Lab Report: Lending Your Name

Zhu Yunqin, PB20061372

Task

Get and store the n-th number of a sequence $F(n)$ in a LC-3 machine using assembly code. The sequence is defined by the following formula.

$$F(0) = 1,\ F(1) = 1,\ F(2) = 2,$$
$$F(n) = [F(n-1) + 2F(n-3)]\ mod\ 1024, when\ 3 \leq n \leq 16384.$$

Requirements

- Store the result into `R7`.
- In the initial state, n is stored in `R0`, and each other register from `R1` to `R7` is initialized to 0.
- Divide the student number into four equal segments, labelling them with a, b, c and d. In this case, the author's student ID is PB20061372, so $a = 17,\ b = 0,\ c = 1,\ d = 44$. Store the value of $F(a), F(b), F(c)$ and $F(d)$ at the end of the code with `.FILL` pseudo command.
- The number of lines should be within 18, excluding `.ORIG` and `.END`.

Procedure

---

1. Write the assembly code in the file *fib.asm*；
2. Write the testing program using C++ and LC3Tools API， which assembles *fib.asm*, simulates a LC-3 machine, generates random initial states and checks the correctness and the efficiency accordingly；
3. Copy the assembly code into the file *fib.txt*.

The testing program written in C++ is as follows.

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <random>
```

```cpp
#define API_VER 2
#include "console_inputter.h"
#include "console_printer.h"
#include "interface.h"

using namespace std;
using namespace lc3;

const int CASE_NUM = 50;
const string ASM_SUFFIX = ".asm";
uint32_t print_level = 4;
bool enable_liberal_asm = false;
bool ignore_privilege = false;
uint32_t inst_limit = 1919810;

uint16_t F(uint16_t n) {
  static uint16_t f[0x4000] = {1, 1, 2};
  return f[n] ? f[n] : f[n] = (F(n - 1) + 2 * F(n - 3)) % 1024;
}

int main(int argc, char* argv[]) {
  if (argc != 2) return 0;
  // Initialize
  ConsolePrinter printer;
  ConsoleInputter inputter;
  lc3::as assembler(printer, print_level, enable_liberal_asm);
  lc3::sim simulator(printer, inputter, print_level);
  simulator.setIgnorePrivilege(ignore_privilege);
  simulator.setRunInstLimit(inst_limit);
  string filename(argv[1]);
  if (filename.size() >= ASM_SUFFIX.size() &&
      equal(ASM_SUFFIX.rbegin(), ASM_SUFFIX.rend(), filename.rbegin()))
    filename = assembler.assemble(filename)->first;
  else {
    cerr << "invalid asm file" << endl;
    return 0;
  }
  // Test
  uint64_t prev_count, sum = 0;
  uint16_t n, result, sample[] = {20, 6, 13, 72, 0, 1, 2, 3};
  mt19937 gen(unsigned(time(0)));
  uniform_int_distribution<uint16_t> dis(0x0000, 0x4000);
  for (int i = 0; i < CASE_NUM; i++) {
    // Generate random numbers
    n = i < 8 ? sample[i] : dis(gen);
```

```
    // Set machine state
    prev_count = simulator.getInstExecCount();
    simulator.zeroState();
    simulator.writeReg(0, n);
    if (!simulator.loadObjFile(filename)) {
      cerr << "invalid obj file" << endl;
      return 0;
    }
    // Run and check
    simulator.runUntilHalt();
    result = static_cast<uint16_t>(simulator.readReg(7));
    if (F(n) != result) {
      cerr << "wrong answer" << endl;
      return 0;
    }
    sum += simulator.getInstExecCount() - prev_count;
    cout << "F(" << n << ") = " << result << endl;
  }
  // Print result
  cout << CASE_NUM << " testing cases passed" << endl;
  cout << "instruction count: " << sum << endl;
  cout << "average instruction count: " << 1.0 * sum / CASE_NUM << endl;
  return 0;
}
```

Execute the testing program *lab2_test.exe* in Windows Powershell and load *fib.asm*. The input and output info would have the following format.

```
PS C:\> .\lab2_test.exe .\fib.asm
attempting to assemble .\fib.asm into .\fib.obj
assembly successful
F(20) = 930
F(6) = 18
F(13) = 710
F(72) = 66
F(...) = ...
F(...) = ...
...
F(...) = ...
50 testing cases passed
instruction count: ...
average instruction count: ...
```

If the assembly program goes wrong in any testing case, the testing program would print the error message as below and exit.

```
wrong answer
```

Accordingly, we are able to evaluate the correctness and the efficiency of the assembly program.

Results

Version 1

- **Basic idea**

  Note that $F(n)$ could be calculated by repeatedly using the recursive formula

  $$F(n) = [F(n-1) + 2F(n-3)] \; mod \; 1024$$

  within a loop structure, controlled by a `BR` instruction. The main problem lies in how to implement the formula with LC-3 instructions.
  - Firstly, it is necessary to store the values of $F(n-1)$ and $F(n-3)$ such that we could read and update them in each iteration. Say we have $F(n-1)$, $F(n-2)$ and $F(n-3)$ in registers `R7`, `R6` and `R5` respectively.
    - Every time we need to calculate $F(n)$, we add `R7` with two times `R5` and store the remainder into `R7`.
    - Then we replace `R5` with `R6` and `R6` with the original `R7`, which has been temporarily memorized in `R2`.
    - As a result, we now have $F(n)$, $F(n-1)$ and $F(n-2)$ in registers `R7`, `R6` and `R5` respectively, ready for the next iteration.
  - Most of the operations above coule be implemented with simple `ADD` instructions, excluding the step of "$mod \; 1024$". If we consider the dividend for this modular operation as a 16-bit binary number

    $$b_{15}b_{14}b_{13}b_{12}b_{11}b_{10}b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0,$$

    the remainder actually equals

    $$000000b_9b_8b_7b_6b_5b_4b_3b_2b_1b_0,$$

    since we only need the part less than 2^{10}, i.e. 1024. Therefore, we just `AND` the dividend with a bit mask `x3ff`, which has been loaded from memory to `R1` with a `LD` instruction.
  - Last but not least, we take account of the cases when n equals 0, 1 or 2 with `BR` instructions.

  The assembly program is shown below.
- **Assembly code**

```
.ORIG x3000
ADD R7, R7, #1        ; case n = 0, 1
ADD R0, R0, #-1
BRnz DONE
ADD R7, R7, #1        ; case n = 2
ADD R0, R0, #-1
BRz DONE
ADD R5, R5, #1        ; case n >= 3
ADD R6, R6, #1
LD R1, BITMASK
LOOP ADD R2, R7, #0
     ADD R7, R7, R5   ; calculate
     ADD R7, R7, R5
     AND R7, R7, R1
     ADD R5, R6, #0   ; update
     ADD R6, R2, #0
     ADD R0, R0, #-1
BRp LOOP
DONE HALT
BITMASK .FILL x3ff
.END
```

- **Output**

```
attempting to assemble fib.asm into fib.obj
assembly successful
F(20) = 930
F(6) = 18
F(13) = 710
F(72) = 66
F(0) = 1
F(1) = 1
F(2) = 2
F(3) = 4
F(157) = 518
F(6009) = 422
F(3852) = 866
F(1277) = 774
F(9620) = 930
F(1090) = 82
F(16136) = 578
F(8533) = 838
F(738) = 338
F(13159) = 950
F(12792) = 450
F(7711) = 758
```

```
F(13333) = 326
F(4623) = 374
F(9014) = 498
F(13924) = 546
F(11511) = 310
F(4236) = 866
F(2909) = 6
F(7650) = 338
F(2559) = 1014
F(10463) = 246
F(5727) = 246
F(15572) = 418
F(7695) = 374
F(12993) = 102
F(8471) = 54
F(13153) = 870
F(12950) = 242
F(4258) = 850
F(2206) = 818
F(15305) = 294
F(7956) = 930
F(4086) = 1010
F(491) = 534
F(4044) = 354
F(12389) = 198
F(12257) = 870
F(13288) = 322
F(9416) = 66
F(9653) = 70
F(3367) = 438
50 testing cases passed
instruction count: 2693229
average instruction count: 53864.6
```

- **Assessment**
  - Correctness: ✓
  - Number of lines: 19
  - Average number of instructions: 53864.6
- **Analysis**

  The program correctly gets the value of $F(n)$ for all 50 cases, with 19 lines of assembly code, which meets the requirements of this lab task. To further cut down the number of lines, we should introduce some tricky optimization to the logic of our program.

Version 2

- **Basic idea**

  The program could be simplified with the following two ideas.

  - In the first version, we add `R0` with `#-1` twice to distinguish between the two cases $n = 0, 1$ and $n = 2$. In fact, it is possible to change `R0` only once and then `BR` to different cases. By `ADD R0, R0, #-2`, we know that if the conditional code `n` is set, $n = 0, 1$; if the conditional code `z` is set, $n = 2$; otherwise, we branch to the case $n \geq 3$.
  - In the first version, we use `R2` as a temp to store $F(n-1)$ in each iteration such that we could update `R6` after changing `R7`, but the storage of `R2` would count as an extra line of assembly code. In fact, if using `R2` as a temp for $2 \cdot F(n-3)$ instead of $F(n-1)$, we could carry out one step of calculation and the storage of `R2` within one instruction, i.e., `ADD R2 R5 R5`.

  The assembly program is shown below.
- **Assembly code**

```
.ORIG x3000
AGAIN ADD R7, R7, #1        ; first optimization
      ADD R0, R0, #-2
BRn DONE
BRz AGAIN
ADD R5, R5, #1
ADD R6, R6, #1
ADD R7, R7, #1
LD R1, BITMASK
LOOP ADD R2, R5, R5         ; second optimization
     ADD R5, R6, #0
     ADD R6, R7, #0
     ADD R7, R7, R2
     AND R7, R7, R1
     ADD R0, R0, #-1
BRp LOOP
DONE HALT
BITMASK .FILL x3ff
.END
```

- **Output**

```
attempting to assemble fib.asm into fib.obj
assembly successful
F(20) = 930
F(6) = 18
F(13) = 710
F(72) = 66
F(0) = 1
F(1) = 1
```

```
F(2) = 2
F(3) = 4
F(7831) = 54
F(1273) = 422
F(13177) = 422
F(13112) = 962
F(3688) = 322
F(10130) = 722
F(263) = 694
F(5025) = 358
F(10360) = 450
F(2719) = 758
F(4862) = 562
F(14423) = 566
F(6283) = 278
F(14337) = 614
F(1965) = 902
F(12560) = 642
F(15052) = 354
F(4257) = 358
F(12608) = 2
F(15562) = 658
F(11969) = 102
F(2789) = 198
F(13318) = 114
F(12791) = 310
F(15584) = 258
F(1026) = 594
F(6874) = 786
F(13609) = 550
F(5644) = 866
F(8159) = 246
F(9101) = 134
F(818) = 978
F(8183) = 310
F(10019) = 342
F(16112) = 386
F(7559) = 694
F(6668) = 866
F(16341) = 838
F(11149) = 134
F(4374) = 242
F(10409) = 550
F(10843) = 150
50 testing cases passed
```

```
instruction count: 2610361
average instruction count: 52207.2
```

- **Assessment**
  - Correctness: ✓
  - Number of lines: 17
  - Average number of instructions: 52207.2
- **Analysis**
  The program correctly gets the value of $F(n)$ for all 50 cases. Compared with the first version, the number of lines is reduced from 19 to 17, and also the average number of instructions is slightly reduced due to the optimization. These facts correspond to our ideas.

According to the testing program, the four encoded segments of the author's student number are,

$$
\begin{gathered}
F(20) = 930, \\
F(6) = 18, \\
F(13) = 710, \\
F(72) = 66.
\end{gathered}
$$

Append `.FILL` pseudo commands to the end of the assembly program, and the complete code is shown below.

```
.ORIG x3000
AGAIN ADD R7, R7, #1
      ADD R0, R0, #-2
BRn DONE
BRz AGAIN
ADD R5, R5, #1
ADD R6, R6, #1
ADD R7, R7, #1
LD R1, BITMASK
LOOP ADD R2, R5, R5
     ADD R5, R6, #0
     ADD R6, R7, #0
     ADD R7, R7, R2
     AND R7, R7, R1
     ADD R0, R0, #-1
BRp LOOP
DONE HALT
BITMASK .FILL x3ff
FA .FILL #930
FB .FILL #18
FC .FILL #710
FD .FILL #66
.END
```

Finally, save the assembly program as *fib.txt*.