Computadores Avanzados

Manual de Verilog

Antonio Morán Muñoz

Manual de Verilog

Escuela superior de ingeniería informática

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

National Institute of Standards and Technology Handbook XXXX
Natl. Inst. Stand. Technol. Handbook XXXX, 28 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

List of Tables

List of Figures

Glossary

Delete if not applicable

0.1. 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 procesos 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.

0.2. 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

0.3. Circuitos combinacionales

0.3.1 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 contínua. El Listing 2 muestra un posible diseño.

Listing 2. Estilo de flujo de datos.

```
module decodificador_2a4(input a0,
                         input a1,
                         input en,
                         output y0,
                         output y1,
                         output y2,
                         output y3);
    assign y0 = en ? ({a1,a0} == 2'b00) : 0;
    assign y1 = en ? ({a1,a0} == 2'b01) : 0;
    assign y2 = en ? ({a1,a0} == 2'b10) : 0;
    assign y3 = en ? ({a1,a0} == 2'b11) : 0;
endmodule
```

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

Listing 3. Estilo conductual.

```
module decodificador_2a4(input a0,
                         input a1,
                         input en,
                         output reg y0,
                         output reg y1,
                         output reg y2,
                         output reg y3);

    always @(*) begin
        if(en) begin
            {y3,y2,y1,y0} = 0
            case ({a1,a0})
```

```
                2'b00: y0 = 1;
                2'b00: y1 = 1;
                2'b00: y2 = 1;
                2'b00: y3 = 1;
                default: {y3,y2,y1,y0} = 0;
            endcase
        end
    end

endmodule
```

Una vez diseñado el circuito, es hora de probarlo.

Listing 4. Banco de pruebas del decodificador.

```
`timescale 1 ns / 1 ps
module tb_decodificador_2a4();

    // declaracion de tipos de datos
    reg a0_p, a1_p, en_p;
    wire y0_p, y1_p, y2_p, y3_p;
    integer i, errores;
    reg [3:0] resultado_esperado;

    decodificador_2a4 UBP (.a0(a0_p),
                           .a1(a1_p),
                           .en(en_p),
                           .y0(y0_p),
                           .y1(y1_p),
                           .y2(y2_p),
                           .y3(y3_p)
                          );
    initial begin
        errors = 0;
        for(i = 0; i < 8; i++) begin
            {en, a0, a1} = i;
            #10;
            resultado_esperado = 4'b0000;
            if(en) resultado_esperado[{a1,a0}] = 1'b1;
            if ({y3,y2,y1,y0} !== resultado_esperado)
               begin
                errores = errores + 1;
            end
        end
        $display("Test completado con %d errores", errores
           );
```

```
30      end
31
32  endmodule
```

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

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.

0.3.2 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.

0.3.3 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 este entrada de selección tendra que ser $log_2(n)$, donde n es el número de fuentes.

0.3.4 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.

0.4. 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.

0.4.1 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.

0.4.2 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.

0.4.3 Puertas OR-exclusivas y funciones de paridad

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

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

0.4.4 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.

0.4.5 Elementos combinacionales aritméticos

Estos elementos incluyen circuitos que son capaces de llevar a cabo funciones aritmeticas, 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.

[1] De 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*

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

Chapter 1

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.*

1.1. 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 númerod 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 estár 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:

- *Biestable SR.*
- *Biestable JK.*
- *Biestable D.*
- *Biestable T.*

1.1.1 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 1.1 y 1.2 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, the 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.

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

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*

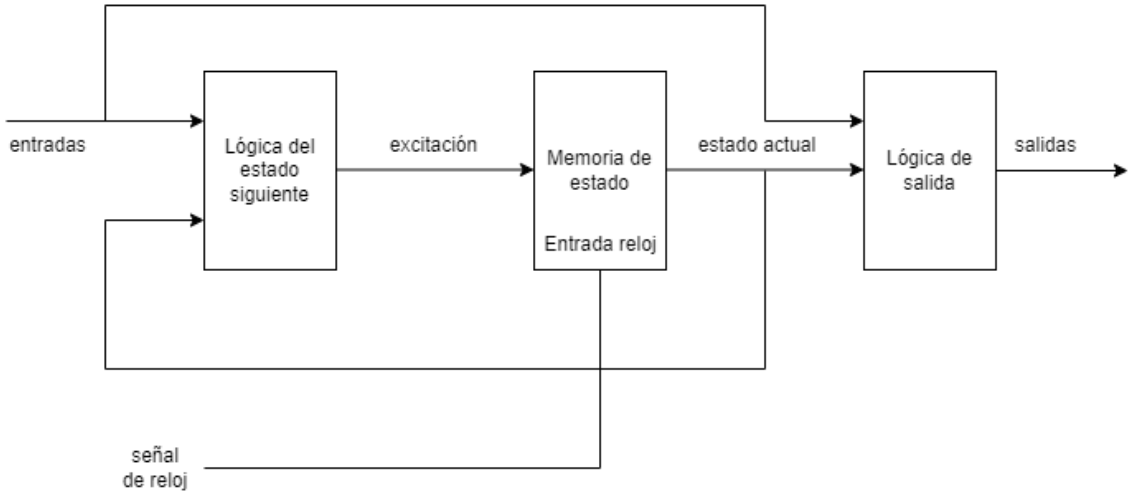


Fig. 1.1. Máquina de estados finitos de Mealy.

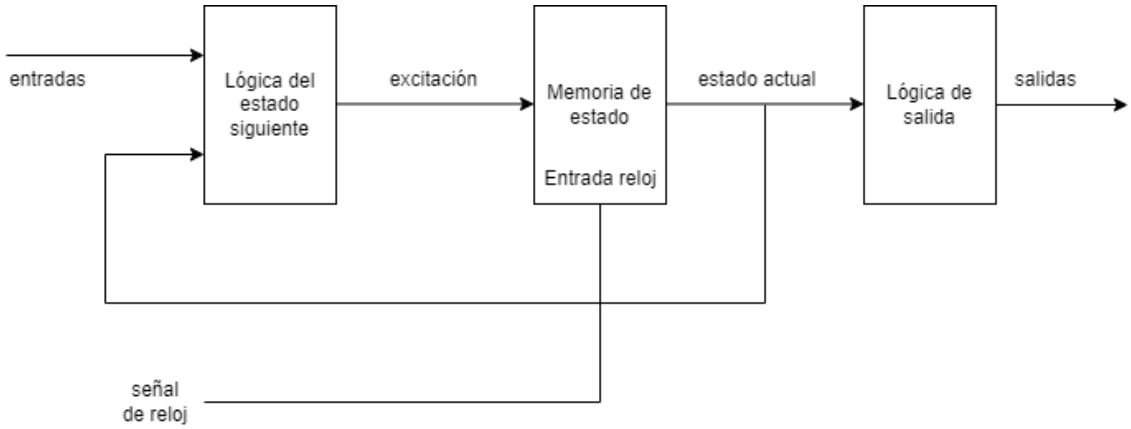


Fig. 1.2. Máquina de estados finitos de Moore.

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.

1.1.2 Temporización de las máquinas de estados

POR HACER

1.2. Diseño de máquinas de estado con tablas de estado

POR HACER (pag. 455)

	AB				
S	***00***	***01***	***11***	***10***	***Z***
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
	S*				

Table 1.1. Tabla de salidas y estados

1.3. Diseño de máquinas de estado con diagramas de estado

POR HACER (pag. 472)

1.4. 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:

- 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 1.1 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 1.1.

Listing 1.1. Máquina de estado en Verilog.

```
module mef (
             // reloj y reset
             input clk,
             input rst,

             // entradas
             input A,
             input B,

             // salidas (como vamos a usar codigo procedural,
                hay que declarar la salida como reg)
             output reg Z
            );

// senales internas
reg estado, estado_siguiente;

// asignacion de variables de estado a parametros
parameter [2:0] INIT = 3'b000;
                A0 = 3'b001;
                A1 = 3'b010;
                OK0 = 3'b011;
                OK1 = 3'b100;

// comportamiento del estado siguiente
always @(*) begin
    case(estado)
        INIT: if (A == 0) estado_siguiente = A0;
              else estado_siguiente = A1;
        A0: if (A == 0) estado_siguiente = OK0;
            else estado_siguiente = A1;
        A1: if (A == 0) estado_siguiente = A0;
            else estado_siguiente = OK1;
        OK0: if (A == 0) estado_siguiente = OK0;
             else if ((A == 0) && (B == 1)) estado_siguiente =
                 OK1;
             else estado_siguiente = A1;
        OK1: if ((A == 0) && (B == 0))
           estado_siguiente = A0;
        else if ((A == 0) && (B == 1)) estado_siguiente = OK0;
        else estado_siguiente = OK1;
        default: estado_siguiente = INIT;
    endcase
end
```

```
42
43 // logica de salida
44 always @(estado) begin
45     case(estado)
46         INIT: Z = 0;
47         A0: Z = 0;
48         A1: Z = 0;
49         OK0: Z = 1;
50         OK1: Z = 1;
51         default: Z = 0;
52     endcase
53 end
54
55 // memoria de estado
56 always @(posedge clk) begin
57     if(rst)
58         estado <= 0;
59     else
60         estado <= estado_siguiente;
61 end
62 endmodule
```

En el Listing 1.2 se muestra el banco de pruebas en C++ [1]. En dicho banco de pruebas vemos cómo se instancia un objeto de la UBP (`Vmef *dut = new Vmef;`). Dicho objeto es generado con Verilator mediante el siguiente comando.

```
verilator -Wall --trace -cc <nombre-de-UBP>.sv
```

Lo que hace dicho comando es leer el código en Verilog/SystemVerilog (en este caso SystemVerilog) y lo compila en un modelo en C++. Dicho modelo consta de una serie de ficheros `.cpp` y `.h` que se alojan dentro del directorio `obj_dir` situado en la ruta desde la cual se ejecutó el comando.

Listing 1.2. Banco de pruebas en C++ que se simula con Verilator.

```
1     #include <stdlib.h>
2     #include <iostream>
3     #include <bitset>
4     #include <verilated.h>
5     #include <verilated_vcd_c.h>
6     //#include <verilated_dpi.h>
7     #include "Vmef.h"
```

[1] En la documentación oficial, este fichero en C++ que sirve de banco de pruebas, se denomina ”wrapper” (lo cual se podría traducir como envoltorio, ya que envuelve la instancia del modelo de la UBP generada por Verilator).

```
8     #include "svdpi.h"
9     #include "Vmef__Dpi.h"
10
11    #define ESTADOS 5
12    char estado = 0;
13    vluint64_t tiempo = 0;
14    int errores = 0;
15
16    using namespace std;
17
18    // funciones
19    void reset(Vmef *dut, VerilatedVcdC *m_trace);
20    void tic(Vmef *dut, VerilatedVcdC *m_trace);
21    void prueba_estado(Vmef *dut, VerilatedVcdC *m_trace, int
         a, int b, int init);
22    void comprueba(const string salida, const string esperado,
          const string prueba, int i, int j);
23
24    // DPI
25    extern void sv_method();
26
27    int main(int argc, char** argv, char** env) {
28        // instanciamos la UBP
29        Vmef *dut = new Vmef;
30
31        Verilated::scopesDump();
32
33        // set the scope
34        const svScope scope = svGetScopeFromName("TOP.mef");
35        assert(scope);
36        svSetScope(scope);
37
38        // activamos la generacion de traza
39        Verilated::traceEverOn(true);
40        VerilatedVcdC *m_trace = new VerilatedVcdC;
41        dut->trace(m_trace, 5);
42        m_trace->open("waveform.vcd");
43
44        // reseteamos la UBP
45        reset(dut, m_trace);
46
47        // PRUEBAS
48        // generamos un tic
49        tic(dut, m_trace);
50
```

```
51          // INIT
52          prueba_estado(dut, m_trace, 0, 0, 0);
53          // A0
54          prueba_estado(dut, m_trace, 0, 0, 1);
55          // A1
56          prueba_estado(dut, m_trace, 0, 0, 2);
57          // OK0
58          prueba_estado(dut, m_trace, 0, 0, 3);
59          // OK1
60          prueba_estado(dut, m_trace, 0, 0, 4);
61
62          // FIN PRUEBAS
63          cout << "El numero de errores totales es " << errores
                << "\n\n";
64
65
66          m_trace->close();
67          delete dut;
68          exit(EXIT_SUCCESS);
69      }
70
71      void reset(Vmef *dut, VerilatedVcdC *m_trace){
72          dut->rst = 1;
73          tic(dut, m_trace);
74          dut->rst = 0;
75          tic(dut, m_trace);
76      }
77
78      void tic(Vmef *dut, VerilatedVcdC *m_trace){
79          dut->clk = 0;
80          dut->eval();
81          // volcamos
82          m_trace->dump(tiempo);
83          tiempo++;
84          dut->clk = 1;
85          dut->eval();
86          // volcamos
87          m_trace->dump(tiempo);
88          tiempo++;
89
90      }
91
92      void prueba_estado(Vmef *dut, VerilatedVcdC *m_trace, int
           a, int b, int inicial){
93          int i,j, entrada = 0;
```

```
94          int intEst, intEst2;
95          int contador = 0;
96          string estado_prueba = "INIT";
97          string estado = "INIT", esperado = "INIT";
98
99          // mostramos el estado al entrar en la funcion
100         std::cout << "\n\n
               ----------------------------------------";
101         std::cout << "\nEl estado al entrar en la funcion es "
                << estado << '\n';
102         std::cout << "----------------------------------------"
               ;
103
104         // for anidado para probar diferentes combinaciones de
                entrada
105         for(i=0; i<2; i++){
106             for(j=0; j<2; j++){
107                 dut->A = 0;
108                 dut->B = 0;
109                 // inicializamos el estado
110                 reset(dut, m_trace);
111                 // nos movemos al estado que queremos probar
112                 switch(inicial){
113                     case 0: reset(dut, m_trace);
114                             tic(dut, m_trace);
115                             break;
116                     case 1: dut->A = 0;
117                             tic(dut, m_trace);
118                             break;
119                     case 2: dut->A = 1;
120                             tic(dut, m_trace);
121                             break;
122                     case 3: dut->A = 0;
123                             tic(dut, m_trace);
124                             dut->A = 0;
125                             tic(dut, m_trace);
126                             break;
127                     case 4: dut->A = 1;
128                             tic(dut, m_trace);
129                             dut->A = 1;
130                             tic(dut, m_trace);
131                             break;
132                     default: estado = "INIT";
133                              tic(dut, m_trace);
134                 }
```

```
135
136                 // comprobamos que el estado inicial es el
                       correcto
137                 std::bitset<3> est(sv_get_estado());
138                 intEst = static_cast<int>(est[7]) ? (est.
                       to_ulong() - 256) : est.to_ulong();
139
140                 switch(intEst){
141                     case 0: estado_prueba = "INIT"; break;
142                     case 1: estado_prueba = "A0"; break;
143                     case 2: estado_prueba = "A1"; break;
144                     case 3: estado_prueba = "OK0"; break;
145                     case 4: estado_prueba = "OK1"; break;
146                     default: estado_prueba = "INIT";
147                 }
148
149                 std::cout << "\n
                       ----------------------------------------\n";
150                 std::cout << "Estado inicializado a " <<
                       estado_prueba << '\n';
151                 std::cout << "
                       ----------------------------------------" <<
                        '\n';
152
153
154                 // calculamos el estado esperado
155                 switch(intEst){
156                     case 0: if((i == 0) && (j == 0)){
157                                 esperado = "A0";
158                             } else if ((i == 0) && (j ==
                                   1)){
159                                 esperado = "A0";
160                             } else if ((i == 1) && (j ==
                                   0)){
161                                 esperado = "A1";
162                             } else if ((i == 1) && (j ==
                                   1)) {
163                                 esperado = "A1";
164                             } else esperado = "INIT";
165                             break;
166                     case 1: if((i == 0) && (j == 0)){
167                                 esperado = "OK0";
168                             } else if ((i == 0) && (j ==
                                   1)){
169                                 esperado = "OK0";
```

```
170                                 } else if ((i == 1) && (j ==
                                        0)){
171                                     esperado = "A1";
172                                 } else if ((i == 1) && (j ==
                                        1)) {
173                                     esperado = "A1";
174                                 } else esperado = "INIT";
175                                 break;
176                         case 2: if((i == 0) && (j == 0)){
177                                     esperado = "A0";
178                                 } else if ((i == 0) && (j ==
                                        1)){
179                                     esperado = "A0";
180                                 } else if ((i == 1) && (j ==
                                        0)){
181                                     esperado = "OK1";
182                                 } else if ((i == 1) && (j ==
                                        1)) {
183                                     esperado = "OK1";
184                                 } else esperado = "INIT";
185                                 break;
186                         case 3: if((i == 0) && (j == 0)){
187                                     esperado = "OK0";
188                                 } else if ((i == 0) && (j ==
                                        1)){
189                                     esperado = "OK0";
190                                 } else if ((i == 1) && (j ==
                                        0)){
191                                     esperado = "A1";
192                                 } else if ((i == 1) && (j ==
                                        1)) {
193                                     esperado = "OK1";
194                                 } else esperado = "INIT";
195                                 break;
196                         case 4: if((i == 0) && (j == 0)){
197                                     esperado = "A0";
198                                 } else if ((i == 0) && (j ==
                                        1)){
199                                     esperado = "OK0";
200                                 } else if ((i == 1) && (j ==
                                        0)){
201                                     esperado = "OK1";
202                                 } else if ((i == 1) && (j ==
                                        1)) {
203                                     esperado = "OK1";
```

```
204                                 } else esperado = "INIT";
205                                 break;
206                         default: esperado = "INIT";
207                 }
208
209
210                 // asignamos las combinaciones de entradas a
                       dicho estado
211                 //std::cout << '\n' << "("<< estado_prueba <<
                       ") A = " << i << "\tB = " << j << '\n';
212                 //std::cout <<
                       "----------------------------------------"
                       << '\n';
213                 dut->A = i;
214                 dut->B = j;
215                 tic(dut, m_trace);
216
217                // comprobamos que el estado final es el
                       correcto
218                 std::bitset<3> est2(sv_get_estado());
219                 intEst2 = static_cast<int>(est2[7]) ? (est2.
                       to_ulong() - 256) : est2.to_ulong();
220
221                 switch(intEst2){
222                     case 0: estado = "INIT"; break;
223                     case 1: estado = "A0"; break;
224                     case 2: estado = "A1"; break;
225                     case 3: estado = "OK0"; break;
226                     case 4: estado = "OK1"; break;
227                     default: estado = "INIT";
228                 }
229
230                 comprueba(estado, esperado, estado_prueba, i,
                       j);
231
232                 contador++;
233             }
234         }
235     }
236
237     void comprueba(const string salida, const string esperado,
            const string prueba, int i, int j){
238         if(salida.compare(esperado) != 0){
239             cout << "\nEl resultado es erroneo, fracasado";
240             std::cout << "\n
```

```
                ----------------------------------------\n";
241             std::cout << "A=" << i << " B=" << j << " -> " <<
                    salida << " != " << esperado << ", inutil mas
                    que inutil." << '\n';
242             std::cout << "
                    ----------------------------------------" << '\n
                    ';
243
244             errores++;
245         }
246     }
```

1.5. Otros elementos de lógica secuencial

A parte de los ya mencionados biestables-D, usados para almacenar el estado de las máquinas de estados finitos, hay otros tipos de elementos de almacenamiento más apropiados para situaciones que no requieran de máquinas de estados finitos. Uno de estos elementos son los *latches*. Estos permiten almacenar información basada en el nivel de una entrada de control con la mitad del coste que un *biestable disparado por flanco* en términos de área del circuito. A continuación, se mostrarán algunos elementos usados en lógica secuencial por complejidad creciente.

1.5.1 Biestable

A pesar de que se ha mencionado el biestable D en secciones anteriores, no se ha llegado a explicar en que consiste. El biestable D es una evolución del biestable[2], que consiste en un par de *inversores* dispuestos de tal manera que forman un *bucle de retroalimentación*. Estos biestables son los más simples y tienen dos salidas, Q y Q_L, pero no tienen entradas (véase la Figura 1.3).

El biestable se llama así, porque el análisis digital muestra que tiene dos estados estables. Si Q es ALTO, entonces Q_L es BAJO, lo cual refuerza el estado de Q. De esta manera tenemos dos estados estables, ALTO y BAJO. Lo mismo ocurre cuando Q es BAJO, ya que esto causa que Q_L sea ALTO, lo que refuerza que Q sea BAJO. Al no disponer de ningún elemento de control o que nos permita cambiar el estado, al iniciar el biestable, este genera aleatoriamente el estado y se mantiene invariable para siempre.

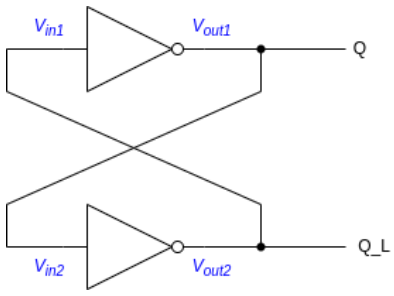


Fig. 1.3. Biestable.

[2]En este caso, biestable se traduce al inglés como bistable, mientras que los biestables que se verán más adelante se traducen como flip-flops.

¿HABLAR DE METAESTABILIDAD AQUÍ?

1.5.2 Latches y biestables

La diferencia entre latch y flip-flop es que mientras el latch refleja inmediatamente en la salida los cambios en la entrada, los flip-flops solo reflejan este cambio en los tics de reloj. A continuación se explican las diferentes variantes de latches y flip-flops.

Latch S-R

Los *Latch S-R* son el circuito secuencial más simple que se puede construir y que tenga entradas de control. Se puede construir con dos puertas NOR de dos entradas (véase la Figura 1.4).

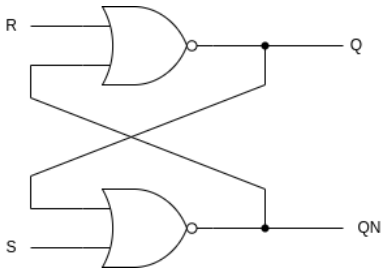
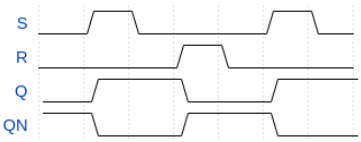
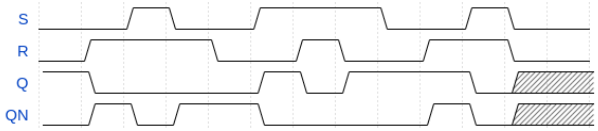


Fig. 1.4. Latch S-R.

La 'S' y la 'R' vienen del inglés *set* y *reset*, y son las señales de control del circuito. Las salidas se denominan 'Q' y 'QN' y no tienen por qué ser una el complemento de la otra, a pesar de que normalmente en la mayoría de las aplicaciones lo suele ser. Por ejemplo, como se observa en la Figura 1.5b, si S y R valen 1 en el mismo instante de tiempo, las salidas cambian ambas a 0. Además, si se niegan tanto S como R al mismo tiempo, el circuito entra en un estado de metaestabilidad, haciendo el siguiente estado de las salidas impredecible. No obstante, si las entradas se niegan en instantes diferentes, las salidas retienen el estado anterior. Esto otorga al circuito su capacidad de almacenamiento.



(a) Operación normal de un latch S-R.



(b) Operación anómala cuando 'S' y 'R' son afirmados a la vez.

Fig. 1.5. Diagrama de ondas de un latch S-R

1.6. 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 *(substractor)*.
- 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)*.
- Biestable disparado por flanco *(edge-triggered flip-flop)*.
-
- Latch S-R *(S-R latch)*.

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/EOSESD.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.