CUCEI

CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

IL365 – Arquitectura de Computadoras

Reporte Python y Decodificador

Profesor: Jorge Ernesto López Arce Delgado

Ciclo: 2025 B **Horario:** lunes y miércoles de 9-11 Am.

NRC: 200204 **SEC**:D03

Fecha: 01/12/2025

Integrantes:

Jiménez Sánchez Antonio

Díaz López Iris Vianney

-Código .py

```
import tkinter as tk
from tkinter import filedialog, scrolledtext, messagebox

INSTR_R = {
    "ADD":  {"opcode": 0, "funct": 32},
    "SUB":  {"opcode": 0, "funct": 34},
    "AND":  {"opcode": 0, "funct": 36},
    "OR":   {"opcode": 0, "funct": 37},
    "SLT":  {"opcode": 0, "funct": 42},
}

INSTR_I = {
    "ADDI":  {"opcode": 8},
    "ANDI":  {"opcode": 12},
    "ORI":   {"opcode": 13},
    "XORI":  {"opcode": 14},
    "SLTI":  {"opcode": 10},
    "LW":    {"opcode": 35},
    "SW":    {"opcode": 43},
    "BEQ":   {"opcode": 4},
}

INSTR_J = {
    "J":     {"opcode": 2},
}

REG_MAP = {
    'zero': 0, 'at': 1, 'v0': 2, 'v1': 3, 'a0': 4, 'a1': 5, 'a2': 6, 'a3': 7,
    't0': 8, 't1': 9, 't2': 10, 't3': 11, 't4': 12, 't5': 13, 't6': 14, 't7': 15,
    's0': 16, 's1': 17, 's2': 18, 's3': 19, 's4': 20, 's5': 21, 's6': 22, 's7': 23,
    't8': 24, 't9': 25, 'k0': 26, 'k1': 27, 'gp': 28, 'sp': 29, 'fp': 30, 'ra': 31
}

def parse_register(r):
    r = r.strip()
    if r.startswith('$'):
        r = r[1:]
    if r in REG_MAP:
        return REG_MAP[r]
    elif r.isdigit():
        return int(r)
    raise ValueError(f"Registro inválido: {r}")

def parse_immediate(imm):
    imm = imm.strip()
    if imm.startswith('0x'):
        return int(imm, 16)
    return int(imm)
```

```
def parse_offset_base(operand):
    operand = operand.strip()
    if '(' in operand:
        offset, base = operand.split('(')
        base = base.rstrip(')')
        return parse_immediate(offset), parse_register(base)
    raise ValueError(f"Formato inválido para offset(base): {operand}")

def encode_r_type(mnemonic, parts):
    opcode = INSTR_R[mnemonic]["opcode"]
    funct = INSTR_R[mnemonic]["funct"]

    rd, rs, rt = parts[1], parts[2], parts[3]
    shamt = 0

    instr = (opcode << 26) | (parse_register(rs) << 21) | (parse_register(rt) << 16) | (parse_register(rd) <<
11) | (shamt << 6) | funct

    return instr.to_bytes(4, byteorder="big")

def encode_i_type(mnemonic, parts):
    opcode = INSTR_I[mnemonic]["opcode"]

    if mnemonic in ["LW", "SW"]:
        rt = parse_register(parts[1])
        offset, rs = parse_offset_base(parts[2])
        immediate = offset & 0xFFFF
    elif mnemonic in ["BEQ"]:
        rs = parse_register(parts[1])
        rt = parse_register(parts[2])
        immediate = parse_immediate(parts[3]) & 0xFFFF
    else:
        rt = parse_register(parts[1])
        rs = parse_register(parts[2])
        immediate = parse_immediate(parts[3]) & 0xFFFF

    instr = (opcode << 26) | (rs << 21) | (rt << 16) | immediate
    return instr.to_bytes(4, byteorder="big")

def encode_j_type(mnemonic, parts):
    opcode = INSTR_J[mnemonic]["opcode"]
    address = parse_immediate(parts[1]) & 0x3FFFFFF
    instr = (opcode << 26) | address
    return instr.to_bytes(4, byteorder="big")

def validate_program_requirements(lines):
    used_instructions = set()
    required_ops = {"BEQ", "J", "LW", "SW"}

    for line in lines:
        if not line.strip() or line.strip().startswith('#'):
            continue
```

```
        mnemonic = line.split()[0].upper()
        used_instructions.add(mnemonic)

    missing_ops = required_ops - used_instructions
    return missing_ops

def decode_line(line):
    line = line.split('#')[0].strip()
    line = line.replace(",", " ").strip()
    parts = line.split()

    if not parts:
        return None

    mnemonic = parts[0].upper()

    if mnemonic in INSTR_R:
        return encode_r_type(mnemonic, parts)
    elif mnemonic in INSTR_I:
        return encode_i_type(mnemonic, parts)
    elif mnemonic in INSTR_J:
        return encode_j_type(mnemonic, parts)
    else:
        raise ValueError(f"Instrucción no soportada: {mnemonic}")

def process_input(text):
    bin_output = bytearray()
    lines = text.strip().splitlines()

    missing_ops = validate_program_requirements(lines)
    if missing_ops:
        return None, f"Faltan instrucciones requeridas: {', '.join(missing_ops)}"

    for i, line in enumerate(lines):
        if not line.strip() or line.strip().startswith('#'):
            continue
        try:
            bin_instr = decode_line(line)
            if bin_instr:
                bin_output.extend(bin_instr)
        except Exception as e:
            return None, f"Error en línea {i+1}: {e}"
    return bin_output, "Decodificación exitosa. Programa cumple con requerimientos."

def bytes_to_binary_string(byte_data):
    binary_strings = []
    for byte in byte_data:
        binary_strings.append(format(byte, '08b'))
    return binary_strings

class MIPSDecoderApp(tk.Tk):
    def __init__(self):
        super().__init__()
```

```
        self.title("Decodificador MIPS - Arquitectura de Computadoras")
        self.geometry("850x500")

        title_label = tk.Label(self, text="Decodificador MIPS Pipeline - Proyecto Final",
                               font=("Arial", 14, "bold"))
        title_label.grid(row=0, column=0, columnspan=4, pady=10)

        tk.Label(self, text="Código Ensamblador MIPS:").grid(row=2, column=0, pady=5, sticky="w",
padx=10)
        self.text_entry = scrolledtext.ScrolledText(self, width=80, height=10)
        self.text_entry.grid(row=3, column=0, columnspan=4, padx=10, pady=4)

        button_frame = tk.Frame(self)
        button_frame.grid(row=4, column=0, columnspan=4, pady=10)

        self.btn_buscar = tk.Button(button_frame, text="Cargar Archivo", width=12, command=self.buscar)
        self.btn_buscar.pack(side="left", padx=2)

        self.btn_deco = tk.Button(button_frame, text="Decodificar", width=12, command=self.decode)
        self.btn_deco.pack(side="left", padx=2)

        self.btn_limpia = tk.Button(button_frame, text="Limpiar", width=12, command=self.limpia)
        self.btn_limpia.pack(side="left", padx=2)

        self.btn_crear_mips = tk.Button(button_frame, text="Crear MIPS", width=12,
command=self.crear_mips)
        self.btn_crear_mips.pack(side="left", padx=2)

        self.btn_expo_txt = tk.Button(button_frame, text="Exportar TXT", width=12,
command=self.exportar_txt)
        self.btn_expo_txt.pack(side="left", padx=2)

        self.btn_expo_asm = tk.Button(button_frame, text="Exportar ASM", width=12,
command=self.exportar_asm)
        self.btn_expo_asm.pack(side="left", padx=2)

        tk.Label(self, text="Binario Generado:").grid(row=5, column=0, pady=5, sticky="w", padx=10)
        self.text_salida = scrolledtext.ScrolledText(self, width=80, height=10, state="disabled")
        self.text_salida.grid(row=6, column=0, columnspan=4, padx=10, pady=4)

        self.status = tk.Label(self, text="Listo para decodificar", fg="green", font=("Arial", 10))
        self.status.grid(row=7, column=0, columnspan=4, pady=5)

    def buscar(self):
        filepath = filedialog.askopenfilename(
            title="Seleccionar archivo MIPS",
            filetypes=[("Text files", "*.txt"), ("ASM files", "*.asm"), ("All files", "*.*")]
        )
        if filepath:
            try:
                with open(filepath, "r", encoding="utf-8") as f:
                    content = f.read()
                    self.text_entry.delete("1.0", tk.END)
```

```
                self.text_entry.insert(tk.END, content)
            self.status.config(text="Archivo cargado exitosamente.", fg="blue")
        except Exception as e:
            self.status.config(text=f"Error al cargar archivo: {e}", fg="red")

    def decode(self):
        text = self.text_entry.get("1.0", tk.END)
        if not text.strip():
            self.status.config(text="Error: No hay código para decodificar.", fg="red")
            return

        bin_output, msg = process_input(text)
        self.text_salida.config(state="normal")
        self.text_salida.delete("1.0", tk.END)

        if bin_output:
            binary_strings = []
            for i in range(0, len(bin_output), 4):
                instruction_bytes = bin_output[i:i+4]
                binary_rep = " ".join(format(b, '08b') for b in instruction_bytes)
                binary_strings.append(binary_rep)

            for b in binary_strings:
                self.text_salida.insert(tk.END, b + "\n")
            self.status.config(text=msg, fg="green")
        else:
            self.status.config(text=msg, fg="red")
            messagebox.showerror("Error de Decodificación", msg)

        self.text_salida.config(state="disabled")

    def limpia(self):
        self.text_entry.delete("1.0", tk.END)
        self.text_salida.config(state="normal")
        self.text_salida.delete("1.0", tk.END)
        self.text_salida.config(state="disabled")
        self.status.config(text="Campos limpiados. Listo para nueva entrada.", fg="gray")

    def crear_mips(self):
        codigo = """ADDI $t0, $zero, 10
ADDI $t1, $zero, 10
SW $t0, 0($zero)
LW $t2, 0($zero)
BEQ $t0, $t1, 1
ADDI $s3, $zero, 99
J 0
ADD $t3, $t0, $t1"""

        ventana_formato = tk.Toplevel(self)
        ventana_formato.title("Seleccionar Formato")
        ventana_formato.geometry("300x150")
        ventana_formato.resizable(False, False)
```

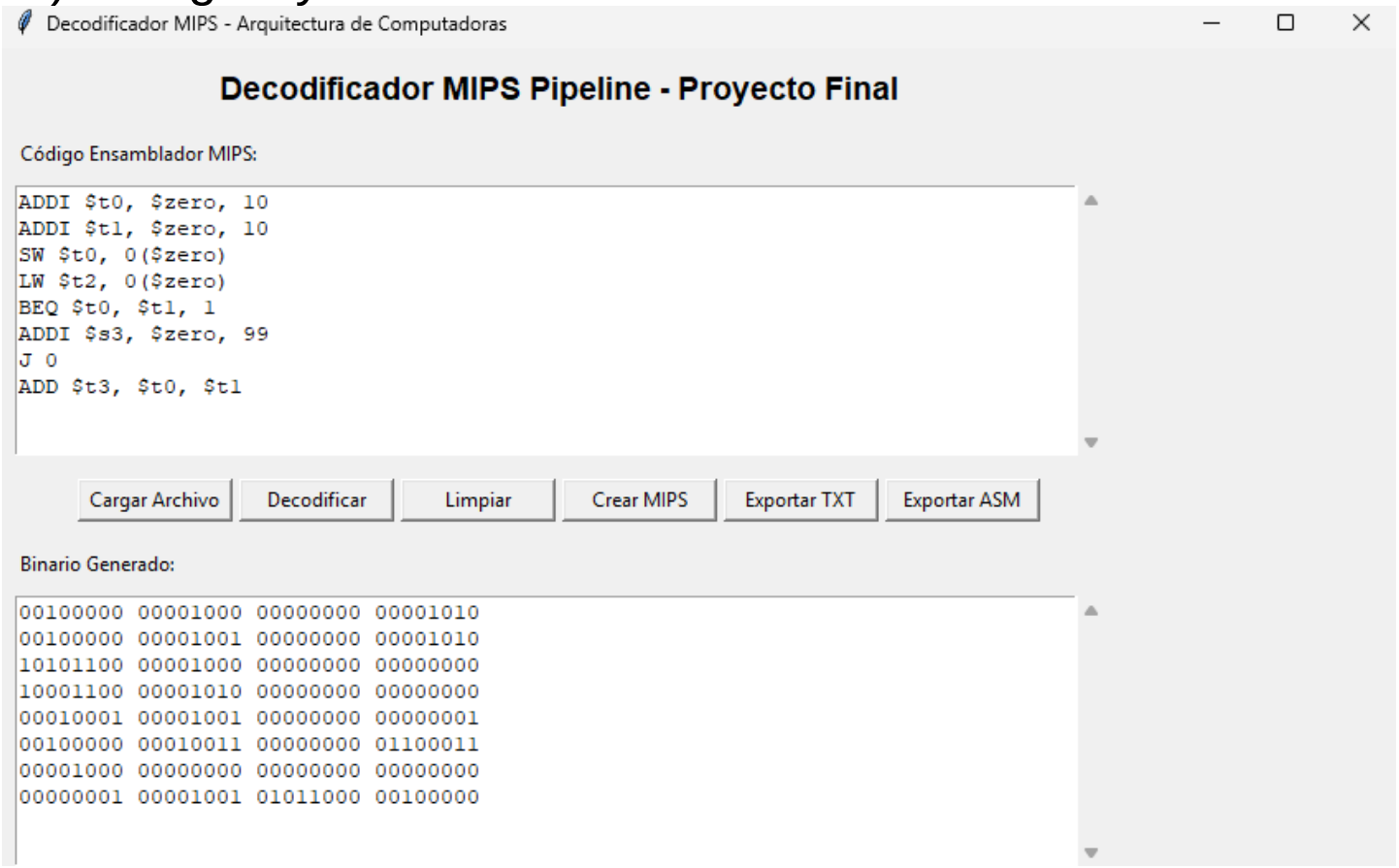
```
    tk.Label(ventana_formato, text="Seleccione el formato de exportación:",
             font=("Arial", 10)).pack(pady=10)

    frame_botones = tk.Frame(ventana_formato)
    frame_botones.pack(pady=20)

    formato_seleccionado = tk.StringVar(value="asm")

    def guardar_archivo():
        formato = formato_seleccionado.get()
        ventana_formato.destroy()

        if formato == "asm":
            archivo = filedialog.asksaveasfilename(
                title="Guardar archivo MIPS .asm",
                defaultextension=".asm",
                filetypes=[("ASM files", "*.asm"), ("All files", "*.*")]
            )
        else:
            archivo = filedialog.asksaveasfilename(
                title="Guardar archivo MIPS .txt",
                defaultextension=".txt",
                filetypes=[("Text files", "*.txt"), ("All files", "*.*")]
            )

        if archivo:
            try:
                with open(archivo, "w", encoding="utf-8") as f:
                    f.write(codigo)
                self.status.config(text=f"Archivo MIPS .{formato} creado: {archivo}", fg="blue")
                messagebox.showinfo("Éxito", f"Archivo MIPS .{formato} creado correctamente.")
            except Exception as e:
                self.status.config(text=f"Error al crear: {e}", fg="red")

    btn_asm = tk.Radiobutton(frame_botones, text="Archivo .asm", variable=formato_seleccionado,
                             value="asm", font=("Arial", 9))
    btn_asm.pack(side="left", padx=10)

    btn_txt = tk.Radiobutton(frame_botones, text="Archivo .txt", variable=formato_seleccionado,
                             value="txt", font=("Arial", 9))
    btn_txt.pack(side="left", padx=10)

    btn_guardar = tk.Button(ventana_formato, text="Guardar", width=15,
                            command=guardar_archivo, bg="#4CAF50", fg="white")
    btn_guardar.pack(pady=10)

    btn_cancelar = tk.Button(ventana_formato, text="Cancelar", width=15,
                             command=ventana_formato.destroy, bg="#f44336", fg="white")
    btn_cancelar.pack()

def exportar_txt(self):
    text = self.text_entry.get("1.0", tk.END)
    if not text.strip():
```

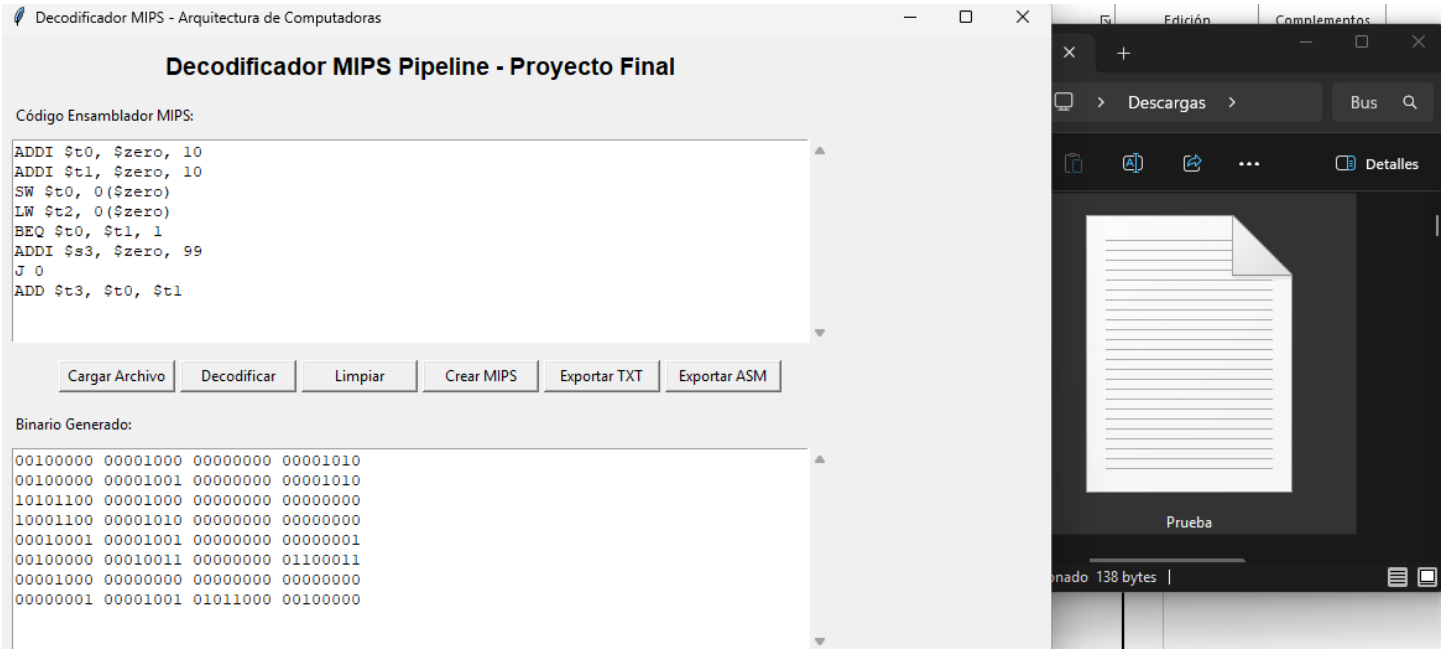
```
            self.status.config(text="Error: No hay código para exportar.", fg="red")
            return

        bin_output, msg = process_input(text)
        if bin_output:
            output_path = filedialog.asksaveasfilename(
                title="Guardar archivo binario TXT",
                defaultextension=".txt",
                filetypes=[("Text files", "*.txt"), ("All files", "*.*")]
            )
            if output_path:
                try:
                    with open(output_path, "w", encoding="utf-8") as f:
                        binary_strings = []
                        for i in range(0, len(bin_output), 4):
                            instruction_bytes = bin_output[i:i+4]
                            binary_rep = " ".join(format(b, '08b') for b in instruction_bytes)
                            binary_strings.append(binary_rep)

                        for b in binary_strings:
                            f.write(b + "\n")

                    self.status.config(text=f"Archivo TXT exportado: {output_path}", fg="green")
                    messagebox.showinfo("Éxito", "Archivo binario TXT creado.")
                except Exception as e:
                    self.status.config(text=f"Error al exportar: {e}", fg="red")
        else:
            self.status.config(text=msg, fg="red")
            messagebox.showerror("Error", msg)

    def exportar_asm(self):
        text = self.text_entry.get("1.0", tk.END)
        if not text.strip():
            self.status.config(text="Error: No hay código para exportar.", fg="red")
            return

        bin_output, msg = process_input(text)
        if bin_output:
            output_path = filedialog.asksaveasfilename(
                title="Guardar archivo binario ASM",
                defaultextension=".asm",
                filetypes=[("ASM files", "*.asm"), ("All files", "*.*")]
            )
            if output_path:
                try:
                    with open(output_path, "w", encoding="utf-8") as f:
                        f.write("# Archivo binario MIPS\n")
                        f.write("# Formato: 32 bits por instrucción\n\n")

                        for i in range(0, len(bin_output), 4):
                            instruction_bytes = bin_output[i:i+4]
                            instruction_int = int.from_bytes(instruction_bytes, byteorder="big")
                            binary_rep = format(instruction_int, '032b')
```

```
                    f.write(f"{binary_rep}\n")

                self.status.config(text=f"Archivo ASM exportado: {output_path}", fg="green")
                messagebox.showinfo("Éxito", "Archivo binario ASM creado.")
            except Exception as e:
                self.status.config(text=f"Error al exportar: {e}", fg="red")
        else:
            self.status.config(text=msg, fg="red")
            messagebox.showerror("Error", msg)

if __name__ == "__main__":
    app = MIPSDecoderApp()
    app.mainloop()
```

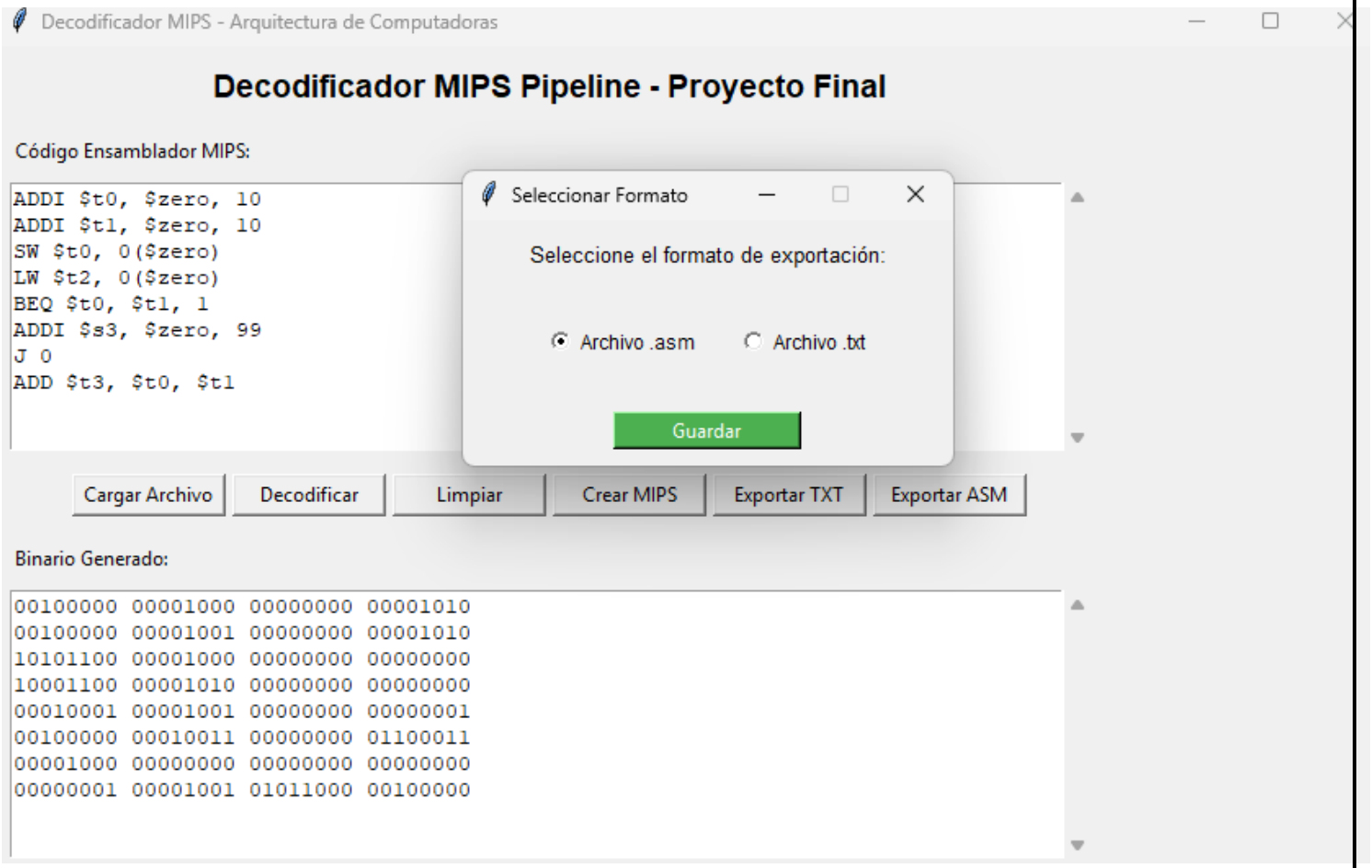
- Evidencia de funcionamiento.

A) Código Python del Decodificador





D) Programa en Ensamblador (.asm y .txt).
Botón Crear MIPS



- Texto Generado

00100000 00001000 00000000 00001010
00100000 00001001 00000000 00001010
10101100 00001000 00000000 00000000
10001100 00001010 00000000 00000000
00010001 00001001 00000000 00000001
00100000 00010011 00000000 01100011
00001000 00000000 00000000 00000000
00000001 00001001 01011000 00100000