UNIVERSIDAD DE GUADALAJARA

CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

Departamento de Innovación Basada en la Información y el Conocimiento

INGENIERÍA EN COMPUTACIÓN

ARQUITECTURA DE COMPUTADORAS D03

Alumno: Rodrigo Sanchez Rivera y Andrés Ortiz Guizar

Profesor: Jorge Ernesto López Arce Delgado

Guadalajara, Jalisco, 30/11/25

PIPELINE MIPS DE 5 ETAPAS Y DECODIFICADOR DE INSTRUCCIONES MIPS A BINARIO.

Introducción

Durante el transcurso de este semestre hemos observado el proceso que hacen las computadoras al ejecutar instrucciones. En las últimas semanas nos mostró el concepto de pipeline de instrucciones, una técnica que nos permite la ejecución simultanea de múltiples instrucciones en diferentes tapas de procesamiento. Específicamente el procesador MIPS, un modelo para entender el cómo es este proceso para gente nueva en el tema y comprender lo básico del proceso. En este proyecto mostrare la implementación de un pipeline MIPS de 5 etapas y una herramienta software de decodificación que planeamos en clase para traducir las instrucciones en ensamblador a binario, para poder usar esas instrucciones en nuestro pipeline.

Objetivo General

Desarrollar un sistema integrado compuesto por un pipeline MIPS de 5 etapas implementado en Verilog y un decodificador de instrucciones en Python con interfaz gráfica, capaz de traducir, cargar y ejecutar programas no triviales que utilicen operaciones aritméticas, lógicas, acceso a memoria y control de flujo.

Objetivos Particulares

Objetivos del Decodificador (Python)

- Implementar una interfaz gráfica intuitiva usando tkinder
- Desarrollar un validador sintáctico para instrucciones MIPS
- Crear un traductor se assembly a binario MIPS32
- Generar archivos de salida en formato Big Endian
- Implementar manejo de errores y mensajes al usuario
- Soportar el conjunto completo de instrucciones requeridas

Objetivos del Pipeline (Verilog)

- Diseñar e implementar las 5 etapas del pipeline
- Desarrollar todos los módulos componentes requeridos
- Implementar buffers inter-etapas para sincronización
- Crear un testbench comprehensivo para verificación
- Garantizar la ejecución correcta de programas complejos

Objetivos de integración

- Establecer un formato de archivo común para comunicación
- Verificar la compatibilidad endianness entre componentes
- Demostrar la ejecución de algoritmos no triviales

- Documentar el flujo completo de desarrollo y ejecución

Desarrollo

1. Decodificador en Python

El decodificar implementado consta de dos componentes principales:

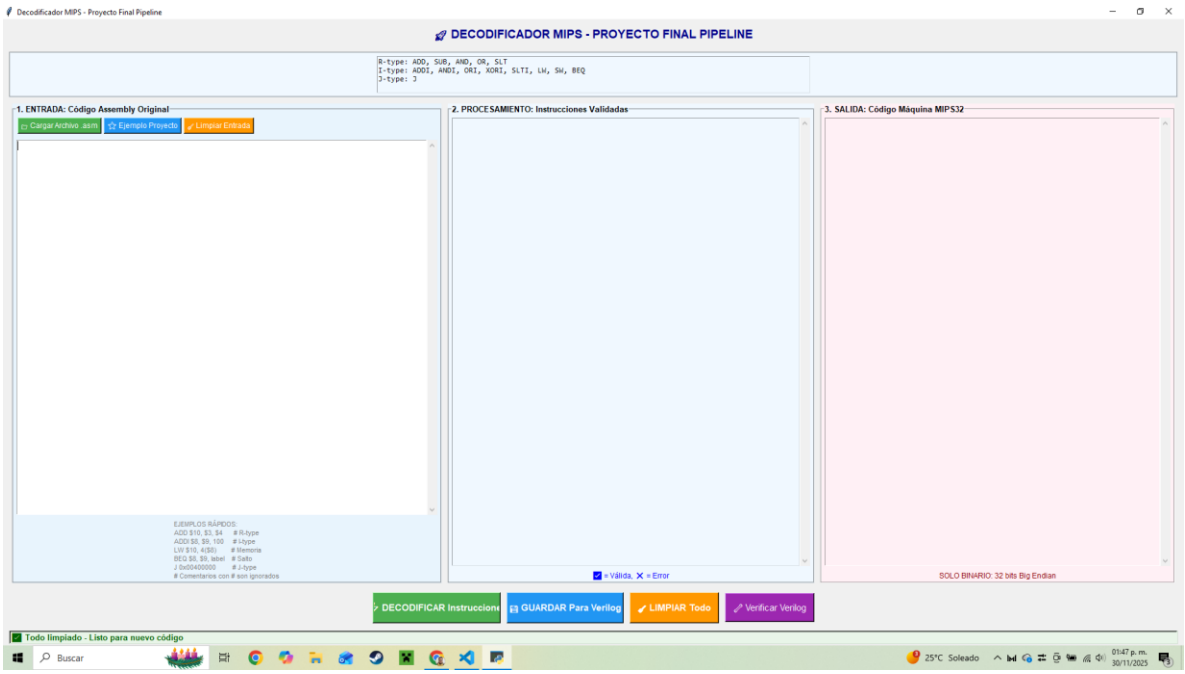
Main.py - Interfaz gráfica con tkinter

Decodificador.py - Motor de traducción y validación

1.1 Características de la interfaz Gráfica

Diseño de tres paneles:

- Panel 1: entrada de código assembly con soporte para carga de archivos
- Panel 2: Validación y procesamiento de instrucciones
- Panel 3: Salida en binario MIPS32



Funcionalidades Implementadas:

- Carga de archivos .asm y .txt
- Validación en tiempo real
- Ejemplos integrados de programas no triviales
- Generación de archivos listos para Verilog
- Mensajes de estado y manejo de errores

1.2 Motor de decodificación

```
INSTRUCCIONES_R = {
    'ADD': {'tipo': 'R', 'opcode': 0b000000, 'funct': 0b100000},  # 0x20
    'SUB': {'tipo': 'R', 'opcode': 0b000000, 'funct': 0b100010},  # 0x22
    'AND': {'tipo': 'R', 'opcode': 0b000000, 'funct': 0b100100},  # 0x24
    'OR':  {'tipo': 'R', 'opcode': 0b000000, 'funct': 0b100101},  # 0x25
    'SLT': {'tipo': 'R', 'opcode': 0b000000, 'funct': 0b101010},  # 0x2A
}

INSTRUCCIONES_I = {
    'ADDI': {'tipo': 'I', 'opcode': 0b001000},  # 0x08
    'ANDI': {'tipo': 'I', 'opcode': 0b001100},  # 0x0C
    'ORI':  {'tipo': 'I', 'opcode': 0b001101},  # 0x0D
    'XORI': {'tipo': 'I', 'opcode': 0b001110},  # 0x0E
    'SLTI': {'tipo': 'I', 'opcode': 0b001010},  # 0x0A
    'LW':   {'tipo': 'I', 'opcode': 0b100011},  # 0x23
    'SW':   {'tipo': 'I', 'opcode': 0b101011},  # 0x2B
    'BEQ':  {'tipo': 'I', 'opcode': 0b000100},  # 0x04
}

INSTRUCCIONES_J = {
    'J':    {'tipo': 'J', 'opcode': 0b000010},  # 0x02
}
```

Proceso de Decodificación (por lo que entiendo que hace cada sección del decodificador):

1. Limpieza y validación de líneas de código
2. Identificación de tipo de instrucción (R, I, J)
3. Chequeo de operandos según el formato
4. Generación de código binario de 32 bits
5. Conversión a Big Endian para compatibilidad

En el desarrollo de este decodificador la gran parte del desarrollo lo llevo mi compañero de equipo y el me mostro cómo funciona, pero el lenguaje de Python no se usarlo en aspectos técnicos muy avanzados como en este trabajo, se hacer un hola mundo, pero fuera de ahí no sé qué hacer, el profesor me dio chance de usar IA, pero fue en un trabajo anterior igualmente con el decodificador para instrucciones tipo R, pero ya para el proyecto Andrés dijo que me ayuda en esa parte y que yo me enfocara en el Pipeline, ya que en ese aspecto si tengo más fuerza que en Python, dejando la mitad y mitad del trabajo.

2. Pipeline MIPS en Verilog

El desarrollo de este Pipeline fue la continuación del todo el progreso de cada módulo que hemos hecho en clase y que fueron necesario para entender cada parte del modelo Pipeline MIPS, como es el ciclo Fetch, cómo funciona el banco de datos y como una instrucción se va moviendo dentro del ciclo.

Arquitectura del Pipeline

IF → [IF/ID] → ID → [ID/EX] → EX → [EX/MEM] → MEM → [MEM/WB] → WB

Componentes principales:

- PC.v: Contador del programa
- MemIns.v : Memoria de instrucciones cargada con el archivo instrucciones.txt generado dede el decodificador (lamentablemente no pude realizar la conexión directa con el programa de ModelSim y lo que hacía era copiar el resultado del txt y pegarlo en el archivo creado con el mismo nombre en el Pipeline).
- Reg_File.v : Banco de registros con 32 espacios.
- ALU.v : Unidad aritmético-lógica con 8 operaciones
- Mem_Datos.v: Memoria de datos con acceso byte-addressable
- UniCon.v: Generador de señales de control
- ALU_Control.v: Decodificador de operaciones ALU

Buffers del Pipeline:

- IF_ID.v: Buffer entre Fetch y Decode
- ID_EX.v: Buffer entre Decode y Execute
- EX_MEM.v: Buffer entre Execute y Memory
- MEM_WB.v: Buffer entre Memory y Write Back

Componentes de Soporte:

- Adder_32bit.v: Sumador para la PC+4 y cálculos de branch
- Shift_Left_2.v: Desplazador para direcciones de salto
- Sign_Extend.v: Extensor de signo para inmediatos
- MUX_2to1_32bit.v, MUX_2to1_5bit.v: Multiplexores de selección

Ejemplo del flujo de datos en el Pipeline

Instrucción usada para la verificación de este proyecto:

ADD $v1, $at, $v0 = ADD $3, $1, $2 = 000000 00001 00010 00011 00000 100000 (En binario, la instrucción original no tiene espacios)

Valores iniciales:

- $1 = 10 (0x0000000A)
- $2 = 20 (0x00000014)
- $3 = 0 (será actualizada)

Ciclo 1: IF (Intruction Fetch)

Proceso:

- PC envía el address: 0x00000000

- MemIns lee la instrucción en la dirección = 00000000001000100001100000100000
- PC+4 = 0x00000004

Señales en este ciclo:

- PC_current = 00000000
- Instruction = 00221820 (hex)
- PC_plus_4 = 00000004

Estado del Pipeline: IF: ADD $3, $1, $2 | ID: - | EX: - | MEM: - | WB: -

Ciclo 2: ID (Instruction Decode)

IF_ID pasa la instrucción a etapa ID

UniCon decodifica opcode 000000 -> R-tipe

El banco de registros lee: read_reg1 ($1) -> read_data1 = 10, read_reg2 ($2) -> read_data2 = 20

Sing Extend = 0, ya que es una instrucción R no se va a usar.

Señales de control generadas:

RegDst = 1 // Usar rd como destino

ALUSrc = 0 // Usar registro como segundo operando

MemtoReg = 0 // Resultado viene de ALU

RegWrite = 1 // Habilitar escritura en registro

MemRead = 0 // No leer memoria

MemWrite = 0 // No escribir memoria

Branch = 0 // No es branch

ALUOp = 2'b10 // Operación R-type

Estado del Pipeline: IF: SUB $5, $3, $4 | ID: ADD $3, $1, $2 | EX: - | MEM: - | WB: -

Ciclo 3: EX (Execute)

Proceso:

ID/EX Buffer pasa datos a etapa EX:

- read_data1 = 10, read_data2 = 20
- rt = 00010, rd = 00011
- funct = 100000
- Señales de control

ALU Control recibe:

- ALUOp = 10 (R-type)
- funct = 100000
- Salida: ALUControl = 0010 (ADD)

MUX RegDst:

- Entrada 0: rt (00010)
- Entrada 1: rd (00011)
- Sel = 1 → Salida: 00011 ($3)

ALU Operación:

- Operando A: 10
- Operando B: 20
- Operación: ADD
- Resultado: 10 + 20 = 30 (0x0000001E)
- Zero: 0

Cálculos:

ALU: 10 + 20 = 30

Write Register: 00011 ($3)

Estado del Pipeline: IF: ADDI $9, $8, 100 | ID: SUB $5, $3, $4 | EX: ADD $3, $1, $2 | MEM: - | WB: -

Ciclo 4: MEM (Memory Access)

Proceso:

EX/MEM Buffer pasa datos a etapa MEM:

- ALU Result = 30
- Write Register = 00011 ($3)
- RegWrite = 1, MemtoReg = 0

Mem_Datos:

- MemRead = 0, MemWrite = 0
- No hay acceso a la memoria por ser una instrucción R

Señales:

- Address = 0000001E (pero no se usa)
- MemRead = 0, MemWrite = 0
- Read Data = 00000000 (pero no se usa)

Estado del Pipeline: IF: LW $10, 4($8) | ID: ADDI $9, $8, 100 | EX: SUB $5, $3, $4 | MEM: ADD $3, $1, $2 | WB: -

Ciclo 5: WB (Write Back)

Proceso:

MEM_WB pasa los datos a etapa WB:

ALU result = 30

Read Data = 0 (no se usó memoria)

Write Register = 00011 ($3)

RegWrite = 1, MemtoReg = 0

MUX

ENTRADA 0: ALU Result (30)

Entrada 1: read data = 0

Sel = 0 -> salida: 30

Banco de registros:

Write_reg = 00011 ($3)

Write_data = 30

Reg_write = 1 -> se habilita la escritura

Resultado Final: REG[$3] = 0000001E (decimal: 30)

Estado del Pipeline: IF: AND $11, $3, $10 | ID: LW $10, 4($8) | EX: ADDI $9, $8, 100 | MEM: SUB $5, $3, $4 | WB: ADD $3, $1, $2

Al final la instrucción pasa por las 5 etapas para ejecutar la instrucción, pero simultáneamente cuando pasa por los buffers la siguiente instrucción ya está entrando, haciendo que el ciclo Pipeline funcione.

Conclusiones

Durante el desarrollo de este proyecto me di cuenta de cómo es el proceso más básico de una computadora para ejecutar una simple instrucción, y si esto solo fue una, ya estoy medio visualizando como estas instrucciones están pasando de manera rápida en mi computadora o mayor en una computadora de gama mucho más alta que la mía. Esta materia me encanto en estos aspectos para poder empezar en el motivo principal en la cual me metí en esta carrera que fue la construcción y desarrollo de hardware, aunque vimos más software, pero es la base para poder implementar la creación del aparato que deseamos construir.

Aunque tengo que admitir que una parte del proyecto si fue basándome en la construcción final de un usuario de GitHub en la cual solo agarre aquellos módulos que todavía lo los había visto (como

el Shift o el adder) y ver como son realmente las conexiones comparado a lo que teníamos antes si me faltaban varias conexiones para que el Pipeline funcionara de manera completa.

Y para del decodificador ahí si admito que, si me ayudo Andrés, yo no sé todavía Python al %100, entiendo como es la "lógica" del código, pero explicar las funciones que hace si me dificulta demasiado.

Muchas gracias, profesor Jorge por este semestre, realmente aprendí demasiado en el transcurso del semestre, espero volver verlo en otra materia, me gusto como enseñas.

Referencias

Maze. (s. f.). GitHub - maze1377/pipeline-mips-verilog: A classic 5-stage pipeline MIPS 32-bit processor. solve every hazard with stall. GitHub. https://github.com/maze1377/pipeline-mips-verilog/tree/master

Patterson, D. A., & Hennessy, J. L. (1993). Computer Organization and Design: the Hardware/Software Interface.