Lab 3 - Project Report

AI24BTECH11031 - Shivram S

Contents

1 Introduction

As part of the CS2323 Computer Architecure course, we had to develop an assembler for the RISC-V instruction set architecture. This report outlines the design of the assembler and the major design decisions taken during its development.

2 Usage

The project includes a `Makefile` which can be used to build the project and test it. To build, run:

```
$ make
```

This produces an executable called `riscv_asm` in the project directory.

The assembler can be run using the following command:

```
$ ./riscv_asm
```

By default, the assembler reads input from `input.s`, and writes output to `output.s`. Input and output files can be specified using command-line arguments:

```
$ ./riscv_asm <input file> <output file>
```

To test the assembler, run

```
$ make test
```

3 Planning

FIrst, the structure of the assembler was outlined, and important design decisions were taken.

3.1 Parsing

The first step in converting assembly code to machine code is to read the program and parse it into semantic elements such as instructions and labels.

Since there problem statement makes several assumptions about the code (such as having one instruction per line, one space after each comma, etc.), it is possible to parse the code using only simple functions like `scanf`.

However, I decided to do for a more structured approach, using separate tokenization and parsing steps, to give practical utility to the assembler.

3.2 One-pass versus Multi-pass

Assemblers can be broadly divided into two categories - one-pass assemblers and multi-pass assemblers.

A one-pass assembler processes the source code once, and patches earlier output based on later definitions of symbols. On the other hand, multi-pass assemblers create a symbol table in the first pass, and then use it to generate code in the second pass.

It is common to see assembly code that refers to labels defined later in the program:

```
    beq x6, x0, Exit    ; Exit has not been defined yet
    ; Some code omitted
Exit:                   ; Exit defined here
    add x0, x0, x0
```

While scanning the program, when we reach line 1, we have no idea where `Exit` is defined and what address it corresponds to.

If the assembler followed a single-pass design, we would need to maintain a table of locations where a label is referenced, and then update the earlier location when we encountered the label.

To avoid this complication, I decided to use a two-pass design. After parsing the program, The first pass creates a list of labels and their addresses, and the second pass emits the machine code for each instruction, substituting labels for their addresses.

3.3 Error Handling

The program that the assembler receives as input need not be correct. It can have syntactic and semantic errors, and the assembler is expected to find and report these errors.

My goal was to build an error reporting system similar to

Unlike other popular languages such as C++ and Java, C lacks an inbuilt exception-handling system. Programmers have to implement error handling themselves.

I had three strategies in mind for error handling:

1. **Using Result types (`std::expected`)**
   The C++ STL has a struct called `std::expected`, which can hold either a return value in the case of success, or an error in the case of failure. This can be implemented in C as:

   ```
   struct Result {
       int is_err;
   ```

```
    void *data;
}
```

The drawbacks of this approach are:

(a) Having to allocate memory in leaf functions where errors will be thrown.

(b) Having to check each function call for an error value.

(c) Less type safety guarantees as the value is returned in a `void` pointer.

2. **Passing a pointer to write errors into**
   In this approach, we simply define a struct for the error:

   ```
   struct ParseErr {
       int is_err;
       char *msg;
       int line, start_col, end_col;
   }
   ```

   and then pass a pointer to functions like so:

   ```
   int parse_register(Parser *p, ParseErr *err);
   ```

   The function sets `is_err` to 1 if an error occurs, and provides helpful data such as an error message and the error location.

   The caller can then return early or halt execution depending on the value of `is_err`.

3. **Using `setjmp` and `longjmp`**
   These C functions are often used to implement try-catch style error handling. However, use of these functions is sometimes considered a bad practice.

After careful consideration, I decided to use method 2 (Passing a pointer to write errors into) for handling errors in the parser and the emitter.

This method was chosen because it is commonly used in larger C projects and also doesn't add as much overhead as the first method.

3.4 Bottom-up Parsing

There are several strategies for parsing a languages. Languages with elaborate syntax use top-down parsers. However, due to the simple syntax of assembly, and due to the simplicity of implementation, i chosse a bottom-up parser design.

The parser has functions which parse small elements such as register names or numeric literals, and these functions are composed together into functions that parse larger elements.

For example, the parser has two functions:

```
int parse_register(Parser *p, ParseErr *err);
void parser_expect(Parser *p, TokenType tt, ParseErr, *err);
```

parse_register parses a register and returns the corresponding register number (0 - 31). parser_expect moves to the next token if the current token type is equal to tt, otherwisre it throws an error.

These two functions can be composed into a function that parses the operands to an R-format instruction (error checks omitted):

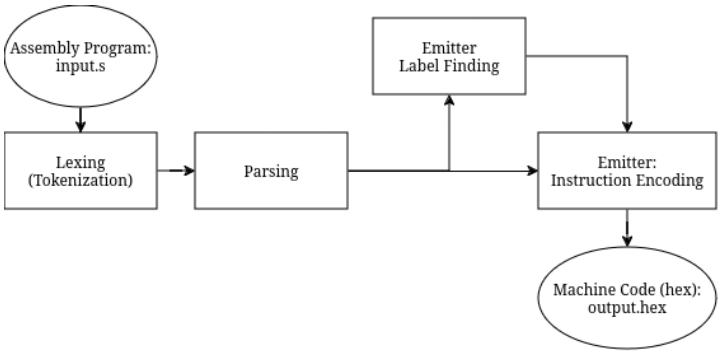
```
RIns parse_r_ins(Parser *p, ParseErr *err) {
    RIns ins = {0};
    ins.rd = parse_register(p, err);    // Parse rd operand
    parser_expect(p, TOK_COMMA, err);   // Expected comma after that
    ins.rs1 = parse_register(p, err);   // Parse rs1 operand
    parser_expect(p, TOK_COMMA, err);   // Expected comma after that
    ins.rs2 = parse_register(p, err);   // Parse rs2 operand
    return ins;
}
```

This is a very useful abstraction, especially when working with languages having elaborate syntax, and also reduces unnecessary repetition of code.

4 Design

The working of the assembler can be broken down into four distinct steps:

1. Lexing/Tokenization: text to tokens
2. Parsing: tokens to semantic elements
3. Label Finding: creates a list of labels
4. Instruction Encoding: instructions to machine code (hex)


Assembly Program: input.s
Lexing (Tokenization)
Parsing
Emitter Label Finding
Emitter: Instruction Encoding
Machine Code (hex): output.hex

4.1 Lexer / Tokenizer

The assembler has a highly optimized lexer which scans the input program and generates *tokens*. A token is the smallest unit of a program, such as an identifier, a numeric literal, or special characters like colons and commas.

$$\underbrace{L1}_{ident}\ \underbrace{:}_{colon}\ \underbrace{lui}_{ident}\ \underbrace{x3}_{ident}\ \underbrace{,}_{comma}\ \underbrace{0{\times}1000}_{hexnum}$$

The lexer assigns types to portions of the assembly program based on their syntactic meaning. A `Token` is defined as follows:

```
typedef struct Token {
    TokenType type;
    Span span;
} Token;
```

`Span` is a struct that holds the start and end positions of a token:

```
typedef struct Span {
    size_t start, end;
} Span;
```

The `TokenType` is an enum which denotes the token type assigned to that portion of the source code.

```
typedef enum TokenType {
    TOK_EOF, TOK_ERR,   // Tokens for end-of-file and syntax errors
    TOK_IDENT,
    TOK_DECNUM, TOK_BINNUM, TOK_HEXNUM, TOK_OCTNUM,
    TOK_COMMA, TOK_COLON, TOK_LPAREN, TOK_RPAREN,
} TokenType;
```

The lexer itself requires a struct storing peroperties such As the source string and the current position.

```
typedef struct Lexer {
    char *src;
    size_t pos, line, lastline;
} Lexer;
```

`line` and `lastline` hold the current line number, and the position of the last newline character respectively. These are used in error reporting.

The lexer API is very simple and has only two functions:

```
void lexer_init(Lexer* l, char *src);
Token lexer_next(Lexer *l);
```

`lexer_init` is used to initialize the lexer object. `lexer_next` returns the next token in the source code, based on the current position of the lexer.

4.2 Lookup Tables

An assembler needs to maintain a list of instructions supported broadly the ISA, the syntax of each instruction, and the operands it takes.

To do this, the assembler uses lookup tables, which map the name of an insturction to attributes such as the instruction format, the operands and `funct3` and `funct7` values.

For example, the lookup table for S-format instructions is defined as follows:

```
SInsTableEntry s_ins_table[] = {
    {"sb", 0b0100011, 0x0},
    {"sh", 0b0100011, 0x1},
    {"sw", 0b0100011, 0x2},
    {"sd", 0b0100011, 0x3},
};
```

where each entry is of the type:

```
typedef struct SInsTableEntry {
    char *key;
    int opcode, funct3;
} SInsTableEntry;
```

Similar tables are defined for instructions in other formats. These lookup tables are used by the parser to identify an instruction's format and syntax, and call the corresponding procedures.

One notable distinction is the existence of two tables for I-format instructions:

```
IInsTableEntry i_ins_table[] = {
    {"addi", 0b0010011, 0x0},
    // ...
    {"srai", 0b0010011, 0x5},
};

IInsTableEntry i_ins_table_2[] = {
    {"lb", 0b0000011, 0x0},
    // ...
    {"jalr", 0b1100111, 0x0}
};
```

This distinction is due to I-format instructions having two different syntaxes. Arithmetic instructions are written as `addi rs, rs1, imm` while memory based instructions are written as `ld rs2, imm(rs1)`.

Having a separate lookup table for each syntax simplifies the parsing of these instructions.

The assembler also maintains a mapping of register names to register numbers:

```
RegTableEntry reg_table[] = {
    {"zero", 0},
    {"ra", 1},
    // ...
};
```

Where each entry is of the form:

```
typedef struct RegTableEntry {
    char *key;
    int value;
} RegTableEntry;
```

4.3 Parser

The parser works in tandem with the lexer to convert sequences of tokens into semantic elements such as instructions and labels.

The parser works in tanden with the lexer. It requests tokens from the lexer and emits a `ParseNode` object, defined as:

```
typedef struct ParseNode {
    NodeType type;
    union {
        Label l;
        RIns r;
        IIns i;
        SIns s;
        BIns b;
        UIns u;
        JIns j;
    } data;
    int line;
} ParseNode;
```

The struct is essentially a tagged union, containing data corresponding either to a label or to one of the six instruction types. The line number of the element is also stored to facilitate error reporting in the emitter.

The enum `NodeType` just indicates the type of the node (i.e which field of the union to access), and is defined as:

```
typedef enum NodeType {
    LABEL, R_INS, I_INS, S_INS, B_INS, U_INS, J_INS
} NodeType;
```

The structs for instructions are pretty similar. They hold data about the instruction's fields (opcode, funct3, funct7), and the operands (registers and immediate). For example, `RIns` is defined as follows:

```
typedef struct RIns {
    RInsTableEntry *entry;
    int rd, rs1, rs2;
} RIns;
```

`entry` is an element of the lookup table for R-instructions, detailed in the previous section.

The parser itself is defined as:

```
typedef struct Parser {
    Token current;
    Lexer *lexer;
    char *src;
} Parser;
```

The section on bottom-up parsing gives an outline of the parser. Two public functions are defined:

```
// Initializes the parser
void parser_init(Parser *p, Lexer *l);
```

```
// Parses and returns a single semantic element
ParseNode parser_next(Parser *p, ParseErr *err);
```

parser_next first compares the current token with various instruction names. By doing this, it identifies the instruction format, and calls the function responsible for parsing that instruction.

If the token does not match any of the instrucion, then it checks if the token corresponds to a label definitions (identifier followed by colon). If that too does not match, then it raises an error.

4.3.1 Parsing Immediates

Instructions often take an immediate as an operand. In the case of I-, S- and U-format instructions, these are often numeric literals, but in the case of B- and J- format instructions, both numeric literals and labels are used as immediates.

Hence, it is necessary to account for both cases - labels and numeric literals. This is done using a struct called NumOrLabel:

```
typedef struct NumOrLabel {
    int is_label;
    union {
        int n;
        Label l;
    } data;
} NumOrLabel;
```

This is a tagged union which can hold either a label or a numeric literal.

4.4 Emitter

The emitter takes the parse tree as input and emits machine code. This is done in two steps. First it finds and enumerates all the labels in the parse tree. Then it converts instructions to the machine code equivalent.

The emitter also has an error handling system which is very similar to the parser. The EmitErr struct is used for this purpose.

```
typedef struct EmitErr {
    int is_err;
    char *msg;
    int line;
} EmitErr;
```

This struct is similar to `ParseErr` except that the column span for the error are not stored.

Since RISC-V immediates are encoded into the instruction, they have to fit in a certain number of bits. The emitter performs bounds checking for these immediate values.

In addition to this, the emitter also handles cases where an undefined label is used in the program, or a label is defined multiple times.

5 Features

The assembler satisfies the following requirements:

- Supports all RV64I Base Integer Instructions
- Supports both standard RISC-V register names and calling convention aliases
- Supports decimal, hexadecimal, binary and octal numeric literals.
- Reports errors in the program along with their line numbers.
- Supports comments starting with the `;` character.
- Flexible parsing allowing for extra whitespace between tokens.

The assembler handles the following kinds of errors:

- Syntax errors
- Invalid instruction and register names
- References to undefined labels
- Redefinition of a label
- Labels without instructions (e.g. at end of file)
- Immediates not fitting in instruction field
- Branch and jump immediates being odd numbers

6 Testing

The project includes a few testcases in the `test/` directory. A bash script, `test.sh`, is used to run all the tests. Each test includes an assembly program `input.s` and the expected output `expected.hex`.

The testcases include examples from the problem statement, as well as some miscellaneous programs.

7 Future Scope

The assembler can be extended further in future to make it a more practical tool. Some possible features include

- Ability to report multiple errors in the same program.
- Support for RISC-V pseudo-instructions.
- Support for `.data` section.

8 Project File Structure

```
.
|-- Makefile
|-- report
|   |-- (...)
|   |-- main.tex    (Source file for the report)
|-- src
|   |-- emitter.c   (Source code for emitter)
|   |-- emitter.h
|   |-- lexer.c     (Source code for lexer)
|   |-- lexer.h
|   |-- main.c      (Driver code for the assembler)
|   |-- parser.c    (Source code for the parser)
|   |-- parser.h
|   |-- tables.c    (Instruction and register tables)
|   `-- tables.h
|-- test            (Test cases)
`-- test.sh         (Script for running tests)
```