

7. Modelos de comportamiento

Describen el funcionamiento del sistema al que representan sin hacer referencia al detalle de subelementos e interconexiones que caracterizan a los modelos estructurales. Si se considera el nivel de abstracción con que se puede describir el comportamiento de sistemas se distinguen dos estilos :

- 1 - Descripción a *nivel de transferencias entre registros* - RTL - o de *flujo de datos* .
- 2 - Descripciones algorítmicas, sin referencia explícita al sistema físico.

Sin embargo, los modelos en sí carecen de sentido: cualquier modelo se desarrolla con vistas a una determinada aplicación, siendo la síntesis aquella aplicación que exige mayor esfuerzo para el desarrollo de modelos, por las restricciones y matices impuestos por las herramientas de síntesis. En los apartados que siguen se indican las características de cada uno de estos estilos y se presentan ejemplos de modelos de varios primitivos de uso frecuente, desarrollados en alguno de los niveles de abstracción citados y que, siguiendo los pasos descritos anteriormente en modelos estructurales, pueden ser declarados como componentes para su utilización en una configuración o estructura de nivel más alto.

7.1 DESCRIPCIONES A NIVEL DE FLUJO DE DATOS

En este estilo se detallan las transferencias de datos a nivel de bloques - *registros* - lo que implica un diseño previo a nivel de particionado del sistema en subelementos , así como del control de las transferencias de datos. Esto supone que en estas descripciones se utilizarán fundamentalmente sentencias concurrentes para asignaciones de señales de tipos condicionales, selectivas y guardadas, así como estructuración de la descripción en bloques cuyo comportamiento esté directamente asociado a este tipo de sentencias, por lo que estas descripciones se caracterizan por :

- Declaraciones de señales que corresponden a movimientos reales de datos y señales de control de transferencias de datos.
- Identificación con un esquema a nivel de bloques en donde se distinguen unidades funcionales clásicas : decodificadores, multiplexores, registros, etc.
- Utilización de tipos con relación directa al hardware, fundamentalmente los tipos BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, o similares definidos por el usuario.
- Organización típica en bloques de datos o proceso y unidades de control.

Este tipo de descripciones corresponde a un nivel de abstracción más alto que aquel de las descripciones estructurales a nivel de puertas o componentes, pero sigue precisando una labor de análisis -diseño *top-down*- para descomponer el sistema y controlar las interacciones entre los subelementos.

Algunos ejemplos de este tipo de modelos son los siguientes :

1- Latch sencillo

```
ENTITY latch IS
    PORT ( dato , reloj : IN BIT ; salida : OUT BIT );
END latch;
ARCHITECTURE guardada OF latch IS
BEGIN
    bloque : BLOCK (reloj = '1')
        BEGIN
            salida <= GUARDED dato AFTER 10 ns;
        END BLOCK bloque;
END guardada;
```

2- Registro con carga en paralelo

```
ENTITY registro IS
    PORT ( reloj , carga, clear : IN BIT ;
          entrada : IN BIT_VECTOR ( 7 DOWNT0 0 );
          salida : OUT BIT_VECTOR ( 7 DOWNT0 0 ) );
END registro;
ARCHITECTURE guardada OF registro IS
BEGIN
    bloque : BLOCK ( NOT reloj'STABLE AND reloj = '1' )
        BEGIN
            salida <= GUARDED X"00" AFTER 10 ns WHEN clear = '1'
                ELSE entrada AFTER 15 ns WHEN carga = '1'
                ELSE salida ;
        END BLOCK bloque;
END guardada;
```

3- Registro de desplazamiento con carga paralela y entradas serie

```
ENTITY regdesp IS
    PORT ( reloj , carga, clear, desder, desizq, entder, entizq : IN BIT ;
          entrada : IN BIT_VECTOR ( 7 DOWNT0 0 );
          salida : OUT BIT_VECTOR ( 7 DOWNT0 0 ) );
END regdesp;
ARCHITECTURE guardada OF regdesp IS
    SIGNAL sel : BIT_VECTOR ( 0 TO 1 );
BEGIN
    sel <= desder & desizq;
    bloque : BLOCK ( NOT reloj'STABLE AND reloj = '1' )
        BEGIN
            salida <= GUARDED X"00" AFTER 10 ns WHEN clear = '1'
                ELSE entrada AFTER 15 ns WHEN carga = '1'
                ELSE entrada( 6 DOWNT0 0 ) & entizq WHEN sel = " 01"
                ELSE entder & entrada ( 7 DOWNT0 1 ) WHEN sel = " 10"
                ELSE salida;
        END BLOCK bloque;
END guardada;
```

4- Sumador serie binario

```

ENTITY sumabin IS
  PORT ( a, b : IN BIT_VECTOR ( 7 DOWNT0 0 );
        sum : OUT BIT_VECTOR ( 7 DOWNT0 0 );
        carry : OUT BIT );
END sumabin;
ARCHITECTURE funcional OF sumabin IS
  SIGNAL carry_vector : BIT_VECTOR ( 8 DOWNT0 0 );
  SIGNAL temp : BIT_VECTOR ( 7 DOWNT0 0 );
BEGIN
  temp    <= a XOR b;
  carry_vector <= ((A AND B) OR ( temp AND carry_vector( 7 DOWNT0 0 ))) & '0' ;
  sum     <= temp XOR carry_vector( 7 DOWNT0 0 );
  carry   <= carry_vector(8);
END funcional;

```

5- Comparador de bits del apartado 6.1

Las ecuaciones booleanas que describen el comparador de un bit son:

$$\begin{aligned}
 amb &= am + b'm + ab' \\
 bma &= b n + a' n + ba' \\
 igl &= abi + a'b'i
 \end{aligned}$$

que expresadas como funciones y almacenadas en el paquete de “modelos” como:

```

FUNCTION fmn(x,y,z: BIT) RETURN BIT IS
BEGIN
  RETURN ( x AND z) OR (NOT y AND z) OR (x AND NOT y) ;
END fmn;

FUNCTION figl(x,y,z: BIT) RETURN BIT IS
BEGIN
  RETURN ( x AND y AND z) OR (NOTx AND NOT y AND z);
END figl;

```

permiten describir el comparador "compbit" con la arquitectura siguiente :

```

USE WORK. modelos.ALL ;
ARCHITECTURE funcional OF compbit IS
BEGIN
  amb <= fmn ( a,b,m ) ;
  bma <= fmn ( b,a,n ) ;
  igl <= figl ( a,b,i ) ;
END funcional;

```

Si comparamos esta descripción con la del apartado 6.1 resulta evidente que usando subprogramas, es decir, estableciendo modularidad en los diseños, los modelos de VHDL, aparte de tener una mejor estructura, serán más reducidos y más fácilmente interpretables.

7.2 DESCRIPCIONES ALGORÍTMICAS

El estilo algorítmico considera al sistema como una *caja negra* cuya descripción se centra fundamentalmente en relacionar entradas y salidas, sin entrar en el detalle de cómo se obtienen unas a partir de otras. Mientras el estilo flujo de datos indica *cómo opera* el sistema, el estilo algorítmico describe *qué hace* el sistema, por lo que este tipo de descripciones se encuentran organizadas en PROCESOS por su capacidad de uso de sentencias secuenciales y estructuras algorítmicas del tipo IF-THEN-ELSE, etc. A diferencia de los otros estilos de descripción, el planteamiento previo al desarrollo del modelo consiste en :

- Asociar las especificaciones del sistema a procesos VHDL .
- Dividir éstos en actividades u operaciones, es decir, determinar que señales deben obtenerse como resultado de la actividad del proceso y determinar que señales lo controlan o disparan.
- Desarrollar el modelo con las sentencias más apropiadas para realizar el control de las operaciones o actividades del proceso.

Todo lo anterior equivale a particionar el sistema en módulos operativos cuya acción viene determinada por la actividad de unos procesos que controlan a otros.

Si en los estilos de descripción estructural o de flujo de datos es fácil asociar *hardware* al modelo, en el estilo algorítmico es típico acudir a representaciones por medio de diagramas de flujo o grafos, tablas de estados, etc. lo que supone un nivel de abstracción por encima del utilizado en los otros estilos de modelado de sistemas.

7.3 DESCRIPCIONES ALGORÍTMICAS PARA SÍNTESIS

Si bien no existe un estilo que sea óptimo para todas las aplicaciones donde se precise una descripción de *hardware*, y aunque el VHDL pueda utilizarse para describir sistemas con diferentes niveles de abstracción, la existencia de los dos estilos descritos anteriormente está en parte relacionada al control que se desea tener sobre la operación del sistema, las restricciones que se desea imponer a su diseño concreto y, también, a la potencia o capacidad de las herramientas CAD para síntesis.

Por síntesis se entiende en este contexto la capacidad de transformar automáticamente una descripción desde un nivel de abstracción a otro de nivel inferior. Esta capacidad, hasta tiempo muy reciente, ha estado limitada a transformaciones desde el nivel RTL hasta el de puertas lógicas, lo que forzaba al empleo de descripciones a nivel flujo de datos cuando el destino final del modelo era un proceso de síntesis lógica.

Aunque con ciertas limitaciones que decrecen progresivamente, se dispone ya de potentes herramientas con capacidad de *síntesis algorítmica*, que permiten transformar automáticamente modelos con descripciones algorítmicas a descripciones RTL, a partir de las que se realiza la transformación a nivel de puertas. Esto es posible porque los *compiladores de comportamiento*, además de aceptar la descripción explícita de registros, pueden *inferir* la existencia de otros a partir de las sentencias que definen comportamiento.

Esta posibilidad de *síntesis algorítmica* es de uso muy reciente y, por tanto, se puede

decir que no solo depende de la herramienta, sino también de su versión, condicionando el desarrollo de modelos al soporte ofrecido por la herramienta y, más particularmente, a su estilo de *inferencias*, es decir, la forma en que diseña *hardware* a partir del código VHDL.

Las ventajas inherentes a la síntesis algorítmica son importantes:

- Reducción de los tiempos para diseñar e introducir nuevos sistemas en el mercado, debido a una automatización desde niveles de abstracción altos.
- Reducción de los costes de diseño, ya que en gran medida estos están relacionados directamente a los tiempos de diseño.
- Reducción de los errores de diseño, especialmente cuando el tamaño de los proyectos o su complejidad crece, ya que con ellos crece la probabilidad de que los diseñadores cometan errores, así como la dificultad de depurar los diseños, con incidencia en el tiempo y coste del diseño.
- Posibilidad de obtener varios diseños o alternativas para un juego de especificaciones. Esto permite configurar o elegir la más conveniente después de simular, así como fijar estrategias de gamas o familias de productos o actualización - *upgrading* - sin que ello suponga rediseños básicos.
- Posibilidad de introducir cambios de especificaciones y alterar los diseños sin cambios importantes en la planificación o en los costes. En ausencia de estos sistemas de desarrollo, un cambio de especificaciones puede suponer el reinicio del proyecto.

Dado que una de las aplicaciones más importantes del VHDL es el diseño automático o síntesis y en el supuesto de que las ventajas explicadas arriba presumiblemente condicionarán no solo el estilo de diseño o modelado de sistemas, sino también las herramientas elegidas al efecto, se exponen algunos modelos algorítmicos desarrollados con esa orientación. Compárense con aquellos similares en estilo *flujo de datos*.

7.4 INFERENCIAS BÁSICAS DE SÍNTESIS

Dos de los factores de mérito que se consideran para valorar la eficiencia de un circuito son su tamaño y su velocidad. Por tanto, el diseño y el modelo deberán enfocarse a fin de reducir tamaño y maximizar velocidad, dentro de los márgenes que fijen especificaciones como el consumo de potencia, el coste, etc. En consecuencia, hay que modelar utilizando aquellas sentencias que infieran el hardware deseado, pero no más, lo que supone utilizar apropiadamente las sentencias a partir de las que se infieren los elementos básicos, es decir, puertas o registros. Registrar señales innecesariamente, en general, implica componentes innecesarios -tamaño- y retardos -velocidad-, por lo que interesa saber que tipo de sentencias infieren registros, ya sean sensibles a nivel - *latches* - o disparados por flancos, - *flip-flops* -, y evitar inferencias innecesarias de registros.

Sentencias empleadas normalmente para detectar flancos de subida de señales son:

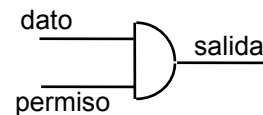
```
WAIT UNTIL      onda' EVENT AND onda = '1' ;  
WAIT UNTIL NOT onda'STABLE AND onda = '1' ;  
IF ( onda'EVENT AND onda = '1' ) THEN
```

todas secuenciales y, por tanto, incluidas en Procesos, y de las que se inferirán *flip-flops* en el caso de `WAIT` y *flip-flops* o *latches* según se construya la sentencia `IF`. La elección de un tipo de sentencia u otro puede ser una recomendación de la herramienta de síntesis, pero, normalmente, la sentencia `IF` permite mejor control de las inferencias y admite más posibilidades. También, consecuentemente, exige más cuidado en la descripción. En general, las reglas básicas a aplicar son :

- Los procesos síncronos, que computan valores solamente en los instantes de flancos, deben ser sensibles a la señal de reloj utilizada.
- Los procesos asíncronos, que computan valores con los flancos de reloj y cuando se cumplen las condiciones asíncronas, deben ser sensibles a la señal de reloj, si existe, y a las señales que afectan al comportamiento asíncrono. Véanse los ejemplos

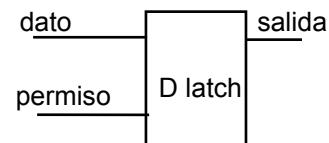
1- La descripción completa de la sentencia `IF` infiere una puerta

```
IF ( permiso = '1' ) THEN
    salida <= dato;
ELSE
    salida <= '0';
END IF;
```



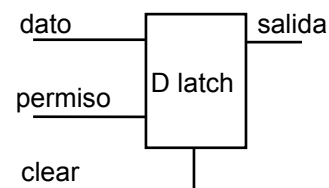
2- Se infieren *latches* al utilizar sentencias condicionales incompletas:

```
PROCESS ( dato, permiso )
BEGIN
    IF ( permiso = '1' ) THEN
        salida <= dato;
    END IF;    -- no incluye cláusula ELSE
END PROCESS;
```



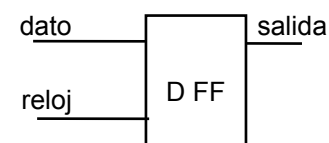
3- El modelo anterior se puede ampliar con un *clear* asíncrono

```
PROCESS ( dato, permiso, clear )
BEGIN
    IF ( clear = '1' ) THEN
        salida <= '0';
    ELSIF ( permiso = '1' ) THEN
        salida <= dato;
        -- no incluye cláusula ELSE
    END IF;
END PROCESS;
```



4- Se infieren *flip-flops* a partir de sentencias de detección de flancos

```
PROCESS ( dato, reloj )
BEGIN
    IF ( reloj'EVENT AND reloj = '1' ) THEN
        salida <= dato;
    END IF;
END PROCESS;
```

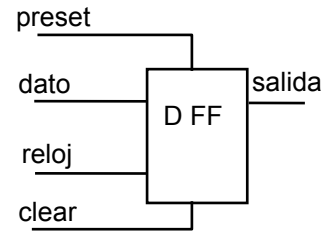


5- Pueden incorporarse controles asíncronos en la inferencia

```

PROCESS ( dato, reloj , preset, clear )
BEGIN
  IF ( preset = '1' ) THEN
    salida <= '1';
  ELSIF ( clear = '1' ) THEN
    salida <= '0' ;
  ELSIF ( reloj'EVENT AND reloj = '1' ) THEN
    salida <= dato;
  END IF;
END PROCESS;

```

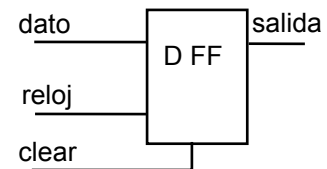


6- Es posible inferir controles síncronos, alterando el orden de cláusulas IF

```

PROCESS ( reloj , clear )
BEGIN
  IF ( reloj'EVENT AND reloj = '1' ) THEN
    IF clear = '1' THEN
      salida <= '0' ;
    ELSE
      salida <= dato;
    END IF;
  END IF;
END PROCESS;

```



A partir de las anteriores inferencias pueden modelarse primitivos de nivel superior, como por ejemplo, un contador con control de cuenta ascendente/descendente :

```

ENTITY contador IS
  PORT( reset, enable, up, clk :IN BIT; salida : INOUT BIT_VECTOR( 7 DOWNT0 0 ) );
END contador;

USE WORK.modelos.ALL;
ARCHITECTURE contbyte OF contador IS
  BEGIN
    cont : PROCESS( clk )
      BEGIN
        IF clk = '1' THEN
          IF reset = '1' THEN
            salida <= X"00";
          ELSIF enable = '1' THEN
            IF up = '1' THEN
              salida <= incrementar(salida);
            ELSE
              salida <= decrementar(salida);
            END IF;
          END IF;
        END IF;
      END PROCESS cont;
    END contbyte;

```

-- para visualizar funciones internas
 -- de incrementar o decrementar
 -- detección de flanco sin WAIT
 -- reset síncrono
 -- incrementar cuenta
 -- llamada a función
 -- decrementar cuenta
 -- llamada a función

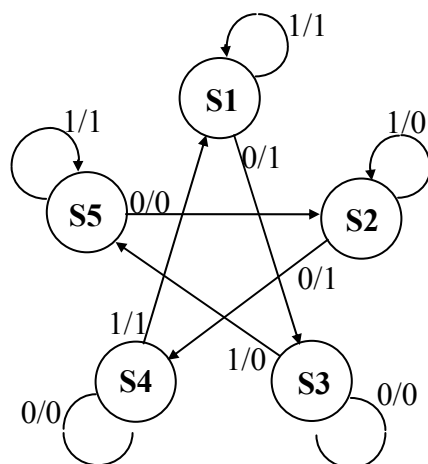
7.5 MÁQUINAS DE ESTADOS FINITOS

Con las ideas expuestas acerca de inferencias se puede avanzar en el desarrollo de modelos sintetizables. Conviene siempre distinguir qué señales se quiere que cambien de forma síncrona, con flancos de reloj, y cuales de forma asíncrona, o entre flancos de reloj. Para esto se debe separar la funcionalidad del circuito y asociarla a procesos síncronos, con WAIT o sentencias IF-THEN-ELSE, o asíncronos, en las que no se usa WAIT.

Un ejemplo interesante y de uso frecuente son las máquinas de estados finitos. Supuesto que el diseñador ha decidido si conviene un tipo Mealy o un Moore, el problema para modelar en VHDL se basa en un fácil análisis de la tabla de transiciones o el diagrama de estados, con referencias a las entradas y salidas relacionadas a cada estado.

7.5.1 MÁQUINA DE MEALY

Supongamos que el diagrama de estados y la tabla correspondiente son las siguientes:



ESTADO ACTUAL	ESTADO SIGUIENTE		SALIDA Z	
	X = 0	X = 1	X = 0	X = 1
S1	S3	S1	1	1
S2	S4	S2	1	0
S3	S5	S3	0	0
S4	S4	S1	0	1
S5	S2	S5	0	1

Dividimos el modelo en dos procesos, uno cambia la lógica combinacional, y otro la secuencial que provoca la transición entre estados.

```

ENTITY mealyfsm IS
    PORT ( X, CLK : IN BIT; Z : OUT BIT);
END mealyfsm;

ARCHITECTURE comport OF mealyfsm IS
    TYPE tipo_estado IS ( s1,s2,s3,s4, s5);
    SIGNAL estado_actual, estado_sigte : tipo_estado;
BEGIN
    combinacional : PROCESS ( estado_actual, X)
    BEGIN
        CASE estado_actual IS
    
```



```

        WHEN s1 =>    IF X = '0'    THEN      Z <= '1';
                                estado_sigte <= s3;
                                ELSE      Z <= '1';
                                estado_sigte <= s1;

        WHEN s2 =>    IF X = '0'    THEN      Z <= '1';
                                estado_sigte <= s4;
                                ELSE      Z <= '0';
                                estado_sigte <= s2;

        WHEN s3 =>    IF X = '0'    THEN      Z <= '0';
                                estado_sigte <= s5;
                                ELSE      Z <= '0';
                                estado_sigte <= s3;

        WHEN s4 =>    IF X = '0'    THEN      Z <= '0';
                                estado_sigte <= s4;
                                ELSE      Z <= '1';
                                estado_sigte <= s1;

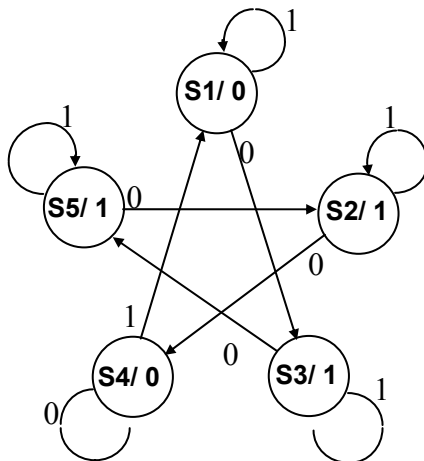
        WHEN s5 =>    IF X = '0'    THEN      Z <= '0';
                                estado_sigte <= s2;
                                ELSE      Z <= '1';
                                estado_sigte <= s5;

                                END IF;

        END CASE;
    END PROCESS;
conmutacion: PROCESS
    BEGIN
        WAIT UNTIL clk'EVENT AND clk = '1';
        estado_actual <= estado_sigte;
    END PROCESS;
END comport;

```

7.5.2 MÁQUINA DE MOORE



ESTADO ACTUAL	ESTADO SIGUIENTE		SALIDA
	X = 0	X = 1	
S1	S3	S1	0
S2	S4	S2	1
S3	S5	S3	1
S4	S4	S1	0
S5	S2	S5	1

El modelo correspondiente es muy similar al del ejemplo de Mealy

```
ENTITY moorefsm IS
PORT ( X, CLK : IN BIT; Z : OUT BIT);
END moorefsm;

ARCHITECTURE comport OF moorefsm IS
    TYPE tipo_estado IS ( s1,s2,s3,s4, s5);
    SIGNAL estado_actual, estado_sigte : tipo_estado;
BEGIN
    combinacional : PROCESS ( estado_actual, X)
    BEGIN
        CASE estado_actual IS
            WHEN s1 =>
                Z <= '0';
                IF X = '0' THEN estado_sigte <= s3;
                ELSE estado_sigte <= s1;
                END IF;
            WHEN s2 =>
                Z <= '1';
                IF X = '0' THEN estado_sigte <= s4;
                ELSE estado_sigte <= s2;
                END IF;
            WHEN s3 =>
                Z <= '1';
                IF X = '0' THEN estado_sigte <= s5;
                ELSE estado_sigte <= s3;
                END IF;
            WHEN s4 =>
                Z <= '0';
                IF X = '0' THEN estado_sigte <= s4;
                ELSE estado_sigte <= s1;
                END IF;
            WHEN s5 =>
                Z <= '1';
                IF X = '0' THEN estado_sigte <= s2;
                ELSE estado_sigte <= s5;
                END IF;
        END CASE;
    END PROCESS;

    conmutacion: PROCESS
    BEGIN
        WAIT UNTIL clk'EVENT AND clk = '1';
        estado_actual <= estado_sigte;
    END PROCESS;
END comport;
```

7.6 MODELO DE MEMORIA RAM ESTÁTICA

Por el amplio uso en sistemas digitales y por sus características como dispositivo con salidas bidireccionales y multinivel, las memorias RAM son dispositivos que se usan como componentes en muchos diseños con VHDL. Por esto, es conveniente tener algún modelo que pueda ser utilizado para configurarlo estructuralmente en otros modelos.

Para evitar el desarrollo de nuevos tipos y subprogramas, se utilizan tipos *std_logic* y alguna de las funciones de conversión de este empaquetamiento.

Por otra parte, dado que existen multitud de modelos y configuraciones de RAMs, definiremos con genéricos algunos de los parámetros más significativos, por ejemplo el número de palabras -memdim- está implícitamente definido por la dimensión del bus de direcciones -direc- relacionado al genérico `addrbits`, particularizado a 9 en el ejemplo, para una SRAM de 256KB.

La RAM se controla únicamente con las señales `ncs` -chip select negado- y la señal `nwe` -write enable negado- para control de escribir o leer.

El bus de datos se declara como un array de modo INOUT y tipo `std_logic_vector`, es decir, utiliza una señal de tipo resuelto, que se inicializa al valor enumerado más débil:

```
datos : INOUT STD_LOGIC_VECTOR(7 DOWNT0) := "ZZZZZZZZ";
```

Las señales de control son todas de tipo BIT. Así, la entidad de la RAM es :

```
USE IEEE.STD_LOGIC_1164.ALL;

USE WORK.modelos.ALL;      -----
                           ---- RAM cuadrada => addrbits = 50% de address ----
                           -----
ENTITY stat_ram IS
    GENERIC(addrbits : POSITIVE := 9;      -- 50 % de address bits
            wrd       : TIME      := 3 ns;  -- write delay
            rdd       : TIME      := 4 ns;  -- read delay
            hzd       : TIME      := 2 ns;  -- HighZ delay
    PORT(  datos      : INOUT STD_LOGIC_VECTOR(7 DOWNT0) := "ZZZZZZZZ";
            direc      : IN BIT_VECTOR((2*addrbits - 1) DOWNT0 0);
            ncs        : IN BIT := '1';
            nwe        : IN BIT := '1');
END stat_ram;
```

En la región declarativa de la arquitectura se declaran :

- La constante `memdim` que fija la dimensión de la memoria en función de los bits de dirección.
- El tipo memoria, que define a la memoria como un array lineal de (0 TO `memdim`) elementos constituido por bytes del tipo `STD_ULOGIC_VECTOR (7 DOWNT0 0)`.

```
TYPE memoria IS ARRAY ( 0 TO memdim ) OF STD_ULOGIC_VECTOR (7 DOWNT0 0);
```

lo que servirá, entre otras cosas, para detectar valores distintos a los permitidos '1' o '0' por medio de la función `PUNTERO`, definida para que devuelva un valor entero que apunte a la dirección de memoria -(0 TO `memdim`)- donde se va a leer o escribir y que, por medio de una sentencia `ASSERT`, detecta valores no permitidos en el array `ULOGIC_VECTOR` que convierte a entero.

- Las señales internas "palabra" y "memdato" en relación a lo anterior y para estructurar y facilitar la comprensión del algoritmo de la RAM.

Así, la arquitectura y su región declarativa se describen como :

```
ARCHITECTURE sencilla OF stat_ram IS
  CONSTANT memdim    : POSITIVE := 2**(2*addrbits) -1;
  TYPE memoria IS ARRAY ( 0 TO memdim ) OF STD_ULOGIC_VECTOR (7 DOWNT0 0);
  SIGNAL memdato      : memoria;
  SIGNAL palabra       : INTEGER;

  FUNCTION puntero(val: STD_ULOGIC_VECTOR) RETURN INTEGER IS
    VARIABLE suma: INTEGER;
  BEGIN
    suma := 0;
    FOR N IN val'LOW to val'HIGH loop
      ASSERT NOT (val(N) = 'X' or val(N) = 'Z')
      REPORT "entradas a PUNTERO con valor 'X' o 'Z'"
      SEVERITY WARNING;
      IF val(N) = '1' THEN
        suma := suma + (2**N);
      END IF;
    END LOOP;
    RETURN suma;
  END puntero;
END sencilla;
```

Finalmente la región ejecutable se inicia con la obtención del entero que apunta a la palabra de la memoria que indican los bits de la dirección, convertidos de BIT_VECTOR a STD_ULOGIC_VECTOR por la función de conversión **To_stdULogicVector** definida en el paquete STD_LOGIC_1164 y mostrada a continuación para referencia

Para la escritura de datos se hace uso de la función **To_stdULogicVector** aplicada a los datos en el bus (datos) , que son de tipo resuelto y se convierten a MVL9. De forma similar la sentencia de lectura utiliza la función **To_stdLogicVector** , complementaria de la anterior, para convertir los datos que existen en la memoria, de tipo no resuelto o MVL9 , al tipo resuelto que debe existir en el bus de datos.

```
BEGIN
  palabra <= puntero(To_stdULogicVector(direc));
funcionram : PROCESS(ncs)
  BEGIN
    IF ncs = '0' THEN
      IF nwe = '0' THEN      -- escribir dato en la palabra(direc) de la RAM
        memdato(palabra)<= To_stdULogicVector(datos) AFTER wrd;

      ELSE
        -- leer dato en la palabra(direc) de la RAM
        datos <= TO_STDLogicVector(memdato(palabra)) AFTER rdd;

      END IF;
      -- poner databus en alta impedancia ( ncs = '1' )
    ELSE
      datos <= "ZZZZZZZZ" AFTER hzd;
    END IF;
  END PROCESS funcionram;
END sencilla;
```

Las funciones de conversión del paquete std_logic_1164 que se utilizan son:

```
FUNCTION To_StdLogicVector ( b : BIT_VECTOR ) RETURN std_logic_vector IS
  ALIAS bv : BIT_VECTOR ( b'LENGTH-1 DOWNT0 0 ) IS b;
  VARIABLE result : std_logic_vector ( b'LENGTH-1 DOWNT0 0 );
BEGIN
  FOR i IN result'RANGE LOOP
    CASE bv(i) IS
      WHEN '0' => result(i) := '0';
      WHEN '1' => result(i) := '1';
    END CASE;
  END LOOP;
  RETURN result;
END;
----- OBSERVAR LA SOBRECARGA EXISTENTE-----
```

```
FUNCTION To_StdLogicVector ( s : std_ulogic_vector ) RETURN std_logic_vector IS
  ALIAS sv : std_ulogic_vector ( s'LENGTH-1 DOWNT0 0 ) IS s;
  VARIABLE result : std_logic_vector ( s'LENGTH-1 DOWNT0 0 );
BEGIN
  FOR i IN result'RANGE LOOP
    result(i) := sv(i);
  END LOOP;
  RETURN result;
END;
```

```
FUNCTION To_StdULogicVector ( b : BIT_VECTOR ) RETURN std_ulogic_vector IS
  ALIAS bv : BIT_VECTOR ( b'LENGTH-1 DOWNT0 0 ) IS b;
  VARIABLE result : std_ulogic_vector ( b'LENGTH-1 DOWNT0 0 );
BEGIN
  FOR i IN result'RANGE LOOP
    CASE bv(i) IS
      WHEN '0' => result(i) := '0';
      WHEN '1' => result(i) := '1';
    END CASE;
  END LOOP;
  RETURN result;
END;
----- OBSERVAR LA SOBRECARGA EXISTENTE -----
```

```
FUNCTION To_StdULogicVector ( s : std_logic_vector ) RETURN std_ulogic_vector IS
  ALIAS sv : std_logic_vector ( s'LENGTH-1 DOWNT0 0 ) IS s;
  VARIABLE result : std_ulogic_vector ( s'LENGTH-1 DOWNT0 0 );
BEGIN
  FOR i IN result'RANGE LOOP
    result(i) := sv(i);
  END LOOP;
  RETURN result;
END;
```