

Proyecto Final

Diego Jared Jimenez Silva

Gael Ramses Alvarado Lomeli

220559845

220529474

18/05/2025

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, éstas 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), Immediate 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 específica 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 aritmética 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.

Pipeline y buffers:

El pipeline es una forma de procesar instrucciones, en donde no esperamos todo un ciclo de una instrucción para procesar la siguiente, si no que dividimos los procesos en diferentes etapas, haciendo que haya diferentes instrucciones en cada etapa al mismo tiempo. Con esto mejoramos el rendimiento en el procesador a través de esta segmentación, y la arquitectura MIPS hace uso de ello.

Las etapas básicas del pipeline en MIPS y las cuales implementamos son: IF, ID, EX, MEM, WB. Instruction Fetch hace eso mismo, obtiene la instrucción desde la memoria por medio del ciclo fetch. Instruction Decode decodifica la instrucción dada, lee los registros y pasa las señales de control. Execute para las operaciones aritméticas y lógicas de la ALU y administra las señales de memoria para la siguiente etapa. Memory Acces para todo lo que tenga que ver con operaciones de memoria. Write Back para escribir los resultados en los registros. Nosotros

incorporamos estos 4 en el proyecto de la misma manera y con el mismo funcionamiento en base al siguiente grafico:

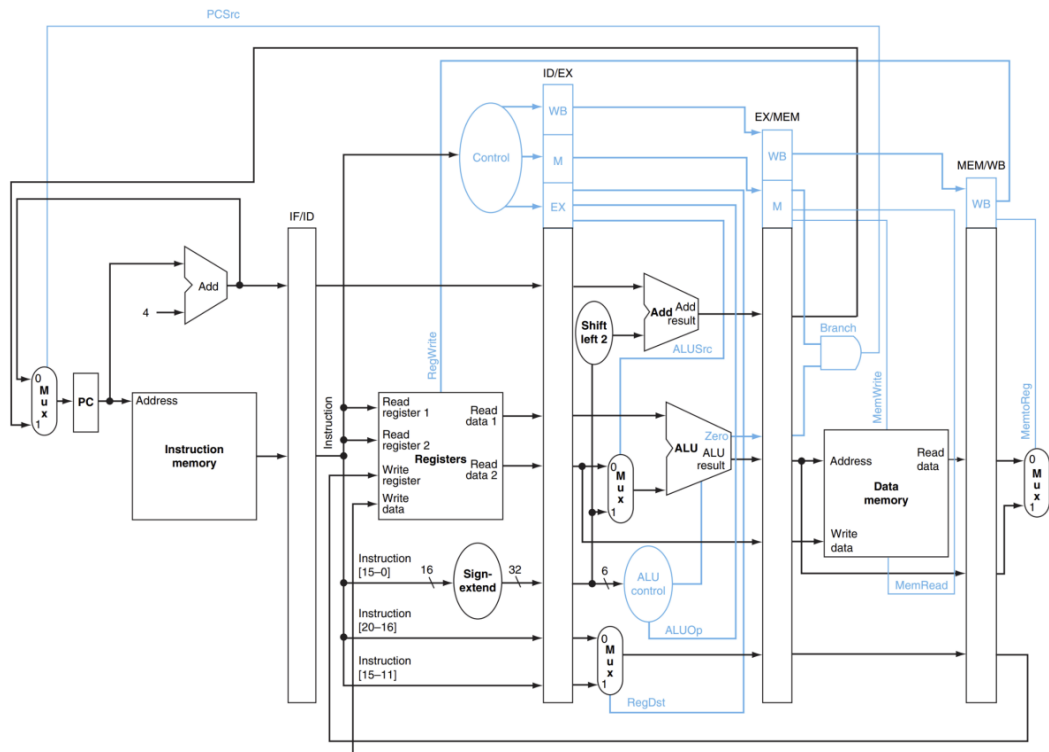


Figure 1: MIPS Datapath with Interstage Registers.

Lo bueno del pipeline es que podemos procesar múltiples instrucciones al mismo tiempo, aumenta el rendimiento del procesador y mejoramos el uso del hardware teniendo más optimización.

El funcionamiento que tiene es permitir que cada etapa trabaje de forma independiente, coordinar el flujo de datos entre las etapas que hay y resolver problemas de dependencias donde el resultado de una depende de otra, por ejemplo.

Decodificador

Conversor ASM a Binario (MIPS)

Este programa es una herramienta desarrollada en Python para convertir instrucciones en ensamblador MIPS a su representación binaria de 32 bits. Incluye una interfaz gráfica simple usando Tkinter que permite cargar archivos .ASM, convertir

su contenido a binario, visualizar el resultado y guardarlo en un archivo de texto.

Características

- Soporta instrucciones MIPS de tipo R, I y J.
- Traduce instrucciones como ADD, LW, BEQ, J, entre muchas otras.
- Soporta notación de registros por número (\$4) o por alias (\$A0, \$T1, etc.).
- Muestra errores detallados si las instrucciones no son válidas.
- Guarda la salida binaria en formato de 8 bits por línea, hasta 1000 líneas (8×1000 bits).
- Interfaz gráfica amigable construida con tkinter.

Instrucciones Soportadas

- Tipo R: ADD, SUB, SLT, OR, AND, NOP
- Tipo I: SW, LW, ADDI, SUBI, SLTI, ANDI, ORI, BEQ, BNE, BGTZ
- Tipo J: J

Requisitos

- Python 3.x
- Tkinter (ya viene incluido con la mayoría de distribuciones de Python)

Cómo usar

1. Clona o descarga este repositorio.
2. Ejecuta el script principal:
3. En la ventana que se abre:
 - o Usa "Cargar ASM" para seleccionar un archivo .ASM con instrucciones MIPS.
 - o Haz clic en "Convertir a Binario" para traducir el contenido.
 - o Revisa la salida binaria en la sección inferior.
 - o Guarda la salida si lo deseas con "Guardar Binario".

Formato del archivo ASM

Cada línea debe contener una única instrucción MIPS válida. Se permite usar

[illegible]

\$7 00000000000000000000000000000000
 \$8 00000000000000000000000000000010
 \$9 00000000000000000000000000001100100
 \$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 máximo (hasta 100)
- \$4 = dirección base para guardar datos
- \$5 = contador de números primos (también servirá para calcular offsets)
- \$6 = divisor actual
- \$7 = es primo (bandera que indica si un número es primo o no)
- \$11 = registro para cálculo del resto (simula módulo)
- \$12 = dirección de escritura
- \$13, \$14 = registros temporales (comparaciones y cálculos)

El algoritmo funciona de la siguiente manera:

1. Se inicializa el número actual en 2 y el límite en 100.
2. Se compara número_actual con número_máximo. Si ya se pasó del límite, el algoritmo termina.
3. Se inicializa divisor = 2 y es_primo = 1.
4. Se verifica si el divisor aún es menor que el número actual.
5. Se copia número_actual a un registro de trabajo para simular el módulo (restas sucesivas).
6. Se realizan restas hasta que el resto sea menor que el divisor.
7. Si el resto es exactamente 0, el número no es primo y se establece es_primo = 0.
8. Si el número es primo, se guarda en memoria.
9. Se incrementa número_actual y se repite el ciclo.

Pseudocódigo:

```
Inicio:
    numero_actual = 2                // Comenzamos desde el primer primo (2)
    limite = 100                    // Límite superior (modificable)
    contador_primos = 0              // Cuenta cuántos primos se han encontrado
    direccion_base = 0              // Dirección base en memoria para almacenar primos

    Mientras numero_actual <= limite:
        es_primo = VerificarSiEsPrimo(numero_actual)

        Si es_primo es Verdadero:
            direccion_almacenamiento = direccion_base + (contador_primos * 4) // Offset de 4 bytes
            Memoria[direccion_almacenamiento] = numero_actual // Guardar primo en memoria
            contador_primos = contador_primos + 1

            numero_actual = numero_actual + 1 // Siguiendo número

    Fin del Programa (Bucle Infinito)

// Función para verificar si un número es primo
Función VerificarSiEsPrimo(n):
    Si n < 2:
        Retornar Falso

    divisor = 2
    Mientras divisor < n:
        resto = n
        Mientras resto >= divisor: // Calcula resto por restas sucesivas
            resto = resto - divisor

        Si resto == 0: // Si el resto es 0, no es primo
            Retornar Falso

        divisor = divisor + 1

    Retornar Verdadero // Si ningún divisor lo divide, es primo
```

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 100.

Instrucción 2: ADDI \$4, \$0, 0: \$4 va a ser la dirección base de memoria (empezamos en 0).

Instrucción 3: ADDI \$5, \$0, 0: Inicializamos el contador de primos en \$5 con 0.

Instrucción 4: SUB \$13, \$8, \$9: Calculamos número_actual - número_máximo para comprobar si superamos el límite.

Instrucción 5: BGTZ \$13, 21: Si la resta es positiva, hemos superado el límite y saltamos al final.

Instrucción 6: ADDI \$6, \$0, 2: Empezamos probando con el divisor 2.

Instrucción 7: ADDI \$7, \$0, 1: Asumimos que el número sí es primo (bandera en 1).

Instrucción 8: SLT \$13, \$6, \$8: Verificamos si el divisor es menor que el número actual.

Instrucción 9: BEQ \$13, \$0, 9: Si el divisor ya no es menor, salimos del ciclo de divisores.

Instrucción 10: ADD \$11, \$8, \$0: Copiamos el número actual a \$11, lo vamos a usar para calcular el resto.

Instrucción 11: SUB \$11, \$11, \$6: Le restamos el divisor al número. Esto simula la operación módulo.

Instrucción 12: SLT \$14, \$11, \$6: Comprobamos si el resto es menor que el divisor.

Instrucción 13: BEQ \$14, \$0, -3: Si el resto todavía es mayor o igual, seguimos restando.

Instrucción 14: BEQ \$11, \$0, 1: Si el resto es exactamente 0, el número es divisible, ósea no es primo.

Instrucción 15: J 17: Si hay resto, el número no es divisible por el divisor actual.

Instrucción 16: ADDI \$7, \$0, 0: Marcamos el número como NO primo.

Instrucción 17: ADDI \$6, \$6, 1: Aumentamos el divisor para probar con el siguiente.

Instrucción 18: J 8: Volvemos a probar con el nuevo divisor.

Instrucción 19: BEQ \$7, \$0, 5: Si no es primo, saltamos la escritura en memoria.

Instrucción 20: ADD \$14, \$5, \$5: Calculamos el offset (multiplicamos por 4).

Instrucción 21: ADD \$14, \$14, \$14: Completamos la multiplicación por 4.

Instrucción 22: ADD \$12, \$4, \$14: Calculamos la dirección de memoria.

Instrucción 23: SW \$8, 0(\$12): Guardamos el número primo en memoria.

Instrucción 24: ADDI \$5, \$5, 1: Aumentamos el contador de primos.

Instrucción 25: ADDI \$8, \$8, 1: Vamos al siguiente número a revisar.

Instrucción 26: J 4: Volvemos al inicio del ciclo principal.

Instrucción 27: J 27: Bucle infinito al finalizar (marca el fin del programa).

Código ensamblador completo:

Aquí inicializamos los registros que utilizaremos y configuramos valores iniciales:

```
ADDI $8, $0, 2    # $8 = Número actual (comienza en 2)
ADDI $9, $0, 100  # $9 = Límite superior (100)
ADDI $4, $0, 0    # $4 = Dirección base de memoria (primer primo en 0x0)
ADDI $5, $0, 0    # $5 = Contador de primos (inicia en 0)
```

Bucle principal que verifica cada número hasta el límite:

```
SUB $13, $8, $9    # Calcula $13 = $8 - $9 (número actual - límite)
BGTZ $13, 21       # Si $8 > 100 → salta a SALIDA (offset 21)
ADDI $6, $0, 2     # Inicializa divisor ($6 = 2)
ADDI $7, $0, 1     # Bandera "es_primo" ($7 = 1)
```

Verificación de divisores:

```
SLT $13, $6, $8    # ¿Divisor ($6) < Número ($8)? ($13 = 1 si sí)
BEQ $13, $0, 9      # Si no, salta a almacenar primo
ADD $11, $8, $0     # Copia el número a $11 para calcular el resto
```

Bucle que calcula del resto mediante restas sucesivas:

```
SUB $11, $11, $6    # Resta el divisor al número ($11 -= $6)
SLT $14, $11, $6    # ¿Resto ($11) < divisor ($6)? ($14 = 1 si sí)
BEQ $14, $0, -3     # Si no, repite la resta (vuelve a BucleResto)
BEQ $11, $0, 1      # Si resto == 0 → no es primo
J 17                # Salta a incrementar divisor
ADDI $7, $0, 0      # Marca como no primo ($7 = 0)
```

Incrementa el divisor y aplica verificaciones, si el número no es primo, no se guarda:

```
ADDI $6, $6, 1      # Incrementa divisor ($6++)
J 8                 # Vuelve a VerificarDivisor
BEQ $7, $0, 5       # Si no es primo, saltamos la escritura en memoria.
```

Almacenamiento de primos calculado dirección de memoria:

```
ADD $14, $5, $5     # Offset = $5 * 2 (primer paso para *4)
ADD $14, $14, $14    # Offset = $5 * 4 (cada primo ocupa 4 bytes)
ADD $12, $4, $14     # Dirección = base ($4) + offset
```

```
SW $8, 0($12)    # Guarda el primo en memoria
ADDI $5, $5, 1    # Incrementa contador de primos ($5++)
```

Avanza al siguiente número incrementando el numero actual y repitiendo el proceso hasta generar un bucle infinito de nop, J salida, nop, J salida para tener una terminación controlada.

```
ADDI $8, $8, 1    # Incrementa número actual ($8++)
J 4               # Vuelve a LoopPrincipal
J 27              # Bucle infinito al finalizar (SALIDA)
```

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  # Límite superior (100)
```

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

Este algoritmo será puesto a prueba 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.

```
`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
    end
endmodule
```

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
```

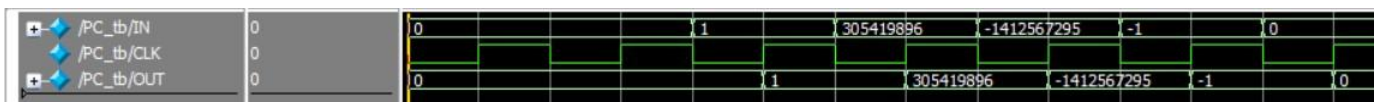
El testbench para probar que funcione correctamente este módulo es:

```
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:

/ADD4_tb/A	0	0	1	4	-4	-1
/ADD4_tb/RES	4	4	5	8	0	3

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:

MEMI_tb/DR	xxxxxxxxxxxx...	00000000000000000000000000000000	00000000000000000000000000000000	00000000000000000000000000000000
MEMI_tb/INS	xxxxxxxxxxxx...	00000000111101001101000000000000	00000000000000000000000000000000	00000010100010011010000000000000

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;
```

```

        MemRead = 1'b0;
        ALUSrc = 1'b0;

        case (OpCode)
            6'b000000: begin // Instrucción 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;
    end
endmodule

```

```

// 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;

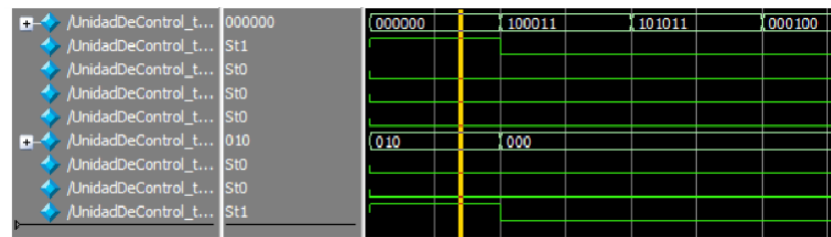
end

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 entradas 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
endmodule
```

```

// 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:

/BancoReg_tb/dk	-No Data-								
/BancoReg_tb/RegEn	-No Data-								
+ /BancoReg_tb/Rea...	-No Data-	0				2			
+ /BancoReg_tb/Rea...	-No Data-	1				3			
+ /BancoReg_tb/Writ...	-No Data-	2						3	
+ /BancoReg_tb/Writ...	-No Data-	-559038737						-889275714	
+ /BancoReg_tb/Rea...	-No Data-	0				-559038737			
+ /BancoReg_tb/Rea...	-No Data-	0						-889275714	

ALU control:

La ALU control está conformada por 2 entradas y 1 sola salida. Las entradas 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.

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

```
// 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
```

```

`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:

/ALUControl_tb/AL...	010	010					000	
/ALUControl_tb/funct	100000	100000	100010	100100	100101	101010	000000	111111 100000
/ALUControl_tb/AL...	010	010	110	000	001	111	011	

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)
    );
endmodule
```

```
// Estímulos
initial begin
```

```
// Caso AND
ALUCT1 = 3'b000;
OP1    = 32'b11110000111100001111000011110000;
OP2    = 32'b00001111000011110000111100001111;
#10;
```

```
// Caso OR
ALUCT1 = 3'b001;
OP1    = 32'b11110000111100001111000011110000;
OP2    = 32'b00001111000011110000111100001111;
#10;
```

```
// Caso ADD
ALUCT1 = 3'b010;
OP1    = 32'b00000000000000000000000000001010;
OP2    = 32'b00000000000000000000000000001111;
#10;
```

```
// Caso SUB no cero
ALUCT1 = 3'b110;
OP1    = 32'b00000000000000000000000000001010;
OP2    = 32'b0000000000000000000000000000101;
#10;
```

```
// Caso SUB a cero (ZF=1)
ALUCT1 = 3'b110;
OP1    = 32'b00000000000000000000000000000111;
OP2    = 32'b00000000000000000000000000000111;
#10;
```

```
// Caso SLT verdadero
ALUCT1 = 3'b111;
OP1    = 32'b00000000000000000000000000000011;
OP2    = 32'b00000000000000000000000000000100;
#10;
```

```
// Caso SLT falso
ALUCT1 = 3'b111;
OP1    = 32'b00000000000000000000000000000100;
OP2    = 32'b00000000000000000000000000000010;
#10;
```

```
// Caso NOP
ALUCT1 = 3'b011;
OP1    = 32'b0000000000000000000000000000111011;
OP2    = 32'b00000000000000000000000000001100100;
#10;
```

```
// Caso default (invalido)
ALUCT1 = 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:

/ALU_tb/OP1	-252645136	-252645136	10	20	7	3	9	123	1	
/ALU_tb/OP2	252645135	252645135	15	5	7	8	2	456	1	
/ALU_tb/ALUCtl	0	0	1	2	-2		-1	3	-4	
/ALU_tb/Res	0	0	-1	25	15	0	1	0		
/ALU_tb/ZF	1									

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;

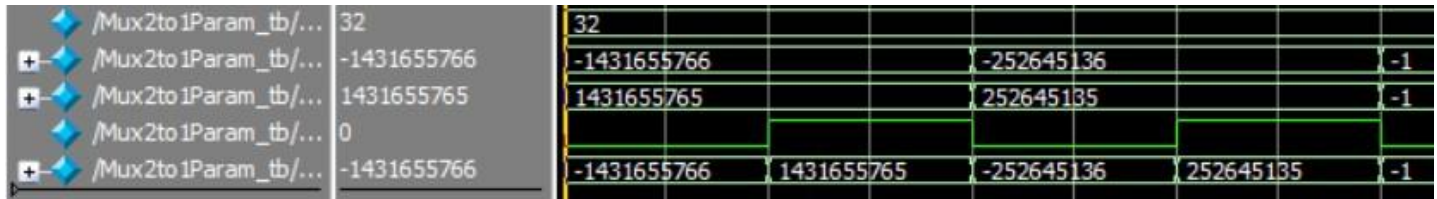
        sel      = 1'b1;
        #10;

        // 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:



/Mux2to1Param_tb/...	32	32						
/Mux2to1Param_tb/...	-1431655766	-1431655766			-252645136			-1
/Mux2to1Param_tb/...	1431655765	1431655765			252645135			-1
/Mux2to1Param_tb/...	0							
/Mux2to1Param_tb/...	-1431655766	-1431655766	1431655765		-252645136	252645135		-1

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:


```

// 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 entraría 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.

/DPTR_tb/dk	0	
/DPTR_tb/Instr	...8098	32088...
/DPTR_tb/PCout	0	0
/DPTR_tb/DUT/BR/ReadReg1	15	15
/DPTR_tb/DUT/BR/ReadReg2	9	9
/DPTR_tb/DUT/BR/ReadData1	999	999
/DPTR_tb/DUT/BR/ReadData2	100	100
/DPTR_tb/DUT/alu/Res	899	899
/DPTR_tb/DUT/BR/WriteData	899	899
/DPTR_tb/DUT/BR/WriteReg	20	20

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

The screenshot displays a digital logic simulation environment. The top portion shows a waveform viewer with a time axis from 0 to 20 ns. The waveform displays various signals, including clock (clk), instruction (Instr), and register file outputs (ReadReg1, ReadReg2, ReadData1, ReadData2, WriteData, WriteReg). The bottom portion shows a Verilog code editor for a module named DPTR. The code defines the module's inputs and outputs, internal buses, and control signals.

```

1 // Diego Jared Jimenez Silva
2 // Gael Ramses Alvarado Lomeli
3
4 module DPTR (
5     input wire      clk,
6     output wire [31:0] Instr,
7     output wire [31:0] PCout
8 );
9 // Buses internos
10 wire [31:0] PCin, PCnext;
11 wire [5:0]  OpCode = Instr[31:26];
12 wire [4:0]  rs     = Instr[25:21];
13 wire [4:0]  rt     = Instr[20:16];
14 wire [4:0]  rd     = Instr[15:11];
15 wire [5:0]  funct  = Instr[5:0];
16
17 // Señales de control
18 wire MemToReg, MemToWrite, RegWrite, RegDst, MemRead, ALUSrc, Branch,
19     ZF, Br_AND_ZF;
20 wire [2:0]  ALUOp;
21
22 // Más buses internos
  
```

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 específica 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, siendo una resta para beq y bne, y bgtz para activar una operación en la ALU que compruebe si el número 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
    end
```

```

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;

```


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)  
    );  
endmodule
```

```

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

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

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

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

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

    // 8) Default (ALUOp=011)
    ALUOp = 3'b011; Funct = 6'bxxxxxxx; #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:

+ /tb_ALUControl/ALUOp	000	000	001	010						100	101	111	110
+ /tb_ALUControl/Funct	xxxxxx			100000	100010	100100	100101	101010	000000				
+ /tb_ALUControl/ALUCtl	010	010	110	010	110	000	001	111	011	000	001	111	100

011

011

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 activar 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'b00000000000000000000000000000100; // 8
        Op2 = 32'b00000000000000000000000000000101; // 5
        #10;

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

```

```

// 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'b00000000000000000000000000000000; // 0
Op2     = 32'b00000000000000000000000000000000; // 0
#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:

Msgs					
/tb_ALU/Op1	10101010101010...	10101010101010101010101010101010			00001111000011110000111100001111
/tb_ALU/Op2	11001100110011...	11001100110011001100110011001100			00110011001100110011001100110011
/tb_ALU/ALUCtl	000	000			001
/tb_ALU/Res	10001000100010...	10001000100010001000100010001000			00111111001111110011111100111111
/tb_ALU/ZF	St0				
/tb_ALU/Op1	10101010101010...	00000000000000000000000000000000			00000000000000000000000000000000
/tb_ALU/Op2	11001100110011...	00000000000000000000000000000000			00000000000000000000000000000000
/tb_ALU/ALUCtl	000	010			110
/tb_ALU/Res	10001000100010...	00000000000000000000000000000000			00000000000000000000000000000000
/tb_ALU/ZF	St0				
/tb_ALU/Op1	10101010101010...	00000000000000000000000000000000			11111111111111111111111111111111
/tb_ALU/Op2	11001100110011...	00000000000000000000000000000000			00000000000000000000000000000000
/tb_ALU/ALUCtl	000	111			100
/tb_ALU/Res	10001000100010...	00000000000000000000000000000000			00000000000000000000000000000000
/tb_ALU/ZF	St0				
		00000000000000000000000000000000			00000000000000000000000000000000
					011
		00000000000000000000000000000000			00000000000000000000000000000000

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.

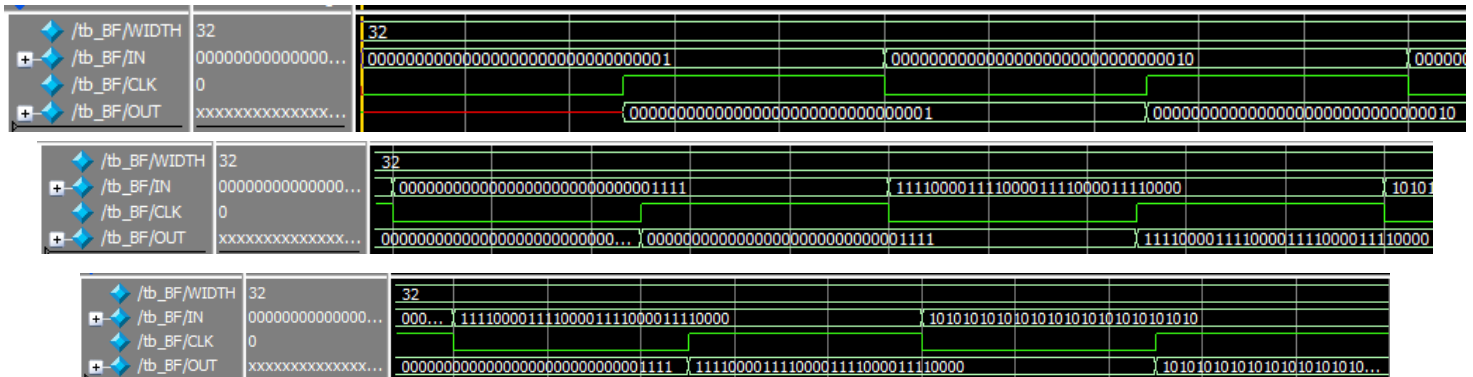
```
module BF #(parameter WIDTH = 32) (  
    input [WIDTH-1:0] IN,  
    input CLK,  
    output reg [WIDTH-1:0] OUT  
);  
  
always @(posedge CLK) begin // Cada positivo en el reloj  
    OUT <= IN; // Utilizamos <= para las distintas asignaciones en paralelo  
end  
  
endmodule
```

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'b00000000000000000000000000000111; // 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'b0000000000000000000000000000100; // 4
        Shifted = 32'b00000000000000000000000000001000; // 8
        #5;

        // 2) Suma con diferentes bits
        PCplus4 = 32'b000000000000000000000000000010010; // 18
        Shifted = 32'b000000000000000000000000000000110; // 6
        #5;

        // 3) Suma con carry a bit alto
        PCplus4 = 32'b111111111111111111111111111111; // -1 (2's complement)
        Shifted = 32'b000000000000000000000000000000001; // 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:

+ /tb_AdderBranch/PCplus4	4	4	18	-1	-143155766
+ /tb_AdderBranch/Shifted	8	8	6	1	1431655765
+ /tb_AdderBranch/PCBranch	12	12	24	0	-1

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 hacia 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'b0000000000000000000000000000001111; // 15 -> 60
        #5;
        In = 32'b11110000111100001111000011110000; // prueba de patrón
        #5;
        In = 32'b10101010101010101010101010101010; // prueba de alternancia
        #5;

        // Fin de la simulación
        $finish;
    end

endmodule
```

Resultados:

The screenshot shows the Logic Analyzer tool with the following data:

Signal	Value
/tb_ShiftLeft2/In	0000000000000000...
/tb_ShiftLeft2/Out	0000000000000000...

The timing diagram shows the relationship between the input and output signals over time. The input signal is a square wave that transitions from 0 to 1 at approximately 10 ns. The output signal is a square wave that transitions from 0 to 1 at approximately 20 ns. The timing diagram shows the relationship between the input and output signals over time.

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 Imm16 = 16'b0000_0000_1001_0001
    #10 Imm16 = 16'b1000_0000_0000_0000
    #10 Imm16 = 16'b1111_1111_1111_1010
    #10 Imm16 = 16'b0111_1111_1111_1111
    #10 Imm16 = 16'b1000_0000_0000_0001

    #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:

+ /tb_SignExtend/Imm16	0000000010010001	0000000000001010			0000000010010001	
+ /tb_SignExtend/Imm32	0000000000000000...	000000000000000000000000001010			0000000000000000000000000010010001	
+ /tb_SignExtend/Imm16	0000000010010001	1000000000000000			11111111111010	
+ /tb_SignExtend/Imm32	0000000000000000...	11111111111111100000000000000000			111111111111111111111111111010	
+ /tb_SignExtend/Imm16	0000000010010001	0111111111111111			10000000000000001	
+ /tb_SignExtend/Imm32	0000000000000000...	00000000000000001111111111111111			111111111111111110000000000000001	

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 módulos.
- 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 el buffer número 3.

Aquí instanciamos los distintos componentes que obedezcan el siguiente gráfico

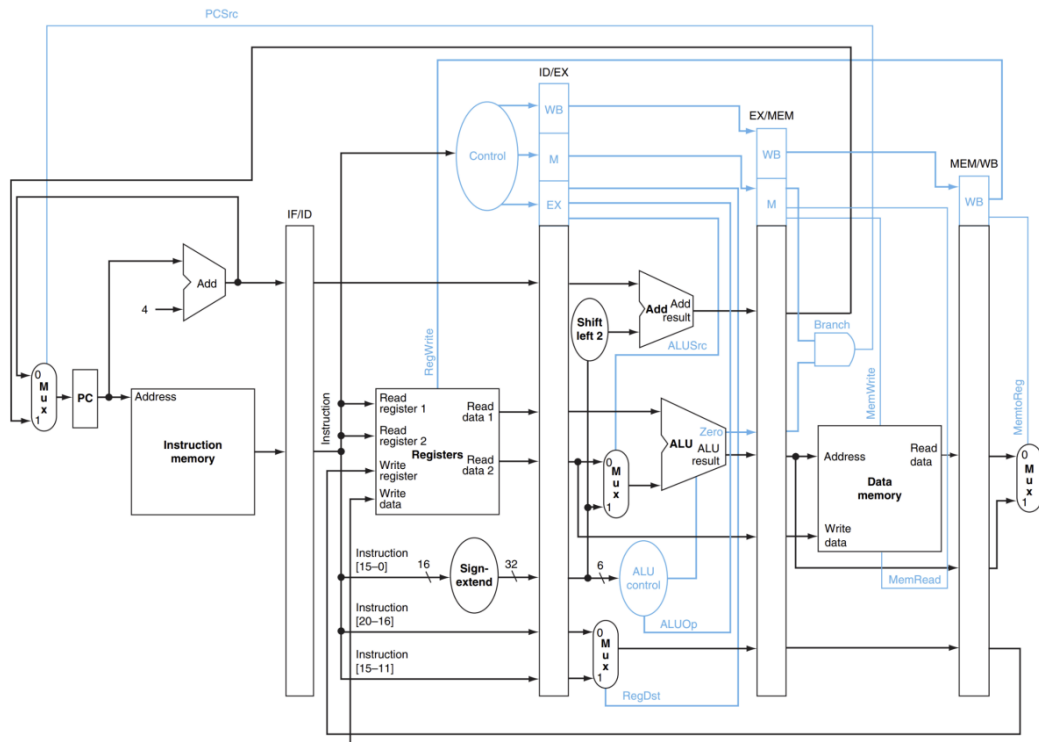


Figure 1: MIPS Datapath with Interstage Registers.

Los módulos que se trabajan aquí son:

- PC
- ADD4
- Mux2to1Param
- Meml
- 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)
);

// 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)

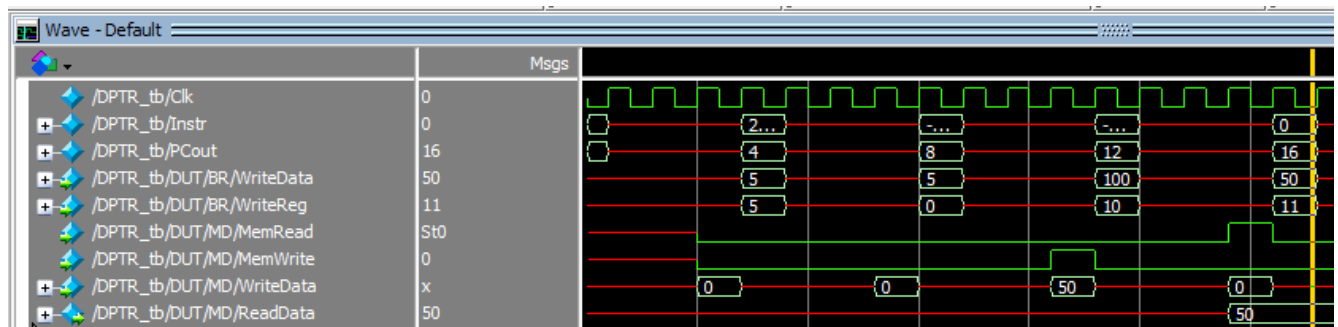
En la Unidad de Control, se activan las señales MemRead, MemToReg, RegWrite y ALUSrc. Después la etapa de ejecución, la ALU calcula la dirección de memoria igualmente: $100 + 0 = 100$. Y esta se lee en el buffer 3.

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/Mem

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 porque 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**

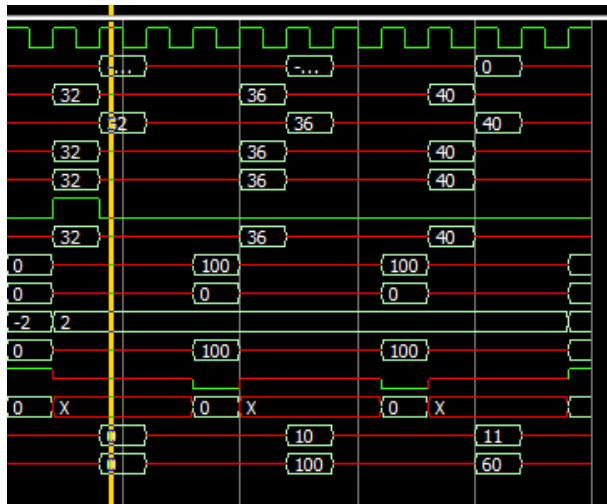
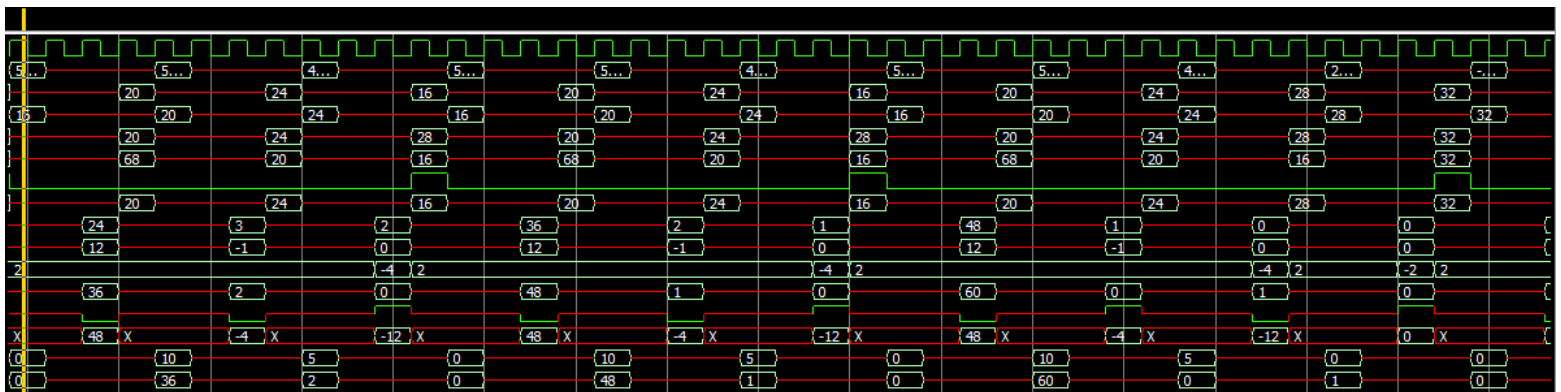
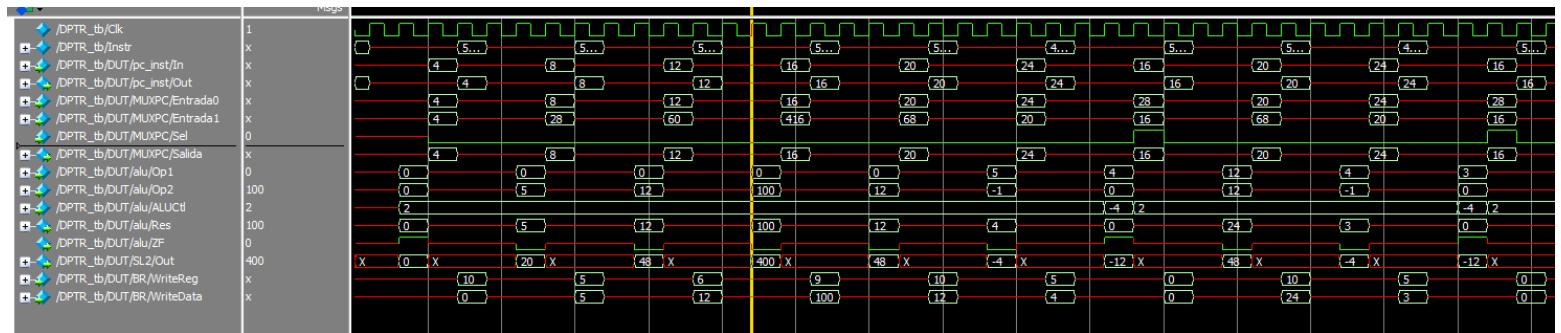
Llegado aquí tendríamos que \$5 es 0, \$6 es 12, \$9 es 100, y \$10 es 60, aquí escribimos el 60 guardado en \$10 en la posición 100 en memoria:

- **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:



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

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
104	0

Fase 3

Módulo modificado:

Unidad de control:

En la unidad de control utilizamos 000010 como nueva entrada que habilita la señal de Jump para dicha operación. Igualmente, en el módulo deshabilitamos esta señal de forma predeterminada para las demás operaciones.

```
        ALUOp = 3'b110;
    end
    6'b000010: begin // J jump
        Jump = 1'b1;
    end
endcase
end
endmodule
```

Modulo nuevo:

Jump Adrees:

Este módulo es similar al módulo shift left 2. El módulo ShiftLeft2 está conformado por 2 entradas y 1 salida. Las entradas son las siguientes: Pcplus4 (32 bits) y JumpField (32bits). La salida es la siguiente: JumpAddr (32 bits).

El módulo JumpAddr funciona para concatenar los 4 bits más significativos de PC+4 para que se haga el salto en la región de memoria donde nos encontramos, 26 bits del campo de dirección de la instrucción y 2 bits a la izquierda para multiplicar por 4.

En la funcionalidad del módulo, utilizamos un assign para que JumpAddr tome inmediatamente el valor calculado.

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

```
module JumpAddress(
    input wire [31:0] PCplus4,
    input wire [25:0] JumpField,
    output wire [31:0] JumpAddr
);

    // La dirección de salto para instrucciones J se calcula:
    // - Tomando los 4 bits más significativos de PC+4 (bits 31-28)
    // - Concatenándolos con los 26 bits del campo de dirección de la instrucción
    // - Desplazando 2 bits a la izquierda (multiplicando por 4)
    assign JumpAddr = {PCplus4[31:28], JumpField, 2'b00};

endmodule
```

Testbench:

```
`timescale 1ns / 1ps

module JumpAddress_tb;
    reg [31:0] PCplus4;
    reg [25:0] JumpField;
    wire [31:0] JumpAddr;

    // Instancia
    JumpAddress uut (
        .PCplus4(PCplus4),
        .JumpField(JumpField),
        .JumpAddr(JumpAddr)
    );

    initial begin
        // Caso 1
        PCplus4 = 32'd4;
        JumpField = 26'd10;
        #10;

        // Caso 2
        PCplus4 = 32'd16;
        JumpField = 26'd50;
        #10;

        // Caso 3
        PCplus4 = 32'd32;
        JumpField = 26'd99;
        #10;

        // Fin de la simulación
        #10;
        $finish;
    end
endmodule
```

Aquí utilizamos distintos valores. Podemos ver que el JumpAddr es el JumpField pero multiplicado por 4, esto porque las instrucciones están posicionadas en múltiplos de 4. Por ello si vamos a la instrucción 10, está se encontrará en la posición 40, esto lo podemos ver en el gráfico.

Cuando utilizamos valores más grandes, JumpField no completa los 32 bits de una dirección, por ello completamos con los bits más significativos de Pcpus 4, con esto nos posicionamos en esa región de la memoria para ahora si hacer la asignación.

Resultados:

	Msgs							
+ /JumpAddress_tb/P...	4	4	16	32				
+ /JumpAddress_tb/J...	10	10	50	99				
+ /JumpAddress_tb/J...	40	40	200	396				

Fase 3 Datapath Completo (instrucciones tipo R, tipo I, tipo J):

El módulo DPTR (o data path) 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 representa la implementación de una CPU simplificada tipo MIPS, capaz de ejecutar instrucciones tipo R, I y J. En el módulo DPTR definimos varios buses internos para interconectar sus componentes, estos son:

Para conectar componentes:

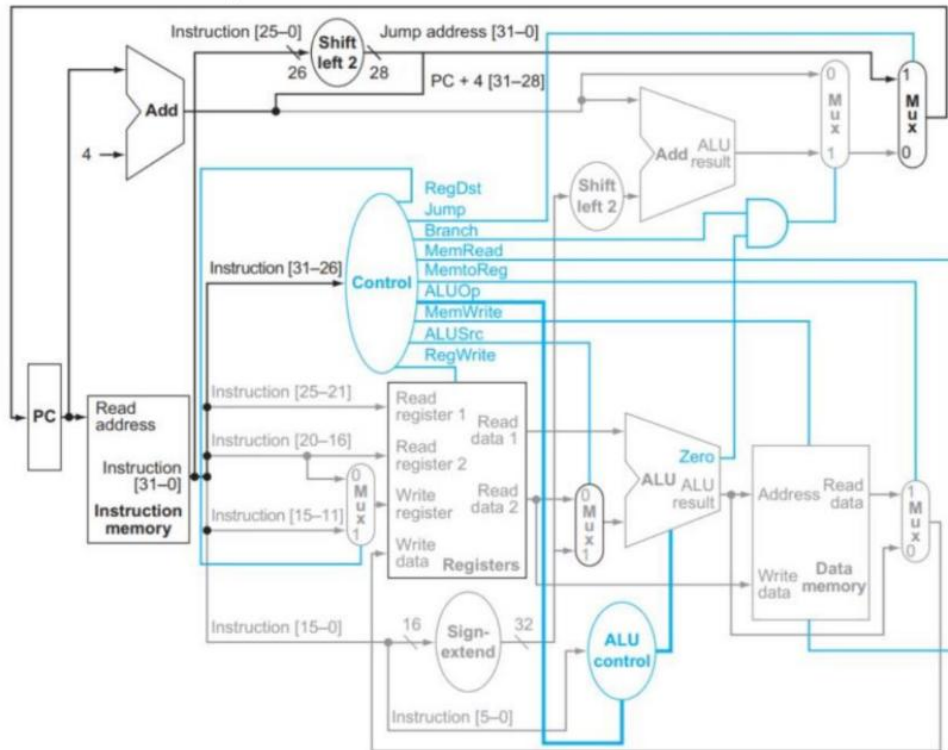
- PCin, PCnext (32 bits): Estos para las conexiones del ciclo fetch.
- OpCode (6 bits): Esto para el código de operación de la instrucción.
- Rs, Rt, Rd (5 bits): Estos para los registros fuente y destino.
- Funct (6 bits): Esto para las instrucciones tipo R.
- Señales de control (1 bit cada una): MemToReg, MemWrite, RegWrite, RegDst, ALUSrc, Branch, ZF, Br_AND_ZF, MemRead, Jump.
- 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 módulos.
- Extended, Shifted, AddRes (32 bits): Esto para el manejo 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.
- JumpField_B1 (26 bits): Campo para saltos tipo J.
- JumpAddr (32 bits): Dirección calculada para saltos tipo J.

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 (187 bits)
- Cable_combinado3, Cable_salida3 (172 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 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
- JumpAddress
- BancoReg
- SignExtend
- ALUControl
- ALU
- ShiftLeft2
- AdderBranch
- MemDatos
- BF (Buffer)

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, Jump;
    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 para Jump
    wire [25:0] JumpField_B1;
    wire [31:0] JumpAddr;

    // Cables de buffer
    wire [63:0] Cable_combinado1, Cable_salida1;
    wire [186:0] Cable_combinado2, Cable_salida2;
    wire [171:0] 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, Jump_B2;
    wire [2:0]  ALUOp_B2;
    wire [31:0] Read1_B2, Read2_B2, Extended_B2, PCnext_B2, JumpAddr_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, JumpAddr_B3;
    wire ZF_B3, MemRead_B3, MemWrite_B3, MemToReg_B3, RegWrite_B3, Branch_B3, Jump_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)
    );

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

```

// 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];
assign JumpField_B1 = Instr_B1[25:0];

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

// Dirección de salto J
JumpAddress JumpA (
    .PCplus4 (PCnext_B1),
    .JumpField (JumpField_B1),
    .JumpAddr (JumpAddr)
);

// 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, Jump, ALUOp, Read1, Read2, Extended, Rt, R
BF #(187) B2 (
    .IN(Cable_combinado2),
    .CLK(Clk),
    .OUT(Cable_salida2)
);

// Separamos las señales en B2
assign MemRead_B2 = Cable_salida2[186];
assign MemToReg_B2 = Cable_salida2[185];
assign MemWrite_B2 = Cable_salida2[184];

```



```

assign RegWrite_B2      = Cable_salida2[183];
assign RegDst_B2        = Cable_salida2[182];
assign ALUSrc_B2        = Cable_salida2[181];
assign Branch_B2        = Cable_salida2[180];
assign Jump_B2          = Cable_salida2[179];

assign ALUOp_B2         = Cable_salida2[178:176];

assign Read1_B2         = Cable_salida2[175:144];
assign Read2_B2         = Cable_salida2[143:112];
assign Extended_B2      = Cable_salida2[111:80];

assign Rt_B2            = Cable_salida2[79:75];
assign Rd_B2            = Cable_salida2[74:70];

assign PCnext_B2        = Cable_salida2[69:38];
assign Funct_B2         = Cable_salida2[37:32];
assign JumpAddr_B2      = Cable_salida2[31:0];

// 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
assign Cable_combinado3 = {PCnext_B2, WriteReg, ALURes, ZF, AddRes, MemRead_B2, MemWrite_B2, MemToReg_B2, RegWrite_B2, Read2_B2, Bran
BF #(172) B3 (
    .IN(Cable_combinado3),
    .CLK(Clk),
    .OUT(Cable_salida3)
);

// Separamos las señales en B3
assign PCnext_B3      = Cable_salida3[171:140];
assign WriteReg_B3    = Cable_salida3[139:135];

```

```

assign ALURes_B3      = Cable_salida3[134:103];
assign ZF_B3          = Cable_salida3[102];
assign AddRes_B3      = Cable_salida3[101:70];
assign MemRead_B3     = Cable_salida3[69];
assign MemWrite_B3    = Cable_salida3[68];
assign MemToReg_B3    = Cable_salida3[67];
assign RegWrite_B3    = Cable_salida3[66];
assign Read2_B3       = Cable_salida3[65:34];
assign Branch_B3      = Cable_salida3[33];
assign Jump_B3        = Cable_salida3[32];
assign JumpAddr_B3    = Cable_salida3[31:0];

// 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];

// MUX Branch vs PC+4
wire [31:0] PC_branch_mux;
Mux2to1Param #(32) MUX_BR (
    .Entrada0(PCnext_B3),
    .Entrada1(AddRes_B3),
    .Sel      (Br_AND_ZF),
    .Salida   (PC_branch_mux)
);

// MUX Branch/PC+4 o Jump
Mux2to1Param #(32) MUXPC (
    .Entrada0(PC_branch_mux),
    .Entrada1(JumpAddr_B3),
    .Sel      (Jump_B3),
    .Salida   (PCin)
);

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

endmodule

```

Paso a paso:

Explicaremos paso a paso cómo funciona el módulo con la instrucción tipo J en base a la memoria de instrucciones proporcionada por el profesor:

```
00000000000000000000000000000000  nop
001000 00000 01000 0000010100111001  addi $t0, $zero, 1337
00000000000000000000000000000000  nop
000010 0000000000000000000000000000  j #0
00000000000000000000000000000000  nop
00000000000000000000000000000000  nop
001000 10000 10001 111110100101100  addi $s1, $s0, -724
```

...

Instrucción 1: nop

Primero tenemos la instrucción nop, esta está conformada por 32 ceros en binario, de esta forma: 00000000000000000000000000000000.

Lo primero que pasa es que el ciclo fetch le da 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 Unidad de Control, al entrar OpCode 000000, se activan las señales RegWrite y RegDst, y ALUOp se establece en 010 (para instrucciones tipo R). Esto ocurre porque la Unidad de Control solo ve el OpCode y no el campo Funct, por lo que trata la instrucción como si fuera de tipo R.

En el banco de registros se leen los valores de Rs y Rt (ambos 0).

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

En la etapa de ejecución, el multiplexor selecciona Rd como posición de escritura debido a que RegDst está activo. El módulo ALUControl recibe el AluOp siendo 010 y el Funct siendo 000000, y establece ALUCtl en 011. La ALU recibe este control, entendiendo que es una instrucción NOP por lo que setea Res = 0, esto hace que encienda ZF. Estos resultados pasan por el buffer 3.

Como esta instrucción no utiliza memoria, no se realizan operaciones en esta etapa. Los valores pasan al buffer 4. Aquí, el resultado 0 se escribe en el registro

\$0 porque RegWrite está activo. Como en MIPS, el registro \$0 siempre contiene el valor 0, hace que esta escritura no tenga efecto realmente.

Instrucción 2: addi \$t0, \$zero, 1337

Primero tenemos la instrucción addi \$t0, \$zero, 1337, que en formato binario sería: 00100000000010000000010100111001.

El ciclo fetch le otorga al módulo MEMI la posición 4 y sus 3 instrucciones seguidas. Con ello MEMI lee la instrucción de 32 bits.

Esta instrucción pasa al buffer 1, este almacena la instrucción y el valor de PC+4 (8).

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

- OpCode: 001000 (addi)
- Rs: 00000 (zero)
- Rt: 01000 (t0)
- Inmediato: 0000010100111001 (1337)

En la Unidad de Control, al entrar OpCode, se activan las señales RegWrite y ALUSrc, y ALUOp con el valor de 000. RegDst se desactiva porque utilizaremos Rt como dirección de escritura.

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

Todas estas señales y valores pasan al buffer 2.

En la etapa de ejecución, el multiplexor selecciona Rt como posición de escritura. ALUSrc está activo, haciendo que el inmediato extendido pase como segundo operando en la ALU. La ALU realiza la suma de $0 + 1337 = 1337$. Estos resultados pasan al buffer 3.

Como esta instrucción no utiliza memoria, no se realizan operaciones en esta etapa. Los valores pasan al buffer 4. Aquí, el resultado 1337 se escribe en el registro \$t0 (posición 8) porque RegWrite está activo.

Instrucción 3: nop

Esta instrucción es idéntica a la primera instrucción nop. Aquí PC avanza a 12.

Instrucción 4: j #0

Primero tenemos la instrucción j #0, que en formato binario sería: 00001000000000000000000000000000.

El ciclo fetch le otorga al módulo MEMI la posición 12 y las 3 instrucciones seguidas. Con ello MEMI lee la instrucción de 32 bits.

Esta instrucción pasa al buffer 1, este almacena la instrucción y el valor de PC+4 (16).

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

- OpCode: 000010 (j)
- JumpField: 0000000000000000000000000000 (dirección 0)

En la Unidad de Control, al entrar OpCode, se activa la señal Jump, indicando que se trata de una instrucción de salto.

El módulo JumpAddress calcula la dirección de salto combinando los 4 bits más significativos de PC+4, el JumpField y 2 bits a la izquierda, resultando en la dirección 0.

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

En la etapa de ejecución, no se realizan operaciones. La dirección de salto pasa al buffer 3.

En la etapa de memoria, como Jump está activo, el multiplexor final selecciona JumpAddr (0) como próxima dirección para el PC. Esto hace que el programa salte a la dirección 0, comenzando de nuevo el ciclo de instrucciones anterior.

Instrucción 5 y 6: nop (No se ejecutan en este ciclo)

Debido al salto a la dirección 0, estas instrucciones no se ejecutan.

Instrucción 7: addi \$s1, \$s0, -724 (No se ejecuta en este ciclo)

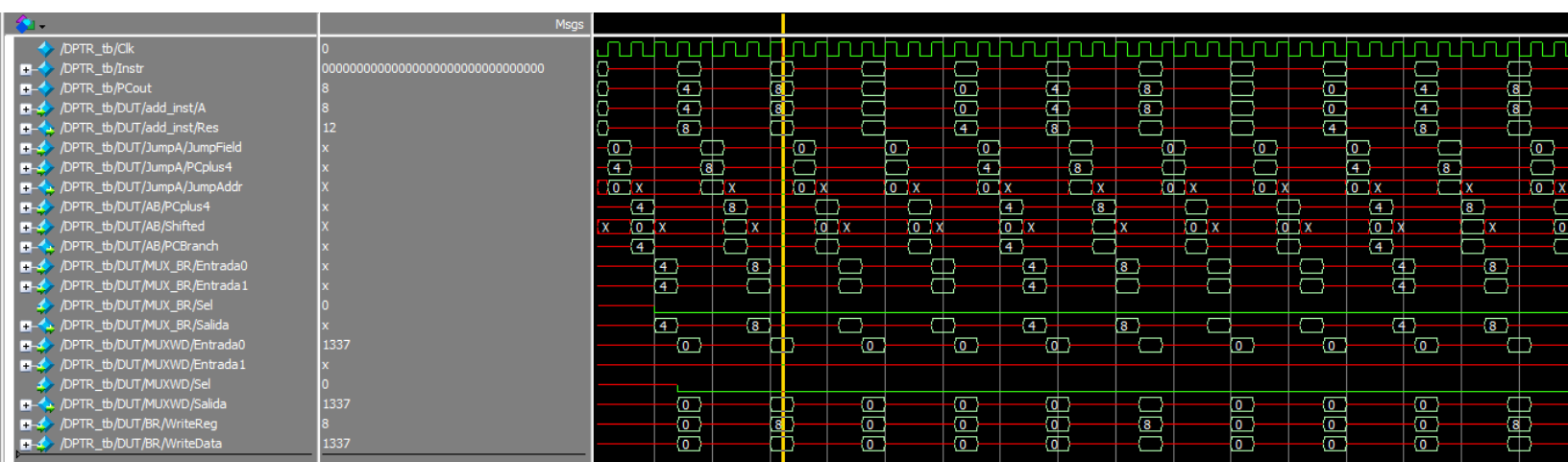
Esta instrucción tampoco se ejecuta en este ciclo debido al salto anterior. Se crea un bucle infinito que nunca llegará a ejecutar las siguientes instrucciones por el salto constante a la instrucción 1 nuevamente.

De esta forma vemos como se ejecuta la instrucción Jump de tipo J.

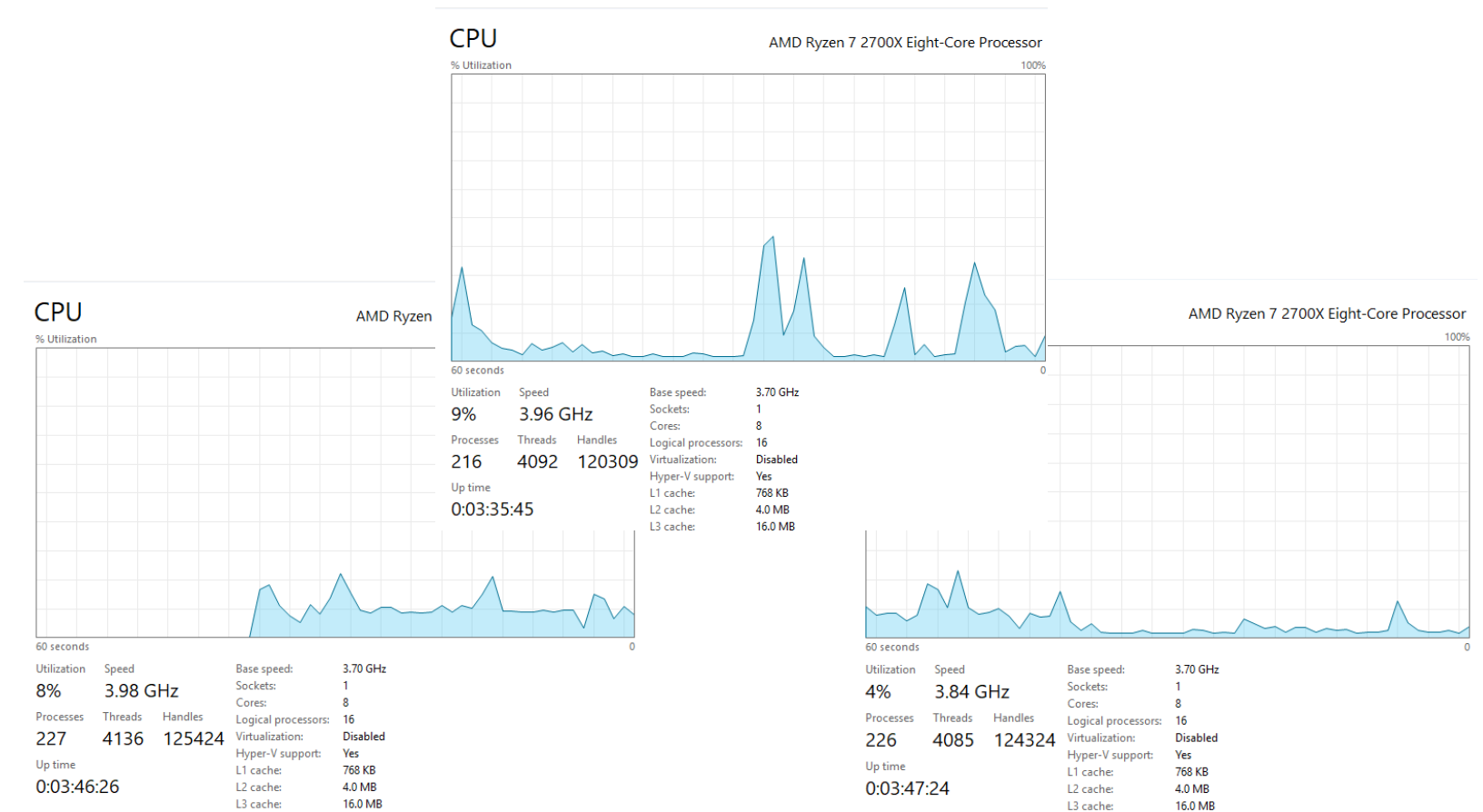
Graficos

En los siguientes gráficos podemos ver la simulación de las instrucciones mencionadas. Podemos ver el nop, seguido del addi donde WriteReg es 8 y WriteData es 1337 exactamente como lo mencionamos, seguido de un nop y del

salto de vuelta a la primera instrucción. Con ello podemos ver el bucle infinito que esto genera:

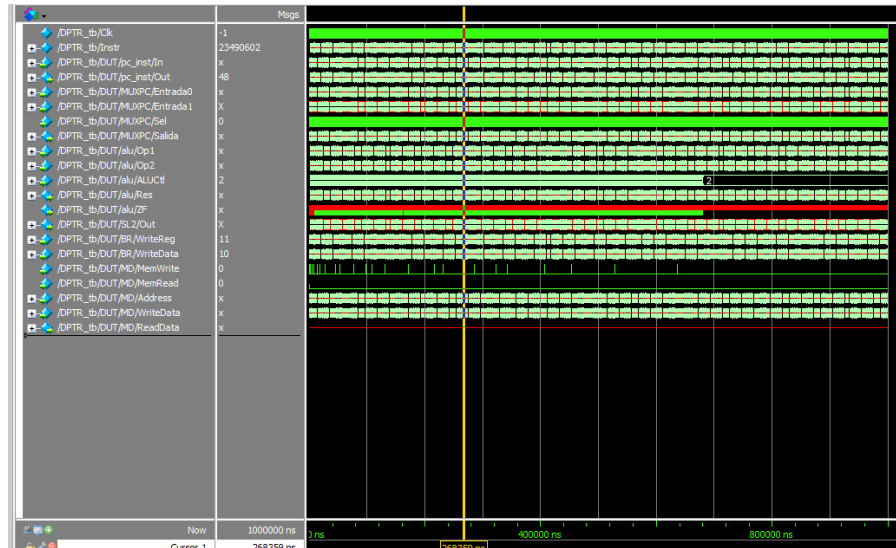


A continuación, capturas del procesador, antes, durante y después de la ejecución de esta prueba de instrucciones tipo J:



Algoritmo, numero primos del 2 al 100:

A continuación, probamos todo el módulo DPTR con todas sus instrucciones en un algoritmo que calcula los números primos del 2 al 100. Este fue descrito a detalle en la introducción, ahora probaremos si funciona correctamente:



Como podemos ver, obtenemos una wave inmensa ya que se hacen muchos ciclos, sobre todo restas para calcular correctamente los números primos. Pero podemos ver que los resultados son los esperados, comprobando que nuestro datapath completo realmente es funcional para todos los tipos de instrucciones solicitadas a lo largo de las 3 fases. A continuación, el banco de registros y memoria de datos:

0	0	0	2	0	0
1	0	3	0	3	0
2	0	6	0	0	5
3	0	9	0	0	0
4	0	12	7	0	0
5	25	15	0	11	0
6	100	18	0	0	13
7	0	21	0	0	0
8	101	24	17	0	0
9	100	27	0	19	0
10	0	30	0	0	23
11	1	33	0	0	0
12	96	36	29	0	0
13	1	39	0	31	0
14	1	42	0	0	37
15	0	45	0	0	0
16	0	48	41	0	0
17	0	51	0	43	0
18	0	54	0	0	47
19	0	57	0	0	0
20	0	60	53	0	0
21	0	63	0	59	0
22	0	66	0	0	61
23	0	69	0	0	0
24	0	72	67	0	0
25	0	75	0	71	0
26	0	78	0	0	73
27	0	81	0	0	0
28	0	84	79	0	0
29	0	87	0	83	0
		90	0	0	89
		93	0	0	0
		96	97	0	0

Como podemos ver, nuestro algoritmo para obtener los números primos fue exitoso. Primero, en el banco de registros tenemos \$5 que cuenta los números primos, deberían de ser 25, y efectivamente tiene dicho valor. Seguido de esto el \$8 contiene el numero actual, este está en 101 ya que el bucle termina al superar los 100. \$9 se mantiene con su valor de 100 al ser el limite superior señalado. \$4 es cero por ser la dirección inicial para guardar datos en la memoria. Por último \$7 siendo el verificador de primos, tiene un valor de 0 al final ya que el numero 100 no es primo. \$12 es la última dirección que guardamos en memoria. Y los otros dos datos son temporales.

Por otra parte, vemos que en la memoria de datos se guardaron los números primos que encontramos, estos se guardaron cada multiplo de 4 igual que con la memoria de instrucciones, teniendo \$0=2, \$4=3, \$8=5... y finalmente \$96=97. Corroborando que nuestro código fue exitoso.

Con esto completamos nuestro datapath corroborando que efectivamente es correcta su implementación para procesar instrucciones tipo I, J y R. Esperamos que este nos sirva mas adelante como referencia para futuros trabajos.

Extra: Código en C++ a ASM

Se nos pidió hacer el algoritmo en C++ para comparar nuestro código asm con el código asm que nos generaba nuestro programa de C++. Utilizamos la herramienta godbolt para pasar nuestro código en C++ a Mips clang 20.1.0.

Podemos ver que las instrucciones no son iguales a nuestro código asm pero analizando podemos establecer una correspondencia entre ambos códigos. Las principales diferencias entre MIPS clang son el manejo de memoria y registros, la estructura de datos, las comparaciones y los accesos a memoria.

El código asm que tenemos utiliza registros directos como \$9 \$8, mientras que el código generado almacena las variables en la pila y las carga y guarda constantemente. Nuestro código utiliza etiquetas numéricas y saltos absolutos, mientras que el generado utiliza etiquetas como \$BB0_1, \$BB0_3. Nuestro código usa un direccionamiento simple a memoria, mientras que el generado incluye cálculos de dirección más complejos con deslizamiento haciendo uso de \$fp.

A pesar de las diferencias, el código ensamblador implementa el mismo algoritmo que nosotros, al traducir código de alto nivel a ensamblador hay diferencias estructurales y de optimización, pero la lógica se llega a mantener.


```

3 int main() {
4     int current = 2; // $8
5     const int limit = 100; // $9
6     const int maxPrimes = 100;
7     int primes[maxPrimes]; // Memoria base a partir de $4
8     int primeCount = 0; // $5
9
10    // LoopPrincipal:
11    while (current <= limit) {
12        int divisor = 2; // $6
13        bool isPrime = true; // $7
14
15        // VerificarDivisor:
16        while (divisor < current) {
17            // Calculamos el resto de current/divisor por restas
18            int rem = current; // copia a $11
19            while (rem >= divisor) {
20                rem -= divisor; // SUB $11, $11, $6
21            }
22            // Si rem == 0, no es primo
23            if (rem == 0) {
24                isPrime = false; // ADDI $7, $0, 0
25                break;
26            }
27            // Siguiendo divisor (ADDI $6, $6, 1)
28            ++divisor;
29        }
30
31        // Si sigue siendo primo, lo almacenamos
32        if (isPrime) {
33            primes[primeCount] = current;
34            ++primeCount;
35        }
36
37        // Siguiendo número (ADDI $8, $8, 1)
38        ++current;
39    }
40
41    // Al terminar, imprimimos los primos y el conteo
42    std::cout << "Primos entre 2 y 100 (" << primeCount << " encontrados):\n";
43    for (int i = 0; i < primeCount; ++i) {
44        std::cout << primes[i] << " ";
45    }

```

A ▾ ⚙ Output... ▾ Filter... ▾ Libraries ▾ Overrides

```

1 main:
2     lui    $2, %hi(_gp_disp)
3     addiu  $2, $2, %lo(_gp_disp)
4     addiu  $sp, $sp, -464
5     sw     $ra, 460($sp)
6     sw     $fp, 456($sp)
7     move   $fp, $sp
8     addu   $1, $2, $25
9     sw     $1, 16($fp)
10    addiu   $1, $zero, 0
11    sw     $zero, 452($fp)
12    addiu   $1, $zero, 2
13    sw     $1, 448($fp)
14    addiu   $1, $zero, 100
15    sw     $1, 444($fp)
16    addiu   $1, $zero, 100
17    sw     $1, 440($fp)
18    addiu   $1, $zero, 0
19    sw     $zero, 36($fp)
20
21 $BB0_1:
22     lw     $2, 448($fp)
23     addiu  $1, $zero, 100
24     slt    $1, $1, $2
25     xori   $1, $1, 1
26     bgtz   $1, $BB0_3
27     nop
28     b      $BB0_18
29     nop
30
31 $BB0_3:
32     addiu  $1, $zero, 2
33     sw     $1, 32($fp)
34     addiu  $1, $zero, 1
35     sb     $1, 31($fp)
36
37 $BB0_4:
38     lw     $1, 32($fp)
39     lw     $2, 448($fp)
40     slt    $1, $1, $2
41     bgtz   $1, $BB0_6
42     nop
43     b      $BB0_14

```

Output (0/0) mips clang 20.1.0 i -963ms (65338B) ~5018 lines fit

```

40     b      $BB0_14
41     nop
42 $BB0_6:
43     lw     $1, 448($fp)
44     sw     $1, 24($fp)
45
46 $BB0_7:
47     lw     $1, 24($fp)
48     lw     $2, 32($fp)
49     slt    $1, $1, $2
50     xori   $1, $1, 1
51     bgtz   $1, $BB0_9
52     nop
53     b      $BB0_10
54     nop
55
56 $BB0_9:
57     lw     $2, 32($fp)
58     lw     $1, 24($fp)
59     subu   $1, $1, $2
60     sw     $1, 24($fp)
61     b      $BB0_7
62     nop
63
64 $BB0_10:
65     lw     $1, 24($fp)
66     addiu  $2, $zero, 0
67     xor    $1, $1, $zero
68     sltiu  $1, $1, 1
69     bgtz   $1, $BB0_12
70     nop
71     b      $BB0_13
72     nop
73
74 $BB0_12:
75     addiu  $1, $zero, 0
76     sb     $zero, 31($fp)
77     b      $BB0_14
78     nop
79
80 $BB0_13:
81     lw     $1, 32($fp)
82     addiu  $2, $zero, 1
83     addu   $1, $1, $2
84     sw     $1, 32($fp)
85     b      $BB0_14
86     nop

```

```

79     sw     $1, 32($fp)
80     b      $BB0_4
81     nop
82
83 $BB0_14:
84     lbu    $1, 31($fp)
85     andi   $1, $1, 1
86     bgtz   $1, $BB0_16
87     nop
88     b      $BB0_17
89     nop
90
91 $BB0_16:
92     lw     $1, 448($fp)
93     lw     $3, 36($fp)
94     addiu  $2, $fp, 40
95     sll    $3, $3, 2
96     addu   $2, $2, $3
97     sw     $1, 0($2)
98     lw     $1, 36($fp)
99     addiu  $2, $zero, 1
100    addu   $1, $1, $2
101    sw     $1, 36($fp)
102
103 $BB0_17:
104     lw     $1, 448($fp)
105     addiu  $2, $zero, 1
106     addu   $1, $1, $2
107     sw     $1, 448($fp)
108     b      $BB0_1
109     nop
110
111 $BB0_18:
112     lw     $1, 16($fp)
113     lw     $4, %got(_ZSt4cout)($1)
114     lw     $2, %got($$.str)($1)
115     addiu  $5, $2, %lo($$.str)
116     lw     $25, %got(_ZStlsISt11char_traitsIcEERSt13basic_ostreamI
117     jalr   $25
118     nop
119     lw     $1, 16($fp)
120     move   $4, $2
121     lw     $5, 36($fp)
122     lw     $25, %got(_ZStlsISt11char_traitsIcEERSt13basic_ostreamI

```

```

118     lw     $25, %got(_ZStlsISt11char_traitsIcEERSt13basic_ostreamI
119     jalr   $25
120     nop
121     lw     $1, 16($fp)
122     move   $4, $2
123     lw     $2, %got($$.str.1)($1)
124     addiu  $5, $2, %lo($$.str.1)
125     lw     $25, %got(_ZStlsISt11char_traitsIcEERSt13basic_ostreamI
126     jalr   $25
127     nop
128     addiu  $1, $zero, 0
129     sw     $zero, 20($fp)
130
131 $BB0_19:
132     lw     $1, 20($fp)
133     lw     $2, 36($fp)
134     slt    $1, $1, $2
135     bgtz   $1, $BB0_21
136     nop
137     b      $BB0_23
138     nop
139
140 $BB0_21:
141     lw     $1, 16($fp)
142     lw     $3, 20($fp)
143     addiu  $2, $fp, 40
144     sll    $3, $3, 2
145     addu   $2, $2, $3
146     lw     $5, 0($2)
147     lw     $4, %got(_ZSt4cout)($1)
148     lw     $25, %got(_ZStlsISt11char_traitsIcEERSt13basic_ostreamI
149     jalr   $25
150     nop
151     lw     $1, 16($fp)
152     move   $4, $2
153     lw     $5, 20($fp)
154     lw     $25, %got(_ZStlsISt11char_traitsIcEERSt13basic_ostreamI
155     jalr   $25
156     lw     $1, 20($fp)

```

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 reforzar 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.

Tercera fase:

Esta fase fue más sencilla, aunque con algunos problemitas en la parte del algoritmo ya que no habíamos calculado bien los offset para las operaciones de Branch. Me gustó realizar el proyecto, aunque si sufrí de mucho estrés porque fueron tres semanas de este proyecto y muchas cosas no salen, quieres buscar el error, preguntas, nadie te responde. Muy frustrante porque estuve a full con esto descuidado otras materias, pero al final algo se logró al menos. Una experiencia y al menos puedo decir que entendí todo lo que entregué.

General de la clase:

Fue una clase bastante estresante porque en el aula me sentía muy inútil, creo que no se explicaba muy bien, lo que hacía que no tuviera dudas en ese momento pero al intentar hacerlo salieran mil y un dudas, lo que me lleva a mi critica siguiente, y es que si tienes dudas después de la clase es muy probable que nadie te las responda, mis compañeros me decepcionaron al no ayudarme, y bueno un poco en parte el profe, me hubiera gustado que al menos me dijera que no en vez de dejarme en visto, duele menos jajaja.

De todas formas y pese a las dificultades, me las pude ingeniar y realmente aprendí muchas cosas, ratos de pensar y pensar hicieron que entendiera muy bien

cada parte del datapath y me ayudaron a generar esa adaptabilidad (sacrificando los fines de semana XD). Me llevo muy buen aprendizaje y los recuerdos gratos.

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 aún nos falta la tercera fase, pero eso tendrá solución la siguiente semana, esperando que el algoritmo funcione correctamente.

Tercera fase: Sinceramente pensé que sería más fácil incluir las instrucciones tipo J, ya que en mi cabeza dije “solo es agregar otro multiplexor y otro modulo similar al shift left 2 ¿qué tan difícil puede ser?” Y pues hacer los módulos no fue lo complicado, lo complicado fue conectarlos en el datapath y que pasaran por los buffers, pero al final se logró. Ahora había que pasar al siguiente reto, hacer que funcionara el algoritmo, ya que si bien ya lo teníamos, lo cierto es que no sabíamos si funcionaba, y en efecto no funciono, por lo que tuvimos que reestructurar gran parte de las instrucciones, por un lado era corregir la lógica del algoritmo, y por el otro, calcular de manera adecuada los offsets de las instrucciones que usaban Branch, casi nos rendimos, pero cuando la esperanza parecía perdida, logramos hacer que funcionara, a tan solo horas antes de la entrega, fue increíble la verdad.

General de la clase: Tengo que decir que le tengo un gran amor-odio a esta clase, porque si bien probablemente sea la clase en la que mas crisis de identidad he tenido y donde mi autoestima fue brutalmente sabotada, lo cierto es que esta ha sido la clase donde mas he aprendido y por la cual mas me he interesado, sobre todo después de empezar a ver las instrucciones mips que fue donde me empezó a interesar más esta clase, el proyecto final y sus 3 fases me gusto bastante hacerlo, a pesar de que no haya visto la luz del día en varios días.

Como critica hacia el profesor puedo decir que me gustaría que organizara mas sus clases, y con esto me refiero a no explicar toda la clase ahí en el momento, ya que eso ocasionaba que el profe cometiera errores y yo no comprendiera bien el tema, además me gustaría que organizara mejor las tareas del classrom, ya que varias tareas e incluso en las 3 fases del proyecto final, había pdfs con información antigua o errónea sobre lo que realmente íbamos a hacer o entregar, me gustaría que el profe especificara mejor las instrucciones de cada tarea, y pdfs actualizados con la información e instrucciones que necesitamos. Tengo que decir que el profesor es atento a las dudas y las responde en el salón de clases, pero a veces quedaba uno que otro cabo suelto.

Para finalizar solo diré que, buena clase uwu.

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/>

Stryker, C. (14 de Junio de 2024). *IBM*. Obtenido de ¿Qué es data pipeline?: <https://www.ibm.com/mx-es/topics/data-pipeline#:~:text=Un%20data%20pipeline%20o%20pipeline,de%20datos%2C%20para%20su%20análisis.>