

FACULTAD REGIONAL BUENOS AIRES

ROBÓTICA

TESIS FINAL

Tema: Embedded + Compilador (RT2: Apolo)

Alumnos:

- Bole, Nicolás
- Ríos, Leandro
- Zbucki, Hernán
-

Profesor: MaS. Ing. Hernán Gianetta

Revisión del trabajo: 1.4

Año Lectivo: 2009

Índice

0. Objetivos de la Tesis	3
1. Generación de Compiladores	3
1.1 Scanner vs Parser	3
1.2 Scanner	3
1.2.1 Definición	3
1.2.2 Funcionamiento	4
1.2.3 Entradas/salidas.	7
1.2.4 Ejemplos	7
1.3 Parser	9
1.3.1 Definición	9
1.3.2 Funcionamiento	9
1.3.3 Entradas y salidas.	11
1.3.4 Ejemplos	12
1.4 Compilador realizado (Complex)	13
1.4.1 Idea Conceptual	13
1.4.2 Bases	13
1.4.3 Código	13
1.4.4 Highlights	17
1.4.5 Funcionamiento y Ejemplos.	17
2. Embedded	17
2.1 Esquema General.	18
2.2 Entradas/Salidas del sistema.	19
2.3 VHDL	20
2.3.1 Introducción	20
2.3.2 Código utilizado en el FPGA	21
2.3.3 TestBench utilizado.	23
2.3.4 Simulación en ModelSim	26
2.4 Control de Potencia	27
2.4.1 Puente H	27
2.4.1.1 Esquema	27
2.4.1.2 Modo de funcionamiento	28
2.4.1.3 Oscilogramas	29
3. Conclusiones finales	30
4. Bibliografía	31

Objetivos de la Tesis:

- Dar breves reseñas teóricas de lo aprendido en la materia.
- Reinterpretar por escrito toda la información averiguada
- Generar un modelo simulado de un controlador embedded de los ejes de robot de 6 GDL propuesto por la cátedra.
- Generar un documento claro para ser integrado con los demás trabajos realizado sobre el robot.

1. Generación de Compiladores.

1.1 Scanner + Parser

Se puede decir que un Scanner y un Parser son partes o módulos fundamentales de lo que en conjunto generan: un compilador o más sencillo aun un intérprete. En las siguientes secciones veremos en detalle estos componentes por separado. Y luego enteremos cómo es que interactúan entre ellos. Cabe aclarar que un módulo de Scanner puede llegar cumplir su función por si solo; pero esto no sucede con el Parser. De hecho una función para la cual puede ser utilizado un Scanner, fuera del ámbito de compiladores e intérpretes, podría ser dentro de un procesador de texto al realizar la función "Buscar" o "Buscar y reemplazar".

1.2 Scanner

Nuestro proyecto está basado en una herramienta Open-Source llamada FLEX. "Flex, a Fast scanner generator" esta es una de las primeras referencias que se encuentran acerca del nombre de este programa. Si se indaga un poco más, se puede llegar a encontrar información que Flex es una evolución de LEX. Este último es una herramienta clásica de Unix para la generación de analizadores léxicos (recordemos que en inglés sería LEXical Analyzer).

Por último se puede destacar que Flex es compatible casi al 100% con Lex, pero es un desarrollo diferente realizado por GNU bajo licencia GPL, es por esto que hemos dicho que es un Open-Source.

1.2.1 Definición

Estas son las definiciones de Scanners o Analizadores Léxicos que se pueden encontrar en diferentes tipos de documentos, libros, tutoriales y HOW TO:

- "The lexical phase (scanner) groups characters into lexical units or tokens."
(§1)
[La fase léxica (scanner) agrupa caracteres en unidades léxicas o tokens]

- “The Function of a lexical Analyzer is to read the input stream representing the Source program, one character at a time and to translate it into valid tokens.” (§2)
[La función del Analizador léxico es leer desde la entrada representada por la el Código Fuente del programa, un carácter por vez y traducirlo a tokens válidos]
- “The lexical analyzer matches strings in the input, based on your patterns, and converts the strings to tokens. Tokens are numerical representations of strings, and simplify processing.” (§3)
[El analizador léxico iguala cadena de caracteres en la entrada, basada en sus patrones, y convierte la cadena de caracteres a tokens. Los tokens son representaciones numéricas de las cadenas de caracteres, y simplifican el procesamiento]

Luego de estas definiciones el lector puede tener la siguiente pregunta: ¿Qué es un token?

Podríamos seguir dando más definiciones y aun así no llegaríamos a un respuesta concreta. Notemos que la tercera definición que hemos incluido, intenta dar una idea aproximada a la pregunta planteada. Pero todavía no cumple con el objetivo de esclarecer el interrogante planteado. Conocer la respuesta a esta pregunta es fundamental para entender que hace un Scanner o un Analizador Léxico.

Asumamos, tan solo por un instante, que la traducción de “token” sea “comodín”. A partir de esto podríamos decir que, un Scanner o un Analizador Léxico es un programa que busca caracteres y/o grupos de caracteres, según ciertos patrones previamente definidos. Por lo tanto cada vez que se encuentra cierto carácter se asociara con cierto comodín y grupo de caracteres se asociará con otro comodín.

Si esta explicación no ha sido lo suficientemente aclaratoria, confiamos que un ejemplo cumplirá esta función. Dicho ejemplo se encuentra a continuación en la sección 1.2.4 Ejemplos.

1.2.2 Funcionamiento

La estructura de un archivo Flex debe ser la siguiente:

```
.....definiciones.....  
%%  
.....reglas o patrones.....  
%%  
.....código.....
```

Esta claro que el símbolo “%%” separa las tres secciones. Cabe destacar que la última sección, código, puede no estar. Mientras que la sección de reglas o patrones debe estar presente si o si.

A continuación daremos una explicación de las tres secciones:

1er) Sección de definiciones

Esta se encuentra dividida en tres:

a) *Código C*: Cualquier código que este entre "%{" y "%}" será copiado al archivo C (nos estamos refiriendo a la salida lex.yy.c) . En general se utiliza para definir variables y prototipos de funciones que estarán definidas en la sección de código.

b) *Definiciones*: son parecidas a los comandos de preprocesador del C tales como el #define, por ejemplo:

Letra	[a-zA-Z]
Dígito	[0-9]

Luego estas definiciones pueden ser usadas en la sección de reglas o patrones.

c) *Definiciones de estado*: si una regla o patrón depende de cierto contexto o cierta condición, es posible introducir estados e incorporarlos a dicha regla o patrón. Un estado es similar a lo que se conoce en la jerga de programación como flag o bandera.

2da) Sección de reglas o patrones:

El formato de esta está definido por el siguiente formato:

regla o patrón {acción}

El término correcto para referirnos a una regla o un patrón es "regular expression" (en inglés) lo que traducido sería "expresión regular". Es bueno conocer este término en caso de querer profundizar, la infinita cantidad de formas y variables de generar expresiones regulares. A continuación se dará una extensa lista pero aun así siempre habrá distintas formas y nuevas alternativas para generar el mismo resultado. Algo así como resolver un problema de lógica computacional por medio de varios algoritmos diferentes.

Operadores	Ejemplo	Explicación
[]	[a-z]	Marca un rango de caracteres. Si su primer carácter es un "^", entonces coincidirá con cualquier carácter fuera del rango.
*	[\n\t]*	Este operador indica que se va a coincidir con cadenas formadas por ninguna o varias apariciones del patrón que lo antecede.
+	[0-9]+	Todas las cadenas que se puedan formar, excepto cadenas vacías. En el ejemplo se aceptan a todos los números naturales y al cero.
.	.+	Este es una expresión regular que coincide con cualquier entrada excepto el retorno de carro ("\n").
{}	a{3,6}	Indica un rango de repetición cuando contiene dos

		números separados por comas.
?	-?[0-9]+	Indica que el patrón que lo antecede es opcional, es decir, puede existir o no.
	(- + -)?[0-9]+	Este hace coincidir, al patrón que lo precede o lo antecede y puede usarse consecutivamente.
" "	"bye"	Las cadenas encerradas entre " y " son aceptadas literalmente.
\	\.	Indica a lex que el caracter a continuación será tomado literalmente, como una secuencia de escape, este funciona para todos los caracteres reservados para lex y para C por igual.

Quizá toda esta información se vea mejor en un ejemplo. Ejemplo:

```
%%
([0-9])+                {printf("NumeroEntero");}
([0-9])+ "." ([0-9])*   {printf("NumeroDecimal");}
([a-zA-Z])              {printf("carácter");}
([a-zA-Z])+             {printf("Palabra");}
%%
```

Y este mismo código se puede escribir de otra forma si previamente en la sección de definiciones subsección *definiciones*, hacemos lo siguiente. Ejemplo:

```
DIGITO [0-9]
LETRA [a-zA-Z]
%%
{DIGITO}+                {printf("NumeroEntero");}
{DIGITO}+ "." {DIGITO}*   {printf("NumeroDecimal");}
{LETRA}                  {printf("carácter");}
{LETRA}+                 {printf("Palabra");}
%%
```

3ra) Sección de Código:

Tal como se dijo antes esta sección puede ser obviada. Pero en el caso que no sea así, debe estar la función main y dentro un llamado a la función yylex().

Un ejemplo mínimo podría ser:

```
int main (void)
{
    yylex();
    return 0;
}
```

Otros programas pueden definir la función yywrap(), y hasta incluso reabrir dentro de ella otro archivo de entrada yyin distinto al estándar input que es el teclado.

A continuación una breve tabla con funciones y variables para implementaciones.

Nombre	Explicación
int yylex(void)	Llamada para invocar al Analizador Léxico, devuelve tokens.
char *yytext	Puntero al string encontrado.
yylen	Longitud del string encontrado.
yyval	Valor asociado con el token.
int yywrap(void)	Cierra, retorna 1 si terminó y sino 0.
FILE *yyout	Archivo de salida.
FILE *yyin	Archivo de entrada.
ECHO	Escribe el string encontrado

1.2.3 Entradas/salidas

El programa FLEX necesita como archivo de entrada algún archivo con extensión ".l". La salida generada será el archivo lex.yy.c

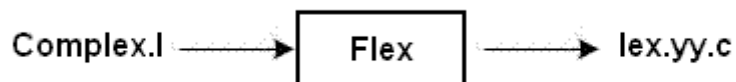


Figura 1

1.2.4 Ejemplos

A continuación se muestra una imagen, la cual hemos editado para tener una mejor comprensión del código, de la entrada y de la salida.

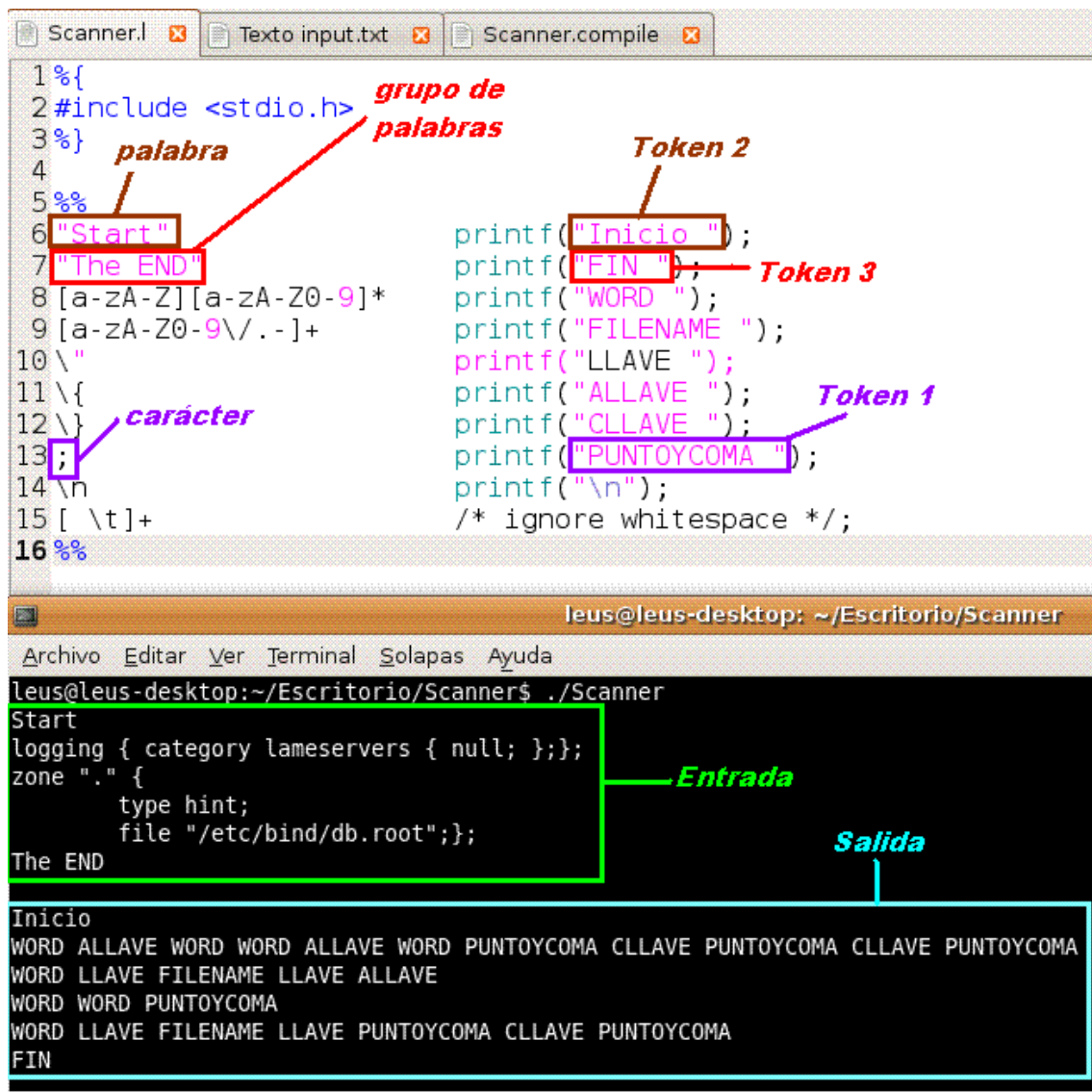


Figura 2

Se puede ver en la figura 2 que ante una cierta **Entrada**, el Scanner va recorriendo el texto ingresado y a medida que encuentra coincidencia con los patrones predefinidos los reemplazará por el token asociado en la **Salida**.

En el ejemplo: cada vez que se encuentra un ";" se reemplazará por el token "PUNTOYCOMA". En este caso el patrón es un solo carácter ";". Pero el patrón podría haber sido un conjunto de caracteres: ejemplos típicos podrían ser "main", "init" o "end". En nuestro ejemplo vemos como la palabra "Start" será sustituida por el token "Inicio". Y dado que el espacio también es un carácter podríamos generar patrones separados por espacios, podemos ver como la palabra "The END" será asociada por el token "FIN".

Finalmente podemos hacer un comentario sobre la línea 8 de nuestro código. Se define un patrón como `[a-zA-Z][a-zA-Z0-9]*` este es el patrón genérico asociado a FLEX para decir que considere cualquier palabra que empiece con una letra minúscula o mayúscula, seguida o no de letras/dígitos y el token que identificará este patrón será "WORD".

1.3 Parser

Una vez más utilizaremos, para el parser, una herramienta Open-Source llamada Bison. Es compatible casi al 100% con YACC, una herramienta clásica de Unix para generación de analizadores sintácticos, pero es un desarrollo diferente realizado por GNU bajo licencia GPL.

1.3.1 Definición

Empecemos por definir lo que significa la palabra "parser". En diccionario de Oxford encontramos la siguiente definición:

Parse /paarz/

- *verb 1 analyse (a sentence) into its component parts and describe their syntactic roles. 2*

Computing analyse (text) into logical syntactic components.

— *DERIVATIVES parser noun. — ORIGIN perhaps from Old French pars 'parts'.*

Por lo que se ve aquí se puede decir que "to parse sth" significaría "analizar sintácticamente algo". De este modo podemos entender que "parser" es el "analizador sintáctico". La sintaxis de un lenguaje de programación esta dada por la forma en la que se ingresan las instrucciones. Por ejemplo: todas las instrucciones deben terminar con punto y coma. Otra forma de definición de sintaxis puede ser la forma de separar los parámetros cuando se invoca a una función; o que la asignación de las variables sea de derecha a izquierda ejemplo "i=1"

1.3.2 Funcionamiento

La forma general de una gramática de un archivo fuente de Bison (*.y) es la siguiente:

```
%{  
  declaraciones en C  
}%  
Declaraciones de Bison  
%%  
Reglas gramaticales  
%%  
Código C adicional
```

Se podría decir que en este caso tenemos 4 secciones. Una de ellas son las declaraciones en C están delimitadas por "%{" y "%}". Mientras que "%%" hace el resto de las divisiones son signos de puntuación que aparecen en todo archivo de gramática de Bison para separar las secciones.

Las declaraciones en C pueden definir tipos y variables utilizadas en las acciones. Puede también usar comandos del preprocesador para definir macros, y utilizar `#include` para incluir archivos de `"*.h"` que realicen cualquiera de estas cosas.

Las Declaraciones de Bison declaran los nombres de los símbolos terminales y no terminales, y también podrían describir la precedencia de operadores y los tipos de datos de los valores semánticos de varios símbolos.

Las Reglas gramaticales son las producciones de la gramática, que además pueden llevar asociadas acciones, código en C, que se ejecutan cuando el analizador encuentra las reglas correspondientes.

El Código C adicional puede contener cualquier código C que desee utilizar. A menudo suele ir la definición del analizador léxico `yylex()`, más subrutinas invocadas por las acciones en las reglas gramaticales. En un programa simple, todo el resto del programa puede ir aquí.

Los símbolos terminales de la gramática en Bison, se denominan tokens y deben declararse en la sección de definiciones; en gral son los mismos tokens que se han definido previamente en el archivo del código fuente para Flex. Por convención se suelen escribir los tokens en mayúsculas y los símbolos no terminales en minúsculas. Una forma de escribir sibilos terminales es con el identificador `%token`.

Una Regla gramatical de Bison tiene la siguiente forma general:

```
resultado: componentes...
```

```
;
```

Siendo *resultado* el símbolo no terminal que describe esta regla y *componentes* son los diversos símbolos terminales y no terminales que forma esta regla. Por ejemplo:

```
exp: exp '+' exp
```

```
;
```

El ejemplo nos indica que dos agrupaciones de tipo `exp`, con un símbolo terminal de carácter `'+'` en el medio, puede combinarse en una agrupación mayor de tipo `exp`. Otra forma de escribir lo mismo hubiese sido:

```
exp: exp MAS exp
```

```
;
```

Es de notar que en este caso `MAS` sería un token previamente definido. Distribuidas en medio de los componentes pueden haber acciones que determinan la semántica de la regla. Normalmente hay una única {acción} que sigue a los componentes. Se pueden escribir por separado varias reglas para el mismo resultado o pueden unirse con el carácter de barra vertical `'|'` así:

```
resultado: componentes-regla1...
```

```
      | componentes-regla2...
```

```
      ...      {acción}
```

```
;
```

A continuación veremos cómo definir una secuencia separada por comas de cero o más agrupaciones exp:

```
expseq: /* vacío */  
      | expseq1  
      ;
```

```
expseq1: exp  
      | expseq1 ',' exp  
      ;
```

Una regla se dice recursiva cuando su no-terminal resultado aparezca también en su lado derecho. Casi todas las gramáticas de Bison hacen uso de la recursión, ya que es la única manera de definir una secuencia de cualquier número de cosas.

Una {acción} consiste en sentencias de C rodeadas por llaves. Se pueden situar en cualquier posición dentro de la regla. El código C en una acción puede hacer referencia a los valores semánticos de los componentes reconocidos por la regla con la construcción \$n. El valor semántico para la agrupación que se está construyendo es \$\$.

Aquí hay un ejemplo típico:

```
exp: ...  
   | exp '+' exp  
   { $$ = $1 + $3; }
```

En la acción, \$1 y \$3 hacen referencia a los valores semánticos de las dos agrupaciones exp componentes, que son el primer y tercer símbolo en el lado derecho de la regla. Si hubiese un valor semántico útil asociado con el token '+', debería hacerse referencia con \$2.

1.3.3 Entradas y salidas

Bison toma como archivos de entrada aquellos que tienen extensión "*.y". La salida generada serán dos archivos, una librería con un nombre de la forma "*.tab.h" y luego un archivo en código C con el nombre "*.tab.c"

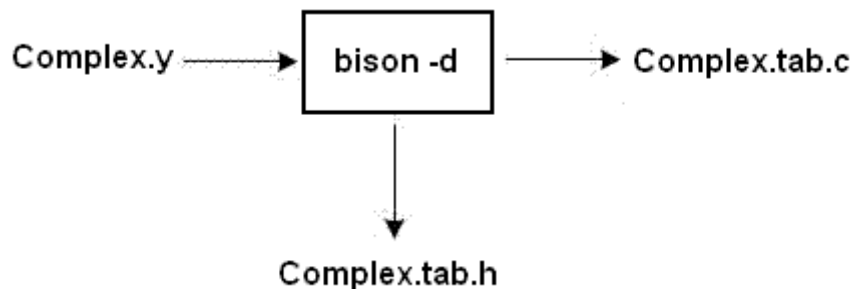
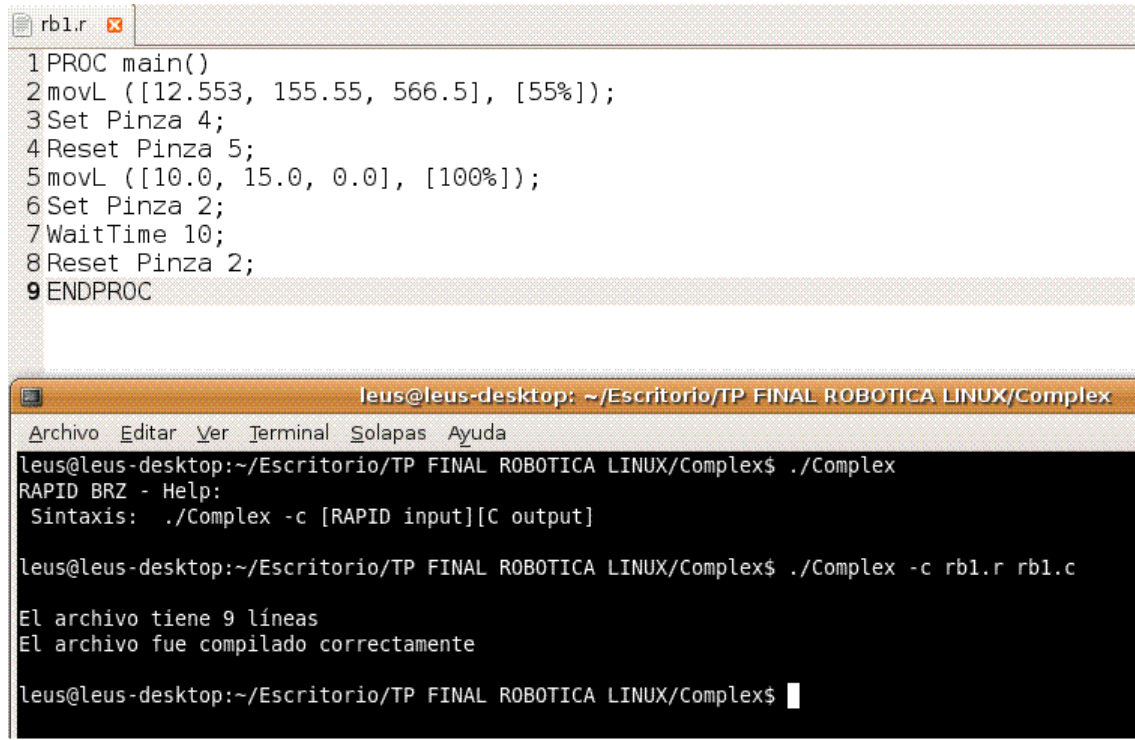


Figura 3

1.3.4 Ejemplos

En el ejemplo que se presenta a continuación vemos un ejemplo de un programa tipo rb1.r que está escrito de manera correcta, esto se puede ver en la salida que realiza ./Complex en la pantalla.



The screenshot shows a code editor window titled 'rb1.r' with the following content:

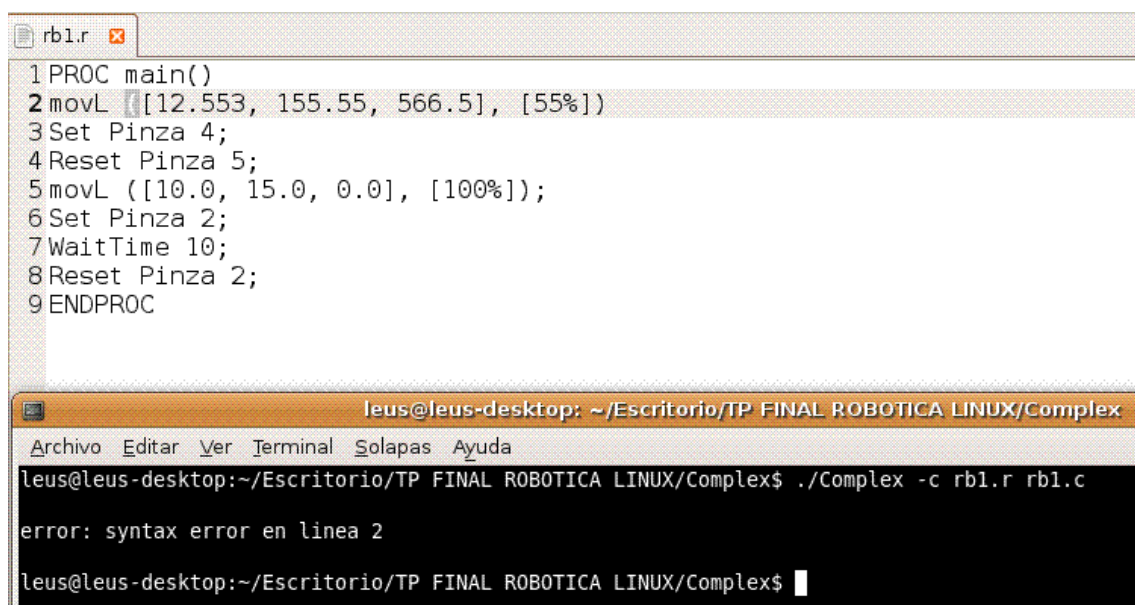
```
1 PROC main()  
2 movL ([12.553, 155.55, 566.5], [55%]);  
3 Set Pinza 4;  
4 Reset Pinza 5;  
5 movL ([10.0, 15.0, 0.0], [100%]);  
6 Set Pinza 2;  
7 WaitTime 10;  
8 Reset Pinza 2;  
9 ENDPROC
```

Below the editor is a terminal window titled 'leus@leus-desktop: ~/Escritorio/TP FINAL ROBOTICA LINUX/Complex'. The terminal shows the following commands and output:

```
leus@leus-desktop:~/Escritorio/TP FINAL ROBOTICA LINUX/Complex$ ./Complex  
RAPID BRZ - Help:  
Sintaxis: ./Complex -c [RAPID input][C output]  
  
leus@leus-desktop:~/Escritorio/TP FINAL ROBOTICA LINUX/Complex$ ./Complex -c rb1.r rb1.c  
  
El archivo tiene 9 líneas  
El archivo fue compilado correctamente  
  
leus@leus-desktop:~/Escritorio/TP FINAL ROBOTICA LINUX/Complex$
```

Figura 4

A continuación hemos modificado el archivo rb1.r y en la línea 2 quitamos el punto y coma de finalización de la instrucción. La salida de ./Complex detectará este error.



The screenshot shows the same code editor window as in Figure 4, but with a modification to line 2:

```
1 PROC main()  
2 movL [[12.553, 155.55, 566.5], [55%]]  
3 Set Pinza 4;  
4 Reset Pinza 5;  
5 movL ([10.0, 15.0, 0.0], [100%]);  
6 Set Pinza 2;  
7 WaitTime 10;  
8 Reset Pinza 2;  
9 ENDPROC
```

The terminal window below shows the command and the resulting error message:

```
leus@leus-desktop:~/Escritorio/TP FINAL ROBOTICA LINUX/Complex$ ./Complex -c rb1.r rb1.c  
  
error: syntax error en linea 2  
  
leus@leus-desktop:~/Escritorio/TP FINAL ROBOTICA LINUX/Complex$
```

Figura 5

1.4 Compilador realizado (Complex)

El compilador diseñado durante la tesis fue llamado Complex. Complex trabajo sobre la línea de comandos de DOS o de Linux y compila archivos .r de lenguaje RAPID a archivos .C de targets genéricos, para ser compilados por CodeWarrior, Keil, GCC, GCC++, etc. Un ejemplo del mismo se puede observar en la Figura 5.

1.4.1 Idea Conceptual

La idea de generar un compilador para nuestro robot, surge del siguiente problema. Nosotros somos desarrolladores de tecnología pero si se quiere hacer un robot para lanzar al mercado y que cualquiera pueda hacer uso de el, no se debería exigirle al usuario tener conocimientos de microcontroladores, programación en C, FPGA y conocimientos de robótica.

En base a esta necesidad, vemos que es imperioso desarrollar un tipo de lenguaje intuitivo para comandar el robot sin estar al lado de un especialista.

1.4.2 Bases

La base del diseño es lograr un puente entre la arquitectura de programación de alto nivel de RAPID con los códigos .C generalmente usados en targets (uC) con tecnologías como la DSP u de otro tipo. De esta forma no sólo se le simplifica al usuario el uso sino que se homologa el lenguaje de programación usado hoy en día en el setting up de manipuladores de campo. Complex también es compatible con la interpretación automática de usos de "Beans" y de esa forma puede traducir lenguaje RAPID a Beans del target conocidos.

1.4.3 Código

- complex.l

```
%{
#include <stdio.h>
#include <string.h>
#include "Complex.tab.h"
extern int linenumber;
%}
digito                                [0-9]
%%
generic_command                      return GENERIC_CMD;
movL                                 return MOVL_CMD;
Set                                  return SET_CMD;
Reset                                return RESET_CMD;
WaitTime                             return WAITTIME_CMD;
Pinza                                return PINZA;
CONS                                  return CONS_VAR;
VAR                                   return VAR_VAR;
PERS                                  return PERS_VAR;
"PROC main()"                        return PROMPT_START;
ENDPROC                              return PROMPT_FINISH;
-?(digito)+                          (yyval.entero = atoi(yytext); return INT;
/el "?" sirve para preguntar si el simbolo que lo antecede "-"
```

```

//puede ir o no
}
-?{digito}+ "." {digito}* {yyval.flotante = atof(yytext); return FLOAT;
//-?{digito}+ "." {digito}* si se hubiese escrito asi se prodria
//tomar como numero flotante 268.
// en cambio con -?{digito}+ "." {digito}+ es valido 215.695
}
[a-zA-Z] {yyval.string=strdup(yytext);return LETRA;
[a-zA-Z][a-zA-Z0-9]+ {yyval.string=strdup(yytext);return WORD;
//cualquier palabra o letra que empiece con minuscuala o mayuscula
//seguida de una o varias letras o de un o varios digitos

"(" return OPAR;
\) {return CPAR;
//notar que tambien sirce esta nomenclatura

"[" return OCOR;
"]" return CCOR;
"," return COMMA;
"%" return PERCENTAGE;
; {return SEMICOLON;
//para los caracteres especiales se puede no poner nada

\n {linenumber++;
/* ignoro el enter. Se puede contar las lineas debo poner un
contador. Ver como se lo paso a BISON */
}

[\t]+ /* ignoro los espacios, los tabs o cualquier combincacion de ellos*/;
%%

```

- complex.y

```

%{
#include <stdio.h>
#include <string.h>

#define SPECIAL_CHAR '{'

int linenumber = 0;
char Start=0, Success=1;
FILE *fp_output, *fp_preforma;

void yyerror(const char *str)
{
    fprintf(stderr, "\nerror: %s en linea %d\n\n", str, linenumber);
    Success = 0;
}

int yywrap()
{
    return 1;
}

int main(int argc, char *argv[])
{
    char filename_input[30];
    char filename_output[30];
    char filename_preforma[30];

    extern FILE *yyin;
    size_t len;

    char cTemp = 0;
    int iSpecialCharCounter = 0;

    switch(argc)
    {
        default:
        case 1:
            printf("RAPID BRZ - Help:\n Sintaxis: ./Complex -c [RAPID input][C output]\n\n");
            break;
    }
}

```

```

case 4:
if(argv[1][1]=='c' && argv[1][0]!='.')
{
sprintf(filename_input,"%s",argv[2]);
sprintf(filename_output,"%s",argv[3]);
sprintf( filename_preforma, "Template Robotica.c");
if(!(yyin=fopen(filename_input,"r"))) //FILE pointer dedicado del flex para el archivo a scannear...
{
printf("Imposible abrir archivo %s para lectura",filename_input);
}
else
{
if( !( fp_output = fopen( filename_output, "w" ) ) )
{
printf("Imposible abrir archivo %s para escritura\n",filename_output);
}
else
{
if( !(fp_preforma = fopen( filename_preforma, "r" ) ) )
{
printf( "Imposible abrir archivo de preforma\n" );
}
else
{
//Primera parte del archivo de preforma
cTemp = getc( fp_preforma );
while( iSpecialCharCounter < 2 )
{
fputc( cTemp, fp_output );
if( cTemp == SPECIAL_CHAR )
iSpecialCharCounter++;
cTemp = fgetc( fp_preforma );
}
yyparse(); //Trabaja el Bison....
// Aca se puede escribir con fprintf en el archivo
// Ultima parte del archivo de preforma
cTemp = fgetc( fp_preforma );
while( !feof( fp_preforma ) )
{
fputc( cTemp, fp_output );
cTemp = fgetc( fp_preforma );
}
if(Success == 1)
{
printf("\nEl archivo tiene %d líneas\n",linenumber);
printf("El archivo fue compilado correctamente\n\n");
}
fclose( fp_preforma );
fclose(yyin); //En Simple.y no lo cierra...
fclose( fp_output );
}
}
}

else
{
printf("ERROR: Parametros mal utilizados");
}
break;
}

return 0;
}

```

```
char *heater="default";
```

```
%}
```

```
%token GENERIC_CMD MOVL_CMD SET_CMD RESET_CMD WAITTIME_CMD
```

```

%token OPAR CPAR COMMA SEMICOLON OCOR CCOR PERCENTAGE PINZA CONS_VAR VAR_VAR PERS_VAR
%token PROMPT_FINISH PROMPT_START

%union
{
    int entero;
    float flotante;
    char *string;
    int nrolinea;
}

%token <entero> INT
%token <flotante> FLOAT
%token <string> WORD
%token <string> LETRA
%%

commands:
    |
    commands command
    ;

command:
    generic_cmd | movL_cmd | set_cmd | reset_cmd | prompt_start | prompt_finish | wait_time

generic_cmd:
    GENERIC_CMD OPAR INT COMMA FLOAT COMMA WORD COMMA LETRA CPAR SEMICOLON
    {
        if(!Start)
        {
            yyerror("PROC main () no encontrado");
        }
        printf("Comando generico para testeo recibe:\n");
    }
    ;

movL_cmd:
    MOVL_CMD OPAR OCOR FLOAT COMMA FLOAT COMMA FLOAT CCOR COMMA OCOR INT PERCENTAGE
    CCOR CPAR SEMICOLON
    {
        if(!Start)
        {
            yyerror("PROC/ENDPROC no respetado");
        }
        if($12>100)
        {
            yyerror("Porcentaje de velocidad mayor a 100 %");
        }
        fprintf(fp_output, "\n\t PWM(cin_inverse (%4.2f,%4.2f,%4.2f), %d);\n", $4,$6,$8,$12 );
    }
    ;

set_cmd:
    SET_CMD PINZA INT SEMICOLON
    {
        if(!Start)
        {
            yyerror("PROC/ENDPROC no respetado");
        }
        fprintf(fp_output, "\n\t P1.%d = 1;\n", $3);
    }
    ;

reset_cmd:
    RESET_CMD PINZA INT SEMICOLON
    {
        if(!Start)
        {

```



```
        yyerror("PROC/ENDPROC no respetado");
    }
    fprintf( fp_output, "\n\t P1.%d = 0;\n", $3);
}
;

wait_time:
    WAITTIME_CMD INT SEMICOLON
    {
        if(!Start)
        {
            yyerror("PROC/ENDPROC no respetado");
        }
        fprintf( fp_output, "\n\t Sleep(%d000);\n", $2);
    }
;

prompt_start:
    PROMPT_START
    {
        Start=1;
    }
;

prompt_finish:
    PROMPT_FINISH
    {
        Start=0;
    }
;
;
```

1.4.4 Highlights

- Manejo por línea de comandos
- Soporte de ayuda en el mismo compilador
- Acopable a makefiles o procesos por lotes.

1.4.5 Funcionamiento y Ejemplos.

El funcionamiento es simple ya que se explicó con anterioridad como funcionan los lex y los yacc (Scanners/Parsers). Se recomienda utilizar la ayuda de Complex que se ejecuta con ejecutar el comando sin parámetros adicionales en un Terminal. Sírvese como ejemplos los screenshots que se muestran en la Figura 4 y 5.

2. Embedded

2.1 Esquema General.

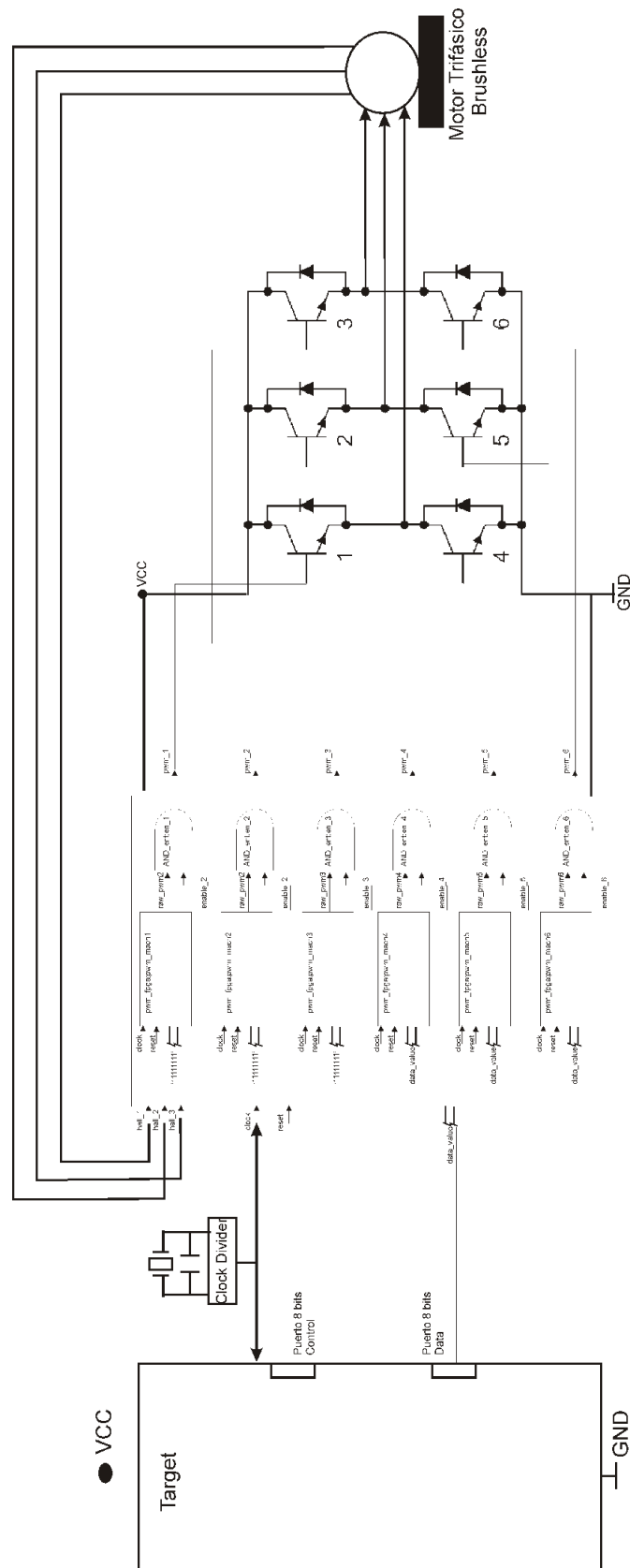


Figura 6

El esquema general que utilizamos está basado en lo que aprendimos durante el desarrollo del TP2 de la cursada. En la Figura 6 se muestra un esquema general donde conectamos un target, el cuál podría ser compatible con tecnología DSP (como el estudiado en el TP1), un FPGA diseñado por nosotros (compatible con las necesidades planteadas) y un driver de potencia para motor trifásico brushless. A continuación se describen las entradas / salidas de cada módulo. Esta configuración debe ser repetida 'n' cantidad de veces para 'n' motores, ya que la misma esta pensada y diseñada para comandar un solo motor.

2.2 Entradas/Salidas del sistema.

- 2.2.1 Target (uC)

Se considera que para el uso de un FPGA y el manejo respectivo de un motor vamos a utilizar un puerto de 8 bits, que será la salida lógica de valores a inyectar al PWM. El rango de valores aceptados queda inscripto entre 0 a 255 correspondiendo a 0 el 0% de velocidad y el 255 al 100% de velocidad. Para el manejo de más ejes se pueden manejar tantos puertos de 8 bits como ejes se tengan, en nuestro caso serán 6 y eso equivale a 6 puertos de 8 bits de datos. Cabe acotar que utilizaremos un puerto mas con salidas de control bit a bit, que se utilizaran para resetear los módulos PWM (FPGA).

- 2.2.2 FPGA (Módulo controlador del driver)

Este módulo tiene como entradas un puerto de 8 bits digital indicando el valor de velocidad a imprimir en el motor. También se manejan señales de reset y de clock pero lo más importante de todo son las entradas hall_1, hall_2, hall_3. Cada una de estas entradas esta designada para las salidas de los sensores de efecto hall que detectan la posición del rotor y hacen al sincronismo del movimiento del motor y la excitación de las bobinas del mismo.

A su vez el módulo tiene como salidas seis señales nombradas convenientemente como pwm_1 a pwm_6; cada una de ellas es la señal lógica con el ancho de pulso modulado para ser conectadas en las bases de los transistores del driver o en los gates de los MOSFET del mismo.

- 2.2.3 Puente H y Motor Brushless trifásico

Las entradas del puente H son las seis señales lógicas con la modulación de ancho de pulso impuesta por el sistema lógico predecesor al puente. El puente interpreta a estas señales como cortes y saturaciones de las llaves de estado sólido. Cada una de estas llave genera la excitación necesaria para que energizar las tres bobinas del motor, en sincronía con el movimiento del rotor. Se mide la posición del rotor con los sensores de efecto Hall, que son las salidas de la etapa.

2.3 VHDL

VHDL es el acrónimo que representa la combinación de **VHSIC** y **HDL**, donde VHSIC es el acrónimo de *Very High Speed Integrated Circuit* y HDL es a su vez el acrónimo de *Hardware Description Language*.

Es un **lenguaje** usado por ingenieros definido por el **IEEE** (*Institute of Electrical and Electronics Engineers*) (ANSI/IEEE 1076-1993) que se usa para **diseñar circuitos** digitales. Otros métodos para diseñar circuitos son la captura de esquemas (con herramientas **CAD**) y los diagramas de bloques, pero éstos no son prácticos en diseños complejos. Otros lenguajes para el mismo propósito son **Verilog** y **ABEL**.

Aunque puede ser usado de forma general para describir cualquier circuito se usa principalmente para programar PLD (*Programmable Logic Device* - Dispositivo Lógico Programable), **FPGA** (*Field Programmable Gate Array*), **ASIC** y similares.

(Fuente: Wiki)

2.3.1 Introducción

Es importante entender el ambiente de diseño del VHDL antes de entrar en el lenguaje. En un procedimiento de diseño (flujo de diseño) basado en VHDL hay varios pasos a seguir.

El primer paso del diseño consiste en la construcción del diagrama en bloque del sistema. Al igual que en el desarrollo de software, en el diseño de VHDL también hay cierta jerarquía que se debe ser respetada. VHDL ofrece un buen marco de trabajo para definir los módulos que integran el sistema y sus interfaces, dejando los detalles para pasos posteriores.

El segundo paso es la elaboración del código en VHDL para cada módulo, para sus interfaces y sus detalles internos. Como el VHDL es un lenguaje basado en texto, se puede utilizar cualquier editor para esta tarea, aunque el entorno de los programas de VHDL incluye su propio editor de texto. Después que se ha escrito algún código se hace necesario compilarlo. El compilador de VHDL analiza este código y determina los errores de sintaxis y chequea la compatibilidad entre módulos. Crea toda la información necesaria para la simulación. El próximo paso es la simulación, el cual le permite establecer los estímulos a cada módulo y observar su respuesta. El VHDL da la posibilidad de crear bancos de prueba que automáticamente aplica entradas y compara las salidas con las respuestas deseadas. La simulación es un paso dentro del proceso de verificación. El propósito de la simulación es verificar que el circuito trabaja como se desea, es más que comparar entradas y salidas. En proyectos complejos se hace necesario invertir un gran tiempo en generar pruebas que permitan evaluar el circuito en un amplio rango de operaciones de trabajo. Encontrar errores en este paso del diseño es mejor que al final, en donde hay que repetir entonces una gran cantidad de pasos del diseño. Hay dos dimensiones a verificar:

- Su comportamiento funcional en donde se estudia su comportamiento lógico independiente de consideraciones de tiempo como las demoras en las compuertas.
- Su verificación en el tiempo, en donde se incluye las demoras de las compuertas y otras consideraciones de tiempo como los tiempos de establecimiento (set-up time) y los tiempos de mantenimiento (hold time).

Después de la verificación se está listo para entrar en la fase final del diseño. La naturaleza y herramientas en esta fase dependen de la tecnología, pero hay tres pasos básicos. El primero es la síntesis que convierte la descripción en VHDL en un conjunto de componentes que pueden ser realizados en la tecnología seleccionada. Por ejemplo con PLD se generan las ecuaciones en suma de productos. En ASIC genera una lista de compuertas y un netlist que especifica como estas compuertas son interconectadas. El diseñador puede ayudar a la herramienta de síntesis especificando requerimientos a la tecnología empleada, como el máximo número de niveles lógicos o la capacidad de salida que se requiere. En el siguiente paso de ajuste (fitting) los componentes se ajustan a la capacidad del dispositivo que se utiliza. Para PLD esto significa que acopla las ecuaciones obtenidas con los elementos AND – OR que dispone el circuito. Para el caso de ASIC se dibujarían las compuertas y se definiría como conectarlas. En el último paso se realiza la verificación temporal ya que a esta altura es que se pueden calcular los elementos parásitos como las capacidades de las conexiones. Como en cualquier otro proceso creativo puede ser que ocasionalmente se avance dos pasos hacia delante y uno hacia atrás (o peor).

Estructura de programa

VHDL fue diseñado en base a los principios de la programación estructurada. La idea es definir la interfaz de un módulo de hardware mientras deja invisible sus detalles internos. La entidad (ENTITY) en VHDL es simplemente la declaración de las entradas y salidas de un módulo mientras que la arquitectura (ARCHITECTURE) es la descripción detallada de la estructura interna del módulo o de su comportamiento. (§4)

2.3.2 Código utilizado en el FPGA

Se procede a transcribir el código en lenguaje VHDL utilizado dentro del FPGA, que describe las arquitecturas implementadas y sus respectivos comportamientos para complementar las entradas con las salidas.

Código Implementado

```
library IEEE;  
USE      ieee.std_logic_1164.all;  
USE      ieee.std_logic_arith.all ;  
USE      work.user_pkg.all;
```

```
ENTITY ModuloDriver IS  
PORT ( clock,reset           :in STD_LOGIC;  
       hall_1                :in STD_LOGIC;
```

```

        hall_2                :in STD_LOGIC;
        hall_3                :in STD_LOGIC;
        data_value            :in std_logic_vector(7 downto 0);
        pwm_1                 :out STD_LOGIC;
        pwm_2                 :out STD_LOGIC;
        pwm_3                 :out STD_LOGIC;
        pwm_4                 :out STD_LOGIC;
        pwm_5                 :out STD_LOGIC;
        pwm_6                 :out STD_LOGIC;

    );

END ModuloDriver;

ARCHITECTURE behv1 OF ModuloDriver IS

    SIGNAL enable_1           : STD_LOGIC;
    SIGNAL enable_2           : STD_LOGIC;
    SIGNAL enable_3           : STD_LOGIC;
    SIGNAL enable_4           : STD_LOGIC;
    SIGNAL enable_5           : STD_LOGIC;
    SIGNAL enable_6           : STD_LOGIC;

    SIGNAL raw_pwm_1          : STD_LOGIC;
    SIGNAL raw_pwm_2          : STD_LOGIC;
    SIGNAL raw_pwm_3          : STD_LOGIC;
    SIGNAL raw_pwm_4          : STD_LOGIC;
    SIGNAL raw_pwm_5          : STD_LOGIC;
    SIGNAL raw_pwm_6          : STD_LOGIC;

    SIGNAL all_ones           :STD_LOGIC_VECTOR(7 downto 0);

    component pwm_fpga
    PORT ( clock,reset        :in STD_LOGIC;
          Data_value         :in std_logic_vector(7 downto 0);
          pwm                 :out STD_LOGIC
    );
    end component;

    component AND_ent
    port(      x: in std_logic;
             y: in std_logic;
             F: out std_logic
    );
    end component;

BEGIN

    en_1: AND_ent port map (enable_1,raw_pwm_1,pwm_1);
    en_2: AND_ent port map (enable_2,raw_pwm_2,pwm_2);
    en_3: AND_ent port map (enable_3,raw_pwm_3,pwm_3);
    en_4: AND_ent port map (enable_4,raw_pwm_4,pwm_4);
    en_5: AND_ent port map (enable_5,raw_pwm_5,pwm_5);
    en_6: AND_ent port map (enable_6,raw_pwm_6,pwm_6);
    pwm_1_mach: pwm_fpga port map (clock,reset,all_ones,raw_pwm_1);
    pwm_2_mach: pwm_fpga port map (clock,reset,all_ones,raw_pwm_2);
    pwm_3_mach: pwm_fpga port map (clock,reset,all_ones,raw_pwm_3);
    pwm_4_mach: pwm_fpga port map (clock,reset,data_value,raw_pwm_4);
    pwm_5_mach: pwm_fpga port map (clock,reset,data_value,raw_pwm_5);
    pwm_6_mach: pwm_fpga port map (clock,reset,data_value,raw_pwm_6);

    init_task: PROCESS (reset)
    begin
        all_ones <="00000001";
    END PROCESS init_task;

```

```
task_1: PROCESS (hall_1,hall_2,hall_3)
```

```
    BEGIN
```

```
    if(hall_1 = '1') THEN
```

```
        enable_1 <= '1' ;  
        enable_4 <= '1' ;  
        enable_2 <= '0' ;  
        enable_5 <= '0' ;  
        enable_3 <= '0' ;  
        enable_6 <= '0' ;
```

```
    END IF;
```

```
    if(hall_2 = '1') THEN
```

```
        enable_1 <= '0' ;  
        enable_4 <= '0' ;  
        enable_2 <= '1' ;  
        enable_5 <= '1' ;  
        enable_3 <= '0' ;  
        enable_6 <= '0' ;
```

```
    END IF;
```

```
    if(hall_3 = '1') THEN
```

```
        enable_1 <= '0' ;  
        enable_4 <= '0' ;  
        enable_2 <= '0' ;  
        enable_5 <= '0' ;  
        enable_3 <= '1' ;  
        enable_6 <= '1' ;
```

```
    END IF;
```

```
    END PROCESS task_1;
```

```
END behv1;
```

2.3.3 TestBench utilizado

La generación de FPGA's requiere de testings exhaustivos debido a que el programa generado en VHDL no es un programa de SW sino de HW. Una vez generado el HW no deben aparecer bugs ya que, obviamente no es flexible a modificaciones on-the-fly (sobre la marcha). Por esta razón, las compañías de desarrollo de FPGA's hacen mucho hincapié en sus plataformas de desarrollo (SDK), complementándolas con debuggers gráficos que permiten ver las señales lógicas internas, las señales de entrada y de salida, por ejemplo.

Para realizar el debug correspondiente, la arquitectura desarrollada es incluida dentro de una arquitectura más grande que la instancia; esta arquitectura es el *Testbench*. El comportamiento del Testbench es el conjunto de todos los casos que uno desea analizar y para ello uno diseña las excitaciones de entrada que se deseen analizar para observar cómo responden las salidas. Este método de debug es conocido en la industria del Software como "inspección de caja negra".

Durante el desarrollo utilizamos un SW de simulación de entornos lógicos (ModelSIM) de la empresa Mentor Graphics. A continuación se transcribe la arquitectura del testbench implementada y el comportamiento de análisis.

Código Implementado

```

LIBRARY ieee;

use ieee.std_logic_1164.all;

ENTITY pwm_fpga_test_bench IS

END pwm_fpga_test_bench;

ARCHITECTURE arch_test_bench OF pwm_fpga_test_bench IS

COMPONENT ModuloDriver
PORT (
    clock: in std_logic;
    reset: in std_logic;
    hall_1           :in STD_LOGIC;
    hall_2           :in STD_LOGIC;
    hall_3           :in STD_LOGIC;
    data_value: in std_logic_vector(7 downto 0);
    pwm_1: out std_logic;
    pwm_2: out std_logic;
    pwm_3: out std_logic;
    pwm_4: out std_logic;
    pwm_5: out std_logic;
    pwm_6: out std_logic
);
END COMPONENT;

-- Internal signal declaration
SIGNAL sig_clock      : std_logic;
SIGNAL sig_reset      : std_logic;
SIGNAL sig_data_value : std_logic_vector(7 downto 0);
SIGNAL sig_pwm_1 :std_logic;
SIGNAL sig_pwm_2 :std_logic;
SIGNAL sig_pwm_3 :std_logic;
SIGNAL sig_pwm_4 :std_logic;
SIGNAL sig_pwm_5 :std_logic;
SIGNAL sig_pwm_6 :std_logic;
SIGNAL sig_hall_1 :std_logic;
SIGNAL sig_hall_2 :std_logic;
SIGNAL sig_hall_3 :std_logic;

shared variable ENDSIM: boolean:=false;
constant clk_period:TIME:=100 ns;

BEGIN
clk_gen: process

    BEGIN

    IF ENDSIM = FALSE THEN
        sig_clock <= '1';
        wait for clk_period/2;
        sig_clock <= '0';
        wait for clk_period/2;
    else
        wait;
    end if;
end process;

hall_gen: process

    BEGIN
    IF ENDSIM = FALSE THEN
        sig_hall_1<= '1';
        sig_hall_2<= '0';
        sig_hall_3<= '0';
        wait for 338 us;

```



```

        sig_hall_1<= '0';
        sig_hall_2<= '1';
        sig_hall_3<= '0';
        wait for 338 us;
        sig_hall_1<= '0';
        sig_hall_2<= '0';
        sig_hall_3<= '1';
        wait for 338 us;
    else
        wait;
    end if;
end process;

inst_pwm_fpga : ModuloDriver
PORT MAP(
    clock      => sig_clock,
    reset      => sig_reset,
    hall_1     => sig_hall_1,
    hall_2     => sig_hall_2,
    hall_3     => sig_hall_3,
    data_value => sig_data_value,
    pwm_1      => sig_pwm_1,
    pwm_2      => sig_pwm_2,
    pwm_3      => sig_pwm_3,
    pwm_4      => sig_pwm_4,
    pwm_5      => sig_pwm_5,
    pwm_6      => sig_pwm_6
);

stimulus_process: PROCESS

BEGIN

    If ENDSIM = FALSE THEN
        sig_reset <= '1';
        wait for 100 ns;
        sig_reset <= '0';
        sig_data_value <= "11111000";
        wait for 26 us;
        sig_data_value <= "11011111";
        wait for 26 us;
        sig_data_value <= "01111111";
        wait for 26 us;
        sig_data_value <= "01000000";
        wait for 26 us;
        sig_data_value <= "00100000";
        wait for 26 us;
        sig_data_value <= "00010000";
        wait for 26 us;
        sig_data_value <= "00001000";
        wait for 26 us;
        sig_data_value <= "00010000";
        wait for 26 us;
        sig_data_value <= "00100000";
        wait for 26 us;
        sig_data_value <= "01000000";
        wait for 26 us;
        sig_data_value <= "01111111";
        wait for 26 us;
        sig_data_value <= "11001111";
        wait for 26 us;
        sig_data_value <= "11111000";
        wait for 26 us;

    else
        wait;
    end if;

END PROCESS stimulus_process;
END arch_test_bench;

```

2.3.4 Simulación en ModelSim

Referencias

sig_clock = Clock de excitación.

sig_reset=Reset del driver.

sig_data= 8 bits indicando la velocidad propuesta para el sistema.

sig_pwm_x= Salida individual de cada pwm del driver, listo para conectarse con el motor trifásico.

sig_hall_x= Entrada individual de los sensores de efecto de hall para detectar la posición del rotor.

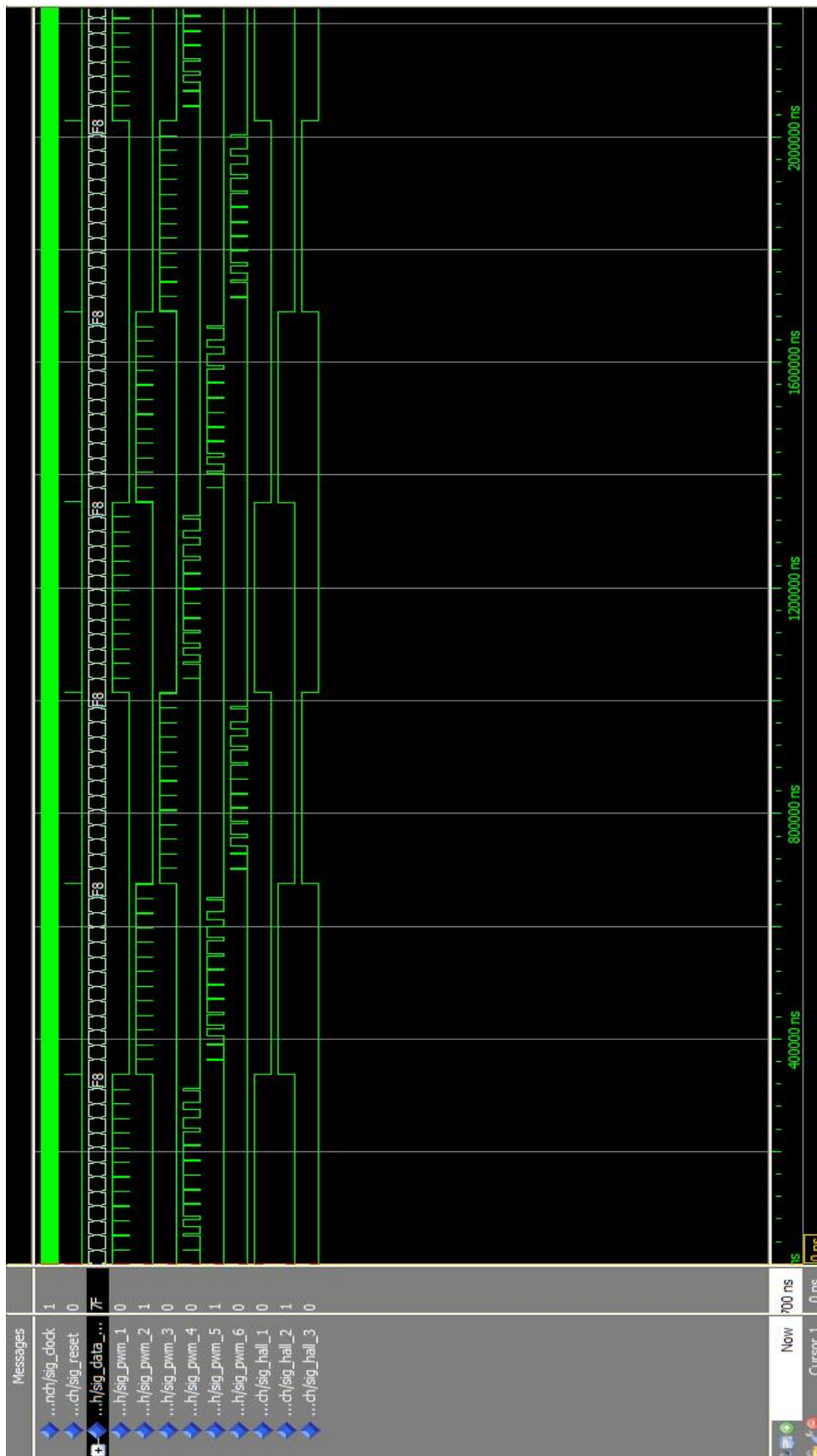


Figura 7

2.4 Control de Potencia

A partir de esta sección se hablará de una etapa muy importante a la hora de controlar cualquier dispositivo dotado de motores y es la denominada Control de Potencia. El objetivo principal de la misma es la de poder controlar a los motores que forman al Robot para realizar el movimiento deseado, también corregir momento a momento la posición para que el objetivo del movimiento se cumpla. Y por último también controlar los parámetros eléctricos para no dañar a los distintos componentes del motor.

2.4.1 Puente H

Para hacer el control de movimiento el Control de Potencia consta de dos etapas principales. La primera es la encargada de realizar el control de señales que envía al Puente H y a la vez recibe las señales que sirven de Feedback para saber la posición del Motor. De esta manera esta etapa corrige momento a momento la posición del motor modificando las señales del Puente H. La segunda etapa es la llamada Puente H y es la que se describirá con mayor detalle en esta sección, esta etapa es la que se encarga de controlar directamente al Motor a través de su interfaz de potencia que permite accionar las distintas posiciones del motor de continua.

2.4.1.1 Esquema

En el siguiente esquema se pueden observar las partes principales de un controlador de motor. Como ya se ha mencionado anteriormente el Control de Potencia consta de una lógica que controla las señales del Puente H, así como también un microcontrolador/procesador que da las órdenes de mayor jerarquía.

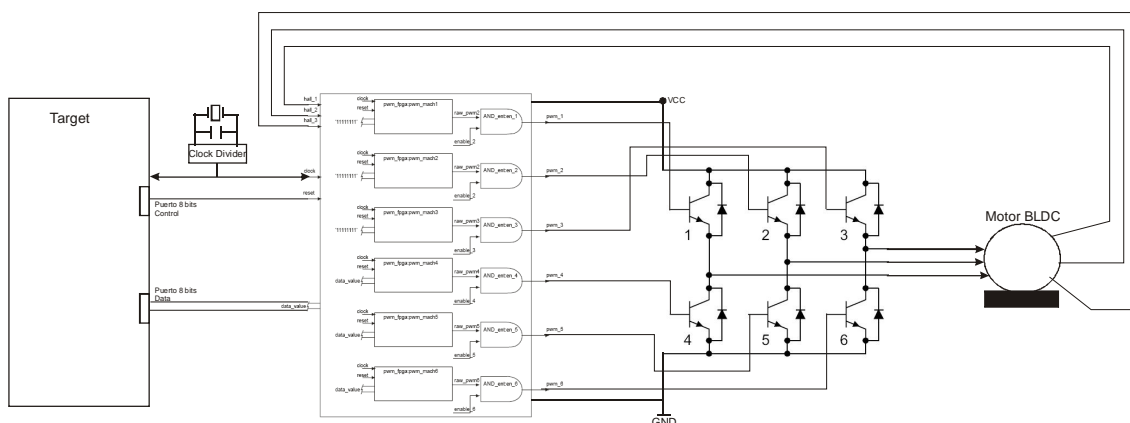


Figura 8

En el segundo esquema se puede diferenciar con mayor claridad los distintos bloques que componen al sistema y como se ha mencionado anteriormente se puede

ver el controlador lógico o Drive controller al motor, la carga y el puente H que hace de interfaz de potencia.

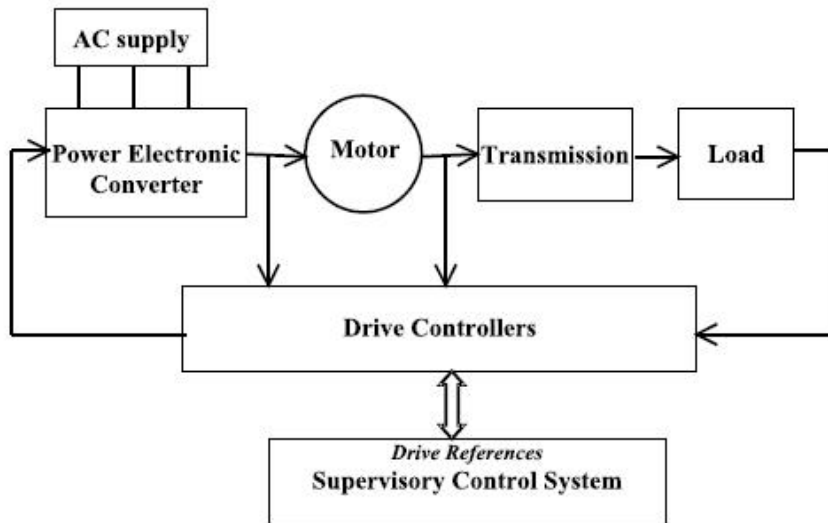


Figura 9

2.4.1.2 Modo de funcionamiento

El puente H es una interfaz de potencia utilizada para controlar el movimiento y velocidad del motor, básicamente para generar los distintos movimientos se elige una combinación de transistores para ponerlos en conducción y así generar una corriente que vaya en un sentido y en otro, eso produce el movimiento del motor en distintos sentidos.

El circuito Puente H sólo permite un funcionamiento SÍ-NO del motor, a plena potencia en un sentido o en el otro (además del estado de detención, por supuesto), pero no ofrece un modo de controlar la velocidad. Si es necesario hacerlo, se puede apelar a la regulación del voltaje de la fuente de alimentación, variando su potencial de 7,2 V hacia abajo para reducir la velocidad. Esta variación de tensión de fuente produce la necesaria variación de corriente en el motor y, por consiguiente, de su velocidad de giro. Es una solución que puede funcionar en muchos casos, pero se trata de una regulación primitiva, que podría no funcionar en aquellas situaciones en las que el motor está sujeto a variaciones de carga mecánica, es decir que debe moverse aplicando fuerzas diferentes. En este caso es muy difícil lograr la velocidad deseada cambiando la corriente que circula por el motor, ya que ésta también será función — además de serlo de la tensión eléctrica de la fuente de alimentación— de la carga mecánica que se le aplica (es decir, de la fuerza que debe hacer para girar).

Una de las maneras de lograr un control de la velocidad es tener algún tipo de realimentación, es decir, algún artefacto que permita medir a qué velocidad está girando el motor y entonces, en base a lo medido, regular la corriente en más o en menos. Este tipo de circuito requiere algún artefacto de sensado (sensor) montado sobre el eje del motor. A este elemento se le llama tacómetro y suele ser un generador de CC (otro motor de CC cumple perfectamente la función, aunque podrá ser uno de mucho menor potencia), un sistema de tacómetro digital óptico, con un disco de

ranuras o bandas blancas y negras montado sobre el eje, u otros sistemas, como los de pickups magnéticos.

2.4.1.3 Oscilogramas

Como se puede ver en la siguiente oscilograma, cada uno de los motores consta de un par de accionadores que hacen de secuencia para que se genere una curva determinada de velocidad, aceleración y por ende posición. En este ejemplo se ven, combinación de los actuadores, velocidad angular ω y corriente del motor.

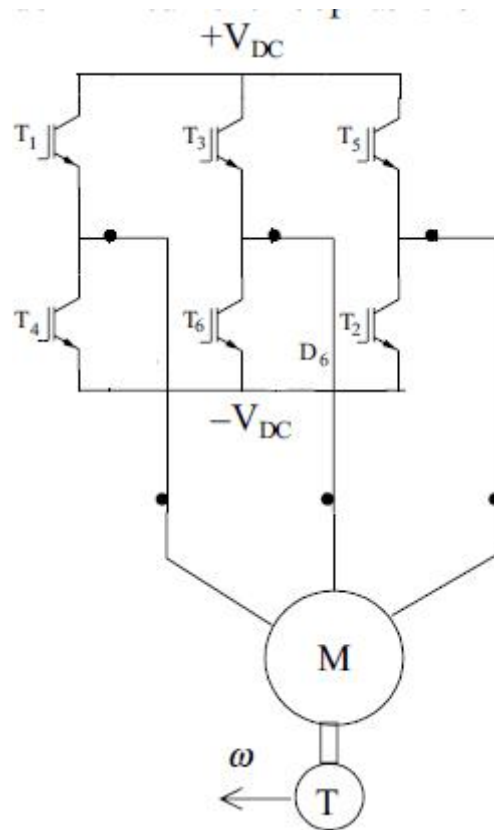


Figura 10

Teniendo en cuenta la figura 9, si se desea tomar una curva de velocidad determinada se deberán tomar las siguientes combinaciones de pulsos para activar a los pares de transistores.

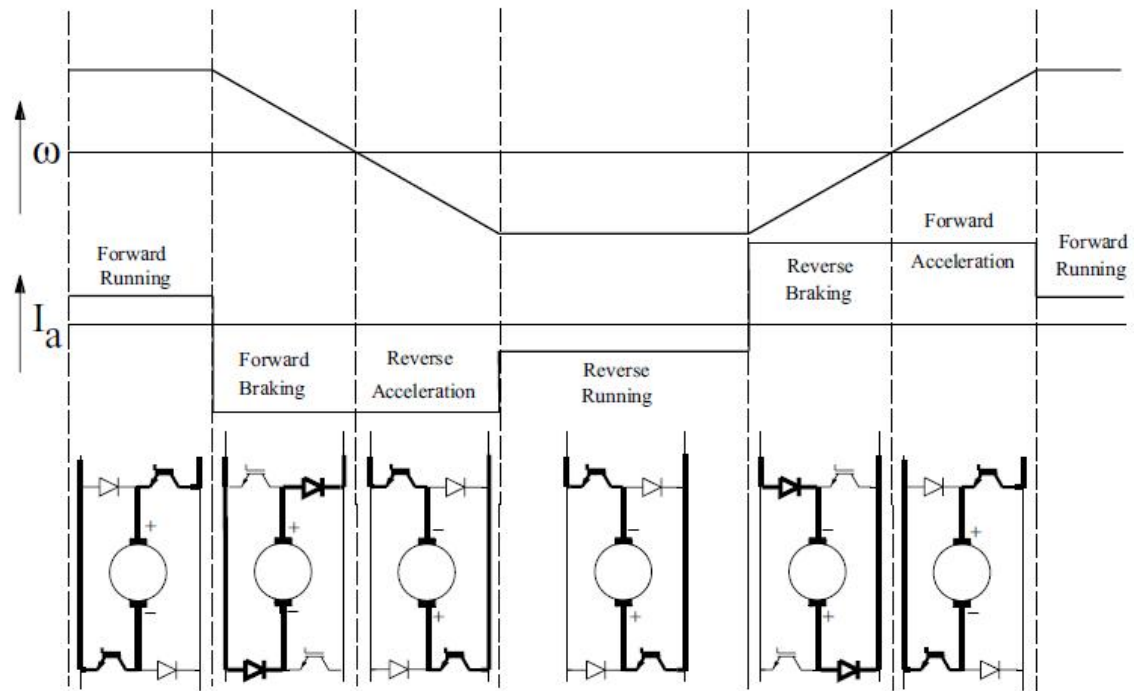


Figura 11

La combinación en función de lo visto arriba sería de la siguiente manera.

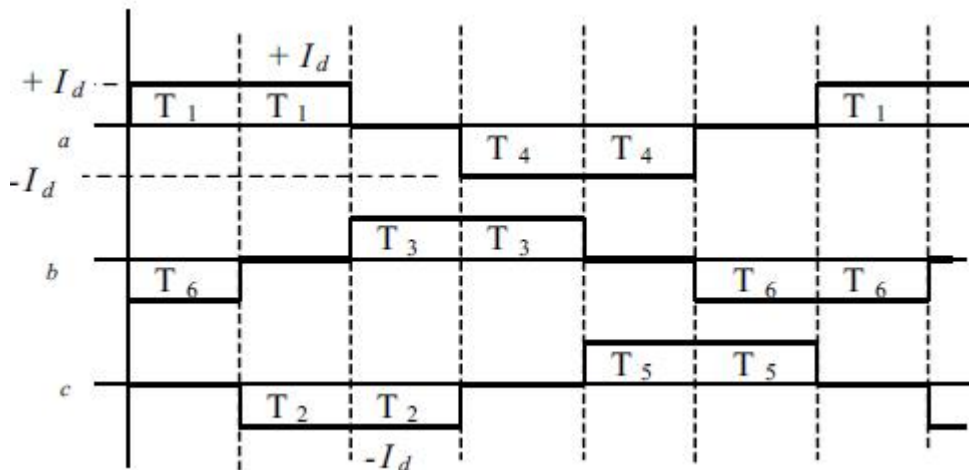


Figura 12

3. Conclusiones finales

- Teniendo en cuenta los objetivos planteados en esta tesis, se necesitó el manejo de herramientas nuevas en un muy corto tiempo. Podemos afirmar que un nuevo panorama se nos ha presentado. En este podemos ver que el desarrollo de una máquina-herramienta tan compleja como un brazo robótico es algo posible. Esto sirve ejemplo para todas aquellas personas que deseen apostar al desarrollo nacional.
- Si bien la tesis está finalizando, nuestra tarea recién comienza ya que consideramos que esta es la primera etapa de muchas etapas futuras que vendrán para poder hacer un desarrollo completo.

- Sentimos que con este trabajo y se nos puso una barrera de potencial. Es vital y gratificante reconocer hasta donde hemos llegado y que hemos logrado. A pesar de estar tan cerca de terminar la carrera, aun nos queda un largo camino por recorrer. Pero por suerte los conocimientos adquiridos, ganas, esfuerzo y perseverancia lo pueden contra todo.

4. Bibliografía

* (§1) Compiler Construction using Flex and Bison. Anthony A. Aaby. Walla Walla College cs.wwc.edu aabyan@wwc.edu Version of April 22, 2005
<http://foja.dcs.fmph.uniba.sk/kompilatory/docs/compiler.pdf>

* (§2) How to Write a Simple Parser by Ashim Gupta
<http://ashimg.tripod.com/Parser.html>

* (§3) A COMPACT GUIDE TO LEX & YACC by Tom Niemann
<http://epaperpress.com/lexandyacc/download/lexyacc.pdf>

* (§4) Wikipedia

* (§5) Electrónica de Potencia M. Rashid – Ed. Prentice Hall