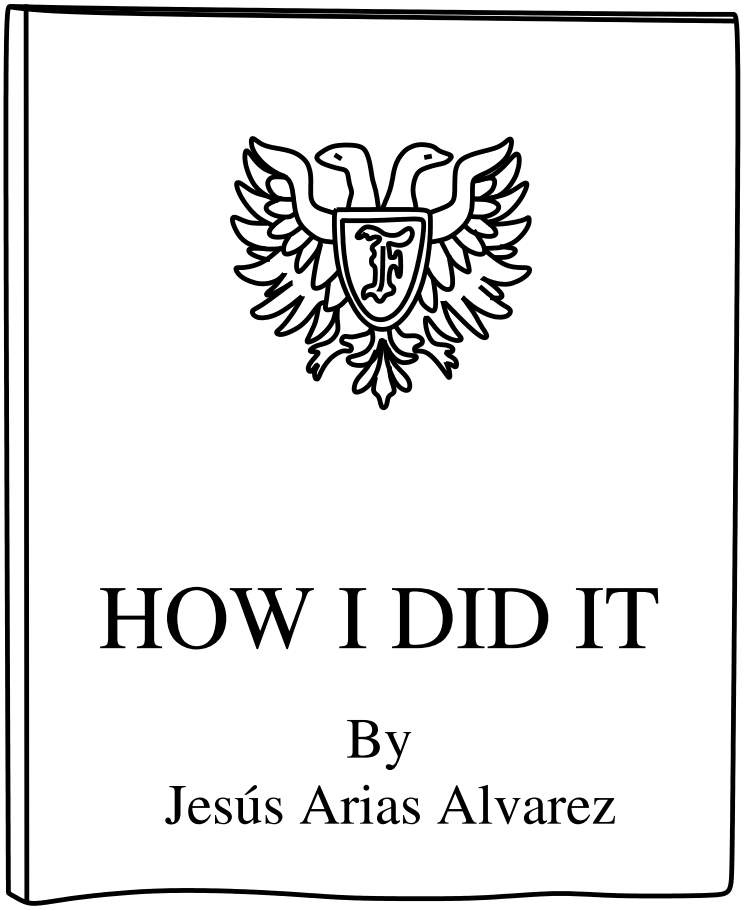
Diseño Verilog.



1 Introducción.

Con permiso de Victor Fronkonstin (¿o era Frankenstein?) para copiarle el título, en este documento pretendo recopilar una serie de ejemplos de circuitos digitales escritos en Verilog con la idea de ilustrar de forma rápida cómo describir circuitos similares durante el diseño de sistemas digitales complejos. Este documento no es una introducción al lenguaje Verilog. Este lenguaje de descripción de hardware es harto complejo, redundante, confuso, y con una curva de aprendizaje lenta. Sin embargo tampoco es necesario ser un experto en Verilog para poder diseñar bloques de circuitos digitales, simular su comportamiento, e incluso sintetizar dichos circuitos en una FPGA. Por ello he pensado que unos ejemplos de circuito que muestren alguna de las muchas formas posibles en las que se pueden describir en lenguaje Verilog pueden ser de ayuda para quienes tengan que enfrentarse al problema de un diseño similar.

Así que sin más preámbulos comencemos.

Contents

2 Módulos, Diseño jerárquico, Test Benches

En Verilog un circuito digital está encapsulado dentro de un módulo que presenta al exterior una serie de terminales de entrada “input” y de salida “output”. También pueden tener señales bidireccionales “inout”, aunque estas últimas apenas se pueden usar en la síntesis en FPGA, por lo que es buena idea evitarlas. Estos módulos se pueden a su vez incluir en otros módulos de nivel superior como “instancias”, hasta llegar al módulo más alto de la jerarquía. Este módulo puede ser un banco de pruebas para simulación “testbench”, o alternativamente un módulo tonto que dirija las señales hacia los pines concretos de la FPGA en el caso de la síntesis.

A continuación se muestra un ejemplo de módulo sencillo, con 4 bits de entrada y uno de salida. La salida se pone en 1 si alguno de los bits de entrada es uno (OR lógica):

```
module puerta_or4(
    input [3:0]in,
    output out
);
assign out = (in!=0);
endmodule
```

En este módulo se incluye la lógica combinacional necesaria en la línea “assign”, donde a su vez se hace una comparación lógica de una variable de 4 bits, “in”, con una constante de valor 0. Esto se corresponde con la tabla de verdad de una puerta OR de 4 entradas.

Nuestro módulo de ejemplo se puede instanciar en otros módulos de la siguiente manera:

```
puerta_or4 puerta1 (.in(cnt), .out(out4));
reg [3:0]cnt=0;
wire out4;
```

En este caso las señales de entrada, “in”, provienen de una señal de tipo registro de 4 bits, “cnt”, que comienza valiendo cero mientras que la salida se conecta a una señal de tipo “wire” llamada “out4”. Nótese que la sintaxis empleada no es la única posible, aunque es la más versátil ya que nos permite interconectar los terminales del módulo en cualquier orden.

Los módulos también pueden contener parámetros cuyo valor se puede indicar en el momento de instanciarlos. Veamos un ejemplo:

```
module puerta_orX(
    input [nbits-1:0]in,
    output out
);
parameter nbits=8;
assign out = (in!=0);
endmodule
```

En este caso el número de bits de la puerta OR es el indicado en el parámetro “nbits” y por defecto vale 8. Si quisiéramos instanciar este módulo para un número de bits distinto deberemos indicarlo de la siguiente forma:

```
puerta_orX #(.nbits(4)) puerta1 (.in(cnt), .out(out4));
```

Observemos que la lista de parámetros comienza con “#“ y va delante del nombre que damos a la instancia del módulo.

En líneas generales, no conviene abusar de los módulos. Es decir: no vamos a hacer un módulo de un subcircuito sencillo o de aquellos que nunca se van a instanciar más de una sola vez, pues esto al final se traduce en multitud de terminales a interconectar que hay que cambiar cada vez que se modifica un módulo. Y además complican la vida a las herramientas de síntesis (el procesador GUS16 inicialmente tenía 3 niveles de jerarquía: nivel 1: CPU, nivel 2: banco de registros, ALU, PC, Flags, registro de instrucción, operandos inmediatos, interrupciones. Nivel 3: registros simples. Tras “planarizar” el diseño y dejarlo en un único nivel de jerarquía el número total de celdas lógicas se redujo en unas 30)

Veamos ahora un banco de pruebas para simular nuestro módulo. Aquí lo que vamos a hacer es probar todas las combinaciones posibles de las entradas:

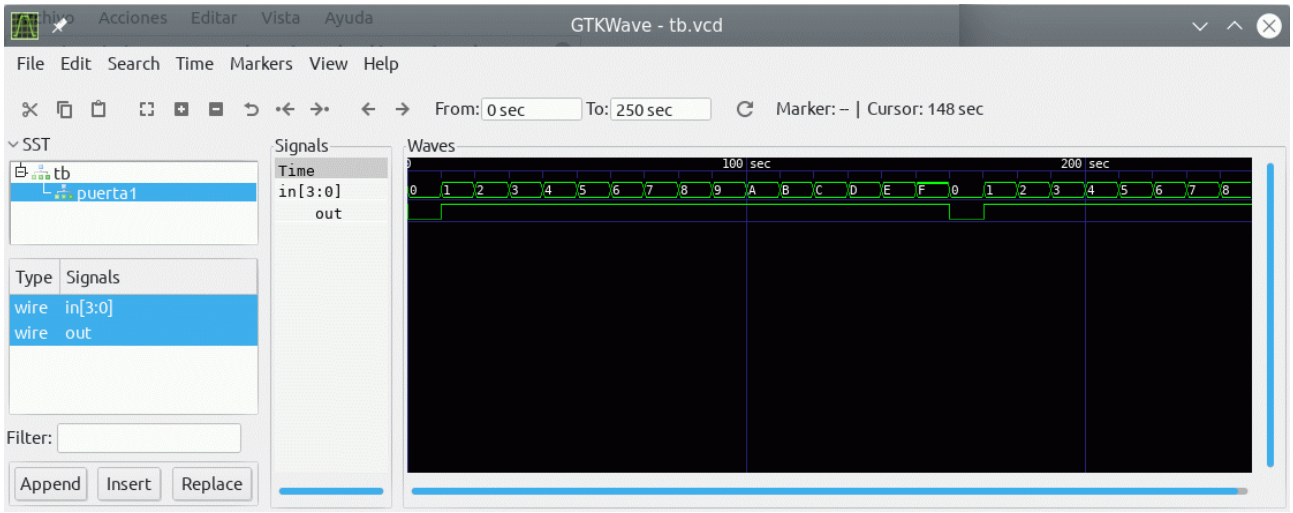
```
//-------------------------------------------------------------------
//-- Banco de pruebas
//-------------------------------------------------------------------
module tb();
reg [3:0]in=0;  //  señales de entrada
wire outsys;    //  señales de salida
// Instancia del módulo a probar
puerta_orX #(.nbits(4)) puerta1 (.in(in), .out(outsys));
// Generación de señales periódicas
always #10 in=in+1;  // nuevo estado cada 10 unidades de tiempo
// Proceso inicial
initial begin
$dumpfile("tb.vcd"); // Fichero donde almacenar los resultados
$dumpvars(0, tb);    // Grabar datos de todos los niveles de jerarquía
# 250 $finish;       // Terminamos 25 estados después del comienzo
end
endmodule
```

El módulo del banco de pruebas “tb” es el más alto de la jerarquía, y contiene las señales que son entradas al módulo bajo prueba como variables de tipo “reg”. Las señales de salida, “outsys”, por el contrario son de tipo “wire”, y no es estrictamente necesario incluirlas pues estamos grabando los datos de todas las señales del diseño y ya están disponibles en nuestro módulo. En este ejemplo la señal “in”, de 4 bits se incrementa cada 10 unidades de tiempo (en la simulación no se usan unidades absolutas para los tiempos) en la línea “always”. Por último, se incluye un bloque “initial” en el que se indica el nombre del archivo en el que se van a grabar los resultados, cuántos niveles de jerarquía se van a grabar (0=todos), y cuánto tiempo se va a simular.

El bloque initial también podría contener secuencias concretas para señales no periódicas antes del comando “$finish”. Este no es el caso del ejemplo, pero alternativamente se podría escribir algo como:

```
    in=0;
#10 in=1;
#10 in=2;
#10 in=3;
...
```

Donde se indica un retardo antes de asignar un nuevo valor a la señal. Esta secuencia iría dentro del bloque "initial" y se generaría una sola vez. En la siguiente figura se muestra el resultado de la simulación:



Para la síntesis lo que necesitamos es un módulo donde los nombres de las señales coincidan con los del archivo ".pcf" (Physical Constraints File), que a su vez se corresponden con pines concretos de la FPGA, por ejemplo:

```
set_io CLK 21
set_io IN[0] 144
set_io IN[1] 143
set_io IN[2] 142
set_io IN[3] 141
set_io LED[0] 78
set_io LED[1] 79
set_io LED[2] 80
```

El módulo más alto es ahora "main" y su contenido es básicamente el de un adaptador de conectores:

```
module main(
    input [3:0]IN,
    output [2:0]LED
);
puerta_orX #(.nbits(4)) puerta1 (.in(IN), .out(LED[0]));
endmodule
```

La síntesis del este módulo, junto con el de "puerta_orX" y del archivo ".pcf", nos genera un archivo "bitstream" para la configuración de la FPGA. El log de la herramienta "Place & Route" nos informa:

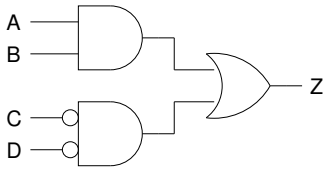
```
...
Info: Device utilisation:
Info:            ICESTORM_LC:     2/ 1280     0%
Info:           ICESTORM_RAM:     0/   16     0%
Info:                  SB_IO:     7/  112     6%
Info:                  SB_GB:     0/    8     0%
Info:           ICESTORM_PLL:     0/    1     0%
Info:            SB_WARMBOOT:     0/    1     0%
...
Info: Max delay <async> -> <async>: 3.78 ns
...
```

Donde vemos que nuestra puerta OR parametrizable utiliza 2 celdas lógicas (ICESTORM_LC), nada de RAM, y 7 pines (SB_IO), ninguno de ellos global (SB_GB, usado comunmente para señales de reloj). Además nos proporciona una estimación del retardo para un caso peor (3.78ns)

3 Circuitos combinacionales

En mi opinión, la forma más sencilla de describir un circuito combinacional es mediante el operador "=", bien en un comando "assign" o directamente en la declaración del nodo de salida ("wire"). La otra construcción sumamente útil para este cometido es el operador de asignación condicional, con una sintaxis similar a la empleada en el lenguaje C: "valor **=** condición **?** caso_1 **:** caso_0;"

3.1 Lógica simple




```
wire Z = (A&B)|((~C)&(~D));
```

En este ejemplo vemos la descripción de un circuito combinacional simple como una combinación de funciones AND, "&", OR, "|", e inversiones "~", donde además se hace uso de paréntesis para evitar ambigüedades acerca del orden de las operaciones.

Hay que notar que en el caso de que las señales sean vectores de más de un bit las operaciones AND, OR, y XOR, "^", y NOT, "~", se aplican en todos los bits de los vectores:

```
wire [2:0]A; wire [2:0]B; wire [2:0]Z;
Z=A&B; // equivale a: Z[0]=A[0]&b[0]; Z[1]=A[1]&b[1]; Z[2]=A[2]&b[2];
```

Pero si el operador AND, OR, o XOR se aplica delante de un vector se entiende que es la función lógica de todos los bits del vector. Veamos como ejemplo un generador de paridad:

```
wire [7:0]d;
wire paridad;
assign paridad = d[7]^d[6]^d[5]^d[4]^d[3]^d[2]^d[1]^d[0];
// o más compacto:
assign paridad = ^d;
```

El operador de asignación condicional también puede ser de utilidad para realizar una función en todos los bits de un vector:

```
wire [7:0]d;
wire enable;
wire [7:0]out = enable ? d : 0; // AND de enable y d[i] para cada bit i
```

3.2 Multiplexores

Uno de los bloques combinacionales más frecuentes es el multiplexor (especialmente en el diseño para FPGAs donde no tenemos triestados) y no disponemos de una primitiva de Verilog para implementarlo así que hay que ser creativos. Si sólo tiene dos entradas, de uno o varios bits, es muy fácil describirlo mediante una asignación condicional:

```
assign mux_out = seleccion ? entrada_1 : entrada_0;
```

Pero si es de más entradas (4, 8, ...) la cosa se complica. Se puede recurrir a estructuras "case", y eso precisamente era lo que inicialmente hacía hasta que encontré otra forma más simple de describirlos: Basta con crear un vector con las entradas para luego seleccionar la que corresponda:

```
wire [7:0]mux_in = {i7,s6,s5,s4,s3,entrada2,entrada1,input0};
wire mux_out = mux_in[seleccion];
```

Observemos el operador de concatenación, "{...}", que permite agrupar señales para formar un vector con un número de bits que es la suma de todos los bits de las señales concatenadas. Para terminar de complicar las cosas supongamos ahora tenemos un multiplexor de 4 entradas, cada una de 8 bits:

```
wire [7:0]mux_in[0:3];
assign mux_in[0]=entrada0;
assign mux_in[1]=entrada1;
assign mux_in[2]=entrada2;
assign mux_in[3]=entrada3;
wire [7:0]mux_out=mux_in[seleccion];
```

En este caso estamos haciendo un array de 4 vectores de 8 bits con las señales de entrada para luego seleccionar el vector deseado.

3.3 Buses de entrada

Otro caso frecuente es el de un multiplexor con unas señales de selección decodificadas. Esto es: una señal de selección por cada entrada, con solo una de ellas activa. En este caso el número de entradas no tiene que ser necesariamente una potencia de 2. Una forma práctica de describir este circuito es mediante una estructura AND-OR, como en el siguiente ejemplo en el que tenemos 3 datos de entrada:

```
wire [7:0]busin =
    (sel0 ? entrada0 : 0) |
    (sel1 ? entrada1 : 0) |
    (sel2 ? entrada2 : 0);
```

Observemos además que si ninguna señal de selección está activa en "busin" vamos a tener un valor de 0. Si deseásemos una salida con todos los bits en 1 por defecto usaríamos un estructura OR-AND:

```
wire [7:0]busin =
    (sel0 ? entrada0 : 8'hff) &
    (sel1 ? entrada1 : 8'hff) &
    (sel2 ? entrada2 : 8'hff);
```

3.4 Codificador de prioridad

Como ejemplo de "abuso" del operador de asignación condicional veamos ahora un codificador de prioridad de 8 entradas:

```
wire [7:0]I;
wire valid = |I;
wire [2:0]code = I[7] ? 3'b111 : (
                 I[6] ? 3'b110 : (
                 I[5] ? 3'b101 : (
                 I[4] ? 3'b100 : (
                 I[3] ? 3'b011 : (
                 I[2] ? 3'b010 : (
                 I[1] ? 3'b001 : 3'b000 )))))));
```

Aquí la entrada I[7] es la más prioritaria. La señal "valid" es la OR lógica de todas las entradas y sirve para distinguir entre tener la entrada I[0] activa o inactiva, pues en ambos casos el código generado es 0.

3.5 Aritmética: Sumadores, Restadores...

La operación aritmética básica en los sistemas digitales es la suma, que en el lenguaje Verilog se indica mediante el operador "+". Es conveniente dejar que las herramientas de síntesis se encarguen de generar la lógica correspondiente a los sumadores en lugar de descrbir nosotros cómo se realiza la suma con todo detalle. Esto se debe a que las FPGAs disponen de lógica rápida específica para los acarreos que se va a usar cuando se sintetiza un sumador, pero si lo describimos con demasiado detalle la herramienta de síntesis puede pasar por alto la existencia de la cadena de acarreo y el resultado es un circuito más lento. Por ello un sumador se describe simplemente como:

```
wire [7:0]A;
wire [7:0]B;
wire [7:0]suma;
wire acarreo;
assign {acarreo,suma} = A + B;
```

Fijémonos que el resultado de la suma tiene un bit más que el operando más grande, que aquí hemos llamado "acarreo" y que hemos dejado en una señal aparte.

La resta es similar a la suma:

```
assign {acarreo,resta}= A - B;
```

Tan sólo hemos de tener en cuenta que "acarreo" va a valer "1" cuando **no hay llevada** al restar. Alternativamente podemos realizar la resta sumando el complemento a dos de "B":

```
assign {acarreo,resta}= A + (~B) + 1;
```

En Verilog también tenemos el operador multiplicación "*", pero hemos de ser cautos con su uso pues un multiplicador combinacional es un circuito muy complejo que puede agotar rápidamente los recursos de hardware. A título de ejemplo veamos un multiplicador para dos datos sin signo de 8 bits:

```
wire [7:0]A;
wire [7:0]B;
wire [15:0]producto = A * B;
```

Este multiplicador ocupó 161 celdas lógicas en una FPGA ICE40HX, lo que es asumible, pero la complejidad de estos circuitos crece con el cuadrado del número de bits de modo que un multiplicador para datos de 32 bits ocuparía 16 veces más, y sería unas 4 veces más lento.

3.6 Tablas de verdad, case, casex

Un circuito combinacional cualquiera se podría describir mediante una tabla de verdad. En Verilog esto se puede hacer mediante la construcción "case". Veamos un ejemplo:

```
wire A,B,C;
reg [3:0]out;
always @*
  case ({C,B,A})
    3'b000 : out = 4'h0;
    3'b001 : out = 4'h2;
    3'b010 : out = 4'h4;
    3'b100 : out = 4'h9;
    default: out = 4'hX;
  endcase
```

En este ejemplo hay que señalar varios detalles: En primer lugar que la construcción "case" va asociada al comando "always" y en consecuencia la señal de salida, "out" tiene que ser de tipo "reg". Esto nos puede hacer pensar que estamos construyendo un circuito secuencial, que no es el caso ya que aquí no tenemos ninguna señal de reloj. ¿Había dicho que el lenguaje Verilog podía ser confuso? Aquí tenemos un ejemplo.

El comando "always @*" indica al interprete de Verilog que se reevalue la salida cada vez que haya un cambio en cualquiera de las variables de entrada, A, B, o C. Esto está más relacionado con el funcionamiento interno del simulador de Verilog que con el circuito que se está describiendo.

La combinación "default" sirve para rellenar la tabla con un valor concreto para las combinaciones de entrada no especificadas. En este caso se ha asignado un valor "No Importa", o "X", a los 4 bits de salida, que a su vez están indicados en base hexadecimal. Los casos "No Importa" pueden ayudar a simplificar la lógica durante la síntesis.

Una variante muy interesante de esta construcción es "casex", que nos permite descartar algunos bits de las entradas en la descripción de la salida. Es por lo tanto muy adecuada para la descripción de PLAs. Veamos de nuevo el ejemplo del codificador de prioridad:

```
wire [7:0]I;
reg valido;
reg [2:0]code;
always @*
  casex (I)
    8'b0000_0000 : {valido, code} = {1'b0, 3'dX};
    8'b0000_0001 : {valido, code} = {1'b1, 3'd0};
    8'b0000_001x : {valido, code} = {1'b1, 3'd1};
    8'b0000_01xx : {valido, code} = {1'b1, 3'd2};
    8'b0000_1xxx : {valido, code} = {1'b1, 3'd3};
    8'b0001_xxxx : {valido, code} = {1'b1, 3'd4};
    8'b001x_xxxx : {valido, code} = {1'b1, 3'd5};
    8'b01xx_xxxx : {valido, code} = {1'b1, 3'd6};
    8'b1xxx_xxxx : {valido, code} = {1'b1, 3'd7};
endcase
```

Algún purista de Verilog podría poner el grito en el Cielo porque "casex" jamás debería usarse en un código que se vaya a sintetizar y habría que sustituirlo por "casez" al igual que las "x" por "?" en el lado de la izquierda. Pero "yosys" no tiene ningún problema en sintetizar "casex" y Verilog ya es bastante confuso con dos versiones de "case" como para encima tener hasta tres (La diferencia entre "casex" y "casez" es tan sutil que casi nadie sabe en qué consiste realmente). Y personalmente prefiero las "x" a las "?".

Lo cierto es que este tipo de construcciones, especialmente "case", se asemejan a una memoria ROM y podrían implementarse como tal. Pero las memorias las veremos de forma explícita más adelante.

4 Circuitos Secuenciales

Me refiero a circuitos secuenciales **Síncronos** en los que el estado del circuito cambia en un determinado flanco de una señal de reloj. Estos circuitos emplean Flip-flops como su principal elemento constructivo. Por otra parte, los circuitos secuenciales asíncronos son bloques combinacionales con realimentación. En las herramientas de síntesis se denominan como "Combinational Loops" y se consideran un error de diseño, hasta tal punto que si se quisieran sintetizar explícitamente es necesario desactivar estos errores con la opción "--ignore-loops" en la aplicación de "Place & Route".

El principal comando Verilog para la descripción de estos bloques es "always @(posedge clock)" o alguna variante similar.

4.1 Registros

Posiblemente el bloque secuencial más simple es un registro que se escribe en los flancos de subida de una señal de reloj, "clk", si su entrada de habilitación de escritura, "we", está activa:

```
wire [7:0]D;
wire clk;
wire we;
reg [7:0]Q = 0;
always @(posedge clk) if (we) Q <= D;
```

Aquí hay que señalar un par de detalles: Primero, el haber asignado un valor inicial al registro "Q" (se podría haber hecho en un bloque "initial", pero es más engorroso). Esto es muy conveniente para evitar valores

desconocidos ("X") en las simulaciones, y en las implementaciones en FPGA no supone coste alguno pues todos los Flip-Flops comienzan tras el "reset" de la FPGA en un estado conocido. Y segundo, el uso del operador de asignación simultánea "<=". En este ejemplo se podría haber usado una asignación normal, "=", ya que dentro de "always" hay una única asignación, pero si el bloque "always" fuese más complejo es necesario usar la asignación simultánea pues todas las señales deben cambiar al mismo tiempo cuanto se tiene el flanco seleccionado en el reloj.

Y por supuesto, la versión "always @(negedge clk)" es también posible.

Una versión un poco más sofisticada incluye una señal de "reset" asíncrono:

```
wire [7:0]D;
wire clk;
wire we;
wire reset,
reg [7:0]Q = 0;
always @(posedge clk or posedge reset)
    if (reset) Q<=0; else if (we) Q<=D;
```

La señal "reset" es una entrada asíncrona pues cambia la salida del registro en el mismo instante en el que se activa, sin esperar al flanco del reloj. Veamos ahora una variante con reset síncrono:

```
wire [7:0]D;
wire clk;
wire we;
wire reset;
reg [7:0]Q = 0;
always @(posedge clk) if (reset) Q<=0; else if (we) Q<=D;
```

Aquí el valor de "reset" sólo es relevante en el instante del flanco del reloj.

4.2 Contadores

4.2.1 Contador binario síncrono

Este es posiblemente el contador más fácil de implementar en lenguaje Verilog:

```
reg [3:0]Q = 0;
always @(posedge clk) Q <= Q + 1;
```

Como vemos en el ejemplo, tan sólo consta de un registro y un sumador que incrementa en uno su contenido en cada flanco del reloj. Por supuesto, este contador básico se puede modificar para añadirle otras funciones. Veamos como ejemplo un contador con una entrada de habilitación de cuenta, "en", una entrada de carga en paralelo síncrona, "pl", y una salida de final de cuenta, "tc":

```
wire [7:0]D;
wire en;
wire pl;
reg [7:0]Q = 0;
always @(posedge clk) Q <= pl ? D : (Q + en);
wire tc = &Q;
```

4.2.2 Contador síncrono de módulo arbitrario

Ahora vamos a describir un contador cuyo módulo (número de ciclos hasta repetir estado) no tenga que ser necesariamente una potencia de 2. La idea básica es generar una señal de final de cuenta, “tc”, cuando se llegue al valor del módulo menos uno, y cuando esta señal esté activa el contador se reiniciará con cero en el siguiente ciclo. Y además vamos a hacer que el tamaño del contador dependa del módulo:

```
parameter MODULO = 125;
localparam NBITS = $clog2(MODULO-1);
reg [NBITS-1:0]Q=0;
wire tc = (Q == (MODULO-1));
always @(posedge clk) Q <= tc ? 0 : Q+1;
```

Como podemos observar, estamos generando un parámetro local, “NBITS” en el que usamos la función “$clog2” (logaritmo en base dos redondeado hacia arriba) para calcular cuántos bits vamos a necesitar en el contador. La señal “tc” valdrá 1 cuando la cuenta, “Q” sea igual a “MODULO -1”, y ello hará que el contador se reinicie con cero de forma síncrona (en el siguiente flanco del reloj). Destaquemos el operador de comparación, “==”, que no hay que confundir con el de asignación, “=”.

Lo cierto es que la lógica de “tc” se podría simplificar aún más pues en su tabla de verdad todas las combinaciones de “Q” mayores que “MODULO-1” son casos “No Importa”. Sin embargo no he encontrado una forma simple de describir esta función lógica de forma genérica. Aunque para valores particulares de “MODULO” sí que se puede simplificar manualmente. Supongamos un caso concreto: Queremos un módulo de 100. Eso implica que hemos de contar hasta 99, que en binario es 7’b1100011. “tc” será la AND de los bits de Q que sean 1 en el valor anterior, eso es: Q[0], Q[1], Q[5], y Q[6]:

```
wire tc = Q[0] & Q[1] & Q[5] & Q[6];
```

En este ejemplo el cambio de la función “tc” reduce el número de celdas lógicas de 12 a 11. La mejora es escasa y el esfuerzo no parece merecer la pena, más aún si consideramos que necesitamos una ecuación distinta para cada valor de “MODULO”.

4.3 Contador de rizado.

Este no es un circuito muy habitual y su uso casi se limitaría a prescalers que dividen una frecuencia de reloj por una potencia de 2. En este contador tenemos unicamente Flip-Flops, cada uno con una señal de reloj que proviente del bit anterior. Como ejemplo veamos un contador de rizado de 4 bits:

```
reg [3:0]Q = 0;
always @(negedge clk ) Q[0] <= ~Q[0];
always @(negedge Q[0]) Q[1] <= ~Q[1];
always @(negedge Q[1]) Q[2] <= ~Q[2];
always @(negedge Q[2]) Q[3] <= ~Q[3];
```

Observemos que en este contador los Flip-Flops cambian a su valor contrario en los flancos de bajada de sus respectivas señales de reloj. Por otra parte este es un circuito en el que se podría usar al comando “generate” para hacer un contador con un número de bits parametrizable:

```
// Contador de rizado N-bits
parameter NBIT=14;
reg [NBIT-1:0]Q=0;
generate
    genvar i;
    for (i=0; i<NBIT; i=i+1) begin
      always @(negedge (i==0)? clk : Q[i-1]) Q[i]<=~Q[i];
    end
endgenerate
```

Donde vemos que "generate" es en realidad una macro que al expandirse nos generará un código Verilog similar al anterior.

4.4 Registros de desplazamiento

En estos registros es el operador de concatenación, "{...}", el truco a utilizar. Veamos un registro de 8 bits que desplaza sus datos hacia la izquierda:

```
reg [7:0]shreg = 0;
wire serial_in;
always @(posedge clk) shreg <= {shreg[6:0],serial_in};
```

La versión que desplaza los datos a la derecha sería:

```
always @(posedge clk) shreg <= {serial_in,shreg[7:1]};
```

Por supuesto, estos circuitos básicos se pueden hacer más sofisticados añadiendo señales de control adicionales como habilitación de desplazamiento, carga en paralelo, reset, etc.

4.4.1 Contador de Johnson

Como un caso particular de uso de los registros de desplazamiento tenemos los contadores de Johnson, donde la entrada serie es el último bit del registro complementado. Veamos uno de estos contadores con 5 Flip-Flops y módulo 10:

```
reg [4:0]jc = 0;
always @(posedge clk) jc <= {jc[3:0],~jc[4]};
```

4.4.2 Generador de secuencia pseudoaleatoria (PRBS)

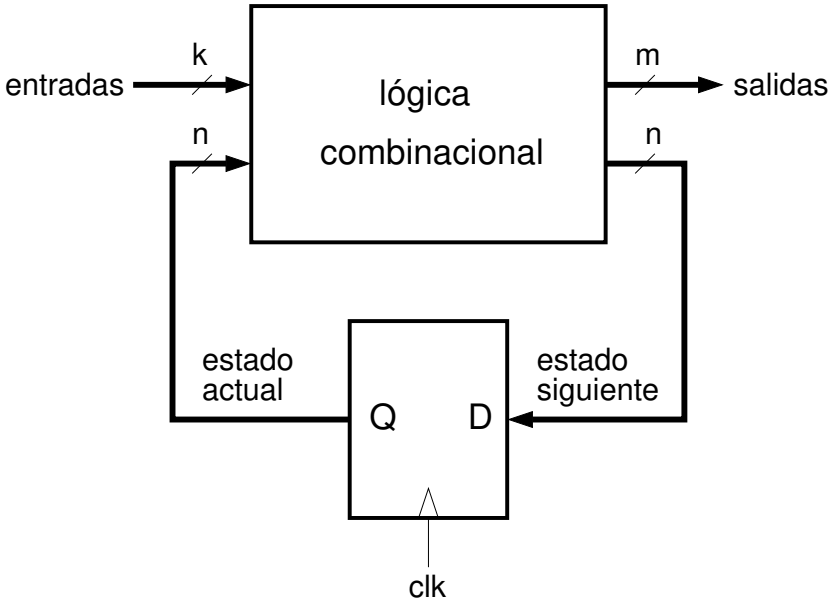
Otro caso particular son los generadores de señales pseudoaleatorias, "PRBS", en los que la entrada serie es la paridad de un determinado conjunto de bits del registro. Veamos un ejemplo con un registro de 15 bits:

```
reg [14:0]prbs = 0;
always @(posedge clk) prbs <= {prbs[13:0], ~(prbs[14]^prbs[13])};
```

Hay que destacar que este circuito pasa por 32767 estados distintos, pero no pasa por el valor que tiene todos los bits en 1, de modo que el módulo es $2^n - 1$ que es el máximo posible. Para conseguir secuencias pseudoaleatorias con un módulo máximo hay que elegir adecuadamente qué bits del registro se incluyen en la generación de paridad. Para una explicación en mayor profundidad búsquese bibiografía acerca de:

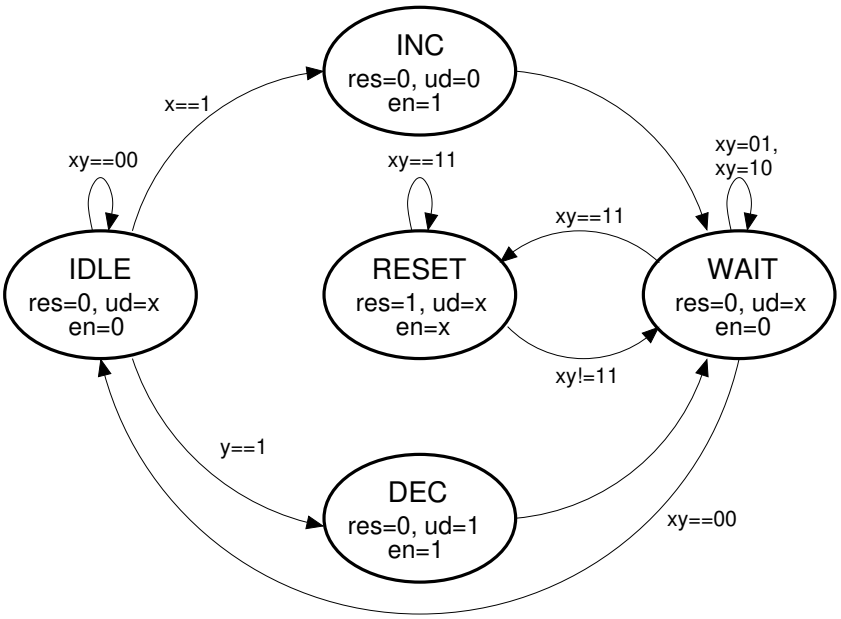
- Pseudo Random Binary Sequence (PRBS)
- Linear Feedback Shift Register (LFSR)
- Cyclic Redundance Check (CRC)
- Évariste Galois. La mente que estuvo detrás de estos temas y que acabó parando una bala en un duelo estúpido.

4.5 Máquinas de estados



Una máquina de estados, o autómata, es un circuito digital que encaja en el diagrama de bloques de la figura anterior, donde vemos que consta de un registro de n bits que almacena el estado actual y de un bloque combinacional que genera las señales para el estado siguiente y las salidas a partir del estado actual y las entradas. Si las salidas dependen únicamente del estado actual se dice que la máquina de estados es un "Autómata de Moore", mientras que el caso más general en el que las salidas dependen también de las entradas sería un "Autómata de Mealy".

No es difícil constatar que los contadores y los registros de desplazamiento son casos particulares de máquinas de estado, pero lo que me propongo ilustrar aquí es una forma genérica de describir cualquier tipo de máquina de estados. Para ello me valdré del siguiente ejemplo en el que comenzamos mostrando su diagrama de estados:



Como vemos se trata de una máquina de Moore con 5 estados y por lo tanto necesitaremos 3 bits para el registro de estado. Dejando de momento pendiente la generación de las señales de salida, podemos describir así la lógica del estado siguiente:

```
parameter IDLE = 0;
parameter INC  = 1;
parameter DEC  = 2;
parameter RESET= 3;
parameter WAIT = 4;
reg [2:0]estado = IDLE;
always @(posedge clk)
    case (estado)
        IDLE:    estado <= x ? INC : (y ? DEC : IDLE);
        INC:     estado <= WAIT;
        DEC:     estado <= WAIT;
        WAIT:    estado <= (x&y)? RESET : ( (x|y)? WAIT : IDLE);
        RESET:   estado <= (x&y)? RESET : WAIT;
        default: estado <= 3'bxxx;
endcase
```

Y la generación de las salidas:

```
reg res,ud,en; // No son registros
always @*
case (estado)
        IDLE:    {res,ud,en} <= 3'b0x0;
        INC:     {res,ud,en} <= 3'b001;
        DEC:     {res,ud,en} <= 3'b011;
        WAIT:    {res,ud,en} <= 3'b0x0;
        RESET:   {res,ud,en} <= 3'b1xx;
        default: {res,ud,en} <= 3'bxxx;
endcase
```

5 Memorias

5.1 Síncronas o asíncronas

Este apartado tiene su razón de ser a causa de los bloques de memoria RAM de las FPGAs. Estas memorias, a diferencia de las memorias RAM y ROM habituales, tienen una lectura síncrona. Esto significa que debemos poner un dato válido en las entradas de dirección de la memoria antes de aplicar un flanco en la señal de reloj, y el dato leído está disponible en las salidas después del flanco. Por el contrario, las memorias normales no tienen una señal de reloj y un cambio en las entradas de dirección se traduce inmediatamente en un cambio en el dato de salida. Por ello estas memorias se denominan asíncronas.

Una memoria **ROM asíncrona** es un circuito puramente combinacional y se podría describir por ejemplo mediante una tabla "case". Pero aquí vamos a describirlo como una memoria:

```
wire [7:0]addr;
wire [8:0]q;

reg [8:0]memoria[0:255];
assign q=memoria[addr];

initial begin
  $readmemh("seno.hex", memoria);
end
```

En este ejemplo hay que destacar el uso de una tabla de datos, "memoria", que no es realmente un registro (no hay reloj), y de un bloque initial que da unos valores iniciales a la tabla a partir de un archivo con 256 datos hexadecimales de 9 bits, "seno.hex" (muestras de 1/4 de ciclo de sinusoide), y cuyas primeras líneas son:

```
001
004
007
00a
00e
011
...
```

La prueba de que en realidad este es un circuito combinacional es que tras una síntesis para FPGA se ocupan 162 celdas lógicas pero ningún bloque de memoria.

Una memoria **RAM asíncrona** sí que es ya un circuito secuencial en el que las escrituras tienen lugar en el flanco activo de su señal "we" (write enable):

```
wire [7:0]addr;
wire [7:0]q;
wire [7:0]d;
wire we;

reg [7:0]memoria[0:255];
assign q=memoria[addr];
always @(posedge we) memoria[addr] <=d;
```

En este caso la síntesis ha ocupado nada menos que 4066 celdas lógicas, lo que es aproximadamente dos celdas lógicas por cada bit de la memoria RAM. No es por lo tanto una buena idea sintetizar este tipo de memorias dentro de una FPGA, aunque podemos dar uso a esta construcción para simular memorias externas. También se podría añadir un bloque "initial" para dar un valor inicial al contenido de esta memoria aunque luego, a diferencia de la ROM, este contenido puede cambiar.

5.2 RAM (y ROM) síncrona

Este es el tipo de memoria que tenemos disponible dentro de la FPGA. En particular las FPGA de la familia ICE40HX tienen esta memoria dividida en un número variable de bloques, todos ellos de un tamaño de 256 datos de 16 bits, aunque se pueden reconfigurar como 512×8, o 1024×4. El número de bloques depende del modelo concreto de la FPGA, pero típicamente hablamos de 16 o 32 bloques como máximo en esta familia.

Un aspecto destacable de estas BRAMs es que disponen de líneas de dirección separadas para la lectura y la escritura, así como también relojes independientes, lo que las hace muy adecuadas para su uso como buffers. Veamos un ejemplo en el que describimos una de estas memorias:

```
wire clkr;        // reloj de lectura
wire clkw;        // reloj de escritura
wire [8:0]addr;   // dirección de lectura
wire [8:0]addw;   // dirección de escritura
wire we;          // Habilitación de escritura
wire [7:0]d;      // datos a escribir
reg  [7:0]q;      // datos leídos

reg [7:0]memoria[0:511];
always @(posedge clkw) if (we) memoria[addw] <=d;
always @(posedge clkr) q <= memoria[addr];

initial begin
  $readmemh("RAMinit.hex", memoria);
end
```

Por descontado, los dos relojes podrían ser el mismo al igual que las entradas de dirección, y tendríamos una memoria síncrona de un único puerto, que seguramente va a ser más habitual.

Las herramientas de síntesis son capaces de identificar la presencia de la RAM síncrona y la asignan a los bloques BRAM de la FPGA:

```
Info: Device utilisation:
Info:            ICESTORM_LC:     2/ 1280     0%
Info:           ICESTORM_RAM:     1/   16     6%
```

Hay que destacar que esta memoria ocupa un único bloque BRAM, y que memorias más pequeñas van a seguir ocupando un mínimo de un bloque. Por otra parte, si el ancho de los datos es de más de 16 bits se van a emplear como mínimo dos bloques BRAM.

Por último, también podríamos eliminar las señales de escritura, "clkw" y "we", y obtendríamos una memoria ROM síncrona que va a ocupar un bloque BRAM y cuyos datos se especifican en el bloque initial.