Verilog: un enfoque basado en estudio de casos

Felipe Hans Mauricio Fernando

Septiembre, 2013

Prólogo

Indice

Prólogo **iii**

1 Introducción **1**
1.1 FPGA: *Field-Programmable Gate Array* . . . 1
1.1.1 Arquitectura de una FPGA . . . 1
1.1.2 FPGAs del Mercado: Comparación . . . 3
1.1.3 Alternativas a una FPGA: hardware programable . . . 5
1.1.3.1 Cuándo elegir por una FPGA . . . 6
1.2 Verilog: el lenguaje de descripción de hardware . . . 6
1.3 Verilog vs. VHDL . . . 7
1.4 Fuentes . . . 8

2 Verilog: el lenguaje **9**
2.1 Introdución . . . 9
2.2 Describiendo Hardware en Verilog . . . 9
2.2.1 Módulos . . . 10
2.2.2 Tipos de datos: *wire*, *reg*, constantes y *define* y *parameters* . . . 13
2.2.2.1 Wires . . . 13
2.2.2.2 Registros . . . 15
2.2.2.3 Vectores . . . 16
2.2.2.4 Literales numéricos . . . 17
2.2.3 Definiciones . . . 18
2.2.4 Parámetros . . . 19
2.2.5 Bloque *always* . . . 21
2.2.5.1 Asignaciones: bloqueante y no bloqueante . . . 22
2.2.6 Estructuras de decisión . . . 25
2.2.6.1 *if-else* . . . 26
2.2.6.2 *case* . . . 27
2.2.6.3 El operador *?* . . . 28
2.2.7 Diseño estructural y comportamental . . . 29
2.3 Testbenches: simulando el hardware . . . 29
2.3.1 Bloques *initial* . . . 30

2.3.2 Sentencias *delay* 31
2.3.3 El *reloj perpetuo* 33
2.4 Ejemplo: diseñando multiplexores 34
2.5 Ejemplo: Diseñando contadores 38
2.6 Fuentes 41

3 Diseño de circuitos combinacionales 43

4 Diseño de circuitos secuenciales 45
4.1 El Flip-Flop 45
4.2 Maquinas de Moore y Mealy 50
4.2.1 Diagramas de estado 51
4.3 Describiendo Circuitos Secuenciales 52
4.4 Ejemplo: Cerradura con clave 57
4.5 Ejemplo: Registros de corrimiento 63
4.6 Ejemplo: Decodificador de código Morse 68
4.7 Ejemplo: Codificador rotatorio 80
4.8 Fuentes 88

5 Mi primer diseño 89

6 Buenas prácticas en Verilog 91

7 El Multiplexor y la ALU 93

8 Encriptación y Desencriptación 95

9 Pulsadores, displays, switches y leds 97

10 Puertos: Serial, VGA, PS2, Ethernet y USB 99

11 Robots (motores) 101

12 El ascensor 103

13 Casos avanzados 105

Indice de figuras

1.1 Arquitectura de una FPGA. 2
1.2 Tarjeta de desarrollo Anvyl. 4

2.1 Síntesis de *assigns*. 15
2.2 Resultados de testbench para compuerta *and* 32
2.3 Multiplexor 2x1. 34
2.4 Resultados de testbench para multiplexor 2x1. 36
2.5 Multiplexor 4x1. 36
2.6 Resultados de testbench para multiplexor 4x1. 38
2.7 Resultados de testbench para multiplexor 2x1. 40

4.1 Latch SR 45
4.2 Gated Latch sr NAND 47
4.3 Latch D 47
4.4 Flip Flop D 49
4.5 Flip Flop D con preset y reset 50
4.6 Diagramas de máquinas de estado. 51
4.7 Ejemplo de diagrama de estados para máquina de Moore . . 52
4.8 Ejemplo de diagrama de estados para máquina de Mealy . . . 52
4.9 Diagrama Circuito Secuencial 53
4.10 Diagrama de estados utilizando Mealy 58
4.11 Conección de la señal rst en los flip-flops del registro para dejar el registro en el estado E1 61
4.12 Esquema para visualizar la conexión de las señales reset y preset 63
4.13 Ejemplo de registro de desplazamiento. 64
4.14 Esquema circuital de registro de desplazamiento. 65
4.15 Resultados *testbench* registro de corrimiento. 66
4.16 Esquema circuital de reconocedor de secuencia. 67
4.17 Resultados *testbench* reconocedor de secuencia. 68
4.18 Diagrama de estados de detector de símbolos de código Morse. 69
4.19 Diagrama circuital para detector de símbolos de código Morse. 72
4.20 Testbench detector de símbolos de código Morse. 73
4.21 Diagrama de estados de traductor de código Morse. 75

4.22 Testbench decodificador de código Morse completo. 78
4.23 Diagrama de bloques de decodificador de código Morse. . . . 80
4.24 Esquema de un codificador incremental. 81
4.25 Diagrama de estados de codificador rotatorio. 82
4.26 Testbench codificador rotatorio. 84
4.27 Diagrama circuital de módulo contador. 86
4.28 Testbench codificador rotatorio junto con contador. 88

Indice de tablas

1.1 Tabla comparativa de FPGAs en el mercado. 4

2.1 Tabla de verdad de multiplexor 2x1 34

4.1 Tabla de verdad de un latch sr nand 46
4.2 Tabla de verdad de gated latch rs 47
4.3 Tabla de verdad de un latch-D 48
4.4 Tabla de verdad 62
4.5 Caracteres en código morse 74
4.6 Tabla de transición de estados para codificador rotatorio . . . 82

Capítulo 1

Introducción

1.1 FPGA: *Field-Programmable Gate Array*

Una FPGA es un dispositivo semiconductor compuesto por una matriz de bloques de lógica configurable (CLB - *Configurable Logic Block*) que permiten la programación de circuitos lógicos mediante un lenguaje de descripción de Hardware (HDL - *Hardware Description Language*).

Basicamente, se puede pensar en una FPGA como un chip que contienene todas las componentes básicas para "armar" un circuito digital, y será nuestro deber describir como conectar éstas para obtener un circuito que realice una funciń específica.

La FPGA es ampliamente utilizada en el desarrollo de hardware digital porque da la posibilidad de implementar, probar y rediseñar circuitos digitales sin tener que esperar el proceso de fabricación ni pagar por costos de rediseño.

1.1.1 Arquitectura de una FPGA

Cada FPGA está fabricada con un número limitado de recursos predefinidos con interconexiones programables que son los que describen la capacidad y la complejidad de circuitos que se pueden llegar a configurar. Entre estos recursos se pueden identificar los siguientes tipos:

a) Una **Celda Lógica (LC - *Logic Cell*)** es la unidad básica de la FPGA normalmente hecha de dos componentes: un flip-flop y una o más tablas de consulta (LUT - *Look-Up Table*). Se encarga de implementar una función lógica no muy compleja (comunmente de hasta cuatro entradas) pero que puede formar parte de diseños mucho más complejos al ser interconectadas.

b) La unión de dos celdas lógicas conforman lo que se denomina un **Slice**; de esta manera en cada Slice podemos encontrar dos flip-flops junto a múltiples tablas de verdad y multiplexores. El Slice es considerado

como el *micro bloque* de la FPGA. Su implementación varía según cada fabricante, pero suelen añadir funciones específicas (con respecto a las celdas lógicas) como la posibilidad de incorporar cadenas de acarreo.

c) El **Bloque de lógica configurable (CLB)** se compone por un grupo de bloques programables (normalmente 4 Slices) que proporcionan una expansión a las funcionalidades de los Slices. El CLB es considerado como el *macro bloque* de la FPGA. Es en el proceso de programación de la FPGA donde se define la función específica que debe cumplir cada CLB.

d) Los **Bloques de E/S (IOB - *Input/Output Block*)** son los encargados de otorgar a los circuitos diseñados señales tanto de entrada como de salida y así permitir la interacción con dispositivos digitales externos. Normalmente incorporan buffers además de protección ante la descarga electroestática (*EDS protection*).

e) En algunas FPGAs se proporcionan **bloques de hardware dedicado**, un ejemplo de estos bloques son los **Shift-registers**. Un shift-register esta compuesto por una cadena de flip-flops y un conjunto de compuertas lógicas que detectan una secuencia de valores binarios. Sin embargo, en una FPGA se implementan con algunas diferencias con el fin de ahorrar hardware sin afectar su funcionamiento.

f) Finalmente los **Componentes de interconexión** incorporan una estructura jerárquica de enrutamiento que es la que permite la conexión configurable entre CLB's e IOB's.

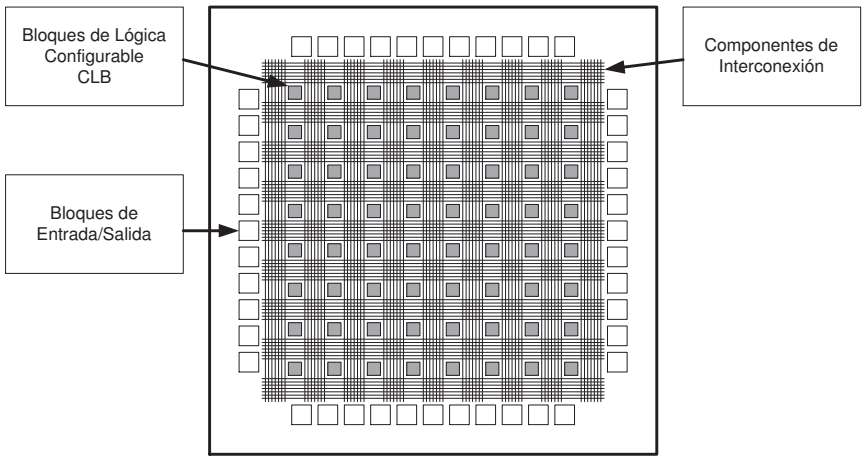


Figura 1.1: Arquitectura de una FPGA.

Si bien los lenguajes de descripción de hardware nos permitirán abstraernos de la arquitectura interna de la FPGA, resulta interesante saber cómo funciona la tecnologíía que vamos utilizar. Teniendo una noción básica del funcionamiento de la FPGA nos ayuda también a comprender las fortalezas y limitaciones de la plataforma.

1.1.2 FPGAs del Mercado: Comparación

Actualmente son tres los fabricantes más reconocidos de FPGA: *Xilinx*, *Altera* y *Actel*. Cada fabricante ofrece una variedad de FPGAs, con distintas capacidades.

Para el desarrollo de aplicaciones, existen placas de desarrollo que incorporan una FPGA junto a múltiples periféricos de uso común. Una de estas es, por ejemplo, la tarjeta de desarrollo del laboratorio: Anvyl (Figura 1.2).

Si bien es importante contar con los periféricos necesarios para nuestro proyecto, resulta más importante aún tener una FPGA que soporte nuestro diseño. Para poder escoger correctamente una FPGA hay que reconocer aquellas características a las que hay que tomar especial atención al momento de comparar cada modelo:

a) El **Número de celdas lógicas (LC)** puede considerarse como uno de los puntos más importantes a la hora de elegir una FPGA ya que afecta directamente a la complejidad del circuito que se puede llegar a configurar en la FPGA; por lo anterior, es necesario planificar y estimar el número de CLBs y, por consiguiente, el número de celdas lógicas que el circuito a diseñar pueda llegar a necesitar y conseguir una FPGA que sobrepase este número.

b) El **Número de pines** que posea la FPGA es otro punto importante a considerar ya que puede limitar las interacciones del circuito que se desea diseñar; precisamente por esto el diseñador debe determinar con anterioridad el número de entradas y salidas que deberá poseer el circuito digital que se desea desarrollar. Otro punto importante en relación a los pines es determinar si la FPGA permite implementar pines de entrada-salida (señales *inout*) y si son suficientes para el diseño del circuito.

c) La **Máxima frecuencia de reloj** de una FPGA es otra característica que tiene una importancia crucial ya que dependiendo del circuito a diseñar puede no ser suficiente para los propósitos del proyecto y no permitir el procesamiento veloz de las señales.

d) Finalmente los **Componentes periféricos** presentes en las placas de desarrollo tales como memorias, interfaces de entrada-salida o

Tabla 1.1: Tabla comparativa de FPGAs en el mercado.

Fabricante	**Modelo**	**LCs (miles)**	**Número de pines**	**Máxima Frecuencia de Reloj (MHz)**
Xilinx	**Spartan-6**	147	576	160
	Artix-7	215	500	660
	Kintex-7	478	500	700
Altera	**Stratix**	1000	1932	125
	Arria	190	1517	125
	Cyclone iv	150	896	133
Actel	**IGLOO**	600	620	160
	Axcelerator AX 250	154	248	350
	Axcelerator AX 1000	612	648	350

conectores de expansión pueden ser un punto decisivo dependiendo de los propósitos del proyecto.

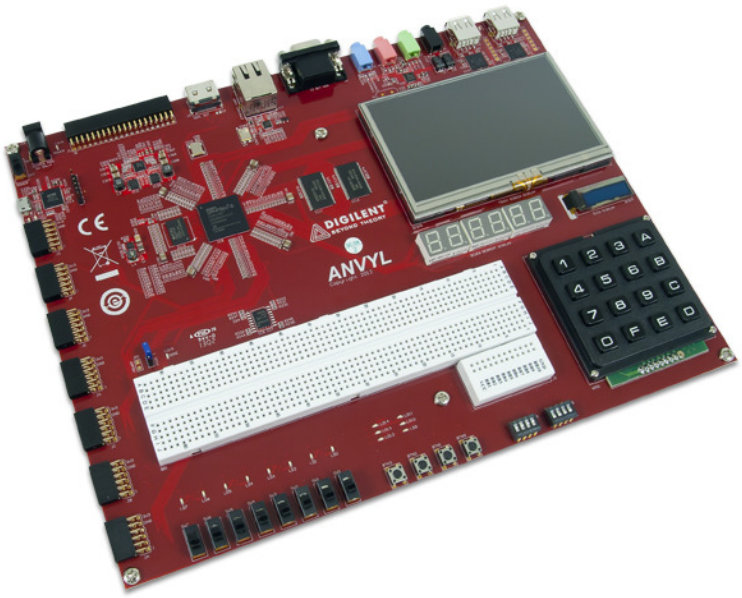


Figura 1.2: Tarjeta de desarrollo Anvyl.

En la Tabla 1.1 se puede observar una comparación de algunas de las FPGAs que ofrecen en el mercado, destacando las características de importancia mencionadas anteriormente.

1.1.3 Alternativas a una FPGA: hardware programable

En todo proyecto digital hay que decidir qué tipo de hardware programable utilizar para lograr procesar las señales digitales adecuadamente. Por esto es necesario conocer las distintas alternativas a una FPGA que ofrece el mercado reconociendo la aplicación más adecuada para cada una de ellas.

a) **Microcontrolador:**
El microcontrolador es uno de los circuitos integrados programables más utilizados debido a su facilidad de uso, bajo costo en la industria y muy bajo consumo. Es muy similar a una computadora ya que contiene una unidad central de procesamiento (CPU - *Central Processing Unit*), memoria y periféricos de entrada/salida. La principal diferencia que tiene respecto a la FPGA es que en el microcontrolador uno programa **software**, mientras en que la FPGA uno describe **hardware**. Normalmente utilizamos microcontroladores en proyectos donde hay "una sola" tarea principal o varias que sean posibles de atender en tiempo compartido.

b) **Procesador Digital de Señales - DSP:**
Un procesador digital de señales (DSP - *Digital Signal Processor*), al igual que el microcontrolador, ya posee un hardware diseñado por lo que sólo se debe programar el software. En general posee características muy similares a un microcontrolador pero la principal diferencia es que la velocidad de procesamiento en un DSP es mucho mayor a la de un microcontrolador, permitiendo así el procesamiento de señales análogicas en tiempo real. Es por esto que es muy utilizado en la industria de las telecomunicaciónes y en el procesamiento de audio/video.

c) **Arreglo Lógico Programable - PLA:**
Un arreglo lógico programable (PLA - *Programmable Logic Array*) es un tipo de dispositivo lógico programable (PLD - *Programmable Logic Device*) que se utiliza para implementar circuitos combinacionales mediante dos matrices programables: una matriz AND y una matriz OR. La principal desventaja del PLA es que sólo es posible programarlo una vez durante la fabricación debido a que se programa mediante fusibles y antifusibles. En general no poseen muy buen rendimiento en cuanto a consumo y a tiempos de propagación, es por esto que no soy muy utilizados en la actualidad de manera independiente.

d) **Lógica de Arreglos Programable - PAL:**
Una lógica de arreglos programables (PAL - *Programmable Array Logic*) es un PLD muy similar al PLA pero con la diferencia de que sólo la matriz AND es programable mientras que la matriz de OR es

fija e incorpora lógica de salida. Este cambio (que se realiza con el fin de superar las desventajas del PLA tales como los largos retardos de programación) se basa en el conocimiento de que cualquier función lógica puede expresarse como una suma de productos. Actualmente se puede encontrar dentro de otros dispositivos digitales como los CPLD.

c) **Dispositivo Lógico Programable complejo - CPLD:**
Un dispositivo lógico programable complejo (CPLD - *Complex Programmable Logic Device*) es un tipo de PLD que tiene una complejidad superior a la PAL pero inferior a una FPGA. Un CPLD comparte ciertas características de la FPGA como el que ambos están conformados por bloques lógicos similares a PLDs y ambos suelen contener miles de compuertas lógicas que permiten la implementación de complejos circuitos digitales. Por otra parte también comparte características con las PAL como el que ambos no requieren de una memoria ROM que almacene la configuración a implementar, permitiendo así su uso inmediato desde que se encienden. La CPLD puede ser útil para proyectos con circuitos digitales ya definidos y de una complejidad media.

1.1.3.1 Cuándo elegir por una FPGA

Los casos en que una FPGA conviene más que todas las alternativas recién vistas son proyectos donde:

- Se requiera un hardware para proósitos generales y no se tenga un circuito digital ya definido, por lo que es necesario desarrollarlo mediante prototipos y pruebas.
- Se requieren circuitos lógicos de gran complejidad donde cientos de miles de compuertas lógicas sean necesarias.
- El tiempo sea un factor determinante y se necesite una alta velocidad de procesamiento.
- El consumo de potencia no sea crítico.

1.2 Verilog: el lenguaje de descripción de hardware

Verilog es el lenguaje de descripción de hardware (HDL) más usado en la industria de semiconductores de la actualidad y, como todo HDL, permite el diseño, prueba e implementación de circuitos digitales.

Un lenguaje de descripción de hardware difiere de los lenguajes de programación convencionales debido principalmente a que se utilizan para **diseñar hardware**. Es por esto que en este tipo de lenguajes la ejecución de las sentencias no es estrictamente lineal, o dicho de otra forma, las instrucciones **no tiene un comportamiento secuencial**.

Tanto Verilog así como otros tipos de HDL son necesarios para el diseño de hardware digital ya que hoy en día la fabricación de circuitos de pruebas se ha reducido a la programación de los distintos tipos de hardware reprogramable, permitiendo un ahorro de tiempo y dinero.

1.3 Verilog vs. VHDL

Entre los lenguajes de descripción de hardware disponibles destacan dos: Verilog y VHDL. Aquí se realiza una breve comparación entre ellos.

a) **Verilog:**
Este lenguaje suele ser más simple de aprender para los ingenieros dada su similitud con la sintaxis del lenguaje de programación C y tiende a ser algo más conciso que VHDL. Verilog permite trabajar directamente con hardware de bajo nivel utilizando compuertas lógicas. Sin embargo puede ser un lenguaje no determinístico si no se emplean bien algunas reglas de programación (buenas prácticas) y al no ser un lenguaje fuertemente tipado puede llegar a compilar diseños que tienen un comportamiento distinto al esperado. No soporta tipos de datos personalizados y los aprendices suelen confundirse con las diferencias entre *wire* y *reg* así como las diferencias entre las asignaciones bloqueantes y las no-bloqueantes.

b) **VHDL:**
Por otro lado este lenguaje es fuertemente tipado, lo que conlleva a una compilación más compleja; pero una vez compilado es mucho más probable que se comporte como uno espera, es decir, VHDL tiende a ser un lenguaje más determinísta. Este lenguaje permite definir tipos de datos personalizados mejorando la comprensión. No obstante VHDL generalmente lleva a códigos muy extensos debido principalmente a que hay que definir y declarar cada modulo VHDL antes de ser usado. Es necesario definir conversiones de tipos para usar variables de distintos tipos en conjunto. Cabe resaltar que éste libro no cubrirá tópicos de descripción de hardware en VHDL.

1.4 Fuentes

- http://www.ni.com/white-paper/6983/es
- http://aguilarmicros.mex.tl/imagesnew2/0/0/0/0/2/1/4/2/9/6/PLDs.pdf
- http://electronicdesign.com/what-s-difference-between/what-s-difference-between-vhdl-verilog-and-systemverilog
- http://www.dc.uba.ar/materias/disfpga/2011/c2/descargas/Arquitectura.pdf
- http://www.xilinx.com/products/silicon-devices/fpga.html
- http://www.doc.ic.ac.uk/ wl/papers/08/kuon08survey.pdf

Capítulo 2

Verilog: el lenguaje

2.1 Introdución

Probablemente esté familiarizado con la idea de que al escribir software, lo que se realiza es una descripción de la secuencia de tareas necesarias para obtener un resultado deseado. Sin embargo, este enfoque no es el adecuado para el trabajo con lenguajes de descripción de hardware, ya que en estos no existe una estructura temporal rígida. Es decir, podremos tener tanto operaciones que ocurren concurrentemente como secuencialmente, según sea necesario. Es por esto que si se está acostumbrado a la programación de software, es fundamental realizar este cambio de mentalidad al realizar el diseño de hardware. Esto puede sonar más complicado de lo que realmente es. Para intentar dar una explicación más simple, se puede pensar que al describir hardware se especifica cómo se conectan las diferentes componentes en un protoboard.

En este capítulo se estudiarán los conceptos fundamentales de *Verilog*, para realizar un primer acercamiento a la descripción de hardware. Se verá también cómo probar nuestros diseños, mediante un *testbench*, para verificar su correcto funcionamiento previo a su implementación en una *FPGA*.

2.2 Describiendo Hardware en Verilog

Verilog es el medio que nos permitirá llevar a cabo la implementación de nuestros diseños. Con la ayuda de éste podemos abstraernos de la parte electrónica "dura" del diseño, para enfocarnos en la parte lógica. Éste, junto con las *FPGA*, nos permiten un desarrollo rápido de circuitos complejos.

El lenguaje en sí no es complicado y veremos en esta sección que sólo basta entender un puñado de los conceptos y palabras claves para poder describir un circuito. Se recomienda leer completamente esta sección debido a la fuerte vinculación que existe entre los conceptos básicos, para lograr una buena comprensión.

2.2.1 Módulos

Los módulos representan la estructura básica de un diseño en *Verilog* y son los que contienen la funcionalidad de nuestro diseño. Se definen para cada módulo puertos de entrada y salida, los cuales permiten el intercambio de información con su "exterior". Para obtener un mejor entendimiento de estos, en este punto se explicará la estructura de un módulo:

```
module <nombre_del_modulo>(<entradas>, <salidas>);
  input <entradas>;
  output <salidas>;

  ...

endmodule
```

- La sintaxis de un módulo, comienza con el keyword *module*, seguido por el nombre del módulo y entre paréntesis un listado de las entradas y salidas separadas por comas. Si bien no es una regla que se deben escribir primero las entradas y después las salidas en el listado, se recomienda realizarlo de esta forma, para mantener el orden. Se termina con punto y coma.

- Dentro del módulo, primero que todo se debe definir explícitamente la naturaleza del listado de entradas y salidas. Para las entradas se utiliza *input* y para las salidas *output*. Existen también los puertos *inout* que pueden actuar como entrada y salida a la vez.

- El resto del contenido del módulo corresponde a la descripción de los datos y funcionamiento del módulo. Esto puede incluir además la inclusión de otros módulos.

- Para simbolizar el fin de un módulo se utiliza la palabra clave *endmodule*

- Como consejo se sugiere siempre usar nombres descriptivos para los nombres de los módulos, entradas y salidas; y como práctica general al escribir código. Esto resulta no solo de utilidad para ayudar la comprensión del programa a alguien que lo lee por primera vez, sino que muchas veces se dá el caso que nosotros mismos olvidamos el funcionamiento del código que hemos escrito.

De las anteriores, una característica de importancia es la habilidad de declarar un módulo dentro de otro. Esto significará que una copia del módulo

declarado existirá junto con toda su funcionalidad dentro del módulo que estamos escribiendo. Esto se realiza dentro del módulo de alguna de las dos formas siguientes:

```
<nombre_del_modulo> <nombre_de_instancia>(<lista_de_puertos
    >);

<nombre_del_modulo> <nombre_de_instancia>(
  .<nombre_de_puerto1>(puerto1),
  .<nombre_de_puerto1>(puerto2),
  ...
);
```

En ambos casos se comienza por el nombre del módulo, que definirá el tipo de módulo, seguido por un nombre que sirve como identificador para esa instancia específica del módulo. Luego, entre paréntesis se detalla como se conectan los puertos.

En el primer caso, se deben listar las redes a las que se conecta cada puerto en el mismo orden en como fueron definidos al escribir el módulo a declarar.

Para evitar esta última regla, la cual podría dar problemas al haber una modificación del módulo a incluir, se utiliza la segunda forma, en donde los elementos del listado tienen el formato "*.<nombre_del_puerto>(<puerto>)*". Primero se especifica el nombre del puerto, antecedido por un punto; y luego a que se va a conectar.

Ambas formas no son compatibles y se debe optar por una o la otra, pero como siempre, se recomienda utilizar la segunda forma que ayuda a mantener el orden y el entendimiento.

Ejemplo 2.1

Para tener una mejor idea de cómo se realiza la declaración de los módulos, se presenta el siguiente código

```
module principal(clock, datos, resultado);
  input clock, datos;
  output resultado;

  componente comp_1(datos, resultado, datos);
  componente comp_2(
    .clk(clock),
    .entrada(datos),
    .salida(resultado));
endmodule
module componente(entrada, salida, clk);
  input entrada, clk;
  output salida;
  ...
endmodule
```

Aquí tenemos un módulo llamado *principal*, que tiene como entradas *clock* y *datos*, junto con una única salida llamada *resultado*.

Dentro de este módulo se tienen dos módulos del tipo *componente*, que tiene puertos *entrada*, *salida* y *clk*, a los que en ambos casos se han conectado las redes del módulo *principal datos*, *resultado* y *clock* respectivamente.

En el caso del módulo *comp_1*. No hemos utilizado los nombres de los puertos al declararlo, por lo que debemos conectar las entradas y salidas en el mismo orden en que fueron definidas en el módulo *componente*. No así, en el caso del módulo *comp_2*, utilizando los nombres de los puertos podemos reordenarlos para permitirnos, en este caso, declarar primero las entradas y luego las salidas.

La gran utilidad de esta característica es que nos permite construir módulos cada vez más complejos a partir de módulos básicos, esto es lo que se denomina un diseño jerárquico. Si bien siempre se podría implementar toda la funcionalidad necesaria dentro de un mismo módulo, las ventajas que nos ofrece este modelo son claras:

- En primer lugar, nos permite distribuir la funcionalidad de nuestro proyecto en módulos con funciones específicas, sin tener un único programa que realice todo. Además de ayudar a la comprensión del diseño, nos ayuda a aislar problemas, pudiendo probar individualmente cada parte.

- Al tener módulos con funcionalidad definida, nos permite reutilizarlos en futuros proyectos o incluso para poder utilizar múltiples y evitar repetir parte del código innecesariamente. Esto es muy similar al concepto de funciones en lenguajes de programación de software.

- Entre más general sea la función de cada módulo, mayor reusabilidad tendrá.

En general la estructura de nuestro diseño consistirá de un módulo principal dentro del cual se incluirán los módulos necesarios para completar la funcionalidad requerida. Esto es muy similar al concepto de conectar las diferentes componentes de un circuito.

2.2.2 Tipos de datos: *wire*, *reg*, constantes y *define* y *parameters*

Hasta el momento hemos definido el concepto de módulos y hemos dicho también que utilizamos sus puertos para conectarlos. Además sabemos que dentro de los módulos existirá manipulación de datos, pero no hemos explicado los medios por los cuales se realizan ninguno de estos actos. Para poder realizar esto, necesitamos conocer los tipos de datos presentes en *Verilog*. Se pueden dividir estos en dos tipos: redes (o *nets*) y *registers*. Dentro de las redes, estudiaremos los *wires*

En general, los datos solo pueden tomar 4 posibles valores:

- 0: Representa un cero lógico

- 1: Representa un uno lógico

- X: Representa un valor lógico no definido

- Z: Representa alta impedancia. Es de gran utilidad para la comunicación bidireccional, la cual no será tratada en este capítulo.

2.2.2.1 Wires

Los *wires* son un tipo de datos que no almacenan información, ni posee lógica, sino que su principal función es encargarse de la conectividad dentro de un módulo. Como su nombre lo dice, se puede pensar en su funcionamiento como aquel de un cable. Su valor dependerá de a que esté conectado o a que esté asignado; y reflejará inmediatamente en su valor cualquier cambio de esta componente.

Las definiciones de los *wires* se realizan sólo dentro de un módulo y consisten de la palabra *wire*, seguida por el nombre que se quiera dar a la red, terminado por punto y coma. Al no poseer memoria, los *wires* no pueden tener un valor inicial.

Para asignar un valor a un cable se utiliza el keyword *assign*, con la sintaxis:

```
assign <nombre_del_wire> = <expresion>;
```

En este caso la expresión puede corresponder a otro dato, o bien una expresión conformada a partir de operaciones sobre varios datos. En este punto podrá pensar que lo anterior entra en conflicto con la descripción original de un *wire* como un tipo de dato sin lógica, pero en el caso de existir una expresión que la requiera, se entiende que el circuito lógico está implícitamente definido por el *assign*, con el *wire* conectado a su salida.

Otra observación importante acerca de las asignaciones es que cada *wire* puede ser asignado solo una vez. Esto se debe a que las asignaciones son continuas, por lo que haber más de una llevaría a que el *wire* pueda potencialmente tener dos valores diferentes, lo que no está permitido. Si realizamos la analogía con a un cable real, este tiene el mismo potencial en toda su extensión, si le conectamos dos voltajes diferentes crearemos un cortocircuito.

También es importante notar que conectar un dato declarado como *wire* a un puerto de salida al declarar un módulo, es equivalente a asignarle un valor. Por lo tanto no podremos usar *assign* para este *wire*.

Por defecto, cualquier identificador que no tenga un tipo declarado es un *wire*. De esta forma los *input* y *output* son por defecto *wires*. En el caso de los *input*, estos no pueden ser de otro tipo.

Ejemplo 2.2

Analicemos el siguiente módulo

```
module asignaciones(A, B, C, N);
  input A, B, C;
  output N;

  wire D;

  assign D = A & B;
  assign N = D | C;
endmodule
```

Este módulo tiene entradas A, B, y C; y salida N. Dentro del módulo se creó el *wire* D. Usando *assign* le asignamos a D el valor del resultado de la operación *and* entre A y B, mientras que a la salida

N- que como dijimos es por defecto un *wire* y por tanto podemos asignarlo- le asignamos el resultado entre el *or* de *D* y *C*.

En este caso, por ejemplo, no podríamos realizar además la asignación *assign D = A ^ B;*, puesto a que ya hemos asignado una vez a *D*.

Si escribiéramos el módulo anterior como un circuito lógico obtendremos lo siguiente:

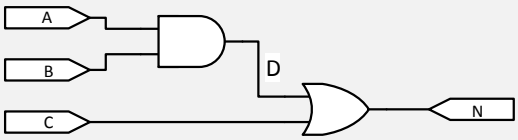


Figura 2.1: Síntesis de *assigns*.

en donde las compuertas lógicas son resultado de los *assigns*.

2.2.2.2 Registros

Los *registers* o registros, a diferencia de los wires, corresponden a un tipo de datos pueden mantener su estado, hasta que sean cambiados. De esta forma pueden representar un componente que posea memoria, o bien pueden no hacerlo, según como se les utilice.

La declaración de un registro sigue las mismas reglas que la de un *wire* con la diferencia de que se utiliza el keyword *reg*.

A diferencia de los *wires*, a los registros no se les puede dar un valor con *assign*, y su valor solo puede ser cambiado con una asignación (que no es sinónimo de utilizar *assign*)dentro de un bloque *always* o *initial*.

Un *output* puede ser declarado como un *register*, añadiendo el keyword *reg* justo despues del keyword *output* o declarando un registro con el mismo nombre y tamaño que el *output*.

Ejemplo 2.3

Los siguientes son ejemplos de declaraciones de un *reg*

```
reg enable;
output reg ready;
```

- En el primer caso tenemos simplemente un registro llamado *enable*.
- En el segundo tenemos una salida que a su vez se ha declarado como un registro.

2.2.2.3 Vectores

Hasta el momento hemos trabajado con *wire* y *reg*, trabajandolos siempre como bits individuales. Sin embargo esto puede resultar poco práctico cuando se necesita trabajar con señales que requieren manejar más información y por tanto necesitan una mayor cantidad de bits.

Verilog cuenta con vectores o buses, que nos permiten agrupar más de un *wire* o *reg* bajo un mismo nombre. De esta forma tenemos redes que pueden tomar una cantidad mayor de valores diferentes.

Para crear un vector basta especificar el bit mas significativo (MSB) y el menos significativo (LSB) después del tipo de red, con el formato *[msb:lsb]*, lo que definirá a su vez el tamaño del bus.

Ejemplo 2.4

Algunos ejemplos de vectores son:

```
wire [2:0] A; //Bus de 3 bits
wire [3:1] B; //Tambien posee 3 bits
reg  [15:0] data; //Registro de 16 bits
```

Notar que en los primeros dos casos se definen el MSB y LSB diferentes, pero se obtienen buses con las mismas características.

Los *wires* *A* y *B*, son ambos de 3 bits y por tanto, si los usáramos para representar números decimales, permiten la representación de números del 0 al 7 u ocho valores diferentes. Por otra parte, el registro *data*, es de 16 bits y nos permite representar 65535 o 2^{16} valores diferentes.

Es posible también seleccionar parte de un vector. Esto se realiza indicando entre paréntesis el bit o el rango de bits que se quiere seleccionar.

Por ejemplo si tenemos el siguiente *wire*: *wire [7:0] data*, correspondiente a un bus de 8 bits; data[5] representaría el bit 5 (comenzando desde el 0) de este, y data[3:0] representaría un bus de 4 bits formado por los bits 3, 2, 1 y 0.

También es posible crear vectores a partir de diferentes datos, utilizando concatenación. Para esto se escribe entre llaves { y }, los datos a concatenar separados por comas. A los vectores concatenados se les pueden asignar valores, que es equivalente a realizar la asignación de cada uno de sus componentes por separado; o se pueden utilizar dentro de una expresión.

2.2.2.4 Literales numéricos

En muchas ocasiones resulta útil dar un valor específico a un dato. Para esto utilizamos literales numéricos. En *Verilog*, para escribir literales numéricos especificamos tanto el tamaño en bits del número como su base. Esto se realiza según el formato:

<bits>'<base><número>.

Las bases permitidas son binaria, octal, decimal y hexadecimal; y cada una se representa por su inicial: *b*, *o*, *d* y *h* respectivamente.

Ejemplo 2.5

Consideremos los siguientes tres casos:

```
wire a [2:0] = 3'b111;
wire b [4:0] = 3'd9;
wire c [3:0] = 5'h11;
```

- Primero que todo, notar que la asignación de los wires se realiza de una manera diferente. También es posible asignarles un valor al mismo tiempo en que se declarar, sin necesidad de usar *assign*, agregando un signo = junto con una expresión
- En el caso del *wire* a. Se tiene un bus de 3 bits al que se le asigna el valor 111 en binario también de 3 bits. Por lo tanto no hay problema y todos los bits de a tendrán el valor 1 lógico.
- Para el *wire* c de 5 bits, se quiere asignar el valor 9 en decimal. Esto corresponde a 1001 en binario, por lo que en cuanto al tamaño del bus no habría problema. Sin embargo el literal numérico se declaró de tamaño 3 bits. En este caso se trunca el valor binario, con lo que b tomaría el valor de 00001, perdiéndose el bit más significativo de 9.

- El último caso es c. Para este se tiene el número 11 en hexadecimal, correspondiente a 10001 en binario. El literal numérico es de 5 bits, por lo tanto no hay pérdida de información a causa de éste. En este caso, sin embargo, el tamaño del *wire* es el limitante. Se tiene un bus de 4 bits al que se quiere asignar un valor de 5 bits. En este caso también se trunca el valor del literal numérico, obteniéndose 0001 como valor asignado al *wire*

2.2.3 Definiciones

Muchas veces un simple número no nos dice mucho acerca de su función en nuestro diseño. Los *defines* o definiciones nos permiten crear "etiquetas" con valores numéricos. Para esto se utiliza el siguiente formato:

'define <ETIQUETA><valor>

Notar que se utiliza un acento invertido antes de la palabra *define*.

En general la única regla para los *defines* es que se deben definir antes de utilizarlos. Por ésto, y también para mantener el orden, se suelen ubicar al comienzo del archivo, fuera del módulo.

Ejemplo 2.6

Considerar el siguiente código:

```
`define RED  2'b00
`define GREEN 2'b01
`define BLUE 2'b10
`define MASCARA 4'b1101

wire [1:0] color1 = `RED;
wire [1:0] color2 = `GREEN;
wire [1:0] color2 = `BLUE;
wire [3:0] dir = `MASCARA & entrada;
```

Hemos utilizado definiciones para definir las etiquetas *RED*, *GREEN* y *BLUE*, que luego hemos asignado a los *wires* para diferentes colores. Es claro como en este caso las etiquetas nos entregan más información contextual que el simple valor numérico.

Otro uso que se da aquí es para definir una máscara para datos de entrada. Para obtener el valor de *dir*, se hace un *and* entre la máscara y entrada. En este caso, además de dar información de la función del literal numérico, nos permite poder usar este valor multiples veces; y de haber necesidad de cambiarlo, solo se necesita cambiar la definición.

El propósito de los *defines* no es únicamente crear constantes, ni tampoco forman parte de nuestro código en sí, sino que corresponden a una directiva del preprocesador de texto, que le dicen que debe reemplazar cada ocurrencia de la etiqueta en el programa, por el valor especificado. De esta forma los *defines* permiten un aplicación mucho más amplia, pero es debatible si su uso más allá de la definición de constantes contribuye a una buena práctica. Las constantes serán el único uso que se dará a las definiciones dentro del contexto de este libro.

2.2.4 Parámetros

Los parámetros nos permiten generar módulos que se adapten a nuestras necesidades a partir de un módulo que defina un esquema o función general.

Básicamente un parámetro es un valor definido dentro de un módulo utilizando la palabra clave *parameter*, un nombre y un valor por defecto. En el caso de los parámetros, puesto a que no constituyen un tipo de dato, no requieren de un tamaño. Este valor podrá ser usado para la definición de datos dentro del módulo o dentro de expresiones

Al momento de declarar un módulo, se pueden cambiar los parámetros,escribiendo el caracter # seguido por los valores de los parámetros separados por comas en el orden en que fueron definidos, después de especificar el tipo de módulo:

```
<tipo> #(<valores>) <nombre_del_modulo>(<puertos>);
```

Si solo se desea cambiar el valor de ciertos parámetros, se puede realizar el listado de valores con el formato *.<parametro>(<valor>)*, idéntico a la forma usada para definir los puertos de un módulo.

Alternativamente se puede usar *defparam* en líneas separadas para definir el valor de los parámetros. La sintaxis es:

```
defparam <nombre_del_modulo>.<parametro> = <valor>;
```

Ejemplo 2.7

Miremos como ejemplo el siguiente módulo:

```
module detector(in, result);
  parameter size = 1;
  parameter number = 1'b1;

  input [size-1:0] in;
  output result;

  assign result = in == number;
endmodule
```

La función de este módulo es detectar un número específico en la entrada. Podemos ver que a la salida se le ha asignado el resultado de la operación == entre *in* y *number*. Este operador tiene como resultado 1 si ambos argumentos son iguales y 0 en otro caso.

En este caso se definieron 2 parámetros para el módulo: *size* y *number*. El parámetro *size* es utilizado para definir el tamaño en bits de la entrada, mientras que *number* se utiliza para realizar la comparación. Por defecto el módulo tiene una entrada de 1 bit y detecta el valor 1.

Ahora podemos invocar instancias de éste módulo con diferentes características:

```
detector det1(in1, out1);

detector #(.size(3), .number(3'd6)) det2(in2, out2);

detector det3(in3, out3);
defparam det3.size = 8;
defparam det3.number = 8'h1F;
```

El módulo *det1*, puesto a que no se especificaron valores para los parámetros será configurado con los valores por defecto. Con los otros dos módulos podemos ver las dos formas de pasar los parámetros. El módulo *det2* define un detector con una entrada de 3 bits, que detecta el número 6, mientras que el módulo *det3* corresponde a un detector de 8 bits, que detecta el número 31.

2.2.5 Bloque *always*

El bloque *always* es una estructura dentro de un módulo cuya ejecución es controlada por una lista de sensibilidad. Dentro de este bloque se realizarán las asignaciones a los registros.

La estructura de un bloque *always* es:

```
always @(<lista_de_sensibilidad>) begin
  ... //asignaciones
end
```

Se comienza con la palabra clave *always*, seguido del signo @ y un paréntesis que contiene una lista de eventos separados por *or* llamada la lista de sensibilidad que indican cuando se debe ejecutar el bloque. Los posibles eventos en esta lista son:

- Un dato del módulo como un *wire* o un *reg*, que causarán una ejecuación del bloque always ante cualquier cambio de estos.
- Se puede especificar el canto de subida de un dato de la forma *"posedge <dato>"*. Esto causa la ejecución del bloque always cuando el dato en cuestión pasa de 0 a 1, o lo que es equivalente, de bajo a alto.
- De igual forma, se puede especificar *"negedge <dato>"* para describir un canto de bajada, o lo que es lo mismo, una transición de alto a bajo.
- Alternativamente, se puede definir la lista de sensibilidad como el caracter * lo que simboliza que se debe evaluar el bloque ante cualquier cambio. Se utilizará esto al describir circuitos combinacionales en donde queremos que ante cualquier cambio de las variables involucradas, el resultado se refleje inmediatamente en la salida.

Ahora, dentro del bloque se tienen las asignaciones que consisten en un operador de asignación, una parte izquierda y una derecha.

La parte de la izquierda corresponde a los datos a asignar que siempre deben ser de tipo *reg*, mientras que el lado derecho corresponde a la expresión que determina el valor a asignar, que puede incluir tanto datos del tipo *reg* como del tipo *wire*.

En cuanto al tipo de asignación, el operador = denota una asignación bloqueante y <= una asignación no bloqueante. La diferencia entre ambos tipos será descrita en la sección siguiente.

La función de *begin* y *end* es simplemente agrupar las asignaciones dentro del bloque, similar a la función de las llaves ({ y }) en un lenguaje como C.

Ejemplo 2.8

Consideremos el siguiente módulo:

```
module xor(A, B, N);
  input [7:0] A, B;
  output [7:0] N;

  reg [7:0] N;

  always @(A or B) begin
    N = A ^ B;
  end
endmodule
```

En éste se tienen las entradas A y B de 8 bits; y la salida N también de 8 bits, pero declarada como un registro.

El bloque always tiene como eventos en la lista de sensibilidad a A y B, por lo que se evaluará cada vez que haya un cambio de A o B. En tal caso, se asignará a la salida N el resultado de la operación *xor* entre los bits de A y B.

Notar que nunca puede ocurrir el caso en que cambiando las entradas no se reevalúe la salida, por lo tanto el resultado de este módulo es puramente combinacional, de esta forma es equivalente a haber usado * como lista de sensibilidad:

```
module xor(A, B, N);
  input [7:0] A, B;
  output [7:0] N;

  reg [7:0] N;

  always @(*) begin
    N = A ^ B;
  end
endmodule
```

2.2.5.1 Asignaciones: bloqueante y no bloqueante

Se mencionó la existencia de dos tipos de asignaciones: bloqueante (=) y no bloqueante (<=). La diferencia entre ambas asignaciones es que, como

su nombre lo dice, la asignación bloqueante no permite que ocurran más asignaciones hasta que se completa la asignación actual. Por otro lado, la asignación no bloqueante permite múltiples asignaciones simultáneas, que sólo se hacen válidas al finalizar el evento que las provocó.

La diferencia entre ambas puede ser un tanto difícil de vislumbrar, pero se puede entender como que las asignaciones bloqueantes nos entregan una acción inmediata y por tanto importa el orden en que se ejecutan, mientras que las no bloqueantes, son evaluadas primero todas y luego asignadas, por la que no importa su orden.

¿Porqué es importante esta diferencia? Claramente al modelar circuitos combinacionales no hay diferencia, puesto a que el cambio de una variable causará que se reevalúen las demás llegando siempre a un mismo resultado. De esta forma el uso de una o la otra no significarán diferencias en el momento de la síntesis. A pesar de ésto por motivos de eficiencia en la simulación se prefiere utilizar siempre asignaciones bloqueantes para circuitos combinacionales.

La diferencia se hace más importante al describir circuitos con componentes que poseen "memoria", en que si importará el orden de ejecución. El problema que surge acá, es que, como se mencionó inicialmente, al describir hardware no se describe una secuencia de operaciones, sino que el funcionamiento de nuestro diseño. Esto significará que no se tiene completo control sobre el orden de las asignaciones, por tanto no podemos asegurar que al utilizar asignaciones bloqueantes estas se realizarán en el orden que queremos. Por esta razón, para estos casos se prefiere asignaciones no bloqueantes.

Ejemplo 2.9

Para lograr un mejor entendimiento del problema descrito anteriormente examinemos el siguiente módulo:

```
module asig_1(clk, rst, x, y)
  input clk, rst;
  output reg x, y;

  always @(posedge rst) begin
    x = 1'b0;
    y = 1'b1;
  end

  always @(posedge clk) begin
    x = y;
  end

  always @(posedge clk) begin
    y = x;
  end
endmodule
```

En este módulo, al haber un canto de subida de *rst*, los registros x e y toman los valores 0 y 1 respectivamente. Para este evento no hay problemas, puesto a que los valores son constantes y no dependen los unos de otros.

Supongamos ahora, que hubo un canto de subida de *rst* y ahora acaba de ocurrir un canto de subida de *clk*. Tenemos dos asignaciones que deben ocurrir, pero no sabemos en que orden suceden. Se pueden tener dos resultados:

- Ocurre primero la asignación de x: Con esto se tiene que a x se le asigna el valor 1, y luego a y se le asigna el valor de x que ahora es 1. Ambos registros terminan con el valor 1.
- Ocurre primero la asignación de y: El valor de x en ese momento es 0, por lo que a y se le asigna 0. Luego, al asignar a x el valor de y, este corresponde a 0. Ambos registros terminan en 0.

Se puede apreciar en este ejemplo como las asignaciones no bloqueantes llevan a ambigüedades en los resultados.

Analicemos ahora el mismo módulo, esta vez utilizando asignaciones no bloqueantes:

```
module asig_2(clk, rst, x, y)
  input clk, rst;
  output reg x, y;

  always @(posedge rst) begin
    x <= 1'b0;
    y <= 1'b1;
  end

  always @(posedge clk) begin
    x <= y;
  end

  always @(posedge clk) begin
    y <= x;
  end
endmodule
```

El comportamiento ante el canto de subida de *rst* es el mismo, se asigna 0 y 1 a x e y. Ahora veamos lo que sucede cuando ocurre un canto de subida de *clk*:

- Se evalúa primero el lado derecho de cada asignación no bloqueante. En el caso de la asignación de x, se tiene que el valor de y es 1. En el caso de la asignación de y, puesto a que la asignación de x aún no se hace efectiva, el lado derecho se evalúa en cero.
- La asignación se hace efectiva y x toma el valor 1 e y toma el valor 0.
- De haber otro canto de subida de *clk*, se invertirán los valores de la misma forma.

En este caso no hay ambigüedades y podemos decir a ciencia cierta que es lo que sucederá con las salidas.

2.2.6 Estructuras de decisión

Hasta ahora hemos visto cómo podemos diseñar módulos cuya salida sea una función de los diferentes datos dentro de éste. Ahora se introduce el concepto de condiciones, las que nos permitirán tomar una decisión u otra

dependiendo de si esta se cumple o no.

2.2.6.1 *if-else*

El *if-else* funciona igual y tiene la misma estructura que en cualquier lenguaje de programación:

```
if (<condicion>) begin
  ... //Se cumple la condicion.
end
else begin
  ... //No se cumple la condicion.
end
```

Si la condición del *if* se cumple se evalúa el contenido de este, de otro modo se evalúa el contenido del *else*. Como ya se mencionó *begin* y *end* cumplen la función de agrupar sentencias.

Los *if-else* sólo se pueden utilizar dentro de un bloque always o un bloque initial.

Ejemplo 2.10

Se presenta como ejemplo el siguiente módulo:

```
module suma_resta(A, B, ctrl, resultado);
  input [15:0] A, B;
  input ctrl;
  output reg [15:0] resultado;

  always @(*) begin
    if (ctrl == 0) begin
      resultado = A + B;
    end
    else begin
      resultado = A - B;
    end
  end
endmodule
```

Se tienen como entradas A y B de 16 bits y *ctrl* de 1 bit. Si *ctrl* es 0, a la salida *resultado* se le asigna el valor de la suma de A y B, de otra forma se evalúa el else, asignándose a la salida la resta de A y B.

2.2.6.2 *case*

La estructura *case* nos permite definir un comportamiento diferente según el valor de un dato.

```
case (<dato>)
  <caso_1>: <sentencia_1>;
  <caso_2>: <sentencia_1>;
  <caso_3>: <sentencia_1>;
  ...
  default: <sentencia_default>;
endcase
```

La estructura del *case* tiene las siguientes características:

- La estructura comienza con la palabra clave *case*, seguida por el dato a comparar entre paréntesis.
- Cada caso consta de una expresión con la cual comparar, junto con una sentencia a evaluar en caso de haber coincidencia entre el dato y el caso. Se puede definir además un caso *default* que se evaluará en caso de no haber coincidencia con ninguno de los casos.
- Es posible también tener multiples casos para una única sentencia. Para esto listamos los casos separados por comas, antes de los dos puntos.
- Se cierra el *case* con el keyword *endcase*.

Ejemplo 2.11

El siguiente ejemplo ilustra el uso del *case*:

```
module decodificador(in, out);
  input [2:0] in;
  output reg [7:0] out;

  always @(*) begin
    case (in)
      3'b000: out = 8'b00000001;
      3'b001: out = 8'b00000010;
      3'b010: out = 8'b00000100;
      3'b011: out = 8'b00001000;
      3'b100: out = 8'b00010000;
      3'b101: out = 8'b00100000;
      3'b110: out = 8'b01000000;
      default: out = 8'b10000000;
    endcase
  end
endmodule
```

La función de este módulo es entregar una salida con exactamente un bit en alto, correspondiente al indicado por la entrada. Para realizar esto, hacemos un *case* sobre la entrada y asignamos la salida correspondiente según el valor de esta.

2.2.6.3 El operador *?*

A diferencia del *if-else* y el *case*, el operador *?* forma parte de una expresión, permitiéndonos condicionar el resultado de ésta. Así, por ejemplo, es posible añadir condiciones a un assign.

La sintaxis de este operador es: *<condición>? <verdadero>: <falso>* Si se cumple la condición (es decir, si tal expresión es distinta de cero), se evalúa la expresión para el caso *verdadero*, de lo contrario se evalúa la expresión *falso*.

Como este es un operador, nos es posible combinarlo con otras operaciones para lograr describir operaciones mucho más complejas de manera resumida.

Ejemplo 2.12

Veamos cómo se construiria el módulo *suma_resta*, usado para ejemplificar el *else-if*, esta vez utilizando el operador *?*

```
module suma_resta(A, B, ctrl, resultado);
  input [15:0] A, B;
  input ctrl;
  output [15:0] resultado;

  assign resultado = ctrl == 0 ? (A + B) : (A − B);
endmodule
```

Podemos resumir todo el módulo a un simple *assign*. En este caso la condición es *ctrl == 0*. Si es verdadera, se evalúa *A + B*, de lo contrario se evalúa *A - B*.

2.2.7 Diseño estructural y comportamental

Una opción para implementar nuestro diseño en verilog, es describir nuestro diseño al nivel de compuertas lógicas. Esto se conoce como *diseño estructural*. Para este propósito *Verilog* define módulos genéricos para compuertas lógicas: *not* (equivalente a un inversor), *and*, *or*, *nand nor* y *xor*.

Estos se utilizan como módulos normales, con la diferencia de que no es necesario darles un nombre. El primer puerto corresponde a la salida de la compuerta, a la que sigue las entradas, que pueden ser tantas como se desee. La única excepción a esta regla es el *not*, que sólo tiene una entrada.

Si bien es posible realizar un diseño de esta forma, resulta poco práctico y no aprovecha al máximo la capacidad de abstraerse de los niveles bajos de diseño.

Otra opción es el *diseño comportamental*. El cual consiste no en describir directamente la estructura del circuito, sino más bien en describir las funciones del circuito, lo que resulta mucho más intuitivo.

El diseño estructural comprende el uso de *assign*, bloques *always* y estructuras de decisión. Con éstos, podemos escribir expresiones que permiten obtener el resultado o comportamiento deseado, sin preocuparnos mayormente por su implementación circuital.

2.3 Testbenches: simulando el hardware

Sucede muchas veces que pensamos que nuestros diseños han sido correctamente realizado, sin embargo, al momento de probarlos en la *FPGA* vemos como nada funciona según lo esperado. En vez de realizar una gran cantidad de pruebas iterativas para descubrir el problema, observando de manera indirecta los procesos que hemos modelado, por lo general resulta

mucho más facil hallar el problema por medio de simulaciones. Esto se realiza por medio de un *testbench*

Como dijimos, los módulos son las estructuras básicas en *Verilog* por tanto un *testbench* también lo será. En general estos tienen dos partes importantes: el módulo a probar y la lógica para la generación de señales de prueba.

La estructura general de un testbench es:

```
module <nombre_del_testbench>();
  reg <entradas>;
  wire <salidas>;

  <modulo_a_probar> <nombre de instancia>(puertos);

  initial begin
    ... //Sentencias para generar senales de prueba
  end
endmodule
```

Podemos ver que este es un módulo sin entradas o salidas. Esto es porque el objetivo de éste no es formar parte de algún otro módulo, ni ser implementado en una *FPGA*. Su objetivo es netamente la simulación en cuyo caso no necesitamos puertos para observar lo que sucede con las señales en su interior.

Dentro del módulo del *testbench* tenemos una instancia del módulo a probar, a cuyos puertos de entrada se han conectado *registers*, ya que son estos los datos que querremos modificar para causar un efecto en el módulo; mientras que a la salida se conectan *wires*, para poder observar la respuesta de estos.

Lo ideal es generar señales de prueba de forma de probar nuestro módulo de manera exhaustiva, considerando todos los posibles escenarios.

2.3.1 Bloques *initial*

Los bloques *initial* han sido nombrados anteriormente un par de veces, pero no se han introducido hasta ahora. Estos corresponden a bloques que sólo son evaluados una vez al comienzo del programa. Si bien esto puede sonar bastante útil la razon para no introducirlos antes es que estos bloques, en general, no son sintetizables, es decir que no será posible implementar su funcionamiento en una *FPGA* y por tanto no son útiles para el diseño. Sin embargo, resultan muy útiles para las simulaciones y será donde realizaremos la generación de señales.

La sintaxis de un bloque *initial* es:

```
initial begin
  ... //Sentencias
end
```

Se comienza con el keyword *initial*, seguido por las sentencias a ejecutar, las que serán ejecutadas por orden, encerradas por *begin* y *end*.

Puede haber más de un bloque initial en un testbench y todos serán ejecutados concurrentemente al inicio de la simulación.

2.3.2 Sentencias *delay*

Si bien se dijo que los bloques *initial* permiten tener asignaciones que se realizan por orden, en lo que respecta a la simulación, estas aún ocurren todas en una unidad de tiempo. Esto presenta dos problemas: Primero que nada, no es una situación realista puesto a que nunca tendremos dos señales que cambien exactamente al mismo tiempo. En segundo lugar, significará que el bloque que realiza la generación de señales siempre terminará en una unidad de tiempo lo que no nos permitirá observar lo que sucede en los estados intermedios.

Para solucionar esto, se utilizan delays. Un delay se puede utilizar por si sólo o asociado a una sentencia. En ambos casos existirá un intervalo de tiempo antes de que la siguiente sentencia se ejecute. La sintaxis de un delay es: #<n>, en donde n corresponde a la duración del *delay*.

Ejemplo 2.13

Realicemos ahora nuestro primer testbench, sobre una compuerta *and*:

```
module test_and();
  reg A, B;
  wire X;

  and (X, A, B);

  initial begin
    A = 1'b0;
    B = 1'b0;
    #10 A = 1'b1;
    #10 B = 1'b1;
    #10 A = 1'b0;
  end
endmodule
```

En el módulo anterior tenemos una compuerta *and*, que tiene como entradas *A* y B, declaradas como *reg* puesto a que serán asignadas en el bloque initial para generar las señales de prueba; y el *wire X* como salida. En la siguiente figura tenemos la simulación realizada con el software *ISim* de *Xilinx*.

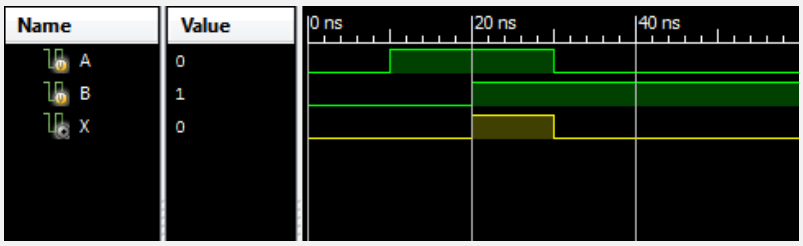


Figura 2.2: Resultados de testbench para compuerta *and*

Es posible observar la correspondencia entre las señales mostradas por la simulación y lo especificado en el bloque *initial*:

- Al comienzo de la simulación, se asigna 0 a ambas entradas de la compuerta. Se puede ver que la salida *X* se mantiene en 0, de acuerdo con el comportamiento esperado para la compuerta.
- Después de estas dos asignaciones, se tiene una asignación con un *delay* de 10 unidades de tiempo a *A*. Vemos en la simulación que transcurrido este periodo, se realiza la asignación y el valor de *A* está en alto.

- Se tiene otra asignación con un *delay* de 10, esta vez de B. Podemos ver nuevamente un intervalo de tiempo hasta que el valor de B pasa a alto. Desde este momento, como ambas entradas de la compuerta *and* están en alto, también lo está su salida.
- Se tiene un último *delay*, tras el cual se asigna 0 a A. Junto con esto la salida X vuelve a 0.

Por medio de este testbench probamos todos las posibles combinaciones de entradas para una compuerta *and* de dos entradas, pudiendo verificar que sus salidas eran las correctas.

2.3.3 El *reloj perpetuo*

Al trabajar con circuitos sincrónicos, se trabajará con una señal de reloj, que consistirá de pulsos periódicos. Para esto generalmente se cuenta con un oscilador junto con la *FPGA*, que provee de esta señal.

Para realizar pruebas sobre circuitos sincrónicos, necesitaremos una señal que se comporte como tal. Hay diversas formas de realizar esto, pero una solución simple es utilizar la estructura *forever*:

```
reg clk;

initial begin
  clk = 0;
  forever begin
    #10 clk = ~clk;
  end
end
```

El bloque *forever* indica una bucle infinito, que comienza nuevamente inmediatamente después de terminar. Este siempre debe ir dentro de un bloque *initial*. En este caso dentro del *forever* tenemos un *delay* junto con una inversión del registro *clk*. Como resultado tendremos una señal que cambia periódicamente entre alto y bajo, lo que cumple la función de un oscilador.

Otro método de implementar una señal de reloj, que nos da un resultado idéntico al anterior, es mediante un bloque *always*:

```
reg clk = 0;
```

```
always begin
  #10 clk = ~clk;
end
```

2.4 Ejemplo: diseñando multiplexores

Un multiplexor es un dispositivo que nos permite seleccionar una de entre múltiples señales. Basicamente actúa como un interruptor digital con muchas entradas y una salida.

Un multiplexor 2x1 es aquel con 2 entradas y una salida. Para poder seleccionar entre una u otra necesitamos una señal de control. Este multiplexor se simboliza de la siguiente forma:

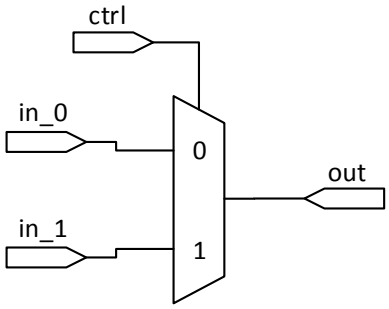


Figura 2.3: Multiplexor 2x1.

y podemos resumir su tabla de verdad como sigue:

Tabla 2.1: Tabla de verdad de multiplexor 2x1

in_0	in_1	ctrl	out
0	X	0	0
1	X	0	1
X	0	1	0
X	1	1	1

En *Verilog* podemos implementarlo de la siguiente forma:

```
module mux2x1(in, ctrl, out);
  input [1:0] in;
  input ctrl;
  output out;

  reg out;

  always @(*) begin
    if (ctrl == 0)
      out = in[0];
    else
      out = in[1];
  end
endmodule
```

Basicamente tenemos nuestras 2 entradas en un bus y la salida *out* que asignamos condicionalmente dependiendo de la entrada *ctrl*. Notar que este módulo es completamente combinacional.

Para verificar el correcto funcionamiento de nuestro multiplexor realizamos un *testbench*:

```
module test_mux2x1();
  reg [1:0] in;
  reg ctrl;
  wire out;

  mux2x1 uut(in, ctrl, out);

  initial begin
    {ctrl, in} = 0;
    repeat (8) begin
      #10 {ctrl, in} = {ctrl, in} + 3'b1;
    end
  end
endmodule
```

En la generación de señales en este *testbench* tenemos un par de peculiaridades:

- Se utiliza el bucle *repeat* que nos permite ejecutar su contenido repetidamente, un numero fijo de veces.
- Para asignar los registros, lo realizamos a la concatenación de *ctrl* e

in. De esta forma, al aumentar el valor de esta concatenación en 1 en cada paso, podemos probar todos los casos facilmente.

- Al realizar 8 repeticiones se prueban todos los casos posibles por lo que este constituye un test exhaustivo.

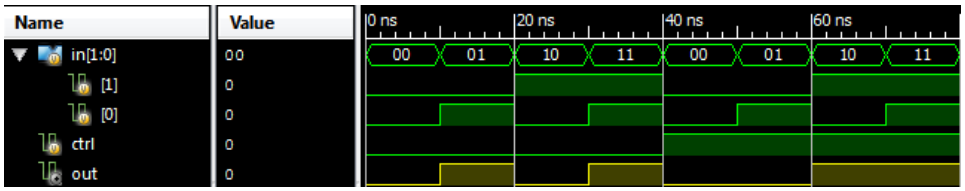


Figura 2.4: Resultados de testbench para multiplexor 2x1.

En la figura anterior tenemos los resultados de la simulación en donde podemos observar el funcionamiento esperado:

- Cuando *ctrl* está en 0, la salida *out* sigue al bit 0 de *in*.
- En el caso contrario, cuando *ctrl* es 1, *out* sigue al bit 1 de *in*.

¿Como podemos construir un multiplexor mas grande? Una posibilidad es cambiar nuestro código para el multiplexor 2x1, para considerar más señales de control, sin embargo también es posible construir multiplexores con más entradas a partir del multiplexor que ya tenemos y cuyo funcionamiento hemos verificado:

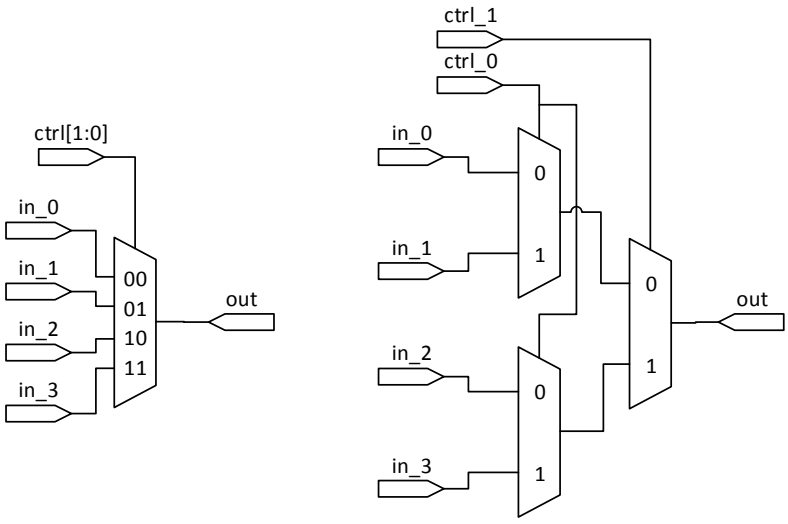


Figura 2.5: Multiplexor 4x1.

Podemos de esta forma crear un módulo multiplexor 4x1 de esta forma:

```
module mux4x1(in, ctrl, out);
  input [3:0] in;
  input [1:0] ctrl;
  output out;

  wire [1:0] sel;

  mux2x1 mux0(in[1:0], ctrl[0], sel[0]);
  mux2x1 mux1(in[3:2], ctrl[0], sel[1]);
  mux2x1 mux2(sel, ctrl[1], out);
endmodule
```

Este módulo consiste basicamente en conectar multiplexores 2x1 según el esquema anterior. Notar que se debe agregar *wires* para realizar las conexiones entre el primer y segundo nivel de multiplexores.

Realizamos también el testbench respectivo, esta vez probando los 64 posibles casos.

```
module test_mux4x1();
  reg [3:0] in;
  reg [1:0] ctrl;
  wire out;

  mux4x1 uut(in, ctrl, out);

  initial begin
    {ctrl, in} = 0;
    repeat (64) begin
      #10 {ctrl, in} = {ctrl, in} + 1;
    end
  end
endmodule
```

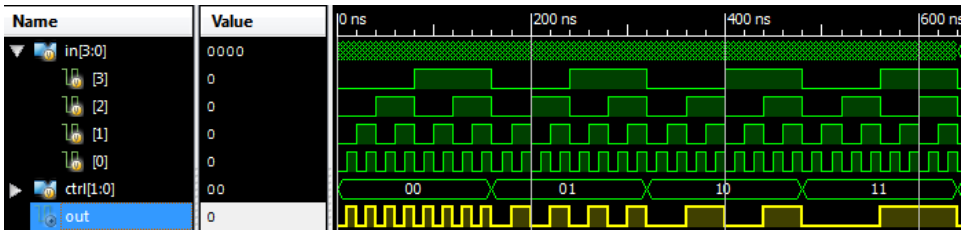


Figura 2.6: Resultados de testbench para multiplexor 4x1.

Podemos ver como al cambiar el valor de la señal de control, cambia la salida seleccionada. De esta forma se logra construir el multiplexor sin necesidad de realizar un módulo que considere cada uno de los casos por separado. Siguiendo este esquema podemos construir multiplexores cada vez más grandes, que sabemos que funcionarán.

A fin de cuentas este es un ejemplo simple cuyo objetivo es mostrar como el diseño jerarquico nos permite aumentar la complejidad del diseño a partir de un módulo simple sin mayores dificultades, y manteniendo un orden comprensible.

2.5 Ejemplo: Diseñando contadores

En el ejemplo anterior se diseñaron multiplexores, que resultan ser circuitos completamente combinacionales. En este ejemplo se diseña un contador, un dispositivo que almacenará el número de veces que ocurre un suceso, en este caso el canto de subida de una señal de reloj. Claramente esto implica la existencia de memoria por lo que ya no se tendrá solamente un circuito secuencial.

Analicemos primero las necesidades de nuestro diseño:

- En cuanto a entradas y salidas necesitamos, primero que todo, una entrada correspondiente a la señal de reloj y una salida correspondiente a la cuenta actual. Se utilizará además una señal de entrada que nos permita resetear la cuenta.
- Se necesita un elemento de memoria para poder almacenar el valor de la cuenta actual. Esta corresponde a la parte secuencial del circuito.
- Es necesario determinar el valor siguiente de la cuenta. Este valor sólo depende de la cuenta actual, por lo tanto solo tiene una componente combinacional.
- Sería util poder adaptar nuestro diseño para lograr cuentas de diferentes tamaños.

Las anteriores características las podemos resumir al siguiente módulo:

```
module contador(clk, rst, cuenta);
  parameter N = 4;

  input clk, rst;
  output reg [N−1:0] cuenta;

  reg [N−1:0] cuenta_next;

  always @(*) begin
    cuenta_next = cuenta + 1;
  end

  always @(posedge clk or posedge rst) begin
    if (rst)
      cuenta <= 0;
    else
      cuenta <= cuenta_next;
  end
endmodule
```

En este código se tiene lo siguiente:

- El parametro *N*, nos permite elegir el tamaño en bits del contador, al utilizarlo para definir los registros *cuenta* y *cuenta_next*.
- El registro *cuenta_next*, se asigna en el primer bloque *always* y corresponde al siguiente valor de la cuenta. Si bien se declara como un registro, dado a que se asigna en un bloque *always* con lista de sensibilidad *, su valor es completamente combinacional-
- El segundo bloque *always* representa la parte secuencial del diseño. En este caso, si *rst* está en alto la cuenta se vuelve a cero, en otro caso se asigna a *cuenta* el valor siguiente.
- Una observación importante es que la función de *rst* no es solo proporcionar un método para resetear la cuenta. Debido a que el registr *cuenta* no tiene un valor inicial, este se encuentra inicialmente en un valor indeterminado (representado por X), de esta forma el valor de *cuenta_next* es también indeterminado. De esta forma, puesto que *rst* asigna un valor constante a *cuenta*, nos proporciona un metodo de iniciar el contador.

Construyamos ahora un *testbench* para nuestro módulo:

```
module test_contador();
  reg clk = 0;
  reg rst = 0;
  wire [3:0] cuenta_1;
  wire [1:0] cuenta_2;

  contador uut_1(clk, rst, cuenta_1);
  contador #(2) uut_2(clk, rst, cuenta_2);

  always begin
    #10 clk = ~clk;
  end

  initial begin
    #20 rst = 1;
    #5 rst = 0;
    #95 rst = 1;
    #5 rst = 0;
  end
endmodule
```

En este testbench tenemos dos instancias de nuestro modulo: una con la configuración por defecto de una cuenta de 4 bits y el otro con una cuenta de 2 bits. Tenemos también registros *clk* y *rst* utilizados como entradas para ambos módulos y *wires cuenta_1* y *cuenta_2* utilizados como las salidas.

Para generar la señal de reloj usamos uno de los métodos mencionados anteriormente. La señal de rst es controlada por el bloque *initial* y genera dos pulsos de 5 unidades de tiempo.

Al simular se obtiene el siguiente resultado

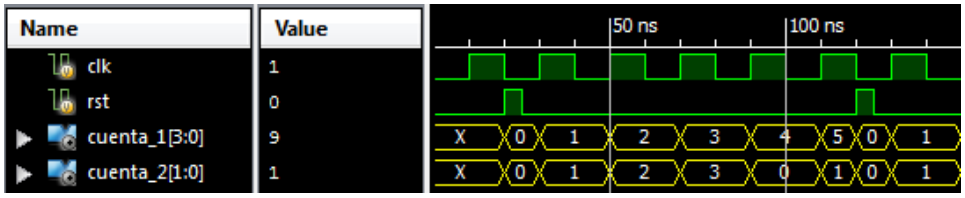


Figura 2.7: Resultados de testbench para multiplexor 2x1.

De este podemos notar que:

- Los valores iniciales de las cuentas son indeterminados y se mantienten así hasta el primer pulso de *rst*. En particular podemos ver que después

del primer canto de subida de *clk*, se mantiene indeterminado.

- Después del primer pulso de *rst*, las cuentas se van a cero y aumentan en 1 con cada canto de subida del reloj. Además, con el segundo pulso de *rst* las cuentas vuelven a cero. Esto confirma el correcto funcionamiento de nuestro módulo.

- Podemos ver además como *cuenta_2* solo llega cuenta hasta 3, lo que es de esperar dado a que corresponde al contador de 2 bits.

2.6 Fuentes

- Michael D. Ciletti, "Advanced Digital Design with the Verilog HDL".

- http://www.asic-world.com/verilog/veritut.html

- http://numato.com/tutorials/learning-fpga-and-verilog-a-beginners-guide-part-1-introduction/

- http://www.sunburst-design.com/papers/CummingsSNUG2000SJ_NBA_rev1_2.pdf

- http://www.iuma.ulpgc.es/ñunez/clases-FdC/verilog/Verilog Tutorial v1.pdf

Capítulo 3

Diseño de circuitos combinacionales

Capítulo 4

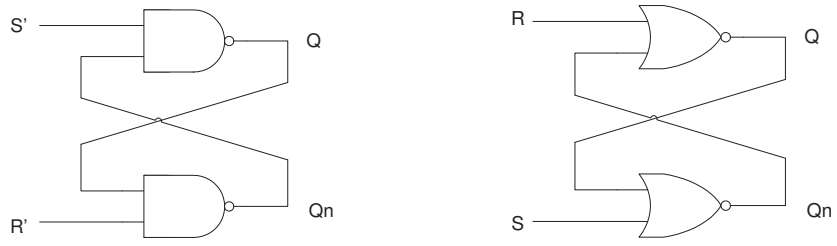
Diseño de circuitos secuenciales

4.1 El Flip-Flop

Un circuito secuencial es aquel cuya salida y estado no solo dependen de su entrada actual, sino también del estado en que se encuentra, es por ello, que antes de diseñar un circuito secuencial es necesario contar con algún tipo de componente capaz de mantener de manera estable un estado lógico (cero o uno). Este componente es el flip-flop.

Un flip-flop puede ser seteado o reseteado y además puede mantener ese estado durante el tiempo que sea necesario. El entendimiento de este circuito es muy importante a la hora de diseñar, pues nunca hay que dejar de pensar que lo que se está "programando" dará como resultado un circuito lógico, es decir, hardware.

El primer acercamiento que se conoce de circuitos capaces de almacenar un bit, corresponde al "latch SR", el cual puede ser construído basándose en la compuerta NAND o NOR, tal cual muestra la figura: (4.1)



(a) Latch sr NAND (b) Latch sr NOR

Figura 4.1: Latch SR

Un latch de NANDs puede ser descrito en verilog de la siguiente manera:

```
module sr_latch_nan(s,r,q,qn);
  input s,r;
  output q,qn;
  wire sn,rn;

  not(sn,s);
  not(rn,r);
  nand(q,sn,qn); //q=~(sn & qn);
  nand(qn,rn,q); //qn=~(rn & q);

endmodule
```

cuya tabla de verdad se puede ver en (4.1)

Tabla 4.1: Tabla de verdad de un latch sr nand

s	r	sn	rn	q	qn	
0	0	1	1	q	qn	mantiene el estado
0	1	1	0	0	1	
1	0	0	1	1	0	
1	1	0	0	1	1	estado no permitido

A éste, se le hizo una pequeña modificación para habilitar o deshabilitar la escritura en el latch, dando origen al chip "Gated Latch SR", como se muestra en la figura (4.2) el cual puede ser descrito en verilog de la siguiente manera:

```
module gated_rs_nan_latch(s,r,en,q,qn);
  input s,r,en;
  output q,qn;
  wire ss,rs;

  nand(ss,s,en);//ss=~(s&en)
  nand(rs,r,en);//rs=~(r&en)

  //sr_latch_nan
  //... se puede emplear estrategia modular
  nand(q,ss,qn);
  nand(qn,rs,q);
endmodule
```

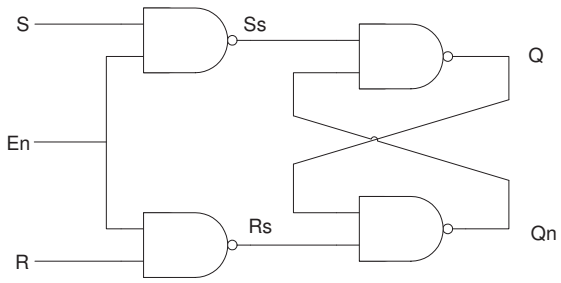


Figura 4.2: Gated Latch sr NAND

cuya tabla de verdad se puede ver en 4.2

en	s	r	ss	rs	q	qn	
0	X	X	1	1	q	qn	se mantiene el estado
1	0	0	1	1	q	qn	se mantiene el estado
1	0	1	1	0	0	1	
1	1	0	0	1	1	0	
1	1	1	0	0	1	1	estado no permitido

Tabla 4.2: Tabla de verdad de gated latch rs

Una manera de evitar que el circuito entrara al estado no permitido, fue poner una sola entrada que se conectara directamente a la entrada “S” y su complemento a la entrada “R” como se muestra en la figura 4.3, dando origen al Latch-D o latch transparente.

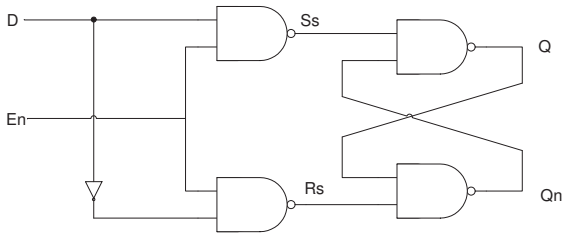


Figura 4.3: Latch D

Este circuito se puede describir en verilog de esta forma:

```
module latch_D(en,d,q,qn);
  input en,d;
  output q, qn;
```

```
    wire ss,rs,dn;

    not(dn,d);//dn=~d
    nand(ss,d,en);//ss=~(d&en)
    nand(rs,dn,en);//rs=~(dn&en)

    nand(q,ss,qn);
    nand(qn,rs,q);

endmodule
```

El problema de este latch D es que funciona por nivel, es decir, mientras la entrada *en* esté en nivel alto, se pueden cambiar el estado del latch tal como lo muestra la tabla (4.3), es por ello, que también recibe el nombre de latch transparente. ¡Mientras la señal *en* esté en alto, la salida será igual a la entrada!

en	*D*	*Siguiente estado*
0	X	se mantiene el estado
1	0	$Q = 0$
1	1	$Q = 1$

Tabla 4.3: Tabla de verdad de un latch-D

Con todos estos avances se logró contar un circuito capaz de almacenar un bit y además controlar su escritura con una señal habilitadora. El problema era que si la señal de entrada cambiaba mientras la señal habilitadora estaba en alto, se podrían generar inconsistencias. Con este inconveniente en mente, se creó a partir de los diseños anteriores un circuito que respondiera a transiciones de la señal habilitadora, es decir, que reaccionara a "cantos de subida" o "cantos de bajada". Los circuitos de la figuras 4.4 y 4.5 fueron llamados Flip-Flops D. El primero no es posible setearlo o resetearlo de manera asíncrona, mientras que el segundo tiene la entrada PRE y RST que sí lo permiten.

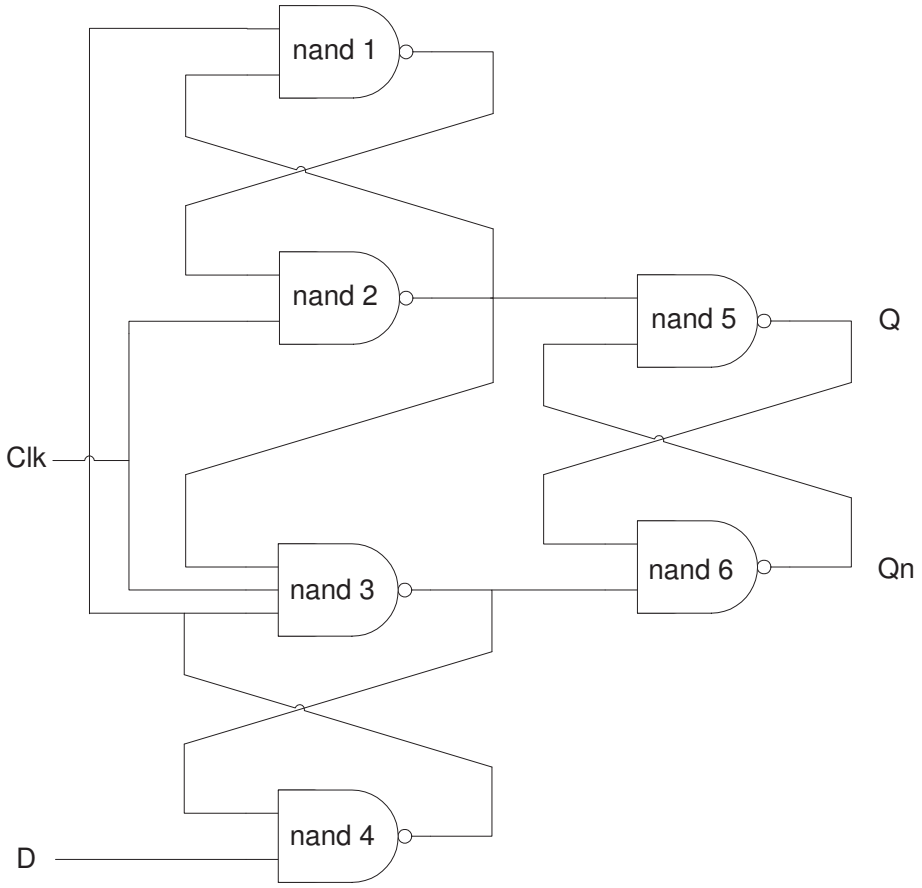


Figura 4.4: Flip Flop D

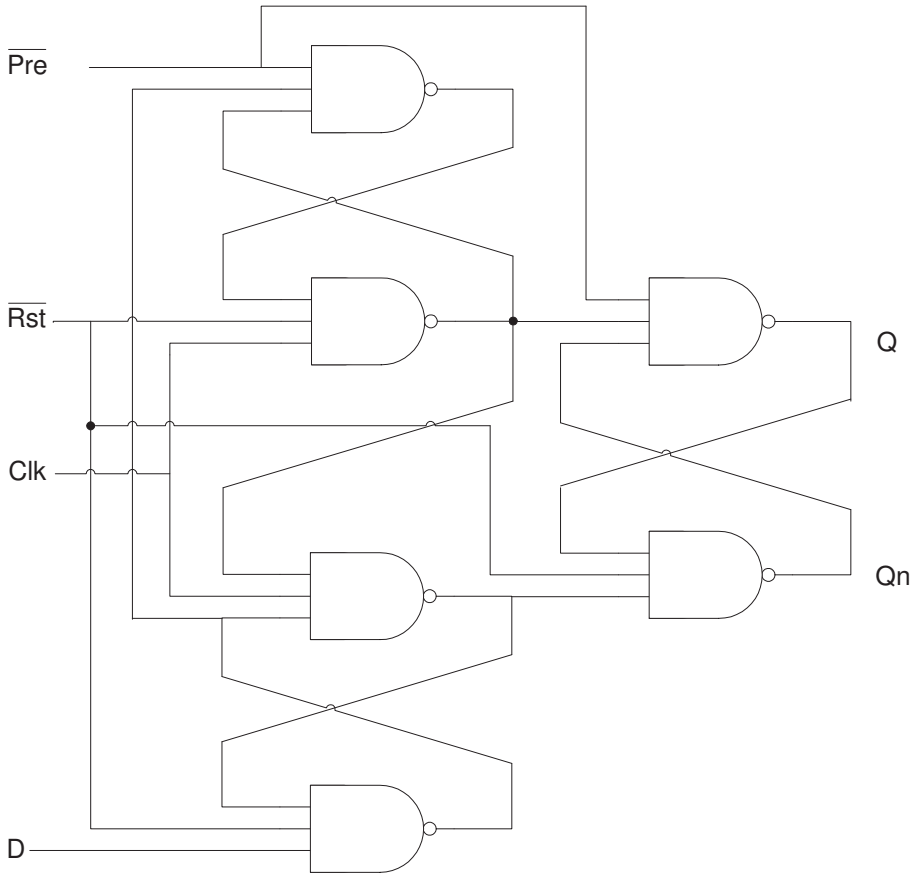


Figura 4.5: Flip Flop D con preset y reset

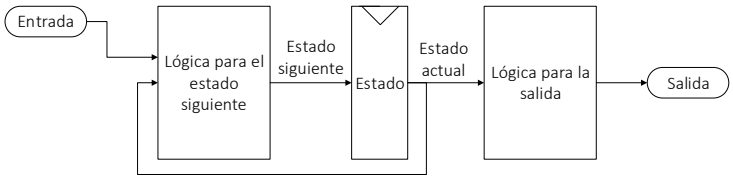
El software que compila lenguaje Verilog, intentará traducir el código a circuitos lógicos y determinará cuando se deberán utilizar flip-flops, latch, etc. En general se recomienda no utilizar latch a menos que se esté muy seguro de lo que se quiere conseguir.

4.2 Maquinas de Moore y Mealy

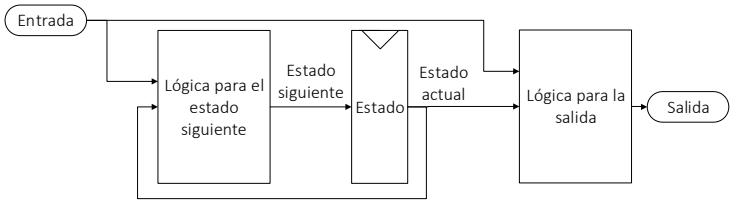
Dos modelos comunes de circuitos secuenciales son las máquinas de Moore y Mealy. En estos circuitos consisten en una serie finita de estados que son recorridos en función de tanto el estado actual como las entradas al circuito.

La diferencia entre ambos modelos radica en como se determinan sus salidas. En una máquina de Moore, la salida del circuito es una función del estado actual, es decir, el estado determina cual es la salida del circuito y esta sólo cambiará al haber un cambio de estado. Por otra parte, en una máquina de Mealy la salida depende de la transición de estados que se realiza, de esta

forma la salida no sólo depende del estado actual sino que también de la entrada. En la figura 4.6 se muestran esquemas de ambas máquinas.



(a) Máquina de Moore.



(b) Máquina de Mealy.

Figura 4.6: Diagramas de máquinas de estado.

Podemos ver que en cuanto a la implementación circuital de estas, la única diferencia está en la lógica combinacional correspondiente a la salida. En la máquina de Moore solo participan señales pertenecientes al estado, mientras que en la de Mealy participan tanto señales de entrada como de estado.

4.2.1 Diagramas de estado

Una herramienta muy útil, que nos permite analizar y elaborar facilmente máquinas de estado, son los diagramas de estado. Estos consisten en una representasión gráfica de nuestra máquina. Cada estado se representa por un círculo dentro del cual se etiqueta como tal, junto con la salida que genera de ser una máquina de Moore. Los estados se unen con flechas que indican las posibles transiciones. Cada flecha es acompañada por la entrada que genera tal transicion y, de ser una máquina de Mealy, la salida asociada a esta. En las figuras 4.7 y 4.8 se muestran ejemplos de diagramas.

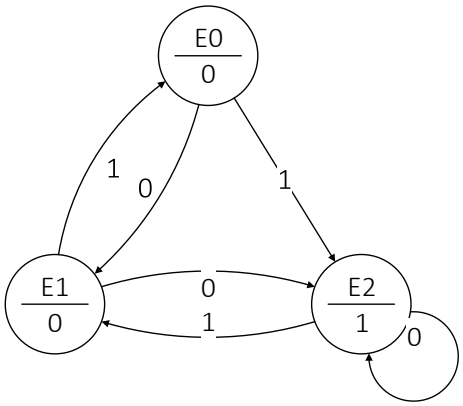


Figura 4.7: Ejemplo de diagrama de estados para máquina de Moore

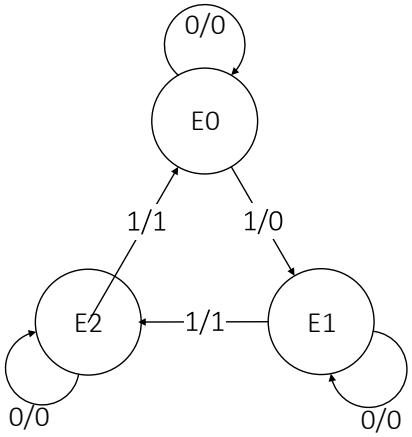


Figura 4.8: Ejemplo de diagrama de estados para máquina de Mealy

4.3 Describiendo Circuitos Secuenciales

La clave de los circuitos secuenciales es que la actualización del estado, se produce de manera sincrónica con el canto de subida o bajada de una señal que llamaremos reloj o clk. A diferencia de los circuitos combinacionales, los secuenciales deben ser descritos dentro de un bloque *always* en cuya lista de sensibilidad se utilizará la palabra *posedge* para referirnos a un canto de subida y *negedge* para referirnos a un canto de bajada. En general no

crearemos instancias de los flip-flops disponibles, sino que más bien se dejará en manos del compilador la tarea de decidir cómo implementar el código.

Todo circuito secuencial puede esquematizarse como una red combinacional seguida de una memoria o registro que almacena el estado del sistema, cuya salida realimenta la entrada del mismo tal cual se puede ver en el esquema de la figura(4.9).

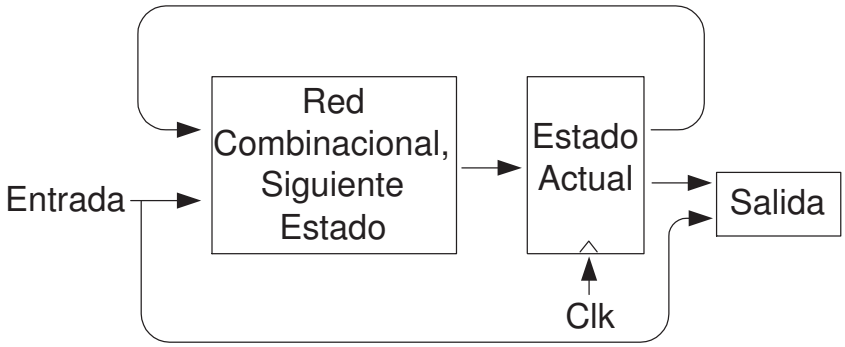


Figura 4.9: Diagrama Circuito Secuencial

Una manera de describir este diagrama en Verilog es:

```
module circuito_sec(reloj_ext, reset_ext, entrada, salida);
  input reloj_ext, reset_ext;
  input [3:0] entrada;
  output [1:0] salida;
  reg [2:0] estado, estado_futuro;

  //Logica para el siguiente estado que depende
  //del estado actual y de las entradas
  always @(*)
  begin
    case(estado)
      3'b000:
        if(entrada==4'b0001)
          estado_futuro=3'b010;
        else if(entrada==4'b0011)
          estado_futuro=3'b001;
        else
          estado_futuro=3'b000;
      3'b001://....
```

```
        3'b010://....
        3'b011,3'b100: //...
        default://estado_futuro=3'b000
      endcase
  end

  always @(posedge reloj_ext or negedge reset_ext)
  begin
    if(reset_externo==1'b0)
      estado<=3'b000;
    else
      estado<=estado_futuro;
  end

  //asignacion de la salida
  assign salida[0]=(estado==3'b010)?1:0;
  assign salida[1]=((estado==3'b110)&&
              (entrada==4'b0101))?1:0;

endmodule
```

Se puede ver claramente que el primer bloque *always* corresponde a un circuito combinacional que se utiliza para calcular el siguiente estado (*estado_futuro*), luego está el bloque *always* que "actualiza" o registra el estado (*estado*) y finalmente está la asignación de la salida, que en este caso está compuesta por un bit asociado directamente a un estado (máquina de Moore) y otro bit que está asociado a un estado y a la entrada (máquina de Mealy). De este esquema se pueden desprender varias cosas:

- Recordar que la palabra *reg* no quiere decir que la variable quedará almacenada en algún tipo de memoria (Flip-Flop o latch), pues la variable *estado_futuro* debería corresponder a la salida de una red combinacional y esto queda reforzado por el echo de que la lista de sensibilidad del bloque *always* corresponde a un asterisco (*) y no a eventos del tipo *posedge* o *negedge*.

- La manera más sencilla de definir la escritura en Flip-Flops es por medio de un bloque *always* cuya lista de sensibilidad contenga eventos del tipo *posedge* o *negedge* y asignar el valor con asignación "no bloqueante", sin embargo puede haber casos en que, a pesar de cumplir con estas dos condiciones, el resultado sea un circuito combinacional como veremos en el capítulo 13

- No se recomienda mezclar en la lista de sensibilidad señales que actúan

por nivel y señales que actúan por cantos.

- No existe una palabra reservada para indicar al compilador que cierta señal será la que se conectará la entrada de reloj de los Flip-Flops ni tampoco una palabra reservada para indicar que señal se conectará la entrada "reset" o "clear" o a la entrada "preset".

- La notación no bloqueante <= solo se debe utilizar en descripciones de circuitos secuenciales.

La lista de sensibilidad de un bloque *always* que define un circuito secuencial, debe tener como mínimo una señal de tipo *posedge* o *negedge* y como máximo tres. Es muy importante destacar que solo una de estas será la que se conecte a la entrada *clk* de los Flip-Flops que almacenarán el estado del circuito, y el resto se conetarán de alguna manera a las entradas *reset* y/o *preset* de los Flip-Flops. Como se mencionó anteriormente, no existe una palabra reservada para indicarle al compilador cual señal se conectará a que entrada de los Flip-Flops, pero si existe convención para ello.

- Si en la lista de sensibilidad hay una sola señal (de tipo *edge*) entonces será esta la que conectará a la entrada *clk* de los Flip-Flops.

- Si en la lista de sensibilidad hay dos señales (de tipo *edge*) una de ellas se deberá utilizar para conectarla directamente al *reset* o al *preset* de los Flip-Flop o de manera indirecta a través de una red combinacional y la manera de indicárselo al compilador es mediante un "if" que "encuesta" el estado de una de las señales de la lista de sensibilidad evaluando su estado (*if(reset_ext==1'b0)*) y dependiendo del valor que deba escribirse, conectará esa señal al *reset* de los Flip-Flops (en caso que deba escribirse un cero), al *preset* de los Flip-Flops (en caso que deba escribirse un uno) o se generará una red combinacional que tendrá dicha señal como entrada y sus salidas se conectarán al *reset* y *preset* del o los Flip-Flops (en caso de que se desee escribir un valor que no sea constante). De esta forma la señal que no sea encuestada, corresponderá a la entrada *clk* de los Flip-Flops.

- Si en la lista de sensibilidad hay tres señales (de tipo *edge*), se sigue el mismo procedimiento que para dos solo que se deberán encuestar dos de las señales, de esta manera la que no sea encuestada se conectará al *clk* del o los Flip-Flops.

- El compilador no acepta más de 3 señales de tipo *edge* en la lista de sensibilidad de un bloque *always*

- Si bien los puntos anteriores son de suma importancia al momento de describir un circuito secuencial. Es también importante saber que el

comportamiento anteriormente mencionado de la lista de sensibilidad *NO* es una característica del lenguaje *Verilog*, si no que es causa de limitaciones del hardware sobre el que se trabaja.

Para examinar detalladamente los casos anteriormente expuestos, junto con su significado a nivel de hardware, se presenta un ejemplo en la sección siguiente.

Otro aspecto muy importante de los circuitos secuenciales es la forma en que se hace la asignación de un registro, pues el lenguaje soporta dos tipos, la asignación bloqueante que corresponde al signo igual (=) y la asignación no-bloqueante que corresponde al signo menor igual (<=). Si bien se pueden utilizar cualquiera de ellas, es mucho más fácil entender y manejar la no-bloqueante, pues permite independizarse del orden en que se escriben las asignaciones, mientras que en la bloqueante, hay que ser muy cuidadosos a la hora de escribirlas.

Por ejemplo si deseamos describir un registro de desplazamiento podemos hacerlo con la asignación bloqueante y con la no-bloqueante de la siguiente manera:

```
module shift_register_nonblock(clk,entrada,salida,rst);
  input clk,rst;
  input entrada;
  output salida;
  reg A0,B0,C0,D0;

  always@(posedge clk or posedge rst)
  begin
    if(rst==1'b1)
      {A0,B0,C0,D0}<=4'b0000;
    else
    begin
      A0<=entrada;
      B0<=A0;
      C0<=B0;
      D0<=C0;
    end
  end
  assign salida=D0;
endmodule
```

Con la asignación no-bloqueante el orden en que se escriban las asignaciones no importa, es decir, podríamos escribir:

```
module shift_register_block(clk,entrada,salida,rst);
 input clk,rst;
 input entrada;
 output salida;
 reg A1,B1,C1,D1;

 always@(posedge clk or posedge rst)
 begin
   if(rst==1'b1)
     {A1,B1,C1,D1}=4'b0000;
   else
   begin
     D1=C1;
     C1=B1;
     B1=A1;
     A1=entrada;
   end
 end
 assign salida=D1;
endmodule
```

Cuando se describen circuitos secuenciales es recomendable tener muy en cuenta los puntos descritos anteriormente, en especial el problema de las señales que actúan por cantos y la manera en el compilador las implementa. Además se debería intentar seguir un estándar de diseño y utilizar estrategias de diseño modular como las que se verán en el capítulo 5.

4.4 Ejemplo: Cerradura con clave

Se quiere diseñar un circuito que sirva para detectar la correcta digitación de una clave de acceso según las siguientes especificaciones:

1 La clave secreta corresponde a los 7 primeros números de la secuencia de Fibonacci (0,1,1,2,3,5,8).

2 Debe haber una salida que se ponga en alto justo en el momento en que se presente la clave correcta.

3 Debe haber una salida que se ponga en alto cuando se ingrese correctamente hasta el cuarto término de la serie.

4 En caso que se cometa un error en la secuencia, entonces se deberá volver al estado inicial y otra salida se pondrá en alto.

5 Modelar el problema con una máquina de Mealy.

Un diagrama de estados posible para este problema podría ser el de la figura (4.10)

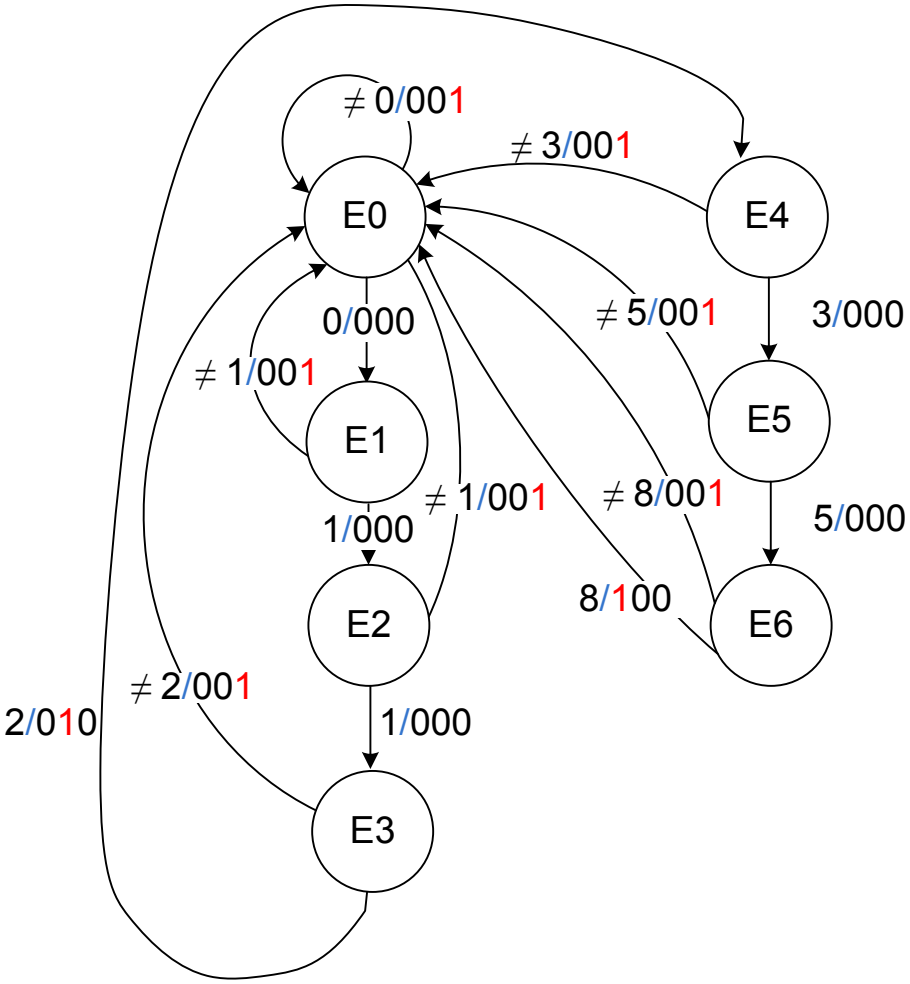


Figura 4.10: Diagrama de estados utilizando Mealy

En Verilog podría describirse de la siguiente manera:

```
`define E0 3'b000
`define E1 3'b001
`define E2 3'b010
`define E3 3'b011
`define E4 3'b100
`define E5 3'b101
```

```
`define E6 3'b110
module fibonacci_mealy(clk,entrada,salida);
  input clk;
  input[3:0] entrada;
  output[2:0] salida;
  reg estado_futuro, reg estado_presente;

  always @(*)//calculo el siguiente estado
  begin
    case (estado_presente)
      `E0:estado_futuro=(entrada==4'b0000)?`E1:`E0;
      `E1:estado_futuro=(entrada==4'b0001)?`E2:`E0;
      `E2:estado_futuro=(entrada==4'b0001)?`E3:`E0;
      `E3:estado_futuro=(entrada==4'b0010)?`E4:`E0;
      `E4:estado_futuro=(entrada==4'b0011)?`E5:`E0;
      `E5:estado_futuro=(entrada==4'b0101)?`E6:`E0;
      `E6:estado_futuro=(entrada==4'b1000)?`E0:`E0;
      default estado_futuro=`E0;
    endcase
  end
  always @(posedge clk)//registro el estado
    estado_presente<=estado_futuro;

  //asigno las salidas
    assign salida[0]=
  (((estado_presente==`E0)&&(entrada!=4'd0))||
  ((estado_presente==`E1)&&(entrada!=4'd1))||
  ((estado_presente==`E2)&&(entrada!=4'd1))||
  ((estado_presente==`E3)&&(entrada!=4'd2))||
  ((estado_presente==`E4)&&(entrada!=4'd3))||
  ((estado_presente==`E5)&&(entrada!=4'd5))||
  ((estado_presente==`E6)&&(entrada!=4'd8)))?1:0;
  assign salida[1]=((estado_presente==`E3)
  &&(entrada==4'd2))?1:0;
  assign salida[2]=((estado_presente==`E6)
  &&(entrada==4'd8))?1:0;
endmodule
```

Se puede ver claramente que al momento de registrar el estado, solo tenemos una señal de tipo canto de subida (*clk*), por lo que, al momento de sintetizar el diseño, dicha señal será la que se conecte a la entrada de reloj de los Flip-Flops que componen el registro *estado_presente*. Ahora se

agregará una señal que vuelva al estado inicial la máquina de estados, a la que llamaremos *rst*.

```
...
module fibonacci_mealy(clk,rst,entrada,salida);
  input clk,rst;
  input[3:0] entrada;
  output[2:0] salida;
  reg estado_futuro, reg estado_presente;

  always @(*)//calculo el siguiente estado
  ...

  //registro el estado
  always @(posedge clk or posedge rst)
  begin
    if(rst==1'b1)
      estado_presente<=`E0;
    else
      estado_presente<=estado_futuro;
  end
  //asigno las salidas
  ...
endmodule
```

Esta vez hay dos señales del tipo *edge*, por lo que, es necesario encuestar al menos una de ellas, pero hay que tener cuidado al momento de elegir cual, pues se debe recordar que aquella que no será encuestada, se conectará a la entrada *clk* de los flip-flops. En este caso se desea que *clk* se conecte a la entrada clock de los flip-flops, por lo tanto, se encuesta la señal *rst*. El circuito que se generará, tendrá conectada la entrada *reset* de los flip-flops a la señal rst. Distinto sería el caso si se decidiera que la señal *rst* dejara el *estado_presente* en *E1* (001), pues el circuito generado tendría que setear (dejar en uno lógico) el flip-flop menos significativo del registro *estado_presente* y en cero el resto. Esto se logra conectando directamente *rst* a la entrada *reset* de los dos flip-flops que están en las posiciones más significativas y a la entrada *preset* del que está en la posición menos significativa como se muestra en la figura (4.11)

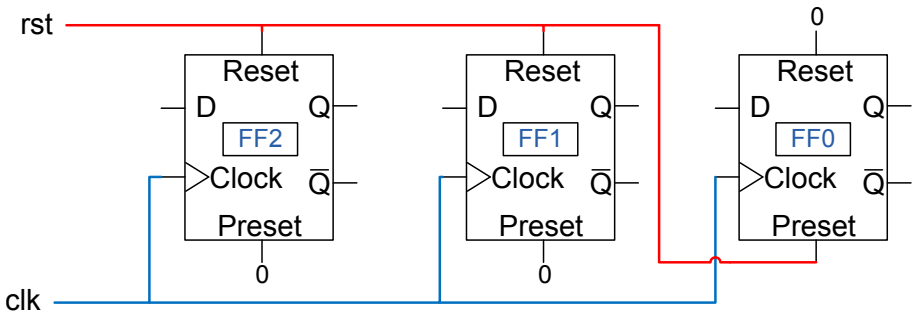


Figura 4.11: Conección de la señal rst en los flip-flops del registro para dejar el registro en el estado E1

De este ejemplo se puede decir que solo la señal *clk* actúa por cantos, mientras que *rst* lo hace por nivel, es decir, que mientras esta última esté en alto, se escribirá en el registro el valor E1. Esto es muy importante, pues es muy común creer que solo si ocurre un canto de subida (*posedge*) en la señal *rst*, se producirá el cambio, pero en realidad este ocurre mientras *rst* esté en alto, lo cual además deja sin efecto los cambios que puedan generarse por cantos de la otra señal (*clk*).

Ahora se agregarán otras dos entradas para poder setear el registro *estado_presente* en cualquier estado posible. A la entrada que gatille el cambio se le llamará *pre* y a la que tenga el valor a setear se le llamará *estado_q*.

```
module fibonacci_mealy(clk,rst,pre,entrada,estado_q,salida);
  input clk,rst;
  input [2:0]estado_q;
  input[3:0] entrada;
  output[2:0] salida;
  reg estado_futuro, reg estado_presente;

  always @(*)//calculo el siguiente estado
  ...

  //registro el estado
  always @(posedge clk or posedge rst or posedge pre)
  begin
    if(rst==1'b1)
      estado_presente<=`E1;
    else if(pre==1'b1)
      estado_presente<=estado_q;
    else //clk se conecta a la entrada clock.
```

```
        estado_presente<=estado_futuro;
    end

    //asigno las salidas
    ...
endmodule
```

Esta vez tenemos 3 señales que actúan por cantos (*clk*, *rst* y *pre*), pero en estricto rigor se podría decir que en realidad solo una de ellas lo hace y el resto actúa por nivel tal cual se explicó en el caso anterior. Para poder lograr un circuito que cumpla con las especificaciones escritas, el software tendrá que crear una red combinacional que se conecte a las entradas *reset*, *preset* y *clock* de los registros. Ya sabemos que la señal *clk* se conectará al clock, pero ocurre algo diferente para las otras. Una manera de modelar o visualizar el circuito combinacional que relaciona las señales *rst*, *pre*, *estado_q*, *reset* y *preset*, es la tabla de verdad (4.4), en la que los reset y preset de cada flip-flop están agrupados en los buses *bus_reset* y *bus_preset*. En ella se puede ver que si las señales *rst* y *pre* están en nivel bajo, los flip-flops del registro estado_presente cambiarán de estado cuando ocurra un canto de subida de la señal *clk*, en cambio si por ejemplo la señal *pre* está en nivel alto (*rst* en bajo), entonces se setearán las señales preset y reset de cada flip-flop para producir el estado querido. Nótese que en general el *bus_reset* es el negado del *bus_preset* excepto cuando las señales *pre* y *rst* están en nivel bajo.

estado_q			*rst*	*pre*	*bus_reset*			*bus_preset*			
X	X	X	0	0	0	0	0	0	0	0	Actúa clk
X	X	X	1	X	1	1	0	0	0	1	E1
0	0	0	0	1	1	1	1	0	0	0	
0	0	1	0	1	1	1	0	0	0	1	
0	1	0	0	1	1	0	1	0	1	0	
0	1	1	0	1	1	0	0	0	1	1	
1	0	0	0	1	0	1	1	1	0	0	
1	0	1	0	1	0	1	0	1	0	1	
1	1	0	0	1	0	0	1	1	1	0	
1	1	1	0	1	0	0	0	1	1	1	

Tabla 4.4: Tabla de verdad

En la figura 4.12 se puede ver un diagrama de conexiones posibles para implementar la tabla de verdad 4.4 mediante multiplexores. Nótese la manera en que se puede definir la prioridad entre las señales pre y rst cuando ambas están en nivel alto.

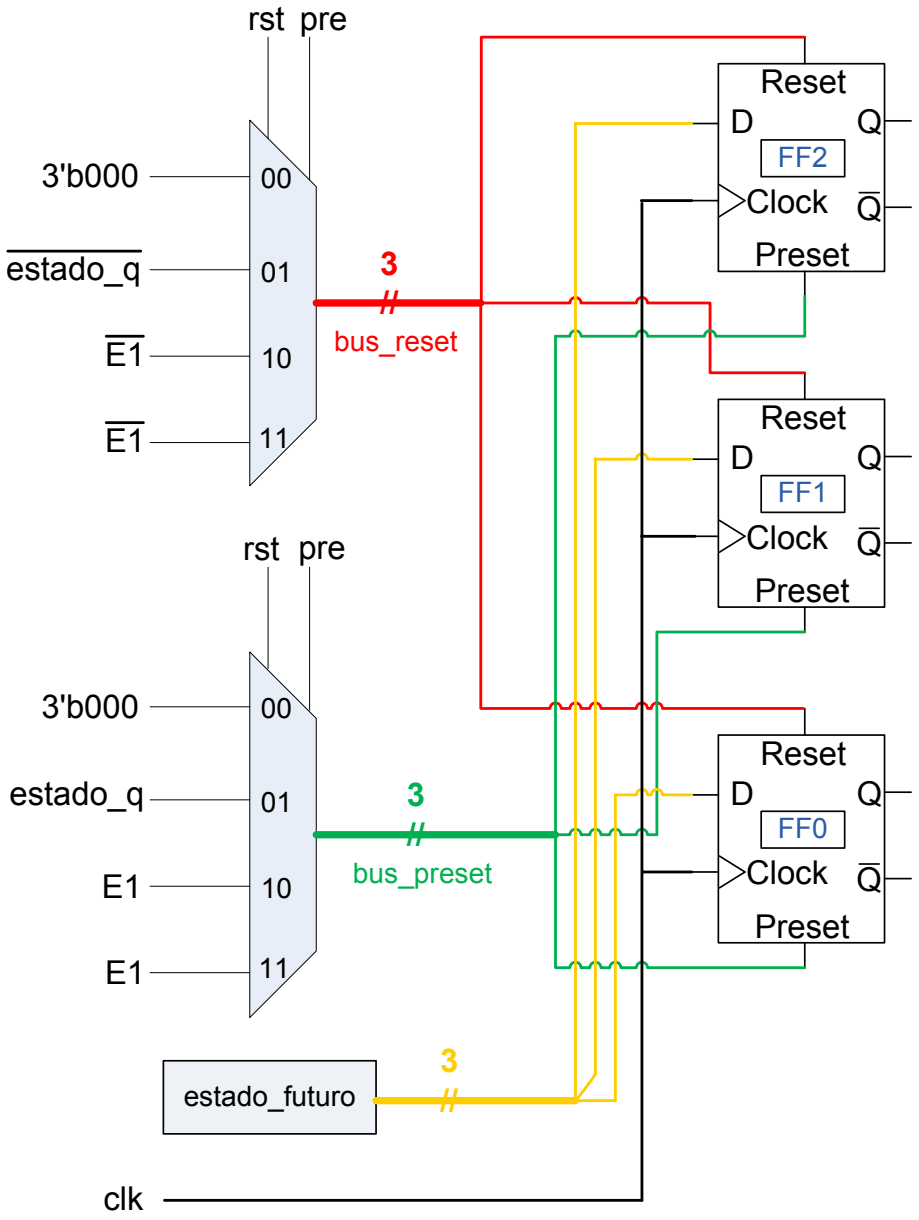


Figura 4.12: Esquema para visualizar la conexión de las señales reset y preset

4.5 Ejemplo: Registros de corrimiento

Un circuito secuencial muy comúnmente usado y que fue mencionado con anterioridad es el registro de corrimiento. Este consiste básicamente en una serie de elementos de memoria interconectados en cadena, de manera que la entrada de cada uno corresponde al estado del anterior de esta forma

logrando que los datos se "desplazen" a través de este.

Existen diferentes tipos de registros de corrimiento. Por ejemplo podemos tener: registros de entrada y salida serial o paralela, o una combinación de ambos; de desplazamiento unidireccional o bidireccional.

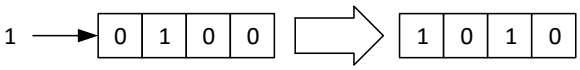


Figura 4.13: Ejemplo de registro de desplazamiento.

Por ejemplo en la figura 4.13 tenemos un registro de corrimiento de entrada serial que actualmente almacena el valor 0100. Si la entrada es 1, al ciclo siguiente se almacena este valor en el primer bit, mientras que los demás se desplazan, perdiéndose el último valor. De esta forma el nuevo estado es 1010

Si quisieramos describir un registro como éste en *Verilog* lo podríamos hacer de la siguiente forma:

```
module shift_register(clk, rst, entrada, dato);
  input clk, rst;
  input entrada;
  output reg [3:0] dato;

  wire [3:0] dato_siguiente;

  assign dato_siguiente = {entrada, dato[3:1]};

  always @(posedge clk or posedge rst)
    if (rst)
      dato <= 4'b0;
    else
      dato <= dato_siguiente;
endmodule
```

Este corresponde a un registro de corrimiento de entrada serial y salida paralela de 4 bits. Este posee como entradas la señal de reloj, un reset y los datos que entran al registro. Como salida tiene el dato almacenado.

Si quisieramos implementar este módulo, el circuito resultante sería el de la figura 4.14. El cual corressponde a una serie de Flip-Flops en cadena, como ya se mencionó. La salida corresponde a el conjunto de los estados de éstos.

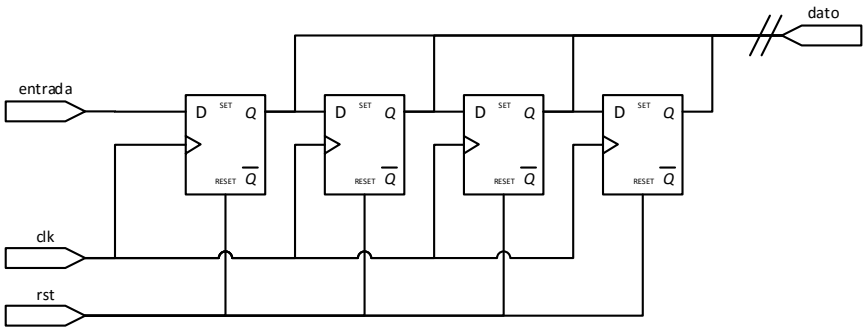


Figura 4.14: Esquema circuital de registro de desplazamiento.

Procedemos a realizar el respectivo *testbench de nuestro módulo*:

```
module test_shift_register();
  reg clk = 0;
  reg rst = 0;
  reg entrada = 0;
  wire [3:0] dato;

  shift_register uut_1(clk, rst, entrada, dato);

  always begin
    #5 clk = ~clk;
  end

  initial begin
    #5 rst <= 1;
    #10 rst <= 0;
        entrada <= 1;
    #20 entrada <= 0;
    #10 entrada <= 1;
  end
endmodule
```

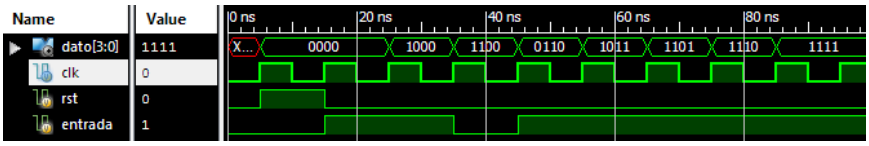


Figura 4.15: Resultados *testbench* registro de corrimiento.

En este *testbench* la señal de entrada sigue la secuencia 0→1→1→0→1. Podemos ver como tras la subida de la señal de reset, el registro toma el valor cero. Tras esto, con cada canto de subida se almacena el valor de entrada en el bit más significativo, el resto se desplaza descartando el menos significativo

Una posible aplicación para un registro de desplazamiento es hacer un reconocedor de secuencia. Por ejemplo queremos hacer un módulo que ponga su salida en alto cada vez que en la entrada se tenga la secuencia 0→0→1→1:

```
`define SECUENCIA 4'b1100
module reconocedor(clk, rst, entrada, salida);
  input clk, rst;
  input entrada;
  output salida;

  reg [3:0] dato;
  wire [3:0] dato_siguiente;

  assign dato_siguiente = {entrada, dato[3:1]};

  always @(posedge clk or posedge rst)
    if (rst)
      dato <= 4'b0;
    else
      dato <= dato_siguiente;

  assign salida = dato == `SECUENCIA;
endmodule
```

Éste módulo es básicamente el registro de corrimiento con la diferencia de que cambiamos la salida por una que es el resultado de una operación de comparación. El equivalente circuital también es bastante similar, como podemos ver en la figura 4.16. Lo único que se agrega es la lógica para determinar la salida.

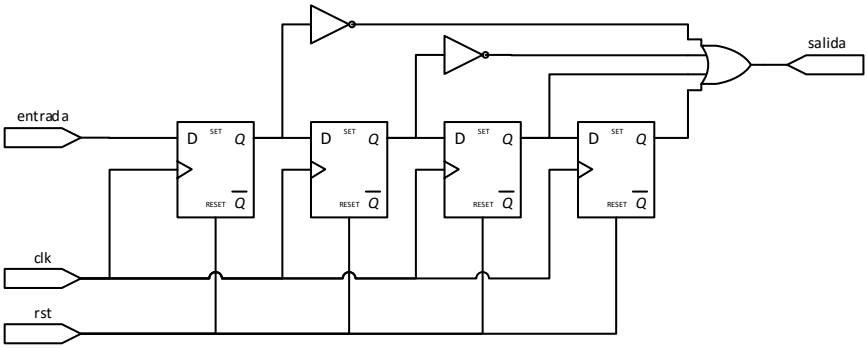


Figura 4.16: Esquema circuital de reconocedor de secuencia.

Realizamos un testbench simple para revisar su correcto funcionamiento:

```
module test_reconocedor();
  reg clk = 0;
  reg rst = 0;
  reg entrada = 0;
  wire salida;

  reconocedor uut_1(clk, rst, entrada, salida);

  always begin
    #5 clk = ~clk;
  end

  initial begin
    #5 rst <= 1;
    #10 rst <= 0;
        entrada <= 1;
    #10 entrada <= 0;
    #20 entrada <= 1;
    #20 entrada <= 0;
  end
endmodule
```

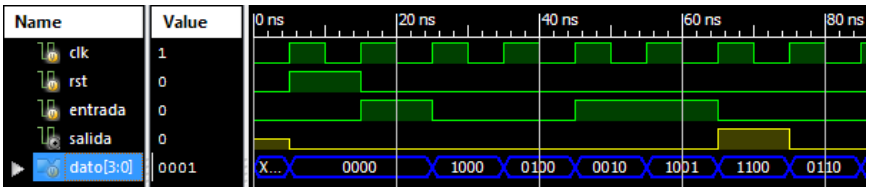


Figura 4.17: Resultados *testbench* reconocedor de secuencia.

En este testbench la entrada presenta la secuencia $0\rightarrow1\rightarrow0\rightarrow0\rightarrow1\rightarrow1\rightarrow0\rightarrow\ldots$ Observamos nuevamente el funcionamiento del registro de corrimiento y vemos que la salida sólo se pone en alto cuando el valor almacenado es el correspondiente a la secuencia deseada.

4.6 Ejemplo: Decodificador de código Morse

El código morse es un método de transmitir información utilizando una única señal que alterna entre encendido y apagado. Mediante este código es posible codificar codificar caracteres en una secuencia de "puntos" (pulsos cortos) y "rayas" (pulsos largos). Fue ampliamente utilizado en los principios de las telecomunicaciones debido a su simplicidad y facilidad de implementación.

Se quiere construir un circuito que sea capaz de reconocer los símbolos del código Morse. Para este ejemplo se utilizará una simplificación del código Morse que consta de las siguientes consideraciones:

1. Un intervalo de tiempo será considerado como un ciclo de la señal de reloj.

2. Un punto debe ser reconocido cuando la señal se mantenga en alto (uno lógico) por un único intervalo de tiempo.

3. Una raya será reconocida para cualquier pulso en alto de mayor duración.

4. Entre cada símbolo correspondiente a un mismo caracter la señal se mantendrá en bajo por un intervalo de tiempo. Si la señal se mantiene en bajo por una cantidad mayor de tiempo se considerará como fin del caracter.

Según estas especificaciones, podemos implementar un circuito lógico que conste de cuatro posibles salidas: Se recibe un punto, se recibe una raya, se recibe el término de caracter, aún no se termina de recibir.

Una posible implementación de este circuito como una máquina de Mealy se presenta en el esquema siguiente (figura 4.18).

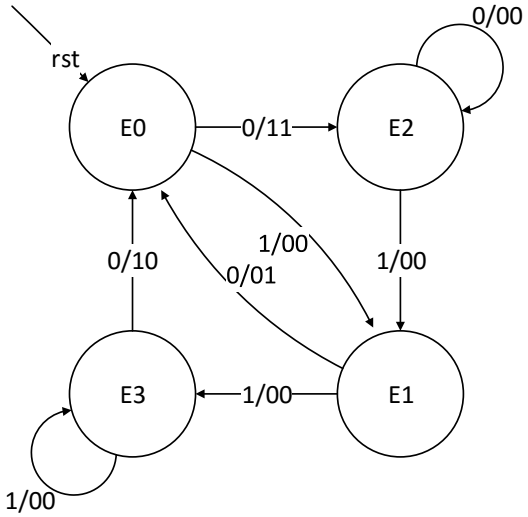


Figura 4.18: Diagrama de estados de detector de símbolos de código Morse.

Este diagrama se elaboró según las siguientes reglas de funcionamiento:

- Los estados *E0* y *E1* corresponden a los casos en que la señal de entrada a mantenido su valor durante un único intervalo de tiempo. Nivel bajo para *E0* y nivel alto para *E1*.
- De manera análoga, los estados *E2* y *E3* corresponden a los casos en que la señal de reloj se ha mantenido en alto o en bajo respectivamente, durante más de un intervalo de tiempo.
- Nuestro estado inicial será *E0* y consecuentemente, la señal de reset, nos llevará a este estado.
- La señal de salida sólo tiene un valor distinto de 00 por un ciclo de reloj, al detectar un símbolo. Se codifican las posibles salidas del circuito de la siguiente forma: 00 para indicar que aún no se recibe un símbolo; 01 para indicar que se recibe un punto; 10 para indicar una raya y 11 para indicar fin de caracter.

La anterior máquina de estados se puede implementar en Verilog de la siguiente forma:

```
`define ESTADO_E0 2'b00
`define ESTADO_E1 2'b01
```

```
`define ESTADO_E2 2'b10
`define ESTADO_E3 2'b11
`define SIMBOLO_ESPERA 2'b00
`define SIMBOLO_PUNTO 2'b01
`define SIMBOLO_RAYA 2'b10
`define SIMBOLO_FIN  2'b11

module detector_morse(clk, rst, senal, simbolo);
  input senal, clk, rst;
  output reg [1:0] simbolo;

  reg [1:0] estado, estado_next;

  //Calculo del estado siguiente
  always @(*) begin
    case(estado)
      `ESTADO_E0: estado_next = senal ? `ESTADO_E1 :
          `ESTADO_E2;
      `ESTADO_E1: estado_next = senal ? `ESTADO_E3 :
          `ESTADO_E0;
      `ESTADO_E2: estado_next = senal ? `ESTADO_E1 :
          `ESTADO_E2;
      default: estado_next = senal ? `ESTADO_E3 : `ESTADO_E0;
    endcase
  end

  //Asignacion sincronica
  always @(posedge clk or posedge rst) begin
    if (rst == 1'b1)
      estado <= `ESTADO_E0;
    else
      estado <= estado_next;
  end

  //Calculo de la salida
  always @(*) begin
    case ({estado, senal})
      {`ESTADO_E1, 1'b0}: simbolo = `SIMBOLO_PUNTO;
      {`ESTADO_E3, 1'b0}: simbolo = `SIMBOLO_RAYA;
      {`ESTADO_E0, 1'b0}: simbolo = `SIMBOLO_FIN;
      default: simbolo = `SIMBOLO_ESPERA;
    endcase
```

```
    end
endmodule
```

Podemos realizar algunas observaciones acerca de este módulo:

- Hay una clara separación entre la parte combinacional y la secuencial en el circuito. En el primer bloque always se calcula el estado siguiente, mientras que en el segundo se realiza la asignación sincrónica.

- En el primer bloque always, utilizamos un *case* sobre el estado actual para determinar el estado siguiente. Notar que se utilizó el caso *default*. Como existen casos para todos los estados excepto para *E3*, el caso *default* solo se evalúa para este estado. Si bien se podría simplemente haber incluido *ESTADO_E3* como un caso más, se sugiere siempre tener un *default* en un *case* para asegurar que se cubran todos los casos posibles.

- El cálculo de la salida siguiente se realiza dentro de un bloque always. Recordar que el uso de * como lista de sensibilidad indica que se está describiendo un circuito combinacional. Dentro de este bloque always hay un *case* sobre una concatenación. Esto es muy util para reducir las condiciones adicionales dentro de cada caso.

En la figura 4.19 se presenta el circuito equivalente a la parte sincrónica y de cálculo del estado siguíente de la máquina secuencial.

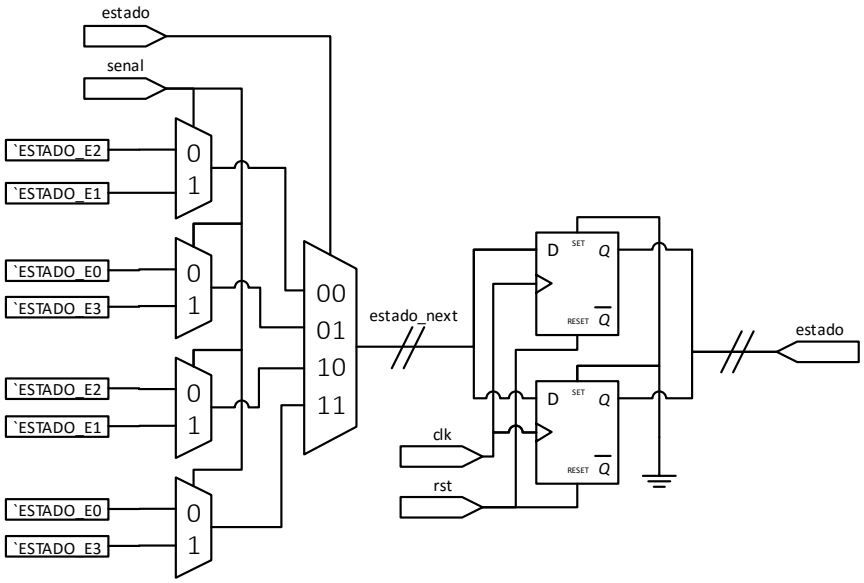


Figura 4.19: Diagrama circuital para detector de símbolos de código Morse.

En este caso en el bloque *always* que describe la parte secuencial se tienen dos señales del tipo *edge*. Como en primera instancia se pregunta por la señal *rst*, al momento de sintetizar el circuito, se inferirá que esta corresponde a las asignaciones asincrónicas de los Flip-Flops. En este caso el *rst* pone todos los bits del registro *estado* en cero, esto es equivalente a conectar la entrada *rst* a los puertos reset de los Flip-Flop, mientras que los puertos set se conectan a cero lógico.

Teniendo nuestro módulo listo, para verificar el correcto funcionamiento de nuestro módulo es importante realizar un *testbench*. El siguiente testbench realiza la prueba con una señal que posee todos los posibles símbolos que definimos anteriormente para el código morse:

```
module test_detector_morse();
  reg clk = 0;
  reg rst = 0;
  reg senal = 0;
  wire [1:0] simbolo;

  detector_morse uut_1(clk, rst, senal, simbolo);
```

```
    always begin
      #5 clk = ~clk;
    end

    initial begin
      rst <= 1;
      #5 rst <= 0;
      senal <= 1;   //punto
      #20 senal <= 0;
      #10 senal <= 1; //raya
      #10 senal <= 0;
      #10 senal <= 1; //punto
      #30 senal <= 0;
      #10 senal <= 1; //raya
      #10 senal <= 0;
    end
endmodule
```

Al ejecutar la simulación obtenemos las señales resultantes mostradas en la figura 4.20. En donde podemos ver que se identifica la siguiente secuencia esperada.

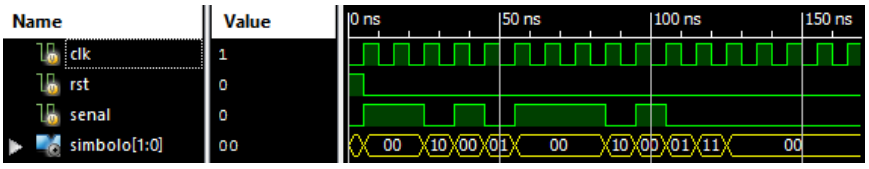


Figura 4.20: Testbench detector de símbolos de código Morse.

Primero que todo, notar que en el *testbench* realizamos la inversión de la señal *clk* cada 5 unidades de tiempo lo que significa que un ciclo completo del reloj toma 10 unidades de tiempo. Como todo nuestro circuito está síncronizado con esta señal, representará el intervalo de tiempo tras el cual se actualiza el estado de la máquina. Acorde con esto, asumiendo que la señal de entrada con la que se trabajará es una señal síncrona, para obtener los resultados correctos en la simulación, su valor es actualizado al final (o comienzo) de cada ciclo mediante asignación no bloqueante.

En un comienzo se pone en alto la señal de *rst* tras lo cual aseguramos que nuestra máquina se encuentra en su estado inicial. Si bien no se evidencia en este ejemplo, un punto importante de mencionar es la diferencia entre el comportamiento de la simulación y la implementación de las señales de *reset*. Como se mencionó anteriormente, si bien se utilizan estas en la lista

de sensibilidad como señales de tipo *edge*, al momento de implementarlas estas actúan de manera asincrónica. Sin embargo al realizar una simulación, puesto a que no estamos ligados a limitaciónes del hardware a utilizar, estas señales se comportan como se esperaría según lo descrito, es decir, todas las señales en la lista de sensibilidad actuarán por cantos si así se especifica. A causa de esto al describir circuitos secuenciales podemos encontrarnos con que si bien tenemos un *testbench* que funciona según lo descrito en todas las situaciones, al momento de implementarlo no se comporta como se esperaba.

Analizando ahora la respuesta del circuito, tenemos que en primera intancia, nuestra señal de entrada se mantiene en uno lógico durante 2 ciclos de reloj, intervalo durante el cual la salida se mantiene en 00, sólo cuando la entrada ha cambiado a cero, la salida cambia a 10, valor que le habíamos designado a la detección de una "raya". Estando sólo un ciclo en cero, la señal de entrada vuelve a uno y se mantiene ahí también por un ciclo tras lo cual la salida cambia a 01, reflejando que se detectó un "punto". A continuación la entrada se mantiene en alto por 3 ciclos durante los cuales la salida se mantiene en cero. Al bajar la señal de entrada, se detecta una "raya", con lo que se verifica que el circuito detecta este símbolo correctamente si la duración de la entrada en alto es mayor a 2 ciclos. Finalmente se tiene nuevamente un "punto", tras lo cual la señal de entrada permanece en nivel bajo, y podemos observar que al segundo ciclo de la señal en bajo tenemos en la salida 11, lo que indica término de caracter.

Habiendo verificado el funcionamiento de nuestro detector, el siguiente desafío es poder traducir la salida de éste, para poder identificar los caracteres que se estan recibiendo. En la tabla 4.5, se presenta la codificación para las primeras 5 letras del alfabeto. Se diseñará un circuito que tenga como entrada la salida del modulo detector, y como salida un código que identifique cada caracter.

Tabla 4.5: Caracteres en código morse

Caracter	Codificación
A	·—
B	— · ··
C	— · —·
D	— · ·
E	·

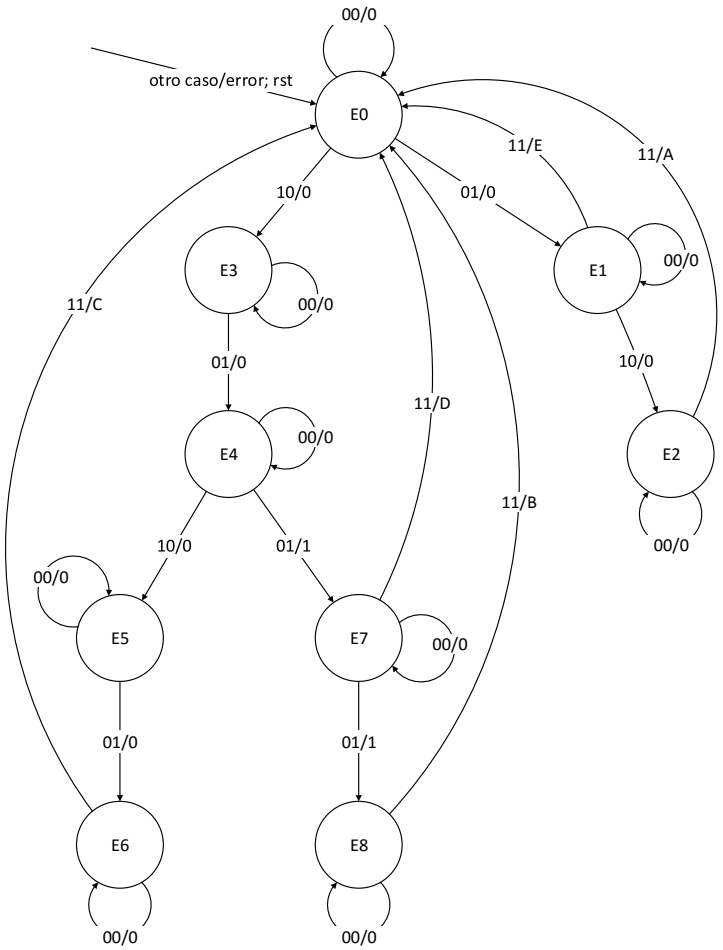


Figura 4.21: Diagrama de estados de traductor de código Morse.

En la figura 4.21 se presenta una maquina de Mealy, capaz de realizar esta tarea. En este diagrama los valores de la entrada corresponden a la salida del módulo detector diseñado anteriormente. Si bien este tiene una cantidad no menor de estados, no es dificil llegar a este diagrama, basta comprender que el estado actual debe ser resultado de los valores anteriores de la entrada, por tanto al recibir un "punto" o una "raya" se debe siempre "avanzar" de estado, mientras que si tenemos una entrada 00 nos mantenemos en el estado actual ya que no se ha detectado un símbolo. La única vez en que nos devolveremos, será en caso de obtener una señal de fin de caracter o al recibir una secuencia que no es compatible con ninguno de los caracteres definidos. En tal caso volvemos al estado inicial.

Una posible implementación de esta máquina en *Verilog* podría ser:

```
`define ESPERA 2'b00
`define PUNTO 2'b01
`define RAYA 2'b10
`define FIN  2'b11
`define CARACTER_A  3'b001
`define CARACTER_B  3'b010
`define CARACTER_C  3'b011
`define CARACTER_D  3'b100
`define CARACTER_E  3'b101
`define CARACTER_ERROR 3'b111

module traductor_morse(clk, rst, simbolo, caracter);
  input clk, rst;
  input [1:0] simbolo;
  output reg [2:0] caracter;

  reg [3:0] estado, estado_next;

  //Calculo del estado siguiente
  always @(*) begin
    if (simbolo == `FIN)
      estado_next = 4'd0;
    else if (simbolo == `ESPERA)
      estado_next = estado;
    else case(estado)
      4'd0: estado_next = simbolo == `PUNTO ? 4'd1 : 4'd3;
      4'd1: estado_next = simbolo == `PUNTO ? 4'd0 : 4'd2;
      4'd3: estado_next = simbolo == `PUNTO ? 4'd4 : 4'd0;
      4'd4: estado_next = simbolo == `PUNTO ? 4'd7 : 4'd5;
      4'd5: estado_next = simbolo == `PUNTO ? 4'd6 : 4'd0;
      4'd7: estado_next = simbolo == `PUNTO ? 4'd8 : 4'd0;
      default: estado_next = 4'b0000;
    endcase
  end

  //Asignacion sincronica
  always @(posedge clk or posedge rst) begin
    if (rst == 1'b1)
      estado <= 4'b0;
    else
      estado <= estado_next;
  end
```

```
  //Calculo de la salida
  always @(*) begin
    if (simbolo == `FIN) begin
      case (estado)
        4'd2: caracter = `CARACTER_A;
        4'd8: caracter = `CARACTER_B;
        4'd6: caracter = `CARACTER_C;
        4'd7: caracter = `CARACTER_D;
        4'd1: caracter = `CARACTER_E;
        default: caracter = `CARACTER_ERROR;
      endcase
    end
    else
      caracter = 3'b0;
  end
endmodule
```

Notar que en este módulo se hicieron uso de condiciones fuera de las estructuras *case* para casos en que una misma entrada daba como resultado el mismo comportamiento sin importar cual fuera el estado.

Teniendo este módulo, podemos ahora anexarlo al testbench realizado anteriormente y observar los resultados:

```
module test_detector_morse();
  reg clk = 0;
  reg rst = 1;
  reg senal = 0;
  wire [1:0] simbolo;
  wire [2:0] caracter;

  detector_morse uut_1(clk, rst, senal, simbolo);
  traductor_morse uut_2(clk, rst, simbolo, caracter);

  ...
endmodule
```

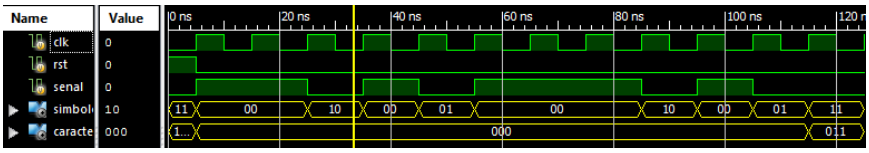


Figura 4.22: Testbench decodificador de código Morse completo.

Los resultados mostrados en la figura 4.22 se obtuvieron utilizando la misma señal de entrada utilizada en el primer *testbench* de este ejemplo. Recordar que en este se pudo observar que se detecto correctamente la secuencia: "raya", "punto", "raya", "punto" y fin de caracter. Observamos como la salida del traductor se mantiene en cero hasta recibir el símbolo de fin de caracter, tras lo cual esta cambia al valor 011, correspondiente a la letra "C" según la asignación utilizada. Podemos realizar la prueba para el resto de las letras y se verificará que estas también funcionan.

Si bien logramos describir un módulo que nos permite traducir 5 caracteres, esto fue logrado usando un diagrama de estados relativamente complejo, en donde debemos ser cuidadosos con las diferentes transiciones y asignaciones de la salida. ¿Existe una manera de realizar esto de manera más práctica?

Una posibilidad es realizar una asignación de estados de manera que sus valores se puedan calcular facilmente. Por ejemplo, podríamos dar a cada estado el valor de la secuencia de "puntos" y "rayas" actual, de esta forma nuestro estado podría ser simplemente un registro de corrimiento:

```
module traductor_morse(clk, rst, simbolo, caracter);
  input clk, rst;
  input [1:0] simbolo;
  output reg [2:0] caracter;

  reg [7:0] estado, estado_next;

  always @(*) begin
    if (simbolo == `FIN)
      estado_next = 4'd0;
    else if (simbolo == `ESPERA)
      estado_next = estado;
    else
      estado_next = {estado[5:0], simbolo};
  end
  ...
```

```
  always @(*) begin
    if (simbolo == `FIN) begin
      case (estado)
        {`PUNTO, `RAYA}: caracter = `CARACTER_A;
        {`RAYA, `PUNTO, `PUNTO, `PUNTO}: caracter =
            `CARACTER_B;
        {`RAYA, `PUNTO, `RAYA, `PUNTO}: caracter =
            `CARACTER_C;
        {`RAYA, `PUNTO, `PUNTO}: caracter =
            `CARACTER_D;
        {`PUNTO}: caracter = `CARACTER_E;
        default: caracter = `CARACTER_ERROR;
      endcase
    end
    else
      caracter = 3'b0;
  end
endmodule
```

Podemos ver que con esta estructura ya no debemos preocuparnos por las transiciones de cada estado, y podemos realizar la identificación de los caracteres de manera simple e intuitiva. Sin embargo, esto no viene sin desventajas, ahora se tiene un registro de 8 bits para almacenar el estado, lo que se traduce a 8 Flip-Flops a diferencia de los 4 utilizados en la implementación obtenida a partir del diagrama de estados. En casos en que se tienen recursos limitados, podríamos no tener la posibilidad de derrocharlos a cambio de la simplicidad de escritura.

Teniendo ahora listos ambos módulos, podemos ahora juntarlos en único módulo que realice la decodificación completa del codigo morse:

```
module decodificador_morse(clk, rst, senal, caracter);
  input clk, rst, senal;
  output [2:0] caracter

  wire [1:0] simbolo;

  detector_morse detector(clk, rst, senal, simbolo);
  traductor_morse traductor(clk, rst, simbolo, caracter);
endmodule
```

Si bien en este caso ambos módulos realizan una tarea conjunta, siempre existen beneficios en modularizar nuestros diseños. En primer lugar,

poniendo cada maquina de estado en un módulo separado, se puede tener una visión mucho más clara de cuales son las señales que participan en cada una de estas. En segundo lugar, esta estructura nos da la posibilidad de cambiar una única parte del proceso sin alterar la parte restante. Por ejemplo podríamos cambiar el módulo detector para interpretar simbolos con diferentes estándares para su duración; por otra parte podríamos cambiar el módulo traductor para leer un mensaje cuyos caracteres se codifiquen de manera diferente.

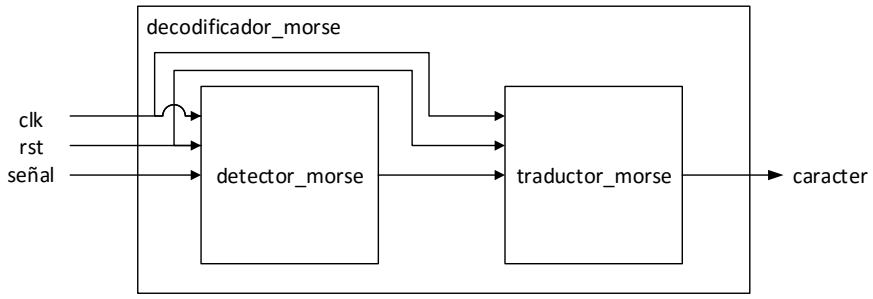


Figura 4.23: Diagrama de bloques de decodificador de código Morse.

4.7 Ejemplo: Codificador rotatorio

Los codificadores rotatorios son dispositivos cuyo propósito es seguir la posición de un eje. Por lo general esto se realiza de dos maneras: los codificadores absolutos entregan la posición del eje con respecto a una referencia sin importar su posición al momento de ser encendidos, por otra parte un codificador incremental no es consiente de la posición del eje al momento de encenderse y no entrega su posición exacta sino que entrega información acerca de como se mueve éste.

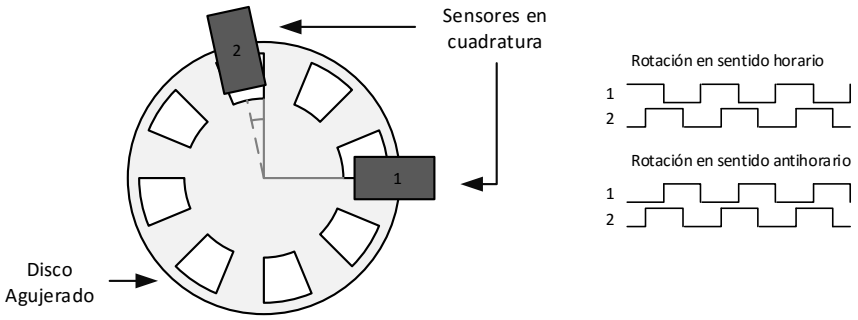


Figura 4.24: Esquema de un codificador incremental.

Una forma de implementar un codificador incremental se muestra en la figura 4.24, y consiste en usar sensores en cuadratura, es decir, en un desfase de 90 grados, con el fin de obtener dos señales que generarán un patron diferente dependiendo de la dirección de la rotación. En la figura se esquematiza un codificador que utiliza sensores ópticos, en donde un disco agujerado interrumpe un haz de luz para producir las señales.

En este ejemplo se buscará implementar un circuito secuencial que nos permita interpretar las señales entregadas por los sensores para determinar si el eje está en movimiento y en que dirección se mueve. Para esto se tienen las siguientes consideraciones:

- Las únicas señales de entrada a nuestro módulo serán las dos señales correspondientes a los sensores.
- Se tienen dos posibles secuencias para las señales de entrada, correspondientes a las dos posibles direcciones de rotación: 00→10→11→01→... y 00→01→11→10→... Notar que una secuencia es la inversa de la otra.
- Como salidas se debe tener señales que indiquen si se está en moviemiento y la dirección. Se agregará además una señal de error que indicará si hay una transición no permitida de las entradas.
- En caso de una transición no permitida, el valor de las señales de movimiento y de dirección se vuelven irrelevantes (*don't care*). La señal de dirección tampoco es relevante en el caso en que no hay movimiento.

Según esto podemos elaborar una tabla de transición de estados (tabla 4.6) y elaborar el diagrama de estados que vemos en la figura 4.25 en donde la salida representan las señales {movimiento, dirección, error}.

Tabla 4.6: Tabla de transición de estados para codificador rotatorio

Estado Actual	00	01	11	10
E0	E0/0*0	E1/100	E2/**1	E3/110
E1	E0/110	E1/0*0	E2/100	E3/**1
E2	E0/**1	E1/110	E2/0*0	E3/100
E3	E0/100	E1/**1	E2/110	E3/0*0

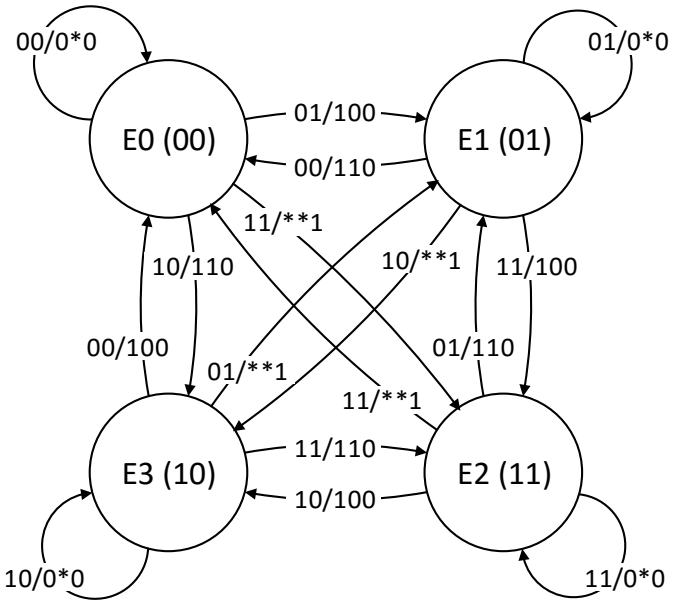


Figura 4.25: Diagrama de estados de codificador rotatorio.

Notar que el estado siguiente es sólo una función de la entrada y no depende del estado actual. Además utilizando la codificación mostrada en el diagrama de estados, el estado siguiente tiene el mismo valor de la entrada actual.

Esta máquina se puede implementar en Verilog de la siguiente manera:

```
module encoder(clk, sensores, mov, dir, error);
  input clk;
  input [1:0] sensores;
  output mov, dir, error;
```

```
  reg [1:0] estado;
  wire [1:0] estado_siguiente;

  assign estado_siguiente = sensores;
  always @(posedge clk)
    estado <= estado_siguiente;

  assign mov = sensores != estado;
  assign dir = (sensores == 2'b10 && estado == 2'b00) ||
       (sensores == 2'b11 && estado == 2'b10) ||
       (sensores == 2'b01 && estado == 2'b11) ||
       (sensores == 2'b00 && estado == 2'b01);
  assign error = (sensores ^ estado) == 2'b11; //Es verdadero si
       ambos bits de la entrada sensor son diferentes al estado.
endmodule
```

En este código se observa lo siguiente:

- Se tiene en primer lugar el cálculo del estado siguiente que, como ya mencionamos, gracias a una favorable asignación de estados corresponde simplemente a la entrada actual. Luego de lo cual tenemos un bloque *always* que responde al canto de subida de la señal de reloj y se encarga de asignar el valor siguiente al registro de estado. Notar que para este caso no se tiene una señal de reset. Esto se puede justificar por el hecho de que el estado siguiente no depende del estado actual y por tanto no hay necesidad de un estado inicial.

- Para el caso de la señal *mov* que indica si hubo movimiento, intuitivamente podemos realizar la comparación entre las señales de entrada y el estado actual. En caso de que sean diferentes indicará movimiento del disco.

- Para el valor de la señal *dir* que indica la dirección del movimiento, podemos obtener los casos en que la señal debe estar en alto directamente desde la tabla de transición de estados. De esta forma dejamos el resto de los casos, incluyendo los *don't care* en cero lógico.

- Al realizar la operacion *xor* entre dos valores se mantendrán en alto sólo los bits que sean diferentes en ambos operandos. Utilizamos esto para determinar si hay una transición no permitida, ya que estas se dan cuando hay un cambio de ambas señales de entrada de un ciclo al siguiente.

Realizamos además el respectivo *testbench* en donde simularemos giros en ambas direcciónes, momentos de detención y transiciones no permitidas:

```
module test_encoder();
  reg clk = 1'b1;
  reg [1:0] sensores = 2'b00;

  wire mov, dir, error;

  encoder uut1(clk, sensores, mov, dir, error);

  always begin
    #5 clk = ~clk;
  end

  initial begin
    #20 sensores <= 2'b01;
    #10 sensores <= 2'b11;
    #10 sensores <= 2'b10;
    #10 sensores <= 2'b00;
    #10 sensores <= 2'b11;
    #10 sensores <= 2'b01;
    #20 sensores <= 2'b00;
    #10 sensores <= 2'b10;
  end
endmodule
```

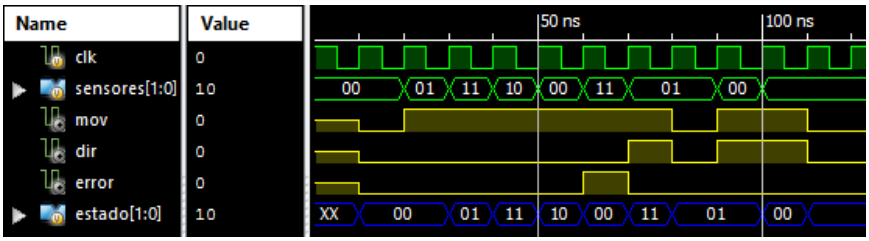


Figura 4.26: Testbench codificador rotatorio.

En la figura 4.26 tenemos los resultados de este *testbench* a partir del cual realizamos las siguientes observaciones:

- Al comienzo, la entrada se mantiene en 00 durante dos ciclos, lo que significa que no se percibe movimiento durante este lapso de tiempo y así lo refleja la salida *mov* que se mantiene en bajo.

- Tras esto, con cada ciclo de reloj se recorre la secuencia la secuencia 00→01→11→10→00 lo que significará movimiento y observamos acorde a ésto la señal de movimiento en alto y la señal de dirección en cero.
- Tenemos después la transición 00→11 con la cual se indica correctamente un error debido a que corresponde a una de las transiciones no permitidas en donde las señales de ambos sensores cambias.
- Por último, se tiene en la entrada la secuencia 11→01→00→10. Se detecta correctamente el movimiento, incluyendo la pausa intermedia, esta vez con el bit de dirección en alto lo que indica un movimiento en dirección contraria al realizado en primera instancia lo que era de esperar, puesto a que esta vez se utiliza la secuencia opuesta.

Hasta ahora nos ha sido posible identificar como se mueve el disco, pero podría ser de interés saber cual es su posición con respecto al punto de partida. Esto lo podemos realizar con otro circuito secuencial: un contador. Si bién en el capitulo 2 diseñamos un contador, éste solo permitía una cuenta ascendente. En este caso necesitaremos un contador que pueda contar en ambos sentidos, y que además se pueda habilitar y deshabilitar. Debemos añadir estas dos entradas a nuestro contador, junto con las condiciones correspondientes al cálculo del estado siguiente:

```
module contador(clk, rst, enable, sentido, cuenta);
  parameter N = 4;

  input clk, rst;
  input enable, sentido;
  output reg [N-1:0] cuenta;

  reg [N-1:0] cuenta_next;

  always @(*)
    case({enable, sentido})
      2'b10: cuenta_next = cuenta + 1'b1;
      2'b11: cuenta_next = cuenta - 1'b1;
      default: cuenta_next = cuenta;
    endcase

  always @(posedge clk or posedge rst) begin
    if (rst)
      cuenta <= 0;
```

```
    else
      cuenta <= cuenta_next;
endmodule
```

Para la lógica de los diferentes modos de operación usamos un *case* para seleccionar entre los posibles valores para la cuenta siguiente, mientras que para la asignación secuencial usamos las consideraciones usuales: se pregunta primero por las señales asincrónicas, en este caso *rst*. Notar además, que el tamaño de la cuenta puede ser ajustado mediante el parámetro N para obtener un contador del tamaño que se desee.

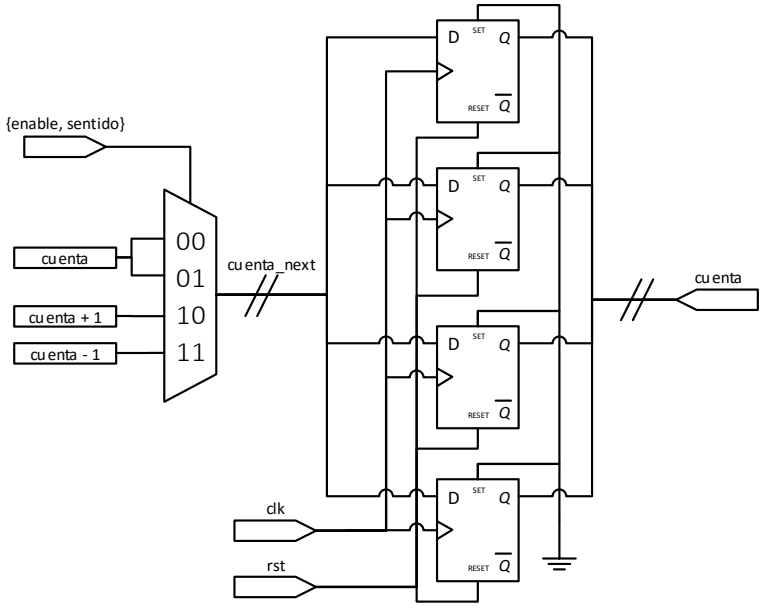


Figura 4.27: Diagrama circuital de módulo contador.

En la figura 4.27 se presenta el diagrama circuital de nuestro módulo contador, con las nuevas funciones añadidas. Para la parte combinacional, la estructura case se puede interpretar como un multiplexor cuyas entradas son los diferentes casos y señal de control seran las entradas de habilitación y sentido. En cuanto a la parte secuencial, la cuenta actual se debe almacenar en los Flip-Flops. Como nuestra señal de *reset* siempre lleva cuenta a cero, ésta se conectará a los puertos *reset* de los Flip-Flops, mientras que el *set* no se utilizará y por tanto se conecta a cero lógico. La señal por la que no

preguntamos, en este caso *clk*, se conectará entonces a la entrada de reloj de los Flip-Flops.

Notar que en nuestro módulo utilizamos un parámetro para especificar el tamaño de la cuenta. En términos del circuito esto significará simplemente la un cambio del tamaño en bits de las señales (de los buses) y la cantidad de Flip-Flops utilizados.

Ahora necesitamos que nuestro módulo contador trabaje en conjunto con la máquina de estados para el codificador. Claro está que para lograr ésto, la entrada *enable* será conectada a la salida *mov* mientras que *sentido* se conecta con *dir*. Podemos ahora realizar un *testbench* con ambos módulos:

```
module test_encoder();
  reg clk = 1'b1;
  reg rst = 1'b1;
  reg [1:0] sensores = 2'b00;

  wire mov, dir, error;
  wire [4:0] cuenta;

  encoder uut1(clk, sensores, mov, dir, error);
  contador #(5) uut2(clk, rst, mov, dir, cuenta);

  always begin
    #5 clk = ~clk;
  end

  initial begin
    rst <= 0;
    repeat (3) begin
    #10 sensores <= 2'b01;
    #10 sensores <= 2'b11;
    #10 sensores <= 2'b10;
    #10 sensores <= 2'b00;
    end
    #10;
    repeat (2) begin
    #10 sensores <= 2'b10;
    #10 sensores <= 2'b11;
    #10 sensores <= 2'b01;
    #10 sensores <= 2'b00;
    end
  end
endmodule
```

Los resultados de este *testbench* los vemos en la figura 4.28. En éste se utiliza la palabra clave *repeat* para repetir sucesivamente los patrones en la entrada. Podemos ver que para el primer patron la cuenta aumenta en uno con cada ciclo de reloj, mientras que el segundo corresponde a la dirección opuesta con lo que se obtiene una cuenta descendente. En este *testbench* podemos también observar las formas de las entradas las cuales podemos comparar con las mostradas anteriormente en la figura 4.24.

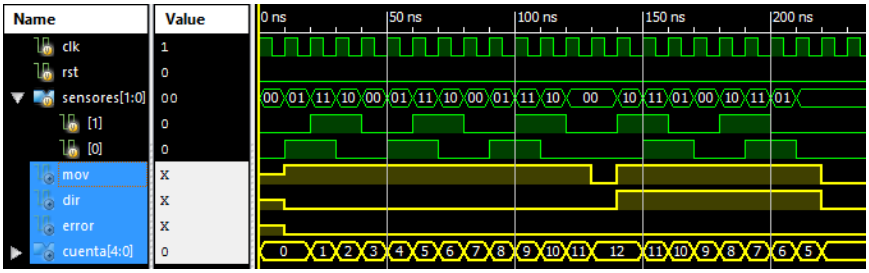


Figura 4.28: Testbench codificador rotatorio junto con contador.

4.8 Fuentes

- Nagle, H. Troy, Bill D. Carroll, and J. David Irwin. "Introduction to Sequential Devices." *An Introduction to Computer Logic*. Englewood Cliffs, NJ: Prentice-Hall, 1974.
- http://nrich.maths.org/2198
- http://machinedesign.com/sensors/basics-rotary-encoders-overview-and-new-technologies-0

Capítulo 5

Mi primer diseño

Por ejemplo: el crónometro

Incluir testbench

Estrategias de diseño: Cómo mejorar nuestro hardware

- modular vs. no modular
- Diseño según estados

¿Cómo implementar? Describir cómo se implementa el diseño

Capítulo 6

Buenas prácticas en Verilog

Casos específicos

- Un if siempre con un else
- If-else anidados
- El default obligatorio del Case (explicación)
- listas de sensibilidad
- máximo número de cantos en la lista de sensibilidad
- Eventos asincrónico
- Sincrozación de reloj externo
- Errores más comunes
- Asignaciones bloqueantes vs. no bloqueantes: diferencias fundamentales
- Tips de implementación (errores comunes)
- ...

Capítulo 7

El Multiplexor y la ALU

Partiendo con Alu de 4 bits Mux uni y bidimensionales Recalcar el circuito que hay de trasfondo

Testbench

Implementación

Capítulo 8

Encriptación y Desencriptación

Ver algoritmo con Daniel
- testbench
- Implementación

Capítulo 9

Pulsadores, displays, switches y leds

Ejemplo: contadores, expansión de hardware (usando sensores, como los de proximidad o presencia)

Testbench

Implementación

Capítulo 10

Puertos: Serial, VGA, PS2, Ethernet y USB

Capítulo 11

Robots (motores)

Capítulo 12

El ascensor

Capítulo 13

Casos avanzados

pines in/out estados tri Memoria (ram, eeprom)