
Arquitectura MIPS

Fase 1



24 ABRIL

Agredano Daniel
Gomez Alexia
Marquez Cristian
Román Carlos



Conejos MIPS

Introducción

En el siguiente documento se plasma el inicio de la documentación del proyecto final. Este proyecto se centra principalmente en la construcción de un procesador iniciando en la construcción del “Single Datapath” quien será el encargado de que pueda implementar/decodificar instrucciones básicas tipo R.

Objetivos

Crear un DataPath capaz de ejecutar las instrucciones ADD, SUB, AND, OR, y SLT integrando los módulos de BAM (ALU, Banco de Registros, Memoria de datos) y añadiendo una Unidad de control y una ALU control. Así como la creación de un programa en python para crear un archivo .txt con instrucciones en ensamblador, visualizar el contenido del archivo, realizar una conversión de instrucciones tipo R en ensamblador a máquina y crear un nuevo archivo.txt con las instrucciones en máquina

Objetivos particulares -Módulos Nuevos-

Unidad de Control (UC)

- Una Entrada de 6 bits donde recibe los bits del apartado del Código de Operación (Opcode).
- Tres salidas de 1 bit, MemToReg, RegWrite, y MemToWrite.
- Una salida de 3 bits, ALUOp que se conecta a la ALUControl.

Este módulo debe definir que señales mandar a los diferentes elementos del sistema dependiendo del OpCode que reciba.

Alu-Control

- Una entrada de 6 bits, que recibe los bits 0:5 de la instrucción.
- Una entrada de 3 bits, que llega de la UC.
- Una salida de 3 bits, que es enviada a la ALU para indicar qué operación debe realizar.

Recibe datos de la UC y los reconvierte para que la ALU sepa qué instrucción realizar

Mux 2:1

-
- Dos entradas de 32 bits cada una.
 - Una entrada de 1 bit, para seleccionar si la entrada se “pasa” a la salida o no.
 - Una salida de 32 bits, donde se canaliza el dato seleccionado.

TestBench de DataPath Tipo-R (DPTR)

Debe hacer un testbench capaz de observar que sirva cada uno de nuevos módulos de forma correcta.

EL módulo que instancie a todos (DPTR), será probado enviando 10 instrucciones, dos de cada una de las instrucciones mencionadas en la introducción. ALU, SUB, OR, AND, SLT. Hay que hacer una tabla con las instrucciones en formato ensamblador y otra equivalente en código máquina, los datos de esta última tabla serán la entrada de su TB.

Investigar principios de desarrollo de GUI y manejo de archivos

Realizar una investigación sobre:

- Qué es la GUI
- Funcionamiento de librería TKINTER
- Los principios de desarrollo de GUI
- Formas de crear, abrir, leer y modificar archivos .txt

Prototipo de interfaz de usuario con Python

Mostrar una interfaz gráfica para:

- Cargar un archivo .txt con instrucciones en ensamblador
- Visualizar el contenido del archivo
- Realizar una conversión de instrucciones tipo R en ensamblador a máquina
- Crear un nuevo archivo.txt con las instrucciones en máquina

Investigación

En la arquitectura MIPS, las instrucciones tipo R son aquellas que realizan operaciones aritméticas y lógicas entre registros. Estas instrucciones tienen un formato común que incluye tres registros de propósito general y se ejecutan completamente en el hardware.

Tienen un formato común de 6 campos, donde se especifican los registros de origen, destino y la operación a realizar.

Algunos ejemplos de estas instrucciones son: add, sub, slt, sll, srl, etc.

Este tipo de instrucciones vienen dadas en un formato de 32 bits, “seccionados” de la siguiente manera para que pueda realizar las operaciones correctamente

- **Código de operación (opcode):** Identifica la operación a realizar. 6 bits
- **Registros de origen (rs, rt):** Identifican los registros fuente para la operación. 5 bits cada uno.
- **Registro de destino (rd):** Identifica el registro donde se almacenará el resultado de la operación. 5 bits.
- **Desplazamiento (shamt):** En instrucciones como desplazamiento lógico, este campo especifica la cantidad de bits a desplazar. 5 bits.
- **Código de función (funct):** Especifica la operación a realizar dentro de un conjunto más amplio de operaciones definidas por el opcode. 6 bits

En especial la instrucción slt (set less than) se utiliza para establecer el valor de un registro a 1 si el valor en un registro fuente es menor que el valor en otro registro fuente, y a 0 en caso contrario teniendo como sintaxis la siguiente:

slt \$rd, \$rs, \$rt

en donde \$rd es el registro donde se almacenará el resultado (1 si \$rs es menor que \$rt, 0 en caso contrario). \$rs es el primer operando y \$rt es el segundo operando.

Es importante destacar que la instrucción slt compara los valores considerando la interpretación de los registros como enteros con signo.

Por otra parte está la operación ternaria, que es una operación que se lleva a cabo en algunos lenguajes de programación y se denota típicamente con el símbolo “? : ”. Esta operación toma tres operandos y se utiliza para tomar decisiones simples en función de una condición booleana.

La estructura básica de la operación ternaria es la siguiente:

condición ? resultado si verdadero : resultado si falso

Este tipo de operación se utiliza principalmente de manera similar a estructuras If - Else.

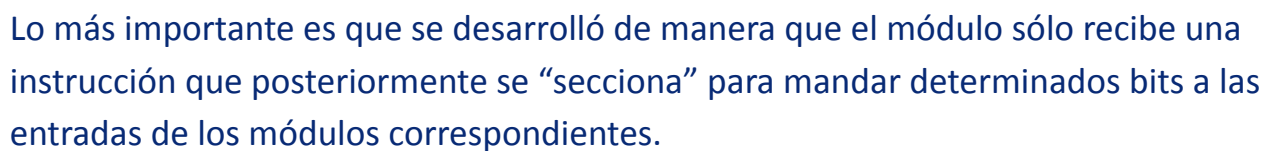
Otro punto a tratar en este documento es el proceso de compilación. El proceso de compilación es el conjunto de pasos que se lleva a cabo para traducir un programa escrito en un lenguaje de alto nivel (como C, C++, Java, etc.) a un lenguaje de bajo nivel (normalmente lenguaje máquina) que la computadora pueda entender y ejecutar directamente. Este proceso es realizado por un programa llamado compilador.

Este proceso cuenta con las siguientes etapas:

- **Análisis:**
 - **Análisis léxico (Scanner):** En esta etapa, el código fuente se divide en unidades léxicas o tokens, como palabras clave, identificadores, operadores, etc. Este proceso elimina los comentarios y los espacios en blanco que no son significativos para el programa.
 - **Análisis sintáctico (Parser):** Aquí, los tokens generados en la etapa de análisis léxico se organizan en una estructura jerárquica que sigue las reglas de la gramática del lenguaje de programación. Esto se conoce como árbol de sintaxis abstracta (AST).
 - **Análisis semántico:** En esta fase, se verifica si el código cumple con las reglas semánticas del lenguaje de programación. Esto incluye la comprobación de tipos, la detección de errores de declaración y uso de variables, entre otros.
- **Generación y optimización de código:** Una vez que el código ha sido analizado y validado, se genera un código intermedio que representa el programa de una manera más abstracta que el código fuente original. Este código intermedio es más fácil de manipular y optimizar que el código fuente para mejorar su eficiencia en tiempo de ejecución y/o consumo de recursos.

-
- **Generación de código objeto:** Finalmente, el código intermedio optimizado se traduce en código objeto, que es un código de máquina específico para la arquitectura de la máquina objetivo. Este código objeto puede estar en forma de código ensamblador o en código binario directamente ejecutable.
 - **Enlace (Linking):** Si el programa consta de varios archivos fuente o bibliotecas, es necesario enlazar o vincular estos archivos para formar un ejecutable completo. Durante este proceso, se resuelven las referencias a funciones y variables externas y se genera un único archivo ejecutable.

Utilizando el diagrama que se encuentra en la imagen se llevó a cabo la creación de un módulo que incorpora los módulos de BAM y los nuevos módulos mencionados en los objetivos.



La manera en la que se envían los bits “seleccionados” es la mostrada en la tabla siguiente:

Set de bits	A dónde se envían	Función
31:26	Unidad de Control	Identificar el tipo de instrucción (R, J o I) y habilitar escritura en BR y MD
25:21	Banco de Registros	Es la dirección del primer Registro del BR
20:16	Banco de Registros	Es la dirección del segundo Registro del BR
15:11	Banco de Registros	Es la dirección donde se guardan datos en el Registro del BR
10:6	Banco de Registros	Shamt (no hace nada aún)
5:0	ALU control	Indica a la ALU qué operación realizar

Por otra parte las funciones de la Unidad de Control de ALU son:

Instrucción	Function Code	ALU Code
ADD	100000	0010
SUB	100010	0110
AND	100100	0000
OR	100101	0001
NOR	100111	1100
SLT	101010	0111

Verilog

Para comprobar que el código en verilog funciona realizamos la siguiente tabla con instrucciones en ensamblador y en máquina para mandarlo a la entrada del módulo.

Instrucción en ensamblador	instrucción en binario	Dato en \$RS	Dato en \$RT	Dato esperado en \$RD
add \$0 \$1 \$2	000000 00001 00010 00000 00000 100000	15	15	30
sub \$3 \$4 \$5	000000 00100 00101 00011 00000 100010	24	56	-32
slt \$6 \$7 \$8	000000 00111 01000 00110 00000 101010	89	2	0
and \$9 \$10 \$11	000000 01010 01011 01001 00000 100100	1001010 = 74	1010110 = 86	1000010 = 66
or \$12 \$13 \$14	000000 01101 01110 01100 00000 100101	0101011 = 43	1001010 = 74	1101011 = 107

El banco de registros fue precargado con los siguientes datos:

Registro	Dato en BIN	Dato en DEC	Dato en HEX
1	00000000 00000000 00000000 00001111	15	F
2	00000000 00000000 00000000 00001111	15	F
4	00000000 00000000 00000000 00011000	24	18
5	00000000 00000000 00000000 00111000	56	38
7	00000000 00000000 00000000 01011001	89	59
8	00000000 00000000 00000000 00000010	2	2
10	00000000 00000000 00000000 01001010	74	4A
11	00000000 00000000 00000000 01010110	86	56
13	00000000 00000000 00000000 00101011	43	2B
14	00000000 00000000 00000000 01001010	74	4A

Datos esperados en los registros de destino			
0	00000000 00000000 00000000 00011110	30	1E
3	11111111 11111111 11111111 11100000	-32	FFFFFFE0
6	00000000 00000000 00000000 00000000	0	0
9	00000000 00000000 00000000 01000010	66	42
12	00000000 00000000 00000000 01101011	107	6B

También se realizó una tabla por módulo para identificar los nombres del módulo y de las variables de entrada y salida (input, output y output reg) con sus respectivas longitudes de bits. Las tablas son las siguientes:

Banco		
Nombre	Tipo	Longitud
ra1	Input	05 bits
ra2	Input	05 bits
di	Input	32 bits
dir	Input	05 bits
reg_write	Input	01 bits
dr1	Output Reg	32 bits
dr2	Output Reg	32 bits

Multiplexor		
Nombre	Tipo	Longitud
a	Input	32 bits
b	Input	32 bits
sel	Input	01 bits
salida	Output Reg	32 bits

ALU Control		
Nombre	Tipo	Longitud
funct	Input	06 bits
sel	Input	03 bits
AluOp	Output Reg	04 bits

Unidad de Control		
Nombre	Tipo	Longitud
op	Input	06 bits
MemToReg	Output Reg	01 bits
MemToWrite	Output Reg	01 bits
AluOp	Output Reg	03 bits
RegWrite	Output Reg	01 bits

Todo		
Nombre	Tipo	Longitud
InstruccionTR	Input	32 bits
Tr_zf	Output	01 bits

ALU		
Nombre	Tipo	Longitud
op1	Input	32 bits
op2	Input	32 bits
sel	Input	04 bits
res	Output Reg	32 bits
zf	Output Reg	01 bits

Memoria		
Nombre	Tipo	Longitud
dato	Input	32 bits
direccion	Input	32 bits
sel	Input	01 bits
salida	Output Reg	32 bits

Código:

/// ALU.v ///

```
module ALU(  
    input [31:0] op1,  
    input [31:0] op2,  
    input [3:0] sel,  
    output reg [31:0] res,  
    output reg zf  
);  
  
always @(*) begin  
    case (sel)  
        4'b0000: res = op1 & op2;  
        4'b0001: res = op1 | op2;  
        4'b0010: res = op1 + op2;  
        4'b0110: res = op1 - op2;  
        4'b0111: res = (op1 < op2) ? 1 : 0;  
        4'b1100: res = ~(op1 | op2);  
        default: res = 0;  
    endcase  
    zf = (res == 0) ? 1 : 0;  
end  
endmodule
```

```
/// AluControl.v ///
```

```
module AluControl(  
    input [5:0]funct,  
    input [2:0]sel,  
  
    output reg [3:0]AluOp  
);  
  
always @(*)  
begin  
    if(sel==3'b000)begin  
        case (funct)  
            6'b100000:  
                AluOp = 4'b0010;  
            6'b100010:  
                AluOp = 4'b0110;  
            6'b100100:  
                AluOp = 4'b0000;  
            6'b100101:  
                AluOp = 4'b0001;  
            6'b100111:  
                AluOp = 4'b1100;  
            6'b101010:  
                AluOp = 4'b0111;  
        endcase  
    end  
end  
endmodule
```

/// Banco.v ///

```
module Bancoreg(
    input [4:0]ra1,
    input [4:0]ra2,
    input [31:0]di,
    input [4:0]dir,
    input reg_write,

    output reg [31:0]dr1,
    output reg [31:0]dr2

);

reg [31:0] banco [0:31];

assign dr1 = banco[ra1];
assign dr2 = banco[ra2];

initial begin
    $readmemh("memoria.txt",banco);
end

always @(*) begin
    if(reg_write)
        banco[dir]=di;
end
endmodule
```

```
/// Memoria.v ///
```

```
`timescale 1ns/1ns

module Memoria(
    input [31:0]dato,
    input [31:0]direccion,
    input sel,

    output reg[31:0]salida
);

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

initial
begin
    $readmemb("data.txt", registro);
end

always @*
begin
    if(sel)
        registro[direccion]=dato;
    else
        salida=registro[direccion];
end
endmodule
```

```
/// Multiplexor.v ///
```

```
module Multiplexor(  
    input [31:0]a,  
    input [31:0]b,  
    input sel,  
    output reg [31:0]salida  
);  
  
always @(*)  
begin  
    if (sel == 0)  
        salida = a;  
    else  
        salida = b;  
end  
endmodule
```

```
/// UnidadControl.v ///
```

```
module UnidadControl(  
    input [5:0]op,  
  
    output reg MemToReg,  
    output reg MemToWrite,  
    output reg [2:0]AluOp,  
    output reg RegWrite  
);  
  
always @(*)  
begin  
    case (op)  
        5'b00000:begin  
            MemToReg    =1'b0 ;  
            MemToWrite  =1'b0 ;  
            AluOp       =3'b000 ;  
            RegWrite    =1'b1 ;  
        end  
    endcase  
end  
endmodule
```

/// Todo.v ///

```
module Todo(  
    input [31:0]instruccionTR,  
    output Tr_zf  
);  
  
reg [31:0]instruccion;  
assign instruccion=instruccionTR;  
  
wire [31:0]uno;  
wire [31:0]dos;  
wire [31:0]tres;  
wire [3:0]cuatro;  
wire [31:0]cinco;  
wire [31:0]seis;  
wire siete;  
wire ocho;  
wire nueve;  
wire [2:0]diez;  
  
Bancoreg ba(  
    .ra1(instruccion[25:21]),  
    .ra2(instruccion[20:16]),  
    .dir(instruccion[15:11]),  
    .di(seis),  
    .reg_write(siete),  
    .dr1(dos),  
    .dr2(uno)  
);  
  
ALU alu(  
    .op1(dos),  
    .op2(uno),  
    .sel(cuatro),  
    .zf(Tr_zf),  
    .res(tres)  
);  
  
Memoria mem(  
    .direccion(tres),
```

```
.dato(unos),
.sel(ocho),
.salida(cinco)
);

Multiplexor mux(
    .a(tres),
    .b(cinco),
    .sel(nueve),
    .salida(seis)
);

AluControl aluc(
    .funct(instruccion[5:0]),
    .sel(diez),
    .AluOp(cuatro)
);

UnidadControl uc(
    .op(instruccion[31:26]),
    .MemToReg(nueve),
    .MemToWrite(ocho),
    .AluOp(diez),
    .RegWrite(siete)
);
endmodule
```

```
/// Todo_TB.v ///
```

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

reg [31:0]instruccionTR;

wire once;

Todo t1(
    .Tr_zf(once) ,
    .instruccionTR(instruccionTR)
);

initial
    begin
        instruccionTR=32'b000000_00001_00010_00000_00000_100000;
        #100;

        instruccionTR=32'b000000_00100_00101_00011_00000_100010;
        #100;

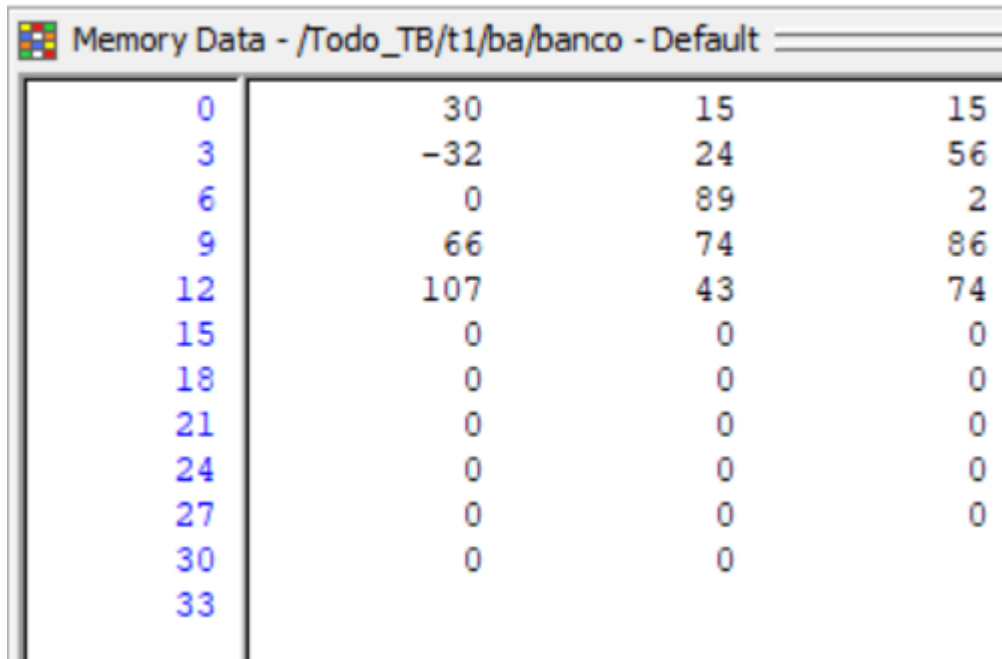
        instruccionTR=32'b000000_00111_01000_00110_00000_101010 ;
        #100;

        instruccionTR=32'b000000_01010_01011_01001_00000_100100;
        #100;

        instruccionTR=32'b000000_01101_01110_01100_00000_100101 ;
        #100;
        $stop;
    end
endmodule
```

Una vez completadas las tablas, el código y tener identificados los resultados esperados en el banco de registros, se llevó a cabo la simulación del módulo “Todo_TB” en el cual se esperaba que el banco de registros fuera modificado de acuerdo a las instrucciones plasmadas en las tablas previamente mostradas.

Estos fueron los resultados en el banco de registros:



The image shows a screenshot of a window titled "Memory Data - /Todo_TB/t1/ba/banco - Default". The window displays a table with four columns. The first column contains register addresses (0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33) in blue text. The second, third, and fourth columns contain numerical values representing the data stored in those registers. The values for registers 0 through 12 are non-zero, while registers 15 through 33 all contain the value 0.

Register Address	Value 1	Value 2	Value 3
0	30	15	15
3	-32	24	56
6	0	89	2
9	66	74	86
12	107	43	74
15	0	0	0
18	0	0	0
21	0	0	0
24	0	0	0
27	0	0	0
30	0	0	0
33	0	0	0

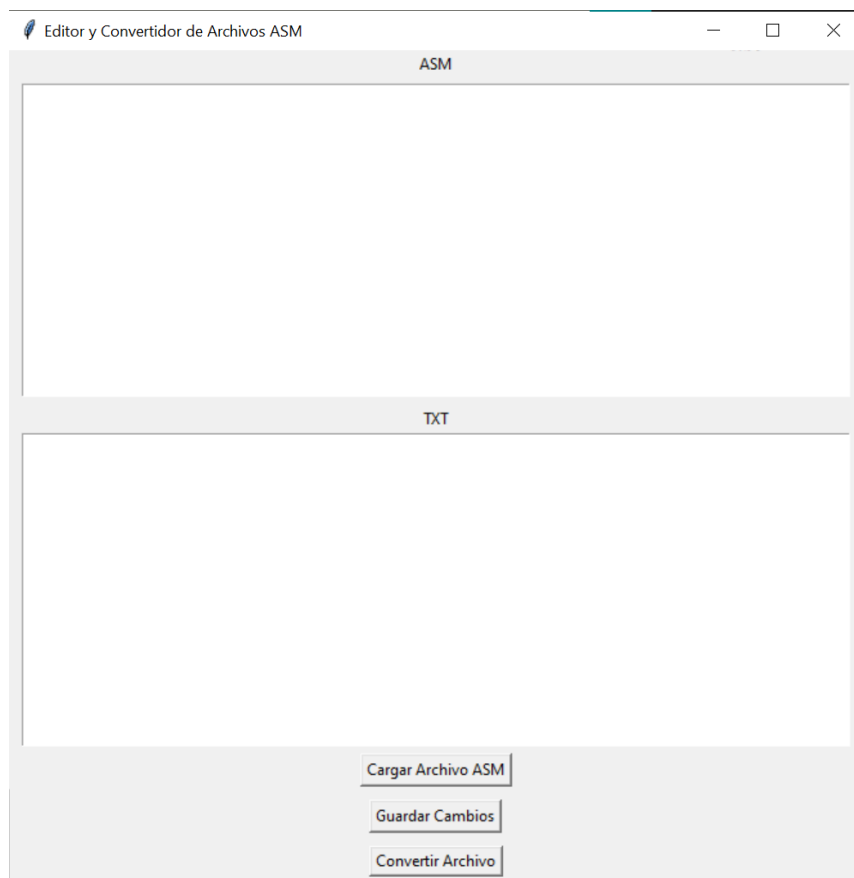
En esta imagen se muestra la primera columna con los resultados esperados o los registros que se indicaron como \$rd (registros destino), mientras que las columnas 2 y 3 son los registros de los operandos, lo que nos indica que las operaciones fueron ejecutadas correctamente y que el módulo “Todo” funciona correctamente.

Python

A partir de las siguientes líneas se muestra el código implementado en python.

Las funciones que realiza el código son las siguientes:

- Genera una ventana de interfaz gráfica donde muestra 2 cuadros de texto y 3 botones (Abrir archivo ASM, Guardar cambios y Convertir archivo).



- El botón de “Abrir archivo ASM” permite abrir el explorador de archivos para cargar la información de algún archivo con extensión “.asm” en el cuadro de texto superior.
- El botón de “Convertir archivo” se encarga de convertir el archivo con extensión “.asm” a código binario por medio del uso de diccionarios y funciones que detectan el tipo de instrucción y arman el código binario en base a esto, y Crea una copia en la misma dirección del archivo “.asm” con el código pero esta vez con la extensión “.txt”.
- El botón “Guardar Cambios” guarda en ambos archivos (tanto .txt como .asm) lo que se muestre escrito en los cuadros de texto de la interfaz respectivamente.

Código:

A continuación se muestra el código en python con las funciones para generar la GUI y crear las instrucciones en binario a partir de instrucciones en ensamblador.

```
import tkinter as tk

from tkinter import filedialog

import os

filepath = ""

instructions = { #Instrucciones codificadas {<instrucción> : [<tipo>,<func>]}

    "add"    : ["R","100000"],
    "sub"    : ["R","100010"],
    "and"    : ["R","100100"],
    "or"     : ["R","100101"],
    "nor"    : ["R","100111"],
    "slt"    : ["R","101010"]
}

def filepath_data(path):

    global filepath # Modifica la variable global

    filepath = path.removesuffix("asm") #Le quita la extensión

    return filepath #Retorna el nuevo valor por si lo necesita otra función

def cargar_archivo_asm():

    filepath = filedialog.askopenfilename(filetypes=[("Archivos ASM", "*.asm")])
    #Solicita un archivo al usuario

    if filepath: #Verifica si el usuario dió un archivo

        filepath = filepath_data(filepath) #Guarda la dirección sin extensión

        with open(filepath+"asm", "r") as file: #Abre el archivo de ensamblador

            contenido_text.delete("1.0", tk.END) # Limpia contenido anterior
```

```

        contenido_text.insert(tk.END, file.read()) # Muestra contenido en el
Text

        if(os.path.exists(filepath+"txt")): #Verifica que el txt exista

            with open(filepath+"txt", "r") as file: #Abre el txt

                bin_text.delete("1.0", tk.END) # Limpia contenido anterior

                bin_text.insert(tk.END, file.read()) # Muestra contenido en el
Text

def guardar_cambios():

    with open(filepath+"asm", "w") as asm_file: #Abre el archivo de ensamblador

        asm_file.write(contenido_text.get("1.0", tk.END).removesuffix("\n"))
#Sobreescribe el archivo

def IntStr_Bin(cadena, ancho):

    return (format(int(cadena), 'b')).zfill(ancho) #Convierte un entero dado como
cadena a un binario de longitud especificada

def instruccion_R(line): #Arma el formato de la instrucción

    ins = ""

    ins += "000000" #Op code

    ins += IntStr_Bin(line[2],5) #rs

    ins += IntStr_Bin(line[3],5) #rt

    ins += IntStr_Bin(line[1],5) #rd

    ins += "00000" #Shamt

    ins += instructions[line[0]][1] #Funct

    ins += "\n"

    return ins

def convertir_archivo():

    text = contenido_text.get("1.0", tk.END) #Lee el contenido del cuadro de texto

    text = text.replace("$", "") #Quita los '$'

```

```

words = [line.split(" ") for line in text.splitlines()] #Separa el texto en
lineas y palabras (arreglo bidimensional)

binary = ""

for line in words: #Revisa cada línea

    if line[0] in instructions: #Si la primera palabra corresponde a una
instrucción válida

        if instructions[line[0]][0]=="R": #Checa si es de tipo R (en el
diccionario viene especificado)

            binary += instruccion_R(line) #Arma la línea y la agrega al texto
resultado

binary = binary.removesuffix("\n") #Quita el último salto de línea

with open(filepath+"txt", "w") as txt_file: #Abre/crea el archivo txt

    txt_file.write(binary) #Escribe el texto resultado

    bin_text.delete("1.0", tk.END) # Limpia contenido anterior

    bin_text.insert(tk.END, binary) # Muestra contenido en el Text

root = tk.Tk() #Inicializa la ventana
root.title("Editor y Convertidor de Archivos ASM") #Le da un título a la ventana
#Etiqueta
label1 = tk.Label(root, text="ASM")
label1.pack()
#Cuadro de texto (ASM)
contenido_text = tk.Text(root, wrap=tk.WORD, height=15, width=80)
contenido_text.pack(padx=10, pady=5)

#Etiqueta
label2 = tk.Label(root, text="TXT")

```



```
label2.pack(padx=1, pady=1)

#Cuadro de texto (TXT)

bin_text = tk.Text(root, wrap=tk.WORD, height=15, width=80)
bin_text.pack()

#Botón para abrir archivo asm

cargar_asm_button = tk.Button(root, text="Cargar Archivo ASM",
command=cargar_archivo_asm)

cargar_asm_button.pack(pady=5)

#Botón para guardar cambios en el archivo asm

modificar_button = tk.Button(root, text="Guardar Cambios",
command=guardar_cambios)

modificar_button.pack(pady=5)

#Botón para convertir el asm a binario, guardar el archivo y mostrarlo en la
ventana

guardar_button = tk.Button(root, text="Convertir Archivo",
command=convertir_archivo)

guardar_button.pack(pady=5)

root.mainloop()
```

Bibliografía

Conditional (ternary) operator - JavaScript | MDN. (2023, September 6). MDN Web Docs.
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Conditional_Operator

MIPS Instruction Reference. (n.d.).
<https://phoenix.goucher.edu/~kelliher/f2009/cs220/mipsir.html>

CUADRO RESUMEN DEL LENGUAJE ENSAMBLADOR BÁSICO DEL MIPS-R2000. (s.f.).
<https://lorca.act.uji.es/asignatura/ig09/practicas/documentos/ensambladorR2000.pdf>

Compile - Glossary de MDN Web Docs: Definiciones de términos relacionados con la Web | MDN. (2023, November 13). MDN Web Docs.
<https://developer.mozilla.org/es/docs/Glossary/Compile>