UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

ORGANIZACIÓN Y ARQUITECTURA DE COMPUTADORAS

CLAVE 1867

Profesor: M. I. José Luis Cruz Mora

Integrantes del equipo:

García Fernández Jesús Alejandro

López Santibañez Jimenez Luis Gerardo

Moreno Guerra Marco Antonio

Proyecto Final

Fecha de entrega: 27 de noviembre del 2019

Grupo: 1

Semestre: 2020-1

REPORTE DEL PROYECTO FINAL

OBJETIVO

Hacer uso de la Arquitectura RISC vista en clase para realizar el desarrollo e implementación, mostrando el resultado de una serie matemática.

INTRODUCCIÓN

Una serie matemática es la suma de los términos de una sucesión:

Se representa una serie con términos como:

$$\sum_{k=1}^{n} a_k$$

donde n es el índice final de la serie.

La serie matemática a resolver es la siguiente:

$$\sum_{i=1}^{4} 2(i) + i$$

donde el desarrollo de ésta se conforma de:

$$\sum_{i=1}^{4} i + 2(i) \; = (1+2(1))+(2+2(2))+(3+2(3))+(4+2(4))$$

Para lograr realizar este desarrollo, fue conveniente implementar el control de riesgos por dependencia de datos, así como el control de riesgos por anticipación de datos, ya que como más adelante se mencionara, existen distintos momentos en donde se requerirá escribir y leer de un mismo lugar y de dependencia de datos en algunas instrucciones, lo cual requería implementar por Software una gran cantidad de instrucciones de retraso (NOP).

Control de riesgos por dependencias de datos por medio de detenciones

Lo más sencillo para resolver los riesgos por dependencias de datos a través de Hardware es detener las instrucciones en el pipeline hasta que se resuelva el riesgo de querer leer y escribir en Memoria de Datos al mismo pulso de reloj.

Un riesgo por dependencia de datos se presenta cuando una instrucción trata de leer en su etapa 2 el mismo registro que una instrucción anterior intenta escribir en su etapa 4.

Este tipo de detenciones se conocen con el sobrenombre de **burbujas**, con esta estrategia, primero se detecta un riesgo por dependencia de datos y después se detienen las instrucciones en el pipeline (insertándose burbujas) hasta que se resuelve el riesgo.

Esta resolución del riesgo significa que la arquitectura detecta cuando una instrucción en la etapa 4 va a guardar su resultado en la Memoria de Datos y una instrucción en la etapa 2 va a leer un registro de la misma memoria, por lo que detiene en la primera etapa a la instrucción más reciente y le da prioridad a la lectura de resultados de la instrucción en la etapa 4. Es decir, generar un retraso de 1 pulso de reloj (burbuja) para dar oportunidad de guardar los resultados en la memoria.

Control de riesgos por dependencias de datos por medio de anticipaciones

En ocasiones se puede presentar el caso en que alguna instrucciones necesite el valor escrito por la instrucción antecesora; sin embargo, puede que el resultado o el dato necesario aún no esté listo cuando se le requiere.

Es por esto, que la unidad de anticipaciones detecta estos casos y retroalimenta el dato del resultado de la operación en la etapa 3 a sí misma para tenerlo disponible de operar con él en el siguiente pulso de reloj para la instrucción siguiente y evitar este tipo de conflictos.

DESARROLLO

Como primer lugar decidimos realizar la implementación de las dos unidades de control de riesgo.

Para la **unidad de detenciones**, que se encuentra en la etapa 2, se inició por realizar el multiplexor, que nos permitirá elegir las señales respectivas a la siguiente instrucción o generar las señales correspondientes a una instrucción NOP (burbuja):

```
architecture Behavioral of mux_detencion is
begin
```

```
    process
(selregri,sels1i,sri,cini,sels2i,seldatoi,selsrci,seldiri,selopi,selresulti,se
lci,cadji,selfalgsi,selbranchi,vfi,selregwi,memwi,seldirwi,selctrl)
    begin
        if(selctrl = '1') then
            selregro <= X"0";
            sels1o <= '0';
            sro <= '0';
            cino <= '0';
            sels2o <= '0';
            seldatoo <= '0';
            selsrco <= "000";
            seldiro <= "00";
            selopo <= X"0";
            selresulto <= "00";
            selco <= '0';
            cadjo <= '0';
            selfalgso <= X"0";
            selbrancho <= "000";
            vfo <= '1';
            selregwo <= "000";
            memwo <= '0';
            seldirwo <= "00";
        else
            selregro <= selregri;
            sels1o <= sels1i;
            sro <= sri;
            cino <= cini;
            sels2o <= sels2i;
            seldatoo <= seldatoi;
            selsrco <= selsrci;
            seldiro <= seldiri;
            selopo <= selopi;
            selresulto <= selresulti;
            selco <= selci;
            cadjo <= cadji;
            selfalgso <= selfalgsi;
            selbrancho <= selbranchi;
            vfo <= vfi;
            selregwo <= selregwi;
```

```
            memwo <= memwi;
            seldirwo <= seldirwi;
        end if;
    end process;
end Behavioral;
```

Pero lo más importante es como tal la unidad que controla este multiplexor a través de una señal de control, que le indicará al multiplexor si debe enviar señales de instrucciones NOP al detectar un riesgo de dependencia de datos o dejar pasar las señales de la instrucción del usuario:

```
architecture Behavioral of u_detencion is

    begin
        process (SelSrc,MemWE4) begin
            if SelSrc = "010" and MemWE4 = '1' then
                SelD <= '1';
                PCWrite <= '0';
                IFIDWrite <= '0';
                SelCtrl <= '1';

            elsif SelSrc = "100" and MemWE4 = '1' then
                SelD <= '1';
                PCWrite <= '0';
                IFIDWrite <= '0';
                SelCtrl <= '1';

            elsif SelSrc = "110" and MemWE4 = '1' then
                SelD <= '1';
                PCWrite <= '0';
                IFIDWrite <= '0';
                SelCtrl <= '1';

            else
                SelD <= '0';
                PCWrite <= '1';
                IFIDWrite <= '1';
                SelCtrl <= '0';
            end if;
```

El proceso anterior sensa el valor de SelSrc y de MemW ya que esto implica que escribiremos y leeremos de memoria, lo que nos provocará conflicto, por lo cual, se mandan las señales que seleccionan en el multiplexor una señal de burbuja y con las otras señales elegirán en multiplexores de registros que mantengan su valor anterior, similar a un comportamiento de detener el reloj de la etapa.

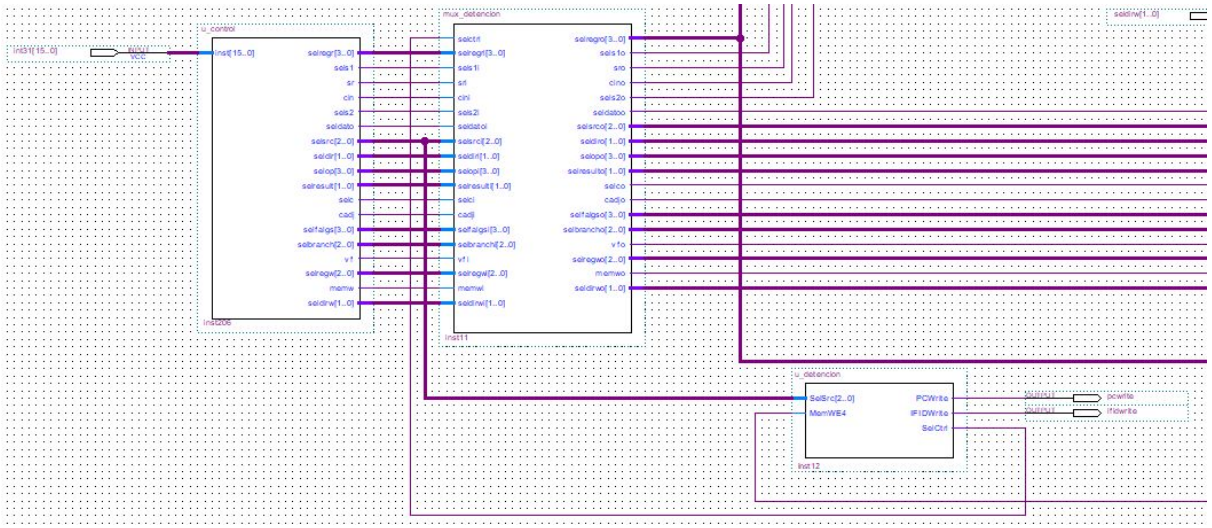

Figura 1. Diagrama de bloques de la Unidad de Control, Unidad de Detención y su Multiplexor como buffer de las señales de la siguiente instrucción a ejecutar.

Para implementar la **unidad de control por anticipaciones**, que se encuentra en la etapa 3, se realizó un módulo que sensa las señales SelRegR y SelRegW con las cuales, nos damos cuenta si una lectura de algún registro requiere el valor que se escribirá en una etapa que actualmente se está ejecutando y que quizá no esté lista para operar, es decir, que no tenga el dato actualizado; por lo tanto, dependiendo de la situación se generará una señal que controla multiplexores que decidirán si se pasara un dato que venga de una etapa anterior o si se operará el resultado que en ese momento se generó en la UPA:

```
architecture Behavioral of u_anticipacion is
    begin
        process (SelRegR,SelRegWE4)
        begin
            if SelRegR = X"1" and SelRegWE4 = "001" then
                SelA <= '1';
                SelB <= '0';

            elsif SelRegR = X"1" and SelRegWE4 = "100" then
                SelA <= '0';
                SelB <= '1';
```

```
elsif SelRegR = X"2" and SelRegWE4 = "100" then
    SelA <= '1';
    SelB <= '0';

elsif SelRegR = X"2" and SelRegWE4 = "010" then
    SelA <= '0';
    SelB <= '1';

elsif SelRegR = X"3" and SelRegWE4 = "100" then
    SelA <= '1';
    SelB <= '0';
```

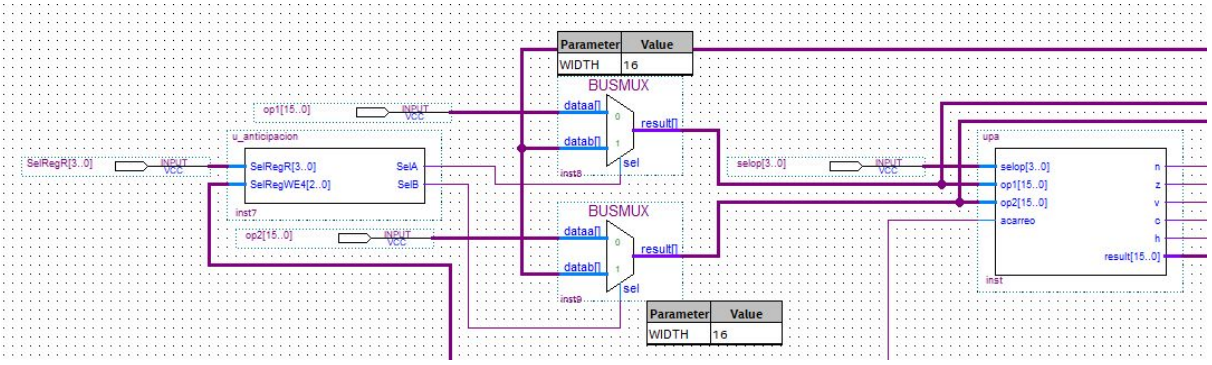

Figura 2. Diagrama de bloques de la Unidad de Anticipación y Multiplexores del dato que se elegirán como OP1y OP2 para procesar en la UPA.

Una vez que se implementaron estas unidades procedimos a implementar el conjunto de instrucciones que nos permitirán resolver la serie planteada en la introducción de este reporte.

Para ello se planteó la modificación de la **Memoria de Instrucciones** y la **Unidad de Control**. La primera se encargará de almacenar el código en ensamblador tipo RISC del usuario con todas las instrucciones que usará. Mientras que la Unidad de Control se modificó para agregar todas las señales que debe emitir la arquitectura cuando encuentre una instrucción que esté usando el usuario.

En la unidad de control se describieron cada una de las instrucciones necesarias para cumplir con nuestro objetivo, resultando similares al siguiente segmento de código ejemplo:

```
inst = X"0058" then -- lslb (inh)
            selregr <= X"5";
            sels1 <= '0';
            sr <= '1';
            cin <= '0';
            sels2 <= '0';
            seldato <= '1';
            selsrc <= "010";
            seldir <= "00";
            selop <= X"6";
            selresult <= "01";
            selc <= '1';
            cadj <= '0';
            selfalgs <= X"3";
            selbranch <= "000";
            vf <= '1';
            selregw <= "100";
            memw <= '0';
            seldirw <= "00";
```

Se usaron y describieron todas las señales de las siguientes instrucciones en la Unidad de Control:

```
architecture Behavioral of u_control is
begin
    process (inst)
    begin
        if inst = X"001B" then  -- aba (inh)
        elsif inst = X"0086" then -- ldaa #dato_16bits (imm)
        elsif inst = X"00C6" then -- ldab #dato_16bits (imm)
        elsif inst = X"0096" then -- ldaa #dir_8bits (dir)
        elsif inst = X"00D6" then -- ldab #dir_8bits (dir)
        elsif inst = X"00B7" then -- staa #dir_16bits (ext)
        elsif inst = X"00D7" then -- stab #dir_16bits (ext)
        elsif inst = X"004C" then -- inca (inh)
        elsif inst = X"005C" then -- incb (inh)
        elsif inst = X"0011" then -- cba (inh)
        elsif inst = X"007E" then -- jmp #dir_16bits (ext)
        elsif inst = X"0001" then -- nop (inh)
        elsif inst = X"0027" then -- jz #dir_16bits (ext)
        elsif inst = X"0070" then -- neg #dir_16bits (ext)
        elsif inst = X"004F" then -- clra (inh)
        elsif inst = X"005F" then -- clrb (inh)
        elsif inst = X"0058" then -- lslb (inh)
```

Figura 3. Vista general de todas las instrucciones que se implementaron en la Unidad de Control.

Una vez descritas e implementadas las instrucciones a utilizar, seguimos con el orden en la ejecución de las instrucciones dentro de nuestra memoria de instrucciones, que le darán lógica al desarrollo de nuestra serie objetivo:

```
-- Inicio:
memoria(0) <= x"00860000"; -- LDAA #0000 ; (ACCA) <- 0x0000
memoria(1) <= x"00010000"; -- NOP
memoria(2) <= x"00B70001"; -- STAA $#0001 ; (mem[1])<-(ACCA) =>init acumulator
memoria(3) <= x"00010000"; -- NOP
memoria(4) <= x"00010000"; -- NOP
memoria(5) <= x"004C0000"; -- INCA ; (ACCA) <- (ACCA) + 1
memoria(6) <= x"00010000"; -- NOP
memoria(7) <= x"00B70000"; -- STAA $#0000 ; (mem[0]) <- (ACCA) => init counter
memoria(8) <= x"00010000"; -- NOP
memoria(9) <= x"00D60000"; -- LDAB $#0000 ; (ACCB) <- (mem[0]) => counter

-- Límite ACCB = 5 (no se llega a este valor)
memoria(10)<= x"00860005"; -- LDAA #0005 ; (ACCA) <- 0x0005

-- IF (ACCB >= límite) THEN: reset
memoria(11)<= x"00110000"; -- CBA ; (ACCA) - (ACCB) ;update flags (N, Z, V, C)
memoria(12)<= x"0027001D"; -- JZ #0013 ; IF (Z == 1): PC <- (0x001D = 0d0029)
memoria(13)<= x"00010000"; -- NOP
memoria(14)<= x"00010000"; -- NOP
memoria(15)<= x"00010000"; -- NOP

-- ELSE: (ACCA + ACCB; ACCB <- ACBB + 1)
memoria(16)<= x"00960000"; -- LDAA $#0000 ; (ACCA) <- (mem[0]) => counter
memoria(17)<= x"00580000"; -- LSLB ; ACCB <<  => 2^ACCB
memoria(18)<= x"001B0000"; -- ABA ; (ACCA) <- (ACCA) + (ACCB)
memoria(19)<= x"00D60001"; -- LDAB $#0001 ; (ACCB) <- (mem[1])=>old acumulator
memoria(20)<= x"001B0000"; -- ABA ; (ACCA)<- (ACCA) + (ACCB) => new acumulator
memoria(21)<= x"00B70001"; -- STAA $#0001 ; (mem[1]) <- (ACCA) => acumulator
memoria(22)<= x"00960000"; -- LDAA $#0000 ; (ACCA) <- (mem[0]) => counter
memoria(23)<= x"004C0000"; -- INCA ; (ACCA) <- (ACCA) + 1
memoria(24)<= x"00B70000"; -- STAA $#0000 ; (mem[0]) <- (ACCA) => acumulator+1
memoria(25)<= x"007E0009"; -- JMP #0009 ; PC <- (0x0009)
memoria(26)<= x"00010000"; -- NOP
memoria(27)<= x"00010000"; -- NOP
memoria(28)<= x"00010000"; -- NOP
```

```
-- RESET (FINAL)
memoria(29)<= x"004F0000"; -- CLRA ; ACCA <- 0x0000
memoria(30)<= x"005F0000"; -- CLRB ; ACCB <- 0x0000
memoria(31)<= x"007E0000"; -- JMP #0000 ; PC <- (0x0000)
memoria(32)<= x"00010000"; -- NOP
memoria(33)<= x"00010000"; -- NOP
memoria(34)<= x"00010000"; -- NOP
```

Como se observa, utilizaremos los registros ACCA y ACCB, así como localidades en Memoria de Datos que servirán para guardar valores como el valor actual del **contador** y el **acumulador** o **suma**.

A grandes rasgos las instrucciones que se utilizaron realizan las siguientes tareas:

- ➔ **LDAA:** carga un valor en el registro ACCA.
- ➔ **STAA:** carga el valor de del registro ACCA en una localidad de memoria.
- ➔ **INCA:** incrementa el registro ACCA en una unidad.
- ➔ **LDAB:** carga el valor de una localidad de memoria en el registro ACCB.
- ➔ **CBA:** compara los valores del registro ACCA y ACCB, realizando una resta y actualizando la bandera N.
- ➔ **JZ:** realiza un salto si se cumple la condición Z=1.
- ➔ **LSLB:** realiza un corrimiento a la izquierda del registro ACCB.
- ➔ **ABA:** realiza una suma del registro ACCA y el registro ACCB.
- ➔ **JMP:** realiza un salto incondicional.
- ➔ **CLRA:** hace un reset del registro ACCA.
- ➔ **CLRB:** hace un reset del registro ACCB.
- ➔ **NOP:** envía todas las señales en 0 a excepción de VF para evitar un salto y generar un retraso de 1 ciclo de reloj.

CONCLUSIONES

Al realizar este proyecto se puso en práctica el conocimiento adquirido durante el curso, el cual brindó la habilidad de entender la estructura de las arquitecturas CISC y RISC. Para el desarrollo de nuestro proyecto se determinó usar la arquitectura RISC o Pipeline debido a la simplicidad que representa en el código ensamblador de dicho procesador.

Para llevar a cabo el programa planteado fue necesario hacer un análisis de las instrucciones que se deseaban usar para establecer sus señales de manera correcta en la Unidad de Control de la Etapa 2.

Como punto adicional, se decidió implementar las Unidades de Detención y Anticipación/Adelantos debido a que se buscó un enfoque de sencillez de programación para el usuario y aprovechar al máximo la filosofía de Pipeline de ejecutar una instrucción cada ciclo de reloj y que el usuario implemente su programa bajo este enfoque, dejando en manos del Hardware la detección y análisis de cuándo debe incluir NOPs o retrasos. Ya que de no ser así el programa debería implementar muchos NOPs (a través de Software) y complicaría el trabajo al usuario.

Sin embargo, los NOPs continúan siendo requeridos cuando se hace cualquier tipo de salto, sea incondicional o condicional.