MIPS tips and tricks

General tips

A lot of these follow from the principle of trying to write as little code as possible. The lesser code you write, the lesser opportunity you have to mess something up, and the lesser code you have to debug, so it's a win-win.

Construct values in appropriate registers and avoid unnecessary moves

If you're constructing a function argument, place it in the appropriate $a register directly instead of using a $t register and then moving it. Similarly, if you're constructing a return value, put it in $v0 directly.

This is bad:

```
add     $t2, $t0, $t1
move    $a0, $t2
```

This is good:

```
add     $a0, $t0, $t1
```

And this is completely redundant, but I've seen a surprising number of people do it:

```
move    $v0, $v0
```

Use enhanced and pseudo instructions

A pseudo instruction is one that doesn’t exist at all in the ISA and is synthesized from other instructions, whereas an enhanced instruction is one that exists in the ISA but is provided with superpowers. Here’s a good list of SPIM instructions and pseudo-instructions, although it’s not 100% complete. For an example of an enhanced instruction, `beq` in MIPS can only compare two registers, but SPIM will happily compare a register and an intermediate. `addi` can only accept 16-bit immediate, but if you supply `add` with a 32-bit immediate, it will generate the correct sequence of instructions for you. `addi` won’t, however, so a corollary is that you should never explicitly use the immediate forms of instructions - let the assembler figure it out for you instead.

If you're not sure if something will work or not, the best way to find out is to try to assemble it and see if SPIM yells at you. If you're feeling more adventurous, you can try and grok SPIM's parser, which is pretty omnipotent.

Make use of extended addressing modes

This is similar to the above. While MIPS provides only one addressing mode - the `constant (register)` that you all know and love - SPIM adds a bunch more on top. Patterson and Hennessy Appendix A section A.10 has a list, although it’s missing the `label (register)` mode. For example, here's some MIPS that declares a global variable `index` and later on increments it:
And here's some code that loads some values from an array of words which is labeled array:

```assembly
lw $t0, array      # load array[0]
lw $t1, array + 4  # load array[1]
# suppose $a0 holds an index and we want array[$a0]
sll $t2, $a0, 2     # since each element is 4 bytes
lw $t2, array($t2) # load array[$a0]
```

### Don't set up a stack frame unless necessary

If your function is a leaf function (i.e. it doesn't call any other functions), you don't need to allocate a stack frame unless you run out of caller-saved registers (and given MIPS' abundance of registers, that's pretty unlikely). This applies to parts of functions as well, so for example, if you're writing a recursive function and the base case doesn't call any functions, you only need to set up a stack frame in the recursive part.

There's a rare but useful special case, which is when you only need to set up a stack frame because of a function call that's the very last thing in your function. This is known as a tail call, and instead of setting up a stack frame, saving your return address, calling the function, getting its return value, restoring your return address and then returning that value to your caller, you can cut out the middleman, jump to the function instead of jumping-and-linking to it, and let it return to your caller directly. For an example of this, see `print_hex_and_space` from the QtSpim lab exercise. `delete_node`, `remove_value` and the recursive case of `insert_value` from Lab 8 can also benefit from this.

### If you need a stack frame, set it up and tear it down exactly once, and have a single exit point

Setting up a stack frame refers to subtracting from `$sp` to make space for it, and tearing it down means adding back to `$sp`. You should set up your stack frame exactly once, usually at the very beginning of your function (or at the very beginning of your recursive case for many recursive functions). Similarly, you should tear it down exactly once, usually at the very end of your function. Don't allocate stack frames willy-nilly in the middle of functions, particularly in loop bodies - that makes me a sad panda.

Since stack frame teardown can involve a decent amount of code if you need to restore callee-saved registers, you should have a single exit point which handles it - duplicating stack teardown code is recipe for disaster. Even if your function has multiple return statements, you can just put the appropriate value into `$v0` and then jump to the exit point.

### If you need a stack frame, use callee-saved registers

This one is more subjective, but I find that if you need to set up a stack frame anyway, it's a lot easier to use callee-saved registers. The idea is that your function ends up with the following structure:

```assembly
# set up stack frame
# save $s registers
# move values into $s registers as needed
# actual function code
# restore $s registers
# tear down stack frame
```
All stack-related operations occur at the start and end. The actual function code remains clean uncluttered, which is good since it’s the interesting part. To get a better sense of how caller and callee saved registers stack up (if you’ll pardon the terrible pun), you can take a look at Professor Zilles’ videos.

**Follow proper saving and restoring conventions**

Callee-saved registers (and remember that this includes $ra) should be saved once on stack frame setup and restored once on stack frame teardown. For caller-saved registers, the rules are trickier - you want to make sure that the saved value doesn’t go stale (i.e. become inconsistent with the actual register value), and you want to restore the value before it’s used following a function call. A conservative method is that if you need to preserve a caller-saved register across a call, save it immediately before the call and restore it immediately afterwards. There’s some potential for optimization here, particularly in the saving step; just keep the two guidelines in mind.

**Don’t blindly save/restore or use callee-saved registers for everything**

The whole point of saving and restoring values on the stack or using callee-saved registers is preserving values across calls. It follows that if you don’t need a value preserved across a call, you can just use a caller-saved register for it and not worry about saving/restoring. It’s a good idea to analyze the function you have to translate, as Professor Zilles does in his videos, to figure out which values you need preserved and which you can just discard.

**Neat things**

**Named constants**

As you would have already seen in your labs, SPIM supports named constants, which can be used in place of immediates. Named constants make code a lot easier to understand, since e.g. `BONK_ACKNOWLEDGE` is immediately obvious whereas `0xffff0060` probably isn’t. We’ve provided you with a bunch of SPIMbot related constants already, so make use of those and feel free to define your own as needed.

**extern and $gp**

Since load and store instructions are limited to a 16-bit immediate, loads and stores to labels often get expanded into two instructions: one to deal with the upper 16 bits and one to deal with the lower 16. The first 64 kB of the data segment are set aside, however, and $gp points to the middle of this sub-segment, so you can reference any address in it with a single instruction using an offset from $gp. The best part is that SPIM takes care of the offset for you, so you don’t even need to put in any extra work.

The way you actually tap into this sub-segment is through the

```
.extern label_name, data_size
```

directive, which takes a name and a size and associates the name with a storage area of the requested size, provided there’s enough space left in the sub-segment. (Stock SPIM actually limits the size to 8 bytes or less, for reasons I disagree with, so I’ve removed this restriction in the versions of QtSpim and QtSpimbot we use.) Unfortunately, there’s no way to initialize the data - it’s guaranteed to be zero-initialized, but you can’t do anything other than that.

To see this in action, you can use this example code. You’ll notice the instructions referencing `foo_gp` get converted to a single instruction using an offset from $gp, whereas the instructions referencing `foo_actual` get expanded to two instructions.

(If we’re being pedantic, which I love to be, .extern is meant for global symbols, and .lcomm is its equivalent for local symbols. You can just stick with .extern for simplicity, however.)

**Optimization tips**

There are two major things to keep in mind. Firstly, you should figure out what you need to optimize. QtSpimbot has a very cool command line option, `-prof_file <filename>`, which writes execution counts for each instruction into the specified file - see the SPIMbot documentation for more details. You can use this to get a sense of where you’re spending the most time and optimize accordingly - innermost loops are often good candidates. Secondly, algorithm improvements beat micro-optimizations almost all the time - my highly tuned bubblesort will lose to even an extremely unoptimized quicksort.

With that out of the way, here are some general tips that can come in handy when you’re coding for performance.
Beware of enhanced and pseudo instructions

Enhanced and pseudo instructions are great for expressing something concisely, but they don't always generate the most efficient code. For example, if I do

```plaintext
li $t0, 0xffffffff
```

SPIM generates the equivalent of

```plaintext
lui $at, 0xffff
or $t0, $at, 0xffff
```

whereas it can be done in a single instruction. (Hint: think 2's complement.) Similarly,

```plaintext
sle $t2, $t0, $t1
```

gets converted to

```plaintext
bne $t0, $t1, sle_slt
li $t2, 1
j sle_done

sle_slt:
slt $t2, $t0, $t1

sle_done:
```

which is pretty atrocious - you can do this in two instructions.

Most of the time, however, the assembler's pretty smart, but it's always a good idea to keep an eye on the code it generates for you. (Of course, I could simply fix the non-optimal cases in the code generator itself as I found them, but what fun would that be?)

Count loops towards zero

One of the interesting things about MIPS is that there are specialized instructions for comparisons with zero - namely `bgez`, `bgtz`, `blez` and `bltz` - but arbitrary branch comparisons usually need to be expanded into an `slt` or `slti` followed by a comparison with zero. For this reason, whenever it doesn't affect computation, you should count loops towards zero, so that the termination check is one actual instruction instead of two.

Use pointer manipulations instead of naive array accesses

Accessing arrays is a bit of drag in MIPS - you need to multiply the index by the size of each element, add that to the base pointer and then perform a load. You can skip the first two steps if you just manipulate pointers directly instead of doing a literal translation of the C code. For example, here's a function which sums up all the elements of an array:
Here's a literal MIPS translation - this should be pretty straightforward.

```mips
array_sum:
    li        $v0, 0                # $v0 = sum
    li        $t0, 0                # $t0 = i

as_loop:
    bge        $t0, $a1, as_done
    sll        $t1, $t0, 2        # i * 4
    add        $t1, $a0, $t1        # &array[i]
    lw        $t1, 0($t1)        # array[i]
    add        $v0, $v0, $t1        # sum += array[i]
    add        $t0, $t0, 1        # ++i
    j        as_loop

as_done:
    jr        $ra
```

We can still do better if we check size before entering the loop, which enables us to convert the loop to a do-while loop:

```mips
array_sum:
    li        $v0, 0                # $v0 = sum
    sll        $a1, $a1, 2        # size * 4
    add        $a1, $a0, $a1        # get pointer to end of the array

as_loop:
    bge        $a0, $a1, as_done
    lw        $t0, 0($a0)        # *array
    add        $v0, $v0, $t0        # sum += *array
    add        $a0, $a0, 4        # make array point to next element
    j        as_loop

as_done:
    jr        $ra
```
array_sum:
    li      $v0, 0                # $v0 = sum
    blez    $a1, as_done        # bail out if size <= 0
    sll     $a1, $a1, 2        # size * 4
    add     $a1, $a0, $a1        # get pointer to end of the array

as_loop:
    lw      $t0, 0($a0)        # *array
    add     $v0, $v0, $t0        # sum += *array
    add     $a0, $a0, 4        # make array point to next element
    bne     $a0, $a1, as_loop

as_done:
    jr      $ra

The obvious benefit is that we avoid a j instruction. The perhaps not-so-obvious benefit is that we’ve gone from a pseudo-branch to a real branch, saving another instruction. We’re saving two instructions over the previous version, which was already two instructions better than the original, which is pretty good.

**Keep an eye out for obscure instructions**

MIPS has some instructions which aren’t commonly used but could come in handy in specific situations - some examples I can think of are bgezal, bltzal, clo and clz. The best reference for such things is Volume II of the MIPS architecture manuals - you’ll need to create a free account to view the manuals, but it’s definitely worth the trouble.

**Cache expensive computations in a lookup table**

Lookup tables are a common technique to replace computation with lookups. The idea is that, instead of performing an extensive computation at run-time, you store pre-computed values in an array from which they can be retrieved. This was very commonly used in old video games (think the original Sonic the Hedgehog) for things like sines and arctangents - I don’t know if it is nowadays, but considering that game programmers care about performance enough to come up with crazy things like fast inverse square root, it might be. On a real machine, you would need to take memory access time into account, particularly in the case of a cache or TLB miss, but SPIMbot doesn’t have caching or virtual memory so that’s not a concern.