Compiler Compiler: Flex and Bison

1 Today's goal

- Learn how to use lexical scanner Flex and context parser Bison.
- Learn how to write Flex grammar files.
- Learn how to write Bison grammar files.
- Learn how to write assembler MICROC.

2 Today's contents

Step 1 *Check 1* Write a Flex grammar file (List 1) for evaluating the postfix formula below and check if it works correctly.

```
5 24 8 - * 9 4 + 8 - *
```

Step 2 Write a Bison grammar file (List 2) for evaluating postfix formulas and check if it works correctly.

Step 3 Write a Bison grammar file (List 3) for evaluating the infix formula below and check if it works correctly.

```
12 - 4 * 5 + 11 * 3
```

Step 4 *Check 2* Write a Flex grammar file (List 4) and a Bison grammar file (List 5) for evaluating infix formulas and check if it works correctly using the infix formula above.

Step 5 Write a Flex grammar file (List 6) and a Bison grammar file (List 7) for MICROC and check if it works correctly.

Step 6 Write a Flex grammar file (List 6) and a Bison grammar file (List 7) to support shift operations `<<` and `>>`, and bitwise operations `&`, `|`, and `^`.

Step 7 Write a following program using MICROC.

1. n is given from input port `in`;
2. the following operation is repeated until n is 1.

   if n is even, $n \leftarrow n/2$.
   if n is odd, $n \leftarrow n * 3 + 1$.
3. in each iteration, the current value of n is written in the output buffer.

For example, if $n = 3$, then we have the following output.

$$3 \rightarrow 10 \rightarrow 5 \rightarrow 16 \rightarrow 8 \rightarrow 4 \rightarrow 2 \rightarrow 1$$

Step 8 *Check 3* Compile the MICROC program in Step 7 using the extended MICROC obtained in Step 6. Perform the simulation to see if it works properly.

Step 9 *Check 4* Generate the bit file for Step 8, and implement it in the FPGA.

3 Flex and postfix formula evaluation

Flex is a tool for generating a scanner, which recognizes lexical patterns in a text file. List 1 is a Flex grammar file. Using Flex, this grammar file converted into a C program that scans a text file and evaluates a formula in it. We assume that a formula is a postfix form of

- non-negative integers, and
- three operators `+`, `-`, and `*`.

A flex grammar file consists of four main sections as shown in Figure 1. In List 1, each section has following elements:

C declarations This section contains the definition of array s and prototype definitions of five functions, which are used in actions of grammar rules. This section may be empty for simple grammars.

Flex declarations It contains simple name definitions. In List 1, DIGIT is defined as `[0-9]`.

```
%{
C declarations
%}
Flex declarations
%%
Flex grammar rules
%%
Additional C code
```

Figure 1: Four main sections in Flex grammar files

Grammar rules This section contains one or more Flex grammar rules. List 1 has four grammar rules. Each rule is a pair of a pattern and an action. The action can be empty. In the the first line, if new line(`\n`) is matched, action `printf("result=%d\n",s[0]);` is executed. In the next line, `{DIGIT}+` is matched with a non-negative integer. Note that `DIGIT` is defined to be `[0-9]`. If matched, action `push(atoi(yytext)); printstack();` is executed. These functions are defined in Additional C code section. The following three patterns `\+`, `-`, and `\*` are matched with `+`, `-`, and `*`, respectively. Since `+` and `*` are special symbols used for regular expressions, escape character `\` is used.

Additional C code The additional C code section is copied to the end of scanner C program. In List 1, five functions are defined, used in actions of Flex grammar rules.

To generate the scanner C program for Flex grammar file postfix.l in List 1 and the executable scanner, execute

```
$ flex postfix.l
$ gcc -o postfix lex.yy.c -lfl
```

In the first line, Flex program generates scanner C program lex.yy.c. The second line generates executable scanner postfix.

The executable scanner postfix read the input from the standard input. If the following line is given,

```
4 3 2 * + 1 -
```

then the output shows the values of the stack and the resulting value as follows:

```
4 0 0 0
3 4 0 0
2 3 4 0
6 4 0 0
```

List 1: Flex grammer file postfix.l to evaluate a postfix formula

```
1  %{
2  #include <stdio.h>
3  int s[4];
4  void printstack();
5  void push(int);
6  void add();
7  void sub();
8  void mul();
9  %}
10 DIGIT [0−9]
11 %%
12 \n {printf("result=%d\n",s[0]);}
13 {DIGIT}+ {push(atoi(yytext)); printstack();}
14 \+ {add(); printstack();}
15 − {sub(); printstack();}
16 \∗ {mul(); printstack();}
17 %%
18 void printstack(){
19   printf("%d %d %d %d\n",s[0],s[1],s[2],s[3]);
20 }
21 void push(int x){
22   s[3]=s[2];
23   s[2]=s[1];
24   s[1]=s[0];
25   s[0]=x;
26 }
27 void add(){
28   s[0]=s[1]+s[0];
29   s[1]=s[2];
30   s[2]=s[3];
31 }
32 void sub(){
33   s[0]=s[1]−s[0];
34   s[1]=s[2];
35   s[2]=s[3];
36 }
37 void mul(){
38   s[0]=s[1]∗s[0];
39   s[1]=s[2];
40   s[2]=s[3];
41 }
```

```
10 0 0 0
1 10 0 0
9 0 0 0
result=9
```

4 Bison and postfix formula evaluation

Bison is a tool for generating parsers, which analyzes input text based on rules defined by context-free grammar. List 2 is a Bison grammar file. Using Bison, this Bison grammar file is converted into a C program that parses an input text and evaluates a formula in it.

Similarly to Flex, a Bison grammar file consists of four main sections as shown in Figure 2. In List 2, each section has following elements:

C declarations The C declarations section contains macro definitions and declarations of functions and variables used in the actions in the grammar rules. List 2 has `#include <stdio.h>` in this section because function printf is used in an action.

Bison declarations This section contains declarations that define terminal and non-terminal symbols. In List 2, NUMBER is defined as a token (terminal symbol).

Bison grammar rules This section contains one or more Bison grammar rules, which are pairs of a rule and an action. A grammar rule is in the form of

A : B ;

where A is a non-terminal symbol and B is a sequence of non-terminal/terminal symbols. Intuitively, the rule means that, the right side sequence is matched with a substring in the input text, the substring is substituted with the non-terminal symbol in the left side. If two or more rules have the same non-terminal symbol in the left side they can be merged into one by using separator `|`. In other words,

A : B1;
A : B2;
A : B3;

can be rewritten with

A : B1 | B2 | B3;

```
%{
C declarations
%}
Bison declarations
%%
Bison grammar rules
%%
Additional C code
```

Figure 2: Four main sections in a Bison grammar file

For example, List 2 has four rules for non-terminal symbol expr, which mean that,

- expr← NUMBER
- expr← expr expr +
- expr← expr expr -
- expr← expr expr *

Each rule may have an action which is included in brackets `{ }`. If these rules are applied, the corresponding action is executed if exists.

Additional C code The additional C code section is copied to the end of parser C program. In List 2, three functions are defined. Function yyerror is called if error occurs. To start parsing we need to call yyparse. Thus, function main calls yyparse. Function yylex is called when the parser needs input text. It needs to return a terminal symbol including a token or character in the input. In List 2, function yylex reads a character from the standard input, and returns token NUMBER if it is a digit, returns the input character verbatim if it is `+`,`-`,`*`, or `\n`.

To generate a Bison parser, we execute

```
$ bison -y postfix.y
$ gcc -o postfix y.tab.c
```

In the first line, bison program generates parser C program y.tab.c. By compiling this program using gcc, we can obtain executable parser postfix.

In the Bison grammar file of List 2, two non-terminal symbols input, expr are used. Also, it has five terminal symbols `\n`, NUMBER (token), `+`, `-`, and `*`. It contains eight grammar rules as follows:

1 input← *empty*

List 2: Bison grammer file postfix.y to evaluate a postfix formula

```
1  %{
2  #include <stdio.h>
3  %}
4  %token NUMBER
5  %%
6  input :
7        | input expr '\n' { printf("result=%d\n",$2); }
8  ;
9  expr : NUMBER
10       | expr expr '+' { $$ = $1 + $2; }
11       | expr expr '-' { $$ = $1 - $2; }
12       | expr expr '*' { $$ = $1 * $2; }
13 ;
14 %%
15 int yylex(){
16   int c;
17   while((c=getchar())!=EOF){
18     if(isdigit(c)){
19       ungetc(c,stdin);
20       scanf("%d",&yylval);
21       return(NUMBER);
22     } else if(c=='+'|c=='-'|c=='*'|c=='\n')
23       return(c);
24   }
25 }
26 int yyerror(char *s){printf("%s\n",s);}
27 int main(){yyparse();}
```

2 input← input expr \n

3 expr← NUMBER

4 expr← expr expr +

5 expr← expr expr -

6 expr← expr expr *

The parser read the input using yylex and apply these rules. The parsing is successful if we have non-terminal input when we have read the whole input. For example, input 432*+1- is parsed as follows:

$$
\begin{aligned}
\underline{4}32*+1-\texttt{\textbackslash n} &\xrightarrow{yylex} \underline{\text{NUMBER}}32*+1-\texttt{\textbackslash n}\\
&\xrightarrow{3} \text{expr}\underline{3}2*+1-\texttt{\textbackslash n}\\
&\xrightarrow{yylex} \text{expr}\underline{\text{NUMBER}}2*+1-\texttt{\textbackslash n}\\
&\xrightarrow{3} \text{exprexpr}\underline{2}*+1-\texttt{\textbackslash n}\\
&\xrightarrow{yylex} \text{exprexpr}\underline{\text{NUMBER}}*+1-\texttt{\textbackslash n}\\
&\xrightarrow{3} \underline{\text{exprexprexpr}*}+1-\texttt{\textbackslash n}\\
&\xrightarrow{6} \underline{\text{exprexpr}+}1-\texttt{\textbackslash n}\\
&\xrightarrow{4} \text{expr}\underline{1}-\texttt{\textbackslash n}\\
&\xrightarrow{yylex} \text{expr}\underline{\text{NUMBER}}-\texttt{\textbackslash n}\\
&\xrightarrow{3} \underline{\text{exprexpr}-}\texttt{\textbackslash n}\\
&\xrightarrow{5} \underline{\textit{empty}\text{expr}}\texttt{\textbackslash n}\\
&\xrightarrow{1} \underline{\text{inputexpr}}\texttt{\textbackslash n}\\
&\xrightarrow{2} \text{input}
\end{aligned}
$$

When the whole input is reduced to first non-terminal symbol input, the parsing is successful. Also, when the rule is applied, the corresponding action is executed.

A token can take a *semantic value*. By default, semantic values are integers. In the definition of function yylex, the value of c-'0' is assigned yylval if c is a digit, and returns token NUMBER. This means that, the digit is changed to token NUMBER with semantic value c-'0'. Thus, if c is '5', then it is converted to token NUMBER with semantic value 5. In actions of Bison grammar rules, the semantic values can be used. For example, rule expr← expr expr + has action `$$=$1+$2`, in which, the semantic values of the first and the second exprs are added and the resulting value are assigned to

the semantic value of the left side expr. In general, for any integer i, `$i` is the semantic value of i-th non-terminal/terminal symbol in the right side, and `$$` is the semantic value of the left side non-terminal symbol.

5 Infix formula evaluation using Bison

List 3 is a Bison grammar file to evaluate infix formulas. In the Bison declaration section, associativity and precedence of operators are defined. In this Bison declaration section, 3 operators `+`, `-`, and `*` are defined to be left associative. The left associative means that the same operators are evaluated from the left. For example, since `-` is left associative, $3-2-1$ is evaluated such that $(3-2)-1$. Further, associativity defined later has higher precedence. For example, `*` has higher precedence than `+` and `-`.

If we give the following input to a parser generated for List 3,

```
4 + 3 * 2 - 1
```

we have the correct result as follows:

```
result=9
```

6 Combining Flex and Bison

In List 3, function yylex defined in the additional C code section works as a scanner. We modify List 3 such that the scan operation is defined by Flex.

List 4 is a Flex grammar file to scan infix formulas. A non-negative integer is converted to token NUMBER with semantic value being its integer value. Characters `+`, `-`, `*`, and `\n` are converted to terminal symbols verbatim. Tokens and terminal symbols are returned to a parser generated by Bison in List 5. Note that "y.tab.h", which is a definition file containing ID numbers of tokens etc assigned using Bison grammar file (List 5), is included in C declaration section.

We can obtain the executable program for evaluating infix formulas as follows:

```
$ flex infix2.l
$ bison -d -y infix2.y
$ gcc -o infix2 lex.yy.c y.tab.c
```

Flex program generates scanner "lex.yy.c". Bison program generates parser "y.tab.c" and its header file "y.tab.h".

List 3: Bison grammer file infix.y to evaluate infix formulas

```
1  %{
2  #include <stdio.h>
3  %}
4  %left '+' '-'
5  %left '*'
6  %token NUMBER
7  %%
8  input :
9          | input expr '\n' { printf("result=%d\n",$2); }
10 ;
11 expr : NUMBER
12       | expr '+' expr { $$ = $1 + $3;}
13       | expr '-' expr { $$ = $1 - $3;}
14       | expr '*' expr { $$ = $1 * $3;}
15 ;
16 %%
17 int yylex(){
18   int c;
19   while((c=getchar())!=EOF){
20     if(isdigit(c)){
21       ungetc(c,stdin);
22       scanf("%d",&yylval);
23       return(NUMBER);
24     } else if(c=='+'|c=='-'|c=='*'|c=='\n')
25       return(c);
26   }
27 }
28 int yyerror(char *s){printf("%s\n",s);}
29 int main(){yyparse();}
```

List 4: Flex grammer file infix2.l to scan infix formulas

```
1  %{
2  #include "y.tab.h"
3  %}
4  %%
5  [0-9]+ {yylval=atoi(yytext);return(NUMBER);}
6  \+|-|\*|\n {return(yytext[0]);}
7  %%
8  int yywrap(){return(1);}
```

List 5: Bison grammer file infix2.y to parse infix formulas

```
%{
#include <stdio.h>
%}
%left '+' '−'
%left '∗'
%token NUMBER
%%
input :
        | input expr '\n' { printf("result=%d\n",$2); }
;
expr : NUMBER
        | expr '+' expr { $$ = $1 + $3;}
        | expr '−' expr { $$ = $1 − $3;}
        | expr '∗' expr { $$ = $1 ∗ $3;}
;
%%
int yyerror(char ∗s){printf("%s\n",s);}
int main(){yyparse();}
```

7 C-like language MICROC

The specification of MICROC is as follows:

Name A string starts with a English letter followed by English letters or Arabic numbers. Name is used to define variable names or labels. Name must be no more than 16 letters. If it has more, first 16 letters are used.

integer A decimal integer.

label A label is in the form "label:", which indicates the address to be jumped by goto statement.

variable definition Integer variables are defined such as "*int n=0,m=1;*". If initialization is omitted, it takes initial value zero.

goto Goto statement takes the form "goto label;" and jumps to the label.

if-goto **and** unless-goto These statements takes form "if(formula) goto label;" and "unless(formula) goto label;". if-goto statement jumps to label if the value of formula is not zero. unless-goto statement jumps to label if the value of formula is zero.

halt terminate the execution of the program.

out It takes form " out(formula)" and outputs the resulting value of the formula.

assignment It takes form "variable = formula;" and the resulting value of formula is stored in the variable.

formula It consists of variables, numbers, arithmetic and logic operations including addition+, subtraction-, multiplication ∗, and equality==.

7.1 Flex grammar file for MICROC

List 6 is a Flex grammar file microc.l. By rule `[ \t\n\r]` with no action, blank characters are ignored. Tokens EQ, GOTO, HALT, IF, IN, INT, OUT, UNLESS, NUMBER, and NAME are retured to the parser. Usually, token is assigned to reserved words or operations with two or more characters. If it returns NUMBER or NAME, the matched string up to 16 letters are assigned to the corresponding token as a semantic value. Note that yylval in List 6 takes a string of characters, while it takes an integer value by default.

List 6: Flex grammer file microc.l for MICROC

```
%{
 #include "y.tab.h"
%}
%%
[ \t\n\r]
== {return(EQ);}
goto {return(GOTO);}
halt {return(HALT);}
if {return(IF);}
in {return(IN);}
int {return(INT);}
out {return(OUT);}
unless {return(UNLESS);}
[0−9]+ {strncpy(yylval.s,yytext,16);return(NUMBER);}
[a−zA−Z][a−zA−Z0−9]∗ {strncpy(yylval.s,yytext,16);return(
      NAME);}
. {return(yytext[0]);}
%%
int yywrap(){ return(1);}
```

List 7 is a Bison grammar file microc.y for MICROC. In C declarations section defines the data type of the semantic value. More specifically, "`%union{char s[17];}`" means that yylval can store a string up to 17 characters. After that, "`%token <s> NUMBER NAME`" defines that they takes a semantic value with `char s[17]`.

List 7: Bison grammer file microc.y for MICROC

```
%union{char s[17];}
%token <s> NUMBER NAME
%token EQ GOTO IF UNLESS INT IN OUT HALT
%left EQ
%left '+' '−'
%left '∗'
%%
input : statement | input statement
;
statement : label | intdef | goto | if | unless | halt | out | assign
```

```
11  ;
12  label : NAME ':' {printf("%s:\n",$1);}
13  ;
14  intdef: INT intlist ';'
15  ;
16  intlist: integer
17          | intlist ',' integer
18  ;
19  integer: NAME {printf("%s: 0\n",$1);}
20          | NAME '=' NUMBER {printf("%s: %s\n",$1,$3);}
21          | NAME '=' '-' NUMBER {printf("%s: -%s\n",$1,$4
                );}
22  ;
23  goto: GOTO NAME ';' {printf("\tJMP %s\n",$2);}
24  ;
25  if: IF '(' expr ')' GOTO NAME ';' {printf("\tJNZ %s\n",$6);}
26  ;
27  unless: UNLESS '(' expr ')' GOTO NAME ';' {printf("\tJZ %s
        \n",$6);}
28  ;
29  halt : HALT ';' {printf("\tHALT\n");}
30  ;
31  out: OUT '(' expr ')' ';' {printf("\tOUT\n");}
32  ;
33  assign: NAME '=' expr ';' {printf("\tPOP %s\n",$1);}
34  ;
35  expr: NAME {printf("\tPUSH %s\n",$1);}
36        | NUMBER {printf("\tPUSHI %s\n",$1);}
37        | IN {printf("\tIN\n");}
38        | expr '+' expr {printf("\tADD\n");}
39        | expr '-' expr {printf("\tSUB\n");}
40        | expr '*' expr {printf("\tMUL\n");}
41        | expr EQ expr {printf("\tEQ\n");}
42  ;
43  %%
44  int yyerror(char *s){ printf("%s\n",s); }
45  int main(){ yyparse(); }
```

We can obtain MICROC compiler tinyc using Flex and Bison as follows:

```
$ flex microc.l
$ bison -d -y microc.y
$ gcc -o microc lex.yy.c yy.tab.c
```

Also, MICROC countdown program count.c (List ??) can be converted to an assembly language program as follows:

```
$ ./microc < count.c > count.asm
```

Finally, we can obtain the Verilog HDL code to be inserted in ram.v.

```
$ ./tinyasm < count.asm | ./mac2mem
```

List 8: Countdown program count.c using MICROC

```
1   n=in;
2   L1:
3     out(n);
4     unless(n) goto L2;
5     n=n-1;
6     goto L1;
7   L2:
8     halt;
9     int n;
```

List 9: Assembly language program count.asm for countdown

```
1           IN
2           POP n
3   L1:
4           PUSH n
5           OUT
6           PUSH n
7           JZ L2
8           PUSH n
9           PUSHI 1
10          SUB
11          POP n
12          JMP L1
13  L2:
14          HALT
15  n: 0
```

8 Homework

Homework 1 Modify Flex and Bison grammar files (Lists 4 and 5 to support all binary operations in ALU module alu.v. Give an infix formula containing all binary operations to confirm that infix formula is evaluated correctly.

Homework 2 Modify MICROC to support all binary operations in ALU module alu.v. Write a MICROC formula containing all binary operations and generate the corresponding Verilog HDL using the compiler and the assembler. Perform the simulation to confirm if everything is ok.