



Intro to MIPS

COMP1521 Week 2



What is MIPS?

- An assembly language. But what's that?
 - A low-level language that runs on a particular architecture (i.e. a particular kind of CPU). Examples include MIPS, x86, and ARM.
 - Compilers turn our C code into assembly language (which can then be turned into .o files by the assembler, and into ELF files by the linker — beyond the scope of this course).
- Why do we need assembly?
 - Computers don't understand C code. However, assembly languages are much closer to the actual machine code that CPUs do understand. A MIPS CPU, for example, can actually execute an "add" instruction.
- What are registers, by the way?
 - Tiny regions of memory in the CPU that are extremely fast to access (much faster than RAM; much, much faster than disk). MIPS is a 32-bit architecture, so each register holds 32 bits.

Meme

This was too big for the previous slide :(

Java

COBOL

Assembly

Physically inputting 0's
and 1's into the CPU slot
with electricity through a
copper wire



What is mipsy (and mipsy_web), and what do we use them for in COMP1521?

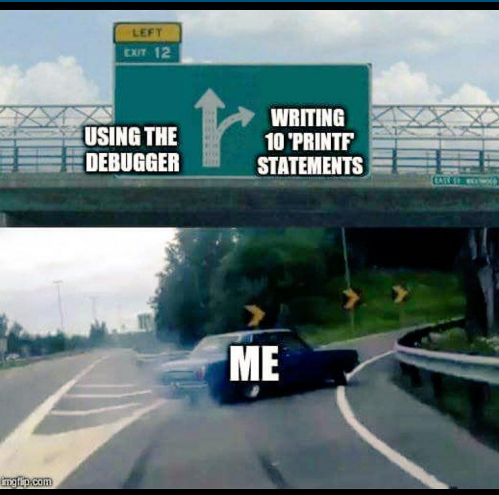
- Mipsy is a MIPS emulator which simulates the execution of a MIPS CPU and lets you run MIPS assembler on any computer (regardless of native architecture).
- Mipsy-web is a web version of mipsy that allows you to view your program state more easily.

A helpful extension

If you use VSCode, may want to install 'Mipsy Editor Features', which provides autocomplete, syntax highlighting, and more. I would highly recommend doing so.

Let's discuss debugging!

- A short demonstration? Or discussion?
- But first, some things not to do:



Registers and their “types”

- Different registers have different purposes. A lot of them are simply convention, but it is important to adhere to convention so that others can read and use our code.
- Broadly, there are three types:
 - General use registers
 - Function call registers
 - Other registers

Some MIPS conventions

Evil Andrew be like

```
# print a string in MIPS assembly
main:
    # ... print string to stdout as argument
    la $a0, string
    # ... 4 is printf "%s" syscall number
    li $v0, 4
    syscall
    li $v0, 0      # return 0
    jr $ra

.data
string:
    .asciiz "I love MIPS\n"
```

"i_love_mips.s" 14L, 265B 1,1 All

[mipsy] load i_love_mips.s
success: file loaded

[mipsy] step

start:
0x80000000 I [0x3c041001] li \$a0, 64
[mipsy] 0x80000004 ori \$k0, \$k0, 0
[mipsy] 0x80000008 kernel [0x0340f809] jalr \$ra, \$k0
[mipsy]

main:
0x00400000 5 [0x3c041001] li \$a0, 64
string
[mipsy]

Don't follow MIPS conventions

```
// print a string in C
int main(void) {
    printf("I love MIPS\n");
    return 0;
}
```

"i_love_mips.c" 9L, 107B 1,1 All

mipsy web

i_love_mips.s (unsaved file changes)

```
1 main:
2     address of string a:
3     a $a0, string
4     printf "%s" syscall
```



General use registers

- \$8-\$15, \$24, \$25
 - Temporary registers. But they can be referred to in other ways. How do we do that?
 - Yep, \$t0-\$t9.
- \$16-\$23
 - Saved registers, generally referred to as \$s0-\$s7.
 - These are no different from temporary registers until we get to MIPS functions.

Function call registers

- `$a0-$a3`
 - First four arguments to a function call (including syscalls, though these are a bit different from a normal function call).
 - Arguments that don't fit into 32-bits are placed on the stack (not commonly seen in this course, but worth noting)
- `$v0, $v1`
 - Return value(s) from a function. We will generally only use `$v0` in this course.
 - Also note that the return of syscall 5 (scanning in an integer), will be placed in `$v0`.
- `$ra`
 - Return address of the most recent function caller.
- `$sp, $fp`
 - Stack pointer, points to the top of the stack/frame pointer, points to bottom of stack frame.

Other registers

- \$zero
 - Always gives 0 upon read (ignores writes)
- \$at
 - Assembler temporary register. You can't use this register. Often used for pseudo-instructions.
- \$k0, \$k1
 - Reserved for kernel (i.e. operating system) use. You can't use these.