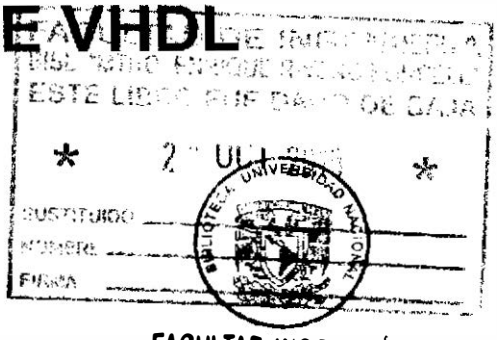
Lenguaje de descripción de hardware VHDL

Jorge
Valeriano Assem
Norma Elva
Chávez Rodríguez

LENGUAJE DE DESCRIPCIÓN DE HARDWARE VHDL



FACULTAD DE INGENIERÍA
DIVISIÓN DE INGENIERÍA ELÉCTRICA
DEPARTAMENTO DE INGENIERÍA EN COMPUTACIÓN
E INGENIERÍA EN ELECTRÓNICA

ING. JORGE VALERIANO ASSEM
ING. NORMA ELVA CHÁVEZ RODRÍGUEZ

CIUDAD UNIVERSITARIA, AGOSTO 2002

ÍNDICE

1. INTRODUCCIÓN

Con las complejidades de los circuitos integrados (chips), cerca de 50 millones de dispositivos activos, ya no es posible el diseño de hardware usando métodos basados unicamente en esquemáticos. Así inevitablemente los diseños deben ahora tambièn utilizar especificaciones de alto nivel y procesos de síntesis.

De ésta forma actualmente se dispone de los llamados HDL's, que son lenguajes de descripción de hardware los cuales nos permiten describir hardware utilizando especificaciones de alto nivel; es decir utilizando un lenguaje. Haciendo una analogía los HDL's son lenguajes de alto nivel que nos sirven para crear hardware; Así como el lenguaje "C" nos sirve para generar software.

Los HDL's disponibles actualmente nos permiten diferentes formas de especificar un diseño; a éstas formas se les conoce como *estilos de escritura*, los cuales van desde especificaciones puramente funcionales hasta especificaciones estructurales, lo mismo que un esquemático.

Así entonces tenemos que **VHDL** (Very High Speed Integrated Circuit Hardware Description Languaje) es uno de éstos lenguajes de descripción de hardware. VHDL el cual fue originado por el Departamento de Defensa de Estados Unidos de América a mediados de 1980.

VHDL fue establecido como un estándar IEEE 1076 en 1987, y en 1993 fue actualizado para dar lugar al estándar IEEE 1164. De ésta manera VHDL se ha convertido en un estándar industrial para la descripción, modelado y síntesis de circuitos y sistemas digitales. Debemos destacar dos aspectos importantes, en primer lugar VHDL nos sirve para diseñar solamente circuitos digitales y en segundo lugar las herramientas comerciales que existen y aceptan VHDL, sintetizan sólo un subconjunto del lenguaje. Éstas ideas se muestran en la Fig. 1.1 y Fig. 1.2.

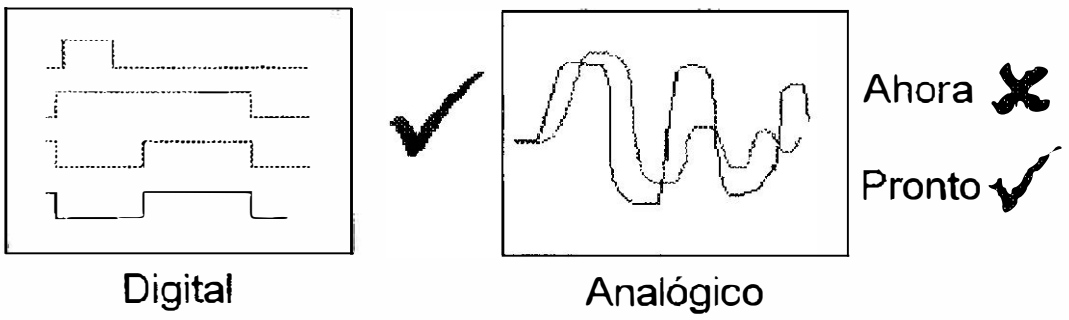


Fig. 1.1. Hoy podemos diseñar solamente circuitos digitales con VHDL.

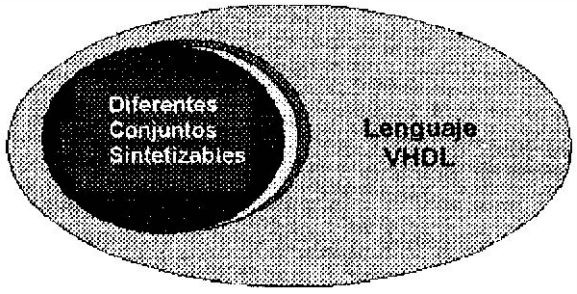


Fig. 1.2. Las herramientas actuales sintetizan diferentes conjuntos del lenguaje VHDL.

¿Por qué utilizar VHDL?

El lenguaje tiene las siguientes características:

- Los diseños se pueden descomponer en forma jerárquica.
- Cada elemento de diseño tiene una interfaz bien definida (para conectarlo a otros elementos) y una especificación precisa del comportamiento para simularlo.
- Las especificaciones de comportamiento pueden usar un algoritmo o una estructura actual de hardware para definir la operación que realiza un elemento.
- Se puede modelar la concurrencia, inherente al hardware. VHDL puede manejar estructuras de circuitos secuenciales asíncronos y síncronos.
- La operación lógica y el comportamiento en el tiempo de un diseño se puede simular.

VHDL comenzó como un lenguaje de documentación y modelado, permitiendo que el comportamiento de diseños digitales se especificaran y simularan de forma precisa. La utilidad y popularidad de VHDL se ha incrementado con el desarrollo comercial de las herramientas de síntesis para VHDL. Éstos programas pueden crear estructuras de circuitos lógicos directamente de la descripción de comportamiento de VHDL. Es decir, usando VHDL se puede diseñar, simular y sintetizar en un chip cualquier diseño desde un circuito combinacional simple hasta un sistema de microprocesador completo.

La razón principal para utilizar VHDL es porque con éste se pueden describir y sintetizar rápidamente circuitos de 5, 10 ò 20 mil compuertas. Diseños equivalentes descritos con esquemáticos ò ecuaciones booleanas en el nivel de transferencia de registros pueden requerir varios meses de trabajo por una persona. A continuación citamos algunas otras razones para utilizar VHDL.

Potencia y flexibilidad.

VHDL tiene poderosos constructores con los cuales es muy sencillo escribir código que describa lógica de control compleja. Soporta librerías de diseño y la creación de componentes re-utilizables. Provee jerarquía de diseño para crear diseños modulares.

Diseño independiente del dispositivo.

VHDL permite crear un diseño sin tener que escoger a priori un dispositivo en el cual implementarlo. Con una sola descripción del diseño, éste se puede implementar en muchas dispositivos con arquitecturas diferentes. No es necesario concentrarse en la arquitectura particular de un dispositivo a fin de optimizar el diseño. VHDL también permite múltiples estilos de descripción de un diseño.

Portabilidad.

VHDL permite simular la misma descripción del diseño que se sintetiza. Lo cual permite localizar errores en el funcionamiento del sistema y corregirlos antes de que se implemente físicamente el diseño. Debido a que la descripción de un diseño en VHDL es estándar, éste puede ser llevado de un simulador a otro, de una herramienta de síntesis a otra y de una plataforma a otra.

Migración a un ASIC.

Cuándo los volúmenes de producción alcanzan niveles apropiados, VHDL facilita el desarrollo de un ASIC (Application Specific Integrated Circuit), el cual generalmente reduce el costo del componente y su manufactura, ya que se reduce el tamaño físico del chip y su consumo de potencia, además generalmente ofrecen un desempeño mayor. Así algunas veces el código usado con un PLD puede ser usado con un ASIC. Además como VHDL es un lenguaje bien definido, se puede éstar seguro de que el vendedor de ASIC's entregará un dispositivo con la funcionalidad que se espera.

Flujo de diseño.

Hay varios pasos en el proceso de un diseño basado en VHDL, a éstos usualmente se le conoce como el flujo de diseño. Éstos pasos se aplican a cualquier HDL.

El flujo comienza con la descripción del sistema a un nivel de bloques. Los diseños lógicos grandes son usualmente jerárquicos y VHDL da un buen marco de trabajo para definir módulos y sus interfaces, complementándolos en detalle después.

El siguiente paso consiste en escribir el código VHDL para los módulos, sus interfaces y sus detalles internos. Dado que VHDL es un lenguaje basado en texto en principio se puede escribir el código en cualquier editor de texto, sin embargo, la mayoría de los ambientes de diseño incluyen un editor de texto especializado para VHDL, el cual hace el trabajo más fácil. Éstos editores incluyen herramientas

como sobresaltado de palabras reservadas de VHDL, indentación automática, patrones para estructuras de programa frecuentemente usadas, etc.
Después de escribir el código se continua con la compilación del mismo. Un compilador de VHDL analiza en el código los errores de sintaxis y checa la compatibilidad con otros módulos. También crea la información interna necesaria para que un simulador procese posteriormente el diseño.

El siguiente paso es la simulación. Un simulador de VHDL permite definir y aplicar entradas al diseño y observar sus salidas, sin tener que construir físicamente el sistema. Además para proyectos grandes VHDL da la habilidad para crear "test benches" los cuales automáticamente aplican entradas y las comparan con las salidas esperadas.

A continuación prosiguen las etapas de síntesis, fitting/place+route, timing verification, las cuales dependen en gran medida de la tecnología del dispositivo en la que se va a implementar el diseño. La etapa de síntesis convierte la descripción VHDL en un conjunto de primitivas o componentes que pueden ser ensamblados en la tecnología del dispositivo elegido. Por ejemplo con PLDs o CPLDs la herramienta de síntesis puede generar ecuaciones de suma de productos, con ASICs, ésta puede generar una lista de compuertas y un netlist que especifique como deberían ser interconectadas. Aquí el diseñador puede ayudar a la herramienta de síntesis especificando ciertas restricciones en la tecnología especifica que eligió para implementar su diseño con un número máximo de niveles lógicos o los buffers lógicos a usar. En el paso de fitting, una herramienta mapea las primitivas o componentes sintetizados en los recursos disponibles del dispositivo. Para un PLD o CPLD esto puede significar el asignar ecuaciones o los elementos AND-OR disponibles. Para un ASIC, esto puede significar la creación de capas (layers) necesarias para compuertas individuales que sigan un patrón y encontrar la forma de conectarlas dentro de las restricciones físicas del ASIC; a esto se le conoce como el proceso de place-and-route. El diseñador puede adicionalmente especificar restricciones adicionales en ésta etapa, tales como la colocación de módulos en el chip o la asignación de pines a entradas y salidas externas.

El último paso es la verificación de tiempos del circuito implementado. En ésta etapa es donde se pueden calcular con razonable precisión los retrasos en el circuito debidos a la longitud de los alambre (internos), carga eléctrica, etc. Es usual que durante éste paso se apliquen los mismo casos de prueba que fueron usados en la simulación, sólo que en éste caso se aplican al circuito como quedará al construirse.

índice

2.ENTIDADES Y ARQUITECTURAS

El objetivo de ésta sección es entender el concepto, utilidad y utilización de las dos entidades principales del lenguaje VHDL, llamadas *Entidades y Arquitecturas*. Entidades que definen la interfaz y la funcionalidad de un sistema.

2.1. Introducción

Los bloques constitutivos básicos de cualquier diseño en VHDL se llaman ***declaración de entidad*** y ***cuerpo de la arquitectura***. También serán explicados tres estilos básicos de codificación, haciendo analogías con diseños esquemáticos y programación en lenguajes de alto nivel con la intención de ayudar a colocar a VHDL en un marco familiar. Sin embargo exhortamos a los lectores a ser muy cuidadosos y no abusar de las analogías porque codificar en VHDL es muy diferente en relación a codificar en lenguajes de programación. Siempre debemos tener en mente que al usar VHDL lo hacemos para diseñar **hardware**. El código producido de ésta manera será sintetizado en lógica digital para ser programado en dispositivos lógicos.

índice

2.2. Entidades

La entidad es la unidad primaria del lenguaje que identifica a los dispositivos mediante la definición de su interfaz, esto es su nombre, terminales de conexión y parámetros de instanciación. Es posible utilizarla también para declarar objetos señales y constantes, tipos de datos y subprogramas. Se puede decir que la entidad desempeña una función equivalente a la del símbolo de un dispositivo en los esquemas de circuitos.

Ésta unidad de descripción se construye con los elementos predefinidos del lenguaje como son los identificadores de unidades, sentencias, declaraciones, tipos de datos, operadores etc.

Bajo éstos términos una entidad es una abstracción de un diseño, que puede representar un sistema completo, una tarjeta, un chip, una pequeña función o simplemente una compuerta lógica.

Así podemos pensar que una entidad en VHDL está constituida por un par llamado *declaración de la entidad* y *cuerpo de la arquitectura* éste último será visto más adelante. Ésta idea se ilustra en la Fig. 2.1.

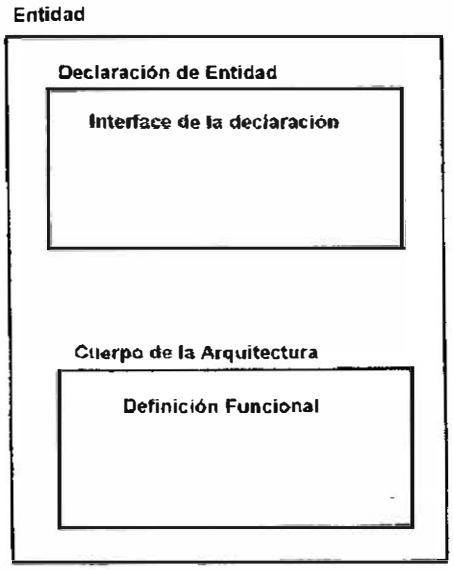


Fig. 2.1. Entidad en VHDL

2.2.1 Declaración de las Entidades

indice

La declaración de una entidad describe las entradas y salidas del diseño de una entidad. También puede describir valores parametrizados. Ésta descripción de entradas y salidas es útil en la utilización de la entidad dentro de un diseño jerárquico. Enseguida mostramos la declaración de una entidad que corresponde a un sumador de cuatro bits cuyo símbolo se ilustra en la Fig. 2.2. Se observa la analogía entre la declaración de la entidad y el símbolo.

```
entity add4 is port (
        a, b  :  in std_logic_vector(3 downto 0);
        cin   :  in std_logic;
        sum:  out std_logic_vector(3 downto 0);
        co    :  out std_logic);
end add4;
```

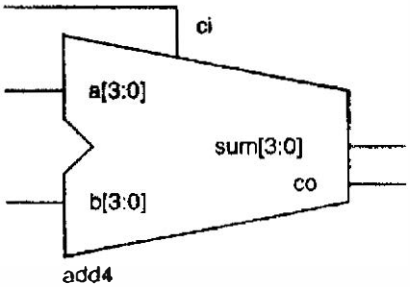


Fig. 2.2. Símbolo equivalente a la entidad add4

2.2.2 Puertos

índice

Cada señal de E/S en la declaración de una entidad se refiere con el nombre de **port** (puerto) que es la analogía con un pin en el símbolo esquemático. Un puerto es un objeto de datos, por tanto, como cualquier objeto de datos se le pueden asignar valores y usar en expresiones. El conjunto de puertos definidos para una entidad se refiere como **port declaration** (declaración de puerto). Cada puerto que se declare debe tener un nombre, una dirección (conocida como *modo*) y un tipo de dato asociado.

Modos. El modo describe la dirección en la cual los datos son transferidos hacia el puerto. El modo puede ser uno de cuatro valores: *in, out, inout* y *buffer*. En el caso de que el modo de un puerto no sea especificado se asume que es *in*. El uso de éstos modos es el siguiente y se ilustran en la Fig. 2.3.

- **In.** Los datos fluyen únicamente hacia el interior de la entidad. El operador para el puerto en éste modo es externo a la entidad. Éste modo es utilizado principalmente para transferir señales de entrada de reloj, entradas de control (como carga, inicialización, habilitación), y entradas de datos unidireccionales.

- **Out.** Los datos fluyen únicamente desde la fuente en el interior de la entidad hacia el puerto de salida de la entidad. Éste modo es utilizado para salidas tales como salidas de terminación de conteo, salidas de control, etc.

- **Buffer.** Se utiliza para retroalimentación interna. Esto es, el puerto también se usa como un operador dentro de la misma arquitectura. Un puerto que es declarado en modo buffer es similar a un puerto que es declarado como modo de salida, excepto que el de salida no permite retroalimentación. Un puerto del modo buffer se puede conectar solamente a una señal interna o a un puerto de modo buffer de otra entidad, pero nunca a un puerto del tipo *out* o *inout*.

- **Inout.** Para el manejo de señales bidireccionales se debe declarar un puerto del modo *inout*, el cual permite que los datos fluyan hacia el interior o exterior de la entidad, éste modo también permite la retroalimentación interna.

Tipos. Adicionalmente a los identificadores de los puertos y sus modos, también se debe declarar los tipos de datos de los puertos. Éstos tipos son provistos por el estándar IEEE 1076/93, que son los más usuales y soportados por las herramientas de síntesis. Éstos tipos son: *boolean, bit, bit_vector y integer.* Los más útiles y bien soportados son los provistos por el paquete IEEE *std_logic_1164*, éstos tipos son *std_ulogic* y *std_logic* así como arreglos de éstos tipos. Para que éstos tipos sean reconocidos es necesario hacerlos visibles a la entidad mediante la forma de una librería y cláusulas como en los lenguajes de programación. De ésta manera la declaración de la entidad *add4* descrita

anteriormente no puede ser procesada por las herramientas de software hasta ser modificada de la siguiente manera:

```
library ieee;
use ieee.std_logic_1164.all;
entity add4 is port (
        a, b  :  in std_logic_vector(3 downto 0);
        cin   :  in std_logic;
        sum:  out std_logic_vector(3 downto 0);
        co    :  out std_logic);
end add4;
```

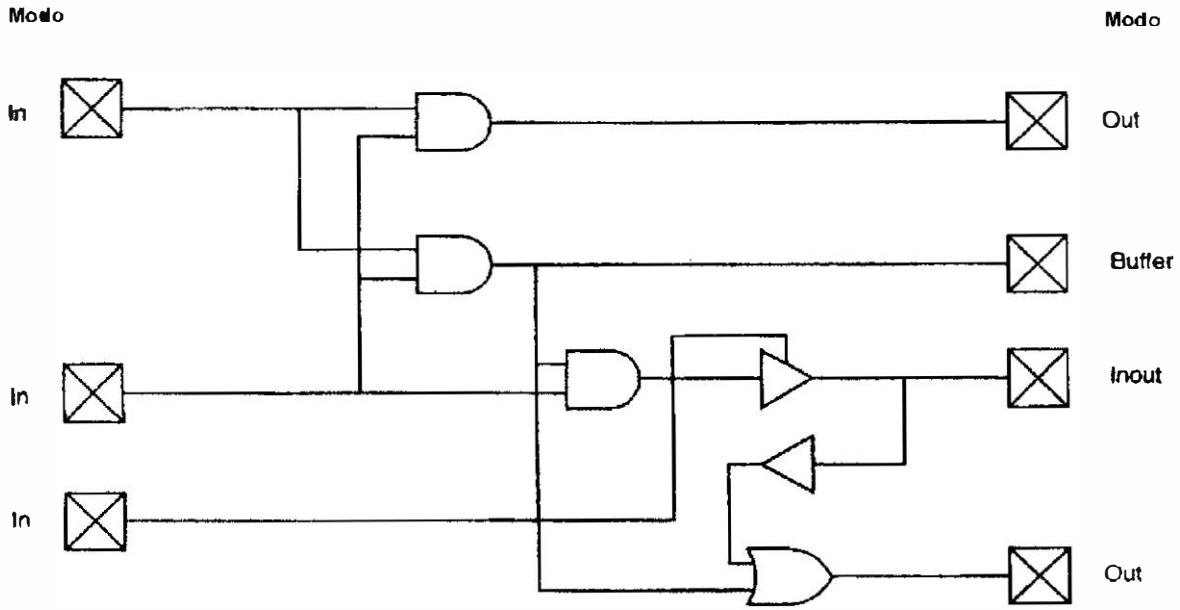


Fig. 2.3. Diferentes modos con sus señales de origen

2.3. Cuerpo de la arquitectura

indice

La arquitectura es una unidad secundaria del lenguaje. Ésta asociada a una determinada declaración de entidad, se encarga de describir el funcionamiento del dispositivo identificado por ésta. Para ello se sirve de los procesos, en las descripciones funcionales y de los componentes en las estructurales. Esto significa que la entidad es vista como la *caja negra*, donde no sabemos lo que hay dentro, pero en el cuerpo de la arquitectura se describe el contenido de ella.

VHDL permite escribir los diseños utilizando los siguientes tres estilos de arquitecturas diferentes:

- Descripción de flujo de datos **(data-flow)**
- Descripción por comportamiento **(behavioral)**
- Descripción estructural **(estructural)**

Cada uno de éstos estilos permite escribir un diseño con niveles diferentes de abstracción, incluso la combinación de ellos.

2.3.1. Descripción de flujo de datos (data-flow)

índice

En éste estilo de diseño las siguientes dos características son utilizadas en el diseño:

- Se especifica como los datos son transferidos de los puertos de entrada a los puertos de salida.
- Se utilizan únicamente asignaciones mediante expresiones en las que se indica como cambian los puertos de salida en función de los puertos de entrada, ya sean asignaciones condicionales mediante instrucciones concurrentes o simples ecuaciones.

Cabe destacar que en lenguajes de programación como "C" o Pascal, cada instrucción de asignación es ejecutada una después de otra en un orden específico, dependiendo del orden de aparición en el archivo.

En VHDL todos los bloques son concurrentes, es decir se están ejecutando en todo momento. Esto significa que dentro de una arquitectura en VHDL, no existe un orden específico de ejecución de las asignaciones, por tanto el orden en que las instrucciones son ejecutadas depende de los eventos ocurridos en las señales, como sucede en un circuito. Finalmente esto significa que en éste estilo no se utilizan sentencias secuenciales, sino solamente asignaciones concurrentes. El siguiente código VHDL define en éste estilo un comparador de cuatro bits.

```
library ieee;
use ieee.std_logic_1164.all;
entity eqcomp4 is
port (a, b:      in std_logic_vector(3 downto 0);
      equals:  out std_logic);
end eqcomp4;

architecture dataflow of eqcomp4 is
begin
equals <= '1' when (a = b) else '0';
end dataflow;
```

2.3.1.1 Estructuras de Ejecución Concurrente

índice

En el estilo de descripción *dataflow* existen estructuras que se ejecutan concurrentemente e indican asignación a puertos de salida en función de los puertos de entrada. Las estructuras comunes son:

➢ Asignación Condicional WHEN ... ELSE

Sintaxis:

```
Identificador_de_señal <= valor_a WHEN condición ELSE
                          valor_b WHEN condición ELSE
                          ...
                          ...
                          valor_n WHEN condición ELSE
                          valor_default
```

Ejemplo:

```
x <= '1' WHEN seleccion = "4" ELSE '0';
```

➢ Asignación WHIT ... SELECT ... WHEN

Sintaxis:

```
WITH identificador SELECT
Señal_salida <= valor_a WHEN identificador_valor1,
                valor_b WHEN identificador_valor2,
                ...
                ...
                valor_n WHEN OTHERS;
```

Ejemplo:

```
WITH estados SELECT
salida <= "000" WHEN state0,
          "001" WHEN state1,
          "010" WHEN state2,
          "100" WHEN state3,
          "000" WHEN OTHERS;
```

➢ Ecuaciones Booleanas

Sintaxis:

```
identificador_de_señal <= ecuación_booleana;
```

Ejemplos:

```
x <= y and z;
a <= (b or c or d) and e;
op1 <= op2 nor op3 nor op4;
```

En el siguiente ejemplo presentamos el diseño de una ALU (Unidad Aritmético Lógica) utilizando el estilo *dataflow*. Las funciones de la ALU están descritas en la siguiente tabla:

ENTRADAS		SALIDAS	
s1	**s0**	**z**	**co**
0	0	X and Y	0
0	1	X or Y	0
1	0	X xor Y	0
1	1	X + y + cin	cout

El código que describe la ALU es:

```
library ieee;
use ieee.std_logic_1164.all ;
use ieee.numeric_std.all;

entity alu is
port ( x, y    : in std_logic;
       s0, s1 : in std_logic;
       ci      : in std_logic;
       z       : out std_logic;
       co      : out std_logic);
end alu;

architecture  a_alu of alu is
  signal seleccion : unsigned (1 downto 0) ;
  signal suma      : std_logic;
  signal and_op    : std_logic;
  signal or_op     : std_logic;
  signal xor_op    : std_logic;
  signal acarreo   : std_logic;

begin
  and_op <= x and y;
  suma <= x  xor y  xor ci;
```

```
  or_op <= x or y;
  acarreo <= (x and y) or  (x and ci) or (y and ci);
  xor_op <= x xor y;
with seleccion select
   z <= or_op when "01",
        and_op when "00",
        suma when "11",
        xor_op when "10",
        '0' when others;
seleccion <= s 1 & s0;
co <= acarreo when seleccion = 3 else '0';
end a_alu;
```

2.3.2. Descripción por comportamiento (behavioral)

indice

Las descripciones por comportamiento son similares a las descripciones hechas en un lenguaje de programación de alto nivel, debido a su nivel de abstracción. En una descripción de éste tipo no se especifica la estructura o la forma en que se deben conectar los componentes, sino únicamente nos limitamos a describir su comportamiento.

Una descripción por comportamiento consiste de una serie de instrucciones, que ejecutadas secuencialmente, modelan el comportamiento del circuito. Con éste grado de abstracción no requerimos enfocarnos a un nivel de compuerta para implementar un diseño.

En VHDL las descripciones por comportamiento implican el uso de por lo menos un bloque **Process**, en el interior de éstos bloques se escriben las instrucciones de VHDL que son ejecutadas secuencialmente una después de otra.

Los procesos se pueden utilizar en el interior de cualquier arquitectura definiendo para sí mismo una región de declaraciones y otra para la codificación secuencial. Ésta región de codificación puede contener únicamente instrucciones secuenciales. En la zona de declaraciones se permite designar constantes, señales, tipos de datos o algún alias. Lo que hay dentro de un proceso tiene mucha semejanza a la programación utilizando un lenguaje de alto nivel.

La sintaxis de un proceso es la siguiente:

```
process (lista sensitiva)
        -- declaraciones
begin
        -- instrucciones secuenciales
end process;
```

La *lista sensitiva* es opcional y define que señales provocan que las instrucciones dentro del bloque comiencen a ser ejecutadas. De ésta manera cualquier cambio en alguna de las señales de esa lista provoca que el proceso sea llamado. Si un proceso no tiene lista sensitiva, entonces debe contener una instrucción *wait* (que se verá más adelante) para especificar cuándo deben ser ejecutadas las instrucciones dentro del bloque.

La especificación de ésta lista nos puede servir para especificar si estamos modelando lógica combinacional o secuencial. Se dice que una lista sensitiva es parcialmente declarada cuándo alguna de las señales que intervienen del lado derecho de una ecuación o de alguna instrucción secuencial no es mencionada dentro de la lista.

El funcionamiento del proceso es de la siguiente manera:

- Las señales dentro de la lista sensitiva funcionan como entradas de interrupción y las instrucciones secuenciales se encuentran dentro de la rutina única de servicio de interrupción.

- Cuándo alguna de las señales de la lista sensible cambia, provoca que el proceso comience a funcionar y a ejecutar toda ésta rutina de ejecución secuencial con la particularidad de que lo que resulte de éste procesamiento se asigne únicamente al final de la estructura.

Enseguida mostramos el código utilizado para generar un comparador de dos palabras de cuatro bits.

```
library ieee;
use ieee.std_logic_1164.all;
entity eqcomp4 is port (
        a, b    : in std_logic_vector(3 downto 0);
        equals : out std_logic);
end eqcomp4;

architecture behavioral of eqcomp4 is
begin
 comp: process (a, b)
        begin
          if a = b then
            equals <= '1';
          else
            equals <= '0';
          end if;
        end process comp;
 end behavioral;
```

Ésta arquitectura únicamente tiene una instrucción concurrente constituida por el bloque *process* el cual es sensible a los vectores de entrada. Siempre que exista

un cambio en alguno de ellos, el proceso será llamado y generará la lógica de salida. Podemos observar que la lista sensitiva está completa. Cada instrucción dentro del proceso será ejecutada en orden secuencial y cuándo todas hallan sido ejecutadas, entonces se asigna el valor procesado a los nodos que se vieron afectados durante el proceso. Una vez terminado de ejecutar el proceso, éste se mantendrá inactivo hasta que alguno de los dos elementos en la lista sensitiva cambie. Las instrucciones que se pueden ejecutar en los bloques secuenciales son presentadas a continuación.

2.3.2.1 Estructuras de Ejecución Secuencial

indice

- **IF - THEN - ELSE**

Ésta instrucción secuencial permite probar una condición y elegir dos caminos de acción diferentes dependiendo de si la condición se cumple o no. Es equivalente a un multiplexor de dos entradas y una línea de selección. La sintaxis es la siguiente:

Sintaxis:

```
If condición then
   ...
elseif condición then
   ...
end if;
```

Ejemplo:

```
signal conteo: unsigned (3 downto 0);
   ...
if conteo = "9" then
  conteo <= (others => '0');
else
  conteo <= conteo + 1;
end if;
```

- **CASE - WHEN**

Ésta instrucción secuencial permite una bifurcación múltiple dependiendo del valor resultante de una expresión, se elige alguna alternativa, que es la que se ejecuta; en caso de que ninguna alternativa se cumpla, se ejecuta lo que contenga la alternativa **others**. La sintaxis es:

Sintaxis:

```
case expresión is
when alternativa1 =>
      ...
when alternativa2 =>
      ...
when others  =>
      ...
end case;
```

Ejemplo:

```
type éstados is (éstado1, éstado2, éstado3, éstado4)
signal éstado_máquina: éstados;
signal motor, alarma: bit;
constant encedido: bit :='1';
constant apagado: bit :='0';
...
case éstado_máquina is
  when éstado0 =>
        motor <= apagado;
  when éstado1 =>
        motor <= encedido;
  when (éstado3 or éstado4) =>
        alarma <= encendido;
  when others =>
        motor <= apagado;
        alarma <= apagado;
end case;
```

- **FOR - LOOP**

Ésta instrucción secuencial es un ciclo iterativo que permite repetir un bloque secuencial un determinado número de veces conocido previamente mediante una variable conocida que sirve para contar las iteraciones. Su sintaxis es la siguiente:

Sintaxis:

```
for identificador in rango loop
    ...
end loop;
```

Ejemplo:

```
for i in 3 downto 0 loop -- i es una variable y no necesita ser declarada
  If reset(i) = '1' then
     Data_out(i) <= '0';
  end if;
end loop;
```

➢ **WHILE - LOOP**

Ésta instrucción secuencial es un ciclo iterativo que permite repetir un bloque secuencial un número de veces no determinado, el número de iteraciones depende de la condición que se prueba al inicio del ciclo. Su sintaxis es la siguiente:

Sintaxis:

```
while condición loop
...
end loop;
```

Ejemplo:

```
contador := 0;
resultado_tmp := 0;
while contador > 0 loop
contador := contador – 1;
resultado_tmp := resultado_tmp + data_in;
end loop;
resultado <= resultado_tmp;
```

➢ **WAIT**

Ésta instrucción es utilizada en procesos que no tienen lista sensitiva, debido a que ésta instrucción define implícitamente la lista sensible del proceso. Existen tres variantes de ésta instrucción:

Wait on: Espera los cambios de las señales especificadas. Ésta instrucción no es aceptada por la mayoría de las herramientas de síntesis.

Wait for: Detiene la simulación durante el tiempo especificado, sólo se utiliza para simulaciones.

Wait until: Espera a que se cumpla la condición especificada. Ésta forma si es sintetizable y puede utilizarse sin tener problemas.

2.3.3. Descripción estructural (estructural)

indice

En una descripción estructural se declaran los componentes que se utilizan, se crean las instancias de los mismos (es decir se ponen los componentes) y después mediante los nombres de los nodos, se realizan las conexiones entre las instancias de los componentes. Las descripciones estructurales son útiles cuándo se trata de diseños jerárquicos bien modularizados.

Un componente representa a una entidad declarada en un diseño o librería. Para poder utilizar una entidad que está dentro de otro diseño, es necesario llamar a la librería y el paquete dentro del cual se encuentra ésta entidad.

Sintaxis de la declaración de componentes

Component *identificador_componente*
Port*(identificador {, identificador}: modo tipo_de_dato*
{; identificador {, identificador}: modo tipo_de_dato });
end component

Ejemplo

component *add*
port*(a, b, cin : **in** std_logic;*
*suma, co : **out** std_logic);*
end component*;*

La instanciación del componente puede entenderse como una instrucción concurrente que especifica la interconexión de las señales del componente dentro del diseño en el que está siendo utilizado. Existen dos formas de hacer la instanciación de los componentes.

- **Por asociación de identificadores.**

En éste tipo es necesario utilizar el operador de asociación "=>" para indicar como se conectan los puertos del componente con los puertos o señales de la arquitectura en la que está siendo utilizado dicho componente.
Por ejemplo en la asociación "a =>", tenemos que "a" pertenece al componente y "b" es una señal, variable o incluso una ecuación booleana en la que intervienen objetos de datos que pertenecen a la arquitectura donde se usa el componente

Ejemplo:

architecture *a_reg8* **of** *reg8* ***is***
signal *clock, reset, enable : std_logic;*

```
   signal data_in, data_out : std_logic_vector (7 downto 0);
begin
  register8 port map (clk=>clock, rst=>reset, en=>enable, data=> data_in,
                      q=>data_out);
end a_reg8;
```

- **Por asociación de posición.**

En éste tipo no es necesario nombrar los puertos del componente. Solamente se colocan las señales, variables, o expresiones en el lugar donde deseamos que sean conectadas. El orden en el que fueron declarados los puertos del componente es el orden que se debe utilizar cuándo se haga la instanciación del componente.

Ejemplo:

```
architecture a_reg8 of reg8 is
   signal clock, reset, enable : std_logic;
   signal data_in, data_out : std_logic_vector (7 downto 0);
begin
  register8 port map (clock, reset, enable, data_in, data_out);
end a_reg8;
```

A continuación mostramos un ejemplo, que muestra la similitud entre la descripción estructural y la descripción esquemática Fig. 2.4. Éste ejemplo describe el comparador de dos señales de cuatro bits diseñado anteriormente. Aquí se crean las instancias de todas las compuertas a utilizar y se interconectan entre si utilizando asociación por posición.

```
library ieee;
use ieee.std_logic_1164.all;
entity eqcomp4 is port(
        a, b   : in std_logic_vector(3 downto 0);
        aeqb : out std_logic);
end eqcomp4;

use work.gatespkg.all;
architecture struct of eqcomp4 is
  signal x : std_logic_vector(0 to 3);
begin
  u0 : xnor2 port map (a(0),b(0),x(0));
  u1 : xnor2 port map (a(1),b(1),x(1));
  u2 : xnor2 port map (a(2),b(2),x(2));
  u3 : xnor2 port map (a(3),b(3),x(3));
  u4 : and4 port map (x(0),x(1),x(2),x(3),equals);
end struct;
```

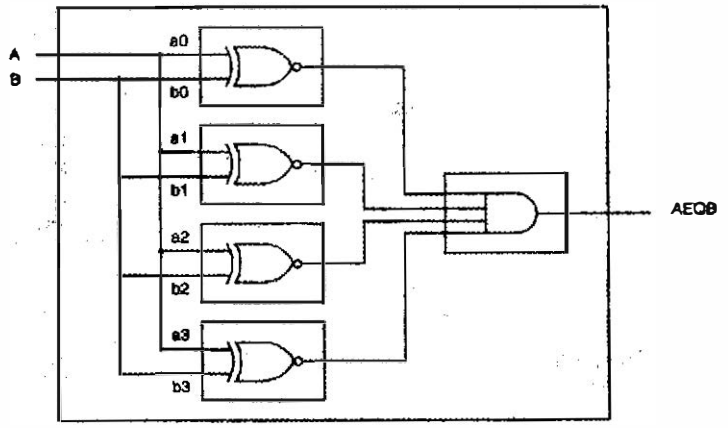


Fig. 2.4. Representación esquemática

2.4. Identificadores, objetos de datos, tipos de datos y atributos.

El objetivo de ésta sección es discutir detalles acerca de los identificadores, objetos de datos, tipos de datos y atributos, utilizando ejemplos para entender los conceptos básicos.

2.4.1 Identificadores

indice

Los identificadores son formados a partir de caracteres alfabéticos y numéricos que siguen las siguientes reglas:

- El primer caracter debe ser una letra
- El último caracter no puede ser un guión bajo
- Dos guiones bajos consecutivos no son permitidos.

Para el caso de los identificadores, mayúsculas y minúsculas es indistinto, es decir son equivalentes. Por ejemplo los siguientes son identificadores legales: *tx_clk, Three_State_Enable, sel7D, HIT_1124.*

2.4.2. Objetos de datos

indice

Los objetos de datos retienen valores de tipos específicos. Ellos pertenecen a una de cuatro clases: *constantes, señales, variables o archivos,* y deben ser declaradas antes de utilizarse.

- **Constantes.** Una constante mantiene un valor que no puede cambiar dentro de la descripción del diseño. Éste valor es usualmente asignado durante la declaración. Generalmente las constantes se utilizan para mejorar la descripción del código haciendo más sencillo cambios posteriores. Por ejemplo la siguiente constante representa la anchura de un registro:

constant *width: integer := 8;*

El identificador *width* puede ser utilizado en varios puntos del código, de ésta manera cuándo se desea cambiar la anchura del registro sólo es necesario cambiar el valor de la constante, y afectar de ésta manera todos los puntos donde se usa la constante.

Las constantes se pueden declarar en la zona declarativa de las entidades, arquitecturas o procesos, y será visible sólo dentro del ámbito que se declara.

- **Señales.** Las señales pueden representar alambres y por tanto pueden interconectar componentes. Los puertos son señales; de hecho, los puertos pueden ser específicamente declarados como señales. Las señales pueden ser entradas o salidas de compuertas lógicas. Por ejemplo la siguiente es una señal:

signal *count : bit_vector(3* ***downto*** *0);*

La señal "*count*" puede representar el éstado actual de un contador. Tal cual en éste caso entonces representa elementos de memoria o más específicamente alambres conectados a las salidas de esos elementos de memoria. Valores iniciales pueden ser asignados a señales pero realmente no tienen significado para la síntesis. Ejemplo:

signal *count ; bit_vector(3* ***downto*** *0) := "0101";*

El símbolo ":=" indica asignamiento inmediato y es utilizado para indicar el valor inicial de la señal. Para simulación es útil, pero para síntesis no lo es, debido a que no se sabe el éstado inicial al encender el dispositivo.

- **Variables.** Las variables son únicamente utilizadas en procesos y subprogramas (funciones y procedimientos), y deben ser declaradas en la zona declarativa de un proceso o un subprograma. Las variables no representan señales ni elementos de memoria. Las variables son utilizadas para propósitos computacionales. El siguiente es un ejemplo de una declaración e inicialización de variable:

variable *result : std_logic := '0';*

Los asignamientos a las variables son inmediatos. Las variables no tienen formas de onda de salida, por tanto ellas sólo mantienen un valor en un tiempo, para las variables el símbolo ":=" significa asignación inmediata.

Para propósitos de síntesis, el uso más común de las variables es para el manejo de índices, así como para el almacenamiento temporal de datos. El alcance de la variable es únicamente en el proceso en el cual ha sido declarada. El siguiente código muestra el modelado de una AND de 8 entradas utilizando variables.

```
architecture will_work of my_and is
begin
  process(a_bus)
    variable tmp: bit;
  begin
    tmp := '1';
    for i in 7 downto 0 loop
      tmp := a_bus(i) and tmp;
    end loop;
    x <= tmp;
  end process
end will_work;
```

El símbolo ":=" es utilizado para indicar una asignación inmediata a una variable, de ésta manera las iteraciones del loop resultan en una asignación inmediata. Al final del proceso se ejecuta la instrucción de asignar el resultado a la variable "x", usando el operador "<=" que asigna el resultado al final del proceso.

- **Archivos.** Los archivos pueden contener valores de un tipo específico. Se utilizan los archivos para leer datos que sirven de estímulos y escribir datos a la salida de los procesos de simulación.

- **Alias.** Un alias es un sobrenombre con el que se conoce también a cualquier identificador de algún objeto existente. Al referenciar el alias es lo mismo que referenciar al identificador original. Por ejemplo:

```
signal address : std_logic_vector(31 downto 0);
alias top_ad : std_logic_vector(3 downto 0) is address (31 downto 28);
alias bank : std_logic_vector(3 downto 0) is address (27 downto 24);
alias row_ad : std_logic_vector(11 downto 0) is address (23 downto 12);
```

2.4.3. Tipos de datos

indice

Cada objeto que podemos representar con VHDL, debe pertenecer a un conjunto bien definido, el cual se conoce como tipo de datos y se encarga de agrupar objetos con características comunes.

VHDL es un lenguaje fuertemente tipificado, lo cual significa que objetos de diferentes tipos no pueden ser asignados entre si sin la previa conversión de tipo. A continuación estudiaremos los diferentes tipos de datos.

- **Tipo escalar.** Los tipos escalares tienen un orden, lo cual permite que los operadores de relación sean utilizados. Hay cuatro categorías de tipos escalares: *enumeración, enteros, flotantes y fisicos.*

- **Tipo enumeración.** Un tipo enumeración es una lista de valores que un objeto de ese tipo puede tener. Son muy útiles cuándo se trabaja con máquinas de éstados. Ejemplo:

```
type states is (idle, preamble, data, jam, nosfd, error);
signal current_state : states;
```

El orden en el cual son listados en la declaración del tipo define su relación. El valor de más a la izquierda es menor que los otros valores.

Existen dos tipos enumerados predefinidos por el estándar IEEE 1076: *bit* y *boolean* que son definidos como:

```
type  boolean is (FALSE, TRUE);
type  bit is ('0', '1');
```

El estándar IEEE 1164 define un tipo adicional llamado *std_ulogic:*

```
Type std_ulogic is ( 'U',   -- no inicializado
                     'X',   -- Desconocido
                     '0',   -- Cero
                     '1',   -- uno
                     'Z',   -- alta impedancia
                     'W',   -- desconocido débil
                     'L',   -- cero débil
                     'H',   -- uno débil
                     '-'    -- no importa
                   );
```

También define un tipo llamado *std_logic* que es idéntico al *std_ulogic* con la diferencia de que tiene asociada una función de resolución que se utiliza en cada asignación. Los valores '0', '1', 'L' y 'H' son soportados por la síntesis. Los valores 'Z' y '-' también son soportados por la síntesis por los drivers de tres éstados y los valores no importan. Los valores 'U', 'X' y 'W' no son soportados por la síntesis.

Para poder usar éstos tipos es necesario escribir las siguientes líneas:

```
library ieee;
use ieee.std_logic_1164.all;
```

- **Tipo entero.** Los enteros y sus operadores aritméticos y relacionales son predefinidos en VHDL. Los enteros soportados son de –2,147,483,647 $-(2^{31}-1)$ a 2,147,483,647 $(2^{31}-1)$. Una señal o variable que es del tipo entero y que va a ser sintetizada debe tener asociado un intervalo. Por ejemplo:

***Variable** a: **integer range** -255 **to** 255;*

- **Tipo flotante.** El tipo punto flotante es utilizado para aproximar números reales. También como en los enteros se debe especificar un intervalo. El intervalo máximo definido es –1.0E38 a +1.0E38. Éste tipo no es comúnmente soportado por las herramientas de síntesis por la cantidad de recursos requeridos.

- **Tipo físico.** Los valores de tipos físicos son utilizados como unidades de medición, el único tipo predefinido es tipo *time*. Su intervalo incluye al mínimo el de los enteros. Su unidad primaria es el "fs" (femtosegundo) y se define como:

```
type time is range -2147483647 to 2147483647
  units
    fs;
    ps = 1000 fs;
    ns = 1000 ps;
    us = 1000 ns;
    ms = 1000 us;
    sec = 1000 ms;
    min = 60 sec;
    hr = 60 min;
  End units;
```

El tipo físico no es sintetizable, es útil solamente en las simulaciones.

- **Tipos Compuéstos.** Los objetos del tipo escalar pueden mantener sólo un valor en un tiempo de simulación. Los objetos de datos del tipo compuesto, pueden mantener múltiples valores en un tiempo dado. Éstos consisten del tipo **arreglo** y tipo **registro.**

Tipo arreglo. Un objeto del tipo arreglo consiste de múltiples elementos del mismo tipo, los más comunes son los siguientes:

```
type bit_vector is array (natural range <>) of bit;
type std_ulogic_vector is array (natural range <>) of std_ulogic;
type std_logic_vector is array (natural range <>) of std_logic;
```

La cláusula "**range** <>" significa que el número de bits no es especificado, es decir lo hará el usuario en el momento de utilizar éste tipo. El número de elementos en los tipos arreglos está acotado sólo por números positivos enteros. Éstos tipos son comúnmente utilizados para utilizar buses. Por ejemplo:

```
signal a: std_logic_vector (3 downto 0);
```

Sin embargo un bus puede también ser definido con nuestros propios tipos:

```
type word is array(15 downto 0) of bit;
signal b: word;
```

Si el usuario define, sus propios tipos, entonces también tiene que definir y sobrecargar los operadores para ese tipo. Por ejemplo los arreglos de dos dimensiones son útiles para crear tablas de verdad:

```
type table8x4 is array(0 to 7, 0 to 3) of bit;
constant exclusive_or: table8x4 := (
            "000_0",
            "001_1",
            "010_1",
            "011_0",
            "100_1",
            "101_0",
            "110_0",
            "111_1");
```

El caracter de subrayado se inserta solamente para distinguir entre las entradas y salidas representadas en la tabla.

Una asignación del tipo:

```
a <= X"7A";
```

requiere que "a" sea de ocho bits de longitud, dado que se le ésta asignando un número hexadecimal de dos dígitos. Los especificadores de base son "X" para hexadecimal, "O" para octal y "B" para binario

- **Tipo Registro.** Un objeto del tipo registro tiene múltiples elementos de diferentes tipos, los elementos individuales del registro pueden ser referenciados por el nombre del elemento. El siguiente es un ejemplo:

```
type iocell is record
  buffer_inp: bit_vector(7 downto 0);
  enable : bit;
  buffer_out : bit_vector(7 downto 0);
end record;

signal busa, busb, busc: iocell;
signal vec: bit_vector(7 downto 0);

busa.buffer_inp <= vec;   -- se asigna un bit_vector a otro bit_vector
busb.buffer_inp <= busa.buffer_inp;   -- se asigna un campo entre registros
busc <= busb;   -- se asigna el objeto entero
```

2.4.4. Tipos y subtipos

indice

A partir de los tipos base podemos crear otros tipos:

type *byte_size* ***is integer range*** *0* ***to*** *255;*
signal *my_int : byte_size;*

Aunque "byte_size" se basa en el tipo entero, éste tiene su propio tipo. De manera que para éste tipo operan todas las reglas especificas de los tipos, de ésta forma si declaramos la señal:

signal *your_int:* ***integer range*** *0* ***to*** *255;*

La siguiente operación producirá un error de compilación:

If *my_int = your_int* ***then*** ***...***

Los operandos en ésta comparación son de los tipos *byte_size* e *integer*, lo cual resulta en una mezcla de tipos.

Un subtipo es igual a su tipo base sólo que con algunas restricciones en su tamaño, por ejemplo, comparemos:

Subtype *byte* ***is*** *bit_vector (7* ***downto*** *0);*
Signal *byte1, byte2: byte;*
Signal *data1, data2: byte;*
Signal *addr1, addr2: byte;*

Con las declaraciones individuales:

Signal *byte3, byte4: bit_vector(7* ***downto*** *0);*
Signal *data3, data4: bit_vector(7* ***downto*** *0);*
Signal *addr3, addr4: bit_vector(7* ***downto*** *0);*

En éste caso, el siguiente código no generará un error en tiempo de compilación, debido a que "byte" está declarado como un subtipo entonces nunca existe mezcla de tipos:

If *byte1 = byte3* ***then*** ***...***

2.4.5. Atributos

indice

Los atributos extraen información acerca de entidades, arquitecturas, tipos y señales. Tomando en consideración las siguientes declaraciones, mostramos los ejemplos de la Tabla 2.1.

type *count* ***is*** *integer* ***range*** *0* ***to*** *127;*
type *states* ***is*** *(idle, decision, read, write);*
type *word* ***is array*** *(15* ***downto*** *0)* ***of*** *std_logic;*

Valor retornado	Descripción del atributo
Count'left = 0	'left obtiene el valor de más a la izquierda del tipo
States'left = idle	
Word'left = 15	
Count'right = 127	'right obtiene el valor de más a la derecha del tipo
States'right =write	
Word'right = 0	
Count'high = 127	'high obtiene el valor más grande del tipo
States'high = write	
Word'high = 15	
Count'low = 0	'low obtiene el valor más pequeño del tipo
States'low = idle	
Word'low=0	
Count'length = 128	'length obtiene el número de elementos de un arreglo
States'length = 4	
Word'length = 16	

Tabla 2.1. Descripciones de los atributos.

2. CREANDO LÓGICA COMBINACIONAL Y SÍNCRONA

El objetivo de ésta sección es describir la creación de lógica combinacional y síncrona utilizando los varios estilos de diseño descritos en la sección anterior (estructural, flujo de datos y por comportamiento).

3.1. Introducción indice

En la sección anterior estudiamos los conceptos de ***declaración de entidad*** y ***cuerpo de la arquitectura***. En ésta sección aprenderemos con ejemplos sencillos muchos de los constructores del lenguaje para definir el cuerpo de la arquitectura, es decir crearemos el hardware que involucra la caja negra que conocemos como *arquitectura.* Éste enfoque se hará para describir arquitecturas combinacionales y síncronas.

3.2. Ejemplo de diseño indice

Mostramos el código VHDL de un ejemplo que maneja conceptos vistos en la sección anterior, así como conceptos que serán aprendidos a lo largo de ésta sección. Se trata de una FIFO (cola) de 8 palabras cada una de 9 bits, con la conocida regla que el primero que entra es el primero que sale. Éste ejemplo nos va a servir para introducir constructores del lenguaje VHDL. No es necesario leer a detalle el código descrito a continuación, sino sólo identificar los nuevos constructores.

Dos paquetes son utilizados en éste diseño: *std_logic_1164* y *std_arith*. El primero es incluido para poder utilizar tipos como *std_logic* y *std_logic_vector*. El segundo paquete es incluido para poder acceder al operador sobrecargado "+" y poder sumar enteros con vectores.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;

entity fifo8x9 is port (
        clk, rst            : in std_logic;
        rd, wr, rdinc, wrinc : in std_logic;
        rdptrclr, wrptrclr   : in std_logic;
        data_in             : in std_logic_vector(8 downto 0);
        data_out            : out std_logic_vector(8 downto 0);
end fifo8x9;

architecture archfifo8x9 of fifo8x9 is
  type fifo_array is array(7 downto 0) of std_logic_vector(8 downto 0);

  signal fifo: fifo_array;
```

```
  signal wrptr, rdptr: std_logic_vector(2 downto 0);
  signal en: std_logic_vector(7 downto 0);
  signal dmuxout: std_logic_vector(8 downto 0);

begin

-- arreglo de registros de la fifo
reg_array : process (rst, clk)
begin
  if rst = '1' then
    for i in 7 downto 0 loop
      fifo(i) <= (others => '0'); -- agregación
    end loop;
  elsif (clk'event and clk = '1') then
    if wr = '1' then
      for i in 7 downto 0 loop
        if en(i) = '1' then
          fifo(i) <= data_in;
        else
          fifo(i) <= fifo(i);
        end if;
      end loop;
    end if;
  end if;
end process;

-- apuntador de lectura
read_count: process (rst, clk)
begin
  if rst = '1' then
    rdptr <= (others => '0');
  elsif (clk'event and clk = '1') then
    if rdptrclr = '1' then
      rdptr <= (others => '0');
    elsif rdinc = '1' then
      rdptr <= rdptr + 1;
    end if;
  end if;
end process;

-- apuntador de escritura
write_count: process (rst, clk)
begin
  if rst = '1' then
    wrptr <= (others => '0');
  elsif (clk'event and clk = '1') then
```

```
    if wrptrclr = '1' then
      wrptr <= (others => '0');
    elsif wrinc = '1' then
      wrptr <= wrptr + 1;
    end if;
  end if;
end process;

--multiplexor de datos de salida 8 a 1

with rdptr select
  dmuxout <=  fifo(0) when "000",
              fifo(1) when "001",
              fifo(2) when "010",
              fifo(3) when "011",
              fifo(4) when "100",
              fifo(5) when "101",
              fifo(6) when "110",
              fifo(7) when others;

-- decodificador selector del registro de la fifo

with wrptr select
  en <=   "00000001" when "000",
          "00000010" when "001",
          "00000100" when "010",
          "00001000" when "011",
          "00010000" when "100",
          "00100000" when "101",
          "01000000" when "110",
          "10000000" when others;

-- control tres éstados para las salidas

three-state: process (rd, dmuxout)
begin
  if rd = '1' then
    data_out <= dmuxout;
  else
    data_out <= (others => 'Z');
  end if;
end process;

end archfifo8x9;
```

3.3. Lógica combinacional indice

La lógica combinacional puede ser descrita de diferentes formas. En el ejemplo de la FIFO la lógica combinacional está descrita por las señales "dmuxout" y "en", así como por los buffers tres éstados. Las señales "dmuxout" y "en" utilizan constructores del estilo *flujo de datos* mediante el uso de sentencias *with-select-when*. Los buffers tres éstados son descritos con el estilo de *comportamiento* utilizando sentencias *if-then-else*. Es decir es posible escribir lógica combinacional tanto con sentencias concurrentes como con sentencias secuenciales. Las sentencias concurrentes son utilizadas en los estilos *flujo de datos* y *estructural*. Las sentencias secuenciales son utilizadas en descripciones por comportamiento.

3.3.1 Utilizando sentencias concurrentes indice

Las sentencias concurrentes siempre se localizan fuera de cualquier proceso. Conceptualmente éstas sentencias se ejecutan concurrentemente, por lo tanto el orden en que aparezcan las diferentes sentencias no es importante, a continuación vamos a describir tres tipos de sentencias concurrentes utilizadas en el estilo de descripción de flujo de datos.

Ecuaciones Booleanas.

Las ecuaciones booleanas pueden ser utilizadas en sentencias tanto concurrentes como secuenciales de asignamiento. El siguiente listado describe un multiplexor de cuatro buses de cuatro bits cada uno, como lo ilustra la Fig. 3.1.

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is port (
        a, b, c, d:     in std_logic_vector(3 downto 0);
        s       :       in std_logic_vector(1 downto 0);
        x       :       out std_logic_vector(3 downto 0);
end mux;

architecture archmux of mux is

begin
   x(3) <=      (a(3) and not(s(1)) and not(s(0)))
             or (b(3) and not(s(1)) and s(0))
             or (c(3) and s(1) and not s(0)))
             or (d(3) and s(1) and s(0));

   x(2) <=      (a(2) and not(s(1)) and not(s(0)))
             or (b(2) and not(s(1)) and s(0))
```

```
            or  (c(2)  and s(1) and not s(0)))
            or  (d(2)  and s(1) and s(0));

  x(1) <=       (a(1)  and not(s(1)) and not(s(0)))
            or  (b(1)  and not(s(1)) and s(0))
            or  (c(1)  and s(1) and not s(0)))
            or  (d(1)  and s(1) and s(0));

  x(0) <=       (a(0)  and not(s(1)) and not(s(0)))
            or  (b(0)  and not(s(1)) and s(0))
            or  (c(0)  and s(1) and not s(0)))
            or  (d(0)  and s(1) and s(0));
end archmux
```

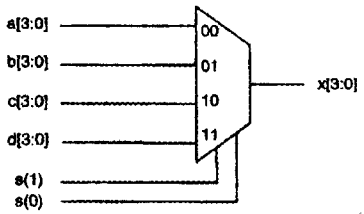


Fig. 3.1. Multiplexor cuádruple de cuatro a uno

Los operadores lógicos son la clave fundamental para el uso de ecuaciones booleanas. Los operadores lógicos "and, or, nand, xor, xnor" se encuentran predefinidos para los tipos *bit* y *bolean,* así como también para arreglos de una dimensión de esos mismos tipos. Cabe hacer notar que los dos operandos deben ser de la misma longitud en número de bits. Los operadores lógicos no tienen un orden de precedencia por tanto es necesario el uso de paréntesis para dar precedencia a los operadores.

with-select-when

Ésta sentencia permite una asignación selectiva de señales, esto significa que un valor es asignado a una señal con base en el valor de otra señal de selección. El constructor es el siguiente:

```
with señal_de_selección select
   nombre_de_la_señal <= valor_a when valor_1_de_la_señal_de_selección,
                         valor_b when valor_2_de_la_señal_de_selección,
                         valor_c when valor_3_de_la_señal_de_selección,
                                 .....

                         valor_x when valor_n_de_la_señal_de_selección;
```

A la señal "nombre_de_la_señal" se le asigna un valor con base en el valor actual de la señal "señal_de_selección". Todos los valores de la "señal_de_selección" deben ser listados y mutuamente excluyentes. A continuación se lista el código que representa el multiplexor de la Fig. 3.1.

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is port (
    a, b, c, d:    in std_logic_vector(3 downto 0);
    s          :   in std_logic_vector(1 downto 0);
    x          :   out std_logic_vector(3 downto 0);
end mux;

architecture archmux of  mux is
begin
    x <= a when "00",
         b when "01",
         c when "10",
         d when others;
end archmux;
```

El valor lógico "--" puede ser utilizado para asignar valores "no importa" a la señal "x" como sigue:

```
 architecture archmux of  mux is
begin
    x <= a when "00",
         b when "01",
         c when "10",
         d when "11",
         "--" when others;
end archmux;
```

when-else

Ésta sentencia ésta disponible para realizar asignamiento condicional de señales, lo cual significa que un valor es asignado a una señal con base en una condición. El constructor es el siguiente:

```
nombre_de_la_señal <= valor_a when condición1 else
                      valor_b when condición2 else
                      valor_c when condición3 else ...
                      valor_x;
```

A la señal "nombre_de_la_señal" se le asigna un valor con base en la evaluación de las condiciones. De ésta manera se le asigna el valor con base en la primera

condición que se evalúa a "verdadero". El siguiente listado muestra la descripción del multiplexor de la Fig. 3.1, utilizando éste constructor.

```
library ieee;
use ieee.std_logic_1164.all;
entity mux is port (
      a, b, c, d :    in std_logic_vector(3 downto 0);
      s          :    in std_logic_vector(1 downto 0);
      x          :    out std_logic_vector(3 downto 0);
end mux;

architecture  archmux of mux is
begin
    x <= a when (s = "00") else
         b when (s = "01") else
         c when (s = "10") else
         d;
end archmux;
```

Operadores Relacionales

Los operadores relacionales son utilizados para probar igualdad, desigualdad y ordenamiento. Los operadores de igualdad y desigualdad son "=" y "/=" los cuales son definidos para todos los tipos discutidos en éste manual. Los operadores de magnitud son " <, <=, >, >= " los cuales son definidos para tipos escalares o tipos arreglo con un rango discreto. Los resultados de los operadores relacionales es un valor de verdad "verdadero" o "falso". Como ejemplo ésta el siguiente código, donde se observa que en el caso de comparar arreglos, éstos deben ser de la misma longitud.

```
signal stream, instrm, oldstrm :  std_logic_vector(3 downto 0);
signal state : states;
signal we : std_logic;
signal id :   std_logic_vector(15 downto 0);
....

stream  <= "0000" when (state=idle and start='0') else
           "0001" when (state=idle and start='1') else
            instrm when (state=incoming) else
            oldstrm;

we <= '1' when (state=write and id < x"1FFF") else '0';
```

Sobrecarga de operadores

La sobrecarga de operadores permite utilizar los mismos operadores con múltiples tipos de datos, en especial para aquellos que no son predefinidos en el estándar

IEEE 1076. Los operadores pueden ser sobrecargados utilizando funciones definidas por el usuario, pero muchos de los operadores sobrecargados son ya definidos en los estándares IEEE 1164 y 1076.3.

Instanciación de Componentes.

La instanciación de componentes son sentencias concurrentes que especifican la interconexión de señales en el diseño. Como se ilustra en el siguiente código que implementa un comparador de cuatro bits. Esto nos sirve para ilustrar que la instanciación de componentes puede ser utilizada para implementar lógica combinacional.

```
library ieee;
use work.std_logic_1164.all;
entity compare is port (
        a, b     :   in std_logic_vector(3 downto 0);
        aeqb     :   out std_logic);
end compare;

use work.gatespkg.all;
architecture archcompare of compare is
   signal c : std_logic_vector(3 downto 0);
begin
   x0: xor2 port map(a(0), b(0), c(0));
   x1: xor2 port map(a(1), b(1), c(1));
   x2: xor2 port map(a(2), b(2), c(2));
   x3: xor2 port map(a(3), b(3), c(3));

   n1: nor4  port map(c(0), c(1), c(2), c(3), aeqb);
end;
```

Los componentes de las compuertas no son definidos por VHDL en su estándar, por tanto éste diseño requiere que las compuertas *xor2* y *nor2* sean definidas en otro paquete, ya sea creado por uno mismo o suministrado por el vendedor de la herramienta.

3.3.2 Usando sentencias secuenciales.

indice

Las sentencias secuenciales son todas aquellas contenidas en un proceso, una función o un procedimiento. Éstos dos últimos no son tratados en éste manual. La colección de sentencias que constituyen el proceso a su ves constituyen una sola sentencia concurrente. Si un diseño tiene múltiples procesos entonces éstos procesos son concurrentes entre si mismos. Sin embargo dentro del proceso, todas las sentencias son secuenciales y por tanto es importante el orden en que aparezcan. En ésta sección vamos a describir como utilizar procesos y sentencias secuenciales para describir lógica combinacional.

if-then-else

Éste constructor es utilizado para elegir un conjunto de sentencias a ser ejecutadas con base en la prueba de una condición que puede ser "verdadera" o "falsa". La sintaxis del constructor es la siguiente:

```
If (condición) then
    hacer_algo;
else
    hacer_algo_diferente;
end if;
```

Si la condición se evalúa a verdadero se ejecuta el primer bloque de sentencias "*hacer_algo*", de lo contrario se evalúa el segundo bloque. Como ejemplo mostramos el siguiente listado que representa un decodificador de dirección para el mapa de memoria de la Fig. 3.2.

```
library ieee;
use ieee.std_logic_1164.all;
entity decode is port(
        address   :        in std_logic_vector(15 downto 3);
        valid, boot-up :   in std_logic;
       sram, prom, eeprom, shadow,
       periphl, periph2 : out std-logic);
end decode;
architecture mem_decode of decode is
begin
   mapper: process  (address, valid, boot-up) begin
              shadow  <= '0';
              prom    <= '0';
              periph1 <= '0';
              periph2 <= '0';
              sram    <= '0';
              eeprom  <= '0';
              if valid = '1' then
                 if  address >= x"0000" and address < x"4000" then
                    if boot_up = '1'  then
                       shadow <= '1';
                    else
                       prom <= '1';
                    end if;
                 elsif address >= x"4000" and address < x"4008" then
                    periph1 <= '1';
                 elsif address >= x"4008" and address < x"4010" then
                    periph2 <= '1';
                 elsif address >= x"8000" and address < X"C000" then
```

```
            sram <= '1';
          elsif address >= X"C000" then
            eeprom <= '1';
          end if;
        end if;
      end process;
end mem-decode;
```

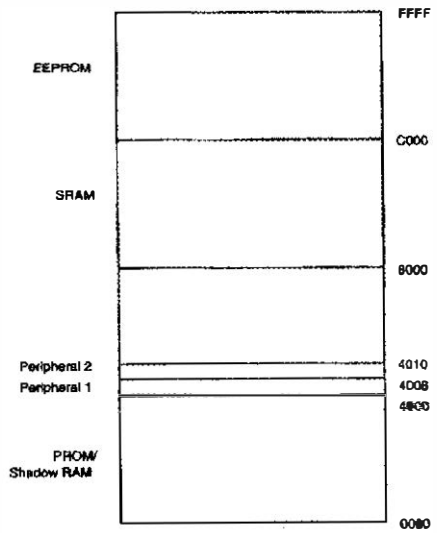


Fig. 3.2. Mapa de memoria

Éste diseño utiliza el constructor *if-then-else* y los operadores relacionales para identificar áreas de memoria y en consecuencia habilitar las señales correspondientes de acuerdo al mapa de memoria de la Fig. 3.2.

Cuándo se utiliza éste constructor para especificar lógica combinacional es muy importante especificar completamente la sentencia *if-then-else* o especificar el valor por default de la asignación esto para evitar generar un lazo de memoria no deseado. Observemos los siguientes tres fragmentos de código.

```
Signal step : std_logic;
Signal addr : std_logic_vector(7 downto 0);
....

similar1: process (addr)
  begin
    step <= '0';
    if addr > x"0F" then
      step <= '1';
    end if;
  end process;
```

```
similar2: process (addr)
  begin
    if addr > x"0F" then
      step <= '1'
    else
      step <= '0';
    end if;
  end process;

no_similar: process (addr)
  begin
    if addr > x"0F" then
      step <= '1';
    end if;
  end process;
```

Cualquiera de los procesos *similar1* y *similar2* asume el valor de '1' si "addr" es mayor de "0F" en hexadecimal y '0' en el otro caso. Sin embargo el proceso *no_similar* no describe la misma lógica debido a que no existe un valor por default. Esto da como resultado una implicación de ciclo de memoria no deseado como se ilustra en la Fig. 3.3.

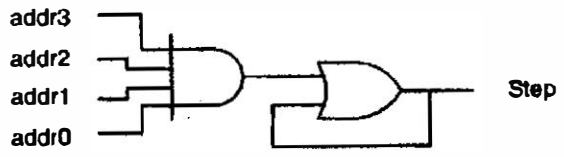


Fig. 3.3. Memoria implicada resultante del constructor if-then

case-when

Ésta sentencia es utilizada para especificar un conjunto de sentencias a ejecutarse con base en el valor de una señal de selección. La sintaxis es la siguiente:

```
Case señal_de_selección is
  when valor_1_de_la_señal_de_selección =>
       (hacer algo)   -- conjunto de sentencias 1
  when valor_2_de_la_señal_de_selección =>
       (hacer algo)   -- conjunto de sentencias 2
  when valor_3_de_la_señal_de_selección =>
       (hacer algo)   -- conjunto de sentencias 3
  .....

  when valor_último_de_la_señal_de_selección =>
       (hacer algo)   -- conjunto de sentencias n
```

El siguiente listado describe otro decodificador de dirección utilizando ésta sentencia.

```
library ieee;
use ieee.std_logic_1164.all;
entity test_case is
   port (address : in std_logic_vector(2 downto 0);
         decode  : out std_logic_vector(7 downto 0);
end test_case;

architecture design of test_case is
begin
   process (address)
   begin
      case address is
         when "001" =>  decode  <= x"11";
         when "111" =>  decode  <= x"42";
         when "010" =>  decode  <= x"44";
         when "101" =>  decode  <= x"88";
         when others => decode  <= x"00";
       end case;
    end process;
end design;
```

Ésta sentencia describe como el decodificador es manipulado con base en el valor de la dirección de entrada. La palabra reservada "others" se utiliza para definir completamente el comportamiento del decodificador para todos los posibles valores de entrada de la entrada "address".

3.4. Lógica síncrona*

indice

En ésta sección mostraremos como escribir código VHDL para modelar lógica síncrona, la cual esencialmente implica bloques de lógica combinacional conectada a elementos de memoria como flip-flops controlados por señales de reloj. El siguiente código describe un flip-flop tipo "D" ilustrado en la Fig. 3.4.

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_logic is port (
   d, clk     :    in std_logic;
   q          :    out std_logic);
end dff_logic;

architecture example of dff_logic is
begin
```

```
  process (clk) begin
    if (clk'event and clk = '1') then
      q <= d;
    end if;
  end process;
end example;
```

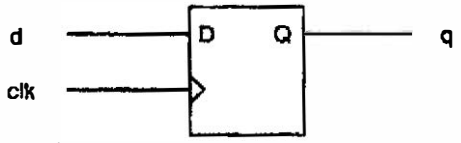


Fig. 3.4. Diagrama a bloques del Flip-flop D

Éste proceso es sensitivo únicamente a cambios en la señal "clk", esto significa que el proceso se ejecuta solamente cuándo existe una transición en la señal "clk". La condición *if clk'event* es verdadera sólo cuándo hay un cambio en la señal "clk", en otras palabras ésta condición detecta tanto el flanco de subida como el de bajada en la señal "clk". Debido a que éste cambio en la señal puede ser de '0' a '1' (flanco de subida) o de '1' a '0' (flanco de bajada), podemos adicionar una condición adicional para referirnos sólo a uno de los flancos; por ejemplo la condición *clk='1'* es utilizada para detectar solamente el flanco de subida. Las siguientes sentencias se utilizan para detectar una flanco de subida y después uno de bajada respectivamente:

```
If (clk'event and clk='1') then    --flanco de subida

If (clk'event and clk='0') then    --flanco de bajada
```

También podemos describir un latch sensitivo por nivel en lugar del flip-flop disparado por flanco. Lo que se tiene que hacer es detectar el evento de *clk* e insertar la señal de entrada "d" en la lista sensitiva, como se ilustra en el siguiente código que representa el latch de la Fig. 3.5.

```
process (clk,d)  begin
  if (clk='1') then
    q <= d;
  end if;
end process;
```

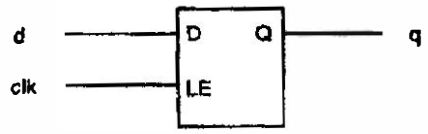


Fig. 3.5. Diagrama a bloques del latch

En éste caso cuándo *clk* está en nivel alto entonces se asigna la entrada "q" a la salida "d", describiendo un latch.

En el caso de la descripción del flip-flop o del latch, la especificación incompleta de la sentencia "if" al no incluir la parte "else", implica un elemento de memoria en el cual si la condición no se cumple se mantiene el valor de "q" lo cual es consistente con el funcionamiento deseado. Éste es el motivo por el cual los siguientes dos fragmentos de código son equivalentes:

```
if (clk'event and clk = '1') then
    q <= d;
end if;
```

es equivalente a:

```
if (clk'event and clk = '1') then
    q <= d;
else
    q <= q;
end if;
```

En el siguiente listado mostramos el código que implementa un flip-flop tipo "T" (toggle) y en la Fig. 3.6 el diagrama a bloques de una posible implementación. Note como es mucho más sencillo describirlo utilizando VHDL.

```
library ieee;
use ieee.std_logic_1164.all;
entity tff_logic is port (
  t, clk    : in std_logic;
  q         : buffer std_logic;
end dff_logic;

architecture t_example of dff_logic is
begin
  process (clk) begin
    if (clk'event and clk = '1') then
      if (t = '1') then
        q <= not(q);
      else
        q <= q;
      end if;
    end if;
  end process;
end t_example;
```

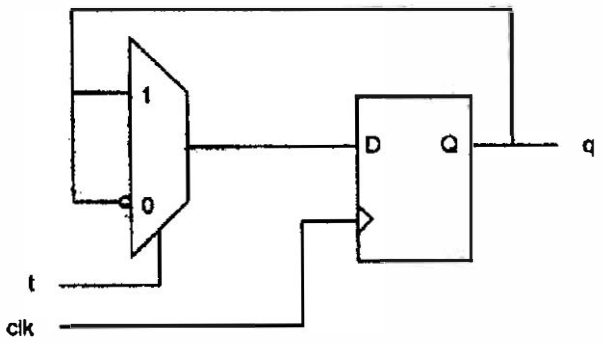


Fig. 3.6. Diagrama a bloques de un flip-flop T

Aquí es muy importante notar que todos los asignamientos de señales dentro del proceso ocurren después del evento en el flanco de subida de la señal *clk* por tal motivo todo está sincronizado con respecto a la señal *clk*. La Fig. 3.6 muestra que lógica secuencial es descrita por éste código. El siguiente código describe un registro de 8 bits.

```
library ieee;
use ieee.std_logic_1164.all;
entity reg_logic is port (
   d     : in std_logic_vector(0 to 7);
   clk   : in std_logic;
   q     : out std_logic_vector(0 to 7)
);
end reg_logic;

architecture r_example of reg_logic is
begin
   process (clk) begin
      if (clk'event and clk = '1') then
         q <= d;
      end if;
   end process;
end r_example;
```

Sentencia wait until

El comportamiento de un flip-flop o de un registro también es posible describirlo utilizando la sentencia *wait until* en lugar de la sentencia *if (clk'event and clk = '1')* como se ilustra en el siguiente código.

```
architecture example2 of dff_logic is
begin
   process begin
```

```
    wait until (clk = '1');
      q <= d;
    end process;
end example2;
```

Éste proceso no utiliza una lista sensitiva, sin embargo el proceso inicia con la sentencia *wait until*. Por ésta razón un proceso que utiliza ésta sentencia no debe tener lista sensitiva. La interpretación es que el proceso está suspendido hasta que la condición establecida por la sentencia *wait until* se hace "verdadera". Cuándo esto ocurre entonces las sentencias que aparecen después del *wait until* son ejecutadas, al terminar las sentencias se vuelve a esperar hasta que ocurra otro flanco de subida en la señal *clk*. De ésta manera existe una sincronía.

Funciones rising_edge y falling_edge.

El paquete *std_logic_1164* define dos funciones llamadas *rising_edge* y *falling_edge* que son útiles para detectar flancos de subida y de bajada, éstas se pueden utilizar para sustituir las expresiones *if (clk'event and clk = '1')* y *if (clk'event and clk = '1')* respectivamente. A continuación listamos el código que describe un flip-flop D utilizando la función *rising_edge*.

```
library ieee;
use ieee.std_logic_1164.all;
entity dff_logic is port (
  d, clk  : in std_logic;
  q       : out std_logic
end dff_logic;

architecture example of dff_logic is
begin
  process (clk) begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;
end example;
```

3.4.1 Empleo de señales de reset en lógica síncrona.

indice

En el mundo del hardware no siempre los circuitos se inicializan en un valor determinado. Entonces si uno desea tener señales de reset o preset globales es necesario incluirlas explícitamente en el código para poder manipularlas. A continuación mostramos el código que define un flip-flop D con reset asíncrono y que corresponde al diagrama a bloques de la Fig. 3.7. Éste reset es asíncrono porque el proceso se activa por un cambio en cualquiera de las señales *clk* o

reset. Si el *reset* es activado entonces independientemente de la señal *clk* el flip-flop es puesto a cero.

```
architecture rexample of dff_logic is
  begin
    process (clk, reset) begin
      if reset = '1' then
        q <= '0';
      elsif rising_edge(clk) then
        q <= d;
      end if;
    end process;
  end rexample;
```

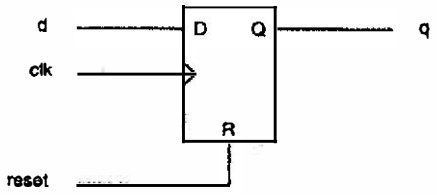


Fig. 3.7. Diagrama a bloques de un flip-flop D con reset asíncrono

Si lo que deseamos es tener un *reset* síncrono entonces debemos quitar ésta señal de la lista sensitiva y colocarla sólo dentro del proceso después de la detección del flanco de la señal *clk* para que el *reset* quede sincronizado con respecto a ésta señal, tal como se ilustra en el siguiente código. Su diagrama a bloques se muestra en la Fig. 3.8.

```
architecture sync_rexample of dff_logic is
  begin
    process (clk) begin
      if rising_edge(clk) then
        if reset = '1' then
          q <= '0';
        else
          q <= d;
        end if;
      end if;
    end process;
  end sync_rexample;
```

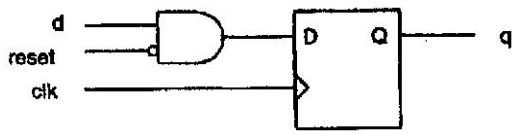


Fig. 3.8. Diagrama a bloques de un flip-flop D con reset síncrono

3.4.2 Operadores aritméticos. indice

Los operadores aritméticos más comúnmente utilizados para síntesis son la suma y la résta, los cuales son muy útiles para describir sumadores, réstadores, incrementadores, decrementadores, contadores etc. Todos los operadores aritméticos son predefinidos para los tipos entero y flotante. A continuación mostramos el código de un sumador de dos palabras de cuatro bits cada una.

```
entity myadd is port (
   a, b   : in integer range 0 to 3;
   sum  : out integer range 0 to 3);
end myadd;

architecture archmyadd of myadd is
begin
   sum <= a + b;
end archmyadd;
```

Aquí el resultado de la suma es asignado a un objeto del mismo tipo entero. Esto significa que para poder utilizar éstos operadores los tipos de los datos deben ser enteros y no del tipo *bit* o *bit_vector*. Si deseáramos sumar dos datos de tipo por ejemplo *std_logic_vector*, sería necesario sobrecargar los operadores aritméticos ya sea mediante funciones hechas por el usuario o dadas por la herramienta para poder sintetizar esas operaciones. En caso de no disponer de los operadores sobrecargados se deberá usar el tipo entero. El estándar IEEE 1076.3 define funciones para sobrecargar el operador "+" para pares de operandos de las siguientes características: (unsigned, unsigned), (unsigned, integer), (signed, signed), (signed, integer). Un ejemplo de como usarlos se ilustra en el siguiente código, note que la definición de los operadores sobrecargados están contenidas en el paquete *numeric_std*, el cual se hace visible con *use work.numeric_std.all*.

```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;
entity add_vec is port (
    a, b          :   in unsigned(3 downto 0);
    suml, sum2    :   out unsigned(3 downto 0);
end add_vec;
architecture adder  of add_vec is
begin
   suml <= a + b;
   sum2 <= c + 1;
end;
```

El par de códigos anteriores para la suma de dos números no toma en cuenta la posibilidad de existencia del acarreo resultado de la suma. Es decir si los números a sumar son de 4 bits cada uno el resultado puede llegar a requerir cinco bits. Para considerar éste bits de acarreo podemos escribir el siguiente código.

```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;
entity add_vec is port (
   a, b     :  in unsigned(3 downto 0);
   sum      :  out unsigned(4 downto 0);
end add_vec;

architecture adder of add_vec is
begin
   sum <= ('0' & a) + b;
end;
```

En éste caso uno de los operadores a sumar es *('0' & a)*. El operador "&" es el operador de concatenación; de manera que aquí el resultado es un vector no signado de un bit de longitud mayor que "a" al concatenarlo con '0'.

3.4.3 Resets y presets asíncronos.

indice

El siguiente listado describe un contador de ocho bits con una señal asícrona (grst), la cual coloca el valor del contador a "00111010". También el contador tiene dos señales síncronas de "load" y "enable". El operador "+" es sobrecargado y se utilizan tipos no signados.

```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;
entity cnt8 is port (
   txclk, grst     :    in std_logic;
   enable, load    :    in std_logic;
   data            :    in unsigned(7 downto 0);
   cnt             :    buffer unsigned(7 downto 0);
end cnt8;

architecture archcnt8 of cnt8 is
begin
   count : process (grst, txclk)
      begin
         if grst = '1' then
            cnt <= "00111010";
         elsif (txclk'event and txclk='1') then
```

```
            if load = '1' then
               cnt <= data;
            elsif enable = '1' then
               cnt <= cnt + 1;
            end if;
         end if;
      end process count;
end archcnt8;
```

El proceso es sensitivo a cambios en las señales *grst* y *txclk*. Si la señal *grst* es habilitada, entonces en forma asíncrona el valor del contador es puesto a "00111010". En forma sincronizada con el flanco de subida en la señal *txclk* el contador se carga con los datos de entrada si la señal *load* está habilitada y se incrementa si la señal *enable* ésta habilitada.

En el siguiente código mostramos como utilizar un par de señales asíncronas de reset y preset (grst y gpst). El secreto es hacer sensible al proceso para todas las señales que se desea sean asíncronas, además de la señal de reloj que sincroniza el resto del funcionamiento del contador como la carga y el incremento.

```
library ieee;
use ieee.std_logic_1164.all;
use work.numeric_std.all;
entity cnt8 is port (
   txclk, grst, gpst    :  in std_logic;
   enable, load         :  in std_logic;
   data                 :  in unsigned(7 downto 0);
   cnt                  :  buffer unsigned(7 downto 0);
end cnt8;

architecture archcnt8 of cnt8 is
begin
   count: process (grst, gpst, txclk)
      begin
         if grst = '1' then
            cnt <= (others => '0');
         elsif gpst = '1' then
            cnt <= (others => '1');
         elsif (txclk'event and txclk='1') then
            if load = '1' then
               cnt <= data;
            elsif enable = '1' then
               cnt <= cnt + 1;
            end if;
         end if;
      end process count;
end archcnt8;
```

La sentencia (others => '0') representa una agregación de elementos separados por comas, por tanto:

```
signal a: std_logic_vector(7 downto 0);
....
a <= ('1', '0', others => '1');
```

entrega el resultado para "a" de "10111111".

3.4.4 Buffers tres estados y señales bidireccionales. indice

La mayoría de los dispositivos de lógica programable tienen salidas tres éstados o bidireccionales. Los buffers de salida son puéstos en alta impedancia cuándo no manejan el bus en un tiempo determinado. Los puertos bidireccionales permite compartir un conjunto de puertos para funcionar tanto como entradas como salidas. Cabe hacer notar que no todos los dispositivos programables permiten la utilización de buses internos tres éstados, lo cual significa que en ellos no pueden ser implementados como buses internos, sino sólo como buses externos. Los diseños aquí presentados que utilizan éstas características están descritos utilizando el estilo de descripción por comportamiento.

Tres éstados.

Los valores que una señal tres éstados puede tener son '0', '1' y 'Z' (alta impedancia), todos ellos son soportados por el tipo *std_logic*. A continuación mostramos un contador de ocho bits que tiene salidas tres éstados.

```
library ieee;
use ieee.std_logic_1164. all;
use work.std_arith.all;
entity cnt8 is port (
   txclk, grst     :  in std_logic;
   enable, load :  in std_logic;
   oe              :  in std_logic;      --habilitador de los datos de salida
   data            :  in std_logic_vector(7 downto 0);
   cnt_out         :  buffer std_logic_vector(7 downto 0)); --salida de conteo
end cnt8;

architecture  archcnt8 of cnt8 is
  signal cnt : std_logic_vector(7 downto 0);  -- señal para el conteo
  begin
     count : process (grst, txclk)
        begin
           if grst = '1' then
```

```
            cnt <= "00111010";
        elsif rising_edge(txclk) then
            if load = '1' then
                cnt <= data;
            elsif enable = '1' then
                cnt <= cnt + 1;
            end if;
        end if;
    end process count;

    oes: process (oe, cnt)          --buffers tres éstados
    begin
        if oe = '0' then
            cnt_out <= (others => 'Z');
        else
            cnt_out <= cnt;
        end if;
    end process oes;
end archcnt8;
```

El proceso llamado "oes" es utilizado para describir las salidas tres éstados del contador. Éste proceso indica que si la señal "oe" es habilitada, entonces el valor de "cnt" que es de uso local a la arquitectura es asignado a "cnt_out" que es la salida del diseño; pero si "oe" no es habilitado entonces a la salida "cnt_out" se le asigna un valor de alta impedancia.

Bidireccional

Las señales bidireccionales son descritas con mucha facilidad, como se describe en el siguiente código. El valor del contador es cargado con el valor actual en los pines asociados con las salidas del contador (funcionan como entradas), o éstas salidas muestran el valor de la cuenta actual (funcionan como salidas) éste funcionamiento bidireccional depende del éstado del habilitador de salida "oe".

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity cnt8 is port (
    txclk, grst     :   in std_logic;
    enable, load :   in std_logic;
    oe              :   in std_logic;
    cnt_out         :   inout std_logic_vector(7 downto 0));  -- I/O bidireccional
end cnt8;

architecture archcnt8 of cnt8 is
   signal cnt : std_logic_vector(7 downto 0);
begin
```

```
  count: process (grst, txclk)
    begin
    if grst = '1' then
      cnt <= "00111010";
    elsif (txclk'event and txclk='1') then
      if load = '1' then
        cnt <= cnt_out;  -- cnt_out port funciona como entrada
      elsif enable = '1' then
        cnt <= cnt + 1;
      end if;
    end if;
  end process count;

  oes: process (oe, cnt) begin
    if oe = '0' then
cnt_out <= (others => 'Z');
    else
      cnt_out <= cnt; -- cnt_out funciona como salida
    end if;
  end process oes;
end archcnt8;
```

Los detalles importantes es que en éste código la señal *cnt_out* es de tipo *inout* debido a que va a mostrar y recibir datos.

3.5. Diseñando una FIFO

indice

Hasta éste punto hemos discutido como crear lógica combinacional y síncrona, de manera que éstamos listos para diseñar la FIFO introducida en la sección 3.2, y explicar algunos otros conceptos como el de creación de ciclos utilizando iteraciones. Recordando deseamos crear una estructura de datos FIFO de ocho palabras cada una de nueve bits. Cuándo la señal de lectura "rd" éste habilitada deseamos mostrar la salida "data_out" de la FIFO. Cuándo la señal de lectura "rd" no éste habilitada deseamos que la salida de la FIFO "data_out" se encuentre en éstado de alta impedancia. Cuándo habilitemos la señal de escritura "wr" deseamos escribir alguno de los ocho registros de nueve bits. Las señales "rdinc" y "wrinc" son utilizadas para incrementar los apuntadores de lectura y escritura los cuales apuntan al registro de la FIFO que será leído o escrito. "rdptrclr" y "wrptclr" hacen reset a los apuntadores de lectura y escritura respectivamente. "data_in" es el dato a ser almacenado en la FIFO. La Fig. 3.9 es un diagrama a bloques que representa el hardware descrito en el listado de la FIFO de la sección 3.2.

El primer nuevo concepto introducido en el listado de la FIFO es la declaración del tipo *fifo_array*, que es un constructor que utilizamos para crear un arreglo de ocho vectores *std_logic_vectors*, de nueve bits cada uno. A partir de éste tipo creamos una señal llamada "fifo" que es declarada de ese tipo y constituye los ocho

elementos de nueve bits cada uno. Esto nos permite acceder los elementos de la FIFO utilizando índices: fifo(6), fifo(5) etc.

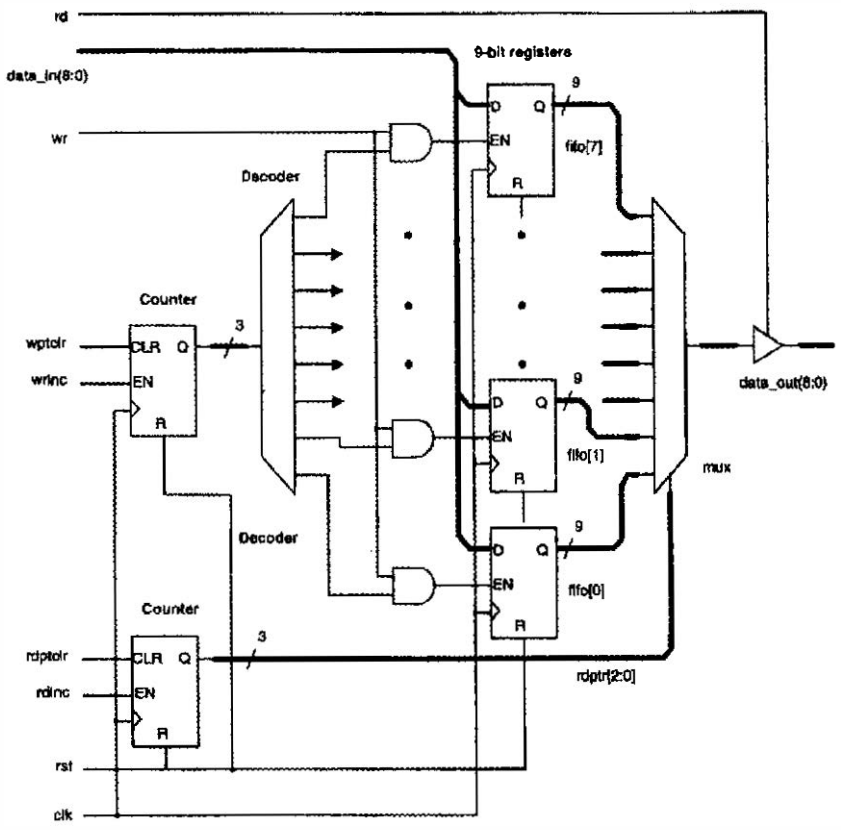


Fig. 3.9. Diagrama a bloques de la FIFO

3.5.1 Loops

indice

Las sentencias de loops (ciclos) son utilizadas para implementar operaciones repetitivas y consisten de ciclos "for" y "while". El ciclo for es ejecutado un número específico de iteraciones con base en un valor de control. Sin embargo el ciclo "while" se ejecuta continuamente hasta que una condición lógica lo termina al hacerse verdadera. El ciclo "for" es sintetizable por casi todas las herramientas mientras que el ciclo "while" no lo es.

En el diseño de la FIFO utilizamos el siguiente ciclo "for" para dar reset asíncrono a todo el arreglo FIFO.

```
for i in 7 downto 0 loop
    fifo(i) <= (others => '0');
end loop;
```

En los ciclos "for" la variable de conteo es automáticamente declarada. Un ciclo while puede ser utilizado, sin embargo se requiere declarar e inicializar las

variables del ciclo dentro del proceso, como se ilustra en el siguiente fragmento de código.

```
reg_array : process (rst, clk)
   variable i : integer := 0;
begin
   if rst = '1' then
      while  i < 8  loop
         fifo(i) <= (others => '0');
         i := i+1;
      end loop;
....
```

Iteraciones condicionales

La sentencia *next* es utilizada para saltar alguna operación con base en condiciones específicas. Por ejemplo supongamos que cuándo la señal de reset "rst" es habilitada todos los registros de la FIFO con borrados excepto el registro cuatro, entonces requerimos el siguiente fragmento de código:

```
reg_array : process (rst, clk)
begin
   if rst = '1' then
      for i in 7 downto 0 loop
        if i = 4 then
           next;
        else
           fifo(i) <= (others => '0');
      end loop;
...
```

Salida forzada de una iteración

La sentencia *exit* es utilizada para salir de un ciclo, y puede ser utilizada para verificar alguna condición ilegal. Esto debe verificarse en tiempo de compilación. Por ejemplo supongamos que la FIFO diseñada va a ser instanciada en algún otro diseño jerárquico y es necesario especificar su tamaño mediante algún parámetro; pero deseamos que no supere un tamaño máximo y cuándo eso pase termine el ciclo y genere un error. El siguiente código muestra éste caso.

```
reg_array : process (rst, clk)
   begin
      if rst = '1' then
         loop1: for i in deep downto 0 loop
                   if  i > 20 then
                      exit loop1;
```

```
        else
          fifo(i) <= (others => '0');
      end loop;
```

La etiqueta "loop1" es agregada sólo por claridad. En éstos casos el ciclo fue utilizado para una tarea muy sencilla, sin embargo pueden ser utilizados para tareas más complicadas. Por ejemplo el segundo ciclo en la FIFO verifica cual es el registro que va a ser escrito, como se observa en el código siguiente:

```
if wr = '1' then
    for i in 7 downto 0 loop
      if en(i) = '1' then
        fifo(i) <= data_in;
      else
        fifo(i) <= fifo(i);
      end if;
    end loop;
end if;
```

En nuestra FIFO no hay nada nuevo que no hemos visto. Los apuntadores de lectura y escritura son simples contadores de tres bits que indican cual registro de la FIFO será escrito o leído.

3. DISEÑO DE MÁQUINAS DE ESTADOS indice

En las secciones pasadas hemos analizado los constructores básicos del lenguaje VHDL. El objetivo en ésta sección es aplicarlos al diseño de máquinas de éstados. Es importante tratar éste tema porque las máquinas de éstado se utilizan e implementan muy comúnmente en dispositivos lógicos programables porque una máquina de éstados es una forma de representar un algoritmo o secuencia de pasos que se desea implantar en hardware. Las máquinas de éstados por tanto son muy útiles en el diseño de sistemas digitales dado que nos permite diseñar algoritmos de control para alguna arquitectura. Por ejemplo nos pueden servir para diseñar un controlador de memoria RAM.

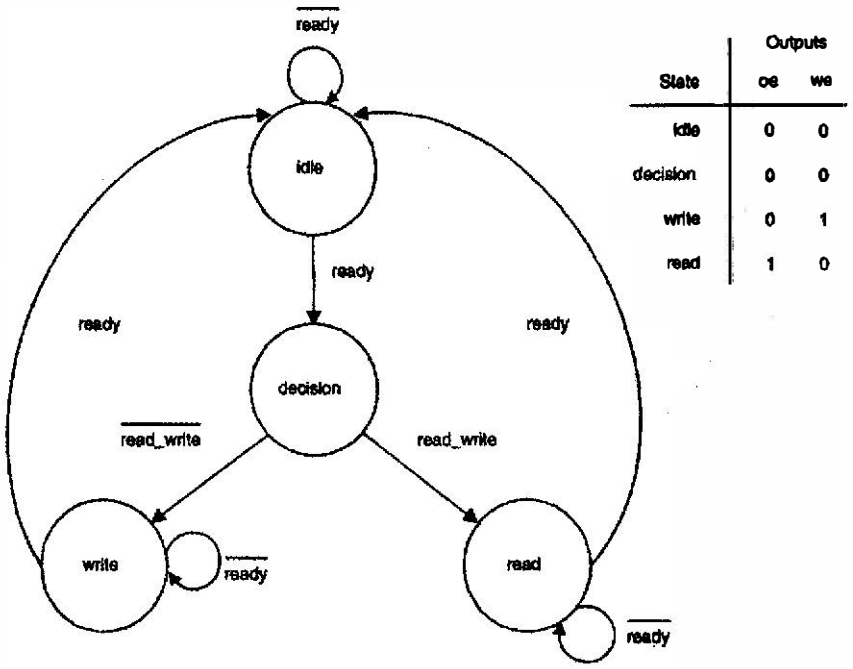
Nos enfocamos a implementar máquinas de éstado utilizando el estilo de descripción por comportamiento, debido a que hacerlo de ésta manera es tan sencillo como traducir el diagrama de éstados o carta ASM correspondiente.

Como es conocido las máquinas de éstado pueden ser de dos tipos el primero de ellos es el tipo MOORE donde las salidas son función únicamente del éstado presente y el segundo tipo son las máquinas tipo MEALY donde las salidas son función tanto del éstado presente como de las entradas. Aquí trataremos detalladamente las máquinas Moore y después indicaremos las mínimas modificaciones que son necesarias para implementar una máquina Mealy.

4.1 Ejemplo de diseño indice

Vamos a diseñar un controlador que es usado para habilitar y deshabilitar el habilitador de escritura "we" y el habilitador de salida "oe" de un buffer de memoria durante peticiones de lectura/escritura. Las señales "ready" y "read_write" son salidas de un microprocesador y entradas a nuestro controlador. "we" y "oe" son salidas del controlador. Una nueva petición comienza con la habilitación de la señal "ready". Un ciclo de reloj después de iniciada la petición el valor de "read_write" si la petición es de lectura o escritura. Si "read_write" es habilitada entonces se trata de un ciclo de lectura; de otro modo se trata de un ciclo de escritura. Un ciclo es completado habilitando la señal de "ready" después de la cual una nueva petición puede ser aceptada.

En la Fig. 4.1 mostramos el diagrama de éstados que se obtiene a partir de la descripción del problema. Éste diagrama de éstados no representa más que una máquina de éstados del tipo Moore, porque como podemos observar en la tabla adjunta las salidas "oe" y "we" son función únicamente del éstado presente.



State	Outputs oe	we
idle	0	0
decision	0	0
write	0	1
read	1	0

Fig. 4.1. Diagrama de éstados para el controlador de memoria

4.2 Máquina de éstados Moore en VHDL indice

El diagrama de éstados mostrado en la Fig. 4.1 puede ser fácilmente traducido a descripción de alto nivel usando VHDL sin tener que hacer asignación de éstados, generación de la tabla de transición o determinación de las ecuaciones del éstado siguiente con base en el tipo de flip-flop disponible. Utilizando VHDL cada éstado puede ser traducido a un caso específico utilizando el constructor ***case-when***. Las transiciones del diagrama de éstados pueden ser especificadas utilizando sentencias *if-then-else*. Existen fundamentalmente dos alternativas para describir una máquina de éstados, en función del número de procesos utilizados. Así es posible describir la misma máquina de éstados utilizando dos procesos o utilizando un sólo proceso como se verá a continuación.

Máquina de éstados utilizando dos procesos

En éste método uno de los procesos indica que la siguiente asignación de éstado se hará con base en el presente éstado y en las entradas presentes, pero no indica cuándo el siguiente éstado llega a ser el presente éstado. Esto sucede en forma síncrona con la señal de reloj de la forma en que se describe en un segundo proceso. Debido a éstos dos procesos es que recibe el nombre de máquina de éstados utilizando dos procesos. El código que describe ésta máquina de éstados es el siguiente:

```
library ieee;
use ieee.std_logic_1164.all;

entity fsm2p is port(
  read_write, ready, clk    : in bit;
  oe, we                    : out bit);
end fsm2p;

architecture state_machine of fsm2p is
  type statetype is (idle, decision, read, write);
  signal present_state, next_state :statetype;
begin
  process(present_state, read_write, ready) begin
    case present_state is
      when idle =>
        oe <= '0'; we <= '0';
        if ready = '1' then
          next_state <= decision;
        else
          next_state <= idle;
        end if;
      when decision =>
        oe <= '0'; we <= '0';
        if read_write = '1' then
          next_state <= read;
        else
          next_state <= write;
        end if;
      when read =>
        oe <= '1'; we <= '0';
        if ready = '1' then
          next_state <= idle;
        else
          next_state <= read;
        end if;
      when write =>
        oe <= '0'; we <= '1';
        if ready = '1' then
          next_state <= idle;
        else
          next_state <= write;
        end if;
    end case;
```

```
  end process;
    process(clk) begin
      if (clk'event and clk='1') then
        present_state <= next_state;
      end if;
  end process;
end architecture state_machine;
```

Al inicio de la arquitectura definimos un tipo llamado "statetype" que es un enumerado de cuatro valores "idle, decision, read, write" los cuales son nombres que nos representan cada uno de los cuatro éstados disponibles en la máquina de éstados del la Fig. 4.1. Después definimos dos señales de uso local a la arquitectura llamadas "present_state y next_state" que son del tipo que acabamos de crear y que nos van a servir para representar el éstado actual y el éstado siguiente.

El primer proceso es sensible al éstado presente (present_state) así como a las señales de lectura escritura (read_write) y listo (ready). Éste proceso lo que hace es probar utilizando la sentencia *case* el valor del "present_state", el número de casos corresponde con el número de éstados, por tanto el constructor *case* tendrá tantas opciones como número de éstados tenga la máquina de éstados.

En cada caso contemplado por el *case* dos son las acciones a realizar. La primera consiste en éstablecer el valor de las salidas "oe" y "we" para ese éstado, debido a que se trata de una máquina de tipo Moore, éstos valores son directamente tomados del diagrama de éstados de la Fig. 4.1. La segunda acción para cada caso consiste en definir las transiciones del éstado actual hacia el éstado siguiente, en otras palabras calculamos cual es el siguiente éstado en función de las entradas que sean importantes en el éstado actual (ready o read_write); ésta prueba se hace utilizando el constructor *if-else* y anidándolos en caso de tener que probar varias entradas. De manera que el resultado de ésta prueba es asignar a "next_state" el éstado siguiente que puede ser alguno de los cuatro éstados posibles definidos por "statetype" según corresponda.

El segundo proceso es sensible a la señal de reloj "clk" y sincronizado con respecto al flanco de subida de la misma señal de reloj. El objetivo de éste éstado es actualizar el valor de la señal del éstado presente "present_state" con el valor de la señal del éstado siguiente "next_state", que fue calculado en el primer proceso.

Esto significa que la transición del éstado presente al éstado siguiente está sincronizada con respecto a la señal de reloj del sistema y aunque se calcula en el proceso uno, no se asigna sino sincronizadamente con la señal de reloj en el proceso dos. Las salidas como son función del éstado presente son actualizadas continuamente en el proceso uno en función del éstado presente solamente debido a que se trata de una máquina Moore.

El funcionamiento de la máquina de éstados puede verificarse rápidamente utilizando cualquier simulador que acepte entrada VHDL. El código mostrado para ésta máquina fue simulado utilizando el simulador de la herramienta Max+Plus II de Altera. El resultado de la simulación se muestra en la Fig. 4.2. Donde podemos observar los valores de los éstados presente y siguiente desfasados un ciclo de reloj, así como el valor de las salidas de acuerdo al éstado presente. La transición al éstado siguiente es función del éstado actual así como de las entradas. El funcionamiento está sincronizado con respecto al flanco de subida de la señal de reloj.

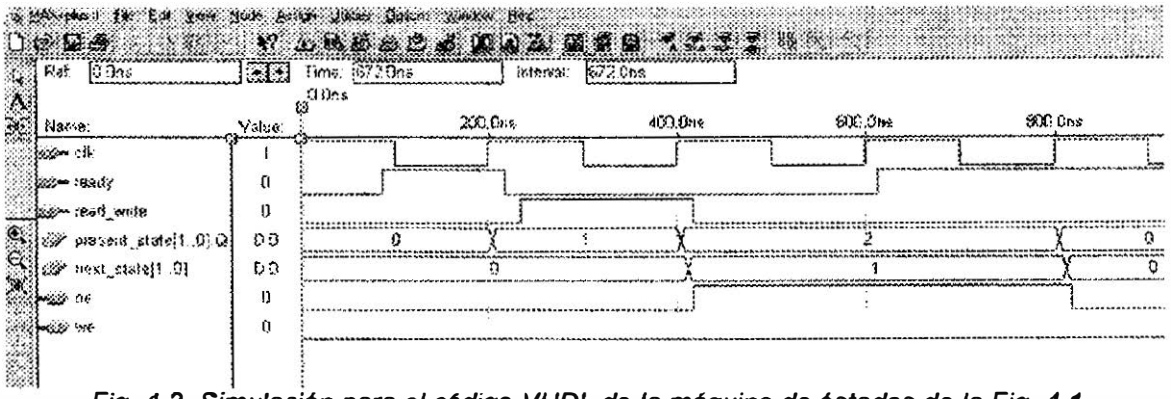


Fig. 4.2. Simulación para el código VHDL de la máquina de éstados de la Fig. 4.1

4.2.1 Controlador de memoria utilizando Máquina Moore de dos procesos.

indice

El siguiente ejemplo de diseño es también un controlador de memoria, pero es más práctico y tiene mayor funcionalidad que el de la sección anterior. La Fig. 4.3 muestra el diagrama a bloques de un sistema que utiliza una máquina de éstados para el controlador de memoria.

El controlador trabaja de la siguiente manera: Cualquier dispositivo en el bus inicializa un acceso al buffer de memoria habilitando su identificación de bus, en éste caso F3 en hexadecimal. Un ciclo después la señal "read_write" es habilitada para indicar una lectura desde el buffer de memoria, la señal es deshabilitada para indicar un ciclo de escritura al buffer de memoria. Si el acceso a la memoria es de lectura, la lectura puede ser lectura de una simple palabra o una lectura *burst* de cuatro palabras. Éste tipo de lectura burst se indica habilitando la señal "burst" durante el primer ciclo de lectura, después del cual el controlador accede cuatro localidades del buffer. Localidades consecutivas son accedidas siguiendo sucesivas habilitaciones de la señal "ready". El controlador habilita "oe" (habilitador de salida) hacia la memoria durante el ciclo de lectura, e incrementa los dos bits menos significativos de la dirección en el modo burst. Una escritura al buffer es siempre una escritura de una sola palabra, nunca en modo burst. Durante un ciclo de escritura la señal "we" es habilitada, permitiendo a la señal "data" ser escrita en la localidad de memoria especificada por la señal "address".

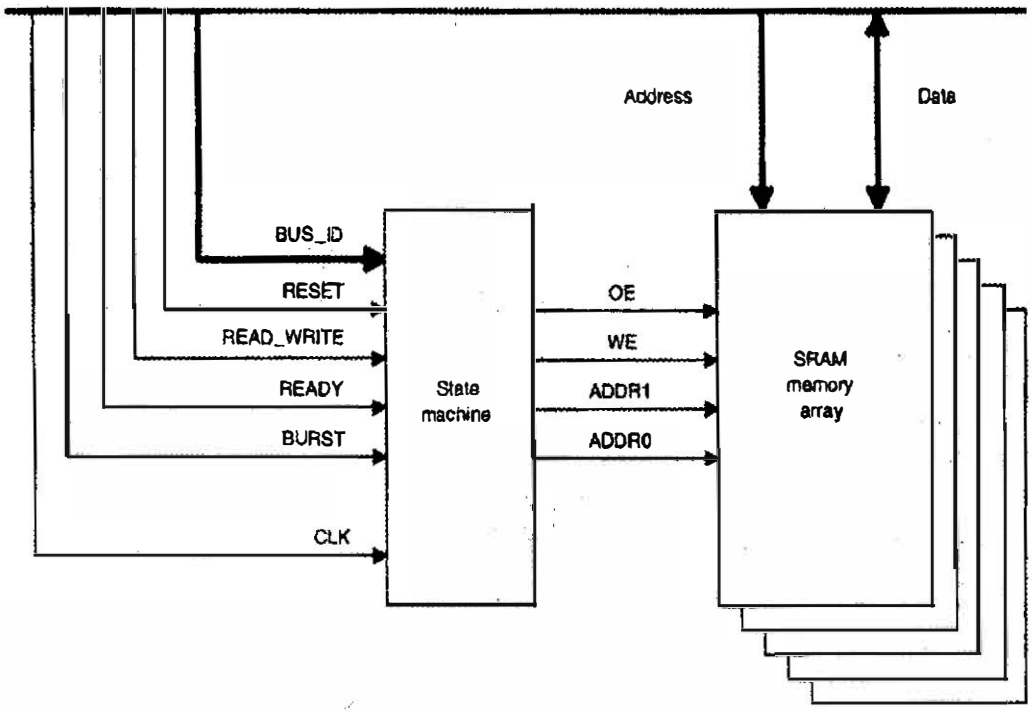
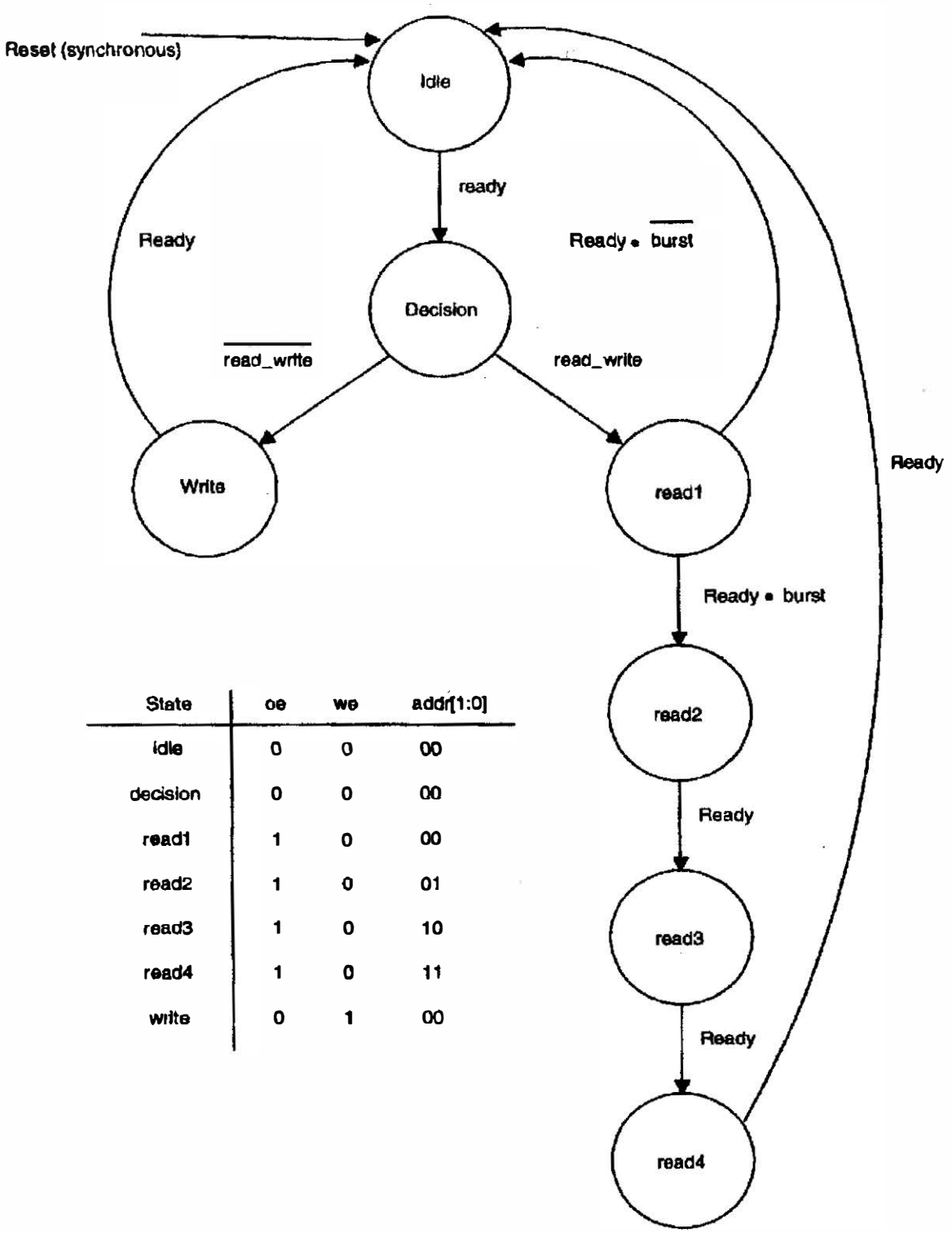


Fig. 4.3. Diagrama a bloques del controlador de memoria.

La Fig. 4.4 ilustra el diagrama de éstados de éste controlador. El diagrama muestra como una señal de "reset" síncrona pone el controlador en el éstado "idle". Cuándo el buffer de memoria no va a ser accedido, el controlador se mantiene en el éstado "idle". Si el identificador de bus "bus_id" es habilitado a F3 mientras el controlador está en el éstado "idle" entonces la máquina de éstados hace transición al éstado "decision". En el siguiente ciclo de reloj, el controlador hará transición a cualquiera de dos éstados "read1 ò write" dependiendo del valor de la señal "read_write". Si el acceso es para lectura, entonces el controlador salta a la sección de lectura de la máquina de éstados. Una simple palabra para lectura es indicada habilitando "ready" sin la habilitación de "burst" mientras se está en el éstado "read1". En éste caso el controlador regresa al éstado "idle". Una lectura "burst" es indicada habilitando tanto la señal "ready" como "burst" mientras se está en el éstado "read1". En éste caso la máquina transita a través de cada uno de los éstados de lectura cada que se habilita "ready". La señal "oe" es habilitada durante cada uno de los ciclos de reloj de lectura. La señal "addr" es incrementada en ciclos sucesivos de lectura siguiendo al primero.

Si el acceso es para escritura, solamente se puede escribir una sola palabra. Por lo tanto después de determinar que el acceso es para escritura (read_write = 0) éstando en el éstado "decision", el controlador salta a la sección de escritura. Simplemente habilita la señal de "we" del buffer de memoria, esperando por la señal de "ready" del bus y retornando al éstado "idle.



State	oe	we	addr[1:0]
idle	0	0	00
decision	0	0	00
read1	1	0	00
read2	1	0	01
read3	1	0	10
read4	1	0	11
write	0	1	00

Fig. 4.4. Máquina de éstados del controlador de memoria.

La forma de traducir a lenguaje VHDL la máquina de éstados de la Fig. 4.4, se hace de la misma manera que con la máquina de éstados de la Fig. 4.1. El código completo utilizando dos procesos y un reset síncrono se muestra en el siguiente código:

```
library ieee;
use ieee.std_logic_1164.all;

entity fsm2psr is port(
  reset, read_write, ready, clk, burst    : in std_logic;
  bus_id                                  : in std_logic_vector(7 downto 0);
  oe, we                                  : out std_logic;
  addr                                    : out std_logic_vector(1 downto 0));
end fsm2psr;

architecture state_machine of fsm2psr is
  type statetype is (idle, decision, read1, read2, read3, read4, write);
  signal present_state, next_state :statetype;
begin
  process(reset, bus_id, present_state, burst, read_write, ready) begin
    if (reset = '1') then
      oe <= '-'; we <= '-'; addr <= "--";  --no importa
      next_state <= idle;
    else
    case present_state is
      when idle =>
          oe <= '0'; we <= '0'; addr <= "00";
          if bus_id = "11110011" then   -- identificador F3
            next_state <= decision;
          else
            next_state <= idle;
          end if;
      when decision =>
          oe <= '0'; we <= '0'; addr <= "00";
          if read_write = '1' then
            next_state <= read1;
          else
            next_state <= write;
          end if;
      when read1 =>
          oe <= '1'; we <= '0'; addr <= "00";
          if ready = '0' then
            next_state <= read1;
          else if burst = '0' then
            next_state <= idle;
          else
```

```
            next_state <= read2;
          end if;
        end if;
      when read2 =>
          oe <= '1'; we <= '0'; addr <= "01";
          if ready = '1' then
            next_state <= read3;
          else
            next_state <= read2;
          end if;
      when read3 =>
          oe <= '1'; we <= '0'; addr <= "10";
          if ready = '1' then
            next_state <= read4;
          else
            next_state <= read3;
          end if;
      when read4 =>
          oe <= '1'; we <= '0'; addr <= "11";
          if ready = '1' then
            next_state <= idle;
          else
            next_state <= read4;
          end if;
      when write =>
          oe <= '0'; we <= '1'; addr <= "00";
          if ready = '1' then
            next_state <= idle;
          else
            next_state <= write;
          end if;
    end case;
    end if;
  end process;

  process(clk) begin
    if (clk'event and clk='1') then
      present_state <= next_state;
    end if;
  end process;
end architecture state_machine;
```

El código del listado anterior es una descripción de máquina de éstados de dos procesos. Uno de los procesos describe la lógica combinacional del valor de las salidas y las transiciones utilizando constructores *case-when, if-theh-else* y el segundo proceso describe la sincronización de las transiciones de los éstados en

relación a la señal de reloj "clk". Ésta forma de implementación se muestra en diagrama a bloques en la Fig. 4.5 y se puede utilizar para cualquier máquina de éstados tipo Moore.

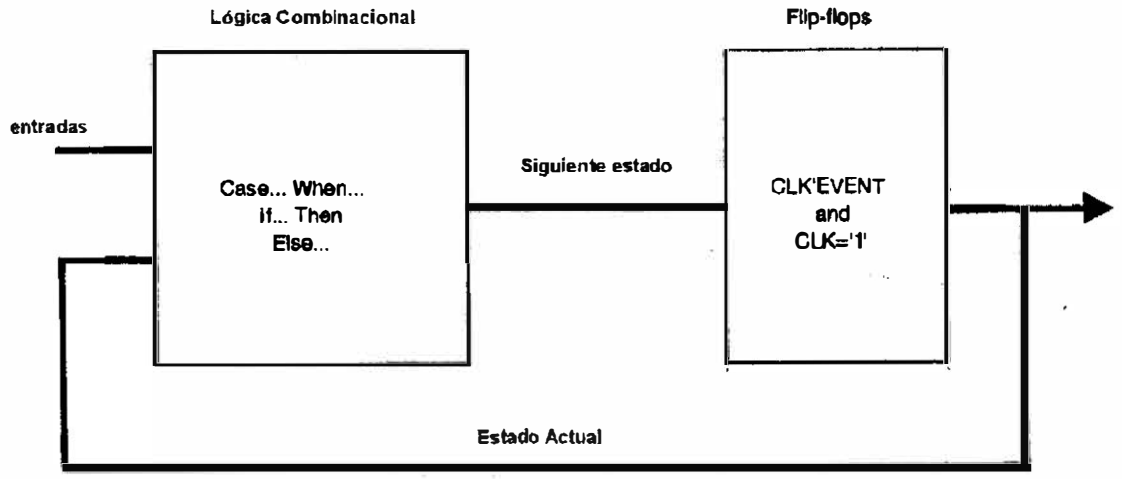


Fig. 4.5. Esquema general de la máquina de éstados Moore.

Si deseamos un reset asíncrono en lugar del reset síncrono utilizado en la sección anterior, solamente requerimos hacer sensible el proceso que sincroniza las transiciones no solamente a la señal de reloj "clk" sino también a la señal de reset, como se ilustra en el siguiente fragmento de código.

```
process(clk, reset) begin
  if (reset = '1') then
    present_state <= idle;
  else if(clk'event and clk='1') then
    present_state <= next_state;
  end if;
  end if;
end process;
```

4.2.2 Controlador de memoria utilizando Máquina Moore de un proceso.

indice

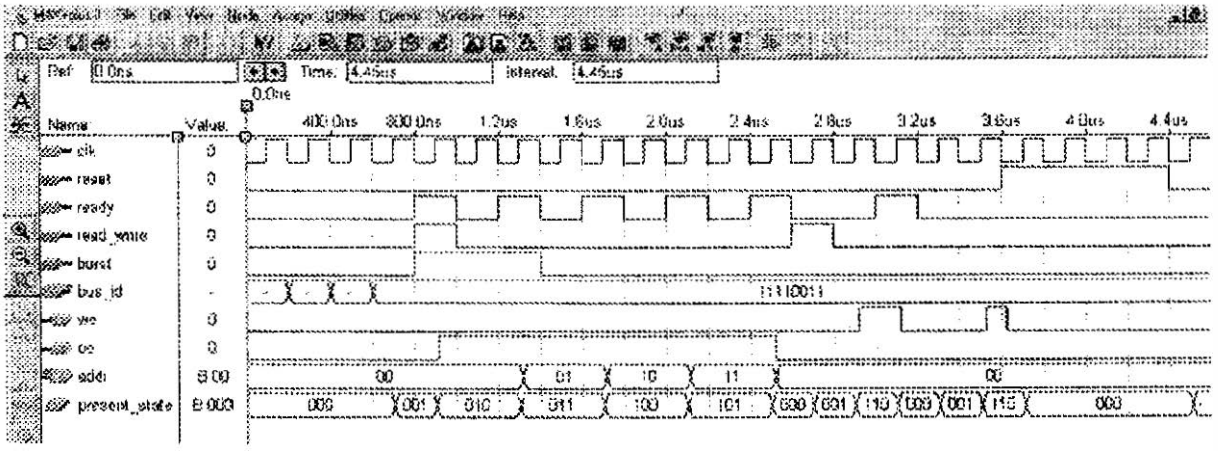
El mismo controlador de memoria puede ser implementado en VHDL utilizando un sólo proceso que se utiliza para describir las transiciones entre los éstados y la sincronía de las mismas con relación a la señal de reloj "clk". Sin embargo las salidas de la máquina de éstados son decodificadas utilizando lógica combinacional en función de los éstados donde se deben verificar. Por supuesto ésta decodificación se hace fuera del proceso. Éste es el motivo por el cual se llama máquina de un proceso. El funcionamiento es equivalente al de una máquina de dos procesos, sólo que descrita de diferente manera. A continuación

se muestra el código completo de la máquina de un proceso para el controlador de memoria.

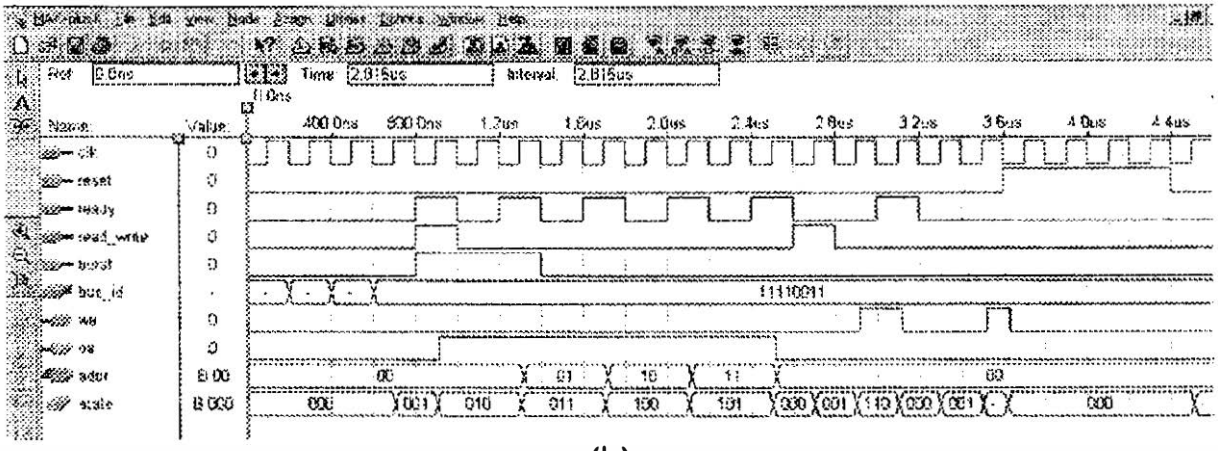
```
library ieee;
use ieee.std_logic_1164.all;

entity fsm1par is port(
  reset, read_write, ready, clk, burst  : in std_logic;
  bus_id                                : in std_logic_vector(7 downto 0);
  oe, we                                : out std_logic;
  addr                                  :  out std_logic_vector(1 downto 0));
end fsm1par;

architecture state_machine of fsm1par is
  type statetype is (idle, decision, read1, read2, read3, read4, write);
  signal state   : statetype;
begin
  process(reset, clk) begin  --reset asincrono
    if (reset = '1') then
      state <= idle;
    else if (clk'event and clk='1') then
    case state is
      when idle =>
          if bus_id = "11110011" then
            state <= decision;
          else
            state <= idle;
          end if;
      when decision =>
          if read_write = '1' then
            state <= read1;
          else
            state <= write;
          end if;
      when read1 =>
          if ready = '0' then
            state <= read1;
          else if burst = '0' then
            state <= idle;
          else
            state <= read2;
          end if;
          end if;
      when read2 =>
          if ready = '1' then
            state <= read3;
          end if;       -- state <= read2 ésta implícito
```

```
    when read3 =>
        if ready = '1' then
          state <= read4;
        else
          state <= read3;  -- no se requiere else, puede ser implícito
        end if;
    when read4 =>
        if ready = '1' then
          state <= idle;
        else
          state <= read4;
        end if;
    when write =>
        if ready = '1' then
          state <= idle;
        else
          state <= write;
        end if;
  end case;
  end if;
  end if;
end process;

-- Salidas decodificadas combinacionalmente

 with state select
    oe <= '1' when read1 | read2 | read3 | read4,
          '0' when others;
    we <= '1' when state = write else '0';

 with state select
    addr <= "01" when read2,
            "10" when read3,
            "11" when read4,
            "00" when others;

end architecture state_machine;
```

Observamos claramente como las salidas "oe", "we" y "addr" se decodifican combinacionalmente fuera del proceso utilizando sentencias *with-select*, ésta decodificación se obtiene directamente de la tabla adjunta en la Fig. 4.4. En la Fig. 4.6 mostramos la simulación en (a) del controlador utilizando dos procesos y en (b) la simulación del controlador utilizando un proceso. Podemos observar que las simulaciones son equivalentes y funcionan como lo indica la máquina de éstados ilustrada en la Fig. 4.4.



(a)



(b)

Fig. 4.6. Simulación del controlador de memoria usando (a) dos procesos y (b) un proceso.

4.3 Máquina de éstados Mealy en VHDL

indice

Hasta aquí hemos estudiado únicamente máquinas tipo Moore, en las cuales las salidas son estrictamente función del éstado actual. En las máquinas tipo Mealy podemos tener salidas que son función del éstado presente así como de las señales de entrada actuales, ésta idea y diferencia entre máquinas Moore y Mealy se ilustra en la Fig. 4.7.

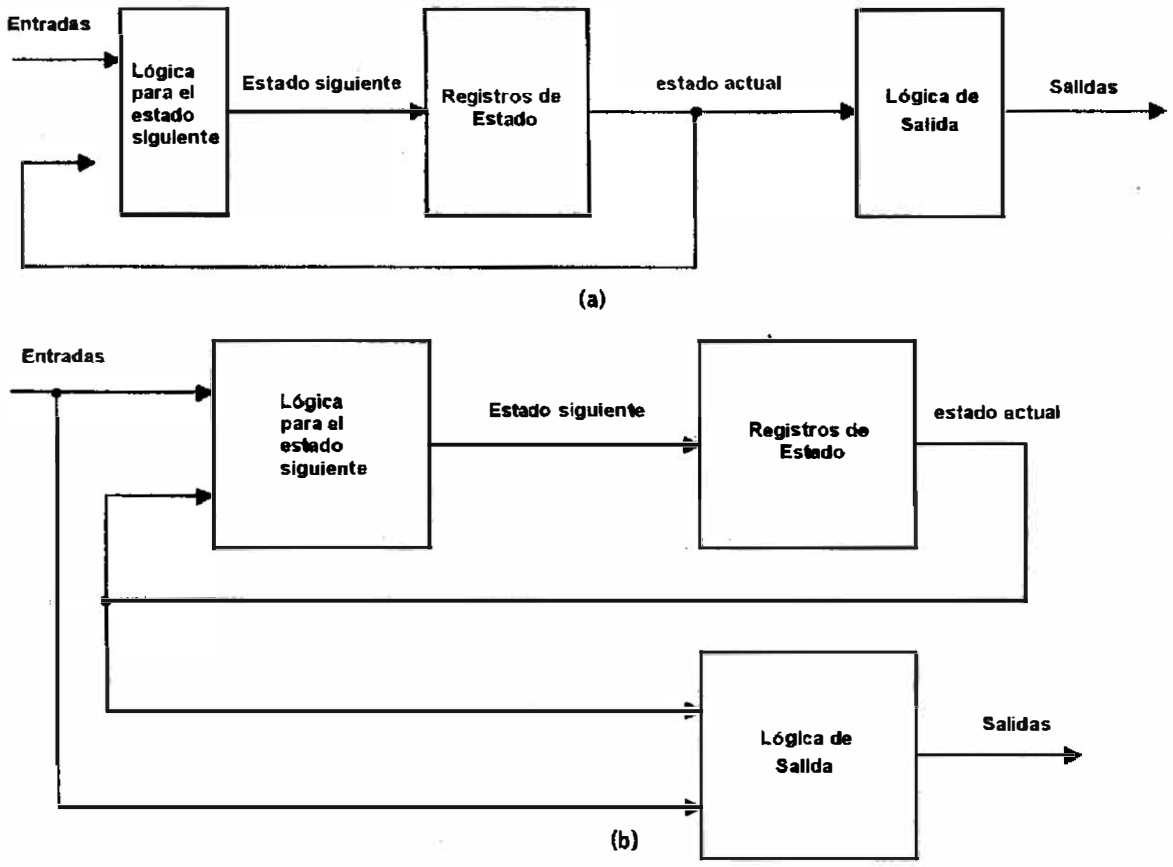


Fig. 4.7. (a) Máquina Moore y (b) Máquina Mealy.

El trabajo adicional requerido para describir una máquina Mealy en contraste con una máquina Moore es absolutamente mínimo. Para implementar una máquina Mealy simplemente se tiene que describir la salida como una función tanto de los bits de éstado como de las entradas que las afecten. Por ejemplo, si tuviéramos una entrada adicional al controlador de memoria llamada "write_mask" la cual cuándo se habilite prevenga que la escritura "we" sea habilitada. Esa nueva condición implica que la salida "we" no solamente depende del éstado en que se habilita que es el éstado seis (write), sino que también depende de la nueva entrada "write_mask". Ésta condición se puede reflejar con el siguiente código:

```
if (present_state = write) and (write_mask = '0') then
   we <= '1';
else
   we <= '0';
end if;
```

Ésta ligera modificación hace que la salida "we" sea una salida Mealy, lo mismo puede hacerse para cualquier otra salida que lo requiera.

4. BIBLIOGRAFÍA

indice

a) *VHDL for Programmable Logic*. K. Skahill. Addison-Wesley. First Edition. 1996.
b) *The VHDL Reference*. U. Heinkel, et al. John Wiley & Sons. First Edition. 2000.
c) *A VHDL Primer*. J. Bhasker. Prentice Hall. Third Edition. 1999.
d) *Fundamentals of Digital Logic with VHDL Design*. S. Brown, Z. Vranesic. McGraw-Hill. First Edition. 2000.
e) *Digital Systems Design with VHDL and Synthesis*. K.C. Chang. IEEE Computer Society. First Edition. 1999.
f) *Logic and Computer Design Fundamentals*. M. Morris Mano, Charles R. Kime. Prentice Hall. 1999.
g) *Modern Digital Systems Design*. John Y. Cheung. Ed. West. 1991.

Universidad Nacional
Autónoma de México

Facultad
de
Ingeniería