EmuStudio-v0

Author:	Ben Vanik (ben at vanik dot net) : http://www.noxa.org
Release date:	2004-02-07
Version:	0.1

Table of contents

[top] - About

Foreword:	The goal of this project was the create a SPIM-like simulator for both school and experience. It is not intended to be a SPIM replacement, containing significantly less features and support (no floating points, for example), however it will run most basic SPIM code that doesn't rely on user input. Originally I was going to build in a compiler as well, and at this time it's about half done, but since it wasn't required by the assignment I decided to wait on it.
Motivation:	Required for "Computer Organization" (CDA3103 @ UCF), as well as a desire to learn more about emulation. The assignment required 5% of what EmuStudio-v0 currently is, but I learned a lot more this way.
Future plans:	Rewrite of the system from scratch using C#. This will allow for better design, greater flexibility, and will motivate me to code more architectures (x86, etc).

[top] - Features

TODO:	Write something here

[top] - Usage

Concepts:

- **Fragment**: A general purpose text/data environment. Contains memory (code and data), runtime and debug information.
- **Parser**: Subsystem that will take an input file in a given format and build a fragment from it.
- **Compiler**: Subsystem that will take an input file, compile it, and build a fragment from it.
- **Processor**: Architecture-specific processor; interprets and executes code.
- **Execution engine**: Logical code structure that uses a Processor on a Fragment to run an application.
- **Surface**: Display output for various systems. See Appendix: Surface types for an overview.

- **Console**: Interactive command line interface for user control.

Flow:

1. Check fragment generation mode
   1. Compile -> store fragment
   2. Parse -> store fragment
2. Create processor/support
3. Check run mode
   1. Auto
      1. Execute code
      2. Dump all
   2. Interactive
      1. Get and act on user command
      2. Repeat until exit
4. Cleanup

Command line:

```
Usage: Entry [-arch architecture] [-auto [dump]] {-compile | (-parse format)} filename
```

arch (optional)	
Description:	Specify the architecture to use. Pass nothing to see a list of available architectures.
Example:	`-arch MIPSR3000`
auto (optional)	
Description:	Enable automated execution. If this is given, instead of dropping to the interactive console the 'run' command will be issued right after parsing/compilation and the simulator will exit after it's completion. If 'dump' is specified then after execution a 'dump all' will be performed. Note: 'directconsole' is set to 1 when 'dump' is not specified, and 0 when it is.
Example:	`-auto dump`
compile (exclusive)	
Description:	Set mode to compile - 'filename' will be assumed to be source and will be compiled into a fragment.
Example:	`-compile`
parse (exclusive)	
Description:	Set mode to parse - 'filename' will be assumed to be of valid parse format for the 'format' specified. Pass nothing to see a list of available formats, or see Appendix: File formats.
Example:	`-parse simple`
filename	
Description:	The input file for the compile/parse.
Example:	`test.o`

Examples:

```
Entry -auto dump -parse simple test.o
```

Parse (using the simple format) 'test.o' and automatically run it, dumping all info when done.

```
Entry -compile source.asm
```

Compile 'source.asm' and enter the interactive console.

Interactive console:

Supported commands	
exit	
Description:	Exit execution engine.

Usage:	`exit`
Example:	`exit`
help	
Description:	Display command list or help on a particular command.
Usage:	`help [command]`
Example:	`help run`
reset	
Description:	Reset the processor context. Depending on the architecture this may or may not reset memory - do not assume it is clean.
Usage:	`reset`
Example:	`reset`
run	
Description:	Execute until breakpoint, error, or done - display extra information if 'v' set (for 'verbose'). This will pick up from the current position; to start from the beginning, perform a `reset` first.
Usage:	`run ['v']`
Example:	`run v`
step	
Description:	Execute one instruction - display info if 'v' set (for 'verbose')
Usage:	`step ['v']`
Example:	`step v`
peek	
Description:	Display instruction that will be executed next.
Usage:	`peek`
Example:	`peek`
dump	
Description:	View the given surface or a list of surfaces if none specified. See available surfaces and parameters in Appendix: Surface types.
Usage:	`dump [class [params]]`
Example:	`dump registers`
vars	
Description:	List all settings and values. See Appendix: Variables for information on a specific variable.
Usage:	`vars`
Example:	`vars`
set	
Description:	Set the value of the given setting. See Appendix: Variables for information on a specific variable.
Usage:	`set [var value]`
Example:	`set directconsole 1`
.	
Description:	Repeat last command.

Usage:	.
Example:	.

Examples:

```
-> set directconsole 0
-> run
-> dump console
```

Disable direct console printing, execute the code, and dump the results on the console surface.

```
-> peek
-> step v
-> .
-> .
-> run
-> reset
-> peek
```

View the next instruction waiting to execute, step into it (and print debug info), repeat the step (x2), finish running the program, reset the processor, view the next instruction waiting to execute (should be the same as the first peek).

[top] - Known issues

arch.Segment	This class should really throw exceptions (ie, java.lang.IndexOutOfBoundsException); as of now it just fails gracefully.
arch.Compiler	The entire compile system is currently not implemented; this includes any support code like in Entry. It's about 40% completed - a little more work and it will be done.
[any]	There is no way for a Processor to get input from the user. It would be useful to add a callback system to allow this.

[top] - Developer notes

Tools used:	Java runtime:	Sun Java JRE SE 1.4.2_03-b02
	Java IDE:	IBM Eclipse M6 w/ Metrics 1.3.4
	Text editor:	Microsoft Visual Studio.NET 2003
	Hex editor:	010 Editor
	Environment:	Cygwin 1.5.7-1 on Microsoft Windows XP SP2
Time to complete:	4 straight days	
Classes:	96	
Lines of code:	3227	

[top] - Credits

Design, coding:	Ben Vanik (ben at vanik dot net) : http://www.noxa.org
Int**ToHex:	Dr. Ricardo Lent : rlent@cs.ucf.edu (?)
MIPS	

reference:	Computer Organization and Design, 2nd ed.; Hennessy and Patterson
Background noise:	Sakamoto Maaya (my hero!) - some mp3's/info here

[top] - Appendix: File formats

Binary: **Not yet implemented**

Debug: **Not yet implemented**

Simple:

```
address data [extra]
address data [extra]
...
```

Address: Hexadecimal 32-bit memory address to load data at (the first address is assumed to be the text entry point)
Data: Hexadecimal 32-bit word data
Extra: All text until the end of the line will be attached to the address

This format allows for sparse memory segments. This is useful when defining clear text and data regions.
Example:

```
0x00001000 0x00000000 // text
0x00001004 0x00000000 // text
0x0000F0F0 0xFFFFFFFF // data
0x0000F0F4 0xFFFFFFFF // data
.....
```

In this example, the range from 0x00001008 to 0x0000F0F0 will not be explicitly allocated.

Note this format is not strict - one may omit the '0x' prefix on the hex numbers, as well as provide wrapping braces ('[', ']') around the address (this makes copying and pasting from SPIM painless).

Serialized: **Not yet implemented**

[top] - Appendix: Surface types

All: **Description**: Display all surfaces
Parameter: Not used

Registers: **Description**: Current processor context - fields dependent on architecture
Parameter: Not used

```
     PC : 0x0040013C      HI : 0x00000001      LO :
0x00000005
00 ( $0): 0x00000000 11 ($t3): 0x00000000 22 ($s6):
0x00000F0F
01 ($at): 0x000F0000 12 ($t4): 0x00000000 23 ($s7):
0x0000001A
...
```

Status: **Description**: Runtime statistics - fields dependent on architecture
Parameter: Not used

```
Instructions executed: 219
JIT hits: 158
JIT faults: 61
Memory allocated: 1966080b
Memory accesses: 111
.....
```

Code: **Description**: Code trail - instructions in the order they were executed - format dependent

on architecture
Parameter: Not used

```
Address    Data       JIT  Src  Code
0x00400000 0x3C011001 [JF] in:1 [disabled/na]
0x00400004 0x34280058 [JF] in:2 [disabled/na]
0x00400008 0x8D080000 [JF] in:3 [disabled/na]
.....
```

Memory: **Description**: Memory map (if no parameter passed) or memory locale (if valid address passed)
Parameter: Memory address in hex format to view around - i.e., say you give address 0x10040D04, the system may print memory from the range 0x10000000 to 0x100F0000
Note: Large ranges of empty memory will be collapsed into a single line

```
[memory locales currently allocated]
```

-or-

```
0x0FFF0000-0x10010000 = 0x00000000
0x10010000 0x73696854 0x20736920 0x65742061
0x6F0A7473
0x10010010 0x72702066 0x5F746E69 0x69727473
0x000A676E
.....
```

Debug: **Description**: Architecture-specific debug surface
Parameter: Dependent in implementation

Console: **Description**: Output from the processor
Parameter: Not used

[top] - Appendix: Variables

directconsole: boolean: When enabled (1), all console output from the processor will go directly to the user console and not to the console surface. This is enabled by default to make quick execution easier, but is disabled during automated dumps because it makes things easier to read.

emitsource: boolean: When enabled (1), any associated information/source will be attached to a line and displayed on the code surface. Since this can get messy, it is disabled by default.

[top] - Appendix: MIPSR3000(A)

Unsupported: copX, syscall input, lwl, lwr, swl, swr, lwcX, swcX - probably more

Known issues:

- lh/sh don't currently twiddle
- $gp, $fp don't get set right
- Use of stacks is currently untested (probably doesn't work)
- Segment does not check chunk boundaries on multi-byte read/writes (syscall print_string may be bad too)
- Signed/unsigned stuff has been ignored! There are probably a ton of places this will mess things up! Sign extension may be flakey as well

Future plans: Implement everything, test everything - finish the compiler!