LC-3 Executor

1.Algorithm

1. **Initialization:**
    - This phase is responsible for setting up the initial state of the simulation.
    - Memory, registers, program counter (PC), program status register (PSR), and instruction register (IR) are all initialized.
    - The initial value of PC is set to the specified ORIG address, indicating the starting point of program execution.
2. **Reading Instructions:**
    - The simulator reads LC-3 assembly instructions, which are provided either from an input file or through standard input (keyboard).Each 16-character binary string representing an instruction is converted into a 16-bit block using the `stob()` function. These converted instructions are then stored sequentially in memory, starting from the ORIG address.
    - ```
      block i = ORIG;
          while (~scanf("%s", p))
          {
              mem[i++] = stob(p);
          }
      ```
3. **Fetch Phase:**
    - In each iteration of the execution loop, an instruction is fetched from memory using the current value of PC (program counter).After fetching, the PC is incremented to point to the next instruction in memory.
    - ```
      IR = mem [PC ++]
      ```
4. **Decode and Execute**
    - The core of the simulation is an infinite loop, which continues fetching, decoding, and executing instructions until a trap is met.

```
### Pseudocode: ###
Initialize Memory, Registers, PC, PSR, IR
main(){
    boot()   // initial Mem with 0x7777
    readin() // read instructions from keyboard input;
    execute()// depends ont instructions inputed, execute them
    print final status()// just print all regs' data
}
boot(){
    set up mem[] with specific number (here is 0x7777)
    change Pc = Orig
    Psr = 0x8002
    Ir  = 0x7777
}
```

```
readin(){
    buffer p[]
    linecnt i = Orig
    while(~getline(p)) mem[i++] = p;
}
execute(){
    IR = mem[Pc++]  // fetch one instruction
    OP = IR >> IR   // get OPcode of now instruction
    switch (OP){    // decode instruction by OPcode
        case ADD:
            if(!bit(IR,5)) // IR[5] == 1 , then use ADD with reg
            DR = SR1 + SR2
            else // ADd with imm5
            DR = SR1 + imm5
            setcc();
            break;
        case AND:
            if(!bit(IR,5)) // IR[5] == 1  then use ADD with reg
            DR = SR1 & SR2
            else // ADd with imm5
            DR = SR1 & imm5
            setcc();
            break;
        case BR:            // check Ben by CC
            BEN = (bit(IR, 11) & bit(PSR, 2)) + (bit(IR, 10) & bit(PSR, 1)) +
(bit(IR, 9) & bit(PSR, 0));
            if(BEN) // if Ben == 1
                PC = PC + imm9
            break;
        case JMP:
            PC = SR1
            break;
        case JSR:
            if(bit(IR,11)){ // JSR
                reg[7] = PC; PC = PC + imm11
            }else{ // jsrr
                reg[7] = PC; PC = SR1
            }
            break;
        case LD:
            DR = MEM[PC + imm9]
            setcc();
            break;
        case LDI:
            DR = MEM{MEM[PC + imm9]}
            setcc();
            break;
        case LDR:
            DR = mem[IR +imm6];
            setcc();
            break;
        case LEA :
            DR = PC + imm9;
            break;
        case NOT:
```

```
            DR = ~SR1
            setcc();
            break;
        case RTI: // don't use
            break;
        case ST:
            mem[PC + imm9] = DR
            break;
        case STI:
            mem[mem[PC + imm9] = DR
            break;
        case STR:
            mem[SR1 + imm9)] = IR;
            break;
        case TRAP: // meet TRAP then finish program
            goto OUT;
            break;
        case RESERVED:
            break;
    }
}
```

2.Essential parts of code with sufficient comments

```
#define bit(num,id) ((num >> id) % 2)
#define ADD 1
.. omit some define like ADD 1
typedef unsigned short block;
FILE* fp;
// MASK used to get SR DR imm
enum Mask {
    DR_MASK = 0x0E00,
    SR1_MASK = 0x01C0,
    SR2_MASK = 0x0007,
    IMM5_MASK = 0x001F,
    OFFSET9_MASK = 0x01FF,
    OFFSET11_MASK = 0x07FF,
    OFFSET6_MASK = 0x003F,
    PSR_MASK = 0x8002
};

// mem size
#define MEM_SIZE 65024

// mem and reg here
block mem[MEM_SIZE];
block reg[8];

//Program counter, program status register, instruction register, initial
address, condition codes
block PC, PSR, IR, ORIG, CC;
```

```
// booting program
void boot() {
    for (block i = 0; i < MEM_SIZE; i++)
        mem[i] = 0x7777;    // set all mem 0x7777
    for (block i = 0; i < 8; i++)
        reg[i] = 0x7777;    // set all reg 0x7777
    PC = ORIG;              //  Pc init
    PSR = PSR_MASK;         // PSR init
    IR = 0x7777;            // IR init
}

// bin to Hex , ORIG position use it
block stob(char *p) {
    block sum = 0;
    for (block i = 0; i < 16; i++)
        sum += (p[i] - '0') * pow(2, 15 - i);
    return sum;
}

// read instructions
void readin(char *p, block ORIG) {
    block i = ORIG;
    while (~scanf("%s", p))
    {
        mem[i++] = stob(p);

    }
}

// Get DR
block get_dr(block num) {
    return (num & DR_MASK) >> 9;
}

// Get Sr1
block get_sr1(block num) {
    return (num & SR1_MASK) >> 6;
}

// Get Sr2
block get_sr2(block num) {
    return num & SR2_MASK;
}

// Get imm5
block get_imm5(block num) {
    block imm5 = num & IMM5_MASK;
    if (imm5&0x10)
        imm5 |= 0xFFE0;
    return imm5;
}

// get imm9
block get_offset9(block num) {
```

```
    block offset9 = num & OFFSET9_MASK;
    if (bit(offset9,8))
        offset9 |= 0xFE00;
    return offset9;
}

// get imm11
block get_offset11(block num) {
    block offset11 = num & OFFSET11_MASK;
    if (bit(offset11,10))
        offset11 |= 0xF800;
    return offset11;
}

// get imm6
block get_offset6(block num) {
    block offset6 = num & OFFSET6_MASK;
    if (bit(offset6,5))
        offset6 |= 0xFFC0;
    return offset6;
}

// set condition code by DR if necessary
void setcc() {
    int num = reg[get_dr(IR)];
    if (bit(num, 15)) PSR = 0x8004;
    else if (num) PSR = 0x8001;
    else PSR = 0x8002;
}
// decode and exectue
block BEN;
void execute() {
    while(1) // exectue until Meet Trap instruction
    {
        IR = mem[PC++];     // fetch one inst
        int OP = IR >> 12;  // get OPcode
        switch (OP) {       //execute instruction by OPcode and mem or Reg(if
used)
        case ADD:
            if(!bit(IR,5)) // ADD with reg
            reg[get_dr(IR)] = reg[get_sr1(IR)] + reg[get_sr2(IR)];
            else // ADd with imm5
            reg[get_dr(IR)] = reg[get_sr1(IR)] +  get_imm5(IR);
            setcc();
            break;
        case AND:
            if(!bit(IR,5)) // ADD with reg
            reg[get_dr(IR)] = reg[get_sr1(IR)] & reg[get_sr2(IR)];
            else // ADd with imm5
            reg[get_dr(IR)] = reg[get_sr1(IR)] & get_imm5(IR);
            setcc();
            break;
        case BR:
            BEN = (bit(IR, 11) & bit(PSR, 2)) + (bit(IR, 10) & bit(PSR, 1)) +
(bit(IR, 9) & bit(PSR, 0));
```

```
            if(BEN) PC = PC + get_offset9(IR); break;
        case JMP:
            PC = reg[get_sr1(IR)];          break;
        case JSR:
            if(bit(IR,11)){ // JSR
                reg[7] = PC;    PC = PC + get_offset11(IR);
            }else{ // jsrr
                reg[7] = PC;    PC = reg[get_sr1(IR)];
            } break;
        case LD:
            reg[get_dr(IR)] = mem[(block)(PC + get_offset9(IR))];
            setcc();  break;
        case LDI:
            reg[get_dr(IR)] = mem[mem[(block)(PC + get_offset9(IR))]];
            setcc();  break;
        case LDR:
            reg[get_dr(IR)] = mem[(block)(reg[get_sr1(IR)]+get_offset6(IR))];
            setcc();  break;
        case LEA :
            reg[get_dr(IR)] = PC + get_offset9(IR);  break;
        case NOT:
            reg[get_dr(IR)] = ~reg[get_sr1(IR)];
            setcc(); break;
        case RTI:
            break;
        case ST:
            mem[(block)(PC + get_offset9(IR))] = reg[get_dr(IR)];
            break;
        case STI:
            mem[mem[(block)(PC + get_offset9(IR))]] = reg[get_dr(IR)];
            break;
        case STR:
            mem[(block)(reg[get_sr1(IR)]+get_offset6(IR))] = reg[get_dr(IR)]
;break;
        case TRAP:
            goto OUT;break;
        case RESERVED:break;
        }
    }
    OUT:return;
}
int main() {
    char p[17];
    scanf("%s", p);
    ORIG = stob(p); // Read start Position
    boot();         // initial mem and regs
    readin(p, ORIG);// readin instructions
    execute();      // execute instrucions
    for (block i = 0; i < 8; i++) // print final status of all regs
    printf("R%u = x%04hX\n", i, reg[i]);
    return 0;
}
```