



## **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 y 2>>**

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

**García Ramírez José Manuel**

**Rubio Andrade Athziri Magdalena>>**

**Profesor: López Arce Delgado Jorge Ernesto**

**Fecha: <<Jueves 10 Junio 2021 >>**

## 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 procesado.

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 maquina a través del uso de la segmentacion(proceso paralelo consistente en el procesador simultaneo de instrucciones en diferente fase).

En 1984 Hennessy deajo 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

Comentaré 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 es capaz 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

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



## Error TestF1\_MemInst

Original	Corrección
00000000	00000000
00100010	00100010
10100000	10100000
00000000	00100000 -- ADD
00000000	00000000
10100100	10100100
10101000	10101000
00100010	00100010 -- SUB
00000001	00000001
00000100	00000100
11000000	11000000
00100100	00100100 -- AND
00000001	00000001
01001010	01001010
11001000	11001000
00100101	00100101 -- OR
00000000	00000000
10101010	10101010
11010000	11010000
00100101	00100101 -- SLT
00000000	00000000
00000000	00000000
00000000	00000000
00000000	00000000 -- NOP

## 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
```

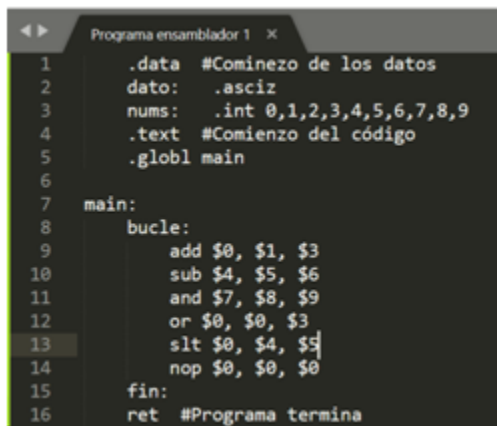
```
or $0, $0, $3
```

```
slt $0, $4, $5
```

```
nop $0, $0, $0
```

fin:

ret #Programa termina



```
1 .data #Comienzo 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
```

**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.

## Objetivos

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.

## **Desarrollo**

### ***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 triestado 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 estado 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 leebus 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 MIPS<sub>m</sub>.

### ***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.

## **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 medio 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 repase 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

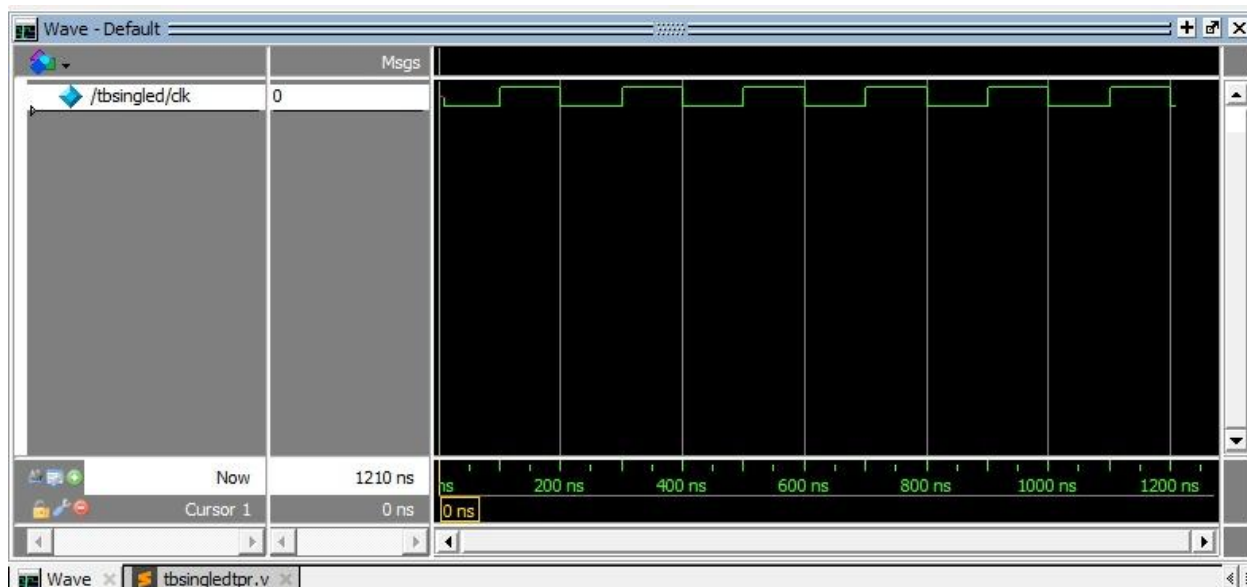
La Fase 1 del proyecto tuvo de todo, felicidad, desesperación, trabajo en equipo y tristeza. a mi me toco la parte del Código que creo que a diferencia de mis compañeros es lo más “divertido” por hacer cuando lo logras entender, comencé el Lunes y al ver lo que pedía no sabía que iba hacer ni como empezar,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

## Referencias

- <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](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](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>

## Test Bench



## Memoria

Memory Data - /tbsingled/datapath/regb/memory - Default		
0	0	1
2	2	3
4	4	5
6	6	7
8	8	9
10	10	11
12	12	13
14	14	15
16	16	17
18	18	19
20	3	1
22	22	23
24	0	10
26	1	27
28	28	29

## Memoria de instrucciones

Memory Data - /tbsingled/datapath/instr/instrMemory - Default				
0	00000000	00100010	10100000	00100000
4	00000000	10100100	10101000	00100010
8	00000001	00000100	11000000	00100100
12	00000001	01001010	11001000	00100101
16	00000000	10101010	11010000	00101010
20	00000000	00000000	00000000	00000000
24	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
28	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
32	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
36	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
40	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
44	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
48	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX
52	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

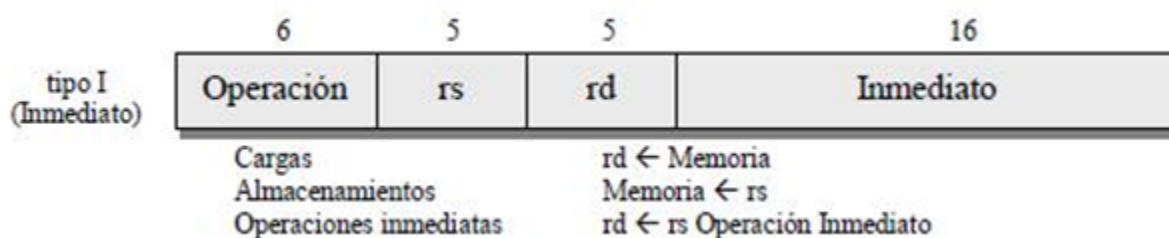
## Introducción fase 2

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, Fifth 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.

### Instrucciones de tipo "I":

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:



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.

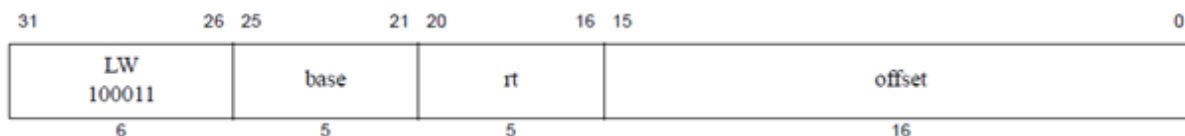
- Bits 15-0:

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:



❖ 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

--	--

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

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 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



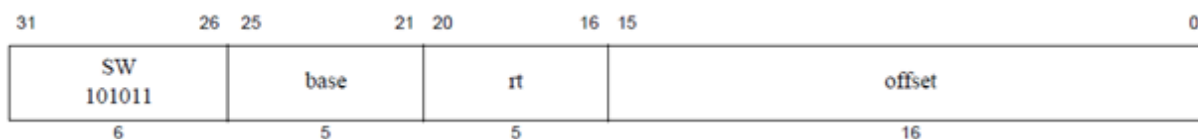


- ❖ base: 00001
- ❖ rt: 00100
- ❖ offset: 00000000000000011

Por lo tanto el código sería: 10001100001001000000000000000011

- • SW:

- ❖ Formato:



- ❖ 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												
<table><tr><th>Dirección</th><th>Dato</th></tr><tr><td>\$0</td><td>3</td></tr><tr><td>\$1</td><td>0</td></tr></table>	Dirección	Dato	\$0	3	\$1	0	<table><tr><th>Dirección</th><th>Dato</th></tr><tr><td>\$0</td><td>4</td></tr><tr><td>\$1</td><td>33</td></tr></table>	Dirección	Dato	\$0	4	\$1	33
Dirección	Dato												
\$0	3												
\$1	0												
Dirección	Dato												
\$0	4												
\$1	33												

\$2	1	\$2	12
\$3	6	\$3	123
\$4	9	\$4	6

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

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:



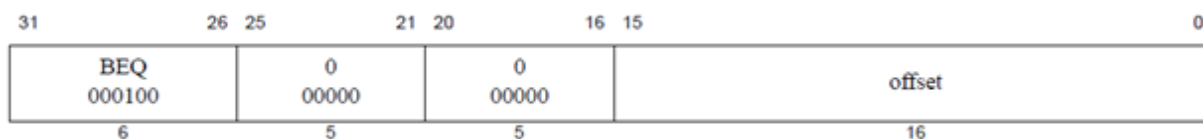
Por lo tanto nuestro offset en binario sería: 0000 0000 0000 0011

CREA

- ❖ SW: 101011
- ❖ base: 00001
- ❖ rt: 00100
- ❖ offset: 00000000000000011

Por lo tanto el código sería: 10101100001001000000000000000011

- BEQ(Branch on Equal):  
  - ❖ Formato:



BEQ rs, rt, offset

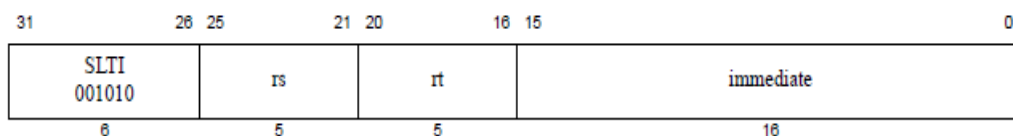
- ❖ Descripción:

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:



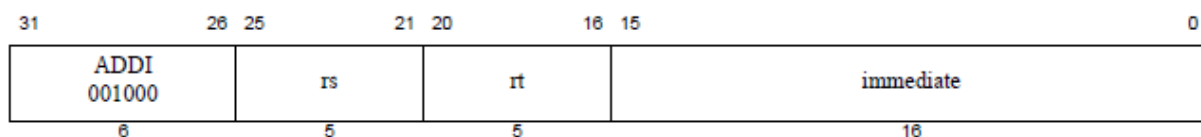
SLTI rt, rs, immediate

- ❖ 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:



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.

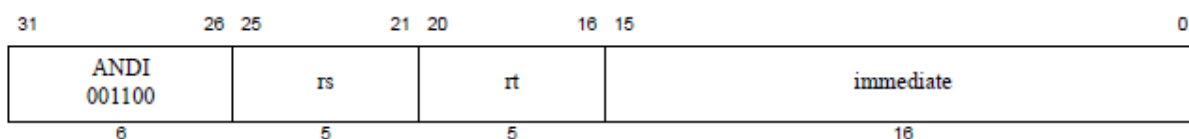
- Si la suma da como resultado un desbordamiento aritmético del complemento 2 de 32 bits, el registro de destino no se modifica y

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:

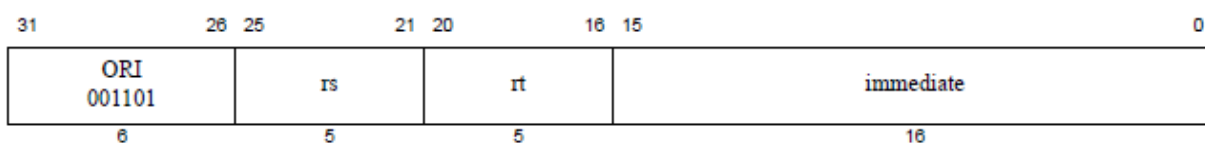


ANDI rt, rs, immediate

❖ 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:



ORI rt, rs, immediate

❖ Descripción:

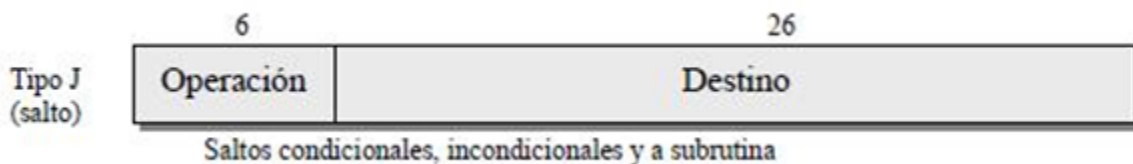
GPR [rt] GPR [rs] o inmediato

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 "J":*

El formato tipo J es utilizado para las instrucciones de bifurcación. Su formato se representa de la siguiente manera:



Como podemos observar se divide en dos secciones:

- Bits 31-26:

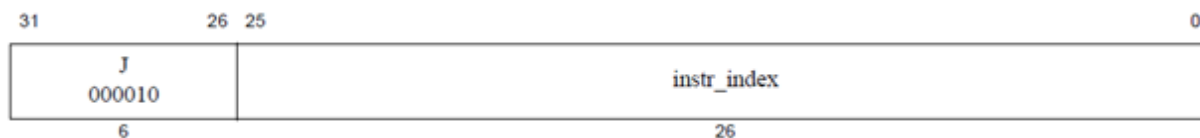
En estos primeros seis bits encontramos el espacio para el código de operación que identifica a cada instrucción, en este caso por ser el tipo de instrucción J será el código "000010".

- Bits 25-0:

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

La instrucción de tipo J que se van a analizar en esta fase 2, aunque no se ejecutará es:

- J:
  - ❖ Formato:



### Descripción:

También llamadas instrucciones de salto condicional, una bifurcación se puede ver como un salto incondicional, es decir, la instrucción obliga a la máquina a seguir siempre el salto. Para seguir entre saltos condicionales e incondicionales, el nombre MIPS para este tipo de instrucciones es jump.

### Tablas de instrucciones:

Tabla 1

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

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)

Benl	I	bnel \$rs \$rt #offset(base)
Xor	R	xor \$rd \$rs \$rt

### *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”.

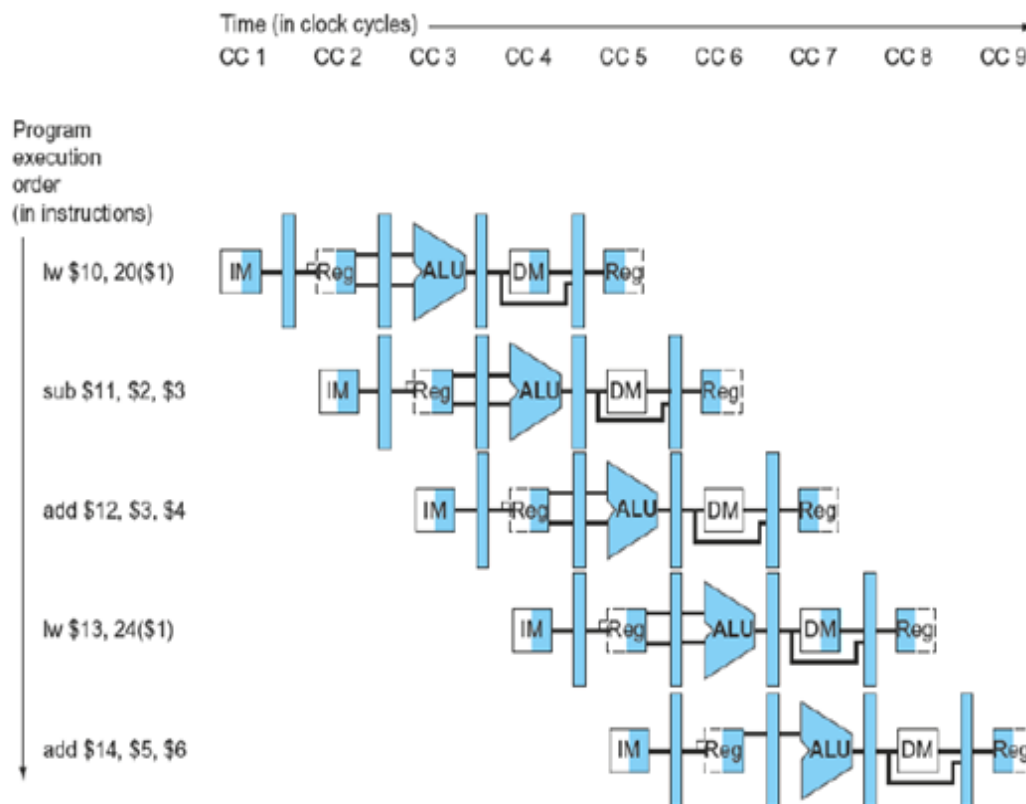
- Memory:

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, Fifth 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.

## Objetivos



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 1 y tabla 2.

## Desarrollo

- **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:  $R = X \mid Y$ ; //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,

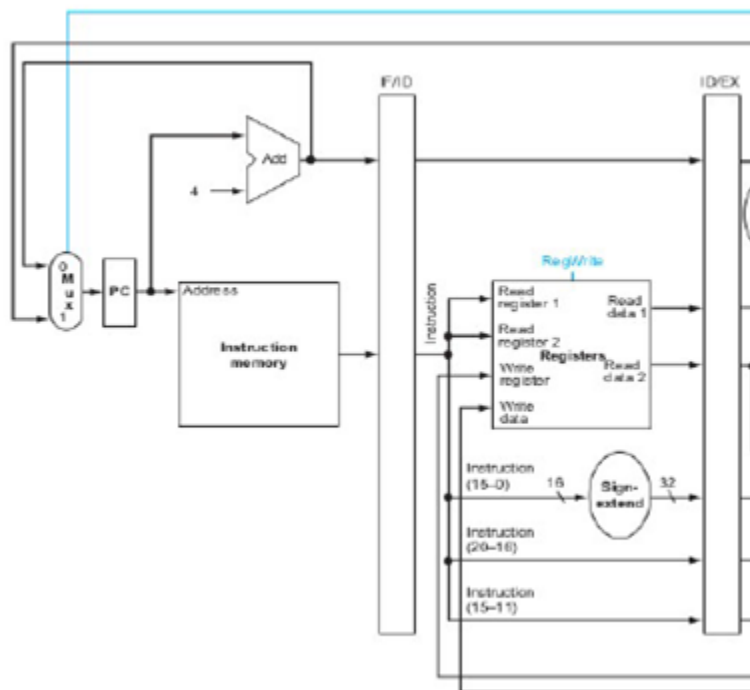
[illegible]

```
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
```





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;
```

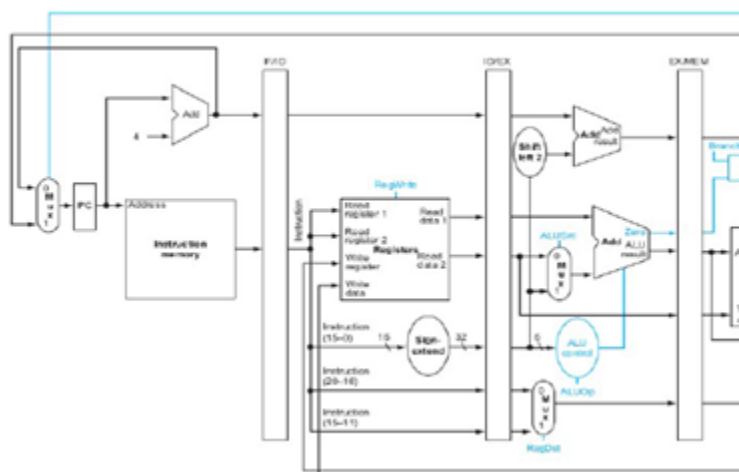


```
        outputRr2 = inputRr2;  
  
    end  
  
endmodule
```

- **buffer3.v**

En la creación del módulo del buffer tres, nos basamos en las inputs y outpus de 32 bits y solo una entrada y salida de 1 bit, con nombre 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
```



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 direcci3n, ya que tanto la entrada como la salida est3n conectadas en la misma posici3n.

always @(posedge clk)

begin

outputAddB3 = inputAddB3;

outputAddrst = inputAddrst;

outputRD2B3 = inputRD2B3;

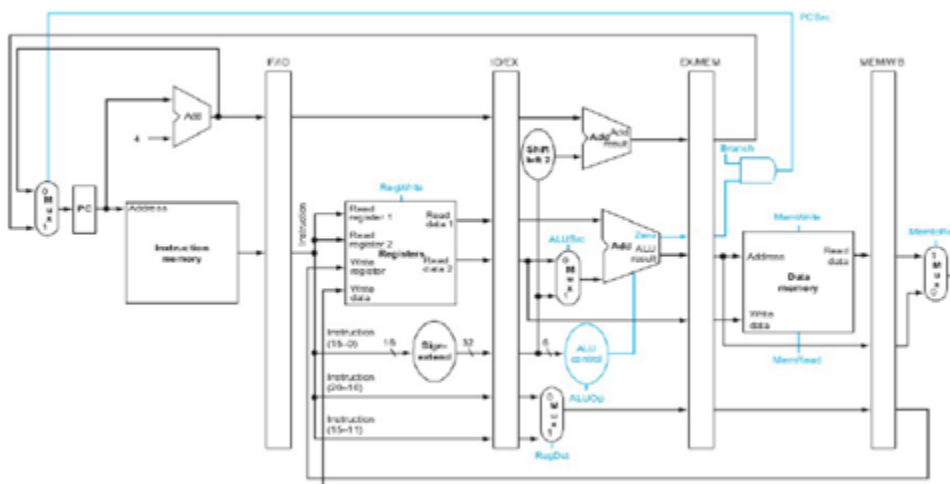
outputmux = inputmux;

end

endmodule

- **buffer4.v**

Y es as3 como llegamos al cuarto buffer que con ayuda de todo el esquema mostrado en la imagen que adjuntamos a continuaci3n, con entradas y salidas de 32 bits seg3n su procedencia. El nombre asignado a cada input output fue basado en las conexiones. Ejemplo:



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 direcci3n, ya que tanto la entrada como la salida est1n conectadas en la misma posici3n.

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

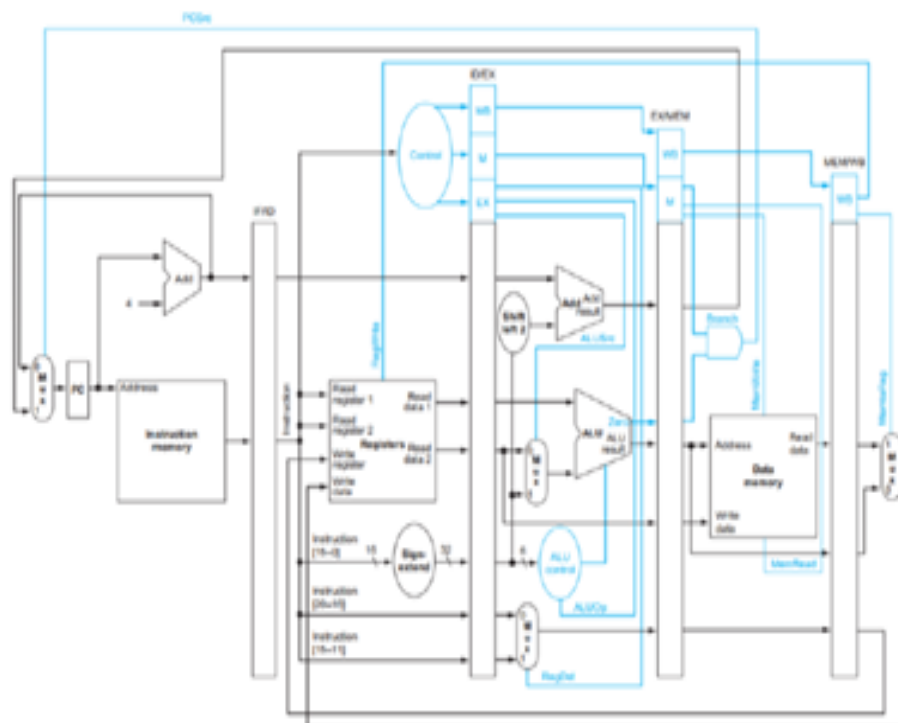


realizar todas y cada una de nuestras conexiones dentro del módulo, también se agregaron wires externos a nuestros buffer que hacen conexión única. A continuación se presentan nuestras conexiones adjuntando imagen de Fase 2, debido a que nos basamos en la imagen para poder lograrlo.

```

101 wire PCLSrc;
102
103 buffer1 b1(clk, inputAdd, inputInsMem, outputAdd, OutputInsMem);
104
105 buffer2 b2(INB2MemReg, INB2RegWRITE, INB2Branch, INB2MemWRITE, INB2MemRead, INB2RegDst, INB2ALUOP, INB2ALUSrc, clk, outputAdd, inputRD1, inputRD2, inputSinex, OutputInsMem[20:16], OutputInsMem[21:26]);
106
107 buffer3 b3(OTB2MemReg, OTB2RegWRITE, OTB2Branch, OTB2MemWRITE, OTB2MemRead, clk, inputAddB3, Z_flag, inputAddrst, outputRD2, inputmux, OTB3MemReg, OTB3RegWRITE, OTB3Branch, OTB3MemWRITE, OTB3MemRead, clk, inputDatmem, outputAddrst, outputmux, OTB4MemReg, OTB4RegWRITE, outputDatmem, outputAddrstB4, outputmuxB4);
108
109 unit_control ucontrol(OutputInsMem[31:26], INB2MemReg, INB2RegWRITE, INB2MemWRITE, INB2Branch, INB2MemRead, INB2ALUSrc, INB2RegDst, INB2ALUOP);
110
111 BankReg regb(OutputInsMem[25:21], OutputInsMem[20:16], outputmuxB4, OTB4RegWRITE, MuxBR, inputRD1, inputRD2);
112
113 alu_ctrl1 aluctrl1(outputSinex[5:0], OTB2ALUOP, ALUctr1ALU);
114
115 insmemx insm(PcInsMemYAdd, inputInsMem);
116
117 Suma_4b sumador(PcInsMemYAdd, 32'd4, inputAdd);
118
119 Suma_4b sumador2(outputAddB2, SL2Add, inputAddB3);
120
121 pc pc(MuxPc, clk, PcInsMemYAdd);
122
123 ALU_operator(outputRD1, MuxALU, ALUctr1ALU, inputAddrst, Z_flag);
124
125 mem_x_memory(outputAddrst, outputRD2B3, OTB3MemWRITE, OTB3MemRead, inputDatmem);
126
127 mux1 mux1(OTB2RegDst, outputRr1, outputRr2, inputmux);
128
129 mux2 mux2(PCSrc, outputAddB3, inputAdd, MuxPc);
130
131 mux2 mux3(OTB2ALUSrc, outputSinex, outputRD2, MuxALU);
132
133 mux2 mux4(OTB4MemReg, outputDatmem, outputAddrstB4, MuxBR);
134
135 MAnd andy(OTB3Branch, output_Z_flag, PCSrc);
136
137 shiftL shift(outputSinex, SL2Add);
138
139 signextend sign(OutputInsMem[15:0], inputSinex);
140
141 endmodule

```



**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 reorganizaron los cables conectando buffers y módulos entre sí.

## Memoria de instrucciones

ory Data - /tbsingled/datapath/memory/memory - Default

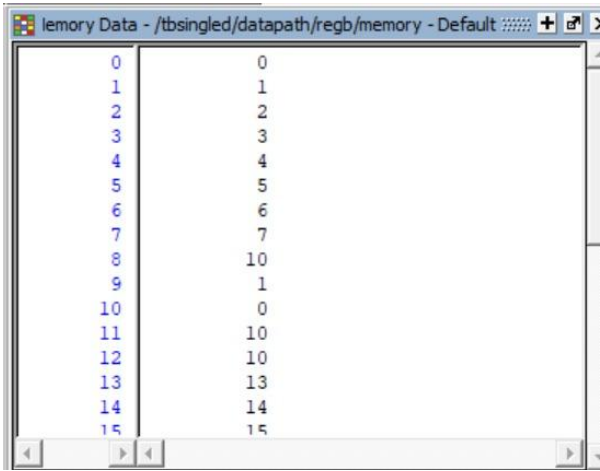
0	X
1	8
2	X
3	X
4	X
5	X
6	X
7	X
8	X
9	X
10	X
11	X
12	X
13	X
14	X
15	X
16	X
17	--

Memoria

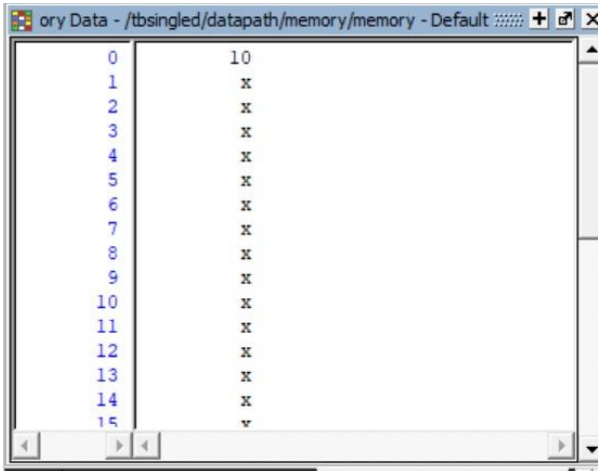


Memory Data - /tbsingled/datapath/regb/memory - Default	
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12
13	13
14	14
15	15
16	16
17	17

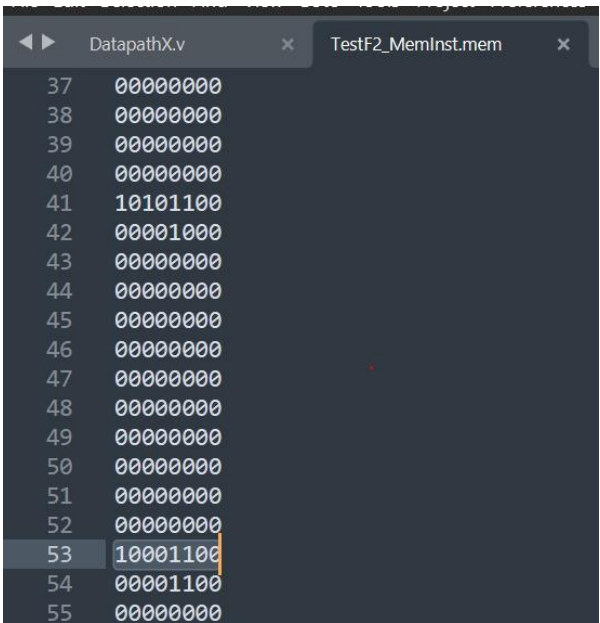
**Modificaciones**



Address	Value
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

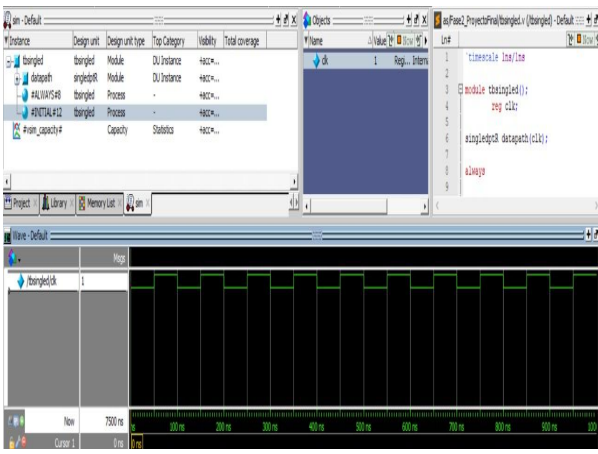


Address	Value
0	10
1	X
2	X
3	X
4	X
5	X
6	X
7	X
8	X
9	X
10	X
11	X
12	X
13	X
14	X
15	X



```

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
          
```



**Objects:**

Instance	Design Unit	Design Unit Type	Top Category	Visibility	Total Coverage
changed	changed	Module	DU Instance	400%	
changed	changed	Module	DU Instance	400%	
changed	changed	Process	-	400%	
changed	changed	Process	-	400%	
changed	changed	Capacity	Statistics	400%	

**Waveform:**

Signal: changed

Time: 0 ns to 100 ns

Value: 0, 1

**Reescribimos de forma correcta la instrucción para la "LW"**

**-Antes 11000100**

**-Ahora 10001100**

**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 parte aguas 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 mi 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 dificulto mucho así que pedí ayuda a mi equipo con la elaboración de este módulo, puedo decir que fue un buen trabajo en equipo para esta área porque se complemento 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 presentacion solo modificar cosas fue lo mas 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 como programar en ensamblador por que lo que nosotros estamos haciendo es investigado y no sabemos muy bien que es lo que estamos haciendo, Creí que la fase 2 seria mas sencilla ya que creí que agregaríamos nada mas unos detalles pero si se extendió muchísimo mas de lo esperado, Espero que la fase 3 si sea mucho mas sencillo que esta 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 mas en el lenguaje ensamblador, Había muy poco tiempo para juntamos entre los 3 casi nulo asi que fue mas difícil aun por que entre nosotros nos ayudamos en cosas que no sabemos, de ahi en mas creo que lo mas necesario hubiera sido saber como programar bien en lenguaje ensamblador pero creo que lo que se hizo estuvo bien.

## Memoria de instrucciones

## Referencias

- 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](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/jjruiz/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/1IRbFXDOvbtHinygvWsq1p3nRs02uxlmu/view?usp=drivesdk>
- [https://drive.google.com/file/d/1DfejVf4Xk2Sn4Cmyjw\\_Hc5NKZJulwLyp/view?usp=drivesdk](https://drive.google.com/file/d/1DfejVf4Xk2Sn4Cmyjw_Hc5NKZJulwLyp/view?usp=drivesdk)
- <https://drive.google.com/file/d/1RdrdOPbYSD092rNYEUBgbcvyA9SloAAT/view?usp=drivesdk>

#### Fase 2

