UNIVERSIDAD AUTONOMA DE SAN LUIS POTOSI

FACULTAD DE CIENCIAS

DESARROLLO DE SISTEMAS DIGITALES CON VHDL

TESIS PROFESIONAL

para obtener el título de

INGENIERO ELECTRONICO

PRESENTA:

JORGE GALVAN CORPUS

SAN LUIS POTOSI, S. L. P. OCTUBRE DE 2004

UNIVERSIDAD AUTONOMA DE SAN LUIS POTOSI

FACULTAD DE CIENCIAS

DESARROLLO DE SISTEMAS DIGITALES CON VHDL

TESIS PROFESIONAL

para obtener el título de

INGENIERO ELECTRONICO

PRESENTA:

JORGE GALVAN CORPUS

ASESOR DE TESIS:

DR. JOEL URIEL CISNEROS PARRA

SAN LUIS POTOSI, S. L. P. OCTUBRE DE 2004

DEDICATORIA

A mis padres, Ma. De los Angeles Corpus Hernández y Rafael Galván Ramírez, por todo el amor, compañía y consejos que me han dado desde siempre.

AGRADECIEMIENTOS

Al Dr. Joel U. Cisneros Parra por su asesoría y por todos los conocimientos transmitidos.

A mis amigos Calvillo, Pineda, Stevens, Dr. Balderas, Rosali, Angel, Josué, Javier, Daniel y Hugo, por su apoyo y motivación.

ÍNDICE

1. INTRODUCCIÓN

Una de las técnicas de desarrollo de sistemas electrónicos digitales que han revolucionado esta área, es la utilización de dispositivos de lógica reconfigurable. Con esta tecnología, mediante programación, es posible describir el funcionamiento que tendrá el dispositivo lógico.

La descripción del funcionamiento se puede realizar en forma gráfica, mediante un diagrama de conexión de componentes, o bien, mediante un lenguaje descriptor de hardware.

El avance tecnológico logrado en estos dispositivos ha permitido el diseño de sistemas electrónicos cada vez más complejos, por lo que el uso del método gráfico no es muy recomendable. Esto se debe a que es complicada la interconexión manual de grandes cantidades de componentes, por lo que su uso es poco práctico para fines de análisis y desarrollo.

Por otro lado, tenemos la opción de usar un lenguaje descriptor de hardware, de manera que podemos expresar el funcionamiento del diseño en forma de texto.

El VHDL (VHSIC *Hardware Description Language* y VHSIC *Very High Speed Integrated Circuit*) es el lenguaje de mayor uso en la descripción de componentes electrónicos. Es un lenguaje estandarizado, independiente de los sistemas de diseño asistido por computadora (CAD) y ofrece grandes ventajas en el desarrollo de sistemas digitales.

La utilización conjunta del VHDL con algún sistema de CAD permiten obtener como resultado final nuestro prototipo funcional. Las herramientas de CAD efectúan una serie de procesos en el que destaca uno muy importante llamado síntesis, el cual se encarga de traducir nuestro diseño de VHDL a un diagrama lógico equivalente que se utiliza para programar el dispositivo lógico.

Mediante el VHD elaboramos nuestros diseños, de manera que describimos su funcionamiento a través de instrucciones que equivalen al comportamiento de un componente electrónico. Podemos describir el funcionamiento en diferentes niveles de abstracción del lenguaje, es decir, utilizar un nivel alto significa hacer uso de las instrucciones especiales del

VHDL y un nivel bajo significa utilizar expresiones básicas de lógica combinacional.

Una vez que se tiene el modelo del sistema descrito en VHDL y se realiza el proceso de síntesis, podemos verificar su funcionamiento mediante una simulación; de esta manera nos aseguramos que el funcionamiento es el deseado y proseguimos al siguiente paso, que sería grabar el diseño en un dispositivo de lógica reconfigurable determinado. (Ver Fig. 1.)

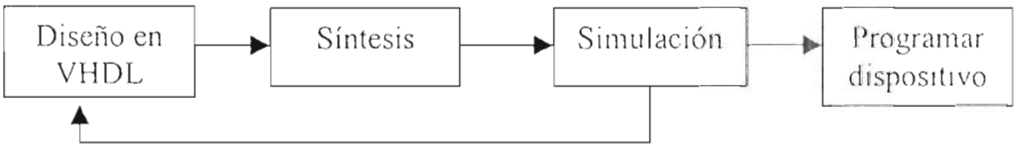


Fig. 1

Con la finalidad de señalar las ventajas que ofrece el diseño con VHDL, podemos mencionar otras técnicas utilizadas en el diseño electrónico.

Una técnica tradicional del diseño de sistemas electrónicos consiste en construir sistemas en base a la conexión de componentes electrónicos individuales y de funcionamiento estándar, por ejemplo los dispositivos TTL. De esta forma se interconectan todos los componentes necesarios para lograr un funcionamiento integral de algún sistema. Esta conexión se realiza generalmente en un circuito impreso, basándose en un esquema lógico. En el análisis del diseño se deben tomar en cuenta las características físicas de los distintos componentes utilizados, como son los tiempos de respuesta, voltajes, calidad del impreso, etc. Esta técnica es muy utilizada en la actualidad, si embargo, no se puede comparar con todas las ventajas que ofrece el uso del VHDL; al diseñar con VHDL podemos centrar la atención en el desarrollo funcional de nuestro sistema sin preocuparnos en gran medida por las cuestiones de conexión interna. Los procesos involucrados en el desarrollo final de un prototipo, los realizan la herramienta de CAD dependiendo del dispositivo seleccionado.

Otra alternativa del diseño electrónico es el uso de un microcontrolador. Esta opción ofrece grandes ventajas y tiene una amplia gama de aplicaciones. Lo primero es escoger el microcontrolador adecuado para nuestra aplicación. La programación de los microcontroladores varía según el fabricante, por lo que debemos conocer su lenguaje de programación específico. El nivel de programación generalmente es ensamblador, lo que hace más complicado el desarrollo de los programas. El uso de un microcontrolador es muy buena opción para el desarrollo de sistemas digitales, sin embargo, el VHDL ofrece mayores ventajas en diferentes aspectos. Con el VHDL podemos manejar diferentes niveles de abstracción del lenguaje, por lo que el uso de un nivel alto facilita en gran medida la programación, además, no está sujeta a muchas restricciones como en el caso de los microcontroladores en el que su programación es en bajo nivel. El VHDL y los dispositivos lógicos avanzados ofrecen la flexibilidad de poder describir el funcionamiento, incluso, de un microcontrolador.

La utilización de esta tecnología con VHDL no sólo facilita el diseño, sino que también permite al diseñador elaborar prototipos personalizados reduciendo costos, tiempo y espacio, sin necesidad de procesos sofisticados y externos de fabricación de un prototipo.

El VHDL también es muy utilizado para especificación y diseño de modelos para simulación.

El presente trabajo es un estudio del VHDL, con la finalidad de describir su funcionamiento y la metodología de diseño que se puede seguir para el desarrollo óptimo de sistemas electrónicos digitales, mostrando las grandes ventajas que ofrece esta tecnología sobre otras técnicas de diseño.

A continuación se señala brevemente el contenido de cada capítulo.

Capítulo 2.

En este capítulo se describen las principales características del VHDL, así como sus ventajas y desventajas.

Capítulo 3.

Se describen brevemente los dispositivos de arquitectura reconfigurable con la finalidad de conocer de una manera generalizada la forma en que es posible la implementación del diseño.

Capítulo 4

Se explican los diferentes niveles de abstracción del lenguaje utilizados en el VHDL para la descripción de componentes.

Capítulo 5.

Se describen los procesos simultáneos y secuenciales involucrados en la descripción de la funcionalidad de un componente.

Capítulo 6.

Se explica la forma en que podemos utilizar el VHDL en forma estructurada para la descripción de un diseño en base a jerarquías.

Capítulo 7.

Se desarrolla una metodología para una descripción óptima de sistemas electrónicos digitales.

Capítulo 8.

Se ejemplifica la metodología de diseño con el desarrollo de un procesador basado en la arquitectura RISC.

Apéndice A.

Aquí se incluyen los conceptos básicos de cómo empezar a utilizar el VHDL, con una descripción de sus principales instrucciones, tipos de datos, operadores, etc.

Apéndice B

Contiene algunos modelos de memorias prediseñados por un programa de CAD.

2. VHDL, LENGUAJE DESCRIPTOR DE HARDWARE

A principio de los años ochenta, el departamento de defensa norteamericano comenzó el desarrollo de un proyecto llamado VHSIC (*Very High Speed Integrated Circuit*). Esto surge ante la necesidad de contar con una técnica estándar para describir el funcionamiento de sistemas electrónicos y hacer más sencillo su mantenimiento, así como la depuración de los algoritmos de diseño. También se pretendía con ello resolver el problema de modificar el hardware diseñado en un proyecto para utilizarlo en otro, lo que no era posible hasta entonces porque no existía una herramienta adecuada.

En 1983 el pentágono comisionó a las empresas IBM, Intermetrics y Texas Instruments para el desarrollo de un lenguaje de diseño, como una herramienta que permitiera la estandarización de los procesos antes mencionados. El resultado fue el VHDL, el cual tiene mucha similitud al lenguaje de alto nivel llamado ADA (lenguaje usado por los militares) debido a que la empresa Intermetrics tenía amplios conocimientos y experiencia con este lenguaje.

El IEEE propuso su estándar en 1984 y tras varias versiones llevadas a cabo con la colaboración de la industria y de las universidades, el IEEE publicó en diciembre de 1987 el estándar IEEE std 1076-1987.

El VHDL se estandarizó como un lenguaje para modelar y especificar circuitos. De esta forma, con el VHDL es posible describir el funcionamiento de sistemas electrónicos a través de una forma basada en texto, lo que permite una comprensión más rápida y clara del sistema a diferencia de analizar diagramas con gran cantidad de componentes.

Se debe tener muy en cuenta que si bien el VHDL se parece a un lenguaje de alto nivel, su funcionamiento es muy distinto debido a que el VHDL describe estructuras de hardware, por lo que sus operaciones se ejecutan en forma simultánea a diferencia de la ejecución secuencial de instrucciones de un programa de alto nivel.

Actualmente el VHDL es el lenguaje más utilizados para especificar, modelar y diseñar sistemas electrónicos digitales. VHDL (VHSIC

Hardware Description Language) por sus siglas en inglés, es un lenguaje descriptor de hardware para dispositivos de muy alta velocidad.

Con el desarrollo de las herramientas de CAD, se logró simular los modelos descritos en VHDL, lo que aportó una ventaja notable en el diseño de sistemas electrónicos.

Los avances en la tecnología de cómputo y en los dispositivos lógicos programables, abren una nueva posibilidad para el diseño electrónico, ya que ahora no solo es posible simular el sistema electrónico sino también lograr su implementación física en un dispositivo con miles de compuertas y flip-flops configurables.

Con esto comienza una nueva etapa en el diseño de sistemas electrónicos, de manera que es posible diseñar prototipos en un circuito integrado reduciendo tiempo y costos sin necesidad de un proceso externo de fabricación.

2.1 Síntesis

Para realizar la implementación física del diseño en el dispositivo reconfigurable, se involucra un proceso de transformación muy importante denominado síntesis. Este proceso consiste en traducir el modelo digital descrito en VHDL a un nivel de abstracción más bajo y detallado, que genera y realiza la conexión de las compuertas y flip-flops necesarios para realizar la función descrita por el VHDL. En este proceso se genera un archivo que equivale al diagrama lógico, y contiene la información requerida para su implementación. Este proceso es semejante al efectuado con los lenguajes de alto nivel en el que se compila el programa para traducirlo a lenguaje máquina.

Es importante señalar que el VHDL es un lenguaje estandarizado para especificación y no para diseño, es decir, los procesos de síntesis no están estandarizados lo que significa que dependen del programa de síntesis utilizado.

Generalmente existen dos fases en la traducción de nuestro diseño en VHDL para poder programar el dispositivo.

- Síntesis: traduce el funcionamiento descrito en VHDL a su equivalente electrónico en términos de compuertas y flip-flops. En este proceso se aplican diferentes algoritmos de reducción y optimización de funciones. También realiza una medición del total de componentes necesarios para poder efectuar el funcionamiento descrito en el dispositivo seleccionado. Después de esta fase podemos realizar una simulación funcional.
- Implementación: se generan los mapas, rutas y colocación de las conexiones de los elementos **físicos** necesarios en el dispositivo lógico. De esta forma se genera un archivo que contiene la información necesaria para programarlo. Con esto es posible realizar una simulación en tiempo, debido a que se toman las características físicas del dispositivo, como pueden ser los retardos y tiempos de respuesta.

2.2 Ventajas y desventajas del VHDL

La elección de utilizar VHDL, no sólo se debe a que es un lenguaje descriptor de hardware estandarizado por la IEEE, sino también, por ser el de mayor uso. Además ofrece grandes ventajas para el diseño y mucha claridad para la interpretación y desarrollo de los sistemas electrónicos.

Existen otros lenguajes para la descripción de hardware, de los que destaca el llamado Verilog. Este lenguaje también está siendo utilizado ampliamente y su proceso de estandarización está en trámite. Verilog es muy semejante al lenguaje de alto nivel C y ofrece muy buenas opciones para el diseño.

Con la finalidad de mostrar las ventajas del VHDL sobre otros lenguajes podemos señalar las siguientes características:

- Es una lenguaje estándar, lo que lo hace independiente de los sistemas de CAD.
- Permite describir el funcionamiento del sistema electrónico con instrucciones en un nivel de abstracción alto, lo que facilita en gran medida el análisis y el diseño.
- Es posible verificar la funcionalidad del componente descrito, antes de su implementación.
- Los programas pueden utilizarse para especificación.

- Existen modelos comerciales en VHDL de componentes estándar.
- Al diseñar con VHDL podemos centrar la atención en el comportamiento funcional del componente y dejar los procesos de síntesis e implementación a las herramientas de CAD.
- Los componentes diseñados en VHDL pueden ser reutilizados por varios sistemas.

La única desventaja de esta tecnología es que los procesos de síntesis no están estandarizados.

3. TECNOLOGÍA DE LÓGICA PROGRAMABLE

Los dispositivos lógicos de campo programable (FPLD) son circuitos integrados utilizados para la implementación de hardware digital, permitiendo al usuario definir su operación funcional interna.

Una vez que se describe en software el funcionamiento que tendrá el dispositivo y después de la realización de una serie de procesos como el de síntesis, se envía la información requerida a una unidad de programación que configura físicamente el FPLD para que realice la función descrita.

Los avances logrados en la tecnología de lógica programable han permitido la obtención de dispositivos cada vez más complejos, logrando un mayor grado de integración y mayores velocidades de comunicación interna, con lo que se extienden las posibilidades de implementar una amplia gama de sistemas digitales.

El VHDL es utilizado para efectuar el diseño en dispositivos avanzados como son los CPLD y los FPGA:

- CPLD: es un PLD complejo, que consiste en la repetición de una serie de múltiples bloques tipo PLD en un solo dispositivo, con mayor grado de sofisticación. Los CPLD comerciales comúnmente usados son del tipo EPROM y EEPROM.
- FPGA : Es un arreglo de compuertas programables en campo que ofrece una estructura general y permite una capacidad de lógica muy alta. Los FPGA comerciales pueden ser arquitecturas basadas en SRAM o "antifuse".

Los CPLD proporcionan capacidades de lógica muy grandes, pero es algo difícil de extender esta arquitectura a densidades más altas. Para diseñar dispositivos con capacidad de lógica muy alta, se necesita una estructura diferente, la respuesta es la tecnología FPGA, la cual esta siendo ampliamente utilizada y se sigue desarrollando por las grandes capacidades que se pueden obtener a diferencia de la técnica de desarrollo de los CPLD, los FPGA han sido los responsables del cambio en la manera que se diseñan los circuitos electrónicos digitales.

3.1 FPGA , dispositivo de arquitectura reconfigurable

La estructura básica de los FPGAs es de arreglos, lo que significa que cada dispositivo comprende una serie bidimensional de bloques de compuertas lógicas que pueden interconectarse a través de los canales de asignación de rutas horizontales y verticales. (Ver Fig. 3.1.)

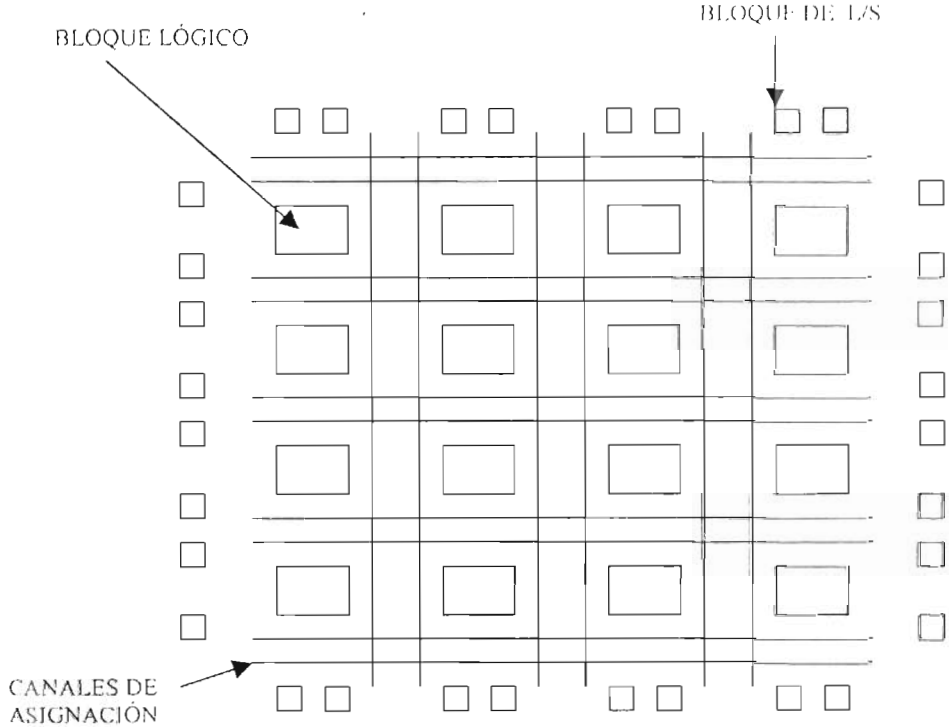


Fig. 3.1

Los bloques lógicos se conocen como bloques CLB y están conformados por tablas de búsqueda basadas en SRAM, llamadas LUTs. Un LUT es un pequeño arreglo de localidades de memoria de 1 bit de ancho, donde las líneas de dirección son las entradas del bloque lógico, y el dato de un bit de la memoria es la salida del LUT. Un LUT puede realizar cualquier función lógica de sus K entradas programando la tabla de verdad de la función lógica directamente en las localidades de memoria. El CLB contiene varios LUTs separados. Este arreglo le permite al CLB llevar a cabo una amplia gama de funciones lógicas de varias entradas, y también contiene circuitería que le permite realizar operaciones aritméticas eficazmente. Las LUTs en un CLB pueden ser configuradas como celdas de RAM de lectura/escritura. Cada CLB también contiene flip-flops de manera que la salida de los LUTs se conectan a la entrada de los flip-flops.

Además de la lógica programable, la otra característica importante de un FPGA es su estructura de interconexión. La interconexión se realiza mediante canales horizontales y verticales. Cada canal contiene algún número de segmentos de cables cortos que recorren un solo CLB (el número de segmentos en cada canal depende del modelo específico), segmentos más largos que recorren varios CLBs, y segmentos muy largos que recorren la longitud entera o el ancho del chip.

La relación velocidad-actuación de un circuito diseñado depende en parte de cómo se asignan los segmentos del cable a las señales individuales por las herramientas de software.

Existe una gran variedad de dispositivos comerciales que están siendo utilizados en distintas áreas, como el procesamiento digital de señales, comunicaciones, filtros, controladores de dispositivos, procesadores, co-procesadores matemáticos, robótica, etc.

4. DESCRIPCIÓN DE COMPONENTES

Al diseñar sistemas digitales en VHDL, encontramos elementos básicos conocidos como componentes, que en ocasiones pueden ser el sistema mismo. Al describir componentes en VHDL se involucran dos conceptos elementales: entidad y arquitectura que a continuación se describen.

Entidad

En esta sección se declaran los puertos de entrada y salida que utiliza el componente, con esto se define la interfaz de interconexión con el sistema. En VHDL se realiza de la siguiente manera:

Entity *nombre_entidad* is
  port (*nombre_señal* : *modo tipo*);
end *nombre_entidad*;

Donde:

- *nombre_entidad*: es el nombre del componente.
- *nombre_señal*: es el nombre del elemento de E/S del puerto.
- *modo*: define de qué forma opera la interfaz de la señal con el sistema.
- *tipo*: es el tipo de información que contiene la señal.

Los diferentes tipos y modos de la señal pueden verse en al apéndice A.

Arquitectura

En esta sección se define el funcionamiento interno del componente, puede estar conformado por funciones, operaciones, o bien, en forma estructural.

Una arquitectura siempre debe estar ligada a una entidad y se declara de la siguiente forma:

Architecture *nom_arq* of *nombre_entidad* is
  sección_declarativa
begin
  descripción_funcionamiento
end *nom_arq*;

Donde:

- *nom_arq*: es el nombre de la arquitectura.
- *nombre_entidad*: es el nombre de la entidad con la que se relaciona la arquitectura.
- *sección_declarativa*: se declaran objetos, tipos, componentes, etc.
- *descripción_funcionamiento*: describe el funcionamiento del componente.

Una característica importante de la descripción de componentes en VHDL, es la utilización de diferentes niveles de abstracción del lenguaje, es decir, nos permite describir componentes utilizando un conjunto de instrucciones especiales para definir el funcionamiento del componente o bien describirlo con operaciones básicas de álgebra booleana que es el nivel más bajo de descripción.

Los diferentes niveles son los siguientes:

- Comportamiento: en este nivel tenemos la posibilidad de describir el funcionamiento utilizando funciones especiales y características de tiempo sin tener que utilizar contadores, registros, etc. Se utiliza para modelar y simular componentes.
- RTL: el nivel de transferencia de registros es el más utilizado y describe el funcionamiento utilizando instrucciones que representan registros, operadores, multiplexores, máquinas de estado lógico, etc.
- Lógico: es el nivel en que se describen funciones del tipo Booleano.

Por ejemplo, para describir un multiplexor sencillo de 2 a 1, lo podríamos hacer de las siguientes maneras:

Nivel lógico	Nivel RTL
Entity mux is Port (a,b,sel : in std_logic; so :out std_logic); end mux; Architecture nl_arq of mux is Begin So<=(a and (not sel)) or (b and sel); End nl_arq;	Entity mux is Port (a,b,sel : in std_logic; so :out std_logic); end mux; Architecture rtl_arq of mux is Begin So<=a when sel='0' else b; End rtl_arq;

Una descripción en nivel comportamiento utilizando el ejemplo del multiplexor, puede ser la siguiente:

Nivel comportamiento
Entity mux is Port (a,b,sel : in std_logic; so :out std_logic); end mux; Architecture nc_arq of mux is Begin So<=a **after 20 ns** when sel='0' else b **after 20 ns**; End nc_arq;

Una descripción del último tipo, se utiliza para simulación de componentes, y hay que señalar que las instrucciones de retardo no se sintetizan. Esto significa que no todas las instrucciones del VHDL tienen un equivalente electrónico en el dispositivo. Inicialmente el VHDL se utilizaba para especificación y simulación, por lo que el nivel de comportamiento cuenta con un extenso juego de instrucciones para modelos de este tipo.

Diferentes procesos de síntesis se ejecutan dependiendo el nivel de abstracción que se esté utilizando, por lo que entre más bajo es el nivel, el proceso es más rápido, sin embargo, la descripción se complica.

Lo más indicado es elegir un nivel alto, de preferencia el nivel RTL, de ésta forma se facilita la descripción y el análisis, dejando la posibilidad de implementar el diseño en el dispositivo.

5. PROCESOS SIMULTÁNEOS Y SECUENCIALES

Como ya se mencionó con anterioridad, la descripción de componentes electrónicos en VHDL involucra un funcionamiento simultáneo de las diferentes operaciones que se describen, sin embargo, el VHDL también permite describir procesos secuenciales.

Las instrucciones de ejecución secuencial se describen dentro de una estructura especial llamada "process", la cual a su vez, se ejecuta en forma simultánea con respecto a las demás operaciones de la sección de arquitectura.

Una estructura secuencial se especifica de la siguiente forma:

Architecture *nom_arq* of *nombre_entidad* is
sección_declarativa
begin

instrucciones_de_ejecución_simultánea

Process (*lista sensitiva*)
sección declarativa
begin
    instrucciones_de_ejecución_secuencial
end process;

end *nom_arq*;

La instrucción "process" permanece en un estado de espera, hasta que ocurra algún cambio de valor en alguna señal de la lista sensitiva.

Es importante verificar cuáles señales se deben incluir en la lista sensitiva, tomando en cuenta que son las que activan la ejecución del proceso secuencial. De ésta forma nos aseguramos que el simulador genere los resultados correctos, tal y como resultarían en el dispositivo físico. Si se omite alguna señal, las herramientas de síntesis no lo señalan como un error, sin embargo, el resultado de la simulación será diferente al comportamiento que tendrá en el dispositivo lógico.

En la estructura "process", podemos describir el comportamiento de componentes electrónicos controlados por reloj y también circuitos combinacionales.

Lógica secuencial controlada por reloj

La estructura "process" se utiliza en los sistemas digitales donde encontramos elementos de lógica secuencial. Por ejemplo, la salida de un registro básico (flip-flop), depende de la señal de reloj, es decir, la asignación está sujeta a una condición que se debe cumplir previamente. Esto nos da una idea de la necesidad de una ejecución secuencial al describir este comportamiento en VHDL.

En la lógica secuencial encontramos que los circuitos síncronos son los más utilizados, por lo tanto es importante ver cómo utilizar las señales de reloj en estos procesos.

La señal de reloj se debe incluir en la lista sensitiva, para que cuando ocurra un evento en la señal, se active el proceso.

Una forma muy utilizada para describir señales de reloj es la siguiente:

```
Process (clk)
begin
        If clk'event and clk='1' then
            <instrucciones>
end process;
```

La opción 'event es un atributo de la señal "clk", que genera un valor de falso o verdadero si ocurre un evento de cambio en un periodo de tiempo; la condición " clk='1' " define el flanco de activación de la señal de reloj, en este caso, es un flanco positivo.

Otra forma de descripción de señales de reloj es mediante la instrucción wait until, por ejemplo:

```
Process
begin
        wait until clk'event and clk='1';
            <instrucciones>
end process;
```

Nótese que la señal "clk" no está en la lista sensitiva debido a que el estado de espera lo produce la instrucción wait until, de forma que al cumplirse la condición, las instrucciones se ejecutan.

Circuitos combinacionales

En la estructura "process" también podemos describir circuitos combinacionales, en los cuales su salida cambia de forma instántanea, únicamente al cambiar su señal de entrada.

En los procesos combinacionales es importante incluir en la lista sensitiva, todas las señales que se utilizan como operandos en las funciones lógicas. Si no se incluye alguna señal, el resultado será erróneo, por ejemplo:

```
Entity ej_1 is
  port ( a ,b: in std_logic;
          s :out std_logic);
end ej_1;

Architecture ej_1_arq of ej_1  is
begin
Process (b)
begin
      s<= a or b;
end process;
end ej_1_arq;
```

Como se puede apreciar en la lista sensitiva, la señal "a" no está incluida, lo que provoca un resultado erróneo en la señal "s", debido a que si hay un cambio en la señal "a", no se activará el proceso secuencial, por lo que el resultado "s" no se actualiza.

En la estructura de proceso secuencial se permite el uso de variables, por lo que al describir un funcionamiento secuencial es importante tener clara la diferencia que tienen con las señales. Las variables toman el valor en forma instantánea y en las señales existe un retardo, por lo que su comportamiento es distinto.

Por ejemplo, si deseamos utilizar algún acumulador que guarde el resultado de una suma previa, la forma correcta es utilizar variables, como se muestra a continuación:

```
suma:=suma+1;
acum:=acum+ suma;
```

El funcionamiento de las variables es idéntico al que tienen en un programa de alto nivel.

Por otro lado si se describen esas mismas operaciones con señales:

```
suma<=suma+1;
acum<=acum+ suma;
```

el resultado es distinto, debido al retardo existente en la asignación. Mientras se realiza la asignación de "suma", el acumulador, "acum", tomará el valor de "suma" que tenga en ese instante y no el actualizado en la operación anterior.

Ejemplos:

Ejemplo 1.

Un flip-flop tipo D con activación de flanco positivo, puede ser descrito en VHDL de la siguiente forma:

```
Entity flip_flop is
  port ( din, clk : in std_logic;
         dout              :out std_logic);
end flip_flop;

Architecture ff_funcion of flip_flop  is
begin
```

```
        Process (clk)
        Begin
           if clk'event and clk='1' then
              dout <= din;
           end if;
        end process;
end ff_funcion;
```

La siguiente Tabla resume las diferentes formas en que se puede describir un flip-flop tipo D en la estructura del proceso secuencial.

Flip-flop tipo D	Con reset síncrono

Process (clk)
Begin
If clk'event and clk='1' then
dout <= din;
end if;
end process;
	Process (clk)
	Begin
	If clk'event and clk='1' then
	if rst='1' then
	dout <= '0';
	else
	dout <= din,
	end if;
	end if;
	end process;

Con habilitador de reloj	**Con reset asíncrono**

Process (clk)
Begin
If (clk'event and clk='1') then
if (en='1') then
dout <= din;
end if;
end if;
end process;
	Process (clk, rst)
	Begin
	If rst='1' then
	dout <= '0';
	elsif (clk'event and clk='1') then
	dout <= din,
	end if;
	end process;

Ejemplo 2.

Se diseñará un contador de 4 bits ascendente/descendente, con habilitador de conteo, reset asíncrono y con carga de número. (Ver fig. 5.1.)

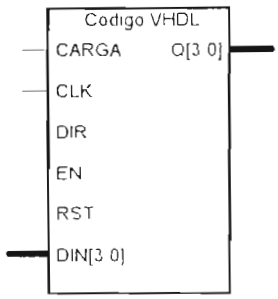


Fig. 5.1

```
--se incluyen las bibliotecas necesarias.
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

-- Nombre de las señaless para:
-- reloj y reset  :                          clk y rst respectivamente
-- selector de  conteo ascendente/descendente   dir
-- habilitador de conteo:                      en
-- habilitador para carga de numero:           carga
-- numero para carga:                          din
-- salida de conteo:                           q

--Definimos señaes de E/S
entity contador is
   port (
      clk,rst        : in STD_LOGIC;
      dir,en,carga   : in STD_LOGIC;
      din            : in integer range 0 to 15;
      q              :  out integer range 0 to 15
         );
end contador;

architecture contador_arch of contador is
signal suma:integer range 0 to 15;  --señal de uso interno para guardar la salida del
                                    -- contador
begin


q<=suma;




process (clk,rst)
begin
```

Operación de asignación de la señal interna a la salida del componente.

Inicio del proceso de descripción secuencial; se activa con algún cambio en el valor de la señal de reloj o la de reset.

La estructura secuencial se ejecuta como si fuera una sola instrucción, en forma simultánea a la asignación

```
--reset asincrono, activo en alto
if rst='1' then
        suma <= 0;

--si no se cumple la condicion de reset
--se verifica la siguiente condicion,

   elsif clk='1' and clk'event then
       if carga = '1' then      --verificamos si hay carga de dato
          suma <= din;
       else
          if en = '1' then   --verificamos si esta habilitado el contador, activo en alto
            if dir='1' then    --si esta habilitado el contador verificamos si es conteo
                                    --ascendente(dir=1)/ descendente(dir   0)
                    suma <= suma + 1;
              else
                    suma <= suma - 1;
              end if;  --fin de condicion del tipo de conteo
           end if;    --fin de condicion del habilitador de conteo
        end if;  --fin de condicion de verificacion de carga
  end if;     --fin de condicion de reset y de reloj
end process;
end contador_arch;
```

Se utiliza una señal interna llamada "suma", que es la encargada de llevar el control de conteo, otra alternativa para prescindir de la señal interna, es declarar la salida "q" en un modo de e/s tipo "buffer", de manera que la señal de salida puede ser releída internamente y así llevar el control del conteo.

El resultado de la simulación funcional del componente se puede ver en la Figura 5.2.

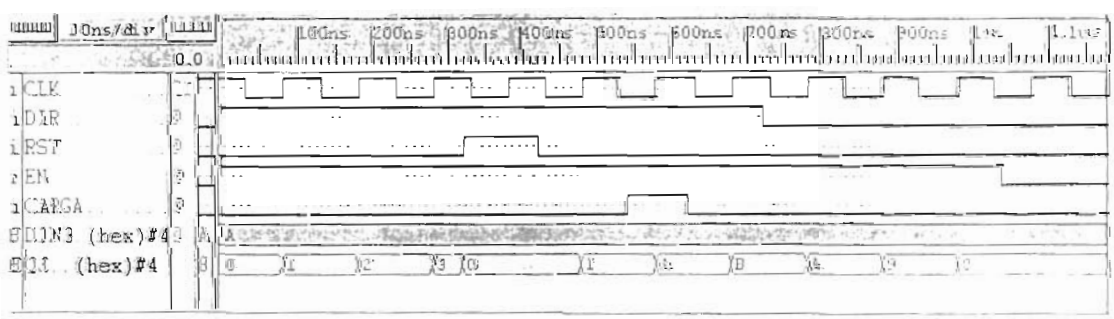


Fig. 5.2

Se asignan los valores a las señales de entrada para realizar la simulación, de manera que podamos analizar el funcionamiento del contador.

En la salida "Q" se visualiza el valor hexadecimal que tiene el contador.

Ejemplo 3.

Se diseñará un divisor de frecuencia a partir de una señal de entrada determinada.

Basándose en el desarrollo de un contador podemos elaborar el divisor, tomando alguno de los bits del contador como una nueva señal. (Ver Fig. 5.3.)

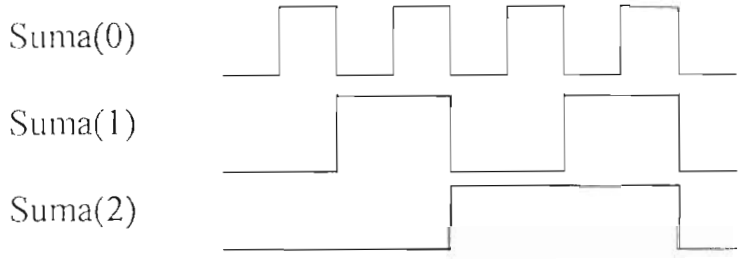


Fig. 5.3

Así, podemos generar nuevas frecuencias de reloj con los bits 0, 1 y 2 del registro de suma.

Las señales de e/s serán las siguientes:

Clk: frecuencia de entrada,
Rst: señal de reset,
Sel_f: selector de frecuencia,
Clk_out: frecuencia de salida seleccionada.

La descripción en VHDL es la siguiente:

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

entity divisor is
```

```
    port (    clk,rst        : in std_logic;
       sel_f                 : in std_logic_vector(2 downto 0),
       clk_out                : out std_logic    );
end divisor;

architecture div_arch of divisor is
signal suma:std_logic_vector(2 downto 0);

begin

clk_out<=suma(0) when sel_f = "00"else
            suma(1) when sel_f =  "01" else
            suma(2) when  sel_f = "10" else
            clk;


process (clk,rst)
begin

    if rst='1' then
         suma <= "000";

    elsif clk='0' and clk'event then
            suma <= suma + 1;
    end if;
end process;

end div_arch;
```

También se deja como posibilidad de elección de la señal de entrada "clk", esto por si se desea reutilizar el componente dentro de un sistema, también tenga esa opción.

Máquinas de estado lógico

El VHDL ofrece grandes ventajas para la descripción de máquinas de estado lógico. Esto se debe a que las estructuras de decisión facilitan el cálculo de cambio de estado.

Generalmente las herramientas de software ofrecen una opción gráfica para construir el diagrama de estados y automáticamente generar el código, sin embargo, lo ideal es aprovechar la flexibilidad que ofrece el

VHDL para diseñar de manera personalizada, ajustando el diseño a nuestras necesidades.

Una vez que diseñamos nuestro diagrama de estados, el procedimiento para describirlo en VHDL podría ser el siguiente:

- Definir un tipo de variable que enumere los diferentes estados.
- Construir un proceso secuencial activado con una señal de reloj.
- Verificar con una estructura de decisión, generalmente una de opción múltiple, el estado actual y la condición necesaria para que cambie al estado siguiente.
- La salida que se genere en cada estado puede estar en una estructura diferente, ya sea en la parte general de la arquitectura o bien en la parte secuencial. Una forma recomendada es colocar la salida en una estructura secuencial activada por reloj. De esta manera, el proceso de síntesis genera flip-flops a la salida de la señal, lo que produce una señal muy estable.

Ejemplo:

La descripción de un componente de control para un motor a pasos se puede realizar de la siguiente manera:

Las señales para el control serán:

Clk: señal de reloj .
Dir : selector de sentido de giro, cuando es '1' gira
a la derecha, '0' a la izquierda.
En : habilitador de control.
Rst: señal de reset.
S[3:0]: vector de salida, señales de control para el motor.

El componente de VHDL lo podemos representar como se aprecia en la Figura 5.4.

Fig. 5.4

La secuencia de la señal de salida para producir el giro en el motor, la podemos representar con un diagrama de estados como el de la Figura 5.5.

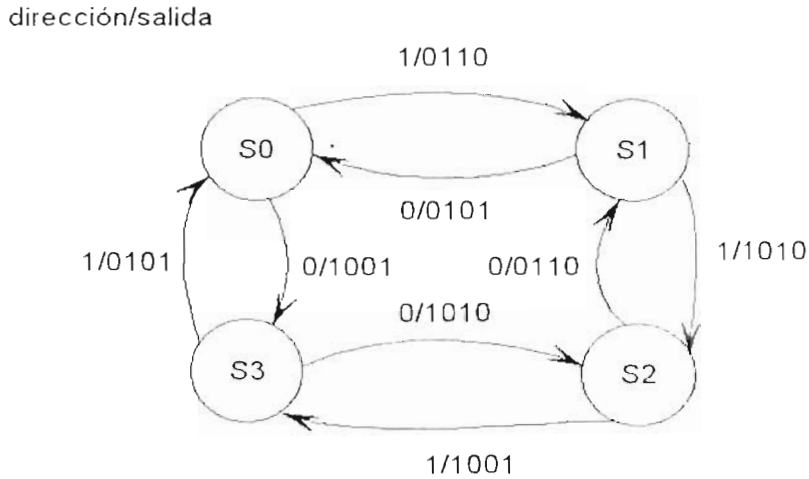


Fig. 5.5

El diseño en VHDL es el siguiente:

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


entity motorpasos is
   port (
      clk: in STD_LOGIC;
      dir: in STD_LOGIC;
      rst,en: in STD_LOGIC;
      s: out STD_LOGIC_vector(3 downto 0)
   );
end motorpasos;

architecture motorpasos_arch of motorpasos is
type estado is (s0,s1,s2,s3);    --definimos un tipo de variable que contiene los
                                 --cuatro diferentes posibles estados.
signal paso:estado;

begin
paso_a: process (clk,rst)
begin
      if rst = '1' then
               paso<=s0;          --en el reset definimos el estado inicial
```

```
            s<= "0101";
        elsif clk'event and clk='0' then
            if en = '1' then
                case paso is
                        when s0 => if dir = '1' then
                                        paso<=s1;
                                        s<= "0110";
                                   else
                                        paso<=s3;
                                        s<= "1001";
                                   end if;
                        when s1 => if dir = '1' then
                                        paso<=s2;
                                        s<= "1010";
                                   else
                                        paso<=s0;
                                        s<= "0101";
                                   end if;

                        when s2 => if dir = '1' then
                                        paso<=s3;
                                        s<= "1001";
                                   else
                                        paso<=s1;
                                        s<= "0110";
                                   end if;
                        when s3 => if dir = '1' then
                                        paso<=s0;
                                        s<= "0101";
                                   else
                                        paso<=s2;
                                        s<= "1010";
                                   end if;
                end case;
            end if;
        end if;
    end process;
    end motorpasos_arch;
```

6. DISEÑO BASADO EN JERARQUÍAS

El VHDL estructurado tiene otra gran ventaja para el diseño de componentes que permite describir el diseño en base a jerarquías, es decir, podemos agrupar componentes que a su vez contengan otros componentes. De esta manera se van formando diferentes niveles como en un diagrama de árbol. Los módulos que se generan en cada nivel de jerarquía se basan en el concepto de "caja negra" en el que solo se visualizan las entradas y salidas.

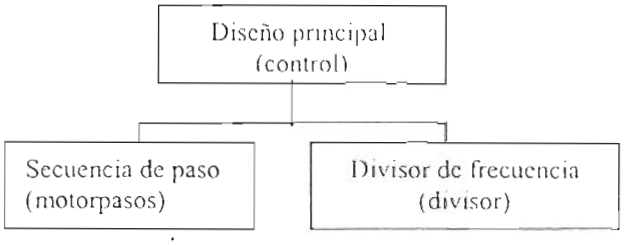
De esta forma, diseñamos un componente que se describe con base al funcionamiento de varios subcomponentes. Para esto se crean instancias de los subcomponentes, es decir, se hace un llamado a ellos, en el que únicamente se relaciona las señales de e/s del subcomponente con la nueva interfase diseñada en el nuevo sistema.

Las principales consideraciones que se deben tener en cuenta al diseñar en forma jerárquica son las siguientes:

1. Aplicar esta técnica en diseños que necesitan más de 2000 compuertas para efectuar su función.
2. Cada nivel debe tener un mínimo de 500 compuertas y un máximo de 5000, preferentemente.
3. No exceder de cuatro niveles de jerarquía.
4. Tomar en cuenta los tipos de datos que se utilizan al crear la instancia.
5. Cada nivel deberá tener una funcionalidad homogénea y sólo una señal de reloj.

Con la finalidad de mostrar la forma en que se realiza el diseño en base a jerarquías, se diseñará un control para un motor a pasos de velocidad variable, creando instancias de los componentes que se describieron en el capítulo anterior.

Los niveles serían los siguientes:



Las señales des e/s del puerto de control principal a diseñar son las siguientes:

clk_in: señal para la frecuencia de entrada,
rst_in: reset principal,
en_in: habilitador del control,
dir_in: dirección de rotación,
sel: selector de frecuencia,
q_sec: salida de secuencia de control para el motor.
frec_o: esta señal la definimos para poder visualizar en el simulador la frecuencia seleccionada.

El componente de VHDL lo podemos representar como se aprecia en la Figura 6.1.

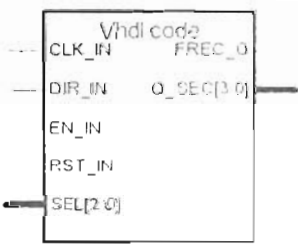


Fig. 6.1

Se definirá una señal de uso interno llamada "frec", de manera que almacene la salida de la frecuencia seleccionada en el subcomponente llamado "divisor" y a su vez la asigne a la entrada de frecuencia del subcomponente de control llamado "motorpasos", también se asignará a la señal "frec_o", para visualizarla en el simulador. (Ver Fig. 6.2.)

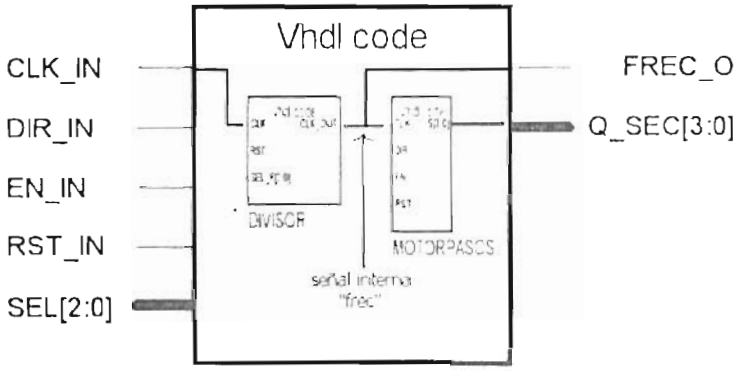


Fig. 6.2

En la sección declarativa de la arquitectura se definen los componentes que se utilizarán, y en la parte de descripción del funcionamiento se interconectan las señales internas de los componentes con su nuevo sistema.

La descripción completa del control principal es la siguiente:

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

entity control is
   port (
      clk_in,rst_in,en_in,dir_in  : in std_logic;
      frec_o                      : out std_logic;
      sel                         : in std_logic_vector(2 downto 0);
      q_sec                       : out std_logic_vector(3 downto 0)
       );
end control;

architecture control_arch of control is
signal frec: std_logic;

component divisor
port (  clk,rst      : in std_logic;
   sel_f  : in std_logic_vector(2 downto 0);
      clk_out      : out std_logic
     );
end component;
```

Se especifican los componentes a los que se les va a crear una instancia, describiendo sus señales de e/s. El orden de las señales debe ser el mismo que el descrito en el diseño del componente. Generalmente conviene copiar la definición de puerto de la entidad del componente y después pegarla en la especificación de la instancia.

```
component motorpasos
port (
      clk: in std_logic;
      dir: in std_logic;
      rst,en: in std_logic;
      s: out std_logic_vector(3 downto 0)
  );
end component;

begin

comp1: divisor port map(clk => clk_in, rst=>
rst_in,
                sel_f => sel, clk_out => frec);
comp2:motorpasos port map ( clk => frec, dir=>
dir_in, rst =>rst_in, en=>en_in, s=>q_sec);


frec_o<=frec;      --se copia la frecuencia seleccionada a la
                   --señal de salida del componente

end control_arch;
```

Se interconectan las señales de los subcomponentes con el nuevo sistema de control.

La instrucción "port map" nos permite interconectar las señales del subcomponente con el nuevo ambiente funcional; se debe tener muy en cuenta que los tipos de señal que se hayan definido en los subcomponentes deben ser iguales a los de las señales del control principal con los que se conectan.

Una vez que se definen los componentes de instancia mediante la instrucción "component" se pueden utilizar varias copias del mismo, por ejemplo, si quisiéramos agregar otro control de motor a pasos, sólo se define otro "port map" con sus conexiones, y no es necesario definirlo como otro subcomponente en la sección declarativa debido a que ya esta especificado, por ejemplo:

```
Comp3: motorpasos port map ( clk => señal , dir=> señal,
                             rst => señal, en=> señal, s=> señal);
```

Es recomendable etiquetarlos para diferenciar las diferentes instancias que se definan.

En la Figura 6.3 se muestra el resultado de la simulación.

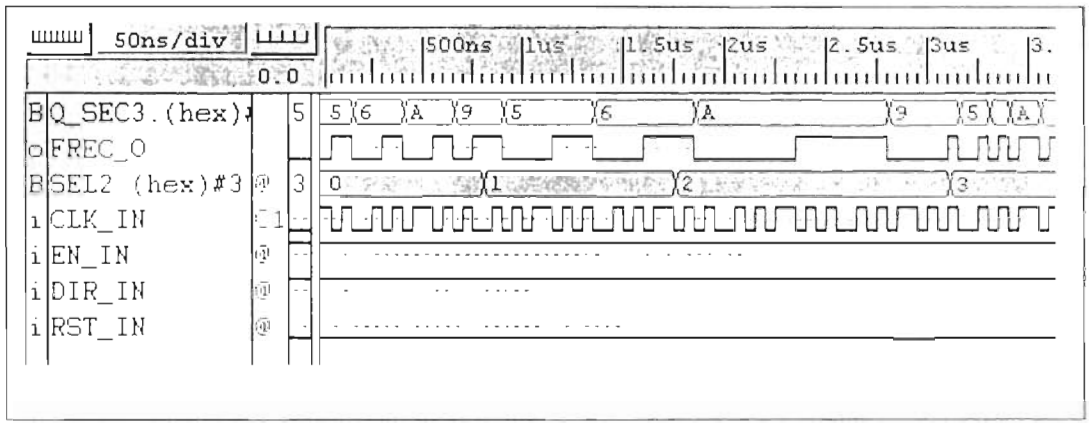


Fig. 6.3

En la gráfica se puede apreciar la variación de frecuencia en la salida "q_sec", que es la que genera la secuencia de pasos para el motor, dependiendo del cambio que haya en el selector de frecuencias "sel", sin embargo, hay que señalar que en el componente "divisor" la selección de frecuencias no depende de un proceso de reloj, por lo que la salida "q_sec" no esta sincronizada con el reloj principal, por eso se producen cambios de paso fuera de tiempo. Esto se soluciona, sólo con poner el selector de frecuencia en un proceso de reloj, de manera que el cambio de paso se efectúa de forma sincronizada con la señal de reloj, por ejemplo, la arquitectura del componente "divisor" quedaría de la siguiente forma:

```
architecture div_arch of divisor is
signal suma:std_logic_vector(2 downto 0);
signal clk_int:std_logic;
begin
clk_int<=clk;
process (clk,rst)
begin
    if rst='1' then
                suma <= "000";
    elsif clk='0' and clk'event then
          case sel_f is
            when "000" =>clk_out<=suma(0);
            when "001" =>clk_out<=suma(1);
            when "010" =>clk_out<=suma(2);
            when others =>clk_out<=clk_int;
```

```
            end case;
        suma <= suma + 1;
    end if;
end process;
end div_arch;
```

Debido a que la señal de reloj no se puede usar como dato de asignación dentro del proceso secuencial, se define una señal interna "clk_int", para copiar la señal .de reloj y así poder utilizarla como otra opción de selección dentro del proceso.

En la Figura 6.4 se muestra el resultado de la simulación.

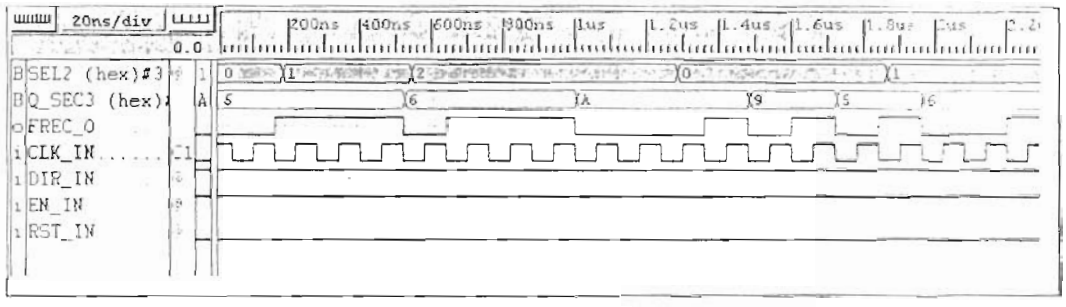


Fig. 6.4

En este caso, una vez que se introduce el valor de selección de la frecuencia deseada, el "divisor" espera la próxima señal de reloj en flanco negativo para tomar la nueva selección. De esta forma se sincroniza el componente de secuencia de pasos con la señal seleccionada.

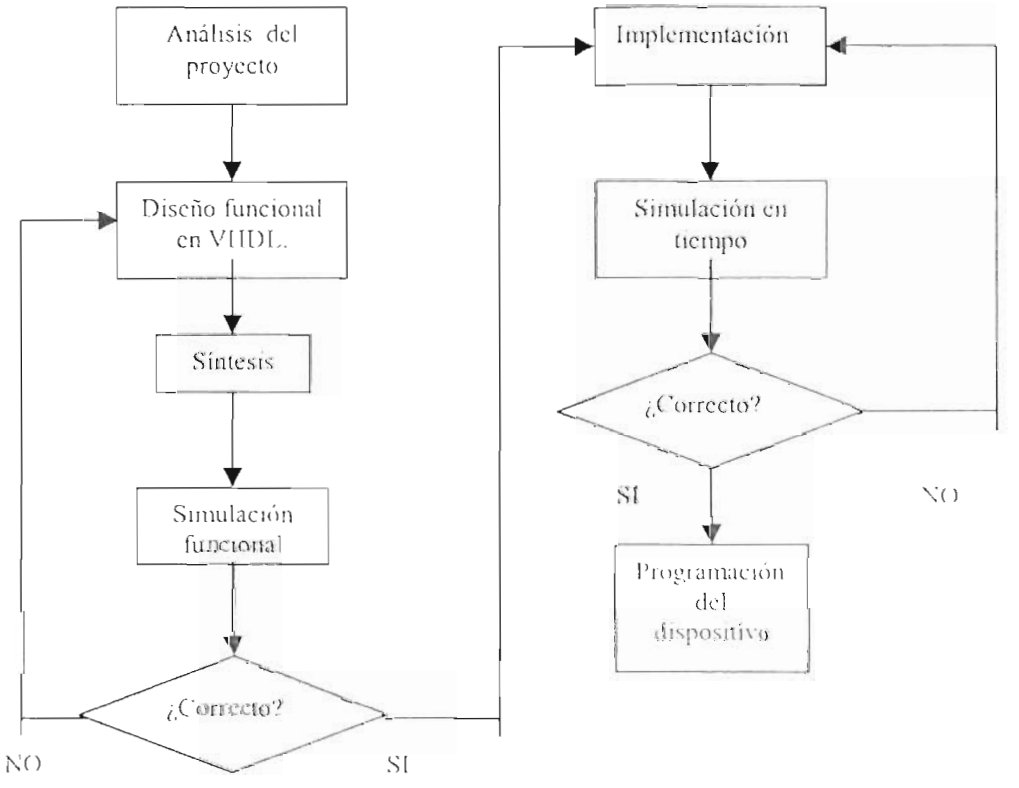
7. MÉTODO DE DESARROLLO

Al diseñar algún sistema electrónico con la utilización del VHDL, podemos considerar dos fases de desarrollo:

- Diseño funcional: descripción en VHDL utilizando un nivel de abstracción determinado.
- Síntesis: traducción del diseño en VHDL a nivel lógico para realizar verificación o proceder a la implementación física.

Es importante señalar que la fase de diseño en VHDL es un proceso independiente de las herramientas de CAD a diferencia del proceso de síntesis, que si depende del sistema de CAD seleccionado. Recordemos que el proceso de síntesis no está estandarizado por lo que su resultado se vuelve dependientes del sistema.

Tomando en cuenta las dos fases de diseño podemos describir el procedimiento de desarrollo mediante el siguiente diagrama de flujo.



Una vez que se realiza el análisis del proyecto y se determinan las especificaciones, podemos describir el funcionamiento del sistema. El procedimiento más recomendado para describir el diseño funcional en VHDL es el llamado "top-down".

Este método de diseño consiste en representar inicialmente el comportamiento funcional del sistema en un nivel de abstracción alto y a través del proceso de síntesis desarrollar el nivel de detalle del equivalente electrónico de nuestro modelo de VHDL. De esta manera partimos de un modelo del sistema en VHDL para llevarlo hasta su equivalente eléctrico en el dispositivo programable.

Al iniciar la descripción funcional, es recomendable utilizar el nivel de abstracción RTL, considerando que el proyecto es un modelo para implementar y no sólo para simular, como el caso del nivel de comportamiento.

Después de realizar la síntesis podemos efectuar una simulación funcional, es decir, únicamente se verifica que el diseño está haciendo lo descrito en VHDL de forma adecuada.

Una vez que se obtienen los resultados apropiados podemos ejecutar el proceso de implementación. En este proceso se toman en cuenta todas las características del dispositivo lógico seleccionado, así como la obtención de información necesaria para poder efectuar una simulación en tiempo y tener nuestro diseño listo para programarse.

La simulación en tiempo toma en cuenta las características de retardo y tiempos de respuesta de los diferentes componentes electrónicos generados. Esta simulación da una buena aproximación a los resultados que se obtendrían al probar físicamente el dispositivo.

La síntesis, la simulación y la implementación involucran una serie de procesos que en ocasiones pueden tomar un tiempo de realización considerable. El tiempo de proceso depende de la complejidad del diseño y del equipo utilizado, pero también depende en gran medida de la forma en que se describió el funcionamiento de los componentes.

Como ya se comentó en capítulos anteriores, existen varias alternativas para describir un mismo funcionamiento, sin embargo, no todas producen los resultados apropiados, por lo que se deben tener en cuenta algunas consideraciones en la forma de describir el funcionamiento del componente.

Con la finalidad de realizar una descripción adecuada de manera que se optimicen los procesos y sus tiempos de realización se deben tomar en cuenta las siguientes recomendaciones:

En la arquitectura:

- Describir el funcionamiento del componente utilizando el nivel de abstracción RTL. En ocasiones podemos describir algún componente directamente en nivel de compuertas, sin embargo, no siempre es recomendado porque además de hacer la descripción menos comprensible nos puede conducir fácilmente a cometer errores.
- Procurar realizar la descripción en forma concreta con el menor número de instrucciones y de señales internas.
- Utilizar el mínimo de procesos secuenciales en una arquitectura.
- El diseño debe ser síncrono.

En los procesos secuenciales:

- Determinar cuáles señales deben estar incluidas en la lista sensitiva para evitar señales que no son necesarias.
- En los procesos con operaciones de lógica combinacional se deben incluir en la lista sensitiva todas las señales involucradas en las operaciones y en el caso de que exista alguna asignación que esté sujeta a una estructura de decisión se deberán contemplar todos los casos condicionados, ya que si se ejecuta el proceso y no se asigna ningún valor a la señal de salida, al sintetizar se genera automáticamente un "latch", que conduce a resultados inapropiados en términos de número de compuertas generadas y de retardos en el sistema.
- Utilizar variables en lugar de señales en la medida que sea posible.
- Definir solamente una señal de reloj.

En el caso de diseños complejos que requieren del uso de un modelo estructurado en jerarquías se deben tomar en cuenta las consideraciones hechas en el capítulo anterior.

Generalmente las diferentes herramientas de programación existentes ofrecen una variedad de componentes de uso común; de esta forma podemos crear instancias de estos componentes en el desarrollo de nuestros proyectos; el uso de estos componentes facilitan la descripción pero también debemos tomar en cuenta que esto convierte a nuestro proyecto en un diseño que depende del sistema de CAD que se haya seleccionado.

Durante el proceso de síntesis generalmente el usuario puede señalar las limitantes de tiempo y espacio del dispositivo, de esta forma los algoritmos de síntesis buscarán las alternativas que satisfagan estas características así como para la implementación en la que se generan una serie de reportes en los que es posible revisar los resultados de los tiempos de respuesta así como del total de componentes y conexiones necesarias para desempeñar el funcionamiento descrito.

La ejecución de los diferentes procesos de síntesis y de implementación dependen del modelo de dispositivo seleccionado, de esta forma, durante la síntesis además de traducir a un nivel mas detallado también realiza una medición del diagrama generado con las capacidades del dispositivo.

8. DESARROLLO DE UN PROCESADOR BASADO EN LA ARQUITECTURA RISC

Con la finalidad de ejemplificar el desarrollo de un sistema digital con la utilización del VHDL, se realizará la descripción del funcionamiento de un procesador. Los procesadores son las máximas expresiones de los sistemas electrónicos digitales; de esta forma se pueden visualizar los alcances que se tienen con la utilización del VHDL.

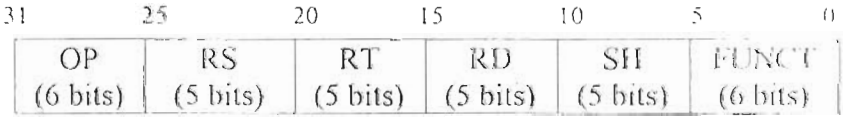
ANÁLISIS

El procesador a diseñar se basa en una arquitectura RISC, más en específico en un procesador MIPS. La principal característica de estos procesadores es su juego de instrucciones reducidas ejecutadas en un ciclo de reloj.

El tamaño de las instrucciones de un procesador MIPS es de treinta y dos bits y existen tres formatos que a continuación se especifican:

- Formato tipo R.

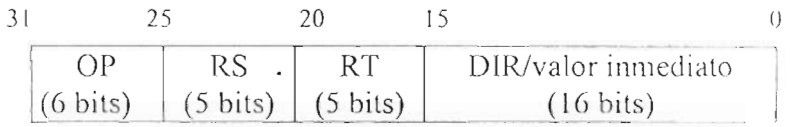
Las instrucciones con formato tipo R realizan operaciones únicamente de datos contenidos en los registros.



1. OP: código de operación, especifica qué operación representa esa instrucción.
2. RS: primer registro del operando fuente.
3. RT: segundo registro del operando fuente.
4. RD: registro del operando destino.
5. SH: cantidad de desplazamiento.
6. FUNCT: función, extensión del campo OP.

- Formato tipo I .

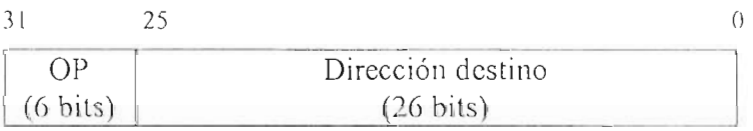
Las instrucciones del tipo I realizan transferencia de datos; dentro de este tipo de instrucciones se encuentran "load" y "store" las cuales son las únicas que realizan operaciones de transferencia a memoria.



1. OP: código de operación.
2. RS: primer registro del operando fuente.
3. RT: segundo registro del operando fuente.
4. Dir: dirección de memoria/valor inmediato.

- Formato tipo J.

Dentro de este formato están contenidas las instrucciones de salto.



Las principales instrucciones que se definen en el procesador a realizar se muestran en la Tabla 8.1.

Instrucción	Mnemónico	OP	Extensión de OP	Formato
Suma	Add	0	32	R
Resta	Sub	0	34	R
Y lógico	And	0	36	R
O lógico	Or	0	37	R
Bifurcación	Beq	4	-	I
Carga	Lw	35	-	I
Almacenamiento	Sw	43	-	I
Salto	Jmp	2	-	J

Tabla 8.1

Además de las características de las instrucciones, el procesador a diseñar contará con ocho registros de propósito general de ocho bits al igual

que el canal de datos, de esta forma se podrán realizar pequeños programas que muestren el funcionamiento del procesador descrito de una manera muy clara.

Se desarrollará el diseño basándose en un modelo en jerarquías; de tal forma que el nivel más alto será el que realiza la conexión de cinco módulos del segundo nivel de jerarquía. La división de estos módulos se realiza para facilitar el desarrollo del sistema y para describir el comportamiento de las diferentes etapas del procesador. (Ver fig. 8.1.)

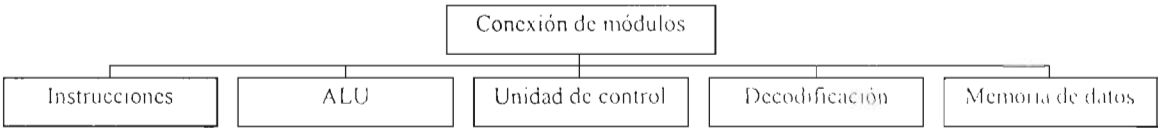


Fig. 8.1

Los módulos se describen a continuación:

- Unidad de control: analiza el código de operación (OP) de la instrucción y se generan las señales de control usadas para el funcionamiento de otras etapas del procesador.
- Instrucciones: funcionamiento del contador de programa, la memoria de instrucción y las operaciones para calcular la siguiente dirección de instrucción a ejecutar.
- Decodificación: configura la etapa de registros de propósito general y se decodifica parte de la instrucción (bit cero al veinticinco).
- ALU: unidad aritmética y multiplexores utilizados para señales de otras etapas.
- Memoria de datos: se describe la memoria de datos.

La definición de los módulos se realizó basándose en el diagrama de un procesador RISC (ver Fig 8.2), de tal forma, que cada módulo contenga los componentes necesarios de características homogéneas.

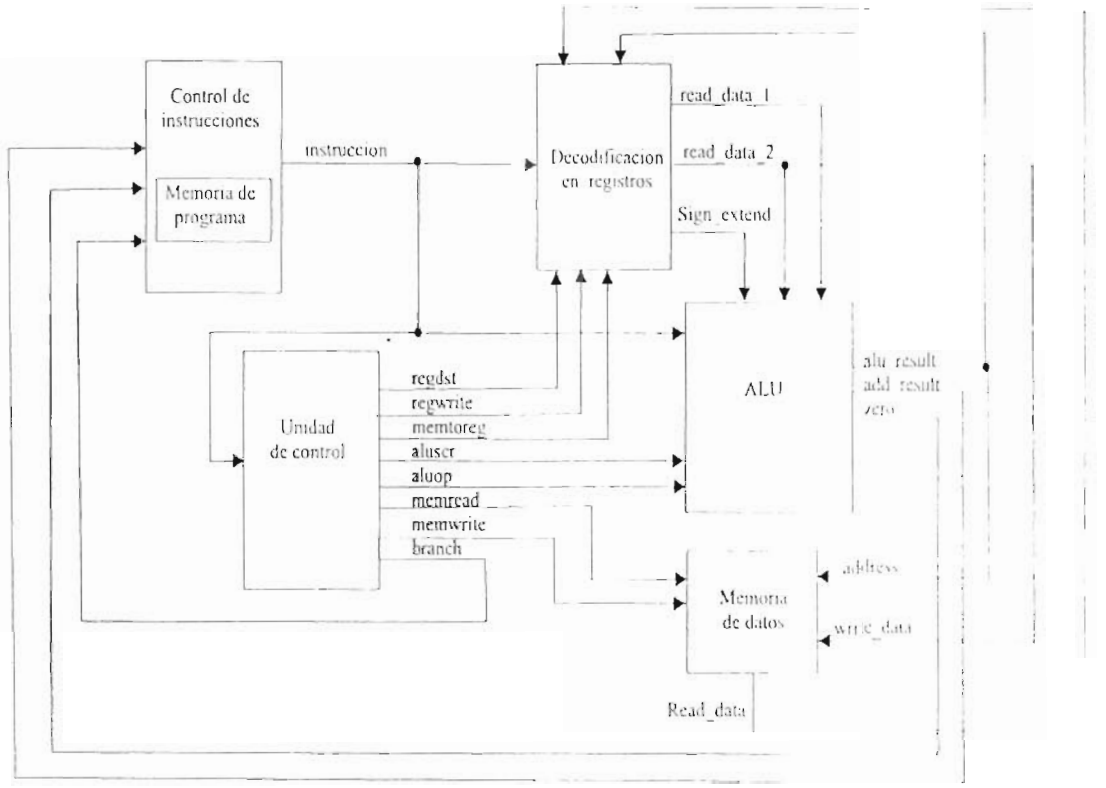


Fig. 8.2

DESCRIPCIÓN FUNCIONAL

Los nombres de las señales principales utilizados en la descripción funcional, serán los mismos que se especifican en la Fig. 8.2 para facilitar su interpretación.

Comenzamos con la descripción del módulo principal (conexión de módulos), en el que se definen las señales internas para interconectar los módulos del segundo nivel de jerarquía.

A continuación se muestra la descripción en VHDL del módulo principal, mostrando una explicación de cada fase.

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

--El puerto principal contiene las señales de entrada, reloj y reset, y las
--principales señales de salida para monitorear el funcionamiento en el
```

--simulador.

```
entity mips_conec is
  port (
        rst,clk: in std_logic;
        pc,alu_result_out,read_data_1_out,
        read_data_2_out,write_data_out:out   std_logic_vector(7 downto 0);
        instruccion_out:out std_logic_vector(31 downto 0);
        branch_out,jump_out,zero_out, memwrite_out,regwrite_out:out std_logic);
end mips_conec;

architecture conect of mips_conec is
```

--En la parte declarativa de la arquitectura comenzamos por especificar las
--señales internas que se utilizan para la interconexión de los módulos.

```
signal instruccion      :std_logic_vector(31 downto 0);
signal pc_b4            :std_logic_vector(7 downto 0);
signal read_data        :std_logic_vector(7 downto 0);
signal read_data_1      :std_logic_vector(7 downto 0),
signal read_data_2      :std_logic_vector(7 downto 0);
signal sign_extend      :std_logic_vector(7 downto 0);
signal alu_result       :std_logic_vector(7 downto 0);
signal add_result       :std_logic_vector(7 downto 0);
signal aluop            :std_logic_vector(1 downto 0);
signal alusrc           :std_logic;
signal zero             :std_logic;
signal branch           :std_logic;
signal jump             :std_logic;
signal regdst           :std_logic;
signal regwrite         :std_logic;
signal memwrite         :std_logic;
signal memtoreg         :std_logic;
signal memread          :std_logic;
```

-- Se especifican los componentes de instancia (los cinco módulos).

```
component inst_busc is
port (
      clk: in std_logic;
      rst: in std_logic;
      instruccion: out std_logic_vector (31 downto 0);
      pc_b4_out: out std_logic_vector (7 downto 0);
      add_result: in std_logic_vector (7 downto 0);
      branch: in std_logic;
      jump: in std_logic;
```

```
        zero: in std_logic;
        pc_out: out std_logic_vector (7 downto 0)
    );
end component;


component decodifica is
port (
        clk: in std_logic;
        rst: in std_logic;
        read_data_1: out std_logic_vector (7 downto 0);
        read_data_2: out std_logic_vector (7 downto 0);
        instruccion: in std_logic_vector (31 downto 0);
        read_data: in std_logic_vector (7 downto 0);
        alu_result: in std_logic_vector (7 downto 0);
        regwrite: in std_logic;
        memtoreg: in std_logic;
        regdst: in std_logic;
        sign_extend: out std_logic_vector (7 downto 0)
    ),
end component;

component unidad_control is
port (
        clk: in std_logic;
        rst: in std_logic;
        opcode: in std_logic_vector (5 downto 0);
        regdst: out std_logic;
        alusrc: out std_logic;
        memtoreg: out std_logic;
        regwrite: out std_logic;
        memread: out std_logic;
        memwrite: out std_logic;
        branch: out std_logic;
        jump: out std_logic;
        aluop: out std_logic_vector (1 downto 0)
    );
end component;

component alu is
port (
        clk: in std_logic;
        rst: in std_logic;
        read_data_1: in std_logic_vector (7 downto 0);
        read_data_2: in std_logic_vector (7 downto 0);
        sign_extend: in std_logic_vector (7 downto 0);
        function_opcode: in std_logic_vector (5 downto 0);
```

```
      aluop: in std_logic_vector (1 downto 0);
      alusrc: in std_logic;
      zero: out std_logic;
      alu_result: out std_logic_vector (7 downto 0);
      add_result: out std_logic_vector (7 downto 0);
      pc_b4: in std_logic_vector (7 downto 0)
       );
end component;

component memoria_datos is
port (
      clk: in std_logic;
      rst: in std_logic;
      read_data: out std_logic_vector (7 downto 0);
      address: in std_logic_vector (7 downto 0);
      write_data: in std_logic_vector (7 downto 0);
      memread: in std_logic;
      memwrite: in std_logic
   );
end component;

begin

--Comenzamos la descripción funcional conectando las señales de los
--módulos de instancia con el nuevo ambiente funcional.

inst:inst_busc
port map(
      clk               =>clk,
      rst               =>rst,
      instruccion       =>instruccion,
      pc_b4_out         =>pc_b4,
      add_result        =>add_result,
      branch            =>branch,
      jump              =>jump,
      zero              =>zero,
      pc_out            =>pc
   );

decod:decodifica
        port map(
              clk               =>clk,
              rst               =>rst,
              read_data_1       =>read_data_1,
              read_data_2       =>read_data_2,
              instruccion       =>instruccion,
              read_data=>read_data,
```

```
            alu_result      =>alu_result,
            regwrite  =>regwrite,
            memtoreg        =>memtoreg,
            regdst          =>regdst,
            sign_extend     =>sign_extend
           );

ctrl:unidad_control
port map(
      clk               =>clk,
      rst               =>rst,
      opcode            =>instruccion(31 downto 26),
      regdst            =>regdst,
      alusrc            =>alusrc,
      memtoreg          =>memtoreg,
      regwrite          =>regwrite,
      memread           =>memread,
      memwrite          =>memwrite,
      branch            =>branch,
     jump               =>jump,
      aluop             =>aluop
   );

alu1:alu
port map(
      clk               =>clk,
      rst               =>rst,
      read_data_1       =>read_data_1,
      read_data_2       =>read_data_2,
      sign_extend       =>sign_extend,
      function_opcode   =>instruccion(5 downto 0),
      aluop             =>aluop,
      alusrc            =>alusrc,
      zero              =>zero,
      alu_result        =>alu_result,
      add_result        =>add_result,
      pc_b4             =>pc_b4
       );

mem:memoria_datos
port map(
      clk               =>clk,
      rst               =>rst,
      read_data         =>read_data,
      address           =>alu_result,
      write_data        =>read_data_2,
      memread           =>memread,
```

```
    memwrite        =>memwrite
  );

--Asignamos las señales internas a las señales de salida del puerto
--principal, para visualizarlas en el simulador.

instruccion_out  <= instruccion;
alu_result_out   <= alu_result;
read_data_1_out <= read_data_1;
read_data_2_out <= read_data_2;
write_data_out   <= read_data when memtoreg = '1' else alu_result;
branch_out       <= branch;
jump_out         <=jump;
zero_out         <= zero;
regwrite_out     <=regwrite;
memwrite_out     <=memwrite;
end conect;
```

Una vez que conocemos las señales de e/s de los diferentes módulos y su interconexión, podemos comenzar a describir su funcionamiento teniendo una visión global del sistema.

Control de instrucciones

En el módulo de instrucciones describimos el contador de programa, la memoria de programa y los cálculos para obtener la siguiente instrucción a ejecutar. (Ver Fig. 8.3.)

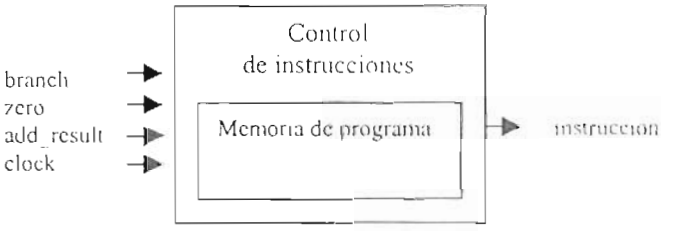


Fig. 8.3

La memoria de programa se puede describir de distintas maneras, sin embargo, con la finalidad de optimizar los procesos de síntesis, podemos crear una instancia de un modelo determinado en el CAD. Esto nos facilitará la escritura del programa de prueba, así como la simulación.

Los modelos en VHDL de la memoria de programa y de datos se pueden ver en el apéndice B.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all,

entity inst_buse is
   port (
      clk: in std_logic;
      rst: in std_logic;
      instruccion: out std_logic_vector (31 downto 0);
      pc_b4_out: out std_logic_vector (7 downto 0);
      add_result: in std_logic_vector (7 downto 0);
      branch: in std_logic;
      jump: in std_logic;
      zero: in std_logic;
      pc_out: out std_logic_vector (7 downto 0)
   );
end inst_buse;

architecture inst_buse_arch of inst_buse is

--Se definen señales internas, utilizadas para realizar la conexión con la
--memoria de programa y realizar algunos cálculos.

signal inst :std_logic_vector(31 downto 0);
signal pc,pc_b4,jmp_inst:std_logic_vector(7 downto 0);
signal pcprox :std_logic_vector(7 downto 0);

--Definimos instancia de la memoria de programa.

component mem_prog
  port(
    a: in std_logic_vector(7 downto 0);
    do: out std_logic_vector(31 downto 0));
end component;

begin

--Conectamos las señales de la memoria de programa con las señales del
--control de instrucciones.

 mod_prog : mem_prog port map
(a => pc,
 do => inst),

--Se realiza el cálculo de la dirección de la siguiente instrucción a ejecutar, lo que
```

```
--equivale al sumador en el diagrama.
--Nótese que se incrementa en 1, debido a que la memoria se definió en 256
--localidades de 32 bits, es decir, no es direccionable en byte.

pc_b4<= pc +1;

--En el caso de que haya una instrucción de salto, se toman directamente
--los bits de la localidad objetivo de memoria. En este caso solo los ocho
--bits menos significativos, debido a que la memoria de programa es de 256
--localidades.

jmp_inst<=inst(7 downto 0);

--La instrucción próxima a ejecutar toma algún valor dependiendo de la
--condición que se cumpla. Existen tres casos:
--    Bifurcación: salto condicionado relativo al PC, la dirección contenida en la
--                 señal "add_result" se calcula en  ALU.
--                 La condición depende de la bandera de cero y de la señal
--                 de control.
--   Salto incondicional: toma la dirección objetivo inmediata, dependiendo de
--                 la señal de control que indica que es una instrucción de salto.
--      Incremento en uno: secuencia normal de programa.

 peprox<=add_result when ((branch = '1')  and (zero ='1'))   else
         jmp_inst when jump = '1'          else
         pc_b4;

--Asignamos los valores de las señales internas a las de salida, para
--visualizarlas en el simulador.

instruccion<=inst;
pc_out <= pc;
pc_b4_out <= pc_b4;

--Describimos un proceso secuencial controlado por reloj, para asignar  la
--dirección de la próxima instrucción a ejecutar al contador de programa.
--De esta forma se actualiza en cada flanco positivo de la señal de reloj.

process(clk)
begin
        if (clk'event) and (clk ='1') then
                if rst ='1' then
                        pc <= "00000000";
                else
                        pc<= peprox;
                end if;
        end if;
```

```
end process;
end inst_busc_arch;

--Recuérdese que el proceso secuencial se ejecuta en forma simultánea a las
--otras operaciones.
```

Unidad de control

En esta fase se generan las señales de control de los otros módulos analizando el código de operación de la instrucción. (Ver Fig. 8.4.)

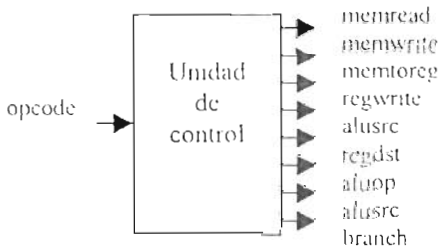


Fig. 8.4

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

entity unidad_control is
   port (
      clk: in std_logic;
      rst: in std_logic;
      opcode: in std_logic_vector (5 downto 0);
      regdst: out std_logic;
      alusrc: out std_logic;
      memtoreg: out std_logic;
      regwrite: out std_logic;
      memread: out std_logic;
      memwrite: out std_logic;
      branch: out std_logic;
      jump: out std_logic;
      aluop: out std_logic_vector (1 downto 0)
   );
end unidad_control;

architecture unidad_control_arch of unidad_control is

signal r,lw,sw,beq,jmp :std_logic;
```

```
begin

--Se analiza el código de operación de la instrucción, de manera que
--determina a qué formato pertenece (R, I o J). De esta forma envía las
--señales para habilitar diferentes módulos con determinado
--funcionamiento.

r   <= '1' when opcode  = "000000" else '0';
lw  <= '1' when opcode  = "100011" else '0';
sw  <= '1' when opcode  = "101011" else '0';
beq <= '1' when opcode  = "000100" else '0';
jmp <= '1' when opcode  = "000010" else '0';
regdst <= r;
alusrc <= lw or sw;
memtoreg <= lw;
regwrite <= r or lw;
memread <=lw;
memwrite <= sw;
branch <= beq,
jump<=jmp;
aluop(1) <= r;
aluop(0) <=beq;

end unidad_control_arch;
```

Decodificación de registros

La decodificación es el siguiente módulo a describir. En este se configura la memoria de registros de propósito general. Se incluyen dos puertos de lectura y uno de escritura. (Ver Fig. 8.5.)

Fig. 8.5

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all,
```

```
entity decodifica is
   port (
      clk: in std_logic;
      rst: in std_logic;
      read_data_1: out std_logic_vector (7 downto 0);
      read_data_2: out std_logic_vector (7 downto 0);
      instruccion: in std_logic_vector (31 downto 0);
      read_data  in std_logic_vector (7 downto 0);
      alu_result: in std_logic_vector (7 downto 0);
      regwrite: in std_logic;
      memtoreg: in std_logic;
      regdst: in std_logic;
      sign_extend: out std_logic_vector (7 downto 0)
   );
end decodifica,

architecture decodifica_arch of decodifica is
--Para los registros de propósito general, definimos un tipo de dato que
--consiste en una arreglo matricial de 8X8, con esto tenemos 8 registros de
--8 bits de propósito general.
type register_file is array (0 to 7) of std_logic_vector(7 downto 0);

--Definimos señales de uso interno.

signal register_array:register_file;
signal write_register_address   :std_logic_vector(4 downto 0);
signal write_data               :std_logic_vector(7 downto 0);
signal read_register_1_address:std_logic_vector(4 downto 0);
signal read_register_2_address:std_logic_vector(4 downto 0);
signal write_register_address_1:std_logic_vector(4 downto 0);
signal write_register_address_0:std_logic_vector(4 downto 0);
signal instruccion_inmediate_value:std_logic_vector(15 downto 0),

begin

--Decodificamos la instrucción , tomando los bits para los registros fuente y
--destino.

        read_register_1_address<=instruccion(25 downto 21);
        read_register_2_address<=instruccion(20 downto 16);
        write_register_address_1<=instruccion(15 downto 11); --formato R
        write_register_address_0<=instruccion(20 downto 16); --formato I
        instruccion_inmediate_value<=instruccion(15 downto 0);

--La información que se asigna a las siguientes señales puede ser datos,
--que procesará el módulo ALU, o información que necesite la memoria
```

```
--de datos, por ejemplo en la instrucción Sw.

    read_data_1<=register_array(conv_integer(read_register_1_address(2 downto 0)));
    read_data_2<=register_array(conv_integer(read_register_2_address(2 downto 0)));

--En la siguiente asignación, la señal contendrá la posición del registro de
--propósito general en donde se almacenará algún dato.
--Se realiza la asignación en forma condicionada. El valor que tome
--dependerá del tipo de instrucción.
--En el caso del formato R, toma la dirección del registro destino, que
--contendrá el resultado de alguna operación.
--En caso contrario, formato I, toma los bits como dirección para almacenar
--un dato proveniente de la memoria de datos. Por ejemplo en la instrucción
--LW.

write_register_address<=write_register_address_1 when regdst='1' --formato R
                        else write_register_address_0;           --formato I

--Ahora decidimos cuál dato se almacenará en la localidad ya determinada.
--El dato puede ser el resultado de alguna operación efectuada por el ALU
--o algún valor leído de la memoria de datos.
--Esta decisión depende de la señal de control que habilita la
--lectura de memoria de datos a registro.

    write_data<=alu_result(7 downto 0) when (memtoreg='0')
                else read_data;

--En el caso de instrucciones de formato I, la siguiente señal contiene la
--dirección o el valor de asignación inmediata.
--Por ejemplo en la instrucción Sw, la señal contendrá la dirección de
--memoria de datos en donde se almacenará algún dato.

    sign_extend<=instruccion_inmediate_value(7 downto 0);

--Solo se toman ocho bits, debido al tamaño de la memoria.
--Se describe un proceso controlado por reloj.

process(clk)
begin
 if clk'event and clk='1' then
--Contiene un reset síncrono que nos permite inicializar los registros,
--asignándoles un índice de control.
    if rst='1' then
            for i in 0 to 7 loop
                register_array(i)<=conv_std_logic_vector(i,8);
            end loop;
```

```
--En este proceso también se verifica la condición que permite almacenar el
--dato en el registro determinado.

    elsif regwrite = '1' and write_register_address /=0 then
        register_array(conv_integer(write_register_address(2 downto 0))) <= write_data;
    end if;

--La función de conversión se utiliza para obtener el número entero que
--equivale al número de registro determinado.

end if;
end process;
end decodifica_arch;
```

Módulo ALU

En esta etapa se interpreta el código para determinar la operación aritmética que se va a efectuar. También se describen los multiplexores que nos permiten seleccionar la información que se utiliza como entrada o salida de este módulo, según el formato de la instrucción. (Ver Fig. 8.6.)

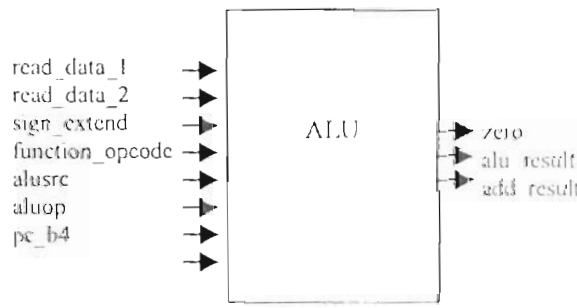


Fig. 8.6

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

entity alu is
   port (
      clk: in std_logic;
      rst: in std_logic;
      read_data_1: in std_logic_vector (7 downto 0);
      read_data_2: in std_logic_vector (7 downto 0);
      sign_extend: in std_logic_vector (7 downto 0);
      function_opcode: in std_logic_vector (5 downto 0);
      aluop: in std_logic_vector (1 downto 0);
```

```
      alusrc: in std_logic;
      zero: out std_logic;
      alu_result: out std_logic_vector (7 downto 0);
      add_result: out std_logic_vector (7 downto 0);
      pc_b4: in std_logic_vector (7 downto 0)
      );
end alu;

architecture alu_arch of alu is

signal a,b :std_logic_vector(7 downto 0);
signal alu_output_mux:std_logic_vector(7 downto 0);
signal branch_add :std_logic_vector(7 downto 0);
signal alu_ctl :std_logic_vector(2 downto 0);
begin

      a<=read_data_1;
```

--Toma la información proveniente de la etapa de decodificación. En este caso la --obtiene a través de un multiplexor que toma el dato según el tipo de instrucción a --procesar. Esta decisión la toma con base a la señal de control "alusrc" .

```
      b<=read_data_2
            when(alusrc ='0')
            else sign_extend(7 downto 0);   --formato I
```

--Decodifica la instrucción a ejecutar, asignando señales de control.

```
      alu_ctl(0)<= (function_opcode(0) or function_opcode(3)) and aluop(1);
      alu_ctl(1)<= (not function_opcode(2)) or (not aluop(1));
      alu_ctl(2)<= (function_opcode(1) and aluop(1)) or aluop(0);
```

--Condición para poner bandera de cero.

```
      zero<='1'
            when (alu_output_mux(7 downto 0) = "00000000")
            else '0';
```

--Resultado de la operación aritmética lógica efectuada por ALU.

```
      alu_result<=alu_output_mux(7 downto 0);
```

--Se realiza el cálculo de desplazamiento relativo a la posición del PC (salto --condicionado).

```
      branch_add<=pc_b4 + sign_extend;
```

```
--Se almacena en una señal externa para después ser usada en el módulo de
--instrucciones. Este salto se realiza en caso de que se cumpla una condición que
--depende del valor de la bandera de cero y del valor de la señal de control (ver
--modulo de instrucciones).
--La ejecución de la instrucción BEQ depende del valor de una resta de dos datos,
--de manera que cuando es cero, se concluye que son iguales y actualiza los valores
--que hacen que se cumpla la condición de salto.

add_result <= branch_add(7 downto 0);

--El siguiente proceso secuencial se encarga de efectuar las operaciones aritméticas
--y lógicas, dependiendo del código obtenido en la decodificación.
--Se activa al existir una cambio en los operandos a, b o en el código de
--decodificación.

process(alu_ctl,a,b)
begin
case alu_ctl is
        when "000" =>alu_output_mux <= a and b;
        when "001" =>alu_output_mux <= a or b;
        when "010" =>alu_output_mux <= a + b;
        when "011" =>alu_output_mux <= a - b;
        when "100" =>alu_output_mux <= "00000000";
        when "101" =>alu_output_mux <= "00000000";
        when "110" =>alu_output_mux <= a - b;
        when "111" =>alu_output_mux <= "00000000";
        when others =>alu_output_mux <="00000000";
end case;
end process;
end alu_arch;
```

Memoria de datos

En este módulo se describe el funcionamiento de la memoria de datos (ver Fig. 8.7). Al igual que la memoria de programa se crea una instancia de un modelo de memoria existente en el CAD que se muestra en el Apéndice B.

Fig. 8.7

La descripción de este módulo define el funcionamiento de una memoria RAM. Únicamente se realiza la instancia del modelo de memoria y se conectan sus señales con las señales del sistema MIPS.

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

entity memoria_datos is
   port (
      clk: in std_logic;
      rst: in std_logic;
      read_data: out std_logic_vector (7 downto 0);
      address: in std_logic_vector (7 downto 0);
      write_data: in std_logic_vector (7 downto 0);
      memread: in std_logic;
      memwrite: in std_logic
   );
end memoria_datos;

architecture memoria_datos_arch of memoria_datos is
signal mod_write:std_logic;

component mem_data
  port(
    a: in std_logic_vector(7 downto 0);
    do: out std_logic_vector(7 downto 0);
    di: in std_logic_vector(7 downto 0);
    wr_en: in std_logic;
    wr_clk: in std_logic);
end component;
begin
instance_name : mem_data port map
(a => address,
 do => read_data,
 di => write_data,
 wr_en => mod_write,
 wr_clk => clk );

--La habilitación de la memoria depende del valor que tome la siguiente señal. La
--negación en la señal de reloj sirve para retardar la señal mientras se toma la
--dirección y el dato.
mod_write<=memwrite and (not clk);
end memoria_datos_arch;
```

SIMULACIÓN

Una vez que se describió el funcionamiento del sistema y se realice la síntesis, podemos efectuar una simulación del procesador.

De esta forma se diseñan dos programas que ejecutará el procesador. Estos programas contienen las instrucciones básicas de manera que se pueda analizar el resultado de la simulación de manera clara.

En el primer programa se desea que realice lo siguiente:

1. Cargar el dato de la localidad "0" de la memoria de datos y almacenarlo en el registro de propósito general con dirección "1".
2. Cargar el dato de la localidad "1" de la memoria de datos y almacenarlo en el registro de propósito general con dirección "2".
3. Cargar el dato de la localidad "2" de la memoria de datos y almacenarlo en el registro de propósito general con dirección "4".
4. Sumar el contenido de los registros "1" y "2" y guardarlo en el registro "3".
5. Almacenar el dato del registro "3" en la localidad de memoria de datos con dirección "2".
6. Verificar el funcionamiento de salto condicionado comparando el contenido del registro "3" con el contenido del registro "4", y en caso de ser iguales que regrese al inicio del programa.

En la Tabla 8.2 se puede ver el programa en código nemotécnico, binario y su equivalente hexadecimal.

Programa	Instrucción en bits según formato.	Hexadecimal
Lw $1,0	100011 00000 00001 0000000000000000	8C010000,
Lw $2,1	100011 00000 00010 0000000000000001	8C020001,
Lw $4,2	100011 00000 00100 0000000000000002	8C040002,
Add $3,$1,$2	000000 00001 00010 00011 00000 100000	00221820,
Sw $3,3	101011 00000 00011 0000000000000011	AC030003,
Beq $3,$4,-5	000100 00011 00100 1111111111111010	1064FFFA

Tabla 8.2

Con la finalidad de comprobar el funcionamiento del salto condicional, la memoria de datos contiene inicialmente la información mostrada en la Tabla 8.3.

Localidad	Contenido
00	09
01	05
02	0E

Tabla 8.3

Con ésta información es posible realizar la simulación. (Ver Fig. 8.8.)

Las señales de entrada son la de reloj y la de reset. Inicialmente se activa la señal de reset y se mantiene hasta que haya un flanco positivo de la señal de reloj. De esta forma se inicializa el contador de programa y los registros de propósito general.

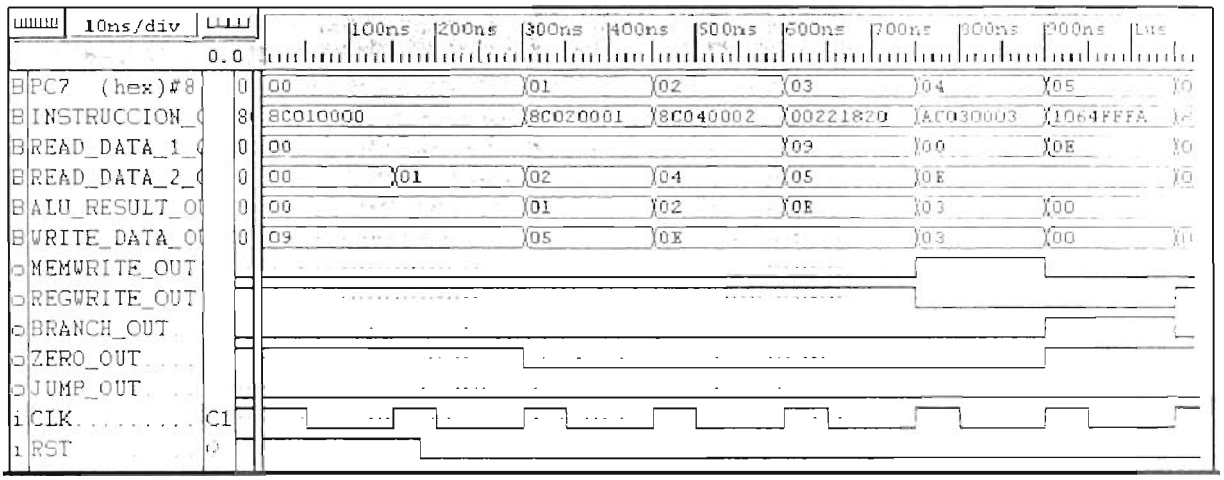


Fig. 8.8

El segundo ejemplo es muy semejante al anterior, sólo que en lugar de efectuar una suma, se realiza una resta, y en lugar del salto condicionado, efectuará un salto incondicional al inicio del programa. (Ver Tabla 8.4.)

Programa	Instrucción en bits según formato.	Hexadecimal
Lw $1,0	100011 00000 00001 0000000000000000	8C010000
Lw $2,1	100011 00000 00010 0000000000000001	8C020001
Sub $3,$1,$2	000000 00001 00010 00011 00000 100010	00221822
Sw $3,2	101011 00000 00011 0000000000000010	AC030002
Jmp 0	000010 00000000000000000000000000	08000000

Tabla 8.4

Los resultados de la simulación se aprecian en la Figura 8.9.

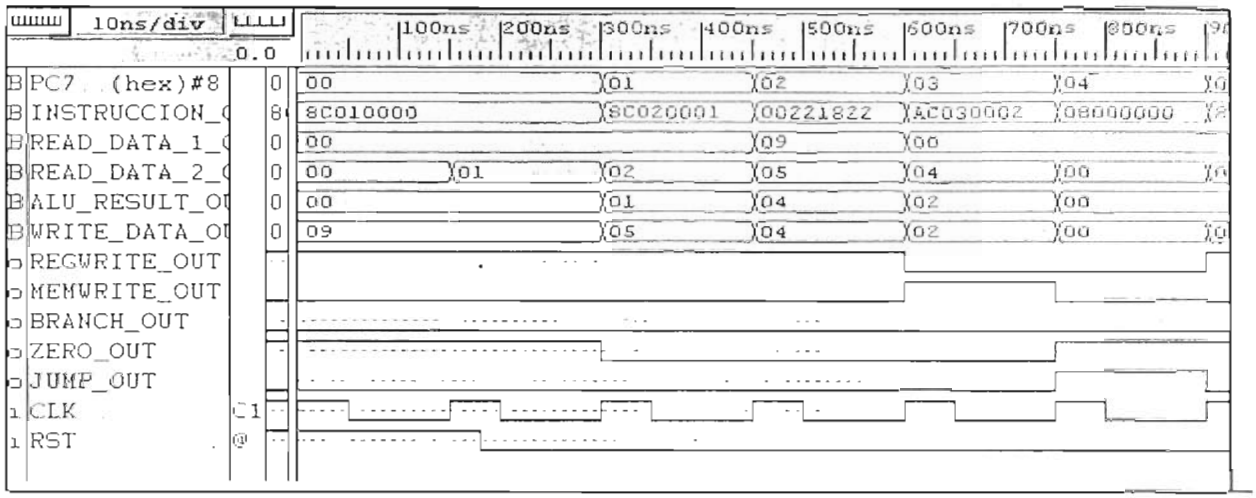


Fig. 8.9

9. CONCLUSIONES

De las experiencias obtenidas al utilizar VHDL en el diseño de sistemas digitales podemos concluir los siguientes puntos:

- El diseñar en base a niveles de jerarquía permite simplificar en gran medida la descripción del sistema. De esta forma podemos centrar la atención en el funcionamiento de los módulos individuales y al final únicamente realizar una interconexión entre ellos.
- La flexibilidad de manejar el lenguaje en diferentes niveles de abstracción permite elaborar diseños claros y comprensibles para especificación y simulación.
- El utilizar un método de desarrollo "top down", partiendo de un nivel de lenguaje RTL, permite centrar la atención en la funcionalidad del sistema, dejando abierta la posibilidad de implementarlo.
- El proceso de síntesis no está estandarizado, por lo tanto, el diseño para implementación dependerá del software utilizado.

APÉNDICE A

¿Cómo empezar en VHDL?

Los componentes se describen definiendo una entidad y una arquitectura:

- Entidad: En esta sección se declaran los puertos de entrada y salida que utiliza el componente, con esto se define la interfaz de interconexión con el sistema. En VHDL se realiza de la siguiente manera:

Entity *nombre_entidad* **is**
port (nombre_señal : modo tipo);
end *nombre_entidad*;

Donde:

- *nombre_entidad*: es el nombre del componente,
- *nombre_señal*: es el nombre del elemento de E/S del puerto,
- *modo*: define de qué forma opera la interfaz de la señal con el sistema,
- *tipo*: es el tipo de información que contiene la señal.

Los modos de interfaz de las señales son los siguientes:

In	Define señal de lectura.
Out	Define una señal para escritura.
Inout	Define señales bidireccionales.
Buffer	Define una señal de escritura y que puede volverse a leer.

Las opciones de tipo se describen más adelante.

- Arquitectura: el funcionamiento interno del componente puede estar conformado por funciones, operadores, o bien, en forma estructural. Una arquitectura siempre debe estar ligada a una entidad y se declara de la siguiente forma:

Architecture *nom_arq* **of** *nombre_entidad* **is**
sección_declarativa
begin
descripción_funcionamiento
end *nom_arq*;

Donde:

- *nom_arq*: es el nombre de la arquitectura,
- *nombre_entidad*: es el nombre de la entidad con la que se relaciona la arquitectura,
- *sección_declarativa*: se declaran objetos, tipos, componentes, etc.,
- *descripción_funcionamiento*: describe el funcionamiento del componente.

En VHDL, el definir objetos significa crear instancias de los elementos básicos que son las señales, variables y constantes.

Los objetos tienen un clase y un tipo, de manera que la clase define si es una señal, una constante o una variable y el tipo indica las características de los valores que puede contener.

Clases:

- **Señales**: son los objetos más importantes al describir circuitos electrónicos, ya que su valor cambia en relación al tiempo en forma simultánea con otras señales y representan la conexión con otros componentes. La asignación de información a señales se realiza mediante el símbolo "<=".
- **Variables**: son utilizadas en los procesos secuenciales, su valor puede cambiar en forma instantánea a diferencia de las señales en las que existe un retardo, además de que no pueden transferir información fuera de la estructura secuencial. Su comportamiento es el mismo que en los lenguajes de alto nivel. La asignación de información se realiza mediante el símbolo ":=".
- **Constantes:** su valor una vez definido no puede cambiar, el valor constante que representará se asigna con el símbolo ":= ".

En la entidad, al definir la interfaz de E/S, los objetos siempre son señales, por lo que no es necesario especificar la clase. Por otro lado, al declarar un objeto en la arquitectura siempre se debe indicar su clase.

Los tipos de datos más importantes definidos en el estándar para el VHDL se describen enseguida:

- **Std_ulogic**. Este tipo de datos puede tomar alguno de los siguientes valores:

'Z'	Alta impedancia
'0'	Valor 0
'1'	Valor 1
'X'	Desconocido
'U'	No inicializado
'-'	Valor cualquiera
'L'	Reacción en 0
'H'	Reacción en 1
'W'	Reacción desconocida.

- **Std_logic**. Es un subtipo del Std_ulogic por lo que puede tomar los mismos valores, la diferencia es que el std_logic es un tipo resuelto en la cuestión de manejar una señal, en la que tiene una asignación simultánea en diferentes operaciones. Por ejemplo:

```
Sig<=a when selec='0' else 'Z';
 Sig <=b when op='1' else 'Z';
```

  Std_logic soluciona el conflicto en caso de que las dos condiciones fueran ciertas y decide cuál señal usará, la decisión la toma con base a una función predeterminada por el estándar VHDL.

- **Bit**. El tipo bit solo puede manejar los valores de "1" ó "0".
- Integer. Define el dato como tipo entero y podemos definir un rango de operación, los límites máximos dependen del software utilizado.
- **Boolean**. Este tipo de datos puede tomar los valores de falso o verdadero.

- **Std_logic_vector**. Es utilizado para definir datos tipo std_logic en forma de vectores, por ejemplo:

  a: in std_logic_vector(3 downto 0)

- Bit_vector. Define un vector de datos tipo bit, por ejemplo.

  a: in bit_vector(3 downto 0)

En los datos tipo vector, su tamaño se pueden declarar de 2 formas, por ejemplo

a: in bit_vector(3 downto 0)

o bien

a: in bit_vector(0 to 3)

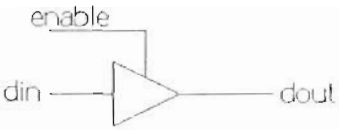
En la primer opción, el bit más significativo será último de la izquierda y en la segunda opción será el primero de la derecha, por lo que se debe tener cuidado al mezclar ambas formas.

El VHDL permite crear nuestros propios tipos de datos en la arquitectura y al declarar un objeto se debe de especificar su clase, por ejemplo:

```
Architecture nom_arq of nombre_entidad is
type tam_bus is array (31 downto 0) of std_logic;
signal dato:tam_bus;
signal s:integer range 0 to 15;
 begin
        <descripción_funcionamiento>
end nom_arq;
```

En la descripción en VHDL podemos introducir comentarios mediante el símbolo "--".

A continuación se muestra un ejemplo sencillo de cómo se describiría un buffer de tercer estado:



-- declaración de señales de e/s del componente

```
Entity buff3e is
  port ( din,énable : in std_logic;
         dout    :out std_logic);
end buffer3e;
-- descripción del funcionamiento interno
Architecture buff_funcion of buffer3e  is
begin
      dout <= din when enable='1' else 'Z';
end buff_funcion;
```

Generalmente, el VHDL no distingue diferencia entre mayúsculas y minúsculas, excepto en el caso de que la letra este entre comillas simples, que debe ser siempre mayúscula y se utiliza para designar algún valor de la señal, por ejemplo, 'Z' que representa un estado de alta impedancia.

En el VHDL encontramos diversos tipos de datos, que no son combinables entre si en forma directa, en estos casos se requiere del uso de funciones de conversión, a continuación muestran las de uso más frecuente:

- To_stdlogicvector(bit_vector): convierte un vector de tipo bit al tipo std_logic_vector.
- To_std_logicvector(std_ulogic_vector): convierte el tipo std_ulogic_vector a std_logic_vector
- Conv_std_logic_vector(integer,bits): convierte un entero a un vector tipo std_logic.
- Conv_integer(std_logic_vector): convierte el vector tipo std_logic a un entero.

Ejemplo:

```
Entity ej_1 is
  port ( a : in integer range 0 to 7:
         s :out std_logic_vector(0 to 3));
end ej_1;
```

```
Architecture ej_1_conv of ej_1 is
begin

        s<=conv_std_logic_vector(a, 3);

end ej_1;
```

El ejemplo anterior muestra una conversión de un dato de entrada en formato tipo entero que se convierte al vector tipo std_logic para que se pueda realizar la asignación. El rango del entero está entre 0 y 7, al convertirlo se indican el número de bits con los que pueden ser representados los números que estén dentro de ese rango. El uso de estas funciones no repercute en el resultado final de la síntesis.

Tipos de operadores

Al diseñar en VHDL debemos tener en cuenta los operadores que son permitidos. En la siguiente Tabla se muestran los operadores utilizados en el VHDL:

Aritméticos		**Relacionales**		**Lógicos**
Operador	Símbolo	Operador	Símbolo	Operador
Suma	+	Igual	=	And
Resta	-	Diferente	/=	Or
Multiplica	*	Menor que	<	Not
Divide	/	Mayor que	>	Nand
Valor absoluto	Abs	Menor o igual que	<=	Nor
Resto	Rem	Mayor o igual que	>=	Xor
Módulo	Mod			Xnor
Exponente	**			
Concatenación	&			

Simultaneidad

Al describir el funcionamiento del componente sabemos que sus operaciones internas se realizarán en forma paralela, lo que quiere decir que su ejecución se realiza en forma simultánea. Las siguientes instrucciones son las de uso más frecuente para estos procesos:

- Asignación de señales. Es un proceso indispensable del funcionamiento simultáneo, en el que se le da un valor a una señal que puede ser la salida del componente o bien una señal de uso interno. Cuando existen distintas asignaciones todas se realizan en forma simultánea. La asignación de señales se hace utilizando el símbolo "<="; por ejemplo, si se describen las siguientes operaciones:

```
a<=not (b and c);
d<=b or c;
```

las señales "a" y "d" obtendrán su resultado al mismo tiempo, por lo que no importa el orden en el que se describan.

- **When else**. Con esta estructura podemos escoger el valor que será asignado a la señal de salida dependiendo del resultado de una condición dada, que puede ser controlada con diferentes señales en el caso de utilizar varias estructuras a la vez, como se muestra en el siguiente ejemplo:

```
S<=a when sel = "00" else
      b when sel_2= "01" else c;
```

Como se aprecia en el ejemplo, la estructura debe finalizar con un else debido a que siempre se debe asignar un valor a la señal.

- **With select**. Con esta estructura podemos asignar diferentes valores a la señal de salida dependiendo del valor que tome la señal de selección, que en este caso es única a diferencia del caso anterior, por ejemplo :

```
With sel select
```

```
S<= a when "00",
        b when "01",
        c when "10",
        d when others;
```

Otra instrucción que se realiza en forma simultánea es "Process" en la que están incluidas las operaciones secuenciales.

Proceso secuencial

Una estructura secuencial se especifica de la siguiente forma:

```
Process (lista sensitiva)
<sección declarativa>
begin
        <descripción_del funcionamiento>
end process;
```

En la lista sensitiva se incluyen las señales que se utilizan en el proceso secuencial, de manera que un evento de cambio en su valor activa la ejecución del proceso.

Las principales instrucciones que pueden utilizarse dentro de un proceso secuencial son las siguientes:

- Asignación a variables, ":=".
- Asignación a señales, "<=".
- Cuando tenemos múltiples opciones de selección para asignar algún valor podemos utilizar la siguiente instrucción:

```
Case <expresion> is
        When <opcion_1> => <instrucciones>;
        When <opcion_1> => <instrucciones>;
        .....
        when others => <instrucciones>;
end case;
```

El uso de others es opcional.

- Instrucción condicional:

```
If <condición> then <instrucciones>
 elsif <condición> then <instrucciones>
Else
 <instrucciones_secuenciales>
end if;
```

El "elsif" es opcional y puede haber varios a la vez, de manera que podemos crear múltiples condiciones; "else" también es opcional pero sólo puede haber uno y se cumple cuando la condición del "if" y las de "elsif" son falsas.

- Wait until, es otra instrucción muy utilizada en procesos síncronos de la lógica secuencial, la cual espera a que se cumpla la condición para continuar con la ejecución de las instrucciones .

APÉNDICE B

Modelos de memorias predefinidos por el programa de CAD

La descripción del siguiente modulo corresponde a una memoria ROM (memoria de programa).

```
-------------------------------------------------
-- logiblox rom module "mem_prog"
-- created by logiblox version c.16
-- attributes
--    modtype = rom
--    bus_width = 32
--    depth = 256
--    memfile = mem_prog1
--    trim = false
--    style = max_speed
--    use_rpm = false

library ieee;
use ieee.std_logic_1164.all;
library logiblox;
use logiblox.mvlutil.all;
use logiblox.mvlarith.all;
use logiblox.logiblox.all;

entity mem_prog is
  port(
    a: in std_logic_vector(7 downto 0);
    do: out std_logic_vector(31 downto 0));
end mem_prog;

architecture sim of mem_prog is
  signal start_pulse: std_logic := '1';
  type mem_data is array (255 downto 0) of std_logic_vector(31 downto 0);
begin
  process
  variable vd: mem_data;
  variable first_time: boolean := true;
  begin
    if (first_time) then
      vd(0) :=
('1','0','0','0','1','1','0','0','0','0','0','0','0','0','0','1','0','0','0','0','0','0','0','0','0','0','0','0','0','0');
      vd(1) :=
('1','0','0','0','1','1','0','0','0','0','0','0','0','0','1','0','0','0','0','0','0','0','0','0','0','0','0','0','0','1');
```

```
    vd(2) :=
('0','0','0','0','0','0','0','0','0','0','1','0','0','0','1','0','0','0','0','1','1','0','0','0','0','0','1','0','0','0','1','0');
    vd(3) :=
('1','0','1','0','1','1','0','0','0','0','0','0','0','0','1','1','0','0','0','0','0','0','0','0','0','0','0','0','0','1','0');
    vd(4) :=
('0','0','0','0','1','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0');
    vd(5) :=
('0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0').
          .
          .

    vd(254) :=
('0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0'),
    vd(255) :=
('0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0','0').
     first_time := false;
   end if;
   if (mvlvec_not01(a)) then
    do <=
('x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x','x'
);
   else
    do <= vd(mvlvec2int(a));
   end if;
    wait on a, start_pulse;
 end process;
end sim;
```

La descripción del siguiente modulo corresponde a una memoria RAM (memoria de datos).

```
-------------------------------------------------------
-- logiblox sync_ram module "mem_data"
-- created by logiblox version c.16
-- attributes
--    modtype = sync_ram
--    bus_width = 8
--    depth = 32
--    memfile   mem_data
--    style = max_speed
--    use_rpm = false

library ieee;
use ieee.std_logic_1164.all;
library logiblox;
use logiblox.mvlutil.all;
use logiblox.mvlarith.all;
```

```
use logiblox.logiblox.all;

entity mem_data is
  port(
    a: in std_logic_vector(4 downto 0);
    do: out std_logic_vector(7 downto 0);
    di: in std_logic_vector(7 downto 0),
    wr_en  in std_logic;
    wr_clk: in std_logic);
end mem_data;

architecture sim of mem_data is
  signal start_pulse: std_logic := '1';
  type mem_data is array (31 downto 0) of std  logic  vector(7 downto 0),
begin
  process
  variable vd: mem_data;
  variable first_time: boolean := true;
  begin
    if (first_time) then
      vd(0) := ('0','0','0','0','1','0','0','1');
      vd(1) := ('0','0','0','0','0','1','0','1');
      vd(2) := ('0','0','0','0','0','0','0','0');
            .
            .
            .
      vd(30) := ('0','0','0','0','0','0','0','0');
      vd(31) := ('0','0','0','0','0','0','0','0');
      first_time := false;
    end if;
    if (wr_clk'event and stdbit2mvl(wr_clk)='1' and stdbit2mvl(wr  clk'last  value)  '0')
    and (wr_en='1') and (not mvlvec  not01(a)) then
      vd(mvlvec2int(a)) := stdvec2mvl(di);
    end if;
    if (mvlvec_not01(a) or
       (stdbit2mvl(wr_clk) = 'x')
       or (wr_clk'event and stdbit2mvl(wr_clk)='1' and stdbit2mvl(wr clk'last  value)  '0'
          and stdbit2mvl(wr_en) = 'x' )
      ) then
        do <= ('x','x','x','x','x','x','x','x');
    else
      do <= vd(mvlvec2int(a));
    end if;
      wait on a, di, wr_en, wr  clk, start_pulse;
  end process,
end sim,
```