Universidad Nacional Autónoma de México

Facultad de Ingeniería

Proyecto Final: Ensamblador Básico para Arquitectura IA-32

Materia: Estructura y programación de computadoras

Semestre: 2025-2

Profesora: Ariel Adara Mercado Martínez

Equipo:

Garcia Nava Diego

Ramírez Rodríguez Oscar Gael

Freaner Kuri Omar Ariel

Fecha: 07/12/2025

---

2. Descripción del Problema

El objetivo de este proyecto es desarrollar un ensamblador básico para la arquitectura IA-32 (x86) de 32 bits. El software debe ser capaz de leer un archivo de texto con código fuente en lenguaje ensamblador (.asm) y realizar el proceso de traducción a lenguaje máquina.

Según las especificaciones del proyecto, el sistema debe cumplir con los siguientes requisitos funcionales:

1. **Generación de Archivos:** Debe producir tres salidas:
    - **Tabla de Símbolos:** Mapeo de etiquetas y variables a sus direcciones de memoria.
    - **Tabla de Referencias:** Registro de dónde se utiliza cada etiqueta en el código.
    - **Código Máquina:** El archivo hexadecimal (.hex) ejecutable.
2. **Soporte de Instrucciones:** Debe reconocer y codificar instrucciones de movimiento (MOV, PUSH, POP), aritmética (ADD, SUB, MUL, INC, DEC), lógica (AND, OR, XOR, CMP, TEST), y control de flujo (JMP, JE, JNE, LOOP, CALL, RET, INT).
3. **Modos de Direccionamiento:** Debe soportar direccionamiento registro-registro, inmediato-registro y operaciones con memoria (etiquetas y variables).

El reto principal consiste en implementar dos estrategias de ensamblado:

- **Ensamblador de 2 Pasadas:** Una pasada para mapear símbolos y otra para generar código.
- **Ensamblador de 1 Pasada:** Generación directa utilizando técnicas de *backpatching* para resolver referencias hacia adelante.

---

3. Análisis

Para resolver el problema, se descompuso el sistema en los siguientes componentes lógicos:

3.1. Gestión de Memoria y Direcciones

La arquitectura IA-32 utiliza direcciones de 32 bits. Se determinó que el segmento de código (.text) iniciaría en la dirección virtual 0x1000. Las instrucciones tienen tamaños variables (de 1 a 6 bytes en este subconjunto), por lo que es necesario llevar un contador de programa preciso.

3.2. Cálculo de Saltos (Offsets)

Las instrucciones de salto condicional (como JE, JNE, LOOP) y el salto corto (JMP) utilizan direccionamiento relativo de 8 bits.

- **Fórmula:** Offset = direcciónDestino - (direcciónInstrucciónSalto + tamañoInstrucción)
- **Reto:** Si el salto es hacia atrás, el resultado es negativo y debe representarse en complemento a dos.

3.3. Codificación de Instrucciones (Opcode y ModR/M)

Cada mnemónico (MOV, ADD, etc.) tiene múltiples opcodes dependiendo de sus operandos.

- *Ejemplo:* MOV EAX, EBX usa el opcode 89, mientras que MOV EAX, 1 usa B8.
- **Byte ModR/M:** Para operaciones entre registros o memoria, se debe calcular el byte ModR/M siguiendo la estructura: Mod (2 bits) | Reg (3 bits) | R/M (3 bits).

3.4. Estrategia de Una Pasada (Backpatching)

El problema principal en una pasada es encontrar una instrucción como JMP fin antes de que la etiqueta fin: haya sido leída.

- **Solución:** Se emite un byte temporal (00) en el código máquina y se registra la posición en una lista de "parches pendientes". Al encontrar la etiqueta fin:, se recorre la lista y se actualiza el código binario con el valor real calculado.

---

4. Diseño

La solución se implementó en Python siguiendo un diseño modular orientado a objetos.

4.1. Arquitectura del Sistema

- **main.py:** Punto de entrada. Configura los directorios y ejecuta ambos ensambladores secuencialmente para verificar consistencia.
- **AssemblerI (Interfaz):** Define la estructura base que comparten ambos ensambladores.
- **Parser:** Módulo encargado de análisis léxico. Limpia líneas, separa tokens, identifica directivas y detecta etiquetas.
- **Instruction:** Clases que representan cada tipo de instrucción facilitando el uso de polimorfismo.

4.2. Módulo de 1 Pasada (OnePassAssembler)

Diseñado con un buffer de bytes mutable.

- **Diccionario de Parches:** pending_patches: dict[label, list].
- **Flujo:**
    1. Leer línea.
    2. Si es etiqueta: Registrar en Tabla de Símbolos y resolver parches pendientes.
    3. Si es instrucción: Generar código. Si usa una etiqueta desconocida, escribir 00 y registrar parche.

4.3. Módulo de 2 Pasadas (TwoPassAssembler)

Diseñado para simplicidad lineal.

- **Pasada 1 (Parser):** Recorre el archivo solo calculando tamaños de instrucciones para llenar la Tabla de Símbolos.
- **Pasada 2 (CodeGenerator):** Recorre las instrucciones nuevamente. Como la Tabla de Símbolos ya está llena, traduce directamente a hexadecimal sin necesidad de parches.

---

5. Implementación

El proyecto fue desarrollado utilizando **Python 3.14**.

5.1. Detalles Técnicos Clave

- **Manejo de Codificación:** Se implementó lectura forzada en **UTF-8** para evitar errores de decodificación comunes al leer archivos .asm.
- **Cálculo de ModR/M:** Se implementó una función genérica _encode_reg_reg_byte que realiza operaciones a nivel de bits.
- **Little Endian:** Se utilizó int.to_bytes para asegurar que los números de 32 bits (direcciones y valores inmediatos) respeten el formato de la arquitectura x86.

5.2. Algoritmos Destacados

- **Detección de MOV:** Se implementó lógica condicional para distinguir entre:
    - MOV REG, REG (Opcode 89)
    - MOV REG, [MEM] (Opcode 8B)
    - MOV [MEM], REG (Opcode 89 con dirección)
    - MOV REG, IMM (Opcode B8 + reg_id)
- **Backpatching:** Se implementó soporte tanto para saltos relativos de 8 bits (REL8) como para direcciones absolutas de 32 bits (ABS32) en llamadas a memoria futura.

---

6. Comentarios y Resultados

6.1. Pruebas Realizadas

El ensamblador fue sometido a tres pruebas de complejidad incremental, verificando que **ambos métodos (1 y 2 pasadas) generaran archivos .hex idénticos**:

1. **Fibonacci (fib.asm):** Validó el manejo de bucles (LOOP), variables en memoria y aritmética básica.
2. **Factorial (test2.asm):** Validó la multiplicación (MUL), saltos condicionales complejos (JLE) y el *backpatching* de saltos hacia adelante.
3. **Sumador y Potencia (test3.asm, potencia.asm):** Validó comparaciones con memoria (CMP reg, [mem]), comparaciones con inmediatos y saltos lógicos (JG, JE).

6.2. Conclusiones

El proyecto cumple satisfactoriamente con todos los requisitos planteados.

- Se logró una **precisión total** en la generación de código máquina, coincidiendo con los opcodes estándar de Intel.
- La implementación de **dos arquitecturas distintas** (1 y 2 pasadas) que convergen en el mismo resultado binario demuestra la robustez del análisis de direcciones y el manejo de símbolos.
- El sistema es capaz de manejar instrucciones complejas y resolver referencias cruzadas de manera eficiente.