



# UNIVERSIDAD DE GUADALAJARA

Red Universitaria e Institución Benemérita de Jalisco

- **Universidad:**

Centro Universitario de Ciencias Exactas e Ingenierías

- **Materia:**

Arquitectura de computadoras

- **Nombre del maestro:**

Jorge Ernesto López Arce Delgado

- **Equipo Tr3s:**

Hugo Gabriel Garcia Saldivar

Oswaldo Daniel Maciel Vargas

- **Nombre de la actividad:**

Actividad 12 – Fase 3 proyecto final

## Tabla de contenido

- <b>Introducción:</b> .....	4
<b>Introducción fase 1:</b> .....	4
- <i>La arquitectura MIPS32</i> .....	4
- <i>Pipeline de 5 etapas para MIPS32</i> .....	4
- <i>Instrucciones tipo R, I, J:</i> .....	6
<b>Introducción fase 2:</b> .....	8
- <i>Instrucciones tipo I:</i> .....	8
<b>Introducción fase 3:</b> .....	9
- <i>Instrucciones tipo J:</i> .....	9
- <b>Objetivos</b> .....	10
<b>Objetivo fase 1:</b> .....	10
<b>Objetivo fase 2:</b> .....	10
<b>Objetivo fase 3:</b> .....	10
- <b>Desarrollo</b> .....	11
<b>Desarrollo fase 1:</b> .....	11
- <i>Ciclo Fetch:</i> .....	11
- <i>Códigos en Verilog del ciclo fetch:</i> .....	11
- <i>Definición de instrucciones tipo R:</i> .....	14
- <i>Datapath:</i> .....	15
- <i>Códigos en Verilog del datapath:</i> .....	16
- <i>Simulación y pruebas:</i> .....	23
- <i>Decodificar en Python:</i> .....	25
- <i>Simulación:</i> .....	27
<b>Desarrollo fase 2:</b> .....	29
- <i>Datapath para instrucciones tipo I:</i> .....	29
- <i>Operaciones tipo I a utilizar:</i> .....	30
- <i>Nuevos módulos para las operaciones tipo I:</i> .....	30
- <i>Módulos actualizados:</i> .....	35
- <i>Simulación y resultados:</i> .....	37
<b>Desarrollo fase 3:</b> .....	38
- <i>Datapath para instrucciones tipo J:</i> .....	38
- <i>Operaciones tipo J a utilizar:</i> .....	39

- Instrucciones añadidas R: .....	39
- Nuevos módulos para operaciones tipo J: .....	40
- Módulos actualizados Fase 3: .....	41
- Problemas en el desarrollo y solución: .....	43
- Construcción case 1: .....	46
- Construcción case 2: .....	48
- Construcción case 3: .....	50
- Construcción switch y programa completo: .....	52
- Pruebas algoritmo completo y simulación: .....	53
- <b>Conclusión</b> .....	59
<b>Conclusión fase 1:</b> .....	59
<b>Conclusión fase 2:</b> .....	59
<b>Conclusión fase 3:</b> .....	60
- <b>Referencias:</b> .....	61

## **- Introducción:**

### **Introducción fase 1:**

#### *- La arquitectura MIPS32*

La arquitectura MIPS32 es una ISA (Instruction Set Architecture) de 32 bits de tipo RISC (Reduced Instruction Set Computer) es decir de un enfoque que se aprovecha de instrucciones simples requiriendo de un menor gasto energético y un diseño simple en comparación con otros enfoques como el CISC (Complex Instruction Set Computer). Sin embargo, las arquitecturas con RISC requieren de muchas más instrucciones para realizar tareas complejas.

La ventaja con el conjunto de instrucciones de tipo RISC en comparación al CISC es que estas están altamente optimizadas y funcionan de manera muy eficiente en pipelines.

Estas instrucciones realizan operaciones básicas (como ADD, SUB, SLT, OR, LW) en formatos de tipo R, I y J lo que simplifica enormemente el diseño del hardware y permite al usuario tener en todo momento el control de las operaciones que se están realizando y hacia donde se están dirigiendo los resultados de dichas instrucciones.

Además, requieren de una menor cantidad de transistores dedicados lo que causa un menor consumo de energía y una mejor posibilidad de escalar dichas instrucciones, por ello la decodificación de sus instrucciones es bastante rápida lo que disminuye la latencia y garantiza la velocidad en arquitecturas de este tipo.

#### *- Pipeline de 5 etapas para MIPS32*

El pipeline son una serie de pasos secuenciales que se utilizan para llegar a un objetivo en particular similar a la definición de un algoritmo. La diferencia es que esta serie de pasos se dividen en distintas etapas en las que hasta terminar una de ellas con los datos obtenidos se operará la siguiente y así secuencialmente hasta la última etapa. Al comenzar una etapa ya se estará esperando una salida por parte de la etapa previa por lo que el pipeline siempre estará procesando información.

El MIPS32 tiene 5 etapas pertenecientes al pipeline. Cada etapa dura un ciclo de reloj y esto permite una etapa actual entregue un dato a la etapa siguiente y esté lista para obtener el resultado de la etapa previa a ella (como se mencionó anteriormente).

### 1. IF (Instruction Fetch):

El ciclo fetch tiene como objetivo leer las instrucciones desde la memoria de instrucciones.

#### - Módulos de IF:

Principalmente hacemos uso de un contador del programa (PC) sensible a un ciclo de reloj para que el programa funcione secuencialmente sin tener que forzar las instrucciones manualmente, por lo tanto, debemos de guardar dichas instrucciones en una memoria de instrucciones y gracias al PC leerlas para decodificarlas.

Finalmente hacemos uso de un sumador (de +4) conectado al contador del programa, esto porque cada instrucción para el MIPS32 se forma con una palabra de 32 bits, dicha palabra se guarda en la memoria de instrucciones en 4 partes diferentes de manera secuencial. Por lo tanto 4 espacios de memoria secuenciales partiendo del primer espacio (\$0 hasta \$3) representan una única instrucción.

### 2. ID (Instruction Decode):

ID se encarga de la decodificación de la instrucción y la lectura respectiva del banco de registros.

#### - Módulos de ID:

Una vez decodificada la instrucción, se leerán los respectivos registros del banco de registros en función de \$Rs y \$Rt. Una vez leídos Control Unit (unidad de control) generara señales dependiendo del código de operación (OP) estas señales indicaran a distintos módulos (banco de registro, ALU, RAM) que señales deberán activarse o desactivarse para que la instrucción sea procesada correctamente y realice lo deseado.

Y finalmente hacemos uso de un extensor de signo para convertir el campo inmediato de 16 bits a uno de 32 bits (esto solo para instrucciones de tipo I donde no sigue el mismo rango de bits que las instrucciones tipo R).

### 3. EX (Execute):

Ejecución se encarga de realizar las operaciones aritméticas y lógicas además de calcular direcciones para la memoria de instrucciones.

#### - Módulos de EX:

Esta etapa hace uso de la unidad aritmética-lógica (ALU) para realizar operaciones básicas como ADD, SUB, SLT, OR, AND, etc. Además, se apoya de multiplexores para seleccionar la entrada necesaria para la ALU dependiendo del tipo de instrucción.

Finalmente utiliza una unidad de salto (Branch) para evaluar condiciones de salto en la memoria de instrucciones.

#### 4. MEM (Memory Access):

Esta etapa se centra en la escritura y lectura para la memoria de datos (únicamente para instrucciones LW y SW).

##### - Módulos de MEM:

Hace uso de su respectiva memoria de datos (RAM) para almacenar y leer los datos que hayan sido guardados en ella. En conjunto con la unidad de control que activara la lectura o escritura de la memoria de datos.

#### 5. WB (Write Back):

Escribe los resultados obtenidos de las operaciones en el banco de registros.

##### - Módulos de WB:

Si la instrucción escribe en el banco de registro, este mismo será utilizando para almacenar el resultado de una operación dada ya sea en función de \$RD o \$Rt.

Para obtener este resultado se utiliza un multiplexor que decidirá si el dato a escribir viene de la ALU o de la memoria de datos.

##### - Instrucciones tipo R, I, J:

MIPS32 utiliza 3 tipos de instrucciones principales que son procesadas en la etapa "instruction decode" dependiendo del tipo de instrucción la unidad de control decidirá que módulos se activan o desactivan con el fin de que los datos pasen exitosamente y cumplan la función solicitada, a su vez encargándose de que no se guarde o lea ningún dato no requerido que pueda dificultar o dañar la lógica principal.

#### 1. Instrucciones de tipo R (Register):

OP	Rs	Rt	RD	Shamt	Funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
32 bits					

OP: Código de operación (generalmente en 0 para instrucciones tipo R).

Rs: Registro fuente.

Rt: Registro destino temporal.

RD: Registro destino final.

Shamt: Cantidad de desplazamiento (shift).

Funct: Subcódigo que especifica la operación exacta.

## 2. Instrucciones de tipo I (Immediate):

<b>OP</b>	<b>RS</b>	<b>RT</b>	<b>Immediate</b>
6 bits	5 bits	5 bits	16 bits
32 bits			

OP: Código de operación.

Rs: Registro fuente.

Rt: Registro destino o fuente.

Immediate: Valor constante de 16 bits (se extiende a 32 bits por tipo I).

## 3. Instrucciones de tipo J (Jump):

<b>OP</b>	<b>Target</b>
6 bits	26 bits
32 bits	

OP: Código de operación.

Target: Dirección de salto (se escala a 26 bits y se combina con PC).

## Introducción fase 2:

### - Instrucciones tipo I:

Esta fase 2 del proyecto final consiste en la implementación de módulos para el MIPS32, estos módulos se encargarán de que nuestro circuito sea capaz de procesar exitosamente las instrucciones de tipo I (o inmediatas).

Además de que ya se implementaran directamente los pipelines (o buffers) en el circuito para que este tenga un control en cada momento y los datos sean procesados en el momento en que se requieren.

Como mencionamos en la introducción de la fase 1: Para las instrucciones tipo I tomaremos el siguiente formato de palabra de instrucción:

OP	RS	RT	Immediate
6 bits	5 bits	5 bits	16 bits
32 bits			

OP: Código de operación.

Rs: Registro fuente.

Rt: Registro destino o fuente.

Immediate: Valor constante de 16 bits (se extiende a 32 bits por tipo I).

Esta palabra de 32 bits representa una instrucción completa para la arquitectura MIPS32 de tipo I, con esta instrucción podemos manipular operaciones que requieren del uso de registros y de un valor inmediato (constante) para algún calculo específico. Por ejemplo, si quieres hacer una suma con un numero n, pero dicho número no se encuentra ni en el banco de registros ni en la memoria de datos; por lo que este tipo de instrucciones nos dan la capacidad de ingresar directamente dicha constante n para alguna operación.



### Introducción fase 3:

#### - Instrucciones tipo J:

Las instrucciones de tipo J nos ayudan a brincar de instrucción a instrucción, estas instrucciones son muy útiles porque no depende de una condición como las instrucciones BEQ y BNE. Sobre todo, la instrucción jump sirve para saltar directamente a otra instrucción.

OP	Target
6 bits	26 bits
32 bits	

OP: Código de operación.

Target: Dirección de salto (se escala a 26 bits y se combina con PC).

Esta es la división de la palabra que se utiliza para una instrucción de tipo J, el opcode mandara a la unidad de control una señal para que esta prepare los distintos módulos para que pueda procesar una señal de este estilo. Por otro lado, el target (por ejemplo, en la instrucción jump) indica hacia que instrucción se dirigirá el flujo del programa una vez aplicada esta instrucción.

## **- Objetivos**

### **Objetivo fase 1:**

El objetivo de este reporte es realizar el ciclo fetch y el datapath pertenecientes a las instrucciones de tipo R, por lo que la fase 1 es un avance del proyecto principal.

Se describirán las funciones de tipo R que implementamos así como los módulos del ciclo fetch y el datapath.

Lo anterior se realizará en un lenguaje de descripción de hardware, en este caso Verilog por su sintaxis tan sencilla de comprender y su entendimiento de diversas funciones.

### **Objetivo fase 2:**

Para el objetivo de la fase 2 se busca evolucionar el datapath anterior (perteneciente a la fase 1 donde implementábamos las instrucciones de tipo R) a un nuevo datapath para que este nuevo circuito sea capaz de procesar instrucciones de tipo I.

Así mismo describiremos el funcionamiento de las nuevas instrucciones que agregamos así como los módulos necesarios para el funcionamiento óptimo del programa con su debida descripción.

### **Objetivo fase 3:**

Por último y para finalizar con el proyecto final de la materia arquitectura de computadoras, más a incluir soporte en nuestro MIPS32 para que este pueda manipular las instrucciones de tipo J y con ello hacer saltos donde nosotros queramos hacia donde lo necesitemos dándonos la posibilidad de establecer estructuras de control iterativas al igual que en los lenguajes de programación de un alto nivel.

De añadido incluiremos nuevas instrucciones tipo R y actualización a la ALU y ALU control.

A su vez para comprobar el funcionamiento de todas las instrucciones, módulos y pipelines diseñaremos un algoritmo para el MIPS32. Este algoritmo se basa en un menú similar a una estructura switch-case en la que dependiendo de la selección el programa este realizara 3 distintos algoritmos en los que se encuentran el invertir un arreglo, la multiplicación de 2 matrices de 2x2 y saber si un número del 2 al 100 es o no un número primo.

## - Desarrollo

### Desarrollo fase 1:

#### - Ciclo Fetch:

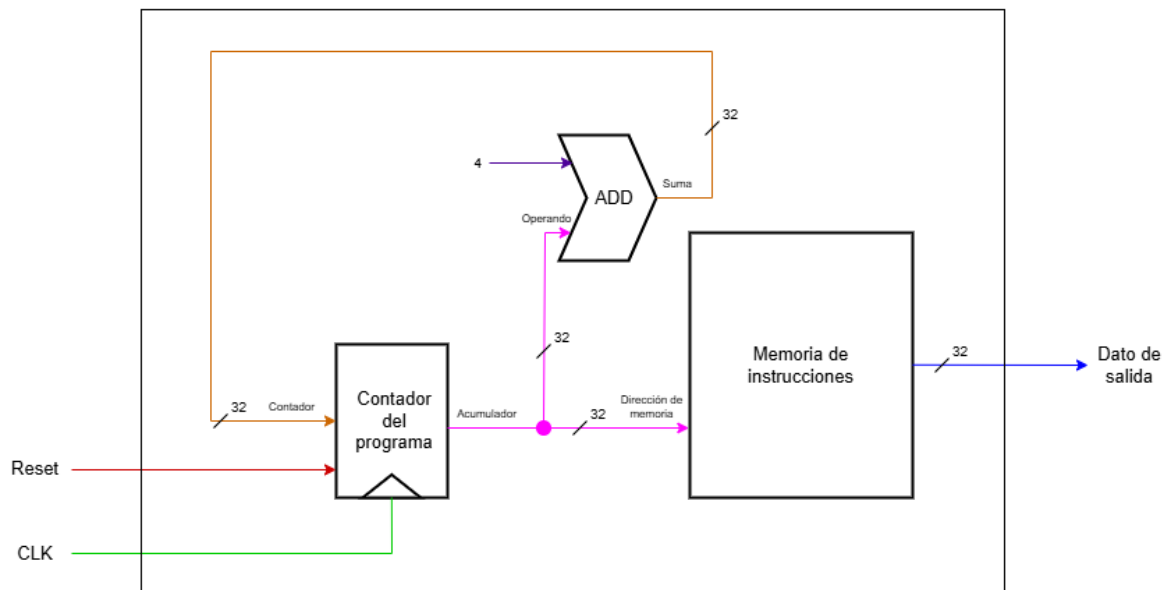
Siguiendo el pipeline, lo primero que debemos diseñar es nuestro ciclo fetch que se encargará de indicarle a la memoria de instrucciones cuales y en qué momento mandar las instrucciones para que sean decodificadas.

Este reporte al representar la fase 1 del proyecto final no vamos a incorporar el ciclo fetch en su totalidad, es decir; el ciclo fetch únicamente nos permitirá realizar operaciones de tipo R dejando las operaciones de tipo I y J para las fases 2 y 3 respectivamente.

Por lo tanto, no haremos uso del módulo Branch para esta primera fase. Únicamente del contador, sumador y memoria de instrucciones.

El ciclo fetch lo haremos en base al siguiente diagrama para visualizar de una mejor manera las conexiones que haremos:

Ciclo fetch



#### - Códigos en Verilog del ciclo fetch:

Siguiendo el diagrama anterior vamos a diseñar cada uno de los módulos necesarios para implementar el ciclo fetch de tal manera que este funcione correctamente y obtengamos la salida correcta en el momento preciso.

A continuación, nuestra definición de los módulos en verilog para el ciclo fetch acompañados de una breve descripción de cada uno de ellos:

### A. Contador del programa:

Se definio un contador utilizando un ciclo de reloj, cada que el programa detecte un flanco de subida y mientras que reset no esté activado; este mismo mandará la entrada del circuito hacia la salida. Este módulo no hace la suma directamente de ello se encarga el siguiente modulo:

```
1 module PC(  
2     input [31:0]contador,  
3     input clk,  
4     input reset,  
5     output reg [31:0]acumulador  
6 );  
7  
8 always @(posedge clk or posedge reset) begin  
9     if(reset)begin  
10        acumulador = 32'd0;  
11    end  
12    else begin  
13        acumulador = contador;  
14    end  
15 end  
16  
17 endmodule  
18
```

### B. Sumador:

La función del sumador es como su nombre lo dice, sumar el valor de entrada por una constante numérica, este caso el numero 4 esto porque como se mencionó en la introducción cada instrucción dentro de la memoria de instrucciones es equivalente a 4 registros de esta memoria.

```
1 module SUMADOR (  
2     input [31:0]operando,  
3     output reg [31:0]suma  
4 );  
5  
6 always @* begin  
7     suma = operando + 4;  
8 end  
9  
10 endmodule  
11
```

### C. Memoria de instrucciones:

Esta memoria en definición no es muy diferente a otras, esta memoria en teoría es una memoria ROM pues cuenta únicamente con lectura. El añadido que se le hizo fue que la memoria utiliza una concatenación en la salida de datos para unir 4 registros de 8 bits cada uno dando una palabra de 32 bits, es decir una instrucción completa para la arquitectura MIPS32.

```
1 module MEM_INST (
2     input [31:0] address,
3     output reg [31:0] dataOut
4 );
5
6 reg [7:0] mem_inst [0:999];
7
8 initial begin
9     $readmemb("Instrucciones.txt", mem_inst);
10 end
11
12 always @* begin
13     dataOut = {mem_inst[address], mem_inst[address+1],
14               mem_inst[address+2], mem_inst[address+3]};
15 end
16
17 endmodule
```

### D. Instancia ciclo fetch:

Por último, instanciamos estos módulos en un módulo principal, en el que las únicas señales de entrada pertenecen al ciclo de reloj y al reset del contador. Y la única señal de salida es la lectura de la memoria de instrucciones que nos entrega una palabra de 32 bits.

```
1 module FETCH (
2     input clk,
3     input reset,
4     output [31:0] instruccion
5 );
6
7 wire [31:0] pc_sumador_address;
8 wire [31:0] sumador_pc;
9
10 PC pc_fetch(
11     .contador(sumador_pc),
12     .clk(clk),
13     .reset(reset),
14     .acumulador(pc_sumador_address)
15 );
16
```

```
17 SUMADOR sumador_fetch(
18     .operando(pc_sumador_address),
19     .suma(sumador_pc)
20 );
21
22 MEM_INST mem_inst_fetch(
23     .address(pc_sumador_address),
24     .dataOut(instruccion)
25 );
26
27 endmodule
```

*- Definición de instrucciones tipo R:*

Antes de pasar con el datapath es importante describir las instrucciones de tipo R que se implementaron para este circuito, recordemos que estas instrucciones se almacenaran directamente en la memoria de instrucciones para automatizar la obtención y decodificación de estas instrucciones y que el código funcione de manera automática.

<b>Instrucciones tipo R</b>				
<b>No.</b>	<b>Mnemónico</b>	<b>Funct</b>	<b>Ensamblador</b>	<b>Descripción</b>
<b>1</b>	ADD	100000	ADD \$RD, \$Rs, \$Rt	Realiza una operación suma de los registros \$Rs y \$Rt y el resultado lo guarda en \$RD
<b>2</b>	SUB	100010	SUB \$RD, \$Rs, \$Rt	Realiza una operación resta de los registros \$Rs y \$Rt y el resultado lo guarda en \$RD
<b>3</b>	SLT	101010	SLT \$RD, \$Rs, \$Rt	Si el registro \$Rs es menor a \$Rt, escribe un 1 en \$RD, sino escribe un 0 en \$RD
<b>4</b>	AND	100100	AND \$RD, \$Rs, \$Rt	Realiza una operación AND de los registros \$Rs y \$Rt y el resultado lo guarda en \$RD
<b>5</b>	OR	100101	OR \$RD, \$Rs, \$Rt	Realiza una operación OR de los registros \$Rs y \$Rt y el resultado lo guarda en \$RD

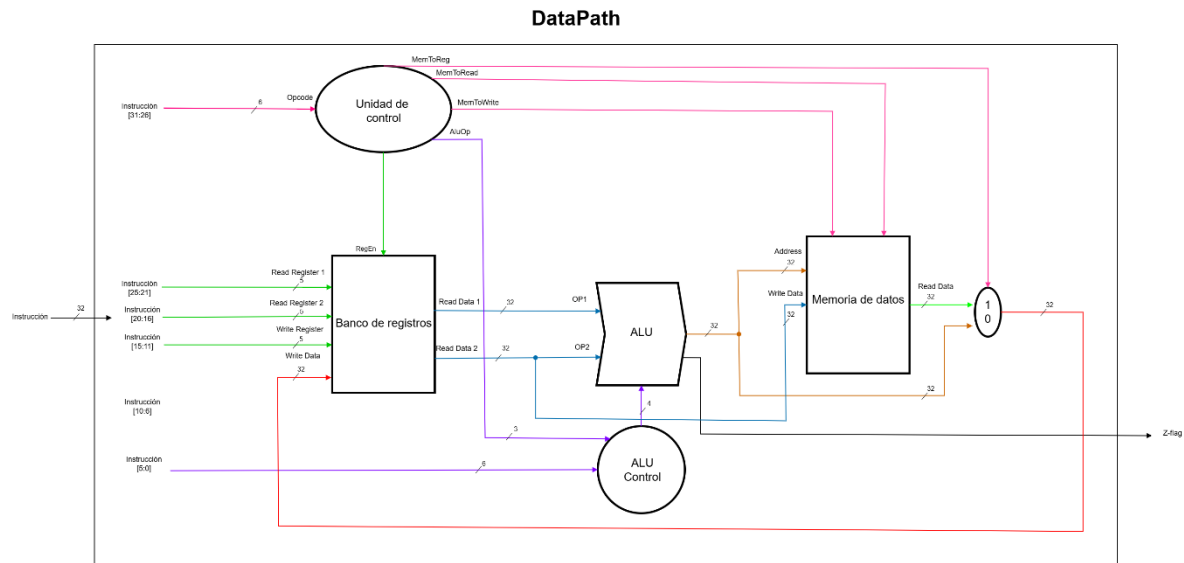
<b>X</b>	NOP	000000	NOP	No hacer nada
----------	-----	--------	-----	---------------

Recordemos que para las instrucciones de tipo R el funct será el indicado de definir la operación que se va a realizar, en este contexto el opcode se define únicamente con 000000, mandando un aviso a la unidad de control para que prepare los módulos para un procesamiento de una instrucción tipo R.

- *Datapath:*

Ahora es momento de construir un datapath tomando como referencia la arquitectura MIPS32 pero haciendo uso únicamente de las instrucciones tipo R pues es materia de esta fase del proyecto.

Para cumplir con dicho objetivo utilizaremos este datapath adaptado para que funcione únicamente con instrucciones de tipo R, con ello nos aseguramos que solo maneje estas instrucciones y de insertar una instrucción de tipo I o J no suceda nada pues se inhabilitan funciones para estas señales.



- Códigos en Verilog del datapath:

#### A. Banco de registros:

En el banco de registros se almacenan los diversos datos para realizar operaciones dentro de nuestro circuito, el banco de registros es la principal forma en la que nuestro programa obtiene datos e información que serán utilizados posteriormente para operaciones más complejas.

Se espera que el banco de registros ya venga con instrucciones definidas pues este mismo no puede leer o generar datos, pero si puede retroalimentarse con datos previamente obtenidos.

```
1 module BR (
2     input clk,
3     input [4:0] R_register_1,
4     input [4:0] R_register_2,
5     input [4:0] W_register,
6     input [31:0] W_data,
7     input RegEn,
8     output reg [31:0] R_data_1,
9     output reg [31:0] R_data_2
10 );
11
12     reg [31:0] MEM_BANCO [0:31];
13
14     initial begin
15         $readmemb("DatosBR.txt", MEM_BANCO);
16     end
17
18     always @* begin
19         R_data_1=MEM_BANCO[R_register_1];
20         R_data_2=MEM_BANCO[R_register_2];
21     end
22
23
24     always @(posedge clk) begin
25         if (RegEn) begin
26             MEM_BANCO[W_register] <= W_data;
27         end
28     end
29
30 endmodule
```



## B. Unidad de control:

Conforme al opcode recibido, la unidad de control le dictara a los demás módulos la función que deberán implementar ya sea activando o desactivando sus señales.

Gracias a la unidad de control es el programa puede entender del tipo de instrucción que se trata y con ello actuar para que el resultado sea el esperado.

```
1 module UC (
2     input [5:0] OpCode,
3     output reg MemToReg,
4     output reg MemToRead,
5     output reg MemToWrite,
6     output reg [2:0] AluOp,
7     output reg RegWrite
8 );
9
10 always @* begin
11     case (OpCode)
12         //Type-R
13         6'b000000:
14             begin
15                 MemToReg = 1'b0;
16                 MemToRead = 1'b0;
17                 MemToWrite = 1'b0;
18                 AluOp = 3'b010;
19                 RegWrite = 1'b1;
20             end
21         //SW
22         6'b101011:
23             begin
24                 MemToReg = 1'b0;
25                 MemToRead = 1'b0;
26                 MemToWrite = 1'b1;
27                 AluOp = 3'b000;
28                 RegWrite = 1'b0;
29             end
```

```
30 //LW
31 6'b100011:
32     begin
33         MemToReg = 1'b1;
34         MemToRead = 1'b1;
35         MemToWrite = 1'b0;
36         AluOp = 3'b000;
37         RegWrite = 1'b0;
38     end
39 endcase
40 end
41
42 endmodule
```

### C. ALU control:

ALU control es un módulo que con ayuda de la unidad de control define la operación que realizara la ALU, se implementaron 2 switch-case anidados con el objetivo de que la ALU control identifique cuando se trata de una instrucción R o I y una vez decidido, con el siguiente switch escoger específicamente el código de operación para que la ALU realice la acción requerida.

```
1 module AC (
2     input [2:0] AluOp,
3     input [5:0] Funct,
4     output reg [3:0] Op
5 );
6
7 always @* begin
8     case (AluOp)
9         //R-type
10        3'b010:
11            case (Funct)
12                //add
13                6'b100000:
14                    begin
15                        Op = 4'b0010;
16                    end
17                //subtract
18                6'b100010:
19                    begin
20                        Op = 4'b0110;
21                    end
22                //AND
23                6'b100100:
24                    begin
25                        Op = 4'b0000;
26                    end
27                //OR
28                6'b100101:
29                    begin
30                        Op = 4'b0001;
31                    end
32                //set on less than
33                6'b101010:
34                    begin
35                        Op = 4'b0111;
36                    end
37                6'b000000:
38                    begin
39                        Op = 4'b1111;
40                    end
41            endcase
42        //SW/LW
43        3'b000:
44            begin
45                Op = 4'b0010;
46            end
47        //branch equal
48        3'b001:
49            begin
50                Op = 4'b0110;
51            end
52    endcase
53 end
54 endmodule
```

#### D. ALU:

El módulo de la ALU realiza nuestras operaciones aritméticas para nuestros códigos de operación previamente definidos, dependiendo de los operandos y nuestro selector de operador; la ALU arrojará en una línea de salida el valor correspondiente a dicha operación.

Además, se añadió una bandera “zero flag” que arrojará un único bit, 1 cuando el resultado de la operación sea 0. Sino es así zero flag permanecerá en 0. Esto nos ayuda a tener un control de nuestras operaciones con el fin de comprender de una mejor manera la lógica que se está llevando a cabo en nuestras operaciones.

```
1  `timescale 1ns/1ns
2  module ALU (
3      input [31:0] Op_1,
4      input [31:0] Op_2,
5      input [3:0] Op_Alu,
6      output reg ZF,
7      output reg [31:0] Res
8  );
9
10 always @*
11 begin
12     case (Op_Alu)
13         //ADD
14         4'b0010:
15         begin
16             Res = Op_1 + Op_2;
17         end
18         //SUB
19         4'b0110:
20         begin
21             Res = Op_1 - Op_2;
22         end
23         //AND
24         4'b0000:
25         begin
26             Res = Op_1 & Op_2;
27         end
28         //OR
29         4'b0001:
30         begin
31             Res = Op_1 | Op_2;
32
33         //SLT
34         4'b0111:
35         begin
36             Res = (Op_1 < Op_2)? 32'd1 : 32'd0;
37         end
38
39         4'b1111:
40         begin
41             Res = Op_2 << 1;
42         end
43
44         default:
45             Res = 32'd0;
46     endcase
47
48     ZF = (Res == 1'b0)? 1'b1 : 1'b0;
49     // #100;
50 end
51
52
53 endmodule
```

### E. Memoria de datos:

La memoria de datos (RAM) es una memoria que guardara o leerá datos únicamente en las operaciones SW (store word) y LW (load word) de otra manera la memoria de datos NO se activara pues no se requiere de su funcionamiento (en este caso no hacemos uso de estas instrucciones pues son de tipo I).

La memoria de datos funciona como cualquier memoria, el añadido que se le hizo fue la implementación de 2 señales para leer y escribir respectivamente. Además de una serie de validaciones para que se decida la función que empleara la memoria y evitar errores a la hora de obtener los resultados.

```
1 module MEM (  
2     input [31:0] address,  
3     input [31:0] W_data,  
4     input Write,  
5     input Read,  
6     output reg [31:0] R_data  
7 );  
8  
9 reg [31:0] MEM_DATOS [0:127];  
10  
11 always @* begin  
12     if (Write && Read) begin  
13         R_data = 32'd0;  
14     end  
15     else if (Write) begin  
16         MEM_DATOS[address] = W_data;  
17     end  
18     else if (Read) begin  
19         R_data = MEM_DATOS[address];  
20     end  
21     else begin  
22         R_data = 32'd0;  
23     end  
24 end  
25  
26 endmodule
```

#### F. Multiplexor:

El multiplexor sirve para decidir qué dato se guardará en el banco de registros ya sea un dato proveniente de la memoria de datos o el resultado de una operación dada por la ALU.

```
1 module MUX (  
2     input [31:0] Mem_data,  
3     input [31:0] Alu_data,  
4     input Sel,  
5     output reg [31:0] W_data_BR  
6 );  
7  
8 always @* begin  
9     if (Sel) begin  
10        W_data_BR <= Mem_data;  
11    end  
12    else begin  
13        W_data_BR <= Alu_data;  
14    end  
15 end  
16  
17 endmodule
```

## G. Instancia datapath:

Finalmente hicimos la unión de todos nuestros módulos mediante una instancia con el objetivo de que nuestro proyecto quede de la misma manera que el datapath propuesto al inicio de esta sección. Colocando los cables y con los tamaños dados; definidos en el diagrama mismo.

### - Ciclo fetch y datapath:

```
1 module DPTR (
2     input clk_dtptr,
3     input [31:0] instruccion,
4     output ZF_DPTR
5 );
6
7 wire uc_br;
8 wire W_uc_mem;
9 wire R_uc_mem;
10 wire uc_mux;
11 wire [2:0] uc_ac;
12 wire [31:0] c1;
13 wire [31:0] c2;
14 wire [31:0] c3;
15 wire [3:0] c4;
16 wire [31:0] c5;
17 wire [31:0] c6;
18
19 UC uc_dptr(
20     .OpCode(instruccion[31:26]),
21     .MemToReg(uc_mux),
22     .MemToRead(R_uc_mem),
23     .MemToWrite(W_uc_mem),
24     .AluOp(uc_ac),
25     .RegWrite(uc_br)
26 );
27
28 BR br_dptr(
29     .clk(clk_dtptr),
30     .R_register_1(instruccion[25:21]),
31     .R_register_2(instruccion[20:16]),
32     .W_register(instruccion[15:11]),
33     .W_data(c6),
34     .RegEn(uc_br),
35     .R_data_1(c2),
36     .R_data_2(c1)
37 );
38
39 AC ac_dptr(
40     .AluOp(uc_ac),
41     .Func(instruccion[5:0]),
42     .Op(c4)
43 );
44
45 ALU alu_dptr(
46     .Op_1(c2),
47     .Op_2(c1),
48     .Op_Alu(c4),
49     .ZF(ZF_DPTR),
50     .Res(c3)
51 );
52
53 MEM mem_dptr(
54     .address(c3),
55     .W_data(c1),
56     .Write(W_uc_mem),
57     .Read(R_uc_mem),
58     .R_data(c5)
59 );
60
61 MUX mux_dptr(
62     .Mem_data(c5),
63     .Alu_data(c3),
64     .Sel(uc_mux),
65     .W_data_BR(c6)
66 );
67
68 endmodule
```

Ya para terminar, unimos la salida de nuestro ciclo fetch con la entrada de instrucciones de nuestro datapath, obteniendo de esta manera un módulo capaz de realizar instrucciones de tipo R y automatizado con una memoria de instrucciones.

```

1 module DPTR_FETCH_F1 (
2     input clk_f1,
3     input reset_f1,
4     output zf_f1
5 );
6
7 wire [31:0] c1;
8
9 FETCH fetch_f1 (
10     .clk(clk_f1),
11     .reset(reset_f1),
12     .instruccion(c1)
13 );
14
15 DPTR dtpr_f1(
16     .clk_dtpr(clk_f1),
17     .instruccion(c1),
18     .ZF_DPTR(zf_f1)
19 );
20
21 endmodule

```

*- Simulación y pruebas:*

Para comprobar que nuestro modulo funciona correctamente, se nos pidió resolver un problema con las siguientes instrucciones:

Instrucciones		
No.	Ensamblador	Lenguaje maquina
1	SUB \$20, \$15, \$9	000000 01111 01001 10100 00000 100010
2	NOP	000000 00000 00000 00000 00000 000000
3	SUB \$20, \$20, \$9	000000 10100 01001 10100 00000 100010
4	NOP	000000 00000 00000 00000 00000 000000
5	ADD \$15, \$5, \$15	000000 00101 01111 01111 00000 100000
6	NOP	000000 00000 00000 00000 00000 000000
7	ADD \$15, \$9, \$15	000000 01001 01111 01111 00000 100000
8	NOP	000000 00000 00000 00000 00000 000000
9	SLT \$21, \$20, \$15	000000 10100 01111 10101 00000 101010

Con el siguiente banco de registros (inicial):

Registro	Dato
...	0
5	20
6	12
7	55
8	72
9	100
...	0
15	999
...	0
20	0
21	0

Al momento de finalizar nuestro programa con las instrucciones anteriores, el banco de registros debería quedar de la siguiente manera para comprobar que el programa funciona correctamente:

Registro	Dato
...	0
5	20
6	12
7	55
8	72
9	100
...	0
15	1119
...	0
20	799
21	1

Veamos si esto se cumple en la simulación:



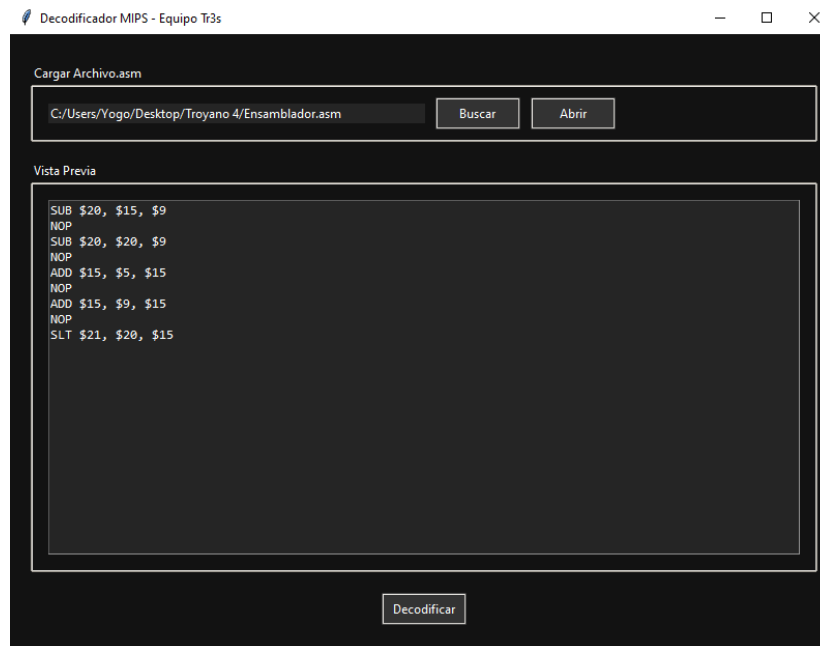
- *Decodificar en Python:*

Primeramente, utilizando un decodificador hecho en lenguaje Python decodificaremos nuestras instrucciones de lenguaje ensamblador (de un archivo.asm) a un archivo de texto con el propósito de que este mismo sea leído por la memoria de instrucciones de nuestro ciclo fetch.

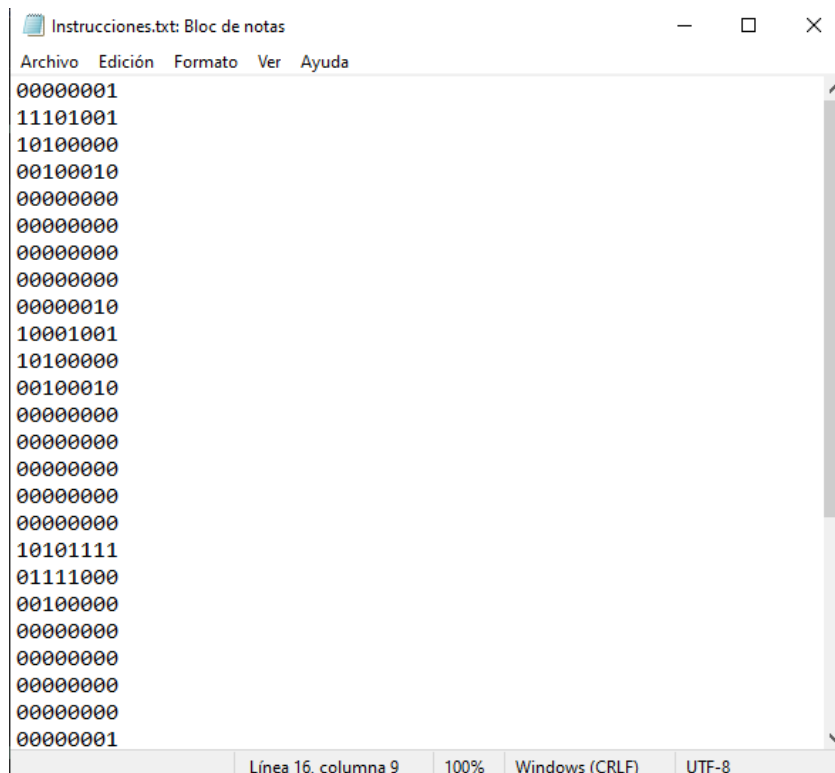
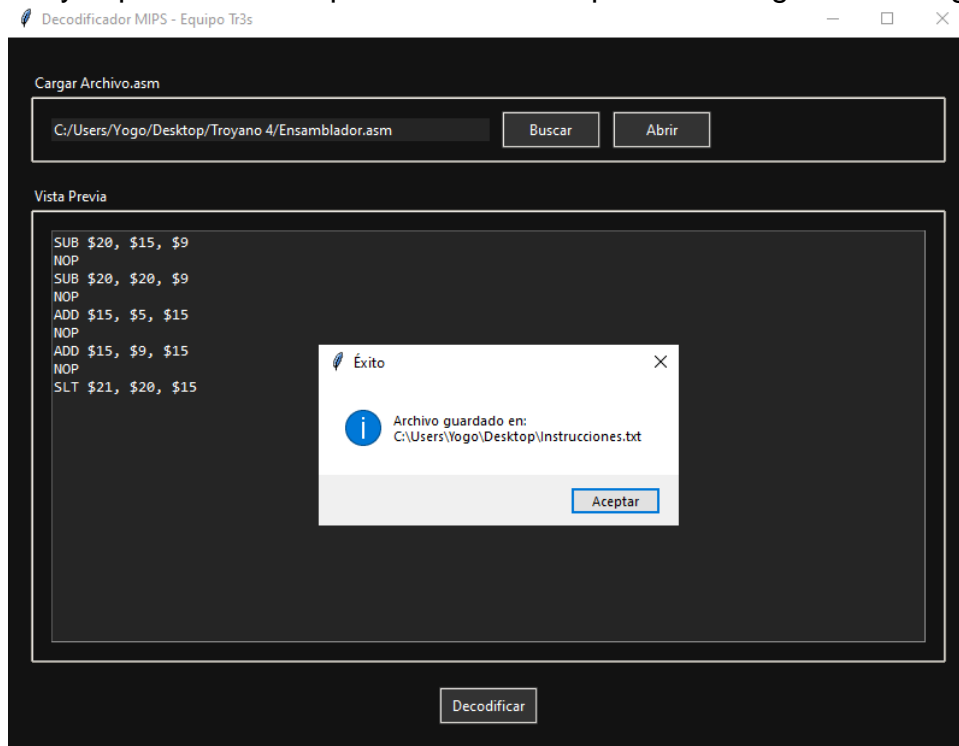
Este es nuestro archivo en ensamblador el cual será decodificado por el código de Python:

```
1  SUB $20, $15, $9
2  NOP
3  SUB $20, $20, $9
4  NOP
5  ADD $15, $5, $15
6  NOP
7  ADD $15, $9, $15
8  NOP
9  SLT $21, $20, $15
```

Cargamos nuestro archivo en el código y se vería de la siguiente manera:



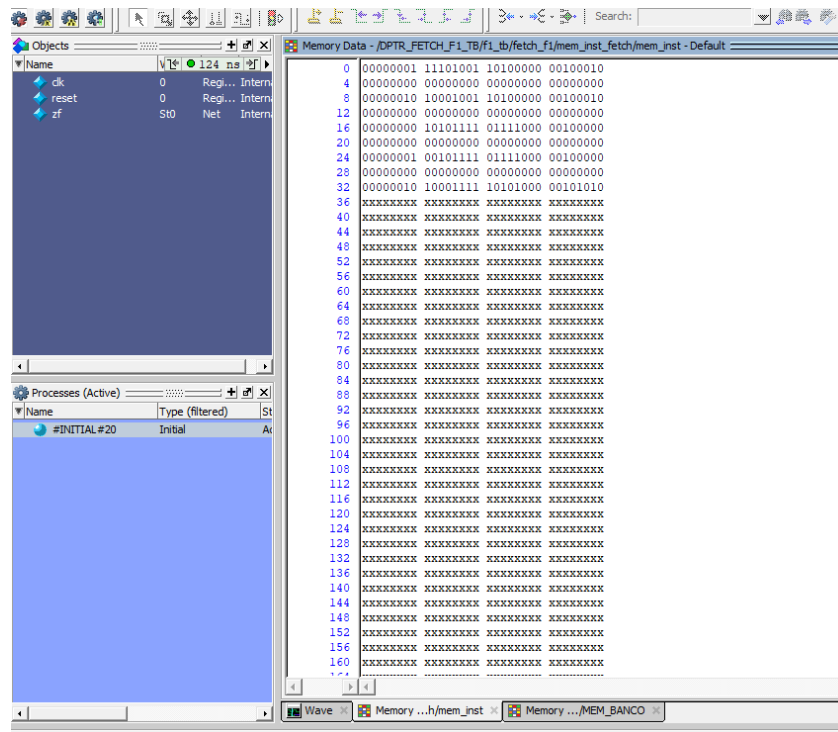
La decodificación fue exitosa y esta misma nos entregó un archivo de texto en lenguaje maquina con las instrucciones definidas, correspondiente a su tipo de instrucción y separándolas en palabras de 8 bits por cada renglón del código:



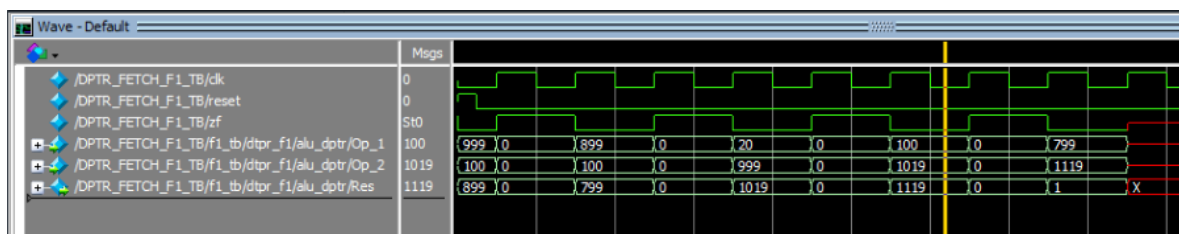
### - Simulación:

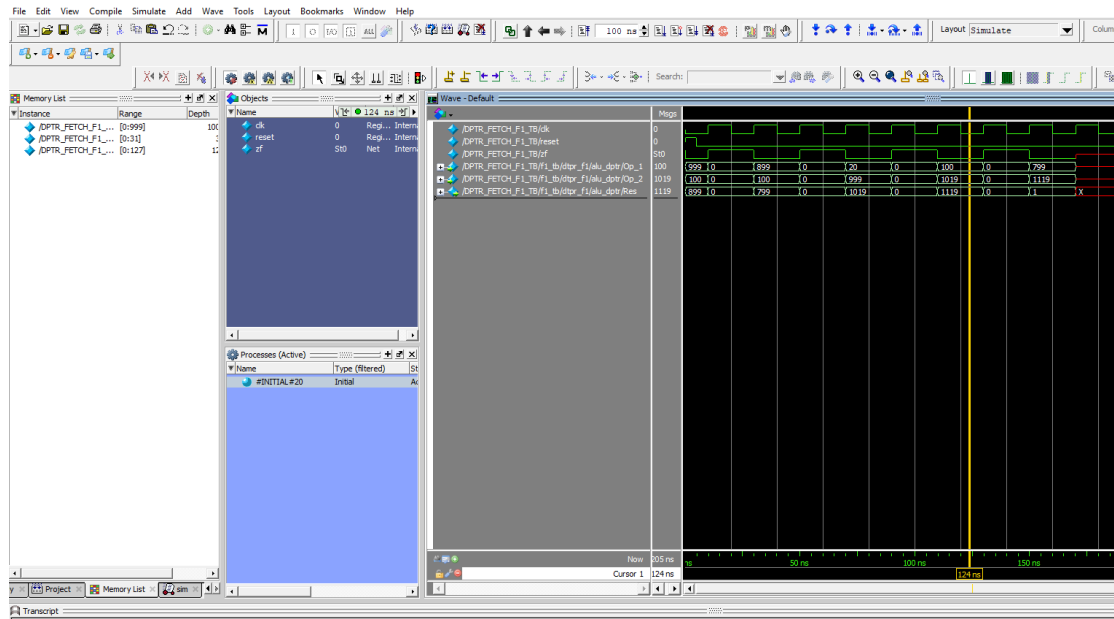
Realizamos la simulación a partir del módulo que combina el ciclo fetch con el datapath además de ingresarle las instrucciones previamente definidas a la memoria de instrucciones.

En la siguiente imagen se muestra la comprobación de que efectivamente se esta leyendo correctamente la memoria de instrucciones:

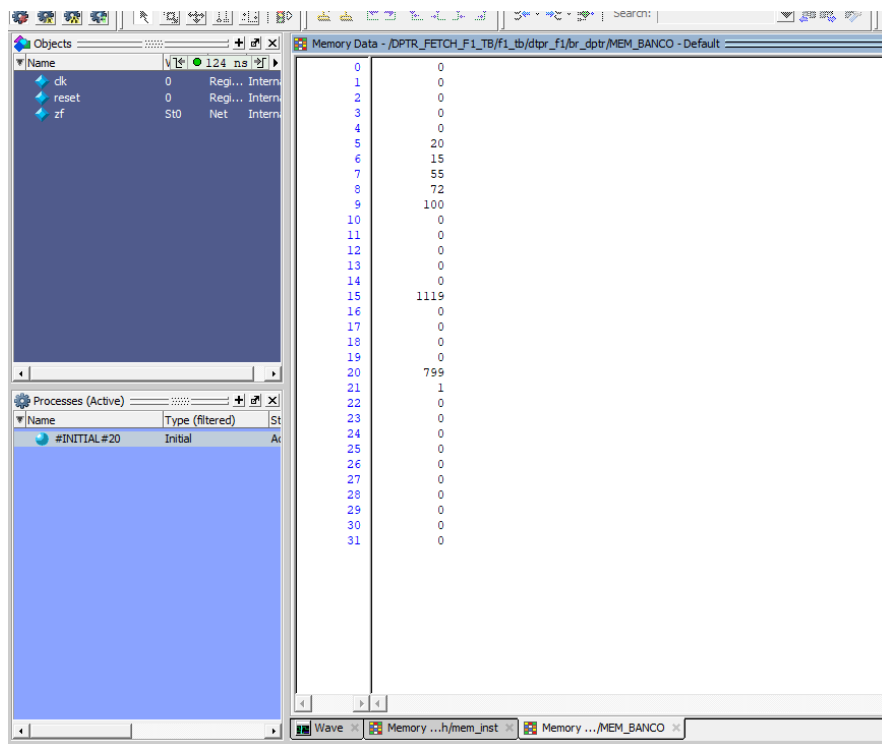


Las siguientes imágenes muestran las señales de nuestra instancia principal además de 3 señales importadas de otros módulos, 2 de ellas pertenecientes al dato que se está leyendo del banco de registros y la ultima el resultado que se está obteniendo de dicha operación.





Finalmente, en nuestro banco de registro obtuvimos los siguientes resultados:



Que comparándola con la tabla de los resultados deseados nos podemos dar cuenta de que son iguales y efectivamente el circuito funciona correctamente mostrando los resultados deseados y de la misma manera escribiéndolos en el banco de registros.

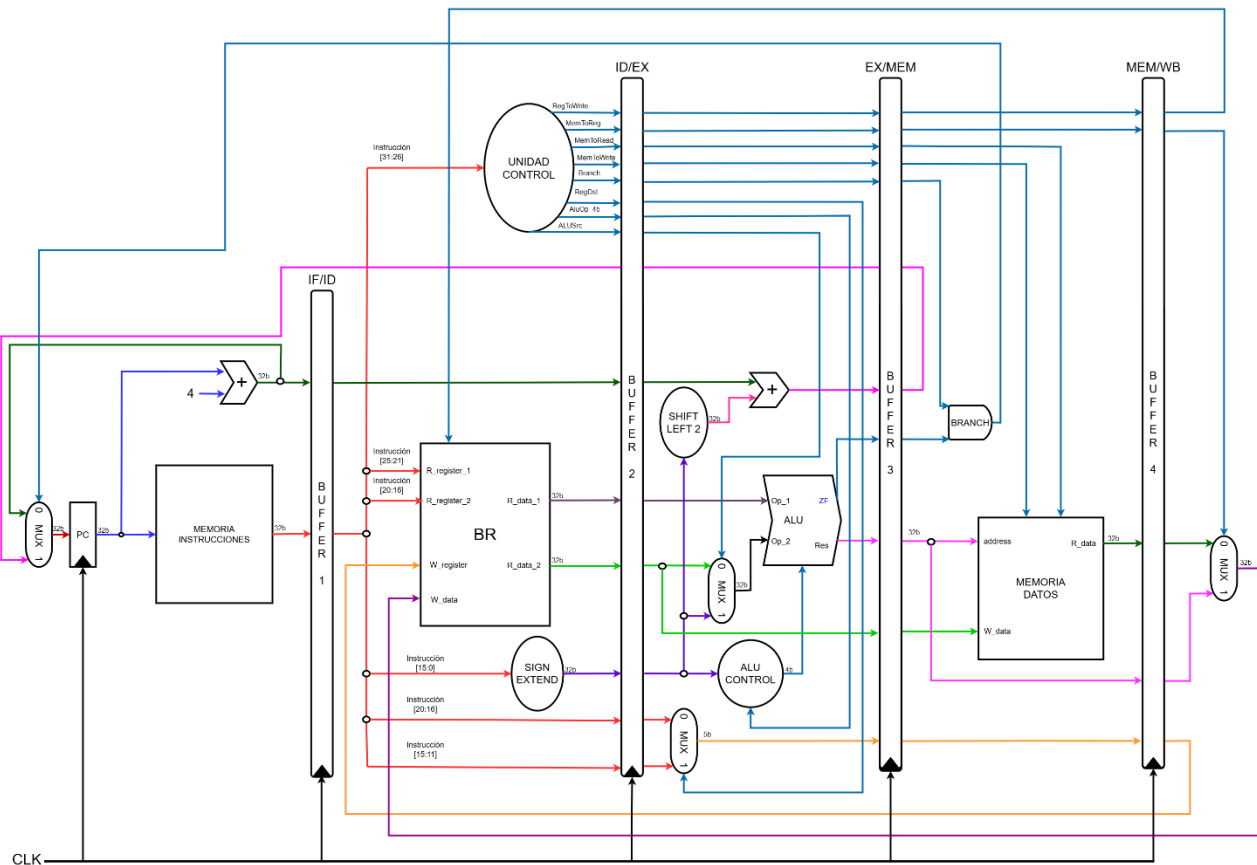
## Desarrollo fase 2:

### - Datapath para instrucciones tipo I:

El datapath de la fase 2 en comparación a la fase 1 tiene algo muy curioso, y es que este mismo añade varios multiplexores a la fórmula, principalmente estos toman el rol de servir como un control de paso para evitar que un dato no deseado viaje por un registro erróneo o que se active un módulo no requerido (en todo caso que empeoraría el funcionamiento del circuito).

Por ello mismo y para guiarnos correctamente diseñamos el siguiente diagrama para la construcción de la fase 2, como se mencionó en la introducción de esta fase; este diagrama ya incluye los distintos pipelines para mantener el control del MIPS32 y que cada señal se activa o desactive cuando sea necesaria en el momento en que es necesaria.

A continuación, el diagrama del datapath para las instrucciones de tipo I:



*- Operaciones tipo I a utilizar:*

En esta práctica implementaremos diversas instrucciones tipo I, estas instrucciones se definen por el opcode, en comparación con las instrucciones de tipo R que se definían por el funct, en este caso el funct pasa a ser parte del valor inmediato que se utiliza para obtener dicho valor constante que utilizamos en estas instrucciones.

<b>Instrucciones tipo R</b>				
<b>No.</b>	<b>Mnemónico</b>	<b>Opcode</b>	<b>Ensamblador</b>	<b>Descripción</b>
<b>1</b>	ADDI	001000	ADDI \$Rt, \$Rs, #Immediate	Realiza una operación suma del registro \$Rt con el valor inmediato, el resultado lo guarda en \$Rs
<b>2</b>	ORI	001101	ORI \$Rt, \$Rs, #Immediate	Realiza una operación OR del registro \$Rt con el valor inmediato, el resultado lo guarda en \$Rs
<b>3</b>	ANDI	001100	ANDI \$Rt, \$Rs, #Immediate	Realiza una operación AND del registro \$Rt con el valor inmediato, el resultado lo guarda en \$Rs
<b>4</b>	SW	101011	SW \$Rt, \$Rs, #Immediate	Calcula la posición relativa [\$Rs + #In] y guarda el dato \$Rt (memoria de datos)
<b>5</b>	LW	100011	LW \$Rt, \$Rs, #Immediate	Calcula la posición relativa [\$Rs + #In] y extrae el dato \$Rt (banco de registros)
<b>6</b>	BEQ	000100	BEQ \$Rs, \$Rt, #Immediate	Compara los registros \$Rs y \$Rt, si son iguales se activa el salto (#in), si no el programa continúa con la siguiente instrucción.

*- Nuevos módulos para las operaciones tipo I:*

Para que el circuito funcione correctamente fue indispensable añadir nuevos módulos para mantener el control entre las instrucciones de tipo R con las de tipo I, a continuación, la definición y declaración de los módulos:

### A. Multiplexor PC:

Este multiplexor toma como selector la señal de salida del Branch para realizar operaciones de tipo Branch y de esta forma saltar instrucciones con apoyo del contador del programa, es muy importante el uso de un multiplexor aquí porque no siempre se va a activar este salto, únicamente cuando la condición lo requiera. Por ello tener un control de paso para esta señal es indispensable.

```
1 module MultiplexorPC (  
2     input [31:0] add_fetch,  
3     input [31:0] add_result,  
4     input PCSrc,  
5     output reg [31:0] W_data_BR  
6 );  
7  
8 always @* begin  
9     if (PCSrc) begin  
10        W_data_BR <= add_result;  
11    end  
12    else begin  
13        W_data_BR <= add_fetch;  
14    end  
15 end  
16  
17 endmodule
```

### B. Multiplexor Instrucción I:

Este multiplexor identifica (con ayuda de la unidad de control) que rango de bits de la instrucción ingresada será destinada al write register del banco de registros, esto es indispensable porque la separación de bits cambia entre instrucciones de tipo R e I, en las instrucciones de tipo I \$RD es tomado como parte del inmediato.

```
1 module MultiplexorTipoI (  
2     input [4:0] InstruccionRT,  
3     input [4:0] InstruccionRD,  
4     input RegDst,  
5     output reg [4:0] W_register_BR  
6 );  
7  
8 always @* begin  
9     if (RegDst) begin  
10        W_register_BR <= InstruccionRD;  
11    end  
12    else begin  
13        W_register_BR <= InstruccionRT;  
14    end  
15 end  
16  
17 endmodule
```

### C. Multiplexor ALU:

Este multiplexor es un apoyo para el Branch, de activarse mandará el sign-extend esta señal a la ALU en donde se comprobará una condición para una instrucción Branch, dependiendo de este resultado puede o no activarse la zero flag.

Por otro lado, si no se activa ALUSrc el dato 2 que compara la ALU se tratará de la lectura del segundo registro del banco de registros.

```
1 module MultiplexorALU (  
2     input [31:0] R_data_2_BR,  
3     input [31:0] signextend,  
4     input ALUSrc,  
5     output reg [31:0] Data_2_ALU  
6 );  
7  
8 always @* begin  
9     if (ALUSrc) begin  
10        Data_2_ALU <= signextend;  
11    end  
12    else begin  
13        Data_2_ALU <= R_data_2_BR;  
14    end  
15 end  
16  
17 endmodule
```

### D. Multiplexor Memoria Datos (era el multiplexor anterior para la fase 1):

Este multiplexor se utiliza para definir si el dato a guardar en el banco de registros proviene directamente de la ALU o en su lugar, proviene de la memoria de datos. Este módulo es una evolución del único multiplexor que teníamos en la fase 1.

```
1 module MultiplexorMemDatos (  
2     input [31:0] result_ALU,  
3     input [31:0] read_MEM,  
4     input MemToReg,  
5     output reg [31:0] W_data_BR  
6 );  
7  
8 always @* begin  
9     if (MemToReg) begin  
10        W_data_BR <= read_MEM;  
11    end  
12    else begin  
13        W_data_BR <= result_ALU;  
14    end  
15 end  
16  
17 endmodule
```



#### E. Shift left two:

El módulo shift left two se encarga de recorrer 2 bits a la izquierda de la palabra de bits de entrada, este proceso da como resultado una multiplicación del valor de entrada por una constante 4 (en decimal). Principalmente este módulo es útil para operaciones Branch y la razón de multiplicar la palabra de entrada por una constante 4 es porque cada instrucción definida en la memoria de instrucciones utiliza 4 espacios de memoria. Por lo que al hacer esto podemos definir más fácilmente a que instrucción queremos llegar.

```
1 module ShiftLeftTwo(  
2     input [31:0] entrada,  
3     output reg [31:0] salidaLeftTwo  
4 );  
5  
6 always @* begin  
7     salidaLeftTwo = entrada << 2;  
8 end  
9 endmodule
```

#### F. Sign Extend:

Se encarga de añadir bits a la palabra de entrada tomando en cuenta de si se trata de un valor positivo o negativo (complemento a 2).

```
1 module SignExtend(  
2     input [15:0] instruccionOriginal,  
3     output reg [31:0] instruccionExtendida  
4 );  
5  
6 always @* begin  
7     case (instruccionOriginal[15])  
8     1'b0: instruccionExtendida = {16'b0000000000000000, instruccionOriginal};  
9     1'b1: instruccionExtendida = {16'b1111111111111111, instruccionOriginal};  
10    endcase  
11 end  
12  
13 endmodule
```

#### G. Branch:

Este módulo utiliza una entrada zero flag como se mencionó anteriormente, esto con el objetivo de identificar una si la condición que se hizo para una instrucción Branch es correcta, para que se active este módulo se necesita que ambas entradas estén activas, por ello se utiliza una operación de tipo AND.

```
1 module Branch(  
2     input Zero_Flag,  
3     input Branch_UC,  
4     output reg Salida_MUXPC  
5 );  
6  
7 always @* begin  
8     Salida_MUXPC = Zero_Flag & Branch_UC;  
9 end  
10 endmodule
```

#### H. Sumador 2:

Este segundo sumador se encarga de calcular la instrucción que queremos ejecutar dependiendo directamente de una instrucción Branch.

```
1 module SUMADOR_2 (  
2     input [31:0] resultado_sumador,  
3     input [31:0] resultado_shift,  
4     output reg [31:0] suma  
5 );  
6  
7 always @* begin  
8     suma = resultado_sumador + resultado_shift;  
9 end  
10  
11 endmodule
```

## - Módulos actualizados:

### A. Unidad de Control:

El primer módulo actualizado fue la unidad de control, esto porque se requerían una mayor cantidad de señales de salida para definir los módulos que se activarían para mantener un control estable por tipo de instrucción. En ese contexto se añadieron 3 señales de salida pertenecientes a los multiplexores de la ALU, tipo I y memoria de datos.

Además de añadir nuevas validaciones para las instrucciones de tipo I:

```

1 module UC (
2     input [5:0] OpCode,
3
4     output reg MemToReg,
5     output reg MemToRead,
6     output reg MemToWrite,
7     output reg [2:0] AluOp,
8     output reg RegWrite,
9
10    output reg RegDst,
11    output reg Branch,
12    output reg ALUSrc
13 );
14
15 always @* begin
16     case (OpCode)
17         //Type-R
18         6'b000000:
19             begin
20                 MemToReg = 1'b0;
21                 MemToRead = 1'b0;
22                 MemToWrite = 1'b0;
23                 AluOp = 3'b010;
24                 RegWrite = 1'b1;
25
26                 RegDst = 1'b1;
27                 Branch = 1'b0;
28                 ALUSrc = 1'b0;
29             end
30             //SW
31             6'b101011:
32                 begin
33                     MemToReg = 1'b0;
34                     MemToRead = 1'b0;
35                     MemToWrite = 1'b1;
36
37                     AluOp = 3'b000;
38                     RegWrite = 1'b0;
39
40                     RegDst = 1'b0;
41                     Branch = 1'b0;
42                     ALUSrc = 1'b1;
43                 end
44             //LW
45             6'b100011:
46                 begin
47                     MemToReg = 1'b1;
48                     MemToRead = 1'b1;
49                     MemToWrite = 1'b0;
50                     AluOp = 3'b000;
51                     RegWrite = 1'b1;
52
53                     RegDst = 1'b0;
54                     Branch = 1'b0;
55                     ALUSrc = 1'b1;
56                 end
57             //ORI
58             6'b001101:
59                 begin
60                     MemToReg = 1'b0;
61                     MemToRead = 1'b0;
62                     MemToWrite = 1'b0;
63                     AluOp = 3'b010;
64                     RegWrite = 1'b1;
65
66                     RegDst = 1'b0;
67                     Branch = 1'b0;
68                     ALUSrc = 1'b1;
69                 end
70             //ANDI
71             6'b001100:
72                 begin
73                     MemToReg = 1'b0;
74                     MemToRead = 1'b0;
75                     MemToWrite = 1'b0;
76                     AluOp = 3'b011;
77                     RegWrite = 1'b1;
78
79                     RegDst = 1'b0;
80                     Branch = 1'b0;
81                     ALUSrc = 1'b1;
82                 end
83             //ADDI
84             6'b001000:
85                 begin
86                     MemToReg = 1'b0;
87                     MemToRead = 1'b0;
88                     MemToWrite = 1'b0;
89                     AluOp = 3'b000;
90                     RegWrite = 1'b1;
91
92                     RegDst = 1'b0;
93                     Branch = 1'b0;
94                     ALUSrc = 1'b1;
95                 end
96             //BEQ
97             6'b000100:
98                 begin
99                     MemToReg = 1'b0;
100                    MemToRead = 1'b0;
101                    MemToWrite = 1'b0;
102                    AluOp = 3'b001;
103                    RegWrite = 1'b0;
104
105                    RegDst = 1'b0;
106                    Branch = 1'b1;
107                    ALUSrc = 1'b0;
108                end
109            endcase
110        end
111    endmodule

```

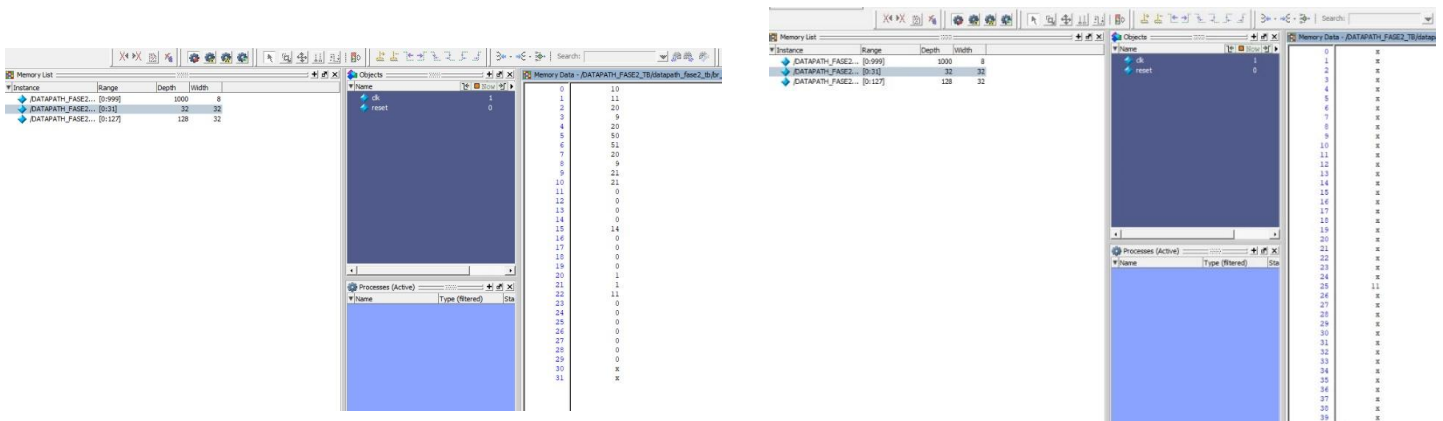
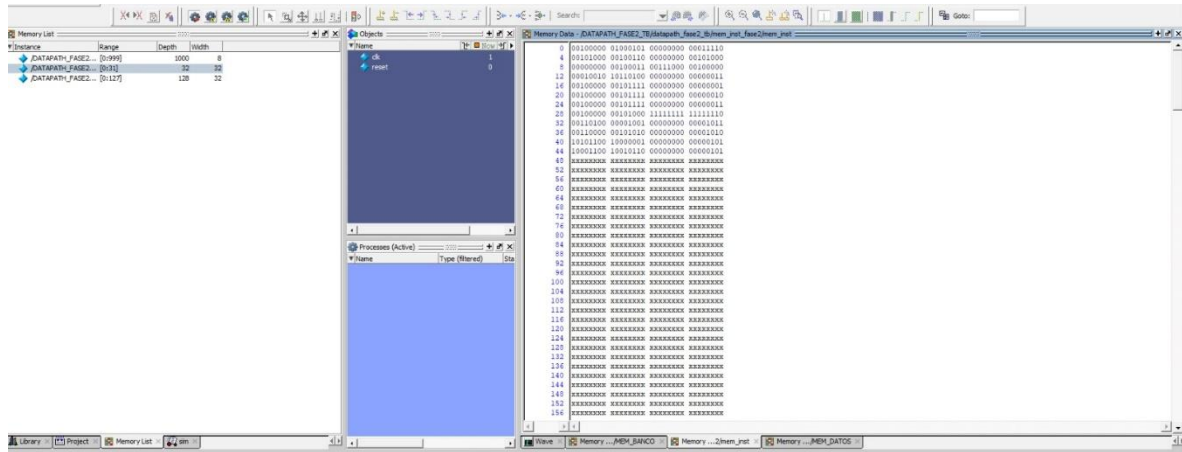
## B. ALU control:

Para la ALU control solamente se añadieron nuevas validaciones para las instrucciones de tipo I, las señales y los códigos de las instrucciones de tipo R no cambiaron.

```
43 //SW/LW/ADDI
44 3'b000:
45 begin
46     Op = 4'b0010;
47 end
48 //branch equal
49 3'b001:
50 begin
51     Op = 4'b0110;
52 end
53 //ORI
54 3'b010:
55 begin
56     Op = 4'b0001;
57 end
58 //ANDI
59 3'b011:
60 begin
61     Op = 4'b0000;
62 end
63 endcase
64 end
65
66 endmodule
```

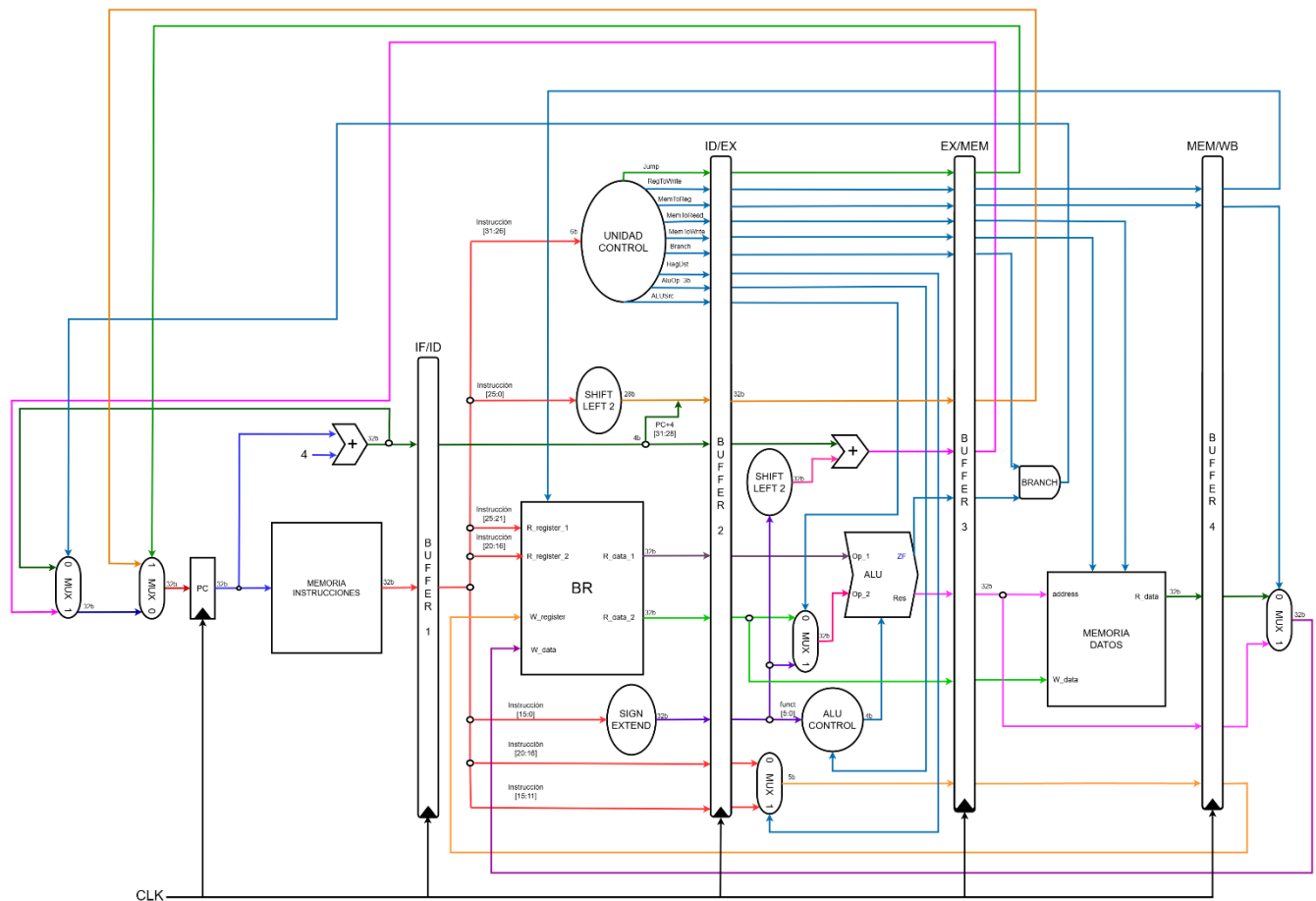
### - Simulación y resultados:

Esta simulación únicamente es para comprobar que la conexión, lógica y procesamiento de instrucciones es correcta y el circuito puede manipular exitosamente instrucciones de tipo R e I sin errores:



- *Datapath para instrucciones tipo J:*

- *Datapath para instrucciones tipo J:*  
Para guiarnos con la construcción de la fase 3 al igual que con las fases previas decidimos diseñar primeramente un diagrama para tener una guía visual de como solucionaremos el problema de la construcción de esta fase, a continuación, nuestro diagrama perteneciente al datapath de la fase 3:



Realmente a diferencia de la fase 2 el nuevo datapath no incluye una gran cantidad de cambios, pero si es muy importante observar detenidamente las conexiones porque algunos cables como los pertenecientes al multiplexor cambian de destino. Si esto no se observa correctamente obviamente el circuito no funcionara y no realizara las operaciones jump correctamente.

Sin embargo, podemos decir que fue muy sencillo encontrar la solución a este problema una vez planteamos el diagrama, con este mismo nos guiamos directamente entre los cables y logramos una conexión exitosa del circuito.

- *Operaciones tipo J a utilizar:*

Por sencillez y para no comprometernos con el diseño de un predicador de saltos únicamente añadimos una instrucción de tipo J, y la que consideramos la más importante siendo esta la instrucción jump.

Instrucciones tipo J				
No.	Mnemónico	Opcode	Ensamblador	Descripción
1	J	000010	J #target	Realiza un salto hacia la instrucción definida en #target

Esta instrucción nos ayudara para crear estructuras de control iterativas similares a las vistas en lenguajes de programación conocidos. Utilizando instrucciones J y BEQ podemos acercarnos a estructuras similares a do-while y for. El principal beneficio que veo en este diseño de arquitectura es poder implementar un contador dentro de algún algoritmo para mantener un control y una vez llegada a una condición salir de esta estructura con un resultado ya definido.

- *Instrucciones añadidas R:*

Para poder realizar los algoritmos que queremos implementar es necesario definir nuevas instrucciones de tipo R porque de no tenerlas sería demasiado complicado realizar los algoritmos que estamos buscando implementar.

Nuevas instrucciones tipo R definidas:

Nuevas instrucciones tipo R				
No.	Mnemónico	Funct	Ensamblador	Descripción
1	DIVU	011011	DIVU \$RD, \$Rs, \$Rt	Realiza una operación módulo de los registros \$Rs y \$Rt y el resultado lo guarda en \$RD
2	RSQRT	010110	RSQRT \$RD, \$Rs, \$0	Realiza una operación raíz cuadrada de los registros \$Rs y el resultado lo guarda en \$RD
3	MUL	000010	MUL \$RD, \$Rs, \$Rt	Realiza una operación multiplicación de los registros \$Rs y \$Rt y el resultado lo guarda en \$RD

- *Nuevos módulos para operaciones tipo J:*

Muy bien ahora guiándonos con el diagrama anteriormente proporcionado vamos a diseñar los nuevos módulos para que nuestro datapath pueda procesar instrucciones de tipo J y no tenga problemas al seguir procesando las mismas instrucciones de tipo R e I.

A. Multiplexor jump:

Este multiplexor fue el último en ser añadido, el funcionamiento del mismo consiste en decidir si la instrucción se trata de un tipo jump o no, de ser una instrucción de tipo jump el multiplexor dejara pasar un cálculo que se realizó dentro del circuito para calcular la dirección hacia donde saltara el flujo del programa, de no activarse significara que no se trata de una instrucción de tipo jump por lo que el programa continuara de forma secuencial.

```
1 module MUX_JUMP (  
2     input [31:0] entrada_Jump,  
3     input [31:0] entrada_muxPC,  
4     input Jump,  
5     output reg [31:0] salida_a_PC  
6 );  
7  
8 always @* begin  
9     if (Jump) begin  
10        salida_a_PC <= entrada_Jump;  
11    end  
12    else begin  
13        salida_a_PC <= entrada_muxPC;  
14    end  
15 end  
16  
17 endmodule
```

B. Shift left two jump:

El uso que se le da a este módulo es ayudar al circuito a calcular la dirección en la que el jump debe saltar, para lograr esto añadiremos 2 bits del lado derecho de la señal de entrada dando como resultado una multiplicación por una constante 4 y una salida de 28 bits. Como a un nos faltan 4 bits para la arquitecta de 32 ahora este resultado obtenido lo vamos a concatenar con los 4 bits más significativos obtenidos del sumador del PC para que obtengamos una formula parecida a esta:

**{Sumador [31:28] , ShiftLeftTwoJump}**



```

1 module SHIFT_LEFT_2_JUMP (
2     input [25:0] entrada,
3     output reg [27:0] salidaLeftTwo
4 );
5
6 always @* begin
7     salidaLeftTwo = {entrada, 2'b00};
8 end
9
10 endmodule

```

- Módulos actualizados Fase 3:

A. Unidad de Control:

El paso más importante aquí es añadirle un nuevo case y señal destinada únicamente para las instrucciones de tipo J específicamente la instrucción jump, prácticamente lo único que necesitamos de esta instrucción es que se active y prepare el salto hacia el #target por lo que no guardaremos nada en ningún lugar, entonces desactivamos prácticamente todos los demás módulos para evitar un problema o que se llegara a guardar basura en algún dato o registro no deseado, únicamente dejando activa la señal jump para hacer el salto.

<pre> 1 module UC ( 2     input [5:0] OpCode, 3 4     output reg MemToReg, 5     output reg MemToRead, 6     output reg MemToWrite, 7     output reg [2:0] AluOp, 8     output reg RegWrite, 9 10    output reg RegDst, 11    output reg Branch, 12    output reg ALUSrc, 13    output reg Jump 14 ); </pre>	<pre> 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 </pre>	<pre> //JUMP 6'b000010:     begin         MemToReg = 1'b0;         MemToRead = 1'b0;         MemToWrite = 1'b0;         AluOp = 3'b000;         RegWrite = 1'b0;          RegDst = 1'b0;         Branch = 1'b0;         ALUSrc = 1'b0;         Jump = 1'b0;         Jump = 1'b1;     end </pre>
--	--	---

## B. Buffer ID/EX:

Si seguimos el diagrama del datapath nos daremos cuenta que las nuevas señales pasan por 2 buffers diferentes comenzando con el buffer ID/EX, esta es la primera definición de dicho pipeline (únicamente con los añadidos).

```
21      input in_UC_MuxJumper_Jump,
22      input [31:0] in_Shift_MuxJumper,

40      output reg out_UC_MuxJumper_Jump,
41      output reg [31:0] out_Shift_MuxJumper

59      out_UC_MuxJumper_Jump <= in_UC_MuxJumper_Jump;
60      out_Shift_MuxJumper <= in_Shift_MuxJumper;
```

## C. Buffer EX/MEM:

Esta es la segunda definición, para el buffer EX/MEM (el tercer buffer) con esto la instrucción jump funcionara correctamente y brincara en el momento y lugar adecuado.

```
18      input in_UC_MuxJumper_Jump,
19      input [31:0] in_Shift_MuxJumper,

33      output reg out_UC_MuxJumper_Jump,
34      output reg [31:0] out_Shift_MuxJumper

48      out_UC_MuxJumper_Jump <= in_UC_MuxJumper_Jump;
49      out_Shift_MuxJumper <= in_Shift_MuxJumper;
```

## D. ALU control

Para la ALU control añadimos señales que irán a la unidad de control que hacen referencia a las nuevas instrucciones de tipo R que añadimos en esta fase 3.

```
42      //mul
43      6'b000010:
44      begin
45          Op = 4'b0011;
46      end
47      //mod (DIVU)
48      6'b011011:
49      begin
50          Op = 4'b0100;
51      end
52      //sqrt (RSQRT)
53      6'b010110:
54      begin
55          Op = 4'b0101;
56      end
```

## E. ALU:

Esta fue la definición que utilizamos para nuestras operaciones dentro de la unidad lógica aritmética, la instrucción RSQRT se encarga de obtener la raíz cuadrada de un número, sin embargo, hacer este proceso por métodos convencionales nos llevaría varios problemas respecto a los ciclos de reloj. Para ello preferimos realizar una tabla de consulta (LUT) con el objetivo de obtener las raíces aproximadas en enteros de una forma rápida, pero ocupando una mayor cantidad de memoria.

```
43 //MUL
44 4'b0011:
45 begin
46     Res = Op_1 * Op_2;
47 end
48 //DIVU
49 4'b0100:
50 begin
51     Res = Op_1 % Op_2;
52 end
53 //RSQRT
54 4'b0101:
55 begin
56     case(Op_1)
57         1: Res = 1;
58         2, 3: Res = 1;
59         4, 5, 6, 7, 8: Res = 2;
60         9, 10, 11, 12, 13, 14, 15: Res = 3;
61         16, 17, 18, 19, 20, 21, 22, 23, 24: Res = 4;
62         25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35: Res = 5;
63         36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48: Res = 6;
64         49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63: Res = 7;
65         64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80: Res = 8;
66         81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99: Res = 9;
67         100: Res = 10;
68         default: Res = 0;
69     endcase
70 end
```

### - Problemas en el desarrollo y solución:

Al hacer algunas pruebas en el desarrollo del MIPS32 tuvimos algunos problemas que con ingenio pudimos solucionar y que el programa siguiera funcionando correctamente.

## A. Timing hazards:

En la fase de pruebas nos encontramos un problema al realizar operaciones similares a estas:

AND \$10, \$10, \$5

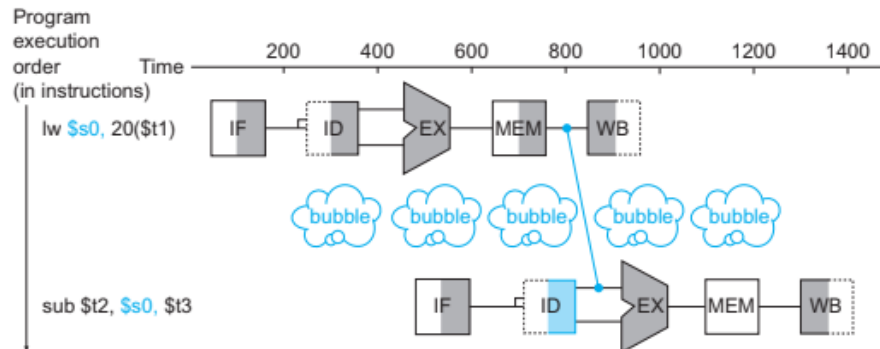
En ocasiones al hacerlo una única vez no habia problemas, pero al repetir este proceso con otras instrucciones como en este ejemplo:

AND \$10, \$10, \$5

AND \$10, \$10, \$4

Resultaba en un problema bastante curioso en el que el dato no se guardaba correctamente, sino que estaba guardando el dato anterior a este (el dato de la primera instrucción a pesar de que el dato de la segunda instrucción ya había salido de la ALU).

Al leer un poco el libro nos dimos cuenta que esto era un problema de hazards en los que la instrucción para escribir el dato no alcanzaba a escribirse en el banco de registros



Para solucionar esto como el libro lo menciona, debemos añadir burbujas entre las instrucciones para que el dato se alcance a guardar y después podamos recuperarlo para operar con este mismo.

Para solucionar este problema se necesitaron añadir como máximo 5 operaciones NOPs para limpiar todas las señales y que el dato pueda ser leído correctamente.

#### B. Burbuja para tipo I:

Curiosamente lo mismo sucedía con las instrucciones inmediatas incluso aunque en el grupo de instrucciones no se estuvieran guardando en el mismo lugar (para todas) con unas cuentas pruebas logre capturar que únicamente son necesarias 2 NOPs entre cada instrucción inmediata que opere algún valor como ADDI, ORI, ANDI, pero para la instrucción BEQ fue necesarios el uso de 3 NOPs.

#### C. Burbuja para tipo J:

En las tipo J al igual que las BEQ fueron necesarias 3 instrucciones NOPs esto para que se realizara el salto en el momento correcto (porque en las 3 instrucciones siguientes no hacía el salto) de no ponerlos el programa seguiría ejecutando instrucciones, pasando 3 instrucciones realizaría el salto en el siguiente ciclo de reloj cosa que nosotros no estamos buscando (mismo caso con BEQ).

#### D. Problemas para invertir el arreglo iterativo:

Este punto es únicamente para mostrar que por mas que intentábamos invertir un arreglo de forma iterativa se nos fue imposible porque dentro de una arquitectura y set de instrucciones no podemos saber el numero de dirección de un x dato, es decir no tenemos una forma directa de guardar el valor de un apuntador en un banco de registros.

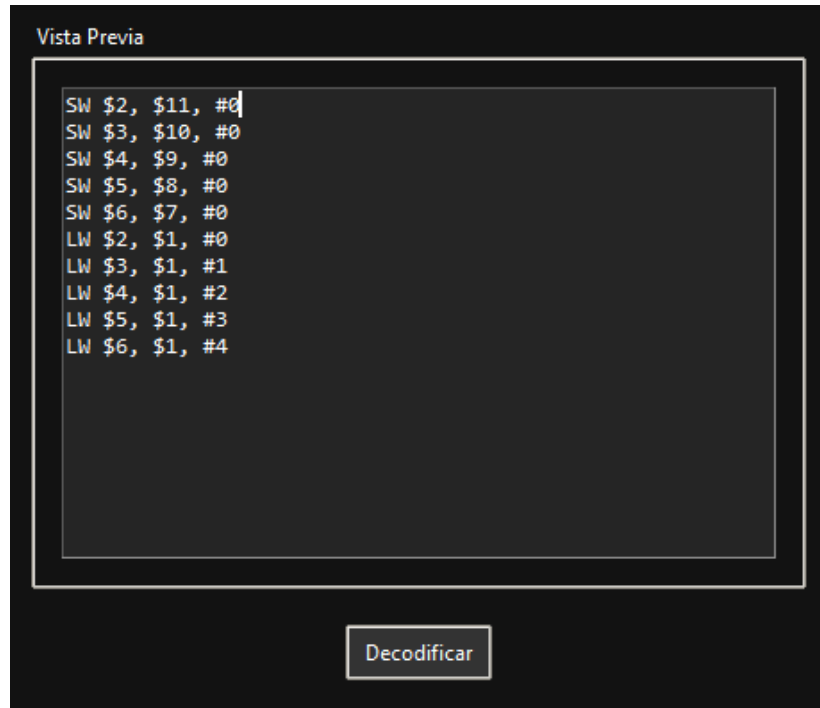
Entonces esta es una captura de algunas pruebas que hicimos en las que estamos haciendo una corrida en el escritorio de este algoritmo.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	Dirección	2	3	4	5	6	25	26	27	28			1
2													
3	Valores	1	2	3	4	5	2	5	6	3			
4		\-----/					\-----/		\-----/		\-----/		
5				a			i	j	n	k			
6							puntero a br						
7							(no existe puntero a br dentro de br)					a[2]=b[5]	
8	1 Inst	sw \$2, \$26, #0											
9	2	addi \$26, \$26, #-1											
10	3	addi \$25, \$25, #1											
11													
12	4	beq \$25, \$27, #1											
13	5	jump #1											
14													
15													
16													
17		x \$25,\$10											

Por lo que al no tener manera de encontrar este dato imprescindible para este método lo decidimos hacer de la manera hardcodeda, se nos hace un poco mas fea y robusta, pero funciona correctamente siempre y cuando si se agranda o minimiza el arreglo se tome en consideración para las instrucciones.

- *Construcción case 1:*

En el punto anterior observamos que no se nos fue posible invertir un arreglo de forma iterativa, si bien la idea ya estaba casi concreta la nula existe de una forma de conocer la ubicación en la que se está guardando el siguiente dato a invertir nos obligó a usar métodos más convencionales:



```
Vista Previa
SW $2, $11, #0
SW $3, $10, #0
SW $4, $9, #0
SW $5, $8, #0
SW $6, $7, #0
LW $2, $1, #0
LW $3, $1, #1
LW $4, $1, #2
LW $5, $1, #3
LW $6, $1, #4

Decodificar
```

Este programa en ensamblador da la solución al algoritmo para invertir un arreglo de 5 números, honestamente es un algoritmo muy simple lo que hace principalmente es guardar el arreglo original (ubicado de \$2 hasta \$6) en la memoria de datos, las direcciones que van desde la \$11 hasta la \$7 nos demuestran porque el programa iterativo no funcionaba sin hardcodear las direcciones, nosotros no podemos recorrer la dirección del registro que estamos tomando para recorrer el arreglo.

Porque con nuestra arquitectura no hay manera de diseñar apuntadores que guarden una dirección de memoria dentro de un registro.

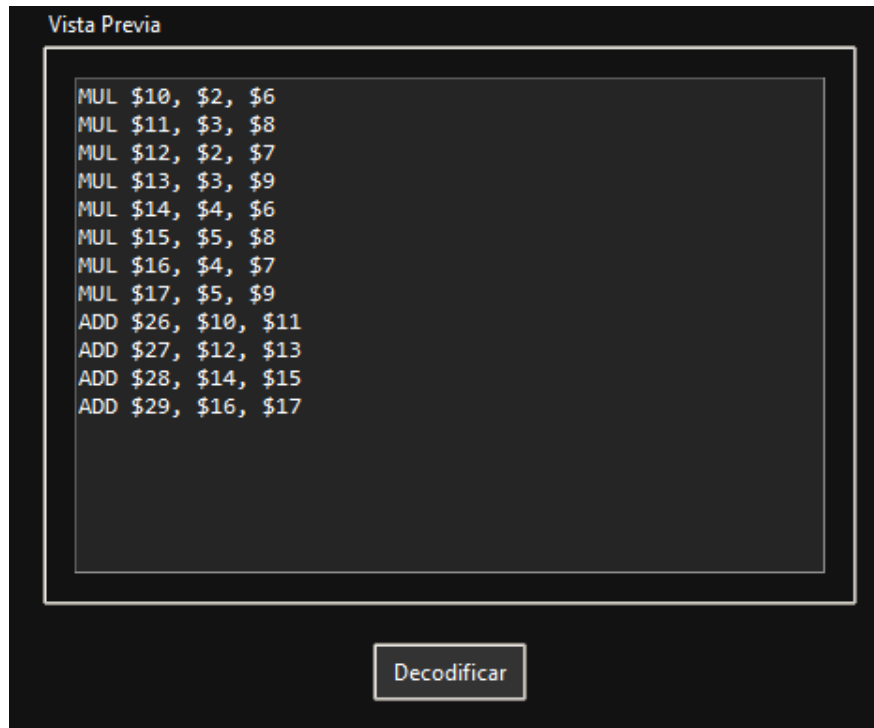
Bueno, una vez en el banco de registros regresamos el arreglo con una instrucción LW (ya se encuentra invertido en la memoria de datos) calculando la dirección relativa con un offset.

De esta forma dividiremos el banco de registros para el case 1:

Case 1 invertir arreglo					
Address	Dato				
\$0	0	Zero			
\$1	1	Base pointer			
\$2	5				
\$3	6				
\$4	7	Arreglo a invertir (aquí se guarda ya invertido)			
\$5	8				
\$6	2				
\$7	1				
\$8	2				
\$9	3	Direcciones del arreglo a invertir			
\$10	4				
\$11	5				
\$12					
\$13					
\$14					
\$15					
\$16					
\$17					
\$18					
\$19					
\$20	1	Selector			
\$21	1	Case 1			
\$22	2	Case 2			
\$23	3	Case 3			
\$24					
\$25					
\$26					
\$27					
\$28					
\$29					
\$30					
\$31					

- *Construcción case 2:*

Para construir este algoritmo honestamente fue muy sencillo porque ya hay muchas maneras de saber los datos que se toman para las multiplicaciones de matrices, en un principio pensábamos hacer esta multiplicación con una matriz de 3x3, pero nos dimos cuenta que el uso de registros e instrucciones aumentaba considerablemente en comparación a una multiplicación de matrices de 2x2, por lo que por temas de tiempo y para ejemplificar esto decidimos hacerla con este enfoque:



Por fortuna ya contamos con formulas para realizar esto entonces la solución es la siguiente: primeramente, multiplicamos la matriz A con la matriz B de tal forma que se multipliquen filas por columnas. Guardando en el mismo lugar en forma de suma el producto de los elementos relacionados con la fila por las columnas (en este caso 2 columnas, por lo tanto 2 productos que se están sumando). Una vez hecho esto realizamos la suma y guardamos los datos de tal forma que coincidan con la matriz principal y listo, la matriz C representara esta multiplicación.

$$\begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \times \begin{pmatrix} a_2 & b_2 \\ c_2 & d_2 \end{pmatrix} = \begin{pmatrix} a_1a_2 + b_1c_2 & a_1b_2 + b_1d_2 \\ c_1a_2 + d_1c_2 & c_1b_2 + d_1d_2 \end{pmatrix}$$



De esta forma dividiremos el banco de registros para el case 2:

Case 2 multiplicación matriz 2x2			
Address	Dato		
\$0	0	Zero	
\$1	1	Base pointer	
\$2	5	a1	
\$3	6	b1	Matriz 1
\$4	7	c1	
\$5	8	d1	
\$6	2	a2	
\$7	1	b2	Matriz 2
\$8	2	c2	
\$9	3	d2	
\$10	4	a1a2	
\$11	5	b1c2	
\$12		a1b2	
\$13		b1d2	Resultado Multiplicaciones
\$14		c1a2	
\$15		d1c2	
\$16		c1b2	
\$17		d1d2	
\$18			
\$19			
\$20	2	Selector	
\$21	1	Case 1	
\$22	2	Case 2	
\$23	3	Case 3	
\$24			
\$25			
\$26		a3	
\$27		b3	Matriz 3 resultante
\$28		c3	
\$29		d3	
\$30			
\$31			

- *Construcción case 3:*

El ultimo algoritmo y del cual estamos muy orgullosos es el algoritmo para saber si un numero se trata de un primo del 2 al 100, este algoritmo si lo pudimos hacer de forma iterativa de tal manera que el programa buscará con un contador que el dato sea primo (o que no lo sea) en cualquiera de los 2 casos el programa tomará una decisión y saldrá del programa con la decisión ya tomada:

```
Vista Previa
RSQRT $12, $2, $0
ADDI $3, $1, #0 //Inicialziar
NOP
NOP
ADDI $3, $3, #1
NOP
NOP
BEQ $2, $3, #18
NOP
NOP
NOP
DIVU $4, $2, $3
NOP
NOP
BEQ $0, $4, #11
NOP
NOP
NOP
BEQ $12, $3, #7
NOP
NOP
NOP
J #46 //salta a ADDI $3, $3, #1
NOP
NOP
NOP
SLT $10, $0, $4
```

Decodificar

Este algoritmo usa el método de encontrar si un numero es primo a partir de sus módulos tomando como limite la raíz cuadrada del número. El único caso a tomar en cuenta es cuando queremos saber si el 2 es numero primo porque el modulo de 2 con 2 es 0; dando como resultado 0. Pero 2 efectivamente es primo por eso hay un caso especial para esa excepción.

De esta forma dividiremos el banco de registros para el case 3:

Case 3 saber si un número es primo		
Address	Dato	
\$0	0	Zero
\$1	1	Base pointer
\$2	5	Es primo?
\$3	6	Contador i
\$4	7	Módulo de \$2 (resultado)
\$5	8	
\$6	2	
\$7	1	
\$8	2	
\$9	3	
\$10	4	1:Es primo, 0:No es primo
\$11	5	
\$12		
\$13		
\$14		
\$15		
\$16		
\$17		
\$18		
\$19		
\$20	3	Selector
\$21	1	Case 1
\$22	2	Case 2
\$23	3	Case 3
\$24		
\$25		
\$26		
\$27		
\$28		
\$29		
\$30		
\$31		

- Construcción switch y programa completo:

Casi por terminar lo que nos queda es diseñar la estructura switch y juntar todos nuestros algoritmos. Para lograr esto utilizamos una instrucción BEQ donde en el registro \$20 seleccionaremos el algoritmo que queramos utilizar:

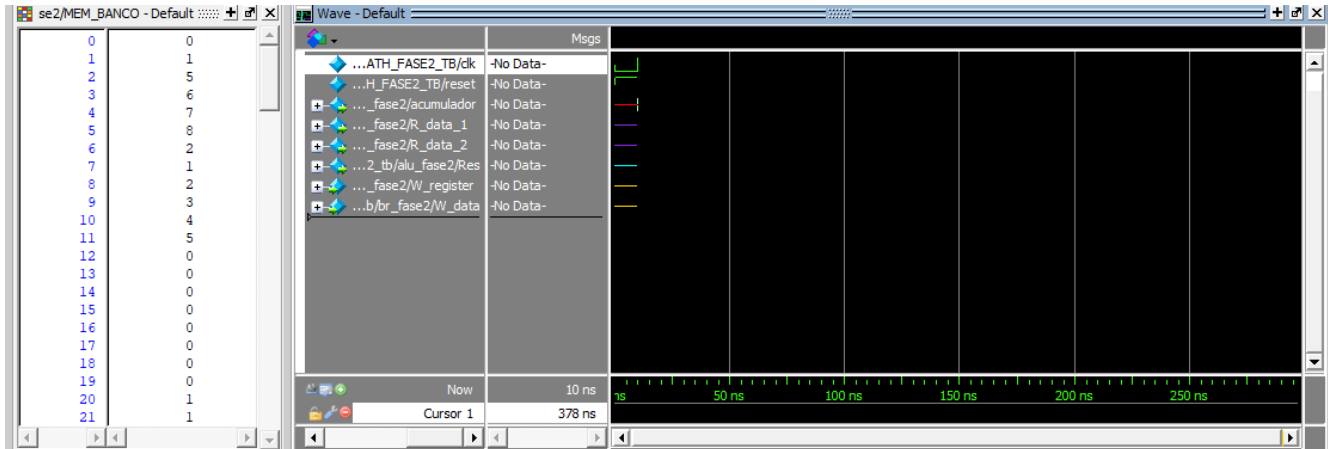
BEQ \$20, \$21, #11	NOP	NOP
NOP	MUL \$10, \$2, \$6 //BEQ2	NOP
NOP	MUL \$11, \$3, \$8	NOP
NOP	MUL \$12, \$2, \$7	DIVU \$4, \$2, \$3
BEQ \$20, \$22, #21	MUL \$13, \$3, \$9	NOP
NOP	MUL \$14, \$4, \$6	NOP
NOP	MUL \$15, \$5, \$8	BEQ \$0, \$4, #11
NOP	MUL \$16, \$4, \$7	NOP
BEQ \$20, \$23, #33	MUL \$17, \$5, \$9	NOP
NOP	ADD \$26, \$10, \$11	NOP
NOP	ADD \$27, \$12, \$13	BEQ \$12, \$3, #7
NOP	ADD \$28, \$14, \$15	NOP
SW \$2, \$11, #0 //BEQ1	ADD \$29, \$16, \$17	NOP
SW \$3, \$10, #0	J #100	NOP
SW \$4, \$9, #0	NOP	J #46 //salta a ADDI \$3, \$3, #1
SW \$5, \$8, #0	NOP	NOP
SW \$6, \$7, #0	NOP	NOP
LW \$2, \$1, #0	RSQRT\$12,\$2,\$0//BEQ3	NOP
LW \$3, \$1, #1	ADDI \$3, \$1, #0	NOP
LW \$4, \$1, #2	NOP	SLT \$10, \$0, \$4
LW \$5, \$1, #3	NOP	J #100
LW \$6, \$1, #4	ADDI \$3, \$3, #1	NOP
J #100	NOP	NOP
NOP	NOP	NOP
NOP	BEQ \$2, \$3, #18	

- Pruebas algoritmo completo y simulación:

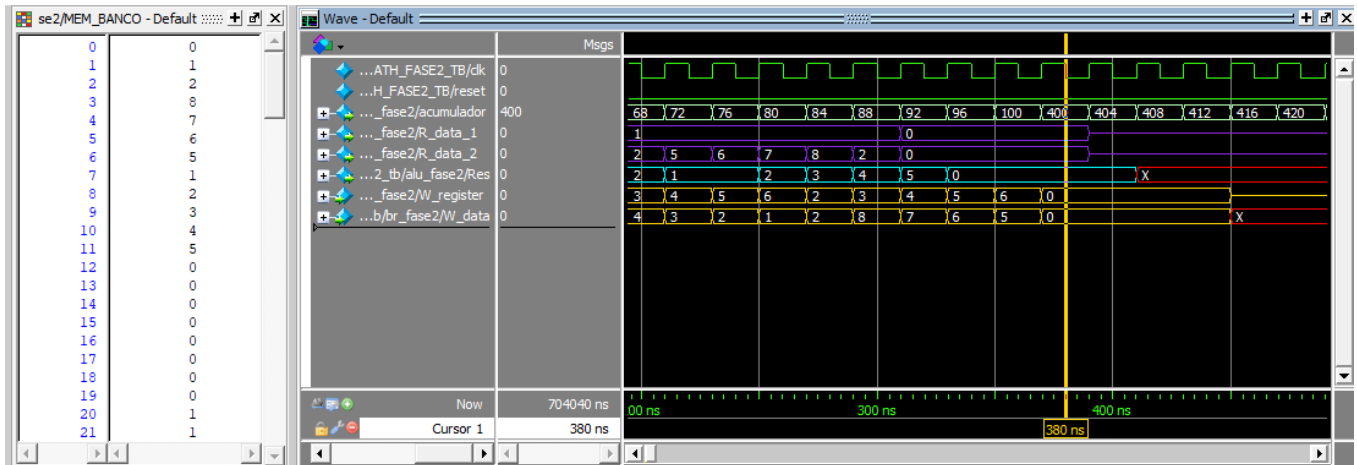
A. Pruebas case 1, \$20=1

Prueba 1: Invertir el arreglo {5,6,7,8,2}

Inicio de la simulación:



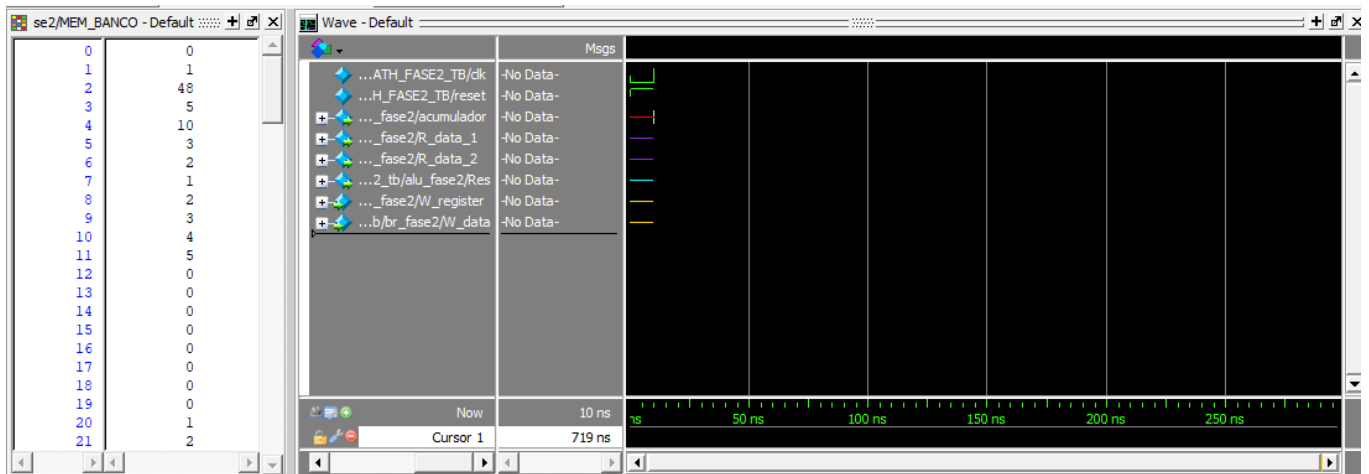
Fin de la simulación:



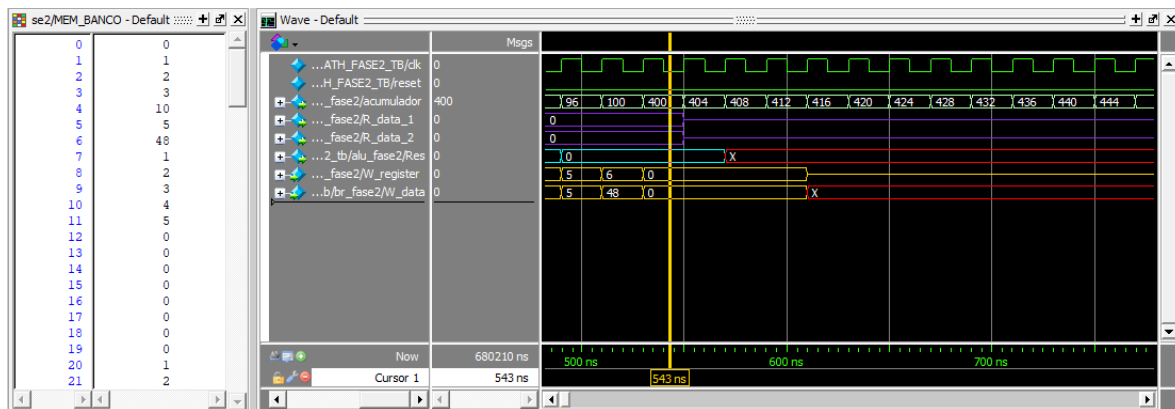
Funciona.

## Prueba 1: Invertir el arreglo {48,5,10,3,2}

Inicio de la simulación:



Fin de la simulación:



Funciona.

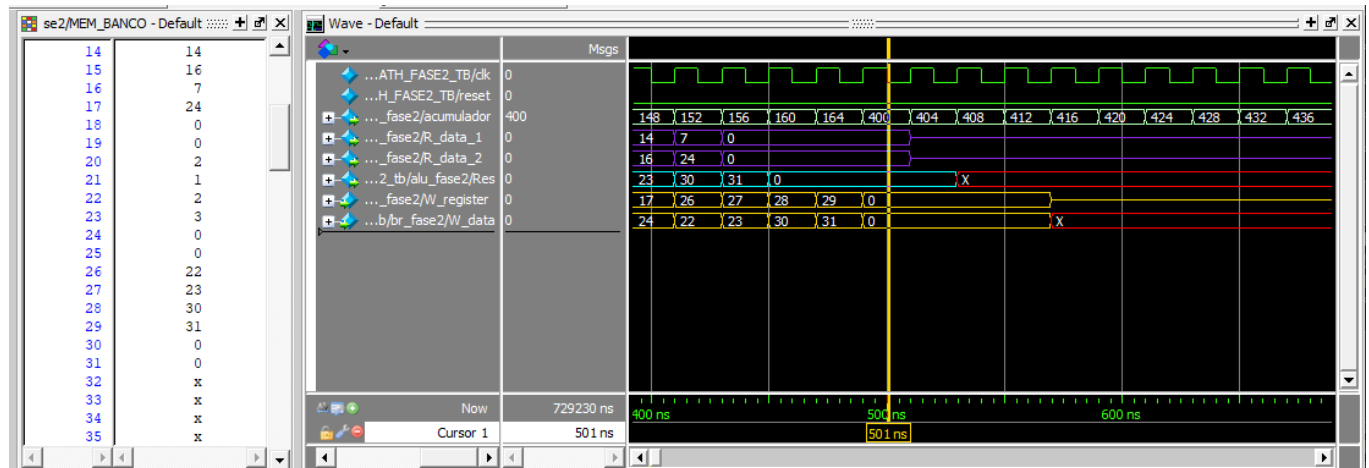
B. Pruebas case 2, \$20=2

Prueba 1: multiplicar  $\begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix} \times \begin{pmatrix} 2 & 1 \\ 2 & 3 \end{pmatrix}$

Inicio de la simulación:



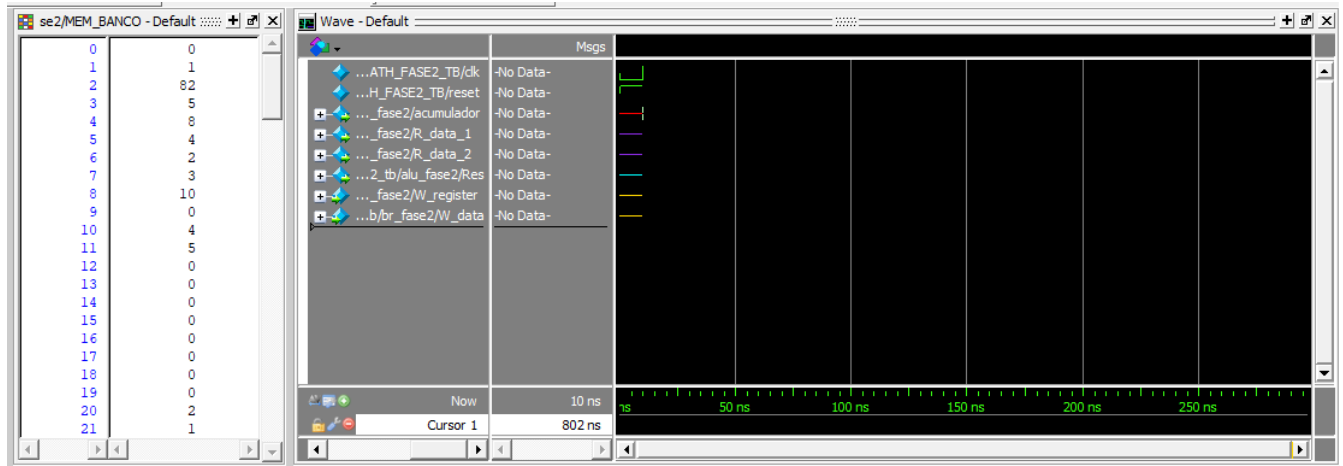
Fin de la simulación:



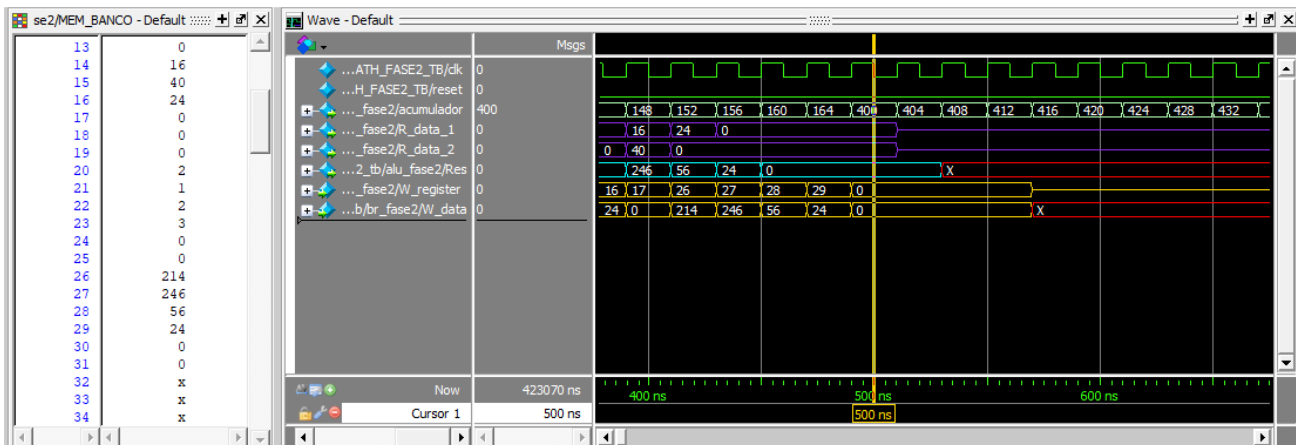
Funciona.

Prueba 2: multiplicar  $\begin{pmatrix} 82 & 5 \\ 8 & 4 \end{pmatrix} \times \begin{pmatrix} 2 & 3 \\ 10 & 0 \end{pmatrix}$

Inicio de la simulación:



Fin de la simulación:



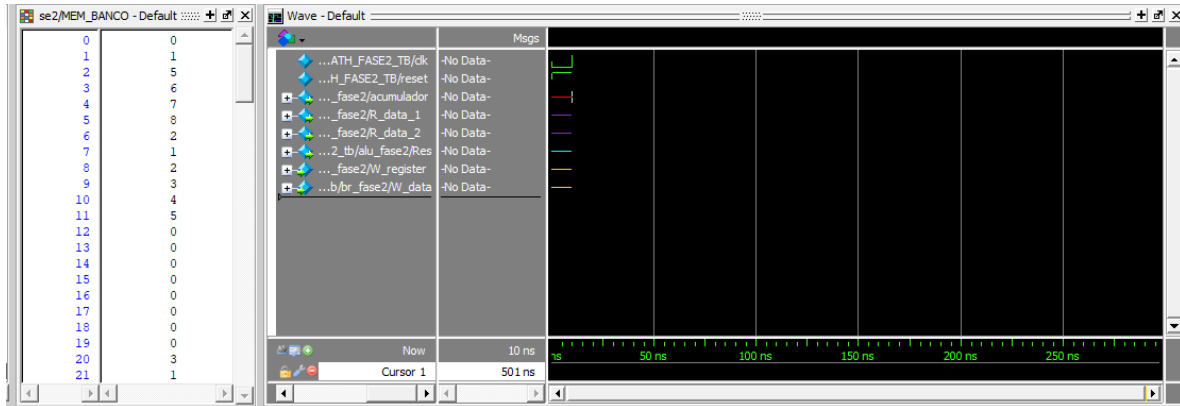
Funciona.



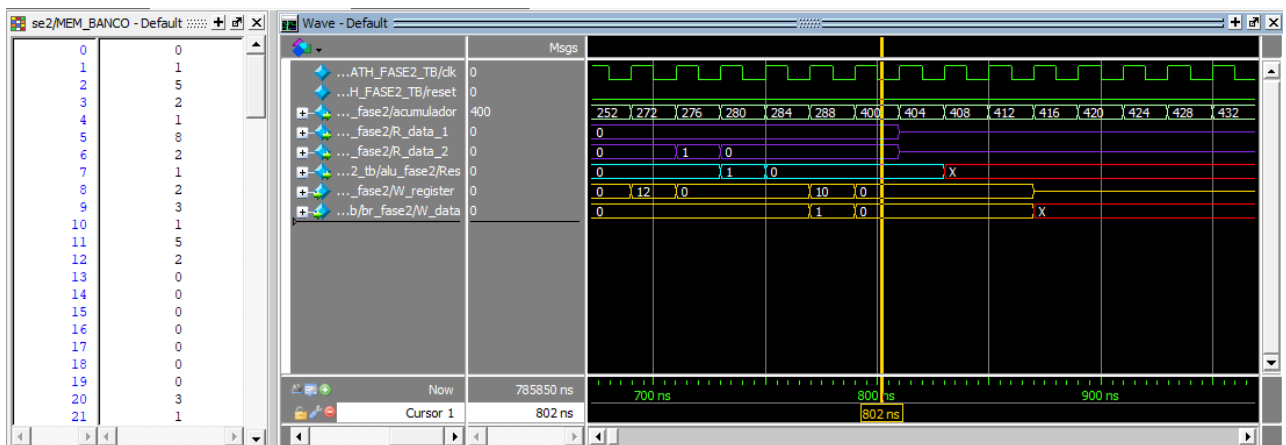
### C. Pruebas case 3, \$20=3

Prueba 1: 5 es primo?

Inicio de la simulación;



Fin de la simulación:



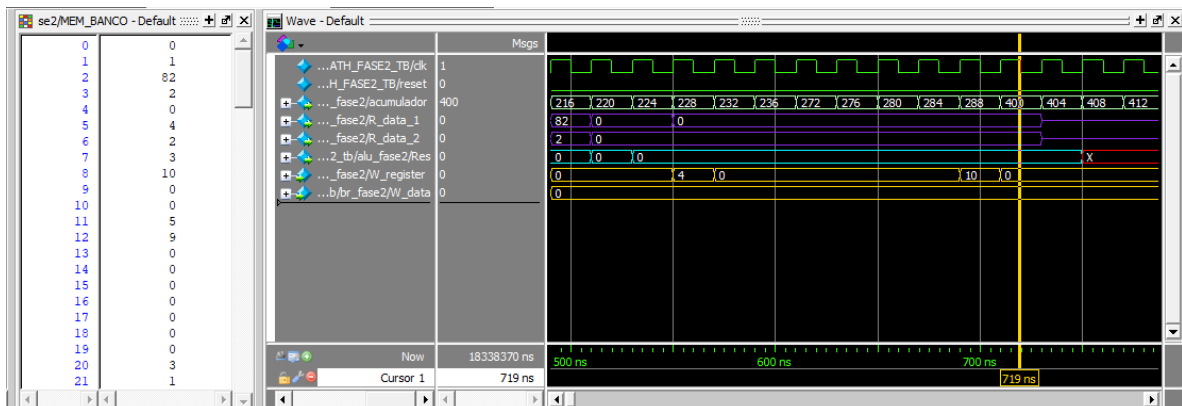
Funciona \$10=1.

Prueba 2: 82 es primo?

Inicio de la simulación:



Fin de la simulación:



Funciona, no es primo.

## **- Conclusión**

### **Conclusión fase 1:**

**Hugo Gabriel:**

Como equipo nos pareció una actividad muy agradable y que sobre todo nos ayuda de una manera muy positiva a darle un seguimiento estable a nuestro proyecto final pues de esta manera construimos de poco a poco el proyecto.

Tenemos que decir que encontramos algunas inconsistencias en el manejo de las instrucciones, pero esto es debido a que el proyecto aún no está completo y es por ello que duramente el proceso de creación de la fase 1 tuvimos ciertas complicaciones.

Pero al final nos sentimos satisfechos con nuestro trabajo y estamos completamente dispuestos a solucionar algunos de los pocos errores que tuvimos en la fase 2 de este proyecto donde añadiremos las instrucciones de tipo I.

**Oswaldo Maciel:**

Después de realizar varias actividades donde aprendimos a implementar distintos circuitos en Verilog, donde estábamos construyendo por piezas lo que sería nuestro proyecto final, por fin llegamos a implementar todos esos componentes en uno solo.

Sinceramente el desarrollo de la actividad fue muy sencillo, ya que habíamos logrado entender muy bien cada componente, y para este punto ya habíamos realizado un proyecto similar donde habíamos instanciado varios componentes con buffers (pipeline), por lo tanto, este trabajo fue sencillo ya que no había muchísimos cables ni conexiones complejas.

Fue muy entretenido realizar la actividad, aunque sinceramente quedé con dudas acerca del formato de las instrucciones ya que yo me dedicaba a crear los módulos y mi compañero a todo lo relacionado con las instrucciones, pero gracias a su ayuda, pude lograr comprender de igual manera este aspecto del proyecto.

### **Conclusión fase 2:**

**Hugo Gabriel:**

Honestamente la fase 2 se nos fue un poco complicada por la cantidad de módulos y cables que se añadieron al nuevo circuito, sin embargo, fue una actividad muy satisfactoria de terminar sobre todo cuando comenzamos a hacer pruebas con las instrucciones de tipo I, el ver que estas instrucciones funcionaban correctamente ha sido muy satisfactorio para nosotros, pero esperamos que la fase 3 no sea tan tediosa de realizar.

### **Oswaldo Daniel:**

Sinceramente fue muy difícil esta fase. En la fase anterior no habíamos implementado buffers, pero en este punto a pesar de que no eran obligatorios según los requerimientos de la actividad, ya eran necesarias para el DataPath, por lo tanto, comenzamos con la instancia. Lamentablemente no pudimos implementar la actividad de la manera correcta, ya que por la inmensa cantidad de cables debido a la necesidad el pipeline, no se realizó correctamente la instancia.

A pesar de los problemas, logramos realizar la correcta implementación de las instrucciones tipo I pero aún había problemas.

### **Conclusión fase 3:**

#### **Hugo Gabriel:**

En comparación con la fase 3 consideramos que fue muchísimo mas simple de implementar en nuestro MIPS32, honestamente donde mas nos tardamos fue en el diseño e implementación de los algoritmos porque el datapath lo obtuvimos a las horas de ponernos a trabajar. Sin embargo, no podemos negar que fue una actividad muy entretenida y divertida de hacer sobre todo por las grandes posibilidades de cosas que este circuito si bien es cierto que aún faltan muchísimos más módulos, instrucciones, señales, etc. Fue un gran proyecto y del cual me quedo muy satisfecho.

#### **Oswaldo Maciel:**

Por fin llegamos a la fase final, donde después de mucho esfuerzo, por fin quedaría implementado todo lo que realizamos. Seré sincero, comenzamos mal, ya que veníamos arrastrando un error desde la fase 2 y era que las instrucciones tipo no se realizaban cuando se necesitaba, es por ello que tuvimos que leer acerca del funcionamiento del pipeline, y es por ello por lo que nos dimos cuenta de que en ciertas instrucciones necesitabas burbujas, para que las siguientes se realizaran correctamente. Definitivamente gracias a descubrir esto, fue que pudimos lograr la correcta implementación del datapath completo (y a 5 horas debuggeando).

Solo faltaba una cosa, los algoritmos. Decidimos implementar 3 algoritmos distintos en un menú, aunque al principio parecía imposible. Gracias a la cooperación y al trabajo en equipo, fue que pudimos encontrar las instrucciones adecuadas para poder implementar los algoritmos.

Definitivamente, podría decir que este ha sido mi curso favorito hasta el momento, agradezco a mi compañero Hugo por realizar esto juntos, y al Mtro. Jorge Arce por todas sus enseñanzas.

### **- Referencias:**

MIPS Technologies. (2016). \*MIPS® architecture for programmers volume II-A: The MIPS32® instruction set manual\*. MIPS Technologies, Inc. <https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-05.04.pdf>

Patterson, D. and Hennessy, J. (2014). Computer Organization And Design: The Hardware/Software Interface. Book Aid International. <https://www.cse.iitd.ac.in/~rijurekha/col216/edition5.pdf>

William, S. (2007). Organización y Arquitectura de Computadoras. Pearson Education. <http://biblioteca.univalle.edu.ni/files/original/c1b1f5d1c12abc60a246e2a0d784f7c9d163ee81.pdf>