Proyecto Final

Diego Jared Jimenez Silva

Gael Ramses Alvarado Lomelí

220559845

220529474

11/05/2025

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 número 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.

Algoritmo ensamblador a implementar:

Este algoritmo en ensamblador MIPS identifica números primos dentro de un rango de 2 a 100 y los almacena en memoria. Además, mantiene un contador con la cantidad total de números primos encontrados.

Fundamento matemático:

Un número primo es un número natural mayor que 1 que solo tiene dos divisores positivos distintos: 1 y él mismo.

Para determinar si un número n es primo, el algoritmo prueba dividirlo por todos los enteros desde el 2 hasta n−1. Si ninguno divide exactamente a n, entonces es primo.

Esto se implementa usando restas sucesivas para calcular el módulo (n % d), ya que en nuestro procesador simple no se tiene una instrucción de división directa.

El banco de registros inicial para este algoritmo:

$0 00000000000000000000000000000000

$1 00000000000000000000000000000000

$2 00000000000000000000000000000000

$3 00000000000000000000000000000000

$4 00000000000000000000000000000000

$5 00000000000000000000000000000001

$6 00000000000000000000000000000000

$7 00000000000000000000000000000000

$8 00000000000000000000000000000010

$9 00000000000000000000000001100100

$10 00000000000000000000000000000000

$11 00000000000000000000000000000000

$12 00000000000000000000000000000000

$13 00000000000000000000000000000000

$14 00000000000000000000000000000000

$15 00000000000000000000000000000000

$16–$31 00000000000000000000000000000000

Variables importantes

- $8 = número actual (empieza en 2)
- $9 = número maximo (hasta 100)
- $10 = contador de primos
- $4 = dirección base para guardar datos
- $5 = desplazamiento de escritura (inicia en 1 porque mem[0] guarda el total de números primos)
- $11 = divisor actual
- $12 = es primo (bandera que indica si un número es primo o no)
- $13, $14, $15 = registros temporales (temp, bandera, trabajo_mod)

El algoritmo funciona de la siguiente manera:

1. Se compara numero_actual con numero_maximo. Si ya se pasó del límite, el algoritmo termina.
2. Se inicializa divisor = 2 y es_primo = 1.
3. Se compara el divisor con el numero_actual. Si el divisor es mayor o igual a numero_actual, se salta a la evaluación final.
4. Se copia numero_actual a trabajo_mod para simular el módulo (restas sucesivas).
5. Se hacen las restas hasta que trabajo_mod sea menor o igual a cero.

   Si trabajo_mod es igual a 0, el número es divisible, por lo que no es primo y es_primo = 0.
6. Si no fue divisible por ningún divisor anterior, se guarda en memoria.
7. Finalmente, se incrementa numero_actual y se repite el ciclo.

Seudocódigo:

```
numero_actual ← 2
numero_maximo ← 100
contador_primos ← 0
direccion_base ← 0
desplazamiento ← 1

Mientras numero_actual ≤ numero_maximo - 1 Hacer:
    divisor ← 2
    es_primo ← 1

    Mientras divisor < numero_actual Hacer:
        trabajo_mod ← numero_actual

        Mientras trabajo_mod > 0 Hacer:
            trabajo_mod ← trabajo_mod - divisor

        Si trabajo_mod == 0 Entonces
```

```
            es_primo ← 0
            Terminar

        divisor ← divisor + 1
    Terminar mientras

    Si es_primo == 1 Entonces
        memoria[direccion_base + desplazamiento] ← numero_actual
        desplazamiento ← desplazamiento + 1
        contador_primos ← contador_primos + 1
    Termir Si

    numero_actual ← numero_actual + 1
Terminar Mientras

memoria[direccion_base] ← contador_primos
```

Estas son las instrucciones que conforman el algoritmo:

Instrucción 0: ADDI $8, $0, 2: Ponemos el número 2 en $8, que será el primer número para revisar.

Instrucción 1: ADDI $9, $0, 100: Guardamos el número máximo (100) en $9. Vamos a revisar del 2 al 99.

Instrucción 2: ADDI $10, $0, 0: Inicializamos el contador de primos en $10 con 0.

Instrucción 3: ADDI $4, $0, 0: $4 va a ser la dirección base de memoria (empezamos en 0).

Instrucción 4: ADDI $5, $0, 1: Ponemos 1 en $5 porque mem[0] la dejamos para el total de primos. Empezamos a guardar desde mem[1].

Instrucción 5: SUB $13, $9, $8: Restamos el número actual al máximo, para ver si aún hay números que revisar.

Instrucción 6: SLT $14, $0, $13: Si el resultado fue positivo, significa que aún no llegamos al límite.

Instrucción 7: BEQ $14, $0, 21: Si ya no quedan más números (comparación dio 0), salimos del programa.

Instrucción 8: ADDI $11, $0, 2: Empezamos probando con el divisor 2.

Instrucción 9: ADDI $12, $0, 1: Asumimos que el número sí es primo (bandera en 1).

Instrucción 10: SUB $13, $8, $11: Calculamos si el divisor todavía es menor que el número.

Instrucción 11: SLT $14, $0, $13: Si el divisor ya no es menor, salimos del ciclo de divisores.

Instrucción 12: BEQ $14, $0, 9: Salimos del ciclo si ya probamos todos los divisores.

Instrucción 13: ADD $15, $0, $8: Copiamos el número actual a $15, lo vamos a usar para dividir.

Instrucción 14: SUB $15, $15, $11: Le vamos restando el divisor al número. Esto es como una división por restas.

Instrucción 15: SLT $14, $0, $15: Si el número ya no es mayor que 0, terminamos las restas.

Instrucción 16: BEQ $14, $0, 1: Si terminó, vamos a verificar si sobró algo.

Instrucción 17: J 14: Si todavía no llegamos a 0, seguimos restando.

Instrucción 18: BEQ $15, $0, 2: Si al final $15 quedó en 0, el número es divisible, NO es primo.

Instrucción 19: ADDI $11, $11, 1: Aumentamos el divisor (probamos con el siguiente).

Instrucción 20: J 10: Volvemos a probar con el nuevo divisor.

Instrucción 21: ADDI $12, $0, 0: Ponemos $12 en 0 porque no es primo.

Instrucción 22: BEQ $12, $0, 4: Si no es primo, saltamos y no lo guardamos.

Instrucción 23: ADD $13, $4, $5: Calculamos en qué dirección de memoria guardarlo.

Instrucción 24: SW $8, 0($13): Guardamos el número primo en memoria.

Instrucción 25: ADDI $5, $5, 1: Aumentamos el desplazamiento para guardar el siguiente**.**

Instrucción 26: ADDI $10, $10, 1: Aumentamos el contador de números primos.

Instrucción 27: ADDI $8, $8, 1: Vamos al siguiente número a revisar.

Instrucción 28: J 5: Volvemos al inicio del ciclo.

Instrucción 29: SW $10, 0($4): Guardamos en mem[0] la cantidad total de primos que encontramos.

Instrucción 30: NOP: No hace nada. Es solo para marcar el fin del programa.

Código ensamblador completo:

```
    ADDI  $8,  $0, 2        # numero_actual = 2
    ADDI  $9,  $0, 100      # numero_maximo = 100
    ADDI  $10, $0, 0        # contador_primos = 0
    ADDI  $4,  $0, 0        # direccion_base para datos
    ADDI  $5,  $0, 1        # desplazamiento_escritura inicia en 1
# Se preparan los registros para comenzar. El número actual es 2, y queremos llegar hasta el 99.
    SUB   $13, $9, $8       # temp = numero_maximo - numero_actual
    SLT   $14, $0, $13      # bandera = (0 < temp)
    BEQ   $14, $0, 21       # si temp ≤ 0 → SALIDA (fin)
# Si ya revisamos todos los números hasta 99, el programa termina. Si no, sigue.
    ADDI  $11, $0, 2        # divisor = 2
    ADDI  $12, $0, 1        # es_primo = 1
# Queremos probar si numero_actual es divisible entre 2, 3, ..., hasta uno antes de él mismo.
    SUB   $13, $8, $11      # temp = numero_actual - divisor
    SLT   $14, $0, $13      # bandera = (0 < temp)
    BEQ   $14, $0, 9        # si divisor ≥ numero_actual → FIN_DIV
    ADD   $15, $0, $8       # trabajo_mod = numero_actual
# En vez de dividir, restamos el divisor varias veces.
    SUB   $15, $15, $11     # trabajo_mod -= divisor
    SLT   $14, $0, $15      # bandera = (0 < trabajo_mod)
    BEQ   $14, $0, 1        # si trabajo_mod ≤ 0 → VERIFICAR_MOD
    J     14                # sino → seguir MOD_LOOP
#  Restamos hasta que ya no podamos. Si el resultado final fue 0, es divisible
    BEQ   $15, $0, 2        # si trabajo_mod == 0 → NO_ES_PRIMO
    ADDI  $11, $11, 1       # divisor++
```

```
    J     10              # volver a DIV_LOOP
```

Si encontramos algún divisor que lo divida exacto, ya no es primo.

```
    ADDI  $12, $0, 0      # es_primo = 0
```

Marcamos que **no es primo**.

```
    BEQ   $12, $0, 4      # si no es primo → OMITIR_ESCRITURA
    ADD   $13, $4, $5     # calcular dirección de escritura
    SW    $8, 0($13)      # guardar numero_actual en memoria
    ADDI  $5, $5, 1       # desplazamiento_escritura++
    ADDI  $10, $10, 1     # contador_primos++
```

Guardamos el número primo en la memoria, y aumentamos el contador y el offset.

```
    ADDI  $8, $8, 1       # numero_actual++
    J     5               # volver a LOOP_MAIN
```

Revisamos el siguiente número, y repetimos todo.

SALIDA:

```
    SW    $10, 0($4)      # guardar contador_primos en mem[0]
    NOP                   # fin de programa
```

Al final, nos aseguramos de que el total de primos quede guardado en la memoria.

Por ahora usaremos el algoritmo para calcular cuantos números primos hay hasta el número 100 y saber cuáles son, pero si quisiéramos aumentar o disminuir el límite de 100, solo sería necesario modificar esta línea del código:

```
ADDI  $9,  $0, 100     # numero_maximo = 100.
```

Y modificar esta parte del banco de registros:

$9 00000000000000000000000001100100.

$9 = número maximo (hasta 100).

Modificando esta parte del código y del banco de registros, el algoritmo tendría que ser capas de calcular los números primos del nuevo limite y guardarlos en la memoria de datos.

Este algoritmo será puesto aprueba una vez que hayamos terminado las tres fases del procesador mips.

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 porqué 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 módulo 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.

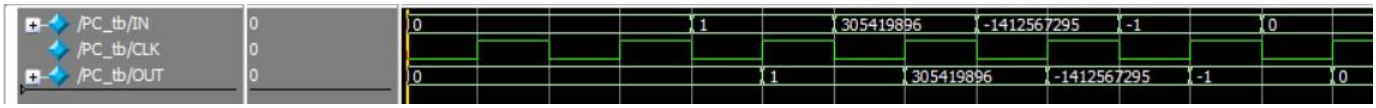
A continuación, se adjunta una imagen del código en Verilog.

```
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.

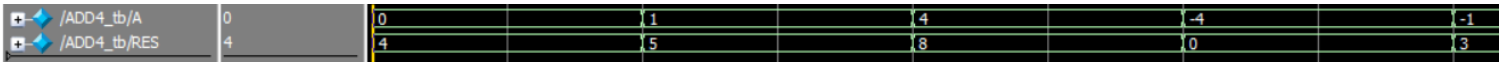
A continuación, se adjunta una imagen del código en Verilog.

```
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.

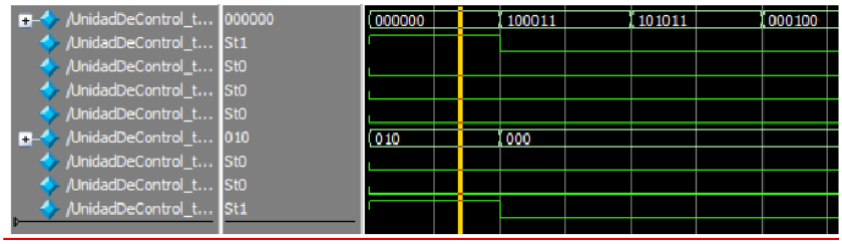
```
// UnidadDeControl
module UnidadDeControl (
    input  wire [5:0]  OpCode,
    output reg         RegDst,
    output reg         Branch,
    output reg         MemRead,
    output reg         MemToReg,
    output reg [2:0]   ALUOp,
    output reg         MemToWrite,
    output reg         ALUSrc,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

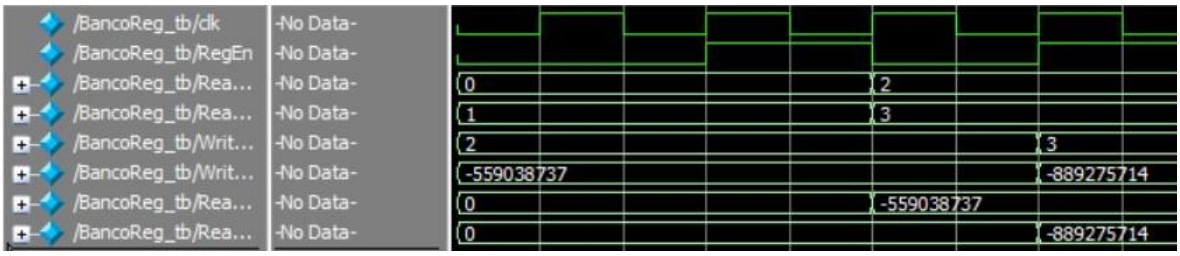
El testbench para probar que funcione correctamente este módulo es:

```
`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.

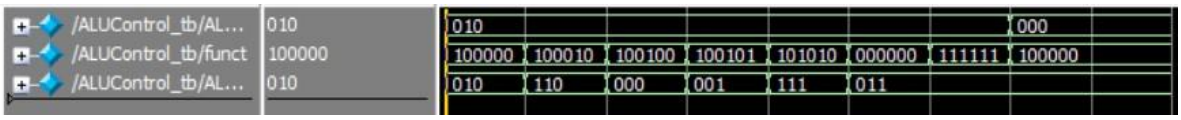
```v
// ALuControl
module ALuControl (
    input  wire [2:0] ALUOp,
    input  wire [5:0] funct,
    output reg  [2:0] ALUCtl
);
    always @(*) begin
        case (ALUOp)
            3'b010: begin // instrucciones R
                case (funct)
                    6'b100000: ALUCtl = 3'b010; // ADD
                    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
```

```v
`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
```

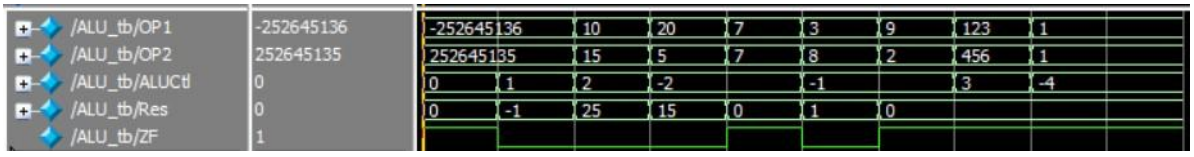
El testbench para probar que funcione correctamente este módulo es:

```
`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.

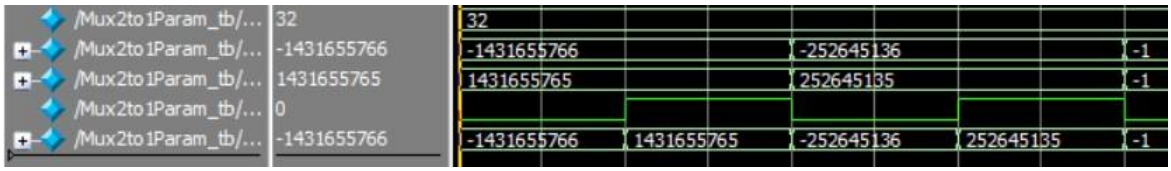
```
module MuxWriteData #(
    parameter WIDTH = 32  // Permite configurar el tamaño al instanciarlo
)(
    input  wire [WIDTH-1:0] entrada0,  // opción 0
    input  wire [WIDTH-1:0] entrada1,  // opción 1
    input  wire             sel,       // selector
    output reg  [WIDTH-1:0] salida     // resultado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



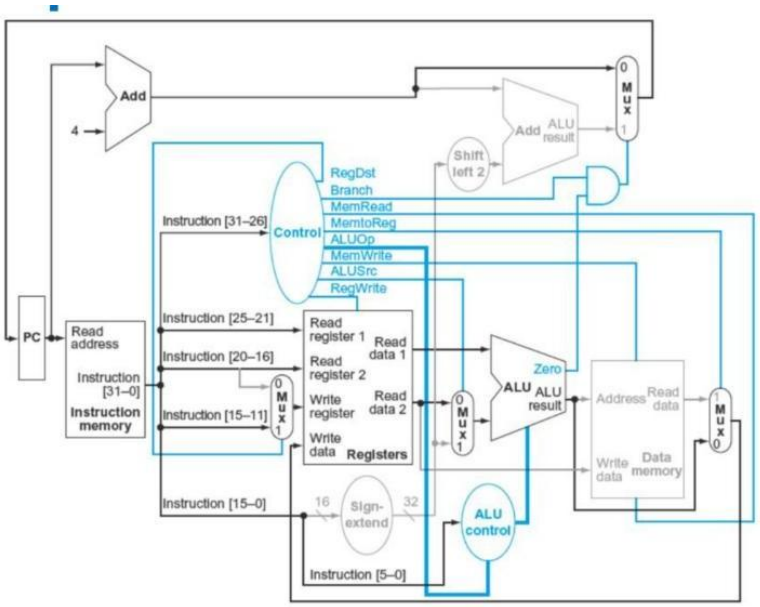
Fase 1 Datapath Completo (instrucciones tipo R):

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 módulo PC.

Con ello MEMI lee la posición dada más 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.

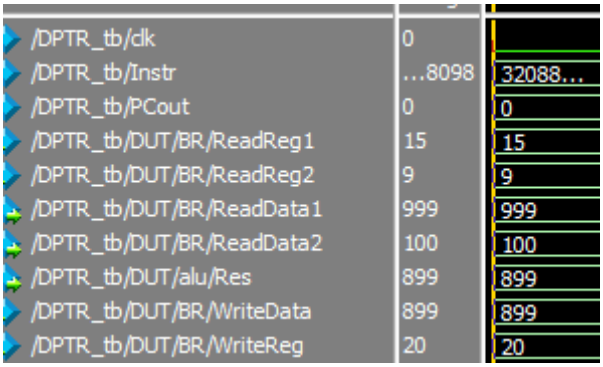
- Opcode: 000000.
- rs: 01111.
- rt: 01001.
- rd: 10100.
- funct: 100010.

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 mandamos 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.

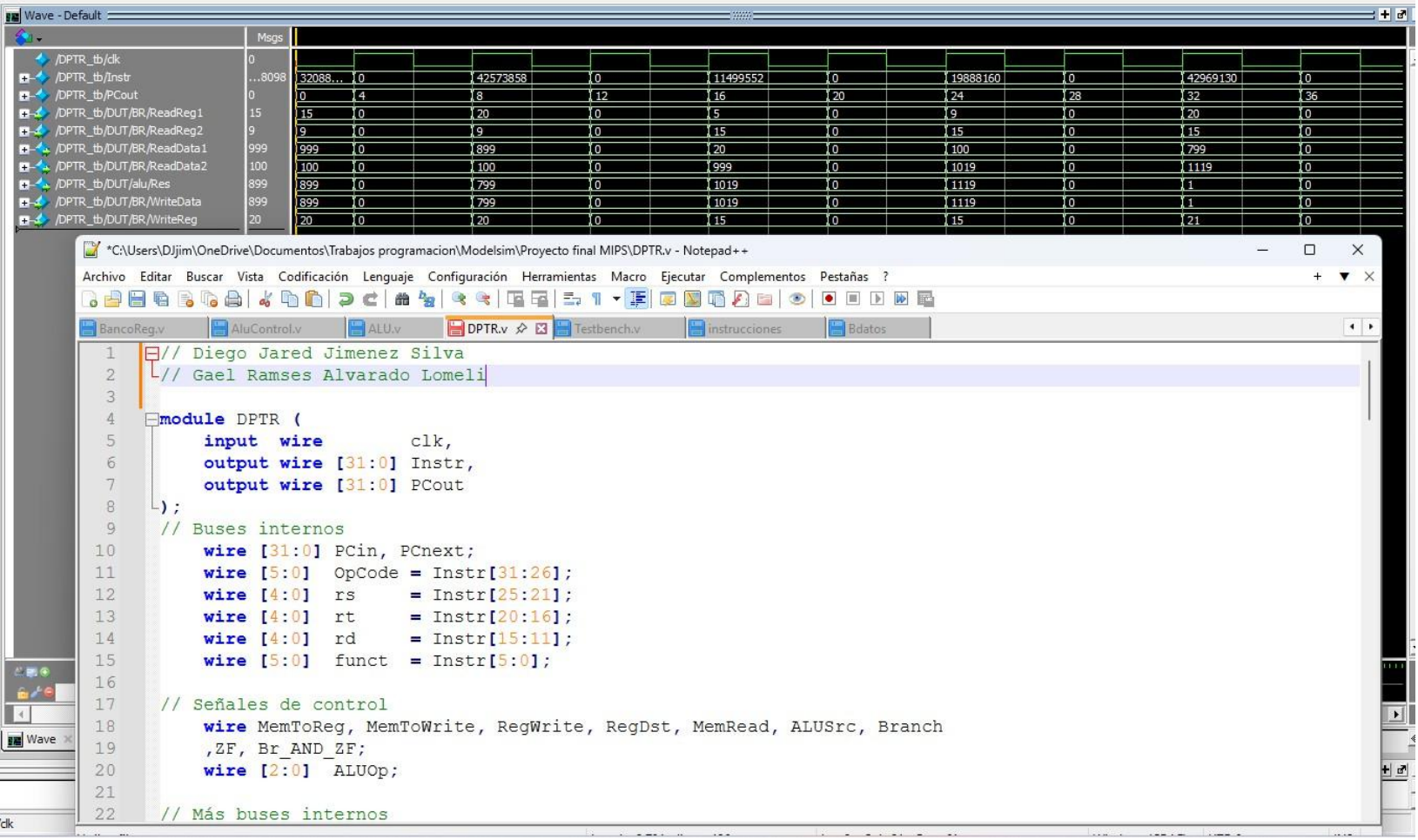
Con esto podemos ver el funcionamiento del módulo y el cómo interactúan todos los componentes para procesar una instrucción de tipo R en su totalidad.



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

Fase 2

Modulos modificados:

Unidad de control:

Seguidamente, en la fase 2 utilizamos distintas instrucciones tipo I, donde los OpCodes varían, y cada uno está relacionado a una operación especifica de acuerdo con el set de instrucciones tipo MIPS. Aquí utilizamos Addi, Ori, Andi, Slti, Lw, Sw, Beq, Bne, Bgtz. En las terminaciones i (addi, ori, etc) activamos ALUSrc (para sumar el valor inmediato), RegWrite (para escribir en el banco de registros) y ALUOp (para indicar la operación que tiene que hacer la ALU). Por otra parte, para Lw activamos RegWrite porque escribiremos en el banco de registros, ALUOp que indica la operación que hará la ALU (suma), ALUSrc porque utilizaremos el inmediato (offset), MemRead porque leeremos de memoria, y MemToReg porque guardaremos lo que nos de la memoria al banco de registros. Para Sw utilizamos ALUOp para indicar la operación (suma), ALUSrc porque utilizaremos el inmediato (offset) y MemWrite porque escribiremos en memoria. Como ultimo para las instrucciones Branch, utilizamos ALUOp para todas, siedo una resta para beq y bne, y bgtz para actividad una operación en la ALU que compruebe si el nuemero es mayor que cero (positivo), en todos estos activamos el Branch.

```
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         MemWrite,
    output reg         ALUSrc,
    output reg         RegWrite
);

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

    // Nota: Como 'subi' no existe como instrucción oficial en MIPS
    // Se utiliza 'addi' con un inmediato negativo

    case (OpCode)
        6'b000000: begin // R-type
            RegWrite = 1'b1;
            RegDst   = 1'b1;
            ALUOp    = 3'b010;
        end
        6'b001000: begin // Addi
            ALUSrc   = 1'b1;
            RegWrite = 1'b1;
            ALUOp    = 3'b000;
        end
        6'b001101: begin // Ori
            ALUSrc   = 1'b1;
            RegWrite = 1'b1;
            ALUOp    = 3'b101;
        end
        6'b001100: begin // Andi
            ALUSrc   = 1'b1;
            RegWrite = 1'b1;
            ALUOp    = 3'b100;
        end
```

```
        6'b001010: begin // Slti
            ALUSrc   = 1'b1;
            RegWrite = 1'b1;
            ALUOp    = 3'b111;
        end
        6'b100011: begin // lw
            RegWrite  = 1'b1;
            ALUOp     = 3'b000;
            ALUSrc    = 1'b1;
            MemRead   = 1'b1;
            MemToReg  = 1'b1;
        end
        6'b101011: begin // sw
            ALUOp     = 3'b000;
            ALUSrc    = 1'b1;
            MemWrite  = 1'b1;
        end
        6'b000100: begin // beq
            Branch = 1'b1;
            ALUOp  = 3'b001;
        end
        6'b000101: begin // bne
            Branch = 1'b1;
            ALUOp  = 3'b001;
        end
        6'b000111: begin // bgtz
            Branch = 1'b1;
            ALUOp  = 3'b110;
        end
    endcase
end
endmodule
```

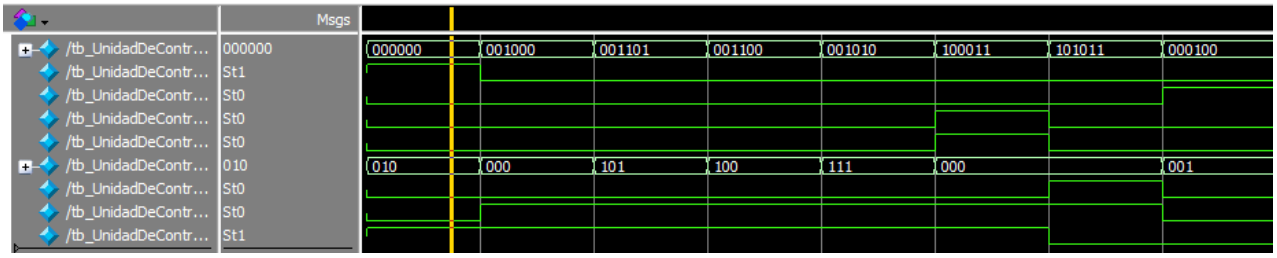
Testbench:

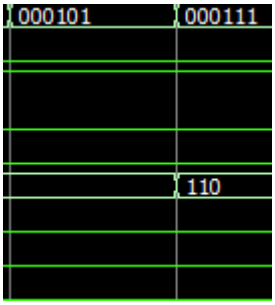
```
`timescale 1ns / 1ps

module tb_UnidadDeControl;

  // Señales de prueba
  reg  [5:0] OpCode;
  wire       RegDst;
  wire       Branch;
  wire       MemRead;
  wire       MemToReg;
  wire [2:0] ALUOp;
  wire       MemWrite;
  wire       ALUSrc;
  wire       RegWrite;

  // Instanciación del DUT
  UnidadDeControl uut (
    .OpCode   (OpCode),
    .RegDst   (RegDst),
    .Branch   (Branch),
    .MemRead  (MemRead),
    .MemToReg (MemToReg),
    .ALUOp    (ALUOp),
    .MemWrite (MemWrite),
    .ALUSrc   (ALUSrc),
    .RegWrite (RegWrite)
  );

  initial begin

    // R-type (000000)
    OpCode = 6'b000000; #5;

    // addi (001000)
    OpCode = 6'b001000; #5;

    // ori (001101)
    OpCode = 6'b001101; #5;

    // andi (001100)
    OpCode = 6'b001100; #5;

    // slti (001010)
    OpCode = 6'b001010; #5;

    // lw (100011)
    OpCode = 6'b100011; #5;
```

```
    // sw (101011)
    OpCode = 6'b101011; #5;

    // beq (000100)
    OpCode = 6'b000100; #5;

    // bne (000101)
    OpCode = 6'b000101; #5;

    // bgtz (000111)
    OpCode = 6'b000111; #5;

    $finish;
  end

endmodule
```

El testbench cambia el valor del OpCode cada 5 nanosegundos para representar diferentes tipos de instrucciones (como R-type, addi, lw, sw, etc.), permitiendo observar cómo responde la UnidadDeControl a cada uno

Resultados:





Alucontrol:

En la fase 2. Las instrucciones de tipo I no utilizan el campo Funct. En su lugar, utilizamos el OpCode para definir la operación a realizar, permitiendo el uso de los inmediatos. Esto hace que se procesen diferente los bits, esto puesto que el Funct de las instrucciones tipo R y un segundo operando se remplazan por un inmediato en estas instrucciones tipo I. Aquí utilizamos las siguientes operaciones: ADD, SUB, AND, OR, BGTZ, SLT, NOP.

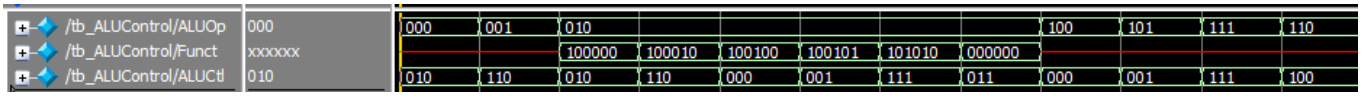
```
module ALUControl(
    input  wire [2:0] ALUOp,
    input  wire [5:0] Funct,
    output reg  [2:0] ALUCtl
);

always @(*) begin
    case (ALUOp)
        3'b000: ALUCtl = 3'b010; // ADD (para addi, lw, sw)
        3'b001: ALUCtl = 3'b110; // SUB (para beq, bne)

        // Instrucciones tipo R
        3'b010: begin
            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
            endcase
        end

        3'b100: ALUCtl = 3'b000; // AND (para andi)
        3'b101: ALUCtl = 3'b001; // OR  (para ori)
        3'b110: ALUCtl = 3'b100; // BGTZ
        3'b111: ALUCtl = 3'b111; // SLT (slti)
        default: ALUCtl = 3'b011; // NOP / default
    endcase
end

endmodule
```

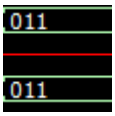
Testbench:

```
`timescale 1ns / 1ps

module tb_ALUControl;

  // Señales de prueba
  reg  [2:0]  ALUOp;
  reg  [5:0]  Funct;
  wire [2:0]  ALUCtl;

  // Instanciación del DUT
  ALUControl uut (
    .ALUOp  (ALUOp),
    .Funct  (Funct),
    .ALUCtl (ALUCtl)
  );
  initial begin

    // 1) ADDI/LW/SW (ALUOp=000)
    ALUOp = 3'b000; Funct = 6'bxxxxxx; #5;

    // 2) BEQ/BNE (ALUOp=001)
    ALUOp = 3'b001; Funct = 6'bxxxxxx; #5;

    // 3) R-type (ALUOp=010)
    ALUOp = 3'b010;
    Funct = 6'b100000; #5; // ADD
    Funct = 6'b100010; #5; // SUB
    Funct = 6'b100100; #5; // AND
    Funct = 6'b100101; #5; // OR
    Funct = 6'b101010; #5; // SLT
    Funct = 6'b000000; #5; // NOP

    // 4) ANDI (ALUOp=100)
    ALUOp = 3'b100; Funct = 6'bxxxxxx; #5;

    // 5) ORI (ALUOp=101)
    ALUOp = 3'b101; Funct = 6'bxxxxxx; #5;

    // 6) SLTI (ALUOp=111)
    ALUOp = 3'b111; Funct = 6'bxxxxxx; #5;

    // 7) BGTZ (ALUOp=110)
    ALUOp = 3'b110; Funct = 6'bxxxxxx; #5;

    // 8) Default (ALUOp=011)
    ALUOp = 3'b011; Funct = 6'bxxxxxx; #5;

    $finish;
  end
```

Simula instrucciones tipo R, así como instrucciones inmediatas como ADDI, ANDI, ORI, etc. Para cada combinación, espera un breve tiempo y observa la salida ALUCtl, que indica qué operación debe hacer la ALU.

Resultados:





ALU

En la fase 2 simplemente agregamos BGTZ, aquí se comprueba si el número es mayor a cero, si es así entonces es 0, lo que activa inmediatamente el ZF, si no es así entonces es 1. Esto para activad una operación Branch más adelante.

```
module ALU(
    input  wire [31:0] Op1,
    input  wire [31:0] Op2,
    input  wire [2:0]  ALUCtl,    // Lee del ALUControl
    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'b100: Res = ($signed(Op1) > 0) ? 32'd0 : 32'd1; //BGTZ
        3'b011: Res = 32'd0;                            // NOP
        default: Res = 32'd0;
    endcase
end

assign ZF = (Res == 32'd0);

endmodule
```

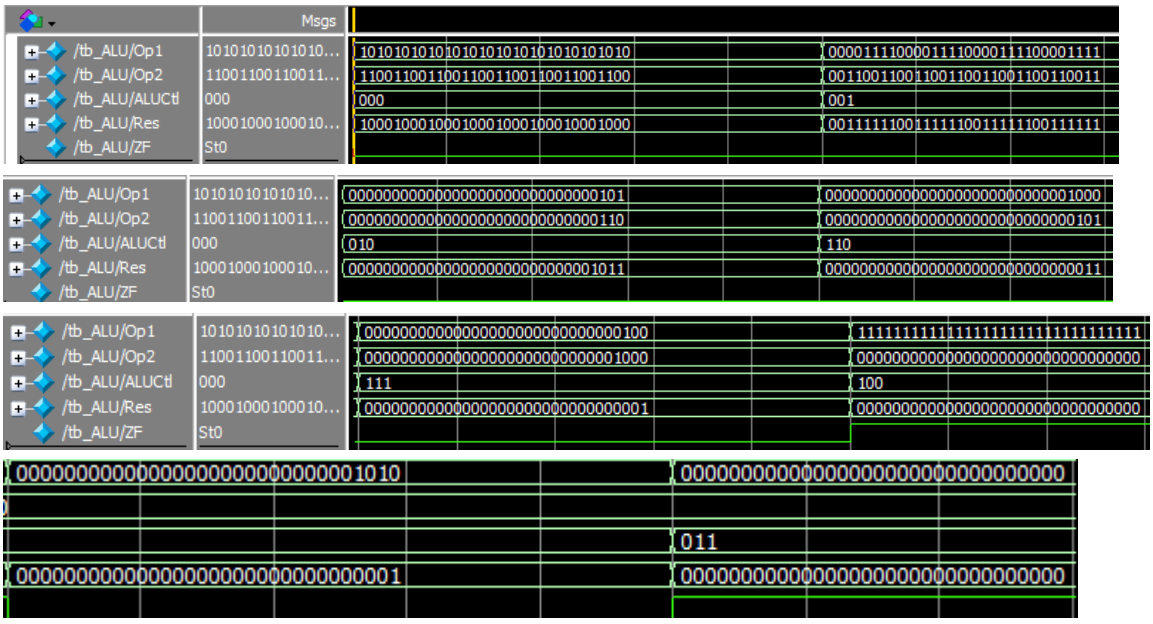
Testbench:

```
`timescale 1ns / 1ps

module tb_ALU;

  // Señales de prueba
  reg  [31:0] Op1;
  reg  [31:0] Op2;
  reg  [2:0]  ALUCtl;
  wire [31:0] Res;
  wire        ZF;

  // Instanciación del DUT
  ALU uut (
    .Op1(Op1),
    .Op2(Op2),
    .ALUCtl(ALUCtl),
    .Res(Res),
    .ZF(ZF)
  );

  initial begin
    // Caso 1: AND
    ALUCtl = 3'b000; // AND
    Op1    = 32'b10101010101010101010101010101010;
    Op2    = 32'b11001100110011001100110011001100;
    #10;

    // Caso 2: OR
    ALUCtl = 3'b001; // OR
    Op1    = 32'b00001111000011110000111100001111;
    Op2    = 32'b00110011001100110011001100110011;
    #10;

    // Caso 3: ADD
    ALUCtl = 3'b010; // ADD
    Op1    = 32'b00000000000000000000000000000101; // 5
    Op2    = 32'b00000000000000000000000000000110; // 6
    #10;
```

```
    // Caso 4: SUB
    ALUCtl = 3'b110; // SUB
    Op1    = 32'b00000000000000000000000000001000; // 8
    Op2    = 32'b00000000000000000000000000000101; // 5
    #10;

    // Caso 5: SLT
    ALUCtl = 3'b111; // SLT
    Op1    = 32'b00000000000000000000000000000100; // 4
    Op2    = 32'b00000000000000000000000000001000; // 8
    #10;

    // Caso 6: BGTZ (Op1 > 0?)
    ALUCtl = 3'b100; // BGTZ
    Op1    = 32'b11111111111111111111111111111111; // -1 (signed)
    Op2    = 32'b00000000000000000000000000000000; // no usado
    #10;
    ALUCtl = 3'b100; // BGTZ
    Op1    = 32'b00000000000000000000000000001010; // 10
    #10;

    // Caso 7: NOP
    ALUCtl = 3'b011; // NOP
    Op1    = 32'b00000000000000000000000000000000;
    Op2    = 32'b00000000000000000000000000000000;
    #10;

    // Fin de simulación
    $finish;
  end

endmodule
```

Este testbench prueba las operaciones básicas de una ALU (Unidad Aritmético Lógica). Le envía diferentes combinaciones de operandos (Op1 y Op2) junto con el código de operación (ALUCtl) para simular instrucciones como AND, OR, ADD, SUB, SLT, BGTZ y NOP. Luego espera un breve tiempo para observar el resultado (Res) y la bandera de cero (ZF), permitiendo verificar que la ALU esté funcionando correctamente.

Resultados:



Modulos nuevos:

Buffer

El módulo BF 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 módulo BF es utilizado para almacenar temporalmente valores y transferirlos a la salida en un siguiente pulso de reloj.

En la funcionalidad del módulo, utilizamos un bloque always @(posedge CLK) para que, en que, en cada subida del reloj, la entrada IN sea asignada a la salida OUT y libere esos valores guardados. Con ello hacemos uso del operador <= para hacer uso de asignaciones en paralelo.

La lógica consistiría en guardar un dato de entrada y mantenerlo hasta un próximo ciclo de reloj.

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

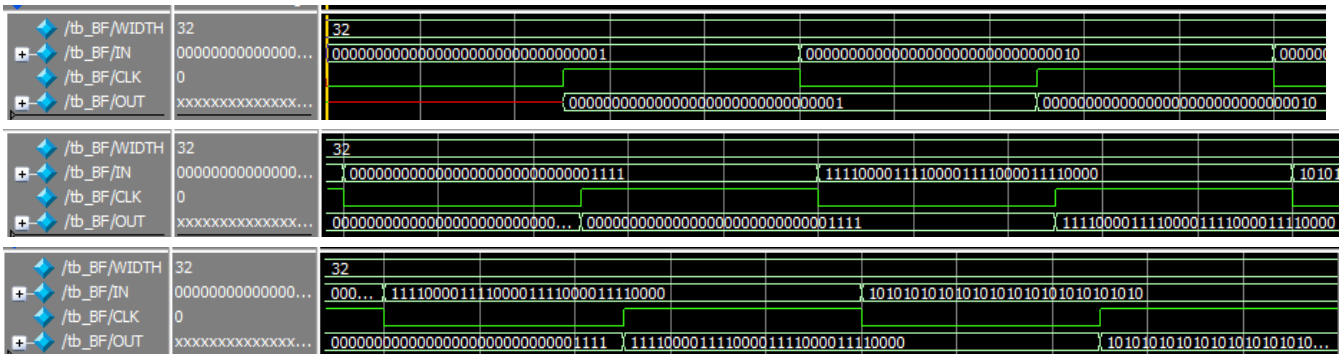
```
module BF #(parameter WIDTH = 32) (
    input [WIDTH-1:0] IN,
    input CLK,
    output reg [WIDTH-1:0] OUT
);

always @(posedge CLK) begin // Cada positivo en el reloj
    OUT <= IN;  // Utilizamos <= para las distintas asignaciones en paralelo
end

endmodule
```

Testbench:

```
`timescale 1ns / 1ps

module tb_BF;

  // Parámetros y señales
  localparam WIDTH = 32;
  reg  [WIDTH-1:0] IN;
  reg              CLK;
  wire [WIDTH-1:0] OUT;

  // Instanciación del DUT
  BF #(
    .WIDTH(WIDTH)
  ) uut (
    .IN(IN),
    .CLK(CLK),
    .OUT(OUT)
  );

  // Generación de reloj: periodo 10ns
  initial begin
    CLK = 0;
    forever #5 CLK = ~CLK;
  end

  initial begin

    // Vectores de prueba: aplicados en flanco de reloj
    IN = 32'b00000000000000000000000000000001;  // 1
    #10;
    IN = 32'b00000000000000000000000000000010;  // 2
    #10;
    IN = 32'b00000000000000000000000000001111;  // 15
    #10;
    IN = 32'b11110000111100001111000011110000;  // patrón
    #10;
    IN = 32'b10101010101010101010101010101010;  // alternancia
    #10;

    // Fin de simulación
    $finish;
  end

endmodule
```

Este testbench toma una entrada de 32 bits (IN) y la transfiere a la salida (OUT) sincronizada con un reloj (CLK). Se genera un reloj de 10 ns de periodo, y se aplican distintos valores a la entrada en cada flanco de subida para observar cómo se comporta la salida.

Resultados:



AdderBranch

El módulo AdderBranch está conformado por 2 entradas y 1 salida. Las entradas son las siguientes: PCplus4 (32 bits) y Shifted (32 bits). La salida es la siguiente: PCBranch (32 bits).

El módulo AdderBranch se utiliza para calcular un salto a una nueva posición

dentro de la memoria de instrucciones, esto sumando el valor de PCplus4 y Shifted.

Aquí utilizamos un assign para tomar el valor de la suma de PCplus4 y Shifted y guardarla en PCBranch. Con esto hacemos una actualización instantánea del valor sin esperar un pulso de reloj como en otros casos.

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

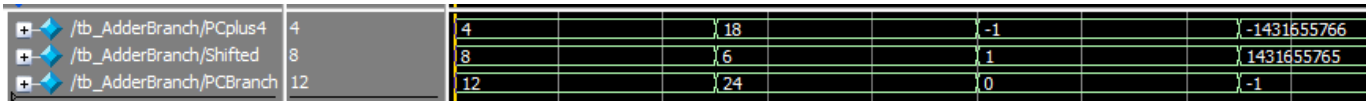
```
module AdderBranch(
   input  wire [31:0] PCplus4,
   input  wire [31:0] Shifted,
   output wire [31:0] PCBranch
);

assign PCBranch = PCplus4 + Shifted;

endmodule
```

Testbench:

```
`timescale 1ns / 1ps

module tb_AdderBranch;

  // Entradas y salidas
  reg  [31:0] PCplus4;
  reg  [31:0] Shifted;
  wire [31:0] PCBranch;

  // Instanciación del DUT (Device Under Test)
  AdderBranch uut (
    .PCplus4(PCplus4),
    .Shifted(Shifted),
    .PCBranch(PCBranch)
  );

  initial begin

    // Vectores de prueba
    // 1) Suma simple: 4 + 8 = 12
    PCplus4 = 32'b00000000000000000000000000000100;  // 4
    Shifted = 32'b00000000000000000000000000001000;  // 8
    #5;

    // 2) Suma con diferentes bits
    PCplus4 = 32'b00000000000000000000000000010010;  // 18
    Shifted = 32'b00000000000000000000000000000110;  // 6
    #5;

    // 3) Suma con carry a bit alto
    PCplus4 = 32'b11111111111111111111111111111111;  // -1 (2's complement)
    Shifted = 32'b00000000000000000000000000000001;  // 1
    #5;

    // 4) Suma de patrones
    PCplus4 = 32'b10101010101010101010101010101010;
    Shifted = 32'b01010101010101010101010101010101;
    #5;

    // Fin de la simulación
    $finish;
  end

endmodule
```

Este testbench calcula la dirección de salto en instrucciones de tipo branch. Le entrega dos entradas de 32 bits (PCplus4 y Shifted) que representan la dirección del siguiente salto y el desplazamiento. Para cada caso de prueba, se suman y se observa la salida PCBranch, asegurando que el sumador funcione correctamente.

Resultados:



ShiftLeft2

El módulo ShiftLeft2 está conformado por 1 entrada y 1 salida. La entrada es la siguiente: In (32 bits). La salida es la siguiente: Out (32 bits).

El módulo ShiftLeft2 funciona para hacer un desplazamiento del número binario 2 posiciones hacia la izquierda. Esto lo hacemos porque en la memoria guardamos las instrucciones cada 4 posiciones, y al multiplicar el valor de entrada por 4, se recorre el mismo número dos espacios hacía la izquierda.

En la funcionalidad del módulo, utilizamos un assign para que Out tome inmediatamente el valor de In ya desplazado. Aquí no recorremos como tal, sino que concatenando dos bits 0 a la derecha de In para multiplicar por 4 el valor.

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

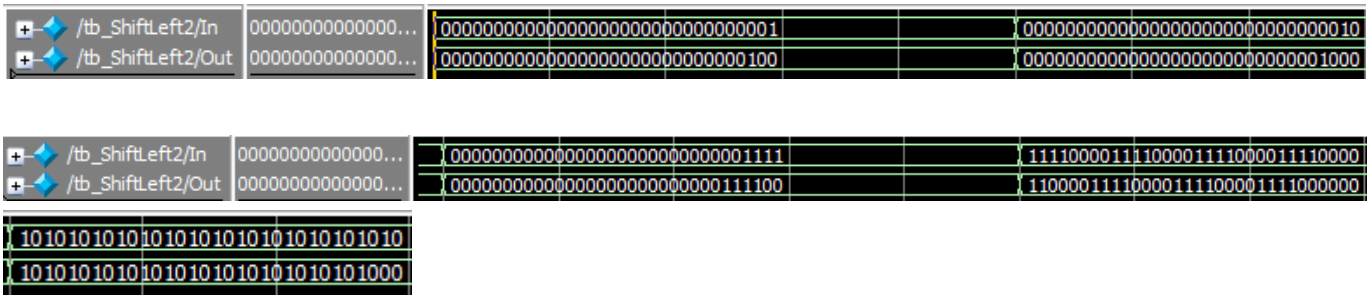
```
module ShiftLeft2(
   input  wire [31:0] In,
   output wire [31:0] Out
);

// Hacemos un recorrido concatenando dos bits 0
// Con ello multiplicamos x4 en binario
assign Out = {In[29:0], 2'b00};

endmodule
```

Testbench:

```
`timescale 1ns / 1ps

module tb_ShiftLeft2;

  // Entradas y salidas
  reg  [31:0] In;
  wire [31:0] Out;

  // Instanciación del DUT (Device Under Test)
  ShiftLeft2 uut (
    .In(In),
    .Out(Out)
  );

  initial begin

    // Vectores de prueba
    In = 32'b00000000000000000000000000000001;  // 1 -> 4
    #5;
    In = 32'b00000000000000000000000000000010;  // 2 -> 8
    #5;
    In = 32'b00000000000000000000000000001111;  // 15 -> 60
    #5;
    In = 32'b11110000111100001111000011110000;  // prueba de patrón
    #5;
    In = 32'b10101010101010101010101010101010;  // prueba de alternancia
    #5;

    // Fin de la simulación
    $finish;
  end

endmodule
```

Se aplican diferentes patrones de entrada (valores pequeños, binarios específicos y patrones alternantes) y se espera un breve tiempo para observar el resultado desplazado en la salida Out.

Resultados:



SignExtend

El módulo SignExtend está conformado por 1 entrada y 1 salida. La entrada es la siguiente: Imm16 (16 bits). La salida es la siguiente: Imm32 (32 bits).

El módulo SignExtend hace que un numero binario de 16 bits pase a ser de 32 bits y que este represente el mismo valor. Esta operación se utiliza para manejar los inmediatos en arquitecturas de procesadores MIPS.

En la funcionalidad del módulo, utilizamos un bloque always @(*), lo que permite actualizar el valor Imm32 si es que Imm16 cambia. La operación se realiza mediante el operador {} que sirve para concatenar, aquí replicamos el bit de signo del bit más significativo de Imm16, esto 16 veces para tener un valor de 32 bits.

La lógica consiste en mantener el valor del número mientras se ajusta a una representación de 32 bits.

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

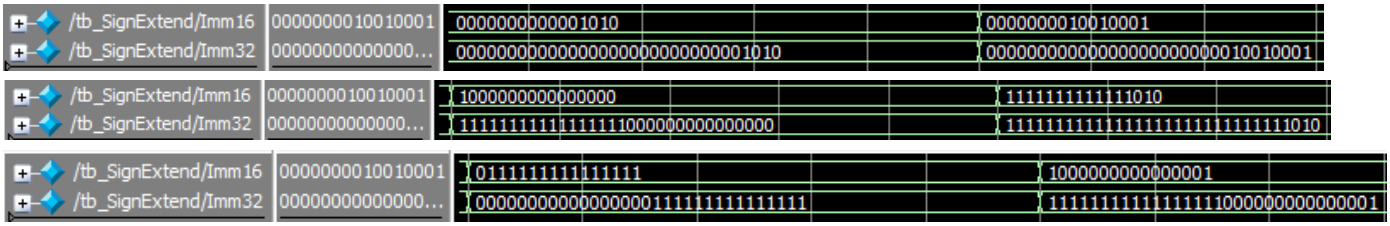
```
module SignExtend(
  input  wire [15:0] Imm16,
  output reg  [31:0] Imm32
);

// El operador {} replica el bit de signo 16 veces y se concatena con Imm16
always @(*) Imm32 = {{16{Imm16[15]}}, Imm16};

endmodule
```

Testbench:

```
`timescale 1ns / 1ps

module tb_SignExtend;
    // Señales de prueba
    reg  [15:0] Imm16;
    wire [31:0] Imm32;

    // Instanciación del módulo bajo prueba
    SignExtend uut (
        .Imm16(Imm16),
        .Imm32(Imm32)
    );

    initial begin

        // Vector de pruebas
        #10 Imm16 = 16'b0000_0000_0000_1010; // +10
        #10 Imm16 = 16'b0000_0000_1001_0001; // +145
        #10 Imm16 = 16'b1000_0000_0000_0000; // -32768
        #10 Imm16 = 16'b1111_1111_1111_1010; // -6
        #10 Imm16 = 16'b0111_1111_1111_1111; // +32767
        #10 Imm16 = 16'b1000_0000_0000_0001; // -32767

        #10 $finish;
    end
endmodule
```

Se prueban valores positivos y negativos, incluyendo los extremos del rango de 16 bits con signo. Cada valor se aplica por 10 ns y luego se observa el resultado extendido.

Resultados:



Fase 2 Datapath Completo (instrucciones tipo R y tipo I):

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:

Para conectar componentes:

- PCin, PCnext (32 bits): Estos para las conexiones del ciclo fetch.
- OpCode (6 bits): Esto para el código de operación de la instrucción.
- Rs, Rt, Rd (5 bits): Estos para los registros fuente y destino.
- Funct (6 bits): Esto para las instrucciones tipo R.
- Señales de control (1 bit cada una): MemToReg, MemWrite, RegWrite,

RegDst, ALUSrc, Branch, ZF, Br_AND_ZF, MemRead.

- ALUOp (3 bits): Esto para indicar el tipo de operación a realizar.
- Read1, Read2, ALURes, ReadMem, WriteDataBr, OP2 (32 bits): Estos para mover datos entre modulos.
- Extended, Shifted, AddRes (32 bits): Esto para el anejo de instrucciones tipo I y saltos.
- AluCtrl (3 bits): Esto para el control de la ALU.
- WriteReg (5 bits): Esto para la posición de escritura del banco de registros.

Para los buffers:

Todos estos cables segmentan variables. Primero, al Cable_combinado(numero) asignamos diferentes variables concatenadas, y es utilizado como entrada en el buffer. Luego en Cable_salida(numero) separamos para asignar los bits concretos a sus variables iguales solo que con el número del buffer para identificar (variable_B(número del buffer), este lo utilizamos como salida del buffer.

- Cable_combinado1, Cable_salida1 (64 bits)
- Cable_combinado2, Cable_salida2 (154 bits)
- Cable_combinado3, Cable_salida3 (139 bits)
- Cable_combinado4, Cable_salida4 (71 bits)

Aquí también declaramos las variables salidas del buffer, estas son de igual tamaño que las originales, pero con el característico nombre: (variable_B(número del buffer). Ejemplo: WriteReg_B3, este es WriteReg pero ya habiendo pasado por

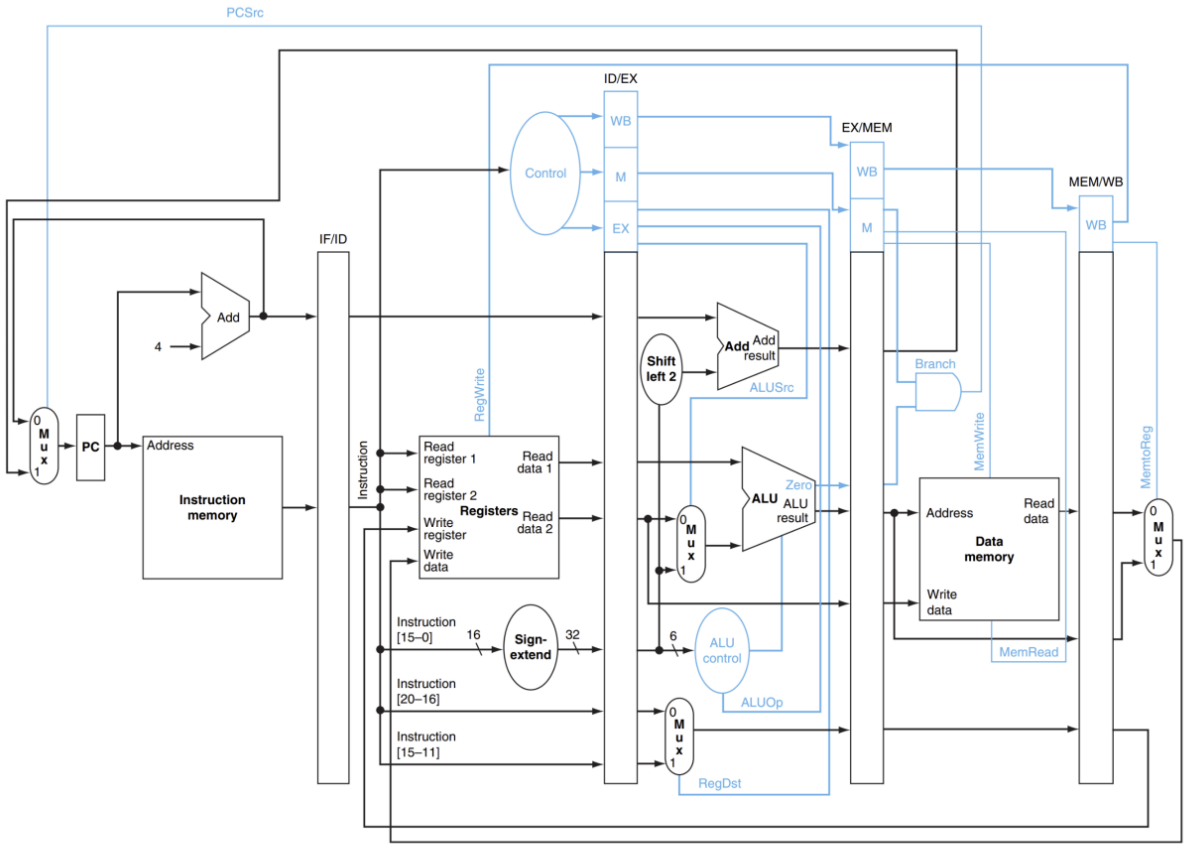


Figure 1: MIPS Datapath with Interstage Registers.

el buffer número 3.
Aquí instanciamos los distintos componentes que obedezcan el siguiente gráfico:

Los módulos que se trabajan aquí son:

- PC
- ADD4
- Mux2to1Param
- MemI
- UnidadDeControl
- BancoReg
- SignExtend
- ALUControl
- ALU
- ShiftLeft2
- AdderBranch
- MemDatos

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

```
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;
    wire [4:0]  Rs;
    wire [4:0]  Rt;
    wire [4:0]  Rd;
    wire [5:0]  Funct;

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

    // Más buses internos
    wire [31:0] Read1, Read2, ALURes, ReadMem, WriteDataBr, OP2, Extended, Shifted, AddRes;
    wire [2:0]  AluCtrl;
    wire [4:0]  WriteReg;

    // Cables de buffer
    wire [63:0] Cable_combinado1, Cable_salida1;
    wire [153:0] Cable_combinado2, Cable_salida2;
    wire [138:0] Cable_combinado3, Cable_salida3;
    wire [70:0] Cable_combinado4, Cable_salida4;

    // Señales de los buffers
    // Buffer 1
    wire [31:0] Instr_B1, PCnext_B1;
    // Buffer 2
    wire MemRead_B2, MemToReg_B2, MemWrite_B2, RegWrite_B2, RegDst_B2, ALUSrc_B2, Branch_B2;
    wire [2:0] ALUOp_B2;
    wire [31:0] Read1_B2, Read2_B2, Extended_B2, PCnext_B2;
    wire [4:0] Rt_B2, Rd_B2;
    wire [5:0] Funct_B2;
    // Buffer 3
    wire [31:0] PCnext_B3;
    wire [4:0] WriteReg_B3;
    wire [31:0] ALURes_B3, AddRes_B3, Read2_B3;
    wire ZF_B3, MemRead_B3, MemWrite_B3, MemToReg_B3, RegWrite_B3, Branch_B3;
    // Buffer 4
    wire [31:0] ReadMem_B4, ALURes_B4;
    wire MemToReg_B4, RegWrite_B4;
    wire [4:0] WriteReg_B4;
```

```
        // 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 #(32) MUXPC (
            .Entrada0(PCnext_B3),
            .Entrada1(AddRes_B3),
            .Sel(Br_AND_ZF),
            .Salida(PCin)
        );

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


    // Buffer uno IF/ID
    assign Cable_combinado1 = {Instr, PCnext};
    BF #(64) B1 (
        .IN(Cable_combinado1),
        .CLK(Clk),
        .OUT(Cable_salida1)
    );

    // Separamos las señales en el buffer 1
    assign Instr_B1   = Cable_salida1[63:32];
    assign PCnext_B1  = Cable_salida1[31:0];

    // Asignamos las variables de instruccion
    assign OpCode = Instr_B1[31:26];
    assign Rs     = Instr_B1[25:21];
    assign Rt     = Instr_B1[20:16];
    assign Rd     = Instr_B1[15:11];
    assign Funct  = Instr_B1[5:0];

    // Control
    UnidadDeControl UC (
        .OpCode(OpCode),
        .MemRead(MemRead),
        .MemToReg(MemToReg),
        .MemWrite(MemWrite),
        .ALUOp(ALUOp),
        .RegWrite(RegWrite),
        .RegDst(RegDst),
        .ALUSrc(ALUSrc),
        .Branch(Branch)
    );

    // Banco de registros
    BancoReg BR (
        .Clk(Clk),
        .RegEn(RegWrite_B4),
        .ReadReg1(Rs),
        .ReadReg2(Rt),
        .WriteReg(WriteReg_B4),
        .WriteData(WriteDataBr),
        .ReadData1(Read1),
        .ReadData2(Read2)
    );

    // Extensor de signo
    SignExtend SE (
        .Imm16(Instr_B1[15:0]),
        .Imm32(Extended)
    );

    // Buffer dos ID/EX
    assign Cable_combinado2 = {MemRead, MemToReg, MemWrite, RegWrite, RegDst, ALUSrc, Branch, ALUOp, Read1, Read2, Extended, Rt, Rd, PCnext_B1, Funct};
    BF #(154) B2 (
        .IN(Cable_combinado2),
        .CLK(Clk),
        .OUT(Cable_salida2)
    );
```

```
// Separamos las señales en B2
assign MemRead_B2      = Cable_salida2[153];
assign MemToReg_B2     = Cable_salida2[152];
assign MemWrite_B2     = Cable_salida2[151];
assign RegWrite_B2     = Cable_salida2[150];
assign RegDst_B2       = Cable_salida2[149];
assign ALUSrc_B2       = Cable_salida2[148];
assign Branch_B2       = Cable_salida2[147];

assign ALUOp_B2        = Cable_salida2[146:144];

assign Read1_B2        = Cable_salida2[143:112];
assign Read2_B2        = Cable_salida2[111:80];
assign Extended_B2     = Cable_salida2[79:48];

assign Rt_B2           = Cable_salida2[47:43];
assign Rd_B2           = Cable_salida2[42:38];

assign PCnext_B2       = Cable_salida2[37:6];
assign Funct_B2        = Cable_salida2[5:0];

// Multiplexor 2
Mux2to1Param #(5) MUXWR (
    .Entrada0(Rt_B2),
    .Entrada1(Rd_B2),
    .Sel(RegDst_B2),
    .Salida(WriteReg)
);

// ALU control
ALUControl AC (
    .ALUOp(ALUOp_B2),
    .Funct(Funct_B2),
    .ALUCtl(AluCtrl)
);

// Multiplexor 3
Mux2to1Param #(32) MUXAL (
    .Entrada0(Read2_B2),
    .Entrada1(Extended_B2),
    .Sel(ALUSrc_B2),
    .Salida(OP2)
);

// ALU
ALU alu (
    .Op1(Read1_B2),
    .Op2(OP2),
    .ALUCtl(AluCtrl),
    .Res(ALURes),
    .ZF(ZF)
);

// Shift left 2
ShiftLeft2 SL2 (
    .In(Extended_B2),
    .Out(Shifted)
);

// AdderBranch
AdderBranch AB (
    .PCplus4(PCnext_B2),
    .Shifted(Shifted),
    .PCBranch(AddRes)
);

// Buffer tres EX/MEM modificado:
assign Cable_combinado3 = {PCnext_B2, WriteReg, ALURes, ZF, AddRes, MemRead_B2, MemWrite_B2, MemToReg_B2, RegWrite_B2, Read2_B2, Branch_B2};
BF #(139) B3 (
    .IN(Cable_combinado3),
    .CLK(Clk),
    .OUT(Cable_salida3)
);

// Separamos las señales en B3
assign PCnext_B3  = Cable_salida3[138:107];
assign WriteReg_B3  = Cable_salida3[106:102];
assign ALURes_B3    = Cable_salida3[101:70];
assign ZF_B3        = Cable_salida3[69];
assign AddRes_B3    = Cable_salida3[68:37];
assign MemRead_B3   = Cable_salida3[36];
assign MemWrite_B3  = Cable_salida3[35];
assign MemToReg_B3  = Cable_salida3[34];
assign RegWrite_B3  = Cable_salida3[33];
assign Read2_B3     = Cable_salida3[32:1];
assign Branch_B3    = Cable_salida3[0];

// AND para branch y flag zero
assign Br_AND_ZF = ZF_B3 & Branch_B3;
```

```
    // MemDatos
    MemDatos MD (
        .Clk(Clk),
        .MemWrite(MemWrite_B3),
        .MemRead(MemRead_B3),
        .Address(ALURes_B3),
        .WriteData(Read2_B3),
        .ReadData(ReadMem)
    );

    // Buffer cuatro MEM/WB
    assign Cable_combinado4 = {ReadMem, MemToReg_B3, RegWrite_B3, WriteReg_B3, ALURes_B3};
    BF #(71) B4 (
        .IN(Cable_combinado4),
        .CLK(Clk),
        .OUT(Cable_salida4)
    );

    // Separamos las señales en B4
    assign ReadMem_B4   = Cable_salida4[70:39];
    assign MemToReg_B4  = Cable_salida4[38];
    assign RegWrite_B4  = Cable_salida4[37];
    assign WriteReg_B4  = Cable_salida4[36:32];
    assign ALURes_B4    = Cable_salida4[31:0];

    // Multiplexor 4
    Mux2to1Param #(32) MUXWD (
        .Entrada0(ALURes_B4),
        .Entrada1(ReadMem_B4),
        .Sel(MemToReg_B4),
        .Salida(WriteDataBr)
    );

endmodule
```

Explicaremos paso a paso cómo funciona el módulo utilizando distintas instrucciones de tipo I:

Instrucción 1: addi $5, $0, 5

Primero tenemos la instrucción addi $5, $0, 5, que en formato binario sería: 00100000000001010000000000000101.

En base a esto, lo primero que sucede es que el ciclo fetch le otorga al módulo MEMI la posición por la cual va a empezar, en este caso es cero porque sería el primer ciclo del módulo PC. Con ello MEMI lee la posición dada más las 3 siguientes, generando así la instrucción de 32 bits para todo el sistema.

Esta instrucción pasa al buffer 1, este almacena la instrucción y el valor de PC+4 por la salida del add.

En la siguiente etapa, la instrucción se divide en los respectivos cables:

- OpCode: 001000 (addi)
- Rs: 00000 ($0)
- Rt: 00101 ($5)
- Inmediato: 0000000000000101 (5)

En la Unidad de Control, al entrar OpCode, se activan las señales RegWrite y ALUSrc, y ALUOp las cuales indicarían una suma. RegDst se desactiva porque utilizaremos Rt como dirección de escritura, y con 0 activamos el multiplexor que

deja pasar esta señal.

En el banco de registros se lee el valor de Rs (0). Esto Mientras que el módulo SignExtend opera el inmediato para que pase de 16 a 32 bits.

Todas estas señales y valores pasan al buffer 2 avanzando a la etapa de ejecución.

En la etapa de ejecución, el multiplexor selecciona Rt_B2 como posición de escritura. ALUSrc_B2 está activo, haciendo que el inmediato extendido pase como segundo operando en la ALU. La ALU realiza la suma de 0 + 5 = 5. Estos resultados pasan al buffer 3 y avanzamos a la memoria.

Como esta instrucción no utiliza memoria, no se realizan operaciones en esta etapa. Los valores pasan al buffer 4. Aquí, el resultado 5 se escribe en el registro $5 porque RegWrite_B4 está activo.

Instrucción 2: beq $5, $0, #0

La segunda instrucción es beq $5, $0, #0, que en binario sería: 00010000101000000000000000000000.

Esta instrucción sigue el mismo proceso que el anterior en el fetch y decodificación, y se divide en:

- OpCode: 000100 (beq)
- Rs: 00101 ($5)
- Rt: 00000 ($0)
- Inmediato: 0000000000000000 (0)

En la Unidad de Control, se activa la señal Branch y ALUOp, activándose una operación de resta (esto para comparar).

En la etapa de ejecución, la ALU resta los valores de $5 (que ahora tiene el valor de 5) y $0 (que tiene el valor de 0): 5 - 0 = 5, esto no activaría el zflag.

En la etapa de memoria o B3, se asigna Br_AND_ZF = Branch_B3 & ZF_B3 = 1 & 0 = 0. Como es 0 y no se activa, entonces no hace realiza el salto y sigue la siguiente instrucción normalmente.

Instrucción 3: sw $10, $9, #0

La tercera instrucción es sw $10, $9, #0, que en binario sería: 10101100101010100000000000000000.

Esta se divide en:

OpCode: 101011 (sw)
Rs: 00101 ($9)
Rt: 01010 ($10)
Inmediato: 0000000000000000 (0)

En la Unidad de Control se activan las señales MemWrite y ALUSrc. Esto hace que, en la etapa de ejecución, la ALU suma el valor guardado en la posición $9 y el inmediato de 32 bits y el resultado sería la posición de memoria. Entonces $9 (100) + 0 = 100.

En la etapa B3 utilizamos el valor del registro $10(50) y se escribe en la posición 100 de la memoria de datos.

Instrucción 4: lw $11, $9, #0

La cuarta instrucción es lw $11, $9, #0, que en binario sería: 10001100101010110000000000000000.

Esta se divide en:

OpCode: 100011 (lw)
Rs: 00101 ($9)
Rt: 01011 ($11)
Inmediato: 0000000000000000 (0)

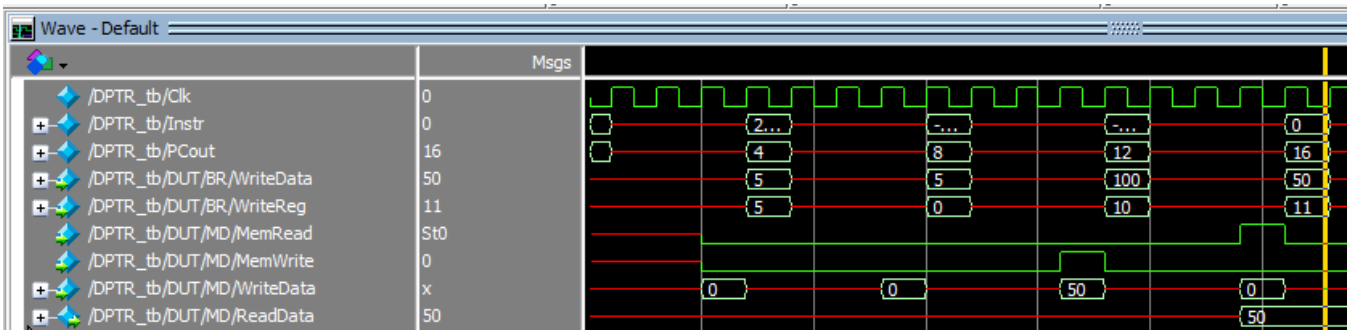
En la Unidad de Control, se activan las señales MemRead, MemToReg, RegWrite y ALUSrc. Despues la etapa de ejecución, la ALU calcula la dirección de memoria igualmente: 100 + 0 = 100. Y esta se lee en el buffer 3.

En la última etapa, en la etapa b4 de escritura, este valor de 50 leído se guarda en el $11, ya que MemToReg_B4 está activo, se utiliza el valor extraído de la memoria de datos.

De esta forma vemos como se ejecutan diferentes instrucciones de tipo I.

Graficos

En los siguientes gráficos podemos ver la simulación de las instrucciones mencionadas:



Memory Data - /DPTR_tb/DUT/BR/Mem

0	0
1	0
2	0
3	0
4	0
5	5
6	0
7	0
8	0
9	100
10	50
11	50
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0
29	0
30	0
31	0

Memory Data - /DPTR_tb/DUT/MD/Me

79	0
80	0
81	0
82	0
83	0
84	0
85	0
86	0
87	0
88	0
89	0
90	0
91	0
92	0
93	0
94	0
95	0
96	0
97	0
98	0
99	0
100	50
101	0

Hicimos un algoritmo que probará todas las instrucciones (diferente al mencionado en la introducción), este algoritmo es un bucle que suma repetidamente, guarda este valor en la memoria de datos y después guardamos lo de la memoria en el banco de registros:

Primero inicializamos la posición 10 del banco de registros en 0:
Instrucción pos 0: addi $10, $0, 0 # $10 = 0

Inicializamos nuestro contador que estará en la posición 5 guarda el valor 5:
Instrucción pos 4: addi $5, $0, 5 # $5

Guardamos en la posición 6 el valor que estaremos sumando en la iteración:
Instrucción pos 8: addi $6, $0, 12 # $6 = 12 (valor que sumamos)

Inicializamos 9 con el valor 100 porqué será nuestro base pointer hacía memoria:
Instrucción pos 12: addi $9, $0, 100 # $9 = 100 (dirección base de memoria)

Empezamos nuestro bucle:

Sumamos 12 a lo que está guardado en 10 (de inicio 0) y reescribimos el resultado en la misma posición:
Instrucción pos 16: addi $10, $10, 12

Restamos uno a nuestro contador:
Instrucción pos 20: subi $5, $5, -1 # $5 -= 1

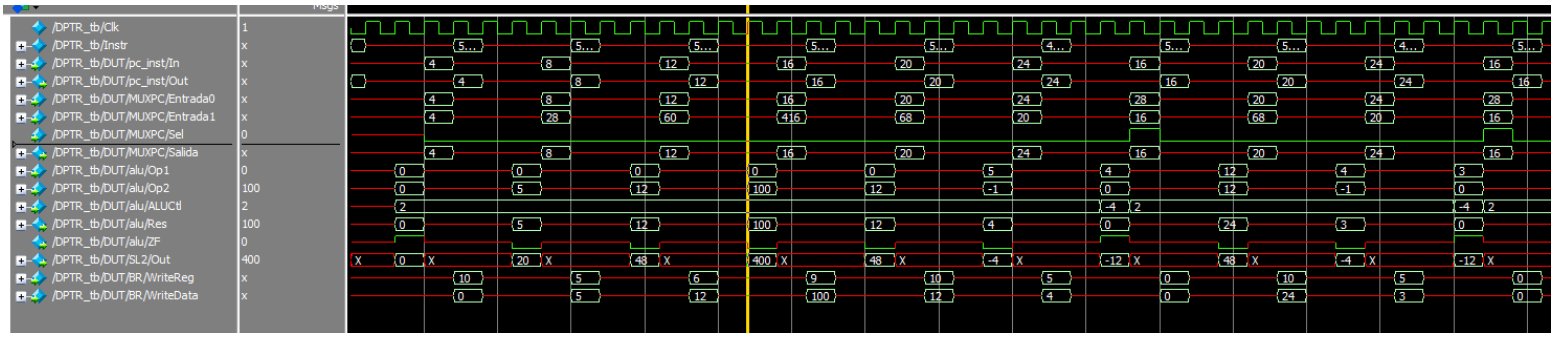
Verificamos, si nuestro contador de la posición 5 llega a 0, entonces pasamos a la instrucción pos 28, en caso contrario empezamos el bucle de nuevo:
Instrucción pos 24: bgtz $5, #-3 # si $5 > 0, salta a 16

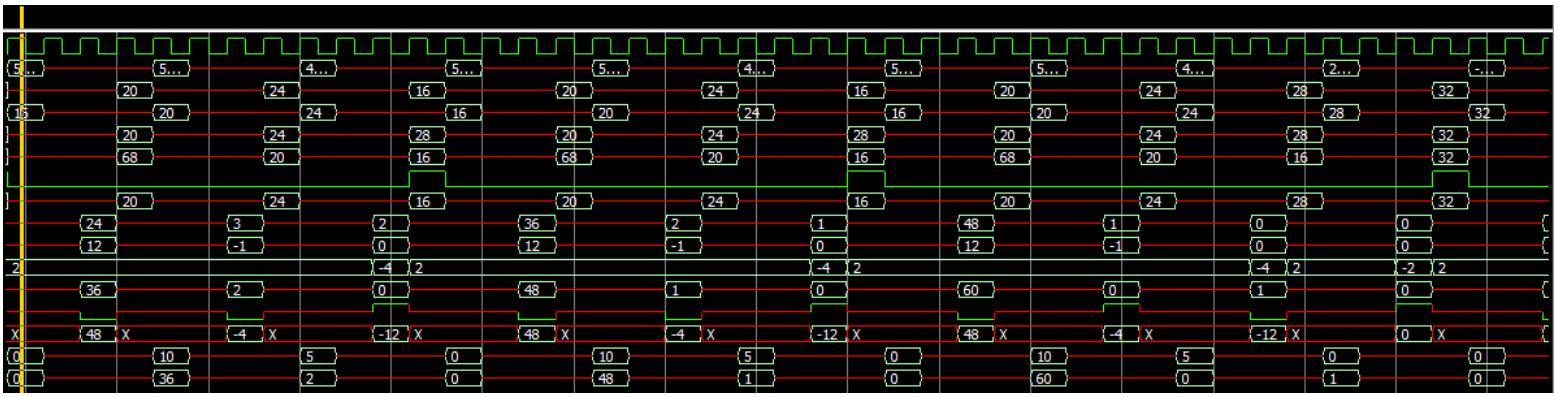
Verificamos, si nuestro contador de la posición 5 llega a 0, si es el caso entonces pasamos a la siguiente instrucción (pos 32):
Instrucción pos 28: beq $5, $0, #3 # si $5 == 0, salta a 32

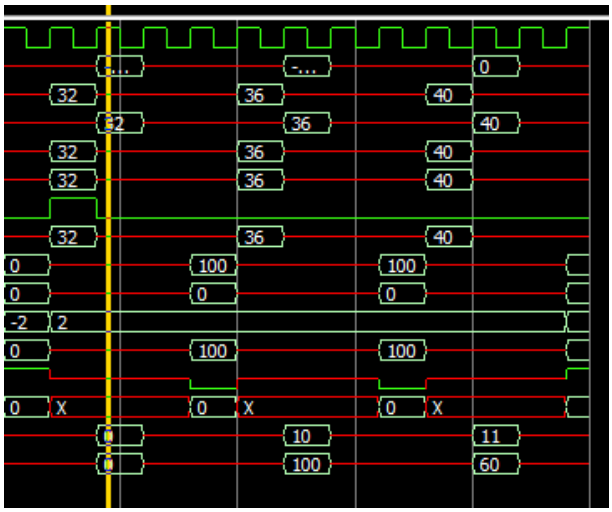
Llegado aquí tendríamos que $5 es 0, $6 es 12, $9 es 100, y $10 es 60, aquí escribimos el 60 guardado en $10 en la posición 100 en memoria:
32: sw $10, $9, #0

Ya teniendo $100= 60 en memoria, este lo cargamos al banco de registros en la posición 11. Teniendo finalmente $11 = 60.
36: lw $11, $9, #0 # $11 = Mem[$9 + 0]

En los siguientes gráficos comprobamos como fluyen las señales y como se modifican el banco de registros y la memoria:







Memory Data - /DPTR_tb/DUT/BR/Mem - Default

0	0
1	0
2	0
3	0
4	0
5	0
6	12
7	0
8	0
9	100
10	60
11	60
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0
29	0
30	0
31	0

Memory Data - /DPTR_tb/DUT/MD/Mem - Default

86	0
87	0
88	0
89	0
90	0
91	0
92	0
93	0
94	0
95	0
96	0
97	0
98	0
99	0
100	60
101	0
102	0
103	0

Conclusiones

Diego Jared Jimenez Silva

Primera fase:
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.

Segunda fase:
Para la segunda fase estuvo más complicado porque fue mucho rollo el estar buscando otros gráficos porque los de la tarea con todo respeto, pero no se veían jajaa. Los buffers añadieron un nivel de dificultad bastante considerable pero finalmente se pudo, nervioso porque se ven las líneas en rojo, pero sé que es porque es por los buffers previamente mencionados. Finalmente fue bastante cansado el tener que hacer este reporte, pero pudimos entregar algo bien documentado, espero poder estilizarlo más.

Gael Ramsés Alvarado Lomelí

Primera fase:
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 módulos de las siguientes fases fue raro, pero lo pudimos hacer con éxito.

Segunda fase:
Hasta esta segunda fase fue que realizamos el algoritmo en ensamblador, el cual fue todo un reto definir, y, sobre todo, definir las instrucciones, pero se logró, por ahora no tenemos como calarlo ya que aun nos falta la tercera fase, pero eso tendrá solución la siguiente semana, esperando que el algoritmo funcione correctamente.

Referencias

August, P. D. (2015). *Computer Architecture and Organization* . Obtenido de princeton: https://www.cs.princeton.edu/courses/archive/fall15/cos375/lectures/07-Datapath.pdf

Instrucciones procesador mips: arquitectura risc y operaciones tipo r. (s.f.). Obtenido de servernet: https://servernet.com.ar/instrucciones-de-un-procesador-mips/

Patterson, D. A. (2024). *omputer Organization and Design: The Hardware/software Interface, RISC-V Edition, Second Edition*. Beijing: Ji xie gong ye chu ban she.

Procesador mips: instrucciones, lenguaje ensamblador y funcionamiento. (s.f.). Obtenido de servernet: https://servernet.com.ar/procesador-mips/