ICS Lab Report #6b

StuID: **Name:**

Problem Setting

At last we are expected to use any high-level programming language to execute LC-3 code. The implemented machine do not have any privilege level, thus related interrupts are also not required except a halt.

Algorithm Specification

We use C++.

Two functions are needed to simulate truncation and sign extension.

```
function trunc(val, l, r):
  return (val >> r)-(val >> l << (l - r))
function sigext(val, l, r):
  val = trunc(val, l, r)
  if l - r == 11 and trunc(val, 11, 10) == 1
    val = val | 0xf800
  if l - r == 9 and trunc(val, 9, 8) == 1
    val = val | 0xfe00
  if l - r == 6 and trunc(val, 6, 5) == 1
    val = val | 0xffc0
  if l - r == 5 and trunc(val, 5, 4) == 1
    val = val | 0xffe0
  return val
```

As seen, trunc() is used to get val[l:r], and sigext is extending an immediate number to 16-bit with its sign.

The main program is as follows; in which run() will be explained in implementing part.

```
input(ip)
addr = ip
while(input(memory[addr++]))
  nop
```

```
run()
output(regs)
```

C++ Implementation

Main program can be easily implemented so ignored.

The LC-3 data and instructions are represented by a struct holding a short (16-bit) integer val.

The hardware memory and registers are represented by several LC3 variables:

addr - The program's start address.

inst - The current instruction preceeding.

ip - current instruction address.

pc - next instruction address.

nzp - the current register value used for branch instructions.

reg(array) - size 8, representing the registers.

memory - a stl map from address to value representing the physical memory.

In construction all LC3's default value is set 0x7777.

Here we give out the run() function:

```
void run(){
    while(1){
        inst = memory[ip]; //get current instruction
        pc = ip.val + 1; //set increment pc
        short opcode = inst.trunc(16, 12);
        int dr = inst.trunc(12, 9);
        int sr1 = inst.trunc(9, 6);
        int usi = inst.trunc(6, 5);
        int sr2 = inst.trunc(3, 0);
        int brn = inst.trunc(12, 11);
        int brz = inst.trunc(11, 10);
        int brp = inst.trunc(10, 9);
        short imm11 = inst.sigext(11, 0);
        short imm9 = inst.sigext(9, 0);
        short imm6 = inst.sigext(6, 0);
        short imm5 = inst.sigext(5, 0);
        switch(opcode){
```

```
            case 0:{
                if(brn && nzp.val < 0) pc = pc.val + imm9;
                else if(brz && nzp.val == 0) pc = pc.val + imm9;
                else if(brp && nzp.val > 0) pc = pc.val + imm9;
                break;
            }//BR
            case 1:{
                if(usi) reg[dr] = reg[sr1].val + imm5; //using immediate
                else reg[dr] = reg[sr1].val + reg[sr2].val; //using register
                nzp = reg[dr]; //update NZP
                break;
            }//ADD
            case 2:{
                reg[dr] = memory[pc.val + imm9];
                nzp = reg[dr]; //update NZP
                break;
            }//LD
            case 3:{
                memory[pc.val + imm9] = reg[dr];
                break;
            }//ST
            case 4:{
                reg[7] = pc;
                if(brn) pc = pc.val + imm11; //JSR
                else pc = reg[sr1].val; //JSRR
                break;
            }//JSR
            case 5:{
                if(usi) reg[dr] = reg[sr1].val & imm5; //using immediate
                else reg[dr] = reg[sr1].val & reg[sr2].val; //using register
                nzp = reg[dr]; //update NZP
                break;
            }//AND
            case 6:{
                reg[dr] = memory[reg[sr1].val + imm6];
                nzp = reg[dr]; //update NZP
                break;
            }//LDR
            case 7:{
                memory[reg[sr1].val + imm6] = reg[dr].val;
                break;
            }//STR
            case 9:{
```

```
                reg[dr] = ~reg[sr1].val;
                nzp = reg[dr]; //update NZP
                break;
            }//NOT
            case 10:{
                reg[dr] = memory[memory[pc.val + imm9]];
                nzp = reg[dr]; //update NZP
                break;
            }//LDI
            case 11:{
                memory[memory[pc.val + imm9]] = reg[dr].val;
                break;
            }//STI
            case 12:{
                pc = reg[sr1];
                break;
            }//JMP
            case 14:{
                reg[dr] = pc.val + imm9;
                break;
            }//LEA
            case 15:{
                return;
            }//TRAP
        }
        ip = pc; //get next instruction address
    }
}
```