Group 1: Final Project Report

VM252 Debugger Usage, Features, and Implementation

Adam Mertzenich, Kritib Bhattarai, Michael Musa

Tuesday, December 14th, 2021

1 TODO Application Usage (Need Images)

In order to use the application you will need to compile the Main.java file. This can be done using the following commands in the root project directory:

```
javac edu/luther/cs252/group1/Main.java
java edu.luther.cs252.group1.Main
```

Alternatively, you can download the precompiled java jar file from GitHub and run it using `java -jar vm252-project.jar`.

After launching the application you can start using it. First, use the file+open menu and select a valid VM252 object file. In this document we will be using a simple multiplication program. You can verify that the object file has been loaded by seeing that the object file location has been loaded in the menu bar and that the memory table has values in the cells. The current program counter will be rendered as the TK.color.yellow.

Running the program to completion can be done by selecting the "Run" menu near the top right of the screen and then pressing the "Run" button within. You can also add a delay in milliseconds in the input box to the right of the "Run" menu and use the "Run", "Start", and "Pause" buttons to slowly step through the program. Alternatively, you can manually click "Execute Next" on the left hand side of the screen to run one instruction at a time. If you need to reset the program counter and accumulator use the "Z" button in the top right of the window's menu bar. You can also change the program counter and accumulator using the related input boxes on the right side panel by entering a number and pressing enter.

Breakpoints will result in the program being paused if it's running and notify you with a popup, or just a popup if you are manually executing each instruction. You can toggle breakpoints at a memory address by right clicking a cell of the memory table and they will change to be TK.color.color. Alternatively you can add breakpoints by inputting the source-line number relating to an instruction in the "Breakpoint Line #" input box on the top left of the left side panel. Below you will find a "Clear Breakpoints" button to disable all breakpoints.

There are two different views of the memory, the single-byte hex representation and the double-byte hex representation. In the single-byte hex view you see each cell representing a byte of memory, the two-byte hex view displays two bytes in each cell. You can double click a cell and enter a new value to change the memory (0-FF for single-byte, 0-FFFF for double-byte). Both tables will automatically be updated to represent changes in the memory model. You can see the next instruction to be executed in the bottom panel of the screen as well as the currently selected memory address location.

2 DONE Minimum Requirements

Getting help is done by hovering over components and reading the tool-tip or reading the popup boxes supplied with the menu bar's "Help" section (`h`). You can quit the application by just closing the window (`q`).

We have met all of the minimally required commands. Altering the accumulator and program counter for the `aa` and `ap` commands can be done using the input boxes on the right side ProgramStatePanel. This also covers the displaying of the program counter and accumulator which is part of the `s` command, the rest of the `s` command is fulfilled with the "Next Instruction" label on the bottom of the screen (instruction tracing `t` is also completed simultaneously).

The contents of the machine's memory and the portions containing object code are displayed in our MemoryTable in the center (`ob` and `mb`). Altering the memory is done by editing the cells of the table with a new unsigned hex value (`amb`).

Breakpoints can be added by right clicking on a cell of the table, changing the text color as an indicator (`ba`). Breakpoints will be triggered when running instructions using the "Execute Next" button on the left side button panel or using the "Run" menu items (`n`). When done executing a program you can reinitialize the program counter and accumulator using the "Z" button in the top menu (`z`).

Our "Run" menu and the related delay input box fulfill the `r` command requirements. Without any delay entered you can select "Run" from the menu to execute the entire program. Alternatively, you can set a delay value in milliseconds in the input box and when you run it will slowly step through the program, with the ability to "Pause" and "Start" again. All views are automatically updated.

3 DONE Beyond the Minimum

Our VM252 application fulfills the majority of the "extra" commands, going beyond the minimally required commands. You may use the "Breakpoint Line #" input box on the left side panel to enter a source line number to add a breakpoint and can click the "Clear Breaks" button below it to clear all breakpoints (`bl` and `cb`).

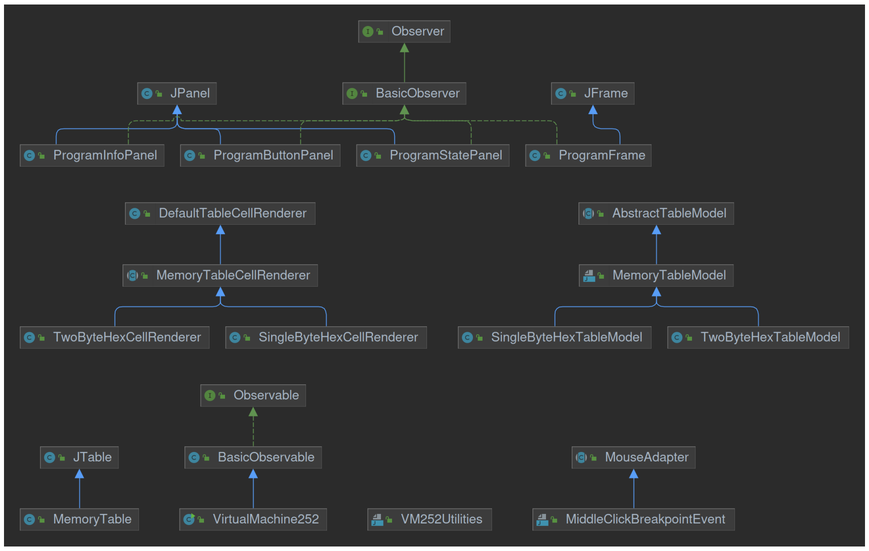
Using the "Double-Byte Hex" tab in the top center of the memory view you can choose to view the machine memory and object code as 2-byte hex data (`od` and `md`). You can edit these cells to new unsigned hex values for the `amdx` command functionality.

4 TODO Roles and Contributions

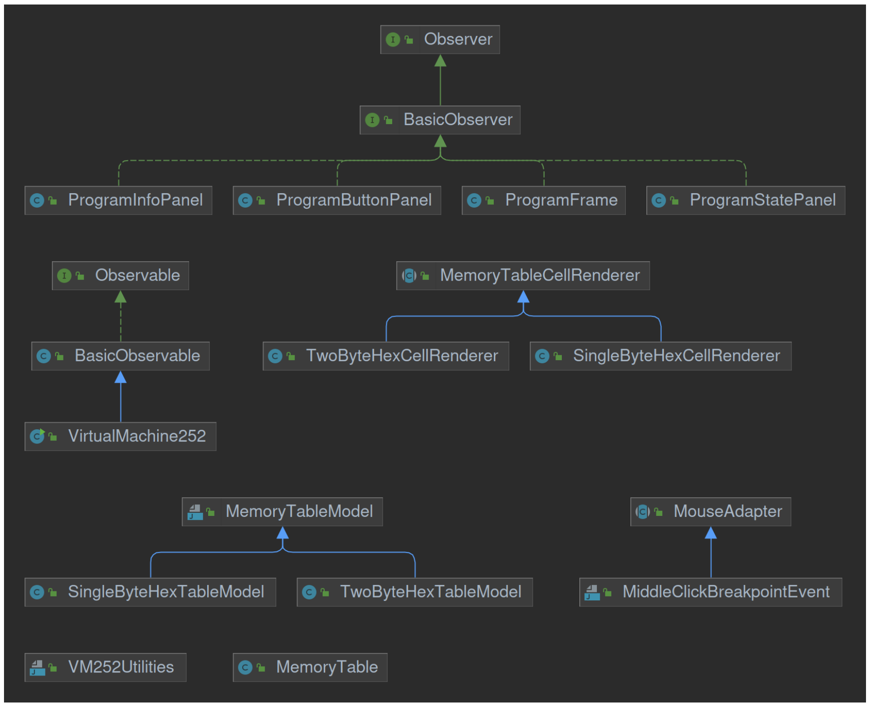
5 DONE Libraries Used

Java's Swing and the associated Abstract Window Toolkit graphical user interface libraries were the primary library for our application. This is what we used to create the entire user interface and all of the sub-components. Nothing outside of the Java Standard Library was used to prevent any unneccesary complication.

6 DONE Class Relationship Family-Tree Diagram



7 DONE Design Patterns



We heavily rely upon both the Observer-Observed and the Model-View-Controller design patterns (hierarchy image at the bottom of this section). As an overview, we used implementations of BasicObserver to watch for changes on our VirtualMachine252 model (BasicObservable). Every relevant change in the model would notify everyone watching it. Our classes are held within packages representing observation, models, and views/controllers.

Our views and controllers were often times combined thanks to the dynamic nature of many Java Swing components and tools at our disposal (contained within our modelview package). Our model, the VirtualMachine252 class, was where all operations on the memory are done through various accessor and mutator methods. At the end of every method that caused changes to the program state the announceChange() method is called to keep every view up to date.

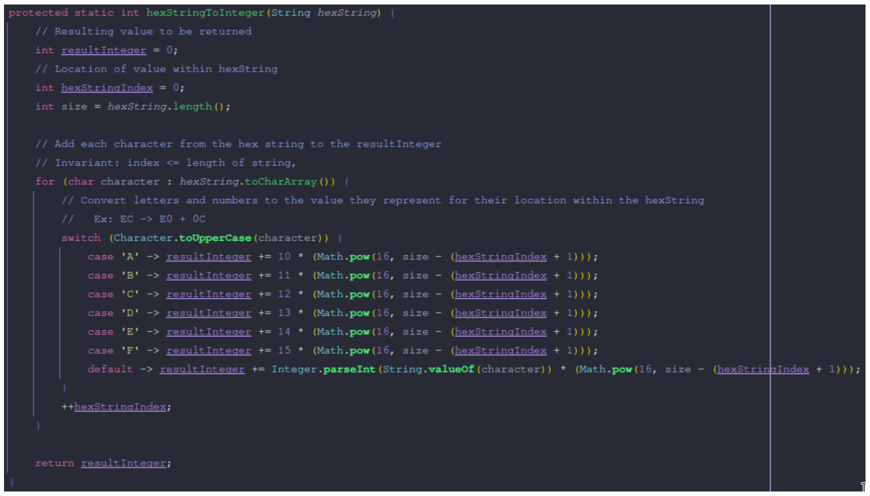
The ProgramStatePanel contains program counter and accumulator input boxes. These serve as both views and controllers. Editing the text of an input box and pressing enter will modify the modify and in turn update the MemoryTable (and other related views). Conversely, when running the program the state panel would also be updated as it views the new memory state. The aforementioned MemoryTable is our primary view and controller since it allows you to access and edit the memory, dynamically updating the "Next Instruction" indicator among others.

The MemoryTable has both a MemoryTableCellRenderer and a MemoryTableModel, each with seperate implementations for the single and two byte views/controllers. These renderers and models are the principal way memory is viewed and controlled in the VirtualMachine252 model (implementation details in the Original Coding section). These classes allow the tables to view and control the memory similar to how a spreadsheet would work, using the VirtualMachine252 memory as the model behind the scenes.

8 DONE Original Coding

The bulk of the original coding in our application belongs to the "Run" menu bar item and both of the table cell renderers and table models (SingleByte and DoubleByte). Utilizing the components such as JTable and by extending the default renderers allowed us to substantially lower the amount of code we otherwise would have needed to add to get our program operational.

First we will discuss our abstract MemoryTableModel class and it's children, Two/SingleByteHexTableModel. The abstract class contains two helper functions. The first is `intToHexString(int originalInteger)` which takes an integer and returns a hex value as a string (used for displaying memory contents in a cell using hex). The other method we have is `hexStringToInteger(string hexString)` which returns an integer value based on a hex string, such as "3D" (used for converting and storing user input in memory). Below you can see the primary relevant portion of the code which adds each element of a string into the result integer.

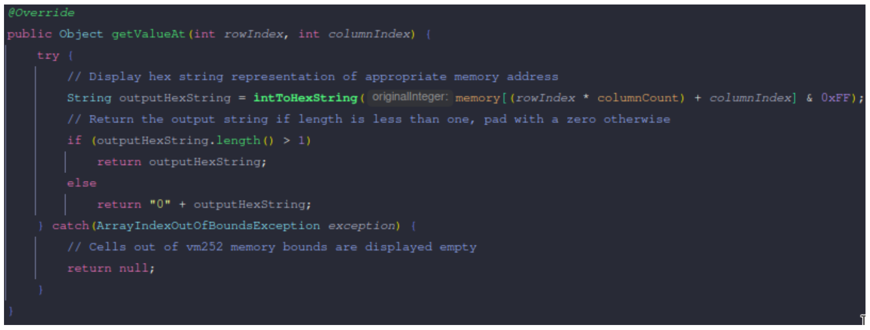


```java
protected static int hexStringToInteger(String hexString) {
    // Resulting value to be returned
    int resultInteger = 0;
    // Location of value within hexString
    int hexStringIndex = 0;
    int size = hexString.length();

    // Add each character from the hex string to the resultInteger
    // Invariant: index <= length of string,
    for (char character : hexString.toCharArray()) {
        // Convert letters and numbers to the value they represent for their location within the hexString
        //   Ex: EC -> E0 + 0C
        switch (Character.toUpperCase(character)) {
            case 'A' -> resultInteger += 10 * (Math.pow(16, size - (hexStringIndex + 1)));
            case 'B' -> resultInteger += 11 * (Math.pow(16, size - (hexStringIndex + 1)));
            case 'C' -> resultInteger += 12 * (Math.pow(16, size - (hexStringIndex + 1)));
            case 'D' -> resultInteger += 13 * (Math.pow(16, size - (hexStringIndex + 1)));
            case 'E' -> resultInteger += 14 * (Math.pow(16, size - (hexStringIndex + 1)));
            case 'F' -> resultInteger += 15 * (Math.pow(16, size - (hexStringIndex + 1)));
            default -> resultInteger += Integer.parseInt(String.valueOf(character)) * (Math.pow(16, size - (hexStringIndex + 1)));
        }
        ++hexStringIndex;
    }

    return resultInteger;
}
```

This loop uses the formula $16^{length-location}$ where the length is the size of the string and the location is the index of the current character. When you iterate over a string adding the result to a variable each time the final value will be the resulting integer (we use the value 16 due to hexadecimal being base 16).

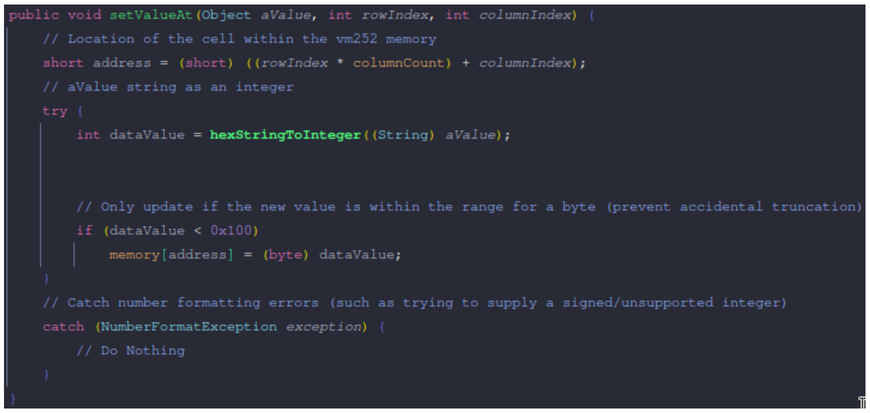
SingleByteHexTableModel and TwoByteHexTableModel both extend the MemoryTableModel and make use of it's methods. Most of the code in these concrete classes is boilerplate because they must implement the AbstractTableModel methods, but the getValueAt and setValueAt methods required original code.

Below we will explain the methods as implemented in the SingleByteHexTableModel. These methods are nearly identical in the TwoByteHexTableModel except for minor changes to use two bytes instead of one, the differences are minimal enough that they do not require focus.



```
@Override
public Object getValueAt(int rowIndex, int columnIndex) {
    try {
        // Display hex string representation of appropriate memory address
        String outputHexString = intToHexString( originalInteger: memory[(rowIndex * columnCount) + columnIndex] & 0xFF);
        // Return the output string if length is less than one, pad with a zero otherwise
        if (outputHexString.length() > 1)
            return outputHexString;
        else
            return "0" + outputHexString;
    } catch(ArrayIndexOutOfBoundsException exception) {
        // Cells out of vm252 memory bounds are displayed empty
        return null;
    }
}
```

The logic for getting the value of a cell, at the cross between rowIndex and columnIndex, is surprisingly rudimentary. Using the formula $((rowIndex * columnCount) + columnIndex)$ we can find the location of that cell in memory. We then use the bitwise AND operator with the hex value FF so that the value of memory doesn't appear signed when converted to an integer. The intToHexString method converts the value of memory into a hex string and a simple if statement pads the value so that each cell is always displayed two characters long. If the cell is out of the memory bounds the value is returned as null.



```
public void setValueAt(Object aValue, int rowIndex, int columnIndex) {
    // Location of the cell within the vm252 memory
    short address = (short) ((rowIndex * columnCount) + columnIndex);
    // aValue string as an integer
    try {
        int dataValue = hexStringToInteger((String) aValue);


        // Only update if the new value is within the range for a byte (prevent accidental truncation)
        if (dataValue < 0x100)
            memory[address] = (byte) dataValue;
    }
    // Catch number formatting errors (such as trying to supply a signed/unsupported integer)
    catch (NumberFormatException exception) {
        // Do Nothing
    }
}
```

Setting the value of a cell is essentially the reverse of accessing it. The same formula is used to find the appropriate address in memory for the rowIndex and columnIndex. The dataValue which the user entered in a cell is treated as a string and is converted to an integer using hexStringToInteger. An if statement makes sure that the dataValue is not too large to fit in a single byte of the memory ray, if it's too large it does not modify the memory.

9 DONE Persistent Information

We chose to not store persistent information for our debugger in favor of getting as many other features implemented with a good quality standard. If we were to store information, such as settings, we would probably use a binary format to store colors, boolean values, etc.

10 TODO Other Features