Proyecto Final

Diego Jared Jimenez Silva

Gael Ramses Alvarado Lomelí

220559845

220529474

4/05/2025

Arquitectura de Computadoras

Introducción

Los procesadores MIPS (Microprocessor without Interlocked Pipeline Stages) son microprocesadores que no tienen interbloqueos de pipeline en sus etapas, lo que significa que no necesitan mecanismos de interbloqueo para manejar instrucciones y evitar conflictos en la ejecución. Tener un mecanismo de interbloqueo puede empeorar el rendimiento y la eficiencia general del procesador. En contra parte, los MIPS optimizan el manejo de instrucciones a través del pipeline, esto consiste en dividir el procesamiento en etapas como por ejemplo fetch, memory y write, permitiendo así que las instrucciones se ejecuten de manera continua y sin interrupciones. Al combinar lo anterior con su tipo de arquitectura RISC (Reduced Instruction Set Computer), hace que los MIPS sean eficientes y simples al mismo tiempo.

Los MIPS tienen distintos tipos de instrucciones, estás pueden ser de tipo R las cuales hacen referencia a operaciones aritméticas, lógicas y de desplazamiento, el tipo I que son para memoria y comparaciones, finalmente el tipo j que son para controlar el flujo del programa.

Para representar las instrucciones utilizamos mnemónicos, estos son palabras clave para recordar y entender las distintas instrucciones, por ejemplo, add representaría una suma.

A continuación, el set de instrucciones de 32 bits que trabajaremos en este proyecto:

Instrucciones de tipo R:

Consisten en instrucciones conformadas por Opcode que indica el tipo de operación (6bits), Rs que indica el registro 1 (5 bits), Rt que indica el registro 2 (5bits), Rd que indica el registro donde se almacenará el dato (5 bits), Shamt que indica la cantidad de desplazamientos (5 bits), Funct que indica la operación exacta dentro del tipo R.

- Add: Suma valores de dos registros y se guarda el resultado.
- Sub: Resta el valor de un registro a otro y guardamos el resultado.
- Mul: Multiplican los valores de dos registros y guarda el resultado.
- Div: Divide el valor de un registro entre otro y guarda el cociente.
- Or: Es la operación lógica OR operando bit por bit dos registros.
- And: Es la operación lógica AND operando bit por bit dos registros.
- Slt (Set on Less Than): Compara dos registros, el primero es menor a el segundo, si se cumple almacena 1, por el contrario, se almacena 0.

- Nop: No realiza ninguna operación.

Instrucciones de tipo I:

Aquí se trabajan valores inmediatos constantes, teniendo así de igual forma Opcode que indica el tipo de operación (6bits), Rs que indica el registro 1 (5 bits), Rt que indica el registro donde se almacenará el dato (5bits), Inmediate que indica el valor constante (16bits).

- Addi: Igual que Add, pero en vez de sumar dos registros, suma un registro y un valor inmediato.
- Subbi: Igual que Subb, pero en vez de restar dos registros, resta el registro con un valor inmediato.
- Ori: Igual que Or, pero en vez de utilizar dos registros, usa un registro y un valor inmediato.
- Andi: Igual que And, pero en vez de utilizar dos registros, usa un registro y un valor inmediato.
- Lw: Carga 32 bits de la memoria.
- Sw: Almacena un registro en memoria.
- Slti: Igual que Slt, pero compara un registro con un valor inmediato, si el registro es menor es 1 de lo contrario es 0.
- Beq: Branch if equal realiza saltos si dos registros son iguales.
- Bne: Branch if not equal realiza saltos si dos registros son diferentes.
- Bgtz: Branch if Greater Than Zero realiza saltos si el valor de un registro es mayor a cero.

Instrucciones de tipo J:

Aquí controlamos saltos en el programa, utilizamos solamente Opcode que indica el tipo de operación (6bits) y Addres que indica la dirección del salto del programa (26bits).

- J: Realiza un salto a una dirección especifica en el programa.

En este proyecto trabajaremos con un datapath completo en la arquitectura MIPS, esto sería el conjunto de componentes que procesan una instrucción. Esta se compondrá por lo siguiente:

- PC o program counter: Se encargará de saltar a la siguiente instrucción con un ciclo fetch.
- Memoria de instrucciones: Almacena instrucciones para la ejecución del programa, estas están en un formato de 32 bits.
- Unidad de control: Controla y dirige el flujo de datos.

- Banco de registros: Almacena valores preestablecidos junto con los resultados obtenidos durante la ejecución.
- ALU: La unidad aritmeticológica se encarga de las operaciones matemáticas (suma) o lógicas (And).
- Memoria de datos: Es utilizada para almacenar y recuperar datos en la ejecución del programa.
- Multiplexores: Estos redirigen los datos por diferentes salidas según ciertas condiciones.
- Shift Left: Hace desplazamientos de bits para calcular direcciones, recorriendo 2 bits, ya que las instrucciones se guardan en múltiplos de 4.
- Sign Ext: Convierte valores de 16 bits a 32 bits rellenando con ceros o unos dependiendo de si el numero es positivo o negativo.
- ALU Control: Este define la operación que realizará la ALU.

Con base a esto haremos un datapath completo en la arquitectura MIPS.

Este proyecto se divide por partes, teniendo un total de 3 fases donde la primera fase corresponde a las instrucciones tipo R y sus módulos necesarios, la segunda fase continuamos con los módulos necesarios para las instrucciones tipo I, y por ultimo los módulos necesarios para las instrucciones tipo J. En base a esto construiremos el datapath completo por medio de etapas.

Finalmente construiremos un código para decodificar instrucciones en ensamblador a código binario para precargar la memoria de instrucciones con dicho archivo.

El código ensamblador a implementar básicamente realiza una serie de operaciones aritméticas y una comparación lógica (SLT) utilizando instrucciones basadas en las instrucciones de los procesadores MIPS.

El propósito de este código ensamblador es realizar las sumas, restas y determinar si un valor es menor que otro utilizando los valores iniciales que se encuentran en el banco de registros, las instrucciones marcan que registros utilizar para hacer las operaciones y donde se guardara el resultado. El código ensamblador a implementar es el siguiente:

Sub $20 $15 $9

NOP

Sub $20 $20 $9

NOP

ADD $15 $5 $15

NOP

ADD $15 $9 $15

NOP

SLT $21 $20 $15.

El banco de registros a utilizar es el siguiente:

00000000000000000000000000000000 // 1 = 0
00000000000000000000000000000000 // 2 = 0
00000000000000000000000000000000 // 3 = 0
00000000000000000000000000000000 // 4 = 0
00000000000000000000000000010100 // 5 = 20
00000000000000000000000000001100 // 6 = 12
00000000000000000000000000110111 // 7 = 55
00000000000000000000000001001000 // 8 = 72
00000000000000000000000001100100 // 9 = 100
00000000000000000000000000000000 // 10 = 0
00000000000000000000000000000000 // 11 = 0
00000000000000000000000000000000 // 12 = 0
00000000000000000000000000000000 // 13 = 0
00000000000000000000000000000000 // 14 = 0
00000000000000000000001111100111 // 15 = 999
00000000000000000000000000000000 // 16 = 0

Las 16 posiciones faltantes son 0.

El funcionamiento de cada instrucción es la siguiente:

Instrucción

SUB $20, $15, $9 R[20] = R[15] - R[9] 999 - 100 = 899

NOP

SUB $20, $20, $9 R[20] = R[20] - R[9] 899 - 100 = 799

NOP

ADD $15, $5, $15 R[15] = R[5] + R[15] 20 + 999 = 1019

NOP

ADD $15, $9, $15 R[15] = R[9] + R[15] 100 + 1019 = 1119

NOP

SLT $21, $20, $15 R[21] = (R[20] < R[15]) ? 1 : 0 799 < 1119 → 1.

Los NOP no realiza ninguna operación, pero sí consume un ciclo de reloj. Su principal función en este contexto es esperar o dar tiempo a que ciertas operaciones se completen dentro del datapath del procesador.

Seudocódigo:

```
R20 = R15 - R9        // R20 = 999 - 100 = 899

R20 = R20 - R9        // R20 = 899 - 100 = 799

R15 = R5 + R15        // R15 = 20 + 999 = 1019

R15 = R9 + R15        // R15 = 100 + 1019 = 1119

if R20 < R15 then

   R21 = 1

else

   R21 = 0
```

Objetivos

Con este trabajo esperamos comprender y desarrollar un programa en ensamblador MIPS, esto por medio de la creación de un datapath completo el cual procesará instrucciones mandando los datos entre registros, unidades y memoria, de esta forma ejecutará correctamente las distintas instrucciones de tipo R, J e I.

Con esto, como objetivo particular, esperamos juntar todo lo aprendido en el ciclo, tener una comprensión clara sobre la creación de todos los módulos y su funcionamiento, comprender globalmente como se procesan los datos y visualizar los resultados esperados para reafirmar el conocimiento. Así también, tenemos el objetivo de documentar correctamente lo aprendido para que quede un conocimiento solido para usar como fuente de consulta más adelante, explicando detalladamente todo el proyecto.

Desarrollo

Explicaremos los módulos paso a paso en este apartado, viendo el por qué de su implementación, su lógica detrás y finalmente comprobando su funcionamiento tal como lo esperado. Priorizaremos describir en orden los módulos según pasen los datos, excluyendo los módulos de las fases siguientes hasta su creación.

Modulo PC:

El módulo PC está conformado por 2 entradas y 1 salida. Las entradas son las siguientes: IN (32 bits) y CLK (1 bit). La salida es la siguiente: OUT (32 bits).

El modulo PC como tal es sencillo, guarda un valor de 32 bits y por medio de un always @(posedge CLK) al detectar un cambio en el reloj, entonces asigna el valor de salida igual al valor de entrada.

Dentro del programa este representa un contador que almacena la dirección de la siguiente instrucción a ejecutar empezando por la posición 0 y por ello se inicializa la salida como tal (32'd0). Con ello actualizamos la dirección almacenada, y generamos una ejecución continua y secuencial de las instrucciones en el sistema.

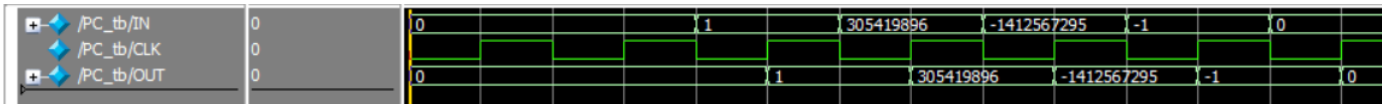
A continuación, se adjunta una imagen del código en Verilog.

```
module PC(
    input [31:0] IN,
    input CLK,
    output reg [31:0] OUT = 32'd0   // Inicializado a 0
);

always @(posedge CLK) begin
    OUT <= IN;
end

endmodule
```

El testbench para probar que funcione correctamente este módulo es:

```
`timescale 1ns / 1ps
module PC_tb;
    // Entradas
    reg [31:0] IN;
    reg CLK;

    // Salida
    wire [31:0] OUT;
    // Instanciar el Módulo Bajo Prueba (UUT)
    PC uut (
        .IN(IN),
        .CLK(CLK),
        .OUT(OUT)
    );

    // Generación del reloj: periodo de 10ns
    initial begin
        CLK = 0;
        forever #5 CLK = ~CLK;
    end
    // Estímulos
    initial begin
        // Inicializar entrada
        IN = 32'b00000000000000000000000000000000;

        // Esperar algunos ciclos de reloj
        #20;

        // Aplicar vectores de prueba
        IN = 32'b00000000000000000000000000000001;
        #10; // esperar un flanco de reloj

        IN = 32'b00010010001101000101011001111000;
        #10;

        IN = 32'b10101011110011011110111100000001;
        #10;

        IN = 32'b11111111111111111111111111111111;
        #10;

        IN = 32'b00000000000000000000000000000000;
        #10;

        // Finalizar simulación
        #20;
        $finish;
    end
endmodule
```

Su objetivo es comprobar que el valor de entrada (IN) se copia correctamente a la salida (OUT) en el flanco positivo del reloj (CLK). Para esto primero se le da diferentes valores a IN. Después de cada cambio, espera 10 ns, para que pase un pulso de reloj. Cada vez que el reloj sube (de 0 a 1), OUT debe tomar el mismo valor que tiene IN en ese momento. Adjunto las pruebas del testbench:



Sumador:

El módulo ADD4 (sumador de 4 bits) está conformado por 1 entrada y 1 salida. La entrada es la siguiente: A (32 bits). La salida es la siguiente: RES (32 bits).

En módulo ADD4 implementamos una suma constante de 4 más la entrada en A, así el resultado se incrementará 4 cada vez.

En la funcionalidad del módulo, utilizamos un bloque always donde al detectar un cambio en la entrada, se realiza la suma, agregando 4 al valor de A y guardándose en RES. La lógica consiste en asegurar que los valores generados sean múltiplos de 4 para su correcto almacenamiento en memoria.

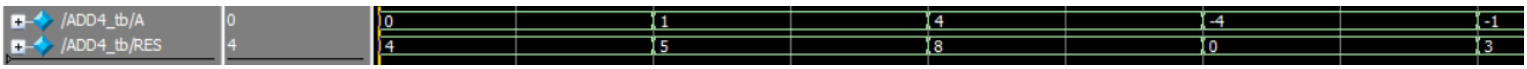
A continuación, se adjunta una imagen del código en Verilog.

```
module ADD4(
    input [31:0] A,
    output reg [31:0] RES
);

// En esta ocasión solo sumaremos 4 cada vez, esto para guardar en múltiplos de 4 en la memoria
always @(*) begin
    RES = A + 32'd4;
end
endmodule
```

El testbench para probar que funcione correctamente este módulo es:

```
`timescale 1ns / 1ps

module ADD4_tb;
    // Entradas
    reg [31:0] A;

    // Salidas
    wire [31:0] RES;

    // Instanciar el módulo bajo prueba (UUT)
    ADD4 uut (
        .A(A),
        .RES(RES)
    );

    // Estímulos
    initial begin

        // Inicializar
        A = 32'b00000000000000000000000000000000;
        #10;

        A = 32'b00000000000000000000000000000001;
        #10;

        A = 32'b00000000000000000000000000000100;
        #10;

        A = 32'b11111111111111111111111111111100;
        #10;

        A = 32'b11111111111111111111111111111111;
        #10;

        // Finalizar simulación
        #20;
        $finish;
    end

endmodule
```

Su objetivo es simplemente sumar 4 al valor de entrada A y muestra el resultado en RES. Inicializamos la entrada A con valores al azar y solo se le sumara 4. Adjunto las pruebas del testbench:



Memoria de instrucciones:

El módulo de memoria de instrucciones (MEMI) está conformado por 1 entrada y 1 salida. La entrada es la siguiente: DR (32 bits). La salida es la siguiente: INS (32 bits).

En la memoria de instrucciones se declara una memoria de 1000 posiciones donde cada una almacena 8 bits. Junto con ello, utilizamos un initial begin para leer el archivo y cargar los datos en mem.

En la funcionalidad de la memoria de instrucciones, utilizamos un bloque always para que, al detectar un cambio de entrada, se asigne el valor de INS tomando cuatro posiciones continuas de la memoria, desde la dirección indicada por DR hasta la siguiente cuarta posición. La lógica consiste en tomar una instrucción de 32 bits guardada en la memoria para su uso dentro del sistema.

A continuación, se adjunta una imagen del código en Verilog.

```
module MEMI(
    input [31:0] DR,        // Dirección
    output reg [31:0] INS   // Instruccion de salida
);

reg [7:0] mem[0:999];    // 1000 posiciones que guardan datos de 1 byte

// Leemos el archivo
initial begin
    #50
    $readmemb("datos", mem);
end

// Asignamos
always @(*) begin
    // Recorremos las posiciones siguientes sumando la direccion dada
    INS[31:24] <= mem[DR];
    INS[23:16] <= mem[DR + 1];
    INS[15:8]  <= mem[DR + 2];
    INS[7:0]   <= mem[DR + 3];
end

endmodule
```

El testbench para probar que funcione correctamente este módulo es:

```
`timescale 1ns / 1ps

module MEMI_tb;
    // Entrada
    reg [31:0] DR;      // Dirección de lectura

    // Salida
    wire [31:0] INS;    // Instrucción leída

    // Instanciar el módulo bajo prueba
    MEMI uut (
        .DR(DR),
        .INS(INS)
    );

    // Estímulos
    initial begin
        // Esperar que se cargue la memoria desde el archivo
        #5;

        // Probar diferentes direcciones (múltiplos de 4)
        DR = 32'd0;
        #10;

        DR = 32'd4;
        #10;

        DR = 32'd8;
        #10;

        DR = 32'd12;
        #10;

        DR = 32'd100;
        #10;

        // Finalizar simulación
        #10;
        $finish;
    end

endmodule
```

Primero el testbench espera un poco (5ns) para dar tiempo a que se cargue la memoria. Luego, prueba varias direcciones: 0, 4, 8, 12 y 100 (todas son múltiplos de 4). Después de cada dirección, espera 10ns para ver qué valor se lee en INS.

Adjunto las pruebas del testbench:

Control:

La Unidad de Control está conformada por 1 entrada y 8 salidas. La entrada es la siguiente: OpCode (6 bits). Las salidas son las siguientes: RegDst (1 bit), Branch (1 bit), MemRead (1 bit), MemToReg (1 bit), ALUOp (3 bits), MemToWrite (1 bit), ALUSrc (1 bit) y RegWrite (1 bit).

La Unidad de Control se encarga del flujo de datos, con ello manda las distintas señales según el OpCode de la instrucción a otros módulos del sistema.

En la funcionalidad del módulo, utilizamos un always que sea sensible a la entrada de OpCode y así las señales de control se asignen de forma adecuada según la instrucción. Primeramente, definimos todas las salidas como 0 por defecto y así modificar su salida según el caso específico.

Primeramente, en la fase uno solo tenemos el tipo de instrucción tipo R donde OpCode es 6'b000000, aquí se activan RegWrite y RegDst, y se asigna ALUOp para determinar que haremos caso al function en esta ocasión.

La lógica consiste en decodificar las instrucciones y activar las señales ciertas salidas correspondientes a el tipo de instrucción.

A continuación, se adjunta una imagen del código en Verilog.

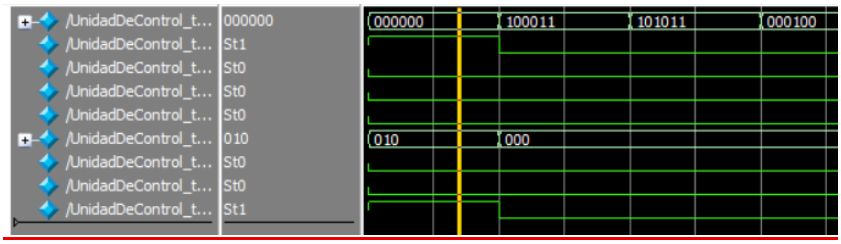
```
// UnidadDeControl
module UnidadDeControl (
    input  wire [5:0]  OpCode,
    output reg         RegDst,
    output reg         Branch,
    output reg         MemRead,
    output reg         MemToReg,
    output reg [2:0]   ALUOp,
    output reg         MemToWrite,
    output reg         ALUSrc,
    output reg         RegWrite

);
    always @(*) begin
        // Valores por defecto
        MemToReg  = 1'b0;
        MemToWrite  = 1'b0;
        ALUOp     = 3'b000;
        RegWrite  = 1'b0;
        RegDst = 1'b0;
        Branch = 1'b0;
        MemRead = 1'b0;
        ALUSrc = 1'b0;

        case (OpCode)
            6'b000000: begin // Instuccion R
                RegWrite = 1'b1;
                ALUOp    = 3'b010;
                RegDst = 1'b1;

            end
            default: begin
                // Aquí se pueden agregar otros opcodes
            end
        endcase
    end
endmodule
```

El testbench para probar que funcione correctamente este módulo es:

```
`timescale 1ns / 1ps
module UnidadDeControl_tb;
    // Señales
    reg [5:0] OpCode;

    wire RegDst;
    wire Branch;
    wire MemRead;
    wire MemToReg;
    wire [2:0] ALUOp;
    wire MemToWrite;
    wire ALUSrc;
    wire RegWrite;

    // Instanciar módulo bajo prueba
    UnidadDeControl uut (
        .OpCode(OpCode),
        .RegDst(RegDst),
        .Branch(Branch),
        .MemRead(MemRead),
        .MemToReg(MemToReg),
        .ALUOp(ALUOp),
        .MemToWrite(MemToWrite),
        .ALUSrc(ALUSrc),
        .RegWrite(RegWrite)
    );
    // Estímulos
    initial begin

        // Prueba para instrucción tipo R (OpCode = 000000)
        OpCode = 6'b000000;
        #10;

        // Prueba para opcode no definido (por defecto)
        OpCode = 6'b100011; // LW, por defecto outputs en cero
        #10;

        OpCode = 6'b101011; // SW, por defecto
        #10;

        OpCode = 6'b000100; // BEQ, por defecto Branch=0
        #10;

        // Finalizar simulación
        #10;
        $finish;
    end

endmodule
```

Primero el testbench recibe un OpCode de 6 bits (parte de una instrucción MIPS). Según el OpCode, activa o desactiva señales como: RegWrite, ALUSrc, Branch, etc. Actualmente solo reconoce instrucciones tipo R (OpCode = 000000). Los demás casos usan valores por defecto (la mayoría en 0). El módulo reacciona correctamente cuando el OpCode es válido (tipo R). Usa valores por defecto cuando el OpCode no está implementado.

Adjunto las pruebas del testbench:



Banco de registros:

El banco de registros está conformado por 6 entradas y 2 salidas. Las entras son las siguientes: Reloj (1bit), RegEn (1bit), ReadReg1 (5 bits), ReadReg2 (5 bits), WriteReg (5 bits), WriteData (32 bits). Las salidas son las siguientes: ReadData1(32 bits), ReadData2(32 bits).

En el banco de registros se declara una memoria de 32 posiciones donde cada una almacena 32 bits. Junto con ello utilizamos un initial begin para leer el archivo y cargar los datos en mem.

En la funcionalidad del banco de datos, utilizamos un bloque always para que al cambio de reloj si RegEn está activado entonces permite escribir el WriteData en la posición dada por WriteReg en memoria. Además, como principal función, tenemos la asignación de los ReadData a la posición dada por ReadReg en memoria para cada uno. La lógica sería escribir y leer datos dentro de este módulo.

A continuación, se adjunta una imagen del código en verilog:

```
// BancoReg
module BancoReg (
    input  wire        clk,
    input  wire        RegEn,
    input  wire [4:0]  ReadReg1,
    input  wire [4:0]  ReadReg2,
    input  wire [4:0]  WriteReg,
    input  wire [31:0] WriteData,
    output reg [31:0] ReadData1,
    output reg [31:0] ReadData2
);
    reg [31:0]mem[0:31];

    initial begin
                #100
                $readmemb("Bdatos", mem);
    end

    // Escritura sincrona
    always @(posedge clk) begin
        if (RegEn)
            mem[WriteReg] <= WriteData;
    end
    // Lectura combinacional

    always @(*) begin
        ReadData1 = mem[ReadReg1];
        ReadData2 = mem[ReadReg2];
    end
endmodule
```

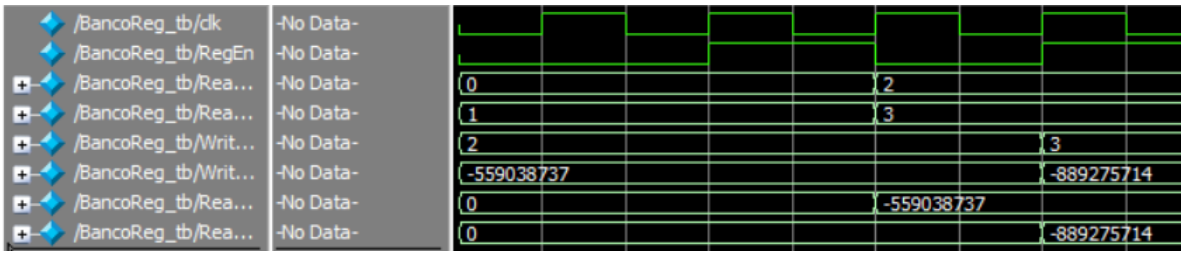
El testbench para probar que funcione correctamente este módulo es:

```
`timescale 1ns / 1ps
module BancoReg_tb;
    // Señales de reloj y habilitación
    reg clk;
    reg RegEn;
    // Índices de registros de lectura y escritura
    reg [4:0] ReadReg1;
    reg [4:0] ReadReg2;
    reg [4:0] WriteReg;
    // Datos de escritura
    reg [31:0] WriteData;
    // Datos de lectura
    wire [31:0] ReadData1;
    wire [31:0] ReadData2;
    // Instanciar el módulo bajo prueba
    BancoReg uut (
        .clk(clk),
        .RegEn(RegEn),
        .ReadReg1(ReadReg1),
        .ReadReg2(ReadReg2),
        .WriteReg(WriteReg),
        .WriteData(WriteData),
        .ReadData1(ReadData1),
        .ReadData2(ReadData2)
    );
    // Generación de reloj: periodo de 10ns
    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    // Estímulos
    initial begin
        // Inicialización
        RegEn = 0;
        ReadReg1 = 5'b00000;
        ReadReg2 = 5'b00001;
        WriteReg = 5'b00010;
        WriteData = 32'hDEADBEEF;
        // Esperar carga inicial desde archivo Bdatos
        #5;
        #10;  // Espera un ciclo de reloj combinacional
        // Habilitar escritura: escribir en reg 2
        RegEn = 1;
        WriteReg = 5'b00010;     // reg 2
        WriteData = 32'hDEADBEEF;
        #10; // Un ciclo de reloj
        // Deshabilitar escritura y leer de reg 2 y reg 3
        RegEn = 0;
        ReadReg1 = 5'b00010;     // reg 2
        ReadReg2 = 5'b00011;     // reg 3
        #10;
        // Escribir otro valor en reg 3
        RegEn = 1;
        WriteReg = 5'b00011;
        WriteData = 32'hCAFEBABE;
        #10;
        // Leer de nuevo reg 2 y reg 3
        RegEn = 0;
        ReadReg1 = 5'b00010;
        ReadReg2 = 5'b00011;
        #10;
        // Finalizar simulación
        #10;
        $finish;
    end

endmodule
```

Primero el testbench lee los registros 0 y 1 (lo que haya en Bdatos). Se escribe -559038737 en el registro 2, se activa RegEn y se espera un flanco de reloj. Se leen los registros 2 y 3 y se espera ver -559038737 en el registro 2 y lo que tenga el 3. Se escribe -889275714 en el registro 3, se activa RegEn otra vez. Se leen otra vez los registros 2 y 3 y ahora debe verse -559038737 en reg 2 y -889275714 en reg 3.

La escritura en los registros solo ocurre cuando RegEn = 1 y hay flanco positivo de reloj.

Adjunto las pruebas del testbench:



00000000	0
00000001	0
00000002	-559038737
00000003	-889275714
00000004	20
00000005	12
00000006	55
00000007	72
00000008	100
00000009	0
0000000a	0
0000000b	0
0000000c	0
0000000d	0
0000000e	999

ALU control:

La ALU control está conformada por 2 entradas y 1 sola salida. Las entras son las siguientes: ALUOp (3 bits) y funct (6 bits). Las salidas son las siguientes: ALUCtl (3 bits).

En la ALU control utilizamos un bloque always. En este usamos la entrada ALUOp para habilitar el uso de este módulo, con ello utilizamos como segundo parámetro funct para determinar la operación a hacer referencia dentro de la ALU. Con esto, según el caso, hace que ALU control determiné la operación que la ALU debe realizar. Este bloque se actualiza cada que cambie la entrada con el uso de @(*).

A continuación, se adjunta una imagen del código en Verilog.

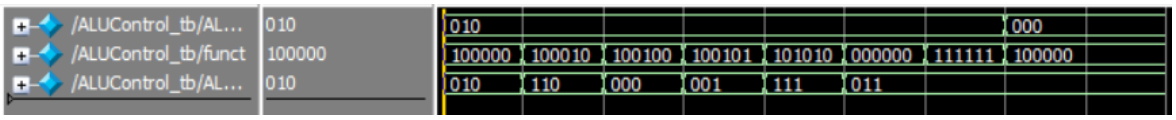
```verilog
// ALuControl
module ALuControl (
    input  wire [2:0] ALUOp,
    input  wire [5:0] funct,
    output reg  [2:0] ALUCtl
);
    always @(*) begin
        case (ALUOp)
            3'b010: begin // instrucciones R
                case (funct)
                    6'b100000: ALUCtl = 3'b010; // ADD
                    6'b100010: ALUCtl = 3'b110; // SUB
                    6'b100100: ALUCtl = 3'b000; // AND
                    6'b100101: ALUCtl = 3'b001; // OR
                    6'b101010: ALUCtl = 3'b111; // SLT
                    6'b000000: ALUCtl = 3'b011; // NOP
                    default:   ALUCtl = 3'b011;
                endcase
            end
        endcase
    end
endmodule
```

```verilog
`timescale 1ns / 1ps
module ALUControl_tb;
    // Señales de entrada
    reg [2:0] ALUOp;
    reg [5:0] funct;
    // Señal de salida
    wire [2:0] ALUCtl;
    // Instanciar el módulo bajo prueba
    ALuControl uut (
        .ALUOp(ALUOp),
        .funct(funct),
        .ALUCtl(ALUCtl)
    );

    // Estímulos
    initial begin
        // Prueba: instrucciones R (ALUOp = 010)
        ALUOp = 3'b010;

        // ADD
        funct = 6'b100000; #10;

        // SUB
        funct = 6'b100010; #10;

        // AND
        funct = 6'b100100; #10;

        // OR
        funct = 6'b100101; #10;

        // SLT
        funct = 6'b101010; #10;

        // NOP
        funct = 6'b000000; #10;

        // Función no definida (default)
        funct = 6'b111111; #10;

        // Prueba ALUOp diferente (sin definición explícita)
        ALUOp = 3'b000;
        funct  = 6'b100000; #10;

        // Finalizar simulación
        #10;
        $finish;
    end
endmodule
```

El testbench para probar que funcione correctamente este módulo es el de la imagen anterior.

Primero el testbench se fija en ALUOp = 3'b010, que indica una instrucción tipo R. Luego se prueban los valores del campo funct, 111111 no está definida y también se produce ALUCtl = 011 (valor por defecto). Se cambia ALUOp a 000, lo que no tiene comportamiento definido dentro del módulo, así que ALUCtl no cambia o se mantiene sin definir.

Adjunto las pruebas del testbench:



ALU:

El módulo ALU está conformado por 3 entradas y 2 salidas. Las entradas son las siguientes: OP1 (32 bits), OP2 (32 bits) y ALUCtl (3 bits). Las salidas son las siguientes: Res (32 bits) y ZF (1 bit).

El módulo ALU, como su nombre lo indica, ejecuta operaciones aritméticas y lógicas sobre sus entradas, esto por medio de la entrada ALUCtl, que especifica la operación a realizar.

En la funcionalidad del módulo, utilizamos un bloque always @(*) para que al detectar un cambio en la entrada ALUCtl, se seleccione la operación pertinente, esto mediante un bloque case. Las operaciones incluyen AND, OR, ADD, SUB, SLT y NOP. Además, hacemos una asignación donde si Res == 32’d0 entonces ZF tendrá un valor de 1. La lógica consiste en hacer cierta operación y devolver el resultado.

A continuación, se adjunta una imagen del código en Verilog.

```
// ALU
module ALU (
    input  wire [31:0] OP1,
    input  wire [31:0] OP2,
    input  wire [2:0]  ALUCtl,
    output reg  [31:0] Res,
    output wire        ZF
);
    always @(*) begin
        case (ALUCtl)
            3'b000: Res = OP1 & OP2;                    // AND
            3'b001: Res = OP1 | OP2;                    // OR
            3'b010: Res = OP1 + OP2;                    // ADD
            3'b110: Res = OP1 - OP2;                    // SUB
            3'b111: Res = (OP1 < OP2) ? 32'd1 : 32'd0;// SLT
            3'b011: Res = 32'd0;                        // NOP
            default: Res = 32'd0;
        endcase
    end
    assign ZF = (Res == 32'd0);
endmodule
```

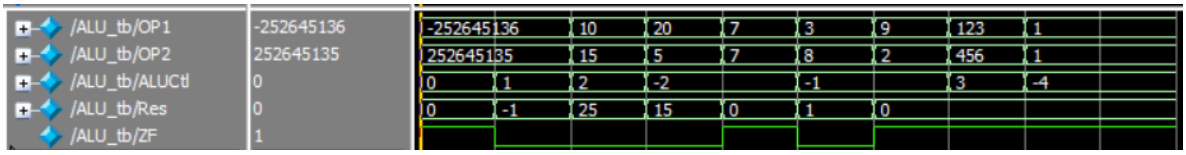
El testbench para probar que funcione correctamente este módulo es:

```
`timescale 1ns / 1ps

module ALU_tb;
    // Entradas al ALU
    reg [31:0] OP1;
    reg [31:0] OP2;
    reg [2:0]  ALUCtl;

    // Salidas del ALU
    wire [31:0] Res;
    wire        ZF;

    // Instanciar el módulo bajo prueba
    ALU uut (
        .OP1(OP1),
        .OP2(OP2),
        .ALUCtl(ALUCtl),
        .Res(Res),
        .ZF(ZF)
    );

    // Estímulos
    initial begin


        // Caso AND
        ALUCtl = 3'b000;
        OP1    = 32'b11110000111100001111000011110000;
        OP2    = 32'b00001111000011110000111100001111;
        #10;

        // Caso OR
        ALUCtl = 3'b001;
        OP1    = 32'b11110000111100001111000011110000;
        OP2    = 32'b00001111000011110000111100001111;
        #10;

        // Caso ADD
        ALUCtl = 3'b010;
        OP1    = 32'b00000000000000000000000000001010;
        OP2    = 32'b00000000000000000000000000001111;
        #10;

        // Caso SUB no cero
        ALUCtl = 3'b110;
        OP1    = 32'b00000000000000000000000000010100;
        OP2    = 32'b00000000000000000000000000000101;
        #10;
```

```
        // Caso SUB a cero (ZF=1)
        ALUCtl = 3'b110;
        OP1    = 32'b00000000000000000000000000000111;
        OP2    = 32'b00000000000000000000000000000111;
        #10;

        // Caso SLT verdadero
        ALUCtl = 3'b111;
        OP1    = 32'b00000000000000000000000000000011;
        OP2    = 32'b00000000000000000000000000001000;
        #10;

        // Caso SLT falso
        ALUCtl = 3'b111;
        OP1    = 32'b00000000000000000000000000001001;
        OP2    = 32'b00000000000000000000000000000010;
        #10;

        // Caso NOP
        ALUCtl = 3'b011;
        OP1    = 32'b00000000000000000000000001111011;
        OP2    = 32'b00000000000000000000000111001000;
        #10;

        // Caso default (invalido)
        ALUCtl = 3'b100;
        OP1    = 32'b00000000000000000000000000000001;
        OP2    = 32'b00000000000000000000000000000001;
        #10;

        // Finalizar simulación
        #10;
        $finish;
    end

endmodule
```

Este testbench verifica que el módulo ALU realice correctamente diferentes operaciones lógicas y aritméticas según la señal de control ALUCtl, esto lo hace haciendo las operaciones con valores al azar, y que también genere correctamente la señal de Zero Flag (ZF) cuando el resultado es cero.

Adjunto las pruebas del testbench:



Multiplexor:

El módulo MuxWriteData está conformado por 3 entradas y 1 salida. Las entradas son las siguientes: entrada0 (WIDTH-1 bits), entrada1 (WIDTH-1bits) y sel (1 bit). La salida es la siguiente: salida (WIDTH-1 bits).

El módulo MuxWriteData está configurado para permitir cambiar el tamaño de los valores de entrada y salida por medio del parámetro WIDTH que al instanciar podemos cambiar, en caso contrario el valor por defecto será de 32 bits.

En la funcionalidad del módulo, primeramente, utilizamos un bloque always @(*) para que al detectar un cambio en la entrada sel, se asigne el valor correspondiente a salida. Si sel es 1'b1, se iguala a la entrada1, en caso contrario, si sel es 1'b0, se iguala a la entrada0. La lógica consiste en alternar entre las dos opciones de entrada y dirigir el resultado hacia la salida.

A continuación, se adjunta una imagen del código en Verilog.

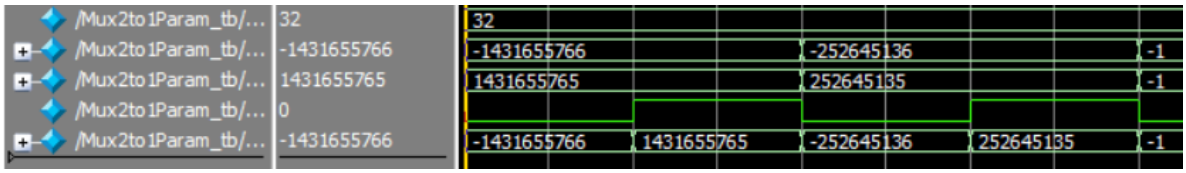
```
module MuxWriteData #(
    parameter WIDTH = 32  // Permite configurar el tamaño al instanciarlo
)(
    input  wire [WIDTH-1:0] entrada0,  // opción 0
    input  wire [WIDTH-1:0] entrada1,  // opción 1
    input  wire             sel,       // selector
    output reg  [WIDTH-1:0] salida     // resultado
);
    always @(*) begin
        salida = sel ? entrada1 : entrada0;
    end
endmodule
```

El testbench para probar que funcione correctamente este módulo es:

```
`timescale 1ns / 1ps
module Mux2to1Param_tb;
    // Parámetro de ancho (utilizando valor por defecto 32)
    localparam WIDTH = 32;
    // Entradas
    reg [WIDTH-1:0] entrada0;
    reg [WIDTH-1:0] entrada1;
    reg sel;
    // Salida
    wire [WIDTH-1:0] salida;
    // Instanciar el MUX parametrizable
    Mux2to1Param #(.WIDTH(WIDTH)) uut (
        .entrada0(entrada0),
        .entrada1(entrada1),
        .sel(sel),
        .salida(salida)
    );
    // Estímulos
    initial begin
        // Patrón de prueba inicial
        entrada0 = 32'b10101010101010101010101010101010;
        entrada1 = 32'b01010101010101010101010101010101;
        sel      = 1'b0;
        #10; // Con sel=0, salida debe ser entrada0

        sel      = 1'b1;
        #10; // Con sel=1, salida debe ser entrada1

        // Cambiar valores y repetir
        entrada0 = 32'b11110000111100001111000011110000;
        entrada1 = 32'b00001111000011110000111100001111;
        sel      = 1'b0;
        #10;

        sel      = 1'b1;
        #10;

        // Probar con ambos entran iguales
        entrada0 = 32'b11111111111111111111111111111111;
        entrada1 = 32'b11111111111111111111111111111111;
        sel      = 1'b0;
        #10;

        sel      = 1'b1;
        #10;

        // Finalizar simulación
        #10;
        $finish;
    end

endmodule
```

Con este testbench se verifica que el mux selecciona correctamente la entrada según la señal sel. La salida cambia de forma **esperada y precisa** en todos los casos. También comprueba que el módulo funciona bien incluso si ambas entradas son iguales.

Adjunto las pruebas del testbench:



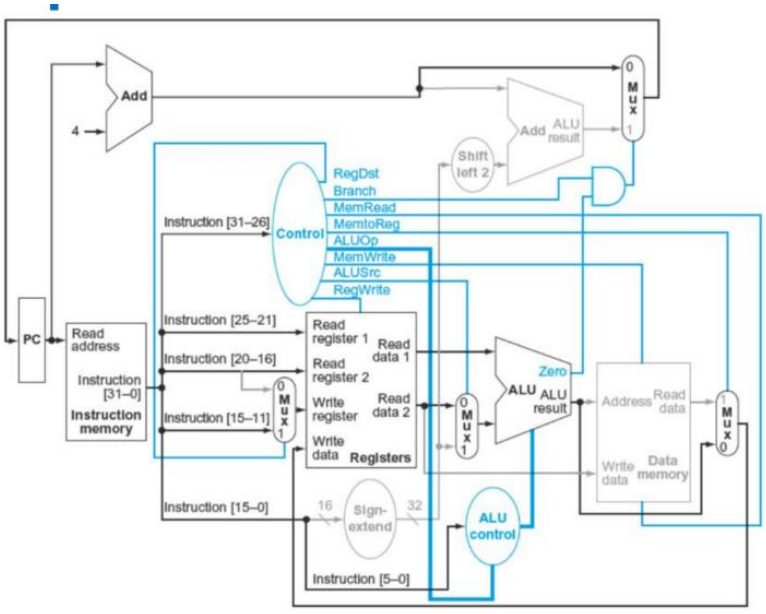
Fase 1 Datapath Completo (instrucciones tipo R):

El módulo DPTR (o data path tipo R) está conformado por 1 entrada y 2 salidas. La entrada es la siguiente: clk (1 bit). Las salidas son las siguientes: Instr (32 bits), PCout (32 bits). Este módulo representamos la implementación de una CPU simplificada, esta de tipo MIPS.

En el módulo DPTR definimos varios buses internos para interconectar sus componentes, estos son los siguientes:

- PCin, PCnext (32 bits). Esto para las conexiones del ciclo fetch.
- OpCode (6 bits). Esto para sacar la OpCode de la instrucción.
- rs, rt, rd (5 bits). Esto para sacar los registros de la instrucción.
- funct (6 bits). Esto para sacar fuction de la instrucción.
- MemToReg, MemToWrite, RegWrite, RegDst, MemRead, ALUSrc, ZF, Br_AND_ZF Branch (1 bit). Esto para conectar salidas del banco de registros y de control.
- ALUOp (3 bits). Esto para enviar el tipo de instrucción (R, I, J).
- C1, C2, C3, C5, WriteData, OP2 (32 bits) y C4 (3 bits). Esto para conectar entre distintos módulos.
- WriteReg (5 bits). Para enviar una posición al banco de registros.

Aquí instanciamos los distintos componentes que obedezcan el siguiente gráfico, omitiendo las partes grises ajenas a las instrucciones de tipo R:



Los módulos que se trabajan aquí son:

- PC
- ADD4
- Mux2to1Param
- MEMI
- UnidadDeControl
- BancoReg
- ALuControl
- ALU

Podemos ver el código que expresa todo esto en las siguientes imágenes:

```
// Diego Jared Jimenez Silva
// Gael Ramses Alvarado Lomeli

module DPTR (
    input  wire        clk,
    output wire [31:0] Instr,
    output wire [31:0] PCout
);
// Buses internos
    wire [31:0] PCin, PCnext;
    wire [5:0]  OpCode = Instr[31:26];
    wire [4:0]  rs     = Instr[25:21];
    wire [4:0]  rt     = Instr[20:16];
    wire [4:0]  rd     = Instr[15:11];
    wire [5:0]  funct  = Instr[5:0];

// Señales de control
    wire MemToReg, MemToWrite, RegWrite, RegDst, MemRead, ALUSrc, Branch
    ,ZF, Br_AND_ZF;
    wire [2:0]  ALUOp;

// Más buses internos
    wire [31:0] C1, C2, C3, C5, WriteData, OP2;
    wire [2:0]  C4;
    wire [4:0]  WriteReg;

// Program Counter
    PC pc_inst (
        .IN(PCin),
        .CLK(clk),
        .OUT(PCout)
    );

// Suma 4 al PC
    ADD4 add_inst (
        .A(PCout),
        .RES(PCnext)
    );

// Multiplexor 1
   Mux2to1Param #(.WIDTH(32)) MUXPC (
    .entrada0(PCnext),
```

```
    .entrada1(32'b0),       // Dirección de salto
    .sel(Br_AND_ZF),
    .salida(PCin)
);

// Memoria de instrucciones
    MEMI memi_inst (
        .DR(PCout),
        .INS(Instr)
    );

// Control
    UnidadDeControl UC(
        .OpCode(OpCode),
        .MemRead(),         // Conectado a modulo aun inexistente
        .MemToReg(MemToReg),
        .MemToWrite(MemToWrite),
        .ALUOp(ALUOp),
        .RegWrite(RegWrite),
        .RegDst(RegDst),
        .ALUSrc(ALUSrc),
        .Branch(Branch)
    );

// Multiplexor 2
    Mux2to1Param #(.WIDTH(5)) MUXWR (
    .entrada0(rt),
    .entrada1(rd),
    .sel(RegDst),
    .salida(WriteReg)
    );


// Banco de registros
    BancoReg BR(
        .clk(clk),
        .RegEn(RegWrite),
        .ReadReg1(rs),
        .ReadReg2(rt),
        .WriteReg(WriteReg),
        .WriteData(WriteData),
        .ReadData1(C1),
        .ReadData2(C2)
    );
```

```
    );


// ALU control
    ALuControl AC(
        .ALUOp(ALUOp),
        .funct(funct),
        .ALUCtl(C4)
    );

// Multiplexor 3
    Mux2to1Param #(.WIDTH(32)) MUXAL (
        .entrada0(C2),
        .entrada1(32'b0),   // Inmediato con extensión de signo
        .sel(ALUSrc),
        .salida(OP2)
    );

// ALU
    ALU alu(
        .OP1(C1),
        .OP2(OP2),
        .ALUCtl(C4),
        .Res(C3),
        .ZF(ZF)
    );

// Agregamos el and que une branch y zflag
assign Br_AND_ZF = ZF & Branch;

// --- NO UTILIZADO PARA TIPO R ---
// MemDatos
    MemDatos MD(
        .clk(clk),
        .MemToWrite(MemToWrite),
        .Address(C3),
        .WriteData(C2),
        .ReadData(C5)
    );




// Multiplexor 4
    Mux2to1Param #(.WIDTH(32)) MUXWD (
    .entrada0(C3),
    .entrada1(C5),
    .sel(MemToReg),
    .salida(WriteData)
    );

endmodule
```

Explicaremos paso a paso cómo funciona el módulo:

Primero tenemos la siguiente instrucción dividida en bytes, siendo repartida en 4 posiciones dentro del txt de la memoria de instrucciones: 00000001111010011010000000100010.

En base a esto, lo primero que sucede es que el ciclo fetch le otorga al módulo MEMI la posición por la cual empezar, en este caso cero porque sería el primer ciclo del modulo PC.

Con ello MEMI lee la posición dada mas las 3 siguientes, generando así la instrucción de 32 bits para todo el sistema. Esto significaría que entaría nuestra instrucción guardada anteriormente (00000001111010011010000000100010).

Con ello ahora esta se divide en los respectivos cables, mandando así la señal a los distintos módulos.

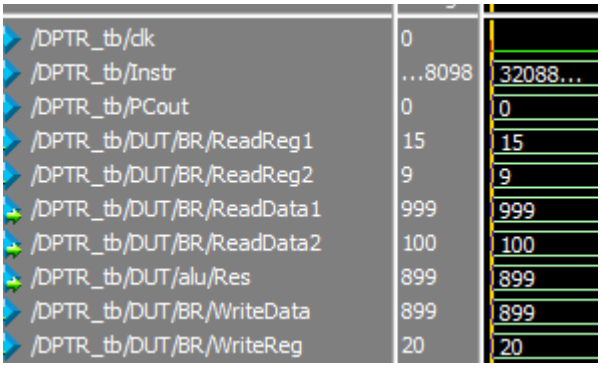
- Opcode: 000000.
- rs: 01111.
- rt: 01001.
- rd: 10100.
- funct: 100010.

Dirigiéndonos primeramente con el Control, aquí entra la señal OpCode, con esta determinaríamos el tipo de instrucción y sus salidas pertinentes, en este caso, apagamos todas las señales por defecto y encendemos RegWrite, ALUOp y RegDst ya que escribiremos al banco de registros, utilizaremos 010 para indicar a la ALU control que se hará caso al funct ya que las instrucciones son tipo r y lo utilizamos para saber la operación exacta a realizar en la alu, y finalmente encendemos RegDst para enviar la señal a un multiplexor (esto para su uso en las futuras fases).

Ahora mientras en el banco de registros se utiliza rs y rt para determinar las posiciones donde leeremos dentro del archivo, por otra parte, utilizaremos rd para determinar la posición donde guardaremos el resultado. Como salida tenemos lo que se obtuvo de las posiciones rs y rt del archivo .txt. Estos los madamos directamente a la ALU (pasa por un multiplexor pero que no es utilizado, esto es para las siguientes fases).

Dentro de la ALU nos llegaron los datos guardados y la operación a la que menciona function, esto nos indica que haremos una resta de los datos guardados, en este caso 999 y 100. Con ello pasamos el resultado directo a el banco de registros, guardando así en la posición 20 por lo que nos indica rd.

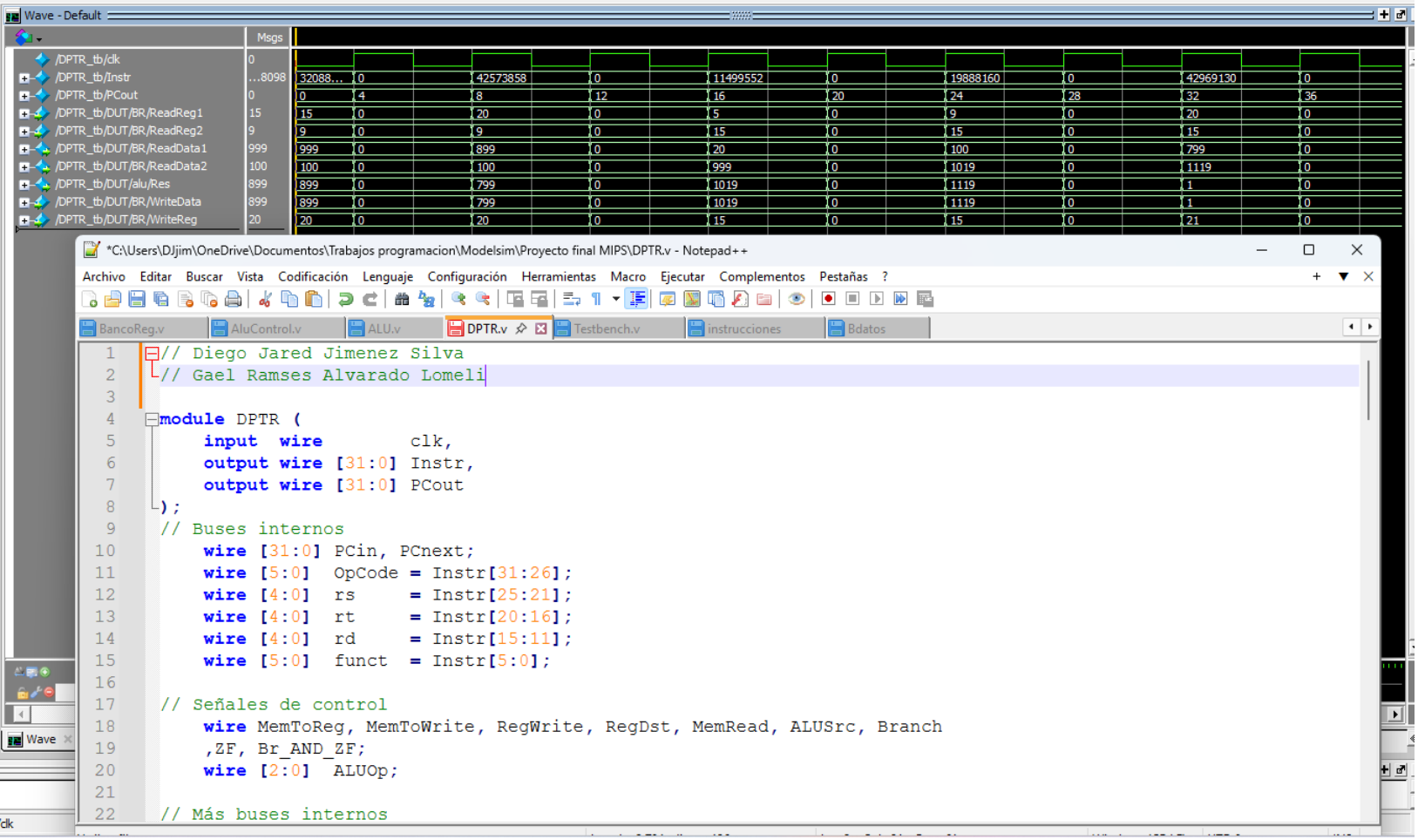
Con esto podemos ver el funcionamiento del modulo y el cómo interactúan todos los componentes para procesar una instrucción de tipo R en su totalidad.



Aquí comprobamos las instrucciones en ensamblador dejadas por el profesor, donde podemos ver cómo se mueven igual que como lo explicamos a través de los módulos y finalmente se guarda en el banco de registros:

Las instrucciones fueron:

- sub $20, $15, $9
- NOP
- sub $20, $20, $9
- NOP
- add $15, $5, $15
- NOP
- add $15, $9, $15
- NOP
- slt $21, $20, $15



Memory Data - /DPTR_tb/DUT/BR/mem - Def

0	0
1	0
2	0
3	0
4	0
5	20
6	12
7	55
8	72
9	100
10	0
11	0
12	0
13	0
14	0
15	1119
16	0
17	0
18	0
19	0
20	799
21	1
22	0
23	0
24	0
25	0
26	0
27	0
28	0
29	0
30	0
31	0

Conclusiones

Diego Jared Jimenez Silva

Primeramente, empezando por partes, la fase uno fue sencillo, me gustó mucho porque yo y mi compañero pudimos refozar una actividad anterior donde trabajamos con el banco de registros + ALU + memoria de datos (jericalla evolution), con la fase uno pudimos reforzar y mejorar en la implementación de los módulos, además me gustó ver la documentación de los MIPS para saber cómo iban a fluir los datos y como se declaraban las instrucciones, me gustó investigar y adaptar esto a instrucciones tipo R.

Gael Ramsés Alvarado Lomelí

Como conclusión de esta primera fase puedo decir que a veces errores tan minúsculos pueden hacer que pases 3 días haciendo lo mismo una y otra vez, pero al final de los errores se aprende. Implementar el ciclo fetch al datapath ya hecho desde la actividad 9 fue interesante ya que había distintas maneras de hacerlo. Hacer los multiplexores sin los demás modulo de las siguientes fases fue raro pero lo pudimos hacer con éxito.

SEGUNDA PARTE

TERCERA PARTE

CURSO