Sistemas digitales: Práctica final

Marc Torres Torres
47409091L

Rubén Palmer Pérez
43474448D

Curso 2019-2020

Índice

1. Introduction

Para esta práctica se nos ha encomendado hacer dos ejercicios centrados en diferentes tipos de circuitos. Cada uno plantea un problema diferente y no están conectados de ninguna manera, esto significa que en esta práctica se pueden definir cada uno de los ejercicios por separado y así tener una diferenciación a la hora de plantear las soluciones para cada uno de ellos.

Esta breve introducción servirá para explicar de manera general de que trata cada uno de los ejercicios además de indicar nuestro acercamiento a cada uno de los problemas que puedan surgir a lo largo de la práctica. Además, en cada uno de los subapartados encontrados más adelante, se podrá discutir su acercamiento de manera general, para de esta manera poder ver otras posibles soluciones.

Generalmente, al enfrentar un problema desconocido, lo primero que se nos paso por la cabeza era hacer una sesión de"brainstorming" donde cada uno de nosotros propondríamos posibles soluciones. En estas no se buscaban la más óptimas o las más sencillas, sino las que se ajustasen a nuestros valores dentro de esta práctica. Para la parte combinacional, se encontró más interesante un acercamiento modular para así poder reciclar módulos de usuarios ya hechos. Este acercamiento nos pareció el más interesante, pues el circuito más óptimo no era tan legible y el más sencillo era extremadamente mejorable. Aclarar que llevar un circuito de manera modular no es la más óptima, pero si la más legible y por lo tanto mejor de arreglar o mejorar en caso de que sufra daños o que se quiera añadir nuevas funciones, respectivamente.

2. Combinacional

A continuación se irán explicando cada uno de los circuitos y sus funcionamientos. Cada circuito tiene un Hyperlink en "Figura X" para saltar a la imagen correspondiente en la descripción para mantener un flujo con un solo click.

2.1. Descripción de las partes identificadas

2.1.1. ADDER

Entradas:

- A3 ... A0: Número en C2 de 4 bits, que en conjunto llamaremos "A"
- B3 ... B0: Número en C2 de 4 bits, que en conjunto llamaremos "B"
- Cin: Carry in del adder

Salidas:

- Over: Parámetro que detecta el overflow
- S3 ... S0: Número en C2 de 4 bits, que en conjunto llamaremos "S"
- Cout: Carry out del circuito

Función: Proporcionado de todos los valores de entrada, sumará ambos valores bit a bit. Está implementado mediante Full adders lo que nos permite mediante una XOR detectar que hay overflow. Para comodidad del circuito se decidió crear una salida extra. El Cout, aunque no se llege a utilizar dentro del circuito es completamente funcional, como se puede ver en la Figura 1

2.1.2. FULL ADDER @

Entradas:

- A: Bit que llamaremos “A”
- B: Bit que llamaremos “B”
- Cin: Carry in del circuto

Salidas:

- S: Salida de A+B
- Cout: Carry out del circuito

Función: Proporcioando de todos los valores de entrada, sumará los bits A y B. En el caso de que la suma de overflow, ese valor será pasado por la salida Cout, de esta manera podemos poner varios de estos circuitos en serie para crear un sumador de varios bits. Este circuito ha sido implementado mediante Half Adders como se puede ver en la Figura 2

2.1.3. HALF ADDER @

Entradas:

- A: Bit que llamaremos “A”
- B: Bit que llamaremos “B”

Salidas:

- S: Salida de A+B
- Cout: Carry out del circuito

Función: Proporcionados A y b, los sumará y en caos de tener overflow será pasado por la salida Cout. Este circutio se comporta de manera parecida al Full Adder, sin embargo el Half Adder no tiene una entrada de Cin como se ve en la Figura 3

2.1.4. SUBTRACTOR

Entradas:

- A3 ... A0: Número en C2 de 4 bits, que en conjunto llamaremos "A"
- B3 ... B0: Número en C2 de 4 bits, que en conjunto llamaremos "B"

Salidas:

- S3 ... S0: Número en C2 de 4 bits

Función: Proporcionados todos los valores de entrada, restará ambos valores A y B bit a bit. El circuito de el restador se ha hecho siguiendo la linea de los apuntes del curso, invirtiendo la entrada B y sumando los valores A y B con el carry in a 1 como se puede ver en la Figura 4

2.1.5. NEG

Entradas:

- A3 ... A0: 4 bits, que en conjunto llamaremos "A"

Salidas:

- S3 ... S0: 4 bits , que en conjunto llamaremos "S"

Función: Proporcionados todos los valores de entrada, negará todos y cada uno de ellos de manera independiente. Este módulo es puramente estético, pues solo interesaba que fuese lo más compacto posible como se puede ver en Subtractor (Figura 4) y su circuiteria se puede encontrar en la Figura 5

2.1.6. MUX@

La @ expresa que hay mas de un circuito que comparte este tipo de entradas, salidas y función y este se puede cambiar por el número del circuito

Entradas:

- A0: Bit, que llamaremos "A0"
- A1: Bit, que llamaremos "A1"
- S: Bit selector del multiplexor

Salidas:

- O: Bit que en este caso puede ser "A0" o "A1"

Función: Proporcionados todos los valores de entrada, dependiendo de si el valor S es 0 o 1 hará que la salida O valga A0 o A1 respectivamente como se muestra en la Figura 6

2.1.7. BNNS TO 7SEG @

Entradas:

- A3 ... A0: Número en C2 de 4 bits, que en conjunto llamaremos "A"

Salidas:

- D6 ... D0: Codificación de A para su display en 7 segments

Función: Codifica el valor de A para poder ser representado con un 7 segment display como se puede ver en la Figura 7 . Este circuito fue totalmente creado por optimización usando los mapas de karnaugh y usando casos don't care, ya que solo nos interesa un display hasta el valor "8" para estos casos.

2.1.8. ABS C2 TO 7SEG

Entradas:

- A3 . . . A0: Número en C2 de 4 bits, que en conjunto llamaremos “A”

Salidas:

- D6 . . . D0: Codificación de A para su display en 7 segments

Función: Codifica el valor de A para poder ser representado con un 7 segment display como se puede ver en la Figura 7 . Este circuito fue totalmente creado por optimización usando los mapas de karnaugh y decidiendo poder representar valores negativos sin su signo como se puede ver en la Figura 8

2.1.9. EQUALS 0

Entradas:

- A3 . . . A0: Número de 4 bits, que en conjunto llamaremos “A”

Salidas:

- S: Valor que determinara si “A” es 0

Función: Dado el valor “A” determinara si es igual a 0000. Su circuito se puede ver en la Figura 9

2.1.10. FULL EQUALS

Entradas:

- A3 . . . A0: Número de 4 bits, que en conjunto llamaremos “A”
- B3 . . . B0: Número de 4 bits, que en conjunto llamaremos “B”

Salidas:

- S: Valor que determinara si “A” y “B” son iguales

Función: Dados los valores “A” y “B”, determinara si ambos son iguales. Su circuito se puede ver en la Figura 10

2.1.11. B MUX

Entradas:

- A4 ... A0: Número de 5 bits, que en conjunto llamaremos “A”
- B4 ... B0: Número de 5 bits, que en conjunto llamaremos “B”
- C4 ... C0: Número de 5 bits, que en conjunto llamaremos “C”
- D4 ... D0: Número de 5 bits, que en conjunto llamaremos “D”
- C1, C0: Valor de selección del multiplexor, que en conjunto llamaremos “C”

Salidas:

- R3 ... R0: Número de 3 bits, que en conjunto llamaremos “R”
- EX: Valor que nos dara, en el ámbito de abstracción, información extra de “R”

Función: Este multiplexor determinará la salida del circuito en su conjunto. Mediante los valores de selección “C”, se podrá elegir entre los valores “A”, “B”, “C” y “D”; valor que pasará a “R” y “EX”. Su circuito se puede ver en la Figura 11

2.2. Circuitos de las partes identificadas

2.2.1. ADDER

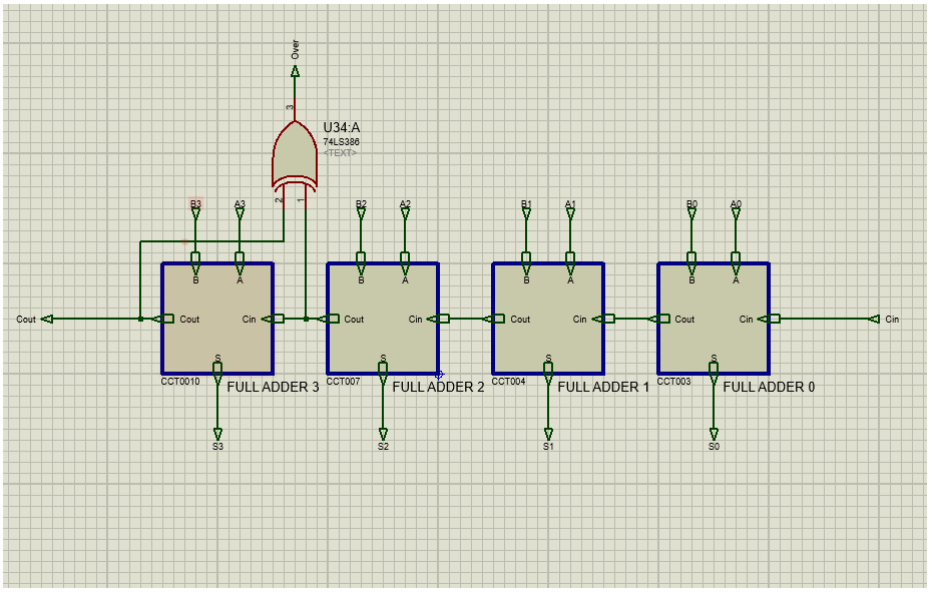


Figura 1: Adder

2.2.2. FULL ADDER @

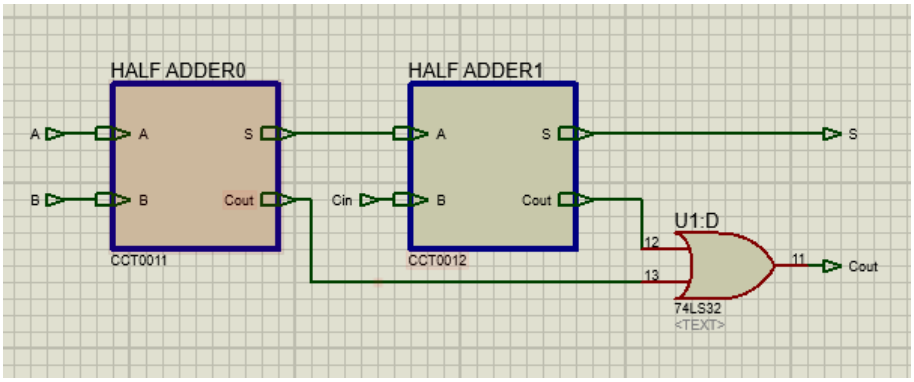


Figura 2: Full Adder

2.2.3. HALF ADDER @

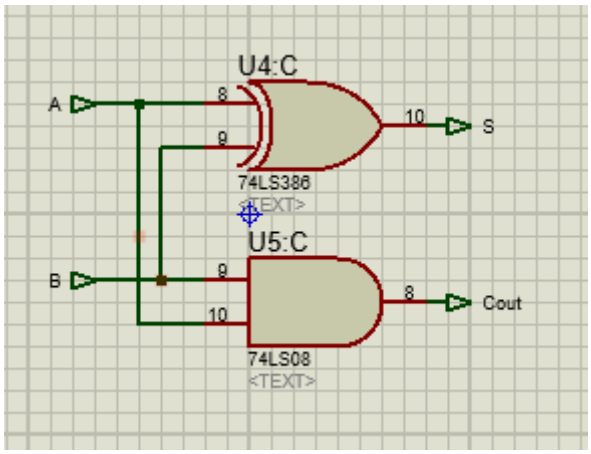


Figura 3: Half Adder

2.2.4. SUBTRACTOR

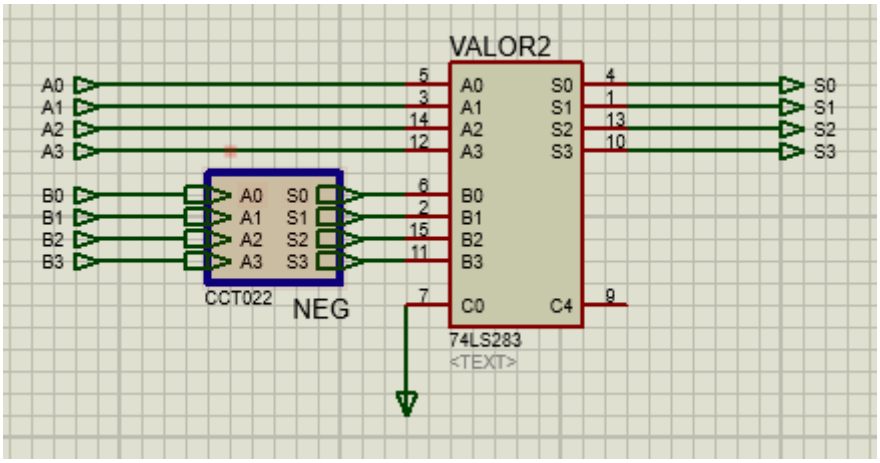


Figura 4: Subtractor

2.2.5. NEG

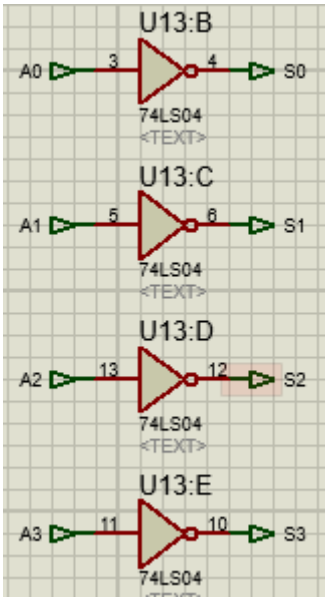


Figura 5: Neg

2.2.6. MUX@

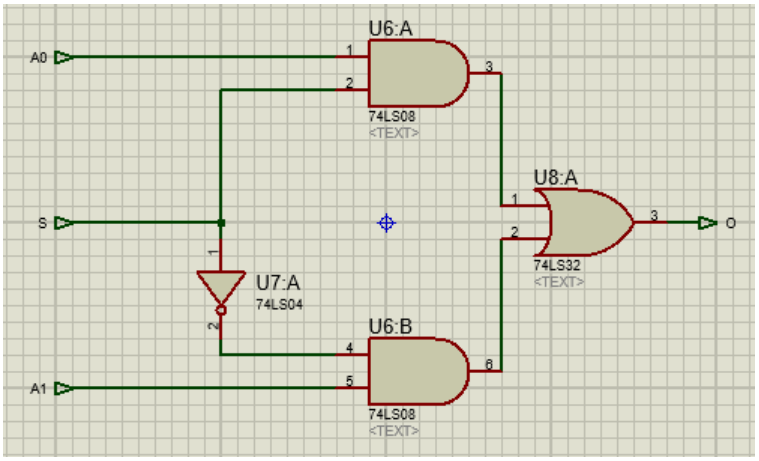


Figura 6: Mux

2.2.7. BNNS TO 7SEG @

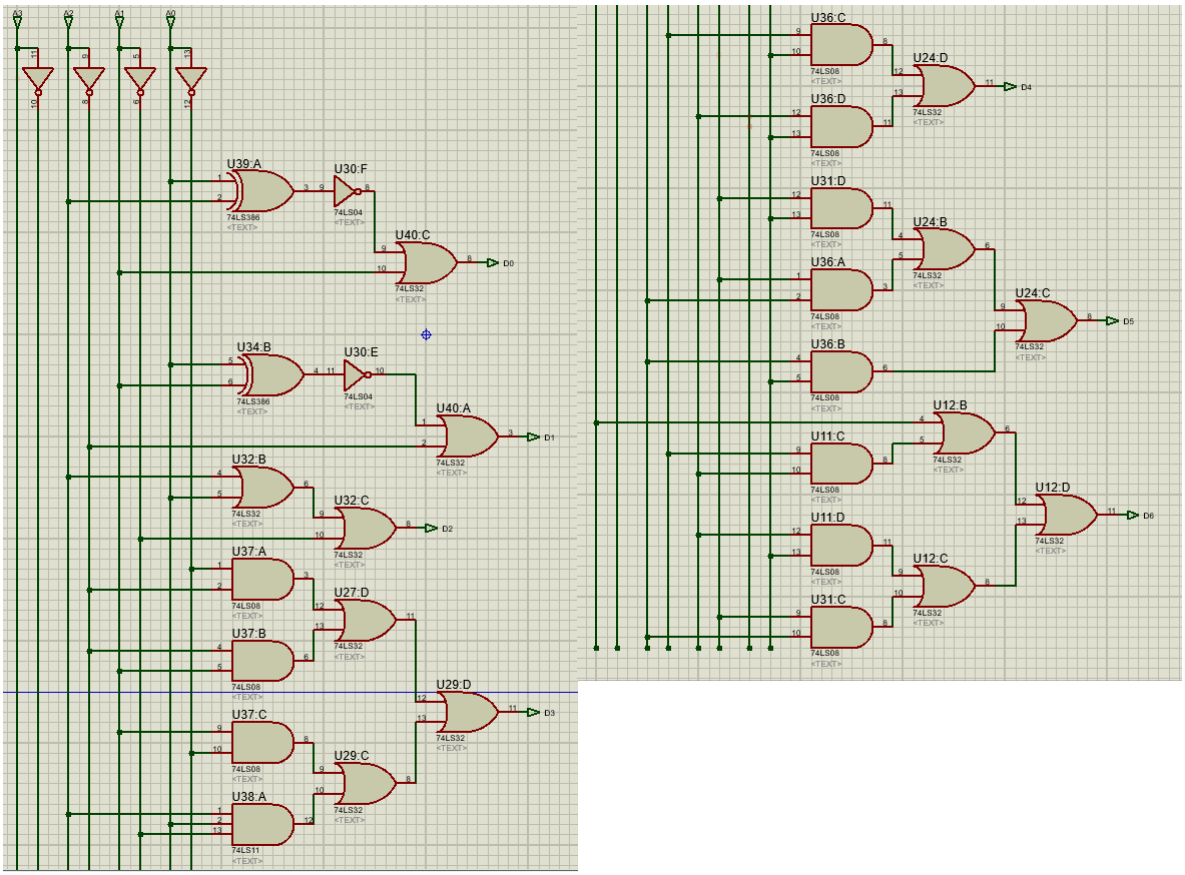


Figura 7: BNNS

2.2.8. ABS C2 TO 7SEG

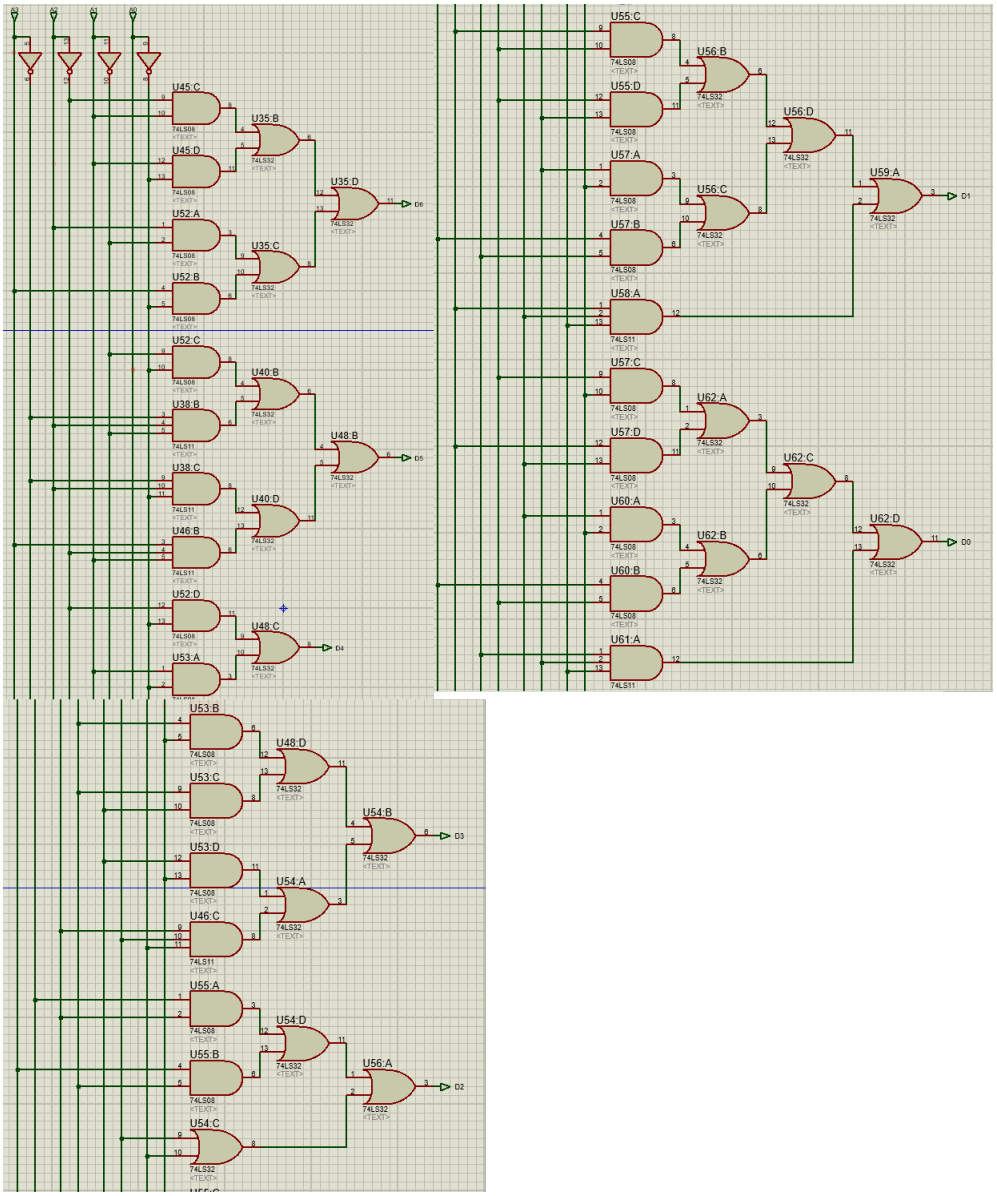


Figura 8: ABS

2.2.9. EQUALS 0

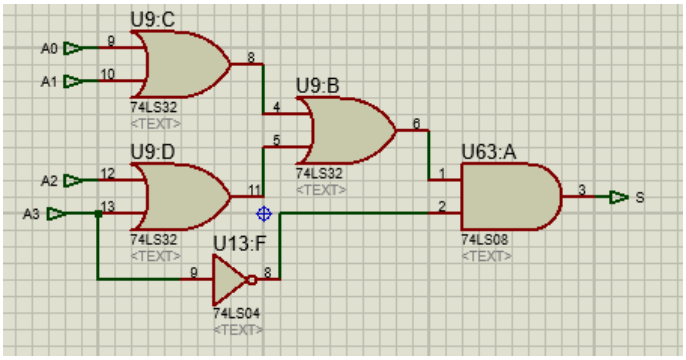


Figura 9: EQUALS

2.2.10. FULL EQUALS

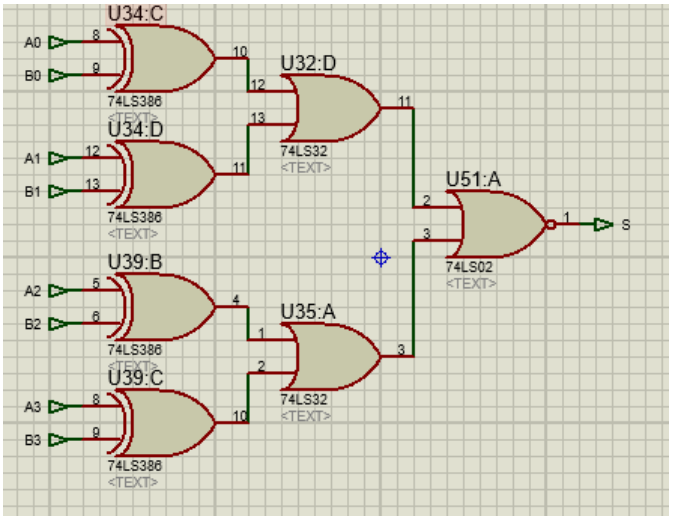


Figura 10: FULLEQUALS

2.2.11. B MUX

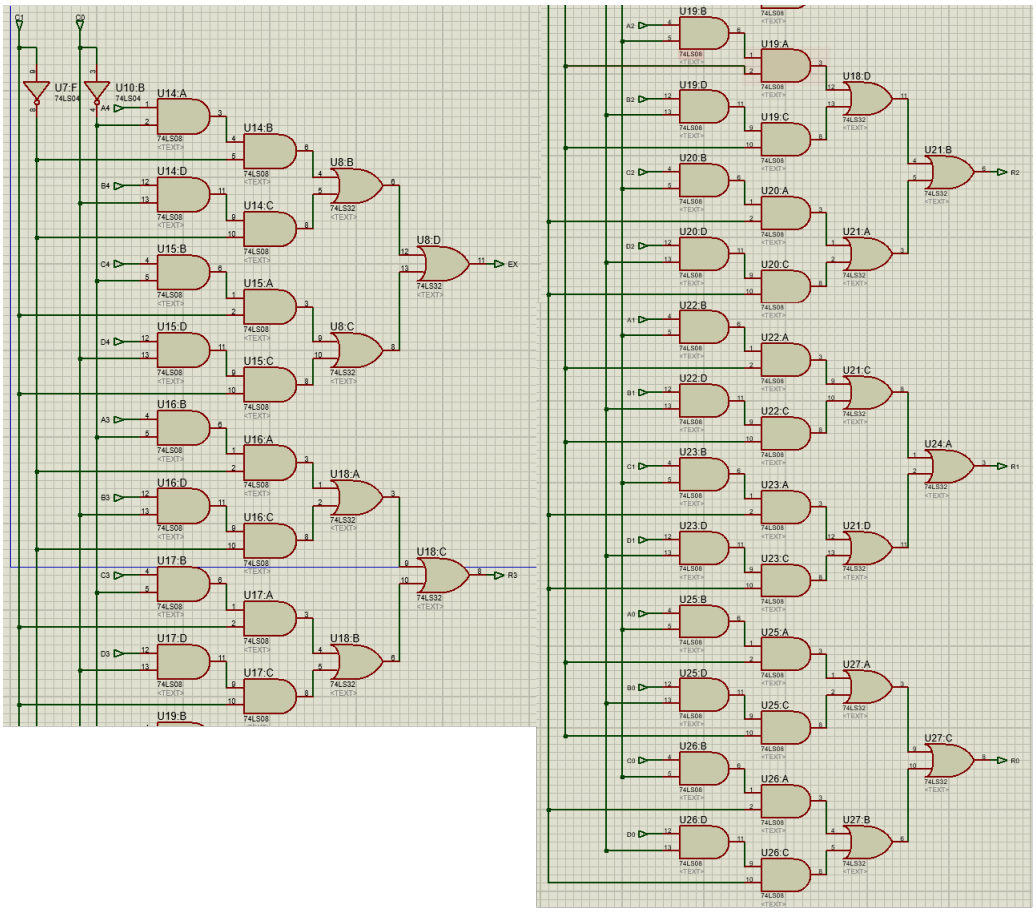


Figura 11: BMUX

2.3. Pruebas realizadas

Daremos un ejemplo para cada uno de los circuitos y, en el caso que tenga algúna entrada "especial" se tomara en cuenta para ver como cambian las salidas. Como valores generales se cogerán -8 para "A" y 2 para "B" que son, respectivamente, "1000" y "0010"

2.3.1. ADDER

En "A" tenemos el valor -8 (1000) y en "B" 2 (0010), Cin se mantiene en 0 para todo los casos del circuito. Mediante un conjunto de FULL ADDER se hace la suma de cada uno de los bits de "A" y "B" individualmente. En este caso "S" nos da (1010) que coincide con -6, el resultado que deberia dar. A la hora del display se hizo la decision de representar el valor absoluto de la suma pues el otro display dira si da overflow o no. Para probar que el overflow funciona, podemos tomar el valor 4 tanto para "A" como para "B". En este caso, la suma da 1000, que en BNSS representa la salida adecuada. Sin embargo, el valor 8 no se puede representar con 4 bits, el primer display nos representa que hay overflow con una raya horizontal. En el caso que haya overflow, el segundo display, el que representa el valor, es totalmente ignorado.

2.3.2. FULL ADDER @

En este caso, no tiene ningun sentido usar -8 y 2 ya que un FULL ADDER solo nos sirve para sumar valores de un solo bit. Si tomamos todas las entradas en 1, "A" + "B" seria 0, pero como tenemos un carryIn, S seria 1 y Cout seria tambien 1.

2.3.3. HALF ADDER @

El HALF ADDER sigue el mismo camino que el FULL ADDER sin embargo no tiene carry in, esa es la unica diferencia con el FULL ADDER, por lo tanto podemos hacer la misma prueba sin el carryIn. "A" + "B" seria 0 con un carryOut de 1.

2.3.4. SUBTRACTOR

EL SUBTRACTOR se comporta como un ADDER, sin embargo el carryIn siempre sera 1 y "B" estará negado. Para nuestros valores se comportaria, esencialmente como un 1000 + 1101 + 0001 y el resultado nos daria overflow ya que -10 no se puede representar con 4 bits.

2.3.5. NEG

No hay mucho que se pueda hacer con NEG ya que simplemente niega todos los bits de la entrada. Si tomamos -8 nos devolveria 0111 y si le damos 2 nos devoleria 1101

2.3.6. MUX @

MUX @ Se comportaría como un multiplexor de un solo bit de selección, dependiendo de este bit, la salida será “A” o “B”.

2.3.7. BNSS TO 7SEG @

BNSS TO 7SEG funciona meramente como un codificador para que el display funcione como nosotros queremos, en este caso nos permite decodificar un valor BNSS para que pueda ser representado en un Display. Cabe decir, que solo se tuvieron en cuenta los valores del 0 al 8. Si diesemos -8, el circuito representaria un 8 en el display.

2.3.8. ABS C2 TO 7SEG

ABS C2 TO 7SEG Es una version mas complicada de BNSS TO 7SEG ya que tiene en cuenta el signo. Se decidió usar esta versión del circuito al final para no hacer todo el proceso previo necesario para que BNSS TO 7SEG funcione correctamente. Al igual que su circuito más simple, solo toma valores del 0 al 8. En el caso que dieses -8 el display daria 8.

2.3.9. EQUALS 0

Este circuito nos permite identificar si un valor es 0000 o no. Si le pasamos el valor 1000, el circuito nos daria un 0. En cambio, si le damos un 0000 nos devolvera un 1.

2.3.10. FULL EQUALS

Este circuito nos permite mirar si dos valores son exactamente iguales. Para ello se hace un recorrido de cada uno de sus bits y mirando si son iguales. Si le pasamos un 1000 y un 0010 el circuito nos devolvera un 0. Si le damos 1110 y 0010 nos seguirá dando 0 aunque los valores absolutos sean iguales. Solo cuando demos dos valores iguales, 1010 y 1010, será cuando el circuito nos devolvera un 1.

2.3.11. B MUX

Al igual que MUX @, dependiendo de unos valores de selección nos permitirá elegir una salida de las 4, en este caso, posibles. Si los bits de seleccion son 00, nos devolverá el valor “A”, 01 el “B”, 10 el “C” y por último 11 el “D”,

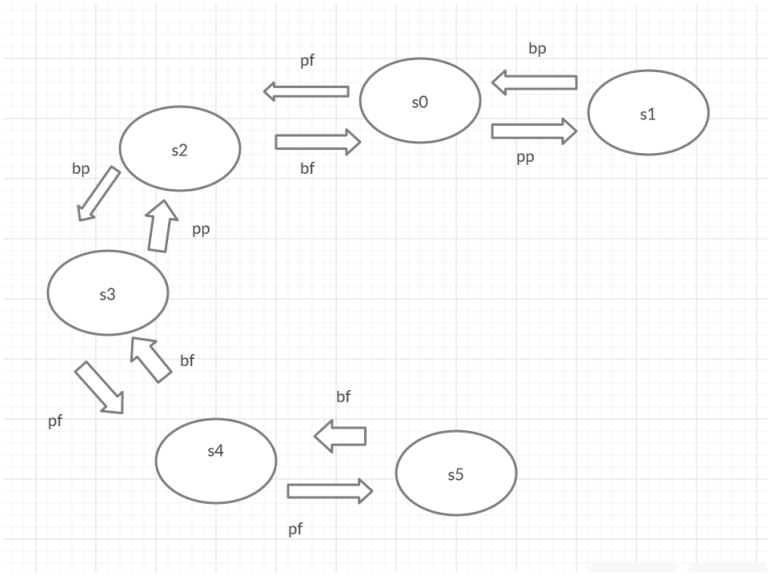
2.3.12. CIRCUITO

Para los valores -8 y 2 iremos mirando cuales son todos los posibles resultados del circuito. Para 00 el circuito nos devuelve la suma de -8 y 2, en el display podemos ver el resultado y observamos que no nos da iverflow y el resultado, aunque este en valor absoluto, nos da 6. Probando 4 y 4, observamos que nos da overflow y el resultado da 8. Para 01 nos tiene que devolver el valor absoluto de A, en el caso de -8 el circuito nos devuelve un 8, el segundo display, que nos muestra el valor de EX queda ignorado. Para 10 nos tiene que informar si el valor absoluto de A es mayor que el valor absoluto de B, para el caso de -8 y 2 el circuito nos devuelve un 1 en el valor EX que queda plasmado en el primer display ocmo una barra horizontal. Para 11 nos tiene que informar si A == B, para -8 y 2, el circuito no nos muestra nada. Sin embargo, si cambiamos -8 por 2, vemos que el primer display nos enseña una barra horizontal, dandonos a entender que A == B.

3. Secuencial

3.1. Diagrama de estados

Para que el diagrama fuese mucho más conciso, solo se muestran las entradas que hacen un cambio en el circuito. Es decir, si no se muestra una flecha λ de S0 significa que esta formando un lazo consigo mismo.



3.2. Codificación de estados

Se debe tener en cuenta que necesitamos de 3 bits para codificar nuestras 6 entradas ya que con solo 2 bits solo podemos codificar 4 estados y eso es insuficiente.

qo Totalmente horizontal	000
q1 Pies elevados frontal horizontal	001
q2 Pies horizontal y frontal a nivel 1	010
q3 Pies bajados y frontal a nivel 1	011
q4 Pies bajados y frontal a nivel 2	100
q5 Pies bajados y frontal a nivel 3	101

Codificamos las 5 entradas en nuestra silla.

No hacer nada (λ)	000
Subir frontal (FP)	001
Bajar frontal (FB)	010
Subir pies (PP)	011
Bajar pies (PB)	100

3.3. Justificación del tipo de máquina

Para la parte secuencial de nuestro trabajo usaremos una máquina de tipo Mealy puesto que al tener que representar cómo funciona una silla, esta depende del estado actual y de la entrada en la que se encuentre.

Esto se ve claramente con el siguiente caso; Si estamos en el estado q2 (pies horizontales y frontal a nivel 1) y nos entra un subir frontal, no haremos nada puesto que no existe el estado pies horizontal y frontal a nivel 2.

3.4. Tabla de trasición

q2	q1	q0	x2	x1	x0	D2	D1	D0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	1	0
0	0	0	0	1	0	0	0	0
0	0	0	0	1	1	0	0	1
0	0	0	1	0	0	0	0	0
0	0	0	1	0	1	x	x	x
0	0	0	1	1	0	x	x	x
0	0	0	1	1	1	x	x	x
0	0	1	0	0	0	0	0	1
0	0	1	0	0	1	0	0	1
0	0	1	0	1	0	0	0	1
0	0	1	0	1	1	0	0	1
0	0	1	1	0	0	0	0	0
0	0	1	1	0	1	x	x	x
0	0	1	1	1	0	x	x	x
0	0	1	1	1	1	x	x	x
0	1	0	0	0	0	0	1	0
0	1	0	0	0	1	0	1	0
0	1	0	0	1	0	0	0	0
0	1	0	0	1	1	0	1	0
0	1	0	1	0	0	0	1	1
0	1	0	1	0	1	x	x	x
0	1	0	1	1	0	x	x	x
0	1	0	1	1	1	x	x	x
0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	0	0
0	1	1	0	1	0	0	1	1
0	1	1	0	1	1	0	1	0
0	1	1	1	0	0	0	1	1
0	1	1	1	0	1	x	x	x
0	1	1	1	1	0	x	x	x
0	1	1	1	1	1	x	x	x
1	0	0	0	0	0	1	0	0
1	0	0	0	0	1	1	0	1
1	0	0	0	1	0	0	1	1
1	0	0	0	1	1	1	0	0
1	0	0	1	0	0	1	0	0
1	0	0	1	0	1	x	x	x
1	0	0	1	1	0	x	x	x
1	0	0	1	1	1	x	x	x
1	0	1	0	0	0	1	0	1
1	0	1	0	0	1	1	0	1
1	0	1	0	1	0	1	0	0
1	0	1	0	1	1	1	0	1
1	0	1	1	0	0	1	0	1
1	0	1	1	0	1	x	x	x
1	0	1	1	1	0	x	x	x
1	0	1	1	1	1	x	x	x
1	0	0	0	0	0	x	x	x
1	0	0	0	0	1	x	x	x
1	0	0	0	1	0	x	x	x
1	0	0	0	1	1	x	x	x
1	1	0	1	0	0	x	x	x
1	1	0	1	0	1	x	x	x
1	1	0	1	1	0	x	x	x
1	1	0	1	1	1	x	x	x
1	1	1	0	0	0	x	x	x
1	1	1	0	0	1	x	x	x
1	1	1	0	1	0	x	x	x
1	1	1	0	1	1	x	x	x
1	1	1	1	0	0	x	x	x
1	1	1	1	0	1	x	x	x
1	1	1	1	1	0	x	x	x
1	1	1	1	1	1	x	x	x

3.5. Minimización

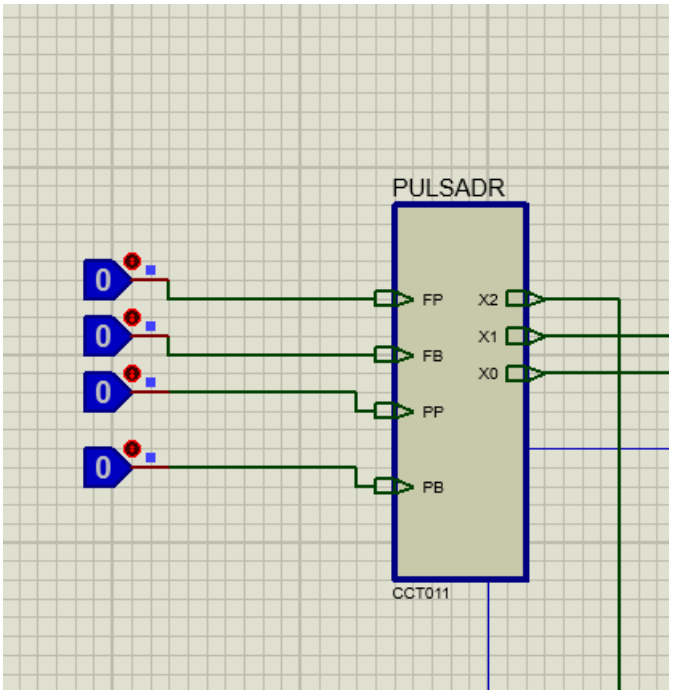
Como los Flip-Flop son de tipo D, podemos coger el estado siguiente (qy') como nuestro valor para cada uno de nuestros Flip-Flop. De esta manera obtenemos las funciones:

- D1: $Y = BD + AC\overline{E} + A\overline{E}F + \overline{B}CF + \overline{A}C\overline{D}F + \overline{A}\overline{B}EF + A\overline{C}E\overline{F}$
- D2: $Y = B\overline{C}\overline{E} + BEF + BC\overline{F} + \overline{A}C\overline{E}F + A\overline{C}E\overline{F}$
- D3: $Y = A\overline{B} + AF + AC + BC\overline{E}F$

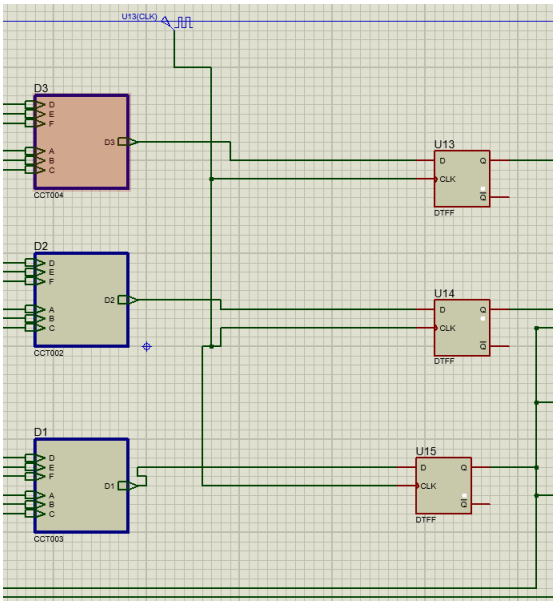
3.6. Implementación

En orden de completar esta práctica hemos tenido que implementar todo el material anterior (tablas de verdad, karnaugh..) usando proteus de la manera siguiente:

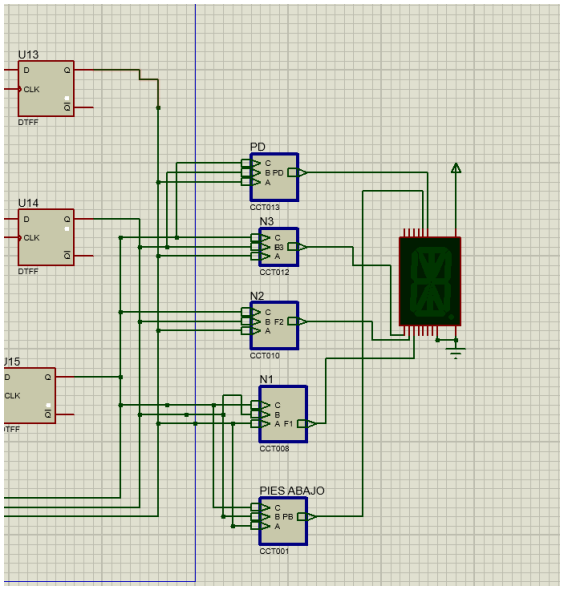
Presentamos nuestro programa con 4 Logic Toggles que simbolizan los 4 botones que tiene el mando de la silla que se nos ha mandado programar, estas 4 entradas se transforman en 3 bits siguiendo un simple mapa de karnaugh hecho teniendo mente el hecho de que según nuestra codificación el botón FP se traducirá en 001 y así con el resto de botones.



Luego, siguiendo los mapas de karnaugh obtenidos anteriormente hemos creado nuestros Flip/-Flops de tipo D, estas implementaciones de los mapas de karnaugh están ocultos en hojas hijas para mejorar la legibilidad de nuestro programa, cabe destacar la retroalimentación de las entradas que caracteriza a los sistemas secuenciales, recordemos también que necesitamos 3 Flip/Flops de clase D ya que tenemos 3 bits para codificar los estados.

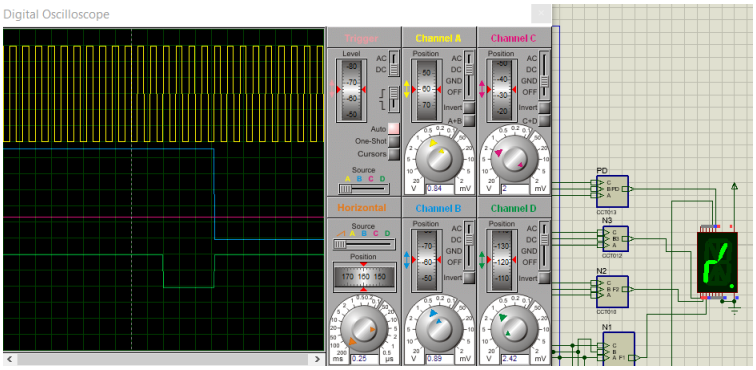


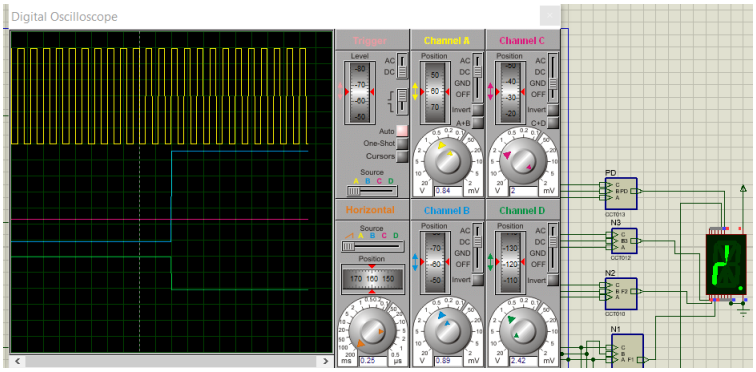
El resultado de este programa se puede entender como la visualización de la silla según los estados que se accionen en nuestros Logic States, para lograr esto hemos tenido que codificar mediante mapas de karnaugh las diferentes partes de los Displays, un ejemplo de esto sería si las 3 d' s son 000 (estado codificado anteriormente) se debe visualizar la silla totalmente horizontal, como en la parte anterior hemos decidido ocultar los circuitos que codifican dichas visualizaciones en hojas hijas para mayor comodidad.



3.7. Juego de pruebas y cronograma de las salidas para un ejemplo

Recordemos que por teoría hemos visto que usando un flip flop de clase D, este solo puede cambiar en el momento en el que el clock vale 1, así pues vemos como nuestras capturas cumplen dicha condición. En este caso, la linea amarilla representa nuestro clock.





4. Conclusiones

Durante todo el desarrollo de este proyecto hemos adquirido una gran canitad de conocimiento y sobre todo habilidad de movimiento dentro de Proteus. Aunque hayamos tenido problemas en algunos puntos contados, en conjunto creemos que la practica ha sido una gran oportunidad para ponerte a prueba y ver cuan bien puedes hacerlo.

Cabe mencionar que no todo fue proteus, sino tambien optimización de tiempo, esfuerzo y funciones lógicas. Hemos tenido que lidiar con la manera de pensar del otro para que nuestros circuitos sean faciles de leer entre autores y poder sacar el mejor circuito que nos agradase a los dos a la vez.

Ademas, ha sido una buena oportunidad para ver y usar un sistema de versiones como Github, donde se fue documentando cada versión. Y por parte de la memoria, el uso de LaTeX lo que nos obligo a hacer una memoria a parte a "sucio" y luego pasarla a un formato más agradable.

En conjunto, ha sido una experiencia que nos ha puesto a prueba y sobre todo nos a ayudado a ver como funcionan este tipo de trabajos.