	Carátula para entrega de prácticas
Facultad de Ingeniería	Laboratorios de docencia

Laboratorio

Profesor: ING. JULIO CESAR CRUZ ESTRADA

Asignatura: LABORATORIO DE ORGANIZACIÓN Y ARQUITECTURA DE COMPUTADORAS

Grupo: 07

No de Práctica: 7

Integrante(s): Jiménez Treviño Emilio Cristóbal

Martinez Perez Brian Erik

No. de Equipo de cómputo empleado: s/n

Semestre: 2026-1

Fecha de entrega: 18/11/2025

Observaciones:

CALIFICACIÓN: _____

Práctica 7. Procesador CISC 68HC11

Objetivo

Diseñar un microprocesador CISC de 8 bits, específicamente un ’clon’ del microprocesador 68HC11 de Motorola R .

Introducción

En esta práctica se estudió el funcionamiento interno de un procesador CISC de 8 bits, específicamente un clon del microprocesador Motorola 68HC11, con el propósito de comprender su arquitectura, sus modos de direccionamiento y la ejecución de instrucciones a nivel de micro-operaciones. Para ello, se descargó y analizó el proyecto base en Quartus, el cual permite modificar la memoria ROM del secuenciador e implementar nuevas instrucciones. A partir de este entorno, se desarrollaron diversos ejercicios orientados a agregar instrucciones, manipular registros internos, gestionar banderas de condición y ejecutar algoritmos simples mediante código máquina.

Desarrollo

Para comenzar primero se descargo el proyecto de Quartus del siguiente repositorio de GitHub: https://github.com/JulioCCruzE/OAC_CISC

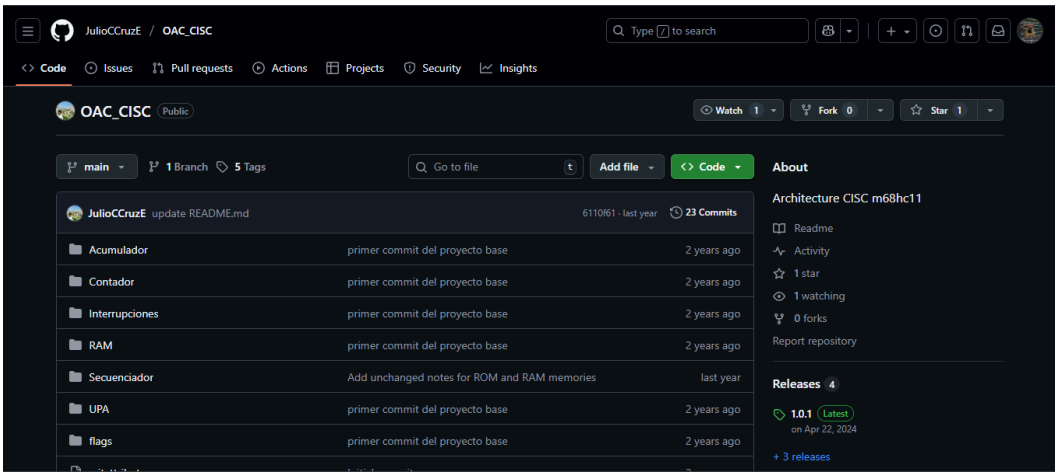

Imagen 1.1 - repositorio del proyecto de Quartus Github.

Una vez que descargamos el proyecto lo descomprimimos y abrimos el proyecto para abrirlo con Quartus y compilarlo en opción de “Start Analysis & Synthesis”.

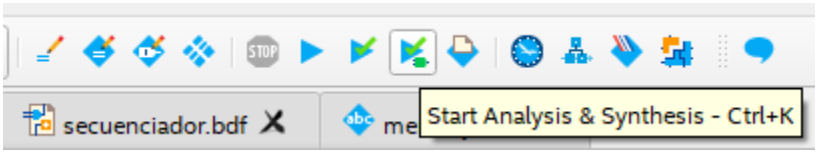

Imagen 1.2 - compilación con opción “Start Analysis & Synthesis”.

Si podemos ejecutarlo y no genera errores, significa que podemos implementar nuestra propias instrucciones en la memoria “rom” y cargar códigos para que el procesador los ejecute.

Ejercicio 1

Para el primer código solo se agrego la instrucción “LAAB” con direccionamiento inmediato, aunque ya estuviera esta instrucción, solo agregamos una más para poder

saber como se agregan más instrucciones al memoria “ROM”, además se tomó en cuenta que teníamos que cambiar el código de operación de la instrucción que recién agregamos para evitar tener problemas.

```
--LDAB (INM)
elsif(dir= X"B60") then data <= "00000000000000000000010010010000000000011011100001110001110000110000111111000000000000000000000010";
elsif(dir= X"B61") then data <= "00000000000000000000010100010000000000011011100001110001110011111110100000000000000000000000010";
elsif(dir= X"B62") then data <= "01111111000000000000010010010000000000011011100001110001110001111111000000010010100011100010";
elsif(dir= X"B63") then data <= "11000001000000001001100100100000000000011011100001110001110000110000111111000000000000000000000010";
```

Imagen 1.3 - contenido de la instrucción LDAB inmediato.

El primer ejercicio que se realizó fue la suma del acumulador A, más el acumulador B, y el resultado de la suma lo guardamos en el acumulador B. La lógica del programa fue la siguiente.

Pseudocódigo:

```
B=2
A=0
While (1){
        A = A+B
}
```

contenido de la memoria de programa:

```
0   :   B6; --LDAB inm
1   :   02;
2   :   86; --LDAA inm
3   :   00;
4   :   1B; -- ABA inh A+B
5   :   7E; -- JMP Ext
6   :   00;
7   :   04;
8   :   86; -- LDAA dir
9   :   FF;
[10..255]   :   00;
END;
```

El contenido de la memoria del programa se guarda en el archivo del proyecto llamado “mem_content.mif”.

Ejercicio 2

Para el segundo código se implementó el algoritmo en el que se comparan los valore cargados en los registros A y B, si los registros tienen diferente valor en el registro A se

guarda un “0B”, si los registros tienen el mismo valor entonces en el aculuador A se carga el valor “06”.

Para poder implementar este algoritmo fue necesario implementar las instrucciones “CMP” y “JE”. La instrucción CMP realiza la resta de A-B, pero no se guarda el resultado, simplemente se realiza la resta para actualizar las banderas de estado. La instrucción JE, realiza un salto condicional, tomando en cuenta el valor de la bandera “Z”, si z es uno significa que el valor de la operación realizada es 0.

```
--CMP (INH)
elsif(dir= X"1A0") then data <= "00000000000000000000011111110000010000111111100001110001110001110001111111000000000000000000000011";
elsif(dir= X"1A1") then data <= "01111111000000000000100100100000000000000111000011100011100011100011111110000000000000001111000010";
elsif(dir= X"1A2") then data <= "11000001000000001001100100100000000000011011100001110001110000110000111111100000000000000000000000010";
```

Imagen 1.4 - contenido de la instrucción CMP inherente.

```
--JE (INH)
elsif(dir= X"7F0") then data <= "00000000000000000000100100100000000000011011100001110001110000110000111111000000000000000000000010";
elsif(dir= X"7F1") then data <= "00000000000000000000100100100000000000011011100001011001110001110011111101100000000000000000000010";
elsif(dir= X"7F2") then data <= "00000000000000000000100100100000000000011011100001110001110000110000111111000000000000000000000010";
elsif(dir= X"7F3") then data <= "10010001011111110101100100100000000000011011100001100111110001110011111101000000000000000000000010";
elsif(dir= X"7F4") then data <= "01111111000000000000100100100000000000011011100001000001110001001011111111000000000000000000000010";
elsif(dir= X"7F5") then data <= "11000001000000001001100100100000000000011011100001110001110000110000111111000000000000000000000010";
```

Imagen 1.5 contenido de la instrucción JE extendido.

Pseudocódigo:

```
B = valor 1
A = valor 2
if (){
        A=0B
}
A=06
```

contenido de la memoria de programa:

```
–Ejercicio 2
0   :  B6; LDAB inm
1   :  02;
2   :  86; LDAA inm
3   :  02;
4   :  1A; CMP inh A-B
5   :  7F; JE inh
6   :  00; 16 BITS
7   :  0A;
8   :  86; LDAA inm
```

9 : 0B;
10 : 86; LDAA inm
11 : 06;
[12..255] : 00;
END;

El contenido de la memoria del programa se guarda en el archivo del proyecto llamado “mem_content.mif”.

Ejercicio 3

Para el tercer ejercicio realizamos lo mismo de comparar los registros A y B, solo que ahora vamos a cargar el valor del resultado en memoria del programa. por lo que necesitamos cargar todos los bits de los registros A y B, para poder implementar este algoritmo fue necesario implementar las instrucciones “STAA”, “LDAA” y “LDAB” en su formato extendido.

```
--STAA (EXT)
elsif(dir= X"A70") then data <= "00000000000000000000100100100000000000011011100001110001110000110000111111000000000000000000000010";
elsif(dir= X"A71") then data <= "00000000000000000000100100100000000000011011100001011001110001110011111101100000000000000000000010";
elsif(dir= X"A72") then data <= "00000000000000000000100100100000000000011011100001110001110000110000111111000000000000000000000010";
elsif(dir= X"A73") then data <= "00000000000000000000100100100000000000011011100001100111100011100111111010000000000000000000000010";
elsif(dir= X"A74") then data <= "00000000000000000000100100100000000000011011100001100011100011100011111100000000000000000000000010";
elsif(dir= X"A75") then data <= "00000000000000000000100101100000000000011011100001110001110001110001111100000000000000000000000010";
elsif(dir= X"A76") then data <= "01111111000000000000100100100000000000011011100001110001110001110001111111000000001001100011100010";
elsif(dir= X"A77") then data <= "11000001000000001001100100100000000000011011100001110001110000110000111111000000000000000000000010";
```

Imagen 1.6 contenido de la instrucción STAA extendido.

```
--LDAA (EXT)
elsif(dir= X"960") then data <= "00000000000000000000100100100000000000011011100001110001110000110000111111000000000000000000000010";
elsif(dir= X"961") then data <= "00000000000000000000100100100000000000011011100001011001110001110011111101100000000000000000000010";
elsif(dir= X"962") then data <= "00000000000000000000100100100000000000011011100001110001110000110000111111000000000000000000000010";
elsif(dir= X"963") then data <= "00000000000000000000100100100000000000011011100001100111100011100111111010000000000000000000000010";
elsif(dir= X"964") then data <= "00000000000000000000100100100000000000011011100000110001110001110001111111000000000000000000000010";
elsif(dir= X"965") then data <= "00000000000000000000100101000000000000011011100001110001110001110001111101000000000000000000000010";
elsif(dir= X"966") then data <= "01111111000000000000100100100000000000011011100001110001110001110001111111000000001001100011100010";
elsif(dir= X"967") then data <= "11000001000000001001100100100000000000011011100001110001110000110000111111000000000000000000000010";
```

Imagen 1.7 contenido de la instrucción LDAA extendido

```
--LDAB (EXT)
elsif(dir= X"D60") then data <= "00000000000000000000100100100000000000011011100001110001110000110000111111000000000000000000000010";
elsif(dir= X"D61") then data <= "00000000000000000000100100100000000000011011100001011001110001110011111101100000000000000000000010";
elsif(dir= X"D62") then data <= "00000000000000000000100100100000000000011011100001110001110000110000111111000000000000000000000010";
elsif(dir= X"D63") then data <= "00000000000000000000100100100000000000011011100001100111100011100111111010000000000000000000000010";
elsif(dir= X"D64") then data <= "00000000000000000000100100100000000000011011100000110001110001110001111111000000000000000000000010";
elsif(dir= X"D65") then data <= "00000000000000000000101000100000000000011011100001110001110001110001111101000000000000000000000010";
elsif(dir= X"D66") then data <= "01111111000000000000100100100000000000011011100001110001110001110001111111000000001001100011100010";
elsif(dir= X"D67") then data <= "11000001000000001001100100100000000000011011100001110001110000110000111111000000000000000000000010";
```

Imagen 1.8 contenido de la instrucción LDAB extendido.

Pseudocódigo:

```
Temp  = valor 1
Res = valor 2
if (res != temp) {
        Res=0B
}
Res = 06
```

contenido de la memoria de programa:

```
– ejercicio 3
0 : D6; LDAB ext
1 : 00;
2 : 17;
3 : 96; LDAA ext
4 : 00;
5 : 18;
6 : 1A; CMP inh A-B
7 : 7F; JE Ext
8 : 00;
9 : 0F;
10 : 86; LDAA imm
11 : 0B;
12 : A7; STAA ext
13 : 00;
14 : 18;
15 : 86; LDAA imm 0FH
16 : 06;
17 : A7; STAA ext
18 : 00;
19 : 18;
20 : 7E; JMP ext 14H
21 : 00;
22 : 14;
23 : 0E; temp 17H
24 : 0E; res 18H
[25..255] : 00;
END;
```

El contenido de la memoria del programa se guarda en el archivo del proyecto llamado “mem_content.mif”.

Ejercicio 4

Para el cuarto y último ejercicio se implementó el algoritmo que obtiene el menor de los números de la lista de tamaño 5, y coloca el resultado en la variable “Res”.

Para solucionar este problema tuvo que implementar una instrucción adicional a las que ya teníamos. La instrucción que agregamos fue “JB”, la cual salta si la bandera de “N” está prendida, en este caso nos ayudó a saber si un número era menor que el otro.

Además como no pudimos implementar un ciclo for para recorrer la lista, tuvimos que comparar todos los valores de la lista con el primer valor e ir actualizando el valor de RES siempre que se encontrara un número menor.

```
--JB (INH)
elsif(dir= X"7B0") then data <= "00000000000000000000100100100000000000011011100001110001110000110000111111000000000000000000000010";
elsif(dir= X"7B1") then data <= "00000000000000000000100100100000000000011011100001011001110001110011111101100000000000000000000010";
elsif(dir= X"7B2") then data <= "00000000000000000000100100100000000000011011100001110001110000110000111111000000000000000000000010";
elsif(dir= X"7B3") then data <= "10011001011110110101100100100000000000011011100001100111110001110011111101000000000000000000000010";
elsif(dir= X"7B4") then data <= "01111111000000000000100100100000000000011011100001000001110001001011111111000000000000000000000010";
elsif(dir= X"7B5") then data <= "11000001000000001001100100100000000000011011100001110001110000110000111111000000000000000000000010";
```

Imagen 1.9 contenido de la instrucción JB inherente.

Pseudocódigo:

```
lista[0..4]
Res=lista[0]
Temp=lista[1]
if Temp < Res {
        Res = Temp
}

Temp=lista[2]
if Temp < Res {
        Res = Temp
}

Temp=lista[3]
if Temp < Res {
        Res = Temp
}

Temp=lista[4]
if Temp < Res {
        Res = Temp
}
```

contenido de la memoria de programa:

```
-- ejercicio 4
0: 96; --LDAA
1: 00; --TEMP1
2: 35;

3: D6; -- LDAB
4: 00; --TEMP1
5: 35;
6: 1A; -- CMP (inh)
7: 7B; -- JB
8: 00; --SALTO
9: 0D;
10: 96; -- LDAB
11: 00; --TEMP1
12: 35;

13: D6; -- LDAB
14: 00; --TEMP2
15: 36; --
16: 1A; -- CMP (inh)
17: 7B; -- JB
18: 00; --SALTO2
19: 17; --SALTO2
20: 96; -- LDAB
21: 00; --TEMP2
22: 36; --

23: D6; -- LDAB
24: 00; --TEMP3
25: 37; --
26: 1A; -- CMP (inh)
27: 7B; -- JB
28: 00; --SALTO3
29: 21;
30: 96; -- LDAB
31: 00; --TEMP3
32: 37; --

33: D6; -- LDAB
34: 00; --TEMP4
35: 38;
36: 1A; -- CMP (inh)
```

37: 7B; -- JB
38: 00; -- SALTO4;
39: 2B; --
40: 96; -- LDAB
41: 00; -- TEMP4
42: 38;

43: D6; -- LDAB
44: 00; --TEMP5
45: 39;
46: 1A; -- CMP (inh)
47: 7B; -- JB
48: 00; --SALTO5;
49: 35;
50: 96; -- LDAB
51: 00; --TEMP5;
52: 39;

53: 01; -- TEMP1
54: 02; -- TEMP2
55: 03; -- TEMP3
56: 04; -- TEMP4
57: 05; -- TEMP5
[58..255] : 00;
END;

El contenido de la memoria del programa se guarda en el archivo del proyecto llamado “mem_content.mif”.

Simulación

Ejercicio 1

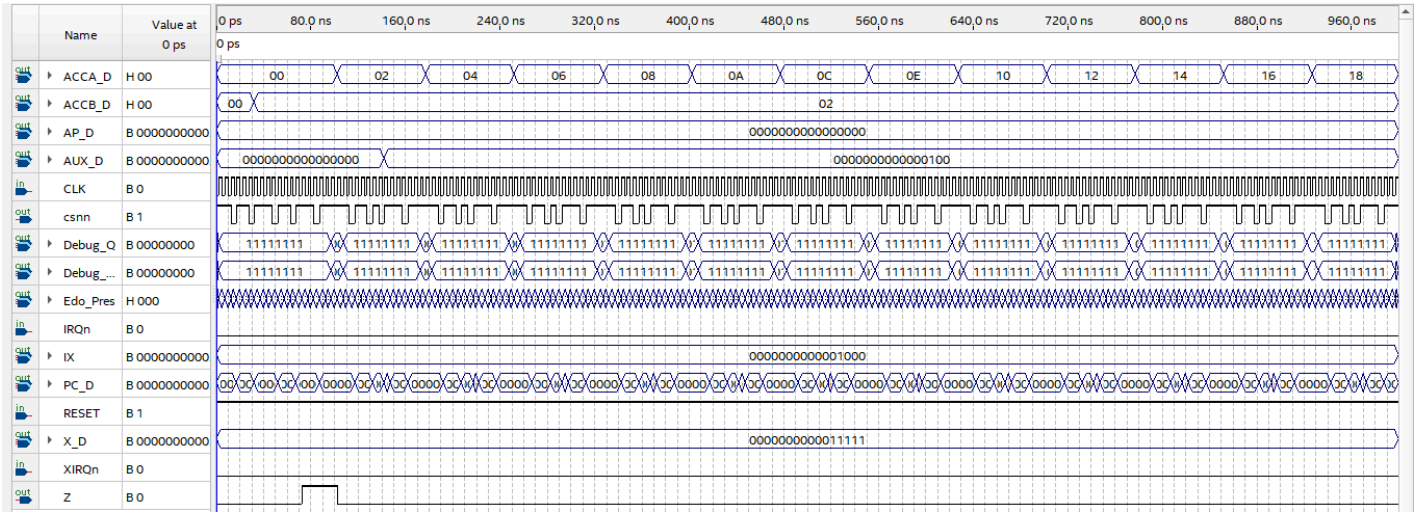

Imagen 2.1 - simulación del ejercicio 1.

ejecución de las instrucciones LDAB

Imagen 2.2 - visualización de los estados de las instrucciones implementadas.

Ejercicio 2

A y B son iguales

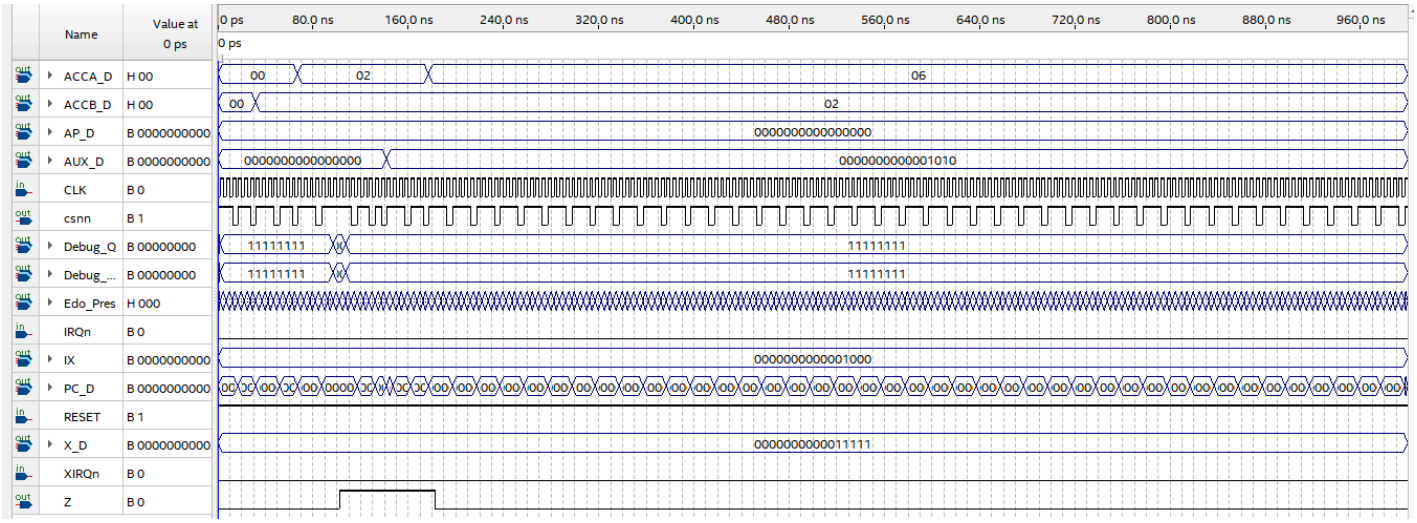

Imagen 2.3 - simulación del ejercicio 2 para números iguales.

A y B son diferentes

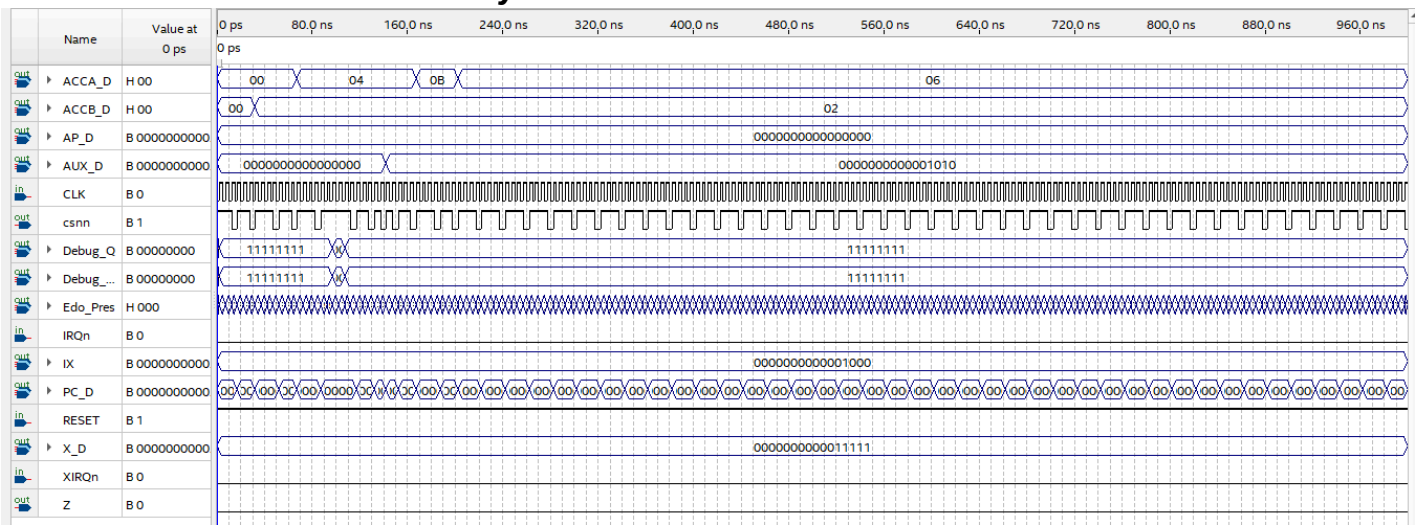

Imagen 2.4 - simulación del ejercicio 2 para números diferentes.

ejecución de las instrucciones CMP y JE

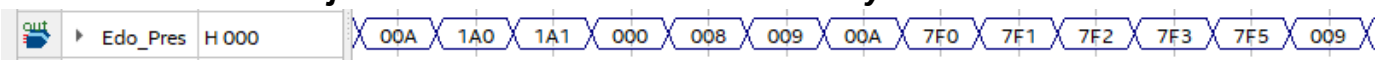

Imagen 2.5 - visualización de los estados de las instrucciones implementadas.

Ejercicio 3

A y B son iguales

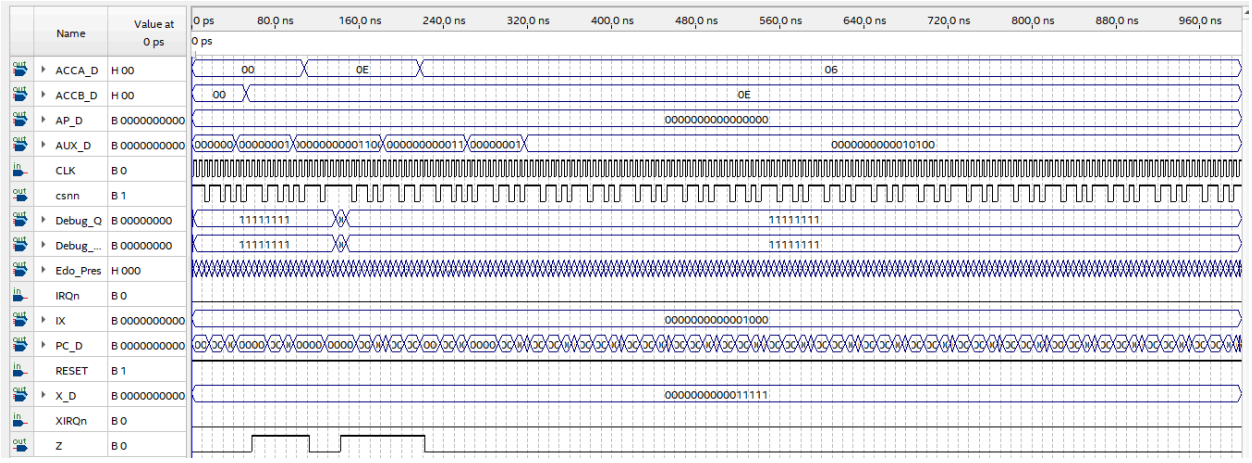

Imagen 2.6 - simulación del ejercicio 3 para números iguales.

A y B son diferentes

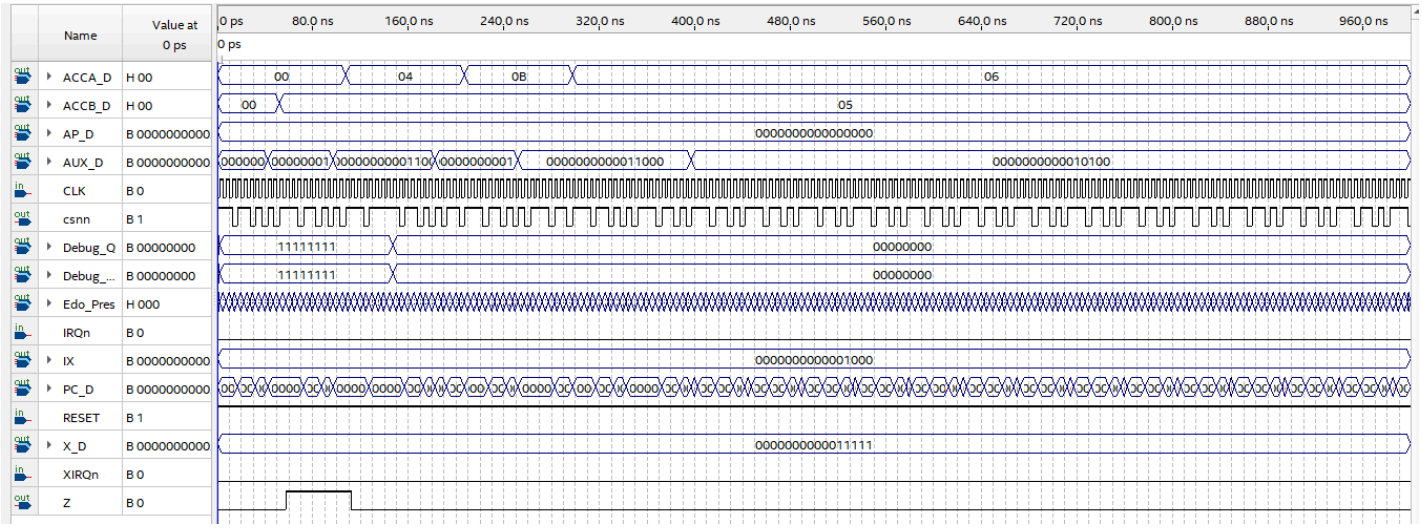

imagen 2.7 - simulación del ejercicio 3 para números diferentes.

Ejecución de las instrucciones STAA, LDAA y LDAB extendidas.

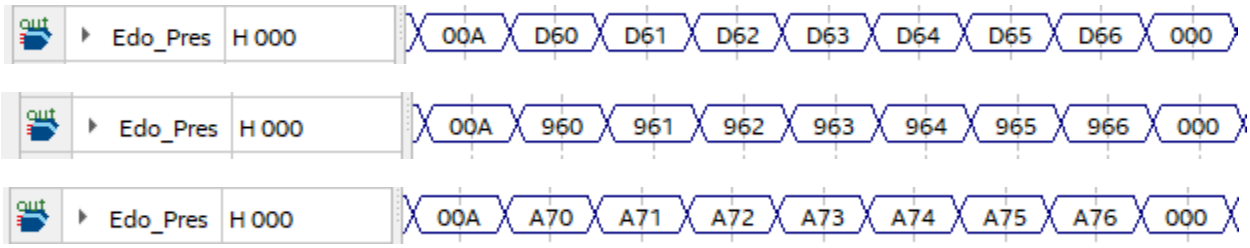

imagen 2.8 - visualización de los estados de las instrucciones implementadas.

Ejercicio 4

Lista = [1, 2, 3, 4, 5], res=1, número positivo

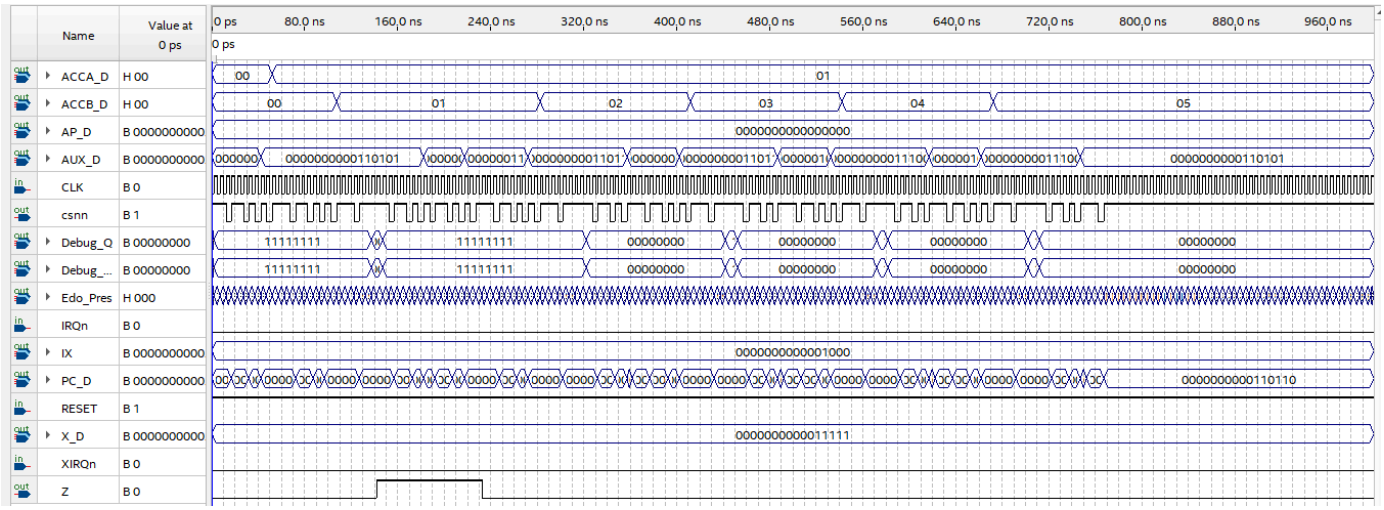

Imagen 2.9 - simulación del ejercicio 4, resultado positivo.

Lista = [5, 6, 1, 4, 2], res=1, número positivo

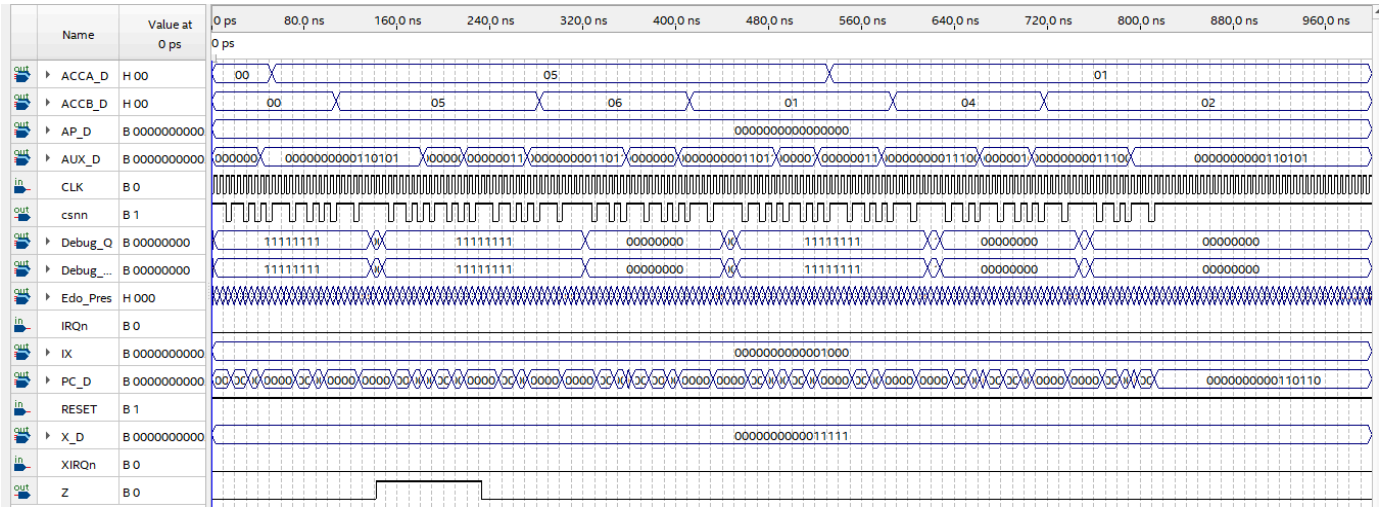

Imagen 2.10 - simulación del ejercicio 4, resultado positivo.

Lista = [5, FF, 1, 4, 2], res=FF, número negativo

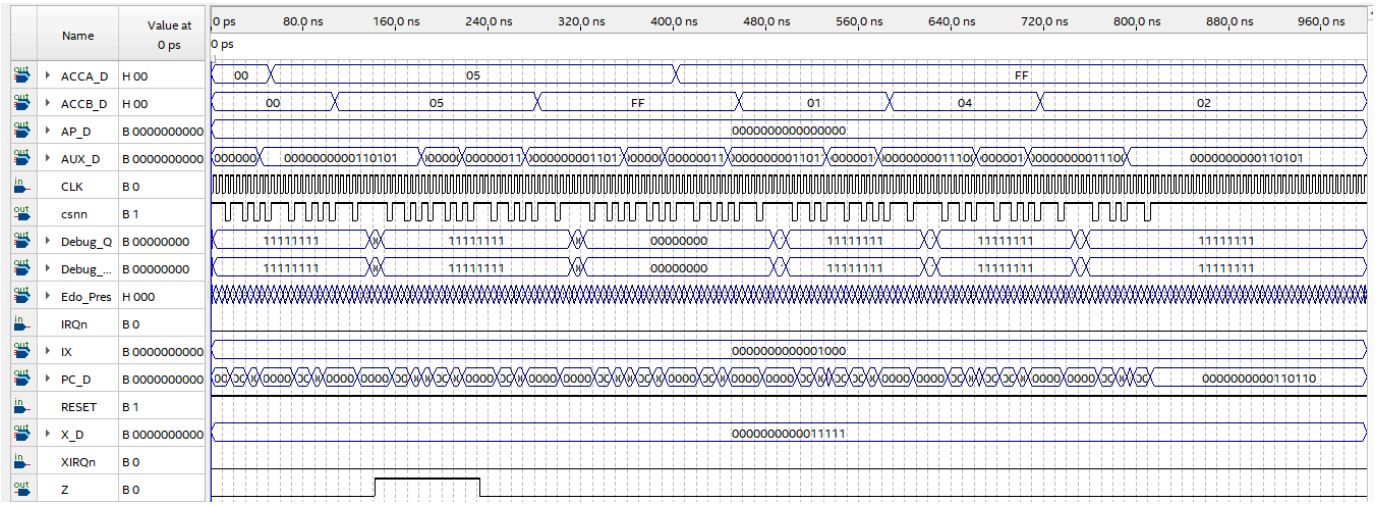

Imagen 2.11 - simulación del ejercicio 4, resultado negativo.

Ejecución de las instrucciones JB

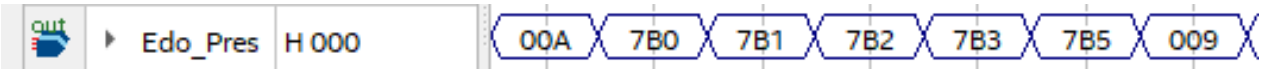

imagen 2.12 - visualización de los estados de las instrucciones implementadas.

Conclusiones

Jiménez Treviño Emilio Cristóbal

En esta práctica se pudo interactuar mediante quartus con el procesador CISC 68HC11 con el cual fuimos implementando diferentes instrucciones la cual le indican al procesador que es lo que debía hacer con los valores dados, en la memoria, se realizaron 4 ejercicios en los cuales tuvimos diferentes complicaciones desde implementar nuevas instrucciones a problemas de llenado de las instrucciones por lo tanto no hacían lo que deberían, se soluciono mediante muchas veces estarlo revisando para ver si habíamos puesto correctamente las banderas de cada instrucción de los diagramas.

Martinez Perez Brian Erik

Esta experiencia permitió comprender la organización de un procesador real, particularmente la interconexión de componentes clave como los Acumuladores A y B, los Registros Índices, y el crítico Registro de Códigos de Condición (CCR). La implementación de la arquitectura demostró cómo las instrucciones CISC se traducen en secuencias detalladas de micro-operaciones. Para ello implementamos códigos de lenguaje máquina que utilizaban las instrucciones que implementamos en la memoria rom del secuenciador, comprobando que realizan las operaciones adecuadas dentro del procesador.

Bibliografía

Laboratorio de Organización y Arquitectura de Computadoras. (2019, octubre 14). *Práctica No. 7 Procesador CISC 68HC11.*