Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías (CUCEI)

25A-Arquitectura de Computadoras

Profesor: JORGE ERNESTO LOPEZ ARCE DELGADO

Alumnos: GAEL RAMSES ALVARADO LOMELI: 220529474

DIEGO JARED JIMENEZ SILVA: 220559845

Nombre del equipo: Daycothy.

[Equipo de 2] [Jericalla-evolución] BR+ ALU + MemDatos

Actividad 06

Semana 06 - ALU + Mem

Introducción:

La ALU (Unidad Aritmético-Lógica), es la parte del procesador de una computadora que se encarga de realizar operaciones matemáticas y lógicas. Para decirlo de una manera más sencilla, podríamos decir que es el cerebro matemático de una computadora.

Algunas de las operaciones aritméticas que hace la ALU son la suma, resta, multiplicación y división, aunque cabe mencionar que la multiplicación y división pueden ser más complejas.

Ahora, para las operaciones lógicas tenemos, por ejemplo: mayor que, menor que, igual, y operadores como AND, OR, NOT.

El funcionamiento de la ALU es muy simple, primero la ALU recibe los datos de la memoria o de los registros del procesador, luego, un circuito de control le dice qué operación realizar y por último, devuelve el resultado para ser almacenado o usado en otras instrucciones.

Objetivo:

El objetivo principal de esta práctica es implementar la ALU realizada en actividades anteriores en un módulo llamado “Jericalla_evolucion”, donde, además, se implementarán los siguientes módulos:

1.- El banco de registros (BR).

2.- La memoria de datos (MemD). 3.-

El módulo de control (Control).

4.- El demultiplexor (Demux).

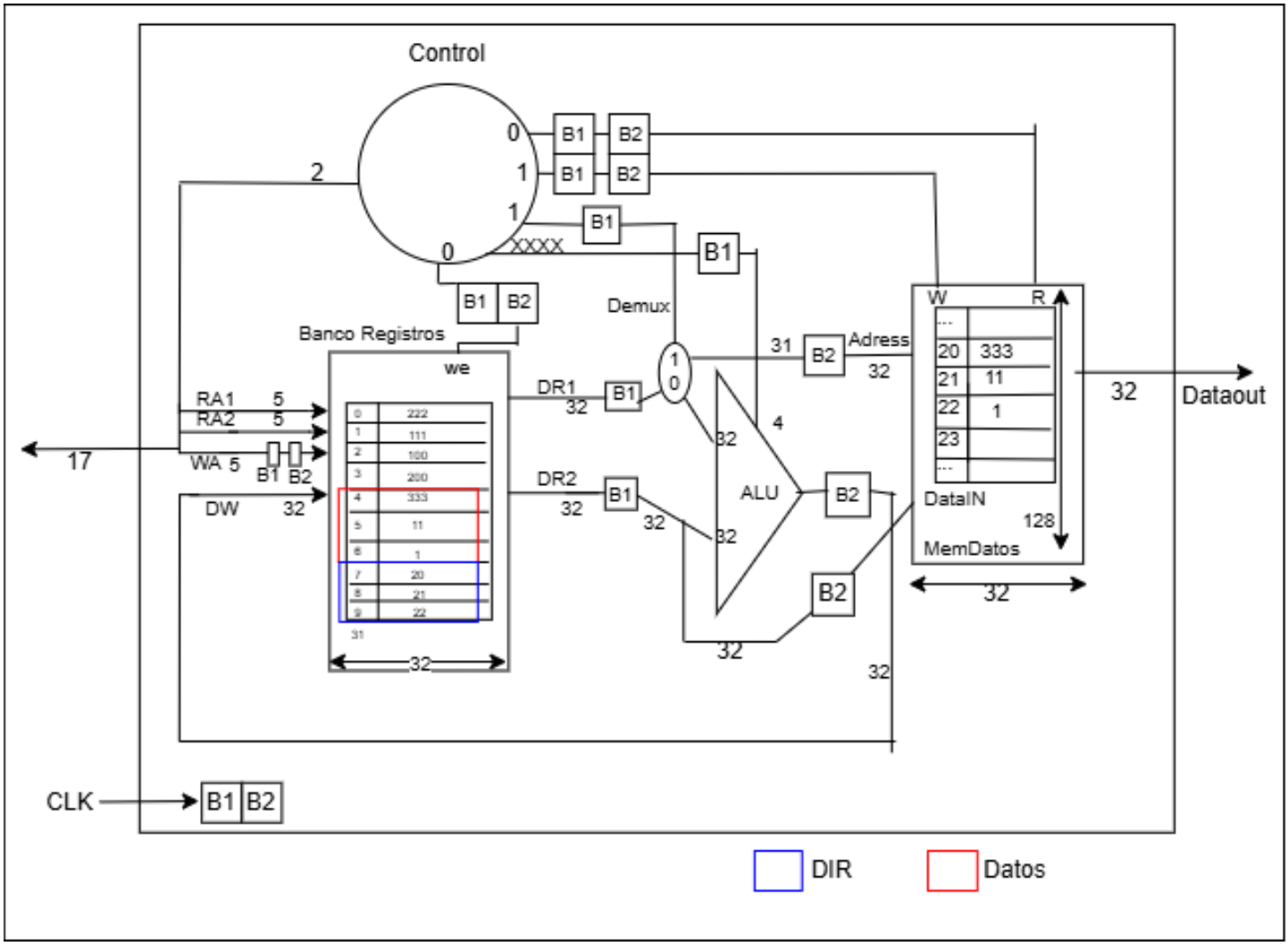
5.- Algún buffer o “Bus” intermedio.

6.- Un archivo de texto con los datos (datos.txt).

Al final “Jericalla_evolucion”, junto con los demás módulos simularan un mini procesador.

Desarrollo (como se hizo el programa y como se comprobó el correcto funcionamiento):

Diagrama propuesto por el profe:



Empezaremos por la ALU ya que ya la teníamos hecha: La ALU es muy similar a como la teníamos anteriormente en otras prácticas, simplemente reordenamos las operaciones según lo estipulado en la actividad y declaramos correctamente cada caso y sus respectivos bits de activación.

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

module ALU(
    input [31:0] A,
    input [31:0] B,
    input [3:0] OP,
    output reg [31:0] RES
);

//En esta ocasion solo mandaremos a llamar
always @(*) begin
    case(OP)
        4'b0000: RES = A & B;       // AND
        4'b0001: RES = A | B;       // OR
        4'b0010: RES = A + B;       // ADD
        4'b0110: RES = A - B;       // SUBSTRACT
        4'b0111: RES = (A < B) ? 32'd1 : 32'd0;  // TERNARIA
        4'b1100: RES = ~(A | B);     //NOR
        default: RES = 32'd0;
    endcase
end
endmodule
```

Banco de registros: Primeramente, definimos nuestro módulo del banco de registros. Con esto declaramos las entradas y las salidas del módulo con su tamaño respectivo y añadiendo comentarios para especificar la función de la variable. Declaramos la variable de 32 bits y de 32 posiciones que

utilizaremos para el módulo. En un initial leemos el archivo Bdatos para operar con los valores dados por el profe. Y finalmente declaramos la funcionalidad, sí Write Enable está activo entonces escribimos el valor de Data Write, en cualquier caso, asignamos los valores de salida a el valor de la memoria que obtengamos según la dirección.

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

module BR(
    input [4:0] RA1,        // Read Adress 1
    input [4:0] RA2,        // Read Adress 1
    input [4:0] WA,         // Write Adress
    input [31:0] DW,        // Data Write
    input WE,               // Write Enable
    output reg [31:0] DR1,  // Data Read 1
    output reg [31:0] DR2   // Data Read 1
);

reg [31:0]mem[0:31]; // Memoria con valores de 32 bits y 32 posiciones 2^5

//Leemos el archivo
initial
begin
    #100
    $readmemb("Bdatos", mem);
end

//Asignamos
always @(*) begin
    if (WE) begin
        mem[WA] <= DW;  // Escribimos si WE está activo
    end
end
always @(*) begin
    DR1 = mem[RA1]; // Leemos mem[RA1]
    DR2 = mem[RA2]; // Leemos mem[RA2]
end
endmodule
//--------------
```

Control:

En el control recibe los primeros bits de lo que sería la instrucción, entonces declaramos la entrada de dos bits y sus salidas derivadas correspondientes, en concreto son 4 de un solo bit y una última de 4 bits, esta última sería la operación de la ALU. Dentro de un always declaramos los casos que se dibujaron en el pizarrón por el maestro, siendo la suma, la resta, la ternaria y el sw. En cada caso asignamos las respectivas salidas correspondientes a las operaciones o al sw junto con la operación respectiva de la ALU para hacer la operación pertinente.

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

module CTRL(
    input [1:0] IN,         // Entrada de 2 bits
    output reg WE,          // Salida del banco de registros (declarada como reg para uso en always)
    output reg DMUX,        // Para filtrar el caso SW
    output reg W,           // Escritura de MemDatos
    output reg R,           // Lectura de MemDatos
    output reg [3:0] ALUOP  // Opción de operación en la ALU
);

always @(*) begin
    // Decodificador
    case (IN)
        // Caso de la suma
        2'b00: begin
            WE = 1'b1;       // Escribimos en el banco de registros
            DMUX = 1'b0;     // Describimos si uitilizaremos la ALU
            W = 1'b0;        // No utilizamos MemDatos
            R = 1'b0;        // No utilizamos MemDatos
            ALUOP = 4'b0010; // Mandamos la instrucción a la ALU
        end

        // Caso de la resta
        2'b01: begin
            WE = 1'b1;
            DMUX = 1'b0;
            W = 1'b0;
            R = 1'b0;
            ALUOP = 4'b0110;
        end

        // Caso de la ternaria
        2'b10: begin
            WE = 1'b1;
            DMUX = 1'b0;
            W = 1'b0;
            R = 1'b0;
            ALUOP = 4'b0111;
        end

        // Caso StoreWord
        2'b11: begin
            WE = 1'b0;
            DMUX = 1'b1;
            W = 1'b1;
            R = 1'b0;        // En esta actividad no leeremos por el momento
            ALUOP = 4'b1111; // No utilizamos la ALU aqui, forzamos el caso default
        end
    endcase
end
endmodule
```

Buffer: En el buffer utilizamos #() para especificar un valor más adelante y que este no sea fijo, no sin antes inicializarlo con el valor de 32 bits. Con esto en mente en la entrada utilizamos el valor recibido menos uno para cumplir con los índices y las declaraciones, agregamos el respectivo reloj y un out del mismo tamaño que in. Agregamos la funcionalidad de que por cada positivo del reloj entonces se liberan los datos de entrada en la salida.

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

module BF #(parameter WIDTH = 32) (
    input [WIDTH-1:0] IN,
    input CLK,
    output reg [WIDTH-1:0] OUT
);
    always @(posedge CLK) begin // Cada positivo en el reloj
        OUT <= IN;  // Utilizamos <= para las distintas asignaciones en paralelo
    end
endmodule

//---------------
```

DMUX: En este módulo utilizamos una entrada de 32 bits pues los redirigiremos a su salida respectiva por medio de este mismo módulo. Declaramos un selecto para saber qué salida tomar y las dos salidas respectivas. En su always utilizamos un case y asignamos las variables, 0 si va a la ALU y 1 Si va a la memoria de datos.

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

module DX(
    input [31:0] IN,
    input select,
    output reg [31:0] OUT0, // Salida 0
    output reg [31:0] OUT1  // Salida 1
);

always @(*) begin
    // Filtramos
    case (select)
        1'b0: begin
            OUT0 = IN;      // Va a la ALU
            OUT1 = 32'b0;   // OUT1 no se usa en este caso
        end
        1'b1: begin
            OUT0 = 32'b0;   // OUT0 no se usa en este caso
            OUT1 = IN;      // Va como dirección de memoria hacia la MD
        end
    endcase
end
endmodule
```

Memoria de datos: Aquí utilizaremos la salida 1 del demux y sus respectivos valores, por ello utilizamos entradas de 32 bits, un w para escribir, un r para leer y una salida del módulo, así como la memoria respectiva, esta vez de 128 posiciones, pero con valores de 32 bits al igual que en el banco de registros. Utilizamos un initial para leer la memoria, esta vez llamada Mdatos y acceder. En su funcionalidad simplemente declaramos un if para asegurarnos de no leer y escribir en una misma instrucción y empezamos, para escribir, utilizamos mem con la dirección dada y así asignamos el valor de entrada, y viceversa en caso de lectura.

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

//Memoria de datos
module MD(
    input [31:0] ADR,
    input [31:0] DIN,
    input W,
    input R,
    output reg [31:0] DOUT
);

reg [31:0]mem[0:127]; // Memoria con valores de 32 bits y 128 posiciones 2^7

//Leemos el archivo
initial
begin
    #100
    $readmemb("Mdatos", mem);
end

//Asignaciones
// Utilizamos <= por ser comun en operaciones de lectura y escritura
// para leer y escribir difernetes datos en paralelo si es necesario
always @(*) begin
    if (W && !R) begin
        mem[ADR] = DIN; // Escritura
    end
    if (R && !W) begin
        DOUT = mem[ADR]; // Lectura
    end
    else begin
        DOUT = 32'd0;   // Default
    end
end
endmodule
```

Finalmente llegamos al módulo final.

Primeramente, usamos una entrada de 17 bits, una entrada del reloj y una salida de 32bits. Seguido de esto empezamos a instanciar. Primeramente, instanciamos el módulo de control, asignando los primeros dos bits de control y las salidas con cables, seguido de esto instanciamos el banco de registros utilizando instrucción, unos cables para las salidas y otros cables pasados ya por buffers a WA y DW.

Terminando con ello agrupamos todo lo que se tenga que pasar por el primer buffer en una variable, está la utilizaremos dentro de la instancia del buffer, después de haberse operado, descompondremos la salida para asignarse a los debidos cables que utilizaremos más adelante, usando B1 como referencia de haber pasado por el primer buffer. Terminando lo anterior ahora si instanciamos dos veces el demultiplexor para pasar los datos del banco de datos, pero ya habiendo pasado por el buffer uno. con ello obtendremos sus respectivas salidas para los dos casos. Instanciamos la ALU, metemos los cables del demultiplexor que están dirigidos a la ALU y operamos, con ello obtendremos un cable de salida que enviaremos a un buffer más tarde. Finalmente usamos el segundo buffer, donde se procesarán más datos puesto que ahora metemos 3 variables de 32 bits,

con ello hacemos lo mismo y obtendremos las nuevas variables finales, estas están con terminación B2 haciendo referencia de haber pasado por el primer buffer.

Finalmente, después de todo el proceso, instanciamos la memoria de datos, asignamos los valores del banco de datos que ya fueron pasados por el buffer 2 entre otros, y finalmente utilizamos la salida del módulo jericalla. Como último le asignamos los valores correspondientes de DW y WE al banco de datos, puesto que estos ya fueron operados por el buffer 2 y ya pueden asignarse ahora sí.

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

// Modulo Jericalla
module Jericalla(
    input [16:0] instruccion,
    input clk,
    output [31:0] salida
);

// Buffer para WA al inicio
wire [4:0] WAB1, WAB2;
// Control
wire C1, C2, C3, C4;
wire [3:0] CALU;
wire [3:0] CALUB1;
wire C1B1, C2B1, C3B1, C4B1;
wire C1B2, C3B2, C4B2;
// Banco de datos
wire [31:0] CDR1, CDR2;
wire [31:0] CDR1B1, CDR2B1;
wire [31:0] CDR1DX, CDR2DX;
wire [31:0] CDR1B2, CDR2B2;
//Buffer
wire [76:0] cable_combinado1;
wire [76:0] cable_salida1;
wire [103:0] cable_combinado2;
wire [103:0] cable_salida2;
// Demultiplexor
wire [31:0]DM1,DM2;
// ALU
wire [31:0] CRES;
wire [31:0] CRESB2;

// Instanciamos
CTRL    CtTest (.IN(instruccion [16:15]),.WE(C1),.ALUOP(CALU),.DMUX(C2),.W(C3),.R(C4));
BR      BrTest (.RA1(instruccion [9:5]),.RA2(instruccion [4:0]),.WA(instruccion [14:10]),.DW(CRESB2),.WE(C1B2),.DR1(CDR1),.DR2(CDR2));
DX      D1Test (.IN(CDR1B1),.select(C2B1),.OUT0(DM1),.OUT1(CDR1DX));
DX      D2Test (.IN(CDR2B1),.select(C2B1),.OUT0(DM2),.OUT1(CDR2DX));
ALU     ALUTest(.A(DM1),.B(DM2),.OP(CALUB1),.RES(CRES));
MD      MDTest (.ADR(CDR1B2),.DIN(CDR2B2),.W(C3B2),.R(C4B2),.DOUT(salida));
```

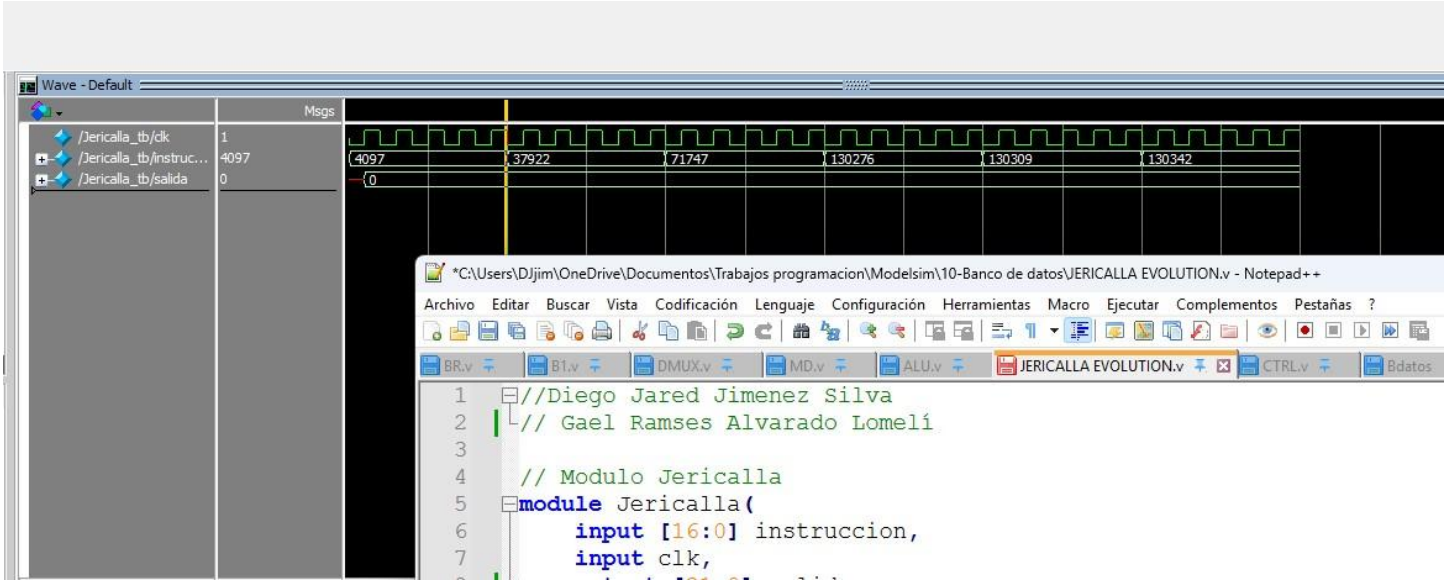
```
// Pasamos todos los datos por el buffer 1
assign  cable_combinado1 = {instruccion[14:10],C1,C2,C3,C4,CALU,CDR1,CDR2};
BF#(77) B1Test (.IN(cable_combinado1),.CLK(clk),.OUT(cable_salida1));

assign WAB1 = cable_salida1[76:72];     // WA ocupa los bits del 76 al 72
assign C1B1 = cable_salida1[71];        // C1 ocupa el bit 71
assign C2B1 = cable_salida1[70];        // C2 ocupa el bit 70
assign C3B1 = cable_salida1[69];        // C3 ocupa el bit 69
assign C4B1 = cable_salida1[68];        // C4 ocupa el bit 68
assign CALUB1 = cable_salida1[67:64];   // CALU ocupa los bits del 67 al 64
assign CDR1B1 = cable_salida1[63:32];   // DR1 ocupa los bits del 63 al 32
assign CDR2B1 = cable_salida1[31:0];    // DR2 ocupa los bits del 31 al 0

// Pasamos todos los datos por el buffer 2
assign  cable_combinado2 = {WAB1,C1B1,C3B1,C4B1,CRES,CDR1DX,CDR2DX};
BF#(104)    B2Test (.IN(cable_combinado2),.CLK(clk),.OUT(cable_salida2));

assign WAB2 = cable_salida2[103:99];    // WA ocupa los bits del 103 al 99
assign C1B2 = cable_salida2[98];        // C1 ocupa el bit 98
assign C3B2 = cable_salida2[97];        // C3 ocupa el bit 97
assign C4B2 = cable_salida2[96];        // C4 ocupa el bit 96
assign CRESB2 = cable_salida2[95:64];   // CRES ocupa los bits del 95 al 64
assign CDR1B2 = cable_salida2[63:32];   // CDR1DX ocupa los bits del 63 al 32
assign CDR2B2 = cable_salida2[31:0];    // CDR2DX ocupa los bits del 31 al 0
endmodule


// Testbench
module Jericalla_tb();
reg [16:0] instruccion;
reg clk;
wire [31:0] salida;

// Instancia del módulo Jericalla
Jericalla JTestBench (.instruccion(instruccion),.clk(clk),.salida(salida));
```

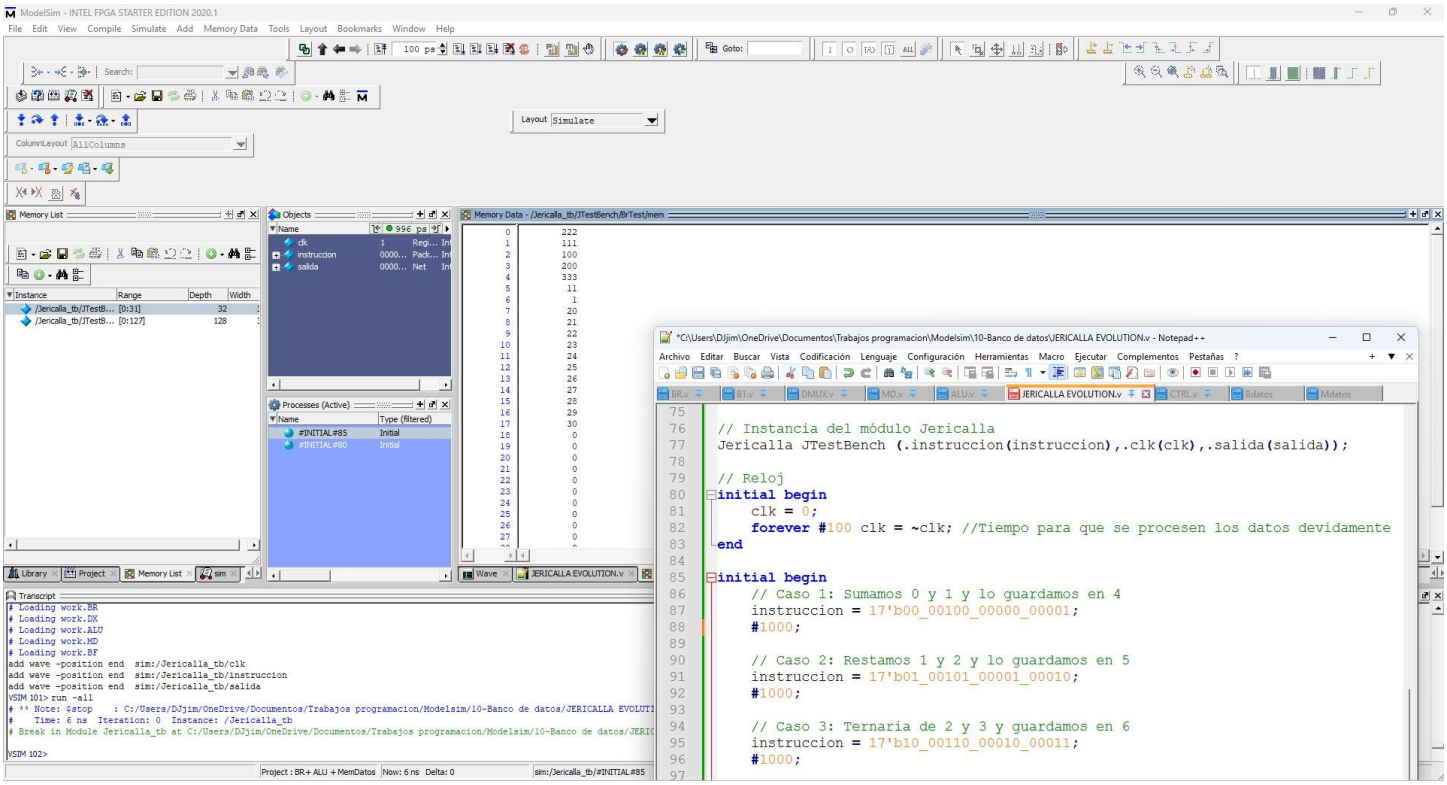
```
// Reloj
initial begin
    clk = 0;
    forever #100 clk = ~clk; //Tiempo para que se procesen los datos devidamente
end

initial begin
    // Caso 1: Sumamos 0 y 1 y lo guardamos en 4
    instruccion = 17'b00_00100_00000_00001;
    #1000;

    // Caso 2: Restamos 1 y 2 y lo guardamos en 5
    instruccion = 17'b01_00101_00001_00010;
    #1000;

    // Caso 3: Ternaria de 2 y 3 y guardamos en 6
    instruccion = 17'b10_00110_00010_00011;
    #1000;

    // Caso 4: Store Word del valor de la pos 4 guardado en el valor de la pos 7 en MemData
    instruccion = 17'b11_11111_00111_00100;
    #1000;

    // Caso 5: Store Word del valor de la pos 5 guardado en el valor de la pos 8 en MemData
    instruccion = 17'b11_11111_01000_00101;
    #1000;

    // Caso 6: Store Word del valor de la pos 6 guardado en el valor de la pos 9 en MemData
    instruccion = 17'b11_11111_01001_00110;
    #1000;

    // Finaliza la simulación
    $stop;
end
endmodule
```

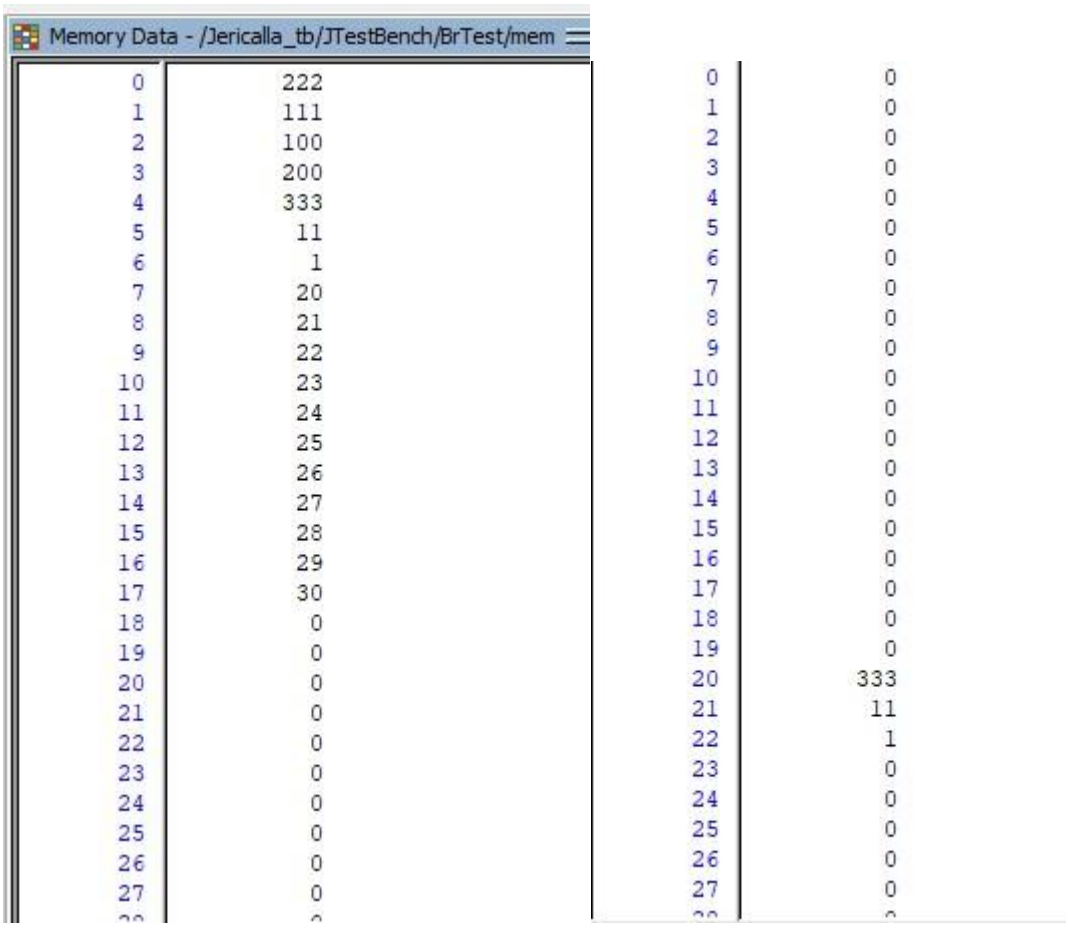
Banco de datos:

```
00000000000000000000000011011110
00000000000000000000000001101111
00000000000000000000000001100100
00000000000000000000000011001000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000010100
00000000000000000000000000010101
00000000000000000000000000010110
00000000000000000000000000010111
00000000000000000000000000011000
00000000000000000000000000011001
00000000000000000000000000011010
00000000000000000000000000011011
00000000000000000000000000011100
00000000000000000000000000011101
00000000000000000000000000011110
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
```

Aquí tenemos el resultado del wave:







Como podemos ver, obtuvimos los resultados esperados.

Tabla de resultados

No. de Instrucción	Instrucción en ensamblador	Resultado	Op	RD	RS1	RS2
0	SUMA $4, $0, $1	333	00	00100	00000	00001
1	RESTA $5, $1, $2	11	01	00101	00001	00010
2	STL $6, $2, $3	1	10	00110	00010	00011
3	SW $7, $4	20	11	11111	00111	00100
4	SW $8, $5	21	11	11111	01000	00101
5	SW $9, $6	22	11	11111	01001	00110

Conclusiones:

En conclusión, esta práctica nos dio pesadillas, nos planteamos si valía la pena seguir en la carrera, además descubrimos el poco apoyo por parte de nuestros compañeros, ahora vemos que cuando estas más rodeado de personas, es cuando estas más solo.

Referencias:

colaboradores de Wikipedia. (2024, October 14). *Unidad aritmética lógica*. Wikipedia, La Enciclopedia Libre.

https://es.wikipedia.org/wiki/Unidad_aritm%C3%A9tica_l%C3%B3gica

Link del github: https://github.com/lomtimothy/Jericalla-evolucion/tree/main

Segunda parte del proyecto:

Esta segunda parte consiste en añadir una gui codificada en Python, esta gui hará lo siguiente:

1.- En la interfaz se nos muestran 3 opciones, cargas ASM, convertir a binario, y guardar binario, además se nos muestran dos ventanas, una que dice contenido asm y contenido binario.

2.- Antes de usar la interfaz, lo primero que debemos de hacer es escribir en un archivo de código asm todas las instrucciones en ensamblador, justo como en la tabla antes mostrada.

3.- Una vez que ya tenemos el archivo, abrimos la interfaz y seleccionamos la opción de cargar asm, y después seleccionaremos nuestro archivo asm con las instrucciones, y veremos que las instrucciones se cargaran en la ventana de contenido asm.

4.- Ahora seleccionamos la opción de convertir a binario, y podremos ver que como se convierten las instrucciones en ensamblador a binario en la ventana de contenido binario, igual que en la tabla anteriormente mostrada.

5.- Ahora seleccionaremos la opción de guardar binario, esta función guardara un archivo text con las instrucciones en binario.

Código ASM

```
1   ADD $4 $0 $1
2   SUB $5 $1 $2
3   TERN $6 $2 $3
4   SW $7 $4
5   SW $8 $5
6   SW $9 $6
```

Interfaz hecha en Python (gui):

Archivo text generado por la interfaz:

```
0000100000000001
0100101000100010
1000110000100011
1111111001100100
1111111010000101
1111111010010110
```

La funcionalidad que tiene este archivo text con las instrucciones en binario generado por la interfaz es modificar el testbench de la jericalla evolution anteriormente mostrado, para que, en lugar de dar las instrucciones nosotros, podamos cargar el archivo asm y en las instrucciones se pongan automáticamente.

Para lograr esto, primero debemos modificar el testbench que ya teníamos hecho, la modificación quedo de la siguiente manera:

```
// Testbench
module Jericalla_tb();

// Parámetro: número de instrucciones en el archivo "datos.txt"
 parameter NUM_INSTR = 6;

reg [16:0] instruccion;
reg clk;
wire [31:0] salida;

// Memoria para almacenar las instrucciones leídas del archivo
  reg [16:0] instr_mem [0:NUM_INSTR-1];

// Instancia del módulo Jericalla
Jericalla JTestBench (.instruccion(instruccion),.clk(clk),.salida(salida));

 integer i;

// Reloj
initial begin
    clk = 0;
    forever #100 clk = ~clk; //Tiempo para que se procesen los datos devidamente
end

initial begin
// Esperar un breve lapso para que se inicialicen otros bloques
    #1000;
    // Cargar el archivo binario generado por Python ("datos.txt") en la memoria instr_mem
    $readmemb("datos.txt", instr_mem);

      // Recorremos las instrucciones una a una
    for (i = 0; i < NUM_INSTR; i = i + 1) begin
      instruccion = instr_mem[i];
      // Esperar tiempo suficiente para que la instrucción se ejecute
      #1000;
    end

    $stop;
  end

endmodule
```

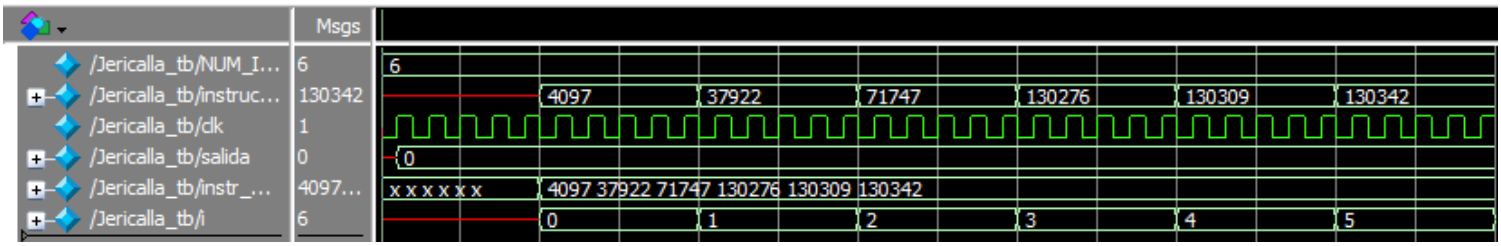
La modificación consiste en lo siguiente:

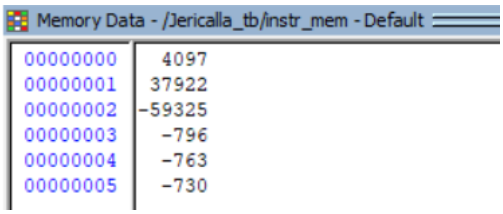
Primero agregamos $readmemb("datos.txt", instr_mem), esta instrucción carga en la memoria instr_mem todas las líneas que están en el archivo que hizo la gui, este archivo lo llamaremos datos.txt.

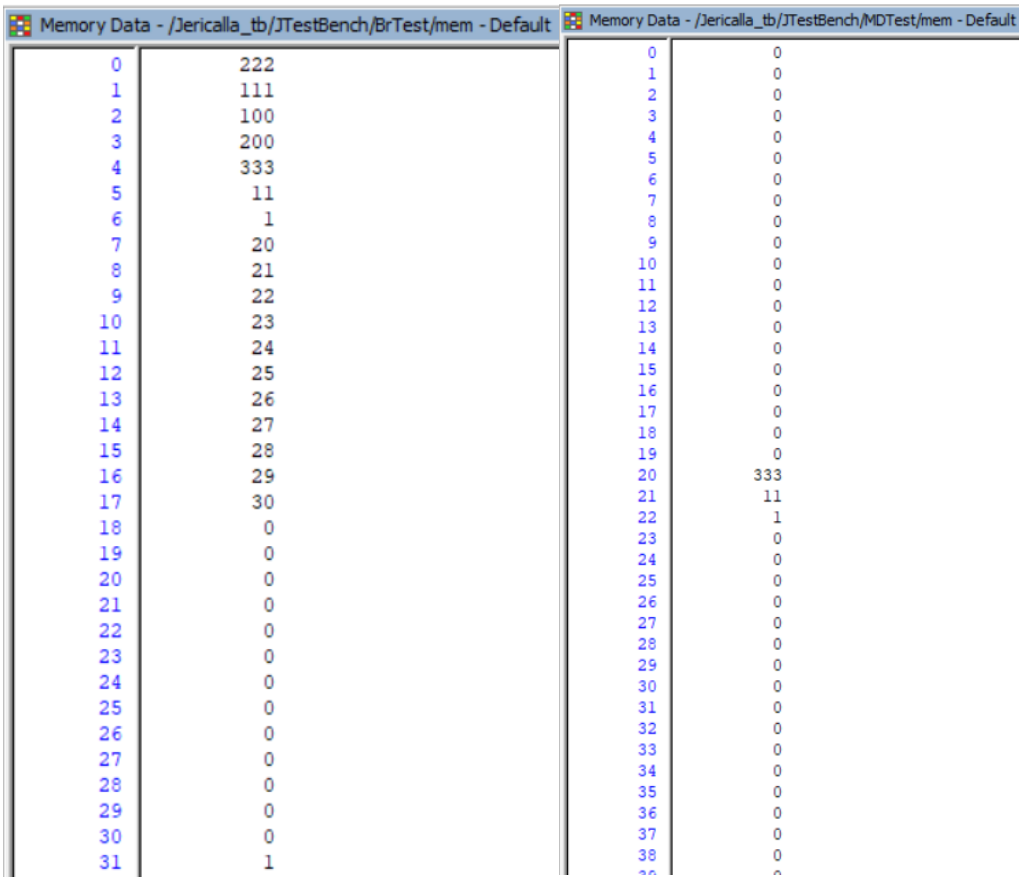
Después con la variable instrucción se asignará en cada iteración con el contenido de la posición actual de instr_mem, de esta forma el módulo jericalla recibe en su entrada instrucción cada una de las instrucciones cargadas desde el archivo.

Ahora añadimos un ciclo for, que básicamente recorre las instrucciones una por una, con un tiempo de espera de 1000, entre cada instrucción para permitir su procesamiento.

Estos son los resultados de la simulación en modelsim con la modificación del testbench y con el archivo txt en binario:







Conclusión: Chat gpt hace buenas interfaces en Python, ahora lo se. Estuvo interesante hacer el ciclo for para las instrucciones, y viva los trabajos en equipo.