DESIGN ASSEMBLER BASED ON LEX AND YACC

Amera Ismail Melhum, *Suzan Abdulla Mahmood
Department of Computer Science, College of Science, University of Duhok. Duhok-Iraq.
* Department of Computer Science, College of Science, University of Sulaimani. Sulaimaniah-Iraq.

Abstract
LEX and YACC are very useful tools for constructing the assemblers; they generate functions that perform standard operations of a lexical analysis and parsing without any effecting on the organization processes for semantic analysis, machine code generation and listing source of compilation. In this work, the cross assembler was designed using some compiler development tools like LEX and YACC having the ability to implement the lexical analyzer and parser, for generating the syntax and parsing modules in a short period of time. These activates are expressed in form of actions, the bigger number of lexical and grammar rules are used by these actions in a simpler way.
Keywords: Assembly language, Assembler, Context Free Grammar, LEX**,** YACC

تصميم المجمع الصلب باستخدام YACC و LEX

أميرة اسماعيل ملحم، *سوزان عبدالله محمود
قسم علوم الحاسبات، كلية العلوم، جامعة دهوك. دهوك–العراق.
*** قسم علوم الحاسبات، كلية العلوم، جامعة السليمانية. السليمانية–العراق.**

الخلاصة
تعد اسلوبي LEX وYACC ذات فائدة كبيرة لبناء المجمعات. اذ تولد وظائف لتنفيذ العمليات القياسية وذلك عن طريق تحليل المفردات بدون تأثير على تنظيم عمليات التحليل الدلالي. في هذا البحث تمت تصميم المجمع الصلب باستخدام بعض ادوات التطوير المترجم مثل LEX وYACC ذلك لقابليتهم لأنجاز تحليل المفردات و تعريب الكلمة و انشاء تركيب في فترة زمنية قليلة مقارنتها مع اللغات البرمجية مثل **C++** و **Visual Basic**و الخ من اللغات البرمجية والجدير بالاشارة ان هذه الفعاليات توصف بشكل صيغ واجراءات بطريقة مبسطة ولها القدرة على معالجة أكبر عدد من القواعد النحوية ومفردات المعاجم.

1. Introduction

In our world, there is a great variety of microprocessors, and the construction of assemblers for these microprocessors requires much man-power and time [1]. Assemblers are often considered as consisting of two phases. The analysis phase, where most of the error checking is done, and the synthesis phase, where the object code is created. During the analysis phase, the assembler source code will be scanned for any lexical or syntactic errors. This is analogous to spelling and grammar checks in a word processor. If errors are found, exceptions are thrown and reported to the developer. If, however, no errors are found, method calls are made to the backend and synthesis begin [2].

This paper is of interest not only to new comers to assembler construction, but to those already familiar with the subject who wishes to follow the development of assembler for microprocessors requires. It can therefore be

used as an extensive programming project in a software engineering setting.

In this research, it shows how to build the assembler using some compiler development tools like LEX and YACC[3] having the ability to implement the lexical analyzer and parser also help us to generate the syntax and parsing modules in a short period of time.

In the next section, the structure of the assembler is implemented by using two pass assembler, the two pass assembler refer to the number of time reading file.

1.1 LEX and Yacc

An assembler, compiler or interpreter for a programming language is often decomposed into two parts:[4]

1-Read the source program and discover its structure.

2-Process this structure, e.g. to generate the target program.

Lex and Yacc can generate program fragments that solve the first task. The task of discovering the source structure again is decomposed into subtasks:

1-Split the source file into tokens (Lex).

2-Find the hierarchical structure of the program (Yacc).

Lex and Yacc are tools that help programmers build compilers and interpreters, but they also have a wider range of applications like converting text data to HTML ,XML or Latex.

1.2 Lex : A lexical Analyzer Generator

Lex helps to write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine. While Lex can be used to generate sequences of tokens for a parser, it can also be used to perform simple input operations. A number of programs have been written to convert simple English text into dialects using only Lex.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream[5, 6], the following is a sample code for Lex part, in the left hand the description of Token and in the right hand the associative action for it.

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}
%%
/* variables */
[a-z]    {yylval = *yytext - 'a';
          return VARIABLE;}

/* integers */
[0-9]+   {  yylval = atoi(yytext);
         return INTEGER;}
          }
/* operators */
[-+()=/*\n]  { return *yytext; }
/* skip whitespace */
[ \t] ;
/* anything else is an error */
.    yyerror("invalidcharacter");
%%
int yywrap(void) {
return 1 ;}
```

1.3 Yacc: Yet Another Compiler – Compiler

Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an ``input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine [5, 6]. The following a sample code for Yacc part, in the left hand the description of Grammar rule and in the right hand the associative action for it.

```
%token INTEGER VARIABLE
%left '+' '-'
```

```
%left '*' '/'
%{
void yyerror(char *);
int yylex(void);
int sym[26];
%}
%%
program:
program statement '\n'
|
;
statement:
expr { printf("%d\n", $1); }
| VARIABLE '=' expr { sym[$1] = $3; }
;
expr:
INTEGER
| VARIABLE { $$ = sym[$1]; }
| expr '+' expr { $$ = $1 + $3; }
| expr '-' expr { $$ = $1 - $3; }
| expr '*' expr { $$ = $1 * $3; }
| expr '/' expr { $$ = $1 / $3; }
| '(' expr ')' { $$ = $2; }
;
%%
```

2. Proposed structure of the Assembler

Since problem of assembler is as old as symbolic languages, there are many different solutions of it described in details in the literature. All the approaches may be roughly classified according to the number of readings (passes) of the input file. So there are one-pass, two pass and multi-pass assembler; they have well known advantages and disadvantages, the two pass assembler is surely the simplest for implementation and that's why we have decided to accept this approach.

In this paper, the whole algorithm has been divided into two parts called "pass 1" and" pass 2" performing the functions as shown in (Figure 1) and (Figure 2).

The functions of Pass_1:

-Checking lexical, syntax and semantic correctness of a program.
-Storing label definitions and calls (references).
-Checking possibility of resolving all internal and external references.
-Printing out the source program merged (if such need) with error messages.

The functions of Pass_2:

-Checking semantic correctness of a program.

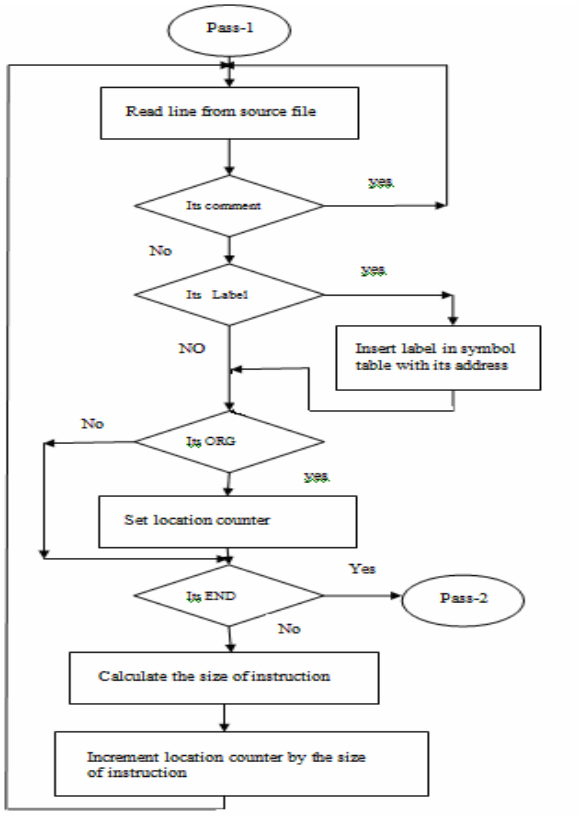

Figure 1: Flowchart of Pass 1

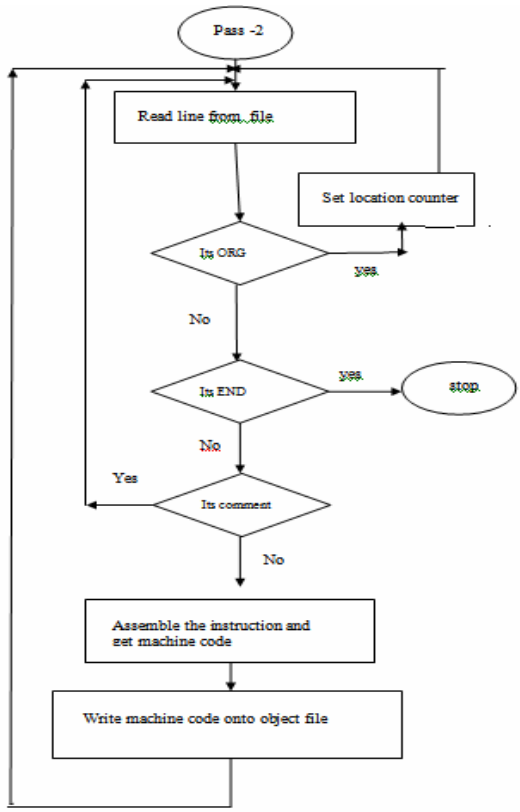

Figure 2: Flowchart of pass2

-Label processing (searching for the required labels).
-Machine code generation.
-Generation of compilation listing- in case of

error suitable message is displayed.

The flowcharts were translated to the C language program with the following structural units:

- data
- main segment
- lexical analyzer routine
- terminal symbol table
- parser routine
- grammar rules actions
- post-pass 1 processing
- post pass 2 processing

The most important problem is distribution of "work" (i.e. function specified for both the passes) between these structural units. Especially roles of lexical analyzer, parser, and terminal symbol actions grammar rules actions are exchangeable in a wide range. It is even possible to organize the process of lexical analysis within the parser routine but this is surely an ineffective approach. On the other hand, the lexical analyzer supported by carefully design terminal symbol actions may shorten the routine and speed up its execution. Assigning strict and unique functions to the structural units would result in inefficiency features of the parser, lexical analyzer and C- language constructions. So, the machine code can be taken by the parser routine, whether from the machine code tables or from the lexical analyzer.

The internal structure of the design assembler has been shown in the (Figure 3).

The first step, the context-free grammar for assembly language was defined, once a context-free grammar for the language has been properly defined, the next step would be used to design tools that will check the developer code for errors, based upon these rules. The LEX / YACC or the GNU(General Public license) Flex/Bison[7], these are software tools that essentially scan a grammar definition, given in BNF, and then generate the lexer and parser accordingly. First, it is need to specify all pattern matching rules for lex (bas.l) and grammar rules for yacc (bas.y).

A context-free grammar consists of four components [8, 9]:

1. A set of tokens, known as terminal symbols.
2. A set of nonterminals.
3. A set of productions, i.e. a nonterminal followed by a sequence of tokens and/or nonterminals
4. A designation of one of the nonterminals as the start symbol.

This definition essentially states a context-free grammar must begin with a start symbol and eventually reduced to a finite set of keywords, known as tokens or terminals, and non-terminals (e.g. variable names, strings of numbers or characters). The rules that define these objects and their order in the language are known as productions. This will be made clearer in the following example. Developed collaboratively with John Backus, the father of FORTRAN, Backus- Noir Format (BNF) is syntax for describing context-free grammars, as given by Chomsky.example of the BNF for a simple calculator is shown below:

expr → mexpr (+ mexpr) *
mexpr → atom (* atom) *
atom → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Here, the tokens are
+ * 0 1 2 3 4 5 6 7 8 9

and the nonterminals are expr, mexpr and atom.

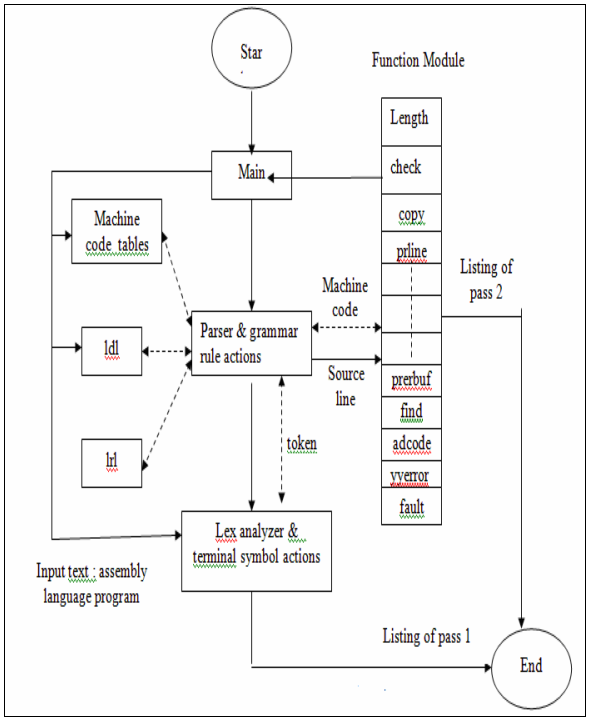

Figure 3: The internal structure of the design –assembler

The start symbol is expr and the first two lines are productions. The | is "or" and → is the symbol for definition such that the last line defines an atom as a zero or a one or a two, and so on. Note the quantifier ( )* allows for zero or more of the objects contained in the parentheses. Hence, the first line essentially defines an expression as a multiplicative expression, optionally added to one or more multiplicative expressions. This allows for statements such as

2+3*4 in this simple example grammar [3].the following is a part of the BNF for the proposal assembler

prog → *stmts* end
stmts → *stmt* newline *stmts* | epsiod
stmt → line | comment | label line | equ *arg*
line → ld *arg* , *arg*
| in *arg* , *arg*
| out *arg* , *arg*
| *brs* *arg* , *arg*
| *al1*
| *al2*
| *al2* *arg* , *arg*
| posh ras
| ret *conr*
arg → a | b | c | d | e | |h | i | r | hl | bc | de | af
| sp |af |ix | iy
| [bc |de | hl | sp |*ixy* | *label* | *number*]
conr → z | c | p | m | nz | nc | po | pe
ixy → *i* *xy* *mp* *number*
xy → x | y
mp → - | +
posh → pop | push
al1 → cp | sub | and | or | xor
al2 → add | adc | sbc
al3 → inc | dec
ras → rlc | rrc | rl | rr | sla | sra
| srl
brs → bit | set | res
.....
.....
.....
comment → *; identifier*
identifier → *letter identifier* | *letter letterdigit*
| *episode*
label → *letter Letterdigit*

letterdigit → *letter letterdigit* | *digit letterdigit*
| _ letterdigit | episode
number → *digit number* | episode
digit → 0 | 1 | 2 | 3|… |9
letter → a | b | c | d |…|z

2.1 Data (machine code for the instructions)

The assembler uses 2 types of data:
-One or two dimension tables contain machine codes of instruction for different operands. The correct association of an instruction and operands correspond to a non-zero value of a machine code table element. Two lists of symbolic names (labels):
- ldl, the list of the label definitions, the address of actual label that appears in the label field .
- lrl, the list of the required labels, the labels that appears in operand field.

Both lists have the same structure:

```
struct list { char   name[10]; // name of label
  int      lnum;  // line number
  int      address;  // address contents of the
                     byte counter
  struct   list  *next}
```

*ldl // pointer to the current element of the ldl
* pd // pointer to the last element of the ldl
* lrl // pointer to the current element of the lrl
*pr // pointer to the last element of the LRL

Applying the list structure is an acute necessity since number of labels and references to them is dependent on a program and can vary in a big range.

2.2 Main segment

In contrast to the name this segment performs only initialization of the lists LDL and LRL, calls parser and after terminating its work, prints the closing message.

2.3 Lexical Analyzer Routine

The lexical analyzer routine identifies terminal symbols in the input text. Each time the routine tries to find the longest string of characters matching one of the given patterns of terminal symbols. In case of ambiguity (there are two or more patterns matching the same string) the first pattern found is taken into account. So the order of patterns have an important role. Each of the patterns correspond to the user define actions coded in the C–language. Recognition of the given pattern in the input text is associated with execution of the suitable action.

This routine was produced by the lexical analyzer generator LEX: input data to this program is just specification of patterns and corresponding to actions (terminal symbol actions). Output file of the LEX is (yy.lex.c) file that contains the C –language lexical analyzer routine.

2.4 Terminal symbol actions

Main job of a terminal symbol action is informing the parser routine about the identified token. Additionally this action organizes of a characters is printed out (by suitable place of the output buffer char linebuf [75]). The number of lexical rules decides about the size of the lexical analyzer routine. It is common view, that an action continues the process of input string

identification instead of multiplication rules it is enough to add one or two statements in the C-language. May be the final specification becomes not so clear, but it undoubtedly produces shorter (both source and object) routine. Usually an action transmits a value corresponding to the token: sometimes it is a machine code, sometime s selector of an instruction belonging to the token and sometimes it is a string of characters.

2.5 Parser routine

The parser routine has to state correctness of the input sentence. The sequence of tokens received from the lexical analyzer is compared with the grammar rules defined within the parser, when such rule is found the tokens are reduced to the nonterminal symbol and the action corresponding to this grammar rule is executed. If there is no such grammar rule the parser calls function errorok( ) sending message about the error and makes recovery (removes effects of the error; usually rest of the line is skipped).

The parser routine was produced by the complier YACC [6, 7] on the base of an user (designer) supplied grammar. This grammar accompanied by actions (grammar rule actions), YACC accepts this data and produces the C-language routine yyparse( ) loaded to the file y.tab.c. To obtain the object code of the assembler the contents of the two files y.tab.c and lex.yy.c must be processed by the C-language compiler (CC).

The output of Lex part will be the input to the Yacc part, its token with its associated value store in variable. The token are NA LABEL JPCAL COMMT ORG ARG CONR LD AL2 AL1 AL3 RAS BRS RAT IN OUT RET POSH END CON DJR RESB ACON, the following are examples about some token with its value:

Token	**value associated with token**
ARG	a | b | c | d | e | |h | i | r | hl | bc | de | af | sp |af |ix | iy | [bc |de | hl | sp |*ixy* | *label* |*number*]
AL2	add | adc | sbc
AL1	cp | sub | and | or | xor
AL3	inc | dec
RAS	rlc | rrc | rl | rr | sla | sra | srl
BRS	bit | set | res
POSH	pop | push
JPCAL	call | jp
END	end
ORG	org
RESB	resb

2.6 Grammar rules actions

Most of the grammar rules are associated with actions. It means that each time a grammar rule is applied, the corresponding action is performed. The top most grammar rule(corresponding to recognition of a complete assembly language line terminated by 'new line' character) calls action printing out the line along with an error message (if any) and the byte counter will be incremented by the machine code length for the current instruction.

Majority of actions cooperates with grammar rules identifying assembly language statements. Each such action identifies the kind of instruction and operand(s) and continues testing syntax correctness of the statement; if nothing wrong is detected, the action finds a machine code corresponding to the instruction (pass 2) or only determines the length of the instruction machine code.

2.7 Post - pass 1 processing

Pass 1 is terminated when the lexical analyzer discovers end of the input file. The lexical analyzer calls a designer supplied function yywrap( ) . This function is called after terminating of the pass 2.

In case of errors coming from syntax analysis of a program or post pass 1 processing, the assembler sends a message and ends its work. Otherwise the input file is reopened and the pass 2 is started.

2.8 Post - pass 2 processing

At the end of file calls again the function yywrap(Function yywrap is called by lex when input is exhausted. Return 1 if you are done or 0 if more processing is required) like in post-pass 1 processing but this time the second part of the function is active, to produce a reference table including information about all defined /declared labels and references to them.

3. Conclusion

It is worth to notice that LEX and YACC seem to be a too advanced tool for constructing of such simple compilers like assemblers. It also helps us to generate the syntax parsing modules in a short period of time, LEX and YACC having the ability to implement the lexical analyzer and parser and both are based on the minimal specification of a programming language which includes the definitions of the set of symbols used (lexical rules), the set of valid programs (syntax rules) and the "meaning"

of valid programs (semantics). Using this way implemented by Z80 assembler.

References

1. Peter, P.K. And sammy, T.K., **1990**.A Generative Approach to universal cross assembler Design, *ACM SIGPLAN* 25(1), pp 43-51.
2. Nakano, K. and Ito, Y. **2008.** Processor, Assembler, and Complier Design Education using an FPGA, *IEEE International Conference on Parallel and Distributed Systems, 14(1)* pp 723-728.
3. Lappalainen, P. **2001.** *Visualization of assembly-code conversion into machine code*. http://www.ee.oulu.fi/~pl1/tkt1/
4. Niemann, T. **2004.** *A Guid to LEX & YACC*. http://www.scribd.com/ doc/ 43874121/A-Guide-to-Lex-Yacc-By-Thomas-Niemann
5. Linseman, A. and Nicol, S. **1988**. *MKS LEX & Yacc reference manual,* Mortice Kerns Systems, Inc. pp 240-270.
6. Levine, J. and Mason, T. **1992**. *Lex & Yacc.* Second Edition O'Reilly and Associates. Brown, pp 235-300.
7. Ashton, B.; Bradley, J.; Dixon, T.; Gaines, J.; Lodder, M. and Ricks, M., **2007.** *DDC Assembly Language Specification and Assembler Implementation,* Engineering Clinic Project Sponsored by Hill Air Force Base, University of Utah.
8. Aho, A.V.; Sethi, R. and Ullman, J. **2006.** *Compilers Principles Techniques and Tools,* 2nd Edition,Dargon Book, pp 156-170 http://www.jim-gaines.com/ research_DDC/ DDCFinalPaper3.pdf
9. Nicol, G.T. **1993.** *Flex: the lexical scanner generator. Free Software Foundation***,** Version 2.3.7.http://www.amazon.com/Flex-Lexical-Scanner-Generator-Version/dp/1882114213