|  |
| --- |
|  |
| Programmer’s Guide |
| Lab 1 Group BEERZ |
|  |
| **Zach Smith** |
| **Brad Kline, Elliot Schumacher, Evan Todd, and Ryan Powers** |
| **1/24/2011** |

|  |
| --- |
|  |

# **Table of Contents**

# Introduction

# Introduction

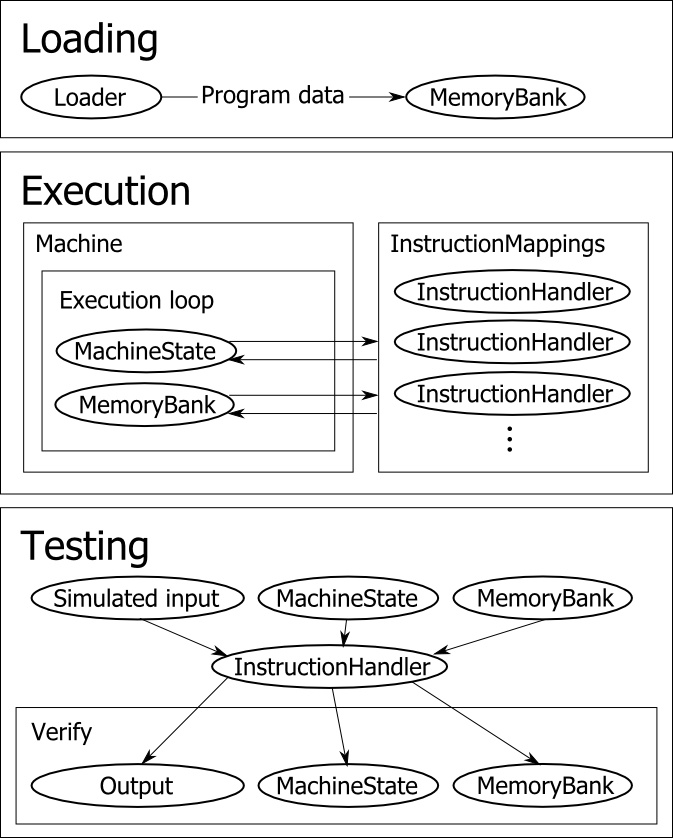
This document contains a summary of the implementation of the “Wi-11 Machine” simulator. As such, it will cover all of the components involved in the execution of the simulator. The system is designed to simulate a machine with the following basic characteristics:

* Memory
  + Word-addressable, 16-bit memory
* Registers
  + Eight general purpose registers of 16-bit length
  + 16-bit program counter (PC)
  + Three 1-bit condition code registers (CCRs)
* Arithmetic
  + Fixed-point arithmetic only
  + Negative numbers represented in two’s complement
  + Positive and negative overflow allowed
* Instruction Set Format
  + 16-bit length
  + 4 high order bits denote the operation code (opcode)
  + Multiple addressing modes allowed

Throughout the document, names of classes will be in **Bold,** with the first letter capitalized. These classes are public unless stated otherwise.

# System Overview

When the simulator program runs, the entry point is in the class entitled **Main**, which is implemented in the file **Main.java**. **Main** is responsible for accepting and handling command line arguments, loading the memory of the simulator, and running the client’s program. **Main** relies on the **Loader** and **Machine** classes respectively to handle the final two tasks, which are implemented in the files **Loader.java** and **Machine.java.** The instructions of the machine are all implemented individually in their own class.

Figure 1: Diagram of interactions between components during different program phases

### Directory Structure

The system’s files are in a parent directory named “cse-560-beerz”, which has several subfolders for different packages in the implementation. The following file is simply under the parent directory:

* Main.java
  + Contains the implementation of the **Main** class, which drives the program through its execution by loading memory first and then passing off the execution of the stored instructions.

These three files are grouped into the “/program” directory, and also into a “program” package:

* Loader.java
  + Contains the implementation of the **Loader** class, which decodes the object file passed in by the user and loads the machine’s memory accordingly.
* Machine.java
  + Implements the **Machine** class, which, through the use of the **MachineState** and **MemoryBank** classes, holds the representation of the machine and its memory. Also passes off execution of the instructions to the **InstructionMappings** class.
* ExecutionMode.java
  + This file creates an enumerated type for the execution mode of the program, namely: QUIET, TRACE, AND STEP.

The following files are in the “/state” directory, and are also together in the “state” package:

* MachineState.java
  + Implements the **MachineState** class, which is used by **Machine** to represent the state of the simulated machine.
* MemoryBank.java
  + Contains the class **MemoryBank,** which represents the memory of the simulated machine.

These files are in the “/instructions” directory, and are all members of the “instructions” package, each implementing the instruction for which they are named:

* AddHandler.java
* AndHandler.java
* BranchHandler.java
* DebugHandler.java
* JumpSubroutineImmediateHandler.java
* JumpSubroutineRegisterHandler.java
* LoadEffectiveAddressHandler.java
* LoadHandler.java
* LoadImmediateHandler.java
* LoadRegisterHandler.java
* NotHandler.java
* ReturnHandler.java
* StoreHandler.java
* StoreImmediateHandler.java
* StoreHandler.java
* TrapHandler.java

These two files are also in the “/instructions” directory, but are more unique than the previous files listed:

* InstructionMappings.java
  + Acts as a lookup table for all of the instruction handlers listed above; creates a specific **InstructionHandler** for the instruction.
* InstructionHandler.java
  + Provides convenience methods for testing instructions, as well as directing the program to the specific instruction handler listed above.

The files that follow are all in the “/testing” directory and “testing” package, and are all JUnit test suites that test a specific area of functionality, as indicated by their names:

* AddTest.java
* AndTest.java
* BranchTest.java
* JumpSubroutineImmediateTest.java
* JumpSubroutineRegisterTest.java
* LoadEffectiveAddressTest.java
* LoaderTest.java
* LoadImmediateTest.java
* LoadRegisterTest.java
* LoadTest.java
* MachineTest.java
* NotTest.java
* ReturnTest.java
* StoreImmediateTest.java
* StoreRegisterTest.java
* StoreTest.java
* TestBase.java
* TrapTest.java

Finally, the only file in the subdirectory “/util” is:

* ByteOperations.java
  + Contains methods for manipulating bits and bytes

## Design Conventions

Across the files in our implementation, several design conventions were followed. These include the capitalization of class names, like **Main** or **TrapHandler**, lower-case names for packages and variables, and mixed-case names for method titles like “getOpCode”. The code on a whole was written to separate behavior from state as much as possible, as can be seen with the separate instruction handlers, none of which rely directly on **Machine** but on **MachineState and MemoryBank** instead. This design also allows for easy testing of independent components.

## Module Inter-Relationships

Due to the design of our system, there are separate components for the behavior of an element and the state of the element. For example, consider the classes **Machine** and **MachineState**; the abstract model of **Machine** encapsulates that of **MachineState**, since one implements the behavior of the simulated machine and the other provides the state. Thus, the abstract client view of the **Machine** class implicitly includes that of **MachineState.** The same is true for **MemoryBank**, since its implementation provides only the behavior of memory.

# Data Structures

The largest and most important data structures are easily the ones representing the machine being simulated. The following is a listing of the data structure, the file it appears in, the element of the machine that it represents, its implementation, and its invariant:

* Object: “data” in MemoryBank.java
  + Represents the memory banks of the machine
  + Implemented as a HashMap<Integer, Short>();
  + Invariant: the address (the Integer of the pairing) must be between 0 and 65,536
* Object: “ccrNegative” in MachineState.java
  + Represents the negative CCR bit
  + Implemented as a boolean
  + Invariant: true if and only if the last value written to a register is negative
* Object: “ccrPositive” in MachineState.java
  + Represents the positive CCR bit
  + Implemented as a boolean
  + Invariant: true if and only if the last value written to a register is positive
* Object: “ccrZero” in MachineState.java
  + Represents the zero CCR bit
  + Implemented as a Boolean
  + Invariant: true if and only if the last value written to a register is zero
* Object: “registers” in MachineState.java
  + Represents the eight general purpose registers
  + Implemented as short[ ] of length 8
* Object: “programCounter” in MachineState.java
  + Represents the PC
  + Implemented as an int
  + Invariant: must be between 0 and 65,536

# Component Descriptions

This section provides a detailed description of both the client- and implementation-side view of every component used in the program, with the exception of the testing package of components.

## Client-Side

* **Main**
  + Description: The **Main** component is the largest section of the simulation, as well as being the entry point of the program. It accepts the command line arguments from the user, like the input file and the different execution modes (quiet, trace, or step), and runs the simulation accordingly.
* **Machine**
  + Description: The **Machine** component holds the representation of the internal state of the machine in the simulation. This class is capable of executing instructions stored in a memory bank. **NOTE:** the abstract view of **Machine** implicitly includes that of **MemoryBank** and **MachineState**, as explained in the Module-Interdependence section.
  + Mathematical Model: machineRep = memory + registers + PC + CCRs

memory = MEM[addr]

* + Constraint: 0 <= addr <= 65,536

0 <= PC <= 65,536

For each register: 0 <= register <= 65,536

CCRs = {0,1} where only one CCR is 1 at a time

* + Initial State: for all addr in MEM[addr] = 0

PC = 0

Registers = 0

CCRZero = 1

* **Loader**
  + Description:

## Implementer-Side

* **Main**
  + Description: Provides the entry point for the program, and accepts command line arguments. Gets the name of the input file, passes it to the loader, and then begins execution of the stored instructions.
  + State: none
  + Algorithm:

If ( |args| < 1 OR args[0] = “--help”) then

Display usage information

For int x = 1, x < |args| do

If ( args[x] = “-o” then

x++ and look at next args[x]

Set up an output file for args[x]

If (args[x] = “-r” then

x++ and look at next args[x]

Set the execution mode

filename = args[0]

load all of filename into a string

pass the file’s data to **Loader**

**Machine.run**(starting address, execution mode)

//User’s code runs

close output streams

* **Methods in Main**
* Method Name: main
  + Description: main in class **Main**.
  + Parameters: args[ ]
  + Requires: true
  + Alters: machineRep
  + Ensures: machineRep = #machineRep [altered by user’s input program]
  + Returns: void
  + Throws: IOException
* Method Name: printUsageInformation
  + Description: Prints usage information to the console. Shows:

“Useage: java Main inputfile [options]

-o outputfile Redirect output to specified file.

-r quiet Run the program in quiet mode.

-r trace Run the program in trace mode.

-r step Run the program in step mode. “

* + Parameters: none
  + Requires: true
  + Alters: System.out
  + Ensures: System.out = #System.out + [usage info]
  + Returns: void
* Method: readAllText
  + Description: Reads all text in the file existing at the given path location into a string.
  + Parameters: filename Path to the desired file
  + Requires: true
  + Alters: true
  + Ensures:
  + Returns: A string containing all the data existing in the desired file.
  + Throws: IOException
* **Machine**
  + Description: This class represents a virtual machine capable of executing instructions stored in a **MemoryBank**.
  + State: private MemoryBank memory, private MachineState state, private PrintStream output, private static final int PG\_LOW\_BIT, private static final int PG\_HI\_BIT
  + Correspondence: machineRep (memory) = **MemoryBank**

machineRep(registers, CCRs, PC) = **MachineState**

* **Machine** methods
* Method: run
  + Description: Begins execution at the given address in memory.
  + Parameters: int startAddress, ExecutionMode mode
  + Requires: 0 <= startAddress <= 65,536
  + Alters: this
  + Ensures: this = #this [altered by user’s program]
  + Returns: void
  + Throws: Exception
* Method: execute
  + Description: Executes the given instruction.
  + Parameters: in instruction
  + Requires: instruction is in **InstructionMappings**
  + Alters: this
  + Ensures: this = #this [modified by instruction]
  + Returns: void
  + Throws: Exception
* Method: getState
  + Description: Gets a new MachineState which describes the current state of this machine.
  + Parameters: none
  + Requires: true
  + Alters: true
  + Ensures: this = #this
  + Returns: a new MachineState with the value of this
* **MemoryBank**
  + Description: Represents the state of memory in a **Machine**. Uses a Map to represent an array of memory.
  + State: private static final in PAGE\_SHIFT, private Map<Integer, Short> data
  + Correspondence: machineRep(memory) = data; where data is a HashMap<Integer, Short>
  + Convention: 0 <= data(address) <= 65,536; 0 <= data(value) <= 65,536
* **MemoryBank** Methods
* Method: write
  + Description: Sets the memory cell at the given address to the given value.
  + Parameters: int address, short value
  + Requires: 0 <= address, value <= 65,536
  + Alters: this.data
  + Ensures: this.data = #this.data + (address, value)
  + Returns: void
* Method: read
  + Description: Gets the value of the memory cell at the given address.
  + Parameters: int address
  + Requires: 0 <= address <= 65,536
  + Alters: true
  + Ensures: this.data = #this.data
  + Returns: this.data[address]
* Method: displayPage
  + Description: Prints the state of the given memory page (bit-shifted all the way to the right) to the given output stream.
  + Parameters: PrintStream output, int page
  + Requires: output is open
  + Alters: output
  + Ensures: output = #output + [page of memory from this.data]
  + Returns: void
* **MachineState**
  + Description: This class represents the state of a virtual machine, not including the memory, which is represented by a MemoryBank.
  + State: public static final int NUM\_REGISTERS, public boolean ccrNegative, public boolean ccrPositive, public boolean ccrZero, public boolean executing, public short[ ] registers, public int programCounter.
  + Convention: [only one CCR may be true at a time]
  + Correspondence: {ccrNegative, ccrPositive, ccrZero} = CCRs

executing = [false iff a HALT is encountered]

short[ ] = registers

programCounter = PC

* **MachineState Methods**
* Method: clone
  + Description: Gets a copy of this MachineState.
  + Parameters: none
  + Requires: true
  + Alters: true
  + Ensures: this = # this
  + Returns: a new MachineState with the value of this
* Method: updateCcr
  + Description: Updates the CCR register in accordance with the given signed 16-bit value.
  + Parameters: short value
  + Requires: true
  + Alters: this.ccrNegative, this.ccrPositive, this.ccrZero
  + Ensures: [correct CCR is set true based on value last written to register; only one CCR is true]
  + Returns: void
* Method: display
  + Description: Outputs this MachineState to the given IO stream accorind go the DEBUG instruction specifications.
  + Parameters: PrintStream output
  + Requires: output be open
  + Alters: output
  + Ensures: output = #output + DBUG(this)
  + Returns: void

# Code