Contents

Annotations

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

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 `#`).

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 4 annotations are grouped and bound to the `nop` instruction

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

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; }@
# @{ }                 }@
```

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.

Annotation API

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:

- `log(msg)` write a message (implicitly converted to string) to the program I/O
- `alert(msg)` show a popup message (implicitly converted to string)
- `getCPU()` get the Java `CPU` object

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

Methods:

- `stop()` stop the simulation (not able to resume)
- `setClockSpeed(tickMillis)` set the simulation speed
- `Word[] getRegisters()`
- `Word getRegister(Register)` get the current value of a register (identified using its enum)

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:

- `DataStructureVisualiser load(name)` load a visualisation by a name
  - 'tower-of-hanoi'
  - 'list'
- `DataStructureVisualiser load(name, showNow)` load a visualisation and optionally hide it for now
- `show()` show the visualisation window if it was hidden by `load(name, false)`

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
- `get()` a method which corresponds to `simulation.getRegister(this.id)`

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

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

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

// Visualisation Bridge
```