



*INSTITUTO POLITECNICO NACIONAL*

*Centro de Investigación en Ciencia Aplicada y Tecnología  
Avanzada*

*Unidad Querétaro*

Desarrollo de un Entorno de Desarrollo  
Integrado (IDE) para el procesador didáctico

SOPHIA

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRO EN TECNOLOGIA AVANZADA

P R E S E N T A

RAYMUNDO RAMIREZ PEDRAZA

QUERETARO, QRO.

MAYO, 2016







**INSTITUTO POLITÉCNICO NACIONAL**  
**SECRETARÍA DE INVESTIGACIÓN Y POSGRADO**

**CARTA CESIÓN DE DERECHOS**

En la Ciudad de México, D.F. el día 27 del mes de mayo del año 2016, el que suscribe Raymundo Ramírez Pedraza alumno del Programa de Maestría en Tecnología Avanzada, con número de registro B140602, adscrito al CICATA-IPN Unidad Querétaro, manifiesto que es el autor intelectual del presente trabajo de Tesis bajo la dirección del Dr. Antonio Hernández Zavala y cede los derechos del trabajo titulado “Desarrollo de un Entorno de Desarrollo Integrado (IDE) para el procesador didáctico SOPHIA”, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a las siguientes direcciones: anhernandezz@ipn.mx o raymundo.r.p@gmail.com. Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Raymundo Ramírez Pedraza

Nombre y firma del alumno



# INSTITUTO POLITÉCNICO NACIONAL SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

## ACTA DE REVISIÓN DE TESIS

En la Ciudad de Querétaro, Qro. siendo las 14:14 horas del día 27 del mes de Mayo del 2016 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación de \_\_\_\_\_ para examinar la tesis titulada:

"Desarrollo de un Entorno de Desarrollo Integrado (IDE) para el procesador didáctico SOPHIA"

Presentada por el alumno:

Ramírez	Pedraza	Raymundo
Apellido paterno	Apellido materno	Nombre(s)
		Con registro: B 1 4 0 6 0 2

aspirante de:

Maestría en Tecnología Avanzada

Después de intercambiar opiniones, los miembros de la Comisión manifestaron **APROBAR LA TESIS**, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

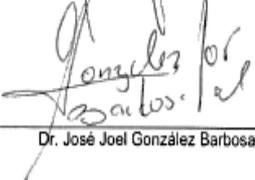
### LA COMISIÓN REVISORA

Director(a) de tesis

  
Dr. Antonio Hernández Zavala

  
Dr. Juan Bautista Hurtado Ramos

  
Dr. Jorge Adalberto Huerta Ruelas

  
Dr. José Joel González Barbosa

  
Dr. Julio César Sosa Savedra

PRESIDENTE DEL COLEGIO DE PROFESORES

  
  
Dra. Eva González Jasso  
UNIDAD QUERÉTARO  
DIRECCIÓN

## DEDICATORIA

A Dios por haberme puesto en este camino.

A mi mamá porque sin ella no estaría aquí.

A mi hermana por siempre estar a mi lado para apoyarme.

A mi hermano por ser un ejemplo a seguir

## AGRADECIMIENTOS

A CONACYT por el apoyo otorgado durante mi estancia en CICATA-QRO.

Al Centro de Investigación en Computación (CIC-IPN) y los doctores que me recibieron en mi estancia en su Centro de Investigación.

A la Dra. Eva González Jasso, directora de este Centro de Investigación por el apoyo brindado durante mi estancia en la maestría.

Al Dr. Antonio Hernandez Zavala por la confianza y apoyo recibido durante el desarrollo del proyecto.

A los miembros de mi comité por su apoyo, asesoría y consejos que me han ayudado a crecer en todos los ámbitos.

A mis amigos de la preparatoria y universidad que pese a la distancia siempre han estado presentes. Y por supuesto a los amistades que se han hecho durante la maestría: Silvia, Rodrigo, Julian y Martin, gracias por estar en mí día a día, por el apoyo y los buenos momentos que se han pasado y por los estoy seguro que vendrán.

## **RESUMEN**

A nivel mundial dentro del área de computación en los cursos de arquitectura de computadoras los recursos didácticos utilizados no favorecen la correcta y completa asimilación de los conceptos. Algunas universidades desarrollan sus propios recursos (principalmente simuladores y hardware), en países desarrollados, cada alumno cuenta con su propia herramienta; en cambio en los cursos que se imparten en las instituciones mexicanas, no se cuenta con una herramienta adecuada para facilitar la comprensión de la estructura interna de una computadora. Por esa razón fue diseñado el procesador con fines didácticos llamado SOPHIA, que ayuda a mostrar cómo se comunican los bloques funcionales del procesador. Para hacer uso de este, se tiene que programar en lenguaje máquina. De esta manera existe la necesidad de una herramienta que facilite la interacción con el procesador, en este proyecto se propone el desarrollo de un compilador para el procesador SOPHIA; con la finalidad de contar con una herramienta didáctica amigable y con los resultados obtenidos en el desarrollo del compilador, también se desarrollo un simulador, obteniendo como resultado un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés, Integrated Development Environment), que facilita la abstracción de los conceptos del curso de arquitectura de computadoras.

## **ABSTRACT**

Worldwide within the area of computing, specifically in computer architecture courses there aren't many teaching tools to be used. The few tools that exist don't help to the correct and complete assimilation of concepts. Some Universities develop their own resources (mainly simulators and hardware). In developed countries, each student has their own tool. In other hand, Mexican institutions there aren't a suitable tool to facilitate understanding of the internal structure of a computer. For that reason the processor SOPHIA was developed for teaching purposes, which helps to show how the functional blocks of the processor communicate between their. However, to can use this, you must be programmed in machine language (0, 1). Thus there is a need for a tool to facilitate interaction with the processor, in this project is proposed the development of a compiler for the processor SOPHIA; in order to have a friendly educational tool that facilitates abstraction of the concepts of computer architecture course and in base the results obtained in the development of the compiler, also was developed a simulator; getting a IDE (Integrated Development Environment). This didactic tool helps in abstracting the concepts of computer architecture course.

## ÍNDICE GENERAL

Índice de figuras .....	IX
Índice de tablas .....	XII
Capítulo 1 Introducción .....	1
1.1 Descripción del problema.....	2
1.2 Justificación.....	3
1.3 Objetivo general.....	3
1.4 Objetivos específicos.....	4
1.5 Estructura de la tesis .....	4
Capítulo 2. Estado del arte .....	5
2.1 Procesadores didácticos .....	8
2.2 Compiladores y herramientas de desarrollo.....	19
2.3 Entorno de desarrollo integrado .....	23
Capítulo 3. Marco teórico.....	26
3.1 Procesador didáctico sophia.....	26
3.1.1 Características y arquitectura .....	26
3.1.2 Conjunto de instrucciones.....	28
3.1.3 Modos de direccionamiento .....	28
3.2 Compiladores.....	34
3.2.1 Análisis léxico.....	37
3.2.1.1 Tabla de transiciones .....	41
3.2.2 Análisis sintáctico .....	44
3.2.2.1 Forma normal de chomsky .....	49
3.2.2.2 Principio del preanálisis .....	51
3.2.2.3 Forma normal de backus .....	53
3.2.3 Análisis semántico .....	54
3.2.4 Generación y optimización de código intermedio.....	55
3.2.5 Generación de código máquina .....	56
3.2.6 Manejador de errores .....	56
3.2.7 Tabla de símbolos .....	58
3.3 Simulador .....	59
Capítulo 4. Entorno de desarrollo integrado.....	60
Capítulo 5. Experimentación y resultados .....	72

<b>Capítulo 6. Conclusión.....</b>	<b>78</b>
<b>Trabajo futuro .....</b>	<b>78</b>
<b>Referencias:.....</b>	<b>79</b>

## ÍNDICE DE FIGURAS

Figura 3. 1 Arquitectura del procesador SOPHIA. ....	27
Figura 3. 2 Direccionamiento directo a memoria, donde el código de operación define el tipo de salto y dato constante el número de la próxima instrucción a ejecutar. ....	28
Figura 3. 3 Direccionamiento instrucciones inmediatas, el dato constante actualiza el valor del registro.....	31
Figura 3. 4 Direccionamiento dato directo desde memoria de datos, el dato constante determina la localidad de memoria la memoria de datos a la que se accederá para obtener su valor, mismo que ha de ser cargado en el registro destino. ....	31
Figura 3. 5 Direccionamiento de Entrada/Salida, el dato constante determina la localidad de memoria la memoria de datos a la que se almacenará el valor del registro.....	32
Figura 3. 6 Direccionamiento directo con dos registros, el valor del registro fuente es almacenado en el registro destino. ....	32
Figura 3. 7 Direccionamiento acceso indirecto a memoria mediante un registro, registro fuente es un apuntador a memoria de datos, el código de operación define si se lee o escribe el valor de la memoria de datos en el registro destino. ....	33
Figura 3. 8 Direccionamiento acceso directo un solo registro, el resultado de la ALU es almacenado en el registro destino. ....	33
Figura 3. 9 Etapas para el desarrollo de un compilador, donde la etapa de análisis esta compuesto por: analizador léxico, sintáctico y semántico y la etapa de síntesis lo compone la generación de código intermedio, su optimización y la obtención de código objeto.....	36
Figura 3. 10 Clasificación de los diferentes lenguajes determinada por Chomsky. ....	37
Figura 3. 11 Gramática que produce frases complejas y completas en el lenguaje español.....	38
Figura 3. 12 Gramática regular correspondiente al lenguaje $L(G)=\{ x^m y^n: m, n \in N\}$ .....	40

Figura 3. 13	Autómata finito que reconoce identificadores válidos. ....	41
Figura 3. 14	Algoritmo general para generar un autómata finito determinista que acepte identificadores válidos. ....	42
Figura 3. 15	Algoritmo del autómata finito que reconoce identificadores válidos. .	43
Figura 3. 16	Estructura de las instrucciones Si, Si-SiNo. ....	45
Figura 3. 17	Autómata de pila que reconoce la expresión $(xy)^*$ . ....	46
Figura 3. 18	Autómata de pila que reconoce cadenas palíndromas de longitud par. .....	47
Figura 3. 19	Gramática que produce la expresión $(xy)^*$ . ....	49
Figura 3. 20	Algoritmo que convierte una GIC a una GFNC. ....	50
Figura 3. 21	Algoritmo para realizar un analizador sintáctico LL(k). ....	52
Figura 3. 22	Algoritmo para realizar un analizador sintáctico LR(k). ....	53
Figura 4. 1	Expresiones regulares utilizadas para generar el analizador léxico. ....	60
Figura 4. 2	Autómata finito determinista correspondiente al análisis léxico. ....	61
Figura 4. 3 a)	Gramática BNF instrucciones de control del compilador. ....	62
Figura 4. 4	Analizador sintáctico mediante un autómata de pila. ....	63
Figura 4. 5	Fragmento de código correspondiente a la generación de código intermedio. ....	65
Figura 4. 6	Interfaz del compilador. ....	68
Figura 4. 7	Botón que permite crear un nuevo programa. ....	68
Figura 4. 8	Botón que permite abrir un programa ya existente. ....	69
Figura 4. 9	Botón que permite guardar el código escrito en el área de trabajo . ....	69
Figura 4. 10a)	Botón que permite copiar el texto seleccionado. ....	69
Figura 4. 11	Botón que permite cortar el texto seleccionado. ....	69
Figura 4. 12	Botón que pega el texto que se haya copiado o cortado previamente. ....	69
Figura 4. 13	Botón compilar. ....	70
Figura 4. 14	Botón ejecutar. ....	70
Figura 4. 15	Botón simular. ....	71
Figura 4. 16	Interfaz del simulador. ....	71

Figura 5. 1 a) Código error imprimir y declaración de variable. b) Mensajes de errores. ....	72
Figura 5. 2 a) Código error mientras y declaración de variable por falta de “;”. b) Mensajes de errores.....	73
Figura 5. 3 a) Código error símbolo no identificado. b) Mensajes de errores.....	74
Figura 5. 4 a) Código error falta de símbolo de concatenación. b) Mensajes de errores. ....	74
Figura 5. 5 a) Código error falta de token aritmético. b) Mensajes de errores...	75
Figura 5. 6 a) Ejecución de código sin errores. b) Resultado de la ejecución de código sin errores.....	76
Figura 5. 7 a) Ejecución del simulador.....	77

## ÍNDICE DE TABLAS

Tabla 2. 1 Métodos de enseñanza utilizados en cursos de arquitectura de computadoras.....	7
Tabla 2. 2 Procesadores didácticos desarrollados en universidades a nivel mundial.....	9
Tabla 3. 1 Conjunto de instrucciones del procesador SOPHIA .....	29
Tabla 3. 2 Tabla de transiciones correspondiente al autómata de la Figura 3.13 .	43
Tabla 3. 3 Traducción de una instrucción de alto nivel a lenguaje ensamblador...	55
Tabla 3. 4 Tabla de símbolos del compilador.....	58
Tabla 4. 1 Traducción de una asignación simple en alto nivel a lenguaje ensamblador.....	65
Tabla 4. 2 Traducción de una instrucción de alto nivel a lenguaje ensamblador...	66
Tabla 4. 3 Descripción del uso de los registros del procesador SOPHIA. ....	67
Tabla 4. 4 Código máquina generado correspondiente a una asignación de una suma a una variable.....	67

# CAPÍTULO 1 INTRODUCCIÓN

Hoy en día las computadoras son una herramienta utilizada para realizar cualquier tipo de cálculos en casi cualquier área: como en el sector de telecomunicaciones, las finanzas, el ramo automotriz, los sistemas de control, entre otras.

Existen diferentes tipos de computadoras, cuya principal diferencia es la cantidad de información que puede procesar. Debido a que la información es agrupada en paquetes de 8 bits, existen procesadores de 8, 16, 32, y 64 bits. Siendo estos dos últimos los más comunes, pues se encuentran en las computadoras de escritorio, laptop, notebook, tabletas, o cualquier otra computadora personal. Las computadoras de 16-bits son usadas comúnmente en el área de las telecomunicaciones y aplicaciones industriales.

A pesar de que la mayoría de las personas no conoce la existencia de procesadores de 8 bits, son los procesadores más vendidos en el mundo, debido a que estos pueden ser usadas como controladores de tareas computacionales sencillas. Entonces se puede puntualizar, las computadoras más simples son aquellas que poseen los procesadores de 8-bits.

Por la necesidad de contar con una herramienta didáctica para los cursos de arquitectura de computadoras se desarrolló el procesador SOPHIA. El procesador posee un bus de datos de 8 bits y un conjunto reducido de instrucciones (*RISC*). Incluye un conjunto de instrucciones consistentes y todos los bloques funcionales, tales como *memoria de datos* y *memoria de programa*, una *unidad lógica aritmética* sencilla (*ALU*) para operaciones básicas. Su modo de funcionamiento es la ejecución multiciclo dada su facilidad de visualización.

El procesador SOPHIA se implementó en un *FPGA* de *Xilinx Spartan-3E* para ser una biblioteca (softcore) que se puede usar como herramienta didáctica en los cursos de arquitectura de computadoras, cuya intención principal es proporcionar a los estudiantes una herramienta flexible pero consistente para la comprensión de arquitectura de computadoras desde sus fundamentos, siendo la base para resolver cualquier tipo de problemas didácticos dentro de los cursos de arquitectura de computadoras, como los pueden ser: realizar actividades prácticas donde se identifiquen y manejen componentes de hardware y comprender su funcionamiento, mediante la formalización de conceptos a través de la observación y reflexión para la posterior resolución de problemas. Este procesador está basado en la arquitectura *Harvard* que consiste en separar la memoria necesaria para el programa y otra más para almacenamiento de datos.

Todo hardware requiere de un software para funcionar. Debido a que ya se cuenta con el procesador (hardware) entonces, en el presente trabajo se realizará el desarrollo de un Entorno de Desarrollo Integrado IDE (software) mismo que será utilizado como interfaz entre el usuario y el procesador, permitiendo el desarrollo de programas y haciendo uso del conjunto de instrucciones que contiene el procesador. Para que sea posible desarrollar este IDE es necesario conocer la arquitectura de la computadora con la que se va a trabajar, las etapas que se deben tomar en cuenta y los recursos que se necesitan para la ejecución de los programas.

## **1.1 DESCRIPCIÓN DEL PROBLEMA**

A nivel mundial dentro del área de computación en los cursos de arquitectura de computadoras los recursos didácticos utilizados no favorecen la correcta y completa asimilación de los conceptos, siendo estos la base para la transversalidad que existe con algunos otros cursos. Ha habido universidades que desarrollaron sus propios recursos (principalmente simuladores y hardware) para ser implementados en asignaturas de la misma universidad, en países desarrollados hay suficiente presupuesto para proporcionar a cada alumno su simulador y/o hardware para la comprensión de los conceptos a través de llevarlos a la práctica.

En cambio en los cursos de arquitectura de computadoras que se imparten en las instituciones de nivel superior en México, no se cuenta con una herramienta adecuada para facilitar la comprensión de la estructura interna de una computadora, porque no se enseña la fabricación de circuitos integrados, pues en ocasiones no se cuenta con las herramientas adecuadas para hacerlo a diferencia de países desarrollados o simplemente no es una prioridad, llevando a utilizar métodos como el papel. Es por esto que existe la necesidad de desarrollar tecnología propia.

Por esta razón se desarrolló el procesador SOPHIA, el cual busca que los alumnos tengan una mayor comprensión del funcionamiento interno de una computadora.

## **1.2 JUSTIFICACIÓN**

Debido a que en la actualidad existen pocas herramientas prácticas para ayudar a los estudiantes a asimilar mejor los conocimientos de los cursos de arquitectura de computadoras, en el presente trabajo se desarrollará un IDE (Entorno de Desarrollo Integrado) que servirá como intermediario entre el usuario y la computadora para que el procesador didáctico SOPHIA sea una herramienta práctica en cursos de arquitectura de computadoras.

## **1.3 OBJETIVO GENERAL**

- Desarrollar un Entorno de Desarrollo Integrado para que el procesador SOPHIA pueda ser usado como herramienta didáctica por los alumnos de los cursos de arquitectura de computadoras a nivel superior y posgrado.

#### **1.4 OBJETIVOS ESPECÍFICOS**

- Definir los elementos léxicos, la estructura del programa y el diccionario de datos.
- Realizar y validar la fase de análisis del compilador para informar al usuario sobre los posibles errores, así como su tipo y ubicación.
- Realizar y validar la fase de síntesis del compilador para generar el código máquina.
- Implementar el IDE y validar su funcionamiento.

#### **1.5 ESTRUCTURA DE LA TESIS**

En el capítulo 2 se presenta el estado del arte respecto al trabajo que ha sido realizado del desarrollo de procesadores con fines didácticos, así como de compiladores y herramientas de desarrollo para los mismos y la creación de entornos de desarrollo integrado. En el capítulo 3 se describe el marco teórico donde se presenta la teoría de autómatas y lenguajes formales, basados en la jerarquía de Chomsky y el principio del preanálisis. En el capítulo 4 se muestra el desarrollo del IDE para el procesador SOPHIA. En el capítulo 5 se dan a conocer las pruebas y resultados. Finalmente en el capítulo 6 se presenta la conclusión y se da una descripción del trabajo futuro.

## CAPÍTULO 2. ESTADO DEL ARTE

En 1950 fue desarrollada la primera computadora de 2 bits por Edmund Berkeley y Robert Jensen, representando esto la punta de lanza para que el desarrollo de procesadores educativos y métodos de enseñanza fueran estudiados para explicar arquitectura de computadoras y asegurar la comprensión de las funciones de cada uno de sus bloques funcionales (partes que lo conforman) [1].

Los diferentes métodos de enseñanza que se han desarrollado desde 1950, se pueden clasificar en 5 categorías debido a su naturaleza y a las necesidades que se tienen, entender cada uno de sus bloques funcionales, el desarrollo de un procesador, la transferencia entre sus bloques, a continuación se muestran las 5 categorías de los métodos didácticos y las diferentes aplicaciones que se desarrollaron para cada una de ellas:

**1.-Papel:** El Dr. Stuart Madnick diseñó la "Little Man Computer LMC" en 1965 [2], para enseñar a sus estudiantes. Otro enfoque de esta técnica, es la computadora "CARDIAC" desarrollado por David Hagelbarger y Saúl Fingerman de los Laboratorios Bell en 1968 [3], es un procesador que en la actualidad se implementa en un software simulador. Por último, el "Paper Processor", desarrollado por Saito Yutaka en 2010 [4]. Estas aplicaciones se han tomado como base para desarrollar versiones modernas sin perder la esencia de mostrar las características básicas de las computadoras, entender su funcionamiento, pero sin dejar de lado la programación de las mismas.

**2.- Hardware:** En este enfoque es común el uso de componentes digitales MSI tales como TTL para construir un equipo como Glen G. Langdon en 1982[5]; procesador "SC-16" de 16 bits [6]; el procesador "DLX" de 32 bits desarrollado por Hennessy-Patterson en 1990 [7]. El procesador de 16 bits "PISC" de Bradford Rodríguez en 1994 [8]. En los cursos de arquitectura de computadoras es usado este enfoque para sincronizar la transferencia de datos entre los bloques funcionales.

**3.- Simulador:** Existen diversas herramientas de simulación que se enfocan a la enseñanza de arquitectura de computadoras [9]–[15], estas permiten la ejecución de un código y se tiene como resultado la visualización de la transferencia de datos entre los bloques funcionales del procesador de forma gráfica.

**4.- Lenguajes de descripción de Hardware (HDL):** Se han desarrollado muchos tipos de procesadores didácticos para este enfoque, existen versiones de 8, 16 y 32 bits [16]–[22]. Esta opción permite la construcción física de una computadora simple mediante el uso de dispositivos FPGA programados por un HDL. La principal ventaja es que el tiempo requerido para desarrollar un procesador es bastante corto, pero la principal desventaja es la falta de comprensión en el flujo de datos inducida por la programación.

**5.- Bloques lógicos:** Este enfoque permite a los estudiantes comprender plenamente un procesador a partir de sus fundamentos, mediante la integración y la sincronización de sus bloques funcionales. Varios autores prefieren este método para sus cursos [19], [23]–[28], pues consideran que esta es la mejor opción de desarrollo, permitiendo la creación de cada bloque funcional para probarlo y lograr una mejor integración [29].

En la Tabla 2.1 se muestran los métodos de enseñanza en los cursos de arquitectura de computadoras donde se muestra una pequeña descripción de cada uno de ellos, sus ventajas y desventajas.

**Tabla 2. 1 Métodos de enseñanza utilizados en cursos de arquitectura de computadoras.**

<b>Método</b>	<b>Descripción</b>	<b>Ventajas</b>	<b>Desventajas</b>
Papel	Primera técnica utilizada.	Costo, simpleza.	No se podía entender el verdadero propósito de una computadora.
Hardware	Este tipo de equipo se construye utilizando cualquier tecnología existente.	Sincronizar la transferencia de datos entre los bloques funcionales.	El tiempo invertido en el cableado.
Simulador	Permite la ejecución de un código para ver la funcionalidad de la computadora.	La mayoría de estos son entornos gráficos, se pueden ver todos los bloques funcionales y la transferencia de datos entre ellos.	Sólo permite ver el funcionamiento completo del procesador, no así de cada unidad funcional.
Lenguajes de descripción (HDL)	Permite la construcción física de una computadora simple.	Tiempo de desarrollo corto.	Falta de comprensión en el flujo de datos.
Bloques lógicos.	Diseñado para dispositivos de lógica programable de cada unidad funcional.	Permite a los estudiantes comprender plenamente un procesador a partir de sus fundamentos.	Tiempo de diseño.

## 2.1 PROCESADORES DIDÁCTICOS

Al utilizar el término **procesador didáctico** se hace referencia tanto a la flexibilidad del procesador para emular máquinas sencillas, como a su capacidad para permitir la observación y/o modificación de su estado interno. El objetivo es facilitar la comprensión en cursos de arquitectura de computadoras mediante material didáctico, que pueda apoyar la enseñanza del funcionamiento detallado del hardware para no limitarse a la teoría, permitiendo ejemplificar los nuevos métodos de diseño de sistemas digitales basados en el uso de lenguajes de descripción de hardware (*HDLs*). Teniendo el código fuente HDL sintetizable abre la posibilidad de utilizar microprocesadores sencillos que pueden ser analizados o diseñados por el estudiante para poder comprender mejor su funcionamiento.

El principal propósito de los procesadores didácticos debe ser mostrar cómo está diseñada una computadora y como se pueden vincular sus componentes mediante la transferencia de información. También deben ser capaces de permitir a los estudiantes desarrollar y probar sus propios microprogramas (pueden ser programados en alto nivel, ensamblador o lenguaje máquina) mediante la ejecución y simulación de los mismos para comprobar que el programa realiza la tarea para el cual fue desarrollado.

En la Tabla 2.2 se pueden observar las principales características de los procesadores didácticos que se han desarrollado en universidades a nivel mundial.

Tabla 2. 2 Procesadores didácticos desarrollados en universidades a nivel mundial.

	Arquitectura	Set de instrucciones	Bits de datos	Registro de Propósito General	Numero de instrucciones	Memoria de datos	Memoria de programa
SOPHIA <sup>[2]</sup>	Harvard	RISC	8-bits	8	29, 16/24-bit	256 x 8-bit	256 x 16-bit
MMP16 <sup>[4]</sup>	Von Neumann	ISA	16 bits	16	35, 16/32-bit	-----	16K x 16-bit
ESCOMIPS <sup>[5]</sup>	Harvard	RISC	16 bits	16	16, 25-bit	64k x 16-bit	64k x 25-bit
*BIP I <sup>[6]</sup>	Harvard	RISC	16-bits	25	8, 16-bit	1K x 16-bit	1K x 16-bit
*BIP II <sup>[6]</sup>	Harvard	RISC	16-bits	25	15, 16 bit	1K x 16-bit	1K x 16-bit
MIC-1 <sup>[7]</sup>	Von Neumann	CISC	16-bits	16	32, 16-bit	-----	16-bit
SWEET-16 <sup>[8]</sup>	Harvard	RISC	16-bits	8	16, 8 bit	16-bit	16-bit
MPD <sup>[9]</sup>	Von Neumann	CISC	8-bits	7	18, 8-bit	256 x 8-bits	256 x 8-bits

	Arquitectura	Set de instrucciones	Bits de datos	Registro de Propósito General	Numero de instrucciones	Memoria de datos	Memoria de programa
NEANDER <sup>[10]</sup>	Von Neumann	ASIC	4-bits	-----	8	-----	256 x 4-bit
NEANDER <sup>[11]</sup>	Von Neumann	CISC	8-bits	-----	9	-----	256 x 8-bit
ANEM <sup>[39]</sup>	Harvard	RISC	16-bits	16	21	64k x 16-bit	64k x 16-bit
ILA9200 <sup>[13]</sup>	Harvard	ASIC	16-bits	20		16-bit	16-bit
CABARE <sup>[14]</sup>	Von Neumann	RISP	32-bit	11	20	-----	8k x 32-bit
TINYCPU <sup>[15]</sup>	Harvard	RISC	16-bits	-----	29	16-bit	256k x 16-bit

Desde que se inició el estudio de procesadores didácticos, ha habido el desarrollo de varios procesadores para dicho fin, como el Microprocesador didáctico microprogramado de 16-bit (**MMP16**), el cual se desarrolló como una herramienta de estudio mediante el uso de los componentes electrónico-digitales y la implementación total de un microprocesador digital.

**MMP16** está enfocado principalmente a la educación, posee una unidad de control, y una unidad de procesamiento de datos, este procesador usa dos diferentes señales de reloj (una para ser usado por la *unidad de control* y definir los estados de la máquina; y el segundo para cargar los registros de la *unidad de procesamiento* de datos y sincronizar las señales de memoria y buses), una arquitectura Von Neumann de 16-bit, Memoria de 16-bit, 16 registros de propósito general, conjunto de instrucciones reducidas y 35 instrucciones (16 / 32 bit) [25].

Por otra parte, los sistemas integrados han venido incrementando su potencial, tanto de procesamiento como de integración en un mismo chip, para poder dar paso a sistemas más complejos. De esta manera es posible utilizar un procesador concreto o hacer uso de la lógica programable, dando paso a los **System on a Programmable Chip (SoPC)** que permiten implementar en un único dispositivo sistemas complejos que antes empleaban varios circuitos integrados en un mismo circuito integrado. Al emplear un **FPGA** se tiene la ventaja de poder emplearse para el prototipado de un **ASIC** o bien ser un sistema final. Además se cuenta con la capacidad de expandir el sistema en caso de ser necesario.

Muchos procesadores han adquirido esta arquitectura por ser una de las más eficientes. Los **Microprocessor without Interlocked Pipeline Stages (MIPS)** son utilizados en aplicaciones como computadoras, routers, consolas de videojuego y algunos sistemas embebidos. Por esta razón, son las arquitecturas didácticas más utilizadas en ambientes universitarios. Estos procesadores han permitido enseñar los elementos básicos de una arquitectura de computadoras para un posterior aprendizaje de arquitecturas más complejas [30].

Académicamente existe una implementación llamada **MIPS-1 ISA**, está diseñada bajo una arquitectura simple que hace al estudiante buscar soluciones entre el desempeño y la disipación de energía, además de poder ser usado en actividades de varios cursos académicos: como *circuitos digitales*, *arquitectura de computadoras*, *sistemas operativos*, *compiladores* y *microprocesadores*.

Las principales características del microprocesador **MIPS-1** son: 32 registros de propósito general de 32-bit, arquitectura *RISC*, conjunto de 20 instrucciones, todas en formato de 32-bit, con 8 bits de operación de código, 4 bits para el registro de código, 4 bits para el registro destino y 16 bits para valores constantes. Son 20 las diferentes operaciones que puede realizar en el CPU y 11 códigos de operación reservados para instrucciones futuras [31].

Otra herramienta didáctica es la llamada **Cabare**, la cual utiliza una unidad de control basada en reconfiguración, puede ser usado en aplicaciones multimedia. **Cabare** es un procesador de propósito general diseñado con pocas instrucciones, está basado en el procesador didáctico **Altera**. Posee 11 registros de propósito general, un registro para el contador de programa (*PC*), una *ALU* con acumulador de registros y un registro *G* donde se mantienen los resultados de las operaciones realizadas por la *ALU*. Cuenta con 8 microinstrucciones de salto, y 8 operaciones que puede realizar la *ALU*. La memoria RAM tiene una longitud de datos de 32 bits [32].

Por su parte **MAC-1** es un procesador basado en una arquitectura microprogramada y usada para propósitos educativos. Está implementada en VHDL, se divide en 4 partes: *Control* (responsable de las operaciones), *Operativo* (operaciones de ellos), *RAM* (memoria que contiene el programa) y *Reloj*. Fue probada en una tarjeta NEXYS 2, con **Xilinx Spartan 3E FPGA XC3S500E** [33]–[35]. Su arquitectura microprogramada implementa el control del microprocesador a través de la memoria de programa [36].

El procesador **Sweet-16**, posee una arquitectura RISC de 16-bit su principal propósito es enseñar a los estudiantes con o sin conocimientos previos de electrónica digital o un HDL, es por esta razón que fue desarrollado para ilustrar y usarse en cursos de arquitectura de computadoras. La arquitectura del procesador Sweet-16, permite el uso del procesador para realizar ejercicios de programación, fue programado en lenguaje ensamblador, permitiendo que esté disponible en varios sistemas operativos [24].

En el *Centro Universitário Positivo de Curitiba, Brasil* en 2001, se llevaron a cabo prácticas donde se involucró un prototipo de un microprocesador de 4 bits con arquitectura **CISC** desarrollado con un HDL y probado mediante simuladores. En 2002 se agregaron pruebas en hardware, el FPGA fue configurado para realizar dichas pruebas. En 2003, se reemplazó la arquitectura CISC por RISC que ya incluyen pruebas de hardware, posteriormente en 2004 se incorporó un compilador para el procesador RISC [37].

El **mPd**, es un microprocesador de 8 bits, diseñado en VHDL, con arquitectura CISC; tiene características de conexionado con circuitos de memoria y sus dispositivos de entrada/salida son: *Bus de datos* de 8 bits, *Bus de direcciones* de 8 bits, *Mapa de memoria* y *mapa de entrada/salida independientes* de 256 direcciones cada uno, Línea M1: activa a nivel alto en los ciclos de búsqueda del código de operación, Dos líneas de petición de interrupción: INT0 e INT1, activas a nivel alto y Una línea de petición de estados de espera, activa a nivel alto.

En cuanto al modelo de programación del mPd, cuenta con siete registros, todos ellos de 8 bits. El conjunto de instrucciones está formado por 19 grupos de instrucciones. La programación del mPd se realiza en lenguaje ensamblador mediante una hoja de Excel que permite elegir, mediante el nemotécnico, cada una de las instrucciones que forman el programa, así como el operando de la instrucción, en caso de que la instrucción elegida lo requiera [38].

El procesador **ANEM**, se desarrolló para ser usado como un microcontrolador. Una de sus principales características es el sistema de acceso a memoria basada en la arquitectura Harvard. Esta arquitectura consiste en usar la separación física de la memoria de datos y la memoria de programa, permitiendo accesos simultáneos entre ambas, dando como resultado un mejor funcionamiento, comparado con la arquitectura Von Neumann.

Para simplificar el diseño de la memoria del programa en *ANEM*, fue implementada usando bloques de memoria RAM disponibles en el propio *FPGA*. *ANEM* tiene canales alternos permitiendo a la memoria del programa sea cargada a través del puerto serial. Para que esto suceda, el programador debe estar habilitado (Prog pin debe mantenerse en nivel lógico alto), y el software debe enviar el código listo en formato binario siguiendo un algoritmo simple. El software fue desarrollado usando lenguaje de procesamiento para ejecutar las tareas del programa [39].

El procesador **ILA9200** está dividido en varias unidades funcionales que han sido implementadas en un *ASIC*: *Unidad de control* que contiene un secuenciador de microinstrucciones, una *memoria de microprograma* de 256 palabras de 48 bits y un *registro de instrucción* de 16 bits; *unidad operativa* que contiene 14 registros de 16 bits distribuidos en dos bancos, 2 *ALUs* cada una con sus registros de banderas y *registros dedicados* para el acceso a la memoria *RAM* de *programa* y *datos*; *unidad de comunicación* con el *Contador de Programa (PC)*, contiene 6 *registros* mapeados en el espacio de memoria del PC mediante los cuales el usuario especifica las órdenes de monitorización del estado de los recursos en las otras 2 unidades [40].

El procesador *ILA9200* utiliza un microensamblador cuya función es traducir un conjunto de mnemónicos predefinidos en microinstrucciones de un microprograma.

Otra herramienta es **Neander** es una arquitectura didáctica y virtual conocida por su software emulador, el cual es usado para la enseñanza de arquitectura de computadoras y lenguaje ensamblador, es un procesador de 4-bit, cuya principal meta es evaluar el potencial de esta tecnología para propósitos educacionales y de investigación, la arquitectura fue adaptada a 4-bit, el conjunto de instrucciones está compuesto por 8 instrucciones [41].

La versión original de *Neander* es un simple procesador didáctico de 8-bit, todo el proyecto fue diseñado con la herramienta *CAD*. Ha sido usado por años en cursos de introducción a la arquitectura de computadoras en universidades brasileñas. Contiene 9 instrucciones, las cuales incluyen lógica y aritmética, bifurcación condicionante e incondicionante y la transferencia de datos en memoria.

Por su parte el procesador **SIC** está conformado de 5 registros internos (*A*, *X*, *L*, *PC* y *SW*) todos estos de 24-bit y con funciones específicas. Contiene 26 instrucciones de operaciones básicas con las características del procesador RISC, las instrucciones están divididas en: operaciones aritméticas, lógicas, comparación, desvío, carga, descarga y de entrada/salida [42].

*Neander* está compuesto por entidades simples que controlan el camino de datos y el flujo del programa, el direccionamiento de memoria es directo, entonces los datos siempre accederán a través de su posición en memoria. ALU considera dos entradas de 8-bit y 5 distintas operaciones. La interfaz de memoria está compuesta por tres buses (dirección, datos de entrada y salida y leer y escribir señales).

Tiene incluido tres modos de prueba, cada uno con un objetivo diferente, el primero es ayudar en las pruebas de registro simple para validar los flip-flop; el segundo modo prueba las funciones de ALU; y el tercero modo de prueba bloquea la prueba del temporizador, el cual controla toda la sincronización del circuito [43].

En la *Universidad Federal de Rio grande del Norte, en Natal, Brasil (UFRN)* se desarrolló un proyecto donde se utilizó el lenguaje VHDL para la descripción de la arquitectura y el procesador didáctico **MIC-1**, cuya parte operativa está constituida por una memoria de 16 registros de 16 bits, una ALU, dos registros con una memoria (*MAR* y *MBR*), shifter lógico (derecha/izquierda) y tres buses, dos de lectura y uno de escritura en memoria, para controlar las vías de datos son necesarias microinstrucciones de 32 bits. *MIC-1* es un microprocesador con un reloj basado en cuatro fases, que garantizan la correcta ejecución de sus instrucciones en un ciclo básico de la parte operativa.

El *MIC-1* no tiene instrucciones de entrada y salida, para esto utiliza, cuatro direcciones de memoria para acomodar un dispositivo de entrada (*0x0FFC* y *0x0FFD*) y un dispositivo de salida (*0x0FFE* y *0x0FFF*), las direcciones *0x0FFD* y *0x0FFF* son utilizados como registradores del estatus del dispositivo, mientras que las direcciones *0x0FFC* y *0x0FFE* almacenan un byte de entrada o salida, respectivamente [44].

En la *Universidad de Hiroshima* se utilizan las siguientes herramientas mediante las cuales los estudiantes pueden aprender los conceptos básicos de informática con la ayuda de un procesador de 16-bit llamado **TINYCPU**, un lenguaje ensamblador **TINYASM** y un compilador **TINYC**, a través de los cuales los estudiantes pueden entender fácilmente la arquitectura del procesador, programación en ensamblador y el diseño del compilador, respectivamente.

*TINYCPU*, *TINYASM* y *TINYC* fueron desarrollados utilizando Verilog HDL, Perl y Flex/Bison, respectivamente, para implementar y ejecutar *TINYCPU* se utilizó un *FPGA* en *Spartan-3E* y *Spartan-3A*.

El compilador *TINYC* traduce un programa basado en C en un lenguaje ensamblador para *TINYASM*. El lenguaje de programación *TINYC* soporta enteros de 16 bits con signo, instrucciones: *if*, *if-else*, *while*, *do while*, y la declaración *goto*. Además cuenta con operaciones aritméticas y lógicas básicas como suma, resta, multiplicación, negación, operaciones lógicas y de comparaciones.

Las herramientas *TINYCPU*, *TINYASM* y *TINYC* son utilizados para los cursos de hardware embebido y software embebido para estudiantes de posgrado [45].

Por otro lado en la *Universidad del Valle de Itajai en Brasil (UNIVALI)* se ha desarrollado los procesadores ***BIP I*** y ***BIP II***, el cual utiliza una arquitectura basada en la arquitectura RISC del microcontrolador PIC de Microchip, utiliza el modelo Harvard, presenta un único registro para almacenamiento de datos, todas las operaciones se desarrollan en el acumulador, y todas las instrucciones están basadas en un único formato de instrucción y en dos modos de direccionamiento [46].

El procesador *BIP I* ésta basado en una arquitectura simple, con un conjunto de instrucciones (*ISA*) que incluye: carga, almacenamiento e instrucciones aritméticas; y al procesador *BIP II* se le agregan instrucciones condicionales e incondicionales. Ambos microprocesadores son empleados en algunos cursos en el área de ciencias de computación de *UNIVALI*, permitiendo que los estudiantes puedan mejorar su comprensión de los conceptos de arquitectura de computadoras.

Los atributos de las arquitectura de los procesadores *BIP I* y *BIP II*, incluyen: Arquitectura orientada al acumulador; ancho de datos de 16-bit; ancho de instrucciones de 16-bit; un solo tipo de dato (integer); un solo tipo de formato de instrucción; dos modos de direccionamiento (directo e indirecto); conjunto de instrucciones reducida (*RIS*); y, mapeo de memoria I/O. El procesador presenta muchas características de *RISC*, pero no puede ser considerado como tal, pues no cuenta con la arquitectura de carga-almacenamiento. El CPU tiene tres registros (*PC*, *ACC* y *STATUS*) [27].

*BIP I* cuenta con 8 instrucciones, mientras que el *BIP II* con 15. Utiliza arquitectura monocyple/Harvard, con dos memorias separadas.

Con el paso del tiempo el estudio y diseño de procesadores ha cambiado, así como las aplicaciones para el fin que han sido perfeccionados, han aparecido los procesadores de red los cuales fueron desarrollados usando modelos de arquitectura como **ASIP** (Application Specific Processor) y SoC (System-on-Chip), añadiendo la arquitectura *RISC* para un mejor desempeño computacional. Estos procesadores tienen un modelo de *ISA* específico para operaciones de red.

En la *Universidad Pontificia de Minas Gerais*, se desarrolló un simulador de un procesador didáctico de red, el cual fue construido con C++ y la principal interfaz es capaz de ayudar y guiar al estudiante en el aprendizaje del procesador de red y su funcionamiento [47].

De igual manera se ha desarrollado un procesador de red llamado *ASIP* que contiene instrucciones especiales para el acceso a nivel-bit a los registro de datos que se requieren al procesamiento de protocolo de comunicación orientado a paquetes. La arquitectura del procesador es *RISC* de 16 bits, con 12 registros de propósito general y extensiones especiales para el acceso a los datos a nivel-bit. Se utilizó el compilador **LANCE** para la optimización de *ANSI C* [48].

## 2.2 COMPILADORES Y HERRAMIENTAS DE DESARROLLO

Uno de los aspectos que han evolucionado de manera más visible son los *lenguajes de programación*. Los lenguajes de programación de *alto nivel* permiten al programador expresar algoritmos de manera concisa, pero ocasionan otro problema conocido como el llamado salto semántico, que es la diferencia entre las operaciones que proporcionan los lenguajes de alto nivel y las que proporciona la arquitectura de computadoras.

Para desarrollar programas de manera eficiente es necesario entender como son ejecutados estos en la computadora, y en ocasiones esto no se cumple, por ejemplo, es difícil entender la estructura de datos “puntero” si no se sabe cómo se implementa en lenguaje máquina.

Desafortunadamente muchos estudiantes inician su aprendizaje de un lenguaje de programación de alto nivel pero sólo a nivel de sintaxis y no se entiende la esencia de cómo se haría en inicio con un pseudolenguaje o en una lenguaje de bajo nivel, llevando a enfrentar un gran problema, la migración de la teoría a la práctica.

Para que un hardware sea funcional requiere de un software que ayude al usuario a interactuar con él. Se han propuesto una gran cantidad de herramientas: Se ha propuesto una herramienta que puede aceptar la sintaxis del lenguaje C y generar muchas versiones de los programas en ensamblador. Estos programas en ensamblador realizan las mismas funcionalidades definidas por el programa de entrada de C pero con diferentes comportamientos de *EMI* (electromagnet interference) conducidas cuando son ejecutados [49].

Es sabido que las herramientas de software de alto nivel (como C, C++, Python, JAVA, etc.) pueden ayudar a generar diferentes programas, como se pueden realizar en lenguaje ensamblador con la misma funcionalidad, pero con mayor número de instrucciones.

La arquitectura de un compilador puede ser dividida en cuatro partes: **Análisis léxico** (es el proceso en el cual se convierten la secuencia de caracteres en secuencia de tokens), **Gramática** (comprueba la gramática y construye una estructura de datos), **AST** (Abstract Syntax Tree, representa la relación de tokens) y **Generador de código** (convierte la optimización de los AST en salida como código ensamblador) [49].

Por otra parte **la optimización de código** representa varias consideraciones, como: *tiempo de ejecución efectivo* (optimización para instrucciones rápidas), *pocas instrucciones* (menor almacenamiento y menos memoria), *exportación de registros disponibles* (eficiente almacenamiento de registros) y *portabilidad*.

Otro diseño de un compilador fue desarrollado basándose en la herramienta **SUIF**. Este es un compilador de código abierto para los lenguajes *C* y *Fortran 77*. Este compilador se centra en identificar los patrones de comunicación en el tiempo de compilación, en otras palabras, se centra en la detección y análisis de los componentes de la comunicación del framework [33].

A lo largo de la historia se han diseñado herramientas que faciliten el desarrollo de un compilador, ya sea por completo o en cualquiera de sus fases como: **JastAdd** [35], es un sistema basado en java para construcción de compiladores, se centra en las representaciones del *árbol de sintaxis abstracta*. Su principal idea es definir cada aspecto del lenguaje en clases separadas para dar la posibilidad de usar todas las características del lenguaje de programación *JAVA*.

**AspectG** [50] fue creado para conseguir la interpretación modular con el lenguaje de definición **ANTLR**; y **AspectASF** [51] tiene como objetivo especificar declarativamente qué reglas debe adaptarse para incorporar una semántica adicional.

Otra herramienta de apoyo para el desarrollo de compiladores es **LISA** el cual genera automáticamente un compilador y otras herramientas lingüísticas relacionadas con las especificaciones de lenguaje basados en atributos gramaticales formales. *LISA* consiste en definiciones regulares, definición de atributos, reglas (son las reglas de sintaxis generalizadas que encapsulan reglas semánticas), y métodos sobre dominios semánticos [52].

Por otra parte, ninguna de las herramientas disponibles en la actualidad apoya el desarrollo de los lenguajes incrementales. Por lo tanto, el diseñador del lenguaje tiene que diseñar nuevos lenguajes a partir de cero o por medio de la modificación de versiones anteriores. Así es como se desarrolló la herramienta *LISA* con la que los estudiantes tienen la posibilidad de experimentar, estimar y probar varios analizadores léxicos y sintácticos, *LISA* es un *IDE* en el cual los usuarios pueden especificar, generar, compilar y ejecutar programas de nuevos lenguajes especificados. El compilador/interprete generado por *LISA* es visualizado de manera similar como se muestra en [53].

*LISA* es una herramienta que facilita el aprendizaje y una comprensión conceptual de la construcción de compiladores. Se generó un analizador léxico o un escáner en Java para los autómatas finitos determinísticos (*AFD*); tokens del código del programa son agrupadas en frases gramáticas; cuando las sentencias son correctas, pueden ser computadas [54]-[55].

La mayor técnica de abstracción en ingeniería de software es dividir el sistema en componentes funcionales de tal manera que se pueda cambiar un componente en particular y no todo el sistema. Las propiedades correspondientes a las definiciones de lenguaje basados en gramáticas atributo son: definiciones regulares léxicas; definición de atributos; reglas generalizadas de sintaxis que encapsulan reglas semánticas; y, operación en dominios sistemáticos [56].

Combinar técnicas orientadas a objetos y conceptos necesarios con técnicas orientadas a aspectos permite lograr una mejor modularidad, extensibilidad y reusabilidad. El *lenguaje* está dividido en dos partes principales: el *tipo de definiciones* y las *definiciones de variable*.

Otra herramienta que ayuda a la generación de compiladores es llamada **Coco/R** la cual toma atributos gramáticos y produce un escaneo analizador descendente, además analiza el contexto gramático (es decir, analiza la sintaxis), verifica atributos que pueden ser considerados como parámetros y acciones semánticas. Cada atributo gramático es procesado de izquierda a derecha. Mientras la sintaxis es analizada las acciones semánticas son ejecutadas [57].

Existen diferentes tipos de compiladores, los más comunes son los que pasan de un código de *alto nivel* (más entendible por el usuario, como código en lenguaje C o JAVA) a uno de *bajo nivel* (código máquina), pero también hay compiladores que pasan de *lenguaje ensamblador* a *código máquina*.

Como es sabido, diferentes combinaciones de código ensamblador pueden realizar la misma tarea, cada versión diferente de código es llamado Template; diferentes códigos pueden causar un comportamiento diferente de *EMI*, en otras palabras, es posible generar las mínimas de combinaciones de código ensamblador. Para realizar esto, primero se miden la ondas de cada una de las instrucciones; segundo, obtener la frecuencia de respuesta basándose en el estudio de la deducción del teorema **DFT**; y tercero elegir la frecuencia mínima de las combinaciones [58]-[59].

Las arquitecturas reales tienen un sinnúmero de referencias en el desarrollo y depuración de aplicaciones, aun así, el detalle de cada arquitectura es muy complejo; en cambio los simuladores proveen un mayor detalle de abstracción de la arquitectura.

Se ha desarrollado una plataforma que está formada por *hardware*, *firmware* y *herramientas de software*. El principal objetivo de este trabajo es el desarrollo de una plataforma donde se combine el software y hardware. Las herramientas de software para el desarrollo de esta plataforma son: *IDE*, *compilador* repositorio y *software* de programador [60].

### **2.3 ENTORNO DE DESARROLLO INTEGRADO**

El estudio de arquitectura de computadoras es fundamental en la formación de alumnos del área de computación, pues si se tiene una correcta asimilación de conceptos de arquitectura de computadoras, ayudan a fijar de mejor manera los conceptos de programación como por ejemplo: variables, declaración de variables, asignación de variables y su correspondencia con las operaciones de acceso a memoria (como asignación), además de vectores, punteros, niveles de lenguaje, operaciones aritméticas entre otros más.

La elección de un procesador para enseñar correctamente la lógica de programación y conceptos de arquitectura de computadoras, debe facilitar el establecimiento de relaciones entre las abstracciones lógicas necesarias para programación y la implementación de estas abstracciones en hardware. La elección del procesador también debe de priorizar aspectos didácticos que favorezcan la comprensión de las relaciones entre software y hardware en un enfoque multidisciplinario.

El objetivo de desarrollar una arquitectura de referencia para la enseñanza de conceptos en el área de arquitectura y organización de computadoras es proveer una base necesaria para la comprensión de abstracciones presentadas en los cursos de algoritmos, programación, sistemas digitales, sistemas operativos.

Existen diferentes disciplinas donde se abarca a detalle el estudio de arquitecturas y sistemas de software relacionadas con esto, existen dos ideas que pueden ser utilizadas: el uso de arquitectura de procesadores reales o de simulaciones.

Como se ha mencionado, los cursos de arquitectura de computadoras juegan un papel importante en el desarrollo de habilidades y destreza en ciencias de computación y en cursos de ingeniería, especialmente para los que trabajan con el diseño de sistemas embebidos.

Los procesadores se deben basar en arquitecturas alternativas más simples con el fin de facilitar la programación y la construcción de un procesador: arquitectura basada en acumulador; todas las instrucciones deben estar basadas en un forma de instrucciones simples; dos modos de direccionamiento (inmediata y directa).

Al uso de procesadores con fines didácticos, se le debe de dar un enfoque *multidisciplinario*, es decir, deberían de ser utilizados en varias asignaturas antes mencionadas, para así, poder ayudar a los alumnos a asimilar mejor los conceptos de cada una de las asignaturas, relacionando conceptos entre sí.

El conocimiento de la operación de las computadoras es muy importante, ya que permiten a los estudiantes entender conceptos del área de programación, de esta manera, los profesores de las materias de introducción a la programación utilizan abstracciones de conceptos de arquitectura de computadoras para representar algunos de sus conceptos para permitir a los alumnos un mejor entendimiento de esta materia.

En la *Universidad de Pisa* desarrollaron un ambiente llamado **Csim2**, el cual permite al estudiante familiarice con conceptos de localidad de programa, estructura del cache y ajuste de rendimiento, mientras la producción actual del dato se analiza por el software que se encuentra con el sistema embebido, de esta manera los estudiantes pueden analizar el comportamiento del programa [61].

El ambiente educacional del *Csim2* ayuda a combinar dos exigencias complementarias de los cursos de arquitectura de computadoras de los sistemas embebidos: *Disponibilidad de una herramienta* (la cual provee ejemplos prácticos para mostrar los principales conceptos de la arquitectura de sistemas embebidos) y por otro lado, *fomentar* en los estudiantes a la *actividad de diseño real de sistemas* orientados a aplicaciones integradas.

En *UNIVALI* [46] se ha desarrollado e implementado un *IDE* (*BIPIDE*) con la finalidad de crear un ambiente amigable para el usuario al momento de estudiar el funcionamiento de los procesadores arquitecturas *RISC* de 16-bit creado con propósitos educativos, en la que se integran dos herramientas: un *compilador* en lenguaje *Portugol* y un *simulador* de un procesador (ya sea el *BIP I* o *BIP II*).

***BIPIDE***, ofrece un enfoque interdisciplinario que reduce el grado de abstracción de conceptos como: uso de memoria, operaciones aritméticas, desvíos y lazos. Presenta tres módulos principales:

- *Programación*: permite que el usuario escriba y compile sus propios programas;
- *Simulación*: permite simular los programas desarrollados; y
- *Ayuda*: presenta información respecto a la funcionalidad del sistema y sobre la arquitectura y organización de los procesadores *BIP* [62].

## **CAPÍTULO 3. MARCO TEÓRICO**

En este capítulo se desarrollarán conceptos teóricos necesarios para poder comprender la implementación del IDE, mostrando la arquitectura, características, modo de direccionamiento y conjunto de instrucciones del procesador SOPHIA. Posteriormente describiremos los tipos de compiladores y sus diferentes fases. Por último se describirán los simuladores y sus características.

### **3.1 PROCESADOR DIDÁCTICO SOPHIA**

El procesador didáctico SOPHIA nace por la necesidad de tener una herramienta que ayude en el proceso de aprendizaje en los cursos de arquitectura de computadoras, está implementado en un *FPGA* para ser una librería.

#### **3.1.1 CARÁCTERÍSTICAS Y ARQUITECTURA**

- Basado bajo la arquitectura Harvard.
- Conjunto de instrucciones RISC
- Instrucción simple – Dato simple (SISD)
- Memoria ROM de 256 localidades de 16-bits.
- Memoria RAM de 256 localidades de 8-bits.

La arquitectura del procesador que se muestra en la Figura 3.1, se puede observar la unidad de control, la memoria de instrucciones, de datos, el decodificador de instrucciones, unidad de control, registros de propósito general, la unidad aritmético-lógica, las banderas y el apuntador de pila.

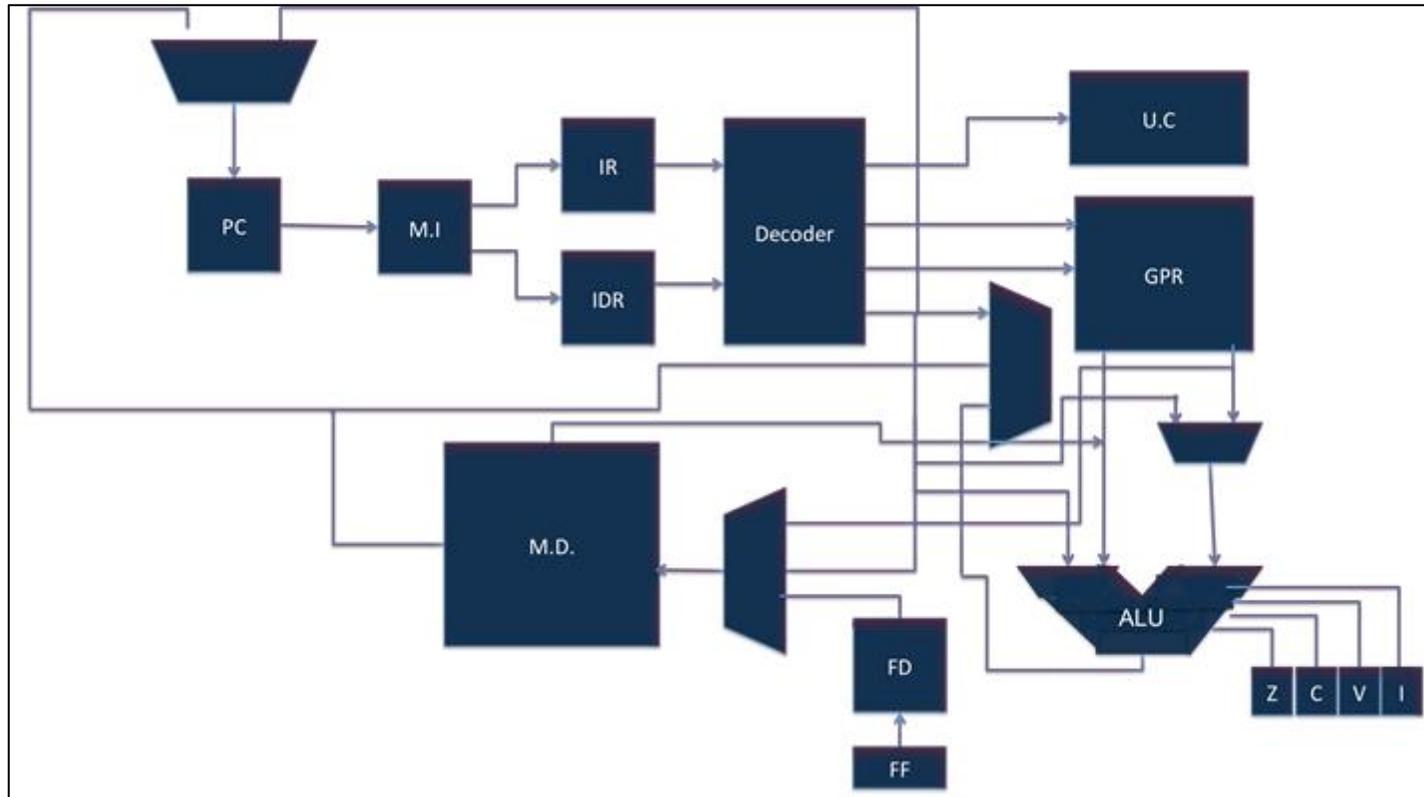


Figura 3. 1 Arquitectura del procesador SOPHIA.

**PC – Contador de programa. M.I – Memoria de Instrucciones. ALU – Unidad Aritmético-Lógica  
 Decoder – Decodificador. U.C – Unidad de Control. GPR – Registros de Proposito General. M.D. – Memoria de Datos.  
 Z – Bandera Cero. C – Bandera Acarreo. V – Bandera Sobreflujo. I – Bandera Interrupción. FF – Última localidad**

### 3.1.2 CONJUNTO DE INSTRUCCIONES

En la Tabla 3.1 se muestra el conjunto de instrucciones que contiene 29 instrucciones de una longitud de 16 bits, las cuales se dividen en cinco grupos (saltos, un registro con datos inmediatos, dos registros, un registro y directas, es decir sólo el nemónico).

### 3.1.3 MODOS DE DIRECCIONAMIENTO

Se tienen diferentes tipos de direccionamiento, dependiendo de la naturaleza de las instrucciones, la distribución de los 16 bits que componen cada instrucción se puede observar en la Tabla 3.1. Los modos de direccionamiento se muestran a continuación:

- **Directo a memoria.**
  - Cuando una dirección en la memoria de programa es direccionada por el contador de programa propicia que el valor de 8 bits se cargue. Este direccionamiento corresponde a la categoría de saltos. Figura 3.2.

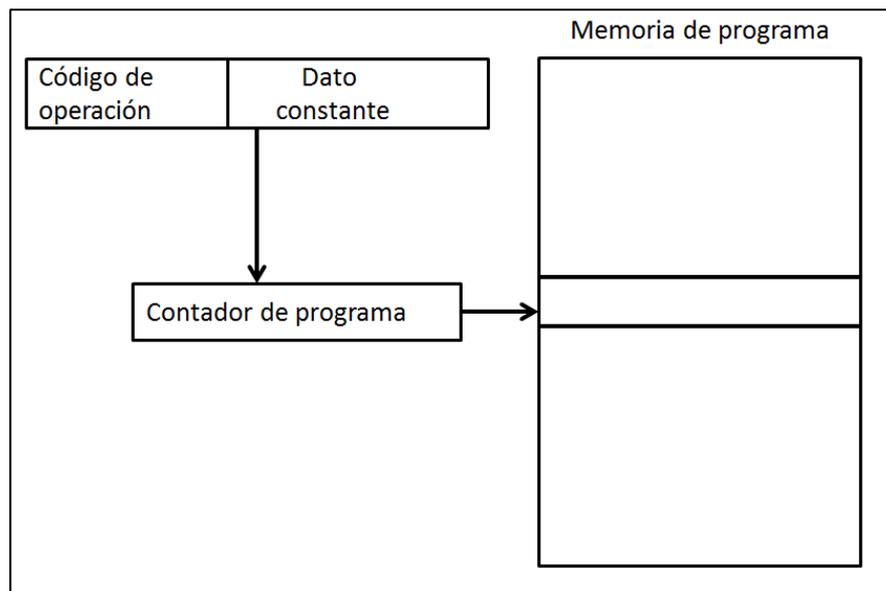


Figura 3. 2 Direccionamiento directo a memoria, donde el código de operación define el tipo de salto y dato constante el número de la próxima instrucción a ejecutar.



					Formato de instrucción de 16 bits																
					OPCODE					PARÁMETROS											
No.	Nemónico	Descripción	Sintaxis	Micro-operación	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	Ciclos
<b>Tipo dos registros</b>																					
15	ADD	Suma	ADD Rd, Rs	$Rd \leftarrow Rd + Rs$	1	0	0	0	0	Rd	Rd	Rd	Rs	Rs	Rs	-	-	-	-	-	
16	SUB	Resta	SUB Rd, Rs	$Rd \leftarrow Rd - Rs$	1	0	0	0	1	Rd	Rd	Rd	Rs	Rs	Rs	-	-	-	-	-	
17	AND	Y lógico	AND Rd, Rs	$Rd \leftarrow Rd \text{ AND } Rs$	1	0	0	1	0	Rd	Rd	Rd	Rs	Rs	Rs	-	-	-	-	-	
18	OR	O lógico	OR Rd, Rs	$Rd \leftarrow Rd \text{ OR } Rs$	1	0	0	1	1	Rd	Rd	Rd	Rs	Rs	Rs	-	-	-	-	-	
19	LDX	Carga indirecta de memoria	LDX Rd, [Rs]	$Rd \leftarrow [Rs]$	1	0	1	0	0	Rd	Rd	Rd	Rs	Rs	Rs	-	-	-	-	-	
20	STX	Almacenamiento indirecto en memoria	STX [Rd], Rs	$[Rd] \leftarrow Rs$	1	0	1	0	1	Rd	Rd	Rd	Rs	Rs	Rs	-	-	-	-	-	
<b>Tipo un registro</b>																					
21	NOT	Suma inmediata	ADDI Rd, k	$Rd \leftarrow Rd + k$	1	1	1	0	0	Rd	Rd	Rd	-	-	-	-	-	-	-	-	
22	SHL	Resta inmediata	SUBI Rd, k	$Rd \leftarrow Rd - k$	1	1	1	0	1	Rd	Rd	Rd	-	-	-	-	-	-	-	-	
23	SHR	Y inmediato	ANDI Rd, k	$Rd \leftarrow Rd \text{ AND } k$	1	1	1	1	0	Rd	Rd	Rd	-	-	-	-	-	-	-	-	
24	SWAP	O inmediato	ORI Rd, k	$Rd \leftarrow Rd \text{ OR } k$	1	1	1	1	1	Rd	Rd	Rd	-	-	-	-	-	-	-	-	
25	PUSH	Carga inmediata	LDI Rd, k	$Rd \leftarrow k$	1	1	0	0	0	Rd	Rd	Rd	-	-	-	-	-	-	-	-	
26	POP	Carga directa de memoria	LDD Rd, [A]	$Rd \leftarrow [A]$	1	1	0	0	1	Rd	Rd	Rd	-	-	-	-	-	-	-	-	
<b>Tipo directo</b>																					
27	RET	Almacenamiento directo a memoria	STD [A], Rd	$[A] \leftarrow Rs$	0	0	1	0	1	-	-	-	-	-	-	-	-	-	-	-	
28	SETI	Lectura de puerto	IN Rd, P	$Rd \leftarrow P$	0	0	1	1	0	-	-	-	-	-	-	-	-	-	-	-	
29	RSTI	Escritura en puerto	OUT P, Rs	$P \leftarrow Rs$	0	0	1	1	1	-	-	-	-	-	-	-	-	-	-	-	

- **Inmediatas.**

- Este tipo de instrucciones utilizan un valor constante, afectando el contenido de un registro del GPR. Figura 3.3.

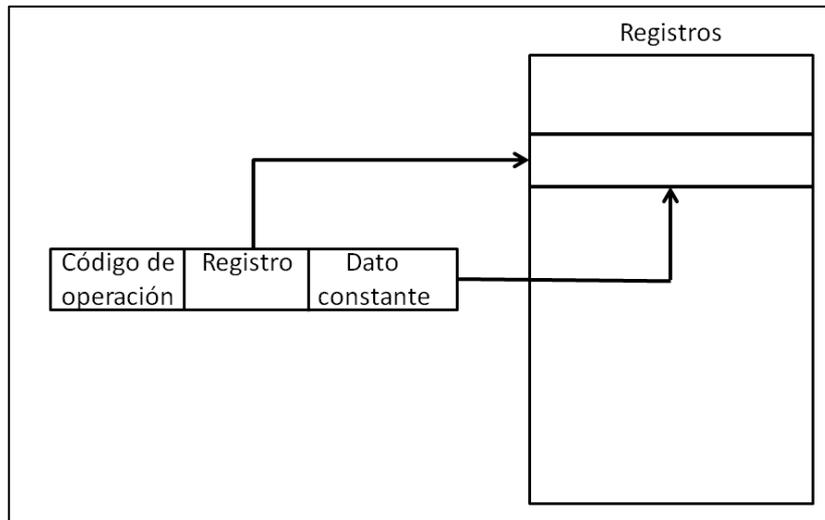


Figura 3. 3 Direccionamiento instrucciones inmediatas, el dato constante actualiza el valor del registro.

- **Dato directo desde memoria de datos.**

- Las instrucciones necesitan un valor constante como dirección para acceder a la memoria de datos, ya sea para realizar operaciones de lectura o escritura. Figura 3.4.

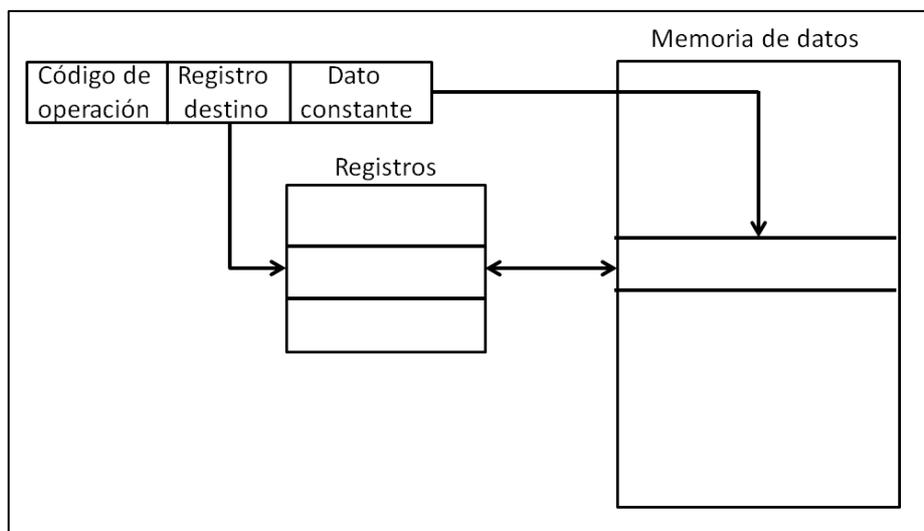


Figura 3. 4 Direccionamiento dato directo desde memoria de datos, el dato constante determina la localidad de memoria la memoria de datos a la que se accederá para obtener su valor, mismo que ha de ser cargado en el registro destino.

- **Entrada/Salida directa.**

- Utilizados con el propósito de comunicar el procesador mediante diferentes periféricos. Figura 3.5.

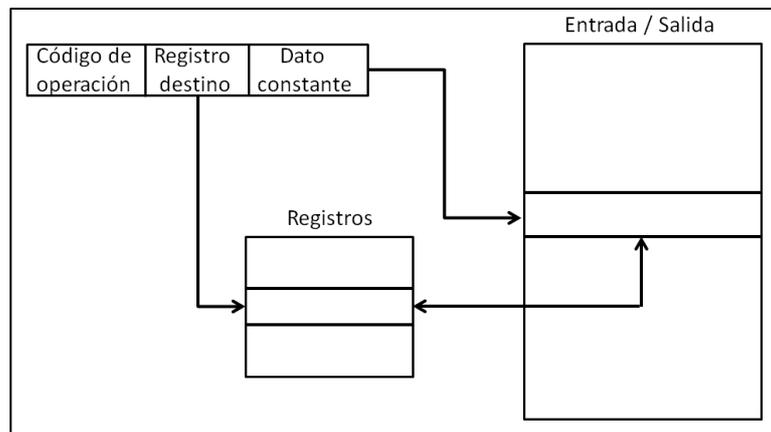


Figura 3. 5 Direccionamiento de Entrada/Salida, el dato constante determina la localidad de memoria la memoria de datos a la que se almacenará el valor del registro.

- **Registro directo, dos registros.**

- Este tipo de direccionamiento utiliza dos registros del GPR para modificar el contenido del registro destino mediante operaciones aritméticas o lógicas, utilizando el valor del registro fuente como se muestra en la Figura 3.6.

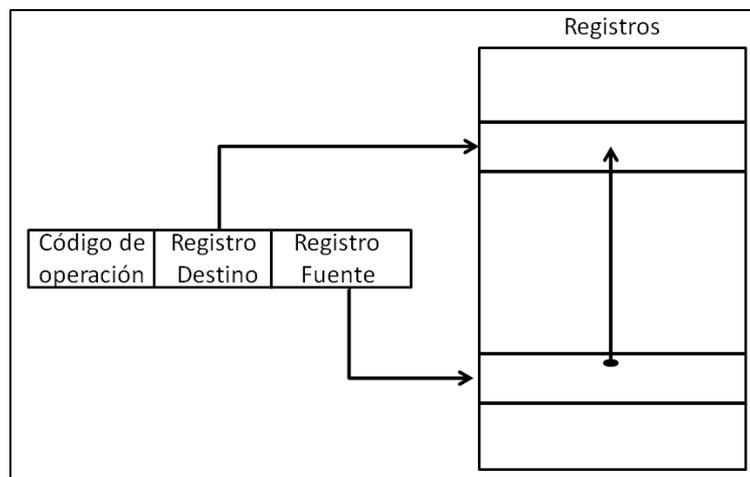
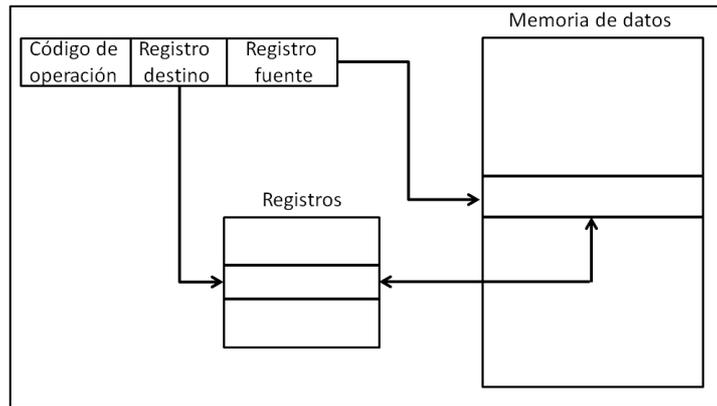


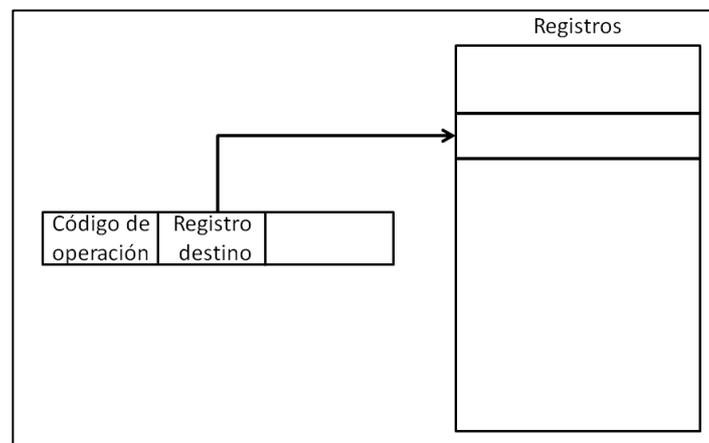
Figura 3. 6 Direccionamiento directo con dos registros, el valor del registro fuente es almacenado en el registro destino.

- **Acceso a memoria de datos indirecto (utilizando un registro).**
  - El valor contenido de un registro fuente es utilizado como apuntador a la memoria de datos, para leer o escribir el valor contenido en un segundo registro. Figura 3.7.



**Figura 3. 7 Direccionamiento acceso indirecto a memoria mediante un registro, registro fuente es un apuntador a memoria de datos, el código de operación define si se lee o escribe el valor de la memoria de datos en el registro destino.**

- **Registro directo (un solo registro).**
  - Utiliza solo un registro del GPR para modificar el contenido con el resultado de una operación de la ALU (aritmética o lógica). Figura 3.8.



**Figura 3. 8 Direccionamiento acceso directo un solo registro, el resultado de la ALU es almacenado en el registro destino.**

## 3.2 COMPILADORES

Existe una gran variedad de tipos de desarrollo de compiladores, de acuerdo al propósito para el cual se desarrollará.

- ***Una pasada.***
  - Examina el código fuente una vez, al mismo tiempo genera el código objeto.
- ***Pasadas múltiples.***
  - Requiere de pasos intermedios (pasadas) para producir un código en otro lenguaje y una última pasada para generar el código objeto.
- ***Optimización.***
  - Lee el código fuente, lo analiza y descubre potenciales errores sin ejecutar el programa.
- ***Compiladores incrementales.***
  - Generan un código objeto instrucción por instrucción cuando el usuario teclea cada orden individual.
- ***Compilador cruzado.***
  - Se genera código en lenguaje objeto para una máquina diferente de la que se está utilizando para compilar. Es perfectamente normal construir un compilador de Pascal que genere código para MS-DOS y que el compilador funcione en Linux y se haya escrito en C++.
- ***Compilador con montador.***
  - Compila distintos módulos de forma independiente y después los enlaza.
- ***Autocompilador.***
  - Está escrito en el mismo lenguaje que va a compilar.

- **Metacompilador.**
  - Se refiere a un programa que recibe como entrada las especificaciones del lenguaje para el que se desea obtener un compilador y genera como salida el compilador para ese lenguaje. Su principal dificultad radica en unir la etapa de análisis con la generación de código (etapa de síntesis).
  
- **Descompilador.**
  - Es un programa que acepta como entrada código máquina y lo traduce a un lenguaje de alto nivel, realizando el proceso inverso a la compilación.
  
- **Compilación dirigida por la sintaxis.**
  - Un compilador necesita contemplar muchas características además de las propias para conseguir el código objeto.

De los diversos tipos de compiladores antes mencionados, para este proyecto se utilizó un híbrido entre compilador de *una pasada*, de *optimización* y *dirigido por la sintaxis*. Al utilizar esta técnica el trabajo de la fase de síntesis es el foco del compilador, requiriendo mayor esfuerzo en su estudio y modelado, para obtener diseños óptimos, llegando a simplificar un poco su implementación obteniéndose como resultado compilador más eficiente.

En la Figura 3.9 se puede observar las fases de las que se compone un compilador, se divide en dos etapas: análisis y síntesis, ayudados de dos herramientas, manejador de errores y tabla de símbolos. La etapa de análisis esta compuesta por: el **análisis léxico** se recibe el texto completo (programa), éste análisis consiste en descomponer el programa en sus unidades mínimas (caracteres) para obtener la lista de lexemas (palabras reservadas, identificadores, operadores o símbolos propios del lenguaje), en la sección 3.2.1 se describe a detalle esta etapa.

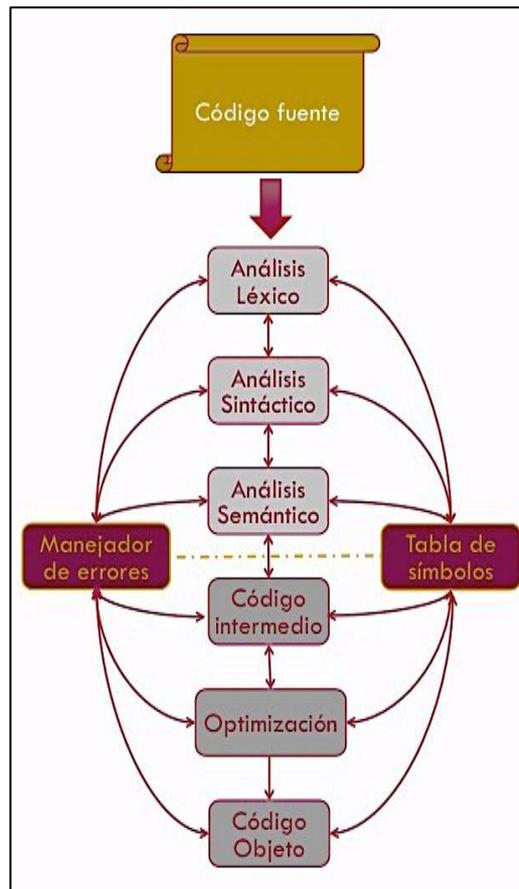


Figura 3. 9 Etapas para el desarrollo de un compilador, donde la etapa de análisis esta compuesto por: analizador léxico, sintáctico y semántico y la etapa de síntesis lo compone la generación de código intermedio, su optimización y la obtención de código objeto.

El **análisis sintáctico** consiste en “armar” las instrucciones, mediante gramáticas libres (sensibles/independientes) del contexto, mismas que ayudan a reducir al máximo cualquier ambigüedad y obteniendo un árbol sintáctico como resultado. Pudiéndose observar los detalles en la sección 3.2.2.

El resultado de la fase de síntesis es utilizado como entrada por el **análisis semántico**, para asegurar que se cumpla con una serie de reglas semánticas, además de realizar un posible renombramiento de identificadores para asegurar que cada uno corresponda a un único nombre (sección 3.2.3).

Una vez realizadas las etapas de análisis léxico, sintáctico y semántico (*fase de análisis*), se realiza la **fase de síntesis**, que es comprendido por las etapas de **generación de código intermedio** y **optimización del código intermedio** (sección 3.2.4) y **generación de código máquina** (sección 3.2.5).

### 3.2.1 ANÁLISIS LÉXICO

El objetivo principal de esta etapa es analizar el código de entrada para identificar y generar una lista de los lexemas y tokens. Para realizar esto, es necesario utilizar máquinas teóricas llamados autómatas finitos, a pesar de su poder limitado, son capaces de reconocer patrones de símbolos llamados: lenguajes regulares.

Como se muestra en la Figura 3.10, observamos que estos lenguajes se encuentran en el nivel más bajo en la jerarquía de **Chomsky**, son de interés fundamental para aplicaciones que requieren técnicas de reconocimiento de patrones, como es el caso del diseño y desarrollo de este compilador.

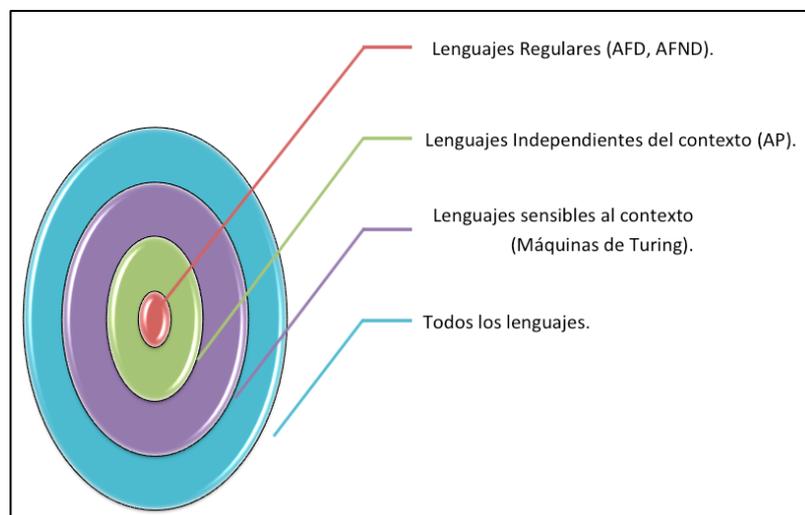


Figura 3. 10 Clasificación de los diferentes lenguajes determinada por Chomsky.

Para el desarrollo del compilador, primero se tiene que definir su lenguaje (palabras reservadas, identificadores, símbolos propios del lenguaje, etc.), es decir, desarrollar el analizador lexicográfico para poder representar de una manera concisa y no ambigua una estructura que acepte dicho lenguaje.

Un autómata finito está definido por la quintupla **(S, Σ, P, I, F)**, donde:

- **S** es el conjunto finito de estados.
- **Σ** es el alfabeto del autómata.
- **P** es un subconjunto de  $S \times \Sigma \times S$  (conjunto de transiciones).
- **I** (es un elemento de S) estado inicial.
- **F** es el conjunto de estados finales.

Como se mencionó anteriormente, estas máquinas teóricas sirven para reconocimiento de patrones en texto, tomando una cadena, la cual es aceptada Sí y sólo sí llega a un estado de aceptación al terminar de leer dicha cadena. Para realizar el análisis léxico es necesario identificar palabras reservadas, identificadores y símbolos (“;”, “(”, “)”, “+”, “\*”, “-”, etc.), que formarán parte del lenguaje del compilador.

Los autómatas finitos generar gramáticas regulares. Para comprender el significado de esto, es necesario hacer uso de ejemplos de lenguajes naturales, pues aquí se usan reglas gramaticales para distinguir aquellas cadenas de símbolos que son aceptados por durante el análisis léxico o no.

Un ejemplo sencillo del lenguaje natural como se muestra en la Figura 3.11 es el obtener una frase sencilla en español.

- 
- **<frase>** → **<sujeito>** **<predicado>** **<punto>**
  - **<sujeito>** → **<sustantivo>**
  - **<sustantivo>** → María
  - **<sustantivo>** → Juan
  - **<predicado>** → **<verbo transitivo>** **<objeto>**
  - **<predicado>** → **<verbo intransitivo>**
  - **<verbo intransitivo>** → patina
  - **<verbo transitivo>** → golpea
  - **<verbo transitivo>** → quiere
  - **<objeto>** → a **<sustantivo>**
  - **<punto>** → .
- 

Figura 3. 11 Gramática que produce frases complejas y completas en el lenguaje español.

La descomposición descendente se puede llegar a hacer a un nivel de detalle máximo donde se describan todos los posibles símbolos literales que aparecen en una frase en español, obteniendo como un ejemplo de ello la frase: “*María quiere a Juan*”.

Se puede observar en la Figura 3.11 en las derivaciones de cada parte de la frase términos encerrados entre corchetes agudos “<...>”, son llamados símbolos **no terminales** (los cuales pueden tener una o varias derivaciones más) y algunos que no están dentro de esos corchetes se les llama **símbolos terminales**.

Uno de los símbolos *no terminales* se toma como símbolo de inicio, cada línea se llama regla de reescritura (o regla de producción) están formadas por dos partes conectadas por una flecha entre las mismas, del lado derecho representa una descripción más detallada de su lado izquierdo.

En la representación de símbolos terminales y no terminales, además de utilizar los corchetes agudos, también se puede hacer uso de *minúsculas* para los símbolos *terminales* y de *mayúsculas* para *no terminales*.

Este conjunto de reglas de producción son llamadas **gramáticas estructuradas por frases**. De manera formal, una gramática se define como una cuádrupla **G (V,T,S,R)**, donde:

- **V** es el conjunto finito de símbolos no terminales.
- **T** es un conjunto finito de símbolos terminales.
- **S** (elemento de V) símbolo inicial.
- **R** es un conjunto finito de reglas de producción (reescritura).

Si los símbolos terminales de una gramática  $G$  son símbolos del alfabeto  $\Sigma$ , se dice que  $G$  es una gramática del alfabeto  $\Sigma$ . Las cadenas que se generen por  $G$  son en realidad cadenas de  $\Sigma^*$ . Por lo tanto una gramática  $G$  de un alfabeto  $\Sigma$  especifica un lenguaje de  $\Sigma$  que consiste en las cadenas generadas por  $G$ . A este lenguaje se le representa como  $L(G)$ .

Una gramática regular se puede definir como aquella gramática cuyas reglas de producción se adhieren a las siguientes restricciones representadas en la Figura 3.12.

---

$$\begin{aligned} S &\rightarrow x S \\ S &\rightarrow y Y \\ Y &\rightarrow y \\ Y &\rightarrow \lambda \end{aligned}$$

---

**Figura 3. 12 Gramática regular correspondiente al lenguaje  $L(G)=\{ x^m y^n: m, n \in \mathbb{N} \}$ .**

En la Figura 3.12 se muestra que una gramática regular debe estar formada por su lado izquierdo por un sólo símbolo no terminal y del lado derecho un terminal y un no terminal, un sólo terminal o por la cadena vacía ( $\lambda$ ).

Los lenguajes regulares se han definido como aquellos que son reconocidos por autómatas finitos, mismos que tienen su equivalencia con las gramáticas regulares, con la diferencia que las gramáticas regulares proporcionan mayores detalles acerca de la composición de dichos lenguajes.

### 3.2.1.1 TABLA DE TRANSICIONES

Para desarrollar una unidad de programa que reconozca las ocurrencias de nombres de variables (identificador), el primer paso que se debe dar es representar de manera concisa y no ambigua los identificadores válidos, para ello se hace el diseño de un autómata finito como se muestra en la Figura 3.13.

Para que un identificador sea considerado como válido, es necesario que inicie con una letra, seguido por una secuencia de numeros o letras. En caso de que se comience con un numero y después haya una secuencia de numero y letras (cualquiera que sea), sera determinado como un identificador incorrecto.

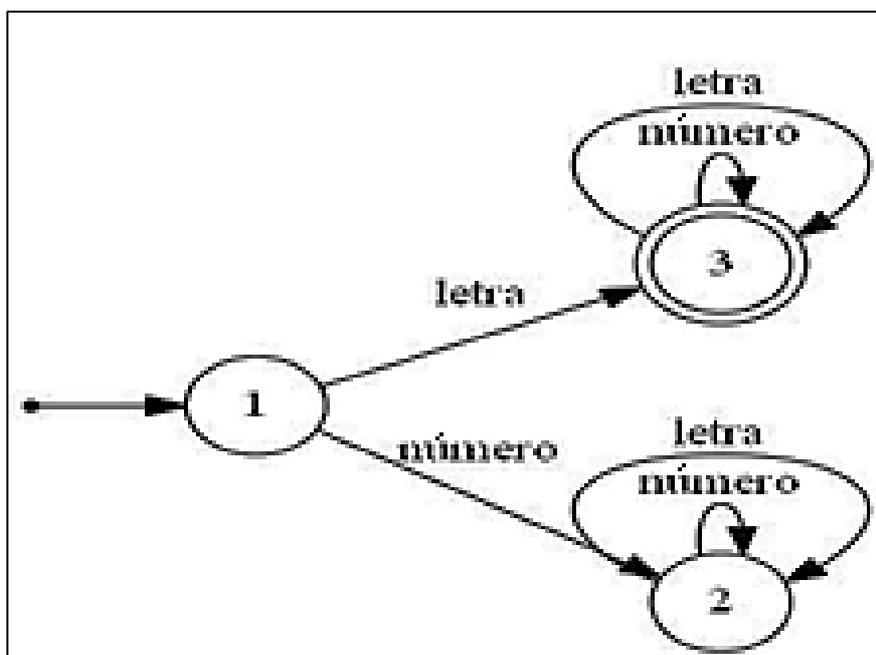


Figura 3. 13 Autómata finito que reconoce identificadores válidos.

El algoritmo correspondiente para la Figura 3.13 que reconoce identificadores válidos es mostrado en la Figura 3.14.

---

```
Inicio
  Estado=1
  Leer el siguiente símbolo de la cadena
  Mientras no se termine de leer la cadena Hacer
    Caso Estado:
      Estado 1:
        Si símbolo actual es una letra
          Estado=3
        Sino
          Si símbolo actual es un digito
            Estado=2
          Sino
            Error, no se reconoce cadena válida
      Estado 2:
        Error, no se reconoce cadena como identificador
      Estado 3:
        Si símbolo actual es una letra
          Estado=3
        Sino
          Si símbolo actual es un digito
            Estado=3
          Sino
            Error, no se reconoce cadena válida
    Leer el siguiente símbolo de entrada
  Fin_Mientras
  Si Estado es diferente de 3
    Error, no es un identificador válido.
Fin
```

---

**Figura 3. 14 Algoritmo general para generar un autómata finito determinista que acepte identificadores válidos.**

La **tabla de transiciones** es una matriz que proporciona el resumen del diagrama de transiciones correspondiente. El elemento que se encuentre en la fila  $m$  y en la columna  $n$  es el estado que se alcanzaría en el diagrama de transiciones al dejar el *estado  $m$*  a través de un *arco con etiqueta  $n$* , sino existe algún arco que salga del *estado  $m$* , no habrá casilla correspondiente de la tabla, por lo cual se marca como un estado de error. Es sencillo diseñar un analizador léxico basándose en una tabla de transiciones. Se tiene que asignar un valor (que va a corresponder al estado inicial) y actualizar el valor de esta conforme se va a analizando el texto de entrada. En las Tabla 3.2 se muestra la tabla de transiciones, correspondiente al autómata de la Figura 3.13.

Tabla 3. 2 Tabla de transiciones correspondiente al autómata de la Figura 3.13

	Letra	Digito	FDC
1	3	2	Error
2	Error	Error	Error
3	3	3	Aceptar

En la Figura 3.15 se muestra el algoritmo correspondiente a este mismo autómata que acepta nombre de variables válidas.

---

```

Estado:=1
Repetir
  Leer el siguiente símbolo de la cadena
  En caso de que símbolo sea:
    Letra: Entrada:="Letra"
    Digito: Entrada:="Digito"
    Fin de cadena: Entrada:="FDC"
  Estado:=Tabla [Estado, Entrada]
  Si Estado="Error"
    Salir
Hasta Estado="aceptar"

```

---

**Figura 3. 15 Algoritmo del autómata finito que reconoce identificadores válidos.**

Se ha definido a los lenguajes regulares como aquellos que son reconocidos por autómatas finitos, y también son generados por gramáticas regulares. Pero se tiene una caracterización más de los lenguajes regulares que proporciona mayores detalles, necesarios para llevar a cabo su programación, las expresiones regulares. Entonces se puede llegar a la conclusión que existe una equivalencia entre los autómatas finitos, gramáticas regulares y expresiones regulares y se puede pasar de uno a otro mediante su respectivo proceso.

### 3.2.2 ANÁLISIS SINTÁCTICO

El **análisis sintáctico** tiene como objetivo identificar instrucciones con base en el resultado del análisis anterior (léxico), basados en el uso de los autómatas de pila. En la Figura 3.2 observamos que los lenguajes independientes (libres o sensibles) del contexto corresponden al segundo nivel de la jerarquía de **Chomsky**, donde se aceptan lenguajes que no se reconocían con los autómatas finitos; es decir, los autómatas de pila son más poderosos, debido que poseen una “memoria” (pila) que permite (en el caso de los compiladores) “balancear” los caracteres de algunas instrucciones de una o varias instrucciones anidadas.

Los autómatas finitos, como se ha venido mencionando reconocen lenguajes regulares, utilizados para el reconocimiento de patrones en texto, como el ejemplo de los identificadores válidos  $[a-z A-Z] ([a-zA-Z0-9])^*$ , aquí solo se busca que cualquier cadena de caracteres sea reconocida como válida si inicia con una letra (mayúscula o minúscula) seguida de cualquier cantidad arbitraria finita de letras del alfabeto o números.

El analizador sintáctico “forma” instrucciones, en la Figura 3.16 se muestra la estructura de las instrucciones **Si** o **Si-SiNo**.

---

```

Si (condición)
{
    <Bloque de instrucciones>
}

Si (condición)
{
    Si (condición)
    {
        <bloque de instrucciones si>
    }
    SiNo
    {
        <bloque de instrucciones Sino>
    }
}

```

---

**Figura 3. 16 Estructura de las instrucciones Si, Si-SiNo.**

En la Figura 3.16 se muestran las estructuras de instrucciones de control, mismas que llevan caracteres como “(”, “)”, “{” y “}” los cuales requieren estar “balanceados”, es decir, por cada paréntesis o llave que se abra, se requiere uno que cierre la expresión para poder estar consideradas como instrucciones correctas.

Los autómatas de pila poseen una “*memoria*”, al hacer uso de ella, nos permite recordar la cantidad de paréntesis o llaves abiertas, o en su defecto si se cerraron más de los que se abrieron.

Un autómata de pila está definido por la séxtupla **(S, Σ, Γ, T, I, F)**, donde:

- **S** es el conjunto finito de estados.
- **Σ** es el alfabeto del autómata.
- **Γ** es la colección finita de símbolos de pila.
- **T** es un conjunto finito de transiciones.
- **I** (es un elemento de S) estado inicial.
- **F**(es un subconjunto de S) es el conjunto de estados finales.

En la Figura 3.17 se muestra un autómata que acepta cadenas cuyo lenguaje es  $L(G)=\{x^m y^m: m \in \mathbf{N}\}$  (cadenas que contengan una cantidad arbitraria finita de “x” seguida de la misma cantidad de “y”). Donde se puede observar que se inicia insertando el símbolo “#” para pasar al estado 2, si se lee una “x” cuando se encuentra en el estado 2, la máquina se inserta una “x” (puede ser cualquier símbolo) y se mantiene en el estado 2 (leyendo cualquier cantidad de “x” metiendo la misma cantidad de símbolos a la pila), en caso se leer una “y” se saca la “x” de la pila y se mantiene en el estado 3 cuando ya haya la misma cantidad de “x” y de “y”, el único símbolo restante en la pila será el “#”, de esta manera se pasa al estado 4 sacando este símbolo de la pila, en este caso llegando al estado de aceptación.

Esto no quiere decir que el autómata llegue a un estado de aceptación inmediatamente después de leer el último carácter de la cadena a analizar, si es el caso de no llegar a un estado de aceptación, significa que sigue habiendo símbolos en la pila y/o no están balanceados los símbolos en la cadena para lo cual fue desarrollado el autómata.

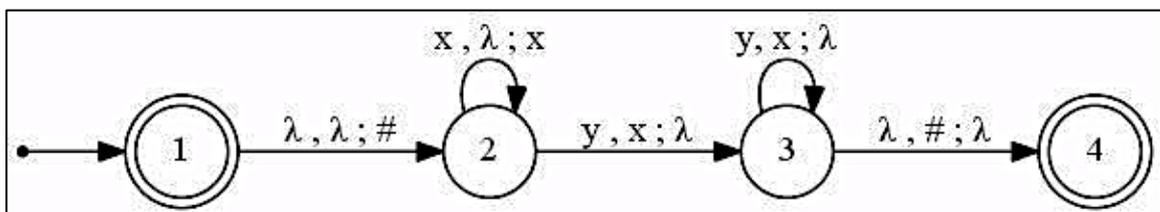


Figura 3. 17 Autómata de pila que reconoce la expresión  $(xy)^*$ .

En la Figura 3.18 se muestra el ejemplo canónico de un autómata de pila como aceptador de lenguaje, en este caso cadenas palíndromas con un alfabeto del autómata  $\{a,b,c\}$ . Las cadenas aceptadas (palíndromas) son todas aquellas que lleguen al estado 7.

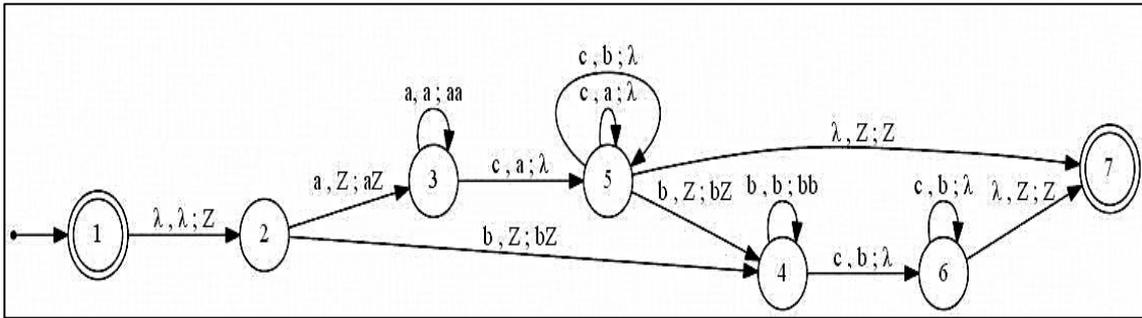


Figura 3. 18 Autómata de pila que reconoce cadenas palíndromas de longitud par.

En los dos ejemplos anteriores los autómatas de pila (AP) nos sirven para analizar cadenas de forma similar a los autómatas finitos, con la diferencia que los AP “recuerdan” la cantidad de símbolos leídos, en este caso se requieren para instrucciones de control como las que se muestran en la Figura 3.16, donde se quiere aceptar instrucciones completas con símbolos pares (apertura y cierre) de paréntesis y llaves.

Siguiendo el camino de los autómatas finitos, el lenguaje aceptado por los autómatas de pila es llamado  $L(M)$ , remarcando que este lenguaje no es cualquier colección de cadenas aceptadas por  $M$ , sino a la colección de todas las cadenas que acepta  $M$ .

Una vez explicado y ejemplificado el funcionamiento de los autómatas de pila y el lenguaje que reconoce. Se explicarán las gramáticas que generan ese mismo lenguaje, llamadas gramáticas independientes (libres) del contexto.

Las **gramáticas independientes del contexto (GIC)** a diferencia de las *gramáticas regulares*, no tienen restricciones con respecto a su escritura del lado derecho, pues las *gramáticas regulares* requieren *un terminal* y *un no terminal*, *un terminal* o una *cadena vacía* ( $\lambda$ ).

En las *GIC* no hay limitantes en cuanto a cantidad y orden de los símbolos terminales. Aunque ambas gramáticas siguen requiriendo un solo no terminal del lado izquierdo.

Estas gramáticas son llamadas “*independientes del contexto*” porque el lado izquierdo de la gramática (*un solo no terminal*) puede aplicarse sin importar el contexto donde se encuentre dicho *no terminal*. Estas gramáticas al igual que las regulares producen cadenas mediante derivaciones.

Al momento de hacer las derivaciones pueden surgir dudas con respecto a cuál será el *no terminal* que deberá reemplazarse en un paso específico de la derivación, se pueden hacer ***derivaciones por la izquierda*** (sustituyendo el no terminal que está situado más a la izquierda) o ***derivaciones por la derecha*** (haciendo lo propio con el de la derecha).

Independientemente de la derivación que se apliquen las reglas de reescritura no se afecta la determinación de si una cadena puede generarse a partir de cierta gramática. Las gramáticas independientes del contexto son exactamente los mismos lenguajes que aceptan los autómatas de pila.

Las derivaciones llevan a la construcción de ***árboles de análisis***, estos no son más que un árbol cuyos nodos representan los símbolos *terminales* y *no terminales* de la gramática, donde el nodo raíz es el símbolo de inicio y los hijos de cada no terminal son los símbolos que lo reemplazan. Estos árboles tienen dos reglas: Ningún símbolo no terminal puede ser una hoja, ni ningún símbolo terminal puede ser un nodo interior del árbol.

En la Figura 3.19 se presenta un ejemplo de una gramática independiente del contexto que genera el mismo lenguaje que una regular como la mostrada en la Figura 3.12.

---


$$S \rightarrow x S y$$

$$S \rightarrow \lambda$$


---

Figura 3. 19 Gramática que produce la expresión  $(xy)^*$ .

### 3.2.2.1 FORMA NORMAL DE CHOMSKY

Si bien las *GIC* no imponen muchas reglas de escritura, los lenguajes independientes del contexto si tienen gramáticas cuyas reglas de escritura se adhieren a formatos muy estrictos, permitiendo que varias gramáticas generen el mismo lenguaje.

Una *GIC*  $\mathbf{G(N,T,P,S)}$  que no genera la cadena vacía, está en la Forma Normal de Chomsky (FNC) cuando todas sus reglas tienen la forma:

$$A \rightarrow BC$$

$$A \rightarrow a$$

Donde  $A$ ,  $B$  y  $C$  son símbolos *no terminales* y  $a$  es símbolo *terminal*, es decir las gramáticas *Forma Normal de Chomsky (FNC)* requieren tener un símbolo *no terminal* del lado izquierdo y del lado derecho dos *no terminales* o bien un solo *terminal*. Todo lenguaje independiente del contexto que no incluya la cadena vacía es generado por una gramática *FNC (GFNC)*.

El proceso de convertir una *GIC* a *GFNC* se muestra en la Figura 3.20, como entrada se requiere una gramática  $\mathbf{G(N,T,P,S)}$  sin cadenas vacías y como resultado se obtendrá  $\mathbf{G(N'',T'',P'',S)}$ .

---

---

Paso 1

$N' = N$

$P' = \emptyset$

Para toda regla  $(A \rightarrow \alpha)$  de P hacer

Si  $|\alpha| = 1$

Añadir la regla a  $P'$

Sino sea  $\alpha = X_1 X_2 \dots X_m : m > 1$

Para  $i=1$  hasta  $m$  hacer

Si  $X_i = a \in \Sigma$

Se añade a  $N'$  un nuevo no terminal  $C_a$  y se añade a  $P'$  una nueva regla  $(C_a \rightarrow a)$

Se añade a  $P'$  una regla  $(A \rightarrow X'_1 X'_2 \dots X'_m)$  con:

$X'_i = X_i$  si  $X_i \in N$

$X'_i = C_a$  si  $X_i \in \Sigma$

Paso 2 (toma como entrada la gramática  $G'$  resultante del Paso 1)

$N'' = N'$

$P'' = \emptyset$

Para toda regla  $(A \rightarrow \alpha)$  de P hacer

Si  $|\alpha| < 3$

Añadir la regla a  $P''$

Sino sea  $\alpha = B_1 B_2 \dots B_m : m > 2$

Para  $i=1$  hasta  $m$  hacer

Añadir a  $N'$  un nuevo no terminal  $\{D_1, D_2, \dots, D_{m-2}\}$

Se añade a  $P''$  el siguiente conjunto de reglas:

$A \rightarrow B_1 D_1$

$D_1 \rightarrow B_2 D_2$

...

$D_{m-3} \rightarrow B_{m-2} D_{m-2}$

$D_{m-2} \rightarrow B_{m-1} D_m$

La gramática resultante es:  $G''(N'', T, P'', S)$

---

Figura 3. 20 Algoritmo que convierte una GIC a una GFNC.

### 3.2.2.2 PRINCIPIO DEL PREANÁLISIS

Después del análisis de los autómatas de pila, los lenguajes que producen y sus respectivas gramáticas. Se pueden desarrollar rutinas de análisis sintáctico a partir de los autómatas de pila.

El ***principio de preanálisis*** es una técnica que permite observar los símbolos siguientes sin leerlos para predecir cual derivación se realizará, lo cual permite superar el ***no determinismo*** de los autómatas de pila. Existen dos tipos de analizadores sintácticos los ***LL (k)*** y ***LR (k)***.

#### ANALIZADORES SINTÁCTICOS LL (K)

Son analizadores sintácticos predictivos y descendentes. Predictivos porque ayudan a predecir cual regla de ***reescritura*** se aplicará y descendentes porque insertan en la pila el símbolo equivalente a la cadena de entrada.

Son llamados ***LL(k)*** por la siguiente razón.

L → Lee la cadena de entrada de izquierda a derecha.

L → Produce una derivación por la izquierda.

k → Es el número de símbolos de preanálisis necesarios.

Cuanto mayor sea k, mayor potencia se tendrá (se podrá reconocer más lenguajes).

El algoritmo genérico para realizar un analizador ***LL(k)*** se muestra en la Figura 3.21.

---

```
Si cima de pila = símbolo no terminal
    Aplicar regla de derivación
Si cima de pila = símbolo terminal y coincide con el símbolo actual de
la cadena
    Sacarlo de la pila, leer de la cadena de entrada (desplazar
cabeza lectora).
    Sino cadena rechazada
Si cima de pila = "#"
    Cadena aceptada
```

---

Figura 3. 21 Algoritmo para realizar un analizador sintáctico LL (k).

## ANALIZADORES SINTÁCTICOS LR (K)

Son analizadores sintácticos ascendentes, insertan en la pila el símbolo inicial de la gramática. Los lenguajes que se pueden reconocer con este tipo de analizadores son independientes de contexto *deterministas*.

Son llamados LR(k) por la siguiente razón.

L → Lee la cadena de entrada de izquierda a derecha.

R → Produce una derivación por la derecha.

k → Es el número de símbolos de preanálisis necesarios.

El algoritmo genérico para realizar un analizador LR(k) se muestra en la Figura 3.22.

---

Insertar # a la pila  
Insertar los símbolos terminales que aparecen a la entrada (desplazar cabeza lectora).  
Cuando los símbolos en la cima de la pila sean igual a la parte derecha de una regla, se sustituye por el símbolo no terminal en la izquierda de la regla.  
Si en la pila solo queda el axioma inicial y símbolo #, la cadena se acepta.

---

Figura 3. 22 Algoritmo para realizar un analizador sintáctico LR (k).

El proceso de preanálisis permite seleccionar entre posibles alternativas a la hora de realizar una derivación o desplazamiento.

### 3.2.2.3 FORMA NORMAL DE BACKUS

Se ha caracterizado los lenguajes independientes del contexto como aquellos que son generados por gramáticas independientes del contexto (GLC) y como aquellos que son aceptados por los autómatas de pila. No obstante no se ha considerado el límite de estos lenguajes. La principal limitante es que no se pueden reconocer expresiones compuestas que definan una sintaxis (combinando símbolos con diferentes jerarquías), pese a esta limitante el principio de preanálisis ayuda a que se generen gramáticas sin ambigüedad.

Las *GLC* no ambiguas pueden ser expresadas por gramáticas con la Forma Normal de Backus (**BNF**, por sus siglas en inglés), las cuales son utilizadas como notación para las gramáticas de los lenguajes de programación.

Un sistema de reglas de derivación, puede ser escrito como:

$$\langle \text{símbolo} \rangle ::= \langle \text{expresión}_1 \rangle \mid \langle \text{expresión}_2 \rangle \mid \dots \mid \langle \text{expresión}_N \rangle$$

Ej:  $\text{digit} ::= "0" \mid "1" \mid "2" \mid "3" \mid "4" \mid "5" \mid "6" \mid "7" \mid "8" \mid "9"$

Al igual que las gramáticas anteriormente mencionadas (regulares e independientes de contexto), se requiere de un símbolo no terminal del lado izquierdo y cualquier combinación entre símbolos terminales y/o no terminales del lado derecho.

### 3.2.3 ANÁLISIS SEMÁNTICO

Una vez que se llega a este punto, ya se identificaron las palabras reservadas, ya se verificó que estén completas las instrucciones (que tengan todos los parámetros).

Ahora es necesario analizar que las instrucciones tengan sentido, es decir, que no se vaya a realizar una operación (matemática, lógica o de control) entre una cadena de caracteres y un número, o una expresión booleana y otro tipo de dato.

Debido a que el compilador es un híbrido entre compilador de una pasada, de optimización y dirigido por la sintaxis, esta etapa es un poco más simple, debido a que se toma como base la estructura de las instrucciones, se verifica parámetro por parámetro su tipo haciendo uso de la tabla de símbolos (sección 3.2.7).

Aquí puede haber dos caminos, uno si se detectan tipos de datos incompatibles se informa de un *error*, dos en caso que además de que la instrucción esta correcta y los tipos de datos sean compatibles, se manda el resultado (árbol semántico) a la *fase de síntesis* (siguiente etapa, generación de código intermedio).

### 3.2.4 GENERACIÓN Y OPTIMIZACIÓN DE CÓDIGO INTERMEDIO

Se traduce la entrada a una representación independiente de la máquina pero fácilmente traducible al código objeto. Algunas de las representaciones más comunes son:

- Código en dos o tres direcciones.
- Código de pila.
- Representaciones en forma de grafo, mixtas, árboles de representación intermedia.

El código de alto nivel se analiza, en este punto ya se obtiene un código **léxica**, **sintáctica** y **semánticamente** correcto, por lo cual se procede a traducirlo a código ensamblador que entiende el microprocesador SOPHIA para el cual es desarrollado este compilador. El conjunto de instrucciones se muestran en la Tabla 3.1.

El *BNF* del lenguaje de alto nivel es traducido a ensamblador, en la Tabla 3.3 se muestra una traducción de una asignación del resultado de una suma en una variable. En esta sección se hace también uso de la tabla de símbolos para ubicar la localidad de memoria de las variables que se van creando y sus valores.

Tabla 3. 3 Traducción de una instrucción de alto nivel a lenguaje ensamblador.

Alto nivel	Ensamblador
X=14+7	LDI R0, 14
	LDI R1, 7
	ADD R0, R1
	LDI R7, 0
	STX [R7], R0

La optimización del código intermedio involucra tanto a la etapa de análisis (que tiende a ser más matemático) como la de síntesis (requiere técnicas más especializadas, como erradicación de código redundante). Por lo mismo se ha hecho de forma implícita en etapas anteriores.

### **3.2.5 GENERACIÓN DE CÓDIGO MÁQUINA**

Una vez obtenido el código intermedio y su optimización, es necesario generar el código objeto. Depende fundamentalmente de la arquitectura concreta para la que se esté desarrollando el compilador. En esta etapa se deben considerar algunos inconvenientes como:

- Elección de los modos de direccionamiento adecuados.
- Utilización eficiente de los registros.
- Empleo eficiente de la memoria caché.

El compilador se desarrolló para una arquitectura específica, por lo cual si se quiere implementar en otro procesador no podrá ser posible hacer uso de este. La arquitectura del procesador SOPHIA se muestra en la Figura 3.1.

### **3.2.6 MANEJADOR DE ERRORES**

Si se hace el diseño e implementación de un compilador donde solo se consideren programas correctos sería muy fácil, pues sólo bastaría con encontrar las expresiones regulares y llevarlas a su **BNF** para su programación.

Pero es muy complicado que una persona realice siempre bien sus programas a la primera, debido a varios inconvenientes, en gran medida responsabilidad del usuario: puede ser que no se haya entendido el problema a resolver, las instrucciones no son las indicadas, instrucciones mal escritas, etc.

A continuación se muestran 4 grupos en los que se pueden categorizar los errores, una breve descripción y ejemplos de cada uno de ellos:

1. **Léxicos.** Se detectan cuando se intenta reconocer componentes léxicos y no se logra reconocer la cadena de entrada porque no hay patrón alguno que lo haga, esto se debe a que se hace uso de caracteres que no forman parte del lenguaje.

- a. *Identificadores, palabras reservadas mal escritas.*

2. **Sintácticos.** Son el tipo de errores más complicado de detectar desde el punto de vista del diseño. Son los indicados para mostrar de manera clara y precisa todo lo referente a los errores.

- a. *Uso de minúsculas.*

- b. *Falta de parámetros.*

3. **Semánticos.** Son detectados en tiempo de compilación.

- a. *Inconsistencia de tipos.*

- b. *Comprobación de flujo de control.*

4. **Lógicos.** Referentes al diseño del programa para resolver un problema. Donde el compilador no tiene el control del resultado a obtener, pero sí de que las operaciones se realicen, es decir, los ciclos se detendrán hasta cuando el usuario lo desee mediante su programación.

- a. *Uso de estructuras iterativas como do while, while o for de forma infinitamente recursiva.*

Los tres primeros los puede detectar e informar el compilador, pero el restante grupo de errores, ¡No!, pues depende más de la lógica del programador y su forma de abstraer el problema a resolver.

Un compilador debe ser diseñado de manera que los errores se puedan detectar, informar y recuperarse para analizar el código hasta el final.

### 3.2.7 TABLA DE SIMBOLOS

Con el paso de las etapas de análisis se va generando gran cantidad de información que se puede considerar en cualquier etapa posterior del compilador ligada a los datos (variables, constantes, instrucciones, etc.) que se van descubriendo con el paso de la compilación y ejecución del programa.

Etapas como el análisis léxico y sintáctico aportan la información necesaria para llenar la Tabla 3.4, en el análisis semántico se accede a esta tabla para verificar que se hayan creado los identificadores, su tipo, ubicación y para validar que las instrucciones tengan sentido. Así mismo, en la etapa de síntesis, específicamente la generación de código intermedio se hace la verificación del tipo, valor y posición de cada variable.

Tabla 3. 4 Tabla de símbolos del compilador.

Posición	Identificador	Valor	Tipo	Etapas

### **3.3 SIMULADOR**

Con base en los resultados obtenidos en el desarrollo del compilador se tienen los parámetros necesarios para mostrar de manera gráfica el funcionamiento interno del procesador SOPHIA.

Las instrucciones de alto nivel, corresponden a una serie de instrucciones en lenguaje ensamblador, las cuales se dividen en cinco grupos (saltos, un registro con un dato inmediato, dos registros, un registro y un nemónico), debido a la naturaleza de las mismas.

## CAPÍTULO 4. ENTORNO DE DESARROLLO INTEGRADO

El IDE se compone de un compilador en alto nivel y un simulador. Los cuales se explican a continuación.

Las expresiones regulares son las utilizadas para programarse y llevar a cabo la fase de análisis léxico, en la Figura 4.1 se muestran algunas expresiones utilizadas en el desarrollo del analizador léxico.

- 
- `Entero = 0 | [1-9][0-9]*`
  - `Rnv = R8|R9|R[1-9]([0-9])+`
  - `Rnvm = r8|r9|r[1-9]([0-9])+`
  - `Identificador = [a-zA-Z][a-zA-Z0-9]*`
  - `CaracterCadena = [^\r\n\"\\] //Para poder detectar "`
- 

Figura 4. 1 Expresiones regulares utilizadas para generar el analizador léxico.

La generación del análisis léxico mediante autómata finito se puede observar en la Figura 4.2.

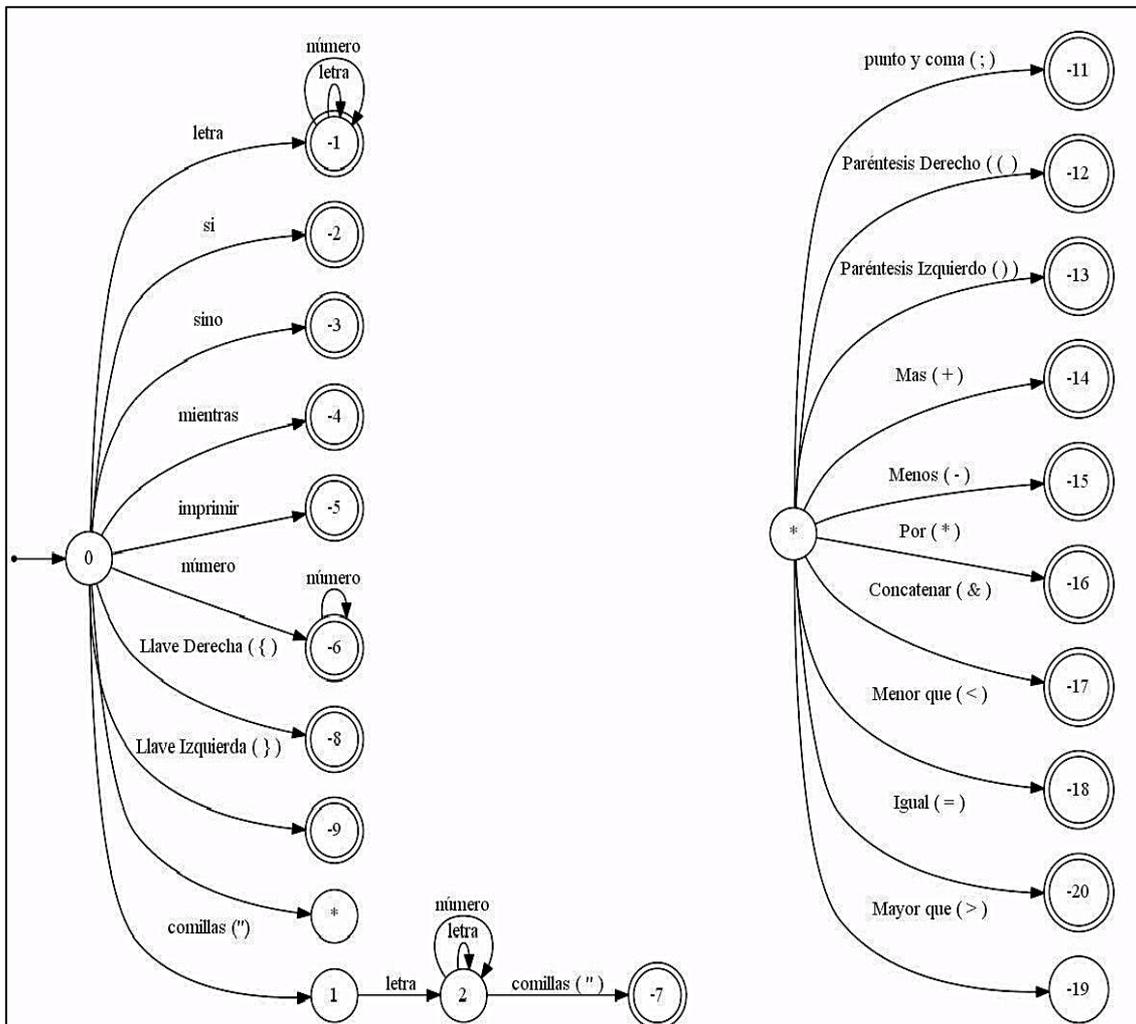


Figura 4. 2 Autómata finito determinista correspondiente al análisis léxico.

El resultado del análisis léxico, es la detección de las palabras reservadas y símbolos del lenguaje, con las cuales se podrá realizar operaciones aritméticas (suma, resta y multiplicación), operaciones lógicas (mayor que y menor que) e instrucciones de control (Si, Si-SiNo y mientras).

En la figura 4.3.a se muestra algunas de las gramáticas independientes del contexto utilizadas para el desarrollo del análisis sintáctico y en la Figura 4.3.b se muestra la estructura del programa.

---

a)

```

Mientras := HACER PDER condición PIZQ LLDER expmientras LLIZQ
Si := SI PDER condición PIZQ LDER expsi LIZQ
Sino := SI PDER condición PIZQ LDER expsi LIZQ
        SINO PDER condición PIZQ LDER expsi LIZQ

```

b)

```

hacer (<condición>)
{
    <expmientras>
}
si (<condición>)
{
    <expsi>
}

```

```

si (<condicion>)
{
    <expsi>
}
sino
{
    <expsino>
}

```

---

Figura 4. 3 a) Gramática BNF instrucciones de control del compilador.

b) Estructura de las instrucciones dentro del programa.

En la figura 4.4 se muestra el autómata de pila que genera las gramáticas mostradas en la figura 4.3.a.

Para hacer la validación del análisis semántico, se deben probar todos los casos posibles para aceptar o modificar las gramáticas BNF. Hacer la prueba de todas las diferentes combinaciones es una tarea muy larga. Por esa razón, fueron agrupados casos de similares características, reduciendo un poco el tiempo de validación de las gramáticas.

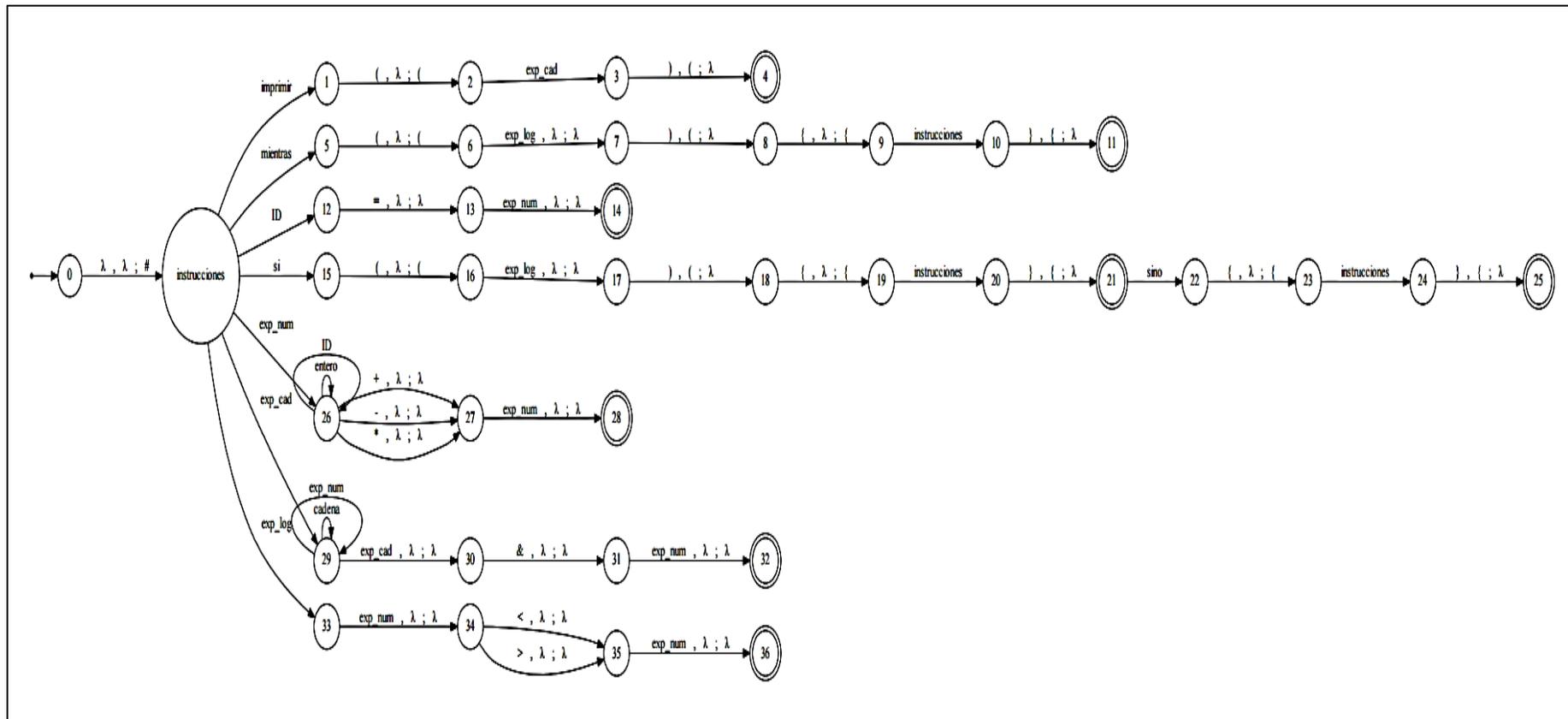


Figura 4. 4 Analizador sintáctico mediante un autómata de pila.

Cada instrucción aritmética (suma, resta o multiplicación) o lógica tiene la estructura:

Operando **operador** Operando,

Dónde: Operando puede ser una variable o un entero y operador (+, -, \*, < >).

De esta manera existen cuatro posibilidades diferentes de hacer cada operación (lógica o aritmética), las cuales se muestran a continuación.

Var + Var	Var * Var	Var - Var
Var + Entero	Var * Entero	Var - Entero
Entero + Var	Entero * Var	Entero - Var
Entero + Entero	Entero * Entero	Entero - Entero
Var < Var		Var > Var
Var < Entero		Var > Entero
Entero < Var		Entero > Var
Entero < Entero		Entero > Entero

Para hacer cada una de las operaciones antes mencionadas el analizador semántico, haciendo uso de la tabla de símbolos, realiza la validación de tipos.

Una vez que se realizó la fase de análisis y no se encontró ningún error se pasa a la etapa de síntesis. La que consiste en generar código intermedio, optimizarlo y generar código máquina.

La generación de código intermedio, es realizar la traducción del programa en lenguaje de alto nivel a lenguaje ensamblador reconocido por el procesador SOPHIA.

Cada instrucción en alto nivel, corresponde a una serie de instrucciones en ensamblador, en la Tabla 4.1 la traducción de una asignación a ensamblador.

Tabla 4. 1 Traducción de una asignación simple en alto nivel a lenguaje ensamblador.

Alto nivel	Ensamblador
X=14+7	LDI R0, 14
	LDI R1, 7
	ADD R0, R1
	LDI R7, 0
	STX [R7], R0

En la Figura 4.5 un fragmento de código correspondiente a la generación de código intermedio, es decir, el paso de la instrucción de alto nivel a su correspondiente a código ensamblador.

```

expr ::= expr:e OP_SUMA factor:f
{
  //System.out.println("s e: "+e+", f: "+f);
  Expression ex = new Expression(e, "+", f);
  int v=e.hashCode();
  String nom1 = e.toString();
  String nom2 = f.toString();
  char var1='';
  char var2='';
  if(nom1.length()!=1)
  {
    for(int zzz=0; zzz<nom1.length(); zzz++)
    {
      if(zzz==1)
      {
        var1=nom1.charAt(zzz);
        zzz=10;
        asignsimple=false;
      }
    }
  }
  if(asignsimple==true)
  {
    System.out.println("LDI R1, "+f);
  }
  else
  {
    for(int zzz=0; zzz<nom1.length(); zzz++)
    {
      if(zzz==1)
      {
        nom1=nom1.substring(1, nom1.indexOf("="));
        zzz=10;
        asignsimple=true;
      }
    }
    for(vjcont=0; vjcont<j; vjcont++)
    {
      if(alvarez[vjcont].equals(nom1))
      {
        System.out.println("LDX R0, "+valvarint[vjcont]+".....*****+*");
        tempsum=valvarint[vjcont];
        vjcont=j;
      }
    }
  }
}
//debe acceder a las variables para ver su posicion y reemplazar el nombre de la variable por su loca

```

Figura 4. 5 Fragmento de código correspondiente a la generación de código intermedio.

En el caso de la optimización de código intermedio, su propósito es minimizar la cantidad de instrucciones a ejecutar (hacer el código más eficiente). En la Tabla 4.2 se muestra una asignación del resultado de una multiplicación a una variable.

Tabla 4. 2 Traducción de una instrucción de alto nivel a lenguaje ensamblador.

Alto nivel	Ensamblador
X=2*5	LDI R7, 1 LDI R0, 2 LDI R1, 5 LDI R2, 0 MULT: ADD R0, R1 PUSH R1 SUB R2, R1 BRZ FINMULT POP R1 ADDI R2, 1 JMP MULT FINMULT STX [R7], R0

La optimización de código intermedio se realiza en la manera de cargar los valores en el orden adecuado, para disminuir la cantidad de operaciones totales, es decir, en el caso de la multiplicación se debe cargar el valor mayor en R0 y el menor en R1.

Para poder realizar cualquier operación (aritmética o lógica), se requiere al menos un registro y un entero u otro registro, dando esto pie a organizar los 8 registros que posee el procesado de la manera que se muestra en la Tabla 4.3.

Tabla 4. 3 Descripción del uso de los registros del procesador SOPHIA.

Registro	Descripción de uso	Registro	Descripción de uso
R0	Primer operando para operaciones aritméticas.	R4	Primer operando para operaciones lógicas.
R1	Segundo operando para operaciones aritméticas.	R5	Segundo operando para operaciones lógicas.
R2	Auxiliar multiplicación.	R6	Push / Pop.
R3	Almacena el valor de veces que se repetirán las instrucciones.	R7	Apuntador de variable a almacenar.

La generación de código máquina se realiza haciendo la traducción correspondiente tomando como base el conjunto de instrucciones Tabla 3.1, la arquitectura de la Figura 3.1 y la instrucción de la Tabla 4.1, se tiene como resultado el código máquina que puede verse en la Tabla 4.4.

Tabla 4. 4 Código máquina generado correspondiente a una asignación de una suma a una variable.

Alto nivel	Ensamblador	Código máquina
X=14+7	LDI R0, 14	01100 000 00001110
	LDI R1, 7	01100 001 00000111
	ADD R0, R1	10000 000 001 00000
	LDI R7, 0	01100 111 00000000
	STX [R7], R0	10101 111 000 00000

De color azul se muestran los bits correspondientes al mnemónico y al código de operación en código máquina; en verde los registros y su respectivo código máquina; de morado los correspondientes a constantes y su código binario finalmente de rojo los bits que no son utilizados.

En la Figura 4.6, se muestra la interfaz completa del compilador. La cual consta de 9 botones, un área de trabajo y un área de visualización de resultados y errores.

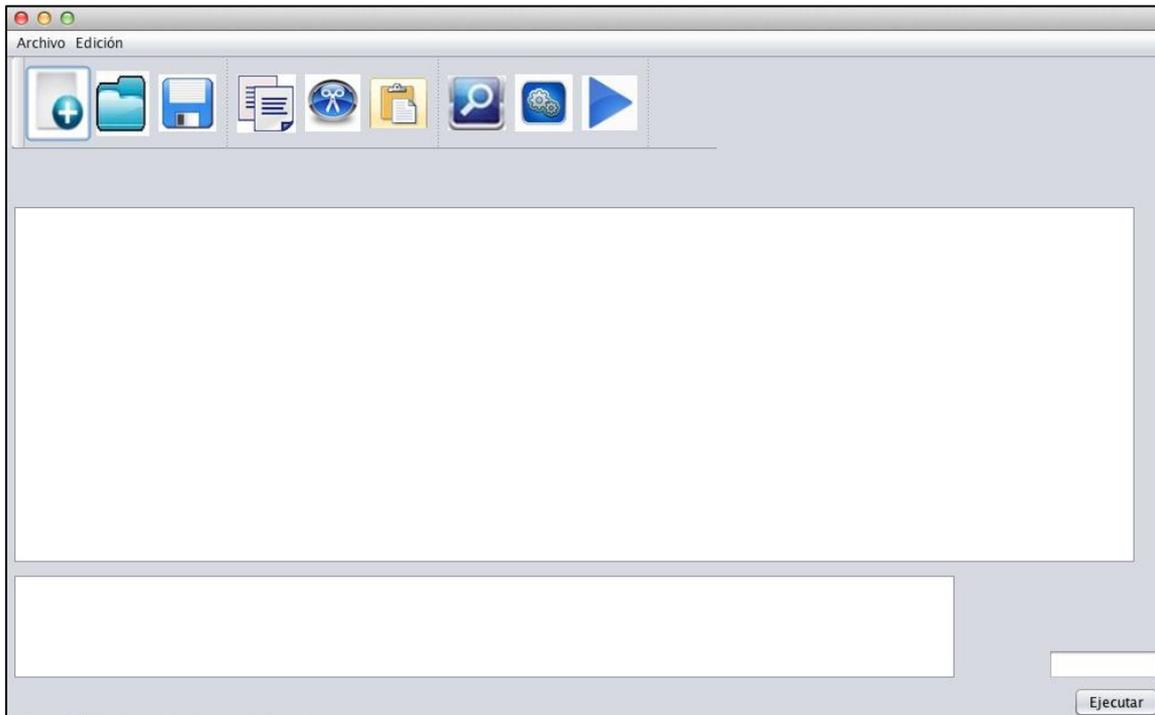


Figura 4. 6 Interfaz del compilador.

**Botón nuevo.** Permite crear un nuevo archivo, antes de crearlo, pregunta si se desea guardar el trabajo que se tiene actualmente en el área de trabajo; si se elige SI, pide que se guarde un archivo con extensión **.vj**; en caso que se elija NO, se perderá el trabajo, Figura 4.7.



Figura 4. 7 Botón que permite crear un nuevo programa.

**Botón abrir.** Selecciona un archivo .vj, para editarlo o compilarlo, Figura 4.8.



Figura 4. 8 Botón que permite abrir un programa ya existente.

**Botón guardar.** Sólo está activo si se ha hecho una modificación en el área de trabajo, el texto que se encuentra en el área de trabajo se podrá guardar con la extensión .vj, Figura 4.9.



Figura 4. 9 Botón que permite guardar el código escrito en el área de trabajo .

**Botones de edición: copiar, cortar y pegar.** Permite hacer lo propio, Figura 4.10a), Figura 4.10b) y Figura 4.10c), respectivamente.



Figura 4. 10a) Botón que permite copiar el texto seleccionado.



Figura 4. 11 Botón que permite cortar el texto seleccionado.

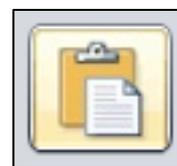


Figura 4. 12 Botón que pegar el texto que se haya copiado o cortado previamente.

**Botón compilar.** Verifica que el código escrito en el área de trabajo sea correcto, mostrará la leyenda “compilado” en el área de resultados; en caso contrario, se desplegará(n) el(los) error(es) en la sección de resultados, Figura 4.11.

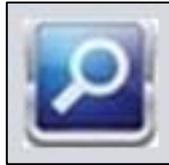


Figura 4. 13 Botón compilar para buscar errores.

**Botón ejecutar.** Se habilita cuando se encuentra con un programa correcto. Realiza la fase de síntesis, genera el lenguaje ensamblador y la traducción de este a código máquina, mismo que es guardado en un archivo .txt para su posterior programación en el procesador, Figura 4.12.



Figura 4. 14 Botón para ejecutar el programa.

Una vez diseñado y desarrollado el compilador se tiene el código en alto nivel analizado, habiendo generado código intermedio (en lenguaje ensamblador, basados en el conjunto de instrucciones) y su correspondiente código máquina, se tiene la base para poder realizar el simulador del procesador SOPHIA. En la Figura 3.1 se muestra la arquitectura del procesador, donde se mostrará la manera en la cual se propagan los datos en los diferentes bloques funcionales.

**Botón simular.** Se habilita después de haberse generado el código objeto. Se muestran los diferentes bloques funcionales del procesador y la manera en la cual se van propagando los datos y la activación de las banderas, Figura 4.13.



Figura 4. 15 Botón para simular el flujo de datos dentro del procesador.

En la Figura 4.14 se muestra la interfaz del simulador del procesador SOPHIA.

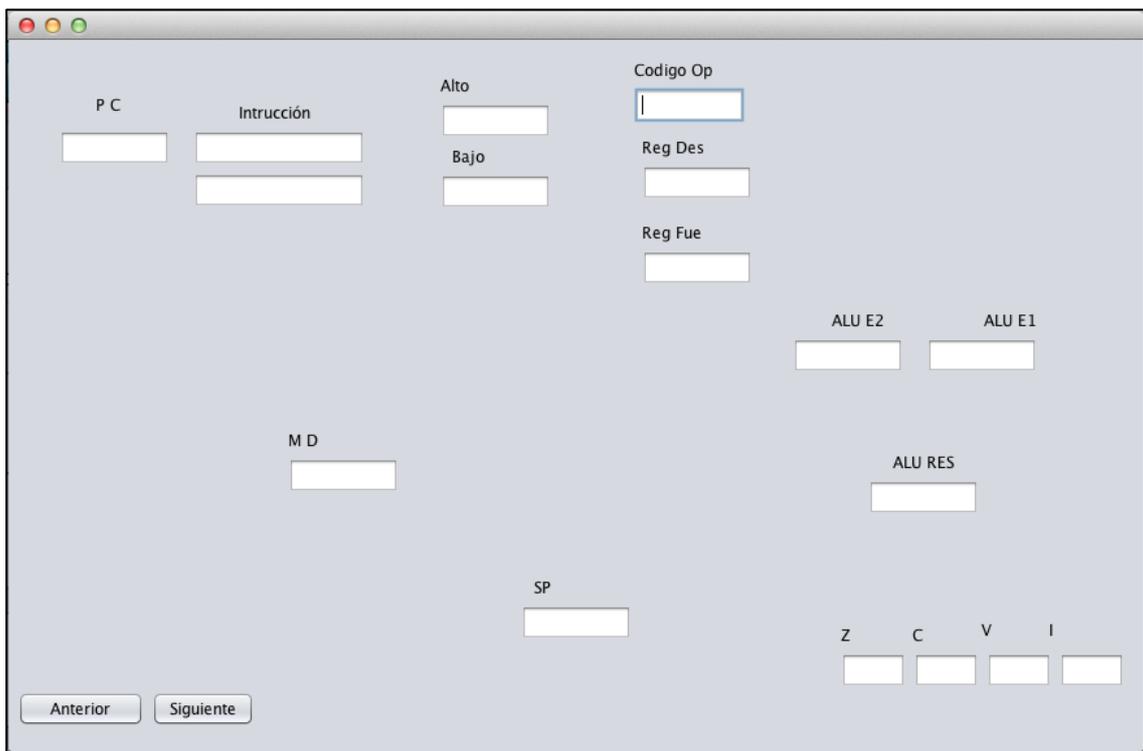
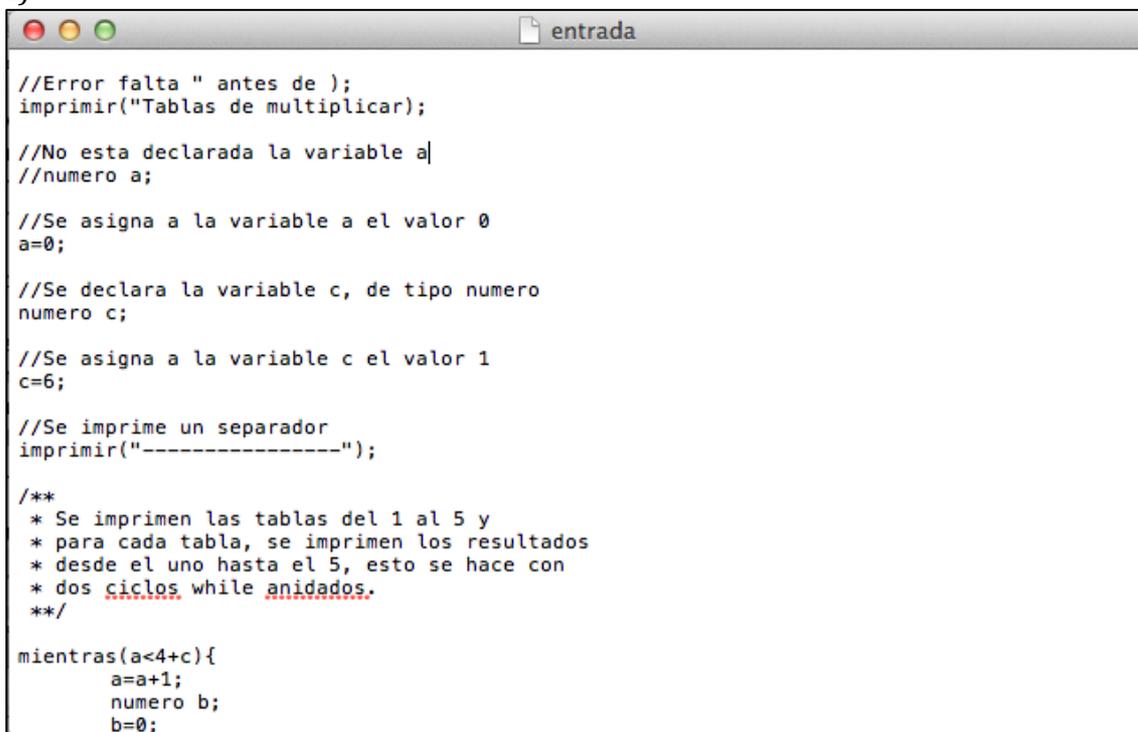


Figura 4. 16 Interfaz del simulador.

## CAPÍTULO 5. EXPERIMENTACIÓN Y RESULTADOS

Para poder validar el funcionamiento de IDE para el procesador SOPHIA, se debió de hacer una serie de pruebas (compilar y ejecutar diferentes códigos) para observar el informe de los errores. En la Figura 5.1a) se muestra un código con diferentes errores donde falta " en la instrucción imprimir y donde no está declarada la variable a, y en la Figura 5.1b) se muestra el informe de dichos errores.

a)



```
//Error falta " antes de );
imprimir("Tablas de multiplicar);

//No esta declarada la variable a|
//numero a;

//Se asigna a la variable a el valor 0
a=0;

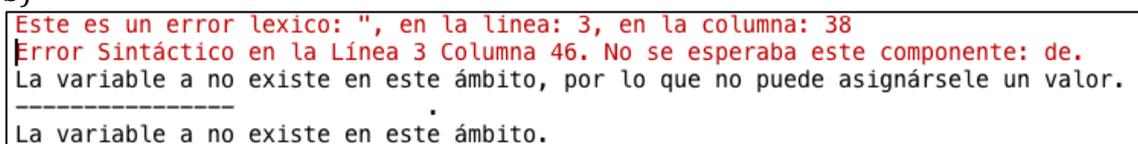
//Se declara la variable c, de tipo numero
numero c;

//Se asigna a la variable c el valor 1
c=6;

//Se imprime un separador
imprimir("-----");

/**
 * Se imprimen las tablas del 1 al 5 y
 * para cada tabla, se imprimen los resultados
 * desde el uno hasta el 5, esto se hace con
 * dos ciclos while anidados.
 */
mientras(a<4+c){
    a=a+1;
    numero b;
    b=0;
```

b)



```
Este es un error lexico: ", en la línea: 3, en la columna: 38
Error Sintáctico en la Línea 3 Columna 46. No se esperaba este componente: de.
La variable a no existe en este ámbito, por lo que no puede asignársele un valor.
-----
La variable a no existe en este ámbito.
```

Figura 5. 1 a) Código error imprimir y declaración de variable. b) Mensajes de errores.

En la Figura 5.2a) se muestra un código con diferentes errores donde esta mal escrita la instrucción imprimir y donde no está declarada la variable b debido a que no esta completa al faltar un “;”, y en la Figura 5.2b) se muestra el informe de dichos errores.

a)

```
//error mientras
mientras(a<4+c){
    a=a+1;
    //falta " ; " lo que no permite declarar bien la variable|
    numero b
    b=0;
    mientras(b<4+c){
        b=b+1;
        imprimir(a & " * " & b & " = " & a * b);
    }
    imprimir("-----");
}
```

b)

```
Error Sintáctico en la Línea 27 Columna 9. No se esperaba este componente: (.
Error Sintáctico en la Línea 30 Columna 3. No se esperaba este componente: b.
Error Sintáctico en la Línea 36 Columna 2. No se esperaba este componente: }.

La variable b no existe en este ámbito, por lo que no puede asignársele un valor.
La variable b no existe en este ámbito.
```

Figura 5. 2 a) Código error mientras y declaración de variable por falta de “;”. b) Menseajes de errores.

En la Figura 5.3a) se muestra un código donde se escribe la palabra “mas” en lugar de su símbolo “+”. En la Figura 5.3b) se muestra el informe de este error.

a )

```
//Ciclo mientras
mientras(a<4+c){           //error a mas 1      mas no se reconoce
    a=a mas 1;
    numero b;
    b=0;
    mientras(b<4+c){
        b=b+1;
        imprimir(a & " * " & b & " = " & a * b);
    }
    imprimir("-----");
}
```

b )

```
Error Sintáctico en la Línea 29 Columna 7. No se esperaba este componente: mas.
```

Figura 5. 3 a) Código error símbolo no identificado. b) Menseajes de errores.

En la Figura 5.4a) se muestra un código donde falta el símbolo de concatenación “&”, por esa razón el siguiente token no se reconoce, y en la Figura 5.4b) se muestra el informe del error.

a )

```
//Ciclo mientras
mientras(a<4+c){
    a=a + 1;
    numero b;
    b=0;
    mientras(b<4+c){
        b=b+1;
        //error se espera imprimir(a & " * " & b & " = " & a * b);      falta & es símbolo de concatenación
        imprimir(a & " * " b & " = " & a * b);
    }
    imprimir("-----");
}
```

b )

```
Error Sintáctico en la Línea 35 Columna 143. No se esperaba este componente: b.
```

Figura 5. 4 a) Código error falta de símbolo de concatenación. b) Menseajes de errores.

En la Figura 5.5a) se muestra un código donde falta el símbolo de concatenación "\*", por esa razón el siguiente token no se reconoce, y en la Figura 5.5b) se muestra el informe del error.

a )

```
//Ciclo mientras
mientras(a<4+c){
    a=a + 1;
    numero b;
    b=0;
    mientras(b<4+c){
        b=b+1;
        //error se espera imprimir(a & "*" & b & " = " & a * b); falta * en este caso falta un operador aritmético
        imprimir(a & "*" & b & " = " & a b);
    }
    imprimir("-----");
}
```

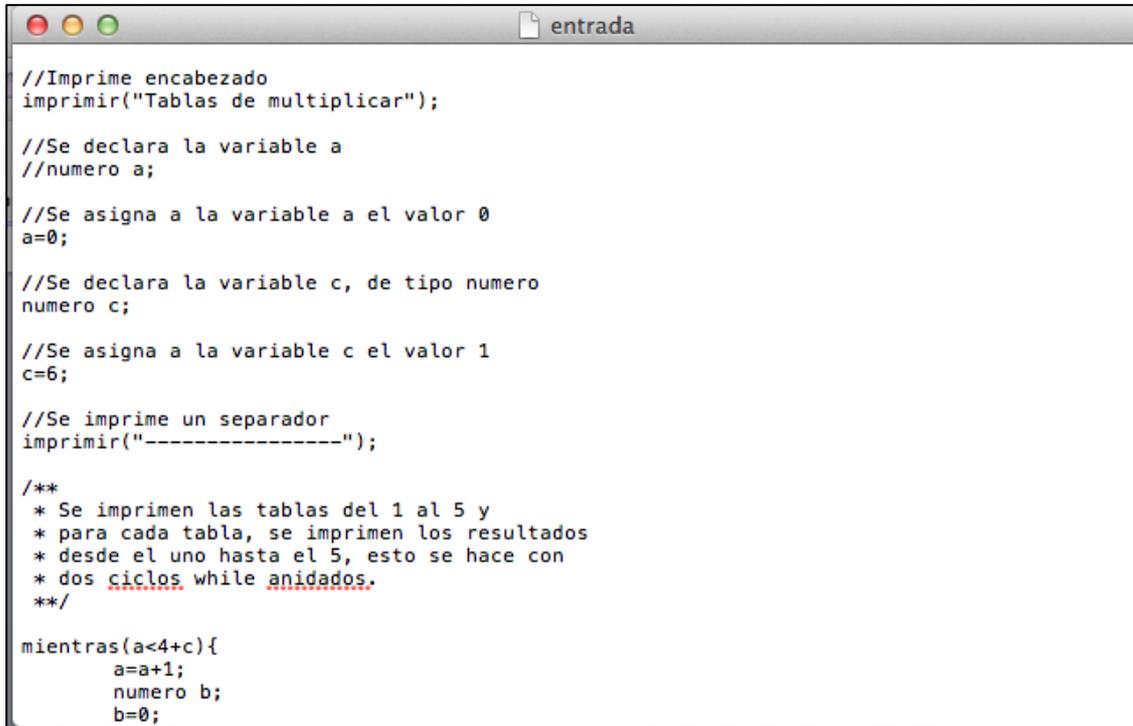
b )

```
Error Sintáctico en la Línea 35 Columna 158. No se esperaba este componente: b.
```

Figura 5. 5 a) Código error falta de token aritmético. b) Menseajes de errores.

En la Figura 5.6a) se muestra el informe de la ejecución del código ya sin errores y en la Figura 5.6b) se muestra el resultado de esa ejecución.

a)



```
//Imprime encabezado
imprimir("Tablas de multiplicar");

//Se declara la variable a
//numero a;

//Se asigna a la variable a el valor 0
a=0;

//Se declara la variable c, de tipo numero
numero c;

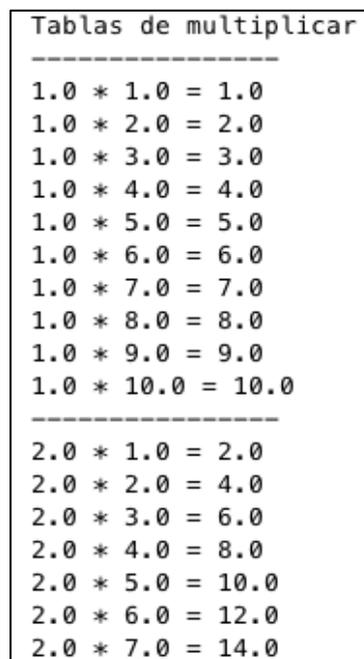
//Se asigna a la variable c el valor 1
c=6;

//Se imprime un separador
imprimir("-----");

/**
 * Se imprimen las tablas del 1 al 5 y
 * para cada tabla, se imprimen los resultados
 * desde el uno hasta el 5, esto se hace con
 * dos ciclos while anidados.
 */

mientras(a<4+c){
    a=a+1;
    numero b;
    b=0;
```

b)



```
Tablas de multiplicar
-----
1.0 * 1.0 = 1.0
1.0 * 2.0 = 2.0
1.0 * 3.0 = 3.0
1.0 * 4.0 = 4.0
1.0 * 5.0 = 5.0
1.0 * 6.0 = 6.0
1.0 * 7.0 = 7.0
1.0 * 8.0 = 8.0
1.0 * 9.0 = 9.0
1.0 * 10.0 = 10.0
-----
2.0 * 1.0 = 2.0
2.0 * 2.0 = 4.0
2.0 * 3.0 = 6.0
2.0 * 4.0 = 8.0
2.0 * 5.0 = 10.0
2.0 * 6.0 = 12.0
2.0 * 7.0 = 14.0
```

Figura 5. 6 a) Ejecución de código sin errores. b) Resultado de la ejecución de código sin errores.

En la Figura 5.7 la ejecución del simulador y la manera en que se propagan los datos.

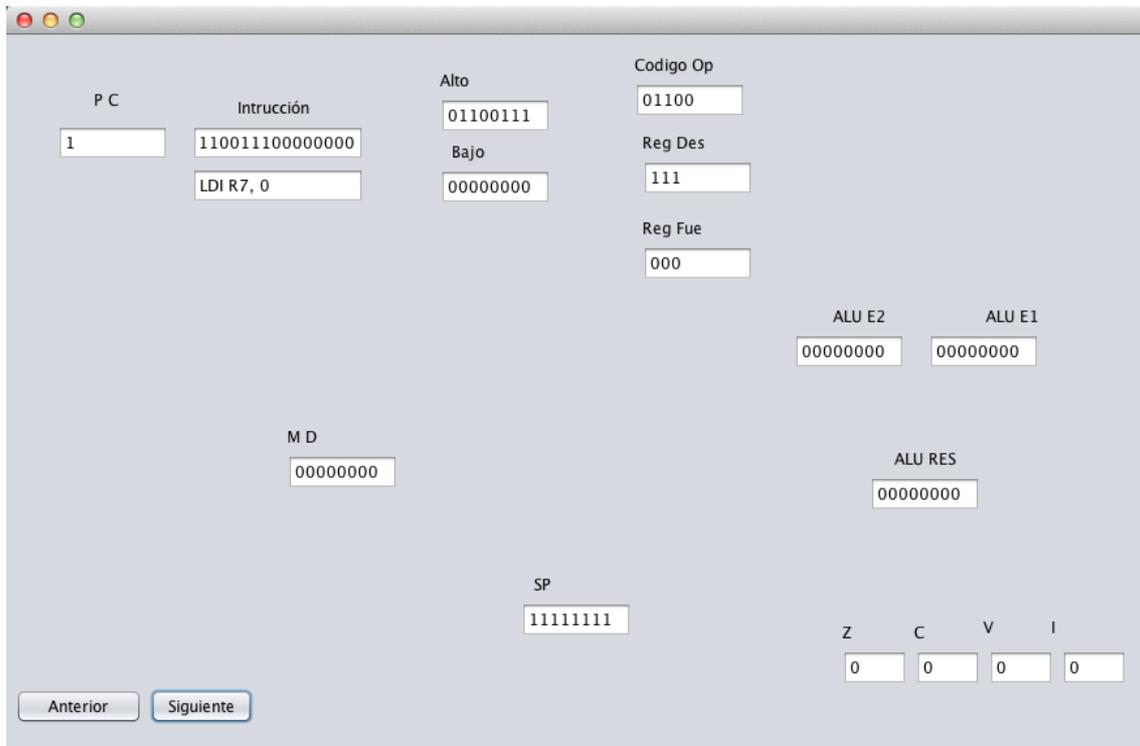


Figura 5. 7 a) Ejecución del simulador.

## **CAPÍTULO 6. CONCLUSIÓN**

Con el desarrollo del IDE se tiene una herramienta que puede ser utilizada en los cursos de arquitectura de computadoras como apoyo didáctico. El compilador ayuda a los estudiantes a programar (en alto nivel) el procesador SOPHIA, a diferencia de cómo se hacía en un principio programando en lenguaje máquina; lo cual facilita su programación, reduciendo su tiempo y complejidad de programación, así mismo, se pueden hacer multiplicaciones, operaciones lógicas y ciclos de manera más fácil mediante el alto nivel, a comparación del lenguaje ensamblador y lenguaje máquina.

Por su parte el simulador ayuda a comprender el flujo de datos, pues se observa la instrucción en ensamblador, su equivalente en lenguaje máquina y como se propaga la instrucción por cada uno de los bloques funcionales del procesador.

### **TRABAJO FUTURO**

Con la finalidad de realizar mejoras que ayuden a los alumnos de la asignatura de arquitectura de computadora, se propone que el flujo de los datos en el simulador sea mostrado con etiquetas en movimiento para facilitar mejor la manera en que se comunican los datos funcionales. Por otra parte, también se propone el uso de periféricos para que el alcance del procesador sea.

## REFERENCIAS:

- [1] R. A. Berkeley, E. C. & Jensen, "No Title," *World's Smallest Electric Brain. Radio-Electronics*. [Online]. Available: <http://www.vintagecomputer.net/simon.cfm>. [Accessed: 30-Aug-2014].
- [2] Illinois State University, "No Title," *Little Man Computer*, 2000. [Online]. Available: [www.acs.ilstu.edu/faculty/javila/lmc/](http://www.acs.ilstu.edu/faculty/javila/lmc/).
- [3] B. L. Record, "Cardboard ' Computer ' Helps Students," 1969.
- [4] S. Yukata, "Paper processor." [Online]. Available: <https://sites.google.com/site/kotukotuzimiti>.
- [5] G. G. Langdon Jr, *Computer Design*. 1982.
- [6] H. ElAarag, "A complete design of a RISC processor for pedagogical purposes.," *J. Comput. Sci. Coll.*, vol. 25, no. 2, pp. 205–2013, 2009.
- [7] J. L. Hennessy and D. A. Patterson., *Computer architecture: a quantitative approach*. 2012.
- [8] B. J. Rodriguez, "A minimal TTL processor for architecture exploration," *Proc. 1994 ACM Symp. Appl. Comput.*, pp. 338–340, 1994.
- [9] P. Verplaetse and J. Campenhout, "ESCAPE: Environment for the Simulation of Computer Architecture for the Purpose of Education," *IEEE Tech. Committe Comput. Archit. Newsl.*, pp. 57–59, 1999.
- [10] J. Djordjevic, A. Milenkovic, and N. Grbanovic, "An Integrated Environment for Teaching Computer Architecture," *IEEE Micro*, vol. 20, no. 3, pp. 66–74, 2000.

- [11] G. Wainer, S. Daicz, L. De Simoni, and D. Wassermann, "Using the Alfa-1 simulated processor for educational purposes," *Resour. Comput.*, vol. 1, no. 4, pp. 111–151, 2001.
- [12] C. A. Martins, J. B. Correa, L. F. Goes, L. E. Ramos, and T. H. Medeiros, "A new learning method of microprocessor architecture," *Front. Educ.*, vol. 3, 2002.
- [13] S. Pizzutilo and F. Tangorra, "A learning environment to teach computer architecture," *Proc. teach WSEAS Int. Conf. Inf. Sci. Appl.*, pp. 770–776, 2003.
- [14] M. Jaumain, M. Osee, A. Richard, A. Vander Biest, and P. Mathys, "Educational simulation of the RISC processor," *Int. Conf. Eng. Educ.*, 2007.
- [15] M. I. Garcia, S. Rodriguez, A. Perez, and A. Garcia, "A Graphical Simulator for Computer Architecture and Organization Courses," *IEEE Trans. Educ.*, vol. 52, no. 2, pp. 248–256, 2009.
- [16] Hsiao-Ping, N. D. Holmes, S. Bakshi, and D. D. Gajski, "Top-down modeling of RISC processors in VHDL," *Des. Autom. Conf. 1993 with EURO-VHDL*, pp. 454–459, 1993.
- [17] Y. Li and W. Chu, "Using FPGA for computer architecture/organization education.," *Proc. 1996 Work. Comput. Archit. Educ.*, pp. 31–35.
- [18] J. Gray, "Hands-on computer architecture: teaching processor and integrated systems design with FPGAs," *Proceedings 2000 Work. Comput. Archit. Educ.*, no. 17, 2000.
- [19] M. Becvar, A. Pluhacek, and J. Danecek, "DOP: a CPU core for teaching basics of computer architecture," *Work. Comput. Archit. Educ.*, no. 4, 2003.

- [20] A. S. Muñoz and J. D. Rodríguez-Morcillo, "Microprocesador RISC sintetizable en FPGA para fines docentes," *VIII Congr. Tecnol. Apl. a la Enseñanza la Electrónica, España*, 2008.
- [21] D. Mandalidis, P. Kenterlis, and J. N. Ellinas, "A Computer Architecture Educational System based on a 32-bit RISC Processor," vol. xx, no. January, 2008.
- [22] V. R. Wadhankar, "A FPGA Implementation of a RISC Processor for Computer Architecture," pp. 24–28, 2012.
- [23] N. L. V. Calazans and F. G. Moraes, "Integrating the teaching of computer organization and architecture with digital hardware design early in undergraduate courses," *IEEE Trans. Educ.*, vol. 44, no. 2, pp. 109–119, 2001.
- [24] V. Angelov and V. Lindenstruth, "The educational processor Sweet-16," *Int. Conf. F. Program. Log. Appl.*, pp. 555–559, 2009.
- [25] J. L. L. and E. P. C. Presa, "A 16-bit Didactic Micro-Programmed Micro-Processor," *Comput. Res. Dev.*, vol. 1, no. IEEE International Conference on., 2011.
- [26] H. Oztekin, F. Temurtas, and A. Gulbag, "BZK.SAU.FPGA10.0: Microprocessor architecture design on reconfigurable hardware as an educational tool," *IEEE Symp. Comput. Informatics*, pp. 385–389, 2011.
- [27] M. C. Pereira, P. V. Viera, A. L. A. Raabe, and C. A. Zeferino, "A basic processor for teaching digital circuits and systems design with FPGA," *SPL 2012 - 8th South. Program. Log. Conf. IEEE*, 2012.
- [28] A. Hernández Zavala, J. Avante Reyes, Q. Duarte Reynoso, and J. D. Valencia Pesqueira, "RISC-Based Architecture for Computer Hardware Introduction," *Int. Conf. Comput. Res. Dev.*, pp. 17–21, 2011.

- [29] Antonio Hernández Zavala, Jorge A. Huerta Ruelas, Arodí R. Carvalho Domínguez, Oscar Camacho Nieto. Design of a General Purpose 8-bit RISC processor for Computer Architecture learning. *Computación y Sistemas*, Vol. 19, No. 2, 2015, Pp. 371-385.
- [30] V. H. G. Ortega, J. C. S. Savedra, S. O. S, and R. H. Tovar, "MICROPROCESADOR DIDACTICO DE ARQUITECTURA RISC IMPLEMENTADO EN UN FPGA," pp. 1–8, 2009.
- [31] L. A. Casillo and I. S. Silva, "A methodology to adapt data path architectures to a MIPS-1 model," *2012 Brazilian Symp. Comput. Syst. Eng.*, pp. 172–177, 2012.
- [32] T. Ferreira Oliveira and I. Saraiva Silva, "CABARE : AN EDUCATIONAL RECONFIGURABLE GENERAL PURPOSE PROCESSOR," *Dep. Informática e Matemática Apl. Univ. Fed. do Rio Gd. do Norte , Natal , Bras.*, pp. 2–5, 2009.
- [33] S. Shao, A. K. Jones, and R. Melhem, "A Compiler-based Communication Analysis Approach for Multiprocessor Systems" *Parallel Distrib. Process. Symp. 2006. IPDPS 2006. 20th Int. IEEE*, 2006.
- [34] L. A. Casillo and I. S. Silva, "A Methodology to Adapt Data Path Architectures to a MIPS-1 Model," *2012 Brazilian Symp. Comput. Syst. Eng.*, pp. 172–177, Nov. 2012.
- [35] D. Rebernak and M. Mernik, "A tool for compiler construction based on aspect-oriented specifications," *Proc. - Int. Comput. Softw. Appl. Conf.*, vol. 2, no. Compsac, pp. 11–16, 2007.
- [36] F. P. Mota, B. Q. Leonardo, and V. S. Rosa, "Design of a synthesizable processor didactic in FPGA."
- [37] E. P. Ferlin and V. . J. Pilla, "Microprocessors: from theory to practice, a didactic experience," *34th Annu. Front. Educ. 2004. FIE 2004.*, pp. 4–7, 2004.

- [38] J. Hernández, C. Sanz, and A. Carpeño, "MPD : MICROPROCESADOR DIDÁCTICO EN VHDL," pp. 1–9.
- [39] G. de Arruda Filho, J. Travassos, J. Rodrigues de Oliveira Neto, and J. P. Cerquinho Cajueiro, "ANEM - A Didactic 16 Bit Microcontroller."
- [40] E. Lecha, "El procesador didáctico microprogramable ILA9200," *Informática Educ. Real. y Futur.* 11, p. 1995, 1995.
- [41] F. R. Schneider, R. E. B. Poli, D. Barden, D. K. Leonel, D. A. Fiorentin, F. M. Trindade, F. S. De Vasconcellos, M. C. D. B. O. Neto, and R. P. Ribas, "Design of a 4-bit Processor for Evaluating of the E / D nMOS Technology from CCS / UNICAMP," pp. 2–3.
- [42] V. C. Filho, L. A. Casillo, S. R. Fernandes, and A. S. Neto, "DESIGN OF AN INTEGRATED ENVIRONMENT FOR A DIDACTIC PROCESSOR," pp. 1–4.
- [43] R. E. B. Poli, F. R. Schneider, D. Barden, D. A. Fiorentin, F. M. Trindade, F. S. De Vasconcellos, and R. P. Ribas, "NEANDER - 8 BITS PROCESSOR DESIGN FLOW," *Inst. Informática – UFRGS*, pp. 1–2.
- [44] K. D. N. Ramos, N. Santiago, L. Casillo, A. Augusto, J. A. N. Oliveira, and I. Saraiva, "PROCESSADOR DIDÁTICO EM FPGA COM INTERFACE OCP."
- [45] K. Nakano and Y. Ito, "Processor, assembler, and compiler design education using an FPGA," *Proc. Int. Conf. Parallel Distrib. Syst. - ICPADS*, vol. 14th IEEE , pp. 723–728, Dec. 2008.
- [46] D. Morandi, M. C. Pereira, A. Luis, A. Raabe, and C. A. Zeferino, "Um Processador Básico para o Ensino de Conceitos de Arquitetura e Organização de Computadores," *Cent. Ciências Univali, Itajaí*, vol. 30, no. 58, 2006.

- [47] H. C. de Freitas and C. A. P. S. Martins, "Didactic Architectures and Simulator for Network Processor Learning," *30th Int. Symp. Comput. Arch. ACM*, 2003.
- [48] J. Wagner and R. Leupers, "C Compiler Design for a Network Processor," *Comput. Des. Integr. Circuits Syst. IEEE*, vol. 20, no. 11, pp. 1302–1308, 2001.
- [49] S. Yuan, W. Su, G. Ni, T. Chi, and S. Kuo, "A Compiler Design Technique for Impulsive VDD Current Minimization," *Electromagn. Compat. IEEE Trans.*, vol. 55, no. 5, pp. 855–866, 2013.
- [50] "AspectG," 2006. [Online]. Available: [www.cis.uab.edu/wuh/ddf/index.html](http://www.cis.uab.edu/wuh/ddf/index.html).
- [51] T. van der S. P. Klint and J. J. Vinju, "Term rewriting meets aspect-oriented programming.," *Tech. Rep. CWI*, 2004.
- [52] H. W. S. W. Co-design, "LISA - Machine Description Language and Generic Machine Model for," 1996.
- [53] K. Andrews, R. Henry, and W. Yamamoto, "Design and implementation of the UW illustrated compiler," *Proc. Sigplan'88 Conf. Program. Lang. Des. Implementation*, pp. 105–114, 1988.
- [54] M. Mernik, M. Lenic, E. Avdic, and V. Žumer, "Multiple attribute grammar inheritance," *Informatica*, vol. 24, no. 3, pp. 319–328, 2000.
- [55] M. Mernik and V. Žumer, "An educational tool for teaching compiler construction," *IEEE Trans. Educ.*, vol. 46, no. 1, pp. 61–68, 2003.
- [56] P. R. Henriques, D. Cruz, M. João, and V. Pereira, "Specifying a language with Aspect-Oriented Approach: AspectLISA," pp. 695–700, 2000.
- [57] A. Wo and M. Lo, "Compiler generation tools for C #," *IEEE Proceedings-Software*, vol. 150, no. 5, pp. 323–327, 2003.

- [58] S. Y. Yuan, W. Su, and H.-P. Ho, "A software technique for EMI optimization.," *Electromagn. Compat. (APEMC), 2012 Asia-Pacific Symp. IEEE*, pp. 58–61, 2012.
- [59] S. Y. Yuan, H. E. Chung, and S. S. Liao, "A microcontroller instruction set simulator for EMI prediction," *IEEE Trans. Electromagn. Compat.*, vol. 51, no. 3 PART 2, pp. 692–699, 2009.
- [60] A. Kaliszan, "Didactic Embedded Platform Atmega128 for Developing Real Time Operating System," no. December, pp. 1–4, 2010.
- [61] R. Giorgi, C. A. Prete, and V. Diotisalvi, "An Educational Environment for Designing and Performance Tuning of Embedded Systems," *Proc. 1998 Work. Comput. Archit. Educ. ACM*, 1998.
- [62] P. V. Vieira, A. Luis, A. Raabe, and C. Aplicada, "Bipide – Ambiente de Desenvolvimento Integrado para a Arquitetura dos Processadores BIP," *Rev. Bras. Informática na Educ.*, vol. 18, no. 01, p. 32, 2010.