UNIVERSIDAD DE GUADALAJARA
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

ARQUITECTURA DE COMPUTADORAS

A - Decodificador de instrucciones tipo MIPS

Alumno: Erick Manuel González Carrillo (219695611)
Profesor: Jorge Ernesto Lopez Arce Delgado

1 de noviembre de 2025

Índice

1. Introducción

El lenguaje de las computadoras suele se complejo para la comprensión del humano puesto que la única forma en que pueda entender nuestras instrucciones es por medio de códigos binarios, sin embargo si queremos realizar demasiadas instrucciones llega a ser bastante tardado y difícil de comprender.
Por eso es que los programadores construyeron un traductor que sea capaz de realizar las mismas instrucciones pero que sea mas claro de entender, entre ellas surgió el lenguaje ensamblador; este lenguaje estaba directamente escrito en la maquina, ademas que las instrucciones se pasaban directamente a binario. por ejemplo, podemos realizar una suma en ensamblador:

```
1        add A, B
```

El ensamblador traducirá esta notación a:

```
1        1000110010100000
```

Es por ello que para esta actividad nos enfocaremos en desarrollar una aplicación que sea capaz de simular un código ensamblador de la arquitectura MIPS.

Este decodificador recibirá las instrucciones en ensamblador MIPS, convirtiéndolas en código binario de 32 bits y que sean guardadas en la memoria de instrucciones.

2. Marco teórico

El desarrollo de lenguajes de alto nivel proporcionaron la orientación de diseñar una nueva forma de construir instrucciones debido a que las instrucciones CISC (Complex Instruction Set Computer) pueden ser bastantes complejas se diseño un nuevo repertorio de instrucciones reducidas RISC (Reduced Instruction Set Computer), este diseño hacia que las transferencias fueran mas optimizadas. Este repertorio de instrucciones han motivado estudios sobre maquinas RISC, optimizando el cauce de instrucciones estas maquinas también se prestan a segmentaciones mas eficientes porque hay menos instrucciones y son mas previsibles [2].

Uno de los primeros procesadores RISC que estuvieron disponibles al mercado fue el MIPS4000 desarrollado por MIPS Techonology Inc. que fue desarrollado en Stanford. Este diseño se divide en dos secciones, una contiene la CPU y la otra un coprocesador de gestion.
MIPS usa un tamaño fijo de 32 bits. Lo que simplifica la capacitación y la decodificación de instrucciones y la interacción de instrucciones y la gestión de memoria virtual. Esta arquitectura tiene tres formatos de instrucciones que comparten un formato común en los códigos de operación y referencias a registros simplificando su decodificación mostrados en las figuras 1, 2 y 3.
Todas las referencias a memoria constan de un desplazamiento de 16 bits y de un registro de 32 bits. Por ejemplo 'cargar palabra' tiene la forma:

```
1        lw r2, 128(r3)
```

Cualquiera de los 32 registros de uso general puede ser usado como registro base.

6	5	5	16
Operación	rs	rt	Inmediato

Tabla 1: Formato tipo I

6	26
Operación	Destino

Tabla 2: Formato tipo J

6	5	5	5	5	6
Operación	rs	rt	rd	Desp.	Función

Tabla 3: Formato tipo R

Las instrucciones de tipo R tienen un formato de instrucción en el que los primeros 6 bits son para el código de operación y estos estarán en 0 que están ubicados desde el bit 31:26, la siguiente instrucción contendrá la dirección del primer registro a operar y esta ubicado desde el bit 25:21, el registro rt es de la segunda dirección del registro que sera operado ubicado del bit 20:16, el siguiente registro es la dirección en donde se guardara el dato ubicados del registro 15:11, los otros 5 bits son de desplazamiento, sin embargo para este diseño no sera usado por lo que estarán en 0 y el ultimo es el de la función que realiza que son del bit 5:0 [1]. Su representacion esta mostrada en la figura 1.

Figura 1: Posiciones de una instrucción de tipo R

Un ejemplo de una instruccion de tipo R es:

```
1         ADD $8, $17, $18;
```

Si realizamos su conversion en los campos de la instruccion quedarian de la forma:

000000	10001	10010	01000	00000	10000

Así pues, para esta actividad se uso el lenguaje de programación Python. Esta herramienta es fácil de comprender y programar ayudando a realizar programas que en otros lenguajes se requieren estructuras bastantes complejas.
La aplicación esta desarrollada en un entorno gráfico (GUI) para eso se uso la librería TKinter. Con esta herramienta se pueden realizar aplicaciones GUI sencillas pero funcionales.
El punto principal de esta aplicación es que reciba una instrucción ensamblador y que este sea traducido en código binario; en una parte estará la instrucción en forma de 32 bits y el otro cuadro estará la misma instrucción en bytes. Ademas de que cuenta con la capacidad de guardar la instrucción en un archivo para que la memoria de datos leerá el código y ser ejecutado.

3. Desarrollo

La aplicacion esta desarrollada en Tkinter facilitando su programacion en entornos GUI, esta aplicacion tiene un tamaño de 700x600 pixeles.
Definimos las operaciones artimeticas que realza nuestro diseño en una estructura de Python llamada libreria facilitando haciendo que la traduccion sea directa y esta mostrada en el listing 3.

```
        instruccion_logica_aritmetica = {
            '100000': 'ADD', '100100': 'AND',
            '100111': 'NOR', '100101': 'OR', '101010': 'SLT'
        }
```

Estas son todas las instrucciones con las que cuenta la arquitectura MIPS lógicas y aritméticas. En el listing 3 definimos las funciones con las que realizamos las extracciones del codigo ensamblador MIPS ademas de limpiar los datos para que despues sean convertidos a valores biarios. Ademas existe una funcion en la que identifica que tipo de instruccion es, esto hara que el codigo sea escalable y en futuras mejoras se puedan ingresar instrucicones de tipo I y J.

```
        def limpiar_dato(dato):
        """Elimina los s mbolos $, , y ; de un dato."""
        return dato.replace('$', '').replace(',', '').replace(';', '').
            strip()

        def extraer_elementos(linea):
        """Devuelve la instrucci n y los datos de una l nea."""
        partes = linea.strip().split()
        if not partes:
        return None, []  # l nea vac a

        instruccion = partes[0].upper()
        datos = [limpiar_dato(p) for p in partes[1:] if limpiar_dato(p)]
        return instruccion, datos

        def obtener_codigo_binario(instruccion):
        """Devuelve el c digo binario asociado a la instrucci n."""
        # Buscar en instrucciones tipo R
        for clave, valor in instruccion_logica_aritmetica.items():
        if valor == instruccion:
        return ('R', clave)

        # Buscar en instrucciones inmediatas
        for clave, valor in instruccion_logica_aritmetica_inmediata.items
            ():
        if valor == instruccion:
        return ('I', clave)

        # Buscar en instrucciones de memoria
        for clave, valor in instrucciones_memoria.items():
        if valor == instruccion:
        return ('M', clave)

        return (None, None)

        def convertir_a_binario(numero, bits=5):
        """Convierte un n mero entero a binario con la cantidad de bits
            especificada."""
        return format(int(numero), f'0{bits}b')
```

A continuacion se incluye el codigo donde realiza la conversión de la instrucción completa, concatenando e identificando que tipo de instruccion. Cabe aclarar que este codigo incluye conversiones de instrucciones de tipo I, pero por ahora este tema no sera abordado ahora mismo:

```
        def convertir():
        texto = in_assembly.get("1.0", tk.END).strip()
        lineas = texto.splitlines()
        instruccion_convertida_text.delete("1.0", tk.END)
        memoria_text.delete("1.0", tk.END)

        for linea in lineas:
        if not linea.strip():
        continue

        instruccion, datos = extraer_elementos(linea)
        if not instruccion or len(datos) < 2:
        messagebox.showerror("Error", f"L nea inv lida: {linea}")
        continue

        tipo, codigo = obtener_codigo_binario(instruccion)
        if not tipo or not codigo:
        messagebox.showerror("Error", f"Instrucci n desconocida: {
            instruccion}")
        continue

        if tipo == 'R':
        # Instrucci n tipo R: $d, $s, $t
        if len(datos) < 3:
        messagebox.showerror("Error", f"Instrucci n tipo R requiere 3
            operandos: {linea}")
        continue

        rd = convertir_a_binario(datos[0])
        rs = convertir_a_binario(datos[1])
        rt = convertir_a_binario(datos[2])

        # Concatenaci n: opcode + rs + rt + rd + shamt + funct
        instruccion_final = valor_operacion_tipo_r + rs + rt + rd +
            valor_shampt + codigo

        elif tipo == 'I':
        # Instrucci n tipo I: $t, $s, inmediato
        if len(datos) < 3:
        messagebox.showerror("Error", f"Instrucci n tipo I requiere 3
            operandos: {linea}")
        continue

        rt = convertir_a_binario(datos[0])
        rs = convertir_a_binario(datos[1])
        inmediato = convertir_inmediato_a_binario(datos[2])

        # Concatenaci n: opcode + rs + rt + inmediato
        instruccion_final = codigo + rs + rt + inmediato

        elif tipo == 'M':
        rt, rs, offset = parsear_instruccion_memoria(datos)

```

```
        if rt is None or rs is None or offset is None:
        # CORRECCI N: Typo en "inv lido"
        messagebox.showerror("Error", f"Formato inv lido para {
            instruccion}. Use: {instruccion} $t, offset($s)")
        continue

        rt_bin = convertir_a_binario(rt)
        rs_bin = convertir_a_binario(rs)
        offset_bin = convertir_inmediato_a_binario(offset)

        instruccion_final = codigo + rs_bin + rt_bin + offset_bin

        else:
        messagebox.showerror("Error", f"Tipo de instrucci n no soportado:
             {tipo}")
        continue

        binario_en_bytes = dividir_en_bytes(instruccion_final)

        # Insertar en los campos de texto
        instruccion_convertida_text.insert(tk.END, instruccion_final + "\n
            ")
        memoria_text.insert(tk.END, binario_en_bytes + "\n")
```

En este codigo se integran los botones que hacen la llamada a las funciones de conversion y exportacion al archivo:

```
        button_frame = tk.Frame(scrollable_frame)
        button_frame.grid(row=1, column=1, padx=20, pady=(0, 10))

        convertir_button = ttk.Button(button_frame, text="Convertir",
            command=convertir)
        convertir_button.pack(pady=5)

        exportar_button = ttk.Button(button_frame, text="Exportar a
            archivo", command=exportar_a_archivo)
        exportar_button.pack(pady=5)

        convertir_binario_assembly = ttk.Button(button_frame, text="
            Convertir Binario a Ensamblador", command=binario_a_ensamblador
            )
        convertir_binario_assembly.pack(pady=5)
```

Ahora si, con esta logica implementada podemos observar la aplicación GUI, donde se puede apreciar que hay tres bloques donde se ingresa la instrucción en ensamblador, en el segundo la instruccion en un formato de 32 bits y el ultimo la misma o mismas instrucciones en forma de bytes para la lectura de la memoria que esta mostrada en la figura —.

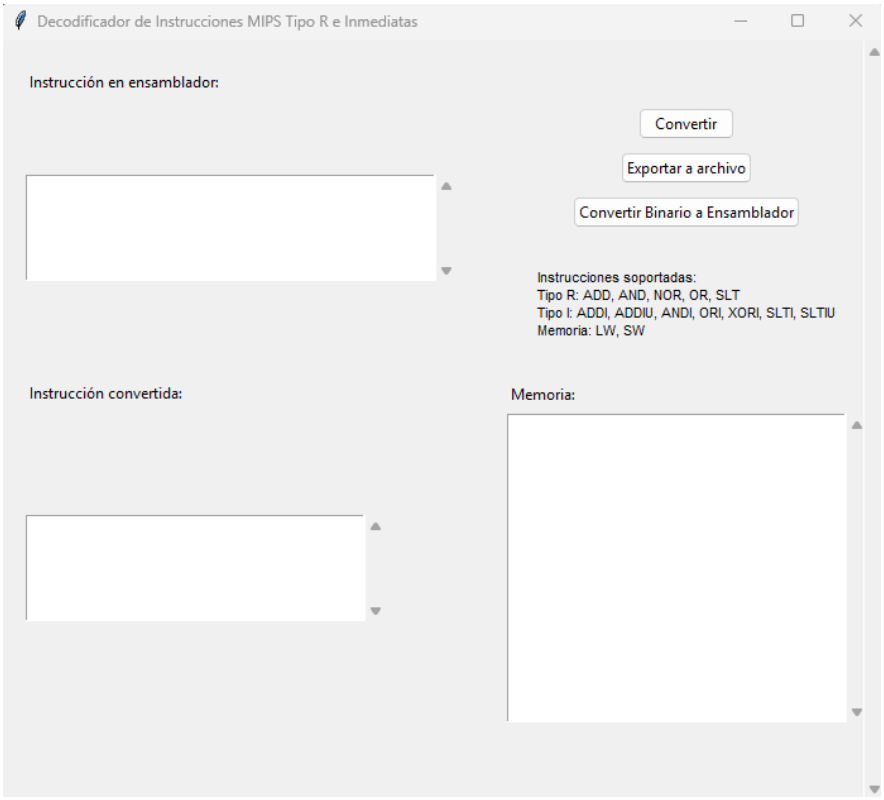

Figura 2: Modelo final de una aplicación GUI de conversión de instrucciones MIPS de 32 bits

Asi pues en la siguiente figura se muestra un ejemplo donde se instroduce la instruccion en ensamblador y convierte la instruccion en binario en la figura 3.

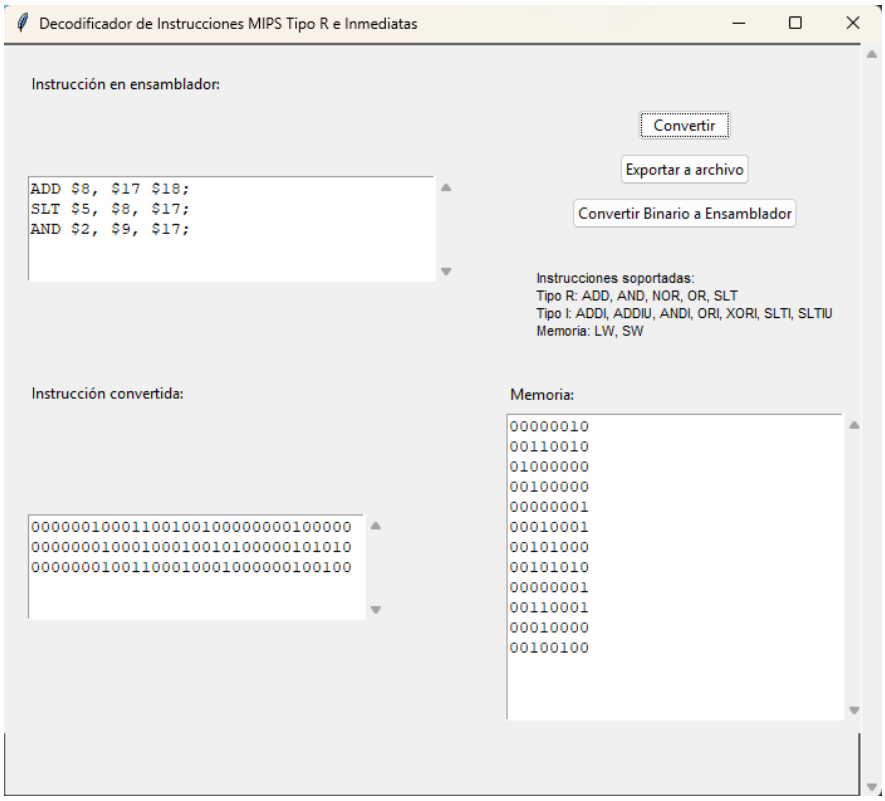

Figura 3: Instrucciones convertidas en binario.

De esta forma, ya es posible darle estos datos al archivo instrucciones.txt copiando los datos que están en el cuadro Memoria, o también esta la función que exporta a un archivo llamado instrucciones.txt y de la misma forma podemos reemplazar el archivo o copiando los datos.

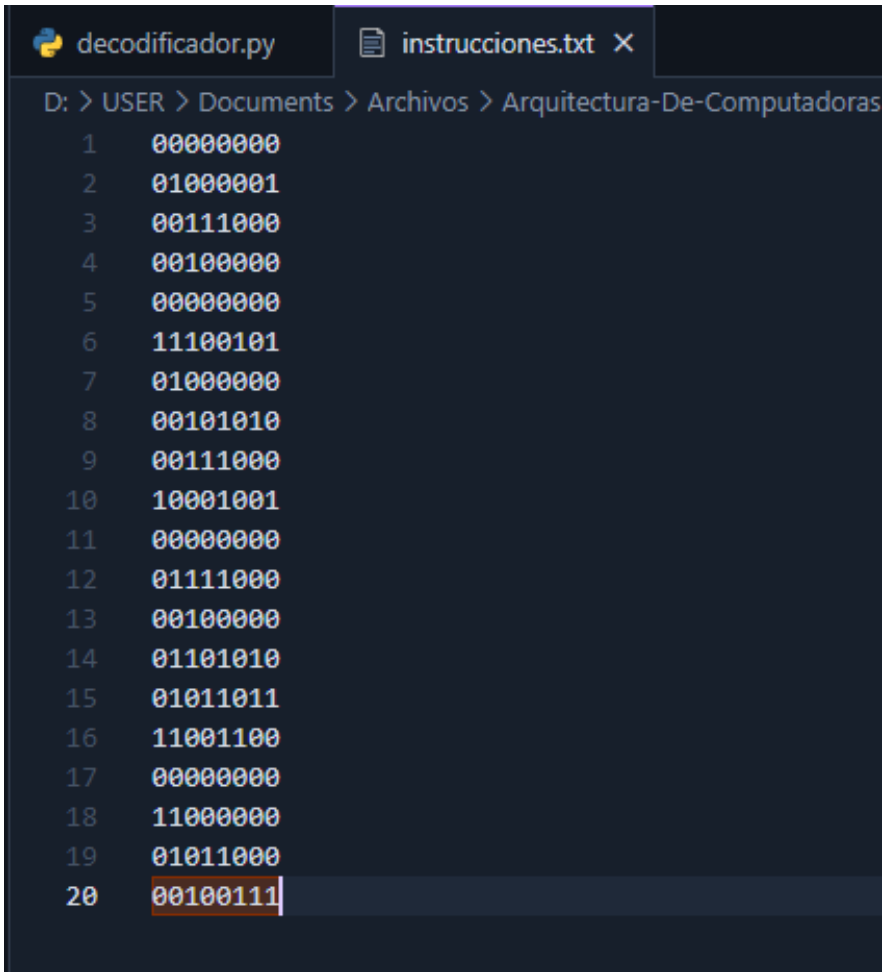

Figura 4: Instrucciones en un archivo de texto.

Referencias

[1] David A. Patterson and John L. Hennessy. *Estructura y diseño de computadores: la interfaz hardware / software*. Reverte, Barcelona, 2011.

[2] William Stallings, Antonio Cañas Vargas, and Alberto Prieto Espinosa. *Organización y arquitectura de computadores*. Pearson Educación, 2006.