



COMP3901 - SPECIAL PROJECT A
SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

MIPSY

An education focused MIPS simulator

Author
Zac KOLOGLU

Supervisor
Andrew TAYLOR

November 26, 2020

Contents

1	Introduction	1
1.1	About mipsy	1
1.2	Problem statement	1
1.3	Accessing mipsy	2
2	Current Work	3
2.1	SPIM	3
2.2	MARS	10
2.3	EduMIPS64 / WepSIM / WebMIPS	12
3	Mipsy: Goals	14
3.1	For students	14
3.2	For educators	15
4	Mipsy: Evaluation	16
4.1	For students	17
4.2	For educators	35
5	Mipsy: Implementation	39
5.1	Rust backend	39
5.2	Parser	40
5.3	Compiler	41
5.4	Runtime	42
5.5	CLI	43
5.6	Challenges faced	43
6	Future Work	45
6.1	Web browser - WASM	45
6.2	Language Server	45
6.3	Completeness	46
7	End Matter	47
7.1	Acknowledgments	47
7.2	Appendixes	47
7.3	References	47

Chapter 1

Introduction

1.1 About mipsy

Mipsy is an education-first MIPS I R2000 simulator, focused on teaching students how to write effective assembly code, rather than trying to teach about the hardware behind a MIPS CPU. It aims to provide specific, targeted error messages at both compile-time and runtime wherever possible, that competing simulators struggle to match in clarity and correctness. It allows additional runtime sanitation, similar to modern C runtimes such as `llvm-msan`, including detecting reads of uninitialised registers and memory. It includes an interactive-mode with `readline` and terminal-colour support, and a powerful time-travel capable debugger.

Mipsy was written primarily for UNSW's [COMP1521](#) - an introductory systems-programming course focusing on low-level C, and MIPS assembly. It was born out of numerous, repeated frustrations from the vast majority of both students and educators of the course regarding the current simulator of choice - SPIM.

1.2 Problem statement

Many higher-educational institutions mandate learning a form of assembly language as part of their typical Computer science / Software engineering degrees. Of these, many opt to teach the MIPS architecture, due to its simplicity in design, small and elegant instruction set, and past prominence. However, as most modern computers are x86 or ARM based, it is not practical for students to run their MIPS assembly code natively. As a result, educators must choose a MIPS simulator for their students to use, of which options are few.

Ideally, this simulator would allow students to efficiently compile, execute, test, and debug their code. Compilation errors would be correct, targeted, and offer useful advice to students. Issues at runtime would be caught early, explained well, and be simple to debug. The simulator would be very simple to use, but extremely powerful for intermediate-advanced users, and forgiving of mistakes in usage. The simulator would be extensible and modular, to allow extra custom behaviour to be programmed in, or to use parts of the simulator for related projects (eg. using the simulator's parser to write a syntax highlighter, language server, etc.).

Unfortunately, current MIPS simulators fail in many of these regards, leading to a poor student experience. Consequently, students spend less time learning how to write quality assembly code, and instead, frustratingly learn how to wrestle with their institution's particular simulator of choice, and spend hours at a time debugging what often ends to be a simple issue.

Mipsy aims to succeed in all these aspects, as a MIPS assembly-education-first simulator. Although it doesn't aim to promise a rigorous implementation of the MIPS specification (as many others do), or 100% instruction-set coverage, it strives to make the experiences of students and educators alike as pleasant as possible. It has been written with a keen focus on modularity and extensibility, splitting the project into a separate parser, library, and binary, and making course-specific extensions such as custom pseudo-instructions, or a modified trap-file very simple to implement.

1.3 Accessing mipsy

Mipsy can be accessed at <https://github.com/insou22/mipsy>, and built / deployed by following the instructions in the README. Mipsy is written in Rust, so installing the Rust toolchain (<https://rustup.rs>) is necessary to compile from source.

Chapter 2

Current Work

2.1 SPIM

<http://spimsimulator.sourceforge.net>

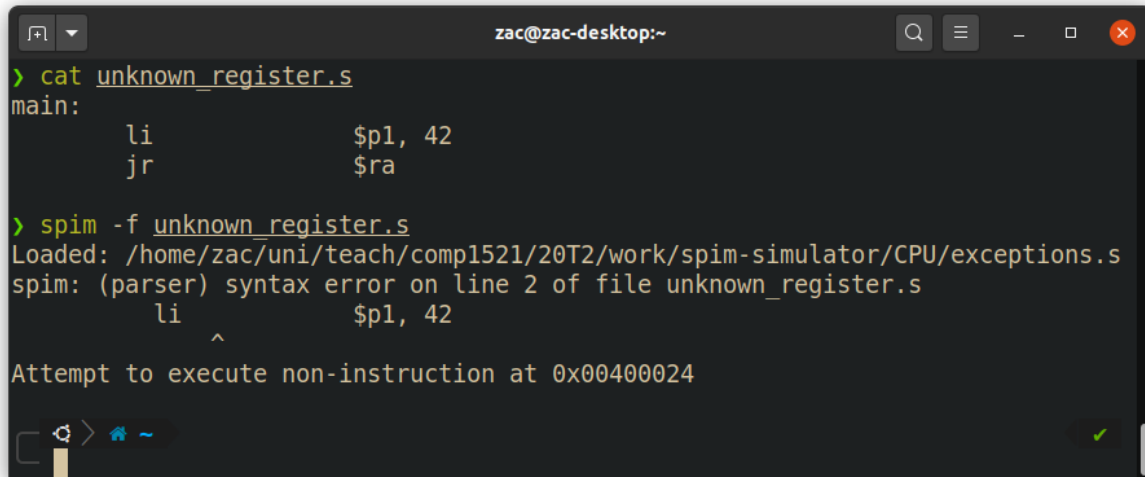
SPIM is easily the most well-known and widely used MIPS simulator currently. It describes itself as "a self-contained simulator that runs MIPS32 programs. It reads and executes assembly language programs written for this processor. Spim also provides a simple debugger and minimal set of operating system services". It was authored in 1990, and although it doesn't receive new features anymore, it is still maintained with bug-fixes as recent as July 2020.

SPIM offers two main mechanisms of interacting with it - a command-line application (`spim`), that can directly execute a file, or run as an interactive shell with debugging capabilities, and a graphical interface built using the Qt framework (`QtSpim`), with similar debugging capabilities to the command-line shell. Both these programs are open source, and provide pre-compiled executables for Windows, MacOS, and Debian distributions of Linux.

SPIM is likely to be the most complete and accurate simulation of the MIPS32 assembler-extended instruction set out of the options available, and is a well-proven solution. Although SPIM tends to work exceptionally well when given a file that compiles correctly, and runs without any poor behaviour, it struggles in other cases. As SPIM is a simulator-first project, optimising its behaviour in error communication and recovery has never been a high priority. Unfortunately, in terms of education, this tends to be one of the most important factors in a given student's experience, as it is decidedly unlikely for a student to write a compiling file on their first attempt, or to program a bug-free runtime.

A few notable examples of SPIM's error reporting follow:

Since the specific register-names are built directly into SPIM's parser, when a student tries to use an incorrectly named register, it fails to provide a useful error message, and often even points to the wrong position of the error with the caret.

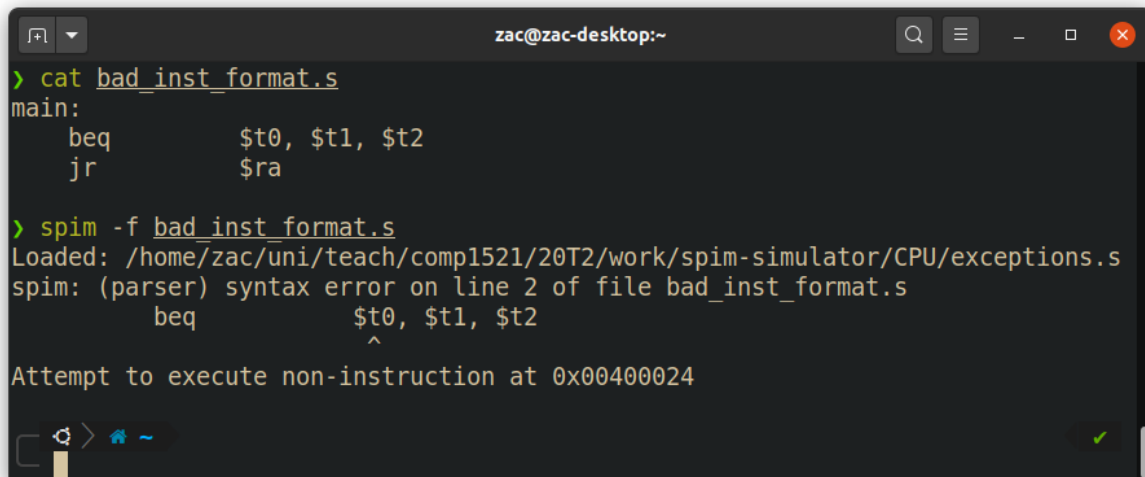
A terminal window titled 'zac@zac-desktop:~' showing the execution of SPIM. The user enters 'cat unknown_register.s' and 'spim -f unknown_register.s'. The assembly code in 'unknown_register.s' is: 'main: li \$p1, 42; jr \$ra'. The SPIM output shows a 'syntax error on line 2 of file unknown_register.s' with a caret pointing to the space before '\$p1'. Below the error, it says 'Attempt to execute non-instruction at 0x00400024'.

```
> cat unknown_register.s
main:
    li    $p1, 42
    jr    $ra

> spim -f unknown_register.s
Loaded: /home/zac/uni/teach/comp1521/20T2/work/spim-simulator/CPU/exceptions.s
spim: (parser) syntax error on line 2 of file unknown_register.s
    li    $p1, 42
           ^
Attempt to execute non-instruction at 0x00400024
```

Figure 2.1: SPIM - Providing an incorrectly-named register (\$p1) to an instruction that is otherwise written correctly

When using an instruction that exists, but with an incorrect set of arguments, SPIM fails to give a specific error message, nor help the student understand what format the arguments should be in.

A terminal window titled 'zac@zac-desktop:~' showing the execution of SPIM. The user enters 'cat bad_inst_format.s' and 'spim -f bad_inst_format.s'. The assembly code in 'bad_inst_format.s' is: 'main: beq \$t0, \$t1, \$t2; jr \$ra'. The SPIM output shows a 'syntax error on line 2 of file bad_inst_format.s' with a caret pointing to the space before '\$t0'. Below the error, it says 'Attempt to execute non-instruction at 0x00400024'.

```
> cat bad_inst_format.s
main:
    beq    $t0, $t1, $t2
    jr    $ra

> spim -f bad_inst_format.s
Loaded: /home/zac/uni/teach/comp1521/20T2/work/spim-simulator/CPU/exceptions.s
spim: (parser) syntax error on line 2 of file bad_inst_format.s
    beq    $t0, $t1, $t2
           ^
Attempt to execute non-instruction at 0x00400024
```

Figure 2.2: SPIM - Formatting the arguments of an instruction incorrectly (beq's third argument should not be a register)

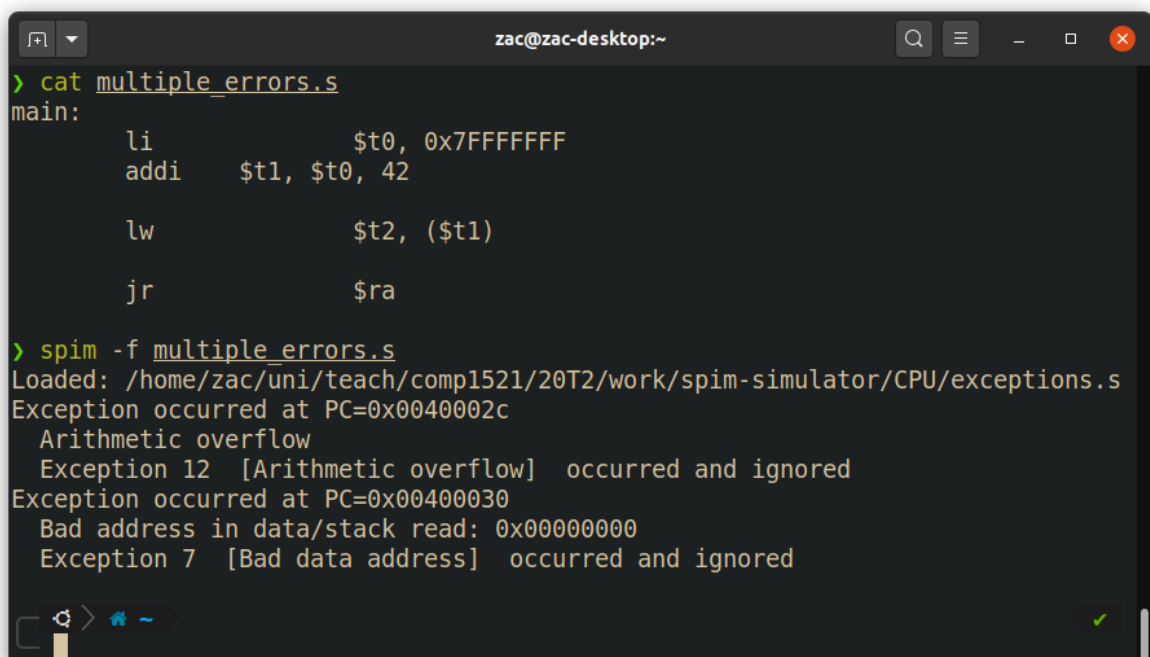
SPIM is often even more undecipherable when it comes to runtime errors, in which it won't even attempt to tell you which line the error occurred on (rather opting for the current PC value, which is quite confusing for students), how the exception occurred, what the values that caused said error were, etc.



```
zac@zac-desktop:~  
> cat overflow_add.s  
main:  
    li    $t0, 0x7FFFFFFF  
    li    $t1, 42  
  
    add   $t2, $t0, $t1  
  
    jr    $ra  
  
> spim -f overflow_add.s  
Loaded: /home/zac/uni/teach/comp1521/20T2/work/spim-simulator/CPU/exceptions.s  
Exception occurred at PC=0x00400030  
    Arithmetic overflow  
Exception 12 [Arithmetic overflow] occurred and ignored
```

Figure 2.3: SPIM - Arithmetic overflow

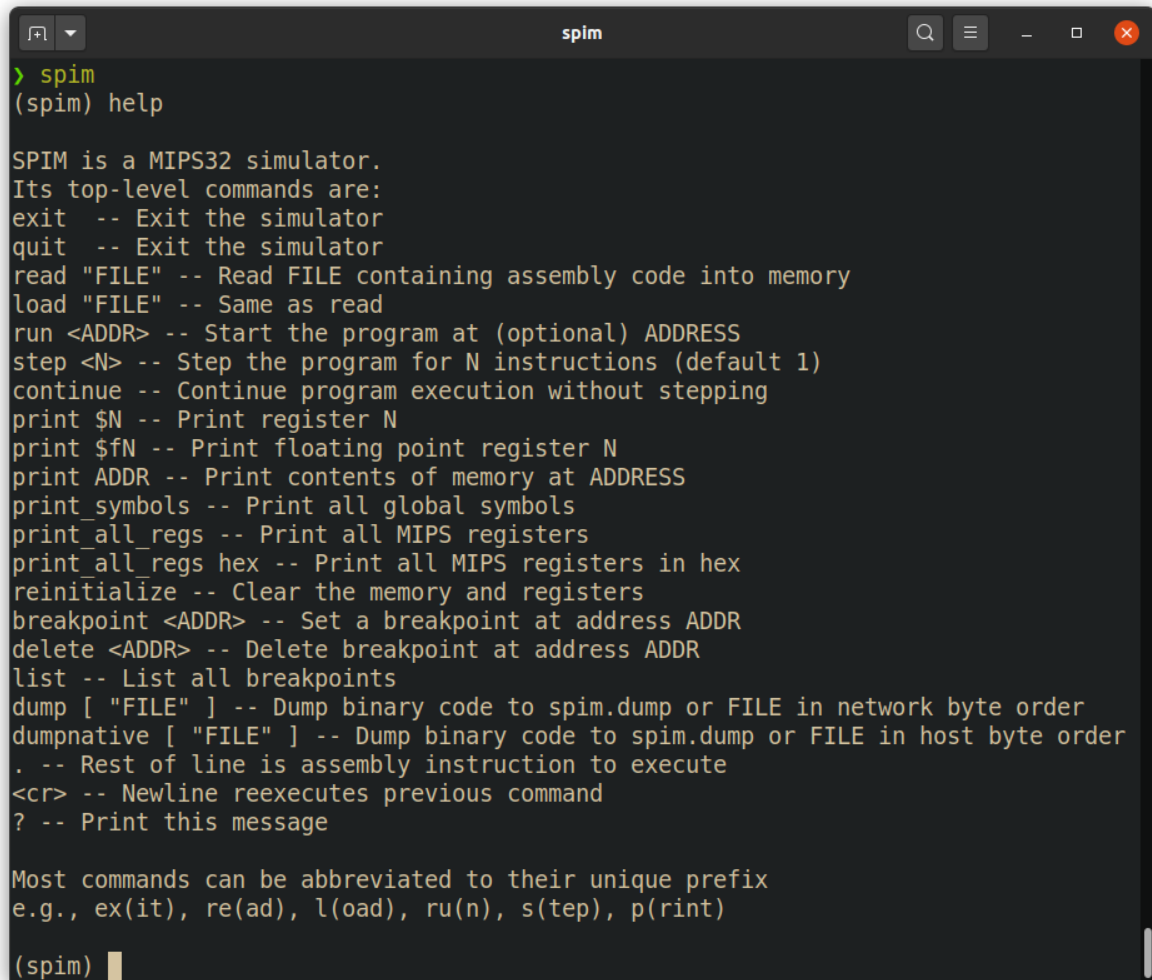
SPIM will not halt the program on most runtime errors, which can lead to a plethora of simultaneous error messages, especially when occurring in loops. As students often only read the most recent error, this often causes substantial confusion.



```
zac@zac-desktop:~  
> cat multiple_errors.s  
main:  
    li    $t0, 0x7FFFFFFF  
    addi   $t1, $t0, 42  
  
    lw    $t2, ($t1)  
  
    jr    $ra  
  
> spim -f multiple_errors.s  
Loaded: /home/zac/uni/teach/comp1521/20T2/work/spim-simulator/CPU/exceptions.s  
Exception occurred at PC=0x0040002c  
    Arithmetic overflow  
Exception 12 [Arithmetic overflow] occurred and ignored  
Exception occurred at PC=0x00400030  
    Bad address in data/stack read: 0x00000000  
Exception 7 [Bad data address] occurred and ignored
```

Figure 2.4: SPIM - Multiple errors in the same program

SPIM's interactive debugger unfortunately also leaves a lot to be desired. It has a fairly comprehensive suite of commands available to the user, which tend to work well, but the interface itself has many issues students often run into.

A screenshot of a terminal window titled "spim". The prompt is "(spim) help". The output lists various commands and their functions: exit, quit, read, load, run, step, continue, print, print \$N, print \$fN, print ADDR, print symbols, print_all_regs, print_all_regs hex, reinitialize, breakpoint, delete, list, dump, dumpnative, ., <cr>, and ?. It also mentions that commands can be abbreviated by their unique prefix.

```
> spim
(spim) help

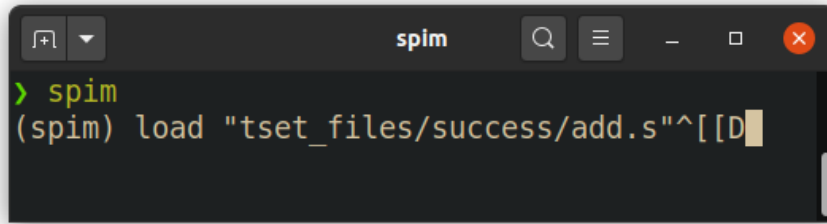
SPIM is a MIPS32 simulator.
Its top-level commands are:
exit  -- Exit the simulator
quit  -- Exit the simulator
read "FILE" -- Read FILE containing assembly code into memory
load "FILE" -- Same as read
run <ADDR> -- Start the program at (optional) ADDRESS
step <N> -- Step the program for N instructions (default 1)
continue -- Continue program execution without stepping
print $N -- Print register N
print $fN -- Print floating point register N
print ADDR -- Print contents of memory at ADDRESS
print symbols -- Print all global symbols
print_all_regs -- Print all MIPS registers
print_all_regs hex -- Print all MIPS registers in hex
reinitialize -- Clear the memory and registers
breakpoint <ADDR> -- Set a breakpoint at address ADDR
delete <ADDR> -- Delete breakpoint at address ADDR
list -- List all breakpoints
dump [ "FILE" ] -- Dump binary code to spim.dump or FILE in network byte order
dumpnative [ "FILE" ] -- Dump binary code to spim.dump or FILE in host byte order
. -- Rest of line is assembly instruction to execute
<cr> -- Newline reexecutes previous command
? -- Print this message

Most commands can be abbreviated to their unique prefix
e.g., ex(it), re(ad), l(oad), ru(n), s(tep), p(rint)

(spim) █
```

Figure 2.5: SPIM - Interactive commands

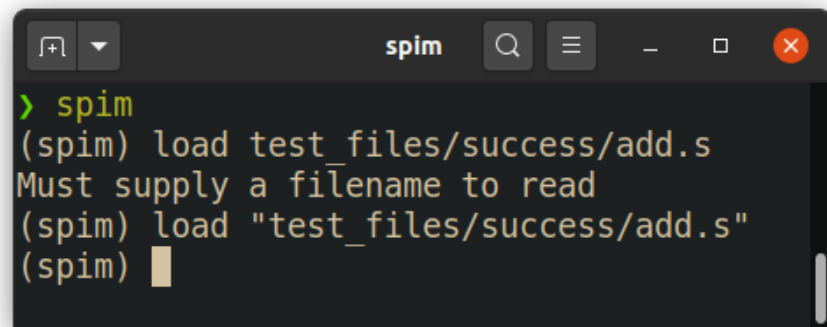
SPIM has very poor readline support, only correctly supporting normal text-input, and the backspace key. Attempting to use any of, for example: arrow keys (left or right to navigate the current line, up or down to browse command history), emacs-style text manipulation (C-a to move the cursor to the beginning of the line, C-e similarly for the end of the line, C-w to delete a word, etc.), reverse searching (C-r, C-s in most modern shells), will result in SPIM simply inserting the control sequence as ASCII input - which must then be manually backspaced, or otherwise completely break SPIM's command line parser, requiring the student to start over.

A screenshot of a SPIM terminal window. The window has a title bar with the text "spim" and standard window controls (minimize, maximize, close). The terminal content shows a prompt "> spim" followed by a command "(spim) load \"tset_files/success/add.s\"^[[D". The cursor is at the end of the command, and the text is partially obscured by a yellow highlight.

```
> spim
(spim) load "tset_files/success/add.s"^[[D
```

Figure 2.6: SPIM - Using an arrow key to attempt to fix a mistake

SPIM also liberally makes use of its yacc parser to parse the interactive commands, which causes copious amounts of absolutely bizarre error-cases, such as forgetting to wrap a file's name in quotation marks, and the blatantly useless message it responds with.

A screenshot of a SPIM terminal window showing a sequence of commands and error messages. The window has a title bar with the text "spim" and standard window controls. The terminal content shows a prompt "> spim" followed by three commands: "(spim) load test_files/success/add.s", "(spim) load \"test_files/success/add.s\"", and "(spim) ". The first command results in the error message "Must supply a filename to read". The second command is followed by a prompt "(spim) " with a yellow highlight.

```
> spim
(spim) load test_files/success/add.s
Must supply a filename to read
(spim) load "test_files/success/add.s"
(spim) 
```

Figure 2.7: SPIM - Strict command parsing

SPIM unfortunately provides no way of gathering context (eg. current and surrounding instructions) on your current position in the program, so the primary way to gather information and debug using SPIM is simply to continually step forwards. This packs the student's terminal with information, making it difficult to pick out what is important to them, with no way to highlight the different sections with colour. Furthermore, it becomes extremely easy to miss when important events, such as system calls, are executed - as seen in the following figure.

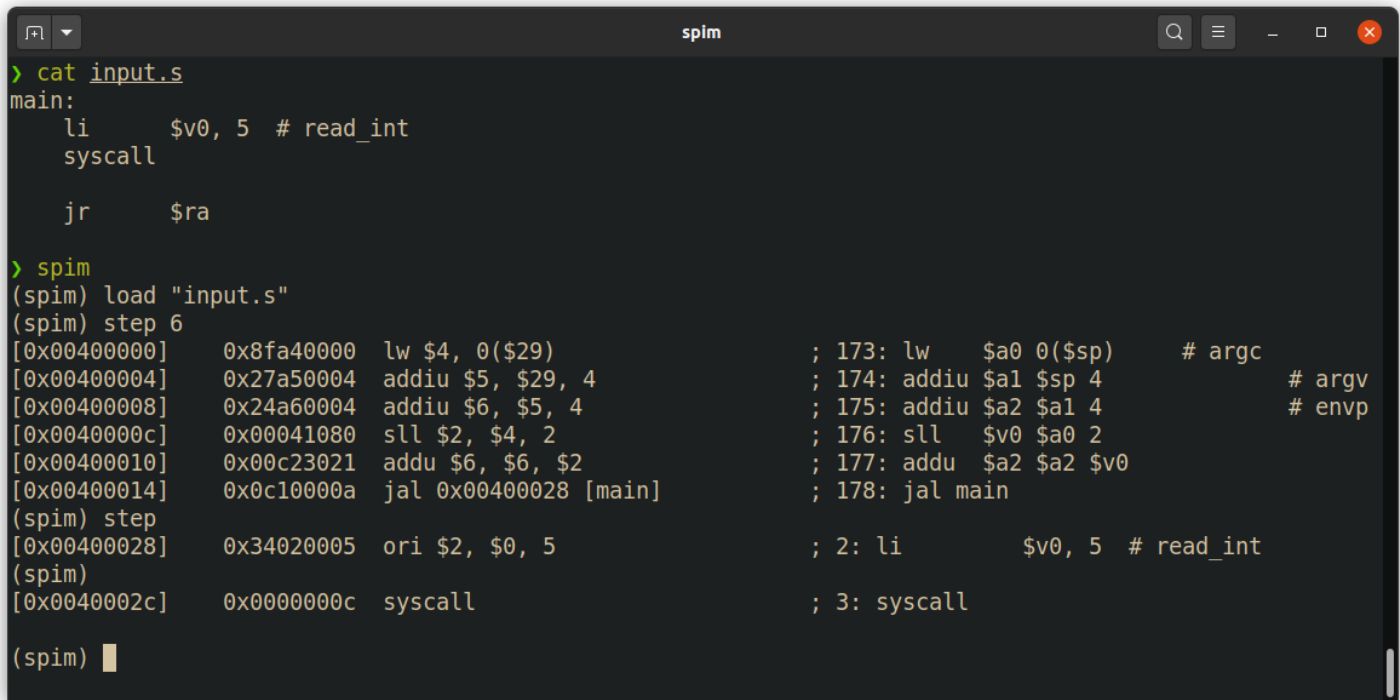
```

> spim
(spim) load "test_files/success/add.s"
(spim) step
[0x00400000] 0x8fa40000 lw $4, 0($29) ; 173: lw $a0 0($sp) # argc
(spim)
[0x00400004] 0x27a50004 addiu $5, $29, 4 ; 174: addiu $a1 $sp 4 # argv
(spim)
[0x00400008] 0x24a60004 addiu $6, $5, 4 ; 175: addiu $a2 $a1 4 # envp
(spim)
[0x0040000c] 0x00041080 sll $2, $4, 2 ; 176: sll $v0 $a0 2
(spim)
[0x00400010] 0x00c23021 addu $6, $6, $2 ; 177: addu $a2 $a2 $v0
(spim)
[0x00400014] 0x0c10000a jal 0x00400028 [main] ; 178: jal main
(spim)
[0x00400028] 0x34080011 ori $8, $0, 17 ; 4: li $t0, 17 # x = 17;
(spim)
[0x0040002c] 0x34090019 ori $9, $0, 25 ; 6: li $t1, 25 # y = 25;
(spim)
[0x00400030] 0x01285020 add $10, $9, $8 ; 8: add $t2, $t1, $t0 # z = x + y
(spim)
[0x00400034] 0x000a2021 addu $4, $0, $10 ; 10: move $a0, $t2 # printf("%d", a0);
(spim)
[0x00400038] 0x34020001 ori $2, $0, 1 ; 11: li $v0, 1
(spim)
[0x0040003c] 0x0000000c syscall ; 12: syscall
42(spim)
[0x00400040] 0x3404000a ori $4, $0, 10 ; 14: li $a0, '\n' # printf("%c", '\n');
(spim)
[0x00400044] 0x3402000b ori $2, $0, 11 ; 15: li $v0, 11
(spim)
[0x00400048] 0x0000000c syscall ; 16: syscall
(spim)
[0x0040004c] 0x34020000 ori $2, $0, 0 ; 18: li $v0, 0 # return 0
(spim)

```

Figure 2.8: SPIM - Stepping through a simple add.s program

This especially is a problem in regards to system calls requiring input from the user. While a student is stepping through their program, pressing enter simply one too many times after a read system call will revoke the student's ability to provide input to the program, and as SPIM has no ability to step backwards, it often requires the student to restart their debugging session from the beginning. In a program that requests input further from the entry point, or also requests multiple pieces of input, the chance for this to accidentally happen even once greatly increases.



```

> cat input.s
main:
    li    $v0, 5 # read_int
    syscall

    jr    $ra

> spim
(spim) load "input.s"
(spim) step 6
[0x00400000] 0x8fa40000 lw $4, 0($29) ; 173: lw $a0 0($sp) # argc
[0x00400004] 0x27a50004 addiu $5, $29, 4 ; 174: addiu $a1 $sp 4 # argv
[0x00400008] 0x24a60004 addiu $6, $5, 4 ; 175: addiu $a2 $a1 4 # envp
[0x0040000c] 0x00041080 sll $2, $4, 2 ; 176: sll $v0 $a0 2
[0x00400010] 0x00c23021 addu $6, $6, $2 ; 177: addu $a2 $a2 $v0
[0x00400014] 0x0c10000a jal 0x00400028 [main] ; 178: jal main
(spim) step
[0x00400028] 0x34020005 ori $2, $0, 5 ; 2: li $v0, 5 # read_int
(spim)
[0x0040002c] 0x0000000c syscall ; 3: syscall
(spim)

```

Figure 2.9: SPIM - Accidentally pressing enter once too far while stepping

Many of the issues raised with SPIM's interactive debugger may be avoided by using the graphical variant, QtSpim, instead. However, this does not mean the issues are of little importance. UNSW teaches C in many of its core courses (eg. [COMP1511](#), [COMP1521](#), [COMP2521](#)), assumes knowledge of C in many computing electives (eg. [COMP2121](#), [COMP3231](#), [COMP4128](#), [COMP6447](#), etc.), and expects students to be very familiar inside a Linux terminal within their first year. Therefore, in the long run it is most beneficial for students to gain experience using command-line tools, rather than graphical applications. It is also especially useful for students to have experience in a simple MIPS debugger before having to dive into much more complex tools later in their degree such as GDB or LLDB.

Furthermore, QtSpim is arguably worse at dealing with errors, and at best is equally as unhelpful as SPIM. As mipsy doesn't currently contain a graphical equivalent (see - Further Work: Web browser - WASM), it will not be compared to QtSpim yet - however, as QtSpim is closely based on the same codebase as SPIM, there is a significant overlap in their educational issues.

2.2 MARS

MARS is the first MIPS simulator specifically designed for education around writing assembly code, developed by Missouri State University in 2002-2004, released in 2005, and last updated in 2014. It describes itself as a "lightweight interactive development environment (IDE) for programming in MIPS assembly language, intended for educational-level use with Patterson and Hennessy's Computer Organization and Design".

MARS exclusively works through the graphical IDE it offers, and as such, it will not be extensively compared with mipsy, for the same reasons argued as for QtSpim - it is more preferable for students to gain experience using the terminal and interactive-debuggers, rather than immediately stepping into using large graphical applications, and mipsy's graphical equivalent is yet to be completed.

MARS's graphical IDE does a reasonably good job at helping students understand errors, but unfortunately can be quite cluttered and confusing at runtime for beginning students. Following is an example of a program that misspells a label, where MARS correctly identifies where the issue originates, but unfortunately doesn't offer any suggestions (eg. the very similarly spelled `my_function`).

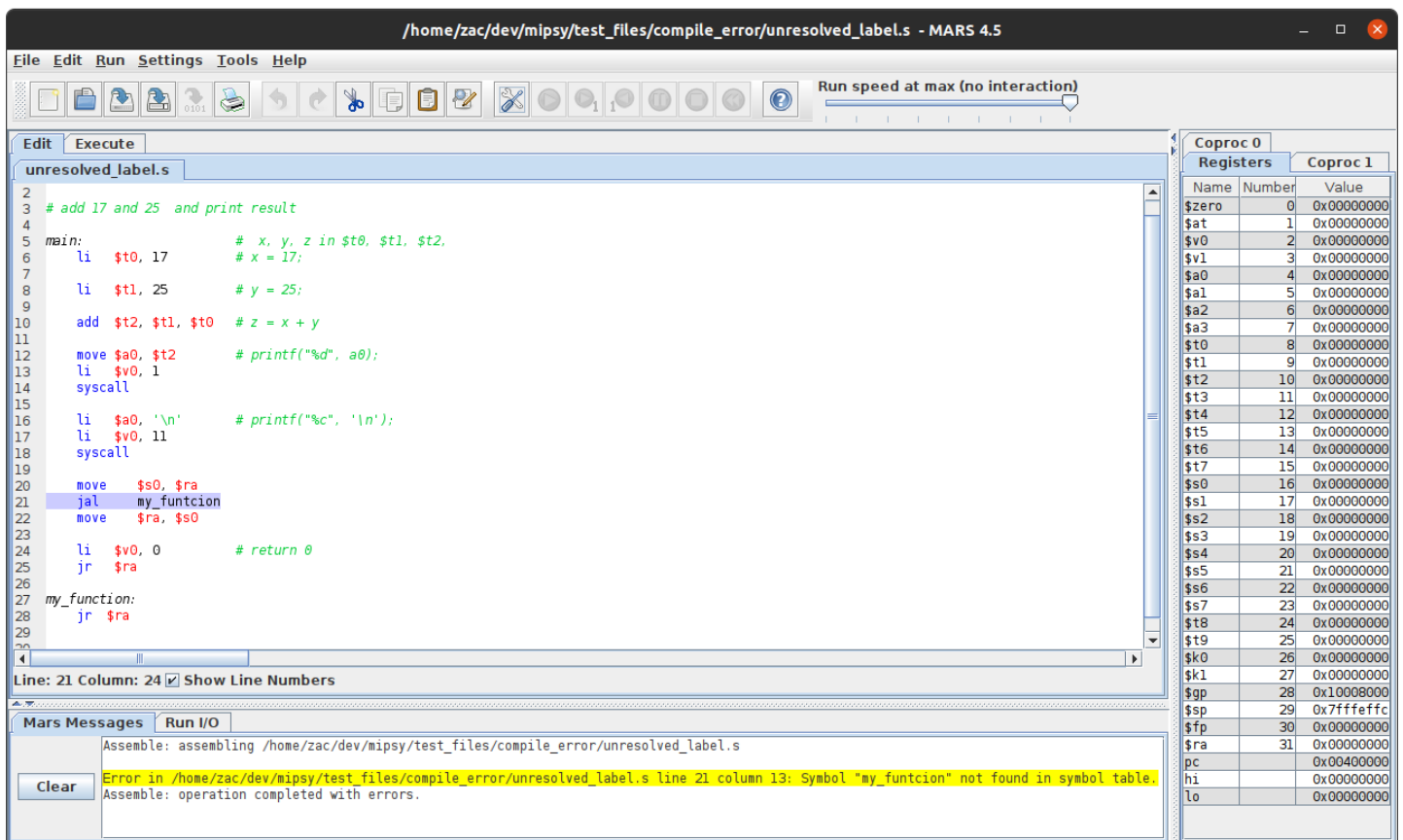


Figure 2.10: MARS - Misspelling a label used in a `jal`

After successfully compiling their program, the following is what a student is greeted with when they begin executing their program. Although it offers many features directly available to the student, it is extremely information-dense, which for beginners can pose very difficult challenges in trying to focus on what is currently important. Furthermore, although the individual panes are resizable, for students with lower resolution displays (especially when running the application on the university’s infrastructure through VNC, as many students do), it can be very difficult to use.

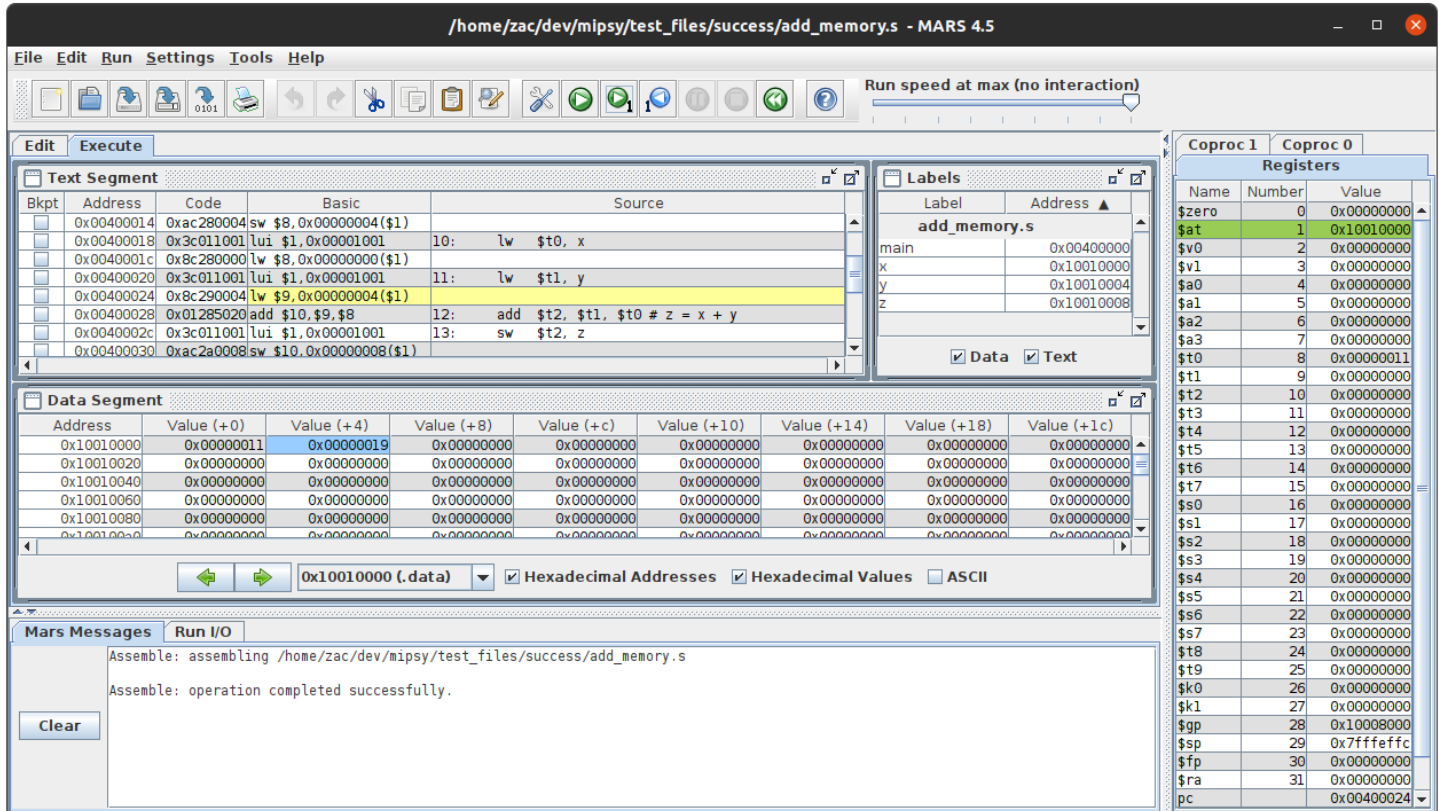


Figure 2.11: MARS - Stepping through a very simple program

MARS also offers a very simple command-line interface, however it has no interactive capabilities, and isn’t intended for use by students. A list of the available options when running on the command line is available at <https://courses.missouristate.edu/KenVollmar/MARS/Help/MarsHelpCommand.html>, prominently including tools for assessing and marking the work of students. Consequently, any error messages generated by the program are relatively unsound, and as such, it isn’t reasonable to compare it with mipsy.

2.3 EduMIPS64 / WepSIM / WebMIPS

Most other MIPS simulators (eg. EduMIPS64, WepSIM, WebMIPS) all exist in the goal of education, but specifically to aid in the teaching of hardware, rather than software. This is something that all of SPIM, MARS, and mipsy do not attempt to partake in at all, as it generally is taught as a separate discipline to assembly programming (for example, UNSW teaches assembly programming in [COMP1521](#), and hardware design in [COMP3211](#) and [COMP3222](#)).

The following figures show what these simulators look like in-use.

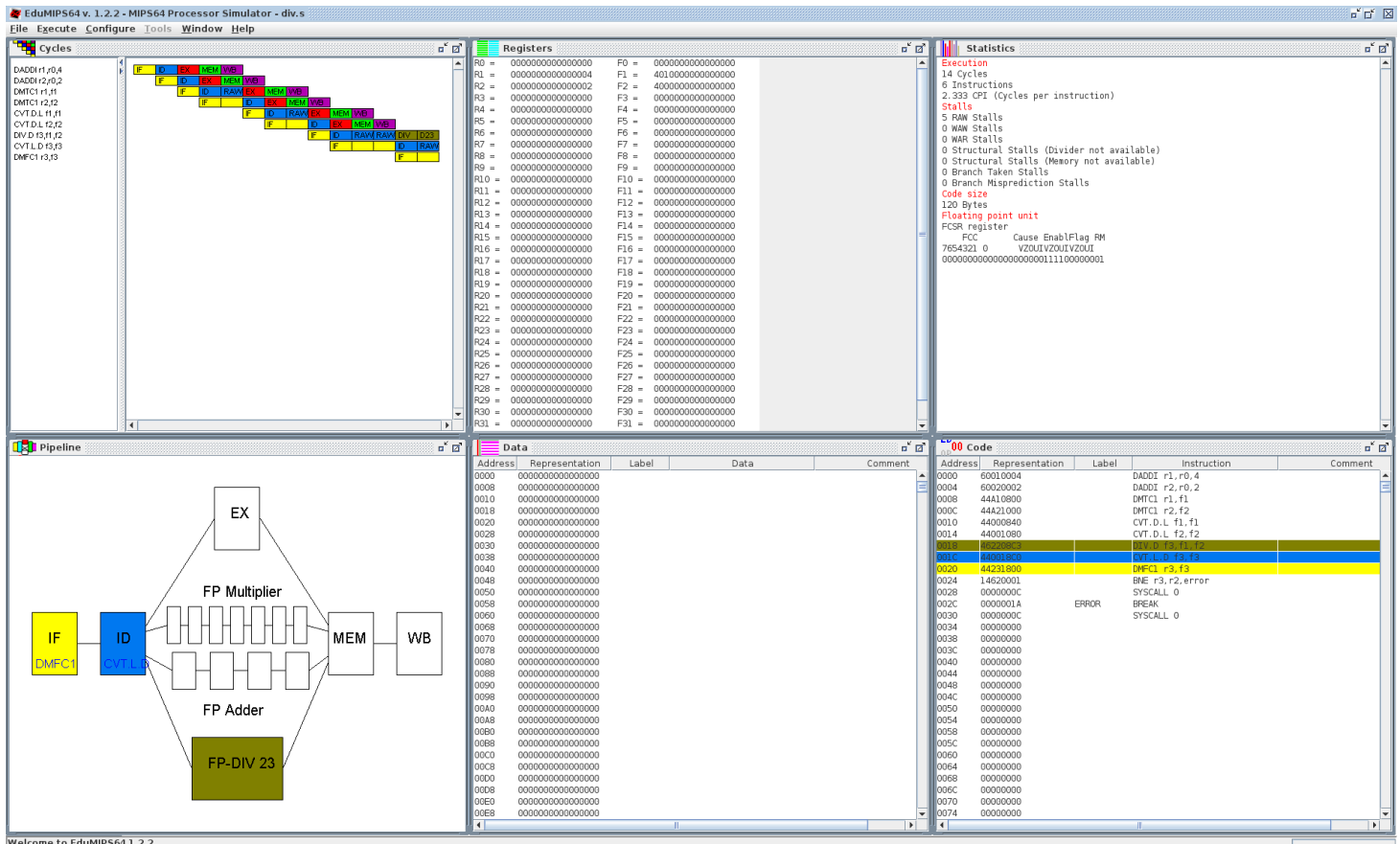


Figure 2.12: EduMIPS64

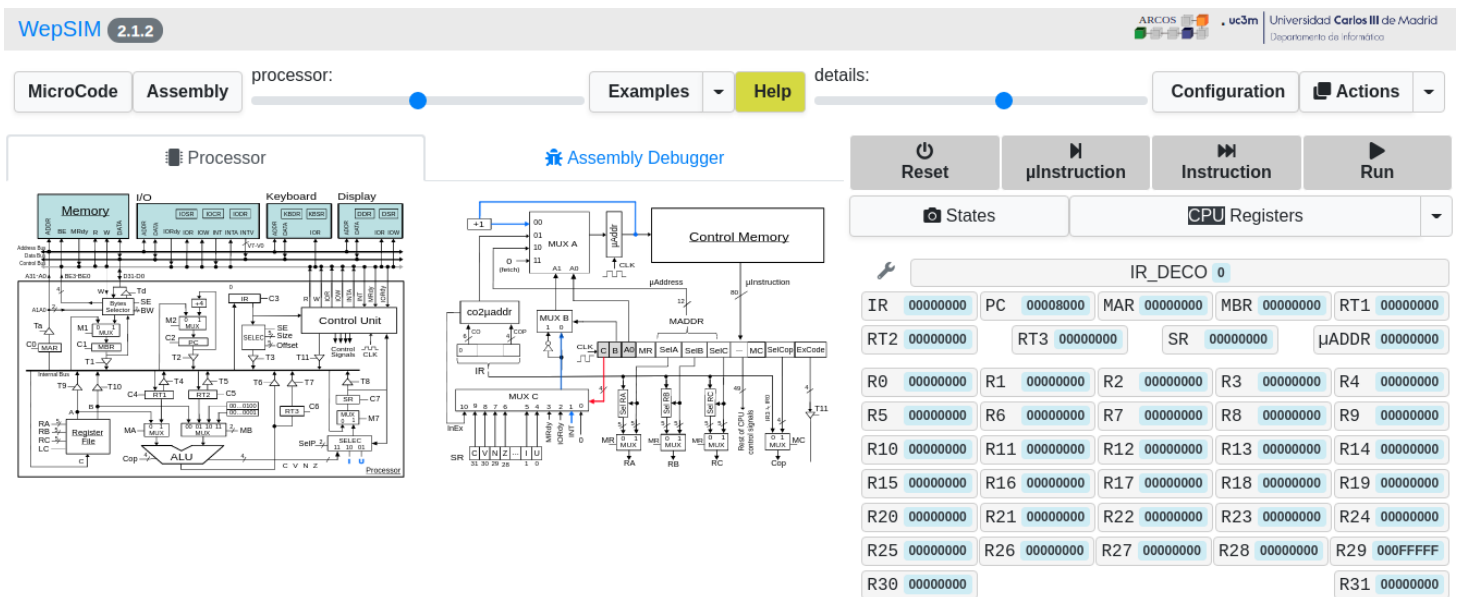


Figure 2.13: WepSIM

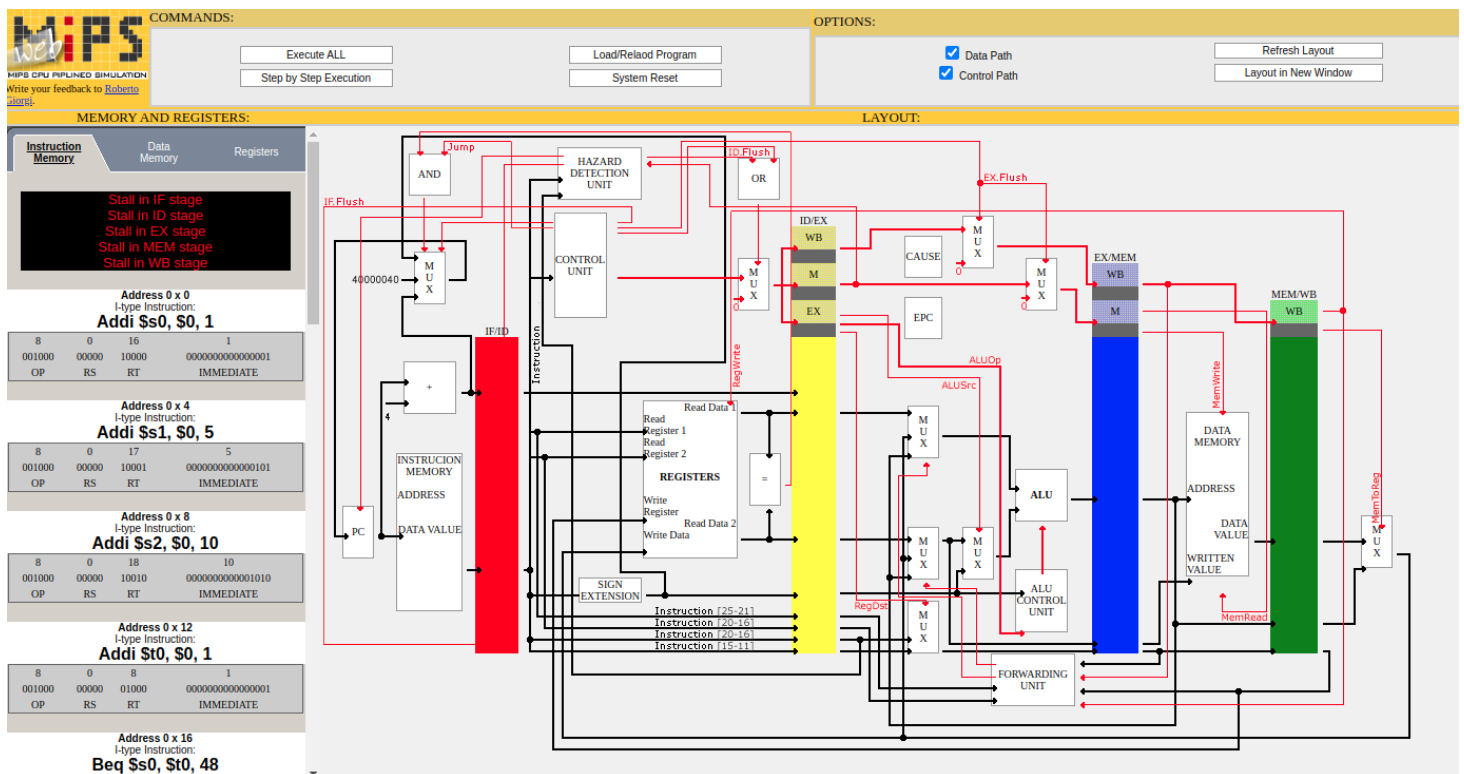


Figure 2.14: WebMIPS

Chapter 3

Mipsy: Goals

Mipsy's goals broadly fall into two distinct categories - for students, and for educators. Therefore in this report, they will be listed and evaluated as such. Each set of goals will be numerically listed, in order to make the following evaluation a more direct comparison of mipsy's goals and its current state.

3.1 For students

1. Mipsy should be able to execute working MIPS programs as expected, without any additional baggage (Similar to `spim -f <file>`).
 - 1.1. It should implement a similar subset of MIPS instructions to SPIM and MARS (i.e. most of the MIPS32 extended assembler instruction set).
2. Mipsy should handle compilation errors gracefully:
 - 2.1. Explaining the specific error that has occurred.
 - 2.2. Providing a useful suggestion wherever possible to help the student correct it.
 - 2.3. In all mechanisms of interaction (eg. `mipsy <file>`, interactive `mipsy`, inline REPL instructions).
3. Mipsy should handle runtime errors gracefully:
 - 3.1. Explaining the specific error that has occurred.
 - 3.2. Providing a useful suggestion wherever possible to help the student correct it.
 - 3.3. In all mechanisms of interaction (eg. `mipsy <file>`, interactive `mipsy`, inline REPL instructions).
 - 3.4. Proactively watch for and error on undefined behaviour (eg. use of an uninitialised register).
4. Mipsy should provide a powerful and intuitive interactive debugger:
 - 4.1. With a comprehensive suite of commands for gathering information and solving issues.
 - 4.2. Allowing students to step both forwards, and backwards instructions.

- 4.3. Aiming to provide maximum visual clarity when debugging.
- 4.4. With full readline support, feeling as similar as possible to a modern shell.

3.2 For educators

1. Mipsy should require minimal supporting education for students to be able to get started.
 - 1.1. Where sensible, it should try to act relatively similar to existing solutions, such as SPIM and MARS.
2. Mipsy should be easily customisable, allowing educators to:
 - 2.1. Modify the existing set of instructions provided in mipsy.
 - 2.2. Add custom pseudo-instructions where desired.
 - 2.3. Modify mipsy's internationalisation.
3. Mipsy should be extensible, allowing educators to use it as a library in order to develop custom tools for their course.
4. Mipsy should be tested, documented, and stable, in order to confidently integrate it into a university course.

Chapter 4

Mipsy: Evaluation

Here, the current state of mipsy will be directly compared to the goals listed in chapter 3. As mipsy was built over the course of only 10 weeks during a single trimester in UNSW's [COMP3901](#), a subset of these goals either haven't been achieved in full, or are even completely unsound. These are considered to be top priority in mipsy's future work, post [COMP3901](#), which is examined more closely in Future Work: Completeness.

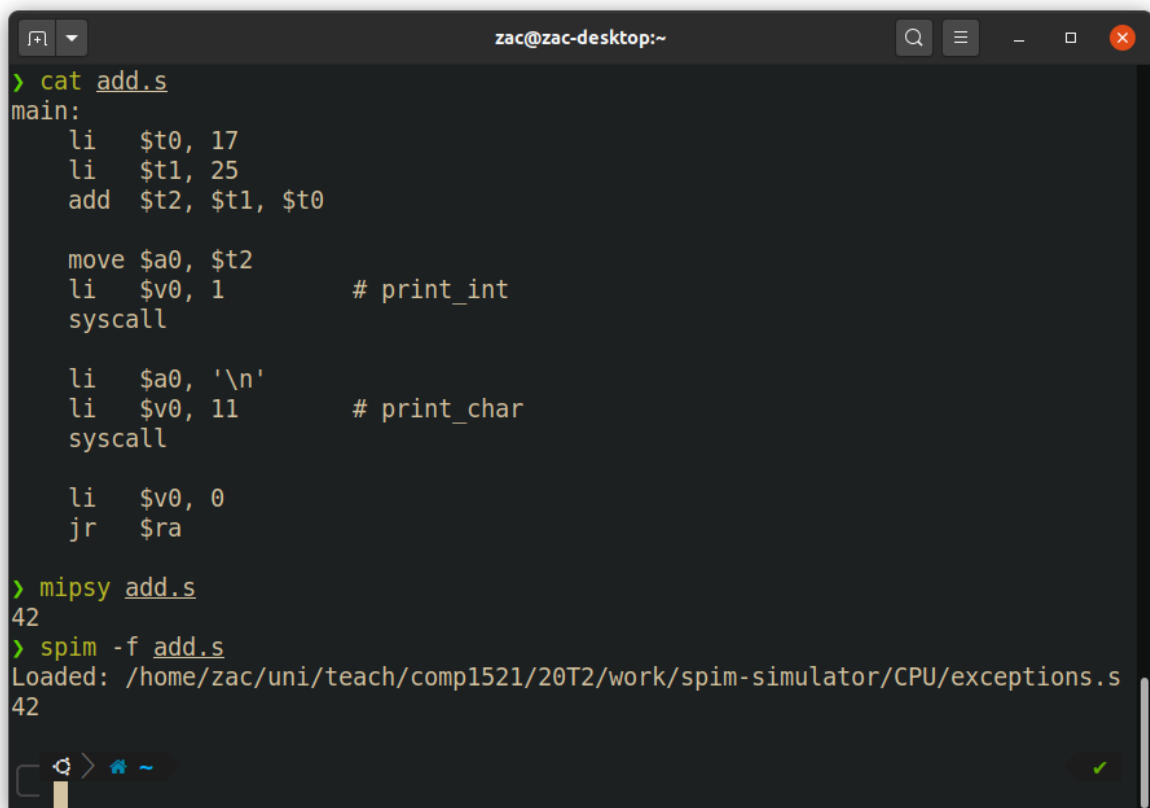
4.1 For students

Goal 1

Mipsy should be able to execute working MIPS programs as expected, without any additional baggage (Similar to `spim -f <file>`).

When given a valid MIPS program without runtime errors as direct command-line input, mipsy will execute exactly like other existing simulators, such as SPIM. This is necessary as there is a large amount of existing infrastructure relying on the behaviour of commands like `spim -f <file>`.

An example can be seen as follows:



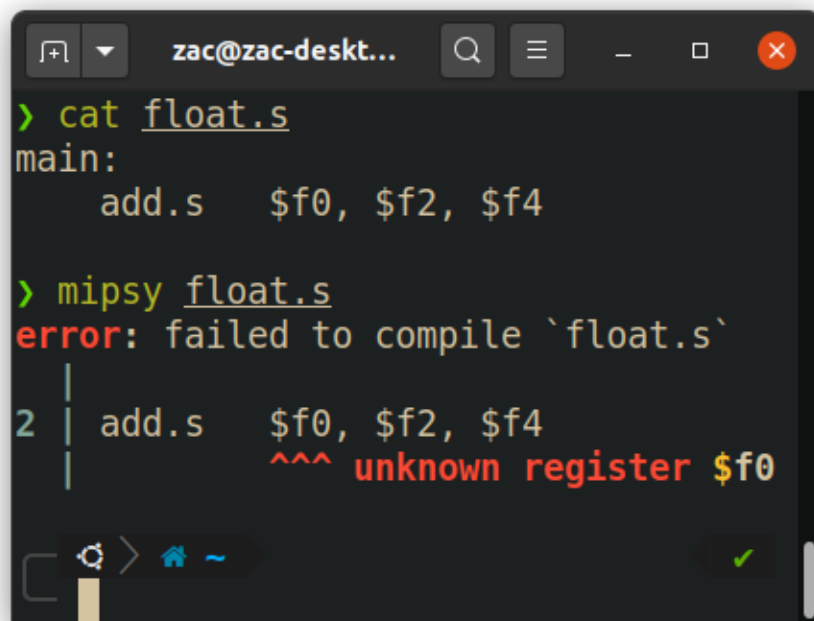
```
zac@zac-desktop:~  
> cat add.s  
main:  
    li    $t0, 17  
    li    $t1, 25  
    add   $t2, $t1, $t0  
  
    move  $a0, $t2  
    li    $v0, 1      # print_int  
    syscall  
  
    li    $a0, '\n'  
    li    $v0, 11     # print_char  
    syscall  
  
    li    $v0, 0  
    jr    $ra  
  
> mipsy add.s  
42  
> spim -f add.s  
Loaded: /home/zac/uni/teach/comp1521/20T2/work/spim-simulator/CPU/exceptions.s  
42
```

Figure 4.1: Mipsy - Run file directly

Goal 1.1

It should implement a similar subset of MIPS instructions to SPIM and MARS (i.e. most of the MIPS32 extended assembler instruction set).

Mipsy currently implements 54 native MIPS instructions, and 73 extra extended-format pseudo-instructions. This provides relatively good coverage of the majority of the MIPS I R2000 specification and the most commonly used MIPS32 instructions, but is a far cry from, for example, the 155 native instructions and 370 pseudo-instructions implemented in simulators like MARS. Mipsy also has quite poor floating-point instruction support currently, but it will not be difficult to implement. This is all planned to be rectified in Future Work: Completeness.

A terminal window titled 'zac@zac-deskt...' with standard window controls. The terminal shows the following commands and output:

```
> cat float.s
main:
    add.s    $f0, $f2, $f4

> mipsy float.s
error: failed to compile `float.s`
2 | add.s    $f0, $f2, $f4
  |           ^^^ unknown register $f0
```

The error message 'error: failed to compile `float.s`' is in red. The line number '2' is in green. The error details '^^^ unknown register \$f0' are in red. The terminal has a dark background with light-colored text.

Figure 4.2: Mipsy - Poor float support currently

Goal 2

Mipsy should handle compilation errors gracefully.

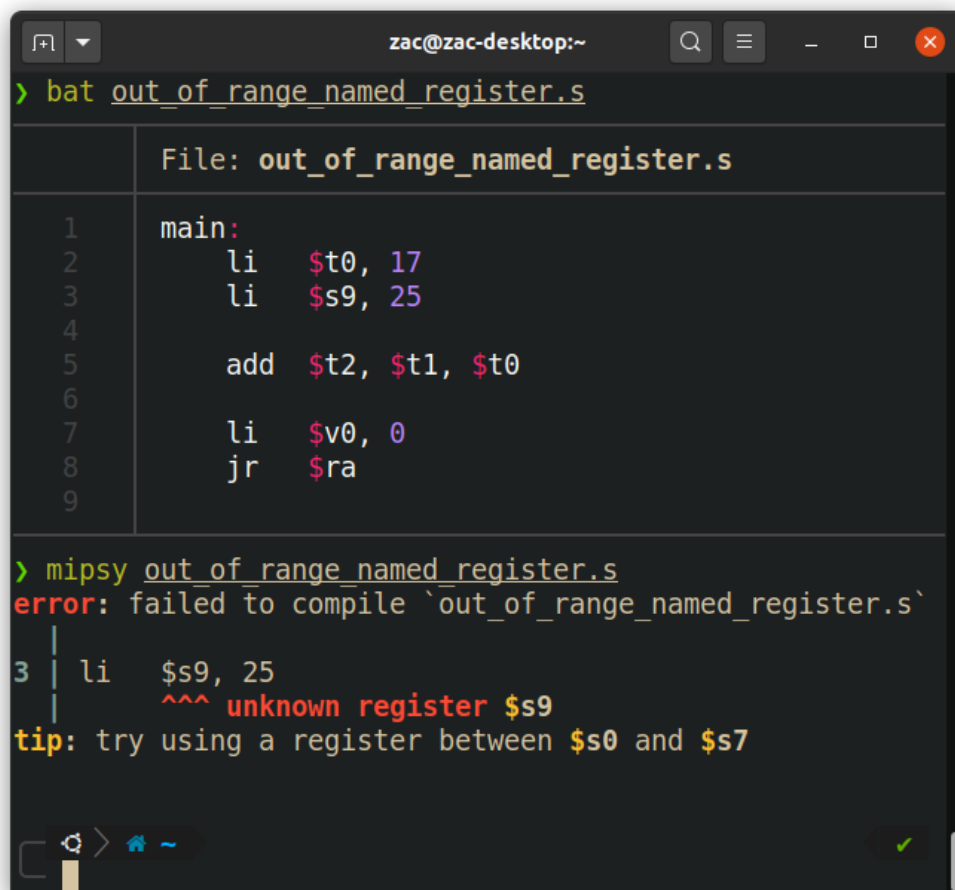
Goal 2.1

Explaining the specific error that has occurred.

Goal 2.2

Providing a useful suggestion wherever possible to help the student correct it.

Mipsy absolutely excels at providing correct, concise, graceful error messages. It always attempts to explain specifically what error has occurred, and if applicable, a useful suggestion to help the student rectify the issue.



```
zac@zac-desktop:~  
> bat out_of_range_named_register.s  
  
File: out_of_range_named_register.s  
1  main:  
2      li    $t0, 17  
3      li    $s9, 25  
4  
5      add   $t2, $t1, $t0  
6  
7      li    $v0, 0  
8      jr    $ra  
9  
  
> mipsy out_of_range_named_register.s  
error: failed to compile `out_of_range_named_register.s`  
3 | li    $s9, 25  
  |      ^^^ unknown register $s9  
tip: try using a register between $s0 and $s7
```

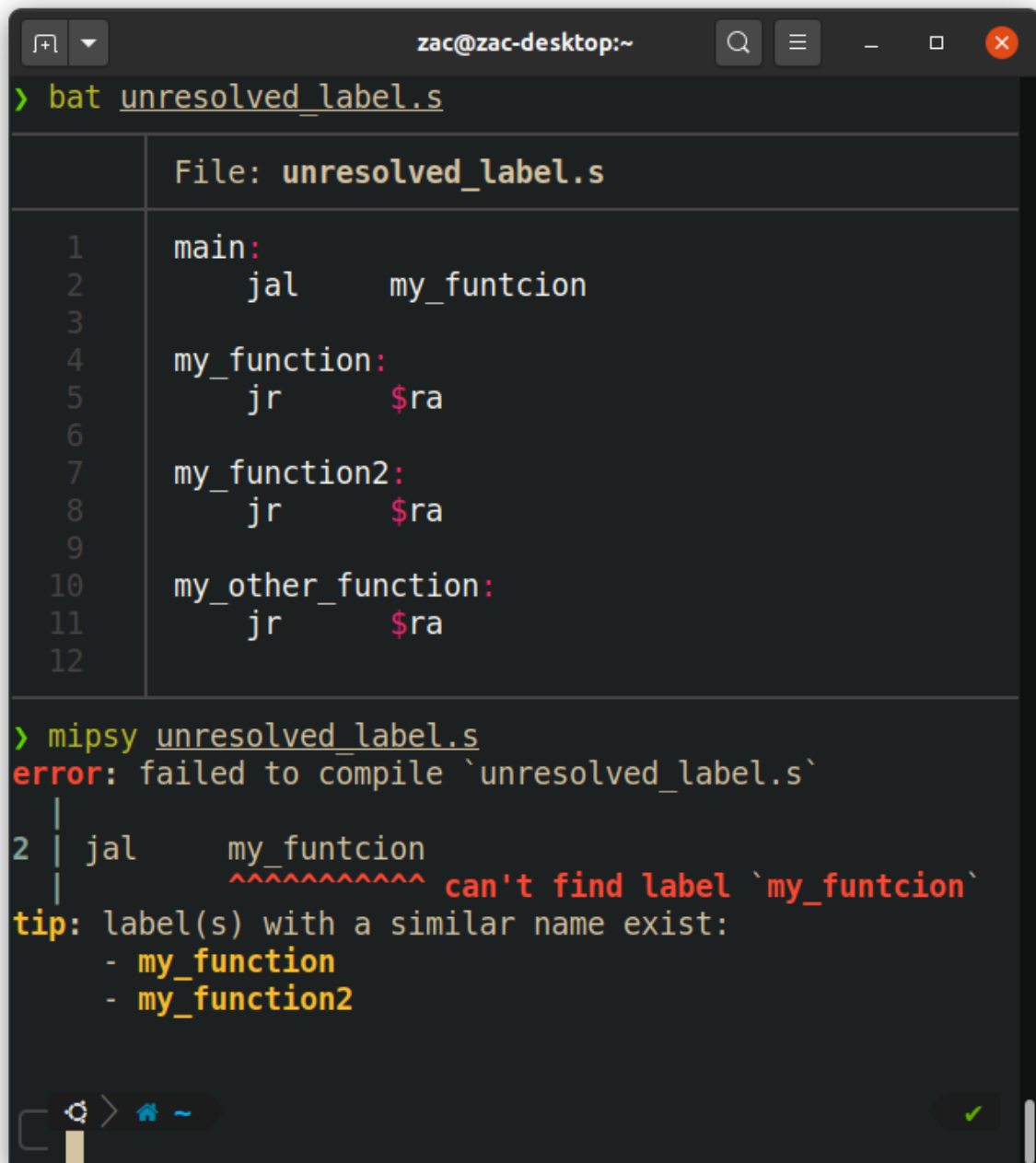
Figure 4.3: Mipsy - Unknown Register

```
zac@zac-desktop:~  
> bat unknown_inst.s  
File: unknown_inst.s  
1  main:  
2      li      $t0, 17  
3      aadiu   $t1, $t0, 25  
4  
5      jr      $ra  
6  
> mipsy unknown_inst.s  
error: failed to compile `unknown_inst.s`  
3 | aadiu $t1, $t0, 25  
  | ~~~~~ unknown instruction `aadiu`  
tip: instruction(s) with a similar name exist!  
try:  
- addiu $Rt, $Rs, i16  
- addiu $Rs, i16
```

Figure 4.4: Mipsy - Unknown Instruction

```
zac@zac-desktop:~  
> bat bad_inst_format.s  
File: bad_inst_format.s  
1  main:  
2      li      $t0, 17  
3      li      $t1, 25  
4  
5      add     $t2, 17, 25  
6  
7      jr      $ra  
8  
> mipsy bad_inst_format.s  
error: failed to compile `bad_inst_format.s`  
5 | add $t2, 17, 25  
  | ~~~~~ instruction `add` exists but was given incorrect arguments  
tip: valid formats for `add`:  
- add $Rd, $Rs, $Rt  
- add $Rs, $Rt, i16
```

Figure 4.5: Mipsy - Bad instruction format



The image shows a terminal window titled 'zac@zac-desktop:~'. The user has run the command `bat unresolved_label.s`, which displays the contents of the file `unresolved_label.s`. The file contains assembly code with three functions: `main`, `my_function`, and `my_function2`. The `main` function calls `my_funtcion` (note the typo). The other two functions use `jr $ra` to return. After viewing the file, the user runs `mipsy unresolved_label.s`. This results in an error: `error: failed to compile `unresolved_label.s``. A detailed error message follows: `2 | jal my_funtcion` with a caret under `my_funtcion` and the text `^^^^^^^^^^ can't find label `my_funtcion``. A tip is provided: `tip: label(s) with a similar name exist:` followed by a list:

- `my_function`
- `my_function2`

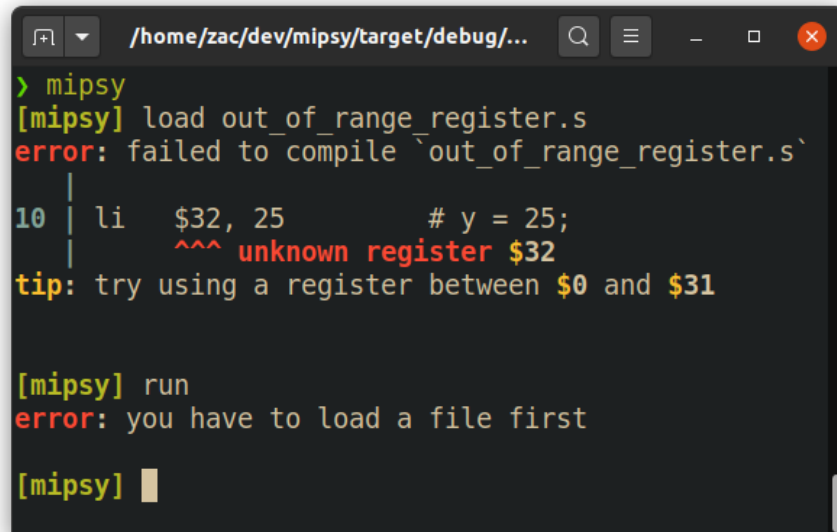
```
zac@zac-desktop:~  
> bat unresolved_label.s  
File: unresolved_label.s  
1  main:  
2      jal    my_funtcion  
3  
4  my_function:  
5      jr     $ra  
6  
7  my_function2:  
8      jr     $ra  
9  
10 my_other_function:  
11     jr     $ra  
12  
  
> mipsy unresolved_label.s  
error: failed to compile `unresolved_label.s`  
2 | jal    my_funtcion  
   |           ^^^^^^^^^^^ can't find label `my_funtcion`  
tip: label(s) with a similar name exist:  
    - my_function  
    - my_function2
```

Figure 4.6: Mipsy - Unresolved label

Goal 2.3

In all mechanisms of interaction (eg. `mipsy <file>`, interactive `mipsy`, inline REPL instructions).

Mipsy is able to give well-formatted error messages, no matter the specific source. Following is an example of loading a file interactively that doesn't compile, and trying to execute a non-compiling instruction with the inline REPL command.

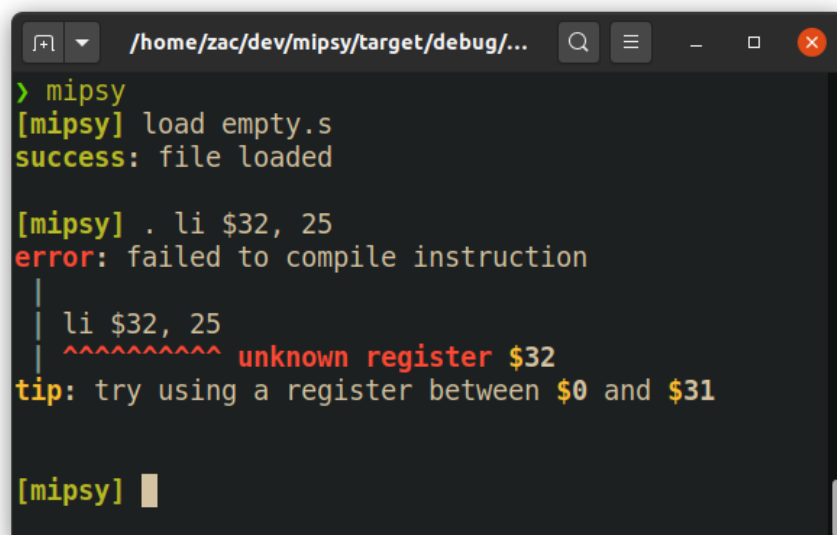


```
> mipsy
[mipsy] load out_of_range_register.s
error: failed to compile `out_of_range_register.s`
|
10 | li    $32, 25          # y = 25;
|      ^^ unknown register $32
tip: try using a register between $0 and $31

[mipsy] run
error: you have to load a file first

[mipsy] █
```

Figure 4.7: Mipsy - Loading a file interactively that doesn't compile



```
> mipsy
[mipsy] load empty.s
success: file loaded

[mipsy] . li $32, 25
error: failed to compile instruction
|
| li $32, 25
| ~~~~~~ unknown register $32
tip: try using a register between $0 and $31

[mipsy] █
```

Figure 4.8: Mipsy - Running an inline REPL command that doesn't compile

Goal 3

Mipsy should handle runtime errors gracefully.

Goal 3.1

Explaining the specific error that has occurred.

Goal 3.2

Providing a useful suggestion wherever possible to help the student correct it.

In the case of a runtime error, mipsy does an outstanding job of providing as much information to the student as possible about what has caused their program to crash, and relevant data. Following will be a few examples of common error cases that students often run into, and how mipsy deals with each error.

```

> bat overflow_add.s
File: overflow_add.s
1  main:
2      li    $t0, 0x7FFFFFFF
3      li    $t1, 42
4
5      add   $t2, $t0, $t1
6
7      li    $v0, 0
8      jr    $ra

```

Figure 4.9: An example MIPS program that will cause an arithmetic overflow exception

```

/home/zac/dev/mipsy/target/debug/mipsy
> mipsy
[mipsy] load overflow_add.s
success: file loaded

[mipsy] run

error: integer overflow

the instruction that failed was:
0x0040000c 5  [0x01095020]  add    $t2, $t0, $t1    # add    $t2, $t0, $t1

values:
- $t0 = 2147483647
- $t1 = 42
this happened because `2147483647` + `42` overflows past the limits of a 32-bit number

tip: if you expected the result to be -2147483607 (i.e. ignore overflow),
     then try using the equivalent unsigned instruction: `addu`

[mipsy] context

main:
0x00400000 2  [0x3c087fff]  lui    $t0, 32767      # li    $t0, 0x7FFFFFFF
0x00400004  [0x3508ffff]  ori    $t0, $t0, 65535
0x00400008 3  [0x3409002a]  ori    $t1, $zero, 42  # li    $t1, 42
0x0040000c 5  [0x01095020]  add    $t2, $t0, $t1  # add   $t2, $t0, $t1
0x00400010 7  [0x34020000]  ori    $v0, $zero, 0   # li    $v0, 0
0x00400014 8  [0x03e00008]  jr     $ra             # jr     $ra

[mipsy]

```

Figure 4.10: Executing the overflow_add.s program in mipsy



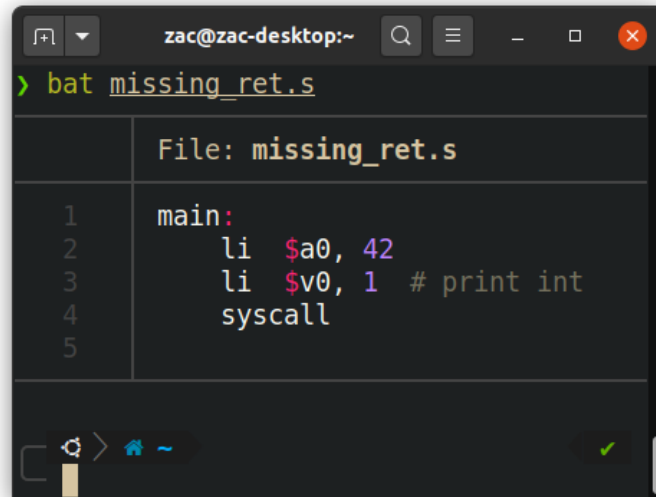
```
zac@zac-desktop:~  
> bat div_zero.s  
File: div_zero.s  
1 main:  
2     li    $t0, 42  
3     li    $t1, 0  
4  
5     div   $t2, $t0, $t1  
6  
7     jr    $ra
```

Figure 4.11: An example MIPS program that will cause an division by 0 exception



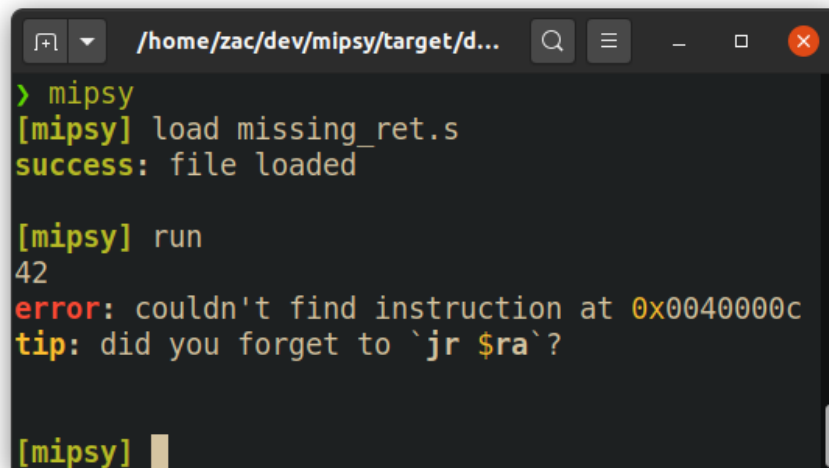
```
/home/zac/dev/mipsy/target/debug/mipsy  
> mipsy  
[mipsy] load div_zero.s  
success: file loaded  
  
[mipsy] run  
error: division by zero  
  
the instruction that failed was:  
0x00400008 5 [0x0109001a] div $t0, $t1 # div $t2, $t0, $t1  
  
values:  
- $t0 = 42  
- $t1 = 0  
  
[mipsy] context  
main:  
0x00400000 2 [0x3408002a] ori $t0, $zero, 42 # li $t0, 42  
0x00400004 3 [0x34090000] ori $t1, $zero, 0 # li $t1, 0  
0x00400008 5 [0x0109001a] div $t0, $t1 # div $t2, $t0, $t1  
0x0040000c [0x00005012] mflo $t2  
0x00400010 7 [0x03e00008] jr $ra # jr $ra  
[mipsy]
```

Figure 4.12: Executing the div_zero.s program in mipsy



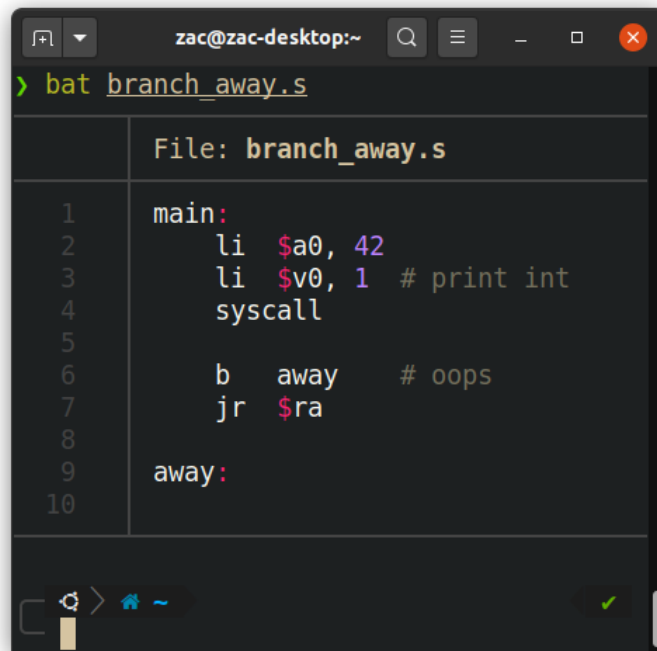
```
zac@zac-desktop:~  
> bat missing_ret.s  
File: missing_ret.s  
1 main:  
2     li $a0, 42  
3     li $v0, 1 # print int  
4     syscall  
5
```

Figure 4.13: An example MIPS program that forgets to return



```
/home/zac/dev/mipsy/target/d...  
> mipsy  
[mipsy] load missing_ret.s  
success: file loaded  
  
[mipsy] run  
42  
error: couldn't find instruction at 0x0040000c  
tip: did you forget to `jr $ra`  
  
[mipsy]
```

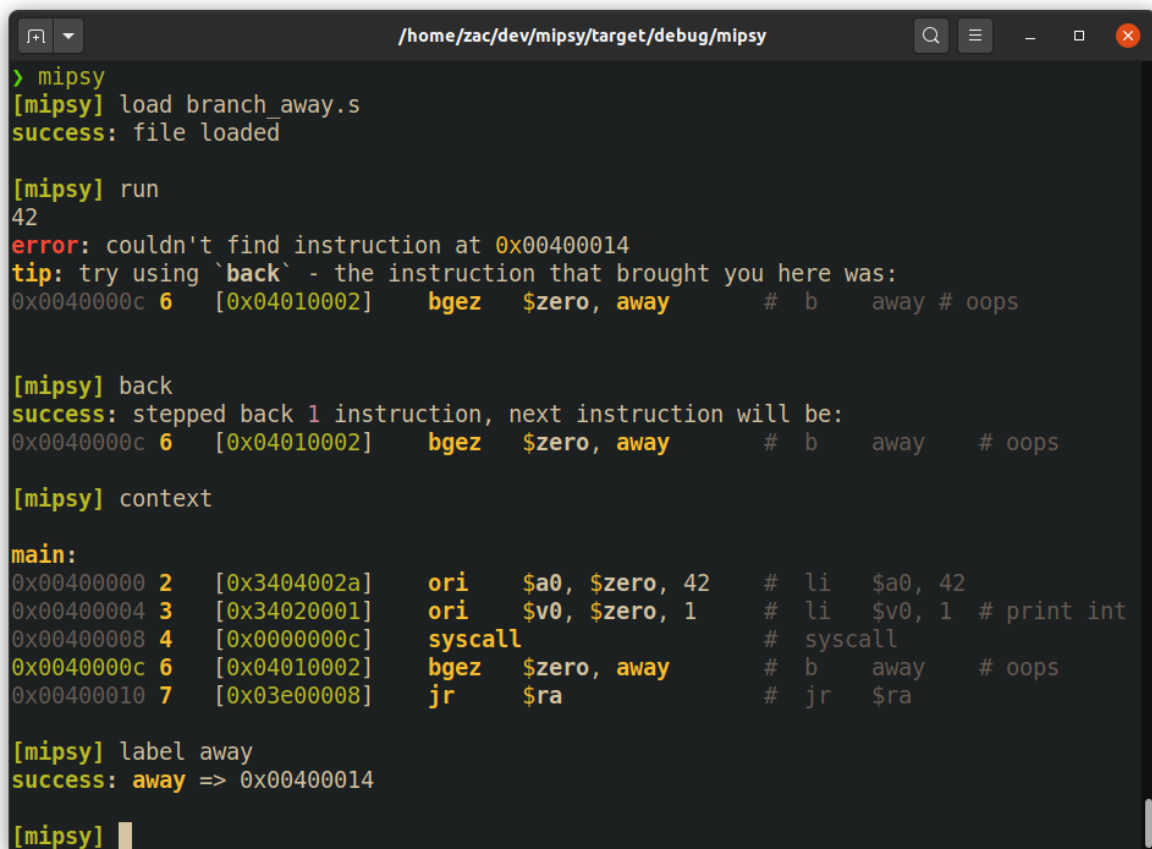
Figure 4.14: Executing the missing_ret.s program in mipsy



```
File: branch_away.s

1  main:
2      li $a0, 42
3      li $v0, 1 # print int
4      syscall
5
6      b  away   # oops
7      jr $ra
8
9  away:
10
```

Figure 4.15: An example MIPS program that branches away from real instructions



```
/home/zac/dev/mipsy/target/debug/mipsy

> mipsy
[mipsy] load branch_away.s
success: file loaded

[mipsy] run
42
error: couldn't find instruction at 0x00400014
tip: try using `back` - the instruction that brought you here was:
0x0040000c 6 [0x04010002] bgez $zero, away # b away # oops

[mipsy] back
success: stepped back 1 instruction, next instruction will be:
0x0040000c 6 [0x04010002] bgez $zero, away # b away # oops

[mipsy] context
main:
0x00400000 2 [0x3404002a] ori $a0, $zero, 42 # li $a0, 42
0x00400004 3 [0x34020001] ori $v0, $zero, 1 # li $v0, 1 # print int
0x00400008 4 [0x0000000c] syscall # syscall
0x0040000c 6 [0x04010002] bgez $zero, away # b away # oops
0x00400010 7 [0x03e00008] jr $ra # jr $ra

[mipsy] label away
success: away => 0x00400014

[mipsy]
```

Figure 4.16: Executing the branch_away.s program in mipsy

Goal 3.3

In all mechanisms of interaction (eg. `mipsy <file>`, interactive `mipsy`, inline REPL instructions).

Mipsy successfully handles errors clearly in all contexts. Following is an example of a division by zero using `mipsy` directly, interactive `mipsy`, and as an inline REPL instruction.



```
/home/zac/dev/mipsy/target/debug/mipsy
> bat quick_div_zero.s
File: quick_div_zero.s
1  main:
2      li $t0, 0
3      li $t1, 0
4      div $t2, $t0, $t1
> mipsy quick_div_zero.s
error: division by zero

the instruction that failed was:
0x00400008 4 [0x0109001a] div $t0, $t1 # div $t2, $t0, $t1

values:
- $t0 = 0
- $t1 = 0
> mipsy
[mipsy] load quick_div_zero.s
success: file loaded

[mipsy] run
error: division by zero

the instruction that failed was:
0x00400008 4 [0x0109001a] div $t0, $t1 # div $t2, $t0, $t1

values:
- $t0 = 0
- $t1 = 0

[mipsy] . div $t2, $t0, $t1
error: division by zero

values:
- $t0 = 0
- $t1 = 0

[mipsy]
```

Figure 4.17: A division by zero runtime error in all contexts

Goal 3.4

Proactively watch for and error on undefined behaviour (eg. use of an uninitialised register).

Mipsy actively watches for use of uninitialised registers or memory, similar to C runtimes such as LLVM's [Memsan](#), or [Valgrind](#). If it notices an occurrence, it will halt the student's program, and provide an explanation as to why. Furthermore, assuming the uninitialised register has been used since the program started executing, it will notify the student exactly where the register obtained that uninitialised value (usually as a result of reading uninitialised memory), and exactly how many steps backwards it will take to rewind to exact point it became uninitialised.



```
zac@zac-desktop:~  
> bat never_init_register.s  
File: never_init_register.s  
1  main:  
2      li      $t0, 1  
3      add     $t2, $t0, $t1  
4  
5      jr      $ra  
6  
> mipsy never_init_register.s  
error: your program tried to read uninitialised memory  
  
the instruction that failed was:  
0x00400004 3 [0x01095020] add $t2, $t0, $t1 # add $t2, $t0, $t1  
  
this happened because $t1 was uninitialised  
| note: $t1 was never initialised
```

Figure 4.18: Mipsy - Using an uninitialised register

```
/home/zac/dev/mipsy/target/debug/mipsy
> bat unittest_read.s
File: unittest_read.s
1  main:
2      li      $t0, 17
3      lw      $t1, my_variable
4
5      nop
6      nop
7      nop
8
9      add     $t2, $t0, $t1
10
11     jr      $ra
12
13     .data
14     my_variable: .space 4 .word 25

> mipsy
[mipsy] load unittest_read.s
success: file loaded

[mipsy] run

error: your program tried to read uninitialised memory

the instruction that failed was:
0x0040001c 9 [0x01095020] add $t2, $t0, $t1 # add $t2, $t0, $t1

this happened because $t1 was uninitialised
| the instruction that caused $t1 to become uninitialised was:
| 0x00400008 [0x8c290000] lw $t1, ($at)
|
| to get back there, use `back 4`

[mipsy] back 4
success: stepped back 4 instructions, next instruction will be:
0x0040000c [0x8c290000] lw $t1, ($at)

[mipsy] █
```

Figure 4.19: Mipsy - Using a register with uninitialised memory

Goal 4

Mipsy should provide a powerful and intuitive interactive debugger

Mipsy's interactive debugger is extremely powerful, and should feel quite familiar for students who have spent even a small amount of time in a Linux shell before. It offers commands to load and run files, step forwards instructions or backwards in time, auto-stepping to the next system call or user-input, adding, listing, and deleting breakpoints, listing program labels, decompiling the whole file or just a few instructions around the current point, printing the current values of registers and memory, and even offers a REPL command to execute MIPS instructions inline. It offers a help command for a broad overview of the available commands, and in-depth help for each specific command, including shortcut aliases.

It uses an extremely forgiving parser to interpret user commands, and uses [shlex](#) lexing to mimic how a shell would interpret arguments.

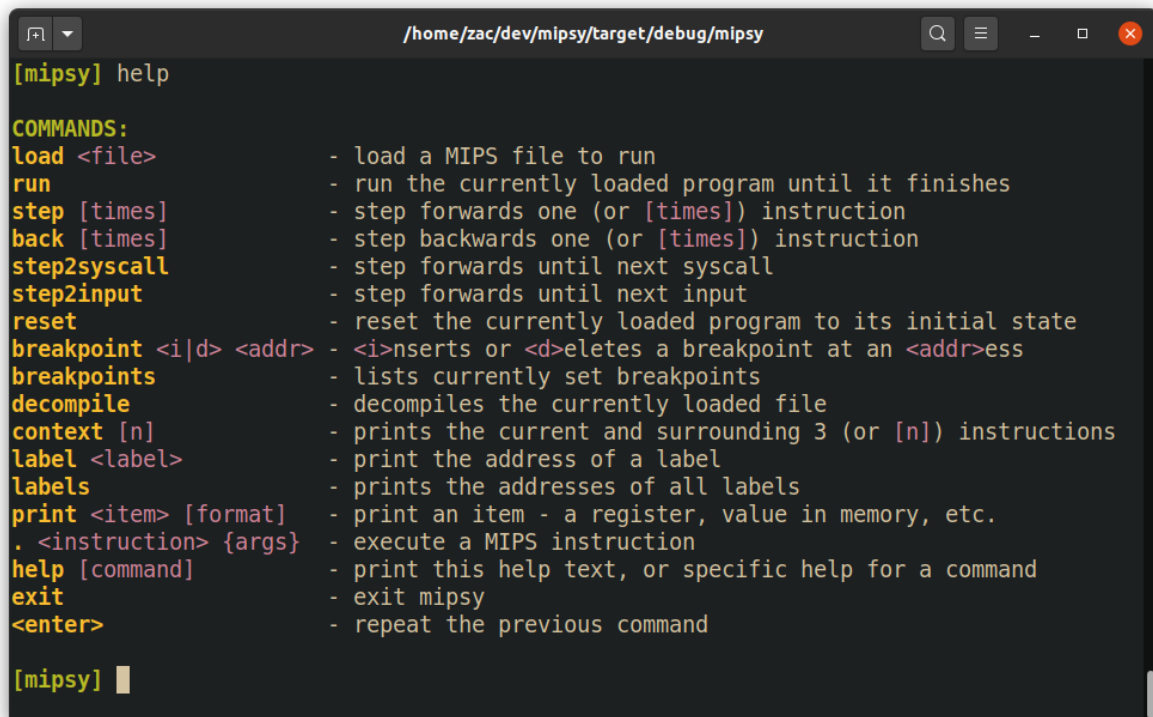
Goal 4.1

With a comprehensive suite of commands for gathering information and locating issues.

Goal 4.2

Allowing students to step both forwards, and backwards instructions.

The following is a list commands mipsy currently offers, available by running the `help` command.



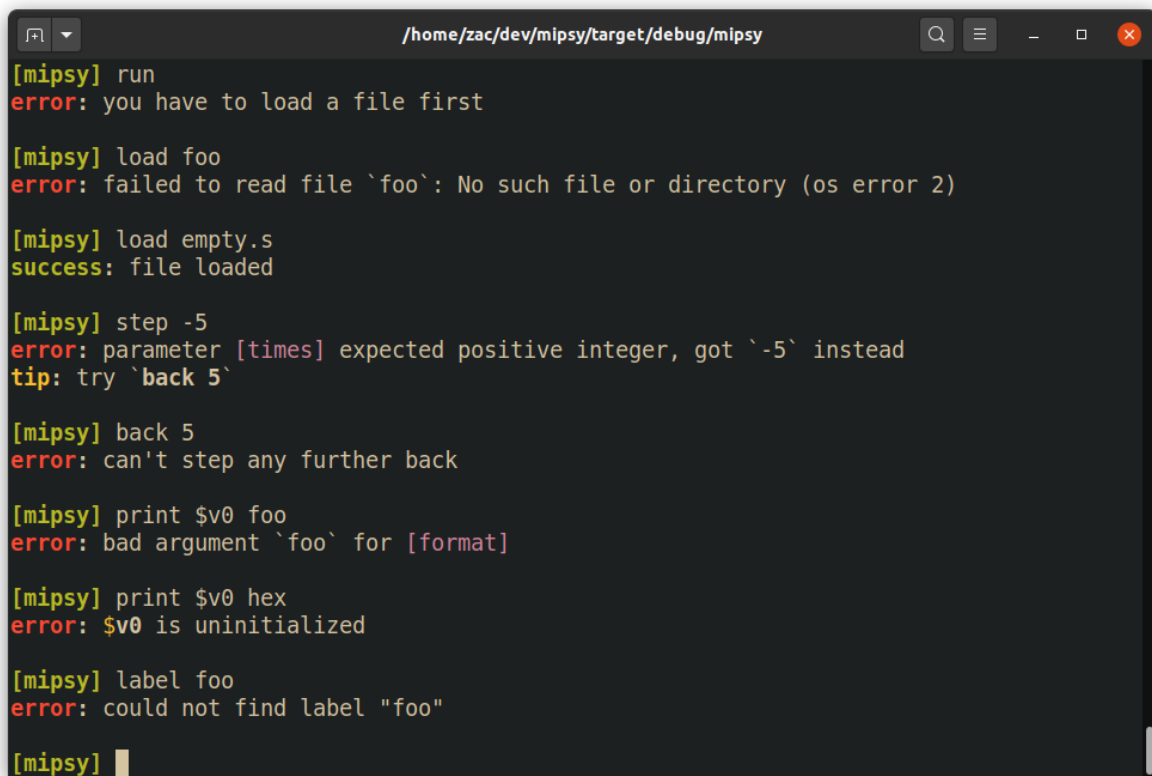
```
/home/zac/dev/mipsy/target/debug/mipsy
[mipsy] help

COMMANDS:
load <file>          - load a MIPS file to run
run                  - run the currently loaded program until it finishes
step [times]         - step forwards one (or [times]) instruction
back [times]         - step backwards one (or [times]) instruction
step2syscall         - step forwards until next syscall
step2input           - step forwards until next input
reset                - reset the currently loaded program to its initial state
breakpoint <i|d> <addr> - <i>nserts or <d>eletes a breakpoint at an <addr>ess
breakpoints          - lists currently set breakpoints
decompile            - decompiles the currently loaded file
context [n]          - prints the current and surrounding 3 (or [n]) instructions
label <label>        - print the address of a label
labels              - prints the addresses of all labels
print <item> [format] - print an item - a register, value in memory, etc.
. <instruction> {args} - execute a MIPS instruction
help [command]       - print this help text, or specific help for a command
exit                - exit mipsy
<enter>             - repeat the previous command

[mipsy] █
```

Figure 4.20: Mipsy's commands

When commands have been used incorrectly, mipsy helps the user understand exactly what has gone wrong.



```
[mipsy] run
error: you have to load a file first

[mipsy] load foo
error: failed to read file `foo`: No such file or directory (os error 2)

[mipsy] load empty.s
success: file loaded

[mipsy] step -5
error: parameter [times] expected positive integer, got `-5` instead
tip: try `back 5`

[mipsy] back 5
error: can't step any further back

[mipsy] print $v0 foo
error: bad argument `foo` for [format]

[mipsy] print $v0 hex
error: $v0 is uninitialized

[mipsy] label foo
error: could not find label "foo"

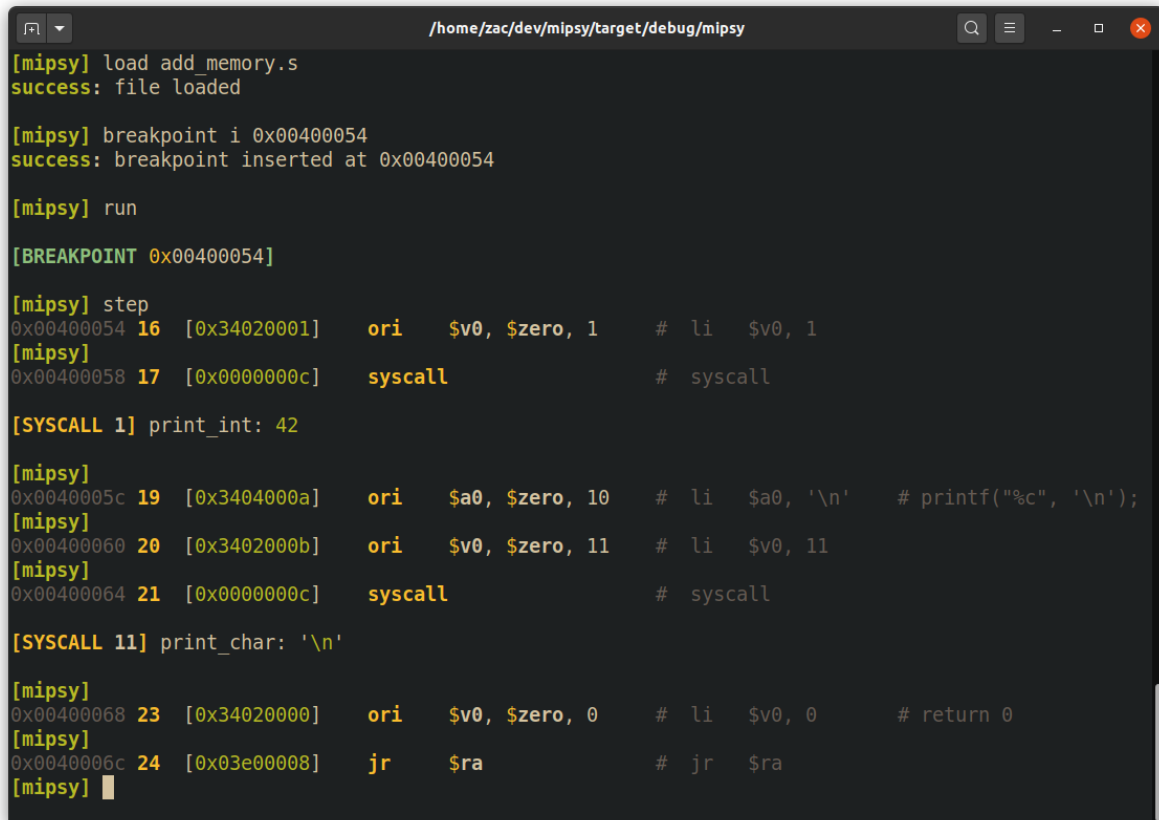
[mipsy] █
```

Figure 4.21: Errors using mipsy commands

Goal 4.3

Aiming to provide maximum visual clarity when debugging.

Mipsy makes liberal use of colour to provide the student visual clarity, helping them focus on which specific elements are important to them. When stepping through a program, the instruction being run, the source-code line number, the original source-code, and any invoked system calls are clear to see, rather than becoming cluttered in a mess of information.



```
[mipsy] load add_memory.s
success: file loaded

[mipsy] breakpoint i 0x00400054
success: breakpoint inserted at 0x00400054

[mipsy] run

[BREAKPOINT 0x00400054]

[mipsy] step
0x00400054 16 [0x34020001]    ori    $v0, $zero, 1    # li    $v0, 1
[mipsy]
0x00400058 17 [0x0000000c]    syscall                # syscall

[SYSCALL 1] print_int: 42

[mipsy]
0x0040005c 19 [0x3404000a]    ori    $a0, $zero, 10   # li    $a0, '\n'    # printf("%c", '\n');
[mipsy]
0x00400060 20 [0x3402000b]    ori    $v0, $zero, 11   # li    $v0, 11
[mipsy]
0x00400064 21 [0x0000000c]    syscall                # syscall

[SYSCALL 11] print_char: '\n'

[mipsy]
0x00400068 23 [0x34020000]    ori    $v0, $zero, 0    # li    $v0, 0        # return 0
[mipsy]
0x0040006c 24 [0x03e00008]    jr     $ra              # jr     $ra
[mipsy]
```

Figure 4.22: Stepping through a program in mipsy

Mipsy tries to help students keep track of where abouts in their program they are, by retaining defined labels, and offering a `context` command to print the current and surrounding instructions.

```

/home/zac/dev/mipsy/target/debug/mipsy

[mipsy] load labels.s
success: file loaded

[mipsy] decompile

main:
0x00400000 2  [0x34080011]  ori  $t0, $zero, 17  # li  $t0, 17
0x00400004 3  [0x34090019]  ori  $t1, $zero, 25  # li  $t1, 25

add:
0x00400008 6  [0x01095020]  add  $t2, $t0, $t1  # add  $t2, $t0, $t1
0x0040000c 8  [0x34020001]  ori  $v0, $zero, 1  # li  $v0, 1          # print_int
0x00400010 9  [0x000a2021]  addu $a0, $zero, $t2 # move $a0, $t2

print:
0x00400014 12 [0x0000000c]  syscall          # syscall
0x00400018 14 [0x03e00008]  jr    $ra        # jr    $ra

[mipsy] breakpoint i add
success: breakpoint inserted at add (0x00400008)

[mipsy] run

[BREAKPOINT add]

[mipsy] context 1
0x00400004 3  [0x34090019]  ori  $t1, $zero, 25  # li  $t1, 25

add:
0x00400008 6  [0x01095020]  add  $t2, $t0, $t1  # add  $t2, $t0, $t1
0x0040000c 8  [0x34020001]  ori  $v0, $zero, 1  # li  $v0, 1          # print_int

[mipsy]

```

Figure 4.23: Staying acquainted while debugging in mipsy

Goal 4.4

With full readline support, feeling as similar as possible to a modern shell.

Mipsy comes with complete readline support, meaning it can be used similar to a terminal shell. For example, left/right arrow keys and other emacs-style bindings (eg. C-a, C-e, C-w, M-b, M-f) can be used to navigate through the current line's buffer, up/down can be used to browse command history, C-r/C-s can be used to search command history, and more.

Mipsy also asks for confirmation after a ^C or ^D before exiting, as accidentally losing a session of debugging can be extremely frustrating and time-consuming. To quickly quit mipsy, it is suggested to either use ^C^C, or use the `q` alias for the `exit` command, which doesn't ask for confirmation.

4.2 For educators

Goal 1

Mipsy should require minimal supporting education for students to be able to get started.

Mipsy will definitely require a small amount of supporting education, but as it is very education-centric, it may end up less than what is currently required for a simulator like SPIM.

Important things to teach students planning to use mipsy:

1. The two main mechanisms of using mipsy: `mipsy <file>`, and interactive `mipsy`.
2. The `help` command, and how to use it to get specific help for a command.
3. The fundamental MIPS types recognised in mipsy:
 - 3.1. Register types: `$Rd`, `$Rs`, `$Rt`
 - 3.2. Immediate types: `i16`, `u16`, `i32`, `u32`.
 - 3.3. Floating types: `f32`, `f64`.
 - 3.4. Offset types: `OffRs`, `OffRt`, `Off32Rs`, `Off32Rt`.
Note: these refer to 16/32bit signed offset address types, eg: `imm($Rs)`.
 - 3.5. Misc. types: `shamt`, `J`.

This is important to ensure students understand error messages regarding incorrect instruction formats.

This will give students the resources they need to start using mipsy independently. Most other features are either self-explanatory (eg. `load`, `run`, `exit`), well-known concepts with many explanations online (eg. `breakpoints`), or are not necessary for a beginner to confidently use mipsy (eg. the `"."` command).

Goal 1.1

Where sensible, it should try to act relatively similar to existing solutions, such as SPIM and MARS.

Many of mipsy's features directly parallel those of which are available in popular simulators such as SPIM and MARS. For example, `load`, `run`, `step`, etc. all act identically to SPIM, `back`, `reset`, and `labels` all act identically to MARS. Commands that don't perfectly match up with a SPIM/MARS feature often have a very obvious counterpart, which achieves a similar goal, but is usually formatted in a slightly different way to remain consistent (eg. `mipsy` vs SPIM's `breakpoint` commands).

Goal 2

Mipsy should be easily customisable, allowing educators to...

Goal 2.1

Modify the existing set of instructions provided in mipsy.

Goal 2.2

Add custom pseudo-instructions where desired.

It is extremely simple to modify mipsy's existing instruction set, and to even create your own custom pseudo-instruction as an educator. One simply has to modify the `mips.yaml` file in the root of mipsy, and build the project.

```
1 instructions:
2   # R-Type Instructions
3   - name: SLL
4     desc_short: Shifts the value in $Rt left by Sa logically, storing ...
5     desc_long: >
6       insert long description here if you so please
7     compile:
8       format: [Rd, Rt, Shamt]
9     runtime:
10      type: R
11      funct: 0x00
12
13   - name: SRL
14     desc_short: Shifts the value in $Rt right by Sa logically, storing ...
15     compile:
16       format: [Rd, Rt, Shamt]
17     runtime:
18      type: R
19      funct: 0x02
20
21   ...
```

For example, in order to add two custom pseudo-instructions PUSH and POP to mipsy, this is all the code necessary to make such a change.

```
1 pseudoinstructions:
2 # ...
3 - name: PUSH
4   compile:
5     format: [Rt]
6   expand:
7     - inst: ADDIU
8       data: [$sp, $sp, -4]
9     - inst: SW
10      data: [$rt, ($sp)]
11
12 - name: POP
13   compile:
14     format: [Rt]
15   expand:
16     - inst: LW
17       data: [$rt, ($sp)]
18     - inst: ADDIU
19       data: [$sp, $sp, 4]
```

Removing existing pseudo-instructions is even easier - simply remove them from `mips.yaml`.

Goal 2.3

Modify mipsy's internationalisation.

Mipsy unfortunately has no internationalisation support, and doesn't provide a way to disable colour support currently. This is one of the highest-priority features to come, in order for mipsy to be viably used in international universities, and shell pipelines. See Future Work: Completeness for more information.

Goal 3

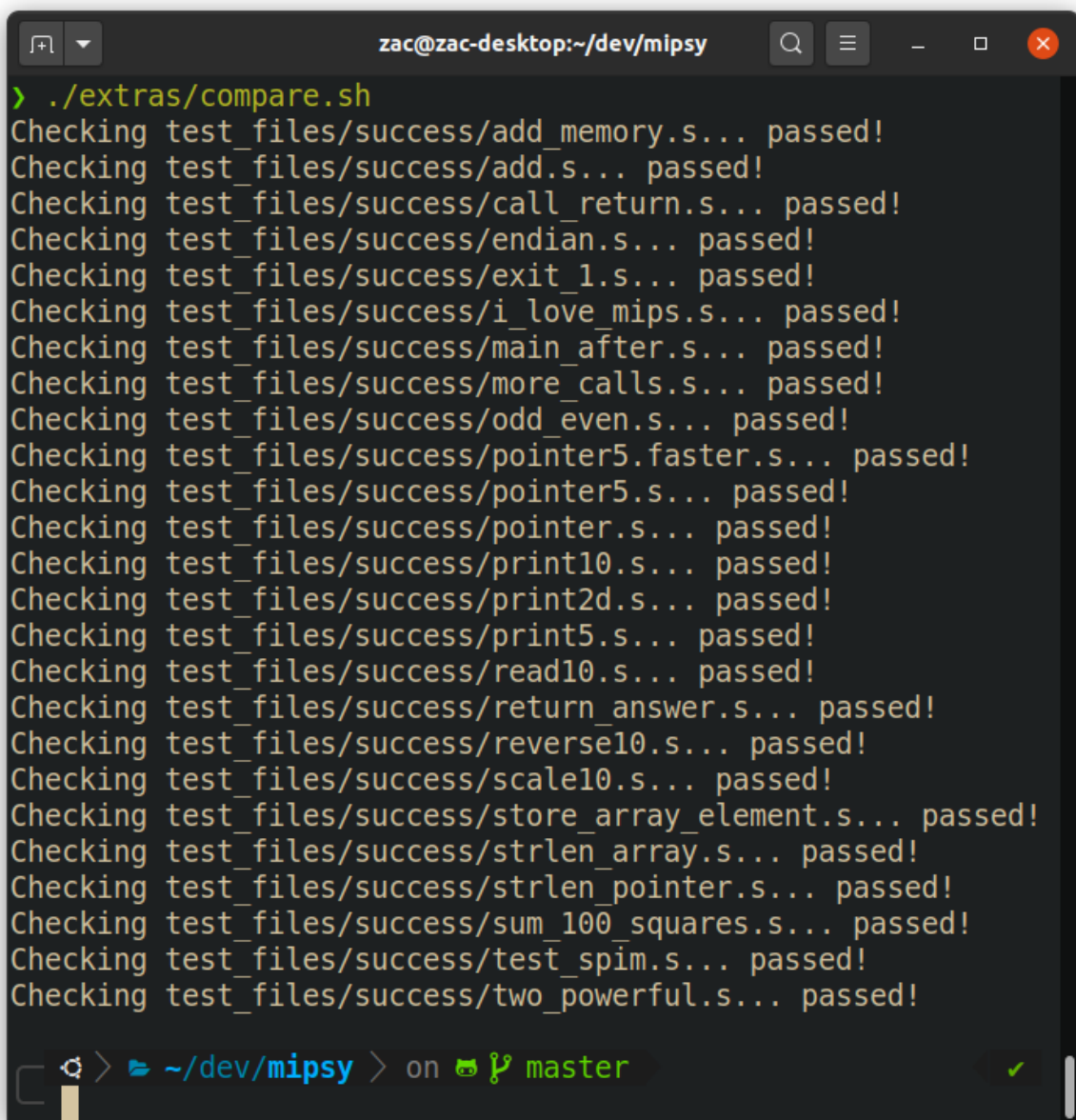
Mipsy should be extensible, allowing educators to use it as a library in order to develop custom tools for their course.

Mipsy has been written as separate packages: `mipsy_parser`, `mipsy_lib`, and `mipsy` (the binary executable). These can be used as independent libraries in order to develop course-specific tooling. For example, if students were expected to develop a heavily simplified MIPS simulator for an assignment, it would be beneficial to have a tool to simply assemble a MIPS file, and output the text segment as hex-codes. Although this is easily possible with `mipsy --hex`, a separate project has been created to demonstrate how to use `mipsy_lib`, available at <https://github.com/insou22/mipsy2hex>.

Goal 4

Mipsy should be tested, documented, and stable, in order to confidently integrate it into a university course.

Mipsy is relatively stable, sparsely tested, and vaguely documented. Overall, this is another top priority goal that is still yet to be realised. It is an unfortunate side-effect of mipsy's extremely rapid development, which underwent several rewrites along the way. It currently has unit testing on certain aspects of the parser, and integration tests, where it is directly compared to the output of SPIM in a suite of correct MIPS files, which is available in `./extras/compare.sh`. The majority of remaining testing is currently carried out manually by inspection, which is far from ideal.



```
zac@zac-desktop:~/dev/mipsy
> ./extras/compare.sh
Checking test_files/success/add_memory.s... passed!
Checking test_files/success/add.s... passed!
Checking test_files/success/call_return.s... passed!
Checking test_files/success/endian.s... passed!
Checking test_files/success/exit_1.s... passed!
Checking test_files/success/i_love_mips.s... passed!
Checking test_files/success/main_after.s... passed!
Checking test_files/success/more_calls.s... passed!
Checking test_files/success/odd_even.s... passed!
Checking test_files/success/pointer5.faster.s... passed!
Checking test_files/success/pointer5.s... passed!
Checking test_files/success/pointer.s... passed!
Checking test_files/success/print10.s... passed!
Checking test_files/success/print2d.s... passed!
Checking test_files/success/print5.s... passed!
Checking test_files/success/read10.s... passed!
Checking test_files/success/return_answer.s... passed!
Checking test_files/success/reverse10.s... passed!
Checking test_files/success/scale10.s... passed!
Checking test_files/success/store_array_element.s... passed!
Checking test_files/success/strlen_array.s... passed!
Checking test_files/success/strlen_pointer.s... passed!
Checking test_files/success/sum_100_squares.s... passed!
Checking test_files/success/test_spim.s... passed!
Checking test_files/success/two_powerful.s... passed!

~/dev/mipsy > on master
```

Figure 4.24: Cross-testing the suite of MIPS programs against SPIM

Chapter 5

Mipsy: Implementation

5.1 Rust backend

Mipsy is implemented using the [Rust programming language](#). It is a relatively new systems-programming language released in 2015, that provides lots of safety guarantees involving memory, executes without a garbage collector, and parallels performance of C / C++ applications.

Rust has an extremely expressive type system, with many parallels to ML family of languages. Mipsy has made use of this to simplify complex structures, such as the parser AST, in a completely type-safe manner. Mipsy is also guaranteed to have no security vulnerabilities related to memory unsafeties commonly associated with lower-level languages, as it does not make use of any unsafe code, and the Rust compiler is able to guarantee memory safety through its borrow-checker.

Rust also has very mature error-handling capabilities, allowing errors to easily be propagated upwards through the use of its `?` operator. This allows mipsy to encode many different variants of errors through Rust's algebraic data types, as described in the following section, and ensures errors don't accidentally get swallowed, or propagated all the way upwards into a process panic.

5.2 Parser

Mipsy's MIPS parser is available at https://github.com/insou22/mipsy/tree/master/mipsy_parser. The parser was implemented as a parser-combinator using the `nom` framework, alongside `nom_locate` for positional tracking during parsing.

It parses into a simple AST that is composed of Directives, Instructions, and Labels at the base of the tree. This was made extremely simple using Rust's algebraic data types, which it calls an `enum`. The following example shows how ADTs are used to encode the base of the AST (an `Item`) and Directives.

```
1  enum MPItem {
2      Instruction(MPInstruction),
3      Directive(MPDirective),
4      Label(String),
5  }
6
7  enum MPDirective {
8      Text,
9      Data,
10     KText,
11     KData,
12     Ascii(String),
13     Asciiiz(String),
14     Byte(Vec<i8>),
15     Half(Vec<i16>),
16     Word(Vec<i32>),
17     // ...
18 }
19
20 // ...
```

A parser-combinator is then used (in this case, `parse_mips_item`) to build off the functionalities of other smaller parsers, recursively applying in order to generate the AST.

```
1 fn parse_mips_item<'a>(i: Span<'a>) -> IResult<Span<'a>, (MPIItem, u32)> {
2     map(
3         tuple((
4             comment_multispace0,
5             position,
6             alt((
7                 map(parse_label,      |l| MPIItem::Label(l)),
8                 map(parse_directive,  |d| MPIItem::Directive(d)),
9                 map(parse_instruction, |i| MPIItem::Instruction(i)),
10            )),
11         comment_multispace0,
12     )),
13     |(_, pos, item, _)| (item, pos.location_line())
14 )(i)
15 }
```

This parser attempts to parse, in sequence:

- A `comment_multispace0` - meaning 0 or more whitespace characters, including comments.
- A `position`, which doesn't parse anything or consume any characters, but acts as a marker to get the current line later.
- An `alternative` of (in the ordered sequence) either a label, a directive, or an instruction, succeeding when one of the sub-parsers succeeds, and `mapping` the value it creates into the specific `MPIItem` ADT variant.
- And finally another `comment_multispace0`.

Rust's pattern matching then allows use of underscores to ignore the results of the multi-space parsers, and simply pull out the position and the item from the larger `tuple` parser, which is then mapped to a new tuple containing just the `MPIItem`, and the line it was parsed on.

This entire parser-combinator is then applied to the input `i` on line 14, which completes the base parser. This same technique is further employed for all of the sub-parsers, which can be inspected in the repository.

5.3 Compiler

The mipsy compiler takes a parsed AST and a MIPS instruction set (generated from `mips.yaml`) as input, and works in 5 stages.

1. It sweeps through the whole program, and attempts to find any obvious errors as soon as possible. This includes issues such as non-existent registers, unresolved labels, bad instruction formats, etc. It will then either report any errors it finds and abort, or allow the process to continue.

2. It will then sweep through the program, compiling only labels and directives. It will keep track of whereabouts in the text segment the program has currently progressed to by incrementing a base address each time an instruction is reached, but the instruction will not be compiled immediately. This is necessary as some instructions accept a label as input, which can point to a section later in the source-code that may not have been resolved yet. The data segment is filled out, and labels are all resolved to their respective addresses.
3. Finally, the program has a final sweep to compile each instruction, and fill out the text segment. This now is possible as all labels have been resolved, and their respective values can be substituted into the instructions' opcodes. This is also where pseudo-instructions are expanded, processed, and compiled.
4. The kernel data segment will then be compiled alongside any labels, in the exact same process as step 2. This must be done after compilation of the userland program, as the kernel file relies on the `main` label as an entry point.
5. The kernel text segment will then be compiled, in the exact same process as step 3.

Note - the contents of the kernel segment are generated dynamically from the [kern.s](#) file.

Once this process is complete, a binary will be generated and returned. This consists of:

- The text segment - a vector of words.
- The data segment - a vector of bytes, which may also be marked as uninitialised (eg. from `.space`, `.align`).
- The ktext segment - the kernel text.
- The kdata segment - the kernel data.
- The labels - a case insensitive hashmap of the label's name to its corresponding MIPS address.
- The globals - a vector of strings, corresponding to labels marked as `.globl`.
- The line_numbers - a hashmap of addresses to known line numbers from the source-code.

5.4 Runtime

The mipsy runtime is implemented as a timeline, of which a user can move forwards and backwards in. The timeline is comprised of a sequence of individual states, which encapsulate the current state of the simulator, between execution of instructions. Each state consists of a single-level page table representing the virtual memory, the value of each register, a program counter, and a heap size.

It exposes several functions to allow stepping forwards and backwards, and retrieval of values in any particular state inside the timeline. When stepping, the simulator will clone the current state, and inside the new state, it will fetch the current instruction, decode it, and execute it. If any errors arise during any of these stages, the cloned state is discarded,

and the error is propagated upwards. Otherwise, the new state is linked to the end of the chain, and a successful result is returned.

In the case of a system call or breakpoint instruction, the mipsy runtime will decode which system call is being requested, and delegate out to a provided "RuntimeHandler". This handler will either receive decoded values in the case of something like a print operation, or will be required to provide a value back in the case of a read operation. This means that no matter which manner mipsy is being used in, the runtime is flexible enough to allow the application to handle external interactions. This is especially important with regards to Future Work: Web browser - WASM.

5.5 CLI

The current CLI is planned to undergo a rewrite at the conclusion of COMP3901, and should be observed akin to a proof-of-concept of the possibilities available when using `mipsy_lib`. This is due to it undergoing extremely rapid development in order to work prior to the conclusion of the course, in which many technical compromises had to be made, making it difficult to make sweeping changes to (eg. internationalisation).

The CLI uses `clap` for argv parsing, and `rustyline` for the interactive readline capabilities. It uses `colored` and `strip-ansi-escapes` for ANSI colouring, and finally `shlex` for interactive argument grouping.

It uses a very simple, extensible, custom command framework, that makes it extremely simple to add new commands (for example, see the `label` command).

Commands can define their own possible errors (eg. `MustLoadFile`), which is then handled by the interactive manager. Any native `mipsy_lib` errors are then delegated out to the global error handler, which deals with failures that arise from within the actual compiler or runtime.

5.6 Challenges faced

The vast majority of the challenges faced during the development of mipsy arose from the extremely short timeline of the project. As UNSW currently works on trimesters of 10 weeks, it left very little time to do a gargantuan amount of work. Sometimes this led to lower quality code (as can be found in the mipsy binary), but throughout the project, the parser and library have been kept to a very high standard, each even undergoing multiple rewrites to ensure the codebase maintained its high quality.

A very prominent challenge also arose from the encoding of the instruction set. This underwent over 10 separate iterations, until the specific `mips.yaml` format was decided on. These iterations included:

- Hard-coding the instructions into the parser (SPIM's approach).
- Hard-coding the instructions into the Rust source-code.
- Generating the instructions from a txt file, in a DSL style (MARS's approach).
- Generating the instructions semi-dynamically from a YAML file, with a few special cases.

- Generating the instructions completely dynamically from a `mips.yaml` file, making use of smaller parser-combinators in `mipsy_parser` to substitute and compile individual instructions (mipsy's approach).

Mipsy's final approach is without a doubt the most flexible, extensible, user-friendly approach, with absolutely no special cases.

Chapter 6

Future Work

6.1 Web browser – WASM

Mipsy currently offers no graphical user interface for users. Instead, it is solely executable on the command line either directly, or by using the interactive debugger. Most other MIPS simulators offer a graphical equivalent (eg. SPIM's QtSpim, MARS's IDE), and the majority of these offerings are all native window applications using frameworks such as Qt and Java swing. However, with WebAssembly's recent popularisation and widespread adoption, targeting the web-browser has become quite a tempting option for mipsy.

This brings many advantages over traditional natively windowed applications - students no longer need to download or install any executables to use mipsy, individual operating systems don't need to be specifically targeted and supported, deploying new updates can be completely automated without building an auto-updater into the binary, and much more.

This is an option for mipsy for multiple reasons - it is written in the Rust programming language, which compiles to LLVM IR. The LLVM project provides a WebAssembly compilation target, and using this alongside libraries such as wasm-pack and wasm-bindgen allow packing a Rust library into a NPM package, with exported JavaScript bindings. Since Rust doesn't make use of heavy runtime features such as garbage collectors, it compiles to a similar binary size of what equivalent C/C++ programs do when statically linked. This means page-load times should be relatively quick with a decent internet connection, and performance should be high.

6.2 Language Server

A language server is the process that runs in the background while using a text editor, that provides suggestions on-the-fly, inline error messaging, line-by-line debugging, and other miscellaneous features. These servers work over the Language Server Protocol (LSP).

Creating a MIPS language server will be incredibly simplified by using `mipsy_lib`. This will allow students to have a better experience while in the process of writing code, due to the instantaneous feedback that will be provided. It will help students to write correct code, on their first attempt.

It is also feasible to include code suggestions, automatic code formatting, and even linting - similar to Rust's clippy, or ECMAScript's eslint. This could be bundled as a plugin for editors such as VSCode, Vim, Emacs, Gedit, and more.

6.3 Completeness

Multiple components of mipsy are incomplete, and are still pending a thorough implementation. This outstanding work includes:

1. The remaining MIPS32 instruction-set, including floating-point instructions.
2. The remaining common extended-format pseudo-instructions.
3. Documentation for the majority of the mipsy library functions.
4. Documentation for the majority of MIPS [pseudo-]instructions.
5. Extensive unit testing.
6. Extensive integration testing.
7. Internationalisation, with support to disable ANSI terminal-colours.

None of these tasks will be particularly difficult to implement in mipsy as it is today - but they all are simply very time-consuming tasks that weren't feasible to accomplish within mipsy's timeline.

Chapter 7

End Matter

7.1 Acknowledgments

A very warm thank you to everyone involved!

Project supervisor:

- Andrew Taylor

Advice and support:

- Dylan Brotherston
- Jashank Jeremy
- Tom Kunc
- Shrey Somaiya
- Olivia Tobin
- Oguz Kologlu
- The 20T3 comp1521 tutors

7.2 Appendixes

todo!()

7.3 References

todo!()