Universidad Autónoma de Zacatecas

"Francisco García Salinas"

Unidad Académica de Ingeniería Eléctrica

Programa: Ingeniería en Computación

Materia: Microprocesadores

Profesor: Remberto Sandoval Arechiga

Reporte Final Micro

Equipo 2: Microbitos

- Javier Carrillo Trejo (IC)
- Christian Josué González Lozano (IC)
- Carlos Alberto Martinez Falcon (IC)
- Jorge Antonio Cruz González (IRM)
- Jhonatan Ivan Davila Muro (IC)
- Andrea Jaramillo Leiva (IC)

1. Resumen

En esta práctica se construye un microprocesador RISC de 8-bits con arquitectura Harvard, dicha arquitectura tiene como características principales la separación de almacenamiento para instrucciones y para datos y por otro lado el tipo de procesamiento RISC que sólo usan instrucciones sencillas que se puedan ejecutar rápidamente. Para el diseño de nuestra arquitectura se utilizó completamente la herramienta de Vivado en su versión 2018.3.

El microprocesador tiene como entrada 9 bits para las instrucciones y 8 bits para los datos y tendrá 3 salidas de 8 bits cada una; una salida correspondiente al direccionamiento de instrucciones, una segunda correspondiente al direccionamiento de datos y una tercera dirigida a la escritura de los datos [1].

Las instrucciones permitidas por el microprocesador son las siguientes:

Instruction	Arguments	Description	Comments
LOAD	RX,#NUM	Load #Num to register X	#Num is 3 bits [0,7]
LOAD	RX,[RY]	Load data at address [RY] from memory	RY and RX are 3 bits[0,7]
STORE	#NUM	Store #Num to [RX] address memory	#Num is 3 bits [0,7]
STORE	[RX],RY	Stores data at Register RY in [RX] memory address	RY and RX are 3 bits [0,7]
MOVE	RX,RY	Move data form register RY to RX	RY and RX are 3 bits [0,7]
MATH	RX,OP	DO MATH OPERATION WITH RX, AND STORES RESULT IN R0	OP: 0: R0=R0+RX 1: R0=R0-RX 2: R0= R0<<RX 3: R0= R0>>RY 4: R0=~RX 5: R0=R0&RX 6: R0 = R0|RX 7: R0=R0^RX
JUMP	[RX],COND	JUMP PC TO [RX] ADDRESS IF COND IS TRUE	COND: 0:NO CONDITION 1: NO CONDITION SAVE PC IN R7 2:Z FLAG IS TRUE 3:Z FLAG IS FALSE 4: C FLAG IS TRUE 5: C FLAG IS FALSE 6: N FLAG IS TRUE 7: N FLAG IS FALSE
NOP		NO OPERATION	

Tabla 1.1. Set de instrucciones permitidos por el microprocesador

2. Índice

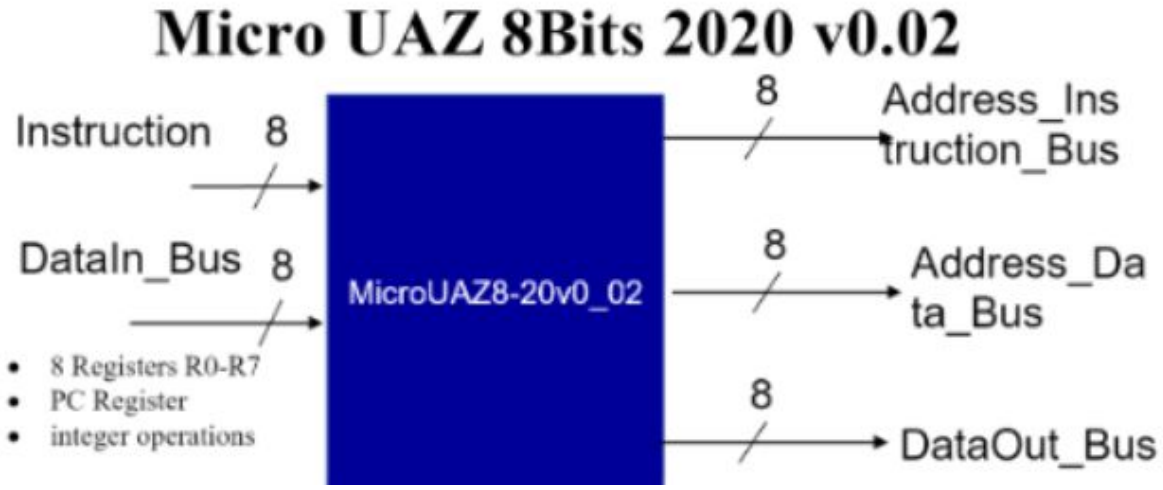
3. Introducción

Los microprocesadores en la actualidad son una herramienta fundamental, para el desarrollo de nuevas tecnologías, también conocidos como sistemas embebidos, que se refiere a un sistema de computación diseñado para realizar algunas funciones dedicadas frecuentemente en algún sistema en tiempo real, estos se pueden programar directamente en el lenguaje ensamblador o microprocesador incorporado sobre él mismo[1], estos sistemas están compuestos por un microprocesador y software que se ejecute sobre éste, se le denomina “micro” por pequeño, en relación a la importancia de su función en un dispositivo, este se encarga de procesar y ejecutar las instrucciones codificadas en números binarios, actualmente es considerado como el componente electrónico más influyente en la vida del ser humano[2].cabe mencionar que en este documento se le darán a conocer los requerimientos para la implementación de este proyecto, el funcionamiento interno, la implementación, las partes que lo componen, la explicación en base a las simulaciones realizadas, así como las pruebas del mismo, con el objetivo de que sea entendible su funcionamiento interno y externo, ya que se utiliza una arquitectura Harvard que es actualmente la arquitectura utilizada para supercomputadoras, en los microcontroladores, y sistemas integrados en general, lo que la caracteriza de las otras arquitecturas es que tiene además de la memoria, el procesador tiene los buses segregados, de modo que cada tipo de memoria tiene un bus de datos, uno de direcciónes y uno de control, y esta arquitectura nos permite adecuar el tamaño de los buses a las características de cada tipo de memoria, por lo que el procesador puede acezar a cada una de ellas de forma simultánea, lo que garantiza velocidad de procesamiento. Es necesario recordar que un microprocesador se puede configurar de diferentes maneras, siempre y cuando se respete el tamaño de memoria para el correcto funcionamiento, está compuesto por diferentes partes básicas como lo son un decodificador que en general se encarga de habilitar solo un subcomponente, Banco de registros permite trabajar a gran velocidad, Data_sel no es más que un selector de datos que se encarga de cual dato entra a el banco de registros ya sea una dirección, número, registro o un resultado de ALU, BIU es la unidad de interfaz de bus, ALU circuito que permite operaciones lógicas y aritméticas, y UCJ se encarga de emitir una señal de control externa para reducir el intercambio de datos con módulos de entrada y

salida, emite una señal interna para transferir datos entre registros, también se encarga de emitir una señal de control interna para transferir datos entre registros, hace que la ALU ejecute una función concreta y regula otras operaciones internas y se encarga de analizar y extraer las instrucciones de memoria central, para lo que ocupa 2 registros, estos componentes de microprocesador son la base para su implementación, ya que se usan desde un notebook, un celular, Tablet entre otros dispositivos.

4. Requerimientos

Diagrama de caja negra



Nombre de la señal	Dirección	Tamaño (bits)	Descripción
i_Instruccion	Entrada	9	Dato de entrada que indica la instrucción que va a realizar el microprocesador además de contener los datos necesarios para realizar la instrucción
i_DataIn_Bus	Entrada	8	Dato de entrada que se almacenará en los registros cuando el programa lo solicite
o_Address_Instruction_Bus	Salida	8	Salida de datos que indica la dirección de la siguiente instrucción a realizar.
			Salida de datos que

o_Address_Data_Bus	Salida	8	indica la dirección de memoria de donde se obtendrá un dato o desde la que se leerá un dato.
o_DataOut_Bus	Salida	8	Salida de datos la cual contiene el dato que se escribirá en memoria
W_R	Salida	1	Le indica a la memoria si la operación a realizar es lectura o escritura de datos

Tabla 4.1. Descripción de entradas y salidas del diagrama de caja negra MicroUaz 8bits.

5. Arquitectura

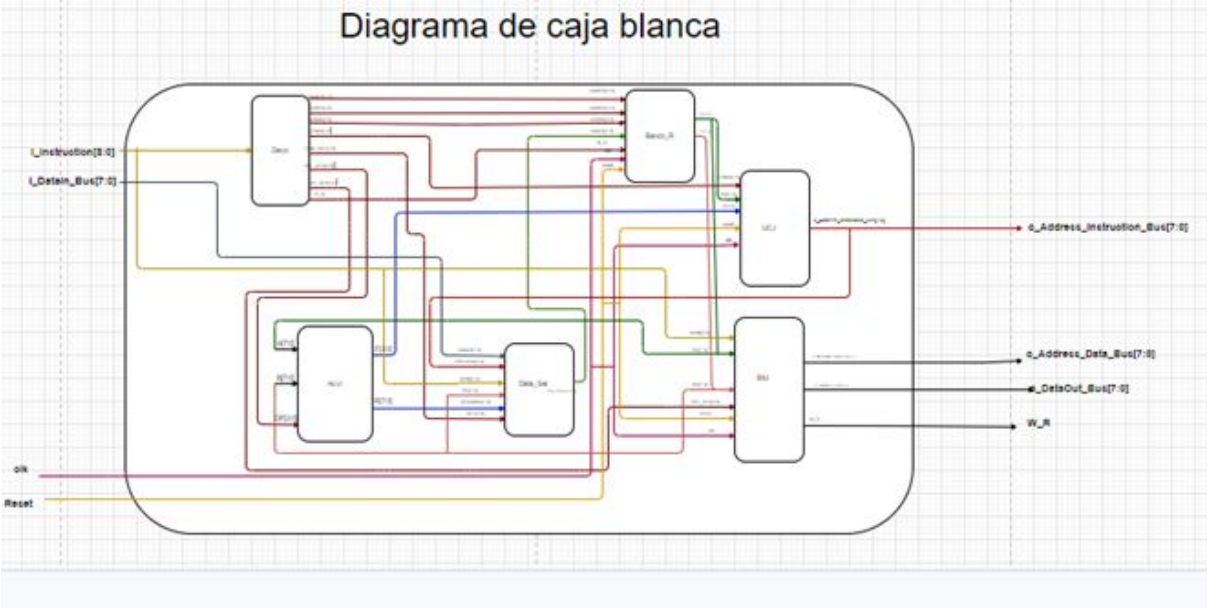


Figura 5. Diagrama de caja blanca

Descripción de cada componente del diagrama de caja blanca.

Decodificador

Un **decodificador** o descodificador es un circuito combinacional, cuya función es inversa a la del codificador, es decir, convierte un código binario (natural, BCD, etc.) de N bits de entrada y M líneas de salida (N puede ser cualquier entero y M es un entero menor o igual a 2^N), tales que cada línea de salida será activada para una sola de las combinaciones posibles de entrada. Normalmente, estos circuitos suelen encontrarse como **decodificador / demultiplexor.** Esto es debido a que un demultiplexor puede comportarse como un decodificador.

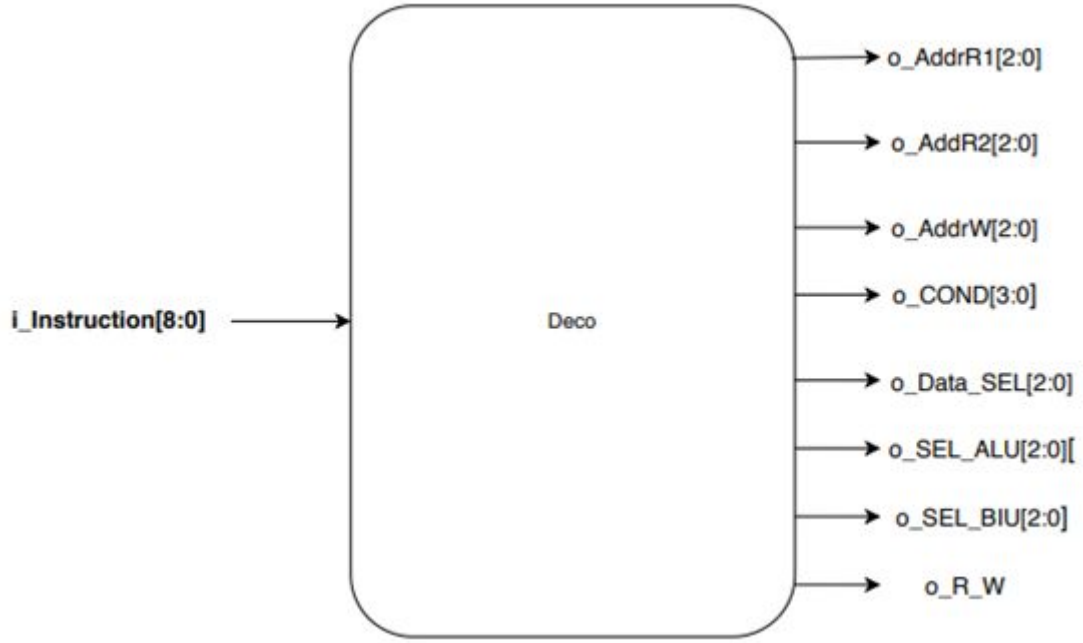


Figura 5.1. Diagrama de caja negra del decodificador

Tabla 5.1. Descripción de entradas y salidas del decodificador

o_R_W	Dirección	Tamaño (Bits)	Descripción
i_Instruction[8:0]	Entrada	9	Son las instrucciones que recibe el decodificador para ser procesada y llevar a cabo tareas que se realizan para completar cualquiera de las instrucciones dentro del set.
o_AddrR1[2:0]	Salida	3	Apunta a una dirección específica dentro del banco de registros
o_AddR2[2:0]	Salida	3	Apunta a una dirección específica dentro del banco de registros
o_AddrW[2:0]	Salida	3	Apunta a un registro específico del banco de registros dónde serán escritos los datos

o_COND[3:0]	Salida	4	Encargada de mandar señal a el módulo UCJ para definir que condición de salto se realizará
o_Data_SEL[2:0]	Salida	3	Encargada de mandar señal a el módulo Data_SEL para seleccionar que dato se utilizará
o_SEL_ALU[2:0][	Salida	3	Encargada de mandar señal a el módulo Data_SEL para seleccionar la operación que se realizará
o_SEL_BIU[1:0]	Salida	3	Encargada de mandar señal a el módulo Data_SEL para seleccionar la operación que se realizará

Tabla de verdad

	i_Instruction[8:0]					o_AddrR1			o_AddrR2			o_AddrW			o_COND				o_Data_SEL			o_SEL_ALU			o_SEL_BIU			o_R_W
	[8:6]			[5:3]	[2:0]	[2:0]			[2:0]			[2:0]			[3:0]				[2:0]			[2:0]			[2:0]			
Load	0	0	0	Rx	NUM	0	0	0	0	0	0		Rx		1	1	1	1	0	1	0	0	0	0	1	1	1	1
Load	0	0	1	Rx	[Ry]	0	0	0		[Ry]			Rx		1	1	1	1	0	0	0	0	0	0	0	0	0	1
Store	0	1	0	[Rx]	NUM	0	0	0	0	[Rx]	0		0		1	1	1	1	1	0	1	0	0	0	0	0	1	0
Store	0	1	1	[Rx]	Ry	0	0	0		Rx			Ry		1	1	1	1	0	1	1	0	0	0	0	1	0	0
Move	1	0	0	Rx	Ry	0	0	0		Ry			Rx		1	1	1	1	0	1	1	0	0	0	1	1	1	1
Math	1	0	1	Rx	OP	0	0	0		Rx		0	0	0	1	1	1	1	1	0	0	OP			1	1	1	1
JMP	1	1	0	Rx	COND		Rx		0	0	0	IF cond= 1, o_AddrW=111 IF cond=1 ,o_AddrW=000			COND				IF cond= 1, data_sel=010 IF cond=1, data_sel=011			0	0	0	1	1	1	IF cond= 1, o_R_W=1 IF cond=1, o_R_W=0
NOP	1	1	1	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	1	1	0

Tabla 5.2. Tabla de verdad del funcionamiento del decodificador

Banco de Registros

Tradicionalmente el banco de registros se ha dividido en dos grupos: los registros de propósito general y los que tienen una función específica. Al primer grupo pertenecen aquellos que el programador puede usar libremente para almacenar temporalmente datos, mientras que los segundos se utilizan de manera indirecta. Los registros con función específica más usuales son:

- **Contador de programa:** Contiene la dirección de la memoria donde está alojada la siguiente instrucción a ejecutar. Actúa, por tanto, como un puntero y, de hecho, en algunos microprocesadores se denomina puntero de instrucción. Es la unidad de control la que utiliza este registro para recuperar las instrucciones del programa, incrementando su contenido a medida que se avanza en la ejecución o modificándolo cuando se encuentra una instrucción de salto.

- **Puntero de pila:** En ocasiones es necesario guardar temporalmente el contador de programa, por ejemplo al saltar a una subrutina o cuando el microprocesador debe atender una interrupción externa, con la intención de recuperarlo posteriormente. Los primeros microprocesadores contaban con una pila interna, en el propio circuito integrado, que tenía una capacidad limitada y solía permitir 5 u 8 niveles como máximo. Actualmente la pila se almacena en la memoria principal, externa al microprocesador de forma que éste lo único que necesita es conocer la dirección donde está el tope o parte alta de la pila. Almacenar dicha dirección es el objetivo del registro del puntero de pila.

- **Acumulador:** Puede ser utilizado como registro de propósito general en muchas situaciones, pero en otras adquiere el papel de registro específico al ser el destinatario de diferentes operaciones aritméticas, lógicas o de entrada/salida.

- **Estado:** Su denominación cambia según el tipo de diseño y fabricante, pero su finalidad es siempre la misma: mantener una serie de bits indicando el estado en que se encuentra el microprocesador. Ese estado proviene normalmente de la ejecución de la última instrucción, pudiendo influir en cómo se ejecutarán las posteriores. También es posible que ciertos bits modifiquen el modo de funcionamiento del procesador, de forma general o ante determinadas instrucciones.

- **Otros registros:** Si bien los cuatro citados pueden considerarse los más importantes, todos los microprocesadores disponen además de otros registros de uso específico, ocultos en su mayor parte que emplean para almacenar el código de la instrucción que está ejecutándose, contener temporalmente datos procedentes de memoria que van a intervenir en un cálculo, etc.

En un principio los microprocesadores contaban sólo con registros de 8 o 16 bits pensados para operar con aritmética entera, pero en la actualidad el tamaño ha crecido hasta los 32, 64 e incluso 80 bits, contemplándose tanto la aritmética entera como la de punto flotante. [4]

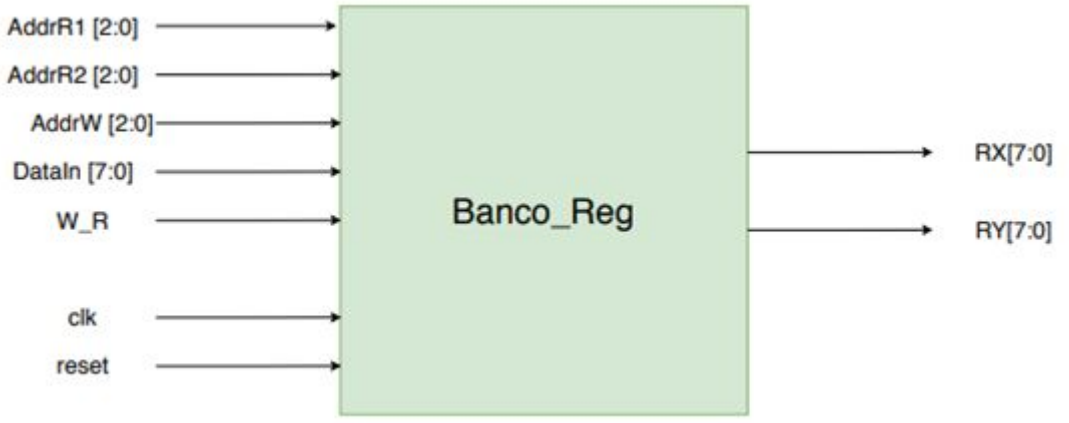


Figura 5.2. Caja negra banco de registros

Tabla 5.3. Descripción de entradas y salidas del banco de registros.

o_R_W	Dirección	Tamaño (Bits)	Descripción
AddR1[2:0]	Entrada	3	Addr1 se utiliza para seleccionar el registro que contiene la dirección de memoria donde se va a guardar un dato
AddR2[2:0]	Entrada	3	a través del Addr2 entran direcciones de registros desde el cual se va a leer el dato
AddW[2:0]	Entrada	3	EL AddrW se utiliza para apuntar a un registro donde se va a almacenar datos de 8 bits
DataIn[7:0]	Entrada	8	DataIn es el dato que se va a almacenar en un registro
W_R	Entrada	1	W_R Selecciona si en los registros que se esta apuntando se va a leer o se va a escribir.
clk	Entrada	1	clk señal de referencia de tiempo de 100mhz

reset	Entrada	1	Señal de reinicio a el estado inicial del banco de registros
RX[7:0]	Salida	8	RX es una señal de salida en el cual sale el dato que se encuentra en la dirección del registro 1 en este caso sería del AddR1
RY[7:0]	Salida	8	RY es una señal de salida en el cual sale el dato que se encuentra en la dirección del registro 2 en este caso sería del AddR2

Tabla 5.4. Tabla de verdad del banco de registros

Entrada R_W	Rx	Ry	AddrW
0	RX = R[AddrR1]	Ry=AddR2[2:0]	no operation
1	RX = R[AddrR1]	Ry=AddR2[2:0]	R[AddrW]=DataIn

UCJ

Unidad de control, esta es la parte de la CPU que se encarga de que las cosas ocurran, esta se encarga de emitir una señal de control externas a la CPU, para reducir el intercambio de datos con los módulos de E/S, también se encarga de emitir una señal de control internas para transferir datos entre registros, hace que la ALU ejecute una función concreta y regular otras operaciones internas. Se encarga de analizar y extraer las instrucciones de memoria central, para lo que necesita dos registros:

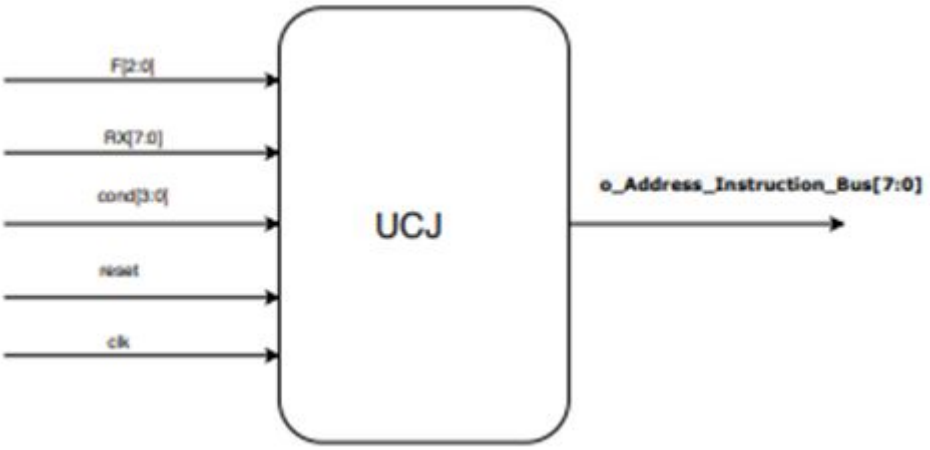


Figura 5.3 Caja negra del USJ

Tabla 5.5. Descripción de entradas y salidas del USJ

Nombre de Señal	Dirección	Tamaño (bits)	Descripción
F[2:0]	Entrada	3	Es una señal de entrada en el que se encuentra el estado de las banderas
RX[7:0]	Entrada	8	Señal de entrada o registro que pueden ser usadas para realizar operaciones
cond[3:0]	Entrada	4	Señal de entrada que decide cual condición se va a tomar para hacer el salto.
reset	Entrada	1	Señal de entrada que reinicia el submódulo UCJ a un estado inicial
clk	Entrada	1	Señal de entrada que se describe como señal de referencia temporal con una frecuencia de 100MHz
	Salida	8	Señal de salida que muestra la dirección de la instrucción, puede ir directamente a la

o_Address_Instruction_Bus[7:0]			salida o a selector de datos como Dirección[7:0]

Tabla 5.6. Descripción funcional del USJ

cond[3:0]	**o_Address_Instruction_Bus[7:0]**
cond=0	o_Address_=Rx
cond=1	o_Address_=Rx
cond=2	if z=1 o_Address=Rx else pc+1
cond=3	if z=0 o_Address=Rx else pc+1
cond=4	if C= 1 o_Address = RX else pc +1
cond=5	if C=0 o_Address=RX else pc + 1
cond=6	if N=1 o_Address=RX else pc +1
cond=7	if N=0 o_Address RX else pc+1
cond=F	pc+1

ALU

Conocida también como ALU (Arithmetic Logic Unit), podría decirse que es la calculadora interna del microprocesador, con capacidad para realizar operaciones aritméticas pero también de tipo lógico.

Las operaciones aritmético-lógicas que puede ejecutar por sí mismo un procesador depende del diseño de la ALU. Los x86 hasta el 80386, por ejemplo, contaban con una ALU que ofrecía únicamente operaciones con aritmética entera y solamente se contemplaban las cuatro operaciones aritméticas básicas. Las aplicaciones que requerían trabajar con coma flotante, y realizar operaciones más complejas, tenían que hacerlo por software, lo cual era lento, o bien requerir que el sistema contase con un coprocesador matemático.

En la actualidad los microprocesadores disponen de una ALU preparada para operar con aritmética entera y de punto flotante, ejecutando por hardware operaciones complejas que, de ser implementadas mediante software, requerirían mucho más tiempo.

La ALU cuenta con dos entradas, asumiendo que una de ellas siempre es el acumulador y que la otra, procedente de cualquier otro registro o de la memoria, está en un registro temporal. La única salida, en la parte superior, se dirige al acumulador. En el funcionamiento de la ALU también interviene el contenido del registro de estado, si bien éste no actúa como un tercer operando sino como un agente externo que puede influir en la operación que realice la ALU. Dicha operación vendrá dictada por la unidad de control que, a través del bus interno, controla también el funcionamiento de la ALU.[5]

Figura 5.3. Caja negra del ALU

Tabla 5.7. Descripción de entradas y salidas de ALU

Nombre de señal	Dirección	Tamaño(bits)	Descripción
A[7:0]	Entrada	8	Señal de entrada de datos almacenada en los registros

B[7:0]	Entrada	8	Señal de entrada de datos almacenada en los registros
OP[2:0]	Entrada	3	Señal de entrada que le permite saber el tipo de operación que está realizando
F[2:0]	Entrada	3	Señal de salida de información sobre los resultados o banderas
R[7:0]	Salida	8	Señal de salida o señal de registro de salida

Tabla 5.8. Descripción funcional ALU

OP	operación y resultado (R)
000	SUMA R0 + RX = se guarda resultado en R0
001	Resta R0 + RX = se guarda resultado en R0
010	Se recorre el numero binario de RX bits a la izquierda del valor R0
011	Se recorre el numero binario de RY bits a la izquierda del valor R0
100	R0 es igual al binario negativo de RX
101	Realiza la operación AND entre RX y R0

110	Realiza la operación OR entre RX y R0
111	Realiza la operación Xor entre RX y R0

Tabla 5.9. Banderas para el funcionamiento del ALU

Bandera	condicion
F=[0], Z=0	RI=0
F=[0], Z=1	R=0
F=[1] C=0	la operacion no genero acarreo
F=[1] C=1	La operacion genero acarreo
F=[2] N=0	El Resultado de la operación no es negativo
F=[2] N=1	Si el resultado de la operación es negativo

Selector de Datos

Un multiplexor también llamado selector de datos, es un circuito combinacional que selecciona una de n líneas de entrada (donde n es un número entero positivo) y transmite su información binaria a la salida. La selección de la entrada es controlada por un conjunto de líneas de selección o de control, La relación de las líneas de entrada y líneas de selección está dada por la expresión 2^n, donde n corresponde al número de líneas de selección y 2^n al número de líneas de entrada.

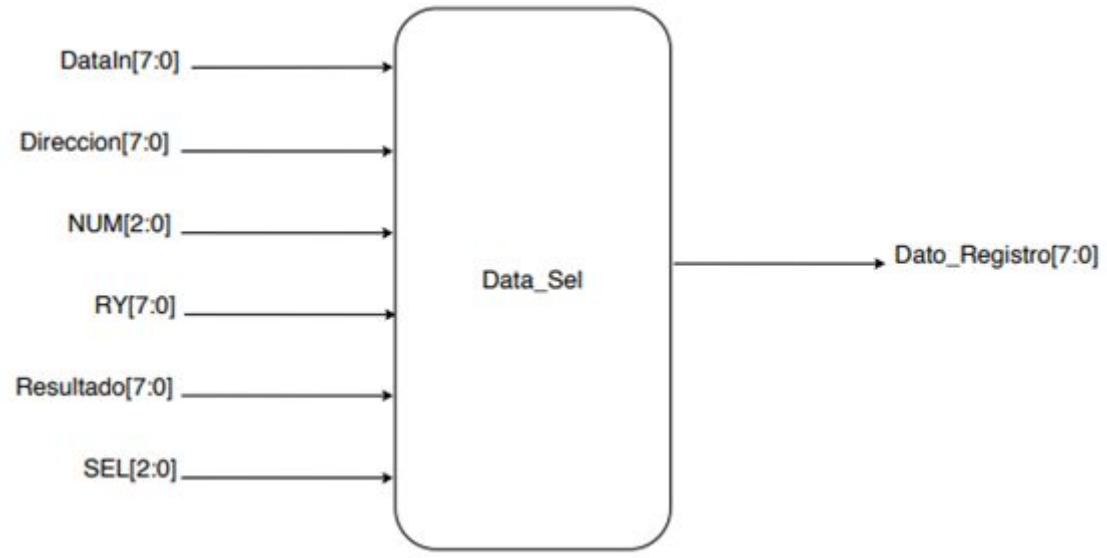


Figura 5.4. Caja negra Data _Sel

Tabla 5.10. Descripción general de entradas y salidas de Selector de datos

Nombre de la seña	Dirección	Tamaño (Bits)	Descripción
DataIn[7:0]	Entrada	8	Entrada de datos que proviene desde la memoria
Direccion[7:0]	Entrada	8	Es la dirección a la cual se va a realizar el salto condicional.
NUM[2:0]	Entrada	3	Es el dato que viene dentro de la instrucción el cual va a ser cargado al registro RX.
RY[7:0]	Entrada	8	Es un dato que se encuentra en el Banco de Registros para asignarlo a otro registro.
Resultado[7:0]	Entrada	8	Resultado de una operación lógica o aritmética entregada por la ALU.

SEL[2:0]	Entrada	3	El selector es el que conecta a una de sus entradas con una salida.
Dato_Registro[7:0]	Salida	8	Es un dato que se manda al Banco de Registros para ser almacenado.

Tabla 5.11. Funcionamiento del selector de datos

SEL	Dato_Registro
000	Dato_Registro=DataIn
001	Dato_Registro=Direccion
010	Dato_Registro=NUM
011	Dato_Registro=RY
100	Dato_Registro=Resultado

BIU

Unidad de interfaz del bus BUS, Es una línea de interconexión portada de información, construida por varios hilos conductores varios canales, por cada uno de los cuales se transporta un bit de información, Donde el número de líneas que forman los buses es fundamental, si un bus está compuesto por 16 líneas, podrá enviar a 16 bits al mismo tiempo, los buses interconectan la circuitería interna, como subsistemas del ordenador intercambian datos gracias a los buses, se clasifican según el criterio de su situación física.

Bus interno: Se encarga de mover datos entre los componentes internos del microprocesador. ·

Bus externo: Útil para comunicar el micro y otras partes, como periféricos y memorias, que son los componentes de un microprocesador.

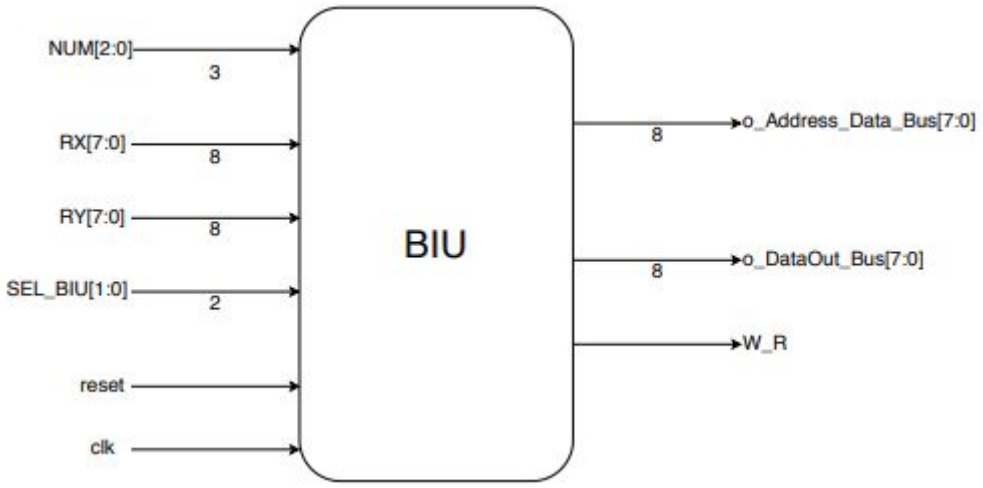


Figura 5.5. Caja negra BIU

Tabla 5.12. Descripción de entradas y salidas del BIU

Nombre de la señal	Dirección	Tamaño (Bits)	Descripción
NUM	Entrada	3	Señal de entrada, la cual corresponde a un dato de entrada proveniente de un modo de dirección inmediato
RX	Entrada	8	Señal de entrada correspondiente a uno de los datos decodificados, con los cuales se podrán manipular en la operación a realizar.
RY	Entrada	8	Señal de entrada correspondiente a uno de los datos decodificados, con los cuales se podrán manipular en la operación a realizar.
SEL_BIU	Entrada	2	Señal de entrada que elige la operación a realizar en el submódulo BIU

reset	Entrada	1	Señal de entrada que reinicia el submódulo BIU a un estado inicial
clk	Entrada	1	Esta señal se describe como señal de referencia temporal con una frecuencia de 100MHz
o_Address_Data_Bus	Salida	8	Señal de salida que seleccionará la dirección de memoria necesaria para que se pueda ejecutar una instrucción.
o_DataOut_Bus	Salida	8	Señal de salida necesaria para poder indicar una dirección de memoria.
W_R	Salida	1	Señal de salida encargada de indicar cuál salida será utilizada, salida de dirección de memoria BIU o la salida de datos de la BIU.

Tabla 5.12. Funcionamiento del BIU

INSTRUCCIÓN	SEL_BIU	W_R	o_DataOut_Bus	o_Address_Data_Bus
LOAD (Dato cargado de [RY])	00	0	0	RY
STORE (NUM será almacenado en [RX])	01	1	NUM	RX

STORE (Almacenará datos del registro RY en [RX])	10	1	RY	RX
NOP (NO OPERATION)	11	0	0	0

Memoria Ram

La **memoria de acceso aleatorio** (*Random Access Memory*, RAM) se utiliza como memoria de trabajo de computadoras y otros dispositivos para el sistema operativo, los programas y la mayor parte del software. En la RAM se cargan todas las instrucciones que ejecuta la unidad central de procesamiento (procesador) y otras unidades del computador, además de contener los datos que manipulan los distintos programas.

Se denominan «de acceso aleatorio» porque se puede leer o escribir en una posición de memoria con un tiempo de espera igual para cualquier posición, no siendo necesario seguir un orden para acceder (acceso secuencial) a la información de la manera más rápida posible.

Memoria Rom

Como su nombre lo indica, una memoria de sólo lectura (ROM) es una unidad de memoria que sólo ejecuta la operación de lectura; no tiene la posibilidad de escritura. Esto implica que la información binaria almacenada en una ROM se hace permanente durante la producción del hardware de la unidad y no puede alterarse escribiendo diferentes palabras en ella.

6.Implementación

Para el correcto funcionamiento del microprocesador se construyeron 5 componentes fundamentales, llamados Deco, ALU, Data_Sel, Banco_R, UCJ y BIU así como una memoria ROM dedicada a instrucciones del microprocesador y una memoria RAM dedicada a los datos, dichos componentes son una colección en conjunto para el funcionamiento del microprocesador teniendo como nombre top del proyecto “ArqHarvard” que componen la unión del microprocesador , y la respectiva memoria de datos.

- El decodificador (Deco) recibirá como entrada una instrucción de 9 bits que se encuentra dentro del set de instrucciones del microprocesador, la decodificará y dividirá las tareas que apuntarán a los diferentes módulos del microprocesador para completar la instrucción ingresada. Se codificó y probó con simulaciones en vivado [apéndice A: Decodificador].
- ALU unidad aritmética lógica, es un circuito que básicamente se encarga de las operaciones lógicas y aritméticas, que cuenta con una serie de registros para almacenar datos y bits, de información sobre los resultados o banderas, donde las entradas de la ALU contienen la longitud y el signo que corresponde a la operación también requiere de un mecanismo de control que le permite saber el tipo de operación que se está realizando, como se muestra en el diagrama de caja negra se tienen dos entradas de 8 bits (A,B) que cuentan con entrada de datos con valor indefinido, cuenta con otra entrada de datos llamada OP que tiene tres tipos de entrada 0,1,x donde cada una de estas tiene su características importantes, donde 1 indica encendido, 0 indica apagado y x indica no importa el estado en el que este, cuenta con dos salidas F de 3 bits, que se envía a la UCJ para mostrar la condición que se va a realizar para hacer el salto y R de 8 bits que muestra el resultado de la operación lógica o aritmética que se entrega como entrada a selector de datos. Se codificó y probó con simulaciones en vivado [apéndice B: Código ALU].
- El Data_Sel es el selector de datos y es el que se encarga de decir cual dato entra a banco de registros ya sea dirección, número, un registro o un resultado de la ALU. Se codificó y probó con simulaciones en vivado [apéndice D: Data_Sel].
- El Banco_R en base a los requerimientos cuenta con 8 registros los cuales son direccionados para acceder a ellos a través de las entradas de direccionamiento de registro. Las acciones que se pueden realizar en ellos son lectura o escritura lo cual se define con la entrada W_R que nos dice que cuando estamos en modo lectura las salidas son equivalentes a los registros seleccionados y cuando es escritura el dato de entrada se guarda en el registro seleccionado con AddrW. Se codificó y probó con simulaciones en vivado [apéndice C: Banco_R].
- UCJ se encarga de que las cosas ocurran, se encarga de emitir una señal de control externas a la CPU, para reducir el intercambio de datos con los módulos de entradas y salidas, se encarga de emitir una señal de control interna para transferir datos entre registros, hace que la ALU ejecute una función concreta y analiza, extrae las instrucciones de memoria central, para lo que necesita dos registros, donde F es una señal de entrada donde se encuentra el estado de las banderas, RX esta se usa para realizar las operaciones, cond decide cuál condición tomar para realizar el salto

y genera una señal de salida que es la dirección de la instrucción. Se codificó y probó con simulaciones en vivado [apéndice F: UCJ].

- La BIU es una línea de interconexión portada de información, construida por varios hilos conductores (varios canales), por cada uno de los cuales se transporta un bit de información. Sirve para dar paso a las instrucciones del programa, de datos para que estos puedan alcanzar los registros de la unidad de control y de la ALU. Se codificó y probó con simulaciones en vivado [apéndice E: BIU]. Es el encargado de controlar la dirección del bus de datos de salida, así como para importar o exportar datos a la memoria RAM. [3]
- Memoria ROM, diseñada para el ingreso de instrucciones requeridas por el microprocesador.
- Memoria RAM, diseñada para el uso de los datos requeridos por el microprocesador.

7. Pruebas

Una vez completado el sistema, se realizan las pruebas correspondientes para revisar que el funcionamiento de nuestro microprocesador sea el adecuado, esto se realiza a través de simulaciones, comenzando con cada uno de los submódulos del microprocesador para continuar con las memorias de datos y al final del sistema que se compone de la memoria de datos, la memoria de instrucciones y el microprocesador.

Para comenzar, tomamos en cuenta el submódulo principal, el Decodificador, el cual se encarga de comenzar con el proceso del microprocesador, ordenandole a cada submódulo qué hacer con las según la instrucción que se ingresa en él. En la figura 7.1 se puede observar el comportamiento de dicho submódulo cuando se le ingresa una a una las diferentes instrucciones disponibles, y se puede observar las señales de control que este produce por cada instrucción. En el Apéndice A “código de simulación” podemos encontrar el código del testbench que se utilizó para esta simulación.

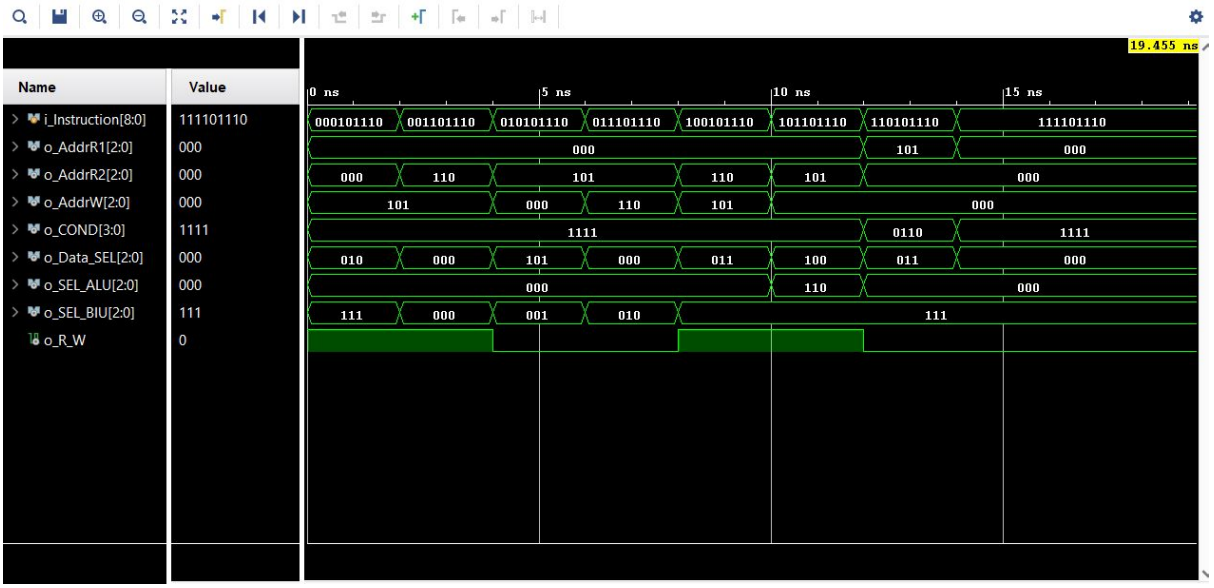


Figura 7.1 Simulación del submódulo Decodificador

En la figura 7.2 podemos observar el comportamiento dentro de nuestro submódulo ALU, aquí se asignan dos entradas u operandos, (A y B), con los cuales se trabajará, y con nuestra entrada OP, se le indicará qué operación se realizará, en este submódulo se asignan las banderas (F), que trabajarán con el UCJ. En este caso se realizó primero la suma donde los dos operandos son 31, luego la resta de estos mismos. En el Apéndice B “código de simulación” podemos encontrar el código del testbench que se utilizó para esta simulación.

Figura 7.2. Simulación del submódulo ALU.

La siguiente simulación (Figura 7.3) es la correspondiente al submódulo Banco de Registros, en el cual, se observa su comportamiento, Addr1 se encarga de seleccionar el registro que contiene los datos que saldrán por Rx, a través de Addr2 entra la direccion del registro que contiene los datos que saldrán por Ry, de registros desde las cuales, se va a leer el dato de entrada, AddrW se utiliza para apuntar a un registro donde se va a almacenar datos de 8 bits, se utiliza solo cuando al banco de registros recibe un uno por la entrada W_R. En el Apéndice C “código de simulación” podemos encontrar el código del testbench que se utilizó para esta simulación.

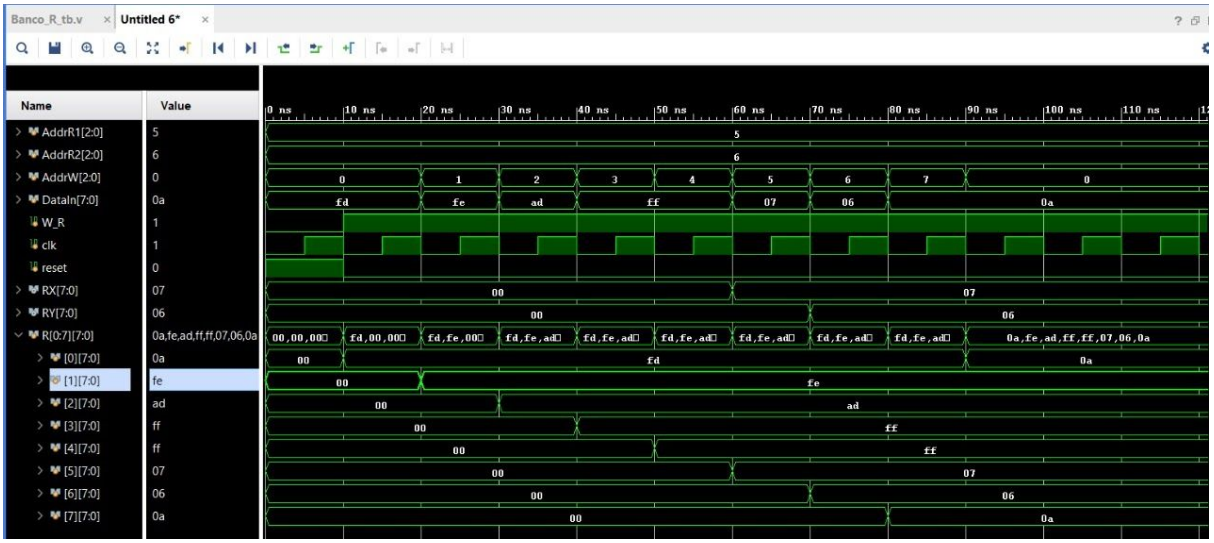


Figura 7.3. Simulación del submódulo Banco_Registros

En la figura 7.4 podemos observar el comportamiento de la simulación correspondiente al Data_Sel, que es el encargado de decir cuál dato entra al banco de registros, ya sea dirección, número, registro o un resultado de la misma ALU.

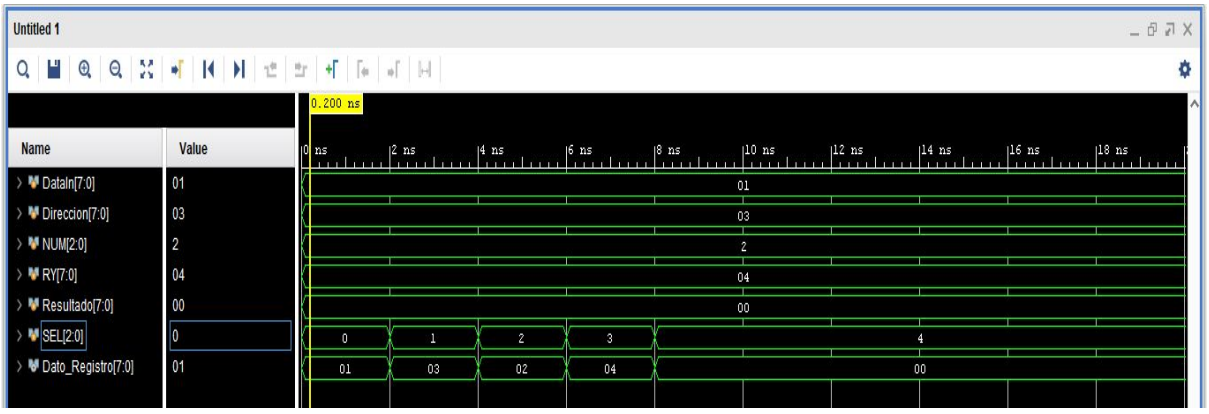


Figura 7.4. Simulación del submódulo Data_Sel

El submódulo BIU es el encargado de dar paso a las instrucciones del programa, de datos para que estos puedan alcanzar los registros de la unidad de control y de la ALU. En la figura 7.5 se puede observar su comportamiento cuando se le ingresan los datos RX:7 RY:6 y NUM:2, las salidas toman diferentes valores correspondientes a la instrucción que se ingresa. En el Apéndice D “código de simulación” podemos encontrar el código del testbench que se utilizó para esta simulación.

Figura 7.5. Simulación del submódulo BIU

En la figura 7.6, correspondiente a la simulación del UCJ, se muestra que la condición 8, indica que no hay ninguna condición, y su salida siempre es igual a ella misma más 1, cuando la condición es 0 y 1 salta a Rx lo asigna la salida, cuando la condición es 2 cuando la bandera es 1 la salida es igual a Rx, cuando la condición es 3 se verifica la bandera z=0 si no, directamente aumenta en 1, en la condición 4 pregunta si la n=1, brinca a Rx que Rx sigue valiendo f0, la condición 5 si la segunda bandera es 0, no es 0 por lo que aumenta en 1, si fuera 0 mandaría Rx, condición 6 si n=1 brinca a Rx=f0, condición 7 cuando n= 1 suma 1, y en esta condición solo va saltando(preguntando si es 0 y como no lo es va aumentando en 1) y va asignar f0 a la salida, la salida no se le asigna ninguna valor hasta que el código comienza a correr. En el Apéndice E “código de simulación” podemos encontrar el código del testbench que se utilizó para esta simulación.

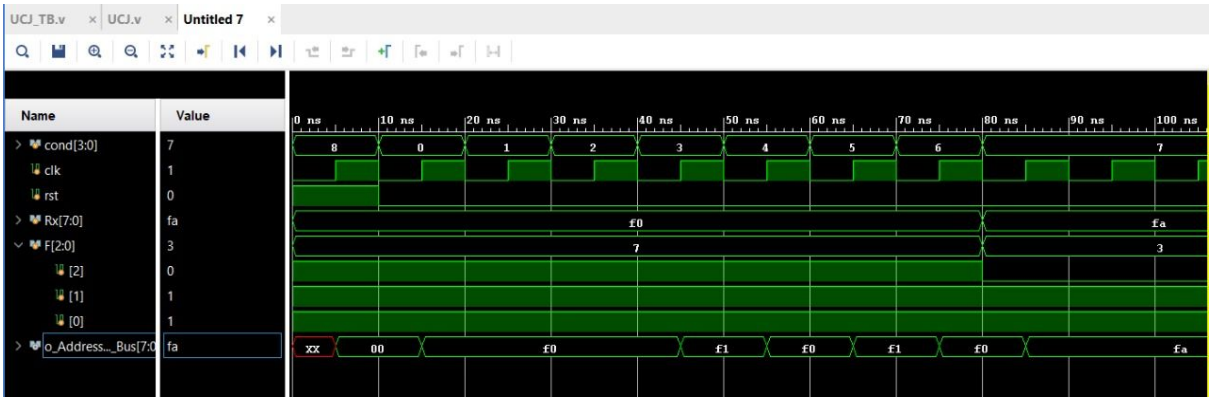


Figura 7.6. Simulación del submódulo UCJ

8.Análisis de resultados

Una vez terminado el sistema, teniendo funcionando el microprocesador, una memoria de datos y una memoria de instrucciones, se procedió al diseño de dos programas que con ayuda de las instrucciones disponibles en el procesador realizaron una multiplicación y una división de dos números, guardando el resultado en un registro de la memoria de datos.

Programa de multiplicación

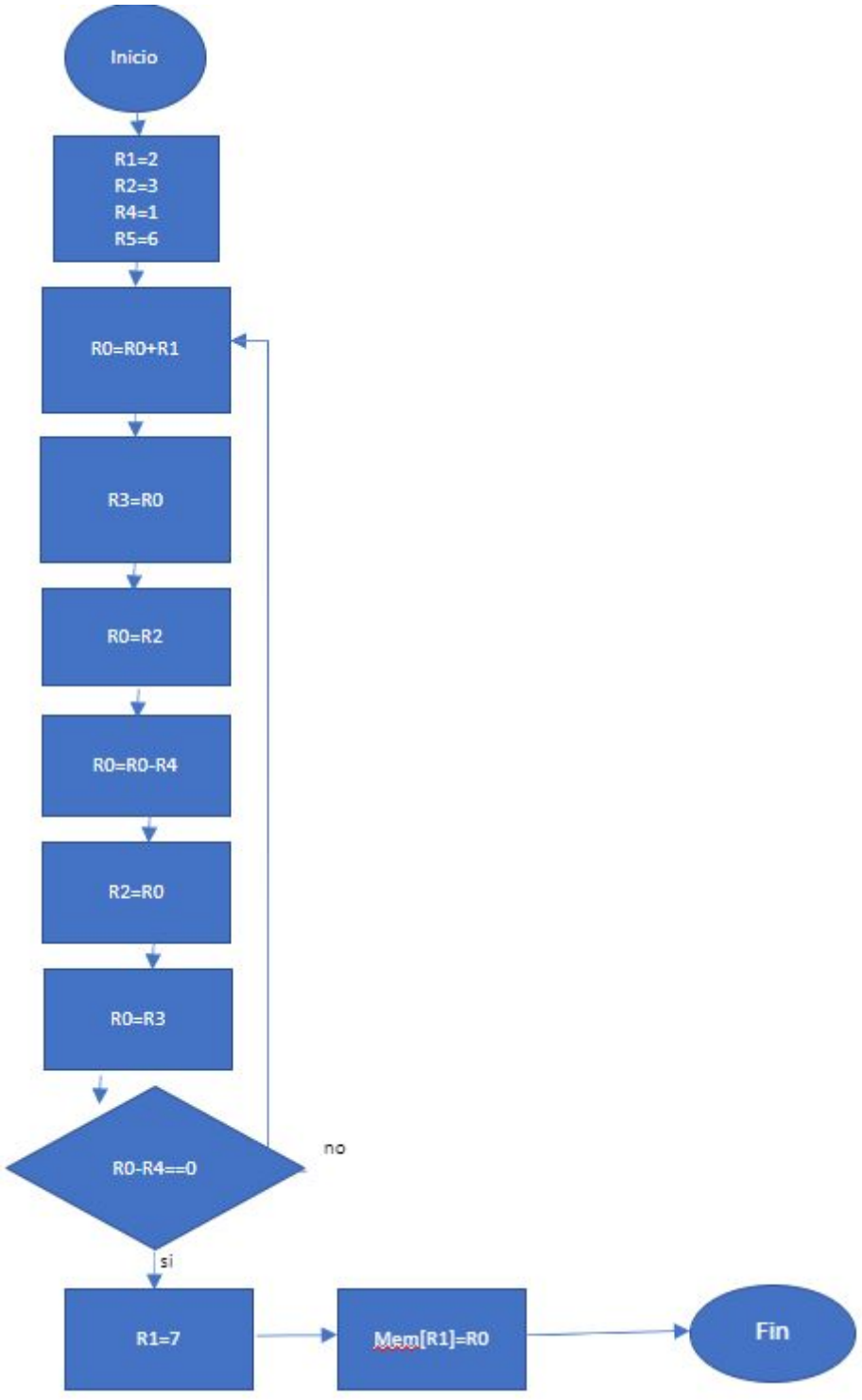


Figura 8 Diagrama de flujo del programa de multiplicación

Código del Programa de Multiplicación

```
1.- 111111111//nop
2.- 000001010;     //load r1, 2
3.- 000010011;     //load r2, 3
4.- 000100001;     //load r4, 1
5.- 000101110;     //load r5, 6
6 .-101001000;     //math r1 suma = r0=0+2 r0=2, r0=2+2 r0=4, r0=4+2 r0=6 loop
7.- 100011000;     // move r3, r0     r3=2  //hexa 118
8.- 100000010;     // move r0, r2  //hexa 102
9.- 101100001;  //math r4, resta r0=3-1 primer r3=2, segunda 2-1 1, tercera 1-1 0 //hexa 161
10.- 100010000;   // move r2, r0  //hexa 110
11.- 100000011;   //move r0,r3 //hexa 103
12.- 110110010;   //jump 110 010 //hexa 182
13.- 110101000;   //jump 101 000 //hexa 1A0
14.- 000001111;   //load r1,7
15.- 011110000;   //store r1, r0
```

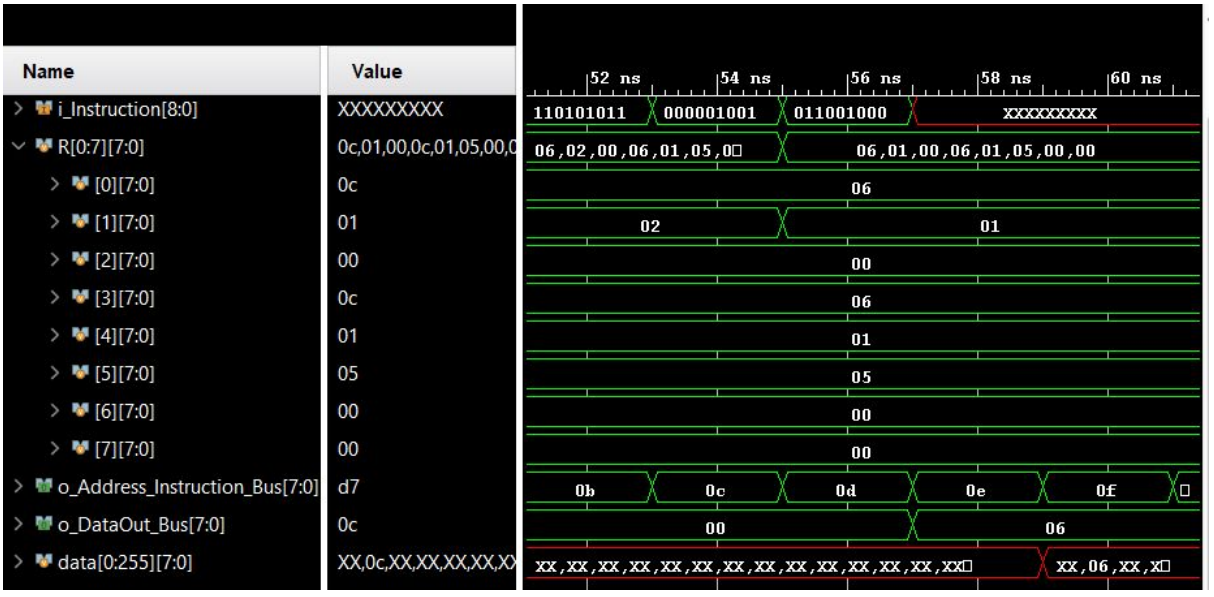


Figura 8.1: Simulación del programa de multiplicación, el cual realiza la operacion 2x3 y obtenemos como resultado en la dirección 1 de la memoria de datos un 6, el cual también se puede visualizar en el registro 0 y la salida de datos de la BIU.

Programa para la división

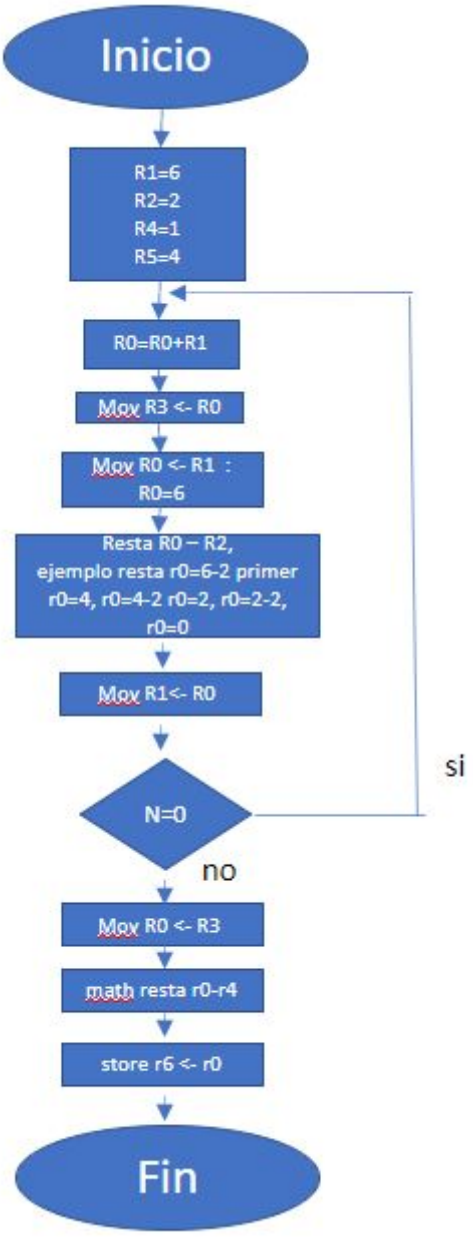


Figura 8.3 Diagrama de flujo del programa de division

Código del programa de division

1.- 000001110; //load r1, 6
2.- 000010010; //load r2, 2
3.- 000100001; //load r4, 1
4.- 000101100; //load r5, 4
5.- 101100000; //math r4,suma r0=0+1 r0=1,r0=1+1 r0=2, r0=2+1 r0=3
6.- 100011000; //mov r3,r0
7.- 100000001; //mov r0,r1
8.- 101010001; //math r2,resta r0=6-2 primer r0=4, r0=4-2 r0=2, r0=2-2, r0=0
9.- 100001000; //mov r1,r0
10.-110101111; //jump si el valor de r0 = 0
11.- 100000011; //mov r0,r3
12.- 101100001; //math resta r0-r4
13.- 011110000; //store r6,r0

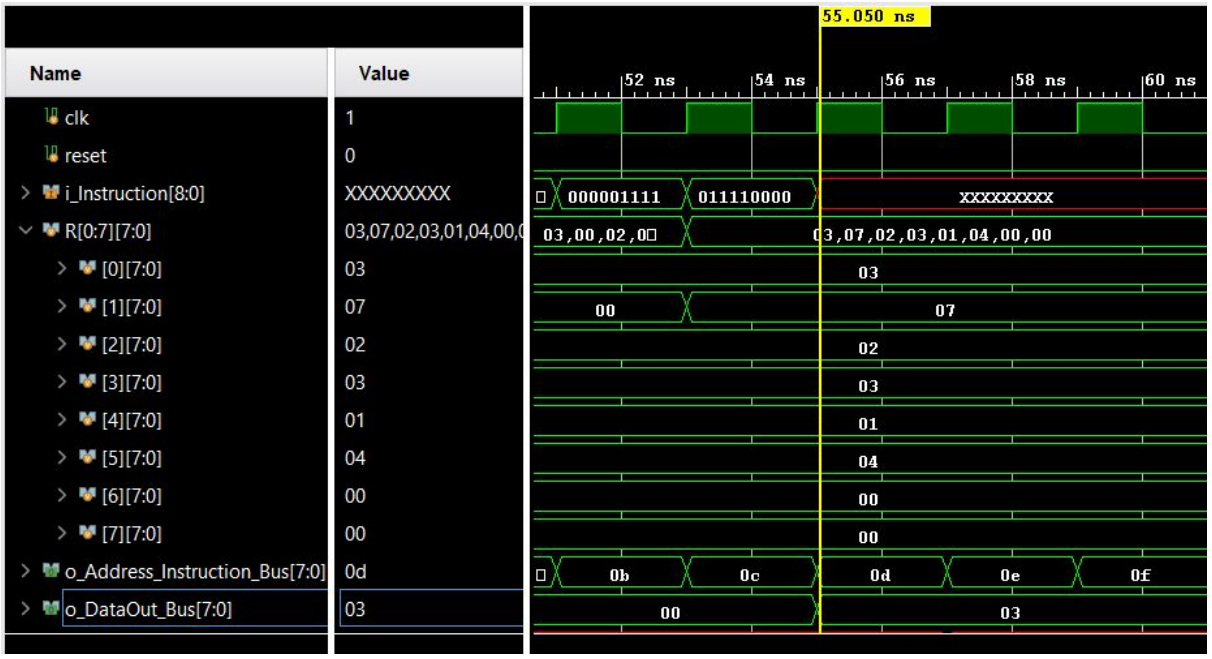


Figura 8.4: Simulación de la división 6/2 el resultado obtenido fue 3 y se visualiza en la salida de datos de la BIU así como en el registro R0.

9. Conclusiones

La implementación de este proyecto se llevó a cabo gracias a la implementación de Arquitectura Harvard, compuesta por pistas de almacenamiento y de señales físicamente separadas para las instrucciones y para los datos, por motivos de rendimiento, para que pueda soportar tareas como la carga de programa desde la unidad de disco como datos para su posterior ejecución, este se caracteriza por que se puede almacenar instrucciones en memoria de solo lectura, y una memoria de datos que requiere una memoria de lectura y escritura, que nos permite acceso simultáneo a más de una memoria del sistema, lo que nos llevo a el desarrollo e implementación de cada parte del microprocesador, utilizando una herramienta desarrollada por Xilinx es una familia de productos de FPGA´s entre ellas el software computacional Vivado Design Suite, que permite implementar la simulación de esta práctica, durante las pruebas se tuvieron diferentes errores, desde los sintácticos que se resolvieron en su momento ,cuando se obtenían errores de salida como cuando no se le asignaba un valor a registro hasta que el código comenzará a correr, cuya soluciones se plantearon y solucionaron en el momento, para lo que se planteó crear un registro y usarlo asignando a la salida, en la actualidad esta arquitectura se utiliza utilizando una memoria caché, donde los procesadores de señales digitales se utilizan principalmente en aplicaciones cuyas compensaciones, como los costos y el ahorro de energía de la omisión de memoria caché superan las desventajas de programación que vienen con tener espacios de direcciones de código y datos diferentes.

10. Referencias

[1] Lifelong Learning, “Ingeniería de los sistemas embebidos.", Ind. Syst. enginering, pp. 1–19, 2011, [Online]. Available: http://www.ieec.uned.es/investigacion/Dipseil/PAC/archivos/Informacion_de_referencia_ISE5_3_1.pdf.

[2] H. Zea, J. David, and J. Alfredo, “Design and implementation of a special-purpose microprocessor,” 2011.

[3].- Sandoval Aréchiga, R. (2020, 23 septiembre). Microprocesador RISC Arquitectura Harvard 8 Bits. GitHub. https://github.com/remberto-uaz/Rob7MicrosAgo2020/tree/master/MicroUAZ_Ago2020

[4].- Mortuux, (23 noviembre 2009), Arquitectura del microprocesador: Banco de Registros. URL:https://mortuux.wordpress.com/2009/11/23/arquitectura-del-microprocesador-el-banco-de-registros/

[5].- Mortuux,(18 noviembre 2009), Arquitectura del microprocesador: Unidad Aritmética Logica. URL: https://mortuux.wordpress.com/2009/11/18/arquitectura-del-microprocesador-unidad-aritmetico-logica/

11. Apéndices

Apéndice A

Código. Deco

```
1.  module Deco(
2.  input [8:0] i_Instruction,
3.  output reg [2:0] o_AddrR1,
4.  output reg [2:0] o_AddrR2,
5.  output reg [2:0] o_AddrW,
6.  output reg [3:0] o_COND,
7.  output reg [2:0] o_Data_SEL,
8.  output reg [2:0] o_SEL_ALU,
9.  output reg [2:0] o_SEL_BIU,
10. output reg o_R_W
11. );
12.
13. always@(i_Instruction)
14. begin
15.         case(i_Instruction[8:6])
16.             0:                                          // LOAD RX, #NUM
17.                 begin
18.                     o_AddrR1 <= 0;
19.                     o_AddrR2 <= 0;
20.                     o_AddrW <= i_Instruction[5:3];
21.                     o_COND <= 4'hF;
22.                     o_Data_SEL <= 3'd4;
23.                     o_SEL_ALU <= 0;
24.                     o_SEL_BIU <= 3'd7;
25.                     o_R_W <= 1;
26.                 end
27.             1:                                          // LOAD RX, [RY]
28.                 begin
29.                     o_AddrR1 <= 0;
30.                     o_AddrR2 <= i_Instruction[2:0];
31.                     o_AddrW <= i_Instruction[5:3];
32.                     o_COND <= 4'hF;
33.                     o_Data_SEL <= 3'd0;
34.                     o_SEL_ALU <= 0;
35.                     o_SEL_BIU <= 3'd0;
36.                     o_R_W <= 1;
37.                 end
38.             2:                                          // STORE #NUM
39.                 begin
40.                     o_AddrR1 <= 0;
```

```
41.                         o_AddrR2 <= i_Instruction[5:3];
42.                         o_AddrW <=0;
43.                         o_COND <= 4'hF;
44.                         o_Data_SEL <= 3'd5;
45.                         o_SEL_ALU <= 0;
46.                         o_SEL_BIU <= 3'd1;
47.                         o_R_W <= 0;
48.                 end
49.         3:                                          // STORE [RX], RY
50.                 begin
51.                         o_AddrR1 <= i_Instruction[5:3];
52.                         o_AddrR2 <= i_Instruction[2:0];
53.                         o_AddrW <= 0;
54.                         o_COND <= 4'hF;
55.                         o_Data_SEL <= 3'd0;
56.                         o_SEL_ALU <= 0;
57.                         o_SEL_BIU <= 3'd0;
58.                         o_R_W <= 0;
59.                 end
60.         4:                                          // MOVE  RX, RY
61.                 begin
62.                         o_AddrR1 <= 0;
63.                         o_AddrR2 <= i_Instruction[2:0];
64.                         o_AddrW <= i_Instruction[5:3];
65.                         o_COND <= 4'hF;
66.                         o_Data_SEL <= 3'd3;
67.                         o_SEL_ALU <= 0;
68.                         o_SEL_BIU <= 3'd7;
69.                         o_R_W <= 1;
70.                 end
71.         5:                                          // MATH  RX, OP
72.                 begin
73.                         o_AddrR1 <= 0;
74.                         o_AddrR2 <= i_Instruction[2:0];
75.                         o_AddrW <= 0;
76.                         o_COND <= 4'hF;
77.                         o_Data_SEL <= 3'd1;
78.                         o_SEL_ALU <= i_Instruction[5:3];
79.                         o_SEL_BIU <= 3'd7;
80.                         o_R_W <= 1;
81.                 end
82.         6:                                          // JMP  RX, COND
83.                 begin
84.                         o_AddrR1 <= i_Instruction[5:3];
85.                         o_AddrR2 <= 0;
86.                         o_SEL_ALU <= 0;
87.                         o_COND = {1'b0,i_Instruction[2:0]};
88.                         o_SEL_BIU <= 3'd7;
```

```
89.                                     if (i_Instruction[2:0]==3'b001)
90.                                             begin
91.                                                     o_Data_SEL <= 3'd2;
92.                                                     o_AddrW <= 7;
93.                                                     o_R_W <= 1;
94.                                             end
95.                                     else
96.                                             begin
97.                                                     o_Data_SEL <= 3'd3;
98.                                                     o_AddrW <= 0;
99.                                                     o_R_W <= 0;
100.                                            end
101.                            end
102.                    7:                                              // NOP
103.                            begin
104.                                    o_AddrR1 <= 0;
105.                                    o_AddrR2 <= 0;
106.                                    o_AddrW <= 0;
107.                                    o_COND <= 4'hF;
108.                                    o_Data_SEL <= 3'd5;
109.                                    o_SEL_ALU <= 0;
110.                                    o_SEL_BIU <= 3'd7;
111.                                    o_R_W <= 0;
112.                            end
113.                    endcase
114.            end
115. endmodule
```

Código de simulación: Deco_TB

```
1.  module Deco_TB;
2.      reg [8:0] i_Instruction;
3.      wire [2:0] o_AddrR1;
4.      wire [2:0] o_AddrR2;
5.      wire [2:0] o_AddrW;
6       wire [3:0] o_COND;
7.      wire [2:0] o_Data_SEL;
8.      wire [2:0] o_SEL_ALU;
9.      wire [2:0] o_SEL_BIU;
10.     wire o_R_W;
11.
12.      Deco uut(
13.             .i_Instruction(i_Instruction),
```

```
14.             .o_AddrR1(o_AddrR1),
15.             .o_AddrR2(o_AddrR2),
16.             .o_AddrW(o_AddrW),
17.             .o_COND(o_COND),
18.             .o_Data_SEL(o_Data_SEL),
19.             .o_SEL_ALU(o_SEL_ALU),
20.             .o_SEL_BIU(o_SEL_BIU),
21.             .o_R_W(o_R_W)
22.       );
23.
24.  initial
25.    begin
26.             #2
27.             i_Instruction= 9'b001101000;          // LOAD RX, #NUM
28.             #2
29.             i_Instruction = 9'b001101010;                 // LOAD RX, [RY]
30.             #2
32.             i_Instruction= 9'b010101011;          // STORE #NUM
33.             #2
34.    i_Instruction= 9'b011101100;           // STORE [RX], RY
35.    #2
36.             i_Instruction= 9'b100101101;          // MOVE  RX, RY
37.    #2
38.             i_Instruction= 9'b101100110;          // MATH  RX, OP
39.             #2
40.             i_Instruction= 9'b110101111;          // JMP  RX, COND
41.             #2
42.             i_Instruction= 9'b111101111;          // NOP
43.      end
44. endmodule
```

Apendice B

Codigo ALU

```
1.module ALU(
2.      input [7:0] A,
3.      input [7:0] B,
4.      input [2:0] OP,
5.      output [2:0] F,
6.      output [7:0] R
7.  );
8.      reg [8:0] Rx;
```

```
9.      always@(*)
10.     begin
11.             case(OP)
12.0:           Rx <= A+B;
13.1:           Rx <= A-B;
14.2:           Rx <= A<<B;
15.3:           Rx <= A>>B;
16. 4:          Rx <= A&B;
17.5:           Rx <= A|B;
18.6:           Rx <= A^B;
19.7:           Rx <= ~A;
20.     endcase
21.end
22.     assign R = Rx[7:0];
23.     assign F[0] = &(Rx);
24.     assign F[1] = Rx[8];
25.     assign F[2] = Rx[7];
26.endmodule
```

Código de simulación: ALU_TB

```
1.module ALU_TB();
2.reg [7:0] A;
3. reg [7:0] B;
4.reg [2:0] OP;
5. wire [7:0] R;
6.wire [2:0] F;
7.ALU uut(
8..A(A),
9..B(B),
10.OP(OP),
11..R(R),
12..F(F)
13.);
14.initial
15. begin
16.     A<="00001111";
      B<="00001111";
      OP<="000";
      #2
      OP<="001";
      end

endmodule
```

Apéndice C

Codigo: Banco_R

```
1.- module Banco_R(
2.-     input [2:0] AddrR1,
3.-     input [2:0] AddrR2,
4.-     input [2:0] AddrW,
5.-     input [7:0] DataIn,
6.-     input W_R,
7.-     input clk,
8.-     input reset,
9.-     output [7:0] RX,
10.-    output [7:0] RY
11.-    );
12.-    reg [7:0] R[0:7];
13.-
14.-    always@(*)
15.-    begin
16.-            if(reset)
17.-                    begin
18.-                            R[0] <= 0;
19.-                            R[1] <= 0;
20.-                            R[2] <= 0;
21.-                            R[3] <= 0;
22.-                            R[4] <= 0;
23.-                            R[5] <= 0;
24.-                            R[6] <= 0;
25-                             R[7] <= 0;
26.-                    end
27.-            else if(W_R)
28.-                    R[AddrW] <= DataIn;
29.-    end
30.-    assign RX = R[AddrR1]; //escribe el dato de la direccion AddrR1 en RX
31.-    assign RY = R[AddrR2]; //escribe el dato de la direccion AddrR1 en RY
32.-endmodule
```

Codigo de simulacion: Banco_R_tb

```
1.- module Banco_R_tb;
2.-     reg [2:0]AddrR1;
3-      reg [2:0]AddrR2;
4.-     reg [2:0]AddrW;
5.-     reg [7:0]DataIn;
6.-     reg W_R;
7.-     reg clk;
8.-     reg reset;
```

```
9.-     wire [7:0] RX;
10.-    wire [7:0] RY;
11.-
12.-    Banco_R uut(
13.-            .AddrR1(AddrR1),
14.-            .AddrR2(AddrR2),
15.-            .AddrW(AddrW),
16.-            .DataIn(DataIn),
17.-            .W_R(W_R),
18.-            .clk(clk),
19.-            .reset(reset),
20.-            .RX(RX),
21.-            .RY(RY)
22.-    );
23.-
24.-    initial
25.-            begin
26.-                    reset <= 1;
27.-                    clk<=0;
28.-                    AddrR1 <=5;
29.-                    AddrR2 <=6;
30.-                    AddrW <=0;
31.-                    DataIn <=8'hFD;
32.-                    W_R <= 0;
33.-                    #10
34.-                    reset <= 0;
35.-                    W_R <= 1;
36.-                    #10 AddrW <= AddrW+1;
37.-                    DataIn <=8'hFE;
38.-                    #10 AddrW <= AddrW+1;
39.-                    DataIn <=8'hAD;
40.-                    #10 AddrW <= AddrW+1;
41.-                    DataIn <=8'hFF;
42.-                    #10 AddrW <= AddrW+1;
43.-                    #10 AddrW <= AddrW+1;
44.-                    DataIn <=8'h7;
45.-                    #10 AddrW <= AddrW+1;
46.-                    DataIn <=8'h6;
47.-                    #10 AddrW <= AddrW+1;
48.-                    DataIn <=8'h0A;
49.-                    #10 AddrW <= AddrW+1;
50.-
51.-            end
52.-    always@(clk)
53.-      #5 clk <= ~clk;
54.- endmodule
```

Apéndice D

Código: Data_Sel

```
1.  module Data_Sel(
2.      input [7:0] DataIn,
3.      input [7:0] Direccion,
4.      input [2:0] NUM,
5.      input [7:0] RY,
6.      input [7:0] Resultado,
7.      input [2:0] SEL,
8.      output reg [7:0] Dato_Registro
9.      );
10.
11.  always@(*)
12.      case(SEL)
13.      0: Dato_Registro <= DataIn;
14.      1: Dato_Registro <= Direccion;
15.      2: Dato_Registro <= NUM;
16.      3: Dato_Registro <= RY;
17.      4: Dato_Registro <= Resultado;
18.       default Dato_Registro <= 0;
19.       endcase
20.  endmodule
```

Código de simulacion: Data_Sel_tb

```
1.   module Data_Sel_tb;
2.
3.       reg [7:0] DataIn;
4.       reg [7:0] Direccion;
5.       reg [2:0] NUM;
6.       reg [7:0] RY;
7.       reg [7:0] Resultado;
8.       reg [2:0] SEL;
9.       wire [7:0] Dato_Registro
10.      ;
11.      .Data_Sel uut(
12.      .DataIn(DataIn),
13.      .Direccion(Direccion),
14.      .NUM(NUM),
15.      .RY(RY),
16.      .Resultado(Resultado),
```

```
17.       .SEL(SEL),
18.       .Dato_Registro(Dato_Registro)
19.        );
20.
21.        initial
22.        begin
23.        DataIn = 8'd1;
24.        Direccion = 8'd3;
25.       NUM = 3'd2;
26.       RY = 8'd4;
27.       Resultado = 8'd0;
28.       SEL = 3'd0;
29.
30.  #2
31.       SEL = 3'd1;
32.  #2
33.       SEL = 3'd2;
34.  #2
35.       SEL = 3'd3;
36.  #2
37.       SEL = 3'd4;
38.
39.       end
40.  endmodule
```

Apéndice E

Código: BIU

```
1.        module BIU(
2.
3.           input [2:0] NUM,
4.           input [7:0] RX,
5.           input [7:0] RY,
6.           input [1:0] SEL_BIU,
7.           input reset,
8.           input clk,
9.           output reg [7:0]o_Address_Data_Bus,
10.          output reg [7:0]o_DataOut_Bus,
11.          output reg W_R
12.         );
13.
14.          always@(posedge clk)
15.
16.          begin
```

```
17.
18.            if(reset)
19.
20.             begin
21.                o_DataOut_Bus <= 0;
22.                o_Address_Data_Bus <= 0;
23.                W_R <= 0;
24.
25.             end
26.             else
27.                case(SEL_BIU)
28.                   0: begin    //Para la instrucción LOAD (Dato cargado de [RY])
29.                      o_DataOut_Bus <= 0;
30.                      o_Address_Data_Bus <= RY;
31.                      W_R <= 0;
32.
33.                   end
34.
35.                   1: begin    //Para la instrucción STORE (NUM será almacenado en [RX])
36.                     o_DataOut_Bus <= {5'b00000,NUM};
37.                      o_Address_Data_Bus <= RX;
38.                      W_R <= 1;
39.
40.                   end
41.
42.                   2: begin    // Para la instrucción STORE (Almacenará datos del registro RY
en [RX])
43.                      o_DataOut_Bus <= RY;
44.                      o_Address_Data_Bus <= RX;
45.                      W_R <= 1;
46.
47.                   end
48.
49.                   3: begin    //NOP (NO OPERATION)
50.                      o_DataOut_Bus <= 0;
51.                      o_Address_Data_Bus <= 0;
52.                      W_R <= 0;
53.
54.                   end
55.
56.                endcase
57.          end
58.       endmodule
```

Código de simulación tb_BIU

```
1.      module tb_BIU;
2.        reg [2:0] NUM;
3.        reg [7:0] RX;
4.        reg [7:0] RY;
5.        reg [1:0] SEL_BIU;
6.        reg reset;
7.        reg clk;
8.
9.        wire [7:0] o_Address_Data_Bus;
10.       wire [7:0] o_DataOut_Bus;
11.       wire W_R;
12.
13.       BIU uut(
14.       .NUM(NUM),
15.       .RX(RX),
16.       .RY(RY),
17.       .SEL_BIU(SEL_BIU),
18.       .reset(reset),
19.       .clk(clk),
20.
21.       .o_Address_Data_Bus(o_Address_Data_Bus),
22.       .o_DataOut_Bus(o_DataOut_Bus),
23.       .W_R(W_R)
24.       );
25.
26.       initial
27.          begin
28.          reset=1;
29.
30.          clk=0;
31.          RX=0;
32.          RY=0;
33.          NUM=0;
34.          SEL_BIU=3;
35.
36.          #2 reset=0; RX=8'd7; RY=8'd6; NUM=3'd2; SEL_BIU=3;
37.          #2 SEL_BIU=2'b00;
38.          #2 SEL_BIU=2'b01;
39.          #2 SEL_BIU=2'b10;
40.          end
41.
42.       always
43.          #1 clk = !clk;
44.
45.
```

```
46.     endmodule
```

Apéndice F

Código UCJ

```
1.      module UCJ(
2.          input [2:0]cond,
3.          input clk,
4.          input rst,
5.          input [7:0]Rx,
6.          input [2:0]F,
7.       output [7:0]o_Address_Instruction_Bus
8.  );
9.  always@(posedge clk)
10.      begin
11.       if(rst)
12.     o_Address_Instruction_Bus <=0;
13.             else
14.             case(cond)
15.                     0: Address_Instruction_Bus <= Rx;
16.                     1: Address_Instruction_Bus <= Rx;
17.
18.             2:
19.             begin
20.             if(F(0))
21.                     Address_Instruction_Bus <= Rx;
22.             else
23.                     Address_Instruction_Bus = Address_Instruction_Bus+1;
24.
25.             end
26.             3:
27.             begin
28.             if(~F(0))
29.                      Address_Instruction_Bus <= Rx;
30.             else
31.             Address_Instruction_Bus = Address_Instruction_Bus+1;
32.
33.     end
34.     4:
35.     begin
36.             if(F(1))
37.              Address_Instruction_Bus <= Rx;
38.             else
39.     Address_Instruction_Bus = Address_Instruction_Bus+1;
```

```
40.
41.     end
42.     5:
43.     begin
44.             if(~F(1))
45.             Address_Instruction_Bus <= Rx;
46.             else
47.     Address_Instruction_Bus = Address_Instruction_Bus+1;
48.
49.     end
50.     6:
51.     begin
52.             if(F(2))
53.             Address_Instruction_Bus <= Rx;
54.             else
55.     Address_Instruction_Bus = Address_Instruction_Bus+1;
56.
57.      end
58.     7:
59.     begin
60.             if(~F(2))
61.             Address_Instruction_Bus <= Rx;
62.             else
63.      Address_Instruction_Bus = Address_Instruction_Bus+1;
64.
65.     end
66.     default: Address_Instruction_Bus<=Address_Instruction_Bus+1;
67.
68.             endcase
69.     end
70. endmodule
```

Código de simulación: UCJ_TB

```
1.      module UCJ_TB(
2.         );
3.       reg[3:0]cond;
4.       reg clk,rst;
5.       reg [7:0]Rx;
6.       reg [2:0]F;
7.       wire[7:0] o_Address_Instruction_Bus;
8.       UCJ uut(
9.      .cond(cond),
10.     .clk(clk),
11.     .rst(rst),
```

```
12.     .Rx(Rx),
13.     .F(F),
14.     .o_Address_Instruction_Bus(o_Address_Instruction_Bus)
15.     );
16.     initial
17.     begin
18.     clk<=0;
19.     rst<=1;
20.     F<=3'b111;
21.     Rx<=8'hF0;
22.     cond<=4'b1000;
23.     #10 rst<=0;
24.     cond<=4'b0000;        //condición que indica que salte sin fijar nada
25.     #10 cond<=4'b0001;  //condición indica salte y guarde el dato
26.     #10 cond<=4'b0010;  //condición indica saltar si es 0
27.     #10 cond<=4'b0011;  //condición indica saltar si no es 0
28.     #10 cond<=4'b0100;  //condición que indica si hay carry
29.     #10 cond<=4'b0101;  //condición que indica si no hay carry
30.     #10 cond<=4'b0110;  //condición que indica si es negativo
31.     #10 cond<=4'b0111;  //condición que indica no es negativo
32.     F<=3'b011;//(donde los bits se leen de derecha izquierda)condición para que
brinque nuevamente a Rx(ncz)
33.     Rx<=8'hFA;// para que brinque a esta dirección
33.     end
34.     always@(clk)
35.     #5 clk<=~clk;
36.     endmodule
```

Apéndice G

Codigos microcontrolador, codigo de memoria de instrucciones, memoria de datos y arquitectura completa.

Codigo Microcontrolador

```
1.  module MicroBitos(
2.      input [7:0] i_DataIn_Bus,
3.      input clk,
4.      input reset,
5.      input [8:0] i_Instruccion,
6.      output W_R,
7.      output [7:0] o_DataOut_Bus,
8.      output [7:0] o_Address_Instruction_Bus,
9.      output [7:0] o_Address_Data_Bus
10.     );
11.
```

```
12.  wire [7:0] RX,RY, DataIn_Registro, Resultado_ALU;
13.  wire [2:0] AddrR1,AddrR2,AddrW,SEL_ALU,F,Data_SELECTOR;
14.  wire Write_Read_Register;
15.  wire [3:0] COND;
16.  wire [1:0] SEL_BIU;
17.
18.  Banco_R B1(
19.      .AddrR1(AddrR1),
20.      .AddrR2(AddrR2),
       a.      .AddrW(AddrW),
21.      .DataIn(DataIn_Registro),
22.      .clk(clk),
23.      .reset(reset),
24.      .W_R(Write_Read_Register),
25.      .RX(RX),
26.      .RY(RY)
27.  );
28.
29.  ALU B2(
30.      .A(RX),
31.      .B(RY),
32.      .OP(SEL_ALU),
33.      .R(Resultado_ALU),
34.      .F(F)
35.  );
36.
37.  UCJ B3(
38.      .cond(COND),
39.      .clk(clk),
40.      .reset(reset),
41.      .RX(RX),
42.      .F(F),
43.      .Address_Instruction(o_Address_Instruction_Bus)
44.  );
45.
46.  BIU B4(
47.      .reset(reset),
48.      .clk(clk),
49.      .SEL_BIU(SEL_BIU),
50.      .NUM(i_Instruccion[5:0]),
51.      .RX(RX),
52.      .RY(RY),
53.      .Address_Data(o_Address_Data_Bus),
54.      .DataOut(o_DataOut_Bus),
55.      .W_R(W_R)
56.  );
57.
58.  DATA_SEL B5(
```

```
59.         .DataIn(i_DataIn_Bus),
60.         .Resultado(Resultado_ALU),
61.         .Direccion(o_Address_Instruction_Bus),
62.         .RY(RY),
63.         .SEL(Data_SELECTOR),
64.         .NUM({2'b00,i_Instruccion[5:0]}),
65.         .Dato_Registro(DataIn_Registro)
66.     );
67.
68.     Deco B6(
69.         .Instruction(i_Instruccion),
70.         .AddrR1(AddrR1),
71.         .AddrR2(AddrR2),
72.         .AddrW(AddrW),
73.         .W_R(Write_Read_Register),
74.         .SEL_ALU(SEL_ALU),
75.         .COND(COND),
76.         .SEL_BIU(SEL_BIU),
77.         .Data_SEL(Data_SELECTOR)
78.     );
79. endmodule
```

Codigo de simulacion MicriBitos TB

```
1.  module MicroBitos_TB;
2.      reg [7:0] i_DataIn_Bus;
3.      reg clk;
4.      reg reset;
5.      reg [8:0] i_Instruccion;
6.      wire W_R;
7.      wire [7:0] o_DataOut_Bus;
8.      wire [7:0] o_Address_Instruction_Bus;
9.      wire [7:0] o_Address_Data_Bus;
10.
11. MicroBitos uut(
12.      .i_DataIn_Bus(i_DataIn_Bus),
13.      .clk(clk),
14.      .reset(reset),
15.      .i_Instruccion(i_Instruccion),
16.      .W_R(W_R),
17.      .o_DataOut_Bus(o_DataOut_Bus),
18.      .o_Address_Instruction_Bus(o_Address_Instruction_Bus),
19.      .o_Address_Data_Bus(o_Address_Data_Bus)
20. );
21.
22.     initial
23.       begin
```

```
24.       reset=1;
25.       clk=0;
26.       i_DataIn_Bus=4;
27.       i_Instruccion=9'b111111111;
28.       #2
29.       reset=0;
30.       #2
31.       i_Instruccion= 9'b001101000;              // LOAD RX, #NUM
32.       #2
33.       i_Instruccion= 9'b001101010;    // LOAD RX, [RY]
34.       #2
35.       i_Instruccion= 9'b010101011;              // STORE #NUM
36.       #2
37.       i_Instruccion= 9'b011101100;               // STORE [RX], RY
38.       #2
39.       i_Instruccion= 9'b100101101;              // MOVE  RX, RY
40.       #2
41.       i_Instruccion= 9'b101100110;              // MATH  RX, OP
42.       #2
43.       i_Instruccion= 9'b110101111;              // JMP  RX, COND
44.       #2
45.       i_Instruccion= 9'b111111111;              // NOP
46.       end
47.
48.    always
49.       #1 clk = !clk;
50.
51.
52. endmodule
```

Codigo RAM

```
1.   module RAM(
2.       input clk,
3.       input [7:0] i_Address,
4.       input [7:0] i_DataIn,
5.       input i_WR,
6.       output reg [7:0] o_DataOut
7.       );
8.
9.       reg [7:0] data [0:255];
10.      always@(posedge clk)
11.      if(i_WR)
12.          data[i_Address] <= i_DataIn;
13.      else
14.          o_DataOut <= data[i_Address];
15.
```

```
16. endmodule
```

Codigo ROM

```
1.  module ROM(
2.      input [7:0] i_Address,
3.      output [8:0]o_Instruction
4.      );
5.
6.      reg [8:0] inst [0:255];
7.      initial begin
8.  //          inst[0] <= 9'b111111111; //nop
9.  //          inst[1] <= 9'b000001010; //load r1, 2
10. //          inst[2] <= 9'b000010011; //load r2, 3
11. //          inst[3] <= 9'b000100001; //load r4, 1
12. //          inst[4] <= 9'b000101101; //load r5, 5
13. //          inst[5] <= 9'b101001000;  //math r1 suma = r0=0+2 r0=2, r0=2+2 r0=4,
    r0=4+2 r0=6 loop
14. //          inst[6] <= 9'b100011000;   // move r3, r0
15. //          inst[7] <= 9'b100000010; // move r0, r2
16. //          inst[8] <= 9'b101100001; //math r4, resta r0=3-1 primer r3=2, segunda 2-1
    1, tercera 1-1 0
17. //          inst[9] <= 9'b100010000; // move r2, r0
18. //          inst[10] <= 9'b100000011;   //move r0,r3
19. //          inst[11] <= 9'b110101011;  //jump 101 011
20. //          inst[12] <= 9'b000001111; //load r1,7
21. //          inst[13] <= 9'b011001000; //store r1, r0
22.
23.     inst[0] <= 9'b000001110;  //load r1, 3
24.     inst[1] <= 9'b000010010;  //load r2, 2
25.     inst[2] <= 9'b000100001; //load r4, 1
26.     inst[3] <= 9'b000101100;  //load r5, 4
27.     inst[4] <= 9'b101100000;  //math r4,suma r0=0+1 r0=1,r0=1+1 r0=2, r0=2+1 r0=3
28.     inst[5] <= 9'b100011000;  //mov r3,r0
29.     inst[6] <= 9'b100000001;  //mov r0,r1
30.     inst[7] <= 9'b101010001;  //math r2,resta r0=6-2 primer r0=4, r0=4-2 r0=2, r0=2-2,
    r0=0
31.     inst[8] <= 9'b100001000;  //mov r1,r0
32.     inst[9] <= 9'b100000011;  //move r0,r3
33.     inst[10] <= 9'b110101111;  //jump si el valor de r0 = 0 N=0
34.     inst[11] <= 9'b000001111; //load r1,7
35.     inst[12] <= 9'b011110000;  //store r6,r0
36.
37.
38. end
```

Codigo ArqHarvard

```
1.  module ArqHarvard(
2.      input clk,
3.      input reset
4.      );
5.
6.
7.      wire [8:0] instructions;
8.      wire [7:0] address_instructions, address_data, dataMem, dataMicro;
9.      wire wr;
10.     RAM P1(
11.     .clk(clk),
12.     .i_Address(address_data),
13.     .i_DataIn(dataMem),
14.     .i_WR(wr),
15.     .o_DataOut(dataMicro)
16.     );
17.
18.     ROM P2(
19.     .i_Address(address_instructions),
20.     .o_Instruction(instructions)
21.     );
22.
23.     MicroBitos P3(
24.     .i_DataIn_Bus(dataMicro),
25.     .clk(clk),
26.     .reset(reset),
27.     .i_Instruccion(instructions),
28.     .W_R(wr),
29.     .o_DataOut_Bus(dataMem),
30.     .o_Address_Instruction_Bus(address_instructions),
31.     .o_Address_Data_Bus(address_data)
32.     );
33. endmodule
```