What good is a cpu without an assembler? In this chapter we use flex and bison (i.e. lex and yacc) to make an assembler for our cpu. The assembler generates a binary file that can be loaded by the verilog $readmemh statement, or in a synthesized FPGA we would use the binary file to initialize the instruction memory inside the FPGAs internal memory blocks.

Flex and Bison

Flex and bison are used to create many compilers for many languages and was also used to create the GCC compiler collection. Flex and Bison are the modern implementations of Lex and Yacc which was originally created for unix systems. Yacc is an acronym for "Yet Another Compiler Compiler", which whimsically states it's purpose.

Text lexer/scanner using Flex

We begin creating the lexer/scanner for our assembler by coding a flex input file. The term scanner and lexer can be used interchangeably. This file defines how the input assember program as text is broken down into tokens, identifiers and constants. The main section of this file lists a collection of regular expressions to match against the input text and a corresponding action to generate a token it's value to pass on to the parser. The action is coded as regular C/C++ but typically only a line or two of code is required for each action. This lexer input file is used by the flex program at compile time to generate a C file containing your specific text input scanner. In fact, we add an extra step in the makefile to automatically regenerate our scanner C file any time the lexer file is changed. Also, the scanner C file is considered an intermediate file and is not required to be included during distribution, issuing a
'make clean'
also deletes this file.

Here are the regular expressions and actions that implement our first assembler. In the top section, before the %% line, we define some simple regular expressions (shortcuts) for common text sequences we expect to encounter. After which, we define the actual scanner tokens and the associated action to execute for each occurrence. Some of the actions simply return a constant integer defined elsewhere that specify the token that was encountered. Others, such as the definition of an identifier and string, also pass a value to the parser via the yylval variable.

Partial listing of the scanner input file (scanner.l):

```
 17:  delim      [ t]
18:  whitesp     {delim}+
```

```
19:  digit         [0-9]
20:  alpha         [a-zA-Z]
21:  alphanum      [a-zA-Z0-9]
22:  number        [-]?{digit}*[.]?{digit}+
23:  integer       [-]?{digit}+
24:  hex       "0x"[0-9a-fA-F]+
25:  string        "[^"]*"
26:  register      [rR][-]?{digit}+
27:  comment       "#"[^n]*
28:  identifier    {alphanum}[a-zA-Z0-9_]*
```

Above we simply declare some regular expression shortcuts for text we expect to encounter. Below are the actual scanner expressions and associated actions. These regular expressions are separated by a %% line in the source file.

```
  32:  {register}   { sscanf(yytext+1, "%d", &yylval); return REG; }
33:  {integer}    { sscanf(yytext, "%d", &yylval); return INTEGER; }
34:  {hex}        { sscanf(yytext+2, "%x", &yylval); return INTEGER; }
35:
36:  "n"        { return NEWLINE; }
37:  ","     { return COMMA; }
38:  ":"     { return COLON; }
39:
40:  "NOP"       { return NOP; }
41:  "LRI"     { return LRI; }
42:  "ADD"       { return ADD; }
43:  "SUB"       { return SUB; }
44:  "OR"        { return OR; }
45:  "XOR"       { return XOR; }
46:  "HALT"      { return HALT; }
47:  "BRA"       { return BRA; }
48:  "BRANZ"     { return BRANZ; }
49:  "BRAL"      { return BRAL; }
50:  "BRALNZ"    { return BRALNZ; }
51:  "CALL"       { return CALL; }
52:
53:  ".imem"     { return sIMEM; }
54:  ".regfile"  { return sREGFILE; }
55:  ".base"     { return sBASE; }
56:  ".define"   { return sDEFINE; }
57:  ".register" { return sREGISTER; }
58:  ".end"       { return END; }
59:
60:  {identifier} {
61:    yylval = yf_getsymbol(yytext);
```

```
 62:    if(yylval<=0)
 63:     yylval = yf_addsymbol(yytext, ST_UNKNOWN, 0);
 64:    return IDENTIFIER;
 65:        }
 66:  {string}    {
 67:    yytext[strlen(yytext)-1] = 0;
 68:    yylval = yf_addstring(&yytext[1]);
 69:    return STRING;
 70:       }
 71:
 72:  {whitesp}   { /* No action and no return */ }
 73:  {comment}   { /* No action and no return */ }
```

The parser using Bison and a grammar definition file

Using a similar process as with the flex input file, we create a bison grammer file to implement our parser. Tokens stream from the lexer/scanner into the parser and our grammar determines which tokens may follow another, i.e. it defines the syntax of our language. The grammar of the grammar used to define the syntax of our language is very similar to BNF grammar form. We often see BNF form when we study the syntax of languages like C/C++, Java, SQL, and more.

For example, here is a sample of a BNF-like grammer from the mysql documentation:

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  { LIKE old_tbl_name | (LIKE old_tbl_name) }
create_definition:
  col_name column_definition
| [CONSTRAINT [symbol]] PRIMARY KEY [index_type] (index_col_name,...)
| {INDEX|KEY} [index_name] [index_type] (index_col_name,...)
| [CONSTRAINT [symbol]] UNIQUE [INDEX|KEY]
    [index_name] [index_type] (index_col_name,...)
| {FULLTEXT|SPATIAL} [INDEX|KEY] [index_name] (index_col_name,...)
| [CONSTRAINT [symbol]] FOREIGN KEY
    [index_name] (index_col_name,...) reference_definition
| CHECK (expr)

column_definition:
  data_type [NOT NULL | NULL] [DEFAULT default_value]
    [AUTO_INCREMENT] [UNIQUE [KEY] | [PRIMARY] KEY]
    [COMMENT 'string'] [reference_definition]
```

Like the flex input file, inside the bison grammar input file actions can be specified for any input token sequence encountered. For our assembler, these actions will emit the binary machine code for our program in a format loadable by the verilog $readmemh statement. In the grammar actions the $n variables are used to access the token value (the contents of the scanner's yylval variable) for previously encountered tokens. These are the only non-C symbols found in the actions and are converted to C expressions in the intermediate C file generated by Bison. Thus, in the following sample grammar input line:

```
result : INTEGER sMUL INTEGER { $$ = $1 * $3; }
```

The variable $1 references the first INTEGER token value, and the $3 references the second INTEGER token value. (It's in the 3rd token location.) The $$ variable references the value of the *result* token, which based on the action, would be the result of the first integer times the second integer. This is an arbitrary example of a grammar that parses mathematical expressions.

Most variables and functions used in the following code listing are defined in another source file and emit the assembly output.

Partial listing of the grammar definition file for our assembler parser (parser.y):

```
  51:  input:              /* empty string */
 52:       | input line { yylineno++; }
 53:       ;
 54:  line: NEWLINE
 55:       | statement NEWLINE
 56:       | asm_expr NEWLINE
 57:       | label_decl NEWLINE
 58:       | register_decl NEWLINE
 59:       | definition NEWLINE
 60:       | END
 61:       ;
 62:
 63:  /* assembler directives */
 64:  asm_expr: sIMEM INTEGER    { alloc_imem( $2 ); }
 65:       | sIMEM INTEGER INTEGER { sys.imem_width = $3; alloc_imem( $2 ); }
 66:          | sREGFILE INTEGER INTEGER { sys.regfile = $2; sys.reg_addr_bits = $3; }
 67:          | sREGFILE INTEGER { sys.regfile = $2; }
```

```
 68:     | sBASE INTEGER { sys.base = $2; }
 69:     ;
 70:
 71:  /* grammers for each cpu mnemonic */
 72:  statement: NOP { gen( xNOP, 0); }
 73:          | LRI format_rd_imm   { gen( xLRI, $2); }
 74:          | ADD format_ra_rb_rd { gen( xADD, $2 ); }
 75:      | SUB format_ra_rb_rd { gen( xSUB, $2 ); }
 76:      | OR format_ra_rb_rd { gen( xOR, $2 ); }
 77:      | XOR format_ra_rb_rd { gen( xXOR, $2 ); }
 78:      | BRA format_0_rb_0 { gen( xBRA, $2 ); }
 79:      | BRANZ format_ra_rb_0 { gen( xBRANZ, $2 ); }
 80:      | BRAL format_0_label { gen( xBRAL, $2 ); }
 81:      | BRALNZ format_label_ra { gen( xBRALNZ, $2 ); }
 82:      | CALL format_label_rd { gen( xCALL, $2 ); }
 83:      | HALT { gen( xHALT, 0xfff ); }
 84:      ;
 85:
 86:  /* grammers for each of the mnemonic formats */
 87:  format_rd_imm: reg COMMA INTEGER { $$ = ENCR( ($3 >> sys.reg_addr_bits) & 0xf, $3
& 0xf, $1 ); };
 88:  format_ra_rb_rd: reg COMMA reg COMMA reg { $$ = ENCR( $1, $3, $5 ); };
 89:  format_0_rb_0: reg { $$ = ENCR( 0, $1, 0 ); };
 90:  format_ra_rb_0: reg COMMA reg { $$ = ENCR($1, $2, 0); };
 91:  format_label_ra: label COMMA reg { $$ = ENCR($3, $1 & 0xf, ($1 >>
sys.reg_addr_bits)&0xf ); };
 92:  format_label_rd: label COMMA reg { $$ = ENCR( ($1 >> sys.reg_addr_bits)&0xf, $1 & 0xf,
$3 ); };
 93:  format_0_label: label { $$ = ENCR(0, $1 & 0xf, ($1 >> sys.reg_addr_bits)&0xf ); };
 94:  //format_0_rb_rd: reg COMMA reg { $$ = ENC( 0, $1, $3 ); };
 95:  //format_ra_imm: reg COMMA INTEGER { $$ = ENCR($1, $3 & 0xf, ($3 >>
sys.reg_addr_bits)&0xf ); };
 96:  //format_imm_0: INTEGER { $$ = ENCR(0, $1, 0); };
 97:
 98:  label_decl: IDENTIFIER COLON { yf_setsymbol( $1, ST_LABEL, sys.base); };
 99:  register_decl: sREGISTER REG IDENTIFIER { yf_setsymbol( $3, ST_REGISTER, $2); };
100:  definition: sDEFINE IDENTIFIER INTEGER { yf_setsymbol( $2, ST_INT, $3); }
101:        | sDEFINE IDENTIFIER STRING { yf_setsymbol( $2, ST_STRING, $3); }
102:        ;
103:
104:  /* a label is a reference to a memory address, a constant is also valid */
105:  label: INTEGER { $$ = $1; }
106:       | IDENTIFIER { yf_symbol s = yf_getsymbol($1);
                  if(s.type==ST_LABEL) $$ = s.lvalue; else yyerror("expected label"); }
107:
108:  /* a reg is a reference to a register */
```

```
109:  reg: REG { $$ = $1; }
110:     | IDENTIFIER { yf_symbol s = yf_getsymbol($1);
                  if(s.type==ST_REGISTER) $$ = s.lvalue; else yyerror("expected register"); }
```

Download the complete source code for Chapter 3

Chapter 3 - Your First Assembler (tgz)

Chapter 3 - Your First Assembler (zip)