Revista Tekhnê
2011, Vol. 8, 47–54

ISSN 1692-8407

Sistema de desarrollo para microcontrolador Atmel orientado a PLC

Development system for Atmel microcontroller oriented towards PLC

Delida Farfán León
Universidad Distrital Francisco José de Caldas
d.farfan.idi@gmail.com

Fernando Martínez Santa
Universidad Distrital Francisco José de Caldas
fmartinezs@udistrital.edu.co

Ricardo Pirajan Cantillo (Q.E.P.D.)
Universidad Distrital Francisco José de Caldas
rpirajan@yahoo.com

Este artículo presenta el diseño y la elaboración de un sistema de desarrollo para microcontrolador Atmel, orientado a Controladores Lógicos Programables, y específicamente aplicado al proyecto de investigación de la Universidad Distrital PLC-UD. El sistema se encarga de generar un código ejecutable para los PLCs, a partir del lenguaje estructurado STL. Se realizó la investigación con el fin de implementar el estudio y la aplicación de la tecnología Atmel en subsiguientes proyectos, como también alcanzar, junto con una serie de trabajos señalados por otros estudiantes, los objetivos generales del proyecto de investigación.

Palabras clave: Atmel, generación de código, microcontrolador, PLC, STL

This paper presents the design and development of a development system for Atmel microcontroller, Programmable Logic Controllers oriented, and applied specifically to the research project of the University District PLC-UD. The system is responsible for generating executable code for PLCs from STL structured language. Research was conducted in order to implement the study and application of Atmel technology in subsequent projects, as achieving, along with a series of works identified by other students, the general objectives of the research project.

Keywords: Atmel, code generation, microcontroller, PLC, STL

Introducción

El proyecto de investigación *Diseño e Implementación del PLC-UD* pretende apropiar, asimilar y generar el conocimiento necesario para crear una gama de PLCs que tengan especificaciones similares a las tecnologías extranjeras. A pesar de esto, se pretende generar el beneficio de un menor costo, a fin de ofrecer una alternativa a las pequeñas y medianas empresas interesadas en aumentar y mejorar sus procesos y su capacidad competitiva. Se necesita ahora un software de programación que se adapte a los requerimientos de tal tarea y que fundamentalmente sea de fácil manejo para el alcance de todos los interesados en desarrollarlo.

Se define el lenguaje STL (*Statement List*) como estructura gramatical según los criterios de la norma estándar IEC 61131 (Otto y Hellmann, 2009; Pereira, Lima, y Martins, 2011) que cubre los lenguajes de programación para PLCs (Economakos y Economakos, 2009; Siemens, 2010). Se trata de un lenguaje estructurado de alto nivel con una sintaxis similar a la del PASCAL, el cual con las adecuadas técnicas de compilación generará un lenguaje de máquina para el desarrollo Atmel.

El artículo se estructura de la siguiente forma. La sección 2 presenta la formulación del problema, en particular el lenguaje STL para el PLC-UD. La sección 3 muestra los deta-

Fecha recepción del manuscrito: Mayo 2, 2011
Fecha aceptación del manuscrito: Agosto 10, 2011

Delida Farfán León, Fernando Martínez Santa y Ricardo Pirajan Cantillo, Facultad Tecnológica, Universidad Distrital Francisco José de Caldas.

Esta investigación fue financiada por: Universidad Distrital Francisco José de Caldas.

Correspondencia en relación al artículo debe ser enviada a: Delida Farfán León. Email: d.farfan.idi@gmail.com

lles del proceso de compilación desde los puntos de vista de léxico y sintáctico. La sección 4 detalla el generador de código y el uso de memoria. La sección 5 muestra los detalles de la implementación hardware, y la sección 6 concluye el artículo.

Formulación del problema: Lenguaje STL para el PLC-UD

La estructura general de un programa en STL se presenta en la Figura 1. La coloración del fondo del texto, utilizada en este articulo, facilita su ubicación dentro del proceso de compilación mostrado en la Figura 2.

```
pragma id
...
pragma id      (*comentarios*)

uses id, ..... id;

program id;

bloque_de_declaraciones_de variables
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 .
```

Figura 1. Estructura general de un programa en STL.

Como se puede observar el programa inicia con la directiva *pragma*, la cual se utiliza para dar información al compilador, como la gama del PLC, o el núcleo utilizado; esta directiva es opcional. La instrucción *uses* le indica al compilador que incluya los archivos de cabecera indicados por un identificador asociado. Este 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. La directriz anterior también es opcional (Vargas y Martínez, 2008).

Se inicia el programa principal con la palabra reservada *program* y un identificador asociado, seguido del símbolo (;). 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 (Economakos y Economakos, 2009).

Una vez realizadas todas las declaraciones requeridas, se inicia la rutina principal propiamente dicha (cuerpo del programa). Esta comienza por la palabra reservada *begin* y continúa por 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.

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

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

Los tipos de datos definidos son: boléanos (BOOL), enteros, enteros sin signo, enteros cortos, y enteros cortos sin signo (Siemens, 2010), especificados en la Tabla 1.

Se utilizarán los cuatro tipos de operadores: aritméticos, lógicos, de comparación y de asignación. Los aritméticos son aquellos usados en expresiones matemáticas, estos son: +, -, *, /, % y corresponden respectivamente a suma, resta, multiplicación, división y módulo. Los operadores lógicos son: *and*, *or* y *not*, correspondientes a las operaciones lógicas y, ó, y negación. Estos se utilizan únicamente para variables de tipo BOOL, y en la evaluación de condiciones. Los operadores de comparación son: =, <>, <, >, <=, >= y corresponden a: igual, diferente, menor a, mayor a, menor o igual y mayor o igual. Los operadores de comparación se utilizan para relacionar expresiones en las estructuras condicionales. El resultado de las operaciones de comparación siempre es un valor booleano. 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) (Vargas y Martínez, 2008).

Proceso de compilación

El compilador funciona 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 al microcontrolador de Atmel, en este caso de la familia MCS 80C51 (AT89S52) y actuará como núcleo del PLC (Lobov, Popescu, y Lastra, 2006; Sun, Tian, y Dong, 2009).

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 (Figura 2) (Sousa y Carvalho, 2003; Zhou y Chen, 2009). Para independizar el compilador del procesador utilizado, y con el fin de generar el 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, este 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 (Aho, Lam, Sethi, y Ullman, 2006; Teufel, 1998).

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 refleja en el código intermedio (Vargas y Martínez, 2008).

Análisis léxico

El proceso de análisis léxico se implementó utilizando una tabla de símbolos, la cual contiene todas las palabras reservadas del lenguaje STL, con algunos otros datos útiles para las siguientes fases de la compilación. También está encargado de eliminar comentarios y caracteres innecesarios tales como tabuladores, espacios y fines de línea, 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 carácter por carácter y se va cargando en un pequeño *buffer*, el cual 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 (Tabla 2) se muestra parte de la tabla de símbolos del compilador (Vargas y Martínez, 2008).

Análisis sintáctico

Para esta parte del proceso se utilizó el método de análisis descendente recursivo. Este consiste en segmentar 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 (Aho et al., 2006).

La técnica de implantación utilizada fue el análisis sintáctico descendente tabular (Figura 3), que es una forma generalizada de análisis descendente. Este tipo de analizador no

Tabla 2
Tabla de símbolos del compilador

1		$Operator	$Nothing	1	65535
...		...	...	...	...
9		$Operator	$Nothing	9	65535
+	ADD	$Operator	$Nothing	10	65535
*	MUL	$Operator	$Nothing	12	65535
%	MOD	$Operator	$Nothing	14	65535
=	EQ	$Operator	$Nothing	15	65535
<=	LE	$Operator	$Nothing	17	65535
>=	GE	$Operator	$Nothing	18	65535

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 (Teufel, 1998).

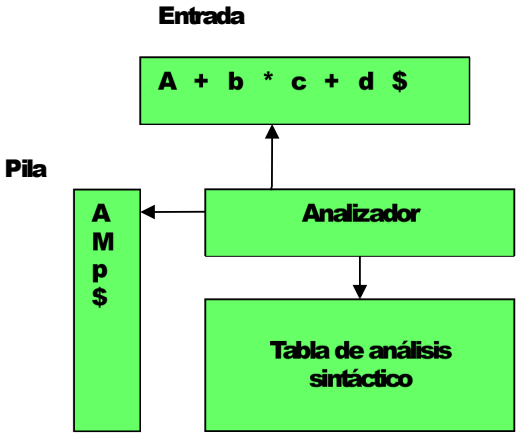

Figura 3. Análisis sintáctico tabular (Vargas y Martínez, 2008).

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 tabla de análisis sintáctico es una matriz bidimensional que asocia el símbolo terminal a reducir, con el símbolo terminal de entrada y contiene la producción correspondiente a cada pareja *Terminal - No_terminal*, tal que el analizador sintáctico tenga solo 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 dentro del análisis.

Analizador semántico

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

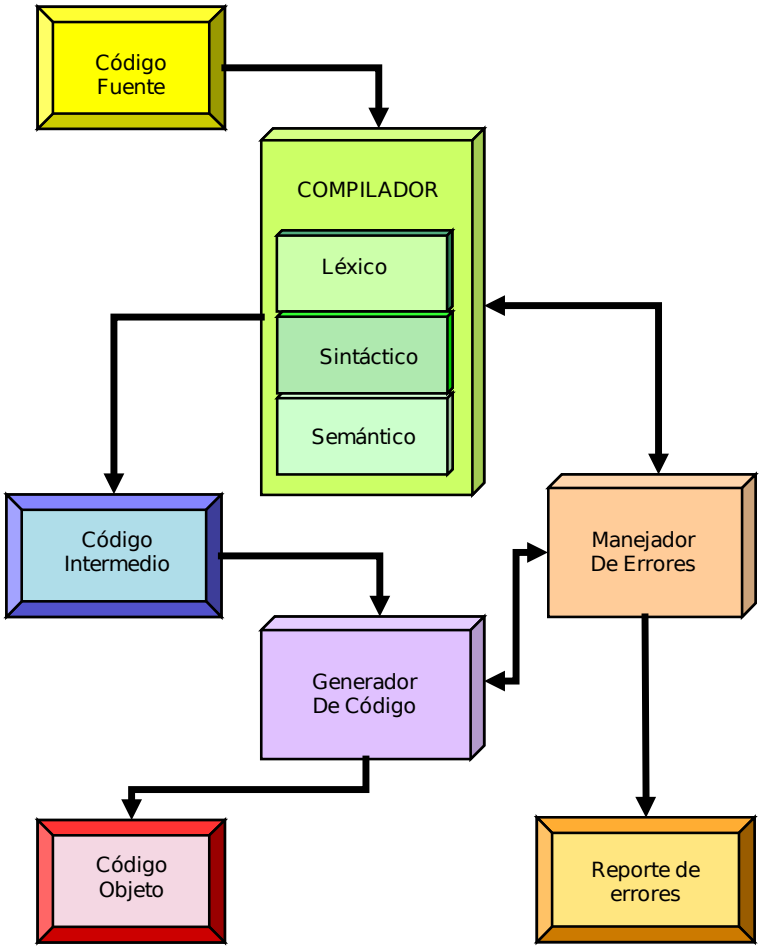

Figura 2. Proceso de compilación (Vargas y Martínez, 2008).

Las acciones semánticas consisten básicamente 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 usan variables temporales, para los cálculos de expresiones de más de un operador que en teoría debe ser un vector infinito de variables temporales. Es claro que existen restricciones en cuanto disponibilidad de memoria. Dichas 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 (Vargas y Martínez, 2008).

Lenguaje intermedio

El lenguaje intermedio PLC-UD se define como un lenguaje de cuádruplos o de tres direcciones, es decir que para cada instrucción o línea de código existirán un máximo de cuatro palabras separadas por espacios. La primera 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 (Vargas y Martínez, 2008). Por ejemplo: Lenguaje intermedio para el STL PLC-UD

Operador	Op1	Op2	Res

Para realizar de una forma mas fácil los saltos entre líneas de código, se incluye una numeración al inicio de cada línea y termina con dos puntos. Por ejemplo:

No. Línea:	Operador	Op1	Op2	Res

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

Existen instrucciones de tres, dos, uno y ningún parámetro, dentro de 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 no reciben parámetros.

A continuación en la Tabla 3 se muestra el listado completo de instrucciones del lenguaje intermedio definido para el STL.

Equivalencias del STL intermedio. Cuando el compilador encuentra expresiones de más de dos operandos, se encargará de reducirlas a expresiones más simples (de máximo dos operandos), utilizando variables temporales (Vargas y Martínez, 2008). Por ejemplo:

```
F7 := (F2 + C) + F1 / F2 * F3;
```

Genera el código intermedio:

```
7:  ADD F2 C __I0
8:  DIV  F1 F2 __I1
9:  MUL __I1  F3 __I2
10: ADD __I0  __I2  __I3
11: MOVE __I3  F7
```

Donde __I0, __I1, __I2, __I3 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. Por ejemplo, el código de la Figura 4 genera el código intermedio de la Figura 5.

```
begin
  repeat
    begin
      if( F1 = #100 )then F2 := #10;
            Q0_0 := true;
    end
  until ( #0 = #1 );
end .
```

Figura 4. Ejemplo saltos condicionales e incondicionales.

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, 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.

Generador de código

La generación de código es la parte del proceso de compilación, en donde se tiene que conocer muy bien las características de la máquina para la cual se tiene que traducir el código intermedio. Para el caso del PLC-UD gama Cero, se planteó generar varias versiones del mismo utilizando

Tabla 3
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
Ó	OR	OR Op1 Op2 Res
Ó 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 literal		LIT Res K
Llamado		CALL Line
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
Colocar en verdadero		SET Op1 Res
Colocar en falso		CLR Op1 Res
Saltar si verdadero		JPT Op1 Lbl
Saltar si falso		JPF Op1 Lbl
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 Line
No opere	NOP	NOP
Retorno de procedimiento		RET

```
7:  MOVE F1 __I0
8:  LIT 100 __I1
9:  EQ __I0 __I1 __B0
10: JPF __B0(14)
11: LIT 10 __I1
12: MOVE __I1 F2
13: JUMP (14)
14: SET __B0
15: MOVE __B0 Q0_0
16: LIT 0 __I0
17: LIT 1 __I1
18: EQ __I0 __I1 __B0
19: JPF __B0(7)
```

Figura 5. Código intermedio para ejemplo saltos condicionales e incondicionales.

diversas tecnologías, en este caso el procesador es AT89S52 de Atmel. Para futuras implementaciones del PLC-UD utilizando alguna tecnología diferente, sería necesario modificar únicamente el generador de código, esto es precisamente la ventaja que trae la compilación a un código intermedio con respecto a la compilación directa a código de máquina (Vargas y Martínez, 2008).

El generador de código recibe como entrada un archivo de texto escrito en lenguaje intermedio STL PLC-UD. Este generador es de tres pasadas, 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 último la optimización de código. La salida del generador de código es un archivo en lenguaje ensamblador con el código generado para el procesador (con extensión *.asm*), y un reporte de errores en un archivo de extensión *.err* (Vargas y Martínez, 2008).

El reconocimiento de variables reservadas se realiza por medio de una tabla de símbolos (diferente a la tabla de símbolos utilizada en las fases anteriores); la cual 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, esta 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 (Vargas y Martínez, 2008).

Asignación de memoria

La lectura del código intermedio se hace en forma secuencial, es decir que la asignación se hace en dependencia del orden en que se encuentren definidas las variables. 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. Por ejemplo, la Figura 6 genera el código mostrado en la Figura 7.

```
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
```

Figura 6. Ejemplo declaración de variables.

```
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
```

Figura 7. Código para ejemplo declaración de variables.

Obsérvese que para las variables INT y UINT se reserva 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.

Instrucciones ejecutables

Una vez hecha la asignación de memoria y completa 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, el reemplazo de operandos y la indicación de los errores detectados (Vargas y Martínez, 2008).

Reconocimiento de instrucciones. Al leer el archivo de entrada se busca correspondencia con la tabla de símbolos. En este caso, para facilitar el proceso, se dividió dicha tabla en dos: de símbolos, donde están definidas las palabras reservadas y en la otra se definen las variables a utilizar. Una vez se encuentra relación, se leen los demás parámetros de

la tabla, para comprobar los tipos de los 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 a comprobar el tipo de cada operando (este es cargado en la tabla de variables, durante el proceso de asignación de memoria).

Búsqueda e inclusión del archivo ASM. Por medio de la información de las tablas de símbolos y de variables acerca de los tipos de los operandos, se busca el archivo *.asm* con la rutina en lenguaje ensamblador correspondiente a la instrucción reconocida. Una vez ubicado el archivo se carga en una memoria 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) (Vargas y Martínez, 2008).

Las rutinas en lenguaje ensamblador utilizan variables genéricas (inician con doble carácter _), con el fin de que una vez cargadas en la memoria temporal se cambien los nombres de estas por los nombres de los operandos reales. Existen además variables incluidas en el banco de memoria del PLC que se utilizan para propósitos generales, como en la transferencia de registros entre bancos, estas deben ser cargadas con anterioridad con el valor real de los operandos.

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 comparaciones de operandos con signo.

En este caso se incluirá el código de la rutina de dicha comparación al final de todo el código generado. Las etiquetas también son genéricas, y serán reemplazadas en el generador de código, por etiquetas reales en dependencia de un contador de saltos interno. En general una vez cargada la rutina *.asm* en la memoria temporal se procede a reemplazar las variables, indicadores de banco y etiquetas genéricas de este, por los nombres reales de las variables. Estos valores reales a su vez se suman al archivo de salida.

Detección de errores por línea. Durante el reconocimiento de instrucciones y la confirmación de tipos de operando, el generador de código detecta los errores de escritura en el código intermedio, estos 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 alguno el archivo *.asm* se crea vacío (Vargas y Martínez, 2008).

Optimización del código

En este paso del proceso se realiza una pasada por la memoria de salida definitiva antes de guardar el archivo final, con el fin de reducir si es posible el número de líneas de código máquina y ahorrar espacio en la memoria de programa.

En este caso la parte a optimizar del código generado son 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 al banco de memoria donde se encuentra el resultado; en este proceso 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 (Vargas y Martínez, 2008).

Implementación del hardware

Para la programación del microcontrolador Atmel y la verificación del funcionamiento del código generado, es necesario contar con una herramienta que emule funciones propias del PLC-UD gama cero, que cumpla con las especificaciones técnicas y además sea de fácil operación.

Implementamos la programación ISP (*In-System Programming*) utilizando su fácil diseño en estos microcontroladores y demás ventajas propias, entre otras porque funciona con el software gratuito provisto por Atmel. En general, el circuito debe cumplir los siguientes requisitos mínimos (Atmel, 2006).

- Microcontrolador AT89S52 con alimentación (regulada) aplicada, cristal y condensadores asociados conectados (o circuito oscilador externo correctamente conectado y funcionando).
- La interfaz de programación que consiste simplemente en *buffers* para las líneas de señal, teniéndose presente las referencias de los resistores que varía con el circuito integrado utilizado.
- Señales del micro MOSI, MISO, SCK y RESET cableadas al conector previsto para la programación ISP, sin conflictos con posibles conexiones de la aplicación (MOSI, MISO, y SCK utilizadas como entradas/salidas de propósito general: P1.5, P1.6, P1.7) o de *power-on-reset* (RESET).

En la Figura 8 se muestra el esquema de la tarjeta implementada, la cual se basa en un microcontrolador el AT89S52, que se constituye según el grupo de investigación en el PLC de gama cero con microcontrolador Atmel.

Resultados y conclusiones

Éste artículo documenta el diseño e implementación de un sistema de desarrollo para el microcontrolador AT89S52 de Atmel orientado a su uso en un PLC. El trabajo hace parte, se financia y orienta, bajo la coordinación del proyecto

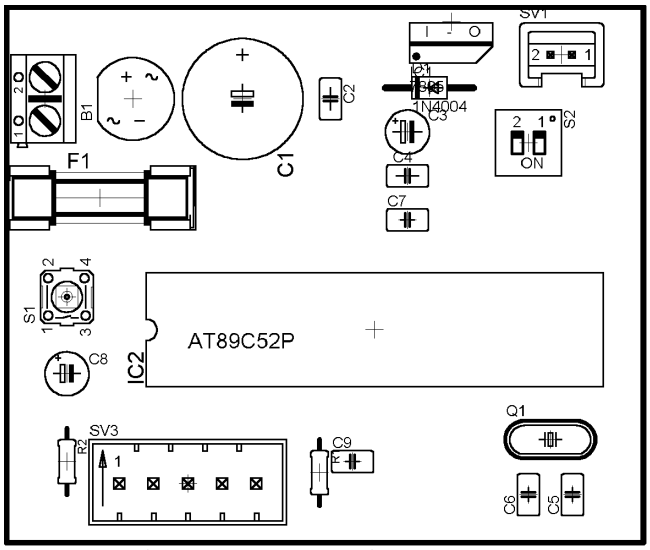

Figura 8. Tarjeta ISP para AT89S52.

de investigación macro *Diseño e Implementación del PLC-UD* de la Universidad Distrital Francisco José de Caldas. En tal sentido, en éste proyecto se logró complementar desarrollos previos, tomando bases del trabajo desarrollado en (Vargas y Martínez, 2008), realizado también para el grupo de investigación. El proyecto da continuidad a esta labor, y a su vez sirve de impulso y orientación para futuras implementaciones en diversos proyectos de investigación.

Como meta destacable a mencionar, se logró, al implementar esta herramienta de compilación a lenguaje de máquina, un progreso fundamental en la exploración de nuevas tecnologías, en particular de Atmel, ya que esta no ha sido muy desarrollada a nivel de la Facultad Tecnológica de la Universidad Distrital, y en general en el campo nacional.

Se definió un lenguaje STL, basándose en las normas estándar de la IEC 61131 y teniendo en cuenta los alcances del PLC, y las limitaciones propias del hardware utilizado.

Se proporcionó una herramienta hardware sencilla y funcional, emuladora de funciones propias del PLC-UD gama cero, basada en software provisto por Atmel.

Referencias

Aho, A., Lam, M., Sethi, R., y Ullman, J. (2006). *Compilers: Principles, techniques, and tools* (2.ª ed.). Addison Wesley.

Atmel. (2006). *At89isp programmer cable.* On line. (Application Note)

Economakos, C., y Economakos, G. (2009). An architectural exploration framework for efficient FPGA implementation of PLC programs. En *17th mediterranean conference on control and automation med '09* (p. 1172-1177).

Lobov, A., Popescu, C., y Lastra, J. L. M. (2006). An algorithm for Siemens STL representation in TNCES. En *Ieee conference on emerging technologies and factory automation etfa* (p. 641-647).

Otto, A., y Hellmann, K. (2009). IEC 61131: A general overview and emerging trends. *IEEE Industrial Electronics Magazine*, *3*(4), 27-31.

Pereira, A., Lima, C., y Martins, J. F. (2011). The use of IEC 61131-3 to enhance PLC control and Matlab/Simulink process simulations. En *2011 ieee international symposium on industrial electronics (isie)* (p. 1243-1247).

Siemens. (2010). Statement list (stl) for s7-300 and s7-400 programming [Manual de software informático].

Sousa, M., y Carvalho, A. (2003). An IEC 61131-3 compiler for the MatPLC. En *Ieee conference emerging technologies and factory automation etfa03* (Vol. 1, p. 485-490).

Sun, R., Tian, Y., y Dong, Y. (2009). Design and implementation of industrial multi-parameter data acquisition system based on AT89S52. En *Third international symposium on intelligent information technology application workshops iitaw* (p. 169-172).

Teufel, S. (1998). *Compiladores conceptos fundamentales* (1.ª ed.). Addison Wesley Longman.

Vargas, D., y Martínez, F. (2008). Diseño e implementación del compilador STL para el PLC-UD. *Visión Electrónica*, *1*(1), 8-23.

Zhou, C., y Chen, H. (2009). Development of a PLC virtual machine orienting IEC 61131-3 standard. En *International conference on measuring technology and mechatronics automation icmtma09* (Vol. 3, p. 374-379).