Manual GHDL + GTKWave

Febrero 2022

1. Introducción

Este manual tiene la intención de educar al usuario en el uso de GHDL y GTKWave como herramientas de desarrollo de hardware utilizando VHSIC Hardware Description Language (VHDL). Se busca que el usuario aprenda a utilizar GHDL para compilar y simular código de VHDL, y que pueda visualizar los datos de sus simulaciones utilizando GTKWave, haciendo uso de las diferentes herramientas de navegación para analizar los resultados. El entorno utilizado para el desarrollo de este manual fue de sistemas operativos basados en Linux, sin embargo, excluyendo la instalación, las instrucciones deberán funcionar de forma uniforme independientemente de la plataforma del usuario.

2. Instalación

Para instalar los paquetes de software de GHDL y GTKWave, se debe utilizar el gestor de paquetes de la distribución que se tenga así como verificar si el repositorio del gestor cuenta con los paquetes de GHDL y GTKWave. En caso contrario, se puede entrar al sitio web de GTKWave o GHDL para obtener binarios pre-compilados o código fuente para compilar los programas. En el caso de Arch Linux,Debian y Ubuntu, existen paquetes en los repositorios oficiales por lo que, en el caso del primero, se utiliza el gestor de paquetes *pacman* como se muestra a continuación:

```
# pacman -S gtkwave ghdl
```

En el caso de los dos últimos, se utiliza el gestor *apt* de la siguiente forma:

```
# apt install gtkwave ghdl
```

En ambos casos, al momento de instalar GHDL se deberá elegir un proveedor de herramientas para compilar el código. Existen tres opciones:

- GCC
- LLVM
- mcode

3. Instrucciones de Uso

Existen tres pasos para realizar una simulación con GHDL: análisis, elaboración y simulación, estos tres pasos se llevan a cabo utilizando diferentes comandos de GHDL. A continuación, se describen brevemente cada una de estas etapas, así como algunas opciones útiles.

3.1. Análisis

Durante esta etapa el compilador revisa cada unidad de código y la procesa para elaborar archivos objeto (*object files*) para su uso en las siguientes etapas. En este paso se realiza el análisis semántico y sintáctico del código. La ejecución de este paso se lleva a cabo con el siguiente comando:

```
$ ghdl -a [opciones] archivos
```

Los argumentos de este comando consisten de una lista de archivos (`archivos`) cuyo orden puede ser relevante dependiendo de la organización del código.

3.2. Elaboración

En este paso se vuelve a realizar el análisis de todas las unidades de código para generar archivos objecto que son vinculados para producir el ejecutable que será utilizado para llevar a cabo la simulación. El comando para este paso es el siguiente:

```
$ ghdl -e [opciones] entidad
```

Donde `entidad` es la entidad principal del diseño.

3.3. Opciones para análisis y elaboración

Las siguientes opciones aplican para los pasos de análisis y elaboración:

-std=<STANDARD> Permite elegir el estándar de VHDL con el que se lleva a cabo la compilación. Por defecto, el estándar es VHDL-93 (93c)

-work=<DIR> Especifica la dirección del directorio (DIR) donde se encuentra el código perteneciente a la biblioteca *work*. En caso de no especificarlo, se toma por defecto la carpeta en la que se invoque el comando

-p=<DIR> Especifica la dirección del directorio (DIR) donde se encuentra el código perteneciente a otras bibliotecas externas que sean requeridas

-fsynopsys Sirve para utilizar los paquetes `std_logic_arith,std_logic_signed,std_logic_unsigned,std_logic_textio`, que no son parte, de forma oficial, de las librerías estándar IEEE.

3.4. Simulación

Este es el último paso donde se lleva a cabo la ejecución o la simulación del código. Este es invocado de la siguiente forma:

```
$ ghdl -r [opciones] entidad [opciones de simulación]
```

Este comando acepta un primer bloque de opciones, que son las mismas utilizadas en el análisis y la elaboración, el segundo bloque contiene opciones específicas para la simulación que son descritas más adelante. Finalmente, la entidad que se pasa como argumento es aquella que se busca simular.

3.4.1. Opciones

Algunas opciones útiles para la simulación son descritas a continuación.

--disp-time Muestra el tiempo y los ciclos delta (*delta cycles*) que lleva la simulación mientras se ejecuta

--stop-delta=<N> Detiene la simulación después de *N* ciclos delta. El valor por defecto para esta opción es de 5000 ciclos

--stop-time=<T> Detiene la simulación después de un intervalo de tiempo *T*, donde *T* esta expresado en unidades de una variable *time* de VHDL. Cabe resaltar que este tiempo es relativo a la simulación y no al reloj interno del diseño.

--wave=<FILE> Guardar las salidas de las señales en un archivo con dirección *FILE* en formato GHDL Waveform (GHW)

--vcd=<FILE>/-vcdgz=<FILE> Guardar las salidas de las señales en un archivo con dirección FILE con formato VCD. La opción *-vcdgz* tiene la característica adicional que comprime los datos de las señales con gzip.

El uso de GTKWave para visualizar las simulaciones se describe utilizando los siguientes ejemplos.

4. Ejemplo 1: Simulación simple

En este ejemplo se lleva a cabo la simulación de una arquitectura que describe una señal de reloj mediante un proceso. Este diseño no tiene ninguna entrada y solo una salida (*CLK*). A continuación, se muestra el código de esta entidad:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Clock is
    port(CLK :   out STD_LOGIC);
end Clock;

architecture Behavioral of Clock is
    constant P : time := 200 ns;
begin
    process begin
        CLK <= '0';
        wait for P/2;
        CLK <= '1';
        wait for P/2;
    end process;
end Behavioral;
```

clock.vhd

El paso de análisis se lleva a cabo con el siguiente comando:

```
$ ghdl -a clock.vhd
```

Posteriormente, la elaboración se lleva a cabo con el siguiente comando:

```
$ ghdl -e clock
```

Es importante resaltar que para este paso, el argumento del comando es el nombre de la entidad (*Clock*) y no el nombre del archivo, además, se tiene que considerar que VHDL no es un lenguaje que distinga entre letras mayúsculas y minúsculas, por lo que tampoco importa que el nombre de la entidad sea idéntico al del argumento del comando, al menos en este aspecto.

Finalmente, para realizar la simulación se debe ejecutar el siguiente comando:

```
$ ghdl -r clock --wave=clock_sim.ghw
```

En este caso, el archivo de salida para las señales es *clock_sim.ghw*. Con estas opciones, la operación de simulación deberá ser detenida manualmente, utilizando CTRL+C. Otra forma de realizar esta operación es utilizando la opción *--stop-time* como se muestra a continuación:

```
$ ghdl -r clock --wave=clock_sim.ghw --stop-time=5us
```

De esta forma, la simulación termina de forma automática, después de 5µs.

Para analizar la simulación con GTKWave, se puede invocar al programa desde la consola utilizando como argumento, el archivo de salida con las señales de la simulación (en este caso `clock_sim.gwh`), como se muestra en el siguiente comando:

```
$ gtkwave clock_sim.ghw
```

O bien, arrancar directamente el programa y abrir desde ahí el archivo de las señale con la opción *File >Open New Tab*. Una vez que GTKView ha arrancado, se muestra la pantalla principal (Figura 1).

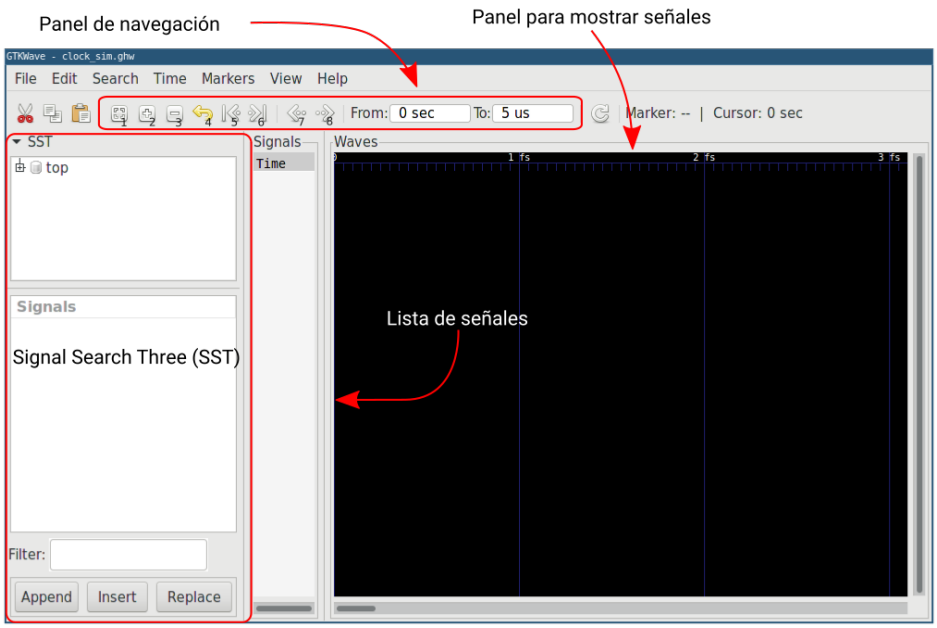


Figura 1: Pantalla principal de GTKWave

Algunos de sus elementos más importantes se describen en seguida.

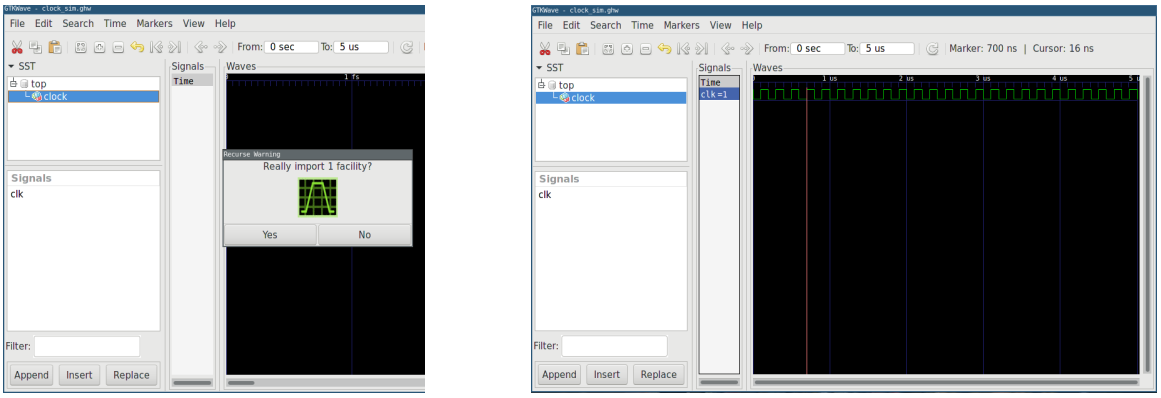
Signal Search Tree (SST) Es un menú acomodado de forma jerárquica de acuerdo con el diseño de la entidad. Al seleccionar un elemento del menú, sus señales aparecerán en el apartado inferior con la etiqueta *Signals*. Este panel puede ser usado para arrastrar las señales que se quieran visualizar, arrastrándolas hacía la lista que se encuentra la derecha (*Signals*). Otra forma de hacer esto es dar clic derecho en la entidad o la señal y seleccionar la opción *Recurse Import* y elegir si se quiere agregar al final (*append*), al inicio (*insert*) o reemplazarla(*replace*) en la lista de señales de la derecha.

Signals Es una lista que contiene todas las señales que se muestran, así como su valor en el instante en que se encuentre parado el cursor. Al hacer clic derecho en los elementos contenidos en esta lista se pueden hacer ajustes para la visualización de las señales como cambiar el color de la onda o seleccionar la representación con la que muestran sus valores (binario, decimal, hexadecimal, etc.)

Waves Es el panel donde se muestran los datos de la simulación en forma de ondas. Dentro de este espacio se puede utilizar la funcionalidad de los cursores primario y secundario, ajustando cada uno con el clic izquierdo y el clic intermedio del ratón, respectivamente, para analizar las señales.

Panel de navegación Este panel contiene botones para moverse a lo largo de las señales de la simulación. El primer botón (1) sirve para ajustar la escala de la señal al tamaño de la ventana, mientras que los siguientes dos sirven para aumentar(2) o reducir(3) la escala, el siguiente botón(4) sirve para deshacer la última modificación a la escala que se haya hecho mediante estos botones. Los siguientes dos botones sirven para recorrer el área que se muestra de las señales, al final (5) o al inicio(6) de la simulación. Finalmente, los últimos dos botones sirven para moverse entre el flanco anterior (7) o el siguiente(8) de cada señal. Los campos *From* y *To* sirven para delimitar el segmento de tiempo de la simulación que es mostrado.

Una vez que se ha abierto el archivo, se tienen que elegir las señales que se quieren mostrar, para este ejemplo se agregan las señales de toda la entidad desde el SST (Figura 2a).



(a) Agregando señales

(b) Simulación de la entidad

Figura 2: Simulación de la entidad Clock

Finalmente, en la Figura 2b podemos observar la señal del reloj de 5 kHz (T = 200 ns), simulada por 5 µs.

5. Ejemplo 2: Simulación con paquetes

En un diseño más complejo es de utilidad dividir la funcionalidad en diferentes paquetes encargados de realizar tareas específicas, dentro de estos paquetes se pueden definir constantes, componentes y funciones. Para el siguiente ejemplo se simulará una Unidad Aritmética-Lógica (ALU) diseñada para trabajar con *N* bits, esta unidad está diseñada a partir de un componente básico que es la ALU de 1 bit, que a su vez depende del componente sumador de 1 bit (el código de cada entidad se encuentra en el apéndice Código de pruebas). La jerarquía de componentes se muestra en la Figura 3:

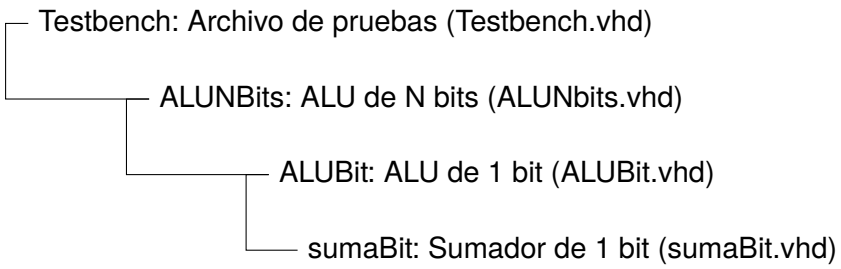


Figura 3: Jerarquía de componentes para la ALU de *N* bits

Esta jerarquía es importante al momento invocar los comandos de análisis, elaboración y simulación. Dado que las definiciones del sumador de un bit y la ALU de un bit se encuentran dentro del paquete *ALUPackage*, este se tiene que colocar al principio de la lista, seguido de los archivos que describen estos componentes (ALU y sumador), para que al momento que se analice la entidad *ALUNBits*, el compilador ya tenga registrados esos componentes y lo mismo suceda al analizar el testbench. La invocación del comando para analizar es la siguiente:

```
$ ghdl -a ALUPackage.vhd sumaBit.vhd ALUBit.vhd ALUNbits.vhd Testbench.vhd
```

La entidad que es pasada como argumento debe de ser la de mayor jerarquía por lo que utilizamos *ALUNBits* como se muestra a continuación:

```
$ ghdl -e ALUNbits
```

Una vez que se ha elaborado la entidad que va a ser simulada, se elabora el testbench:

```
$ ghdl -e Testbench.
```

Finalmente, se invoca al comando de simulación con la opción *--wave* para exportar los datos de las señales a un archivo *GHW*. Como el proceso de este testbench acaba con una instrucción *wait* con tiempo indeterminado, la simulación sale de la ejecución de forma automática, por lo que no se requiere la opción *--stop-time* ni la salida manual del programa.

```
$ ghdl -r testbench --wave=alun_tb.ghw
```

Posteriormente abrimos el archivo *alun_tb.ghw* con GTKWave, al expandir el árbol del SST podemos observar que aparecen las señales del testbench así como todas las señales internas de la entidad *ALUNbits* (Figura 4). Elegimos una mezcla de señales internas de la entidad *alu* como las banderas *overflow* (OV), cero(Z), *carry*(C), negativo (N), los buses de las dos entradas *a* y *b* y la salida de datos *s*. En la Figura 5 se pueden ver todas estas señales ya agregadas a la lista de señales. Después de haber agregado estas señales, podemos utilizar las opciones del menú de cada señal (Figura 6) para cambiar la representación en la que se muestran los valores de las señales por algo más útil como decimal con signo, que nos permite apreciar los resultados de operaciones aritméticas. Como se puede ver en la Figura 7, en los primeros 10 ns se hace una suma, mientras que en los siguientes 10 ns se hace la resta.

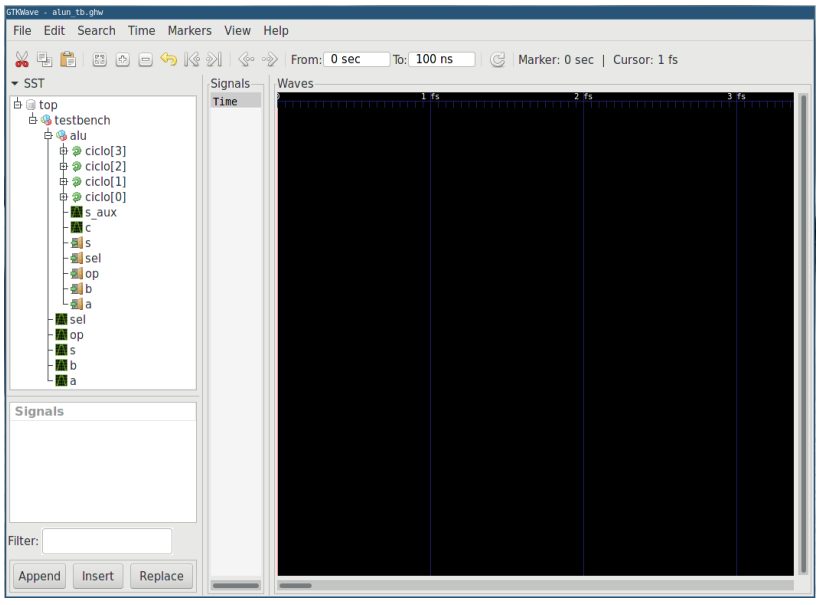


Figura 4: GTKWave con las señales cargadas desde el archivo *alun_tb.ghw* mostradas en el panel del SST

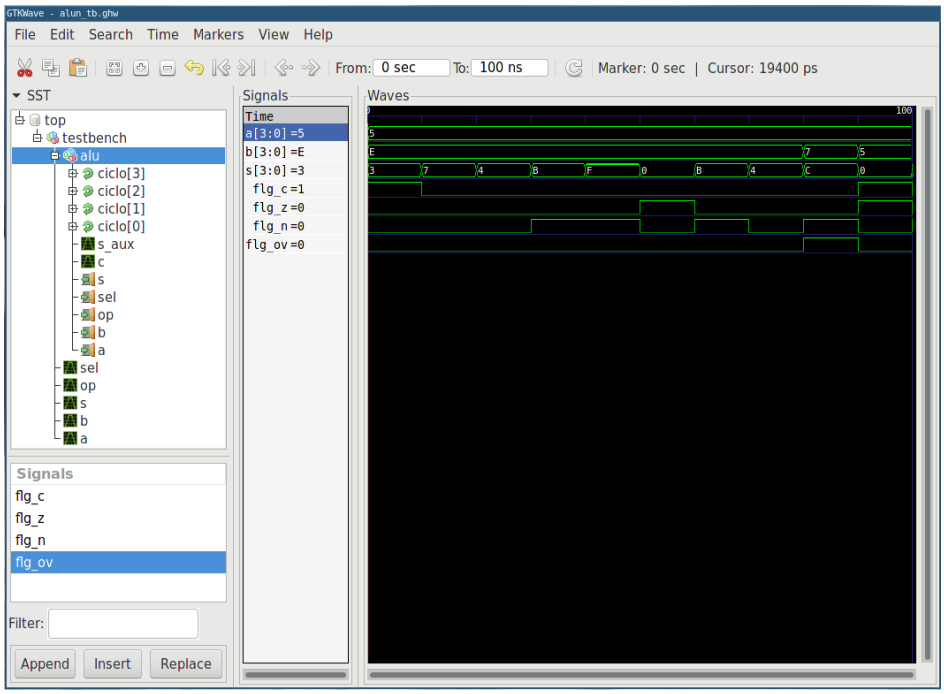


Figura 5: Señales seleccionadas de la ALU en el panel de visualización de señales

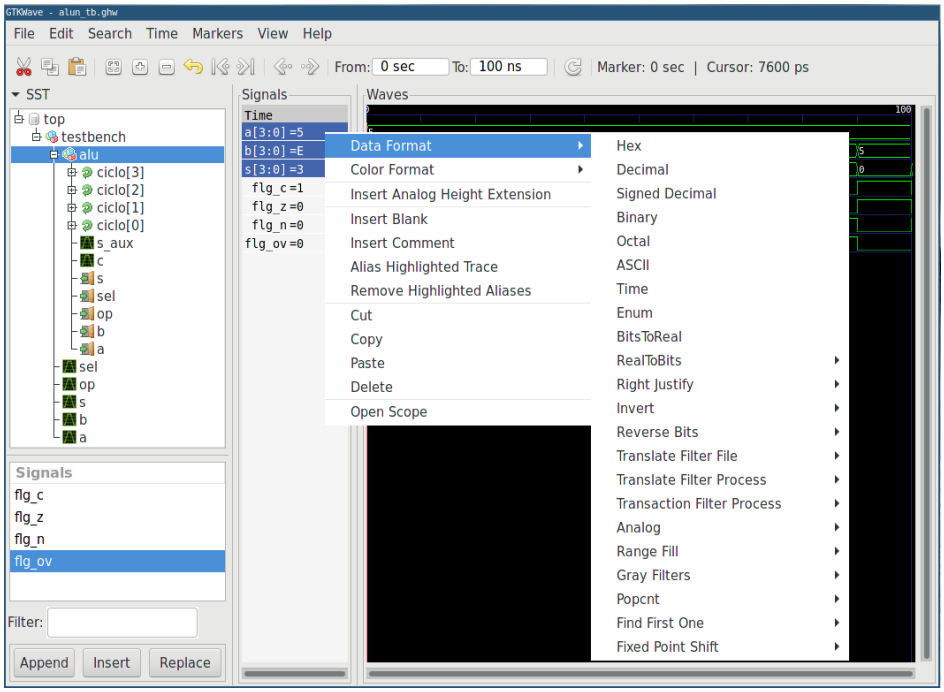


Figura 6: Menú para cambiar la representación de los valores de las señales, se pueden ver las diferentes opciones como binario, decimal, hexadecimal, octal, etc. Se elige la representación

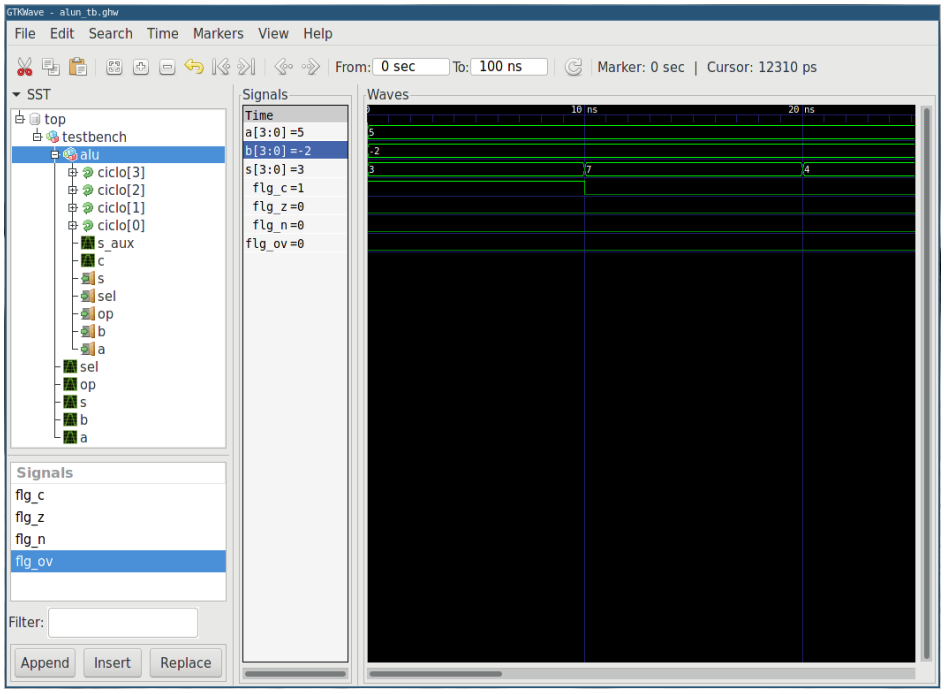


Figura 7: Primeros 20 ns de la simulación donde se hace una suma ($s = a + b$ donde $a = 5$,$b = -2$) y una resta ($s = a - b$)

6. Lectura adicional

Hasta ahora hemos explorado las funciones más básicas de GHDL y GTKWave, sin embargo, existen otras funciones que también pueden ser útiles para el usuario en el desarrollo de diseños digitales, todas estas funciones se encuentran documentadas en el manual de GTKWave y la documentación de GHDL, cuya lectura es recomendada.

A. Código de pruebas

A.1. Sumador de 1 bit (*sumaBit.vhd*)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity sumaBit is
    Port ( a,b,cin : in STD_LOGIC;
           s,cout : out STD_LOGIC);
end sumaBit;

architecture Behavioral of sumaBit is
begin
    s <= a xor b xor cin;
    cout <= (a and cin) or (b and cin) or (b and a);
end Behavioral;
```

A.2. ALU de 1 bit (*ALUBit.vhd*)

```
library IEEE;
library work;
use IEEE.STD_LOGIC_1164.ALL;
use work.ALUPackage.SumaBit;

entity ALUBit is
    Port ( a,b,cin : in STD_LOGIC;
    op,sel : in STD_LOGIC_VECTOR(1 downto 0);
    s,cout : out STD_LOGIC);
end ALUBit;

--              Tabla de control
--
--   SEL(1) | SEL(0) | OP(1) | OP(0) | Operación
--  --------|--------|-------|-------|------------------
--     0    |    0   |   0   |   0   | AND
--     0    |    0   |   0   |   1   | OR
--     0    |    0   |   1   |   0   | XOR
--     1    |    1   |   0   |   1   | NAND
--     1    |    1   |   0   |   0   | NOR
--     0    |    1   |   1   |   0   | XNOR
--     1    |    1   |   0   |   1   | NOT (NAND)
--     1    |    1   |   0   |   0   | NOT (NOR)
--     0    |    0   |   1   |   1   | SUMA
```

```
--     0    |    1    |    1    |    1    | RESTA
--
architecture Behavioral of ALUBit is

    signal aux_a, aux_b, and1, or1, xor1, sum1, carry : STD_LOGIC;

begin
    aux_a <= a when ( sel(1) = '0') else not a;
    aux_b <= b when ( sel(0) = '0') else not b;
    and1 <= aux_a and aux_b;
    or1 <= aux_a or aux_b;
    xor1 <= aux_a xor aux_b;
    suma : sumaBit port map (
                                a => aux_a,
                                b => aux_b,
                                cin => cin,
                                s => sum1,
                                cout => carry);
    process(op, and1, sum1, or1, xor1)
    begin
        case op is
            when "00" =>
                cout <= '0';
                s <= and1;
            when "01" =>
                cout <= '0';
                s <= or1;
            when "10" =>
                cout <= '0';
                s <= xor1;
            when others =>
                cout <= carry;
                s <= sum1;
        end case;
    end process;
end Behavioral;
```

A.3. ALU de N bits (*ALUNbits.vhd*)

```
library IEEE;
library work;
use IEEE.STD_LOGIC_1164.ALL;
use work.ALUPackage.ALL;

entity ALUNbits is
```

```
    generic (N : integer := 4);
    Port ( a,b : in STD_LOGIC_VECTOR (N - 1 downto 0);
    op,sel : in STD_LOGIC_VECTOR (1 downto 0);
    s : out STD_LOGIC_VECTOR (N - 1 downto 0);
    flg_ov, flg_n, flg_z, flg_c : out STD_LOGIC);
end ALUNbits;

architecture Behavioral of ALUNbits is

    signal c : STD_LOGIC_VECTOR(N downto 0);
    signal s_aux : STD_LOGIC_VECTOR(N - 1 downto 0);

begin
    c(0) <= sel(0);
    s <= s_aux;
    ciclo : for i in 0 to N - 1 generate
        alun : ALUBit port map(
                                    a => a(i),
                                    b => b(i),
                                    sel => sel,
                                    cin => c(i),
                                    op => op,
                                    s => s_aux(i),
                                    cout => c(i + 1));
    end generate;
    -- Bandera z
    process(s_aux, a, b, sel, op)
        variable aux : STD_LOGIC;
    begin
        aux := '0';
        zfor : for i in 0 to N - 1 loop
            aux := aux or s_aux(i);
        end loop;
        flg_z <= not aux;
    end process;
    -- Banderas
    flg_ov <= c(N - 1) xor c(N);
    flg_n <= s_aux(N - 1);
    flg_c <= c(N);
end Behavioral;
```

A.4. Paquete con definiciones para la ALU (*ALUPackage.vhd*)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
package ALUPackage is
-- Sumador 1 bit
    component sumaBit is
        Port(a, b, cin : in STD_LOGIC;
        s , cout :  out STD_LOGIC);
    end component;
-- ALU 1 bit
    component ALUBit is
        Port ( a,b,cin : in STD_LOGIC;
        op,sel : in STD_LOGIC_VECTOR(1 downto 0);
        s,cout : out STD_LOGIC);
    end component;
-- ALU N bits
    component ALUNbits is
        generic ( N : integer := 4);
        Port ( a,b : in STD_LOGIC_VECTOR (N - 1 downto 0);
        op,sel : in STD_LOGIC_VECTOR (1 downto 0);
        s : out STD_LOGIC_VECTOR (N - 1 downto 0);
        flg_ov, flg_n, flg_z, flg_c : out STD_LOGIC);
    end component;
end package;
```

A.5. Testbench (*Testbench.vhd*)

```
library IEEE;
library work;
use IEEE.STD_LOGIC_1164.ALL;
use work.ALUPackage.ALL;

entity Testbench is
end Testbench;

architecture Behavioral of Testbench is

component ALUNbits is
    Generic (N : integer := 4);
    Port ( a,b : in STD_LOGIC_VECTOR (N - 1 downto 0);
           op,sel : in STD_LOGIC_VECTOR (1 downto 0);
           s : out STD_LOGIC_VECTOR (N - 1 downto 0);
           flg_ov, flg_n, flg_z, flg_c : out STD_LOGIC);
end component;

signal a,b,s : STD_LOGIC_VECTOR(3 downto 0);
signal flg_ov, flg_n, flg_z, flg_c : STD_LOGIC;
```

```
signal op,sel : STD_LOGIC_VECTOR(1 downto 0);

begin

ALU: AlUNBits
    port map(
    a => a,
    b => b,
    op => op,
    sel => sel,
    s => s,
    flg_ov => flg_ov,
    flg_n => flg_n,
    flg_z => flg_z,
    flg_c => flg_c);
process
begin
    a <= "0101"; -- A = 5
    b <= "1110"; -- B = -2
    op <= "11"; -- Suma
    sel <= "00";
    wait for 10 ns;
    sel <= "01"; -- Resta
    wait for 10 ns;
    sel <= "00"; -- AND
    op <= "00";
    wait for 10 ns;
    sel <= "11"; -- NAND
    op <= "01";
    wait for 10 ns;
    sel <= "00"; -- OR
    op <= "01";
    wait for 10 ns;
    sel <= "11"; -- NOR
    op <= "00";
    wait for 10 ns;
    sel <= "00"; -- XOR
    op <= "10";
    wait for 10 ns;
    sel <= "01"; -- XNOR
    op <= "10";
    wait for 10 ns;
    b <= "0111"; -- B = 7
    sel <= "00"; -- Suma
    op <= "11";
    wait for 10 ns;
    b <= "0101"; -- B = 5
    sel <= "01"; -- Resta
```

```
    wait for 10 ns;
    sel <= "11";
    op <= "01"; -- NAND (NOT)
    wait;
end process;
end Behavioral;
```
