Lab 3 :: CPU Lab 2

Gabriele Nicula

ID: 9192

CSC137-02

Theory of Operation

Four Bit CLA:

The 4-bit CLA follows the implementation shown in the ZyBooks chapter 8.3 in activity 2. It is composed of 4 full 1-bit adders that output sum, generate, and propagate. These adders are connected to the Carry-lookahead logic implemented from the equations highlighted in orange in the cited ZyBooks activity. The output of a 4-bit CLA is the sum of the inputs and a carry-out bit. Two of these 4-bit CLA subcircuits are used together by connecting the carry-out from the first one to the carry-in of the second one to create an 8-bit adder which is subsequently used in the ALU.

Display Decoder:

Uses four 7-Segment Displays from Logisim Input/Output library to display signed 3-digit decimal values. These 7-Segment Displays are each driven by a BCD to 7-segment library circuit which is controlled by a custom combinatorial circuit. This circuit does the signed 8-bit to three-digit BCD decoding. This combinatorial circuit was synthesized based on a truth table for 8-bit to 3-digit BCD decoding. The truth table was manually written respecting Logisim import table format and then imported in Logisim. The synthesized circuit was used to create the subcircuit.

ALU:

The ALU's Arithmetic Unit uses the 8-bit Adder described above to perform 8-bit addition and subtraction. For subtraction, a signal is extracted by a 4-bit ALU opcode input decoder to select the inverted output for the second operand (b) to perform a+NOT(b). Logical operations, AND, OR, and NOT, are performed by 8-bit input gates corresponding to the operation. The SHL and ASR are both implemented using the same combinational shifter circuit. It can be controlled from the 4-bit ALU opcode input decoder which selects either logical shift left (SHL) or arithmetic shift right (ASR). All of the function outputs, arithmetic, logical, and shift are multiplexed together and the output result is selected using the ALU opcode input. Additionally, ALU outputs a 'Z' flag which is 1 when the result is 0, and a 'C' flag which is 1 when there is an overflow. Also, the result multiplexer allows a passthrough of the 'A' input when the ALU opcode corresponds to a MOV instruction.

Register File:

Uses four 8-bit registers for CPU registers R0, R1, R2, and R3. Registers are selected for writing by a decoder based on a 2-bit input (WA0). Registers are selected for reading using multiplexers based on three 2-bit signals RA0, RA1, and RA2. RA2 2-bit input is currently used for selecting the read data register for ST operation.

Control CPU:

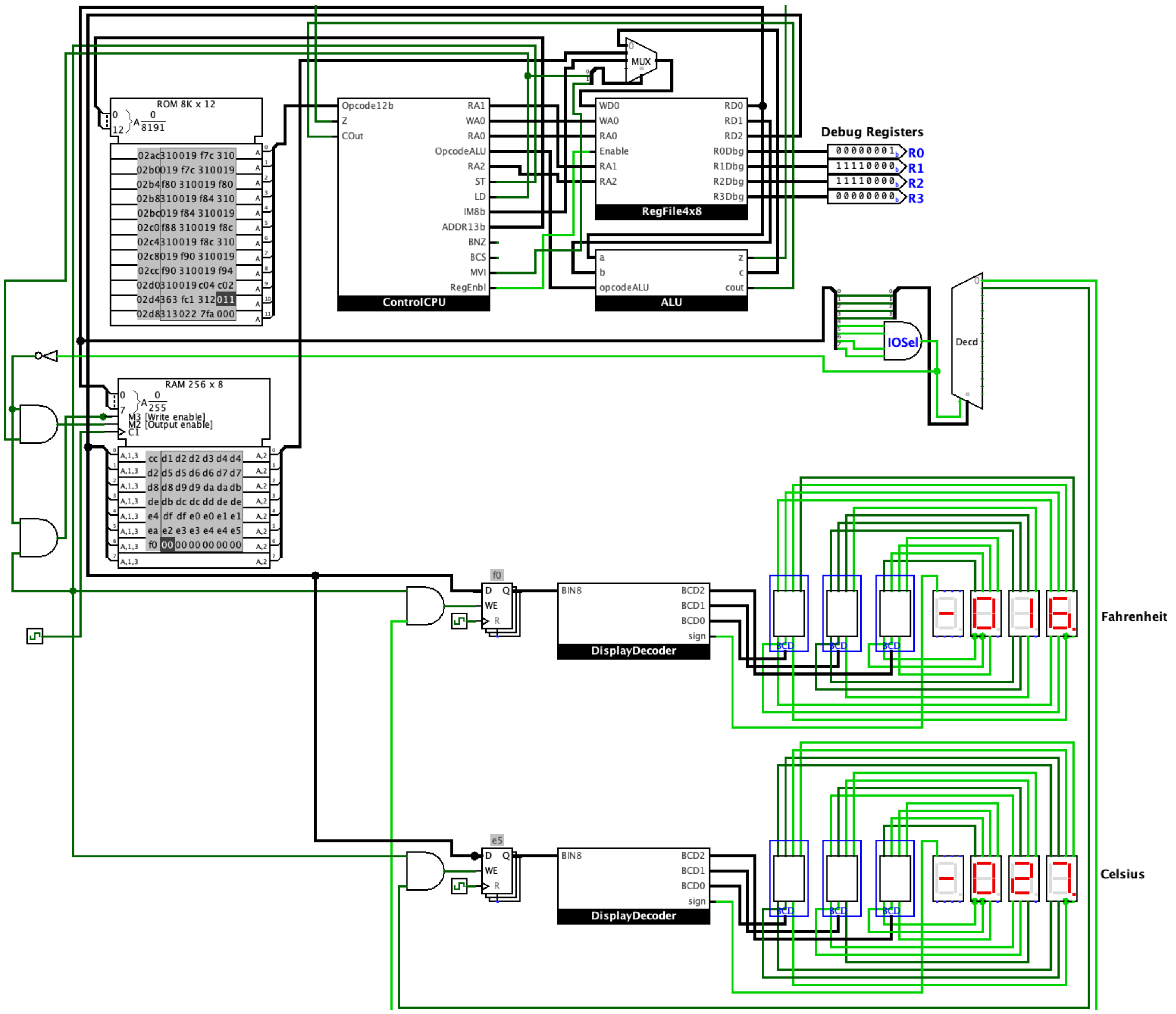
Generates all the control signals for the register file, ALU opcode, and data for immediate load instruction (MVI). The instruction opcode is used by the Control CPU to also decode and output signals for certain instructions. (MVI, ST, LD, etc.) It also implements the program counter which uses a multiplexer and a small combinatorial circuit to perform relative jumps as shown in class. Control CPU also generates a RegEnbl signal which allows writing in RegFile.

Full Circuit:

Control CPU starts generating 12-bit addresses from address 0. Instructions are fetched from ROM and decoded by Control CPU. Registers are selected by the RA0-2, WA0 signals, and instructions are executed based on their opcode and ALU opcode if applicable. There is an I/O mapping to memory addresses 0xF0 and 0xF1 implemented with an I/O select and a 4-bit decoder. Reads/Writes to RAM 0xF0 and 0xF1 are directed to two registers which are each connected to 4-digit decimal displays. There is also logic to select the 256x8bit RAM chip when executing ST/LD instructions. The write source (ALU, RAM, Immediate op) for RegFile is selected by a MUX based on MVI/LD signals. The circuit was simulated up to 2MHz clock frequency on both programs.

An assembler program was written in Java which was used to generate the machine code for the CPU. Two programs were written, one that populates a conversion table in RAM for 240 Fahrenheit values 0->127 and -128->-16, and another program that uses an approximation formula C = F/2 - 15.

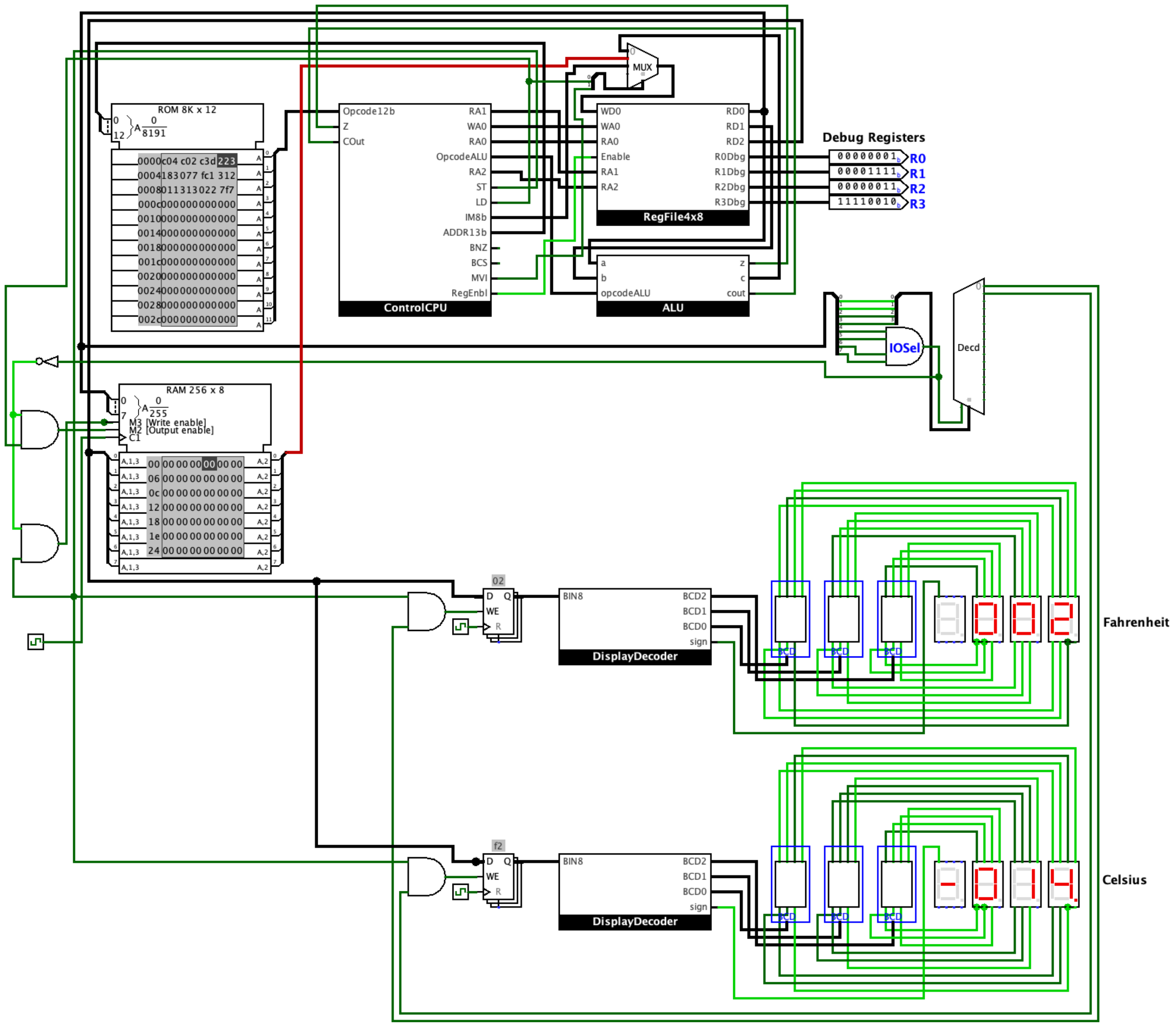
CPU Main circuit running tablef2.s (Assignment Part A):



```
02ac 310019 f7c 310
02b0 019 f7c 310019
02b4 f80 310019 f80
02b8 310019 f84 310
02bc 019 f84 310019
02c0 f88 310019 f8c
02c4 310019 f8c 310
02c8 019 f90 310019
02cc f90 310019 f94
02d0 310019 c04 c02
02d4 363 fc1 312 011
02d8 313022 7fa 000
```

```
cc d1 d2 d2 d3 d4 d4
d2 d5 d5 d6 d6 d7 d7
d8 d8 d9 d9 da da db
de db dc dc dd de de
e4 df df e0 e0 e1 e1
ea e2 e3 e3 e4 e4 e5
f0 00 00 00 00 00 00
```

CPU Main circuit running formulaf2.s (Assignment Part B):



```
0000 c04 c02 c3d 223
0004 183 077 fc1 312
0008 011 313 022 7f7
000c 000 000 000 000
0010 000 000 000 000
0014 000 000 000 000
0018 000 000 000 000
001c 000 000 000 000
0020 000 000 000 000
0024 000 000 000 000
0028 000 000 000 000
002c 000 000 000 000
```

```
00 00 00 00 00 00 00
06 00 00 00 00 00 00
0c 00 00 00 00 00 00
12 00 00 00 00 00 00
18 00 00 00 00 00 00
1e 00 00 00 00 00 00
24 00 00 00 00 00 00
```

```
00000001 R0
00001111 R1
00000011 R2
11110010 R3
```

Part A Assembler program (building and using RAM table to convert):

tablef2.s

```
; Load initial RAM address 0 in R1
        MVI R1 0        ; 0xC01
; Load 1 into R2 for increment
        MVI R2 1        ; 0xC06
; Move immediate celsius value in R0
        MVI R0 0xEE     ; 0xFB8
; Store celsius value in RAM at address in R1
        ST R0 [R1]      ; 0x310
; Increment register containing RAM address
        ADD R1 R1 R2    ; 0x019
; Repeat move from table to memory
        MVI R0 0xEF     ; 0xFBC
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xEF     ; 0xFBC
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xEF     ; 0xFBC
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xF0     ; 0xFC0
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xF0     ; 0xFC0
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xF1     ; 0xFC4
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xF2     ; 0xFC8
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xF2     ; 0xFC8
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xF3     ; 0xFCC
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xF3     ; 0xFC8
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019
```

```
MVI R0 0xF4     ; 0xFD0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xF4     ; 0xFD0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xF5     ; 0xFD4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xF5     ; 0xFD4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xF6     ; 0xFD8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xF7     ; 0xFDC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xF7     ; 0xFDC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xF8     ; 0xFE0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xF8     ; 0xFE0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xF9     ; 0xFE4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xF9     ; 0xFE4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xFA     ; 0xFE8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xFA     ; 0xFE8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xFB     ; 0xFEC
```

```
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xFC     ; 0xFF0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xFC     ; 0xFF0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xFD     ; 0xFF4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xFD     ; 0xFF4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xFE     ; 0xFD8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xFE     ; 0xFD8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xFF     ; 0xFFC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xFF     ; 0xFFC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x00     ; 0xC00
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x01     ; 0xC04
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x01     ; 0xC04
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x02     ; 0xC08
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x02     ; 0xC08
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019
```

```
MVI R0 0x03     ; 0xC0C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x03     ; 0xC0C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x04     ; 0xC10
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x04     ; 0xC10
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x05     ; 0xC14
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x06     ; 0xC18
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x06     ; 0xC18
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x07     ; 0xC1C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x07     ; 0xC1C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x08     ; 0xC20
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x08     ; 0xC20
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x09     ; 0xC24
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x09     ; 0xC24
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x0A     ; 0xC28
```

```
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x0B     ; 0xC2C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x0B     ; 0xC2C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x0C     ; 0xC30
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x0C     ; 0xC30
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x0D     ; 0xC34
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x0D     ; 0xC34
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x0E     ; 0xC38
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x0E     ; 0xC38
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x0F     ; 0xC3C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x10     ; 0xC40
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x10     ; 0xC40
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x11     ; 0xC44
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x11     ; 0xC44
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019
```

```
MVI R0 0x12     ; 0xC48
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x12     ; 0xC48
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x13     ; 0xC4C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x13     ; 0xC4C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x14     ; 0xC50
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x15     ; 0xC54
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x15     ; 0xC54
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x16     ; 0xC58
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x16     ; 0xC58
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x17     ; 0xC5C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x17     ; 0xC5C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x18     ; 0xC60
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x18     ; 0xC60
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x19     ; 0xC64
```

```
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x1A     ; 0xC68
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x1A     ; 0xC68
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x1B     ; 0xC6C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x1B     ; 0xC6C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x1C     ; 0xC70
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x1C     ; 0xC70
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x1D     ; 0xC74
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x1D     ; 0xC74
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x1E     ; 0xC78
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x1F     ; 0xC7C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x1F     ; 0xC7C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x20     ; 0xC80
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x20     ; 0xC80
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019
```

```
MVI R0 0x21     ; 0xC84
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x21     ; 0xC84
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x22     ; 0xC88
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x22     ; 0xC88
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x23     ; 0xC8C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x24     ; 0xC90
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x24     ; 0xC90
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x25     ; 0xC94
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x25     ; 0xC94
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x26     ; 0xC98
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x26     ; 0xC98
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x27     ; 0xC9C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x27     ; 0xC9C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x28     ; 0xCA0
```

```
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x29     ; 0xCA4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x29     ; 0xCA4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x2A     ; 0xCA8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x2A     ; 0xCA8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x2B     ; 0xCAC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x2B     ; 0xCAC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x2C     ; 0xCB0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x2C     ; 0xCB0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x2D     ; 0xCB4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x2E     ; 0xCB8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x2E     ; 0xCB8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x2F     ; 0xCBC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0x2F     ; 0xCBC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019
```

```
        MVI R0 0x30     ; 0xCC0
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0x30     ; 0xCC0
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0x31     ; 0xCC4
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0x31     ; 0xCC4
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0x32     ; 0xCC8
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0x33     ; 0xCCC
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0x33     ; 0xCCC
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0x34     ; 0xCD0
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0x34     ; 0xCD0
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0x35     ; 0xCD4
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

; After this address fahrenheit
; becomes negative
        MVI R0 0xA7     ; 0xE9C
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xA8     ; 0xEA0
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xA8     ; 0xEA0
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019
```

```
MVI R0 0xA9     ; 0xEA4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xA9     ; 0xEA4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xAA     ; 0xEA8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xAA     ; 0xEA8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xAB     ; 0xEAC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xAC     ; 0xEB0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xAC     ; 0xEB0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xAD     ; 0xEB4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xAD     ; 0xEB4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xAE     ; 0xEB8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xAE     ; 0xEB8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xAF     ; 0xEBC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xAF     ; 0xEBC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB0     ; 0xEC0
```

```
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB1     ; 0xEC4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB1     ; 0xEC4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB2     ; 0xEC8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB2     ; 0xEC8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB3     ; 0xECC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB3     ; 0xECC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB4     ; 0xED0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB4     ; 0xED0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB5     ; 0xED4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB6     ; 0xED8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB6     ; 0xED8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB7     ; 0xEDC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB7     ; 0xEDC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019
```

```
MVI R0 0xB8     ; 0xEE0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB8     ; 0xEE0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB9     ; 0xEE4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xB9     ; 0xEE4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xBA     ; 0xEE8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xBB     ; 0xEEC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xBB     ; 0xEEC
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xBC     ; 0xEF0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xBC     ; 0xEF0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xBD     ; 0xEF0
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xBD     ; 0xEF4
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xBE     ; 0xEF8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xBE     ; 0xEF8
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xBF     ; 0xEFC
```

```
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC0     ; 0xF00
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC0     ; 0xF00
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC1     ; 0xF04
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC1     ; 0xF04
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC2     ; 0xF08
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC2     ; 0xF08
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC3     ; 0xF0C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC3     ; 0xF0C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC4     ; 0xF10
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC5     ; 0xF10
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC5     ; 0xF14
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC6     ; 0xF18
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC6     ; 0xF18
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019
```

```
MVI R0 0xC7     ; 0xF1C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC7     ; 0xF1C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC8     ; 0xF20
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC8     ; 0xF20
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xC9     ; 0xF24
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xCA     ; 0xF28
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xCA     ; 0xF28
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xCB     ; 0xF2C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xCB     ; 0xF2C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xCC     ; 0xF30
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xCC     ; 0xF30
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xCD     ; 0xF34
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xCD     ; 0xF34
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xCE     ; 0xF38
```

```
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xCF     ; 0xF3C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xCF     ; 0xF3C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD0     ; 0xF40
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD1     ; 0xF44
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD1     ; 0xF44
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD2     ; 0xF48
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD2     ; 0xF48
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD3     ; 0xF4C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD4     ; 0xF50
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD4     ; 0xF50
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD5     ; 0xF54
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD5     ; 0xF54
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD6     ; 0xF58
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019
```

```
MVI R0 0xD6     ; 0xF58
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD7     ; 0xF5C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD7     ; 0xF5C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD8     ; 0xF60
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD9     ; 0xF64
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xD9     ; 0xF64
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xDA     ; 0xF68
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xDA     ; 0xF68
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xDB     ; 0xF6C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xDB     ; 0xF6C
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xDC     ; 0xF70
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xDC     ; 0xF70
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xDD     ; 0xF74
ST R0 [R1]      ; 0x310
ADD R1 R1 R2    ; 0x019

MVI R0 0xDE     ; 0xF78
```

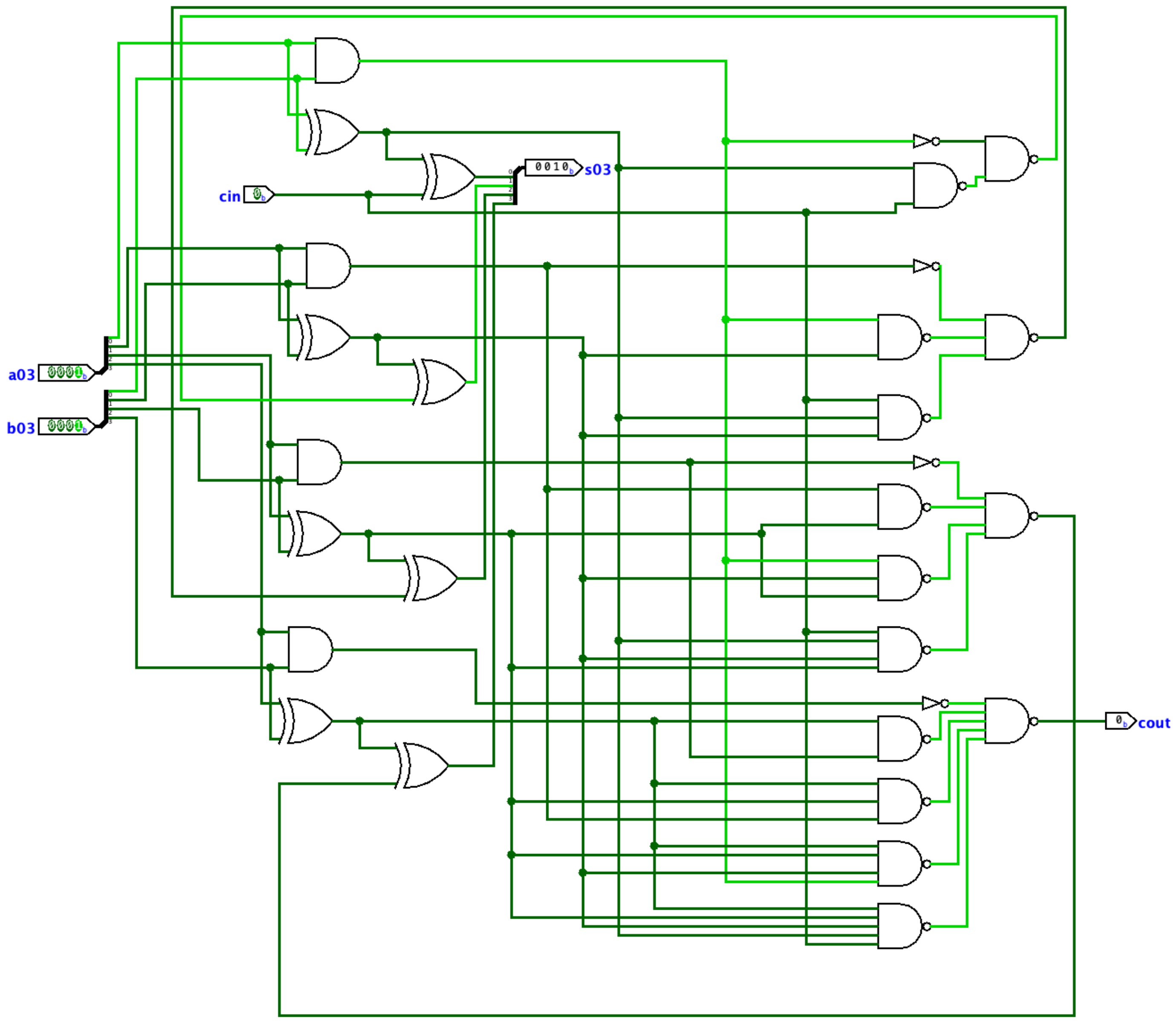
```
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xDE     ; 0xF78
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xDF     ; 0xF7C
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xDF     ; 0xF7C
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xE0     ; 0xF80
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xE0     ; 0xF80
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xE1     ; 0xF84
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xE1     ; 0xF84
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xE2     ; 0xF88
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xE3     ; 0xF8C
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xE3     ; 0xF8C
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xE4     ; 0xF90
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

        MVI R0 0xE4     ; 0xF90
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

; This is the last valid RAM address EF
        MVI R0 0xE5     ; 0xF94
        ST R0 [R1]      ; 0x310
        ADD R1 R1 R2    ; 0x019

; Reached address F0 stopping.
; The next 16 RAM addresses are inaccessible

; Now the table is loaded in RAM
; Iterate through RAM addresses and
; display the conversion
; Load 1 in R0 and 0 in R2
        MVI R0 0x1      ; 0xC04
        MVI R2 0x00     ; 0xC02
        LD R3 [R2]      ; 0x363
; Send fahrenheit to port address 0xF0
        MVI R1 0xF0     ; 0xFC1
        ST R2 [R1]      ; 0x312
; Increment port address
        ADD R1 R1 R0    ; 0x011
; Send Celsius to port in R1
        ST R3 [R1]      ; 0x313
; Increment address of Fahrenheit value
        ADD R2 R2 R0    ; 0x022
; Jump to load Celsius -6 instructions
        BNZ -6          ; 0x7FA
```

Part B Assembler program (using formula to convert):

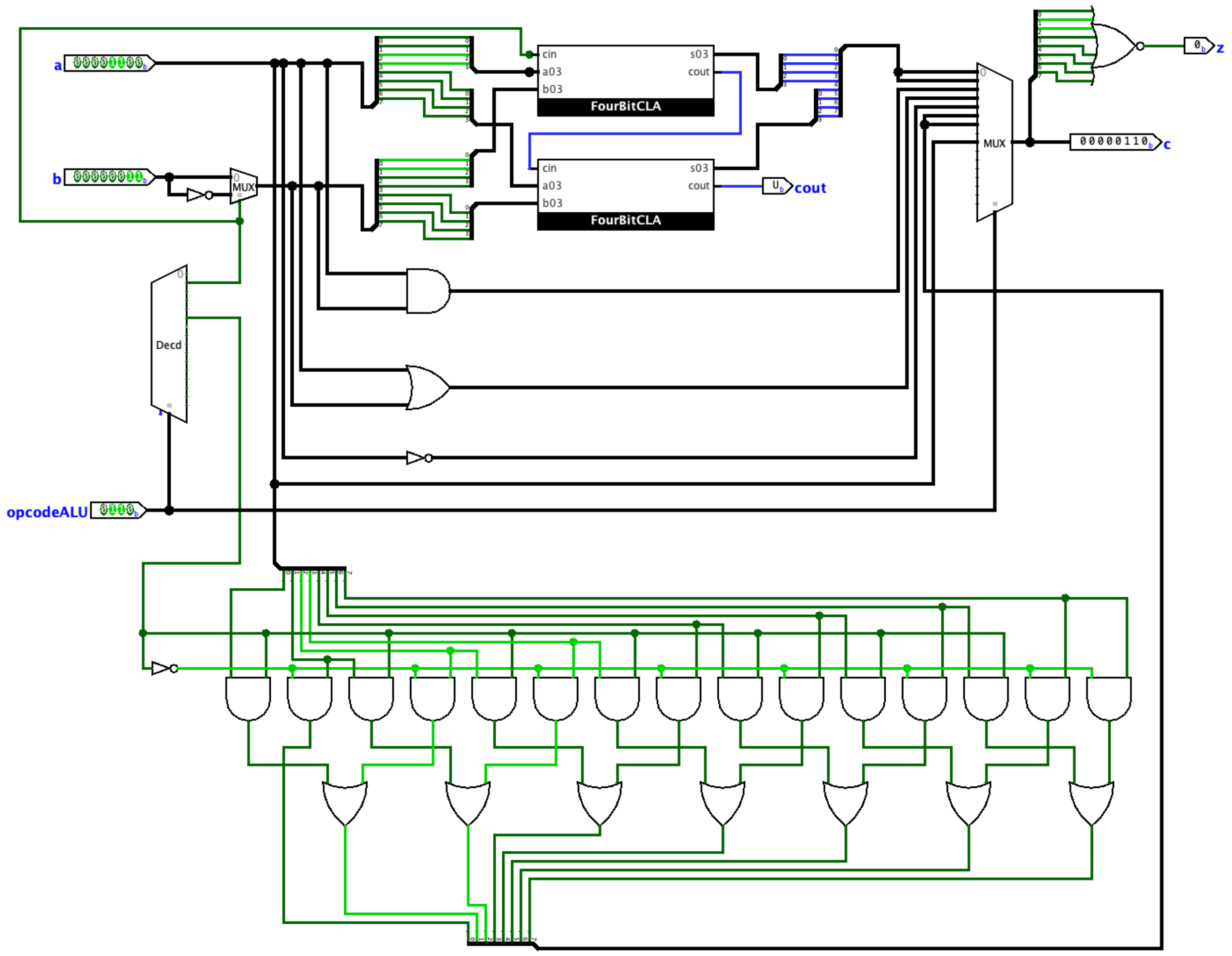
formulaf2.s

```
; Load 1 in R0 and 0 in R2
        MVI R0 0x1      ; 0xC04
; Load 0 degrees Fahrenheit in R2
        MVI R2 0x0      ; 0xC02
; Loop begin
; Using C = F/2 - 15
        MVI R1 0xF      ; 0xC3D
; Fahrenheit kept in R2
; Celsius computed in R3
        MOV R3 R2       ; 0x223
        ASR R3          ; 0x183
        SUB R3 R3 R1    ; 0x077
; Load first port address in R1
; Send Fahrenheit to port R1
        MVI R1 0xF0     ; 0xFC1
        ST R2 [R1]      ; 0x312
; Increment port address
        ADD R1 R1 R0    ; 0x011
; Send Celsius to port in R1
        ST R3 [R1]      ; 0x313
        ADD R2 R2 R0    ; 0x022
; Jump to Loop: 9 instructions back
        BNZ -9;         ; 0x7F7
```

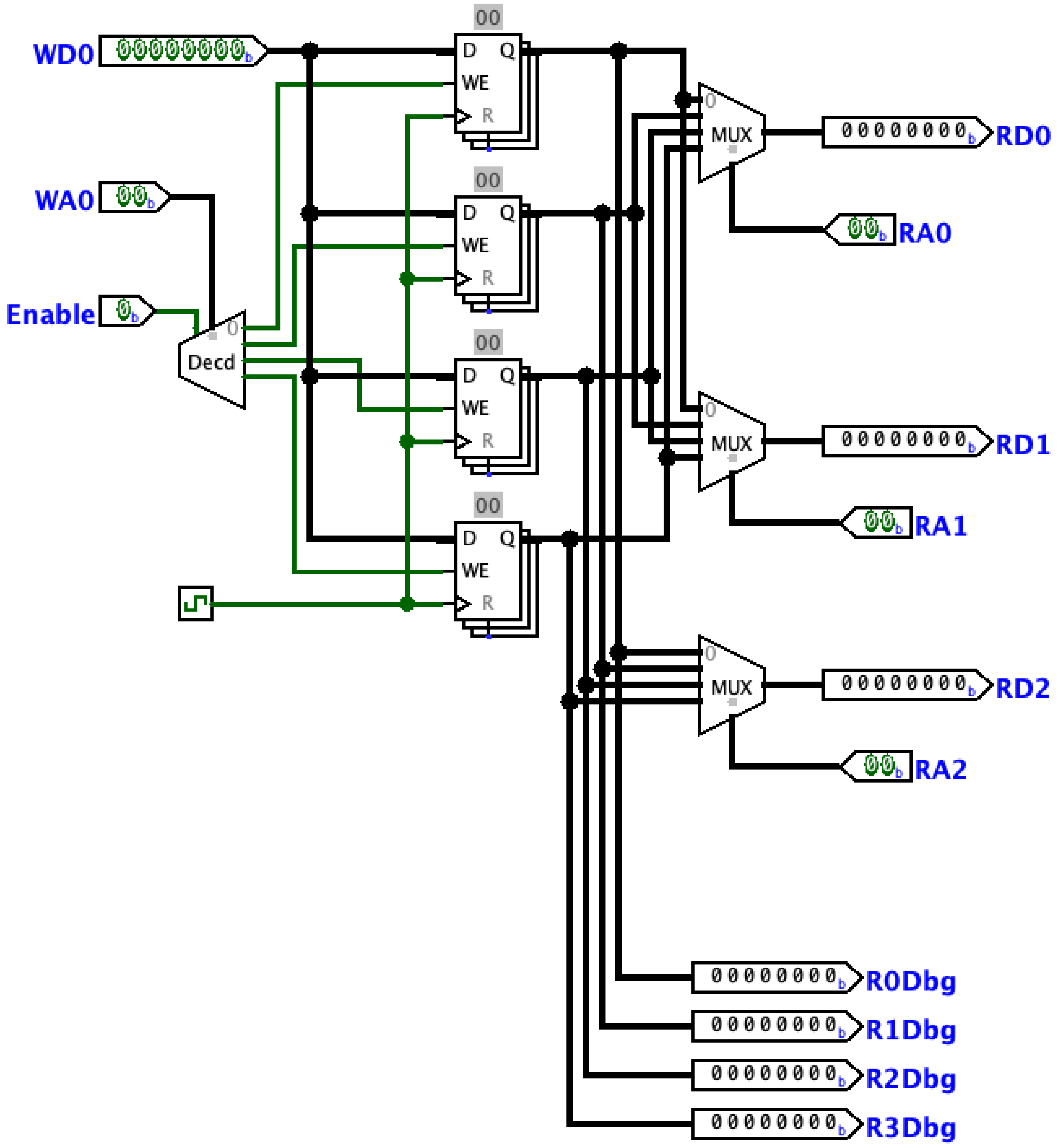
Subcircuits: Four bit CLA



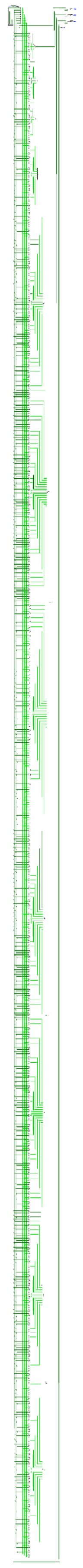
Subcircuits: ALU



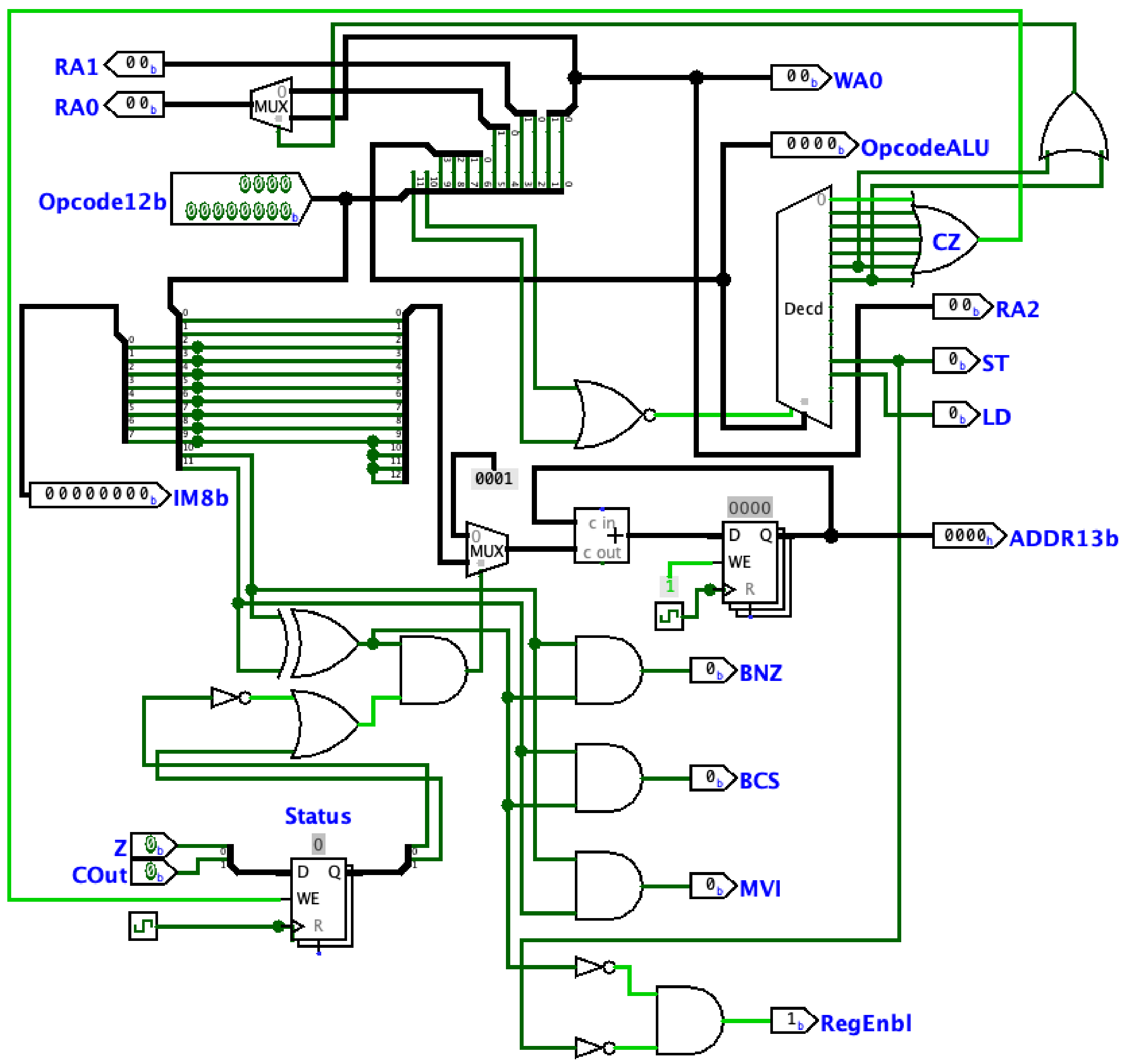
Subcircuits: RegFile 4x8bit registers



Subcircuits: Display Decoder (Image is very large)



Subcircuits: Control CPU



Annex 1

MISCAS Assembler

```
// Student Name: Gabriele Nicula
// Student ID: 219969192
// Lab 3 :: CPU Lab 2

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

public class miscas {

  public final int OPMASK = 0x00000FFF;

  public interface Instruction {
    public int toMachineCode();
  }

  // Base class for 3 operand instructions.
  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 int toMachineCode() {
      int opBits = opcode << 6;
      int rnBits = rn << 4;
      int rmBits = rm << 2;
      int rdBits = rd;
      int machineCode = opBits | rnBits | rmBits | rdBits;
```

```java
    return machineCode & OPMASK;
  }

  @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 SUB_Instruction extends ThreeOpInstruction {

  public SUB_Instruction(int rd, int rn, int rm) {
    super("sub", 0x1, rd, rn, rm);
  }

}

public class AND_Instruction extends ThreeOpInstruction {

  public AND_Instruction(int rd, int rn, int rm) {
    super("and", 0x2, rd, rn, rm);
  }

}

public class OR_Instruction extends ThreeOpInstruction {

  public OR_Instruction(int rd, int rn, int rm) {
    super("or", 0x3, 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 = 0x04;
  }
  public int toMachineCode() {
    int opBits = opcode << 6;
    int rnBits = rn << 4;
    int machineCode = opBits | rnBits | rd;
    return machineCode & OPMASK;
  }

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

public class SHL_Instruction implements Instruction {

  private final int opcode;
  private final int rd;

  public SHL_Instruction(int rd) {
    this.rd = rd;
    opcode = 0x05;
  }

  public int toMachineCode() {
    int opBits = opcode << 6;
    int machineCode = opBits | rd;
    return machineCode & OPMASK;
  }

  @Override
  public String toString() {
    return "shl r" + rd;
  }
}

public class ASR_Instruction implements Instruction {

  private final int opcode;
  private final int rd;

  public ASR_Instruction(int rd) {
    this.rd = rd;
    opcode = 0x06;
  }

  public int toMachineCode() {
    int opBits = opcode << 6;
    int machineCode = opBits | rd;
    return machineCode & OPMASK;
  }

  @Override
  public String toString() {
    return "asr r" + rd;
  }
}

public class MOV_Instruction implements Instruction {

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

  public MOV_Instruction(int rd, int rn) {
    this.rd = rd;
    this.rn = rn;
    opcode = 0x08;
  }

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

  @Override
  public String toString() {
    return "mov r" + rd + " r" + rn;
  }
}

public class ST_Instruction implements Instruction {

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

  public ST_Instruction(int rd, int rn) {
    this.rd = rd;
    this.rn = rn;
    opcode = 0x0C;
    // System.out.println("ST rn: " + rn);
  }

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

  @Override
  public String toString() {
    return "st r" + rd + " [r" + rn + "]";
  }
}
```

```
public class LD_Instruction implements Instruction {

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

  public LD_Instruction(int rd, int rn) {
    this.rd = rd;
    this.rn = rn;
    opcode = 0x0D;
  }

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

  @Override
  public String toString() {
    return "ld r" + rd + " [r" + rn + "]";
  }
}

public class BNZ_Instruction implements Instruction {
  private final int opcode;
  private final int offset;

  public BNZ_Instruction(int offset) {
    this.offset = offset;
    opcode = 0x10;
  }

  @Override
  public int toMachineCode() {
    int opBits = opcode << 6;
    int offBits = offset & 0x000003FF;
    int machineCode = opBits | offBits;
    return machineCode & OPMASK;
  }

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

public class BCS_Instruction implements Instruction {
  private final int opcode;
  private final int offset;

  public BCS_Instruction(int offset) {
    this.offset = offset & 0x000003FF;
    opcode = 0x20;
  }

  @Override
  public int toMachineCode() {
    int opBits = opcode << 6;
    int offBits = offset & 0x000003FF;
    int machineCode = opBits | offBits;
    return machineCode & OPMASK;
  }

  @Override
  public String toString() {
    return "bcs " + offset;
  }
}

public class MVI_Instruction implements Instruction {

  private final int opcode;
  private final int rd;
  private final int value;

  public MVI_Instruction(int rd, int value) {
    this.rd = rd;
    this.value = value & 0xFF;
    opcode = 0x30;
  }

  public int toMachineCode() {
    int opBits = opcode << 6;
    int valueBits = value << 2;
    int machineCode = opBits | valueBits | rd;
    return machineCode & OPMASK;
  }

  @Override
  public String toString() {
    return "mvi r" + rd + " " + value;
  }
}

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

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

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

public void parse() throws IOException {

  try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
    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();
      }

      // Parse instruction
      boolean isCorrectInstruction = true;
      if (!line.isEmpty()) {
        String[] tokens = line.split("\\s+");
        String opcode = tokens[0].toLowerCase();
        if (opcode.equals("add") || opcode.equals("sub")
            || opcode.equals("and") || opcode.equals("or")
            || opcode.equals("not") || opcode.equals("shl")
            || opcode.equals("asr") || opcode.equals("mov")
            || opcode.equals("st") || opcode.equals("ld")
            || opcode.equals("bnz") || opcode.equals("bcs")
            || opcode.equals("mvi")) {
          boolean checkRegCount = true;
          boolean checkRegNames = true;
          switch (opcode) {
            case "add":
            case "sub":
            case "and":
            case "or":
              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":
case "mov":
  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 "shl":
case "asr":
  checkRegCount = tokens.length == 2;
  if (!checkRegCount) {
    System.err
        .println("Invalid number of registers for instruction <"
            + tokens[0] + "> on line <" + lineNo + ">");
    ++numParseErrors;
    isCorrectInstruction = false;
  } else {
    checkRegNames = checkRegisterNames(tokens, 1);
    if (!checkRegNames) {
      System.err.println(" on line <" + lineNo + ">");
      ++numParseErrors;
      isCorrectInstruction = false;
    }
  }
  break;
case "st":
case "ld":
  checkRegCount = tokens.length == 3;
  if (!checkRegCount) {
    System.err
        .println("Invalid number of registers for instruction <"
            + tokens[0] + "> on line <" + lineNo + ">");
    ++numParseErrors;
    isCorrectInstruction = false;
  } else {
    // check and remove square brackets.
    if (tokens[2].charAt(0) != '['
        || tokens[2].charAt(tokens[2].length() - 1) != ']') {
      System.err
          .println("Invalid address register for instruction <"
              + tokens[0] + "> on line <" + lineNo + ">");
      ++numParseErrors;
      isCorrectInstruction = false;
```

```
              } else {
                tokens[2] = tokens[2].substring(1,
                  tokens[2].length() - 1);
                checkRegNames = checkRegisterNames(tokens, 1);
                if (!checkRegNames) {
                  System.err.println(" on line <" + lineNo + ">");
                  ++numParseErrors;
                  isCorrectInstruction = false;
                }
              }
            }
            break;
          case "bnz":
          case "bcs":
            boolean checkLength = tokens.length == 2;
            if (!checkLength) {
              System.err.println(
                  "Invalid argument count for bnz instruction on line <"
                      + lineNo + ">");
              ++numParseErrors;
              isCorrectInstruction = false;
            }
            break;
          case "mvi":
            if (tokens.length != 3) {
              System.err.println(
                  "Invalid argument count for bnz instruction on line <"
                      + lineNo + ">");
              ++numParseErrors;
              isCorrectInstruction = false;
            }
            break;
        }
        if (isCorrectInstruction) {
          line = line.replaceAll("\\[", "")
            .replaceAll("\\]", "");
          // System.out.println("Adding line: " + line);
          instructionStatements.put(lineNo, line);
        }
      } else {
        System.err.println("Invalid opcode: <" + opcode + "> on line <"
            + lineNo + ">");
        ++numParseErrors;
      }
    }
  }
}
// System.out.println("Instruction statements size " +
// instructionStatements.size());
// Now generate machine code based on the parsed instructions.
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 "sub":
    instructions.add(new SUB_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 "or":
    instructions.add(new OR_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 "shl":
    instructions.add(new SHL_Instruction(parseRegister(tokens[1])));
    break;
  case "asr":
    instructions.add(new ASR_Instruction(parseRegister(tokens[1])));
    break;
  case "mov":
    instructions.add(new MOV_Instruction(parseRegister(tokens[1]),
        parseRegister(tokens[2])));
    break;
  case "st":
    instructions.add(new ST_Instruction(parseRegister(tokens[1]),
        parseRegister(tokens[2])));
    break;
  case "ld":
    instructions.add(new LD_Instruction(parseRegister(tokens[1]),
        parseRegister(tokens[2])));
    break;
  case "bnz":
    instructions.add(new BNZ_Instruction(Integer.decode(tokens[1])));
    break;
  case "bcs":
    instructions.add(new BCS_Instruction(Integer.decode(tokens[1])));
    break;
  case "mvi":
    instructions.add(new MVI_Instruction(parseRegister(tokens[1]),
        Integer.decode(tokens[2])));
    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() > 8192) {
      System.err.println(
          "Too many instructions: object file is larger than 64 byte " +
              "system memory.");
      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("%03X", 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;
  }

} // END AssemblyParser

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

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
  if (args.length < 2 || args.length > 3) {
    System.out.println("USAGE: java miscas <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 printListing = false;
  if (args.length == 3 && args[2].equals("-l")) {
    printListing = true;
  }

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

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

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

  // Print the assembled source if requested.
  if (printListing) {
```

```
            printMachineCode(assembler.getInstructions());
        }
    }
}
```