LC2K Assembler

Rijun Cai
12348003

January 4, 2014

1 Introduction

In this laboratory, an LC2K assembler was implemented in C++. In the following, I will give some detailed explaination of my source code of the assembler.

2 Source Code Explaination

2.1 Types & Classes Summary

To organized the program better, several types and classes are defined. Here is a list of them.

`mc_t` An alias of `uint_least32_t`. The type of a machine code.

`mc_t_s` An alias of `int_least32_t`. A signed version of `mc_t`. Used to output.

`SyntaxError` Exception to indicate the syntax errors in the input.

`IOError` Exception to indicate the errors occur in IO.

`Assembler` The class of an assembler. For more details, see 2.3.

`Assembler.Instruction` The class of an instruction.

2.2 Workflow

The `main` function will first read the `.asm` file with `readFromFile` function. Then an assembler will be created and fed with the strings of the input file. Then the assembler will analyse the input strings with `Assembler.synAnalyse`. Then the Pass One and Pass Two processes will be executed and the machine codes will be generated. Finally, `writeToFile` will be called to write the machine codes to the output file. Here is the main part of the `main` function.

```
try
{
    vector<string> as = readFromFile(argv[1]);
    Assembler assembler;
    assembler.setAssembly(as);

    assembler.synAnalyse();
    assembler.passOne();
    assembler.passTwo();

    writeToFile(argv[2], assembler.getMechineCode());
}
catch(SyntaxError e)
{
```

```
    std::cerr << "Error occured when assembling.\n" << e.what() << std::endl;
    return EXIT_FAILURE;
}
catch(IOError e)
{
    std::cerr << "IOError occured: " << e.what() << std::endl;
    return EXIT_FAILURE;
}
```

2.3 `Assembler` Class in Detail

`Assembler` class (along with `Assembler.Instruction` structure) implements the main functions of the assembler. Here are some details of it.

The `*_INS` sets will be initialized when the assembler is constructed, and will be used to identify the type of the instructions.

In the syntax analysis process (`synAnalyse` function), every line of the assembly code will be parsed as list of string tokens by `std::stringstream`. Then the first token will be checked and if it is a name of an instruction, the following fields will be treated as the fields of the instruction, otherwise, the first token will be treated as a label. The result of `synAnalyse` will be a list of `Instruction` (`Assembler._ins`).

In `passOne`, every label will be checked if it is duplicated and then mapped to the address of the instruction. The result will be stored in `Assembler._label_table`.

In `passTwo`, every instruction will be passed to the corresponding translation function and translated into a machine code.

The definitation of `Assembler` class is as follow.

```
class Assembler
{
    struct Instruction
    {
        Instruction() {}
        Instruction(
                const string &_label,
                const string &_instruction,
                const vector<string> _fields):
            label(_label), instruction(_instruction), fields(_fields) {}

        string label;
        string instruction;
        vector<string> fields;
    };

    private:
        vector<string> _as;
        vector<Instruction> _ins;
        map<string, size_t> _label_table;
        vector<mc_t> _mc;

        set<string> _INS;
        set<string> _R_INS;
        set<string> _I_INS;
        set<string> _J_INS;
        set<string> _O_INS;
        set<string> _PSEUDO_INS;

        inline int interpreOffsetField(const string &field,
                bool *pLabelSign = 0, bool half_word = true);
```

```
        inline mc_t translateAdd(const Instruction &ins);
        inline mc_t translateNand(const Instruction &ins);
        inline mc_t translateLw(const Instruction &ins);
        inline mc_t translateSw(const Instruction &ins);
        inline mc_t translateBeq(const Instruction &ins, size_t pc);
        inline mc_t translateJalr(const Instruction &ins);
        inline mc_t translateHalt(const Instruction &ins);
        inline mc_t translateNoop(const Instruction &ins);
        inline mc_t translateDotFill(const Instruction &ins);

        inline static mc_t convertRegisterField(const string &field);

    public:
        Assembler();
        void setAssembly(const vector<string> &as);
        void synAnalyse();
        void passOne();
        void passTwo();
        vector<mc_t> getMechineCode();
};
```