UNIVERSIDAD DE GUADALAJARA.
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS.

MATERIA:
S. S. P. DE ARQUITECTURA DE COMPUTADORAS

SECCIÓN: D11

PROYECTO FINAL FASE 3

ALUMNOS:

NATALIA ISABEL MARISCAL NAPOLES
MARÍN GONZÁLEZ ANDRÉ JOSUÉ
JUAN SILVERO VALENCIA
ERICK JARED GUTIERREZ CORREA

DOCENTE:

J. Ernesto López Arce Delgado

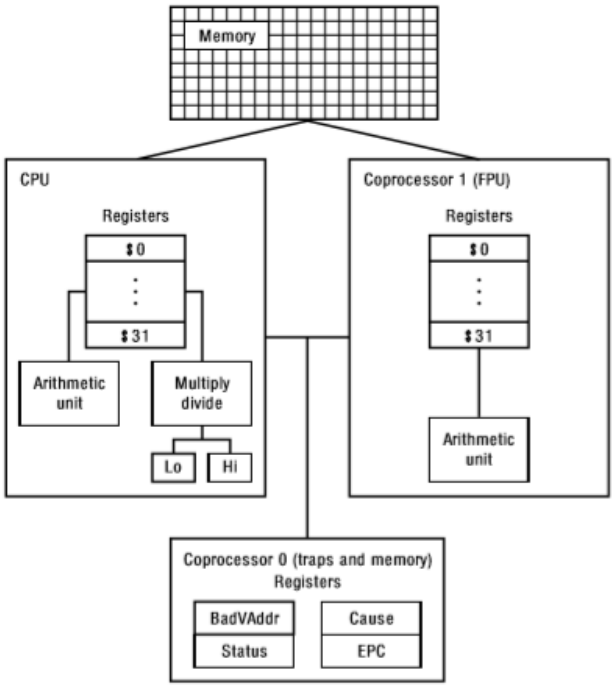
CARRERA:

ING. INFORMÁTICA

2023

INTRODUCCIÓN.

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

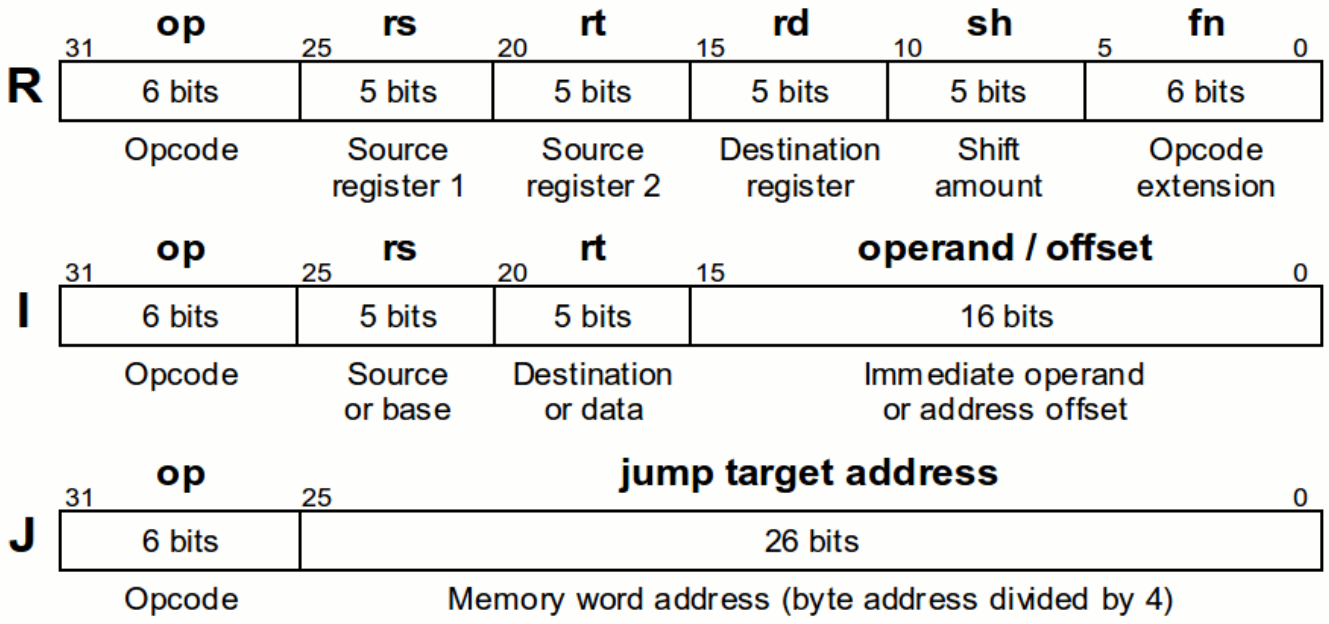


Nos podemos encontrar arquitecturas MIPS en muchos sistemas embebidos, en dispositivos para Windows CE, routers Cisco y consolas como la Nintendo 64 o las PlayStation, PlayStation 2 y PSP.

CARACTERÍSTICAS DE LA ARQUITECTURA DEL MIPS

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

LAS INSTRUCCIONES EN MIPS LAS PODEMOS ENCONTRAR EN 3 FORMATOS



Instrucciones Tipo J: utilizado por las instrucciones de bifurcación

Inst.	Definición	Formato
j	Modifica el valor del PC para ejecutar la instrucción siguiente a la etiqueta	j etiqueta
jal	Modifica el valor del PC por aquel al que apunta la etiqueta y almacena la dirección actual del PC en $ra	jal etiqueta

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

Inst.	Definición	Formato
addi	Se almacena en el registro $1 la suma del registro $2 y el valor de la constante	addi $1, $2, constante
andi	El registro $1 tiene el resultado de una operación AND entre $2 y la constante	andi $1, $2, constante
ori	El registro $1 tiene el resultado de una operación OR entre $2 y la constante	ori $1, $2, constante
slti	Almacena en $1 un 1 si el valor registro $2 es menor que el de la constante. En caso contrario almacena un 0	slti $1, $1, constante
lui	Carga constante en los 16 bits superiores del registro $1	lui $1, direcConst
lw	Carga en el registro $1 la palabra almacenada en la dirección de memoria que contiene el registro $2 más el desplazamiento. La nueva dirección calculada debe ser múltiplo de cuatro	lw $1, desplazamiento($2)
lb	Carga en el registro $1 el byte de memoria apuntado por la dirección almacenada en el registro $2 más el desplazamiento	lb $1, desplazamiento($2)
sw	Almacena en memoria en la posición obtenida de sumarle el desplazamiento a la dirección del registro $2, la palabra del registro $1. La dirección debe ser múltiplo de 4	sw $1, desplazamiento($1)
sb	Almacena en la posición de memoria correspondiente al valor de $2 más el desplazamiento, el primer byte de la palabra almacenada en $1	sb $1, desplazamiento($2)
beq	Si el valor de $1 y $2 es igual se modifica el valor del PC para pasar a ejecutar el trozo de código apuntado por la etiqueta	beq $1, $1, etiqueta
bne	Si el valor de $1 y $2 no es igual se modifica el valor del PC para pasar a ejecutar el trozo de código apuntado por la etiqueta	bne $1, $1, etiqueta

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

Inst.	Definición	Formato
add	Almacena en el registro $1 el valor de la suma de los otros dos	add $1, $2, $3
sub	Almacena en el $1 el valor de la resta de $2 (minuendo), menos el $3 (sustraendo)	sub $1, $2, $3
and	Almacena en el $1 el resultado de hacer una operación AND entre los otros dos registros	and $1, $2, $3
or	Almacena en el $1 el resultado de hacer una operación OR entre los registros $2 y $3	or $1, $2, $3
slt	Coloca un 1 en el registro $1 si el valor almacenado en $2 es menor que el de $3. Si no lo es se almacena un 0 en $1.	slt $1, $2, $3
jr	Modifica la dirección del PC por aquel valor almacenado en el registro $1	jr $1
sll	Almacena en el registro $1 el valor del registro $2 desplazado n bits a la izquierda	sll $1, $2, n
srl	Almacena en el registro $1 el valor del registro $2 desplazado n bits a la derecha	srl $1, $2, n
sra	Almacena en el registro $1 el valor del registro $2 después de hacer un desplazamiento aritmético de n bits a la derecha	sra $1, $2, n

Objetivo

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

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

Debe de elegir un algoritmo previamente aprobado por su profesor, y que sea posible implementar con el set reducido de instrucciones de la tabla 1 y 2. Este programa previamente definido en ensamblador debe ser codificado a código binario y precargado en la memoria de instrucciones para que el datapath lo ejecute, recuerde definir cada uno de los aspectos de dicho programa, secciones de los registros en el banco de registros para base pointers, resultados, resultado de comparaciones, etc. Asi como los datos pre-cargados en su memoria de datos.

Código de Ensamblador en Phyton

```
from tkinter import *
from tkinter import filedialog
import io
```

1. **from tkinter import :**
    - Importa todos los elementos (clases, funciones, etc.) de la biblioteca tkinter. Esto incluye todo lo necesario para crear y gestionar ventanas, widgets (como botones y cuadros de texto), y otros elementos de una interfaz gráfica.
2. **from tkinter import filedialog:**
    - Importa el módulo filedialog de tkinter, que proporciona funciones para mostrar cuadros de diálogo de apertura y guardado de archivos. Estos cuadros de diálogo permiten a los usuarios seleccionar archivos y carpetas.
3. **import io:**
    - Importa el módulo io de la biblioteca estándar de Python. io proporciona herramientas para trabajar con corrientes de entrada/salida (streams), como leer o escribir en búferes de memoria (BytesIO y StringIO).

```
def vistaPrevia():
    archivo=open(ruta.cget("text"), "r")
    vistaArchivo.config(state=NORMAL)
    vistaArchivo.delete('1.0', END)
    for linea in archivo:
        datos=""
        if(("lw" in linea) or ("sw" in linea)):
            linea=linea.replace(' ', '')
            linea=linea.replace('#', ',')
            linea=linea.replace('$', ',')
            linea=linea.replace('(', '')
            linea=linea.replace(')', '')
            numeros=linea.split(',')
            numeros.pop(0)
        else:
            linea=linea.replace(',', '')
            linea=linea.replace('#', ',')
            linea=linea.replace('$', ',')
            numeros=linea.split(',')
            numeros.pop(0)
        #addi
        if("addi" in linea):
            linea=linea.replace('addi', '')
            datos+="001000"
            datos+=convertirBinarioR(int(numeros[0]))
            datos+=convertirBinarioR(int(numeros[1]))
            datos+=convertirBinarioI(int(numeros[2]))
            datos+="\n"
            vistaArchivo.insert(INSERT, datos)
        #add
        if("add" in linea):
            linea=linea.replace('add', '')
            datos+="000000"
            datos+=convertirBinarioR(int(numeros[1]))
            datos+=convertirBinarioR(int(numeros[2]))
            datos+=convertirBinarioR(int(numeros[0]))
            datos+="xxxxx100000\n"
            vistaArchivo.insert(INSERT, datos)
```

Esta función se encarga de realizar una vista previa de un archivo, interpretar las líneas y convertirlas en instrucciones MIPS. La entrada es el contenido de un archivo de texto, y la salida se inserta en un widget de texto (vistaArchivo).

El código procesa cada línea del archivo, identifica el tipo de instrucción MIPS y genera el código binario correspondiente.

La función utiliza la variable ruta para obtener la ruta del archivo seleccionado mediante un cuadro de diálogo proporcionado por filedialog. Es probable que exista código adicional que llame a este fragmento de código cuando se selecciona un archivo.

```
def devolverRuta():
    direccion=filedialog.askopenfilename(initialdir="",
                                         title="Selecciona un archivo asm",
                                          filetypes=(("Texto", "*.txt"),
                                                     ("Todos", "*.*")))
    if(direccion=="" and ruta.cget("text")=="Ruta del archivo"):
        ruta.config(text="Ruta del archivo")
    elif(direccion!=""):
        ruta.config(text=direccion)
        vistaPrevia()


def convertirArchivo():
    archivoBinario=open("instruccionesBinarias.txt", "w")
    texto=vistaArchivo.get('1.0', END).splitlines()
    contador=0
    for line in texto:
        entero=0
        for i in line:
            if(contador==8):
                contador=0
                archivoBinario.write("\n")
            contador+=1
            archivoBinario.write(line[entero])
            entero+=1
    vistaArchivo.config(state=NORMAL)
    vistaArchivo.delete('1.0', END)
    vistaArchivo.insert(INSERT, 'Tu archivo se creo con exito\n')
    vistaArchivo.config(state=DISABLED)
    archivoBinario.close()
```

1. **devolverRuta:**
    - Esta función esta diseñada para obtener la ruta de un archivo mediante un cuadro de diálogo proporcionado por filedialog.askopenfilename.
    - Se establece un directorio inicial (initialdir) como vacío, el título del cuadro de diálogo es "Selecciona un archivo asm", y se especifica que se deben mostrar archivos de texto (*.txt) o todos los archivos (*.*).
    - Si el usuario selecciona un archivo (direccion != ""), se actualiza el texto del widget llamado ruta con la ruta seleccionada y luego se llama a la función vistaPrevia.
2. **convertirArchivo:**
    - Esta función esta diseñada para convertir el contenido del widget vistaArchivo a un archivo binario llamado "instruccionesBinarias.txt".
    - Se abre el archivo en modo de escritura ("w").
    - Luego, se obtiene el contenido de vistaArchivo línea por línea y se itera a través de cada carácter.
    - Se escribe cada carácter en el archivo binario, y cuando se alcanza un contador de 8 caracteres, se inserta un salto de línea.
    - Después de escribir el contenido, se limpia el contenido de vistaArchivo y se le inserta un mensaje indicando que el archivo binario se creó con éxito.

```
def convertirBinarioR(num):
    if(num>31):
        print("Tu numero excede el limite\n")
        return '11111'
    elif(num==0):
        return '00000'
    binario=''
    while num-1 != 0:
        if(num%2 ==0):
            binario+='0'
            num=num/2
        else:
            binario+='1'
            num=int(num/2)
    binario+='1'
    while(len(binario)<5):
        binario+='0'
    return binario[::-1]

def convertirBinarioI(num):
    if(num>65535):
        print("Tu numero excede el limite\n")
        return '1111111111111111'
    elif(num==0):
        return '0000000000000000'
    binario=''
    while num-1 != 0:
        if(num%2 ==0):
            binario+='0'
            num=num/2
        else:
            binario+='1'
            num=int(num/2)
    binario+='1'
    while(len(binario)<16):
        binario+='0'
    return binario[::-1]
```

1. **convertirBinarioR(num):**
    - Esta función convierte un número entero num en una representación binaria de 5 bits, que es comúnmente utilizada para los registros en el conjunto de instrucciones MIPS.

- Si num excede el límite de 31, imprime un mensaje de advertencia y devuelve '11111'.
- Si num es igual a 0, devuelve '00000'.
- En caso contrario, realiza la conversión binaria del número num.
  - Utiliza un bucle para obtener los bits de la representación binaria y construir la cadena binario.
  - Asegura que la cadena binaria tenga una longitud de 5 bits, agregando ceros a la izquierda si es necesario.
  - Devuelve la representación binaria invertida, ya que se está construyendo desde los bits menos significativos.

2. **convertirBinarioI(num):**
   - Similar a convertirBinarioR, esta función convierte un número entero num en una representación binaria de 16 bits, que es comúnmente utilizada para las instrucciones de tipo I en MIPS.
   - Si num excede el límite de 65535, imprime un mensaje de advertencia y devuelve '1111111111111111'.
   - Si num es igual a 0, devuelve '0000000000000000'.
   - Realiza la conversión binaria de num utilizando un bucle similar al de convertirBinarioR.
   - Asegura que la cadena binaria tenga una longitud de 16 bits, agregando ceros a la izquierda si es necesario.
   - Devuelve la representación binaria invertida.

```
def convertirBinarioJ(num):
    if(num==0):
        return '00000000000000000000000000'
    binario=''
    while num-1 != 0:
        if(num%2 ==0):
            binario+='0'
            num=num/2
        else:
            binario+='1'
            num=int(num/2)
    binario+='1'
    while(len(binario)<26):
        binario+='0'
    return binario[::-1]

ventana=Tk()
ventana.title("Decodificador")
ventana.resizable(False, False)
```

3. **convertirBinarioJ(num):**
    - Esta función convierte un número entero num en una representación binaria de 26 bits, que es comúnmente utilizada para las instrucciones de tipo J en el conjunto de instrucciones MIPS.
    - Si num es igual a 0, devuelve '00000000000000000000000000'.
    - Realiza la conversión binaria de num utilizando un bucle similar al de las funciones anteriores.
    - Asegura que la cadena binaria tenga una longitud de 26 bits, agregando ceros a la izquierda si es necesario.
    - Devuelve la representación binaria invertida.

La función utiliza un enfoque de división sucesiva para convertir el número en binario y asegura que la cadena binaria resultante tenga la longitud deseada.

```
ventana=Tk()
ventana.title("Decodificador")
ventana.resizable(False, False)

frame=Frame(ventana)
frame.pack()

titulo=Label(frame, text="Busca el archivo para convertir", width="45")
titulo.grid(row=0, column=0, columnspan=3)
titulo.config(fg="blue", font="12")
titulo.config(padx=10, pady=10)

selector=Button(frame, text="Buscar archivo", command=devolverRuta, width="15")
selector.grid(row=1,column=0, sticky="ne")
selector.config(activeforeground="blue", bd="5")

ruta=Label(frame, text="Ruta del archivo", width="37")
ruta.grid(row=1, column=1, columnspan=2, sticky="w")
ruta.config(bg="lightgrey", fg="blue")

aplicar=Button(frame, text="Exportar", command=convertirArchivo, width="15")
aplicar.grid(row=2, column=0, sticky="ne")
aplicar.config(activeforeground="red", bd="5")

vistaArchivo=Text(frame, width="35", height="10")
vistaArchivo.grid(row=2, column=1)
vistaArchivo.config(bg="lightgrey")

scroll=Scrollbar(frame, command=vistaArchivo.yview)
scroll.grid(row=2, column=2, sticky="nsew")

vistaArchivo.config(yscrollcommand=scroll.set)

ventana.mainloop()
```

Creación de la Ventana (`Tk()`):

- Se crea una instancia de la clase `Tk`, que representa la ventana principal de la interfaz gráfica.
- Se establece el título de la ventana como "Decodificador".

- Se hace que la ventana no sea redimensionable tanto horizontal como verticalmente.

Creación de un Marco (`Frame`) y Empaquetado:

- objeto Frame que actuará como un contenedor para otros widgets.
- El marco se empaqueta en la ventana principal.

Creación de Etiquetas y Botones:

```
python
titulo=Label(frame, text="Busca el archivo para convertir", width="45")
titulo.grid(row=0, column=0, columnspan=3)
titulo.config(fg="blue", font="12")
titulo.config(padx=10, pady=10)
```

- Se crea una etiqueta (Label) con el texto "Busca el archivo para convertir" y otras configuraciones.
- Se coloca la etiqueta en la primera fila (row=0) y ocupa tres columnas (column=0, columnspan=3).

```
selector=Button(frame, text="Buscar archivo", command=devolverRuta,
width="15")
selector.grid(row=1,column=0, sticky="ne")
selector.config(activeforeground="blue", bd="5")
```

- Se crea un botón (Button) con el texto "Buscar archivo" y se asocia con la función devolverRuta cuando se hace clic.
- Se coloca el botón en la segunda fila (row=1) y primera columna (column=0) con opciones de alineación y configuraciones adicionales.

```
ruta=Label(frame, text="Ruta del archivo", width="37")
ruta.grid(row=1, column=1, columnspan=2, sticky="w")
ruta.config(bg="lightgrey", fg="blue")
```

- Se crea otra etiqueta (Label) con el texto "Ruta del archivo" y otras configuraciones.
- Se coloca la etiqueta en la segunda fila (row=1) y ocupa dos columnas (column=1, columnspan=2) con alineación a la izquierda.

```
aplicar=Button(frame, text="Exportar", command=convertirArchivo,
width="15")
aplicar.grid(row=2, column=0, sticky="ne")
aplicar.config(activeforeground="red", bd="5")
```

- Se crea otro botón (Button) con el texto "Exportar" y se asocia con la función convertirArchivo cuando se hace clic.
- Se coloca el botón en la tercera fila (row=2) y primera columna (column=0) con opciones de alineación y configuraciones adicionales.

Creación de un Área de Texto (`Text`) y Barra de Desplazamiento:

```
vistaArchivo=Text(frame, width="35", height="10")
vistaArchivo.grid(row=2, column=1)
vistaArchivo.config(bg="lightgrey")
```

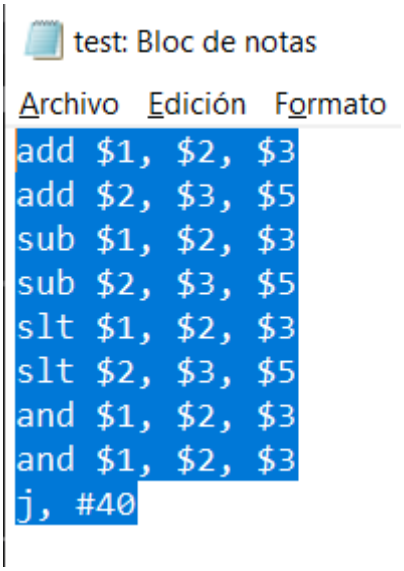
- Se crea un widget de texto (Text) con un ancho de 35 caracteres y una altura de 10 líneas.
- Se coloca el widget de texto en la tercera fila (row=2) y segunda columna (column=1).

```
scroll=Scrollbar(frame, command=vistaArchivo.yview)
scroll.grid(row=2, column=2, sticky="nsew")
```

- Se crea una barra de desplazamiento (Scrollbar) y se asocia con el método yview del widget de texto.
- Se coloca la barra de desplazamiento en la tercera fila (row=2) y tercera columna (column=2) con opciones de alineación.

```
vistaArchivo.config(yscrollcommand=scroll.set)
```

- Se configura el widget de texto para usar la barra de desplazamiento vertical.



Cada línea representa una instrucción específica que se ejecutará en una arquitectura MIPS. A continuación, se describe cada instrucción:

1. **`add $1, $2, $3`**
   - Suma el contenido de los registros `$2` y `$3` y almacena el resultado en el registro `$1`.
2. **`add $2, $3, $5`**
   - Suma el contenido de los registros `$3` y `$5` y almacena el resultado en el registro `$2`.
3. **`sub $1, $2, $3`**
   - Resta el contenido de los registros `$2` y `$3` y almacena el resultado en el registro `$1`.
4. **`sub $2, $3, $5`**
   - Resta el contenido de los registros `$3` y `$5` y almacena el resultado en el registro `$2`.
5. **`slt $1, $2, $3`**
   - Establece el registro `$1` en 1 si el contenido de `$2` es menor que el contenido de `$3`; de lo contrario, establece `$1` en 0.
6. **`slt $2, $3, $5`**
   - Establece el registro `$2` en 1 si el contenido de `$3` es menor que el contenido de `$5`; de lo contrario, establece `$2` en 0.
7. **`and $1, $2, $3`**
   - Realiza una operación lógica AND entre los contenidos de los registros `$2` y `$3` y almacena el resultado en el registro `$1`.
8. **`and $1, $2, $3`**
   - Nota: Esta instrucción es idéntica a la anterior; podría ser un error tipográfico o duplicación.
9. **`j, #40`**
   - Salta a la dirección de memoria especificada, en este caso, a la dirección de memoria 40.

Módulos de verilog

26_28_Shift_Left_2.v

```
module Shift_Left_26_28(
input [25:0]SL_in,
output reg[27:0]SL_out
);

always @*
    begin
       SL_out = SL_in << 2;
    end
endmodule
```

Parámetros del módulo:

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

Bloque always @*:

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

Bloque de código dentro de always @*:

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

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

ADD.v

```
module ADDER(
input [31:0] O1,
input [31:0] O2,
output reg [31:0] Res);
```

```
always@*
begin
    Res = O1 + O2;
end

endmodule
```

Parámetros del módulo:

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

Bloque always @*:

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

Bloque de código dentro de always @*:

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

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

Buffer 1_IFID.v

```
//BUFFER 1: IF/ID
module IFID(
input clk,
input [31:0]Next_address,
input [31:0]Instruction,
output reg [31:0]O_Next_address,
output reg [31:0]O_Instruction);

always @(posedge clk)
begin
    O_Next_address = Next_address;
        O_Instruction = Instruction;
end

endmodule
```

Parámetros del módulo:

- input clk: Es una entrada que representa la señal de reloj.
- input [31:0] Next_address: Es una entrada de 32 bits que representa la dirección de la siguiente instrucción en el pipeline.
- input [31:0] Instruction: Es otra entrada de 32 bits que representa la instrucción actual en el pipeline.
- output reg [31:0] O_Next_address: Es una salida de 32 bits que retiene la dirección de la siguiente instrucción.
- output reg [31:0] O_Instruction: Es otra salida de 32 bits que retiene la instrucción actual.

Bloque always @(posedge clk):

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

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

- O_Next_address = Next_address;: En cada flanco de subida del reloj, la salida O_Next_address se actualiza con el valor presente en la entrada Next_address.
- O_Instruction = Instruction;: De manera similar, la salida O_Instruction se actualiza con el valor presente en la entrada Instruction en cada flanco de subida del reloj.

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

Buffer 2_IDEX.v

```
//BUFFER 2: ID/EX
module IDEX(
input clk,
input [1:0]I_WB,
input [2:0]I_M,
input [4:0]I_EX,
input [31:0]I_Next_address,
input [31:0]I_O1,
input [31:0]I_O2,
input [31:0]I_Ext_Inmed,
input [4:0]I_RT,
input [4:0]I_RD,
output reg [1:0]O_WB,
output reg [2:0]O_M,
output reg O_EX_RegDst,
output reg [2:0]O_EX_ALUOp,
```

```
output reg O_EX_ALUSrc,
output reg [31:0]O_Next_address,
output reg [31:0]O_O1,
output reg [31:0]O_O2,
output reg [31:0]O_Ext_Inmed,
output reg [4:0]O_RT,
output reg [4:0]O_RD);

always @(posedge clk)
begin
    O_WB=I_WB;
    O_M=I_M;
    O_EX_RegDst=I_EX[0];
    O_EX_ALUOp=I_EX[3:1];
    O_EX_ALUSrc=I_EX[4];
    O_Next_address=I_Next_address;
    O_O1=I_O1;
    O_O2=I_O2;
    O_Ext_Inmed=I_Ext_Inmed;
    O_RT=I_RT;
    O_RD=I_RD;
end

endmodule
```

Parámetros del módulo:

- input clk: Es una entrada que representa la señal de reloj.
- input [1:0] I_WB: Es una entrada de 2 bits que parece estar relacionada con la escritura de datos de vuelta (write back).
- input [2:0] I_M: Es una entrada de 3 bits, posiblemente relacionada con el manejo de memoria.
- input [4:0] I_EX: Es una entrada de 5 bits que parece estar relacionada con la etapa de ejecución.
- input [31:0] I_Next_address: Es una entrada de 32 bits que representa la dirección de la siguiente instrucción en el pipeline.
- input [31:0] I_O1: Es una entrada de 32 bits.
- input [31:0] I_O2: Es otra entrada de 32 bits.
- input [31:0] I_Ext_Inmed: Es otra entrada de 32 bits, posiblemente una extensión inmediata.
- input [4:0] I_RT: Es una entrada de 5 bits.
- input [4:0] I_RD: Es otra entrada de 5 bits.
- output reg [1:0] O_WB: Es una salida de 2 bits.
- output reg [2:0] O_M: Es una salida de 3 bits.
- output reg O_EX_RegDst: Es una salida de 1 bit.

- output reg [2:0] O_EX_ALUOp: Es una salida de 3 bits relacionada con la operación de la ALU.
- output reg O_EX_ALUSrc: Es una salida de 1 bit relacionada con la fuente de datos de la ALU.
- output reg [31:0] O_Next_address: Es una salida de 32 bits que retiene la dirección de la siguiente instrucción.
- output reg [31:0] O_O1: Es una salida de 32 bits.
- output reg [31:0] O_O2: Es otra salida de 32 bits.
- output reg [31:0] O_Ext_Inmed: Es otra salida de 32 bits.
- output reg [4:0] O_RT: Es una salida de 5 bits.
- output reg [4:0] O_RD: Es otra salida de 5 bits.

Bloque always @(posedge clk):

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

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

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

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

Buffer 3_EXMEM.v

```
//BUFFER 3: EX/MEM
module EXMEM(
input clk,
input [1:0]I_WB,
input [2:0]I_M,
input [31:0]I_ADD_Res,
input I_ZF,
input [31:0]I_ALU_Res,
input [31:0]I_DatWri_Mem,
input [4:0]I_Addr_Reg_Wri,
output reg [1:0]O_WB,
output reg O_M_Branch,
output reg O_M_MemRead,
output reg O_M_MemWrite,
output reg [31:0]O_ADD_Res,
output reg O_ZF,
output reg [31:0]O_ALU_Res,
output reg [31:0]O_DatWri_Mem,
```

```
output reg [4:0]O_Addr_Reg_Wri);

always @(posedge clk)
begin
    O_WB=I_WB;
    O_M_Branch=I_M[0];
    O_M_MemRead=I_M[1];
    O_M_MemWrite=I_M[2];
    O_ADD_Res=I_ADD_Res;
    O_ZF=I_ZF;
    O_ALU_Res=I_ALU_Res;
    O_DatWri_Mem=I_DatWri_Mem;
    O_Addr_Reg_Wri=I_Addr_Reg_Wri;
end

endmodule
```

Parámetros del módulo:

- input clk: Es una entrada que representa la señal de reloj.
- input [1:0] I_WB: Es una entrada de 2 bits que parece estar relacionada con la escritura de datos de vuelta (write back).
- input [2:0] I_M: Es una entrada de 3 bits que posiblemente esté relacionada con operaciones de memoria (memoria de lectura/escritura, operaciones de branch).
- input [31:0] I_ADD_Res: Es una entrada de 32 bits que representa el resultado de una operación de suma realizada en la etapa de ejecución.
- input I_ZF: Es una entrada de 1 bit, posiblemente relacionada con la bandera de cero (zero flag).
- input [31:0] I_ALU_Res: Es una entrada de 32 bits que representa el resultado de una operación de la Unidad Lógico-Aritmética (ALU).
- input [31:0] I_DatWri_Mem: Es una entrada de 32 bits que representa los datos que se escribirán en memoria.
- input [4:0] I_Addr_Reg_Wri: Es una entrada de 5 bits que representa la dirección del registro que se actualizará en la etapa de escritura de vuelta.
- output reg [1:0] O_WB: Es una salida de 2 bits.
- output reg O_M_Branch: Es una salida de 1 bit que indica si se realizará una operación de branch en la etapa de memoria.
- output reg O_M_MemRead: Es una salida de 1 bit que indica si se realizará una operación de lectura de memoria en la etapa de memoria.
- output reg O_M_MemWrite: Es una salida de 1 bit que indica si se realizará una operación de escritura de memoria en la etapa de memoria.
- output reg [31:0] O_ADD_Res: Es una salida de 32 bits que retiene el resultado de la operación de suma.

- output reg O_ZF: Es una salida de 1 bit que retiene el estado de la bandera de cero.
- output reg [31:0] O_ALU_Res: Es una salida de 32 bits que retiene el resultado de la operación de la ALU.
- output reg [31:0] O_DatWri_Mem: Es una salida de 32 bits que retiene los datos que se escribirán en memoria.
- output reg [4:0] O_Addr_Reg_Wri: Es una salida de 5 bits que retiene la dirección del registro que se actualizará en la etapa de escritura de vuelta.

Bloque always @(posedge clk):

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

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

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

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

Buffer 4_MEMWB.v

```
//BUFFER 4: MEM/WB
module MEMWB(
input clk,
input [1:0]I_WB,
input [31:0]I_ReDat_Mem,
input [31:0]I_ALU_Res,
input [4:0]I_Addr_Reg_Wri,
output reg O_WB_RegWrite,
output reg O_WB_MemtoReg,
output reg [31:0]O_ReDat_Mem,
output reg [31:0]O_ALU_Res,
output reg [4:0]O_Addr_Reg_Wri);

always @(posedge clk)
begin
    O_WB_RegWrite=I_WB[0];
    O_WB_MemtoReg=I_WB[1];
    O_ReDat_Mem=I_ReDat_Mem;
    O_ALU_Res=I_ALU_Res;
    O_Addr_Reg_Wri=I_Addr_Reg_Wri;
end
```

```
endmodule
```

Parámetros del módulo:

- input clk: Es una entrada que representa la señal de reloj.
- input [1:0] I_WB: Es una entrada de 2 bits que parece estar relacionada con la escritura de datos de vuelta (write back).
- input [31:0] I_ReDat_Mem: Es una entrada de 32 bits que representa los datos leídos de memoria.
- input [31:0] I_ALU_Res: Es una entrada de 32 bits que representa el resultado de una operación de la Unidad Lógico-Aritmética (ALU).
- input [4:0] I_Addr_Reg_Wri: Es una entrada de 5 bits que representa la dirección del registro que se actualizará en la etapa de escritura de vuelta.
- output reg O_WB_RegWrite: Es una salida de 1 bit que indica si se realizará una operación de escritura de vuelta.
- output reg O_WB_MemtoReg: Es una salida de 1 bit que indica si se realizará una operación de escritura de memoria a registro.
- output reg [31:0] O_ReDat_Mem: Es una salida de 32 bits que retiene los datos leídos de memoria.
- output reg [31:0] O_ALU_Res: Es una salida de 32 bits que retiene el resultado de la operación de la ALU.
- output reg [4:0] O_Addr_Reg_Wri: Es una salida de 5 bits que retiene la dirección del registro que se actualizará en la etapa de escritura de vuelta.

Bloque always @(posedge clk):

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

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

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

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

Fase 2_tb.v

```
`timescale 1ns/1ns
module Fase2_tb();

reg clk_tb;

MyDatapath _Datapath(.clk(clk_tb));
```

```
always #100 clk_tb = ~clk_tb;

initial
begin
    clk_tb = 1'b0;

    #20000
    $stop;
end

endmodule
```

Parámetros del módulo:

- reg clk_tb;: Declara una señal de reloj llamada clk_tb que será utilizada para clockear el módulo MyDatapath.

Instancia del módulo MyDatapath:

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

Generación de la señal de reloj:

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

Bloque inicial:

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

Este banco de pruebas (Fase2_tb) configura una simulación con un reloj (clk_tb) que cambia su valor cada 100 unidades de tiempo, instanciando el módulo MyDatapath y conectándolo a esta señal de reloj. La simulación se detendrá después de 20000 unidades de tiempo.

MyADD_4.v

```
module MyADD_4(
    input [31:0] operand,
    output reg [31:0] result
);

always @*
begin
```

```
    result = operand + 4;
end

endmodule
```

Parámetros del módulo:

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

Bloque always @*:

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

Bloque de código dentro de always @*:

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

Este módulo MyADD_4 toma un operando de 32 bits, le suma la constante 4 y devuelve el resultado en la salida result. Este tipo de módulos pueden ser útiles en diversas aplicaciones, como en el diseño de operaciones aritméticas específicas.

MyALU.v

```
module MyALU(
    input [31:0] operand1,
    input [31:0] operand2,
    input [3:0] alu_select,
    output reg zero_flag,
    output reg [31:0] result
);

always @*
begin
    case (alu_select)
        4'b0000:
        begin
            result = operand1 & operand2; // AND
        end

        4'b0001:
        begin
            result = operand1 | operand2; // OR
        end

        4'b0010:
```

```
        begin
            result = operand1 + operand2; //ADD
        end

        4'b0110:
        begin
            result = operand1 - operand2; //SUB
        end

        4'b0111:
        begin
            result = operand1 < operand2?1:0; // SLT
        end
        default:
        begin
            result=32'd0;
        end
    endcase

    zero_flag = result == 0;
end

endmodule
```

Parámetros del módulo:

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

Bloque always @*:

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

Bloque de código dentro de always @*:

- case (alu_select): Este bloque de código selecciona la operación de la ALU según el valor de alu_select.
- Cada begin y end dentro de los case contienen la lógica para una operación específica.

- zero_flag = (result == 32'd0);: Se establece la bandera de cero en 1 si el resultado es igual a cero, de lo contrario, se establece en 0.

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

MyALUControl.v

```
module MyALUControl(
    input [2:0] alu_operation,
    input [5:0] function_code,
    output reg [3:0] alu_select
);

always @*
begin
    case (alu_operation)
    3'b010:
    begin
        case (function_code)
            6'b100100:
                alu_select = 4'b0000; // AND
            6'b100101:
                alu_select = 4'b0001; // OR
            6'b100000:
                alu_select = 4'b0010; // ADD
            6'b100010:
                alu_select = 4'b0110; // SUBTRACT
            6'b101010:
                alu_select = 4'b0111; // SET ON LESS THAN (SLT)
            6'b000000:
                alu_select = 4'b0000; // Soporte a la operacion Nop
        endcase
    end
    3'b000:
    begin
    alu_select = 4'b0010; // ADD
    end
    3'b001:
    begin
    alu_select = 4'b0110; // SUBTRACT
    end
    3'b011:
```

```
    begin
    alu_select = 4'b0000; // AND
    end
    3'b100:
    begin
    alu_select = 4'b0001; // OR
    end
    3'b101:
    begin
    alu_select = 4'b0111; // SLT
    end

    endcase
end

endmodule
```

Parámetros del módulo:

- input [2:0] alu_operation: Es una entrada de 3 bits que representa la operación de la ALU.
- input [5:0] function_code: Es una entrada de 6 bits que representa el código de función.
- output reg [3:0] alu_select: Es una salida de 4 bits que representa la selección de operación de la ALU.

Bloque always @*:

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

Bloque de código dentro de always @*:

- case (alu_operation): Este bloque de código selecciona la operación de la ALU basándose en el valor de alu_operation.
- Dentro de cada caso (begin y end), hay un segundo case que selecciona la operación de la ALU basándose en el valor de function_code.
- Dependiendo de los valores de alu_operation y function_code, se asigna la señal alu_select con la operación correspondiente.

Este módulo MyALUControl toma las señales de control alu_operation y function_code como entradas y genera la señal de selección de operación de la ALU (alu_select) de acuerdo con la operación deseada.

MyBR.v

```
module MyBR(
    input [4:0] address_reg1,
    input [4:0] address_reg2,
```

```
    input [4:0] address_write,
    input [31:0] data_write,
    input enable, //Habilitar la escritura
    output reg [31:0] data_read1,
    output reg [31:0] data_read2
);

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

/*initial
begin
    memory[0] = 32'd0;
    memory[1] = 32'd100;
    memory[2] = 32'd120;
    memory[3] = 32'b10101;
    memory[4] = 32'b01010;
    memory[31] = 32'd6;
end*/

always @*
begin
    if (enable && data_write !== 32'bx)
    begin
        memory[address_write] = data_write; // escribe en la direcci□n
address
    end

    data_read1 = memory[address_reg1]; // lee la memoria en la direcci□n
address
    data_read2 = memory[address_reg2];
end

endmodule
```

Parámetros del módulo:

- input [4:0] address_reg1: Es una entrada de 5 bits que representa la dirección de memoria desde la cual leer data_read1.
- input [4:0] address_reg2: Es otra entrada de 5 bits que representa la dirección de memoria desde la cual leer data_read2.
- input [4:0] address_write: Es una entrada de 5 bits que representa la dirección de memoria en la cual escribir data_write.
- input [31:0] data_write: Es una entrada de 32 bits que representa los datos a escribir en memoria.
- input enable: Es una entrada que habilita la escritura en memoria.

- output reg [31:0] data_read1: Es una salida de 32 bits que retiene los datos leídos desde la dirección address_reg1.
- output reg [31:0] data_read2: Es otra salida de 32 bits que retiene los datos leídos desde la dirección address_reg2.

Arreglo de memoria:

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

Bloque always @*:

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

Bloque de código dentro de always @*:

- if (enable && data_write !== 32'bx): Verifica si la habilitación está activa y si los datos de escritura son distintos de "x" (no definidos).
- memory[address_write] = data_write;: Si se cumple la condición anterior, se escribe data_write en la dirección de memoria especificada por address_write.
- data_read1 = memory[address_reg1];: Lee la memoria en la dirección especificada por address_reg1 y retiene los datos en data_read1.
- data_read2 = memory[address_reg2];: Lee la memoria en la dirección especificada por address_reg2 y retiene los datos en data_read2.

Este módulo MyBR implementa una memoria simple que puede leer de dos direcciones de memoria y escribir en una dirección de memoria específica cuando la habilitación está activa. Los datos se retienen en data_read1 y data_read2. La memoria puede ser inicializada descomentando la sección initial que está comentada en el código.

MyControlUnit.v

```
module MyControlUnit(
    input [31:26] opcode,
    output reg reg_Dst,
    output reg branch,
    output reg memory_read,
    output reg memory_register,
    output reg [2:0] alu_operation,
    output reg memory_write,
    output reg ALU_Src,
    output reg reg_write,
    output reg jump
);
```

```
always @*
begin
    case (opcode)
        6'b000000://Instrucciones Tipo R
        begin
            reg_Dst=1'b1;
            branch=1'b0;
            memory_read = 1'b0;
            memory_register = 1'b0;
            alu_operation = 3'b010;
        memory_write = 1'b0;
            ALU_Src=1'b0;
            reg_write = 1'b1;
        jump=1'b0;
        end
        6'b001000://ADDI
        begin
            reg_Dst=1'b0;
            branch=1'b0;
            memory_read = 1'b0;
            memory_register = 1'b0;
            alu_operation = 3'b000;
        memory_write = 1'b0;
            ALU_Src=1'b1;
            reg_write = 1'b1;
        jump=1'b0;
        end
        6'b001100://ANDI
        begin
            reg_Dst=1'b0;
            branch=1'b0;
            memory_read = 1'b0;
            memory_register = 1'b0;
            alu_operation = 3'b011;//C mo hay que ponerlo?
        memory_write = 1'b0;
            ALU_Src=1'b1;
            reg_write = 1'b1;
        jump=1'b0;
        end
        6'b001101://ORI
        begin
            reg_Dst=1'b0;
            branch=1'b0;
            memory_read = 1'b0;
            memory_register = 1'b0;
```

```
            alu_operation = 3'b100;//C mo hay que ponerlo?
        memory_write = 1'b0;
            ALU_Src=1'b1;
            reg_write = 1'b1;
            jump=1'b0;
        end
        /*6'b000000://SUBI??
        begin
            reg_Dst=1'b0;
            branch=1'b0;
            memory_read = 1'b0;
            memory_register = 1'b0;
            alu_operation = 3'b001;
        memory_write = 1'b0;
            ALU_Src=1'b1;
            reg_write = 1'b1;
            jump=1'b0;
        end*/
        6'b100011://LW
        begin
            reg_Dst=1'b0;
            branch=1'b0;
            memory_read = 1'b1;
            memory_register = 1'b1;
            alu_operation = 3'b000;
        memory_write = 1'b0;
            ALU_Src=1'b1;
            reg_write = 1'b1;
            jump=1'b0;
        end
        6'b101011://SW
        begin
            reg_Dst=1'b0;
            branch=1'b0;
            memory_read = 1'b0;
            memory_register = 1'b0;
            alu_operation = 3'b000;
        memory_write = 1'b1;
            ALU_Src=1'b1;
            reg_write = 1'b0;
            jump=1'b0;
        end
        6'b001010://SLTI
        begin
            reg_Dst=1'b0;
```

```
            branch=1'b0;
            memory_read = 1'b0;
            memory_register = 1'b0;
            alu_operation = 3'b101;//�C�mo hay que ponerlo?
        memory_write = 1'b0;
            ALU_Src=1'b1;
            reg_write = 1'b1;
            jump=1'b0;
        end
        6'b000100://BEQ
        begin
            reg_Dst=1'b0;
            branch=1'b1;
            memory_read = 1'b0;
            memory_register = 1'b0;
            alu_operation = 3'b001;//�Est� bien?
        memory_write = 1'b0;
            ALU_Src=1'b0;
            reg_write = 1'b0;
            jump=1'b0;
        end
        /*6'b000101://BNE
        begin
            reg_Dst=1'b0;
            branch=1'b1;
            memory_read = 1'b0;
            memory_register = 1'b0;
            alu_operation = 3'b001;//�C�mo implementar esta operaci�n?
        memory_write = 1'b0;
            ALU_Src=1'b0;
            reg_write = 1'b0;
            jump=1'b0;
        end
        6'b000111://BGTZ
        begin
            reg_Dst=1'b0;
            branch=1'b1;
            memory_read = 1'b0;
            memory_register = 1'b0;
            alu_operation = 3'b101; //Esta operaci�n falla cuando se elige
un 0, porque 0>0 es falso y la manera en que implement� con SLT hace que se
ejecut� el branch
        memory_write = 1'b0;
            ALU_Src=1'b0;
            reg_write = 1'b0;
```

```
            jump=1'b0;
        end*/
        6'b000010://Instrucci□n J
    begin
            reg_Dst=1'b0;
            branch=1'b0;
            memory_read = 1'b0;
            memory_register = 1'b0;
            alu_operation = 3'b000;//MODIFICAR
        memory_write = 1'b0;
            ALU_Src=1'b0;
            reg_write = 1'b0;
        jump=1'b1;
    end
    endcase
end

endmodule
```

Parámetros del módulo:

- input [31:26] opcode: Es una entrada de 6 bits que representa el código de operación de la instrucción.
- output reg reg_Dst: Salida que indica si la instrucción es de tipo R.
- output reg branch: Salida que indica si la instrucción es de salto condicional.
- output reg memory_read: Salida que indica si la instrucción realiza una lectura de memoria.
- output reg memory_register: Salida que indica si la instrucción involucra operaciones de memoria y registros.
- output reg [2:0] alu_operation: Salida que especifica la operación de la ALU.
- output reg memory_write: Salida que indica si la instrucción realiza una escritura en memoria.
- output reg ALU_Src: Salida que indica si la fuente de datos para la ALU es inmediata (1) o un registro (0).
- output reg reg_write: Salida que indica si la instrucción realiza una escritura en registros.
- output reg jump: Salida que indica si la instrucción es de salto incondicional.

Bloque always @*:

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

Bloque de código dentro de always @*:

- case (opcode): Este bloque de código selecciona las señales de control basándose en el código de operación (opcode).

- Dentro de cada caso (begin y end), se establecen las señales de control correspondientes según la instrucción especificada por el opcode.

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

MyDatapath.v

```
module MyDatapath(
    input clk
);

wire [31:0] instruction;
wire control_memreg;
wire control_memwrite;
wire control_memread;
wire control_regwrite;
wire [2:0] control_aluop;
wire [31:0] data_write;
wire [31:0] data_read1;
wire [31:0] data_read2;
wire [3:0] control_alusel;
wire [31:0] alu_result;
wire [31:0] address_result;
wire [31:0] data_mem;
wire RegDst;
wire [4:0] Dir_Wri_BR;
wire ALU_Src;
wire [31:0] Oper_2;
wire [31:0] Mux_ALU_in2;
wire [31:0] pc_out;
wire [31:0] add_result;
wire [31:0] SL2_adder;
wire [31:0] Add_In2_MuxPC;
wire Branch;
wire tr_zf;
wire [31:0] MuxPC_PCin;
wire Jump;
wire [27:0]JAddress;
wire [31:0]MPC_MJ;


MyPC _PC(.clk(clk),.in(MuxPC_PCin),.out(pc_out));
```

```
MyADD_4 _ADD(.operand(pc_out),.result(add_result));
MyIMem _IM(.address(pc_out),.data(instruction));
MyBR _BR(.address_reg1(instruction[25:21]),
.address_reg2(instruction[20:16]), .address_write(Dir_Wri_BR),
.data_write(data_write), .enable(control_regwrite), .data_read1(data_read1),
.data_read2(data_read2));
MyControlUnit _ControlUnit(.opcode(instruction[31:26]), .reg_Dst(RegDst),
.branch(Branch), .memory_read(control_memread),
.memory_register(control_memreg), .alu_operation(control_aluop),
.memory_write(control_memwrite), .ALU_Src(ALU_Src),
.reg_write(control_regwrite), .jump(Jump));
MyALUControl _ALUControl(.alu_operation(control_aluop),
.function_code(instruction[5:0]), .alu_select(control_alusel));
MyALU _ALU(.operand1(data_read1), .operand2(Oper_2),
.alu_select(control_alusel), .zero_flag(tr_zf) ,.result(alu_result));
MyMem _Mem(.address(alu_result), .data_write(data_read2),
.write_enable(control_memwrite), .read_enable(control_memread),
.data_read(data_mem));
MyDatapathMux
_MuxWriteBR(.control_signal(control_memreg),.input_data_1(alu_result),.input
_data_2(data_mem), .output_data(data_write));
MyDatapathMux_5B _Mux_Dir_BR(.control_signal(RegDst),
.input_data_1(instruction[20:16]), .input_data_2(instruction[15:11]),
.output_data(Dir_Wri_BR));
MyDatapathMux _Mux_D2_ALU(.control_signal(ALU_Src),
.input_data_1(data_read2), .input_data_2(Mux_ALU_in2),
.output_data(Oper_2));
sign_extend _Sig_Ext(.Data_in(instruction[15:0]), .Data_out(Mux_ALU_in2));
Shift_Left_2 _Sh_Lef_2(.SL_in(Mux_ALU_in2), .SL_out(SL2_adder));
ADDER _Sumador(.O1(add_result), .O2(SL2_adder), .Res(Add_In2_MuxPC));
MyDatapathMux _Mux_PC(.control_signal(Branch&tr_zf),
.input_data_1(add_result), .input_data_2(Add_In2_MuxPC),
.output_data(MPC_MJ));
MyDatapathMux _Mux_J(.control_signal(Jump), .input_data_1(MPC_MJ),
.input_data_2(    {add_result[31:28],JAddress}   ),
.output_data(MuxPC_PCin));
Shift_Left_26_28 _SL_26_28(.SL_in(instruction[25:0]), .SL_out(JAddress));



initial
begin
$readmemb("Inicializaci?n Memoria de Instrucciones.txt",_IM.memory);
$readmemb("Inicializaci?n BR.txt",_BR.memory);
$readmemb("Inicializaci?n Memoria de Datos.txt",_Mem.memory);
```

```
end

endmodule
```

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

MyDatapathMux.v

```
module MyDatapathMux(
    input control_signal,
    input [31:0] input_data_1, //0
    input [31:0] input_data_2, //1
    output reg [31:0] output_data
);

always @*
begin
    if(control_signal)
    begin
        output_data = input_data_2;
    end
    else
    begin
        output_data = input_data_1;
    end
end

endmodule
```

Parámetros del módulo:

- input control_signal: Es una señal de control que determina qué entrada se selecciona.
- input [31:0] input_data_1: Es la primera entrada del multiplexor.
- input [31:0] input_data_2: Es la segunda entrada del multiplexor.
- output reg [31:0] output_data: Es la salida del multiplexor.

Bloque always @*:

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

Bloque de código dentro de always @*:

- El bloque de código utiliza una estructura condicional (if-else) para seleccionar la salida del multiplexor en función de la señal de control.
- Si control_signal es verdadero (1), selecciona input_data_2.
- Si control_signal es falso (0), selecciona input_data_1.

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

MyDatapathMux5bits.v

```
module MyDatapathMux_5B(
    input control_signal,
    input [4:0] input_data_1, //0
    input [4:0] input_data_2, //1
    output reg [4:0] output_data
);

always @*
begin
    if(control_signal)
    begin
        output_data = input_data_2;
    end
    else
    begin
        output_data = input_data_1;
    end
end

endmodule
```

Parámetros del módulo:

- input control_signal: Es una señal de control que determina qué entrada se selecciona.
- input [4:0] input_data_1: Es la primera entrada del multiplexor.
- input [4:0] input_data_2: Es la segunda entrada del multiplexor.
- output reg [4:0] output_data: Es la salida del multiplexor.

Bloque always @*:

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

Bloque de código dentro de always @*:

- El bloque de código utiliza una estructura condicional (if-else) para seleccionar la salida del multiplexor en función de la señal de control.

- Si control_signal es verdadero (1), selecciona input_data_2.
- Si control_signal es falso (0), selecciona input_data_1.

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

MyIMem.v

```
module MyIMem(
    input [31:0] address,
    output reg [31:0] data
);

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

initial
begin
    /*{memory[0],memory[1],memory[2],memory[3]}       = 32'b0;
    {memory[4],memory[5],memory[6],memory[7]}       =
32'b00000000000000010010100000100000;
    {memory[8],memory[9],memory[10],memory[11]}     =
32'b00000000001000100011000000100000;
    {memory[12],memory[13],memory[14],memory[15]}   =
32'b00000000010000010011100000100010;
    {memory[16],memory[17],memory[18],memory[19]}   =
32'b00000000001000100100000000100010;
    {memory[20],memory[21],memory[22],memory[23]}   =
32'b00000000011001000100100000100100;
    {memory[24],memory[25],memory[26],memory[27]}   =
32'b00000000100000110101000000100101;
    {memory[28],memory[29],memory[30],memory[31]}   =
32'b00000000011010100101100000100100;
    {memory[32],memory[33],memory[34],memory[35]}   =
32'b00000000111010000110000000100101;
    {memory[36],memory[37],memory[38],memory[39]}   =
32'b00000000011001000110100000101010;
    {memory[40],memory[41],memory[42],memory[43]}   =
32'b00000000001000100111000000101010;*/

end

always @*
begin
    data = {memory[address], memory[address + 1], memory[address + 2],
memory[address + 3]};
```

```
end

endmodule
```

Parámetros del módulo:

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

Bloque initial:

- Contiene la inicialización de la memoria con valores comentados.

Bloque always @*:

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

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

MyMem.v

```
module MyMem(
    input [31:0] address,
    input [31:0] data_write,
    input write_enable, // es como el enable
    input read_enable, // es como el enable
    output reg [31:0] data_read
);

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

/*initial
begin
    memory[0]=32'd1;
    memory[1]=32'd2;
    memory[2]=32'd6;
    memory[30]=32'd10;
end*/


always @*
begin
```

```
    if (write_enable == 1'b1 & read_enable == 1'b0)
    begin
        memory[address] = data_write; // escribe en la direcci□n address
    end
    else if (write_enable == 1'b0 & read_enable == 1'b1)
    begin
        data_read = memory[address]; // lee la memoria en la direcci□n
address
    end
end

endmodule
```

Parámetros del módulo:

- input [31:0] address: Es la dirección de memoria para la operación.
- input [31:0] data_write: Es la información que se escribirá en la memoria.
- input write_enable: Es una señal que habilita la escritura.
- input read_enable: Es una señal que habilita la lectura.
- output reg [31:0] data_read: Es la información leída desde la memoria.
- Bloque reg [31:0] memory[0:255];: Declara una memoria de palabras de 32 bits llamada memory con 256 ubicaciones.

Bloque always @*:

- Utiliza always @* para que el bloque de código se ejecute siempre que haya un cambio en las entradas.
- Si write_enable está habilitado (1) y read_enable está deshabilitado (0), se escribe data_write en la dirección especificada por address.
- Si write_enable está deshabilitado (0) y read_enable está habilitado (1), se lee la memoria en la dirección especificada por address y se almacena en data_read.

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

MyPC.v

```
module MyPC(
    input clk,
    input [31:0]in,
    output reg [31:0]out);

initial
begin
    out = 32'b0;
end
```

```
always@(clk)
begin
    if(clk)
    begin
        out = in;
    end
end
endmodule
```

Parámetros del módulo:

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

Bloque initial:

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

Bloque always @(posedge clk):

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

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

Shift_Left_2.v

```
module Shift_Left_2(
input [31:0]SL_in,
output reg[31:0]SL_out
);

always @*
    begin
       SL_out = SL_in << 2;
    end
endmodule
```

Parámetros del módulo:

- input [31:0] SL_in: Es la entrada de 32 bits en la que se realizará el desplazamiento a la izquierda.

- output reg [31:0] SL_out: Es la salida de 32 bits que contendrá el resultado del desplazamiento.

Bloque always @*:

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

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

Sign_extend.v

```
module sign_extend(
input [15:0] Data_in,
output reg [31:0] Data_out);

always @*
begin
    if(Data_in[15])
    begin
        Data_out={16'b1111111111111111,Data_in};
    end
    else
    begin
    Data_out={16'b0000000000000000,Data_in};
    end
end

endmodule
```

Parámetros del módulo:

- input [15:0] Data_in: Es la entrada de 16 bits que se extenderá.
- output reg [31:0] Data_out: Es la salida de 32 bits que contendrá el resultado de la extensión.

Bloque always @*:

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

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

Conclusiones

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

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

JUAN SILVERO VALENCIA: En conclusión puede decir que el proyecto fue algo no tan fácil de realizarse ya que por el tiempo y la falta de experiencia en el tema nos mantuvo un poco confusos a muchos de mis compañeros y a mi más en lo personal ya que también es la primera vez que veo los números Binarios de tema y todo este tipo de instrucciones, es interesante la materia pero debes de tener mucha paciencia y debes de comprender realmente lo que estás haciendo, no era lo que esperaba de la materia porque tenía otra idea de esta realmente creo que este tema no lo voy a volver a ver en mi carrera pero en algo más adelante puede servir.

ERICK JARED GUTIERREZ CORREA