LC-3 Executor

1 Introduction

In this project, we need to write a program to execute LC-3 binary code via **C** or other high level programming language.

To write the LC-3 executor, we need to implement the instructions like “BR”, “ADD”, “LD”, “ST”, “JSR”, “AND”, “LDR”, “STR”, “NOT”, “LDI”, “STI”, “JMP”, “LEA”. The only TRAP isntruction we need to implement is HALT instruction, which can stop and exit our executor. Also, the privilege mode, ACV and instructions like RTI, 1101 is not required.

As for registers, the default values of all registers and memory locations are x7777. When the HALT instruction executed by the program, the value of R0,R1,R6 and R7 will remain unchanged, then the program should print the value of all registers.

2 Algorithm Specification

In our program we need to imitate the LC-3 machine to operates the binary code. Therefore, we also need to complete the Instruction Cycle, which includes “Fetch”, “Decode”, “Evaluate Address”, “Fetch operands”, “Execute” and “Store Result” six parts.

However, not all instructions will do all parts of the Instructon Cycle. Consequently, we divide the Instruction Cycle into three parts: “Fetch”, “Decode” and “Execute accordingly”.

2.1 Data Structure

To represent a instruction, we define a class called BinaryCode, in which we use a string to represent the binary code and a unsigned int to represent the address of the instruction. Also, there will be some function in the class.

```
1     class BinaryCode {
2         private:
3             string code;
4             unsigned short address;
5         public:
6             ......
7
8     };
9
```

Also, we use a vector to store the instructions inputted and use another vector to serve as a storage of data. Then an array of unsigned int is used to represent the register.

2.2 Algorithm

Firstly, we need to process the inputted instruction and store it.

```
        str $\leftarrow$ input
        beginAddr $\leftarrow$ str.transferToDigit()
        while(!EOF)
            str $\leftarrow$ input
            form BinaryCode by str
            store the BinaryCode

```

Then, the program will decode the instructions and process them one by one.

```
    for each instruction i in storage
        opcode = i.getOpcode();
        if (opcode == "0001")add(current_code, nzp_ref);
        if (opcode == "0101")m_and(current_code, nzp_ref);
        if (opcode == "0000")br(current_code, PC_ref, nzp_ref);
        if (opcode == "1100")jump(current_code, PC_ref);
        if (opcode == "0100")jsr(current_code, PC_ref);
        if (opcode == "0010")ld(current_code, PC_ref, nzp_ref);
        if (opcode == "1010")ldi(current_code, PC_ref, nzp_ref);
        if (opcode == "0110")ldr(current_code, nzp_ref);
        if (opcode == "1110")lea(current_code, PC_ref);
        if (opcode == "1001")m_not(current_code, nzp_ref);
        if (opcode == "1000")rti(current_code);
        if (opcode == "0011")st(current_code, PC_ref);
        if (opcode == "1011")sti(current_code, PC_ref);
        if (opcode == "0111")str(current_code);
        if (opcode == "1111")break;//halt

```

Finally, after the program stops, we need to output the values in the registers in the specific format.

```
    for each register r
        output $\leftarrow$ r.value

```

After knowing the framework of the program, what matters is the detail of the implementation of each instruction.

For "ADD" instruction, we initially need to check whether the 5th bit is 0 or 1. If the 5th bit is 0, we need to do $dr = sr1 + sr2$ and then do setcc(). If the 5th bit is 1, we need to do $dr = sr1 + imm5$ and then do setcc()

```
    if (instruction[10] == '0')
        dr $\leftarrow$ stringToDigit(instruction.substr(4, 3));
        sr1 $\leftarrow$ stringToDigit(instruction.substr(7, 3));
        sr2 $\leftarrow$ stringToDigit(instruction.substr(13, 3));
        mRegister[dr] $\leftarrow$ mRegister[sr1] + mRegister[sr2];
        setcc();
    else
        dr $\leftarrow$ stringToDigit(instruction.substr(4, 3));
        sr1 $\leftarrow$ stringToDigit(instruction.substr(7, 3));
        imm5 $\leftarrow$ stringToDigit(instruction.substr(11, 5));
        mRegister[dr] $\leftarrow$ mRegister[sr1] + signExtension5(imm5);
```

```
        setcc();

```

For "AND" instruction, we initially need to check whether the 5th bit is 0 or 1. If the 5th bit is 0, we need to do $dr = sr1 \& sr2$ and then do setcc(). If the 5th bit is 1, we need to do $dr = sr1 \& imm5$ and then do setcc()

```
    if (instruction[10] == '0')
        dr ← stringToDigit(instruction.substr(4, 3));
        sr1 ← stringToDigit(instruction.substr(7, 3));
        sr2 ← stringToDigit(instruction.substr(13, 3));
        mRegister[dr] ← mRegister[sr1] & mRegister[sr2];
        setcc();
    else
        dr ← stringToDigit(instruction.substr(4, 3));
        sr1 ← stringToDigit(instruction.substr(7, 3));
        imm5 ← stringToDigit(instruction.substr(11, 5));
        mRegister[dr] ← mRegister[sr1] & signExtension5(imm5);
        setcc();

```

For "BR" instruction, we use "100" represents "N" state, use "010" represents "Z" state and use "001" represents "P" state. So the only task is to do $nzp \& state$

```
        nzp ← stringToDigit(instruction.substr(4, 3));
        check ← state & nzp;
        if (check == 0) do nothing;
        else  change PC;

```

For "JMP" instruction, we just simply add the value of PC with the value in specific register.

For "JSR" instruction, we initially need to store the PC into memory $TEMP = PC$, then we check whether the 11th bit is 0 or 1. If the 11th bit is 0, we need to do $PC = BaseR$. If the 11th bit is 1, we need to do $PC = PC + PCoffset11$. Finally, we store the value of TEMP into the 7th register.

```
        temp ← PC;
        if (code.getCode()[4] == '0')
            PC ← mRegister[stringToDigit(instruction.substr(7, 3))];

        else
            PC ← PC signExtension11(stringToDigit(instruction.substr(5, 11)));
        mRegister[7] ← temp;

```

For store instruction like "ST", "STR" and "STI", we initially calculate the target address. If the target address is in the range of instruction, we just overwrite the memory for instructions, which is the vector called "instructions". However, if the target address is out of the range of instruction, it means we cannot find the memory in vector "instructions", so we need to use the vector "storage". We firstly search the vector "storage" for the memory that has the same address as the target address. If we find, then we just overwrite it. If we cannot find, we build a new memory and store it into vector "storage".

```
        sr ← stringToDigit(code.getCode().substr(4, 3));
        value ← mRegister[sr];
        calculate address;//target address
        if (address < beginAddr || address >= beginAddr + instructions.size())
```

```
            //out of range of instructions
            for each item in storage
                if (item.getAddress() == address)
                    insertToStorage(value);
        else
            //in the range of instructions
            instructions[address - beginAddr] ← value;

```

Just opposite to the store instructions, for the load instructions like "LD", "LDR" and "LDI", we initially calculate the target address. If the target address is in the range of instruction, we just read the memory for instructions, which is the vector called "instructions". However, if the target address is out of the range of instruction, it means we cannot find the memory in vector "instructions", so we need to use the vector "storage". We read the vector "storage" for the memory that has the same address as the target address. Finally, we setcc() according to the value we read from memory.

```
        dr ← stringToDigit(instruction.substr(4, 3));
        calculate address;//target address
        if (address < beginAddr || address >= beginAddr + instructions.size())
            //out of range of instructions
            for each item in storage
                if (item.getAddress() == address)
                    mRegister[dr] = stringToDigit(item.getCode());
        else
            //in the range of instructions
            mRegister[dr] = stringToDigit(instructions[address - beginAddr].getCode());
        setcc();

```

For "LEA" instruction, we just add the PC value with the offset9 and then put it into specific register.

For "Not" instruction, we just get the value, and then do $result = value^0 xFFFF$. Finally, we setcc acorrding to the result value;

3 essential parts of code

Fig 1 is the implement of decode part

Fig 2 is the implement of add

Fig 3 is the implement of and

Fig 4 is the implement of br

Fig 5 is the implement of jsr

Fig 6 is the implement of load

Fig 7 is the implement of not

Fig 8 is the implement of store

```
void run(void)
{
    BinaryCode current_code;
    unsigned short PC = beginAddr;
    unsigned short& PC_ref = PC;
    unsigned short nzp = 0;//use the 1\2\3 bit to indicate
    unsigned short& nzp_ref = nzp;
    while (true)
    {
        //fetch instruction
        current_code = instructions[PC - beginAddr];
        PC++;
        //decode and operate

        string opcode = current_code.getInstruction();
        if (opcode == "0001")add(current_code, nzp_ref);
        if (opcode == "0101")m_and(current_code, nzp_ref);
        if (opcode == "0000")br(current_code, PC_ref, nzp_ref);
        if (opcode == "1100")jump(current_code, PC_ref);
        if (opcode == "0100")jsr(current_code, PC_ref);
        if (opcode == "0010")ld(current_code, PC_ref, nzp_ref);
        if (opcode == "1010")ldi(current_code, PC_ref, nzp_ref);
        if (opcode == "0110")ldr(current_code, nzp_ref);
        if (opcode == "1110")lea(current_code, PC_ref);
        if (opcode == "1001")m_not(current_code, nzp_ref);
        if (opcode == "1000")rti(current_code);
        if (opcode == "0011")st(current_code, PC_ref);
        if (opcode == "1011")sti(current_code, PC_ref);
        if (opcode == "0111")str(current_code);
        if (opcode == "1111") {
            break;
        }//trap;
    }
}
```

Figure 1: Fig 1

```
void add(BinaryCode code, unsigned short& nzp)
{
    if (code.getCode()[10] == '0') {
        unsigned short dr, sr1, sr2;
        dr = stringToDigit(code.getCode().substr(4, 3));
        sr1 = stringToDigit(code.getCode().substr(7, 3));
        sr2 = stringToDigit(code.getCode().substr(13, 3));
        mRegister[dr] = mRegister[sr1] + mRegister[sr2];
        short check = (short)mRegister[dr];
        if (check > 0) nzp = 1;
        else if (check == 0) nzp = 2;
        else nzp = 4;
    }
    else {
        unsigned short dr, sr1, imm5;
        dr = stringToDigit(code.getCode().substr(4, 3));
        sr1 = stringToDigit(code.getCode().substr(7, 3));
        imm5 = stringToDigit(code.getCode().substr(11, 5));
        mRegister[dr] = mRegister[sr1] + signExtension5(imm5);
        short check = (short)mRegister[dr];
        if (check > 0) nzp = 1;
        else if (check == 0) nzp = 2;
        else nzp = 4;
    }
}
```

Figure 2: Fig 2

```
void m_and(BinaryCode code, unsigned short& nzp)
{
    if (code.getCode()[10] == '0') {
        unsigned short dr, sr1, sr2;
        dr = stringToDigit(code.getCode().substr(4, 3));
        sr1 = stringToDigit(code.getCode().substr(7, 3));
        sr2 = stringToDigit(code.getCode().substr(13, 3));
        mRegister[dr] = mRegister[sr1] & mRegister[sr2];
        short check = (short)mRegister[dr];
        if (check > 0) nzp = 1;
        else if (check == 0) nzp = 2;
        else nzp = 4;
    }
    else {
        unsigned short dr, sr1, imm5;
        dr = stringToDigit(code.getCode().substr(4, 3));
        sr1 = stringToDigit(code.getCode().substr(7, 3));
        imm5 = stringToDigit(code.getCode().substr(11, 5));
        mRegister[dr] = mRegister[sr1] & signExtension5(imm5);
        short check = (short)mRegister[dr];
        if (check > 0) nzp = 1;
        else if (check == 0) nzp = 2;
        else nzp = 4;
    }
}
```

Figure 3: Fig 3

```
void br(BinaryCode code, unsigned short& PC, unsigned short& nzp)
{
    unsigned short nzp_check = stringToDigit(code.getCode().substr(4, 3));
    unsigned short check = nzp_check & nzp;
    if (check == 0)return;
    else {
        PC = PC + signExtension9(stringToDigit(code.getCode().substr(7, 9)));
    }
}
```

Figure 4: Fig 4

```
void jsr(BinaryCode code, unsigned short& PC)
{
    unsigned short temp = PC;
    if (code.getCode()[4] == '0') {
        PC = mRegister[stringToDigit(code.getCode().substr(7, 3))];
    }
    else {
        PC += signExtension11(stringToDigit(code.getCode().substr(5, 11)));
    }
    //if (stringToDigit(code.getCode().substr(7, 3)) == 7)return;
    mRegister[7] = temp;
}
```

Figure 5: Fig 5

```
void ld(BinaryCode code, unsigned short& PC, unsigned short& nzp)
{
    unsigned short dr = stringToDigit(code.getCode().substr(4, 3));
    unsigned short address = PC + signExtension9(stringToDigit(code.getCode().substr(7, 9)));

    if (address < beginAddr || address >= beginAddr + instructions.size()) {
        //mRegister[dr] = stringToDigit(instructions[0].getCode());//dr = mem[PC+sext(offset9)]

        for (vector<BinaryCode>::iterator p = storage.begin(); p < storage.end(); p++) {
            if ((*p).getAddress() == address) {
                mRegister[dr] = stringToDigit((*p).getCode());//dr = mem[PC+sext(offset9)]
            }
        }

    }
    else {
        mRegister[dr] = stringToDigit(instructions[address - beginAddr].getCode());//dr = mem[PC+sext(offset9)]
    }
    short check = (short)mRegister[dr];
    if (check > 0) nzp = 1;
    else if (check == 0) nzp = 2;
    else nzp = 4;
}
```

Figure 6: Fig 6

```
void m_not(BinaryCode code, unsigned short& nzp)
{
    unsigned short dr = stringToDigit(code.getCode().substr(4, 3));
    unsigned short sr = stringToDigit(code.getCode().substr(7, 3));
    unsigned short content = mRegister[sr];
    content = content ^ 0xffff;
    mRegister[dr] = content;
    short check = (short)mRegister[dr];
    if (check > 0) nzp = 1;
    else if (check == 0) nzp = 2;
    else nzp = 4;
}
```

Figure 7: Fig 7

```
void st(BinaryCode code, unsigned short& PC)
{
    unsigned short sr = stringToDigit(code.getCode().substr(4, 3));
    unsigned short content = mRegister[sr];
    unsigned short address = PC + signExtension9(stringToDigit(code.getCode().substr(7, 9)));
    BinaryCode content_binary(content, address);
    if (address < beginAddr || address >= beginAddr + instructions.size()) {
        insertToStorage(content_binary);
    }
    else {
        instructions[address - beginAddr] = content_binary;
    }


}
```

Figure 8: Fig 8