SIMULIZER

User Manual

Charlie Street Kelsey McKenna Matthew Broadway
Michael Oultram Theo Styles

Thursday 17th March, 2016

1 Introduction

Thanks for downloading Simulizer! Simulizer is a piece of software that allows you simulate and visualize (hence Simulizer) the running of a MIPS processor in the comfort of your own home. With Simulizer, you don't get to just run programs, you get to write and edit files with our own integrated text editor. As well as this, Simulizer provides a whole range of options to help you understand your MIPS code (and the processor) that little bit more. If you want to see your code moving through the CPU, then choose our CPU visualization! If you want to see your algorithm running to check you've got you logic correct then a high-level visualization is the option for you. If you've never been able to get your head round pipelining then nows your chance, Simulizer visualises that too!

Simulizer also provides you with loads of handy debugging tools, such as a logger for standard, error and debug streams, as well as windows to let you peak inside the memory/registers of the CPU. Another really clever thing included is a JavaScript based annotation system. Gone are the days of adding print statements throughout your code, instead you can put it in the comments. You can control the high level visualizations, as well as large amounts of the CPU in just a little bit of JavaScript!

Finally, enjoy your time using Simulizer!

Contents

2 User Interface

The user interface is designed to be as configurable as possible, so that the application can fulfil your needs. Don't need to visualise the internals of the CPU? Just close the CPU visualiser. Need to make the editor a bit bigger? Then resize the editor. It's very simple.

2.1 Menu Bar

- **File**: Contains the standard settings found in most applications
  - **New** (`CTRL+N`): Creates a new blank program and opens the Editor Internal Window.
  - **Open** (`CTRL+O`): Opens an existing program and puts it in the Editor.
  - **Save** (`CTRL+S`): Saves the current program to the file loaded in the Editor.
  - **Save as**: Saves the current program to a new file.
  - **Options**: Opens the Options Internal Window.
  - **Exit**: Exits Simulizer.
- **Simulation**:
  - **Assemble and Run** (`F5`): Assembles the SIMP Program and (if it is a valid program) executes it. On an invalid program, hints to what went wrong will be displayed in the Editor.
  - **Pause/Resume Simulation** (`F6`): Pauses or Resumes the currently running/paused SIMP program.
  - **Single Step** (`F7`): On a paused SIMP program, this option completes one cycle of the simulated CPU.
  - **End Simulation** (`F8`): Completely ends the simulation and resets the CPU to it's initial state.
  - **Toggle CPU Pipelining**: Switches between the pipelined and non-pipelined CPU.
  - **Set clock speed**: Opens a dialog box so that you can change at what speed the simulated CPU is running at. Note: this is measured in Hertz, and setting this value too high may have performance issue.
- **Windows**: This contains a sub-menu with all the Internal Windows. This allows you to open and close each Internal Windows more easily.
  - **Close All**: Closes all open Internal Windows
- **Layouts**: Contains a list of all layouts saved in the layouts folder. This allows you to easily switch between different common workspace layouts.
  - **Save Layout**: Saves the current workspace layout to a new file
  - **Refresh Layouts**: Refreshes the list of layouts
- **Help**:
- **Debug**:

2.2 Internal Windows

Each pane inside the application is called an Internal Window. This section will give a brief description of what all the different Internal Windows are for, and why you might want to use them.

2.2.1 CPU Visualisation

CPU visualisation is for demonstrating how the MIPS processor fetches, decodes and executes assembly instructions. To use this view, you must set the clock speed to below 2Hz (see clock speed).

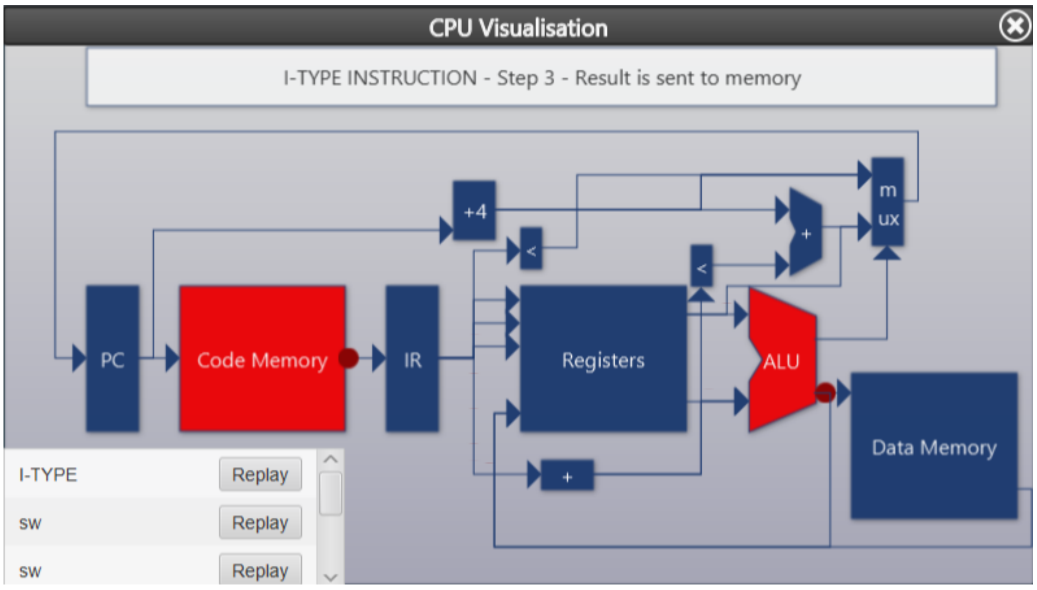
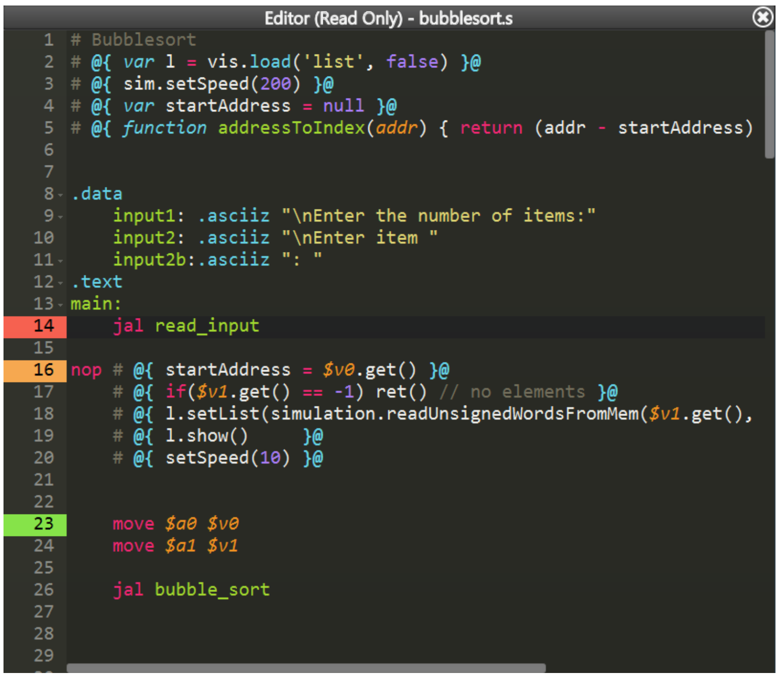


Figure 1: CPU Visualisation executing an I-Type instruction

For more information about the CPU Visualiser other low level visualisation see Low Level Visualisations

2.2.2 Editor

The editor is the place to write assembly code. The program that is contained in the Editor is the one to be run on the MIPS processor. You will most likely want to keep this window open (as without it, you can't run any assembly code).



```
# Bubblesort
# @{ var l = vis.load('list', false) }@
# @{ sim.setSpeed(200) }@
# @{ var startAddress = null }@
# @{ function addressToIndex(addr) { return (addr - startAddress)


.data
    input1: .asciiz "\nEnter the number of items:"
    input2: .asciiz "\nEnter item "
    input2b:.asciiz ": "
.text
main:
    jal read_input

nop # @{ startAddress = $v0.get() }@
    # @{ if($v1.get() == -1) ret() // no elements }@
    # @{ l.setList(simulation.readUnsignedWordsFromMem($v1.get(),
    # @{ l.show()      }@
    # @{ setSpeed(10) }@


    move $a0 $v0
    move $a1 $v1

    jal bubble_sort
```

Figure 2: Editor running Bubble Sort

2.2.3 High Level Visualisation

The High Level Visualisation window is where visualisations from the annotations are displayed. There purpose is to demonstrate what your SIMP program is actually doing from a more human understandable view.

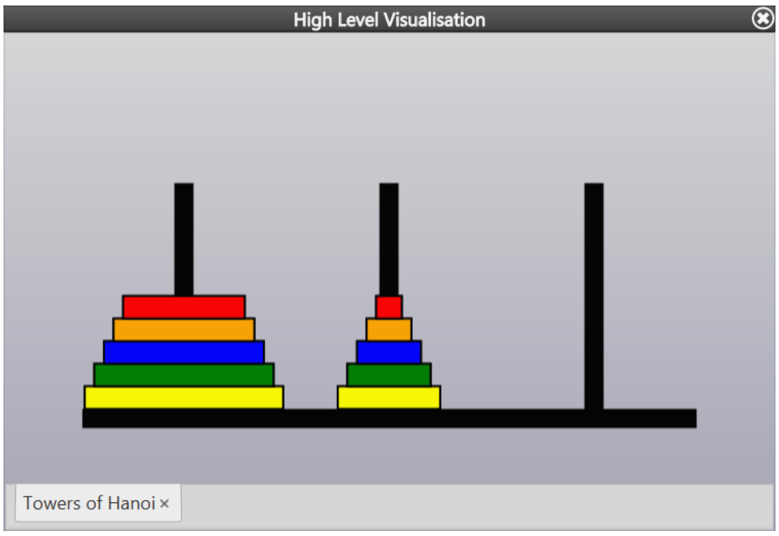


Figure 3: High Level Visualiser with Towers of Hanoi open

For more information about the different data structures Simulizer can visualise, see High Level Visualisation

2.2.4 Labels

TODO: write section

2.2.5 Program I/O

TODO: write section

2.2.6 Memory View

TODO: write section

2.2.7 Options

TODO: write section

2.2.8 Pipeline View

TODO: write section

2.2.9 Registers

TODO: write section

2.3 Layouts

Layouts determine the configuration that all the Internal Windows are in. They allow you to quickly switch between different arrangements to optimise your workflow.

2.3.1 Loading a Layout

TODO: write section

2.3.2 Saving a Layout

If none of the included layouts are up to your standards then why not make your own. Add/Remove and rearrangement the Internal Windows until it is in a configuration that you are happy with. You can then save the layout by clicking Layouts → Save Layout.

`segments/save-layout.png`

Figure 4: Saving a Layout

Enter a name for this new layout. In this case we called the layout `XXX`. Now click the save button and that new layout should show up on the Layouts drop down menu.

2.4 Themes

TODO: write section

3 Editor

TODO: write section

4 Annotations

The annotation system in Simulizer is a mechanism for tagging SIMP statements with JavaScript code which is executed *after* the statement has executed.

4.1 Syntax

The syntax is as follows:

```
add $s0 $s0 $s1   # comment @{ // annotation }@
```

The annotation begins with `@{` and ends with `}@`. These must be placed inside a comment of the assembly program (denoted using `#`).

4.2 Targets

Annotations may be placed *before* any `.data` or `.text` segments, in which case they are executed before the first instruction of the program executes. This is useful for setting up the environment for the duration of the simulation, for example getting handles to high level visualisations or setting an appropriate clock speed.

Annotations may be placed after statement, and before any label or another statement. In this case the annotation is bound to that statement.

Annotations may be placed after a label and before the next statement, in which case the annotation binds to the statement which the label binds to. This works with multiple labels. In the example below all 5 annotations are grouped and bound to the `nop` instruction

```
    syscall
label1: # @{ // annotation 1 }@
label2: # @{ // annotation 2 }@
        # @{ // annotation 3 }@
    nop # @{ // annotation 4 }@
        # @{ // annotation 5 }@
```

4.3 Grouping

Annotations bound to the same target are concatenated with newline characters placed in between, this allows more complex expressions to be written clearly such as:

```
# @{ function f(x) {   }@
# @{     if(x)         }@
# @{         return 1; }@
# @{     else          }@
# @{        return 0; }@
# @{ }                }@
```

4.4 Scope

Any variables defined at the scope of an annotation (ie not inside an inner code block or function, is accessible throughout the duration of the simulation (global). This is regardless of using `var`, ie `var x = 10; y = 20` both have the same scope.

5 Annotation API

5.1 Debug Bridge

The debug bridge (named `debug` in JS) gives the annotations access to components of the system that are useful for tracing the execution of the program and relaying information to the user for debugging purposes. Also during the development of Simulizer, the debug gives access to the runtime system which can be useful for introspection.

Methods: - `debug.log(msg)` write a message (implicitly converted to string) to the DEBUG output of program I/O - `debug.assertTrue(cond)` check that a condition holds. If it does not then a helpful message is displayed in the program I/O - `debug.alert(msg)` show a popup message (implicitly converted to string) - `debug.getCPU()` get the Java `CPU` object

5.2 Simulation Bridge

The simulation bridge (named `simulation` and `sim` in JS) gives limited access to the internals of the simulation, for example reading register values and setting the clock speed

(note: the methods for accessing registers are more easily accessed through the register global variables (see below))

Methods: - `sim.pause()` pause the simulation (*are* able to resume) - `sim.stop()` stop the simulation (not able to resume) - `sim.setSpeed(frequency)` set the simulation speed (cycle frequency) - `Word[] sim.getRegisters()` - `long sim.getRegisterS(Register)` get the current signed value of a register (identified using its enum) - `long sim.getRegisterU(Register)` get the current unsigned value of a register (identified using its enum) - `sim.setRegisterS(Register, long)` set the value (treated as signed) of a register (identified using its enum) - `sim.setRegisterU(Register, long)` set the value (treated as unsigned) of a register (identified using its enum)

5.3 Visualisation Bridge

The visualisation bridge (named `visualisation` and `vis` in JS) manages the high level visualisation window, can load high level visualisations and feed them information about the state of the simulation so that they can visualise and animate the algorithm running in the simulation.

The annotations have full public access to the methods and attributes of the `DataStructureVisualisation` that it requests, see their documentation for details about what they are capable of.

Methods: - `DataStructureModel viz.load(name)` load a visualisation by name and show the visualisation in the High Level Visualisation window (whether the window is also opened is determined by the setting: `high-level.autoopen`) - 'tower-of-hanoi' - 'list' - 'frame' - `DataStructureModel loadHidden(name)` load a visualisation by name but *do not* show it in the High Level Visualisation window (call `show` on the model later to show it) (whether the window is also opened is determined by the setting: `high-level.autoopen`). - `viz.show()` show the visualisation window (no effect if already showing) - `viz.hide()` hide the visualisation window (no effect if already hidden)

5.4 Global Variables

Each of the 32 general purpose registers are assigned as global variables (named with the dollar prefix eg `$s0`) with the following members: - `id` the enum value of the register - `long getS()` a method which corresponds to `simulation.getRegisterS(this.id)` - `long getU()` a method which corresponds to `simulation.getRegisterU(this.id)` - `setS(long)` a method which corresponds to `simulation.setRegisterS(this.id, long)` - `setU(long)` a method which corresponds to `simulation.setRegisterU(this.id, long)` - `long get()` a method which corresponds to `simulation.getRegisterS(this.id)` - `set(long)` a method which corresponds to `simulation.setRegisterS(this.id, long)`

Other variables - The variables `Register` and `reg` refer to the `Register` enum class in Java. - `convert` refers to the `DataConverter` class in java which encodes and decodes from signed/unsigned integer representations

5.5 Global Functions

To increase brevity, certain commonly used methods from the bridges are assigned to global functions which can be called without qualification:

```
// Debug Bridge
log    = debug.log
print  = debug.log
alert  = debug.alert
assert = debug.assertTrue

// Simulation Bridge
pause    = simulation.pause
stop     = simulation.stop
exit     = simulation.stop
quit     = simulation.stop
setSpeed = simulation.setSpeed

// Visualisation Bridge

// Misc
ret() // behaves like a return statement, stops execution of the current annotation
```

6 High-level Visualisations

The high-level visualisations in Simulizer allow you to view the output of your algorithms in a more visual way than tediously scanning through the contents of registers, variables, etc. There are several high-level visualisations available in Simulizer, including a list visualisation and Tower of Hanoi. There are features for the list visualisation that allow you to swap and highlight elements, and place markers above them. This could be useful for algorithms such as binary search, where you may want to highlight the current element being inspected, and place `L` and `R` markers over the left and right end points of the current section of the list.

As well as making your programs easier to view and debug, the high-level visualisations also provide a satisfying result when an algorithm is implemented correctly, which will hopefully make writing assembly more fun!

6.1 Tower of Hanoi

TODO: take picture

Tower of Hanoi is a simple game where the player's goal is to move all n discs from the initial peg to another peg by moving one disc at a time without ever placing a larger disc on top of a smaller disc.

To use the Tower of Hanoi visualisation, add the comment

```
# @{ var h = vis.load('tower-of-hanoi') }@
```

to the start of your program. To set the initial number of discs, add `#@{h.setNumDisks(4)}@`, where 4 can be replaced with any positive integer you wish. To show the visualiser window, call `#@{vis.show()}@`. Finally, to indicate that you want to move a disc from peg i to peg j, write `#@{ h.move(i,j)}@`.

6.2 List Visualisation

TODO: take picture

To use the list visualisation, add the comment

```
# @{ var l = vis.load('list') }@
```

to the start of your program. To set the list, write

```
# @{ l.setList(simulation.readUnsignedWordsFromMem(start, end)) }@
```

To show the visualiser, write `#@{vis.show()}@`. To swap elements with indices i and j, write `#@{l.swap(i,j)}@`. To emphasise the element as position i write `#@{l.emph(i)}@`. To add a marker over the element at position i write

```
# @{l.setMarker(i,"<label-text>")}@
```

7 Assembler

TODO: write section

8 Low-level Visualisations

8.1 CPU Visualisation

TODO: take picture

8.2 Pipeline Visualisation

TODO: take picture The pipeline visualisation window allows you to view the contents of the pipeline, as well as the waiting and completed instructions, as the CPU is running. The middle third of the window shows the fetch, decode, and execute portions of the pipeline at each state, including hazards where appropriate (represented as red circles).

Firstly, to use the pipeline view, the CPU must be running and in pipelined mode (to turn on pipelining, go to the `Simulation` menu and make sure `Toggle CPU Pipelining` is selected). Once running, the window will start to fill up from left to right with instructions being processed.

Let's look at the components in the control bar at the bottom of the window:

- `Follow` checkbox: allows you to toggle whether or not you want to snap to the most recent stage of the pipeline.
- Left/Right buttons: allows you to move backwards/forwards cycles in the pipeline (this can also be achieved by using the left and right keys on your keyboard).
- `Goto:` field: allows you to jump to a specified cycle (indicated at the bottom of each stage of the pipeline) by entering the cycle number, e.g. 67, and then pressing enter. If you enter a number greater than the number of cycles, then it will jump to the last cycle.

You can view information about instructions in the window by hovering over them with your mouse. Summary information about the instruction will appear at the bottom of a screen. If you hover a hazard, the window will tell you what type of hazard was encountered at this time.

You can highlight all occurrences of a particular instruction by clicking on it. This can be useful to track the state of the instruction through the pipeline and to spot past/future occurrences of the instruction.

A maximum of 10,000 pipeline stages can currently be displayed (this is to mitigate problems with infinite loops etc.)

8.3 Memory View

TODO: take picture