CS2651 Computer Architecture (Fall 2022) Lab 3: LC-3 Simulator

Xiangyuan Xue (521030910387)

1 Control Logic

1.1 Program Target

The simulator is partitioned into the shell and the simulation routines. Since the shell is already provided, we are only required to implement the simulation routines. Observe the cycle function for the shell. It simply does three things:

1. Process a single instruction.
2. Update the current latches.
3. Increase the instruction counter.

Therefore, our target function is expected to finish the following tasks:

1. Execute the current instruction.
2. Update the states stored in latches.
3. Make corresponding modification in memory.

All the states must be updated accurately to keep the program running correctly.

1.2 Execution Steps

Generally, an instruction cycle is divided into 6 phases: F (Fetch Instruction), D (Decode), EA (Evaluate Address), OP (Fetch Operands), EX (Execute) and SR (Store Results). In order to simulate the procedure, we describe the steps in another way as follow:

1. Copy the current latches and fetch `PC`.
2. Load the instruction `IR` in memory location specified by `PC`.
3. Increase `PC` by 1.
4. Decode `IR` and fetch `OPCODE`.
5. Further decode `IR` according to `OPCODE`.
6. Execute the specific instruction.

The details of execution are listed as follow:

1. Fetch registers such as `DR`, `SR` and `BaseR`.
2. Fetch the data stored in registers.
3. Fetch extra information, such as offset, immediate and vector.
4. Calculate the target memory address.
5. Calculate the results of operation.
6. Update the data in registers or memory.
7. Update the condition code if necessary.

There are some other details for some special instructions, which will be described later. Anyway, the general procedure has been included by above discussion.

1.3 Data Type

We should note that the simulator is written in C, and the given part uses 32-bit integer to store all the information. However, the LC-3 ISA is completely based on 16-bit operations, which will be wrongly simulated by 32-bit integers. Therefore, we include `stdint.h` and define two extra data types:

```c
typedef int16_t DATA;
typedef uint16_t uDATA;
```

where `DATA` is a 16-bit integer type for the arithmetic and logic operations, and `uDATA` is the unsigned version for other values such as `IR` and `PC`. Every time we fetch information from registers and memory, we convert the integers into these specific data types, which avoids unexpected simulation errors. When a 16-bit integer is converted into a 32-bit integer, a bitwise operation is required to cover directly:

```c
#define To32bits(x) ((x) | (0x00000000))
NEXT_LATCHES.REGS[DR] = To32bits(RESULT);
```

When a 32-bit integer is converted into a 16-bit integer, a bitwise operation is needed to fetch the lower 16 bits:

```c
#define Low16bits(x) ((x) & (0xFFFF))
DATA PC = Low16bits(NEXT_LATCHES.PC);
```

With these methods, simulation problems related to data range can be effectively handled.

2 Important Functions

2.1 Operation Code

There are 16 operation codes in LC-3 ISA, which takes 4 bits for expression. In order to express the intuitively, we construct an enumeration type called `OPCODE`:

```
typedef enum
{
    OPCODE_ADD = 0b0001,
    OPCODE_AND = 0b0101,
    OPCODE_BR = 0b0000,
    OPCODE_JMP = 0b1100,
    OPCODE_JSR = 0b0100,
    OPCODE_LD = 0b0010,
    OPCODE_LDI = 0b1010,
    OPCODE_LDR = 0b0110,
    OPCODE_LEA = 0b1110,
    OPCODE_NOT = 0b1001,
    OPCODE_RTI = 0b1000,
    OPCODE_ST = 0b0011,
    OPCODE_STI = 0b1011,
    OPCODE_STR = 0b0111,
    OPCODE_TRAP = 0b1111,
    OPCODE_RESERVED = 0b1101,
} OPCODE;
```

This enumeration type maps the operation codes to their names, so we can directly enter the branch of the specific operation by `switch` command:

```
OPCODE OP = (IR >> 12);
switch (OP)
{
    case ...:
    {
        ...;
        break;
    }
}
```

This structure builds the basic framework of the process function.

2.2 Sign Extension

Nearly half of the instructions contain immediate and offset. These values are encoded in the instruction and vary a lot in length. For example, ADD and AND supports a 5-bit immediate, while LD and ST supports a 9-bit offset, which will be operated together with the 16-bit data. Thus, these values are required to be extended. This extension converts the n-bit value x to a 16-bit data while maintaining its sign. The algorithm is described as follow:

- If x is non-negative, append digits 0 to the most significant side of x until x is 16-bit-long.
- If x is negative, append digits 1 to the most significant side of x until x is 16-bit-long.

which can be conveniently implemented with bitwise operation. This procedure is encapsulated into a function called `sext`, which is shown as follow:

```
DATA sext(DATA value, DATA bits)
{
    if (value & (1 << (bits - 1)))
    {
        DATA mask = 0xFFFF ^ ((1 << bits) - 1);
        return value | mask;
    }
    return value;
}
```

where the parameter `value` is the x to be extended and `bits` denotes the length n. Now the immediate and offset encoded in the instructions can be extended conveniently.

2.3 Condition Code

In LC-3 ISA, there are 6 instructions that will change the condition code: ADD, AND, NOT, LD, LDI, LDR. Each time that are executed, the condition code will be updated by the result. Thus, such procedure will be similarly repeated. To simplify the implementation, this procedure is encapsulated into a function called `setcc`:

```
void setcc(DATA value)
{
    if (value == 0)
    {
        NEXT_LATCHES.N = 0;
        NEXT_LATCHES.Z = 1;
        NEXT_LATCHES.P = 0;
    }
    else if (value > 0)
    {
        NEXT_LATCHES.N = 0;
        NEXT_LATCHES.Z = 0;
        NEXT_LATCHES.P = 1;
    }
    else
    {
        NEXT_LATCHES.N = 1;
        NEXT_LATCHES.Z = 0;
        NEXT_LATCHES.P = 0;
    }
}
```

where the parameter `value` is the result of a specific instruction. This function updates the condition code according to `value` conveniently.

2.4 Processing

First, we should fetch the instruction `IR`, increase the program counter `PC` and extract the operation code `OPCODE`:

```
NEXT_LATCHES = CURRENT_LATCHES;
DATA PROGRAM_COUNTER = Low16bits(NEXT_LATCHES.PC);
uDATA IR = Low16bits(MEMORY[PROGRAM_COUNTER]);
PROGRAM_COUNTER = PROGRAM_COUNTER + 1;
NEXT_LATCHES.PC = PROGRAM_COUNTER;
OPCODE OP = (IR >> 12);
```

The basic framework implemented by `switch` command has been introduced above, so we will skip this part. The ADD operation is implemented as follow:

```
uDATA DR = Low3bits(IR >> 9);
DATA RESULT;
uDATA SR1 = Low3bits(IR >> 6);
DATA OPERAND1 = Low16bits(NEXT_LATCHES.REGS[SR1]);
if (IR & (1 << 5)) // immediate mode
{
    DATA IMM5 = sext((IR & ((1 << 5) - 1)), 5);
    RESULT = OPERAND1 + IMM5;
}
else // register mode
{
    uDATA SR2 = Low3bits(IR);
    DATA OPERAND2 = Low16bits(NEXT_LATCHES.REGS[SR2]);
    RESULT = OPERAND1 + OPERAND2;
}
NEXT_LATCHES.REGS[DR] = To32bits(RESULT);
setcc(RESULT);
```

The AND operation is implemented as follow:

```
uDATA DR = Low3bits(IR >> 9);
DATA RESULT;
uDATA SR1 = Low3bits(IR >> 6);
DATA OPERAND1 = Low16bits(NEXT_LATCHES.REGS[SR1]);
if (IR & (1 << 5)) // immediate mode
{
    DATA IMM5 = sext((IR & ((1 << 5) - 1)), 5);
    RESULT = OPERAND1 & IMM5;
}
else // register mode
{
    uDATA SR2 = Low3bits(IR);
    DATA OPERAND2 = Low16bits(NEXT_LATCHES.REGS[SR2]);
    RESULT = OPERAND1 & OPERAND2;
}
NEXT_LATCHES.REGS[DR] = To32bits(RESULT);
setcc(RESULT);
```

The BR operation is implemented as follow:

```
uDATA NZP = Low3bits(IR >> 9);
uDATA STATE = Low16bits((NEXT_LATCHES.N << 2) | (NEXT_LATCHES.Z << 1) | (
   NEXT_LATCHES.P));
if (NZP & STATE) // condition satisfied
{
    DATA OFFSET9 = sext((IR & ((1 << 9) - 1)), 9);
    DATA PC = Low16bits(NEXT_LATCHES.PC);
    PC += OFFSET9;
    NEXT_LATCHES.PC = To32bits(PC);
}
```

The JMP operation is implemented as follow:

```
uDATA BASER = Low3bits(IR >> 6);
DATA PC = Low16bits(NEXT_LATCHES.REGS[BASER]);
NEXT_LATCHES.PC = To32bits(PC);
```

The JSR operation is implemented as follow:

```
DATA PC = Low16bits(NEXT_LATCHES.PC);
NEXT_LATCHES.REGS[7] = PC;
if (IR & (1 << 11)) // type JSR
{
    DATA OFFSET11 = sext((IR & ((1 << 11) - 1)), 11);
    PC += OFFSET11;
    NEXT_LATCHES.PC = To32bits(PC);
}
else // type JSRR
{
    uDATA BASER = Low3bits(IR >> 6);
    PC = Low16bits(NEXT_LATCHES.REGS[BASER]);
    NEXT_LATCHES.PC = To32bits(PC);
}
```

The LD operation is implemented as follow:

```
uDATA DR = Low3bits(IR >> 9);
DATA OFFSET9 = sext((IR & ((1 << 9) - 1)), 9);
DATA PC = Low16bits(NEXT_LATCHES.PC);
DATA ADDRESS = PC + OFFSET9;
DATA RESULT = Low16bits(MEMORY[ADDRESS]);
NEXT_LATCHES.REGS[DR] = To32bits(RESULT);
setcc(RESULT);
```

The LDI operation is implemented as follow:

```
uDATA DR = Low3bits(IR >> 9);
DATA OFFSET9 = sext((IR & ((1 << 9) - 1)), 9);
DATA PC = Low16bits(NEXT_LATCHES.PC);
DATA ADDRESS = PC + OFFSET9;
ADDRESS = Low16bits(MEMORY[ADDRESS]);
DATA RESULT = Low16bits(MEMORY[ADDRESS]);
NEXT_LATCHES.REGS[DR] = To32bits(RESULT);
setcc(RESULT);
```

The LDR operation is implemented as follow:

```
uDATA DR = Low3bits(IR >> 9);
uDATA BASER = Low3bits(IR >> 6);
DATA BASE = Low16bits(NEXT_LATCHES.REGS[BASER]);
DATA OFFSET6 = sext((IR & ((1 << 6) - 1)), 6);
DATA ADDRESS = BASE + OFFSET6;
DATA RESULT = Low16bits(MEMORY[ADDRESS]);
NEXT_LATCHES.REGS[DR] = To32bits(RESULT);
setcc(RESULT);
```

The LEA operation is implemented as follow:

```
uDATA DR = Low3bits(IR >> 9);
DATA OFFSET9 = sext((IR & ((1 << 9) - 1)), 9);
DATA PC = Low16bits(NEXT_LATCHES.PC);
DATA RESULT = PC + OFFSET9;
NEXT_LATCHES.REGS[DR] = To32bits(RESULT);
```

The NOT operation is implemented as follow:

```
uDATA DR = Low3bits(IR >> 9);
uDATA SR = Low3bits(IR >> 6);
DATA OPERAND = Low16bits(NEXT_LATCHES.REGS[SR]);
DATA RESULT = ~OPERAND;
NEXT_LATCHES.REGS[DR] = To32bits(RESULT);
setcc(RESULT);
```

The ST operation is implemented as follow:

```
uDATA SR = Low3bits(IR >> 9);
DATA OFFSET9 = sext((IR & ((1 << 9) - 1)), 9);
DATA PC = Low16bits(NEXT_LATCHES.PC);
DATA ADDRESS = PC + OFFSET9;
DATA RESULT = Low16bits(NEXT_LATCHES.REGS[SR]);
MEMORY[ADDRESS] = To32bits(RESULT);
```

The STI operation is implemented as follow:

```
uDATA SR = Low3bits(IR >> 9);
DATA OFFSET9 = sext((IR & ((1 << 9) - 1)), 9);
DATA PC = Low16bits(NEXT_LATCHES.PC);
DATA ADDRESS = PC + OFFSET9;
ADDRESS = Low16bits(MEMORY[ADDRESS]);
DATA RESULT = Low16bits(NEXT_LATCHES.REGS[SR]);
MEMORY[ADDRESS] = To32bits(RESULT);
```

The STR operation is implemented as follow:

```
uDATA SR = Low3bits(IR >> 9);
uDATA BASER = Low3bits(IR >> 6);
DATA BASE = Low16bits(NEXT_LATCHES.REGS[BASER]);
DATA OFFSET6 = sext((IR & ((1 << 6) - 1)), 6);
DATA ADDRESS = BASE + OFFSET6;
DATA RESULT = Low16bits(NEXT_LATCHES.REGS[SR]);
MEMORY[ADDRESS] = To32bits(RESULT);
```

The TRAP operation is implemented as follow:

```
uDATA VECTOR = (IR & 0xFF);
if (VECTOR == 0x20) // trap GETC
{
    DATA VALUE = getchar();
    NEXT_LATCHES.REGS[0] = To32bits(VALUE);
}
else if (VECTOR == 0x21) // trap OUT
{
    DATA VALUE = Low16bits(NEXT_LATCHES.REGS[0]);
    printf("%c", (VALUE & 0xFF));
}
else if (VECTOR == 0x22) // trap PUTS
{
    DATA ADDRESS = Low16bits(NEXT_LATCHES.REGS[0]);
    while (1)
    {
        DATA VALUE = Low16bits(MEMORY[ADDRESS]);
        if (VALUE == 0)
                break;
        printf("%c", (VALUE & 0xFF));
        ADDRESS = ADDRESS + 1;
    }
}
else if (VECTOR == 0x23) // trap IN
{
    printf("Input a character>");
    DATA VALUE = getchar();
    NEXT_LATCHES.REGS[0] = To32bits(VALUE);
    printf("%c\n", (VALUE & 0xFF));
}
else if (VECTOR == 0x24) // trap PUTSP
{
    DATA ADDRESS = Low16bits(NEXT_LATCHES.REGS[0]);
    while (1)
    {
        DATA VALUE = Low16bits(MEMORY[ADDRESS]);
        if (VALUE == 0)
            break;
        DATA FIRST = (VALUE & 0xFF);
        printf("%c", (FIRST & 0xFF));
        DATA SECOND = (VALUE >> 8);
        if (SECOND)
            printf("%c", (SECOND & 0xFF));
        ADDRESS = ADDRESS + 1;
    }
}
else if (VECTOR == 0x25) // trap HALT
{
    NEXT_LATCHES.PC = 0;
    printf("\n----- Halting the processor -----\n");
}
NEXT_LATCHES.PC = 0;
```

Since the RTI operation and the RESERVED operation are not specified, they are left empty and do nothing. Memory-mapped I/O is not implemented. Note that according to the requirement, the LEA instruction does not change the condition code and after the TRAP instruction, the program counter PC is set to 0.

3 Verification

3.1 Outline

Since the simulator has been completely implemented, we compile the C program into executable file:

```
gcc -std=c99 -o simulate lc3sim.c
```

Then we can simulate the LC-3 program stored in hexadecimal file:

```
./simulate *.hex
```

In this part, we will run three LC-3 programs on our simulator and compare the results that provided by the standard simulator, in order to test the correctness and accuracy of our simulator. The three LC-3 programs come from the former labs and are slightly modified for the convenience of testing. They are exactly `fibonacci`, `rightshift` and `calculator`, and more details will be specified later.

3.2 Fibonacci Series

The assembly program `fibonacci.asm` calculates the n-th term f_n in fibonacci series, where n is specified in memory location `0x300F` and the result will be stored into memory location `x3010`. Following LC-3 instructions are included in this program:

- ADD, AND, BR, TRAP, LD, ST

whose accuracy will be examined in this running. The results are as follow:

```
LC-3 Simulator
Read 17 words from program into memory.

LC-3-SIM> mdump 0x3000 0x3010
Memory content [0x3000..0x3010] :
-------------------------------------
  0x3000 (12288) : 0x220e
  0x3001 (12289) : 0x54a0
  0x3002 (12290) : 0x56e0
  0x3003 (12291) : 0x16e1
  0x3004 (12292) : 0x5920
  0x3005 (12293) : 0x1260
  0x3006 (12294) : 0xc06
  0x3007 (12295) : 0x1883
  0x3008 (12296) : 0x14e0
  0x3009 (12297) : 0x1720
  0x300a (12298) : 0x127f
  0x300b (12299) : 0x401
  0x300c (12300) : 0xffa
  0x300d (12301) : 0x3802
  0x300e (12302) : 0xf025
  0x300f (12303) : 0x0a
  0x3010 (12304) : 0x00

LC-3-SIM> go
Simulating...
----- Halting the processor -----
Simulator halted

LC-3-SIM> rdump
Current register/bus values :
-------------------------------------
Instruction Count : 68
PC                : 0x0000
CCs: N = 0  Z = 1  P = 0
Registers:
0: 0x0000
1: 0x0000
2: 0x0037
3: 0x0059
4: 0x0059
5: 0x0000
6: 0x0000
7: 0x0000

LC-3-SIM> mdump 0x3000 0x3010
Memory content [0x3000..0x3010] :
-------------------------------------
  0x3000 (12288) : 0x220e
  0x3001 (12289) : 0x54a0
  0x3002 (12290) : 0x56e0
  0x3003 (12291) : 0x16e1
  0x3004 (12292) : 0x5920
  0x3005 (12293) : 0x1260
  0x3006 (12294) : 0xc06
  0x3007 (12295) : 0x1883
  0x3008 (12296) : 0x14e0
  0x3009 (12297) : 0x1720
  0x300a (12298) : 0x127f
  0x300b (12299) : 0x401
  0x300c (12300) : 0xffa
  0x300d (12301) : 0x3802
  0x300e (12302) : 0xf025
  0x300f (12303) : 0x0a
  0x3010 (12304) : 0x59

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

In this examination, we calculate f_{10} and the program yields 89, which is the correct answer. Compared to the results provided by the standard LC-3 simulator, the data in registers and memory are completely correct.

3.3 Right Shift

The assembly program `rightshift.asm` implements the right shift function which has been described in former labs. The value to be shifted x is specified in memory location `0x3015` and the amount to shift k is specified in memory location `x3016`. The result will be stored into memory location `x3100`. Following LC-3 instructions are included in this program:

- ADD, AND, BR, TRAP, LD, ST, STI

whose accuracy will be examined in this running. The results are as follow:

```
LC-3 Simulator
Read 24 words from program into memory.

LC-3-SIM> mdump 0x3015 0x3016
Memory content [0x3015..0x3016] :
-------------------------------------
  0x3015 (12309) : 0xb251
  0x3016 (12310) : 0x09

LC-3-SIM> mdump 0x3100 0x3100
Memory content [0x3100..0x3100] :
-------------------------------------
  0x3100 (12544) : 0x00

LC-3-SIM> go
Simulating...
----- Halting the processor -----
Simulator halted

LC-3-SIM> rdump
Current register/bus values :
-------------------------------------
```

```
Instruction Count : 950
PC                : 0x0000
CCs: N = 0  Z = 1  P = 0
Registers:
0: 0x0059
1: 0x0000
2: 0x0059
3: 0xffff8000
4: 0x0000
5: 0x0000
6: 0x0000
7: 0x0000

LC-3-SIM> mdump 0x3015 0x3016
Memory content [0x3015..0x3016] :
-------------------------------------
  0x3015 (12309) : 0xb251
  0x3016 (12310) : 0x09

LC-3-SIM> mdump 0x3100 0x3100
Memory content [0x3100..0x3100] :
-------------------------------------
  0x3100 (12544) : 0x59

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

In this examination, we right shift `0b1011001001010001` by 9 bits and the program yields `0b0000000001011001`, which is the correct answer. Compared to the results provided by the standard LC-3 simulator, the data in registers and memory are completely correct.

3.4 Assembly Calculator

The assembly program `calculator.asm` implements an assembly calculator in LC-3 ISA, which has been described in former labs. Following operations are supported by this calculator:

- ADD, MUL, NEG, DIV, MOD, NOT, XOR

In addition, the user can insert a value, display top value, clear value stack and exit the calculator. This calculator is interactive, so the examination can be performed in console. Following LC-3 instructions are included in this program:

- ADD, AND, BR, JMP, JSR, LD, LDR, LEA, NOT, ST, STR, TRAP

For the TRAP instruction, four types of TRAP are included: GETC, OUT, PUTS, HALT.

Almost all the LC-3 instructions are used by this calculator, which promises the effectiveness of this examination. The results are as follow:

```
LC-3 Simulator
Read 973 words from program into memory.

LC-3-SIM> go
Simulating...

Assembly Calculator by xxyQwQ

Command: 19
19
[Insert]

Command: 47
47
[Insert]

Command: *
[Multiply]

Command: D
+893
[Display]

Command: -
[Negate]

Command: D
-893
[Display]

Command: 144
144
[Insert]

Command: +
[Plus]

Command: D
-749
[Display]
```

```
Command: 999
999
[Insert]

Command: -
[Negate]

Command: +
+
[Error] range limit exceeds
[Plus]

Command: C
C
[Clear]

Command: D
D
[Error] stack is empty
[Display]

Command: X
[Exit]

----- Halting the processor -----
Simulator halted

LC-3-SIM> rdump
Current register/bus values :
-------------------------------------
Instruction Count : 2000
PC                : 0x0000
CCs: N = 0  Z = 0  P = 1
Registers:
0: 0x0058
1: 0x0000
2: 0x0000
3: 0x0000
4: 0x0000
5: 0x0000
6: 0x33c8
```

```
7: 0x300c


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

In this examination, we try to perform some operations by the assembly calculator in our LC-3 simulator, and it yields the correct answer. Compared to the results provided by the standard LC-3 simulator, the data in registers and memory are completely correct. Therefore, we can conclude that our LC-3 simulator works accurately.

References

[1] Yale N. Patt and Sanjay J. Patel. Introduction to Computing Systems. New York: McGraw-Hill, 2020.

[2] Github. LC3sim-C. https://github.com/rpendleton/lc3sim-c.