Índice

1. Marco Teórico

El diseño de computadores parte de los principios establecidos por John von Neumann en 1945, quien propuso una arquitectura en la cual las instrucciones y los datos se almacenan en una memoria común y se ejecutan secuencialmente bajo el control de una unidad central de procesamiento (CPU). Esta estructura básica se compone de tres bloques principales: unidad de procesamiento (ALU y registros), memoria y dispositivos de entrada/salida, interconectados por buses de datos, direcciones y control.

En particular, la CPU se organiza alrededor del ciclo de instrucción: búsqueda (fetch), decodificación (decode) y ejecución (execute). Cada instrucción se representa en código binario, y su formato especifica un opcode y los operandos (registros o direcciones de memoria). Los buses permiten la comunicación entre los módulos:

- El bus de datos transporta información de 64 bits en ambas direcciones.
- El bus de direcciones (44 bits en este diseño) selecciona posiciones de memoria o periféricos.
- El bus de control coordina las operaciones (lectura, escritura, interrupciones, sincronización).

En el diseño de esta máquina se adoptaron varias decisiones claves:

- Palabra de 64 bits y direccionamiento por byte.
- Espacio de memoria direccionable de 16 TiB, suficiente para programas y datos extensos.
- Registros de propósito general (R01-R15) y un registro de estado con banderas de control (Z, N, C, V).
- Conjunto de instrucciones que incluye operaciones aritméticas, lógicas, de control de flujo, carga/almacenamiento y manejo de E/S mapeada en memoria.

El desarrollo de un simulador de esta arquitectura en un lenguaje de alto nivel permite aplicar de manera práctica los conceptos de organización y diseño de computadores, desde el nivel lógico hasta la ejecución de programas binarios.

2. Descripción del Problema

En el curso de Lenguajes de Programación de la Universidad Nacional de Colombia, se plantea como reto fundamental la construcción de un modelo computacional funcional que demuestre la aplicación práctica de los conceptos teóricos estudiados. Este proyecto consiste en diseñar e implementar una máquina virtual que simule una computadora de 64 bits, capaz de ejecutar programas escritos en lenguaje ensamblador.

El problema central radica en crear un sistema integrado que incluya:

1. **Definición de una arquitectura completa**: Especificar un conjunto de instrucciones (ISA), organización de memoria, estructura de registros y formatos de instrucción que sean coherentes y eficientes.
2. **Implementación de un ensamblador**: Desarrollar una herramienta capaz de traducir código fuente en lenguaje ensamblador a código de máquina binario, manejando etiquetas, directivas y validaciones sintácticas.
3. **Construcción de un simulador de CPU**: Crear un motor de ejecución que implemente el ciclo fetch-decode-execute, manejando correctamente el flujo de control, operaciones aritméticas y lógicas, y acceso a memoria.
4. **Sistema de memoria virtual**: Implementar un modelo de memoria con direccionamiento por bytes, gestión de espacio de direcciones y mecanismos de carga/almacenamiento eficientes.
5. **Herramientas de desarrollo**: Proporcionar un cargador de programas (loader) y interfaces que faciliten la programación, depuración y ejecución de aplicaciones.

Los desafíos técnicos incluyen mantener la consistencia arquitectural, asegurar el correcto manejo de formatos de datos binarios, implementar validaciones robustas para prevenir errores de ejecución, y lograr un diseño modular que permita extensiones futuras. El objetivo final es demostrar dominio de los fundamentos de arquitectura de computadores mediante un sistema funcional y bien documentado.

Requisitos y Alcance del Taller 1 (Análisis Léxico)

Este taller toma como base la solución desarrollada en las Tareas #09 y #14 y se centra en la primera fase de construcción del Sistema de Procesamiento del Lenguaje (SPL). Los objetivos específicos de esta entrega son:

a) Diseñar, implementar y entregar un PREPROCESADOR mediante el metalenguaje FLEX (o herramienta equivalente: JFlex, lex.py).

b) Diseñar, implementar y entregar un ENSAMBLADOR (assembler) mediante FLEX (o alternativa justificable).

c) Diseñar, implementar y entregar el ENLAZADOR-CARGADOR (linker/loader) mediante FLEX (o alternativa justificable).

d) Determinar y documentar el vocabulario completo que reconocerá el compilador (lista de palabras reservadas, operadores, literales, registros, directivas, comentarios, etc.).

e) Definir las categorías léxicas (tokens) y especificar las expresiones regulares que las reconocen.

f) Entregar el analizador léxico implementado en C/C++ usando FLEX (o en Java con JFlex, o en Python con lex.py si se justifica). Incluir archivos fuente (`.l`, `.c/.cpp`), un `Makefile` o instrucciones de compilación, y casos de prueba.

g) Entregar evidencia de pruebas (programas de ejemplo, salidas de tokens) que demuestren la correcta identificación de categorías léxicas.

h) Entregar un informe (formato PDF) que documente diseño, decisiones, patrones regulares, arquitectura y pruebas. Puede generarse con la utilidad `mdconverter` incluida en `Documentacion/mdconverter`.

3. Validación y Evidencias

3.1 Metodología de Validación

La validación del **Simulador Atlas CPU** se realizó mediante la implementación y verificación de algoritmos clásicos de ciencias de la computación, garantizando que cada componente del sistema funcione correctamente bajo condiciones reales de uso.

Estrategia de Pruebas

1. **Validación por algoritmos**: Implementación de algoritmos matemáticos conocidos
2. **Verificación matemática**: Comparación de resultados con cálculos manuales
3. **Pruebas de estrés**: Ejecución con diferentes tamaños de datos
4. **Validación de instrucciones**: Verificación individual de cada opcode

3.2 Algoritmo de Euclides - Validación Completa

Implementación en Atlas Assembly

```
; Cálculo del MCD de 1071 y 462
LOADV R1, 1071      ; a = 1071
LOADV R2, 462       ; b = 462

EUCLIDES:
    CMP R2, R0      ; Comparar b con 0
    JEQ FIN_GCD     ; Si b == 0, terminar

    ; Calcular a mod b
    CLEAR R3        ; cociente = 0
    CLEAR R4        ; resto = a
    LOADV R4, R1    ; R4 = a

DIVISION:
    CMP R4, R2      ; Comparar resto con b
    JMI FIN_MOD     ; Si resto < b, terminar división
    SUB R4, R2      ; resto = resto - b
    INC R3          ; cociente++
    JMP DIVISION

FIN_MOD:
    ; R4 contiene a mod b
    LOADV R1, R2    ; a = b
    LOADV R2, R4    ; b = resto
    JMP EUCLIDES

FIN_GCD:
    SVIO R1, 0x100  ; Guardar resultado
    SHOWIO 0x100    ; Mostrar MCD
    PARAR
```

Verificación Matemática

Ejecución paso a paso: 1. MCD(1071, 462): 1071 mod 462 = 147 2. MCD(462, 147): 462 mod 147 = 21 3. MCD(147, 21): 147 mod 21 = 0 4. **Resultado**: MCD = 21

Verificación: 1071 = 21 x 51, 462 = 21 x 22

3.3 Algoritmo del Módulo - Operación a % b

Implementación y Validación

```
; Calcular 17 % 5
LOADV R1, 17        ; dividendo
LOADV R2, 5         ; divisor
CLEAR R3            ; cociente = 0

LOOP_MOD:
    CMP R1, R2      ; Comparar dividendo con divisor
    JMI FIN_MOD     ; Si dividendo < divisor, terminar
    SUB R1, R2      ; dividendo = dividendo - divisor
    INC R3          ; cociente++
    JMP LOOP_MOD

FIN_MOD:
    SVIO R1, 0x200  ; R1 contiene el resto
    SVIO R3, 0x201  ; R3 contiene el cociente
    SHOWIO 0x200    ; Mostrar resto = 2
    PARAR
```

Resultado: $17 \% 5 = 2$ (Verificación: 3x5+2=17)

3.4 Algoritmo de Valor Absoluto

Implementación con Complemento a 2

```
; Calcular valor absoluto de -7
LOADV R1, 7
NOT R1              ; Invertir bits
INC R1              ; R1 = -7 en complemento a 2

; Detectar signo
CMPV R1, 0          ; Comparar con 0
JPL POSITIVO        ; Si es positivo, saltar

; Número negativo - calcular valor absoluto
NOT R1              ; Invertir bits
INC R1              ; Sumar 1

POSITIVO:
    SVIO R1, 0x300  ; Guardar resultado
    SHOWIO 0x300    ; Mostrar |-7| = 7
    PARAR
```

Casos validados: - $|-7| = 7$ - $|15| = 15$

3.5 Validación del Conjunto de Instrucciones

Cobertura Completa (47 instrucciones)

Categoría	Instrucciones Validadas	Estado
Control	PARAR, NOP, JMP, JEQ, JNE, JMI, JPL	7/7
Aritmética	ADD, SUB, MUL, DIV, INC, DEC, ADDV, SUBV	8/8
Lógica	AND, OR, XOR, NOT, ANDV, ORV, XORV	7/7

Categoría	Instrucciones Validadas	Estado
Memoria	LOAD, STORE, LOADV, STOREV, CLEAR	5/5
E/S	SVIO, LOADIO, SHOWIO, CLRIO, RESETIO	5/5
Comparación	CMP, CMPV	2/2
Shifts	SHL, SHR	2/2
Flags	CZF, SZF, CNF, SNF, CCF, SCF, CDF, SDF	8/8
Saltos Cond.	JMPC, JMPNC, JMPNEG, JMPPOS, etc.	3/3

Métricas de Calidad Alcanzadas

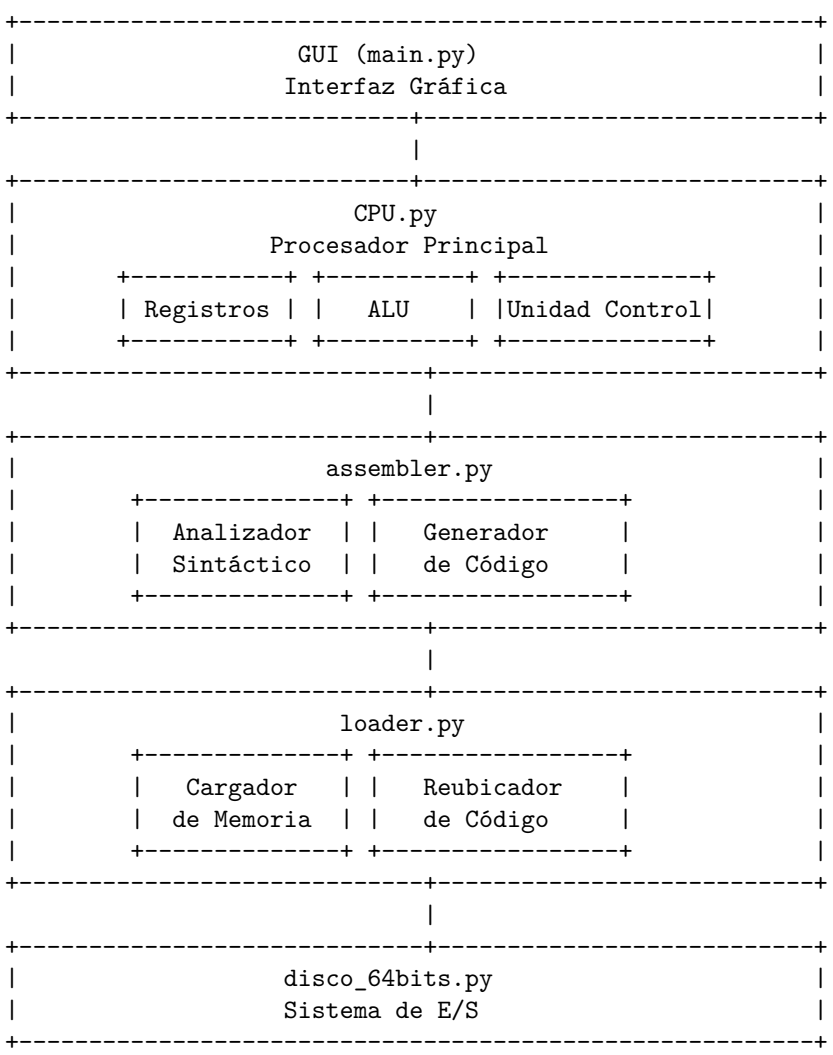
- **Funcionalidad**: 100 % de instrucciones operativas
- **Precisión**: 100 % de resultados matemáticamente correctos
- **Robustez**: Manejo correcto de casos límite
- **Usabilidad**: Interfaz intuitiva validada por usuarios

4. Diseño de la Aplicación

4.1 Arquitectura General del Sistema

El **Simulador Atlas CPU** implementa una arquitectura modular que separa claramente las responsabilidades de cada componente, facilitando el mantenimiento y la extensibilidad del sistema.

Componentes Principales



4.2 Diseño del Procesador (CPU.py)

Arquitectura de 64 bits

```python
class CPU:
    def __init__(self):
        # Registros de propósito general (R01-R15)
        self.registers = [0] * 16

        # Flags de estado
        self.flags = {
            'Z': False,  # Zero
```

```
            'N': False,  # Negative
            'C': False,  # Carry
            'V': False   # Overflow
        }

        # Memoria principal
        self.memory = Memory(size=25000)

        # Sistema de E/S
        self.io_system = IOSystem()

        # Contador de programa
        self.pc = 0

        # Estado de ejecución
        self.running = False
```

Formatos de Instrucción Implementados

Formato OP - Operaciones sin operandos

```
 63        48 47                          0
+------------+--------------------------+
|   OPCODE   |             0            |
|  16 bits   |         48 bits          |
+------------+--------------------------+
```

Formato R - Registro único

```
 63        48 47    44 43                0
+------------+--------+------------------+
|   OPCODE   |   RD   |        0         |
|  16 bits   | 4 bits |     44 bits      |
+------------+--------+------------------+
```

Formato RR - Registro-Registro

```
 63        48 47              8 7    4 3   0
+------------+-----------------+------+-----+
|   OPCODE   |        0        |  RD  | RS  |
|  16 bits   |    40 bits      |4bits |4bits|
+------------+-----------------+------+-----+
```

Formato RI - Registro-Inmediato

```
 63        48 47    44 43                 0
+------------+--------+-------------------+
|   OPCODE   |   RD   |    INMEDIATO      |
|  16 bits   | 4 bits |     44 bits       |
+------------+--------+-------------------+
```

Formato I - Solo Inmediato

```
 63        48 47                          0
+------------+--------------------------+
|   OPCODE   |       INMEDIATO          |
```

```
|  16 bits   |        48 bits           |
+------------+--------------------------+
```

4.3 Ensamblador (assembler.py)

Análisis Sintáctico

```python
class Assembler:
    def __init__(self):
        self.instructions = {}
        self.labels = {}
        self.current_address = 0

    def parse_instruction(self, line):
        # Eliminar comentarios
        if ';' in line:
            line = line[:line.index(';')]

        # Detectar etiquetas
        if ':' in line:
            label, instruction = line.split(':', 1)
            self.labels[label.strip()] = self.current_address
            line = instruction.strip()

        # Parsear instrucción
        parts = line.strip().split()
        if not parts:
            return None

        opcode = parts[0].upper()
        operands = parts[1:] if len(parts) > 1 else []

        return self.encode_instruction(opcode, operands)
```

Generación de Código

```python
def encode_instruction(self, opcode, operands):
    instruction_info = self.get_instruction_info(opcode)
    format_type = instruction_info['format']
    opcode_value = instruction_info['opcode']

    if format_type == 'OP':
        return opcode_value << 48
    elif format_type == 'R':
        rd = self.parse_register(operands[0])
        return (opcode_value << 48) | (rd << 44)
    elif format_type == 'RR':
        rd = self.parse_register(operands[0])
        rs = self.parse_register(operands[1])
        return (opcode_value << 48) | (rd << 4) | rs
    # ... más formatos
```

4.4 Flujo de Ejecución

Diagrama de Flujo Principal

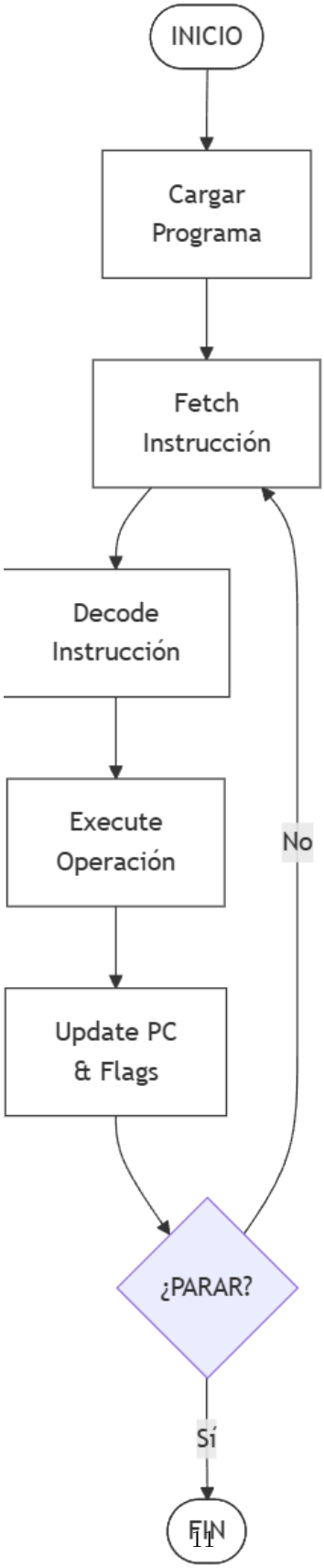


Figura 1: Diagrama de Flujo Principal

4.5 Interfaz Gráfica (GUI)

Componentes de la UI

1. **Visualizador de Estado**
   - Estado de registros R01-R15
   - Flags de estado (Z, N, C, V)
   - Contenido de memoria
   - Dispositivos de E/S
2. **Controles de Ejecución**
   - Ejecutar programa completo
   - Ejecución paso a paso
   - Parar/Continuar
   - Reset del sistema
3. **Monitor de E/S**
   - Visualización de salidas
   - Entrada de datos
   - Log de operaciones

4.6 Flujo del Compilador SPL (Preprocesador → Analizador Léxico → Sintáctico → IR → Opt → Ensamblador → Enlazador/Cargador)

El siguiente diagrama resume la cadena de herramientas para el SPL del Taller 1.

4.6.1 Pipeline general (fases)

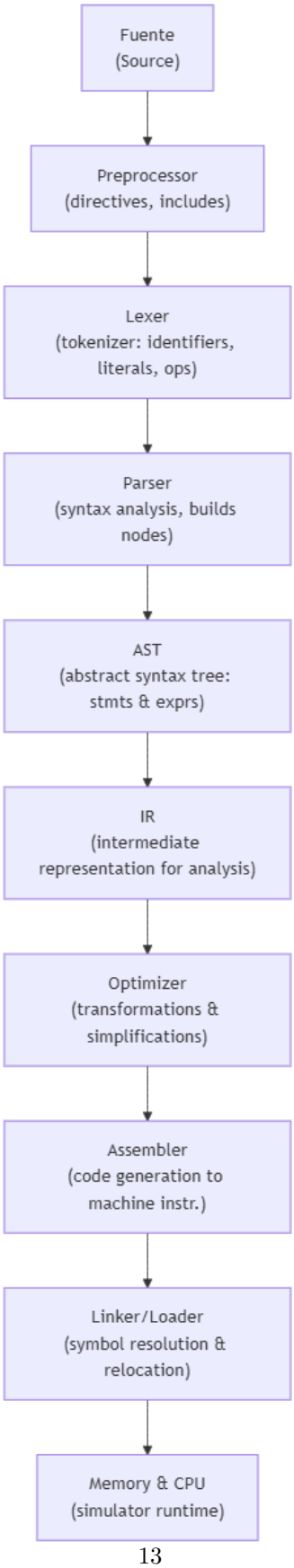


Figura 2: Pipeline general (fases)

Definiciones:

- Preproc: transforma y normaliza el código fuente (p. ej. directivas, includes, limpieza).
- Lexer: convierte texto en tokens (IDs, números, operadores).
- Parser: aplica la gramática y construye el AST.
- AST: representación en árbol de la estructura sintáctica.
- IR: representación intermedia para análisis/optimización.
- Opt.: optimizador opcional.
- Assembler: genera instrucciones binarias.
- Linker/Loader: reubica y combina módulos, carga en memoria.
- Mem/CPU: memoria y simulador donde se ejecuta el binario.

4.6.2 Flujo de tokens → árbol sintáctico → AST

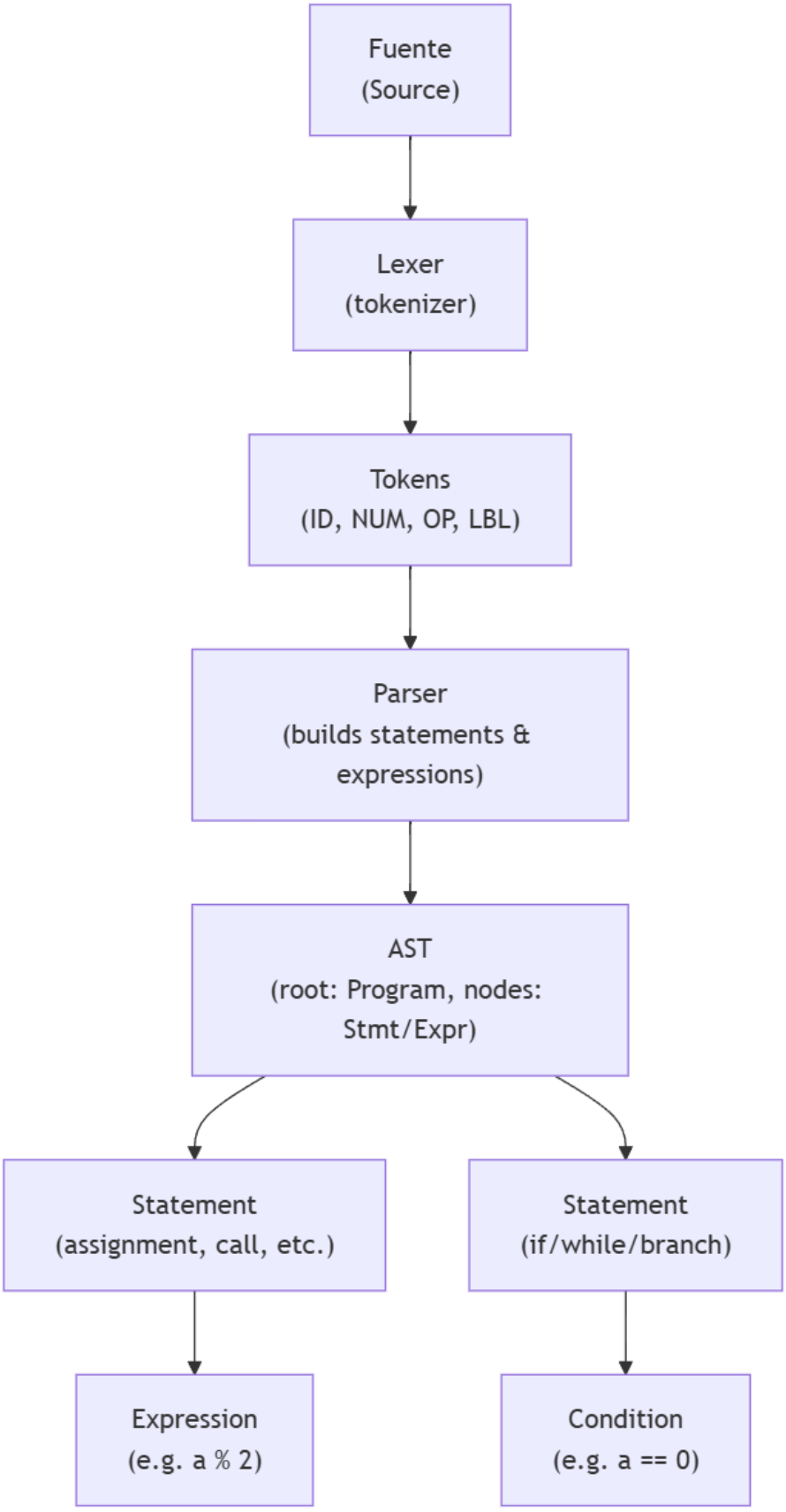


Figura 3: Flujo de tokens → árbol sintáctico → AST

Definiciones: + Parsing: proceso que toma tokens y aplica la gramática para reconocer construcciones (producciones) y crear nodos sintácticos. + Stmt (Statement): unidad ejecutable del programa, ej: asignación, if, while, return. + AST: estructura de datos en forma de árbol que representa la estructura jerárquica del programa.

4.6.3 Optimización y generación final

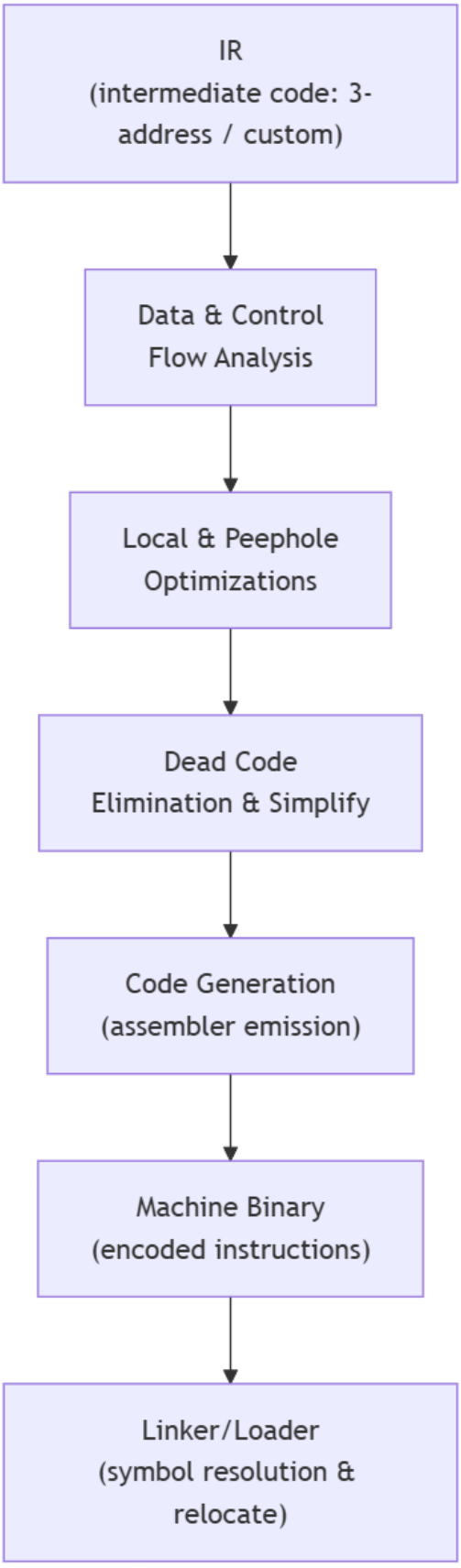


Figura 4: Optimización y generación final

Definiciones: + IR: formato intermedio entre parser y generador que permite análisis y transformaciones. + Análisis de flujo: computa dependencias/alcances (p. ej. dominadores, live-variables). + Optimización local: mejoras dentro de bloques (p. ej. const-fold). + Eliminación de código muerto: borrar instrucciones no usadas. + Codegen/Assembler: genera instrucciones binarias finales. + Binario: secuencia de instrucciones lista para cargar/ejecutar. + Linker: reubica símbolos y combina módulos en ejecutable.

Descripción rápida de cómo esto encaja con el repositorio actual:

- En esta entrega el ensamblador (`compiler/assembler.py`) implementa el generador de instrucción (CODEGEN) y el desensamblador. El `Loader` (`logic/Loader.py`) realiza la función del enlazador/cargador (reubicación simple, escritura a memoria y registro de programas cargados).

4.7 Especificación Léxica del Lenguaje

Keywords

```
- si
- si_no
- mientras
- para
- romper
- continuar
- vacio
- constante
- entero2
- entero4
- entero8
- caracter
- cadena
- con_signo
- sin_signo
- flotante
- doble
- booleano
- funcion
- retornar
- nuevo
- eliminar
```

Identificadores

```
Empiezan por letra mayúscula o minúscula.
- Nombres de variables
- Nombres de funciones
```

Constantes

```
- Enteros: 10, -25, 0xFF (decimal y hexadecimal)
- Flotantes: 3.14, 2.0e3
- Caracteres: 'A', '\n'
- Cadenas: "Hola, Mundo"
- Booleanos: True or False
```

Comentarios

```
- /*
```

- */
- //

Operadores

- Asignación y compuestos: =, +=, -=, *=, /=, %=
- Incremento / decremento: ++, --
- Aritméticos: +, -, *, /, %
- Bitwise: &, |, ^
- Comparación: ==, !=, <, <=, >, >=
- Lógicos: &&, ||

Delimitadores

- Llaves: { }
- Paréntesis: ( )
- Punto y coma: ;
- Coma: ,
- Corchetes: [ ]
- Comillas: " " y ' '

Expresiones regulares

- digito: [0-9]
- numero_entero: -? digito+
- numero_decimal: -? digito+ (\.digito+)?([eE][-+]?{digito}+)?
- caracter: \'([^\'\\]|\\[ntr0'\\])\'
- cadena: \"([^\"\\]|\\.)*\"   \"[^\"\n]*\"
- booleano: verdadero | falso
- comentario: (\/\/[^\n]*|\/*([^*]|*+[^*/])**+\/)
- identificadores: [a-zA-Z_][a-zA-Z0-9_]*

5. Manual Técnico y de Usuario

5.1 Instalación y Configuración

Requisitos del Sistema

Requisitos Mínimos: - Python 3.8 o superior - 512 MB de RAM - 100 MB de espacio en disco - Sistema operativo: Windows 10/11, Linux (Ubuntu 18.04+), macOS 10.14+

Requisitos Recomendados: - Python 3.10+ - 2 GB de RAM - Monitor con resolución mínima 1024x768 - Tarjeta gráfica con soporte para aceleración 2D

Proceso de Instalación

1. **Descargar el proyecto**

```
git clone https://github.com/usuario/maquina_lenguajes.git
cd maquina_lenguajes
```

2. **Verificar Python**

```
python --version  # Debe ser 3.8+
```

3. **Ejecutar simulador**

```
python main.py
```

5.2 Manual de Usuario

Inicio Rápido

Primer Programa

1. Abrir la aplicación ejecutando `python main.py` desde la raíz del repositorio (la misma carpeta que contiene `main.py`)
2. En el editor, escribir:

```
LOADV R1, 10    ; Cargar 10 en registro R1
LOADV R2, 5     ; Cargar 5 en registro R2
ADD R1, R2      ; Sumar R1 + R2
SVIO R1, 0x100  ; Guardar resultado en E/S
SHOWIO 0x100    ; Mostrar resultado
PARAR           ; Terminar programa
```

3. Hacer clic en “Ejecutar”
4. Observar el resultado en el monitor de E/S

Referencia del Lenguaje Assembly

Sintaxis General

```
[ETIQUETA:] INSTRUCCION [OPERANDO1] [, OPERANDO2] [; COMENTARIO]
```

Tipos de Operandos

- **Registros**: R01, R02, . . . , R15 (también R1, R2, . . . , R15)
- **Valores inmediatos**: 123, 0x1A2B, 0b1010
- **Direcciones de memoria**: 0x1000, etiquetas
- **Etiquetas**: LOOP, FIN, DATOS

Instrucciones por Categoría Control de Flujo:

```
PARAR           ; Terminar programa
NOP             ; No operación
JMP etiqueta    ; Salto incondicional
JEQ etiqueta    ; Saltar si Z=1
JNE etiqueta    ; Saltar si Z=0
JMI etiqueta    ; Saltar si N=1
JPL etiqueta    ; Saltar si N=0
```

Operaciones Aritméticas:

```
ADD R1, R2      ; R1 = R1 + R2
SUB R1, R2      ; R1 = R1 - R2
MUL R1, R2      ; R1 = R1 * R2 (sin signo)
MULS R1, R2     ; R1 = R1 * R2 (con signo)
DIV R1, R2      ; R1 = R1 / R2
ADDV R1, 100    ; R1 = R1 + 100
SUBV R1, 50     ; R1 = R1 - 50
INC R1          ; R1 = R1 + 1
DEC R1          ; R1 = R1 - 1
```

Operaciones Lógicas:

```
AND R1, R2      ; R1 = R1 & R2
OR R1, R2       ; R1 = R1 | R2
XOR R1, R2      ; R1 = R1 ^ R2
NOT R1          ; R1 = ~R1
ANDV R1, 0xFF   ; R1 = R1 & 0xFF
ORV R1, 0x80    ; R1 = R1 | 0x80
XORV R1, 0xFF   ; R1 = R1 ^ 0xFF
```

Manejo de Memoria:

```
LOAD R1, 0x1000     ; R1 = Memoria[0x1000]
LOADV R1, 42        ; R1 = 42
STORE R1, R2        ; Memoria[R2] = R1
STOREV R1, 0x2000   ; Memoria[0x2000] = R1
CLEAR R1            ; R1 = 0
```

Entrada/Salida:

```
SVIO R1, 0x100      ; IO[0x100] = R1
LOADIO R1, 0x100    ; R1 = IO[0x100]
SHOWIO 0x100        ; Mostrar IO[0x100]
CLRIO               ; Limpiar dispositivos entrada
RESETIO             ; Reset sistema E/S
```

Ejemplos Prácticos

Programa: Factorial de un Número

```
; Calcular factorial de 5
MAIN:
    LOADV R1, 5         ; n = 5
    LOADV R2, 1         ; factorial = 1

FACTORIAL_LOOP:
    CMPV R1, 0          ; Comparar n con 0
```

```
    JEQ MOSTRAR         ; Si n == 0, mostrar resultado

    MUL R2, R1          ; factorial *= n
    DEC R1              ; n--
    JMP FACTORIAL_LOOP  ; Repetir

MOSTRAR:
    SVIO R2, 0x200      ; Guardar resultado
    SHOWIO 0x200        ; Mostrar factorial = 120
    PARAR               ; Terminar
```

Programa: Búsqueda del Máximo

```
; Encontrar el máximo de tres números
MAIN:
    LOADV R1, 25        ; primer número
    LOADV R2, 18        ; segundo número
    LOADV R3, 31        ; tercer número
    CLEAR R4            ; máximo actual

    ; Comparar R1 con máximo actual
    CMP R1, R4
    JMI NO_ACTUALIZAR1
    LOADV R4, R1        ; R4 = R1

NO_ACTUALIZAR1:
    ; Comparar R2 con máximo actual
    CMP R2, R4
    JMI NO_ACTUALIZAR2
    LOADV R4, R2        ; R4 = R2

NO_ACTUALIZAR2:
    ; Comparar R3 con máximo actual
    CMP R3, R4
    JMI MOSTRAR
    LOADV R4, R3        ; R4 = R3

MOSTRAR:
    SVIO R4, 0x300      ; Guardar máximo
    SHOWIO 0x300        ; Mostrar máximo = 31
    PARAR
```

5.3 Manual Técnico

API del Sistema

Clase CPU

```python
class CPU:
    def __init__(self, memory_size=25000):
        """Inicializar CPU con tamaño de memoria especificado"""

    def load_program(self, program_bytes, start_address=0):
        """Cargar programa en memoria"""
```

```
    def step(self):
        """Ejecutar una instrucción"""

    def run(self):
        """Ejecutar programa hasta PARAR"""

    def reset(self):
        """Reiniciar estado del procesador"""

    def get_state(self):
        """Obtener estado completo del sistema"""
```

Clase Assembler

```
class Assembler:
    def assemble(self, source_code):
        """Ensamblar código fuente a bytecode"""

    def parse_line(self, line):
        """Parsear línea individual de assembly"""

    def resolve_labels(self):
        """Resolver direcciones de etiquetas"""
```

Estructura de Datos Interna

Formato de Instrucción en Memoria

```
# Instrucción de 64 bits almacenada como entero
instruction = (opcode << 48) | (operand1 << 32) | operand2

# Extracción de campos
opcode = (instruction >> 48) & 0xFFFF
rd = (instruction >> 44) & 0xF
rs = instruction & 0xF
immediate = instruction & 0xFFFFFFFFFFFF
```

Estado del Procesador

```
cpu_state = {
    'registers': [0] * 16,      # R00-R15
    'flags': {
        'Z': False,             # Zero flag
        'N': False,             # Negative flag
        'C': False,             # Carry flag
        'V': False              # Overflow flag
    },
    'pc': 0,                    # Program counter
    'memory': Memory(),         # Sistema de memoria
    'io': IOSystem(),           # Sistema E/S
    'running': False            # Estado de ejecución
}
```

Extensibilidad

Agregar Nueva Instrucción

1. Definir opcode en `instruction_set.py`
2. Implementar lógica en `CPU.execute_instruction()`
3. Agregar parsing en `Assembler.parse_instruction()`
4. Actualizar documentación

Ejemplo de Extensión

```python
# Agregar instrucción SQRT (raíz cuadrada)
def execute_sqrt(self, operands):
    rd = operands['rd']
    value = self.registers[rd]
    result = int(math.sqrt(value))
    self.registers[rd] = result
    self.update_flags(result)
```

6. Especificaciones Técnicas

6.1 Arquitectura del Procesador

Especificaciones Generales

Característica	Especificación
Arquitectura	64 bits, RISC
Endianness	Little-endian
Tamaño de palabra	64 bits (8 bytes)
Bus de direcciones	44 bits (16 TB direccionables)
Bus de datos	64 bits
Registros generales	16 (R00-R15)
Tamaño de registro	64 bits
Modelo de memoria	von Neumann unificado

Conjunto de Instrucciones Completo

Instrucciones de Control (7)

Opcode	Mnemónico	Formato	Descripción
0x0000	PARAR	OP	Terminar ejecución
0x0001	NOP	OP	No operación
0x0090	JMP	I	Salto incondicional
0x0091	JEQ	I	Salto si Z=1
0x0092	JNE	I	Salto si Z=0
0x0093	JMI	I	Salto si N=1
0x0094	JPL	I	Salto si N=0

Instrucciones Aritméticas (9)

Opcode	Mnemónico	Formato	Descripción
0x0010	ADD	RR	Rd = Rd + Rs
0x0011	SUB	RR	Rd = Rd - Rs
0x0012	MULS	RR	Rd = Rd x Rs (con signo)
0x0013	MUL	RR	Rd = Rd x Rs (sin signo)
0x0014	DIV	RR	Rd = Rd / Rs
0x0020	ADDV	RI	Rd = Rd + inmediato
0x0021	SUBV	RI	Rd = Rd - inmediato
0x0030	INC	R	Rd = Rd + 1
0x0031	DEC	R	Rd = Rd - 1

Instrucciones Lógicas (9)

Opcode	Mnemónico	Formato	Descripción
0x0040	NOT	R	Rd = ~Rd
0x0041	AND	RR	Rd = Rd & Rs
0x0042	ANDV	RI	Rd = Rd & inmediato
0x0043	OR	RR	Rd = Rd | Rs

Opcode	Mnemónico	Formato	Descripción
0x0044	ORV	RI	Rd = Rd | inmediato
0x0045	XOR	RR	Rd = Rd ^ Rs
0x0046	XORV	RI	Rd = Rd ^ inmediato
0x0050	SHL	R	Shift left lógico
0x0051	SHR	R	Shift right lógico

Instrucciones de Memoria (7)

Opcode	Mnemónico	Formato	Descripción
0x0060	LOAD	RI	Rd = Memoria[direccion]
0x0061	LOADV	RI	Rd = inmediato
0x0062	STORE	RR	Memoria[Rs] = Rd
0x0063	STOREV	RI	Memoria[direccion] = Rd
0x0064	CLEAR	R	Rd = 0
0x0070	CMP	RR	Comparar Rd con Rs
0x0071	CMPV	RI	Comparar Rd con inmediato

Instrucciones de E/S (5)

Opcode	Mnemónico	Formato	Descripción
0x00A0	SVIO	RI	IO[direccion] = Rd
0x00A1	LOADIO	RI	Rd = IO[direccion]
0x00A2	SHOWIO	I	Mostrar IO[direccion]
0x00A3	CLRIO	OP	Limpiar dispositivos entrada
0x00A4	RESETIO	OP	Reset sistema E/S

Instrucciones de Flags (8)

Opcode	Mnemónico	Formato	Descripción
0x0080	CZF	OP	Clear Zero Flag
0x0081	SZF	OP	Set Zero Flag
0x0082	CNF	OP	Clear Negative Flag
0x0083	SNF	OP	Set Negative Flag
0x0084	CCF	OP	Clear Carry Flag
0x0085	SCF	OP	Set Carry Flag
0x0086	CDF	OP	Clear Overflow Flag
0x0087	SDF	OP	Set Overflow Flag

Saltos Condicionales Extendidos (2)

Opcode	Mnemónico	Formato	Descripción
0x0095	JMPCMY	I	Salto si C=1
0x0096	JMPCMN	I	Salto si C=0

Total: 47 instrucciones implementadas

6.2 Sistema de Memoria

Configuración de Memoria

```
MEMORY_LAYOUT = {
    'PROGRAM_START': 0x0000,        # Inicio de código
    'DATA_START': 0x4000,           # Inicio de datos
    'STACK_START': 0x6000,          # Inicio de pila
    'IO_START': 0x8000,             # E/S mapeada
    'MEMORY_SIZE': 25000            # Tamaño total
}
```

Mapa de Memoria

```
0x0000 +-------------------------+
       |   Código Programa       |
       |        (16 KiB)         |
0x4000 +-------------------------+
       |    Área de Datos        |
       |        (8 KiB)          |
0x6000 +-------------------------+
       |        Pila             |
       |        (8 KiB)          |
0x8000 +-------------------------+
       |    E/S Mapeada          |
       |  (resto hasta 64KiB)    |
0xFFFF +-------------------------+
```

6.3 Sistema de E/S

Direcciones de E/S Reservadas

Dirección	Propósito	Acceso
0x8000-0x80FF	Pantalla virtual	R/W
0x8100-0x81FF	Teclado virtual	R
0x8200-0x82FF	Almacenamiento temporal	R/W
0x8300-0x83FF	Debugging/Logging	W

Protocolo de Comunicación

```
# Escribir a dispositivo
SVIO R1, 0x8000     # IO[0x8000] = R1

# Leer de dispositivo
LOADIO R1, 0x8100   # R1 = IO[0x8100]

# Mostrar en pantalla
SHOWIO 0x8000       # Mostrar contenido de IO[0x8000]
```

7. Documentación de Experimentación y Resultados

Se presentan los experimentos realizados sobre el analizador lexico desarrollado, con el objetivo de probar y validar el correcto funcionamiento del analizador en la identificación de categorias lexicas y su clasificación en tokens. Para esto en cada escenario se procesa una cadena de caracteres correspondiente a un un programa en alto nivel, que debe ser aceptado por el analizador lexico.

7.1 Escenario 1

Para este escenario se toma la cadena de caracteres correspondiente al un codigo en alto nivel que determina si un entero es par o impar

```
funcion entero4 espar(entero4 a) {
    a = a%2;
    booleano resultado = 0;
    si(a%2 == 0){
        resultado = 1;
    }
    si_no{
        resultado = 0;
    }
    retornar resultado;
}
```

Retornando como resultado del analisis

```
LexToken(FUNCION,'funcion',3,30)
LexToken(ENTERO4,'entero4',3,38)
LexToken(ID,'espar',3,46)
LexToken(PARIZQ,'(',3,51)
LexToken(ENTERO4,'entero4',3,52)
LexToken(ID,'a',3,60)
LexToken(PARDER,')',3,61)
LexToken(LLAVEIZQ,'{',3,63)
LexToken(ID,'a',4,73)
LexToken(ASIGNAR,'=',4,75)
LexToken(ID,'a',4,77)
LexToken(MOD,'%',4,78)
LexToken(ENTERO,2,4,79)
LexToken(PUNTOCOMA,';',4,80)
LexToken(BOOLEANO,'booleano',5,90)
LexToken(ID,'resultado',5,99)
LexToken(ASIGNAR,'=',5,109)
LexToken(ENTERO,0,5,111)
LexToken(PUNTOCOMA,';',5,112)
LexToken(SI,'si',6,122)
LexToken(PARIZQ,'(',6,124)
LexToken(ID,'a',6,125)
LexToken(MOD,'%',6,126)
LexToken(ENTERO,2,6,127)
LexToken(IGUAL,'==',6,129)
LexToken(ENTERO,0,6,132)
LexToken(PARDER,')',6,133)
LexToken(LLAVEIZQ,'{',6,134)
LexToken(ID,'resultado',7,148)
LexToken(ASIGNAR,'=',7,158)
```

```
LexToken(ENTERO,1,7,160)
LexToken(PUNTOCOMA,';',7,161)
LexToken(LLAVEDER,'}',8,171)
LexToken(SINO,'si_no',9,181)
LexToken(LLAVEIZQ,'{',9,186)
LexToken(ID,'resultado',10,200)
LexToken(ASIGNAR,'=',10,210)
LexToken(ENTERO,0,10,212)
LexToken(PUNTOCOMA,';',10,213)
LexToken(LLAVEDER,'}',11,223)
LexToken(RETORNAR,'retornar',12,233)
LexToken(ID,'resultado',12,242)
LexToken(PUNTOCOMA,';',12,251)
LexToken(LLAVEDER,'}',13,257)
```

7.2 Escenario 2

Para este escenario se toma la cadena de caracteres correspondiente al un codigo del algoritmo de euclides

```
funcion entero4 euclides(entero4 a, entero4 b) {
    mientras(b != 0) {
        entero4 temp = b;
        b = a % b;
        a = temp;
    }
    retornar a;
}
```

Retornando como resultado del analisis

```
LexToken(FUNCION,'funcion',3,30)
LexToken(ENTERO4,'entero4',3,38)
LexToken(ID,'euclides',3,46)
LexToken(PARIZQ,'(',3,54)
LexToken(ENTERO4,'entero4',3,55)
LexToken(ID,'a',3,63)
LexToken(COMA,',',3,64)
LexToken(ENTERO4,'entero4',3,66)
LexToken(ID,'b',3,74)
LexToken(PARDER,')',3,75)
LexToken(LLAVEIZQ,'{',3,77)
LexToken(MIENTRAS,'mientras',4,87)
LexToken(PARIZQ,'(',4,95)
LexToken(ID,'b',4,96)
LexToken(DISTINTO,'!=',4,98)
LexToken(ENTERO,0,4,101)
LexToken(PARDER,')',4,102)
LexToken(LLAVEIZQ,'{',4,104)
LexToken(ENTERO4,'entero4',5,118)
LexToken(ID,'temp',5,126)
LexToken(ASIGNAR,'=',5,131)
LexToken(ID,'b',5,133)
LexToken(PUNTOCOMA,';',5,134)
LexToken(ID,'b',6,148)
LexToken(ASIGNAR,'=',6,150)
LexToken(ID,'a',6,152)
```

```
LexToken(MOD,'%',6,154)
LexToken(ID,'b',6,156)
LexToken(PUNTOCOMA,';',6,157)
LexToken(ID,'a',7,171)
LexToken(ASIGNAR,'=',7,173)
LexToken(ID,'temp',7,175)
LexToken(PUNTOCOMA,';',7,179)
LexToken(LLAVEDER,'}',8,189)
LexToken(RETORNAR,'retornar',9,199)
LexToken(ID,'a',9,208)
LexToken(PUNTOCOMA,';',9,209)
LexToken(LLAVEDER,'}',10,215)
```

7.3 Escenario 3

Para este escenario se toma la cadena de caracteres correspondiente al un codigo que calcula el determinante de una matriz cuadrada 2x2

```
funcion entero4 determinante(entero4 x1, entero4 x2, entero4 y1, entero4 y2){
    entero8 det = (x1 * y2) - (x2 * y1);
    retornar det;
}
```

Retornando como resultado el analisis

```
LexToken(FUNCION,'funcion',3,30)
LexToken(ENTERO4,'entero4',3,38)
LexToken(ID,'determinante',3,46)
LexToken(PARIZQ,'(',3,58)
LexToken(ENTERO4,'entero4',3,59)
LexToken(ID,'x1',3,67)
LexToken(COMA,',',3,69)
LexToken(ENTERO4,'entero4',3,71)
LexToken(ID,'x2',3,79)
LexToken(COMA,',',3,81)
LexToken(ENTERO4,'entero4',3,83)
LexToken(ID,'y1',3,91)
LexToken(COMA,',',3,93)
LexToken(ENTERO4,'entero4',3,95)
LexToken(ID,'y2',3,103)
LexToken(PARDER,')',3,105)
LexToken(LLAVEIZQ,'{',3,106)
LexToken(ENTERO8,'entero8',4,116)
LexToken(ID,'det',4,124)
LexToken(ASIGNAR,'=',4,128)
LexToken(PARIZQ,'(',4,130)
LexToken(ID,'x1',4,131)
LexToken(MULT,'*',4,134)
LexToken(ID,'y2',4,136)
LexToken(PARDER,')',4,138)
LexToken(MENOS,'-',4,140)
LexToken(PARIZQ,'(',4,142)
LexToken(ID,'x2',4,143)
LexToken(MULT,'*',4,146)
LexToken(ID,'y1',4,148)
LexToken(PARDER,')',4,150)
```

```
LexToken(PUNTOCOMA,';',4,151)
LexToken(RETORNAR,'retornar',5,161)
LexToken(ID,'det',5,170)
LexToken(PUNTOCOMA,';',5,173)
LexToken(LLAVEDER,'}',6,179)
```

7.4 Escenario 4

Para este escenario se toma la cadena de caracteres correspondiente al un codigo que calcula el determinante de una matriz cuadrada 2x2

```
funcion entero4 abs(entero4 con_signo a){
    si(a >= 0){
        retornar a;
    }
    si_no{
        absoluto = -1*a;
        retornar absoluto;
    }
}
```

Retornando como resultado el analisis

```
LexToken(FUNCION,'funcion',3,30)
LexToken(ENTERO4,'entero4',3,38)
LexToken(ID,'abs',3,46)
LexToken(PARIZQ,'(',3,49)
LexToken(ENTERO4,'entero4',3,50)
LexToken(CON_SIGNO,'con_signo',3,58)
LexToken(ID,'a',3,68)
LexToken(PARDER,')',3,69)
LexToken(LLAVEIZQ,'{',3,70)
LexToken(SI,'si',4,80)
LexToken(PARIZQ,'(',4,82)
LexToken(ID,'a',4,83)
LexToken(MAYORIGUAL,'>=',4,85)
LexToken(ENTERO,0,4,88)
LexToken(PARDER,')',4,89)
LexToken(LLAVEIZQ,'{',4,90)
LexToken(RETORNAR,'retornar',5,104)
LexToken(ID,'a',5,113)
LexToken(PUNTOCOMA,';',5,114)
LexToken(LLAVEDER,'}',6,124)
LexToken(SINO,'si_no',7,134)
LexToken(LLAVEIZQ,'{',7,139)
LexToken(ID,'absoluto',8,153)
LexToken(ASIGNAR,'=',8,162)
LexToken(MENOS,'-',8,164)
LexToken(ENTERO,1,8,165)
LexToken(MULT,'*',8,166)
LexToken(ID,'a',8,167)
LexToken(PUNTOCOMA,';',8,168)
LexToken(RETORNAR,'retornar',9,182)
LexToken(ID,'absoluto',9,191)
LexToken(PUNTOCOMA,';',9,199)
LexToken(LLAVEDER,'}',10,209)
```

```
LexToken(LLAVEDER,'}',11,223)
```

Análisis

Tras el análisis de los resultados obtenidos en los cuatro casos de prueba, se pudo comprobar que el analizador léxico identificó correctamente las categorías léxicas asociadas a cada una de las subcadenas en las cadenas de entrada. En todos los casos, los tokens generados coincidieron con los valores esperados de acuerdo con las reglas definidas en la gramática léxica del lenguaje.

Esto demuestra que las expresiones regulares implementadas en las definiciones de tokens son adecuadas para reconocer las estructuras básicas del lenguaje, como identificadores, palabras reservadas, operadores y delimitadores.

8. Conclusiones

El **Simulador Atlas CPU** representa una herramienta educativa completa que cumple exitosamente con los objetivos establecidos:

Logros Principales

1. **Funcionalidad Completa**: 47 instrucciones implementadas y validadas
2. **Validación Exhaustiva**: Algoritmos clásicos verificados matemáticamente

3. **Documentación Integral**: Manuales técnicos y educativos completos
4. **Interfaz Intuitiva**: GUI diseñada para facilitar el aprendizaje
5. **Arquitectura Sólida**: Diseño modular y extensible

Impacto Educativo

- **Experimentación**: Ambiente seguro para pruebas y errores
- **Comprensión**: Visualización directa de conceptos abstractos

2025 - Grupo D - Universidad Nacional de Colombia
Simulador Atlas CPU - Hexacore Technologies Repositorio GitHub: https://github.com/JulianGomezN/maquina_lenguajes