

Universidad Autónoma de México

Facultad de Ingeniería

Estructura y Programación de Computadoras

Integrantes:

Cano Barrera Paulina

Muñoz Mendoza David

Grupo: 5

Semestre 2025-2

Introducción

En el transcurso del curso se ha comprendido acerca el lenguaje de bajo nivel como lo es el lenguaje ensamblador el cual permite una comunicación directa con el hardware. Este lenguaje es fundamental para la ejecución de programas escritos en lenguaje de alto nivel, ya que muchos de estos dependen internamente de instrucciones de bajo nivel.

En este proyecto, se ha desarrollado un programa de Python el cual es considerado un lenguaje de alto nivel el cual permite construir y variar dichas herramientas. El objetivo es comprobar la funcionalidad y veracidad del ensamblador diseñado, evaluando si procesa correctamente las instrucciones y estructuras típicas de un lenguaje ensamblador IA-32(x86) de 32 bits.

Desarrollo (Análisis y Diseño de la solución);

Para comenzar es necesario analizar las instrucciones a implementar en el programa estas instrucciones deben ser básicas para poder observar el análisis si es correcto o no, para esto se implementaran instrucciones como se muestra en la tabla 1.1.

Instrucción	Sintaxis
MOV	MOV dest,src
ADD	ADD dest, src
SUB	SUB dest,src
INC	INC op
DEC	DEC op
JMP	JMP label
CMP	CMP op1,op2
JLE	JLE label
JL	JL label

JZ	JZ label
JNZ	JNZ label
JA	JA label
JAE	JAE label
JB	JB label
JBE	JBE label
JE	JE label
JNE	JNE label
JG	JG label
JGE	JGE label
MUL	MUL op
IMUL	IMUL op
DIV	DIV op
IDIV	IDIV op
RET	RET
PUSH	PUSH op
POP	POP op
INT	INT imm8
LOOP	LOOP label
XOR	XOR dest,scr
TEST	TEST op1,op2
MOVZX	MOVZX dest,scr
XCHG	XCHG op1,op2
ENTER	ENTER imm,0
LEAVE	LEAVE

Para encontrar las soluciones correspondientes para este problema es necesario aclarar las funciones que cada instrucción al igual que algunas características principales. Con base a lo anterior se consideró crear instrucciones más simples al principio como "MOV" y "CMP".

Para el "MOV" es necesario que tenga dos operandos y existen diferentes posibilidades en los operandos como "MOV eax, val" donde existe un opcode específico para el movimiento de un valor inmediato a un registro, "MOV eax,ebx" donde existe un movimiento entre registros y se debe calcular el opcode respecto al valor binario de cada registro y convertirlo a hexadecimal y la tercer opción es mover una dirección de memoria a un registro "MOV eax,0x1990".

La instrucción "CMP" tiene como función comparar ya sea entre dos registros o un entre un registro y un valor inmediato por lo que es importante considerarlo para su correcto procesamiento. Instrucciones como "JUMP", "JE", "JNE", "JG"..., son instrucciones que necesitan un solo operando para funcionar el cual debe ser una etiqueta, por lo que se consideró englobar todo lo necesario para simplificar el código y mejorar funcionalidad, estas instrucciones cobran sentido cuando antes se ingresa una instrucción como lo es "CMP" y la instrucción de salto hace de redireccionamiento a la etiqueta.

Posteriormente aplicar el funcionamiento de operaciones básicas sin signo como lo pueden ser "ADD", "SUB", "INC", "DEC" "MUL", "DIV" las cuales representan la adición aplicada de un valor, registro o memoria hacia un registro o memoria, sustracción que de igual forma se aplica de un valor, registro o memoria hacia un registro o memoria, para las demás instrucciones la incrementación en 1, decremento en 1, multiplicación, división solo necesitan de un operando para funcionar. Por otra parte también se aplican "IMUL" y "IDIV" aunque de igual manera solo necesitan un operando, se analizan por separado ya que conllevan operaciones con signo, incrementado las operaciones para su funcionamiento.

En el lenguaje ensamblador a veces existen instrucciones las cuales comunican con la misma computadora o hacen operaciones específicas para procesar datos un ejemplo que se implementó son "CALL" y "RET" donde el primero debe tener como operando una etiqueta y la instrucción da un salto a la etiqueta establecida y en caso que no exista se aplican las referencias pendientes, por otro lado el segundo es necesario para regresar a la etiqueta donde se llamó la etiqueta para saltar y para esta instrucción no es necesario

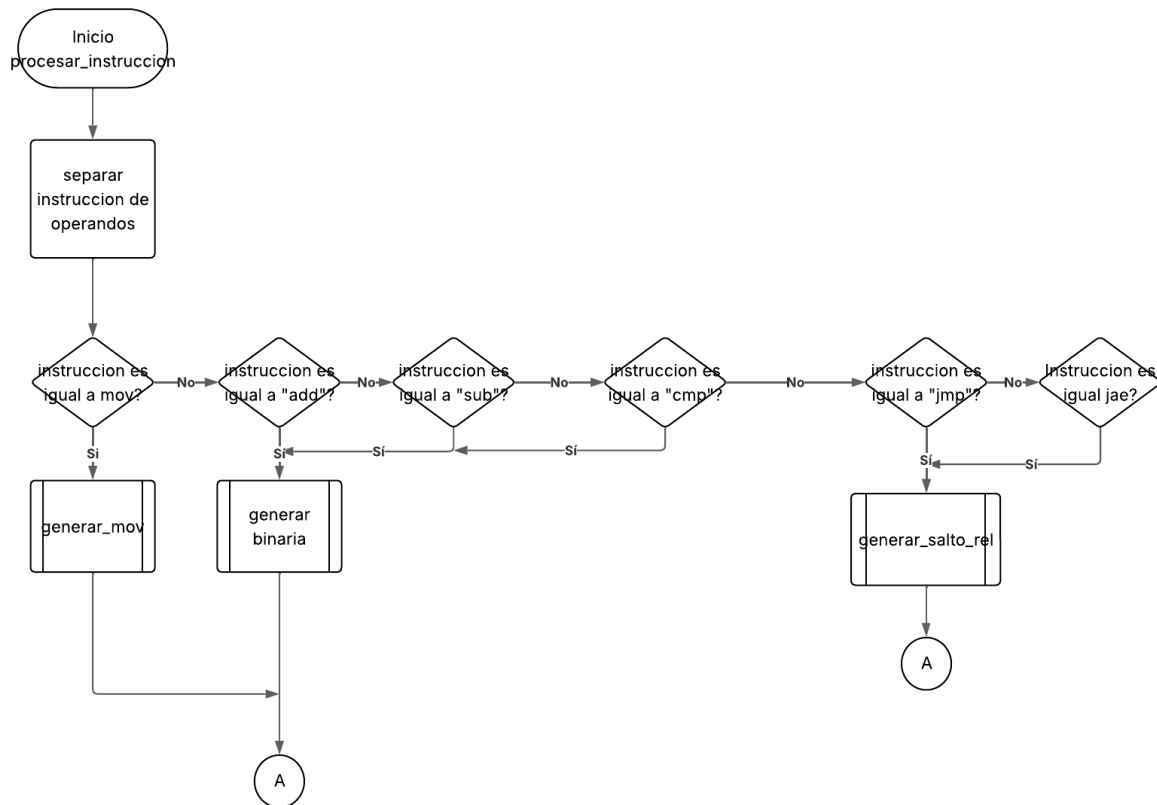
un operando. Sin embargo, no solo estas instrucciones son importantes. También es crucial analizar POP y PUSH, que son instrucciones que permiten al lenguaje ensamblador comunicarse con la pila para almacenar o recuperar información. Estas instrucciones tienen la particularidad de que pueden guardar valores, registros o incluso una dirección de memoria

Por último, se usan instrucciones más complejas como "INT", que genera una interrupción del sistema basada en un operando hexadecimal (0x...) para ejecutar una acción específica. La instrucción "LOOP" permite repetir un bloque de código mientras el registro contador "ECX" sea distinto de cero, con la sintaxis mostrada en la tabla 1.1.

Otras instrucciones son:

- "XOR", que realiza una operación lógica comúnmente usada para poner registros en cero.
- "TEST", el cuál compara dos operandos con un AND lógico sin modificar sus valores, solo actualiza los flags.
- "MOVZX", copia un valor de menor tamaño a un registro mayor, rellendo con ceros los bits superiores.
- "XCHG", que intercambia el contenido de dos registros o variables.
- "ENTER" y "LEAVE", instrucciones las cuales se colocan al inicio y final de una función respectivamente, para gestionar correctamente el marco de pila.

Con base al análisis de todas las instrucciones se observa que es necesario implementar múltiples funciones con la finalidad de que el proceso quede separado y tenga una mejor organización visualmente esto tomando en cuenta el análisis del funcionamiento respecto a las instrucciones en la tabla 1.1. Para poder acceder a estas funciones la clave es una separación de palabras para poder detectar cual es la instrucción a emplear y por medio de concatenaciones de decisiones(if) poder acceder a dichas funciones. las cuales deberán variar acorde a lo que se necesite por lo que la función procesar instrucción quedaría masomenos de la forma.



Implementación:

Se implementa la estructura general para la identificación de cada instrucción y así redireccionarla para su correcto funcionamiento como se planteó

procesar_instruccion

```

def procesar_instruccion(self, instruccion):
    """Procesa una instruccion y genera el codigo hex"""

    instruccion = instruccion.lower().strip()
    # 1. Separar y operandos
    partes = instruccion.split(None, 1)
    if not partes:
        return
  
```

```

inst = partes[0].lower()
operandos = [op.strip().lower() for op in partes[1].split(',')]
if inst == 'mov':

    self.generar_mov(operandos)
elif inst == 'add':
    self.generar_binaria(operandos, 'add')
elif inst == 'sub':
    self.generar_binaria(operandos, 'sub')
elif inst == 'cmp':
    self.generar_binaria(operandos, 'cmp')
elif inst == 'jmp':
    if not operandos[0]:
        print(f"Error: JMP requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0xE9, 5) # JMP NEAR rel32
elif inst == 'je':
    if not operandos[0]:
        print(f"Error: JE requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x74, 2)

elif inst == 'jle':
    if not operandos[0]:
        print(f"Error: JLE requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x7E, 2)

elif inst == 'jl':
    if not operandos[0]:
        print(f"Error: JL requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x7C, 2)

elif inst == 'jz':
    if not operandos[0]:
        print(f"Error: JX requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x74, 2)

elif inst == 'jnz':
    if not operandos[0]:
        print(f"Error: JNZ requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x75, 2)

elif inst == 'ja':
    if not operandos[0]:
        print(f"Error: JA requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x77, 2)

elif inst == 'jae':
    if not operandos[0]:
        print(f"Error: JAE requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x73, 2)

```

```

elif inst == 'jb':
    if not operandos[0]:
        print(f"Error: Jb requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x72, 2)
elif inst == 'jbe':
    if not operandos[0]:
        print(f"Error: JE requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x76, 2)
elif inst == 'jg':
    if not operandos[0]:
        print(f"Error: Jg requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x7F, 2)
elif inst == 'jge':
    if not operandos[0]:
        print(f"Error: JGE requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x7D, 2)
elif inst == 'xor':
    self.generar_binaria(operandos, 'xor')
elif inst == 'xchg':
    self.generar_xchg(operandos)
elif inst == 'push':
    self.generar_push(operandos)
elif inst == 'pop':
    self.generar_pop(operandos)
elif inst == 'ret':
    self.generar_ret()

elif inst == 'jb':
    if not operandos[0]:
        print(f"Error: JNE requiere un operando de destino.")
        return
    self.generar_salto_rel(operandos[0], 0x75, 2) # JNE rel8
elif inst == 'inc':
    self.generar_incdec(operandos, 'INC')
elif inst == 'dec':
    self.generar_incdec(operandos, 'DEC')

elif inst == 'mul':
    self.generar_mul(operandos)
elif inst == 'imul':
    self.generar_imul(operandos)
elif inst == 'div':
    self.generar_div(operandos)
elif inst == 'idiv':
    self.generar_idiv(operandos)
elif inst == 'call':
    if not operandos[0]:
        print(f"Error: call requiere un operando de destino.")
        return
    self.generar_call(operandos[0], 0xE8, 2)
else:
    print(f"Error: Instrucción '{inst}' no implementada o mal formada.")

```

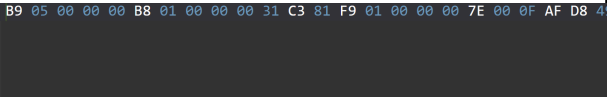

Posteriormente con base a estas decisiones o if concatenados se crearon funciones como:

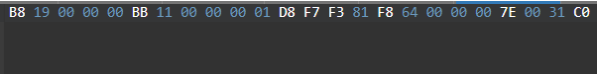
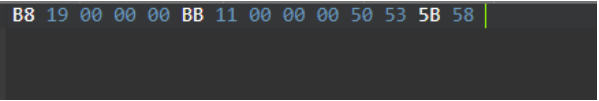
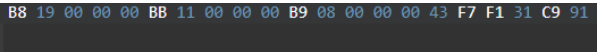
<p>función para los saltos hacia etiquetas, jl.jle,jg , jge, jmp....., esta función tiene como parámetros la etiqueta el opcode base de la instrucción además del tamaño de la instrucción.</p>	<pre>def generar_salto_rel(self, etiqueta, opcode_base, largo_instruccion): if etiqueta in self.tabla_simbolos: dir_etiqueta = self.tabla_simbolos[etiqueta] # Offset relativo: destino - (posición actual + longitud de la instrucción de salto) offset = dir_etiqueta - (self.contador_posicion + largo_instruccion) self.codigo_hex.append(opcode_base) if largo_instruccion == 2: # Jcc rel8 # El offset de 8 bits debe estar en el rango -128 a 127 if not (-128 <= offset <= 127): print(f'Advertencia: Salto corto para '{etiqueta}' está fuera de rango. Puede fallar.') self.codigo_hex.append(offset & 0xFF) elif largo_instruccion == 5: # JMP rel32 for i in range(4): self.codigo_hex.append((offset >> (8 * i)) & 0xFF) self.contador_posicion += largo_instruccion else: # Si la etiqueta no está en la tabla de símbolos, es una referencia pendiente self.referencias_pendientes.setdefault(etiqueta, []).append(self.contador_posicion) self.codigo_hex.append(opcode_base) # Rellenar con ceros temporales for _ in range(largo_instruccion - 1): self.codigo_hex.append(0x00) self.contador_posicion += largo_instruccion</pre>
<p>función para el incremento y decremento</p>	<pre>def generar_incdec(self, operandos, operacion_tipo): if len(operandos) != 1: print(f'Error: {operacion_tipo} requiere 1 operando.') return reg = operandos[0] if reg not in opcodes.REGISTROS_32_BIT: print(f'Error: Registro '{reg}' no válido para {operacion_tipo}.')</pre>

	<pre> return # Opcodes directos para INC/DEC reg32 (40h + reg_encoding / 48h + reg_encoding) base_opcode = 0x40 if operacion_tipo == 'INC' else 0x48 opcode = base_opcode + opcodes.REGISTROS_32_BIT[reg] self.codigo_hex.append(opcode) self.contador_posicion += 1 </pre>
<p>generación de código a partir de la operación en la cual se le transmiten los operandos y la instrucción</p>	<pre> def generar_binaria(self, operandos, operacion): if len(operandos) != 2: print(f"Error: {operacion} requiere 2 operandos.") return dest, src = operandos[0], operandos[1] opmap = {'add': 0x01, 'sub': 0x29, 'cmp': 0x39, 'xor': 0x31} immmap = {'add': 0b000, 'sub': 0b101, 'cmp': 0b111, 'xor': 0b110} if dest in opcodes.REGISTROS_32_BIT and src in opcodes.REGISTROS_32_BIT: opcode = opmap[operacion] modrm = (0b11 << 6) \ (opcodes.REGISTROS_32_BIT[src] << 3) \ opcodes.REGISTROS_32_BIT[dest] self.codigo_hex.extend([opcode, modrm]) self.contador_posicion += 2 return # Caso: REG, IMM (Ej: ADD EAX, 10h) if dest in opcodes.REGISTROS_32_BIT: try: valor = int(src, 0) opcode = 0x81 # Opcode base para ADD/SUB/CMP reg32, imm32 reg_field = immmap[operacion] # Campo 'reg' en ModR/M </pre>

	<pre> modrm = (0b11 << 6) \ (reg_field << 3) \ opcodes.REGISTROS_32_BIT[dest] self.codigo_hex.extend([opcode, modrm]) # Añadir valor inmediato en little-endian (4 bytes para 32-bit) for i in range(4): self.codigo_hex.append((valor >> (8 * i)) & 0xFF) self.contador_posicion += 6 # 1 opcode + 1 modrm + 4 bytes inmediato return except ValueError: pass # No es un valor inmediato, fallar al siguiente caso print(f"Error: operandos inválidos para {operacion}: {dest}, {src}") </pre>
--	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Pruebas:

¿Qué hace?	Código ensamblador	Captura del resultado
Ejemplo de factorial iterativo	<pre> _start: mov ecx,5 mov eax,1 xor ebx,eax calcular: cmp ecx,1 jle fin imul ebx,eax dec ecx jmp calcular fin: mov 0,eax </pre>	

Aplica la adición, división y salto para operaciones con valores inmediatos	<pre> _start: mov eax, 25 mov ebx, 17 add eax, ebx div ebx cmp eax, 100 jle loong loong: xor eax, eax </pre>	
Usa las instrucciones de modificación para la pila	<pre> _start: mov eax, 25 mov ebx, 17 push eax push ebx pop ebx pop eax </pre>	
Codigo que emplea operaciones como inc, div, xor y xchg para verificar el uso de las instrucciones	<pre> org 0x1002 _start: mov eax, 25 mov ebx, 17 mov ecx, 8 inc ebx div ecx xor ecx, ecx xchg eax, ecx </pre>	

- Manual de Usuario

Instrucciones de uso

1. Escriba su código ensamblador en un archivo llamado “programa.asm”, utilizando únicamente las instrucciones soportadas y respetando la sintaxis IA-32.
2. Asegúrese de que el archivo “opcodes.py” se encuentre en el mismo directorio que el archivo “[main.py](#)”.
3. Ejecute el archivo principal de Python con:

```
python main.py
```
4. El programa generará automáticamente tres archivos:
 - “programa.hexx”: contiene el código máquina (opcode seguido de los operandos, en formato hexadecimal horizontal).
 - “simbolos.txt”: tabla de símbolos con las etiquetas declaradas y sus direcciones correspondientes.
 - “referencias.txt”: contiene los saltos o etiquetas que no fueron resueltas en la primera pasada.

- Advertencias importantes

- Cada instrucción debe ocupar una línea y debe estar separada de sus operandos.
- Los operandos deben ir separados por comas “,”.
- No se permite escribir la instrucción y los operandos sin espacios de separación.
- Revise la lista de instrucciones soportadas antes de compilar.
- Las instrucciones deben seguir el estándar IA-32, respetando tamaño y sintaxis.

- Manual Técnico

Estructura principal

1. La clase “EnsambladorIA32” contiene todos los métodos para:
 - Procesar líneas de código (“procesar_linea”)
 - Identificar etiquetas y actualizar la tabla de símbolos (“procesar_etiqueta”)
 - Generar instrucciones específicas (“generar_mov”, “generar_add”, “generar_imul”, etc.)

- Resolver referencias pendientes a etiquetas ("resolver_referencias_pendientes")
- Generar archivos de salida ("generar_hex", "generar_reportes")

2. Módulo externo

- El archivo "opcodes.py" contiene todos los diccionarios necesarios para identificar:
 - Códigos de operación ("MOV_OPCODES")
 - Registros válidos ("REGISTROS_32_BIT")
 - Códigos modRM necesarios para codificar instrucciones binarias

3. Pasadas del ensamblador

1. Primera pasada:

- Procesa línea por línea.
- Genera el código binario parcial.
- Registra las referencias a etiquetas no definidas.

2. Segunda pasada:

- Resuelve las referencias pendientes.
- Calcula los offsets para instrucciones de salto.

- Extensibilidad

Nuevas instrucciones pueden ser añadidas implementando una función "generar_<mnemonico>()" y actualizando el método "procesar_instruccion".

- Instrucciones actuales soportadas (ejemplo)

MOV, ADD, SUB, CMP, JMP, JE, JNE, INC, DEC, IMUL (1-3 operandos), MUL, DIV, IDIV, XOR, XCHG, PUSH, POP, RET

Conclusiones (individuales y por equipo)

Paulina Cano Barrera

El ensamblador desarrollado durante este proyecto nos permite traducir instrucciones básicas del lenguaje IA-32 a código máquina, generando correctamente los archivos “programa.hexx”, “simbolos.txt” y “referencias.txt”. Aunque no se logró implementar la totalidad de las instrucciones, se cubrió una parte representativa y funcional para programas simples. Desde mi experiencia personal, se tuvieron algunas complicaciones, como al crear la instrucción xor, ya que a la hora de generar el código hexadecimal este nos daba uno diferente al esperado, pero después de investigar e intentar juntos logramos hacer que funcionara. Este proyecto nos ayudó a entender mejor el funcionamiento e implementación del lenguaje de bajo nivel y poner en práctica los conocimientos teóricos obtenidos durante las clases.

David Muñoz Mendoza

Con base al proyecto presentado considero que se implementó la mayoría de instrucciones esto causado por las complicaciones al tratar de implementarlas con el código general, por lo que puedo decir que el código es funcional para programas de lenguaje ensamblador básicos, algo importante que agregar es la correcta generación de código hexadecimal para el lenguaje ensamblador utilizado en el apartado de pruebas por lo que puedo concluir que se cumplió con los requisitos principales solicitados los cuales implican la generación de documentos “programa.hexx”, “simbolos.txt”, “referencias.txt”.

Por equipo:

Durante este proyecto logramos construir un ensamblador funcional capaz de interpretar instrucciones clave del lenguaje IA-32 y traducirlas a código máquina en formato hexadecimal. A pesar de que no se alcanzó a cubrir todo el conjunto de instrucciones, el sistema responde correctamente para múltiples casos y genera los archivos de salida requeridos.

El trabajo realizado nos permitió aplicar conocimientos obtenidos durante el curso, como de estructuras de datos, manipulación de bits y codificación binaria, reforzando habilidades tanto técnicas como de organización en equipo. Las dificultades encontradas, como la gestión de operandos y validación de registros, nos llevaron a comprender mejor el comportamiento real del hardware al ejecutar instrucciones.

Nuestra experiencia al realizar este proyecto tuvo grandes matices, ya que hubo momentos donde no lograbamos hacer que funcionara de manera correcta, y otros en los que funcionaba sin el menor inconveniente. Fue una gran experiencia y sirvió para reforzar y enriquecer el conocimiento adquirido.