Lab 1: FISCAS and FISCSIM

Billy Ouattara
StudentID: *****3707
ECS 154A

Theory of operation

In the lab, the programmed assembler takes as a command line argument a source file, an object file, and an optional argument used to output listings. The source file is the input file containing assembly, the object file is the name of the output hex file containing machine code. In my fiscas program, the program reads the input file in a sequence of first and second passes. The first pass parses each line from the input file and stores the assembly code in an instruction object. The instruction object represents each assembly line. While on the first pass, the program keeps track of defined labels and stores their addresses in a hashtable. Output error is emitted when there is an invalid label or the file is larger than the system memory. In the second pass, the program converts each assembly instruction into an 8-bit binary representation using the *AssemblyInstruction* object. The *writeData* method in the *Assembler* object converts each 8-bit binary instruction to hexadecimal, writes it into the object file, and adds it into a vector of string called *hexCode* for reference, for the output listing.

To test the assembler, I first tested the command line input to make sure that any input error is displayed to the user. I also modified the sample input file in the lab to check for instruction errors. For instance, I tested the input file with upper and lower cases, out-of-bound index for the registers, and undefined and duplicate labels. We output error messages for different types of errors in the code. When done with testing the errors, I tested the program with the input sample of the lab and got the desired output file.

For the simulator, the program reads as an argument, an object file which is the file containing the hex codes generated by the assembler, an integer number representing the number of cycles, and an optional argument used to display disassembly. The simulator compiles the object file by reading each hex line and storing it in an *InstructionMemory* object. This object has for attribute an array of unsigned 8-bit integers representing the binary code of the hex line. That instruction memory is then used to initialize a *DisassemblerInstruction* object. The disassembler instruction object converts the instruction memory into assembly code. After translating each line of machine code into their respective binary representation and disassembly code, we make use of the simulator's *computeInstruction* method to compute bitwise operation for the instructions. After performing the instruction's operation, the CPU states are displayed to the user. The disassembly is displayed according to the user's input for the optional command line argument.

When testing the simulator, I made sure to check for the presence of the header file "v2.0 raw". Error output is displayed when the header is different or not present. I used the sample input hex file in the lab to make sure that the simulator generate the right output for the cycles and disassembly.

Discussion

fibo1.s

```
start:   not r0 r1      ; negate the value of r1 to r0
         and r0 r0 r1   ; and r1 and r0 result in 0 for r0
         not r1 r0      ; negate the value in r0 to r1
         add r1 r1 r1   ; r1 contains 254
         not r1 r1      ; negating 254 results in 1
         and r3 r0 r0   ; r3 = 0
         and r3 r1 r1   ; r3 = 1
loop:    add r3 r0 r1   ; r3 = r0 + r
         and r0 r1 r1   ; r0 = r1
         and r1 r3 r3   ; r1 = r3
         bnz loop       ; never stop because the z-flag is never set
```

Generated list output:

```
*** LABEL LIST***
loop    07
start   00
*** MACHINE PROGRAM ***
00:90   not r0 r1
01:44   and r0 r0 r1
02:81   not r1 r0
03:15   add r1 r1 r1
04:91   not r1 r1
05:43   and r3 r0 r0
06:57   and r3 r1 r1
07:07   add r3 r0 r1
08:54   and r0 r1 r1
09:7D   and r1 r3 r3
10:C7   bnz loop
```

1.What is the last valid Fibonacci number output by your code?
The last valid Fibonacci number output by the code is hexadecimal E9 which is 233 in decimal representation.

2.How many cycles does it take to compute this number?
It takes 52 cycles to compute this number.

3.What value is output for the first invalid Fibonacci number?
The first invalid Fibonacci number displayed by the code is hexadecimal 79 which is 121 in decimal value.

4.Explain why this number is invalid, that is, what goes wrong with the arithmetic computation?
This value is invalid because, in terms of decimal values, 144(previous valid Fibonacci) is added to 233(last valid Fibonacci) in r3. However, the addition results in an overflow. So when doing the addition, the value of r3 goes to zero when reaching 256, and the remaining value is added back to it. Thus resulting in a value of 121 which led to a hexadecimal output of 79.

fibo2.s

```
start:   not r0 r1      ; negate the value of r1 to r0
         and r0 r0 r1   ; and r1 and r0 result in 0 for r0
         not r1 r0      ; negate the value in r0 to r1
         add r1 r1 r1   ; r1 contains 254
         not r1 r1      ; negating 254 results in 1
         add r2 r1 r1   ; r2 = r1 + r1 = 1 + 1 = 2
         add r2 r2 r1   ; r2 = r2 + r1 = 2 + 1 = 3
         add r2 r2 r2   ; r2 = r2 + r2 = 3 + 3 = 6
         add r2 r2 r2   ; r2 = r2 + r2 = 6 + 6 = 12
         add r2 r2 r2   ; r2 = r2 + r2 = 12 + 12 = 24
         add r2 r2 r1   ; r2 = r2 + r1 = 24 + 1 = 25
         and r3 r0 r0   ; r3 = 0
         and r3 r1 r1   ; r3 = 1
loop:    add r3 r0 r1   ; r3 = r0 + r1
         and r0 r1 r1   ; r0 = r1
         and r1 r3 r3   ; r1 = r3
         add r2 r2 r3   ; r2 = r2 + r3
         bnz loop       ; repeat until r2 = 256 = 0
stop:    and r3 r3 r3   ; set the flags
         bnz stop       ; effectively stops
```

Generated list output:

```
*** LABEL LIST***
stop    18
loop    13
start   00
*** MACHINE PROGRAM ***
00:90   not r0 r1
01:44   and r0 r0 r1
02:81   not r1 r0
03:15   add r1 r1 r1
04:91   not r1 r1
05:16   add r2 r1 r1
06:26   add r2 r2 r1
```

```
07:2A   add r2 r2 r2
08:2A   add r2 r2 r2
09:2A   add r2 r2 r2
10:26   add r2 r2 r1
11:43   and r3 r0 r0
12:57   and r3 r1 r1
13:07   add r3 r0 r1
14:54   and r0 r1 r1
15:7D   and r1 r3 r3
16:2E   add r2 r2 r3
17:CD   bnz loop
18:7F   and r3 r3 r3
19:D2   bnz stop
```

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

- To start, I tried to manually add 12 to r2 register and used that value to decrement it so that when we reach 0, the z-flag is set to exit the loop. However, doing that would have required an additional resistor which would hold the value of -1 to decrement r2.
- To minimize my code, I manually multiplied 3 by 8 into r2 to obtain 24. I incremented the value by the value of r1 which was 1 to get 25.
- In the loop, I add the value in r3 to r2. When the value of r3 gets to 89, r2 = r2 + 89 = 256 which set the z-flag and exit the loop.

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

One limitation that made this problem challenging was that we had to manually add 24 into the register so that when adding r2 + r3 into r2 we reach 256 at some point to exit the loop.

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

- *Explain how this instruction would help.*
  I think having a "mov" instruction that could move a constant value into a register would be beneficial. However, that constant value would have to be between the range 0 and 3.

- *Outline how you think that this instruction would fit into the current design*
  - This instruction would allow me to move the constant value 0 into r0 and 1 into r1 instead of using "add", "and", or "not" operations to obtain those values.
  - It also would allow me to move the constant value 3 into the register r2.
  - This would have reduced the number of lines of code by 4.

- *How many arguments does your instruction take? (Note, you must be able to fit it into the current design, think carefully about this)*
  This instruction would take two arguments, a destination register, and a constant value.

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

An example of line assembly for the instruction would be: mov r2 3
This line moves the decimal value 3 into the register r2.

- *What is the opcode of your new instruction?*
  Since we already have all the combinations for the 2-bit representation of opcodes, we could use the unused 2-bits of the not opcode(10) to store the opcode of the new instruction. The opcode of the new instruction would be "11". When matching the "not" opcode, we check if the unused 2-bits is "11". If that is the case then we have to perform the "mov" operation.

- *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 binary machine code for the instruction is $(10111110)_2$. The hexadecimal representation of that binary code is $(BE)_{16}$

Conclusion

To summarize, in the lab, we built an assembler which we used to convert the assembly code we wrote for fibo1 and fibo2 to machine code. We also built a simulator that we used to check the states of different cycles for an assembly program. We used the simulator to compile our machine code for fibo1 and fibo2. The simulator allowed us to verify the behavior of our assembly code. We were able to check if the operation was made correctly by verifying the states of the registers as well as the program counter. Fibo1 and fibo2 also helped us have a better understanding of overflow and its importance. The concept of overflow was useful when writing the assembly for fibo2 as it helped us exit the loop.

In terms of operations, this lab helped us have a clear idea of digital gates and how they are used in the assembler. We could understand how an inverter can negate a value and how the AND gate was used for the "and" operations. Using those binary operations we were able to generate 0s and 1's to assign to registers and use those values to generate different decimal values. For the "add" operation, the system uses a series of full and half adders to perform carry operations. An overflow in addition results in an additional bit to the size of the original bit. In our case, the overflow for the unsigned eight-bit results in a value of 0 which sets the z-flag. For the "bnz" operation, the program counter uses the six least significant bits of the operation to branch into the targeted address. The program counter also uses the two most significant bits of each instruction to determine the type of operation to be performed. Doing those operations allowed us to have a clear understanding of the structure of an assembler as well as its digital architechture.

Appendix A: Fibo1 simulator disassembly

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

Cycle:2 States:PC:02 Z:1 R0:00 R1:00 R2:00 R3:0
Disassembly: and r0 r0 r1

Cycle:3 States:PC:03 Z:0 R0:00 R1:FF R2:00 R3:0
Disassembly: not r1 r0

Cycle:4 States:PC:04 Z:0 R0:00 R1:FE R2:00 R3:0
Disassembly: add r1 r1 r1

Cycle:5 States:PC:05 Z:0 R0:00 R1:01 R2:00 R3:0
Disassembly: not r1 r1

Cycle:6 States:PC:06 Z:1 R0:00 R1:01 R2:00 R3:0
Disassembly: and r3 r0 r0

Cycle:7 States:PC:07 Z:0 R0:00 R1:01 R2:00 R3:1
Disassembly: and r3 r1 r1

Cycle:8 States:PC:08 Z:0 R0:00 R1:01 R2:00 R3:1
Disassembly: add r3 r0 r1

Cycle:9 States:PC:09 Z:0 R0:01 R1:01 R2:00 R3:1
Disassembly: and r0 r1 r1

Cycle:10 States:PC:10 Z:0 R0:01 R1:01 R2:00 R3:1
Disassembly: and r1 r3 r3

Cycle:11 States:PC:07 Z:0 R0:01 R1:01 R2:00 R3:1
Disassembly: bnz 7

Cycle:12 States:PC:08 Z:0 R0:01 R1:01 R2:00 R3:2
Disassembly: add r3 r0 r1

Cycle:13 States:PC:09 Z:0 R0:01 R1:01 R2:00 R3:2
Disassembly: and r0 r1 r1

Cycle:14 States:PC:10 Z:0 R0:01 R1:02 R2:00 R3:2
Disassembly: and r1 r3 r3

Cycle:15 States:PC:07 Z:0 R0:01 R1:02 R2:00 R3:2
```

```
Disassembly: bnz 7

Cycle:16 States:PC:08 Z:0 R0:01 R1:02 R2:00 R3:3
Disassembly: add r3 r0 r1

Cycle:17 States:PC:09 Z:0 R0:02 R1:02 R2:00 R3:3
Disassembly: and r0 r1 r1

Cycle:18 States:PC:10 Z:0 R0:02 R1:03 R2:00 R3:3
Disassembly: and r1 r3 r3

Cycle:19 States:PC:07 Z:0 R0:02 R1:03 R2:00 R3:3
Disassembly: bnz 7

Cycle:20 States:PC:08 Z:0 R0:02 R1:03 R2:00 R3:5
Disassembly: add r3 r0 r1

Cycle:21 States:PC:09 Z:0 R0:03 R1:03 R2:00 R3:5
Disassembly: and r0 r1 r1

Cycle:22 States:PC:10 Z:0 R0:03 R1:05 R2:00 R3:5
Disassembly: and r1 r3 r3

Cycle:23 States:PC:07 Z:0 R0:03 R1:05 R2:00 R3:5
Disassembly: bnz 7

Cycle:24 States:PC:08 Z:0 R0:03 R1:05 R2:00 R3:8
Disassembly: add r3 r0 r1

Cycle:25 States:PC:09 Z:0 R0:05 R1:05 R2:00 R3:8
Disassembly: and r0 r1 r1

Cycle:26 States:PC:10 Z:0 R0:05 R1:08 R2:00 R3:8
Disassembly: and r1 r3 r3

Cycle:27 States:PC:07 Z:0 R0:05 R1:08 R2:00 R3:8
Disassembly: bnz 7

Cycle:28 States:PC:08 Z:0 R0:05 R1:08 R2:00 R3:d
Disassembly: add r3 r0 r1

Cycle:29 States:PC:09 Z:0 R0:08 R1:08 R2:00 R3:d
Disassembly: and r0 r1 r1

Cycle:30 States:PC:10 Z:0 R0:08 R1:13 R2:00 R3:d
Disassembly: and r1 r3 r3
```

```
Cycle:31 States:PC:07 Z:0 R0:08 R1:13 R2:00 R3:d
Disassembly: bnz 7

Cycle:32 States:PC:08 Z:0 R0:08 R1:13 R2:00 R3:15
Disassembly: add r3 r0 r1

Cycle:33 States:PC:09 Z:0 R0:13 R1:13 R2:00 R3:15
Disassembly: and r0 r1 r1

Cycle:34 States:PC:10 Z:0 R0:13 R1:21 R2:00 R3:15
Disassembly: and r1 r3 r3

Cycle:35 States:PC:07 Z:0 R0:13 R1:21 R2:00 R3:15
Disassembly: bnz 7

Cycle:36 States:PC:08 Z:0 R0:13 R1:21 R2:00 R3:22
Disassembly: add r3 r0 r1

Cycle:37 States:PC:09 Z:0 R0:21 R1:21 R2:00 R3:22
Disassembly: and r0 r1 r1

Cycle:38 States:PC:10 Z:0 R0:21 R1:34 R2:00 R3:22
Disassembly: and r1 r3 r3

Cycle:39 States:PC:07 Z:0 R0:21 R1:34 R2:00 R3:22
Disassembly: bnz 7

Cycle:40 States:PC:08 Z:0 R0:21 R1:34 R2:00 R3:37
Disassembly: add r3 r0 r1

Cycle:41 States:PC:09 Z:0 R0:34 R1:34 R2:00 R3:37
Disassembly: and r0 r1 r1

Cycle:42 States:PC:10 Z:0 R0:34 R1:55 R2:00 R3:37
Disassembly: and r1 r3 r3

Cycle:43 States:PC:07 Z:0 R0:34 R1:55 R2:00 R3:37
Disassembly: bnz 7

Cycle:44 States:PC:08 Z:0 R0:34 R1:55 R2:00 R3:59
Disassembly: add r3 r0 r1

Cycle:45 States:PC:09 Z:0 R0:55 R1:55 R2:00 R3:59
Disassembly: and r0 r1 r1
```

```
Cycle:46 States:PC:10 Z:0 R0:55 R1:89 R2:00 R3:59
Disassembly: and r1 r3 r3

Cycle:47 States:PC:07 Z:0 R0:55 R1:89 R2:00 R3:59
Disassembly: bnz 7

Cycle:48 States:PC:08 Z:0 R0:55 R1:89 R2:00 R3:90
Disassembly: add r3 r0 r1

Cycle:49 States:PC:09 Z:0 R0:89 R1:89 R2:00 R3:90
Disassembly: and r0 r1 r1

Cycle:50 States:PC:10 Z:0 R0:89 R1:144 R2:00 R3:90
Disassembly: and r1 r3 r3

Cycle:51 States:PC:07 Z:0 R0:89 R1:144 R2:00 R3:90
Disassembly: bnz 7

Cycle:52 States:PC:08 Z:0 R0:89 R1:144 R2:00 R3:e9
Disassembly: add r3 r0 r1
```

Appendix B: Fibo2 simulator disassembly

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

Cycle:2 States:PC:02 Z:1 R0:00 R1:00 R2:00 R3:0
Disassembly: and r0 r0 r1

Cycle:3 States:PC:03 Z:0 R0:00 R1:FF R2:00 R3:0
Disassembly: not r1 r0

Cycle:4 States:PC:04 Z:0 R0:00 R1:FE R2:00 R3:0
Disassembly: add r1 r1 r1

Cycle:5 States:PC:05 Z:0 R0:00 R1:01 R2:00 R3:0
Disassembly: not r1 r1

Cycle:6 States:PC:06 Z:0 R0:00 R1:01 R2:02 R3:0
Disassembly: add r2 r1 r1

Cycle:7 States:PC:07 Z:0 R0:00 R1:01 R2:03 R3:0
Disassembly: add r2 r2 r1

Cycle:8 States:PC:08 Z:0 R0:00 R1:01 R2:06 R3:0
Disassembly: add r2 r2 r2

Cycle:9 States:PC:09 Z:0 R0:00 R1:01 R2:12 R3:0
Disassembly: add r2 r2 r2

Cycle:10 States:PC:10 Z:0 R0:00 R1:01 R2:24 R3:0
Disassembly: add r2 r2 r2

Cycle:11 States:PC:11 Z:0 R0:00 R1:01 R2:25 R3:0
Disassembly: add r2 r2 r1

Cycle:12 States:PC:12 Z:1 R0:00 R1:01 R2:25 R3:0
Disassembly: and r3 r0 r0

Cycle:13 States:PC:13 Z:0 R0:00 R1:01 R2:25 R3:1
Disassembly: and r3 r1 r1

Cycle:14 States:PC:14 Z:0 R0:00 R1:01 R2:25 R3:1
Disassembly: add r3 r0 r1

Cycle:15 States:PC:15 Z:0 R0:01 R1:01 R2:25 R3:1
```

```
Disassembly: and r0 r1 r1

Cycle:16 States:PC:16 Z:0 R0:01 R1:01 R2:25 R3:1
Disassembly: and r1 r3 r3

Cycle:17 States:PC:17 Z:0 R0:01 R1:01 R2:26 R3:1
Disassembly: add r2 r2 r3

Cycle:18 States:PC:13 Z:0 R0:01 R1:01 R2:26 R3:1
Disassembly: bnz 13

Cycle:19 States:PC:14 Z:0 R0:01 R1:01 R2:26 R3:2
Disassembly: add r3 r0 r1

Cycle:20 States:PC:15 Z:0 R0:01 R1:01 R2:26 R3:2
Disassembly: and r0 r1 r1

Cycle:21 States:PC:16 Z:0 R0:01 R1:02 R2:26 R3:2
Disassembly: and r1 r3 r3

Cycle:22 States:PC:17 Z:0 R0:01 R1:02 R2:28 R3:2
Disassembly: add r2 r2 r3

Cycle:23 States:PC:13 Z:0 R0:01 R1:02 R2:28 R3:2
Disassembly: bnz 13

Cycle:24 States:PC:14 Z:0 R0:01 R1:02 R2:28 R3:3
Disassembly: add r3 r0 r1

Cycle:25 States:PC:15 Z:0 R0:02 R1:02 R2:28 R3:3
Disassembly: and r0 r1 r1

Cycle:26 States:PC:16 Z:0 R0:02 R1:03 R2:28 R3:3
Disassembly: and r1 r3 r3

Cycle:27 States:PC:17 Z:0 R0:02 R1:03 R2:31 R3:3
Disassembly: add r2 r2 r3

Cycle:28 States:PC:13 Z:0 R0:02 R1:03 R2:31 R3:3
Disassembly: bnz 13

Cycle:29 States:PC:14 Z:0 R0:02 R1:03 R2:31 R3:5
Disassembly: add r3 r0 r1

Cycle:30 States:PC:15 Z:0 R0:03 R1:03 R2:31 R3:5
Disassembly: and r0 r1 r1
```

```
Cycle:31 States:PC:16 Z:0 R0:03 R1:05 R2:31 R3:5
Disassembly: and r1 r3 r3

Cycle:32 States:PC:17 Z:0 R0:03 R1:05 R2:36 R3:5
Disassembly: add r2 r2 r3

Cycle:33 States:PC:13 Z:0 R0:03 R1:05 R2:36 R3:5
Disassembly: bnz 13

Cycle:34 States:PC:14 Z:0 R0:03 R1:05 R2:36 R3:8
Disassembly: add r3 r0 r1

Cycle:35 States:PC:15 Z:0 R0:05 R1:05 R2:36 R3:8
Disassembly: and r0 r1 r1

Cycle:36 States:PC:16 Z:0 R0:05 R1:08 R2:36 R3:8
Disassembly: and r1 r3 r3

Cycle:37 States:PC:17 Z:0 R0:05 R1:08 R2:44 R3:8
Disassembly: add r2 r2 r3

Cycle:38 States:PC:13 Z:0 R0:05 R1:08 R2:44 R3:8
Disassembly: bnz 13

Cycle:39 States:PC:14 Z:0 R0:05 R1:08 R2:44 R3:d
Disassembly: add r3 r0 r1

Cycle:40 States:PC:15 Z:0 R0:08 R1:08 R2:44 R3:d
Disassembly: and r0 r1 r1

Cycle:41 States:PC:16 Z:0 R0:08 R1:13 R2:44 R3:d
Disassembly: and r1 r3 r3

Cycle:42 States:PC:17 Z:0 R0:08 R1:13 R2:57 R3:d
Disassembly: add r2 r2 r3

Cycle:43 States:PC:13 Z:0 R0:08 R1:13 R2:57 R3:d
Disassembly: bnz 13

Cycle:44 States:PC:14 Z:0 R0:08 R1:13 R2:57 R3:15
Disassembly: add r3 r0 r1

Cycle:45 States:PC:15 Z:0 R0:13 R1:13 R2:57 R3:15
Disassembly: and r0 r1 r1
```

```
Cycle:46 States:PC:16 Z:0 R0:13 R1:21 R2:57 R3:15
Disassembly: and r1 r3 r3

Cycle:47 States:PC:17 Z:0 R0:13 R1:21 R2:78 R3:15
Disassembly: add r2 r2 r3

Cycle:48 States:PC:13 Z:0 R0:13 R1:21 R2:78 R3:15
Disassembly: bnz 13

Cycle:49 States:PC:14 Z:0 R0:13 R1:21 R2:78 R3:22
Disassembly: add r3 r0 r1

Cycle:50 States:PC:15 Z:0 R0:21 R1:21 R2:78 R3:22
Disassembly: and r0 r1 r1

Cycle:51 States:PC:16 Z:0 R0:21 R1:34 R2:78 R3:22
Disassembly: and r1 r3 r3

Cycle:52 States:PC:17 Z:0 R0:21 R1:34 R2:112 R3:22
Disassembly: add r2 r2 r3

Cycle:53 States:PC:13 Z:0 R0:21 R1:34 R2:112 R3:22
Disassembly: bnz 13

Cycle:54 States:PC:14 Z:0 R0:21 R1:34 R2:112 R3:37
Disassembly: add r3 r0 r1

Cycle:55 States:PC:15 Z:0 R0:34 R1:34 R2:112 R3:37
Disassembly: and r0 r1 r1

Cycle:56 States:PC:16 Z:0 R0:34 R1:55 R2:112 R3:37
Disassembly: and r1 r3 r3

Cycle:57 States:PC:17 Z:0 R0:34 R1:55 R2:167 R3:37
Disassembly: add r2 r2 r3

Cycle:58 States:PC:13 Z:0 R0:34 R1:55 R2:167 R3:37
Disassembly: bnz 13

Cycle:59 States:PC:14 Z:0 R0:34 R1:55 R2:167 R3:59
Disassembly: add r3 r0 r1

Cycle:60 States:PC:15 Z:0 R0:55 R1:55 R2:167 R3:59
Disassembly: and r0 r1 r1

Cycle:61 States:PC:16 Z:0 R0:55 R1:89 R2:167 R3:59
```

```
Disassembly: and r1 r3 r3

Cycle:62 States:PC:17 Z:1 R0:55 R1:89 R2:00 R3:59
Disassembly: add r2 r2 r3

Cycle:63 States:PC:18 Z:1 R0:55 R1:89 R2:00 R3:59
Disassembly: bnz 13

Cycle:64 States:PC:19 Z:0 R0:55 R1:89 R2:00 R3:59
Disassembly: and r3 r3 r3

Cycle:65 States:PC:18 Z:0 R0:55 R1:89 R2:00 R3:59
Disassembly: bnz 18

Cycle:66 States:PC:19 Z:0 R0:55 R1:89 R2:00 R3:59
Disassembly: and r3 r3 r3
```

APPENDIX C: Fiscas.cpp

```
#include <iostream>
#include <iomanip>
#include <unordered_map>
#include <vector>
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

/*
   An Instruction is line containing assembly code.
   An assembly code contains labels, comments and operands
   and an instruction mnemonic.
*/
class Instruction {
   private:
      vector<string> line;
   public:
      Instruction(vector<string> line) {
         this->line = line;
      }

      vector<string> getInstruction() { return line; }

      /*
         This function returns a string version of an intruction line
         excluding the defined labels.
      */
      string toString() {
         string s = "";
         for (auto l : line) {
            if(l[l.size()-1] != ':') s += l + " ";
         }
         return s;
      }
};

/*
   An Assembly Instruction is a word instruction consisting of an 8-bit
   binary numbers. In 2-bit representation of opCode and operands it is
   stored in the following order: Op Rn Rm Rd.
   This Class make use of an unordered map called OPS_REF which holds
   2-bits binary values representing an instruction mnemonic or operand.
*/
class AssemblyInstruction {
private:
   vector<string> word;
   unordered_map<string, string> OPS_REF = {
      {"add", "00"}, {"and", "01"}, {"not", "10"},
      {"bnz", "11"}, {"r0", "00"}, {"r1", "01"},
      {"r2", "10"}, {"r3", "11"}
   };
public:
   /* This function initializes the word instruction to empty 2-bits */
   AssemblyInstruction() {
      for (int i = 0; i < 4; i++) { word.push_back(""); }
   }

   /*
      The setWord function sets the binary representation of an assembly
      instruction. The six least significants bit is manupulated in
      accordance of the opCode.
      The "and", "add", and "not" opCodes shift rd to the end. However,
      the "bnz" converts it last six significants bit to a decimal value.
   */
   void setWord(int start, vector<string> &line,
      unordered_map<string, int> labels, int lCount) {
      int count = 1, index = 1;
      string opCode = line[start];
      toLower(opCode);

      word[0] = OPS_REF[opCode];

      for (int i = start+1; i < line.size(); i++) {
         toLower(line[i]);

         if (line[i].size() > 2 && line[i][0] == 'r') line[i].erase(2);

         if (line[i][0] == 'r' && OPS_REF.find(line[i]) != OPS_REF.end()) {
            word[index++] = OPS_REF[line[i]];
         } else if (line[i][0] == 'r') {
            cout << "<Invalid register " << line[i] << ">\n";
            exit(0);
         }
      }

      if (opCode == "not") {
         if (line.size() - start != 3) {
            cout << "<Instructure is missing at least one operand>" << endl;
            exit(0);
         }

         string temp = word[1];
         word[3] = word[1];
         word[1] = word[2];
         word[2] = "00";
```

```
      }
      else if (opCode == "bnz") {
         if (line.size() - start != 2) {
            cout << "<Instruction is missing at least one operand>" << endl;
            exit(0);
         }

         unordered_map<string, int>::iterator it=labels.find(line[start+1]);

         if (it == labels.end()) {
            cout << "<Label <" << line[start + 1];
            cout << "> on line <" << lCount;
            cout << "> is undefined.>" << endl;
            exit(0);
         }

         string bnzWord = decToBin(labels[line[start+1]]);

         int index = 1;
         for (int i = 0; i < bnzWord.size(); i += 2) {
            word[index] += bnzWord[i];
            word[index++] += bnzWord[i + 1];
         }
      }
      else {
         if (opCode != "and" && opCode != "add") {
            cout << "<Invalid operand for the opCode>" << endl;
         }

         if (line.size() - start != 4) {
            cout << "<Instruction is missing at least one operand>" << endl;
            exit(0);
         }

         for (int i = 1; i < word.size()-1; i++) {
            string temp = word[i];
            word[i] = word[i + 1];
            word[i + 1] = temp;
         }
      }
   }

   /* returns the vector containing the instruction word */
   vector<string> getWord() { return word; }

   /* Converts a string into lower case */
   void toLower(string &s) {
      for (int i = 0; i < s.size(); i++) {
         s[i] = tolower(s[i]);
      }
   }

   /* This function is used to convert an integer into binary */
   string decToBin(int dec) {
      string bin = "";
      while (dec != 0) {
         bin = to_string(dec % 2) + bin;
         dec /= 2;
      }
      while (bin.size() != 6) { bin = '0' + bin; }
      return bin;
   }
};

/*
   This Assembler class is used to mimic the behavior of an assembler.
   It has a label map which stores the addresses of the different labels,
   a vector of instructions representing the instructions in the input
   file, a vector of assembly instructions consisting of 8-bit binary
   instructions, and a vector named hexCode containing the machine code.
*/
class Assembler {
   private:
      unordered_map<string, int> labels;
      vector<Instruction> instructions;
      vector<AssemblyInstruction> computerInstructions;
      vector<string> hexCode;

   public:
      /*
         This method reads an input file and converts the instructions into
         into binary representation.
      */
      void readData(string path) {
         firstPass(path);
         secondPass();
      }

      /*
         The firstPass reads an input file and converts each instruction line
         into an instruction object. It also keep track of labels and their
         addresses.
         It also ouput error such invalid label definitions and use, and out
         of bound memory storage.
      */
      void firstPass(string path) {
         ifstream inputFile;
         inputFile.open(path, ios::in);

         if (inputFile) {
```

```
      string line;
      int count = 0;

      while (getline(inputFile, line)) {
         if (count > 63) {
            cout << "<Output file is larger than system memory>\n";
            exit(0);
         }

         if (line.empty()) continue;
         if (isComment(line)) continue;

         vector<string> splittedLine = split(line, ' ');
         Instruction i(splittedLine);
         instructions.push_back(i);

         string firstWord = splittedLine[0];

         if (firstWord.size() != 0 && isLabel(firstWord)) {
            firstWord.erase(firstWord.size() - 1);
            if (labels.find(firstWord) != labels.end()) {
               cout << "Label <"<< firstWord << "> on line <";
               cout << count << "> is already defined." << endl;
               exit(0);
            }
            labels.emplace(firstWord, count);
         }
         count++;
      }
      inputFile.close();
   }
   else {
      cout << "<File <" <<path<< "> was not found>\n";
      exit(0);
   }
}

/*
   The second pass converts each instruction object into an assembly
   instruction object, and adds it to the computersInstruction vector.
*/
void secondPass() {
   for (int i = 0; i < instructions.size(); i++) {
      vector<string> instruction = instructions[i].getInstruction();
      AssemblyInstruction ai;
      if (isComment(instruction[0])) {
         continue;
      }
      else if(isLabel(instruction[0])) {
         ai.setWord(1, instruction, labels, i);
      }
      else {
         ai.setWord(0, instruction, labels, i);
      }
      computerInstructions.push_back(ai);
   }
}

/*
   WriteData converts each 8-bit instruction word into hexadecimal
   and stores the resulting value into an object file, and the
   hexCode vector.
*/
void writeData(string path) {
   ofstream outputFile;

   outputFile.open(path, ios::out);

   if (outputFile) {
      outputFile << "v2.0 raw" << endl;

      for (auto ci : computerInstructions) {
         vector<string> word = ci.getWord();
         string leftHex = binToHex(word[0]+word[1]);
         string rightHex = binToHex(word[2] + word[3]);
         string hexLine = "";
         hexLine.append(leftHex);
         hexLine.append(rightHex);
         outputFile << hexLine << endl;
         hexCode.push_back(hexLine);
      }
      outputFile.close();
   }
   else {
      cout << "<Invalid input for object file ";
      cout << "<" << path << " >>" << endl;
      exit(0);
   }
}

/* This method return a string with a leading 0 for single digits */
/* and return a string representation of non single digits  */
string strDig(int d) {
   if (d <= 9) {
      return "0" + to_string(d);
   }
   else return to_string(d);
}

/* Displays the listing for -l command option */
```

```
void displayListing() {
   cout << "*** LABEL LIST***" << endl;

   unordered_map<string, int>::iterator it = labels.begin();

   for (it; it != labels.end(); it++) {
      cout << setw(8) << left << it->first;
      cout << setw(4)<< strDig(it->second) << endl;
   }

   cout << "*** MACHINE PROGRAM ***" << endl;
   for (int i = 0; i < hexCode.size(); i++) {
      cout << strDig(i) << ":" <<left << setw(5) << hexCode[i];
      cout <<setw(15)<< instructions[i].toString() << endl;
   }
}

/* Converts a binary string into hexadecimal */
string binToHex(string bin) {
   unordered_map<int, string> hexVal = {
      {10, "A"}, {11, "B"}, {12, "C"},
      {13, "D"}, {14, "E"}, {15, "F"}
   };

   int power = 1;
   int val = 0;

   for (int i = bin.size() - 1; i >= 0; i--) {
      if (bin[i] == '1') { val += power; }
      power *= 2;
   }

   if (val > 9) {
      return hexVal[val];
   } else {
      return to_string(val);
   }
}

/*
   This is a custum split method which splits instruction lines without
   including comments.
*/
vector<string> split(string str, char sep) {
   vector<string> strList;
   string newStr = "";

   for (int i = 0; i < str.size(); i++) {
      if (str[i] == ';') { return strList; }
```

```
            if (str[i] != sep) {
               newStr += str[i];
            }
            else {
               if (newStr.size() != 0) {
                  strList.push_back(newStr);
               }
               newStr = "";
            }
         }

         if (newStr.size() != 0) {
            strList.push_back(newStr);
         }
         return strList;
      }
      /* Check if a word is a label */
      bool isLabel(string word) {
         if (word.size() == 0) return false;
         return word[word.size() - 1] == ':' ? true : false;
      }
      /* Checks if a word is beginning of a comment */
      bool isComment(string word) {
         int count = 0;
         if (word.size() == 0) return false;
         for (auto s : word) {
            if (s == ' ') count++;
            else break;
         }
         return word[count] == ';' ? true : false;
      }
};

/* Prints error message for invalid command operstions */
void errorMessage() {
   cout << "USAGE:  fiscas <source file> <object file> [-l]\n";
   cout << "        -l : print listing to standard error" << endl;
   exit(0);
}

int main(int argc, char** argv) {
   bool showListing = false;
   if (argc < 3 || argc > 4) {
      errorMessage();
   }
   else if (argc == 4) {
      string input = argv[3];
      if (input == "-l") {
         showListing = true;
      }
      else {
         errorMessage();
      }
   }
   Assembler assembler;
   assembler.readData(argv[1]);
   assembler.writeData(argv[2]);
   if (showListing) assembler.displayListing();
}
```

Appendix D: Fiscsim.cpp

```
#include <iostream>
#include <stdlib.h>
#include <cstdio>
#include <fstream>
#include <string>
#include <sstream>
#include <cmath>
#include <cstdint>
#include <vector>
#include <map>

using namespace std;

/* An intruction memory is an 8-bit representation of an hex machine code */
class InstructionMemory {
private:
   uint8_t im[8] = {};

public:
   /* Initializes the instruction memory using an input binary string */
   InstructionMemory(string inst) {
      int index = 0;

      for (auto i : inst) {
         if (i == '1') {
            im[index] = 1;
         }
         else {
            im[index] = 0;
         }
         index++;
      }
   }

   /* Returns 8-bit array instruction memory */
   uint8_t* getIM() { return im; }

   /* Converts the uint8_t instruction into string */
   string toString() {
      string str = "";

      for (int i = 0; i < 8; i++) {
         str += to_string(im[i]);
      }
      return str;
   }
};
```

```
/*
   This object is used to represent assembly intructions. It uses maps to
   map any 2-bit binary input to its opCode or operand.
   The constructor converts an 8-bit binary instruction to an assembly
   instruction.
*/
class DisassemblerInstructions {
private:
   string word[4];
   map<string, string> opCodeMap = {
      {"00", "add"}, {"01", "and"}, {"10", "not"}, {"11", "bnz"}
   };

   map<string, string> registerMap = {
      {"00", "r0"}, {"01", "r1"}, {"10", "r2"}, {"11", "r3"}
   };

public:
   DisassemblerInstructions(string line) {
      int wordIndex = 0;
      string strAddress = "";

      for (int i = 0; i < line.size(); i+=2) {
         string s =  "";
         s += line[i];
         s += line[i + 1];
         strAddress += s;

         if (i == 0) {
            word[wordIndex] = opCodeMap[s];
         }
         else {
            word[wordIndex] = registerMap[s];
         }
         wordIndex++;
      }

      if (word[0] == "not") {
         string temp = word[1];
         word[1] = word[3];
         word[2] = temp;
         word[3] = "";
      }
      else if (word[0] == "bnz") {
         int address = 0, power = 5;
         strAddress.erase(0, 2);

         for (int i = 0; i < strAddress.size(); i++) {
            if (strAddress[i] == '1') { address += pow(2, power); }
            power--;
         }

         word[1] = to_string(address);
         word[2] = "";
         word[3] = "";
      }
      else {
         for (int i = 3; i > 1; i--) {
            string temp = word[i];
            word[i] = word[i - 1];
            word[i - 1] = temp;
         }
      }
   }

   /* Returns a vector containing each assembly word */
   vector<string> getDI() {
      vector<string> vec;
      for (int i = 0; i < 4; i++) vec.push_back(word[i]);
      return vec;
   }
};

/*
   The simulator object compiles an input file by converting hex machine
   codes into 8-bit binary code, and uses the 8-bit binary instruction
   to convert it into assembly code.
*/
class Simulator{
private:
   char zFlag = '0';
   uint8_t registers[4] = {0, 0, 0, 0};
   vector<InstructionMemory> instructions;
   vector<DisassemblerInstructions> decodes;

public:
   void compileFile(string pathname, int numOfCycle, bool show_disassembly) {
      ifstream inputFile;

      inputFile.open(pathname, ios::in);

      if (inputFile) {
         string line;
         int index = 0;

         while (getline(inputFile, line)) {
            if (index == 0 && line != "v2.0 raw") {
               cout << "Invalid header file <"<< line <<">" << endl;
               exit(0);
            }

            if (line != "v2.0 raw") {
               InstructionMemory ins(hexToBin(line[0])+hexToBin(line[1]));
               instructions.push_back(ins);
               DisassemblerInstructions dec(ins.toString());
               decodes.push_back(dec);
            }
            index = 1;
         }
      }
      else {
         cout << "<File <" << pathname << "> not found>" << endl;
      }

      int cycle = 1, PC = 0;
      int loop = 0;

      /* Displays cycles as well as disassembly code */
      while (PC < decodes.size()) {
         if (cycle <= numOfCycle) {
            computeInstruction(decodes[PC], PC, loop);
            displayStates(cycle, PC);

            if (show_disassembly) {
               int index = loop > PC? loop : PC - 1;
               displayDisassembly(decodes[index].getDI());
               loop = 0;
            }
         }
         else { break; }
         cycle++;
      }
   }

   /* Takes care of intruction mnemonic operations */
   void computeInstruction(DisassemblerInstructions instruction,
      int &PC, int &loop ) {
      vector<string> word = instruction.getDI();

      if (word[0] == "not") {
         int dest = customCharToInt(word[1][1]);
         int rn = customCharToInt(word[2][1]);

         registers[dest] = ~registers[rn];
         zFlag = registers[dest] == 0? '1' : '0';
      }
      else if(word[0] == "bnz") {
         /* If in a loop, jump to targeted address */
         /* If zFlag is set, break loop and jump to next address */
         if (zFlag == '1') {
            PC = PC + 1;
            loop = false;
         }
         else {
            loop = PC;
            PC = stoi(word[1]);
         }
         return;
      }
      else {
         int dest = customCharToInt(word[1][1]);
         int rn = customCharToInt(word[2][1]);
         int rm = customCharToInt(word[3][1]);

         if (word[0] == "add") {
            registers[dest] = registers[rn] + registers[rm];
         }
         else {
            registers[dest] = registers[rn] & registers[rm];
         }
         zFlag = registers[dest] == 0? '1' : '0';
      }
      PC++;
   }

   /* Displays each cycle with the different states */
   void displayStates(int cycle, int PC) {
      stringstream r3("");
      r3 << hex << (int)registers[3];
      cout << "Cycle:" <<to_string(cycle)<< " States:PC:" << disNum(PC);
      cout << " Z:" << zFlag;
      cout << " R0:" <<disNum(registers[0])<< " R1:" << disNum(registers[1]);
      cout << " R2:" <<disNum(registers[2])<< " R3:" <<hex<< r3.str();
      cout << endl;
   }

   /* Displays a vector of disassemly as a string line */
   void displayDisassembly(vector<string> word) {
      cout << "Disassembly: ";
      for (int i = 0; i < 4; i++) {
         if (word.size() != 0) {
            cout << word[i] << " ";
         }
      }
      cout << endl << endl;
   }

   /* Returns single digits as string with leading 0's, 255 as FF */
   /* and 254 as FE, and any other numbers as string */
```

```
string disNum(uint8_t num) {
   if (num == 255) return "FF";
   if (num == 254) return "FE";

   if (num < 10) {
      return '0' + to_string(num);
   }
   else {
      return to_string(num);
   }
}

/* Converts characters 0, 1, 2, 3 to their respective integer values */
int customCharToInt(char c) {
   switch (c) {
   case '0':
      return 0;
   case '1':
      return 1;
   case '2':
      return 2;
   case '3':
      return 3;
   default:
      return -1;
   }
}

/* Converts an hexadecimal to binay */
string hexToBin(char hex) {
   map<char, int> hexVal = {
      {'A', 10}, {'B', 11}, {'C', 12},
      {'D', 13}, {'E', 14}, {'F', 15}
   };
   string bin = "";
   int val = 0;

   if (hexVal.find(hex) != hexVal.end()) {
      val = hexVal[hex];
   }
   else {
      val = stoi(to_string(hex));
      char* newHex = &hex;
      sscanf(newHex, "%d", &val);
   }

   while (val != 0) {
      bin = to_string(val % 2) + bin;
      val /= 2;
   }
```

```
      while (bin.size() < 4) { bin = '0' + bin; }
      return bin;
   }
};

/* Output error message for invalid command inputs */
void errorMessage() {
   cout << "USAGE:  fiscsim  <object file> [cycles] [-l]\n";
   cout << "    -d : print disassembly listing with each cycle\n";
   cout << "    if cycles are unspecified the CPU will run for 20 cycles\n";
   exit(1);
}

/* Check if a string is a number */
bool isNumber(string str) {
   for (auto s : str) {
      if (s < '0' || s > '9') return false;
   }
   return true;
}

int main(int argc, char** argv)
{
   int cycles = 20;
   bool showDisassembly = false;

   if (argc < 2 || argc > 4) {
      errorMessage();
   }

   if (argc == 3) {
      string input = argv[2];
      if (isNumber(input)) {
         cycles = stoi(argv[2]);
      }
      else if (input == "-d") {
         showDisassembly = true;
      }
      else errorMessage();
   }
   else if(argc == 4) {
      string input1 = argv[2], input2 = argv[3];
      bool validIn = false;

      if (isNumber(input1) || isNumber(input2)) {
         cycles = isNumber(input1) ? stoi(input1) : stoi(input2);
         validIn = true;
      }
      if (input1 == "-d" || input2 == "-d") {
         showDisassembly = true;
```

```
            validIn = true;
        }
        if(!validIn) { errorMessage(); }
    }

    Simulator simu;
    simu.compileFile(argv[1], cycles, showDisassembly);
}
```