Assembly Language Style Guide

Style is especially important in assembly code, since you have so much control over the environment. Some conventions must be used so that other people (mostly the TAs) can understand what you are doing. ***It is not the goal of CS31 to teach you to write the fastest assembly code possible***, but instead to teach you how high-level languages can be reduced to assembly language instructions.

The coding conventions for cs31 are very much like those for any other CS class, except in some ways they are more significant. In a high-level language, if you don't name an instance with an underscore, you can still depend on your code working, and everyone else's. However, in assembly, if you fail to follow the conventions for register usage, you could do serious damage to other parts of the program. For example, if you write a subroutine called by someone else's code (okay, I know that no one is going to go out and write MIPS assembly for a living, but work with me), and you use $s registers without saving and restoring them, their code will fail. By the same token, if you use a $t register, call a subroutine, and expect the value in the register to be there when the subroutine returns, you might be sadly mistaken.

Commenting

There are a few basic commenting conventions we will watch very closely in your programs. First, every procedure should have a header which contains the procedure name, the parameters, the return values, the registers used, which procedures call the procedure, which procedures this procedure calls, and a brief description of the procedure's function. Exactly how this header looks is up to you, as long as it contains this information.

All inline comments should be in pseudocode, or C or Java equivalents. We understand that not every line translates to a line of high-level code, but most control structures and arithmetic operations can be represented in a high-level way. Also remember to indent your inline comments as you would lines of pseudocode or a high-level language

Finally, at the top of your program you should have a program header and a list of equates (constants) with comments. The program header should have your name, account number, and a description of your program. The equates are similar to constants in a high-level language, and you need to explain what each one is for. Also, the variables in your data section should have explanations.

Inline Comments

Your comments should be in pseudocode that corresponds roughly to Java or C code. It should explain what your assembly code is doing by ADDING information. (Don't just repeat what the line of assembly obviously does.) Give variables meaningful names. Try to show relationships across lines, not just information contained in a single line.

Example of bad commenting: (this is from the exponentiate function in sample code)

```
exponentiate:
(stuff deleted)
     move      $t0, $a0       # $t0 = $a0
     move      $t1, $a1       # $t1 = $a1
     li        $t2, 1         # $t2 = 1

powerloop:
     beqz $t1, endexponentiate# if $t1 = 0 goto endexponentiate

     mul       $t2, $t2, $t0  # $t2 = $t2 * $t0
     sub       $t1, $t1, 1    # $t1 = $t1 - 1
     j    powerloop
```

Why is this commenting bad? Two reasons. First, TAs already know MIPS; they wouldn't be TAing if they didn't, so telling them that mult $t2, $t2, $t0 is the same as $t2 = $t2 * $t0 is not only uninformative, it is annoying. For an example of good commenting of this subroutine, see the "exponentiate" subroutine in the **Sample Code** section. Keep in mind that you don't necessarily have to comment every line of code (you probably shouldn't). Use your brain. The goal is to communicate, not to overwhelm.

You should break long comments up into several lines (each beginning with a #, of course). Long lines in the .mal file can cause SPIM to crash.

Register Usage Table

Every subroutine should have a register usage table. A register usage table tells what registers are used in a sub, and what they are used for (e.g., if you could give a pseudocode variable name for a register, tell what that name is in the register usage table).

Flow of Control Structures

Control structures are an essential part of structured programming. Since assembly language doesn't have any flow of control structures like Java and C do, you must code the more complex structures yourself. The simplest flow of control statement is the IF-THEN(-ELSEIF-ELSE) statement. For this structure, the most important thing to avoid is a long series of conditional branch statements in a row followed by sections of code. This would be impossible to comment. A better structure is to do a comparison, then if the branch is not taken, execute some code and jump to the end of the code block. If the branch is taken, execute some code which finishes at the end of the code block. For example:

```
     .
     bgt  $t0,$t1,bigger # if (t0<=t1) {
     .                   #    code if the value is smaller
     b    endif          # }
bigger:                  # else {
     .                   #    code if the value is bigger
endif:                   # }
```

```
.                       #continue with the program
```

Notice that the comparison (bgt--branch if greater) is the opposite of the commented comparison (`if (t0<=t1)`). It is often necessary to have this reverse logic in order to make the comments make sense.

Other loop structures should be set up in a similarly easy to understand manner. One note about while and for loops: all branches which exit these loops should go to one place. Occasionally, this may mean branching to a branch statement. This is OK. We want code that is easy to read, not code that is completely optimized. Also, all branches to the beginning of the loop should go to the same place.

Design

As with other programming classes, design is very important in the implementation of your programs. Your procedures should be modular and general. They should perform one specific function and they should error-check and work for a wide variety of parameter values. One important mistake to avoid is accidental use of a 'global' variable. You should load the address of a memory location only once. If you need to use that variable in another procedure, you should either pass the address of the variable (passing by reference) or you should pass the value of that variable (pass by value.) You should not load the address of the variable in the other procedure. (Note that this restriction applies only to variables that would appear as variables in a high-level language. You are permitted, for instance, to load the address of a string you need to print whenever you need to print it; you don't have to pass around all your string addresses as parameters.)

Register Usage

Register usage is an extremely important topic in the style guide. The **a** registers should be used to pass parameters to subroutines and should not be saved on the stack across function calls. If you have more than four parameters, you should push the extras on the stack.

The **v** registers should contain the return values from the procedures. Obviously, these should not be saved across function calls.

The **s** registers can be used for anything. They should also be saved across procedure calls, so they have the same values as they started with after a procedure call. **You must save all s registers to the stack that you overwrite in procedure calls, even if you know that they will not be used elsewhere in your program.**

The **t** registers should be used for temporary calculations. They should not be saved across function calls, nor should you expect them to be saved. **Note: this includes syscalls as well!**

The **ra** register is a special register that contains the return address after a procedure call. It should be saved after every call, just in case you call another procedure which would overwrite the ra register with a new return address.

In summary, you should use the following registers, and only the following registers, and you must use them correctly (if you use other registers you will lose credit)

$a0 - $a3 - argument (for passing parameters)
$v0 - $v1 - return value
$t0 - $t9 - temporary registers - not preserved across calls

$s0 - $s7 - saved registers - preserved across function calls
$sp - stack pointer
$ra - return address

If you want a value to be available after you make a subroutine call, you should put it in an $s register - and remember that a syscall is a subroutine call!

Always use the first available register - if you have just entered a subroutine, and you want to use a saved register, you should always use $s0 first. If you want a temporary register, use $t0. (The point here is that you can't get away with the following: say I'm in main and I use $s0. Then I call a subroutine. I know that main hasn't used $s1, so I can save time and use $s1 in my subroutine, and I don't have to bother saving $s1. ALWAYS save $s registers. NEVER assume that a $t register will be valid after a subroutine call).

Never pass parameters in any register besides $a0 - $a3 (i.e., don't pass a parameter in a $t or $s register). Only pass parameters on the stack if there aren't enough $a registers (i.e. there are 4 $a registers, so if you have 5 parameters, one has to go on the stack)

Steps of a subroutine call (based on Patterson and Hennessy)

What the caller does, when it is ready to call a subroutine:
1. Pass arguments - put arguments in argument registers (registers $a0- $a3 should be used to pass arguments). If (and ONLY if) the argument registers are not enough (i.e. you have more than 4 arguments), push the remaining arguments on the stack.
2. Execute a jal instruction - jumps to callee and saves the return address in ra.

What the callee does when it is called (before doing anything else)
1. Push $ra (return address) onto stack.
2. Push any saved registers the callee will use onto the stack (if a subroutine doesn't use any saved registers, it doesn't have to save any on the stack, but even if you as the programmer know that the caller didn't use any saved registers, you still must save any registers that the subroutine uses).
3. Decrement stack pointer.

What the callee does before it returns (after it is done)
1. If subroutine is a function, put return value into $v0 register.
2. Increment stack pointer.
3. Restore saved registers that were saved at beginning of subroutine.
4. Restore return address to $ra.
5. Register-jump to location in $ra.

Global Variables

In assembly, while every variable is accessable or 'global' to the entire program you don't want to start accessing them from anywhere. Keep in mind that we want well structured assembly. This extends beyond loops and 1-in-1-out constructs. This means that you should restrict using variables as 'globals' only for special cases. Basically, the idea is you should never use a global variable instead of passing something as a parameter to a subroutine. If you think about it from a modeling view a subroutine should know about only what it is told it needs to know about. If it needs to know about some data named Bob, the caller should tell it about Bob by passing Bob's

address instead of the subroutine loading the address of Bob on its own. There are two exceptions to this rule.

1. Data that from a modeling point-of-view should be accessable to EVERYONE, such as constant strings.

2. Computed constants, because SPIM won't allow them but any real compiler will:

```
.eq const1 4
.eq const2 3
.eq const3 const1 * const2
```

This is not accepted by SPIM but we will allow you to get around it by using a global variable which you initialize very early in the program.

In case this long-winded theological mumbo-jumbo still leaves doubt in your mind, besides constant strings we really only want you to use one 'global' variable during your cs031 tour of duty. This mystical beast is the freelist in Asgn3:Questions. Everything else, you should be passing around like a 3-foot bong at a Phish show.

Sample Code

```
# a program that does meaningless math operations
# Note, the main point of this code is to demonstrate good register usage
# techniques. For a sample of good commenting of high level control
# structures, see the "exponentiate" subroutine

.data

# strings to print out messages to the user
first_text:.asciiz"\n Do stuff returned: "
second_text:.asciiz "\n Next result is: "
third_text:.asciiz "\n to the power of "
fourth_text:.asciiz " is "
fifth_text:.asciiz" ,times the value returned by dostuff is "
sixth_text:.asciiz"\n Multiply it all together to get: "
newline_string:.asciiz"\n"

            .eq print_string 4
            .eq print_number 1


.text


__start:

      li    $t0, 1            # I just need $t0, and $t1
```

```
li    $t1, 7            # to calculate $s0, so I use temporary register
add   $s0, $t0, $t1     # I want to have the

jal   dostuff           # go to dostuff, it'll dostuff

move  $s1, $v0          # save the return value - I'll want this later,
                        # so use $s register

la    $a0, first_text   # print some stuff
jal   printstr          # a subroutine that makes outputting a bit nicer

move  $a0, $s1          # print out the value we calculated before
jal   printnum

sub   $s2, $s1, $s0     # subtract the two values

la    $a0, second_text  # print out a string
jal   printstr

move  $a0, $s2          # print out the result of the subtraction
jal   printnum


jal   newline           # print a newline - another subroutine

li    $s3, 5            # get five ready for some operations

move  $a0, $s3          # print out base
jal   printnum

la    $a0, third_text   # Print out base string
jal   printstr

li    $s4, 7            # get seven ready for some operations

move  $a0, $s4          # print out exponent
jal   printnum

la    $a0, fourth_text  # print out exponent string
jal   printstr

move  $a0, $s3          # calculate the 5 to the power of 7
move  $a1, $s4
jal   exponentiate

move  $s3, $v0          # save the returned value - note, I don't care
                        # about the previous value of $s3 or $s4, so I
                        # can recycle them

move  $a0, $s3          # print the result
jal   printnum

la    $a0, fifth_text   # separator string
jal   printstr
```

```
        mul   $s4, $s3, $s1     # multiply result be value from dostuff
        move  $a0, $s4          # print it out
        jal   printnum

        la    $a0, sixth_text   # print multiply string
        jal   printstr

        mul   $s0, $s0, $s1     # compute the new value
        mul   $s0, $s0, $s2
        mul   $s0, $s0, $s3

        move  $a0, $s0          # print the value out
        jal   printnum

        done









#*****************************************************************
dostuff:


        sw    $ra, 0($sp)       # push ra
        sw    $s0, -4($sp)      # save s0
        sw    $s1, -8($sp)      # save s1
        sw    $s2, -12($sp)     # save s2
        sub   $sp, $sp, 16      # decrement stack pointer

        li    $s0, 5            # set up some intial values
        li    $s1, 17
        li    $s2, -2
        add   $t0, $s0, $s2
        mul   $t1, $s2, $s2

        move  $a0, $t0          # pass parameters -
        move  $a1, $t1          # exponentiate ( x, y)

        jal   exponentiate

        mul   $s0, $s0, $v0
        mul   $s0, $s0, $s1
        mul   $s0, $s0, $s2

        move  $v0, $s0          # return the result


        add   $sp, $sp, 16      # increment stack pointer
```

```
      lw    $s2, -12($sp)     # restore s2
      lw    $s1, -8($sp)      # restore s1
      lw    $s0, -4($sp)      # restore s0
      lw    $ra, 0($sp)       # pop ra
      jr    $ra               # return to caller



#*****************************************************************
exponentiate: # exponentiate (baseparam, exponentparam)

#**************
# raises baseparam ($a0) to the power of exponentparam ($a1)
#
#
# Register usage
# $a0 - first param - x (where exponentiate = x to the y)
# $a0 - second param - y (where exponentiate = x to the y)

# $t0 - number being raised to power - I'm gonna call it the base
# $t1 - counter - this is the exponent, see the algorithm I use
# $t2 - "accumulator" - it holds the accumulated value of the exponentiation
# $v0 - return value - holds the result
#**************

      sw    $ra, 0($sp)       # push ra
      sub   $sp, $sp, 4       # decrement stack pointer

                  # save parameters
                  # I don't call any subroutines, so I don't need $s registers
      move  $t0, $a0          # base = baseparam
      move  $t1, $a1          # count = exponentparam
      li    $t2, 1            # accumulator = ONE (initialize the acc)


powerloop:
      beqz  $t1, endexponentiate# if( ! count = zero ) {

      mul   $t2, $t2, $t0     #     accumulator = accumulator * base
      sub   $t1, $t1, 1       #     count = count - 1
      j     powerloop         #   }

endexponentiate:

      move  $v0, $t2          # return accumulator

      add   $sp, $sp, 4       # increment stack pointer
      lw    $ra, 0($sp)       # pop ra
      jr    $ra               # return to caller

#*****************************************************************

printnum:
      sw    $ra, 0($sp)       # push ra
```

```
      sub   $sp, $sp, 4       # decrement stack pointer

      move  $a0, $a0
      li    $v0, print_number
      syscall

      add   $sp, $sp, 4       # increment stack pointer
      lw    $ra, 0($sp)       # pop ra
      jr    $ra               # return to caller

#*****************************************************************

printstr:
      sw    $ra, 0($sp)       # push ra
      sub   $sp, $sp, 4       # decrement stack pointer

      move  $a0, $a0
      li    $v0, print_string
      syscall

      add   $sp, $sp, 4       # increment stack pointer
      lw    $ra, 0($sp)       # pop ra
      jr    $ra               # return to caller

#*****************************************************************

#*****************************************************************

newline:
      sw    $ra, 0($sp)       # push ra
      sub   $sp, $sp, 4       # decrement stack pointer

      la    $a0, newline_string
      jal   printstr

      add   $sp, $sp, 4       # increment stack pointer
      lw    $ra, 0($sp)       # pop ra
      jr    $ra               # return to caller
```