

**UNIVERSIDAD DE GUADALAJARA.**

**CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS.**

**MATERIA:**

**S. S. P. DE ARQUITECTURA DE COMPUTADORAS**

**SECCIÓN: D11**

**PROYECTO FINAL FASE 3**

**ALUMNOS:**

NATALIA ISABEL MARISCAL NAPOLES

MARÍN GONZÁLEZ ANDRÉ JOSUÉ

JUAN SILVERO VALENCIA

ERICK JARED GUTIERREZ CORREA

**DOCENTE:**

J. Ernesto López Arce Delgado

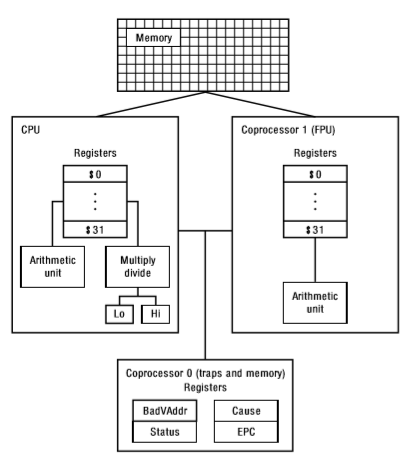
**CARRERA:**

ING. INFORMÁTICA

2023

**INTRODUCCIÓN**.

Con el nombre de MIPS (siglas de Microprocessor without Interlocked Pipeline Stages o dicho en castellano microprocesador sin bloqueos en las etapas de segmentación) se conoce a toda una familia de microprocesadores de arquitectura RISC desarrollados por MIPS Technologies.

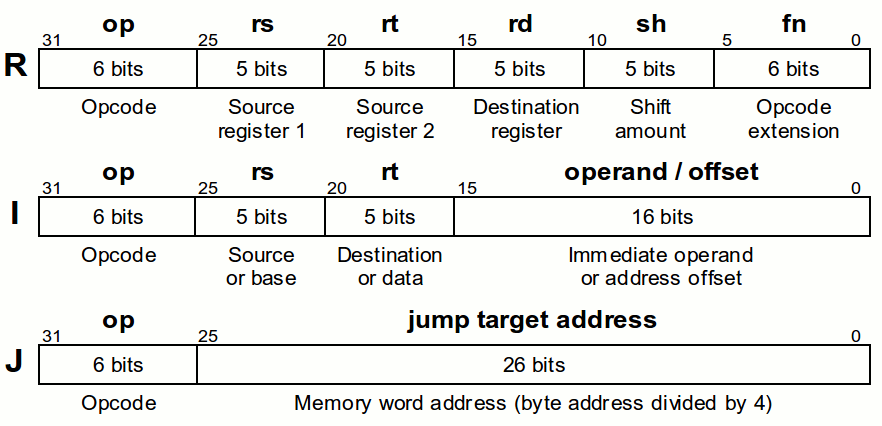


Nos podemos encontrar arquitecturas MIPS en muchos sistemas embebidos, en dispositivos para Windows CE, routers Cisco y consolas como la Nintendo 64 o las PlayStation, PlayStation 2 y PSP.

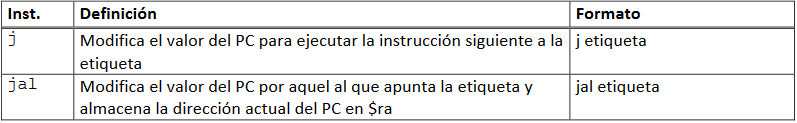
**CARACTERÍSTICAS DE LA ARQUITECTURA DEL MIPS**

* La longitud de todas las instrucciones es fija y son de 32 bits.
* El tamaño de las palabras es siempre de 4 bytes (4\*8 = 32bits)
* Los operandos de las operaciones aritméticas son siempre registros. MIPS es, por tanto, una arquitectura de carga/almacenamiento (registro-registro).
* El acceso a memoria se hace a través de operaciones de carga/almacenamiento (transferencia de datos).
* La mayor parte de las instrucciones que acceden a memoria lo hacen de forma alineada, por lo que la dirección a la que se accede debe ser múltiplo de 4.

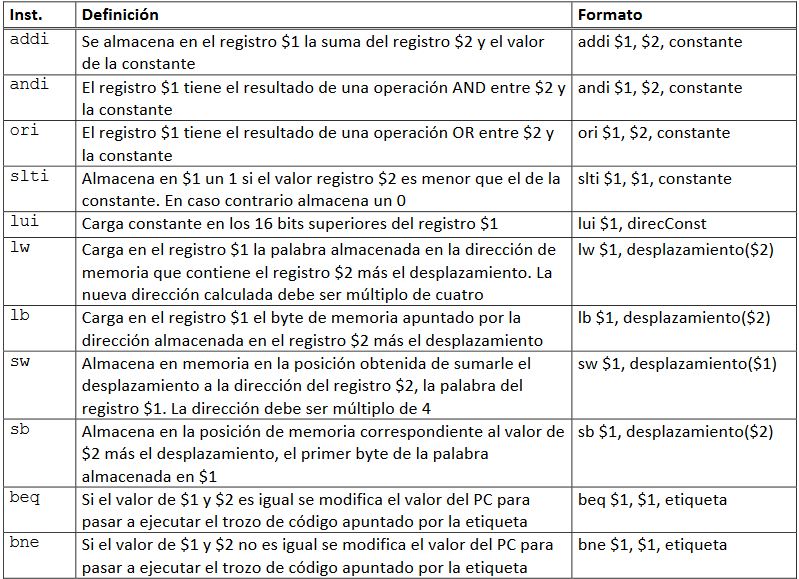
**LAS INSTRUCCIONES EN MIPS LAS PODEMOS ENCONTRAR EN 3 FORMATOS**



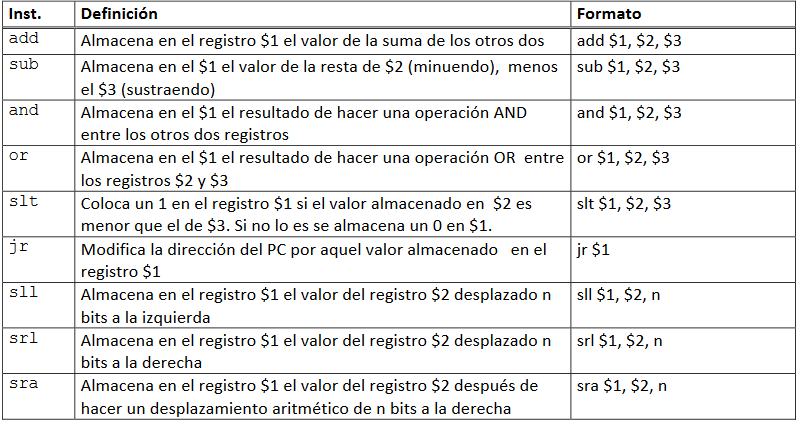
**Instrucciones Tipo J:** utilizado por las instrucciones de bifurcación



**Instrucciones Tipo I:** utilizado por las instrucciones de transferencia, las de salto condicional y las instrucciones con operandos inmediatos.



**Instrucciones tipo R:**utilizado por las instrucciones aritméticas y lógicas.



**Objetivo**

Diseñar un “datapath” con arquitectura tipo MIPS de 32 bits capaz de ejecutar las siguientes instrucciones de las siguientes tablas:

Tabla, Calendario

Descripción generada automáticamente

**Módulos**

**26\_28\_Shift\_Left\_2.v**

module Shift\_Left\_26\_28(

input [25:0]SL\_in,

output reg[27:0]SL\_out

);

always @\*

    begin

       SL\_out = SL\_in << 2;

    end

endmodule

Parámetros del módulo:

* input [25:0] SL\_in: Es una entrada de 26 bits (SL\_in), donde los bits están indexados desde 25 hasta 0. Este rango especifica que SL\_in es un bus de 26 bits.
* output reg [27:0] SL\_out: Es una salida de 28 bits (SL\_out), donde los bits están indexados desde 27 hasta 0. Este rango especifica que SL\_out es un registro (indica reg) de 28 bits.

Bloque always @\*:

* always @\* indica que el bloque de código dentro de él se ejecutará cada vez que cambie alguna de las entradas del módulo.

Bloque de código dentro de always @\*:

* SL\_out = SL\_in << 2;: Esta línea realiza una operación de desplazamiento a la izquierda (<<) en la entrada SL\_in de 2 bits. En términos simples, esto significa que todos los bits en SL\_in se desplazarán dos posiciones hacia la izquierda, y los bits vacíos a la derecha se llenarán con ceros. El resultado se asigna a la salida SL\_out.

Este módulo toma una entrada de 26 bits (SL\_in), realiza un desplazamiento a la izquierda en 2 bits y asigna el resultado a una salida de 28 bits (SL\_out). Este tipo de operación de desplazamiento a menudo se utiliza en arquitecturas como MIPS para realizar operaciones de multiplicación o desplazamientos en instrucciones de máquina.

**ADD.v**

module ADDER(

input [31:0] O1,

input [31:0] O2,

output reg [31:0] Res);

always@\*

begin

    Res = O1 + O2;

end

endmodule

Parámetros del módulo:

* input [31:0] O1: Es una entrada de 32 bits (O1), donde los bits están indexados desde 31 hasta 0. Este rango especifica que O1 es un bus de 32 bits.
* input [31:0] O2: Es otra entrada de 32 bits (O2), con la misma configuración que O1.
* output reg [31:0] Res: Es una salida de 32 bits (Res), donde los bits están indexados desde 31 hasta 0. Este rango especifica que Res es un registro (indica reg) de 32 bits.

Bloque always @\*:

* always @\* indica que el bloque de código dentro de él se ejecutará cada vez que cambie alguna de las entradas del módulo.

Bloque de código dentro de always @\*:

* Res = O1 + O2;: Esta línea realiza una operación de suma (+) entre las entradas O1 y O2. El resultado se asigna a la salida Res.

Este módulo toma dos entradas de 32 bits (O1 y O2), realiza una operación de suma y asigna el resultado a una salida de 32 bits (Res).

**Buffer 1\_IFID.v**

//BUFFER 1: IF/ID

module IFID(

input clk,

input [31:0]Next\_address,

input [31:0]Instruction,

output reg [31:0]O\_Next\_address,

output reg [31:0]O\_Instruction);

always @(posedge clk)

begin

    O\_Next\_address = Next\_address;

        O\_Instruction = Instruction;

end

endmodule

Parámetros del módulo:

* input clk: Es una entrada que representa la señal de reloj.
* input [31:0] Next\_address: Es una entrada de 32 bits que representa la dirección de la siguiente instrucción en el pipeline.
* input [31:0] Instruction: Es otra entrada de 32 bits que representa la instrucción actual en el pipeline.
* output reg [31:0] O\_Next\_address: Es una salida de 32 bits que retiene la dirección de la siguiente instrucción.
* output reg [31:0] O\_Instruction: Es otra salida de 32 bits que retiene la instrucción actual.

Bloque always @(posedge clk):

* always @(posedge clk) indica que el bloque de código dentro de él se ejecutará en el flanco de subida del reloj.

Bloque de código dentro de always @(posedge clk):

* O\_Next\_address = Next\_address;: En cada flanco de subida del reloj, la salida O\_Next\_address se actualiza con el valor presente en la entrada Next\_address.
* O\_Instruction = Instruction;: De manera similar, la salida O\_Instruction se actualiza con el valor presente en la entrada Instruction en cada flanco de subida del reloj.

Este módulo actúa como un registro que retiene la dirección de la siguiente instrucción (O\_Next\_address) y la instrucción actual (O\_Instruction) en el pipeline, actualizándolas en cada ciclo de reloj. Este tipo de registros son típicos en el diseño de procesadores para mantener el estado de las etapas del pipeline.

**Buffer 2\_IDEX.v**

//BUFFER 2: ID/EX

module IDEX(

input clk,

input [1:0]I\_WB,

input [2:0]I\_M,

input [4:0]I\_EX,

input [31:0]I\_Next\_address,

input [31:0]I\_O1,

input [31:0]I\_O2,

input [31:0]I\_Ext\_Inmed,

input [4:0]I\_RT,

input [4:0]I\_RD,

output reg [1:0]O\_WB,

output reg [2:0]O\_M,

output reg O\_EX\_RegDst,

output reg [2:0]O\_EX\_ALUOp,

output reg O\_EX\_ALUSrc,

output reg [31:0]O\_Next\_address,

output reg [31:0]O\_O1,

output reg [31:0]O\_O2,

output reg [31:0]O\_Ext\_Inmed,

output reg [4:0]O\_RT,

output reg [4:0]O\_RD);

always @(posedge clk)

begin

    O\_WB=I\_WB;

    O\_M=I\_M;

    O\_EX\_RegDst=I\_EX[0];

    O\_EX\_ALUOp=I\_EX[3:1];

    O\_EX\_ALUSrc=I\_EX[4];

    O\_Next\_address=I\_Next\_address;

    O\_O1=I\_O1;

    O\_O2=I\_O2;

    O\_Ext\_Inmed=I\_Ext\_Inmed;

    O\_RT=I\_RT;

    O\_RD=I\_RD;

end

endmodule

Parámetros del módulo:

* input clk: Es una entrada que representa la señal de reloj.
* input [1:0] I\_WB: Es una entrada de 2 bits que parece estar relacionada con la escritura de datos de vuelta (write back).
* input [2:0] I\_M: Es una entrada de 3 bits, posiblemente relacionada con el manejo de memoria.
* input [4:0] I\_EX: Es una entrada de 5 bits que parece estar relacionada con la etapa de ejecución.
* input [31:0] I\_Next\_address: Es una entrada de 32 bits que representa la dirección de la siguiente instrucción en el pipeline.
* input [31:0] I\_O1: Es una entrada de 32 bits.
* input [31:0] I\_O2: Es otra entrada de 32 bits.
* input [31:0] I\_Ext\_Inmed: Es otra entrada de 32 bits, posiblemente una extensión inmediata.
* input [4:0] I\_RT: Es una entrada de 5 bits.
* input [4:0] I\_RD: Es otra entrada de 5 bits.
* output reg [1:0] O\_WB: Es una salida de 2 bits.
* output reg [2:0] O\_M: Es una salida de 3 bits.
* output reg O\_EX\_RegDst: Es una salida de 1 bit.
* output reg [2:0] O\_EX\_ALUOp: Es una salida de 3 bits relacionada con la operación de la ALU.
* output reg O\_EX\_ALUSrc: Es una salida de 1 bit relacionada con la fuente de datos de la ALU.
* output reg [31:0] O\_Next\_address: Es una salida de 32 bits que retiene la dirección de la siguiente instrucción.
* output reg [31:0] O\_O1: Es una salida de 32 bits.
* output reg [31:0] O\_O2: Es otra salida de 32 bits.
* output reg [31:0] O\_Ext\_Inmed: Es otra salida de 32 bits.
* output reg [4:0] O\_RT: Es una salida de 5 bits.
* output reg [4:0] O\_RD: Es otra salida de 5 bits.

Bloque always @(posedge clk):

* always @(posedge clk) indica que el bloque de código dentro de él se ejecutará en el flanco de subida del reloj.

Bloque de código dentro de always @(posedge clk):

* Este bloque asigna las entradas a las salidas en cada flanco de subida del reloj, efectivamente registrando los valores presentes en las entradas.

Este módulo IDEX retiene y registra diversos campos relacionados con la etapa de ejecución de un pipeline en un procesador. Las entradas representan información de la etapa de instrucción/decodificación, y las salidas retienen y proporcionan esta información para la etapa de ejecución siguiente.

**Buffer 3\_EXMEM.v**

//BUFFER 3: EX/MEM

module EXMEM(

input clk,

input [1:0]I\_WB,

input [2:0]I\_M,

input [31:0]I\_ADD\_Res,

input I\_ZF,

input [31:0]I\_ALU\_Res,

input [31:0]I\_DatWri\_Mem,

input [4:0]I\_Addr\_Reg\_Wri,

output reg [1:0]O\_WB,

output reg O\_M\_Branch,

output reg O\_M\_MemRead,

output reg O\_M\_MemWrite,

output reg [31:0]O\_ADD\_Res,

output reg O\_ZF,

output reg [31:0]O\_ALU\_Res,

output reg [31:0]O\_DatWri\_Mem,

output reg [4:0]O\_Addr\_Reg\_Wri);

always @(posedge clk)

begin

    O\_WB=I\_WB;

    O\_M\_Branch=I\_M[0];

    O\_M\_MemRead=I\_M[1];

    O\_M\_MemWrite=I\_M[2];

    O\_ADD\_Res=I\_ADD\_Res;

    O\_ZF=I\_ZF;

    O\_ALU\_Res=I\_ALU\_Res;

    O\_DatWri\_Mem=I\_DatWri\_Mem;

    O\_Addr\_Reg\_Wri=I\_Addr\_Reg\_Wri;

end

endmodule

Parámetros del módulo:

* input clk: Es una entrada que representa la señal de reloj.
* input [1:0] I\_WB: Es una entrada de 2 bits que parece estar relacionada con la escritura de datos de vuelta (write back).
* input [2:0] I\_M: Es una entrada de 3 bits que posiblemente esté relacionada con operaciones de memoria (memoria de lectura/escritura, operaciones de branch).
* input [31:0] I\_ADD\_Res: Es una entrada de 32 bits que representa el resultado de una operación de suma realizada en la etapa de ejecución.
* input I\_ZF: Es una entrada de 1 bit, posiblemente relacionada con la bandera de cero (zero flag).
* input [31:0] I\_ALU\_Res: Es una entrada de 32 bits que representa el resultado de una operación de la Unidad Lógico-Aritmética (ALU).
* input [31:0] I\_DatWri\_Mem: Es una entrada de 32 bits que representa los datos que se escribirán en memoria.
* input [4:0] I\_Addr\_Reg\_Wri: Es una entrada de 5 bits que representa la dirección del registro que se actualizará en la etapa de escritura de vuelta.
* output reg [1:0] O\_WB: Es una salida de 2 bits.
* output reg O\_M\_Branch: Es una salida de 1 bit que indica si se realizará una operación de branch en la etapa de memoria.
* output reg O\_M\_MemRead: Es una salida de 1 bit que indica si se realizará una operación de lectura de memoria en la etapa de memoria.
* output reg O\_M\_MemWrite: Es una salida de 1 bit que indica si se realizará una operación de escritura de memoria en la etapa de memoria.
* output reg [31:0] O\_ADD\_Res: Es una salida de 32 bits que retiene el resultado de la operación de suma.
* output reg O\_ZF: Es una salida de 1 bit que retiene el estado de la bandera de cero.
* output reg [31:0] O\_ALU\_Res: Es una salida de 32 bits que retiene el resultado de la operación de la ALU.
* output reg [31:0] O\_DatWri\_Mem: Es una salida de 32 bits que retiene los datos que se escribirán en memoria.
* output reg [4:0] O\_Addr\_Reg\_Wri: Es una salida de 5 bits que retiene la dirección del registro que se actualizará en la etapa de escritura de vuelta.

Bloque always @(posedge clk):

* always @(posedge clk) indica que el bloque de código dentro de él se ejecutará en el flanco de subida del reloj.

Bloque de código dentro de always @(posedge clk):

* Este bloque asigna las entradas a las salidas en cada flanco de subida del reloj, efectivamente registrando los valores presentes en las entradas.

Este módulo EXMEM retiene y registra diversos campos relacionados con la etapa de memoria de un pipeline en un procesador. Las entradas representan información de la etapa de ejecución, y las salidas retienen y proporcionan esta información para la etapa de escritura de vuelta.

**Buffer 4\_MEMWB.v**

//BUFFER 4: MEM/WB

module MEMWB(

input clk,

input [1:0]I\_WB,

input [31:0]I\_ReDat\_Mem,

input [31:0]I\_ALU\_Res,

input [4:0]I\_Addr\_Reg\_Wri,

output reg O\_WB\_RegWrite,

output reg O\_WB\_MemtoReg,

output reg [31:0]O\_ReDat\_Mem,

output reg [31:0]O\_ALU\_Res,

output reg [4:0]O\_Addr\_Reg\_Wri);

always @(posedge clk)

begin

    O\_WB\_RegWrite=I\_WB[0];

    O\_WB\_MemtoReg=I\_WB[1];

    O\_ReDat\_Mem=I\_ReDat\_Mem;

    O\_ALU\_Res=I\_ALU\_Res;

    O\_Addr\_Reg\_Wri=I\_Addr\_Reg\_Wri;

end

endmodule

Parámetros del módulo:

* input clk: Es una entrada que representa la señal de reloj.
* input [1:0] I\_WB: Es una entrada de 2 bits que parece estar relacionada con la escritura de datos de vuelta (write back).
* input [31:0] I\_ReDat\_Mem: Es una entrada de 32 bits que representa los datos leídos de memoria.
* input [31:0] I\_ALU\_Res: Es una entrada de 32 bits que representa el resultado de una operación de la Unidad Lógico-Aritmética (ALU).
* input [4:0] I\_Addr\_Reg\_Wri: Es una entrada de 5 bits que representa la dirección del registro que se actualizará en la etapa de escritura de vuelta.
* output reg O\_WB\_RegWrite: Es una salida de 1 bit que indica si se realizará una operación de escritura de vuelta.
* output reg O\_WB\_MemtoReg: Es una salida de 1 bit que indica si se realizará una operación de escritura de memoria a registro.
* output reg [31:0] O\_ReDat\_Mem: Es una salida de 32 bits que retiene los datos leídos de memoria.
* output reg [31:0] O\_ALU\_Res: Es una salida de 32 bits que retiene el resultado de la operación de la ALU.
* output reg [4:0] O\_Addr\_Reg\_Wri: Es una salida de 5 bits que retiene la dirección del registro que se actualizará en la etapa de escritura de vuelta.

Bloque always @(posedge clk):

always @(posedge clk) indica que el bloque de código dentro de él se ejecutará en el flanco de subida del reloj.

Bloque de código dentro de always @(posedge clk):

Este bloque asigna las entradas a las salidas en cada flanco de subida del reloj, efectivamente registrando los valores presentes en las entradas.

Este módulo MEMWB retiene y registra diversos campos relacionados con la etapa de escritura de vuelta de un pipeline en un procesador. Las entradas representan información de la etapa de memoria, y las salidas retienen y proporcionan esta información para la etapa final de escritura de vuelta.

**Fase 2\_tb.v**

`timescale 1ns/1ns

module Fase2\_tb();

reg clk\_tb;

MyDatapath \_Datapath(.clk(clk\_tb));

always #100 clk\_tb = ~clk\_tb;

initial

begin

    clk\_tb = 1'b0;

    #20000

    $stop;

end

endmodule

Parámetros del módulo:

* reg clk\_tb;: Declara una señal de reloj llamada clk\_tb que será utilizada para clockear el módulo MyDatapath.

Instancia del módulo MyDatapath:

* MyDatapath \_Datapath(.clk(clk\_tb));: Instancia el módulo MyDatapath y conecta la señal de reloj clk\_tb a su entrada de reloj (clk).

Generación de la señal de reloj:

* always #100 clk\_tb = ~clk\_tb;: Genera una señal de reloj (clk\_tb) que cambia su valor cada 100 unidades de tiempo. Esto simula un reloj con un período de 200 unidades de tiempo.

Bloque inicial:

* initial begin ... end: Este bloque se ejecuta al inicio de la simulación.
* clk\_tb = 1'b0;: Inicializa la señal de reloj en 0.
* #20000 $stop;: La simulación se detendrá después de 20000 unidades de tiempo.

Este banco de pruebas (Fase2\_tb) configura una simulación con un reloj (clk\_tb) que cambia su valor cada 100 unidades de tiempo, instanciando el módulo MyDatapath y conectándolo a esta señal de reloj. La simulación se detendrá después de 20000 unidades de tiempo.

**MyADD\_4.v**

module MyADD\_4(

    input [31:0] operand,

    output reg [31:0] result

);

always @\*

begin

    result = operand + 4;

end

endmodule

Parámetros del módulo:

* input [31:0] operand: Es una entrada de 32 bits llamada operand.
* output reg [31:0] result: Es una salida de 32 bits llamada result y se declara como un registro (reg).

Bloque always @\*:

* always @\* indica que el bloque de código dentro de él se ejecutará siempre que haya un cambio en las entradas.

Bloque de código dentro de always @\*:

* result = operand + 4;: Esta línea realiza una operación de suma entre la entrada operand y la constante 4. El resultado se asigna a la salida result.

Este módulo MyADD\_4 toma un operando de 32 bits, le suma la constante 4 y devuelve el resultado en la salida result. Este tipo de módulos pueden ser útiles en diversas aplicaciones, como en el diseño de operaciones aritméticas específicas.

**MyALU.v**

module MyALU(

    input [31:0] operand1,

    input [31:0] operand2,

    input [3:0] alu\_select,

    output reg zero\_flag,

    output reg [31:0] result

);

always @\*

begin

    case (alu\_select)

        4'b0000:

        begin

            result = operand1 & operand2; // AND

        end

        4'b0001:

        begin

            result = operand1 | operand2; // OR

        end

        4'b0010:

        begin

            result = operand1 + operand2; //ADD

        end

        4'b0110:

        begin

            result = operand1 - operand2; //SUB

        end

        4'b0111:

        begin

            result = operand1 < operand2?1:0; // SLT

        end

        default:

        begin

            result=32'd0;

        end

    endcase

    zero\_flag = result == 0;

end

endmodule

Parámetros del módulo:

* input [31:0] operand1: Es una entrada de 32 bits llamada operand1.
* input [31:0] operand2: Es otra entrada de 32 bits llamada operand2.
* input [3:0] alu\_select: Es una entrada de 4 bits que selecciona la operación a realizar en la ALU.
* output reg zero\_flag: Es una salida de 1 bit que representa la bandera de cero.
* output reg [31:0] result: Es una salida de 32 bits llamada result y se declara como un registro (reg).

Bloque always @\*:

* always @\* indica que el bloque de código dentro de él se ejecutará siempre que haya un cambio en las entradas.

Bloque de código dentro de always @\*:

* case (alu\_select): Este bloque de código selecciona la operación de la ALU según el valor de alu\_select.
* Cada begin y end dentro de los case contienen la lógica para una operación específica.
* zero\_flag = (result == 32'd0);: Se establece la bandera de cero en 1 si el resultado es igual a cero, de lo contrario, se establece en 0.

Este módulo MyALU implementa una ALU que realiza operaciones lógicas (AND, OR), aritméticas (ADD, SUB), y una operación de comparación (SLT). La bandera de cero (zero\_flag) se establece según si el resultado de la operación es igual a cero. Este tipo de módulos son esenciales en el diseño de procesadores para realizar diversas operaciones.

**MyALUControl.v**

module MyALUControl(

    input [2:0] alu\_operation,

    input [5:0] function\_code,

    output reg [3:0] alu\_select

);

always @\*

begin

    case (alu\_operation)

    3'b010:

    begin

        case (function\_code)

            6'b100100:

                alu\_select = 4'b0000; // AND

            6'b100101:

                alu\_select = 4'b0001; // OR

            6'b100000:

                alu\_select = 4'b0010; // ADD

            6'b100010:

                alu\_select = 4'b0110; // SUBTRACT

            6'b101010:

                alu\_select = 4'b0111; // SET ON LESS THAN (SLT)

            6'b000000:

                alu\_select = 4'b0000; // Soporte a la operacion Nop

        endcase

    end

    3'b000:

    begin

    alu\_select = 4'b0010; // ADD

    end

    3'b001:

    begin

    alu\_select = 4'b0110; // SUBTRACT

    end

    3'b011:

    begin

    alu\_select = 4'b0000; // AND

    end

    3'b100:

    begin

    alu\_select = 4'b0001; // OR

    end

    3'b101:

    begin

    alu\_select = 4'b0111; // SLT

    end

    endcase

end

endmodule

Parámetros del módulo:

* input [2:0] alu\_operation: Es una entrada de 3 bits que representa la operación de la ALU.
* input [5:0] function\_code: Es una entrada de 6 bits que representa el código de función.
* output reg [3:0] alu\_select: Es una salida de 4 bits que representa la selección de operación de la ALU.

Bloque always @\*:

* always @\* indica que el bloque de código dentro de él se ejecutará siempre que haya un cambio en las entradas.

Bloque de código dentro de always @\*:

* case (alu\_operation): Este bloque de código selecciona la operación de la ALU basándose en el valor de alu\_operation.
* Dentro de cada caso (begin y end), hay un segundo case que selecciona la operación de la ALU basándose en el valor de function\_code.
* Dependiendo de los valores de alu\_operation y function\_code, se asigna la señal alu\_select con la operación correspondiente.

Este módulo MyALUControl toma las señales de control alu\_operation y function\_code como entradas y genera la señal de selección de operación de la ALU (alu\_select) de acuerdo con la operación deseada.

**MyBR.v**

module MyBR(

    input [4:0] address\_reg1,

    input [4:0] address\_reg2,

    input [4:0] address\_write,

    input [31:0] data\_write,

    input enable, //Habilitar la escritura

    output reg [31:0] data\_read1,

    output reg [31:0] data\_read2

);

reg [31:0] memory[0:31];

/\*initial

begin

    memory[0] = 32'd0;

    memory[1] = 32'd100;

    memory[2] = 32'd120;

    memory[3] = 32'b10101;

    memory[4] = 32'b01010;

    memory[31] = 32'd6;

end\*/

always @\*

begin

    if (enable && data\_write !== 32'bx)

    begin

        memory[address\_write] = data\_write; // escribe en la direcci�n address

    end

    data\_read1 = memory[address\_reg1]; // lee la memoria en la direcci�n address

    data\_read2 = memory[address\_reg2];

end

endmodule

Parámetros del módulo:

* input [4:0] address\_reg1: Es una entrada de 5 bits que representa la dirección de memoria desde la cual leer data\_read1.
* input [4:0] address\_reg2: Es otra entrada de 5 bits que representa la dirección de memoria desde la cual leer data\_read2.
* input [4:0] address\_write: Es una entrada de 5 bits que representa la dirección de memoria en la cual escribir data\_write.
* input [31:0] data\_write: Es una entrada de 32 bits que representa los datos a escribir en memoria.
* input enable: Es una entrada que habilita la escritura en memoria.
* output reg [31:0] data\_read1: Es una salida de 32 bits que retiene los datos leídos desde la dirección address\_reg1.
* output reg [31:0] data\_read2: Es otra salida de 32 bits que retiene los datos leídos desde la dirección address\_reg2.

Arreglo de memoria:

* reg [31:0] memory[0:31];: Declara un arreglo de memoria llamado memory que puede almacenar 32 palabras de 32 bits.

Bloque always @\*:

* always @\* indica que el bloque de código dentro de él se ejecutará siempre que haya un cambio en las entradas.

Bloque de código dentro de always @\*:

* if (enable && data\_write !== 32'bx): Verifica si la habilitación está activa y si los datos de escritura son distintos de "x" (no definidos).
* memory[address\_write] = data\_write;: Si se cumple la condición anterior, se escribe data\_write en la dirección de memoria especificada por address\_write.
* data\_read1 = memory[address\_reg1];: Lee la memoria en la dirección especificada por address\_reg1 y retiene los datos en data\_read1.
* data\_read2 = memory[address\_reg2];: Lee la memoria en la dirección especificada por address\_reg2 y retiene los datos en data\_read2.

Este módulo MyBR implementa una memoria simple que puede leer de dos direcciones de memoria y escribir en una dirección de memoria específica cuando la habilitación está activa. Los datos se retienen en data\_read1 y data\_read2. La memoria puede ser inicializada descomentando la sección initial que está comentada en el código.

**MyControlUnit.v**

module MyControlUnit(

    input [31:26] opcode,

    output reg reg\_Dst,

    output reg branch,

    output reg memory\_read,

    output reg memory\_register,

    output reg [2:0] alu\_operation,

    output reg memory\_write,

    output reg ALU\_Src,

    output reg reg\_write,

    output reg jump

);

always @\*

begin

    case (opcode)

        6'b000000://Instrucciones Tipo R

        begin

            reg\_Dst=1'b1;

            branch=1'b0;

            memory\_read = 1'b0;

            memory\_register = 1'b0;

            alu\_operation = 3'b010;

        memory\_write = 1'b0;

            ALU\_Src=1'b0;

            reg\_write = 1'b1;

        jump=1'b0;

        end

        6'b001000://ADDI

        begin

            reg\_Dst=1'b0;

            branch=1'b0;

            memory\_read = 1'b0;

            memory\_register = 1'b0;

            alu\_operation = 3'b000;

        memory\_write = 1'b0;

            ALU\_Src=1'b1;

            reg\_write = 1'b1;

        jump=1'b0;

        end

        6'b001100://ANDI

        begin

            reg\_Dst=1'b0;

            branch=1'b0;

            memory\_read = 1'b0;

            memory\_register = 1'b0;

            alu\_operation = 3'b011;//�C�mo hay que ponerlo?

        memory\_write = 1'b0;

            ALU\_Src=1'b1;

            reg\_write = 1'b1;

        jump=1'b0;

        end

        6'b001101://ORI

        begin

            reg\_Dst=1'b0;

            branch=1'b0;

            memory\_read = 1'b0;

            memory\_register = 1'b0;

            alu\_operation = 3'b100;//�C�mo hay que ponerlo?

        memory\_write = 1'b0;

            ALU\_Src=1'b1;

            reg\_write = 1'b1;

            jump=1'b0;

        end

        /\*6'b000000://SUBI??

        begin

            reg\_Dst=1'b0;

            branch=1'b0;

            memory\_read = 1'b0;

            memory\_register = 1'b0;

            alu\_operation = 3'b001;

        memory\_write = 1'b0;

            ALU\_Src=1'b1;

            reg\_write = 1'b1;

            jump=1'b0;

        end\*/

        6'b100011://LW

        begin

            reg\_Dst=1'b0;

            branch=1'b0;

            memory\_read = 1'b1;

            memory\_register = 1'b1;

            alu\_operation = 3'b000;

        memory\_write = 1'b0;

            ALU\_Src=1'b1;

            reg\_write = 1'b1;

            jump=1'b0;

        end

        6'b101011://SW

        begin

            reg\_Dst=1'b0;

            branch=1'b0;

            memory\_read = 1'b0;

            memory\_register = 1'b0;

            alu\_operation = 3'b000;

        memory\_write = 1'b1;

            ALU\_Src=1'b1;

            reg\_write = 1'b0;

            jump=1'b0;

        end

        6'b001010://SLTI

        begin

            reg\_Dst=1'b0;

            branch=1'b0;

            memory\_read = 1'b0;

            memory\_register = 1'b0;

            alu\_operation = 3'b101;//�C�mo hay que ponerlo?

        memory\_write = 1'b0;

            ALU\_Src=1'b1;

            reg\_write = 1'b1;

            jump=1'b0;

        end

        6'b000100://BEQ

        begin

            reg\_Dst=1'b0;

            branch=1'b1;

            memory\_read = 1'b0;

            memory\_register = 1'b0;

            alu\_operation = 3'b001;//�Est� bien?

        memory\_write = 1'b0;

            ALU\_Src=1'b0;

            reg\_write = 1'b0;

            jump=1'b0;

        end

        /\*6'b000101://BNE

        begin

            reg\_Dst=1'b0;

            branch=1'b1;

            memory\_read = 1'b0;

            memory\_register = 1'b0;

            alu\_operation = 3'b001;//�C�mo implementar esta operaci�n?

        memory\_write = 1'b0;

            ALU\_Src=1'b0;

            reg\_write = 1'b0;

            jump=1'b0;

        end

        6'b000111://BGTZ

        begin

            reg\_Dst=1'b0;

            branch=1'b1;

            memory\_read = 1'b0;

            memory\_register = 1'b0;

            alu\_operation = 3'b101; //Esta operaci�n falla cuando se elige un 0, porque 0>0 es falso y la manera en que implement� con SLT hace que se ejecut� el branch

        memory\_write = 1'b0;

            ALU\_Src=1'b0;

            reg\_write = 1'b0;

            jump=1'b0;

        end\*/

        6'b000010://Instrucci�n J

    begin

            reg\_Dst=1'b0;

            branch=1'b0;

            memory\_read = 1'b0;

            memory\_register = 1'b0;

            alu\_operation = 3'b000;//MODIFICAR

        memory\_write = 1'b0;

            ALU\_Src=1'b0;

            reg\_write = 1'b0;

        jump=1'b1;

    end

    endcase

end

endmodule

Parámetros del módulo:

* input [31:26] opcode: Es una entrada de 6 bits que representa el código de operación de la instrucción.
* output reg reg\_Dst: Salida que indica si la instrucción es de tipo R.
* output reg branch: Salida que indica si la instrucción es de salto condicional.
* output reg memory\_read: Salida que indica si la instrucción realiza una lectura de memoria.
* output reg memory\_register: Salida que indica si la instrucción involucra operaciones de memoria y registros.
* output reg [2:0] alu\_operation: Salida que especifica la operación de la ALU.
* output reg memory\_write: Salida que indica si la instrucción realiza una escritura en memoria.
* output reg ALU\_Src: Salida que indica si la fuente de datos para la ALU es inmediata (1) o un registro (0).
* output reg reg\_write: Salida que indica si la instrucción realiza una escritura en registros.
* output reg jump: Salida que indica si la instrucción es de salto incondicional.

Bloque always @\*:

* always @\* indica que el bloque de código dentro de él se ejecutará siempre que haya un cambio en las entradas.

Bloque de código dentro de always @\*:

* case (opcode): Este bloque de código selecciona las señales de control basándose en el código de operación (opcode).
* Dentro de cada caso (begin y end), se establecen las señales de control correspondientes según la instrucción especificada por el opcode.

Aborda varias instrucciones comunes en arquitecturas MIPS, configurando las señales de control apropiadas para cada tipo de instrucción. Nota que algunas instrucciones están comentadas y pueden necesitar ser modificadas o implementadas dependiendo de los requerimientos específicos del procesador MIPS que estás diseñando.

**MyDatapath.v**

module MyDatapath(

    input clk

);

wire [31:0] instruction;

wire control\_memreg;

wire control\_memwrite;

wire control\_memread;

wire control\_regwrite;

wire [2:0] control\_aluop;

wire [31:0] data\_write;

wire [31:0] data\_read1;

wire [31:0] data\_read2;

wire [3:0] control\_alusel;

wire [31:0] alu\_result;

wire [31:0] address\_result;

wire [31:0] data\_mem;

wire RegDst;

wire [4:0] Dir\_Wri\_BR;

wire ALU\_Src;

wire [31:0] Oper\_2;

wire [31:0] Mux\_ALU\_in2;

wire [31:0] pc\_out;

wire [31:0] add\_result;

wire [31:0] SL2\_adder;

wire [31:0] Add\_In2\_MuxPC;

wire Branch;

wire tr\_zf;

wire [31:0] MuxPC\_PCin;

wire Jump;

wire [27:0]JAddress;

wire [31:0]MPC\_MJ;

MyPC \_PC(.clk(clk),.in(MuxPC\_PCin),.out(pc\_out));

MyADD\_4 \_ADD(.operand(pc\_out),.result(add\_result));

MyIMem \_IM(.address(pc\_out),.data(instruction));

MyBR \_BR(.address\_reg1(instruction[25:21]), .address\_reg2(instruction[20:16]), .address\_write(Dir\_Wri\_BR), .data\_write(data\_write), .enable(control\_regwrite), .data\_read1(data\_read1), .data\_read2(data\_read2));

MyControlUnit \_ControlUnit(.opcode(instruction[31:26]), .reg\_Dst(RegDst), .branch(Branch), .memory\_read(control\_memread), .memory\_register(control\_memreg), .alu\_operation(control\_aluop), .memory\_write(control\_memwrite), .ALU\_Src(ALU\_Src), .reg\_write(control\_regwrite), .jump(Jump));

MyALUControl \_ALUControl(.alu\_operation(control\_aluop), .function\_code(instruction[5:0]), .alu\_select(control\_alusel));

MyALU \_ALU(.operand1(data\_read1), .operand2(Oper\_2), .alu\_select(control\_alusel), .zero\_flag(tr\_zf) ,.result(alu\_result));

MyMem \_Mem(.address(alu\_result), .data\_write(data\_read2), .write\_enable(control\_memwrite), .read\_enable(control\_memread), .data\_read(data\_mem));

MyDatapathMux \_MuxWriteBR(.control\_signal(control\_memreg),.input\_data\_1(alu\_result),.input\_data\_2(data\_mem), .output\_data(data\_write));

MyDatapathMux\_5B \_Mux\_Dir\_BR(.control\_signal(RegDst), .input\_data\_1(instruction[20:16]), .input\_data\_2(instruction[15:11]), .output\_data(Dir\_Wri\_BR));

MyDatapathMux \_Mux\_D2\_ALU(.control\_signal(ALU\_Src), .input\_data\_1(data\_read2), .input\_data\_2(Mux\_ALU\_in2), .output\_data(Oper\_2));

sign\_extend \_Sig\_Ext(.Data\_in(instruction[15:0]), .Data\_out(Mux\_ALU\_in2));

Shift\_Left\_2 \_Sh\_Lef\_2(.SL\_in(Mux\_ALU\_in2), .SL\_out(SL2\_adder));

ADDER \_Sumador(.O1(add\_result), .O2(SL2\_adder), .Res(Add\_In2\_MuxPC));

MyDatapathMux \_Mux\_PC(.control\_signal(Branch&tr\_zf), .input\_data\_1(add\_result), .input\_data\_2(Add\_In2\_MuxPC), .output\_data(MPC\_MJ));

MyDatapathMux \_Mux\_J(.control\_signal(Jump), .input\_data\_1(MPC\_MJ), .input\_data\_2(    {add\_result[31:28],JAddress}   ), .output\_data(MuxPC\_PCin));

Shift\_Left\_26\_28 \_SL\_26\_28(.SL\_in(instruction[25:0]), .SL\_out(JAddress));

initial

begin

$readmemb("Inicializaci�n Memoria de Instrucciones.txt",\_IM.memory);

$readmemb("Inicializaci�n BR.txt",\_BR.memory);

$readmemb("Inicializaci�n Memoria de Datos.txt",\_Mem.memory);

end

endmodule

Datapath básico de un procesador MIPS, donde diferentes módulos están interconectados para realizar operaciones específicas, como la ejecución de instrucciones, el control de la unidad, el acceso a memoria, etc. Las memorias de instrucciones y datos están inicializadas a partir de archivos de memoria. La simulación de este diseño en un entorno de simulación de Verilog permitirá observar el comportamiento del datapath y las interacciones entre los diferentes módulos durante la ejecución de programas MIPS.

**MyDatapathMux.v**

module MyDatapathMux(

    input control\_signal,

    input [31:0] input\_data\_1, //0

    input [31:0] input\_data\_2, //1

    output reg [31:0] output\_data

);

always @\*

begin

    if(control\_signal)

    begin

        output\_data = input\_data\_2;

    end

    else

    begin

        output\_data = input\_data\_1;

    end

end

endmodule

Parámetros del módulo:

* input control\_signal: Es una señal de control que determina qué entrada se selecciona.
* input [31:0] input\_data\_1: Es la primera entrada del multiplexor.
* input [31:0] input\_data\_2: Es la segunda entrada del multiplexor.
* output reg [31:0] output\_data: Es la salida del multiplexor.

Bloque always @\*:

* always @\* indica que el bloque de código dentro de él se ejecutará siempre que haya un cambio en las entradas.

Bloque de código dentro de always @\*:

* El bloque de código utiliza una estructura condicional (if-else) para seleccionar la salida del multiplexor en función de la señal de control.
* Si control\_signal es verdadero (1), selecciona input\_data\_2.
* Si control\_signal es falso (0), selecciona input\_data\_1.

Este módulo permite seleccionar entre dos conjuntos de datos en función de la señal de control proporcionada. Es fundamental en el diseño del datapath para enrutar datos según las necesidades del procesador.

**MyDatapathMux5bits.v**

module MyDatapathMux\_5B(

    input control\_signal,

    input [4:0] input\_data\_1, //0

    input [4:0] input\_data\_2, //1

    output reg [4:0] output\_data

);

always @\*

begin

    if(control\_signal)

    begin

        output\_data = input\_data\_2;

    end

    else

    begin

        output\_data = input\_data\_1;

    end

end

endmodule

Parámetros del módulo:

* input control\_signal: Es una señal de control que determina qué entrada se selecciona.
* input [4:0] input\_data\_1: Es la primera entrada del multiplexor.
* input [4:0] input\_data\_2: Es la segunda entrada del multiplexor.
* output reg [4:0] output\_data: Es la salida del multiplexor.

Bloque always @\*:

* always @\* indica que el bloque de código dentro de él se ejecutará siempre que haya un cambio en las entradas.

Bloque de código dentro de always @\*:

* El bloque de código utiliza una estructura condicional (if-else) para seleccionar la salida del multiplexor en función de la señal de control.
* Si control\_signal es verdadero (1), selecciona input\_data\_2.
* Si control\_signal es falso (0), selecciona input\_data\_1.

Este módulo permite seleccionar entre dos conjuntos de datos de 5 bits en función de la señal de control proporcionada, similar al MyDatapathMux anterior, pero con una entrada y salida de 5 bits.

**MyIMem.v**

module MyIMem(

    input [31:0] address,

    output reg [31:0] data

);

reg [7:0] memory[0:255];

initial

begin

    /\*{memory[0],memory[1],memory[2],memory[3]}       = 32'b0;

    {memory[4],memory[5],memory[6],memory[7]}       = 32'b00000000000000010010100000100000;

    {memory[8],memory[9],memory[10],memory[11]}     = 32'b00000000001000100011000000100000;

    {memory[12],memory[13],memory[14],memory[15]}   = 32'b00000000010000010011100000100010;

    {memory[16],memory[17],memory[18],memory[19]}   = 32'b00000000001000100100000000100010;

    {memory[20],memory[21],memory[22],memory[23]}   = 32'b00000000011001000100100000100100;

    {memory[24],memory[25],memory[26],memory[27]}   = 32'b00000000100000110101000000100101;

    {memory[28],memory[29],memory[30],memory[31]}   = 32'b00000000011010100101100000100100;

    {memory[32],memory[33],memory[34],memory[35]}   = 32'b00000000111010000110000000100101;

    {memory[36],memory[37],memory[38],memory[39]}   = 32'b00000000011001000110100000101010;

    {memory[40],memory[41],memory[42],memory[43]}   = 32'b00000000001000100111000000101010;\*/

end

always @\*

begin

    data = {memory[address], memory[address + 1], memory[address + 2], memory[address + 3]};

end

endmodule

Parámetros del módulo:

* input [31:0] address: Es la dirección de memoria de la que se debe leer.
* output reg [31:0] data: Es la salida que contiene los datos leídos desde la memoria.
* Bloque reg [7:0] memory[0:255];: Declara una memoria de bytes llamada memory con 256 ubicaciones, cada una de 8 bits.

Bloque initial:

* Contiene la inicialización de la memoria con valores comentados.

Bloque always @\*:

* Usa always @\* para que el bloque de código se ejecute siempre que haya un cambio en las entradas.
* Lee la memoria en la dirección especificada por address y concatena los 32 bits correspondientes para formar la salida data.

Este módulo simula la lectura de instrucciones de la memoria según la dirección proporcionada.

**MyMem.v**

module MyMem(

    input [31:0] address,

    input [31:0] data\_write,

    input write\_enable, // es como el enable

    input read\_enable, // es como el enable

    output reg [31:0] data\_read

);

reg [31:0] memory[0:255];

/\*initial

begin

    memory[0]=32'd1;

    memory[1]=32'd2;

    memory[2]=32'd6;

    memory[30]=32'd10;

end\*/

always @\*

begin

    if (write\_enable == 1'b1 & read\_enable == 1'b0)

    begin

        memory[address] = data\_write; // escribe en la direcci�n address

    end

    else if (write\_enable == 1'b0 & read\_enable == 1'b1)

    begin

        data\_read = memory[address]; // lee la memoria en la direcci�n address

    end

end

endmodule

Parámetros del módulo:

* input [31:0] address: Es la dirección de memoria para la operación.
* input [31:0] data\_write: Es la información que se escribirá en la memoria.
* input write\_enable: Es una señal que habilita la escritura.
* input read\_enable: Es una señal que habilita la lectura.
* output reg [31:0] data\_read: Es la información leída desde la memoria.
* Bloque reg [31:0] memory[0:255];: Declara una memoria de palabras de 32 bits llamada memory con 256 ubicaciones.

Bloque always @\*:

* Utiliza always @\* para que el bloque de código se ejecute siempre que haya un cambio en las entradas.
* Si write\_enable está habilitado (1) y read\_enable está deshabilitado (0), se escribe data\_write en la dirección especificada por address.
* Si write\_enable está deshabilitado (0) y read\_enable está habilitado (1), se lee la memoria en la dirección especificada por address y se almacena en data\_read.

Este módulo permite leer y escribir en la memoria según las señales de control proporcionadas. La operación de escritura se lleva a cabo si write\_enable está activo, y la operación de lectura se realiza si read\_enable está activo.

**MyPC.v**

module MyPC(

    input clk,

    input [31:0]in,

    output reg [31:0]out);

initial

begin

    out = 32'b0;

end

always@(clk)

begin

    if(clk)

    begin

        out = in;

    end

end

endmodule

Parámetros del módulo:

* input clk: Es la señal de reloj.
* input [31:0] in: Es la entrada que indica el nuevo valor del contador de programa.
* output reg [31:0] out: Es la salida que representa el valor actual del contador de programa.

Bloque initial:

* Inicializa la salida out con un valor de 32 bits en cero al comienzo de la simulación.

Bloque always @(posedge clk):

* Utiliza always @(posedge clk) para que el bloque de código se ejecute en cada flanco de subida del reloj.
* Actualiza la salida out con el valor de la entrada in en cada pulso del reloj.

Este módulo simula un contador de programa que se actualiza en cada pulso del reloj con el valor proporcionado en la entrada in.

**Shift\_Left\_2.v**

module Shift\_Left\_2(

input [31:0]SL\_in,

output reg[31:0]SL\_out

);

always @\*

    begin

       SL\_out = SL\_in << 2;

    end

endmodule

Parámetros del módulo:

* input [31:0] SL\_in: Es la entrada de 32 bits en la que se realizará el desplazamiento a la izquierda.
* output reg [31:0] SL\_out: Es la salida de 32 bits que contendrá el resultado del desplazamiento.

Bloque always @\*:

* Utiliza always @\* para que el bloque de código se ejecute cada vez que haya un cambio en las entradas.
* Realiza la operación de desplazamiento a la izquierda de 2 bits en la entrada SL\_in y almacena el resultado en la salida SL\_out.

Este módulo simplemente realiza la operación de desplazamiento a la izquierda de 2 bits en la entrada proporcionada.

**Sign\_extend.v**

module sign\_extend(

input [15:0] Data\_in,

output reg [31:0] Data\_out);

always @\*

begin

    if(Data\_in[15])

    begin

        Data\_out={16'b1111111111111111,Data\_in};

    end

    else

    begin

    Data\_out={16'b0000000000000000,Data\_in};

    end

end

endmodule

Parámetros del módulo:

* input [15:0] Data\_in: Es la entrada de 16 bits que se extenderá.
* output reg [31:0] Data\_out: Es la salida de 32 bits que contendrá el resultado de la extensión.

Bloque always @\*:

* Utiliza always @\* para que el bloque de código se ejecute cada vez que haya un cambio en las entradas.
* Verifica el bit de signo (Data\_in[15]), y si es 1, realiza la extensión de signo copiando el bit de signo a la izquierda.
* Si el bit de signo es 0, simplemente agrega ceros a la izquierda.

Este módulo se utiliza comúnmente para extender la representación de un número con signo de 16 bits a una de 32 bits, manteniendo el mismo valor numérico pero con la extensión de signo adecuada.

**Conclusiones**

NATALIA ISABEL MARISCAL NAPOLES: Sinceramente mis conclusiones con el proyecto y la materia son que en si con el proyecto al principio fue fácil y poco a poco se fue haciendo enredoso con el tema de cables y entradas o salidas, pero al final y prestando mucha atención se logró hacer, puede que no del todo bien o con algunas cosas raras y bastante presión pero se logró; y con respecto a la materia, aunque no era lo que yo esperaba, ni lo que nos dijeron que sería realmente me gusto la materia un poco confusa al principio pero bastante interesante y en verdad me hubiera gustado más que hubiéramos tenido el tiempo suficiente como para comprender del todo la materia, pero igual si aprendí mucho y espero no olvidar todo para cuando vuelva a ver el tema en la carrera de INRO.

MARÍN GONZÁLEZ ANDRÉ JOSUÉ: A lo largo del desarrollo del proyecto me perdí muchas veces a la hora de comprender el panorama general del mismo, no fue hasta la realización de este reporte donde pude más o menos (de manera sencilla), comprender el funcionamiento de cada uno de los módulos que lo componen (en lo que respecta a los módulos añadidos en esta ultima fase, el datapath tipo R si lo comprendo bien a nivel de funcionamiento). En lo que respecta a la materia esta última iteración me pareció la mas sencilla para comprender, aunque no por ello fácil, lo único que al final se me dificulto durante todo la materia fueron las conexiones entre módulos, entendía como hacerlo, pero al momento de hacer el primer datapath si me perdí mucho, ya que estoy muy acostumbrado a que las cosas se ejecuten de manera secuencial, pero en verilog el flujo de ejecución es diferente, lo que confunde mucho.

JUAN SILVERO VALENCIA

ERICK JARED GUTIERREZ CORREA