Programmer’s Guide

Lab 2 Group BEERZ

**Brad Kline**

**Elliot Schumacher, Evan Todd, Ryan Powers, and Zach Smith**

**2/23/2011**

Contents

[Introduction 3](#_Toc286160193)

[Assembler Overview 3](#_Toc286160194)

[Directory Structure 3](#_Toc286160195)

[Design Conventions 4](#_Toc286160196)

[Module Inter-Relationships 4](#_Toc286160197)

[Data Structures 4](#_Toc286160198)

[Component Descriptions 5](#_Toc286160199)

[Client-Side 5](#_Toc286160200)

[Implementer-Side 6](#_Toc286160201)

[Main 6](#_Toc286160202)

[Assembler 7](#_Toc286160203)

[ByteOperations 9](#_Toc286160204)

[Instruction 10](#_Toc286160205)

[InstructionDefinition 10](#_Toc286160206)

[LiteralTable 10](#_Toc286160207)

[Operand 10](#_Toc286160208)

[Operand 11](#_Toc286160209)

[OperandDefinition 12](#_Toc286160210)

[OperandType 13](#_Toc286160211)

[Program 13](#_Toc286160212)

[Symbol 14](#_Toc286160213)

[SymbolTable 15](#_Toc286160214)

# Introduction

This document contains a summary of the implementation of the “Wi-11 Machine” assembler. This guide will cover all of the components that make up and are used in the execution of the assembler.

# Assembler Overview

When the assembler runs, the entry point is in the class entitled **Main**, which is implemented in the **Assembler** package in the **Main.java** file. **Main** is responsible for accepting and handling command line arguments, creating the object file and generating a program listing. **Main** uses the **Assembler** and **Program** classes in order to complete the tasks of creating the object file and listing.

## Directory Structure

The assembler’s files are in a parent directory named “cse-560-beerz,” that contains one subfolder named Assembler. The following files are in the Assembler directory:

* **Main.java**
  + Contains the implementation of the **Main** class, which controls the creation of the object file and program listing, as well as handling all of the I/O with the end user in the form of command line arguments and error and usage out to the console.
* **Assembler.java**
  + Contains the implementation of the **Assembler** class that takes a given assembly source file and assembles it into an instance of a **Program** class.
* **Program.java**
  + Implements the **Program** class, which is used to represent the assembled program in memory and provides the method for rendering the program into binary form.
* **Instruction.java**
  + Each instance of the **Instruction** class represents a single “Wi-11” assembly code machine instruction.
* **InstructionDefinition.java**
  + The **InstructionDefinition** class is used by the **Assembler** to turn the source code into the binary representation once an assembly instruction has been matched with the proper definition.
* **LiteralTable.java**
* **Operand.java**
* **OperandDefinition.java**
* **Symbol.java**
* **SymbolTable.java**

The Assembler folder contains the subfolder “/tests” that contains the test assembly files that were used in the testing of the Assembler

The Assembler folder also contains the subfolder “/docs” that contains the User’s Guide, Programmer’s Guide and the Testing Plan. Contained within the “/docs” directory is another subfolder “/Meeting Minutes” that contains the minutes for each design meeting that took place.

## Design Conventions

Across the files in our implementation, several design conventions were followed. These include the capitalization of class names, like **Main** or **InstructionDefinition**, lower-case names for packages and variables, and mixed-case names for method titles like “getSource”.

## Module Inter-Relationships

# Data Structures

There are several important data structures that are used by the assembler in the creation of the object file and the listing file. The following is a listing of the data structure, the file it appears in, the usage of the data structure, its implementation, and its invariant.

* Object: “Instruction” defined by Instruction.java
  + Represents an individual “Wi-11 Machine” instruction
  + Implemented as an **InstructionDefinition,** an array of **Operand**’s and two strings defining the name and the source of the **Instruction**
  + Invariant: the **Instruction** must be a properly formed “Wi-11 Machine” instruction
* Object: “InstructionDefinition” defined by InstructionDefinition.java
  + Represents a certain type of assembly instruction
  + Implemented as a **String** representing the instruction name, an array of **int** that represent the binary values of the final binary code, an array of **OperandDefinition**’s, and an **int** representing the size of the instruction
  + Invariant: the provided **Instruction** must be a properly form “Wi-11 Machine” instruction
* Object: “LiteralTable” defined by LiteralTable.java
  + Represents a table of literals accumulated during the first assembler pass
  + Implemented as a **HashMap<Integer, Integer>();**
  + Invariant: each literal in the table must have a unique address
* Object: “Operand” defined by Operand.java
  + Represents a value (a **Symbol**, register id, literal, or immediate value) to be used by an **Instruction**
  + Implemented as an **OperandDefinition**, a **String**, and an **OperandType**
  + Invariant: An **Operand** must be a valid **Symbol**, register, literal or immediate value
* Object: “OperandDefinition” defined by OperandDefinition.java
  + Represents the acceptable values of **Operand**’s defined for the “Wi-11 Machine”
  + Implemented as an array of **OperandType**
  + Invariant: the **Operand** must be a valid **OperandType**
* Object: “OperandType” defined by OperandType.java
  + Represents the operand types available in the “Wi-11 Machine”
  + Implemented as an enumeration of valid operand types
* Object: “Symbol” defined by Symbol.java
  + Represents the name/value pair of a defined symbol
  + Implemented as **String** and an **int** as well as a **boolean** that determines whether or not the **Symbol** is relocatable or not
* Object: “SymbolTable” defined by SymbolTable.java
  + Represents the internal mappings of names to **Symbols**
  + Implemented as a **HashMap<String, Symbol>();**
  + Invariant: Only unique **Symbols** can be entered into the **SymbolTable**

# 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 entry point into the assembler program. It accepts the command line arguments from the user, such as, the file containing the source code, the specified object output file, and an option to generate a listing.
* **Program**
  + Description: The **Program** component holds the in-memory representation of the assembly program. The **Program** can then be rendered into its binary representation.
* **Assembler**
  + Description: The **Assembler** component

## Implementer-Side

### Main

* Description: The **Main** class contains the main method which is the entry point into the assembler program.
* State: None
* Algorithm:

If ( |args| < 2 OR args[0] = “--help” OR args[0]=”/h”) then

Display usage information

For int x = 2, x < |args| do

If ( args[x] = “-l” then

x++ and look at next args[x]

Set generateListing to true

Else

Display usage information

filename = args[0]

outfile = args[1]

load all of filename into a string

create new Assembler

assemble code

generate code

output code to outfile

close output streams

* **Methods in Main**
* Method Name: **Main**
  + Description: main class in class **Main**
  + Parameteres: args[]
  + Requires: input file
  + Alters: outfile
  + Returns: void
  + Throws: IOException
* Method Name: printUsageInformation
  + Description: Prints usage information to the console. Shows:

"Usage: java Main inputfile outputfile [options]

Inputfile Specify path to input assembly file.

Outputfile Specify path to output object file.

-l Generate and display source code listing."

* + Parameters: None
  + Alters: System.out
  + Ensures: System.out = #System.out + [usage info]
  + Returns: void
* Method Name: readAllText
  + Description: Reads all text in the file at the given path location into a string.
  + Parameters: **String** filename - Path to desired file
  + Requires: valid file
  + Returns: A string containing all the data existing in the desired file.
  + Throws: IOException
* Method Name: writeAllText
  + Description: Writes all the given text to the file existing at the path location
  + Parameters: **String** filename - Desired output file name, **String** data – data to be written to file
  + Alters: The specified file on the file system
  + Ensures: The specified file is overwritten or created if it does not exist on the file system
  + Returns: void
  + Throws: IOException

### Assembler

* + Description: The **Assembler** uses a constant instruction definition table to complete the first pass of the assembly process.
  + State: none
  + Alogrithm:

begin

foreach (line in lines)

if (first char is ‘;’)

line is a comment

increment line number

continue

endif

if (chars between label, operation, and operands != spaces)

add new Error message at current line number

if (Symbol Table contains this symbol)

symbol already exists

add new Error message at current line number

endif

else

if (instruction != “.EQU”)

add symbol to table

endif

if (instruction is “.ORIG”)

if (“.ORIG” is properly formed)

store segment information

else

add Error message

endif

else

if (instruction = “.EQU”)

if (“.EQU” is properly formed)

add to Symbol Table

else

add Error message

endif

else

if (instruction = “.FILL”)

if (“.FILL” is properly formed)

create new Operand

else

add Error message

endif

else

if (instruction = “.STRZ”)

if (“.STRZ” is properly formed)

create series of Operands representing the string ending with last Operand equal to 0x000

else

add Error message

endif

else

if (instruction = “.END”)

if (“.END” is properly formed)

add to the Symbol Table

else

add Error message

endif

else

if (instruction = “.BLKW”)

if (“.BLKW” is properly formed)

create Operand and update Location Counter to next position

after .BLKW

else

add Error message

endif

else

invalid instruction

add Error message

endif

if (program length exceeds one page of memory)

add Error message

endif

if (program loads outside of addressable memory range)

add Error message

endif

if (Error messages > 0 )

output Error messages

endif

return new program

end

### ByteOperations

* Description: A utility class that is used to obtain specific bits in a given hexadecimal number.
* **ByteOperations Methods:**
* Method: parseHex
  + Description: This method converts a hex number represented in a String to an integer value.
  + Parameters: **String** hex
  + Ensures: hex is a valid hexadecimal number representation.
  + Returns: an integer value representing the input hex value.
* Method: extractValue
  + Description: This method extracts bit values from a given integer.
  + Parameters: **int** value, **int** start, **int** end
  + Requires: start < end
  + Returns: an integer value representing the extracted bits.
* Method: getHex
  + Description: This method gets a string representation of a given integer value.
  + Parameters: **int** value, **int** numCharacters
  + Ensures: All but the least significant hex digits specified by numCharacters are removed.
  + Returns: a string representation of the integer value in hexadecimal form.
* Method: getBit
  + Description: This method returns true if the bit in “value” at the specified right-aligned index is 1.
  + Parameters: **int** value, **int** index
  + Returns: A **boolean** value indicating if the specified bit is 1.

### Instruction

* Description: Each instance of this class represents an assembly instruction that contains a definition, a name and a collection of **Operand**s.
* State: private **InstructionDefinition** definition, private **Operand**[] operands, private **String** source, private **String** name
* **Instruction** Methods
* Method: setDefinition
  + Description: Sets the definition of this Instruction, as well as the definition of all this Instruction's Operands. Note: setOperands must be called first.
  + Parameters: InstructionDefinition definition

### InstructionDefinition

### LiteralTable

### Operand

* Description: An **Operand** instance represents a value (a **Symbol**, register identifier, literal, or immediate value) to be inserted into an instruction.
* State: private **OperandDefinition** definition, private **String** value, private **OperandType** type
* **Operand** Methods:
* Method: setDefinition
  + Description: Sets the definition of this **Operand**. No work is done at this point, except some basic error checking for immediate values.
  + Parameters: **OperandDefinition** definition
  + Ensures: this.definition = definition
  + Alters: this
  + Throws: Exception e
* Method: getType
  + Description: Returns what type of **Operand** this is.
  + Returns: **OperandType** the type of the current **Operand**
* Method: insert
  + Description: Inserts the binary value of this **Operand** into the given instruction (ops). The given **SymbolTable** and **LiteralTable** are used to resolve the binary value of the **Operand.**
  + Parameters:
    - **int**[] ops
      * the op code of the assembly instruction
    - **SymbolTable** symbols
      * the **SymbolTable** used to create this instruction
    - **LiteralTable** literals
      * the **LiteralTable** used to create this instruction
  + Throws: Exception e
* Method: determineType
  + Description: Given the string representation of an **Operand**, returns the **OperandType** that describes it.
  + Parameters: **String** value
  + Returns **OperandType**
* Method: getValue
  + Description: Gets the binary **Operand** value represented by the given string. The **SymbolTable** and **LiteralTable** are used to resolve the binary value. Note: relocatable immediate values are NOT relocated by this function. No **OperandDefinition** is taken in this version, so no sanity checking is done.
  + Parameters: **String** value, **SymbolTable** symbols, **LiteralTable** literals
  + Returns: **int** the value of the Operand
  + Throws: Exception e
* Method: getValue
  + Description: Gets the binary **Operand** value represented by the given string. The **SymbolTable** and **LiteralTable** are used to resolve the binary value. Note: relocatable immediate values are NOT relocated by this function.
  + Parameters: **String** value, **SymbolTable** symbols, **OperandDefinition** definition, **LiteralTable** literals
  + Returns: **int** the binary value of the Operand
  + Throws: Exception e
* Method: parseConstant
  + Description: Parses a register identifier, literal value, decimal value, or hexadecimal value into an integer number.
  + Parameters: **String** value
  + Returns: **int** value of the **Operand**
  + Throws: Exception e

### OperandDefinition

* Description: An **OperandDefinition** states which values are acceptable for **Operands** that use this definition, and where to insert the binary value into the resulting executable code.
* State:
  + private **int** mostSignificant
    - Right-aligned index of the most significant bit this definition's **Operands** are allowed to affect in the executable code.
  + private **int** leastSignificant
    - Right-aligned index of the least significant bit this definition's **Operands** are allowed to affect in the executable code.
  + private **int** opIndex
    - For **Instructions** that map to multiple memory addresses, this index identifies which memory address to insert the **Operand** into.
  + private **boolean** relocatable
    - True if this definition's **Operands** are relocatable, false if not.
  + private **OperandType**[] acceptableTypes
    - Any **Operands** that use this definition must have a type that is contained in this collection.
* **OperandDefinition** Methods:
* Method: isAcceptable
  + Description: Returns true if the given **Operand** meets the requirements of this definition. Namely, the type of the **Operand** must be contained in this definition's collection of acceptable **OperandTypes**.
  + Parameters: **Operand** operand
  + Returns: **boolean** true if and only if the **Operand** is valid
* Method: isAcceptable
  + Description: Returns true if the given **OperandType** is contained in this definition's collection of acceptable **OperandTypes**.
  + Parameters: **OperandType** type
  + Returns: **boolean** true if and only if the **OperandType** is valid
* Method: getMask
  + Description: Returns a bitmask which, when logically anded together with a binary **Operand** value, will zero out any bits which are not allowed to be affected by the value.
  + Returns: **int** bitmask to eliminate unused bits in the **Operand**
* Method: getMostSignificantBit
  + Description: Gets the right-aligned index of the most significant bit this definition's **Operands** are allowed to affect in the executable code.
  + Returns: **int** the **Operand’s** most significant bit
* Method: getLeastSignificantBit
  + Description: Gets the right-aligned index of the least significant bit this definition's **Operands** are allowed to affect in the executable code.
  + Returns: **int** the **Operand’s** least significant bit
* Method: getMinimumAllowedValue
  + Description: Gets the minimum value this definition's **Operands** are allowed to have.
  + Returns: **int** the minimum allowed **Operand** value
* Method: getMaximumAllowedValue
  + Description: Gets the maximum value this definition's **Operands** are allowed to have.
  + Returns: **int** the maximum allowed **Operand** value
* Method: isSigned
  + Description: Returns true if and only if this definition accepts signed **Operand** values.
  + Returns: **boolean** indication of whether this definition accepts signed **Operand** values
* Method: getOperationIndex
  + Description: For **Instructions** that map to multiple memory addresses, this index identifies into which memory address to insert the **Operand**.
  + Returns: **int** the memory address into which to insert the **Operand**
* Method: isRelocatable
  + Description: Returns true if this definition’s **Operands** are relocatable, false if not.
  + Returns: boolean true if and only if the **Operand** is relocatable
* Method: getAcceptableTypes
  + Description: Any **Operands** that use this definition must have a type that is contained in this collection.
  + Returns: **OperandType**[] of valid **OperandTypes**

### OperandType

* Description: Enumerated type that establishes the possible operand types:
  + IMMEDIATE
    - represents a constant value
  + REGISTER
    - corresponds to a register R0-R7
  + LITERAL
    - references a literal
  + SYMBOL
    - references a **Symbol** defined in the **SymbolTable**
  + STRING
    - used for the .STRZ pseudo-operation

### Program

* Description: A **Program** instance contains an in-memory representation of an assembly program, which can be rendered into binary form with the getCode function.
* State:
  + private **SymbolTable** symbols
    - Contains the names and values of all **Symbols** defined in this **Program**.
  + private **LiteralTable** literals
    - Contains the addresses and values of all literals defined in this **Program**.
  + private **List**<**Instructions**> instructions
    - Contains all **Instructions** (source code, binary code, and Operand values) in this **Program**.
  + private **int** startAddress
    - The address (relative to the origin) at which to start execution.
  + private **int** origin
    - The absolute address of the first memory slot used by the **Program**. Any relocatable values in the **Program** are offset by this address.
  + private **String** name
    - Name for the program. Used as the segment name in the header record.
  + private **boolean** isRelocatable.
    - True if the program is relocatable.
* **Program** Methods:
* Method: getCode
  + Description: Gets the object code for this, optionally displaying a listing for the user.
  + Parameters: **boolean** printListing
  + Returns: a **String** representing the object file; if (printListing) then outputs the listing to the screen.
  + Throws: Exception e

### Symbol

* Description: A **Symbol** is a name-value pairing that has a flag for whether or not the symbol is relocatable or absolute.
* State: private **String** name, private **int** value, private **boolean** isRelocatable
* **Symbol** Methods:
* Method: getName
  + Description: Returns the name of this **Symbol**
  + Returns: **String** this.name
* Method: getValue
  + Description: Gets the value of this **Symbol**
  + Returns: **int** this.value
* Method: isRelocatable
  + Description: Returns true if this **Symbol** is relocatable, false if not.
  + Returns: **boolean** this.isRelocatable

### SymbolTable

* Description: An instance of this class maintains a mapping between **Symbols** and their values.
* State: private **Map**<**String, Symbol**>
* **SymbolTable** Methods:
* Method: define
  + Description: Adds a given **Symbol** to this **SymbolTable**.
  + Parameters: **Symbol** symbol
  + Alters: this
  + Ensures: this = #this + symbol
* Method: define
  + Description: Defines an alias for another **Symbol**.
  + Parameters: **String** alias, **String** target
  + Alters: this
  + Ensures: this = #this + alias
* Method: hasSymbol
  + Description: Returns true if this table contains a **Symbol** that matches the given name.
  + Parameters: **String** name
  + Returns: **boolean** true if and only if name matches a **Symbol** in this
* Method: get
  + Description: Gets the **Symbol** mapped to the given name, or null if none exists in this.
  + Parameters**: String** name
  + Returns: **Symbol** corresponding to name input
* Method: size
  + Description: Gets the number of defined **Symbols** in this table.
  + Returns: **int** number of defined **Symbols**