Organización y Arquitectura de Computadoras
Proyecto Final

Martínez López Andrés
Mendoza Toledo Oscar

25 de Enero del 2021

1 Objetivo

Implementar la arquitectura RISC que permita el control de riesgos por dependencia de datos.

2 Desarrollo

2.1 Construir la arquitectura pipeline con el lenguaje VHDL.

2.1.1 Arquitectura pipeline

La principal característica de la arquitectura pipeline para el diseño de microprocesadores es que se pueden ejecutar múltiples instrucciones de forma simultánea, produciendo una mejora en el rendimiento e incrementando la productividad de las instrucciones a diferencia de las instrucciones ejecutadas de forma secuencial.
El pipeline consiste en la ejecución de estas cuatro instrucciones ejecutándose al mismo tiempo.

- Búsqueda de la instrucción.
- Decodificación.
- Búsqueda de datos o traer operandos.
- Ejecutar instrucción.

Por lo tanto la arquitectura que del microprocesador diseñado ejecutará los pasos del pipeline en las siguientes tres etapas:

1. Etapa 1: *LECTURA DE LA INSTRUCCIÓN*
   Donde se obtiene el código de operación de las instrucciones. Ya que algunas instrucciones requieren direcciones de memoria, para propiciar que el flujo de información solo sea hacia adelante y no retroceder a etapas anteriores la instrucción a ejecutar tendrá 32 bits de información donde estarán concatenados el código de operación, los registros en donde se va a operar y direcciones de memoria dependiendo el tipo de direccionamiento de cada instrucción.

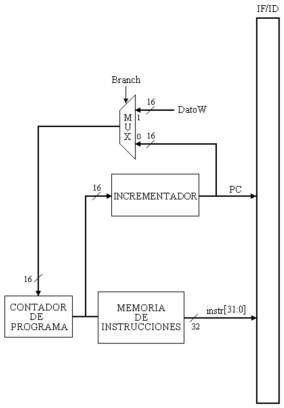

Figure 2: Hardware para la etapa de Lectura de la instrucción.

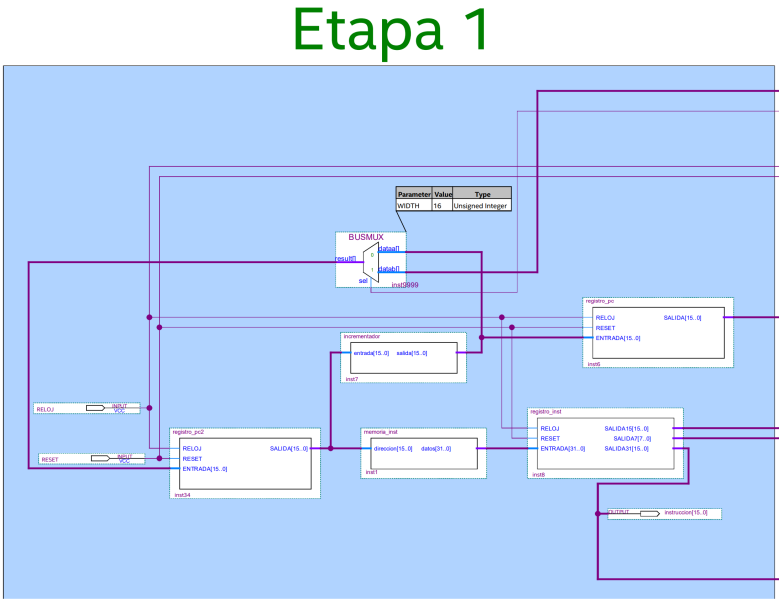

Figure 3: Diseño en VHDL para la etapa de Lectura de la instrucción.

2. Etapa 2: *DECODIFICACIÓN DE LA INSTRUCCIÓN*
   Es donde el módulo de control genera señales de control hacia los demás componentes para indicarles con que información deben de operar y el tipo de operación que realizarán.

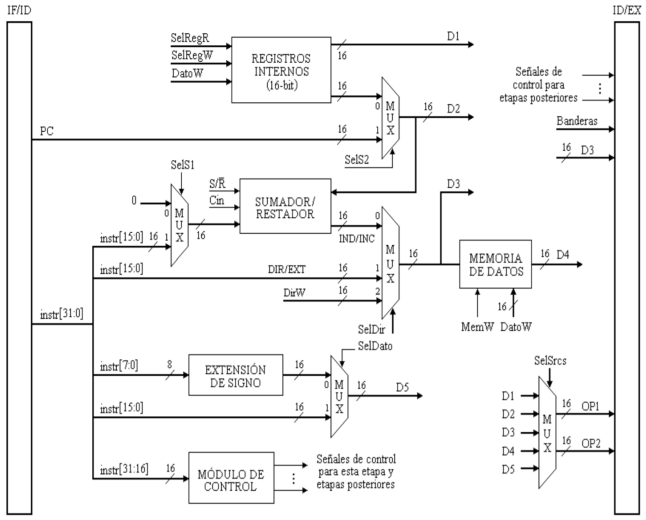

Figure 4: Hardware para la etapa de Decodificación de la instrucción y lectura de operandos.

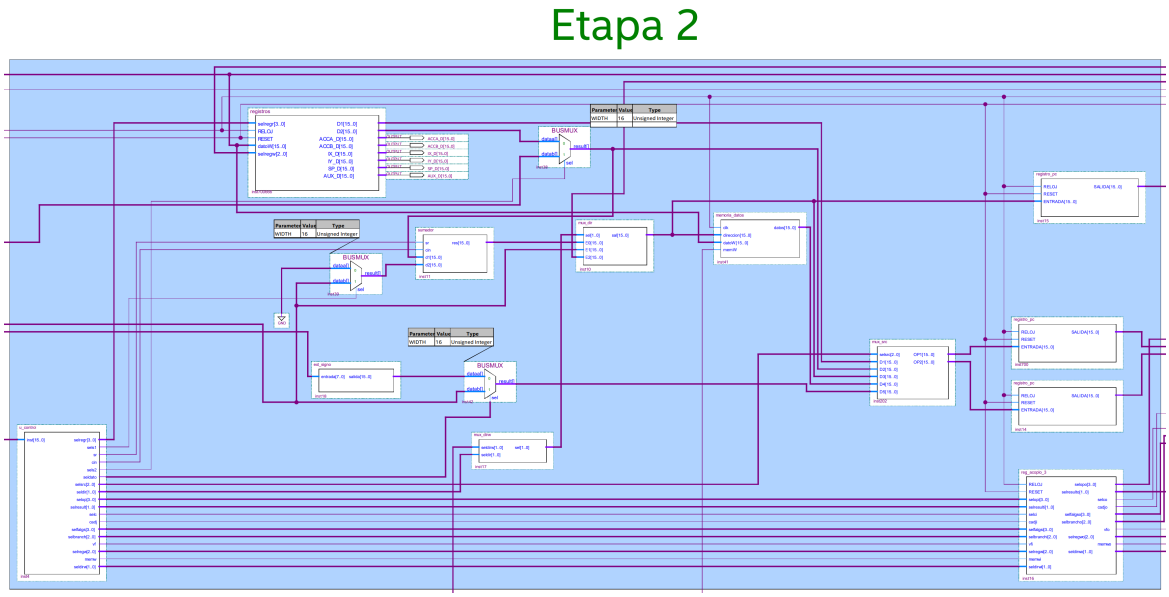

Figure 5: Diseño en VHDL para la etapa de Decodificación de la instrucción y lectura de operandos.

3. Etapa 3: *EJECUCIÓN / CÁLCULO DE BANDERAS Y SALTOS*
   Es en donde se operan los datos obtenidos de la etapa 2, donde se actualiza el registro de estados o de banderas y donde se calcula la condición de salto.

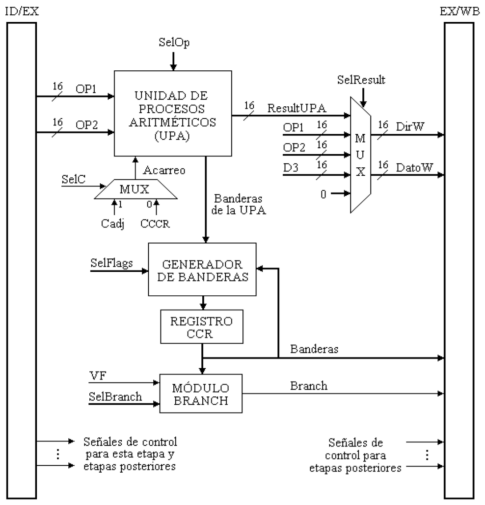

Figure 6: Hardware para la etapa de Ejecución.

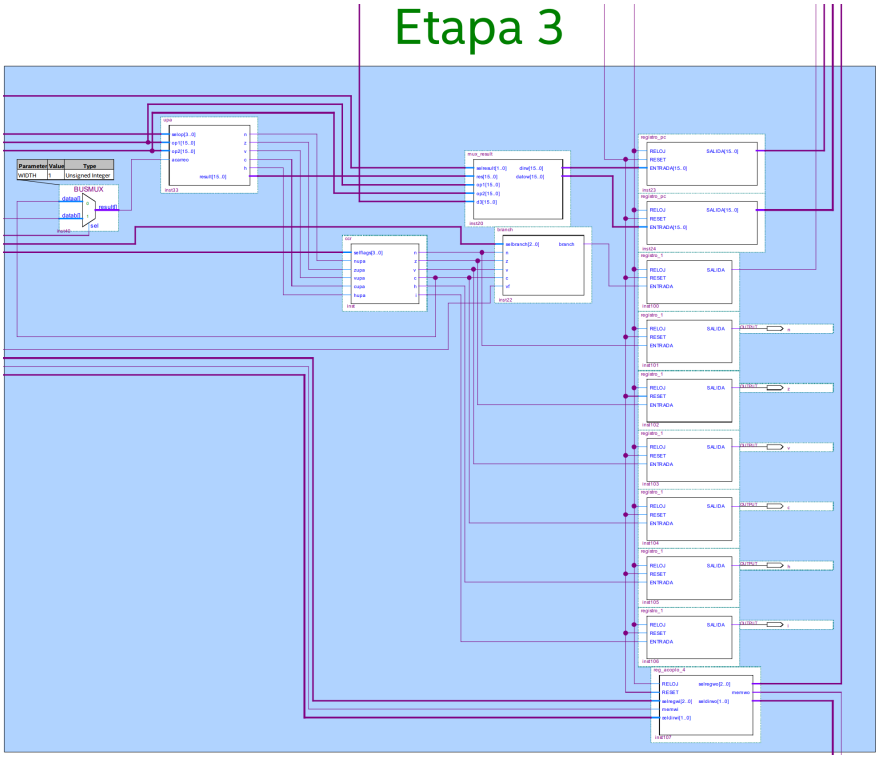

Figure 7: Diseño en VHDL para la etapa de Ejecución.

2.2 Añadir al set de instrucciones de la arquitectura las instrucciones LDAB, LDAA, ABA y JMP.

2.2.1 Instrucción LDAB

```
-- LDAB
elsif (inst = X"00A5") then
   selregr <= X"0";
   sels1 <= '0';
   sr <= '1';
   cin <= '0';
   sels2 <= '0';
   seldato <= '1';
   selsrc <= "011";
   seldir <= "00";
   selop <= X"4";
   selresult <= "01";
   selc <= '1';
   cadj <= '0';
   selfalgs <= X"1";
   selbranch <= "000";
   vf <= '1';
   selregw <= "100";
   memw <= '0';
   seldirw <= "00";
```

Figure 8: Instrucción LDAB

- Código de operación: 00A5.
- Descripción: Carga el dato en el acumulador ACCB.
- Tipo de acceso: Inmediato.
- Banderas que afecta: N, Z, V = 0.

2.2.2 Instrucción LDAA

```
-- LDAA
elsif (inst = X"0086") then
   selregr <= X"0";
   sels1 <= '0';
   sr <= '1';
   cin <= '0';
   sels2 <= '0';
   seldato <= '1';
   selsrc <= "011";
   seldir <= "00";
   selop <= X"4";
   selresult <= "01";
   selc <= '1';
   cadj <= '0';
   selfalgs <= X"1";
   selbranch <= "000";
   vf <= '1';
   selregw <= "001";
   memw <= '0';
   seldirw <= "00";
```

Figure 9: Instrucción LDAA

- Código de operación: 0086.
- Descripción: Carga el dato en el acumulador ACCA.
- Tipo de acceso: Inmediato.
- Banderas que afecta: N, Z, V = 0.

2.2.3 Instrucción ABA

```
-- ABA
if (inst = X"001B") then
   selregr <= X"1";
   sels1 <= '0';
   sr <= '1';
   cin <= '0';
   sels2 <= '0';
   seldato <= '1';
   selsrc <= "001";
   seldir <= "00";
   selop <= X"1";
   selresult <= "01";
   selc <= '1';
   cadj <= '0';
   selfalgs <= X"2";
   selbranch <= "000";
   vf <= '1';
   selregw <= "001";
   memw <= '0';
   seldirw <= "00";
```

Figure 10: Instrucción ABA

- Código de operación: 001B.
- Descripción: Realiza la suma entre los acumuladores ACCA y ACCB y el resultado se almacena en el acumulador ACCA.
- Tipo de acceso: Inherente.
- Banderas que afecta: N, Z, V, C, H.

2.2.4 Instrucción JMP

```
-- JMP
elsif (inst = X"007E") then
   selregr <= X"0";
   sels1 <= '0';
   sr <= '0';
   cin <= '0';
   sels2 <= '0';
   seldato <= '1';
   selsrc <= "011";
   seldir <= "00";
   selop <= X"4";
   selresult <= "01";
   selc <= '0';
   cadj <= '0';
   selfalgs <= X"0";
   selbranch <= "000";
   vf <= '0';
   selregw <= "000";
   memw <= '0';
   seldirw <= "00";
```

Figure 11: Instrucción JMP

- Código de operación: 007E.
- Descripción: Realiza el salto a la dirección de memoria del programa especificada.
- Tipo de acceso: Extendido.
- Banderas que afecta: No se modifica el CCR.

2.3 Probar la arquitectura con el siguiente código de ejemplo

```
LDAB #$02
LDAA #$00
ABA
JMP #$0002
```

Figure 12: Programa 1

```
8  entity memoria_inst is
9     Port( direccion : in STD_LOGIC_VECTOR (15 downto 0);
10          datos : out STD_LOGIC_VECTOR (31 downto 0));
11 end memoria_inst;
12
13 architecture Behavioral of memoria_inst is
14
15    type memory is array(0 to 50) of std_logic_vector(31 downto 0);
16    signal memoria : memory;
17       begin
18
19       -- Programa 1
20       memoria(0) <= x"00A50002"; -- LDAB  #$0002    (B <- 2)
21       memoria(1) <= x"00860000"; -- LDAA  #$0000    (A <- 0)
22       memoria(2) <= x"00010000"; -- NOP             (Nops de salto)
23       memoria(3) <= x"00010000"; -- NOP
24       memoria(4) <= x"001B0000"; -- ABA             (A <- A + B)
25       memoria(5) <= x"007E0002"; -- JMP
26       memoria(6) <= x"00010000"; -- NOP             (Nops de salto)
27       memoria(7) <= x"00010000"; -- NOP
```

Figure 13: Memoria del Programa 1

Pudimos probar con éxito el funcionamiento de nuestra arquitectura ejecutando el Programa 1. La linea remarcada en azul señala los cambios que surgen en el acumulador ACCA durante la ejecución del programa y su comportamiento es justo el esperado, para empezar se carga el dato 02 en el registro acumulador ACCB (LDAB) y un 00 en el ACCA (LDAA), después se realiza la suma entre ambos y el resultado se almacena en ACCA (ABA), por último se realiza un salto a la instrucción 0002 (JMP) y se vuelven a sumar los registros una y otra vez. Por lo tanto el registro ACCA empezará en 00 y aumentará 02 cada vez que se ejecute la instrucción ABA y este es el resultado obtenido en nuestra simulación:

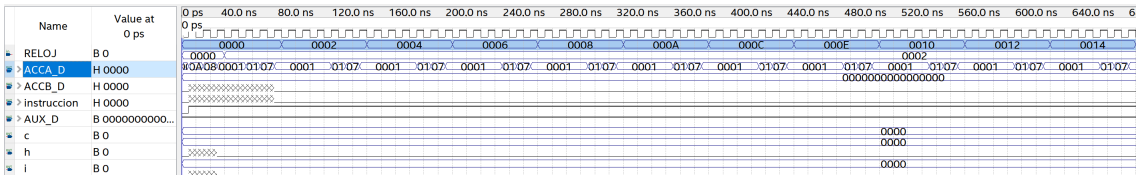

Figure 14: Simulación del Programa 1

La simulación demuestra el correcto funcionamiento, el registro ACCA va aumentando de dos en dos, mientras tanto el acumulador ACCB se mantiene en 02.

2.4 Añadir al set de instrucciones de la arquitectura las instrucciones necesarias para resolver el algoritmo de la suma de números naturales

2.4.1 Instrucción STAA

```
--STAA
elsif (inst = X"00B7") then
   selregr <= X"4";
   sels1 <= '1';
   sr <= '1';
   cin <= '0';
   sels2 <= '0';
   seldato <= '1';
   selsrc <= "001";
   seldir <= "00";
   selop <= X"4";
   selresult <= "01";
   selc <= '1';
   cadj <= '0';
   selfalgs <= X"1";
   selbranch <= "000";
   vf <= '1';
   selregw <= "000";
   memw <= '1';
   seldirw <= "10";
```

Figure 15: Instrucción STAA

- Código de operación: 00B7.
- Descripción: Carga el valor de ACCA en la memoria.
- Tipo de acceso: Extendido.
- Banderas que afecta: N, Z, V = 0.

2.4.2 Instrucción NOP

```
-- NOP
elsif (inst = X"0001") then
   selregr <= X"0";
   sels1 <= '0';
   sr <= '0';
   cin <= '0';
   sels2 <= '0';
   seldato <= '0';
   selsrc <= "000";
   seldir <= "00";
   selop <= X"0";
   selresult <= "00";
   selc <= '0';
   cadj <= '0';
   selfalgs <= X"0";
   selbranch <= "000";
   vf <= '1';
   selregw <= "000";
   memw <= '0';
   seldirw <= "00";
```

Figure 16: Instrucción NOP

- Código de operación: 0001.
- Descripción: No hace ninguna operación, únicamente consume ciclos de reloj.
- Tipo de acceso: Inherente.
- Banderas que afecta: No se modifica el CCR.

2.4.3 Instrucción INCB

```
-- INCB
elsif (inst = X"005C") then
   selregr <= X"5";
   sels1 <= '0';
   sr <= '1';
   cin <= '0';
   sels2 <= '0';
   seldato <= '1';
   selsrc <= "001";
   seldir <= "00";
   selop <= X"1";
   selresult <= "01";
   selc <= '1';
   cadj <= '1';
   selfalgs <= X"C";
   selbranch <= "000";
   vf <= '1';
   selregw <= "100";
   memw <= '0';
   seldirw <= "00";
```

Figure 17: Instrucción INCB

- Código de operación: 005C.
- Descripción: Incrementa en 1 el valor de ACCB.
- Tipo de acceso: Inherente.
- Banderas que afecta: N, Z, V.

2.4.4 Instrucción CMPB

```
-- CMPB
elsif (inst = X"00D1") then
   selregr <= X"5";
   sels1 <= '0';
   sr <= '1';
   cin <= '0';
   sels2 <= '0';
   seldato <= '1';
   selsrc <= "010";
   seldir <= "01";
   selop <= X"2";
   selresult <= "00";
   selc <= '1';
   cadj <= '1';
   selfalgs <= X"3";
   selbranch <= "000";
   vf <= '1';
   selregw <= "000";
   memw <= '0';
   seldirw <= "00";
```

Figure 18: Instrucción CMPB

- Código de operación: 00D1.
- Descripción: Compara el dato almacenado en ACCB con el dato proporcionado.
- Tipo de acceso: Directo.
- Banderas que afecta: N, Z, V, C.

2.4.5 Instrucción BLO

```
-- BLO
elsif (inst = X"0012") then
    selregr <= X"0";
    sels1 <= '0';
    sr <= '0';
    cin <= '0';
    sels2 <= '0';
    seldato <= '1';
    selsrc <= "011";
    seldir <= "00";
    selop <= X"4";
    selresult <= "01";
    selc <= '0';
    cadj <= '0';
    selfalgs <= X"0";
    selbranch <= "110";
    vf <= '1';
    selregw <= "000";
    memw <= '0';
    seldirw <= "00";
```

Figure 19: Instrucción BLO

- Código de operación: 0012.
- Descripción: Salta a la dirección especificada si la bandera N es igual a 1.
- Tipo de acceso: Relativo.
- Banderas que afecta: No se modifica el CCR

Algorithm 1 Suma de numeros naturales desde 1 hasta n

```
1: procedure ΣNN(i, suma, n)          ▷ La sumatoria de numeros naturales
2:     i ← 1
3:     suma ← 0
4:     while i ≤ n do                 ▷ Termina cuando i = n
5:         suma ← suma + i
6:         i ← i + 1
7:     end while
8: end procedure
```

En Algorithm 1, `n` (en la línea 1), corresponde a una constante.

Figure 20: Algoritmo de la suma de números naturales

2.5 Probar el nuevo set de instrucciones de la arquitectura con el algoritmo de la suma de números naturales.

```
 8  entity memoria_inst is
 9      Port( direccion : in STD_LOGIC_VECTOR (15 downto 0);
10            datos : out STD_LOGIC_VECTOR (31 downto 0));
11  end memoria_inst;
12
13  architecture Behavioral of memoria_inst is
14
15      type memory is array(0 to 50) of std_logic_vector(31 downto 0);
16      signal memoria : memory;
17          begin
18
19          --Código - Suma de numeros naturales 1 hasta n
20
21          memoria(0) <= x"00860005"; --    LDAA  #$0005    (A <- 5)
22          memoria(1) <= x"00010000"; --    NOP             (Nops de salto)
23          memoria(2) <= x"00010000"; --    NOP
24          memoria(3) <= x"00B70004"; --    STAA  #0004     (SUMA <- A=5)
25          memoria(4) <= x"00010000"; --    NOP             (Nops de salto)
26          memoria(5) <= x"00010000"; --    NOP
27          memoria(6) <= x"00860000"; --    LDAA  #$0000    (A <- 0)
28          memoria(7) <= x"00A50001"; --    LDAB  #$0001    (B <- 1)
29          memoria(8) <= x"00010000"; --    NOP             (Nops de salto)
30          memoria(9) <= x"00010000"; --    NOP
31          memoria(10) <= x"00010000"; --   NOP
32          memoria(11) <= x"00010000"; --   NOP
33          memoria(12) <= x"005C0000"; --   INCB            Incrementa a B
34          memoria(13) <= x"00010000"; --   NOP             (Nops de salto)
35          memoria(14) <= x"00010000"; --   NOP
36          memoria(15) <= x"001B0000"; --   ABA             (A <- A + B)
37          memoria(16) <= x"00010000"; --   NOP             (Nops de salto)
38          memoria(17) <= x"00D10004"; --   CMPB            (B>=5) SIGUE
39          memoria(18) <= x"0012000C"; --   BLO
40          memoria(19) <= x"00010000"; --   NOP             (Nops de salto)
41          memoria(20) <= x"00010000"; --   NOP
42          memoria(21) <= x"007E0013"; --   JMP
43          memoria(22) <= x"00010000"; --   NOP             (Nops de salto)
44          memoria(23) <= x"00010000"; --   NOP
```

Figure 21: Memoria del programa Algoritmo 1

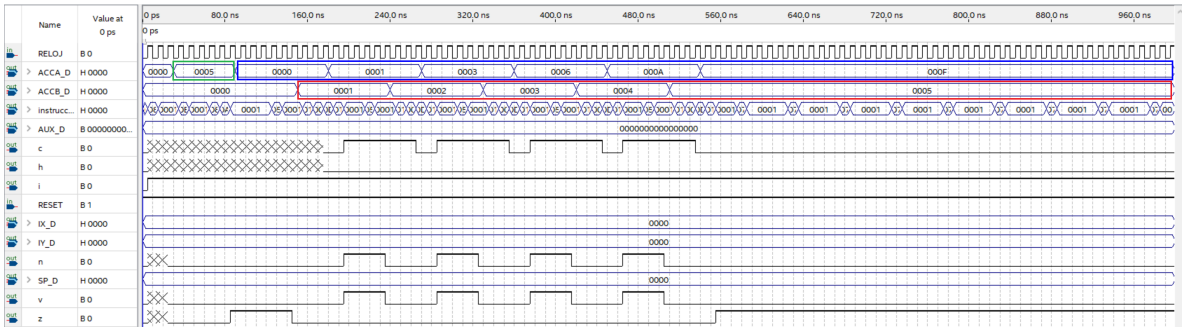

Figure 22: Simulación del programa Algoritmo 1

En la simulación podemos observar el correcto funcionamiento del algoritmo que suma los N números naturales, primero tenemos que cargar en ACCA el valor 0005 (LDAA) el cual será las N repeticiones que queremos que realice nuestro programa, lo podemos observar en la figura 22 remarcado en verde. Después almacenamos el valor de ACCA en la memoria en la dirección 0004 (STAA), después cargamos los valores iniciales en ACCA y ACCB. Aquí es donde comienza el ciclo while, donde se estarán ejecutando las siguientes instrucciones:

- Se incrementa en 1 el valor de B (INCB). $i = i + 1$. Podemos visualizar cómo se modifica este registro en la simulación marcado en rojo (Fig. 22).

- Se realiza la suma entre ACCA y ACCB, después se almacenará en ACCA (ABA). $suma = suma + 1$. Al inicio como ACCA es cero se realiza la suma $0 + 1 = 1$, sin embargo, en las siguientes pasadas se sumará el nuevo valor de ACCB con el valor que ya se encontraba en ACCA, por lo tanto se sumará $1 + 2 = 3$, luego $3 + 3 = 6$, luego $6 + 5 = 10$ y así hasta que termine el ciclo. EL cambio de este registro ACCA se encuentra marcado con azul fuerte (Fig. 22).

- Hay que verificar que i sea menor o igual que N, por lo tanto se realiza la comparación entre ACCB y el valor ubicado en la dirección 0004 la cual es en donde esta N.

- Se comprueba si se cumple la condición para terminar el ciclo, la cual es: Si ACCB es mayor a N en este caso $N = 5$, continua con la siguiente instrucción, de lo contrario saltará a la instrucción 0004 (BLO) e iniciará nuevamente el ciclo.

Como se puede observar en la figura 22 al final del recuadro rojo, una vez que $i = N$, se sale del ciclo y ya no se aumentan nuevamente los registros.

3 Reporte de actividades

Martínez López Andrés:

- *Aportación teórica:* Realice la investigación de algunas de las instrucciones que ocupa el procesador 68HC11 para visualizar que instrucciones ocuparíamos para llevar a cabo de manera exitosa nuestro proyecto.

- *Aportación práctica:* Implemente las instrucciones necesarias para ejecutar los programas solicitados en la unidad del control en lenguaje VHDL, además de modificar algunos aspectos en la arquitectura para cumplir con los requerimientos del proyecto.

Mendoza Toledo Oscar

- *Aportación teórica:* Realicé la investigación de las etapas que conforman el pipeline para la implementación de la arquitectura RISC en VHDL.

- *R*ealicé la implementación de las instrucciones del programa 1 y las simulaciones de los programas solicitados con el fin de verificar su correcto funcionamiento y expresar a detalle los procesos que se desarrollaron en la práctica.

4 Conclusiones

- *Martínez López Andrés:* Gracias al procedimiento realizado en este proyecto, logré entender de mejor manera como es que funcionaba la arquitectura RISC y la ejecución de las instrucciones en cada ciclo de reloj, además el implementar cada una de las instrucciones que ocupamos de manera manual, es decir, activando cada uno de los módulos del microprocesador, logré asimilar el funcionamiento y la importancia de cada uno de los componentes que lo conforman. En conclusión, el implementar nuestro propio microprocesador con la arquitectura RISC junto con nuestro propio set de instrucciones y entendiendo conceptualmente todo lo que involucra este tipo de arquitectura, nos permitió manejar el control de dependencia a través del software, ejecutando de esta manera el programa propuesto de manera óptima.

- *Mendoza Toledo Oscar:* Cumplimos con el objetivo del proyecto, implementamos la arquitectura RISC de un procesador en Quartus, logramos diseñar una arquitectura que siguiera las etapas del pipeline para ejecutar instrucciones de forma paralela. A diferencia de la arquitectura CISC, fue más sencillo desarrollar las instrucciones, debido a que se simplifica mucho la unidad de control y no se trabaja con tantas señales, además de que en una sola instrucción viene indicado el código de operación y los datos con los que se va a trabajar. Otra ventaja de trabajar con esta arquitectura es que la memoria del programa y de los datos son independientes, facilitando el acceso de información a la memoria de datos. Nuestras simulaciones demuestran el correcto funcionamiento de nuestra arquitectura y realizan las operaciones indicadas en los programas asignados.