
Arquitectura MIPS

Fase 2



12 MAYO

Agredano Daniel
Gomez Alexia
Marquez Cristian
Román Carlos



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.

Ljubisa Bajic es un ingeniero eléctrico y científico de la computación mejor conocido por sus aportaciones en la microarquitectura de los procesadores. Sus contribuciones son muy amplias, las cuales incluyen ciertas innovaciones en la gestión de la energía para mejorar el rendimiento de los chips, la ejecución especulativa, por mencionar algunos. La aportación de Ljubisa Bajic en la microarquitectura, fue haber contribuido a la mejora de la eficiencia energética y el rendimiento de los chips.

Jim keller es un ingeniero muy conocido por su aportación en ciertos trabajos como Apple, Tesla y AMD, por mencionar los más conocidos, su enfoque ha contribuido en el aumento del rendimiento de los procesadores modernos, el cual estuvo involucrado en muchas ocasiones en la microarquitectura, como ser el arquitecto de la clave k8, y estar involucrado en el diseño de ISA x86-64 o AMD64, y la arquitectura x86 utilizadas en los procesadores AMD Ryzen y los procesadores móviles de Apple.

Raja Koduri es un ingeniero informático y ejecutivo de hardware de gráficos por computadora, trabajó en Intel como vicepresidente ejecutivo en la división de arquitectura, en el diseño del GPU y en la integración de gráficos integrados en procesadores. Tuvo una participación importante en el desarrollo de arquitecturas como las series Radeon R9 y Radeon RX.

Los formatos de **Big Endian** y **Little Endian** son dos formas de ordenar los bytes en la memoria de una computadora, especialmente cuando son números enteros de muchos bytes.

- Formato de Big Endian, el byte más significativo (el más grande) de un número se almacena en la dirección de memoria más baja.
- Formato de Little Endian, el byte menos significativo (el más pequeño) de un número se almacena en la dirección de memoria más baja.

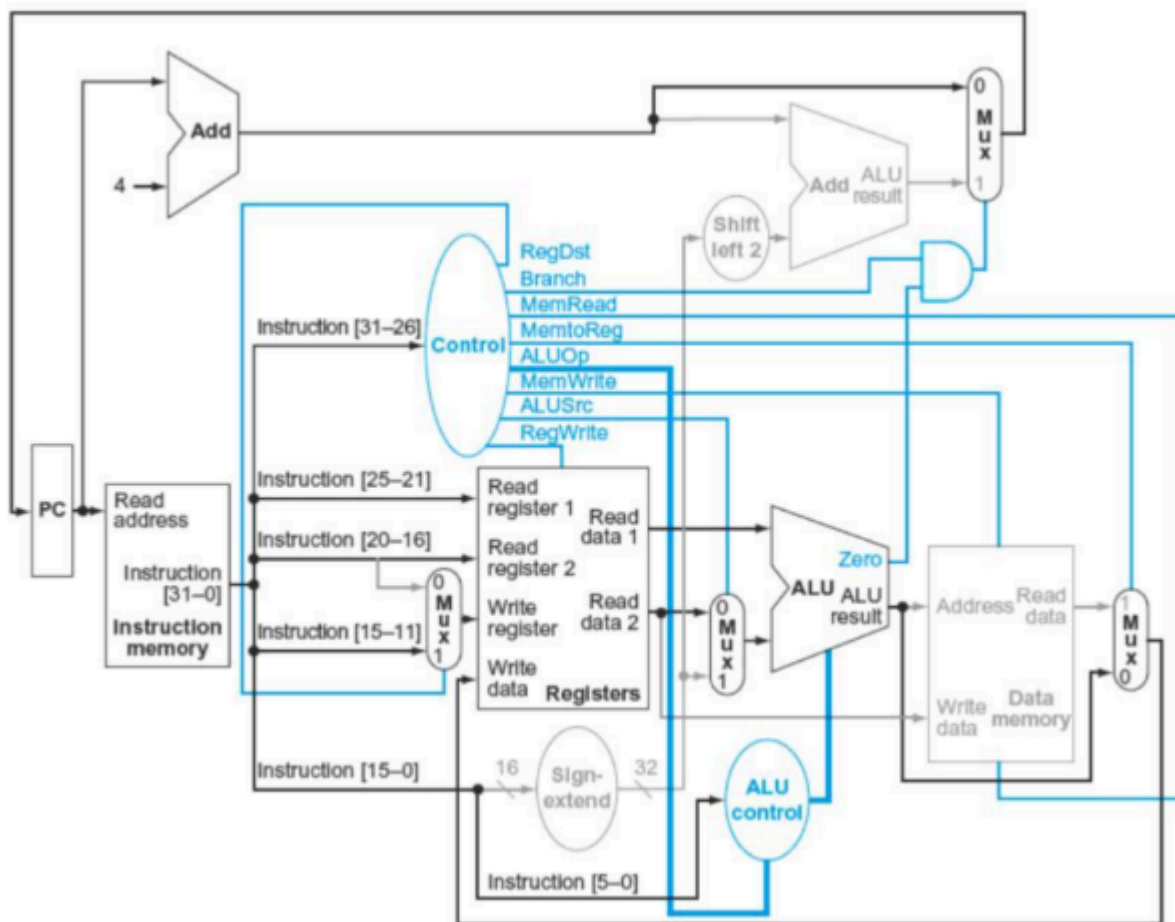
Estos formatos ayudan en el diseño de la arquitectura de un procesador, porque no es necesario realizar conversiones para la interpretación de los datos.

Tipos de arquitecturas (computadoras modernas):

- **Harvard:** es una arquitectura que cuenta con memorias de almacenamiento y de señal físicamente separadas para las instrucciones y los datos. Cuenta con dos buses de datos, uno es para transferir datos entre la unidad central de procesamiento y la memoria de instrucciones, y otro para transferir datos entre la CPU y la memoria de datos.
- **Von Neumann:** esta arquitectura es muy versátil porque tiene una unidad central de proceso (CPU), la cual está conectada a una memoria (RAM) compartida para almacenar las instrucciones de datos. El tamaño de la unidad de datos es fijo de 8 bytes.
- **SOA:** es la arquitectura orientada a los servicios, se utiliza principalmente en HTTP, XML y SOAP para diseñar el software y desarrollar sistemas distribuidos. Siendo muy flexible en la integración de sistemas.
- **Pipeline:** transforma el flujo de datos en un proceso de varias fases, es intérprete de comandos. Se utiliza para mejorar el rendimiento de los procesadores, porque se puede ejecutar varias instrucciones en diferentes etapas.

Desarrollo

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.



Lo más importante es que se desarrolló de manera que el módulo sólo recibe una instrucción que posteriormente se “secciona” para mandar determinados bits a las entradas de los módulos correspondientes.

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
RDir1	Input	05 bits
RDir2	Input	05 bits
WData	Input	32 bits
WDir	Input	05 bits
RW	Input	01 bits
ROut1	Output Reg	32 bits
ROut2	Output Reg	32 bits

ALU Control		
Nombre	Tipo	Longitud
Funct	Input	06 bits
Sel	Input	03 bits
Out	Output Reg	04 bits

Instruction memory		
Nombre	Tipo	Longitud
Dir	Input	32 bits
Out	Output Reg	32 bits

ALU		
Nombre	Tipo	Longitud
A	Input	32 bits
B	Input	32 bits
Sel	Input	4 bits
Out	Output Reg	32 bits
ZF	Output Reg	01 bits

Multiplexor		
Nombre	Tipo	Longitud
A	Input	32 bits
B	Input	32 bits
Sel	Input	01 bits
Out	Output Reg	32 bits

MUX 5		
Nombre	Tipo	Longitud
A	Input	05 bits
B	Input	05 bits
Sel	Input	01 bits
Out	Output Reg	05 bits

Memoria		
Nombre	Tipo	Longitud
WData	Input	32 bits
WDir	Input	32 bits
Sel	Input	01 bits
Out	Output Reg	32 bits

Unidad de Control		
Nombre	Tipo	Longitud
op	Input	6 bits
RegDst	Output Reg	01 bits
Branch	Output Reg	01 bits
MemRead	Output Reg	01 bits
MemToReg	Output Reg	01 bits
AluOp	Output Reg	01 bits
MemWrite	Output Reg	01 bits
AluSrc	Output Reg	01 bits
RegWrite	Output Reg	01 bits

ADD		
Nombre	Tipo	Longitud
In	Input	32 bits
Out	Output Reg	32 bits

PC		
Nombre	Tipo	Longitud
In	Input	32 bits
CLK	Input	01 bits
Out	Output Reg	32 bits

Band		
Nombre	Tipo	Longitud
A	Input	01 bits
B	Input	01 bits
Out	Output Reg	01 bits

Código:

/// ALU.v ///

```
module ALU(
    input [31:0] A,
    input [31:0] B,
    input [3:0] Sel,
    output reg [31:0] Out,
    output reg zf
);

always @(*) begin
    case (sel)
        4'b0000: res = A & B;
        4'b0001: res = A | B;
        4'b0010: res = A + B;
        4'b0110: res = A - B;
```

```

        4'b0111: res = (A < B) ? 1 : 0;
        4'b1100: res = ~(A | B);
        default: res = 0;
    endcase
    zf = (res == 0) ? 1 : 0;
end
endmodule

```

/// AluControl.v ///

```

module ALU_CTRL(
    input [5:0]Funct,
    input [2:0]Sel,

    output reg [3:0]Out
);

always @(*)
begin
    if(Sel==3'b000)begin
        case (Funct)
            6'b100000:
                Out = 4'b0010;
            6'b100010:
                Out = 4'b0110;
            6'b100100:
                Out = 4'b0000;
            6'b100101:
                Out = 4'b0001;
            6'b100111:
                Out = 4'b1100;
            6'b101010:
                Out = 4'b0111;
        endcase
    end
end
endmodule

```

/// Banco.v ///

```
module REG_BANK(  
    input [4:0]RDir1,  
    input [4:0]RDir2,  
    input [31:0]WData,  
    input [4:0]WDir,  
    input RW,  
  
    output reg [31:0]ROut1,  
    output reg [31:0]ROut2  
);  
  
reg [31:0] MEM [0:31];  
  
assign ROut1 = MEM[RDir1];  
assign ROut2 = MEM[RDir2];  
  
initial begin  
    $readmemh("Memory/BR1.txt",MEM) ;  
end  
  
always @(*) begin  
    if(RW)  
        MEM[WDir]=WData;  
end  
endmodule
```

/// Memoria.v ///

```
`timescale 1ns/1ns  
  
module RMEM(  
    input [31:0]WData,  
    input [31:0]WDir,  
    input Sel,  
  
    output reg[31:0]Out  
);
```

```

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

initial
begin
    $readmemb("Memory/MEM1.txt", MEM);
end

always @(*)
begin
    if(Sel)
        MEM[WDir]=WData;
    else
        Out=MEM[WDir];
end
endmodule

```

/// Multiplexor.v ///

```

module MUX(
    input [31:0]A,
    input [31:0]B,
    input Sel,
    output reg [31:0]Out
);

always @(*)
begin
    if (Sel == 0)
        Out = A;
    else
        Out = B;
end
endmodule

```

/// Multiplexor_5.v ///

```

module MUX_5(
    input [4:0]A,
    input [4:0]B,
    input Sel,
    output reg [4:0]Out

```

```
);

always @(*)
begin
    if (Sel == 0)
        Out = A;
    else
        Out = B;
end
endmodule
```

/// UnidadControl.v ///

```
module CTRL_U(
    input [5:0]op,

    output reg RegDst,
    output reg Branch,
    output reg MemRead,
    output reg MemToReg,
    output reg [2:0]AluOp,
    output reg MemWrite,
    output reg AluSrc,
    output reg RegWrite
);

always @(*)
begin
    case (op)
        5'b00000:begin
            RegDst      = 1'b1 ;
            Branch      = 1'b0 ;
            MemRead     = 1'bx ;
            MemToReg    = 1'b0 ;
            AluOp       = 3'b000 ;
            MemWrite    = 1'b0 ;
            AluSrc      = 1'b0 ;
            RegWrite    = 1'b1 ;

            end
    end
```



```
        endcase
    end
endmodule
```

/// Instruction_Memory.v ///

```
module INS_MEM(
    input [31:0]Dir,

    output reg [2:0]Out
);

reg [7:0]MEM[0:255];

initial
begin
    $readmemb("Memory/INS1.txt",MEM);
end

always @(*)
begin
    Out={
        MEM[Dir],
        MEM[Dir + 1],
        MEM[Dir + 2],
        MEM[Dir + 3]
    };
    end
endmodule
```

/// Branch_AND.v ///

```
module BAND(
    input A,
    input B,

    output reg Out
```

```
);  
  
always @(*)  
begin  
    Out= A + B;  
end  
endmodule
```

```
/// ADD.v ///
```

```
module ADD(  
    input [31:0]In,  
  
    output reg [31:0]Out  
);  
  
always @(*)  
begin  
    Out= In + 4;  
end  
endmodule
```

```
/// PCC.v ///
```

```
module ADD(  
    input [31:0]In,  
    input CLK,  
  
    output reg [31:0]Out  
);  
  
always @(posedge CLK)  
begin  
    Out = In;  
end  
  
initial  
begin  
    Out = 31'd0;
```

```
end  
endmodule
```

```
/// Todo_Fase2.v ///
```

```
module Todo_F2 ( );  
  
reg CLK;  
  
wire [31:0] W_PC1;  
wire [31:0] W_ADD1;  
wire        W_AND1;  
wire [31:0] W_INS1;  
wire [31:0] W_BR1_O1;  
wire [31:0] W_BR1_O2;  
wire [31:0] W_ALU1;  
wire        W_ALU1_ZF;  
wire [31:0] W_MEM1;  
wire [31:0] W_ALUC1;  
wire        W_UC1_RD;  
wire        W_UC1_Br;  
wire        W_UC1_MR;  
wire        W_UC1_MtR;  
wire [2:0]  W_UC1_AOp;  
wire        W_UC1_MW;  
wire        W_UC1_MW;  
wire        W_UC1_RW;  
wire [31:0] W_MUX1;  
wire [31:0] W_MUX2;  
wire [31:0] W_MUX3;  
wire [4:0]  W_MUX4;  
  
PC PC1(  
    .In(W_MUX2) ,  
    .CLK(CLK) ,  
  
    .Out(W_PC1)  
);
```

```

ADD ADD1 (
    .In(W_PC1) ,

    .Out(W_ADD1)
);

BAND AND1 (
    .A(W_UC1_Br) ,
    .B(W_ALU1_ZF) ,

    .Out(W_AND1)
);

INS_MEM INS1 (
    .Dir(W_PC1) ,

    .Out(W_INS1)
);

REG_BANK BR1 (
    .RDir1(W_INS1[25:21]) ,
    .RDir2(W_INS1[20:16]) ,
    .WDir(W_MUX4) ,
    .WData(W_MUX1) ,

    .RW(W_UC1_RW) ,

    .ROut1(W_BR1_O1) ,
    .ROut2(W_BR1_O2)
);

ALU ALU1 (
    .A(W_BR1_O1) ,
    .B(W_MUX3) ,

    .Sel(W_ALUC1) ,

    .ZF(W_ALU1_ZF) ,

```

```

        .Out(W_ALU1)

);

RMEM MEM1(
    .WData(W_BR1_O2),
    .WDir(W_ALU1),

    .Sel(W_UC1_MW),

    .Out(W_MEM1)

);

ALU_CTRL ALUC1(
    .Funct(W_INS1[5:0]),

    .Sel(W_UC1_AOp),

    .Out(W_ALUC1)

);

CTRL_U UC1(
    .op(W_INS1[31:26]),

    .RegDst(W_UC1_RD),
    .Branch(W_UC1_Br),
    .MemRead(W_UC1_MR),
    .MemToReg(W_UC1_MtR),
    .AluOp(W_UC1_AOp),
    .MemWrite(W_UC1_MW),
    .AluSrc(W_UC1_ASr),
    .RegWrite(W_UC1_RW)

);

MUX MUX1(
    .B(W_MEM1),
    .A(W_ALU1),

```

```

        .Sel(W_UC1_MtR) ,

        .Out(W_MUX1)
    );

MUX MUX2 (
    .A(W_ADD1) ,
    .B() ,//Vacio

    .Sel(W_AND1) ,

    .Out(W_MUX2)
);

MUX MUX3 (
    .A(W_BR1_O2) ,
    .B() ,//vacio

    .Sel(W_UC1_ASr) ,

    .Out(W_MUX3)
);

MUX_5 MUX4 (
    .A(W_INS1[20:16]) ,
    .B(W_INS1[15:11]) ,

    .Sel(W_UC1_RD) ,

    .Out(W_MUX4)
);

initial
begin
    CLK = 1'b0;
    forever #100 CLK = ~CLK;
end
endmodule

```

/// TB.v ///

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

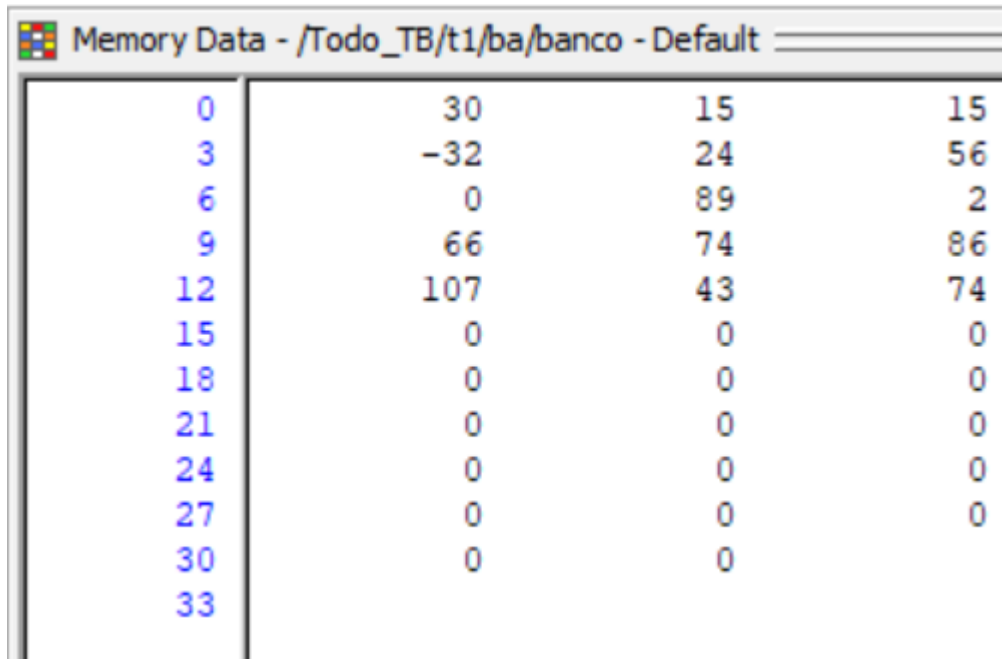
reg [31:0]in;
reg [31:0]out;

initial
    begin
        in=32'd0;
        #100;

        in=32'd4;
        #100;
    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:



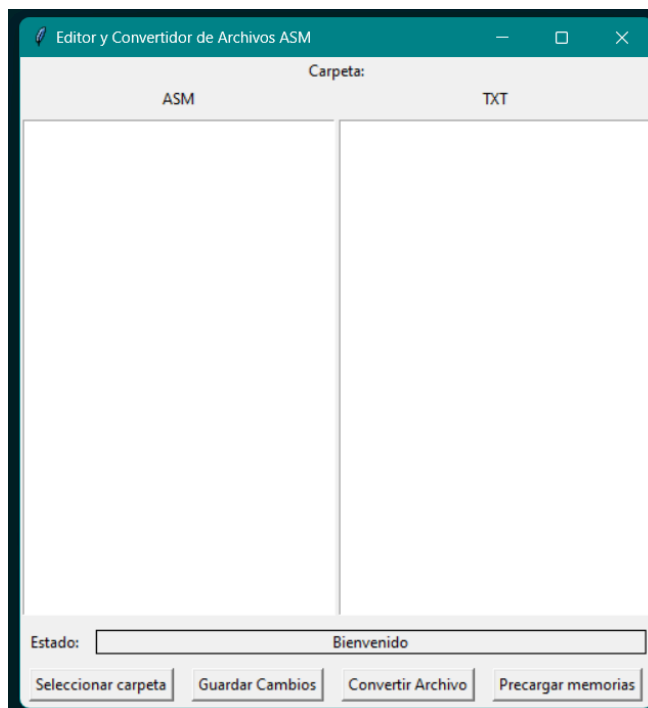
The image shows a screenshot of a software window titled "Memory Data - /Todo_TB/t1/ba/banco - Default". The window displays a table with four columns. The first column lists register addresses from 0 to 33 in increments of 3. The second, third, and fourth columns show the values stored in those registers. The values for registers 0 through 12 are non-zero, while registers 15 through 33 all contain the value 0.

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

Interfaz Principal



Selección de Carpeta:

Al abrir el programa, verás una etiqueta "Carpeta:" y un botón "Seleccionar carpeta". Haz clic en el botón "Seleccionar carpeta" para elegir la carpeta que contiene los archivos de memorias y ensamblador que desees editar y convertir.

Edición de Código ASM:

En el lado izquierdo de la ventana, encontrarás un área de texto etiquetada como "ASM".

Aquí puedes ver y editar el código ASM del archivo seleccionado en la carpeta.

Si no hay ningún archivo de ASM en la carpeta seleccionada, este área estará vacía.

Visualización de Código Binario:

En el lado derecho de la ventana, encontrarás otra área de texto etiquetada como "TXT".

Esta área mostrará el código binario correspondiente al código ASM editado.
El código binario se generará automáticamente cuando conviertas el archivo .ASM si no se presentó ningún error.

Botones de Acción:

En la parte inferior de la ventana, verás varios botones para realizar acciones:

"Guardar Cambios":

Guarda los cambios realizados en el código ASM.

"Convertir Archivo":

Convierte el código ASM a su representación binaria y muestra el resultado en el área de texto "TXT".

"Precargar memorias":

Abre una ventana para editar las memorias.

Edición de Memorias



Contenido de las memorias:

En la ventana de edición de memorias, verás dos áreas de texto: una para el "Banco de Memoria" y otra para la "Memoria".

El "Banco de Memoria" muestra el contenido de los registros.

La "Memoria" muestra el contenido de la memoria del sistema.

Botones de Carga y Guardado:

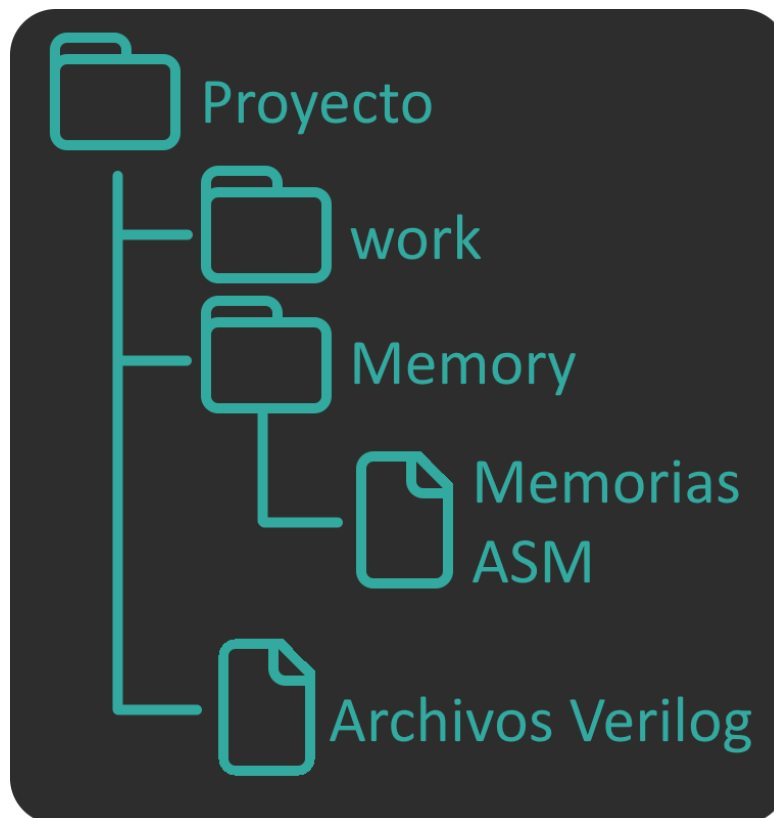
Utiliza los botones "Abrir banco" y "Cargar banco" para cargar y guardar el contenido del banco de memoria.

Utiliza los botones "Abrir memoria" y "Cargar memoria" para cargar y guardar el contenido de la memoria del sistema.

Consejos Adicionales

Asegúrate de seleccionar la carpeta correcta que contiene los archivos ASM y TXT que deseas editar y convertir.

Memory es la carpeta desde la que la simulación carga las memorias.



Si se detecta algún error, este se mostrará en la etiqueta de estado, cualquier acción que realices se reflejará como exitosa en esta misma etiqueta.

A continuación se muestra una breve descripción de las funciones que se implementaron en el código de python (Manual de desarrollo)

Instrucciones en lenguaje ensamblador codificadas

Instrucciones codificadas de la siguiente manera:

<instrucción> : [<tipo>,<func>]

```

instructions ={
    #Instrucciones R
    "add" : ["R","100000"],
    "sub" : ["R","100010"],
    "and" : ["R","100100"],
    "or"  : ["R","100101"],
    "nor" : ["R","100111"],
    "slt" : ["R","101010"],
    "nop" : ["R","000000"],
    #Instrucciones I
    #Aritméticas
    "addi" : ["I","001000"],
    "andi" : ["I","001100"],
    "ori"  : ["I","001101"],
    "slti" : ["I","001010"],
    #Memoria
    "lw"   : ["I","100011"],
    "sw"   : ["I","101011"],
    #Ramificación
    "beq"  : ["I","000100"],
    "bne"  : ["I","000101"],
    "bgtz" : ["I","000111"],
    #Instrucciones J
    "j"    : ["J","000010"]
}

```

__init__(self) (Constructor de la clase `MainWindow`):

Constructor de la clase ``MainWindow``. Este método se llama automáticamente cuando se crea un objeto de la clase ``MainWindow``. En este método, se inicializan los atributos de la clase, se crea la interfaz gráfica de la ventana principal (``Tkinter``), se definen los elementos de la interfaz como etiquetas, botones y cuadros de texto, se establecen los eventos de los botones y se inicia el bucle principal de la aplicación.

cargar_archivo_asm(self):

Este método se encarga de cargar el contenido de un archivo ASM y mostrarlo en el cuadro de texto correspondiente en la ventana principal. Utiliza un cuadro de diálogo para que el usuario seleccione la carpeta que contiene el archivo ASM. Luego, verifica si existen los archivos "INS1.asm" e "INS1.txt" en la carpeta seleccionada. Si existen, lee el contenido de estos archivos y lo muestra en los cuadros de texto respectivos.

guardar_cambios(self):

Este método guarda los cambios realizados en el archivo ASM. Obtiene el contenido del cuadro de texto que contiene el código ASM y lo guarda en el archivo "INS1.asm" en la carpeta seleccionada.

crear_ventana(self):

Este método crea una nueva ventana para editar las memorias. Se instancia la clase ``MemWindow``, pasando como argumento la carpeta seleccionada en la ventana principal.

IntStr_Bin(self, cadena, ancho):

Este método convierte una cadena de números enteros en su representación binaria, con un ancho especificado. También maneja números negativos utilizando el método de complemento a 2.

splitLines(self, string, n):

Este método divide una cadena en subcadenas de longitud ``n`` y las une con saltos de línea.

instruccion_R(self, line):

Este método construye la representación binaria de una instrucción de tipo R (registro), utilizando los valores de los registros y la operación específica definida en la instrucción.

instruccion_I(self, line):

Este método construye la representación binaria de una instrucción de tipo I (inmediato), utilizando los valores de los registros y el valor inmediato especificado en la instrucción.

instruccion_J(self, line):

Este método construye la representación binaria de una instrucción de tipo J (salto), utilizando la dirección de salto especificada en la instrucción.

linea_valida(self, linea):

Este método valida si una línea de código en formato ASM es válida, verificando si tiene el número correcto de operandos y si los tipos de operandos son correctos.

procesar_instrucciones(self, words):

Este método procesa las instrucciones en formato ASM, convirtiéndolas a su representación binaria correspondiente. Itera sobre cada línea de código, valida su formato y construye la representación binaria utilizando los métodos `instruccion_R`, `instruccion_I` y `instruccion_J`.

convertir_archivo(self):

Este método se encarga de convertir el código ASM ingresado en el cuadro de texto a su representación binaria correspondiente y mostrarlo en el cuadro de texto de la ventana principal. Utiliza el método `procesar_instrucciones` para realizar la conversión.

__init__(self, path):

Constructor de la clase `MemWindow`. Se encarga de inicializar la ventana de edición de memorias, creando la interfaz gráfica y los elementos necesarios.

cargar_memoria(self, archivo, texto):

Este método carga el contenido de un archivo de memoria y lo muestra en el cuadro de texto correspondiente en la ventana de edición de memorias.

escribir_memoria(self, archivo, texto):

Este método guarda el contenido modificado de un archivo de memoria en el sistema de archivos.

Código:

A continuación se muestra el código en python con las funciones para generar la GUI y poder editar y convertir archivos de ensamblador MIPS a formato binario, además de permitir cargar y guardar archivos de memoria.

Las clase MainWindow representa la ventana principal y nos permite cargar archivos ASM, guardar cambios, convertir archivos y manejar errores, además de que incluimos una ventana secundaria MemWindow la cual nos permite editar archivos de memoria.

Al instanciar un objeto de la clase MainWindow, se inicia la aplicación y se muestra la interfaz gráfica al usuario, para que se pueda editar y trabajar con archivos de ensamblador MIPS y de memoria.

```
import tkinter as tk
from tkinter import filedialog
from tkinter import ttk
import os

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

    #Instrucciones R

    "add"    : ["R","100000"],
    "sub"    : ["R","100010"],
    "and"    : ["R","100100"],
    "or"     : ["R","100101"],
    "nor"    : ["R","100111"],
    "slt"    : ["R","101010"],
    "nop"    : ["R","000000"],

    #Instrucciones I

    #Aritméticas

    "addi"   : ["I","001000"],
    "subi"   : ["I","000000"],#No encontrado
    "andi"   : ["I","001100"],
    "ori"    : ["I","001101"],
```

```

"nori" : ["I","000000"],#No encontrado
"slti" : ["I","001010"],
#Memoria
"lw" : ["I","100011"],
"sw" : ["I","101011"],
#Ramificación
"beq" : ["I","000100"],
"bne" : ["I","000101"],
"bgtz" : ["I","000111"],
#Instrucciones J
"j" : ["J","000010"]
}

class MainWindow:
    def __init__(self):
        self.path = ""
        self.estatus = True
        self.error = ""

        self.root = tk.Tk()
        self.root.title("Editor y Convertidor de Archivos ASM")
        #Divisiones
        self.fTexts = ttk.Frame(self.root, width=80)
        self.fASM = ttk.Frame(self.fTexts)
        self.fBIN = ttk.Frame(self.fTexts)
        self.fLog = ttk.Frame(self.root)
        self.fButtons = ttk.Frame(self.root)
        #Elementos
        self.label_path = tk.Label(self.root, text="Carpeta:")

```

```

        self.label_log = tk.Label(self.fLog, text="Estado:")

        self.label_mensajes = tk.Label(self.fLog, text="Bienvenido",
borderwidth=1, relief="solid")

        self.label1 = tk.Label(self.fASM, text="ASM")

        self.asm_text = tk.Text(self.fASM, wrap=tk.WORD, width=30)

        self.label2 = tk.Label(self.fBIN, text="TXT")

        self.bin_text = tk.Text(self.fBIN, wrap=tk.WORD, width=30)

        self.cargar_asm_button = tk.Button(self.fButtons, text="Seleccionar
carpeta", command=self.cargar_archivo_asm)

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

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

        self.memoria_button = tk.Button(self.fButtons, text="Precargar memorias",
command=self.crear_ventana)

        #Acomodo

        self.label_path.pack(fill="x")

        self.fTexts.pack(fill="y")

        self.fASM.pack(side=tk.LEFT)

        self.fBIN.pack(side=tk.LEFT)

        self.fLog.pack(fill="x")

        self.fButtons.pack(fill="x")


        self.label1.pack(padx=1, pady=1)

        self.asm_text.pack(padx=2, pady=5, expand=True)

        self.label2.pack(padx=2, pady=1)

        self.bin_text.pack(padx=1, pady=5, expand=True)

        self.label_log.pack(padx=5, pady=5, side=tk.LEFT)

        self.label_mensajes.pack(padx=5, pady=5, side=tk.LEFT, expand=True,
fill='x')

        self.cargar_asm_button.pack(padx=5, pady=5, side=tk.LEFT, expand=True)

        self.modificar_button.pack(padx=5, pady=5, side=tk.LEFT, expand=True)

```

```

self.guardar_button.pack(padx=5, pady=5, side=tk.LEFT, expand=True)
self.memoria_button.pack(padx=5, pady=5, side=tk.LEFT, expand=True)

self.root.mainloop()

def cargar_archivo_asm(self):
    path = filedialog.askdirectory()

    if path: #Verifica si el usuario dió una carpeta
        message="Carpeta seleccionada"

        self.path = path

        self.label_path.configure(text="Carpeta: "+self.path) #Muestra el
directorio

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

        if(os.path.exists(path+"/INS1.asm")): #Verifica que el ASM exista
            with open(path+"/INS1.asm", "r") as file:
                message += ", ASM cargado"

                self.asm_text.insert(tk.END, file.read())

        if(os.path.exists(path+"/INS1.txt")):
            with open(path+"/INS1.txt", "r") as file: #Abre el TXT
                message += ", TXT cargado"

                self.bin_text.insert(tk.END, file.read())

        self.label_mensajes.config(text=message)

def guardar_cambios(self):
    self.label_mensajes.config(text="Cambios guardados")

```

```

        with open(self.path+"/INS1.asm", "w") as asm_file: #Abre el archivo de
ensamblador

            asm_file.write(self.asm_text.get("1.0", tk.END).removesuffix("\n"))

def crear_ventana(self):

    self.label_mensajes.config(text="Abriendo memorias")

    MemWindow(self.path) #Crea una nueva ventana para editar las memorias

def IntStr_Bin(self, cadena, ancho):

    cadena = cadena.replace("$","")

    num = int(cadena)

    fill = '1' if num<0 else '0' #selecciona el caracter con el que se
rellenará la cadena (0 positivo/1 negativo)

    num = (pow(2,ancho)+num) if num<0 else num #Le hace complemento A2 al
número

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

def splitLines(self, string, n):

    return '\n'.join([(string[i:i+n]) for i in range(0, len(string), n)])

def instruccion_R(self, line): #Arma el formato R

    ins = ""

    ins += "000000" #Op code

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

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

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

    ins += "00000" #Shamt

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

    return self.splitLines(ins,8) #Secciona la instrucción en 4 partes

```

```

def instruccion_I(self, line): #Arma el formato I
    ins = ""
    ins += instructions[line[0]][1] #Opcode
    ins += self.IntStr_Bin(line[2],5) #rs
    ins += self.IntStr_Bin(line[1],5) #rt
    ins += self.IntStr_Bin(line[3],16) #Inmediato
    return self.splitLines(ins,8) #Secciona la instrucción en 4 partes

def instruccion_J(self, line): #Arma el formato J
    ins = ""
    ins += instructions[line[0]][1] #Opcode
    ins += self.IntStr_Bin(line[1],26) #Dirección
    return self.splitLines(ins,8) #Secciona la instrucción en 4 partes

def linea_valida(self, linea):
    tipo = instructions[linea[0]][0] #Guarda el tipo de instrucción para
evaluarla
    if tipo == "R":
        if len(linea)<4:
            self.error = "Faltan operandos"
            return False
        for i in range(1,4):
            if linea[i][0] != '$':
                self.error = "Se esperaba una direccion"
                return False
    if tipo == "I":
        if len(linea)<4:
            self.error =
            return False
        for i in range(1,3): #Para los primeros 2 operandos

```

```

        if linea[i][0] != '$':
            self.error = "Se esperaba una direccion"
            return False

        if linea[3][0] == '$': #Para el último operando verifica que no tenga
'$'
            self.error = "Se esperaba un valor inmediato"
            return False

    if tipo == "J":
        if len(linea)<2:
            self.error = "Faltan operandos"
            return False

        if linea[1][0] == '$':
            self.error = "Se esperaba un valor inmediato"
            return False

    return True

def procesar_instrucciones(self, words):
    binary = []

    for n,line in enumerate(words): #Revisa cada línea y le asigna un número
        if line[0] in instructions:
            if(not self.linea_valida(line)):
                return "Error en linea "+str(n)+": "+self.error

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

            if instructions[line[0]][0]=="I":
                binary.append(self.instruccion_I(line))

            if instructions[line[0]][0]=="J":
                binary.append(self.instruccion_J(line))

```

```

        else: #Si es una insrucción inválida
            return "Error en linea "+str(n)+": Instruccion no valida"

    return binary

def convertir_archivo(self):
    text = self.asm_text.get("1.0", tk.END) #Lee el contenido del cuadro de
texto
    words = [line.split(" ") for line in text.splitlines()] #Separa el texto
en lineas y palabras (arreglo bidimensional)

    binary = self.procesar_instrucciones(words)

    if(type(binary)==str):
        self.label_mensajes.config(text=binary)
        return
    self.label_mensajes.config(text="Convertido sin errores")

    binary = '\n'.join(binary) #Recibe las instrucciones procesadas y las une
con saltos de linea

    with open(self.path+"/INS1.txt", "w") as txt_file:
        txt_file.write(binary)
        self.bin_text.delete("1.0", tk.END)
        self.bin_text.insert(tk.END, binary)

class MemWindow:
    def __init__(self, path):
        self.path = path

        self.memories = tk.Toplevel() #Inicializa la ventana

```



```

self.memories.title("Editor de memorias")

#Divisiones

self.fTexts = ttk.Frame(self.memories, width=80)

self.fASM = ttk.Frame(self.fTexts)

self.fBIN = ttk.Frame(self.fTexts)

self.fButtons = ttk.Frame(self.memories)

#Elementos

self.label3 = tk.Label(self.fASM, text="Banco de memoria")

self.bm_text = tk.Text(self.fASM, wrap=tk.WORD, width=32, height=32)

self.label4 = tk.Label(self.fBIN, text="Memoria")

self.m_text = tk.Text(self.fBIN, wrap=tk.WORD, width=32, height=32)

self.cargar_bm_button = tk.Button(self.fButtons, text="Abrir banco",
command=lambda: self.cargar_memoria("BR1.txt",self.bm_text))

self.guardar_bm_button = tk.Button(self.fButtons, text="Cargar banco",
command=lambda: self.escribir_memoria("BR1.txt",self.bm_text))

self.cargar_m_button = tk.Button(self.fButtons, text="Abrir memoria",
command=lambda: self.cargar_memoria("MEM1.txt",self.m_text))

self.guardar_m_button = tk.Button(self.fButtons, text="Cargar memoria",
command=lambda: self.escribir_memoria("MEM1.txt",self.m_text))

#Acomodo

self.fTexts.pack(fill="y")

self.fASM.pack(side=tk.LEFT)

self.fBIN.pack(side=tk.LEFT)

self.fButtons.pack(fill="x")

self.label3.pack(padx=1, pady=1)

self.bm_text.pack(padx=2, pady=5, expand=True)

self.label4.pack(padx=2, pady=1)

self.m_text.pack(padx=1, pady=5, expand=True)

self.cargar_bm_button.pack(padx=5, pady=5, side=tk.LEFT, expand=True)

self.guardar_bm_button.pack(padx=5, pady=5, side=tk.LEFT, expand=True)

self.cargar_m_button.pack(padx=5, pady=5, side=tk.LEFT, expand=True)

```

```
self.guardar_m_button.pack(padx=5, pady=5, side=tk.LEFT, expand=True)

self.cargar_memoria("BR1.txt",self.bm_text) #Muestra el banco de registros
self.cargar_memoria("MEM1.txt",self.m_text) #Muestra la memoria

def cargar_memoria(self, archivo, texto):
    texto.delete("1.0", tk.END) #Limpia contenido anterior
    if(os.path.exists(self.path+"/"+archivo)):
        with open(self.path+"/"+archivo, "r") as file:
            texto.insert(tk.END, file.read()) # Muestra contenido en el Text

def escribir_memoria(self, archivo, texto):
    with open(self.path+"/"+archivo, "w") as asm_file:
        asm_file.write(texto.get("1.0", tk.END).removesuffix("\n"))

MainWindow()
```

Conclusiones

Arturo Daniel Agredano Gutiérrez:

El trabajo realizado en la implementación del Editor y Convertidor de Archivos ASM representa un gran avance en términos de funcionalidad y utilidad. La adición de características como la edición de memorias, el manejo de diferentes tipos de instrucciones (I y J), y la verificación de la sintaxis del código ASM agregan un valor significativo al programa.

La capacidad de editar y guardar el contenido de las memorias proporciona a los usuarios una herramienta integral para trabajar con sus programas de ensamblador, permitiéndoles simular y modificar el estado de la memoria del sistema según sea necesario. La inclusión de los tipos de instrucciones I y J amplía la versatilidad del programa, ya que ahora puede manejar una gama más amplia de instrucciones, lo que lo hace más útil para una variedad de proyectos de ensamblador. Además, la verificación de la sintaxis del código ASM ayuda a prevenir errores comunes y proporciona retroalimentación inmediata al usuario, lo que facilita el proceso de desarrollo y depuración.

Carlos Eduardo Román Bravo:

Por mi parte, en esta segunda fase pude entender mejor la manera de funcionar del proyecto desde la perspectiva de verilog, sobre todo la parte del PC, que es el encargado de acomodar las instrucciones de la memoria de instrucciones, de bloques de 8 bits, a una sola línea de 32. El proyecto ya tenía mucho avance, puesto que no se tuvieron que crear ni modificar en demasía todos los demás módulos. Por esa razón, considero que no fue muy pesado.

Márquez Canizales Cristian:

En esta segunda parte del proyecto final me tocó cumplir con el papel del PM, fue una tarea que no me imaginé que iba a llevar una mayor crocentacion que el rol pasado que me tocó realizar; ya que tuve que acompañar y aprender de los demás integrantes sobre lo que estaban realizando en su rol para así yo poder llevar registro y observar la creación de los nuevos elementos para esta fase del proyecto.

Alexia Gómez Rubio:

Mi aportación para esta segunda fase fue realizar la documentación, la cual implicó realizar una investigación y una revisión de los códigos en Verilog y Python. Sin duda alguna en esta fase pude adquirir un conocimiento tanto teórico como práctico, porque aunque no estuve tan involucrada con los códigos, si los tuve que revisar detalladamente para poder hacer sus respectivos reportes y de igual forma el entender cómo es que mis compañeros trabajan.

A lo largo de estas dos fases hemos trabajado de manera colaborativa, y desde mi punto de vista eso es lo que nos ha funcionado porque entre todos nos apoyamos para poder concluir con las asignaciones de la semana y lograr los objetivos.

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

Compile - Glosario 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>

Isaac. (2023, April 6). *Jim Keller ya está diseñando un nuevo procesador en Tenstorrent.*

Profesional Review; Miguel Ángel Navas.

<https://www.profesionalreview.com/2023/04/06/jim-keller-tenstorrent/>

Wikipedia contributors. (2024, February 22). *Raja Koduri*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Raja_Koduri&oldid=1209494921

Estas Palabras se Usan Para Referirse a las dos Formas en Que se Pueden Guardar Los

Números Que Ocupan Más, de un B. (n.d.). *El formato “Big Endian” y el “Little Endian.”*

Uaa.Mx. Retrieved May 6, 2024, from

https://docentes.uaa.mx/guido/wp-content/uploads/sites/2/2014/10/BE_LE.pdf

Tipos de Arquitecturas. (2016, August 9). Arquitectura de Computadoras.

<https://is603arquicom2016.wordpress.com/tipos-de-arquitecturas/>