Lab 1 :: FISCAS and FISCSIM

Konsing Ham Lopez, 1037, ECS 154A 001

Table of Contents

Introduction

In this lab, I made a total of two files in c++ to simulate the assembly process of a CPU. The first is called fiscas.cpp. It works by taking in an assembly file (.s) and transforming it into a hex (.h) file. This process required taking in the assembly file line by line, and translating it to 8-bit binary numbers. The 8-bit binary numbers each reserved 2 bits for the instructions, register rm, register rn, and register rd, in that order. Fiscas supports an optional command line argument "-l" which prints out the label list, hex, address, and assembly file instructions. The second file is called fiscsim.cpp. True to its name, the fiscsim program simulates what the CPU would do with the hex code that we generated in the first program, and displays the cycles, program counter, BNZ flag, and four registers (ranging from r0 to r4). Fiscsim also has an optional command "-d" which lists the disassembly and an optional integer input to simulate a specific number of cycles.

Theory of Operation

fiscas.cpp

My fiscas program contains everything within the main function. I first declare 2 strings, line and parsed. Line is the string that will be used with the getline to temporarily hold the strings from the assembly files. Parsed will hold the 'parsed' section of the line (the instruction, and 3 registers. I then declare 3 maps (symbolTable which holds the labels and their corresponding address, order which contains the address and instructions, and orderFinal which contains address and hex). After this, I check if the user does not provide between 2-3 command line arguments with a simple if/else statement. If the user does provide the appropriate amount of arguments, I pass the assembly file into the ifstream and open it. Then I read the file line by line with getline and convert it to lower case to prevent uppercase - lowercase incompatibility. Once this is complete, I begin the parsing process. I first look for the comments and delete them entirely until the last position of that lie (including the white space). If no comment is present, I just grab the entire line and place it in the parsed string. I then start an if else statement that parses the newly parsed string. First I check if the new parsed string is empty, if it is, I just continue to the next iteration of the loop. If it isn't empty, I first if the string is less than 9 characters (which would mean that this line has no instructions). If this is the case, I append white spaces so that I can still perform the parsing operations for the registers and instructions. Once this is complete, I've made

sure that my string has instructions or white space, so I grab a sub string of this string that contains everything starting from the 8th character. I then make another substr for just the first 3 characters which correspond to the instruction. I use if/elseif statements to check for the instructions. For example, if the parsed instruction is equivalent to "add", we know that it's an add operation. However, if the operation is a white space (no operation), I remove everything after the label if there is one. Otherwise, I just go to the next iteration with a continue statement. Once this loop has finished, I store the newly identified label into the symbolTable map and check for duplicate labels. I then provide an else statement for the white space instruction (if) scenario and return an error for incorrect formatting. At this point, I get the label for the add, or, bnz, and add cases since I had previously only acquire the no instruction scenario's label. Once again, I check for duplicate labels. This concludes the first while loop. To recap, this loop simply gets the label and stores it in the map if it's valid.

The next part of my program clears the getline and positions it at the top. Then I begin my second iteration of the file with the getline. This while loop also converts the strings to lower case and removes the comments in the same way mentioned above. I also parse the string with substr, this time doing so 4 times to extract the 3 registers and the instruction. I store these into the 8 bit containers with the help of bitshifting. For example, if I want to store NOT, I store the int value of NOt (2 in decimal, stored in the notVar int variable) and shift it to the left 6 times. I then do this for all the other registers and their appropriate int value (r0 being 00, r1 being 01, and so on). I also make sure to store the address and instructions into the order map after every iteration. I also have if/else statements that check to make sure that the registers exist. For example, a NOT instruction would not use the rm register because it only takes 2 registers. In the first register check, I made sure to include an else statement that would handle errors such as no instruction present, and in this case, store the label in the map if needed. I do a similar process with register two and three.

Once all 3 of these register locations have been checked, I perform an OR operation on all the newly identified registers and instructions. I then store it in the orderFinal map that contains the address and hex values of the instructions. I then open the output file and check if it's open (returning an error if not) and store the hex value of each instruction in the lines. I also wrote the required "v2.0 raw" line in top line of the file. At the very end of the code, I check if the user provided a 4th command line operator and check if it's -l. If it isn't I return an error. If it is "-l", I print the label list and machine program.

fiscsim.cpp

I begin my fisc simulator program with a checkDigit function at the top which checks if a string is an int value or just a string (I use this later for checking command line operator arguments 3 and 4). The beginning of my main function contains many variables that will hold all instructions (inst, rn, rm, rd), the cycle, userCycle for user specified cycles, the program counter, and the branch target address. I also declare 2 8 bit int arrays (one for the 64 instruction memory slots and one for the 4 registers). I also declare some strings for holding command line arguments 3 and 4 if they exist. Lastly, I have a string for the parsed line that will be taken in and 2 boolean values (one that determines if I should display disassembly and 1 for the z flag.

Onto the code, I first check if the user specified the correct number of command line arguments (3 or 4) and return an error otherwise. After this, I made an if/elseif statement that checks the validity of the command line arguments and the order. My program accepts an int value or "-d" in either the 3rd or 4th command line argument location and in any order. It does not accept 2 "-d" or 2 int values. The if else statements check this in particular by using the checkDigit function that I mentioned above. For example, if only 3 command line arguments are present, I check if it's a "-d" and set the disassembly boolean value to true accordingly. If it's not, I use checkdigit to see if it's a value and set the user cycles accordingly. Otherwise I return an error. I also have an else if branch that checks if the user provided 4 arguments. Within this branch, the first thing I do is check if both arguments 3 and 4 are numbers or "-d" (in which case I return an error). After this, I check if the 3rd argument is "-d" and set the bool to true if so. Otherwise, I check if it's a digit. I then perform the exact same process on the 4th argument. I don't have to worry about error checking in these 2 because I already did so above.

I then open the input hex file that was stored in the hexFile string. If it opens properly, I use getline to get rid of the "v2.0 raw" line that we implemented in fiscas. After this we begin the while loop that simply stores the hex values into the registers. This process requires converting the hex to int beforehand (with the help of stringstream). I then set the cycles to 21 to prevent the off by 1 error if the user doesn't specify a particular number of cycles. Once this is complete, I begin the final while loop that handles the majority of the printing and logic behind the simulator. This while loop runs for as long as the user specified (or a default of 20) cycles. I extract the first instruction by shifting 6 bits to the right and determining if its a BNZ, ADD, OR, or AND. If the

instruction is a BNZ, I set my branch target address the rest of that particular index of the instruction memory. I extract it by performing an AND operation with the number 63 because 63 is 0011 1111 in binary, which means an AND operation would extract only the address bits. Before storing this BTA in my PC, I check if the z flag is zero or not. If z flag is not 0, I do it (perform the jump). The else part of this statement handles the scenario where its not a BNZ operation. So in this case, we get rd, rn, and rm if the instruction is not a NOT operation (since NOT does not use the rm bits). I extract these registers by shifting accordingly (rd no shift, rn 4 shift, rm 2 shift) and performing an AND operation with 3. I do this because 3 is 11 in binary and an AND operation would extract only the portion I want after shifting. Once all the registers and the instruction have their appropriate values, I perform the actual operation with an if elseif statement. If it's an add operation, I ADD register rn and rm and store into rd. If its AND, I and rn and rm and store in rd. If it's a NOT operation I not rn and store in rd. I then set the zflag depending on what rd currently has (if 0, true, if 1 false). Lastly, I have an if statement that checks the boolean value of printDisassembly. If it's set to true, I print all the disassembly, otherwise I don't. Before the very end of this for loop I increment my counter.

Discussion

Fibo1.s

1. *What is the last valid Fibonacci number output by your code?*

The last valid Fibonacci output of my assembly code is 233 (in decimal), or E9 (in hexadecimal). The reason behind this is simple, the 8 bit limitation. The largest number you can represent with 8 bits is 255 (total of 256 numbers, 0-255). This means that the Fibonacci sequence's next number, 377 (in decimal), cannot be represented due to the aforementioned limitation. In order to correctly print 377, we would need a minimum of 9 bits to display the binary (01111001).

2. *How many cycles does it take to compute this number?*

My program takes a total of 36 cycles to print the first instance of E9 (in hexadecimal) or 233 (in decimal).

3. *What value is output for the first invalid Fibonacci number?*

The value of the first invalid Fibonacci number is 79 (in hex) or 121 (in decimal) or 01111001 (in binary).

4. *Explain why this number is invalid, that is, what goes wrong with the arithmetic computation?*

When my program attempts to calculate the next number, 377 (in decimal), it performs an addition of the previous 2 numbers (144 + 233) and prints 121 (in decimal) or 79 in (hexadecimal). This is due to the bit overflow, which gets rid of the left-most "1" in 101111001, the binary representation of decimal number 377. The result is 01111001 or (121) in decimal. We get rid of the left-most bit because we are limited to 8 bit registers.

Fibo2.s

Outline your thought process for minimizing this code, what did you try first, what problems did you encounter?

I realized early on that I had to implement a for-loop similar to that of fibo1.s if I wanted to optimize my code as much as possible. I still tried doing the entire Fibonacci process without any loops in order to gain a deeper understanding of what exactly I had to do. Ultimately, I came to the conclusion that I needed a loop to prevent myself from writing the same 5 lines that I utilized to reset negate the r1 register that would then add 1 to the counter. Unfortunately, an issue arose with the printing of r1 due to this optimization; r1 would print the wrong value on the 2nd print cycle. I fixed this by including the r1 print statement at the beginning of the negate-loop "update".

Give at least one limitation of the FISC instruction set that made this problem somewhat challenging?

The limitation of the FISC instruction set is that it only has 4 registers and 4 instructions (and, add, bnz, not). To elaborate, I knew I needed to keep a counter in one of the registers in order to stop the Fibonacci sequence at the 12th number. This meant that I only had 3 registers to work with, one of them being R3 (rd) which I had to be careful with considering that it would serve 2 purposes at once.

Think of one new instruction that might make this problem easier to solve

An instruction that allows to store the value 1 directly without having to bit shift to get to that number would be really helpful for this problem.

1. *Explain how this instruction would help.*

This instruction would allow me to store the value 1 into a register directly instead of having to negate and bit shift. Specifically, This instruction would fit perfectly on lines 25-29 and would replace them completely.

2. *Outline how you think that this instruction would fit into the current design*

Since the NOT instruction only uses the left-most 6 bits, we could store 01 in the 2 right most bits. I would require a check statement that would check if the "NOT" binary strings have 00 or 01 on the 2 right-most bits. If it has 01 in the 2 right-most bits, this would be my new instruction, which I shall call "ONE." If it has 00 in the 2 right-most bits, it would be a NOT instruction. For example, 1000 0000 would be a NOT operation. In contrast, 1000 0001 would be a ONE operation. (Disclaimer: No registers specified in this example).

3. *How many arguments does your instruction take? (Note, you must be able to fit it into the current design, think carefully about this)*

My instruction would just take 1 argument, Rd. The NOT instruction itself uses Rd and Rn. In other words, it performs the NOT operation on Rn and stores it in Rd. My instruction, ONE, would take just 1 argument and would store 1 in the specified Rd register.

4. *Give an example of a line of assembly using your new opcode and describe what the line is doing.*

To put this into perspective, if I was to use this in line 25 of my fibo2.s, I would do "ONE r1", which would then store the value 1 to r1. This would allow me to remove lines 25-29.

5. *What is the opcode of your new instruction?*

The opcode for this instruction would be the same as NOT, but would include a 01 in the 2 right-most bits. For example, 1000 0000 would be

a NOT operation. In contrast, 1000 0001 would be a ONE operation. (Disclaimer: No registers specified in this example).

6. *Give the machine code for your line of code in (2). Note: this must be correct, your instruction must correctly fit into the opcode table using, as yet unused, extra bits. Think carefully about the FISC design.*

The OP code for this movie would be the same as NOT, but would use the 2 right-most bits which are unused in the NOT operation. Essentially, we would check if it's a NOT operation, then check the 2 right-most bits with an if/elseif. If 00, it's NOT, if 01, it's the new ONE operation.

Conclusion

This assignment used my previous knowledge of assembly language, bitwise operations, and C++ skills to generate an assembler and simulator of a four instruction computer. I gained a deeper understanding of how the assembler takes in the assembly language and translates it to hex. I was then able to experiment with various bitwise operations and discovered some interesting methods, such as performing an AND operation with 3 to extract the right-most bits. Fiscsim in particular helped with solidifying the cycles, program counter, and registers. Fibo1 helped me understand the basics of how you perform the ADD, OR, BNZ, and AND operations on the registers. Fibo2 expanded on that knowledge and required an ingenious way of keeping a counter and then utilizing the r3 or (rd) register as a temporary storage. I also discovered that the assemblers output hexadecimal. In general, I believe that I improved on my coding skills. Particularly in string parsing thanks to newly learned utilities such as substr(), string::npos, and .find(). I also solidified my knowledge of maps and how to iterate through 2 of them through a single for loop (which I didn't know was possible). In other words, this was a good refresher of previous concepts and new ones alike. I enjoyed the fibo parts the most as they took much less time than I had anticipated. Roughly 80% of the 55 hours spent was on fiscas/fiscsim. Thankfully, it works error free from the various tests I performed and only suffers from minor repetition of code. To improve, I think I could make a few functions for those repetitive tasks that I did in fiscas and fiscsim such as parsing for registers.

References

GeeksForGeeks - Extract ‘k’ bits from a given position in a number.
https://www.geeksforgeeks.org/extract-k-bits-given-position-number/

CPlusPlus.com - Check if a number is a digit or not
https://cplusplus.com/forum/beginner/85682/

Appendices

fibo1.s

```
start:
        not r0 r1       ; negate r1
        and r0 r0 r1    ; r0 is 0000 0000
        not r1 r0       ; r1 is 1111 1111
        add r1 r1 r1    ; r1 is 1111 1110
        not r1 r1       ; r1 is 0000 0001
        and r3 r0 r0    ; display r0
loop:
        add r2 r0 r1    ; add r0 and r1 and store in r2
        and r3 r2 r2    ; print r2
        add r0 r1 r2    ; add r1 and r2 and store in r0
        and r3 r0 r0    ; print r0
        add r1 r2 r0    ; add r2 and r0 and store in r1
        and r3 r1 r1    ; print r1
        add r2 r0 r1    ; add r0 and r1 and store in r2
        bnz loop        ; jump back to loop
```

fibo2.s

```
start:
        not r0 r1       ; negate r1
        and r0 r0 r1    ; r0 has 0000 0000
        not r1 r0       ; r1 is 1111 1111
        add r1 r1 r1    ; r1 is 1111 1110
        not r1 r1       ; r1 has 0000 0001
        and r3 r0 r0    ; print r0 (0)
        add r2 r1 r1    ; r2 0000 0010
        add r2 r2 r2    ; r2 0000 0100
        add r2 r2 r2    ; r2 0000 1000
        add r2 r1 r2    ; r2 0000 1001
        add r2 r1 r2    ; r2 0000 1010
        add r2 r1 r2    ; r2 0000 1011
        add r2 r1 r2    ; r2 0000 1100
        not r2 r2       ; r2 is 1111 0011 (-12)
        add r2 r2 r1    ; add 1 to r2 because we printed r0
        bnz update      ; jump to update
loop:
        and r1 r3 r3    ; reset r1 to it's previous value
```

```
        add r0 r1 r0    ; r0 is r1 + r0 first fib
        add r1 r0 r1    ; r3 is r0 + r1 2nd fib
        and r3 r0 r0    ; print r0
update:
        and r3 r1 r1    ; print r1
        not r1 r0       ; negate r1
        and r1 r0 r1    ; r1 has 0000 0000
        not r1 r1       ; r1 is 1111 1111
        add r1 r1 r1    ; r1 is 1111 1110
        not r1 r1       ; r1 has 0000 0001
        add r2 r2 r1    ; add 1 to r2 because we printed r0
        add r2 r2 r1    ; add 1 to r2 because we printed r1
        bnz loop        ; go back to loop if not r3 not 0
        and r3 r3 r3    ; resets z flag to be NOT 0
end:
        bnz end         ; loop over and over again
```

fibo1 output

```
Cycle:79 State:PC:18 Z:0 R0: 37 R1: 22 R2: FE R3: 22
Disassembly: add r0 r1 r0

Cycle:80 State:PC:19 Z:0 R0: 37 R1: 59 R2: FE R3: 22
Disassembly: add r1 r0 r1

Cycle:81 State:PC:20 Z:0 R0: 37 R1: 59 R2: FE R3: 37
Disassembly: and r3 r0 r0

Cycle:82 State:PC:21 Z:0 R0: 37 R1: 59 R2: FE R3: 59
Disassembly: and r3 r1 r1

Cycle:83 State:PC:22 Z:0 R0: 37 R1: C8 R2: FE R3: 59
Disassembly: not r1 r0

Cycle:84 State:PC:23 Z:1 R0: 37 R1: 00 R2: FE R3: 59
Disassembly: and r1 r0 r1

Cycle:85 State:PC:24 Z:0 R0: 37 R1: FF R2: FE R3: 59
Disassembly: not r1 r1

Cycle:86 State:PC:25 Z:0 R0: 37 R1: FE R2: FE R3: 59
Disassembly: add r1 r1 r1

Cycle:87 State:PC:26 Z:0 R0: 37 R1: 01 R2: FE R3: 59
Disassembly: not r1 r1

Cycle:88 State:PC:27 Z:0 R0: 37 R1: 01 R2: FF R3: 59
Disassembly: add r2 r2 r1

Cycle:89 State:PC:28 Z:1 R0: 37 R1: 01 R2: 00 R3: 59
Disassembly: add r2 r2 r1

Cycle:90 State:PC:29 Z:1 R0: 37 R1: 01 R2: 00 R3: 59
Disassembly: bnz 16
```

```
Cycle:91 State:PC:30 Z:0 R0: 37 R1: 01 R2: 00 R3: 59
Disassembly: and r3 r3 r3

Cycle:92 State:PC:30 Z:0 R0: 37 R1: 01 R2: 00 R3: 59
Disassembly: bnz 30

Cycle:93 State:PC:30 Z:0 R0: 37 R1: 01 R2: 00 R3: 59
Disassembly: bnz 30

Cycle:94 State:PC:30 Z:0 R0: 37 R1: 01 R2: 00 R3: 59
Disassembly: bnz 30

Cycle:95 State:PC:30 Z:0 R0: 37 R1: 01 R2: 00 R3: 59
Disassembly: bnz 30

Cycle:96 State:PC:30 Z:0 R0: 37 R1: 01 R2: 00 R3: 59
Disassembly: bnz 30
```

fibo2 output

```
Cycle:30 State:PC:06 Z:0 R0: 22 R1: 37 R2: 59 R3: 37
Disassembly: bnz 06

Cycle:31 State:PC:07 Z:0 R0: 22 R1: 37 R2: 59 R3: 37
Disassembly: add r2 r0 r1

Cycle:32 State:PC:08 Z:0 R0: 22 R1: 37 R2: 59 R3: 59
Disassembly: and r3 r2 r2

Cycle:33 State:PC:09 Z:0 R0: 90 R1: 37 R2: 59 R3: 59
Disassembly: add r0 r1 r2

Cycle:34 State:PC:10 Z:0 R0: 90 R1: 37 R2: 59 R3: 90
Disassembly: and r3 r0 r0

Cycle:35 State:PC:11 Z:0 R0: 90 R1: E9 R2: 59 R3: 90
Disassembly: add r1 r2 r0

Cycle:36 State:PC:12 Z:0 R0: 90 R1: E9 R2: 59 R3: E9
Disassembly: and r3 r1 r1

Cycle:37 State:PC:13 Z:0 R0: 90 R1: E9 R2: 79 R3: E9
Disassembly: add r2 r0 r1

Cycle:38 State:PC:06 Z:0 R0: 90 R1: E9 R2: 79 R3: E9
Disassembly: bnz 06

Cycle:39 State:PC:07 Z:0 R0: 90 R1: E9 R2: 79 R3: E9
Disassembly: add r2 r0 r1

Cycle:40 State:PC:08 Z:0 R0: 90 R1: E9 R2: 79 R3: 79
Disassembly: and r3 r2 r2
```

```
Cycle:41 State:PC:09 Z:0 R0: 62 R1: E9 R2: 79 R3: 79
Disassembly: add r0 r1 r2

Cycle:42 State:PC:10 Z:0 R0: 62 R1: E9 R2: 79 R3: 62
Disassembly: and r3 r0 r0
```

fiscas.cpp

```
#include <fstream>
#include <iomanip>
#include <iostream>
#include <map>
#include <string>
#include <cstring>
#include <algorithm>
using namespace std;

int main(int argc, char* argv[]) {
  string line;
  string parsed;
  int operation = 0;
  int operationFirst = 0;
  // symbolTable contains Label and Address
  map<string, int> symbolTable;
  // order contains address and instructions
  map<int, string> order;
  // will contain addresses and hex
  map<int, string> orderFinal;

  if(argc < 3 || argc > 4){
    cerr << "Usage: fiscas <asm file> <hex file> [-l]\n";
    return -1;
  }

  ifstream file(argv[1]);

  if (file.is_open()) { //if file open run 2 while loops
    while (getline(file, line)) { //first while loop, fill symbolTable
      //convert line to lower case
      for_each(line.begin(), line.end(), [](char & c) {
        c = ::tolower(c);
      });
      //clear the comments
      if (line.find(';') != string::npos) {
        int semiCPos = line.find(';');
        line.erase(semiCPos, string::npos);
        parsed = line.substr(0, semiCPos);
      } else {
        parsed = line.substr(0, string::npos);
      }

      //if parsed strring empty, continue, else extend it
```

```
//and checkfor instructions
if (parsed.empty()) {
  continue;
} else {
  string parseInstructions, parseInstruction;

  //if string is less tha 9 wide, extend
  if(parsed.size() < 9){
    parsed.append("            ");
  }

  //get the instruction
  parseInstructions = parsed.substr(8, string::npos);
  parseInstruction = parseInstructions.substr(0, 3);

  //check if its a proper instruction
  if (parseInstruction == "add") {
    operationFirst++;
  } else if (parseInstruction == "and") {
    operationFirst++;
  } else if (parseInstruction == "not") {
    operationFirst++;
  } else if (parseInstruction == "bnz") {
    operationFirst++;
  } else if (parseInstruction == "   ") {
    //get the labels and check for duplicates
    if (parsed.find(':') != string::npos) {
      int colonPos = parsed.find(':');
      parsed.erase(colonPos, string::npos);
      parsed = parsed.substr(0, colonPos);
    } else {
      //if not an instruction, go to next
      continue;
    }

    // store and prevent duplicate labels (labels stored only)
    for (auto i = symbolTable.begin(); i != symbolTable.end(); ++i){
      if (parsed == i->first) {
        cerr << "You cannot have duplicate labels\n";
        return -1;
      }
    }

    symbolTable.emplace(parsed, operationFirst);

    continue;
  } else {
    cerr << "Instruction formating error." << endl;
    return -1;
  }

  // get the labels
  if (parsed.find(':') != string::npos) {
    int colonPos = parsed.find(':');
    parsed.erase(colonPos, string::npos);
```

```
      parsed = parsed.substr(0, colonPos);
    } else {
      continue;
    }

    // store and prevent duplicate labels (labels stored only)
    for (auto i = symbolTable.begin(); i != symbolTable.end(); ++i){
      if (parsed == i->first) {
        cerr << "You cannot have duplicate labels\n";
        return -1;
      }
    }

    symbolTable.emplace(parsed, operationFirst - 1);
  } // end of else statement*/
} // end of ifirst while loop

// clear the getline for 2nd pass through the file
file.clear();
file.seekg(0, file.beg);

// second pass through file, remove comments and
//get the hex code
while (getline(file, line)) {
  //convert line to lower case
  for_each(line.begin(), line.end(), [](char & c) {
    c = ::tolower(c);
  });
  int addVar = 0;
  int andVar = 1;
  int notVar = 2;
  int bnzVar = 3;

  int r0Var = 0;
  int r1Var = 1;
  int r2Var = 2;
  int r3Var = 3;

  int integerValue = 0;
  string rest = "", rest2 = "", rest3 = "", rest4 = "", rest5 = "";
  int iTempVar1 = 0, iTempVar2 = 0, iTempVar3 = 0, iTempVar4 = 0;

  //delete comments
  if (line.find(';') != string::npos) {
    int semiCPos = line.find(';');
    line.erase(semiCPos, string::npos);
    parsed = line.substr(0, semiCPos);
  } else {
    parsed = line.substr(0, string::npos);
  }

  if (parsed.empty()) {
    continue;
  } else if (parsed.size() < 9) {
      // instructions empty
```

```
  continue;
} else {
  // check if instructions valid,
  //if so, increase operation and shift bits
  rest = parsed.substr(8, string::npos);
  rest2 = rest.substr(0, 3);

  if (rest2 == "add") {
    operation++;
    iTempVar1 = addVar;
    iTempVar1 = iTempVar1 << 6;
  } else if (rest2 == "and") {
    operation++;
    iTempVar1 = andVar;
    iTempVar1 = iTempVar1 << 6;
  } else if (rest2 == "not") {
    operation++;
    iTempVar1 = notVar;
    iTempVar1 = iTempVar1 << 6;
  } else if (rest2 == "bnz") {
    operation++;
    iTempVar1 = bnzVar;
    iTempVar1 = iTempVar1 << 6;
  } else if (rest2 == "   ") {
    continue;
  } else {
    cerr << "Instruction formating error." << endl;
    return -1;
  }

  order[operation - 1] = rest;

  // if register 1 exists, create a substr of it
  if (rest.size() >= 6) {
    rest3 = rest.substr(4, 2);
  } else {
    cerr << "Wrong formatting after instruction.\n";
    return -1;
  }

  // first register
  if (rest3 == "r0" || rest3 == "  ") {
    iTempVar2 = r0Var;
    iTempVar2 = iTempVar2;
  } else if (rest3 == "r1") {
    iTempVar2 = r1Var;
    iTempVar2 = iTempVar2;
  } else if (rest3 == "r2") {
    iTempVar2 = r2Var;
    iTempVar2 = iTempVar2;
  } else if (rest3 == "r3") {
    iTempVar2 = r3Var;
    iTempVar2 = iTempVar2;
  } else {
    // extend the line with white space
```

```
  if (rest.size() < 14) {
    rest.append("              ");
  }
  // grab the label in Rd register position (first register)
  string removeFirstInst = rest.substr(4, 10);
  int finalChar = removeFirstInst.find(' ');
  removeFirstInst = removeFirstInst.substr(0, finalChar);

  // check if the label is in the map, if it is, get the address
  //"OR" it with the instruction, and convert it to he
  bool checkIfExists = false;
  for (auto i = symbolTable.begin(); i != symbolTable.end(); ++i) {
    if (removeFirstInst == i->first) {
      checkIfExists = true;
      integerValue = (iTempVar1 | i->second);

      stringstream ss;
      ss << setfill('0') << setw(2) << uppercase
                  << hex << integerValue;
      orderFinal[operation - 1] = ss.str();
    }
  }

  if(checkIfExists == false){
    cerr << "That label in the Rd register does NOT exist.\n";
    exit(0);
  }

  continue;
}

// add white space if register 2 doesn't exist
if (rest.size() < 8) {
  rest.append("   ");
}
rest4 = rest.substr(7, 2);

// second register
if (rest4 == "r0" || rest4 == "  ") {
  iTempVar3 = r0Var;
  iTempVar3 = iTempVar3 << 4;
} else if (rest4 == "r1") {
  iTempVar3 = r1Var;
  iTempVar3 = iTempVar3 << 4;
} else if (rest4 == "r2") {
  iTempVar3 = r2Var;
  iTempVar3 = iTempVar3 << 4;
} else if (rest4 == "r3") {
  iTempVar3 = r3Var;
  iTempVar3 = iTempVar3 << 4;
} else {
  cerr << "Wrong register name for Rn\n";
  return -1;
}
```

```
    // add white space if third register doesn't exist
    if (rest.size() < 10) {
      rest.append("   ");
    }
    rest5 = rest.substr(10, 2);

    // third register
    if (rest5 == "r0" || rest5 == "  ") {
      iTempVar4 = r0Var;
      iTempVar4 = iTempVar4 << 2;
    } else if (rest5 == "r1") {
      iTempVar4 = r1Var;
      iTempVar4 = iTempVar4 << 2;
    } else if (rest5 == "r2") {
      iTempVar4 = r2Var;
      iTempVar4 = iTempVar4 << 2;
    } else if (rest5 == "r3") {
      iTempVar4 = r3Var;
      iTempVar4 = iTempVar4 << 2;
    } else {
      cerr << "Wrong register name for Rm\n";
      return -1;
    }
    //"OR" all the instructions and store them as hex alongside address
    integerValue = (iTempVar1 | iTempVar2 | iTempVar3 | iTempVar4);
    stringstream ss;
    ss << setfill('0') << setw(2) << uppercase << hex << integerValue;
    orderFinal[operation - 1] = ss.str();
  } // end of the else statement
}   // end of second while loop

//output the hex code to a file
ofstream outFile;
outFile.open(argv[2]);

if(!outFile.is_open()){
    cerr << "Hex file could not be opened.\n";
    return -1;
} else {
  outFile << "v2.0 raw";
  for(auto i = orderFinal.begin(); i != orderFinal.end(); ++i){
      outFile << "\n" << i->second;
  }
}

//if user provides 4th command line operator [-l], print all info
if(argc == 4){
  if(strcmp(argv[3], "-l") == 0){
    // Label map output
    cout << "\n*** LABEL LIST ***\n";
    for (auto i = symbolTable.begin(); i != symbolTable.end(); ++i) {
      cout << left << setfill(' ') << setw(8) << i->first;
      cout << right << setfill('0') << setw(2) << i->second << '\n';
    }
    //Prints the key and val of orderFinal map,
```

```
        //then prints the val of order map (instructions)
        cout << "*** MACHINE PROGRAM ***\n";
        for(auto i = order.begin(), j = orderFinal.begin();
          i != order.end(); ++i, ++j){
          cout << setfill('0') << setw(2) << j->first << ':' << j->second;
          cout << '\t' << i->second << '\n';
        }
      } else {
        cerr << "Not the correct output command. Use -l" << endl;
        return -1;
      }
    }
  } // end of if file.is_open()
  return 0;
}
```

fiscsim.cpp

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
#include <stdint.h>
using namespace std;
//function to check the string is a digit or not
bool checkDigit(string str) {
 for (int i = 0; i < str.length(); i++)
   if (isdigit(str[i]) == false){
      return false;
    }
    return true;
}

int main(int argc, char* argv[]){
  string line; //string holding each line
  int inst, rn, rm, rd; //registers/instruction
  int index = 0;
  int cycle = 0; //cycles
  int userCycle = 0; //holder for custom cycles from user
  int pc = 0; //program counter
  int bta; //branch target address
  uint8_t instructionMemory[64];
  uint8_t registers[4] = {0, 0, 0, 0};
  bool zFlag = true; //true = 1, false = 0
  string argv3, argv4;
  bool printDisassembly = false;

  if(argc < 2 || argc > 4){ //if not 3 or 4 command line args
    cerr << "Usage: ./fiscsim <object file> [cycles] [-d]\n";
    return -1;
  }
```

```
string hexFile = argv[1];

if(argc == 3){ //if 3 arguments check if -d or int
  argv3 = argv[2];
  if(argv3 == "-d"){
    printDisassembly = true;
  } else {
    if(checkDigit(argv3)){
      userCycle = (stoi(argv3) + 1);
    } else {
      cerr << "Usage: ./fiscsim <object file> [cycles] [-d]\n";
      return -1;
    }
  }
} else if(argc == 4){ //if argc 4, check if -d or ints in either
  argv3 = argv[2];
  argv4 = argv[3];

  //if both arguments 3 and 4 are -d, return error
  if(argv3 == "-d" && argv4 == "-d"){
    cerr << "Both command line operators cannot be '-d'\n";
    return -1;
  }
  //if both arguments 3 and 4 are ints, return error
  if(checkDigit(argv3) && checkDigit(argv4)){
    cerr << "Both command line operators cannot be ints\n";
    return -1;
  }
  //if argv3 is -d. print d. Else, check if argv3 is an int.
  //if argv3 is int, this is cycles, else, this is not accepted
  if(argv3 == "-d"){
    printDisassembly = true;
  } else {
    if(checkDigit(argv3)){
      userCycle = (stoi(argv3) + 1);
    } else {
      cerr << "Usage: ./fiscsim <object file> [cycles] [-d]\n";
      return -1;
    }
  }
  //if argv4 is -d. print d. Else, check if argv4 is an int.
  //if argv4 is int, this is cycles, else, this is not accepted
  if(argv4 == "-d"){
    printDisassembly = true;
  } else {
    if(checkDigit(argv4)){
      userCycle = (stoi(argv4) + 1);
    } else {
      cerr << "Usage: ./fiscsim <object file> [cycles] [-d]\n";
      return -1;
    }
  }
}

ifstream inputFile(hexFile);
```

```
if(inputFile.is_open()){
  //Call getline once to get rid of v2.0 raw
  getline(inputFile, line);
  if(line.compare("v2.0 raw") == 0){
    //run
  } else{
    cout << "Hex file does not contain v2.0 raw.\n";
    return -1;
  }

  while(getline(inputFile, line)){ //fill the instruct mem
    stringstream ss;
    ss << line;
    int val;
    ss >> hex >> val;
    instructionMemory[index] = val;
    index++;
  } //end of while getline loop

  if(userCycle == 0){//if no user input cycles, default 20
    userCycle = 21;
  }

  while(cycle < userCycle){
    cycle++;

    if (cycle == userCycle){
      break;
    }
    //get instruction of from bits
    inst = instructionMemory[pc] >> 6;

    if(inst == 3){ //if BNZ, get the address only
      bta = instructionMemory[pc] & 63; //bta holds address
      if(zFlag == true){ //jump (pc change) if z != 0
        //nothing
      } else {
        pc = bta-1;
      }
    } else { //if not BNZ, set rd, rn, and rm
      rd = instructionMemory[pc] & 3;
      rn = (instructionMemory[pc] >> 4) & 3;
      if(inst != 2){ //only set rm if inst is "NOT"
      rm = (instructionMemory[pc] >> 2) & 3;
      }
    }

    if(inst == 0){ //add
      registers[rd] = registers[rn] + registers[rm];
    } else if(inst == 1){//and
      registers[rd] = registers[rn] & registers[rm];
    } else if(inst == 2){//not
      registers[rd] = ~registers[rn];
    }
```

```
if(registers[rd] == 0){ //set the zFlag depending on rd
  zFlag = true;
} else {
  zFlag = false;
}

cout << "Cycle:" << dec << cycle;
if(inst == 3){
  cout << " State:PC:" << setfill('0') << setw(2) << pc+1;
} else {
  cout << " State:PC:" << setfill('0') << setw(2) << pc+1;
}
cout << " Z:" << zFlag;
cout << " R0: " << setw(2) << uppercase << hex
  << (unsigned)registers[0];
cout << " R1: " << setw(2) << uppercase << hex
  << (unsigned)registers[1];
cout << " R2: " << setw(2) << uppercase << hex
  << (unsigned)registers[2];
cout << " R3: " << setw(2) << uppercase << hex
  << (unsigned)registers[3];

if(printDisassembly == true){ //print disassembly if -d
  cout << "\nDisassembly: ";

  if(inst == 0){
    cout << "add ";
  } else if(inst == 1){
    cout << "and ";
  } else if(inst == 2){
    cout << "not ";
  } else if(inst == 3){
    cout << "bnz ";
    cout << setfill('0') << setw(2) << dec << bta;
  }

  if(inst != 3){
    if(rd == 0){
      cout << "r0 ";
    } else if(rd == 1){
      cout << "r1 ";
    } else if(rd == 2){
      cout << "r2 ";
    } else if(rd == 3){
      cout << "r3 ";
    }
  }

  if(inst != 3){
    if(rn == 0){
      cout << "r0 ";
    } else if(rn == 1){
      cout << "r1 ";
    } else if(rn == 2){
```

```
            cout << "r2 ";
          } else if(rn == 3){
            cout << "r3 ";
          }
        }

        if(inst != 2 && inst != 3){
          if(rm == 0){
            cout << "r0 ";
          } else if(rm == 1){
            cout << "r1 ";
          } else if(rm == 2){
            cout << "r2 ";
          } else if(rm == 3){
            cout << "r3 ";
          }
        }
      }

      cout << endl << endl;
      pc++;
    }// end of for loop
  } else {
    cerr << "Could not open '.h' file." << endl;
    return -1;
  } //end of "could not open file" else statement

  return 0;
}
```