Computer Architecture Lab 03

Yihang Qiu, 520030910155, 2022/01

The Control Logic of "lc3sim.c"

For the part that I completed in "`lc3sim.c`", the main core is function `process_instruction()`. The function processes one instruction at a time, simulating the instruction process cycle.

First, we fetch the instruction, **i.e.** assign `MEMORY[PC]` to `IR` (Instruction Registers). Meanwhile, increment `PC` in the current latches.

Secondly, we decode the instruction and execute. The opcode is `IR[15:12]`, which is stored in `IST_CODE` in my code. Considering most instructions use `IR[11:9]` and `IR[8:6]` as operands, here we calculate them in advance. Call execution functions corresponding to the opcode `IST_CODE`.

Here we update `NEXT_LATCHES` in the execution phase. Note that `NEXT_LATCHES` derives from `CURRENT_LATCHES` after `CURRENT_LATCHES.PC` is incremented.

The Pseudocode is given below.

Pseudocode for `process_instruction()` in LC-3 Simulation

```
1:  procedure process_instruction()
2:      IR ← MEMORY [PC]
3:      PC ← PC+1                          // fetch the instruction
4:
5:      IST_CODE ← IR[15:12]               // decode the instruction
6:      DST ← IR[11:9]
7:      SRC ← IR[8:6]              // DST and SRC are not used by all the instructions
8:
9:      NEXT_LATCHES ← CURRENT_LATCHES
10:     Switch (IST_CODE)
11:         Case opcode: Call the corresponding function.
12:                      Pass DST and SRC into the called function if they are operands.
13:                                        // execution
14:                                        // update the latch in these functions
15:     End Switch
16: End Procedure
```

Explanation of Some Important Functions

Before we move on to explain some of the functions in my version of "`lc3sim.c`", it is necessary to mention two ways of data storage we used in the simulation.

For `PC`, `IR`, `IST_CODE`(opcode), it is obvious that we need to use the actual value, **i.e.** the value stored in these variables or results related to them are the real value, or you may view them as sign-magnitude representations. Also, these values are non-negative.

But for the value stored in `MEMORY` and `REGS`(registers), we simulate the 2's complement, **i.e.** the value stored is actually the value of 16-bit 2's complement of the real value under type **`int`** in C language.

In the following part, what some critical functions can do are explained.

- **`IST_toBin(instruction)`**

In my version of **`"lc3sim.c"`**, a specific function `IST_toBin(instruction)` is used to convert the instruction stored in `IR` (which should be a positive integer) into binary sequence. The binary sequence is then stored in the array `IR_BIN[]`, which is a global variable.

Some functions are created to complete some operations that are required by several instructions. These functions are listed as follows.

`1) set_cc(result)`

The function sets condition codes (CC), **i.e.** `NEXT_LATCHES.N`, `NEXT_LATCHES.Z`, `NEXT_LATCHES.P`. We know only `ADD`, `AND`, `NOT`, `LD`, `LDI` and `LDR` instructions will change the value of CC. Note that the results of these instructions, **i.e.** the argument `result`, are all stored in the way simulating 2's complements as we mentioned above.

Thus, when `result`$> 32767 = (0111\ 1111\ 1111\ 1111)_2$, the binary of `result` is actually the 2's complement of a negative number. Thus, set `NEXT_LATCHES.N` to 1.

The other two cases are self-evident. When `result`$= 0$, set `NEXT_LATCHES.Z` to 1. Otherwise, set `NEXT_LATCHES.P` to 1.

`2) sext(bits)` and **`sext_offset(bits)`**

Both of the two functions apply sign-bit extension to `IR_BIN[(bits-1):0]`. The only difference is that the result of `sext(bits)` is in the 2's complement form while the result of `sext_offset(bits)` is in the sign-magnitude form and is for sure non-negative.

Functions created to simulate the execution phase are listed below.

`add_(dst,src)`;	`and_(dst,src)`;	`branch_()`;	`jmp_(src)`;
`jsr_(src)`;	`ld_(dst)`;	`ldi_(dst)`;	`ldr_(dst,src)`;
`lea_(dst)`;	`not_(dst,src)`;	`st_(dst)`;	`sti_(dst)`;
	`str_(dst,src)`;	`trap_()`.	

`1) add_(dst,src)`, **`and_(dst,src)`**, and **`not_(dst,src)`**

To maintain the data stored in `MEMORY` and `REGS` to be 16-bit 2's complements, after we calculate the result, we apply `Low16bits()` to it. Moreover, at the end of these functions, we call `set_cc(result)` to update the condition codes.

Moreover, for `add_()` and `and_()`, we discuss whether `IR_BIN[5]` is 0 or 1, which leads to two different operations, **i.e.** "R1+R2" (or "R1 and R2") and "R1+IMM5" (or "R1 and IMM5").

`2) ld_(dst)`, **`ldi_(dst)`**, and **`ldr_(dst,src)`**

At the end of these functions, we call `set_cc(result)` to update the condition codes.

`3) trap_()`

Here we use `sext_offset(bits)` to do the zero-extension. Since for `TRAP` instructions, it is guaranteed that `IR_BIN[8]`$= 0$. Thus, `zext(IR_BIN[7:0])`$=$`sext_offset(IR_BIN[8:0])`.

Verification and Testing of “lc3sim.c”

First, we write a program derived from the following `test1.asm` by the following Linux command

```
lc3as test1.asm
hexdump -v -e ' "0x" 2/1 "%02X" "\n" ' test1.obj | tee test1
```

to test the function of all feasible instructions. Run one instruction at a time.

`test1.asm`

```
                   .ORIG    x3000
x3000              LD   R0, A
x3001              LDI  R1, B
x3002              JMP  R0
x3003   A          .FILL x3006
x3004   B          .FILL x3003
x3005   C          .FILL 30
x3006              LDR  R2, R1, #-1
x3007              ADD  R3, R1, R2
x3008              AND  R5, R0, #10
x3009              LEA  R6, #25
x300a              JSR  RTINE
x300b              NOT  R5, R5
x300c              AND  R7, R7, #0
x300d              BRnp ELSE1
x300e              ADD  R3, R3, #1
x300f   ELSE1      AND  R2, R2, #15
x3010              BRzp ELSE2
x3011              ADD  R3, R3, #1
x3012   ELSE2      ST   R7, A
x3013              LD   R5, A
x3014              STI  R2, B
x3015              LD   R6, A
x3016              STR  R3, R1, #-1
x3017              LD   R7, C
x3018              HALT
x3019   RTINE      ADD  R3, R3, #1
x301a              RET
x301b              .END
```

The testing is as follows.

(**Note:** Lines in **red** are commands inputted. Words in blue are comments. To save the space, we deleted some empty lines.)

```
$ ./simulate test1
LC-3 Simulator
Read 27 words from program into memory.
LC-3-SIM> run 3
Simulating for 3 cycles...
LC-3-SIM> rdump
Current register/bus values :
-------------------------------------
Instruction Count : 3
PC                : 0x3006          // JMP tested.
CCs: N = 0  Z = 0  P = 1
Registers:
0: 0x3006  // LD tested. A is loaded into R0.
1: 0x3006  // LDI tested. MEM[x3003] is loaded into R1.
2: 0x0000
3: 0x0000
4: 0x0000
5: 0x0000
6: 0x0000
7: 0x0000

LC-3-SIM> run 5
Simulating for 5 cycles...
LC-3-SIM> rdump
Current register/bus values :
-------------------------------------
Instruction Count : 8
PC                : 0x3019          // JSR tested.
CCs: N = 0  Z = 0  P = 1
Registers:
0: 0x3006
1: 0x3006
2: 0x001e  // LDR tested. C is loaded into R2. (30=0x1e)
3: 0x3024  // ADD tested. R3=R1+R2.
4: 0x0000
5: 0x0002  // AND tested. A is loaded into R0.
6: 0x3023  // LEA tested. R6=PC (0x3010)+25 (0x0019).
7: 0x300b  // JSR tested. R7←PC at that time.

LC-3-SIM> run 2
Simulating for 1 cycles...
LC-3-SIM> rdump
Current register/bus values :
-------------------------------------
Instruction Count : 10
PC                : 0x300b  // RET (JMP R7) tested.
CCs: N = 0  Z = 0  P = 1
Registers:
0: 0x3006
1: 0x3006
2: 0x001e
3: 0x3025
4: 0x0000
5: 0x0002
6: 0x3023
7: 0x300b
```

```
LC-3-SIM> run 2
Simulating for 5 cycles...
LC-3-SIM> rdump
Current register/bus values :
-------------------------------------
Instruction Count : 12
PC                : 0x300d
CCs: N = 0  Z = 1  P = 0 // Condition Code tested
Registers:
0: 0x3006
1: 0x3006
2: 0x001e
3: 0x3025
4: 0x0000
5: 0xfffd // NOT tested. R5←NOT(0x0002)=0xfffd
6: 0x3023
7: 0x0000

LC-3-SIM> run 3
Simulating for 3 cycles...
LC-3-SIM> rdump
Current register/bus values :
-------------------------------------
Instruction Count : 15
PC                : 0x3010
CCs: N = 0  Z = 0  P = 1
Registers:
0: 0x3006
1: 0x3006
2: 0x000e
3: 0x3026 // BRnp tested. (R3++)
4: 0x0000
5: 0xfffd
6: 0x3023
7: 0x0000

LC-3-SIM> go
Simulating...
Simulator halted        // shell - go tested
LC-3-SIM> rdump
Current register/bus values :
-------------------------------------
Instruction Count : 23
PC                : 0x0000 // TRAP tested
CCs: N = 0  Z = 0  P = 1
Registers:
0: 0x3006
1: 0x3006
2: 0x000e
3: 0x3026
4: 0x0000
5: 0x0000 // ST tested. (R5←A←R7=0)
6: 0x000e // STI tested. (R6←A=MEM[B]←R2=x000e)
7: 0x3026 // STR tested. (R7←C=MEM[R1-1]←R3)

LC-3-SIM> mdump 0x3003 0x3005
Memory content [0x3003..0x3005] :
                          // shell - mdump tested.
-------------------------------------
  0x3003 (12291) : 0x0e       // STI tested.
  0x3004 (12292) : 0x3003
  0x3005 (12293) : 0x3026     // STR tested.

LC-3-SIM> quit
Bye.
```

Then we test our program of **lab 1 Question 2** on our simulator. Made some minor adjustment such that the input is stored at `0x3030` and `0x3031`.

```
lc3as lab1-q2.asm
hexdump -v -e ' "0x" 2/1 "%02X" "\n" ' lab1-q2.obj | tee lab1-q2
```

```
./simulate lab1-q2
LC-3 Simulator
Read 50 words from program into memory.
LC-3-SIM> go
Simulating...
Simulator halted
LC-3-SIM> mdump 0x3030 0x3033
Memory content [0x3030..0x3033] :
-------------------------------------
  0x3030 (12336) : 0x965d // Input 1001 0110 0101 1101 (the original bit pattern).
  0x3031 (12337) : 0x05    // Input 5 (the shift amount).
  0x3032 (12338) : 0x4b2  // Output 0000 0100 1011 0010. The correct answer. This sample is given by Lab1-Question2.
  0x3033 (12339) : 0x00
```

The result is correct.