

VISIÓN INVESTIGADORA
Clasificación del artículo:
INVETIGACIÓN

DISEÑO E IMPLEMENTACIÓN DEL COMPILADOR STL PARA EL PLC-UD

Daniel Vargas Vela*
Fernando Martínez Santa**

Resumen

En este artículo se describe el proceso de diseño e implementación del compilador STL para el PLC-UD, el cual está encargado de generar código ejecutable para los PLC, a partir del lenguaje estructurado STL. El trabajo consta de dos partes fundamentales, la gestión del hardware y el proceso de compilación. Para la primera parte se definen las rutinas en lenguaje máquina necesarias en la compilación, y para la segunda parte se muestran las técnicas de compilación utilizadas y la forma como se implantó el compilador, utilizando el lenguaje C++.

Palabras clave
Compilador, PLC, STL, Generación de código.

Summary
In this paper is described the design and implementation process of the STL compiler for the PLC-UD, the which is in charge of generating executable code for the PLCs, starting from the structured language STL. The work consists of two fundamental parts, the administration of the hardware and the compilation process. For the first part, is defined the routines in object language necessary in the compilation, and for second part it shows the used compilation techniques, and the way in that the compiler was implanted using the C++ language.

Key words
Compiler, PLC, STL, Code generator.

* Ingeniero eléctrico de la Universidad de los Andes, Msc. en Ingeniería Electrónica y de computadores de la Universidad de los Andes. Fundador del grupo de Investigación PLC-UD. Correo electrónico: dvargas00@yahoo.com

** Tecnólogo en electrónica e Ingeniero en Control Electrónico e Instrumentación de la Universidad Distrital Francisco José de Caldas. Integrante del grupo de Investigación PLC-UD. Correo electrónico: fmartinezsanta@hotmail.com

1. Introducción

El proyecto de investigación Controlador lógico programable de la Universidad Distrital (PLC-UD) pretende apropiar, asimilar y generar el conocimiento necesario para crear una serie de PLC que tengan prestaciones similares a las tecnologías extranjeras a un menor costo, con el fin de ofrecer una alternativa a las pequeñas y medianas empresas interesadas en incorporar procesos de automatización industrial, o aumentar y mejorar procesos ya establecidos [1]. Es entonces cuando surge la necesidad de implementar un software de programación para estos PLC que cumpla con sus requerimientos mínimos y que sea fácil de manipular por parte del usuario final.

Para implementar dicho software es necesario definir la estructura gramatical del lenguaje. En este caso,

el lenguaje STL para el PLC-UD se definió tomando como base los criterios de la norma estándar IEC 61131 que cubre los lenguajes de programación para PLCs [2].

2. Lenguaje STL para el PLC-UD

La estructura general de un programa en STL se presenta a continuación:

```
pragma id
…
pragma id      (*comentarios*)

uses id, ….. id;

program id;

bloque_de_declaraciones_devariables
bloque_de_declaraciones_de_constantes
bloque_de_funciones_y_procedimientos

begin
bloque_de_declaraciones_de_variables
bloque_de_declaraciones_de_ constantes

listado_de_instrucciones

end.
```

Como se puede observar, el programa inicia con la directiva "pragma" utilizada para dar información al compilador –como por ejemplo la gama del PLC, o el núcleo utilizado–, sin embargo, esta directiva es opcional. La instrucción "uses" le indica al compilador que incluya los archivos de cabecera indicados por un identificador asociado. Éste se utiliza para incluir librerías de funciones o de módulos del PLC; en caso de incluir más de un archivo se separan los identificadores con comas (esto también es opcional).

Después se inicia el programa principal, con la palabra reservada "program" y un identificador asociado, seguido del símbolo ";". Seguidamente, se procede a declarar las variables, las constantes y, por último, las funciones o procedimientos. Todos estos bloques de declaraciones son opcionales, pero se debe respetar el orden indicado, [2], [3].

Una vez realizadas todas las declaraciones requeridas, se inicia la rutina principal propiamente dicha (cuerpo del programa): ésta empieza con la palabra reservada "begin" y continúa con los bloques de declaración de variables y constantes locales. En este caso también son opcionales y al igual que en las globales se debe respetar el orden de declaración (las funciones y los procedimientos también poseen estos bloques de declaración). El programa finaliza con la palabra reservada "end" seguida del símbolo de punto.

Todas las instrucciones en STL finalizan con ";", a menos que se trate de un bloque de instrucciones que empieza por "begin" y finaliza con "end".

2.1 Tipos de datos

Los tipos de datos definidos son: booleanos, enteros, enteros sin signo, enteros cortos, y enteros cortos sin signo. La tabla 1 muestra los diferentes tipos de datos [2] [3].

2.2 Operadores

Existen cuatro tipos de operadores: aritméticos, lógicos, de comparación y de asignación.

1. Los aritméticos son aquellos usados en expresiones matemáticas; estos son: +, -, *, /, % y corresponden respectivamente a suma, resta, multiplicación, división y módulo.

1 Se trata de un lenguaje estructurado de alto nivel con una sintaxis similar a la del PASCAL.

2 Los efectos de relleno de estas cajas facilita su ubicación dentro del proceso de compilación mostrado en el Diagrama 1.

2. Los operadores lógicos son: “and”, “or” y “not” correspondientes a las operaciones lógicas y, o, y negación. Estos se utilizan únicamente para variables de tipo BOOL y en la evaluación de condiciones.

3. Los operadores de comparación son: =, <>, <, >, <=, >= y corresponden a: igual, diferente, menor a, mayor a, menor o igual y mayor o igual. Estos se utilizan para relacionar expresiones en las estructuras condicionales. El resultado de las operaciones de comparación siempre es un valor booleano.

4. Finalmente, el operador de asignación: =, encargado de transferir el contenido de una variable o el resultado de la evaluación de una expresión a otra variable (ubicada a la izquierda del operador).

Tabla 1. Tipos de datos

Tipo	Identificador	No. De Bits
Booleano	BOOL	1
Entero	INT	16
Entero sin signo	UINT	16
Entero corto	SINT	8
Entero corto sin signo	USINT	8

2.3 Sentencias

Existen básicamente dos tipos de sentencias, las de ejecución condicional y los ciclos condicionales. En las sentencias de ejecución condicional están: la sentencia “if-then” y la sentencia “if-then-else”. Los ciclos son estructuras repetitivas condicionales, en los que se encuentran: “while-do”, “repeat-until”, “for-to” y “for-downto”; [2] [3].

En todas las sentencias se utilizan las palabras reservadas “begin” y “end”, cuando hay más de una instrucción ejecutable por condición; en caso contrario, éstas se pueden obviar. Ejemplo:

```
while x <= y do
        x := x + 1;

while x <= y do
begin
        x:= x + 1;
        y:= y + 2;
end
```

2.4 Procedimientos y funciones

Los procedimientos y funciones son subrutinas o subprogramas que pueden ser llamados dentro de un programa principal. Estas dos estructuras permiten recibir variables como parámetros, durante su invocación (únicamente se pueden pasar parámetros por valor), la diferencia radica en que las funciones permiten devolver valores. Las funciones y procedimientos deben ser declaradas antes del cuerpo de la rutina principal; [2], [3].

La sintaxis de los procedimientos es:

```
Procedure nombre ( _parámetros ) ;
begin
    listado_de_instrucciones
end
```

En la que “procedure”, “begin” y “end” son palabras reservadas del lenguaje; “nombre” es el identificador asociado al procedimiento; “_parámetros” es una serie de declaraciones de variables separadas por el operador “,” y “listado_de_instrucciones” son las instrucciones o sentencias que se ejecutan al invocar el procedimiento.

Por su parte, la sintaxis de las funciones es:

```
function nombre ( _parámetros ) tipo ;
begin
    listado_de_instrucciones
    nombre:= valor;
end
```

En la que "tipo" es el tipo de dato que devuelve la función y "listado_de_instrucciones" son las instrucciones o sentencias que se ejecutan al invocar la función. La forma de devolver valores en una función es asignando el valor deseado al identificador de la función.

Ejemplo:

```
nombre:= valor;
```

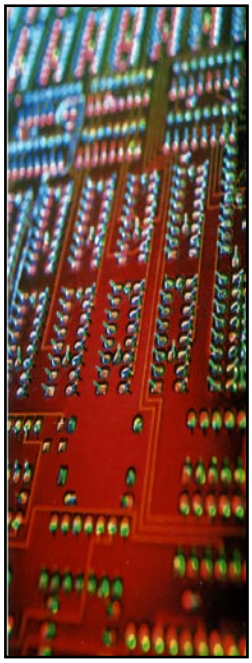
2.5 Declaración variables

Las variables pueden ser globales o locales. En el caso de declararse fuera del campo de declaraciones de la rutina principal o de las funciones y procedimientos, éstas se consideran globales, de no ser así serán locales.

La forma de declarar variables es:

```
id : tipo := valor_inicial ;
```

En la que "id" es el identificador asociado a la variable; "tipo" es el tipo de la variable, y "valor_inicial" es el valor que se le asigna a la variable una vez se le haya reservado memoria (este valor inicial es opcional y en caso de querer obviarlo es necesario obviar también el operador ":=").



En la declaración de variables se debe hacer un bloque, iniciando con la palabra reservada "var" y finalizando el bloque con la palabra "end_var" y el símbolo ";". La única excepción a esta regla es cuando una variable se declara como un parámetro de un procedimiento o una función.

Para las constantes la sintaxis es similar:

```
constant id: tipo:= valor;
```

Con la diferencia de que en este caso el valor no es opcional y para cada constante a declarar se utiliza la palabra reservada "constant".

2.6 Definición formal BNF

Para la definición del lenguaje STL PLC-UD –usando la forma de Backus-Naur– se debe tener en cuenta el tipo de derivaciones (por la derecha o por la izquierda) a utilizar. En este caso se define gramática como tipo LL (derivaciones por la izquierda), para poder realizar un análisis sintáctico descendente.

A continuación se muestra la parte de la definición en BNF del lenguaje STL para el PLC-UD, utilizando una gramática LL, llamada Gstl y de la forma Gstl(Tstl, Nstl, Pstl, Sstl).

T representa el conjunto de todos los símbolos terminales del lenguaje (palabras reservadas, operadores y signos adicionales), N representa el conjunto de todos los símbolos no terminales (variables sintácticas), P representa el conjunto de producciones del lenguaje (definiciones sintácticas) y S el símbolo inicial del lenguaje.

```
Tstl =      { 0-9, +, -, *, /, %, =, <>, <, >, <=, >=, (, ),
,, ., ;, :, :=, #, if, then, else, while ..... , var, end_var,
begin, end .... , program }

Nstl = { NUM, FACT, TERM, EXPR, COMP,
COND, ASIG, PROP .... , PROG }

Pstl ={

FACT          → id | ( EXPR ) | NUM

TERM          → FACT T

T             → * FACT T | / FACT T |
 % FACT T | ε

EXPR          → TERM E | + TERM E |
- TERM E

E             → + TERM E | - TERM E | ε

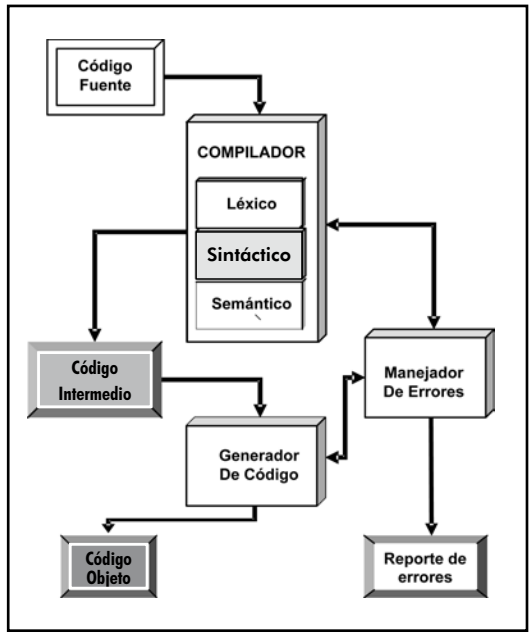
ASIG          → id := EXPR

PROP          → ASIG ;
                | begin PROP P end
                | if COND then PROP Q
                | while COND do PROP
                | repeat PROP until PROP ;
                | for ASIG T_D NUM PROP
.
.
.
PROG          → PRAG INCL program id ; P_F
BLOQUE
}
Sstl          =   { PROG }
```

3. Proceso de compilación

El compilador funciona básicamente recibiendo un archivo de entrada, llamado código fuente y generando un archivo de salida llamado código objeto o código máquina. El código fuente es un archivo escrito en lenguaje STL, el código objeto es un archivo escrito en lenguaje ensamblador correspondiente a un microcontrolador de gama media de la familia Microchip (PIC16F876A) [4][5], que actuará como núcleo del PLC. Dicho proceso de compilación está dividido en varias partes funcionales: los analizadores léxico, sintáctico y semántico, el generador de código y el manejador de errores.

Diagrama 1. Proceso de compilación



Para independizar el compilador del procesador utilizado, con el fin de generar código para diferentes máquinas se dividió la aplicación en dos partes llamadas compilador y generador de código. El compilador recibe como entrada el archivo STL y genera un archivo de salida en un lenguaje intermedio; éste incluye los analizadores léxico, sintáctico y semántico. Por su parte, el generador de código recibe como entrada el archivo en lenguaje intermedio y produce el archivo de salida en lenguaje ensamblador [6][7].

En el compilador la parte de análisis léxico se encarga de recibir un flujo de caracteres (código fuente) y convertirlo en símbolos fácilmente reconocibles y así simplificar el trabajo de las fases posteriores; también está encargado de retirar del código fuente los comentarios y los caracteres innecesarios. Por su parte, el analizador sintáctico

tiene como labor reconocer si las cadenas de símbolos de entrada están correctamente ordenadas en dependencia de la gramática del lenguaje, es decir, si el código fuente está escrito correctamente. El analizador semántico se encarga de interpretar las cadenas de entrada y determinar su significado, es decir, es la parte del compilador que "traduce" el código fuente; para este caso, dicha interpretación se hace por medio del código intermedio.

3.1 Análisis léxico

El proceso de análisis léxico se implementó utilizando una tabla de símbolos que contiene todas las palabras reservadas del lenguaje STL, con algunos otros datos útiles para las siguientes fases de la compilación. El analizador sintáctico también está encargado de eliminar comentarios y caracteres innecesarios tales como: tabuladores, espacios y fines de línea, y además de dejar todo el archivo de entrada en minúsculas (el lenguaje no es case sensitive). Todo esto con el fin de simplificar el proceso de análisis.

La técnica de análisis consiste en una búsqueda lineal sobre la tabla de símbolos, es decir, el archivo de entrada se lee caracter por caracter y se va cargando en un pequeño buffer que es comparado con cada uno de los símbolos de la tabla. Para esto se declara un objeto de la clase SymbolChart y se carga desde un archivo de texto plano con todos los símbolos terminales del lenguaje. A continuación se muestra parte de la tabla de símbolos del compilador:

Al inicio del análisis se genera un archivo de texto de salida, con extensión .sym, en el que se traduce el flujo de símbolos en entrada en un archivo con los indicadores de cada símbolo terminal. Cuando se encuentran coincidencias de la entrada con respecto a la tabla de símbolos, se lee el símbolo (quinta columna) de la tabla y se guarda en el archivo de salida. Al final de este paso se obtiene un archivo de texto con los símbolos correspondientes a cada componente léxico de entrada, el cual es más fácil de interpretar por los analizadores sintáctico y semántico. Cuando el analizador encuentra una palabra no definida en la tabla, éste la incluye al final de la tabla de símbolos, le asigna el siguiente símbolo disponible (índice numérico) y guarda el símbolo en el archivo de salida.

El compilador funciona básicamente recibiendo un archivo de entrada, llamado código fuente y generando un archivo de salida llamado código objeto o código máquina.

Tabla 2. Tabla de símbolos del compilador

0		$ Operator	$ Nothing	0	65535
9		$ Operator	$ Nothing	9	65535
+	ADD	$ Operator	$ Nothing	10	65535
-	SUB	$ Operator	$ Nothing	11	65535
*	MUL	$ Operator	$ Nothing	12	65535
/	DIV	$ Operator	$ Nothing	13	65535
%	MOD	$ Operator	$ Nothing	14	65535
=	EQ	$ Operator	$ Nothing	15	65535
<>	NE	$ Operator	$ Nothing	16	65535
<=	LE	$ Operator	$ Nothing	17	65535
>=	GE	$ Operator	$ Nothing	18	65535

3.2 Análisis sintáctico

Para esta parte del proceso se utilizó el método de análisis descendente recursivo que consiste en partir el análisis desde el símbolo inicial descendiendo por las ramas del árbol de análisis sintáctico hasta las hojas (símbolos no terminales) y así decidir si una cadena de entrada pertenece o no al lenguaje definido [7]. Ejemplo: para reconocer la cadena de entrada "A * B" como perteneciente a un lenguaje con:

```
P = {
FACT    →   id
TERM    →   FACT T
T       →   * FACT T | / FACT T | ε
}
S = {TERM}
```

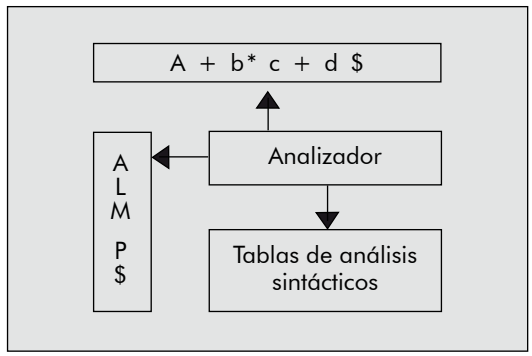
Tendríamos que:

```
TERM      →
FACT T    →
id T      →
id * FACT T →
id * id T →
id * id
```

Teniendo en cuenta que A y B son identificadores, la cadena "A * B" habría sido aceptada por el lenguaje.

La técnica de implantación utilizada fue el análisis sintáctico descendente tabular, que es una forma generalizada de análisis descendente. Este tipo de analizador no es dependiente de la sintaxis del lenguaje, es decir, no está escrito directamente para un lenguaje en específico, sino que toma la información necesaria para la compilación de una matriz de análisis sintáctico [6].

Diagrama 2. Análisis sintáctico tabular



Para el análisis sintáctico tabular se requiere una memoria temporal (buffer) de entrada, una pila para las reducciones, y una tabla de análisis sintáctico. El buffer de entrada contiene la cadena a analizar y en la pila se van cargando cada una de las reducciones hechas por cada producción.

La matriz de análisis sintáctico es una matriz bidimensional que asocia el símbolo terminal a reducir con el símbolo terminal de entrada y tiene como contenido la producción correspondiente a cada pareja Terminal-No_terminal, tal que el analizador sintáctico tenga sólo una posible reducción para cada caso. Esto garantiza que el analizador no caiga en ciclos infinitos de búsqueda o que se tenga que implementar retrocesos en el análisis.

Para completar el contendido de la matriz es necesario conocer los conjuntos PRIMERO(a) y SIGUIENTE(a) de cada uno de los símbolos no terminales, ya que estos hacen referencia a todos los posibles símbolos terminales que pueden aparecer antes y después de cada símbolo no terminal del lenguaje, respectivamente [6]. A continuación se muestra parte de los conjuntos PRIMERO(a) y SIGUIENTE(a) para el lenguaje STL.

Tabla 3. Conjuntos PRIMERO(a) y SIGUIENTE(a)

Símbolos terminales	PRIMERO(a)	SIGUIENTE(a)
D	0..9	),+, -, *, /,%, 0..9
NUM	0..9	),+, -, *, /
N	0..9, ε	),+, -, *, /,%, $
FACT	id, (, 0..9	),+, -, *, /,%
TERM	id, (, 0..9	),+, -
T	*, /,%, ε	),+, -, $
EXPR	+, -, id, (, 0..9	), =, <>, <, >, <=, >=, and, or
COMP	=, <>, <, >, <=, >=	then, do, ;
COND	>=	then, do, ; , $
C	not, +, -, id, (,	; , to, downto
ASIG	0..9	end, $
·	and, or, ε	·
.	id	.
.	.	.
PROG	.	id
	.	
	procedure, function, ε	

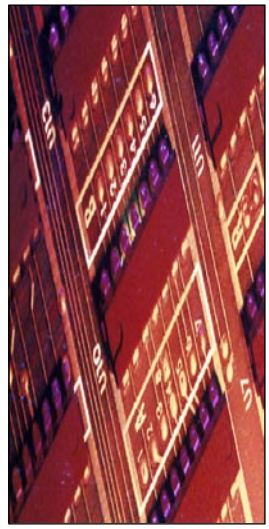
Para implementar la matriz se hizo una matriz de objetos de la clase TextString de un tamaño fijo, correspondiente al número de símbolos terminales (filas) y al número de no terminales (columnas). Esta matriz es cargada desde un archivo de texto plano, que contiene la información de las producciones, y separadores de filas y columnas (caracteres '#' y ' | '). Además, se utilizó una pila de texto (objeto de la clase TextStack) que se inicializa con el símbolo de fin '$' y el símbolo inicial del lenguaje en la cima.

La técnica consiste en direccionar la matriz con el símbolo terminal actual de la cadena de entrada (archivo .sym) y el símbolo no terminal de la cima de la pila. El contenido de la matriz en dicha posición reemplaza el símbolo de la cima de la pila; cuando éste sea un símbolo terminal se compara con el de entrada y si coinciden se saca de la pila. Este proceso se sigue hasta que el símbolo '$' alcanza la cima. En este momento la cadena es aceptada; sin embargo, en caso de llegar al final de la cadena de entrada, y que en la pila se encuentre algún símbolo diferente de $; o si la salida de la matriz es nula, se reporta un error de sintaxis.

3.3 Analizador semántico

El análisis semántico se realiza a la par con el análisis sintáctico y se asocian acciones semánticas a las producciones de la sintaxis; esta técnica se denomina traducción dirigida por la sintaxis, por medio de esquemas de traducción.

Básicamente, las acciones semánticas consisten en el código intermedio asociado a cada producción y algunas operaciones internas, tales como manejo de variables temporales e indicadores. Cuando se produce una reducción de una de las producciones, dicha acción semántica se ejecuta (las acciones semánticas están expresadas como subrutinas). Durante este proceso se utilizan variables temporales, para los cálculos de expresiones de más de un operador; en teoría debe ser un vector infinito de variable temporales, obviamente existen restricciones en cuanto disponibilidad de memoria. Estas variables temporales se declaran (por el compilador) desde la última posición de memoria de datos utilizada por el usuario y su número depende de la cantidad de operadores de la expresión aritmética más larga presente en el código fuente.



3.3.1 Lenguaje intermedio

El lenguaje intermedio PLC UD se define como un lenguaje de cuádruplos o de tres direcciones, es decir, para cada instrucción o línea de código existirán un máximo de cuatro palabras separadas por espacios. La primer palabra indica la instrucción u operador, las siguientes son los nombres de las variables que van a ser los operandos 1 y 2 y el último es el nombre de la variable de resultado [8]. Ejemplo:

Operador	Op1	Op2	Res

Para realizar de una forma más fácil los saltos entre líneas de código se incluye una numeración al inicio de cada línea y es terminada con dos puntos.

Ejemplo:

No.	Operador	Op1	Op2	Res
Línea:				

Op1 y Op2 pueden ser constantes, en este caso se antepone el símbolo # al valor numérico requerido.

Existen instrucciones de tres, dos, uno y ningún parámetro. En las instrucciones de tres parámetros se encuentran las aritméticas, las de comparación y la mayoría de instrucciones lógicas. Dentro de las instrucciones de dos parámetros se encuentran las de movimiento entre variables, de corrimientos y rotaciones, de operaciones con bits y de saltos condicionales. Las instrucciones de definición de variables, de salto incondicional y llamada a procedimiento tienen un único parámetro; en el caso de los dos últimos se trata de una referencia a una línea de programa, y en el de la definición: el nombre de la variable. La instrucción de no operación y la de retorno de procedimiento reciben parámetros.

A continuación se muestra el listado completo de instrucciones del lenguaje intermedio definido para el STL:

Tabla 4. Lenguaje intermedio

Instrucción	Operador	Código intermedio
Suma	+	ADD Op1 Op2 Res
Resta	-	SUB Op1 Op2 Res
Multiplicación	*	MUL Op1 Op2 Res
División	/	DIV Op1 Op2 Res
Módulo	%	MOD Op1 Op2 Res
y	AND	AND Op1 Op2 Res
o	OR	OR Op1 Op2 Res
o Exclusiva	XOR	XOR Op1 Op2 Res
Mayor que	>	GT Op1 Op2 Res
Menor que	<	LT Op1 Op2 Res
Mayor o igual	>=	GE Op1 Op2 Res
Menor o igual	<=	LE Op1 Op2 Res
Igual	=	EQ Op1 Op2 Res
Diferente	<>	NE Op1 Op2 Res
carga indexada		LIDX Dir Ind Res
almacenamiento indexado		SIDX Op1 Dir Ind
Movimiento	:=	MOVE Op1 Res
Valor absoluto		ABS Op1 Res
Corrimiento a la izquierda		SHL Op1 Res
Corrimiento a la derecha		SHR Op1 Res
Rotación a la izquierda		ROL Op1 Res
Rotación a la derecha		ROR Op1 Res
Negación	NOT	NOT Op1 Res
Intercambio		EXCH Op1 Res
Intercambio interno		SWAP Op1 Res
Incremento		INC Op1 Res
Decremento		DEC Op1 Res
Poner en verdadero		SET Op1 Res
Poner en falso		CLR Op1 Res

Saltar si verdadero		JPT Op1 Lbl
Saltar si falso		JPF Op1 Lbl
carga indirecta		LIND Dir Res
Almacenamiento indirecto		SIND Op1 Dir
Captura de Puntero		PTR Op1 Res
Definición de BOOL		BOOL Name
Definición de SINT		SINT Name
Definición de INT		INT Name
Definición de USINT		USINT Name
Definición de UINT		UINT Name
Salto incondicional	GOTO	JUMP Lbl
No opere	NOP	NOP
Retorno de procedimiento		RET

3.3.2 Equivalencias del STL-intermedio

Cuando el compilador encuentra expresiones de más de dos operandos, el compilador se encargará de reducirlas a expresiones más simples (de máximo dos operandos), utilizando variables temporales [8]. Ejemplo:

Para la expresión:

```
A := B + C + ( D * A);
```

Se generará el código intermedio:

```
1:   MUL    D    A    S1
2:   ADD    B    C    S2
3:   ADD    S1   S2   A
```

S1 y S2 son variables temporales.

Esto es también aplicable a las expresiones de comparación.

Para las sentencias se utilizarán los saltos condicionales e incondicionales, así como las instrucciones de comparación.

Ejemplo:

```
IF      A = B    THEN
BEGIN
        A := A + #1;
END

ELSE
BEGIN
        B := A - #1;
END
```

Se generará el código intermedio:

```
1:   EQ     A     B B1
2:   JPC    B1    (5)
3:   INC    A     A
4:   JUMP   (6)
5:   INC    A     B
```

Ejemplo:

```
REPEAT

        A: =  B + C;
        D: =  D + #1;

      UNTIL D = #30;
```

Se generará el código intermedio:

```
1:   ADD  B    C     A
2:   INC  D    D
3:   EQ   D    #30   B1
4:   JPC  B1   (1)
```

Para la implementación de funciones, el compilador debe asignar memoria a las variables usadas como parámetros, hacer una carga de operandos antes de la llamada a la función o procedimiento y después copiar el resultado en la variable de salida (para el caso de las funciones). La variable de salida tendrá el mismo nombre de la función.

Para una función:

```
FUNCTION  RD3(S1 : INT; S2 : INT; S3 : INT ); INT
BEGIN
  RD3 :=  S1 * S2 / S3;
END

A:=  RD3(C, B, D);
```

El cuerpo de la función sería:

```
1:   MUL   S1    S2    I1
2:   DIV   I1    S3    RD3
3:   RET
```

Y cuando es llamada la función:

```
10:    MOVE    C     S1
11:    MOVE    B     S2
12:    MOVE    D     S3
13:    CALL    (1)
14:    MOVE    RD3   A
```

4. Generador de código

La generación de código es la parte del proceso de compilación en la cual se tienen que conocer muy bien las características de la maquina, para la cual se tiene que traducir el código intermedio. Para el caso del PLC-UD gama Cero, se optó por generar varias versiones del mismo, por consiguiente, utilizar diversas tecnologías; para el caso específico de este proyecto el procesador utilizado fue el Microcontrolador PIC16F876A de Microchip. No obstante, para futuras implementaciones del PLC-UD en las que se llegue a utilizar alguna tecnología diferente, sería necesario modificar únicamente el generador de código, puesto que es precisamente la ventaja que trae la compilación a un código intermedio, con respecto a la compilación directa a código máquina.

El generador de código recibe como entrada un archivo de texto escrito en lenguaje intermedio STL PLC-UD. Este generador es de tres barridos, es decir, realiza tres pasos de lectura-reconocimiento: el primer paso es el reconocimiento de las variables definidas y su correspondiente asignación de memoria, el segundo es el reconocimiento de las instrucciones ejecutables y, por ultimo, la optimización de código. La salida del generador de código es un archivo en lenguaje assembler con el código generado para el procesador (con extensión .asm), y un reporte de errores en un archivo de extensión .err.

El reconocimiento de variables reservadas se realiza por medio de una tabla de símbolos (diferente a la tabla de símbolos utilizada en las fases posteriores de la compilación) que se carga desde un archivo externo, a un objeto de tipo SymbolChart. Se definió otro objeto de tipo SymbolChart para las variables declaradas por el usuario; ésta se completa en el paso de asignación de memoria y es utilizada en las otras dos fases de la generación de código. El proceso es similar al análisis léxico nombrado anteriormente.

4.1 Asignación de memoria

La lectura del código intermedio se hace en forma secuencial: la asignación se hace en dependencia del orden en que se encuentren las variables definidas. Cuando se encuentra una instrucción de declaración, el generador de código agrega el nombre de la variable pasada como parámetro en la tabla de variables (en la columna de componente léxico) y genera el código correspondiente a dicha declaración.

Ejemplo:

```
01: BOOL   var1   (*Declaración*)
02: BOOL   var2
03: BOOL   var3
04: BOOL   var4
05: BOOL   var5
06: BOOL   var6
07: SINT     A
08: USINT    B
09: INT      C
10: UINT     D
12: BOOL   var7
13: BOOL   var8
14: BOOL   var9
```

Generará el siguiente código:

```
DEFINE   var1    02e,0
DEFINE   var2    02e,1
DEFINE   var3    02e,2
DEFINE   var4    02e,3
DEFINE   var5    02e,4
DEFINE   var6    02e,5
DEFINE    A      02f
DEFINE    B      030
DEFINE    CL     031
DEFINE    CH     032
DEFINE    DL     033
DEFINE    DH     034
DEFINE   var7    02e, 6
DEFINE   var8    02e,7
DEFINE   var9    035,0
```

Obsérvese que para las variables INT y UINT se reservan dos bytes de memoria que se diferencian por la terminación "H" o "L" para las partes alta y baja respectivamente. Para las variables BOOL se reserva un BIT en específico de un registro. En el ejemplo se muestra que así no se declaren estas variables de forma consecutiva, el generador de código asigna bits consecutivos, hasta que se completan los 8 bits del byte inicial, una vez se acabe el espacio se asigna a la siguiente variable BOOL al primer BIT de la siguiente posición de memoria libre.

4.2 Instrucciones ejecutables

Una vez hecha la asignación de memoria y completada la tabla de variables se procede a hacer la fase de reconocimiento de instrucciones ejecutables. Esta parte de la generación de código se divide en varios pasos: el reconocimiento de la instrucción, la comprobación de tipos, la búsqueda e inclusión del archivo .asm asociado, reemplazo de operandos y la indicación de los errores detectados.

4.2.1 Reconocimiento de instrucciones

Como se mencionó anteriormente el reconocimiento de las palabras reservadas se realiza por medio de la tabla de símbolos; esto se hace leyendo el archivo de entrada y buscando correspondencia con la tabla de símbolos. Cuando se encuentra, se leen los demás parámetros de la tabla que van a ser utilizados en los pasos siguientes.

Después del reconocimiento se procede a comprobar los tipos de operandos en dependencia del tipo de instrucción; las tablas de símbolos y de variables entregan la información necesaria para dicha comprobación. Una vez conocidas las restricciones de tipos de cada instrucción (por medio de la tabla de símbolos) se procede comprobar el tipo de cada operando (éste es cargado en la tabla de variables, durante el proceso de asignación de memoria). La comprobación de tipos depende de cada instrucción, por esto no existe un algoritmo general para esta función.

La generación de código es la parte del proceso de compilación en la cual se tienen que conocer muy bien las características de la maquina, para la cual se tiene que traducir el código intermedio.

4.2.2 Búsqueda e inclusión del archivo ASM

Por medio de la información de la tabla de símbolos y de la tabla de variables acerca de los tipos de operandos se busca el archivo .asm con la rutina en lenguaje assembler correspondiente a la instrucción reconocida. Una vez ubicado el archivo se carga en un buffer temporal, para ser incluido en el archivo de salida. Esta búsqueda se hace necesaria debido a que las rutinas .asm están guardadas en archivos de texto independientes y ordenadas en dependencia del tipo de operandos (rutinas de 1, 8 y 16 bits y rutinas para operandos con signo).

4.2.3 Formato de las rutinas ASM

Las rutinas en lenguaje assembler utilizan variables genéricas (se muestran iniciando con doble carácter "_"), esto con el fin de que una vez cargada en el buffer temporal se cambien los nombres de éstas por los nombres de los operandos reales.

Ejemplo:

```
BANKSEL   __BRES
MOVF              T_OP1,W
ADDWF     T_OP2,W
MOVWF     __RES
```

En este caso las variables genéricas son "__RES" y "__BRES", las cuales serán reemplazadas por el nombre del resultado y el banco de memoria del resultado respectivamente. También se observa que los operandos 1 y 2 no son variables genéricas, porque son variables incluidas en el banco de memoria del PLC, que se utilizan en la transferencia de registros entre bancos de memoria y deben ser cargados con anterioridad con los valores de los operandos reales.

Ejemplo:

```
BANKSEL   __BOP1
MOVF              __OP1, W
MOVWF     T_OP1
BANKSEL   __BOP2
MOVF              __OP2, W
MOVWF     T_OP2
```

Existen instrucciones que requieren llamadas a funciones en lugar de incluir directamente el código, debido a lo extenso de la rutina en sí; este es el caso de las multiplicaciones, divisiones y comparaciones de operandos con signo.

Ejemplo:

```
CALL      MUL_SINT
BANKSEL   __BRES
MOVF              T_RES, W
MOVWF     __RES
```

En este caso se incluirá el código de la rutina de multiplicación al final de todo el código generado.

```
MUL_SINT
CLRF      T_RES
          CLRF      AR2
          MOVF      T_OP2,W
          MOVWF AR0
          MOVLW .8
          MOVWF AR1
__L1      BCF       STATUS,C
          BTFSS     AR0,0
          GOTO      __L2
          MOVF      T_OP1,W
          ADDWF  AR2,F
__L2      RRF       AR2, F
          RRF       T_RES,F
          RRF       AR0,F
          DECFSZ AR1,F
          GOTO      __L1
          RETURN
```

Se puede observar que las etiquetas también son genéricas y serán reemplazadas por el generador de código, por etiquetas reales en dependencia de un contador de saltos interno.

4.2.4 Reemplazo de operandos

Como ya se vio una vez cargada la rutina .asm, en el buffer temporal se procede a reemplazar las variables, indicadores de banco y etiquetas genéricas de éste, por los nombres reales de las variables. Una vez reemplazados por los valores reales se suma al archivo de salida.

4.2.5 Detección de errores por línea

Durante el reconocimiento de instrucciones y la confirmación de tipos, el generador de código detecta los errores de escritura en el código intermedio. Los errores que éste puede detectar son: errores de escritura de la instrucción, operandos de tipos inválidos en dependencia del tipo de instrucción, tipos diferentes entre los operadores de una misma instrucción, referencias a variables no declaradas y detección de parámetros innecesarios. Estos errores se registran en un reporte de errores, un archivo con extensión .err; en caso de encontrarse errores el archivo .asm se crea vació.

4.3 Optimización de código

En este paso del proceso se realiza un barrido por el buffer de salida definitivo antes de guardar el archivo final, esto con el fin de reducir, si es posible, el número de líneas de código máquina y ahorrar espacio en memoria de programa.

En nuestro caso, la parte a optimizar del código generado es los saltos entre bancos de memoria de datos. Cuando se realiza cualquier instrucción de más de un operando se tiene que hacer un salto de banco para transferir los operandos ubicados en diferentes bancos de memoria hacia el banco de memoria en el que se encuentra el resultado; como este proceso se realiza siempre se pueden presentar saltos consecutivos al mismo banco de memoria. El optimizador de código realiza una lectura secuencial del buffer de salida y elimina los saltos de banco innecesarios.

5. Clases base para la implementación

Para la implementación del compilador se utilizó la herramienta Builder C++ 6.0 con licencia propiedad de la Universidad. Para el desarrollo se crearon clases base como: cadenas dinámicas, filtros de texto, listas enlazadas, pilas y tablas dinámicas, con el fin de simplificar el trabajo a la hora de implantar el compilador y a su vez para dejar las librerías creadas como base para trabajos posteriores tanto dentro del grupo de investigación como en la Universidad, en general.

Como base fundamental para la implementación del compilador, se creó una clase llamada TextString que no es más que una cadena de texto dinámica [9] [10], es decir, hace uso de la memoria de forma dinámica en dependencia del número de caracteres que contenga. Para dotar de más flexibilidad y facilidad de uso, la clase TextString tiene sobrecargados varios operadores, como son: la asignación, operadores de igualdad y de desigualdad y el operador de suma (concatenación).

Para la implementación del compilador se utilizó la herramienta Builder C++ 6.0 con licencia propiedad de la Universidad. Para el desarrollo se crearon clases base como: cadenas dinámicas, filtros de texto, listas enlazadas, pilas y tablas dinámicas

Tomando como base la clase TextString se derivó la clase FileText; se trata de una clase de objetos buffer para archivos de texto, ya que tienen la capacidad de cargar en memoria archivos de texto, para facilitar el trabajo con estos. También permiten guardar en disco el contenido del buffer en archivos de texto. Al tratarse de una clase derivada de TextString, para su uso se tiene que incluir el archivo de cabecera TextString.h.

Para facilitar el trabajo del analizador léxico se creó una clase de filtros de texto llamada TextFilter que incluye funciones de filtro de comentarios de una y varias líneas, de caracteres y palabras. Esta clase se deriva de la clase FileText y necesita la inclusión del archivo de cabecera FileText.h.

Para la posterior implementación de pilas dinámicas y tablas de símbolos se creó las clases de lista enlazada de texto y de enteros [11] [12] BaseTextList y BaseIntList, las cuales utilizan datos miembros de las clases BaseTextElement y BaseIntElement, también definidas para este propósito. Se debe incluir el archivo de cabecera TextString. h debido a que las clases definidas utilizan datos miembros pertenecientes a la clase TextString. La clase TextStack define una pila de texto dinámica que es derivada de la clase BaseTextList y necesita la inclusión del archivo de cabecera BaseList.h para su uso.

La clase tabla de símbolos es la estructura de datos más importantes dentro de la compilación, ya que se utiliza en casi todos los pasos del proceso. La clase SymbolChart está definida como un arreglo de listas, es decir, cada casilla de la tabla es una lista enlazada diferente; esta estructura es dinámica sólo en una de las dimensiones.

Se implementó un software de compilación en lenguaje STL para el PLC-UD, lo que permite su programación en un lenguaje estructurado de alto nivel, con la ventaja en cuanto a tiempo de desarrollo que tiene este tipo de programación con respecto a otros de más bajo nivel.

6. Resultados

Se implementó un software de compilación en lenguaje STL para el PLC-UD, lo que permite su programación en un lenguaje estructurado de alto nivel, con la ventaja en cuanto a tiempo de desarrollo que tiene este tipo de programación con respecto a otros de más bajo nivel.

Se definió un lenguaje STL que se basó en las normas estándar de la IEC 61131 y que tuvo en cuenta los alcances del PLC y las limitaciones propias del hardware utilizado.

Se dotó al proyecto de investigación de una herramienta capaz de generar código ejecutable para el PLC, tanto en lenguaje STL como en los otros lenguajes de programación desarrollados o en miras a desarrollarse. De igual manera, se sentaron las bases de conocimiento necesarias para futuras implementaciones, dentro del proyecto de investigación y en la facultad.

Por otra parte, se realizó una selección de tecnología hardware para el núcleo del PLC-UD, teniendo en cuenta sus requerimientos tanto económicos como técnicos. Por último, se proporcionó al proyecto de investigación no sólo una herramienta de compilación a lenguaje máquina de microcontroladores Microchip, sino que también, por medio de la implementación de un lenguaje intermedio, se logró independizar el lenguaje STL del procesador utilizado, lo que permitió futuras implementaciones de PLC utilizando otras tecnologías hardware.

7. Conclusiones

La forma más conveniente de implantar un compilador sin utilizar herramientas com-

putacionales adicionales es utilizando la metodología de análisis descendente, debido a que es posible implementar manualmente dicho análisis, ya sea por medio de funciones o de una matriz de análisis sintáctico. Los puntos en contra al utilizar esta metodología son la rigurosidad que se debe tener al definir la gramática del lenguaje y la poca facilidad que ofrece ésta a los cambios del lenguaje.

El análisis ascendente es demasiado complejo para implementarlo manualmente, para próximas implementaciones es posible utilizar esta metodología empleando para ello generadores automáticos de analizadores sintácticos, disponibles gratuitamente en Internet; esto reduciría el tiempo de desarrollo y permitiría realizar los cambios al lenguaje de forma más eficiente.

La implementación de un lenguaje intermedio permite independizar el lenguaje, del procesador utilizado, de esta forma, se facilita cambiar de máquina, teniendo sólo que rediseñar la parte de generación de código.

El microcontrolador utilizado (PIC16F876A) tiene una arquitectura RISC, lo que implica un mayor trabajo de compilación, debido a la ausencia de instrucciones especializadas, tales como: multiplicación, comparación, comunicación I2C etc., pero a la vez el código generado para este tipo de procesador es mucho más eficiente en cuanto a tiempo de ejecución.

Referencias bibliográficas

[1] Becerra, Cesar. (1997). Estructuras de datos en turbo pascal. Bogotá: Por Computador Ltda.

[2] _____. (1990). Estructuras de datos en C. Por Computador Ltda.

[3] Martínez, F. y Vargas, D. (2005). Lenguaje intermedio para el STL PLC_UD. Bogota: Universidad Distrital Francisco José de Caldas.

[4] _____. (2003). Restricciones del STL para el PLC gama cero. Bogotá: Universidad Distrital Francisco José de Caldas.

[5] _____. (2003). Guía de selección de tecnología. Bogotá: Universidad Distrital Francisco José de Caldas.

[6] Microchip. (2003). PIC16F87XA. Extraído de la World Wide Web: www.microchip.com.

[7] Pozo, Salvador. (2004). C con Clase. Extraído de la World Wide Web: http://c.conclase.net

[8] Schildt, Herbert. (1995). Aplique turbo C++. S.d.: McGraw Hill.

[9] Teufel- Schmidt- Teufel. (1995). Compiladores conceptos fundamentales. Addison-Wesley, Wilmington, [7] A. V. Aho, R. Sethi y J. D. Ullman. Compilers – Principles, Techniques and Tools. Addison-Wesley, 1986.

[10] Valcárcel, John y Vargas, D. (2003) Listado general de STL. Bogotá: Universidad Distrital Francisco José de Caldas.

[11] Vargas Vela, Daniel. (2002). Diseño e implementación del PLC-UD. Bogotá: Universidad Distrital Francisco José de Caldas.