UNIVERSIDAD DE GUADALAJARA

CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERIAS

División de Tecnologías para la Integración Ciber-Humana

Departamento de Ciencias Computacionales

Ingeniería Informática

Actividad

Proyecto Final

PRESENTA

Nombre del alumno(a)s:

López Hernández Emiliano Juan

Huerta Romo Adolfo

Materia: Arquitectura de computadoras, 2025B

Profesor(a):

Lopez Arce Delgado Jorge Ernesto

Introducción

Este proyecto implementa un procesador MIPS Pipeline de 5 etapas en Verilog, siguiendo la arquitectura estándar MIPS32. El diseño incluye todas las etapas del pipeline (IF, ID, EX, MEM, WB) y soporta un subconjunto completo de instrucciones MIPS, incluyendo operaciones aritméticas, lógicas, acceso a memoria y control de flujo.

La implementación se basa en la especificación oficial "MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual" (Document MD00086, Revisión 6.06), garantizando compatibilidad con el estándar MIPS32. El procesador fue diseñado considerando los principios fundamentales de arquitectura de computadoras, optimizando el uso de recursos y manejando eficientemente los hazards estructurales, de datos y de control inherentes a los diseños pipeline.

Como parte integral del proyecto, se desarrolló un decodificador en Python que traduce código assembly MIPS a lenguaje máquina, permitiendo cargar programas directamente en la memoria de instrucciones del procesador. Esta herramienta facilita la verificación y prueba del sistema completo, asegurando que cada instrucción se ejecute correctamente según el estándar MIPS32.

El diseño final representa una implementación educativa pero completamente funcional de un procesador pipeline moderno, demostrando los conceptos teóricos de arquitectura de computadoras en un sistema práctico y verificable.

Objetivos general y particulares

Objetivo General

Desarrollar e implementar un procesador MIPS Pipeline completamente funcional que ejecute programas en lenguaje ensamblador MIPS, demostrando el funcionamiento del pipeline de 5 etapas y resolviendo los hazards de datos y control mediante técnicas estándar. El procesador debe cumplir con la especificación MIPS32 y ser capaz de ejecutar un conjunto diverso de instrucciones de manera eficiente y correcta.

Objetivos Particulares

- Diseñar e implementar todos los módulos del datapath MIPS pipeline, incluyendo PC, memorias de instrucciones y datos, banco de registros, ALU y unidades de control.
- Desarrollar la unidad de control principal para decodificar instrucciones MIPS32 y generar las señales de control apropiadas para cada tipo de instrucción.
- Implementar el banco de registros con treinta y dos registros de treinta y dos bits, con capacidad de lectura combinacional y escritura síncrona, incluyendo forwarding interno para evitar hazards.
- Crear la ALU con operaciones aritméticas y lógicas completas (ADD, SUB, AND, OR, XOR, SLT) siguiendo los códigos de control estándar MIPS.
- Diseñar las memorias de instrucciones (ROM) y datos (RAM) con capacidad de doscientas cincuenta y seis palabras de treinta y dos bits cada una, con interfaces adecuadas para el pipeline.
- Implementar los buffers de pipeline entre etapas (IF/ID, ID/EX, EX/MEM, MEM/WB) para almacenar y propagar señales y datos.
- Desarrollar un decodificador en Python que traduzca código assembly MIPS a lenguaje máquina binario, generando el archivo de carga para la memoria de instrucciones.
- Verificar el funcionamiento completo del procesador mediante simulaciones exhaustivas en Verilog, incluyendo testbenches para cada módulo y para el sistema completo.

- Implementar mecanismos de manejo de hazards, incluyendo detección de data hazards y control hazards, con técnicas de forwarding y flushing cuando sea necesario.
- Documentar todo el diseño, implementación y resultados obtenidos, siguiendo los estándares institucionales para proyectos de arquitectura de computadoras.

Desarrollo

Arquitectura del Pipeline
El procesador implementa las cinco etapas clásicas del pipeline MIPS, diseñadas para ejecutar instrucciones de forma superpuesta y mejorar el rendimiento del sistema:

Etapa IF (Instruction Fetch)
- Lectura de la instrucción desde la memoria de instrucciones usando el valor del Program Counter (PC)
- Cálculo de PC+4 para la siguiente instrucción secuencial
- Almacenamiento temporal en el buffer IF/ID

Etapa ID (Instruction Decode)
- Decodificación del opcode y campos de la instrucción
- Lectura de dos registros del banco de registros
- Extensión de signo del campo inmediato de dieciséis a treinta y dos bits
- Generación de señales de control por la unidad principal

Etapa EX (Execute)
- Ejecución de operaciones aritméticas y lógicas en la ALU
- Cálculo de direcciones de memoria para loads/stores
- Determinación de direcciones de branch
- Selección del registro destino (rt o rd)

Etapa MEM (Memory Access)
- Acceso a memoria de datos para instrucciones load/store
- Escritura de datos en memoria para instrucciones store
- Verificación de condiciones de branch

Etapa WB (Write Back)
- Escritura de resultados en el banco de registros
- Selección entre resultado de ALU o dato de memoria

Módulos Implementados
- Módulos Principales del Datapath
- PC (Program Counter)
- Registro de treinta y dos bits que almacena la dirección de la instrucción actual
- Actualización síncrona en flanco positivo de reloj
- Soporte para reset asíncrono y enable para stalls

Memoria de Instrucciones
Memoria ROM de doscientas cincuenta y seis palabras de treinta y dos bits
Direccionamiento por palabra (ignora dos bits menos significativos)
Carga inicial desde archivo "instrucciones.txt"

Memoria de Datos
- Memoria RAM de doscientas cincuenta y seis palabras de treinta y dos bits
- Lectura combinacional y escritura síncrona

- Soporte para instrucciones LW y SW

Banco de Registros
- Treinta y dos registros de treinta y dos bits cada uno
- Dos puertos de lectura combinacional
- Un puerto de escritura síncrono en flanco negativo
- Registro $0 hardcodeado a cero
- Forwarding interno para evitar hazards de datos

ALU (Arithmetic Logic Unit)
- Operaciones implementadas: AND, OR, ADD, XOR, SUB, SLT
- Entradas de treinta y dos bits, salida de treinta y dos bits
- Bandera Zero para instrucciones de branch

Unidad de Control Principal
- Decodificación del opcode (bits treinta y uno-veintiseis)
- Generación de nueve señales de control
- Soporte para instrucciones R-type, I-type y J-type

ALU Control
- Interpretación del campo funct para instrucciones R-type
- Generación de códigos de control específicos para la ALU
- Integración con señales ALUOp de la unidad principal

Módulos de Soporte
- Sign Extend
- Extensión de signo de valores inmediatos de dieciséis a treinta y dos bits
- Replicación del bit de signo (bit quince) en bits dieciseis-treinta y uno

Shift Left 2
- Desplazamiento izquierda de dos bits (multiplicación por cuatro)
- Usado para cálculo de direcciones de branch y jump

Multiplexores
- Mux 2:1 de cinco bits para selección de registro destino
- Mux 2:1 de treinta y dos bits para ALUSrc y MemtoReg
- Mux 3:1 de treinta y dos bits para selección de próximo PC
- Mux 4:1 de treinta y dos bits para forwarding de datos

Buffers de Pipeline
- IF/ID: Almacena PC+4 e instrucción
- ID/EX: Almacena señales de control y datos de registros
- EX/MEM: Almacena resultados de ALU y señales de memoria
- MEM/WB: Almacena datos para write back

Conjunto de Instrucciones Implementado

Instrucciones Tipo R (Register)
Formato: opcode(seis) | rs(cinco) | rt(cinco) | rd(cinco) | shamt(cinco) | funct(seis)

- ADD: rd = rs + rt (suma de registros)

- SUB: rd = rs - rt (resta de registros)
- AND: rd = rs & rt (AND lógico)
- OR: rd = rs | rt (OR lógico)
- SLT: rd = (rs < rt) ? 1 : 0 (comparación con signo)

Instrucciones Tipo I (Immediate)
Formato: opcode(seis) | rs(cinco) | rt(cinco) | immediate(dieciséis)

- ADDI: rt = rs + sign_extend(immediate)
- ANDI: rt = rs & zero_extend(immediate)
- ORI: rt = rs | zero_extend(immediate)
- XORI: rt = rs ^ zero_extend(immediate)
- SLTI: rt = (rs < sign_extend(immediate)) ? 1 : 0
- LW: rt = memory[rs + sign_extend(immediate)]
- SW: memory[rs + sign_extend(immediate)] = rt
- BEQ: if (rs == rt) PC = PC + 4 + (immediate << 2)

Instrucciones Tipo J (Jump)
Formato: opcode(seis) | address(veintiseis)

- J: PC = {PC[treinta y uno:veintiocho], address, 2'b00}

Señales de Control

La unidad de control genera las siguientes señales basándose en el opcode:

RegDst - Selección de registro destino:
- Cero: rt (instrucciones I-type)
- Uno: rd (instrucciones R-type)

ALUSrc - Fuente del operando B de la ALU:
- Cero: Read Data 2 (banco de registros)
- Uno: Sign-extended immediate

MemtoReg - Fuente del dato para write back:
- Cero: ALU Result
- Uno: Read Data (memoria)

RegWrite - Habilitación de escritura en banco de registros

MemRead - Habilitación de lectura de memoria de datos

MemWrite - Habilitación de escritura en memoria de datos

Branch - Indicación de instrucción de branch condicional

ALUOp - Código de operación para ALU Control:

- Cero cero: LW, SW, ADDI (ADD)
- Cero uno: BEQ (SUB)
- Uno cero: R-type (usa campo funct)
- Uno uno: SLTI (SLT)

Jump - Indicación de instrucción de salto incondicional

Manejo de Hazards

Hazards Estructurales

- Memorias separadas para instrucciones y datos
- Escritura en banco de registros en flanco negativo del reloj
- Múltiples puertos de lectura en banco de registros

Hazards de Datos

- Forwarding Interno en Banco de Registros:
- Detección de escritura y lectura al mismo registro en el mismo ciclo
- Entrega del dato que se está escribiendo directamente a la salida de lectura

Stalls para Load-Use Hazards:

- Detección de instrucción LW seguida de instrucción que usa el mismo registro
- Inserción de burbuja (stall) por un ciclo
- Congelamiento del PC y buffer IF/ID

Hazards de Control

Manejo de Branches:

- Predicción estática: siempre se asume que el branch no se toma
- Si el branch se toma, se realiza flush del buffer IF/ID
- Cálculo de dirección de branch en etapa EX

Manejo de Jumps:

- Flush inmediato del buffer IF/ID
- Cálculo de dirección de jump en etapa ID

Verificación del Sistema Completo

- Ejecución de programas de prueba con diferentes patrones de instrucciones
- Verificación de manejo correcto
- Validación de resultados finales en registros y memoria
- Comprobación de comportamiento en condiciones límite

Ejecución:

Instrucciones:

00100000000010000000000000000101
00100000000010010000000000000011
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000010000100101010000000100000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
10101100000010100000000000000000
00001000000000000000000000001011

FORMATO TIPO R (Register)

op	rs	rt	rd	sh	fn
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
Opcode	Source 1	Source 2	Dest	Shift	Function

Instrucción: ADD $t2, $t0, $t1

Código hexadecimal: `01095020`

Campo	Bits	Binario	Decimal	Significado
op	31-26	000000	0	Tipo R
rs	25-21	01000	8	$t0 (fuente 1)
rt	20-16	01001	9	$t1 (fuente 2)
rd	15-11	01010	10	$t2 (destino)
sh	10-6	00000	0	No se usa
fn	5-0	100000	32	Función ADD

Binario completo: `000000 01000 01001 01010 00000 100000`

FORMATO TIPO I (Immediate)

op	rs	rt	operand / offset
6 bits	5 bits	5 bits	16 bits
Opcode	Source	Dest	Immediate operand or address offset

Instrucción: ADDI $t0, $zero, 5

Código hexadecimal: `20080005`

Campo	Bits	Binario	Decimal	Significado
op	31-26	001000	8	ADDI
rs	25-21	00000	0	$zero (fuente)
rt	20-16	01000	8	$t0 (destino)
imm	15-0	0000000000000101	5	Inmediato = 5

Binario completo: `001000 00000 01000 0000000000000101`

Instrucción: ADDI $t1, $zero, 3

Código hexadecimal: `20090003`

Campo	Bits	Binario	Decimal	Significado
op	31-26	001000	8	ADDI
rs	25-21	00000	0	$zero (fuente)
rt	20-16	01001	9	$t1 (destino)
imm	15-0	0000000000000011	3	Inmediato = 3

Binario completo: `001000 00000 01001 0000000000000011`

Instrucción: SW $t2, 0($zero)

Código hexadecimal: `AC0A0000`

Campo	Bits	Binario	Decimal	Significado
op	31-26	101011	43	SW
rs	25-21	00000	0	$zero (base)
rt	20-16	01010	10	$t2 (dato)
imm	15-0	0000000000000000	0	Offset = 0

Binario completo: `101011 00000 01010 0000000000000000`

FORMATO TIPO J (Jump)

op	jump target address

6 bits	26 bits
Opcode	Memory word address (byte address / 4)

Instrucción: J 11

Código hexadecimal: `0800000B`

Campo	Bits	Binario	Decimal	Significado
op	31-26	000010	2	J (Jump)
addr	25-0	00000000000000000000001011	11	Dirección

Binario completo: `000010 00000000000000000000001011`

Que pasa, paso por paso:

Ciclo 1: Fetch de ADDI $t0, $zero, 5

IF	ID	EX	MEM	WB
ADDI $t0, 5	-	-	-	-

Señales: PC_current = 0x00, Instruction_IF = 0x20080005

Ciclo 2: ADDI avanza a ID

IF	ID	EX	MEM	WB
ADDI $t1, 3	ADDI $t0, 5	-	-	-

Señales: PC = 0x04, RegWrite_ID = 1, ALUSrc_ID = 1, ALUOp = 00

Ciclo 3: Primer ADDI en EX

IF	ID	EX	MEM	WB
NOP	ADDI $t1, 3	ADDI $t0, 5	-	-

Señales: ALUResult = 0x00000005 (0 + 5 = 5)

Ciclo 5: Primer ADDI completa en WB

IF	ID	EX	MEM	WB
NOP	NOP	NOP	ADDI $t1	ADDI $t0

Señales: WB_ALUResult = 0x00000005, WB_WriteReg = 0x08 ($t0)
Resultado: ¡$t0 ahora contiene 5!

Ciclo 6: Segundo ADDI completa en WB

IF	ID	EX	MEM	WB
NOP	NOP	NOP	NOP	ADDI $t1

Señales: WB_ALUResult = 0x00000003, WB_WriteReg = 0x09 ($t1)
Resultado: ¡$t1 ahora contiene 3!

Ciclo 10: ADD en EX - ALU calcula 5 + 3 = 8

IF	ID	EX	MEM	WB
NOP	NOP	ADD $t2	NOP	NOP

Señales: EX_ReadData1 = 0x05, EX_ReadData2 = 0x03, EX_ALUOp = 2 (R-type)
Resultado: ALUResult = 0x00000008 (5 + 3 = 8)

Ciclo 12: ADD completa en WB

IF	ID	EX	MEM	WB
SW	NOP	NOP	NOP	ADD $t2

Señales: WB_ALUResult = 0x00000008, WB_WriteReg = 0x0A ($t2)
Resultado: ¡$t2 ahora contiene 8 (5+3)!

Ciclo 15: SW en MEM - Escribe en memoria

IF	ID	EX	MEM	WB
J	NOP	NOP	SW	NOP

Señales: MEM_MemWrite = 1, MEM_ALUResult = 0x00 (dirección)
Resultado: ¡El valor 8 se escribe en memoria[0]!

Ciclo 17: J (Jump) en ID

Señales: Jump_ID = 1, RegWrite_ID = 0
Resultado: El PC se prepara para saltar a la dirección destino

3. Tabla Resumen de Señales

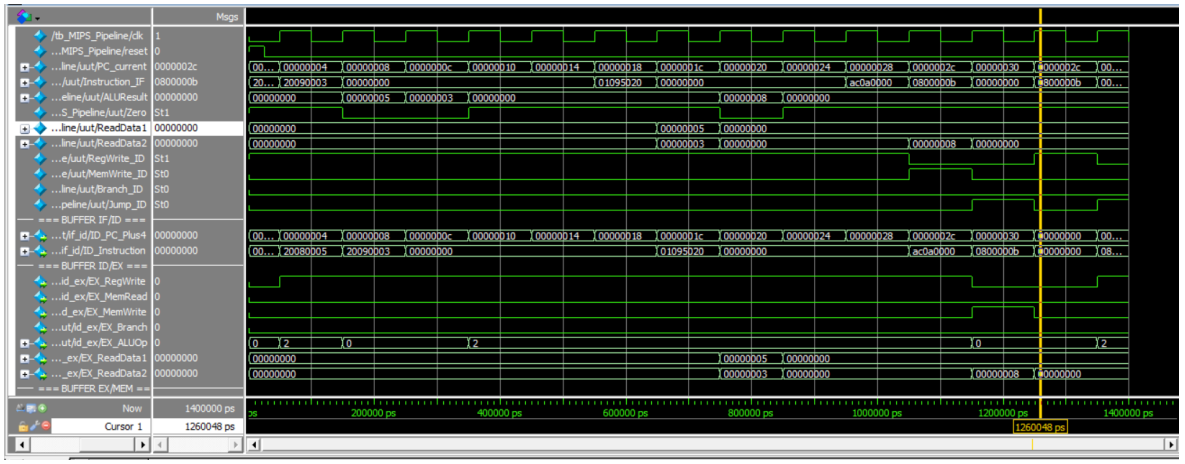
Instrucción	Tipo	RegWrite	MemWrite	Jump	ALUOp	ALUResult
ADDI $t0, $zero, 5	I	1	0	0	00	0x05
ADDI $t1, $zero, 3	I	1	0	0	00	0x03
ADD $t2, $t0, $t1	R	1	0	0	10	0x08
SW $t2, 0($zero)	I	0	1	0	00	0x00
J 11	J	0	0	1	--	--

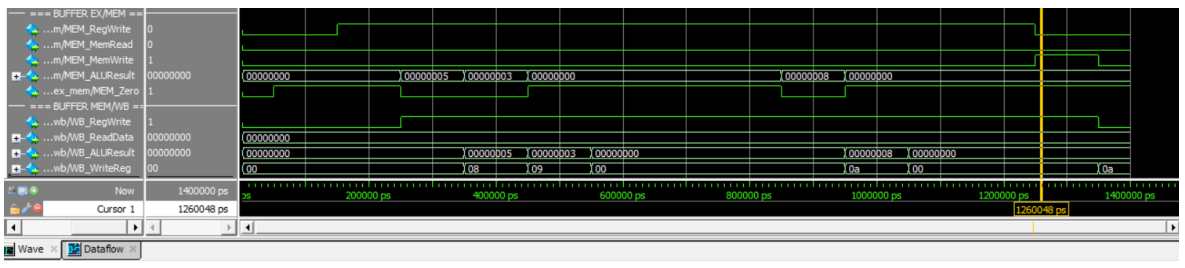
Conclusion:

TABLA DE REGISTROS UTILIZADOS

Nombre	Número	Binario	Uso en el programa
$zero	0	`00000`	Fuente (siempre vale 0)
$t0	8	`01000`	Guarda el valor 5
$t1	9	`01001`	Guarda el valor 3
$t2	10	`01010`	Guarda el resultado 8 (5+3)

Capturas:





Decodificador Python

Características Principales

- Lectura de archivos .asm con código MIPS
- Soporte para todos los formatos de instrucción implementados
- Generación de archivo "instrucciones.txt" en formato binario
- Validación de sintaxis y semántica de instrucciones

Instrucciones en código ensamblador:

```
ADDI $t0, $zero, 5
ADDI $t1, $zero, 3
NOP
NOP
NOP
NOP
ADD  $t2, $t0, $t1
NOP
NOP
NOP
SW   $t2, 0($zero)
J    11
```

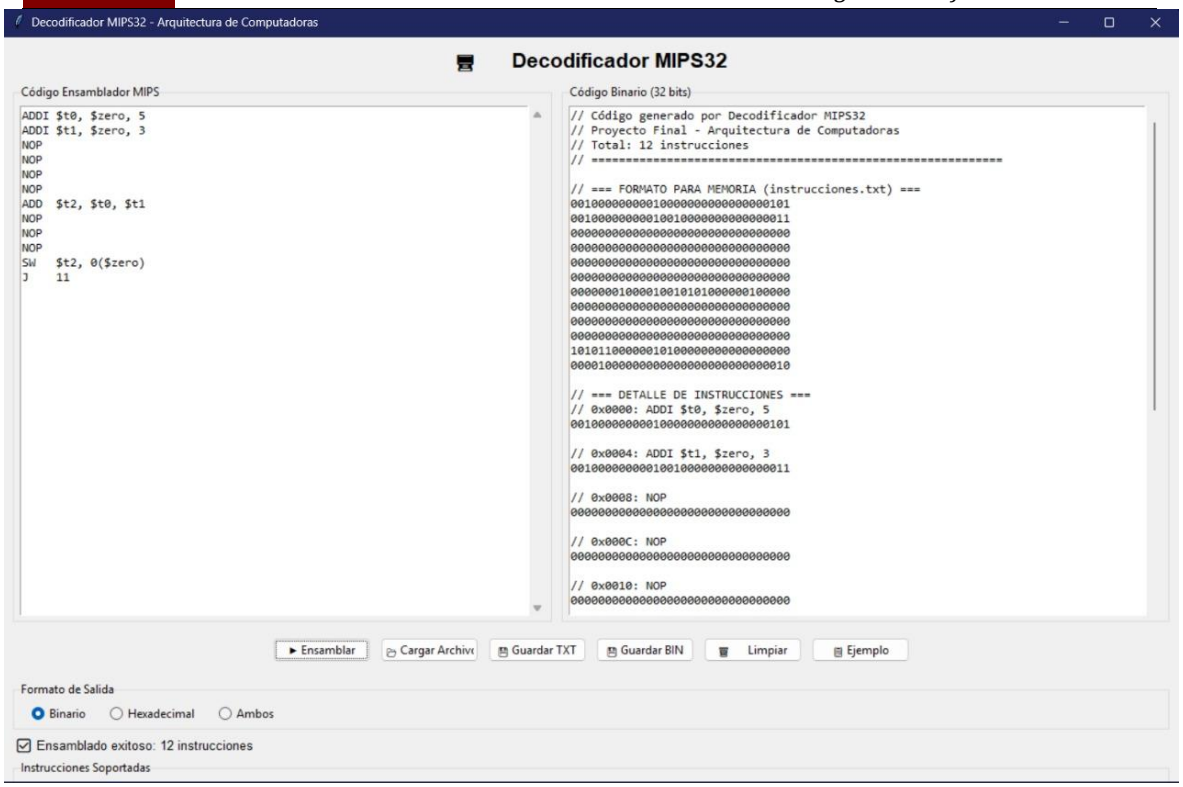
Funcionamiento

- Análisis léxico y sintáctico de cada línea de código
- Identificación del tipo de instrucción (R, I, J)
- Codificación de campos según formato MIPS32
- Generación de representación binaria de treinta y dos bits
- Escritura en archivo de salida para carga en memoria

Instrucciones Soportadas

- Todas las instrucciones R-type: ADD, SUB, AND, OR, SLT
- Instrucciones I-type: ADDI, ANDI, ORI, XORI, SLTI, LW, SW, BEQ
- Instrucciones J-type: J
- Directivas básicas para etiquetas y comentarios

Captura(Programa adjuntado en el repositorio github):



Conclusiones

Se implementó exitosamente un procesador MIPS Pipeline completamente funcional con cinco etapas que cumple con las especificaciones del estándar MIPS32. El diseño logra ejecutar correctamente un conjunto diverso de instrucciones incluyendo operaciones aritméticas, lógicas, acceso a memoria y control de flujo.

El sistema demostró capacidad para manejar eficientemente los hazards estructurales mediante la separación de memorias de instrucciones y datos, así como la implementación de escritura en registros en flanco negativo del reloj. Los hazards de datos se resolvieron satisfactoriamente mediante técnicas de forwarding interno en el banco de registros y stalls para casos de load-use.

El decodificador desarrollado en Python demostró ser una herramienta efectiva para la traducción de código assembly a lenguaje máquina, generando archivos binarios compatibles con la memoria de instrucciones del procesador. Esta herramienta facilitó significativamente las pruebas y verificación del sistema completo.

Referencias

- Trini, S. (2020, November 12). MIPS: datapath. la35.net. https://la35.net/orga/mips-datapath.html
- Trini, S. (2021, October 25). MIPS: pipeline. la35.net. https://la35.net/orga/mips-pipeline.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