Nasr Alae-eddine
119010531
CSC3050 Computer Architecture

Assignment 1 report

Development environment

- Pre-configured VM provided by the TAs

Supposedly : (Reference from the BB VM setup)

- Linux Kernel: 5.10.5 (zen)
- GCC Version: 10.2.0
- Clang Version: 11.0.0
- GNU Make: 4.3
- CMake Suite: 3.19.2
- Ninja: 1.10.2
- Icarus Verilog: 11.0

Project Goal

Since Machines cannot understand higher level language. The code that we usually write needs to be Translated to Binary instructions that are readable by the machine.
In this project we deal with this exact process, MIPS is an assembly language, it is quite low level but still readable by humans (and not by the machine)
Fortunately enough, the instructions map 1 to 1 to their machine code counterpart.
That is for each line, each instruction of the sort `add $t3, $t1, $t2`
It has one and only 1 equivalent "`00000001001010100100000000100000`"
In fact, these Binary instructions follow some strict rules. It is 32 bits always
And we can separate these bits into certain building blocks:

```
lw   $t0, 1200($t1)     # temporary register $t0 gets A[300]
add  $t0, $s2, $t0      # temporary register $t0 gets h +A[300]
sw   $t0, 1200($t1)     # stores h + A[300] back into A[300]
```

35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

This “block” separation has 3 main patterns:
R-Instructions

```
   op      rs       rt       rd     shamt  funct
| 6bits | 5bits | 5bits | 5bits | 5bits | 6bits |
```

I-Instructions

```
  op       rs       rt          immediate
| 6bits | 5bits | 5bits |        16bits          |
```

J-Instructions

```
 op                 address
| 6bits |              26bits                    |
```

Design Idea

Now that we know how MIPS Instructions are made, what are the problems that we may run into and how to design a program to solve this and output correct machine code ?
First we have a more concrete look at what it is we have to do: (reference BB Assignment 1)

```
# MIPS code:
 .text
R: add $s0, $s1, $s2 #r instructions
addu $s0, $s1, $s2
sub $s0, $s1, $s2
subu $s0, $s1, $s2

Machine code:
00000010001100101000000000100000
00000010001100101000000000100001
00000010001100101000000000100010
00000010001100101000000000100011
```

Before all else this assignment is mainly about *Parsing.* Basically we have a text file that we have to read, Recognize different blocks and process them.
So the work relies heavily on string manipulation, streams(string streams) operations …etc
My first thought before starting to operate on the lines is to:

- Remove comments
- Trim lines
- Tokenize each line for convenient assembling

Since in this first project we only care about the .text segment we have to first find it.
Then from there we loop on the next set of lines until the end of file

Labels

If we Just start working directly and translating instructions we will soon run into a problem. That is Labels. First of all the label itself takes up a line which does not translate to any instruction so it must be skipped, and secondly some instructions explicitly call these labels, in such cases the address/ relative address of the label must be conveyed to the instruction at hand. How to Provide this address when the label definition may anywhere before or after the instructions ? The solution provided is to separate the Parsing into 2 phases, first we store all labels, then with that information we should finally be able to assemble our machine code!
But that is not all, to do all this we must first define a fitting data structure to hold the labels. And while we are at it we can make a header file to link up both phases of the model.

LabelTable.hpp

```
class Label{

    public:

        Label(std::string name, int32_t address){
        this->name = name;
        this->address = address;
        }

        int32_t getAddress(){
            return address;
        }

        std::string getName(){
            return name;
        }

    private:

        std::string name;
        int32_t address;

        friend bool operator==(Label label, std::string string);
        friend bool operator==(std::string string, Label label);
};
```

The class above declares everything we may need, from the name and address attributes, their getters and even the overloading of the == operator which will be useful later when iterating over the label list.
Not to make this report too long, in header file fashion the file also defines a storage vector for the labels, and a few maps for the registers, and the 3 kinds of instructions (R,I,J)
And also all the functions declarations

Phase1.cpp

This phase is relatively short and easy.
After we find the .text segment we read every line
The hint to separate instructions from labels is the ":" colon character
When this is encountered we take the label and add it to the storage vector:

```
Label label(trim_line.substr(0, marker), 0x400000 + (inst_counter * 4));
labels_vector.push_back(label);
```

The address is computer with `0x400000 + (inst_counter * 4)`
The instruction counter here gets incremented by 1 each line of the loop that isn't a label

Phase2.cpp

This is the longer phase and the core of the project
First starting with function definitions which will be used later to make machine code from the input tokens, one such function for the R types:

```
string assemble_R(string instruction, string rd, string rs, string rt,
string shamt, string funct){

    string op = "000000";
    string destination = regToAddr(rd);
    string first_reg = regToAddr(rs);
    string second_reg = regToAddr(rt);

    int temp = stoi(shamt);
    string shift_amnt = bitset<5>(temp).to_string();

    if(first_reg.empty()) first_reg = "00000";
    if(second_reg.empty()) second_reg = "00000";
    if(destination.empty()) destination = "00000";

    return op + first_reg + second_reg + destination + shift_amnt + funct;

}
```

Each input token gets taken care of separately (rs, rt, rd, shamt, funct)
Translated into machine code using the maps defined in LabelTable.hpp

Then in the parsing function itself similarly to phase 1 we first find the .text segment.
We keep an instruction counter that ignores label lines.
And all that is left is to call our defined functions with the accurate arguments depending on which operation it is.
So follows is code with about 50+ if else statements regarding the first token that signifies the operation type.
I won't go into detail of how each one is handled but in general we find the funct/op-code in the relevant map and from there we update the result stream with the output of the assemble function relevant to the type of instruction at hand.
Some operations are trickier than the others such as the ones that rely on labels. We first try to find if the label is in the storage, we get the address from the list and we create the relative address from there to input into the assembling function.
E.g:

```
if(tokens[0] == "beq"){

                iter = I_Map.find(tokens[0]);
                int temp = labelNameToAddr(tokens[3]);

                if(temp == -1){

                    result << assemble_I(tokens[0], iter->second,
tokens[1], tokens[2], tokens[3]) << endl;

                } else {

                    int relative_addr = temp - (0x400000 + ((phase2Counter
*4) + 4));
                    result << assemble_I(tokens[0], iter->second,
tokens[1], tokens[2], to_string(relative_addr/4)) << endl;

                }
```

Tester.cpp and Makefile

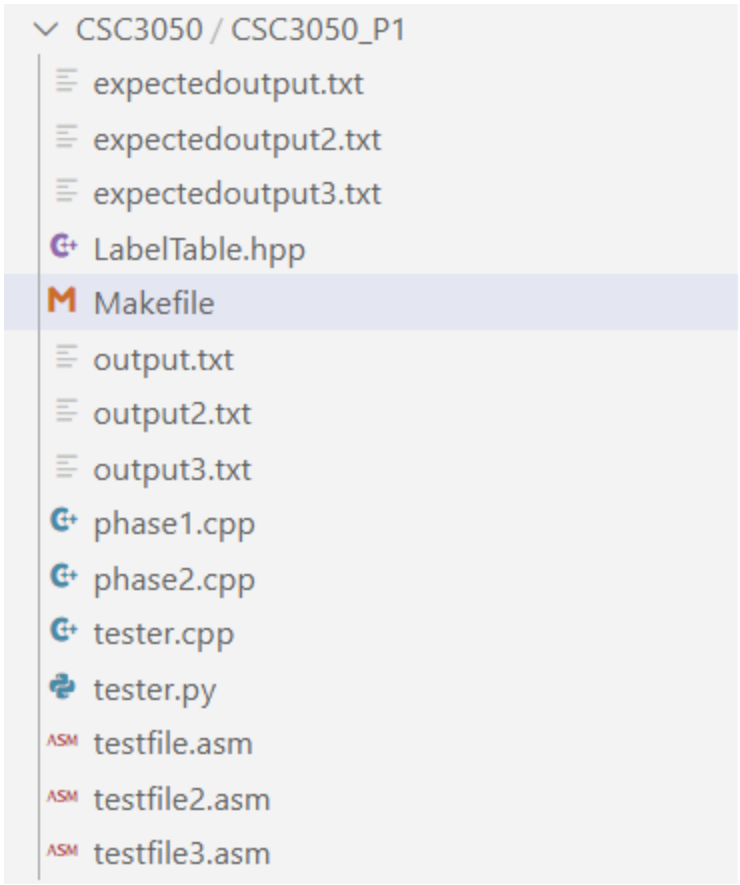
The tester is inspired from the file provided but has been updated to match my own implementation so I feel obligated to comment on that:
After compiling is done we can run the tester to translate MIPS to machine code in this way:
`./testx input.asm output.txt`

This will read the input.asm file and write the result in the output.txt file (creating it if it doesn't already exist)
Another way to run it is to also provide an expected output as such :
`./testx input.asm output.txt expectedoutput.txt`
This will do the same as the last statement. And it will also print on the Terminal the result of the comparison between the output file and its expected counterpart.
To simplify this even further and remove the need to write long lines into the terminal we also provide a makefile.
The make routine will compile and link the c++ code and leave us with a “testx” executable (x for executable)
make clean : will delete this executable along with all the object files. (NOTE: clean intentionally does not remove any output txt files ! this is my own implementation preference, as the output file gets edited each time rather than making many outputs files)
make expect : will run the executable (if not cleaned) in the 2nd style, it will test if the output file is exactly the same as the expectation, print the verdict, then it will **delete** that output. As this routine's sole purpose is to verify if the program can find correct output.
In the above command it requires the existence of a “testfile.asm” and “expectedoutput.txt”

Personally I would advise to just use make and run the testx executable on all the test files you may desire to do as that seems more flexible.

Screenshots

```
~/h/C/CSC3050_P1 make
g++ -c tester.cpp -w
g++ -c phase1.cpp -w
g++ -c phase2.cpp -w
g++ tester.o phase1.o phase2.o -o testx -w
~/h/C/CSC3050_P1 make expect
./testx testfile.asm output.txt expectedoutput.txt
ALL PASSED! CONGRATS :)
rm output.txt
~/h/C/CSC3050_P1 ./testx testfile.asm output.txt expectedoutput.txt
ALL PASSED! CONGRATS :)
~/h/C/CSC3050_P1 ./testx testfile2.asm output2.txt expectedoutput2.txt
ALL PASSED! CONGRATS :)
~/h/C/CSC3050_P1 ./testx testfile3.asm output3.txt expectedoutput3.txt
ALL PASSED! CONGRATS :)
~/h/C/CSC3050_P1 make clean
rm *.o testx
~/h/C/CSC3050_P1
```


CSC3050 / CSC3050_P1
expectedoutput.txt
expectedoutput2.txt
expectedoutput3.txt
LabelTable.hpp
Makefile
output.txt
output2.txt
output3.txt
phase1.cpp
phase2.cpp
tester.cpp
tester.py
testfile.asm
testfile2.asm
testfile3.asm