Arquitectura de Computadores II: Práctica 2

16 de septiembre de 2024

1. Acumulador de 4 bits y creación de test bench

Los objetivos de esta sesión son por un lado consolidar los conocimientos adquiridos en la sesión anterior, a la vez que se presentan nuevos aspectos del lenguaje System Verilog. Veremos cómo construir diseños jerárquicos, la especificación de buses y registros. Estas funcionalidades son extremadamente útiles en la construcción de circuitos complejos, que usualmente se describen de forma estructural. Además, se muestra cómo parametrizar diseños y cómo instanciar e interconectar réplicas de un elemento mediante un constructor del lenguaje. Por otro lado, se introduce la utilización de programas de prueba, escritos en VHDL, para comprobar o verificar si el funcionamiento de un diseño se adecua a las especificaciones.

La principal tarea del laboratorio consistirá en construir y simular un **sumador de 4 bits con propagación serie del acarreo**, utilizando como módulo básico el **sumador de 1 bit (full adder)** diseñado en la sesión anterior. Extenderemos este sumador con un registro para convertirlo en un acumulador, una estructura que permite calcular series de sumas.

En un apéndice se describe la especificación en System Verilog, utilizando constructores condicionales de asignación de señal, de componentes básicos en el diseño de circuitos combinacionales como son el multiplexor, el decodificador y una puerta de tres estados.

2. Fundamentos de los Hardware Description Languages (HDL) 2

En esta sección extenderemos los conceptos básicos ya explicadoes en la práctica anterior. De nuevo explicaremos conceptos aplicables a cualquier lenguaje HDL (VHDL, Verilog, SystemVerilog, etc.). *Pero ahondaremos en conceptos específicos de* ***SystemVerilog*** *y sus ejemplos destacados en itálico.*

2.1. Logica Secuencial

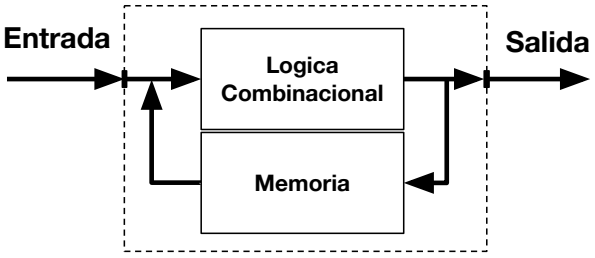


Figura 1: Diagrama conceptual de un circuito secuencial.

Como ya comentamos en la anterior práctica existen dos tipos de circuitos: Combinacional y Secuencial. Ya vimos como especificar circuitos combinacionales utilizando asignaciones concurrentes o bloques combinacionales. Acontinuación abordamos la descripción de circuitos secuenciales, que permiten el almacenamiento de información. Recordemos que en los circuitos secuenciales los valores de las salidas, en un momento dado, no dependen exclusivamente de los valores de las entradas en dicho momento, sino también dependen del

estado interno. Estos circuitos secuenciales actualizan su estado interno en función de las señales de entrada y/o del estado actual, a través de una señal de permiso que marca cuándo debe realizarse la transición entre estados.

Para simplificar la complejidad de los circuitos electrónicos, siempre diseñaremos circuitos síncronos, en los que todos los circuitos secuenciales actualizan su estado en el mismo instante. Para sincronizar todos los circuitos usaremos una señal de reloj, que se caracteriza por ser una señal repetitiva de forma permanente y que tiene un intervalo de tiempo de repetición característico. A este intervalo de tiempo se le denomina periodo o tiempo de ciclo. En un periodo de reloj se distinguen dos subintervalos de tiempo consecutivos. En uno de ellos el nivel lógico es el 0 y en el otro subintervalo de tiempo el nivel lógico es 1. Un cambio de nivel lógico en la señal de reloj se denomina flanco. El flanco se denomina ascendente cuando se pasa de nivel 0 a nivel 1 y descendente en caso contrario.

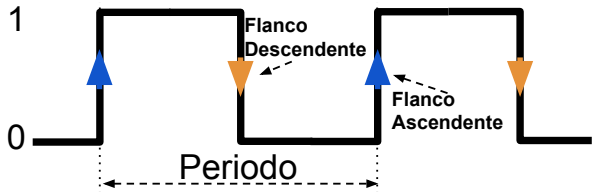


Figura 2: Señal de reloj.

La decisión del instante en el cual se actualiza el estado de los elementos de almacenamiento se denomina metodología de reloj. Es importante distinguir los instantes en que se lee un registro de los instantes en que se escribe. Si las dos acciones se efectúan de forma concurrente, el valor de la lectura puede ser el valor antiguo, el que se está escribiendo o una mezcla de ambos. La metodología de reloj que utilizaremos es por flanco ascentente, es decir el cambio de nivel lógico 0 a nivel lógico 1. Por lo tanto, actualizaremos los elementos de memorización en el flanco ascendente.

A continuación vamos a estudiar un circuito secuencial sencillo y cómo podemos especificarlo en System Verilog. Existen muchos circuitos básicos que podemos estudiar. Para no extendernos nos centraremos en el registro. Un registro es un elemento básico de almacenamiento que actualiza su estado en el flanco ascendente de la señal de reloj. Un registro elemental almacena un bit. Entonces, para construir un registro de n bits se agrupan n registros elementales y se accede a ellos como una unidad.

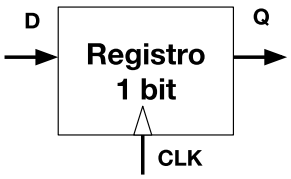


Figura 3: Registro de 1 bit.

En la figura 3 la etiqueta D se corresponde con la entrada de datos del registro y la etiqueta Q se corresponde con la salida del registro. La señal reloj (CLK de clock) se utiliza para sincronizar el instante de actualización del registro. Para realizar la lectura del registro se utiliza la salida del registro (Q), que puede utilizarse en cualquier instante de tiempo como entrada de un circuito combinacional y esta acción no modifica el estado del registro. Para realizar la escritura del registro se utiliza la señal de entrada (D), en el flanco ascendente del reloj actualizaremos el estado interno del registro.

Como ya vimos con los circuitos combinacionales, los componentes electronicos no son perfectos. Y por ello, tenemos que tener en cuenta las diferentes características del circuito secuencial. Existen tres parámetros importantes, dos de ellos son relativos a que la señal de entrada debe ser estable un intervalo de tiempo antes y después del flanco de la señal reloj (Ver Figura 4). Estos intervalos de tiempo se denominan respectivamente tiempo de estabilidad (t_e , 'set-up time') y tiempo de mantenimiento (t_m , 'hold time'). El tercer parametro es el retardo de propagación (t_p, 'delay'), que mide el retardo con que se observa la actualización del registro en la salida.

Como en los circuitos combinacionales, los programadores de RTL o HDL, no tienen en cuenta estos valores en sus diseños iniciales. En una etapa posterior donde ya se ha realizado la síntesis del circuito, se ajustan estos parametros para cumplir las especificaciones. Sin embargo para nuestras realizar estas prácticas emplearemos el tiempo de propagación para caclular el retardo de un circuito, como hicimos en el laboratorio 1.

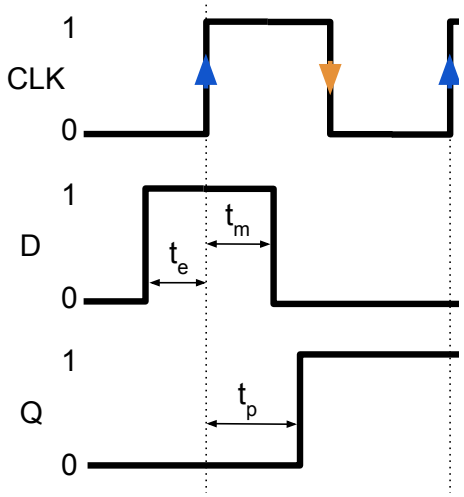


Figura 4: Escritura en un registro. El retardo de propagación es t_p. Los tiempos de estabilidad y mantenimiento son t_e y t_m respectivamente.

El lenguaje Verilog tiene un objeto específico para definir un elemento de memorización ('reg'). Sin embargo, cómo ya hemos comentado para SystemVerilog la utilización del tipo de datos 'reg' está desaconsejada. En su lugar utilizaremos siempre 'logic'. La semántica del lenguaje permite que las señales logic sean interpretadas como un elemento combinacional (cable) o de memorización (registro). En otras palabras, el elemento de memorización es inferido dependiendo de cómo estas señales son asignadas. La clave es que la semántica del lenguaje SystemVerilog estipula que, en casos donde el código no especifica un valor de una señal, la señal mantiene su valor actual. En otras palabras, la señal debe recordar su valor actual y para hacerlo implícitamente se necesita un elemento de memorización. Por ello, de forma no consciente se pueden crear elementos de memorización (celda, "latch"). Una posibilidad para que no ocurra es asignar a todas las señales valores por defecto en el inicio de cada bloque.

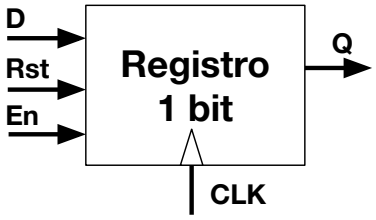
En SystemVerilog existe una única forma de especificar circuitos secuenciales, mediante los bloques secuenciales. Como los bloques combinacionales, un bloque secuencial se define con la secuencia ***always_ff*** *(ff viene de Flip-Flop) seguido de '@' y entre parentesis las señales que activan la evaluación del bloque secuencial. En esta lista de señales de sensibilidad deberemos incluir todas las señales que modifiquen el estado interno del circuito secuencial como el reloj o una señal de reset. Cada una de estas señales de sensibilidad puede ir precedida por la especificación que restringe su activación, como* ***posedge*** *(positive edge) que indica el flanco ascendente o* ***negedge*** *(negative edge) que indica el flanco descendente.*

El bloque secuencial debe estar delimitando su especificación entre las sentencias ***begin*** *y* ***end****. Dentro de un bloque combinacional realizaremos asignaciones bloqueantes utilizando el operador '<='. A continuación se muestra un ejemplo de cómo especificar un registro sencillo de un bit.*

```
/*
 * Especificación de un registro de 1 bit
 */

// Señal de reloj
logic clk_i;

// Señales de entrada y salida
```

```
logic intput_d;
logic output_q;

// Bloque secuencial
always_ff @(posedge clk_i) begin
    output_q <= input_d;
end
```

El registro sencillo sólo nos permite guardar la entrada del ciclo anterior de 1 bit de tamaño. Una buena práctica especificando circuitos secuenciales es la de definir valores de reset, de hecho muchas herramientas de sintesis lanzan mensajes de aviso cuando se escribe un circuito secuencial sin señal de reset. Esta señal de reset nos permite especificar cual es el estado del circuito secuencial al iniciar o reiniciar el circuito. Además de la señal de reset, es muy frecuente que los registros tengan señal de permiso de escritura que permite bloquear el cambio de estado interno a voluntad del diseñador. En la siguiente figura se muestra el diagrama de un registro con función de reset y permiso de escritura, junto con su correspondiente tabla de verdad.



Rst	En	D	Q
0	0	0	Q
0	0	1	Q
0	1	0	0
0	1	1	1
1	x	x	0

Figura 5: Diagrama de un registro con función de reset a 0 y función de escritura; y su correspondiente tabla de verdad.

En las siguientes secciones, veremos como extender este diseño para poder incluir estas funcionalidades. Y en las siguientes prácticas veremos cómo construir una máquina de estados.

2.2. Sentencias Condicionales

Al igual que en los paradigmas de programación tradicionales, al especificar un ciruito en HDL existen casos en los que es necesario poder expresar caminos logicos divergentes dependiendo de condiciones booleanas construidas con señales. Por ello los lenguajes HDL incluyen sentencias condicionales que nos permiten definir bloques de operaciones que activarán dependiendo de una expresión booleana. Estas sentencias pueden expresarse tanto en circuitos secuenciales cómo en circuitos combinacionales. Así mismo, las sentencias condicionales se pueden anidar de forma que podemos crear un árbol de circuitos que se activarán de forma condicional.

*En System Verilog podemos especificar un bloque condicional utilizando la sentencia '**if**' seguido de una condición booleana especificada entre paréntesis. Después deberemos especificar el circuito que será activado de forma condicional delimitado entre **begin** y **end**. Adicionalmente, podemos especificar un circuito alternativo que será activado en caso de no cumplirse la expresión booleana utilizando la sentencia '**else**' seguida del circuito delimitado entre **begin** y **end**. De esta forma si el resultado de la expresión entre paréntesis es verdadera el circuito especificado en el bloque 'if' será activado. Si el resultado es falso, se activará el circuito especificado en el bloque 'else', si existiera. En el siguiente apartado veremos un ejemplo práctico.*

2.3. Parametrización

Una de las claves del diseño de circuitos hardware es la reutilización de componentes básicos, de forma que los diseñadores puedan reducir la carga de trabajo que conlleva especificar circuitos complejos. Sin embargo, la especificación en lenguaje HDL que hemos visto hasta el momento es muy rigida y nos obliga a describir un circuito diferente para cada componente que necesitemos, sin poder reutilizar el código. Un ejemplo sencillo de este problema es la especificación de registros que almacenen diferente número de bits (8, 16, 32 o 64 bits). Con el conocimiento de HDL que tenemos, tendríamos que especificar un modulo diferente para cada

número de bits. Sin embargo, el uso de parametros permite especificar dieseños genericos que se adaptan a cada caso.

*En System Verilog los parametros de un modulo se declaran después de especificar el nombre del módulo y antes de especificar las señales de entrada y salida de la interfaz. Para ello se ha de utilizar la sintáxis descrita a continuación. Primero se delimita el inicio de la lista de parametros utilizando '#('. Despues separados entre comas se especifica cada parametro utilizando la palabra clave '**parameter**', seguido del nombre del parámetro y su valor por defecto. Este valor por defecto es fundamental para poder definir los parámetros base, cuando no se especifiquen al instanciar el módulo. Concluiremos la lista utilizando el cierre de paréntesis ')'.*

*A continuación se muestra un ejemplo de un registro sincrono (con señal de reloj clk_i) con permiso de escritura (señal load_i) y señal de reset negada (señal rstn_i). A veces los diseños requieren negar la señal de reset por cuestiones de optimización. El registro tiene un puerto de escritura (señal input_i) y de lectura (señal output_o) Este módulo nos permetirá especificar un registro de tamaño variable utilzando la variable '**WIDTH**' al instaciar el módulo.*

```
module register #(
    parameter WIDTH = 64
) (
    input logic              clk_i,   // Señal del reloj
    input logic              rstn_i,  // Señal de reset
    input logic              load_i,  // Señal de permiso de escitura
    input logic  [WIDTH-1:0] input_i, // Señal de entrada de WIDTH bits
    output logic [WIDTH-1:0] output_o // Señal de salida de WIDTH bits
);

always_ff @(posedge clk_i, rstn_i) begin
    if (~rstn_i) begin
        output_o <= '0;
    end else if (load_i) begin
        output_o <= input_i;
    end else begin
        output_o <= output_o;
    end
end

endmodule
```

A continuación se muestra como instanciar el anterior registro de forma que nos permita almacenar 8 bits. En caso de reset el circuito adquirirá el valor binario 00000000. Además, especificamos un circuito sencillo que cuando la salida binaria del registro es 00000000 escribe 11111111. Posteriormente, en cada ciclo el circuito almacenará el valor del anterior ciclo desplazado 1 bit, despreciando el bit de más peso e insertando un 0 en el bit de menos peso. De forma que el primer ciclo el registro tiene como salida 00000000, al siguiente ciclo 11111111, al tercer ciclo 11111110, al cuarto 11111100, y así hasta completar el ciclo y volver al primer valor.

```
module register_con_shift  (
    input logic              clk_i,   // Señal del reloj
    input logic              rstn_i,  // Señal de reset
);

// Declaramos las señales de entrada y salida del registro
logic [7:0] d;
logic [7:0] q;

// Instanciamos el registro con 8 bits
```

```
register #(8) instancia_de_register (
   .clk_i(clk_i),   // Pasamos el reloj entre modulos
   .rstn_i(rstn_i), // Pasamos la señal de reset
   .load_i(1'b1),   // Siempre se actualiza el registro
   .input_i(d),     // Próximo valor
   .output_o(q)     // Valor actual
);

// Lógica combinacional del siguiente valor
always_comb begin
    if (q == 'h0 ) begin
        d <= 8'b11111111;
    end else begin
        d <= (q << 1);
    end
end
```

2.4. Generación de estructuras regulares

Algunos diseños hardware se construyen utilizando replicas de mismo elemento. Frecuentemente la interconexión de las réplicas sigue un patrón regular. Para facilitar la descripción de estos diseños, los HDL disponen de la sentencia especiales que permiten generar circuitos regulares de forma concurrente. Esto quiere decir que estos cicuitos trabajan de forma paralela como hemos visto con la instanciación de módulos.

*En System Verilog disponemos de las sentencias '**generate**' y '**endgenerate**'. Entre estas dos sentecias podemos generar diferentes circuitos de forma regular. Existen dos formas para generar circuitos de forma iterativa usando la estructura '**for**'o de forma condicional '**if**'. Previamente a la palabra clave "generate"se puede declarar variables que se utilizarán en el bloque generativo utilizando la sentencia '**genvar**'. A continuación se muestra un ejemplo de como diseñar un mux de 8 entradas utilizando la sentencia 'generate' con una sentencia 'for'. Si bien el mux puede generarse de muchas formas, en este caso al utilizar la sentencia 'generate' nos evita tener que realizar 8 asignaciones condicionales.*

```
/*
* Ejemplo de un mux de 8 entradas de 64 bits
*/
module mux_8(
    input  logic [7:0] [64:0] mux_i,
    input  logic [2:0]        seleccion,
    output logic [64:0]       mux_o
);

// Declaramos 8 buses temporales de 64 bits
logic [7:0] [64:0] temp;

// El bucle for crea 8 circuitos de asignación condicional
genvar i;
generate
  for (i=0; i < 8; i++) begin
    assign temp[i] = (seleccion == i) ? mux_i[i] : 0;
  end
endgenerate

// Sólo uno de los 8 buses temp contiene el resultado seleccionado
// Realizamos una OR para obtener la salida del mux
assign mux_o = temp[0] | temp[1] | temp[2] | temp[3] |
```

```
                temp[4] | temp[5] | temp[6] | temp[7];
endmodule
```

3. Sumador de 4 bits

Para representar un numero natural (x) se utiliza un vector de bits $X = (x_{n-1}, ..., x_1, x_0)$ que se interpreta de forma ponderada:

$$\underline{x} = \sum_{i=0}^{n-1} x_i \times 2^i$$

donde x es el valor númerico que se calcula como la suma ponderada de los bits del vector de bits X.
Suma binaria. Dados los vectores de bits (X,Y) de entrada

$$X = (x_{n-1}, ..., x_1, x_0) \quad Y = (y_{n-1}, ..., y_1, y_0)$$

Que representan lo números naturales a y b respectivamente, la operación suma se expresa como $\underline{s} = ((\underline{x} + \underline{y})\ mod(2^n))$. El resultado s se representa mediante un vector de bits $S = (s_{n-1}, ..., s_1, 1_0)$. La suma de los vectores X e Y se efectúa sumando bit a bit los vectores de bits y propagando el acarreo.

$$a_i + b_i + c_i = 2 \times c_{i+1} + s_i; \quad 0 \leq i < n; \quad c_0 = 0$$

Condición de irrepresentabilidad. Si la suma $\underline{x} + \underline{y}$ es mayor que $2^n - 1$ el resultado no se puede representar con n bits.

3.1. Modelo de comportamiento de un sumador de 4 bits

En la siguiente figura se muestra un esquema del circuito de un sumador. Las señales X y Y son grupos de señales de 4 bits de entrada. La señal de salida S (SUMA) también es un grupo de señales de 4 bits. La señal de entrada C_0 es de un solo bit. La señal de salida C_4 es de un solo bit.

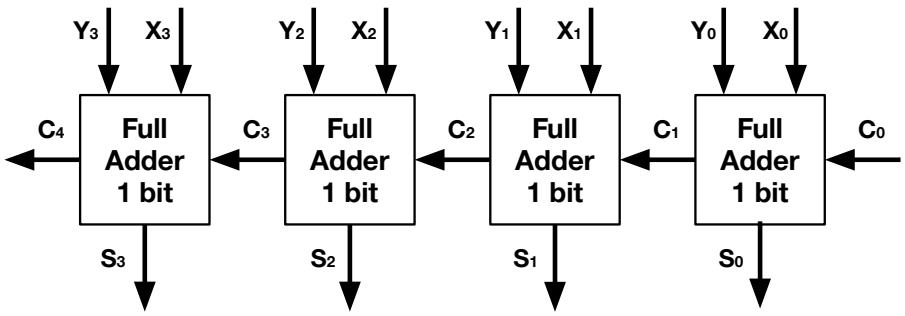


Figura 6: Sumador de 4 bits con propagación serie del acarreo.

Un sumador de 4 bits puede especificarse en System Verilog utilizando un modelo de comportamiento y sentencias de asignación de señales concurrentes, como en el siguiente código.

```
module FA (
input logic [3:0]  X, // Operando 1
input logic [3:0]  Y, // Operando 2
input logic        c_in, // Carry entrada

output logic [3:0] SUM, // Resultado
output logic       c_out // Carry salida
);
```

```
logic [4:0] senyal_interna;

always_comb begin
  senyal_interna = {1'b0, X} + {1'b0, Y} + {4'h0, c_in};

  c_out = senyal_interna[4];
  SUM   = senyal_interna[3:0];
end

endmodule
```

Notemos que en la especificación del sumador de 4 bits se utiliza una señal interna con un bit más para no perder el acarreo de salida. Por lo tanto la operación de suma se efectúa con vectores de 5 bits. Para ello se incrementa el número de bits con el que se representan los datos de entrada. En el caso de X e Y se añade un bit, mientras que en el caso de 'cen' se añaden 4 bits a la izquierda. En estas condiciones, el acarreo de salida es el bit más significativo de la 'senyal_interna' y el valor de la suma está representado por los bits restantes (3 al 0).

La descripción previa puede parametrizarse, de forma que sea más genérica utilizando un parámetro que especifique la longitud de la suma. Sin embargo para el diseño de esta práctica optaremos por un diseño modular, que describimos en las siguientes secciones.

3.2. Diseño jerarquico o estructural

A medida que los diseños son más complejos (incluyen más elementos) es necesario estructurar el diseño de forma modular. Ello ayuda a que el diseño se comprenda mejor, ya que se encapsulan detalles de implementación de algunas funcionalidades. Además, los módulos que implementan las funcionalidades pueden reutilizarse en otros diseños o refinar para el diseño que nos ocupa. Por otro lado, la verificación del correcto funcionamiento de un módulo concreto se puede efectuar de forma aislada e independiente de la interacción con otros módulos, lo cual facilita esta tarea.

Un **modelo estructural** especifica los elementos o componentes que se utilizan y cómo se intercontectan. Eventualmente se puede especificar lógica combinatoria o secuencial para realizar la correcta conexión de los componentes especificados. En la descripción de un modelo estructural, cada componente o elemento se ha definido previamente y puede haber sido descrito utilizando un modelo estructural o un modelo de flujo de datos. Por lo tanto podemos ver una descripción estructural como un diagrama esquemático. Esto es, un esquema de bloques del circuito en el cual se describen los componentes y su interconexión. Por ello, es útil partir del esquema de circuito para efectuar la descripción HDL. En la práctica anterior ya vimos un ejemplo de diseño estructural implementando el siguiente esquema.

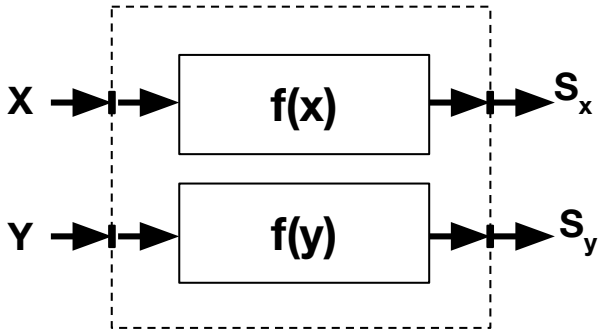


Figura 7: Ejemplo de módulo estructural implementado instanciado dos veces el módulo que implementa la función f().

Para esta práctica, deberéis escribir un módulo en System Verilog llamado 'adder_4bits.sv'. Este módulo deberá implementar un sumador de 4 bits con un patron de diseño estructural que instancie 4 'full adder' como el de la práctica anterior. Para ello os daremos el diseño original 'full_adder.sv'. La siguiente figura muestra el esquema de conexión de 4 sumadores de 1 bit para construir un sumador de 4 bits con propagación serie del acarreo.

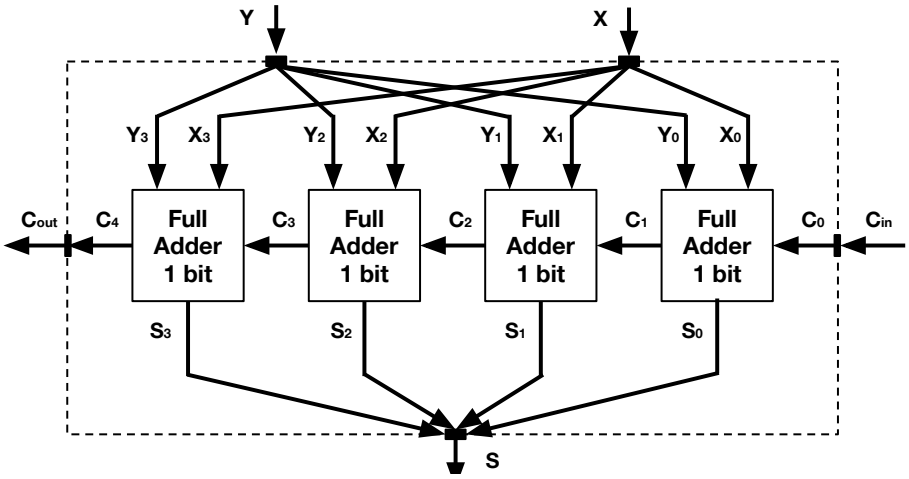


Figura 8: Esquema de un sumador de 4 bits construido con sumadores de 1 bit.

3.3. Arquitectura de un sumador de longitud variable

Ahora que comprendemos el diseño de un sumador de 4 bits, podemos extender este módulo para que pueda realizar otras sumas de diferente longitud (p. ej. 8, 16, 32 o 64 bits). Sin embargo, la carga de trabajo para especificar estos módulos puede ser elevada, al tener que instanciar hasta 64 'Full Adders' en el caso de un sumador de 64 bits. Como ya hemos visto los lenguajes HDL permiten la generación de estructuras regulares.

Para esta práctica, deberéis escribir un módulo en System Verilog llamado 'adder_nbits.sv'. Este módulo deberá implementar un sumador de N bits empleando un sumador de 1 bit de la práctica anterior. El módulo debera utilizar la clausula generate para poder instanciar un numero de sumadores especificado por parametro.

4. Acumulador de N bits

Ahora que ya conocemos la arquitectura de un sumador y de un registro vamos a diseñar nustro primer componente de alto nivel, el acumulador. Un acumulador es un circuitos combinacional y secuencial conformado por un registro y un sumador. El objetivo de un acumulador es almacenar temporalmente los resultados aritméticos intermedios de una operación conformada por varias operaciones aritméticas (p. ej. la suma de todos los elemntos de una lista de enteros)

Sin un componente como el acumulador, sería necesario escribir el resultado de cada cálculo en la memoria o en un registro de proposito general de la CPU, quizá para ser leída inmediatamente después. El acceso a memoria o al banco de registros tiene asociado un coste en ciclos y energia, y por lo tanto el uso de un acumulador es más rápido y eficiente. Además el acumulador nos permite almacenar tipos de datos más precisos que los utilizados en los registros de proposito general. Por ejemplo, un registro general de una CPU de 16 bits podrá representar un entero sin signo del 0 hasta 65535. Una suma lo suficientemente larga puede desbordad dicho registro. En cambio en un acumulador podemos utilizar de 32 bits que nos permite representar del 0 al 4294967296.

El para uso del acumulador, el programador deberá inicializar el registro de este a cero, o a la identidad de la operación que vaya a realizar (p.ej 1 para multiplicaciones). Entonces el programador realizará una

serie de acumulaciones en el que cada elemento es sumado al valor del acumulador y guardado en este al final del ciclo de reloj. La siguiente figura muestra el diagrama de un acumulador.

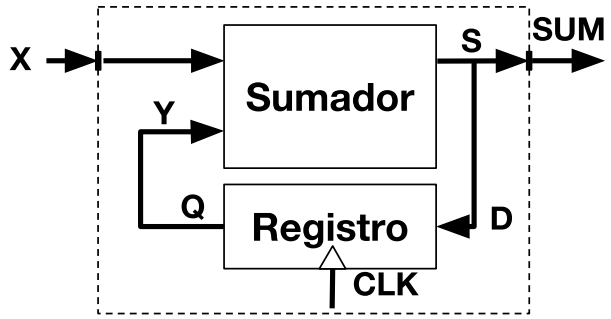


Figura 9: Esquema de un acumulador construido con un sumador y un registro.

Para la última parte de la práctica deberéis implementar el diseño de un acumulador de N bits, reutilizando los modulos del sumador y del registro parametrico especificado en el enunciado. Es necesario que el acumulador tenga una señal de reset para inicializar el registro interno.

5. Test de Verificación ("testbench")

En esta sesión para verificar nuestra implementación del sumador de 4 bits utilizaremos de nuevo un banco de prueba exhaustivo ('tb_adder_4bits.sv') que nos permitirá probar todos los posibles valores de entrada y comprobar así que la salida de nuestro módulo es correcta. Para compilarlo y ejecutarlo usando un 'Makefile'. Debéis usar los siguientes comandos:

```
$ make build
$ make sim
```

Esto os generará un fichero waveform.vcd como el de la práctica anterior. Podeis visualizar su contenido usando:

```
$ gtkwave waveform.vcd
```

Para el sumador de N bits dispondréis de un banco de pruebas adicional ('tb_adder_nbits.sv'). De nuevo podéis ejecutarlos usando los siguientes comandos:

```
$ make build_n
$ make sim_n
```

De nuevo este último comando os generará un fichero waveform.vcd que puede sobre escribir el anterior. Por lo tanto tened cuidado al ejecutar los bancos de prueba.

Para comprobar el correcto funcionamiento del acumulador deberéis escribir vuestro propio banco de pruebas. Para ello teneis a vuestra disposición un pequeño escleto del banco de pruebas llamado 'tb_accum_nsbits'. Este banco de pruebas suma los quince primeros numeros enteros que suman hasta 120.

```
$ make build_accum
$ make sim_accum
```

6. Trabajo a realizar

1) Implementad un sumador de 4 bits utilizando un diseño estructural que instancie 4 'full adders'. Para facilitaros la práctica os proveemos del fichero 'full_adder.sv'. Para ello, deberéis que crear un módulo llamado '*adder_4bits.sv*'. **Debéis entregar dicho módulo al final de la práctica.**

2) Implementad un sumador de N bits utilizando un diseño parametrico que instancie N 'full adders'. Para ello, deberéis que crear un módulo llamado '*adder_nbits.sv*'. **Debéis entregar dicho módulo al final de la práctica.**

3) Implementad un acumulador de N bits (registro y adder de 4 bits). Para ello, deberéis que crear un módulo llamado '*accum_nbits.sv*'. **Debéis entregar dicho módulo al final de la práctica.**

4) Responded a las preguntas de la sigueinte sección.

6.1. Preguntas

Pregunta 1: Modificad las instrucciones de la sección de verificación para estimular las entradas del sumador de 4 bits. Para ello usareis vuestros dos números de DNI modulo 16 (p. ej. para el DNI 12345678-Z, teneis que hacer 12345678 mod 16 = 14). Si sólo hay un estudiante en el grupo, suma 5 al número de DNI modulo 16. Usando 'c_in' = 1, comprobad el resultado obtenido.

DNI (Decimal)	DNI mod 16	Entradas	Binario	Decimal
		A		
		B		
		c_in	1	1

Suma		c_out		

Pregunta 2: La implementación de Supongamos que el sumador de 4 bits tiene todas las entradas estables en el instante t_0. En el instante t_1 (1000 ps después de t_0) un único bit de entrada de los 9 que hay cambia de valor. Al estabilizarse la señal en el instante t_2 el sumador devuelve un valorde suma igual a 2 en representación decimal, sin overflow ($c_out = 0$). Para ello, deberéis utilizar la función '*transition_test*' para establecer las señales de entrada.

Señal	Valor (t0)	Valor (t1)	Valor (t2)
A			
B			
c_in			
S			2
c_out			0
Tiempo	0	1000	

Pregunta 3: Instanciad un sumador de 32 bits en el banco de prueba y tornad a realizar la pregunta 2 sin realizar la operación de módulo 16.

DNI (Decimal)	Entradas	Binario	Decimal
	A		
	B		
	c_in	1	1

Suma		c_out		

Pregunta 3: Instanciad el acumulador con 32-bits y tornad a realizar la pregunta 2. Para ello es deberéis dividir cada DNI en dos partes de 4 digitos decimales (p. ej. para el DNI 12345678-Z debereis usar los números 1234 y el 5678). Cada parte será la entrada en un ciclo del acumulador. Si sólo hay un estudiante en el grupo, utilizad dos veces vuestro DNI.

Ciclo	Entrada X (Dec)	Salida (Bin)	Salida (Dec)
0			
1			
2			
3			