Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías.

Módulo de aprendizaje: Arquitectura de Computadoras

Número y nombre de la actividad: Proyecto parte 2

Equipo: Intel2

Integrantes:

Arias Silva Rubén

Sánchez López Jorge Alberto

Fecha de entrega: 15 de noviembre de 2024

CONTENIDOS

INTRODUCCIÓN

En la fase 1 del proyecto, se implementaron los módulos básicos de la estructura de un procesador MIPS capaz de trabajar únicamente con instrucciones de tipo R. Para esta segunda entrega, se realizaron adaptaciones para permitir la ejecución de instrucciones de tipo I. Esto incluyó la adición de módulos faltantes y conexiones necesarias para garantizar una lógica correcta. Además, el decodificador fue actualizado para codificar las instrucciones de tipo I en los números binarios correspondientes.

Las instrucciones de tipo I (inmediatas) son un conjunto de instrucciones en la arquitectura MIPS diseñadas para operaciones que involucran un registro y un valor constante (inmediato). Este tipo de instrucciones es común en tareas como la carga de valores, comparaciones, operaciones aritméticas o lógicas con constantes, y el cálculo de direcciones de memoria. Su formato compacto incluye un código de operación, dos registros (una fuente y otro destino) y un campo inmediato de 16 bits, lo que permite realizar operaciones más eficientes y flexibles al trabajar con valores constantes sin necesidad de almacenarlos previamente en la memoria.

Los procesadores MIPS, que significan (Microprocesador sin etapas de tubería interbloqueadas) son un tipo de arquitectura de procesador que utiliza la técnica RISC (Conjunto de Instrucciones Reducida). Esta arquitectura se creó en los años 80 y se destaca por tener un conjunto limitado de instrucciones sencillas que pueden ejecutarse rápidamente, en solo un ciclo de reloj. Esto los hace eficientes y fáciles de usar.

Una de sus principales características es el uso de un conjunto de instrucciones reducido, optimizado para que cada instrucción se ejecute en uno o pocos ciclos de reloj, lo que mejora el rendimiento general del procesador. Todos los datos e instrucciones manejados por el MIPS tienen un tamaño de 32 bits, lo que incluye direcciones de memoria, registros e instrucciones, permitiendo direccionar hasta 4 GB de memoria.
El conjunto de instrucciones de MIPS está compuesto por tres tipos principales: las instrucciones R (que operan sobre registros), las instrucciones I (que incluyen constantes inmediatas y acceso a memoria) y las instrucciones J (que manejan saltos en el flujo de ejecución). Además, el procesador cuenta con 32 registros de propósito general (GPR), cada uno de 32 bits, que almacenan temporalmente datos y resultados intermedios, mejorando la eficiencia al minimizar el acceso a la memoria principal.

MIPS sigue una estrategia de carga/almacenamiento (load/store), donde las operaciones aritméticas y lógicas solo ocurren entre registros, y las interacciones con la memoria se realizan a través de instrucciones específicas de carga (load) y almacenamiento (store). Para optimizar la ejecución, el procesador implementa un pipeline de 5 etapas (obtención de instrucción, decodificación, ejecución, acceso a memoria y escritura del resultado), lo que permite la ejecución simultánea de diferentes partes de varias instrucciones. Además, el procesador MIPS admite instrucciones de salto y bifurcación, que permiten

alterar el flujo del programa mediante saltos y bifurcaciones condicionales, como beq(ramificar si es igual) o bne(ramificar si no es igual). También incluye operaciones de multiplicación y división, cuyos resultados de 64 bits se almacenan en los registros especiales HI y LO. Asimismo, MIPS cuenta con mecanismos para manejar excepciones e interrupciones, lo que permite gestionar errores en tiempo de ejecución y eventos externos que requieren atención inmediata.

En términos de operaciones básicas, MIPS ofrece una amplia gama de instrucciones aritméticas y lógicas, como add, y, que operan exclusivamente entre registros. Otro aspecto relevante es la configuración de endianness (big-endian o little-endian), que define cómo se almacenan los bytes en la memoria. MIPS maneja la memoria mediante direccionamiento basado en bytes, interactuando con palabras alineadas, y permite trabajar con datos de diferentes tamaños como bytes y medias palabras. Sub and or.
Por último, las versiones avanzadas de MIPS son compatibles con multiprocesamiento, permitiendo el uso de múltiples núcleos que aumentan la capacidad de procesamiento al manejar tareas simultáneamente en diferentes núcleos, lo que incrementa el rendimiento en aplicaciones de alto rendimiento.

Algunos de sus elementos son:
1. **Registros**:
Registros de Propósito General (GPR) MIPS tiene 32 registros, numerados de $0 a $31, cada uno de 32 bits. Los registros son utilizados para almacenar datos temporales y resultados intermedios.

- $0: Siempre contiene el valor 0.
- $1: Conocido como $at, reservado para el ensamblador.
- $2a $3: Usados para llamadas a funciones.
- $4a $7: Usados para pasar parámetros a funciones.
- $8a $15: Registros temporales.
- $16a $23: Registros salvados, que el llamado función debe preservar.
- $24a $31: Registros de propósito general.

Registros Especiales: HI y LO Usados para almacenar resultados de operaciones de multiplicación y división de 64 bits.
Registro de Estado (Status): Utilizado para gestionar excepciones e interrupciones.

2. **ALU** (Unidad Aritmético Lógica):
Es el componente que realiza las operaciones aritméticas (suma, resta) y lógicas (AND, OR, etc.) sobre los datos almacenados en los registros.

3. **Unidad de Control**:
Coordina el funcionamiento del procesador, decodificando las instrucciones y controlando el flujo de datos entre los registros, la ALU y la memoria. Determine qué operaciones se deben realizar en cada ciclo de reloj.

4. **Memoria caché**:
Puede incluir memoria caché L1 (instrucciones y datos) para acelerar el acceso a datos utilizados frecuentemente, reduciendo el tiempo de espera en la memoria principal.

5. **Sistema de memoria**:
Memoria Principal (RAM): Almacena datos e instrucciones que el procesador utiliza durante la ejecución. MIPS utiliza un esquema de direccionamiento por bytes y permite operaciones de carga y almacenamiento.

6. **Bus de Datos**:
Conecta los diferentes componentes del procesador y permite la transferencia de datos entre los registros, la ALU y la memoria.

7. **Autobús de direcciones**:
Lleva la información de las direcciones de memoria a las que el procesador necesita acceder para leer o escribir datos.

8. **Tubería**:
Aunque no es un componente físico, la técnica de tubería se utiliza en MIPS para permitir la ejecución de múltiples instrucciones simultáneamente, dividiendo el proceso en cinco etapas (IF, ID, EX, MEM, WB). Esto mejora la eficiencia y el rendimiento general del procesador.

9. **Instrucciones de Salto y Control de Flujo**:
Incluyen instrucciones que alteran el flujo de ejecución del programa, como saltos incondicionales (j) y saltos condicionales (beq, bne), esenciales para la implementación de estructuras de control como bucles y condiciones.

10. **Interrupciones y Excepciones**:
El procesador MIPS incluye mecanismos para manejar interrupciones, permitiendo al procesador responder a eventos externos y excepciones que ocurren durante la ejecución, como errores en el acceso a la memoria.

11. **Modo Endianness**:
MIPS puede operar en modo big-endian o little-endian, determinando el orden de los bytes en la memoria, lo cual es relevante en la manipulación de datos

El conjunto de instrucciones en un procesador MIPS es el grupo de operaciones que el procesador puede ejecutar directamente. Estas instrucciones son los comandos básicos que la unidad central de procesamiento (CPU) entiende y ejecuta. En un procesador MIPS, el conjunto de instrucciones sigue el diseño RISC (Computadora con conjunto de

instrucciones reducidas), lo que significa que tiene un número limitado de instrucciones, pero cada una es muy simple y rápida de ejecutar.

El conjunto de instrucciones MIPS se caracteriza por tener instrucciones de 32 bits, lo que facilita su decodificación en hardware. Sigue la arquitectura RISC, enfocada en un conjunto reducido de instrucciones que se ejecutan rápidamente. Las instrucciones se dividen en tres tipos: R-type (operaciones aritméticas y lógicas con registros), I-type (instrucciones con valores inmediatos) y J-type (instrucciones de salto para control de flujo). MIPS usa un modelo de carga/almacenamiento, donde solo las instrucciones de carga (lw) y almacenamiento (sw) interactúan con la memoria.

Ejemplos de instrucciones incluyen: add $t1, $t2, $t3para sumar registros, lw $t0, 4($t1) para cargar un valor de la memoria en un registro, y beq $t0, $t1, label para saltar a una etiqueta si dos registros son iguales.

Grandes personajes que vale la pena conocer:

Ljubisa Bajic: Actualmente director de Tenstorrent, una empresa que trabaja en el diseño de procesadores de alto rendimiento para inteligencia artificial (IA). Bajic, junto a su equipo, ha desarrollado arquitecturas innovadoras de System-on-Chip (SoC) que combinan núcleos dedicados al aprendizaje automático y procesadores RISC-V generales. Esta integración permite manejar tanto tareas de procesamiento general como de alto rendimiento en IA. Los procesadores de Tenstorrent, en los que Bajic ha influido, son diseñados para el futuro de aplicaciones de supercomputación que requieren capacidades avanzadas en IA.
Aportaciones:

- **Desarrollo en RISC-V**: Bajic ha estado involucrado en el diseño de nuevas arquitecturas basadas en el estándar **RISC-V**, una arquitectura **open-source** que se ha vuelto muy popular para aplicaciones embebidas, de servidor y de IA.

- **Procesadores especializados en IA**: Bajic también ha trabajado en el desarrollo de **procesadores dedicados a tareas de inteligencia artificial** (como la computación de redes neuronales), lo cual ha sido clave para mejorar la eficiencia y reducir el consumo energético de los sistemas de IA

Jim Keller: Reconocido ingeniero en microarquitectura y un líder en la industria de semiconductores, Keller ha realizado avances significativos en el desarrollo de procesadores x86 y ARM. En AMD, diseñó arquitecturas clave como la serie Zen, que transformó el rendimiento de los procesadores AMD. Más recientemente, en Tenstorrent, ha trabajado junto a Bajic en el diseño de SoCs optimizados para IA, adoptando la arquitectura RISC-V para satisfacer las crecientes demandas de

software y procesamiento para IA en dispositivos de nueva generación. Keller también ha defendido la necesidad de plataformas flexibles y abiertas en la IA, contribuyendo al ecosistema de software y hardware en RISC-V.
Aportaciones:

- **Zen Architecture de AMD**: Keller fue fundamental en el desarrollo de la **arquitectura Zen**, que revitalizó a **AMD** y permitió competir de manera más eficaz contra Intel en términos de rendimiento. Esta arquitectura fue la base para las exitosas series de procesadores **Ryzen** y **EPYC** de AMD.

- **Trascendencia en arquitectura RISC**: A pesar de que Keller no estuvo directamente involucrado con MIPS, su trabajo en la arquitectura **RISC** se ha visto reflejado en **RISC-V**, y ha influido en el diseño de núcleos eficientes y escalables.

- **Tesla**: Además de su trabajo en AMD, Keller también trabajó en **Tesla**, donde ayudó a desarrollar el chip **Full Self Driving** (FSD), que es crucial para la conducción autónoma.

Raja Koduri: Ingeniero y ejecutivo de gran influencia en el desarrollo de gráficos y computación de alto rendimiento, Koduri ha trabajado en empresas como AMD y, más tarde, en Intel, donde impulsó la creación de la línea de GPU discretas Intel Xe. Ha promovido arquitecturas que integran procesamiento de gráficos y cómputo general para aplicaciones de IA y ha colaborado en la investigación sobre la convergencia entre gráficos y cargas de trabajo en IA, como la computación heterogénea. Su trabajo continúa expandiendo las capacidades de las GPU en el procesamiento paralelo, vital para IA y simulaciones.
Aportaciones:

- **Arquitectura de GPU en AMD**: Koduri jugó un papel fundamental en el desarrollo de **GPU de alto rendimiento** en **AMD**, particularmente con la línea **Radeon** y las arquitecturas **GCN** (Graphics Core Next). Estos avances han permitido que AMD se convierta en un competidor fuerte en el mercado de gráficos y cómputo paralelo.

- **Intel y la creación de chips de cómputo heterogéneo**: En **Intel**, Koduri ha liderado iniciativas para diseñar soluciones de **cómputo heterogéneo**, integrando **GPU** y **CPU** en un solo chip. Esto ha permitido mejorar la eficiencia en aplicaciones de **inteligencia artificial**, **big data** y **cómputo gráfico**.

- **Avances en el cómputo paralelo**: Koduri ha sido un defensor del cómputo paralelo y ha ayudado a impulsar las capacidades de procesamiento gráfico

y de propósito general (GPGPU), que son esenciales en áreas como el **cómputo de alto rendimiento (HPC)** y la **computación científica**.

Lenguaje ensamblador:

El **lenguaje ensamblador** (**assembly language**) es un lenguaje de programación de bajo nivel que permite escribir instrucciones específicas para la arquitectura de un procesador. A diferencia de los lenguajes de alto nivel, que abstraen las operaciones del hardware, el ensamblador se traduce directamente en instrucciones binarias que la CPU puede ejecutar. Cada línea de código en ensamblador generalmente corresponde a una instrucción de máquina, lo que permite a los programadores controlar de manera detallada los registros, la memoria y las operaciones en el procesador.

Cada tipo de procesador (como ARM, x86, o MIPS) tiene su propio conjunto de instrucciones, por lo que los programas en ensamblador son altamente dependientes de la arquitectura del hardware. Al trabajar a nivel de instrucciones de máquina, el ensamblador permite a los programadores manipular registros, la pila de llamadas, y gestionar de manera eficiente la memoria y el flujo de control, aspectos esenciales en sistemas embebidos y programación de sistemas críticos.

Dado que el ensamblador permite el control a nivel de bits, se utiliza frecuentemente en aplicaciones que requieren alta eficiencia, como sistemas embebidos y software de bajo nivel para hardware específico.

El lenguaje ensamblador es particularmente útil en áreas como el desarrollo de sistemas operativos, controladores de hardware, firmware de sistemas embebidos y en aplicaciones de seguridad, donde el rendimiento y el control detallado sobre el hardware son críticos.

La propuesta que elegimos para hacer en lenguaje ensamblador:

Optamos por implementar un método de ordenamiento de datos, el mas famoso es el **bubble sort**, pero a nosotros nos gustaría implementar un algoritmo más complejo, el algoritmo **Shell sort** es un poco mas complejo y efectivo que el **bubble**, normalmente los métodos iterativos suelen ser poco efectivos, pero **Shell sort** siendo iterativo es casi tan efectivo como los métodos de ordenamiento recursivos como el **merge sort** y el **quick sort**.
A continuación tenemos un código en **C++** del algoritmo de ordenamiento **Shell sort**:

```
template<class T, int ARRAYSIZE>
void Lista<T, ARRAYSIZE>::sortDataShell(ComparacionTipo tipo) {
    float factor(1.0 / 2.0);
    int dif((valorMaximo + 1) * factor), i, j;
    while (dif > 0) {
        i = dif;
        while (i <= valorMaximo) {
            j = i;
```

```
                while (j >= dif && canciones[j -
        dif].greaterThan(canciones[j], tipo)) {
                    swapData(canciones[j - dif], canciones[j]);
                    j -= dif;
                }
                i++;
            }
            dif *= factor;
        }
    }
```

OBJETIVOS

El objetivo de esta actividad es comprender la arquitectura MIPS, familiarizándose con sus componentes y cómo interactúan para ejecutar las instrucciones. Se enfocará en la implementación y coordinación de los elementos del procesador, como los registros, la ALU y los multiplexores, para asegurar que funcionen de manera eficiente, especialmente en la ejecución de instrucciones tipo R.

Uno de los aspectos fundamentales será implementar un decodificador de instrucciones, que convierte una instrucción de 32 bits en formato de ensamblador. Este decodificador interpreta los campos de la instrucción, como los registros de origen y destino, y las operaciones a realizar, permitiendo que los componentes del procesador ejecuten la operación correcta.

Además, esta actividad busca desarrollar habilidades en programación y diseño de sistemas, permitiendo a los participantes profundizar en conceptos clave de la arquitectura de computadoras. El trabajo en equipo será esencial, ya que los participantes podrán colaborar y compartir conocimientos sobre el funcionamiento y diseño del procesador MIPS.

DESARROLLO

Decodificador.py

El desarrollo del codificador se realizó tomando como base el código implementado en la actividad anterior. Partimos de ese diseño previo y, en lugar de comenzar desde cero, realizamos los ajustes necesarios para adaptarlo a los requisitos específicos de este proyecto.

```
import tkinter as tk
from tkinter import *
from tkinter import filedialog
from tkinter import messagebox
import re
import tkinter
```

Se importan varias herramientas clave para una interfaz gráfica en Python usando **tkinter**. La primera línea **import tkinter as tk**, carga el módulo principal de **tkinter**, que permite crear y gestionar ventanas y elementos gráficos como botones y etiquetas de manera más sencilla al utilizar el prefijo **tk.**
Además, se importan dos submódulos de **tkinter: filedialog**, que permiten abrir cuadros de diálogo para seleccionar o guardar archivos, y **messagebox**, que facilitan la creación de ventanas emergentes para mostrar mensajes de advertencia, error o confirmación. Finalmente, se importa el módulo **re**, que ofrece funciones para trabajar con expresiones regulares. Esto permite buscar y procesar patrones complejos en cadenas de texto, útiles en aplicaciones que requieren validar o manipular entradas de usuario.
Este código define una clase **Deco** que crea una aplicación gráfica (GUI) utilizando **tkinter**. La aplicación permite cargar archivos de texto, realizar decodificaciones sobre ellos, mostrar los resultados en un área de texto y guardar los resultados decodificados.
El método **__init__**es el constructor de la clase, responsable de inicializar todos los atributos y elementos de la interfaz gráfica. Se configuran tres diccionarios: **original_lines**, que guarda las líneas originales del archivo; **binary_results**, que almacena los resultados de la decodificación en formato binario; y **combined_results**, que combina las líneas originales con sus equivalentes binarios. También se crea la ventana principal de la aplicación con self.window = tk.Tk()y un área de texto, **self.textArea** donde se mostrará el contenido del archivo. Finalmente, se llama a los métodos **createWindow()** para configurar los elementos gráficos y **showGui()** para iniciar el ciclo de la interfaz gráfica.

```
class Deco:
    def __init__(self):
        self.original_lines = {}
        self.binary_results = {}
        self.combined_results = {}
        self.resultado_binario = ""
        self.window = tk.Tk()
        self.textArea = tk.Text(self.window, wrap=tk.WORD, height=25,
         width=80)
        self.createWindow()
        self.showGui()
```

El método **createWindow()** es el encargado de configurar la ventana principal y agregar los elementos gráficos, Se establece el título de la ventana como "**Decodificador**" y se definen sus dimensiones en 700x600 píxeles, proporcionando un espacio adecuado para la interfaz. Entre los elementos gráficos, se incluye una etiqueta (**Label**) que presenta el texto "**Text Analyzer**" en la parte superior de la ventana, lo que indica la función de la aplicación. Además, se añaden cinco botones con funciones clave: el botón "**Import file**" que activa la función para cargar un archivo, el botón "**Save operation**", lo que hace, es guardar

en un archivo "**txt**" la o las operaciones que se han ingresado, ejemplo de una operación: "***ADD*** **$0 $4 $0**", el botón "**Save result**", a diferencia de "**Save operation**", este botón hace que se guarden solamente los resultados de las operaciones, ejemplo: "**00000000100000000100000000100000**", el botón **"Save Both"**, guarda tanto la operación como el resultado, ejemplo: "***ADD*** **$0 $4 $0 = 00000000100000000100000000100000**" y por último el botón "**Decodificar**" que procesa el contenido del archivo cargado. También se incorpora un área de texto donde el usuario puede ver y editar el contenido del archivo. Cada botón está vinculado a un comando específico que ejecuta su función asociada cuando el usuario lo presiona, facilitando así la interacción con la interfaz.

```
def createWindow(self):
    self.window.title("Decoder")
    self.window.geometry("700x600")

    titleLabel = tk.Label(self.window, text="Text Analyzer",
   font=("Arial", 16))
    titleLabel.pack(pady=10)

    button1 = tk.Button(self.window, text="Import file",
   command=self.selectFile)
    button1.pack(pady=10)

    frame = Frame(self.window)
    frame.pack(pady=10)

    opcion1 = Button(frame, text="Save operation", command=lambda:
   self.saveFile(1))
    opcion1.pack(side=LEFT, padx=10)

    opcion2 = Button(frame, text="Save result", command=lambda:
   self.saveFile(2))
    opcion2.pack(side=LEFT, padx=10)

    opcion3 = Button(frame, text="Save Both", command=lambda:
   self.saveFile(3))
    opcion3.pack(side=LEFT, padx=10)

    button3 = tk.Button(self.window, text="Decode",
   command=self.decode)
    button3.pack(pady = 10)

    self.textArea.pack(pady=10)
```

Cuando el usuario selecciona un archivo utilizando el botón "Importar archivo", se activa la función **selectFile()**. Esta función presenta un cuadro de diálogo que permite al usuario elegir un archivo de texto con la extensión .**txt**. Si se selecciona

un archivo, se procede a cargar su contenido mediante la función **loadFile()**. Esta función abre el archivo en modo de lectura y, a continuación, lee todo su contenido. Una vez obtenido, el contenido se inserta en el área de texto de la aplicación, asegurándose de eliminar cualquier contenido anterior que pudiera estar presente.

```
def selectFile(self):
    file = filedialog.askopenfilename(filetypes=[("Text Files",
    "*.txt")], title="Select file")
    if file:
        self.loadFile(file)

def loadFile(self, file):
    with open(file, 'r', encoding='utf-8') as file_:
        content = file_.read()
    self.textArea.delete('1.0', tk.END)
    self.textArea.insert(tk.END, content)
```

El botón "**Guardar archivo**" activa la función **saveFile(opcion)**, que abre un cuadro de diálogo para guardar el contenido decodificado en un nuevo archivo.
Recibe como parámetro "**opción**", que es una variable de tipo entero, la cual tendrá un numero del 1 al 3, esto con la finalidad de hacer las diferentes formas de guardar un archivo, se cargará en el archivo un diccionario según el botón presionado.
Si se presionó el botón "Save operation", operación será igual a 1 y la función **saveFile(opcion)** guardará el contenido del diccionario "original_lines" en el archivo de texto.
Si se presionó el botón "**Save result**", operación será igual a 2 y la función **saveFile(opcion)** guardará el contenido del diccionario "**binary_results**" en el archivo de texto.
Y por último, Si se presionó el botón "**Save Both**", operación será igual a 2 y la función **saveFile(opcion)** guardará el contenido del diccionario "**combined_results**" en el archivo de texto.
Independientemente del botón que se haya presionado para guarda un archivo, se podrá elegir la ubicación y el nombre del archivo de salida.

```
def saveFile(self, opcion):
    file = filedialog.asksaveasfilename(defaultextension=".txt",
   filetypes=[("Text Files", "*.txt")])
    if file:
        with open(file, 'w', encoding='utf-8') as file_:
            if opcion == 1:
                for line, resultado in self.original_lines.items():
                    file_.write(f"{resultado}\n")
            elif opcion == 2:
                for line, resultado in self.binary_results.items():
                    file_.write(f"{resultado}\n")
```

```
            elif opcion == 3:
                for line, resultado in self.combined_results.items():
                    file_.write(f"{resultado}\n")
```

El método **decode()** es una de las partes más críticas del código, ya que se encarga de procesar el texto que se ha ingresado o cargado en el área de texto, buscando palabras clave específicas y convirtiéndolas en su representación binaria correspondiente. Para comenzar, se define un diccionario llamado **keywords**, que asocia palabras clave con sus códigos binarios. Por ejemplo, la palabra "**suma**" se traduce a "000000_100000" y "**resta**" a "000000_100010". El proceso comienza obteniendo el contenido completo del área de texto mediante **self.textArea.get()**, el cual se divide en líneas individuales para su análisis.

Para cada línea, se busca la presencia de alguna palabra clave del diccionario. Al encontrar una, se procede a buscar hasta tres símbolos "**$**", que indican la posición de los números que deben ser convertidos a binario. Si la línea contiene los tres símbolos "**$**" seguidos de sus respectivos números, estos son extraídos y convertidos a su representación binaria utilizando la función **convertBinary()**.

Solamente cuando se asocia la palabra clave "**nop**" solamente se insertaran puros "0" en la instrucción, Posteriormente, se construye una cadena binaria que combina el código asociado con la palabra clave y los números extraídos de la siguiente manera: según la palabra asociada como "**suma**" "000000_100000", se extraen los 6 primeros bits, después se inserta el segundo numero que se encuentra después de un "**$**" en 5 bits, después se inserta el último número adyacente de un "**$**", en 5 bit, después se inserta el primer número adyacente de un "**$**", por último se inserta "00000" que pertenecen a "**SHAMT**" y los 6 bits que pertenecen a.

Una vez que se ha decodificado la línea, el área de texto se actualiza con el resultado, reemplazando el contenido original. Esto permite que el usuario vea tanto la línea original como su equivalente en formato binario, facilitando la comprensión del resultado del procesamiento.

```
def decode(self):
    keywords = {
        # Tipo I
        "addi": "001000",
        "slti": "001010",
        "andi": "001100",
        "ori":  "001101",
        "sw":   "101011",
        "lw":   "100011",
        "beq":  "000100",
        # Tipo R
        "add": "000000_100000",
        "sub": "000000_100010",
        "slt": "000000_101010",
        "and": "000000_100100",
```

```
    "xor": "000000_100110",
    "nor": "000000_100111",
    "or":  "000000_100101",
    "nop": "000000_000000"
}

content = self.textArea.get("1.0", tk.END).strip()  # Obtén todo el
 contenido
lines = content.splitlines()

for line_num, line in enumerate(lines):
    line_lower = line.lower()  # Convierte la línea a minúsculas

    if not line_lower.strip():  # Ignorar líneas vacías
        continue

    for keyword, code in keywords.items():
        if keyword in line_lower:  # Usar la línea en minúsculas
 para buscar
            count_dollar = 0  # Contador de símbolos $
            pos = 0  # Posición para comenzar la búsqueda en la
 línea

            if keyword == "nop":
                count_dollar = 3
                num1 = "00000"
                num2 = "00000"
                num3 = "00000"

            # Tipo I ////////////////////////////////
            if len(code) == 6:
                immediate_value = None  # Para capturar el valor
 inmediato
                while count_dollar < 2:
                    dollar_index = line.find("$", pos)
                    if dollar_index != -1:
                        count_dollar += 1
                        pos = dollar_index + 1

                        rest_of_line = line[pos:].strip()
                        match = re.match(r'(\d+)', rest_of_line)

                        if match:
                            number_after_dollar = match.group(1)
                            pos += len(number_after_dollar)
                            if count_dollar == 1:
                                num1 = number_after_dollar  #
 Registro destino
```

```
                            elif count_dollar == 2:
                                num2 = number_after_dollar  #
Registro fuente
                        else:
                            messagebox.showerror("Error", "No hay
número después de $.")
                            return
                    else:
                        break

                # Captura el valor inmediato (después de #)
                immediate_index = line.find("#", pos)
                if immediate_index != -1:
                    rest_of_line = line[immediate_index +
1:].strip()
                    match = re.match(r'(-?\d+)', rest_of_line)
                    if match:
                        immediate_value = match.group(1)
                    else:
                        messagebox.showerror("Error", "No hay valor
inmediato válido.")
                        return
                else:
                    messagebox.showerror("Error", "Falta el valor
inmediato (#).")
                    return

                if immediate_value is None:
                    messagebox.showerror("Error", "Instrucción tipo
I incompleta.")
                    return

                # Construir binario para tipo I
                resultado_binario =
f"{code}{self.convertBinary(int(num2),
5)}{self.convertBinary(int(num1),
5)}{self.convertBinary(int(immediate_value), 16)}\n"
            else: # Tipo R
                while count_dollar < 3:  # Se esperan 3 símbolos $
                    dollar_index = line.find("$", pos)
                    if dollar_index != -1:
                        count_dollar += 1
                        pos = dollar_index + 1  # Mover la posición
después del $

                        rest_of_line = line[pos:].strip()  # Buscar
el número después del $
                        match = re.match(r'(\d+)', rest_of_line)
```

```
                        if match:
                            number_after_dollar = match.group(1)
                            pos += len(number_after_dollar)  #
Actualiza la posición después del número
                            if count_dollar == 1:  # Guardar los
números en variables
                                num1 = number_after_dollar
                            elif count_dollar == 2:
                                num2 = number_after_dollar
                            elif count_dollar == 3:
                                num3 = number_after_dollar
                        else:
                            messagebox.showerror("Error", "No hay
número después de $.")
                            return
                    else:
                        break

                if count_dollar != 3:
                    messagebox.showerror("Error", "Operación
incompleta, faltan símbolos $.")
                    return

                resultado_binario =
f"{code[:6]}{self.convertBinary(int(num2),
5)}{self.convertBinary(int(num3), 5)}{self.convertBinary(int(num1),
5)}{'00000'}{code[-6:]}\n"

            if line_num not in self.binary_results or
self.binary_results[line_num] != resultado_binario.strip():
                if "=" in line:
                    self.original_lines[line_num] =
line.split('=')[0].strip()
                    self.textArea.insert(f"{line_num + 1}.0",
self.combined_results[line_num])
                else:
                    self.original_lines[line_num] = line
                self.binary_results[line_num] =
resultado_binario.strip()  # Actualiza directamente
                self.combined_results[line_num] =
f"{self.original_lines[line_num]} =
{self.binary_results[line_num]}"
                self.textArea.delete(f"{line_num + 1}.0",
f"{line_num + 1}.end")  # Limpiar la línea
                self.textArea.insert(f"{line_num + 1}.0",
self.combined_results[line_num])  # Mostrar el resultado combinado
            else:
```

```
                    self.textArea.delete(f"{line_num + 1}.0",
        f"{line_num + 1}.end")  # Limpiar la línea
                    self.textArea.insert(f"{line_num + 1}.0",
        self.combined_results[line_num])

                break  # Salir del bucle de keywords, ya que se ha
        encontrado uno

            else:
                messagebox.showwarning("Error", "Operación no válida.")
```

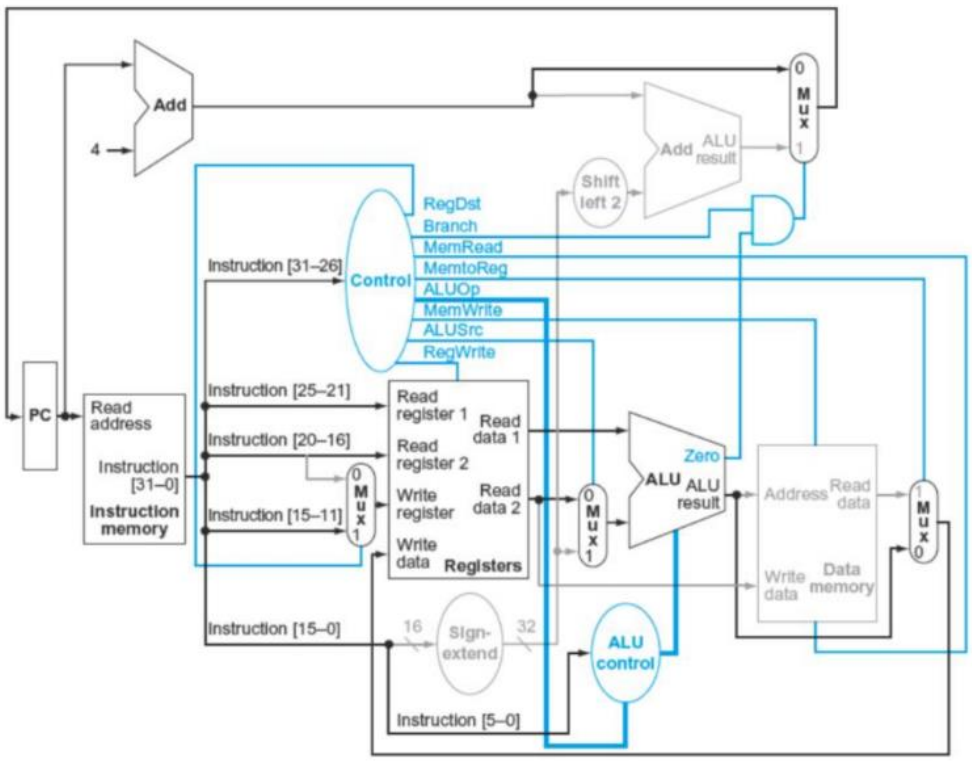
La función **convertBinary()** es de una utilidad simple, que recibe un número entero, lo convierte en su representación binaria, y asegura que esta tenga siempre 5 bits de longitud. Si el número binario es más corto, la función agrega ceros a la izquierda para que el formato sea consistente, algo necesario en muchos sistemas digitales que requieren un tamaño fijo.

Por último, el método **showGui()** se encarga de ejecutar el bucle principal de la aplicación gráfica usando **self.window.mainloop()**. Este bucle mantiene la ventana abierta, permitiendo que la interfaz gráfica responda a las interacciones del usuario, como presionar botones o ingresar texto, hasta que el programa se cierre.

```
def convertBinary(self, number, bits):
    if isinstance(number, int):
        # Verificar rango
        max_value = (1 << bits) - 1
        min_value = -(1 << (bits - 1))
        if not (min_value <= number <= max_value):
            raise ValueError(f"El número {number} está fuera del rango
     para {bits} bits.")

        # Manejo de números negativos (complemento a dos)
        if number < 0:
            number = (1 << bits) + number

        if bits == 5:
            binary = str(bin(number)[2:]).zfill(5)
            return binary
        elif bits == 16:
            binary = str(bin(number)[2:]).zfill(16)
            return binary
    else:
        raise TypeError(f"El numero: {number} es incorrecto.")

def showGui(self):
    self.window.mainloop()
```

DataPath

Para iniciar esta actividad, crearemos un proyecto denominado "datapath" en ModelSim, donde se guardarán los archivos correspondientes en lenguaje Verilog. Cada módulo requerido se implementará en un archivo separado, siguiendo el diagrama proporcionado.



unidadControl.v

El código Verilog del módulo U_controlestá diseñado para generar señales de control en el sistema. El módulo recibe un código de operación de 6 bits, opCode que determina el comportamiento del sistema. A partir de este código, se generan señales de control que incluyen "**ALUOP**" que le manda 3 bits a la unidad "**aluControl**" para que le diga a la "**ALU**" la operación que tiene que realizar, otra señal de control es "**RegWrite**" que sirve para habilitar la escritura del "**Banco de registros**", todas las demás salidas no son útiles momentáneamente ya que se utilizan para las instrucciones de "**tipo I**", y en este caso solamente vamos a implementar instrucciones "**tipo R**".

```
`timescale 1ps/1ps

// Unidad de control ////////////////////////
module U_control (
    input wire [5:0] opCode,     // Entrada: opcode de 6 bits
    output reg RegDst,           // Salida: Mux 3 <--
    output reg Branch,           // Salida: Branch (Mux 2)
    output reg MemRead,          // Salida: Memoria datos leer
    output reg MemToReg,         // Salida: Mux 1 <--
    output reg [2:0] ALUOP,      // Salida: operacion de la ALu (3 bits)
     <--
    output reg MemWrite,         // Salida: Memoria datos escribir
    output reg ALUsrc,           // Salida: Mux 4 <--
    output reg RegWrite          // Salida: Banco de registros <--
);

    always @(*) begin

        case (opCode)
            6'b000000: begin // Tipo R
                RegDst = 1;
                Branch = 0;
                MemRead = 0;
                MemToReg = 0;
                ALUOP = 3'b000;
                MemWrite = 0;
                ALUsrc = 0;
                RegWrite = 1;
            end
            6'b001000: begin // Tipo ADDI
                RegDst = 0;
                Branch = 0;
                MemRead = 0;
                MemToReg = 0;
                ALUOP = 3'b010;
                MemWrite = 0;
                ALUsrc = 1;
```

```
        RegWrite = 1;
    end
    6'b001010: begin // Tipo SLTI
        RegDst = 0;
        Branch = 0;
        MemRead = 0;
        MemToReg = 0;
        ALUOP = 3'b111;
        MemWrite = 0;
        ALUsrc = 1;
        RegWrite = 1;
    end
    6'b001100: begin // Tipo ANDI
        RegDst = 0;
        Branch = 0;
        MemRead = 0;
        MemToReg = 0;
        ALUOP = 3'b011;
        MemWrite = 0;
        ALUsrc = 1;
        RegWrite = 1;
    end
    6'b001101: begin // Tipo ORI
        RegDst = 0;
        Branch = 0;
        MemRead = 0;
        MemToReg = 0;
        ALUOP = 3'b001;
        MemWrite = 0;
        ALUsrc = 1;
        RegWrite = 1;
    end
    6'b101011: begin // Tipo SW
        RegDst = 0;
        Branch = 0;
        MemRead = 0;
        MemToReg = 0;
        ALUOP = 3'b010;
        MemWrite = 1;
        ALUsrc = 1;
        RegWrite = 0;
    end
    6'b100011: begin // Tipo LW
        RegDst = 0;
        Branch = 0;
        MemRead = 1;
        MemToReg = 1;
        ALUOP = 3'b010;
```

```
                MemWrite = 0;
                ALUsrc = 1;
                RegWrite = 1;
            end
            6'b000100: begin // Tipo BEQ
                RegDst = 0;
                Branch = 1;
                MemRead = 0;
                MemToReg = 0;
                ALUOP = 3'b110; // hacer resta par saber si son iguales
                MemWrite = 0;
                ALUsrc = 0;
                RegWrite = 0;
            end

        endcase
    end
endmodule
```

mux2_1 32bits.v

El módulo **mux2_1 32bits** está diseñado para seleccionar entre dos entradas de 32 bits, **a** y **b**, en función de una señal de selección **sel**. El módulo tiene tres puertos: la entrada **sel**, las entradas **a** y **b**, y la salida **c**, que también es un registro de 32 bits. El bloque **always** @*se activa cuando hay cambios en cualquiera de las señales de entrada. Dentro de este bloque, se utiliza una estructura condicional if para determinar cuál de las dos entradas se asignará a la salida c. Si seles verdadero (1), **c** tomará el valor de **b**; De lo contrario, tomará el valor de **a**.

```
`timescale 1ps/1ps

// Mux 2 a 1 ////////////////////////
module mux2_1_32bits (
    input sel,
    input [31:0] a,
    input [31:0] b,
    output reg [31:0] c
);

always @* begin
    if (sel) begin
        c = b;
    end
    else begin
        c = a;
```

```
    end
end
endmodule
```

mux2_1 5bits.v

El módulo **mux2_1 5bits** es exactamente igual al módulo **mux2_1 32bits,** el único cambio es que tanto las entradas como las salidas son de 5 bits.

```
`timescale 1ps/1ps

// Mux 2 a 1 ////////////////////////
module mux2_1_5bits (
    input sel,
    input [4:0] a,
    input [4:0] b,
    output reg [4:0] c
);

always @* begin
    if (sel) begin
        c = b;
    end
    else begin
        c = a;
    end
end
endmodule
```

aluControl.v

El módulo **aluControl** está diseñado para determinar la operación que realizará la ALU en función de las señales de entrada. Recibe dos entradas: **UC_input**, un vector de 3 bits que proviene de la unidad de control, y **funct**, un vector de 6 bits que especifica la función deseada en las operaciones de tipo R. La salida **AC_output** es un registro de 3 bits que indica qué debe ejecutar la ALU.

El bloque **always @(*)** se activa ante cambios en las entradas. Utiliza una estructura **case** para determinar el comportamiento según el valor de **UC_input**. Si **UC_input** es 3'b000, entonces **aluControl** trabajara con instrucciones **tipo R** y le va a seleccionar una salida dependiendo la entrada **Funct** que es de 6 bits, este tipo de instrucciones son las que se utilizan para seleccionar una operación en la arquitectura mips 32, un ejemplo: si se recibe "**100000**" que es una suma (**add**) entonces la salida será "**010**", esto se debe a que la **ALU** con la que estamos trabajando asi tiene asignadas las operaciones.

```
`timescale 1ps/1ps

// ALU Control ////////////////////////////////
```

```
module aluControl(
    input [2:0] UC_input,
    input [5:0] funct,

    output reg [2:0] AC_output
);

always @(*) begin
    case (UC_input)
        3'b000:
            case (funct)
                6'b100000: AC_output = 3'b010; // add
                6'b100010: AC_output = 3'b110; // sub
                6'b101010: AC_output = 3'b111; // slt
                6'b100100: AC_output = 3'b000; // and
                6'b100101: AC_output = 3'b001; // or
                6'b100110: AC_output = 3'b101; // xor
                6'b100111: AC_output = 3'b100; // nor
                6'b000000: AC_output = 3'b011; // nop (no operacion)
            endcase
        3'b010: // add
            AC_output = 3'b010;
        3'b110: // sub
            AC_output = 3'b110;
        3'b111: // slt
            AC_output = 3'b111;
        3'b011: // and
            AC_output = 3'b000;
        3'b001: // or
            AC_output = 3'b001;

        default: AC_output = 3'bxxx;
    endcase
end
endmodule
```

alu.v

El módulo **alu** está diseñado para realizar diversas operaciones aritméticas y lógicas sobre dos entradas de 32 bits, **dataInput1** y **dataInput2**, según una selección proporcionada por la entrada **sel**, que es un vector de 3 bits. Dentro del módulo, se define un bloque always @ (*) que se activa con cambios en cualquiera de las señales de entrada. Este bloque utiliza una estructura case para determinar qué operación se debe realizar en función del valor de **sel**. Las operaciones implementadas son:

- **AND** (sel = 3'b000): Realiza una operación AND bit a bit entre dataInput1y dataInput2.
- **OR** (sel = 3'b001): Realiza una operación OR bit a bit entre las dos entradas.

- **Suma** (sel = 3'b010): Suma los dos valores de entrada.
- **Resto** (sel = 3'b110): Resto dataInput2de dataInput1.
- **Comparación** (sel = 3'b111): Devuelve 1 si dataInput1 es menor que dataInput2, de lo contrario devuelve 0.
- **NOR** (sel = 3'b100): Realiza una operación NOR bit a bit entre las dos entradas.
- **XOR** (sel = 3'b101): Realiza una operación XOR un poco entre las dos entradas. El resultado de la operación se almacena en **dataOutput**, que es un registro de 32 bits.
- **NOP** (sel = 3'b011): El propósito de **nop** es no realizar ninguna operación, es por eso que esta operación simplemente pone **0´s** en la salida **dataOutput.**

```
`timescale 1ps/1ps

// Alu ////////////////////////////////
module alu (
    input [31:0] dataInput1,
    input [31:0] dataInput2,
    input [2:0] sel,
    output reg [31:0] dataOutput,
    output reg zero
);

always @(*) begin
    zero = 0;
    case (sel)
        3'b000: begin
            dataOutput = dataInput1 & dataInput2;
        end
        3'b001: begin
            dataOutput = dataInput1 | dataInput2;
        end
        3'b010: begin
            dataOutput = dataInput1 + dataInput2;
        end
        3'b110: begin
            dataOutput = dataInput1 - dataInput2;
            if (dataOutput == 0) begin
                zero = 1;
            end
        end
        3'b111: begin
            dataOutput = (dataInput1 < dataInput2) ? 32'd1 : 32'd0;
        end
        3'b100: begin
            dataOutput = ~(dataInput1 | dataInput2);
        end
        3'b101: begin
```

```
            dataOutput = dataInput1 ^ dataInput2;
        end
        3'b011: begin
            dataOutput = 32'b0; // NOP (No operación)
        end
    endcase
end
endmodule
```

bancoRegistros.v

El módulo Banco_registros está diseñado para almacenar y gestionar registros de 32 bits, permitiendo operaciones de lectura y escritura. Este módulo tiene varios puertos: **AR1** y **AR2** son las direcciones de los registros que se quieren leer (entrada de 5 bits cada una), **AW** es la dirección del registro donde se desea escribir (también de 5 bits), **DW** es el dato de 32 bits que se escribirá, y **EnW** es una señal de habilitación para la escritura. Las salidas **DR1** y **DR2** son los datos leídos de los registros correspondientes.

La memoria se implementa como un arreglo de registros **mem**, que puede contener hasta 64 palabras de 32 bits cada una. En el bloque initial, se carga la memoria desde un archivo de texto llamado "**Datos.txt**" utilizando $readmemh, que permite inicializar la memoria con datos predefinidos. El bloque always @(*) se verifica si **EnW** está activado; Si es así, escriba el valor de **DW** en el registro correspondiente a **AW**. Luego, se leen los valores de los registros indicados por **AR1** y **AR2**, que se asignan a las salidas **DR1** y **DR2**, respectivamente.

```
`timescale 1ps/1ps

// Banco de registros ////////////////////////
module Banco_registros (
    input[4:0] AR1,
    input[4:0] AR2,
    input[4:0] AW,
    input[31:0] DW,
    input EnW,
    output reg[31:0] DR1,
    output reg[31:0] DR2
);

reg [31:0] mem [0:63];

initial begin
    $readmemh("Datos.txt", mem);
    #10;
end

always @(*) begin
```

```
    if (EnW) begin
        mem[AW] = DW;
    end
    DR1 = mem[AR1];
    DR2 = mem[AR2];
end
endmodule
```

pc.v

Este módulo tiene como propósito, contar desde el 0 hasta el 32, (posteriormente se puede modificar esto), este contrador es útil para poder acceder a las direcciones del componente **readAddress** que es una **ROM** la cual almacena todas las instrucciones que realizará todo el módulo, el contador se retroalimenta con un sumador el cual siempre le va a sumar 4 al lo que saque el contador y se regrasara este resultado al contador para que ahora tome este valor, gracias a esto siempre se tendrán múltiplos de 4, esto es muy conveniente para la arquitectura **mips**.

```
`timescale 1ps/1ps

// Add ////////////////////////////////
module ADD (
    input [4:0] A_ADD,
    input [4:0] B_ADD,
    output [4:0] C_ADD
);
assign C_ADD = A_ADD + B_ADD;
endmodule

// PC /////////////////////////////////
module PC (
    input wire clk,
    input wire [4:0] pc_in,
    output reg [4:0] pc_out
);

    wire [4:0] increment;

    ADD iADD(
        .A_ADD(pc_out),
        .B_ADD(5'd4),
        .C_ADD(increment)
    );

    initial begin
        pc_out = 0;
    end
```

```
    always @(posedge clk) begin
        if (pc_out == 5'd31)
            pc_out <= 0;
        else
            pc_out <= increment;
    end

endmodule
```

readAddress.v

Este modulo es una memoria **ROM**, lo que significa que solamente se va a leer de ella, en esta memoria vamos a almacenar todas las instrucciones que queramos colocarle a todo el módulo, aquí es donde almacenaremos las instrucciones que generemos gracias al programa que hicimos en Python "**Decoder**", esta memoria lee un archivo **.txt** y almaceta el contenido en la memoria. El archivo se llama "**instrucciones.txt**", asi fue como lo guardamos en el programa decoder.

```
`timescale 1ps/1ps

// Read Address ////////////////////////////////
module readAddress(
    input wire [4:0] i,
    output reg [31:0] TB_out
);

    reg [31:0] mem[0:31];

    initial begin
        $readmemb("instrucciones.txt", mem);
    end

    always @(*) begin
        TB_out = mem[i];
    end
endmodule
```

topLevel.v

El módulo **topLevel** es el núcleo del diseño de procesador MIPS, que integra varios componentes para ejecutar instrucciones de 32 bits. Recibe una entrada la cual es el ciclo de reloj (**CLK**)y produce dos salidas, una correspondiente al resultado de las operaciones y otra para visualizar el contador (**PC**). La Unidad de Control (**UC**) genera señales que gestionan el flujo de datos, la escritura en el banco de registros y las operaciones de la **ALU**. La ALU Control (**AC**) define las operaciones específicas a realizar, mientras que el Banco de Registros (**BR**) almacena y permite la lectura y escritura de datos. La ALU realiza operaciones aritméticas y lógicas, y un

multiplexor (**MUX**) selecciona entre la salida de la **ALU** y la memoria de datos, controlada por la **UC**, además tenemos el contador (**PC**) que se encarga de redireccionar al módulo (**readAddress**) el cual es una memoria **ROM**, y esta memoria cargara todas las instrucciones que queramos y que decodificamos gracias al programa hecho en python "**Decoder**".
La salida final del módulo **topLevel** es el resultado del multiplexor, que puede ser un dato de la ALU o de la memoria. En paralelo, el módulo **TB_topLevel** actúa como un banco de pruebas para verificar el funcionamiento del **topLevel.**

```
`timescale 1ps/1ps

// topLevel ////////////////////////////////
module topLevel (
    input wire clk,
    output reg [31:0] salida,
    output reg [4:0] pc
);

    // Salidas de: UNIDAD DE CONTROL (UC)
    wire UC_MUX_3;
    wire UC_MUX_1;
    wire [2:0] UC_AC;
    wire UC_MUX_2; // Branch
    wire UC_MUX_4;
    wire UC_BR;
    wire UC_MD_R;
    wire UC_MD_W;

    // Salidas de: BANCO DE REGISTROS (BR)
    wire [31:0] BR_ALU;
    wire [31:0] BR_MUX_4;

    // Salidas de: MUX 1
    wire [31:0] MUX_1_BR;

    // Salidas de: ALU CONTROL (AC)
    wire [2:0] AC_ALU;

    // Salidas de: ALU (ALU)
    wire [31:0] ALU_MD_MUX;
    wire ALU_MUX_2;

    // Salidas de: MEMORIA DATOS (MD)
    wire [31:0] MD_MUX_1;

    // Salidas de: MUX 2
    wire [4:0] MUX_2_PC;
```

```
// Salidas de: PC
wire [4:0] PC_MUX_2_RA;

// Salidas de: Read Address (RA)
wire [31:0] RA_;

// Salidas de: MUX 3
wire [4:0] MUX_3_BR;

// Salidas de: SE
wire [31:0] SE_MUX_4;

// Salidas de: MUX 4
wire [31:0] MUX_4_ALU;

// Salidas de: SL2
wire [31:0] SL2_MUX_2;

U_control iUnidadControl(
    .opCode(RA_[31:26]),
    .Branch(UC_MUX_2),
    .RegDst(UC_MUX_3),
    .MemRead(UC_MD_R),
    .MemWrite(UC_MD_W),
    .MemToReg(UC_MUX_1),
    .ALUOP(UC_AC),
    .ALUsrc(UC_MUX_4),
    .RegWrite(UC_BR)
);

aluControl iAluControl(
    .UC_input(UC_AC),
    .funct(RA_[5:0]),
    .AC_output(AC_ALU)
);

alu iALU(
    .dataInput1(BR_ALU),
    .dataInput2(MUX_4_ALU),
    .sel(AC_ALU),
    .dataOutput(ALU_MD_MUX),
    .zero(ALU_MUX_2)
);

Banco_registros iBancoRegistros(
    .AR1(RA_[25:21]),
    .AR2(RA_[20:16]),
```

```
    .AW(MUX_3_BR),
    .DW(MUX_1_BR),
    .EnW(UC_BR),
    .DR1(BR_ALU),
    .DR2(BR_MUX_4)
);

mux2_1_32bits_MUX_1 iMUX_1(
    .sel(UC_MUX_1),
    .a(MD_MUX_1),
    .b(ALU_MD_MUX),
    .c(MUX_1_BR)
);

Mem_datos iMemoriaDatos(
    .dataInput(ALU_MD_MUX),
    .EnW(UC_MD_W),
    .EnR(UC_MD_R),
    .dataOutput(MD_MUX_1)
);

mux2_1_5bits iMUX_2(
    .sel(1'b0),
    .a(PC_MUX_2_RA),
    .b(SL2_MUX_2),
    .c(MUX_2_PC)
);

PC iPC(
    .clk(clk),
    .pc_in(MUX_2_PC),
    .pc_out(PC_MUX_2_RA)
);

readAddress iReadAddress(
    .i(PC_MUX_2_RA),
    .TB_out(RA_)
);

mux2_1_5bits iMUX_3(
    .sel(UC_MUX_3),
    .a(RA_[20:16]),
    .b(RA_[15:11]),
    .c(MUX_3_BR)
);

sign_extend iSE(
    .in(RA_[15:0]),
```

```
        .out(SE_MUX_4)
    );

    mux2_1_32bits_MUX_4 iMUX_4(
        .sel(UC_MUX_4),
        .a(BR_MUX_4),
        .b(SE_MUX_4),
        .c(MUX_4_ALU)
    );

    shift_left_2 iSL2(
        .inSingExtemd(SE_MUX_4),
        .inPC(PC_MUX_2_RA),
        .out(SL2_MUX_2)
    );

    always @* begin
        salida = ALU_MD_MUX; //MUX_1_BR
        pc = PC_MUX_2_RA;
    end

endmodule

// TB_topLevel ////////////////////////////////
module TB_topLevel;
    reg TB_clk;
    wire [31:0] TB_out;
    wire [4:0] TB_pc;

    topLevel uut(
        .clk(TB_clk),
        .salida(TB_out),
        .pc(TB_pc)
    );

    initial begin
        TB_clk = 0;
        forever #100 TB_clk = ~TB_clk;
    end
endmodule
```

Este código en Verilog define dos módulos. El módulo ADD realiza una suma de dos entradas de 32 bits y genera una salida con el resultado. El módulo shift_left_2desplaza una entrada de 32 bits a la izquierda por 2 bits y luego suma el resultado con otra entrada de 32 bits utilizando el módulo ADD. El resultado final de esta suma se asigna a la salida out. Ambos módulos realizan operaciones combinacionales de manera sencilla.

```
`timescale 1ps/1ps

// ADD ////////////////////////////////
module ADD (
    input [31:0] A_ADD,
    input [31:0] B_ADD,
    output [31:0] C_ADD
);
    assign C_ADD = A_ADD + B_ADD;
endmodule

// shift_left_2 ////////////////////////////////
module shift_left_2 (
    input  [31:0] inSingExtemd,
    input  [31:0] inPC,
    output [31:0] out
);

    ADD i(
        .A_ADD(inSingExtemd << 2),
        .B_ADD(inPC),
        .C_ADD(out)
    );

endmodule
```

Se implemento un módulo llamado sign_extend, que realiza una extensión de signo. Su propósito es tomar una entrada de 16 bits (in) y generar una salida de 32 bits (out) manteniendo el valor numérico correcto, tanto para números positivos como negativos.

La extensión de signo funciona evaluando el bit más significativo (MSB) de la entrada de 16 bits (in[15]). Si este bit es 1, significa que el número es negativo, por lo que los 16 bits más altos de la salida se llenan con 1s para mantener el signo negativo. Si el MSB es 0, lo que indica un número positivo, los 16 bits superiores de la salida se llenan con 0s.

La lógica utiliza una operación de concatenación y replicación de bits. La expresión {{16{in[15]}}, in}genera 16 copias del MSB de la entrada y las concatena con los originales de 16 bits in, formando un vector de 32 bits que se asigna a la salida out. Este módulo es común en sistemas digitales y procesadores donde se necesita trabajar con números de diferentes tamaños, asegurando que la representación de los datos sea correcta tras la conversión.

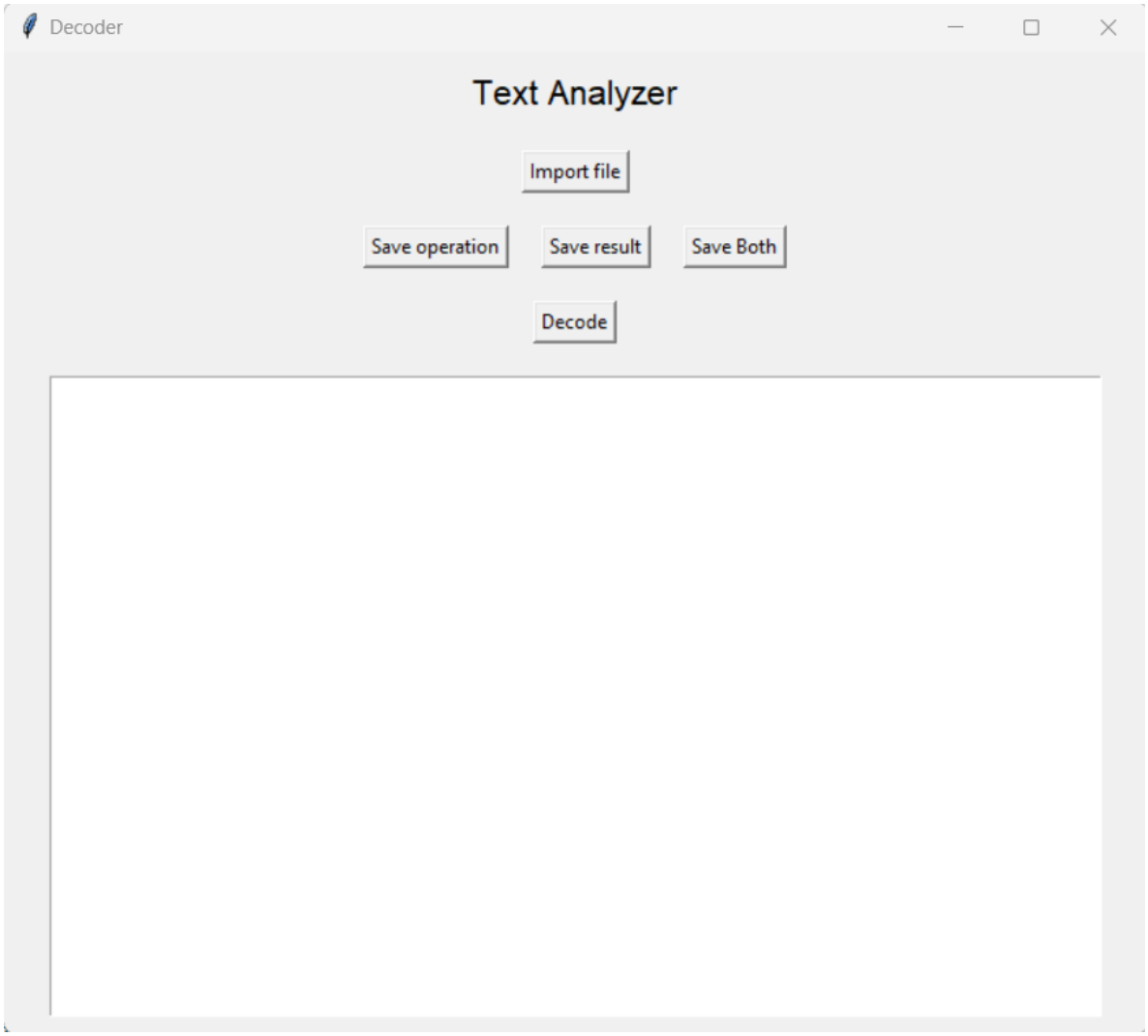
```
`timescale 1ps/1ps
module sign_extend (
    input  [15:0] in,   // Entrada de 16 bits
    output [31:0] out   // Salida de 32 bits
);

    // Si el bit mas significativo de la entrada es 1, los bits
    superiores de la salida se llenan con 1s.
    // Si el MSB es 0, los bits superiores de la salida se llenan con 0s.
    assign out = { {16{in[15]}}, in };
endmodule
```

El módulo Mem_datosen Verilog implementa una memoria de 64 posiciones de 32 bits. Permite escribir datos en la posición mem[0]si la señal EnW está activa y leer desde esa misma posición si EnR está activa. Actualmente, solo opera sobre la posición mem[0], pero puede ampliarse para manejar otras direcciones agregando una señal de dirección. Es una implementación básica para operaciones de lectura y escritura controladas.

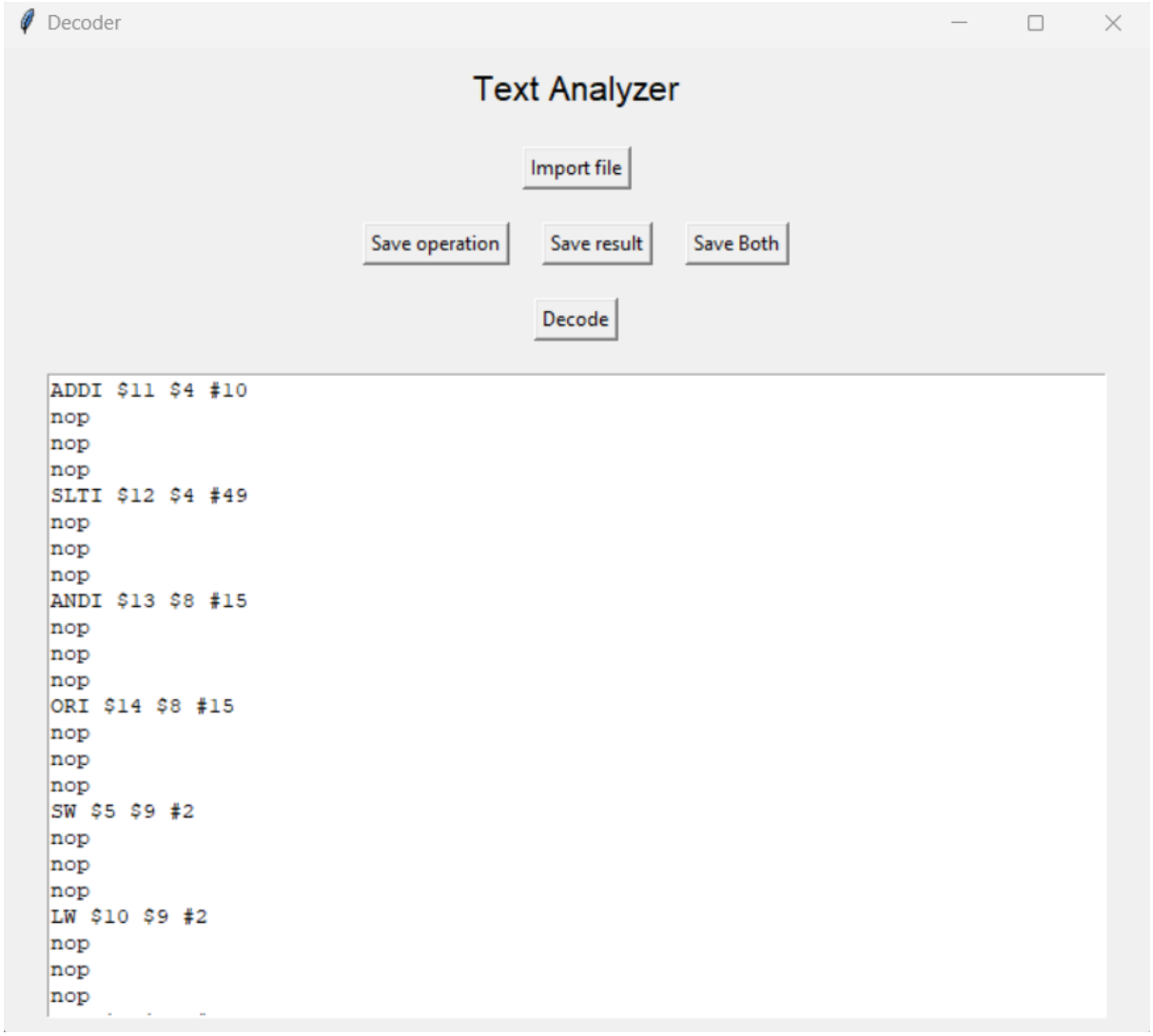
```
`timescale 1ps/1ps

// Memoria de datos ////////////////////////
module Mem_datos (
    input wire [31:0] dataInput,
    input wire EnW,
    input wire EnR,

    output reg[31:0] dataOutput
);

reg [31:0] mem [0:63];

always @(*) begin
    if (EnW) begin
        mem[0] = dataInput;
    end
    else if (EnR) begin
        dataOutput = mem[0];
    end
end
endmodule
```

Resultados

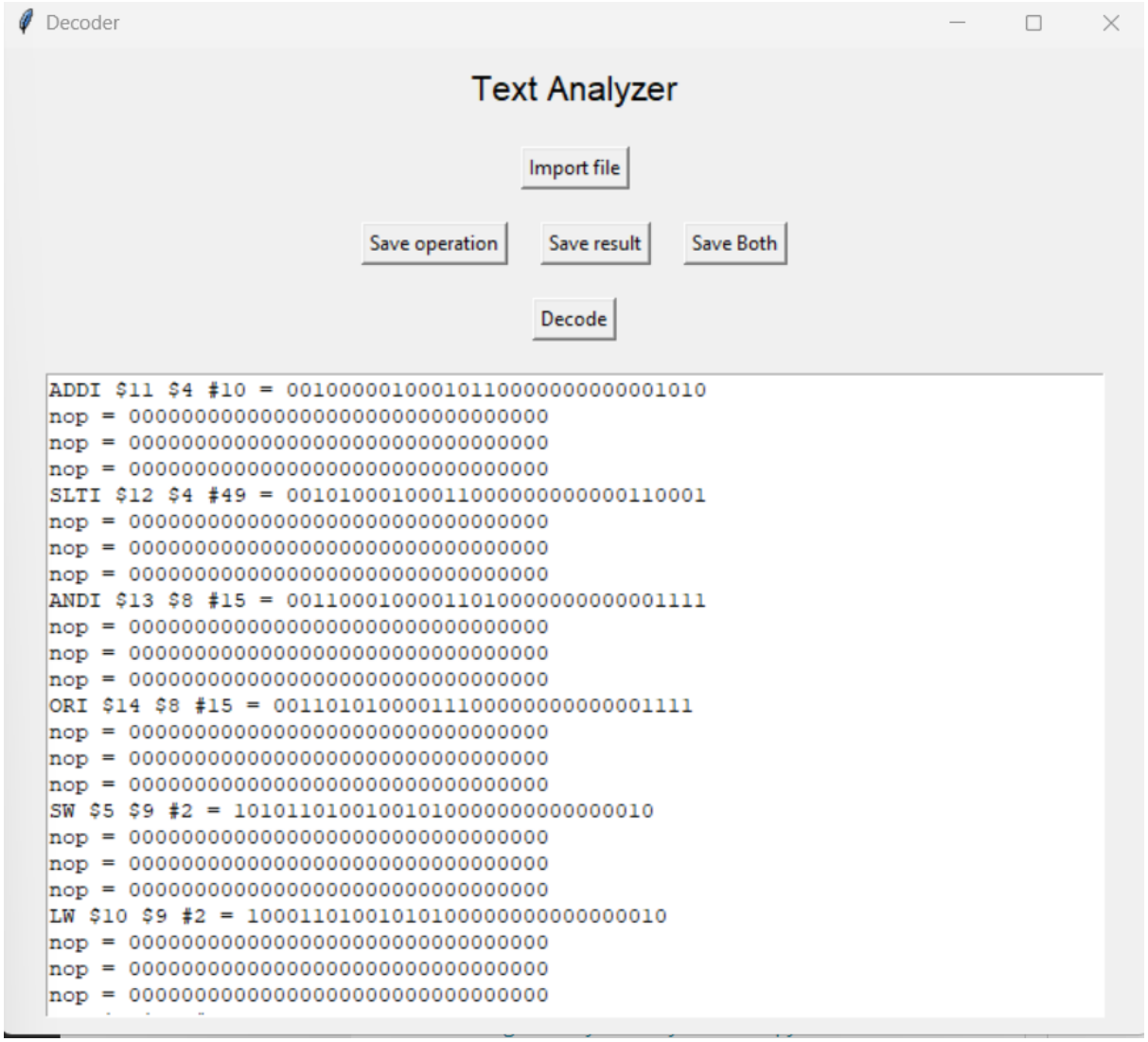
Para comprobar los resultados, primero utilizamos el programa que creamos "**Decodificador.py**"



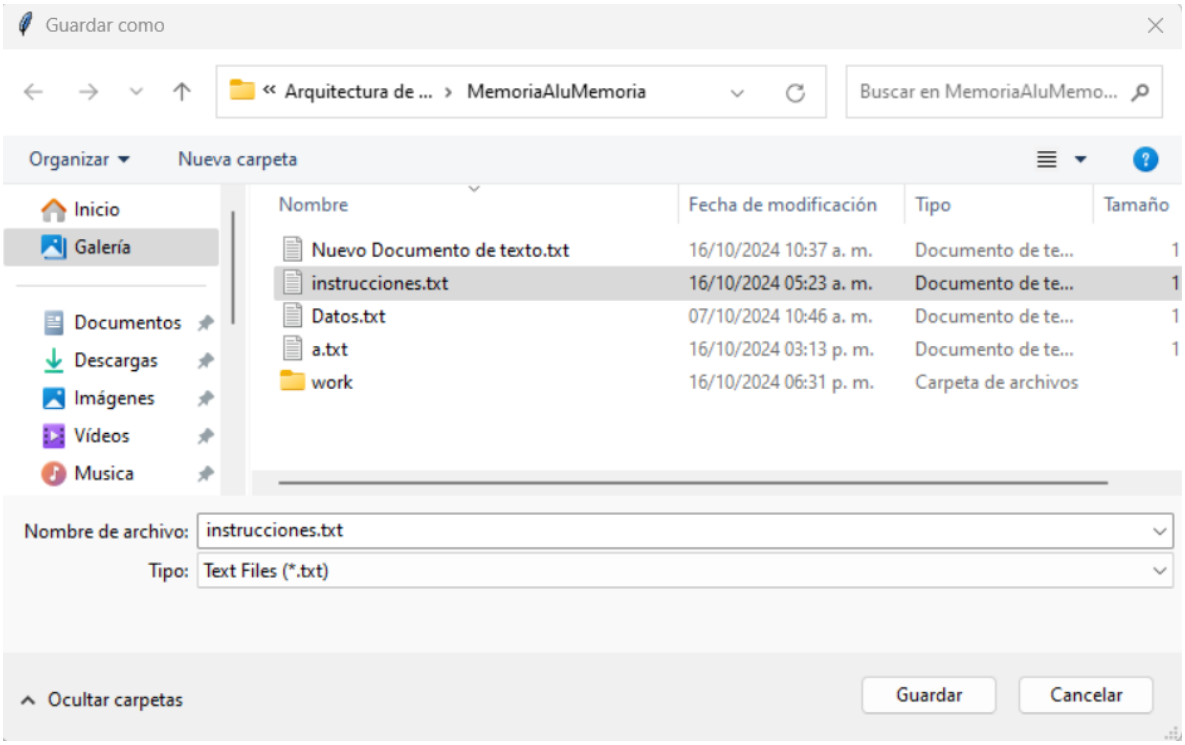
Después ingresamos 7 operaciones uno para cada operación, y colocamos "**nop**" para hacer espacios y asi la memoria "**read address**" pueda leer cada instrución:



Y damos clic en "**Decode**" para decodificar las operaciones:



Una vez que ya se obtuvieron los resultados, guardamos dichos resultados dando clic en "**Save result**", Nos pide que le demos nombre al archivo y que le demos una ubicación para guardar, o bien podemos sobreescribir un archivo existente:



Eso es todo lo que tenemos que hacer con "**Decodificador.py**", ahora probamos a "**datapath**". A la hora de hacer el testbench, ingresamos las instrucciones que se encuentran en la tabla que está a continuación:

OP	RS	RT	IMMEDIATE	Instrucción
001000	11	4	10	ADDI $11 $4 #10
001010	12	4	49	SLTI $12 $4 #49
001100	13	8	15	ANDI $13 $8 #15
001101	14	8	15	ORI $14 $8 #15
101011	5	9	2	SW $5 $9 #2
100011	10	9	2	LW $10 $9 #2
000100	5	10	1	BEQ $5 $10 #1
001000	15	3	325	ADDI $15 $3 #325

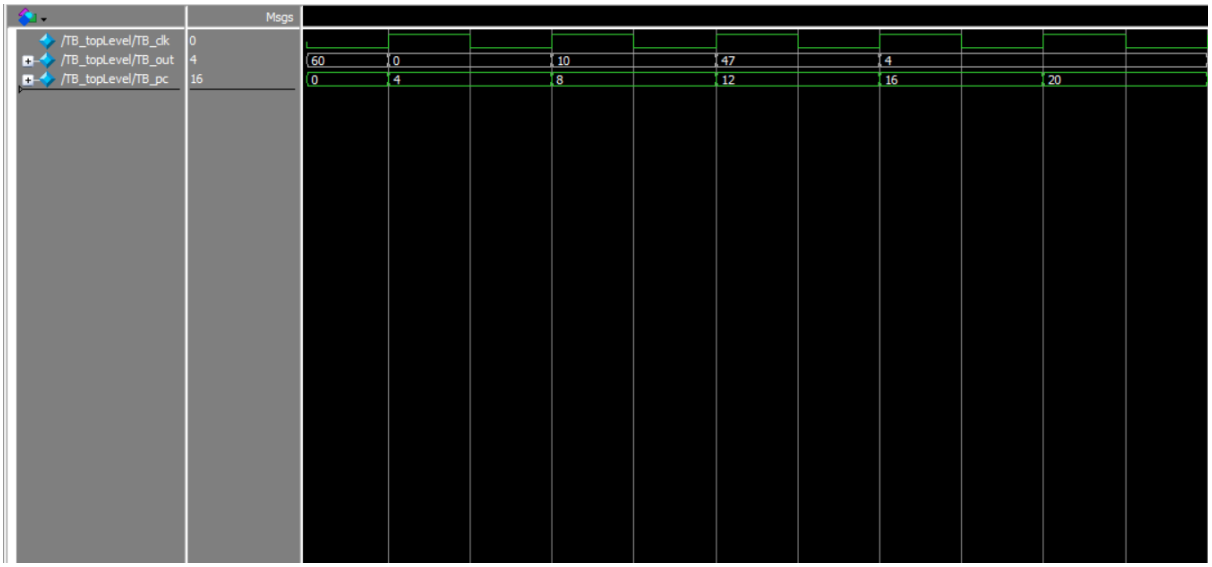
El banco de registros tiene los siguientes datos, y los resultados se van a almacenar después de los datos que ya tiene determinados.

Dir.	Datos
0	$56B_h$
1	$A3B4_h$
2	$1AA_h$
3	$2A3_h$
4	32_h
5	$3C_h$
6	55555555_h
7	$AAAAAAAA_h$

Las operaciones que se realizaron, junto a sus resultados, son los siguientes, (se convirtieron los datos del banco de registros a decimal o binario para apreciar mejor los resultados y se omitieron las instrucciones "**nop**"):

Operación	Resultado
$50_d + 1387_d$	1437_d
$41908_d - 1387_d$	40521_d
$50_d < 60_d$	1_b
$01010101010101010101010101010101_b$ ***and*** $10101010101010101010101010101010_b$	0_b
$01010101010101010101010101010101_b$ ***or*** $10101010101010101010101010101010_b$	$11111111111111111111111111111111_b$
$01010101010101010101010101010101_b$ ***xor*** $10101010101010101010101010101010_b$	$11111111111111111111111111111111_b$
$01010101010101010101010101010101_b$ ***nor*** $10101010101010101010101010101010_b$	0_b

Como se puede apreciar en la siguiente imagen, los resultados fueron correctos (el primer dato pertenece al resultado del "**clk**" el segundo resultado, el que está de color azul, pertenece al resultado de cada operación, y por último, el resultado del "**pc**":



CONCLUSIÓN

En conclusión, la actividad de diseño y simulación del procesador en Verilog nos ha permitido crear una arquitectura funcional basada en los conceptos fundamentales de los procesadores tipo MIPS y las instrucciones tipo R. A través de la implementación de módulos esenciales como la Unidad de Control, la ALU, el Banco de Registros, la Memoria de Datos y el Multiplexor, se ha logrado replicar el flujo de procesamiento de instrucciones típicas de un procesador MIPS, que sigue una estructura sencilla y eficiente, ideal para la enseñanza de conceptos de arquitectura de computadores.

Además, se añadió un contador que avanza en incrementos de cuatro hasta llegar a 32, cuyo objetivo es acceder secuencialmente a las instrucciones almacenadas en la memoria "**readAddress**". Esto permite simular el flujo de instrucciones que un procesador tipo MIPS seguiría al ejecutar su conjunto de instrucciones en orden.

El procesador tipo MIPS se caracteriza por su simplicidad y por utilizar instrucciones de longitud fija, como las instrucciones tipo R, que son instrucciones de registro a registro. Durante esta actividad, hemos trabajado específicamente con estas instrucciones tipo R, que incluyen operaciones aritméticas y lógicas, tales como suma, resta, y operaciones lógicas (AND, OR, XOR, entre otras). La Unidad de Control decodifica estas instrucciones, mientras que la ALU realiza los cálculos basados en los datos proporcionados por el Banco de Registros.

El banco de pruebas desarrollado (TB_topLevel) ha demostrado la capacidad del diseño para procesar correctamente una secuencia de instrucciones almacenadas en memoria. Este enfoque refuerza el entendimiento sobre cómo los procesadores tipo MIPS manejan las instrucciones, especialmente las tipo R, y la importancia de cada componente en el ciclo de instrucción: desde la decodificación hasta la ejecución y almacenamiento de resultados.

REFERENCIAS

- Stallings, W. (2012). Computer organization and architecture: Designing for performance (11.ª ed.). Hoboken, NJ: Pearson Education.
- Hennessy, J., Jouppi, N., Przybylski, S., Rowen, C., Gross, T., Baskett, F., & Gill, J. (1982). MIPS: A microprocessor architecture. ACM SIGMICRO Newsletter, 13(4), 17-22.
- Topiwala, M. N., & Saraswathi, N. (2014, May). Implementation of a 32-bit MIPS based RISC processor using Cadence. In 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies (pp. 979-983). IEEE.
- Kane, G. (1988). mips RISC Architecture. Prentice-Hall, Inc.
- Srinidhi, R. (2012). Implementación del procesador MIPS.
- Hazlett, CD (2020). Un simulador de rutas de datos MIPS para mejorar el aprendizaje visual de la arquitectura informática (tesis doctoral, Universidad de Illinois en Urbana-Champaign).
- https://semianalysis.com/2022/04/12/tenstorrent-blackhole-grendel-and/
- https://tenstorrent.com/vision/ljubisa-and-jasmina
- https://www.tomshardware.com/news/tenstorrent-licenses-risc-v