2020

**Profesor:** López Arce Delgado Jorge Ernesto

**Alumnos:**

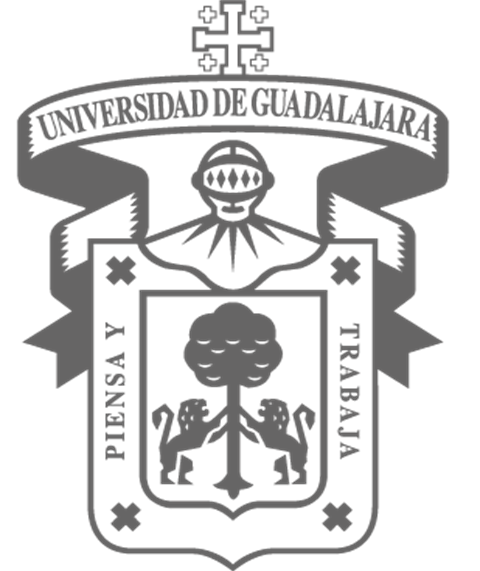
León García Daniel Ernesto

Torné Garza Ander

Pacheco Romero Victor Manuel

Cholico Torres Alejandro

D13



Proyecto Final -Mips

Índice

Introduccion…………………………………. 2-4

Objetivo……………………………………… 4

Fase 1……………………………………….. 5-13

Fase 2……………………………………….. 14-20

Fase 3……………………………………….. 21-23

Fase Final…………………………………… 24-33

Decoder……………………………………... 34-46

Números perfectos…………………………. 47

Programa Ensamblador…………………….48-50

Conclusion……………………………………51

Referencias…………………………………..51

Introducción

MIPS (Microprocessor without Interlocked Pipeline Stages) es una arquitectura diseñada para optimizar la segmentación en unidades de control y para facilitar la generación automática de código máquina por parte de los compiladores. MIPS es RISC (computador con repertorio de instrucciones reducido).

Cuenta con una segmentación de instrucciones: técnica de implementación de unidades de control que permite tratar las instrucciones en serie dividiéndolas en fases.

Con un único cauce de ejecución de instrucciones es posible mantener ejecutándose simultáneamente varias instrucciones, cada una en una fase distinta. Existen múltiples fabricantes de microprocesadores con arquitectura MIPS.

Las características del procesador MIPS32 son las siguientes:

* Máquina RISC (computador con repertorio de instrucciones reducido).
* Ancho de palabra y tamaño de los buses: 32 bits.
* Tamaño de los datos en las instrucciones:
* Byte (b): 8 bits
* Halfword (h): 16 bits
* Word (w): 32 bits
  + \*Doubleword (d): 64 bits
* Arquitectura de carga / almacenamiento:
  + \*Antes de ser utilizado en una instrucción aritmética, todo dato debe ser cargado previamente en un registro de propósito general.
  + \*Instrucciones aritméticas con 3 operandos de 32 bits en registros.
* Esquema de bus único para memoria y E/S.
* Modos de funcionamiento: usuario, núcleo (kernel), supervisor y depuración.

Es importante saber que, en MIPS, el compromiso elegido por los diseñadores de la arquitectura es guardar todas las instrucciones con la misma longitud, por lo que el número de bits en una instrucción de MIPS es siempre de 32, lo mismo que la longitud de una palabra.

A los campos MIPS se les da una serie de nombres para su rápida identificación:

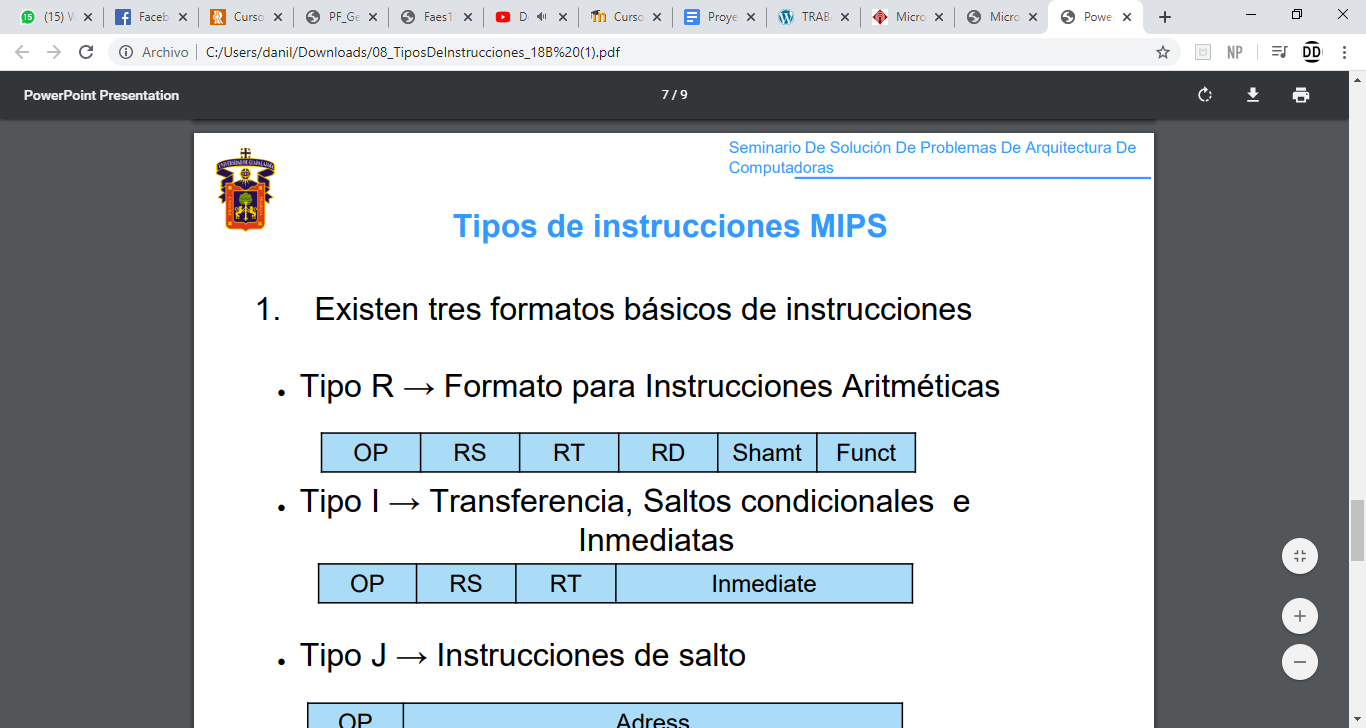
* op: operación básica, llamada “opcode”.
* rs: El registro origen para el primer operando.
* rt: El registro origen para el segundo operando.
* rd:El registro de destino, obtiene el resultado.
* shamt: Usado para instrucciones de desplazamiento.
* funct:Función, este campo (código de función) es usado para seleccionar la variante específica del “opcode”.

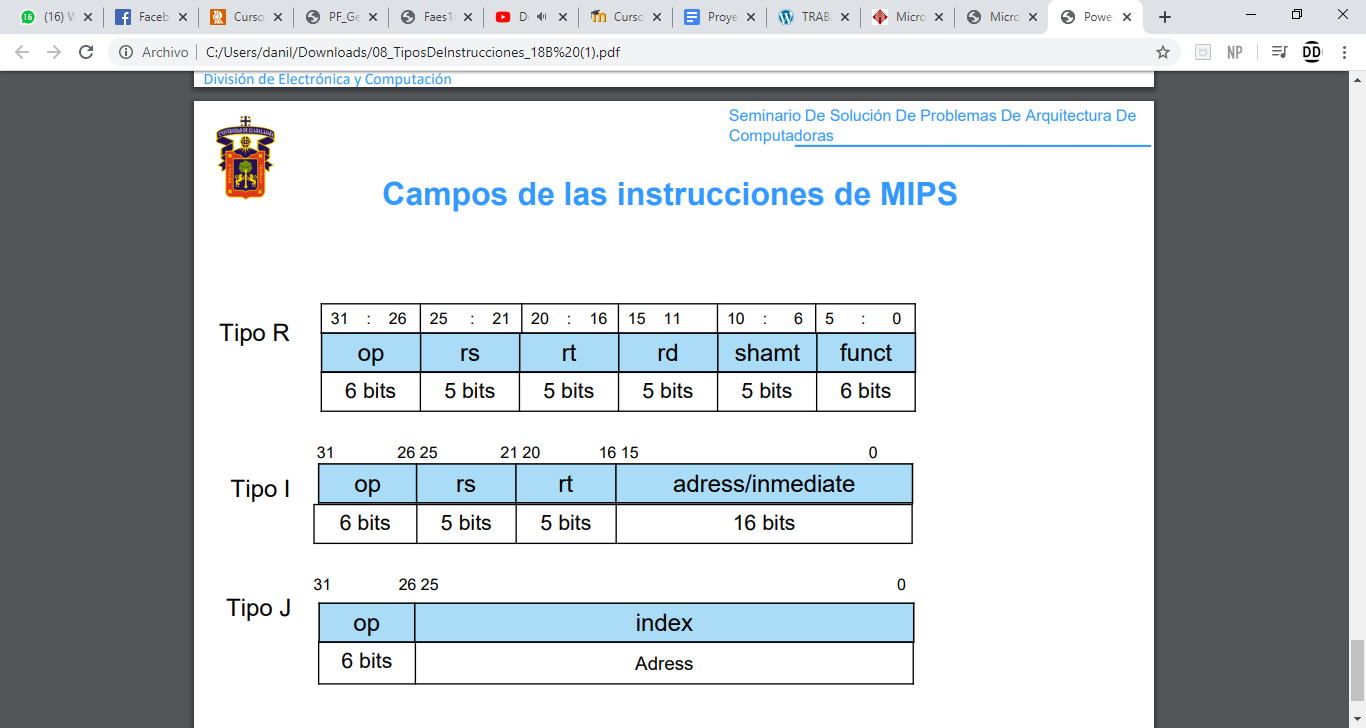
En MIPS se distingue tres clases de instrucciones:

* TIPO R
* TIPO I
* TIPO J

Tipo R:

Aritméticas-lógicas

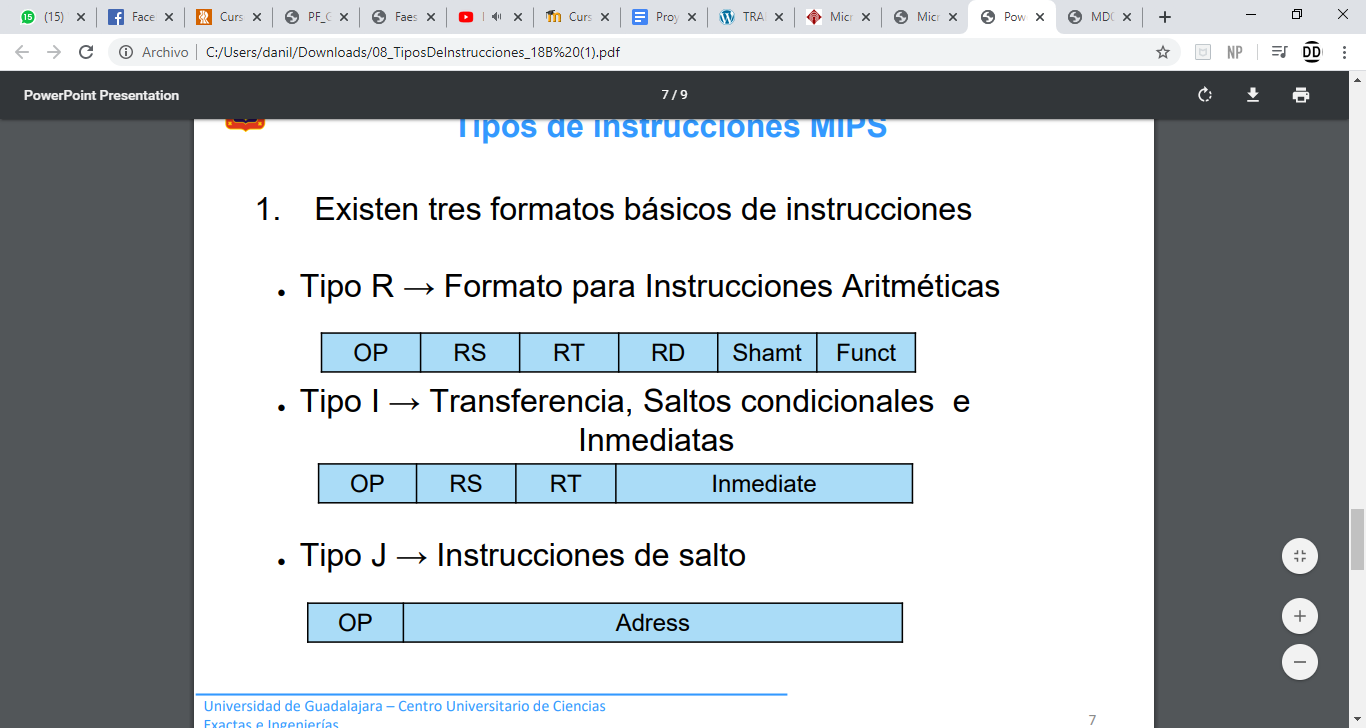


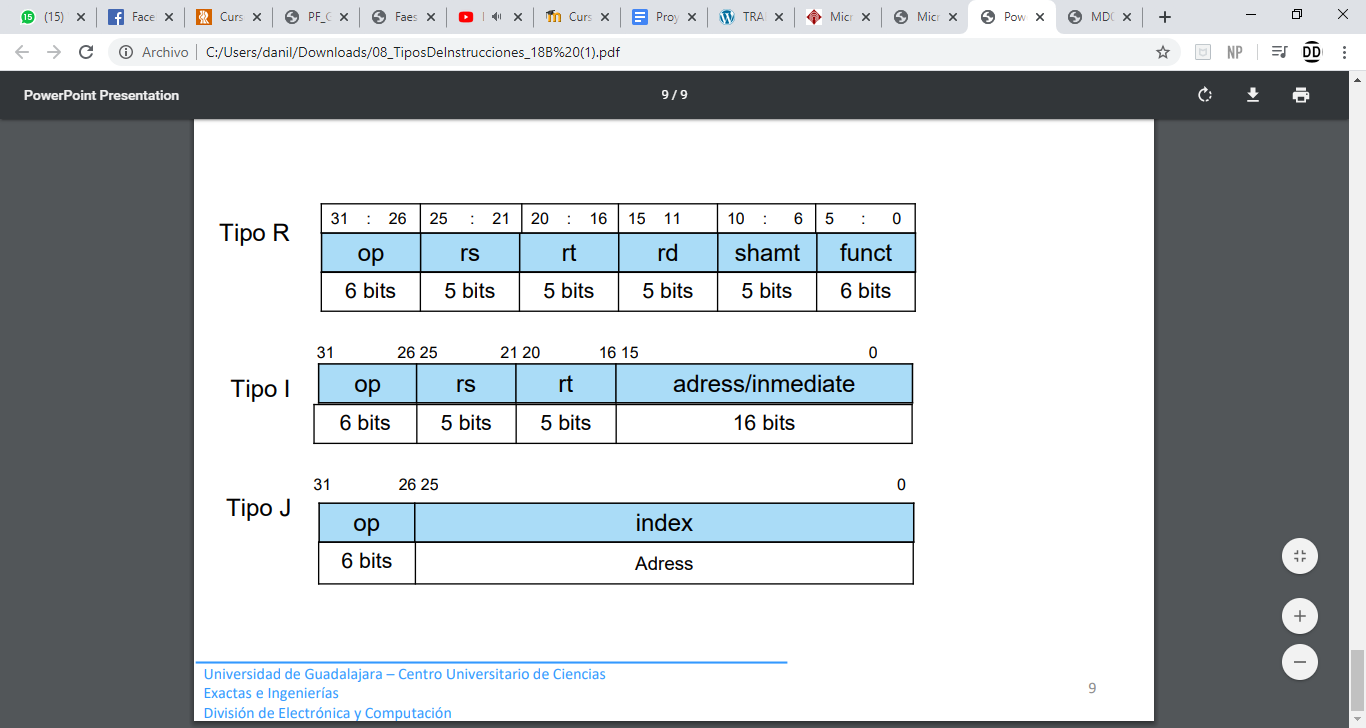


*Ilustración 1 Type R*

Tipo I:

Referencias a memoria, Aritméticas (Inmediato) y Salto condicional.

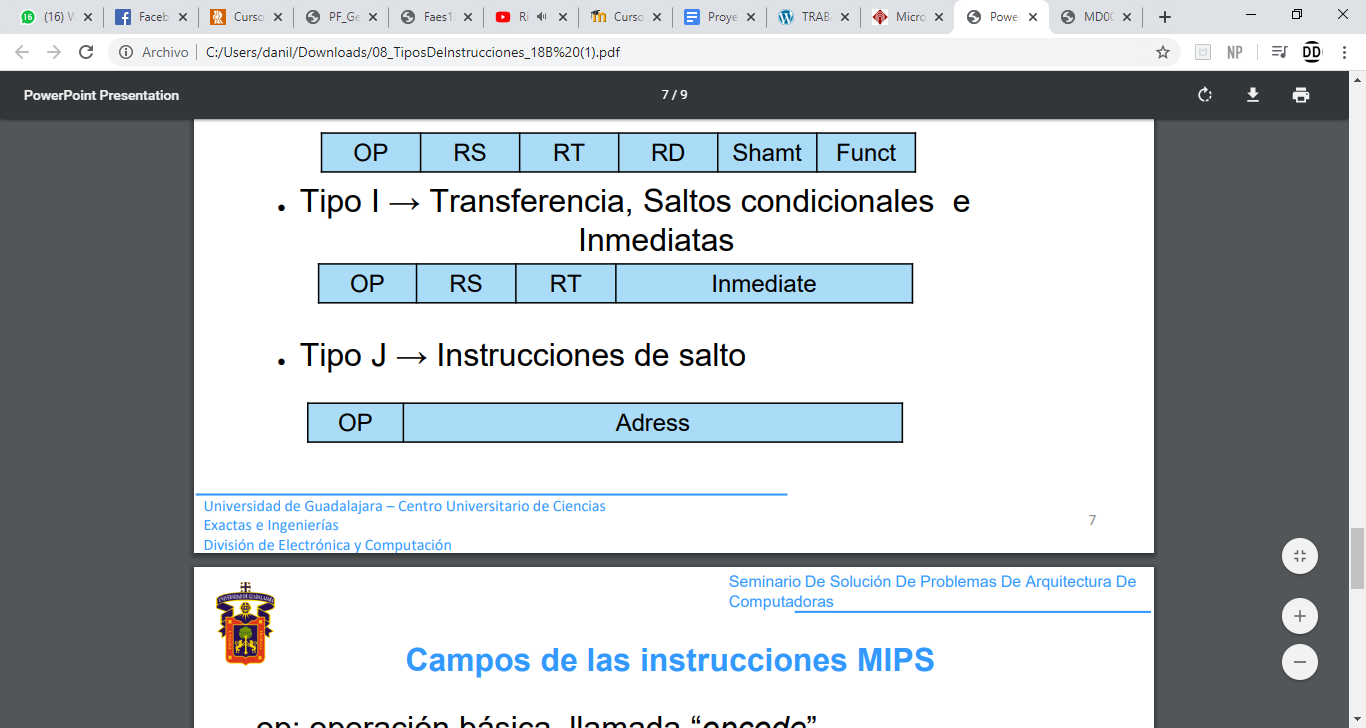


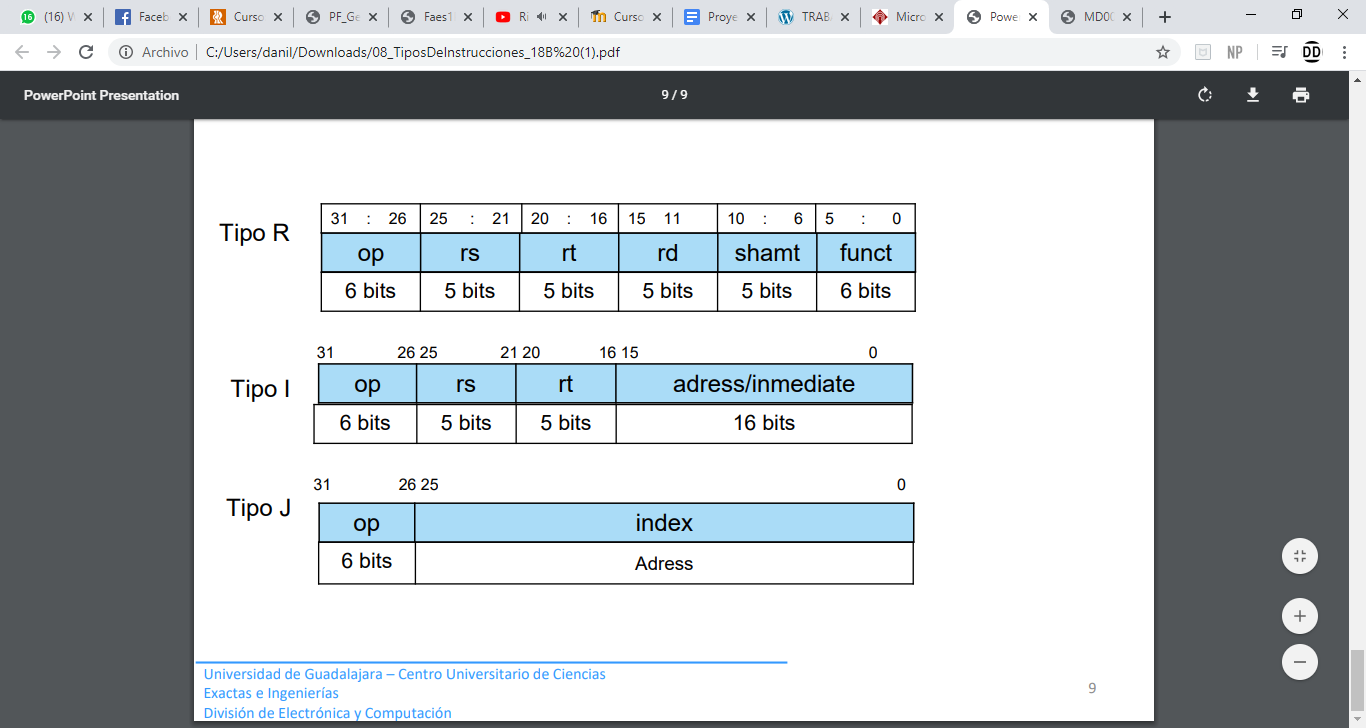


*Ilustración 2 Type I*

Tipo J:

Instrucciones de salto.





*Ilustración 3 Type J*

Objetivo

Desarrollar cada módulo que conforma la arquitectura mips y por supuesto instanciarlos todo, una vez instanciado este debe ser capaz de ejecutar instrucciones tipo R, I, J sin ningún problema, por ultimo desarrollar un programa en ensamblador en base alguna teorema o tema de matemáticas, este programa debería de poder correr en nuestro datapath de mips.

Desarrollo

Fase 1

En esta primera fase las operaciones que podrá realizar nuestro diseño son las operaciones aritméticas-lógicas. Sumas, restas como las instrucciones aritméticas y como las lógicas or y and.

**Tabla de instrucción Tipo R:**

Para que el datapath ejecute estas instrucciones, estas deben de tener el siguiente formato, donde:

* Opcode: Nos dice que tipo de instrucción es, en este caso tipo R.
* Rs: Este campo representa un operando. Este dato representa un registro del banco de registros para lo que allá guardado de este registro lo tomé para operarlo.
* Rt: Este campo representa un operando. Este dato representa un registro del banco de registros para lo que allá guardado de este registro lo tomé para operarlo
* Rd: Este cambo indica donde se tiene que guardar el resultado de esta operación.
* Shamt: Este campo sirve para hacer desplazamientos, pero no lo usaremos.
* Funct: Es la función de la instrucción, sirve para indicarle que va hacer, ya sea una suma, reta o and, cualquiera de las operaciones aritméticas lógicas.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Opcode** | **RS** | **RT** | **RD** | **Shamt** | **Funct** | **Operación** |
| **000000** | 00000 | 00001 | 10000 | 00000 | 100000 | Suma |
| **000000** | 00100 | 00010 | 10001 | 00000 | 100001 | Resta |
| **000000** | 00011 | 00011 | 10010 | 00000 | 100010 | Multiplicación |
| **000000** | 01110 | 00111 | 10011 | 00000 | 100011 | División |
| **000000** | 00010 | 00011 | 10100 | 00000 | 100100 | And |
| **000000** | 00011 | 00001 | 10101 | 00000 | 100101 | Or |
| **000000** | 00101 | 01010 | 10111 | 00000 | 100110 | Xor |
| **000000** | 00000 | 00000 | 00000 | 00000 | 000000 | Not |
| **000000** | 00101 | 00000 | 10101 | 00000 | 101010 | Stl |

|  |  |
| --- | --- |
| **Señales** |  |
| AluOp | 010 |
| RegDst | 1 |
| RegWrite | 1 |
| MemtoReg | 0 |
| AluSrc | 0 |
| Er | 0 |
| Ew | 0 |
| PcSrc | 0 |
| Jump | x |

Debido a que todas las operaciones usan el mismo opcode, la unidad de control lanza las mismas señales sin importar, que operación va hacer.

**Tabla Unidad de Control**

**Tabla de Unidad de Control Alu**

Dependiendo el Funct que recibe la unidad de control de la alu, le indicara a la alu que operación va hacer.

|  |  |  |
| --- | --- | --- |
| **Función** | **Selector** | **Operación** |
| 100000 | 0000 | Suma |
| 100001 | 0001 | Resta |
| 100010 | 0010 | Multiplicación |
| 100011 | 0011 | División |
| 100100 | 0100 | And |
| 100101 | 0101 | Or |
| 000000 | 0111 | Not |
| 101010 | 1000 | Stl |

**¿Cómo le hicimos y funciono?**

Primeramente, lo que hicimos fue probar si funcionaba con las operaciones básicas, lo que podemos ver en la primera instrucción, es que el function está indicando que se va a realizar una suma ya que el 100000 equivale a la suma. Los valores con los que esta se va a realizar son los que están en la dirección RS y RT, estos siendo 0 y 1 respectivamente, y la dirección en donde estos se van a guardar es donde el RD le indique, este siendo en la dirección 16 del banco de registros, el resultado de esta es uno y se puede comprobar en la imagen a continuación, en la segunda instrucción se realizará una resta de los números 4 y 2, el resultado de esta se guardará en la dirección 17 y tendrá como resultado 2. En la tercera instrucción podemos ver que se realizará una multiplicación de los números 3 y 3, el resultado de esta operación se guardará en la dirección 18 y tendrá como resultado el valor de 9. Por último, se realizará una división entre los números 14 y 7 que se guardará en la dirección 19 y esta tendrá como resultado el número 2. Todas estas operaciones las podemos verificar en la siguiente imagen.

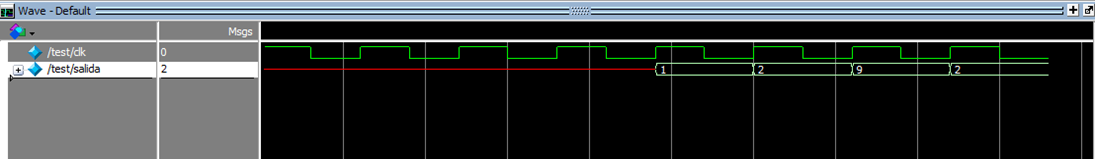


Ilustración 1 Prueba Fase 1

Estos son los datos que se obtienen con las otras cuatro operaciones usando las operaciones AND, OR, Slt y Nop

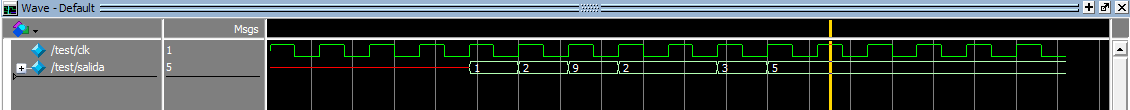


Ilustración 2 Pruebas fase 1

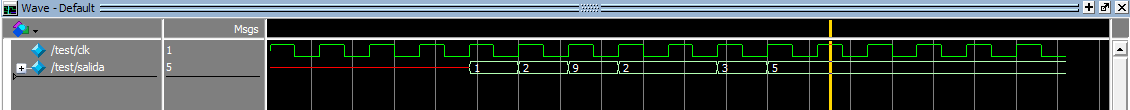
Y finalmente para comprobar que nuestra fase 1 estuviera correcta, utilizamos los archivos que usted envió para probarlos.

Ilustración 3 Prubas Fase 1

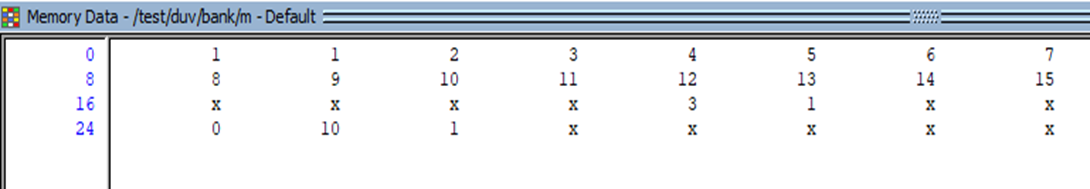


Ilustración 4 Memory List Fase 1

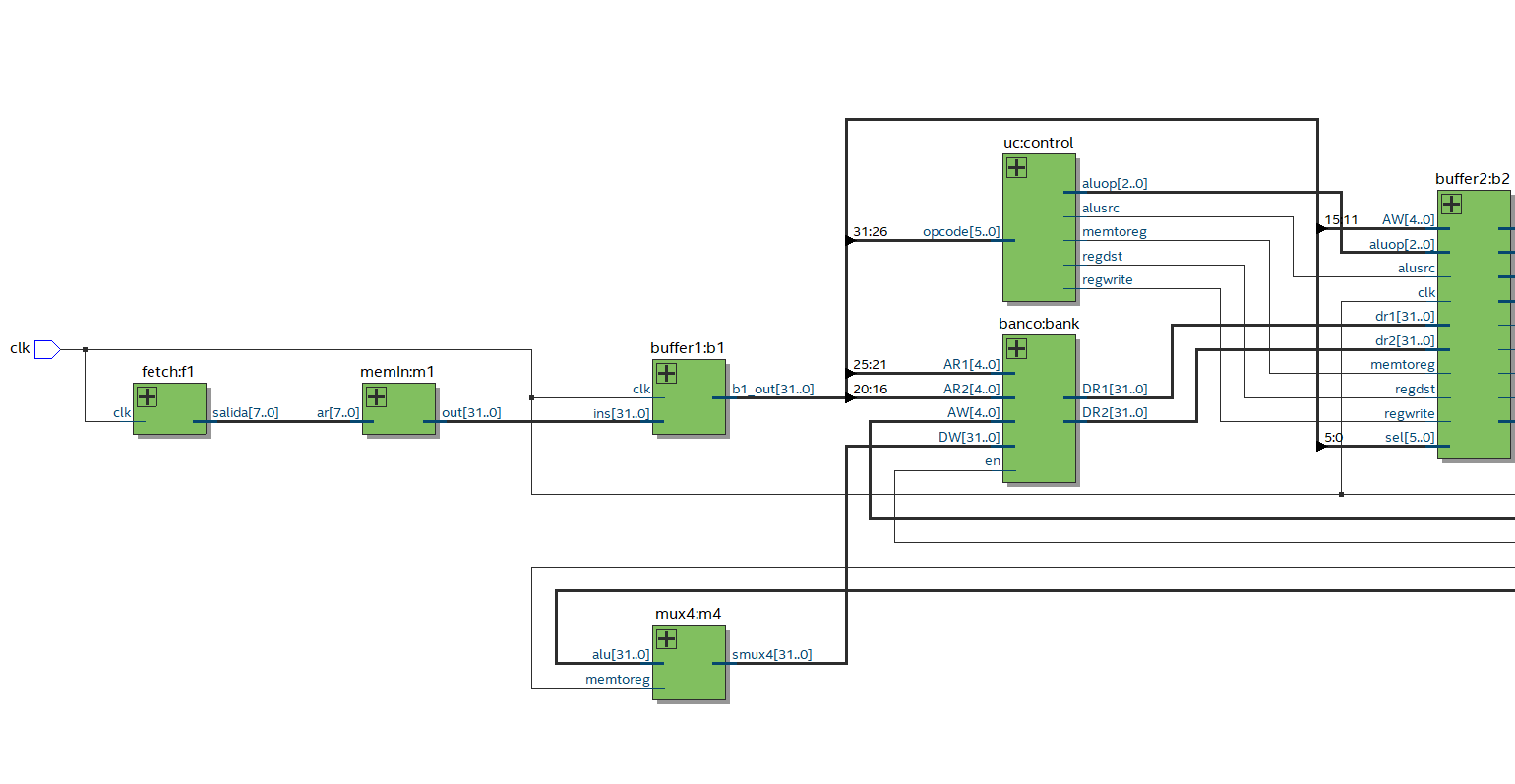


Ilustración 5 Circuito parte 1

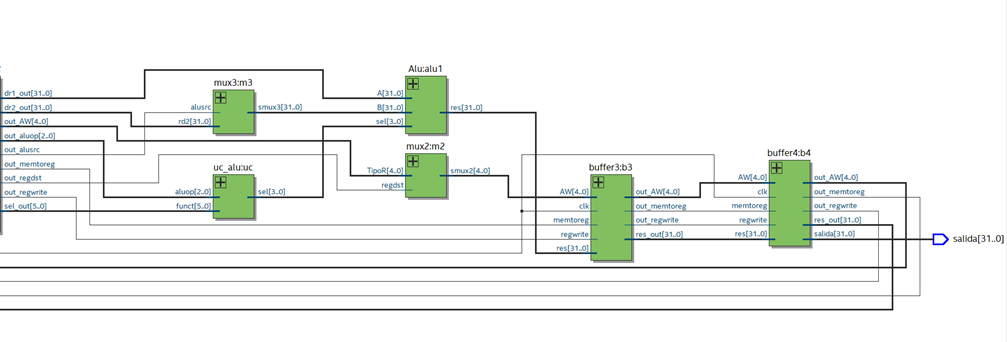


Ilustración 6 Circuito parte 2

**Resumen de esta fase y en que consiste**

La fase uno del proyecto consta de conjunto de módulos que, al conectarlos, estos tienen la capacidad de realizar instrucciones de tipo R, para que este funcione, es necesario implementar los siguientes módulos: el ciclo fetch, la memoria de instrucciones, un banco de registros, una unidad de control, una ALU control y una ALU. Además de ciertos multiplexores y buffers para que estos funcionen correctamente en conjunto. Primeramente, el módulo fetch consta de un contador de programa, un sumador y la memoria de instrucciones. El PC es una memoria de tamaño de palabra de 8 bits, este está inicializado en -4 para que en el primer ciclo este esté en 0 en el que cada que pase un ciclo el sumador le va a ir sumando 4, la memoria de instrucciones, como su nombre lo dice es una memoria en donde se van a guardar las instrucciones con las que se va a trabajar. Esta es una memoria de ancho de palabra de 8 bits y un largo de 129 bits. En esta memoria es donde se guardarán las instrucciones que vamos a ingresar, esto con un $readmemb en donde leerá un archivo de texto en donde están las instrucciones, como las instrucciones son de 32 bits la memoria guarda cada una de estas cada cuatro lugares por lo que en el ciclo fetch cada que pasa un ciclo se le suma cuatro. Después de esto la instrucción pasa por el primer buffer y de aquí esta se divide, los bits 31:26 se van a la unidad de control, aquí dependiendo del Opcode que sea, esta manda una señal a los módulos de la fase, del bit 25:21 estos siendo el RS se dirigen hacia el Read register 1 del banco de memoria donde más adelante se usarán, del bit 20:16 se van hacia el Read register 2 siendo este el RT y hacia un multiplexor, del bit 15:11 siendo este el RD se dirige hacia el multiplexor y dependiendo del tipo de instrucción que sea pasa un dato o el otro, en este caso como es una instrucción de tipo R, en este caso el RegDst de la unidad de control mandara un 1 por lo que los bits de RD pasarán al Write register. Por último los bits 5:0 se dirigirán hacia la ALU control, aquí la unidad de control mandara la señal de 3 bits 010 por lo que dependiendo del function de la instrucción, la ALU control convertirá la instrucción de 6 bits a una de 4 bits para que este pase a la ALU y le indique qué operación tiene que hacer. A continuación, podemos ver una tabla con las diferentes operaciones que se pueden realizar en la ALU Control. Después de esto obtendremos dos datos como “resultado” en el banco de registros, el read data 1 serán 32 bits que irán directamente a una entrada de la ALU, el dato de read data 2 ira hacia un multiplexor, que cuando este sea una instrucción de tipo R, el ALUSrc sea 0 pasará el valor del RD2 hacia la otra entrada de la ALU, después de esto dependiendo del dato que se obtenga del ALU control, este indicará a la ALU qué operación utilizar, como se observa en la tabla de arriba. Por último, el resultado de la operación realizado en la ALU pasa a otro multiplexor, que este al estar en 0, manda el dato hacia el banco de registros en donde se guardará dependiendo de la dirección que contuviera la dirección.

**¿Qué módulos hacen posible la magia y como se programaron?**

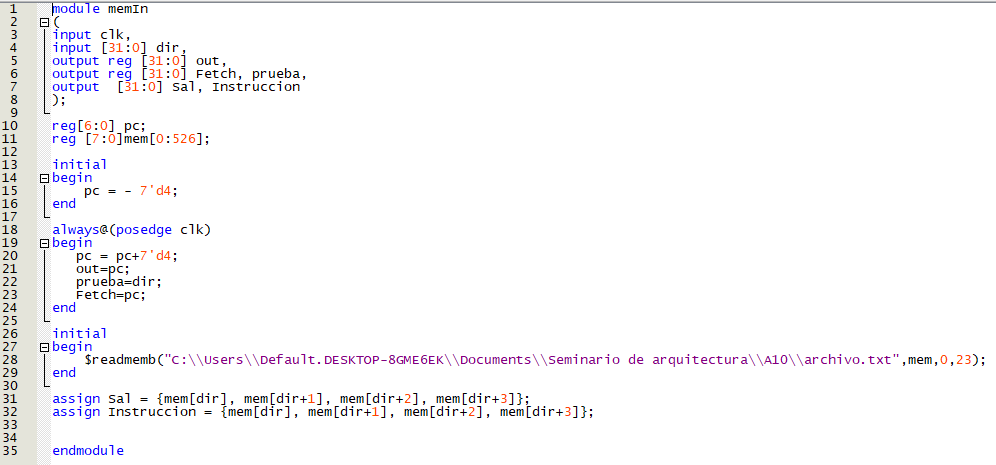


Ilustración 7 Ciclo Fetch/MemIn

Este módulo hacer varias funciones ya que es un ciclo fetch y memoria de instrucciones a la vez, por lo que su función es llevar a cabo el pipeline de todo el datapath y enviar las instrucciones a todo el datapth.

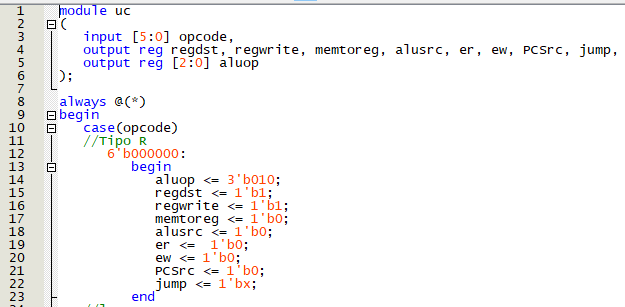


Ilustración 8 Unidad de Contol

Este módulo funciona con un case respondiendo al opcode y lanzando señales a los muxes y la unidad de control.

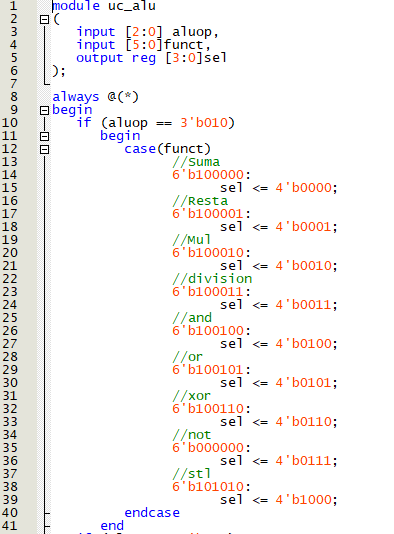


Ilustración 9 Unidad de control de la alu

Este módulo funciona con un if que responde a la señal enviada por la unidad de control y en caso de que se cumpla la condición ahora, dependiendo de que funct llegue, lanzará una señal selectora a la alu que le indicara que va hacer.

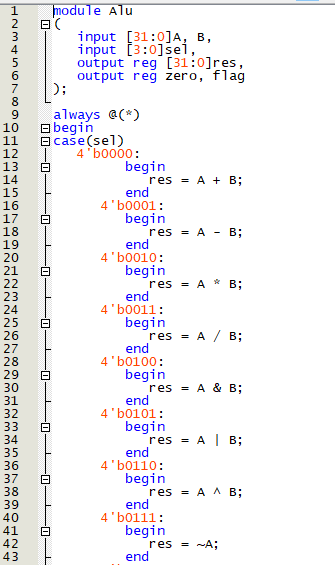


Ilustración 10 ALU

Mediante un case que responde a una señal selectora de 4 bits, dependiendo el caso realizara una operación, este módulo tiene integrado la suma, resta, multiplicación, división, and, or, xor, not y stl.

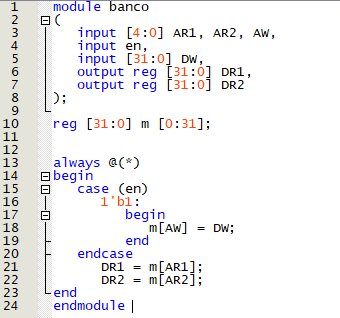


Ilustración 11 Banco

Aquí se van a guardar los datos de las instrucciones, también se toman los registos.

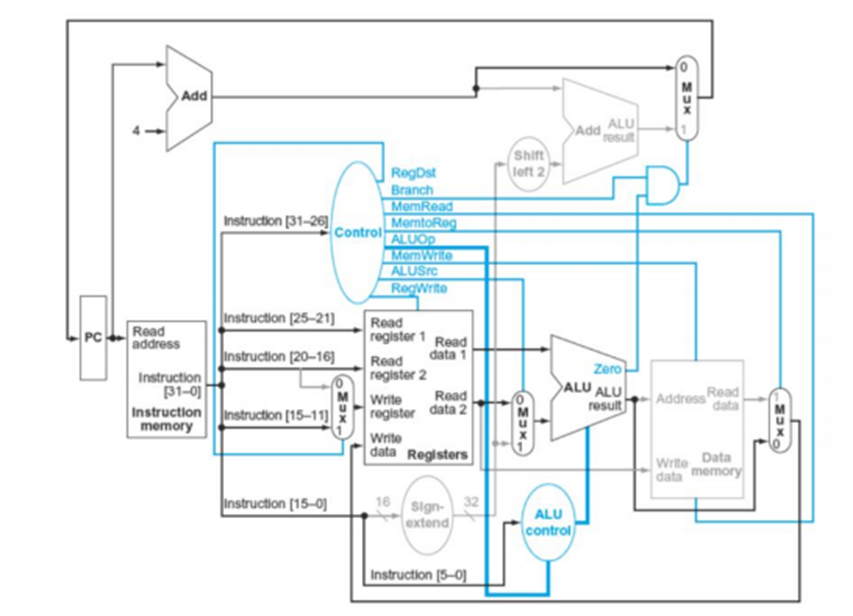


Ilustración 12 Datapth tipo R

Fase 2

Para la fase 2 del proyecto consta de un conjunto de módulos que hacen que el datapath pueda realizar instrucciones de tipo I, como pueden ser la LW y la BEQ. Para la realización de las instrucciones LW consta de los siguientes módulos: Ciclo fetch, Unidad de control, Banco de registros, ALU, Alu control y la memoria de datos. Lo primero es que se inicia con el pc que está con un -4 y cada vez que se repita el ciclo se le va a ir sumando 4, pasa por el sumador y por la memoria de instrucciones, al llegar al sumador lo que hace después de sumarle 4 se va al multiplexor que tiene la señal en 0 y regresa nuevamente al pc. Cuando está en la memoria de instrucciones sale y se dividen los bits en 4 partes diferentes, al pasar por el buffer los bits del 31-26 se van a la unidad de control, los bits de 25-21 se van al read register 1 del banco de registros, los bits del 20-16 se van al multiplexor que está con la señal en 0 y los pasa al write register del banco de registros, y los bits del 15-0 se van al sign-extend. Después obtendremos el “resultado” del banco de registros que es de 32 bits por lo que va directamente a la entrada de la ALU, por el otro lado los 15 bits que están en el sign-extend se convierten en la salida en 32 bits por lo que va directamente a un multiplexor que tiene la señal en 1 por la unidad de control y esto procede a la entrada de la ALU, el alu control mediante la señal que le manda la unidad de control puede saber cuál operación va a ser la que hará y lo que hace es que manda la señal a la ALU, el sign extend lo que hace es que cuando llegan los 16 bits del inmediato le agrega otros 16 bits pero de puros 0, lo que hace que siga siendo el mismo número pero con 32 bits. Después el “resultado” de la alu se va a la memoria de datos y de ahí sale por el read data donde pasa directamente al multiplexor, al estar en el multiplexor si la señal de la unidad de control es 1 manda directamente esos 32 bits hacia el write data del banco de registros. Para la realización de las instrucciones de tipo I como la BEQ consta de los siguientes módulos: ciclo fetch, unidad de control, banco de registros, sign extend, alu control, sumador, shift left 2.

Iniciamos con el ciclo fetch que ya sabemos cómo funcione al iniciarlo en -4 y luego le va sumando 4, lo único que cambia es que cuando va del pc al sumado donde se le va agregando 4 va directamente a la entrada del otro sumador y al multiplexor, si el multiplexor está en 0 pues pasa al pc de nuevo pero la unidad de control también manda la señal al 1 que después veremos qué hace, cuando entran los bits a la memoria de instrucciones mediante read address, sale con 32 bits y esos bits pasan al buffer donde se dividen los bits, del 31-26 se van a la unidad de control, 25-21 van hacia la entrada del banco de registros llamada Read register 1 , los bits del 20-16 van hacia el read register 2 y por último los bits del 15-0 van hacia el sign extend.

En el banco de registros salen por el read data 1 con 32 bits directo hacia el otro buffer y de ahí salen junto con la otra salida del banco de registro read data 2 , el read data 1 va directamente a una entrada de la ALU y la otra salida va directamente al multiplexor donde la señal que manda la unidad de control es 0 por lo que pasa directamente a la otra entrada de la ALU, los bits del sign extend ya convertidos en 32 bits va directamente hacia el shift left 2 y esa va directamente a la otra entrada del sumador de al inicio, esto sale por la salida llamada “ALU result” va directamente al multiplexor, debido a esto en la ALU se hace la operación y con el zero flag va directamente a una compuerta lógica AND que va conectada también al BRANCH que viene de la unidad de control, por lo que la salida de esta compuerta va directamente al multiplexor del que manda una señal en 1 y hace que el resultado de “ALU result” vaya directamente hasta el pc de nuevo.

**Tabla del Opcode de las instrucciones Tipo I**

|  |  |
| --- | --- |
| **Operacion** | **Opcode** |
| Addi | 001000 |
| Andi | 001100 |
| Ori | 001101 |
| Stli | 001010 |
| Sw | 101011 |
| Lw | 100011 |
| Beq | 000100 |

**Tabla de la Unidad de Control**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Op** | **Andi** | | **Ori** | **Stli** | **Sw** | **Lw** | **Beq** | **Addi** |
| Aluop | | 100 | 101 | 110 | 000 | 000 | 001 | 000 |
| MemtoReg | | 0 | 0 | 0 | x | 1 | X | 0 |
| RegDst | | 0 | 0 | 0 | X | 0 | X | 0 |
| Regwirte | | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| AluSrc | | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| Er | | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Ew | | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| PcSrc | | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Jump | | 0 | 0 | 0 | x | x | 0 | 0 |

**Tabla de unidad de control**

|  |  |  |  |
| --- | --- | --- | --- |
| **Operacion** |  | **AluC** | **Selector** |
| Addi | Suma | 000 | 0000 |
| Andi | And | 100 | 0100 |
| Ori | Or | 101 | 0101 |
| Stli | Slt | 110 | 1000 |
| Sw | Suma | 000 | 0000 |
| Lw | Suma | 000 | 0000 |
| Beq | Resta | 001 | 0001 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Opcode | Base | Rd | Offset | Operacion |
| 100011 | 00000 | 00000 | 000000000000000 | Lw |
| 101011 | 00000 | 00000 | 000000000000000 | Sw |
| 000100 | 00000 | 00000 | 000000000000000 | Beq |
| 001000 | 00000 | 00000 | 000000000000000 | Addi |
| 001001 | 00000 | 00000 | 000000000000000 | Subi |
| 001100 | 00000 | 00000 | 000000000000000 | Andi |
| 001101 | 00000 | 00000 | 000000000000000 | Ori |
| 001010 | 00000 | 00000 | 000000000000000 | Slti |

**¿Qué módulos hacen posible la magia y como se programaron?**

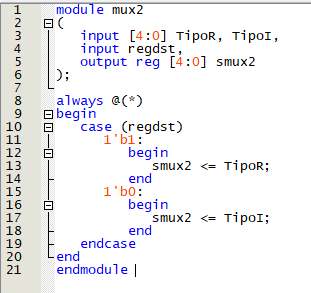


Ilustración 13 Mux 2

Este mux permite tomar el rt como un registro de destino, para guardar el resultado de la operación inmediata.

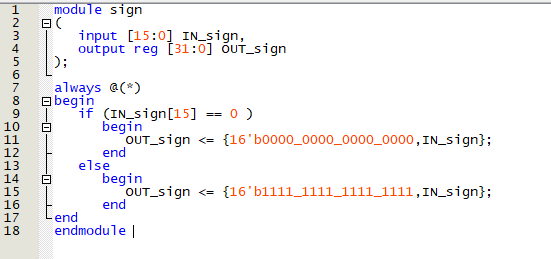


Ilustración 14 sign

Este sign sirve para convertir el valor inmediato en un dato que puede ser operado en alu ya que necesita un dato de 32 bits y este dato inmediato es de 16 bits por lo que este módulo añade los 16 bits ya sea 0 o uno si es negativo.

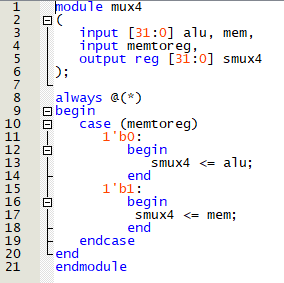


Ilustración 15 Mux 5

Este mux sirve para llevar un dato de la memoria de datos al banco de registros.

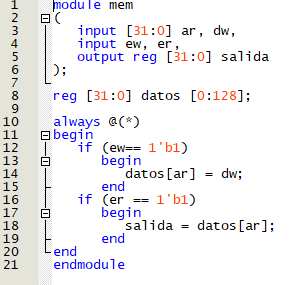


Ilustración 16 Memoria de datos

Este módulo es una memoria las operaciones que interactúan directamente con ella son sw y lw la primera le envía datos desde el banco de registros y la segunda instrucción hace que la memria de datos se los pase al banco.

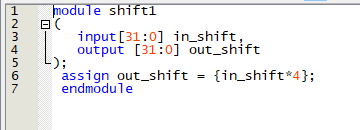


Ilustración 17 Shift left 2

Este módulo sirve para la instrucción beq , debido a que esta instrucción tiene el fin de brincar a otra y esta instrucción le dice al ciclo fetch a cual brincar, por ejemplo si tú quieres brincar a la primera instrucción, y está realmente se encuentra en la posición 4 tú le indicas instrucción 1 por lo que este módulo recibe un lo multiplica por 4 para que tome la instrucción 1 que se encentra en la posición 4 de memoria recordemos que las instrucciones van de 0, 4, 8, 12, 16…

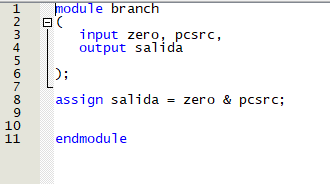


Ilustración 18 Branch

De nuevo este módulo solo sirve para la instrucción beq, lo que hace es recibir una señal habilitadora de la uc y el zero flag de la alu, debido a que la beq le pide a la alu que reste números para ver si son iguales y en caso de ser iguales, al restarse da cero, pue el zero lag de la alu arrojaría uno y este módulo que hace es recibirlo y opéralos con una and el resultado será 1 bit y ese bit será la señal habilitadora del mux que va conectado al pc.

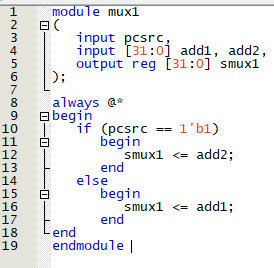


Ilustración 19 Mux 1

Este mux mediante una señal habilitadora manda o el flujo normal del ciclo fetch o el salto que indique la instrucción beq.



Ilustración 20 Fase

Fase 3

La fase 3 es en apariencia la más sencilla y su fin es que se puedan ejecutar las instrucciones de salto, conocidas como tipo j, esta fase implica modificaciones en la unidad de control añadiéndole la salida “Jump” que habilitara si pasa el salto, también en los buffers 2 y 3, esta instrucción junto con la beq solo pasa por 3 ciclos ósea ocupa 4 ciclos para ejecutarse por completo, los otros módulos que se le añaden son otro shift left mas y un multiplexor más que indicara mediante la señal jump antes mencionada si pasa el salto, la instrucción consta solo de dos partes su opcode que indica que la instrucción es j y la dirección a saltar.

**¿Cómo funciona?**

Iniciamos con el pc donde los bits pasan al sumador y a la memoria de instrucciones, de la memoria de instrucciones salen 32 bits los cuales se dividen en diferentes partes, del 25-0 van directamente al shift left 2, lo que hace el shift es que recorre dos espacios los bits es decir si tienes 00010 haces el shift left y te quedaría 01000 se hace esto multiplicando por 4, y los otros bits del 26-31 se van directamente a la unidad de control. En el sumador se le suma 4 y cuando sale por la salida del sumador va directamente a encontrarse con los bits de salida del shift left que salen 28 bits, por lo que se le suman los +4 del sumador y se completa los 31 bits, los bits van directamente al multiplexor, en la unidad de control se activa la señal JUMP que pone al multiplexor en la en 1 por lo que los bits salen directamente al PC de nuevo.

**Tabla instrucción tipo j**

|  |  |
| --- | --- |
| **Operacion** | **Opcode** |
| Jump | 000010 |

**Tabla Unidad de control tipo j**

|  |  |  |
| --- | --- | --- |
| **Op** | **J** | |
| Aluop | | 3’dx |
| MemtoReg | | X |
| RegDst | | X |
| Regwirte | | X |
| AluSrc | | X |
| Er | | X |
| Ew | | X |
| PcSrc | | 0 |
| Jump | | 1 |

**¿Cómo le hicimos y funciono?**

00100000

00001000

00000101

00111001

00100000

00001001

00000000

11001000

00100000

10001010

00000000

00000100

00100000

01001011

00000000

00001010

00001000

00000000

00000000

00000010

00001000

00000000

00000000

00000000

Las primeras 4 instrucciones son sumas inmediatas las últimas dos saltos una salta a la instrucción cero y la otra a la tres ósea a la 8.

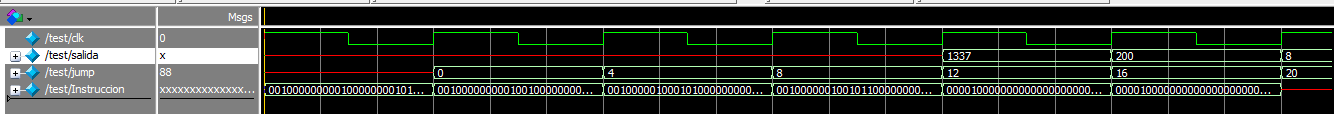


Ilustración 21 Prueba Fase 3

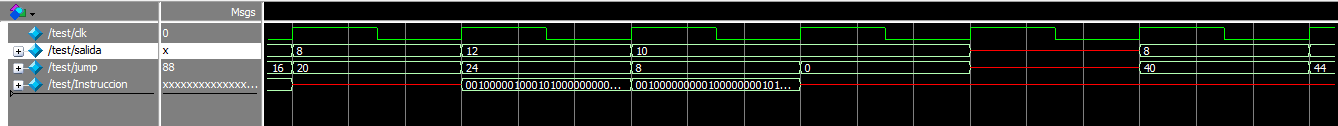


Ilustración 22 Prueba Fase 3

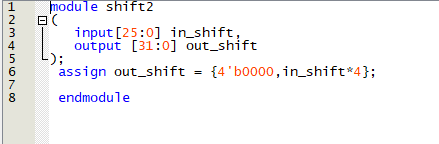


Ilustración 23 Shift left 2 (2)

Este módulo hace exactamente lo mismo que el otro debido a que también sirve para moverte entre instrucciones y debido que recibe solo 25 con el shift se le aumentan dos cero se le agregan otros 4 para que sean 32 que son los que recibe el pc.

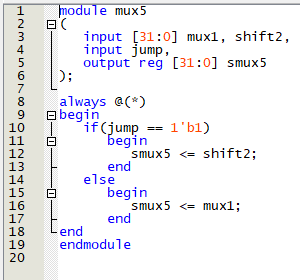
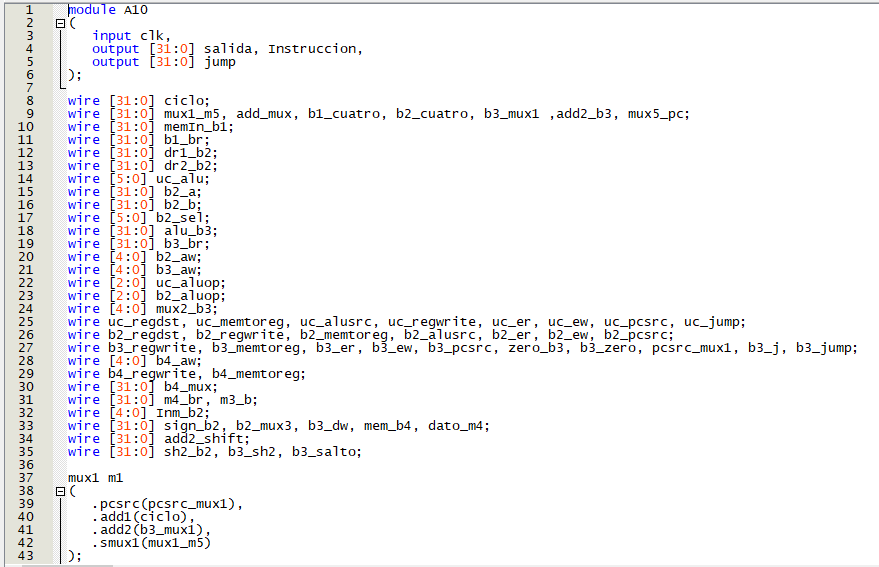
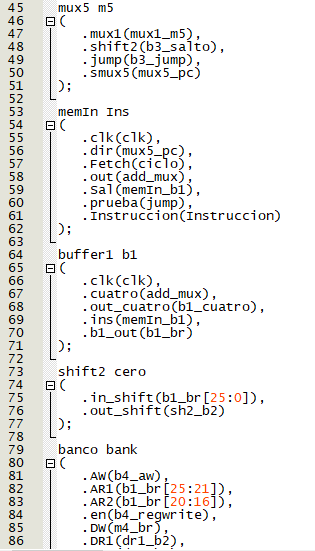


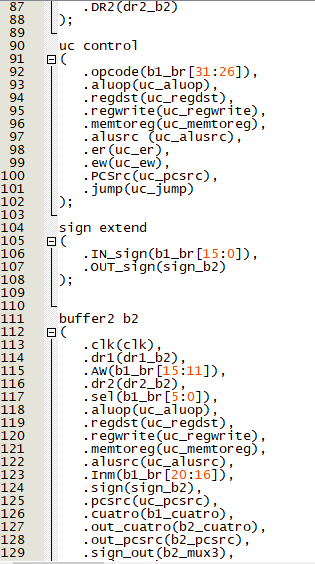
Ilustración 24 mux 5

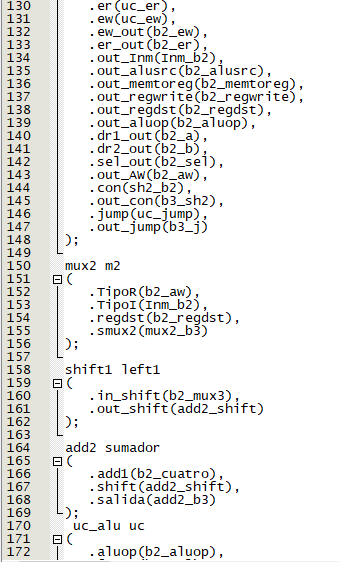
Este mux controlado por la señal jump permitirá pasar ya sea cualquier instrucción I o R o la instrucción j.

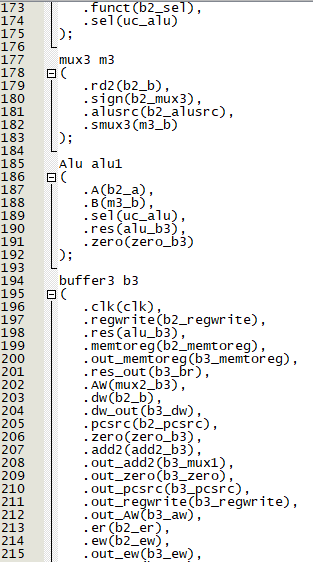
Fase Final

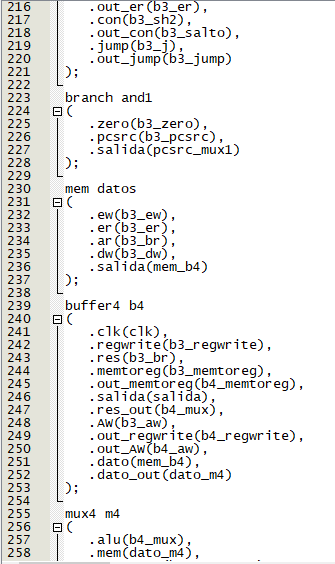


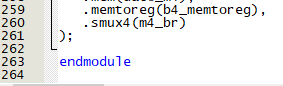












**Test**

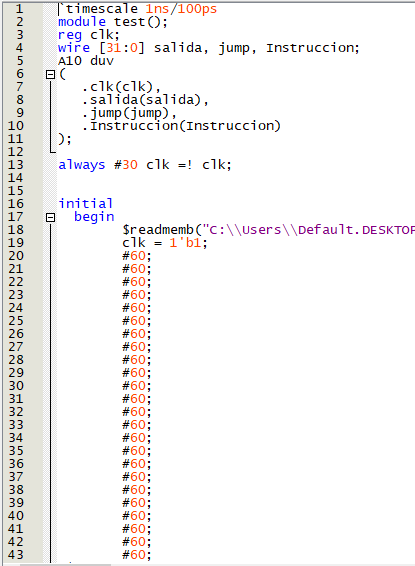


Ilustración Test

**Datpath Mips**

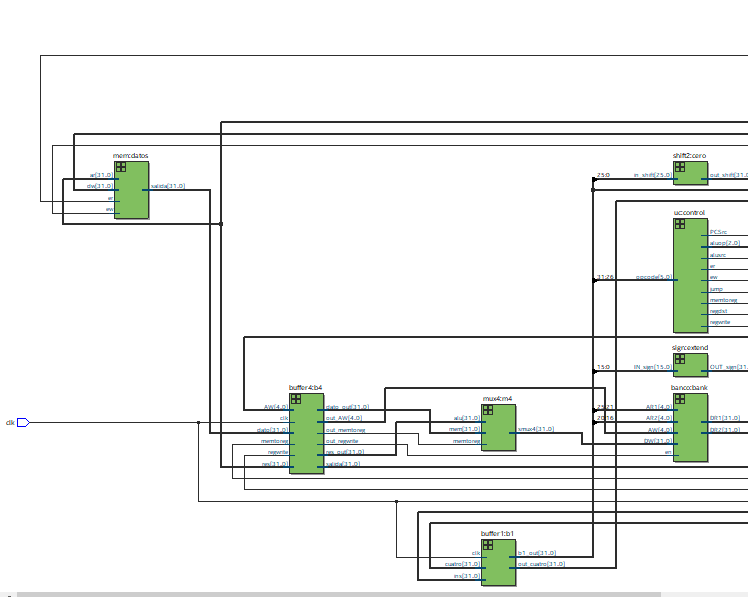


Ilustración 26 Mips parte 1

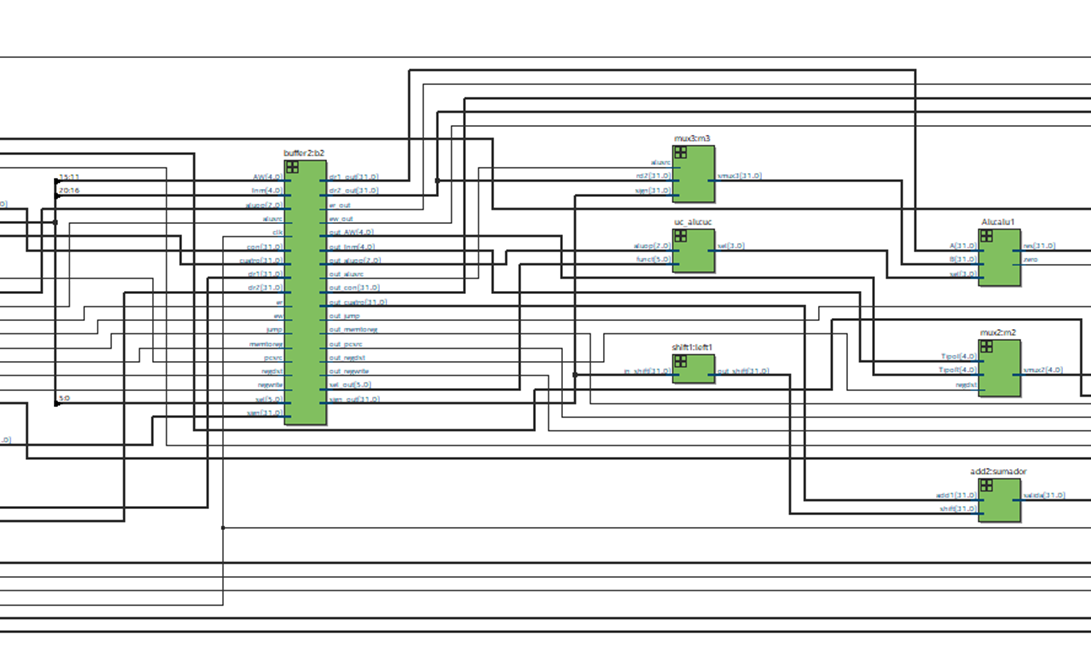


Ilustración 27 Mips parte 2

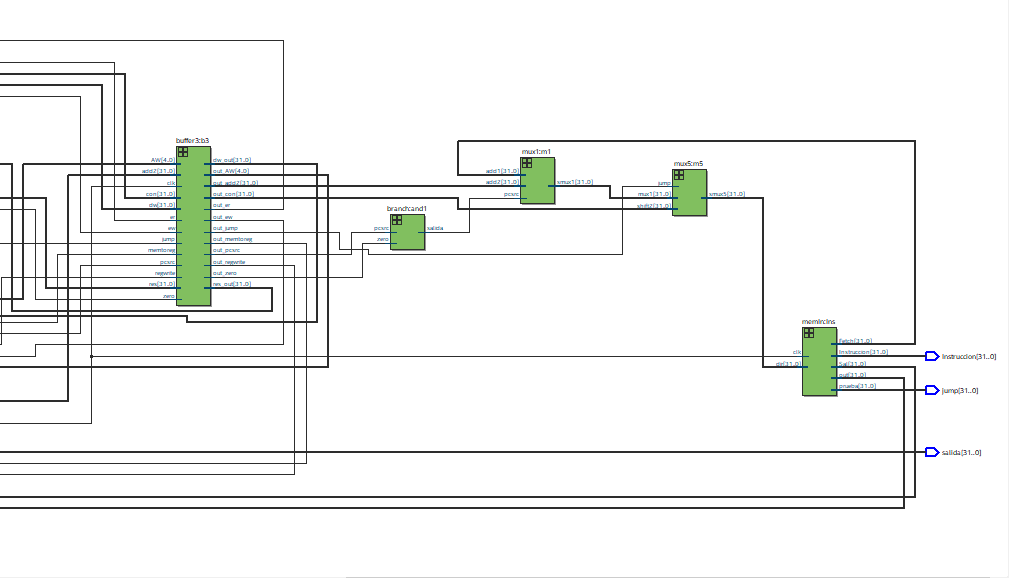


Ilustración Mips parte 3

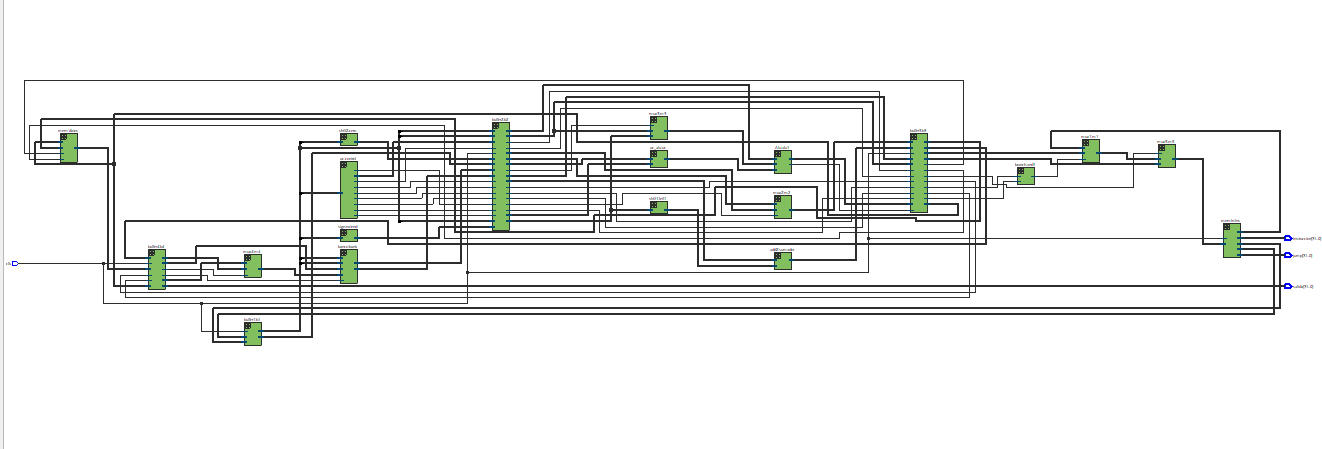
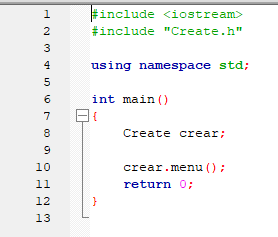


Ilustración 29 The mips the complete edition

Decodificador

Para crear las instrucciones realizamos un programa que es capaz de crear instrucciones tipo R, I y J este está hecho en lenguaje C++ en base al paradigma de programación orientado a objetos y consta de un objeto llamado instrucción que sus atributos son las partes de las instrucciones: op. Rt, rs, rd, shmt, func, inmediato y dirección. Sus métodos realizan la transformación de esos datos a binario, hay otra clase llamada creador que es la encargada de unir la instrucción y cargarla en el archivo de texto, al ejecutarse te pide el tipo de instrucción, su opcode, y dependiendo la instrucción elegida pues sus respectivos campos.



*Ilustración 30 main.cpp*

**Instrucción.h**

1 #ifndef INSTRUCCION\_H

2 #define INSTRUCCION\_H

3 #include <iostream>

4 #include <conio.h>

5 #include <string.h>

6 #include <stdlib.h>

7 #include <stdio.h>

8

9 **using namespace std**;

10

11 **class** Instruccion

12 {

13 **public**:

14 Instruccion();

15

16 **string** getOpcode()**const**;

17 **void** setOpcode();

18

19 **string** getRd()**const**;

20 **void** setRd(**int** valor);

21

22 **string** getRs()**const**;

23 **void** setRs(**int** valor);

24

25 **string** getRt()**const**;

26 **void** setRt(**int** valor);

27

28 **string** getShamt()**const**;

29 **void** setShamt();

30

31 **string** getFunct()**const**;

32 **void** setFunct();

33

34 **string** getInmediato()**const**;

35 **void** setInmediaro(**int** valor);

36

37 **string** getDir()**const**;

38 **void** setDir(**int** valor);

39

40 **friend** ostream& **operator** << (ostream &os, **const** Instruccion& obj);

41

42

43 **private**:

44 **string** Opcode, Rd, Rs, Rt, Shamt, Funct, Dir, Inm;

45 **enum** operaciones

46 {

47 suma,

48 resta,

49 multiplicacion,

50 division,

51 And,

52 Or,

53 Xor,

54 Not,

55 comparacion

56 };

57

58 **enum** OpcI

59 {

60 addi,

61 andi,

62 ori,

63 stli,

64 sw,

65 lw,

66 beq,

67 TypeR,

68 TypeJ

69 };

70

71 };

72

73 #endif // INSTRUCCION\_H

**Instruccion.cpp**

1 #include "Instruccion.h"

2

3 Instruccion::Instruccion()

4 {

5

6 }

7

8 **void** Instruccion::setOpcode()

9 {

10 **int** opc;

11

12 **do**

13 {

14 **cout**<<"Elije el Opcode"<<**endl**<<**endl**

15 <<addi<<".-Suma Inmediata"<<**endl**

16 <<andi<<".- And Inmediata"<<**endl**

17 <<ori<<".- Or inmediato"<<**endl**

18 <<stli<<".- Comparacion Inmediata"<<**endl**

19 <<sw<<".-store word"<<**endl**

20 <<lw<<".-load word"<<**endl**

21 <<beq<<".-beq"<<**endl**

22 <<TypeR<<".-Tipo R"<<**endl**

23 <<TypeJ<<".-Tipo J"<<**endl**

24 <<"Elija: ";

25 **cin** >> opc;

26 **cin**.ignore();

27 system("cls");

28 **switch**(opc)

29 {

30 **case** addi:

31 Opcode = "001000";

32 **break**;

33 **case** andi:

34 Opcode = "001100";

35 **break**;

36 **case** ori:

37 Opcode = "001101";

38 **break**;

39 **case** stli:

40 Opcode = "001010";

41 **break**;

42 **case** sw:

43 Opcode = "101011";

44 **break**;

45 **case** lw:

46 Opcode = "100011";

47 **break**;

48 **case** beq:

49 Opcode = "000100";

50 **break**;

51 **case** TypeR:

52 Opcode = "000000";

53 **break**;

54 **case** TypeJ:

55 Opcode = "000010";

56 **default**:

57 **cout**<<"Te equivocaste"<<**endl**;

58 **break**;

59 }

60 }**while**(opc > 8);

61

62 }

63 **string** Instruccion::getOpcode()**const**

64 {

65 **return** Opcode;

66 }

67

68 **string** Instruccion::getRd()**const**

69 {

70 **return** Rd;

71 }

72 **void** Instruccion::setRd(**int** valor)

73 {

74 **int** residuo, division;

75 **string** num;

76

77

78 **for**(**int** i(0); i < 5; ++i)

79 {

80 **if**(valor == 1)

81 {

82 num += valor + '0' ;

83 valor = 0;

84 i += 1;

85 }

86 residuo = valor % 2;

87 num += residuo + '0';

88 division = valor/2;

89 valor = division;

90 }

91

92 **string** cadena;

93 **for** (**int** i = 4; i>=0;i--)

94 {

95 cadena += num[i];

96 }

97

98 Rd = cadena;

99 }

100

101 **string** Instruccion::getRs()**const**

102 {

103 **return** Rs;

104 }

105 **void** Instruccion::setRs(**int** valor)

106 {

107 **int** residuo, division;

108 **string** num;

109

110 **for**(**int** i(0); i < 5; ++i)

111 {

112 **if**(valor == 1)

113 {

114 num += valor + '0';

115 valor = 0;

116 i += 1;

117 }

118 residuo = valor % 2;

119 num += residuo + '0' ;

120 division = valor/2;

121 valor = division;

122 }

123

124 **string** cadena;

125 **for** (**int** i = 4; i>=0;i--)

126 {

127 cadena += num[i];

128 }

129

130 Rs = cadena;

131 }

132

133 **string** Instruccion::getRt()**const**

134 {

135 **return** Rt;

136 }

137 **void** Instruccion::setRt(**int** valor)

138 {

139 **int** residuo, division;

140 **string** num;

141

142 **for**(**int** i(0); i < 5; ++i)

143 {

144 **if**(valor == 1)

145 {

146 num += valor + '0';

147 valor = 0;

148 i += 1;

149 }

150 residuo = valor % 2;

151 num += residuo + '0';

152 division = valor/2;

153 valor = division;

154 }

155 **string** cadena;

156 **for** (**int** i = 4; i>=0;i--)

157 {

158 cadena += num[i];

159 }

160

161 Rt = cadena;

162 }

163

164 **string** Instruccion::getShamt()**const**

165 {

166 **return** Shamt;

167 }

168 **void** Instruccion::setShamt()

169 {

170 Shamt = "00000";

171 }

172

173 **string** Instruccion::getFunct()**const**

174 {

175 **return** Funct;

176 }

177 **void** Instruccion::setFunct()

178 {

179 **int** opc;

180

181 **do**

182 {

183 **cout**<<"Elije la operacion aritmetica"<<**endl**<<**endl**

184 <<suma<<".-Suma"<<**endl**

185 <<resta<<".-Resta"<<**endl**

186 <<multiplicacion<<".-Multiplicacion"<<**endl**

187 <<division<<".-divison"<<**endl**

188 <<And<<".-and"<<**endl**

189 <<Or<<".-or"<<**endl**

190 <<Xor<<".-xor"<<**endl**

191 <<Not<<".-not"<<**endl**

192 <<comparacion<<".-Comparacion"<<**endl**

193 <<"Elija: ";

194 **cin** >> opc;

195 **cin**.ignore();

196 system("cls");

197 **switch**(opc)

198 {

199 **case** suma:

200 Funct = "100000";

201 **break**;

202 **case** resta:

203 Funct = "100001";

204 **break**;

205 **case** multiplicacion:

206 Funct = "100010";

207 **break**;

208 **case** division:

209 Funct = "100011";

210 **break**;

211 **case** And:

212 Funct = "100100";

213 **break**;

214 **case** Or:

215 Funct = "100101";

216 **break**;

217 **case** Xor:

218 Funct = "100110";

219 **break**;

220 **case** Not:

221 Funct = "000000";

222 **break**;

223 **case** comparacion:

224 Funct = "101010";

225 **break**;

226 **default**:

227 **cout**<<"Te equivocaste"<<**endl**;

228 **break**;

229 }

230 }**while**(opc > 8);

231

232 }

233

234 **string** Instruccion::getInmediato()**const**

235 {

236 **return** Inm;

237 }

238 **void** Instruccion::setInmediaro(**int** valor)

239 {

240 **int** residuo, division;

241 **string** num;

242

243 **for**(**int** i(0); i < 16; ++i)

244 {

245 **if**(valor == 1)

246 {

247 num += valor + '0';

248 valor = 0;

249 i += 1;

250 }

251 residuo = valor % 2;

252 num += residuo + '0' ;

253 division = valor/2;

254 valor = division;

255 }

256 **string** cadena;

257 **for** (**int** i = 15; i>=0;i--)

258 {

259 cadena += num[i];

260 }

261

262 Inm = cadena;

263 }

264

265 **string** Instruccion::getDir()**const**

266 {

267 **return** Dir;

268 }

269 **void** Instruccion::setDir(**int** valor)

270 {

271 **int** residuo, division;

272 **string** num;

273

274 **for**(**int** i(0); i < 26; ++i)

275 {

276 **if**(valor == 1)

277 {

278 num += valor + '0';

279 valor = 0;

280 i += 1;

281 }

282 residuo = valor % 2;

283 num += residuo + '0' ;

284 division = valor/2;

285 valor = division;

286 }

287 **string** cadena;

288 **for** (**int** i = 25; i>=0;i--)

289 {

290 cadena += num[i];

291 }

292

293 Dir = cadena;

294 }

295

296 ostream& **operator** << (ostream& os, **const** Instruccion& obj)

297 {

298 os<<obj.getOpcode()<<"\_"<<obj.getRs()<<"\_"<<obj.getRt()<<"\_"<<obj.getRd()<<"\_"<<obj.getShamt()<<"\_"<<obj.getFunct()<<obj.getInmediato()<<obj.getDir()<<**endl**;

299 **return** os;

300 }

301

**Create.h**

1 #ifndef CREATE\_H

2 #define CREATE\_H

3 #include <iostream>

4 #include "Instruccion.h"

5 #include <stdio.h>

6 #include <stdlib.h>

7 #include <fstream>

8

9 **using namespace std**;

10

11 **class** Create

12 {

13 **public**:

14 Create();

15

16 **void** menu();

17 **void** tipoR();

18 **void** tipoI();

19 **void** tipoJ();

20

21 **private**:

22 **enum** opciones

23 {

24 op\_salir,

25 op\_tipoR,

26 op\_tipoI,

27 op\_tipoJ

28 };

29

30 };

31

32 #endif

33

**Create.cpp**

1 #include "Create.h"

2

3 Create::Create()

4 {

5 //ctor

6 }

7

8 **void** Create::menu()

9 {

10 **int** opc;

11 **do**

12 {

13 **cout**<<"Set instrucciones"<<**endl**<<**endl**

14 <<"Â¿Que desea hacer?"<<**endl**

15 <<op\_tipoR<<"( Tipo R"<<**endl**

16 <<op\_tipoI<<"( Tipo I"<<**endl**

17 <<op\_tipoJ<<"( Tipo J"<<**endl**

18 <<op\_salir<<"( Salir"<<**endl**

19 <<"Seleccione: ";

20 **cin**>>opc;

21 **cin**.ignore();

22 system("CLS");

23 **switch**(opc)

24 {

25 **case** op\_tipoR:

26 tipoR();

27 **break**;

28 **case** op\_tipoI:

29 tipoI();

30 **break**;

31 **case** op\_tipoJ:

32 tipoJ();

33 **break**;

34 **case** op\_salir:

35 **cout**<<"Adios..."<<**endl**;

36 **break**;

37 **default**:

38 **cout**<<"Te equivocaste"<<**endl**;

39 system("PAUSE");

40 system("CLS");

41 break;

42 }

43 }while(opc != op\_salir);

44 }

45

46 void Create::tipoR()

47 {

48 ofstream salida("archivo.txt", ios::app);

49

50 Instruccion ins;

51 int Op, Rd, Rs, Rt, Sh;

52 cout<<"Instruccion Tipo R "<<endl<<endl;

53

54

55 ins.setOpcode();

56

57 cout<<"Registro fuente: ";

58 cin>>Rs;

59 ins.setRs(Rs);

60

61 cout<<"Registro Temporal: ";

62 cin>>Rt;

63 ins.setRt(Rt);

64

65 cout<<"Registro destino: ";

66 cin>>Rd;

67 ins.setRd(Rd);

68

69 ins.setShamt();

70

71 ins.setFunct();

72

73

74 if(!salida.is\_open())

75 {

76 cout<<"Error al cargar el archivo"<<endl;

77 }

78 else

79 {

80 salida <<ins;

81 salida.close();

82 }

83 }

84

85 void Create::tipoI()

86 {

87 ofstream salida("archivo.txt", ios::app);

88

89 Instruccion ins;

90 int Rd, Rs, Inm;

91 cout<<"Instruccion Tipo I "<<endl<<endl;

92

93 cout<<"Ingrese los siguientes datos "<<endl<<endl;

94 ins.setOpcode();

95

96 cout<<"Registro fuente: ";

97 cin>>Rs;

98 ins.setRs(Rs);

99

100 cout<<"Registro destino: ";

101 cin>>Rd;

102 ins.setRd(Rd);

103

104 cout<<"Inmediato: ";

105 cin>>Inm;

106 ins.setInmediaro(Inm);

107

108 if(!salida.is\_open())

109 {

110 cout<<"Error al cargar el archivo"<<endl;

111 }

112 else

113 {

114 salida <<ins;

115 salida.close();

116 }

117

118 }

119

120 void Create::tipoJ()

121 {

122 ofstream salida("archivo.txt", ios::app);

123

124 Instruccion ins;

125 int Op, Dir;

126 cout<<"Instruccion Tipo J "<<endl<<endl;

127

128 ins.setOpcode();

129

130 cout<<"Direccion: ";

131 cin>>Dir;

132 ins.setDir(Dir);

133

134 if(!salida.is\_open())

135 {

136 cout<<"Error al cargar el archivo"<<endl;

137 }

138 else

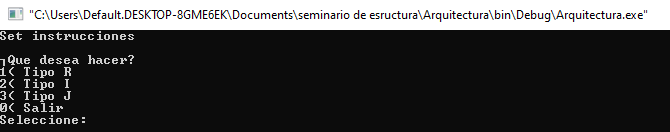
139 {

140 salida <<ins;

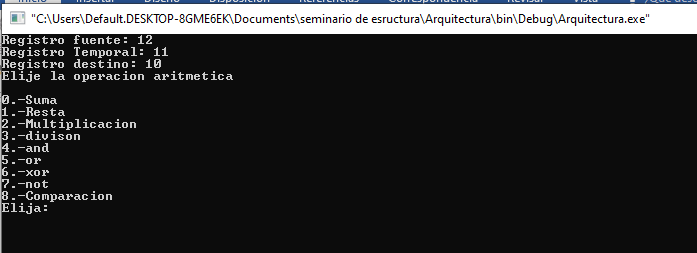
141 salida.close();

142 }

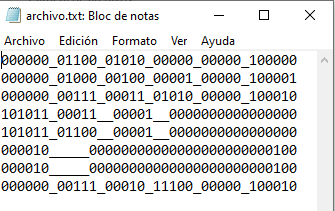
143 }



*Ilustración 31 Menú de selección*



*Ilustración 32 Recolección de datos del programa*



*Ilustración 33 Así los guarda*

Programa Ensamblador

**Números perfectos:**

¿Qué son los números perfectos?

Básicamente un número perfecto es aquél que **es igual a la suma de sus divisores, exceptuando él mismo,** estos divisores que no incluyen al mismo número son los que se conocen como factores o divisores propios.

Un sencillo ejemplo de que es un numero perfecto y no:

El 12 sus divisores que no son el mismo, son 1,2,3,4,6 si estos se suman 1+2+3+4+6 = 16 por lo tanto esto no es un numero perfecto.

Pero el 6 si es porque sus divisores exceptuando el mismo son, 1,2, ,3 si se suman son = 1+2+3 = 6

Para la realización del código ensamblador realizamos este código en c++ para entender cómo funciona el algoritmo

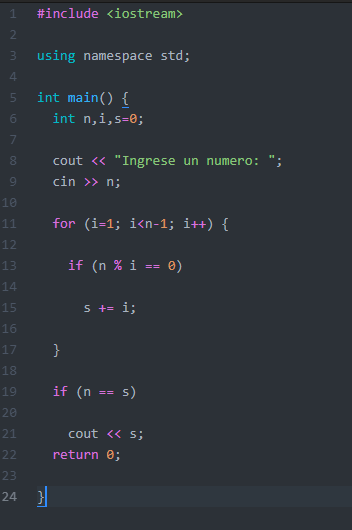


Ilustración 34 C++ números perfectos

**Programa ensamblador**

Esto sería como la función principal:

lui $v0 4

ori $v0 4

lui $v0 5

ori $v0 4

sw $t0 $v0 # cantidad de números a mostrar

lui $t1 0 # contador de números mostrados

ori $t1 0

lui $t7 0 # guardar en el 2 ala n-1 // guarda el numero perfecto

ori $t7 0

lui $t8 0 # guardar en el (2 ala n) – 1

ori $v8 0

# obtiene el numero primo

lui $s0 1 # guardar el primo actual

ori $v0 1

lui $s1 2 # contador, se toma como divisores

ori $s1 2

lui $s2 0 # guardar resto

ori $s2 0

# Potenciación

lui $t2 0 # guardar la base de la potenciación

ori $t2 0

lui $t3 0 # guardar el exponente

ori $t3 0

lui $t4 0 # contador

ori $t4 0

lui $t5 0 # guardar la potenciación

ori $t5 0

lui $t6 1 # cuando el exponente es 1

ori $t6 1

b $0 $0 numeros perfectos\_loop

siguiente es primo:

beq $s0 $s1 regresa el primo

div $s0 $s1

beq $s2 no es primo

addi $s1 $s1 1

b $0 $0 siguiente primo

no es primo:

li $s1 2

ori $s1 2

addi $s0 $s0 1

b $0 $0 siguiente primo

Regresa al primo:

li $s1 2

ori $s1 2

j $ra

loop de números perfectos:

beq $t0 $t1 exit

addi $s0 $s0 1

# 2 ala n-1

sw $t3 $s0

subi $t3 $t3 1

lui $t2 2

sw $t7 $t5

# (2 ala n) - 1

sw $t3 $s0

lui $t2 2

ori $t2 2

sw $t8 $t5

subi $t8 $t8 1

# se multiplicaran estos registros $t8 \* $t7

mulu $t7 $t7 $t8

addi $t1 $t1 1

b $0 $0 numeros perfectos loop

mostrar numero perfecto:

lui $v0 1

ori $v0 1

sw $a0 $t7

lui $v0 4

ori $v0 4

la $a0 espacio

j $ra

potenciación:

beqz $t3 potenciación en cero

beq $t3 $t6 potenciación en uno

lui $t4 1 # cuando el exponente es mayor a 2

ori $v4 1

li $t5 2

ori $t5 2

b $0 $0 potenciación loop

loop de potenciación:

beq $t4 $t3 regresa a la potenciación

addi $t4 $t4 1

mulu $t5 $t5 $t2

b $0 $0 potenciación loop

potenciación uno:

sw $t5 $t2

b $0 $0 potenciacion return

potenciación cero:

lui $t5 1

ori $t5 1

b $0 $0 potenciacion return

regresa a la potenciación:

j $ra

Salir: lui $v0 10

ori $v0 10

**Anotaciones** sobre el programa o algoritmo se supone toma los números perfectos ingresado por un usuario, relamente lo hicimos con lenguaje c y lo traducimos utilizado el programa de godbolt y nos salieron pseudo instrucciones como la li o como mov las cuales no encontramos que clase de recorrido hacían.

Conclusión

Fue un proyecto bastante tardado de hacer debido a la complejidad del datapath que este tiene y maneja no fue tan sencillo desarrollarlo y que funcionara lo que si no pudimos hacer de manera correcta fue crear el set de instrucciones y que lo ejecutara, porque nos encontramos con unas instrucciones bien raras. Y otras psudosinstucciones que nos obligaban a añadir otras cosas al código de verilog que entorpecían nuestro programa.

Referencias

<https://godbolt.org/>

<https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf>

<https://previa.uclm.es/profesorado/licesio/Docencia/ETC/24_CDC-Uniciclo.pdf>

<https://www.youtube.com/channel/UCMhf9Vo2tt_CCJ3qOPLkWNg>

<https://www.youtube.com/watch?v=oETOwVBzu1s>