Lab Report: Reveal Yourself

Zhu Yunqin, PB20061372

Task 1: *rec.txt*

Description

Find out the four missing bits in a program.

Requirements

- Read the damaged machine code in *task1.txt*, in which the missing bits are represented by `x`.
- Write the restored code in *rec.txt*.
- Before the program starts, values in the registers except `PC` are set to zero. At the end of the program, the register status is as follows.

```
R0 = x5, R1 = x0, R2 = x300F, R3 = x0,
R4 = x0, R5 = x0, R6 = x0, R7 = x3003;
```

Solution

If we replace each missing bit with `0` or `1`, there will be finitely $2^4 = 16$ possible combinations. It means that we can quickly solve the problem by enumerating all possibilities of undamaged machine code and checking the operation result one by one. Specifically, we need to do the following steps.

1. **Generate** all possible versions of undamaged machine code and store them into files separately.
2. **Translate** the machine code in each file into assembly code.
3. Attempt to **assemble** each program.
4. For each program assembled successfully, run it to **test** whether the registers are in the correct status.

The following c++ program implements all the functions mentioned above using LC3Tools API.

```cpp
#include <algorithm>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <map>
#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;

// Generate possible machine code
const vector<string> generate(const string& filename) {
  vector<string> out_names;
  if (filename.empty()) return out_names;
  string content;
  getline(ifstream(filename), content, '\0');
  string dir = filename.substr(0, filename.find_last_of('/')) + "/gen";
  filesystem::create_directory(dir);
  vector<int> indices;
  for (int i = 0; i < content.size(); i++)
    if (content[i] == 'x') indices.push_back(i);
```

```
  int n = indices.size();
  for (int cnt = 0; cnt < 1 << n; cnt++) {
    for (int i = 0; i < n; i++)
      content[indices[i]] = '0' + ((cnt & 1 << i) > 0);
    string dir_i = dir + '/' + to_string(cnt);
    filesystem::create_directory(dir_i);
    string out_name = dir_i + '/' + to_string(cnt) + ".txt";
    ofstream(out_name) << content;
    out_names.push_back(out_name);
  }
  return out_names;
}

// Translate machine code into assembly code
const string translate(const string& filename) {
  if (filename.empty()) return "";
  string name = filename.substr(0, filename.find_last_of('.'));
  ifstream in(filename);
  ofstream out(name + ".asm");

  uint16_t PC = 0x3000, opcode, DR, SR1, SR2, BaseR, trapvect8;
  int16_t imm5, PCoffset9, PCoffset11, offset6;
map<uint16_t, string> branch_labels;
map<uint16_t, string> subroutine_labels;
map<uint16_t, string> data_labels;
bool is_data = false;
string line;
stringstream buf;
buf << ".ORIG x3000" << endl;

// Process each instruction
while (!in.eof()) {
  getline(in, line);
  line.erase(remove_if(line.begin(), line.end(), ::isspace), line.end());
  if (line.size() == 0) continue;
  if (line.size() != 16) {
    cerr << "Error: invalid line " << line << "at" << filename << endl;
    return "";
  }
  if (branch_labels.find(PC) != branch_labels.end() ||
      subroutine_labels.find(PC) != subroutine_labels.end())
    is_data = false;
  PC++;
  if (is_data) {
    buf << ".FILL x" << hex << uppercase << stoi(line, nullptr, 2) << endl;
    continue;
  }
  opcode = stoi(line.substr(0, 4), nullptr, 2);
  switch (opcode) {
      /* Since the collected pieces of code do not contain all kinds of
      instructions, I leave those unused cases in blank.*/

    case 0b0001:  // ADD
      DR = stoi(line.substr(4, 3), nullptr, 2);
      SR1 = stoi(line.substr(7, 3), nullptr, 2);
      buf << "ADD R" << DR << ", R" << SR1;
      if (line[10] == '0') {
        if (line.substr(11, 2) != "00") {
          cerr << "Error: invalid line " << line << "at" << filename << endl;
          return "";
        }
        SR2 = stoi(line.substr(13, 3), nullptr, 2);
        buf << ", R" << SR2;
      } else {
        imm5 = stoi(line.substr(11, 5), nullptr, 2);
        if (line[11] == '1') imm5 = imm5 - 32;
        buf << ", #" << imm5;
```

```
      }
      buf << endl;
      break;

    case 0b0101:  // AND
      DR = stoi(line.substr(4, 3), nullptr, 2);
      SR1 = stoi(line.substr(7, 3), nullptr, 2);
      buf << "AND R" << DR << ", R" << SR1;
      if (line[10] == '0') {
        if (line.substr(11, 2) != "00") {
          cerr << "Error: invalid line " << line << "at" << filename << endl;
          return "";
        }
        SR2 = stoi(line.substr(13, 3), nullptr, 2);
        buf << ", R" << SR2;
      } else {
        imm5 = stoi(line.substr(11, 5), nullptr, 2);
        if (line[11] == '1') imm5 = imm5 - 32;
        buf << ", #" << imm5;
      }
      buf << endl;
      break;

    case 0b0000:  // BR
      buf << "BR";
      if (line[4] == '1') buf << 'n';
      if (line[5] == '1') buf << 'z';
      if (line[6] == '1') buf << 'p';
      PCoffset9 = stoi(line.substr(7, 9), nullptr, 2);
      if (line[7] == '1') PCoffset9 = PCoffset9 - 512;
      if (branch_labels.find(PC + PCoffset9) == branch_labels.end())
        branch_labels[PC + PCoffset9] =
            "BRANCH" + to_string(branch_labels.size());
      buf << ' ' << branch_labels[PC + PCoffset9] << endl;
      break;

    case 0b1100:  // JMP && RET
      if (line.substr(4, 3) != "000" || line.substr(10) != "000000") {
        cerr << "Error: invalid line " << line << "at" << filename << endl;
        return "";
      }
      BaseR = stoi(line.substr(7, 3), nullptr, 2);
      if (BaseR == 7)
        buf << "RET" << endl;
      else
        buf << "JMP R" << BaseR << endl;
      is_data = true;
      break;

    case 0b0100:  // JSR && JSRR
      if (line[4] == '1') {
        buf << "JSR";
        PCoffset11 = stoi(line.substr(5, 11), nullptr, 2);
        if (line[5] == '1') PCoffset11 = PCoffset11 - 2048;
        if (subroutine_labels.find(PC + PCoffset11) ==
            subroutine_labels.end())
          subroutine_labels[PC + PCoffset11] =
              "SUB" + to_string(subroutine_labels.size());
        buf << ' ' << subroutine_labels[PC + PCoffset11] << endl;
      } else if (line[5] == '0' && line[6] == '0') {
        // incomplete
        buf << "JSRR";
      } else {
        cerr << "Error: invalid line " << line << "at" << filename << endl;
        return "";
      }
      break;
```

```
    case 0b0010:  // LD
      DR = stoi(line.substr(4, 3), nullptr, 2);
      buf << "LD R" << DR << ", ";
      PCoffset9 = stoi(line.substr(7, 9), nullptr, 2);
      if (line[7] == '1') PCoffset9 = PCoffset9 - 512;
      if (data_labels.find(PC + PCoffset9) == data_labels.end())
        data_labels[PC + PCoffset9] = "DATA" + to_string(data_labels.size());
      buf << data_labels[PC + PCoffset9] << endl;
      break;

    case 0b1010:  // LDI
      DR = stoi(line.substr(4, 3), nullptr, 2);
      buf << "LDI R" << DR << ", ";
      PCoffset9 = stoi(line.substr(7, 9), nullptr, 2);
      if (line[7] == '1') PCoffset9 = PCoffset9 - 512;
      if (data_labels.find(PC + PCoffset9) == data_labels.end())
        data_labels[PC + PCoffset9] = "DATA" + to_string(data_labels.size());
      buf << data_labels[PC + PCoffset9] << endl;
      break;

    case 0b0110:  // LDR
      DR = stoi(line.substr(4, 3), nullptr, 2);
      BaseR = stoi(line.substr(7, 3), nullptr, 2);
      buf << "LDR R" << DR << ", R" << BaseR;
      offset6 = stoi(line.substr(11, 5), nullptr, 2);
      if (line[10] == '1') offset6 = offset6 - 64;
      buf << ", #" << offset6 << endl;
      break;

    case 0b1110:  // LEA
      DR = stoi(line.substr(4, 3), nullptr, 2);
      buf << "LEA R" << DR << ", ";
      PCoffset9 = stoi(line.substr(7, 9), nullptr, 2);
      if (line[7] == '1') PCoffset9 = PCoffset9 - 512;
      if (data_labels.find(PC + PCoffset9) == data_labels.end())
        data_labels[PC + PCoffset9] = "DATA" + to_string(data_labels.size());
      buf << data_labels[PC + PCoffset9] << endl;
      break;

    case 0b1001:  // NOT
      if (line.substr(10) != "111111") {
        cerr << "Error: invalid line " << line << "at" << filename << endl;
        return "";
      }
      DR = stoi(line.substr(4, 3), nullptr, 2);
      SR1 = stoi(line.substr(7, 3), nullptr, 2);
      buf << "NOT R" << DR << ", R" << SR1 << endl;
      break;

    case 0b1000:  // RTI
      // incomplete
      buf << "RTI" << endl;
      break;

    case 0b0011:  // ST
      DR = stoi(line.substr(4, 3), nullptr, 2);
      buf << "ST R" << DR << ", ";
      PCoffset9 = stoi(line.substr(7, 9), nullptr, 2);
      if (line[7] == '1') PCoffset9 = PCoffset9 - 512;
      if (data_labels.find(PC + PCoffset9) == data_labels.end())
        data_labels[PC + PCoffset9] = "DATA" + to_string(data_labels.size());
      buf << data_labels[PC + PCoffset9] << endl;
      break;

    case 0b1011:  // STI
      DR = stoi(line.substr(4, 3), nullptr, 2);
```

```
      buf << "STI R" << DR << ", ";
      PCoffset9 = stoi(line.substr(7, 9), nullptr, 2);
      if (line[7] == '1') PCoffset9 = PCoffset9 - 512;
      if (data_labels.find(PC + PCoffset9) == data_labels.end())
        data_labels[PC + PCoffset9] = "DATA" + to_string(data_labels.size());
      buf << data_labels[PC + PCoffset9] << endl;
      break;

    case 0b0111:  // STR
      DR = stoi(line.substr(4, 3), nullptr, 2);
      BaseR = stoi(line.substr(7, 3), nullptr, 2);
      buf << "STR R" << DR << ", R" << BaseR;
      offset6 = stoi(line.substr(11, 5), nullptr, 2);
      if (line[10] == '1') offset6 = offset6 - 64;
      buf << ", #" << offset6 << endl;
      break;

    case 0b1111:  // TRAP
      // incomplete
      if (line.substr(4, 4) != "0000") {
        cerr << "Error: invalid line " << line << "at" << filename << endl;
        return "";
      }
      trapvect8 = stoi(line.substr(8), nullptr, 2);
      switch (trapvect8) {
        case 0x20:
          buf << "GETC";
          break;
        case 0x21:
          buf << "OUT";
          break;
        case 0x22:
          buf << "PUTS";
          break;
        case 0x23:
          buf << "IN";
          break;
        case 0x24:
          buf << "PUTSP";
          break;
        case 0x25:
          buf << "HALT";
          // Treat following lines as data
          is_data = true;
          break;
        default:
          buf << "TRAP x" << hex << trapvect8;
          break;
      }
      buf << endl;
      break;

    case 0b1101:  // reserved
      buf << ".FILL x" << hex << uppercase << stoi(line, nullptr, 2) << endl;
      break;
  }
}
buf << ".END";

// Output with label
getline(buf, line);
out << line << endl;
PC = 0x3000;
while (!buf.eof()) {
  getline(buf, line);
  if (branch_labels.find(PC) != branch_labels.end())
    out << branch_labels[PC] << ' ';
```

```
    if (data_labels.find(PC) != data_labels.end())
      out << data_labels[PC] << ' ';
    if (subroutine_labels.find(PC) != subroutine_labels.end())
      out << subroutine_labels[PC] << ' ';
    out << line << endl;
    PC++;
  }
  return name + ".asm";
}

// 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 = 1107;
  static lc3::sim simulator(printer, inputter, print_level);
  if (init_flag) {
    simulator.setIgnorePrivilege(ignore_privilege);
    simulator.setRunInstLimit(inst_limit);
    init_flag = false;
  }
  // Set machine state
  simulator.zeroState();
  if (!simulator.loadObjFile(filename)) {
    cerr << "Error: invalid file " << filename << endl;
    return false;
  }
  simulator.runUntilHalt();
  // Check machine state
  return simulator.readReg(0) == 0x5 && simulator.readReg(1) == 0x0 &&
         simulator.readReg(2) == 0x300f && simulator.readReg(3) == 0x0 &&
         simulator.readReg(4) == 0x0 && simulator.readReg(5) == 0x0 &&
         simulator.readReg(6) == 0x0 && simulator.readReg(7) == 0x3003;
}

int main(int argc, char* argv[]) {
  for (auto& file : generate("argv[1]"))
    if (test(assemble(translate(file))))
      cout << "\n---TEST PASSED---\n"
           << file << "\n-----------------\n"
           << endl;
    else
      cout << "\n---TEST FAILED---\n"
           << file << "\n-----------------\n"
           << endl;
  return 0;
}
```

Execute the program *lab4_task1.exe* in Windows Powershell and load *task1.txt*, and the input and output info will be as follows.

```
PS C:\lab4\task1> .\lab4_task1.exe task1.txt
attempting to assemble ./gen/0/0.asm into ./gen/0/0.obj
assembly successful

---TEST FAILED---
```

```
./gen/0/0.txt
-----------------

attempting to assemble ./gen/1/1.asm into ./gen/1/1.obj
./gen/1/1.asm:15:5: error: could not find label
JSR SUB1
    ^~~~

error: assembly failed (pass 2)

---TEST FAILED---
./gen/1/1.txt
-----------------

attempting to assemble ./gen/2/2.asm into ./gen/2/2.obj
assembly successful

---TEST FAILED---
./gen/2/2.txt
-----------------

attempting to assemble ./gen/3/3.asm into ./gen/3/3.obj
./gen/3/3.asm:15:5: error: could not find label
JSR SUB1
    ^~~~

error: assembly failed (pass 2)

---TEST FAILED---
./gen/3/3.txt
-----------------

attempting to assemble ./gen/4/4.asm into ./gen/4/4.obj
assembly successful

---TEST FAILED---
./gen/4/4.txt
-----------------

attempting to assemble ./gen/5/5.asm into ./gen/5/5.obj
./gen/5/5.asm:15:5: error: could not find label
JSR SUB1
    ^~~~

error: assembly failed (pass 2)

---TEST FAILED---
./gen/5/5.txt
-----------------

attempting to assemble ./gen/6/6.asm into ./gen/6/6.obj
assembly successful

---TEST FAILED---
./gen/6/6.txt
-----------------

attempting to assemble ./gen/7/7.asm into ./gen/7/7.obj
./gen/7/7.asm:15:5: error: could not find label
JSR SUB1
    ^~~~

error: assembly failed (pass 2)

---TEST FAILED---
./gen/7/7.txt
-----------------
```

```
attempting to assemble ./gen/8/8.asm into ./gen/8/8.obj
assembly successful

---TEST FAILED---
./gen/8/8.txt
-----------------

attempting to assemble ./gen/9/9.asm into ./gen/9/9.obj
assembly successful

---TEST PASSED---
./gen/9/9.txt
-----------------

attempting to assemble ./gen/10/10.asm into ./gen/10/10.obj
assembly successful

---TEST FAILED---
./gen/10/10.txt
-----------------

attempting to assemble ./gen/11/11.asm into ./gen/11/11.obj
assembly successful


--- Access violation---


---TEST FAILED---
./gen/11/11.txt
-----------------

attempting to assemble ./gen/12/12.asm into ./gen/12/12.obj
assembly successful

---TEST FAILED---
./gen/12/12.txt
-----------------

attempting to assemble ./gen/13/13.asm into ./gen/13/13.obj
assembly successful

---TEST FAILED---
./gen/13/13.txt
-----------------

attempting to assemble ./gen/14/14.asm into ./gen/14/14.obj
assembly successful
warning: 32112: Skipping 'Updating Keyboard' scheduled for 32110
warning: 32112: Skipping 'Updating Display' scheduled for 32110
warning: 32112: Skipping 'No interrupt of higher priority pending' scheduled for 32111

---TEST FAILED---
./gen/14/14.txt
-----------------

attempting to assemble ./gen/15/15.asm into ./gen/15/15.obj
assembly successful

---TEST FAILED---
./gen/15/15.txt
-----------------
```

After the execution, the file tree will be as follows.

```
PS C:\lab4\task1> Tree /F /A
C:.
|   task1.txt
|
\---gen
    +---0
    |       0.asm
    |       0.obj
    |       0.txt
    |
    +---1
    |       1.asm
    |       1.obj
    |       1.txt
    |
    +---10
    |       10.asm
    |       10.obj
    |       10.txt
    |
    +---11
    |       11.asm
    |       11.obj
    |       11.txt
    |
    +---12
    |       12.asm
    |       12.obj
    |       12.txt
    |
    +---13
    |       13.asm
    |       13.obj
    |       13.txt
    |
    +---14
    |       14.asm
    |       14.obj
    |       14.txt
    |
    +---15
    |       15.asm
    |       15.obj
    |       15.txt
    |
    +---2
    |       2.asm
    |       2.obj
    |       2.txt
    |
    +---3
    |       3.asm
    |       3.obj
    |       3.txt
    |
    +---4
    |       4.asm
    |       4.obj
    |       4.txt
    |
    +---5
    |       5.asm
    |       5.obj
    |       5.txt
    |
```

```
    +---6
    |       6.asm
    |       6.obj
    |       6.txt
    |
    +---7
    |       7.asm
    |       7.obj
    |       7.txt
    |
    +---8
    |       8.asm
    |       8.obj
    |       8.txt
    |
    \---9
            9.asm
            9.obj
            9.txt
```

According to the output, only the program in *.\gen\9\9.asm* works correctly with the info `---TEST PASSED---` as its operation result. The content of *9.asm* is shown below.

```
.ORIG x3000
LEA R2, DATA0
AND R0, R0, #0
JSR SUB0
HALT
SUB0 STR R7, R2, #0
ADD R2, R2, #1
ADD R0, R0, #1
LD R1, DATA1
ADD R1, R1, #-1
ST R1, DATA1
BRz BRANCH0
JSR SUB0
BRANCH0 ADD R2, R2, #-1
LDR R7, R2, #0
RET
DATA0 .FILL x0
.FILL x0
.FILL x0
.FILL x0
.FILL x0
.FILL x0
.FILL x0
.FILL x0
.FILL x0
DATA1 .FILL x5
.END
```

According to the restored assembly code, we know that the program implements a subroutine labeled `SUB0`, which changes the value in `DATA1` from `x5` to `x0` recursively. The register `R2` serves as a pointer to the current stack frame and `R7` always holds the PC for the last caller whenever we return. Thus, it is reasonable that the program will halt with `R2 = x300F` (i.e., the address of `DATA0`, the bottom of the stack) and `R7 = x3003` (i.e., the address right after the first `JSR SUB0`).

Results

The content of *9.txt* is shown below.

```
1110010000001110
0101000000100000
0100100000000001
1111000000100101
```

```
0111111010000000
0001010010100001
0001000000100001
0010001000010001
0001001001111111
0011001000001111
0000010000000001
0100111111111000
0001010010111111
0110111010000000
1100000111000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000101
```

The index 9 (i.e., 1001) indicates that the missing bits are `1`, `0`, `0` and `1`.
The machine code has been duplicated into *rec.txt*.

Task 2: *mod.txt*

Description

A program calculates the remainder after dividing one integer by 7, but with 15 bits in the code missing. Find them out.

Requirements

- Read the damaged machine code in *task2.txt*, in which the missing bits are represented by `x`.
- Write the restored code in *mod.txt*.

Solution

Since there are 15 missing bits, it is impractical to generate all $2^{15} = 32768$ possibilities of the original machine code as we did in task 1, which would take excessive time and space.
In order to better understand the given program, we can first translate the undamaged lines into assembly code with the `translate()` function written in task 1. The following are the results.

```
.ORIG x3000
LD R1, DATA0          ; line 1
JSR SUB0
AND R2, R1, #7
ADD R1, R2, R4
ADD R0, R?, ?         ; 00010000xxx11001
BRp ???               ; 00000011xxx11011
ADD R0, R?, ?         ; 00010000xxx11001
BRn BRANCH1
ADD R1, R1, #-7
BRANCH1 HALT
SUB0 AND R2, R2, #0
AND R3, R3, #0
AND R4, R4, #0
ADD R2, R2, #1
ADD R3, R3, #8
BRANCH3 AND R5, R3, R1
BRz BRANCH2
ADD R4, R2, R4
```

```
BRANCH2 ADD R2, R2, R2
ADD R?, R3, R3        ; 0001xxx011000011
BR??? BRANCH3         ; 0000xxx111111010
RET
DATA0 .FILL x120
.END
```

Note that the two `ADD` instructions at line 5 and line 7 must be of immediate mode, or the syntax would be wrong. Therefore, we assert that the 3rd and the 9th missing bits are both `1`. As a result, there would be $2^{12} = 8196$ possible cases left to enumerate and check.
Besides, we can see that the program loads `DATA0` into `R1` at the beginning and changes the value of `R1` right before it halts. It means that `R1` might be the dividend for the modulo operation and then be the remainder as an output. If we replace line 1 with a null instruction `0000000000000000`, we could then test the correctness of each possible combination of code using LC3Tools API.
The modified *task2.txt* is shown below.

```
0000000000000000  ; changed
0100100000001000
0101010001100111
0001001010000100
00010000xx111001  ; changed
00000011xxx11011
0000100000000001
0001001001111001
1111000000100101
0101010010100000
0101011011100000
0101100100100000
0001010010100001
0001011011101000
0101101011000001
0000010000000001
0001100010000100
0001010010000010
0001xxx011000011
0000xxx111111010
1100000111000000
0000000100100000
```

Aiming at less extra storage space, we will use the following `process()` function to process each possible case separately instead of generating all code files at once.

```
const vector<string> process(const string& filename) {
  vector<string> out_names;
  if (filename.empty()) return out_names;
  string content;
  getline(ifstream(filename), content, '\0');
  string dir = filename.substr(0, filename.find_last_of('/')) + "/gen";
  filesystem::create_directory(dir);
  vector<int> indices;
  for (int i = 0; i < content.size(); i++)
    if (content[i] == 'x') indices.push_back(i);
  int n = indices.size();
  for (int cnt = 0; cnt < 1 << n; cnt++) {
    // Generate code
    for (int i = 0; i < n; i++)
      content[indices[i]] = '0' + ((cnt & 1 << i) > 0);
    string dir_i = dir + '/' + to_string(cnt);
    filesystem::create_directory(dir_i);
    string file = dir_i + '/' + to_string(cnt) + ".txt";
    ofstream(file) << content;
    // Test code
    if (test(assemble(translate(file)))) {
      cout << "\n---TEST PASSED---\n"
           << file << "\n----------------\n"
           << endl;
```

```
      // Return right code
      out_names.push_back(file);
    } else {
      cout << "\n---TEST FAILED---\n"
           << file << "\n-----------------\n"
           << endl;
      // Delete wrong code
      filesystem::remove_all(dir_i);
    }
  }
  return out_names;
}
```

The new `test()` and `main()` functions are as follows.

```
// 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 = 2333;
  static lc3::sim simulator(printer, inputter, print_level);
  if (init_flag) {
    simulator.setIgnorePrivilege(ignore_privilege);
    simulator.setRunInstLimit(inst_limit);
    init_flag = false;
  }
  /* PS:
  The data set considers the original input 0x120 and some other crucial samples.
  */
  for (uint16_t i : {0x120, 0, 1, 7, 2006, 1372, 0xffff}) {
    // Set machine state
    simulator.zeroState();
    if (!simulator.loadObjFile(filename)) {
      cerr << "Error: invalid file " << filename << endl;
      return false;
    }
    simulator.writeReg(1, i);
    // Run and check
    simulator.runUntilHalt();
    if (simulator.readReg(1) != i % 7) return false;
  }
  return true;
}
// Main
int main(int argc, char* argv[]) {
  process(argv[1]);
  return 0;
}
```

To execute the c++ program *lab4_task2.exe* in Windows Powershell and load the modified *task2.txt*, the input should be like `PS C:\lab4\task2> .\lab4_task2.exe task2.txt` .

The program would finish with the following output file tree.

```
PS C:\lab4\task2> TREE /F /A
C:.
|   task2.txt
|
\---gen
    \---5982
            5982.asm
            5982.obj
            5982.txt
```

Only the code in the directory 5982 (i.e., 1011101011110) passes the test and remains, meaning that the correct combination of the left 13 missing bits should be `01`, `111`, `01`, `011`, and `101`. The content of *5982.asm* is shown below.

```
.ORIG x3000
BR BRANCH0
BRANCH0 JSR SUB0
AND R2, R1, #7
ADD R1, R2, R4
ADD R0, R1, #-7
BRp BRANCH0
ADD R0, R1, #-7
BRn BRANCH1
ADD R1, R1, #-7
BRANCH1 HALT
SUB0 AND R2, R2, #0
AND R3, R3, #0
AND R4, R4, #0
ADD R2, R2, #1
ADD R3, R3, #8
BRANCH3 AND R5, R3, R1
BRz BRANCH2
ADD R4, R2, R4
BRANCH2 ADD R2, R2, R2
ADD R3, R3, R3
BRnp BRANCH3
RET
.FILL x120
.END
```

Note that the first line of `BR` instruction is originally for param initialization. Replace it with `LD` and re-label each line, and one possible version of fixed assembly code would be like:

```
.ORIG x3000
LD R1, DIVIDEND
LOOP JSR DIVIDE_BY_8
     AND R2, R1, #7
     ADD R1, R2, R4
     ADD R0, R1, #-7
BRp LOOP
ADD R0, R1, #-7
BRn OK
    ADD R1, R1, #-7
OK HALT
DIVIDE_BY_8 AND R2, R2, #0
            AND R3, R3, #0
            AND R4, R4, #0
            ADD R2, R2, #1
            ADD R3, R3, #8
            NEXT_BIT AND R5, R3, R1
                BRz SKIP
                    ADD R4, R2, R4
                SKIP ADD R2, R2, R2
                ADD R3, R3, R3
            BRnp NEXT_BIT
RET
DIVIDEND .FILL x120
.END
```

According to the above assembly code, the modulo operation for a divisor of 7 is completed with the following steps.

1. Divide the dividend n by 8. Let the quotient be q and the remainder be r such that

$$n = 8q + r \equiv q + r \pmod 7.$$

2. Set n to $q + r$. Repeat the above step until $0 \leq n < 7$.

In the first step, the remainder r could be obtained through a single `AND` operation with a bitmask as `0000000000000111`, while the quotient q is calculated by a subroutine labeled `DIVIDE_BY_8`, which actually shifts the dividend n by 3 bits to the right. To be more specific, in the subroutine, `R2` represents the weight of the current bit, `R3` performs as a bitmask for the bit, and `R4` is used to store the result of shifting right, namely the quotient.

Results

Copy the content of *gen\5982\5982.txt* and replace the first line with `0010001000010101`, and the restored machine code would be:

```
0010001000010101
0100100000001000
0101010001100111
0001001010000100
0001000001111001
0000001111111011
0001000001111001
0000100000000001
0001001001111001
1111000000100101
0101010010100000
0101011011100000
0101100100100000
0001010010100001
0001011011101000
0101101011000001
0000010000000001
0001100010000100
0001010010000010
0001011011000011
0000101111111010
1100000111000000
0000000100100000
```

According to the analysis above, the 15 missing bits should be `011`, `111`, `011`, `011`, and `101`.

The code has been stored into *mod.txt*.