

MIPS Assembly Programming

Robert Winkler

Version 0.7.4, 2021-06-18: Beta

Table of Contents

Info	1
Chapter 0: Hello World	2
Prereqs	2
System Setup	2
Hello World	3
Building and Running	4
Conclusion	5
Chapter 1: Data	6
Arrays	7
Chapter 2: System Calls	9
Examples	10
Chapter 3: Branches and Logic	14
Practice	15
Conclusion	24
Chapter 4: Loops	25
Looping Through Arrays	27
Conclusion	32
Chapter 5: Functions and the MIPS Calling Convention	33
Functions	33
The Convention	34
Conclusion	41
Chapter 6: Floating Point Types	42
Floating Point Registers and Instructions	42
Practice	43
Getting Floating Point Literals	43
Branching	44
Functions	45
Conclusion	46
Chapter 7: Tips and Tricks	47
Formatting	47
Misc. General Tips	47
Constants	48
Macros	49
Switch-Case Statements	51
Command Line Arguments	54
References and Useful Links	60

Info

© 2021 [Robert Winkler](#)

Licensed under [Creative Commons](#).

This book is available online in both [HTML](#) and [PDF](#) form.

The repo for the book, where you can get the code referenced and report any errors (submit an issue or even a pull request) is [here](#).

Chapter 0: Hello World

In which we lay the groundwork for the rest of the book...

Prereqs

While someone with no programming experience could probably learn MIPS from this book, it is definitely preferable to have at least some experience in a higher level imperative programming language. I say imperative, because programming in assembly is the antithesis of functional programming; everything is about state, with each line changing the state of the CPU and sometimes memory. Given that, experience in functional languages like Lisp, Scheme etc. are less helpful than experience in C/C++, Java, Python, Javascript etc.

Of all of the latter, C is the best, with C++ being a close second because at least all of C exists in C++. There are many reasons C is the best prior experience when learning assembly (any assembly, not just MIPS), including the following:

- pointers, concepts and symmetry of "address of" and "dereference" operators
- pointer/array syntax equivalence
- stack allocation as the default
- manual memory management, no garbage collector
- global data
- rough equivalence in structure of a C program and an assembly program (vs. say Java)
- pass by value

There's some overlap between those and there're probably more but you can see that most other languages that are commonly taught as first languages are missing most if not all of those things.

Even C++, which technically has all of them, being a superset of C, is usually taught in a way that mostly ignores all of those things. They teach C++ as if it's Java, never teaching the fundamentals. In any case this is getting into my problems with CS pedagogy of the last 20 years based on my experience as a CS major myself ('12) and as a programming tutor helping college students across the country since 2016, and I should save it for a proper essay/rant sometime.

Long story short, I use C code and C syntax to help explain and teach MIPS. I'll try to provide enough explanation regardless of past experience as best I can.

System Setup

As I tell all of my tutoring students, if you're majoring in CS or anything related I highly recommend you use Linux. It's just easier in every way to do dev work on Linux vs Windows or Mac. Many assignments require it, which often necessitates using a virtual machine (which is painful, especially on laptops) and/or ssh-ing into a school Linux server, which is also less than ideal. In general, you'll have to learn how to use the unix terminal eventually and will probably use it to some extent in your career so it also just makes sense to get used to it asap.

That being said, Windows does now have WSL so you can get the full Ubuntu or Debian or Fedora etc. terminal based system on Windows without having to setup a real virtual machine (or dealing with the slowdown that would cause). I've even heard that they'll get support for Linux GUI programs soon.

MacOS on the other hand, is technically a Unix based system and you can use their terminal and install virtually any program from there using Macports or Homebrew or similar.

There are 3 commonly used MIPS simulators that I know of:

- SPIM is probably the oldest and is terminal only
- QtSpim is a GUI front end for SPIM
- MARS is a Java GUI based simulator with dozens of extra syscalls, syntactic sugar and features like graphics, memory mapped I/O, etc.

SPIM and QtSpim are in the Debian/Ubuntu repos so you can install them with the following

```
$ sudo apt install qtspim spim
```

You can probably find them in the repos for other distros too and install them similarly.

If it's not in your repos or you're on Windows (and not using WSL) or Mac you can download them, and MARS at these addresses

- [MARS](#)
- [QtSpim](#)

Hello World

Let's start with the classic hello world program, first in C, then in MIPS, and go over all the pieces in overview. You can copy paste these into your editor of choice (mine being neovim), or just use the files in the associated repo to follow along.

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf("Hello World!\n");
6     return 0;
7 }
```

It's pretty self explanatory. You have to include `stdio.h` so you can use the function `printf` (though in the real world I'd use `puts` here), the function `main` is the start of any C/C++ program, which is a function that returns an `int`. We call `printf` to display the string "Hello World!\n" to the user and then return 0 to exit. 0 indicates success, no error btw.

You can compile and run it in a linux/unix terminal with as show below. You can substitute clang or another compiler for gcc if you want.

```
$ gcc -o hello hello.c
$ ./hello
Hello World!
```

Now, the same program in MIPS:

```
1 .data
2 hello: .asciiz "Hello World!\n"
3
4 .text
5 main:
6     li    $v0, 4      # print string system call
7     la    $a0, hello  # load address of string to print into a0
8     syscall
9
10    li    $v0, 10     # exit syscall
11    syscall
```

The **.data** section is where you declare global variables, which includes string literals as in this case. We'll cover them in more detail later.

The **.text** section is where any code goes. Here we declare a single label `main` indicating the start of our main function.

We then put the number 4 in the `$v0` register to select the print string system call. It takes one argument, the address of the string to print, in the `$a0` register which we do on the next line. Finally, we call the system call using the `syscall` instruction. Finally we call the exit system call which takes no arguments and exits the program. Again we'll cover system calls in a later chapter, this is just an intro/overview.

Building and Running

Now that we have our hello world MIPS program, how do we run it? Well the easiest and quickest^[1] way is of course to do it on the command line, which can be done like this for `spim`:

```
$ spim -file hello.s
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
Hello World!
```

or this for MARS:

```
$ java -jar ~/Mars4_5.jar hello.s
MARS 4.5 Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

Hello World!
```

The name of your MARS jar file may be different^[2], but just use the correct name and path. For me I just keep the jar file in my home directory so I can use tilde to access it no matter where I am. You can also just copy it into your working directory (ie wherever you have your source code) so you don't have to specify a path at all. There are lots of useful command line options that you can use^[3], some of which we'll touch on later.

Running the jar directly on the command line works even in the DOS command line though I've never done it and it's probably not worth it.

Alternatively, you can just start up MARS or QtSpim like a normal GUI application and then load your source file. MARS requires you to hit "assemble" then "run", whereas QtSpim, you just hit "run".

QtSpim does let you start and load the file in one step from the command line

```
$ qtspim hello.s
```

but there is no way to simply run it with out starting the GUI, which makes sense since the whole point is to be a GUI wrapper around spim.

Conclusion

Well, there you have it, you've written and run your first MIPS program. Another few chapters and you'll have no trouble with almost anything you'd want to do in MIPS for a class or just for fun with these simulators.

[1] Starting up the MARS GUI (an old Java app) is often annoyingly slow

[2] Some schools/professors have their own versions with extra features and other improvements over the old version available on the MARS website

[3] <https://courses.missouristate.edu/KenVollmar/mars/Help/MarsHelpCommand.html>

Chapter 1: Data

In MIPS, you can declare global variables in the `.data` section.

At a minimum this is where you would declare/define any literal strings your program will be printing, since virtually every program has at least 1 or 2 of those.

When declaring something in the `.data` section, the format is

```
variable_name: .directive value(s)
```

where whitespace between the 3 is arbitrary. The possible directives are listed in the following table:

Table 1. MIPS data types

Directive	Size	C equivalent
<code>.byte</code>	1	char
<code>.half</code>	2	short
<code>.word</code>	4	int, all pointer types
<code>.float</code>	4	float
<code>.double</code>	8	double
<code>.ascii</code>	NA	char str[5] = "hello"; (no '\0')
<code>.asciiz</code>	NA	char str[] = "hello"; (includes the '\0')
<code>.space</code>	NA	typeless, uninitialized space, can be used for any type/array

As you can see it's pretty straightforward, but there are a few more details about actually using them so let's move onto some examples.

Say you wanted to convert the following simple program to MIPS:

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char name[30];
6     int age;
7     printf("What's your name and age?\n");
8     scanf("%s %d", name, &age);
9     printf("Hello %s, nice to meet you!\n", name);
10    return 0;
11 }
```

The first thing you have to remember is that when converting from a higher level language to assembly (any assembly) is that what matters is whether it's functionally the same, not that

everything is done in exactly the same way. In this instance, that means realizing that your literal strings and the name array become globals in MIPS.

```
1 .data
2 age:      .word 0 # can be initialized to anything
3
4 ask_name:  .asciiz "What's your name and age?\n"
5 hello_space: .asciiz "Hello "
6 nice_meet: .asciiz ", nice to meet you!\n"
7
8 name:      .space 30
9
10 .text
11
12 # main goes here
```

As you can see in the example, we just extract all the string literals and the character array name and declare them as MIPS globals. One thing to note is the second printf. Because it prints a variable, name, using the conversion specifier, we break the literal into pieces around that. Since there is no built-in printf function in MIPS, you have to handle printing variables yourself with the appropriate system calls.

Arrays

Declaring arrays is just an extension of declaring single variables. Obviously strings are special cases, that can be handled with `.ascii` or `.asciiz` for literals, but for other types or user inputted strings how do we do it?

Well the first way, which was demonstrated in the snippet above is to use `.space` to just declare array of the necessary byte size. Keep in mind that the size is specified in bytes not elements, so it only matches for character arrays. For arrays of ints/words, floats, doubles etc. you'd have to multiply by the `sizeof(type)`.

"But, `.space` only lets you declare uninitialized arrays, how do I do initialized ones?"

Well, it's just an extension of declaring a single variable of that type. You specify all the values, comma separated. This actually gives you another way to declare a string or a character array, though I can't really think of a reason you'd want to. You could declare a `.byte` array and list all the characters individually. See below for examples.

```
1 int a[20];
2 double b[20];
3 int c[10] = { 9,8,7,6,5,4,3,2,1,0 };
4 char d[3] = { 'a', 'b', 'c' }
```

becomes

```
1 .data
2 a:      .space 80
3 b:      .space 160
4 c:      .word 9,8,7,6,5,4,3,2,1,0
5 d:      .byte 'a', 'b', 'c'
```

Chapter 2: System Calls

We mentioned system calls (aka syscalls from now on) in chapter 0 when we were going over our "Hello World" program, but what exactly are they?

Essentially, they are the built in functions of a operating system, in this case, the simple operating system of the MIPS simulators. They provide access to all the fundamental features, like input and output to/from both the console and files, allocating memory, and exiting. That covers all the 17 syscalls supported by spim, but MARS supports many more, for things from playing MIDI sounds, to getting a random number, to creating GUI dialogs.^[4]

NOTE

Except for the MARS or SPIM specific chapters/sections, I'll be sticking to code compatible with both throughout this book, meaning we only use the first 17 syscalls, and don't get to use some of the syntactic sugar available in MARS, or any SPIM specific features either.

Table 2. SPIM supported syscalls

Name	\$v0	Arguments	Result
print integer	1	\$a0 = integer to print	
print float	2	\$f12 = float to print	
print double	3	\$f12 = double to print	
print string	4	\$a0 = address of string	
read integer	5		\$v0 = integer read
read float	6		\$f0 = float read
read double	7		\$f0 = double read
read string	8	\$a0 = address of input buffer \$a1 = buffer size	works like C's fgets
sbrk	9	\$a0 = size in bytes to allocate	\$v0 = address of allocated memory (sbrk is basically malloc but there is no free)
exit	10		program terminates
print character	11	\$a0 = character to print (ascii value)	
read character	12		\$v0 = character read
open file	13	\$a0 = address of filename \$a1 = flags \$a2 = mode	\$v0 = file descriptor (negative if error)
read from file	14	\$a0 = file descriptor \$a1 = address of input buffer \$a2 = max characters to read	\$v0 = number of characters read, 0 for end-of-file, negative for error

Name	\$v0	Arguments	Result
write to file	15	\$a0 = file descriptor \$a1 = address of output buffer \$a2 = number of characters to write	\$v0 = number of characters written, negative for error
close file	16	\$a0 = file descriptor	
exit2	17	\$a0 = termination result	program terminates, returning number in \$a0 (only meaningful when run in the terminal, ignored in GUI)

As you can see, it's just the basics. You can read or write the different types, do file I/O using calls identical to POSIX functions (open, read, write, close; see man pages), allocate memory, and exit. Even so, they're sufficient to build anything you want.

So, what does that table mean? How do these actually work?

The process is, put the number for the syscall you want in \$v0, fill in the appropriate arguments, if any, and then do the syscall:

```

1    li    $v0, 1    # 1 is print integer
2    li    $a0, 42   # takes 1 arg in a0, the number to print
3    syscall          # actually execute syscall

```

You can think of the above as `print_integer(42);`. Let's look at an actual program that uses a few more syscalls next.

Examples

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int age;
6     int height;
7     char name[50];
8     printf("What's your name? ");
9     fgets(name, 50, stdin);
10
11     printf("Hello %s", name);
12
13     printf("How old are you? ");
14     scanf("%d", &age);
15
16     printf("Enter your height in inches: ");
17     scanf("%d", &height);
18
19     printf("Your age + height = %d\n", age + height);
20
21     return 0;
22 }

```

I'm using `fgets()` instead of `scanf("%s", name)` because `fgets` works the same as the `read string` syscall (8).

```

1 .data
2
3 name:      .space 50
4
5 nameprompt: .asciiz "What's your name? "
6 hello_space: .asciiz "Hello "
7 how_old:    .asciiz "How old are you? "
8 ask_height: .asciiz "Enter your height in inches: "
9 ageplusheight: .asciiz "Your age + height = "
10
11
12 .text
13 main:
14     li    $v0, 4      # print string system call
15     la    $a0, nameprompt # load address of string to print into a0
16     syscall
17
18     li    $v0, 8      # read string
19     la    $a0, name
20     li    $a1, 50
21     syscall
22
23     li    $v0, 4

```

```

24    la    $a0, hello_space
25    syscall
26
27    la    $a0, name    # note 4 is still in $v0
28    syscall
29
30    # don't print a newline here because
31    # one will be part of name
32
33    li    $v0, 4
34    la    $a0, how_old
35    syscall
36
37    li    $v0, 5    # read integer
38    syscall
39    move $t0, $v0    # save age in t0
40
41    li    $v0, 4
42    la    $a0, ask_height
43    syscall
44
45    li    $v0, 5    # read integer
46    syscall
47    add  $t0, $t0, $v0 # t0 += height
48
49
50    li    $v0, 4
51    la    $a0, ageplusheight
52    syscall
53
54    li    $v0, 1    # print int
55    move $a0, $t0    # a0 = age + height
56    syscall
57
58    # print newline
59    li    $v0, 11    # print char
60    li    $a0, 10    # ascii value of '\n'
61    syscall
62
63
64    li    $v0, 10    # exit syscall
65    syscall

```

There are a few things to note from the example.

We don't declare global variables for age or height. We could, but there's no reason to since we have to have them in registers to do the addition anyway. So we just copy/save height to \$t0 so we can use \$v0 for 2 more syscalls, then add age to t0.

This is generally how it works. Use registers for local variables unless required to do otherwise. We'll cover more about register use when we cover the MIPS calling convention.

Another thing is when we print their name, we don't put 4 in \$v0 again because it is still/already 4 from the lines above. Unless the syscall says it writes to \$v0 you can assume it is unmodified.

Lastly, many people will declare a string `"\n"` and use `print string` to print a newline, but it's easier to just use the `print char` syscall as we do just before exiting.

[4] <https://courses.missouristate.edu/KenVollmar/mars/Help/SyscallHelp.html>

Chapter 3: Branches and Logic

We can't go much farther in our MIPS programming journey without covering branching. Almost every non-trivial program requires some logic, even if it's just a few if or if-else statements. In other words, almost every program requires branching, a way to do code a instead of code b, or to do code a only if certain conditions are met.

You already know how to do this in higher level languages, the aforementioned if statement. In assembly it's more complicated. Your only tool is the ability to jump to a label on another line based on the result of various comparisons. The relevant instructions are listed in the following table:

Table 3. MIPS branching related instructions (and pseudoinstructions)

Name	Opcode	Format	Operation
Branch On Equal	beq	beq rs, rt, label	if (rs == rt) goto label
Branch On Not Equal	bne	bne rs, rt, label	if (rs != rt) goto label
Branch Less Than	blt	blt rs, rt, label	if (rs < rt) goto label
Branch Greater Than	bgt	bgt rs, rt, label	if (rs > rt) goto label
Branch Less Than Or Equal	ble	ble rs, rt, label	if (rs ≤ rt) goto label
Branch Greater Than Or Equal	bge	bge rs, rt, label	if (rs ≥ rt) goto label
Set Less Than	slt	slt rd, rs, rt	rd = (rs < rt) ? 1 : 0
Set Less Than Immediate	slti	slt rd, rs, imm	rd = (rs < imm) ? 1 : 0
Set Less Than Immediate Unsigned	sltiu	slt rd, rs, imm	rd = (rs < imm) ? 1 : 0
Set Less Than Unsigned	sltu	sltu rd, rs, imm	rd = (rs < imm) ? 1 : 0

You can see the same information and more (like which ones are pseudoinstructions) on the MIPS greensheet.^[5]

There are additional pseudoinstructions in the form of beq/bne/blt/bgt/ble/bge + 'z' which are just shortcuts to compare a register against 0, ie the 0 register.

So the following:

```
beq    $t0, $0, label
bne    $t1, $0, label
blt    $t2, $0, label
```

would be equivalent to:

```
beqz   $t0, label
bnez   $t1, label
bltz   $t2, label
```


Note `$0` is the same as `zero` and is just the hard coded 0 register. I'll cover registers in more detail in the chapter on functions and the calling conventions.

One final thing is that labels have the same naming requirements as C variables and functions. They must start with a letter or underscore and the rest can be letters, underscores, and digits.

Practice

The rest of this chapter will be going over many examples, looking at snippets of code in C and translating them to MIPS.

Basics

Let's start with the most basic if statement. The code in and after the if statement is arbitrary.

```
1  if (a > 0) {
2      a++;
3  }
4  a *= 2;
```

Now in MIPS, let's assume that `a` is in `$t0`. The translation would look like this:

```
1  ble  $t0, $0, less_eq_0    # if (a <= 0) goto less_eq_0
2  addi $t0, $t0, 1           # a++
3  less_eq_0:
4  sll  $t0, $t0, 1           # a *= 2 (shifting left by n is multiplying by 2^n)
```

There are a few things to note in this example. The first is that in assembly we test for the opposite of what was in the if statement. This will always be the case when jumping forward because (if we want to keep the same order of code) we can only jump *over* a block of code, whereas in C we fall into the block if the condition is true. In the process of mentally compiling a bit of C to assembly, it can be helpful to some to change to jump based logic first. For example the previous C would become:

```
1  if (a <= 0)
2      goto less_eq_0;
3  a++;
4  less_eq_0:
5  a *= 2;
```

This is obviously still valid C but matches the branching behavior of assembly exactly. You can see I put comments for the equivalent C code in my assembly; it helps with readability to comment every line or group of lines that way.

The second thing to notice is how we handled the multiplication. This has nothing to do with branching but is something we'll touch on multiple times throughout the book. Your job when

acting as a human compiler is to match the *behavior*. You are under no obligation to match the structure or operations of the higher level code exactly (unless your professor stupidly forces you to).

Given that, it is in your best interest to change and rearrange things in order to simplify the assembly as much as possible to make your life easier. Generally speaking, this also tends to result in more performant code, since using fewer instructions and fewer branches (the most common outcomes) saves execution time.

In this case, the standard `mult` instruction (from the green sheet) would have required 3 instructions, and even the `mul` instruction (that does seem to be supported everywhere but is not on the green sheet) would take 2:

```
1    li    $t1, 2
2    mult  $t0, $t1
3    mflo  $t0          # a *= 2
4
5    # or
6
7    li    $t1, 2
8    mul   $t0, $t0, $t1  # a *= 2
```

Given that, when multiplying or dividing by a power of 2 it's common practice, and just common sense, to use `sll` or `sra`. This is true in all assembly languages because multiplication (and division) is a relatively costly operation so using shifts when you can saves performance even if you didn't actually save instructions.

Ok, let's look at an if-else example. Again the actual code is arbitrary and we're assuming a and b are in `$t0` and `$t1` respectively

```
1    if (a > 0) {
2        b = 100;
3    } else {
4        b -= 50;
5    }
```

You could do it something like these two ways

```

1    bgt    $t0, $0, greater_0    # if (a > 0) goto greater_0
2    addi   $t1, $t1, -50         # b -= 50
3    j      less_eq_0
4 greater_0:
5    li     $t1, 100              # b = 100
6 less_eq_0:
7
8    # or
9
10   ble    $t0, $0, less_eq0     $ if (a <= 0) goto less_eq_0
11   li     $t1, 100              # b = 100
12   j      greater_0
13 less_eq_0:
14   addi   $t1, $t1, -50         # b -= 50
15 greater_0:

```

You can see how the first swaps the order of the actual code which keeps the actual conditions the same as in C, while the second does what we discussed before and inverts the condition in order keep the the blocks in the same order. In both cases, an extra unconditional branch and label is necessary so we don't fall through the else case. This is inefficient and wasteful, not to mention complicates the code unnecessarily. Remember how our job is to match the behavior, not the exact structure? Imagine how we could rewrite it in C to simplify it:

```

1    b -= 50;
2    if (a > 0) {
3        b = 100;
4    }

```

which becomes

```

1    addi   $t1, $t1, -50         # b -= 50;
2    ble    $t0, $0, less_eq_0   # if (a <= 0) goto less_eq_0
3    li     $t1, 100              # b = 100
4 less_eq_0:

```

That is just one simple example of rearranging code to make your life easier. In this case, we are taking advantage of what the code is doing to make a default path or default case. Obviously, because of the nature of the code subtracting 50 has to be the default because just setting b to 100 loses the original value in case we were supposed to subtract 50 instead. In cases where you can't avoid destructive changes (like where the condition and the code are using/modifying the same variable), you can use a temporary variable; i.e. copy the value into a spare register. You still save yourself an unnecessary jump and label.

Compound Conditions

These first 2 examples have been based on simple conditions, but what if you have compound

conditions? How does that work with branch operations that only test a single condition? As you might expect, you have to break things down to match the logic using the operations you have.

Let's look at **and** first. Variables a, b, and c are in t0, t1, and t2.

```
1  if (a > 10 && a < b) {
2      c += 20;
3  }
4  b &= 0xFF;
```

So what's our first step? Well, just like previous examples we need to test for the opposite when we switch to assembly, so we need the equivalent of

```
1  if (!(a > 10 && a < b))
2      goto no_add20;
3  c += 20;
4 no_add20:
5  b &= 0xFF;
```

Well, that didn't help us much, we still don't know how to handle that compound condition. In fact we've just made it more complicated. If only there were a way to convert it to **or** instead of **and**. Why would we want that? Because, while both **and** and **or** in C allow for short circuit evaluation (where the result of the whole expression is known early and the rest of expression is not evaluated), with **or**, it short circuits on success while **and** short circuits on failure. What does that mean? It means that with **or**, the whole expression is true the second a single true term is found, while with **and** the whole expression is false the second a single false term is found.

Let's look at the following code to demonstrate:

```
1  if (a || b || c) {
2      something;
3  }
4
5  // What does this actually look like if we rewrote it to show what it's
6  // actually doing with short circuit evaluation?
7
8  if (a) goto do_something;
9  if (b) goto do_something;
10 if (c) goto do_something;
11 goto dont_do_something;
12
13 do_something:
14     something;
15
16 dont_do_something:
17
18 // You can see how the first success is all you need:
19 // Compare that with and below
```

```

20
21     if (a && b && c) {
22         something;
23     }
24
25     if (a) {
26         if (b) {
27             if (c) {
28                 something;
29             }
30         }
31     }
32     // which in jump form is
33
34     if (a)
35         goto a_true;
36     goto failure;
37 a_true:
38     if (b)
39         goto b_true;
40     goto failure;
41
42 b_true:
43     if (c)
44         goto c_true;
45     goto failure;
46
47 c_true:
48     something;
49 failure:
50
51     // Man that's ugly and overcomplicated and hard to read
52     // But what if we did this instead:
53
54     if (!a) goto dont_do_something;
55     if (!b) goto dont_do_something;
56     if (!c) goto dont_do_something;
57
58     something;
59
60 dont_do_something:
61
62     // Clearly you need all successes for and. In other words
63     // to do and directly, you need state, knowledge of past
64     // successes. But what about that second translation of and?
65     // It looks a lot like or?

```

You're exactly right. That final translation of **and** is exactly like **or**.

It takes advantage of De Morgan's laws.^[6] For those of you who haven't taken a Digital Logic course (or have forgotten), De Morgan's laws are 2 equivalencies, a way to change an **or** to an **and**, and

vice versa.

They are (in C notation):

```
!(A || B) == !A && !B
```

```
!(A && B) == !A || !B
```

Essentially you can think of it as splitting the not across the terms and changing the logical operation. The law works for arbitrary numbers of terms, not just 2:

```
(A && B && C)
is really
((A && B) && C)
so when you apply De Morgan's Law recursively you get:
!((A && B) && C) == !(A && B) || !C == !A || !B || !C
```

Let's apply the law to our current example. Of course the negation of comparisons is just covering the rest of the number line so:

```
1  if (a <= 10 || a >= b)
2      goto no_add20;
3  c += 20;
4 no_add20:
5  b &= 0xFF;
```

which turns into:

```
1  li    $t9, 10
2  ble   $t0, $t9, no_add20      # if (a <= 10) goto no_add20
3  bge   $t0, $t1, no_add20      # if (a >= b) goto no_add20
4
5  addi   $t2, $t2, 20           # c += 20
6 no_add20:
7  andi   $t1, $t1, 0xFF         # b &= 0xFF
```

See how that works? **Or**'s do not need to remember state. Just the fact that you reached a line in a multi-term **or** expression means the previous checks were false, otherwise you'd have jumped. If you tried to emulate the same thing with an **and**, as you saw in the larger snippet above, you'd need a bunch of extra labels and jumps for each term.

What about mixed compound statements?

```
1  if (a > 10 || c > 100 && b >= c)
2      printf("true\n");
3
4  b |= 0xAA;
```

Well, the first thing to remember is that `&&` has a higher priority than `||`, which is why most compilers these days will give a warning for the above code about putting parenthesis around the `&&` expression to show you meant it (even though it's completely legal as is).

So with that in mind, let's change it to jump format to better see what we need to do. While we're at it, let's apply De Morgan's law to the `&&`.

```
1  if (a > 10)
2      goto do_true;
3  if (c <= 100)
4      goto done_if;
5  if (b < c)
6      goto done_if;
7 do_true:
8  printf("true\n");
9
10 done_if:
11  b |= 0xAA;
```

This one is trickier because we don't flip the initial expression like normal. Instead of jumping *over* the body which would require testing for the opposite, we jump to the true case. We do this because we don't want to have multiple print statements and it lets us fall through the following conditions. We would need multiple print statements because failure for the first expression is *not* failure for the entire expression. Here's how it would look otherwise:

```
1  if (a <= 10)
2      goto check_and;
3  printf("true\n");
4  goto done_if;
5 check_and:
6  if (c <= 100)
7      goto done_if;
8  if (b < c)
9      goto done_if;
10
11  printf("true\n");
12
13 done_if:
14  b |= 0xAA;
```

That is harder to read and has both an extra print and an extra jump.

So let's convert the better version to MIPS (`a,b,c = $t0, $t1, $t2`):

```

1 .data
2 true_str: .asciiz "true\n"
3
4 .text
5     li    $t8, 10    # just get the necessary literals in some unused regs
6     li    $t9, 100
7
8     bgt    $t0, $t8, do_true    # if (a > 10) goto do_true
9     ble    $t2, $t9, done_if    # if (c <= 100) goto done_if
10    blt    $t1, $t2, done_if    # if (b < c) goto done_if
11
12 do_true:
13     li    $v0, 4        # print string
14     la    $a0, true_str    # address of str in a0
15     syscall
16
17 done_if:
18     ori    $t1, $t1, 0xAA    # b |= 0xAA

```

If-Else Chain

Ok, let's look at a larger example. Let's say you're trying to determine a student's letter grade based on their score. We're going to need a chain of if-else-if's to handle all cases. Assume `score` is declared and set somewhere before.

```

1     char letter_grade;
2     if (score >= 90) {
3         letter_grade = 'A';
4     } else if (score >= 80) {
5         letter_grade = 'B';
6     } else if (score >= 70) {
7         letter_grade = 'C';
8     } else if (score >= 60) {
9         letter_grade = 'D';
10    } else {
11        letter_grade = 'F';
12    }
13
14    printf("You got a %c\n", letter_grade);
15 }

```

With chains like these, you following everything we've learned before, it comes out looking like this (assuming `score` is `$t0` and `letter_grade` is `$t1`):


```

1 .data
2 grade_str: .asciiz "You got a "
3
4 .text
5     li    $t1, 70    # letter_grade default to 'F' ascii value
6
7     li    $t2, 90
8     blt   $t0, $t2, not_a    # if (score < 90) goto not_a
9     li    $t1, 65      # leter_grade = 'A'
10    j     grade_done
11
12 not_a:
13     li    $t2, 80
14     blt   $t0, $t2, not_b    # if (score < 80) goto not_b
15     li    $t1, 66      # leter_grade = 'B'
16     j     grade_done
17
18 not_b:
19     li    $t2, 70
20     blt   $t0, $t2, not_c    # if (score < 70) goto not_c
21     li    $t1, 67      # leter_grade = 'C'
22     j     grade_done
23
24 not_c:
25     li    $t2, 60
26     blt   $t0, $t2, grade_done    # if (score < 60) goto grade_done
27     li    $t1, 68      # leter_grade = 'D'
28
29 grade_done:
30     li    $v0, 4        # print str
31     la    $a0, grade_str
32     syscall
33
34     li    $v0, 11       # print character
35     move  $a0, $t1      # char to print
36     syscall
37
38     move  $a0, 10       # print '\n'
39     syscall

```

You can see how we set a default value and then test for the opposite of each condition to jump to the next test, until we get one that fails (aka was true in the original C condition) and set the appropriate grade.

You can arrange chains like this in either direction, it doesn't have to match the order of the C code. As long as it works the same, do whatever makes the code simpler and more sensible to you.

Conclusion

Branching and logic and learning to translate from higher level code to assembly is something that just takes a lot of practice but eventually it'll become second nature. We'll get more practice in the chapter on looping which naturally also involves branching.

One final note, there's really no reason to use the `slt` family of opcodes *unless* your professor requires it, ie he says you can't use pseudoinstructions so you're left with `beq`, `bne`, `j` and the `slt` ops. I'll show how you can code without using pseudoinstructions in a later chapter.

[5] https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf

[6] https://en.wikipedia.org/wiki/De_Morgan%27s_laws

Chapter 4: Loops

"Insanity is doing the same thing over and over again and expecting different results."

— Unknown, Often misattributed to Albert Einstein

Before we get into the MIPS, I want to cover something that may be obvious to some but may have never occurred to others. Any loop structure can be converted to any other (possibly with the addition of an if statement). So a for can be written as a while and vice versa. Even a do-while can be written as a for or while loop. Let's look at some equivalencies.

```
1  for (int i=0; i<a; i++) {
2      do_something;
3  }
4
5  int i = 0;
6  while (i < a) {
7      do_something;
8      i++;
9  }
10
11 int i = 0;
12 if (i < a) {
13     do {
14         do_something;
15         i++;
16     } while (i < a);
17 }
18 // you could also have an if (i >= a) goto loop_done; to jump over do-while
```

I think in general, when writing assembly, it can help to think more in terms of while or do-while rather than for because the former more resemble what the assembly looks like in terms of what goes where. So just like in the last chapter, we would think of the if-else statements in "jump-form" or "branch-form", we can do the same here, converting for to while in our head as an intermediary step before going to assembly.

Speaking of "jump-form", lets apply it to the loop above:

```

1   int i=0;
2   if (i >= a)
3       goto done_loop;
4 loop:
5   do_something;
6   i++
7   if (i < a)
8       goto loop;
9
10 done_loop:

```

You can see how that starts to look more like assembly. Another thing to note is that unlike with if statements where we test for the opposite to jump over the block of code, when you're doing the test for a loop at the bottom, like a do-while loop, it is unchanged from C, because you are jumping to continue the loop. If you put the test at the top it becomes inverted, and you put an unconditional jump at the bottom:

```

1   int i=0;
2 loop:
3   if (i >= a)
4       goto done_loop;
5   do_something;
6   i++
7   goto loop;
8
9 done_loop:

```

In general it's better to test at the bottom, both because it's closer to C with the matching condition, and because, when you know the loop is going to execute at least once, it requires only one jump + label, rather than 2 because you can forgo the the initial if check:

```

1   for (int i=0; i<10; i++)
2       do_something;
3
4   // becomes
5
6   int i=0;
7 loop:
8   do_something;
9   i++
10  if (i < a)
11      goto loop;

```

Ok, now that we've got the theory and structure out of the way, let's try doing a simple one in MIPS.

```

1  int sum = 0;
2  for (int i=0; i<100; i++) {
3      sum += i;
4  }

```

That's about as basic as it gets, add up 1 to 99.

```

1  li    $t0, 0    # sum = 0
2  li    $t1, 1    # i = 1  we can start at 1 because obviously adding 0 is
pointless
3  li    $t2, 100
4 loop:
5  addi  $t0, $t0, $t1    # sum += i
6  addi  $t1, $t1, 1      # i++
7  blt   $t1, $t2, loop   # while (i < 100)

```

Ok I don't think there's much point in doing any more without getting to what loops are most often used for, looping through data structures, most commonly arrays.

Looping Through Arrays

Looping and arrays go together for obvious reasons. An array a sequence of variables of the same type, almost always related in some way. Naturally you want to operate on them all together in various ways; sorting, searching, accumulating, etc. So in this section we'll cover looping through arrays in various ways, and dealing with multi-dimentional arrays.

1D Arrays

Let's pretend there's an array `int numbers[10];` filled with 10 random numbers.

```

1  int total = 0;
2  for (int i=0; i<10; i++) {
3      total += numbers[i];
4  }

```

There are several ways to do this. The first is the most literal translation.

```

1    li    $t0, 0    # total = 0
2    li    $t1, 0    # i = 0
3    la    $t2, numbers    # t2 = numbers
4    li    $t3, 10
5 sum_loop:
6    sll   $t4, $t1, 2    # t4 = i*sizeof(int) == i*4
7    add   $t4, $t4, $t2  # t4 = &numbers[i]
8    lw    $t4, 0($t4)    # t4 = numbers[i]
9    add   $t0, $t0, $t4  # total += numbers[i]
10
11    addi  $t1, $t1, 1    # i++
12    blt   $t1, $t3, sum_loop    # while (i < 10)

```

So we initialize the relevant variables before hand (numbers and 10 could be set every iteration but that's less efficient). Now what's with the $i*4$? We already discussed using shifts to multiply and divide by powers of 2 in a previous chapter, but here we're doing something that higher level languages do automatically for you every time you do an array access. When you access the i 'th element, under the hood it is multiplying i by the size of the type of the array and adding that number of bytes to the base address and then loading the element located there.

If you're unfamiliar with the C syntax in the comments, `&` means "address of", so `t4` is being set to the address of the i 'th element. Actually that C syntax is redundant because the `&` counteracts the brackets. In C adding a number to a pointer does pointer math (ie it multiplies by the size of the items as discussed above). This means that these 2 expressions are equivalent:

```
&numbers[i] == numbers + i
```

also

```
&numbers[0] == numbers
```

The reason I use the left form in C/C++ even when I can use the right is it makes it more explicit and obvious that I'm getting the address of an element of an array. If you were scanning the code quickly and saw the expression on the right, you might not realize that's an address at all, it could just be some mathematical expression (though the array name would hopefully clue you in if it was picked well).

Anyway, back to the MIPS code. After we get the address of the element we want, we have to load it. Since it's an array of words (aka 4 byte ints) we can use `lw`.

Finally we add that value to total, increment i , and perform the loop check.

Now, I said at the beginning that this was the most literal, direct translation (not counting the restructuring to a do-while form). It is not however my preferred form because it's not the simplest or shortest.

Rather than calculate the element address every iteration, why not just keep a pointer to the current element and iterate through the array with it? In C what we're doing is this:

```

1  int* p = &numbers[0];
2  int i = 0, total = 0;
3  do {
4      total += *p;
5      i++;
6      p++;
7  } while (i < 10);

```

In other words, we set `p` to point at the first element and then increment it every step to keep it pointing at `numbers[i]`. Again, all mathematical operations on pointers in C deal in increments of the byte size of the type, so `p++` is really adding `1*sizeof(int)`.

```

1  li    $t0, 0      # total = 0
2  li    $t1, 0      # i = 0
3  la    $t2, numbers # p = numbers
4  li    $t3, 10
5 sum_loop:
6  lw    $t4, 0($t2)  # t4 = *p
7  add   $t0, $t0, $t4 # total += *p
8
9  addi  $t1, $t1, 1  # i++
10 addi  $t2, $t2, 4   # p++ ie p += sizeof(int)
11 blt   $t1, $t3, sum_loop # while (i < 10)

```

Now, that may not look much better, we only saved 1 instruction total, and if we were looping through a string (aka an array of characters, `sizeof(char) == 1`) we wouldn't have saved any. However, imagine if we weren't using `sll` to do the multiply but `mul`. That would take 3 instructions, not 1. Even `mul` would take 2. And we'd have to use one of those if we were iterating through an array of structures with a size that wasn't a power of 2.

But there is one more variant that you can use that can save a few more instructions. Instead of using `i` and `i<10` to control the loop, use `p` and the address just past the end of the array. In C it would be this:

```

1  int* p = &numbers[0];
2  int* end = &numbers[10];
3  int total = 0;
4  do {
5      total += *p;
6      p++;
7  } while (p < end);

```

You could also use `!=` instead of `<`. This is similar to using the `.end()` method on many C++ data structures when using iterators.

```

1    li    $t0, 0      # total = 0
2    la    $t2, numbers # p = numbers
3    addi   $t3, $t2, 40 # end = &numbers[10] = numbers + 10*sizeof(int)
4 sum_loop:
5    lw     $t4, 0($t2)  # t4 = *p
6    add    $t0, $t0, $t4 # total += *p
7
8    addi   $t2, $t2, 4   # p++ ie p += sizeof(int)
9    blt    $t2, $t3, sum_loop # while (p < end)

```

So we dropped from 10 to 7 and even more if we had had to do mul or mult originally. And this was just for a 1D array. Imagine if you had 2 or 3 indices you had to use to calculate the correct offset. That's in the next section.

2D Arrays

The first thing to understand is what's really happening when you declare a 2D array in C. The contents of a 2D array are tightly packed, in row-major order, meaning that all the elements from the first row are followed by all the elements of the second row and so on. What this means is that a 2D array is equivalent to a 1D array with rows*cols elements in the same order:

```

1    // The memory of these two arrays are identical
2    int array2d[2][4] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 } };
3    int array1d[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };

```

See the code example 2d_arrays.c for more details.

What this means is that when we declare a 2D array, it's basically a 1D array with the size equal to the rows * columns. Also when we loop through a 2D array, we can often treat it like a 1D array with a single loop. So everything that we learned before applies.

Let's do an example.

```

1    for (int i=0; i<rows; i++) {
2        for (int j=0; j<cols; ++j) {
3            array[i][j] = i + j;
4        }
5    }
6
7    // becomes
8
9    int r, c;
10   for (int i=0; i<rows*cols; i++) {
11       r = i / cols;
12       c = i % cols;
13       array[i] = r + c;
14   }

```


So assuming rows and cols are in a0 and a1 (and nonzero), it'd look like this:

```
1    la    $t0, array    # p = &array[0]
2    li    $t1, 0        # i = 0
3    mult  $a0, $a1       # a0 = rows, a1 = cols
4    mflo  $t2           # t2 = rows * cols
5 loop:
6    div   $t1, $a1       # r = i / cols
7    mflo  $t3           # r = i / cols
8    mfhi  $t4           # c = i % cols
9    add   $t3, $t3, $t4   # t3 = r + c
10
11   sw    $t3, 0($t0)     # array[i] = *p = r + c
12
13   addi  $t1, $t1, 1     # i++
14   addi  $t0, $t0, 4     # p++ (keep pointer in sync with i, aka p = &array[i])
15   blt   $t1, $t2, loop  # while (i < rows*cols)
```

You might ask if it's worth it to convert it to a single loop when you still need the original i and j as if you were doing nested loops. Generally, it is much nicer to avoid nested loops in assembly if you can. There are many cases when you get the best of both worlds though. If you're doing a clear for example, setting the entire array to a single value, there's no need to calculate the row and column like we did here. I only picked this example to show how you could get them back if you needed them.

For comparison here's the nested translation (while still taking advantage of the 1D arrangement of memory and pointer iterators):

```
1    la    $t0, array    # p = &array[0]
2    li    $t1, 0        # i = 0
3 looprows:
4    li    $t2, 0        # j = 0
5 loopcols:
6    add   $t3, $t1, $t2   # t3 = i + j
7    sw    $t3, 0($t0)     # array[i][j] = *p = i + j
8
9    addi  $t2, $t2, 1     # j++
10   addi  $t0, $t0, 4     # p++ (keep pointer in sync with i and j, aka p =
    &array[i][j])
11   blt   $t2, $a1, loopcols # while (j < cols)
12
13   addi  $t1, $t1, 1     # i++
14   blt   $t1, $a0, looprows # while (i < rows)
```

It's a bit shorter but again, how much are those extra labels and branching worth? For me, this one's a toss up. On the other hand either of the last 2 versions are better than the literal translation imo:

```

1    la    $t0, array    # p = &array[0]
2    li    $t1, 0        # i = 0
3 looprows:
4    li    $t2, 0        # j = 0
5 loopcols:
6    add    $t3, $t1, $t2    # t3 = i + j
7
8    # need to calculate the byte offset of element array[i][j]
9    mult   $t1, $a1
10   mflo   $t4            # i * cols
11
12   add    $t4, $t4, $t2    # t4 = i * cols + j
13
14   sll    $t4, $t4, 2      # t4 = (i * cols + j) * sizeof(int)
15   add    $t4, $t4, $t0    # t4 = &array[i][j] (calculated as array + (i*cols +
    j)*4)
16
17   sw     $t3, 0($t4)      # array[i][j] = i + j
18
19   addi   $t2, $t2, 1      # j++
20   blt    $t2, $a1, loopcols # while (j < cols)
21
22   addi   $t1, $t1, 1      # i++
23   blt    $t1, $a0, looprows # while (i < rows)

```

That chunk in the middle calculating the offset of every element? Not only is it far slower than just iterating the pointer through the array, but you can imagine how much worse it would be for a 3D array with 3 nested loops.

Conclusion

Hopefully after those examples you have a more solid understanding of looping in MIPS and how to transform various loops and array accesses into the form that makes your life the easiest. There is more we could cover here, like looping through a linked list, but I think that's beyond the scope of what we've covered so far. Perhaps in a later chapter.

Chapter 5: Functions and the MIPS Calling Convention

While I'm sure everyone here probably knows what functions are, and we'll cover them in assembly shortly, you might be wondering what a "Calling Convention" is. In short, it is an agreement between the caller and callee about how to treat/use certain registers. We'll get to the why and how later.

Functions

In assembly, a function is just a label with a return instruction associated with it; because this is far more ambiguous than a function in a higher level language, it is good practice to only have a single return instruction associated with a function.^[7] A comment above the label is also helpful. Together those help you quickly see the start and end of the function.

```
1 void func1() {}
```

would be

```
1 # void func1()
2 func1:
3     # body goes here
4     jr     $ra
```

As you can see my policy is to put a single line comment of the C prototype above label.

But how do you call a function in assembly? You use the opcode `jal func_label`. Let's change the hello world program from chapter 0 to call a function:

```

1 .data
2 hello:  .ascii "Hello World!\n"
3
4 .text
5 main:
6     jal  hello_world
7
8     li   $v0, 10      # exit syscall
9     syscall
10
11
12 # void hello_world()
13 hello_world:
14     li   $v0, 4        # print string system call
15     la   $a0, hello    # load address of string to print into a0
16     syscall
17
18     jr   $ra

```

The op `jal` stands for "jump and link". What it actually does is save the address of the next instruction to `$ra` and then do an unconditional jump to the function label. So you could achieve the same results with the following:

```

jal    func

# is equivalent to

la     $ra, next_instr
j      func
next_instr:

```

That would get tiring and ugly fast though, having to come up with unique labels for the next instruction every time. You also might be confused about why the greensheet says `jal` saves PC+8 in `$ra` instead of PC+4. The reason for that is that MIPS technically has delayed branching, i.e. a single instruction after every branch instruction is executed before the jump actually happens. So `jal` adds 8 instead of 4 to account for that extra instruction delay. However, every simulator we've mentioned does non-delayed branching by default so you can ignore it.

The Convention

We've gone as far as we can without starting to talk about registers and their purposes in functions. You can think about registers as variables^[8] that are part of the CPU. In this case, since we're dealing with a 32-bit MIPS architecture, they are 4 byte/word variables. Since they're part of the CPU, they exist for the life of the program and the whole program shares the same registers.

But how does that work? If all parts of the program use the same 32 registers, how does one function not stomp all over what another was doing when they use them? In fact, how do functions

communicate at all? How do they pass arguments or return results? All these questions are solved by deciding on a "Calling Convention". It's different for different architectures and even different operating systems on the same architecture. This is because different architectures have different numbers of registers, and some registers like `$ra` have hardcoded uses. The op `jal` modifies `$ra` and `$0` is a constant 0 and there's no way to change either of those facts. That still leaves a lot of flexibility about designing a calling convention. While they mostly match, you'll find several variations of MIPS calling conventions online. They usually differ in how they setup a stack frame. The convention covered in this chapter is consistent with, and sufficient for almost every college course I've ever heard of.

Regardless, what matters, is that the calling convention works by setting rules (and guidelines) for register use, and when/how to use the stack.

If you're unfamiliar with the runtime stack, it's exactly what it sounds like. It's a First-In-Last-Out (FILO) data structure that you can use to store smaller values in a program. It grows in a negative direction, so to allocate 12 bytes, you would subtract 12 from the stack pointer (in MIPS that's `$sp`).

MIPS specifically, designates certain registers to be used for passing arguments (at least the first 4), others for return values, and others as misc temporary saved respectively. The rest are special use registers like `$ra`.

The quickest way to summarize is to look at the table on the greensheet which is reproduced (with some modifications) below:

Table 4. MIPS Registers and Uses

Name	Number	Use	Preserved Across a Call
<code>\$zero</code>	0	Constant 0	N.A.
<code>\$at</code>	1	Assembler Temporary (used to expand pseudo-ops)	No
<code>\$v0-\$v1</code>	2-3	Function Results and Expression Evaluation	No
<code>\$a0-\$a3</code>	4-7	Arguments	No
<code>\$t0-\$t7</code>	8-15	Temporaries	No
<code>\$s0-\$s7</code>	16-23	Saved Temporaries	Yes
<code>\$t8-\$t9</code>	24-25	Temporaries	No
<code>\$k0-\$k1</code>	26-27	Reserved for OS Kernel	No
<code>\$gp</code>	28	Global Pointer	Yes
<code>\$sp</code>	29	Stack Pointer	Yes
<code>\$fp</code> (or <code>\$s8</code>)	30	Frame Pointer if necessary or can be another saved reg	Yes
<code>\$ra</code>	31	Return Address	No

To summarize, you have 16 registers that can be used anytime for temporary values, though some have special uses too (the v, a and t registers). You have 8 s registers that have to be saved on the

stack if you use them, plus `$ra` as well. The `$zero` register is obviously a special case.

The `$sp` register is technically preserved but not in the same way. Basically what you allocate (subtract) you have to deallocate (add) before returning from a function, thus preserving the original value.

You can ignore `$at`, `$k0-$k1`, `$gp` and most of the time `$fp` too. In 5 years of tutoring I've helped students with MIPS from at least 2 dozen different colleges and I think a professor I've only seen a professor force his students to use `$fp` or pass more than 4 arguments twice. I've actually seen^[9] register 30 referred to as `$s8` rather than, or in addition to, `$fp` which shows you how rarely it's actually used as a frame pointer.

Basic example

Let's start with something simple that doesn't use the stack.

```
int hello_name_number(char* name, int number)
{
    printf("Hello %s!\n", name);
    return number*10;
}
```

According to the convention that becomes:

```
.data
hello_space: .asciiz "Hello "
exclaim_nl:  .asciiz "!\n"

.text
#int hello_name_number(char* name, int number)
hello_name_number:
    move    $t0, $a0    # save name in t0 since we need a0 for the syscall

    li      $v0, 4       # print string
    la      $a0, hello_space
    syscall

    move    $a0, $t0     # print name (v0 is still 4)
    syscall

    la      $a0, exclaim_nl
    syscall

    addi    $v0, $a1, 10 # return number+10
    jr      $ra
```

Some things to note, syscalls are not function calls, so we can "save" `$a0` in a `t` register and know

that it'll still be there when the syscall is done. In the same way, we know that `$v0` is still the same so we don't have to keep setting it to 4 for print string. Lastly, to return a value, we just make sure that value is in `$v0` before returning.

Using the Stack

Ok first let's establish the rules on when you *have* to use the stack (You can always use it for arbitrary local variables, like a local array for example but generally don't if you don't have a good reason).

1. You call another function, ie you're a non-leaf function.

This means you have to save `$ra` on the stack at the very least, otherwise when you do your `jr $ra` you'd jump back into yourself (right after the last `jal` instruction). This does not apply to main because you don't/shouldn't return from main, you should call the `exit` (or `exit2`) syscall (10 or 17).

2. You need to save values across a function call (automatically includes reason 1).

This is fairly common for non-trivial functions. Obvious examples are calling a function in a loop or loops (you'd have to preserve the iterator(s), and many recursive functions.

3. You run out of temporary registers and overflow into the s registers.

This is very rare. The most common reason this "happens" is people forget they have 10 t registers instead of 8 like s registers and even if they remember that they forget they can also use the a and v registers for temporaries. 16 is more than enough to handle pretty much any function because you rarely need 17 discrete values at the same time.

Ok let's look at an example for the first two. Any example for the last rule would be prohibitively large/complicated.

```
1 int non_leaf()  
2 {  
3     func1();  
4     return 42  
5 }
```

This just calls the empty function discussed at the top of this chapter.

```

1 #int non_leaf()
2 non_leaf:
3     addi    $sp, $sp, -4 # space to save 1 register, $ra
4     sw      $ra, 0($sp)
5
6     jal     func1
7
8     li      $v0, 42      # return 42
9
10    lw      $ra, 0($sp)  # restore original $ra
11    addi    $sp, $sp, 4  # pop the stack
12    jr      $ra

```

The bit of code at the top and bottom of the function are called the prologue and the epilogue for obvious reasons. You can see how we allocate 4 bytes on the stack by subtracting 4 (I add a negative rather than subtract, both because I can copy-paste the line with a single character change for the epilogue). Then we store the current `$ra` in that space at the new top of the stack. Then before we exit we have to load it back and pop the stack.

If we didn't save and restore `$ra` we would jump to line 7 when we do our `jr $ra` and then we'd be in an infinite loop.

Next we have the second case, where we need to preserve regular local values across a function call.

```

1 void print_letters(char letter, int count)
2 {
3     for (int i=0; i<count; i++) {
4         putchar(letter);
5     }
6     putchar('\n');
7 }
8
9 int save_vals()
10 {
11     for (int i=0; i<10; i++) {
12         print_letters('A'+i, i+1);
13     }
14     return 8;
15 }

```

That becomes this in mips:


```

1 #void print_letters(char letter, int count)
2 print_letters:
3     ble     $a1, $0, exit_pl
4     li      $v0, 11          # print character
5 pl_loop:
6     syscall
7     addi    $a1, $a1, -1     # count--
8     bgt     $a1, $0, pl_loop # while (count > 0)
9
10    li      $a0, 10          # '\n'
11    syscall
12
13 exit_pl:
14    jr      $ra
15
16
17 #int save_vals()
18 save_vals:
19    addi     $sp, $sp, -12
20    sw       $ra, 0($sp)
21    sw       $s0, 4($sp)
22    sw       $s1, 8($sp)
23
24    li       $s0, 0 # i = 0
25    li       $s1, 10
26 sv_loop:
27    addi     $a0, $s0, 65    # i + 'A'
28    addi     $a1, $s0, 1     # i + 1
29    jal      print_letters
30
31    addi     $s0, $s0, 1     # i++
32    blt      $s0, $s1, sv_loop # while (i < 10)
33
34    lw       $ra, 0($sp)
35    lw       $s0, 4($sp)
36    lw       $s1, 8($sp)
37    addi     $sp, $sp, 12
38    jr      $ra

```

Notice that for `print_letters`, we not only convert the loop to a do-while, but we also use the parameter `count` as the iterator to count down to 0. It saves us an instruction initializing an `i`.

Secondly, for `save_vals`, we save not only `$ra` because we call another function, but also two registers to save `i` and our stopping point. The second is not actually necessary. Because it's a constant, we could just load 10 into a register right before the check every iteration of the loop. Which is better depends on several factors, like how long/complex the loop is, how many times it executes, and of course personal preference.

Recursive Functions

Let's do a classic recursive function, the fibonacci sequence.

```
1 int fib(int n)
2 {
3     if (n <= 1)
4         return n;
5
6     return fib(n-2) + fib(n-1);
7 }
```

You can see how at the very least, we'll have to save `$ra` and `n` because we need the original even after the first recursive `fib` call. It's not as obvious but we'll also have to save the return value of the first call so we'll still have it to do the addition after the second. You might think this would require using two `s` regs, but does it? Let's see...

```
1 #int fib(int n)
2 fib:
3     addi    $sp, $sp, -8
4     sw      $ra, 0($sp)
5     sw      $s0, 4($sp)
6
7     move    $v0, $a0      # prepare to return n
8     li      $t0, 1
9     ble     $a0, $t0, exit_fib # if (n <= 2) goto exit_fib (ie return n)
10
11    move     $s0, $a0      # save n
12
13    addi     $a0, $a0, -2
14    jal      fib           # fib(n-2)
15
16    addi     $a0, $s0, -1   # prep arg first so we can use s0 to save v0
17    move     $s0, $v0      # save return of fib(n-2) in s0
18    jal      fib           # fib(n-1)
19
20    add      $v0, $v0, $s0  # v0 = fib(n-1) + fib(n-2)
21
22 exit_fib:
23    lw       $ra, 0($sp)
24    lw       $s0, 4($sp)
25    addi     $sp, $sp, 8
26    jr       $ra
```

Notice how we don't have to save `n` any sooner than necessary, ie right before we have to use `a0` to setup the first recursive call. Also the ordering of lines 16 and 17 is important. We needed the original `n` to calculate `n-1` but once that's in `$a0` ready for the call, because we won't need `n` again afterward we can now use `$s0` to preserve the return value of the first call.

Some of you, if you were paying attention, might point out that you could save a few instructions of performance if you moved the base case testing before the prologue as long as you put the exit label after the epilogue. This is true, but I'd recommend against it unless you were really trying to eke out every last microsecond. It's just nicer/cleaner to keep the prologue and epilogue as the first and last thing; they're one more thing to catch your eye and help delineate where functions start and end. Regardless, if you're curious, you can see that version, along with every other function in this chapter in the included program `calling.s`.

Conclusion

While grasping the basics of a calling convention is not too difficult, it takes practice to get used to it and there are many things that we haven't covered in this chapter, like how to pass more than 4 arguments or use `$fp` or handle floating point arguments or return values.

[7] I do not agree with an ironclad "one return" policy in higher level languages. Sometime returning early results in cleaner code, sometimes not. Similarly, ``goto`` is not evil and there are rare cases where using it creates the best code.

[8] Obviously the zero register is not really a variable. I never understood how people could say "const variable" with a straight face, it's literally an oxymoron.

[9] It's an [old link](#), but not as old as SPIM so maybe using it for a frame pointer was added later

Chapter 6: Floating Point Types

Up to this point we haven't really mentioned floating point values or instructions at all, except how to declare them in the `.data` section and the syscalls for reading and printing them. There are two reasons we've left them alone till now. First, they use a whole separate set of registers and opcodes. Secondly, and partly because of the first reason, most MIPS college courses do not ever require you to know or use floating point values. Since this book is targeted at college students, if you know you won't need to know this feel free to skip this chapter.

Floating Point Registers and Instructions

While the greensheet contains a nice table for the normal registers it is completely lacking for the floating point registers. There are 32 32-bit floating point registers. You can use them all for floats but they're paired even-odd for doubles. In other words you can only use even numbers for doubles, because storing a double at `$f0` actually uses `$f0` and `$f1` because it takes 64 bits/8 bytes.

As far as the calling conventions for floating point registers, it is actually hard to find anything definitive even for the basics. Obviously you could make up your own but the float/double syscalls, and the tiny code snippet in Patterson and Hennessy were at least consistent with this old website so I'll go with that. I have seen at least one course page where the prof wanted *all* float registers preserved which seems excessive and ridiculous but prof's are gonna prof.

Table 5. MIPS Floating Point Registers and Uses

Name	Use	Preserved Across a Call
<code>\$f0-\$f2</code>	Function Results	No
<code>\$f4-\$f10</code>	Temporaries	No
<code>\$f12-f14</code>	Arguments	No
<code>\$f16-f18</code>	Temporaries	No
<code>\$f20-f30</code>	Saved Temporaries	Yes

This table is based on doubles so it may look like it's skipping odd registers but they're included where the even they're paired with is. So, for example you actually have 4 registers for float arguments `$f12` through `$f15` but only 2 for doubles `$f12` and `f14`. Similarly you have 12 saved registers for floats but 6 for doubles.

Most of the next table is actually on the Greensheet but not all of it and I thought it worth reproducing here.

Table 6. MIPS floating point instructions (and pseudoinstructions)

Name	Opcode	Format	Operation
Load Word to Coprocessor 1	<code>lwc1</code> (or <code>l.s</code>)	<code>lwc1 ft, n(rs)</code>	$F[ft] = M[R[rs]+n]$
Store Word from Coprocessor 1	<code>swc1</code> (or <code>s.s</code>)	<code>swc1 ft, n(rs)</code>	$M[R[rs]+n] = F[ft]$

Name	Opcode	Format	Operation
Load Double to Coprocessor 1	ldc1 (or l.d)	ldc1 ft, n(rs)	$F[ft] = M[R[rs]+n];$ $F[ft+1] = M[R[rs]+n+4]$
Store Double from Coprocessor 1	sdc1 (or s.d)	sdc1 ft, n(rs)	$M[R[rs]+n] = F[ft];$ $M[R[rs]+n+4] = F[ft+1]$
Move From Coprocessor 1	mfc1	mfc1 rd, fs	$R[rd] = F[fs]$
Move To Coprocessor 1	mtc1	mtc1 rd, fs	$F[fs] = R[rd]$
Convert Word To Single Precision	cvt.s.w	cvt.s.w fd, fs	$F[fd] = (\text{float})F[fs]$
Convert Single Precision To Word	cvt.w.s	cvt.w.s fd, fs	$F[fd] = (\text{int})F[fs]$
Convert Word To Double Precision	cvt.d.w	cvt.d.w fd, fs	$F[fd] = (\text{double})F[fs]$
Convert Double Precision To Word	cvt.w.d	cvt.w.d fd, fs	$F[fd] = (\text{int})F[fs]$
Branch on FP True	bc1t	bc1t label	if (FPcond) goto label;
Branch on FP False	bc1f	bc1f label	if (!FPcond) goto label;
FP Compare	c.y.x	c.y.x fs, ft	$FPcond = (F[fs] \text{ op } F[ft])$ $? 1 : 0$
Absolute Value	abs.x	abs.x fs, ft	$F[fs] = (F[ft] > 0) ? F[ft] : -F[ft]$
Add	add.x	add.x fd, fs, ft	$F[fd] = F[fs] + F[ft]$
Subtract	sub.x	sub.x fd, fs, ft	$F[fd] = F[fs] - F[ft]$
Multiply	mul.x	mul.x fd, fs, ft	$F[fd] = F[fs] * F[ft]$
Divide	div.x	div.x fd, fs, ft	$F[fd] = F[fs] / F[ft]$
Negation	neg.x	neg.x fs, ft	$F[fs] = -F[ft]$
Move	mov.x	mov.x fd, fs	$F[fd] = F[fs]$

With all of the opcodes that end in .x, the x is either s for single precision or d for double precision.

The y in the Compare instructions are one of eq, ne, lt, le, gt, ge. Since only (eq, lt, and le are on the greensheet it's safe to assume the other 3 are pseudoinstructions). Naturally op would be the matching ==, !=, <, <=, >, or >=.

Practice

We're going to briefly go over some of the more different aspects of dealing with floating point numbers but since most of it is the same with just a different set of registers and calling convention, we won't be rehashing most concepts.

Getting Floating Point Literals

The first thing to know when dealing with floats is how to actually get float (or double) literals into registers where you can actually operate on them.

There are 2 ways. The first, and simpler way is to just declare them as globals and then use the `lwc1` or `ldw1` instructions:

```
1 .data
2 a:    .float 3.14159
3 b:    .double 1.61
4
5 .text
6 main:
7
8     la    $t0, a
9     lwc1  $f0, 0($t0)    # get a into $f0
10
11     la    $t0, b
12     ldc1  $f2, 0($t0)    # get b into $f2-3
13
14     # other code here
```

The second way is to use the regular registers and convert the values. Of course this means unless you want a whole value, you'd have to actually do it twice and divide. It looks like this.

```
1     mtc1   $0, $f0      # move 0 to $f0 (0 integer == 0.0 float)
2
3     # get 4 to 4.0 in $f2
4     li     $t0, 4
5     mtc1   $t0, $f2
6     cvt.s.w $f2, $f2    # convert 4 to 4.0
```

As you can see, other than 0 which is a special case, it requires at least 3 instructions, more than the 2 (or 1 if you load directly from the address) of the first method.

Branching

Branching based on floating point values is slightly different than normal. Instead of being able to test and jump in a single convenient instruction, you have to test first and then jump in a second instruction if the test was true or not. This is the same way x86 does it. The test sets a special control/flag register (or a certain bit or bits in the register) and then all jumps are based on its state.

Using it looks like this

```

1    c.lt.s  $f0, $f2    # fpcond = f0 < f2
2    bc1t    was_less   # if (f0 < f2) goto was_less
3
4    # do something for f0 >= f2
5
6    j        blah
7 was_less:
8
9    # do something for f0 < f2
10
11 blah:

```

Functions

Lastly, lets do a simple example of writing a function that takes a float and returns a float. I'm not going to bother doing one for doubles because it'd be effectively the same, or doing one that requires the stack, because the only difference from normal is a new set of registers and knowing which ones to save or not from the table above.

So, how about a function to convert a fahrenheit temperature to celsius:

```

1 .data
2
3 # 5/9 = 0.5 with 5 repeating
4 fahrenheit2celsius: .float 0.555555
5
6 .text
7 # float convert_F2C(float degrees_f)
8 convert_F2C:
9     la      $t0, fahrenheit2celsius
10    lwc1     $f0, 0($t0)    # get conversion factor
11
12    # C = (F - 32) * 5/9
13    li      $t0, 32
14    mtc1     $t0, $f1        # move int 32 to f1
15    cvt.s.w  $f1, $f1        # convert to 32.0
16
17
18    sub.s    $f12, $f12, $f1 # f12 = degrees - 32
19
20    mul.s    $f0, $f0, $f12 # f0 = 0.555555 * f12
21
22    jr      $ra

```

You can see we follow the convention with the argument coming in f12 and the result being returned in f0. In this function we use both methods for getting a value into float registers; one we load from memory and the other, being an integer, we move and convert.

Conclusion

As I said before, it is rare for courses to even bother covering floating point instructions or assign any homework or projects that use it, but hopefully this brief overview, combined with the knowledge of previous ones is sufficient.

There are also 2 example programs `conversions.s` and `calc_pi.s` for you to study.

Chapter 7: Tips and Tricks

This chapter is a grab bag of things you can do to improve your MIPS programs and make your life easier.

Formatting

You may have noticed I have a general format I like to follow when writing MIPS (or any) assembly. The guidelines I use are the following

1. 1 indent for all code excluding labels/macros/constants.

I use hard tabs set to a width of 4 but it really doesn't matter as long as it's just 1 indent according to your preferences.

2. Use **spaces** to align the first operand of all instructions (syscall obviously

doesn't have any) out far enough. Given my 4 space tabs, this usually means column 11-13 for me. The reason to use spaces is to prevent the circumstances that gave hard tabs a bad name. When you use hard tabs for alignment, rather than indentation, and then someone else opens your code with their tab set to a different width, suddenly everything looks like crap. Tabs for indentation, spaces for alignment. Or as is increasingly common (thanks Python), spaces for everything but I refuse to do that to the poor planet.`{green_tabs}`

3. A comma and a single space between operands. The simulators don't actually

require the comma but since other assembly languages/assemblers do, you might as well get used to it. Besides I think it's easier to read with the comma, though that might just be me comparing it to passing arguments to a function.

4. Comment every line or group of closely related lines with the purpose,

often just the equivalent C code. You can relax this a little as you get more experience.

5. Use a blank line to separate logically grouped lines of code. While you

can smash everything together vertically, I definitely wouldn't recommend it, even less than I would in a higher level language.

6. Put the `.data` section at the top, similar to declaring globals in C.

There are exceptions for this. When dealing with a larger program with lots of strings, it can be convenient to have multiple `.data` sections with the strings you're using declared close to where you use them. The downside is you have to keep swapping back and forth between `.text` and `.data`.

Misc. General Tips

1. Try to use registers starting from 0 and working your way up. It helps

you keep track of where things are (esp. combined with the comments). This obviously can fall apart when you discover you forgot something or need to modify the code later and it's often not worth changing all the registers you're already using just so you have that nice sequence. When that happens I'll sometimes just pick the other end of sequence (ie `$t9` or `$s7`) since if it's out of order I might as well make it obvious.

2. Minimize your jumps, labels, and especially your level of nested loops.

This was already covered in the chapters on branching and loops but it bears repeating.

3. In your prologue save `$ra` first (at the top of the stack) if necessary

and all s regs you used (starting at `$s0`) in order going down. Then copy paste the whole thing to the bottom, move the first line to the bottom and change the number to positive and change all the `sw` to `lw`.

```
func:
    addi    $sp, $sp, -20
    sw      $ra, 0($sp)
    sw      $s0, 4($sp)
    sw      $s1, 8($sp)
    sw      $s2, 12($sp)
    sw      $s3, 16($sp)

    # body of func here that calls another function or functions
    # and needs to preserve 4 values across at least one of those calls

    lw      $ra, 0($sp)
    lw      $s0, 4($sp)
    lw      $s1, 8($sp)
    lw      $s2, 12($sp)
    lw      $s3, 16($sp)
    addi    $sp, $sp, 20
```

Constants

One of the easiest things you can do to make your programs more readable is use defined constants in your programs. Both MARS and SPIM have ways of defining constants similar to how C defines macro constants; ie they aren't "constant variables" that take up space in memory, it's as if a search+replace was done on them right before assembling the program.

Let's look at our Hello World program using constants for SPM and MARS

SPIM:

```

1 sys_print_str = 4
2 sys_exit = 10
3
4 .data
5 hello:  .asciiz "Hello World!\n"
6
7 .text
8 main:
9     li    $v0, sys_print_str
10    la    $a0, hello # load address of string to print into a0
11    syscall
12
13    li    $v0, sys_exit
14    syscall

```

MARS:

```

1 .eqv sys_print_str 4
2 .eqv sys_exit 10
3
4 .data
5 hello:  .asciiz "Hello World!\n"
6
7 .text
8 main:
9     li    $v0, sys_print_str
10    la    $a0, hello # load address of string to print into a0
11    syscall
12
13    li    $v0, sys_exit
14    syscall

```

Macros

MARS supports function style macros that can shorten your code and improve readability in some cases (though I feel it can also make it worse or be a wash).

The syntax looks like this:

```

1 .macro macroname
2 instr1 a, b, c
3 instr2, b, d
4 # etc.
5 .end_macro
6
7 # or with parameters
8 .macro macroname(%arg1)
9 instr1 a, %arg1
10 instr2 c, d, e
11 # etc.
12 .end_macro

```

Some common examples are using them to print strings:

```

1 .macro print_str_label(%x)
2 li $v0, 4
3 la $a0, %x
4 syscall
5 .end_macro
6
7 .macro print_str(%str)
8 .data
9 str: .asciiz %str
10 .text
11 li $v0, 4
12 la $a0, str
13 syscall
14 .end_macro
15
16 .data
17
18 str1: .asciiz "Hello 1\n"
19
20 .text
21 # in use:
22     print_str_label(str1)
23
24     print_str("Hello World\n")
25
26     ...

```

You can see an example program in `macros.s`

Unfortunately, as far as I can tell SPIM does not support function style macros despite what MARS's documentation implies about using a \$ instead of a % for arguments.

Switch-Case Statements

It is relatively common in programming to have to compare a integral type variable (ie basically any built in type but float and double) against a bunch of different constants and do something different based on what it matches or if it matches none.

This could be done with a long if-else-if chain, but the longer the chain the more likely the programmer is to choose a switch-case statement instead.

Here's a pretty short/simple example in C:

```
1  printf("Enter your grade (capital): ");
2  int grade = getchar();
3  switch (grade) {
4  case 'A': puts("Excellent job!"); break;
5  case 'B': puts("Good job!"); break;
6  case 'C': puts("At least you passed!"); break;
7  case 'D': puts("Probably should have dropped it..."); break;
8  case 'F': puts("Did you even know you were signed up for the class?"); break;
9  default: puts("You entered and invalid grade!");
10 }
```

You could translate this to its equivalent if-else chain and handle it just like we cover in the chapter on branching. However, imagine if this switch statment had a dozen cases, two dozen etc. The MIPS code for that quickly becomes long and ugly.

So what if we implemented the switch in MIPS the same way it is semantically in C? The same way compilers often (but not necessarily) use? Well before we do that, what is a switch actually doing? It is *jumping* to a specific case label based on the value in the specified variable. It then starts executing, falling through any other labels, till it hits a **break** which will jump to the end of the switch block. If the value does have its own case label, it will jump to the default label.

Compilers handle it by creating what's called a jump table, basically an array of label addresses and using the variable to calculate an index in the table to use to jump to.

The C equivalent of that would look like this:

```
1  #include <stdio.h>
2
3
4  // This compiles with gcc, uses non-standard extension
5  // https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html
6
7  int main()
8  {
9
10     // jump table
11     void* switch_table[] =
12     { &a_label, &b_label, &c_label, &d_label, &default_label, &f_label };
```

```

13
14     printf("Enter your grade (capital): ");
15     int grade = getchar();
16     grade -= 'A'; // shift to 0
17
18     if (grade < 0 || grade > 'F' - 'A')
19         goto default_label;
20
21     goto *switch_table[grade];
22
23 a_label:
24     puts("Excellent job!");
25     goto end_switch;
26
27 b_label:
28     puts("Good job!");
29     goto end_switch;
30
31 c_label:
32     puts("At least you passed?");
33     goto end_switch;
34
35 d_label:
36     puts("Probably should have dropped it...");
37     goto end_switch;
38
39 f_label:
40     puts("Did you even know you were signed up for the class?");
41     goto end_switch;
42
43 default_label:
44     puts("You entered an invalid grade!");
45
46
47 end_switch:
48
49
50     return 0;
51 }

```

The `&&` and `goto *var` syntax are actually not standard C/C++ but are GNU extensions that are supported in gcc (naturally) and clang, possibly others.^[10]

Notice how the size of the jump table is the value of the highest valued label minus the lowest + 1. That's why we subtract the lowest value to shift the range to start at 0 for the indexing. Secondly, any values without labels within that range are filled with the default label address. Lastly there has to be an initial check for values outside the range to jump to default otherwise you could get an error from an invalid access outside of the array's bounds.

The same program/code in MIPS would look like this:

```

1 .data
2
3 a_str: .asciiz "Excellent job!\n"
4 b_str: .asciiz "Good job!\n"
5 c_str: .asciiz "At least you passed?\n"
6 d_str: .asciiz "Probably should have dropped it...\n"
7 f_str: .asciiz "Did you even know you were signed up for the class?\n"
8
9 invalid_str: .asciiz "You entered an invalid grade!\n"
10
11 enter_grade: .asciiz "Enter your grade (capital): "
12
13 switch_labels: .word a_label, b_label, c_label, d_label, default_label, f_label
14
15 .text
16
17 main:
18
19     li        $v0, 4
20     la        $a0, enter_grade
21     syscall
22
23     li        $v0, 12    # read char
24     syscall
25
26     li        $t2, 5     # f is at index 5
27
28     la        $t0, switch_labels
29     addi      $t1, $v0, -65 # t1 = grade - 'A'
30     blt      $t1, $0, default_label # if (grade-'A' < 0) goto default
31     bgt      $t1, $t2, default_label # if (grade-'A' > 5) goto default
32
33     sll      $t1, $t1, 2    # offset *= 4 (sizeof(word))
34     add      $t0, $t0, $t1  # t0 = switch_labels + byte_offset =
    &switch_labels[grade-'A']
35     lw      $t0, 0($t0)    # load address from jump table
36     jr      $t0            # jump to
37
38 a_label:
39     la      $a0, a_str
40     j      end_switch
41
42 b_label:
43     la      $a0, b_str
44     j      end_switch
45
46 c_label:
47     la      $a0, c_str
48     j      end_switch
49

```

```

50 d_label:
51     la      $a0, d_str
52     j       end_switch
53
54 f_label:
55     la      $a0, f_str
56     j       end_switch
57
58 default_label:
59     la      $a0, invalid_str
60
61
62 end_switch:
63     li      $v0, 4
64     syscall
65
66     li      $v0, 10    # exit
67     syscall

```

It's easy to forget that `jr` does not actually stand for "jump return" even though it's almost always used for that purpose. Instead it stands for "jump register" and we can use it to do the equivalent of the computed goto statement in C.

While this example probably wasn't worth making switch style, because the overhead and extra code of making the table and preparing to jump balanced out or even outweighed the savings of a branch instruction for every case, as the number of options increases, the favor tilts toward using a jump table like this as long as the range of isn't too sparse. If the range of values is in the 100's or 1000's but you only have cases for a dozen or so, then obviously that isn't worth it to create a table that large just to fill it almost entirely with the default label.

Just to reiterate, remember it is not about the magnitude of the actual values you're looking for, just the difference between the highest and lowest because $\text{high} - \text{low} + 1$ is the size of your table.

Command Line Arguments

Command line arguments, also known as program arguments, or command line parameters are strings that are passed to the program on startup. In high level languages like C, they are accessed through the parameters to the main function (naturally):


```

1 #include <stdio.h>
2
3 int main(int argc, char** argv)
4 {
5     printf("There are %d command line arguments:\n", argc);
6
7     for (int i=0; i<argc; i++) {
8         printf("%s\n", argv[i]);
9     }
10
11     return 0;
12 }

```

As you can see, `argc` contains the number of parameters and `argv` is an array of C strings that are those arguments. If you run this program you'll get something like this:

```

$ ./args 3 random arguments
There are 4 command line arguments:
./args
3
random
arguments

```

Notice that the first argument is the what you actually typed to invoke the program, so you always have at least one argument.

MIPS works the same way. The number of arguments is in `a0` and an array of strings is in `a1` when `main` starts. So the same program in MIPS looks like this;

```

1 .data
2
3 there_are: .asciiz "There are "
4 arguments: .asciiz " command line arguments:\n"
5
6 .text
7
8 main:
9     move    $t0, $a0 # save argc
10
11     li      $v0, 4
12     la      $a0, there_are
13     syscall
14
15     move    $a0, $t0
16     li      $v0, 1 # print int
17     syscall
18
19     li      $v0, 4
20     la      $a0, arguments
21     syscall
22
23     li      $t1, 0 # i = 0
24     j       arg_loop_test
25
26 arg_loop:
27     li      $v0, 4
28     lw      $a0, 0($a1)
29     syscall
30
31     li      $v0, 11
32     li      $a0, 10 # '\n'
33     syscall
34
35     addi    $t1, $t1, 1 # i++
36     addi    $a1, $a1, 4 # argv++ ie a1 = &argv[i]
37 arg_loop_test:
38     blt     $t1, $t0, arg_loop # while (i < argc)
39
40     li      $v0, 10
41     syscall

```

This program works exactly like the C program when using SPIM:

```
$ spim -file args.s 3 random arguments
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /usr/lib/spim/exceptions.s
There are 4 command line arguments:
args.s
3
random
arguments
```

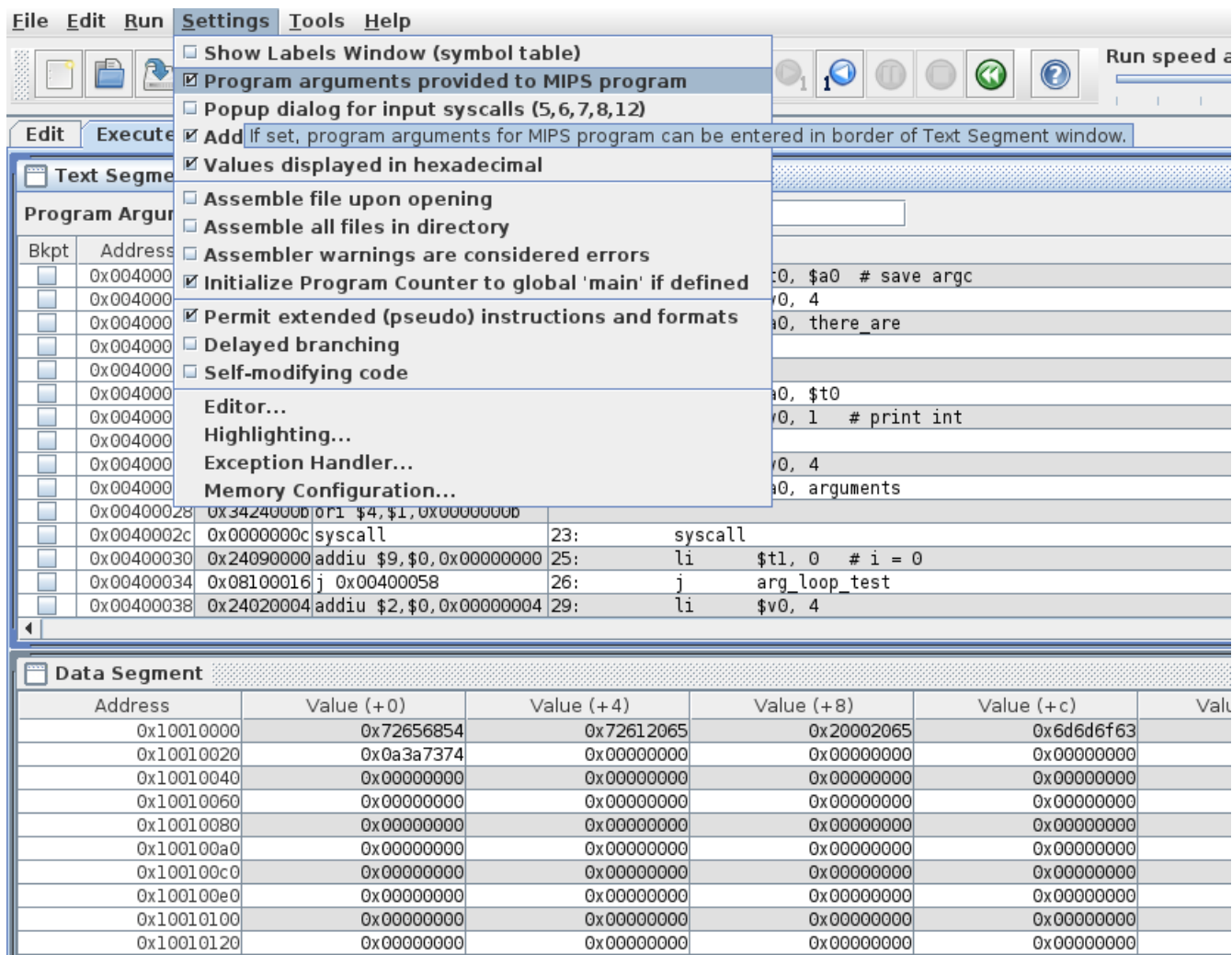
Obviously the commands for SPIM itself are not included but the file name (args.s) takes the place as the "executable".

Unfortunately, MARS works differently, probably because it's more GUI focused. It does not pass the program/file name as the first argument, so you can actually get 0 arguments:

```
$ java -jar ~/Mars4_5.jar args.s pa 3 random arguments
MARS 4.5 Copyright 2003-2014 Pete Sanderson and Kenneth Vollmar

There are 3 command line arguments:
3
random
arguments
```

You can see that you have to put "pa" (for "program arguments") to indicate that the following strings are arguments. In the GUI, there is an option in "Settings" called "Program arguments provided to MIPS program" which if selected will add a text box above the Text Segment for you to enter in the arguments to be passed.



References and Useful Links

- [Greensheet](#)
- [MARS syscall list](#)
- [learnxinyminutes page](#)
- [Randolph-Macon College MIPS Reference](#)
- [CCSU MIPS Reference](#)