Instituto Tecnológico de Costa Rica
Escuela de Ingeniería Electrónica
EL3313 Taller de Diseño Digital
Prof. M.Sc. Kaled Alfaro Badilla
II semestre 2024

Documentación del laboratorio 1: Introducción al diseño digital con HDLs y herramientas EDA de síntesis

Grupo 1

Integrantes:
Brayan de Jesús Barquero Madrigal
Keylor David Muñoz Soto
Jeffrey Isaac Salas Quiel

21 de agosto del 2024

Índice

1. Ejercicio 1

En este ejercicio se diseño un codificador para una matriz de 16 teclas. El codificador convertirá la posición de una tecla presionada en un valor binario de 4 bits. Luego se diseñan los bloques combinacionales requeridos, minimizando el circuito para usar el menor número posible de compuertas, para el circuito se utilizaron compuertas NOT y NAND.

El circuito genera señales que indican cuál fila está activa. El decodificador debe producir una señal binaria que indique cuál fila está activa y mostrar el dato de la columna en esa fila.

- Decodificador 2 a 4

Entrada B	**Entrada A**	**X4**	**X3**	**X2**	**X1**
0	0	0	1	1	1
0	1	1	0	1	1
1	0	1	1	0	1
1	1	1	1	1	0

Tabla 1: Tabla de verdad del decodificador

De esta tabla de verdad se obtienen la siguientes ecuaciones booleanas, con las cuales se realiza el esquema de la figura 1.

$$\overline{X_1} = B \cdot A$$
$$\overline{X_2} = B \cdot \overline{A}$$
$$\overline{X_3} = \overline{B} \cdot A$$
$$\overline{X_4} = \overline{B} \cdot \overline{A}$$

- Codificador 4 a 2

Y4	**Y3**	**Y2**	**Y1**	**Salida D**	**Salida C**
0	1	1	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	0	1	1

Tabla 2: Tabla de verdad codificador

De la tabla de verdad se obtienen las ecuaciones booleanas, con las que se realiza el esquema de la figura 2.

$$\overline{D} = y_2 \cdot y_1$$
$$\overline{C} = y_4 \cdot y_3$$

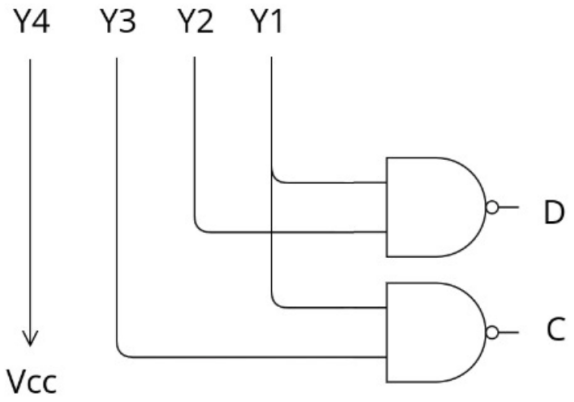

Figura 2: Codificador 4 a 2.

2. Ejercicio 2

Este proyecto tiene como objetivo diseñar un módulo en una FPGA que convierta el código ingresado por medio de 4 interruptores (SW[3:0]) a su complemento a 2, y muestre el resultado en 4 LEDs (LED[3:0]). El diseño incluye la implementación del módulo principal y un banco de pruebas (testbench) para verificar su funcionamiento de forma automática.

La tabla de verdad muestra las salidas esperadas (complemento a 2) para cada posible combinación de entradas.

Tabla de verdad

SW (Input)	**Led (Output)**
0000	0000
0001	1111
0010	1110
0011	1101
0100	1100
0101	1011
0110	1010
0111	1001
1000	1000
1001	0111
1010	0110
1011	0101
1100	0100
1101	0011
1110	0010
1111	0001

Tabla 3: Tabla de verdad codificador

Entrada de Datos a través de Interruptores

El módulo recibe un número binario de 4 bits mediante cuatro interruptores (SW[3:0]), donde:

- SW[3] representa el bit más significativo (MSB).
- SW[0] representa el bit menos significativo (LSB).

Cálculo del Complemento a 2

El complemento a 2 es una técnica para representar números negativos en binario, y se calcula en dos pasos:

1. **Invertir los bits**: Se cambia cada bit del número binario ingresado. Si `sw` es el número ingresado, `sw` representa el número con todos los bits invertidos.
2. **Sumar 1**: Al número con los bits invertidos, se le suma 1 para obtener el complemento a 2.

Salida de Datos a través de LEDs

El resultado del cálculo del complemento a 2 se muestra en cuatro LEDs (LED[3:0]), donde:

- LED[3] representa el bit más significativo del complemento a 2.
- LED[0] representa el bit menos significativo.

Así, los LEDs reflejan el complemento a 2 del número ingresado a través de los interruptores.

3. Ejercicio 3

Para este ejercicio se diseño un Multiplexor de 4-a-1 para un ancho de datos de entrada y salida variable. Para esto se realizó la lógica básica de un Mux 4-a-1, pero se colocó el Parámetro “WIDTH“ en la entrada para cambiar el ancho de datos según corresponda en el testbench, el parámetro “TESTS“ del testbench se usó para definir el número de pruebas, en este testbench se probó un ancho de datos de 4, 8 y 16 bits.

A continuación se muestra la tabla de control para un Mux 4 a 1 con ancho de datos variable.

- Mux 4-a-1

S1	**S0**	**Salidas[`WIDTH-1:0`]**
0	0	M1
0	1	M2
1	0	M3
1	1	M4

Tabla 4: Tabla de verdad Mux 4 a 1

4. Ejercicio 4

Descripción del Módulo PWM

El módulo `PWM` genera una señal de modulación por ancho de pulso (PWM) a partir de una señal de reloj y una entrada de 4 bits que define el ciclo de trabajo de la señal PWM.

- **Entradas:**
  - `clk`: Señal de reloj.
  - `rst`: Señal de reinicio asíncrono.
  - `c_trabajo`: Valor de 4 bits que determina el ciclo de trabajo de la señal PWM.
- **Salida:**
  - `pwm_out`: Señal de salida PWM.
- **Funcionamiento:**
  - El módulo utiliza un contador de 15 bits (`contador`) para contar los ciclos de reloj hasta alcanzar el valor definido por el parámetro `PERIODO`, que en este caso equivale a 1 ms (27000 ciclos a una frecuencia de 27 MHz).
  - El valor de `corte` se calcula en función de `c_trabajo`, que define el ciclo de trabajo deseado. Este valor determina el umbral para la señal PWM.
  - Durante cada ciclo del reloj, el contador se incrementa hasta alcanzar el valor de `PERIODO`. Una vez alcanzado, el contador se reinicia.
  - La salida PWM (`pwm_out`) se actualiza comparando el contador con el valor de `corte`. Si el contador es menor que `corte`, `pwm_out` es 1; de lo contrario, es 0.

  Este módulo es útil para generar señales PWM que pueden ser utilizadas en aplicaciones como el control de velocidad de motores o la modulación de intensidad de luces.

5. Ejercicio 5

En este ejercicio se realiza el diseño de una ALU (Unidad Aritmética Lógica) parametrizable de n bits de entrada y salida. La ALU a diseñar se observa en la siguiente figura:

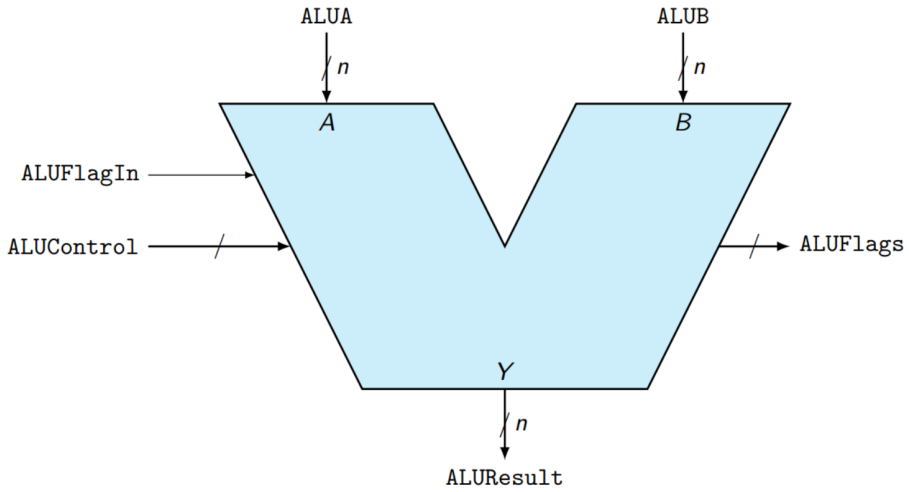

Figura 3: Diagrama de bloques de la ALU a diseñar.

ALUA, *ALUB* y *ALUResult* son definidas por el parámetro n, son variables de n-1 bits.

La salida *ALUFlags* se definió como una señal de dos bits, el bit 0 se utiliza para la bandera "C"que indica el bit de salida al hacer un corrimiento, y el bit 1 se usa para la bandera "Z"que indica cuándo el resultado es cero (1 si el resultado es cero).

La entrada *ALUFlagIn* es una señal de un bit. Se utiliza como acarreo de entrada para la suma y resta. Selecciona el bit de entrada en los corrimientos. En negación, incremento y decremento, selecciona cuál de los operandos utilizar (0 selecciona A y 1 selecciona B).

La entrada *ALUControl* es una señal de 4 bits que selecciona la operación a realizarse, de un total de nueve operaciones disponibles, las cuales se describen a continuación:

- **0h - and**

  Realiza la operación lógica AND bit a bit entre los operandos ALUA y ALUB. El resultado es dado según la siguiente tabla:

Entrada A	Entrada B	Salida
0	0	0
0	1	0
1	0	0
1	1	1

Tabla 5: Tabla de verdad de una compuerta AND.

  Para implementar esta función se utiliza &, ya definida en SystemVerilog para la operación lógica AND.

- **1h - or**

  Realiza la operación lógica OR bit a bit entre los operandos ALUA y ALUB. El resultado es dado según la siguiente tabla:

Entrada A	Entrada B	Salida
0	0	0
0	1	1
1	0	1
1	1	1

Tabla 6: Tabla de verdad de una compuerta OR.

Para implementar esta función se utiliza |, ya definida en SystemVerilog para la operación lógica OR.

- **2h - suma (en complemento a dos)**
  Suma los operandos ALUA y ALUB considerando que ambas entradas están representadas en complemento a dos. ALUFlagIn se ingresa como acarreo de entrada, agregándolo a la suma. Se obtiene el resultado sumando las tres entradas (ALUA + ALUB + ALUFlagIn).
- **3h - incrementar en uno el operando**
  Suma uno a alguno de los operandos dependiendo de ALUFlagIn, si ALUFlagIn es 1 entonces la suma se hace al operando ALUB, si es 0 se hace la suma al operando ALUA. Esto se implementa mediante condicionales de SystemVerilog.
- **4h - decrementar en uno el operando**
  Resta uno a alguno de los operandos dependiendo de ALUFlagIn, si ALUFlagIn es 1 entonces la resta se hace al operando ALUB, si es 0 se hace la resta al operando ALUA. Esto se implementa mediante condicionales de SystemVerilog.
- **5h - not (sobre un operando)**
  Realiza la operación lógica de negación bit a bit, el resultado depende de la siguiente tabla:

Entrada	Salida
0	1
1	0

Tabla 7: Tabla de verdad de una compuerta NOT.

Al operando que se aplique depende de ALUFlagIn, si ALUFlagIn es 1 entonces la negación se hace al operando ALUB, si es 0 se aplica al operando ALUA. Esto se implementa mediante condicionales y ∼, ya definida en SystemVerilog para la operación lógica NOT.

- **6h - resta (en complemento a dos)**
  Resta los operandos ALUA y ALUB considerando que ambas entradas están representadas en complemento a dos. ALUFlagIn se ingresa como acarreo de entrada, agregándolo a la resta. Se obtiene el resultado restando las tres entradas (ALUA - ALUB - ALUFlagIn).
- **7h - xor**
  Realiza la operación lógica XOR bit a bit entre los operandos ALUA y ALUB, el resultado depende de la siguiente tabla:

Entrada A	Entrada B	Salida
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 8: Tabla de verdad de una compuerta XOR.

Se implementa con ^, ya definida para la operación lógica XOR en SystemVerilog.

- **8h - corrimiento a la izquierda de ALUA**

  Desplaza los bits del operando ALUA hacia la izquierda por la cantidad de posiciones indicada por el operando ALUB. Los nuevos bits introducidos en las posiciones menos significativas se rellenan con el valor indicado por ALUFlagIn. La bandera de salida C captura el último bit que salió del extremo más significativo de ALUA.

  Para esto, el código toma en cuenta ciertas condiciones, si ALUB es mayor o igual al tamaño de n, todos los bits se reemplazan por el bit de ALUFLagIn y se guarda el último bit desplazado. Si ALUB es cero, ALUA permanece igual. Si ALUB es menor a n, los bits se desplazan, se rellenan los espacios vacíos con el valor de ALUFlagIn, y se captura el bit más significativo que salió.

- **9h - corrimiento a la derecha de ALUA**

  Desplaza los bits del operando ALUA hacia la derecha por la cantidad de posiciones indicada por el operando ALUB. Los nuevos bits introducidos en las posiciones más significativas se rellenan con el valor indicado por ALUFlagIn. La bandera de salida C captura el último bit que salió del extremo menos significativo de ALUA.

  La implementación de esto es igual al caso del corrimiento a la izquierda, solo que ahora se usa el corrimiento a la derecha de SystemVerilog.

El diagrama de bloques implementado en el diseño es el siguiente:

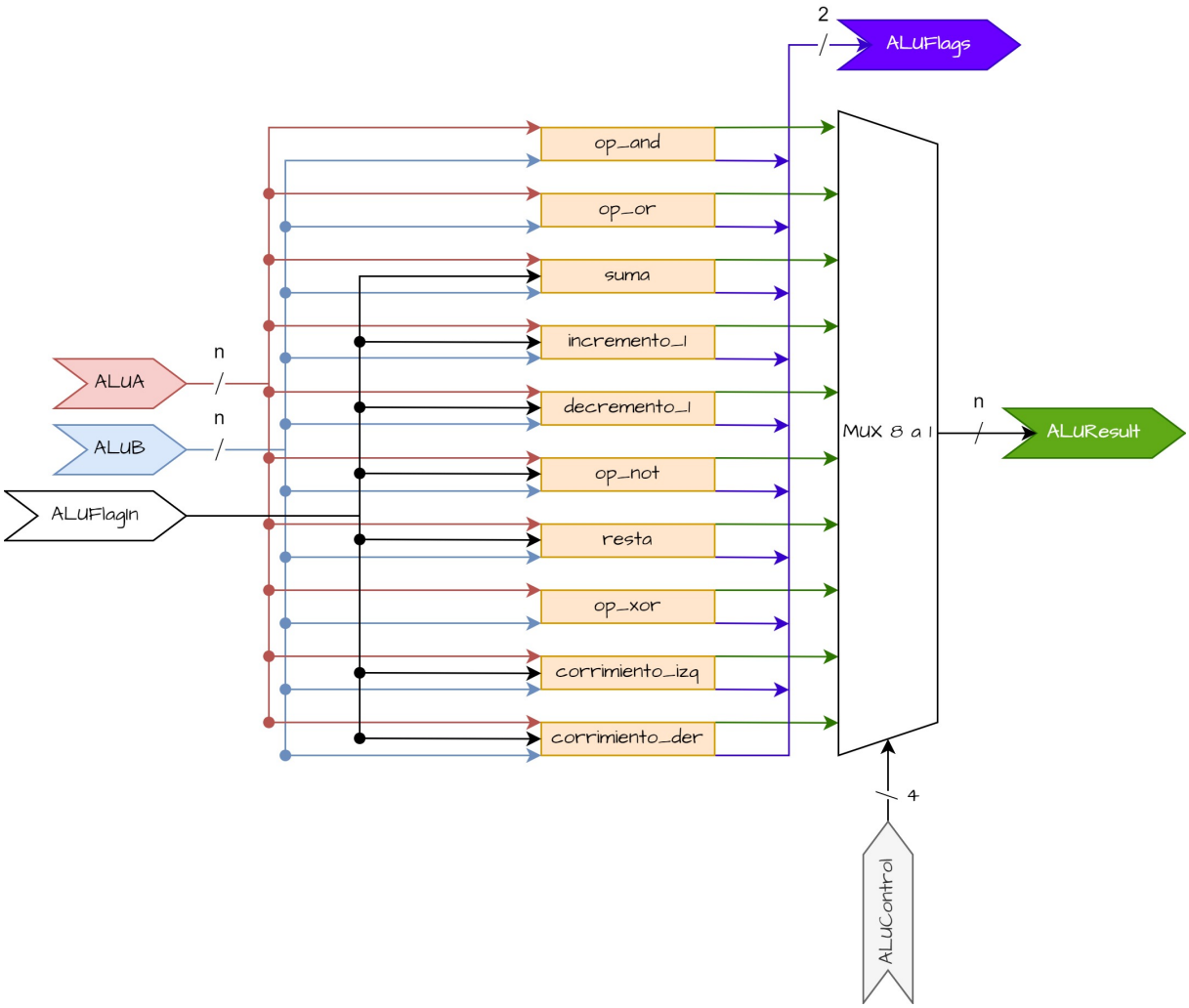

Figura 4: Diagrama de bloques ALU.

Con este diagrama se realizó la implementación en el código de SystemVerilog. Al realizar el testbench y la simulación usando DSim se comprobó su correcto funcionamiento:

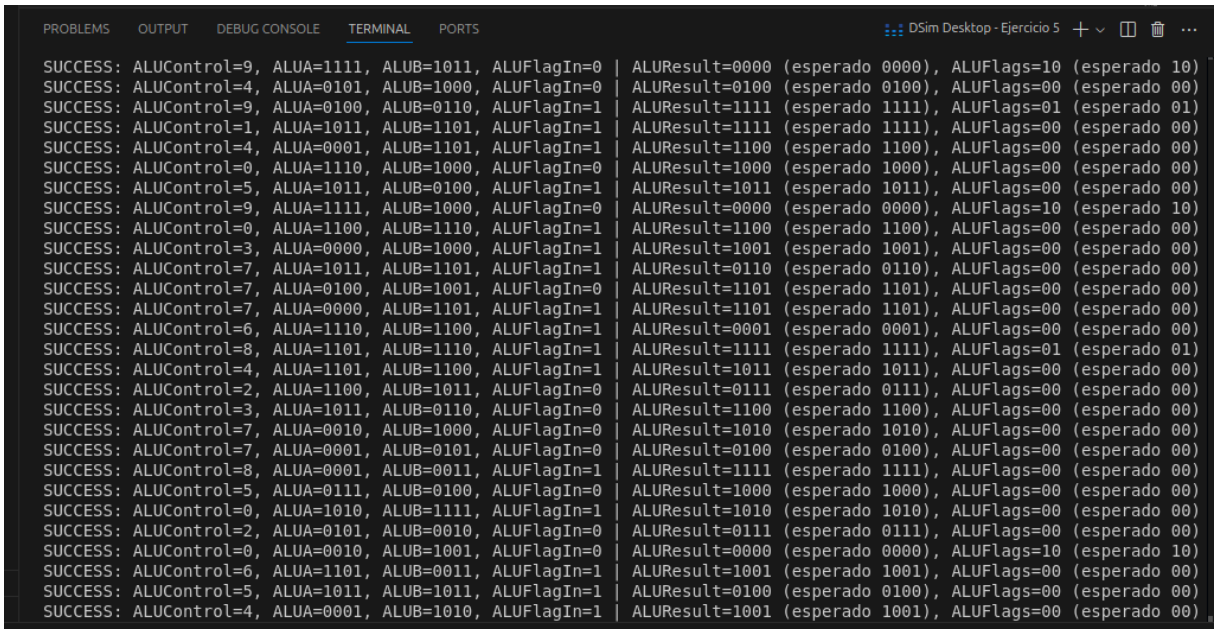

Figura 5: Resultados obtenidos para la simulación.

Además, se sintetizó correctamente utilizando YOSYS:

Figura 6: Resultados de síntesis.