MIPS Assembly Style Guide

CSc 252 – Computer Organization
Version 1.3 (27 Dec 2017)

Intro

Assembly language is a very low-level programming language. As such, it's very easy to get lost in the details. Far more than any other language you've ever used, good style – and especially good comments – will be critical.

We will be grading you for style, and also for your comments. **Follow this style guide,** or expect to lose points on every programming assignment.

Exceptions
My intent in writing this document is not to cause you trouble or worry while you write your code; rather, it is to help you write good, clean, readable code. Doing this requires that you put a little bit of work into good style and commenting.

However, if you believe that you could make your program **more** readable by violating a rule, that's OK. Just remember: you can violate a rule to make your program more readable; never break one **just to save yourself work.**

Elements

Good assembly language style has several elements:

- Clear and consistent indentation (of the instruction names **and** the operands)
- Abundant and informative line comments
- Occasional block comments
- Good label names and block ordering

Indentation

Your code must be indented. Use a consistent indentation style in the entire file. You may choose to use either tabs or spaces; however, you must **use the same throughout.**[1]

You must use indentation at **two points in each line.** The first, you have already seen in other languages: indent every line (except for labels and directives, which we'll discuss in a moment). Make the start of each line line up. That is, the first character of each instruction should be at the same column.

In addition, you must also line up the first operand of each instruction. The second and third operands do not have to line up (although it's not a bad idea!).

Good Example

1 If you use tabs on some lines and spaces on others, it may look like it lines up on your screen – but someone else, with a different tab-stop setting, might see something very different!

```
    la    $t1, foo        # t1 = &foo
    addi  $t2, $t3, $t4   # x  = y+z
```

Bad Example

```
    la $t1, foo   # t1 = &foo
    addi $t2, $t3, $t4   # x  = y+z
```

Non-Indented Lines

The following lines should not be indented:

- Labels (anything with a colon)
- `.data`
- `.text`
- `.globl <LABEL>`

Storage Directives

Storage directives, like `.word` , should be indented exactly the same as instructions. If you use a label, immediately followed by a storage directive such as `.word` , do not indent the label – but **do** indent the storage directive. Likewise, always indent the operand of a storage directive, just like the operand of an instruction.

Good Example

```
# both of these examples allocate a single word.
foo:
    .word     42    # this form is OK
bar: .word    83    # so is this
```

Line Comments

Comment most of your lines of code (at least 50% of them). Line comments should be on the same line as the instruction – unless they are so long that they cannot fit.

As much as possible, use high-level names and expressions (or English) in your comments. For instance, imagine that the register `$s0` holds the variable `foo`, `$s1` holds the variable `bar`, and `$s2` holds the variable `baz`. The instruction

```
    add  $s0, $s1, $s2
```

should generally be commented something (roughly) like this:

```
    add  $s0, $s1, $s2       # foo = bar + baz
```

Loading and Storing Variables

`la` instructions, and the load/store instructions that follow, should generally use C syntax in comments:

```
    la   $t0, foo       # t0 = &foo
    lw   $t0, 0($t0)    # t0 =  foo
    la   $t1, bar       # t1 = &bar
    sw   $t0, 0($t1)    # bar = foo
```

Temporary Variables

When a temporary variable is used to hold a result (that is, a value which would not be given a name in C/Java), it is perfectly fine to use the register name as the name of the variable. However, when that

register is later used in another instruction, use the original expression (not the register name), to make things clearer:

```
add  $t0, $s0, $s1       # t0 = foo + bar
add  $s2, $t0, $s1       # fred = foo + bar + baz
```

Comparisons and Conditional Branches

`beq/bne` instructions should be commented using C/Java style, or pseudocode. Simply restating the assembly language instruction, such as "branch if $t0 is zero" is **not sufficient.**

However, given the limited amount of space on a line, typically I describe the *branch* that is occurring, rather than the C code that would be executed inside the block.

Example C Code

```
if (x == y)
    x++;
```

Equivalent MIPS Assembly (my style)

```
     bne  $s0, $s1, AFTER_IF       # if (x != y) skip ahead
     addi $s0, $s0, 1              # x++
AFTER_IF:
```

Alternate Comment Style

```
     bne  $s0, $s1, AFTER_IF       # if (x == y) then...
     addi $s0, $s0, 1              # x++
AFTER_IF:
```

Inequalities should typically have comments on both the `slt` and the `beq/bne` instructions, since it is often difficult to keep track of what is going on. As mentioned above, when you set a temporary register, it's OK to simply name the register. However, when you **use** it, drop in the C/Java expression instead of the register name:

```
     slt  $t0, $s0, $s1            # t0 = (x < y)
     beq  $t0, $zero, AFTER_IF     # if (x >= y) skip ahead
     addi $s0, $s0, 1              # x++
AFTER_IF:
```

Syscalls

Syscalls are very common in our test programs. You do not have to comment the `syscall` instruction; however, you should generally comment the instructions which set the `$v0` and/or `$a0` registers:

```
     addi $v0, $zero, 1            # print_int(myVar)
     add  $a0, $s3, $zero
     syscall

     addi $v0, $zero, 11           # print_char
     addi $a0, $zero, 0xa          # ASCII '\n'
     syscall
```

Block Comments

Non-trivial sections of code should, **in addition to line comments,** have block comments before them, explaining the purpose of the code which follows. While sometimes it may make sense to use text, I often find that the clearest way to describe a block of assembly is with the corresponding C.

It is often helpful (though not strictly required) that you list out all of the registers that you plan to use, and what they will be used for. Most registers will hold a single variable in some block of code, and you should document the register and the name of the variable it is holding. If you plan to use other registers as transient temporaries (for instance, to hold the result of an `slt`), then it is often useful to document this as well.

```
# for (int i=0; i+1<count; i++)
# {
#     if (array[i] > array[i+1])
#         printf("Out of order at position %d\n", i);
# }
#
# REGISTERS
#   s0 – i
#   s1 – count
#   s2 – array
#   t0 – array[i]
#   t1 – array[i+1]
#   t2 - various temporaries
.data
MSG1:
     .asciiz "Out of order at position "
.text
     addi  $s0, $zero,0       # i  = 0
     la    $s1, count         # s1 = &count
     lw    $s1, 0($s1)        # s1 =  count
     ...
```

All loops should have block comments at the head of the loop. Nested loops typically would have a block comment at the head of both loops – although occasionally, it is OK to have a single comment which describes both. Remember, the point of comments is clarity!

Simple `if()` statements do not necessarily need block comments – although their operation should be easy to understand from a **previous** block comment. Complex `if()` statements generally should have their own block comments.

Repeated Blocks

Occasionally, you find yourself cut-n-pasting the same code over and over again, with small variations. In this case, it is permissible to simply refer back to the previous code:

```
# we repeat the same thing as just above, 5 more
# times...we just replace $s0 with the proper
# register for the variable we're using
```

Label Names and Block Ordering

You may use whatever label names you want. However, try to make them as readable as practical.

Blocks should be ordered in roughly the same order that they would be present in C/Java code: that is, loops should jump backwards (not forwards) and `if()` statements should jump forwards (not backwards). The prologue of a procedure should be the first block, and the epilogue the last; if you exit from the middle of a function, then typically you should jump **ahead** to the epilogue, rather than making a duplicate copy in the block where you are.