, etc, que se verán a continuación.

WHEN ... ELSE

Sentencia de selección múltiple. En hardware es necesario incluir todas las opciones posibles. En este caso es obligatorio siempre acabar la expresión con un ELSE.

```
<señal> <= <asignación1> WHEN <condición1> ELSE
        <asignación2> WHEN <condición2> ELSE
                ...
        <asignaciónN> WHEN <condiciónN> ELSE
        <asignaciónM>;
```

Un posible ejemplo de este tipo de sentencias podría ser la siguiente.

```
s <= "00" WHEN a = b ELSE
     "01" WHEN a > b ELSE
     "11";
```

Siempre es obligatorio asignar algo, aunque es posible no realizar ninguna acción, para ello se utiliza la palabra reservada UNAFFECTED. De esta forma se asignará el mismo valor que tenía la señal.

```
s1 <= d1 WHEN control = '1' ELSE UNAFFECTED;
s2 <= d2 WHEN control = '1' ELSE s2;
```
Las dos sentencias anteriores parecen iguales, pero en la segunda se produce una transacción, aspecto que en la primera no sucede.

WITH ... SELECT ... THEN

Es similar a las sentencias CASE o SWITCH de C. La asignación se hace según el contenido de un objeto o resultado de cierta expresión.

```
WITH <señal1> SELECT
  <señal2> <= <asignación1> WHEN <estado_señal1>,
            <asignación2> WHEN <estado_señal2>,
                        ...
            <asignaciónN> WHEN OTHERS;
```
Un ejemplo de esta sentencia es la siguiente.

```
WITH estado SELECT
  semaforo <= "rojo"     WHEN "01",
            "verde"    WHEN "10",
            "amarillo" WHEN "11",
            "roto"     WHEN OTHERS;
```

La cláusula WHEN OTHERS especifica todos los demás valores que no han sido contemplados. También es posible utilizar la opción que se contempló en el caso anterior (UNAFFECTED).

BLOCK

En ocasiones interesa agrupar un conjunto de sentencias en bloques. Estos bloques permiten dividir el sistema en módulos, estos módulos pueden estar compuestos de otros módulos. La estructura general es la siguiente.

```
block_id;
BLOCK(expresión de guardia)
  cabecera
  declaraciones
BEGIN
  sentencias concurrentes
END BLOCK block_id;
```
El nombre del bloque es opcional (block_id), al igual que la expresión de guardia. Un ejemplo de esto podría ser el siguiente.

```
latch: BLOCK(clk='1')
BEGIN
  q <= GUARDED d;
END BLOCK latch;
```

Descripción de comportamiento

Como la programación concurrente no siempre es la mejor forma de describir ideas, VHDL incorpora la programación serie, la cual se define en bloques indicados con la sentencia PROCESS. En un mismo diseño puede haber varios bloques de este tipo, cada uno de estos bloques corresponderá a una instrucción concurrente. Es decir, internamente la ejecución de las instrucciones de los PROCESS es serie, pero entre los bloques es concurrente.

A continuación se verán la estructuras más comunes de la ejecución serie y sus características.

PROCESS

Un PROCESS, como se ha dicho antes, es una sentencia concurrente en el sentido de que todos los PROCESS y todas las demás sentencias concurrentes se ejecutarán sin un orden establecido. No obstante las sentencias que hay dentro del PROCESS se ejecutan de forma secuencial.

Por lo tanto se puede decir que una estructura secuencial va en el interior de un PROCESS.

La estructura genérica de esta sentencia es:

```
PROCESS [lista de sensibilidad]
  [declaración de variables]
BEGIN
  [sentencias secuenciales]
END PROCESS;
```

La lista de sensibilidad es una serie de señales que, al cambiar de valor, hacen que se ejecute el PROCESS.

Un ejemplo sería:

```
PROCESS(señal1, señal2)
...
```

El PROCESS anterior sólo se ejecutará cuando señal1 o señal2 cambien de valor.

Variables y Señales

Hay que distinguir las señales y las variables, las señales se declaran entre ARCHITECTURE y su correspondiente BEGIN mientras que las variables se declaran entre PROCESS y su BEGIN. Dentro de un PROCESS pueden usarse ambas, pero hay una diferencia importante entre ellas: las señales sólo se actualizan al terminar el proceso en el que se usan, mientras que las variables se actualizan instantáneamente, es decir, su valor cambia en el momento de la asignación.

Unos ejemplos son:

```
ENTITY ejemplo
  PORT (c: IN std_logic;
    d: OUT std_logic);
END ENTITY;

ARCHITECTURE ejemplo_arch OF ejemplo IS
  SIGNAL a,b: std_logic;
BEGIN
  PROCESS(c)
```

```
        VARIABLE z: std_logic;
      BEGIN
        a<= c and b;    --asignación de señales: después de ejecutarse esta línea a seguirá valiendo lo mismo, sólo se actualiza al acabar el PROCESS
        z:= a or c;     --asignación de variables: en el momento de ejecutarse esta línea z valdrá a or c (el valor que tenía a cuando empezó el PROCESS)
      END PROCESS;
    END ARCHITECTURE;
```

Sentencias secuenciales

IF ... THEN ... ELSE

Permite la ejecución de un bloque de código dependiendo de una o varias condiciones.

```
    IF <condición1> THEN
      [sentencias 1]
    ELSIF <condición2> THEN
      [sentencias 2]
    ELSE
      [sentencias N]
    END IF;
```

Un ejemplo es:

```
    IF (reloj='1' AND enable='1') THEN
      salida<=entrada;
    ELSIF (enable='1') THEN
      salida<=tmp;
    ELSE
      salida<='0';
    END IF;
```

CASE

Es parecido al anterior porque también ejecuta un bloque de código condicionalmente, pero en esta ocasión se evalúa una expresión en vez de una condición. Se debe recordar que se deben tener en cuenta todos los casos, es decir, incluir como última opción la sentencia WHEN OTHERS.

```
   CASE <expresión> IS
     WHEN <valor1> => [sentencias1]
     WHEN <valor2> => [sentencias2]
     WHEN <rango de valores> => [sentenciasN]
     WHEN OTHERS => [sentenciasM]
   END CASE;
```

Un ejemplo es:

```
CASE a IS
  WHEN 0        =>    B:=0;
  WHEN 1 to 50  =>    B:=1;
  WHEN 99 to 51 =>    B:=2;
  WHEN OTHERS   =>    B:=3;
END CASE;
```

LOOP

LOOP es la forma de hacer bucles en VHDL. Sería el equivalente a un FOR o WHILE de un lenguaje convencional.

Su estructura es:

```
[etiqueta:] [WHILE <condición> | FOR <condición>] LOOP
  [sentencias]
  [exit;]
  [next;]
END LOOP [etiqueta];
```

Un ejemplo de bucles anidados es:

```
bucle1: LOOP
  a:=A+1
  b:=20;
  bucle2: LOOP
    IF b < (a*b) THEN
      EXIT bucle2;
    END IF;
    b:=b+a;
  END LOOP bucle2;
  EXIT bucle1 WHEN a>10;
END LOOP bucle1;
```

Otro ejemplo, este con FOR es:

```
bucle1: FOR a IN 1 TO 10 LOOP
  b:=20;
  bucle2: LOOP
    IF b<(a*a) THEN
      EXIT bucle2;
    END IF;
    b:=b-a;
  END LOOP bucle2;
END LOOP bucle1;
```

Otro más con WHILE

```
    cuenta := 10;
    bucle1: WHILE cuenta >= 0 LOOP
      cuenta := cuenta + 1;
      b:=20;
      bucle2: LOOP
        IF b<(a*a) THEN
          EXIT bucle2;
        END IF;
        b := b-a;
      END LOOP bucle2;
    END LOOP bucle1;
```

NEXT y EXIT

NEXT permite detener la ejecución actual y seguir con la siguiente.

```
    [id_next:]
    NEXT [id_bucle] [WHEN condición];
```

Como se puede suponer, la sentencia EXIT hace que se salga del bucle superior al que se ejecuta.

```
    [id_exit:]
    EXIT [id_bucle] [WHEN condición];
```

Se puede ver su uso en los ejemplos del apartado anterior.

WAIT

La ejecución de un bloque PROCESS se realiza de forma continuada, como si de un bucle infinito se tratara (se ejecutan todas las sentencias y se vuelven a repetir). Esto no tiene mucho sentido, puesto que continuamente se ejecutaría lo mismo una y otra vez, sería interesante poder parar la ejecución. Una forma de hacerlo e mediante las listas de sensibilidad, las cuales se han visto anteriormente, aunque existe otra forma de hacerlo mediante la sentencia WAIT, pero es algo más complejo.

```
    WAIT ON lista_sensible UNTIL condicion FOR timeout;
```

La lista_sensible es un conjunto de señales separadas por comas. La condición es una sentencia que activará de nuevo la ejecución. El timeout es el tiempo durante el cual la ejecucion esta detenida.

No es necesario utilizar las tres opciones, en caso de hacerlo la primera condición que se cumpla volverá a activar la ejecución.

```
    WAIT ON pulso;
    WAIT UNTIL counter = 5;
    WAIT FOR 10 ns;
    WAIT ON pulso, sensor UNTIL counter = 5;
    WAIT ON pulso UNTIL counter = 5 FOR 10 ns;
```

Si se utiliza una lista de sensibilidad no es posible utilizar la sentencia WAIT, sin embargo si es posible utilizar varias sentencias WAIT cuando esta actúa como condición de activación. Este comando se estudiará en el subcapítulo de retrasos, en la sección de bancos de prueba.

Descripción estructural

Definición de componentes

En VHDL es posible declarar componentes dentro de un diseño mediante la palabra COMPONENT. Un componente se corresponde con una entidad que ha sido declarada en otro módulo del diseño, o incluso en alguna biblioteca, la declaración de este elemento se realizará en la parte declarativa de la arquitectura del módulo que se está desarrollando. La sintaxis para declarar un componente es muy parecida a la de una entidad.

```
COMPONENT nombre [IS]
  [GENERIC(lista_parametros);]
  [PORT(lista_de_puertos);]
END COMPONENT nombre;
```

Si se dispone de un compilador de VHDL'93 no será necesario incluir en los diseño la parte declarativa de los componentes, es decir se pasaría a referenciarlos de forma directa. Un ejemplo de un componentesería:

```
---------------------------------------------------------------------------
  COMPONENT mux IS
    GENERIC(
      C_AWIDTH : integer;
      C_DWIDTH : integer );
    PORT(
      control  : IN  bit;
      entrada1 : IN  bit;
      entrada2 : IN  bit;
      salida   : OUT bit );
  END COMPONENT mux;
---------------------------------------------------------------------------
```

Referencia de componentes

La referencia de componentes consiste en copiar en la arquitectura aquel componente que se quiera utilizar, tantas veces como sea necesario para construir el diseño. Para ello, la sintaxis que presenta la instanciación de un componente es la siguiente.

```
ref_id:
  [COMPONENT] id_componente | ENTITY id_entidad [(id_arquitectura)] | CONFIGURATION id_configuración
  [GENERIC MAP (parametros)]
  [PORT MAP (puertos)];
```

Un ejemplo de referenciación del componente anterior sería.

```
---------------------------------------------------------------------------
  mux_1 : mux
    GENERIC MAP (
      C_AWIDTH => C_AWIDTH,
      C_DWIDTH => C_DWIDTH   )
    PORT MAP (
      control  => ctrl,
      entrada1 => e1,
      entrada2 => e2,
      salida   => sal );
---------------------------------------------------------------------------
```

Las señales ctrl, e1, e2 y sal deben ser declaradas previamente en la sección de declaraciones de la arquitectura, estas señales sirven para poder conectar unos componentes con otros. También deben declararse las variables que se utilizan en la sección GENERIC.

Subprogramas

Como en otros lenguajes de programación, en VHDL es posible estructurar el código mediante el uso de subprogramas. Realmente, un subprograma es una función o procedimiento que realiza una determinada tarea, aunque existen ciertas diferencias entre ambas.

Una función devuelve un valor y un procedimiento devuelve los valores a través de los parámetros que le han sido pasados como argumentos. Por ello, las primeras deberán contener la palabra reservada RETURN, mientras que las segundas no tienen necesidad de disponer dicha sentencia, en caso de tener una sentencia de ese tipo interrumpirá la ejecución del procedimiento.
A consecuencia de la anterior, en una función todos sus parámetros son de entrada, por lo que sólo pueden ser leidos dentro de la misma, por el contrario en un procedimiento los parámetros pueden ser de entrada, de salida o de entrada y salida (unidireccionales o bidireccionales).
Las funciones se usan en expresiones, sin embargo, los procedimientos se llaman como una sentencia secuencial o concurrente.
Los procedimientos pueden tener efectos colaterales al poder cambiar señales externas que han sido pasadas como parámetros, por otro lado las funciones no poseen estos efectos.
Las funciones nunca pueden tener una sentencia WAIT, pero los procedimientos sí.

Declaración de funciones y procedimientos

Las declaraciones de estos elementos pueden realizarse en la parte declarativas de las arquitecturas, bloques, paquetes, etc. A continuación, se muestra la estructura de un procedimiento.

```
PROCEDURE nombre[(parámetros)] IS
  [declaraciones]
BEGIN
  [sentencias]
END [PROCEDURE] [nombre];
```

La estructura de las funciones corresponden a las siguientes líneas.

```
[PURE | IMPURE]
FUNCTION nombre[(parámetros)] RETURN tipo IS
  [declaraciones]
BEGIN
  [sentencias]        -- Debe incluir un RETURN
END [FUNCTION] [nombre];
```

Como ya se explicó, la lista de parámetros es opcional en ambos casos. Estos parámetros se declaran de forma similar a como se hacen los puertos de una entidad.

```
<nombre del puerto> : <tipo de puerto> <tipo de objeto>
```

Dependiendo de la estructura que se utilice, funciones o procedimientos, los parámetros tendrán un significado u otro. En las funciones sólo es posible utilizar el tipo de puerto IN, mientras que en los procedimientos pueden usarse los tipos IN, OUT e INOUT. Además, en las funciones el parámetro puede ser CONSTANT o SIGNAL, por defecto es CONSTANT. Por otro lado, en los procedimientos los parámetros de tipo IN son CONSTANT por defecto y VARIABLE para el resto, aunque también es posible utilizar SIGNAL siempre que se declare explícitamente. No se aconseja utilizar señales como parámetros, por los efectos que pueden tener en la ejecución de un programa.

Las funciones PURE o puras devuelven el mismo valor para unos parámetros de entrada determinados. Mientras que una función es IMPURE o impura si para los mismos valores de entrada se devuelve distinto valor. Estas últimas pueden depender de una variable o señal global. Realmente, estas palabras reservadas hacen de comentario, puesto que una función no se hace impura o pura por indicarlo.

Un ejemplo de una función sería el siguiente.

```
--------------------------------------------------------------------------------

    FUNCTION es_uno(din : std_logic_vector(31 downto 0)) RETURN std_logic IS
      VARIABLE val : std_logic;
    BEGIN
      IF din = X"00000001" THEN
        val := '1';
      ELSE
        val := '0';
      END IF;
      RETURN val;
    END es_uno;

--------------------------------------------------------------------------------
```

Por otro lado, un ejemplo de un procedimiento podría pertenecer a las siguientes líneas.

```
--------------------------------------------------------------------------------

    PROCEDURE es_uno(din : std_logic_vector(31 downto 0)
                     dout : std_logic) IS
    BEGIN
      IF din = X"00000001" THEN
        dout := '1';
      ELSE
        dout := '0';
      END IF;
    END es_uno;

--------------------------------------------------------------------------------
```

En la asignación a la señal de salida se realiza con el operador :=, puesto que no es una señal, es decir que se trata de un tipo VARIABLE. Si se tratara de una señal se haría con el operador de asignación <=.

Llamadas a subprogramas

Es muy sencillo realizar una invocación a un subprograma, basta con indicar el nombre de dicho subprograma seguido de los argumentos, los cuales irán entre paréntesis. En VHDL existen varias formas de pasar los parámetros a un subprograma.

Asociación implícita: Poniendo los parámetros en el mismo orden en el que se declaran en el subprograma. Por ejemplo, si se dispone del siguiente procedimiento.

```
--------------------------------------------------------------------------------

    PROCEDURE limite(CONSTANT conj : IN std_logic_vector(3 DOWNTO 0);
                     VARIABLE min, max : INOUT integer) IS
                          ...

    limite(cjt(31 DOWNTO 28), valmin, valmax); -- Llamada al procedimiento

--------------------------------------------------------------------------------
```

Asociación explícita: Los parámetros se colocan en cualquier orden. Un ejemplo de la llamada al procedimiento anterior podría ser la siguiente.

```
limite(min=>valmin, max=>valmax, conj=>cjt(31 DOWNTO 28));
```

Parámetros libres: En VHDL es posible dejar parámetros por especificar, de forma que tengan unos valores por defecto, y en el caso de pasar un valor a dichos argumentos puedan tomar el valor especificado. Un ejemplo de este tipo de llamadas sería el siguiente.

```
PROCEDURE lee(longuitud : IN integer := 200) IS
BEGIN
        ....
END PROCEDURE;

-- Posibles llamadas
lee;
lee(350);
```

Como ya se indicó al inicio de este capítulo, los subprogramas pueden ser llamados desde un entorno secuencial o concurrente. En caso de ser llamado en un entorno concurrente el procedimiento se ejecutará de forma similar a un bloque PROCESS, por lo que hay que tener alguna sentencia que permita suspender la ejecución, porque podría ejecutarse de forma continua. Este tipo de sentencias podría ser una lista sensible o una sentencia WAIT, como ya se explicó en los bloques PROCESS, como se indicó, no es posible utilizar la lista de sensibilidad junto con una sentencia WAIT. Este tipo de sentencias es posible que detengan un procedimiento para siempre si son utilizadas en entornos secuenciales.

Sobrecarga de operadores

Como en otros lenguajes de programación, en VHDL también es posible sobracargar los métodos, es decir, tener funciones con el mismo nombre pero con distintos parámetros. Aunque, en este lenguaje hay que tener un poco de cuidado a la hora de declarar los procedimientos, puesto que al ser posible dejar libres algunos argumentos es muy probable encontrar situaciones en las que dos funciones son llamadas de forma idéntica cuando se hace sin parámetros. Por ejemplo, si se dispone de los siguientes procedimientos la llamada a éstos sin parámetros sería la misma, y no habría forma de diferenciar a qué método se refiere.

```
PROCEDURE lee(longuitud : IN integer := 20) IS
BEGIN
        ....
END PROCEDURE;

PROCEDURE lee(factor : IN real := 100.0) IS
BEGIN
        ....
END PROCEDURE;

lee;
```

Paquetes

Como se comentó al principio, un paquete consta de un conjunto de subprogramas, contantes, declaraciones, etc., con la intención de implementar algún servicio. Así se pueden hacer visibles las interfaces de los subprogramas y ocultar su descripción.

Definición de paquetes

Los paquetes se separan en dos zonas: declaraciones y cuerpo, aunque esta última puede ser eliminada si no se definen funciones y/o procedimientos. Las siguientes líneas muestra la estructura de un paquete.

```
-- Declaración de paquete
PACKAGE nombre IS
  declaraciones
END [PACKAGE] [nombre];

-- Declaración del cuerpo
PACKAGE BODY nombre IS
  declaraciones
  subprogramas
    ...
END [PACKAGE BODY] [nombre];
```

El nombre del paquete debe coincidir en la declaración del paquete y del cuerpo. A continuación, se muestra un ejemplo de un paquete.

```
-- Declaración de paquete
PACKAGE mi_paquete IS
  SUBTYPE dir_type IS std_logic_vector(31 DOWNTO 0);
  SUBTYPE dato_type IS std_logic_vector(15 DOWNTO 0);
  CONSTANT inicio : dir_type;    -- Hay que definirlo en el BODY
  FUNCTION inttodato(valor : integer) RETURN dato_type;
  PROCEDURE datotoint(dato : IN dato_type; valor : OUT integer);
END mi_paquete;

-- Declaración del cuerpo
PACKAGE BODY mi_paquete IS
  CONSTANT inicio : dir_type := X"FFFF0000";

  FUNCTION inttodato(valor : integer) RETURN dato_type IS
    -- Cuerpo de la función
  END inttodato;

  PROCEDURE datotoint(dato : IN dato_type; valor : OUT integer) IS
    -- Cuerpo del procedimiento
  END datotoint;

END [PACKAGE BODY] [nombre];
```

Para acceder a los tipos creados en un paquete, se debe indicar el nombre del paquete seguido del elemento que se desea utilizar, separados por un punto. Para el ejemplo anterior sería algo parecido a las siguientes líneas.

```
VARIABLE dir : work.mi_paquete.dir_type;
dir := work.mi_paquete.inicio;
```

Existe otra forma de realizarlo, haciendo visible al paquete de esta forma no se necesitará el punto ni tampoco indicar el nombre del paquete. Para ello, se debe utilizar la sentencia USE seguido del paquete que se va a utilizar y el método a utilizar, todo ello separado por puntos. El ejemplo anterior se podría realizar de la siguiente forma.

```
USE work.mi_paquete.ALL
VARIABLE dir : dir_type;
dir := inicio;
```

Programación en VHDL/Bancos de pruebas

En VHDL es posible describir modelos para la simulación. Estos modelos no tienen demasiadas restricciones, necesitando unicamente un intérprete de las instrucciones VHDL. En cambio, en la síntesis se añaden una cantidad de restricciones como pueden ser aquellas que tienen que ver con las del tiempo, ya que no es posible aplicar retardos a la hora de diseñar un circuito.

Retrasos

El retraso es uno de los elementos más importantes de la simulación, puesto que el comportamiento de un circuito puede cambiar dependiendo del cambio de las diferentes señales. Cuando se realiza una asignación se produce de forma inmediata, puesto que no se ha especificado ningún retraso. Este comportamiento puede ser alterado mediante la opción AFTER cuando se asigna un valor a una señal. Su sintaxis corresponde a la siguiente línea.

```
señal <= valor AFTER tiempo;
```

Donde tiempo es un valor de tiempo indicado en us, ns, ms, ... Un ejemplo puede ser el siguiente.

```
rst <= '0' AFTER 15 ns;
```

Pero esta sentencia es mucho más compleja, por ejemplo se puede asignar inicialmente un valor y modificarlo posteriormente o incluso que sea modificado cada cierto tiempo.

```
signal clk : std_logic := '0';
      ....
```

```
rst <= '1', '0' AFTER 15 ns; -- Inicialmente rst=1, despues de 15 ns rst=0
clk <= not clk AFTER 5 ns; -- Cada 5 ns clk cambia de valor
```

También es posible introducir una espera entre dos sentencias mediante la palabra reservada WAIT. La sintáxis de esta operación e más compleja que la anterior.

```
WAIT [ON lista | UNTIL condicion | FOR tiempo];
```

Lista corresponde a una lista de sensibilidad de señales, es decir, se permanecerá en espera hasta que se produzca un cambio en alguna de las señales de la lista. Condición se trata de una espera indeterminada hasta que la condición sea verdadera. Por último, el tiempo es un valor definido en una unidad de tiempo. A continuación se muestra un ejemplo de las tres esperas posibles.

```
-- WAIT ON
stop <= '1';
WAIT ON semaforo; -- Hasta que semaforo no cambie se permanecerá en el WAIT
stop <= '0';

  ....

-- WAIT UNTIL
ack <= '1';
WAIT UNTIL clk'event and clk = '1'; -- Hasta que no exista un evento de clk y sea uno se permanece en WAIT
ack <= '0';

  ....

-- WAIT FOR
start <= '0';
WAIT FOR 50 ns; -- Se espera 50 ns
start <= '1';
```

Descripción de un banco de pruebas

Una de las partes más importantes en el diseño de cualquier sistema son las pruebas para la verificación del funcionamiento de un sistema. Con las metodologías tradicionales la verificación sólo era posible tras su implementación física, lo que se traducía en un alto riesgo y coste adicional. Lo más sencillo es cambiar las entradas para ver cómo son las salidas, en una herramienta de simulación, siempre que su diseño sea sencillo, en caso contrario lo más cómodo sería crear un banco de pruebas.

Un banco de pruebas es una entidad sin puertos, cuya estructura contiene un componente que corresponde al circuito que se desea simular y la alteración de las diferentes señales de entrada a dicho componente, para poder abarcar un mayor número de casos de prueba. Es recomendable realizar ls descripción del banco de pruebas de un sistema a la vez que se describe su diseño. Las siguientes líneas muestra la sintaxis de un banco de pruebas.

```
ENTITY nombre_test IS
END nombre_test;
ARCHITECTURE test OF nombre_test IS
  -- Declaraciones
```

```
BEGIN
  -- Cuerpo de las pruebas
END test;
```

A continuación se muestran las diferentes metodologías que se pueden llevar a cabo para la realización de un banco de pruebas. Para su explicación se utilizará un ejemplo de un diseño muy sencillo, donde un vector es rotado un bit hacia la izquierda o derecha dependiendo de la entrada met, su entidad corresponde al siguiente trozo de código.

```
ENTITY round IS
  clk, rst, met : IN std_logic;
  e : IN std_logic_vector(3 DOWNTO 0);
  s : OUT std_logic_vector(3 DOWNTO 0)
END round;
```

Método tabular

Para verificar la funcionalidad de un diseño se debe elaborar una tabla con las entradas y las respuestas que se esperan a dichas entradas. Todo ello se deberá relacionar mediante código VHDL. Las siguientes líneas muestran un ejemplo con el diseño que se expuso anteriormente.

```
USE work.round;

ENTITY test_round IS
END test_round;

ARCHITECTURE test OF test_round IS
  SIGNAL clk, rst, met : std_logic;
  SIGNAL e, s : std_logic_vector(3 DOWNTO 0);
  TYPE type_test IS RECORD
    clk, rst, met : std_logic;
    e, s : std_logic_vector(3 DOWNTO 0);
  END RECORD;
  TYPE lista_test IS ARRAY (0 TO 6) OF type_test;
  CONSTANT tabla_test : lista_test :=(
    (clk='0', rst ='1', met='0', e="0000", s="0000"),
    (clk='1', rst ='0', met='0', e="0000", s="0000"),
    (clk='1', rst ='0', met='0', e="0001", s="0010"),
    (clk='1', rst ='0', met='0', e="1010", s="0101"),
    (clk='1', rst ='0', met='1', e="0000", s="0000"),
    (clk='1', rst ='0', met='1', e="0001", s="1000"),
    (clk='1', rst ='0', met='1', e="1001", s="1100"));
BEGIN
  r : ENTITY work.round
  PORT MAP(clk => clk, rst => rst, met => met, e => e, s => s);

  PROCESS
    VARIABLE vector : type_test;
    VARIABLE errores : boolean := false;
  BEGIN
```

```
    FOR i IN 0 TO 6 LOOP
      vector := tabla_test(i);
      clk<=vector.clk; rst<=vector.rst; met<=vector.met; e<=vector.e;
      WAIT FOR 20 ns;
      IF s /= vector.s THEN
        ASSERT false REPORT "Salida incorrecta" SEVERITY error;
        errores:=true;
      END IF;
    END LOOP;
    ASSERT errores REPORT "Test OK" SEVERITY note;
    WAIT;
  END PROCESS;
END test;
```

Uso de archivos (vectores de test)

En el caso antrior los casos de prueba y el código de simulación permanecían juntos, pero es posible separarlos de forma que por un lado se encuentren las pruebas y por otro el código. Esto es posible ya que VHDL dispone de paquetes de entradas/salida para la lectura/escritura en archivos de texto, como ya se comento el paquete textio dispone de los subprogramas necesarios para el acceso a dichos archivos.

Supóngase los casos de pruebas desarrollados en el caso anterior, los cuales se han escrito en el siguiente archivo de texto (vectores de test).

```
 clk  rst  met   e     s
  0    1    0   0000  0000
  1    0    0   0001  0010
  1    0    0   1010  0101
  1    0    1   0000  0000
  1    0    1   0001  1000
  1    0    1   1001  1100
```

A continuación se muestra el código relacionado con la simulación, en él se incluye el acceso al archivo anterior, el cual contiene los diferentes vectores de test.

```
USE std.textio.ALL; -- No es necesario se incluye por defecto
USE work.round;

ENTITY test_round IS
END test_round;

ARCHITECTURE test OF test_round IS
  SIGNAL clk, rst, met : std_logic;
  SIGNAL e, s : std_logic_vector(3 DOWNTO 0);
BEGIN
  r : ENTITY work.round
  PORT MAP(clk => clk, rst => rst, met => met, e => e, s => s);

  PROCESS
    FILE vector_test : text OPEN read_mode IS "test.txt";
    VARIABLE errores : boolean := false;
    VARIABLE vector : line;
    VARIABLE clk_tmp, rst_tmp, met_tmp : std_logic;
```

```
    VARIABLE e_tmp, s_tmp : std_logic_vector(3 DOWNTO 0);
  BEGIN
    readline(vector_test,vector); -- Lee los nombres (la primera linea)
    WHILE NOT endfile(vector_test) LOOP
      readline(vector_test,vector);
      read(vector,clk_tmp);
      read(vector,rst_tmp);
      read(vector,met_tmp);
      read(vector,e_tmp);
      read(vector,s_tmp);
      clk <= clk_tmp; rst <= rst_tmp; met <= met_tmp; e <= e_tmp;
      WAIT FOR 20 ns;
      IF s_tmp /= s THEN
        ASSERT false REPORT "Salida incorrecta" SEVERITY error;
        errores:=true;
      END IF;
    END LOOP;
    file_close(vector_test);
    ASSERT errores REPORT "Test OK" SEVERITY note;
    WAIT;
  END PROCESS;
END test;
```

Metodología algorítmica

Existe otro tipo de test, los cuales se basan en realizar algoritmos para cubrir el mayor número de casos posibles. A continuación se muestra un ejemplo, el cual aplica esta metodología.

```
USE work.round;

ENTITY test_round IS
END test_round;

ARCHITECTURE test OF test_round IS
  SIGNAL clk : std_logic := '0';
  SIGNAL rst, met : std_logic;
  SIGNAL e, s : std_logic_vector(3 DOWNTO 0);
BEGIN
  clk <= NOT clk after 10 ns;

  r : ENTITY work.round
  PORT MAP(clk => clk, rst => rst, met => met, e => e, s => s);

  PROCESS
  BEGIN
    rst <= '1'; met <= '0'; e <= "0000";
    WAIT FOR 20 ns;
    ASSERT (s="0000") REPORT "Error en reset" SEVERITY error;

    rst <= '0';

    e <= "0000";
    WAIT FOR 20 ns;
```

```
    ASSERT (s="0000") REPORT "Error desplazamiento izquierda" SEVERITY error;

    e <= "0001";
    WAIT FOR 20 ns;
    ASSERT (s="0010") REPORT "Error desplazamiento izquierda" SEVERITY error;

    e <= "1010";
    WAIT FOR 20 ns;
    ASSERT (s="0101") REPORT "Error desplazamiento izquierda" SEVERITY error;

    met <= '1';

    e <= "0000";
    WAIT FOR 20 ns;
    ASSERT (s="0000") REPORT "Error desplazamiento derecha" SEVERITY error;

    e <= "0001";
    WAIT FOR 20 ns;
    ASSERT (s="1000") REPORT "Error desplazamiento derecha" SEVERITY error;

    e <= "1001";
    WAIT FOR 20 ns;
    ASSERT (s="1100") REPORT "Error desplazamiento derecha" SEVERITY error;

    ASSERT false REPORT "Test Finalizado" SEVERITY note;
    WAIT;
  END PROCESS;
END test;
```

Ejemplos

Compuerta tres estados

El objetivo es crear una compuerta que tenga una señal de operación la cual, a estado alto, habilite la salida, por lo tanto el valor de la entrada pasará a la salida. Cuando la señal de operación esté a nivel bajo la compuerta no sacará una señal, es decir, estará en alta impedancia.

Entradas:
entrada: entrada de datos.
op: señal que indica el modo de funcionar de la compuerta.
Salidas:
salida: salida de datos.

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

ENTITY triestado IS
  PORT(op, entrada: IN std_logic;
       salida: OUT std_logic);
END triestado;

ARCHITECTURE synth OF triestado IS
BEGIN
  PROCESS(entrada,op)
  BEGIN
    IF op='1' THEN
      salida <= entrada;
    ELSE
      salida <= 'Z';
    END IF;
  END PROCESS;
END ARCHITECTURE synth;
```

Multiplexor

El objetivo es crear un sistema que devuelva un valor dependiente de otra señal de entrada, la cual será la encargada de seleccionar la salida. Además se definirán varias entradas de datos que actuarán como salidas. Cuando la señal de selección este a cero no se producirá ninguna salida, es decir el valor será cero.

Entradas:
EntradaX: entradas de datos.
sel: señal que indica la señal que va a ser devuelta.
Salidas:
salida: salida de datos.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

ENTITY mux IS
  PORT(c      : IN std_logic_vector(3 DOWNTO 0);
       sel    : IN std_logic_vector(1 DOWNTO 0);
       salida : OUT std_logic_vector(3 DOWNTO 0));
END mux;

ARCHITECTURE synth OF mux IS
BEGIN

  PROCESS (sel, a, b, c) IS
  BEGIN
    CASE sel IS
      WHEN "00" => salida <= (others => '0');
      WHEN "01" => salida <= a;
      WHEN "10" => salida <= b;
      WHEN "11" => salida <= c;
      WHEN OTHERS => salida <= (others => '0');
    END CASE;
  END PROCESS;
END synth;
```

SUMADOR

El objetivo es crear un sumador que dadas dos entradas de datos devuelva la suma de estos.

Entradas:
a: operando 1.
b: operando 2.
Salidas:
salida: suma de las entradas.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

ENTITY sum IS
  PORT (a : IN std_logic_vector(3 DOWNTO 0);
        b : IN std_logic_vector(3 DOWNTO 0);
        salida : OUT std_logic_vector(4 DOWNTO 0));
END sum;

ARCHITECTURE synth OF sum IS
BEGIN

  PROCESS (a, b) IS
  BEGIN
```

```
    salida <= a + b;
  END PROCESS;
END synth
```

Contador

El objetivo es crear un contador con dos salidas, las cuales corresponderán a un contador rápido y a otro lento. Ademas, se dispondrá de cuatro señale de entrada, donde habrá dos reseteadores (uno para cada contador) y dos enables, que permitirán la parada e inicio de los contadores. También se incluirá el reset y el reloj del sistema.

Entradas:
rst: Reset del sistema. clk: Reloj del sistema. reset1: Reset del contador rápido.
reset2: Reset del contador lento.
enable1: Activación-Desactivación del contador rápido. enable2: Activación-Desactivación del contador lento.

Salidas:
count1: Salida del contador rápido. count2: Salida del contador lento.
Los enables (enable1 y enable2) no actúan como enables puros, realmente cuando se produce un evento en dicha entrada pasará a activarse o desactivarse el contador correspondiente.

```
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY count IS
  PORT (clk : IN std_logic;
        rst : IN std_logic;
        enable1 : IN std_logic;
        enable2 : IN std_logic;
        reset1 : IN std_logic;
        reset2 : IN std_logic;
        count1 : OUT std_logic_vector(7 DOWNTO 0);
        count2 : OUT std_logic_vector(31 DOWNTO 0));
END count;

ARCHITECTURE synth OF sum IS
  SIGNAL cnt1 : std_logic_vector(7 DOWNTO 0) := (others => '0');
  SIGNAL cnt2 : std_logic_vector(31 DOWNTO 0) := (others => '0');
  SIGNAL en1 : boolean := false;
  SIGNAL en2 : boolean := false;
BEGIN

  pSeq : PROCESS (clk, rst) IS
  BEGIN
    IF rst = '1' THEN
      cnt1 <= (others => '0');
      cnt2 <= (others => '0');
    ELSIF clk='1' AND clk'event THEN
      IF en1 THEN
        cnt1 <= cnt1 + 1;
      END IF;

      IF en2 THEN
        cnt2 <= cnt2 + 1;
      END IF;
    END IF;
```

```
  END PROCESS;

  pCom : PROCESS (enable1, enable2) IS
  BEGIN
    IF enable1 = '1' THEN
      en1 <= NOT en1;
    END IF;

    IF enable2 = '1' THEN
      en2 <= NOT en2;
    END IF;
  END PROCESS;

  count1 <= cnt1;
  count2 <= cnt2;

END synth;
```

Biestable-Latch

El objetivo es crear un dispositivo capaz de reproducir en cada tic de reloj la entrada en la salida. Para ello será necesario diponer del reloj y reset del sistema.

Entradas:
rst: Reset del sistema.
clk: Reloj del sistema.
a: Entrada de datos.
Salidas:
b: Salida de datos.

```
--------------------------------------------------------------------------------

  LIBRARY IEEE;
  USE IEEE.STD_LOGIC_1164.all;

  ENTITY biestable IS
    PORT (clk : IN std_logic;
          rst : IN std_logic;
          a : IN std_logic_vector(31 DOWNTO 0);
          b: OUT std_logic_vector(31 DOWNTO 0));
  END biestable;

  ARCHITECTURE synth OF biestable IS
  BEGIN

    pSeq : PROCESS (clk, rst) IS
    BEGIN
      IF rst = '1' THEN
        b <= (others => '0');
      ELSIF clk='1' AND clk'event THEN
        b <= a;
      END IF;
    END PROCESS;
   END synth;
--------------------------------------------------------------------------------
```

También es posible realizar esta operación a través de una señal de activacion (enable), pero a este tipo de diseños se les denomina latch. En este caso no hará falta la señal de reloj, pero sí el reset.

Entradas:
rst: Reset del sistema.
en: Enable de almacenamiento.
a: Entrada de datos.
Salidas:
b: Salida de datos.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.all;

ENTITY latch IS
  PORT (en : IN std_logic;
        rst : IN std_logic;
        a : IN std_logic_vector(31 DOWNTO 0);
        b: OUT std_logic_vector(31 DOWNTO 0));
END latch;

ARCHITECTURE synth OF latch IS
BEGIN

  pSeq : PROCESS (en, rst) IS
  BEGIN
    IF rst = '1' THEN
      b <= (others => '0');
    ELSIF en = '1' THEN
      b <= a;
    END IF;
  END PROCESS;

END synth;
```

Máquinas de estados

El objetivo es crear un sistema que genere unas salidas determinadas, dependiendo de los estados por donde va fluyendo la ruta de datos. Por consiguiente, hará falta el reset y reloj del sistema como entradas del sistema. También se añade una señal para sacar el estado al exterior.

Entradas:
rst: Reset del sistema.
clk: Reloj del sistema.
Salidas:
a: Salida de datos.
b: Salida de datos.
estado: Salida del estado.

```
LIBRARY IEEE;
```

```
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;

ENTITY maquina_estados IS
  PORT (clk : IN std_logic;
        rst : IN std_logic;
        a : OUT std_logic;
        b : OUT std_logic;
        estado : OUT std_logic_vector(3 downto 0));
END maquina_estados;

ARCHITECTURE synth OF maquina_estados IS
  SIGNAL pstate, n_state : std_logic_vector(3 downto 0);
BEGIN

-- maquina de estados

PROCESS (clk, rst)
BEGIN
  IF rst = '1' THEN
    pstate <= "0000";
  ELSIF clk = '1' AND clk'event THEN
    pstate <= n_state;
  END IF;
END PROCESS;

estado <= pstate;

n_state <= "0001" WHEN (pstate = "0000") ELSE
              "0010" WHEN (pstate = "0001") ELSE
           "0011" WHEN (pstate = "0010") ELSE
              "0100" WHEN (pstate = "0011") ELSE
           "0101" WHEN (pstate = "0100") ELSE
              "0011" WHEN (pstate = "0101") ELSE
              "0111" WHEN (pstate = "0011") ELSE
              "1000" WHEN (pstate = "0111") ELSE
              "1001" WHEN (pstate = "1000") ELSE
           "0000";

a <= '1' WHEN pstate = "0000" ELSE --0
        '0' WHEN pstate = "0001" ELSE --1
        '0' WHEN pstate = "0010" ELSE --2
        '1' WHEN pstate = "0011" ELSE --3
        '1' WHEN pstate = "0100" ELSE --4
        '1' WHEN pstate = "0101" ELSE --5
        '1' WHEN pstate = "0110" ELSE --6
        '1' WHEN pstate = "0111" ELSE --7
        '1' WHEN pstate = "1000" ELSE --8
        '1' WHEN pstate = "1001" ELSE --9
        '0';

b <= '1' WHEN pstate = "0000" ELSE --0
        '1' WHEN pstate = "0001" ELSE --1
        '1' WHEN pstate = "0010" ELSE --2
        '1' WHEN pstate = "0011" ELSE --3
        '1' WHEN pstate = "0100" ELSE --4
```

```
        '0' WHEN pstate = "0101" ELSE --5
        '0' WHEN pstate = "0110" ELSE --6
        '1' WHEN pstate = "0111" ELSE --7
        '1' WHEN pstate = "1000" ELSE --8
        '1' WHEN pstate = "1001" ELSE --9
        '0';

END synth;
```

ALU

El objetivo es crear un dispositivo capaz de realizar cualquier operación dependiendo del valor de una señal de entrada, además se dispondrá de dos entradas de datos.

Entradas:
a: Entrada de datos 1.
b: Entrada de datos 2.
proceso: Entrada de la operación.
Salidas:
c: Salida de datos.

```
LIBRARY ieee;
USE iee.std_logic_1164.ALL;

ENTITY alu IS
PORT(a : IN std_logic_vector(7 DOWNTO 0);--entrada 2
        b : IN std_logic_vector(7 DOWNTO 0);--entrada 1
    proceso : IN std_logic_vector(3 DOWNTO 0);--que hara la alu
        c : OUT std_logic_vector(15 DOWNTO 0));
END alu;

ARCHITECTURE synth OF alu IS
BEGIN
  PROCESS (a, b, proceso)
  BEGIN
    CASE proceso IS
         WHEN "0000" => c <= "0000000" & (a + b);
      WHEN "0001" =>
        IF a < b THEN
          c <= "1000000" & (b - a);
        ELSE
          c <= "00000000" & (a - b);
        END IF;
      WHEN "0010" =>
        .........
      WHEN OTHERS => null;
    END CASE;
  END PROCESS;
END synth;
```

ANEXOS

EJEMPLOS BÁSICOS

```
----------------------------------------
-- MUX 2 A 1 ESTRUCTURAL
----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
-- Uncomment the following library if instantiating Xilinx primitives
library UNISIM;
use UNISIM.VComponents.all;

entity mux21_est is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          s : in  STD_LOGIC;
          c : out  STD_LOGIC);
end mux21_est;

architecture Behavioral of mux21_est is
signal aux1,aux2,s_neg:std_logic;
begin
inversor: inv port map(i=>s,o=>s_neg);
and_1: and2 port map(i0=>a,i1=>s,o=>aux1);
and_2: and2 port map(i0=>b,i1=>s_neg,o=>aux2);
or_1: or2 port map(i0=>aux1,i1=>aux2,o=>c);
end Behavioral;

----------------------------------
-- mux21 estilo flujo de datos
----------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux21_flujo is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          s : in  STD_LOGIC;
          c : out  STD_LOGIC);
end mux21_flujo;

architecture Behavioral of mux21_flujo is
signal not_s,aux1,aux2:std_logic;
begin
not_s<=not s;
aux1<=a and s;
aux2<=b and not_s;
c<=aux1 or aux2;
end Behavioral;

--------------------------------------------------------------------------------
-- mux21 funcional
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux21_func is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          s : in  STD_LOGIC;
          c : out  STD_LOGIC);
end mux21_func;

architecture Behavioral of mux21_func is
begin
process (a,b,s)
begin
  if s='0' then
    c<=a;
  else
    c<=b;
  end if;
end process;

end Behavioral;

----------------------------------
-- mux21_func_01
----------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux21_func_01 is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          s : in  STD_LOGIC;
          c : out  STD_LOGIC);
end mux21_func_01;

architecture Behavioral of mux21_func_01 is

begin
  c <= a when s='1' else
       b when s='0';
end Behavioral;
--------------------------------------------------------------------------------
-- mux_21_func_02
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux21_func_02 is
   Port ( a : in  STD_LOGIC;
          b : in  STD_LOGIC;
          s : in  STD_LOGIC;
          c : out  STD_LOGIC);
end mux21_func_02;

architecture Behavioral of mux21_func_02 is

begin

  with s select
    c <= a when '1',
         b when others;
end Behavioral;

----------------------------------------------------
--mux21_esquema
```

```
----------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
library UNISIM;
use UNISIM.Vcomponents.ALL;

entity mux21_esquema is
  port ( a : in    std_logic;
         b : in    std_logic;
         s : in    std_logic;
         c : out   std_logic);
end mux21_esquema;

architecture BEHAVIORAL of mux21_esquema is
  attribute BOX_TYPE   : string ;
  signal XLXN_1 : std_logic;
  signal XLXN_5 : std_logic;
  signal XLXN_6 : std_logic;
  component INV
    port ( I : in    std_logic;
           O : out   std_logic);
  end component;
  attribute BOX_TYPE of INV : component is "BLACK_BOX";

  component AND2
    port ( I0 : in    std_logic;
           I1 : in    std_logic;
           O  : out   std_logic);
  end component;
  attribute BOX_TYPE of AND2 : component is "BLACK_BOX";

  component OR2
    port ( I0 : in    std_logic;
           I1 : in    std_logic;
           O  : out   std_logic);
  end component;
  attribute BOX_TYPE of OR2 : component is "BLACK_BOX";

begin
  XLXI_1 : INV
    port map (I=>s,
              O=>XLXN_1);

  XLXI_2 : AND2
    port map (I0=>s,
              I1=>a,
              O=>XLXN_5);

  XLXI_3 : AND2
    port map (I0=>XLXN_1,
              I1=>b,
              O=>XLXN_6);

  XLXI_4 : OR2
    port map (I0=>XLXN_6,
              I1=>XLXN_5,
              O=>c);

end BEHAVIORAL;

---------------------------------------
-- MUX 2 A 1 ESTRUCTURAL_LUT
---------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Uncomment the following library declaration if instantiating
-- any Xilinx primitives in this code.
library UNISIM;
use UNISIM.VComponents.all;

entity mux21_LUT is
    Port ( a : in  STD_LOGIC;
           b : in  STD_LOGIC;
           s : in  STD_LOGIC;
           c : out  STD_LOGIC);
end mux21_LUT;

architecture Behavioral of mux21_LUT is
begin
 LUT3_inst : LUT3
   generic map (
      INIT => b"11011000")-- msb........lsb
-- a          b          s          |          c
-----------------
-- 0          0          0          |          0
-- 0          0          1          |          0
-- 0          1          0          |          0
-- 0          1          1          |          1
-- 1          0          0          |          1
-- 1          0          1          |          0
-- 1          1          0          |          1
-- 1          1          1          |          1

   port map (
      O => c,   -- LUT general output
      I0 => s, -- LUT input
      I1 => b, -- LUT input
      I2 => a  -- LUT input
   );

end Behavioral;

----------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY prueba IS
END prueba;

ARCHITECTURE behavior OF prueba IS

    -- Component Declaration for the Unit Under Test (UUT)

    COMPONENT mux21_est
    PORT(
         a : IN  std_logic;
         b : IN  std_logic;
         s : IN  std_logic;
         c : OUT  std_logic
        );
    END COMPONENT;


   signal a : std_logic := '0';
   signal b : std_logic := '0';
   signal s : std_logic := '0';
   signal c : std_logic;

BEGIN

        -- Instantiate the Unit Under Test (UUT)
```

```
  uut: mux21_est PORT MAP (
        a => a,
        b => b,
        s => s,
        c => c
       );

 a_proc: process
variable temp_a:std_logic:='0';
  begin
    a<= temp_a;
     wait for 1 us;
    temp_a:=not temp_a;
  end process;

 b_proc: process
variable temp_b:std_logic:='0';
  begin
    b<= temp_b;
     wait for 4 us;
    temp_b:=not temp_b;
  end process;

 s_proc: process
variable temp_s:std_logic:='0';
  begin
    s<= temp_s;
     wait for 20 us;
    temp_s:=not temp_s;
  end process;
END;


--------------------------------------------------------------------------------
-- sumador de 4 bits, tiene carry in y carry out
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sumador_main is
    Port ( a : in  STD_LOGIC_VECTOR (3 downto 0);
           b : in  STD_LOGIC_VECTOR (3 downto 0);
           c : out  STD_LOGIC_VECTOR (3 downto 0);
           ci : in  STD_LOGIC;
           co : out  STD_LOGIC);
end sumador_main;

architecture Behavioral of sumador_main is
COMPONENT fulladder
              PORT(       a : IN std_logic; b : IN std_logic;
                          ci : IN std_logic;c : OUT std_logic;
                          co : OUT std_logic);
              END COMPONENT;
signal c0,c1,c2:std_logic;
begin
FA0: fulladder PORT MAP(a => a(0) ,b =>b(0) ,c =>c(0) ,ci =>ci ,co =>c0);
FA1: fulladder PORT MAP(a => a(1) ,b =>b(1) ,c =>c(1) ,ci =>c0 ,co =>c1);
FA2: fulladder PORT MAP(a => a(2) ,b =>b(2) ,c =>c(2) ,ci =>c1 ,co =>c2);
FA3: fulladder PORT MAP(a => a(3) ,b =>b(3) ,c =>c(3) ,ci =>c2 ,co =>co);
end Behavioral;

--------------------------------------------------------------------------------
-- Fulladder
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity fulladder is
    Port ( a : in  STD_LOGIC;  b : in  STD_LOGIC; c : out  STD_LOGIC;
           ci : in  STD_LOGIC; co : out  STD_LOGIC);
end fulladder;

architecture Behavioral of fulladder is
begin
c <= (a xor b) xor ci;
co <= ((a xor b)andci)or (a and b);
end Behavioral;

--------------------------------------------------------------------------------
--sumador de 8 bits con generate, tiene carry in y carry out
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity main is
    Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
           b : in  STD_LOGIC_VECTOR (7 downto 0);
           c : out  STD_LOGIC_VECTOR (7 downto 0);
           ci : in  STD_LOGIC;
           co : out  STD_LOGIC);
end main;

architecture Behavioral of main is
              COMPONENT fulladder
              PORT(
                          x : IN std_logic; y : IN std_logic;
                          ci : IN std_logic;z : OUT std_logic;
                          co : OUT std_logic);
              END COMPONENT;
signal cy: std_logic_vector          (7 downto 0);

begin
generar: for i in 1 to 7 generate
              suma: fulladder port map (a(i),b(i),cy(i-1),c(i),cy(i));
end generate;
suma0: fulladder port map( a(0),b(0),ci,c(0),cy(0));
co<=cy(7);
end Behavioral;

--------------------------------------------------------------------------------
-- full adder
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity fulladder is
    Port ( x : in  STD_LOGIC;  y : in  STD_LOGIC;  z : out  STD_LOGIC;
           ci : in  STD_LOGIC; co : out  STD_LOGIC);
end fulladder;

architecture Behavioral of fulladder is
begin
z <= (x xor y) xor ci;
co <= ((x xor y)and ci)or (x and y);
end Behavioral;

--------------------------------------------------------------------------------
-- sumador con generate versión 2 , tiene carry in y carry out
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity main is
    Port ( a : in  STD_LOGIC_VECTOR (7 downto 0);
```

```
        b : in  STD_LOGIC_VECTOR (7 downto 0);
        c : out  STD_LOGIC_VECTOR (7 downto 0);
        ci : in  STD_LOGIC;
        co : out  STD_LOGIC);
end main;

architecture Behavioral of main is
            COMPONENT fulladder
            PORT(
                        x : IN std_logic; y : IN std_logic; ci : IN std_logic;
                        z : OUT std_logic; co : OUT std_logic);
            END COMPONENT;
signal cy: std_logic_vector         (6 downto 0);
begin
generar: for i in 0 to 7 generate
un:         if i=0 generate
                        suma0: fulladder port map (a(i),b(i),ci,c(i),cy(i));
                        end generate;
dos:        if (i>0) and (i<7) generate
                        suma: fulladder port map (a(i),b(i),cy(i-1),c(i),cy(i));
                        end generate;
tres:       if (i=7) generate
                        suman: fulladder port map (a(i),b(i),cy(i-1),c(i),co);

                        end generate;
            end generate;

end Behavioral;

---------------------------------------------
-- prueba del sumador de 8 bits
---------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use IEEE.std_logic_arith.all;

ENTITY prueba IS
END prueba;
 ARCHITECTURE behavior OF prueba IS
    COMPONENT sumador_main
   PORT(
       a : IN  std_logic_vector(3 downto 0);
       b : IN  std_logic_vector(3 downto 0);
       c : OUT  std_logic_vector(3 downto 0);
       ci : IN  std_logic;
       co : OUT  std_logic
      );
   END COMPONENT;


  signal a : std_logic_vector(3 downto 0) := (others => '0');
  signal b : std_logic_vector(3 downto 0) := (others => '0');
  signal ci : std_logic := '0';
  signal c : std_logic_vector(3 downto 0);
  signal co : std_logic;

BEGIN

            -- Instantiate the Unit Under Test (UUT)
  uut: sumador_main PORT MAP (
        a => a,
        b => b,
        c => c,
        ci => ci,
        co => co
       );

  stim_proc: process
  begin
    -- hold reset state for 100 ns.
    wait for 100 ns;
                ci <='0';
                a<= conv_std_logic_vector(3,4);
                b<=conv_std_logic_vector(4,4);
                        wait for 1 us;
                a<= conv_std_logic_vector(2,4);
                b<=conv_std_logic_vector(15,4);

    wait;
  end process;

END;
```

CONTADORES Y MAQUINAS SECUENCIALES (FSM)

```
----------------------------------------------------------------------------------
-- LED_ 1Hz  Contador de 8 bits, se muestra en los LED
-- usa el reloj de 50 MHz que se dispone en la tarjeta
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity conta03 is
    Port ( CLK_50MHZ : IN STD_LOGIC;
           LED: out std_logic_vector(7 downto 0) );
end conta03;

architecture Behavioral of conta03 is
signal main_counter: natural range 0 to 25000000 := 0;
signal led_aux : std_logic_vector(7 downto 0):= "00000000";
begin

main_process: process(CLK_50MHZ)
begin
if rising_edge(CLK_50MHZ) then
            if main_counter = 0 then
                        main_counter <= 25000000;
                         led_aux <= led_aux+1;
            else
                        main_counter <= main_counter - 1;
            end if;
end if;
end process;
LED <= led_aux;
end Behavioral;

----------------------------------------------------------------------------------
-- ARCHIVO UCF
----------------------------------------------------------------------------------
net "clk_50mhz" loc = "c9" | iostandard = lvcmos33 ;
net "clk_50mhz" period = 20.0ns high 8.0 ns;
net "led<7>" loc = "f9" | iostandard = lvttl | slew = slow | drive = 8 ;
net "led<6>" loc = "e9" | iostandard = lvttl | slew = slow | drive = 8 ;
net "led<5>" loc = "d11" | iostandard = lvttl | slew = slow | drive = 8 ;
net "led<4>" loc = "c11" | iostandard = lvttl | slew = slow | drive = 8 ;
net "led<3>" loc = "f11" | iostandard = lvttl | slew = slow | drive = 8 ;
net "led<2>" loc = "e11" | iostandard = lvttl | slew = slow | drive = 8 ;
net "led<1>" loc = "e12" | iostandard = lvttl | slew = slow | drive = 8 ;
net "led<0>" loc = "f12" | iostandard = lvttl | slew = slow | drive = 8 ;


----------------------------------------------------------------------------------
-- CONTADOR 4 BITS con flip flop, el reloj se toma de los 50 MHz
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity conta04 is
    Port ( CLK_50MHZ : in  STD_LOGIC;
           borrar : in  STD_LOGIC;
           q3 : out  STD_LOGIC;
           q2 : out  STD_LOGIC;
           q1 : out  STD_LOGIC;
           q0 : out  STD_LOGIC);
end conta04;

architecture Behavioral of conta04 is
signal t0: std_logic := '0';
signal t1: std_logic := '0';
signal t2: std_logic := '0';
signal t3: std_logic := '0';
signal reloj: std_logic := '0';

signal contador: natural range 0 to 25000000 := 0;
signal reloj_aux : std_logic := '0';

begin

ft0:
process (reloj,borrar)
begin
  if borrar = '1' then
    t0<='0';
  elsif (reloj'event and reloj = '0') then
    t0<=not t0;
  end if;
end process;
q0<=t0;

ft1:
process (t0,borrar)
begin
  if borrar = '1' then
    t1<='0';
  elsif (t0'event and t0 = '0') then
    t1<=not t1;
  end if;
end process;
q1<=t1;

ft2:
process (t1,borrar)
begin
  if borrar = '1' then
    t2<='0';
  elsif (t1'event and t1 = '0') then
    t2<=not t2;
  end if;
end process;
q2<=t2;

ft3:
process (t2,borrar)
begin
  if borrar = '1' then
    t3<='0';
  elsif (t2'event and t2 = '0') then
    t3<=not t3;
  end if;
end process;
q3<=t3;

reloj_process: process(CLK_50MHZ)
begin
if rising_edge(CLK_50MHZ) then
            if contador = 0 then
                        contador <= 25000000;
                        reloj_aux <= not reloj_aux;
            else
                        contador <= contador - 1;
            end if;
end if;
end process;
reloj <= reloj_aux;

end Behavioral;
```

```
----------------------------------------------------------------------------------
-- ARCHIVO UCF
----------------------------------------------------------------------------------
net "clk_50mhz" loc = "c9" | iostandard = lvcmos33 ;
net "clk_50mhz" period = 20.0ns high 40%;
net "borrar" loc = "v16" | iostandard = lvttl | pulldown ;
net "q3" loc = "f11" | iostandard = lvttl | slew = slow | drive = 8 ;
net "q2" loc = "e11" | iostandard = lvttl | slew = slow | drive = 8 ;
net "q1" loc = "e12" | iostandard = lvttl | slew = slow | drive = 8 ;
net "q0" loc = "f12" | iostandard = lvttl | slew = slow | drive = 8 ;
----------------------------------------------------------------------------------
-- FSM simple, el  1 HZ se deriva de los 50MHZ, como módulo aparte
-- codificación de estados binaria
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity fsm is
    Port ( ini_fin : in  STD_LOGIC;
           direccion : in  STD_LOGIC;
           LED : out  STD_LOGIC_VECTOR (3 downto 0);
           clk50mhz : in  STD_LOGIC);
end fsm;

architecture Behavioral of fsm is

COMPONENT reloj_1hz
        PORT(
                clk_50MHZ : IN std_logic;
                clk_1hz : OUT std_logic
                );
        END COMPONENT;

signal clk_1hz:std_logic;

begin

Inst_reloj_1hz: reloj_1hz PORT MAP(
                clk_50MHZ =>clk50mhz ,
                clk_1hz => clk_1hz
        );

maq_estados:process(clk_1hz)
        variable actual: bit_vector(1 downto 0):="00";
begin
        if ini_fin='0' and rising_edge(clk_1hz) then
                if direccion='1' then
                case (actual) is
                        when "00" =>
                                actual:="01" ;
                                led<="0001";
                        when "01" =>
                                actual:="10";
                                led<="1000";

                        when "10" =>
                                actual:="11";
                                led<="0100";

                        when  "11" =>
                                actual:="00";
                                led<="0010";

                end case;
                else
                case (actual) is
                        when "00" =>
                                actual:="11" ;
                                led<="0001";
                        when "11" =>
                                actual:="10";
                                led<="0010";

                        when "10" =>
                                actual:="01";
                                led<="0100";

                        when "01" =>
                                actual:="00";
                                led<="1000";

                end case;
                end if;
        end if;
end process;
end Behavioral;

----------------------------------------------------------------
--Modulo extra para generar 1Hz
----------------------------------------------------------------------------------


----------------------------------------------------------------------------------
--reloj_ 1Hz
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity reloj_1hz is
    Port ( clk_50MHZ : in  STD_LOGIC;
           clk_1hz : out  STD_LOGIC);
end reloj_1hz;

architecture Behavioral of reloj_1hz is
begin

reloj_process: process(CLK_50MHZ)
variable contador:natural range 0 to 25000000 := 0;
variable reloj_aux:std_logic:='0';
begin
if rising_edge(CLK_50MHZ) then
        if contador = 0 then
                contador := 25000000;
                reloj_aux := not reloj_aux;
        else
                contador := contador - 1;
        end if;
end if;
clk_1hz <= reloj_aux;
end process;

end Behavioral;

----------------------------------------------------------------------------------
-- ARCHIVO UCF
----------------------------------------------------------------------------------

NET "LED<0>" LOC = "F12" | IOSTANDARD = LVTTL | SLEW =
SLOW | DRIVE = 8 ;
NET "LED<1>" LOC = "E12" | IOSTANDARD = LVTTL | SLEW =
SLOW | DRIVE = 8 ;
NET "LED<2>" LOC = "E11" | IOSTANDARD = LVTTL | SLEW =
SLOW | DRIVE = 8 ;
NET "LED<3>" LOC = "F11" | IOSTANDARD = LVTTL | SLEW =
SLOW | DRIVE = 8 ;
NET "ini_fin" LOC = "L13" | IOSTANDARD = LVTTL | PULLUP ;
NET "direccion" LOC = "L14" | IOSTANDARD = LVTTL | PULLUP ;

NET "CLK50mhz" LOC = "C9" | IOSTANDARD = LVCMOS33 ;
# Define clock period for 50 MHz oscillator (40%/60% duty-cycle)
NET "CLK50mhz" PERIOD = 20.0ns HIGH 40%;
```

```
--------------------------------------------------------------------------------
-- FSM codificacion binaria, máquina de Moore
-- las salidas se generan en un bloque aparte
-- reloj de un Hz derivado en un módulo aparte
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity fsm is
    Port ( ini_fin : in  STD_LOGIC;
           direccion : in  STD_LOGIC;
           LED : out  STD_LOGIC_VECTOR (3 downto 0);
           clk50mhz : in  STD_LOGIC);
end fsm;

architecture Behavioral of fsm is

COMPONENT reloj_1hz
        PORT(
                clk_50MHZ : IN std_logic;
                clk_1hz : OUT std_logic
                );
        END COMPONENT;

signal clk_1hz:std_logic;
signal actual: bit_vector(1 downto 0):="00";
begin

Inst_reloj_1hz: reloj_1hz PORT MAP(
                clk_50MHZ =>clk50mhz ,
                clk_1hz => clk_1hz
        );
maq_estados:process(clk_1hz)

begin
        if ini_fin='0' and rising_edge(clk_1hz) then
                if direccion='1' then
                case (actual) is
                        when "00" =>
                                actual<="01" ;
                        when "01" =>
                                actual<="10";
                        when "10" =>
                                actual<="11";
                        when  "11" =>
                                actual<="00";
                end case;
                else
                case (actual) is
                        when "00" =>
                                actual<="11" ;
                        when "11" =>
                                actual<="10";
                        when "10" =>
                                actual<="01";
                        when "01" =>
                                actual<="00";
                end case;
                end if;
        end if;
end process;



salida: with actual select
     led <= "0001" when "00",
        "1000" when "01",
        "0100" when "10",
        "0010" when "11";

end Behavioral;

--------------------------------------------------------------------------------
-- ARCHIVO UCF
--------------------------------------------------------------------------------
Archivo UCF y el módulo de reloj igual que el ejemplo anterior




--------------------------------------------------------------------------------
-- FSM codificacion ONE-HOT, reloj generado en módulo aparte
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity fsm is
    Port ( ini_fin : in  STD_LOGIC;
           direccion : in  STD_LOGIC;
           LED : out  STD_LOGIC_VECTOR (3 downto 0);
           clk50mhz : in  STD_LOGIC);
end fsm;

architecture Behavioral of fsm is

COMPONENT reloj_1hz
        PORT(   clk_50MHZ : IN std_logic;
                clk_1hz : OUT std_logic);
        END COMPONENT;

signal clk_1hz:std_logic;
signal actual: std_logic_vector(3 downto 0):="0001";
begin

Inst_reloj_1hz: reloj_1hz PORT MAP(
                clk_50MHZ =>clk50mhz ,
                clk_1hz => clk_1hz );

maq_estados:process(clk_1hz)

begin
        if ini_fin='0' and rising_edge(clk_1hz) then
                if direccion='1' then
                case (actual) is
                        when "0001" =>
                                actual<="0010" ;
                        when "0010" =>
                                actual<="0100";
                        when "0100" =>
                                actual<="1000";
                        when  "1000" =>
                                actual<="0001";
                        when others =>
                                actual<="0001";
                end case;
                else
                case (actual) is
                        when "0001" =>
                                actual<="1000" ;
                        when "1000" =>
                                actual<="0100";
                        when "0100" =>
                                actual<="0010";
                        when "0010" =>
                                actual<="0001";
                        when others =>
                                actual<="0001";
```

```
                        end case;
                        end if;
                end if;
end process;

salida: led <= actual;

end Behavioral;
```

```
--------------------------------------------------------------------------------
-- ARCHIVO UCF
--------------------------------------------------------------------------------
Archivo UCF y el módulo de reloj igual que el ejemplo anterior
```

```
--------------------------------------------------------------------------------
-- FSM con estados enumerados
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity fsm is
    Port ( ini_fin : in  STD_LOGIC;
           direccion : in  STD_LOGIC;
           LED : out  STD_LOGIC_VECTOR (3 downto 0);
           clk50mhz : in  STD_LOGIC);
end fsm;

architecture Behavioral of fsm is

COMPONENT reloj_1hz
        PORT(
                clk_50MHZ : IN std_logic;
                clk_1hz : OUT std_logic
                );
        END COMPONENT;

signal clk_1hz:std_logic;

type estados is (st0, st1, st2, st3);
signal actual: estados;

begin

Inst_reloj_1hz: reloj_1hz PORT MAP(
                clk_50MHZ =>clk50mhz ,
                clk_1hz => clk_1hz
        );

maq_estados:process(clk_1hz)

begin
        if ini_fin='0' and rising_edge(clk_1hz) then
                if direccion='1' then
                case actual is
                        when st0 =>
                                actual<=st1 ;
                        when st1 =>
                                actual<=st2;
                        when st2 =>
                                actual<=st3;
                        when  st3 =>
                                actual<=st0;
                        when others =>
                                actual<=st0;
                end case;
                else
                case (actual) is
                        when st0 =>
                                actual<=st3 ;
                        when st3 =>
                                actual<=st2;
                        when st2 =>
                                actual<=st1;
                        when st1 =>
                                actual<=st0;
                        when others =>
                                actual<=st0;
                end case;
                end if;
        end if;
end process;

salidas: with actual select
    led <= "0001" when st0,
           "0010" when st1,
           "0100" when st2,
           "1000" when st3,
           "0000" when others;
end Behavioral;
```

```
--------------------------------------------------------------------------------
-- ARCHIVO UCF
--------------------------------------------------------------------------------
Archivo UCF y el módulo de reloj igual que el ejemplo anterior
```

MANEJO DE LA PERILLA

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity main is
    Port ( clk_50mhz: in std_logic;
           rot_a : in  STD_LOGIC;
           rot_b : in  STD_LOGIC;
           rot_center : in  STD_LOGIC;
           led : out  STD_LOGIC_VECTOR (7 downto 0));
end main;

architecture Behavioral of main is
signal led_tempo:std_logic_vector(7 downto 0):="00000000";
signal reloj:std_logic:='0';
signal temp_a,temp_b:std_logic;
begin

reloj_process: process(clk_50mhz)
variable reloj_tempo: natural range 0 to 5000:=5000;
begin
        if clk_50mhz'event and  clk_50mhz='1' then
                if reloj_tempo=0 then
                        reloj_tempo := 5000;
                        reloj <= not reloj;
                else
                        reloj_tempo:=reloj_tempo-1;
                end if;
        end if;
end process;

filtro: process(reloj)
begin
        if reloj'event and reloj='1' then
                temp_a<=rot_a;
                temp_b<=rot_b;
        end if;
end process;

proceso1:process(temp_a,rot_center)

begin
        if rot_center='1' then
                led_tempo<="00000000";
        elsif temp_a'event and temp_a='1' then
                if temp_b = '1' then
                        led_tempo <= led_tempo + 1;
                else
                        led_tempo <= led_tempo - 1;
                end if;
        end if;
end process;
led<=led_tempo;

end Behavioral;
--------------------------------------------------------------------------------
```

```
-- CONTADOR+MEMORIA
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity xx_main is
    Port ( CLK_50MHZ : in  STD_LOGIC;
           PARAR : in  STD_LOGIC;
           LED: out  STD_LOGIC_vector (7 downto 0));
end xx_main;

architecture Behavioral of xx_main is

signal reloj_uno: std_logic;
signal direccion: std_logic_vector(3 downto 0);
signal datos: std_logic_vector(7 downto 0);

component xx_memoria
            port (
            a: in std_logic_vector(3 downto 0);
            spo: out std_logic_vector(7 downto 0));
end component;

-- Synplicity black box declaration
attribute syn_black_box : boolean;
attribute syn_black_box of xx_memoria: component is true;


begin

  inst_memoria: xx_memoria          port map (
                                    a => direccion,
                                    spo => datos);

process (CLK_50MHZ)
variable contador: natural range 0 to 25000000 := 0;
variable reloj_aux: std_logic;

begin
if rising_edge(CLK_50MHZ) then
        if contador=0 then
                contador:=2500000;
                reloj_aux:=not reloj_aux;
        else
                contador:=contador-1;
        end if;
end if;
reloj_uno<=reloj_aux;
end process;

process (reloj_uno, parar)
begin
        if rising_edge(reloj_uno) and parar='0' then
                direccion<=direccion+1;
                end if;
end process;
LED<=datos;

end Behavioral;
```

```
; Archivo de inicalizacion de una RAM
; 8-bit ancho por 16 localidades RAM
memory_initialization_radix = 2;
memory_initialization_vector =
11000000
11100000
01110000
00111000
00011100
00001110
00000111
00000011
00000001
00000011
00000111
00001110
00011100
00111000
01110000
11100000;
```