Lab 2 :: CPU Lab 1

Gabriele Nicula

ID: 9192

CSC137-02

FISCAS Assembler

Fiscas.java is conceptually divided into three modules: the driver method in main(), the AssemblyParser class that takes care of parsing and creating the machine code and four Instruction classes that model each of the CPU's available instructions. Overall the Fiscas assembler works as follows:

a) The input file is read line by line, ignoring everything after and including the comment character ';'.
b) In the first pass, it looks for label definitions and extracts the available label names and puts them in a Map<String, Integer> where the key is the label name and the value is the corresponding address. The corresponding address is the address of the next syntactically valid instruction that immediately follows the label.
c) In the first pass it also saves the substring which is the representation of any syntactically valid encountered instruction in a SortedMap<Integer, String> where the key is the line number and the value is the String instruction from the input. These cached strings will be used in the second pass to create the Instruction objects. This is an optimization so we don't scan the input file a second time.
d) When first pass is done, it has the symbol table map and the instructionStatements sorted map filled in. First pass also does basic syntax checks for instruction and register names and number of registers per instruction.
e) In the second pass we can generate machine code. This is done by iterating through the instructionStatements sorted map that has the strings of all the valid instructions and create the associated instruction objects, store them in an array, obtain their machine code by calling toMachineCode() on each of them and stream the bytecodes to the hex(object) file.

Fiscas assembler testing

Fiscas was tested on the two examples in the assignment for output conformance, on handcrafted simple programs and on modified examples that contained errors. For example, invalid instruction names, invalid register names, invalid number of registers, unused or duplicate labels.
In terms of error checking, Fiscas parses the whole file and displays all the encountered errors rather than stopping at the first error. The errors are annotated with their original line number in the input.

FISCSIM Simulator

Implementation is in a single Java file, Fiscsim.java and it is conceptually divided into:

a) Loading the program - the input ".hex" file is read line by line and the opcodes are stored in the "program" byte array of size 64 which models the simulator's ROM memory. The index in the "program" byte array maps to the physical memory address.
b) Executing the program - this is done by calling execute() method and the execute method conceptually does three steps:
    i) Fetch the opcode - this is a simple read from array at index 'pc'
    ii) Decode the opcode into instruction, rn, rm rd (or address) parts. This is done using masking for each part of the opcode and bit shifting.
    iii) Execute the instruction through a switch/case and simple programming logic that follows the instruction purpose. Each executed instruction increments the PC with one notable exception, when a jump is executed to absolute address, the PC is loaded with that address.
    Each ALU instruction also updates the Z flag as per assignment documentation.

Fiscsim testing

Fiscsim was tested with the example hex files from the assignment and hex files generated from the assembler Fiscas test files. The disassembled instructions were tested vs assembler input. Fiscsim was also tested by comparing the R3 values with the outputs from Logisim circuit on the same ".hex" file.

Discussion on fibo1.s, fibo2.s

1. fibo1, program continually computes and outputs Fibonacci numbers (in hex)

   a) fibo1.s source code

```
; assembler code that continually computes and
; outputs Fibonacci numbers (in hex) starting with F(0) = 0, F(1) = 1.
; must output both initial values of the sequence in your output

start:  not r0 r1
        and r1 r0 r1  ; obtained 0 value in r1
        not r0 r1
        add r2 r0 r0
        not r2 r2     ; obtained 1 in r2
; send both initial values r1 and r2 to output in r3
        and r3 r1 r1
        and r3 r2 r2
; computation loop that computes result in r3
loop:   add r3 r2 r1  ; compute next fibonacci
        and r1 r2 r2  ; move r2 to r1
        and r2 r3 r3  ; move r3 to r2
        bnz loop      ; rd is never 0 so loops forever
```

   b) fibo1.s generated list output from: % java Fiscas fibo1.s fibo1.hex -l

```
*** LABEL LIST ***
start   00
loop    07
*** MACHINE PROGRAM ***
00:90 not r0 r1
01:45 and r1 r0 r1
02:90 not r0 r1
03:02 add r2 r0 r0
04:A2 not r2 r2
05:57 and r3 r1 r1
06:6B and r3 r2 r2
07:27 add r3 r2 r1
08:69 and r1 r2 r2
09:7E and r2 r3 r3
0A:C7 bnz loop
```

c) fibo1.s Questions

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

   Last valid output: 0xE9 which is 233 decimal and fits in 8 bit register R3.

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

   52 cycles.

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

   First invalid output: 0x79 which is 121 decimal. Should have been 377 but it is out of range.

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

   The first 15 Fibonacci numbers starting from F(0)=0 and F(1)=1 are: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377.
   In the listed fibo1.s the result is always stored in 8 bit register R3 and it is the sum of R1+R2 which are also 8 bit registers that can hold positive integers 0-255.
   The first invalid value happens when adding 144+233 which is 10010000_b + 11101001_b and results in overflow at bit 7 MSB. The 8 bit result without the lost carry bit is 01111001_b which is equal to the 0x79 output by the simulator on cycle 56.

2) fibo2.s, computes and outputs only the first 12 Fibonacci numbers (in hex)

a) fibo2.s source code

```
; assembler code that computes and outputs only the
; first 12 Fibonacci numbers (in hex) starting with F(0) = 0, F(1) = 1.
; first 12 Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89=0x59
; Must output both initial values of the sequence in your output.

start:  not r0 r1
        and r1 r0 r1  ; obtained 0 value in r1
        not r0 r1
        add r2 r0 r0
        not r2 r2     ; obtained 1 in r2

; obtain 5 then 250 in r0 because our loop computes two fibonacci
; numbers per loop execution
        add r0 r2 r2  ; r0 = 2
        add r0 r0 r0  ; r0 = 4
        add r0 r0 r2  ; r0 = 5
        not r0 r0     ; r0 = 250

; send both initial values r1 and r2 to output in r3
        and r3 r1 r1
        and r3 r2 r2
; hold r1 with value 1
        and r2 r1 r1  ; put 0 in r2
        and r1 r3 r3  ; put 1 in r1

; computation loop that alternatively computes result in r3 and r2
; loops five times computing next 10 Fibonacci numbers since
; we already have F(0) = 0 and F(1) = 1
loop:   add r3 r2 r3  ; compute next fibonacci
        add r2 r2 r3  ; compute next fibonacci again
        add r0 r0 r1  ; increment r0 with one
        bnz loop      ; when r0 becomes 0 loop ends

        and r3 r3 r3  ; side effect, sets z to false
end:    bnz end       ; loops forever because z is false
```

b) fibo2.s generated list output from: % java Fiscas fibo2.s fibo2.hex -l

```
*** LABEL LIST ***
start   00
loop    0D
end     12
*** MACHINE PROGRAM ***
00:90 not r0 r1
01:45 and r1 r0 r1
02:90 not r0 r1
03:02 add r2 r0 r0
04:A2 not r2 r2
```

```
05:28 add r0 r2 r2
06:00 add r0 r0 r0
07:08 add r0 r0 r2
08:80 not r0 r0
09:57 and r3 r1 r1
0A:6B and r3 r2 r2
0B:56 and r2 r1 r1
0C:7D and r1 r3 r3
0D:2F add r3 r2 r3
0E:2E add r2 r2 r3
0F:04 add r0 r0 r1
10:CD bnz loop
11:7F and r3 r3 r3
12:D2 bnz end
```

c) fibo2.s Questions

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

   Initially I thought about using the same procedure like I did for fibo1.s which uses 3 registers to compute Fibonacci numbers however the loop counter increment needs 2 registers so the code was more complicated. I then thought of alternating between two registers when computing the next Fibonacci number, effectively using only two registers and this idea freed up one register so the counter increment become very simple. The two register version works like this:
   R3 = R2 + R3
   R2 = R2 + R3 and repeat.

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

   Limitation 1: no immediate load value in register; all needed values must be prepared.
   Limitation 2: no increment by 1 instruction; all increments needed two registers, one of them = 1.

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

   I would pick increment register by 1, something like:
   ```
   inc Rd  ; Rd <- Rd+1
   ```

   1. *Explain how this instruction would help.*

      First, it would free up one register that it is now used to hold the 1 value.
      Second, blocks of statements that work to setup an initial value of 1 in one of the registers would be simplified, for example:
      ```
      ;-----------------------
      not r0 r1
      and r1 r0 r1  ; obtained 0 value in r1
      not r0 r1
      add r2 r0 r0
      not r1 r2     ; obtained 1 in r1
      ```

```
;----------------------
```
can be replaced by:
```
;----------------------
not r0 r1
and r1 r0 r1  ; obtained 0 value in r1
inc r1        ; obtained 1 in r1
;----------------------
```

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

   I think this instruction can be introduced in the current design by using the opcode unused bits for the 'not' instruction. The 'not' instruction should mandate its unused bits to be both 0 after this change.

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 new 'inc' instruction takes one argument, a destination register Rd.

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

   ```
   inc r0         ; increments value in r0 by 1
   ```

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

   Opcode as instruction id would be 2, the same as 'not' instruction.
   The unused bits {3,2} in 'not' would be used to extend the opcode for example by using bit3=0 and bit2=1. The bits {5, 4} would be unused for this 'inc' instruction.

Op	-	'01'	Rd
2	2	2	2

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.*

   Considering new inc instruction used like:
   ```
   ;-----
   inc r0
   ;-----
   ```
   the opcode would be "10 00 01 00" which is 0x84.
   This opcode will not overlap with any other instruction assuming 'not' instruction expects bits {3, 2} both be 0.
   This new design could work with the current FISC hardware limitations because it doesn't need to change the existing opcodes, except limiting the values for unused bits for 'not' instruction to be both 0.

Conclusion

FISC CPU was an engaging project to work on. From CPU hardware specifications to parsers/assemblers and software CPU simulators back to hardware implementation in Logisim.

In my opinion, this is the best way to understand, or at least try to understand how a CPU conceptually works.
The FISC CPU project is minimal but it also has all the important steps to learn the steps from assembly code to execution on actual hardware:

A. There is a well specified instruction set and the instruction set is easy to work with (compared to a real world CPU specification manual).
B. There is a well defined assembler project with specific requirements going from top level requirements like two-step parsing and symbol table requirements down to error checking. Same for the software CPU simulator.
C. The CPU can actually be fully implemented in Logisim with simple components and the schematic can still be investigated with Logisim's capabilities.
D. The CPU can be simulated on software side on the same exact input that it can be simulated in Logisim.
E. The project components are each well defined in their own, ex. assembler, software CPU simulator and CPU schematic. Each can be implemented separately and tested individually during project's development.
F. The project fully defines an end to end pipeline from CPU instruction specifications to testing the written program's assembled object file on the hardware schematic.

Things I have learned while working on this project:

1. Register mapping - how each register defined in the specification and used in an instruction is actually selected in the schematic with a decoder.
2. Address jumps - how a jump to address is defined and enabled in the assembler source file and how the actual jump is executed through the address multiplexer in the schematic.
3. Opcode decoding - how each part of the opcode is used to select specific components like registers, ALU components to perform the instruction.

This lab really connected the dots for me on a set of tools that I was aware of such as assemblers, executable object files, CPU registers, but did not think of the intricacies of how exactly they can work together.

Appendix

1. fibo1.hex simulator output with disassembly until the first invalid Fibonacci number appears (56 cycles)

gabenicula@Gabes-MacBook-Air Lab2 % java Fiscsim fibo1.hex 56 -d
Cycle:1 State:PC:01 Z:0 R0: FF R1: 00 R2: 00 R3: 00
Disassembly: not r0 r1

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

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

Cycle:4 State:PC:04 Z:0 R0: FF R1: 00 R2: FE R3: 00
Disassembly: add r2 r0 r0

Cycle:5 State:PC:05 Z:0 R0: FF R1: 00 R2: 01 R3: 00
Disassembly: not r2 r2

Cycle:6 State:PC:06 Z:1 R0: FF R1: 00 R2: 01 R3: 00
Disassembly: and r3 r1 r1

Cycle:7 State:PC:07 Z:0 R0: FF R1: 00 R2: 01 R3: 01
Disassembly: and r3 r2 r2

Cycle:8 State:PC:08 Z:0 R0: FF R1: 00 R2: 01 R3: 01
Disassembly: add r3 r2 r1

Cycle:9 State:PC:09 Z:0 R0: FF R1: 01 R2: 01 R3: 01
Disassembly: and r1 r2 r2

Cycle:10 State:PC:0A Z:0 R0: FF R1: 01 R2: 01 R3: 01
Disassembly: and r2 r3 r3

Cycle:11 State:PC:07 Z:0 R0: FF R1: 01 R2: 01 R3: 01

Disassembly: bnz 7

Cycle:12 State:PC:08 Z:0 R0: FF R1: 01 R2: 01 R3: 02
Disassembly: add r3 r2 r1

Cycle:13 State:PC:09 Z:0 R0: FF R1: 01 R2: 01 R3: 02
Disassembly: and r1 r2 r2

Cycle:14 State:PC:0A Z:0 R0: FF R1: 01 R2: 02 R3: 02
Disassembly: and r2 r3 r3

Cycle:15 State:PC:07 Z:0 R0: FF R1: 01 R2: 02 R3: 02
Disassembly: bnz 7

Cycle:16 State:PC:08 Z:0 R0: FF R1: 01 R2: 02 R3: 03
Disassembly: add r3 r2 r1

Cycle:17 State:PC:09 Z:0 R0: FF R1: 02 R2: 02 R3: 03
Disassembly: and r1 r2 r2

Cycle:18 State:PC:0A Z:0 R0: FF R1: 02 R2: 03 R3: 03
Disassembly: and r2 r3 r3

Cycle:19 State:PC:07 Z:0 R0: FF R1: 02 R2: 03 R3: 03
Disassembly: bnz 7

Cycle:20 State:PC:08 Z:0 R0: FF R1: 02 R2: 03 R3: 05
Disassembly: add r3 r2 r1

Cycle:21 State:PC:09 Z:0 R0: FF R1: 03 R2: 03 R3: 05
Disassembly: and r1 r2 r2

Cycle:22 State:PC:0A Z:0 R0: FF R1: 03 R2: 05 R3: 05
Disassembly: and r2 r3 r3

Cycle:23 State:PC:07 Z:0 R0: FF R1: 03 R2: 05 R3: 05
Disassembly: bnz 7

Cycle:24 State:PC:08 Z:0 R0: FF R1: 03 R2: 05 R3: 08
Disassembly: add r3 r2 r1

Cycle:25 State:PC:09 Z:0 R0: FF R1: 05 R2: 05 R3: 08
Disassembly: and r1 r2 r2

Cycle:26 State:PC:0A Z:0 R0: FF R1: 05 R2: 08 R3: 08
Disassembly: and r2 r3 r3

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

Cycle:28 State:PC:08 Z:0 R0: FF R1: 05 R2: 08 R3: 0D
Disassembly: add r3 r2 r1

Cycle:29 State:PC:09 Z:0 R0: FF R1: 08 R2: 08 R3: 0D
Disassembly: and r1 r2 r2

Cycle:30 State:PC:0A Z:0 R0: FF R1: 08 R2: 0D R3: 0D
Disassembly: and r2 r3 r3

Cycle:31 State:PC:07 Z:0 R0: FF R1: 08 R2: 0D R3: 0D
Disassembly: bnz 7

Cycle:32 State:PC:08 Z:0 R0: FF R1: 08 R2: 0D R3: 15
Disassembly: add r3 r2 r1

Cycle:33 State:PC:09 Z:0 R0: FF R1: 0D R2: 0D R3: 15
Disassembly: and r1 r2 r2

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

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

Cycle:36 State:PC:08 Z:0 R0: FF R1: 0D R2: 15 R3: 22
Disassembly: add r3 r2 r1

Cycle:37 State:PC:09 Z:0 R0: FF R1: 15 R2: 15 R3: 22
Disassembly: and r1 r2 r2

Cycle:38 State:PC:0A Z:0 R0: FF R1: 15 R2: 22 R3: 22
Disassembly: and r2 r3 r3

Cycle:39 State:PC:07 Z:0 R0: FF R1: 15 R2: 22 R3: 22
Disassembly: bnz 7

Cycle:40 State:PC:08 Z:0 R0: FF R1: 15 R2: 22 R3: 37
Disassembly: add r3 r2 r1

Cycle:41 State:PC:09 Z:0 R0: FF R1: 22 R2: 22 R3: 37
Disassembly: and r1 r2 r2

Cycle:42 State:PC:0A Z:0 R0: FF R1: 22 R2: 37 R3: 37
Disassembly: and r2 r3 r3

Cycle:43 State:PC:07 Z:0 R0: FF R1: 22 R2: 37 R3: 37
Disassembly: bnz 7

Cycle:44 State:PC:08 Z:0 R0: FF R1: 22 R2: 37 R3: 59
Disassembly: add r3 r2 r1

Cycle:45 State:PC:09 Z:0 R0: FF R1: 37 R2: 37 R3: 59
Disassembly: and r1 r2 r2

Cycle:46 State:PC:0A Z:0 R0: FF R1: 37 R2: 59 R3: 59
Disassembly: and r2 r3 r3

Cycle:47 State:PC:07 Z:0 R0: FF R1: 37 R2: 59 R3: 59
Disassembly: bnz 7

Cycle:48 State:PC:08 Z:0 R0: FF R1: 37 R2: 59 R3: 90

Disassembly: add r3 r2 r1

Cycle:49 State:PC:09 Z:0 R0: FF R1: 59 R2: 59 R3: 90
Disassembly: and r1 r2 r2

Cycle:50 State:PC:0A Z:0 R0: FF R1: 59 R2: 90 R3: 90
Disassembly: and r2 r3 r3

Cycle:51 State:PC:07 Z:0 R0: FF R1: 59 R2: 90 R3: 90
Disassembly: bnz 7

Cycle:52 State:PC:08 Z:0 R0: FF R1: 59 R2: 90 R3: E9
Disassembly: add r3 r2 r1

Cycle:53 State:PC:09 Z:0 R0: FF R1: 90 R2: 90 R3: E9
Disassembly: and r1 r2 r2

Cycle:54 State:PC:0A Z:0 R0: FF R1: 90 R2: E9 R3: E9
Disassembly: and r2 r3 r3

Cycle:55 State:PC:07 Z:0 R0: FF R1: 90 R2: E9 R3: E9
Disassembly: bnz 7

Cycle:56 State:PC:08 Z:0 R0: FF R1: 90 R2: E9 R3: 79
Disassembly: add r3 r2 r1

2. fibo2.hex simulator output with disassembly until Fibonacci (12) is calculated plus five cycles shown that the program is effectively halted.

```
gabenicula@Gabes-MacBook-Air Lab2 % java Fiscsim fibo2.hex 43 -d
Cycle:1 State:PC:01 Z:0 R0: FF R1: 00 R2: 00 R3: 00
Disassembly: not r0 r1

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

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

Cycle:4 State:PC:04 Z:0 R0: FF R1: 00 R2: FE R3: 00
Disassembly: add r2 r0 r0

Cycle:5 State:PC:05 Z:0 R0: FF R1: 00 R2: 01 R3: 00
Disassembly: not r2 r2

Cycle:6 State:PC:06 Z:0 R0: 02 R1: 00 R2: 01 R3: 00
Disassembly: add r0 r2 r2

Cycle:7 State:PC:07 Z:0 R0: 04 R1: 00 R2: 01 R3: 00
Disassembly: add r0 r0 r0

Cycle:8 State:PC:08 Z:0 R0: 05 R1: 00 R2: 01 R3: 00
Disassembly: add r0 r0 r2

Cycle:9 State:PC:09 Z:0 R0: FA R1: 00 R2: 01 R3: 00
Disassembly: not r0 r0

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

Cycle:11 State:PC:0B Z:0 R0: FA R1: 00 R2: 01 R3: 01
```

Disassembly: and r3 r2 r2

Cycle:12 State:PC:0C Z:1 R0: FA R1: 00 R2: 00 R3: 01
Disassembly: and r2 r1 r1

Cycle:13 State:PC:0D Z:0 R0: FA R1: 01 R2: 00 R3: 01
Disassembly: and r1 r3 r3

Cycle:14 State:PC:0E Z:0 R0: FA R1: 01 R2: 00 R3: 01
Disassembly: add r3 r2 r3

Cycle:15 State:PC:0F Z:0 R0: FA R1: 01 R2: 01 R3: 01
Disassembly: add r2 r2 r3

Cycle:16 State:PC:10 Z:0 R0: FB R1: 01 R2: 01 R3: 01
Disassembly: add r0 r0 r1

Cycle:17 State:PC:0D Z:0 R0: FB R1: 01 R2: 01 R3: 01
Disassembly: bnz D

Cycle:18 State:PC:0E Z:0 R0: FB R1: 01 R2: 01 R3: 02
Disassembly: add r3 r2 r3

Cycle:19 State:PC:0F Z:0 R0: FB R1: 01 R2: 03 R3: 02
Disassembly: add r2 r2 r3

Cycle:20 State:PC:10 Z:0 R0: FC R1: 01 R2: 03 R3: 02
Disassembly: add r0 r0 r1

Cycle:21 State:PC:0D Z:0 R0: FC R1: 01 R2: 03 R3: 02
Disassembly: bnz D

Cycle:22 State:PC:0E Z:0 R0: FC R1: 01 R2: 03 R3: 05
Disassembly: add r3 r2 r3

Cycle:23 State:PC:0F Z:0 R0: FC R1: 01 R2: 08 R3: 05
Disassembly: add r2 r2 r3

Cycle:24 State:PC:10 Z:0 R0: FD R1: 01 R2: 08 R3: 05
Disassembly: add r0 r0 r1

Cycle:25 State:PC:0D Z:0 R0: FD R1: 01 R2: 08 R3: 05
Disassembly: bnz D

Cycle:26 State:PC:0E Z:0 R0: FD R1: 01 R2: 08 R3: 0D
Disassembly: add r3 r2 r3

Cycle:27 State:PC:0F Z:0 R0: FD R1: 01 R2: 15 R3: 0D
Disassembly: add r2 r2 r3

Cycle:28 State:PC:10 Z:0 R0: FE R1: 01 R2: 15 R3: 0D
Disassembly: add r0 r0 r1

Cycle:29 State:PC:0D Z:0 R0: FE R1: 01 R2: 15 R3: 0D
Disassembly: bnz D

Cycle:30 State:PC:0E Z:0 R0: FE R1: 01 R2: 15 R3: 22
Disassembly: add r3 r2 r3

Cycle:31 State:PC:0F Z:0 R0: FE R1: 01 R2: 37 R3: 22
Disassembly: add r2 r2 r3

Cycle:32 State:PC:10 Z:0 R0: FF R1: 01 R2: 37 R3: 22
Disassembly: add r0 r0 r1

Cycle:33 State:PC:0D Z:0 R0: FF R1: 01 R2: 37 R3: 22
Disassembly: bnz D

Cycle:34 State:PC:0E Z:0 R0: FF R1: 01 R2: 37 R3: 59
Disassembly: add r3 r2 r3

Cycle:35 State:PC:0F Z:0 R0: FF R1: 01 R2: 90 R3: 59
Disassembly: add r2 r2 r3

Cycle:36 State:PC:10 Z:1 R0: 00 R1: 01 R2: 90 R3: 59
Disassembly: add r0 r0 r1

Cycle:37 State:PC:11 Z:1 R0: 00 R1: 01 R2: 90 R3: 59
Disassembly: bnz D

Cycle:38 State:PC:12 Z:0 R0: 00 R1: 01 R2: 90 R3: 59
Disassembly: and r3 r3 r3

Cycle:39 State:PC:12 Z:0 R0: 00 R1: 01 R2: 90 R3: 59
Disassembly: bnz 12

Cycle:40 State:PC:12 Z:0 R0: 00 R1: 01 R2: 90 R3: 59
Disassembly: bnz 12

Cycle:41 State:PC:12 Z:0 R0: 00 R1: 01 R2: 90 R3: 59
Disassembly: bnz 12

Cycle:42 State:PC:12 Z:0 R0: 00 R1: 01 R2: 90 R3: 59
Disassembly: bnz 12

Cycle:43 State:PC:12 Z:0 R0: 00 R1: 01 R2: 90 R3: 59
Disassembly: bnz 12

3. Java code for Fiscas.java

```
// Student Name: Gabriele Nicula
// Student ID: 219969192
// CPU Lab 1

import java.io.*;
import java.util.*;

public class Fiscas {

  public interface Instruction {
    public byte toMachineCode();
  }

  public class ThreeOpInstruction implements Instruction {
    private final String name;
    private final int opcode;
    private final int rd;
    private final int rn;
    private final int rm;

    public ThreeOpInstruction(String name, int opcode, int rd, int rn, int rm) {
      this.name = name;
      this.opcode = opcode;
      this.rd = rd;
      this.rn = rn;
      this.rm = rm;
    }

    public int getRd() {
      return rd;
    }

    public int getRn() {
      return rn;
    }

    public int getRm() {
      return rm;
    }

    public byte toMachineCode() {
      int opBits = opcode << 6;
      int rnBits = rn << 4;
      int rmBits = rm << 2;
      int rdBits = rd;
      int machineCode = opBits | rnBits | rmBits | rdBits;
      return (byte) machineCode;
    }

    @Override
    public String toString() {
```

```
    return name + " r" + rd + " r" + rn + " r" + rm;
  }

}

public class ADD_Instruction extends ThreeOpInstruction {

  public ADD_Instruction(int rd, int rn, int rm) {
    super("add", 0x0, rd, rn, rm);
  }

}

public class AND_Instruction extends ThreeOpInstruction {

  public AND_Instruction(int rd, int rn, int rm) {
    super("and", 0x1, rd, rn, rm);
  }

}

public class NOT_Instruction implements Instruction {

  private final int opcode;
  private final int rd;
  private final int rn;

  public NOT_Instruction(int rd, int rn) {
    this.rd = rd;
    this.rn = rn;
    opcode = 0x02;
  }

  @Override
  public byte toMachineCode() {
    int opBits = opcode << 6;
    int rnBits = rn << 4;
    int machineCode = opBits | rnBits | rd;
    return (byte) machineCode;
  }

  @Override
  public String toString() {
    return "not r" + rd + " r" + rn;
  }
}

public class BNZ_Instruction implements Instruction {
  private final int opcode;
  private final int address;
  private final String label;

  public BNZ_Instruction(int address, String label) {
```

```
    this.address = address;
    this.label = label;
    opcode = 0x03;
  }

  @Override
  public byte toMachineCode() {
    int opBits = opcode << 6;
    int machineCode = opBits | address;
    return (byte) machineCode;
  }

  @Override
  public String toString() {
    return "bnz " + label;
  }
}

private class AssemblyParser {
  private final String filename;
  private final Map<String, Integer> symbolTable;
  private final SortedMap<Integer, String> instructionStatements;
  private final ArrayList<Instruction> instructions;
  private int numParseErrors;

  public AssemblyParser(String filename) {
    this.filename = filename;
    this.symbolTable = new HashMap<>();
    this.instructionStatements = new TreeMap<Integer, String>();
    this.instructions = new ArrayList<Instruction>();
    numParseErrors = 0;
  }

  public ArrayList<Instruction> getInstructions() {
    return instructions;
  }

  public void parse() throws IOException {
    // First pass: build symbol table
    try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
      int address = 0;
      int lineNo = 0;
      String line;
      while ((line = br.readLine()) != null) {
        ++lineNo;
        line = line.trim();
        // Ignore comments
        int semicolonIndex = line.indexOf(';');
        if (semicolonIndex != -1) {
          line = line.substring(0, semicolonIndex).trim();
        }
        // System.out.println("While Line " + line);
        if (!line.isEmpty()) {
```

```
// Check for label definition
int colonIndex = line.indexOf(':');
if (colonIndex != -1) {
  String label = line.substring(0, colonIndex).trim();
  if (!symbolTable.containsKey(label)) {
    symbolTable.put(label, address);
  } else {
    System.err.println("Label <" + label + "> on line <" + lineNo
        + "> is already defined.");
  }
  // Remove label from line
  line = line.substring(colonIndex + 1).trim();
}

// Parse instruction
boolean isCorrectInstruction = true;
if (!line.isEmpty()) {
  String[] tokens = line.split("\\s+");
  String opcode = tokens[0].toLowerCase();
  if (opcode.equals("add") || opcode.equals("and")
      || opcode.equals("not") || opcode.equals("bnz")) {
    boolean checkRegCount = true;
    boolean checkRegNames = true;
    switch (opcode) {
    case "add":
    case "and":
      checkRegCount = tokens.length == 4;
      if (!checkRegCount) {
        System.err
            .println("Invalid number of registers for instruction <"
                + tokens[0] + "> on line <" + lineNo + ">");
        ++numParseErrors;
        isCorrectInstruction = false;
      } else {
        checkRegNames = checkRegisterNames(tokens, 3);
        if (!checkRegNames) {
          System.err.println(" on line <" + lineNo + ">");
          ++numParseErrors;
          isCorrectInstruction = false;
        }
      }
      break;
    case "not":
      checkRegCount = tokens.length == 3;
      if (!checkRegCount) {
        System.err
            .println("Invalid number of registers for instruction <"
                + tokens[0] + "> on line <" + lineNo + ">");
        ++numParseErrors;
        isCorrectInstruction = false;
      } else {
        checkRegNames = checkRegisterNames(tokens, 2);
        if (!checkRegNames) {
```

```
                System.err.println(" on line <" + lineNo + ">");
                ++numParseErrors;
                isCorrectInstruction = false;
              }
            }
            break;
          case "bnz":
            boolean checkLength = tokens.length == 2;
            if (!checkLength) {
              System.err.println(
                  "Invalid argument count for bnz instruction on line <"
                      + lineNo + ">");
              ++numParseErrors;
              isCorrectInstruction = false;
            }
            break;
          }
          if (isCorrectInstruction) {
            instructionStatements.put(lineNo, line);
            address++;
          }
        } else {
          System.err.println("Invalid opcode: <" + opcode + "> on line <"
              + lineNo + ">");
          ++numParseErrors;
        }
      }
    }
  }
}
// System.out.println("Instruction statements size " +
// instructionStatements.size());
// Second pass: generate machine code
for (Integer lineNo : instructionStatements.keySet()) {
  String line = instructionStatements.get(lineNo);
  // System.out.println(line);
  line = line.trim();
  // Parse instruction
  String[] tokens = line.split("\\s+");
  String opcode = tokens[0].toLowerCase();
  switch (opcode) {
  case "add":
    instructions.add(new ADD_Instruction(parseRegister(tokens[1]),
        parseRegister(tokens[2]), parseRegister(tokens[3])));
    break;
  case "and":
    instructions.add(new AND_Instruction(parseRegister(tokens[1]),
        parseRegister(tokens[2]), parseRegister(tokens[3])));
    break;
  case "not":
    instructions.add(new NOT_Instruction(parseRegister(tokens[1]),
        parseRegister(tokens[2])));
    break;
```

```
    case "bnz":
      String label = tokens[1];
      // System.out.println("debug label " + label);
      if (symbolTable.containsKey(label)) {
        instructions
            .add(new BNZ_Instruction(symbolTable.get(label), label));
      } else {
        System.err.println(
            "Label <" + label + "> on line <" + lineNo + "> is undefined");
        ++numParseErrors;
      }
      break;
    default:
      throw new IllegalArgumentException("Invalid opcode: " + opcode);
    }
  }

  if (numParseErrors > 0) {
    System.err.println("There were " + numParseErrors + " parsing errors.");
  }
  boolean outOfBoundsError = false;
  if (instructions.size() > 64) {
    System.err.println(
        "Too many instructions: object file is larger than 64 byte " +
        "system memory.");
    outOfBoundsError = true;
  }
  for (String lbl : symbolTable.keySet()) {
    // System.out.println("byte value " + symbolTable.get(lbl).byteValue());
    if (symbolTable.get(lbl) > 63) {
      System.err.println(
          "Invalid address <" + String.format("%02X", symbolTable.get(lbl))
              + "> for label <" + lbl + ">.");
      outOfBoundsError = true;
    }
  }
  // If any errors were detected, exit!
  if (numParseErrors > 0 || outOfBoundsError) {
    System.exit(1);
  }
}

private void writeMachineOutput(String outputFileName) {
  try {
    BufferedWriter writer = new BufferedWriter(
        new FileWriter(outputFileName));
    writer.write("v2.0 raw");
    writer.newLine();
    for (Instruction instruction : instructions) {
      writer.write(String.format("%02X", instruction.toMachineCode()));
      writer.newLine();
    }
    writer.close();
```

```
    } catch (IOException e) {
      System.err.println("Error writing to file: " + e.getMessage());
    }
  }

  private int parseRegister(String regName) {
    switch (regName.toLowerCase()) {
    case "r0":
      return 0;
    case "r1":
      return 1;
    case "r2":
      return 2;
    case "r3":
      return 3;
    default:
      return -1;
    }
  }

  private boolean checkRegisterNames(String[] instructionTokens, int numReg) {
    for (int i = 1; i < numReg + 1; ++i) {
      int regValue = parseRegister(instructionTokens[i]);
      if (regValue == -1) {
        System.err
            .print("Invalid register name <" + instructionTokens[i] + ">");
        return false;
      }
    }
    return true;
  }

  public Map<String, Integer> getSymbolTable() {
    return symbolTable;
  }
} // END AssemblyParser

private AssemblyParser createParser(String filename) {
  return new AssemblyParser(filename);
}

private static void printSymbolTable(Map<String, Integer> symbolTable) {
  System.err.println("*** LABEL LIST ***");
  List<Map.Entry<String, Integer>> entries = new ArrayList<>(
      symbolTable.entrySet());
  Collections.sort(entries, Map.Entry.comparingByValue());
  for (Map.Entry<String, Integer> entry : entries) {
    System.err.printf("%-8s%02X%n", entry.getKey(), entry.getValue());
  }
}

public static void printMachineCode(ArrayList<Instruction> instructions) {
  System.err.println("*** MACHINE PROGRAM ***");
```

```
    for (int i = 0; i < instructions.size(); i++) {
      Instruction instruction = instructions.get(i);
      String address = String.format("%02X", i);
      String machineCode = String.format("%02X", instruction.toMachineCode());
      System.err
          .println(address + ":" + machineCode + "\t" + instruction.toString());
    }
  }

  public static void main(String[] args) throws IOException {

    // Check that the correct number of arguments were provided
    // NOTE: there's and extra q in the doc provided usage string.
    // USAGE: java Fiscas.javaq <source file> <object file> [-l]
    // -l : print listing to standard error
    if (args.length < 2 || args.length > 3) {
      System.out.println("USAGE: java Fiscas <source file> <object file> [-l]");
      System.out.println("\t-l : print listing to standard error");
      return;
    }

    // Parse the command line arguments
    String sourceFile = args[0];
    String objectFile = args[1];
    boolean printSymbolTable = false;
    if (args.length == 3 && args[2].equals("-l")) {
      printSymbolTable = true;
    }

    // Create the Fiscas object
    Fiscas as = new Fiscas();
    AssemblyParser assembler = as.createParser(sourceFile);

    // Assemble the source file
    assembler.parse();

    // Now write the object file
    assembler.writeMachineOutput(objectFile);

    // Print the symbol table if requested
    if (printSymbolTable) {
      Map<String, Integer> symbolTable = assembler.getSymbolTable();
      printSymbolTable(symbolTable);
      printMachineCode(assembler.getInstructions());
    }
  }
}
```

4. Java code for Fiscsim.java

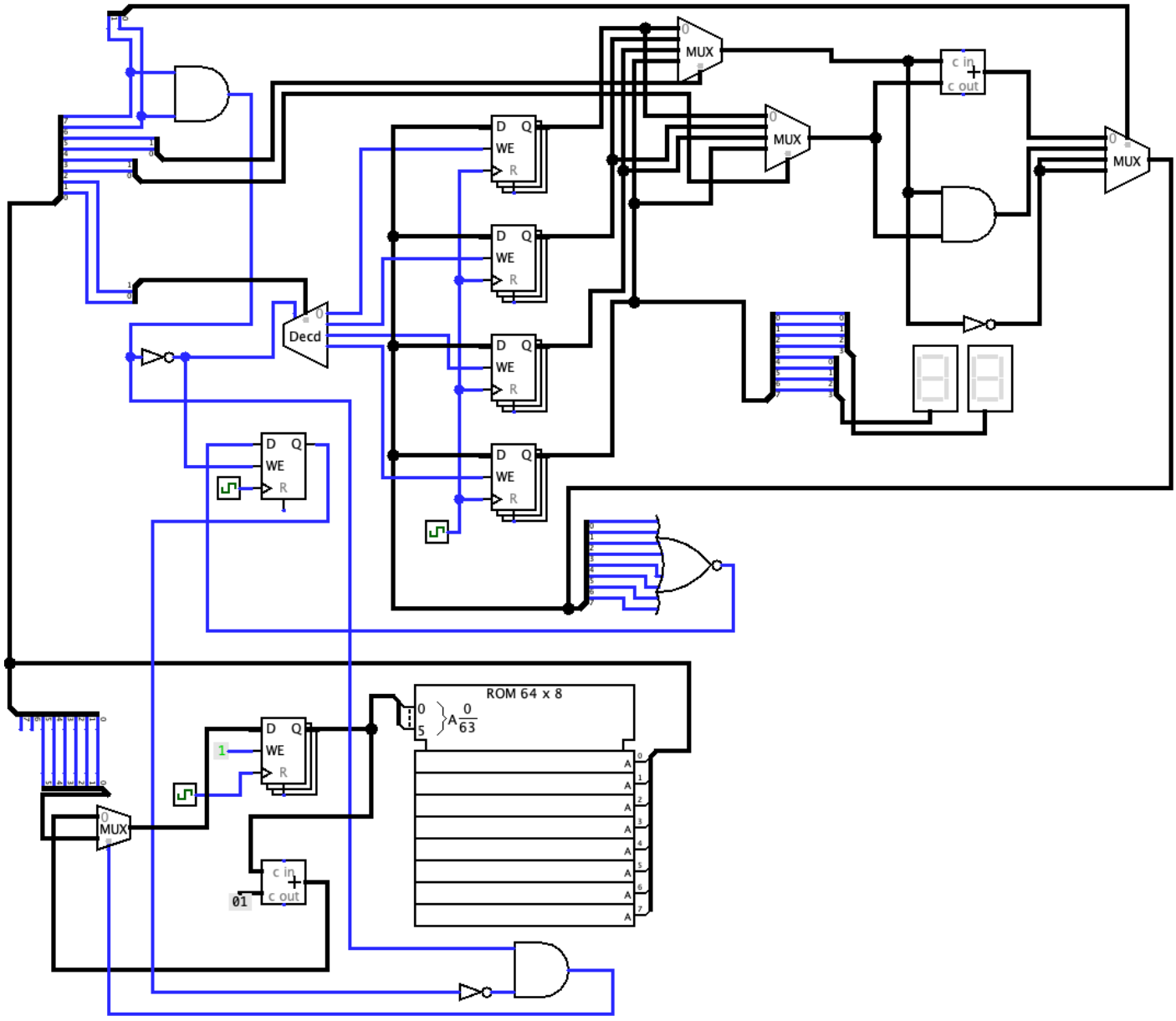
```
// Student Name: Gabriele Nicula
// Student ID: 219969192
// CPU Lab 1

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

public class Fiscsim {

  private static String versionline = "";
  private static int maxCycles = 20;
  private static boolean showDisassembly = false;
  public static final int MAX_ROM_CAPACITY = 64;

  private static class CPU {
    private byte r0;
    private byte r1;
    private byte r2;
    private byte r3;
    private int pc;
    private int cycle;
    private int z;

    public CPU() {
      r0 = 0;
      r1 = 0;
      r2 = 0;
      r3 = 0;
      pc = 0;
      cycle = 0;
      z = 0;
    }

    private int getRegister(int opcodeRegister) {
      switch (opcodeRegister) {
      case 0:
        return r0;
      case 1:
        return r1;
      case 2:
        return r2;
      case 3:
        return r3;
      default:
        System.err
            .println("Trying to read a value from an invalid register ID: <"
                + opcodeRegister + ">");
        return -1;
      }
    }
```

```
private void setRegister(int opcodeRegister, byte toWrite) {
  switch (opcodeRegister) {
  case 0:
    r0 = toWrite;
    break;
  case 1:
    r1 = toWrite;
    break;
  case 2:
    r2 = toWrite;
    break;
  case 3:
    r3 = toWrite;
    break;
  default:
    System.err.println("Trying to set a value to an invalid register ID: <"
        + opcodeRegister + ">");
  }
}

public void execute(byte[] program) {
  while (cycle < maxCycles) {
    String disassembledInstruction = "Disassembly: ";
    // Fetch opcode from memory.
    byte opcode = program[pc];
    // Decode opcode
    int instr = (opcode & 0xC0) >> 6; // Extract instruction code from the
                                      // two most significant bits
    int rn = (opcode & 0x30) >> 4; // Extract register rn from bits 5-4
    int rm = (opcode & 0x0C) >> 2; // Extract register rm from bits 3-2
    int rd = opcode & 0x03; // Extract register rd from bits 1-0
    int address = opcode & 0x3F; // Mask for 6 bit BNZ address (5-0)

    int result = 0;
    // Execute instruction
    switch (instr) {
    case 0: // ADD
      result = getRegister(rn) + getRegister(rm);
      setRegister(rd, (byte) result);
      z = getRegister(rd) == 0 ? 1 : 0;
      ++pc;
      disassembledInstruction += "add " + "r" + rd + " r" + rn + " r" + rm;
      break;
    case 1: // AND
      result = getRegister(rn) & getRegister(rm);
      setRegister(rd, (byte) result);
      z = getRegister(rd) == 0 ? 1 : 0;
      ++pc;
      disassembledInstruction += "and " + "r" + rd + " r" + rn + " r" + rm;
      break;
    case 2: // NOT
      result = ~getRegister(rn) & 0xFF;
```

```
        setRegister(rd, (byte) result);
        z = getRegister(rd) == 0 ? 1 : 0;
        ++pc;
        disassembledInstruction += "not " + "r" + rd + " r" + rn;
        break;
      case 3: // BNZ
        if (z == 0) {
          pc = address;
        } else {
          ++pc;
        }
        // NOTE: It is not clear from the assignment if the address printed
        // in the disassembly must be in hex format or decimal. Ex. <bnz 7>.
        // Leaving it on not formated hex, switch by uncommenting the decimal
        // version or the formatted version if so desired.
        // Hex
        disassembledInstruction += "bnz " + String.format("%X", address);
        // disassembledInstruction += "bnz " + String.format("0x%02X", address);
        // Decimal
        // disassembledInstruction += "bnz " + address;
        break;
      default:
        System.err.printf("Unknown instruction code: %d\n", instr);
        break;
      }
      ++cycle;
      System.err.println(toString());
      if (showDisassembly) {
        System.err.println(disassembledInstruction);
        System.err.println();
      }
    }
  }

  public String toString() {
    return String.format(
        "Cycle:%d State:PC:%02X Z:%d R0: %02X R1: %02X R2: %02X R3: %02X",
        cycle, pc, z, r0, r1, r2, r3);
  }
}

public static void main(String[] args) {
  if (args.length < 1 || args.length > 3) {
    System.err.println(
        "USAGE:  java Fiscsim.java  <object file> [<cycles>] [-l]\n"
        + "      -d : print disassembly listing with each cycle\n"
        + "      if cycles are unspecified the CPU will run for 20 cycles");
    System.exit(1);
  }
  maxCycles = 20;
  showDisassembly = false;

  String fileName = args[0];
```

```
    String arg1 = args.length > 1 ? args[1] : null;
    if (arg1 != null) {
      if (arg1.equals("-d")) {
        showDisassembly = true;
      } else {
        maxCycles = Integer.parseInt(arg1);
      }
    }
    String arg2 = args.length > 2 ? args[2] : null;
    if (arg2 != null) {
      if (arg2.equals("-d")) {
        showDisassembly = true;
      } else {
        maxCycles = Integer.parseInt(arg2);
      }
    }

    CPU cpu = new CPU();
    try (BufferedReader br = new BufferedReader(new FileReader(fileName))) {
      byte[] program = new byte[MAX_ROM_CAPACITY];
      String line;
      versionline = br.readLine();
      int i = 0;
      while ((line = br.readLine()) != null) {
        byte opcode = (byte) Integer.parseInt(line, 16);
        program[i] = opcode;
        ++i;
      }
      // for (int j = 0; j < i; ++j) {
      // System.out.printf("%02X\n", program[j]);
      // }
      cpu.execute(program);

    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Circuit schematic exported from Logisim



Circuit running in Logisim with fibo2.hex loaded