**Programmers Guide**

Team IDK

CSE 560, Lab 1

July 6th, 2009

Primary author: Michael Irwin

Secondary authors: Dan Stottlemire

Jim Powers

Michael Pinneger

**Table of Contents**

1. Title Page
2. Table of Contents
3. Outline of the problem
4. Our approach
5. Data Object Library
6. Simulator
7. SummiX\_Machine
8. Loader
9. Interpreter
10. Executor
11. SummiX\_Utilities
12. Thoughts on design and possible improvements
13. – 17. Meeting Notes (will be relocated for final documentation)

**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**

insert graphic created by Michael Pinnegar

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.

**Data Object Library**

**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); | |  | | --- | | Methods |   hexstringToInt(CharSequence);  getPage(int);  getOffset(int);  getHeader(SummiX\_Machine);  fillMemory(SummiX\_Machine);  Loader(String, SummiX\_Machine(); |

**Interpreter Executer 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

\*/

**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 \*/

**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

\*/

**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.

\*/

**Executer**

public Executer() {}

/\*\*

\* Executes instruction given by the opcode

\*/

**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

\*/

**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.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Meeting Called By: | Team IDK | | Type of Meeting: | Project overview | | Date: | June 23rd, 2009 | | Time: | 7:30 PM | | Location: | Caldwell Labs | | |  |  | | --- | --- | | Attendees: | Dan Stottlemire | |  | Michael Pinnegar | |  | Michael Irwin\* | |  | James Power |   \*secretary |

Record

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | Java DOC / Documentation | Presenter Name: Team IDK | |
| |  | | --- | | Discussion: Programmers guide vs. User guide, specifics needed for documentation. “Abstract Kernel” idea from 321 to form the documentation for the peer group review. Composed a meeting schedule for before and after class on Monday and Wednesday as well as 12-1pm on Thursday. Jim is going to look into setting up a section for code.google.com in order to be able to work from home on the same source code. | |  |  |  |  | | --- | --- | | Type: Documentation |  | |  |  |  |  | | --- | | Other Information: | | Monday 7:30 - ?, Wednesday 12pm – 1pm, 7:30 - ?, Thursday 12pm – 1pm | | Conclusions: Use multiple classes in Java to accomplish our goal. We will be keeping the simulator and the main SummiX machine separate. | |  |  |  |  |  | | --- | --- | --- | | Action Items: | Person Responsible: | Deadline: | | Look into using Google Code for Project | James Power | June 24th, 2009 | |  |  |  | |  |  |  | |  |  |  | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Meeting Called By: | Team IDK | | Type of Meeting: | Project overview | | Date: | June 29rd, 2009 | | Time: | 7:00 PM | | Location: | Caldwell Labs | | |  |  | | --- | --- | | Attendees: | Dan Stottlemire | |  | Michael Pinnegar | |  | Michael Irwin\* | |  | James Power |   \*secretary |

Record

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | High Level Design / Understanding Concepts | Presenter Name: Team IDK | |
| |  | | --- | | Discussion: Discussed the exact way that all of our methods would function together as well as drew out a diagram to further the understanding of all members of the group. The group decided that it would separate the function of the interpreter and an “executer.” This separation would further the flexibility the group has for further additions and lab assignments. The group also decided that a generic “getBits” function would be useful in order to gather the specific bits needed for each operation. This function would be written in a utility class that is extended by the other classes in order to spread its availability. | |  |  |  |  | | --- | --- | | Type: Planning |  | |  |  |  |  | | --- | | Other Information: | | Setup a schedule to be doing coding and begin testing. We decided that Wednesday, July 8th that we would be done programming and begin testing. Doing so would give us roughly a week to test our program and work out any bugs. We are also going to try and be completely done with the majority of the programmers guide by Sunday, July 5th. | | The group decided to use meetings on Tuesday and Thursday to be the primary code laying for the group and setting up the exact structure while Monday and Wednesday were going to be more planning and discussion. This pattern may however change as we get closer to the deadline. | |  |  |  |  |  | | --- | --- | --- | | Action Items: | Person Responsible: | Deadline: | | Documentation Outline | Michael Irwin | June 30th, 2009 | | Group Understanding of Problem | All Members | June 30th, 2009 | | Strategy for completing specific methods | All Members | June 30th, 2009 | |  |  |  | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Meeting Called By: | Team IDK | | Type of Meeting: | Project overview | | Date: | June 30rd, 2009 | | Time: | 12:30 PM | | Location: | Caldwell Labs | | |  |  | | --- | --- | | Attendees: | Dan Stottlemire | |  | Michael Pinnegar | |  | Michael Irwin\* | |  | James Power |   \*secretary |

Record

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | Interpreter / Executer | Presenter Name: Team IDK | |
| |  | | --- | | Discussion: Primary method for passing instruction op codes. Decided to making enumerated values in a main utility class and pass that a long from the interpreter to the executer. | |  |  |  |  | | --- | --- | | Type: Enum values / interpreter |  | |  |  |  |  | | --- | | Other Information: | | Created working groups for a couple of lab meetings, Dan and Mike Irwin as well as James and Mike Pinnegar will be working together on different sections of the labs | | Dan and Michael Irwin worked on the interpreter while James and Michael Pinnegar were working on the executer | |  |  |  |  |  | | --- | --- | --- | | Action Items: | Person Responsible: | Deadline: | | Interpreter | Jim / Dan | July 1st, 2009 | | Documentation | Michael Irwin | July 5th, 2009 | | Logic behind simulator functionality | Michael Pinnegar | July 1st, 2009 | |  |  |  | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Meeting Called By: | Team IDK | | Type of Meeting: | Project overview | | Date: | July 1st, 2009 | | Time: | 7:00 PM | | Location: | Caldwell Labs | | |  |  | | --- | --- | | Attendees: | Dan Stottlemire | |  | Michael Pinnegar | |  | Michael Irwin\* | |  | James Power |   \*secretary |

Record

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | Interpreter / Executer | Presenter Name: Team IDK | |
| |  | | --- | | Discussion: Continued working on code for executer and simulator. Decided how the group would prompt the user in regard to modes of operation (decided upon a combination of prompting and user guided codes) | |  |  |  |  | | --- | --- | | Type: Coding |  | |  |  |  |  | | --- | | Other Information: | | Dan and Mike worked on code for the executer while Jim and Mike P. were working on the simulator and the modes of operation. | | Looked over the beginning stages of the Programmers guide and discussed possible changes. |  |  |  |  | | --- | --- | --- | | Action Items: | Person Responsible: | Deadline: | | Simulator | Jim / Mike P. | July 6th, 2009 | | Programmers Guide | Michael Irwin | July 5th, 2009 | | Executer | Dan | July 5th. 2009 | |  |  |  | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | Meeting Called By: | Team IDK | | Type of Meeting: | Project overview | | Date: | July 2nd, 2009 | | Time: | 7:00 PM | | Location: | Caldwell Labs | | |  |  | | --- | --- | | Attendees: | Dan Stottlemire | |  | Michael Pinnegar | |  | Michael Irwin\* | |  | James Power |   \*secretary |

Record

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | Interpreter / Executer | Presenter Name: Team IDK | |
| |  | | --- | | Discussion: No real discussion about any problems, continued filling in gaps in the code for the remaining classes. | |  |  |  |  | | --- | --- | | Type: Coding |  | |  |  |  |  | | --- | | Other Information: | | Dan and Mike worked on code for the executer while Jim and Mike P. were working on the simulator and the modes of operation. | | Looked over the beginning stages of the Programmers guide and discussed possible changes. |  |  |  |  | | --- | --- | --- | | Action Items: | Person Responsible: | Deadline: | | Simulator | Jim / Mike P. | July 6th, 2009 | | Programmers Guide | Michael Irwin | July 5th, 2009 | | Executer | Dan | July 5th. 2009 | |  |  |  | |