PROYECTO FINAL FASE-1

JAUREGUI RABELERO HUGO JOHNATHAN
HERMOSILLO GUIZAR IRVIN YAEL
CRUZ SANCHEZ JUAN PABLO
COVARRUBIAS BECERRA JOSE ANTONIO

15/11/2023

Link de GitHub: [HilloTS/Mips-Equipo-1 (github.com)]

Investigación

Procesador MIPS de 32 bits

El procesador MIPS (Microprocessor without Interlocked Pipeline Stages) de 32 bits es una arquitectura de procesador basada en RISC (Reduced Instruction Set Computing). A continuación se detallan algunos de sus elementos y características generales:

1. Unidad de control (Control Unit): es la parte del procesador que se encarga de controlar el flujo de datos y de instrucciones a través de las diferentes unidades del procesador.
2. Unidad aritmético-lógica (Arithmetic Logic Unit, ALU): es la unidad encargada de realizar las operaciones aritméticas y lógicas básicas, como sumar, restar, multiplicar, dividir, comparar y realizar operaciones booleanas
3. Arquitectura RISC: El procesador MIPS utiliza una arquitectura RISC (Reduced Instruction Set Computing) que se caracteriza por tener un conjunto de instrucciones reducido y simple. Esto permite que el procesador pueda ejecutar instrucciones de manera más rápida y eficiente
4. Memoria caché: el procesador MIPS utiliza una memoria caché para acelerar el acceso a los datos almacenados en memoria. La memoria caché es una memoria de acceso rápido que almacena copias de los datos más utilizados en la memoria principal.
5. FPU (Floating Point Unit): es la unidad encargada de realizar operaciones aritméticas en punto flotante, como sumas, restas, multiplicaciones y divisiones.
6. Arquitectura de canalización: El procesador MIPS utiliza una arquitectura de canalización para aumentar la eficiencia en la ejecución de instrucciones. La canalización permite que varias instrucciones se ejecuten al mismo tiempo, 1 reduciendo el tiempo de espera entre instrucciones.
7. Registro de propósito general: El procesador MIPS tiene 32 registros de propósito general de 32 bits, que se utilizan para almacenar datos y direcciones de memoria.

Set de instrucciones

INSTRUCCION ADD: La instrucción "add" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La instrucción "add" se utiliza para sumar dos operandos y almacenar el resultado en un registro de destino.

INSTRUCCION SUB: La instrucción "sub" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La

instrucción "sub" se utiliza para restar dos operandos y almacenar el resultado en un registro de destino.

INSTRUCCION OR: La instrucción "or" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La instrucción "or" se utiliza para realizar una operación lógica OR a nivel de bit entre dos operandos y almacenar el resultado en un registro de destino

INSTRUCCION AND: La instrucción "and" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La instrucción "and" se utiliza para realizar una operación lógica AND a nivel de bit entre dos operandos y almacenar el resultado en un registro de destino.

INSTRUCCION SLT: La instrucción "slt" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La instrucción "slt" se utiliza para comparar dos valores y almacenar el resultado de la comparación en un registro de destino.

INSTRUCCION NOP: La instrucción "nop" es una instrucción de tipo R utilizada en la arquitectura de computadoras y en el lenguaje de programación ensamblador. La instrucción "nop" (abreviatura de "no operation" o "sin operación" en español) se utiliza para indicar que no se realizará ninguna operación en la ejecución de una instrucción y se utiliza comúnmente como un espacio reservado para futuras expansiones del programa

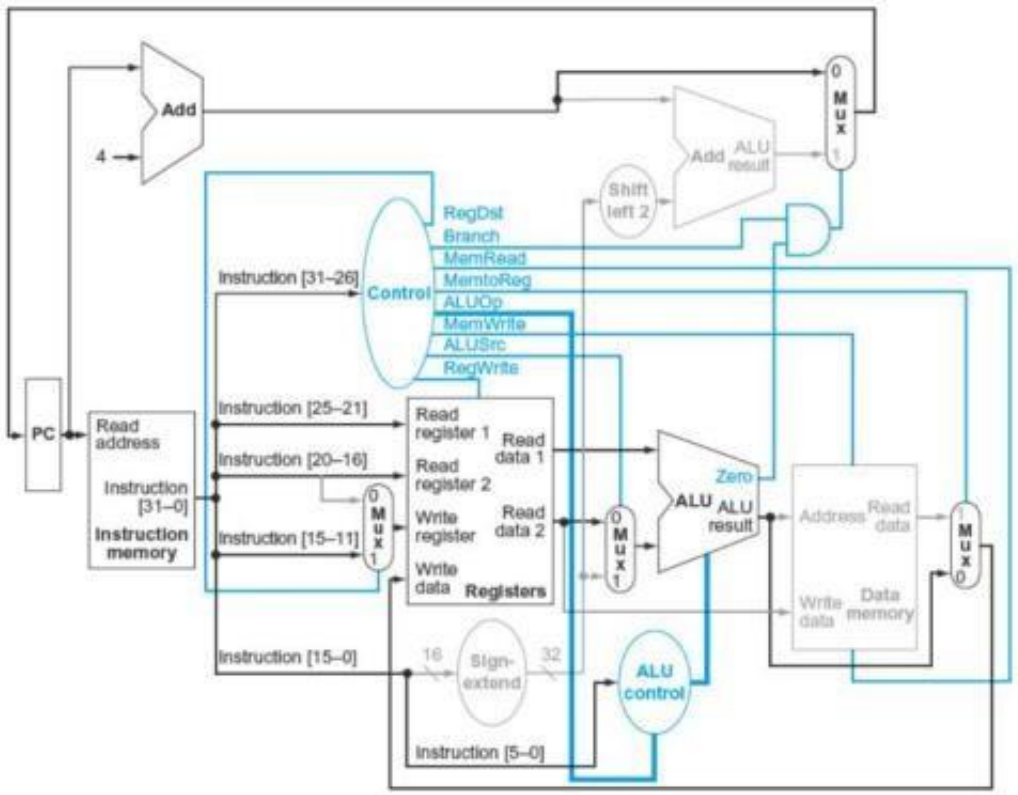
Personajes

LJUBISA BAJIC: Ljubisa Bajic es un ingeniero en computación que ha trabajado en el diseño de microprocesadores durante más de 30 años. Una de sus contribuciones más importantes en microarquitectura ha sido en la técnica de pipeline en paralelo, que permite que el procesador ejecute múltiples instrucciones simultáneamente. Bajic también ha trabajado en la mejora de la eficiencia energética de los procesadores y ha desarrollado técnicas para reducir el consumo de energía de los circuitos de procesamiento de datos.

JIM KELLER: Jim Keller es otro ingeniero en computación que ha sido fundamental en la evolución de los microprocesadores. Keller ha trabajado en el diseño de procesadores de alta gama durante más de 20 años y ha liderado equipos de diseño en empresas como AMD, Apple e Intel. Una de sus contribuciones más importantes ha sido en la arquitectura de los procesadores x86 de 64 bits, que se utilizan en la mayoría de los ordenadores personales modernos. Keller también ha trabajado en la optimización de la eficiencia energética de los procesadores, la mejora de la velocidad de reloj y la reducción del tamaño de los transistores en los circuitos integrados

Desarrollo

Se implementaron módulos correspondientes a la siguiente imagen:



En código se ve de la siguiente manera:

Adder:

```
1   `timescale 1ns/1ns
2   module Adder(
3       input [31:0] suma,
4       input [31:0] numprimero,
5       output reg [31:0] fuera
6   );
7
8   assign fuera = numprimero + suma;
9
10  endmodule
```

ALU:

```
module alu(
        input [31:0]op1,
        input [31:0]op2,
        input [3:0]sel,
        output reg [31:0]Resultado,
        output reg zeroflag
);



always @*
begin
        case (sel)
                4'b0000:
                        Resultado= op1&op2;
                4'b0001:
                        Resultado = op1|op2;
                4'b0010:
                        Resultado = op1+op2;
                4'b0110:
                        Resultado = op1-op2;
                4'b0111:
                        Resultado = op1<op2?1:0;
                4'b1100:
                        Resultado = ~(op1 | op2);
                default:
                        Resultado =0;
        endcase
        if(Resultado==0)
                zeroflag=1;
```

Alu control:

```
module ALU_Control(

        input [5:0]functionfield,
        input [1:0]Alu_op,
        output reg [3:0]operacion
);

always@*
begin
        case(Alu_op)
        2'b10:
                case(functionfield)
                6'b100000:
            operacion=4'b0010;
        6'b100010:
            operacion=4'b0110;
        6'b100101:
            operacion=4'b0001;
        6'b100100:
            operacion=4'b0000;
        6'b101010:
            operacion=4'b0111;
        endcase
    endcase
end

endmodule
```

Memoria:

```
`timescale 1ns/1ns
module Memoria(
        input Wen,//Senial para escribir
        input [31:0]Adress,//La direccion de la memoria
        input [31:0]DataW,//Escribir un valor nuevo en la memoria
        input Ren,//Senial para leer
        output reg [31:0]DataR//Valor para leer
);

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

always@*
begin
        if(Wen)
                ROM[Adress]=DataW;
        if(Ren)
                DataR=ROM[Adress];
        if(Wen&Ren)
                DataR=0;
end
endmodule
```

Mux 5 bits:

```
`timescale 1ns/1ns
module Mux5bits(
    input [4:0]Valoren1,
    input [4:0]Valoren0,
    input selector,
    output reg [4:0]WriteReg
);

always @*
    begin
        case(selector)
        1'b1:
            WriteReg=Valoren1;
        1'b0:
            WriteReg=Valoren0;
        endcase
    end
endmodule
```

Y así se creó modulo por modulo hasta que cada uno de los módulos se instancio en un TestBench.

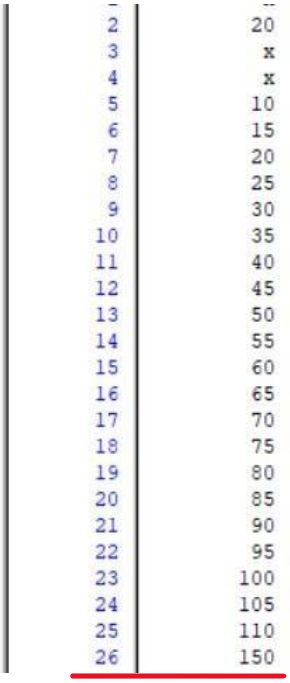
Resultados:

Una vez creado el TestBench e instanciado los módulos se corrió la simulación y lo primero fue cargar la memoria de instrucciones ADD:

```
5   000000_10//ADD $26, $17, $19 70+80=150
6   001_10011
7   11010_000
8   00_100000
9
10
```

Diciéndole que vamos a sumar lo que este en la posición 17,19 y se lo vamos a agregar a la posición 26.

Corroboramos los resultados de revisando nuestro banco de registros:



Referencias:

MIPS Architecture. (n.d.). Retrieved from https://mips.com/

LjubisaBajic, Founder and CEO, Tenstorrent - Topio Networks. https://www.topionetworks.com/people/ljubisa-bajic-588b09eb2c537740e300002a

Cutress, I. (2021, May 27). An Interview with Tenstorrent: CEO Ljubisa Bajic and CTO Jim Keller. AnandTech.

https://www.anandtech.com/show/16709/an-interview-with-tenstorrent-ceo-ljubisa-bajic-and-ctojim-keller

Fase 2-Instrucciones tipo I

ARQUTECTURA DE COMPUTADORAS

- **Hermosillo Guizar Irvin Yael**
- **Jauregui Rabelero Hugo Johnathan**
- **Cruz Sanchez Juan Pablo**
- **Covarrubias Becerra Juan Pablo**

Maestro: Jorge Ernesto López Arce

Link GitHub: [HilloTS/Mips-Equipo-1 (github.com)]

Introducción Fase 2-Instrucciones tipo I.

La implementación del proyecto en la primera fase es capaz de realizar operaciones aritméticas y lógicas simples (suma, resta, OR, AND, SLT y NOP), esta segunda fase permitirá la ejecución de instrucciones de tipo I. Este tipo de operaciones son **de transferencia, salto condicional e instrucciones con operandos inmediatos**.

Este tipo de operaciones (en especial transferencia) serán de gran ayuda para comenzar a integrar la memoría de datos en algoritmos cargados en la memoría de instrucciones y para comenzar a implementar estructuras cíclicas y condicionales.

Objetivos Fase 2-Instrucciones tipo I.

General:

Agregar los módulos necesarios al datapath para poder ejecutar las instrucciones tipo I de la tabla 1 y tabla 2. Vea el archivo Fase2.PDF para más detalles.

Instruccion	Tipo	sintaxis
Add	R	Add $rd, $rs, $rt
Sub	R	
Mul	R	
Div	R	
Or	R	
And	R	
Addi	I	
Subi	I	
Ori	I	
Andi	I	

Tabla 2

Instruccion	Tipo	sintaxis
Lw	I	
Sw	I	
slt	R	
Slti	I	
beq	I	
bne	I	
j	J	
nop	R	
bgtz	i	

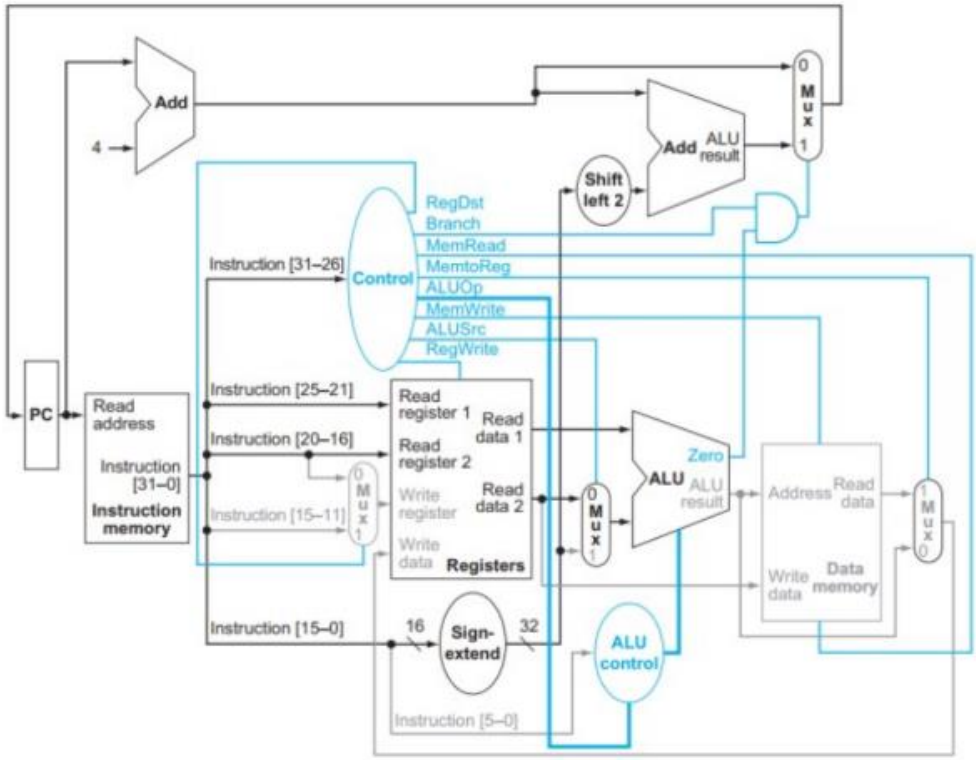
Tabla 1

Desarrollo de Verilog:
Incluir 2 nuevos módulos para implementación de tipo **I** (signExtend 32 y shift2left), con ellas podremos realizar saltos condicionales, referente a instrucciones inmediatas y carga o lectura de **datos en memoría.**
Desarrollo del decodificador:
Traducir las instrucciones tipo **I** incluidas en *Tabla 1* y *Tabla 2* a código ensamblador.

Desarrollo Fase 2-Instrucciones tipo I.

Esta fase requiere la incorporación de 2 módulos nuevos (**sign extend 32** y **shift left 2**) estas operaciones nos serán de ayuda para las instrucciones de salto y para las instrucciones inmediatas, por su parte, las instrucciones de transferencia (**SW**,

LW) necesitan sobre todo cambios en la unidad de control. El resultado de las nuevas implementaciones debe ser el descrito en la siguiente imagen.



La extensión de signo de 32 bits tomará una entrada de 16 bits. A esta entrada se le añadirán 16 bits adicionales en los espacios más significativos (a la izquierda). Estos bits adicionales se asignarán con un valor de “1” o “0”, dependiendo del bit más significativo de la entrada proporcionada. Esta salida ampliada de 32 bits puede tener dos posibles destinos:

- La Unidad Lógica Aritmética (ALU): Esto ocurre si es una instrucción inmediata o para instrucciones como Beq o similares.
- Desplazamiento a la Izquierda 2 (Shift 2 Left): Esto ocurre si es una instrucción de salto (Beq o Bnq).

Además, se requerirá un segundo sumador en el ciclo de búsqueda (fetch) para las instrucciones de tipo J (que no corresponden a esta fase).

Volviendo a las instrucciones de tipo I, el valor de este sumador solo se considerará si la salida del módulo ALU resulta en un zeroFlag de “1” (lo que significa que el resultado de la ALU es “0”) y si la unidad de control identificó la instrucción como una de salto (branch).

Cuando verificamos que el resultado de la ALU es “0”, estamos hablando de una instrucción de salto de tipo beq (branch). Su objetivo es comparar dos entradas numéricas proporcionadas. Esta comparación se realiza a través de una operación

de resta, y si el zeroFlag resulta en 1 en este caso, indica que los dos valores numéricos son iguales.

Hemos descrito la instrucción de salto condicional, que puede ser utilizada para instrucciones condicionales, pero no para instrucciones cíclicas. Esto se debe a que solo puede realizar saltos de instrucciones en adiciones, es decir, solo puede saltar hacia adelante en el documento de instrucciones y no puede volver atrás. Para ello, se implementa una instancia de desplazamiento a la izquierda 2 (shift left 2) en el ciclo de búsqueda que se desarrollará en la siguiente fase.

Módulos Implementados en esta fase:

El módulo PC es el encargado de gestionar la salida de la dirección de lectura de la “memoria de instrucciones”. Su función es parecida a la de un almacenador temporal o búfer.

Por otro lado, el Add o Adder lleva a cabo una operación de adición con las dos entradas que se le proporcionan, y emite el resultado de dicha operación como salida.

Módulos de Utilidad en esta fase:

Multiplexores:

Los multiplexores son dispositivos que reciben dos entradas y, según el valor de un "selector" controlado principalmente por la unidad de control, emiten una de las dos entradas como salida. La dimensión de las entradas y salidas puede variar según su implementación, pero en general, la salida tiene la misma longitud que la entrada. Es posible que se requieran múltiples multiplexores, diferenciándose solo en la longitud de la palabra.

Módulos Nuevos de Utilidad:

1.Sign Extend:

Este módulo toma una entrada de 16 bits. En términos generales, extrae los 16 bits menos significativos de la instrucción y los extiende a 32 bits. El proceso de extensión implica replicar el valor del bit más significativo de la entrada en los 16 bits de extensión. En otras palabras, si el bit más significativo es 1, los 16 bits de relleno también serán 1; lo mismo se aplica si el bit más significativo es 0.

2.Shift Left 2:

Este módulo realiza un desplazamiento de 2 bits a la izquierda en la entrada recibida. La entrada puede tener diferentes tamaños, similar a los multiplexores, ya que también se implementará en la fase 3. Por ejemplo, si la entrada es 0011001, la salida será 1100100.

Estos módulos y multiplexores son componentes esenciales en la arquitectura de un sistema, permitiendo la manipulación y extensión de datos, así como la selección de entradas en función de un controlador.

Desarrollo del decodificador Fase 2-Instrucciones tipo I.

```
operationCodeTypeR = [  #Instrucciones de tipo R
    "ADD", "SUB", "SIT", "AND", "OR"
]

operationCodeTypeI=[ #Instrucciones de tipo I
    "ADDI", "ANDI", "ORI", "LW", "SW"
]

functTypeR = { # Funct de las instrucciones tipo R
    "SIT": "101010",
    "ADD": "100000",
    "SUB": "100010",
    "AND": "100100",
    "OR" : "100101"
}

opCodeTypeI = { #OpCode de instrucciones tipo I
    "ADDI": "001000",
    "ANDI": "001100",
    "ORI": "001101",
    "LW": "100011",
    "SW": "101011"
}


def operationType(word): #Se selecciona de que tipo es la instruccion
    if word[0] in operationCodeTypeR: #Se busca dentro de las operaciones de tipo R
        r = typeR(word)
    if word[0] in operationCodeTypeI:#Se busca dentro de las operacioness de tipo I
        r = typeI(word)

```

```
    return r


def typeR(instruction): #Convertidor de bits para instruccion tipo R
    op = "000000"     #Valor op por defecto
    shampt = "00000"  #Valor de shampt por defecto
    prueba = ""   #Aqui se almacenara toda la operacion ordenada

    prueba += op
    rs = int(instruction[2].replace("$", "")) #Se reemplaza el $ por nada
    rs = bin(rs)[2:].zfill(5)       #Se pasa a binario y se asegura de tener 5 bits
    prueba += str(rs)    #Se concatena

    rt = int(instruction[3].replace("$", ""))#Se reemplaza el $ por nada
    rt = bin(rt)[2:].zfill(5)#Se pasa a binario y se asegura de tener 5 bits
    prueba += str(rt)#Se concatena

    rd = int(instruction[1].replace("$", ""))#Se reemplaza el $ por nada
    rd = bin(rd)[2:].zfill(5)#Se pasa a binario y se asegura de tener 5 bits
    prueba += str(rd)#Se concatena

    prueba += shampt

    prueba+= functTypeR[instruction[0]]#Se busca dentro del diciconario para añadir el funct
    return prueba #Retorna la cadena
```

```
def typeI(instruction):
    opcode = opCodeTypeI[instruction[0]]  #Se busca su OP en su diccionario
    rs = int(instruction[2].replace("$", "")) #Se reemplaza el $ por nada
    rs = bin(rs)[2:].zfill(5)#Se pasa a binario y se asegura de tener 5 bits
    rt = int(instruction[1].replace("$", ""))#Se reemplaza el $ por nada
    rt = bin(rt)[2:].zfill(5)#Se pasa a binario y se asegura de tener 5 bits
    immediate = bin(int(instruction[3]))[2:].zfill(16)#convierte el valor a binario, elimina
                                                      #'0b' y se asegura que tenga una longitud de 16 bits

    return opcode + str(rs) + str(rt) + immediate  #Se concatena todo lo anterior y se regresa

condicion = True
print("BIENVENIDO")
while condicion:
    print("Ingrese la opcion a realizar")
    print("1.Leer archivo")
    print("2.Salir")
    opcion = input("Opcion: ")

    if(opcion == "1"):
        file_name = input("Ingrese el nombre de su archivo a leer: ")#Se ingresa el nombre del archivo
        with open(file_name, 'r') as asm_file:#Se abre el archihivo en modo de lectura
            asm_instructions = asm_file.readlines() #Se separan las lineas

        with open("prueba.bin", 'w') as bin_file: #Se abre un archivo donde se guardaran los bits
            for instruction in asm_instructions:
                word = instruction.strip().split() #Se separa cada palabra de la instruccion
                bin_instruction = operationType(word) #Llama para identificar el tipo de instruccion
                for i in range(0, len(bin_instruction), 8): #Se separa en 8 bits
                    bin_file.write(bin_instruction[i:i+8] + '\n')
```

```
        with open("prueba.bin", 'w') as bin_file: #Se abre un archivo donde se guardaran los bits
            for instruction in asm_instructions:
                word = instruction.strip().split() #Se separa cada palabra de la instruccion
                bin_instruction = operationType(word) #Llama para identificar el tipo de instruccion
                for i in range(0, len(bin_instruction), 8): #Se separa en 8 bits
                    bin_file.write(bin_instruction[i:i+8] + '\n')

    elif(opcion == "2"):
        condicion = False
    else:
        print("OPCION NO VALIDA\n")

print("HASTA LUEGO")

with open("prueba.bin", "rb") as bin_file:
                    binary_content = bin_file.read()
                    print(binary_content)
```

Funcionamiento Detallado de Instrucciones Tipo I:

def typeI(instruction):

opcode = opCodeTypeI[instruction[0]]

Se busca el OpCode correspondiente en el diccionario para la instrucción tipo I.

rs = int(instruction[2].replace("$", ""))

Se obtiene el valor del registro rs, eliminando el símbolo '$'.

rs = bin(rs)[2:].zfill(5)

Se convierte rs a binario y se asegura de que tenga 5 bits.

rt = int(instruction[1].replace("$", ""))

Se obtiene el valor del registro rt, eliminando el símbolo '$'.

rt = bin(rt)[2:].zfill(5)

Se convierte rt a binario y se asegura de que tenga 5 bits.

immediate = bin(int(instruction[3]))[2:].zfill(16)

Se convierte el valor inmediato a binario, se elimina '0b' y se asegura de tener 16 bits.

return opcode + str(rs) + str(rt) + immediate

Se concatenan todos los elementos para formar la representación binaria de la instrucción tipo I.

Explicación línea por línea:

1. **opcode = opCodeTypeI[instruction[0]]**: Se busca en el diccionario **opCodeTypeI** el valor correspondiente al OpCode de la instrucción actual (**instruction[0]** es el mnemónico de la instrucción, por ejemplo, "ADDI"). El resultado se almacena en la variable **opcode**.
2. **rs = int(instruction[2].replace("$", ""))**: Se obtiene el valor del registro rs desde la instrucción. Se elimina el símbolo '$' si está presente, y el resultado se almacena en la variable **rs**.
3. **rs = bin(rs)[2:].zfill(5)**: Se convierte el valor de **rs** a binario (**bin(rs)**), se eliminan los dos primeros caracteres ('0b') y se asegura de que tenga una longitud de 5 bits utilizando **zfill(5)**.
4. **rt = int(instruction[1].replace("$", ""))**: Similar a la línea 2, se obtiene el valor del registro rt desde la instrucción, eliminando el símbolo '$'. El resultado se almacena en la variable **rt**.
5. **rt = bin(rt)[2:].zfill(5)**: Similar a la línea 3, se convierte el valor de **rt** a binario y se asegura de que tenga una longitud de 5 bits.
6. **immediate = bin(int(instruction[3]))[2:].zfill(16)**: Se convierte el valor inmediato a binario, se eliminan los dos primeros caracteres ('0b') y se asegura de que tenga una longitud de 16 bits.
7. **return opcode + str(rs) + str(rt) + immediate**: Se concatenan los elementos (**opcode**, **rs**, **rt**, y **immediate**) para formar la representación binaria completa de la instrucción tipo I. El resultado se devuelve.

(Solo es la explicación de las instrucciones de Tipo I del decodificador).

A continuación, una breve explicación de las instrucciones tipo J, recién agregadas:

```
operationTypeJ = ["J"]

opCodeTypeJ = {
    "J": "000010"
}


def operationType(word): #Se selecciona de que tipo es la instruccion
    if word[0] in operationCodeTypeR: #Se busca dentro de las operaciones de tipo R
        r = typeR(word)
    elif word[0] in operationCodeTypeI:#Se busca dentro de las operacioness de tipo I
        r = typeI(word)
    elif word[0] in operationTypeJ:
        r = typeJ(word)
    else:
        r = "00000000000000000000000000000000"

    return r
```

Las instrucciones de tipo J (Jump) son utilizadas en arquitecturas de computadoras para realizar saltos incondicionales en el flujo de ejecución de un programa. Estas instrucciones permiten alterar la secuencia normal de ejecución y dirigirse a una dirección de memoria específica.

En el código que proporcionaste, la instrucción de tipo J es representada por la operación "J". A continuación, te explico brevemente cada parte de la instrucción de tipo J:

- Operación (Opcode): En el caso de las instrucciones de tipo J, el opcode específico para la operación "J" es "000010". Esto se almacena en el diccionario `opCodeTypeJ`.

- Dirección de destino: La dirección de destino es la dirección de memoria a la que se realizará el salto. En tu implementación, la dirección de destino se ingresa como un valor inmediato (por ejemplo, `"10"`), y se convierte a binario con `bin(int(instruction[1]))[2:].zfill(26)` para asegurar que tenga una longitud de 26 bits.

En resumen, una instrucción de tipo J consta del opcode correspondiente a la operación de salto (en este caso, "000010") y la dirección de destino de 26 bits.

Por ejemplo, si tu instrucción es `"J 10"`, en binario podría ser algo como: `"00001000000000000000000000001010"`. Esta instrucción provocaría un salto incondicional a la dirección de memoria 10.

Conclusiones:

El desarrollo de esta fase dentro de la implementación en verilog fue relativamente sencilla, al considerar solo 2 módulos adicionales y cambios en la unidad de control. Su implementación nos hará capaces de aprovechar mejor los componentes ya incluidos en la fase 1 para realizar tareas un poco más elaboradas que puedan contener saltos de instrucciones condicionados, carga y almacenamiento de datos e instrucciones inmediatas cuando no se necesite utilizar datos de memoria.

Referencias:

(n.d.). ARQUITECTURA MIPS. Retrieved 2023, from

https://www.infor.uva.es/~bastida/OC/TRABAJO2_MIPS.pdf

Fase 3

ARQUTECTURA DE COMPUTADORAS

- **Hermosillo Guizar Irvin Yael**
- **Jauregui Rabelero Hugo Johnathan**
- **Cruz Sanchez Juan Pablo**
- **Covarrubias Becerra Juan Pablo**

Maestro: Jorge Ernesto López Arce

Link GitHub: [HilloTS/Mips-Equipo-1 (github.com)]

Introducción Fase 3

Para concluir con el proyecto del procesador de 32 bits se tomaron las 2 fases anteriores, donde el procesador es capaz de realizar operaciones aritméticas y lógicas simples con las instrucciones de Tipo R, además de la implementación de las instrucciones de tipo I y la tipo J.

En esta fase se implementaron los buffers, los cuales son una región de almacenamiento temporal que se utiliza para guardar los datos mientras se transfieren de un lugar a otro. Funciona como un área intermedia que ayuda a gestionar las diferencias en las tasas de producción o consumo de datos entre dos sistemas o dispositivos, permitiendo una transferencia eficiente.

Además de la implementación de los buffers, se realizo un programa en ensamblador para probar el funcionamiento del mismo procesador.

El programa que pensamos desarrolla en el lenguaje ensamblador será un algoritmo en el cual se ingrese los grados en Celsius, y se puedan pasar a grados Kelvin y Fahrenheit realizando las formulas ya conocidas.

OBJETIVOS

Objetivo General

Diseñar, implementar y optimizar un procesador de 32 bits con arquitectura MIPS, capaz de ejecutar instrucciones tipo R, I y J, integrando módulos de buffer y ajustando conexiones para gestionar eficientemente el flujo de datos. Además, desarrollar un programa en ensamblador para realizar conversiones de temperatura.

Objetivos Particulares

-Agregar los módulos necesarios para completar el datapath de MIPS

de 32 bits para que sea capaz de ejecutar las instrucciones tipo J.

-Implementación de Instrucciones: Codificar en Verilog las instrucciones tipo R, I y J para el procesador MIPS, asegurando su correcta ejecución y manejo de datos.

-Decodificación de Instrucciones: Desarrollar un decodificador en Verilog para interpretar las instrucciones de ensamblador y dirigir adecuadamente las operaciones del procesador.

-Integración de Módulos de Buffer: Crear cuatro módulos de buffer con capacidades específicas para optimizar la gestión de datos entre diferentes partes del procesador, mejorando la eficiencia del flujo de información.

-Ajustes en Conexiones e Instancias: Modificar las conexiones y las instancias en el código en Verilog para adaptarse a la introducción de los

módulos de buffer, asegurando una integración coherente y funcional del sistema.

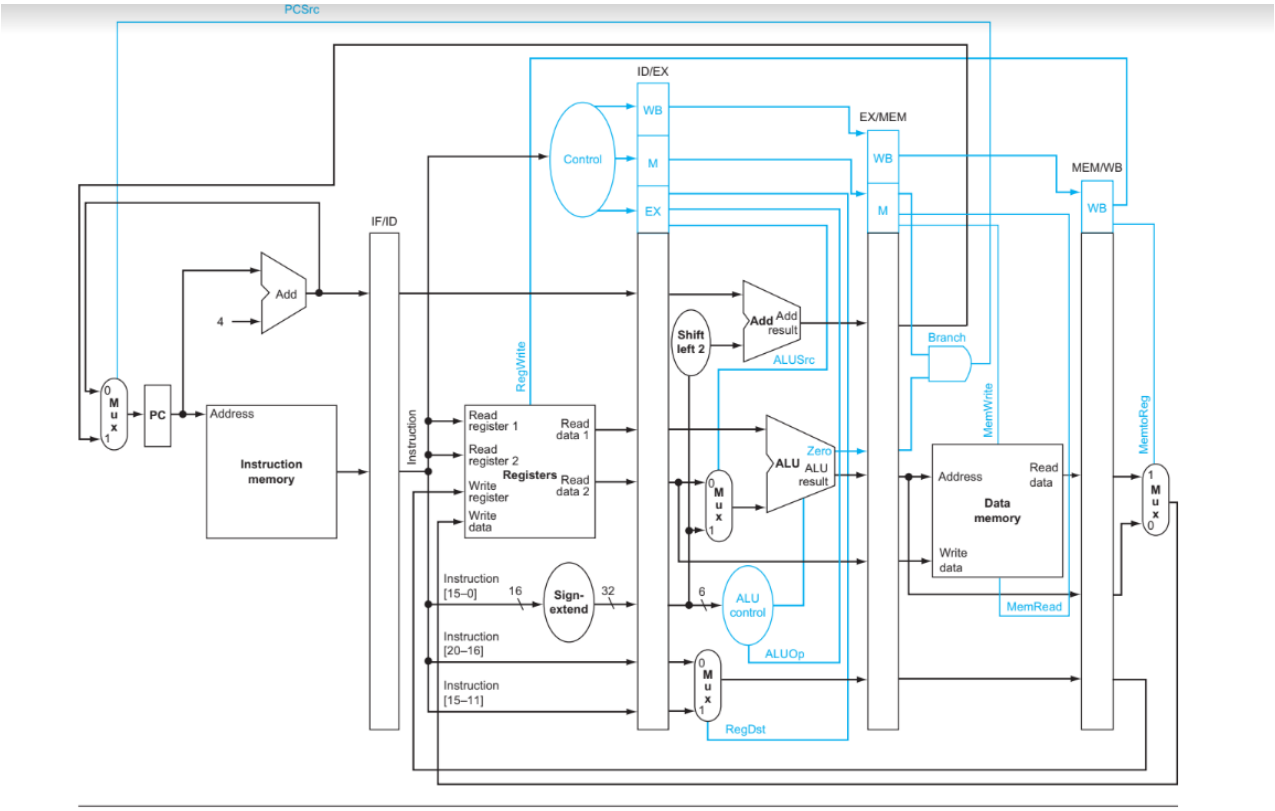
-Programa en Ensamblador: Diseñar un programa en ensamblador que utilice las capacidades del procesador, específicamente realizando conversiones de temperatura: de grados Celsius a Kelvin, de Kelvin a Fahrenheit y de Fahrenheit a Celsius.

-Validación Funcional: Realizar pruebas exhaustivas para validar el correcto funcionamiento del procesador, asegurando que las conversiones de temperatura sean precisas y que los módulos de buffer mejoren la eficiencia del flujo de datos.

-Optimización del Rendimiento: Realizar ajustes en el diseño del procesador y los módulos de buffer para optimizar el rendimiento general del sistema, buscando una ejecución eficiente de las operaciones.

-Documentación y Presentación: Elaborar documentación detallada del diseño, implementación y pruebas realizadas en cada fase del proyecto, preparando una presentación que destaque los logros y decisiones tomadas.

Desarrollo Fase 3 implementación Buffers.



Instruccion	Tipo	sintaxis
Add	R	Add $rd, $rs, $rt
Sub	R	
Mul	R	
Div	R	
Or	R	
And	R	
Xor	R	
slt	R	
nop	R	
Tabla 2	R	
	R	
	R	

Instruccion	Tipo	sintaxis
Lw	I	
Sw	I	
Slti	I	
beq	I	
j	J	
	J	
	I	
Tabla 1	I	
	I	
	I	

Desarrollo de Verilog:

Se implementaron 4 módulos específicos destinados a la gestión de los buffers. Estos módulos desempeñan un papel critico en la optimización de la trasferencia de datos entre diferentes partes del sistema. Cada uno se diseñó considerando las distintas entradas y salidas que deben de tener según el diagrama.

Además de los buffers, se tuvo que editar el modulo donde se hacen todas las instancias(conexiones) para adaptarlas a las entradas y salidas del buffer.

Desarrollo de los módulos nuevos en Verilog:

-Buffer IF/ID:

Este es el primer buffer el cual tiene 3 entradas: instructionmem(32 bits), pcAdder(32 bits) y clk(1 bit). Y 2 salidas: instruccionfetch(32 bits) y pcAdderFetch(32 bits)

Lo que reciben las entradas es la salida del primer sumador(pcAdder), las instrucciones en binario de la memoria de instrucciones(instructionmem) y la señal de reloj(clk) la cual será la señar que se utilizara para decirle al buffer cuando debe de dejar pasar a pcAdder e instructionmem.

Las salidas son registros a los cuales se les asignara el valor de instructionmem y pcAdder, cuando el valor de clk sea el apropiado.

Para realizar esto, primeramente, dentro de un initial begin le dimos valores predefinidos de 0 a nuestras 2 entradas de 32 bits y después, tenemos un bloque “always” el cual su sensibilidad de evento es “posedge clk”, lo que significa que ese bloque se ejecutara cada vez que se detecte un flanco de subido en la señal de reloj que tenemos como entrada. Dentro del bloque “always” ocurren las asignaciones de los valores de las entradas (instructionmem y pcAdder) a los registros de salida(pcAdderFetch y instructionfetch) dándole, por así decirlo, permiso al buffer para que mande las salidas su siguiente destino.

Una vez el buffer de salida, pcAdder se dirigirá directamente al buffer Id/Ex e instructionfetch entrara al banco de registros, unidad de control, signExtend, etc.

```
`timescale 1ns/1ns
module IF_ID(
    input clk,
    input [31:0]instructionmem,
    input [31:0]pcAdder,
    output reg [31:0]instruccionfetch,
    output reg [31:0]pcAdderFetch
);

initial begin
    instruccionfetch=0;
    pcAdderFetch=0;
end

always @(posedge clk)begin
    pcAdderFetch <= pcAdder;
    instruccionfetch <= instructionmem;
end

endmodule
```

-Buffer ID/EX

El segundo buffer a implementar tiene 9 entradas: WB(2 bits), M(3 bits), EX(5 bits), Next_Addres(32 bits), OP1(32 bits), OP2(32 bits), SignExt(32 bits), RT(5 bits), RD(5 bits) y clk(1 bit).

Y 11 salidas: O_WB(2 bits), O_M(3 bits),O_RegDst(1 bit),O_Aluop(3bits) O_Alusrc(1 bit), O_Next_Addres(32 bits), O_OP1(32 bits), O_OP2(32 bits), O_SignExt(32 bits), O_RT(5 bits), O_RD(5 bits)

Las entradas reciben el WB, M y EX de la unidad de control, NextAddress es la salida del primer buffer pcAddressFetch; OP1 y OP2 vienen del banco de registros, SingExt entra desde la salida del modulo “Sign extend” y; RT y RD se toman de la salida del buffer instructionfetch el tienen las instrucciones a realizar, exactamente se toman de los bits [26-16] para RT, y [15-11] para RD

Igual que en el buffer anterior su comportamiento va a depender de la señal de reloj de “clk” la cual se implementa en un bloque “always” el cual su sensibilidad de evento es “posedge clk” y dentro de este bloque se hacen las asignaciones de las entradas a sus respectivas salidas, aunque es importante resaltar que la entrada EX(5 bits) se divide en 3 salidas: RegDest, AluOp, y AluSrc.

Una vez el buffer tenga el permiso de dar salida a los datos, O_WB y O_M se hiran directamente al buffer Ex/Mem; RegDst se dirigirá al multiplexor para elegir si almacenar en RT o RD; AluOp ira directamente a la AluControl; AluSrc ira al multiplezor que elige si tomar la salida del signExtend o el OP2 para realizar la operación en la Alu; O_NextAddress ira al segundo sumador del ciclo fetch; O_OP1 ira directamente a la Alu; O_OP2 ira al multiplexor para ver si se elije a él o a la salida del signExtend; O_SignExtend ira a la Alu control para especificar la instrucción a realizar, al Mux para elegir entre el OP2 o él y al shift left 2 que se conecta con el Adder del ciclo fetch; y finalmente RD y RT irán al multiplexor el cual dependiendo del valor de RegDst se elegirá uno u otro para seleccionar la ubicación del registro en el banco de registros.

```
module ID_EX(
    input [31:0]OP1,
    input [31:0]OP2,
    input [31:0]SignExt,
    input [4:0]RT,
    input [4:0]RD,
    output reg [1:0]O_WB,
    output reg [2:0]O_M,
    output reg O_RegDst,
    output reg [2:0]O_Aluop,
    output reg O_Alusrc,
    output reg [31:0]O_Next_Adress,
    output reg [31:0]O_OP1,
    output reg [31:0]O_OP2,
    output reg [31:0]O_SignExt,
    output reg [4:0]O_RT,
    output reg [4:0]O_RD
);

always @(posedge clk)begin
    O_WB=WB;
    O_M = M;
    O_RegDst = EX[4];
    O_Aluop = EX[3:1];
    O_Alusrc = EX[0];
    O_Next_Adress = Next_Adress;
    O_OP1= OP1;
    O_OP2 = OP2;
    O_SignExt = SignExt;
    O_RT = RT;
    O_RD = RD;
end

endmodule
```

-Buffer EX/MEM

Este es el 3er buffer a implementar. Cuenta con 8 entradas: WB(1 bit), M(3bits), Add_Result(32 bits), zeroFlag(1 bits), Alu_Result(32 bits), OP2(32 bits), RegDestination(5 bits) y clk(1 bit). Y 9 salidas: o_wb(2 bits), Branch1(1 bit), MemtoRead(1 bit), MemtoWrite(1 bit), muxPc(32 bits), Branch2(1 bit), O_Alu_Result(32 bits), WriteData(32 bits), O_RegDestination(5 bits)

Las entradas de WB y M vienen de las salidas O_WB y O_M del buffer Id/Ex; Add_Result viene de la salida del modulo ADD el cual recibe como entrada la salida O_Next_Address del buffer Id/Ex y del modulo “Shift Left 2”; zeroFlag y Alu_result vienen de la salida del Adder que recibe la salida del buffer Id/Ex el O_OP1, y el O_OP2; OP2 recibe directamente la salida del buffer O_OP2; y el RegDestination viene del multiplexor el cual decide si utilizar como destino de almacenamiento RD o RT.

En el bloque “always” con sensibilidad de evento dependiendo de “posedge clk” se realizan las asignaciones correspondientes, aunque es importante aclarar que la entrada M(3 bits) se dividirá en 3 salidas: Branch1, MemtoRead y MemToWrite.

Las salidas se dirigirán a los siguientes moduios: o_wb ira directamente al siguiente buffer Mem/Wb, memToRead y memToWrite iran a la memoria para indicarle si se debe de leer o escribir; O_AluResult ira a la memoria para indicarle la dirección en donde almacenar el O_OP2 del buffer anterior que en este buffer es WriteData;O_RegDestiantion ira directamente al siguiente buffer, al Mem/Wb; Branch1, Branch2 iran al modulo Branch para realizar un AND, el valor de salida (PCSrc) ira al multiplexor que se conecta con PC; muxPC ira también al multiplexor que conecta con PC y dependiendo del valor que regrese el modulo Branch, se escogerá entre lo que llegue de muxPC o del primer sumador del ciclo fetch.

```
module EX_MEM(
    input clk,
    input [1:0]WB,
    input [2:0]M,
    input [31:0]Add_Result,
    input zeroflag,
    input [31:0]Alu_Result,
    input [31:0]OP2,
    input [4:0]RegDestination,
    output reg [1:0]o_wb,
    output reg Branch1,
    output reg MemtoRead,
    output reg MemtoWrite,
    output reg [31:0]muxPC,
    output reg Branch2,
    output reg [31:0]O_Alu_Result,
    output reg [31:0]WriteData,
    output reg [4:0]O_RegDestination
);


always @(posedge clk)begin
   o_wb=WB;
   Branch1 = M[2];
   MemtoRead = M[1];
   MemtoWrite = M[0];
   muxPC= Add_Result;
   Branch2 = zeroflag;
   O_Alu_Result = Alu_Result;
   WriteData = OP2;
   O_RegDestination = RegDestination;

end

endmodule
```

-Buffer MEM/WB

El utlimo buffer a implementar contara con 5 entradas: WB(2 bits), readData(32 bits), AluResult(32 bits), RegDestination(5 bits). Y 5 salidas: O_RegWrite(1 bit), O_MemToReg(1 bit), O_ReadDataOp(32 bits), O_op2(32 bits), O_RegDestination(5 bits)

La entrada WB viene de la salida O_wb del buffer Ex/Mem; readData viene de la salida de la memoria; AluResult viene de la salida O_Alu_Result del buffer Ex/Mem; y RegDestiantion viene directamente de la salida O_RegDestination del buffer Ex/Mem.

Como en todos los los buffers se tiene un bloque always con su evento de sensibilidad dictaminado por “posedge clk” y adentro se hacen las respectivas asignaciones de las entradas a las salidas.

Una vez dando la indicación de “dejar salir” los datos, O_RegWrite servirá para decirle al banco de registros si debe de escribir, O_MemToReg servirá como bandera del ultimo multiplexor, el que se conecta con la memoria, servirá como bandera para decidir cual valor se toma entre O_ReadDataOp u O_Op2; y finalmente O_RegDestination le indicara al banco de registros en que registro se debe de “guardar” el dato

```
module MEM_WB(
    input clk,
    input [1:0]WB,
    input [31:0]readData,
    input [31:0]AluResult,
    input [4:0]RegDestination,
    output reg O_RegWrite,
    output reg O_MemtoReg,
    output reg [31:0]O_ReadDataop,
    output reg [31:0]O_op2,
    output reg [4:0]O_RegDestination
);


always @(posedge clk)begin
    O_RegWrite = WB[0];
    O_MemtoReg = WB[1];
    O_ReadDataop = readData;
    O_op2 = AluResult;
    O_RegDestination = RegDestination;
end

endmodule
```

Instancias(ProyectoTb)

Con la implementación de los 4 buffers anteriores se tuvieron que realizar cambios en el modulo ProyectoTb, que es donde se realizan todas las conexiones necesarias para el funcionamiento del procesador. Los cambios en las instancias son los que se explicaron en cada Buffer.

```
`timescale 1ns/1ns
module proyect();

reg clk_tb = 1'b0;
/*Cables del Banco*/
wire [31:0]C1;//Dato2
wire [31:0]C2;//Dato1

/*Elementos de la ALU*/
wire [31:0]C3;//Resultado de la ALU
wire Flag;//Es el zeroflag

/*Operacion que se selecciona en el Alu control*/
wire [3:0]C4;

/*Data Memory*/
wire [31:0]C5;//Es lo que sale de la memoria

wire [31:0]respMux;

/*Cables para las salidas de la Unidad de Control*/
wire RegDstValue;
wire BranchValue;
wire AlusrcValue;
wire bankwrite;
wire selmemWrt;
wire selmemRed;
wire [2:0]Alupocion;
wire selMux;

wire [31:0]Mux31Alu;//Cable del MUx que le da el segundo operador a la ALU para que haga la operacion

wire [31:0]PcOutValue;//Es lo que saca la PC
wire [31:0]fueraValue; //El resultado del Adder del pc cuando suma 4
wire [31:0]resultMuxValue;
wire [31:0]instruccionTR; //Es la instruccion que sale de la memoria de instrucciones
```

```
/*Cables del Mux de 31 bits que le da el dato a escribir al banco de registros*/
wire [31:0]Muxmembankvalue;//El dato que vamos a escribir en el banco de registros
/*Cables de Mux 5 bits a Banco*/
wire [4:0]Mux5result;//Es donde dice donde vamos a escribir
/*Componentes del Shiftleft*/
wire [31:0]Shift_leftinput;
wire [31:0]AdderShiftsrc;
/*Cable del Adder dos de 31 bits al Mux de 31 bits*/
wire [31:0]MuxAdderdossrc;
/*Mux con la compuerta AND del branch y el zeroflag*/
reg MuxFetchsrc;


always #50 clk_tb = ~clk_tb;

/*Cables Instruction Fetch (IF)*/
wire [31:0] valuePcAdder;
wire [31:0] valueInstruction_IF;


/*Cables Instruction Decode (ID/EX)*/
wire [1:0]O_WB_input;
wire [2:0]O_M_input;
wire O_RegDst_input;
wire [2:0]O_Aluop_input;
wire O_Alusrc_input;
wire [31:0]O_Next_Adress_input;
wire [31:0]O_OP1_input;
wire [31:0]O_OP2_input;
wire [31:0]O_SignExt_input;
wire [4:0]O_RT_input;
wire [4:0]O_RD_input;
//
```

```
/*Cables de Write Back (MEM/WB)*/
wire O_RegWrite_WB;
wire O_MemtoReg_WB;
wire [31:0]O_ReadDataop_WB;
wire [31:0]O_op2_WB;
wire [4:0]O_RegDestination_WB;

/*Cables Execution /Adress Calculation (EX/MEM)*/
wire [1:0]o_wb_input;
wire O_Branch1;
wire O_MemtoRead_EX;
wire O_MemtoWrite_EX;
wire [31:0]O_muxPC;
wire O_Branch2;
wire [31:0]O_Alu_Result_EX;
wire [31:0]O_WriteData;
wire [4:0]O_RegDestination_EX;

assign MuxFetchsrc = O_Branch2 & O_Branch1;
```

```
/*Mux que va al PC terminar me falta este*/
MuxAdder muxadd1(
    .val0(MuxAdderdossrc),
    .val1(O_muxPC),
    .selector(MuxFetchsrc),
    .resultMUx(resultMuxValue)
);


Adder adderuno(
    .suma(4),
    .numprimero(PcOutValue),
    .fuera(fueraValue)
);

Pc pc1(
    .PcSrc(resultMuxValue),
    .clk(clk_tb),
    .PcOut(PcOutValue)
);

Memoriainstrucciones mem1(
    .RedAddress(PcOutValue),
    .instruction_mem(instruccionTR)
);

IF_ID buffIF_ID(
    .clk(clk_tb),
    .pcAdder(fueraValue),
    .instructionmem(instruccionTR),
    .instruccionfetch(valueInstruction_IF),
    .pcAdderFetch(valuePcAdder)
);
```

```
bank banco_uno(
    .AR1(valueInstruction_IF[25:21]),
    .AR2(valueInstruction_IF[20:16]),
    .AW(O_RegDestination_WB),
    .Wen(O_RegWrite_WB),
    .Datow(Muxmembankvalue),
    .Dato1(C2),
    .Dato2(C1)
);

ControlUnit unidad_uno(
    .instruction(valueInstruction_IF[31:26]),
    .RegDst(RegDstValue),
    .Branch(BranchValue),
    .MemtoRead(selmemRed),
    .MemtoWrite(selmemWrt),
    .ALUop(Alupocion),
    .ALUSrc(AlusrcValue),
    .RegWrite(bankwrite),
    .MemToReg(selMux)
);
```

```
ID_EX buffID_EX(
    .clk(clk_tb),
    .WB({bankwrite,selMux}),
    .M({BranchValue,selmemRed,selmemWrt}),
    .EX({RegDstValue,Alupocion,AlusrcValue}),
    .Next_Adress(valuePcAdder),
    .OP1(C2),
    .OP2(C1),
    .SignExt(Shift_leftinput),
    .RT(valueInstruction_IF[20:16]),
    .RD(valueInstruction_IF[15:11]),
   //SALIDAS
    .O_WB(O_WB_input),
    .O_M(O_M_input),
    .O_RegDst(O_RegDst_input),
    .O_Aluop(O_Aluop_input),
    .O_Alusrc(O_Alusrc_input),
    .O_Next_Adress(O_Next_Adress_input),
    .O_OP1(O_OP1_input),
    .O_OP2(O_OP2_input),
    .O_SignExt(O_SignExt_input),
    .O_RT(O_RT_input),
    .O_RD(O_RD_input)
);

/*Hecho*/
alu alu_uno(
    .op1(O_OP1_input),
    .op2(Mux31Alu),
    .sel(C4),
    .Resultado(C3),
    .zeroflag(Flag)
);
```

```
/*Mux que va a la ALU*/
/*Hecho*/
Mux31bits muxbancoAlu(
    .valor0(O_OP2_input),
    .valor1(O_SignExt_input),
    .select(O_Alusrc_input),
    .ValOut31(Mux31Alu)
);
/*Hecho*/
ALU_Control alucontrol_uno(
    .functionfield(O_SignExt_input[5:0]),
    .Alu_op(O_Aluop_input),
    .operacion(C4)
);


/*Para ver donde se va a guardar en el banco de registros*/
/*Hecho*/
Mux5bits muxinstrucbanco(
    .Valoren0(O_RT_input),
    .Valoren1(O_RD_input),
    .selector(O_RegDst_input),
    .WriteReg(Mux5result)
);
```

```
EX_MEM bufferEXMEM(
    .clk(clk_tb),
    .WB(O_WB_input),
    .M(O_M_input),
    .Add_Result(MuxAdderdossrc),
    .zeroflag(Flag),
    .Alu_Result(C3),
    .OP2(O_OP2_input),
    .RegDestination(Mux5result),
    //SALIDAS
    .o_wb(o_wb_input),
    .Branch1(O_Branch1),
    .MemtoRead(O_MemtoRead_EX),
    .MemtoWrite(O_MemtoWrite_EX),
    .muxPC(O_muxPC),
    .Branch2(O_Branch2),
    .O_Alu_Result(O_Alu_Result_EX),
    .WriteData(O_WriteData),
    .O_RegDestination(O_RegDestination_EX)
);

/*Hecho*/
Memoria memoria_uno(
    .Wen(O_MemtoWrite_EX),
    .Adress(O_Alu_Result_EX),
    .DataW(O_WriteData),
    .Ren(O_MemtoRead_EX),
    .DataR(C5)
);
```

```
MEM_WB buffMEMWB(
    .clk(clk_tb),
    .WB(o_wb_input),
    .readData(C5),
    .AluResult(O_Alu_Result_EX),
    .RegDestination(O_RegDestination_EX),
    //SALIDAS
    .O_RegWrite(O_RegWrite_WB),
    .O_MemtoReg(O_MemtoReg_WB),
    .O_ReadDataop(O_ReadDataop_WB),
    .O_op2(O_op2_WB),
    .O_RegDestination(O_RegDestination_WB)
);

/*Mux que va con MEMWB*/
/*Hecho*/
Mux31bits muxmembank(
    .valor0(O_op2_WB),
    .valor1(O_ReadDataop_WB),
    .select(O_MemtoReg_WB),
    .ValOut31(Muxmembankvalue)
);

/*Hecho*/
Sign_Extend signuno(
    .Instruccion(valueInstruction_IF[15:0]),
    .Instruction_config(Shift_leftinput)
);

/*Hecho*/
Shift_left shiftuno(
    .Signextend_src(O_SignExt_input),
    .result_shift(AdderShiftsrc)
);
```

```
/*Hecho*/
Adder adderdos(
    .suma(AdderShiftsrc),
    .numprimero(O_Next_Adress_input),
    .fuera(MuxAdderdossrc)
);


endmodule
```

Desarrollo del Programa de Ensamblador:

Lamentablemente no fuimos capaces de adaptar este código de ensamblador para que pudiera ser utilizado por nuestro decodificador. Por lo que no pudimos usarlo en el procesador de 32 bits que desarrollamos.

Conclusiones:

En la Fase 3 de nuestro proyecto de construcción de un procesador de 32 bits con arquitectura MIPS, hemos logrado una integración exitosa de cuatro módulos de buffer. Estos módulos desempeñan un papel crítico en la gestión eficiente del flujo de datos dentro del procesador. Se realizaron ajustes en las conexiones y las instancias para adaptarse a estos módulos, asegurando una transición fluida de datos entre los componentes.

Sin embargo, no fuimos capaces de implementar un programa en ensamblador que aprovecha las capacidades del procesador, realizando conversiones de temperatura: de grados Celsius a Kelvin, de Kelvin a Fahrenheit y de Fahrenheit a Celsius.

En conclusión, la Fase 3 representa un hito crucial, consolidando la eficiencia en la gestión de datos y dando como concluido el proyecto, donde se espera un rendimiento sólido y confiable del procesador en su conjunto.

Referencias:

(n.d.). ARQUITECTURA MIPS. Retrieved 2023, from

https://www.infor.uva.es/~bastida/OC/TRABAJO2_MIPS.pdf