Universidad de Guadalajara

Centro Universitario de Ciencias Exactas e Ingenierías.

Módulo de aprendizaje: Arquitectura de Computadoras

Número y nombre de la actividad: 06. [Chocorrol] Mem + ALU + Mem

Equipo: Intel2

Integrantes:

Arias Silva Rubén

Sánchez López Jorge Alberto

Fecha de entrega: 21 de octubre de 2024

Contenido

INTRODUCCIÓN

Un microprocesador está compuesto por varias unidades, diseñada cada una de ellas para realizar un trabajo específico. Las unidades específicas, junto con su diseño y organización se denominan arquitectura de la computadora.

La arquitectura determina el conjunto de instrucciones y el procedimiento que se sigue para ejecutar esas instrucciones. Cuatro unidades básicas que son comunes a todos los microprocesadores son la unidad aritmeticológica (ALU, *Arithmetic Logic Unit*), el decodificador de instrucciones, la matriz de registros y la unidad de control, como se muestra en la Figura 1. [1]

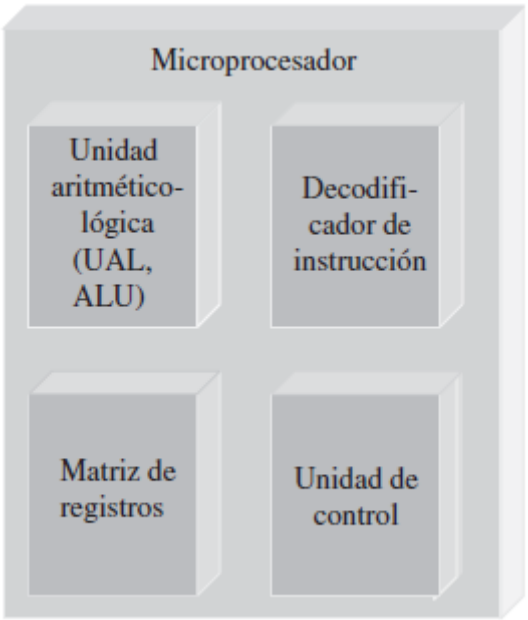


Figura 1, las diferentes unidades de un microprocesador.

Unidad aritmeticológica. La ***ALU*** es el elemento de procesamiento clave del microprocesador. Realiza, dirigida por la unidad de control, operaciones aritméticas (suma, resta, multiplicación y división) y operaciones lógicas (NOT, AD, OR y OR-

exclusiva), así como muchos otros tipos de operaciones. Los datos con los que trabaja la ALU se obtienen de la matriz de registros.

Procesador del Nintendo switch:

El procesador de la Nintendo Switch es un **NVIDIA Tegra X1**, basado en la arquitectura ARM Cortex-A57, junto con un GPU Maxwell de NVIDIA. Este SoC (System on Chip) incluye una CPU de 4 núcleos ARM Cortex-A57 y 4 núcleos ARM Cortex-A53, lo que le permite realizar tareas complejas con un buen balance entre rendimiento y eficiencia energética. El Tegra X1 también incorpora una GPU Maxwell con 256 núcleos CUDA, optimizada para proporcionar gráficos de alta calidad a una velocidad adecuada para un dispositivo portátil.

La Switch está diseñada para funcionar tanto en modo portátil como en una configuración de consola doméstica, ajustando automáticamente la potencia de su procesador según el modo de uso. Este procesador permite manejar una resolución de hasta 1080p en modo dock y 720p en modo portátil [2]

En Verilog, asignación por bloqueo y no bloqueo.

En Verilog, existen dos tipos de asignación dentro de los bloques always: bloqueante (blocking) y no bloqueante (non-blocking). Cada una afecta a la ejecuta del código durante la simulación y síntesis.

Asignación bloqueante "=". Se evalúa y asigna inmediatamente el valor de la expresión del lado derecho a la variable del lado izquierdo, bloqueando la ejecución de las siguientes instrucciones hasta que se complete. Este tipo de asignación es adecuado para lógica combinacional, ya que garantiza que las operaciones se ejecuten secuencialmente. Por ejemplo:

$$a = b + c;$$
$$d = a + e;$$

Aquí, la segunda línea no se ejecuta hasta que se complete la primera, lo que genera una ejecución estrictamente secuencial dentro del bloque.

Asignación no bloqueante "<=". La evALUación de la expresión se realiza inmediatamente, pero la asignación a la variable ocurre al final del ciclo de simulación. Esto permite que múltiples asignaciones ocurran de manera "simultánea" desde el punto de vista del tiempo de simulación, siendo adecuada para la lógica secuencial y registros. Un ejemplo sería:

$$always\ @(posedge\ clk)\ begin$$
$$a <= b;$$
$$b <= c;$$
$$end$$

En este caso, las dos asignaciones ocurren al final del ciclo del reloj, lo que permite que a y b reciban sus nuevos valores sin depender del orden en que se escriben.

Es importante no mezclar estos dos tipos de asignaciones en el mismo bloque always, ya que puede causar comportamientos indeterminados y problemas de simulación. [3]

Complemento a 1 (C1):

En este formato, el bit más significativo (MSB) indica el signo. Los números negativos se obtienen invirtiendo los bits del número positivo correspondiente.

Complemento a 2 (C2):

Similar a C1, pero los números negativos se generan invirtiendo los bits y sumando 1 al resultado. C2 es ampliamente utilizado debido a su eficiencia en operaciones aritméticas.

Punto flotante:

Utilizado para representar números muy grandes o pequeños, su estructura sigue el estándar IEEE 754, con un bit de signo, un campo de exponente sesgado y una mantisa normalizada. Los números se interpretan $(-\mathbf{1})^{\boldsymbol{S}} - \boldsymbol{M} - \mathbf{2}^{\boldsymbol{E}}$, donde $\boldsymbol{S}$ es el bit de signo, $\boldsymbol{M}$ es la mantisa, y $\boldsymbol{E}$ es el exponente ajustado. [4]

Contenido

OBJETIVOS

DESARROLLO

Decodificador.py

El desarrollo del codificador se llevó a cabo de la siguiente manera: inicialmente, nos apoyamos en una demostración de Tkinter proporcionada por el profesor y su ayudante. A partir de la base de ese código, realizamos los ajustes necesarios para asegurar su correcto funcionamiento. A continuación, detallamos las librerías utilizadas.

```
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: "**suma $4 $5 $0**", el botón "**Save result**", a diferencia de "**Save operation**", este botón hace que se guarden solamente los resultados de las operaciones, ejemplo: "**01001010100000000100**", el botón **"Save Both"**, guarda tanto la operación como el resultado, ejemplo: "**suma $4 $5 $0 = 01001010100000000100**" 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 "01_010" y "**resta**" a "01_110". 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 cunado se asocia la palabra clave "**leer**" solo se buscará un símbolo "**$**", Posteriormente, se construye una cadena binaria que combina el código asociado con la palabra clave y los números extraídos de la siguiente manera: según la palabra asociada como "**suma**" "01_010", se extraen los dos primeros bits, después se inserta el segundo numero que se encuentra después de un "**$**" en 5 bits, ahora si se ingresan los otros 3 bits de la palabra asociada, después se inserta el último número adyacente de un "**$**", en 5 bit y por ultimo se inserta el primer numero adyacente de un "**$**".

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 = {
        "suma": "01_010",
        "resta": "01_110",
        "and": "01_000",
        "menorQ": "01_111",
        "leer": "00_000"
    }

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

    for line_num, line in enumerate(lines):
        if not line.strip():  # Ignorar líneas vacías
```

```
            continue

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

                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 keyword == "leer":
                                count_dollar = 3
                                num1 = number_after_dollar
                                num2 = "0"
                                num3 = "0"
                            elif 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[:2]}{self.convertBinary(int(num2))}
                            {code[-
                            3:]}{self.convertBinary(int(num3))}{self.co
                            nvertBinary(int(num1))}\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):
    if isinstance(number, int):
        binary = str(bin(number)[2:]).zfill(5)
        return binary

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

Chocorol.v

Para llevar a cabo el módulo chocorol y realizar todas sus especificaciones optamos por crear varios módulos de manera manual los módulos que creamos son: AND, OR, ADD, SUB, SOLT (Set On Less Than) y NOR.

```
`timescale 1ps/1ps
// And ////////////////////////////////
module AND (
    input [31:0] A_AND,
    input [31:0] B_AND,
    output [31:0] C_AND
);
assign C_AND = A_AND & B_AND;
endmodule

// Or ////////////////////////////////
module OR (
    input [31:0] A_OR,
    input [31:0] B_OR,
    output [31:0] C_OR
);
assign C_OR = A_OR | B_OR;
endmodule

// 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

// Subtract ////////////////////////////////
module SUB (
    input [31:0] A_SUB,
    input [31:0] B_SUB,
    output [31:0] C_SUB
```

```
);
assign C_SUB = A_SUB - B_SUB;
endmodule

// Set On Less Than  ////////////////////
module SOLT (
    input [31:0] A_SOLT,
    input [31:0] B_SOLT,
    output [31:0] C_SOLT
);
assign C_SOLT = (A_SOLT < B_SOLT) ? 32'd1 : 32'd0;
endmodule

// Nor ////////////////////////////
module NOR (
    input [31:0] A_NOR,
    input [31:0] B_NOR,
    output [31:0] C_NOR
);
assign C_NOR = ~(A_NOR | B_NOR);
endmodule
```

Lo siguiente fu generar la primera memoria la cual contiene todos los operandos El módulo "**Mem_operandos**" es un componente de hardware que facilita el acceso a una memoria de 64 ubicaciones, cada una con un tamaño de 32 bits. Este módulo está diseñado para leer operandos a partir de dos direcciones de entrada, "**dirLec1**" y "**dirLec2**", que son especificadas con 5 bits. El módulo recibe tres entradas: dos direcciones "**dirLec1**" y "**dirLec2**", una señal de escritura "**we**", aunque la escritura no se implementa en esta versión. Las salidas son "**salida1**" y "**salida2**", que contienen los datos leídos desde las posiciones de memoria indicadas. La memoria se define como un arreglo de 64 elementos de 32 bits y se inicializa mediante "**$readmemh**" al comienzo de la simulación, cargando datos desde un archivo externo llamado "**Datos.txt**". Los datos que contiene el archivo están en la siguiente tabla (están en hexadecimal):

Posición	Valor
0	56B
1	A3B4
2	1AA
3	2A3
4	32
5	3C
6	55555555
7	AAAAAAAA

Un bloque "**always @(*)**" actualiza las salidas con los valores de memoria correspondientes a las direcciones de lectura. Esto permite que cualquier cambio

en las direcciones de entrada resulte en una lectura dinámica y rápida de los operandos.

```
// Memoria operadores//////////////////////
module Mem_operandos (
    input[4:0] dirLec1,
    input[4:0] dirLec2,
    input we,
    output reg[31:0] salida1,
    output reg[31:0] salida2
);

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

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

always @(*) begin
    salida1 = mem[dirLec1];
    salida2 = mem[dirLec2];
end
endmodule
```

Se implemento la "**ALU**" de la siguiente manera: primero se declararon las entradas: "op1" de 32 bits, "op2" de 32 bits y "sel" de 3 bits y una salida "resultado" de 32 bits, la ALU va a recibir dos valores de "**Mem_operandos**", con los que se va a realizar las operaciones, y el resultado dependiendo de la operación, ira directo a la salida.
A continuación, se hace una instancia de los módulos para cada operación: AND, OR, ADD, SUB, SOLT (Set On Less Than) y NOR.
En un bloque "**always @(*)**" se hará un "**case**" para hacer el multiplexor.
Cuando en la siguiente tabla se puede apreciar las diferentes líneas de control de la ALU:

ALU control lines	Function
000	AND
001	OR
010	ADD
110	SUBTRACT
111	SET ON LESS THAT
100	NOR

Se creó otra memoria llamada "**Mem_resultados**" que almacena los resultados de la ALU, esta memoria es similar a la memoria "**Mem_operandos**" solamente con algunos cambios, se eliminó "**dirLec2**" y "**salida2**", y se añade una entrada de 32 bits para recibir el resultado de la ALU. En un bloque "**always @(*)**" se hace una evaluación condicional con la entrada "**we**" en donde si es 1 se van a escribir los

datos que la ALU le mande a la memoria en la dirección que tenga "**dirEsc**", pero si "**we**" es 0 entonces la memoria solamente va a leer lo que tenga la memoria en la posición "**dirLec1**" y se ira directo a la salida.

```
// Memoria resulatdos//////////////////////
module Mem_resultados (
    input[31:0] datos,
    input[4:0] dirEsc,
    input[4:0] dirLec1,
    input we,
    output reg[31:0] salida1
);

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

always @(*) begin
    if (we)
        mem[dirEsc] = datos;
    else
        salida1 = mem[dirLec1];
end
endmodule
```

El módulo "**chocorol**" es la combinación de todos los módulos anteriores, una memoria que contiene valores "**Mem_operandos**", una ALU que hace operaciones con dichos valores "**ALU**" y una segunda memoria que almacena el resultado de las operaciones hechas por la ALU "**Mem_resultados**".
Este módulo tiene una salida de 32 bits y una entrada de 20 bits, los cuales son repartidos de la siguiente manera: el primer bit va al "**we**" de "**Mem_operandos**", el segundo bit es para "**we**" de "**Mem_resultados**", del tercero al séptimo van a "**dirLec1**" de "**Mem_operandos**", del octavo al décimo van a "**sel**" de la "**ALU**", del onceavo al quinceavo van a "**dirLec2**" de "**Mem_operandos**" y, por último, del dieciseisavo al veinteavo van a "**dirLec1**" de "**Mem_resultados**". Además, hay conexiones entre los componentes: la "**salida1**" de "**Mem_operandos**" se conecta con "**op1**" de la "**ALU**", la "**salida2**" de "**Mem_operandos**" se conecta con "**op2**" de la "**ALU**", "**resultado**" de la "**ALU**" se conecta con "**datos**" de "**Mem_resultados**", y por último "**salida1**" se conecta con la salida de "**chocorol**".

```
// Chocorol//////////////////////
module Chocorol(
    input [19:0] in,
    output [31:0] sal
);

wire [31:0] C1;
wire [31:0] C2;
wire [31:0] C3;
```

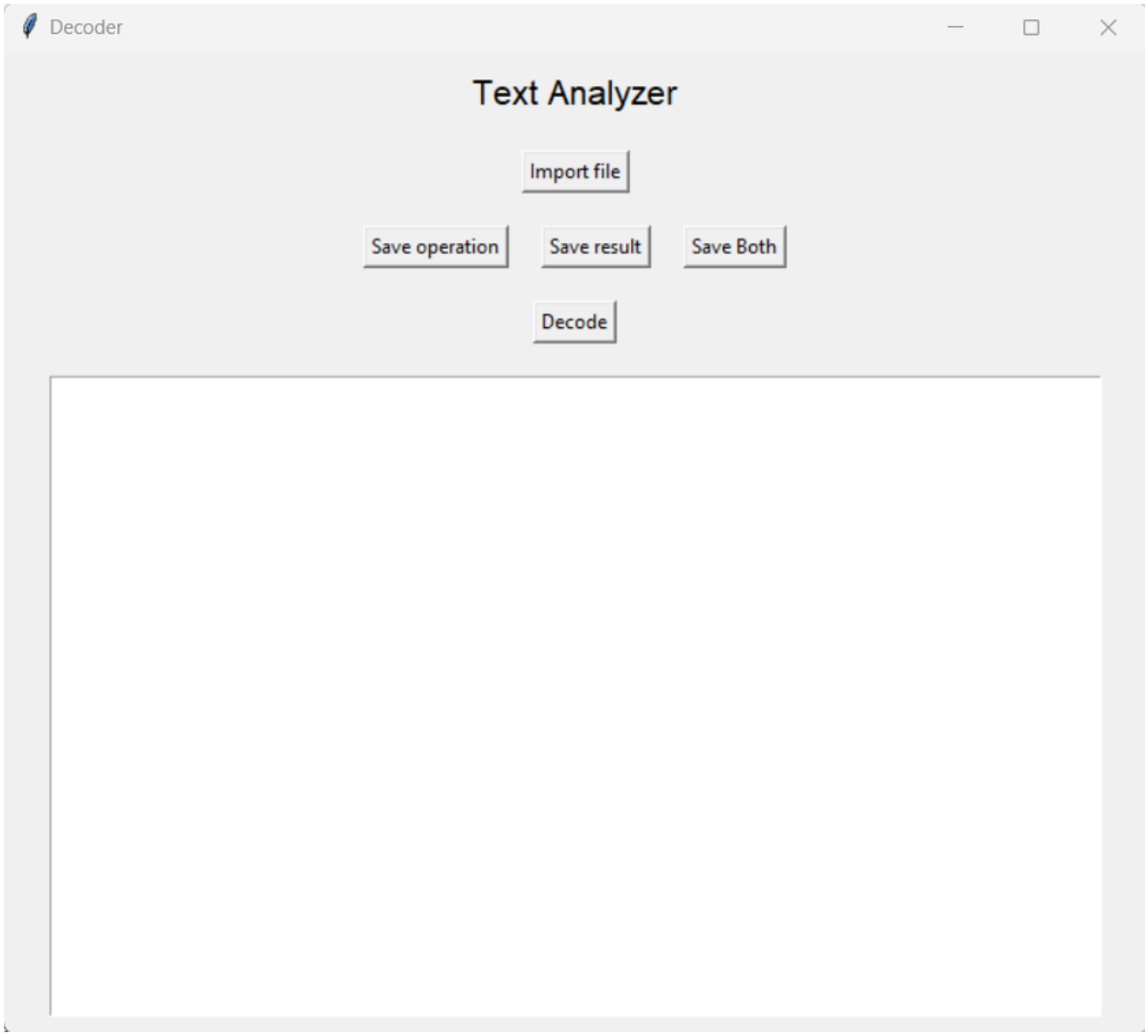
```
Mem_operandos I1 (.dirLec1(in[17:13]), .dirLec2(in[9:5]),
.we(in[19:19]), .salida1(C1), .salida2(C2));
alu I2 (.op1(C1), .op2(C2), .sel(in[12:10]), .resultado(C3));
Mem_resultados I3 (.datos(C3), .dirEsc(in[4:0]), .dirLec1(in[4:0]),
.we(in[18:18]), .salida1(sal));

endmodule
```

Por último, tenemos el TestBench "**TB_Chocorol**" que tiene las mismas entradas que "**chocorrol**" e internamente tiene una instancia de "**chocorrol**", este es para hacer pruebas del modulo que se quiere probar, para ello se tenemos que ingresarle datos para comprobar el funcionamiento del módulo, en este caso podemos cargar los valores desde un archivo de texto que llamamos "**instrucciones.txt**" gracias a "**$readmemb**".

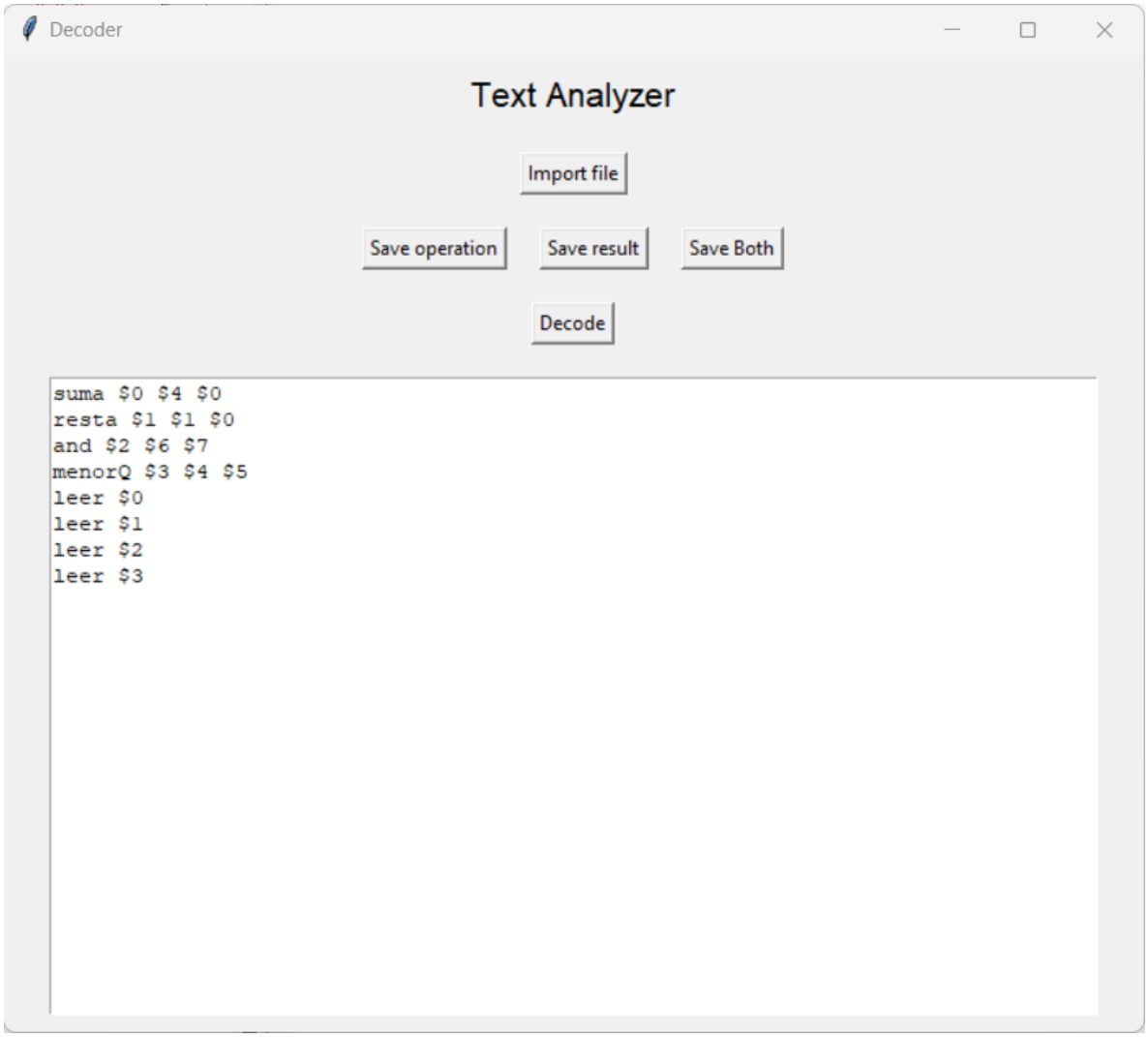
```
// TB_Chocorol//////////////////////////
module TB_Chocorol;
reg [19:0] TB_in;
wire [31:0] TB_out;

reg [19:0] mem[0:20];
integer i;

Chocorol I1(.in(TB_in), .sal(TB_out));

initial begin
    $readmemb("instrucciones.txt", mem);
    for (i = 0; i < 20; i = i + 1) begin
        TB_in = mem[i];
        #10;
    end
end

endmodule
```

Resultados

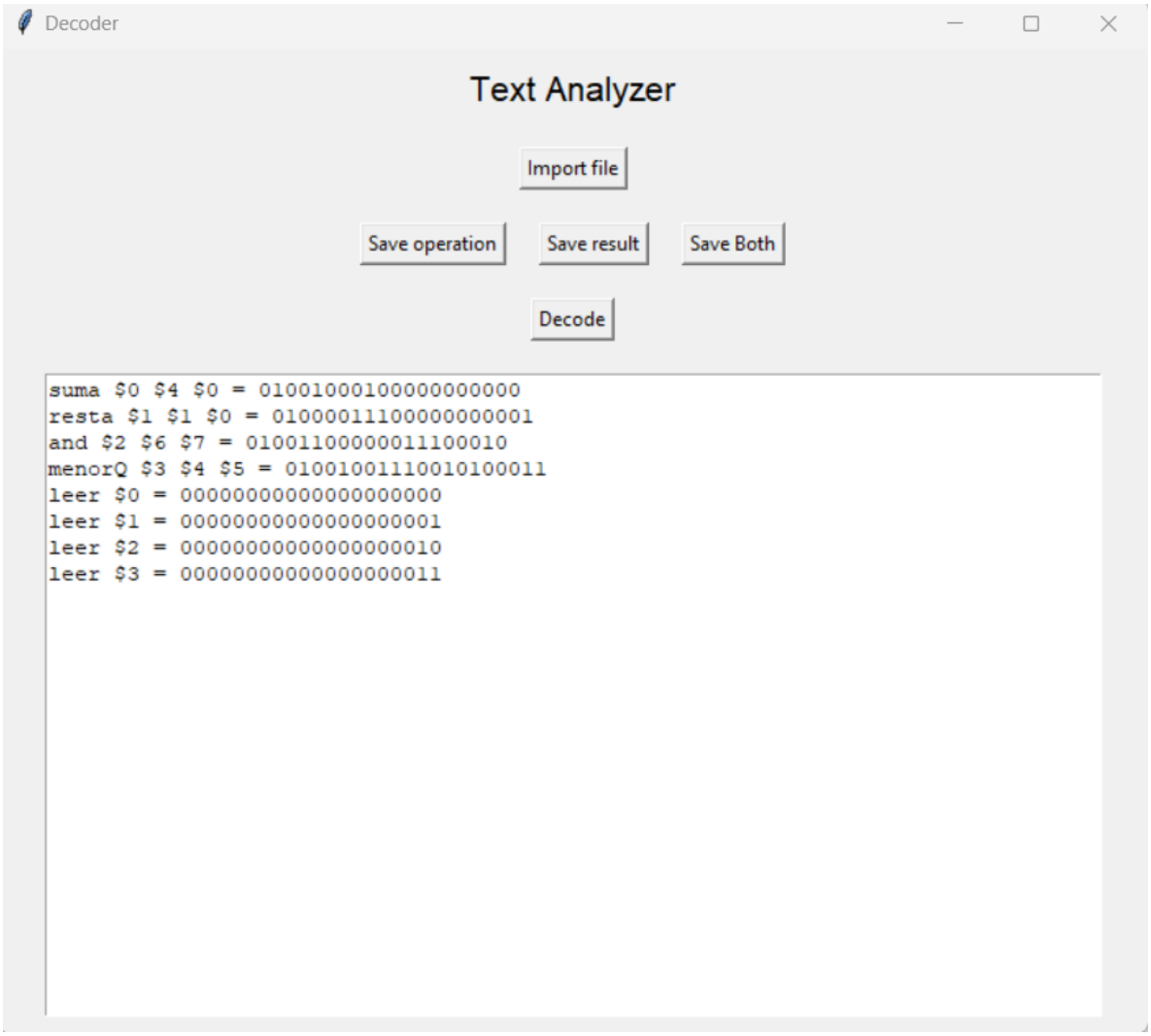
para comprobar los resultados, primero utilizamos el programa "**Decodificador.py**"



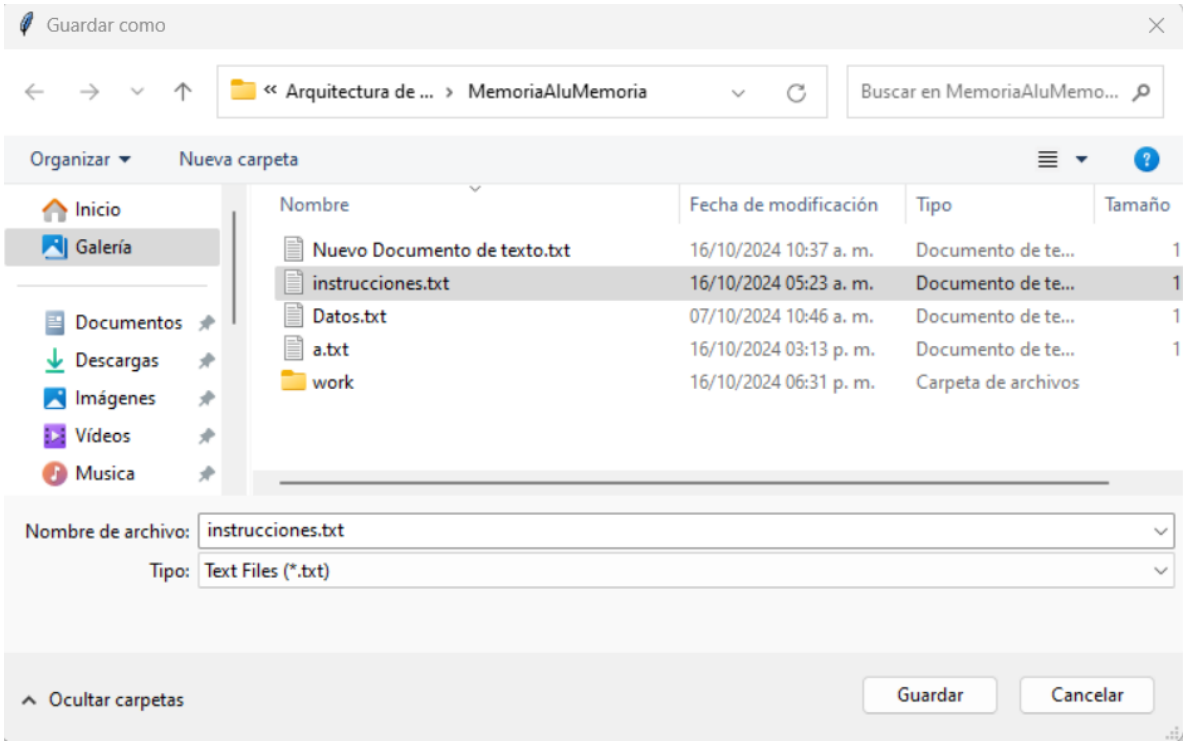
Después ingresamos 8 operaciones, 4 para probar operaciones y 4 para leer los resultados:



Y damos clic en “**Decode**” para decodificar las operaciones:



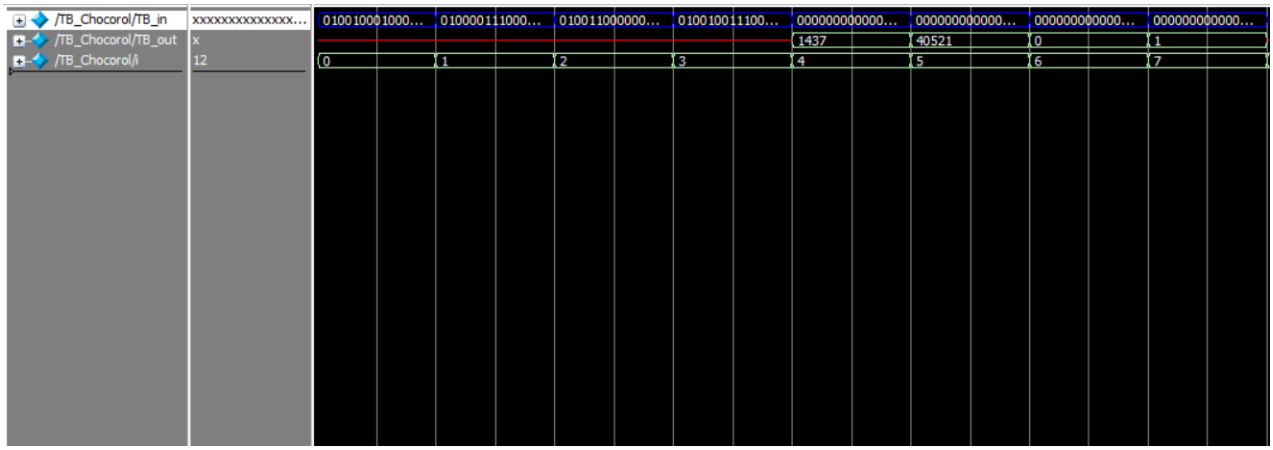
Una vez que ya se obtuvieron los resultados, guardamos dichos resultados dando clic en "**Save result**", Nos pide que le demos nombre al archivo y que le demos una ubicación para guardar, o bien podemos sobreescribir un archivo existente:



Eso es todo lo que tenemos que hacer con "**Decodificador.py**", ahora probamos "**Chocorol.v**", los resultados que se esperan son los siguientes:

- $50 + 1387 = 1{,}437$, guardar "**Mem_resultados**" posición 0
- $41908 - 1387 = 40521$, guardar "**Mem_resultados**" posición 1
- $55555555_{16}\ and\ AAAAAAAA_{16} = 0$, guardar "**Mem_resultados**" posición 2
- $32 < 3C = 1$, guardar "**Mem_resultados**" posición 3
- leer "**Mem_resultados**" posición 0
- leer "**Mem_resultados**" posición 1
- leer "**Mem_resultados**" posición 2
- leer "**Mem_resultados**" posición 3

Como se puede apreciar en la siguiente imagen, los resultados fueron correctos:



CONCLUSIÓN

Un microprocesador está compuesto por varias unidades especializadas, como la ALU, el decodificador de instrucciones, la matriz de registros y la unidad de control, que trabajan en conjunto para ejecutar instrucciones.

En este caso implementamos una ALU (Unidad Aritmeticológica) con dos memorias una para obtener los valores con los que la ALU va a trabajar y otra para almacenar los resultados de las operaciones, al realizar la implementación de una ALU se nos permitió entender como interactúan los componentes internamente, con el fin de que realicen operaciones aritméticas y lógicas, utilizando datos almacenados en una memoria interna, también fue muy interesante la implementación de las memorias, al poder crear las memorias, pudimos comprender como funcionan las memorias, guardan señales eléctricas las cuales las representamos como 0 o 1, y podemos acceder a conjuntos de bits mediante un índice para leer lo que tiene o para escribir o sobrescribir valores.

En cuanto a la creación de un programa para decodificar las instrucciones fue un gran aprendizaje sobre todo porque utilizamos una interfaz gráfica, en general esta actividad nos sirvió mucho para acercarnos más a comprender tanto el software como el hardware de una computadora.

REFERENCIAS

[1] L. Floyd, T. (2006) Fundamentos de sistemas digitales (9.ª ed., p. 785). Madrid: Miguel Martín Romo.

[2] Limelight Networks. (2019). *The State of Online Gaming – 2019*. Limelight. Retrieved from https://www.limelight.com/resources/white-paper/state-of-online-gaming-2019/

[3] CSAIL, Massachusetts Institute of Technology. (2007). *Blocking vs. Nonblocking Assignments in Verilog*. Retrieved from [https://courses.csail.mit.edu](https://courses.csail.mit.edu/6.111/f2007/handouts/L06.pdf).

[4] Stallings, W. (2017). Computer Organization and Architecture: Designing for Performance. Pearson Education.