CPU Lab 1

Brian Beilby, 3524

CSC137-02, Computer Organization, Spring 2023

04/15/2023

Introduction

The assembler, that I built in Java, translates assembly code into machine code given a source file as input and outputs the corresponding machine code in an object file. It is a two-pass assembler; on the first pass, it puts each label and the label's address into a symbol table that is implemented using a HashMap. On the second pass, it generates machine code for each instruction in the source file. A message explaining how to use the assembler is displayed if the user tries to run it with no arguments. If the input file has errors such as an unsupported instruction or an invalid register, an error message would be displayed.

The simulator, that I also built in Java, represents a CPU. It takes input from a file containing machine code instructions in hexadecimal, loads the instructions into memory, and then executes them. First, the instructions are read in from the file, the state of the CPU is initialized (values such as memory and registers). Then, for a user-specified number of cycles, the instructions are executed and the state of the CPU is printed for each cycle. To extract the fields from the hexadecimal instructions such as their opcodes, bit manipulation was used. The user also has the option to have the instructions disassembled and displayed in human-readable format on every cycle. If the user tries to run the simulator with no arguments, a message explaining how to use the simulator is displayed. If the user inputs an invalid number of cycles, an invalid file, or an invalid option, an error message would be displayed.

Discussion

fibo1 Questions and Answers:

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

The last valid Fibonacci number output by my code was 233 which is E9 in hexadecimal and it is the 14th Fibonacci number in the sequence.

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

It takes 46 cycles to compute the number with my code.

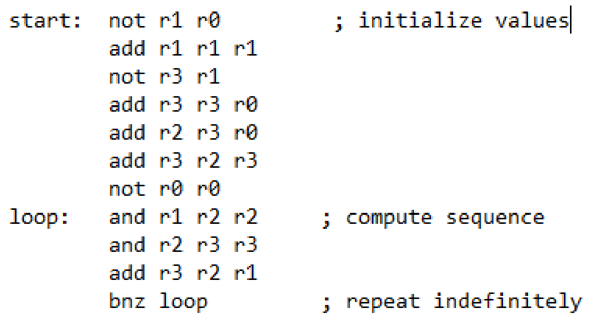
3. *What value is output for the first invalid Fibonacci number?*

The first invalid Fibonacci number output was 121 which is 79 in hexadecimal.

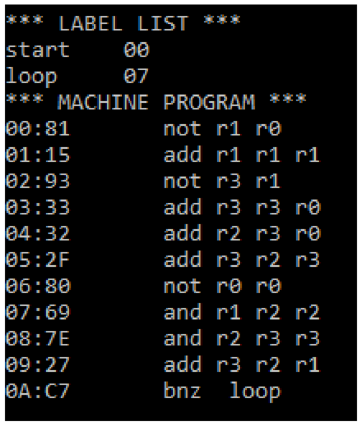
4. *Explain why this number is invalid, that is, what goes wrong with the arithmetic computation?*

I believe the problem is with the registers. The 15th number in the sequence is 377 which exceeds the maximum value that the registers can hold which is 256 so an overflow occurs.

fibo1 Source and List Output:



```
start:  not r1 r0         ; initialize values
        add r1 r1 r1
        not r3 r1
        add r3 r3 r0
        add r2 r3 r0
        add r3 r2 r3
        not r0 r0
loop:   and r1 r2 r2      ; compute sequence
        and r2 r3 r3
        add r3 r2 r1
        bnz loop          ; repeat indefinitely
```



```
*** LABEL LIST ***
start    00
loop     07
*** MACHINE PROGRAM ***
00:81       not r1 r0
01:15       add r1 r1 r1
02:93       not r3 r1
03:33       add r3 r3 r0
04:32       add r2 r3 r0
05:2F       add r3 r2 r3
06:80       not r0 r0
07:69       and r1 r2 r2
08:7E       and r2 r3 r3
09:27       add r3 r2 r1
0A:C7       bnz  loop
```

fibo2 Questions and Answers:

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

   I struggled, at first, with initializing the first two values of the sequence as 0 and 1. Once I figured that out, I figured out how to compute the next value in the sequence by adding the two previous values and storing the sum in r3.

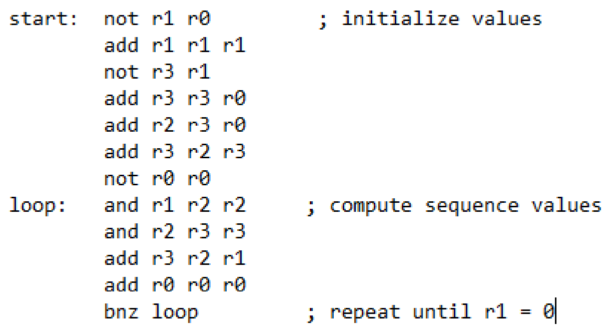
2. *Give at least one limitation of the FISC instruction set that made this problem somewhat challenging?*

   The limited set of arithmetic operations made this problem challenging and also the inability to store constant values in registers.

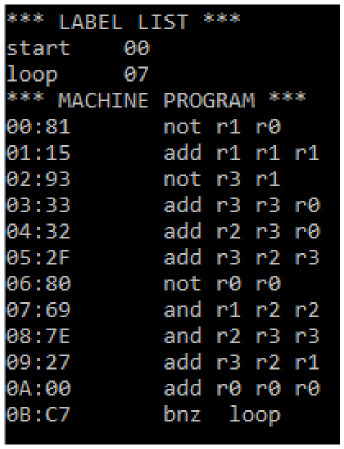
3. *Think of one new instruction that might make this problem easier to solve.*

   An instruction that, I think, would make this problem much simpler is an instruction to shift the value of a register to the right. This would make it possible to shift the value of a register to the right by 1 to get the previous number. This could be added into the current design by shifting r2 right by 1 to get the previous number and also by shifting r3 right by 1 to get the next previous number. This instruction would take two arguments. The first being the register to shift and the second being either a constant value of how many bits to shift by or if that implementation isn't possible, the second argument could be a register holding the number of bits to shift by. A line using this new instruction could be shr r3, r2. This operation would shift the value of r3 to the right by the value in r2. The opcode for this new hypothetical instruction could be 101. The machine code of the line previously mentioned would be 101100011.

fibo2 Source and List Output:



```
start:  not r1 r0           ; initialize values
        add r1 r1 r1
        not r3 r1
        add r3 r3 r0
        add r2 r3 r0
        add r3 r2 r3
        not r0 r0
loop:   and r1 r2 r2      ; compute sequence values
        and r2 r3 r3
        add r3 r2 r1
        add r0 r0 r0
        bnz loop          ; repeat until r1 = 0
```



```
*** LABEL LIST ***
start    00
loop     07
*** MACHINE PROGRAM ***
00:81       not r1 r0
01:15       add r1 r1 r1
02:93       not r3 r1
03:33       add r3 r3 r0
04:32       add r2 r3 r0
05:2F       add r3 r2 r3
06:80       not r0 r0
07:69       and r1 r2 r2
08:7E       and r2 r3 r3
09:27       add r3 r2 r1
0A:00       add r0 r0 r0
0B:C7       bnz  loop
```

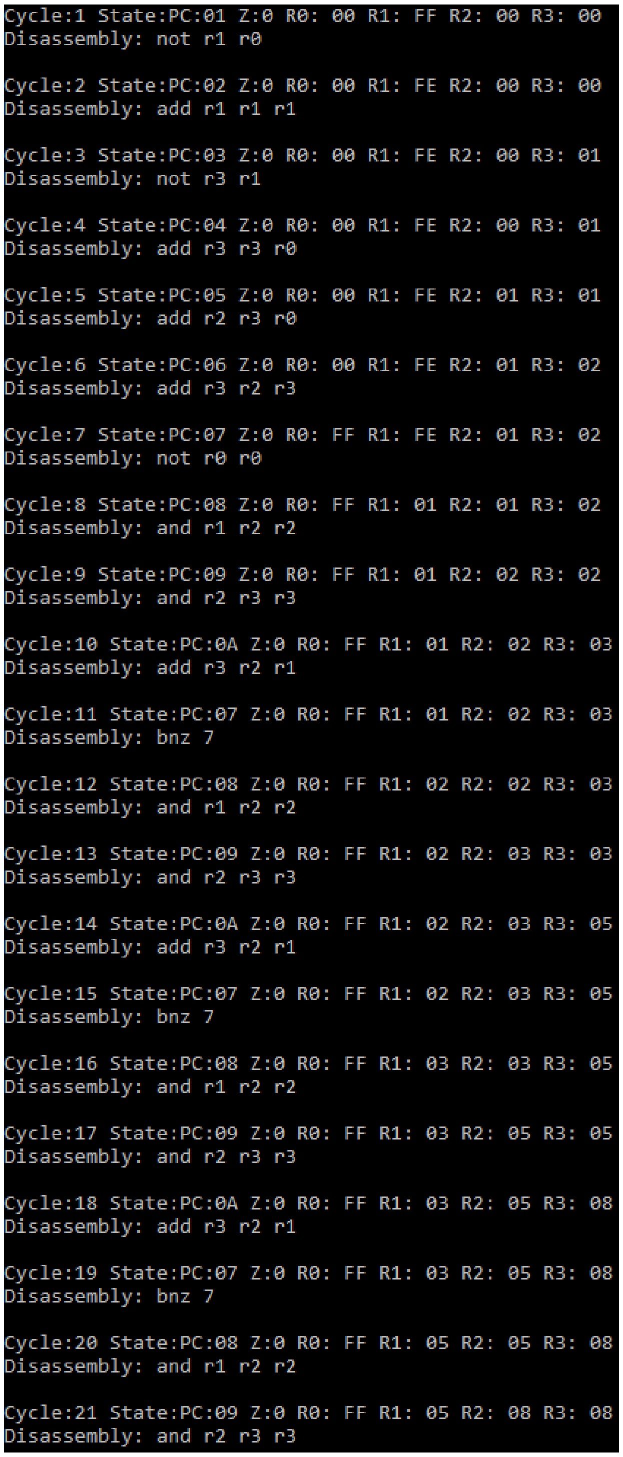
Conclusion

In this lab, I built an assembler and simulator for a simple FISC CPU. I then designed a circuit for the simulator in Logisim. I then wrote two assembly programs and tested my assembler, simulator, and simulator circuit with those assembly programs. Overall, this lab was beneficial in several ways. From building the assembler, I gained a much greater understanding of how assembly code is translated into machine code that a CPU can then execute. From building the simulator, I gained a much greater understanding of how a CPU processes instructions, uses memory, and how it uses its registers. From designing the circuit, I learned how many different individual components can be used to create something much more complex and how they interact together. From writing the assembly and using it to test my assembler and simulator, I gained a better understanding of how to effectively test my code and how to improve upon it.

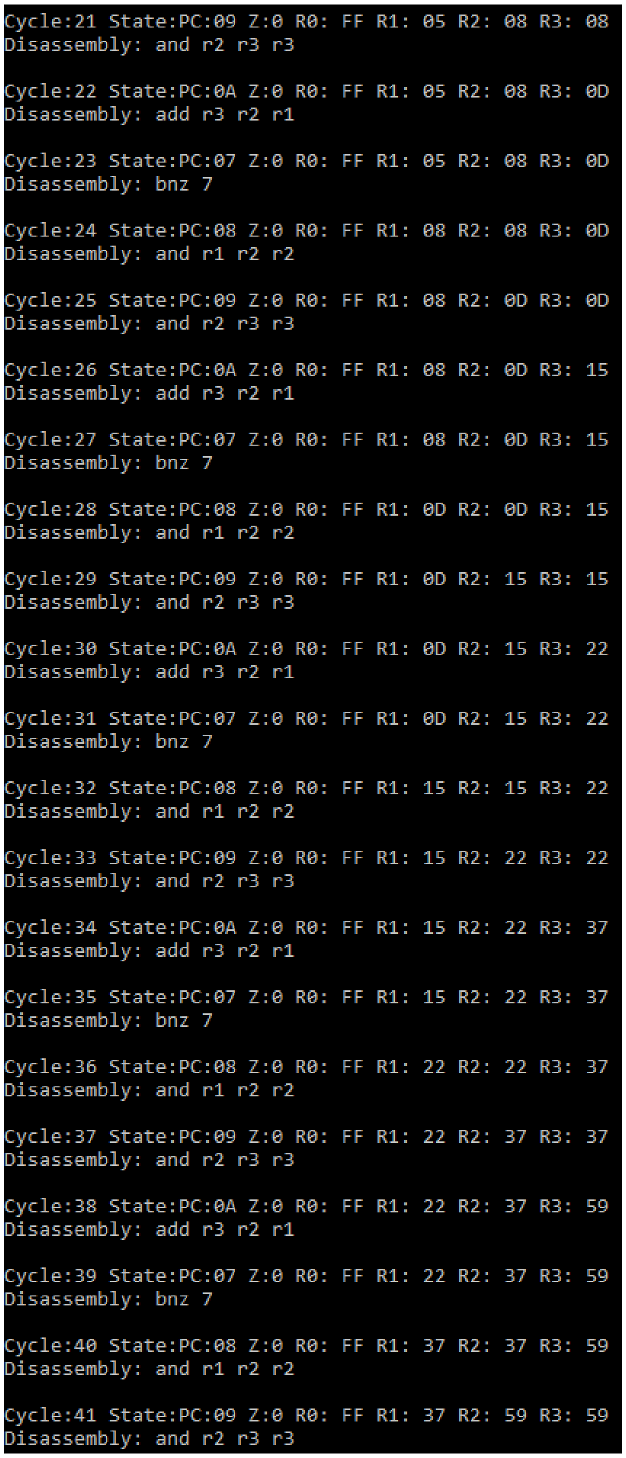
Appendix

Runtime Output of the Simulator With Disassembly:

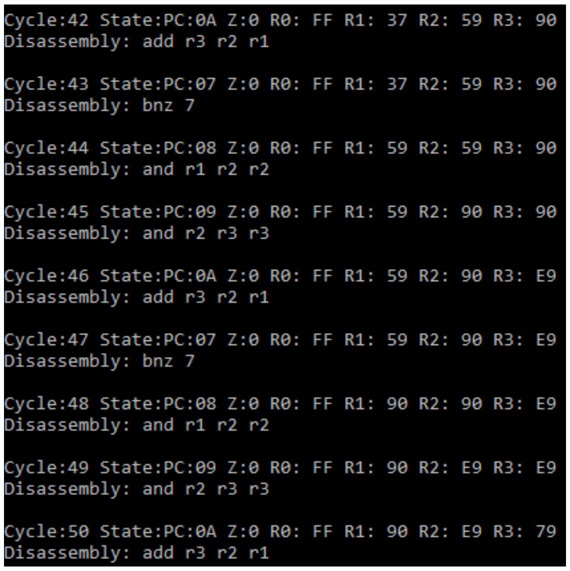
fibo1:



```
Cycle:1 State:PC:01 Z:0 R0: 00 R1: FF R2: 00 R3: 00
Disassembly: not r1 r0

Cycle:2 State:PC:02 Z:0 R0: 00 R1: FE R2: 00 R3: 00
Disassembly: add r1 r1 r1

Cycle:3 State:PC:03 Z:0 R0: 00 R1: FE R2: 00 R3: 01
Disassembly: not r3 r1

Cycle:4 State:PC:04 Z:0 R0: 00 R1: FE R2: 00 R3: 01
Disassembly: add r3 r3 r0

Cycle:5 State:PC:05 Z:0 R0: 00 R1: FE R2: 01 R3: 01
Disassembly: add r2 r3 r0

Cycle:6 State:PC:06 Z:0 R0: 00 R1: FE R2: 01 R3: 02
Disassembly: add r3 r2 r3

Cycle:7 State:PC:07 Z:0 R0: FF R1: FE R2: 01 R3: 02
Disassembly: not r0 r0

Cycle:8 State:PC:08 Z:0 R0: FF R1: 01 R2: 01 R3: 02
Disassembly: and r1 r2 r2

Cycle:9 State:PC:09 Z:0 R0: FF R1: 01 R2: 02 R3: 02
Disassembly: and r2 r3 r3

Cycle:10 State:PC:0A Z:0 R0: FF R1: 01 R2: 02 R3: 03
Disassembly: add r3 r2 r1

Cycle:11 State:PC:07 Z:0 R0: FF R1: 01 R2: 02 R3: 03
Disassembly: bnz 7

Cycle:12 State:PC:08 Z:0 R0: FF R1: 02 R2: 02 R3: 03
Disassembly: and r1 r2 r2

Cycle:13 State:PC:09 Z:0 R0: FF R1: 02 R2: 03 R3: 03
Disassembly: and r2 r3 r3

Cycle:14 State:PC:0A Z:0 R0: FF R1: 02 R2: 03 R3: 05
Disassembly: add r3 r2 r1

Cycle:15 State:PC:07 Z:0 R0: FF R1: 02 R2: 03 R3: 05
Disassembly: bnz 7

Cycle:16 State:PC:08 Z:0 R0: FF R1: 03 R2: 03 R3: 05
Disassembly: and r1 r2 r2

Cycle:17 State:PC:09 Z:0 R0: FF R1: 03 R2: 05 R3: 05
Disassembly: and r2 r3 r3

Cycle:18 State:PC:0A Z:0 R0: FF R1: 03 R2: 05 R3: 08
Disassembly: add r3 r2 r1

Cycle:19 State:PC:07 Z:0 R0: FF R1: 03 R2: 05 R3: 08
Disassembly: bnz 7

Cycle:20 State:PC:08 Z:0 R0: FF R1: 05 R2: 05 R3: 08
Disassembly: and r1 r2 r2

Cycle:21 State:PC:09 Z:0 R0: FF R1: 05 R2: 08 R3: 08
Disassembly: and r2 r3 r3
```

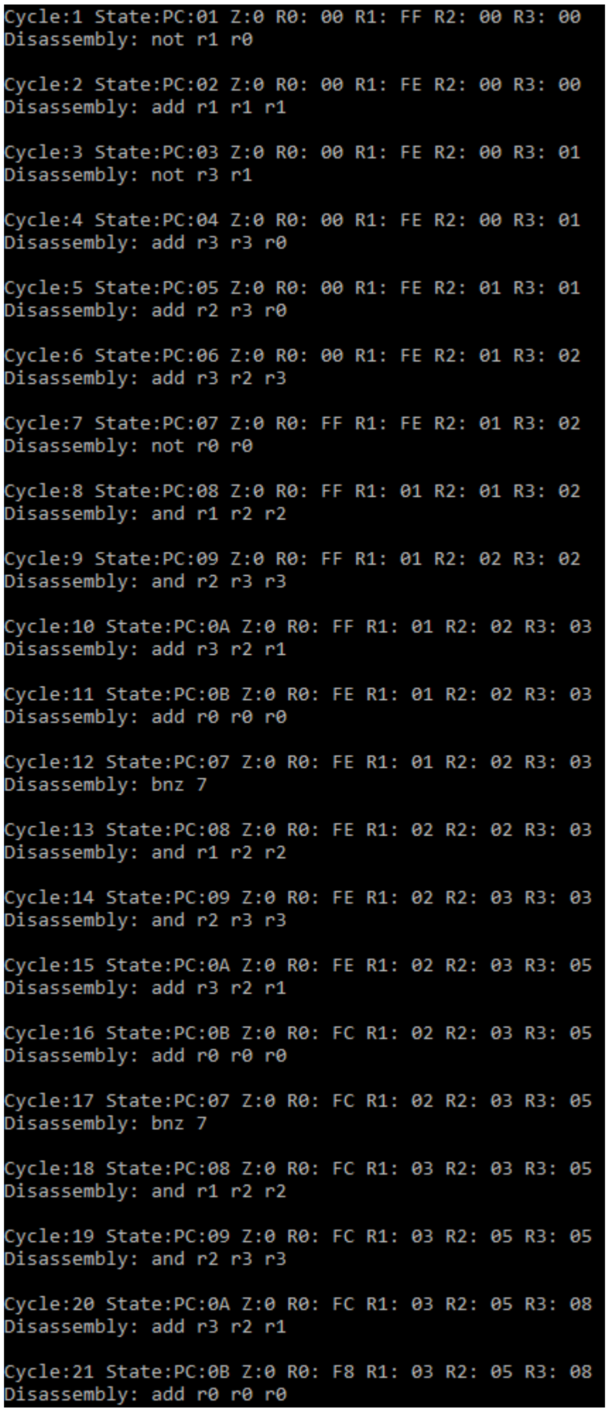


```
Cycle:21 State:PC:09 Z:0 R0: FF R1: 05 R2: 08 R3: 08
Disassembly: and r2 r3 r3

Cycle:22 State:PC:0A Z:0 R0: FF R1: 05 R2: 08 R3: 0D
Disassembly: add r3 r2 r1

Cycle:23 State:PC:07 Z:0 R0: FF R1: 05 R2: 08 R3: 0D
Disassembly: bnz 7

Cycle:24 State:PC:08 Z:0 R0: FF R1: 08 R2: 08 R3: 0D
Disassembly: and r1 r2 r2

Cycle:25 State:PC:09 Z:0 R0: FF R1: 08 R2: 0D R3: 0D
Disassembly: and r2 r3 r3

Cycle:26 State:PC:0A Z:0 R0: FF R1: 08 R2: 0D R3: 15
Disassembly: add r3 r2 r1

Cycle:27 State:PC:07 Z:0 R0: FF R1: 08 R2: 0D R3: 15
Disassembly: bnz 7

Cycle:28 State:PC:08 Z:0 R0: FF R1: 0D R2: 0D R3: 15
Disassembly: and r1 r2 r2

Cycle:29 State:PC:09 Z:0 R0: FF R1: 0D R2: 15 R3: 15
Disassembly: and r2 r3 r3

Cycle:30 State:PC:0A Z:0 R0: FF R1: 0D R2: 15 R3: 22
Disassembly: add r3 r2 r1

Cycle:31 State:PC:07 Z:0 R0: FF R1: 0D R2: 15 R3: 22
Disassembly: bnz 7

Cycle:32 State:PC:08 Z:0 R0: FF R1: 15 R2: 15 R3: 22
Disassembly: and r1 r2 r2

Cycle:33 State:PC:09 Z:0 R0: FF R1: 15 R2: 22 R3: 22
Disassembly: and r2 r3 r3

Cycle:34 State:PC:0A Z:0 R0: FF R1: 15 R2: 22 R3: 37
Disassembly: add r3 r2 r1

Cycle:35 State:PC:07 Z:0 R0: FF R1: 15 R2: 22 R3: 37
Disassembly: bnz 7

Cycle:36 State:PC:08 Z:0 R0: FF R1: 22 R2: 22 R3: 37
Disassembly: and r1 r2 r2

Cycle:37 State:PC:09 Z:0 R0: FF R1: 22 R2: 37 R3: 37
Disassembly: and r2 r3 r3

Cycle:38 State:PC:0A Z:0 R0: FF R1: 22 R2: 37 R3: 59
Disassembly: add r3 r2 r1

Cycle:39 State:PC:07 Z:0 R0: FF R1: 22 R2: 37 R3: 59
Disassembly: bnz 7

Cycle:40 State:PC:08 Z:0 R0: FF R1: 37 R2: 37 R3: 59
Disassembly: and r1 r2 r2

Cycle:41 State:PC:09 Z:0 R0: FF R1: 37 R2: 59 R3: 59
Disassembly: and r2 r3 r3
```

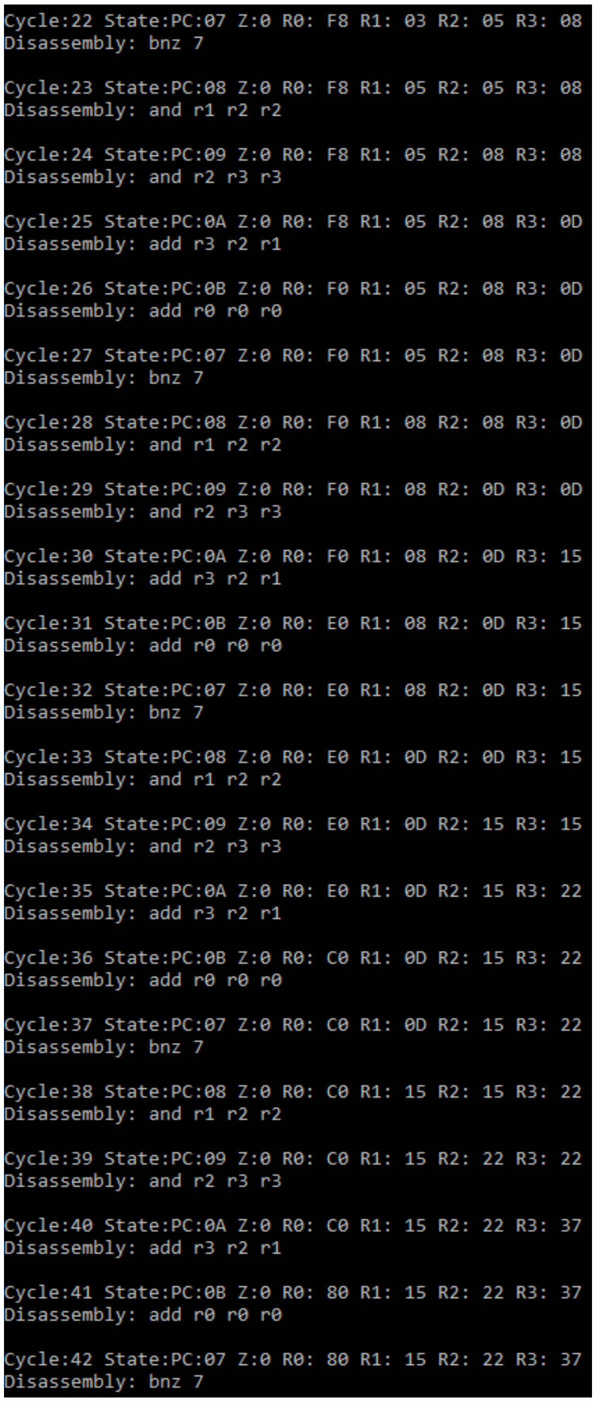


```
Cycle:42 State:PC:0A Z:0 R0: FF R1: 37 R2: 59 R3: 90
Disassembly: add r3 r2 r1

Cycle:43 State:PC:07 Z:0 R0: FF R1: 37 R2: 59 R3: 90
Disassembly: bnz 7

Cycle:44 State:PC:08 Z:0 R0: FF R1: 59 R2: 59 R3: 90
Disassembly: and r1 r2 r2

Cycle:45 State:PC:09 Z:0 R0: FF R1: 59 R2: 90 R3: 90
Disassembly: and r2 r3 r3

Cycle:46 State:PC:0A Z:0 R0: FF R1: 59 R2: 90 R3: E9
Disassembly: add r3 r2 r1

Cycle:47 State:PC:07 Z:0 R0: FF R1: 59 R2: 90 R3: E9
Disassembly: bnz 7

Cycle:48 State:PC:08 Z:0 R0: FF R1: 90 R2: 90 R3: E9
Disassembly: and r1 r2 r2

Cycle:49 State:PC:09 Z:0 R0: FF R1: 90 R2: E9 R3: E9
Disassembly: and r2 r3 r3

Cycle:50 State:PC:0A Z:0 R0: FF R1: 90 R2: E9 R3: 79
Disassembly: add r3 r2 r1
```

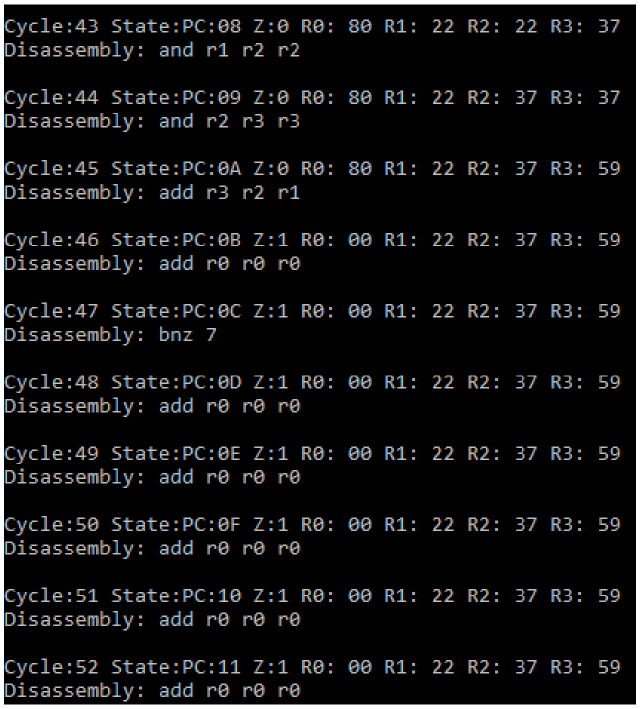
fibo2:



```
Cycle:1 State:PC:01 Z:0 R0: 00 R1: FF R2: 00 R3: 00
Disassembly: not r1 r0

Cycle:2 State:PC:02 Z:0 R0: 00 R1: FE R2: 00 R3: 00
Disassembly: add r1 r1 r1

Cycle:3 State:PC:03 Z:0 R0: 00 R1: FE R2: 00 R3: 01
Disassembly: not r3 r1

Cycle:4 State:PC:04 Z:0 R0: 00 R1: FE R2: 00 R3: 01
Disassembly: add r3 r3 r0

Cycle:5 State:PC:05 Z:0 R0: 00 R1: FE R2: 01 R3: 01
Disassembly: add r2 r3 r0

Cycle:6 State:PC:06 Z:0 R0: 00 R1: FE R2: 01 R3: 02
Disassembly: add r3 r2 r3

Cycle:7 State:PC:07 Z:0 R0: FF R1: FE R2: 01 R3: 02
Disassembly: not r0 r0

Cycle:8 State:PC:08 Z:0 R0: FF R1: 01 R2: 01 R3: 02
Disassembly: and r1 r2 r2

Cycle:9 State:PC:09 Z:0 R0: FF R1: 01 R2: 02 R3: 02
Disassembly: and r2 r3 r3

Cycle:10 State:PC:0A Z:0 R0: FF R1: 01 R2: 02 R3: 03
Disassembly: add r3 r2 r1

Cycle:11 State:PC:0B Z:0 R0: FE R1: 01 R2: 02 R3: 03
Disassembly: add r0 r0 r0

Cycle:12 State:PC:07 Z:0 R0: FE R1: 01 R2: 02 R3: 03
Disassembly: bnz 7

Cycle:13 State:PC:08 Z:0 R0: FE R1: 02 R2: 02 R3: 03
Disassembly: and r1 r2 r2

Cycle:14 State:PC:09 Z:0 R0: FE R1: 02 R2: 03 R3: 03
Disassembly: and r2 r3 r3

Cycle:15 State:PC:0A Z:0 R0: FE R1: 02 R2: 03 R3: 05
Disassembly: add r3 r2 r1

Cycle:16 State:PC:0B Z:0 R0: FC R1: 02 R2: 03 R3: 05
Disassembly: add r0 r0 r0

Cycle:17 State:PC:07 Z:0 R0: FC R1: 02 R2: 03 R3: 05
Disassembly: bnz 7

Cycle:18 State:PC:08 Z:0 R0: FC R1: 03 R2: 03 R3: 05
Disassembly: and r1 r2 r2

Cycle:19 State:PC:09 Z:0 R0: FC R1: 03 R2: 05 R3: 05
Disassembly: and r2 r3 r3

Cycle:20 State:PC:0A Z:0 R0: FC R1: 03 R2: 05 R3: 08
Disassembly: add r3 r2 r1

Cycle:21 State:PC:0B Z:0 R0: F8 R1: 03 R2: 05 R3: 08
Disassembly: add r0 r0 r0
```



```
Cycle:22 State:PC:07 Z:0 R0: F8 R1: 03 R2: 05 R3: 08
Disassembly: bnz 7

Cycle:23 State:PC:08 Z:0 R0: F8 R1: 05 R2: 05 R3: 08
Disassembly: and r1 r2 r2

Cycle:24 State:PC:09 Z:0 R0: F8 R1: 05 R2: 08 R3: 08
Disassembly: and r2 r3 r3

Cycle:25 State:PC:0A Z:0 R0: F8 R1: 05 R2: 08 R3: 0D
Disassembly: add r3 r2 r1

Cycle:26 State:PC:0B Z:0 R0: F0 R1: 05 R2: 08 R3: 0D
Disassembly: add r0 r0 r0

Cycle:27 State:PC:07 Z:0 R0: F0 R1: 05 R2: 08 R3: 0D
Disassembly: bnz 7

Cycle:28 State:PC:08 Z:0 R0: F0 R1: 08 R2: 08 R3: 0D
Disassembly: and r1 r2 r2

Cycle:29 State:PC:09 Z:0 R0: F0 R1: 08 R2: 0D R3: 0D
Disassembly: and r2 r3 r3

Cycle:30 State:PC:0A Z:0 R0: F0 R1: 08 R2: 0D R3: 15
Disassembly: add r3 r2 r1

Cycle:31 State:PC:0B Z:0 R0: E0 R1: 08 R2: 0D R3: 15
Disassembly: add r0 r0 r0

Cycle:32 State:PC:07 Z:0 R0: E0 R1: 08 R2: 0D R3: 15
Disassembly: bnz 7

Cycle:33 State:PC:08 Z:0 R0: E0 R1: 0D R2: 0D R3: 15
Disassembly: and r1 r2 r2

Cycle:34 State:PC:09 Z:0 R0: E0 R1: 0D R2: 15 R3: 15
Disassembly: and r2 r3 r3

Cycle:35 State:PC:0A Z:0 R0: E0 R1: 0D R2: 15 R3: 22
Disassembly: add r3 r2 r1

Cycle:36 State:PC:0B Z:0 R0: C0 R1: 0D R2: 15 R3: 22
Disassembly: add r0 r0 r0

Cycle:37 State:PC:07 Z:0 R0: C0 R1: 0D R2: 15 R3: 22
Disassembly: bnz 7

Cycle:38 State:PC:08 Z:0 R0: C0 R1: 15 R2: 15 R3: 22
Disassembly: and r1 r2 r2

Cycle:39 State:PC:09 Z:0 R0: C0 R1: 15 R2: 22 R3: 22
Disassembly: and r2 r3 r3

Cycle:40 State:PC:0A Z:0 R0: C0 R1: 15 R2: 22 R3: 37
Disassembly: add r3 r2 r1

Cycle:41 State:PC:0B Z:0 R0: 80 R1: 15 R2: 22 R3: 37
Disassembly: add r0 r0 r0

Cycle:42 State:PC:07 Z:0 R0: 80 R1: 15 R2: 22 R3: 37
Disassembly: bnz 7
```



```
Cycle:43 State:PC:08 Z:0 R0: 80 R1: 22 R2: 22 R3: 37
Disassembly: and r1 r2 r2

Cycle:44 State:PC:09 Z:0 R0: 80 R1: 22 R2: 37 R3: 37
Disassembly: and r2 r3 r3

Cycle:45 State:PC:0A Z:0 R0: 80 R1: 22 R2: 37 R3: 59
Disassembly: add r3 r2 r1

Cycle:46 State:PC:0B Z:1 R0: 00 R1: 22 R2: 37 R3: 59
Disassembly: add r0 r0 r0

Cycle:47 State:PC:0C Z:1 R0: 00 R1: 22 R2: 37 R3: 59
Disassembly: bnz 7

Cycle:48 State:PC:0D Z:1 R0: 00 R1: 22 R2: 37 R3: 59
Disassembly: add r0 r0 r0

Cycle:49 State:PC:0E Z:1 R0: 00 R1: 22 R2: 37 R3: 59
Disassembly: add r0 r0 r0

Cycle:50 State:PC:0F Z:1 R0: 00 R1: 22 R2: 37 R3: 59
Disassembly: add r0 r0 r0

Cycle:51 State:PC:10 Z:1 R0: 00 R1: 22 R2: 37 R3: 59
Disassembly: add r0 r0 r0

Cycle:52 State:PC:11 Z:1 R0: 00 R1: 22 R2: 37 R3: 59
Disassembly: add r0 r0 r0
```

Designed Circuit For The Simulator:

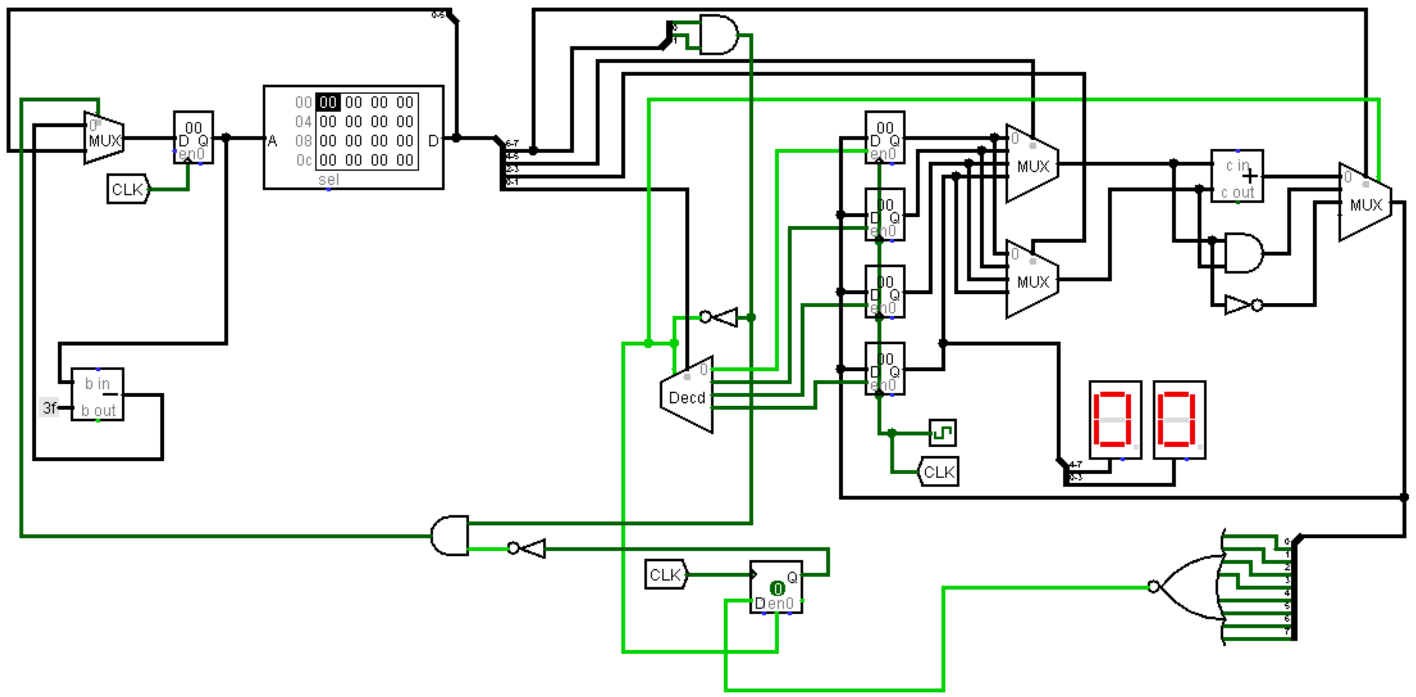
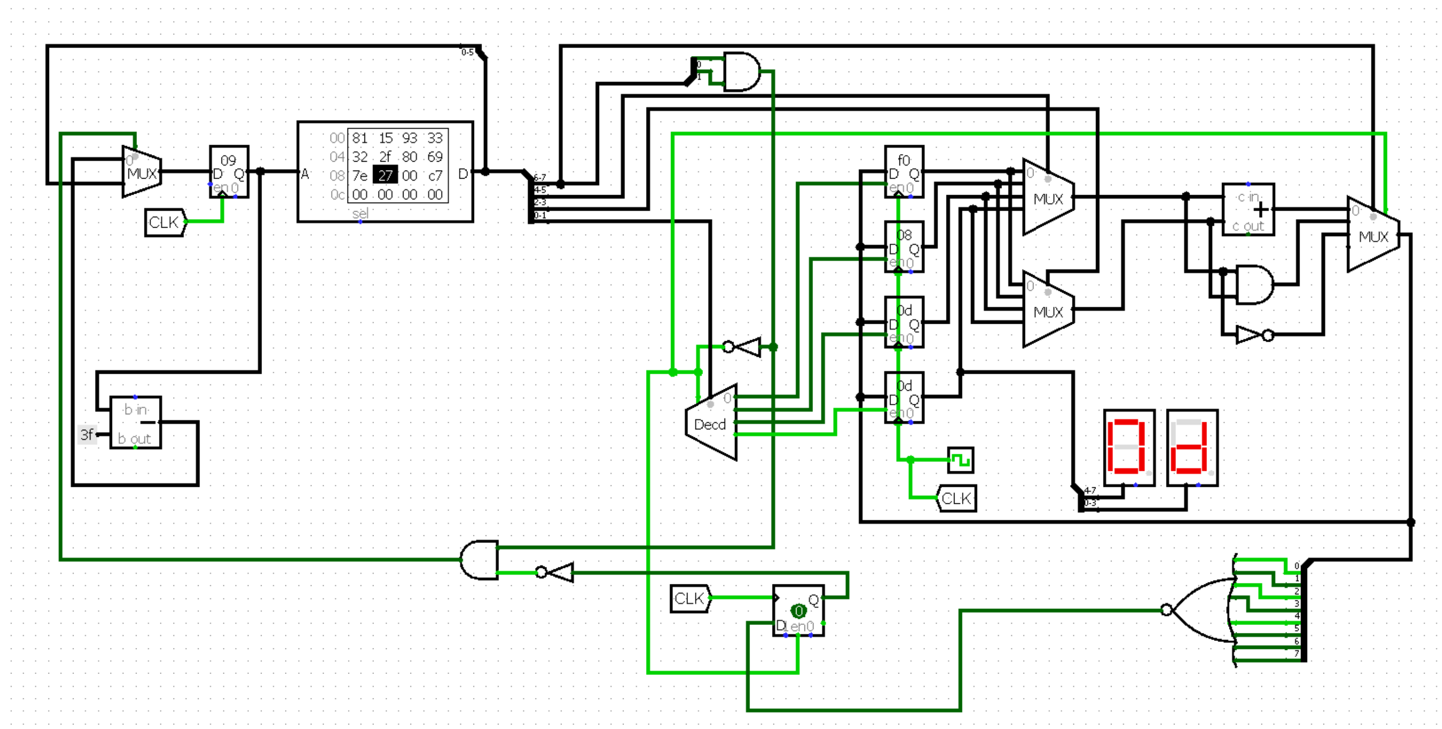


Image of The Simulator Circuit Running The fibo2.s code:



Java Code For the Assembler:

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class Fiscas {

    private static HashMap<String, Integer> symbolTable;
    private int currentAddress;

    public Fiscas() {
        symbolTable = new HashMap<>();
        currentAddress = 0;
```

```
}

public static void main(String[] args) throws Exception {
  // If user doesn't specify both files,
  // A USAGE message is displayed.
    if (args.length < 2) {
        System.out.println("USAGE:  "
                + "java Fiscas.java <source file> <object file> [-l]\r\n"
                + "          -l : print listing to standard error");
        return;
    }

    // Get input and output file name
    String inputFileName = args[0];
    String outputFileName = args[1];

    // Create an instance of this assembler
    Fiscas assembler = new Fiscas();

    // Check for the -l option, if it is specified
    // set the flag variable to true
    boolean printListing = false;
    if (args.length == 3 && args[2].equals("-l")) {
         printListing = true;
    }

   // read input from the source file
    StringBuilder sourceBuilder = new StringBuilder();
    try (BufferedReader reader = new BufferedReader(
                new FileReader(inputFileName))) {
        String line;
        while ((line = reader.readLine()) != null) {
            sourceBuilder.append(line).append('\n');
        }
    }

    String source = sourceBuilder.toString();

    // Assemble the source code
```

```
String output = assembler.assemble(source, printListing);

// write output to file
try (BufferedWriter writer = new BufferedWriter(
            new FileWriter(outputFileName))) {
     // Writes the version header to be compatible with Logisim
     writer.write("v2.0 raw\n");
     // Write the assembled output
    writer.write(output);
}

// If specified by the user, print listing to standard error
if (printListing) {
     System.err.println("*** LABEL LIST ***");
     List<Map.Entry<String, Integer>> labelList = new ArrayList<>(
                  // Get list of symbol table entries
                  assembler.symbolTable.entrySet());
     // Reverse list so it's in order of appearance
     Collections.reverse(labelList);
     // Iterate through the list printing the symbol name
     // and address
     for (Map.Entry<String, Integer> entry : labelList) {
         System.err.printf("%-8s %02X\n", entry.getKey(),
                  entry.getValue());
     }

     System.err.println("*** MACHINE PROGRAM ***");
     // Split assembled output into individual instructions
     String[] instructions = output.split("\\r?\\n");
     int address = 0;
     int labelAddress = 1;
     List<Map.Entry<String, Integer>> labelList2 = new ArrayList<>(
                  assembler.symbolTable.entrySet());
     Collections.reverse(labelList2);

     for (String instruction : instructions) {
         String machineCode = "";
         int i = 0;
         for (Map.Entry<String, Integer> entry : labelList2) {
                 String key = entry.getKey();
                 // Convert the instruction to machine code
                 machineCode = Fiscas.toMachineCode(instruction);
                 if ((machineCode.startsWith("bnz")
                             && i == labelAddress) ||
                             (machineCode.startsWith("bnz")
                                          && labelList2.size() == 1)) {
                      // Print the address, instruction, and
                      // machine code
                     System.err.printf("%02X:%-8s %s %s\n",
                             address, instruction, machineCode, key);
                     labelAddress++;
                     i++;
                 }
                 i++;
             }
             if (!machineCode.startsWith("bnz")) {
                 System.err.printf("%02X:%-8s %s\n", address,
                             instruction, machineCode);
             }
             address += 1;
         }

     }
 }

 public String assemble(String source, boolean printListing) {
     // First pass: generate symbol table
   int lineNum = 0;
   // Split the source into individual lines
     String[] lines = source.split("\\r?\\n");
     for (String line : lines) {
          lineNum++;
          // Remove any leading or trailing white space
         line = line.trim();
         if (line.isEmpty() || line.startsWith(";")) {
             continue; // ignore comments and blank lines
         }
         String[] tokens = line.split("\\s+");
```

```
        if (tokens[0].endsWith(":")) {
            // label definition
            String label = tokens[0].substring(0, tokens[0].length() - 1);
            if (symbolTable.containsKey(label)) {
                // duplicate label definition
                return "Label " + label + " on line " + lineNum +
                             " is already defined.";
            }
            symbolTable.put(label, currentAddress);
            // If there are more tokens on the same line,
            // treat them as a regular instruction
            if (tokens.length > 1) {
                currentAddress += 1;
            }
        } else {
            // instruction
            currentAddress += 1;
        }
    }

    // Second pass: generate machine code
    // Holds the output machine code
    StringBuilder output = new StringBuilder();
    currentAddress = 0;
    for (String line : lines) {
        line = line.trim();
        String[] tokens = line.split("\\s+");
        // Initialize opcode, registers, and flag
        String opcode = "";
        int rn = 0, rd = 0, rm = 0;
        int instruction = 0;
        boolean isBnz = false;
        if (line.isEmpty() || line.startsWith(";")) {
            continue; // ignore comments and blank lines
        }
        if (tokens[0].endsWith(":")) {
            if (tokens.length > 1) {
                currentAddress += 1;
                // Create a new array with all tokens except the label
```

```
String[] newTokens = new String[tokens.length - 1];
System.arraycopy(tokens, 1, newTokens, 0,
             tokens.length - 1);
// Check the opcode and parse registers
// accordingly
switch (newTokens[0]) {
case "add":
    opcode = "00";
    rn = parseRegister(newTokens[1]);
    rd = parseRegister(newTokens[2]);
    rm = parseRegister(newTokens[3]);
    break;
case "and":
    opcode = "01";
    rn = parseRegister(newTokens[2]);
    rd = parseRegister(newTokens[1]);
    rm = parseRegister(newTokens[3]);
    break;
case "not":
    opcode = "10";
    rn = parseRegister(newTokens[2]);
    rd = parseRegister(newTokens[1]);
    rm = 0; // ignore Rm for not instruction
    break;
case "bnz":
      opcode = "11";
    String secondToken = newTokens[1];
    if (symbolTable.containsKey(secondToken)) {
        int labelAddress = symbolTable.get(secondToken);
        rn = labelAddress;
    } else {
        rn = parseRegister(secondToken);
    }
    rd = 0; // ignore Rd for bnz instruction
    rm = 0; // ignore Rm for bnz instruction
    isBnz = true;
    break;
default:
      // check if it's a label
```

```
                if (symbolTable.containsKey(newTokens[0].substring(0,
                         newTokens[0].length() - 1))) {
                    // label definition (already processed)
                    continue;
                } else {
                  // Throw an exception if the instruction
                  // is unknown
                    throw new RuntimeException(
                                "Unknown instruction: " +
                newTokens[0]);
                }
            }
            // Generate the instruction based on the opcode
            // and operands
            if (isBnz) instruction = Integer.parseInt(opcode +
                         toBinary(rn, 6), 2);
            else instruction = Integer.parseInt(opcode +
                         toBinary(rn, 2) +
                         toBinary(rm, 2) + toBinary(rd, 2), 2);
            // Append to the output string
            output.append(toHex(instruction));
            output.append("\n");
            currentAddress += 1;
        }
    } else if (symbolTable.containsKey(tokens[0])) {
     // label reference
     // Get the address of the label from the symbol table
        int labelAddress = symbolTable.get(tokens[0]);
        // Concatenate the opcode, register numbers,
        // and label address
        // to create the instruction and convert to an integer
        instruction = Integer.parseInt(opcode + toBinary(rn, 2) +
                  toBinary(rd, 2) + toBinary(labelAddress, 2), 2);
        // Append to the output string
        output.append(toHex(instruction));
        output.append("\n");
        currentAddress += 1;
    } else {
        // instruction
```

```
switch (tokens[0]) {
    case "add":
        opcode = "00";
        rn = parseRegister(tokens[2]);
        rd = parseRegister(tokens[1]);
        rm = parseRegister(tokens[3]);
        break;
    case "and":
        opcode = "01";
        rn = parseRegister(tokens[2]);
        rd = parseRegister(tokens[1]);
        rm = parseRegister(tokens[3]);
        break;
    case "not":
        opcode = "10";
        rn = parseRegister(tokens[2]);
        rd = parseRegister(tokens[1]);
        rm = 0; // ignore Rm for not instruction
        break;
    case "bnz":
          opcode = "11";
        String secondToken = tokens[1];
        if (symbolTable.containsKey(secondToken)) {
            int labelAddress = symbolTable.get(secondToken);
            rn = labelAddress;
        } else {
            rn = parseRegister(secondToken);
        }
        rd = 0; // ignore Rd for bnz instruction
        rm = 0; // ignore Rm for bnz instruction
        isBnz = true;
        break;
    default:
          // check if it's a label
        if (symbolTable.containsKey(tokens[0].substring(0,
                 tokens[0].length() - 1))) {
            // label definition (already processed)
            continue;
        } else {
```

```java
                        throw new RuntimeException(""
                                   + "Unknown instruction: " + tokens[0]);
                    }
            }
            if (isBnz) instruction = Integer.parseInt(opcode +
                      toBinary(rn, 6), 2);
            else instruction = Integer.parseInt(opcode +
                      toBinary(rn, 2) + toBinary(rm, 2) +
                      toBinary(rd, 2), 2);
            // Append to the output string
            output.append(toHex(instruction));
            output.append("\n");
            currentAddress += 1;
        }
    }

    return output.toString().trim();
}

// This method takes a register string in the form of "rX",
// where X is the register number,
// and returns the integer value of the register number
private int parseRegister(String register) {
    int index = register.indexOf('r');
    if (index != 0 || register.length() != 2) {
        throw new RuntimeException("Invalid register: " + register);
    }
    return Integer.parseInt(register.substring(1));
}

// This method takes an integer value and the desired
// number of digits and returns the binary string
// representation of the integer value
// with the specified number of digits
private String toBinary(int value, int digits) {
    String binaryString = Integer.toBinaryString(value);
    int numZeroes = digits - binaryString.length();
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < numZeroes; i++) {
            sb.append('0');
        }
        sb.append(binaryString);
        return sb.toString();
    }

    // This method takes an integer value and returns the
    // hex string representation of the integer value
    private String toHex(int value) {
        return String.format("%02X", value);
    }

    // This method takes a hex string representation
    // of a machine code instruction
    // and converts it to the corresponding
    // assembly language instruction.
    public static String toMachineCode(String instruction) {
        int inst = Integer.parseInt(instruction, 16);
        String binary = String.format("%8s",
                    Integer.toBinaryString(inst)).replace(' ', '0');
        int opcode = Integer.parseInt(binary.substring(0, 2), 2);
        int rd = Integer.parseInt(binary.substring(6, 8), 2);
        int rn = Integer.parseInt(binary.substring(2, 4), 2);
        int rm = Integer.parseInt(binary.substring(4, 6), 2);
        String[] registers = { "r0", "r1", "r2", "r3" };
        String rdReg = registers[rd];
        String rnReg = registers[rn];
        String rmReg = registers[rm];
        switch (opcode) {
            case 0:
                return String.format("add %s %s %s", rdReg, rnReg, rmReg);
            case 1:
                return String.format("and %s %s %s", rdReg, rnReg, rmReg);
            case 2:
                return String.format("not %s %s", rdReg, rnReg);
            case 3:
                return "bnz ";
            default:
                return "";
        }
    }
}
```

Java Code For the Simulator:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class Fiscsim {

    // Instance variables for CPU
    private int[] registers;
    private byte[] memory;
    private boolean zeroFlag;
    private int pc;       // Program Counter

    // Constructor to initialize CPU state
    public Fiscsim() {
        // Initialize registers
      // creates an array of 4 integers to hold register values
        this.registers = new int[4];
        for (int i = 0; i < this.registers.length; i++) {
            this.registers[i] = 0;            // initializes each register to 0
        }

        // Initialize memory
        // creates a byte array to hold memory values
        this.memory = new byte[256];

        this.zeroFlag = false;         // Initialize flags
        this.pc = 0;                         // Initialize program counter
    }

    // Main method to run simulator
    public static void main(String[] args) {
      String fileName = null;
      // boolean variable to indicate if disassembly should be printed
```

```
boolean disassemble = false;
// number of cycles to run the simulator for. Default is 20
int cycles = 20;

// Prints a message on how to use this
// program and exits if used incorrectly
if (args.length < 1 || args.length > 3) {
      System.out.println("USAGE: "
                   + "java Fiscsim <object file> [<cycles>] [-d]\n"
                   + "-d : print disassembly listing with each cycle\n"
                   + "if cycles are unspecified the "
                   + "CPU will run for 20 cycles");
     System.exit(1);
 }

fileName = args[0];       // sets filename to the first argument

// If the number of cycles is specified, try to
// parse it as an integer.
// If it cannot be parsed as an integer, an error
// message is printed and program terminates.
if (args.length >= 2) {
     try {
         cycles = Integer.parseInt(args[1]);
     } catch (NumberFormatException e) {
         System.out.println("Invalid number of cycles specified.");
         System.exit(1);
     }
 }

// If the disassembly option is specified, set the flag.
// If it's an invalid option, print error message and exit
 if (args.length >= 3) {
     if (args[2].equals("-d")) {
         disassemble = true;
     } else {
         System.out.println("Invalid option specified.");
         System.exit(1);
     }
```

```
        }

        // create an instance of the simulator
        Fiscsim simulator = new Fiscsim();

        try {
             // load the instructions from the file
            simulator.loadInstructions(fileName);
        } catch (IOException e) {
            System.err.println("Error: " + e.getMessage());
            System.exit(1);
        }

        if (disassemble) {
             // run the disassembler if the flag is true
            simulator.runDisassembler(cycles);
        } else {
             // run the standard simulator if the flag is false
            simulator.run(cycles);
        }
    }

    // Method to load instructions from file into memory
    private void loadInstructions(String filename) throws IOException {
        BufferedReader reader = new BufferedReader(new FileReader(filename));
        String line = reader.readLine();

        // Verify that the first line matches the expected format
        if (!line.equals("v2.0 raw")) {
            throw new IOException("Invalid file format");
        }

        // Loop through each line of the file and store them in memory.
        int address = 0;
        while ((line = reader.readLine()) != null) {
            String[] hexValues = line.split(" ");
            for (String hex : hexValues) {
                memory[address++] = (byte) Integer.parseInt(hex, 16);
            }
        }

        reader.close();
    }

    // Method to run the simulator
    private void run(int cycles) {
       for (int i = 1; i <= cycles; i++) {
             // fetch the instruction from memory
            byte instruction = memory[pc++];
            // call the executeInstruction() method on the instruction
            executeInstruction(instruction);

            // Print out the current state of the simulator
            System.out.printf("Cycle:%d State:PC:%02X Z:%d "
                    + "R0: %02X R1: %02X R2: %02X R3: %02X\n",
                    i, pc, (zeroFlag ? 1 : 0), registers[0],
                    registers[1], registers[2], registers[3]);
        }
    }

    // Method to run the disassembler
    private void runDisassembler(int cycles) {
        for (int i = 1; i <= cycles; i++) {
             // fetch the instruction from memory
            byte instruction = memory[pc++];
            // call the executeInstruction() method on the instruction
            executeInstruction(instruction);

            // Disassemble the instruction and print it out along with
            // the current state of the simulator
            System.out.printf("Cycle:%d State:PC:%02X Z:%d R0: "
                    + "%02X R1: %02X R2: %02X R3: %02X\nDisassembly: "
                    + "%s\n\n", i,
                    pc, zeroFlag ? 1 : 0, registers[0], registers[1],
                                 registers[2],
                                 registers[3],
                                 disassembleInstruction(instruction));
        }
    }

    // Method to execute a single instruction
    private void executeInstruction(byte instruction) {
       // Extract the fields from the instruction using bit manipulation
       int opcode = (instruction >> 6) & 0x3;
       int rn = (instruction >> 4) & 0x3;
       int rm = (instruction >> 2) & 0x3;
       int rd = instruction & 0x3;
        int targetAddress = instruction & 0b00111111;

        // Decode and execute the instruction based on the opcode
        switch (opcode) {
            case 0: // ADD
                add(rd, rn, rm);
                break;
            case 1: // AND
                and(rd, rn, rm);
                break;
            case 2: // NOT
                not(rd, rn);
                break;
            case 3: // BNZ
                bnz(targetAddress);
                break;
        }
    }

    // Method to disassemble an instruction into its original assembler code
    private String disassembleInstruction(byte instruction) {
       // Extract the fields from the instruction using bit manipulation
        int opcode = (instruction >> 6) & 0x3;
        int rn = (instruction >> 4) & 0x3;
        int rm = (instruction >> 2) & 0x3;
        int rd = instruction & 0x3;
        int targetAddress = instruction & 0b00111111;

        // Determine the mnemonic for the instruction
        String mnemonic = "";
        switch (opcode) {
            case 0: // ADD
                mnemonic = "add";
                break;
            case 1: // AND
                mnemonic = "and";
                break;
            case 2: // NOT
                mnemonic = "not";
                break;
            case 3: // BNZ
                mnemonic = "bnz";
                break;
        }

        // Determine the operands for the instruction
        String operands = "";
        switch (opcode) {
            case 0: // ADD
            case 1: // AND
                operands = String.format("r%d r%d r%d", rd, rn, rm);
                break;
            case 2: // NOT
                operands = String.format("r%d r%d", rd, rn);
                break;
            case 3: // BNZ
                operands = String.format("%x", targetAddress);
                break;
        }

        // return the original assembler code as a string
        return String.format("%s %s", mnemonic, operands);
    }

    // Helper methods for specific instructions
    private void add(int destReg, int srcReg1, int srcReg2) {
       // add the values of the two source registers,
       // masking the result to 8 bits
       int result = registers[srcReg1] + registers[srcReg2] & 0xFF;
        setZeroFlag(result);     // update the zero flag based on the result
        // store the result in the destination register
        registers[destReg] = result;
    }

    private void and(int destReg, int srcReg1, int srcReg2) {
       // compute the bitwise AND of the values of the two source registers,
       // masking the result to 8 bits
       int result = registers[srcReg1] & registers[srcReg2] & 0xFF;
       // update the zero flag based on the result
        setZeroFlag(result);
        // store the result in the destination register
        registers[destReg] = result;
    }

    private void not(int destReg, int srcReg) {
       // compute the bitwise NOT of the value in the source register,
       // masking the result to 8 bits
       int result = ~registers[srcReg] & 0xFF;
       // update the zero flag based on the result
        setZeroFlag(result);
        // store the result in the destination register
        registers[destReg] = result;
    }

    private void bnz(int targetAddress) {
       // Check if the zeroFlag is false
       if (!zeroFlag) {
             // set the program counter to the target address
            pc = targetAddress;
         // check to prevent going out of bounds
            if (pc >= memory.length) {
                pc = memory.length - 1;
            }
        }
    }

    // Helper methods for updating flags
```

```
    private void setZeroFlag(int value) {
        zeroFlag = (value == 0);
    }
}
```