Lab Report: Riddle

Zhu Yunqin, PB20061372

Task

Description

Reproduce the following C++ program functionally in LC-3.

```
int judge(int r0) {
    int i = 2;
    r1 = 1;
    while (i * i <= r0) {
        if (r0 % i == 0) {
            r1 = 0;
            break;
        }
        i++;
    }
    return r1;
}
```

Requirements

- An integer `R0` is given before the program executes.
- The program should return the final result in `R1`.
- The program should follow a specific framework shown below.

```
.ORIG x3000
... ; TO BE DONE
HALT
JUDGE ... ; TO BE DONE
... ; TO BE DONE
RET
... ; TO BE DONE
.END
```

- Store the code in *prime.txt*.

Solution

In order to present the solution clearly, assume that there are following instructions in LC-3:

1. `MUL` instruction, which multiplies two integers and returns the product;
2. `MOD` instruction, which divides one unsigned integer by another and returns the remainder;
3. `CMP` instruction, which compares two unsigned integers and returns a flag -1, 0 or 1 for less than, equal to or greater than.

Then, a modified `void judge()` function can be translated into a subroutine as:

```
JUDGE       AND R1, R1, #0  ;void judge() { r1 = 0;
            ADD R1, R1, #1  ; r1 += 1;
            ADD R2, R1, #1  ; i = &r2; i = r1 + 1;
JUDGE_LOOP  MUL R3, R2, R2  ; while (1) { r3 = i * i;
            CMP R3, R0, R3  ;   r3 = compare(r0, r3);
            BRp JUDGE_BREAK ;   if (r3 > 0) break;
            MOD R3, R2, R0  ;   r3 = r0 % i;
```

```
            BRnp JUDGE_IF   ;   if (r3 == 0) {
            AND R1, R1, #0  ;     r1 = 0;
            BR JUDGE_BREAK  ;     break;
JUDGE_IF    ADD R2, R2, #1  ;    } i++;
            BR JUDGE_LOOP   ;   continue;
JUDGE_BREAK RET             ; } return; }
```

The reproduced function above has two main problems:

1. `r0` is not the formal parameter, and `r1` is not the return value.
2. `r2` and `r3` are not local variables.

For the first problem, we maintain a stack to store the formal parameter `R0` and the return value `R1` such that we pop `R0` at the beginning of the subroutine and push `R1` at the end.

For the second one, we allocate a memory unit to save the original value of `r3` and load it when the subroutine ends.

Besides, we need to implement the `MUL`, `MOD` and `CMP` instructions with other subroutines and replace them with `JSR` instructions. Following the idea of using a stack, we use subroutines like `PUSH` and `POP` to push and pop formal parameters and return values.

Considering the changes of `R0`, `R2` and `R7`, we need to save and load them in the same manner as `R3`.

The renewed subroutine is as follows.

```

JUDGE           ST R0, JUDGE_SAVE_R0
                ST R2, JUDGE_SAVE_R2
                ST R3, JUDGE_SAVE_R3
                ST R7, JUDGE_SAVE_R7

                JSR POP_R0
                AND R1, R1, #0
                ADD R1, R1, #1
                ADD R2, R1, #1
JUDGE_LOOP      JSR PUSH_R2
                JSR PUSH_R2
                JSR MUL
                JSR PUSH_R0
                JSR CMP
                JSR POP_R3
                BRp JUDGE_BREAK
                JSR PUSH_R0
                JSR PUSH_R2
                JSR MOD
                JSR POP_R3
                BRnp JUDGE_IF
                AND R1, R1, #0
                BR JUDGE_BREAK
JUDGE_IF        ADD R2, R2, #1
                BR JUDGE_LOOP
JUDGE_BREAK     JSR PUSH_R1

                LD R0, JUDGE_SAVE_R0
                LD R2, JUDGE_SAVE_R2
                LD R3, JUDGE_SAVE_R3
                LD R7, JUDGE_SAVE_R7
                RET

                JUDGE_SAVE_R0 .BLKW #1
                JUDGE_SAVE_R1 .BLKW #1
                JUDGE_SAVE_R2 .BLKW #1
                JUDGE_SAVE_R3 .BLKW #1
                JUDGE_SAVE_R7 .BLKW #1
```

Below, we discuss how to implement all the other subroutines.

About Stack

First, we write the subroutine `PUSH` for pushing `R0` onto the stack and `POP` for popping `R0` from the stack.

```
POP             ST R1, STACK_SAVE_R1
                ST R2, STACK_SAVE_R2
                ST R5, STACK_SAVE_R5
                ST R6, STACK_SAVE_R6

                LD R1, STACK_EMPTY
                LD R6, STACK_POINTER
                AND R5, R5, #0
                ADD R2, R6, R1
                BRnp POP_SUCCESS
                ADD R5, R5, #1
                BR POP_OK
POP_SUCCESS     LDR R0, R6, #0
                ADD R6, R6, #1
POP_OK          ST R5, STACK_FAIL_FLAG
                ST R6, STACK_POINTER

                LD R1, STACK_SAVE_R1
                LD R2, STACK_SAVE_R2
                LD R5, STACK_SAVE_R5
                LD R6, STACK_SAVE_R6
                RET

PUSH            ST R1, STACK_SAVE_R1
                ST R2, STACK_SAVE_R2
                ST R5, STACK_SAVE_R5
                ST R6, STACK_SAVE_R6

                LD R1, STACK_FULL
                LD R6, STACK_POINTER
                AND R5, R5, #0
                ADD R2, R6, R1
                BRnp PUSH_SUCCESS
                ADD R5, R5, #1
                BR PUSH_OK
PUSH_SUCCESS    ADD R6, R6, #-1
                STR R0, R6, #0
PUSH_OK         ST R5, STACK_FAIL_FLAG
                ST R6, STACK_POINTER

                LD R1, STACK_SAVE_R1
                LD R2, STACK_SAVE_R2
                LD R5, STACK_SAVE_R5
                LD R6, STACK_SAVE_R6
                RET

STACK_FAIL_FLAG .FILL #0
STACK_POINTER   .FILL x4000
STACK_EMPTY     .FILL x-4000
STACK_FULL      .FILL x-3FF0
STACK_SAVE_R1   .BLKW #1
STACK_SAVE_R2   .BLKW #1
STACK_SAVE_R5   .BLKW #1
STACK_SAVE_R6   .BLKW #1
```

We use a labeled memory unit `STACK_POINTER` to represent the address of the stack top and `STACK_FAIL_FLAG` to represent if the operation failed.

When we push `R0`, we decrement `STACK_POINTER` and store `R0` at the address pointed by `STACK_POINTER`. `STACK_FAIL_FLAG` is set to 1 iff. `STACK_POINTER` has already reached the address `x3FF0`, which means the stack is full.

When we pop `R0`, we load `R0` from the address pointed by `STACK_POINTER` and increment `STACK_POINTER`. `STACK_FAIL_FLAG` is set to 1 iff. `STACK_POINTER` has already reached the address `x4000`, which means the stack is empty.

To push and pop values for any register, we further wrap the push operation with subroutines from `PUSH_R0` to `PUSH_R7` and, similarly, the pop operation.

```
POP_R0          ST R7, STACK_R_SAVE_R7
                JSR POP
                LD R7, STACK_R_SAVE_R7
                ADD R0, R0, #0
                RET

POP_R1          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R1, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R1, R1, #0
                RET

POP_R2          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R2, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R2, R2, #0
                RET

POP_R3          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R3, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R3, R3, #0
                RET

POP_R4          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R4, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R4, R4, #0
                RET

POP_R5          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R5, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R5, R5, #0
                RET

POP_R6          ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                JSR POP
                ADD R6, R0, #0
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                ADD R6, R6, #0
                RET

PUSH_R0         ST R7, STACK_R_SAVE_R7
                JSR PUSH
```

```
                LD R7, STACK_R_SAVE_R7
                RET

PUSH_R1         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R1, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET

PUSH_R2         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R2, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET

PUSH_R3         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R3, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET

PUSH_R4         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R4, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET

PUSH_R5         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R5, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET

PUSH_R6         ST R0, STACK_R_SAVE_R0
                ST R7, STACK_R_SAVE_R7
                ADD R0, R6, #0
                JSR PUSH
                LD R0, STACK_R_SAVE_R0
                LD R7, STACK_R_SAVE_R7
                RET

STACK_R_SAVE_R0 .BLKW #1
STACK_R_SAVE_R7 .BLKW #1
```

Note that we use an instruction like `ADD R?, R?, #0` to update `CC` for each `POP_R?`.

About Multiplication

Second, we consider the multiplication operation. According to the L-version solution for lab 1, we have the following subroutine.

```
MUL             ST R0, MUL_SAVE_R0
                ST R1, MUL_SAVE_R1
                ST R2, MUL_SAVE_R2
                ST R3, MUL_SAVE_R3
                ST R7, MUL_SAVE_R7
```

```
                JSR POP_R1
                JSR POP_R0
                AND R2, R2, #0
                AND R3, R3, #0
                ADD R3, R3, #1
MUL_LOOP        AND R7, R0, R3
                BRz MUL_SKIP
                ADD R2, R2, R1
MUL_SKIP        ADD R1, R1, R1
                ADD R3, R3, R3
                BRnp MUL_LOOP
                JSR PUSH_R2

                LD R0, MUL_SAVE_R0
                LD R1, MUL_SAVE_R1
                LD R2, MUL_SAVE_R2
                LD R3, MUL_SAVE_R3
                LD R7, MUL_SAVE_R7
                RET

MUL_SAVE_R0     .BLKW #1
MUL_SAVE_R1     .BLKW #1
MUL_SAVE_R2     .BLKW #1
MUL_SAVE_R3     .BLKW #1
MUL_SAVE_R7     .BLKW #1
```

Here, we do not repeat the details of the multiplication algorithm. The subroutine `MUL` pops the two operands from the stack, multiplies them, and then pushes the result back to the stack. `MOD`, `CMP` and the others process the operands in the same way.

About Modulo

Inspired by the pen-and-paper division of multi-digit decimal numbers, we have the algorithm for division with a binary radix. Suppose we are to divide N by D, placing the quotient in Q and the remainder in R. The following is the pseudo-code from *Wikipedia*.

```
Q := 0                  -- Initialize quotient and remainder to zero
R := 0
for i := n − 1 .. 0 do  -- Where n is number of bits in N
  R := R << 1           -- Left-shift R by 1 bit
  R(0) := N(i)          -- Set the least-significant bit of R equal to bit i of the numerator
  if R ≥ D then
    R := R − D
    Q(i) := 1
  end
end
```

Translate the pseudo-code into LC-3 language, and then we have the following subroutine.

```
MOD             ST R0, MOD_SAVE_R0
                ST R1, MOD_SAVE_R1
                ST R2, MOD_SAVE_R2
                ST R3, MOD_SAVE_R3
                ST R4, MOD_SAVE_R4
                ST R7, MOD_SAVE_R7
                JSR POP_R1
                JSR POP_R0

                NOT R3, R1
                ADD R3, R3, #1
                AND R4, R4, #0
                ADD R4, R4, #-16
MOD_INIT_LOOP   AND R7, R0, #-1
                BRn MOD_INIT_BREAK
                ADD R0, R0, R0
                ADD R4, R4, #1
```

```
                BR MOD_INIT_LOOP
MOD_INIT_BREAK  AND R2, R2, #0
MOD_MAIN_LOOP   ADD R4, R4, #1
                BRp MOD_MAIN_BREAK
                ADD R2, R2, R2
                AND R7, R0, #-1
                BRzp MOD_SKIP
                ADD R2, R2, #1
MOD_SKIP        JSR PUSH_R1
                JSR PUSH_R2
                JSR CMP
                JSR POP_R5
                BRp MOD_IF
                ADD R2, R2, R3
MOD_IF          ADD R0, R0, R0
                BR MOD_MAIN_LOOP

MOD_MAIN_BREAK  JSR PUSH_R2
                LD R0, MOD_SAVE_R0
                LD R1, MOD_SAVE_R1
                LD R2, MOD_SAVE_R2
                LD R3, MOD_SAVE_R3
                LD R4, MOD_SAVE_R4
                LD R7, MOD_SAVE_R7
                RET

MOD_SAVE_R0     .BLKW #1
MOD_SAVE_R1     .BLKW #1
MOD_SAVE_R2     .BLKW #1
MOD_SAVE_R3     .BLKW #1
MOD_SAVE_R4     .BLKW #1
MOD_SAVE_R7     .BLKW #1
```

Note that at each stage, we use a `CMP` subroutine to compare R in `R2` and D in `R1`. Then, we subtract D from R if R is greater than D. The `CMP` subroutine guarantees that any unsigned integers could be compared correctly, regardless of the sign bits.

About Comparison

In the `CMP` subroutine, we need to compare the sign bits of the two operands initially. If one is greater, return. Otherwise, we branch to different instructions according to the result of `SUB` subroutine, which substracts the one operand from the other.

The assembly code is as follows.

```
CMP             ST R0, CMP_SAVE_R0
                ST R1, CMP_SAVE_R1
                ST R2, CMP_SAVE_R2
                ST R7, CMP_SAVE_R7

                AND R2, R2, #0
                ADD R2, R2, #-1
                JSR POP_R1
                JSR POP_R0
                AND R7, R0, R1
                BRn CMP_SUB
                AND R7, R0, #-1
                BRn CMP_GREATER
                AND R7, R1, #-1
                BRn CMP_LESS
CMP_SUB         JSR PUSH_R0
                JSR PUSH_R1
                JSR SUB
                JSR POP_R0
                BRn CMP_LESS
                BRz CMP_ZERO
CMP_GREATER     ADD R2, R2, #1
CMP_ZERO        ADD R2, R2, #1
```

```
CMP_LESS        JSR PUSH_R0

                LD R0, CMP_SAVE_R0
                LD R1, CMP_SAVE_R1
                LD R2, CMP_SAVE_R2
                LD R7, CMP_SAVE_R7
                RET
CMP_SAVE_R0     .BLKW #1
CMP_SAVE_R1     .BLKW #1
CMP_SAVE_R2     .BLKW #1
CMP_SAVE_R7     .BLKW #1
```

About Substraction

Finally, we give the assembly code for the `SUB` subroutine, which is not necessary but convenient.

```

SUB             ST R0, SUB_SAVE_R0
                ST R1, SUB_SAVE_R1
                ST R7, SUB_SAVE_R7

                JSR POP_R1
                JSR POP_R0
                NOT R1, R1
                ADD R1, R1, #1
                ADD R0, R0, R1
                JSR PUSH_R0

                LD R0, SUB_SAVE_R0
                LD R1, SUB_SAVE_R1
                LD R7, SUB_SAVE_R7
                RET

SUB_SAVE_R0     .BLKW #1
SUB_SAVE_R1     .BLKW #1
SUB_SAVE_R7     .BLKW #1
```

Put Them Together

Combine the subroutines mentioned above, and then we have the complete assembly code for the function `int judge(int r0)`. The main body of the program is shown below.

```

.ORIG x3000

                JSR PUSH_R0
                JSR JUDGE
                JSR POP_R1
                HALT

JUDGE           ...
...

; Subtraction
SUB             ...
...

; Multiplication
MUL             ...
...

; Modulo
MOD             ...
...
```

```
; Comparison
CMP             ...
...

; Stack
POP            ...
...

.END
```

The complete assembly code has been stored in *prime.txt*.

The following program is written with C++ and LC3Tools API, which assembles *prime.txt*, simulates a LC-3 machine and check if the subroutine `JUDGE` has the same return value as the function `int judge(int r0)`. Note that we judge a prime number using another function instead of `int judge(int r0)`.

```
#include <algorithm>
#include <iostream>
#include <numeric>
#include <vector>
using namespace std;

#define API_VER 2
#include "console_inputter.h"
#include "console_printer.h"
#include "interface.h"
lc3::ConsolePrinter printer;
lc3::ConsoleInputter inputter;
uint32_t print_level = 4;

bool is_prime(int n) {
  if (n <= 3) return true;
  if (n % 2 == 0 || n % 3 == 0) return false;
  for (int i = 5; i * i <= n; i += 6) {
    if (n % i == 0 || n % (i + 2) == 0) return false;
  }
  return true;
}

// Assemble assembly code
const string assemble(const string& filename) {
  if (filename.empty()) return "";
  // Initialize
  static bool enable_liberal_asm = false;
  static lc3::as assembler(printer, print_level, enable_liberal_asm);
  return assembler.assemble(filename)->first;
}

// Test assembled program
bool test(const string& filename) {
  if (filename.empty()) return false;
  // Initialize
  static bool init_flag = true;
  static bool ignore_privilege = false;
  static uint32_t inst_limit = 1919810;
  static lc3::sim simulator(printer, inputter, print_level);
  if (init_flag) {
    simulator.setIgnorePrivilege(ignore_privilege);
    simulator.setRunInstLimit(inst_limit);
    init_flag = false;
  }
  vector<uint16_t> data(10000);
  iota(data.begin(), data.end(), 0);
  for (uint16_t r0 : data) {
    // Set machine state
```

```
    simulator.zeroState();
    if (!simulator.loadObjFile(filename)) {
      cerr << "Error: invalid file " << filename << endl;
      return false;
    }
    simulator.writeReg(0, r0);
    // Run and check
    simulator.runUntilHalt();
    cout << r0 << "\t" << simulator.readReg(1) << endl;
    if (simulator.readReg(1) != is_prime(r0)) return false;
  }
  return true;
}

int main(int argc, char* argv[]) {
  if (test(assemble("lab5/prime.txt")))
    cout << "---TEST PASSED---" << endl;
  else
    cout << "---TEST FAILED---" << endl;
  return 0;
}
```

The testing program prints the following output.

```
0       1
1       1
2       1
3       1
4       0
...
9997    0
9998    0
9999    0
---TEST PASSED---
```

Clearly, the assembly code is correct.