



Centro Universitario de Ciencias Exactas e Ingenierías Departamento de Ciencias Computacionales

Asignatura: **Seminario de Solución de Problemas de Arquitectura de Computadoras**

Clave de Asignatura: **I7024**

<<Proyecto Final: Fase 1, 2 y 3>>

Alumno: **<<Casas Chavarría Diego Maximiliano**

García Ramírez José Manuel

Rubio Andrade Athziri Magdalena>>

Profesor: **López Arce Delgado Jorge Ernesto**

Fecha: **<<Jueves 24 Junio 2021 >>**



Contenido

Introducción fase 1	2
Introducción fase 2	7
Introducción fase 3	22
Objetivo Fase 1	30
Objetivo Fase 2	30
Desarrollo Fase 1	31
Desarrollo Fase 2	35
Desarrollo Fase 3	45
Conclusión fase 1	53
Conclusión fase 2	55
Conclusión fase 3	57
Referencias	59



1. Introducción fase 1

Este reporte incluye una breve introducción a los antecedentes del procesador MIPS y una explicación de los principios y técnicas utilizadas en la implementación de un procesador.

Los procesadores MIPS fueron creados en el año 1998.

En la universidad de Standford un equipo liderado por Jonh L. quien fue fundador de MIPS Technologies. La idea de Jonh era mejorar el rendimiento de la máquina a través del uso de la segmentación (proceso paralelo consistente en el procesador simultáneo de instrucciones en diferente fase).

En 1984 Hennessy dejó Standford y fundar MIPS Computer Systems. Gracias a este personaje obtenemos su primer diseño llamado R2000, creado en 1985.

Las características generales de los procesadores MIPS. Recordaremos el significado de MIPS el cual es Microprocessor Without Interlocked Pipeline Stages en inglés, pero su significado en español es Microprocesador sin enclavamiento de estado de tuberías o también referenciado a Microprocesador sin Bloqueos en las Etapas de Segmentación. Se basa al conjunto de microprocesadores desarrollados por MIPS Technologies, provenientes de la arquitectura RISC y registros de tipo propósito general de clasificación registro - registro, es importante saber que la mayoría de instrucciones no acceden a la memoria a excepción de la instrucciones de carga y descarga, por otro lado las instrucciones de los procesadores cuentan con dos operandos, Fuente y Resultado.

A continuación se muestran algunas características del procesador MIPS:

- Un PC (Contador de programa)
- Un incrementador para el PC
- Una ALU que opera con longitudes de palabra de 32 bits
- Una memoria de instrucciones de 64 posiciones por 32
- Una memoria de datos
- Una unidad de extensión de signo
- 32 registros de 32 bits
- Unidad de control
- Control de la ALU
- Multiplexores

Comentará brevemente sobre el conjunto de instrucciones el cual permite realizar instrucciones de carga y de almacenamiento desde la memoria y hacia ella, cuenta con la capacidad de desarrollar programas que resuelven problemas aritméticos y lógicos, también puede controlar el flujo de la ejecución del programa mediante instrucciones de salto, tanto condicional como incondicional.

Cuando hablamos de nuestro procesador MIPS de 32 bits. Las instrucciones se pueden clasificar en función de los elementos que utiliza (banco de registros, memoria de datos, ALU). Los componentes de las instrucciones se especifican en una serie de bits ya que los distintos tipos de instrucciones constan de diferentes tamaños para los espacios de esos bits o también llamados campos, utilizan formatos diferentes para codificar sus campos.



En el caso de esta primera fase se distingue el tipo de formato de instrucción: Formato R o de tipo registro.

En esta primera fase llevaremos a cabo el siguiente set de instrucciones específico para el proyecto:

Las instrucciones que tienen la capacidad de reconocer el procesador, junto con sus códigos de operación, son las siguientes:

Formato R: add , sub

Lógicas: or, and

Transferencia de datos: lw , sw

Salto condicional: slt , beq

Bifurcación incondicional: j

Burbuja: nop

Parada del procesador: halt

Instrucciones tipo “R”:

Nombre	Instrucción	Tipo	Sintaxis
Suma (add)	add	R	add \$rd \$rs \$rt
Resta (subtract)	sub	R	sub \$rd \$rs \$rt
Multiplicación (multiply)	mul	R	mul \$rd \$rs
División (divide)	div	R	div \$rs \$rt
O (or)	or	R	or \$rd \$rs \$rt
Y (and)	and	R	and \$rd \$rs \$rt
Sin operación (No operation)	nop	R	nop \$rd \$rs \$rt
Establecer menos de (Set less than)	slt	R	slt \$rd \$rs \$rt

Tabla 1.1



Error TestF1_MemInst

Original	Corrección
00000000	00000000
00100010	00100010
10100000	10100000
00000000	00100000 -- ADD Aquí se encuentra el error porque cada 4 direcciones debe ir una instrucción y esta era la faltante
00000000	00000000
10100100	10100100
10101000	10101000
00100010	10101000
00000001	00100010 -- SUB
00000100	00000001
11000000	00000100
00100100	11000000
00000001	00100100 -- AND
01001010	00000001
11001000	01001010
00100101	11001000
00000000	00100101 -- OR
10101010	00000000
11010000	10101010
00100101	11010000
00000000	00100101 -- SLT
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000 -- NOP

Tabla 1.2

Algoritmo Código Ensamblador

Lenguaje ensamblador:

El único lenguaje que entienden los microcontroladores es el código máquina formado por ceros y unos del sistema binario, expresando las instrucciones de una forma más natural al hombre a la vez que muy cercana al microcontrolador, ya que cada una de esas instrucciones se corresponde con otra en código máquina. El lenguaje ensamblador trabaja con mnemónicos, que son grupos de caracteres alfanuméricos que simbolizan las órdenes o tareas a realizar. La traducción de los mnemónicos a código máquina entendible por el microcontrolador la lleva a cabo un programa ensamblador. El programa escrito en lenguaje ensamblador se denomina código fuente (*.asm). El programa ensamblador proporciona a partir de este fichero el correspondiente código máquina, que suele tener la extensión *.hex.



Propuesta de algoritmo:

```
.data #Cominezo de los datos  
dato: .asciz  
nums: .int 0,1,2,3,4,5,6,7,8,9  
.text #Comienzo del código  
.globl main
```

main:

bucle:

```
    add $0, $1, $3  
    sub $4, $5, $6  
    and $7, $8, $9  
    or $0, $0, $3  
    slt $0, $4, $5  
    nop $0, $0, $0
```

fin:

ret #Programa termina

```
Programa ensamblador 1 >  
1     .data #Cominezo de los datos  
2     dato: .asciz  
3     nums: .int 0,1,2,3,4,5,6,7,8,9  
4     .text #Comienzo del código  
5     .globl main  
6  
7     main:  
8         bucle:  
9             add $0, $1, $3  
10            sub $4, $5, $6  
11            and $7, $8, $9  
12            or $0, $0, $3  
13            slt $0, $4, $5  
14            nop $0, $0, $0  
15     fin:  
16     ret #Programa termina
```

Imagen 1.1



Comentario:

Tuve muchas complicaciones al estar investigando sobre el tema, nunca había programado en lenguaje ensamblador, mi razonamiento para proponer este código fue el siguiente: defini en la data un tipo “arreglo” de 10 valores, después declaré un bucle para cumplir con la iteración, y dentro de este escribí las instrucciones de esta primera fase, seleccionando direcciones de mi “arreglo”. No tengo la seguridad de que esté correcto, sin embargo según mi investigación fue lo que pude razonar.



2. Introducción fase 2

Este reporte correspondiente a la fase 2 del proyecto, contiene una descripción de las instrucciones del tipo “I” para un “datapath” con arquitectura tipo MIPS de 32 bits, usando como base el “datapath” descrito en el libro de “Computer Organization and Design, the hardware/software interface, David A. Patterson; John L. Hennessy, Fifht Ed.” Capítulo 4.

A manera de introducción sencilla, se van a describir de manera general para posteriormente expresar específicamente las instrucciones y el funcionamiento en el desarrollo de este mismo reporte.

El formato tipo I es utilizado para las instrucciones de transferencia, las de salto condicional y las instrucciones con operandos inmediatos. Su formato se representa de la siguiente manera:

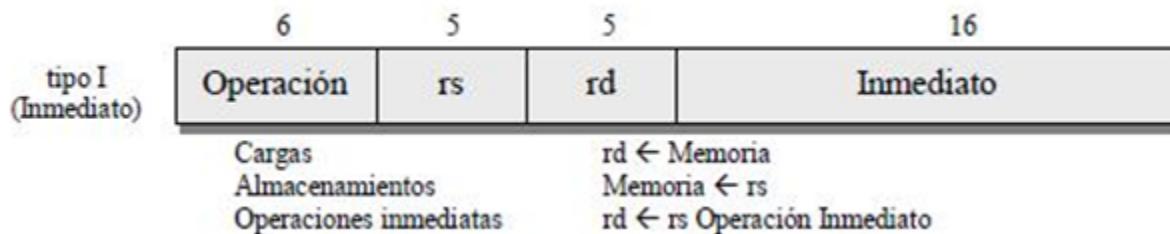


Imagen 2.1

Como podemos observar se divide en las siguientes secciones:

- Bits 31-26:

En estos primeros seis bits encontramos el espacio para el código de operación que identifica a cada instrucción, en este caso por ser el tipo de instrucción I será el código correspondiente a LW, SW, BEQ, SLTI, ADDI, ORI, ANDI

- Bits 25-21:

Los siguientes cinco bits son los destinados a el registro base.

- Bits 20-16:

Los siguientes cinco bits son los que representan al registro destino.

- Bits 15-0:



Estos últimos dieciséis bits son los destinados al apartado conocido como “desplazamiento”.

Las instrucciones de tipo I que se van a analizar en esta fase 2 y su breve descripción son:

- LW:
 - ❖ FORMATO:

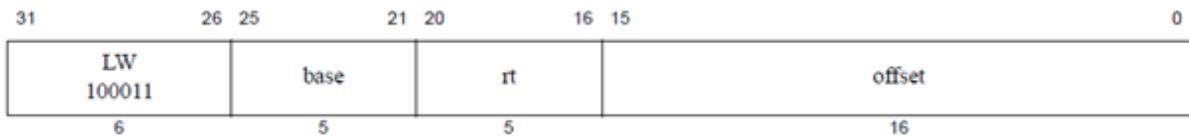


Imagen 2.2

- ❖ Descripción:

Es mover un dato de la memoria hacia el banco de registros, de nuestro “datapath”. La base es la dirección del banco de registros donde se encuentra la *dirección base* de nuestra memoria (es decir, el valor que hay en esa dirección es la dirección de referencia en nuestra memoria), el rt es la dirección donde queremos guardar el dato que vamos a obtener de la memoria y por último el offset es la cantidad de direcciones que nos tenemos que mover para encontrar el dato de la memoria que deseamos.

Ejemplo: Queremos mover el dato “123” que se encuentra en nuestra memoria a la dirección \$4 de nuestro banco de registros, se puede ver expresado en la siguiente secuencia de pasos:

IDENTIFICACIÓN DE DATOS:



Banco de registros		Memoria	
Dirección	Dato	Dirección	Dato
\$0	3	\$0	4
\$1	0	\$1	33
\$2	1	\$2	12
\$3	6	\$3	123
\$4	9	\$4	6

Tabla 2.1

Aquí observamos de color amarillo la dirección donde queremos almacenar el dato (rt) y vemos de rojo el dato que queremos mover de nuestra memoria.

DEFINICIÓN DE LA BASE:

Banco de registros		Memoria	
Dirección	Dato	Dirección	Dato
\$0	3	\$0	4
\$1	0	\$1	33
\$2	1	\$2	12
\$3	6	\$3	123
\$4	9	\$4	6

Tabla 2.2

Aquí lo que podemos observar es que se toma como “base” la dirección \$1 de nuestro banco de registros que almacena el dato “0”, que será nuestra dirección base (\$0 de la memoria) a partir de la cual obtendremos nuestro offset.

GENERACIÓN DEL OFFSET:



Ahora simplemente hay que contar, cuántas direcciones nos hacen falta para llegar de la dirección base a la dirección que almacena el dato que deseamos, en este caso el “123” con dirección \$3 en la memoria.

Dirección base = 0

Dirección que almacena el “123” = 3

Offset: $0 + x = 3$

$x = 3$

Por lo tanto nuestro offset en binario sería: 0000 0000 0000 0011

CREACIÓN DEL CÓDIGO:

- ❖ LW: 100011
- ❖ base: 00001
- ❖ rt: 00100
- ❖ offset: 000000000000000011

Por lo tanto el código sería: 1000110000100100000000000000000011

• SW:

- ❖ Formato:

31	26 25	21 20	16 15	0
SW 101011	base	rt		offset

Imagen 2.3

- ❖ Descripción:

Es mover un dato del banco de registros hacia la memoria, de nuestro “datapath”. La base es la dirección del banco de registros donde se encuentra la *dirección base* de nuestra memoria (es decir, el valor que hay en esa dirección es la dirección de referencia en nuestra memoria), el



rt es la dirección que tiene el dato que vamos a mover a la memoria y por último el offset es la cantidad de direcciones que nos tenemos que mover para encontrar la dirección dónde queremos almacenar el dato.

Ejemplo: Queremos mover el dato “9” que se encuentra en nuestro banco de registros en la dirección \$4 a la dirección \$3 de nuestra memoria, se puede ver expresado en la siguiente secuencia de pasos:

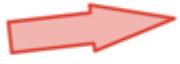
IDENTIFICACIÓN DE DATOS:

Banco de registros		Memoria	
Dirección	Dato	Dirección	Dato
\$0	3	\$0	4
\$1	0	\$1	33
\$2	1	\$2	12
\$3	6	\$3	123
\$4	9	\$4	6

Tabla 2.3

Aquí observamos de color amarillo la dirección donde queremos almacenar el dato y vemos de rojo la dirección y el dato que queremos mover a la memoria.

DEFINICIÓN DE LA BASE:

Banco de registros		Memoria	
Dirección	Dato	Dirección	Dato
\$0	3	\$0	4
\$1	0	\$1	33
\$2	1	\$2	12
\$3	6	\$3	123
\$4	9	\$4	6

Tabla 2.4

Aquí lo que podemos observar es que se toma como “base” la dirección \$1 de nuestro banco de registros que almacena el dato “0”, que será nuestra dirección base (\$0 de la memoria) a partir de la cuál obtendremos nuestro offset.

GENERACIÓN DEL OFFSET:

Ahora simplemente hay que contar, cuántas direcciones nos hacen falta para llegar de la dirección base a la dirección a donde deseamos guardar el dato, que en este caso sería \$3 en la memoria.

Dirección base = 0

Dirección que almacena el “123” = 3

Offset: $0 + x = 3$

$x = 3$

Por lo tanto nuestro offset en binario sería: 0000 0000 0000 0011

CREACIÓN DEL CÓDIGO:

- ❖ SW: 101011
- ❖ base: 00001
- ❖ rt: 00100
- ❖ offset: 0000000000000011

Por lo tanto el código sería: 10101100001001000000000000000011



- BEQ(Branch on Equal):
 - ❖ Formato:

31	26 25	21 20	16 15	0
BEQ 000100	0 00000	0 00000		offset 16

Imagen 2.4

BEQ rs, rt, offset

- ❖ Descripción:

Esta instrucción consiste en si GPR [rs] = GPR [rt] entonces bifurca. Un desplazamiento con signo de 18 bits (el campo de desplazamiento de 16 bits desplazado a la izquierda 2 bits) se agrega a la dirección de la siguiente instrucción la rama (no la rama en sí), en la ranura de retardo de la rama, para formar una dirección de destino efectiva relativa a la PC. Si los contenidos de GPR rs y GPR rt son iguales, bifurque a la dirección de destino efectiva después de la instrucción en el retardo se ejecuta la ranura.

- SLTI (Set on Less Than Immediate):
 - ❖ Formato:

31	26 25	21 20	16 15	0
SLTI 001010	rs 5	rt 5		immediate 16

Imagen 2.5

SLTI rt, rs, immediate

- ❖ Descripción:

GPR [rt] (GPR [rs] <sign_extend (inmediato))
Compare el contenido de GPR rs y el inmediato con signo de 16



bits como enteros con signo; registrar el resultado booleano de la comparación en GPR rt. Si GPR rs es menor que inmediato, el resultado es 1 (verdadero); de lo contrario, es 0 (falso).

La comparación aritmética no provoca una excepción de desbordamiento de enteros.

- ADDI(Add Immediate Word):
 - ❖ Formato:

31	26 25	21 20	18 15	0
ADDI 001000	rs	rt	immediate	16

Imagen 2.6

ADDI rt, rs, immediate

- ❖ Descripción:

GPR [rt] GPR [rs] + inmediato

El inmediato con signo de 16 bits se agrega al valor de 32 bits en GPR rs para producir un resultado de 32 bits.

- Si la suma da como resultado un desbordamiento aritmético del complemento 2 de 32 bits, el registro de destino no se modifica y se produce una excepción de desbordamiento de enteros.
 - Si la suma no se desborda, el resultado de 32 bits se coloca en GPR rt.
- ANDI(and immediate):
 - ❖ Formato:



31	26 25	21 20	16 15	0
	ANDI 001100	rs	rt	immediate

Imagen 2.7

ANDI rt, rs, immediate

- ❖ Descripción:
- ❖ GPR [rt] GPR [rs] y zero_extend (inmediato)
El inmediato de 16 bits se extiende a cero a la izquierda y se combina con el contenido de GPR rs en un AND lógico bit a bit operación. El resultado se coloca en GPR rt.

- ORI (Or Immediate):

- ❖ Formato:

31	26 25	21 20	16 15	0
	ORI 001101	rs	rt	immediate

Imagen 2.8

ORI rt, rs, immediate

- ❖ Descripción:

GPR [rt] GPR [rs] o inmediato

El inmediato de 16 bits se extiende a cero a la izquierda y se combina con el contenido de GPR rs en un OR lógico bit a bit operación. El resultado se coloca en GPR rt.

Instrucciones de tipo “I”:



Instrucción	Tipo	Sintaxis
Add	R	add \$rd \$rs \$rt
Sub	R	sub \$rd \$rs \$rt
Mul	R	mul \$rd \$rs
Div	R	div \$rs \$rt
Or	R	or \$rd \$rs \$rt
And	R	and \$rd \$rs \$rt
Addi	I	addi \$rt \$rs \$inmediate
Subi	I	subi \$rt \$rs \$inmediate
Ori	I	ori \$rt \$rs \$inmediate



Andi	I	andi \$rt \$rs \$inmediate
Madd	R	madd \$rs \$rt
Nor	R	nor \$rd \$rs \$rt

Tabla 2.5

Instrucciones tipo “I”:

Instrucción	Tipo	Sintaxis
Lw	I	lw \$rt #offset(base)
Sw	I	sw \$rt #offset(base)
Slt	R	slt \$rd \$rs \$rt
Slti	I	slti \$rt \$rs \$inmediate
Beq	I	beq \$rs \$rt #offset(base)



Bne	I	bne \$rs \$rt #offset(base)
Nop	R	nop \$rd \$rs \$rt
Bgtz	I	bgtz \$rs \$etiqueta
Beql	I	beql \$rs \$rt #offset(base)
Bnel	I	bnel \$rs \$rt #offset(base)
Xor	R	xor \$rd \$rs \$rt

Tabla 2.6

Pipelining

El pipeline es una técnica para implementar simultaneidad a nivel de instrucciones dentro de un solo procesador. Pipelining intenta mantener ocupada a cada parte del procesador, dividiendo las instrucciones entrantes en una serie de pasos secuenciales, que se realizan por diferentes unidades del procesador que trabajan de forma simultánea.

Hacer uso del pipelining aumenta el rendimiento de nuestro “datapath” ya que al segmentarlo en cinco secciones lo que hacemos es optimizar tiempos, las secciones y su descripción son:

- Instruction fetch:



Esta es la primera sección, en la que se inicializa nuestro “datapath” con la selección de la instrucción de la memoria de instrucciones y por otro lado a esa misma indicación se le suman las cuatro para la siguiente instrucción. Todo esto entra al primer buffer “IF/ID”.

- Instruction decode:

En esta segunda sección se toman los datos del buffer “IF/ID” para decodificar la instrucción, tomando los datos de nuestro banco de registros y haciendo lo determinado al sign extend, todo esto entra al segundo buffer “ID/EX”.

- Execution:

Para la tercera sección se toman los datos del buffer “ID/EX” y lo que se realiza en pocas palabras es la ejecución de la instrucción y los resultados pasan al buffer llamado “EX/MEM”.

- Memory:

Para esta la cuarta sección se toman los datos del buffer “EX/MEM” y se determina si se escribe o se lee en memoria y los resultados pasan al buffer llamado “MEM/WB”.

- Write-back:

Para la quinta y última sección se toman los datos del buffer “MEM/WB” y su función es determinar si se reescribe un valor en nuestro banco de registros o no.

Lo que hace implementar esta técnica es que por un decir, se da un pulso de reloj y se ejecuta la primera sección, al segundo se ejecutará la primera sección y la segunda al mismo tiempo, al siguiente se ejecutará al mismo tiempo la primera, segunda y tercera sección, y así sucesivamente hasta terminar. Adjuntaré una foto que facilita el entendimiento de cómo funciona esto, que es cortesía de “Computer Organization and Design, the hardware/software interface, David A. Patterson; John L. Hennessy, Fifht Ed.”

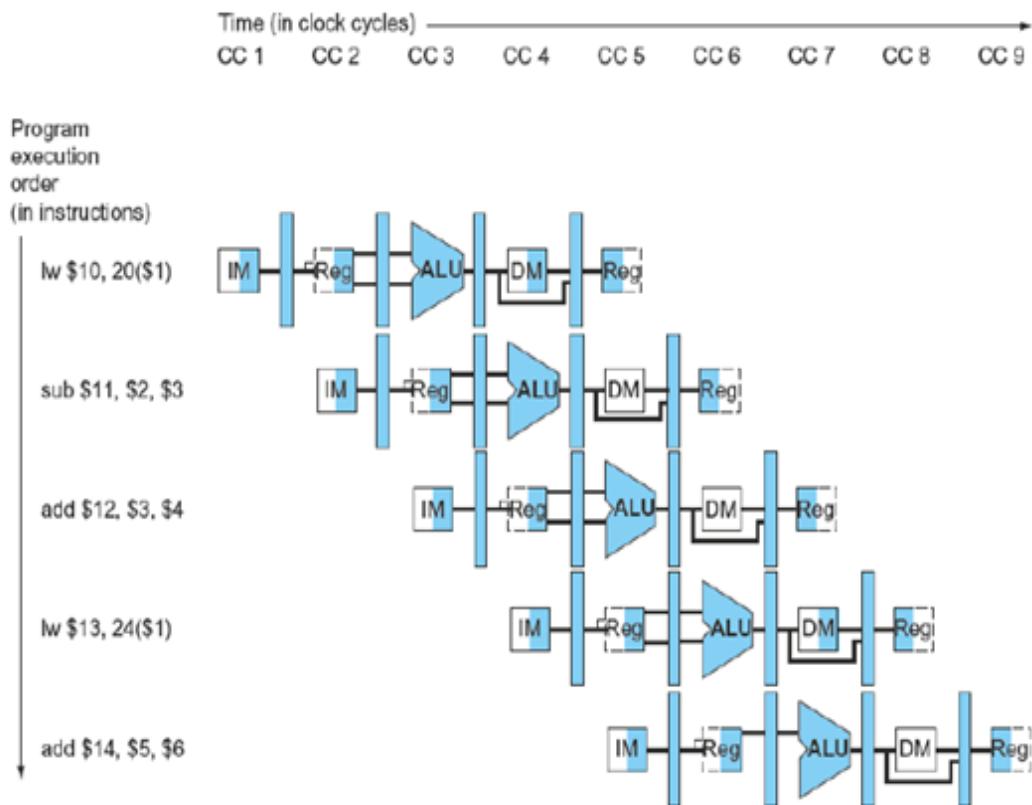


FIGURE 4.43 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure. Instructions are listed in instruction execution order from top to bottom, and clock cycles move from left to right. Unlike Figure 4.28, here we show the pipeline registers between each stage. Figure 4.44 shows the traditional way to draw this diagram.

Imagen 2.9



3. Introducción fase 3

Esta es la tercera parte del reporte y última donde nos introduce a redactar módulo por módulo de nuestro datapath y con sus modificaciones para que puedan ser utilizadas las instrucciones tipo J.

También se presentará el diagrama de flujo de nuestro lenguaje en ensamblador que se implementará en nuestro código verilog para que haga las operaciones que nosotros le pedimos.

Instrucciones Tipo “J”:

La instrucción Jump es tipo “J” y es la que hace un salto incondicional o una bifurcación es decir, la instrucción obliga a la máquina a seguir siempre el salto. Para distinguir entre saltos condicionales e incondicionales, el nombre MIPS para este tipo de instrucción es Jump.

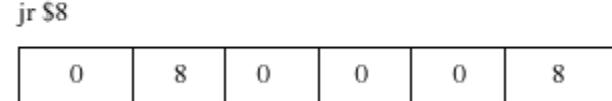
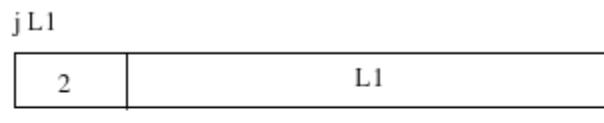


Figura 2.11: Estructura de las instrucciones *jump* y *jump register* en el MIPS

Imagen 3.1

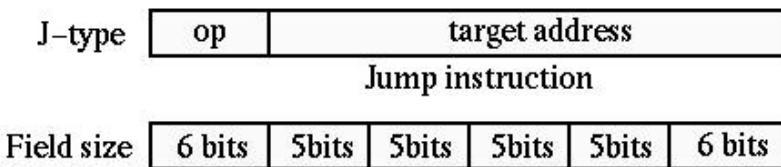


Imagen 3.2



Como podemos observar se divide en dos secciones:

- Bits 31-26:

En estos primeros seis bits encontramos el espacio para el código de operación que identifica a cada instrucción, en este caso por ser el tipo de instrucción J será el código “000010”.

- Bits 25-0:

Los siguientes veintiséis bits son los destinados a la dirección destino.

A continuación hablaremos de CRC.

La **verificación por redundancia cíclica** (CRC) es un código de detección de errores usado frecuentemente en redes digitales y en dispositivos de almacenamiento para detectar cambios accidentales en los datos.¹ Los bloques de datos ingresados en estos sistemas contiene un *valor de verificación adjunto*, basado en el residuo de una división de polinomios; el cálculo es repetido, y la acción de corrección puede tomarse en contra de los datos presuntamente corruptos en caso de que el valor de verificación no concuerde. Este código es un tipo de función que recibe un flujo de datos de cualquier longitud como entrada y devuelve un valor de longitud fija como salida. El término suele ser usado para designar tanto a la función como a su resultado. Pueden ser usadas como suma de verificación para detectar la alteración de datos durante su transmisión o almacenamiento. Las CRC son populares porque su implementación en *hardware* binario es simple, son fáciles de analizar matemáticamente y son particularmente efectivas para errores ocasionados por ruido en los canales de transmisión. La CRC fue inventada y propuesta por W. Wesley Peterson.



Algoritmo código ensamblador

BANK		DATA	
i	0	0	1
I	1	1	1
t	2	2	0
	3	3	0
	4	4	0
	5	5	0
	6	6	0
	7	7	1
	8	8	0
	9	9	1
	10	10	0
	11	11	0
	12	12	1
	13	13	0

Tabla de verdad de la compuerta XOR

Entrada A	Entrada B	Salida A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

#i=\$0

#l=límite=\$1

#t=\$2 (registro temporal)

#Cargamos el mensaje en la memoria:(1101)

```
lw $0 , $5 , $0 #El dato 0 se guarda en el registro 5 en el banco  
lw $0 , $6 , $1 #El dato 1 se guarda en el registro 6 en el banco  
lw $0 , $7 , $2 #El dato 2 se guarda en el registro 7 en el banco  
lw $0 , $8 , $3 #El dato 3 se guarda en el registro 8 en el banco
```

#Ahora cargamos el siguiente mensaje en los siguientes 4 registros

```
lw $0 , $9 , $4 #El dato 4 se guarda en el registro 9 en el banco  
lw $0 , $10 , $5 #El dato 5 se guarda en el registro 10 en el banco  
lw $0 , $11 , $6 #El dato 6 se guarda en el registro 11 en el banco  
lw $0 , $12 , $7 #El dato 7 se guarda en el registro 12 en el banco
```

#Una vez guardados los datos en los registros correspondientes, se implementan las operaciones necesaria para nuestra primera división sintética

#contador:

```
addi $i , $0 , 0 #Inicio
```

```
addi $I , $0 , 4 #Límite
```



#for:

xor \$14 , \$18 , \$5 #Se aplica Xor a los registros 6 y 10 para almacenar el resultado en el registro 5 del banco

xor \$7 , \$11 , \$6 #Se aplica Xor a los registros 7 y 11 para almacenar el resultado en el registro 6 del banco

xor \$8 , \$12 , \$7 #Se aplica Xor a los registros 8 y 12 para almacenar el resultado en el registro 7 del banco

sub \$i , \$i , \$8 #

beq \$i , \$l , 2 #si nuestro registro (i) es igual al registro (l) hay que salir del ciclo for y saltar dos operaciones j y addi

#suma de nuestro índice (++)

addi \$i , \$i , 1 #el registro (i) lo guardas en el registro i y sumamos el 1 y si no son iguales se hace i++ y pasamos a la siguiente instrucción j

j 13 #Regresamos al inicio del ciclo, brincamos a la dirección 13 donde se tiene el inicio de ciclo

DATA

BANK		
i	0	1
l	1	4
t	2	0
	13	1
	14	1
	15	0
	16	1
	17	1
	18	0
	19	0
	20	1

Tabla de verdad de la compuerta XOR

Entrada	Entrada	Salida
A	B	A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

8	0
9	1
10	1
11	1
12	1
13	0
14	0
15	1

#Segunda división sintética con residuo obtenido de la anterior división sintética
#Cargamos el mensaje en la memoria:(0111)

lw \$0 , \$13 , \$8 #El dato 8 se guarda en el registro 13 en el banco

lw \$0 , \$14 , \$9 #El dato 9 se guarda en el registro 14 en el banco

lw \$0 , \$15 , \$10 #El dato 10 se guarda en el registro 15 en el banco

lw \$0 , \$16 , \$11 #El dato 11 se guarda en el registro 16 en el banco



#Ahora cargamos el siguiente mensaje en los siguientes 4 registros
#Se cargan los datos y donde se obtiene el CRC (polinomio $x^3+1=1001$)

```
lw $0 , $17 , $12 #El dato 12 se guarda en el registro 17 en el banco
lw $0 , $18 , $13 #El dato 13 se guarda en el registro 18 en el banco
lw $0 , $19 , $14 #El dato 14 se guarda en el registro 19 en el banco
lw $0 , $20 , $15#El dato 15 se guarda en el registro 20 en el banco
#Una vez guardados los datos en los registros correspondientes, se implementan las
operaciones necesaria para nuestra primera división sintética
```

#contador:

```
addi $i , $0 , 0 #Inicio
addi $l , $0 , 4 #Límite
```

#for:

```
xor $14 , $18 , $13 #Se aplica Xor a los registros 14 y 18 para almacenar el resultado en
el registro 13 del banco
xor $15 , $19 , $14 #Se aplica Xor a los registros 15 y 19 para almacenar el resultado en
el registro 14 del banco
xor $16 , $20 , $15 #Se aplica Xor a los registros 16 y 20 para almacenar el resultado en
el registro 15 del banco
sub $i , $i , $12
beq $i , $l , 2 #si nuestro registro (i) es igual al registro (l) hay que salir del ciclo for y saltar
dos operaciones j y addi
#suma de nuestro índice (++)
addi $i , $i , 1 #el registro (i) lo guardas en el registro i y sumamos el 1 y si no son iguales
se hace i++ y pasamos a la siguiente instrucción j
j 13 #Regresamos al inicio del ciclo, brincamos a la dirección 13 donde se tiene el inicio
de ciclo
```

Tabla 3.1

Instrucciones tipo “J”:



Instrucción	Tipo	Sintaxis
Add	R	add \$rd \$rs \$rt
Sub	R	sub \$rd \$rs \$rt
Mul	R	mul \$rd \$rs
Div	R	div \$rs \$rt
Or	R	or \$rd \$rs \$rt
And	R	and \$rd \$rs \$rt
Addi	I	addi \$rt \$rs \$inmediate
Subi	I	subi \$rt \$rs \$inmediate
Ori	I	ori \$rt \$rs \$inmediate



Andi	I	andi \$rt \$rs \$inmediate
Madd	R	madd \$rs \$rt
Nor	R	nor \$rd \$rs \$rt

Tabla 3.2

Instrucción	Tipo	Sintaxis
Lw	I	lw \$rt #offset(base)
Sw	I	sw \$rt #offset(base)
Slt	R	slt \$rd \$rs \$rt
Slti	I	slti \$rt \$rs \$inmediate
Beq	I	beq \$rs \$rt #offset(base)



Bne	I	bne \$rs \$rt #offset(base)
J	J	j \$target
Nop	R	nop \$rd \$rs \$rt
Bgtz	I	bgtz \$rs \$etiqueta
Beql	I	beql \$rs \$rt #offset(base)
Bnel	I	bnel \$rs \$rt #offset(base)
Xor	R	xor \$rd \$rs \$rt

Tabla 3.3



4. Objetivo Fase 1

Los objetivos esperados para esta primera fase los consultamos en equipo, llegando a la conclusión de que nuestro objetivo principal debe ser el reporte ya que este será utilizado para el funcionamiento del código, basándonos en la información que el profesor nos ha brindado a lo largo de este semestre y en conjunto con la información de este reporte esperamos poder implementar un código que pueda funcionar y ayudarnos a que sea más fácil el desarrollo del siguiente código, para disminuir los posibles obstáculos que se nos puedan presentar en las siguientes fases. Los módulos que realizaremos los complementaremos con algunos que ya tenemos de clases anteriores, realizaremos nuevos módulos con ayuda de este reporte y el libro para el funcionamiento de nuestro procesador MIPS de 32 bits. Realmente este será un gran reto para el equipo, pero con buena organización, conocimiento, aprendizaje, ganas y tiempo lo lograremos.

5. Objetivo Fase 2

Diseñar un “datapath” con arquitectura tipo MIPS de 32 bits capaz de ejecutar las 28 instrucciones previamente definidas por ustedes, complete la tabla 1 y tabla 2 con la sintaxis y 5 instrucciones elegidas por usted.

Debe de elegir un algoritmo previamente aprobado por su profesor, y que sea posible implementar con el set reducido de instrucciones de la tabla 1 y 2. Este programa previamente definido en ensamblador debe ser codificado a código binario y precargado en la memoria de instrucciones para que el datapath lo ejecute, recuerde definir cada uno de los aspectos de dicho programa, secciones de los registros en el banco de registros para base pointers, resultados, resultado de comparaciones, etc. Así como los datos pre-cargados en su memoria de datos.

-Objetivo particular fase 2:

Agregar los módulos necesarios al datapath para poder ejecutar las instrucciones tipo I de la tabla 2.5 y tabla 2.6.

6. Objetivo Fase 3

Diseñar un “datapath” con arquitectura tipo MIPS de 32 bits capaz de ejecutar las 28 instrucciones previamente definidas por ustedes, complete la tabla 1 y tabla 2 con la sintaxis y 5 instrucciones elegidas por usted.

Los objetivos principales de la fase 3 serán agregar los módulos necesarios para completar el datapath de MIPS de 32 bits para que sea capaz de ejecutar las instrucciones tipo “J”.

Que el código verilog logre correr correctamente todas las instrucciones y de los resultados esperados.



Y lo más importante de esta fase es terminar el algoritmo de lenguaje ensamblador para decodificarlo en binario y que nuestro código en verilog lo corra y de el resultado esperado..



7. Desarrollo Fase 1

Lógica de la unidad de control

Nuestro diseño es basado a nivel de puertas lógicas. Ya que sólo reconoce las instrucciones de tipo R, lw, sw, beq; en el simulador y se agregó la parada del procesador (halt), bifurcación a una dirección (j), para su funcionamiento agregamos dos puertas AND además de las 4 existentes de que disponía la UC.

Lógica de ALU control

Su diseño es a nivel de puertas. Le llegan dos entradas y su lógica genera una salida que va directa a la ALU, esta salida es de 3 bits y dice a la ALU si tiene que efectuar una operación aritmética, lógica, de transferencia de datos, o de salto ya sea condicional o incondicional. Una de las entradas proviene de decodificar la instrucción y corresponden a los 6 LSB de dicha instrucción, es el campo función, y los otros bits de entrada corresponden a los que genera la UC.

Unidad extensión de signo

Se encarga de convertir un bus de 16 líneas en otro de 32. Al método run de la clase Signo Extendido le llega un array de entrada de tipo WIRE que es de 16 posiciones, entonces en un array de salida de 32 posiciones, se copia en las primeras 16 el array de entrada y el MSB es copiado 16 veces a la izquierda hasta completar el array de salida, ya se ha convertido el bus de entrada de 16 bits a uno de 32.

PC

Para su implementación se ha utilizado un registro de tipo D, pero que se le ha añadido dos señales que corresponden al reset y clear.

Incrementación del PC

Está construido a partir de una ALU normal de 32 bits, pero el código de operación que se le pasa es el de la suma, éste es generado en la unidad ALU control y consta de 3 bits

Bus de multiplexores

En realidad no se tiene un multiplexor, sino 32 multiplexores, esto es debido a



que cada línea que es representada en los gráficos que aparecen en los libros y que le llega al multiplexor es realmente un bus de 32 líneas, ya que esta es la longitud de palabra del procesador. Entonces a cada uno de los multiplexores le va a llegar una línea de cada bus, que al final las salidas están cortocircuitadas formando otro bus de 32 bits.

Construcción de un multiplexor

Estos se diseñan a partir de buffers triestados y un decodificador. Un buffer triestado va a dejar pasar corriente o no en función de si está a alta impedancia o no. Explicado de mejor forma un buffer triestado, es un inversor de dos estados lógicos, con una línea de control que permite desconectar la salida de la entrada. Los tres estados de salida son:

nivel bajo (cuando la entrada está a nivel alto), nivel alto (cuando la entrada está a nivel bajo) y alta impedancia (cuando la salida está desconectada de la entrada). Cuando la línea de control está a nivel alto, el buffer está activo; y cuando la línea de control está a nivel bajo, el buffer está en estado de alta impedancia.

Ahora que se conoce el funcionamiento de un buffer triestado se explicará el funcionamiento del multiplexor con estos componentes. Cada una de las líneas de cada bus tiene asociado un buffer triestado, entonces cada una de las líneas de control de cada buffer se encuentra enganchado a la salida de un decodificador, éste seleccionará qué conjunto de buffers son los que se tienen que activar, entonces se permitirá el paso de corriente correspondiente a un bus en concreto.

En el simulador del programa se han utilizado la construcción de arrays dinámicos que contienen las señales de cada bus, es decir, si a un multiplexor le llegan dos buses, se construirá un array de tamaño 64 ($2 * 32$ bits), de 0 a 31 se encontrará el primer bus y de 32 a 63 se encontrarán las señales correspondientes al segundo bus.

Si el multiplexor tiene dos líneas de control (se puede direccionar 4 buses de datos) y, sólo le llegan tres buses entonces la última entrada del multiplexor hay que ponerla a tierra.

Diseño de la memoria SRAM

Es una memoria de 64 posiciones por 32bits, se compone de 4 bloques de 16 por 32, es decir, hacen falta 4 líneas para direccionar cada bloque ($2^4=16$) y cada bloque puede ser seleccionado de forma independiente. Aparte se encuentra la línea write, que permitirá la lectura o escritura de la memoria y el bus de entrada de datos y el de salida, ambos claramente de una anchura de 32 líneas.



Cargador de datos e instrucciones

El primer paso es leer el archivo de datos o instrucciones mediante el método lee bus que se encuentra en bus.cpp. Segundo se direcciona la memoria, son necesarias 6 iteraciones debido al tamaño de la memoria ($2^6 = 64$). En el caso de leer el contenido de la memoria la señal write debe estar a GND, si lo que se va a hacer es escribir en ella entonces write tiene que estar a VCC.

Cuando se realiza el volcado de la memoria a un archivo, lo que se está haciendo es copiar las 64 posiciones de ésta en un archivo final en formato binario, tal y como está almacenado en la propia memoria SRAM, durante este proceso por Input Bus lógicamente no va a entrar nada.

Orden de parada del procesador(instrucción Halt)

La instrucción halt se reconoce por el código de operación 63, es decir, todas las líneas de entrada de la puerta AND que se encuentra en el PLA de la unidad de control tienen que estar activas (a 1), no se utiliza a la entrada de esta puerta ningún inversor, ya que la salida debe ser igual a un nivel bajo o alto, en función de si las 6 entradas a la puerta tiene todas uno de estos dos niveles. Ha sido necesario añadir una puerta AND en la unidad de control para esta instrucción.

Compilación:

En archivos separados en formato txt, se deben escribir en uno las instrucciones en ensamblador y en el otro los datos. La sintaxis para las instrucciones es la siguiente, para las de transferencia de datos es

lw registro destino, registro del dato de memoria, desplazamiento
sw dato de registro, registro destino, desplazamiento

Con los ejecutables Convdat y Convinst se compilan los archivo de datos e instrucciones y se pasan a un formato binario, estos después son leídos una vez se ha ejecutado el simulador MIPSm.

Simulación:

Mientras la línea de ControlLines[8] no contenga '1' se estará realizando la simulación, ya que si contiene un nivel alto significa que se ha leído la instrucción halt y por tanto hay que parar la simulación del procesador.

En cuanto al reseteo del procesador que se realiza siempre antes de comenzar a utilizar el micro, consiste en poner la señal clear a GND, el valor de iteraciones del reloj a '0', inicializar el PC para que apunte a la primera posición de memoria y ControlLines[8], que como se ha dicho antes era la señal de halt, ponerla a GND para poder proceder a la simulación.

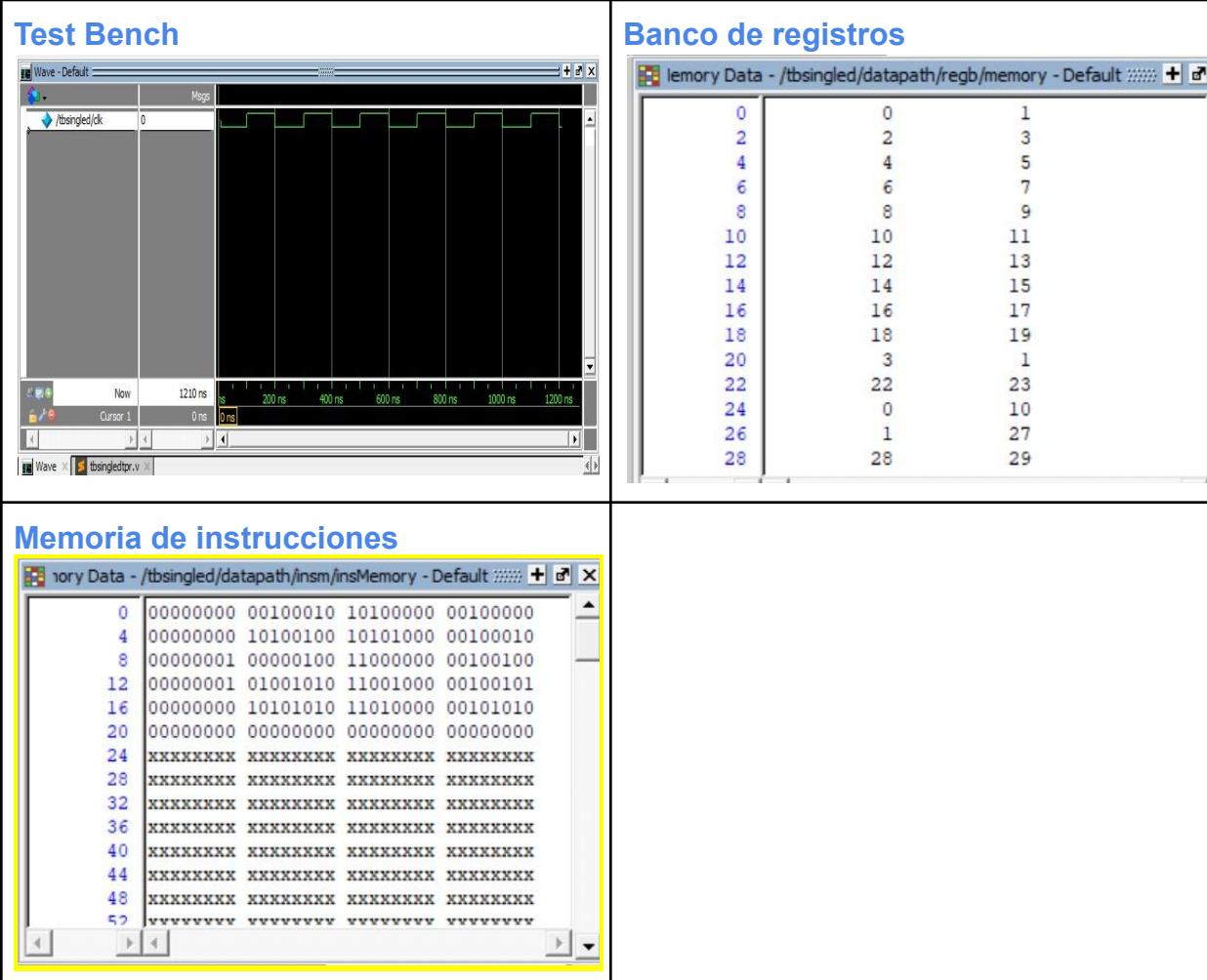


Tabla 1.3



8. Desarrollo Fase 2

ALU.v

Alu es un módulo que consiste en 2 inputs de 32 bits (X,Y), un input de 4 bits (SEL), un output reg que cuenta con 32 bits (R) y por último un output reg de 1 bit (Z_flag). También cuenta con un bloque always el cual se encarga de asignar a R operando destino, X operando fuente 1, Y operando fuente 2.

5'd9: R = X + Y;//addi

5'd10: R = X < Y ;//slti

5'd11: R = X & Y;//andi

5'd12: R = X | Y;//ori

En este módulo se agregaron 4 nuevas instrucciones, mencionadas anteriormente las cuales son aritméticas (addi), lógicas (andi,ori) y de comparación (slti).

alu_ctrl.v

A continuación se mostrarán las modificaciones en alu control, las cuales fueron:

6'b001000: OP = 5'd9; //addi...

6'b001010: OP = 5'd10; //slti...

6'b001100: OP = 5'd11; //andi...

6'b001101: OP = 5'd12; //ori...

Se agregaron las instrucciones addi, slti, andi y ori, con su respectivo Opcode/Function y asignando a OP el número de opción que corresponde a cada una de las instrucciones.

Unit_control.v



En este módulo se hizo la modificación de agregar las instrucciones requeridas para esta fase, comenzando por LW,SW,BEQ,ADDI,ANDI,ORI,SLTI que anteriormente se mencionó cada una de sus funciones.

- Los módulos presentados a continuación, son los módulos nuevos de nuestra fase 2 los cuales se encargan de la transferencia de datos según sean sus conexiones en el datapath
- Cuentan con un input en común que llamamos clk que cuenta con 32 bits

buffer1.v

El buffer lo implementamos con 2 output y 2 input de 32 bits, con nombre haciendo referencia a donde hacen conexión. Ejemplo:

```
input [31:0] inputAdd,  
input [31:0] inputInsMem,  
output reg [31:0] outputAdd,  
output reg [31:0] OutputInsMem
```

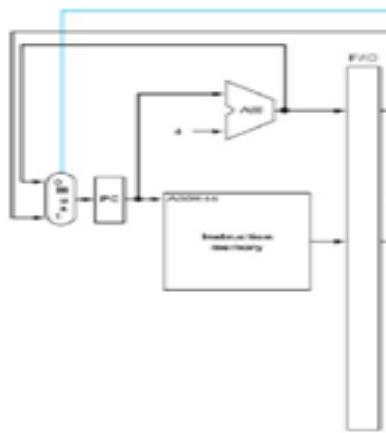


Imagen 2.10

Ahora bien, la parte de nuestro bloque always tiene la funcionalidad de igualar o conectar tanto entrada como salida para que vayan hacia la misma dirección, ya que tanto la entrada como la salida están conectadas en la misma posición.



```
always @(posedge clk)
begin
    outputAdd = inputAdd;
    OutputInsMem = inputInsMem;
end
endmodule
```

buffer2.v

De igual manera en este módulo implementamos 4 input de 32 bits y 2 input de 5 bits, 4 input de 32 bits y 2 output de 5 bits, con nombre haciendo referencia a donde hacen conexión. Ejemplo:

```
input [31:0] inputAddB2,
input [31:0] inputRD1,
input [31:0] inputRD2,
input [31:0] inputSinex,
input [4:0] inputRr1,
input [4:0] inputRr2,
output reg [31:0] outputAddB2,
output reg [31:0] outputRD1,
output reg [31:0] outputRD2,
output reg [31:0] outputSinex,
output reg [4:0] outputRr1,
output reg [4:0] outputRr2
```

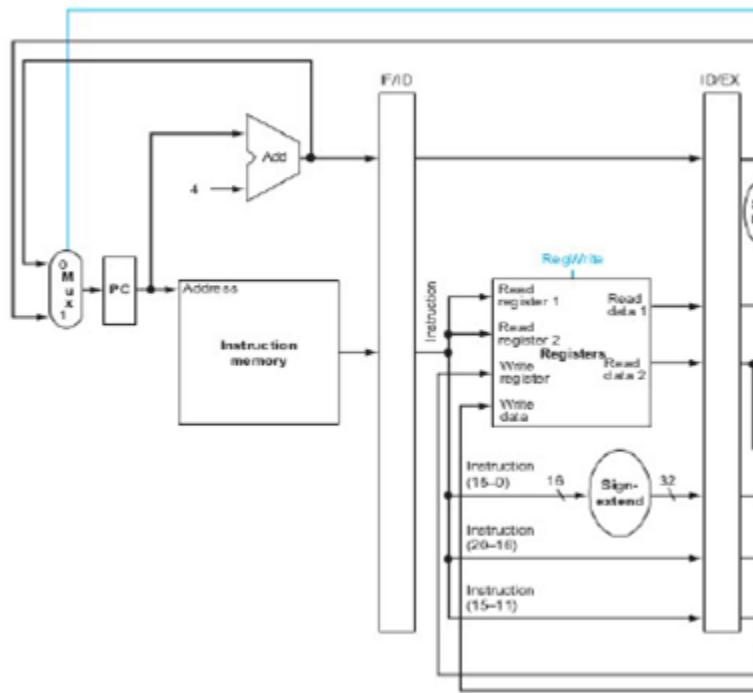


Imagen 2.11

Ahora bien, la parte de nuestro bloque always tiene la funcionalidad de igualar o conectar tanto entrada como salida para que vayan hacia la misma dirección, ya que tanto la entrada como la salida están conectadas en la misma posición.

```

always @(posedge clk)
begin
    outputAddB2 = inputAddB2;
    outputRD1 = inputRD1;
    outputRD2 = inputRD2;
    outputSinex = inputSinex;
    outputRr1 = inputRr1;
    outputRr2 = inputRr2;

```



```
    end  
  
endmodule
```

buffer3.v

En la creación del módulo del buffer tres, nos basamos en las inputs y outputs de 32 bits y solo una entrada y salida de 1 bit, con nombres relacionados a la imagen que a continuación se adjuntará. Ejemplo :

```
input [31:0] inputAddB3,  
input Z_flag,  
input [31:0] inputAddrst,  
input [31:0] inputRD2B3,  
input [4:0] inputmux,  
output reg [31:0] outputAddB3,  
output reg [31:0] outputAddrst,  
output reg [31:0] outputRD2B3,  
input output_Z_flag,  
output reg [4:0] outputmux
```

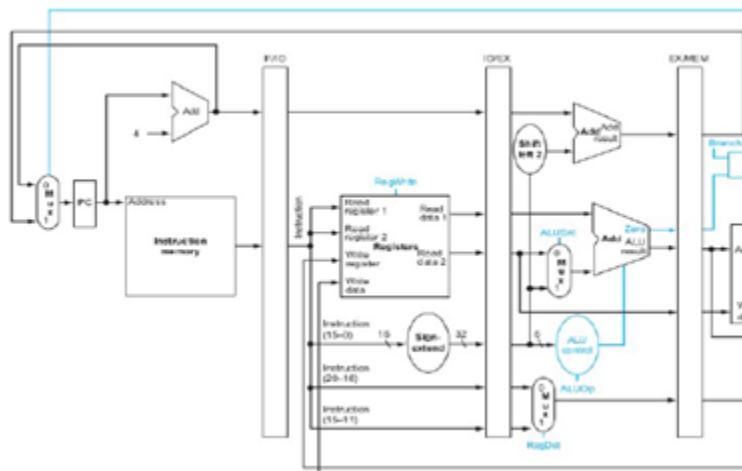


Imagen 2.12

Ahora bien, la parte de nuestro bloque always tiene la funcionalidad de igualar o conectar tanto entrada como salida para que vayan hacia la misma dirección, ya que tanto la entrada como la salida están conectadas en la misma posición.

```

always @(posedge clk)
begin
    outputAddB3 = inputAddB3;
    outputAddrst = inputAddrst;
    outputRD2B3 = inputRD2B3;
    outputmux = inputmux;
end
endmodule

```

buffer4.v

Y es así como llegamos al cuarto buffer que con ayuda de todo el esquema mostrado en la imagen que adjuntamos a continuación, con entradas y salidas de 32 bits según su procedencia. El nombre asignado a cada input output fue basado en las conexiones. Ejemplo:

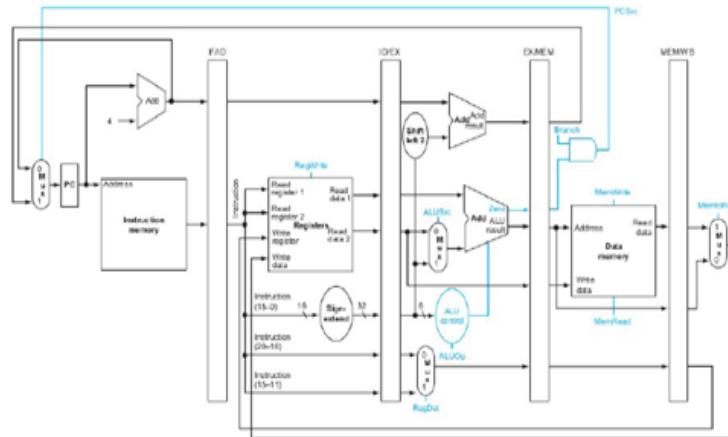


Imagen 2.13

Ahora bien, la parte de nuestro bloque always tiene la funcionalidad de igualar o conectar tanto entrada como salida para que vayan hacia la misma dirección, ya que tanto la entrada como la salida están conectadas en la misma posición.

```
always @(posedge clk)
begin
    outputDatmem = inputDatmem;
    outputAddrstB4 = inputAddrstB4;
    outputmuxB4 = inputmuxB4;
end
endmodule
```

DatapathX.v

En este Datapath se implementaron wires basados en las entradas y salidas (input,output) de nuestros buffer, ya que gracias a ellos podremos realizar todas y cada una de nuestras conexiones dentro del módulo, también se agregaron wires externos a nuestros buffer que hacen conexión única. A continuación se presentan nuestras



conexiones adjuntando imagen de Fase 2, debido a que nos basamos en la imagen para poder lograrlo.

```

DatapathXv
92
93    wire [PC];
94
95    buffer1 b1(clk, inputAdd, inputInsMem, outputAdd, outputInsMem);
96
97    buffer2 b2(INB2MemREG, INB2RegRlTE, INB2Branch, INB2MemRlTE, INB2MemRead, INB2RegDst, INB2ALUOp, INB2ALUSrc, clk, outputAdd, outputRD1, inputRD2, inputSinex, outputInsMem[20:16], outputRD2);
98
99    buffer3 b3(OTB2MemREG, OTB2RegRlTE, OTB2Branch, OTB2MemRlTE, OTB2MemRead, clk, inputAddB3, z_flag, inputAddrst, outputRD2, inputmux, OTB3MemREG, OTB3RegRlTE, OTB3Branch, OTB3MemRlTE);
100
101    buffer4 b4(OTB3MemREG, OTB3RegRlTE, clk, inputDatmem, outputAddrst, outputmux, OTB4MemREG, OTB4RegRlTE, outputDatmem, outputAddrstB4, outputmuxB4);
102
103    unit_control control1(outputInsMem[25:21], outputInsMem[20:16], outputmuxB4, OTB4RegRlTE, MuxBR, inputRD1, inputRD2);
104
105    BankReg reg1(outputInsMem[25:21], outputInsMem[20:16], outputmux, OTB4RegRlTE, MuxBR, inputRD1, inputRD2);
106
107    alu_ctrl aluCtrl1(outputInsMem[8:0], OTB2ALUDP, ALUctr1ALU);
108
109    insmem insm1(PcInsMemAdd, inputInsMem);
110
111    Suma_4b sumador1(PcInsMemAdd, 32'd4, inputAdd);
112
113    Suma_4b sumador2(outputAddB2, SL2Add, inputAddB3);
114
115    pc pc(NxPC, clk, PcInsMemAdd);
116
117    ALU operator(outputRD1, MuxALU, ALUctr1ALU, inputAddrst, z_flag);
118
119    mem_x memory(outputAddrst, outputRD2B3, OTB3MemRlTE, OTB3MemRead, inputDatmem);
120
121    mux1 mux1(OTB2RegDst, outputRr2, outputRr2, inputmux);
122
123    mux2 mux2(PCSrc, outputAddB3, inputAdd, NxPC);
124
125    mux3 mux3(OTB4ALUSrc, outputSinex, outputRD2, MuxALU);
126
127    mux2 mux4(OTB4MemREG, outputDatmem, outputAddrstB4, MuxBR);
128
129    Muxd_andy(OTB3Branch, output_z_flag, PCSrc);
130
131    shiftl shiftl(outputSinex, SL2Add);
132
133    signextend sign(signExtendInMem[15:0], inputSinex);
134
135 endmodule

```

Imagen 2.14

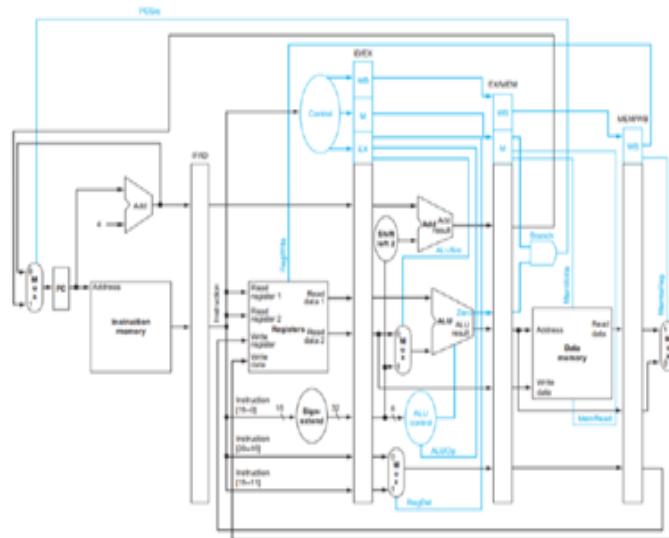


Imagen 2.15



Memoria de instrucciones		Banco de registros																																																																									
 The screenshot shows a memory dump window titled "Memory Data - /tbsingled/datapath/memory/memory - Default". It displays a grid of 17 rows and 2 columns. The first column contains addresses from 0 to 17, and the second column contains binary values. Addresses 0, 1, 2, 3, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, and 16 have 'x' values. Addresses 4 and 10 have '0' values. Address 17 has a double-dot value. <table border="1"><tr><td>0</td><td>x</td></tr><tr><td>1</td><td>0</td></tr><tr><td>2</td><td>x</td></tr><tr><td>3</td><td>x</td></tr><tr><td>4</td><td>0</td></tr><tr><td>5</td><td>x</td></tr><tr><td>6</td><td>x</td></tr><tr><td>7</td><td>x</td></tr><tr><td>8</td><td>x</td></tr><tr><td>9</td><td>x</td></tr><tr><td>10</td><td>0</td></tr><tr><td>11</td><td>x</td></tr><tr><td>12</td><td>x</td></tr><tr><td>13</td><td>x</td></tr><tr><td>14</td><td>x</td></tr><tr><td>15</td><td>x</td></tr><tr><td>16</td><td>x</td></tr><tr><td>17</td><td>..</td></tr></table>		0	x	1	0	2	x	3	x	4	0	5	x	6	x	7	x	8	x	9	x	10	0	11	x	12	x	13	x	14	x	15	x	16	x	17	..	 The screenshot shows a memory dump window titled "Memory Data - /tbsingled/datapath/regb/memory - Default". It displays a grid of 17 rows and 2 columns. Both columns contain binary values from 0 to 17 sequentially. <table border="1"><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr><tr><td>2</td><td>2</td></tr><tr><td>3</td><td>3</td></tr><tr><td>4</td><td>4</td></tr><tr><td>5</td><td>5</td></tr><tr><td>6</td><td>6</td></tr><tr><td>7</td><td>7</td></tr><tr><td>8</td><td>8</td></tr><tr><td>9</td><td>9</td></tr><tr><td>10</td><td>10</td></tr><tr><td>11</td><td>11</td></tr><tr><td>12</td><td>12</td></tr><tr><td>13</td><td>13</td></tr><tr><td>14</td><td>14</td></tr><tr><td>15</td><td>15</td></tr><tr><td>16</td><td>16</td></tr><tr><td>17</td><td>17</td></tr></table>		0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	11	11	12	12	13	13	14	14	15	15	16	16	17	17
0	x																																																																										
1	0																																																																										
2	x																																																																										
3	x																																																																										
4	0																																																																										
5	x																																																																										
6	x																																																																										
7	x																																																																										
8	x																																																																										
9	x																																																																										
10	0																																																																										
11	x																																																																										
12	x																																																																										
13	x																																																																										
14	x																																																																										
15	x																																																																										
16	x																																																																										
17	..																																																																										
0	0																																																																										
1	1																																																																										
2	2																																																																										
3	3																																																																										
4	4																																																																										
5	5																																																																										
6	6																																																																										
7	7																																																																										
8	8																																																																										
9	9																																																																										
10	10																																																																										
11	11																																																																										
12	12																																																																										
13	13																																																																										
14	14																																																																										
15	15																																																																										
16	16																																																																										
17	17																																																																										

Tabla 2.7

Modificaciones:

- **ALU**

Se agregaron las operaciones correspondientes a la fase 2, en este caso las de tipo I solicitadas (LW, SW, BEQ, ADDI, ANDI, ORI)

- **ALU control**

Se agregaron las opciones para ejecutar las operaciones correspondientes a la fase 2, en este caso las de tipo I solicitadas (LW, SW, BEQ, ADDI, ANDI, ORI)

- **Unidad de control**

Se agregaron las opciones para ejecutar las operaciones correspondientes a la fase 2, en este caso las de tipo I solicitadas (LW, SW, BEQ, ADDI, ANDI, ORI) con sus respectivas salidas hacia buffers.

- **Buffers**

Se elaboraron 4 buffers, con la descripción IF/ID, ID/EX, EX/MEM y MEM/WB, los cuales están en ese orden y en un cierto acomodo de cables para llevar a cabo la



técnica conocida como “pipeline” que nos ayuda a optimizar tiempos en relación a las operaciones a realizar.

- **DataPath**

Se organizaron de nuevo los cables conectando buffers y módulos entre sí.

Modificaciones Fase 1 y 2

The screenshot displays four windows from a simulation environment:

- Banco de registros:** Shows a memory dump of a register file. The data is as follows:

0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	10
9	1
10	0
11	10
12	10
13	13
14	14
15	15

- Memoria de instrucciones:** Shows a memory dump of the instruction memory. The data is as follows:

0	10
1	x
2	x
3	x
4	x
5	x
6	x
7	x
8	x
9	x
10	x
11	x
12	x
13	x
14	x
15	v

Reescribimos de forma correcta la instrucción para la "LW"
-Antes 11000100
-Ahora 10001100

TestF2_MemInst.mem shows the memory dump for the LW reimplementation. The data is as follows:

37	00000000
38	00000000
39	00000000
40	00000000
41	10101100
42	00001000
43	00000000
44	00000000
45	00000000
46	00000000
47	00000000
48	00000000
49	00000000
50	00000000
51	00000000
52	00000000
53	10001100
54	00001100
55	00000000

Waveform Analysis: Shows the signal `tbsingled/clk` over time, with a period of approximately 100 ns. The signal alternates between high and low levels.

Tabla 2.8



9. Desarrollo Fase 3

MODIFICACIONES:

Los cambios realizados para esta fase 3 del proyecto final son los siguientes:

- Unidad de control:

Se modificó la unidad de control agregando la salida Jump modificando así las entradas y salidas de los buffers, también para cada operación de tipo I y tipo R a este valor llamado Jump se le daría un 0 y un 1 cuando fuera Jump.

```
unit_control.v | timescale 1ns/1ns
module unit_control(
    input[5:0] OPCODE,
    output reg MemREG, RegWRITE, MemWRITE,
    output reg Branch, MemRead, ALUSrc, RegDst, Jump,
    output reg [2:0] ALUOP
);
always @* begin
    case (OPCODE)
        6'b000000: begin //TIPO R
            MemREG = 0;
            RegWRITE = 1;
            MemWRITE = 0;
            Branch = 0;
            MemRead = 0;
            ALUSrc = 0;
            RegDst = 1;
            Jump = 0;
            ALUOP = 3'b010;
        end
        6'b000010: begin //J
            MemREG = 1'bx;
            RegWRITE = 1'bx;
            MemWRITE = 1'bx;
            Branch = 1'bx;
            MemRead = 1'bx;
            ALUSrc = 1'bx;
            RegDst = 1'bx;
            Jump = 1; //Ejecuta el jump
            ALUOP = 3'bxx;
        end
    endcase
end
```

Tabla 3.4

- Shift left 2 para Jump:

Se hizo la creación de un módulo llamado Shift left dos que era específicamente para las instrucciones de tipo J, en el cual se toman 26 bits de la instrucción del 25 al 0 y dentro no se hace como tal un corrimiento de 2 bits sino que se agregan dos ceros a la derecha dando así una salida de 28 bits.



```
|> Shift_left_2_JUMP.v x DatapathX.v
1 `timescale 1ns/1ns
2
3 module SL2_J(
4     input [25:0] X,
5     output [27:0] Y
6 );
7
8
9 assign Y = {X, 1'b0, 1'b0};
10
11 endmodule
12
```

Imagen 3.3

- Concatenación:

Para la concatenación no supe cómo realizarlo por cable y decidí crear un módulo, tomando como entradas la salida del Shift left 2 creado para las instrucciones de tipo J y la salida de la suma de las instrucciones más cuatro, tomando los bits del 31 al 28 de esta última y colocándolos a la izquierda de la primera entrada, dando así una salida de 32 bits que viajará por todos los buffers hasta un multiplexor del que hablaremos más adelante.

```
|> Concatenacion_Jump.v x DatapathX.v
1 `timescale 1ns/1ns
2
3 module conca_Jump(
4     input [27:0] X,
5     input [31:0] Y,
6     output [31:0] Z
7 );
8
9
10 assign Z = {Y[31:28], X};
11
12 endmodule
13
14
```

Imagen 3.4



- Nuevo multiplexor:

Por último se creó un multiplexor nuevo, que realmente se crea mentalmente porque son instancias, que iría entre el multiplexor que conectaba antes directamente al PC, las entradas de este serían, el enable “Jump” que viaja a través de los buffers 2, 3 y 4, la salida de 32 bits que se hizo en la concatenación (que viaja a través de los buffers 2, 3 y 4) y la salida del multiplexor que antes iba directo al PC, y el resultado de esto se conecta al PC.

- Buffers:

Solamente para los buffers se les agregaron nuevas entradas del jump y sus respectivas salidas tal como en la imagen (esto es para los buffers 2, 3 y 4).

```
1 | timescale 1ns/1ns
2 | //1.Creacion del modulo
3 | //Declaracion del modulo con sus entradas y salidas
4 | module buffer2
5 | (
6 |   //UNIDAD DE CONTROL
7 |   //ENTRADAS
8 |   //WB
9 |   input INB2MemREG,
10 |  input INB2RegWRITE,
11 |  //M
12 |  input INB2Branch,
13 |  input INB2MemNRITE,
14 |  input INB2MemRead,
15 |  //EX
16 |  input INB2RegDst,
17 |  input [2:0]INB2ALUOP,
18 |  input INB2ALUSrc,
19 |  //J
20 |  input [31:0]INB2DirJump,
21 |  input INB2Jump,
22 |  //RESTO
23 |  input clk,
24 |  input [31:0] inputAddB2,
25 |  input [31:0] inputRD1,
26 |  input [31:0] inputRD2,
27 |  input [31:0] inputSinex,
28 |  input [4:0] inputRr1,
29 |  input [4:0] inputRr2,
30 |  //UNIDAD DE CONTROL
31 |  //SALIDAS
```

Imagen 3.5

- Modificación de la ALU:

Se le agregó la nueva instrucción “XOR” y se cambió el tamaño del selector y de OP, y respectivamente de los cables, porque antes eran de 3 bits pero ahora son de 4.



```
1 `timescale 1ns/1ns
2
3 module ALU(
4     input [31:0] X,
5     input [31:0] Y,
6     input [3:0] SEL,
7     output reg [31:0] R,
8     output reg Z_flag
9 );
10
11 always@* begin
12     case(SEL)
13         //R operando destino, X operando fuente 1,Y operando fuente 2
14         4'd0: R = X + Y; //ADD ADDI LW SW
15         4'd1: R = X - Y; //SUB BEQ
16         4'd2: R = X & Y; //AND ANDI
17         4'd3: R = X | Y; //OR ORI
18         4'd4: R = X < Y; //SLT SLTI
19         4'd5: R = X * Y; //Mul
20         4'd6: R = X / Y; //Div
21         4'd7: R = X << 0; //NOP
22         4'd8: R = X ^ Y; //XOR
23         default: R <= 32'bx;
24     endcase
25
26     Z_flag <=(R) ? 0:1;
27
28 end
29 endmodule
```

Imagen 3.6

- Datapath:

Y por último en el datapath lo único que fue de cambios es en la implementación de los nuevos módulos por que se crearon nuevas entradas y salidas, se conecto todo como se debería y listo.

```
145
146     SL2_J jump(OutputInsMem[25:0],SL2_JConca);
147
148     conca_Jump jump1(SL2_JConca,outputAdd,INB2DirJump);
149
150     mux2_jumpPC(OTB4Jump,OTB4DirJump,MuxMux,MuxPc);
151
152 endmodule
153
```

Imagen 3.7

- Testbench:

Modificación de tiempos del testbench para dar paso a todas las instrucciones.



```
▶ ▶ tbsingled.v x Datapath.X.v
1 `timescale 1ns/1ns
2
3 module tbsingled();
4     reg clk;
5
6     singledpR datapath(clk);
7
8     always
9
10    #50 clk = ~clk;
11
12     initial
13     begin
14         clk <= 0;
15         #10000
16         $stop;
17
18     end
19
20 endmodule
```

Imagen 3.8

- Archivo de memoria de instrucciones:

En la memoria de instrucciones el cambio que se hizo fue mover de lugar la primera instrucción "JUMP" hasta el final, porque en su posición inicial solo ciclaba una suma inmediata.

Archivo Edición Form
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00000000
00001000
00000000
00000000
00000000|
0
0

Imagen 3.9

Comentario:



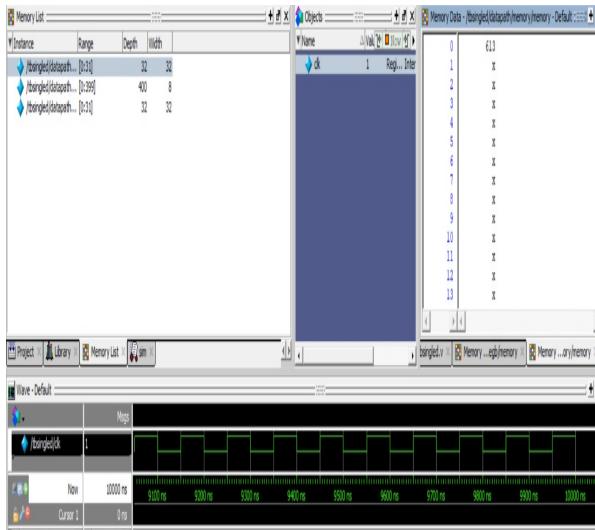
Para implementar nuestro intento de código o lenguaje ensamblador en este caso “CRC” que es para identificar errores, lo que teníamos que hacer es eliminar una instrucción y agregar la instrucción “XOR” La cual se basa para hacer las divisiones.

```
1 `timescale 1ns/1ns
2
3 module alu_ctrl(
4     input[5:0] FUNCTION,
5     input[2:0] ALUOP,
6     output reg [3:0] OP
7 );
8
9     always @(*) begin
10         case (ALUOP)
11             3'b010: begin
12                 case (FUNCTION)
13                     6'b100000: OP = 4'd0; //ADD
14                     6'b100010: OP = 4'd1; //SUB
15                     6'b100100: OP = 4'd2; //AND
16                     6'b100101: OP = 4'd3; //OR
17                     6'b101010: OP = 4'd4; //slt
18                     6'b011000: OP = 4'd5; //mul
19                     6'b011010: OP = 4'd6; //div
20                     6'b000000: OP = 4'd7; //nop
21                     6'b100110: OP = 4'd8; //XOR
22                 endcase
23             end
24             3'b000: OP = 4'd0; //addi...
25             3'b001: OP = 4'd4; //slti...
26             3'b100: OP = 4'd2; //andi...
27             3'b011: OP = 4'd3; //ori...
28         endcase
29     end
30 endmodule
```

Imagen 3.10



TEST BENCH



BANCO DE REGISTROS

0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
613	

MEMORIA DE INSTRUCCIONES

0	00000000 00000000 00000000 00000000
4	00100000 00001000 00000101 00111001
8	00000000 00000000 00000000 00000000
12	00000000 00000000 00000000 00000000
16	00000000 00000000 00000000 00000000
20	00100001 00001000 11111101 00101100
24	00000000 00000000 00000000 00000000
28	00001000 00000000 00000000 00010010
32	00000000 00000000 00000000 00000000
36	00000000 00000000 00000000 00000000
40	00000000 00000000 00000000 00000000
44	00100001 00001000 11111111 11111001
48	00000000 00000000 00000000 00000000
52	00001000 00000000 00000000 00000110

MEMORIA

0	613
1	x
2	x
3	x
4	x
5	x
6	x
7	x
8	x
9	x
10	x
11	x
12	x
13	x

Tabla 3.5

Comentario:

En la fase 2 había una imagen para saber si todo lo que se guardaba y mostraba era correcto y en qué dirección deberían de ir los resultados. En esta fase no había ninguna imagen que nos diera a entender donde deberían de quedar los resultados o que resultados serían.

Después de consultarla con el profe nos dimos cuenta que había un error en el código binario del profe que teníamos que encontrar y después lo reparamos y el código ya va como debería de correr.



10. Conclusión fase 1

Manuel

Mi participación en esta primera fase del proyecto no fue la que hubiera deseado, tuve muchos problemas en la investigación, es un campo que no manejo, que se me complica demasiado, sin embargo hice mi mayor esfuerzo para poder sacarlo adelante. En los primeros días me dispuse a leer el libro de la clase en las secciones que se encontraba la fase uno, que era a partir de la sección 4, pero en cuanto llegó la parte de investigar la forma de programar para la propuesta del programa ensamblador comenzó mi pesadilla. Tardé más de dos días en comprender el tema, busqué ejemplos, pero no sé si no busqué en el lugar correcto o qué fue lo que pasó, pero no terminé de entender. Por otro lado, me siento orgulloso de haber podido ayudar a Diego en su parte, ya que entre todo el equipo en la única reunión que tuvimos realizamos por completo el modulo “Single Datapath”, interconectando todos los módulos interiores, también fui quién logró encontrar el error en el archivo que nos proporcionaba el maestro y eso me hizo sentir mejor, aunque sé que no pude solucionar por completo el problema del programa ensamblador, ayudé en otros aspectos. Estoy muy contento con mi equipo, buscamos apoyarnos y trabajar juntos, espero que mi desempeño sea mejor en la siguiente fase.

Athziri

Fue un gran reto para mi poder encontrar información viable y confiable para el reporte, ya que de esto depende un poco el código, debido a que lo tomaríamos ligeramente como guía o pseudocódigo. La información que se presenta aquí debía ser clara para que se entendiera de lo que está conformado cada módulo y así Diego pudiera hacer el código y comprender mejor de qué trata y no solo hacerlo como tal. Me siento satisfecha porque siento que mi investigación aportó mucho a la comprensión del funcionamiento de algunos códigos. La verdad me gusto mucho hacerlo a pesar de las dificultades presentadas de este reporte porque también aprendí más sobre cada uno de los módulos y repasé algunos temas ya vistos en clases pasadas. Considero que me esforcé en la investigación y también me gustó que mi equipo me diera su aprobación de cada documento que consultaba, de ahí también tomaba en cuenta la información que pondría y si la entienden o no. Fue un buen trabajo en equipo porque todos aportaron su conocimiento y solidaridad, espero que todo siga así para mantener el equilibrio del equipo.

Diego

La Fase 1 del proyecto tuvo de todo, felicidad, desesperación, trabajo en equipo y tristeza. A mi me toco la parte del Código que creo que a diferencia de mis compañeros es lo más “divertido” por hacer cuando lo logras entender, comencé el Lunes y al ver lo que pedía no sabía que iba hacer ni cómo empezar, después agregué



todos los módulos que veía y agregue los tres nuevos módulos (PC, SignExtend, ShiftLeft), me intereso mucho el hecho de que al final con tantos errores y leyendo logre masomenos entender qué es esto y cómo funcionan algunos módulos de este SingleDatapath, Las cosas que tal vez hubiera visto mejor para la fase 1 es que tal vez no queda tan claro para nosotros los principiantes en esto el dibujo de el modulo grande, Hay muchas cosas que no sabía si eran 4, 5 o 32 bits y el non-blocking y cosas asi, pero con ayuda de los compañeros pues logré entender cómo iban las cosas, el tiempo tal vez sea algo corto por el hecho de que no podemos dedicar todo el tiempo haciendo este proyecto cuando tenemos más clases y nos retacan de más información pero se logro, mis compañeros estuvieron individualmente pero siempre aclarando dudas y nos juntamos para adelantar mucho en las dudas que había, Sobre el github creo que tuvo que haberse mostrado o explicado desde unas clases antes por que aun no se ni como subir mis avances para que los vea pero ahí están, fue realmente muy interesante la parte del código cuando logras entenderlo. cuando no es estresante.



11. Conclusión fase 2

Manuel

Soy de las personas que usualmente toman un fracaso de forma no tan positiva, sin embargo, las equivocaciones y adversidades que presentamos en la fase 1 nos hicieron más fuertes como equipo, ya que unimos fuerzas, no dejamos dividido el trabajo, comenzamos a realizarlo juntos, demostrando lo que es el verdadero trabajo en equipo. Arreglar el código de verilog fue todo un reto, ya que no solo era dejarlo listo para las instrucciones de esta fase, si no, que en nuestro caso era como comenzar desde cero, dispusimos varias horas de trabajo para dejarlo listo, enfrentamos muchos problemas en el camino, desde “warnings” inexistentes, hasta la conexión de buffers. En esta fase me concentré mucho en desarrollar y comprender el código, ya que para la siguiente fase me tocará a mí. Considero que esta fase 2, es como un parteaguas para mi equipo, nos unió y nos hizo aprender mucho, aunque tengo la creencia que hay otras formas de aprendizaje más efectivas que lanzarse con los ojos vendados y tus propias armas a una batalla que desconoces, espero y confío en que nos vaya mejor que en las actividades pasadas y logremos un mejor desarrollo.

Athziri

En esta fase 2 me surgieron muchas dificultades, ya que estoy un poco confundida.

La creación de buffer fue un gran reto para mí porque la verdad no sabía realmente cómo crear los módulos, conforme fui avanzando más errores y más dudas me surgieron, investigué a más no poder hasta lograr realizar cada buffer, espero poder comprenderlo mejor con la siguiente fase. Lo que más problemas me causó fue la ALU y ALU control debido a que no sabía los operadores para las nuevas instrucciones, así que eso fue de mayor dificultad para mí y en el caso de la ALU control investigar sobre el Opcode/Function fue un gran desafío porque no encontraba mucha información, porque no sabía el nombre de esos números, pero gracias a esta investigación ya puedo reconocer este nuevo concepto. Ahora es importante mencionar que al cargar el TestF2_MemInst.mem me fue imposible ya que no puedo lograr que se vea bien la simulación y los valores en los registros de la memoria solo salen x, pero este problema se debió a que no descargue bien el documento y hubo cambios. Para las conexiones en mi top level se me dificultó mucho así que pedí ayuda a mi equipo con la elaboración de este módulo, puede decir que fue un buen trabajo en equipo para esta área porque se complementó todo con el conocimiento de Manuel y la buena observación de Diego, fue menos tedioso y nos divertimos un poco porque ya se entendió más que es lo que se quería lograr.

Diego



Sobre esta fase creo que base a la presentación solo modificar cosas fue lo más sencillo, Tiempo hubo mucho pero creo que dudas mas, de mi parte en el lenguaje ensamblador es algo que yo nunca en mi vida había visto y creo que hubiera preferido mucho que en las clases se hubiera demostrado cómo programar en ensamblador por que lo que nosotros estamos haciendo es investigado y no sabemos muy bien qué es lo que estamos haciendo, Creí que la fase 2 sería más sencilla ya que creí que agregaremos nada mas unos detalles pero sí se extendió muchísimo más de lo esperado, Espero que la fase 3 si sea mucho más sencillo que está para mejorar nuestra calificación, El código en verilog fue lo que nos dio tantas fallas que se extendió mucho y nos quedamos sin tiempo para especificarnos más en el lenguaje ensamblador, Había muy poco tiempo para juntamos entre los 3 casi nulo así que fue más difícil aún porque entre nosotros nos ayudamos en cosas que no sabemos, de ahí en mas creo que lo más necesario hubiera sido saber como programar bien en lenguaje ensamblador pero creo que lo que se hizo estuvo bien.



12. Conclusión fase 3

Manuel:

Aunque como equipo nos esforzamos mucho, sabemos que seguimos cometiendo errores. Hemos trabajado arduamente a lo largo de estas 3 fases pero nunca logramos definir bien lo que era nuestro código ensamblador, semana tras semana corrímos con el problema de que como no sabíamos bien y no contábamos con el conocimiento previo adecuado no sabíamos cómo definir uno sobre el cual trabajar, y al estar atrasados en este rubro ya no nos quedaban muchos programas por los cuales apelar debido a su grado de dificultad, es decir, los más sencillos ya no los habían ganado, una semana teníamos uno, a la siguiente semana teníamos otro y al final no lo terminaron cambiando en la última semana por el "CRC" que es un código para detección de errores, pero que no supimos implementar. Por parte del código de verilog fue muy sencillo implementarlo, ya que con todo el conocimiento previo que teníamos no era más que aplicarlo, sin embargo la mayor traba que vimos fue el código ensamblador no supimos definirlo y como cada fase está dividida por encargados todos teníamos algo que hacer en esta última sección, y algo en que enfocarnos específicamente, y es por eso que no pudimos reunirnos para poder solucionar nuestra situación, por más esfuerzo que pusieron a veces hay cosas que no nos dejan y son externas a nosotros, no me voy muy contento en esta última fase, ya que me hubiera gustado tener un desempeño diferente, sin embargo creo que el puntaje que obtengamos lo merecemos.

Athziri:

Considero que esta es una de las fases más complicadas, porque en esta ocasión en el algoritmo CRC y el lenguaje máquina nos fue muy complicado de implementarlo, ya que, tenemos muy poco conocimiento sobre la implementación de CRC para nuestro modelo, en Internet sólo salían algunos códigos ya hechos pero ninguno nos servía para podernos basar. La parte que me corresponde entendí que es el CRC, comprendí la manera en que funciona y saber si un dato es correcto o no, pero la verdad no logré comprender del todo en cómo implementarlo.

Fue muy complicado también porque fue un cambio que no esperábamos en el algoritmo porque ya teníamos uno en mente, la verdad parecía sencillo ese algoritmo, pero la idea era complicarnos más, se logró, la verdad si estuvo muy complicado, pero al final solo pudimos implementar solo una parte de nuestro algoritmo en lenguaje máquina. Fue muy abrumador hacer esta parte de la fase 3, me siento mal conmigo misma por no poderlo hacer del todo bien y no obtener respuesta a su implementación correcta me hace sentir una gran frustración ya que es la fase final y no se logró.

Espero que más adelante pueda comprender un poco más sobre el algoritmo y su implantación acorde a mi modelo "Datapath".



Por último quiero agregar que la parte de ser Admin fue fácil porque mi equipo siempre trabaja excelente y no tengo que estar detrás de ellos para que trabajen, todas estas fases hemos trabajado como un gran equipo y me da gusto haberlos tenido conmigo para este proyecto, nunca nos dejamos solos y algunos problemas que surgían los arreglamos en equipo, también hubo mucha apoyo emocional y eso es algo muy importante para no dejar caer uno al otro, todos vamos de la mano, ayudándonos entre sí.

Diego:

Realmente me gusto trabajar con este equipo, creo que concordamos muy bien a pesar de ni si quiera poder conectarnos más que 3 veces, cada quien tenía sus problemas exteriores y lo bonito es que ahí estamos para ayudarnos todos, sea la calificación que tengamos creo que estoy satisfecho con el desempeño que tuvimos a pesar de tanta dificultad que se nos produjo, Sobre la fase 3 creo que la parte del reporte es algo más sencillo, Realmente no fue difícil la fase 3 ni en el código ni en el reporte, creo que lo que más se complicó fue la parte del ensamblador, nosotros sin tener conocimiento fuimos el equipo con más complicaciones en ese tema por falta de tiempo, se nos apoyo usando un algoritmo fácil, después ya no, después sí y al final terminamos por “implementar” el nuevo algoritmo CRC, por falta de tiempo más que nada tuvimos solo una semana para investigar sobre ese algoritmo y se entiende que es lo que hace pero no se sabe cómo implementarlo al código, Por no lograr lo esperado también faltó en el reporte cosas pero creo que logramos mucho y aprendimos mucho con este proyecto final, el profesor dio lo que tenía a las manos para ayudarnos pero nosotros de aprendices nos faltó más facilidad al comprenderlo. Después de todo creo que nos irá bien aun asi.

13. Referencias

Referencias Fase 1

- <http://ocw.uc3m.es/cursos-archivados/arquitectura-de-ordenadores/lecturas/html/asm.html#asm:fig:ejemplo>
- <http://ocw.uc3m.es/ingenieria-informatica/estructura-de-computadores/ejercicios-resueltos/ejercicios-resueltos-tema-3-v4.pdf>
- https://www.infor.uva.es/~bastida/OC/TRABAJO1_MIPS.pdf
- http://www.hpca.ual.es/~vruiz/docencia/laboratorio_arquitectura/proyectos/00-01/RoceroBlanes/exe/Doc_HTML/docu.html

Libro de clase

<https://drive.google.com/file/d/1JC6DHZF7tgkSR9VvVGRDaA7oZnv6sVtN/view>

Referencias Fase 2

- Profesores.elo. (2012). Procesador MIPS. Recuperado el 02 de Junio del 2021 de http://profesores.elo.utfsm.cl/~tarredondo/info/comp-architecture/paralelo2/C03_MIPS.pdf
- Documento online sobre los tipos de instrucciones (2011). MIPS. Recuperado el día 02 de junio de 2021 de <http://www.fdi.ucm.es/profesor/jiruz/ec-is/temas/Tema%205%20-%20Repaso%20ruta%20de%20datos.pdf>

Libro de clase

<https://drive.google.com/file/d/1JC6DHZF7tgkSR9VvVGRDaA7oZnv6sVtN/view>

Referencias Fase 3

- Documento online sobre los tipos de instrucciones (2011). MIPS. Recuperado el día 02 de junio de 2021 de <http://www.fdi.ucm.es/profesor/jiruz/ec-is/temas/Tema%205%20-%20Repaso%20ruta%20de%20datos.pdf>

Libro de clase

<https://drive.google.com/file/d/1JC6DHZF7tgkSR9VvVGRDaA7oZnv6sVtN/view>

Evidencia de reuniones

Fase 1

- <https://drive.google.com/file/d/1IRbFXDOvbHinygvWsq1p3nRs02uxlmu/view?usp=drivesdk>
- https://drive.google.com/file/d/1DfejVf4Xk2Sn4Cmyjw_Hc5NKZJulwLyp/view?usp=drivesdk
- <https://drive.google.com/file/d/1RdrdOPbYSD092rNYEUBqbcvyA9SloAAT/view?usp=drivesdk>

Fase 2



Fase 3

"En esta ocasión no logramos conectarnos mas que un rato después de las clases, estamos en tiempos de exámenes y proyectos entonces es muy difícil dedicar tiempo a una sola cosa"