

Computadores Avanzados

Manual de Verilog

Antonio Morán Muñoz

Manual de Verilog



Diseño Digital

Manual de Verilog

Antonio Morán Muñoz
Instituto de Investigación en Informática (I3A)

Esta publicación está basada en la plantilla NIST Handbook de Overleaf

Septiembre 2023



Colegio Oficial de
Ingeniería Informática
de Castilla-La Mancha

Departamento de Sistemas Informáticos

Universidad de Castilla-La Mancha

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by the National Institute of Standards and Technology, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

National Institute of Standards and Technology Handbook XXXX
Natl. Inst. Stand. Technol. Handbook XXXX, xxi pages (Month Year)

This publication is available free of charge from:
<https://doi.org/10.6028/NIST.HB.XXXX>

Foreword

Delete if not applicable

Preface

Delete if not applicable

Abstract

Required

Key words

Required, alphabetized, separated by semicolon, and end in a period.

Table of Contents

1 Simulación	iv
2 Bancos de pruebas	v
3 Circuitos combinacionales	vi
3.1 Decodificadores	vi
3.1.1 Tipos de decodificadores	viii
3.2 Codificadores binarios	viii
3.3 Multiplexores	viii
3.4 Demultiplexores	viii
4 Dispositivos triestado	ix
4.1 Buffers triestado	ix
4.2 Codificación con prioridad	ix
4.3 Puertas OR-exclusivas y funciones de paridad	ix
4.4 Comparadores	ix
4.5 Elementos combinacionales aritméticos	x
5 Circuitos secuenciales	xii
5.1 Máquinas de estado.	xii
5.1.1 Estructura de las máquinas de estado	xiii
5.1.2 Temporización de las máquinas de estados	xiv
5.2 Diseño de máquinas de estado con tablas de estado	xiv
5.3 Diseño de máquinas de estado con diagramas de estado	xiv
5.4 Diseño de máquinas de estado con Verilog	xiv
6 Glosario	xviii
References	xx
Appendix A: Supplemental Materials	xxi
Appendix B: Change Log	xxi

List of Tables

Table 1 Tabla de salidas y estados	xv
------------------------------------	----

List of Figures

Fig. 1 short	xiii
Fig. 2 short	xiv

Glossary

Delete if not applicable

1. Simulación

Una vez que la tenemos un modelo de Verilog con sintaxis y semántica correcta, se puede y debe utilizar un simulador para observar su operación.

El simulador opera comenzando en un *tiempo de simulación* de cero. En este punto, el simulador inicializa todas las señales a un valor por defecto de x. A continuación, el simulador inicializa las señales o variables cuyos valores iniciales han sido declarados explícitamente. Tras esto, el simulador comienza la ejecución de las sentencias concurrentes del diseño.

Aclaración

El simulador en realidad no puede ejecutar las sentencias concurrentes simultáneamente. Sin embargo, crea esa ilusión mediante el uso de una *lista de eventos* basados en tiempo, y una matriz de sensibilidad basada en todas las listas de sensibilidad individuales.

Cada sentencia concurrente se da lugar a un proceso software, mientras que una instanciación de módulo puede dar lugar a uno o más procesos. En tiempo de simulación cero, todos los procesos generados se planifican para ejecución, de los cuales se selecciona uno. La ejecución del código procedural se lleva a cabo secuencialmente. Si se ha especificado una demora, la ejecución del proceso se suspende. Cuando esto ocurre (o cuando el proceso se ha terminado de ejecutar) se selecciona otro proceso para su ejecución.

Cuando todos los procesos se han ejecutado, se completa lo que se denomina un *ciclo de simulación*.

Cuando el simulador se encuentra una asignación no bloqueante sin demora, ésta se supone que se ejecuta en tiempo de simulación cero. No obstante, lo que ocurre realmente es que se planifica su ejecución para el tiempo de simulación actual al que se le suma una *demora delta*.

Cada vez que un ciclo de simulación es completado, la lista de eventos es escaneada para buscar las señales que antes cambien. Debido a que algunos procesos pueden ser sensibles al cambio de señales, la matriz de sensibilidad indica, para cada señal, qué procesos tienen dicha señal en su lista de sensibilidad. Cada proceso que sea sensible a una señal, será planificado para que se ejecute en el siguiente ciclo de simulación. Este proceso se repite hasta que la lista de eventos queda vacía, momento en el cual la simulación termina.

2. Bancos de pruebas

Un *banco de prueba* especifica una secuencia de entradas que serán aplicadas por el simulador al modelo HDL que será probado. Dicho modelo puede ser un módulo Verilog o un diseño más grande. La entidad sujeta a las pruebas se denomina *unidad bajo prueba (UBP)*.

En bancos de pruebas es frecuente emplear otro tipo de sentencia concurrente, el bloque *initial*. Esta sentencia se ejecuta una sola vez, en tiempo de simulación cero. Al igual que el bloque *always*, se puede incluir un bloque *begin-end* para declarar el código procedural.

Listing 1. Banco de prueba para un circuito detector de numeros primos.

```
1  `timescale 1 ns / 100 ps
2
3  module Vrprime_tb1();
4  reg [3:0] Num;
5  wire Prime;
6
7  Vrprimed UUT (.N(Num), .F(Prime));
8
9      initial begin: TB
10         integer i;
11         for (i = 0; i <= 15; i = i+1) begin #10
12             Num = i;
13         end
14     endmodule
```

Los simuladores suelen generar un fichero que especifica las formas de onda que cada señal toma durante la simulación. Dicho fichero puede ser leído por aplicaciones específicas para su visualización. El correcto funcionamiento la UBP quedará a cargo del usuario, cuya función es interpretar si las formas de onda tienen sentido. Debido a que este proceso es tedioso, es conveniente formatear la salida del simulador para que sea más amigable. Para ello se pueden usar tareas del sistema como `$write` o `$display` para mostrar información que se considere útil por consola o, incluso, redirigirla a un fichero.

Hay situaciones en las que la combinación de entradas que una UBP puede recibir es tan elevada. En estos casos, puede no resultar factible para el diseñador examinar los resultados obtenidos para cada combinación de entradas. En estos casos, conviene escribir un banco de prueba autoevaluado

3. Circuitos combinacionales

3.1 Decodificadores

A pesar de que los decodificadores en Verilog suelen formar parte de diseños mayores, también se pueden definir y probar de manera aislada. Para diseñar un decodificador, existen las siguientes aproximaciones:

- *Estilo estructural.* Consistiría en definir la operación del decodificador mediante puertas lógicas. Esta aproximación daría lugar a un código difícil de leer y de mantener.
- *Estilo de flujo de datos.* Se define el comportamiento del circuito mediante sentencias de asignación continua. El Listing 2 muestra un posible diseño.

Listing 2. Estilo de flujo de datos.

```
1 module decodificador_2a4(input a0 ,
2                           input a1 ,
3                           input en ,
4                           output y0 ,
5                           output y1 ,
6                           output y2 ,
7                           output y3);
8     assign y0 = en ? ({a1,a0} == 2'b00) : 0;
9     assign y1 = en ? ({a1,a0} == 2'b01) : 0;
10    assign y2 = en ? ({a1,a0} == 2'b10) : 0;
11    assign y3 = en ? ({a1,a0} == 2'b11) : 0;
12 endmodule
```

- *Estilo conductual.* Se modela el circuito por medio de código prodedural. El Listing 3 muestra un posible diseño.

Listing 3. Estilo conductual.

```
1 module decodificador_2a4(input a0 ,
2                           input a1 ,
3                           input en ,
4                           output reg y0 ,
5                           output reg y1 ,
6                           output reg y2 ,
7                           output reg y3);
8
9     always @(*) begin
10         if(en) begin
11             {y3,y2,y1,y0} = 0
12             case ({a1,a0})
```

```

13             2'b00: y0 = 1;
14             2'b00: y1 = 1;
15             2'b00: y2 = 1;
16             2'b00: y3 = 1;
17             default: {y3,y2,y1,y0} = 0;
18         endcase
19     end
20 end
21
22 endmodule

```

Una vez diseñado el circuito, es hora de probarlo.

Listing 4. Banco de pruebas del decodificador.

```

1  `timescale 1 ns / 1 ps
2  module tb_decodificador_2a4();
3
4      // declaracion de tipos de datos
5      reg a0_p, a1_p, en_p;
6      wire y0_p, y1_p, y2_p, y3_p;
7      integer i, errores;
8      reg [3:0] resultado_esperado;
9
10     decodificador_2a4 UBP (.a0(a0_p),
11                           .a1(a1_p),
12                           .en(en_p),
13                           .y0(y0_p),
14                           .y1(y1_p),
15                           .y2(y2_p),
16                           .y3(y3_p)
17                           );
18     initial begin
19         errors = 0;
20         for(i = 0; i < 8; i++) begin
21             {en, a0, a1} = i;
22             #10;
23             resultado_esperado = 4'b0000;
24             if(en) resultado_esperado[{a1,a0}] = 1'b1;
25             if ({y3,y2,y1,y0} !== resultado_esperado)
26                 begin
27                     errores = errores + 1;
28                 end
29             end
30         end
31         $display("Test completado con %d errores", errores);
32     end

```

```

30     end
31
32 endmodule

```

Si la compilación con iverilog ha sido correcta, el banco de pruebas debería mostrar 0 errores.

3.1.1 Tipos de decodificadores

- *Decodificadores binarios.* Un decodificador binario n -a- 2^n es un circuito con una entrada de n bits y una salida activa de las 2^n totales.
- *Decodificadores de siete segmentos.* Estos circuitos tienen una entrada de 4 bits (Decimal Codificado en Binario) y una salida de 1 bit para cada segmento del LCD.

3.2 Codificadores binarios

Un codificador binario es un circuito que hace la acción opuesta al decodificador binario. Si un decodificador binario convierte un código de n bits en otro de m bits, el codificador binario correspondiente debe ser capaz de transformar el código de m bit en el de n bits.

3.3 Multiplexores

Una operación muy común en diseño digital es escoger una fuente de datos entre varias para transferirla a su destino a través de un medio compartido, como un bus. Esta operación es tan común que tiene su propio nombre, *multiplexación* o *mux* para abreviar.

Una definición más precisa de multiplexador es la de *conmutador digital que conecta los datos de 1 entre n fuentes a una sola salida*. Para ello, al multiplexor hay que conectarle una entrada que permita hacer una selección entre n fuentes. La anchura de esta entrada de selección tendrá que ser $\log_2(n)$, donde n es el número de fuentes.

3.4 Demultiplexores

Los demultiplexores son los dispositivos digitales que realizan la operación opuesta a los multiplexores, es decir, conectar una entrada (de b bits) a una de entre n destinos/salidas.

4. Dispositivos triestado

Los *dispositivos triestado* son muy importantes porque, a nivel de la placa de circuito impreso son una alternativa ampliamente utilizada a los multiplexores. Existen diferentes tipos de dispositivos triestado.

4.1 Buffers triestado

Es el dispositivo triestado más sencillo. También se le puede llamar *conductor triestado*. El buffer triestado es como un buffer normal al que se le conecta una tercera señal llamada *habilitación triestado*. Cuando esta señal está activa, el buffer actúa como un buffer corriente, conectando la entrada a la salida. Por el contrario, cuando habilitación triestado es cero, la salida pasa a un estado de alta impedancia y el buffer se comporta como si ni siquiera estuviese ahí.

Los buffers triestado son raramente usados en chip, es decir, dentro de ASICs o FPGAs, ya que los multiplexores suelen ofrecer un mayor desempeño.

4.2 Codificación con prioridad

En la sección de codificadores binarios (**REFERENCIAR**) se vio que estos dispositivos seleccionan la entrada activa de entre 2^n para codificarla en la salida pero, ¿qué ocurre si varias entradas están activas al mismo tiempo? En este caso convendría asignar una *prioridad* a cada una para que la entrada con más prioridad sea la seleccionada por el codificador. Los dispositivos que hacen esto se denominan *codificadores con prioridad*.

Los codificadores con prioridad tienen una salida *OCIOSA* que se activa si ninguna de las entradas está activa.

4.3 Puertas OR-exclusivas y funciones de paridad

Las puertas *OR-exclusivas* son importante en cuatro aplicaciones:

- *Comparación.*
- *Generación de paridad.*
- *Suma.*
- *Contaje.*

4.4 Comparadores

La comparación de dos palabras binarias es una operación frecuentemente usada en computadores. El circuito que se encarga de comparar dos palabras binarias y de decir si son iguales o no se denomina *comparador*. Algunos comparadores son capaces de interpretar las palabras a comparar como numeros con o sin signo, y realizar comparaciones de mayor/menor o igual. En este caso hablamos de comparadores de magnitudes.

4.5 Elementos combinacionales aritméticos

Estos elementos incluyen circuitos que son capaces de llevar a cabo funciones aritméticas, como sumas, multiplicaciones, divisiones o desplazamientos. Normalmente existen operadores que se encargan de estas operaciones, dejando al sintetizador la responsabilidad de generar el circuito a partir de dicho operador. En el caso concreto de las FPGA, estas consisten exclusivamente de TCs (LUTs) interconectadas, por lo que será responsabilidad del sintetizador adaptar el código a dicha infraestructura. Los elementos combinacionales aritméticos son los siguientes:

- *Semisumador y Sumador completo.* El semisumador suma dos operandos de un bit produciendo una suma de dos bits, el bit de bajo orden es la *semisuma* (S) mientras que el de alto orden es el *acarreo* ($COUT$). Los sumadores completos realizan la misma operación de suma pero con números de más de un bit. Un sumador completo tiene, además de los operandos y el resultado de dos bit, una señal de entrada para el acarreo de la suma del bit inmediatamente anterior CIN .
- *Sumador con propagación.* Estos circuitos suman palabras de n bits y consisten en una cascada de n sumadores completos, cada uno de los cuales se encarga de la suma de un bit. El $COUT$ de cada sumador completo se conecta con el CIN del sumador completo siguiente (exceptuando la del último)¹. Además, al CIN del sumador completo menos significativo se le suele conectar un valor de cero. La desventaja de este tipo de circuitos es que son lentos, especialmente aquellos que sumen números muy grandes.
- *Sumador con anticipación de acarreo.* Este sumador mejora el desempeño del sumador con propagación. Al igual que en el anterior, en este sumador existen una serie de fases que suman un bit cada una. Pero en este caso ocurren pueden ocurrir dos cosas.
 1. *Generación de acarreo.* Esto ocurre cuando una fase i produce un acarreo de salida independientemente de que haya acarreo de entrada, es decir, cuando ambos sumandos valen uno.
 2. *Propagación de acarreo.* Esto ocurre cuando se genera un acarreo siguiendo la lógica anterior, es decir, cuando ambos sumandos valen uno.
- *Sumadores con propagación grupal*
- *Sumadores con anticipación de acarreo grupal*
- *Restadores.* Realizan la operación opuesta al sumador. **DESARROLLAR.**

¹De ahí el nombre de este sumador, ya que el acarreo se va propagando a través del circuito.

- *Desplazadores y Rotadores.* El desplazamiento y la rotación son dos operaciones muy utilizadas. El desplazamiento es la operación de mover los bits de una palabra de n bits una o mas posiciones hacia la izquierda/derecha. Los bits que se caigan de dicha palabra serán reemplazados por nuevos bits (normalmente ceros). La rotación funciona de manera similar solo que en este caso los bits que se caigan retornan llenando las posiciones de los bits vacantes en el otro extremo de la palabra ². El circuito más representativo de esta categoría es el Desplazador en barril.
- *Multiplicadores*
- *Divisores*

²Por esto, a la rotación se le suele denominar también desplazamiento circular.

5. Circuitos secuenciales

Aunque hemos comentado con profusión los circuitos combinacionales, lo que realmente impera en la práctica son los circuitos secuenciales. Mientras que hemos visto que en los circuitos combinacionales la salida depende exclusivamente de los valores de las entradas en el momento actual, en los circuitos secuenciales se tienen los valores de las entradas en el pasado. Por ejemplo, si tenemos un ventilador con diferentes velocidades y queremos aumentarla, tenemos que saber algo más aparte de una entrada que indique el incremento de la velocidad. Es decir, tendría que haber un estado que nos indique la velocidad actual del sistema para saber cómo incrementar la velocidad. Es aquí donde entran en juego las *máquinas de estado*.

5.1 Máquinas de estado.

El funcionamiento de un sistema como el ventilador puede ser conceptualizada como un conjunto de estados interrelacionados, es decir, una máquina de estados. El estado de un circuito secuencial puede definirse como “una colección de *variables de estado* cuyos valores en cualquier instante contienen toda la información acerca del pasado necesaria para predecir el comportamiento del circuito en el futuro” **CITAR LIBRO DE HERERT HELLERMAN DIGITAL COMPUTER SYSTEM PRINCIPLES DE 1967**. No obstante, son necesarias más cosas aparte de variables de estado, por ejemplo, se necesita saber hacia qué estado. Este no solo dependerá del valor de las entradas del sistema, sino del estado en el que se encuentre en ese momento. En el ejemplo del ventilador, si activamos la señal de incremento de la velocidad y nos encontramos en el estado *VELOCIDAD_MEDIA*, el circuito se moverá al estado *VELOCIDAD_ALTA*. Por el contrario, para la misma entrada, si el circuito se encontrara en el estado *VELOCIDAD_BAJA*, éste cambiaría a *VELOCIDAD_MEDIA*. Como el número de estados de un sistema no es infinito, las máquinas de estado (y los circuitos secuenciales en general) se suelen conocer también por el nombre de *máquinas de estados finitos*.

En los circuitos secuenciales cobra gran importancia el *reloj*, ya que los cambios entre estados deben llevarse a cabo de manera sincronizada. En la mayoría de sistemas, el cambio de estado se lleva a cabo en los *flancos* del reloj, o *flancos activos/gatillo*, ya sea de subida o de bajada (aunque es más frecuente en el de subida). Una señal de reloj es *activa en alto* si los cambios de estado se producen en el flanco de subida, o *activa en bajo* si los cambios se llevan a cabo en el flanco de bajada. Este tipo de relojes debe estar activo, funcionando a una frecuencia dada y regular, mientras el circuito esté activo. Esta señal de reloj con frecuencia es generada por un oscilador de cristal de cuarzo, cuya frecuencia varía dependiendo del circuito en el que se integre.

Otro elemento a tener en cuenta en los circuitos secuenciales es el de biestable. Estos son elementos con memoria, que permiten mantener el valor de las variables de estado, entre otras cosas. Un biestable se puede implementar por medio de un *circuito secuencial retroalimentado*, en el cual un bucle de retroalimentación se incluye en el circuito secuencial para añadir memoria al sistema. Hay multitud de tipos de biestables:

- *Bi estable SR.*
- *Bi estable JK.*
- *Bi estable D.*
- *Bi estable T.*

5.1.1 Estructura de las máquinas de estado

La mayoría de máquinas de estado actuales son *máquinas de estado sincronizadas por reloj* que usan biestables D disparados por flanco. Son sincronizados por reloj porque todos los biestables están conectados a la misma señal de reloj, cambiando al mismo tiempo su estado en respuesta a los flancos del reloj. Las Figuras 1 y 2 muestran la estructura general de una máquina de estados finitos de Mealy y de Moore, respectivamente. Ambas tienen en común una *memoria de estado*, que está constituida por un conjunto de n biestables que almacenan el estado del circuito. A estos biestables se les conecta una *señal de reloj* común, de tal manera que el estado sólo cambia en los *tics* del reloj. El *tic* (se podría definir como el instante en el que cambia el estado) depende del tipo de biestable usado. Por ejemplo, para los biestables que se activan en el flanco de subida, el *tic* es el flanco de subida del reloj. El estado siguiente de la máquina de estados viene determinado por la *lógica del estado siguiente* que es función del estado actual y las entradas. En cambio, la *lógica de salida* depende del tipo de circuito del que se trate. En las máquinas de estados finitos de Mealy, la lógica de salida viene determinada por las entradas y el estado actual.

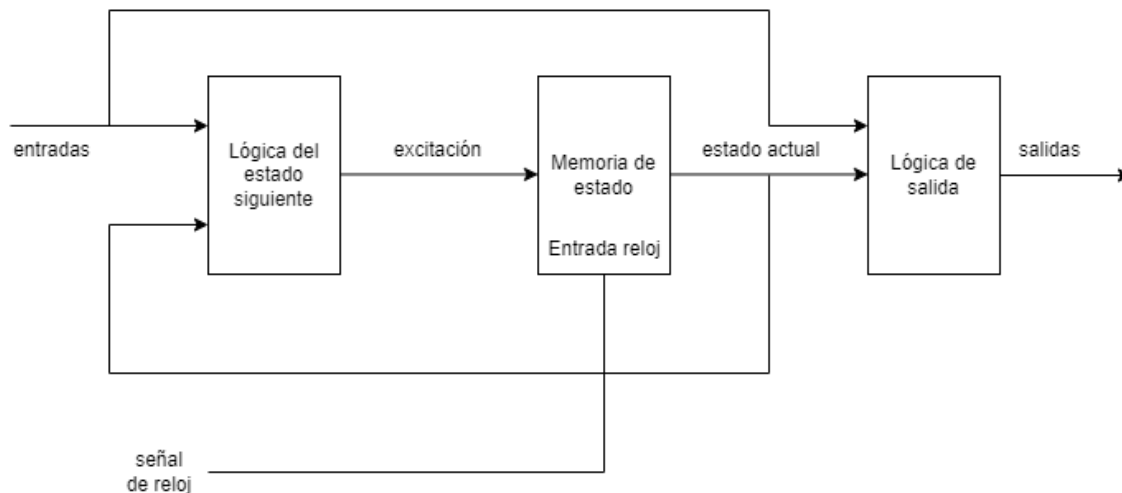


Fig. 1. Máquina de estados finitos de Mealy.

En cambio, en una máquina de estados finitos de Moore, la lógica de salida viene determinada exclusivamente del estado actual.

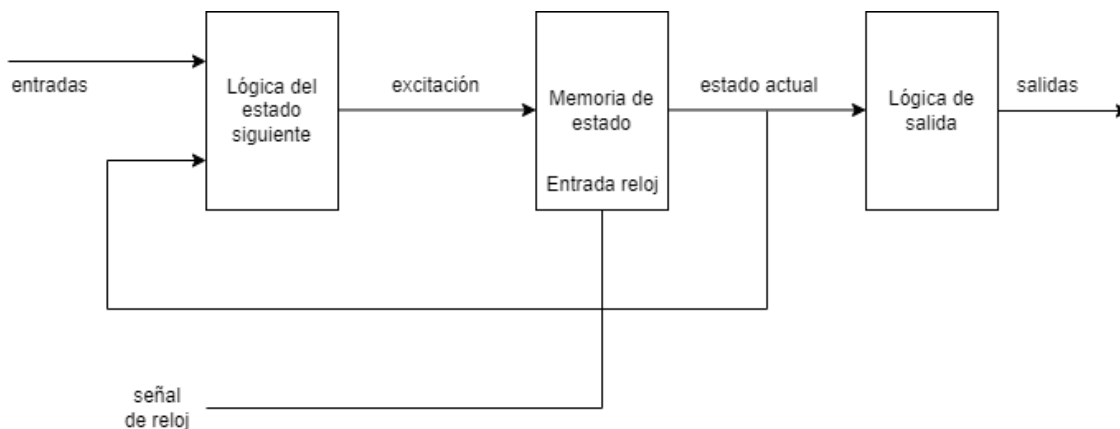


Fig. 2. Máquina de estados finitos de Moore.

Como se puede observar en los diagramas, la única diferencia entre ambos tipos de modelos radica en cómo se generan las salidas.

En el diseño de circuitos de gran velocidad es necesario que las salidas estén disponibles lo antes posible. Por ello, en ocasiones se codifican las variables de estado para que sirvan de salida también. Este tipo de diseño se denomina *asignación de estados codificada en salidas*. En este caso la salida del circuito consiste en wires. Otra manera de abordar el diseño de una máquina de estados es haciendo que la salida durante un periodo de reloj dependa de la salida en el periodo de reloj anterior. Esto se denomina *salida por etapas* y se consigue uniendo otra etapa de memoria, llamada *memoria de salida por etapas*, a las salidas del circuito.

5.1.2 Temporización de las máquinas de estados

POR HACER

5.2 Diseño de máquinas de estado con tablas de estado

POR HACER (pag. 455)

5.3 Diseño de máquinas de estado con diagramas de estado

POR HACER (pag. 472)

5.4 Diseño de máquinas de estado con Verilog

Para explicar el diseño de máquinas de estados vamos a partir de las siguientes especificaciones:

Diseña una máquina de estados síncrona con reloj con dos entradas, A y B, y una salida Z que es 1 si:

<i>S</i>	<i>AB</i>				<i>Z</i>
	<i>00</i>	<i>01</i>	<i>11</i>	<i>10</i>	
INIT	A0	A0	A1	A1	0
A0	OK0	OK0	A1	A1	0
A1	A0	A0	OK1	OK1	0
OK0	OK0	OK0	OK1	A1	1
OK1	A0	OK0	OK1	OK1	1
<i>S*</i>					

Table 1. Tabla de salidas y estados

- A tiene el mismo valor que en los dos tics de reloj previos.
- B vale uno tras la última vez que la primera condición fue cierta.

Si las condiciones no se cumplen, Z vale 0.

Para visualizar mejor estas condiciones, lo mejor es comenzar describiendo una tabla de estados y salidas. La Tabla 1 muestra el resultado sobre el cual vamos a trabajar para escribir el código en Verilog.

El correspondiente módulo en Verilog se compondrá de cinco secciones:

1. La declaración de las entradas, salidas y variables internas al módulo.
2. Sentencias parameter para asignar las variables de estado a su nombre.
3. Un primer bloque always para crear la memoria de estado.
4. Un segundo bloque always para definir el comportamiento del estado siguiente.
5. Un tercer bloque always para definir la lógica de salida.

El código en Verilog resultante se muestra en el Listing 5.

Listing 5. Máquina de estado en Verilog.

```

1 module mef (
2     // reloj y reset
3     input clk,
4     input rst,
5
6     // entradas
7     input A,
8     input B,
9

```

```

10         // salidas (como vamos a usar codigo procedural,
11         hay que declarar la salida como reg)
12         output reg Z
13     );
14
15 // senales internas
16
17 reg estado, estado_siguiente;
18
19 // asignacion de variables de estado a parametros
20 parameter [2:0] INIT = 3'b000;
21             A0 = 3'b001;
22             A1 = 3'b010;
23             OK0 = 3'b011;
24             OK1 = 3'b100;
25
26 // comportamiento del estado siguiente
27 always @(*) begin
28     case(estado)
29         INIT: if (A == 0) estado_siguiente = A0;
30             else estado_siguiente = A1;
31         A0: if (A == 0) estado_siguiente = OK0;
32            else estado_siguiente = A1;
33         A1: if (A == 0) estado_siguiente = A0;
34            else estado_siguiente = OK1;
35         OK0: if (A == 0) estado_siguiente = OK0;
36            else if ((A == 0) && (B == 1)) estado_siguiente =
37                OK1;
38            else estado_siguiente = A1;
39         OK1: if ((A == 0) && (B == 0)) estado_siguiente = A0;
40            else if ((A == 0) && (B == 1)) estado_siguiente = OK0;
41            else estado_siguiente = OK1;
42         default: estado_siguiente = INIT;
43     endcase
44 end
45
46 // logica de salida
47 always @(estado) begin
48     case(estado)
49         INIT: Z = 0;
50         A0: Z = 0;
51         A1: Z = 0;
52         OK0: Z = 1;
53         OK1: Z = 1;
54         default: Z = 0;
55     endcase

```

```
53 end
54
55 // memoria de estado
56 always @(posedge clk) begin
57     if(rst)
58         estado <= 0;
59     else
60         estado <= estado_siguiete;
61 end
```

6. Glosario

- Sentencia de demora (*delay statement*).
- Tiempo de simulación (*simulation time*).
- Lista de eventos (*event list*).
- Matriz de sensibilidad (*sensitivity matrix*).
- Ciclo de simulación (*simulation cycle*).
- Banco de prueba (*test bench*).
- Unidad bajo prueba (*unit under test (UUT)*).
- Decodificador de siete segmentos (*seven-segment decoder*).
- Multiplexación (*multiplexing*).
- Dispositivo triestado (*three-state device*).
- Conductor triestado (*three-state driver*).
- Habilitación triestado (*three-state enable*).
- Codificador con prioridad (*priority encoder*).
- OCIOSA (*IDLE*).
- Comparador (*comparator*).
- Comparador de magnitudes (*magnitude comparator*).
- Semisumador (*half adder*).
- Sumador completo (*full adder*).
- Sumador con propagación (*ripple adder*).
- Sumador con anticipación de acarreo (*carry-lookahead adder*).
- Sumador con propagación grupal (*carry-ripple adder*).
- Sumador con anticipación de acarreo grupal (*group-lookahead adder*).
- Restador (*subtractor*).
- Desplazador (*shifter*).

- Rotador (*rotator*).
- Desplazador en barril (*barrel shifter*).
- Máquina de estados (*state-machine*).
- Flanco (*edge*).
- Flanco activo/gatillo (*active/triggering edge*).
- Activo en alto (*active high*).
- Activo en bajo (*active low*).
- Biestable (*flip-flop*).
- Circuito secuencial retroalimentado (*feedback sequential circuit*).
- Assignación de estados codificada en salidas (*output-coded state assignment*).
- Salida por etapas (*pipelined outputs*).

Acknowledgments

Delete if not applicable

References

- [1] Wilkinson JP (1990) Nonlinear resonant circuit devices. United States Patent 3 624 125.
- [2] Joint Task Force Transformation Initiative Interagency Working Group (2013) Security and privacy controls for federal information systems and organizations (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-53, Rev. 4, Includes updates as of January 22, 2015. <https://doi.org/10.6028/NIST.SP.800-53r4>
- [3] National Institute of Standards and Technology (2001) Security requirements for cryptographic modules (U.S. Department of Commerce, Washington, D.C.), Federal Information Processing Standards Publications (FIPS PUBS) 140-2, Change Notice 2 December 03, 2002. <https://doi.org/10.6028/NIST.FIPS.140-2>
- [4] Xiong H (2015) Multi-level bell-type inequality from information causality and noisy computations. *Chinese Journal of Electronics* 24(2):408–413. <https://doi.org/10.1049/cje.2015.04.031>
- [5] Prives L (2016) For whom the bell tolls: Inventing success through creativity and analytical skills [wie from around the world]. *IEEE Women in Engineering Magazine* 10(1):37–39. <https://doi.org/10.1109/MWIE.2016.2535841>
- [6] Roberts LJ (1982) Cameras and systems: A history of contributions from the bell; howell co. (part i). *SMPTE Journal* 91(10):934–946. <https://doi.org/10.5594/J00229>
- [7] Maloney TJ (2016) Unified model of 1-d pulsed heating, combining wunsch-bell with the dwyer curve: This paper is co-copyrighted by intel corporation and the esd association. *38th Electrical Overstress/Electrostatic Discharge Symposium (EOS/ESD)* (Publisher name, location), Vol. 22, pp 1–8. <https://doi.org/10.1109/EOS/ESD.2016.7592562>
- [8] Giancoli D (2008) *Physics for Scientists and Engineers with Modern Physics* (Pearson Education), 4th Ed.
- [9] Eston P (1993) *Book section title* (The name of the publisher, The address of the publisher), Vol. 4, Chapter 8, 3rd Ed., pp 201–213.
- [10] Behrends R, Dillon LK, Fleming SD, Stirewalt REK (2006) White paper: Programming according to the fences and gates model for developing assured, secure software systems (Department of Computer Science, Michigan State University, East Lansing, Michigan), MSU-CSE-06-2.
- [11] Farindon P (1993) The title of the collection section. *The title of the book*, ed Lastname F (The name of the publisher, The address of the publisher), Vol. 4, pp 201–213.
- [12] Marcheford P (1993) The title of the unpublished work.

- [13] Joslin P (1993) *The title of the PhD Thesis*. Ph.D. thesis. The school of the thesis, The address of the publisher. An optional note.
- [14] Caxton P (1993) The title of the booklet. How it was published, The address of the publisher. An optional note.
- [15] Isley P (1993) The title of the webpage. Available at <https://nist.gov>.

Appendix A: Supplemental Materials

Brief description of supplemental files

Appendix B: Change Log

If updating document with errata, detail changes made to document – delete if not applicable.