Final Project: VM252Debugger

Supreme Paudel, Sabina Dahal, and AbbyWood

Contents

User-manual

User Manual.pdf

Capabilities

Our application(debugger) fulfills all the minimum functionalities and a few additional as well. The major functionalities this debugger performs are altering the contents of accumulator(aa), altering the contents of the program counter (ap), altering contents of the byte at MA to unsigned hex value HB (amb), setting a breakpoint at memory address(ba), prints help message(h), display all of the machine memory as bytes in hex(mb), execute next machine instruction(n), display the portion of machine memory holding object code as bytes in hex(ob), quit (q), run machine until error occurs or stop instruction is executed (r), display machine state (accumulator, program counter, etc.) (s), and reinitialize program counter to zero (z). There are GUI representations like display text area, buttons, pop-up box, and some icons in our debugger that perform user interaction, load and run object files, and perform other functions which are discussed in detail below.

Load and Run Object File:
With the help of a JFileChooser, the user can select a .vm252obj file for execution. The GUI gives an error message popup if the user tries to open some file with a different extension. Also, it gives a pop-up message when a user tries to press any button without selecting the object file to run. A small text field just below the SelectFile button shows the user-selected object file. When a user presses the Start button, given the user has already selected a file, the load and run method of the guiController class helps to run the file.

AA (alter contents of the accumulator)
Users can edit the accumulator counter values. To do that, they just need to type the integer in the text field and hit enter to trigger the actionPerformed method of the actionListener class and this will update the values and reflect those changes accordingly. The change in accumulator value is set using the setAccumulator method and updated using the overriding updateAccumulator method of the accumulatorPrinter class and the update is displayed in the Events Text area.

Ap(alters contents of the program counter)

Users can edit the programCounter values. To do that, they just need to type the integer in the text field and hit enter to trigger the actionPerformed method of the actionListener class and this will update the values and reflect those changes accordingly. The change in accumulator value is set using the setProgramCounter method and updated using the overriding updateProgramCounter method of the programCounterPrinter class and the update is also displayed in the Events Text area.

Amb(Alter contents of the byte at MA to unsigned hex value HB)

The class called alter_memory_address helps with altering the contents of the byte at the memory address to an unsigned hex value. We have a button called “Edit Memory”, once it is triggered, users are prompted for memory address input for both the memory address(program counter) and the hex value that they want to change. If the user provides an invalid address they get an error message. If the correct format of address for both memory value and hex value is given, the string input is parsed into memory address and hex value. Then, the setMemoryByte method from the VM252Model class is called to update the memory byte at a given address with a new value, causing it to alter the hex value at the specific memory address.

S (displays machine state):

The current machine state shows the updated accumulator value, program counter value, breakpoints, yellow highlights, output, and Events textarea. As soon as the program is executed, the user can see the accumulator and program counter value changing and when the user sets new AC and PC values, that is also displayed in both the AC or PC text field as well as in the Events Text area. The output is displayed in the Output Text area by just appending into it. Similarly, when the breakpoint is set, it is displayed by highlighting the lines and storing them in array lists. The yellow highlight is used to show the execution of the source code smoothly.

prints help message(h)

There is a “help” sign icon on the top of the right side that pops up a help box once a user clicks which explains major functionalities of the debugger like selecting files, and how the buttons work.

display all of machine memory as bytes in hex(mb)

The GUI displays the contents of the entire memory as a program runs in a scrollable text display. The display_entire_memory() method from the code_display class is used to deal with displaying all of machine memory as bytes in hex. Within the method, the for loop is used to iterate over all memory addresses of the virtual machine, from 0 to the total size of the VM's memory and fetches the data stored at that address and is displayed in a specific format in memory_display_one text Area.

Z (reinitialize program counter to zero)
The execute_Again(reset) button is used to restart the program from the start resetting the Accumulator and Program counter value to zero. To do that, the restart icon needs to be clicked to trigger the actionPerformed method of the actionListener class. This also clears instruction_Display, input_code_area, memory_diaplay_one, memory_diaplay_two text areas, and breakpoints. Then create_simulation_machine() method is called to initialize the simulation machine and simulate the execution of the first instruction. Even though the machine restarts, the selected files remain loaded so no need to select a new object file to rerun.

Ba (set a breakpoint at address MA)
We created a "breakpointHandler" class that stores breakpoints at different memory addresses, and stores breakpoints at different line numbers of the text areas. The method called toggleBreakPoints within this class utilizes two ArrayLists, "breakpoints" for storing line numbers and "programCounterBreakpoints" for corresponding memory addresses, ensuring a synchronized tracking of breakpoints in the code and its execution state. The values at the same index in both ArrayLists are related, for example, if the "breakpoints" ArrayList has 1 in its 0th index and "programCounterBreakpoints" ArrayList has 2 in its 0th index, it means program counter of the program is 2 when we reach line 1. This class also has methods for adding and removing highlights and synchronizes the highlight across both the human-readable code and the text area designated for machine memory holding object code. The method check_if_breakpoints_hit() checks if the line is in programCounterBreakpoints ArrayList, if it is, the simulator is paused and the processBreakpoints method is called.

Ob (display the portion of machine memory holding object code as bytes in hex)
Within the class code_display, the display_code_in_memory_bytes_format () method is used to display the portion of machine memory holding object code as bytes in hex in a specific format. We used a specific format to display the machine memory i.e. "**[Addr %d] %02x %02x\n" and** appended it to the memory_display_two textArea to display in the application. We are displaying it in a format similar to human_readable format so that it makes more sense visually.

N (execute next machine instruction)
The "NextLine" button lets users go single-step through a program. This button throws a pop error message when the user tries to run the NextLine button without selecting the object file. As long as the machine is not stopped, In the text field " instruction_to_be_executed", the next instruction from a specific program counter is displayed.

R (Run machine until an error occurs or stop instruction is executed)
When a user clicks the run button, the instructions keep getting executed one after another unless the VM252Model object's stoppedCategory is either paused or stopped. The run functionality is built on top of next instruction's functionality.

- **Pause**
  The user can pause the program which stops the simulator. The user will need to click on 'run' or 'next instruction' to run the simulation again. We added a new category called "paused" to the VM252Model.stoppedCategory to implement the pause button which sets our VM252Model object's stoppedCategory to paused, and the "Step" method of the VM252Stepper now only runs when the stoppedCategory is not stopped and not paused.
- **Resume**
  The user can resume the program by just pressing the 'run' button again which just resumes the execution again.
- **Stop**
  Stopping the simulation sets the VM252Model object's stoppedCategory to stopped. The user is unable to continue/ pause the application once a simulation stops. The only option is then to restart the simulation again from the beginning.
- **Speed**
  Users can select different speeds to determine the rate at which the instructions should be executed. The speed feature has been implemented with the use of the Timer class of the swing api. You can find more information about speed on this section here

Above Minimum requirements:
Oi (Display the portion of machine memory holding object code as instructions, data, and labels)

We are displaying code in human-readable format by fetching memory bytes one after another and converting them to instructions. If the instruction only consists of the symbolic opcode such as STOP, or INPUT, (instructions with a single word), we just display their symbolic code. If the memory address we are fetching is object code that holds data, then we display what the variable name is and the data the address is storing which consists of 2 bytes of data. The variable name is found with the help of VM252Utilities.addressesWhichHoldsObjectCodeData which is a hash map with address(PC) values as hash keys, and the variable name designated for that address as the values. If the memory address we are fetching holds executable code but belongs to the VM252Utilities.addressesWhichHoldsObjectCodeData hash map, this means this is of the type (variableName: INSTRUCTION) e.g main: INPUT. These are just instructions that are involved in jump statements/ while loops used to jump the PC to a certain PC value(example JUMP main) will change(jump) to wherever main is(wherever the memory address of main is).

Cb (clearAllbreakpoints)
To implement the clearAllBreakPoints() method, the clear() method is used to remove all breakpoints from “breakpoints” and “programCounterBreakpoints” ArrayLists that remove breakpoints from both human_readable-format and memory_address.

md(Display all of machine memory as 2-byte data in hex)
The GUI displays the contents of the entire memory as a program runs in a scrollable text display. The display_entire_memory() method from the code_display class is used to deal with displaying all of machine memory as 2-byte data in hex. Within the method, the for loop is used to iterate over all memory addresses of the virtual machine, from 0 to the total size of the VM's memory and fetches the data stored at that address and is displayed in a specific format in memory_display_one text Area.

q (Quit)
For the quit, the user can just close the debugger window.

Contributions

Sabina- Coded for Breakpoints, clearAllBreakpoints, AlterMemoryAddress(EditMemory), worked on Redo button functionalities, worked on the StopAnnouncer class, worked on loading object files with validation, wrote Capabilities documentation and special design and Implementation (Breakpoints and Highlights)

Supreme: Displaying and updating object code as both human-readable code and bytes and the entire memory with pseudo random values to uninitialized addresses at the start, implementing speed feature, adding yellow highlighter for the next instruction, let user modify ACC/ PC values and update accordingly, handle input, implement button functionalities

Abby- Created the GUI interface, focused on visuals and GUI panels management, coded the help button and helped with other button functionalities, handled validation for button presses, scheduled time and place to meet for group work, wrote User Manual and 3rd Party Library Usage Document.

Standard libraries used

Standard 3rd Party Libraries

- java.util.
    - ArrayList
    - Arrays
    - HashMap
    - Map
    - Scanner
    - Set
    - regex.Matcher
    - regex.Pattern
- javax.swing.
    - *
    - text.
        - BadLocationException
        - DefaultHighlighter
        - Highlighter.
            - HighlightPainter
        - AbstractDocument
        - Document
        - Element
- java.awt.
    - Color
    - Point
    - event.
        - MouseEvent
        - ActionEvent
        - ActionListener
        - MouseAdapter
- java.io.
    - File
    - IOException
- java.nio.
    - file.Paths

Java.util

ArrayList
Used to hold a list of elements like: elements in ‘button list’ and list of breakpoints placed.
Found in Program: DebugFrame.java, guiController.java, SimpleObservable.java, VM252Utilities.java.
Arrays

Used to list the buttons in ‘buttonlist’.
Found in Program: DebugFrame.java.
HashMap
Used to handle breakpoints, where they are and if they are placed or not.
Found in Program: DebugFrame.java, VMArchitectureSpecifications.java, VM252Utilities.java.
Map
Used to map out where breakpoints can be placed.
Found in Program: DebugFrame.java.
Scanner
Used to scan an imputed value and be able to make it any type needed.
Found in Program: DebugFrame.java , guiController.java, VMStepper.java
regex.Matcher
Used to interpret the compiled expressions the Pattern forms and displays that information in the ‘Event’ display and the speed button display.
Found in Program: DebugFrame.java.
regex.Pattern
Definition: A compiled representation of a regular expression
Used to acquire the value for the Program Counter and execution speed.
Found in Program: DebugFrame.java.

Javax.swing.

This library holds most/ all GUI capabilities. Used throughout the whole program, but mostly used to create GUI elements like: panels, buttons, labels, frames, and layout tools.
text.

- BadLocationException
  - Used to report bad locations JTextArea(trying to reference a location that does not exist
  - Found in Program: DebugFrama.java when catching and throwing this exception.
- DefaultHighlighter/HighlightPainter
  - Used to implement the yellow highlighter(showing what instruction is the simulation currently on) and the red highlighter(where the breakpoints are placed)
  - Found in Program: DebugFrame.java.
- AbstractDocument
  - Used to allow multiple readers but only one writer to implement various types of documents
  - Found in Program: DebugFrame.java.
- Document
  - Used to be the container that holds the text from the file selected so it can be read by the user.
  - Found in Program: DebugFrame.java.
- Element

- Used to determine if a particular line in the JTextArea can be clicked on. It allows the program to determine the exact text position of what has been clicked.
- Found in Program: DebugFrame.java.

Java.awt.

Color
Used to color the red and yellow highlighters and to color backgrounds of panels.

Found in program: DebugFrame.java
Point

event.

- ➔ MouseEvent
- ➔ ActionEvent
- ➔ ActionListener
- ➔ MouseAdapter

Java.io.

File
Used to create an abstract representation of file and directory pathnames. Allows users to go through files in their device and have the Debugger read the file and create a simulated version of the file.

Found in program: DebugFrame.java around line 1470 while selecting a file, VM252Utilities.java while processing the FileInputStream.

IOException
Used to signal when an IOException has occurred, constructing an error detail message. Examples include, but are not limited to: when the file chosen is not an .obj file, when input is not the right datatype, and when you press start before a file is selected.

Found in program: DebugFrame.java 5 while catching and throwing IOExceptions, guiController.java , VM252Stepper.java , VMUtilities.java.

Java.nio.

file.Paths
Used to get the icon images assigned to the respective button from the icons' respective .png file.

Found in program: in DebugFrame.java while creating each button.

Class relationships

UML class.pdf

MVC pattern

Our GUI debugger for the VM252 implements the Model-View-Controller (MVC) pattern in several key areas, ensuring efficient handling of interactions between data, user interface, and functionality. The main idea behind our MVC design principle is that ***changes in controller(user interactions) should lead to changes in model(if appropriate) and changes in model should then lead to changes in view(the GUI display).***

Here's an overview of how the MVC pattern is employed within our application:

1. **VM252Model :** This is the heart of the simulation and represents the code being simulated. Changes in user interactions (e.g., altering AC/PCC values from the GUI) updates the model, triggering appropriate update methods after they are announced. These changes in the model subsequently reflect in the associated views (GUI) through overridden methods
    - However, because we assign pseudo-random values to non-object code memory addresses, the **MemoryBytePrinter** calls its update method excessively at the start of the simulation when we load the file. So, rather than using this update method to update the GUI, we implemented a code_display class to make our GUI faster.

2. **Code Display :**
    - When a user performs actions affecting memory bytes(like pressing any of the program execution control buttons or altering memory bytes), the VM252 model for the memory addresses gets updated. This updated data is then displayed in various formats (as instructions, memory bytes, etc.). This implementation ensures that changes in the model reflect in the displayed code, thereby following the MVC paradigm by updating the view (code display) based on model updates.

3. **Handling Breakpoints:**
    - The MVC pattern is applied to handle breakpoints. The **BreakpointDisplay** class is responsible for highlighting text areas in the GUI where breakpoints are set or removed. Another class is the **breakpointHandler** class which has two data models(both arrays) one for the line where breakpoints have been set and the other for the memory address that those lines have, which is found by using regex as those lines have their memory address at the start of the line. These two arrays are related such that the value at index i of the breakpoint lines array will have it's address at index i of the breakpoint pc values array.
    - Changes in the controller (double-clicking on text areas) lead to modifications in the model (adding/removing breakpoint arrays). These changes trigger the addition or removal of highlights, updating the view to reflect changes made to breakpoints in the code.

Non-gui-related coding in the application

The following are some noteworthy highlights of our code that have nothing to do with putting the GUI's components together:

1. **Decoding memory bytes as human-readable instructions:**

    Decoding bytes as instruction involved two different phases:

    I. Reading info from the object file:

The first step involved reading the bytes containing the information about the symbolic address and creating a hash map mapped from the label's memory address to its label (variable name). Reading bytes holding symbolic address information involved reading bytes character by character for the label name until we encountered a '\0,'. After this, we would read the 32-bit integer which would provide the memory address associated with the label.

```
Unset

    61      'a'
```

```
00      '\0'
00      the 32-bit integer 2, hence the label a corresponds to memory address 2
00
00
02
```

Example: Sample symbolic address information bytes for an obj file with only variable 'a' whose data is stored in memory address 2

The second step was read the byte-content map to find the addresses where the object code would hold data. This would help us distinguish the variable names that was used for holding data.

```
Unset

01      the corresponding byte of the object code holds executable code

01      the corresponding byte of the object code holds executable code

00      the corresponding byte of the object code holds data

00      the corresponding byte of the object code holds data
```

Example: Sample bytes of the byte-content map of some obj file. The values of the bytes are either 00 or 01.

Finally, decoding instructions also involved knowing what the size of the byte-content map was as we would need to decode the bytes up to the memory address given by the byte content size.

II. Decoding instructions

All of this was done in the VM252Utilities.java file in the readObjectCodeFromObjectFile method of the VM252Utilities class. After this was done, whenever a user selected a file or restarted the simulation using the execute again method, a new simulation machine was created(a new instance of the Vm252 model and new instance of the

controller in addition to other things) which would also call the display_code_in_human_readable_format method of the code_display class which would decode the instructions in human-readable format.

The logic used while decoding instructions is as follows:

a. Fetch memory bytes one after another while converting them to instructions. Start from memory address of 0, get the instruction for this address using the *Instruction* inner class of the *VM252ArchitectureSpecifications* class and increment to the next address using the *nextMemoryAddress* of the *VM252ArchitectureSpecification*s.

b. If the address is of type object code that holds data, then we display what the variable name is and the data stored in the address is. The variable name is found with the help of *VM252Utilities.addressSymbolHashMap* and the data stored in that address can be fetched.

c. If the address holds executable code rather than data but belongs to the *VM252Utilities.addressesWhichHoldsObjectCodeData* array, then this means this instruction is of the type (variableName : INSTRUCTION) e.g main: INPUT. These could be instructions that are involved in jump statements/ while loops. For example JUMP main will change PC(jump) to wherever main is(wherever the memory address of main is).

d. If an address doesn't qualify for (2) and (3) as mentioned above, then we just decode the instruction normally. We still take care of substituting address byte values with variable names if the value belongs to the *VM252Utilities.addressSymbolHashMap*. For example, if 'subject' is a symbol used to store data in memory address 4, and an instruction is 20 04, this would be decoded to `STORE subject`.

2. Pseudo random memory value

The VM252Model byte value for addresses which don't have the object code have value 0. However, this isn't accurate as physical memory isn't initialized to a special value like 0. So, with the intention to make sure that the uninitialized memory bytes cannot be counted to contain zero, our implementation assigns pseudo-randomly generated values in the memory bytes when first started.

3. Speed feature with threads

The single-thread rule says that "*once a Swing component has been realized, all code that might affect or depend on the state of that component should be executed in the event-dispatching thread.*"

So, any changes to the GUI would have to be made in the event-dispatching thread. So, our implementation uses the Timer swing class to fire ActionEvent at a specified interval calculated based on the speed the user sets.

However, it was also important to stop this timer if the user has paused the simulation or if the simulated has stopped. So, the *timer_object*.stop() method would be called if the stopped status of the simulated machine is either paused or stopped. The actionEvent would call the stop method if the simulated machine was paused/stopped, otherwise call the step method of the machine stepper.

How multiple executions of a file is handled

The handling of multiple executions of the same **obj** file involves the reinitialization of the simulation environment. When a user selects the same **obj** file again for execution, several key actions take place to ensure a fresh start:

1. **Clearing Previous State:**
    - Breakpoints set in the previous execution are cleared.
    - Memory address values that were altered during the previous execution are reset to their initial state as they were when the file was first loaded. We will talk more about how this happens later.
2. **Creation of New Instances:**
    - The user pressing the "execute again" button triggers the ***create_simulation_machine*** method of the ***DebugFrame*** class.
    - This method creates a new instance of the ***VM252*** model and the GUI controller class. Additionally, it creates new instances of subclassed views (such as the accumulator printer, program counter printer, memory byte printer, highlighter printer etc.).
    - These objects are attached to the new model, essentially initializing a new environment for the execution of the **obj** file.
3. **Resetting Memory State:**
    - An important part of this process is calling the **load_file** method within the GUI controller class. This method reads the object code from the **obj** file again.
    - Subsequently, it sets the model's memory bytes from addresses ranging from 0 to 8191, effectively resetting any changes made by the user during the previous execution.

By implementing this approach, the GUI ensures that each execution of the same **obj** file is treated as a separate instance with a clean slate. This prevents interference between executions by clearing previous breakpoints, resetting memory address values, and initializing a new simulation environment. Users can re-run the code without persistence of changes made during prior executions.

Special design and implementation features

Icons

We have implemented icons to our basic buttons(Start, Pause, Next Line, Redo, Stop, and Help) to add a more aesthetic look to our Debugger program. We used java.nio.file.Paths to convert a string into a path to access the proper .png files for each icon. The icons cover the buttons, allowing for the user to click the icon to perform their respective button's actions.

Breakpoint highlights Synchronization

We have implemented synchronized highlights for the breakpoints with two text fields, one with instructions in human-readable format and the other with bytes. For the synchronization to work, we have two different Hashmaps for two areas whose main purpose is to store the highlight tag in the appropriate map. It helps to add highlighted lines to the hash map by checking if the lines have existing highlights. This happens within the addHighlightToLine method whose purpose is to add highlights to the same line in both text Areas. Similarly, the method removeHighlightFromLine() helps to check if the highlight tags are already in the specific hashmap for the text Areas, if found, it removes the highlight associated with this tag. In the method, synchronizeHighlights(), the addHighlightToLine() and removeHighlightFromLine() methods are called to perform synchronization in adding and removing breakpoints.

Code highlight

We have implemented highlights to each line to the human_readable format instructions, data, and labels for which the highlight moves along with the execution of the program. For this purpose, we created the class lineHighLightPrinter which has a method like getCurrentLine() uses a list to get a list of lines that returns the line number based on specific program counter value, and updateHighlighter() method adds and removes highlights to the line based on the line number.

THE END