**Programmers Guide**

Team IDK

CSE 560, Lab 1

July 12th, 2009

Primary author: Michael Irwin

Secondary authors: Dan Stottlemire

Jim Powers

Michael Pinneger

**Table of Contents**

1. Title Page

2. Table of Contents

3. Overview

4. Outline of the problem

5. Our approach

6. Data Object Library

7. Simulator Methods

8. - 11. Simulator Source Code

12. SummiX\_Machine Methods

13. - 17. SummiX\_Machine Source Code

18. Loader Methods

19. - 21. Loader Source Code

22. Interpreter Methods

23. - 25. Interpreter Source Code

26. Executer Methods

27. - 30. Executer Source Code

31. SummiX\_Utilities Methods

32. SummiX\_Utilities Source Code

33. Thoughts on design and possible improvements

**Overview**

The following is intended as an overview of the components of the SummiX\_Machine including its operations and functionality. The following material assumes previous knowledge of Java, memory management and assignment as well as general programming experience. You may assume that all of the material listed in this document is correct and that all of the components and methods found in each subsequent file is listed in it’s entirety. Within this document you will find an overview of the problem that our implementation of SummiX\_Machine is meant to solve, a listing of all methods and variables that are instantiated, as well as an English representation of all possible methods.

**The Problem**

*Using the machine definition and instruction set descriptions given in class, design, code, test, and document a program to simulate the execution of the abstract machine.*

Your program should have three major components: one (the “loader”) to put the input information into the data structures that represent the memory and registers of the machine, one (the “interpreter”) to simulate the operation of the machine as it executes each instruction, and one (the “simulator”) to act as the simulator’s user interface, displaying the state of the machine as appropriate and controlling execution.

**The loader** must detect and provide appropriate messages under the following error condition:

* invalid contents (i.e., illegal characters in an input record)
* other errors you identify

**The instruction interpreter** must be capable of executing each of the machine’s instructions as specified. The interpreter should not be capable of stopping execution, unless the HALT instruction is issued.

**The simulator** must halt instruction interpretation when a user-specified time limit has been exceeded. This time limit should be specified in terms of number of instructions executed. Some reasonable default for this time limit should be provided. In addition, you may also elect to have some I/O errors be fatal. Your simulator should have at least 3 modes: quiet, trace, and step. In quiet mode, it simulates the execution of the loaded program without interruption (unless the time limit is exceeded, of course).

In trace mode, it should generate a trace of execution including:

1. The state of the machine (memory page and registers) immediately after loading but before execution.

2. Each executed instruction, including the memory locations and registers affected or used.

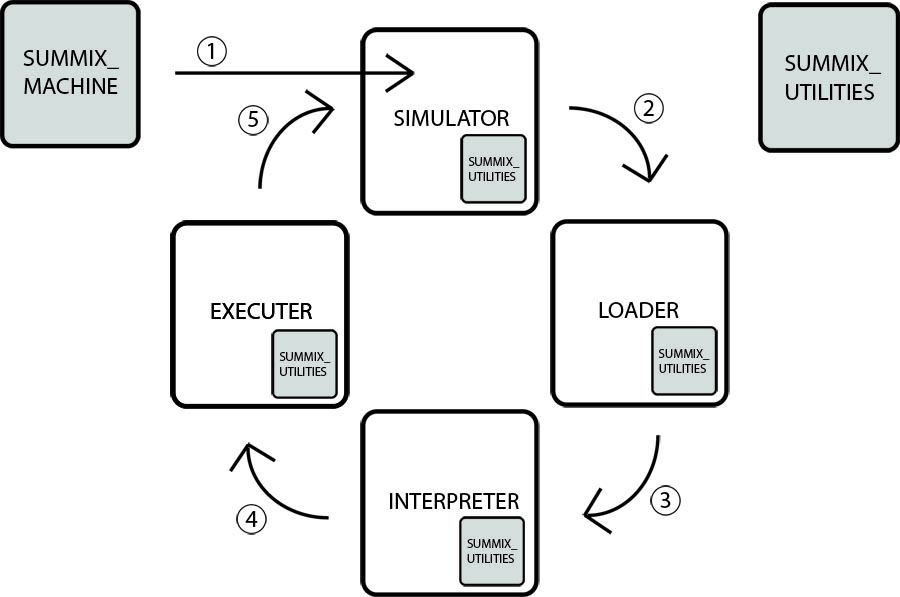
3. The state of the machine (memory page and registers) after execution.

Step mode should be similar to trace mode, except that the user is prompted before each step is taken. You may add other modes as you feel necessary.

*\* The above was supplied by the Lab 1 Documentation, found at:*

*http://www.cse.ohio-state.edu/~giles/560/handouts/simulator.pdf*

**Our Approach**



Background

When trying to decide which programming language and general development strategy our group should use, we looked at several different categories. Firstly, we realized that the lab tended toward one specific item that used different functions and operations in order to accomplish it’s goal. With that in mind, we decided that an object oriented style would prove most beneficial because we have the ability to directly organize and structure the program to have different classes that supported one overall “super class.” Secondly, we decided that due to the structure of the course, there was a likely chance that future labs would directly relate to the current problem which lead us to believe that a structure that was adaptable and easy to change would be the most effective.

Class Structure – Run Time

The concept behind our program is overall quite simple and effective. We decided to use a base “Simulator” class that is the engine for the rest of our code. This simulator is responsible for creating the machine, found in “SummiX\_Machine” and passing it to the loader. The loader is then responsible for loading the set of arrays from the information found in the input file. This loader then continues the process by sending the data to the interpreter. The interpreter is responsible for stripping the first four bits of the instruction and deciding which operation relates to those bits. That operations enumerated value is then sent to the executer. The executer is responsible for changing memory (the set of arrays) and follow the process of the specific instruction. The executer then returns to the simulator where the next instruction is passed to the interpreter, where the process continues until a HALT is discovered

**Simulator SummiX\_Machine Loader**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | Variables |   Int timeOutCounter  Int counter  String filename  BufferedReader br | |  | | --- | | Variables |   short[127][511] mem  short[] reg  short pc  BitSet ccr  int N  int Z  int P | |  | | --- | | Variables |   BufferedReader br  int init  int length  int returnVal |
| |  | | --- | | Methods |   getState(char c) | |  | | --- | | Methods |   randomizeMemory();  setMemory(short, short, short);  loadMemory(short, short);  getN();  getZ();  getP();  setPC(short);  incrementPC();  getPC();  setRegister(int, short);  setSubroutineReturn(short);  loadRegister(short);  outputMemoryPage(int);  outputMachineState(int); | |  | | --- | | Methods |   hexstringToInt(CharSequence);  getPage(int);  getOffset(int);  getHeader(SummiX\_Machine);  fillMemory(SummiX\_Machine);  Loader(String, SummiX\_Machine(); |

**Interpreter Executor SummiX\_Utilities**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  | | --- | | Variables |   int instruction  boolean halt  int bit  InstructionCode op;  short trap; | |  | | --- | | Variables |   short sr1, short sr1value  short sr2, short sr2value  short sr  short dr  short pg  short imm5  short baser  short pgoffset6  short index6  short pgoffset9 | |  | | --- | | Variables |   short s |
| |  | | --- | | Methods |   getInstruction(SummiX\_  Machine, short); | |  | | --- | | Methods |   Executer(SummiX\_Machine, short, InstructionCode); | |  | | --- | | Methods |   getBits(short, int, int);  getAbsoluteBits(short, int, int);  IntructionCode |

**Simulator**

private static Simulator\_State getState(char c) {}

/\*\*

\* Returns the mode that the user would like to operate in, QUIET, STEP, or TRACE

\*/

**Simulator Source Code**

package summixSimulator;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import summixSimulator.SummiX\_Utilities.Simulator\_State;

/\*\*

\* The SummiX simulator is the user interface that displays the

\* state of the machine as appropriate and controls execution.

\*

\* @author Mike/Mike/Dan/Jim

\*

\*/

public class Simulator {

/\*\*

\* @param args command line arguments args[0] - filename, arg[1] - mode of simulator (quiet, trace, or step)

\* @throws IOException

\*/

private static Simulator\_State getState(char c) {

switch (c) {

case 'q':

case 'Q':

return Simulator\_State.QUIET;

case 's':

case 'S':

return Simulator\_State.STEP;

case 't':

case 'T':

return Simulator\_State.TRACE;

default:

return Simulator\_State.ERROR;

}

}

public static void main(String[] args) throws IOException {

/\*\*

\* Main procedure of the simulator

\*

\* @param args filename mode timeout

\*/

Simulator\_State simState = Simulator\_State.ERROR;

int timeOutCounter = 1000;

int counter = 0;

String fileName = "input.txt";

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

SummiX\_Machine machine = new SummiX\_Machine();

//Length of args array indicates args entered (filename == 1, running mode == 2, and timeout == 3)

//ADD ERROR CHECKING ON ARGS LENGTH HERE

//If (they've entered the filename arg)

if (args.length > 0)

{

fileName = args[0];

//new fileName = filename arg

}

else //else prompt for file name

{

System.out.print("Please enter the input file's name: ");

fileName = br.readLine();

}

new Loader(fileName, machine);

//If (they've entered the running mode arg)

if (args.length > 1)

{

//Use the mode arg to set the simState

simState = getState(args[1].charAt(0));

}

else

{

while (simState==Simulator\_State.ERROR) {

//else prompt for running mode

System.out.print("Please enter the simulator mode ([q]uiet, [s]tep, or [t]race: ");

//maybe change this to produce an error if in.next().Length() > 1 ?

try {

simState = getState(br.readLine().charAt(0));

} catch (StringIndexOutOfBoundsException e) {

System.out.println("Default to quiet."); //for testing purposes, we can just make it re-loop for real or set to something else

simState = Simulator\_State.QUIET;

}

}

}

//If (they've entered the timeout arg)

if (args.length > 2)

{

//Use the timeout arg to set the timeOutCounter

timeOutCounter = Integer.valueOf(args[2], 10).intValue();

}

else

{

//else prompt for timeOutCounter (default = 1000)

System.out.print("Please enter the timeout value [press 'ENTER' for default (1000)] : ");

String temp = br.readLine();

if (temp.length() > 0) {

timeOutCounter = Integer.valueOf(temp, 10).intValue();

}

}

machine.setSimState(simState);

//for STEP or TRACE need to print initial values of the machine registers and page of memory

if ((simState == Simulator\_State.STEP) || (simState == Simulator\_State.TRACE)) {

machine.outputMemoryPage(machine.getPage());

machine.outputMachineState();

System.out.println("\n---");

}

while ((!Interpreter.getInstruction(machine, machine.loadMemory(SummiX\_Utilities.getBits(machine.getPC(), 0, 7), SummiX\_Utilities.getBits(machine.getPC(),7,9))))

&& (counter < timeOutCounter)) {

if ((simState == Simulator\_State.STEP) || (simState == Simulator\_State.TRACE)) {

System.out.print("---");

}

if (simState==Simulator\_State.STEP) {

//require user input in step mode between instructions

br.readLine();

}

counter++;

}

if (counter==timeOutCounter) {

System.out.println("\nSystem error: instruction limit exceeded!");

System.exit(-1);

}

if (simState == Simulator\_State.STEP) {

System.out.println("Press enter to continue.");

br.readLine();

machine.outputMemoryPage(machine.getPage());

machine.outputMachineState();

} else if (simState == Simulator\_State.TRACE) {

machine.outputMemoryPage(machine.getPage());

machine.outputMachineState();

}

}

}

**SummiX\_Machine**

private void randomizeMemory() {}

/\*\*

\* Randomize memory which maybe useful for debugging later

\*/

public void setMemory(short page, short offset, short data) {}

/\*\*

\* Sets memory at mem[page][offset]

\*

\* @param page the page of memory

\* @param offset the offset within the page

\* @param data the data to store

\*/

public short loadMemory(short page, short offset) {}

/\*\*

\* Get data from mem[page][offset] and return it

\*

\* @param page the page of memory

\* @param offset the offset within the page

\* @return data the value stored at the desired location in memory

\*/

public boolean getN() {} public boolean getZ() {} public boolean getP() {}

// Get CCR “N” bit // Get CCR “Z” bit // Get CCR “P” bit

public void setPC(short addr) {}

/\*\*

\* Sets the PC to addr

\*

\* @param addr address to be written to PC

\*/

public void incrementPC() {}

// increments the PC

public short getPC() {}

public void setRegister(int register, short data) {}

/\*\*

\* Sets the given register with given data and updates the

\* CCR accordingly

\*

\* @param register the register to be set

\* @param data the data to store in the register

\*/

public void setSubroutineReturn(short addr) {}

/\*\*

\* For the special case of setting register 7 without changing the CCR

\* as for JSR/JSRR with the link bit set to 1.

\*

\* @param addr the return address to store in register 7 \*/

**SummiX\_Machine Source Code**

package summixSimulator;

import java.util.BitSet;

import java.util.Random;

import summixSimulator.SummiX\_Utilities.Simulator\_State;

/\*\*

\* SummiX\_Machine consists of:

\* Memory (mem): Memory is word addressable. There are

\* 2^16 words of memory, addresses 0-65,535. Memory is

\* organized into pages of 512 words. There are 128 pages.

\* A memory address is given by a 16-bit quantity where the

\* upper 7 bits denote the page and the lower 9 bits denote

\* the offset within that page.

\*

\* Registers (reg and pc):

\* There are 8 general purposes registers (R0-R7) and

\* one program counter (PC).

\*

\* Condition Code Registers (ccr):

\* The CCR contains 3 bits, N, Z, and P. They are all

\* updated (set or cleared) every time a value is written

\* to a general purpose register except for JSR/JSRR

\* instructions. The N bit is set to 1 iff the last

\* value written to the register was negative, Z is set

\* to 1 if the last value was 0, and P if it was positive.

\*/

public class SummiX\_Machine {

private short[][] mem = new short[128][512]; //array to represent memory (0-127 pages, 0-511 words per page)

private short[] reg = {0,0,0,0,0,0,0,0}; //initialize all registers to 0

private short pc = 0; //program counter starts at 0

private BitSet ccr = new BitSet(3); //N,Z,P = 0,1,0 (all registers are set to 0) initially

private final int N = 0, Z = 1, P = 2;

private Simulator\_State simState;

private void randomizeMemory() {

/\*\*

\* Randomize memory which maybe useful for debugging later

\*/

for (int i = 0; i < 128; i++) {

for (int j = 0; j < 512; j++) {

Random randomNumbers = new Random();

mem[i][j] = (short) randomNumbers.nextInt(); //random int within the range of 16 bits

}

}

}

public SummiX\_Machine() {

randomizeMemory();

this.ccr.set(Z);

}

public void setSimState(Simulator\_State simState) {

this.simState = simState;

}

public void setMemory(short page, short offset, short data) {

/\*\*

\* Sets memory at mem[page][offset]

\*

\* @param page the page of memory

\* @param offset the offset within the page

\* @param data the data to store

\*/

this.mem[page][offset] = data;

if ((simState == Simulator\_State.STEP) || (simState == Simulator\_State.TRACE)) {

System.out.println("M[" + page + "][" + offset + "] = " + SummiX\_Utilities.shortToHexString(data));

}

}

public short loadMemory(short page, short offset) {

/\*\*

\* Get data from mem[page][offset] and return it

\*

\* @param page the page of memory

\* @param offset the offset within the page

\* @return data the value stored at the desired location in memory

\*/

short data = this.mem[page][offset];

if ((simState == Simulator\_State.STEP) || (simState == Simulator\_State.TRACE)) {

System.out.println("M[" + page + "][" + offset + "] = " + SummiX\_Utilities.shortToHexString(data));

}

return data;

}

public boolean getN() {

return this.ccr.get(N);

}

public boolean getZ() {

return this.ccr.get(Z);

}

public boolean getP() {

return this.ccr.get(P);

}

public void setPC(short addr) {

/\*\*

\* Sets the PC to addr

\*

\* @param addr address to be written to PC

\*/

if ((simState == Simulator\_State.STEP) || (simState == Simulator\_State.TRACE)) {

System.out.println("PC = " + SummiX\_Utilities.shortToHexString(addr));

}

this.pc = addr;

}

public void incrementPC() {

/\*\*

\* Increments the PC

\*/

short page = SummiX\_Utilities.getBits(this.pc, 0, 7);

short offset = SummiX\_Utilities.getBits(this.pc, 8, 9);

if (offset==511) {

page = (short) (((short) (page >>> 9) + 1) << 9);

this.pc = page;

} else {

this.pc++;

}

}

public short getPC() {

return this.pc;

}

public int getPage() {

return SummiX\_Utilities.getBits(this.pc, 0, 7);

}

public int getOffset() {

return SummiX\_Utilities.getBits(this.pc, 7, 9);

}

public void setRegister(int register, short data) {

/\*\*

\* Sets the given register with given data and updates the

\* CCR accordingly

\*

\* @param register the register to be set

\* @param data the data to store in the register

\*/

//store data into register

this.reg[register] = data;

//always update CCR

this.ccr.clear();

if (data < 0) {

this.ccr.set(N);

} else if (data == 0) {

this.ccr.set(Z);

} else {

this.ccr.set(P);

}

if ((simState == Simulator\_State.STEP) || (simState == Simulator\_State.TRACE)) {

System.out.println("R" + register + " = " + SummiX\_Utilities.shortToHexString(data));

}

}

public void setSubroutineReturn(short addr) {

/\*\*

\* For the special case of setting register 7 without changing the CCR

\* as for JSR/JSRR with the link bit set to 1.

\*

\* @param addr the return address to store in register 7

\*/

this.reg[7] = addr;

if ((simState == Simulator\_State.STEP) || (simState == Simulator\_State.TRACE)) {

System.out.println("R7 = " + SummiX\_Utilities.shortToHexString(addr));

}

}

public short loadRegister(int i) {

/\*\*

\* Loads data from a register

\*

\* @param register the register to load data from

\* @return data the value stored in the specified register

\*/

if ((simState == Simulator\_State.STEP) || (simState == Simulator\_State.TRACE)) {

System.out.println("R" + i + " = " + SummiX\_Utilities.shortToHexString(this.reg[i]));

}

return this.reg[i];

}

public void outputMemoryPage(int page) {

System.out.println("Memory Page: " + page);

for (int i = 0; i < 512; i++) {

short data = mem[page][i];

String output = SummiX\_Utilities.shortToHexString(data);

System.out.print("|" + i + ": " + output + "\t");

if ((i % 9 == 0) && (i > 0)) {

System.out.println();

}

}

System.out.println();

}

public void outputMachineState() {

System.out.println();

for (int i=0;i < 8;i++) { //print general registers

System.out.print("| R" + i + ": " + SummiX\_Utilities.shortToHexString(this.reg[i]) + "\t");

}

//output PC and current instruction

short instrAtPC = this.mem[SummiX\_Utilities.getBits(this.pc, 0, 7)][SummiX\_Utilities.getBits(this.pc, 7, 9)];

System.out.print("|\n| PC: 0x" + Integer.toHexString((int)this.pc) + "\t| Instr: " + SummiX\_Utilities.shortToHexString(instrAtPC)+ "\t|");

//output CCR

System.out.print(" CCR: ");

if (this.ccr.get(N)) {

System.out.print("1");

}

else {

System.out.print("0");

}

if (this.ccr.get(Z)) {

System.out.print("1");

}

else {

System.out.print("0");

}

if (this.ccr.get(P)) {

System.out.print("1");

}

else {

System.out.print("0");

}

System.out.print("\t|\t");

}

}

**Loader**

private int hexstringToInt(CharSequence input) {}

/\*\*

\* Takes a CharSequence that is a hex number and converts it to an integer.

\*

\* @param input CharSequence to be converted into an int of its hex value

\*/

private short getPage(int addr) {}

/\*\*

\* Gets the upper 7 bits (page) out of the address given by the input.

\*

\* @param addr the complete address given

\*/

private short getOffset(int addr) {}

/\*\*

\* Gets the lower 9 bits (offset) out of the address given by the input

\*

\* @param addr the complete address given

\*/

private void getHeader(SummiX\_Machine machine) {}

/\*\*

\* Gets the header information out of the input

\*

\* @param machine the SummiX\_Machine to potentially put the header info into

\*/

private void fillMemory(SummiX\_Machine machine) {}

/\*\*

\* Reads values from the input and stores them into the machine's memory

\*

\* @param machine the SummiX\_Machine to store data in

\*/

**Loader Source Code**

package summixSimulator;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.lang.NumberFormatException;

/\*\*

\* The SummiX loader puts the input information into data

\* structures that represent the memory and registers of the

\* machine.

\*

\* @author Mike/Mike/Dan/Jim

\*

\*/

public class Loader {

private BufferedReader br;

private int init;

private int length;

private int hexstringToInt(CharSequence input) {

int returnVal = 0; // needs initialized in the case an exception is caught

/\*\*

\* Takes a CharSequence that is a hex number and converts it to an integer.

\*

\* @param input CharSequence to be converted into an int of its hex value

\*/

//there is a lot of crappy looking casting going on here, is there a better way?

//should probably check for anything other than hex digits in these CharSequence

try {

returnVal = Integer.valueOf((String) input, 16).intValue();

} catch (NumberFormatException e) {

System.out.println("Expected: hex value");

System.exit(-1); //error

}

return returnVal;

}

private short getPage(int addr) {

/\*\*

\* Gets the upper 7 bits (page) out of the address given by the input.

\*

\* @param addr the complete address given

\*/

//upper 7 bits (shift right 9)

return (short) (addr >> 9);

}

private short getOffset(int addr) {

/\*\*

\* Gets the lower 9 bits (offset) out of the address given by the input

\*

\* @param addr the complete address given

\*/

//lower 9 bits (bitmask out upper 7)

return (short) (0x1FF & addr);

}

private void getHeader(SummiX\_Machine machine) throws IOException {

/\*\*

\* Gets the header information out of the input

\*

\* @param machine the SummiX\_Machine to potentially put the header info into

\*/

String input = this.br.readLine();

if (input.charAt(0) != 'H') {

System.out.println("Expected: H");

System.exit(-1); //error

}

this.init = hexstringToInt(input.subSequence(7, 11)); //programs memory begins here

this.length = hexstringToInt(input.subSequence(11, 15)); //length of the segment of memory

}

private void fillMemory(SummiX\_Machine machine) throws IOException {

/\*\*

\* Reads values from the input and stores them into the machine's memory

\*

\* @param machine the SummiX\_Machine to store data in

\*/

String input = this.br.readLine();

if (input.charAt(0) != 'T') {

System.out.println("Expected: T");

System.exit(-1); //error

}

while (input.charAt(0) == 'T') //Text Record

{

int addr = hexstringToInt(input.subSequence(1, 5));

if (addr < this.init) {

System.out.println("Address given (" + addr + " is less than start address (" + this.init + ")");

System.exit(-1);

}

if (addr > (this.init + this.length)) {

System.out.println("Address given (" + addr + " is greater than max address (" + (this.init + this.length) + ")");

System.exit(-1);

}

int data = hexstringToInt(input.subSequence(5, 9));

//store data into machine memory

machine.setMemory(getPage(addr), getOffset(addr), (short) data);

input = this.br.readLine();

}

//all that is left is the end record which sets the PC

if (input.charAt(0) != 'E') {

System.out.println("Expected: E");

System.exit(-1); //error

}

machine.setPC((short)hexstringToInt(input.subSequence(1,5)));

}

public Loader(String filename, SummiX\_Machine machine) throws IOException {

this.br = new BufferedReader(new FileReader(filename));

getHeader(machine);

fillMemory(machine);

this.br.close();

}

}

**Interpreter**

public static boolean getInstruction (SummiX\_Machine machine, short data) {}

/\*\*

\* Using a case select, we take the first 4 bits of the instruction code and

\* decide which specific opcode is present and pass the enum value to the

\* executer.

\*/

**Interpreter Source Code**

/\*\*

\*

\*/

package summixSimulator;

import summixSimulator.SummiX\_Utilities.InstructionCode;

/\*\*

\* The SummiX interpreter simulates the operation of the

\* machine as it executes each instruction.

\*

\* @author Mike/Mike/Dan/Jim

\*

\*/

public class Interpreter {

public static boolean getInstruction(SummiX\_Machine machine, short data)

{

int instruction;

boolean halt = false;

instruction = SummiX\_Utilities.getBits(data, 0, 4);

InstructionCode op=null;

int bit;

switch(instruction) {

case 1: //ADD

bit = SummiX\_Utilities.getBits(data, 10, 1); //get ADD bit

if (bit == 0)

op = InstructionCode.ADD;

else

op = InstructionCode.ADD2;

break;

case 5: //AND

bit = SummiX\_Utilities.getBits(data, 10, 1); //get AND bit

if (bit == 0)

op = InstructionCode.AND;

else

op = InstructionCode.AND2;

break;

case 0: //BRx

op = InstructionCode.BRX;

break;

case 8: //DBUG

op = InstructionCode.DBUG;

break;

case 4: //JSR

op = InstructionCode.JSR;

break;

case 12: //JSRR

op = InstructionCode.JSRR;

break;

case 2: //LD

op = InstructionCode.LD;

break;

case 10: //LDI

op = InstructionCode.LDI;

break;

case 6: //LDR

op = InstructionCode.LDR;

break;

case 14: //LEA

op = InstructionCode.LEA;

break;

case 9: //NOT

op = InstructionCode.NOT;

break;

case 13: //RET

op = InstructionCode.RET;

break;

case 3: //ST

op = InstructionCode.ST;

break;

case 11: //STI

op = InstructionCode.STI;

break;

case 7: //STR

op = InstructionCode.STR;

break;

case 15: //TRAP

//use case selects for multiple traps

short trap = SummiX\_Utilities.getBits(data, 8, 8); // get trapvect8

switch(trap){

case 0x21: //OUT

op = InstructionCode.OUT;

break;

case 0x22: //PUTS

op = InstructionCode.PUTS;

break;

case 0x23: //IN

op = InstructionCode.IN;

break;

case 0x25: //HALT

op = InstructionCode.HALT;

halt = true;

break;

case 0x31: //OUTN

op = InstructionCode.OUTN;

break;

case 0x33: //INN

op = InstructionCode.INN;

break;

case 0x43: //RND

op = InstructionCode.RND;

break;

default:

System.out.println("Error: Invalid TRAP code");

op = InstructionCode.HALT;

halt = true;

break;

}

break;

default:

System.out.println("Error: Invalid OPCODE");

op = InstructionCode.ERR;

halt = true;

break;

}

new Executor(machine, data, op); //PC gets incremented in here

if (instruction == 15) { //traps are special

//all trap instructions set r7 to the address following the trap instruction

machine.setSubroutineReturn(machine.getPC());

}

return halt;

}

}

**Executor**

public Executor() {}

/\*\*

\* Executes instruction given by the opcode

\*/

**Executor Source Code**

package summixSimulator;

import java.util.Scanner;

import summixSimulator.SummiX\_Utilities.InstructionCode;

public class Executor {

public Executor(SummiX\_Machine machine, short data, InstructionCode op) {

short sr1, sr2, sr, dr, pg, imm5, baser, pgoffset6, index6, pgoffset9, addr, valueAtAddr, valueAtBaseR;

Scanner in = new Scanner(System.in);

short oldpc = machine.getPC();

machine.incrementPC();

switch (op) {

case ADD:

sr1 = SummiX\_Utilities.getBits(data, 7, 3);

sr2 = SummiX\_Utilities.getBits(data, 13, 3);

dr = SummiX\_Utilities.getBits(data, 4, 3);

machine.setRegister(dr, (short) (machine.loadRegister(sr1)

+ machine.loadRegister(sr2)));

//System.out.println("ADD1\tdr:" + dr + " sr1: " + sr1 + "(" + Integer.toHexString(machine.loadRegister(sr1)) + ")+ sr2:" + sr2 + "("+Integer.toHexString(machine.loadRegister(sr2))+") value:" + Integer.toHexString(machine.loadRegister(dr)));

break;

case ADD2:

sr1 = SummiX\_Utilities.getBits(data, 7, 3);

if (SummiX\_Utilities.getBits(data, 11, 1)==1) {

//negative value

imm5 = (short) (SummiX\_Utilities.getBits(data, 11, 5) << 27 >> 27);

} else {

imm5 = (short) (SummiX\_Utilities.getBits(data, 11, 5));

}

dr = SummiX\_Utilities.getBits(data, 4, 3);

machine.setRegister(dr, (short) (machine.loadRegister(sr1)

+ imm5));

//System.out.println("ADD2\t dr:" + dr + " sr1: " + sr1 + "(" + Integer.toHexString(machine.loadRegister(sr1)) + ")+ imm5:" + imm5 + " value:" + Integer.toHexString(machine.loadRegister(dr)));

break;

case AND:

sr1 = SummiX\_Utilities.getBits(data, 7, 3);

sr2 = SummiX\_Utilities.getBits(data, 12, 3);

dr = SummiX\_Utilities.getBits(data, 4, 3);

machine.setRegister(dr, (short) (machine.loadRegister(sr1)

& machine.loadRegister(sr2)));

//System.out.println("AND:\t" + Integer.toHexString(machine.loadRegister(dr)));

break;

case AND2:

sr1 = SummiX\_Utilities.getBits(data, 7, 3);

imm5 = SummiX\_Utilities.getBits(data, 11, 5);

dr = SummiX\_Utilities.getBits(data, 4, 3);

machine.setRegister(dr, (short) (machine.loadRegister(sr1)

& imm5));

//System.out.println("AND2:\t" + Integer.toHexString(machine.loadRegister(dr)));

break;

case BRX:

pgoffset9 = SummiX\_Utilities.getBits(data, 7, 9);

//System.out.println("BRX\tZ: " + machine.getZ());

if ((SummiX\_Utilities.getBits(data, 4, 1) == 1) && machine.getN() ||

(SummiX\_Utilities.getBits(data, 5, 1) == 1) && machine.getZ() ||

(SummiX\_Utilities.getBits(data, 6, 1) == 1) && machine.getP()) {

//if any of the above cases are true set the pc

machine.setPC((short) (SummiX\_Utilities.getAbsoluteBits(oldpc, 0, 7) + pgoffset9));

}

break;

case DBUG: //The DBUG instruction displays the contents of PC, general registers, and ccr to the console

System.out.println("SummiX system debug: \n");

machine.outputMachineState();

case JSR:

if (SummiX\_Utilities.getBits(data, 4, 1) == 1) { // link bit is set

machine.setSubroutineReturn(machine.getPC()); //so set r7 to current pc for return

}

pgoffset9 = SummiX\_Utilities.getBits(data, 7, 9);

//jump to first 7 bits of PC plus 9 given by offset

machine.setPC((short) (SummiX\_Utilities.getAbsoluteBits(machine.getPC(), 0, 7) + pgoffset9));

break;

case JSRR: //someone read the directions for this and check to make sure i didn't mess it up

if (SummiX\_Utilities.getBits(data, 4, 1) == 1) { // link bit is set

machine.setSubroutineReturn(machine.getPC()); //so set r7 to current pc for return

}

pgoffset6 = (short) (SummiX\_Utilities.getBits(data, 10, 6));

machine.setPC((short) (pgoffset6 + machine.loadRegister(SummiX\_Utilities.getBits(data, 7, 3))));

break;

case LD:

dr = SummiX\_Utilities.getBits(data, 4, 3);

pgoffset9 = SummiX\_Utilities.getBits(data, 7, 9);

//working through the example1.txt (given on 560 web site) we apparently

//did this backwards, it's supposed to load the value at the address given

//originally we did: machine.setRegister(dr, (short) (SummiX\_Utilities.getAbsoluteBits(machine.getPC(), 0, 7) + pgoffset9));

addr = (short) (SummiX\_Utilities.getAbsoluteBits(machine.getPC(), 0, 7) + pgoffset9);

valueAtAddr = machine.loadMemory(SummiX\_Utilities.getBits(addr, 0, 7), SummiX\_Utilities.getBits(addr, 7, 9));

machine.setRegister(dr, valueAtAddr);

//because of this i think perhaps we need to look at LDR and ST/STR... stupid specs

//System.out.println("LD\tdr: " + dr + " set to: " + Integer.toHexString(machine.loadRegister(dr)));

break;

case LDI:

dr = SummiX\_Utilities.getBits(data, 4, 3);

pgoffset9 = SummiX\_Utilities.getBits(data, 7, 9);

machine.setRegister(dr, (short) (SummiX\_Utilities.getAbsoluteBits(machine.loadRegister(dr), 0, 7) + pgoffset9));

break;

case LDR:

//zero extend index6 and add it to value in BaseR

index6 = SummiX\_Utilities.getBits(data, 10, 6); //zero extend index6

baser = SummiX\_Utilities.getBits(data, 7, 3);

valueAtBaseR = machine.loadRegister(baser);

addr = (short) (index6 + valueAtBaseR);

valueAtAddr = machine.loadMemory(SummiX\_Utilities.getBits(addr, 0, 7), SummiX\_Utilities.getBits(addr, 7, 9));

dr = SummiX\_Utilities.getBits(data, 4, 3);

machine.setRegister(dr, valueAtAddr);

//System.out.println("LDR\t" + dr + "(" + Integer.toHexString(machine.loadRegister(dr)) + ") baser:"+baser + "(" + Integer.toHexString(machine.loadRegister(baser)) + ")");

break;

case LEA:

//15:9 pc + 8:0 pgoffset9

dr = SummiX\_Utilities.getBits(data, 4, 3);

pgoffset9 = SummiX\_Utilities.getBits(data, 7, 9);

machine.setRegister(dr, (short) (SummiX\_Utilities.getAbsoluteBits(machine.getPC(),0,7) + pgoffset9));

//System.out.println("LEA\t" + "dr:"+dr + " ("+Integer.toHexString(machine.loadRegister(dr))+")");

break;

case NOT:

dr = SummiX\_Utilities.getBits(data, 4, 3); // Get destination register

sr = SummiX\_Utilities.getBits(data, 7, 3); // Get source register

sr = (short) ~sr; //Bitwise inversion of the value in source register

machine.setRegister(dr, sr); //Store data from source register into destination register

break;

case RET:

machine.setPC(machine.loadRegister(7)); // copies the contents of R7 to PC

break;

case ST:

sr = SummiX\_Utilities.getBits(data, 4, 3);

pg = SummiX\_Utilities.getBits(machine.getPC(), 0, 7);

pgoffset9 = SummiX\_Utilities.getBits(data, 7, 9);

machine.setMemory(pg, pgoffset9, machine.loadRegister(sr));

break;

case STI:

sr = SummiX\_Utilities.getBits(data, 4, 3);

pg = SummiX\_Utilities.getBits(sr, 0, 7);

pgoffset9 = SummiX\_Utilities.getBits(data, 7, 9);

machine.setMemory(pg, pgoffset9, machine.loadRegister(sr));

break;

case STR:

//zero extend index6 and add it to value in BaseR

index6 = SummiX\_Utilities.getBits(data, 10, 6);

baser = SummiX\_Utilities.getBits(data, 7, 3);

sr = SummiX\_Utilities.getBits(data, 4, 3);

machine.setMemory(machine.loadRegister(baser), index6, machine.loadRegister(sr));

break;

//Trap instructions below

case OUT:

System.out.print((char)SummiX\_Utilities.getBits(machine.loadRegister(0), 8, 8));

break;

case PUTS:

char tempChar;

short memorySpaceToLoadFrom = machine.loadRegister(0);

short page = SummiX\_Utilities.getBits(memorySpaceToLoadFrom, 0, 7);

short offset = SummiX\_Utilities.getBits(memorySpaceToLoadFrom, 8, 9);

tempChar = (char) machine.loadMemory(page, offset);

while (tempChar != '\0')

{

System.out.print(tempChar);

if (offset==511) {

page = (short) (((short) (page >>> 9) + 1) << 9);

memorySpaceToLoadFrom = page;

} else {

memorySpaceToLoadFrom++;

}

page = SummiX\_Utilities.getBits(memorySpaceToLoadFrom, 0, 7);

offset = SummiX\_Utilities.getBits(memorySpaceToLoadFrom, 8, 9);

tempChar = (char) machine.loadMemory(page, offset);

}

break;

case IN:

System.out.print("Please input character to be stored in R0: ");

char ascii = in.next().charAt(0);

System.out.println(ascii);

//machine.setRegister(0,((short) ((ascii << 8) >>> 8)));

machine.setRegister(0,(short)ascii); // may have to use above statement to clear upper 8 bits

break;

case HALT:

System.out.println("\nSystem exited normally.");

break;

case OUTN: //write value of r0 to console as a decimal

System.out.print(machine.loadRegister(0));

break;

case INN:

short input;

System.out.print("Please enter a number between -32768 and 32767 with no commas: ");

input = (short) Integer.getInteger(in.next()).intValue();

machine.setRegister(0, input);

break;

case RND:

machine.setRegister(0, (short)Math.random());

break;

case ERR: //was an error op code

System.out.println("System error: ");

break;

}

}

}

**SummiX\_Utilities**

public static short getBits(short data, int p, int n) {}

/\*\*

\* returns n bits of data starting at p

\*/

public static short getAbsoluteBits(short data, int p, int n) {}

/\*\*

\* Returns n bits of data starting at p and sets all other bits to “0”

\*/

public enum InstructionCode {}

/\*\*

\* Creates enum values of the following

\* ADD, ADD2, AND, AND2, BRX, DBUG, JSR, JSRR, LD, LDI, LDR, LEA

\* NOT, RET, ST, STI, STR, OUT, PUTS, IN, HALT, OUTN, INN, RND, ERR

\*/

**SummiX\_Utilities Source Code**

package summixSimulator;

public class SummiX\_Utilities {

public static short getBits(short data, int p, int n) {

return (short) ((data >>> (16-p-n)) & ((1 << n)-1));

}

public static short getAbsoluteBits(short data, int p, int n) {

short s = (short) ((data >>> (16-p-n)) & ((1 << n)-1));

return (short) (s << (16-p-n));

}

public static String shortToHexString(short data) {

String returnVal = Integer.toHexString((int) data);

if (returnVal.length() > 4) {

returnVal = returnVal.substring(returnVal.length() - 4, returnVal.length());

}

return "0x" + returnVal.toUpperCase();

}

public enum Simulator\_State {

QUIET,

TRACE,

STEP,

ERROR

}

public enum InstructionCode{

ADD,

ADD2,

AND,

AND2,

BRX,

DBUG,

JSR,

JSRR,

LD,

LDI,

LDR,

LEA,

NOT,

RET,

ST,

STI,

STR,

OUT, //TRAP ENUMS THIS LINE AND LOWER

PUTS,

IN,

HALT,

OUTN,

INN,

RND,

ERR

}

}

**Thoughts on our Design**

We believe that our design follows the specific requirements of the lab and does so in the most straight forward, obvious manner. We choose a programming language who’s primary benefits were in regard to adaptability, ease of implementation, as well as sharing of operations. We decided to keep the code organized in such a manner where one class passes to the next in order to keep a well defined structure and pattern of operation. We feel that this design makes it not only easy for a developer to understand, but would also be obvious enough for the user to understand.