Arquitectura de Computadores II: Práctica 1

16 de septiembre de 2024

1. Introducción a SystemVerilog y el entorno de trabajo

Los objetivos de esta sesión son por un lado el aprendizaje de un un subconjunto del lenguaje SystemVerilog. Un lenguaje de descripción de hardware (Hardware Description Language o HDL en ingles) permite especificar circuitos digitales, muy utilizado en la industria. Y por otro lado, el objetivo de esta práctica es familiarizarse con el entorno de trabajo de Verilator y GTKWave.

La principal tarea del laboratorio consistirá en realizar un sumador de un bit, utilizando SystemVerilog. Además utilizaremos diferentes herramientas para simular el circuito y verificar su correcto funcionamiento.

2. Fundamentos de los Hardware Description Languages (HDL)

En esta sección introduce conceptos básicos que son aplicables a cualquier lenguaje HDL (VHDL, Verilog, SystemVerilog, etc.). Estos conceptos son esenciales para describir correctamente circuitos utilizando un HDL. También se describen aquellas buenas practicas que nos ayudan a reducir el numero de errores introducidos en nuestros dieños, a depurar estos errores y a sacar el máximo partido a los módulos que ya hemos descrito. *Los conceptos específicos de **SystemVerilog** están destacados en itálico.*

Los HDL son muy similares a los lenguajes de programación tradicionales cómo C, C++ o Java. Comparten conceptos cómo el tipado de datos (p. ej. int, float en C) y partes de la sintaxis (p. ej. if, else, for, etc.). Sin embargo, el paradigma de programación de los HDL es fundamentalmente distinto de el de un lenguaje de programación imperativo. Un lenguaje HDL se utiliza para efectuar descripciones de circuitos hardware. Mediante HDL se pueden describir circuitos combinacionales y secuenciales. Recordemos que un circuito es un conjunto de elementos que trabajan en paralelo. Por tanto, un el lenguaje HDL permite expresar paralelismo. Al utilizarlo hay que pensar en puertas lógicas y circuitos secuenciales (p. ej. Registros o Latches). También existen variables y funciones, pero rara vez las utilizaremos.

Idealmente, a partir de un circuito descrito en HDL podemos obtener un circuito con puertas lógicas y elementos de memorización. El proceso de traducción de HDL a un circuito lógico se denomina síntesis. Sin embargo no todos los circuitos descritos en un lenguaje HDL pueden ser sintetizados, ya que estos pueden contener descripciones imposibles de realizar en un circuito (p. ej. circuitos combinacionales con retroalimentación). El coste de traducir a hardware cualquier especificación en HDL es muy costoso económicamente (las licencias de estos programas son caras), pero también requieren una cantidad de tiempo y energía elevada (el proceso puede llevar horas incluso para circuitos sencillos).

Un circuito descrito en HDL puede ser simulado utilizando una herramienta que reproduce el funcionamiento del circuito descrito. Los simuladores permiten abaratar los costes de desarrollo de los diseños

digitales. Por ello en este curso únicamente utilizaremos simuladores como herramienta para comprobar que nuestros diseños funcionan.

2.1. Tipos de datos

Cómo hemos comentado anteriormente, una de las características de los HDL es el tipado de las variables. Por lo general, los HDL permiten definir tipos de datos similares a los que existen en C (integer, float, char). Sin embargo, los HDL incluyen tipos de datos específicos para describir hardware: cables, registros, bits, bytes, etc.

El tipo de datos básico de SystemVerilog es ***logic****, que representa un bit. Una variable declarada con logic, puede ser sintetizada en un cable o un registro dependiendo de su uso. Las herramientas de síntesis deberán determinar que tipo de componente electronico representa una variable de tipo logic. Existen otros tipos de datos que reperesentan bits como wire o reg, sin embargo* ***su uso está desaconsejado*** *porque pueden producir circuitos no sintetizables. Podemos extender el tipo de datos básico para que represente más de un bit añadiendo la extensión del tipo entre corchetes. También podemos extender el tipo de datos básico en vectores añadiendo la extensión del vector después del nombre de la variable. Esto nos permite poder declarar multiples instancias de una variable, que es muy util para diseñar estructuras de datos.*

El tipo logic permite 4 estados diferentes: 0, 1, X, Z. Los estados 0 y 1 se corresponden con los valores booleanos. Mientras que X se corresponde con una variable sin valor (desconocido), Z es un estado especial utilizado para codificar circuitos que no están conectados en alguno de sus extremos.

Además del tipo de datos básico, System Verilog permite el uso de otros tipos de datos cómo constantes o parametros, enums (variables que tienen fijados cuales pueden ser sus valores y estos tienen nombre) y tipos de datos estructurados (agrupación do otros tipos de datos). A parte de estos tipos de datos, existen otros como el integer (32 bits con signo) o el real (32 bits de coma flotante con signo), pero no los utilizaremos.

System Verilog permite la definición de tipos de datos personalizados. En general, se recomienda su uso siempre que sea posible, ya que permite que las herramientas detecten asignación entre tipos de datos diferentes y notifique con un aviso ('Warning') al programador.

```
/*
 *  Los comentarios de código tienen el mismo formato que en C
 */
logic        un_bit;  // Declaración de un wire/registro de un bit
logic [7:0]  un_byte; // Declaración de un wire/registro de 8 bits
logic [63:0] 64_bits; // Declaración de 64 bits
logic [7:0]  ocho_bytes [7:0]; // Declaración de 8 bytes (64 bits)

// Declaración de parametros/constantes
parameter   CERO      = 64'h0; // Constante cero
parameter   WordWidth = 63;

// Declaración de un tipo de datos de 64 bits usando una variable
typedef logic [WordWidth:0] bus64_t;

bus64_t     un_bus;           // Instanciación de un bus de 64b
bus64_t     ocho_buses [7:0]; // Instanciación de ocho buses

/* Tipo de datos de enumerado de dos bits
 * Usaremos este tipo de datos para codificar el signo de una
```

```
 * fracción y si hay un error
typedef enum logic [1:0] {
    POSITIVO   = 2'b00,
    NEGATIVO   = 2'b01,
    ERROR      = 2'b10 // Codifica division por cero
} control_bits_t;

// Tipo de datos para una fracción
typedef struct packed {
    bus64_t         numerador:       // Tipo de datos custom
    bus64_t         denominador;
    control_bits_t  control_signo;   // Tipo enum
} fraccion;
```

2.2. Representación Numérica

Los lenguajes HDL permiten representar valores numéricos y constantes utilizando diferentes codificaciones: binario, octal, hexadecimal, decimal, coma flotante, etc.

En SystemVerilog por defecto los valores están escritos en código decimal. El siguiente código resume los principales formatos y su codificación. Por regla general un valor numérico vendrá precedido por dos campos de formato. El primer cammpo codifica la longitud en bits del valor. Después el carácter ' cómo separador. El segundo campo especifica cómo ha de interpretarse el valor numérico (binario, octal, hexadecimal, etc.). El primer campo puede omitirse para algunas expresiones como el 0.

```
//  Los siguientes valores numéricos son equivalentes
logic cero_binario = 1'b0; // 'b codifica valores binarios
logic cero_hex     = 1'h0; // 'h codifica valores hexadecimales
logic cero_octal   = 1'o0; // 'o codifica valores octales
logic cero_decimal = 0;    // Equivalente a 1'd0

// Para variables de más de un bit se especifica el tamaño
logic [7:0] un_byte = 8'hff // byte vale ahora 8'b11111111
logic [4:0] cero    = 'h0   // Equvalente a 5'h0 o 5'b00000
```

2.3. Operadores

Los lenguajes HDL incluyen multitud de operadores para modelar las diferentes funciones lógicas a implementar. El principal operador es la asignación, que puede tener dos tipos de comportamiento para lógica combinacional o lógica secuencial (Para entender la diferencia véase dicho apartado). Además de la asignación, existen muchos otros operadores que se centran en la lógica booleana dependiendo del lenguaje HDL.

*En SystemVerilog hay cuatro tipos de operadores: la asignación, los operadores unarios (negación: NOT), operadores binarios (funciones lógicas clásicas: OR, AND, etc) y operadores ternarios (asignación condicional). La asignación combinacional es la más sencilla, la variable asignada toma inmediatamente (las puertas lógicas pueden tener retardo pero la asignación no) los valores asignados. Esta asignación se realiza escribiendo la sentencia '**assign**' seguida de la variable a asignar, de el carácter igual y de la expresión que produce el valor asignado. Y los operadores binarios se colocan en una expresión entre los dos operandos.*

La siguiente tabla resume las principales operaciones lógicas bit a bit de System Verilog.

Operador	Descripción
$\sim$	NOT
&	AND
|	OR
$\wedge$	XOR
$\sim$ &	NAND
$\sim$ |	NOR
$\sim \wedge$	XNOR

Cuadro 1: Tabla de operadores binarios bit a bit en SystemVeriliog.

Adicionalmente, SystemVerilog soporta operadores aritméticos cómo la suma (+) y la resta (-). Y los operadores relacionales tradicionales de C (mayor que '>', igual '==', etc). Y otros tipos de operadores más complejos que ***no usaremos*** *cómo la multiplicación, la división y el resto. Sin embargo, estos últimos son más complejos y requieren que las herramientas de síntesis dispongan de librerías que puedan realizar estas operaciones complejas.*

Los operadores de desplazamiento desplazan el contenido de una variable hacia la derecha o izquierda un número determinado de posiciones. Son operaciones de desplazamiento lógico, no aritmético, ya que las posiciones de los bits desplazados se rellenan con 0 (no extiende el signo en desplazamientos a la derecha). Los bits desplazados se pierden. Los operadores son '$\ll$' hacia la izquierda y '$\gg$' hacia la derecha.

*Finalmente, los operadores ternarios condicionales permiten la selección de una de dos señales basando la decisión en el valor de una tercera. Es decir, nos permite realizar una selección if-else o un mux de dos puertos, utilizando unos pocos caracteres. La expresión ternaria está compuesta por la condición seguida de un interrogante ('**?**'), seguido de la primera opción, dos puntos y la segunda opción. El siguiente código resume los operadores básicos:*

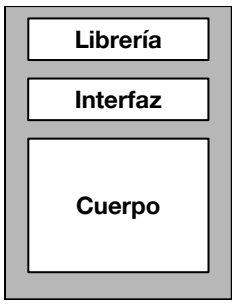
```
// La asignación comibnacional es el operador más frecuente.
assign cero_binario = 1'b0;

// Operadores binarios bit a bit
assign uno_binario = (1'b1 | cero_binario) & 1'b1;

// Operadores unarios
assign otro_cero = ~(uno_binario);

// Operador ternario condicional
assign otro_uno = (otro_cero) ? cero_binario : uno_binario;

/*** Operadores binarios aritmeticos ***/
logic [1:0] dos_bits, otros_dos; // Dos señales de dos bits

// Sumamos 1'b1 + 1'b1 = 2'b10
assign dos_bits = uno_binario + otro_uno;

// Restamos 1'b0 - 1'b1 = 2'b11 con overflow
assign otros_dos = cero_binario - otro_uno;

// Desplazamiento a la izquierda una posición
assign ultimo_cero = dos_bits << 1'b1; // Equivalente a << 1
```

2.4. Estructura básica de un módulo

Al igual que en C++ o Java, la unidad básica de programación es la clase, en HDL la unidad básica de código es el módulo. Por norma general se utiliza para cada modulo un único fichero de HDL. Los módulos, cómo las clases, tienen dos partes: una interfaz que exponen al exterior y la arquitectura o comportamiento del módulo. La interfaz establece cómo se observa el sistema digital desde el exterior. Por arquitectura entenderemos la descripción del circuito en lógica combinacional y secuencial para obtener el comportamiento deseado.

Esta división entre interfaz e implementación permite un modelo de abstracción en el que el usuario de un módulo sólo necesita tener en cuenta la especificación e interfaz de un módulo para utilizarlo. Sin necesitar entender la implementación exacta de éste. Permitiendo así el trabajo en equipo y la reutilización de código.

Para poder reutilizar otros módulos ya existentes, los lenguajes HDL permiten importar liberáis o módulos declarándolos al principio del modulo.



A continuación profundizaremos en cada uno de los elementos que componen un módulo y cómo instanciarlo.

2.4.1. Paquetes

En HDL un *package* (paquete o libreria) es un fichero que contiene declaraciones de objetos, tipos de datos, entre otros, utilizados de forma usual y que se comparten entre diferentes módulos. También se pueden considerar *package* otros módulos escritos en HDL, dependiendo del lenguaje utilizado.

2.4.2. Interfaz

Todo módulo debe tener declarada una interfaz que permita identificar en primer lugar el nombre del módulo, para posteriormente poder ser instanciado. En segundo lugar, la interfaz define que señales de entrada y salida tiene el modulo. Estas señales pueden ser sencillas de un bit, o más complejas cómo los grupos de señales (entraremos en más detalle en la sección de SystemVerilog). Las señales de interfaz pueden ser de entrada, salida o entrada-salida. En general, sólo utilizaremos señales de tipo entrada o salida.

Además de estas dos funciones básicas, algunos HDL permiten la definición de parámetros. Los parámetros permiten a los usuarios de un módulo a instanciar este con diferentes características dependiendo de las necesidades del usuario. Por ejemplo, una cola puede tener parametrizadas su número de entradas. De forma que el usuario puede instanciar colas de diferente tamaño. Parametrizar los módulos es una buena practica que debemos intentar realizar siempre que sea posible.

*En **SystemVerilog** los módulos se declaran con la sentencia **module** seguida del nombre del módulo.*

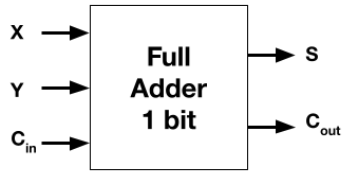


Figura 1: Interfaz de un sumador de 1 bit.

Después entre parentesis se enumeran entre comas cada uno de los puertos de entrada y salida

```
module FA (
  input  logic x,       // Operando 1
  input  logic y,       // Operando 2
  input  logic c_in,    // Carry entrada

  output logic s,       // Resultado
  output logic c_out    // Carry salida
);
```

2.4.3. Arquitectura

La parte de arquitectura de un módulo establece cómo deberá comportarse el módulo dependiendo de las diferentes entradas y el estado interno de este. Para ello, deberemos definir circuitos combinacionales y secuenciales que se ajusten al comportamiento deseado. Debido a la naturaleza paralela de los circuitos hardware, deberemos considerar que cada una de las cláusulas que declaremos funcionará en paralelo al resto. Por regla general, deberemos definir y separar bien estas dos partes de la arquitectura (combinacional y secuencial) para evitar introducir errores en el diseño final. Por ello, la mayoría de los HDL definen dos tipos de cláusulas diferentes para cada tipo de circuito. Ninguna señal debe estar asignada en dos cláusulas a la vez. A continuación se muestra un módulo con dos entradas y dos salidas, que internamente está implementado utilizando dos módulos instanciados en paralelo.

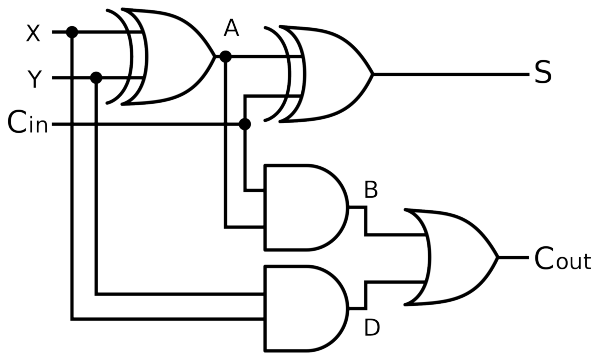


Figura 2: Esquema de puertas de la operación suma de 1 bit.

Además de circuitos combinacionales y secuenciales, también podemos instanciar otros módulos descritos en HDL. Al igual que el resto de cláusulas, un módulo instanciado funcionará en paralelo respecto al resto

de cláusulas del módulo. Aquellos módulos en los que únicamente se instancian otros submodulos y se interconectan se denominan módulos estructurales.

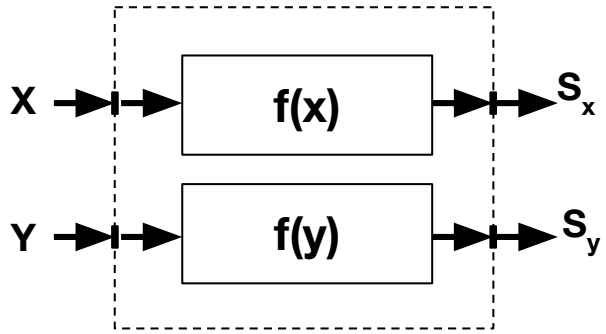


Figura 3: Ejemplo de módulo estructural implementado instanciado dos veces el módulo que implementa la función f().

2.4.4. Instanciación y conexión

Cómo ya hemos comentado en la sección anterior, los lenguajes HDL permiten la instanciación de módulos dentro de módulos. Ésta característica permite a los programadores a reusar modulos y la abstraerse de su implementación. Por ejemplo, dos equipos pueden dividirse el trabajo de implementación del módulo de ejemplo de la Figura 3. El primer equipo puede desarrollar el submódulo que implementa f() y el segundo la instanciación de este, abstrayéndose ambos equipos entre ellos. Únicamente es necesario especificar la interfaz de cada módulo para que la integración de los dos submodulos en el módulo final sea correcta.

*En **SystemVerilog** los módulos se instancian utilizando el nombre del módulo seguido del nombre de la instancia. A continuación se especifica en formato lista (elementos separados por coma) cada una de los puertos/señales de entrada o salida. Para cada puerto se utiliza el nombre del puerto original del módulo y seguido entre paréntesis el nombre de la señal que va conectada a ese puerto. Finalmente se completa la sentencia con un punto y coma.*

```
module parallel (
  input  logic x,
  input  logic y,

  output logic s_x,
  output logic s_y
);
// Instanciación del módulo 1
modulo modulo1_inst (
    .x(x),
    .s_x(s_x)
);
// Instanciación del módulo 2
modulo modulo2_inst (
    .x(y),
    .s_x(s_y)
);
endmodule
```

2.5. Logica Combinacional

La principal tarea del programador de HDL es describir circuitos a partir de una especificación. Para ello, el programador hace uso de dos tipos de circuitos: Combinacionales y Secuenciales. Como habréis visto en cursos anteriores, la principal diferencia entre estos dos circuitos es que el combinacional no tiene estado, mientras que el secuencial sí. Esto significa que los circuitos combinacionales determinan sus valores de salida únicamente en base a sus valores de entrada. Mientras tanto, los circuitos secuenciales sí tienen un estado interno que utilizan para generar tanto su salida como para actualizar su estado. Por lo tanto, el programador debe clasificar los circuitos en estas dos categorías. En esta práctica, únicamente utilizaremos lógica secuencial, pero en futuras prácticas utilizaremos ambas.

En SystemVerilog existen dos tipos de sentencias combinacionales: las asignaciones combinacionales y los bloques combinacionales. Como hemos visto anteriormente en el apartado de operadores, las asignaciones se realizan escribiendo la sentencia ***assign*** *seguida de la variable a asignar, de el carácter igual y de la expresión que produce el valor asignado.*

El otro tipo de sentencia que nos permite describir la lógica combinacional es el bloque combinacional. Éste no permitirá especificar circuitos lógicos más complejos que una simple asignación, y por lo tanto serán de gran utilidad. Sin embargo en ésta práctica no nos hará falta. El bloque combinacional se especifica utilizando la sentencia ***always_comb*** *y delimitando su especificación entre las sentencias* ***begin*** *y* ***end****. Dentro de un bloque combinacional realizaremos asignaciones combinacionales utilizando el operador =. A continuación se muestran dos ejemplos equivalentes utilizando los dos tipos de asignación*

```
// Asignación en una linea
assign c_out = (x & y) | (x & c_in) | (y & c_in);
assign s     = ((x ^ y) ^ c_in);

// Asignación en un bloque combinacional
always_comb begin
    c_out2 = (x & y) | (x & c_in) | (y & c_in);
    s2     = ((x ^ y) ^ c_in);
end
```

2.6. Modelado de Retardos

A pesar de que los circuitos digitales únicamente manejan valores binarios, la transición de 0 a 1 o de 1 a 0 tiene asociado un coste temporal. Cada puerta tendrá un coste temporal que llamaremos retardo y que viene determinado de la tecnología utilizada para fabricar el circuito. El correcto modelado de los retardos de un circuito es fundamental para poder cumplir con los objetivos de rendimiento de la especificación de un diseño.

Por regla general los retardos de cada puerta no se tienen en cuenta a la hora de diseñar un módulo. Los diseñadores especifican el circuito asumiendo que las puertas tienen un retardo instantáneo. En una fase posterior en la que se ha comprobado que el circuito cumple con las especificaciones de comportamiento, se procede ha realizar síntesis del circuito especificado. En esta fase se calculan los retardos de cada puerta para un nodo tecnológico en concreto. Se calcula cual es el camino crítico del circuito, es decir aquel camino que tiene el mayor retardo desde la entrada hasta la salida del circuito. Si el camino crítico es más grande que el retardo máximo especificado, el diseño no cumple con las especificaciones y deben realizarse modificaciones sobre el diseño original para que lo cumpla.

En esta práctica aprenderemos a especificar retardos para nuestras puertas lógicas. Esto nos permitirá

calcular los caminos críticos de nuestros circuitos. Sin embargo, a la hora de realizar un diseño real, esta tarea no corresponde al diseñador de HDL.

En System Verilog podemos especificar un retardo para un circuito. Para ello primero deberemos especificar que unidad de tiempo utilizaremos para medir los retardos, utilizando el comando 'timescale' seguido de la unidad de tiempo en la que mediremos los retardos y de la precisión con la que especificaremos estos (para realizar redondeo). Después para cada circuito que deseemos especificar un retardo deberemos añadir el símbolo de la almohadilla # seguido de un número real positivo antes de la asignación. Este número especifica la cantidad de retardo de la asignación en una unidad de tiempo que hemos especificado. El siguiente ejemplo muestra cómo realizar el retardo.

```
// Especificamos retardos de un nanosegundo (ns) sin decimales
‘timescale 1ns/1ns

// Asignación con un retardo de 1 ns
assign #(1) s = ((x ^ y) ^ c_in);

// Asignación con un retardo de 2 ns (se redondea a dos nanosegundos)
c_out2 #(1.6) and_xy = (x & y);
```

2.7. Verificación

Para evitar introducir errores en el comportamiento del circuito, es necesario realizar una verificación del comportamiento del módulo. Las empresas más importantes del sector tecnológico dedican la mayoría de recursos a esta tarea. Es por ello que este es el trabajo más común en la industria de los semiconductores. A lo largo del curso veremos diferentes métodos para la verificación de código HDL.

En esta sesión para verificar nuestra implementación del Full Adder utilizaremos un banco de prueba ('Testbench') que nos permitirá probar todos los posibles valores de entrada y comprobar así que la salida de nuestro módulo es correcta. Lo que se conoce como banco de prueba exhaustivo. Estos bancos de prueba verifican completamente nuestro módulo, es decir prueban todas las funcionalidades y valores posibles. Por ello, su utilización en modulos más complejos es muy costosa ya que para verificar un simple bus de 128 bits debe probar 2^{128} casos.

Además de bancos de pruebas, podemos verificar nuestro diseño colocando aserciones en nuestro código. Estas aserciones verifican en todo momento que un enunciado se cumple. Por ejemplo, si una señal llamada K de 4 bits sólo puede tomar valores del 0 (4'b0000) al 9 (4'b1001), podemos colocar un assert que especifique que K nunca tome valores superiores al 9.

```
// Especificamos retardos de un nanosegundo (ns) sin decimales
assert (K >= 1'h9);
```

3. Entorno de Trabajo

Para simplificar el entorno de trabajo os hemos provisto de una imagen de Docker donde podréis encontrar todas las herramientas necesarias para realizar la práctica. Si no disponeis de un ordenador con docker, en el apendice A encontrareis como utilizar una imagen de Virtual Box para realizar las prácticas.

Primero de todo tenemos que comprobar que el comando `docker` esta instalado en nuestra maquina. esto lo podemos comprobar rápidamente ejecutando el comando `which docker`. Si `docker` esta instalado en nuestro sistema, nos aparecerá en que ruta esta instalado, si no lo esta, el comando ejecutado no dará ninguna salida.

En el caso de no tenerlo instalado, debéis instalar `docker` usando:

```
$ sudo apt update
$ sudo apt install docker.io
```

Una vez asegurado que disponemos de `docker` podemos proceder a instalar e inicializar la imagen de ac2. Para ello, disponeis de script que simplifica el proceso, pero primero deberéis asegurar que vuestro usuario está en grupo docker para poder ejecutarlo.

```
$ usermod -a -G docker vuestro_usuario
```

Antes de utilizar el script debéis dar permisos de ejecución al script, y seguidamente podéis instalar y usar la imagen con los siguientes comandos:

```
$ chmod +x docker.sh
$ ./docker.sh build
$ ./docker.sh init
```

Como veréis, esta imagen docker contiene las siguientes herramientas que usaremos durante el curso de todo el laboratorio:

- Verilator 5.020
- RISC-V GNU toolchain
- GTKwave

3.1. Verilator

Verilator es un simulador de código abierto que nos permite realizar simulaciones del código RTL/HDL, sin tener que pagar una licencia. Verilator transforma nuestros módulos escritos en SystemVerilog o Verilog en ficheros de C++ que imitan la funcionalidad de los circuitos descritos en HDL. Estos ficheros en C++ se compilan para obtener un binario final que nos permite realizar simulaciones de los dichos circuitos.

Actualmente Verilator no solo és capaz de usar bancos de pruebas en C++ (como originalmente estaba diseñado), si no que ya es capaz de usar bancos de prueba en Verilog/SystemVerilog. Esto nos permite utilizar retardos en las señales de froma sencilla, funcionalidad necesaria para la realización de este laboratorio.

3.1.1. Compilación

Banco de pruebas en C++ Como se menciona, Verilator originalmente requería que un banco de pruebas de C++. Usando este método, no podemos incluir nuestro módulo en SystemVerilog en un banco de pruebas de C++ directamente: primero debemos utilizar Verilator para convertir el código SystemVerilog a C++ lo cual se hace de la siguiente forma:

```
$ verilator --cc modulo.sv
```

Ejecutar el comando anterior genera una nueva carpeta llamada *obj_dir* en nuestro directorio de trabajo. Esta carpeta contiene los archivos .h y .cpp, resultantes de la conversión desde SystemVerilog a C++. Además contiene los archivos .mk generados que se utilizarán con Make para construir nuestro ejecutable de simulación.

A diferencia de otros simuladores, como Modelsim, en los que se pueden abrir los archivos fuentes en la interfaz gráfica o se puede lanzar la simulación desde esta misma interfaz gráfica, Verilator no se utiliza directamente para simular el banco de pruebas. Verilator se utiliza únicamente para convertir Verilog a C++ y generar automáticamente el Makefile que nos permite compilar el ejecutable final. El entorno de simulación en este caso será el propio banco de pruebas en C++. Este banco de pruebas debe gestionar el comportamiento del simulador como la gestión de las señales de entrada y salida de los módulos, la gestión de la señal de reloj, etc. Por lo tanto el binario resultante de compilar el banco de pruebas en C++ y los fuentes HDL es esencialmente una aplicación en C++.

Para construir el ejecutable de simulación, necesitamos ejecutar Verilator nuevamente para generar los archivos .mk (Makefiles) e incluir el banco de pruebas en C++; esto se hace utilizando `--exe tb_modulo.cpp`. Donde el flag `--trace` habilita la generación de waveforms:

```
$ verilator -Wall --trace -cc modulo.sv --exe tb_modulo.cpp
```

Finalmente podemos crear el ejecutable de simulación usando:

```
$ make -C obj_dir -f Vmodulo.mk Vmodulo
```

Si el ejecutable del banco de pruebas se ha generado correctamente, encontraréis un binario llamado *Vmodulo* en la carpeta *obj_dir*.

Banco de pruebas en Verilog/SystemVerilog En este caso, tanto el modulo a simular como el mismo testbench están escritos en SystemVerilog y debemos transformarlos en C++ con el flag `--binary`. A diferencia del caso anterior podemos generar los ficheros de compilación (Makefile) y los ficheros traduccidos en un único paso. Ademas, en este punto podemos habilitar las aserciones con `--assert`, habilitar los delays con `--timing`, y seleccionar las unidades de tiempo usadas para la simulación con `--timescale-override 1ps/1ps`.

```
$ verilator --timing --timescale-override 1ps/1ps --trace --assert --
    binary modulo.sv tb_modulo.sv
$ make -C obj_dir -f Vmodulo.mk Vmodulo
```

3.1.2. Simulación

Una vez compilado el binario, simplemente podéis ejecutar el binario Vmodulo para iniciar la simulación:

```
$ ./obj_dir/Vmodulo
```

El ejecutable simulará el diseño realizando todas las pruebas dentro del banco de pruebas. En caso de tener alguna función de impresión (como printf o $display) dentro del código, el binario nos mostrará la salida de texto en la terminal. En caso de tener funcionalidades de verificación en nuestro diseño, como las aserciones, también nos mostrará si han fallado imprimiendo un mensaje en la terminal. Finalmente, al ejecutar la simulación, se generará un archivo en formato 'waveform' llamado 'waveform.vcd' en nuestro directorio de trabajo. Este archivo almacena el valor obtenido de cada cable y registro para cada instante de tiempo.

3.1.3. Visualización

Para abrir el archivo de wavefrom generado por el simulador utilizaremos GTKwave:

```
$ gtkwave waveform.vcd
```

Se te presentará una ventana de GTKWave que se verá así:

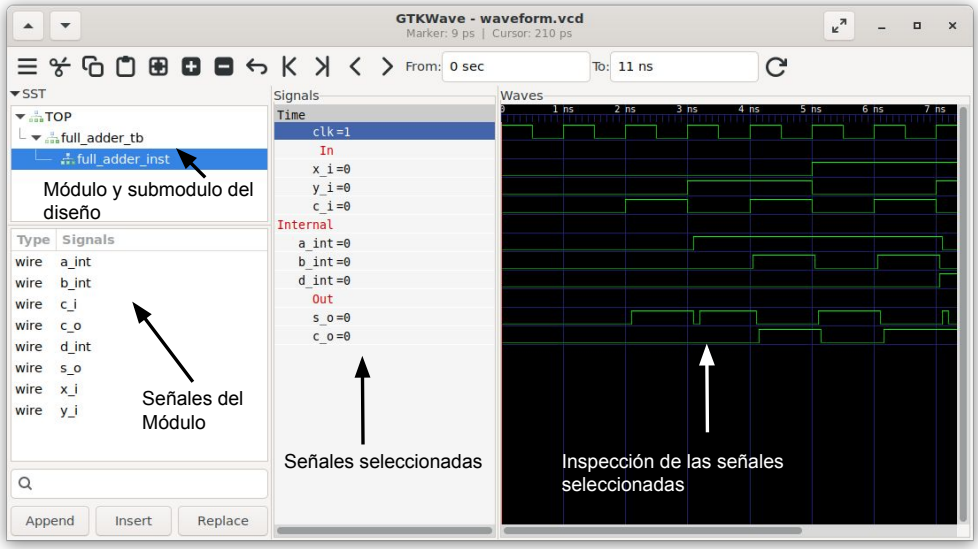


Figura 4: Diferentes secciones de la ventana de GTKwave

Finalmente comentar que gtkwave permite guardar la configuración en cualquier estado para poder rapidamente regenerar el estado de las señales añadidas. Para poder guardar la configuracion tenemos que seleccionar arriba a la izquierda `File > Write Save File` o pulsar '*Cntr+S*'. Para abrir una configuración guardada se puede abrir dentro de la aplicación seleccionando arriba a la izquierda `File > Read Save File` o pulsar '*Cntr+O*'. También se puede añadir el fichero de configuración en la linea de comandos al lanzar la aplicación:

```
$ gtkwave waveform.vcd gtkwave_config.gtkw
```

4. Sumador de 1 bit

La principal tarea de este laboratorio es implementar un modulo sumador de 1 bit o 'full adder' y analizar los retardos de la implementación. Recordemos que un sumador de 1 bit realiza la suma de 3 bits: operando 1 (x), operando 2 (y), acarreo (c_{in}). Un FA genera dos bits de salida: resultado de la suma (s) y el acarreo de salida (c_{out}). Podemos resumir la suma con la siguiente formula:

$$f(x, y, c_{out}) = (c_{out}, s)$$

Podemos generar la tabla de verdad 4 para el *full_adder* en función de las entradas x, y y c_{in}.

c_{in}	x	x	c_{out}	s
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Cuadro 2: Tabla de verdad para un sumador de 1 bit.

A partir de la tabla podemos obtener las funciones que determinan los valores de salida:

$$s = (x \oplus y) \oplus c_{in}$$

$$c_{out} = x \cdot y + c_{in} \cdot y + c_{in} \cdot x$$

Lo que plasmado en un circuito combinacional nos resultaría en el siguiente esquema:

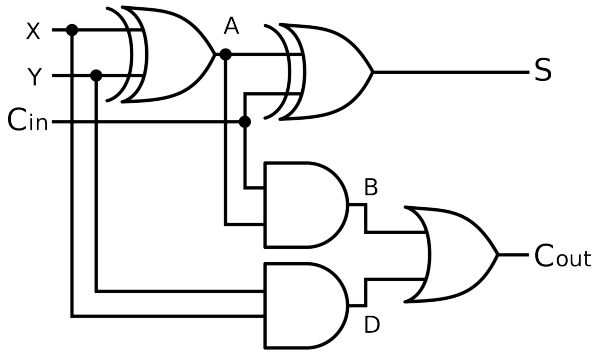


Figura 5: Esquema de puertas de la operación suma de 1 bit.

5. Trabajo a realizar

Antes de empezar con el laboratorio como tal tenéis que descargar e inicializar el entorno de trabajo. En este laboratorio, descargaréis tanto la imagen de Docker usada en AC2 (*ac2_environment.zip*) como los

archivos del laboratorio (*lab1.zip*). Primero, tenéis que descomprimir el entorno de trabajo. Dentro tenéis una imagen de Docker (Dockerfile), un script para compilar e inicializar la imagen Docker, y finalmente, una carpeta *labs* donde tendréis que poner las diversas carpetas de cada uno de los laboratorios. Tened en cuenta que esta carpeta *labs* está montada dentro del Docker; por lo tanto, cualquier archivo que esté dentro de esta carpeta se verá desde vuestro sistema operativo como desde dentro del Docker. Finalmente, ya podéis descargar e descomprimir los archivos del laboratorio 1 dentro de la carpeta *labs*.

Por ultimo antes de empezar, tenéis que instalar e inicializar la imagen docker. Para esto tenéis que seguir los pasos de la sección 3. Una vez inicializado el docker en un terminal, deberéis ejecutar todos los comandos descritos en esta practica en este, ya que estaréis usando todas las herramientas instaladas en la imagen de docker. Ahora podéis empezar las tareas de este laboratorio:

1) Implementad un full adder. Para esto, tenéis que crear un módulo llamado '*full_adder.sv*' y completar el siguiente código:

```
module full_adder (
    input logic x_i,
    input logic y_i,
    input logic c_i,

    output logic s_o,
    output logic c_o
);

// Declaración de señyales intermedias

// Aquitectura de un Full Adder

endmodule
```

2) Seguidamente, analizad la implementación del banco de pruebas proporcionado en esta practica (*tb_full_adder.sv*). El banco de prueba esta dividido en 4 partes principales: declaración de variables, instanciación del diseño bajo test, definición de los diferentes test en forma de *task*, y la ejecución principal del banco de pruebas. Analizad que hacen los dos test preparados (basic_functional_testing y transition_test).

3) Compilad y comprobad la funcionalidad de vuestro full adder. Para facilitar la compilación os proporcionamos un Makefile con las siguientes recetas:

- build: usando verilator, genera el simulador del modulo full adder (*full_adder.sv*) bajo el banco de pruebas *tb_full_adder.sv*
- sim: Ejecuta el simulador, mostrando por terminal los diferentes aserciones de verificación colocadas en el banco de prueba. Tambien genera el *waveform.vcd* con las diferentes formas de onda de las diferentes señales internas del modulo
- view: Abre el GTKWave cargando el fichero *waveform.vcd* y la configuración *gtkwave_setup.gtkw*

4) Añadid retardos en vuestra implementación del full adder tal como se explica en la sección 2.6. Tened en cuenta que en los valores de retardo en esta practica van a ser en pico-segundos (ps). En este caso los retados de las puertas son:

- AND = 50 ps; OR = 100 ps; XOR = 100 ps

5) Finalmente, con ayuda del GTKwave (`make view`), verificar los retardos de la salida para las diferentes transiciones de input pedidas en las preguntas de la practica. Para poder verificar el tiempo de retardo para las diferentes transiciones de las señales de entrada, podeis usar la *task* (*transition_test*) proporcionada en el banco de pruebas. Esta función realiza la transición entre dos entradas cualesquiera que podéis configurar en el propio banco de pruebas y deja pasar un tiempo de ciclo de 500 picosegundos entre cambios de entradas. Para medir el retardo entre dos entradas deberéis observar el instante en el que se realiza la primera transición (valor entre parentesis en el mensaje FROM) dentro del waveform hasta que los valores de salida del modulo sean estables.

5.1. Preguntas a entregar

Responded las siguientes preguntas imprimiendo esta hoja, la cual se deberá entregar junto con los ficheros implementados durante la practica.

1) Ejecuta el test que prueba las 8 posibles combinaciones de los valores de las señales de entrada de un sumador de 1 bit (*basic_functional_testing*). Adjunta una captura de la ventana GTKwave con las señales de entrada y salida.

2) Para cada vector de entrada posibles, indica el retardo observado cuando el vector d'entrada anterior era el x=0, y=0, c=0. Justifica los resultados observados.

entrada	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
retardo								

3) Indica los valores de un vector de entrada E1 y un vector de entrada previo E0 que provoquen el máximo retardo posible, conocido como 'critical path'. Adjunta una captura de pantalla de GTKWave donde se vean los vectores de entrada E1 y E0 junto con la salida para identificar el retardo. Justifica tu respuesta.

4) Haz lo mismo para encontrar dos vectores de entrada E1 y E0 que minimicen el retardo (E1 y E0 no pueden ser iguales). Adjunta una captura de pantalla de GTKWave donde se vean los vectores de entrada E1 y E0 juntamente con la salida para identificar el retardo. Justifica tu respuesta.

Entregad la memoria de la práctica (nombrado *respuestas.pdf*), el archivo *full_adder.sv* y el banco de pruebas donde se prueban los retardos máximos y minimos *tb_full_adder.sv* en racó sin comprimir.

6. Apéndice 1: Imagen de Ubuntu para máquina virtual

Dado que los ordenadores de los laboratórios no disponen de docker os proporcionaremos una imagen virtual 'AC2.ova' que podréis importar en Vritual Box. Esta imagen ya dispone del software necesario para realizar la práctica. Sin embargo se recomienda el uso de la herramienta docker ya que es mucho más ligera y rápixa que virtual box.

La imagen de Virtual Box tiene un único usuario ac2 con password ac2. Para realizar las prácticas poder descargar y descomprimir desde la propia imagen los tar de los laboratorios.

7. Apéndice 2: Recursos para aprender SystemVerilog

Existen multitud de cursos y recursos para aprender a describir circuitos en SystemVerilog. En este apéndice hemos recogido una serie de paginas webs donde se puede consultar información sobre la sintaxis de SystemVerilog, la mayoria de ellas tiene ejemplos y consejos.

ChipVerify es una página web que documenta muchos lenguajes de HDL:

- SystemVerilog: `https://www.chipverify.com/tutorials/systemverilog`
- Verilog: `https://www.chipverify.com/tutorials/verilog`