# SESIONES TUTORIZADAS DE PREPARACIÓN DE LA PRÁCTICA 1

Durante esta sesión trataremos de aproximarnos al uso de Verilog, mediante ejemplos prácticos. El objetivo de estas sesiones es conseguir que se llegue a la primera práctica con una cierta destreza en el uso de este lenguaje y de las herramientas que lo acompañan, mediante ejemplos de las construcciones más importantes que usaremos en esa práctica, y enfatizando el modelado de la lógica combinacional.

El contenido de las sesiones no es un tutorial exhaustivo. Se aconseja haber ojeado previamente los tutoriales recomendados en el aula virtual o consultarlos si surge cualquier duda.

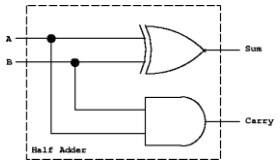
Recuerden que esta sesión, aunque tutorizada, es obligatoria, al igual que las dos sesiones en las que se desarrollará la práctica 1.

## DESARROLLO DE LA SESIÓN

Vamos a considerar una serie de diseños realizados con diferentes construcciones y podremos más adelante comprobar su correcto funcionamiento. Estos diseños de ejemplo serán diferentes tipos de sumadores.

### BLOQUE SEMISUMADOR

Empezamos con el sumador más elemental, el semisumador (*half-adder* o ha). Este bloque posee dos entradas de un bit denominadas **A** y **B**, y produce un bit de suma **Sum** y un bit de acarreo **Carry**. La suma es simplemente una **xor** de ambas entradas y el acarreo una **or**




La forma más simple de modelar en Verilog el semisumador es utilizar el diseño *estructural*. Esto significa que construiremos el semisumador utilizando componentes más simples y describiendo cómo estos componentes se conectan internamente para, a partir de las señales de entrada, proporcionarnos las señales de salida. Una vez que definamos el bloque o módulo semisumador, podremos usarlo para construir otros bloques más complejos. Podemos considerar entonces un diseño en Verilog como una jerarquía de módulos o bloques. Arriba del todo tendremos nuestro diseño de más alto nivel, el módulo denominado "*Top-level*" que incluirá toda una serie de instanciaciones de módulos más simples, junto con sus interconexiones, que a su vez pueden estar formados por otros módulos todavía más simples, y sucesivamente continuar hasta llegar a los módulos más elementales que son *primitivas* de Verilog.

Veamos brevemente el código Verilog de tipo estructural para el módulo semisumador:

```
//Semisumador de dos entradas de 1 bit realizado a partir de puertas

module ha_v1(output wire sum, output wire carry, input wire a, input wire b);

//Declaración de conexiones o variables internas: no hay ninguna

//Estructura interna: Instancias de puertas y sus conexiones

xor xor1(sum, a, b);
and and1(carry, a, b);

endmodule
```

Vemos la declaración del módulo **ha_v1**, con la sentencia **module** indicando en sus argumentos que posee dos entradas y dos salidas de tipo **wire**. Esta declaración es muy similar al prototipo de una función en un lenguaje de programación, sólo que aquí indicamos los tipos de conexiones externas que tendría nuestro módulo o bloque. Dentro de la declaración **module** (entre **module** y **endmodule**) describimos el funcionamiento del bloque. En esta ocasión, al ser una descripción estructural, daremos los submódulos que lo constituyen y su interconexión. Se implementa instanciando dos puertas primitivas del lenguaje, una **xor** y una **and**, a las que se da nombre (xor1 y and1) e indicamos cómo están conectadas, poniendo las señales como "argumentos" de los bloques instanciados. En estas primitivas, la salida siempre es el primer "parámetro" y el resto, en número variable, serán entradas. En este caso todas las señales que aparecen son entradas o salidas del módulo semisumador global, no siendo entonces necesario definir ninguna conexión interna.

Existe otra forma de declarar las entradas y salidas (más arcaica, similar al lenguaje C más antiguo) en la que la declaración de entradas y salidas puede ir dentro del módulo en lugar de acompañar a los nombres de las variables en el 'argumento' del módulo. Además, si se omite el tipo en una variable, se entenderá que es **wire**. Veámoslo en el ejemplo siguiente:

```
//Semisumador de dos entradas de 1 bit realizado a partir de puertas, Notación vieja

module ha_v1_1(sum, carry, a, b);

output wire sum, carry;
input a, b;  //tipo wire por defecto

//Declaración de conexiones o variables internas: no hay ninguna

//Estructura interna: Instancias de puertas y sus conexiones

xor xor1(sum, a, b);
and and1(carry, a, b);

endmodule
```

Una vez definido este módulo **ha_v1** en un fichero ha_v1.v (o el **ha_v1_1** en el fichero ha_v1_1.v), podremos usarlo en otros diseños de una forma muy similar a como hemos usado las primitivas **and** y **xor**, mediante su instanciación en otro módulo. Esto es similar a haber definido en un lenguaje orientado a objetos una clase y ahora emplearla nombrando instancias o ejemplares particulares de objetos de esa clase en un diseño. En el ejemplo siguiente:

```
module pepe(…)
…
ha_v1 mi_semisumador(suma, acarreo, op1, op2);
…
endmodule
```

definimos un módulo llamado pepe que incluye en su interior una instancia del semisumador **ha_v1** que se llama `mi_semisumador` y que utiliza las señales del módulo pepe llamadas **suma**, **acarreo**, **op1**, **op2**

que lo conectarán a otras instancias de otro tipo de módulos o a las entradas y salidas del módulo pepe. Esas señales se identifican con las que declaramos en la definición del módulo **ha_v1** por su orden, es decir, se corresponden con `sum, carry, a y b.`

Existe una notación alternativa para instanciar un módulo de otro, en el que las señales que usamos para conectar la instancia no se identifican con las de la definición del módulo por el orden, sino señalando qué señal del módulo actual se corresponde con la de la definición de la forma siguiente:

```
module pepe(…)
…
ha_v1 mi_semisumador(.sum(suma), .carry(acarreo), .b(op2), .a(op1));
…
endmodule
```

Es decir, cada conexión responde al esquema:

.señal_definición (señal_módulo_actual))

Y entonces ya no es necesario respetar el orden de la definición original.

La instanciación, es decir, el poder usar un ejemplar concreto de un tipo de módulo, es lo que define la jerarquía antes nombrada. Existirá un módulo (el de mayor nivel o *top-level* nombrado antes) que no es instanciado por ningún otro, que será la cima de la jerarquía.

Las declaraciones de variables más comunes utilizan los tipos **wire** y **reg**. El tipo **wire** (cable) declara que la variable se usará como interconexión, es decir, no tiene capacidad para almacenar valores, sino que simplemente se limita a propagar el valor de una salida de un módulo a otros puntos (entradas) de otros módulos. El tipo **reg** (que, a pesar de su nombre, no tiene por qué ser considerado un registro) sí que tiene estado almacenado y ese estado no cambiará mientras no se le asigne otro valor a la variable. Típicamente usaremos este último tipo para modelizar elementos con estado o bien como variables a las que podamos asignar valores en el modelado de comportamiento. Existen además otros tipos de variables que nos resultan más familiares de la programación tradicional como enteros, de punto flotante, etc.

Hasta ahora, un programa Verilog consiste en una declaración de módulo (con sus entradas y salidas y tipos) y dentro del módulo sentencias de declaración de interconexiones o variables (no hemos visto de momento ejemplos de esto) e instanciaciones de otros módulos (que en sus “argumentos” tienen esas interconexiones y/o nombran entradas y salidas de la declaración del módulo global).

Una nueva sentencia muy útil para modelar lógica combinacional es la sentencia **`assign`** o asignación continua. Funciona de forma similar a la asignación a una variable de una expresión. A diferencia de un lenguaje de programación en el cual la asignación se ejecuta en el momento en el que el flujo de ejecución llega a la sentencia, aquí la asignación está activa en todo instante. El nombre proviene de que continuamente se está dando el valor de la expresión a la variable, exactamente como ocurriría en un bloque combinacional real, por lo que se recomienda su uso para modelar este tipo de lógica. En la expresión podemos emplear un conjunto muy variado de operadores similar al que tendríamos en un lenguaje de alto nivel. La sentencia **`assign`** se debe usar dentro de un **module** como una entidad del mayor nivel (como las instancias de otros módulos o las declaraciones de interconexiones) y nunca dentro de otros bloques (que veremos más adelante) porque entonces se interpretará su funcionamiento de forma diferente. La variable de salida a la que se asigna la expresión debe ser de tipo **wire**. Veamos como ejemplo del uso de **`assign`** una nueva implementación **`ha_v2`** del semisumador.

```
//Semisumador de dos entradas de 1 bit realizado a partir de sentencias assign

module ha_v2(output wire sum, output wire carry, input wire a, input wire b);

// semisumador con asignaciones continuas (assign)

assign sum = a ^ b;   //operador xor (bit a bit)
assign carry = a & b; //operador and (bit a bit)

endmodule
```

Veamos como ilustración del **operador concatenación {...}** otra forma ligeramente distinta de implementar el semisumador. Este operador permite concatenar sus argumentos como cadenas de bits y podremos usar su resultado donde nos convenga o también podremos asignar valores a la concatenación. En este último caso cada variable que se concatena recibirá su parte correspondiente del valor asignado. En el ejemplo se usa el operador suma convencional, que realiza la suma del bit a y el b. El resultado, en general, necesita de dos bits, ya que puede producirse acarreo. Aquí lo que se hace es asignar el resultado de la suma a la concatenación de carry y sum, de forma que se producen dos bits siempre, uno correspondiente a la parte más significativa de la concatenación, es decir, el bit de acarreo, y el otro menos significativo que corresponderá al bit de suma.

```
//Semisumador de dos entradas de 1 bit realizado a partir de assign,
// operador concatenación y operador suma

module ha_v3(output wire sum, output wire carry, input wire a, input wire b);

assign {carry, sum} = a + b;  //operador suma junto con operador concatenación

endmodule
```

La sentencia **assign** nos proporciona la capacidad de dar valores a una variable de tipo wire usando una expresión que puede contener operadores muy potentes, de forma similar a cómo escribiríamos esa expresión en un lenguaje de alto nivel. Un paso más en esa dirección nos aleja de la estrategia de describir nuestros módulos en forma estructural y nos acerca a una nueva forma de crearlos, el *modelado de comportamiento*. En esta nueva estrategia, se trata de describir cómo cambian las salidas en función de las entradas, pero sin necesariamente desarrollar una descripción de su estructura interna en términos de módulos más simples. Ahora se trata de codificar esas relaciones de forma procedural, en forma similar a un lenguaje de alto nivel.

Las construcciones que nos permiten realizar el modelado de comportamiento son varias, pero esencialmente lo implementaremos mediante los bloques **always** o **initial**. El interior de cualquiera de estos dos bloques se ejecuta normalmente de forma secuencial, sentencia a sentencia. Muchas de esas sentencias son construcciones muy similares a los lenguajes de alto nivel y en particular al C: asignaciones a variables (con el operador **=**, no con el **assign**), bucles **for**, **while**, **repeat**, etc., sentencias **if..then** y **case**.. La mayor diferencia con la sintaxis del C es que se usan las palabras clave **begin..end** para delimitar un bloque formado por varias sentencias, en lugar de las llaves. Debido a que pretendemos modelar numerosos componentes hardware, no podremos escribir simplemente un programa secuencial que realice una serie de pasos uno detrás de otro, sino que tendremos que reflejar en nuestras simulaciones el que esos componentes están funcionando e interactúan entre sí de forma concurrente, en paralelo. Los bloques nombrados antes tienen que contener **cómo** funciona el componente, de forma procedural, pero también tenemos que tener un mecanismo para decidir **cuándo** debemos ejecutar o activar un bloque de código. Esto normalmente se producirá cuando una señal de entrada al módulo cambie, ya que deberemos rehacer nuestro cálculo de las salidas.

Un bloque **always** entra en funcionamiento en cuanto hay algún evento que lo active. Ese evento o condición lo indicaremos después del **always** como una especie de lista de parámetros entre paréntesis precedida del símbolo **@**. Esto se denomina lista de sensibilidad y contendrá una lista de variables. Mediante la lista de sensibilidad indicamos al sistema que si alguna de las variables de la lista cambiara, se debe ejecutar una reevaluación del bloque. En el caso de usarlo para modelar lógica combinacional, claramente deberíamos poner en la lista de sensibilidad las entradas de dicha función lógica. En este tipo de modelado procedural, la variable a la que asignamos la salida debe ser de tipo **reg**, de forma que mantenga su valor entre las sucesivas reevaluaciones.

El bloque **initial**, por su parte siempre se ejecuta en el momento inicial de la simulación. Podemos tener tantos bloques **initial** o **always** dentro de un módulo como queramos. Todos los **initial** se empiezan a ejecutar simultáneamente en el tiempo 0 de simulación y cada **always** entrará en ejecución en el instante de la simulación en que cambie alguna de las variables de su lista de sensibilidad.

Veamos como ilustración de todo esto el ejemplo del semisumador realizado de forma procedural.

```
//Semisumador de dos entradas de 1 bit realizado con sentencia always

module ha_v4(output reg sum, output reg carry, input wire a, input wire b);

// construcción always (procedural), las asignaciones deben ser a variables con
// estado ('sum' y 'carry' ahora son de tipo reg)

always @(a, b) //alternativamente, always @(a or b)
               // o always @* (automático, considera todas las var. que intervienen)
begin  //always
  sum = a ^ b;
  carry = a & b;
end    //always

endmodule
```

Como se ha señalado, las variables de salida ahora deben ser de tipo **reg** ya que, a diferencia de la asignación continua, no se están calculando en cada instante, sino en las reevaluaciones, es decir, cuando hay un cambio en **a** ó **b**. Vemos que en este caso (lógica combinacional) esta sentencia no aporta mucho frente a la simplicidad de la asignación continua.

Así pues, según lo visto hasta ahora, nuestro módulo Verilog puede contener la declaración de módulo, las declaraciones de variables o interconexiones que sean necesarias, sentencias **assign** y los bloques **always** e **initial** que deseemos.

## LOS BANCOS DE PRUEBA O *TESTBENCHES*

Los *Testbench* son módulos que creamos para comprobar la corrección de nuestros diseños, dando valores a las señales de entrada y permitiéndonos observar las salidas y el funcionamiento del módulo testeado. Dada su finalidad, frecuentemente emplearemos sentencias que sirven para ver por consola los valores de las variables, para guardar sus valores en un fichero, para gestionar el tiempo de la simulación, etc. Está claro que este tipo de sentencias no tienen que ver con la lógica de nuestro diseño, ni mucho menos serían sintetizables en un dispositivo. Además de este tipo de sentencias, también usaremos el resto del lenguaje Verilog convencional que hemos visto hasta ahora y alguna sentencia como las de modelado del tiempo que no hemos usado. La estructura de un testbench es similar a la de los ficheros que hemos visto, sigue existiendo una declaración de módulo pero no tiene argumentos porque es el módulo más alto de la jerarquía y no lo conectaremos a nada. Veamos el código de un testbench creado para comprobar el funcionamiento de nuestro diseño de semisumador (podría ser cualquiera de ellos, pero en esta ocasión el que se ha instanciado dentro del testbench es el ha_v1)

```
// Testbench para modulo semisumador visto antes

`timescale 1 ns / 10 ps
//Directiva que fija la unidad de tiempo de simulación y el paso de simulación

module ha_v1_tb;

//declaracion de señales

reg test_a, test_b; // las señales de entrada al semisumador. Se han declarado reg
// porque queremos inicializarlas
wire test_sum, test_carry; //señales de salida, se declaran como wire porque sus
// valores se fijan por el semisumador

//instancia del modulo a testear, con notación de posiciones de argumentos
ha_v1 ha1(test_sum, test_carry, test_a, test_b);

// Lo siguiente comentado es una notación alternativa para instanciar el módulo, los
// parámetros se denotan con un punto seguido del nombre del parámetro en la definición
// original del módulo y entre paréntesis a qué se conecta en el modulo actual
// no importa el orden de los parámetros definidos así -->
// ha_v1 ha1(.a(test_a), .b(test_b), .sum(test_sum), .carry(test_carry));

initial
begin
  // sentencia para mostrar los valores de tiempo y variables, en cuanto cualquiera de
  // éstas cambie
  $monitor("tiempo=%0d a=%b b=%b suma=%b acarreo=%b", $time, test_a, test_b, test_sum,
test_carry);

  //vector de test 0
  test_a = 1'b0;
  test_b = 1'b0;
  #20;

  //vector de test 1
  test_a = 1'b0;
  test_b = 1'b1;
  #20;

  //vector de test 2
  test_a = 1'b1;
  test_b = 1'b0;
  #20;
  //vector de test 3
  test_a = 1'b1;
  test_b = 1'b1;

  $finish;  //fin simulacion

end
endmodule
```

En este caso el testbench no hace nada complicado, se limita a inyectar valores a las entradas a y b en sucesión e imprimir dichos valores y las salidas. Veamos en detalle cómo se logra esto. La primera sentencia es una directiva, **`timescale**, que va fuera de la declaración de módulo. Tiene que ver con el tiempo de la simulación y fija la unidad de tiempo que se usará en las sentencias de retardo, así como el avance o paso de tiempo que usará el simulador (separadas por '/'), en este caso un nanosegundo y un picosegundo respectivamente. Cuando se compilan juntos los varios ficheros que componen un diseño, normalmente es mejor especificar esta sentencia en un sólo fichero (sólo en el testbench) y poner el testbench como el primer fichero de la lista a compilar, de forma que la directiva fije los parámetros de tiempo para todos ellos.

Después de esta sentencia viene una declaración de módulo (sin parámetros), definiendo el módulo de más alto nivel (el que no es referenciado en ningún otro). Sigue la declaración de variables, las que deseemos inicializar a algún valor de test debido a que son las entradas al módulo que deseamos comprobar, deberán ser declaradas como **reg** para que retengan esos valores. Las que simplemente

sean salidas de un módulo que les da valor pueden ser **wire**. La instanciación del módulo semisumador ocurre a continuación, poniendo en sus parámetros las variables que acabamos de definir a modo de conexión. Seguimos con un bloque de código procedural, el **initial**. Éste, como se dijo antes, se ejecuta siempre en el tiempo 0 de la simulación, es decir, en el instante inicial. Como incluye varias sentencias, éstas se deben rodear de **begin** y **end**. La primera sentencia es **$monitor** y sirve para imprimir en la consola los valores de variables que incluyamos en sus argumentos, su sintaxis es parecida al printf del C: hay una cadena de formato inicial, indicando los tipos de variables y el formato de impresión deseado y después siguen las variables en el mismo orden y número en que aparezcan sus tipos en la cadena de formato. En este caso los formatos usados son %t que formatea valores de tiempo y %b que indica que queremos representar las variables en binario. Las variables a representar son la variable especial del sistema **$time**, que representa el tiempo actual simulado y las entradas y salidas al semisumador. Esta sentencia mostrará los valores de las variables siempre que ocurra un cambio en las mismas **($time** excluido). A continuación se inicializan las variables de entrada y aparece una sentencia de retardo, **#num**. Esta sentencia avanza el tiempo de simulación **num** unidades de las que se definieron en **`timescale** simulando el paso del tiempo. Puede aparecer en sentencias aisladas como la del ejemplo pero es también posible usarla en medio de asignaciones, en declaraciones de variables **wire**, en asignaciones continuas y en múltiples otras construcciones en las que interese simular una duración. Hasta ahora en la simulación no habíamos incluido ninguna sentencia en la que se apreciara el avance del tiempo. Por ejemplo, las funciones que hemos definido con **assign** ocurrirán de forma instantánea, sin hacer transcurrir nada de tiempo de la simulación. Si no incluyéramos algún retardo, todo ocurriría en el tiempo 0 de la simulación. Como nuestra intención en algunos casos es simular la evolución temporal del diseño, es necesario incorporar estos retardos, simulando lo que el dispositivo real tardaría en producir su salida, o el retardo en propagarse a las salidas un cambio en las entradas de una puerta lógica.

En nuestro ejemplo, el cambio en las variables de entrada que ocurre en los instantes 20, 40, etc. causará el cambio en las de salida (en este caso en que no hay retardos en el módulo semisumador, la propagación de las entradas a las salidas ocurrirá de forma instantánea en la simulación) y esos cambios a través de **$monitor** hará que se muestren en pantalla las variables. Por último, la sentencia de sistema **$finish** causa la finalización de la simulación.

## VISUALIZACIÓN CON GTKWAVE

Aunque es posible depurar nuestros diseños simplemente con la salida por pantalla, es mucho más cómodo emplear alguna herramienta que nos permita comprobar gráficamente lo que está ocurriendo. Para ello, debemos insertar los comandos

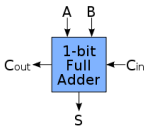
```
$dumpfile("ha_v1_tb.vcd");
$dumpvars;
```

en el bloque **initial** del fichero del testbench, después del **$monitor**, por ejemplo. Estos comandos producen un volcado de todas las variables al fichero `ha_v1_tb.vcd` que podemos visualizar con la utilidad GTKWave.

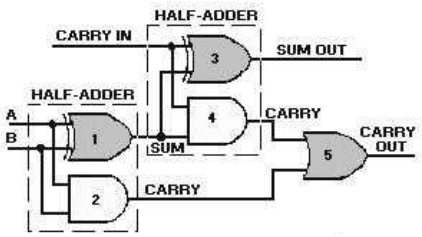
Es imprescindible repetir la compilación y ejecución para producir dicho fichero y observar su contenido con el GTKWave. Si no se observan bien, posiblemente estemos viendo una escala de tiempo demasiado corta. El botón sobre el que aparece la leyenda “Zoom Fit” o la opción Best-Fit del menú Time recalculan el zoom para visualizar toda la simulación.

## BLOQUE SUMADOR COMPLETO Y SUMADOR CON ARRASTRE DE ACARREO

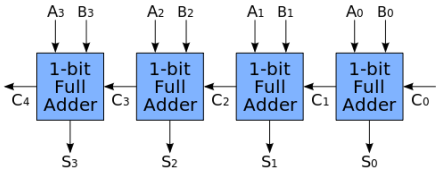
El sumador completo (*full-adder* o fa) incorpora respecto al semisumador la posibilidad de sumar un acarreo de entrada procedente de una etapa previa.




Se puede implementar combinando dos semisumadores, uno para sumar **a** y **b** y otro para añadir al resultado el bit de acarreo previo. El acarreo de salida es simplemente la **or** entre los acarreos de cada semisumador.




La utilidad del sumador completo reside en que se consigue elaborar sumadores de un número superior de bits, simplemente enlazando los acarreos de una etapa con la siguiente (sumador con arrastre de acarreo), como aparece en la siguiente figura para 4 bits




Usando las capacidades de modelado de comportamiento, podemos escribir una implementación del sumador diferente, no estructural, en la que vemos cómo se usan vectores de bits como parámetros:

```
//Sumador de dos entradas de 4 bits realizado a partir de assign y operador suma

module adder(output wire [3:0] S, output wire carry, input wire [3:0] A, input wire
[3:0] B);

assign {carry, S} = A + B;    //operador suma

endmodule
```

En este caso el comportamiento del módulo sería el correcto, pero al ahorrarnos todos los detalles de su estructura interna también se nos hace imposible modelar los detalles de cómo se propagan los acarreos y su evolución en el tiempo. Se ha introducido la notación de los corchetes **[…]** para especificar que en lugar de bits individuales queremos considerar vectores de bits, enumerando dentro el rango de bits que deseemos y poniéndolos justo antes del nombre de la variable.

**Actividades:**

1. **Realiza la implementación de un módulo full adder en el fichero `fa_v1.v`. Emplea una implementación estructural usando puertas lógicas y módulos semisumadores, como se indica en el diagrama de bloques.**
2. **Adapta el módulo testbench para probar este nuevo módulo. Si el nuevo fichero con el testbench se llama `fa_v1_tb.v` , compila todo usando**

   ```
   iverilog -o test fa_v1_tb.v fa_v1.v ha_v1.v
   ```

   **Ejecuta con**

   ```
   vvp test
   ```

3. **Crea un nuevo fichero `fa_vr.v`, que sustituya los semisumadores de `fa_v2.v` por los contenidos en el fichero `ha_vr.v`. Esta nueva definición de semisumador incluye la modelización del tiempo que tardan éstos en producir sus salidas (se ha asumido 1 retardo tanto en el acarreo como en la suma de cada semisumador). Prueba este nuevo sumador completo con su correspondiente testbench y observar las señales con GTKWave.**
4. **Crea un sumador completo de 4 bits en el fichero `sum4.v`, usando los módulos definidos en el fichero fa_v2.v (que incluyen retardos). Intenta comprobar los resultados usando el testbench `sum4_tb.v`. ¿Ves algo extraño? Usa el GTKwave para tratar de encontrar el error.**