

```

/*****
 * James Small & William Elder
 * OS.h
 * 5/7/13
 * This program is the main OS class that uses an assembler and virtual machine
 * objects to run our programs. This class takes in the program files, converts
 * them to object code, loads them all into memory, and then proceeds to run them
 * all using context switches to allow all of the programs to run concurrently,
 * allowing each program to have access to the cpu at some point.
 *****/

#ifndef OS_H
#define OS_H

#include "Assembler.h"
#include "VirtualMachine.h"
#include "PCB.h"
#include <list>
#include <queue>
#include <string>
#include <cstring>

using namespace std;

class OS {
private:
    //variables
    Assembler as;
    VirtualMachine vm;
    list<PCB *> jobs;
    queue<PCB *> readyQ, waitQ;
    PCB * running;
    int masterClock, contextSwitchClock, idleClock;

    // internal functions
    void getPCBFromVM();
    void givePCBToVM();
    void outputErrorCode(string s);
    bool done();

public:
    void start();
    OS();
};

#endif

```

```

/*****
* James Small & William Elder
* OS.h
* 5/7/13
* This program is the main OS class that uses an assembler and virtual machine
* objects to run our programs. This class takes in the program files, converts
* them to object code, loads them all into memory, and then proceeds to run them
* all using context switches to allow all of the programs to run concurrently,
* allowing each program to have access to the cpu at some point.
*****/

```

```

#include "OS.h"

```

```

OS::OS()

```

```

{
    masterClock = contextSwitchClock = idleClock = 0;
}

```

```

void OS::start()

```

```

{
    // read in all s files

    system("ls *.s > progs");

    fstream readProgs;
    readProgs.open("progs", ios::in);

    if (!readProgs.is_open()) {
        cout << "Progs failed to open.\n";
        exit(1);
    }

    string line;
    getline(readProgs, line);

    while (!readProgs.eof()) {

        as.assemble(line); // assemble each file

        //create PCB for each file, put in jobs list, and put in ready queue

        PCB * p = new PCB;
        jobs.push_back(p);
        readyQ.push(p);
        p->state = "ready";

        string fileName = line.erase(line.length() - 2, 2);

        // set file names for input and output
        p->originalfilename = fileName + ".o";
        p->readfilename = fileName + ".in";
        p->writefilename = fileName + ".out";
        p->stfilename = fileName + ".st";

        // Open input and output file streams in PCB
        p->openReadInFileStream();
        p->openWriteOutFileStream();

        //read into memory, modify PCB with values for base and limit

        int base, limit;

        vm.loadIntoMemory(p->originalfilename, &base, &limit);

        p->base = base;
        p->limit = limit;
        p->pc = base;

        getline(readProgs, line);
    } // end reading into memory
}

```

```

readProgs.close();
system("rm progs");

// copy first item in readyQ to running , load pcb data to vm

running = readyQ.front();
readyQ.pop();
givePCBToVM();

int code = 0;

while (!done()) // main process loop
{
    code = 0;

    vm.run(); // runs current proceess in vm, returns error code stored for later

    // get return code from status register
    code = vm.sr;
    code = code & 0xe0;
    code >>= 5;

    // update clocks for context switch
    masterClock += 5;
    contextSwitchClock += 5;

    // update all processes that aren't terminated with addition to context switch time
    for (list<PCB *>::iterator it = jobs.begin(); it != jobs.end(); it++)
        if(!strcmp((*it)->state.c_str(), "terminated"))
            (*it)->contextSwitchTime += 5;

    // First item in main loop

    // move items from waitQ into readyQ if interrupt time hit
    if(!waitQ.empty()) {
        while (waitQ.front()->interruptTime <= masterClock) {
            PCB * temp = waitQ.front();
            waitQ.pop();
            readyQ.push(temp);
            temp->state = "ready";
            if(waitQ.empty())
                break;
        }
    }

    // Second item in main loop

    // Update waiting time for all items in ready queue with latest cycle time from vm.run()
    for (list<PCB *>::iterator it = jobs.begin(); it != jobs.end(); it++)
        if(strcmp((*it)->state.c_str(), "ready"))
            (*it)->waitingTime += vm.clock;

    if(!(running == NULL)) {

        // update master clock and cpu time
        masterClock += vm.clock;
        running->cpuTime += vm.clock;

        // main switch to handle return code, if terminated set, output errorcode.
        switch (code) {
            case 0:
                getPCBFromVM();
                readyQ.push(running);
                break;
            case 1:
                running->state = "terminated";
                break;
            case 2:
                running->state = "terminated";
                outputErrorCode("Out-of-Bound Reference");
                break;
        }
    }
}

```

```

case 3:
    running->state = "terminated";
    outputErrorCode("Stack Overflow");
    break;
case 4:
    running->state = "terminated";
    outputErrorCode("Stack Underflow");
    break;
case 5:
    running->state = "terminated";
    outputErrorCode("Invalid OPCode");
    break;
case 6:
{
    int readIn;

    running->readIntoRegister >> readIn;

    int readReg = vm.sr;

    readReg &= 0x300;
    readReg >>= 8;

    if(readReg < 0 || readReg > 3) {
        outputErrorCode("Invalid IO Register");
        break;
    }

    vm.reg[readReg] = readIn;

    masterClock++;
    running->ioTime += 27;
    running->cpuTime++;
    waitQ.push(running);
    running->state = "waiting";
    getPCBFromVM();
    running->interruptTime = masterClock + 28;

    break;
}

case 7:
{
    int writeToReg = vm.sr;

    writeToReg &= 0x300;

    writeToReg >>= 8;

    int temp = vm.reg[writeToReg] & 0x8000;

    if (temp)
        temp = vm.reg[writeToReg] | 0xffff0000;
    else
        temp = vm.reg[writeToReg] & 0xffff;

    running->writeToRegister << temp << endl;

    masterClock++;
    running->ioTime += 27;
    running->cpuTime++;

    waitQ.push(running);
    running->state = "waiting";
    getPCBFromVM();
    running->interruptTime = masterClock + 28;

    break;
}

default:
    outputErrorCode("Invalid Return Code");

```

```

        running->state = "terminated";
        break;
    }

    running = NULL;
} // end if running loop

// Third item in main loop

// if all items in waitQ, update masterclock to get first item out of waitQ
if(readyQ.empty() && !waitQ.empty()) {

    int temp = waitQ.front()->interruptTime - masterClock;

    masterClock += temp;
    idleClock += temp;
    readyQ.push(waitQ.front());
    waitQ.front()->state = "ready";
    waitQ.pop();

    // update idle time for all non terminated process in jobs list
    for (list<PCB *>::iterator it = jobs.begin(); it != jobs.end(); it++)
        if(!strcmp((*it)->state.c_str(), "terminated"))
            (*it)->idleTime += temp;
}

// move next process from readyQ into running state, copy over pcb contents
if(!readyQ.empty()) {
    running = readyQ.front();
    running->state = "running";
    readyQ.pop();
    givePCBToVM();
}

} // end main process loop

// run program accounting data

// calculate turnaround time for all processes
for (list<PCB *>::iterator it = jobs.begin(); it != jobs.end(); it++)
    (*it)->turnaroundTime = (*it)->cpuTime + (*it)->waitingTime + (*it)->ioTime + (*it)->
idleTime + (*it)->contextSwitchTime;

// run system accounting data

int systemTime = contextSwitchClock + idleClock; // calculate system time

double floatingPointMasterClock = masterClock; // calculate CPU Utilization

double cpuUtilization = (floatingPointMasterClock - idleClock) / floatingPointMasterClock;
// calculate CPU Utilization

int allCpuTime = 0;

// iterate through all processes to add cpuTime
for (list<PCB *>::iterator it = jobs.begin(); it != jobs.end(); it++)
    allCpuTime += (*it)->cpuTime;

double userCpuUtilization = allCpuTime / floatingPointMasterClock; // calculate user cpu
ulitizaton.

double throughPut = jobs.size() / (floatingPointMasterClock / 10000.0); // calculate
throughput

// iterate through pcb list and output throughput and vm utilization to all files

for (list<PCB *>::iterator it = jobs.begin(); it != jobs.end(); it++) {

    fstream writeAccounting;
    writeAccounting.open((*it)->writefilename.c_str(), ios::in | ios::out | ios::ate);

```

```

if (!writeAccounting.is_open()) {
    cout << (*it)->writefilename << " failed to open.\n";
    exit(1);
}

writeAccounting << "\nProcess Specific Accounting Data\n\n";
writeAccounting << "CPU Time = " << (*it)->cpuTime << " cycles\n";
writeAccounting << "Waiting Time = " << (*it)->waitingTime << " cycles\n";
writeAccounting << "Turnaround Time = " << (*it)->turnaroundTime << " cycles\n";
writeAccounting << "IO Time = " << (*it)->ioTime << " cycles\n";
writeAccounting << "Largest Stack Size = " << (*it)->largestStackSize << endl;

writeAccounting << "\nSystem Specific Accounting Data\n\n";
writeAccounting << "System Time = " << systemTime << " cycles\n";
writeAccounting << "System CPU Utilization = " << cpuUtilization * 100 << "%\n";
writeAccounting << "User CPU Utilization = " << userCpuUtilization * 100 << "%\n";
writeAccounting << "ThroughPut = " << throughPut << " processes per second\n";

writeAccounting.close();
}

```

```

// make sure all .st files are deleted.

```

```

system("rm *.st");

```

```

void OS::givePCBToVM()
{

```

```

    //copy contents of PCB running into VM, including stack information

```

```

    vm.pc = running->pc;
    vm.sr = running->sr;
    vm.sp = running->sp;
    vm.base = running->base;
    vm.limit = running->limit;

```

```

    for (int i = 0; i <= vm.REG_FILE_SIZE - 1; i++)
        vm.reg[i] = running->reg[i];

```

```

    if (running->sp < 256) {
        fstream readProgs;
        readProgs.open(running->stfilename.c_str(), ios::in);

```

```

        if (!readProgs.is_open()) {
            cout << running->stfilename << " failed to open.\n";
            exit(1);
        }

```

```

        int stackPointer = running->sp;

```

```

        string line;
        getline(readProgs, line);

```

```

        while (!readProgs.eof()) {
            vm.mem[stackPointer++] = atoi(line.c_str());
            getline(readProgs, line);
        }

```

```

    }

```

```

void OS::getPCBFromVM()
{

```

```

    //copy contents of VM into running PCB, including stack information

```

```

    running->pc= vm.pc;
    running->sr= vm.sr;
    running->sp=vm.sp;
    running->base= vm.base;

```

```

running->limit= vm.limit;

for (int i = 0;i <= vm.REG_FILE_SIZE - 1;i++) {
    running->reg[i] = vm.reg[i];
}

if(running->sp < 256)
{
    fstream readProgs;
    readProgs.open(running->stfilename.c_str(), ios::out);

    if (!readProgs.is_open()) {
        cout << running->stfilename << " failed to open.\n";
        exit(1);
    }

    int stackPointer = running->sp;

    int temp = 256 - stackPointer;

    if(running->largestStackSize < temp)
        running->largestStackSize = temp;

    while (stackPointer<256) {
        readProgs<<vm.mem[stackPointer++]<<endl;
    }
}

}

void OS::outputErrorCode(string s)
{
    fstream output;
    output.open(running->writefilename.c_str(), ios::in | ios::out | ios::ate);

    output << "\nError Code = " << s << endl << endl;
    output.close();
}

// check if all processes have been terminated
bool OS::done()
{
    if(readyQ.empty() && waitQ.empty() && !running)
        return true;

    return false;
}

// main to run OS
int main()
{
    OS os;
    os.start();
}

```

```

/*****
* James Small & William Elder
* PCB.h
* 5/7/13
* This is the process control block class to be used in our OS. We keep a list
* of pointers to these PCB's in order to hold information about all current
* running processes on the system. The information about the processes is copied
* to and from these PCB's in order to facilitate our os simulation.
*****/

#ifndef PCB_H
#define PCB_H

using namespace std;

class PCB {
private:
    int pc,sr,sp,base,limit;
    static const int REG_FILE_SIZE = 4;
    vector < int > reg;

    string readfilename, writefilename, originalfilename, stfilename, state;

    fstream readIntoRegister;
    fstream writeToRegister;

    int cpuTime, waitingTime, turnaroundTime, contextSwitchTime, idleTime,
    ioTime, largestStackSize, interruptTime;

public:
    PCB() {
        reg = vector <int> (REG_FILE_SIZE);
        pc = sr = cpuTime = waitingTime = turnaroundTime = 0;
        ioTime = largestStackSize = contextSwitchTime = idleTime = 0;
        for (int i = 0; i < REG_FILE_SIZE; i++) {
            reg[i] = 0;
        }
        sp = 256;
        state = "new";
    };

    void openReadInFileStream() {
        readIntoRegister.open(readfilename.c_str(), ios::in);

        if (!readIntoRegister.is_open()) {
            cout << readfilename << " failed to open.\n";
            exit(1);
        }
    }

    void openWriteOutFileStream() {
        writeToRegister.open(writefilename.c_str(), ios::out);

        if (!writeToRegister.is_open()) {
            cout << writefilename << " failed to open.\n";
            exit(1);
        }
    }

    ~PCB() {
        readIntoRegister.close();
        writeToRegister.close();
    }

    friend class OS;
};

#endif

```



```

/*****
 * James Small & William Elder
 * VirtualMachine.H
 * 4/17/13
 * This program simulates a 16 bit virtual machine.  It reads in machine code
 * created by the Assembler program, pulls them all into memory, and then executes
 * each instruction one at a time.
 * Modified 5/7/13: Add the ability to run multiple programs at a time.  Main run
 * function was modified to allow this.  Also added the ability to load programs
 * into memory before beginning to run the programs.
 *****/

```

```

#ifndef VIRTUALMACHINE_H
#define VIRTUALMACHINE_H

```

```

#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <vector>
#include <cstdlib>
#include <stdexcept>

```

```

using namespace std;

```

```

class format1 {
public:
    unsigned UNUSED:6;
    unsigned RS:2;
    unsigned I:1;
    unsigned RD:2;
    unsigned OP:5;
};

```

```

class format2 {
public:
    unsigned ADDR:8;
    unsigned I:1;
    unsigned RD:2;
    unsigned OP:5;
};

```

```

class format3 {
public:
    int CONST:8;
    unsigned I:1;
    unsigned RD:2;
    unsigned OP:5;
};

```

```

union instruction {
    int i;
    format1 f1;
    format2 f2;
    format3 f3;
};

```

```

class VirtualMachine {
private:
    //typedefs
    typedef void(VirtualMachine::*FP)();

    // Variables
    static const int REG_FILE_SIZE = 4;
    static const int MEM_SIZE = 256;
    vector < int > reg;
    vector < int > mem;
    instruction finalcode;
    int pc, ir, sr, sp, base, limit, clock, nextMemoryLocation;
    map<int, FP> instr;
    string inputFileName, readFileName, writeFileName;
    bool execute;

```

```
// Operations
```

```
void setCarryBit();
void setRightCarryBit();
void setOverflow();
int getCarryBit();
void setGreaterThan();
int getGreaterThan();
void resetGreaterThan();
void setLessThan();
int getLessThan();
void resetLessThan();
void setEqualTo();
int getEqualTo();
void resetEqualTo();
void returnCode(int i);
void readOrWrite(int i);
```

```
// Instruction Functions
```

```
void load();
void loadi();
void store();
void add();
void addi();
void addc();
void addci();
void sub();
void subi();
void subc();
void subci();
void _and();
void _andi();
void _xor();
void xori();
void _compl();
void shl();
void shla();
void shr();
void shra();
void compr();
void compri();
void getstat();
void putstat();
void jump();
void jumpl();
void jumpe();
void jumpg();
void call();
void _return();
void read();
void write();
void halt();
void noop();
```

```
public:
    VirtualMachine();
    void run();
    void loadIntoMemory(string inputFileName, int * progBase, int * progLimit);
    friend class OS;
};

#endif
```

```

/*****
* James Small & William Elder
* VirtualMachine.cpp
* 4/17/13
* This program simulates a 16 bit virtual machine. It reads in machine code
* created by the Assembler program, pulls them all into memory, and then executes
* each instruction one at a time.
* Modified 5/7/13: Add the ability to run multiple programs at a time. Main run
* function was modified to allow this. Also added the ability to load programs
* into memory before beginning to run the programs.
*****/

#include "VirtualMachine.h"

VirtualMachine::VirtualMachine()
{
    reg = vector<int> (REG_FILE_SIZE);
    mem = vector<int> (MEM_SIZE);
    execute = true;
    nextMemoryLocation = 0;

    instr[0] = &VirtualMachine::load;
    instr[1] = &VirtualMachine::store;
    instr[2] = &VirtualMachine::add;
    instr[3] = &VirtualMachine::addc;
    instr[4] = &VirtualMachine::sub;
    instr[5] = &VirtualMachine::subc;
    instr[6] = &VirtualMachine::_and;
    instr[7] = &VirtualMachine::_xor;
    instr[8] = &VirtualMachine::_compl;
    instr[9] = &VirtualMachine::shl;
    instr[10] = &VirtualMachine::shla;
    instr[11] = &VirtualMachine::shr;
    instr[12] = &VirtualMachine::shra;
    instr[13] = &VirtualMachine::compr;
    instr[14] = &VirtualMachine::getstat;
    instr[15] = &VirtualMachine::putstat;
    instr[16] = &VirtualMachine::jump;
    instr[17] = &VirtualMachine::jumpL;
    instr[18] = &VirtualMachine::jumpe;
    instr[19] = &VirtualMachine::jumpg;
    instr[20] = &VirtualMachine::call;
    instr[21] = &VirtualMachine::_return;
    instr[22] = &VirtualMachine::read;
    instr[23] = &VirtualMachine::write;
    instr[24] = &VirtualMachine::halt;
    instr[25] = &VirtualMachine::noop;
}

void VirtualMachine::loadIntoMemory(string inputFileName, int * progBase, int * progLimit)
{
    // read into memory, increase limit
    fstream readIntoMemory;
    readIntoMemory.open(inputFileName.c_str(), ios::in);

    if (!readIntoMemory.is_open()) {
        cout << inputFileName << " failed to open.\n";
    }

    *progBase = nextMemoryLocation;

    string line;
    getline(readIntoMemory, line);

    while (!readIntoMemory.eof()) {
        mem[nextMemoryLocation++] = atoi(line.c_str());
        getline(readIntoMemory, line);
    }

    *progLimit = nextMemoryLocation - *progBase;

    readIntoMemory.close();
}

```

```

}

void VirtualMachine::run() {

    clock = 0;
    execute = true;
    sr &= 0xfffffffff;

    // execute loop
    while (execute) {
        ir = mem[pc++];
        finalcode.i = ir;
        (this->*instr[finalcode.fl.OP])();
        if(clock >= 15)
            execute = false;
    }

} // run function

void VirtualMachine::setCarryBit()
{
    int temp = reg[finalcode.fl.RD] & 0x10000; // get 17th bit only
    temp = temp >> 17; // right shift 17

    if (sr & 0x1) // check if sr carry bit is 1, if so, subtract 1 from it to reset to 0
        sr -= 1;

    sr += temp; // add current carry value into sr
}

void VirtualMachine::setRightCarryBit()
{
    if (sr & 0x1) // check if sr carry bit is 1, if so, subtract 1 from it to reset to 0
        sr -= 1;

    sr += 1; // add current carry value into sr
}

int VirtualMachine::getCarryBit()
{
    return (sr & 0x1);
}

void VirtualMachine::setOverflow()
{
    if (!(sr & 0x10))
        sr += 16;
}

void VirtualMachine::setGreaterThan()
{
    if (!(sr & 0x2))
        sr += 2;
}

int VirtualMachine::getGreaterThan()
{
    int temp = sr & 0x2;
    temp = temp >> 1;
    return temp;
}

void VirtualMachine::resetGreaterThan()
{
    sr = sr & 0xfffffffff;
}

void VirtualMachine::setLessThan()
{
    if (!(sr & 0x8))
        sr += 8;
}

```

```

int VirtualMachine::getLessThan()
{
    int temp = sr & 0x8;
    temp = temp >> 3;
    return temp;
}

void VirtualMachine::resetLessThan()
{
    sr = sr & 0xffffffff7;
}

void VirtualMachine::setEqualTo()
{
    if (!(sr & 0x4))
        sr += 4;
}

int VirtualMachine::getEqualTo()
{
    int temp = sr & 0x4;
    temp = temp >> 2;
    return temp;
}

void VirtualMachine::resetEqualTo()
{
    sr = sr & 0xffffffffb;
}

void VirtualMachine::load() // r[RD] = mem[ADDR]
{
    if (finalcode.f2.I == 0) {
        if (finalcode.f2.ADDR + base >= base + limit || finalcode.f2.ADDR + base < base) {
            returnCode(2);
            execute = false;
            return;
        }

        clock += 4;

        reg[finalcode.f2.RD] = mem[finalcode.f2.ADDR + base];
    }
    else
        loadi();
}

void VirtualMachine::loadi() // r[RD] = CONST
{
    clock += 1;

    reg[finalcode.f3.RD] = finalcode.f3.CONST;
}

void VirtualMachine::store() // mem[ADDR] = r[RD]
{
    if (finalcode.f2.ADDR + base >= base + limit || finalcode.f2.ADDR + base < base) {
        returnCode(2);
        execute = false;
        return;
    }

    clock += 4;

    mem[finalcode.f2.ADDR + base] = reg[finalcode.f2.RD];
}

```

```

void VirtualMachine::add() // r[RD] = r[RD] + r[RS]
{
    clock += 1;

    if (finalcode.f1.I == 0) {
        int temp = reg[finalcode.f1.RD] + reg[finalcode.f1.RS];

        if (reg[finalcode.f1.RD] >= 0 && reg[finalcode.f1.RS] >= 0 && temp < 0)
            setOverflow();
        else if (reg[finalcode.f1.RD] < 0 && reg[finalcode.f1.RS] < 0 && temp >= 0)
            setOverflow();

        reg[finalcode.f1.RD] = temp;
        setCarryBit();
    }
    else
        addi();
}

void VirtualMachine::addi() // r[RD] = r[RD] + CONST
{
    clock += 1;

    int temp = reg[finalcode.f3.RD] + finalcode.f3.CONST;

    if (reg[finalcode.f3.RD] >= 0 && finalcode.f3.CONST >= 0 && temp < 0)
        setOverflow();
    else if (reg[finalcode.f3.RD] < 0 && finalcode.f3.CONST < 0 && temp >= 0)
        setOverflow();

    reg[finalcode.f3.RD] = temp;
    setCarryBit();
}

void VirtualMachine::addc() // r[RD] = r[RD] + r[RS] + CARRY
{
    clock += 1;

    if (finalcode.f1.I == 0) {
        int carry = getCarryBit();
        int temp = reg[finalcode.f1.RD] + reg[finalcode.f1.RS] + carry;

        if (reg[finalcode.f1.RD] >= 0 && reg[finalcode.f1.RS] >= 0 && carry >= 0 && temp < 0)
            setOverflow();
        else if (reg[finalcode.f1.RD] < 0 && reg[finalcode.f1.RS] < 0 && carry < 0 && temp >= 0)
            setOverflow();

        reg[finalcode.f1.RD] = temp;
        setCarryBit();
    }
    else
        addci();
}

void VirtualMachine::addci() // r[RD] = r[RD] + CONST + CARRY
{
    clock += 1;

    int carry = getCarryBit();
    int temp = reg[finalcode.f3.RD] + finalcode.f3.CONST + carry;

    if (reg[finalcode.f3.RD] >= 0 && finalcode.f3.CONST >= 0 && carry >= 0 && temp < 0)
        setOverflow();
    else if (reg[finalcode.f3.RD] < 0 && finalcode.f3.CONST < 0 && carry < 0 && temp >= 0)
        setOverflow();
}

```

```

        reg[finalcode.f3.RD] = temp;
        setCarryBit();
    }

void VirtualMachine::sub() //  r[RD] = r[RD] - r[RS]
{
    clock += 1;

    if (finalcode.f1.I == 0) {

        int temp = reg[finalcode.f1.RD] - reg[finalcode.f1.RS];
        int complRS = ~reg[finalcode.f1.RS] + 1;

        if (reg[finalcode.f1.RD] >= 0 && complRS >= 0 && temp < 0)
            setOverflow();
        else if (reg[finalcode.f1.RD] < 0 && complRS < 0 && temp >= 0)
            setOverflow();

        reg[finalcode.f1.RD] = temp;
        setCarryBit();
    }
    else
        subi();
}

void VirtualMachine::subi() //  r[RD] = r[RD] - CONST
{
    clock += 1;

    int temp = reg[finalcode.f1.RD] - finalcode.f3.CONST;
    int complCONST = ~finalcode.f3.CONST + 1;

    if (reg[finalcode.f1.RD] >= 0 && complCONST >= 0 && temp < 0)
        setOverflow();
    else if (reg[finalcode.f1.RD] < 0 && complCONST < 0 && temp >= 0)
        setOverflow();

    reg[finalcode.f1.RD] = temp;
    setCarryBit();
}

void VirtualMachine::subc() //  r[RD] = r[RD] - r[RS] - CARRY
{
    clock += 1;

    if (finalcode.f1.I == 0) {

        int temp = reg[finalcode.f1.RD] - reg[finalcode.f1.RS] - getCarryBit();
        int complRS = ~reg[finalcode.f1.RS] + 1;

        if (reg[finalcode.f1.RD] >= 0 && complRS >= 0 && temp < 0)
            setOverflow();
        else if (reg[finalcode.f1.RD] < 0 && complRS < 0 && temp >= 0)
            setOverflow();

        reg[finalcode.f1.RD] = temp;
        setCarryBit();
    }
    else
        subci();
}

void VirtualMachine::subci() //  r[RD] = r[RD] - CONST - CARRY
{
    clock += 1;

    int temp = reg[finalcode.f1.RD] - finalcode.f3.CONST - getCarryBit();
    int complRS = ~finalcode.f3.CONST + 1;

    if (reg[finalcode.f1.RD] >= 0 && complRS >= 0 && temp < 0)
        setOverflow();
    else if (reg[finalcode.f1.RD] < 0 && complRS < 0 && temp >= 0)

```

```

        setOverflow();

        reg[finalcode.f1.RD] = temp;
        setCarryBit();
    }

void VirtualMachine::_and() // r[RD] = r[RD] & r[RS]
{
    clock += 1;

    if (finalcode.f1.I == 0)
        reg[finalcode.f1.RD] = reg[finalcode.f1.RD] & reg[finalcode.f1.RS];
    else
        andi();
}

void VirtualMachine::andi() // r[RD] = r[RD] & CONST
{
    clock += 1;
    reg[finalcode.f1.RD] = reg[finalcode.f1.RD] & finalcode.f3.CONST;
}

void VirtualMachine::_xor() // r[RD] = r[RD] ^ r[RS]
{
    clock += 1;

    if (finalcode.f1.I == 0)
        reg[finalcode.f1.RD] = reg[finalcode.f1.RD] ^ reg[finalcode.f1.RS];
    else
        xori();
}

void VirtualMachine::xori() // r[RD] = r[RD] ^ CONST
{
    clock += 1;
    reg[finalcode.f1.RD] = reg[finalcode.f1.RD] ^ finalcode.f3.CONST;
}

void VirtualMachine::_compl() // r[RD] = ~ r[RD]
{
    clock += 1;
    reg[finalcode.f1.RD] = ~reg[finalcode.f1.RD];
}

void VirtualMachine::shl() // r[RD] = r[RD] << 1, shift-in-bit = 0
{
    clock += 1;
    reg[finalcode.f1.RD] = reg[finalcode.f1.RD] << 1;
    setCarryBit();
}

void VirtualMachine::shla() // shl arithmetic Set CARRY & Sign Ext
{
    clock += 1;
    if (reg[finalcode.f1.RD] < 0) {
        reg[finalcode.f1.RD] = reg[finalcode.f1.RD] << 1;
        reg[finalcode.f1.RD] = reg[finalcode.f1.RD] | 0xffff8000;
    }
    else
        reg[finalcode.f1.RD] = reg[finalcode.f1.RD] << 1;

    setCarryBit();
}

void VirtualMachine::shr() // r[RD] = r[RD] >> 1, shift-in-bit = 0
{
    clock += 1;
    int temp = reg[finalcode.f1.RD];
    temp = temp & 0x1;

```



```

        if (temp)
            setRightCarryBit();

        reg[finalcode.f1.RD] = reg[finalcode.f1.RD] >> 1;
    }

void VirtualMachine::shra() // shr arithmetic Set CARRY & Sign Ext
{
    clock += 1;
    int temp = reg[finalcode.f1.RD];
    temp = temp & 0x1;

    if (temp)
        setRightCarryBit();

    if (reg[finalcode.f1.RD] < 0) {
        reg[finalcode.f1.RD] = reg[finalcode.f1.RD] >> 1;
        reg[finalcode.f1.RD] = reg[finalcode.f1.RD] | 0xffff8000;
    }
    else
        reg[finalcode.f1.RD] = reg[finalcode.f1.RD] >> 1;
}

void VirtualMachine::compr()
{
    clock += 1;
    if (finalcode.f1.I == 0) {

        if (reg[finalcode.f1.RD] < reg[finalcode.f1.RS]) {
            setLessThan();
            resetEqualTo();
            resetGreaterThan();
        }
        else if (reg[finalcode.f1.RD] > reg[finalcode.f1.RS]) {
            setGreaterThan();
            resetEqualTo();
            resetLessThan();
        }
        else {
            setEqualTo();
            resetLessThan();
            resetGreaterThan();
        }
    }
    else
        compri();
}

void VirtualMachine::compri()
{
    clock += 1;
    if (reg[finalcode.f1.RD] < finalcode.f3.CONST) {
        setLessThan();
        resetEqualTo();
        resetGreaterThan();
    }
    else if (reg[finalcode.f1.RD] > finalcode.f3.CONST) {
        setGreaterThan();
        resetEqualTo();
        resetLessThan();
    }
    else if (reg[finalcode.f1.RD] == finalcode.f3.CONST) {
        setEqualTo();
        resetLessThan();
        resetGreaterThan();
    }
}

void VirtualMachine::getstat() // r[RD] = SR
{
    clock += 1;

```

```

        reg[finalcode.f1.RD] = sr;
    }

void VirtualMachine::putstat() // SR = r[RD]
{
    clock += 1;
    sr = reg[finalcode.f1.RD];
}

void VirtualMachine::jump() // pc = ADDR
{
    if (finalcode.f2.ADDR + base >= base + limit || finalcode.f2.ADDR + base < base) {
        returnCode(2);
        execute = false;
        return;
    }

    clock += 1;

    pc = finalcode.f2.ADDR + base;
}

void VirtualMachine::jump1() // if LESS == 1, pc = ADDR, else do nothing
{
    if (finalcode.f2.ADDR + base >= base + limit || finalcode.f2.ADDR + base < base) {
        returnCode(2);
        execute = false;
        return;
    }

    clock += 1;

    if(getLessThan())
        pc = finalcode.f2.ADDR + base;
}

void VirtualMachine::jumpe() // if EQUAL == 1, pc = ADDR, else do nothing
{
    if (finalcode.f2.ADDR + base >= base + limit || finalcode.f2.ADDR + base < base) {
        returnCode(2);
        execute = false;
        return;
    }

    if(getEqualTo())
        pc = finalcode.f2.ADDR + base;
}

void VirtualMachine::jumpg() // if GREATER == 1, pc = ADDR, else do nothing
{
    if (finalcode.f2.ADDR + base >= base + limit || finalcode.f2.ADDR + base < base) {
        returnCode(2);
        execute = false;
        return;
    }

    clock += 1;

    if(getGreaterThan())
        pc = finalcode.f2.ADDR + base;
}

void VirtualMachine::call() // push VM status; pc = ADDR
{
    clock += 4;

    if (sp < limit + 6) {
        returnCode(3);
        execute = false;
        return;
    }
}

```

```

        mem[--sp] = sr;
        mem[--sp] = reg[0];
        mem[--sp] = reg[1];
        mem[--sp] = reg[2];
        mem[--sp] = reg[3];
        mem[--sp] = pc;

        pc = finalcode.f2.ADDR;
    }

void VirtualMachine::_return() // pop and restore VM status
{
    clock += 4;

    if (sp==256){
        returnCode(4);
        execute=false;
        return;
    }

    pc = mem[sp++];
    reg[3] = mem[sp++];
    reg[2] = mem[sp++];
    reg[1] = mem[sp++];
    reg[0] = mem[sp++];
    sr = mem[sp++];
}

void VirtualMachine::read() // read new content of r[RD] from .in file
{
    returnCode(6);
    readOrWrite(finalcode.f1.RD);
    execute = false;
}

void VirtualMachine::write() // write r[RD] into .out file
{
    returnCode(7);
    readOrWrite(finalcode.f1.RD);
    execute = false;
}

void VirtualMachine::halt() // halt execution
{
    clock += 1;
    returnCode(1);
    execute = false;
}

void VirtualMachine::noop() // no operation
{
    clock += 1;
}

void VirtualMachine::returnCode(int i)
{
    sr &= 0xfffffffff;

    int temp = i << 5;

    sr |= temp;
}

void VirtualMachine::readOrWrite(int i)
{
    sr &= 0xfffffcff;

    int temp = i << 8;

    sr |= temp;
}

```

```

/*****
 * James Small & William Elder
 * Assembler.h
 * 4/11/13
 * This program simulates an assembler, translating assembly instructions into
 * binary instruction format.
 *****/

```

```

#ifndef ASSEMBLER_H
#define ASSEMBLER_H

```

```

#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <map>
#include <cstdlib>
#include <stdexcept>

```

```

using namespace std;

```

```

class NullPointerException: public runtime_error {
public:
    NullPointerException(): runtime_error("Invalid Operand Read") { }
};

```

```

class Assembler{
private:

```

```

    //typedefs
    typedef void(Assembler::*FP)(string);

```

```

    // Utility Functions
    void checkRD(int RD);
    void checkRS(int RS);
    void checkADDR(int ADDR);
    void checkCONST(int CONST);
    void write(int objcode);

```

```

    // Instruction Functions
    void load(string s);
    void loadi(string s);
    void store(string s);
    void add(string s);
    void addi(string s);
    void addc(string s);
    void addci(string s);
    void sub(string s);
    void subi(string s);
    void subc(string s);
    void subci(string s);
    void _and(string s);
    void andi(string s);
    void _xor(string s);
    void xori(string s);
    void _compl(string s);
    void shl(string s);
    void shla(string s);
    void shr(string s);
    void shra(string s);
    void compr(string s);
    void compri(string s);
    void getstat(string s);
    void putstat(string s);
    void jump(string s);
    void jumpl(string s);
    void jumpe(string s);
    void jumpg(string s);
    void call(string s);
    void _return(string s);
    void read(string s);
    void write(string s);

```

```
void halt(string s);
void noop(string s);

//variables
string inputFileName, outputFileName;
map<string, FP> instr;

public:

    Assembler();
    void assemble(string filename);
};

#endif
```

```

/*****
* James Small & William Elder
* Assembler.cpp
* 4/11/13
* This program simulates an assembler, translating assembly instructions into
* binary instruction format.
*****/

```

```
#include "Assembler.h"
```

```
Assembler::Assembler()
```

```
{
    instr["load"] = &Assembler::load;
    instr["loadi"] = &Assembler::loadi;
    instr["store"] = &Assembler::store;
    instr["add"] = &Assembler::add;
    instr["addi"] = &Assembler::addi;
    instr["addc"] = &Assembler::addc;
    instr["addci"] = &Assembler::addci;
    instr["sub"] = &Assembler::sub;
    instr["subi"] = &Assembler::subi;
    instr["subc"] = &Assembler::subc;
    instr["subci"] = &Assembler::subci;
    instr["and"] = &Assembler::_and;
    instr["andi"] = &Assembler::_andi;
    instr["xor"] = &Assembler::_xor;
    instr["xori"] = &Assembler::_xori;
    instr["compl"] = &Assembler::_compl;
    instr["shl"] = &Assembler::shl;
    instr["shla"] = &Assembler::shla;
    instr["shr"] = &Assembler::shr;
    instr["shra"] = &Assembler::shra;
    instr["compr"] = &Assembler::compr;
    instr["compri"] = &Assembler::compri;
    instr["getstat"] = &Assembler::getstat;
    instr["putstat"] = &Assembler::putstat;
    instr["jump"] = &Assembler::jump;
    instr["jumpl"] = &Assembler::jumpl;
    instr["jumpe"] = &Assembler::jumpe;
    instr["jumpg"] = &Assembler::jumpg;
    instr["call"] = &Assembler::call;
    instr["return"] = &Assembler::_return;
    instr["read"] = &Assembler::read;
    instr["write"] = &Assembler::write;
    instr["halt"] = &Assembler::halt;
    instr["noop"] = &Assembler::noop;
}
```

```
void Assembler::assemble(string filename)
```

```
{
    string line, opcode;

    // Set File Names for Output and Input
    inputFileNames=filename;
    outputFileNames = filename.erase(filename.length() -2, 2) + ".o";

    // Create Output .o File
    fstream output;
    output.open(outputFileNames.c_str(), ios::out);
    output.close();

    // Open .s file for input
    fstream assemblyProg;
    assemblyProg.open(inputFileNames.c_str(), ios::in);

    if (!assemblyProg.is_open()) {
        cout << inputFileNames << " failed to open.\n";
        exit(1);
    }
}
```

```

getline(assemblyProg, line);

while (!assemblyProg.eof()) {

    if (line[0] == '!') {
        getline(assemblyProg, line);
        continue;
    }
    istringstream str(line.c_str());

    try {
        str >> opcode;
        if (not instr[opcode] )
            throw NullPointerException();
        else
            (this->*instr[opcode])(line.c_str());
    }
    catch (NullPointerException e) {
        cerr << e.what() << endl;
        exit(1);
    }
    getline(assemblyProg, line);
}
assemblyProg.close();
}

void Assembler::write(int objcode)
{
    fstream fout;
    fout.open(outputFileName.c_str(), ios::in | ios::out | ios::ate);

    if (!fout.is_open()) {
        cout << outputFileName << " failed to open.\n";
        exit(1);
    }
    fout << objcode << endl;
    fout.close();
}

void Assembler::checkRD(int RD)
{
    if (RD > 3 || RD < 0){
        cout << "RD value out of range" << endl;
        exit(1);
    }
}

void Assembler::checkRS(int RS)
{
    if (RS > 3 || RS < 0){
        cout << "RS value out of range" << endl;
        exit(1);
    }
}

void Assembler::checkADDR(int ADDR)
{
    if (ADDR > 255 || ADDR < 0) {
        cout << "ADDR value out of range" << endl;
        exit(1);
    }
}

void Assembler::checkCONST(int CONST)
{
    if (CONST > 127 || CONST < -128){
        cout << "CONST value out of range" << endl;
        exit(1);
    }
}

void Assembler::load(string s)

```

```

{
    int RD, ADDR;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> ADDR; // Don't need OPCODE

    checkRD(RD);
    checkADDR(ADDR);

    int finalnum = 0;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += ADDR;

    write(finalnum);
}

void Assembler::loadi(string s)
{
    int RD, CONST, I=1;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> CONST; // Don't need OPCODE

    checkRD(RD);
    checkCONST(CONST);

    int finalnum = 0;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;

    if (CONST < 0)
        CONST = CONST & 0x00FF;

    finalnum += CONST;

    write(finalnum);
}

void Assembler::store(string s)
{
    int RD, ADDR;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> ADDR; // Don't need OPCODE

    checkRD(RD);
    checkADDR(ADDR);

    int finalnum = 1;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += ADDR;

    write(finalnum);
}

void Assembler::add(string s)
{
    int RD, RS, I = 0;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> RS; // Don't need OPCODE

    checkRD(RD);
    checkRS(RS);

```



```

        int finalnum = 2;
        finalnum = finalnum << 11;
        finalnum += RD << 9;
        finalnum += I << 8;
        finalnum += RS << 6;

        write(finalnum);
    }

void Assembler::addi(string s)
{
    int RD, CONST, I = 1;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> CONST; // Don't need OPCODE

    checkRD(RD);
    checkCONST(CONST);

    int finalnum = 2;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;

    if (CONST < 0)
        CONST = CONST & 0x00FF;

    finalnum += CONST;

    write(finalnum);
}

void Assembler::addc(string s)
{
    int RD, RS, I = 0;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> RS; // Don't need OPCODE

    checkRD(RD);
    checkRS(RS);

    int finalnum = 3;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;
    finalnum += RS << 6;

    write(finalnum);
}

void Assembler::addci(string s)
{
    int RD, CONST, I = 1;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> CONST; // Don't need OPCODE

    checkRD(RD);
    checkCONST(CONST);

    int finalnum = 3;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;

    if (CONST < 0)
        CONST = CONST & 0x00FF;

```

```

        finalnum += CONST;

        write(finalnum);
    }

void Assembler::sub(string s)
{
    int RD, RS, I = 0;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> RS; // Don't need OPCODE

    checkRD(RD);
    checkRS(RS);

    int finalnum = 4;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;
    finalnum += RS << 6;

    write(finalnum);
}

void Assembler::subi(string s)
{
    int RD, CONST, I = 1;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> CONST; // Don't need OPCODE

    checkRD(RD);
    checkCONST(CONST);

    int finalnum = 4;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;

    if (CONST < 0)
        CONST = CONST & 0x00FF;

    finalnum += CONST;

    write(finalnum);
}

void Assembler::subc(string s)
{
    int RD, RS, I = 0;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> RS; // Don't need OPCODE

    checkRD(RD);
    checkRS(RS);

    int finalnum = 5;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;
    finalnum += RS << 6;

    write(finalnum);
}

void Assembler::subci(string s)
{
    int RD, CONST, I = 1;

```

```

    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> CONST; // Don't need OPCODE

    checkRD(RD);
    checkCONST(CONST);

    int finalnum = 5;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;

    if (CONST < 0)
        CONST = CONST & 0x00FF;

    finalnum += CONST;

    write(finalnum);
}

void Assembler::_and(string s)
{
    int RD, RS, I = 0;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> RS; // Don't need OPCODE

    checkRD(RD);
    checkRS(RS);

    int finalnum = 6;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;
    finalnum += RS << 6;

    write(finalnum);
}

void Assembler::_andi(string s)
{
    int RD, CONST, I = 1;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> CONST; // Don't need OPCODE

    checkRD(RD);
    checkCONST(CONST);

    int finalnum = 6;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;

    if (CONST < 0)
        CONST = CONST & 0x00FF;

    finalnum += CONST;

    write(finalnum);
}

void Assembler::_xor(string s)
{
    int RD, RS, I = 0;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> RS; // Don't need OPCODE

```

```

    checkRD(RD);
    checkRS(RS);

    int finalnum = 7;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;
    finalnum += RS << 6;

    write(finalnum);
}

void Assembler::xori(string s)
{
    int RD, CONST, I = 1;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD >> CONST; // Don't need OPCODE

    checkRD(RD);
    checkCONST(CONST);

    int finalnum = 7;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;

    if (CONST < 0)
        CONST = CONST & 0x00FF;

    finalnum += CONST;

    write(finalnum);
}

void Assembler::_compl(string s)
{
    int RD, I = 0;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD; // Don't need OPCODE

    checkRD(RD);

    int finalnum = 8;
    finalnum = finalnum << 11;
    finalnum += RD << 9;

    write(finalnum);
}

void Assembler::shl(string s)
{
    int RD, I = 0;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD; // Don't need OPCODE

    checkRD(RD);

    int finalnum = 9;
    finalnum = finalnum << 11;
    finalnum += RD << 9;

    write(finalnum);
}

void Assembler::shla(string s)

```

```

{
    istream stream(s.c_str());
    int RD, I=0;
    string OPCODE;
    stream>>OPCODE>>RD; // Don't need OPCODE
    checkRD(RD);
    int finalnum = 10;
    finalnum = finalnum<<11;
    finalnum += RD<<9;
    write(finalnum);
}

void Assembler::shr(string s)
{
    int RD;
    string OPCODE;

    istream stream(s.c_str());
    stream >> OPCODE >> RD; // Don't need OPCODE

    checkRD(RD);

    int finalnum = 11;
    finalnum = finalnum << 11;
    finalnum += RD << 9;

    write(finalnum);
}

void Assembler::shra(string s)
{
    int RD;
    string OPCODE;

    istream stream(s.c_str());
    stream >> OPCODE >> RD; // Don't need OPCODE

    checkRD(RD);

    int finalnum = 12;
    finalnum = finalnum << 11;
    finalnum += RD << 9;

    write(finalnum);
}

void Assembler::compr(string s)
{
    int RD, RS;
    string OPCODE;

    istream stream(s.c_str());
    stream >> OPCODE >> RD >> RS; // Don't need OPCODE

    checkRD(RD);
    checkRS(RS);

    int finalnum = 13;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += RS << 6;

    write(finalnum);
}

void Assembler::compri(string s)
{
    int RD, I = 1, CONST;
    string OPCODE;

    istream stream(s.c_str());
    stream >> OPCODE >> RD >> CONST; // Don't need OPCODE

```

```

    checkRD(RD);
    checkCONST(CONST);

    int finalnum = 13;
    finalnum = finalnum << 11;
    finalnum += RD << 9;
    finalnum += I << 8;

    if (CONST < 0)
        CONST = CONST & 0x00FF;

    finalnum += CONST;

    write(finalnum);
}

void Assembler::getstat(string s)
{
    int RD;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD; // Don't need OPCODE

    checkRD(RD);

    int finalnum = 14;
    finalnum = finalnum << 11;
    finalnum += RD << 9;

    write(finalnum);
}

void Assembler::putstat(string s)
{
    int RD;
    string OPCODE;
    istringstream stream(s.c_str());

    stream >> OPCODE >> RD; // Don't need OPCODE

    checkRD(RD);

    int finalnum = 15;
    finalnum = finalnum << 11;
    finalnum += RD << 9;

    write(finalnum);
}

void Assembler::jump(string s)
{
    int ADDR;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> ADDR; // Don't need OPCODE

    checkADDR(ADDR);

    int finalnum=16;
    finalnum = finalnum << 11;
    finalnum += ADDR;

    write(finalnum);
}

void Assembler::jumpl(string s)
{
    int ADDR;
    string OPCODE;

```

```

        istringstream stream(s.c_str());
        stream >> OPCODE >> ADDR; // Don't need OPCODE

        checkADDR(ADDR);

        int finalnum = 17;
        finalnum = finalnum << 11;
        finalnum += ADDR;

        write(finalnum);
    }

void Assembler::jumpe(string s)
{
    int ADDR;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> ADDR; // Don't need OPCODE

    checkADDR(ADDR);

    int finalnum = 18;
    finalnum = finalnum << 11;
    finalnum += ADDR;

    write(finalnum);
}

void Assembler::jumpg(string s)
{
    int ADDR;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> ADDR; // Don't need OPCODE

    checkADDR(ADDR);

    int finalnum = 19;
    finalnum = finalnum << 11;
    finalnum += ADDR;

    write(finalnum);
}

void Assembler::call(string s)
{
    int ADDR;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> ADDR; // Don't need OPCODE

    checkADDR(ADDR);

    int finalnum = 20;
    finalnum = finalnum << 11;
    finalnum += ADDR;

    write(finalnum);
}

void Assembler::_return(string s)
{
    int finalnum = 21;
    write(finalnum << 11);
}

void Assembler::read(string s)
{

```

```

    int RD;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD; // Don't need OPCODE

    checkRD(RD);

    int finalnum = 22;
    finalnum = finalnum << 11;
    finalnum += RD << 9;

    write(finalnum);
}

void Assembler::write(string s)
{
    int RD;
    string OPCODE;

    istringstream stream(s.c_str());
    stream >> OPCODE >> RD; // Don't need OPCODE

    checkRD(RD);

    int finalnum = 23;
    finalnum = finalnum << 11;
    finalnum += RD << 9;

    write(finalnum);
}

void Assembler::halt(string s)
{
    int finalnum = 24;
    write(finalnum << 11);
}

void Assembler::noop(string s)
{
    int finalnum = 25;
    write(finalnum << 11);
}

```



```

! main for factorial program
    loadi 0 1      ! line 0, R0 = fact(R1)
    read 1         ! input R1
    call 6         ! call fact
    load 0 33      ! receive result of fact
    write 0
    halt
! fact function
    compri 1 1     ! line 6
    jumpe 14       ! jump over the recursive call to fact if
    jump1 14       ! R1 is less than or equal 1
    call 16        ! call mult (R0 = R0 * R1)
    load 0 34      ! receive result of mult
    subi 1 1       ! decrement multiplier (R1) and multiply again
    call 6         ! call fact
    load 0 33
    store 0 33     ! line 14, return R0 (result of fact)
    return
! mult function
    loadi 2 8      ! line 16, init R2 (counter)
    loadi 3 0      ! init R3 (result of mult)
    shr 1          ! line 18 (loop), shift right multiplier set CARRY
    store 2 35     ! save counter
    getstat 2      ! to find CARRY's value
    andi 2 1
    compri 2 1
    jumpe 25       ! if CARRY==1 add
    jump 26        ! otherwise do nothing
    add 3 0
    shl 0          ! make multiplicand ready for next add
    load 2 35      ! restore counter
    subi 2 1       ! decrement counter
    compri 2 0     ! if counter > 0 jump to loop
    jumpg 18
    store 3 34     ! return R3 (result of mult)
    return
    noop          ! line 33, fact return value
    noop          ! line 34, mult return value
    noop          ! line 35, mult counter

```

257
45568
40966
33
47104
49152
27393
36878
34830
40976
34
8961
40966
33
2081
43008
1288
1792
23040
3107
29696
13569
27905
36889
32794
5632
18432
1059
9473
27904
38930
3618
43008
51200
51200
51200

Process Specific Accounting Data

CPU Time = 1057 cycles
Waiting Time = 3302 cycles
Turnaround Time = 4566 cycles
IO Time = 54 cycles
Largest Stack Size = 36

System Specific Accounting Data

System Time = 1233 cycles
System CPU Utilization = 99.3888%
User CPU Utilization = 73.0845%
ThroughPut = 13.0976 processes per second

```

! main for factorial program
    loadi 0 1      ! line 0, R0 = fact(R1)
    read 1         ! input R1
    call 6         ! call fact
    load 0 33      ! receive result of fact
    write 0
    halt

! fact function
    compri 1 1     ! line 6
    jumpe 14       ! jump over the recursive call to fact if
    jumpl 14       ! R1 is less than or equal 1
    call 16        ! call mult (R0 = R0 * R1)
    load 0 34      ! receive result of mult
    subi 1 1       ! decrement multiplier (R1) and multiply again
    call 6         ! call fact
    load 0 33      !
    store 0 33     ! line 14, return R0 (result of fact)
    return

! mult function
    loadi 2 8      ! line 16, init R2 (counter)
    loadi 3 0      ! init R3 (result of mult)
    shr 1          ! line 18 (loop), shift right multiplier set CARRY
    store 2 35     ! save counter
    getstat 2      ! to find CARRY's value
    andi 2 1
    compri 2 1
    jumpe 25       ! if CARRY==1 add
    jump 26        ! otherwise do nothing
    add 3 0
    shl 0          ! make multiplicand ready for next add
    load 2 35      ! restore counter
    subi 2 1       ! decrement counter
    compri 2 0     ! if counter > 0 jump to loop
    jumpg 18
    store 3 34     ! return R3 (result of mult)
    return
    noop          ! line 33, fact return value
    noop          ! line 34, mult return value
    noop          ! line 35, mult counter

```

257
45568
40966
33
47104
49152
27393
36878
34830
40976
34
8961
40966
33
2081
43008
1288
1792
23040
3107
29696
13569
27905
36889
32794
5632
18432
1059
9473
27904
38930
3618
43008
51200
51200
51200

fact2.o

-25216

fact2.out

Process Specific Accounting Data

CPU Time = 1471 cycles
Waiting Time = 3299 cycles
Turnaround Time = 4824 cycles
IO Time = 54 cycles
Largest Stack Size = 48

System Specific Accounting Data

System Time = 1233 cycles
System CPU Utilization = 99.3888%
User CPU Utilization = 73.0845%
ThroughPut = 13.0976 processes per second


```
loadi 0 0 ! i = 0
compri 0 6 ! 6 pairs to read
jumpe 9 ! i == 6 done
read 1
read 2
add 1 2
write 1
addi 0 1 ! i++
jump 1 ! loop again
halt
```

io.5

256
26886
36873
45568
46080
4736
47616
4353
32769
49152

i0.0

0 1 2 3 4 5 6 7 8 9 10 11

io.in

1
5
9
13
17
21

Process Specific Accounting Data

CPU Time = 58 cycles
Waiting Time = 2325 cycles
Turnaround Time = 3487 cycles
IO Time = 486 cycles
Largest Stack Size = 0

System Specific Accounting Data

System Time = 1233 cycles
System CPU Utilization = 99.3888%
User CPU Utilization = 73.0845%
ThroughPut = 13.0976 processes per second

```
read 0
loadi 1 -2
add 0 1      ! subtract 2 from value read
write 0
halt
```

sub.s

45056
1022
4160
47104
49152

Sub.0

Process Specific Accounting Data

CPU Time = 5 cycles
Waiting Time = 3219 cycles
Turnaround Time = 4416 cycles
IO Time = 54 cycles
Largest Stack Size = 0

System Specific Accounting Data

System Time = 1233 cycles
System CPU Utilization = 99.3888%
User CPU Utilization = 73.0845%
ThroughPut = 13.0976 processes per second


```
loadi 0 1    ! i = 1
loadi 1 0    ! sum = 0
read 2
compr 0 2
jumpe 8      ! done
add 1 0      ! sum += i
addi 0 1     ! i++
jump 3       ! loop again
write 1
halt
```

sum1.S

257
768
46080
26752
36872
4608
4353
32771
47616
49152

Suml. 0

1225

Suml.out

Process Specific Accounting Data

CPU Time = 251 cycles
Waiting Time = 3235 cycles
Turnaround Time = 4298 cycles
IO Time = 54 cycles
Largest Stack Size = 0

System Specific Accounting Data

System Time = 1233 cycles
System CPU Utilization = 99.3888%
User CPU Utilization = 73.0845%
ThroughPut = 13.0976 processes per second

```
loadi 0 1    ! i = 1
loadi 1 0    ! sum = 0
read 2
compr 0 2
jumpe 8      ! done
add 1 0      ! sum += i
addi 0 1     ! i++
jump 3       ! loop again
write 1
halt
```

sum2.S

257
768
46080
26752
36872
4608
4353
32771
47616
49152

Sum2.0

5050

Sum2.out

Process Specific Accounting Data

CPU Time = 506 cycles
Waiting Time = 3233 cycles
Turnaround Time = 4266 cycles
IO Time = 54 cycles
Largest Stack Size = 0

System Specific Accounting Data

System Time = 1233 cycles
System CPU Utilization = 99.3888%
User CPU Utilization = 73.0845%
ThroughPut = 13.0976 processes per second