Universidad de Costa Rica

Escuela de Ingeniería Eléctrica

IE0323 – Circuitos Digitales

Grupo 03

# PROYECTO:
# Verilog

Prof. Diego Dumani

Ricardo Mena Cortés A93776
Melissa Rodríguez Murillo B76529

05 de Diciembre del 2020

# Índice



# Índice de figuras

# Índice de tablas

# 1. Observaciones Iniciales

Se prentende construir la solución de los problemas propuestos en (Dumani, 2020).

Se organizó el proyecto en diferentes carpetas según el problema, esto para facilitar su ejcución, ya que se incluyó un *Makefile* para realizar la compilación del código y su visualización en *GTKwave*. Por lo que para la revisión de los diagramas de tiempo solamente es necesario ejecutar en una terminal siempre que se encuentre dentro del folder del correspondiente problema, el comando:

```
$ make
```

Y el *Makefile* se encargará del resto. De igual forma junto con cada *testbench* en el encabezado del archivo se incluye la línea de compilación en caso de preferirla.

También se debe notar que los módulos utilizados en el [1]problema 1 simplemente fueron copiados a la carpeta *problema_2*, para reutilizar los módulos creados para construir la unidad lógica aritmética solicitada. No se crearon nuevos módulos aparte de ALU.v, nuevamente se realizó lo mismo con la carpeta *problema_2* para realizar el ejercicio extra.

Además se aclara que todos los comentarios dentro del código fueron realizados en inglés(Excluyendo los Makefiles) como se acordó en una consulta realizada al profesor.

[1] AND.v, OR.v XOR.v, MUX4x1.v, fad_cell.v y full_adder.v

# 2. Ejercicio 0

Al realizar varios estímulos a las entradas de los multiplexores, se prefirió apliciar asignaciones no bloqueantes en las entradas, ya que de esta forma permitía más libertad, sin afectar otras secciones, ya que al utilizar entradas bloqueantes el tiempo en alto del *Select* no era independiente de las entradas, y esto causaba que se modificara de forma indirecta el tiempo en alto de la primera vez que ocurra el *Select*.

Además se modificó el color del reloj(en naranja) para distinguirlo con mayor facilidad de las otras señales.


Figura 1: Funcionamiento MUX2x1 sin memoria y con ella

Se realizaró un análisis del comportamiento de ambos multiplexores por separado, basados en el diagrama de tiempo de la figura (1), en la cual tiene ambos tipos de multiplexores, en el que $Y_{out_1}$ corresponde a la salida del multiplexor 2x1 sin memoria y $Y_{out_2}$ corresponde a la salida del multiplexor 2x1 con memoria.

## 2.1. Multiplexor 2x1 sin memoria

De este multiplexor se puede notar que en cuanto el bit de selección cambia su estado, lo hace de la misma forma su salida y no espera los pulsos de reloj, reflejando los cambios en su entrada seleccionada de forma inmediata en su salida.


Figura 2: Funcionamiento MUX2x1 sin memoria

Este hecho puede notarse en la figura (2), en la que se ha puesto el cursos en una zona de interés, donde se puede apreciar que se tiene seleccionada la salida $In_1$ la cual se encuentra en estado alto y antes de la llegada del flanco de reloj, ésta cambia de estado alto a estado bajo, haciéndolo de la misma forma la salida $Y_{out_1}$ de este multiplexor y al llegar el pulso de reloj, la entrada *Select* cambia de estado alto a estado bajo, seleccionando así el puerto $In_0$, y así también cambia la salida del MUX a estado alto nuevamente, no por el flanco de reloj, sino por el cambio de estado en la entrada *Select*.

Se prefirió ejemplificar esto con entradas diferentes a las utilizadas para la figura (1), ya que en el código suministrado se aprovechan los flancos de reloj para cambiar el estado del bit de selección en ambos multiplexores y esto genera un poco más de trabajo en su interpretación al estar relacionados mediante el cambio de reloj a su entrada de selección.

### 2.2. Multiplexor 2x1 con memoria

Como se puede ver de la figura (1) una vez se activa el *reset* y permite el funcionamiento del multiplexor, éste mantiene su salida en el mismo estado en espera del flanco creciente del reloj, al llegar el segundo flanco creciente, el multiplexor verifica la selección de la entrada *Select*, la cual se encuentra en estado alto, lo que indica que se debe redirccionar la entrada $In_1$ y pasarla a la salida $Y_{out_2}$ del multiplexor, poniendolo así en estado alto. El MUX mantiene en alto la salida y espera a la llegada del siguiente flanco de reloj, en el que verifica la entrada *Select* en estado bajo, poniendo de esta forma a la salida $Y_{out_2}$ pero la entrada $In_0$, manteniendo la salida en alto.

Al llegar al cuarto flanco creciente del reloj se seleciona nuevamente la entrada $In_1$ la cual se encuentra en estado bajo, por lo que el MUX pone su salida $Y_{out_2}$ en estado bajo.

El multiplexor continúa de la misma forma esperando los flancos crecientes de reloj para realizar los cambios en su salida aunque las entradas cambien antes de esto, siendo esta la mayor diferencia entre ambos multiplexores, ya que como se pudo observar. En el multiplexor sin memoria, si su salida cambia inmediatamente, cambiara su entrada seleccionada de de igual forma.

## 3. Ejercicio 1

Implementación estructural una compuerta AND, una OR y una XOR de dos entradas y una salida de 4 bits, además de un sumador completo de 4 bits con acarreo de entrada y acarreo de salida, junto a un multiplexor de 4bits

### 3.1. Módulo AND de 4 bits

Tanto su entrada como su salida corresponde a vectores de 4 bits como puede verse en la figura (3).

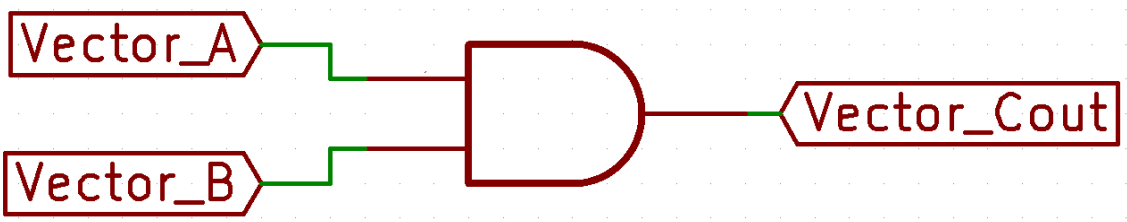


Figura 3: Compuerta AND de 4bits

Aunque una representación estructural y más fiel a su construcción es la representada en la figura (4) en donde se puede notar como cada elemento de los vectores de entrada es comparado entre si para generar un vector de salida $C_{out}$.

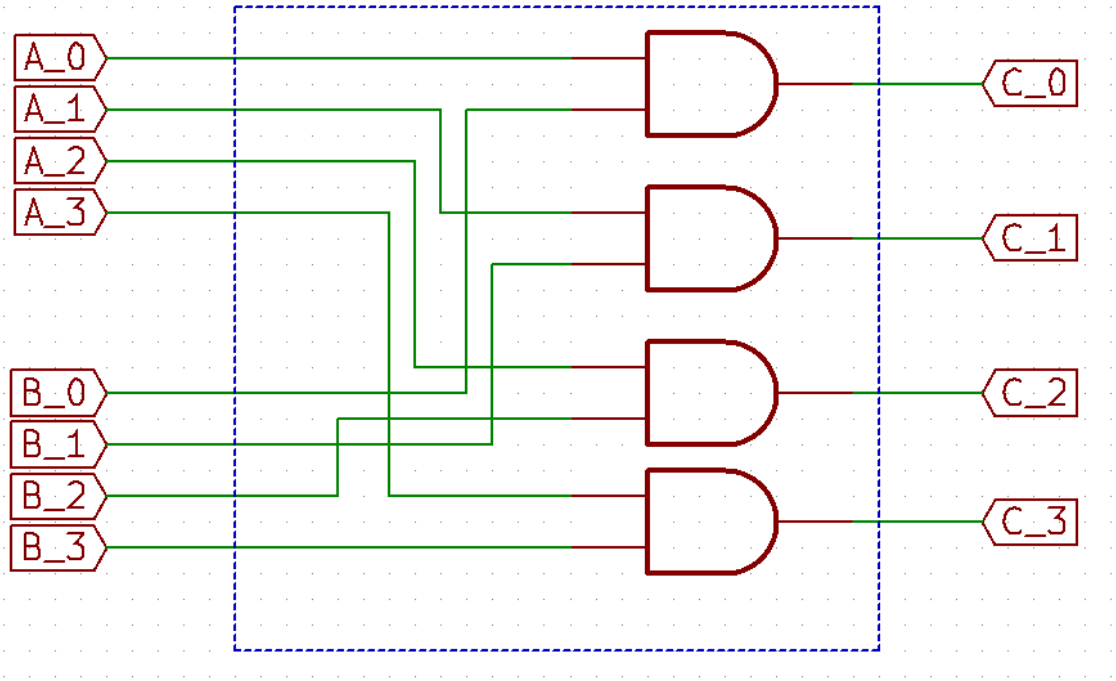


Figura 4: Compuerta AND de 4bits, estructura

Al diseñar este módulo en Verilog se realizó una comparación bit-a-bit de dos vectores que se le envían al módulo y éste asigna el resultado de la operación $AND$ bit por bit a su salida, como se ve en el código(1).

Listing 1: Módulo AND 4 bits

```
1    module AND ( input [3:0] a, b, output [3:0] cout);
2        assign cout = a&b;
3    endmodule
```

## 3.2. Módulo OR de 4 bits

De igual forma que la compuerta anterior, esta compuerta *OR* recibe dos vectores de 4 bits a su entrada para comparar y genera un vector de 4 bits a su salida como se muestra en la figura (5).

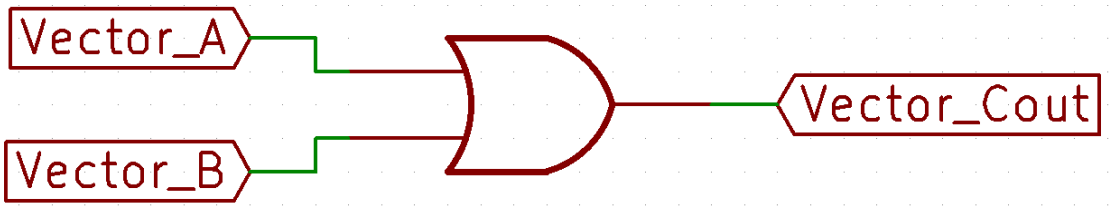


Figura 5: Compuerta OR de 4bits

El modelo estructural de la compuerta OR sería el presentado en la figura 6.

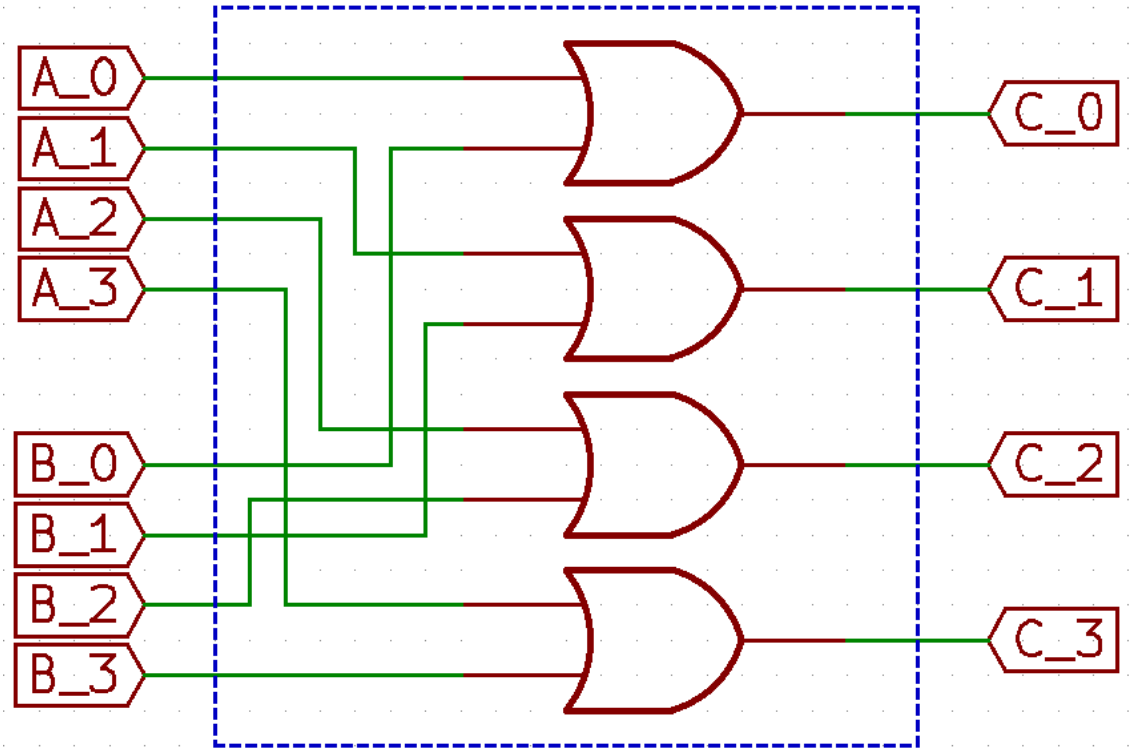


Figura 6: Compuerta OR de 4bits, estructura

De forma similar al módulo construido en Verilog para la compuerta *AND*, para esta compuerta *OR* se realizó una comparación bit por bit de los vectores de entrada y se genera una salida vectorial como se planificó en la figura 5, por lo que se muestra el módulo diseñado en el código (2).

Listing 2: Módulo OR 4 bits

```
1    module OR ( input [3:0] a, b, output [3:0] cout );
2        assign cout = a|b;
3    endmodule
```

### 3.3. Módulo XOR de 4 bits

Se construye ahora una compuerta $XOR$ que recibe dos vectores de 4 bits a su entrada para comparar y genera un vector de 4 bits a su salida de forma similar a las anteriores, como se muestra en la figura (7).

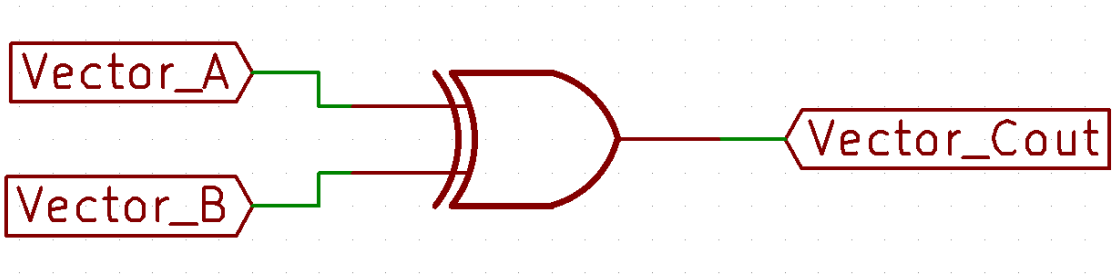


Figura 7: Compuerta XOR de 4bits

El modelo estructural de esta compuerta $XOR$ sería el presentado en la figura 8.

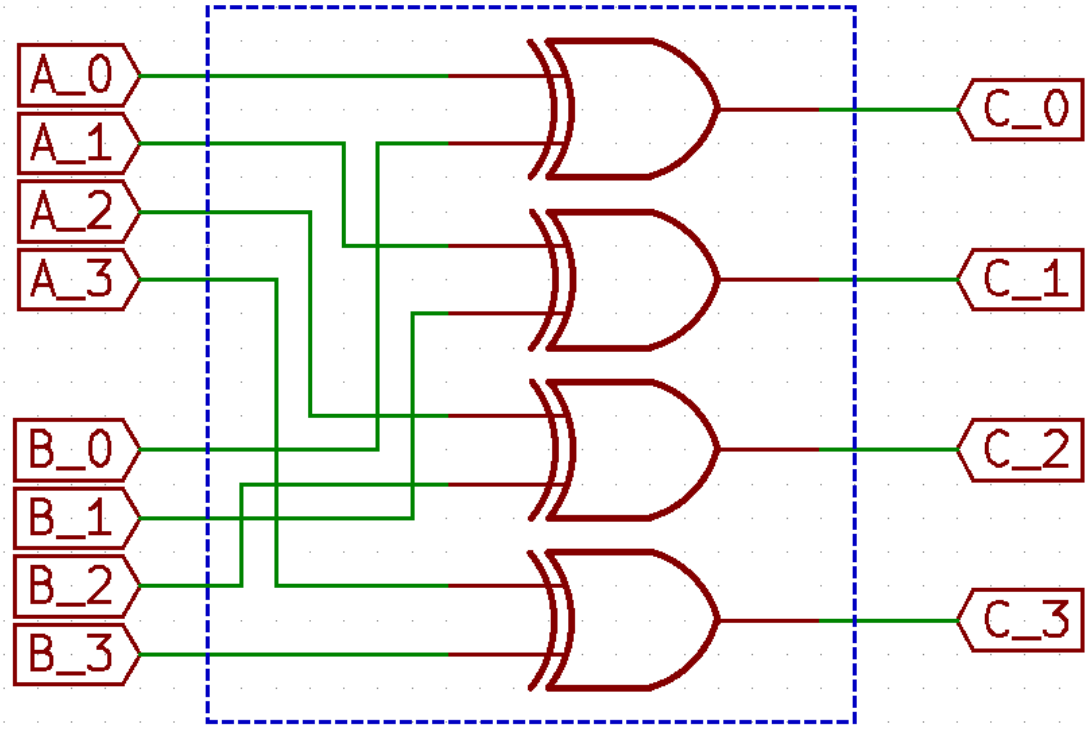


Figura 8: Compuerta XOR de 4bits, estructura

Se creó esta compuerta $XOR$ basados en la construcción estructural de la figura (8) como se realizó con las compuertas anteriores, mediante su implementación en Verilog, generando el código (3), que de igual forma que los módulos anteriores realiza una comparación bit por bit de sus entradas y genera una salida vectorial del resultado.

Listing 3: Módulo XOR 4 bits

```
1    module XOR ( input [3:0] a, b, output [3:0] cout);
2        assign cout = a^b;
3    endmodule
```

## 3.4. Módulo Sumador Completo de 4 bits, con acarreo final

El módulo sumador completo esta construido a partir de cuatro módulos de sumadores completos de 2x1 bit con acarreo de entrada y salida, en los que cada módulo de sumador actua como una celda típica.

Cada celda típica se construyó a partir de la tabla (1) suministrada junto con el enunciado del proyecto.

| A | B | $C_{in}$ | Sum | $C_{out}$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Tabla 1: Tabla de verdad para full-adder de 1 bit

A partir de la tabla (1) se construyeron los mapas de Karnough que determinaron la función de salida para un sumador completo de un bit con acarreo de entrada y salida, como se muestra en las figuras (9) y (10).

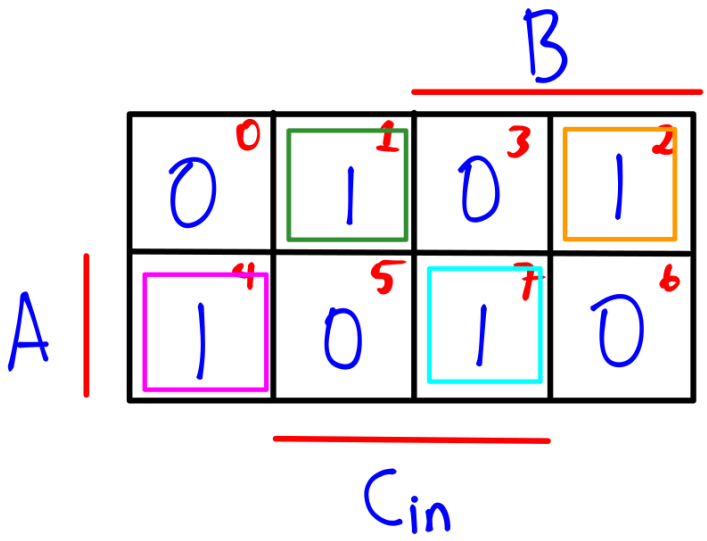


Figura 9: Mapa de Karnaugh para la salida $Sum(A, B, Cin)$

De este mapa podemos observar que todos los implicantes son primos y esenciales, por lo que la función que describre $Sum(A, B, C_{in})$ es la ecuación (1).

$$Sum(A, B, C_{in}) = A \oplus B \oplus C_{in} \tag{1}$$

De forma similar se contruye el mapa de Karnaugh (10) para el acarreo de salida $C_{out}$, donde se tienen tres implicantes primos, los cuales además son todos esenciales.

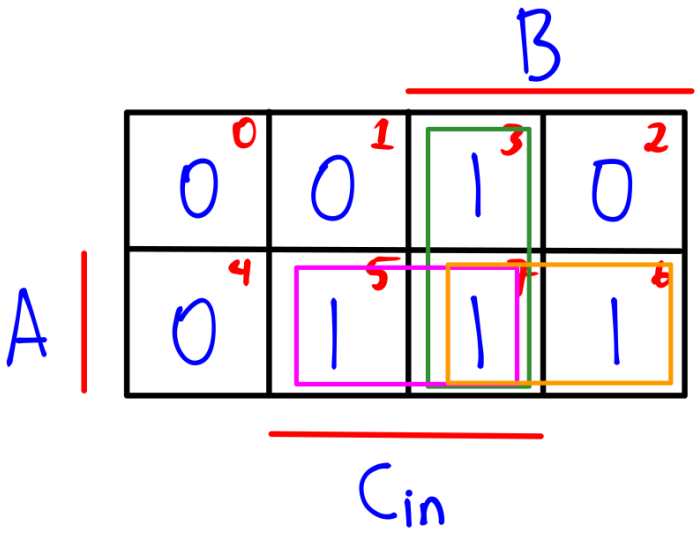


Figura 10: Mapa de Karnaugh para la salida $C_{out}$

Y se obtiene su función de salida con la ecuación (2).

$$C_{out}(A, B, C_{in}) = AC_{in} + BC_{in} + AB \tag{2}$$

A partir de las ecuaciones (1) y (2) se construye el módulo estructural de un sumador completo de un bit con acarreo de entrada y de salida como se muestra en la figura (11).

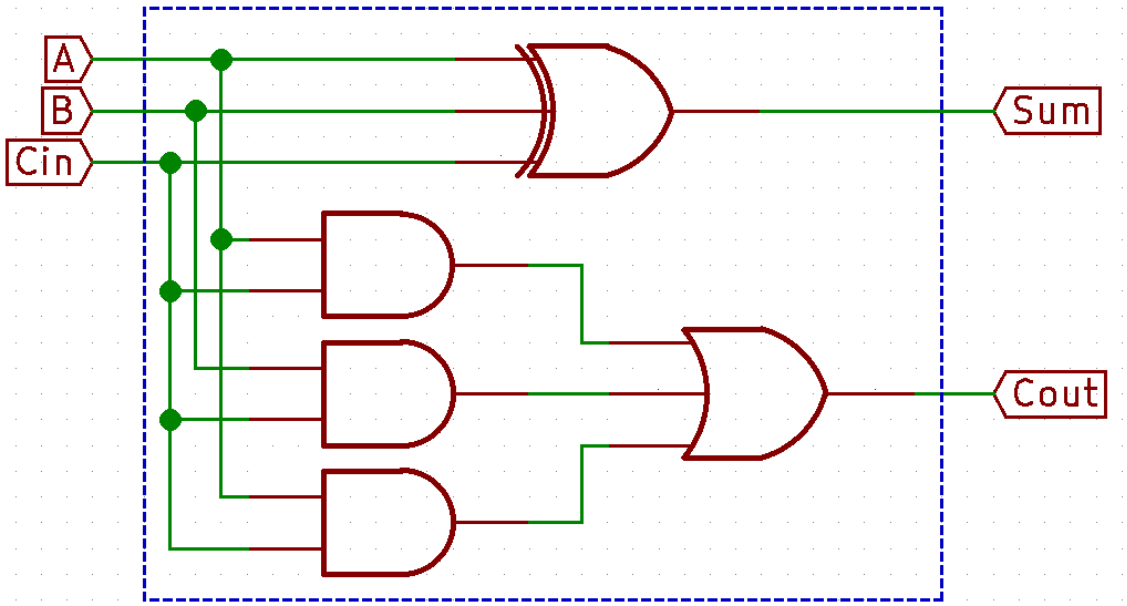


Figura 11: Sumador Completo de un bit con acarreo de entrada y salida

Diseñandose apartir de (11) un módulo en [2]Verilog que modele este comportamiento.

Listing 4: Sumador completo de un bit. Celda típica sumador 4 bits

```
1  module fad_cell(input A, B, Cin, output Sum, Cout);
2     assign Sum = A^B^Cin;
3     assign Cout = A & Cin | B & Cin | A & B;
4  endmodule // fad_cell
```

Con este sumador (11) es posible construir un sumador completo de cuatro bits, utilizando cuatro sumadores de 1 bit que puedan procesar en paralelo una suma de 4 bits, como se muestra en la figura (12).

[2]El termino *fad_cell* del nombre del módulo se deriva de *full adder cell* ya que este sumador de un bit corresponde a una celda típica para un sumador de 4bits y a esto se debe el termino de celda

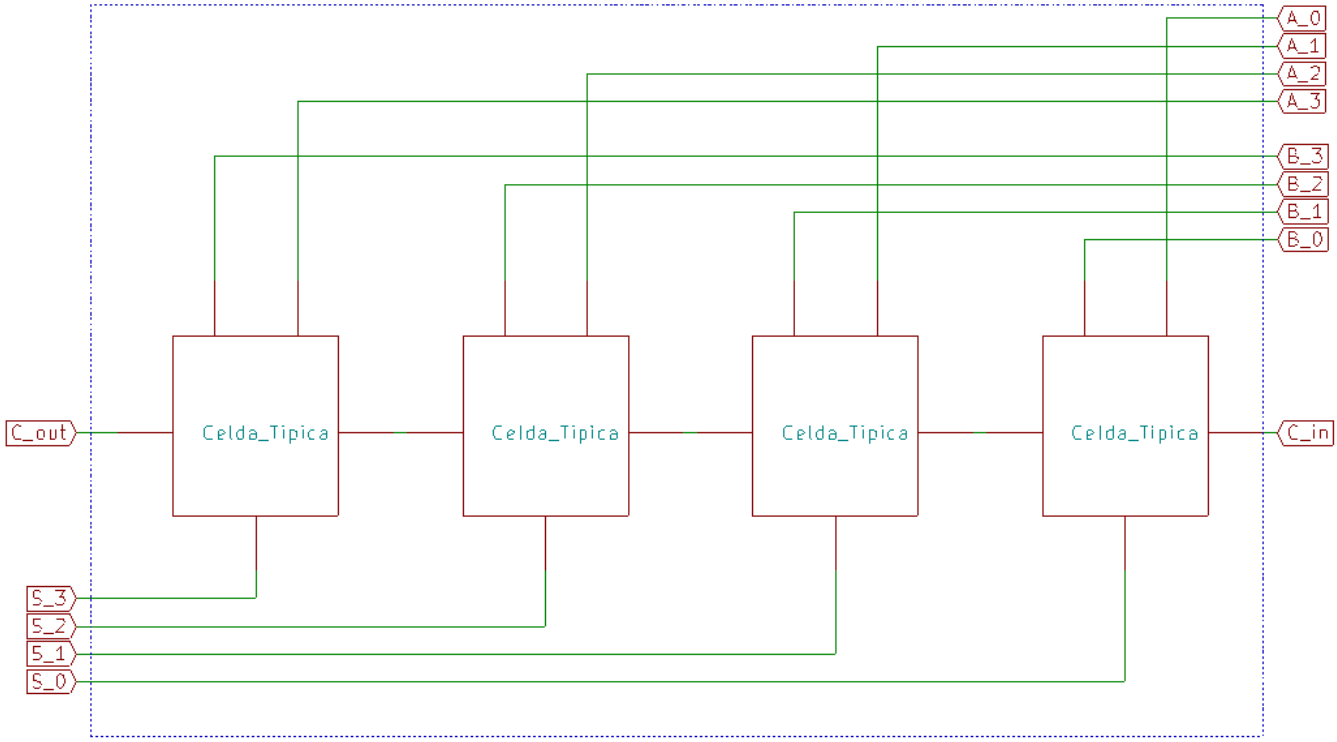


Figura 12: Sumador Completo de 4 bits con acarreo de entrada y salida

Y su implementación en Verilog en el bloque de código (5).

Listing 5: Sumador completo de 4bits con acarreo de entrada y salida

```
1
2   `include "fad_cell.v"
3
4    module fullAdder(input [3:0] A, B, input Cin, output [3:0] ...
        Sum, output Cout);
5        wire C_0, C_1, C_2;//Nets
6        fad_cell fulladder_cell_0((A[0]), (B[0]), Cin, Sum[0], ...
            C_0);//Initial Cell
7        fad_cell fulladder_cell_1((A[1]), (B[1]), C_0, Sum[1], ...
            C_1);//Middle Cell
8        fad_cell fulladder_cell_2((A[2]), (B[2]), C_1, Sum[2], ...
            C_2);//Middle Cell
9       fad_cell fulladder_cell_3((A[3]), (B[3]), C_2, Sum[3], ...
           Cout);//Final Cell
10   endmodule // fullAdder
```

## 3.5. Módulo MUX 4x1 con entradas y salida de 4bits

Para realizar el módulo del multiplexor 4x1 se empleó una defínción conductual mediante expresiones ternarias que determinan la salida a elegir, dependiendo de la selección que se realize. Al igual que las compuertas implementadas todas las entradas y la salida de este multiplexor reciben

un [3]vector de 4 bits y genera un vector de salida. De esta forma se simplifica enormemente la implementación de este tipo de multiplexor como puede verse del código (6) construido en Verilog, que de haberse realizado de forma estructural sería necesario una gran cantidad de compuertas lógicas y trabajo para determinar su función de salida.

Listing 6: Multiplexor 4x1 con entradas y salida de 4bits

```
1     module MUX (input [3:0]A0, A1, A2, A3 , input ...
          [1:0]M_select, output [3:0]M_out);
2        //Handling with ternary expretion to select Input
3        assign M_out = M_select[1] ?
4                    ( M_select[0] ? A3 : A2 ) :
5                    (M_select[0] ? A1 : A0 );
6     endmodule // MUX
```

## 3.6. Testbench

Una vez creados los módulos solicitados, es necesario probarlos, por lo que se realiza el [4]*testbench* que nos proporciona la facilidad de ingresar más de un caso posible de uso, para las entradas del modelo.

Tanto las compuertas, como el sumador tienen como entradas dos vectores de 4 bits, llamados $a$ y $b$ respectivamente; para probar su funcionalidad se buscó realizar todas las combinaciones que abarquen todos los casos posibles para todas las compuertas, por lo que se generan seis en total y se obtienen los resultados esperados.

Los últimos tres casos aplicados a los vectores $a$ y $b$ son de interés únicamente para la operación de suma.

En el MUX 4x1, se ingresa en sus entradas cuatro vectores, de manera aleatoria para probar su funcionamiento y configurar su salida con la entrada de selección. Posteriormente para el ejercicio 2, las entradas del multiplexor serán las salidas de las compuertas y el sumador, el cual debe de arrojar los datos de manera correcta, ya que fue probada su funcionalidad en esta sección.

[3]Fisicamente sería un grupo de 4 pines(1 bit por pin) por vector, por lo que se tendrían en total 16 pines de entrada, 4 de salida y dos de control

[4]Dentro de la carpeta problema_1, nombrado como: *test_b_ejercicio_uno.v*

### 3.6.1. Vectores de entrada para compuertas de 4 bits y sumador

| Entradas | | | Operación | | | Sumador | |
|---|---|---|---|---|---|---|---|
| A | B | C_in | AND | OR | XOR | Suma | Acarreo |
| 0000 | 0000 | 1 | 0000 | 0000 | 0000 | 0001 | 0 |
| 0000 | 0000 | 0 | 0000 | 0000 | 0000 | 0000 | 0 |
| 0001 | 1110 | 0 | 0000 | 1111 | 1111 | 1111 | 0 |
| 1110 | 0001 | 1 | 0000 | 1111 | 1111 | 0000 | 1 |
| 1111 | 1111 | 1 | 1111 | 1111 | 0000 | 1111 | 1 |
| 1111 | 1111 | 0 | 1111 | 1111 | 0000 | 1110 | 1 |
| 0001 | 0001 | 1 | 0001 | 0001 | 0000 | 0011 | 0 |
| 0010 | 0010 | 1 | 0010 | 0010 | 0000 | 0101 | 0 |
| 0100 | 0100 | 1 | 0100 | 0100 | 0000 | 1001 | 0 |

Tabla 2: Vectores de entrada y resultado para compuertas de 4bits y sumador completo

Como puede verse de la tabla (2) este conjunto de casos de entrada logra replicar la tabla de verdad para cada compuerta que constituye cada uno de los módulos creados.

Por ejemplo, si se toma el bit menos significativo(LSB) de cada fila, se pueden formar las entradas para una tabla de verdad de cero a tres y al revisar su respectiva salida para cada tipo de compuerta(de igual forma en su bit menos significativo) se genera el resultado esperado. Lo mismo sucede con cada uno de los bits hasta el más significativo(MSB).

Por lo que al aplicar estas entradas de forma consecutiva nos logra exitar todas las compuertas, en todos los casos posibles, como puede verse de la figura (13).


Figura 13: Testbench aplicado a las Compuertas de 4bits

Estos mismos vectores de entrada utilizados para verificar las compuertas fueron utilizados para verificar el funcionamiento del sumador completo, pero se agregó un vector extra para el acarreo de entrada y lograr replciar de la misma forma la tabla de verdad para cada uno de los sumadores de 1bit de los que este está compuesto.

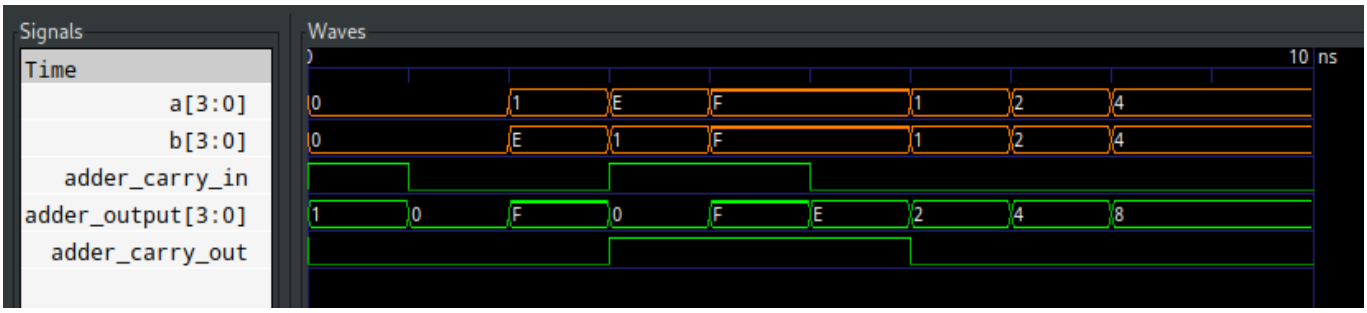


Figura 14: Testbench aplicado al Sumador Completo de 4bits

Ahora como se explicó en la introducción de esta subsección se tomaron cuatro vectores con valores completamente al azar($A_0 = E, A_1 = A, A_2 = B, A_3 = 4$) y se generó un vector de control con valores de 0 hasta 3, de esta forma podemos comprobar que el multiplexor, logra llegar a todos los casos de selección posibles. Y como puede verse de la figura (15) se logró el objetivo de seleccionar todas las entradas en orden respectivo.


Figura 15: Testbench aplicado al Multiplexor de 4bits de entrada y salida

Por último se presenta el diagrama de tiempos completo aplciado a todos los módulos construidos en la figura (16)

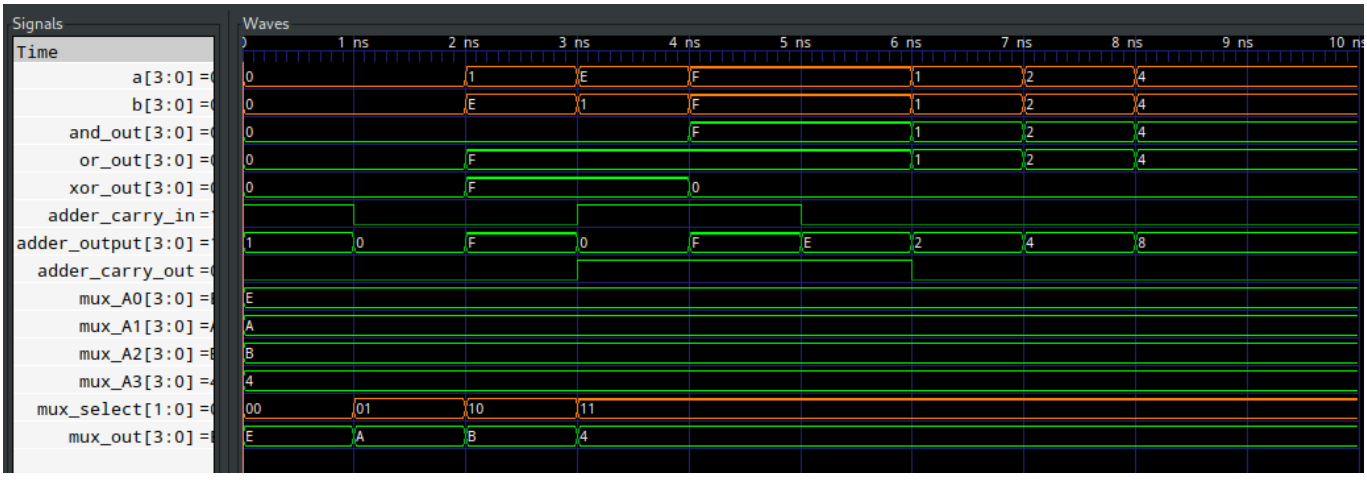


Figura 16: Testbench aplicado a todo el Ejercicio 1

# 4. Ejercicio 2

Se conectaron los módulos que se habian construido en el ejercicio 1: Las compuertas AND, OR y XOR, además del sumador. Se eliminó el acarreo de entrada del sumador como requerimiento de diseño, además se dejaron solamente los vectores de entrada A y B y sus casos de uso, ya que los vectores $A_0...A_3$ fueron remplazados por las salidas de las compuertas, las cuales van directamente a las entradas del multiplexor. Se mantiene además el vector de selección del multiplexor para aplicar todos los casos posibles.

Para las salidas de la ALU, sigue la lógica del MUX4x1 de la sección del ejercicio 1, el cual, según los valores del selector, así será la salida. Es decir, las compuertas se encuentran conectados al multiplexor , descritas anteriormente, y al sumador, por lo que la salida de estos, serán las entradas del MUX.

Una vez ingresados los valores de las entradas del ALU, A y B, se generan las salidas de las compuertas y el sumador, las cuales son recibidas por el multiplexor, sin embargo, según los valores del selector, la salida del ALU será igual a alguna entrada del multiplexor. En el caso de selector igual a 00 la salida final será igual a la primera entrada del MUX, o sea, al resultado de salida de la compuerta AND, si el selector se encuentra en 01 la salida es igual a la operación OR de los vectores A y B, cuando es 10 la salida es la entrada de la tercera conexión de las compuertas con el multiplexor, lo que implica la operación XOR de las entradas Ay B; finalmente en 11 la salida es la que proporciona el sumador como entrada en el multiplexor, así en general para cada vector ingresado y según el valor del selector.

El acarreo se considera una salida independiente, por lo que no se toma encuenta como entrada para el multiplexor, sino que solamente es una salida del ALU, al ejecutarse todos los módulos de las compuertas y el sumador, en paralelo, esto puede generar un acarreo de salida sin estar seleccionada la operación de suma y se dejó de esta manera debido a que el diagrama presentado para la ALU lo especificaba de esta forma.

Como puede verse del diagrama de tiempo en la figura (17) la primera y segunda operacion aplicada es la operación AND a un par de entradas 00 y $FF$ en el primer caso se debió obtener como resultado 0 y en el segundo $F$ y la ALU logró aplicar la operación de forma correcta. En el tercer caso se aplicó una operación OR a las entradas $1|E$ por lo que el resultado resultó correcto, ya que la operación bit a bit de los vectores 0001 y 1110 terminan dando como resultado 1111. Al realizar la operación XOR en el tiempo $3ns$ con los vectores de entrada 1110 y 0001 se obtiene 1111 o en su forma hexadecimal $F$ ya que la comparación bit a bit de estos vectores siempre termina en una cantidad impar de unos.

Por último se realiza una operación de suma de las entras las cuales dan el resultado correcto, que además genera un bit de acarro al final, el cual forma parte de la operación.

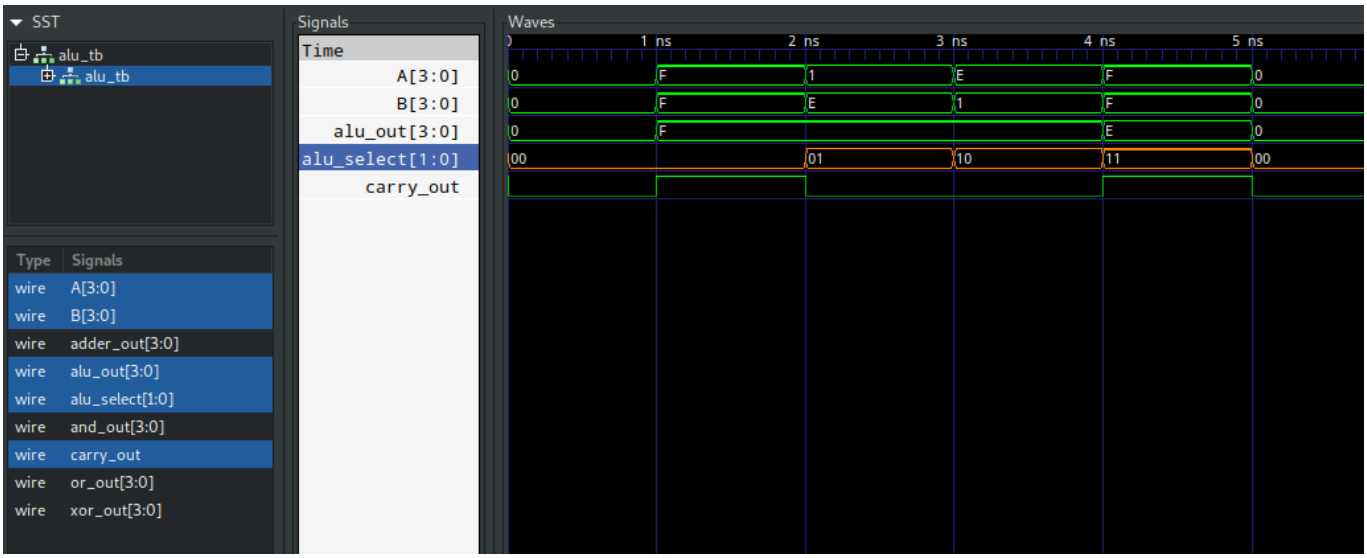


Figura 17: Resultados del ALU

# 5. Ejercicio Extra

Para esta sección fue propuesto expandir las funcionalidades del ALU descrito en el Ejercicio 2 con una compuerta NOT aplicada sobre el vector A y la aplicación de la resta de los vectores A y B, donde se asume que B siempre es positivo.

## 5.1. Compuerta NOT de 4 bits

Se construyó una compuerta *NOT* que recibe un vector de 4 bits y se aplica una negación por bit a cada uno, por lo que produce un vector negado, como se puede ver de la figura (18).

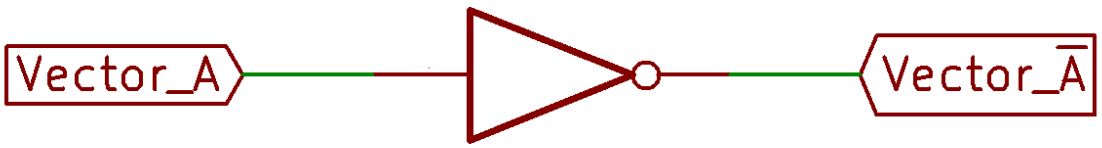


Figura 18: Compuerta NOT de 4bits

Una forma más clara de interpretar esta *NOT* es mediante su modelo estructural mostrado en la figura(19).

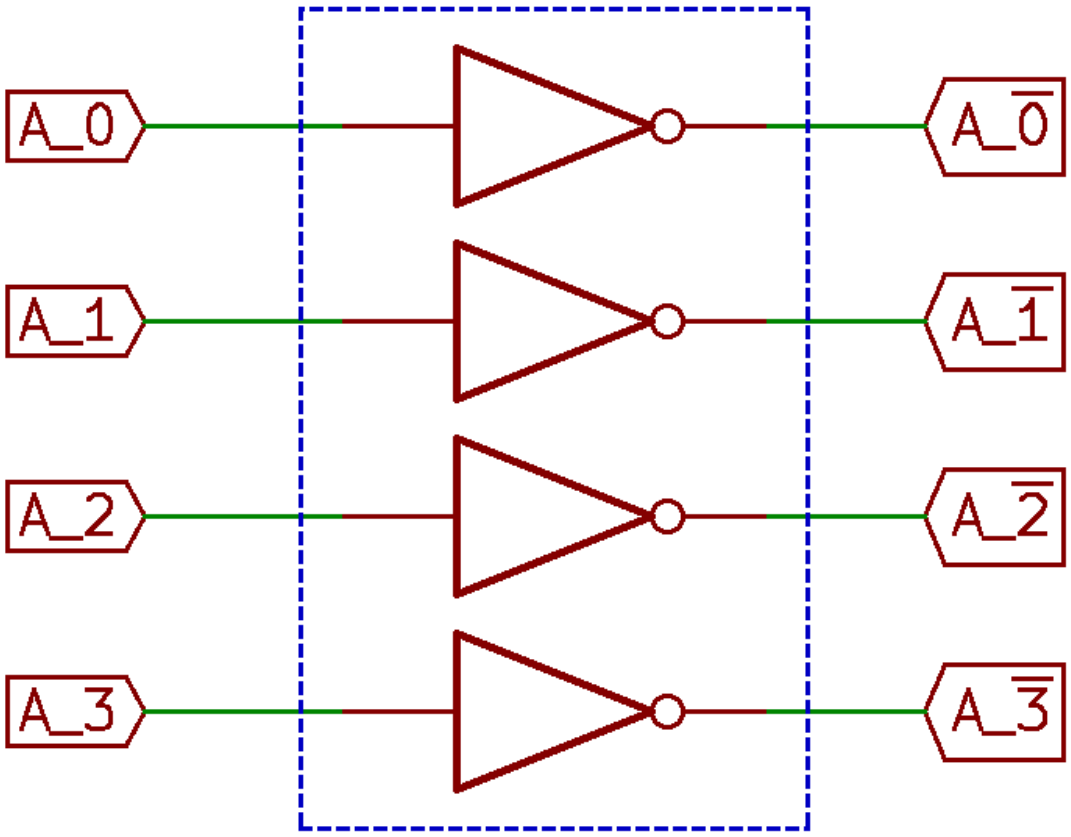


Figura 19: Compuerta NOT de 4bits, modelo estructural

Puede verse el código (7) implementado en verilog para construir la compuerta *NOT* de 4bits.

Listing 7: Compuerta NOT de 4bits

```
1    module NOT ( input [3:0] a, output [3:0] cout );
2       assign cout = ~a;
3    endmodule // NOT
```

Se aplicó unicamente un vector de prueba a esta compuerta ya que contenía los dos casos posibles para una NOT, como puede verse en la figura(20) con el diagrama de tiempos obtenido de toda la ALU.

## 5.2. Módulo Restador de 4 bits

Para crear el módulo restador es importante aclarar un puntos de partida importante, como lo es el tamaños de palabra que acepta el multiplexor, ya que el multiplexor es de 4bits y las palabras A y B se asumen como palabras [5]sin signo y al realizar la operación de resta sobre dos números es necesario aplicar el complemento en base 2 a uno de estos números(en este caso a B), por lo que es necesario ampliar el número a 5bits para agragar el signo, aplicar la operación de complemento en

[5]Se asumen sin signo por falta de especificación en la descripción del enunciado.

base 2 y devolverlo a 4bits.

Por lo que se creó un módulo(8) que permitiera obtener el complemento en base 2 de la palabra B con signo y luego éste se elimina para presentar el resultado, ya que el signo no puede ser incluido por limitaciones del multiplexor.

Listing 8: Módulo Complemento en base 2 de 5bits

```
1
2     module b2comp(input [4:0]a, output [4:0]cout);
3        assign cout =  -a;
4     endmodule // b2comp
```

Se crea un módulo *Restador* que calcúla utiliza el módulo de complemento en base 2 y posteriormente suma ambas palabras(A y B), como se muestra en el código creado en verilog (**??**).

Listing 9: Restador de dos números de 4 bits

```
1
2 module substractor(input [3:0] a, b, output [3:0]cout);
3
4    reg [4:0]signed_B;
5
6    wire [4:0] b2com_out;//2's complente of "b" with sign
7    wire carry_out;
8
9    b2comp b2com_(({1'b0, b}), (b2com_out));
10   fullAdder subst_((a), (b2com_out[3:0]), (1'b0), (cout), ...
        (carry_out));
11
12 endmodule // substractor
```

Es importante hacer notar además que se realizaron dos pruebas de resta en las que se presume $A > B$, en este caso el número resultante corresponde al número como tal de la resta y un segundo caso en el que se presume $A < B$, en tal caso es importante resaltar que el número resultante corresponde a la representación negativa del número(sin incluir el signo, por limitaciones de la plataforma construida), no se le aplicó en este caso el complemento en base 2 nuevamente al número, ya que esto pudo haberse considerado como un la aplicación de un tercer procedimiento(valor absoluto del número respuesta) y el problema se limitaba a la resta de dos números.

### 5.2.1. Caso $A > B$

Para este caso se eligieron los números 1101 y 1000 que al restarlos genera de forma correcta el número [6]0101.

[6]La operación en decimal sería 13-8=5

### 5.2.2. Caso $A < B$

En este otro caso se eligieron los números 0010 y 1001 que al aplicar la operación de resta, genera como resultado [7]1001 que es la representación negativa del resultado. Al aplicar nuevamente la operación de complemento en base 2 manualmente al resultado se obtiene el número 7 que corresponde al valor absoluto del resultado esperado, agregando el signo se obtiene $-7$ que corresponde a la respuesta correcta en representación decimal.

Ambos casos pueden verse en el diagrama de tiempo de la figura (20).

## 5.3. Modificaciones realizadas al MUX

Fue necesario modificar de manera importante la construcción del multiplexor ya que al ser necesarios dos casos extra la cantidad de bits de selección aumentaba a 8 en vez de 4, por lo que fue necesario modificar la descripción costruida para el multiplexor. En esta ocación se prefirió el uso del *CASE* como alternativa a las expresiones ternarias ya que simplificaba enormemente la interpretación del código construido como se puede ver en (10).

Listing 10: Multiplexor 8x1 de 4bits

```
1
2     module MUX(input [3:0] A0, A1, A2, A3, A4, A5, input ...
          [2:0]M_select, output reg [3:0]M_out);
3
4           always @(*) begin
5              case(M_select)
6                3'b000: M_out = A0;
7                3'b001: M_out = A1;
8               3'b010: M_out = A2;
9               3'b011: M_out = A3;
10              3'b100: M_out = A4;
11              3'b101: M_out = A5;
12             endcase // case (M_select)
13         end
14    endmodule // MUX
```

De forma particular a los diagramas de tiempo presentados en prácticamente todo el reporte, en este ejercicio se prefiere mostrar todos los vectores en su representación decimal en vez de la hexadecimal, para proder apreciar de forma más simple la operación NOT aplicada por bit, así como la resta.

No se alteraron los vectores anteriormente seleccionados, ya que enriquecían la verificación de la prueba realizada, ahora con un multiplexor 8x1 en ésta ALU, diferente a la ALU construida con un multiplexor 4x1, por lo que se puede observar su diagrama de tiempo en la figura (20).

[7] La operación en decimal sería 2-9 = -7


Figura 20: Diagrama de tiempos completo con módulos extra: NOT y Resta

Como se tiene ahora un multiplexor 8x1, quedan sobrantes dos casos, los cuales se prefirió ignorar, ya que si se redirigía a la operación cero, puede causar confusión con la operación AND, por lo que, lo que se consideró correcto era ignorar una selección mayor a la permitida de la cantidad de operaciones existentes(En éste caso 6 operaciones).

# Referencias

D. Dumani, *Proycto Verilog, Circuitos Digitales*, 2020.