Universidad de Guadalajara
Centro Universitario de Ciencias Exactas e Ingenierías

Materia:

Arquitectura de Computadoras **D12**

Prof. Jorge Ernesto Lopez Arce Delgado

Alumnos:

Reynoso Natareno Victoria Daniela
Cotero Muñoz Alejandra Quetzali
Estrada Rivera Gustavo de Jesús

Proyecto Final

MIPS Pipeline + Decodificador en Python

24/11/2025

Introducción

La arquitectura MIPS (Microprocessor without Interlocked Pipeline Stages) es una arquitectura RISC desarrollada a principios de los años ochenta por MIPS Technologies. MIPS utiliza instrucciones de tamaño fijo de 32 bits y organiza los datos fundamentalmente a través de 32 registros de propósito general. Estos registros tienen funciones convencionales asignadas: el registro zero contiene siempre el valor 0; los registros de valor de retorno son v0 y v1; los registros de argumentos a0 hasta a3 se usan para pasar parámetros; los temporales t0 a t9 sirven para cálculos intermedios; los registros s0 a s7 se utilizan para almacenar valores que deben conservarse entre llamadas.

Las instrucciones de MIPS se organizan en tres tipos básicos. El **formato R** se usa para operaciones aritméticas y lógicas que operan exclusivamente sobre registros. El **formato I** se usa para instrucciones que necesitan un valor inmediato de 16 bits o para operaciones de carga y almacenamiento hacia la memoria. Finalmente, el **formato J** se usa para saltos largos. Esta clasificación simple reduce la complejidad del hardware, evita decodificaciones complicadas y facilita la implementación de un pipeline.

MIPS introdujo el pipeline de 5 etapas, clave para su eficiencia:

1. IF – Instruction Fetch
2. ID – Instruction Decode
3. EX – Execute
4. MEM – Memory Access
5. WB – Write Back

MIPS se caracteriza por un conjunto reducido de modos de direccionamiento. El más empleado es el modo base más desplazamiento, que combina un registro base con un desplazamiento de 16 bits para acceder a memoria. Para operaciones inmediatas se usa un valor de 16 bits incrustado en la instrucción. Para saltos condicionales se utiliza direccionamiento relativo al contador de programa. La simplicidad de estos modos es clave para que las instrucciones sean uniformes y fáciles de decodificar.

Algoritmo que utilizaremos para esta ocasión: ***“Cambio de monedas”*** (Coin Change greedy)

Objetivo general

Desarrollar un sistema completo compuesto por un pipeline MIPS de cinco etapas implementado en Verilog y un decodificador de instrucciones en Python, capaz de traducir un conjunto extendido de instrucciones MIPS32 a su representación binaria para la correcta ejecución de un programa ensamblador no trivial dentro del procesador diseñado.

Objetivos particulares

- **Diseñar** un decodificador en Python (GUI) que valide la sintaxis de instrucciones MIPS y traduzca mnemónicos al formato binario MIPS32 de 32 bits.
- **Implementar** un procesador MIPS con pipeline de 5 etapas (IF, ID, EX, MEM, WB) utilizando los módulos obligatorios: PC, memorias, ALU, unidad de control, ALU control, banco de registros, multiplexores, etc,
- **Integrar** los buffers inter-etapas (IF/ID, ID/EX, EX/MEM, MEM/WB) asegurando el flujo correcto de datos e instrucciones.
- **Validar** la operación del pipeline con las instrucciones soportadas y verificar la correcta interacción entre hardware y el binario generado por el decodificador.
- **Desarrollar** un programa en ensamblador MIPS no trivial que utilice instrucciones aritméticas, lógicas, de memoria y control de flujo, incluyendo al menos un salto incondicional (J).
- **Probar** el programa dentro del pipeline implementado verificando su correcta ejecución paso a paso y el manejo adecuado de los recursos del sistema.
- **Integrar** el decodificador y el pipeline asegurando que el binario generado sea interpretado correctamente por la memoria de instrucciones.

Desarrollo

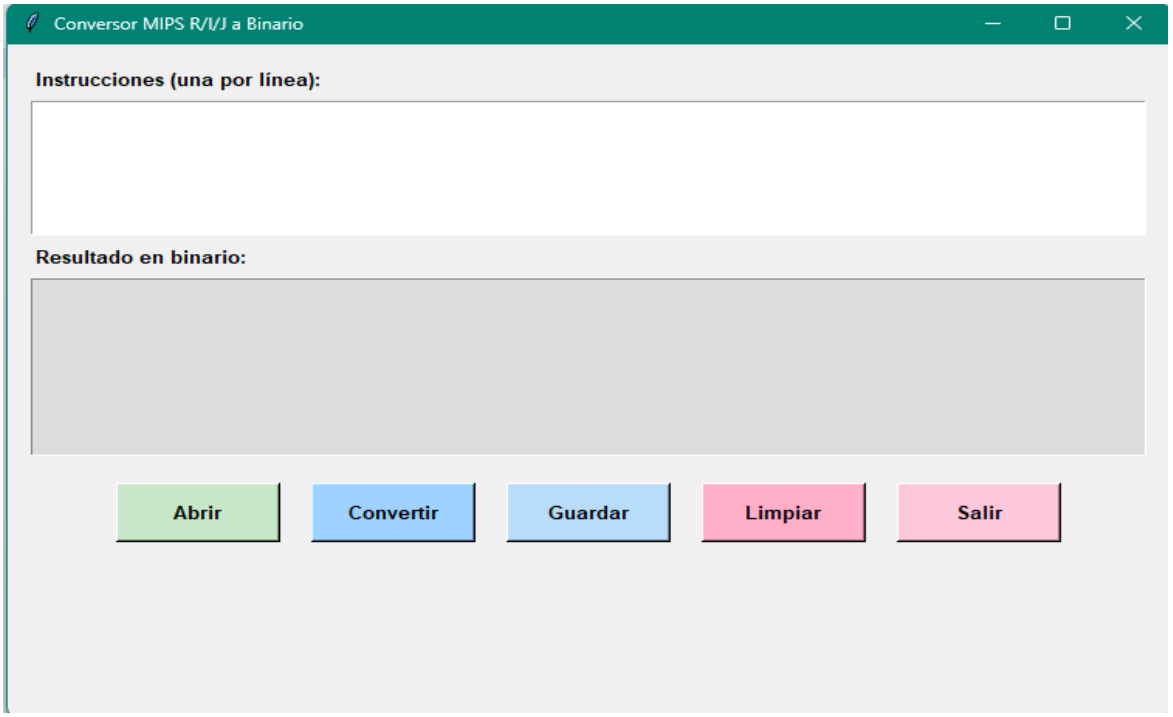
Decodificador de instrucciones en Python

Para poder evitar convertir las instrucciones en binario a mano, se decidió construir un decodificador en lenguaje **python** que pueda “traducir” nuestras instrucciones de un formato ensamblador a un formato binario siguiendo la recomendación de que el orden de bytes resultantes después de la conversión sea en el que el byte más significativo se almacena en la dirección de memoria más baja (**Big endian**) y que la separación sea de cada 8 bits, todo esto por medio de una interfaz gráfica que

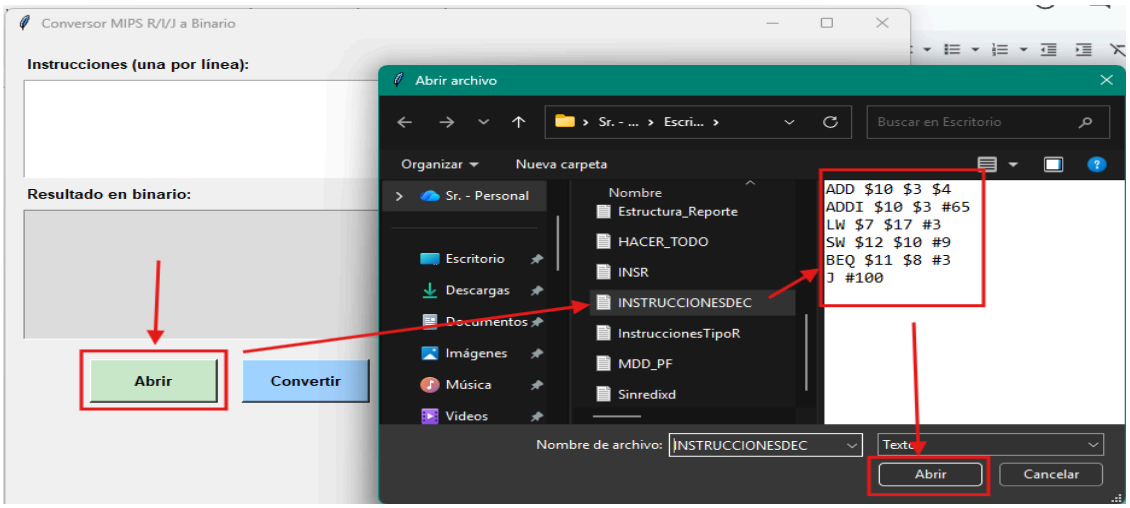
permita la entrada en tiempo real de los datos, permite incluso cargar un archivo de texto ya existente y que genere uno nuevo con las instrucciones ya convertidas a binario, con su separación y su almacenaje respete el formato **Big endian** como se establece.

A continuación se pasará a mostrar el decodificador y su funcionamiento.

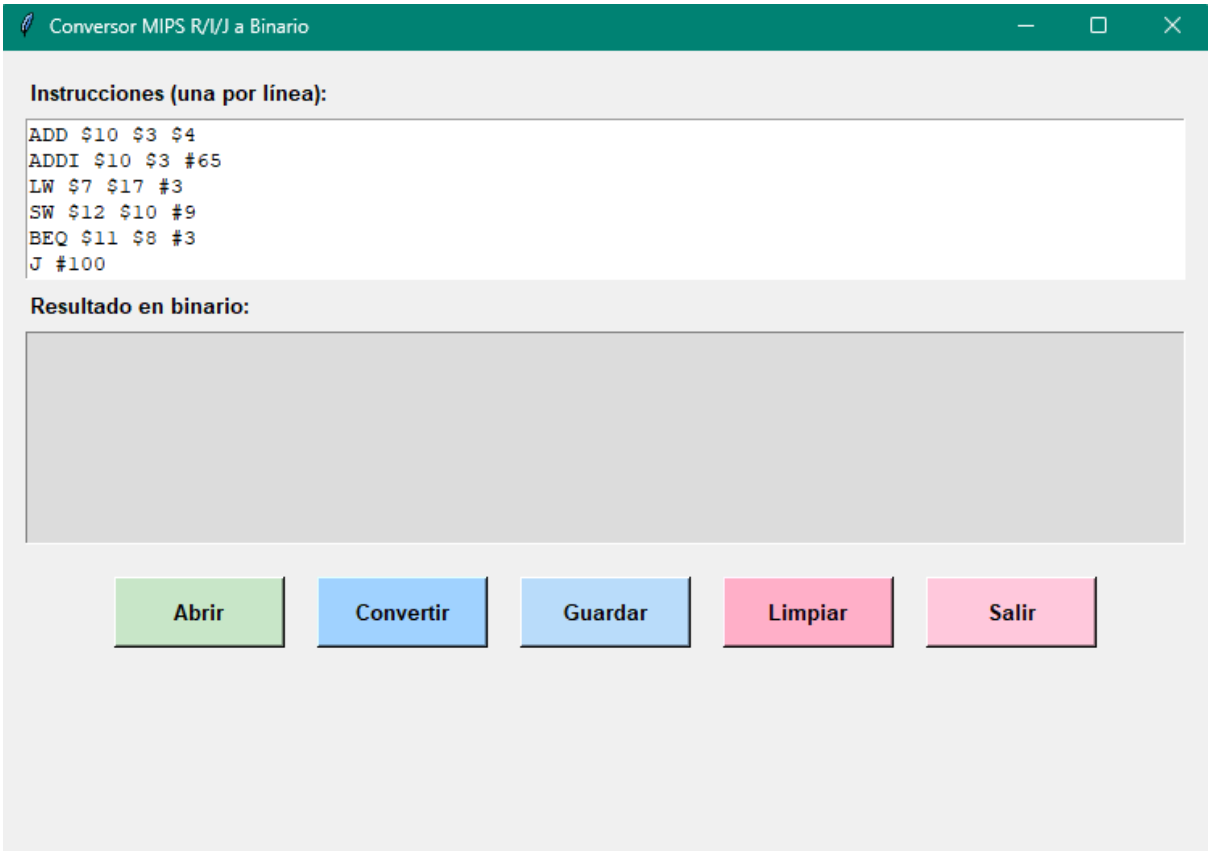
Interfaz:



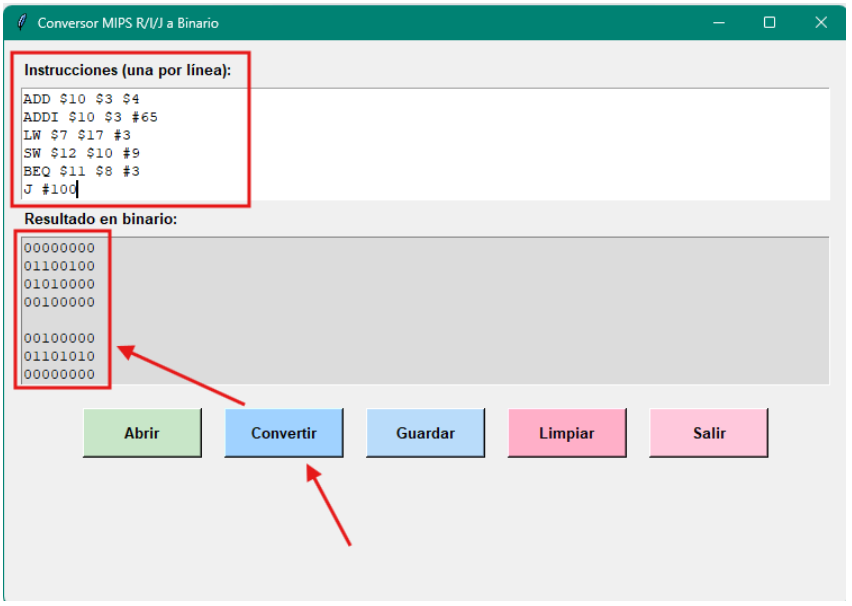
Importación de instrucciones (formato: inst $rd $rs $rt | inst $rt $rs #imm | inst $rt $rs #offset | inst #target):



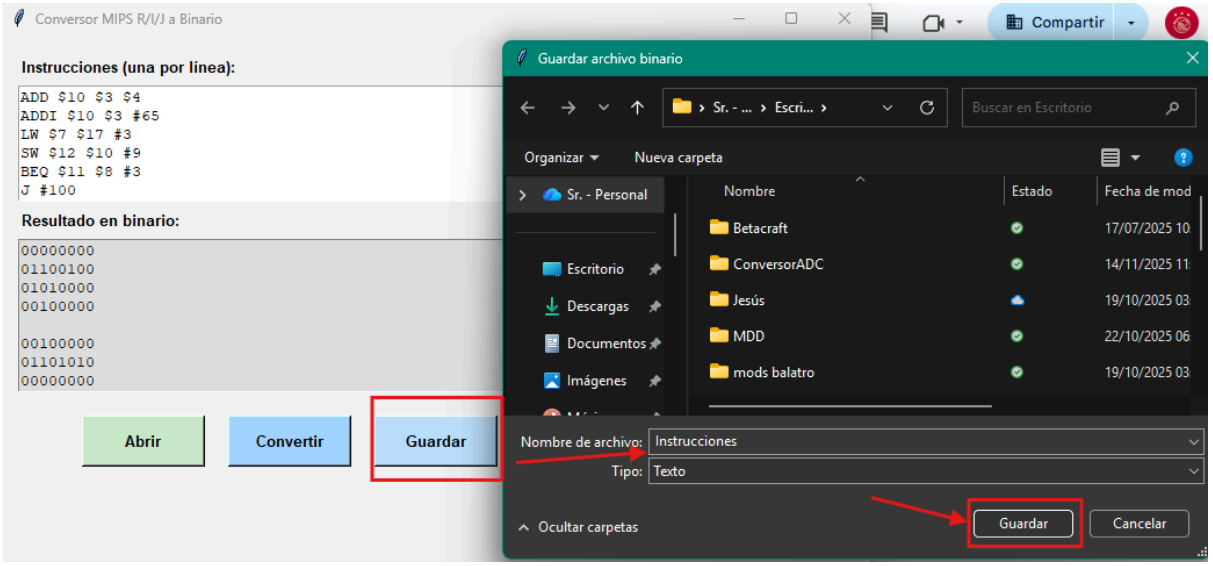
Visualización en el programa:



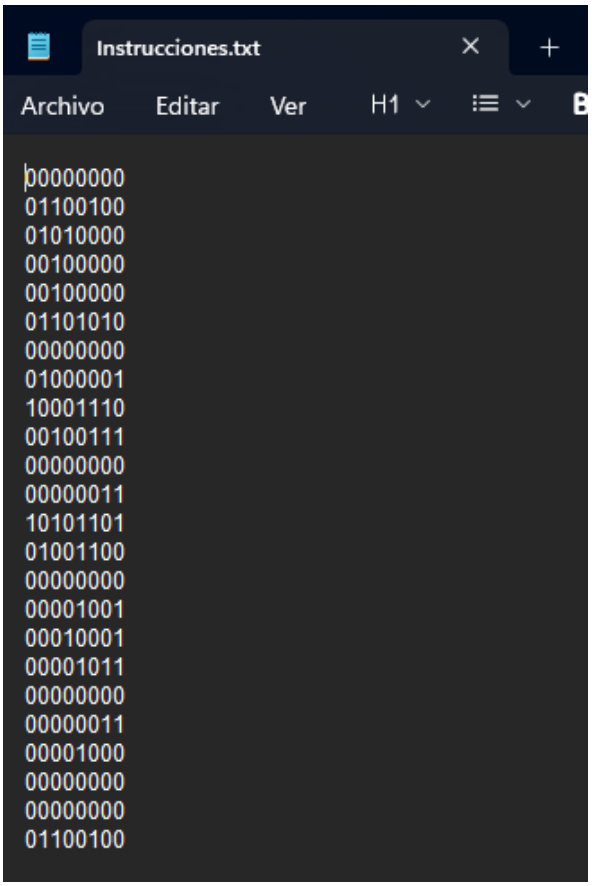
Conversión a binario:



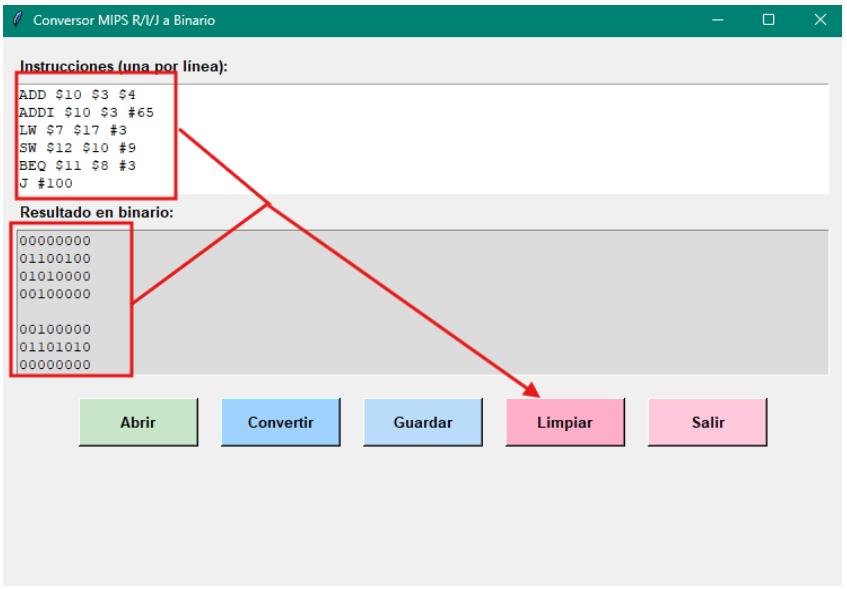
Exportación de las conversiones:

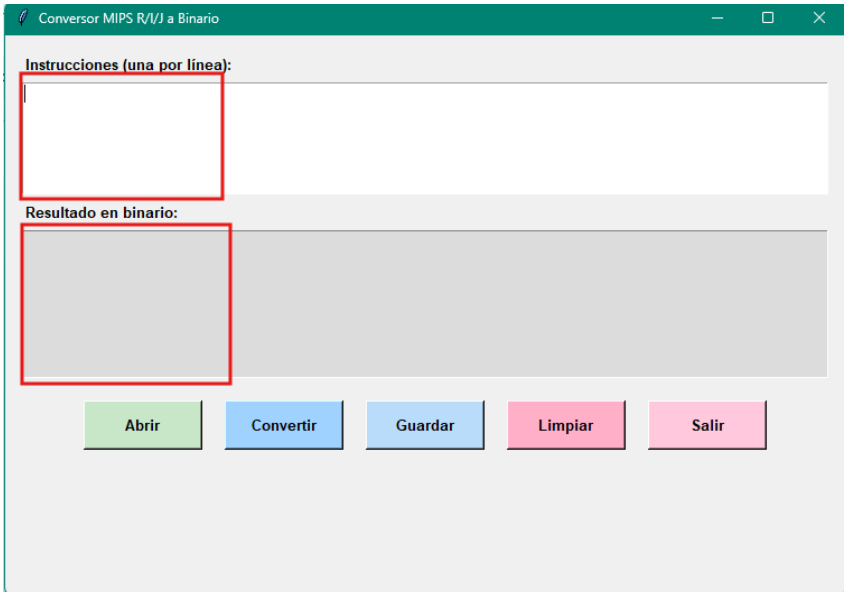


Archivo generado:

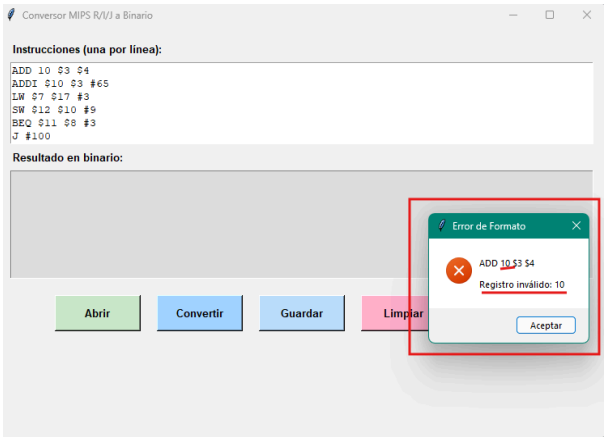


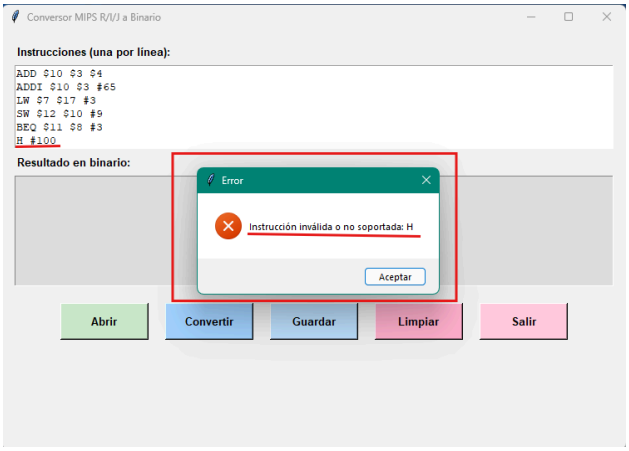
Limpieza de pantalla del programa:

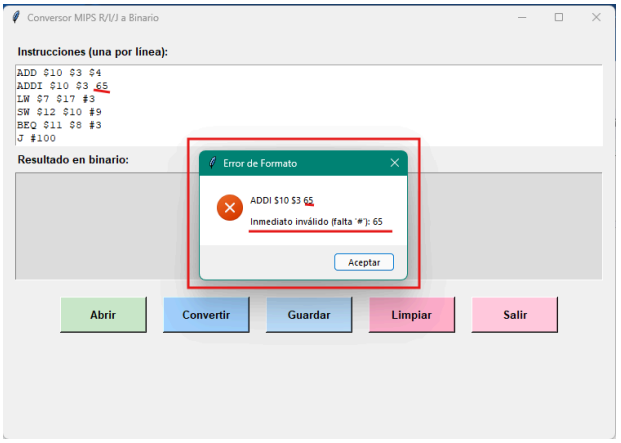


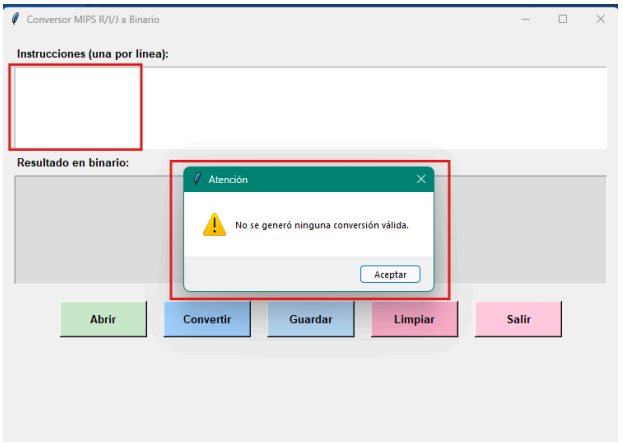


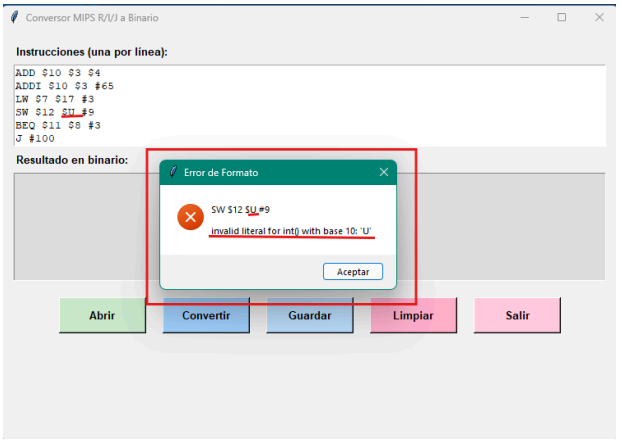
Manejo de errores (Validaciones):



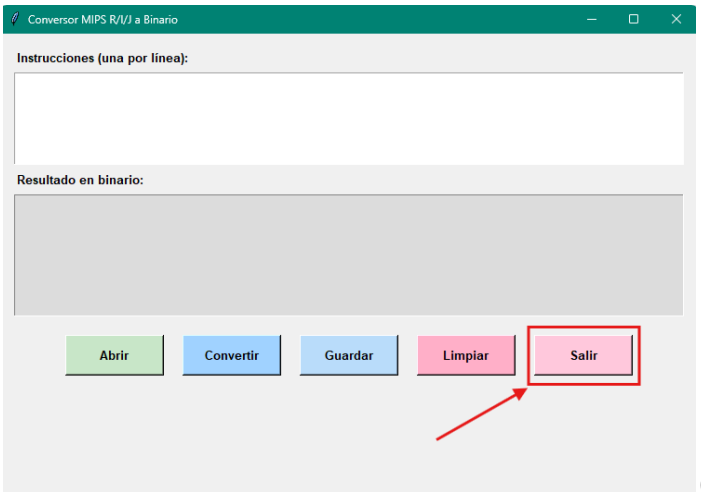

Conversor MIPS R/I/J a Binario
Instrucciones (una por línea):
ADD $10 $3 $4
ADDI $10 $3 #65
LW $7 $17 #3
SW $12 $10 #9
BEQ $11 $8 #3
H #100
Resultado en binario:
Error
Instrucción inválida o no soportada: H
Aceptar
Abrir
Convertir
Guardar
Limpiar
Salir


Conversor MIPS R/I/J a Binario
Instrucciones (una por línea):
ADD $10 $3 $4
ADDI $10 $3 65
LW $7 $17 #3
SW $12 $10 #9
BEQ $11 $8 #3
J #100
Resultado en binario:
Error de Formato
ADDI $10 $3 65
Inmediato inválido (falta '#'): 65
Aceptar
Abrir
Convertir
Guardar
Limpiar
Salir


Conversor MIPS R/I/J a Binario
Instrucciones (una por línea):
Resultado en binario:
Atención
No se generó ninguna conversión válida.
Aceptar
Abrir
Convertir
Guardar
Limpiar
Salir


Conversor MIPS R/I/J a Binario
Instrucciones (una por línea):
ADD $10 $3 $4
ADDI $10 $3 #65
LW $7 $17 #3
SW $12 $U #9
BEQ $11 $8 #3
J #100
Resultado en binario:
Error de Formato
SW $12 $U #9
invalid literal for int() with base 10: 'U'
Aceptar
Abrir
Convertir
Guardar
Limpiar
Salir

Finalización del programa:



(El programa se cierra instantáneamente)

El decodificador nos ayuda a tener una mejor eficiencia en la conversión de nuestras instrucciones a binario, evitando cometer errores comunes si lo hacemos a mano, nos ahorra tiempo y nos ayuda a generar nuestro archivo de instrucciones lista para trabajar.

Programa en Ensamblador

Para nuestro trabajo emplearemos el algoritmo del **"cambio de monedas"** (Coin Change greedy).

Justificación

Empleamos este algoritmo ya que:

- Es **voraz** ya que en cada paso se toma la moneda de mayor valor posible.
- No requiere **recursión** ni instrucciones **avanzadas** (funciona al pie de la letra con instrucciones que ya tenemos contempladas en nuestro procesador y solamente vistas en clase).
- Usa **loops** simples, *restas*, *AND* bitwise para detectar signos y beq/j para control de flujo.
- Nos permite demostrar nuestras instrucciones ya establecidas: **lw, sw, add, sub, and, beq y j**

Ideación

La idea del algoritmo es la siguiente:

Los **datos** en *.data*:

- **amount** — El monto a cambiar.
- **coins** — Un arreglo de monedas: 25, 10, 5, 1.

- **counts** — Un arreglo para guardar cuántas monedas de cada tipo usamos (inicializado en 0).
- **mask** — *0x80000000* (Lo usaremos para extraer el bit de signo tras una resta).
- **one** — Una constante 1 (para incrementar contadores).

En la **salida**:

- **counts** con el número de monedas por tipo y amount_rem con el residuo.

Como demostración del algoritmo se diseñó el siguiente pseudocódigo:

```
while amount >= coin:
    amount = amount - coin
    count[i] = count[i] + 1
store count[i]
siguiente moneda
```

La comprobación `amount >= coin` se hace así:

- tmp = amount - coin
- sign = tmp & *0x80000000*
- si sign == 0 => tmp >= 0 => podemos restar (*branch con beq*).

Implementación

El código en ensamblador es el siguiente:

```
ADDI $6 $0 #0       # $6 = poner 0 para offsets
ADDI $5 $0 #0       # $5 = total_monedas = 0
ADDI $1 $0 #0       # $1 = índice i = 0
LW   $3 $6 #5      # $3 = N = 47
LOOP:
LW   $2 $6 #0       # $2 = denominación actual (son de 20,10,5,2,1) inicia
en 20, accede por instrucción, no por palabra
ADDI $4 $0 #0       # $4 monedas_denom = 0
WHILE:
SLT  $8 $2 $3       # $8 = (denominación <= N)? coin < amount?:
BEQ  $8 $0 NEXT     # Si no, salir a next
SUB  $3 $3 $2       # N -= denominación, restamos la denominación al monto
ADDI $4 $4 #1       # monedas_denom++, aumentamos el contador de la moneda
usada
J    WHILE          # Repetir while hasta que salga
NEXT:
ADD  $5 $5 $4       # total += monedas_denom
ADDI $1 $1 #1       # índice++
ADDI $9 $0 #5       # límite = 5
SLT  $8 $1 $9       # $8 = (índice < 5)?
```

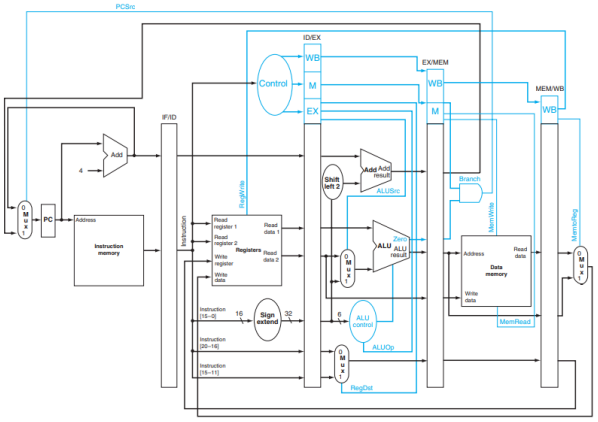
```
BEQ  $8 $0 END      # Si terminó, salir
J    LOOP           # Siguiente denom
END

# $5 = resultado esperado, 4 monedas para un monto de 47
```

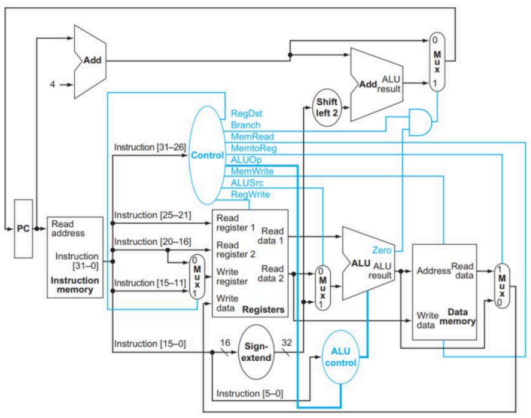
Diseño Procesador MIPS

Para llevar esto a cabo, se usaron los siguientes módulos divididos en un pipeline de 5 etapas con 4 buffers que conectan entre ellas y pasan información. El contenedor de todos estos módulos y que se encarga de conectarlos es mips32 instanciado como procesador y el cuál recibe las señales de reloj, pasandolas a los pipelines y módulos que necesitan actualizarse en tiempo real :

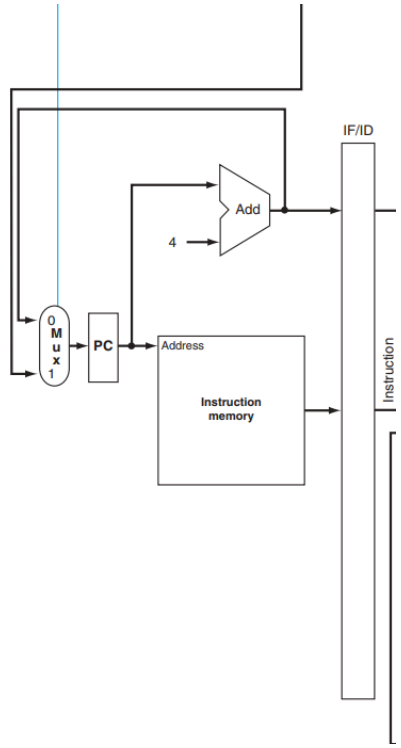
adder.v	✓	Verilog 0	11/30/2025 08:43:18 ...
alu.v	✓	Verilog 1	11/30/2025 08:45:07 ...
alu_control.v	✓	Verilog 2	11/30/2025 08:57:15 ...
alur_brach.v	✓	Verilog 3	11/14/2025 10:56:07 ...
AND.v	✓	Verilog 4	11/30/2025 10:05:45 ...
br.v	✓	Verilog 5	11/29/2025 03:59:40 ...
brMUX.v	✓	Verilog 6	11/14/2025 10:56:25 ...
control_unit.v	✓	Verilog 7	11/30/2025 08:50:35 ...
Datos.txt		Text -	11/29/2025 07:07:46 ...
ex.v	✓	Verilog 8	11/28/2025 12:50:49 ...
ex_mem.v	✓	Verilog 9	11/30/2025 10:21:22 ...
id.v	✓	Verilog 10	11/27/2025 02:43:34 ...
id_ex.v	✓	Verilog 11	11/29/2025 11:47:00 ...
if.v	✓	Verilog 12	11/29/2025 04:04:58 ...
if_id.v	✓	Verilog 13	11/29/2025 11:46:50 ...
instrucciones.txt		Text -	11/30/2025 08:54:12 ...
mem.v	✓	Verilog 14	11/27/2025 01:29:46 ...
mem_inst.v	✓	Verilog 25	11/14/2025 10:56:42 ...
mem_wb.v	✓	Verilog 15	11/29/2025 11:43:08 ...
memoria_datos.v	✓	Verilog 16	11/30/2025 01:07:28 ...
mips.v	✓	Verilog 17	11/30/2025 07:57:31 ...
mips_tb.v	✓	Verilog 18	11/30/2025 08:21:00 ...
muxes.v	✓	Verilog 19	11/28/2025 11:02:05 ...
pc.v	✓	Verilog 20	11/14/2025 11:23:25 ...
sign_extend.v	✓	Verilog 21	11/25/2025 06:56:11 ...
sl2_branch.v	✓	Verilog 22	11/14/2025 10:57:04 ...
sl2_jump.v	✓	Verilog 23	11/14/2025 10:57:08 ...
wb.v	✓	Verilog 24	11/28/2025 01:01:10 ...

El traspaso de información se lleva a cabo de esta manera, donde se pasa la información por cada pipeline register entre ciclos de reloj. La información del salto requiere un multiplexor adicional después de la verificación del multiplexor de la branch.



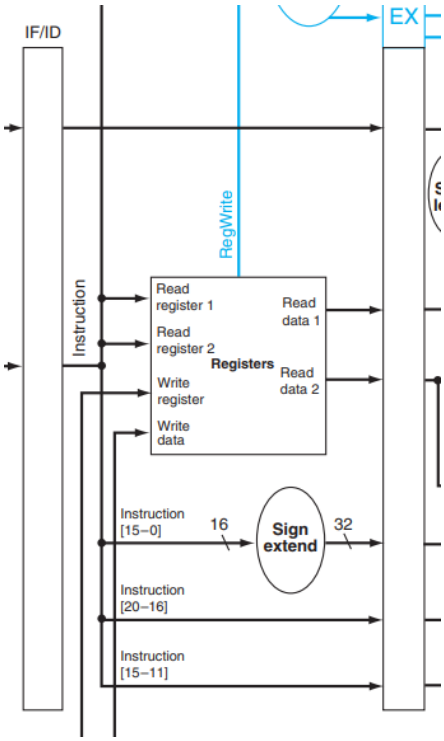


Etapa IF (Instruction Fetch)



La primera etapa se trata de IF (instruction fetch) donde el pc inicialmente recibe un valor de 0 para comenzar la siguiente instrucción y con ayuda del adder, calcula la siguiente dirección, la cual no será usada inmediatamente, ya que se necesitará para la etapa de EX. Para que esta información que generamos no se quede inaccesible en la etapa anterior (ya que no se maneja con registros), creamos el pipeline register IF/ID, el cual funciona como un buffer que traspasa la información (registrada en nuestros módulos con extensión _IN -en su mayoria de inputs-) entre ciclos. En nuestro módulo, nuestras entradas y salidas son: *input clk, reset, input branch input [31:0] next, input [31:0] PCmux_A, input [31:0] PCmux_B,output [31:0] InstQ, output [31:0] prox_dir, output [31:0] muxPC_salida.* Donde prox_dir e InstQ salen hacia el pipeline que sí guarda los registros en ese ciclo. Posteriormente la información será pasada por medio de cables correspondientes a la siguiente etapa ID.

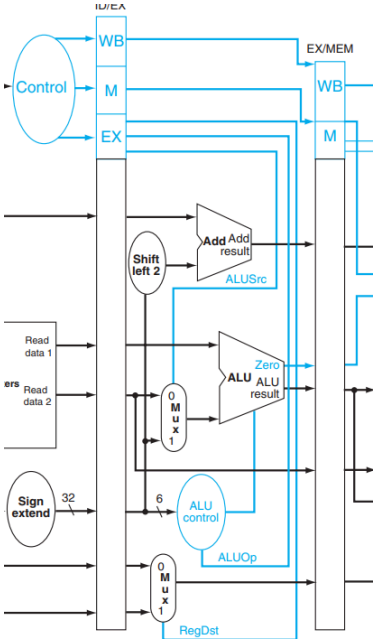
Etapa ID (Instruction decode)



Una vez que la información se pasa desde IF/ID gracias a los _OUTS, la etapa de ejecución consta del banco de registros donde la instrucción se toma desde C_InstQ y cada entrada tanto de BR como de Sign Extend recibe la información que necesita. Aquí es cuando la información que siempre pasa directamente es Read Register 1 (RD1 en nuestro caso) y para RT y RD, la instrucción se divide del 20 al 16 y del 15 al 11, la cual será pasada al próximo pipeline register donde se encuentra su multiplexor encargado de elegir cuál se usa según el tipo de instrucción. Sign Extend realiza sus operaciones independientemente de si se va a utilizar o no y pasa la información al pipeline. Siguiente. Es importante aclarar que RegWrite todavía no recibe señal aquí. La información de DR1 y DR2, que son la información

contenida en el registro dado, salen al siguiente pipeline register ID/EX.

Etapa EX (Execution)

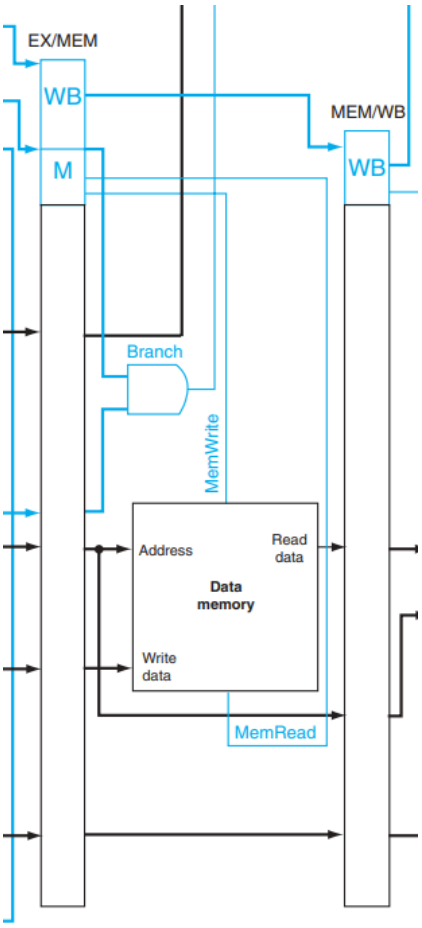


En la etapa de ejecución es donde se realizan cálculos importantes de los cuales depende si la instrucción se realiza correctamente o no, esta se encuentra entre los pipeline registres ID/EX y EX/MEM. Primeramente, es donde la ALU realiza la operación correspondiente, todo esto gracias a la alu_control, la cual, recibiendo el FUNC code desde sign extend y el aluOP desde control, verifica que tipo de instrucción es y la clasifica en un tipo: R, I , J y su subtipo correspondiente. Después, esta información se pasa hacia la alu, la cual toma la salida de alu control y decide qué tipo de operación aritmética hacer. Para que esta instrucción se lleve a cabo correctamente, la ALU debe recibir los valores correctos, por lo cual DR1 nunca cambia y suele ser el registro donde guardamos operaciones de suma, resta, y lw, por ejemplo. Para el registro 2, el cual se usa para instrucciones y también puede usarse para instrucciones beq, el multiplexor de la entrada inferior de la ALU recibe una señal de control ALUSrc, la cual actúa como selector de una de las dos entradas.

Una de las partes más importantes a resaltar de esta etapa, y en concreto la etapa de ID/EX que es la que recibe las señales de control, lo cual es vital para que todas las entradas que requieran un selector puedan trabajar con valores actualizados que vuelven verdadero/falso la toma de decisiones para llevar a cabo las instrucciones. En esta etapa la mayoría de señales de control se guardan en la entrada y salen por el buffer de salida. Resaltamos que AluSrc, AluOP y RegDst se quedan en la etapa de EX porque son usadas inmediatamente, así que estas no pasarán al siguiente buffer de entrada de EX/MEM.

Etapa MEM (Memory Access)

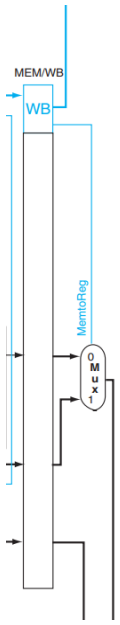
MEM se encuentra entre EX/MEM para entradas y MEM/WB para información que necesita ser traspasada al buffer de salida, consta de una memoria de datos donde se realizan lecturas y escrituras con información clave para nuestro algoritmo, tal como lo son los valores 20, 10, 5, 2 y 1, que son las denominaciones de las monedas, el siguiente índice 5 se usa para guardar el monto ya sea precargado o insertado desde el algoritmo desde un sw. El acceso a esta información para load word se realiza desde instrucciones (es decir, no es byte adressable) y es vital que el primer registro $0 contenga siempre un valor de 0, o en otro caso, el cual es el que usamos, escribimos en el banco de registros este valor que será usado para realizar los offsets de manera correspondiente. Para sw, su acceso si se realiza con un offset de multiplicación de índice por 4, es decir, para acceder a MEM[5], debemos usar un registro con valor de 0 y un offset de #20 (ej: sw $2 $0 #20, guarda la información que hay en el banco de registros $2 en MEM[5]).



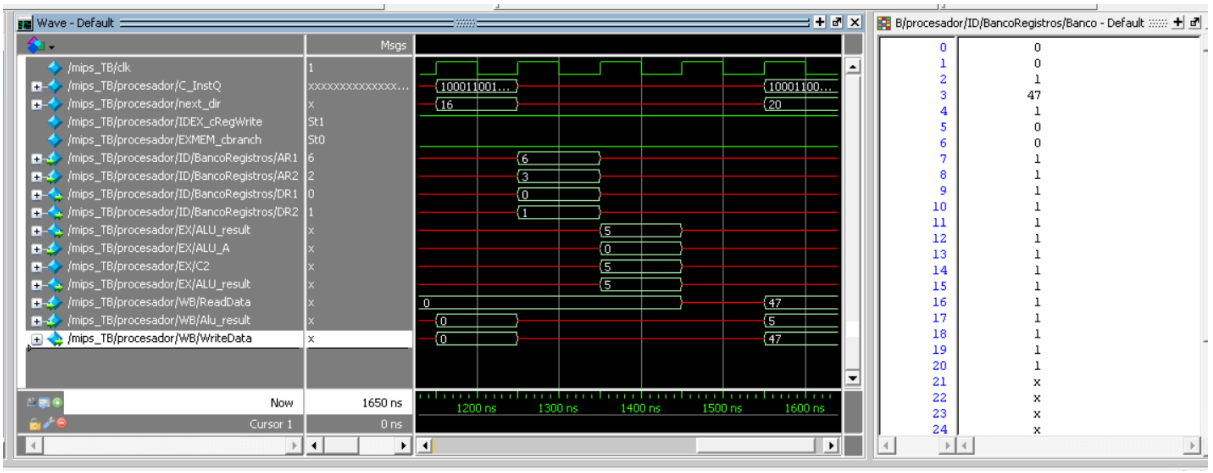
Gracias al buffer de salida de EX/MEM podemos conservar el valor de salida de la ALU, el cual siempre pasa al siguiente pipeline register (posteriormente al multiplexor entrada 0) y es usado o para saber qué información es la que se va a escribir o es para usarla como dirección de índice para la memoria de datos MEM. La información que llega a write data es la traspasada desde EX, con el valor leído del registro 2 (DR2), el cual se usa para la escritura de datos.

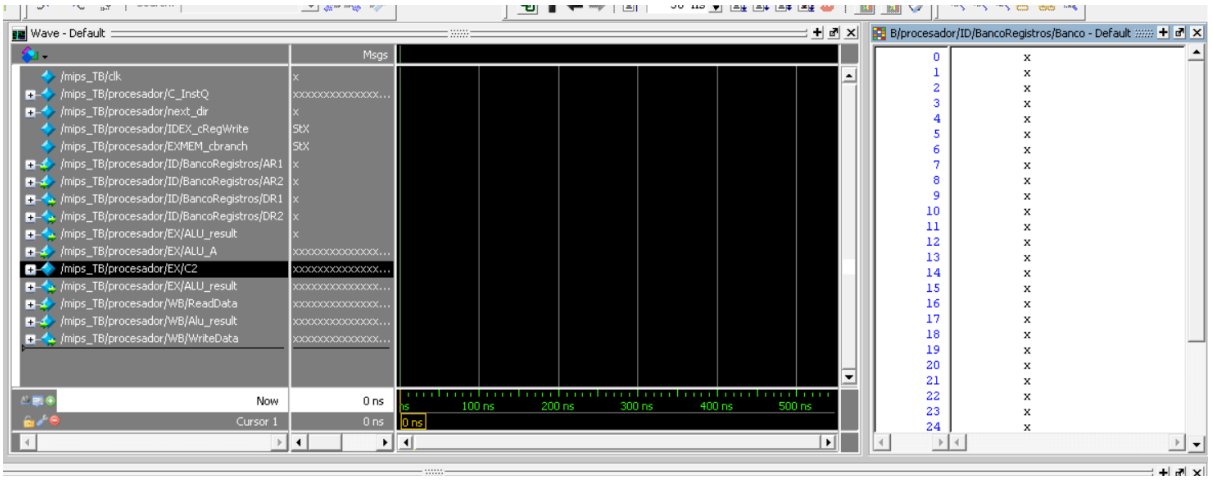
En estas etapas es donde también se usa MemWrite y MemRead como selectores para lw y sw, por lo que solo restan dos señales de control para usar en la siguiente etapa.

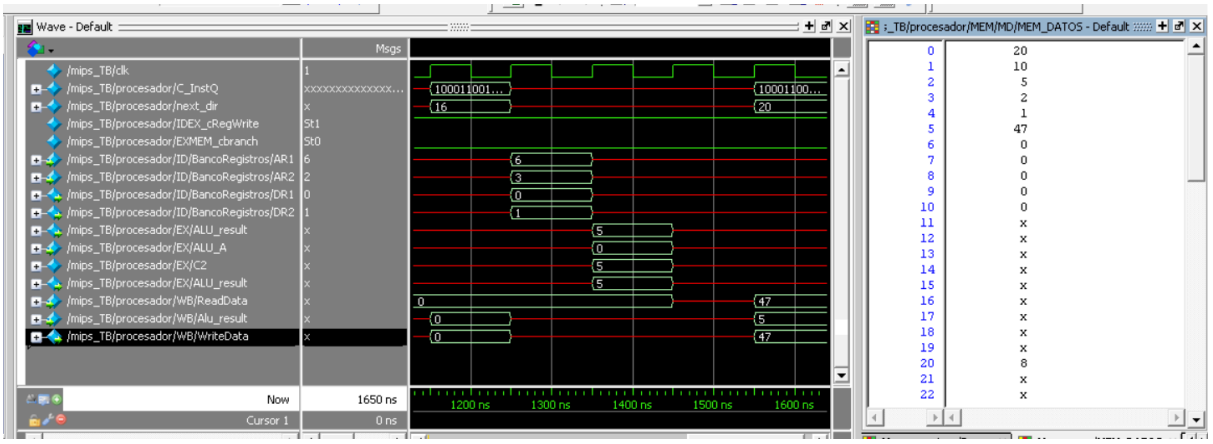
Etapa WB (Write Back) y EX/MEM

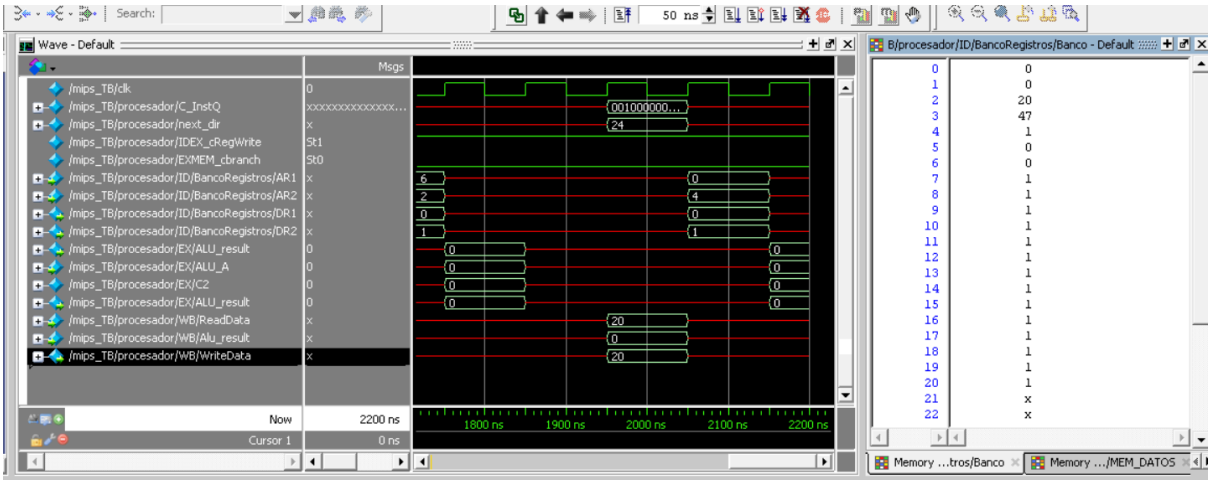


WB consiste simplemente en un multiplexor que elige cual salida usar para escribir en el banco de registros: la salida de la ALU o la read data, que será para guardar un lw, en nuestro caso, las denominaciones de las monedas. Si la instrucción es por ejemplo una add o addi para sumar un índice de cuenta, memtoReg debe tomar el valor de 0 para guardar la información de la ALU en el banco(cabe resaltar que la imagen está mal, pero es para fines ilustrativos de ambos pipelines). Es en la etapa posterior a WB, EX/MEM, donde se toma por fin el valor de RegWrite de forma temprana para saber si vamos a leer o no. Al mismo tiempo, la información que hemos traspasado desde el multiplexor de RT y RD, por fin es devuelta para saber en donde vamos a escribir.






Wave - Default
Msgs
/mips_TB/clk
/mips_TB/procesador/C_InstQ
/mips_TB/procesador/next_dir
/mips_TB/procesador/IDEX_cRegWrite
/mips_TB/procesador/EXMEM_cbranch
/mips_TB/procesador/ID/BancoRegistros/AR1
/mips_TB/procesador/ID/BancoRegistros/AR2
/mips_TB/procesador/ID/BancoRegistros/DR1
/mips_TB/procesador/ID/BancoRegistros/DR2
/mips_TB/procesador/EX/ALU_result
/mips_TB/procesador/EX/ALU_A
/mips_TB/procesador/EX/C2
/mips_TB/procesador/EX/ALU_result
/mips_TB/procesador/WB/ReadData
/mips_TB/procesador/WB/Alu_result
/mips_TB/procesador/WB/WriteData
Now
1650 ns
Cursor 1
0 ns
1200 ns
1300 ns
1400 ns
1500 ns
1600 ns
i_TB/procesador/MEM/MD/MEM_DATOS - Default


Search:
50 ns
Wave - Default
Msgs
/mips_TB/clk
/mips_TB/procesador/C_InstQ
/mips_TB/procesador/next_dir
/mips_TB/procesador/IDEX_cRegWrite
/mips_TB/procesador/EXMEM_cbranch
/mips_TB/procesador/ID/BancoRegistros/AR1
/mips_TB/procesador/ID/BancoRegistros/AR2
/mips_TB/procesador/ID/BancoRegistros/DR1
/mips_TB/procesador/ID/BancoRegistros/DR2
/mips_TB/procesador/EX/ALU_result
/mips_TB/procesador/EX/ALU_A
/mips_TB/procesador/EX/C2
/mips_TB/procesador/EX/ALU_result
/mips_TB/procesador/WB/ReadData
/mips_TB/procesador/WB/Alu_result
/mips_TB/procesador/WB/WriteData
Now
2200 ns
Cursor 1
0 ns
1800 ns
1900 ns
2000 ns
2100 ns
2200 ns
B/procesador/ID/BancoRegistros/Banco - Default
Memory ...tros/Banco
Memory .../MEM_DATOS

Conclusiones

En conclusión, este proyecto nos permitió entender mejor cómo funciona un procesador MIPS, tanto por dentro como por fuera. El diseño del pipeline de cinco etapas en Verilog nos ayudó a ver cómo se procesan las instrucciones paso a paso y cómo interactúan módulos como la ALU, las memorias y los registros.

Además, el decodificador en Python permitió convertir instrucciones MIPS a su forma binaria de manera clara y ordenada, facilitando la carga del programa en la memoria del procesador.

El uso del algoritmo *Coin Change* en ensamblador nos ayudó a comprobar el funcionamiento real del pipeline, utilizando operaciones aritméticas, comparaciones, saltos y acceso a memoria. Gracias a esto pudimos validar que el diseño del procesador y el decodificador funcionaban correctamente.

En general, este proyecto refuerza nuestra comprensión del funcionamiento de un CPU, combinando programación, diseño digital y ensamblador en un solo trabajo práctico. Y finalmente, podemos dar por concluido nuestro proyecto.

Referencias

- MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual: Vol. IV-j (6.06). (2016).
- Performance improvement in MIPS pipeline processor based on FPGA. (2016). *International Journal Of Engineering Technology, Management And Applied Sciences*, *4*. https://www.researchgate.net/profile/Kirat-Singh-2/publication/290429304_Performance_Improvement_in_MIPS_Pipeline_Processor_based_on_FPGA/links/5ce2d47792851c4eabb155f7/Performance-Improvement-in-MIPS-Pipeline-Processor-based-on-FPGA.pdf
- GeeksforGeeks. (2025, October 6). *Minimum number of Coins*. GeeksforGeeks. https://www.geeksforgeeks.org/dsa/greedy-algorithm-to-find-minimum-number-of-coins/
- WiserTheBassist. (2018, May 8). *MIPS pipeline registers length (IF/ID, ID/EX, EX/MEM, MEM/WB)*. Stack Overflow. https://stackoverflow.com/questions/50242300/mips-pipeline-registers-length-if-id-id-ex-ex-mem-mem-wb
- Organization of computer systems: pipelining. (n.d.). https://www.cise.ufl.edu/~mssz/CompOrg/CDA-pipe.html
- Parthasarathi, R. (n.d.). Handling Data Hazards – Computer architecture. https://www.cs.umd.edu/~meesh/411/CA-online/chapter/handling-data-hazards/index.html
- Parthasarathi, R. (n.d.). Pipelining – MIPS implementation – Computer architecture. https://www.cs.umd.edu/~meesh/411/CA-online/chapter/pipelining-mips-implementation/index.html
- Trini, S. (2021, October 25). MIPS: pipeline. la35.net. https://la35.net/orga/mips-pipeline.html