Universidad Autónoma de Querétaro
Facultad de Ingeniería
Campus San Juan del Río

Desarrollo de compilador para lenguaje escalera de controladores lógicos programables para aplicaciones industriales

TESIS

Que como parte de los requisitos para obtener el grado de

Maestro en Ciencias en Mecatrónica

Presenta

Jesús Iván Sánchez Gómez

Asesor: Dr. J. Jesús de Santiago Pérez

Co-asesor: Dr. Roque A. Osornio Ríos

San Juan del Río, Qro., junio de 2013

Universidad Autónoma de Querétaro

Facultad de Ingeniería

Maestría en Mecatrónica

Desarrollo de compilador para lenguaje escalera de controladores lógicos programables para aplicaciones industriales

TESIS

Que como parte de los requisitos para obtener el grado de Maestro en Ciencias en Mecatrónica

Presenta

Jesús Iván Sánchez Gómez

Dirigido por:

Dr. J. Jesús de Santiago Pérez

SINODALES

Dr. J. Jesús de Santiago Pérez
Presidente ________ Firma

Dr. Roque A. Osornio Ríos
Secretario ________ Firma

Dr. Luis Morales Velázquez
Vocal ________ Firma

Dr. Miguel Trejo Hernández
Suplente ________ Firma

Dr. Juan Primo Benitez Rangel
Suplente ________ Firma

Dr. Aurelio Domínguez González
Director de la Facultad

Dr. Irineo Torres Pacheco
Director de Investigación y Posgrado

Centro Universitario
Querétaro, Qro.
mayo, 2013
México.

Resumen

La evolución de la informática y los lenguajes de programación han provocado que el desarrollo de software crezca rápidamente. Cada vez se requieren más herramientas sofisticadas para mejorar la producción en la industria y con esto satisfacer las necesidades que los clientes demanden. Dichas herramientas en general son sistemas totalmente integrales, contando con sus componentes tanto en Hardware como en Software, lo que permite implementar procesos más completos. Por otro lado integrar desarrollos basados en FPGA (Field Programmable Gate Array) a éstos sistemas los mejora considerablemente debido a las ventajas que ofrecen los FPGA, como su reprogramabilidad, reducción de costes de desarrollo, paralelismo, entre otros. Por ésta razón, el presente trabajo muestra el desarrollo de un compilador para un controlador lógico programable (PLC) basado en FPGA, con lo que se hará más fácil y rápida la programación de dicho PLC. Por otro lado también se muestra el desarrollo de una interfaz gráfica que ayudará a una mejor interacción con el usuario, en la que se podrán dibujar los diagramas escalera de algún proceso que se requiera realizar. El desarrollo tanto del compilador como la interfaz gráfica se basa totalmente en una programación estructurada y modular, ésto ofrece diversas ventajas comparados con los software comerciales como que sean actualizables, escalables y portables a futuro, con lo que se genera cierta independencia tecnológica. Para probar la funcionalidad del sistema se realizó un proceso en un invernadero a escala, desde el diseño del diagrama escalera hasta la programación del PLC para comprobar que efectivamente, el proceso se lleve a cabo en el invernadero a escala. Esta investigación pretende ofrecer un sistema integral que satisfaga varios aspectos en la automatización de procesos industriales, debido a la complementación de Software para un controlador basado en FPGA existente (Hardware).

(**Palabras clave:** FPGA, PLC, Lenguaje Escalera, Compilador, Interfaz Gráfica.)

Abstract

The evolution of computing and programming languages have caused software development to grow rapidly. The need of more sophisticated tools has recently increased in order to improve production in industry and meet customer demand . These tools are generally fully integrated systems, with both Hardware and Software components, which allow the implementation of more comprehensive processes. Furthermore, integrated FPGA based developments (Field Programmable Gate Array) to these systems have greatly improved, due to the advantages of the FPGA, for example, reprogrammability, reduced development costs, parallelism. For these reasons, the present work shows the development of a compiler for a programmable logic controller (PLC) based on an FPGA, which will make the programming of the PLC faster and easier. In addition, it also shows the development of a GUI that will improve usability, This improvement enables the drawing of ladder diagrams of some required performance processes. The development of both the compiler and the GUI is entirely based on structured and modular programming, offers several advantages compared to a commercial software since it is upgradeable, scalable and portable, thereby, generating some future technological independence . To test the functionality of the system, a process was performed in a greenhouse to scale, from the design of the ladder diagram to the programming of the PLC to verify that indeed, the process is performed in a greenhouse to scale . This research aims to provide a comprehensive system that fulfills several aspects in the automation of industrial processes, due to the compatibility of Software for an existing FPGA-based controller (Hardware).

(**Keywords:** FPGA, PLC, Ladder Language, Compiler, User Interface.)

Dedicatorias

A mis padres y mis hermanas
que siempre me han brindado su amor
y apoyo incondicional en todo momento.

Agradecimientos

Primeramente quisiera agradecer a mi familia, mis padres que siempre estuvieron ahí cuando los necesitaba y me apoyaron en todas mis decisiones; a mis hermanas que a pesar de todo me soportaron y de igual manera siempre se preocupaban por mi y me apoyaban en todo momento. A mi asesor el Dr. J. Jesús de Santiago Pérez por guiarme en la dirección correcta en el desarrollo de ésta investigación. Al Dr. Luis Morales Velázquez por el apoyo brindado y por los conocimientos compartidos para seguir avanzando. A todos mis compañeros, profesores y personal de la Universidad Autónoma de Querétaro con los que conviví durante éstos dos años. Al Consejo Nacional de Ciencia y Tecnología por la beca otorgada para la realización de los estudios de posgrado en la Universidad Autónoma de Querétaro.

Índice general

Resumen I

Abstract II

Dedicatorias III

Agradecimientos IV

Índice general V

Índice de figuras IX

Índice de tablas XI

1. Introducción 1

1.1. Antecedentes 2

1.2. Hipótesis y Objetivos 4

1.2.1. Hipótesis 4

1.2.2. Objetivo General 4

1.2.3. Objetivos Particulares 4

1.3. Descripción del Problema 5

1.4. Justificación 6

1.5. Planteamiento General 7

2. Revisión de la literatura 9

2.1. Estado del arte 9

2.2. PLC 9

2.3. Procesador PLC basado en FPGA 10

2.3.1. Operaciones Básicas del PLC 14

2.3.2. Mapa de Direcciones de Memoria 18

2.4. Lenguaje Escalera y Lenguaje Booleano 20

2.5. Notación Polaca Inversa 21

2.6. Compilador 22

2.6.1. Estructura Clásica de un compilador 22

2.7. GNU Flex y Bison 24

2.7.1. Generador de Escáneres 24

2.7.2. Generador de Analizadores Sintácticos 26

2.8. Notación BNF 27

2.9. wxWidgets 28

2.9.1. wxFormBuilder 29

2.9.2. wxShapeFramework 29

2.10. Linux 30
2.10.1. Ubuntu 31
2.10.2. Consola de Linux (*Shell*): Comandos Básicos 31
2.11. Licencia GNU GPL 33
2.12. Protocolo RS-232 33

3. Metodología **35**

3.1. Instalación de Herramientas para la Programación del Compilador 36
3.1.1. Flex 36
3.1.2. Bison 37
3.1.3. Compilador GCC 37
3.2. Instalación de Herramientas para la Programación de la Interfaz Gráfica 38
3.2.1. wxWidgets 38
3.2.2. wxShapeFramework 39
3.3. Estructura General del Código Fuente del Compilador 40
3.3.1. Desarrollo del Analizador Léxico 40
3.3.2. Desarrollo del Analizador Semántico y Sintáctico 41
3.4. Desarrollo del Esqueleto de la Interfaz Gráfica 43
3.5. Integración de la Funcionalidad de la Interfaz Gráfica 45

4. Resultados **46**

4.1. Compilador como herramienta independiente 46

4.2. Interfaz Gráfica 51

4.3. Compilador Integrado a la Interfaz Gráfica 53

5. Conclusiones y Prospectivas **58**

Referencias **60**

A. Código Fuente en la Nube **63**

A.1. Git: Control de Versiones 63

A.1.1. Instalación de Git 64

A.2. Repositorio del Código fuente 64

B. Código del Compilador **66**

C. Artículo **85**

Índice de figuras

1.1. Descripción del Problema 5

1.2. Desarrollo Hardware/Software 6

1.3. Diagrama a Bloques General del Sistema 7

2.1. Procesador PLC basado en FPGA 11

2.2. Bloques del Microprocessor 12

2.3. Comunicación Externa con el Microprocesador 13

2.4. Mapa de Direcciones de Memoria 18

2.5. Lenguaje Escalera 20

2.6. Estructura Clásica de un Compilador 23

2.7. Estructura de un Archivo Flex 24

2.8. Diseñador de Interfaces wxFormBuilder 30

2.9. Consola de Linux 31

3.1. Metodología 35

3.2. Lenguaje Propuesto para el Código Fuente 40

3.3. Esqueleto de la Interfaz Gráfica 44

3.4. Generación de Clase Heredada 45

4.1. Llamada del Compilador desde la Terminal en Linux 47

4.2. Diagrama Escalera de Prueba con Motores 47

4.3. Código Fuente de la Prueba con Motores 48

4.4. Análisis Entregado por el Compilador 48

4.5. Simulación de Encendido Secuencial de Motores 51

4.6. Barra de Menús 51

4.7. Barras de Herramientas 53

4.8. Invernadero a Escala 53

4.9. Diseño del Diagrama Escalera 55

4.10. Generación de las Ecuaciones Booleanas 56

4.11. Compilación desde la Interfaz 56

4.12. Programación del PLC desde la Interfaz 57

Índice de tablas

2.1. Instrucciones para Programar el Procesador PLC 18

2.2. Direcciones Base para los Temporizadores 19

2.3. Direcciones Base para los Registros de Entrada 19

2.4. Direcciones Base para los Registros de Salida 19

2.5. Ejemplo de Código Flex 25

2.6. Metasímbolos de la notación BNF 27

3.1. Estructura General del Lenguaje Propuesto 42

3.2. Expresiones del Lenguaje Propuesto 43

3.3. Ejemplo de las Señales Generadas 45

4.1. Código Generado por el Compilador 49

4.2. Fragmento del Archivo de Cabecera con el Mapeo de las Direcciones 50

4.3. Entradas 54

4.4. Salidas 54

Capítulo 1

Introducción

En informática, un lenguaje es una notación formal para describir algoritmos o funciones que serán ejecutadas por un ordenador. Un aspecto muy importante para un lenguaje es la gramática, uno de los objetivos iniciales fue desarrollar una gramática capaz de definir el lenguaje natural inglés. En la actualidad, ese objetivo no se ha llegado a lograr totalmente, por esta razón se enfoco más al desarrollo de lenguajes de programación, dado que, al tratarse de lenguajes mucho más acotados, su descripción gramatical si ha dado resultados (Sánchez et al.,1989). La escritura de compiladores comprende los lenguajes de programación, la arquitectura de los computadores, la teoría de los lenguajes, los algoritmos y la ingeniería del software. Por fortuna, con algunas técnicas básicas de escritura de compiladores se pueden construir traductores para una gran variedad de lenguajes y máquinas (Aho et al., 1998). Con la evolución de la informática y los lenguajes de programación, el desarrollo de software creció rápidamente incluyendo, como se mencionó anteriormente la arquitectura de los sistemas. La mayoría del software y hardware de base comercial maneja una arquitectura cerrada, lo cual, hace poco accesibles los desarrollos de investigación, ya que no se pueden modificar para otro tipo de aplicaciones para los que fueron diseñados originalmente. Además de que tienen un gran costo económico desde que son adquiridos, instalados y se les da mantenimiento.

1.1. Antecedentes

La industria del software se presenta como la de más alto crecimiento y de mayor significación de las SE-I (Sector Electrónico Informático) en el ámbito internacional. Si bien su desarrollo se ha dado en estos últimos años con mayor dinamismo en los países industrializados, particularmente en Estados Unidos, resulta importante destacar que otros países de industrialización tardía o de nueva industrialización, tales como la India, Irlanda e Israel, también han tenido un papel destacado en la producción y exportación de software, pues han logrado buenos niveles de crecimiento e inserción en los mercados internacionales. Asimismo, a partir de la división global de trabajo durante los años noventa, México y otros países latinoamericanos (Brasil, Argentina, Uruguay y Costa Rica) también han intentado fortalecer la producción de software, ya sea para consumo interno o para exportación, aunque de una manera restringida y desigual (Mochi, 2006).

En la Universidad Autónoma de Querétaro existen diversos trabajo referentes al desarrollo de software para procesos industriales. Ugalde (2011) desarrollo un software para un controlador de movimiento basado en FPGA en el cual creó un software capaz de cargar códigos G y DMC que fueron compilados por dicho software para generar curvas NURBS que pudiera interpretar el controlador de movimiento, dicho software fue desarrollado bajo una plataforma linux. Trejo (2006) desarrolló software para realizar el monitoreo y control de los parámetros de maquinado automáticamente en máquinas CNC, el programa fue diseñado en un entorno de desarrollo Visual C++. Se desarrollo un software para la simulación de los movimientos de un manipulador robótico, el cual además de simular dicho movimiento generaba códigos para ejecutar el movimiento en tiempo real (Martínez, 2011). Ugalde (2009) desarrollo una unidad para la generación de trayectorias polinomiales a partir de código G aplicados a sistemas de posición FPGA, éste software fue desarrollado en C bajo un entorno Linux donde fue creado un intérprete de códigos G. Femat (2004) desarrollo e implementó un software para CNC que realiza la compilación de código fuente a código objeto-herramienta por medio de un editor y código G bajo un estándar ISO, todo esto bajo Windows. Sánchez (2011) desarrolló una aplicación para el control y monitoreo de un robot PUMA bajo un entorno Linux, dicha aplicación cuenta con una interfaz gráfica que permite una mejor interacción

por parte del usuario.

De igual manera existen trabajos en la Universidad referentes a los PLC's (controladores lógicos programables), Torres (2004) describió una manera de automatización de una planta de cromado por medio de un PLC de base comercial, explicó a detalle los diagramas de escalera y las variables que se presentan al momento de la instrumentación. Hernández (2004) recopiló información acerca de diferentes tipos de PLC, sus características, configuración, instalación y programación mediante diagramas escalera, todo esto para llevar a cabo el control de procesos secuenciales. Mejía (2004) realizó la descripción de la programación de una máquina de soldar mediante diagramas escalera programados en un PLC de base comercial. Cruz (2008) diseñó y documentó los diagramas de escalera que se necesitan para automatizar una máquina de inyección de plástico con un PLC, además de hacer el análisis de las distintas necesidades de control que se requieren en la máquina. Por otro lado existen trabajos que no manejan PLC's de base comercial como Hernández (2008) quién diseñó y construyó una interfaz para las aplicaciones de control en una máquina de inyección de plástico mediante un lenguaje de descripción de hardware VHDL y lo implementó en un FPGA. Principalmente, el procesador PLC que será ocupado en ésta investigación fue diseñado por Muñoz (2009), éste fue realizado por medio de un lenguaje de descripción de hardware para ser implementado en la automatización de maquinaria CNC, inicialmente.

Como se puede observar en los antecedentes el desarrollo de software en México ha ido tomando gran auge. Esto se debe principalmente a los costos que ofrecen los software comerciales además que, la ser de arquitectura cerrada, no pueden ser modificables a voluntad. Por otro lado, localmente se muestra que en la Universidad Autónoma de Querétaro cada vez existen más investigaciones de este tipo que contribuyen con el desarrollo de herramientas bastante completas para la automatización de procesos industriales. De lo anterior surge la necesidad de continuar con dicha línea de investigación y ofrecer una herramienta más para la automatización de procesos. Además de que el uso de herramientas de carácter libre permiten abaratar los costos de desarrollo y mantenimiento.

1.2. Hipótesis y Objetivos

1.2.1. Hipótesis

Es posible tener un sistema completo de PLC basado en FPGA, si dicho sistema cuenta con un software de control y que a su vez dicho software maneje un traductor entre las instrucciones que el usuario programa y las instrucciones que el sistema de PLC interpretará.

1.2.2. Objetivo General

Desarrollar un compilador que sea capaz de traducir lenguaje escalera a instrucciones que interprete un controlador lógico programable basado en FPGA para implementarse en aplicaciones industriales.

1.2.3. Objetivos Particulares

- Definir el tipo de elementos del lenguaje escalera que serán traducidos por el compilador para su posterior interpretación por el controlador lógico programable.
- Programar en lenguaje C/C++ el módulo encargado del análisis del programa fuente, el cual es la primer fase del compilador y cuyos módulos secundarios son:
    - Análisis léxico
    - Análisis sintáctico
- Programar en lenguaje C/C++ el módulo de síntesis, encargado de la generación del programa objeto que será ejecutado por el controlador lógico programable.
- Desarrollar una interfaz gráfica para la interacción del usuario con el controlador lógico programable basado en FPGA.

- Diseñar la aplicación tomando en cuenta su portabilidad para permitir su ejecución en diversas plataformas (Windows, Linux), utilizando herramientas GNU.
- Realizar pruebas de control en un invernadero a escala para probar la funcionalidad del sistema.

1.3. Descripción del Problema

Actualmente para la programación del procesador PLC se tiene que hacer generando el programa de manera manual y escribir directamente el código en VHDL para así poder realizar las pruebas en una memoria ROM (Fig. 1.1). Por lo que es evidente que se requiere de un software de compilación que facilite la manera en que se genera el código final que será cargado en la memoria del procesador PLC. La presente propuesta presenta el desarrollo de un software de compilación para el PLC basado en FPGA, dicha aplicación contará con una interfaz gráfica para que su uso sea más sencillo y a fin de reducir los costos de creación y mantenimiento será creada en entornos de desarrollo GNU.

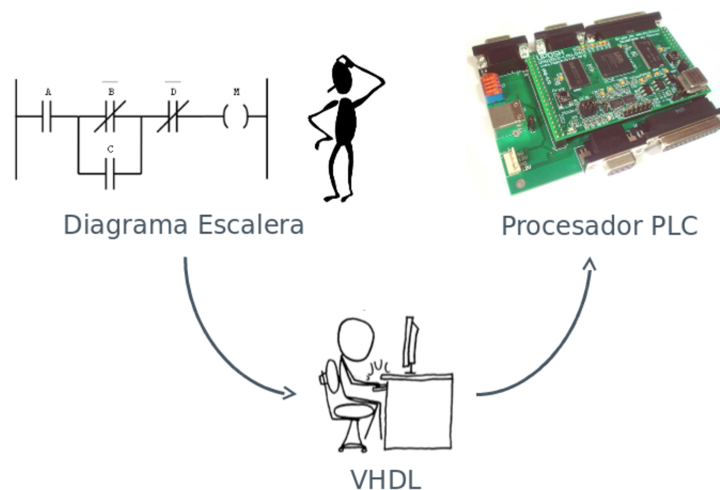

Figura 1.1: Descripción del Problema

1.4. Justificación

La mayoría de las máquinas herramienta actuales requieren que la interacción y el control de las mismas por parte del usuario sea de una manera más rápida, simple y sin tener la necesidad de aprender y memorizar grandes cantidades de instrucciones. Un compilador realiza la tarea de traducir instrucciones simples a instrucciones que entienda la máquina, por lo que el desarrollo del mismo contribuirá con lo requerido actualmente para tener un sistema más automatizado.

Existen algunas otras ventajas a la hora de desarrollo de software propio, sí este se desarrolla con herramientas libres puede significar una gran reducción de costos y además da una mayor portabilidad a las aplicaciones. Al ser aplicaciones de código abierto ofrecen la posibilidad de utilizarse en procesos muy específicos o de igual manera permiten integrar diferentes módulos para otro tipo de tareas según se requiera, por lo que hace que el software sea escalable.

Una de las finalidades de esta investigación es la de disponer de una herramienta más para el diseño de diferentes tipos de sistemas automatizados. Por otro lado también se pretende que en la Universidad Autónoma de Querétaro se siga con el desarrollo de software que sirva de complemento al hardware que actualmente ahí se desarrolla (Fig. 1.2).

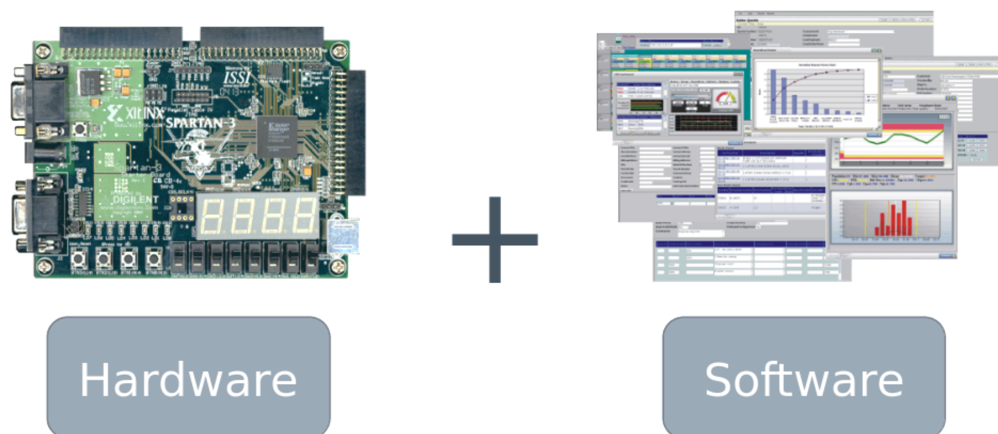

Figura 1.2: Desarrollo Hardware/Software

1.5. Planteamiento General

El desarrollo de la siguiente investigación requiere del uso de un controlador lógico programable basado en FPGA, el diseño de dicho controlador esta descrito por Muñoz (2009). Principalmente este trabajo se enfoca en el desarrollo de un compilador que sea capaz de hacer más rápida y sencilla la programación del controlador lógico programable. De igual manera se integra una interfaz gráfica que haga uso de dicho compilador y la interacción con el usuario se haga más amigable, la Fig. 1.3 muestra un diagrama a bloques general de la interfaz y su interacción con el usuario y el PLC.

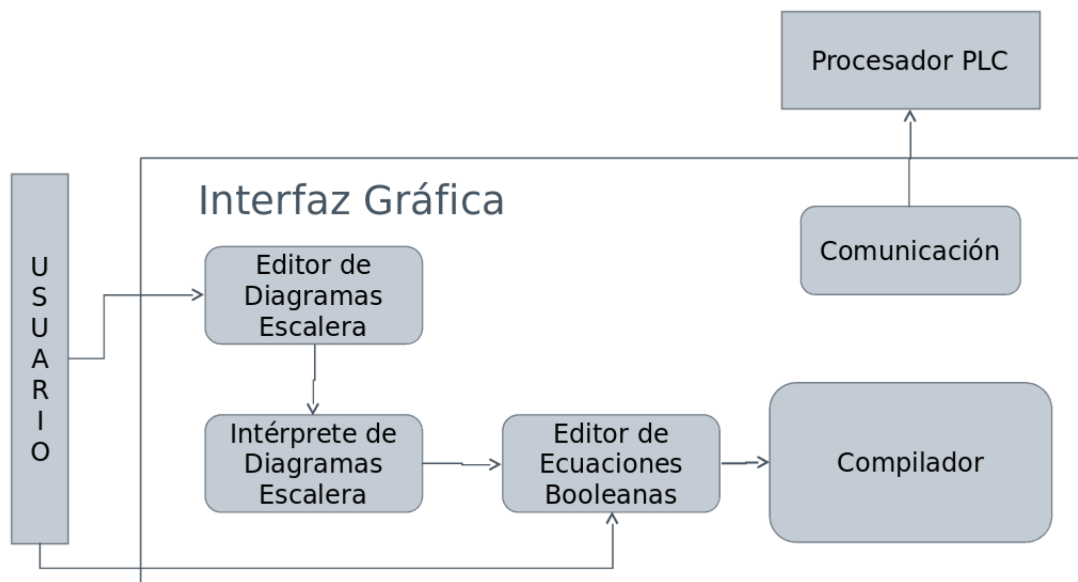

Figura 1.3: Diagrama a Bloques General del Sistema

Editor de Diagramas Escalera. Éste módulo esta centrado principalmente en el diseño y elaboración de los diagramas escalera del proceso que se requiera realizar.

Intérprete de Diagramas Escalera. En éste módulo se realiza la interpretación de los diagramas previamente dibujados y los transforma a su representación en ecuaciones booleanas.

Editor de Ecuaciones Booleanas. Las ecuaciones booleanas pueden ser escritas directamente sin realizar ninguna clase de diagrama en éste módulo.

Compilador. Es el encargado de realizar la traducción de las ecuaciones booleanas para su posterior programación en el PLC.

Comunicación. Aquí se realiza la comunicación con el PLC para que pueda ser programado el proceso que se requiera en el mismo. Dicha comunicación es realizada mediante el protocolo RS-232.

Capítulo 2

Revisión de la literatura

2.1. Estado del arte

El desarrollo de cualquier tipo de software requiere llevar a cabo numerosas tareas que se dividen en etapas por lo que en primera instancia es importante definir claramente los conceptos y herramientas que se utilizarán para el desarrollo del mismo. A continuación se definirán estos conceptos y herramientas que serán utilizadas para el diseño y desarrollo del compilador.

2.2. PLC

El PLC (Controlador Lógico Programable) es un dispositivo que controla una máquina o proceso y puede considerarse simplemente como una caja de control con dos filas de terminales: una para salida y la otra para entrada. Los terminales de salida proporcionan comandos para conectar dispositivos como válvulas solenoides, motores, lámparas indicadoras, indicadores acústicos y otros dispositivos de salida. Los terminales de entrada reciben señales de realimentación (feedback) para conexión de dispositivos como interruptores de láminas, disyuntores de seguridad, sensores de proximidad, sensores fotoeléctricos, pulsadores e interruptores manuales, y otros dispositivos de entrada.

De igual manera un PLC permite mayor libertad de programación. Puede controlar secuencias en paralelo y responder a informaciones no secuenciales para la toma de decisiones. Un PLC puede controlar los bucles del proceso e incluso controlar dos máquinas o más al mismo tiempo, aún cuando éstas funcionan de forma independiente. El método de programación consiste en construir un diagrama de contactos o de instrucciones para la aplicación de control, y a partir de ahí introducir una lista de expresiones lógicas. Esto no es tan directo y sencillo como la programación de un secuenciador, pero no cuesta mucho familiarizarse con el proceso y pronto darse cuenta de la naturaleza flexible de dicha lógica combinacional (Hyde et al., 1997).

Es conveniente usar un PLC cuando el proceso está sujeto a cambios, y el costo del desarrollo y mantenimiento del sistema de control es alto comparado con el de la automatización con un PLC, cuando se cuenta con espacio reducido para implementar otros sistemas de control, o cuando se tiene procesos secuenciales, entre otras situaciones. El PLC puede recibir entradas analógicas y digitales, las cuales procesa de acuerdo a una programación que introduce el usuario para obtener una respuesta de control deseada. Para programar la función del PLC existen diferentes lenguajes que pueden ser utilizados, desde diagramas de escalera hasta lenguajes de programación de alto nivel (Muñoz, 2009).

2.3. Procesador PLC basado en FPGA

Como define Muñoz en 2009, el desarrollo de la arquitectura del procesador para el PLC integrado en la tarjeta PLCUAQ816 fue desarrollado mediante un lenguaje de descripción de hardware VHDL, VHDL es el acrónimo que representa la combinación de VHSIC (Very High Speed Integrated Circuit) y HDL (Hardware Description Language), es decir, lenguaje de descripción de hardware de circuitos integrados de muy alta velocidad. La plataforma FPGA ofrece muchas ventajas como su flexibilidad, ya que es reprogramable. En la actualidad los FPGA permiten ser reconfigurados parcialmente, con esta característica permite que un sistema no se quede obsoleto en poco tiempo.

Utilizando el programa para síntesis de lenguaje de descripción de hardware Xilinx ISE 8.1 se

realizó la síntesis del código VHDL generado para la arquitectura del procesador. Al utilizarse el lenguaje estándar industrial VHDL se asegura que el proyecto puede implementarse en cualquier FPGA con la capacidad de compuertas necesarias haciendo mínimos cambios. El procesador PLC basado en FPGA se muestra en la Figura 2.1.

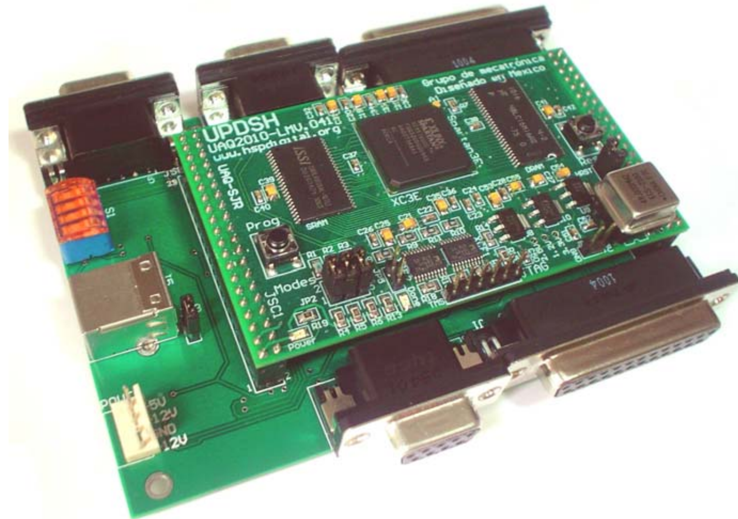

Figura 2.1: Procesador PLC basado en FPGA

La tarjeta PLCUAQ816 cuenta con las siguientes características:

- Capa física
    - Controlador lógico programable
    - 8 salidas analógicas de $\pm$10V
    - 8 entradas analógicas de $\pm$10V
    - 16 salidas digitales TTL
    - 16 entradas digitales TTL
    - 3 DIP switch
    - Puerto RS232
    - Puerto USB
    - Alimentaciones requeridas 5V, 12V y -12V

- Memoria serial de 32KB
- Memoria estática de 512KB
- Memoria dinámica de 4Mb
- FPGA Spartan 3E XC3S200 de 200,000 compuertas
- Configuración vía JTAG o de memoria flash
- Oscilador de 50 MHz.

- Capa Lógica
    - Microprocesador PLC embebido
    - Manejadores de E/S digitales
    - Controladores de ADC y DAC
    - Temporizadores programables
    - Reloj de tiempo real
    - Controladores de memorias
    - Controladores de puertos RS232 y USB

El esquema general del microprocesador diseñado se compone de los bloques mostrados en la Fig. 2.2.

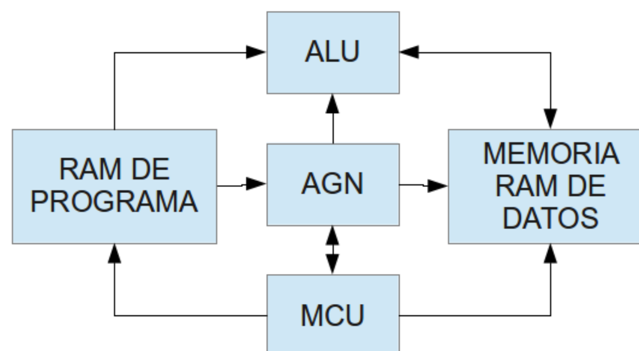

Figura 2.2: Bloques del Microprocessor

RAM de Programa. En esta se almacena el programa, debido a que el microprocesador es de 8 bits, la memoria RAM debe almacenar datos en 8 bits, para la dirección se utilizan 10 bits y se tienen 1024 localidades.

Unidad Aritmética Lógica (ALU). Realiza las operaciones lógicas y aritméticas necesarias para el análisis de datos en la interpretación de señales y datos para ejecutar la rutina de un diagrama escalera.

Unidad de Control de Microprograma (MCU). Es la unidad encargada de decodificar las instrucciones que se encuentran codificadas en el programa. Habilita o deshabilita las memorias, el generador de direcciones (AGN), y las entradas y salidas de los periféricos.

Memoria RAM de Datos (Datos Dual RAM). Esta memoria almacena los datos que se van adquiriendo en la ejecución del programa, la habilitación de escritura o lectura son manejadas por la unidad de control de microprograma.

Generador de Direcciones (AGN). Es la unidad encargada de generar las direcciones para manejar las memorias, tanto la de programa como la de datos.

Por otro lado el microprocesador cuenta también con módulos para realizar la comunicación con el exterior por medio de periféricos mostrados en la Fig. 2.3.

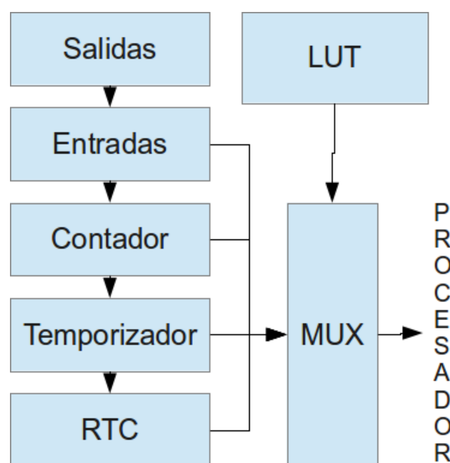

Figura 2.3: Comunicación Externa con el Microprocesador

Salidas. Es la encargada de la comunicación de los periféricos de salida del microprocesador

con las tarjetas de interfaz para el control del proceso.

Entradas. Introduce las señales del proceso hacia el microprocesador. Se comunica con tarjetas de interfaz de entradas, encargadas de monitorear el proceso y sus variables.

Contadores. Se encarga de contar eventos, debe poder ser programable. Es controlado por el microprocesador.

Temporizadores. Es un contador de tiempo real, necesario para procesos que necesiten ejecutarse en un tiempo determinado, de igual manera es programable y controlado por el microprocesador.

Reloj en Tiempo Real (RTC). Se trata de un reloj de tiempo real, el cuál permite almacenar la hora y la fecha actuales, para utilizar estos valores durante el proceso en caso de que se necesite realizar un evento en cierta fecha específica a cierta hora del día.

LUT. Para manejar los periféricos, los contadores y los temporizadores se utiliza una tabla de consulta que es controlada por la unidad de control de microprograma y por el generador de direcciones.

2.3.1. Operaciones Básicas del PLC

Las operaciones básicas que realiza el procesador PLC para poder ejecutar un diagrama escalera son las siguientes:

AND. Operación lógica *AND* entre dos bits.

OR. Operación lógica *OR* entre dos bits.

NOT. Operación *NOT* a un bit.

MOV8. Desplazamiento de datos (bloques de 8 bits) en la memoria de datos.

MOV16. Desplazamiento de datos (16 bits, 2 bloques consecutivos de 8 bits) en la memoria de datos, debe ser condicionada a un valor lógico ‘1’.

ALD. Carga en un registro, que sirve de acumulador, un dato proveniente de la memoria del programa, operación condicionada a un valor lógico '1'de algún resultado.

ARG16. Guarda el dato almacenado en el registro del acumulador, en una dirección indicada de la memoria RAM de datos, operación condicionada a un valor lógico '1'.

RGA16. Carga un registro, que sirve de acumulador un dato proveniente de la memoria del programa, operación condicionada a un valor lógico '1'.

CG8. Hace una comparación entre los datos almacenados en la memoria de datos de la dirección fuente y la dirección destino, indica si el dato de la dirección fuente es mayor.

CGE. Compara si el dato almacenado en la memoria RAM de datos en la dirección fuente y la siguiente dirección fuente + 1 (16 bits) es mayor que el almacenado en la dirección destino y la dirección destino + 1.

CL8. Hace una comparación entre los datos almacenados en la memoria de datos de la dirección fuente y la dirección destino, indica si el dato de la dirección destino es mayor.

CLE. Compara si el dato almacenado en la memoria RAM de datos en la dirección destino y destino + 1 (16 bits) es mayor que el almacenado en la dirección fuente y fuente+1.

CE8. Hace una comparación entre los datos almacenados en la memoria de datos de la dirección fuente y la dirección destino, indica si el dato de la dirección destino es igual al de la dirección fuente.

CEQ. Compara si el dato almacenado en la memoria RAM de datos en la dirección destino y destino + 1 (16 bits) es igual que el almacenado en la dirección fuente y fuente + 1.

LBITX. Almacena el bit x (0 - 7) de una dirección de la memoria RAM de datos, para poder realizar operaciones lógicas entre bits.

MBITX. Almacena el bit x (0 - 7) en una palabra almacenada en la memoria RAM de datos.

SBITX. Almacena el bit x (0 - 7) en una palabra únicamente si este bit representa un valor verdadero, de lo contrario el bit mantiene su valor anterior.

CBITX. Limpia un bit x (0 – 7) de una palabra almacenada en una dirección de la memoria RAM de datos.

Las instrucciones que reconoce el procesador se muestran en la Tabla 2.1, existen dos operaciones que se agregan al procesador, la operación NOP que no realiza función alguna, y la operación FIN cuyo propósito es indicar el final del programa y reiniciar el proceso. Cada instrucción va acompañada de un código de operación, el cual debe ser almacenado en la memoria de programa para que el procesador identifique esa operación.

Operación	Ciclos de Reloj	Código de Operación	Parámetros
AND	3	00000001	
OR	3	00000010	
NOT	2	00000011	
MOV8	10	00001000	Fuente, Destino
MOV16	11, 12	00001001	Fuente, Destino
PUSHSET	1	00000100	
ALD	6	00000101	Dato
ARG16	7	00000110	Destino
RGA16	10	00000111	Destino
CG8	16	00001010	Fuente, Destino
CGE	14	00001011	Fuente, Destino
CL8	14	00001100	Fuente, Destino
CLE	14	00001101	Fuente, Destino
CE8	14	00001110	Fuente, Destino
CEQ	14	00001111	Fuente, Destino
LBIT0	7	00010000	Dirección
LBIT1	7	00010001	Dirección
LBIT2	7	00010010	Dirección
LBIT3	7	00010011	Dirección
LBIT4	7	00010100	Dirección

LBIT5	7	00010101	Dirección
LBIT6	7	00010110	Dirección
LBIT7	7	00010111	Dirección
MBIT0	9	00011000	Dirección
MBIT1	9	00011001	Dirección
MBIT2	9	00011010	Dirección
MBIT3	9	00011011	Dirección
MBIT4	9	00011100	Dirección
MBIT5	9	00011101	Dirección
MBIT6	9	00011110	Dirección
MBIT7	9	00011111	Dirección
SBIT0	9, 10	00100000	Dirección
SBIT1	9, 10	00100001	Dirección
SBIT2	9, 10	00100010	Dirección
SBIT3	9, 10	00100011	Dirección
SBIT4	9, 10	00100100	Dirección
SBIT5	9, 10	00100101	Dirección
SBIT6	9, 10	00100110	Dirección
SBIT7	9, 10	00100111	Dirección
CBIT0	9, 10	00101000	Dirección
CBIT1	9, 10	00101001	Dirección
CBIT2	9, 10	00101010	Dirección
CBIT3	9, 10	00101011	Dirección
CBIT4	9, 10	00101100	Dirección
CBIT5	9, 10	00101101	Dirección
CBIT6	9, 10	00101110	Dirección
CBIT7	9, 10	00101111	Dirección
NOP	4	00110000	

FIN	1	00110001	

Tabla 2.1: Instrucciones para Programar el Procesador PLC

2.3.2. Mapa de Direcciones de Memoria

La memoria de programa cuenta con 1024 localidades de memoria con un ancho de 8 bits, es necesario destacar que no todas las localidades se encuentran disponibles para almacenar datos libremente. Las primeras 55 localidades se encuentran reservadas para funciones especiales, de la dirección 0x0000 a la dirección 0x0036, el resto de las direcciones se encuentran libres para almacenar en ellas banderas o datos y se pueden acceder a cualquier dirección de memoria (Fig. 2.4).

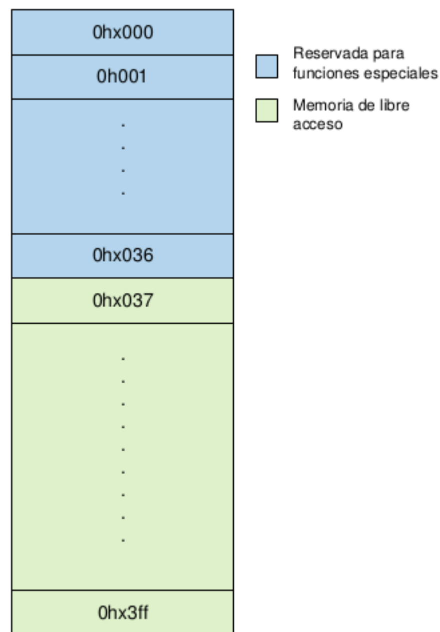

Figura 2.4: Mapa de Direcciones de Memoria

Temporizadores

El procesador PLC tiene implementados cuatro temporizadores, que son los primeros en actualizarse al iniciar el procesador. Los valores que tienen las localidades de memoria reservados para los temporizadores se inicializan con un valor de 0x0000 en todas las localidades. Dichas

localidades de memoria se muestran en la Tabla 2.2.

Número de Temporizador	Dirección Base
Temporizador 0	0x0000
Temporizador 1	0x0005
Temporizador 2	0x000A
Temporizador 3	0x000F

Tabla 2.2: Direcciones Base para los Temporizadores

Entradas

De igual manera se asignan cinco localidades de memoria RAM para almacenar datos, por lo que se cuenta con cinco registros de 8 bits cada uno, es decir, el procesador tiene una capacidad de 40 entradas. Para acceder a los valores de las entradas para poder operar con ellos es necesario hacer referencia a su dirección asignada. En la Tabla 2.3 se muestran las direcciones asignadas a cada registro de entrada.

Número de Entrada	Dirección Base
Registro 0	0x002D
Registro 1	0x002E
Registro 2	0x002F
Registro 3	0x0030
Registro 4	0x0031

Tabla 2.3: Direcciones Base para los Registros de Entrada

Salidas

Nuevamente se asignaron cinco localidades de la Memoria RAM de datos por lo que se cuenta con 40 salidas del sistema, mostradas en la Tabla 2.4.

Número de Salida	Dirección Base
Registro 0	0x0032
Registro 1	0x0033
Registro 2	0x0034
Registro 3	0x0035
Registro 4	0x0036

Tabla 2.4: Direcciones Base para los Registros de Salida

2.4. Lenguaje Escalera y Lenguaje Booleano

Lenguaje escalera

El lenguaje escalera es un lenguaje de programación gráfico utilizado por los autómatas programables debido a que se basa en esquemas de control clásicos. Éstos consisten en una serie de símbolos interconectados por medio de líneas, para indicar el flujo de corriente a través de los distintos dispositivos (Fig. 2.5).

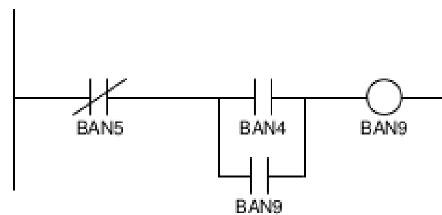

Figura 2.5: Lenguaje Escalera

El procesador de un PLC mantiene y ejecuta el programa de un usuario, para llevar a cabo esta labor, el procesador almacena las condiciones de entrada y salidas. Antes de que un PLC pueda comenzar a controlar un sistema industrial, un usuario debe ingresar las instrucciones codificadas que constituyen el programa. Para organizar y editar los programas, se encuentra conveniente agrupar las instrucciones en pasos de instrucciones, a menudo solo denominados pasos o escalones. La palabra paso se deriva del hecho de que estos grupos de instrucciones recuerdan escalones de una escalera cuando el programa de usuario está representado en formato lógico en escalera. Aun se utilizan este tipo de diagramas pues los eléctricos se encuentran familiarizados con este tipo de simbología, además que es muy fácil representarla.

Con esto se hace notar la importancia que aun tienen los diagramas de escalera dentro de la industria. El software actual para programar PLC's maneja los diagramas de escalera como opción de programación para sus dispositivos. Se utilizan banderas para representar contactos normalmente abiertos o cerrados, y también las banderas juegan el papel que jugaría un solenoide en un sistema electromecánico de control (Muñoz, 2009).

Lenguaje Booleano

El lenguaje de programación booleano esta basado en el álgebra booleana. esta álgebra se utiliza para resolver problemas de lógica. Fue inventada a mediados del siglo XIX por el matemático George Boole. Una de las desventajas de este lenguaje es que es difícil de leer. Por lo tanto, con un diagrama escalera, es fácil escribir un programa con lenguaje booleano. Sin embargo, el procedimiento inverso es complejo; es difícil leer un programa en lenguaje booleano y luego llevarlo al diagrama escalera correspondiente (Wildi, 2007). Tomando en cuenta el diagrama mostrado en la Fig. 2.5 la Ec. 2.1 muestra la representación del mismo en ecuaciones booleanas.

$$\textbf{BAN9} = (\textbf{NOT } BAN5) \textbf{ AND } (BAN4 \textbf{ OR } BAN9) \tag{2.1}$$

2.5. Notación Polaca Inversa

Una organización de pila es muy eficiente para evaluar expresiones aritméticas. El método matemático común de escribir expresiones aritméticas impone dificultades cuando las evalúa una computadora. Las expresiones aritméticas comunes se escriben en *notación interna fija* donde cada operador escrito está entre los operandos. Consideremos la siguiente expresión aritmética simple.

$$A * B + C * D \tag{2.2}$$

El asterisco (que significa multiplicación) está colocado entre dos operandos A y B o C y D. El signo de más está entre los dos productos. Para evaluar esta expresión aritmética es necesario calcular el producto A * B, almacenar este producto mientras se calcula C * D y después sumar los dos productos. En este ejemplo apreciamos que para evaluar expresiones aritméticas en notación interna fija es necesario analizar la expresión en todas sus partes para determinar cuál operación será la siguiente en ejecutarse.

El matemático polaco Lukasiewicz mostró que las expresiones aritméticas pueden representarse en *notación fija previa*. Esta representación, con frecuencia denominada como *notación polaca*, coloca el operador antes de los operandos. La *notación posterior fija*, denominada *notación polaca*

inversa (RPN) coloca el operador después de los operandos. Los siguientes ejemplos muestran las tres representaciones:

A + B	Notación interna fija
+ A B	Notación polaca o previa fija
A B +	Notación polaca inversa o posterior fija

La notación polaca inversa es una forma adecuada para la manipulación de la pila. La expresión mostrada en la Ec. 2.2 se escribe en notación polaca como se muestra en la Ec. 2.3.

$$AB * CD * + \tag{2.3}$$

La expresión se evalúa como sigue: rastrear la expresión de izquierda a derecha. Cuando se encuentra un operador, ejecute la operación de los dos operandos que se encuentran en el lado izquierdo del operador. Quite los dos operandos y el operador y sustitúyalos por el número que se obtiene del resultado de la operación. Continúe analizando la expresión y repita el procedimiento para cada operador encontrado hasta que no haya más operadores (Morris, 1993).

2.6. Compilador

Un compilador puede considerarse como un programa que traduce un lenguaje a otro.

2.6.1. Estructura Clásica de un compilador

La estructura clásica de un compilador es una serie de fases encadenadas, como se muestra en la Figura 2.6.

Analizador Léxico. También denominado *scanner*. Su función consiste básicamente en agrupar los caracteres del texto fuente en grupos de entidad propia denominados ***tokens*** (signos lingüísticos). Ejemplos de tokens son los identificadores, palabras reservadas, separadores, etc. Los tokens reconocidos son la entrada a la siguiente fase, el analizador sintáctico.

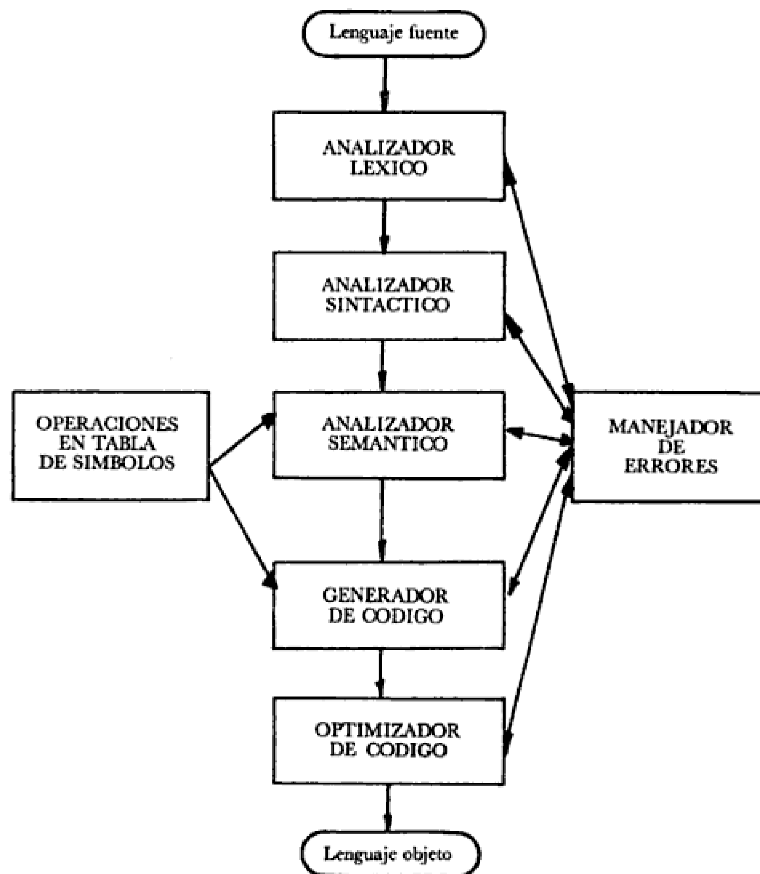

Figura 2.6: Estructura Clásica de un Compilador

Analizador Sintáctico. Se ocupa para analizar la sintaxis de las sentencias (compuestas de tokens), de acuerdo con la descripción sintáctica reflejada en la gramática.

Analizador Semántico. Se ocupa de analizar la semántica de las sentencias, realizando una serie de consultas en unas tablas auxiliares denominadas ***tablas de símbolos***.

Generador de Código. Se ocupa de generar código objeto para una máquina, es decir, donde efectivamente se hace la traducción.

Optimizador de Código. Una fase opcional pero muy usual en los compiladores modernos, se ocupa de optimizar el tamaño y/o velocidad del código generado en la fase anterior.

Además de los procesos descritos, en un compilador hay otras actividades a realizar, de las que destacan dos: el *control de las tablas de símbolos*, que es un conjunto de procedimientos como

introducir un nuevo símbolo, consultar información de un símbolo, modificarla, borrarla, etc., y el *tratamiento de errores* que es el conjunto de rutinas y actividades que tratan la identificación de un error, su posible tratamiento o recuperación y emisión del mensaje correspondiente.

2.7. GNU Flex y Bison

2.7.1. Generador de Escáneres

Flex es una herramienta para generar escáneres. Un escáner, a veces llamado señalizador (tokenizer, en inglés), es un programa que reconoce patrones léxicos en un texto. El programa flex lee archivos de entrada especificados por el usuario o entradas estándar (texto) si ningún nombre de archivo es dado, por una descripción de un escáner a generar.La descripción es en forma de pares de expresiones regulares y código C, denominadas reglas. Flex genera en C un archivo fuente llamado "lex.yy.c", el cual define la función yylex(). El archivo "lex.yy.c"puede ser compilado y vinculado para producir un ejecutable. Cuando el ejecutable se inicia, este analiza el texto de entrada para encontrar expresiones que coincidan con las expresiones regulares de cada regla. Cada que se encuentra una coincidencia, se ejecuta el código C correspondiente.

La estructura de un archivo de Flex se divide en tres secciones separadas por los símbolos %%, como se muestra en la Fig. 2.7.

```
Sección de declaraciones
%%
Sección de reglas
%%
Sección de código en C
```

Figura 2.7: Estructura de un Archivo Flex

La **sección de declaraciones** es donde se definen macros y los archivos de cabecera escritos en C. También es posible escribir código C que será directamente copiado en el archivo fuente generado. Cualquier código escrito debe ir entre los símbolos %{ y %}.

La **sección de reglas** asocia patrones a sentencias en C. Cuando el lexer encuentra un texto en la entrada que es asociable a un patrón dado, ejecuta el código asociado de C. Dichos patrones son expresiones regulares.

La **sección de código C** contiene sentencias en C y funciones que serán copiadas directamente en el archivo fuente generado. Estas sentencias contienen generalmente el código llamado por las reglas en la sección de las reglas.

EL código mostrado en la Tabla 2.5 es un ejemplo de un archivo flex, que reconoce cadenas de números enteros y los imprime en pantalla:

```
/*** Sección de declaraciones ***/

%{

/* Código en C que será copiado */

#include <stdio.h>

%}

/* Esto indica a Flex que lea sólo un fichero de entrada */
%option noyywrap

%%
/*** Sección de reglas ***/

/* [0-9]+ identifica una cadena de uno o más dígitos */

[0-9]+ {
    /* yytext es una cadena que contiene el texto coincidente. */
    printf("Encontrado un entero: %s\n", yytext);
}

. {
    /* Ignora todos los demás caracteres. */
}

%%

/*** Sección de código en C ***/

int main(void){
    /* Ejecuta el ''lexer'', y después termina. */
    yylex();
    return 0;
}
```

Tabla 2.5: Ejemplo de Código Flex

Por ejemplo, si se ingresa la entrada **abc123z.!&*2ghj6**, el programa imprimirá lo siguiente:

- Encontrado un entero: 123
- Encontrado un entero: 2
- Encontrado un entero: 6

Como se observa el programa ignora los demás caracteres y solo muestra los números enteros.

2.7.2. Generador de Analizadores Sintácticos

GNU Bison es un generador de analizadores sintácticos de propósito general (la parte de un compilador que comprueba que la estructura del código fuente se ajusta a la especificación sintáctica del lenguaje) disponible para prácticamente todos los sistemas operativos. Bison convierte la descripción formal de un lenguaje, escrita como una gramática libre de contexto LALR (*Look Ahead Left to Right*), en un programa en C, C++, o Java que realiza análisis sintáctico. Es utilizado para crear analizadores para muchos lenguajes, desde simples calculadoras hasta lenguajes complejos. Para utilizar Bison, es necesaria experiencia con la sintaxis usada para describir gramáticas. Bison se usa normalmente acompañado de Flex, aunque los analizadores léxicos se pueden también obtener de otras formas.

Para que Bison analice un lenguaje, este debe ser descrito por una gramática independiente del contexto. Esto quiere decir que debe especificar uno o más grupos sintácticos y dar reglas para construirlos desde sus partes. Por ejemplo, en el lenguaje C, un tipo de agrupación son las llamadas 'expresiones'. Una regla para hacer una expresión sería, "Una expresión puede estar compuesta de un signo menos y otra expresión". Otra regla sería, "Una expresión puede ser un entero". Como se observa, las reglas son a menudo recursivas, pero debe haber al menos una regla que lleve fuera la recursión.

El sistema formal más común de presentar tales reglas para ser leídas por los humanos es la Forma de Backus-Naur o "BNF", que fue desarrollada para especificar el lenguaje Algol 60. Cualquier gramática expresada en BNF es una gramática independiente del contexto. La entrada de Bison es en esencia una BNF legible por la máquina.

En las reglas gramaticales formales para un lenguaje, cada tipo de unidad sintáctica o agrupación se identifica por un símbolo. Aquellos que son construidos agrupando construcciones más pequeñas de acuerdo a reglas gramaticales se denominan *símbolos no terminales*; aquellos que no pueden subdividirse se denominan *símbolos terminales* o *tipos de tokens*. Denominamos *tokens* a un fragmento de la entrada que corresponde a un solo símbolo terminal, y grupo a un fragmento que corresponde a un solo símbolo no terminal.

2.8. Notación BNF

La notación BNF se usa para especificar las reglas de sintaxis de un lenguaje de programación. BNF son las iniciales de *Backus Naur Form*. John Backus y Peter Naur introdujeron esta notación para describir la sintaxis del lenguaje de programación ALGOL, alrededor de 1960.

Los metasímbolos de la notación se presentan en la siguiente tabla:

::=	Se define como
|	o
<>	Para delimitar categorías, es decir, no terminales
[ ]	Ítem opcional
” ”	Para delimitar terminales de un solo carácter

Tabla 2.6: Metasímbolos de la notación BNF

La notación BNF es un lenguaje para especificar lenguajes de programación de computadores, de ahí que sus símbolos se denominen más precisamente metasímbolos (Hernández et al., 2010).

Ejemplo

La siguiente gramática, en notación BNF, especifica la sintaxis de un lenguaje de programación hipotético, muy limitado por cierto. El lenguaje tiene palabras reservadas en español y utiliza como operador de asignación :=, como lo hace el lenguaje Pascal.

<Digito> ::= ”0” | ”1” | ”2” | ”3” | . . . | ”9”

Cabe destacar que el uso de esta notación en Bison difiere en el símbolo de asignación, ya que ne lugar de utilizar el símbolo := simplemente se utiliza =

2.9. wxWidgets

WxWidgets es un set de herramientas de programador para producir aplicaciones con interfaz gráfica de usuario (GUI), en distintas plataformas. Es un *framework* (marco de trabajo), en el sentido que realiza gran parte del trabajo, y nos entrega, ya programado, el comportamiento que por defecto tiene la mayoría de las aplicaciones. Las librerías de wxWidgets contienen gran cantidad de clases y métodos para que el programador las use y personalice. Una aplicación normalmente muestra ventanas, que contienen controles estándar, que pueden generar y mostrar imágenes y gráficos, y puede responder a entradas desde el teclado, el mouse u otras fuentes, además de comunicarse con otros procesos o programas. WxWidgest le permite al programador reproducir, de manera relativamente sencilla, todos estos comportamientos que son típicos en una aplicación moderna.

Si bien, a menudo, wxWidgets es etiquetado como un set de herramientas para el desarrollo de GUI's, en la práctica es mucho más que eso, teniendo características de utilidad para muchos aspectos del desarrollo de software. WxWidgets entrega soluciones a todas las necesidades de una aplicación, no solo a la interfaz gráfica, sino también manejo de múltiples hilos de ejecución, manejo de archivos y flujos, ajustes de aplicación, comunicación entre procesos, ayuda online, acceso a bases de datos, y mucho más.

Tal y como la mayoría de las frameworks modernas para GUI's, wxWidgets se beneficia del uso de la programación orientada a objetos. Cada ventana es representada como un objeto en C++; estos objetos tienen un comportamiento bien definido, y pueden recibir y responder a eventos. Lo que el usuario ve es la manifestación visual de este sistema interactivo. El trabajo como desarrollador es orquestar el comportamiento colectivo de todos estos objetos, lo que se vuelve más fácil con el comportamiento por defecto que wxWidgets implementa por si mismo.

Por supuesto que no es casualidad que la programación orientada a objetos y las GUI's se integren tan bien, pues sus desarrollos fueron a la par. El lenguaje de programación orientado a objetos Smalltalk, diseñado por Alan Kay entre otros en los '70, fue un hito importante en la historia de la GUI, innovando en la tecnología de la interfaz de usuario y diseño de lenguaje, y aunque wxWidgets ocupa una API y un lenguaje diferente, usa los mismos principios de aquella época (Smart et al., 2006).

2.9.1. wxFormBuilder

wxFormBuilder es un diseñador de interfaces gráficas de código abierto que forma parte del set de herramientas de wxWidgets (Fig. 2.8), el cual permite la creación de aplicaciones multiplataforma. Es una interfaz optimizada y fácil de usar que permite el rápido desarrollo y fácil mantenimiento de software. Esta escrita en C++.

Es una herramienta de desarrollo visual, pero también permite el desarrollo de componentes no visuales. por otro lado puede generar código en C++, Python, PHP y XRC. Dicho código no puede ser modificado directamente en el programa. De igual manera wxFormBuilder utiliza un método de conexiones (*Connect() method*) para el manejo de eventos, o ya sea con una tabla de eventos. Para la mayoría de los controles que existen se pueden crear manejadores de eventos personalizados. Los manejadores de eventos pueden ser cargados desde un archivo XML externo, eliminando la necesidad de reconstruir todo un proyecto (http://www.wxwidgets.org/, 2013).

2.9.2. wxShapeFramework

wxShapeFramework (wxSF) es una biblioteca/marco de trabajo basada en wxWidgets que permite el fácil desarrollo de aplicaciones de manipulación de objetos gráficos (formas) como diversas herramientas CASE (*Computer Aided Software Engineering*, Ingeniería de Software Asistida por Computadora), herramientas de modelado de procesamientos tecnológicos, etc. (http://wxsf.sourceforge.net/, 2013)

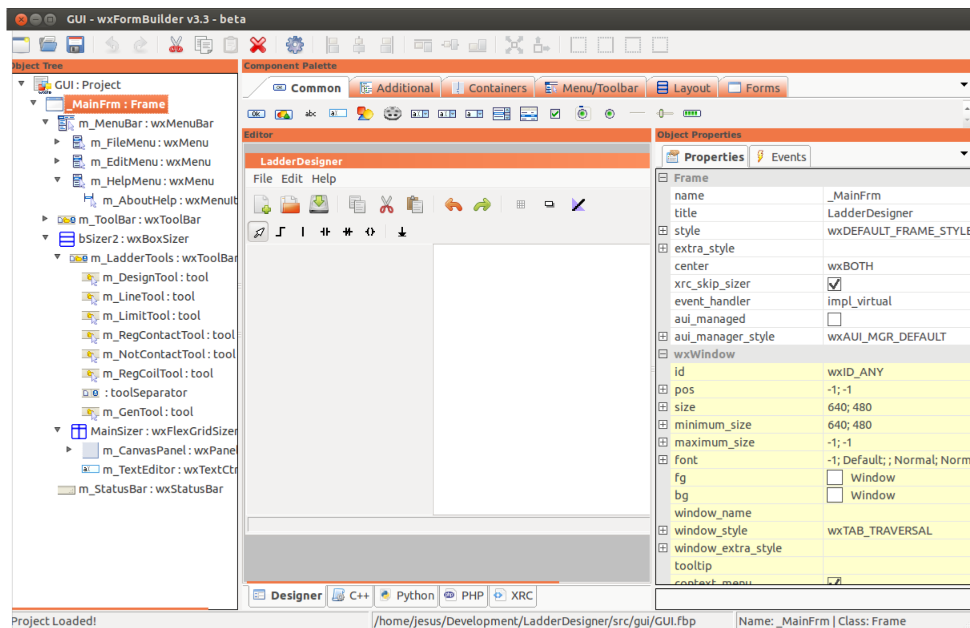

Figura 2.8: Diseñador de Interfaces wxFormBuilder

2.10. Linux

Utilizando el sistema operativo Minix, desarrollado por Andrew S. Tanenbaum con el objetivo de mostrar a sus estudiantes el funcionamiento de un sistema de tipo Unix, en un proyecto escolar sobre el modo protegido de los procesadores intel 386, Linus Torvalds empezó a desarrollar su propio núcleo Unix para añadirle nuevas funcionalidades.

Linux nació, pues, en 1991, gracias a un estudiante de la universidad de Helsinki. El éxito de Linux se basa en una idea ingeniosa de su creador, L. Torvalds: inscribir su proyecto bajo los términos de la licencia GPL y proponer a todos los programadores de internet que le ayudaran.

El impulso de Linux proviene en gran parte del hueco que llenó en términos de núcleo en el proyecto GNU (Pons, 2005).

2.10.1. Ubuntu

Ubuntu es una de las más conocidas distribuciones GNU/Linux que ofrece un sistema operativo predominante enfocado a ordenadores de escritorio aunque también proporciona soporte para servidores.

Sus orígenes están basados en Debian GNU/Linux, Ubuntu concentra su objetivo en la facilidad de uso, la libertad de uso y facilidad de instalación por ser un software libre. Ubuntu está patrocinado por Canonical, una compañía británica, que en vez de vender el software con fines lucrativos se financia por medio de servicios vinculados al sistema operativo y vendiendo soporte técnico. Además al mantenerlo libre y gratuito, la empresa es capaz de aprovechar los desarrolladores de la comunidad en mejorar los componentes de su sistema operativo (Saucedo, 2012).

Históricamente, Ubuntu nació de lo que podría considerarse como una ambigüedad: el multimillonario Mark Shuttleworth funda en 2005 la Ubuntu Foundation, cuyo propósito declarado es contribuir a la popularización del sistema operativo Linux (Chamillar, 2011).

2.10.2. Consola de Linux (*Shell*): Comandos Básicos

La línea de comandos, consola o terminal es un intérprete que espera órdenes escritas por el usuario en el teclado, las interpreta y las entrega al sistema operativo para su ejecución. La respuesta del sistema operativo se muestra al usuario en la misma ventana (Fig. 2.9).

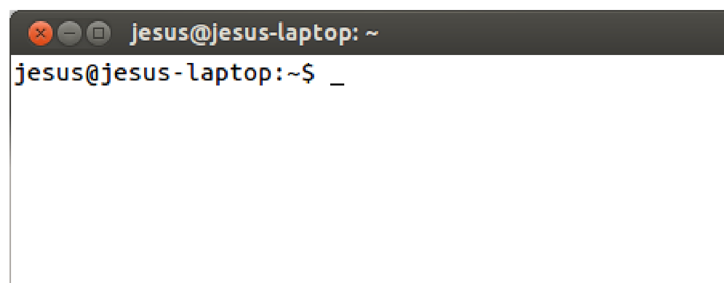

Figura 2.9: Consola de Linux

Acceder a un directorio:

cd directorio

Ejecutar script con extensión ".sh" o ".run":

sh script

Ejecutar comandos con privilegios de superusuario o administrador:

sudo comando

Instalar paquetes en Ubuntu:

sudo apt-get install paquete

Comandos para compilar e instalar paquetes a partir del código fuente, éstos comandos se realizan en el orden en que se muestra a continuación.

Configure, es un script que prepara los archivos para su compilación. Éste informará si hace falta alguna dependencia.

./configure

Make, compila todo el código fuente y lo prepara para su posterior instalación o ejecución.

make

Make install, instala el programa previamente compilado en el sistema. Éste comando requiere permisos de superusuario.

sudo make install

2.11. Licencia GNU GPL

La Licencia Pública General de GNU (GNU GPL, por sus siglas en inglés) es una licencia libre y gratuita con derecho de copia para software. En su preámbulo menciona textualmente:

Las licencias para la mayoría del software y otras obras de índole práctica están diseñadas para privarle de la libertad para distribuir y modificar las obras. Por el contrario, la Licencia Pública General de GNU garantiza la libre distribución y modificación de todas las versiones de un programa, a fin de asegurarle dicha libertad a todos los usuarios. En la Fundación para el Software Libre utilizamos la Licencia Pública General de GNU para la mayoría de nuestro software; también se aplica a cualquier otra obra publicada de esta manera por sus autores. Usted también puede aplicarla a sus programas (www.gnu.org, 2012).

La primera versión de esta licencia fue creada por la *Free Software Foundation* en 1989, principalmente su propósito es declarar que el software es libre y protegerlo de intentos de apropiación que restrinjan la libertad de los usuarios para su uso, modificación o distribución.

Esta licencia es muy común en el software libre o en el desarrollado bajo plataformas libres. Cabe destacar que al referirse a libertad, no precisamente quiere decir que será gratuito. El software creado bajo esta licencia puede ser distribuido sin ningún costo o incluso cobrar por él. De igual manera puede obtener su código fuente, modificarlo o utilizar ciertas partes del mismo para nuevos programas libres, sin problema alguno (Sánchez, 2011).

2.12. Protocolo RS-232

El estándar RS-232 fue redactado por el CCITT (Comité Consultatif International de Télégraphie et Téléphonie) en Europa y la EIA (Electronics Industries Association) en los Estados Unidos para asegurar que hubiera un formato común mundial para comunicaciones en serie entre ordenadores y periféricos. Esta normalización compendia las características de los conectores a utilizar (tamaño, número de patillas, forma, etc.), los niveles de tensión a soportar, y las funciones

de control asignadas a cada patilla. También identifica el protocolo simple de establecimiento de comunicación que determina cuándo está preparado el equipo terminal para enviar y recibir datos.

Los niveles de tensión para la interfaz RS-232 están definidos para extenderse entre -3 V y -15 V para el 1 lógico, y entre +3 V y +15 V para el 0 lógico. Típicamente se establecen en -12 V y +12 V. La interfaz RS-232 utiliza la denominada "Lógica Negativa".

La interfaz esta limitada normalmente a velocidades de 20 kbps para distancias del orden de 15 metros debido a los tiempos de subida de los impulsos en estas longitudes de cable. Sin embargo, son posibles velocidades más altas para distancias mucho más cortas (Bateman, 2003).

Capítulo 3

Metodología

La metodología aplicada en esta investigación es la mostrada en la Fig. 3.1.

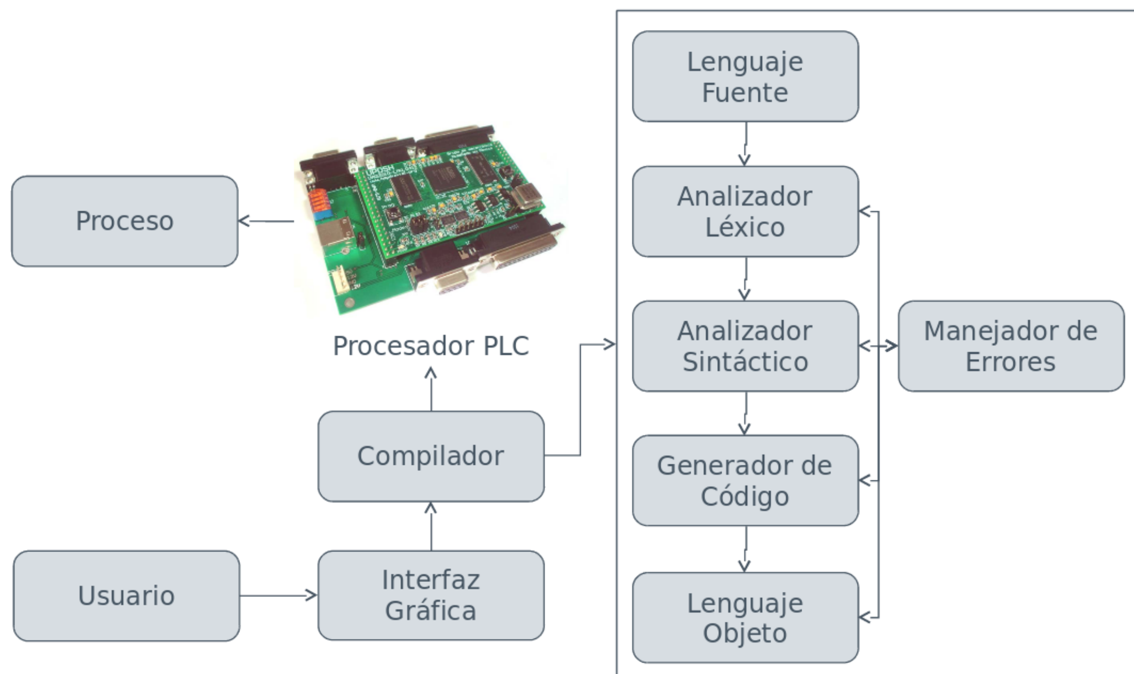

Figura 3.1: Metodología

Inicialmente la programación del controlador lógico programable se hacía de manera manual transcribiendo las ecuaciones de un diagrama en lenguaje escalera a código VHDL que se programaba junto con la estructura del procesador PLC basado en FPGA. Esto hacia que la tarea de

programar algún proceso en el PLC fuera algo tedioso y llevaba bastante tiempo en procesos complejos. Por esta razón se eligió la metodología anterior, en la que claramente se muestra que el procesador PLC interactuará con el usuario por medio de una interfaz gráfica y a su vez a través de un compilador, el cual tiene la estructura básica de un compilador clásico. El usuario solo se tendrá que basarse en un diagrama escalera de algún proceso para programar las ecuaciones booleanas de dicho diagrama y posteriormente hacer uso del compilador integrado a la interfaz gráfica para compilar el diagrama y traducirlo al programa que será cargado en el procesador PLC.

Para llevar acabo la metodología se requieren de diversas herramientas descritas anteriormente, a continuación se describe la instalación de las mismas.

3.1. Instalación de Herramientas para la Programación del Compilador

Para la programación del compilador específicamente se hizo uso de Flex, Bison y el compilador GCC para generar el ejecutable que se encargará de traducir las ecuaciones de lenguaje escalera al lenguaje que será programado en el PLC.

3.1.1. Flex

La herramienta Flex utilizada para generar escáneres que se utilizó, puede instalarse siguiendo los pasos siguientes desde la consola:

Linux

```
$ sudo apt-get install flex
```

La versión de Flex utilizada para el desarrollo de este proyecto es la 2.5.35.

Windows

- Bajar el instalador flex de `http://gnuwin32.sourceforge.net/packages/flex.htm` e instalarlo.

La versión de Flex utilizada para el desarrollo de este proyecto es la 2.5.4.a.

3.1.2. Bison

Al igual que la herramienta anterior, los pasos para la instalación de Bison son similares como se muestra a continuación:

Linux

```
$ sudo apt-get install bison
```

La versión de Bison utilizada para el desarrollo de este proyecto es la 2.5.

Windows

- Bajar el instalador Bison de `http://gnuwin32.sourceforge.net/packages/bison.htm` e instalarlo.

La versión de Bison utilizada para el desarrollo de este proyecto es la 2.4.1.

3.1.3. Compilador GCC

En linux se requiere del software básico de compilación en la cual se incluyen los comandos gcc, g++, make, entre otros. Por otro lado en windows se requiere de la instalación del paquete MinGW. A continuación se mencionan los pasos para instalar cada uno de estos paquetes en su respectiva plataforma:

Linux

```
$ sudo apt-get install build-essential && sudo apt-get install linux-headers-`uname -r`
```

Las versiones utilizadas de estos paquetes son las incluidas en los repositorios de la distribución Linux Ubuntu 12.10.

Windows

- Bajar el instalador MinGW de `http://www.mingw.org/` en la zona de descargas e instalarlo.

La versión de MinGW utilizada es la 4.7.2.

3.2. Instalación de Herramientas para la Programación de la Interfaz Gráfica

3.2.1. wxWidgets

Para realizar la instalación de las librerías wxWidgets en ambas plataformas, tanto en Windows como en Linux se realizan los pasos siguientes.

Linux

```
$ sudo apt-get install libwxgtk2.8-dev libwxgtk2.8-dbg
```

La versión utilizada de wxWidgets para el desarrollo de la interfaz es la 2.8.12.

Windows

Debido a que en Windows no existe paquete de binarios para la instalación de esta versión de la librería se realizó la compilación de las mismas para generar las librerías de enlace dinámico o DLL (por sus siglas en inglés, dynamic-link library).

- Bajar el paquete wxMSW de `http://www.wxwidgets.org/` en la zona de descargas y extraerlo, preferentemente en la raíz ‘c:\’.

- Entrar a la carpeta ‘build\msw’, incluida dentro de la carpeta que se extrajo anteriormente.
- Para compilar las librerías se realiza con el siguiente comando “mingw32-make -f makefile.gcc SHARED=1 UNICODE=1 BUILD=release”

Con esto se tendrán compiladas las librerías de wxWidgets que son la base para la interfaz gráfica.

3.2.2. wxShapeFramework

Al igual que con las librerías de wxWidgets y a razón de que se basa en las mismas, la biblioteca wxSF tiene que compilarse por completo para poder utilizarla, en este caso esto se requiere para ambas plataformas tanto en Linux como en Windows. Para ambos casos se utilizó la versión 1.12.4.

Primeramente se requiere bajar el paquete wxSF_src-1.12.4.tgz de la página oficial del proyecto `http://sourceforge.net/projects/wxsf/`. Posteriormente extraerlo, lo que creará una carpeta llamada ‘wxSF’.

Linux

```
$ cd wxSF/build
$ sh ./create\_build\_files.sh
$ cd ..
$ ./configure
$ make
$ sudo make install
```

Con estos pasos, las librerías quedarán instaladas en un sistema con entorno Linux.

Windows

```
> cd wxSF\build
> create\build\files.bat
> bakefile\gen
> mingw32-make -f makefile.gcc WX\DIR=c:\wxWidgets-2.8.12 WX\UNICODE=1 WX\DEBUG=0 WX\SHARED=1 WX\
    MONOLITHIC=0
```

Cabe destacar que en la opción WX_ DIR se debe ingresar la ruta donde se tengan las librerías compiladas de wxWidgets. Y finalmente con esto se realiza la compilación de la biblioteca wxShapeFramework.

3.3. Estructura General del Código Fuente del Compilador

Al igual que en la mayoría de los lenguajes de programación se propuso un idioma artificial para poder acotar las variables, símbolos y la sintaxis que manejará el código fuente.

3.3.1. Desarrollo del Analizador Léxico

Primeramente se propone un lenguaje para acotar el código fuente, ya que se utilizan solo ecuaciones booleanas y no requiere de condiciones de ningún tipo se definió la sintaxis del lenguaje como se muestra en la Fig. 3.2.

```
init
    /* comandos */
end
```

Figura 3.2: Lenguaje Propuesto para el Código Fuente

Como se puede observar la sintaxis del lenguaje viene acotada por dos instrucciones *init* que nos indica el inicio del programa y *end* que nos indica el final. El usuario deberá ingresar las ecuaciones booleanas entre éstas dos instrucciones, con esto se cuenta con un lenguaje un poco más definido para la descripción de un lenguaje escalera en su versión de ecuaciones booleanas. Por otro lado y de manera más específica cada ecuación representará un escalón de un diagrama y siguiendo con la estructura definida anteriormente del PLC se requieren de diversas variables como son: banderas internas, temporizadores, entradas y salidas.

Tomando en cuenta lo anterior se definen las siguientes palabras reservadas, incluyendo los símbolos de los operadores lógicos, para el desarrollo del analizador léxico.

init Inicio del programa.

end Final del programa.

banX Palabra reservada para el uso de banderas internas, donde la ‘X’ representa un número entero positivo. Puede ser escrita en mayúsculas o minúsculas.

timX Palabra reservada para el uso de temporizadores, donde la ‘X’ representa un número entero positivo. Puede ser escrita en mayúsculas o minúsculas.

inX Palabra reservada para el uso de entradas, donde la ‘X’ representa un número entero positivo. Puede ser escrita en mayúsculas o minúsculas.

outX Palabra reservada para el uso de salidas, donde la ‘X’ representa un número entero positivo. Puede ser escrita en mayúsculas o minúsculas.

Símbolo Asterisco ‘*’ Éste símbolo se utiliza para representar la operación lógica AND.

Símbolo Más ‘+’ Éste símbolo se utiliza para representar la operación lógica OR.

Símbolo Diagonal ‘/’ Éste símbolo se utiliza para representar la operación lógica NOT.

Símbolo Igual ‘=’ Éste símbolo se utiliza para representar la operación de asignación.

Paréntesis ‘(’ ‘)’ Éstos símbolos son para indicar la precedencia de las operaciones.

3.3.2. Desarrollo del Analizador Semántico y Sintáctico

Al tener bien definidas las variables, símbolos y palabras reservadas, lo siguiente es definir las reglas gramaticales por medio de una notación BNF. A continuación se define dichas reglas para el análisis semántico y sintáctico.

Como se observa en la Tabla 3.1, primeramente se define una variable *program* que contiene la estructura general del programa y entre las instrucciones *INIT* y *END* se encuentra la variable *commands*; usando la recursividad ésta variable puede contener una cadena vacía ó a sí misma con la variable *expr* delante de ella. Por último la variable *expr* puede ser del tipo *assign_expr* (expresión de asignación). Ésto nos indica que primeramente el compilador buscará la regla de la

```
program :  /*variables*/
           INIT
              commands
           END
           ;

commands : /* empty */
           | commands expr
           ;

expr :     assign_expr
           ;
```

Tabla 3.1: Estructura General del Lenguaje Propuesto

estructura general del programa y posteriormente buscará las expresiones específicas basadas en los tipos definidos, en éste caso buscará expresiones de asignación. La Tabla 3.2 muestra que una expresión de asignación puede ser del tipo *or_expr* (expresión OR), o de ella misma igualada ('=') a una *or_expr*; de ésta misma manera se hace la siguiente operación *and_expr* (expresión AND), cada una con sus debidos signos *or_expr* ('+'), *and_expr* ('*') y *not_expr* ('/'), siendo ésta última la que contiene la expresión primaria (*primary*) que puede ser del tipo de banderas internas (*BAN*), temporizadores (*TIM*), entradas (*IN*) o salidas (*OUT*), de igual manera puede ser del tipo *expr* entre paréntesis.

Cabe destacar que el orden de éstas reglas es de suma importancia, ya que gracias a éste orden se obtiene la precedencia de las operaciones a realizar. Dicha precedencia puede leerse de abajo hacía arriba, con lo que primeramente el compilador buscaría expresiones primarias o expresiones entre paréntesis, y posteriormente resolvería las operaciones lógicas NOT, AND, OR y por último la asignación. Gracias a ésto se puede convertir dichas operaciones a una notación polaca inversa, que es la requerida para trabajar con el PLC.

Detección de Errores

Los errores se acumulan con cada regla del análisis sintáctico, si se encuentra con alguna regla o palabra reservada que no se haya definido el manejador de errores acumulará una cuenta en una lista, así como también generará una lista de la descripción del tipo de error encontrado, para que al final se imprima en pantalla dicha lista de errores contabilizados. Si no se corrigen todos los errores

```
assign_expr:
    or_expr
    | assign_expr '=' or_expr
    ;

or_expr:
    and_expr
    | or_expr '+' and_expr
    ;

and_expr:
    not_expr
    | and_expr '*' not_expr
    ;

not_expr:
    primary
    | '/' primary

primary:
    BAN
    | TIM
    | IN
    | OUT
    | '(' expr ')'
    ;
```

Tabla 3.2: Expresiones del Lenguaje Propuesto

que se encuentren el compilador no generará el código objeto.

3.4. Desarrollo del Esqueleto de la Interfaz Gráfica

Como se menciono anteriormente la interfaz se hizo con el diseñador de interfaces wxFormBuilder, existen diferentes herramientas que se agregan a la interfaz, desde los botones, menús, barra de herramientas, etc. Todo esto de una rápida y sencilla en la que aún no sabiendo programación el diseñador puede crear su interfaz para que posteriormente el programador le agregue la funcionalidad a la misma. En éste esqueleto se generan también las señales que tiene cada elemento de la interfaz, en la Fig. 3.3 se muestra como se agrega la señal de clic en el botón de guardar (marcado en azul), todo ésto sirve para la programación de las acciones de cada elemento.

Al finalizar de diseñar la interfaz gráfica se deben generar los archivos de C++ de la misma, ésto se logra directamente presionando la tecla F8 o dando clic en el botón *Generate Code*, identificado

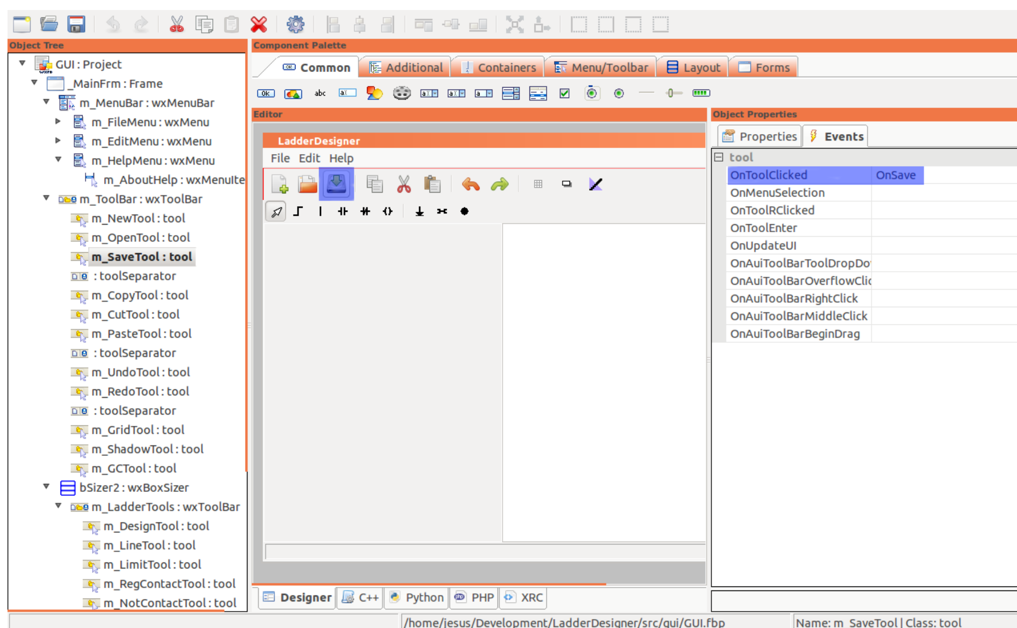

Figura 3.3: Esqueleto de la Interfaz Gráfica

con un símbolo de un engrane, de la barra de herramientas de wxFormBuilder. Con ésto se genera los archivos del esqueleto de la interfaz gráfica, los cuales por ningún motivo deben ser modificados ya que se corre el riesgo de que la interfaz no funcione debidamente o que el código no compile.

Finalmente se genera el archivo con las señales de todos los elementos de la interfaz, siempre y cuando dichas señales se hayan agregado a cada elemento. Para generar los archivos se presiona la tecla F6 o se da clic en el menú *Tools/Generate Inherited Class* de la barra de menús de wx-FormBuilder, ésto muestra el cuadro de dialogo de la Fig. 3.4 en el que se elige la ventana que de la que se generará la clase y se le da un nombre. Lo que hace esto es aprovechar la propiedad de herencia que se tiene en la programación orientada a objetos, con lo que genera dos archivos más que contienen las funciones de las señales de cada elemento de la interfaz así como también las mismas propiedades de los dos archivos generados primeramente. Éstos archivos pueden ser modificados para agregar la funcionalidad a la interfaz gráfica.

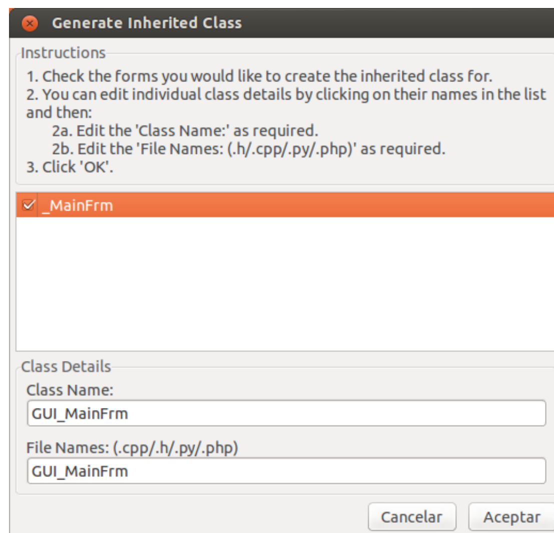

Figura 3.4: Generación de Clase Heredada

3.5. Integración de la Funcionalidad de la Interfaz Gráfica

En ésta sección se trabaja con el código generado anteriormente, en el archivo de cabecera (.h) se encuentra la descripción de todos los nombres de las funciones que se generaron a partir de las señales que se les dio a cada elemento de la interfaz y en los archivos de implementación (.cpp) es donde el programador deberá añadir la funcionalidad de cada señal. Siguiendo con el ejemplo anterior la Tabla. 3.3 muestra la señal del botón *Guardar* de la interfaz gráfica, y como se describe e implementa en ambos archivos generados. De manera análoga se hace lo mismo con las demás funciones que se hayan generado.

```
Fichero de cabecera (.h)

void OnSaveAs( wxCommandEvent& event );


Fichero de implementación (.cpp)

void GUI_MainFrm::OnSaveAs( wxCommandEvent& event )
{
    /* Código que será ejecutado */
    /* Ejemplo: Salvar proyecto */
}
```

Tabla 3.3: Ejemplo de las Señales Generadas

Capítulo 4

Resultados

Éste capítulo describe los resultados obtenidos a partir de la metodología descrita anteriormente, lo cual incluye el funcionamiento tanto del compilador de manera independiente así como su integración con la interfaz gráfica. Para comprobar la funcionalidad del compilador como herramienta independiente se simuló una prueba para el encendido de tres motores. Por otro lado se muestra la experimentación llevada a cabo en un invernadero a escala desde la creación del diagrama escalera para su control, la generación de las ecuaciones booleanas que describen dicho diagrama, hasta la compilación del código objeto que será programado en el controlador lógico programable para controlar el funcionamiento del invernadero a escala.

4.1. Compilador como herramienta independiente

Debido a que la programación utilizada para el desarrollo de esta investigación fue una programación estructurada o modular, es posible utilizar el compilador como una herramienta totalmente independiente y carente de una interfaz como ocurre con el compilador 'gcc' de Linux o su versión portada a Windows 'MinGW'. Al igual que los anteriores se sigue la misma línea ya que el comando se manda llamar con un parámetro, dicho parámetro es el nombre del archivo fuente que se requiere compilar como muestra la Fig. 4.1.

```
jesus@jesus-vaio: ~/Development/uaqc
jesus@jesus-vaio:~/Development/uaqc$ ./uaqc test
```

Figura 4.1: Llamada del Compilador desde la Terminal en Linux

Una vez realizada la compilación correcta sin error alguno, el compilador entrega el archivo ensamblado con las instrucciones que serán interpretadas por el PLC, dichas instrucciones aun pueden ser interpretadas por el usuario si se conoce la estructura del PLC, esto podría ser de utilidad en caso que el usuario quiera depurar o seguir el proceso realizado por el PLC. Por otro lado el compilador cuenta con un módulo de ensamblado el cual también genera el archivo objeto que es el que se programa en el PLC, dicho archivo es más complicado de interpretar por un usuario debido a que se genera un lenguaje de muy bajo nivel.

Para comprobar ésto se plantea la siguiente prueba de simulación un proceso genérico que se puede presentar en cualquier máquina. Dicho proceso consiste en que al ser pulsado un botón se activan tres dispositivos (motores) de manera secuencial y se mantienen ejecutados de forma cíclica, esto es, cada dispositivo se mantiene encendido cierto tiempo y se pasa al siguiente. Para representar dicho proceso se propone el diagrama de la Fig. 4.2 en el que se utilizan tres temporizadores (TIM0, TIM1, TIM2) para mantener encendidos los tres dispositivos (BAN9, BAN10, BAN11). De igual manera se incluye el botón de inicio (BAN4) y un botón de paro de emergencia (BAN5).

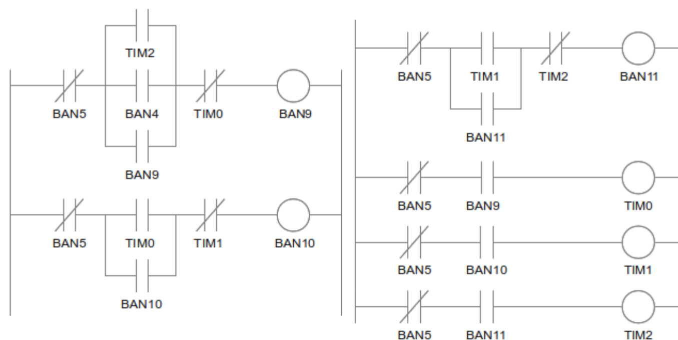

Figura 4.2: Diagrama Escalera de Prueba con Motores

La Fig. 4.3 muestra la representación del diagrama escalera anterior en ecuaciones booleanas descritas bajo las condiciones del lenguaje propuesto. Dicho código contiene intencionalmente

diversos errores de sintaxis (marcados con círculos rojos), para que el compilador, al momento de analizarlo, muestre los errores que contenga el código.

```
1  init
2      BAN9  = /BAN5 * (BAN4 + BAN9 + TIM2) * /TIM0
3      BAN10 = /BAN5 * ((TIM0 + BAN10/) * /TIM1
4      BAN11 = /BAN5 ** (TIM1 + BAN11) * /TIM2
5      TIM0  = /BAN5 * BAN9
6      TIM1  = /BAN5 *+ BAN10
7      TIM2  = /BAN5 * BAN11
8  end
```

Figura 4.3: Código Fuente de la Prueba con Motores

La Fig. 4.4 muestra el análisis que realizó el compilador sobre el código anterior, como se observa, el compilador detecto correctamente los errores y se muestran definiendo primeramente el nombre del archivo, seguido de la línea en la que se encuentra el error, así como también el tipo de error con una breve descripción del mismo y una posible solución.

```
PLC.XPP:3: syntax error, unexpected END, expecting
 EQ or OR or AND or RIGHT_PARENTHESIS
PLC.XPP:3: syntax error, unexpected NOT, expecting
 EQ or OR or AND or RIGHT_PARENTHESIS
PLC.XPP:4: syntax error, unexpected AND, expecting
 ID or NOT or LEFT_PARENTHESIS
PLC.XPP:6: syntax error, unexpected OR, expecting
ID or NOT or LEFT_PARENTHESIS
```

Figura 4.4: Análisis Entregado por el Compilador

Como se mencionó, una vez corregidos los errores se genera la traducción de las ecuaciones en notación infija al código en ensamblador que puede ser interpretado por el usuario (Tabla 4.1), de igual manera el código que será grabado en el PLC.

```
#include "Memory_Mapping.h"

NOT
ALD     0x0000
NOT
ARG16   dout0
NOT
ALD     0x0000
NOT
ARG16   dban0
LBIT1   dban0   -- Ban_1
NOT             -- NOT
LBIT2   dban0   -- Ban_2
LBIT0   dban0   -- Ban_0
OR              -- OR
LBIT0   dtim0   -- Timer_0
OR              -- OR
AND             -- AND
LBIT1   dtim0   -- Timer_1
NOT             -- NOT
AND             -- AND
MBIT0   dban0   -- Ban_0
LBIT1   dban0   -- Ban_1
NOT             -- NOT
LBIT1   dtim0   -- Timer_1
LBIT3   dban0   -- Ban_3
OR              -- OR
AND             -- AND
LBIT2   dtim0   -- Timer_2
NOT             -- NOT
AND             -- AND
MBIT3   dban0   -- Ban_3
LBIT1   dban0   -- Ban_1
NOT             -- NOT
LBIT2   dtim0   -- Timer_2
LBIT4   dban0   -- Ban_4
OR              -- OR
AND             -- AND
LBIT0   dtim0   -- Timer_0
NOT             -- NOT
AND             -- AND
MBIT4   dban0   -- Ban_4
LBIT1   dban0   -- Ban_1
NOT             -- NOT
LBIT0   dban0   -- Ban_0
AND             -- AND
MBIT1   dtim0   -- Timer_1
LBIT1   dban0   -- Ban_1
NOT             -- NOT
LBIT3   dban0   -- Ban_3
AND             -- AND
MBIT2   dtim0   -- Timer_2
LBIT1   dban0   -- Ban_1
NOT             -- NOT
LBIT4   dban0   -- Ban_4
AND             -- AND
MBIT0   dtim0   -- Timer_0
PUSHSET
PUSHSET
PUSHSET
PUSHSET
FIN
```

Tabla 4.1: Código Generado por el Compilador

Como se puede observar en el código se incluye un archivo de cabecera como suele ocurrir en lenguajes como C/C++, ésto es requerido ya que dicho archivo contiene el mapeo de todas las direcciones del PLC (Tabla 4.2), ya que el código original solo genera enlaces a las mismas.

```
#ifndef MEMORY_MAPPING_H
#define MEMORY_MAPPING_H

#define dban0 0x0037
#define dban1 0x0039
#define dban2 0x003A
#define dban3 0x003B
#define dban4 0x003C

#define din0 0x002D
#define din1 0x002E
#define din2 0x002F
#define din3 0x0030
#define din4 0x0031

#define dout0 0x0032
#define dout1 0x0033
#define dout2 0x0034
#define dout3 0x0035
#define dout4 0x0036

#define dtim0 0x0000
#define dtim1 0x0005
#define dtim2 0x000A
#define dtim3 0x000F

#endif
```

Tabla 4.2: Fragmento del Archivo de Cabecera con el Mapeo de las Direcciones

Para realizar dicho mapeo de direcciones se hace uso del precompilador de GCC con el comando siguiente.

- *cpp nombredelarchivo -o nombrenuevo -P*

Con ésto tendremos el archivo generado con las direcciones reales de la memoria. Cabe destacar que éste comando es llamado de manera automática por el compilador.

La prueba anterior se implementó en una tarjeta Spartan-3E en la que los leds representan los motores que se encendieron de manera secuencial, dicha simulación se muestra en la Fig. 4.5.

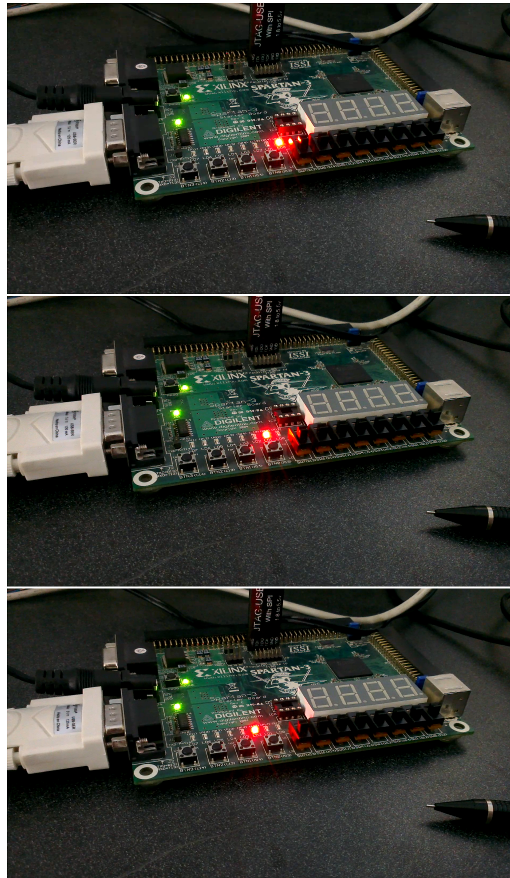

Figura 4.5: Simulación de Encendido Secuencial de Motores

4.2. Interfaz Gráfica

El software cuenta con elementos clásicos en el diseño de una interfaz gráfica, en primera instancia se tiene una barra de menús mostrada en la Fig. 4.6.

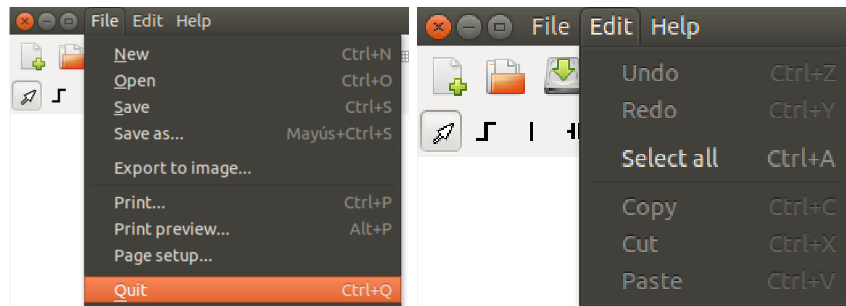

Figura 4.6: Barra de Menús

A continuación se describen los elementos que se incluyen en cada menú.

- File

  New Crea un nuevo proyecto.

  Open Abre un proyecto existente.

  Save Salva un proyecto previamente guardado.

  Save as Salva un proyecto nuevo.

  Export to image Exporta el proyecto a una imagen.

  Print Imprime el proyecto.

  Print preview Vista previa de la impresión.

  Page setup Configurar la página de impresión.

  Quit Salir del programa.

- Edit

  Undo Deshacer una acción.

  Redo Rehacer una acción previamente deshecha.

  Select all Seleccionar todos los elementos del área de trabajo.

  Copy Copiar uno o varios elementos del área de trabajo.

  Cut Cortar uno o varios elementos del área de trabajo.

  Paste Pegar uno o varios elementos del área de trabajo.

Como se puede observar éstas opciones dan una gran funcionalidad al software y hace que la interacción con el usuario sea más sencilla e intuitiva. Para diferenciar los archivos de proyectos creados con éste software se propuso una extensión para los mismos, dicha extensión es '.ldp'.

De igual manera el software incluye dos barras de herramientas (Fig. 4.7), una de ellas muestra los accesos directos a funciones que se encuentran en la barra de menús, más otras tres opciones

extra al final, las cuales son: activar o desactivar la rejilla del área de trabajo, activar o desactivar las sombras de los elementos y activar o desactivar el contexto gráfico, éste último depende de las capacidades del sistema para mejorar o no los gráficos. La segunda barra de herramientas incluye los elementos utilizados para generar los diagramas escalera, desde los contactos, bobinas, conexiones, etc. También se incluyen las opciones de generar ecuaciones a partir del diagrama, compilar y programar el PLC.

Figura 4.7: Barras de Herramientas

4.3. Compilador Integrado a la Interfaz Gráfica

Para la implementación del software como un sistema completo (compilador e interfaz gráfica) se realizó la prueba de control de un invernadero a escala, dicho control solo requiere de eventos de encendido y apagado (ON-OFF). La Fig. 4.8 muestra el invernadero a escala utilizado para la implementación del software.

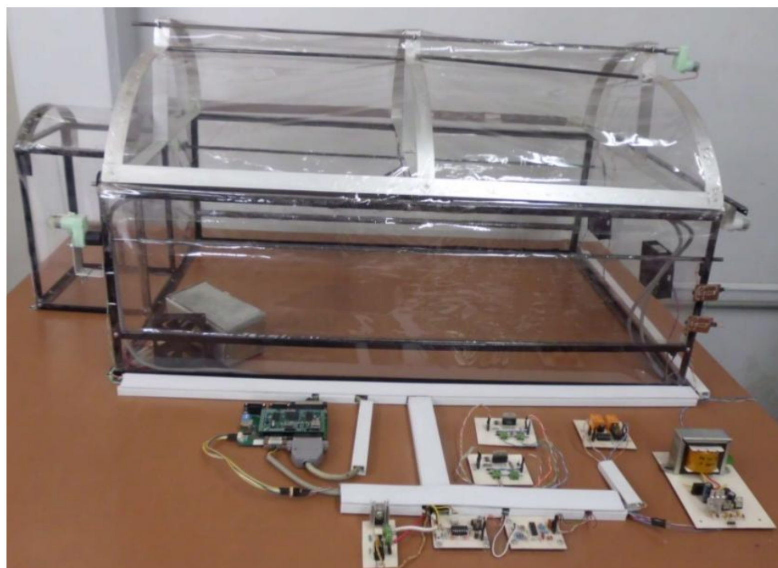

Figura 4.8: Invernadero a Escala

El invernadero a escala está basado en un invernadero tipo raspa y amagado, los materiales utilizados para su construcción fueron aluminio, madera, plástico entre otros. La instrumentación se realizó tomando en cuenta que el control se basara en eventos ON-OFF empleando sensores de temperatura, humedad e iluminación con sus tarjetas de amplificación, los cuales ayudaron a saber el estado de las variables, también se instalaron algunos sensores de limite que se usaran para conocer la posición de ventanas como de la malla sombra, se adaptaron pequeños actuadores en ventanas y malla sombra con sus respectivas tarjetas de potencia, se coloco una resistencia y algunos ventiladores para disipar el calor con sus tarjetas de activación, además de colocar una pequeña bomba de riego con su tarjeta de activación (Saucedo, 2012).

Para poder realizar el control del invernadero primeramente se definen las entradas (Tabla 4.3) y salidas (Tabla 4.4) que se requieran para poder realizar el diagrama escalera en la interfaz gráfica.

Sensor	**Nombre**	**Descripción**
1	IN0	Posición 1 de la ventana lateral
2	IN1	Posición 2 de la ventana lateral
3	IN2	Posición 3 de la ventana lateral
4	IN3	Posición 4 de la ventana lateral
Paro	IN4	Paro de emergencia
5	IN5	Posición inicial de la ventana superior
6	IN6	Posición final de la ventana superior
7	IN7	Posición inicial de la malla sombra
8	IN8	Posición final de la malla sombra

Tabla 4.3: Entradas

Nombre	**Función**
OUT0	Abre ventana lateral
OUT1	Cierra ventana lateral
OUT2	Abre ventana superior
OUT3	Cierra ventana superior
OUT4	Enciende bomba de riego
OUT5	Abre malla sombra
OUT6	Cierra malla sombra
OUT7	Enciende resistencia
OUT8	Enciende ventiladores

Tabla 4.4: Salidas

La Fig. 4.9 muestra el diseño del diagrama escalera para el control del invernadero a escala realizado en la interfaz gráfica con las herramientas que se incluyen en la misma. A pesar de que el área de trabajo es pequeña, tiene la capacidad de extenderse a medida que se agranda el diagrama que se realice, haciendo uso de barras laterales y una herramienta de zoom controlada por la rueda del mouse (ratón).

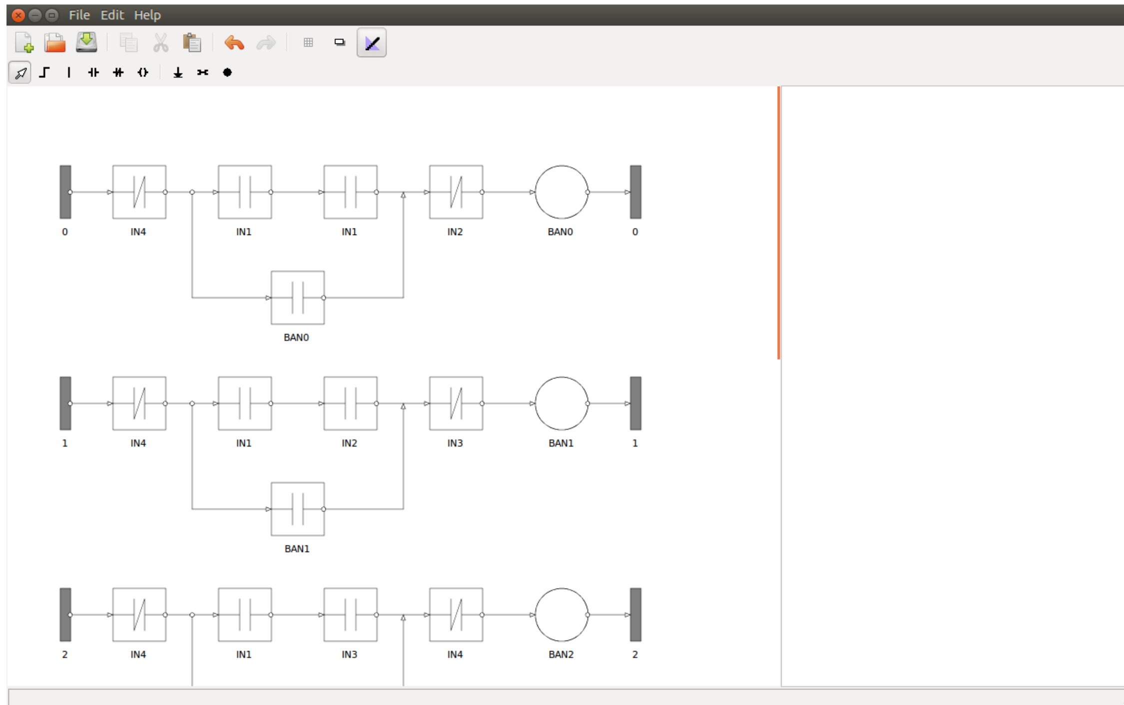

Figura 4.9: Diseño del Diagrama Escalera

Al finalizar el diseño del diagrama escalera se procede a la generación de las ecuaciones booleanas del mismo. Ésto se muestra en la Fig. 4.10.

Debido a que el programa genera de manera automática las ecuaciones, ésto evita los errores de sintaxis que genera un usuario al escribir las ecuaciones de manera manual, por lo que al realizar la compilación, ésta será realizada sin error alguno (Fig. 4.11).

Por último, una vez compiladas las ecuaciones se puede realizar la programación del PLC, como se muestra en la Fig. 4.12.

Cabe destacar que el usuario puede escribir las ecuaciones booleanas de manera manual directamente en la interfaz gráfica y posteriormente realizar los pasos de compilación y programación.

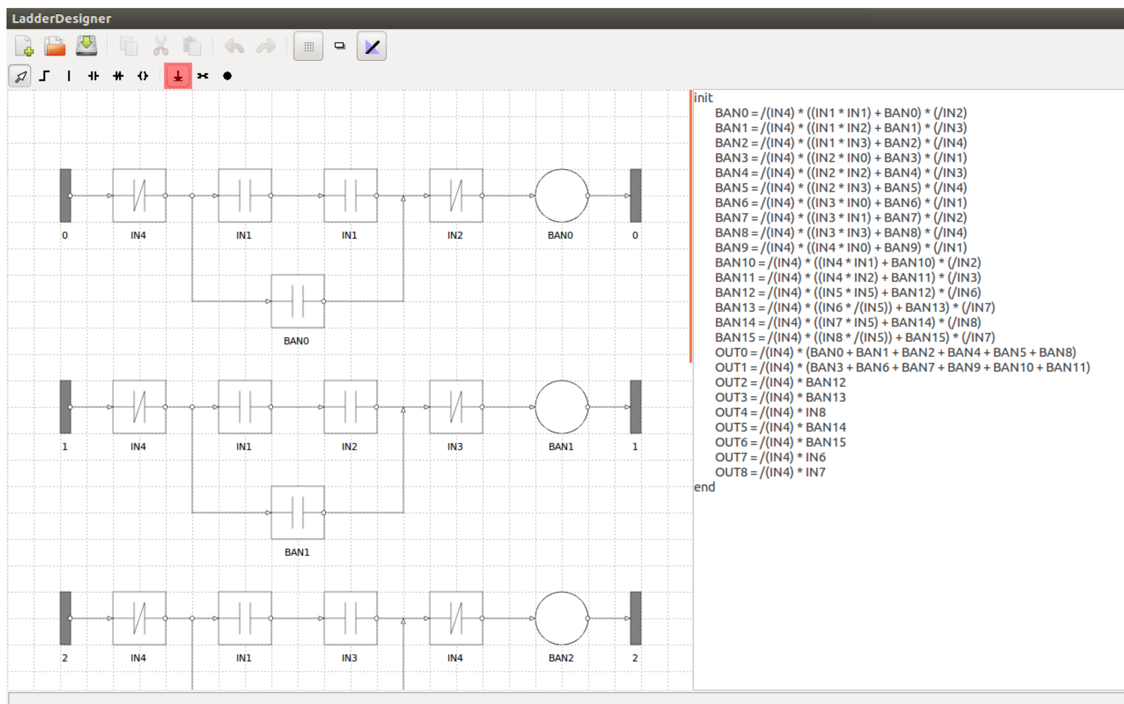

Figura 4.10: Generación de las Ecuaciones Booleanas

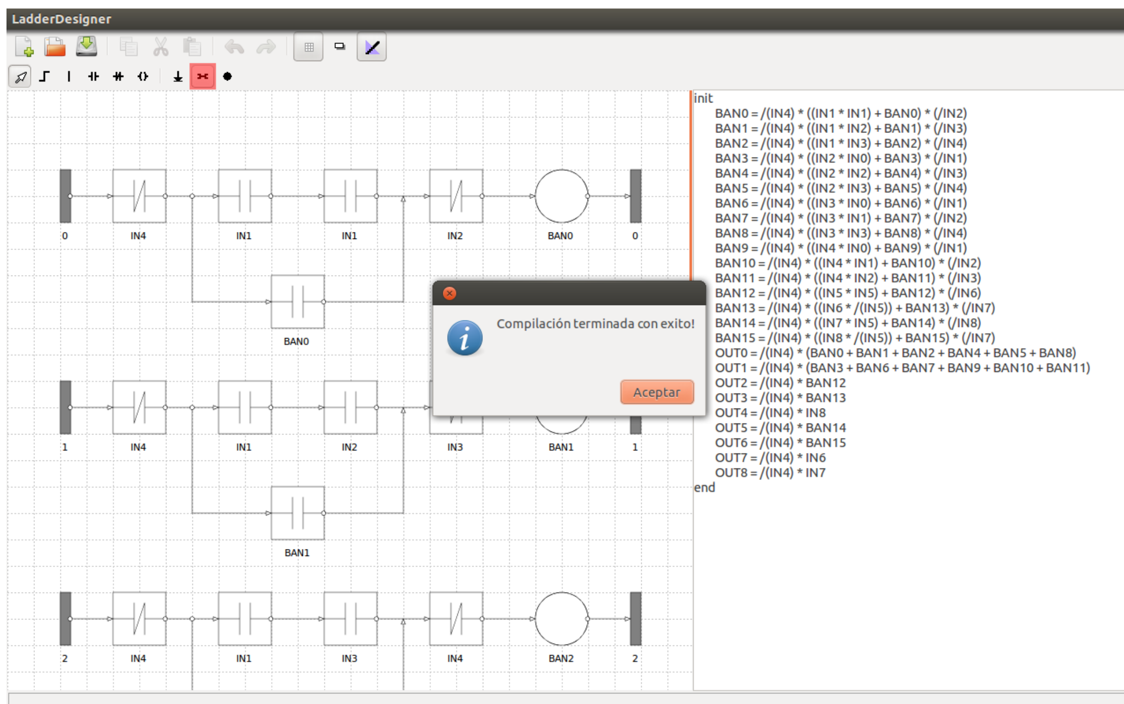

Figura 4.11: Compilación desde la Interfaz

Figura 4.12: Programación del PLC desde la Interfaz

Capítulo 5

Conclusiones y Prospectivas

En base a los resultados obtenidos se puede observar que con un sistema de control y un traductor (compilador) se facilita el diseño y programación del PLC basado en FPGA desarrollado anteriormente. Con ésto, el usuario no sólo tiene la posibilidad de programar un proceso que requiera de lógica programable sino que también gráficamente podrá diseñar los diagramas escalera de dicho proceso gracias a la integración de una interfaz gráfica amigable e intuitiva con la que el usuario podrá interactuar de manera rápida y sencilla.

Como se puede observar es importante contar con sistemas totalmente integrales o embebidos, esto es, herramientas que cuenten con Hardware y Software para complementarse. Por otro lado elaborar dichas herramientas con módulos reconfigurables ofrece diversas ventajas, por ejemplo en el caso del Hardware, debido a que está diseñado bajo una plataforma FPGA permite que se porte a diversos procesos y que de igual manera se mantenga al día con los avances tecnológicos, ya que éstos al estar en constante cambio muchas de las herramientas comerciales quedan obsoletas al paso del tiempo. El FPGA ofrece ésta posibilidad de reconfigurarse y adaptarse a nuevas tecnologías sin necesidad de cambiar todo el proceso. Por el lado del Software, gracias a la programación modular y estructurada se obtiene la misma ventaja que en el Hardware aplicaciones fácilmente actualizables y escalables a futuro. Ésta claro que por separado éstas herramientas ofrecen diversas ventajas, pero al tomar las características de ambas e integrarlas en un solo sistema se obtienen herramientas muy

poderosas para la automatización de procesos industriales.

Al ser la primera versión de ésta aplicación se deja abierta la posibilidad de integrar más módulos que se requieran, así como también la optimización y mejoramiento de algunos de los algoritmos incluidos (por ejemplo, el algoritmo para la generación de las ecuaciones), así como también incluir todas las funciones que el PLC (hardware) puede realizar. De igual manera se pretende que en un futuro se cambie la comunicación RS-232 por una comunicación USB para la programación del PLC, ésto para adaptarse a las computadoras que ya no cuentan con dicho puerto además de que la comunicación sería más rápida.

Referencias

[1] Aho, Alfred V., Sethi, Ravi., Ullman, Jeffrey D. 1998. Compiladores: Principios, técnicas y herramientas. 1a Reimpresión en México. Addison Wesley Longman de México S.A. de C.V. ISBN 968-444-333-1

[2] Bateman, Andy. 2003. Comunicaciones Digitales: Diseño para el mundo real. Marcombo S. A. Barcelona. ISBN 84-267-1337-8.

[3] Chamillar, Gilles. 2011. UBUNTU Administración de un sistema Linux. Ediciones ENI. Barcelona. ISBN 978-2-7460-6669-4.

[4] Cruz López, Víctor. 2008. Controladores lógicos programables con aplicación a una máquina de inyección de hule. Tesina. Universidad Autónoma de Querétaro.

[5] Femat Díaz, Aurora. 2004. Desarrollo de la ingeniería de software para la implementación de un CNC. Tesis. Querétaro, Qro. Universidad Autónoma de Querétaro.

[6] GNU Operating System <http://www.gnu.org/>. 2012.

[7] Hernández Rodríguez, Leonardo Alonso., Jaramillo Valbuena, Sonia., Cardona Torres, Sergio Agusto. 2010. Practique la Teoría de Autómatas y Lenguajes Formales. Ediciones Elizcom. Colombia. ISBN 978-958-44-7913-6

[8] Hernández Salinas, Raúl Fanny. 2004. Automatización de procesos secuenciales discretos. Tesina. Universidad Autónoma de Querétaro.

[9] Hernández Zea, Juan Pablo. 2008. Tarjeta de interfaz para aplicaciones de control en máquina inyectora de plástico usando Lógica Programable. Tesis. Universidad Autónoma de Querétaro.

[10] Hyde, J., Regué, J., Cuspinera, A. 1997. Control Electroneumático y Electrónico. Norgren Biblioteca Técnica. España. ISBN 84-267-1097-2.

[11] Mejía Alonso, Diego. 2004. Programación de una máquina soldadora de Bracket con un PLC 1500 de la marca Allen Bradley. Tesina. Universidad Autónoma de Querétaro.

[12] Mochi Alemán, Prudencio Óscar. 2006. La industría del software en México en el contexto internacional y latinoamericano. Universidad Nacional Autónoma de México. 1a Edición. ISBN 970-32-3095-4.

[13] Morris Mano, M. 1993. Arquitectura de computadoras. Prentice Hall. 3a Edición. México. ISBN 968-880-361-8

[14] Muñoz Barrón, Benigno. 2009. Diseño de un procesador PLC basado en FPGA para aplicación en maquinaria CNC. Tesis. San Juan del Río, Querétaro. Universidad Autónoma de Querétaro.

[15] Pons, Nicolas. 2005. LINUX Principios básicos del uso del sistema. Ediciones ENI. Barcelona. ISBN 2-7460-2796-8.

[16] Sánchez Dueñas, G., Valverde Andreu, J. A. 1989. Compiladores e intérpretes un enfoque pragmático. Ediciones Díaz de Santos. 2a edición. ISBN 84-87189-06-7.

[17] Sánchez Gómez, Jesús Iván. 2011. Desarrollo de interfaz gráfica para robot PUMA bajo plataforma linux. Tesis. San Juan del Río, Querétaro. Universidad Autónoma de Querétaro.

[18] Saucedo Dorantes, Juan José. 2012. Interfaz de usuario para invernadero controlado por PLC basado en FPGA. Tesis. San Juan del Río, Querétaro. Universidad Autónoma de Querétaro.

[19] Smart, Julian., Hock, Kevin., Csomor, Stefan. 2006. Cross-Platform GUI Programming with wxWidgets. Prentice Hall PTR. ISBN 0-13-147381-6

[20] Torres Soria, Adán. 2004. Control Automático de una planta de cromado con PLC. Universidad Autónoma de Querétaro.

[21] Ugalde Estrella, Javier. 2011. Software para controlador de movimiento basado en FPGA. Tesis. San Juan del Río, Querétaro. Universidad Autónoma de Querétaro.

[22] Wildi, Theodore. 2007. Máquinas Eléctricas y Sistemas de Potencia. Pearson Educación de México, S.A. de C.V. 6ta. Edición. ISBN 970-26-0814-7

[23] wxShapeFramework <http://wxsf.sourceforge.net/>. 2013

[24] wxWidgets Cross-Platform GUI Library <http://www.wxwidgets.org/>. 2013

Apéndice A

Código Fuente en la Nube

La computación en nube es un sistema informático basado en Internet y centros de datos remotos para gestionar servicios de información y aplicaciones. La computación en nube permite que los consumidores y las empresas gestionen archivos y utilicen aplicaciones sin necesidad de instalarlas en cualquier computadora con acceso a Internet. Esta tecnología ofrece un uso mucho más eficiente de recursos, como almacenamiento, memoria, procesamiento y ancho de banda, al proveer solamente los recursos necesarios en cada momento.

El término "nube" se utiliza como una metáfora de Internet y se origina en la nube utilizada para representar Internet en los diagramas de red como una abstracción de la infraestructura que representa (`http://www.computacionennube.org/`).

Aprovechando ésta nueva tecnología se puede acceder al código fuente de la aplicación desde cualquier ubicación haciendo uso de un control de versiones.

A.1. Git: Control de Versiones

Git es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran

número de archivos de código fuente. Al principio, Git se pensó como un motor de bajo nivel sobre el cual otros pudieran escribir la interfaz de usuario o front end. Sin embargo, Git se ha convertido desde entonces en un sistema de control de versiones con funcionalidad plena. Hay algunos proyectos de mucha relevancia que ya usan Git, en particular, el grupo de programación del núcleo Linux.

El mantenimiento del software Git está actualmente (2009) supervisado por Junio Hamano, quien recibe contribuciones al código de alrededor de 280 programadores.

A.1.1. Instalación de Git

Linux

La instalación en Linux se realiza como sigue.

```
$ sudo apt-get install git
```

Windows

La instalación de Windows se realiza de manera sencilla descargando el ejecutable desde la página oficial de GitHub `http://msysgit.github.com/` e instalarlo. Una vez instalado, se tendrá tanto la versión de línea de comandos como la interfaz gráfica de usuario estándar.

A.2. Repositorio del Código fuente

Para tener acceso al código fuente tanto de la Interfaz Gŕafica como del Compilador se requiere del siguiente comando:

Compilador

```
$ git clone git@bitbucket.org:jesus_sangomez/uaqc.git
```

Interfaz Gráfica

```
$ git clone git@bitbucket.org:jesus_sangomez/ladderdesigner.git
```

Con éstos simples comandos se puede tener la última versión de todo el código fuente de la Interfaz Gráfica y el Compilador, desde cualquier ubicación.

Apéndice B

Código del Compilador

Los siguientes códigos se crean en una carpeta raíz, en éste caso `uaqc`.

Escribe el siguiente código en un fichero llamado `uaqc.l`:

```
/***************************************************************************
                        Scanner for the Simple language
***************************************************************************/
/*=========================================================================
                        C-libraries and Token definitions
=========================================================================*/

%{
#include <string.h>             /* for strdup */
#include "../src/uaqc.tab.h"    /* for token definitions and yylval */
%}

/*=========================================================================
                        TOKEN Definitions
=========================================================================*/

DIGIT [0-9]
FLG [banBAN][0-9]*
TMR [timTIM][0-9]*
IN [inIN][0-9]*
OUT [outOUT][0-9]*

/*=========================================================================
```

```
24         REGULAR EXPRESSIONS defining the tokens for the Simple language
25  =========================================================================*/
26
27  %%
28  init                    { return(INIT); }
29  end                     { return(END); }
30  {DIGIT}+                { yylval.intval = atoi( yytext ); return(NUMBER); }
31  {FLG}+                  { yylval.id = (char *) strdup(yytext); return(BAN); }
32  {TMR}+                  { yylval.id = (char *) strdup(yytext); return(TIM); }
33  {IN}+                   { yylval.id = (char *) strdup(yytext); return(IN); }
34  {OUT}+                  { yylval.id = (char *) strdup(yytext); return(OUT); }
35  "*"                     { return ('*'); }
36  "+"                     { return ('+'); }
37  "="                     { return ('='); }
38  "/"                     { return ('/'); }
39  "("                     { return ('('); }
40  ")"                     { return (')'); }
41  [ \t\n\r\f]+            { /* eat up whitespace */ }
42  .                       { yyerror("Invalid Character"); printf("%c\n",yytext[0]); }
43  %%
44
45  int yywrap(void){}
46
47  /************************** End Scanner File ****************************/
```

Escribe el siguiente código en un fichero llamado `uaqc.y`:

```
1   /*************************************************************************
2                           Compiler for the Simple language
3   *************************************************************************/
4   /*=======================================================================
5           C Libraries, Symbol Table, Code Generator & other C code
6   =======================================================================*/
7
8   %{
9   #include <stdio.h>              /* For I/O */
10  #include <stdlib.h>             /* For malloc here and in symbol table */
11  #include <string.h>             /* For strcmp in symbol table */
12  #include "../include/ST.h"      /* Symbol Table */
13  #include "../include/SM.h"      /* Stack Machine */
14  #include "../include/CG.h"      /* Code Generator */
15  #define YYDEBUG 1               /* For Debugging */
16  int errors;                     /* Error Count */
```

```
17
18 /*--------------------------------------------------------------------------
19                 Install identifier & check if previously defined.
20 --------------------------------------------------------------------------*/
21
22 install ( char *sym_name, int type)
23 {
24         symrec *s_flag, *s_timer, *s_in, *s_out;
25         switch (type){
26                 case 0:
27                         s_flag = getsym (sym_name);
28                         if (s_flag == 0)
29                                 s_flag = putsym (sym_name);
30                         else {
31                                 //errors++;
32                                 //printf( "%s is already defined\n", sym_name );
33                         }
34                         break;
35                 case 1:
36                         s_timer = gettim (sym_name);
37                         if (s_timer == 0)
38                                 s_timer = puttim (sym_name);
39                         else {
40                                 //errors++;
41                                 //printf( "%s is already defined\n", sym_name );
42                         }
43                         break;
44                 case 2:
45                         s_in = getin (sym_name);
46                         if (s_in == 0)
47                                 s_in = putin (sym_name);
48                         else {
49                                 //errors++;
50                                 //printf( "%s is already defined\n", sym_name );
51                         }
52                         break;
53                 case 3:
54                         s_out = getout (sym_name);
55                         if (s_out == 0)
56                                 s_out = putout (sym_name);
57                         else {
58                                 //errors++;
59                                 //printf( "%s is already defined\n", sym_name );
60                         }
```

```
61                          break;
62          }
63  }
64
65  /*--------------------------------------------------------------------------
66                  If identifier is defined, generate code
67  --------------------------------------------------------------------------*/
68  context_check( enum code_ops operation, char *sym_name, int type)
69  {
70          symrec *identifier, *id_timer, *id_in, *id_out;
71          switch(type){
72                  case 0:
73                          identifier = getsym( sym_name );
74                          if ( identifier == 0 )
75                          {
76                                  //errors++;
77                                  printf( "%s", sym_name );
78                                  printf( "%s\n", " is an undeclared identifier" );
79                          }
80                          else
81                                  gen_code( operation, identifier->offset );
82                          break;
83                  case 1:
84                          id_timer = gettim( sym_name );
85                          if ( id_timer == 0 )
86                          {
87                                  //errors++;
88                                  printf( "%s", sym_name );
89                                  printf( "%s\n", " is an undeclared identifier" );
90                          }
91                          else
92                                  gen_code( operation, id_timer->offset );
93                          break;
94                  case 2:
95                          id_in = getin( sym_name );
96                          if ( id_in == 0 )
97                          {
98                                  //errors++;
99                                  printf( "%s", sym_name );
100                                 printf( "%s\n", " is an undeclared identifier" );
101                         }
102                         else
103                                 gen_code( operation, id_in->offset );
104                         break;
```

```
105                 case 3:
106                         id_out = getout( sym_name );
107                         if ( id_out == 0 )
108                         {
109                                 //errors++;
110                                 printf( "%s", sym_name );
111                                 printf( "%s\n", " is an undeclared identifier" );
112                         }
113                         else
114                                 gen_code( operation, id_out->offset );
115                         break;
116         }
117 }
118
119 %}
120
121 /*===========================================================================
122                         SEMANTIC RECORDS
123 ===========================================================================*/
124
125 %union semrec                   /* The Semantic Records */
126 {
127         int intval;             /* Integer values */
128         char *id;               /* Identifiers */
129 }
130
131 /*===========================================================================
132                                 TOKENS
133 ===========================================================================*/
134
135 %start program
136 %token <intval> NUMBER                  /* Simple integer */
137 %token <id> BAN TIM IN OUT              /* Simple identifier */
138 %token INIT END
139
140 /*===========================================================================
141                 GRAMMAR RULES for the Ladder Boolean language
142 ===========================================================================*/
143
144 %%
145
146 program : /*variables*/
147         INIT
148                 commands
```

```
149         END { gen_code(END_PROGRAM, 0); YYACCEPT; }
150         ;
151
152 /*variables:
153         | variables definitions
154         ;
155
156 definitions: primary_var '=' constant
157         ;
158
159 primary_var:
160         BAN { install($1); }
161         ;
162
163 constant:
164         NUMBER
165         ;*/
166
167 commands: /* empty */
168         | commands expr
169         ;
170
171 expr: assign_expr
172         ;
173
174 assign_expr: or_expr
175         | assign_expr '=' or_expr { gen_code(ASSIGN, 0); }
176         ;
177
178 or_expr: and_expr
179         | or_expr '+' and_expr { gen_code(OR, 0); }
180         ;
181
182 and_expr: not_expr
183         | and_expr '*' not_expr { gen_code(AND, 0); }
184         ;
185
186 not_expr: primary
187         | '/' primary { gen_code(NOT, 0); }
188
189 primary:
190         BAN { install($1, 0); context_check(FLAG, $1, 0); }
191         | TIM { install($1, 1); context_check(TIMER, $1, 1); }
192         | IN { install($1, 2); context_check(INPUT, $1, 2); }
```

```
193         | OUT { install($1, 3); context_check(OUTPUT, $1, 3); }
194         | '(' expr ')'
195         ;
196
197 %%
198
199 /*==========================================================================
200                                 MAIN
201 ==========================================================================*/
202 int main( int argc, char *argv[] )
203 {
204         extern FILE *yyin;
205         ++argv; --argc;
206         yyin = fopen( argv[0], "r" );
207         // yydebug = 1;
208         errors = 0;
209         yyparse ();
210         if ( errors == 0 )
211         {
212                 //print_code ();
213                 fetch_execute_cycle();
214                 printf("Compilacion correcta 0 errores...\n");
215         }
216         return 0;
217 }
218
219 /*==========================================================================
220                                 YYERROR
221 ==========================================================================*/
222 yyerror ( char *s ) /* Called by yyparse on error */
223 {
224         errors++;
225         printf ("%s\n", s);
226 }
227
228 /*************************** End Grammar File **************************/
```

Escribe el siguiente código en un fichero llamado `makefile`:

```
1 # Makefile: A simple makefile for uaqc.
2 default:
3         bison -dv uaqc.y -o src/uaqc.tab.c
4         flex -o src/lex.yy.c uaqc.l
```

```
5	        gcc -c src/uaqc.tab.c -o obj/uaqc.tab.o
6	        gcc -c src/lex.yy.c -o obj/lex.yy.o
7	        gcc -o bin/uaqc obj/uaqc.tab.o obj/lex.yy.o
8	clean:
9	        rm src/*.* obj/*.*
```

Posteriormente se crea una carpeta que contendrá los archivos de cabecera necesarios, en éste caso `include`.

Escribe el siguiente código en un fichero llamado `CG.h`:

```
1	/***************************************************************************
2	                                Code Generator
3	***************************************************************************/
4	/*-------------------------------------------------------------------------
5	                                Data Segment
6	-------------------------------------------------------------------------*/
7	int data_offset = 0;    /* Initial offset */
8	int tim_offset = 0;     /* Initial offset */
9	int in_offset = 0;      /* Initial offset */
10	int out_offset = 0;     /* Initial offset */
11	
12	int data_location(int type) /* Reserves a data location */
13	{
14	        int data;
15	        switch(type){
16	                case 0: data = data_offset++; break;
17	                case 1: data = tim_offset++; break;
18	                case 2: data = in_offset++; break;
19	                case 3: data = out_offset++; break;
20	        }
21	        return data;
22	}
23	
24	/*-------------------------------------------------------------------------
25	                                Code Segment
26	-------------------------------------------------------------------------*/
27	int code_offset = 0; /* Initial offset */
28	int eight_bflag = 0;
29	int eight_tflag = 0;
30	int eight_iflag = 0;
31	int eight_oflag = 0;
32	
```

```
33  int gen_label() /* Returns current offset */
34  {
35          return code_offset;
36  }
37
38  int reserve_loc() /* Reserves a code location */
39  {
40          return code_offset++;
41  }
42
43  /* Generates code at current location */
44  void gen_code( enum code_ops operation, int arg)
45  {
46          code[code_offset].op = operation;
47          code[code_offset].mem_alloc = 0;
48          if(operation == FLAG){
49                  if(arg < 8 )
50                          eight_bflag = 0;
51                  if(arg >= 8 && arg < 16)
52                          eight_bflag = 1;
53                  if(arg >= 16 && arg < 24)
54                          eight_bflag = 2;
55                  if(arg >= 24 && arg < 32)
56                          eight_bflag = 3;
57                  if(arg >= 32 && arg < 40)
58                          eight_bflag = 4;
59                  if(arg >= 40 && arg < 48)
60                          eight_bflag = 5;
61                  // printf("%d %d %d\n", arg, arg - (eight_flag * 8), eight_flag);
62                  code[code_offset].mem_alloc = arg - (eight_bflag * 8);
63
64                  code[code_offset].mem_address = eight_bflag;
65          }
66          if(operation == TIMER){
67                  if(arg < 8 )
68                          eight_tflag = 0;
69                  if(arg >= 8 && arg < 16)
70                          eight_tflag = 1;
71                  if(arg >= 16 && arg < 24)
72                          eight_tflag = 2;
73                  if(arg >= 24 && arg < 32)
74                          eight_tflag = 3;
75                  if(arg >= 32 && arg < 40)
76                          eight_tflag = 4;
```

```
77                  if(arg >= 40 && arg < 48)
78                          eight_tflag = 5;
79                  code[code_offset].mem_alloc = arg - (eight_tflag * 8);
80
81                  code[code_offset].mem_address = eight_tflag;
82          }
83          if(operation == INPUT){
84                  if(arg < 8 )
85                          eight_iflag = 0;
86                  if(arg >= 8 && arg < 16)
87                          eight_iflag = 1;
88                  if(arg >= 16 && arg < 24)
89                          eight_iflag = 2;
90                  if(arg >= 24 && arg < 32)
91                          eight_iflag = 3;
92                  if(arg >= 32 && arg < 40)
93                          eight_iflag = 4;
94                  if(arg >= 40 && arg < 48)
95                          eight_iflag = 5;
96                  code[code_offset].mem_alloc = arg - (eight_iflag * 8);
97
98                  code[code_offset].mem_address = eight_iflag;
99          }
100         if(operation == OUTPUT){
101                 if(arg < 8 )
102                         eight_oflag = 0;
103                 if(arg >= 8 && arg < 16)
104                         eight_oflag = 1;
105                 if(arg >= 16 && arg < 24)
106                         eight_oflag = 2;
107                 if(arg >= 24 && arg < 32)
108                         eight_oflag = 3;
109                 if(arg >= 32 && arg < 40)
110                         eight_oflag = 4;
111                 if(arg >= 40 && arg < 48)
112                         eight_oflag = 5;
113                 code[code_offset].mem_alloc = arg - (eight_oflag * 8);
114
115                 code[code_offset].mem_address = eight_oflag;
116         }
117         code[code_offset++].arg = arg;
118         ban_conf = eight_bflag;
119         tim_conf = eight_tflag;
120         in_conf  = eight_iflag;
```

```
121         out_conf = eight_oflag;
122 }
123
124 /* Generates code at a reserved location */
125 void back_patch( int addr, enum code_ops operation, int arg )
126 {
127         code[addr].op = operation;
128         code[addr].arg = arg;
129 }
130
131 /*-------------------------------------------------------------------------
132                                 Print Code to stdio
133 -------------------------------------------------------------------------*/
134 void print_code()
135 {
136         int i = 0;
137         while (i < code_offset) {
138                 printf("%3d: %-10s%4d\n",i,op_name[(int) code[i].op], code[i].arg );
139                 i++;
140         }
141 }
142
143 /************************* End Code Generator *************************/
```

Escribe el siguiente código en un fichero llamado `SM.h`:

```
1  /*************************************************************************
2                                  Stack Machine
3  *************************************************************************/
4  /*=========================================================================
5                                  DECLARATIONS
6  =========================================================================*/
7  /* OPERATIONS: Internal Representation */
8  enum code_ops {FLAG, INPUT, OUTPUT, TIMER, AND, OR, NOT, ASSIGN, END_PROGRAM};
9
10 /* OPERATIONS: External Representation */
11 char *op_name[] = {"FLAG", "INPUT", "OUTPUT", "TIMER", "AND", "OR", "NOT", "ASSIGN", "END_PROGRAM"
       };
12
13 struct instruction
14 {
15         enum code_ops op;
16         int arg;
```

```
17          int mem_alloc;
18          int mem_address;
19  };
20
21  /* CODE Array */
22  struct instruction code[999];
23
24  /* RUN-TIME Stack */
25  int stack[999];
26
27  /*--------------------------------------------------------------------------
28                                  Registers
29  --------------------------------------------------------------------------*/
30  int pc = 0;
31  struct instruction ir;
32  int ar = 0;
33  int top = 0;
34  char ch;
35
36  /* GenCode */
37  enum code_ops init_op;
38  int init_mem, init_address, init_arg, init_flag;
39  int type;
40  int ban_conf = 0;
41  int tim_conf = 0;
42  int in_conf = 0;
43  int out_conf = 0;
44
45  int push_set = 0;
46
47  FILE *fasm, *fhex;
48
49  void configuration_code()
50  {
51          int i;
52          fprintf( fasm, "#include \"Memory_Mapping.h\"\n\n" );
53          // for(i = 0; i<=in_conf; ++i)
54          //      fprintf( fasm, "NOT\nALD 0x0000\nNOT\nARG16 din%d\n", i );
55          for(i = 0; i<=out_conf; ++i){
56                  fprintf( fasm, "NOT\nALD\t\t0x0000\nNOT\nARG16\tdout%d\n", i );
57                  push_set += 2;
58          }
59          for(i = 0; i<=ban_conf; ++i){
60                  fprintf( fasm, "NOT\nALD\t\t0x0000\nNOT\nARG16\tdban%d\n", i );
```

```
61                  push_set += 2;
62          }
63  }
64
65  /*========================================================================
66                          Fetch Execute Cycle
67  ========================================================================*/
68  void fetch_execute_cycle()
69  {
70          fasm = fopen("uaqc.asm", "w+");
71          fhex = fopen("uaqc.hex", "w+");
72          configuration_code();
73          init_op = ASSIGN;
74          do {
75                  /* Fetch */
76                  ir = code[pc++];
77                  /* Execute */
78                  switch (ir.op) {
79                          case ASSIGN :
80                                  switch (type){
81                                          case 0: fprintf( fasm, "MBIT %d\tdban %d\t-- Ban_%d\n",
                                                    init_mem, init_address, init_arg ); break;
82                                          case 1: fprintf( fasm, "MBIT %d\tdin %d\t-- Input_%d\n",
                                                    init_mem, init_address, init_arg ); break;
83                                          case 2: fprintf( fasm, "MBIT %d\tdout %d\t-- Output_%d\n",
                                                    init_mem, init_address, init_arg ); break;
84                                          case 3: fprintf( fasm, "MBIT %d\tdtim %d\t-- Timer_%d\n",
                                                    init_mem, init_address, init_arg ); break;
85                                  }
86                                  //fprintf( fhex, "%02X\t\t--MBIT %d\ndir %d\t-- Variable_%d\n",
                                        init_mem + 19, init_mem, init_address, init_arg);
87                                  // fprintf( fasm, "MBIT %d\tdir %d_1 dir %d_2\t-- Variable_%d\n",
                                        init_mem, init_address, init_address, init_arg );
88                                  // fprintf( fhex, "%02X\t\t--MBIT %d\ndir %d_1\ndir %d_2\t--
                                        Variable_%d\n", init_mem + 19, init_mem, init_address,
                                        init_address, init_arg);
89                                  break;
90                          case FLAG :
91                                  if(init_op != ASSIGN){
92                                          fprintf( fasm, "LBIT %d\tdban %d\t-- Ban_%d\n", ir.mem_alloc
                                                    , ir.mem_address, ir.arg );
93                                          //fprintf( fhex, "%02X\t\t--LBIT %d\ndir %d\t-- Variable_%d\
                                                    n", ir.mem_alloc + 12, ir.mem_alloc, ir.mem_address,
                                                    ir.arg );
```

```
94                              // fprintf( fasm, "LBIT %d\tdir %d_1 dir %d_2\t-- Variable_%d
                                    \n", ir.mem_alloc, ir.mem_address, ir.mem_address, ir.
                                    arg );
95                              // fprintf( fhex, " %02X\t\t--LBIT %d\ndir %d_1\ndir %d_2\t--
                                    Variable_%d\n", ir.mem_alloc + 12, ir.mem_alloc, ir.
                                    mem_address, ir.mem_address, ir.arg );
96                      }
97                      else{
98                              init_mem = ir.mem_alloc;
99                              init_address = ir.mem_address;
100                             init_arg = ir.arg;
101                             type = 0;
102                     }
103                     break;
104             case INPUT :
105                     if(init_op != ASSIGN){
106                             fprintf( fasm, "LBIT %d\tdin %d\t-- Input_%d\n", ir.
                                    mem_alloc, ir.mem_address, ir.arg );
107                             //fprintf( fhex, " %02X\t\t--LBIT %d\ndir %d\t-- Variable_%d\
                                    n", ir.mem_alloc + 12, ir.mem_alloc, ir.mem_address,
                                    ir.arg );
108                             // fprintf( fasm, "LBIT %d\tdir %d_1 dir %d_2\t-- Variable_%d
                                    \n", ir.mem_alloc, ir.mem_address, ir.mem_address, ir.
                                    arg );
109                             // fprintf( fhex, " %02X\t\t--LBIT %d\ndir %d_1\ndir %d_2\t--
                                    Variable_%d\n", ir.mem_alloc + 12, ir.mem_alloc, ir.
                                    mem_address, ir.mem_address, ir.arg );
110                     }
111                     else{
112                             init_mem = ir.mem_alloc;
113                             init_address = ir.mem_address;
114                             init_arg = ir.arg;
115                             type = 1;
116                     }
117                     break;
118             case OUTPUT :
119                     if(init_op != ASSIGN){
120                             fprintf( fasm, "LBIT %d\tdout %d\t-- Output_%d\n", ir.
                                    mem_alloc, ir.mem_address, ir.arg );
121                             //fprintf( fhex, " %02X\t\t--LBIT %d\ndir %d\t-- Variable_%d\
                                    n", ir.mem_alloc + 12, ir.mem_alloc, ir.mem_address,
                                    ir.arg );
122                             // fprintf( fasm, "LBIT %d\tdir %d_1 dir %d_2\t-- Variable_%d
                                    \n", ir.mem_alloc, ir.mem_address, ir.mem_address, ir.
```

```
                                        arg );
123                                 // fprintf( fhex, "%02X\t\t--LBIT %d\ndir %d_1\ndir %d_2\t--
                                        Variable_%d\n", ir.mem_alloc + 12, ir.mem_alloc, ir.
                                        mem_address, ir.mem_address, ir.arg );
124                         }
125                         else{
126                                 init_mem = ir.mem_alloc;
127                                 init_address = ir.mem_address;
128                                 init_arg = ir.arg;
129                                 type = 2;
130                         }
131                         break;
132                 case TIMER :
133                         if(init_op != ASSIGN){
134                                 fprintf( fasm, "LBIT %d\tdtim %d\t-- Timer_%d\n", ir.
                                        mem_alloc, ir.mem_address, ir.arg );
135                                 //fprintf( fhex, "%02X\t\t--LBIT %d\ndir %d\t-- Variable_%d\
                                        n", ir.mem_alloc + 12, ir.mem_alloc, ir.mem_address,
                                        ir.arg );
136                                 // fprintf( fasm, "LBIT %d\tdir %d_1 dir %d_2\t-- Variable_%d
                                        \n", ir.mem_alloc, ir.mem_address, ir.mem_address, ir.
                                        arg );
137                                 // fprintf( fhex, "%02X\t\t--LBIT %d\ndir %d_1\ndir %d_2\t--
                                        Variable_%d\n", ir.mem_alloc + 12, ir.mem_alloc, ir.
                                        mem_address, ir.mem_address, ir.arg );
138                         }
139                         else{
140                                 init_mem = ir.mem_alloc;
141                                 init_address = ir.mem_address;
142                                 init_arg = ir.arg;
143                                 type = 3;
144                         }
145                         break;
146                 case AND :
147                         fprintf( fasm, "AND\t\t\t\t-- AND\n" );
148                         fprintf( fhex, "02\t\t--AND\n" );
149                         break;
150                 case OR :
151                         fprintf( fasm, "OR\t\t\t\t-- OR\n" );
152                         fprintf( fhex, "01\t\t--OR\n" );
153                         break;
154                 case NOT :
155                         fprintf( fasm, "NOT\t\t\t\t-- NOT\n" );
156                         fprintf( fhex, "03\t\t--NOT\n" );
```

```
                                break;
                }
                init_op = ir.op;
        }
        while (ir.op != END_PROGRAM);
        int i;
        for(i = 0; i < push_set; ++i)
                fprintf( fasm, "PUSHSET\n" );
        fprintf(fasm, "FIN\n" );
        //fprintf( fhex, "33\n33\n33\n33\n33\n33\n33\n33\n33\n33\n33\n33\nFF\n" );
}

/************************** End Stack Machine *************************/
```

Escribe el siguiente código en un fichero llamado `ST.h`:

```
/***************************************************************************
                        Symbol Table Module
***************************************************************************/
/*=========================================================================
                                DECLARATIONS
=========================================================================*/
/*-------------------------------------------------------------------------
                        SYMBOL TABLE RECORD
-------------------------------------------------------------------------*/
struct symrec
{
        char *name; /* name of symbol */
        int offset; /* data offset */
        struct symrec *next; /* link field */
};

typedef struct symrec symrec;

/*-------------------------------------------------------------------------
                        SYMBOL TABLE ENTRY
-------------------------------------------------------------------------*/
symrec *identifier, *id_timer, *id_in, *id_out;

/*-------------------------------------------------------------------------
                        SYMBOL TABLE
                Implementation: a chain of records.
------------------------------------------------------------------------*/
```

```c
symrec *sym_table = (symrec *)0; /* The pointer to the Symbol Table */
symrec *tim_table = (symrec *)0; /* The pointer to the Symbol Table */
symrec *in_table = (symrec *)0; /* The pointer to the Symbol Table */
symrec *out_table = (symrec *)0; /* The pointer to the Symbol Table */

/*========================================================================
                        Operations: Putsym, Getsym
========================================================================*/
symrec * putsym (char *sym_name)
{
        symrec *ptr;
        ptr = (symrec *) malloc (sizeof(symrec));
        ptr->name = (char *) malloc (strlen(sym_name)+1);
        strcpy (ptr->name,sym_name);
        ptr->offset = data_location(0);
        ptr->next = (struct symrec *)sym_table;
        sym_table = ptr;
        return ptr;
}

symrec * getsym (char *sym_name)
{
        symrec *ptr;
        for ( ptr = sym_table; ptr != (symrec *) 0; ptr = (symrec *)ptr->next )
                if (strcmp (ptr->name,sym_name) == 0)
                        return ptr;
        return 0;
}

/*========================================================================
                        Operations: Putsym, Getsym
========================================================================*/
symrec * puttim (char *sym_name)
{
        symrec *ptr;
        ptr = (symrec *) malloc (sizeof(symrec));
        ptr->name = (char *) malloc (strlen(sym_name)+1);
        strcpy (ptr->name,sym_name);
        ptr->offset = data_location(1);
        ptr->next = (struct symrec *)tim_table;
        tim_table = ptr;
        return ptr;
}

```

```
72  symrec * gettim (char *sym_name)
73  {
74          symrec *ptr;
75          for ( ptr = tim_table; ptr != (symrec *) 0; ptr = (symrec *)ptr->next )
76                  if (strcmp (ptr->name,sym_name) == 0)
77                          return ptr;
78          return 0;
79  }
80
81  /*=========================================================================
82                          Operations: Putsym, Getsym
83  =========================================================================*/
84  symrec * putin (char *sym_name)
85  {
86          symrec *ptr;
87          ptr = (symrec *) malloc (sizeof(symrec));
88          ptr->name = (char *) malloc (strlen(sym_name)+1);
89          strcpy (ptr->name,sym_name);
90          ptr->offset = data_location(2);
91          ptr->next = (struct symrec *)in_table;
92          in_table = ptr;
93          return ptr;
94  }
95
96  symrec * getin (char *sym_name)
97  {
98          symrec *ptr;
99          for ( ptr = in_table; ptr != (symrec *) 0; ptr = (symrec *)ptr->next )
100                 if (strcmp (ptr->name,sym_name) == 0)
101                         return ptr;
102         return 0;
103 }
104
105 /*=========================================================================
106                         Operations: Putsym, Getsym
107 =========================================================================*/
108 symrec * putout (char *sym_name)
109 {
110         symrec *ptr;
111         ptr = (symrec *) malloc (sizeof(symrec));
112         ptr->name = (char *) malloc (strlen(sym_name)+1);
113         strcpy (ptr->name,sym_name);
114         ptr->offset = data_location(3);
115         ptr->next = (struct symrec *)out_table;
```

```
116         out_table = ptr;
117         return ptr;
118 }
119
120 symrec * getout (char *sym_name)
121 {
122         symrec *ptr;
123         for ( ptr = out_table; ptr != (symrec *) 0; ptr = (symrec *)ptr->next )
124                 if (strcmp (ptr->name,sym_name) == 0)
125                         return ptr;
126         return 0;
127 }
128
129 /************************* End Symbol Table *************************/
```

Compilación en Linux desde la carpeta raíz usando `make`:

```
~/uaqc$ make
```

Compilación en Windows desde la carpeta raíz usando `make`:

```
\uaqc> mingw32-make
```

Apéndice C

Artículo

En éste capítulo se muestra un artículo derivado de ésta investigación titulado “Compilador para microprocesador basado en FPGA en aplicaciones industriales”, *“Compiler for FPGA based microprocessor in industrial application”*. Dicho artículo fue publicado en “La Ingeniería y sus áreas afines 1”, Memorias del IX Congreso Internacional de Ingeniería, Primera edición, abril de 2013. ISBN: 978-607-513-050-7.

COMPILADOR PARA MICROPROCESADOR BASADO EN FPGA EN APLICACIONES INDUSTRIALES

COMPILER FOR FPGA BASED MICROPROCESSOR IN INDUSTRIAL APPLICATION

Jesús Iván Sánchez-Gómez [1], J. Jesús de Santiago-Pérez [1]
Arturo García-Pérez [2], Juan Ángel Ramírez-Núñez [2]
[1] Universidad Autónoma de Querétaro, Campus San Juan del Río.
San Juan del Río, Querétaro. México.
[2] Universidad de Guanajuato.
Salamanca, Guanajuato. México.
jesus.sangomez@gmail.com, jjdesantiago@hspdigital.org,
agarcia@hspdigital.org, juan.angel.rn@gmail.com

Resumen. La evolución de la informática y los lenguajes de programación han provocado que el desarrollo de software crezca rápidamente. Cada vez se requieren más herramientas sofisticadas para mejorar la producción en la industria y con esto satisfacer las necesidades que los clientes demanden. El presente artículo muestra el desarrollo de un compilador para microprocesadores basados en FPGA, enfocado principalmente a un controlador lógico programable, la estructura de dicho compilador hará que sea fácilmente actualizable, escalable y portable a futuro, esto ofrece una gran ventaja comparado con software comerciales ya que, estos últimos quedan obsoletos cuando los controladores son mejorados, por lo que se genera también una independencia tecnológica. El compilador está enfocado principalmente a la arquitectura de un Controlador Lógico Programable y a la traducción de diagramas escalera, representados por ecuaciones booleanas, de un proceso genérico que puede presentarse en cualquier máquina. Este trabajo pretende ofrecer una herramienta más para la automatización de procesos industriales.

Palabras Clave: FPGA, PLC, Lenguaje Escalera, Compilador.

Abstract. The evolution of computer science and programming languages has led to the software development rapidly grows. Increasingly sophisticated tools needed to improve production in this industry and meet the needs that customers will demand. This paper presents the development of a compiler for FPGA-based microprocessors, focusing mainly on a programmable logic controller, the structure of the compiler will be easily upgradeable, scalable and portable to future, this offers a great advantage compared to commercial software, as these become obsolete when controllers are improved, so it also generates a technological independence. The compiler is mainly focused on the architecture of a Programmable Logic Controller and translation of ladder diagrams, represented by Boolean equations of a generic process that may occur in any machine. This work aims to provide an additional tool for industrial process automation.

Keywords: FPGA, PLC, Ladder Language, Compiler.

1 Introducción

La demanda de los clientes es cada día más exigente, por lo que para los fabricantes crece la necesidad de mejorar su productividad, al igual que ofrecer mejor calidad en sus productos. Debido a esto la automatización de procesos industriales es de las principales prioridades para las empresas fabricantes, por lo cual el mercado industrial a nivel global se vuelve más competitivo y las empresas deben mejorar su tecnología para sobresalir en dicho mercado global. De igual manera un sistema automatizado ofrece diversas ventajas, entre las cuales destacan el control de los procesos, que se encarga de corregir todas las perturbaciones externas e internas que puedan surgir durante el proceso respecto a las variables y condiciones iniciales, y el monitoreo de los procesos, esto simplemente para recabar información acerca del proceso y evaluar si dicho proceso requiere de algún reajuste o si trabaja en sus condiciones más óptimas.

1.1 Dependencia Tecnológica

Entre los problemas más importantes que se tienen en la industria es la falta de desarrollo tecnológico propio, la gran mayoría de las empresas dependen de aquellas que generan tecnología. Por lo que cuando dicha tecnología es retirada o descontinuada provoca severos problemas, más aun cuando todo un sistema o proceso se basó en la misma. Uno de los primeros efectos que refleja la ausencia de dicha tecnología es el económico ya que normalmente requiere de la reestructuración de un proceso y de la compra de nuevo equipo que cumpla nuevamente con las necesidades que se tenían anteriormente.

1.2 Field Programmable Gate Array (FPGA)

Una de las principales razones por las que en la industria se utilizan los FPGA es debido a que son reprogramables o reconfigurable, esto hace que los costos se reduzcan ya que se puede utilizar el mismo Hardware para distintos procesos cambiando exclusivamente su programación interna. Actualmente el FPGA se utiliza en diversas aplicaciones y en distintos campos, como son medicina (Dillinger, 2006), transmisión de audio y video (Bourbakis, 2003), (Ha, 2004), comunicaciones (Alachiotis, 2010), (Wang, 2008), en la industria aeroespacial (Fay, 2007), (Heanut, 2009), robótica y maquinaria CNC (Trejo, 2010), (Osornio, 2007), etc., por lo que se nota su flexibilidad y portabilidad a diferentes procesos. De igual manera otra de las ventajas que ofrecen es su alto rendimiento aprovechando el paralelismo del Hardware lo que lo hace más veloz.

1.3 Hardware/Software

En la informática y automatización la interacción entre el Hardware y el Software es muy importante ya que no se puede tener Hardware funcional sin su respectivo Software, o viceversa. Debido a esto tanto el Hardware como el Software han evolucionado ofreciendo en el caso del Hardware un mejor funcionamiento, mayor capacidad de memoria y un mejor diseño y estructura de sus componentes, en el caso de Software las aplicaciones y sistemas son más fáciles de utilizar, tienen un diseño más amigable y son más eficaces. Este artículo presenta los resultados preliminares de un proyecto que incluye el diseño de un microprocesador, cuyo caso de estudio va enfocado a un controlador lógico programable, y la herramienta que se encargará de facilitar la

programación del mismo. Estos primeros resultados muestran cómo es posible la descripción del código de un diagrama escalera por parte del usuario y la traducción que hace el compilador para devolver el código final que será cargado al microprocesador.

2 Caso de Estudio: Controlador Lógico Programable (PLC)

Esta investigación se enfoca principalmente en el desarrollo de un compilador que sea capaz de hacer más rápida y sencilla la programación de un controlador lógico programable (PLC) basado en FPGA. Dicho PLC está basado en un FPGA de bajo costo y alta capacidad. EL sistema entero del controlador se diseñó para que la ejecución del diagrama escalera sea realizada por el Hardware y la interfaz con el usuario sea por el puerto RS232 o USB por medio de una computadora personal. El microprocesador incluido en el PLC se enfoca inicialmente para el funcionamiento del mismo, aunque puede ser portado a otro tipo de plataformas similares.

3 Descripción de Hardware

Para ésta investigación se utilizó la tarjeta PLCUAQ816 (Figura 3 .1) desarrollada en la Universidad Autónoma de Querétaro, la cual se basa en el FPGA Spartan3E XC3S200 y cuenta con las siguientes características.

- Capa Física
  - Controlador lógico programable
  - 8 salidas analógicas de 10V
  - 8 entradas analógicas de 10V
  - 16 salidas digitales TTL
  - 16 entradas digitales TTL
  - 3 DIP switch
  - Puerto RS232
  - Puerto USB
  - Alimentaciones requeridas 5V, 12V y -12V
  - Memoria serial de 32KB
  - Memoria estática de 512KB
  - Memoria dinámica de 4Mb
  - FPGA Spartan 3E XC3S200 de 200,000 compuertas
  - Configuración vía JTAG o de memoria flash
  - Oscilador de 50 MHz.
- Capa Lógica
  - Microprocesador PLC embebido
  - Manejadores de E/S digitales
  - Controladores de ADC y DAC
  - Temporizadores programables
  - Reloj de tiempo real
  - Controladores de memorias
  - Controladores de puertos RS232 y USB

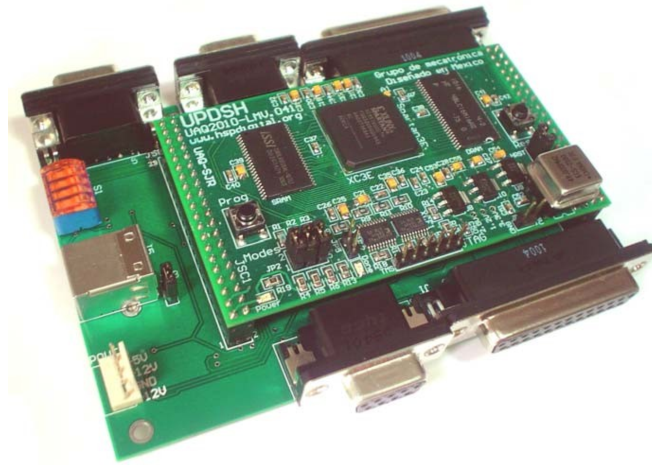

Figura 3.1 Tarjeta PLCUAQ816

El esquema general del microprocesador se compone de los bloques mostrados en la Figura 3.2.

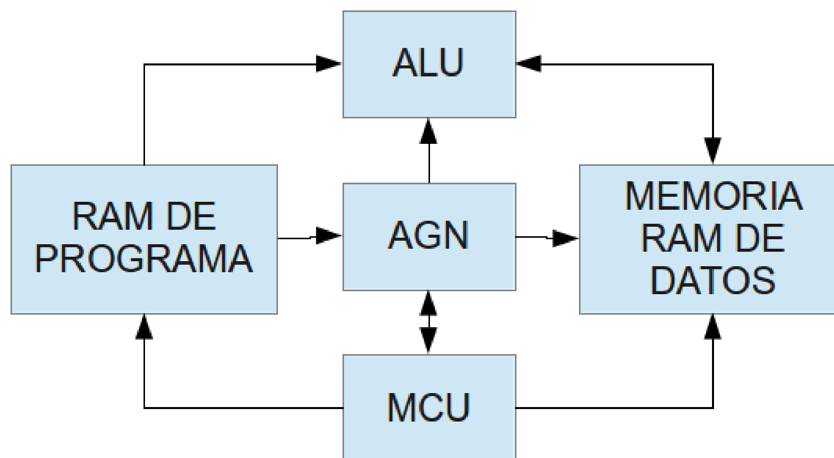

Figura 3.2 Bloques del Microprocesador

RAM de Programa. En esta se almacena el programa, debido a que el microprocesador es de 8 bits, la memoria RAM debe almacenar datos en 8 bits, para la dirección se utilizan 10 bits y se tienen 1024 localidades.

Unidad Aritmética Lógica (ALU). Realiza las operaciones lógicas y aritméticas necesarias para el análisis de datos en la interpretación de señales y datos para ejecutar la rutina de un diagrama escalera.

Unidad de Control de Microprograma (MCU). Es la unidad encargada de decodificar las instrucciones que se encuentran codificadas en el programa. Habilita o deshabilita las memorias, el generador de direcciones (AGN), y las entradas y salidas de los periféricos.

Memoria RAM de Datos (Datos Dual RAM). Esta memoria almacena los datos que se van adquiriendo en la ejecución del programa, la habilitación de escritura o lectura son manejadas por la unidad de control de microprograma.

Generador de Direcciones (AGN). Es la unidad encargada de generar las direcciones para manejar las memorias, tanto la de programa como la de datos.

Por otro lado el microprocesador cuenta también con módulos para realizar la comunicación con el exterior por medio de periféricos mostrados en la Figura 3 .3.

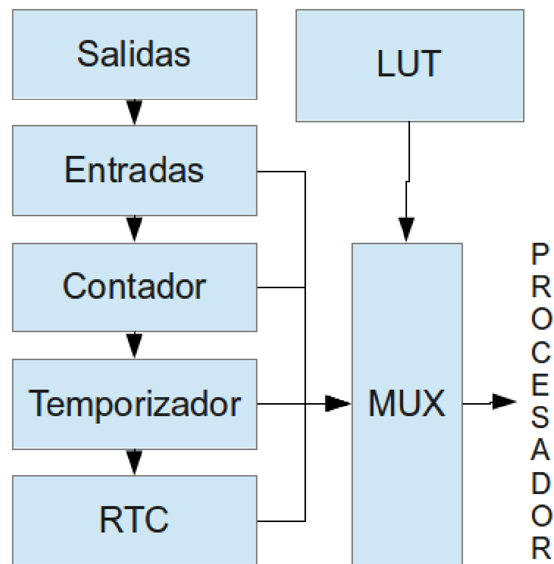

Figura 3.3 Comunicación Externa con el Microprocesador

Salidas. Es la encargada de la comunicación de los periféricos de salida del microprocesador con las tarjetas de interfaz para el control del proceso.

Entradas. Introduce las señales del proceso hacia el microprocesador. Se comunica con tarjetas de interfaz de entradas, encargadas de monitorear el proceso y sus variables.

Contadores. Se encarga de contar eventos, debe poder ser programable. Es controlado por el microprocesador.

Temporizadores. Es un contador de tiempo real, necesario para procesos que necesiten ejecutarse en un tiempo determinado, de igual manera es programable y controlado por el microprocesador.

Reloj en Tiempo Real (RTC). Se trata de un reloj de tiempo real, el cual permite almacenar la hora y la fecha actuales, para utilizar estos valores durante el proceso en caso de que se necesite realizar un evento en cierta fecha específica a cierta hora del día.

LUT. Para manejar los periféricos, los contadores y los temporizadores se utiliza una tabla de consulta que es controlada por la unidad de control de microprograma y por el generador de direcciones.

4 Descripción de Software

Como es común la programación de un controlador lógico programable comercial se realiza por medio del lenguaje escalera, por lo que la programación del microprocesador PLC, en el que se basa esta investigación, se realiza siguiendo la misma línea de los controladores comerciales. La lógica del PLC será descrita por medio de un lenguaje escalera representado por ecuaciones booleanas en notación infija. Esto es, por ejemplo el diagrama mostrado en la Figura 4 .4 será programado como muestra la Ec. 4 .1.

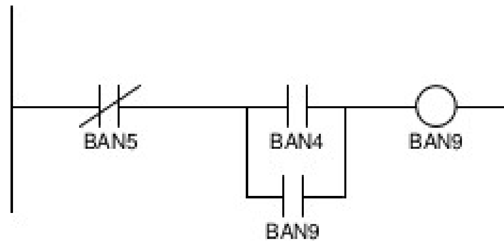

Figura 4.4 Ejemplo de Diagrama Escalera

$$\text{BAN9} = (\textit{NOT}\ \text{BAN5})\ \textit{AND}\ (\text{BAN4}\ \textit{OR}\ \text{BAN9}) \qquad \textbf{Ec. 4.1}$$

Por otro lado el microprocesador trabaja con una notación postfija ya que se basa en una pila para el almacenamiento de las rutinas. Esto ofrece la ventaja de que los cálculos se realizan secuencialmente según se introducen los operadores y no se tiene que esperar a escribir la operación completa. Por lo que es necesario un compilador que traduzca de las ecuaciones en notación infija que el usuario describirá, a las operaciones en notación postfija que el microprocesador interpretará, como se muestra en la Ec. 4 .2.

$$\text{BAN5}\ \textit{NOT}\ \text{BAN4}\ \text{BAN9}\ \textit{OR}\ \textit{AND}\ \text{BAN9} = \qquad \textbf{Ec. 4.2}$$

El compilador sigue la metodología clásica de los compiladores, mostrada en la Figura 4 .5.

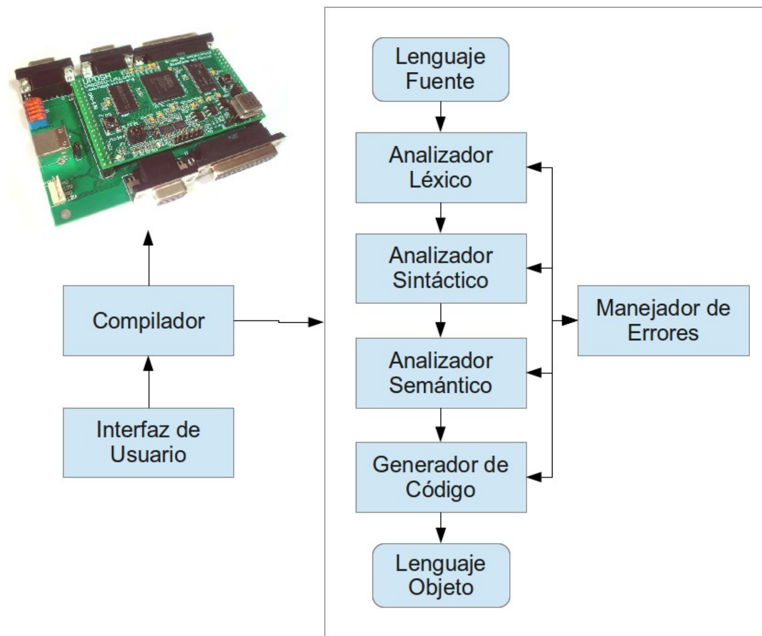

Figura 4.5 Metodología del Compilador

Analizador Léxico. También denominado *scanner*. Su función consiste básicamente en agrupar los caracteres del texto fuente en grupos de entidad propia denominados *tokens* (signos lingüísticos). Ejemplos de *tokens* son los identificadores, palabras reservadas, separadores, etc. Los *tokens* reconocidos son la entrada a la siguiente fase, el analizador sintáctico.

Analizador Sintáctico. Se ocupa para analizar la sintaxis de las sentencias (compuestas de *tokens*), de acuerdo con la descripción sintáctica reflejada en la gramática.

Analizador Semántico. Se ocupa de analizar la semántica de las sentencias, realizando una serie de consultas en unas tablas auxiliares denominadas tablas de símbolos.

Generador de Código. Se ocupa de generar código objeto para una máquina, es decir, donde efectivamente se hace la traducción.

Tratamiento de Errores. Es el conjunto de rutinas y actividades que tratan la identificación de un error, su posible tratamiento o recuperación y emisión del mensaje correspondiente.

El microprocesador puede programarse de manera manual por el usuario, interpretando y describiendo los diagramas escalera en ecuaciones en notación postfija y posteriormente traducir estas ecuaciones a código VHDL para introducirlo dentro de la estructura del microprocesador basado en FPGA. Esto presenta varias desventajas, ya que sería una tarea tediosa para el usuario, además tomaría bastante tiempo la programación del PLC en procesos demasiado complejos, de igual manera es más susceptible a errores humanos.

El compilador es capaz de ahorrar tiempo y evitar errores del usuario al momento de traducir el código fuente ya que el usuario se encarga de escribir cada una de las ecuaciones en notación

infija del diagrama escalera que requiera programar en el controlador para posteriormente utilizar el compilador que hará el análisis y la traducción de dichas ecuaciones para que el controlador las interpreté, en caso de que el usuario haya cometido algún error de sintaxis el compilador resaltará dichos errores. Una vez que el código sea correcto el compilador se encargará de traducir el código de manera automática, devolviendo finalmente el archivo que será cargado al controlador.

5 Resultados y Conclusión

5.1 Recursos del Microprocesador

La implementación se llevó a cabo, como se dijo anteriormente en un FPGA XC3S200 de 200000 compuertas, utilizando el programa de síntesis Xilinx ISE 8.1i que entrego el resumen mostrado en la Tabla 5 .1 de los porcentajes de los recursos totales utilizados por el dispositivo FPGA XC3S200.

Recursos	Requeridos	Disponibles	Usados
Slices	1,657	1,920	86 %
LUT's	2,564	3,840	66 %
Flip-Flops	927	3,840	24 %
I/O	82	173	47 %

Tabla 5.1 Recursos Utilizados por el FPGA XC3S200

5.2 Aplicación

Para comprobar la funcionalidad del compilador, se plantea la siguiente prueba de un proceso genérico que se puede presentar en cualquier máquina. Dicho proceso consiste en que al ser pulsado un botón se activan tres dispositivos (motores) de manera secuencial y se mantienen ejecutados de forma cíclica, esto es, cada dispositivo se mantiene encendido cierto tiempo y se pasa al siguiente. Para representar dicho proceso se propone el diagrama de la Figura 5 .6 en el que se utilizan tres temporizadores (TIM0, TIM1, TIM2) para mantener encendidos los tres dispositivos (BAN9, BAN10, BAN11). De igual manera se incluye el botón de inicio (BAN4) y un botón de paro de emergencia (BAN5).

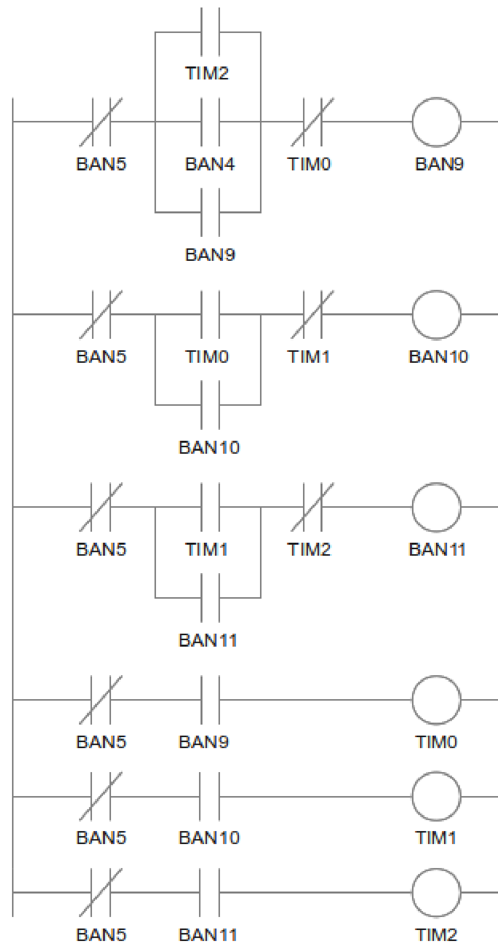

Figura 5.6 Diagrama Escalera de la Prueba Realizada

La Figura 5 .7 muestra las ecuaciones en notación infija del diagrama en escalera de la Figura 5 .6. Dicho código contiene intencionalmente diversos errores de sintaxis (marcados con círculos rojos), para que el compilador, al momento de analizarlo, muestre los errores que contenga el código.

```
PLC.XPP  ×

1   BAN9  = /BAN5 * (BAN4 + BAN9 + TIM2) * /TIM0
2   BAN10 = /BAN5 * ((TIM0 + BAN10) * /TIM1
3   BAN11 = /BAN5 * (TIM1 + BAN11/) * /TIM2
4   TIM0  = /BAN5 ** BAN9
5   TIM1  = /BAN5 * BAN10
6   TIM2  = /BAN5 *+ BAN11
7
8   :END
```

Figura 5.7 Código Fuente

Los operadores que se utilizan en el código representan lo siguiente:

- El operador / representa el operador lógico NOT.
- El operador + representa el operador lógico OR.
- El operador * representa el operador lógico AND.
- El operador = representa el operador de asignación.

La Figura 5 .8 muestra el análisis que realizó el compilador sobre el código anterior, como se observa, el compilador detecto correctamente los errores y se muestran definiendo primeramente el nombre del archivo, seguido de la línea en la que se encuentra el error, así como también el tipo de error con una breve descripción del mismo y una posible solución.

```
PLC.XPP:3: syntax error, unexpected END, expecting
 EQ or OR or AND or RIGHT_PARENTHESIS
PLC.XPP:3: syntax error, unexpected NOT, expecting
 EQ or OR or AND or RIGHT_PARENTHESIS
PLC.XPP:4: syntax error, unexpected AND, expecting
 ID or NOT or LEFT_PARENTHESIS
PLC.XPP:6: syntax error, unexpected OR, expecting
ID or NOT or LEFT_PARENTHESIS
```

Figura 5.8 Análisis Entregado por el Compilador

Una vez corregidos los errores se genera la traducción de las ecuaciones en notación infija a las ecuaciones en notación postfija que es el código intermedio que genera el compilador anterior al código final que interpretará el controlador lógico programable, como se muestra en la Figura 5 . 9.

```
Output ×
1  BAN5 / BAN4 BAN9 + TIM2 + * TIM0 / * BAN9 =
2  BAN5 / TIM0 BAN10 + * TIM1 / * BAN10 =
3  BAN5 / TIM1 BAN11 + * TIM2 / * BAN11 =
4  BAN5 / BAN9 * TIM0 =
5  BAN5 / BAN10 * TIM1 =
6  BAN5 / BAN11 * TIM2 =
7
```

Figura 5.9 Código Intermedio Generado por el Compilador

Finalmente el compilador analizará el código intermedio que se generó y lo traducirá al código final que será interpretado por el compilador. Debido a que el código final es más extenso la Figura 5 .10 muestra una parte del código final correspondiente a la primera línea del diagrama escalera utilizado resaltando las banderas y los *timers*, en estos resultados se omiten los valores de configuración e inicialización.

```
024 LBIT0
025 0HX00
026 0HX2D -- BAN5
027 NOT
028 LBIT3
029 0HX00
02A 0HX2D -- BAN4
02B LBIT1
02C 0HX00
02D 0HX81 -- BAN9
02E OR
02F 0HX00
030 0HX0C
031 LBIT1 -- TIM2
032 OR
033 AND
034 LBIT1
035 0HX00
036 0HX04 -- TIM0
037 NOT
038 AND
```

Figura 5.10 Código Final Generado por el Compilador

6 Conclusión

En la actualidad es de suma importancia el desarrollo de Hardware sin dejar atrás el Software que lo complemente, o viceversa, de esta manera se tendrán herramientas (sistemas embebidos) más completas para la automatización de procesos industriales, en este caso aquellos que requieran de un PLC. Debe recalcarse la importancia que tiene el microprocesador ya que al ser diseñado en una plataforma reconfigurable como es el FPGA ofrece la ventaja de ser portado a procesos diferentes y de mantenerse al día con los avances tecnológicos sin dejar de ser obsoleto. Por el lado de la aplicación, se tiene claro que se necesitan herramientas que faciliten el manejo de la maquinaria, en este caso se hizo más fácil y rápida la programación del controlador lógico programable, además de poder recalcar los errores que podría tener el operador al momento de realizar su proceso. De igual manera al ser utilizada una programación estructurada a la hora de desarrollar el compilador se tiene la ventaja de ser escalable y fácilmente actualizable en caso de que el PLC sea modificado, lo que favorece enormemente ya que al igual que el microprocesador, se genera una independencia tecnológica con lo que finalmente se obtiene una completa herramienta para la automatización de procesos industriales.

REFERENCIAS

D. Fay, S. Campbell, G. Miller, y D. Connors. *Teaching fault tolerant fpga design for aerospace applications*. Microelectronic Systems Education, 2007. MSE ’07. IEEE International Conference, Junio 2007, pp. 61–62.

J. Henaut, D. Dragomirescu, y R. Plana. *Fpga based high date rate radio interfaces for aerospace wireless sensor systems*. Systems, 2009. ICONS '09. Fourth International Conference, Marzo 2009, pp. 173–178.

J. Wang, H. Wang, y Z. jia Yang, *An fpga based slave communication controller for industrial Ethernet.* Solid-State and Integrated-Circuit Technology, 2008. ICSICT 2008. 9th International Conference, Octubre 2008, pp. 2062–2065.

M. Trejo-Hernandez, R. A. Osornio-Rios, R. J. Romero-Troncoso, C. Rodriguez-Donate, A. Dominguez-Gonzalez, y G. Herrera-Ruiz, *Fpga-based fused smart-sensor for tool-wear area quantitative estimation in cnc machine inserts.* Sensors, MDPI, vol. 10, no. 4, pp. 3373–3388, 2010, ISSN 1424-8220.

N. Alachiotis, S. Berger, y A. Stamatakis, *Efficient pc-fpga communication over gigabit Ethernet*. Computer and Information Technology (CIT), 2010 IEEE 10th International Conference, Junio-Julio 2010, pp. 1727–1734.

N. Bourbakis y A. Dollas, *Scan-based compression-encryption-hiding for video on demand*. MultiMedia, IEEE, vol. 10, pp. 79–87, Julio-Septiembre 2003, ISSN 1070-986X.

P. Dillinger, J. Vogelbruch, J. Leinen, S. Suslov, R. Patzak, H. Winkler, y K. Schwan, *Fpga-based real-time image segmentation for medical systems and data processing*. Nuclear Science, IEEE Transactions on, vol. 53, pp. 2097–2101, Agosto 2006, ISSN 0018-9499.

R. A. Osornio-Rios, R. J. Romero-Troncoso, G. Herrera-Ruiz, y R. Castaneda-Miranda, *Computationally efficient parametric analysis of discrete-time polynomial based acceleration-deceleration profile generation for industrial robotics and cnc machinery*. Mechatronics, Elsevier, vol. 17, pp. 511–523, 2007, ISSN 0957-4158.

V. Ha, C. Sung-Kyu, J. Jong-Gu, L. Geon-Hyoung, J. Won-Kap, y S. Woo-Sung, *Fpga-based real-time image segmentation for medical systems and data processing*. System-on-Chip for Real-Time Applications, 2004.Proceedings. 4th IEEE International Workshop, Julio 2004, pp.162–167.