Facultad de Ingeniería

Organización y Arquitectura de Computadoras

Proyecto de Organización y Arquitectura de Computadoras

Alumnos
Brito Sánchez Diego
Castelan Hernandez Mario
Miranda Hernández Alejandro
Romero Andrade Cristian
Solano Morales Isaac Uriel

Grupo: 02

Profesor
Ing. Hugo Enrique Estrada León

Semestre
2022–1

Fecha de Entrega
17 de diciembre de 2021

Proyecto - Organización y Arquitectura de Computadoras

Brito Sánchez Diego, Castelan Hernandez Mario, Miranda Hernández Alejandro, Romero Andrade Cristian y Solano Morales Isaac Uriel

Índice

I. Objetivo

El alumno programará las instrucciones necesarias para poder ejecutar un algoritmo sobre una arquitectura de computadora diseñada por el alumno.

I-A. Algoritmos

Se debe elegir alguno de los algoritmos propuestos e implementar la o las instrucciones necesarias para llevarlo a cabo. Los modos de direccionamiento y la arquitectura son libres de elección. Si se elige la arquitectura RISC, se tendrá un pontaje extra en la calificación. En dado caso que para su algoritmo existan riesgos por dependencia de datos, estos se solucionarían vía software (agregando instrucciones NOP) y no por hardware.

I-B. Área de un octágono

Se debe implementar el algoritmo que permita obtener el área de un octágono. Se usará la siguiente formula: $A = \frac{perimetro \times apotema}{2}$ siendo el *perimetro* y el *apotema* números enteros.

II. Introducción

II-A. Arquitectura y organización de la computadora

La arquitectura de la computadora hace referencia al conjunto de elementos del computador que son visibles desde el punto de vista del programador de ensamblador. La organización de la computadora se refiere a las unidades funcionales del computador y al modo como están interconectadas.

II-B. Arquitectura Von Neumann

El objetivo de la arquitectura Von Neumann es construir un sistema flexible que permita resolver diferentes tipos de problemas. Para conseguir esta flexibilidad, se construye un sistema de propósito general que se pueda programar para resolver los diferentes tipos de problemas. Para cada problema concreto se define un programa diferente. La arquitectura Von Neumann se basa en tres propiedades:

1. Hay un único espacio de memoria de lectura y escritura, que contiene las instrucciones y los datos necesarios.
2. El contenido de la memoria es accesible por posición, independientemente de que se acceda a datos o a instrucciones.
3. La ejecución de las instrucciones se produce de manera secuencial: después de ejecutar una instrucción se ejecuta la instrucción siguiente que hay en la memoria principal, pero se puede romper la secuencia de ejecución utilizando instrucciones de ruptura de secuencia.

II-C. Arquitectura Harvard

La organización del computador según el modelo Harvard, básicamente, se distingue del modelo Von Neumann por la división de la memoria en una memoria de instrucciones y una memoria de datos, de manera que el procesador puede acceder separada y simultáneamente a las dos memorias.

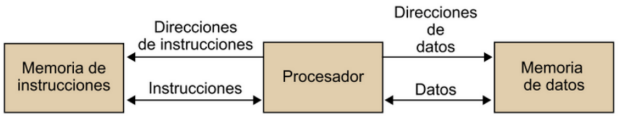


Figura 1: Arquitectura Harvard

II-D. RISC: Reduced instruction set computer processor

Es una arquitectura de procesadores basada en una colección de instrucciones simples y altamente personalizadas. RISC se construye para minimizar el tiempo de ejecución de una instrucción, optimizando y limitando el número de instrucciones. La arquitectura RISC tiene la capacidad de por cada ciclo de instrucción se da solo un ciclo de reloj. Cada ciclo debe contener cuatro etapas: buscar, decodificar, ejecutar y guardar.

II-E. CISC: complex instruction set computer

CISC es un sistema de instrucciones desarrollado por Intel que requieren de mucho tiempo para ser ejecutadas completamente. Lo que sucede en CISC es que se reduce la cantidad de instrucciones de un software y se ignora el número de ciclos por instrucción. Se especializa en crear instrucciones complejas en el hardware, ya que el hardware siempre será mucho más rápido que el software.

III. Desarrollo

Primeramente se diseñan las etapas de la arquitectura RISC, las cuales se dividen en cuatro:

1. Llamada a la instrucción
2. Decodificación de la Instrucción
3. Llamada a los operadores
4. Ejecución

Para la arquitectura 68HC11 cada instrucción ejecuta los siguientes pasos:

1. Obtener instrucción ejecutable de la memoria (bucle de recuperación)
2. Instrucciones de decodificación
3. Si la instrucción solicita leer un operando de la memoria, entonces se calcula la dirección efectiva de ese operando y los datos se leen de la memoria.
4. Si lo requiere la instrucción, los operandos requeridos se leen de los registros internos del microprocesador.
5. Ejecución, es decir, la operación se realiza en un bloque de procesamiento aritmético con operandos leídos previamente
6. Los resultados de la operación se guardan y el registro de banderas se actualiza

Se ve que los pasos son similares a los ejecutados en las cartas ASM para las instrucciones. La arquitectura segmentada 68HC11 también realiza los mismos pasos, pero se agrupará en los siguientes cuatro pasos

1. Etapa IF (instruction fetch). La instrucción a ejecutar es leída de la memoria de instrucciones
2. Etapa ID (instruction decode). Se decodifica la instrucción y se traen los operandos necesarios por la instrucción (tanto de memoria como de registros internos)
3. Etapa EX (execution). Se procesan los operandos en la UPA (unidad de procesos aritméticos)
4. Etapa WB (write back). Se guardan resultados

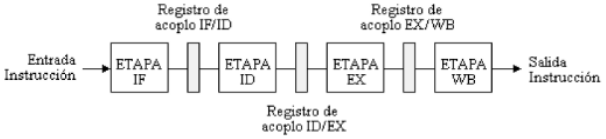


Figura 2: Etapas para la arquitectura segmentada del 68HC11

III-A. Etapa 1

En esta etapa tenemos contadores, incrementadores, multiplexores y memoria de instrucciones conectados entre sí [1, p, 133], arrojando su salida al registro de la PC y las instrucciones que serán tomadas posteriormente por la etapa 2.

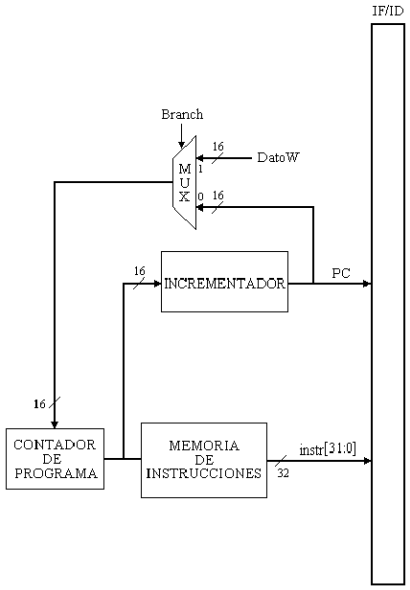


Figura 3: Etapa 1 [1, p, 133]

III-B. Etapa 2

Luego tenemos la etapa 2 con los bloques que se muestran en la introducción, destacando los registros internos básicos, módulos de control, sumadores y registros de acoplamiento para poder ejecutar el pipeline, teniendo sus respectivas salidas para poder hacerlo[1].

[1]Se recomienda apreciar la arquitectura en Quartus descargando el proyecto como se explica en la sección instalación.

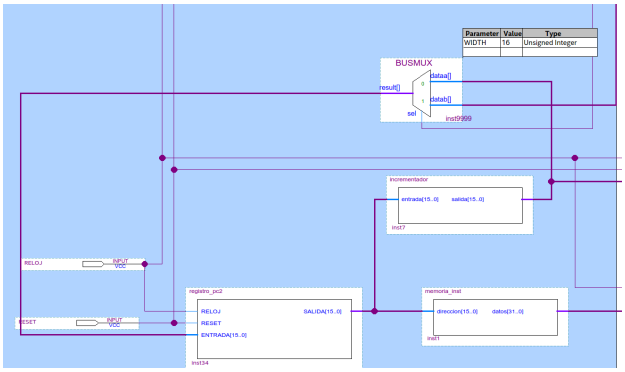


Figura 4: Etapa 1

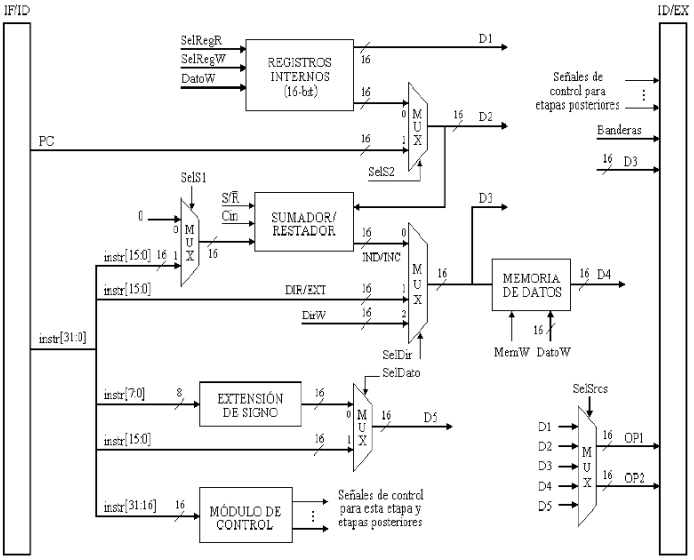


Figura 5: Etapa 2 [1, p, 135]

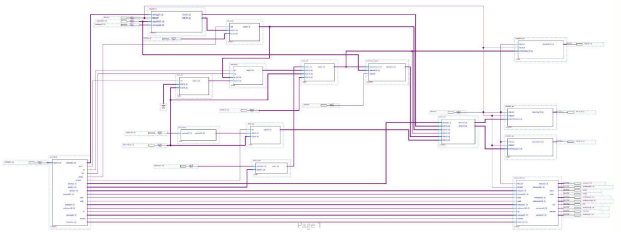


Figura 6: Etapa 2

III-C. Etapa 3

Finalmente, la etapa 03[2], lo cual se hace directamente para poder simularlo de una buena manera en vhdl. Como podemos ver en esta etapa 03 tenemos la UPA y el generador de banderas, los cuales se muestran de forma independiente, cada uno con su bloque para poder controlarlo de mejor manera y poder mostrarlo en la simulación

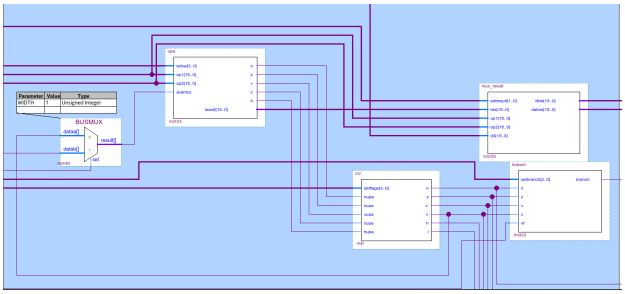


Figura 8: Etapa 3

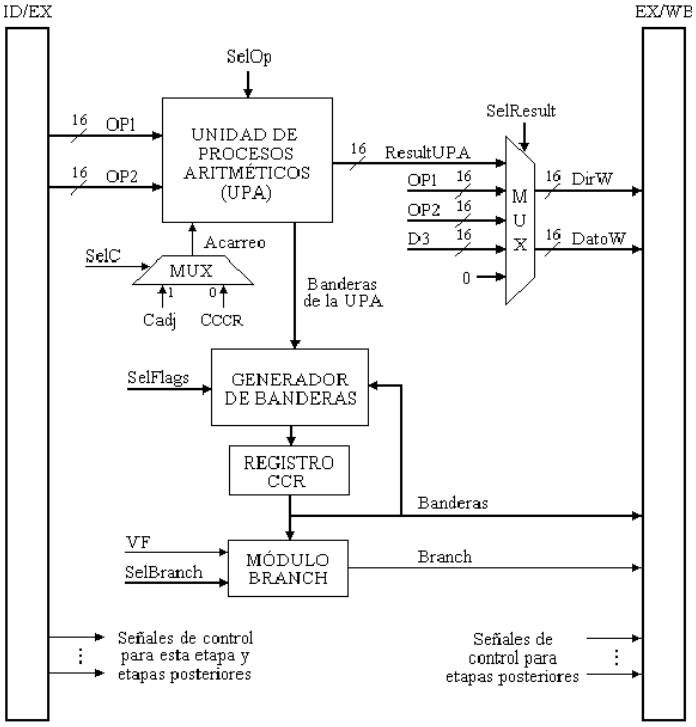
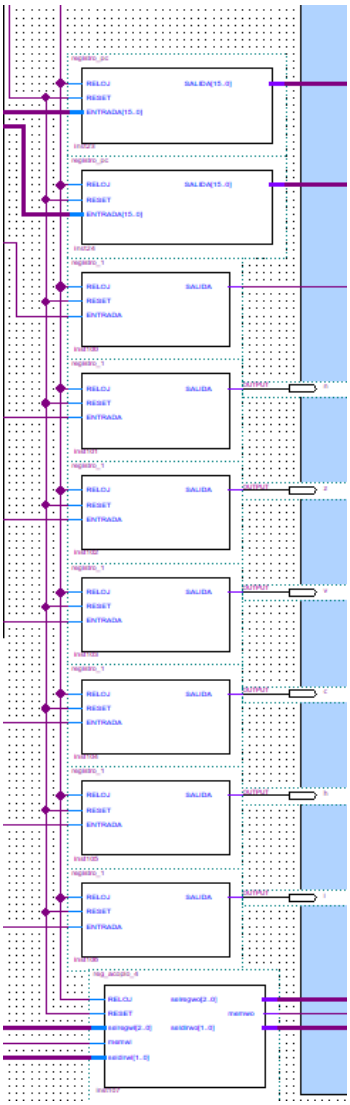


Figura 7: Etapa 3 [1, p, 139]



[2]Ya que la etapa 04 es solo una salida o las banderas que se activan durante todo el proceso

Figura 9: Etapa 4

III-D. Resolución del algoritmo

Primeramente definimos el problema, es la división entre 2 de el producto de dos números.

III-D1. División: La división se puede simular haciendo que se recorra un bit a la derecha, por lo tanto, basándonos del set de instrucciones la instrucción perfecta es ASRB[3] (0057[4]) [2, p. 24], podemos implementarlo en RISC definiendo las siguientes señales de control[5]: Por lo tanto, el

Tabla I: Señales de control de ASRB (0057)

$SelRegR$	5
$SelS1$	0
$S/\bar{R}$	1
Cin	0
$SelS2$	0
$SelDato$	1
$SelSrc$	1
$SelDir$	0
$SelOp$	7
$SelResult$	1
$SelC$	1
$Cadj$	0
$SelFlags$	3
$SelBranch$	0
VF	1
$SelRegW$	4
$MemW$	0
$SelDirW$	0

fragmento en el archivo u_control.vhd seria el siguiente:

Código 1: ASRB en u_control.vhd

```
elsif inst = X"0057" then – asrb (inh) Corrimiento a la derecha en b
                selregr <= X"5";
                sels1 <= '0';
                sr <= '1';
                cin <= '0';
                sels2 <= '0';
                seldato <= '1';
                selsrc <= "001";
                seldir <= "00";
                selop <= X"7";
                selresult <= "01";
                selc <= '1';
                cadj <= '0';
                selfalgs <= X"3";
                selbranch <= "000";
                vf <= '1';
                selregw <= "100";
                memw <= '0';
                seldirw <= "00";
```

[3]Notación Mnemónico.

[4]Instrucción.

[5]Se usa formato para las tablas, en código se varia de hexadecimal x"##" y binario "##".

III-D2. Multiplicación: Tenemos dos opciones:, 1) realizar el módulo de multiplicación para la arquitectura en RISC y 2) realizar la multiplicación usando software. Se optó por usar software, por lo tanto definimos nuestro algoritmo en la cual nos basaremos para implementarlo en ensamblador y después pasarlo a memoria. Teniendo

Algorithm 1: Algoritmo de multiplicación propuesto

Result: $perimetro \times apotema$
$a \leftarrow perimetro$;
$b \leftarrow apotema$;
$suma\_auxiliar \leftarrow a$;
$i \leftarrow 0$;
while $i \neq b$ do
  $suma\_auxiliar \leftarrow suma\_auxiliar + a$;
  $i \leftarrow i + 1$;

este punto resuelto buscamos que instrucciones nos pueden servir[2, pp. 24-26]:

- LDAA
  - Acceso Inmediato: Carga en el registro ACCA un dato inmediato de 16 bits contenido en memoria.
  - Acceso Directo: Carga en el acumulador A, un dato inmediato de 8 bits contenido en memoria.

Tabla II: LDAA

	Acceso Inmediato (0086)	Acceso Directo (0096)
$SelRegR$	0	0
$SelS1$	0	0
$S/\bar{R}$	1	1
Cin	0	0
$SelS2$	0	0
$SelDato$	1	1
$SelSrc$	3	2
$SelDir$	0	1
$SelOp$	4	4
$SelResult$	1	1
$SelC$	1	1
$Cadj$	0	0
$SelFlags$	1	1
$SelBranch$	0	0
VF	1	1
$SelRegW$	1	1
$MemW$	0	0
$SelDirW$	0	0

Código 2: LDAA (0086) de acceso inmediato en u_control.vhd

```
elsif inst = X"0086" then -- ldaa #dato_16bits (imm)
                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";
```

Código 3: LDAA (0096) de acceso directo en u_control.vhd

```
elsif inst = X"0096" then -- ldaa #dir_8bits (dir)
                selregr <= X"0";
                sels1 <= '0';
                sr <= '1';
                cin <= '0';
                sels2 <= '0';
                seldato <= '1';
                selsrc <= "010";
                seldir <= "01";
                selop <= X"4";
                selresult <= "01";
                selc <= '1';
                cadj <= '0';
                selfalgs <= X"1";
                selbranch <= "000";
                vf <= '1';
                selregw <= "001";
                memw <= '0';
                seldirw <= "00";
```

- STAA Suma los contenidos de los registros acumuladores A y B. El resultado es guardado en el acumulador A.

Código 4: STAA ($00B7$) en u_control.vhd

```
elsif inst = X"00B7" then -- staa #dir_16bits (ext)
                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";
```

Tabla III: STAA ($00B7$)

$SelRegR$	4
$SelS1$	1
$S/\bar{R}$	1
Cin	0
$SelS2$	0
$SelDato$	1
$SelSrc$	1
$SelDir$	0
$SelOp$	4
$SelResult$	1
$SelC$	1
$Cadj$	0
$SelFlags$	1
$SelBranch$	0
VF	1
$SelRegW$	0
$MemW$	1
$SelDirW$	2

```
                memw <= '1';
                seldirw <= "10";
```

- LDAB
  - Acceso Inmediato: Carga en el registro ACCB un dato inmediato de 16 bits contenido en memoria.
  - Acceso Directo: Carga en el acumulador B, un dato inmediato de 8 bits contenido en memoria.

Tabla IV: LDAB

	Acceso Inmediato ($00C6$)	Acceso Directo ($00D6$)
$SelRegR$	0	0
$SelS1$	0	0
$S/\bar{R}$	1	1
Cin	0	0
$SelS2$	0	0
$SelDato$	1	1
$SelSrc$	3	2
$SelDir$	0	1
$SelOp$	4	4
$SelResult$	1	1
$SelC$	1	1
$Cadj$	0	0
$SelFlags$	1	1
$SelBranch$	0	0
VF	1	1
$SelRegW$	4	4
$MemW$	0	0
$SelDirW$	0	0

Código 5: LDAB ($00C6$) de acceso inmediato en u_control.vhd

```
elsif inst = X"00C6" then -- ldab #dato_16bits (imm)
                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";
```

Código 6: LDAB ($00D6$) de acceso directo en u_control.vhd

```
elsif inst = X"00D6" then -- ldab #dir_8bits (dir)
            selregr <= X"0";
            sels1 <= '0';
            sr <= '1';
            cin <= '0';
            sels2 <= '0';
            seldato <= '1';
            selsrc <= "010";
            seldir <= "01";
            selop <= X"4";
            selresult <= "01";
            selc <= '1';
            cadj <= '0';
            selfalgs <= X"1";
            selbranch <= "000";
            vf <= '1';
            selregw <= "100";
            memw <= '0';
            seldirw <= "00";
```

- CBA (0011) Suma el acumulador A más el acumulador B y lo almacena en el acumulador A.

Tabla V: CBA

$SelRegR$	1
$SelS1$	0
$S/\bar{R}$	1
Cin	0
$SelS2$	0
$SelDato$	1
$SelSrc$	1
$SelDir$	0
$SelOp$	2
$SelResult$	0
$SelC$	1
$Cadj$	1
$SelFlags$	3
$SelBranch$	0
VF	1
$SelRegW$	0
$MemW$	0
$SelDirW$	0

Código 7: CBA (0011) en u_control.vhd

```
elsif inst = X"0011" then -- cba (inh)
            selregr <= X"1";
            sels1 <= '0';
            sr <= '1';
            cin <= '0';
            sels2 <= '0';
            seldato <= '1';
            selsrc <= "001";
            seldir <= "00";
            selop <= X"2";
            selresult <= "00";
            selc <= '1';
            cadj <= '1';
            selfalgs <= X"3";
            selbranch <= "000";
            vf <= '1';
            selregw <= "000";
            memw <= '0';
            seldirw <= "00";
```

- JMP ($007E$) Salta a una instrucción de la memoria.

Tabla VI: JMP

$SelRegR$	126
$SelS1$	0
$S/\bar{R}$	0
Cin	0
$SelS2$	1
$SelDato$	1
$SelSrc$	3
$SelDir$	0
$SelOp$	4
$SelResult$	1
$SelC$	0
$Cadj$	0
$SelFlags$	0
$SelBranch$	0
VF	0
$SelRegW$	0
$MemW$	0
$SelDirW$	0

Código 8: JMP ($007E$) en u_control.vhd

```
elsif inst = X"007E" then -- jmp #dir_16bits (ext)
            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";
```

Código 9: Pseudocódigo ensamblador que nos auxiliara para implementarlo en la memoria, se usa como entradas 6 y 8

```
ldaa 6 ; Valor de entrada A
staa 2
ldaa 8 ; Valor de entrada B
staa 3

ldaa 0 ; iterador
staa 0

ldaa 2 ; Auxiliar
staa 4

ldab 3 ; B

cba ; Si ACCB es diferente a ACCA, salta la siguiente instrucción, si no, se va a la instrucción 28
jmp 28

ldaa 4
ldab 2
aba
ldab 3
staa 4
ldaa 0
inca

staa 0
jmp 12

ldab 4
acrb
```

Teniendo el código ensamblador de referencia escribimos en memoria (memoria_inst.vhd).

Código 10: memoria_inst.vhd

```
-- memoria de solo lectura

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity memoria_inst is
  Port( direccion : in STD_LOGIC_VECTOR (15 downto 0);
        datos : out STD_LOGIC_VECTOR (31 downto 0));
end memoria_inst;

architecture Behavioral of memoria_inst is

  type memory is array(0 to 67) of std_logic_vector(31 downto 0);
  signal memoria : memory;
begin

  --6 * 8 ----------------------------------------
  memoria(0) <= x"00860028"; -- A=6 declaracion a  «== VALOR DE ENTRADA
  memoria(1) <= x"00010000"; --
  memoria(2) <= x"00B70002"; -- M2 = 6
  memoria(3) <= x"00010000"; --  Fin declaracion a
  memoria(4) <= x"00010000"; --
  memoria(5) <= x"00860006"; -- A=8 declaracion b  «== VALOR DE ENTRADA
  memoria(6) <= x"00010000"; --
  memoria(7) <= x"00B70003"; -- M3 = 8
  memoria(8) <= x"00010000"; --  Fin Declaración b
  memoria(9) <= x"00010000"; --
  memoria(10) <= x"00860001"; -- A=0 declaracion i
  memoria(11) <= x"00010000"; --
  memoria(12) <= x"00B70000"; -- M0 = 1
  memoria(13) <= x"00010000"; --  Fin Declaración i
  memoria(14) <= x"00010000"; --
  memoria(15) <= x"00960002"; -- A=M2 declaracion aux
  memoria(16) <= x"00010000"; --
  memoria(17) <= x"00B70004"; -- M4 = M2
  memoria(18) <= x"00010000"; --  Fin declaracion aux
  memoria(19) <= x"00010000"; --
  memoria(20) <= x"00010000"; --
  memoria(21) <= x"00010000"; --
  memoria(22) <= x"00D60003"; --  Etiqueta
  memoria(23) <= x"00960000"; --
  memoria(24) <= x"00010000"; --
  memoria(25) <= x"00010000"; --
  memoria(26) <= x"00110000"; --  if a = b then div
  memoria(27) <= x"00270031"; --
  memoria(28) <= x"00010000"; --
  memoria(29) <= x"00010000"; --
  memoria(30) <= x"00010000"; --
  memoria(31) <= x"00010000"; --  else
  memoria(32) <= x"00960004"; -- a=m4(aux)
  memoria(33) <= x"00D60002"; -- b=m2(a)
  memoria(34) <= x"00010000"; --
  memoria(35) <= x"00010000"; --
  memoria(36) <= x"001B0000"; -- a=a+b
  memoria(37) <= x"00010000"; --
  memoria(38) <= x"00D60003"; -- b=m3(b)
  memoria(39) <= x"00010000"; --
  memoria(40) <= x"00B70004"; -- M4=a(aux)
  memoria(41) <= x"00960000"; -- a=m0(i)
  memoria(42) <= x"00010000"; --
  memoria(43) <= x"004C0000"; -- a++
  memoria(44) <= x"00010000"; --
  memoria(45) <= x"00010000"; --
  memoria(46) <= x"00B70000"; -- m0(i)=a
  memoria(47) <= x"007E0016"; --
  memoria(48) <= x"00010000"; --
  memoria(49) <= x"00010000"; --
  memoria(50) <= x"00D60004"; -- b=m4(aux) div
  memoria(51) <= x"00010000"; --
  memoria(52) <= x"00010000"; --
  memoria(53) <= x"00010000"; --
  memoria(54) <= x"00570000"; --
  memoria(55) <= x"00000000"; --
  memoria(56) <= x"00000000"; --
  memoria(57) <= x"00000000"; --
  memoria(58) <= x"00000000"; --
  memoria(59) <= x"00000000"; --
  memoria(60) <= x"00000000"; --
  memoria(61) <= x"00000000"; --
  memoria(62) <= x"00000000"; --
  memoria(63) <= x"00000000"; --
  memoria(64) <= x"00000000"; --
  memoria(65) <= x"00000000"; --
  memoria(66) <= x"00000000"; --
  memoria(67) <= x"00000000"; --

process(direccion)
begin
  datos <= memoria(conv_integer(unsigned(direccion)));
```

```
  end process;
end Behavioral;
```

Se añadieron instrucciones NOP para resolver el problema de la dependencia de datos, evitando así diversos retrasos e inconsistencias.

IV. Resultado

IV-A. Prueba

Ahora seguimos las instrucciones de la sección VI-C para ejecutar el algoritmo implementado en una arquitectura RISC.

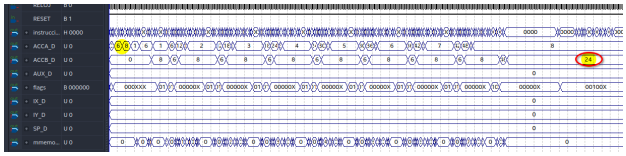


Figura 10: Resultado de $\frac{6\times8}{2}$

IV-B. Octagono "Real"

Se propone un octágono, con valores enteros de un perimetro a $40[u]$ y un apotema de $6[u]$, donde su área resultante seria 120 (En la siguiente figura se muestra un trazo "real" usando geogebra). Modificamos los valores de

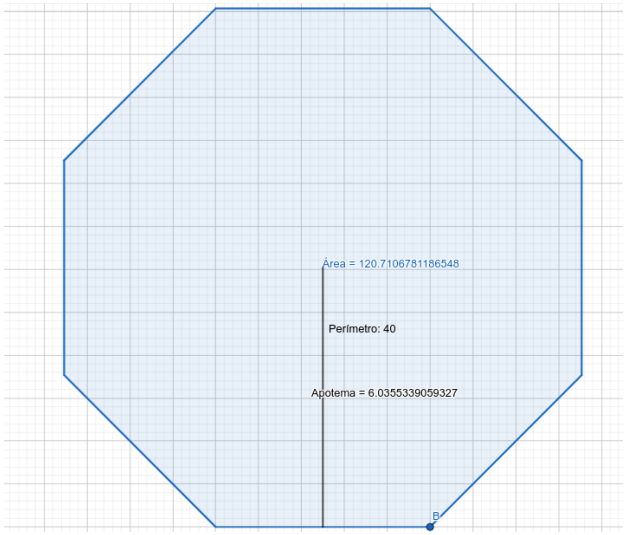


Figura 11: Octágono en Geogebra: *perimetro* $= 40$ y *apotema* ≈ 6

entrada del archivo memoria_inst.vhd

```
– Resto de código
memoria(0) <= x"00860028"; – A=40 declaracion a  «== VALOR DE
↪  ENTRADA
memoria(1) <= x"00010000"; –
memoria(2) <= x"00B70002"; – M2 = 6
memoria(3) <= x"00010000"; –  Fin declaracion a
memoria(4) <= x"00010000"; –
memoria(5) <= x"00860006"; – A=6 declaracion b  «== VALOR DE
↪  ENTRADA
– El resto del código
```

Compilamos y simulamos Donde apreciamos que el valor del área en geogebra es cercano a nuestros resultados

$$A_{geogebra} = \frac{40 \times 6.03}{2} = 120.71$$

$$A_{quartus} = \frac{40 \times 6}{2} = 120$$

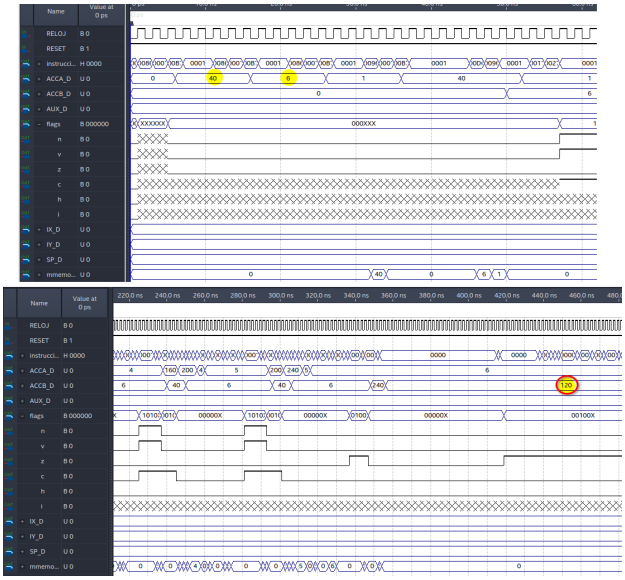


Figura 12: Simulación de: *perimetro* $= 40$ y *apotema* ≈ 6

V. Conclusiones

V-1. Brito Sánchez Diego: Con el desarrollo de este proyecto pusimos en práctica todo lo aprendido a lo largo del curso, tanto teórico como práctico, donde se logró implementar una arquitectura RISC y sobre ella desarrollar las instrucciones necesarias para obtener el algoritmo que calcula el área de un octágono, por lo que podemos decir que se logró el objetivo de este proyecto, ya que comprendimos el funcionamiento de esta arquitectura y como se mostró en las simulaciones también nuestro algoritmo funcionó como se esperaba.

V-2. Castelan Hernandez Mario: Se logró cumplir el objetivo, ya que implementamos las instrucciones necesarias para ejecutar el algoritmo del área de un octágono en una arquitectura RISC que desarrollamos en el laboratorio, además pusimos en práctica los conocimientos adquiridos durante el semestre para poder entender como funciona esta arquitectura y poder desarrollar el programa en ensamblador que realiza el algoritmo

V-3. Miranda Hernández Alejandro: En este proyecto pusimos en práctica lo aprendido en teoría acerca de la arquitectura RISC, como se dividen sus instrucciones en 4 etapas. Se compararon las arquitecturas RISC y CISC, y se observó como una arquitectura RISC puede ejecutar instrucciones en paralelo al ser estas divididas en etapas, mientras que la arquitectura CISC las ejecuta secuencialmente. Por último, se observó que únicamente la arquitectura RISC puede tener errores por dependencia de datos, y estos tienen que ser solventados mediante hardware o software. También comprendimos el funcionamiento de la arquitectura RISC del Motorola 68HC11. Se diseñaron los esquemas correspondientes para instrucciones de multiplicación y de división, posteriormente se configuraron las señales correspondientes en un archivo de Excel. Finalmente se agregaron a la memoria en su archivo VHDL. Después se busco modificar la memoria RAM de instrucciones, para crear una secuencia que nos permitiera calcular el área de un octágono, a partir de que nos proporcionaran su perímetro y su apotema. Por todo lo anteriormente menciona se puede afirmar que se cumplió el objetivo del proyecto.

V-4. Romero Andrade Cristian: Se desarrolló la arquitectura Risc, donde se puede observar que la ejecución de cada instrucción es paralela, esto conlleva a una velocidad de procesamiento considerable en comparación a otras arquitecturas. sin embargo esta contiene un problema ya que tiene una dependencia de datos para cada instrucción y puede causar retrasos e inconsistencias, sin embargo esta se puede solucionar usando la operación NOP (el la práctica estas interrupciones se encarga el compilador o bien ya esta resuelta por hardware).

VI. Manual de usuario

VI-A. Prerrequisitos

- Contar con Git instalado en su sistema operativo (Opcional)
- Contar con alguno de los siguiente sistemas operativos:
  - Windows* 10
  - Windows Server* 2012 Enterprise
  - Windows Server* 2016 Enterprise
  - Windows Server* 2019 Enterprise
  - Red Hat* Enterprise Linux* 7
  - Red Hat* Enterprise Linux* 8
  - CentOS* 7.5
  - CentOS* 8.0
  - SUSE* SLE 12
  - SUSE* SLE 15
  - Ubuntu* 16.04 LTS
  - Ubuntu* 18.04 LTS
  - Ubuntu* 20 LTS
- El tamaño de memoria dependerá de la versión descargada

Tabla VII: Versiónes de Quartus

Software	Espacio minimo
Quartus Prime Pro	$20 - 140[GB]$
Quartus Prime Standard Edition	$15 - 37[GB]$
Quartus Prime Lite Edition	$14[GB]$
Stand-Alone Programmer	$3.3[GB]$
Intel FPGASDK for OpenCL	$2[GB]$
Intel SoC Embedded Development Suite	$8[GB]$
Intel Advanced Link Analyzer	$9[GB]$

VI-B. Instalación

Descargar o clonar el repositorio de Github: github.com/tysyak/OyAC_Proyecto_20221

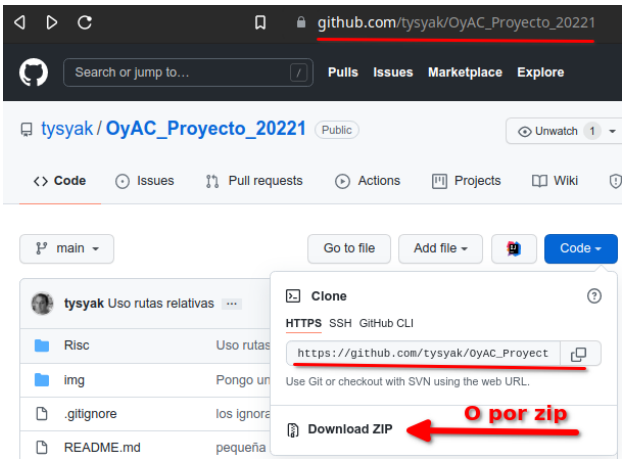
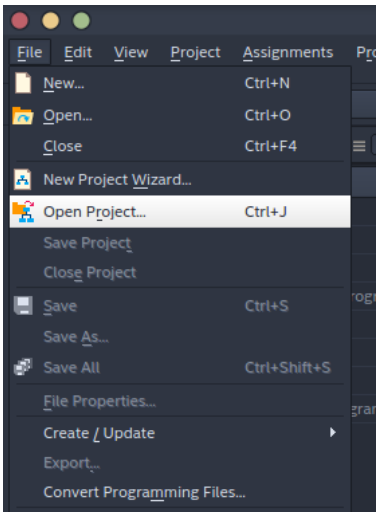


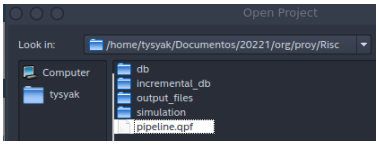
Figura 13: Repositorio del proyecto

VI-C. Uso

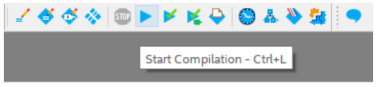
1. Abrir Quartus Prime[6]
2. En el menú File seleccionar abrir proyecto o presionar Control + J



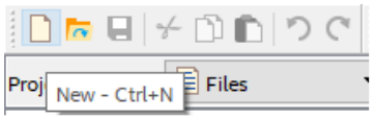
3. Seleccionamos el proyecto (pipeline.qpf)



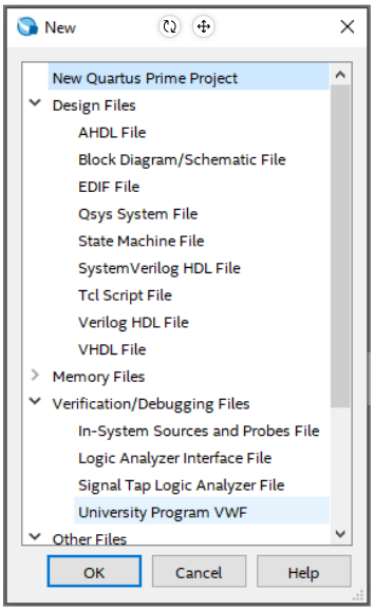
4. Compilar el proyecto con el botón o presionando Control + L



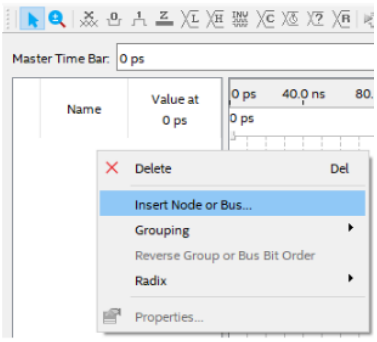
5. Crear un nuevo archivo



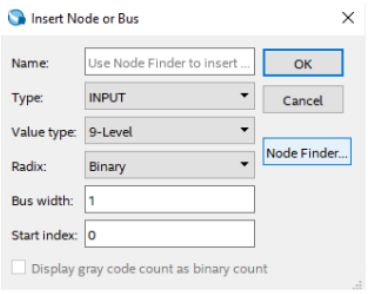
6. Seleccionar el tipo, University Program VWF



7. Presionar click derecho sobre el espacio blanco y seleccionar insert Node or Bus

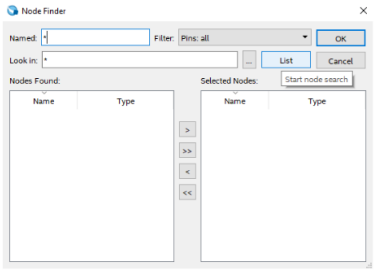


8. Seleccionar Node Finder

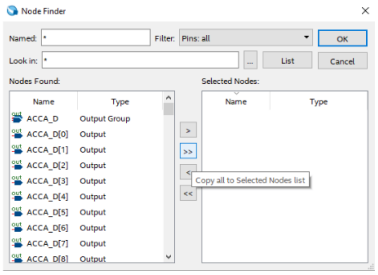


9. Presionar el botón List, esto desplegara los nodos en el proyecto

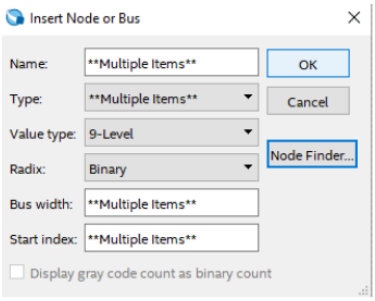
[6] A partir Quartus v21.1 modelsim es sustituido, por lo tanto la solución en la simulación vista en el presente solo sirve para versiones anteriores a 21.1



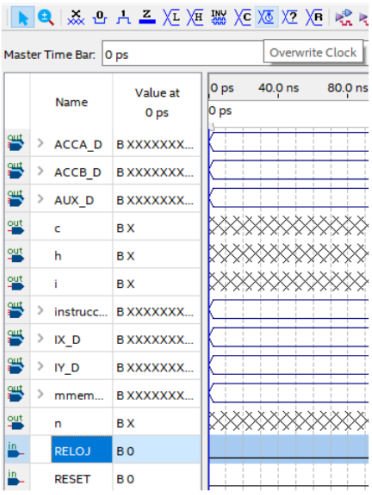
10. Dar click sobre el botón >> y después dar click en el botón OK



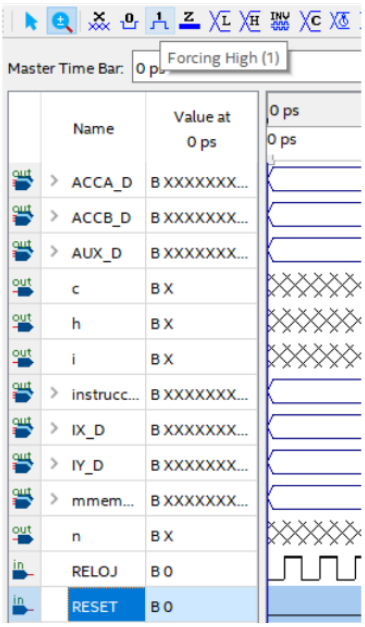
11. Dar click en el botón OK



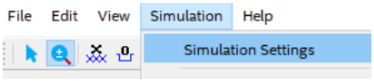
12. Seleccionar el RELOJ y dar click sobre el botón 'Overwrite Clock', mostrado en la parte superior de la imagen



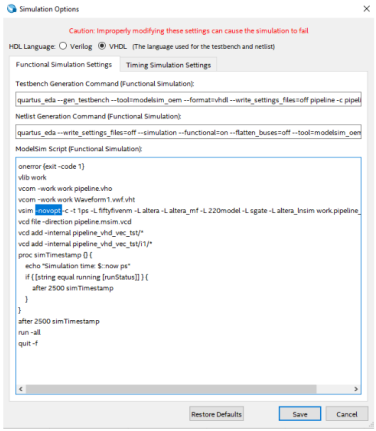
13. Asignar un periodo de 5.0 y dar click sobre 'OK'.
14. Seleccionar RESET y dar click sobre el botón 'ForCing High (1)', mostrado en la parte superior de la imagen



15. Seleccionar del menú Simulation Settings



16. Borrar del Script la opción -novopt (se muestra seleccionado en la imagen siguiente). Después presionar sobre SAVE



17. Presionar el botón Run Functional Simulation

VII. Referencias

Referencias

[1] J. Savage y G. Vázquez, Diseño de microprocesadores. Facultad de Ingenieria.

[2] P. Musumeci, 68HC11 Programmer's Reference Manual. IEEE, 1999, vol. 1.7.

[3] O. M. Albert y G. E. Manonellas, El computador. Univesitat Oberta de Catalunya.

[4] Diferencias RISC y CISC: Comparamos el diseño basico de CPU, jul. de 2021. dirección: https://www.profesionalreview.com/2021/07/18/risc-vs-cisc/.

Índice de figuras

Índice de tablas

Índice de Códigos