Assembly Language Convention Guide

Conventions are especially important in assembly code. Unlike higher-level languages, which provide inherent structure through branches, loops, and functions, assembly language provides almost no structure. As the programmer, you can structure your program in almost any way. Conventions must be followed so that your code is understandable (by yourself, your colleagues, your TAs), and so that your code interfaces properly with other parts of the program.

Conventions are more significant in assembly than in other languages. In a high-level language, if you don't name an instance variable 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 procedure called by someone else's code, and you use $s registers without saving and restoring them, their code will fail. By the same token, if you use $t register, call a procedure, and expect the value in the register to be there when the procedure returns, you might be sadly mistaken.

For these reasons, **assembly conventions are about correctness more so than style**. To reflect this, proper adherence to convention will comprise a significant part of the grade for any assembly assignment. Failure to adhere to convention will result in the loss of points.

Note that the primary goal of teaching you assembly is to teach you how high-level abstractions correspond to low-level constructs. Thus, we have chosen an assembly convention that emphasizes the correspondence with high-level code, rather than a convention that is the most efficient.

1 Commenting

Procedure Header

Every procedure must be preceded by a comment which contains:

- the procedure name
- the argument(s) and return value(s)
- a brief description of the procedure's function
- a register usage table (described below)

Register Usage Table

Every procedure must have a register usage table that lists what registers are used in a procedure and what they are used for. You should include all a, v, and s registers, though you don't have to include t registers which are only used temporarily.

Give each register a meaningful name, and use that name in your inline comments when referencing that register (see next section).

Here is a good example of a register usage table:

```
# Register usage:
#  $s0 - primes_left (number of primes left to find - starts
#                     at NBR_PRIMES and decrements until zero)
#  $s1 - n (next number to try - starts at 2 and increments by 1)
#  $s2 - i (loop counter)
#  $s3 - prime (pointer into primes array)
```

Inline Comments

Almost every line of assembly should have a corresponding inline comment to its right. The inline comment should contain the high-level equivalent of the assembly code. It should be as close to Java, C, or C++ as possible, though since none of these languages translate perfectly to assembly, you are allowed to invent notation as long as it's understandable. Since one line of high-level code may correspond to several lines of assembly, you only need one inline comment for such sequences of code. Note that most high-level code can be done in exactly one line of assembly, and rarely requires more than three lines.

Indent your inline comments as you would lines of a high-level language. However, the comment characters (#) should line up vertically. You should not indent your assembly code.

Sometimes your code will be complicated enough that it warrants a comment in English. You should still write the high-level code, but embed the English comment inside the high-level code.

All comments should *add* information. For example, give variables meaningful names instead of just using register names (use the same names as in the register table). Try to show relationships across lines, not just information contained in a single line.

Example of good commenting:

```
rem     $t1,    $s1,    $t0             #
bnez    $t1,    if_end                  # if (n % prime == 0) {
                                        #       // n is divisible by this prime
                                        #       // so n can't be prime
add     $s0,    $s0,    1               #       nbr_composites++;
if_end:                                 # }
```

Note:

- The first two lines of assembly can be represented in just one comment.
- The comment characters are lined up.
- The body of the if statement is indented.
- There is an embedded English comment represented by the Java/C++ // comment syntax.
- The variables have meaningful names.

Example of bad commenting:

```
rem     $t1,    $s1,    $t0             # $t1 = $s1 % $t0
bnez    $t1,    if_end                  # if ($t1 == 0) goto if_end;
add     $s0,    $s0,    1               # add one to register
if_end:
```

What's bad?

- The comment on line 1 is completely redundant. Anyone reading MIPS assembly already knows what the rem instruction does.
- The registers don't have meaningful names.
- The comment on line 3 is in English, not high-level code.
- The comment on line 2 uses goto, which is not a high-level construct. The comment should represent this code as an if statement, as described in the next section.

2 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 using branch instructions. The simplest flow of control statement is the IF-THEN(-ELSEIFELSE) 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 (x <= y) {
                      #     // 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 (x <= y)). It is often necessary to have this reverse logic in order to make the comments make sense. When you program assembly, you have to think *what do I test in order to* skip *the branch*, whereas in high-level code, you think *what do I test in order to* execute *the branch*.

elseif can be implemented with additional branch instructions, like this:

```
bge $t0, $t1, elseif  # if (x < y) {
                      #     // code if the value is smaller
b endif               # }
elseif:               #
```

```
bne $t0, $t1, else    # else if (x == y) {
                      #     // code if the value is equal
b endif               # }
else:                 # else {
                      #     // code if the value is bigger
endif:                # }
                      # // continue with the program
```

Loops can be implemented like an if statement, except that at the end of the loop body there is an unconditional jump back to the beginning of the loop. Example:

```
loop:                 #
bge $t0, $t1, endloop # while (x < y) {
                      #    // code inside loop body
j loop                # }
endloop:              #
                      # // continue with the program
```

break can be implemented using an unconditional jump to the end loop label, and continue with an unconditional jump to the begin loop label.

Note that all branches which exit 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.

All control structures must mirror higher-level conditionals, loops, or procedures. You should not use jumps for other purposes. For example, trying to replicate Java's exception handling by having an error condition trigger a jump straight to a separate section of error-handling code is not allowed.

3 Register Usage

The *a* registers should only be used to pass arguments to procedures and should not be saved on the stack across procedure calls, nor should you expect them to be saved. Never pass arguments in any other register. If you have more than four arguments, you should push (only) the extras on the stack.

The *v* registers should contain the return values from the procedures. These are not saved across procedure calls, nor should you expect them to be saved. Do not use any other register for returning values.

The *s* registers can be used for holding local variables. They should be saved across procedure calls. 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 procedure calls, nor should you expect them to be saved.

The *ra* register is a special register that contains the return address after a procedure call. You should save it at the beginning of every procedure and restore it at the end.

Note: syscalls are like procedure calls and may trash non-*s* registers.

In summary, you should use the following registers, and only the following registers, and you must use them correctly:

`$a0 - $a3`	procedure arguments
`$v0 - $v1`	return value
`$t0 - $t9`	temporary registers - not preserved across calls
`$s0 - $s7`	saved registers - preserved across calls
`$sp`	stack pointer
`$ra`	return address

4 Procedure calling convention (based on Patterson and Hennessy)

Follow these conventions for *all* procedures, no matter how small. Also follow these conventions for `main`.[1]

What the caller does, when it is ready to call a procedure

1. Pass arguments - put arguments in argument registers (use registers $a0–$a3, in order). 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. Decrement stack pointer.
2. Push $ra (return address) onto stack.
3. Push any saved registers the callee will use onto the stack (if a procedure doesn't use any saved registers, it doesn't have to save any on the stack, but even if you as the programmer think that the caller didn't use any saved registers, you still must save any registers that the procedure uses).

What the callee does before it returns (after it is done)

1. If procedure returns a value, put the return value into $v0 register.
2. Restore saved registers that were saved at beginning of procedure.
3. Restore return address to $ra.
4. Increment stack pointer.
5. Jump to location in $ra.

[1] `main` is called from another procedure inside the system libraries that would get messed up if you did not properly save registers.

5 The Stack

When you return from a procedure, the stack pointer *must* have the same value as when you entered the procedure. If it has a different value, it will cause the caller serious problems.

The stack pointer ($sp) should always point to the next *available* word on the stack. Pay attention to this when you allocate a variable on the stack. In general, you should allocate stack variables by first decrementing $sp by the size of the variable/array. Once you have done this, your freshly allocated variable starts at `$sp + 4`.

Never assume that the stack is initialized to all zeros (even in `main`). It is probably polluted with data from a previously-called procedure.

Note that you always decrement the stack pointer *before* pushing to the stack, and always increment it *after* popping. Although your code will work correctly in MipScope if you do it the other way around, it may fail on a real computer.[2]

6 Global Variables

Although all registers and data section variables are accessible throughout your entire program, you can't access them from wherever you like. We want you to write structured assembly that corresponds to higher-level code. Only use data section variables for data that is truly global to the entire program. In particular, never use them as a substitute for proper argument passing using *a* registers and the stack.

Never use a register to store a global value. For example, it is incorrect to place a value in $s0 at the beginning of the program and then using that value throughout any procedure which are called. The only registers a procedure can assume are properly set when the procedure begins are $sp, $ra, and any *a* registers which are used for arguments.

You may always use global integer constants (e.g. `COLUMNS = 5`). You may also use the data section to store constant strings (`.asciiz` data).

[2]On a real computer, your code may be interrupted at any point to execute a signal or interrupt handler. This handler, like any procedure, will itself use the stack. If you have written to the stack before decrementing the stack pointer, and an interrupt fires before you have updated the stack pointer, the interrupt handler will overwrite what you have just written. When the interrupt handler returns, your stack will be filled with random junk. This is a VERY difficult bug to track down because it will happen completely randomly (you never know when an interrupt will happen), and usually won't happen in a debugger because of this randomness (this is dubbed a "Heisenbug"). Suffice it to say, it's best to just do it the right way.

7 Objects

Note: this section is not relevant until Project Maze.

Methods

Method procedure names should start with a common prefix that indicates their class. For example, the `visit` method inside class `room` should be named `room_visit`.

Method procedures must always take the pointer to the object (`this`) as the first argument.

Useful Constants

At the top of your assembly file, you must declare the following constants for each class:

- *`classname`*`_size` - the number of bytes in an object of this class. It should be the sum of the sizes of each field, plus the size of the vtbl pointer.
- *`classname_fieldname`* - For every field in the class, you must declare a constant that specifies the offset to that field.
- *`clasname`*`_vtbl_`*`methodname`* - For every method in the class, you must declare a constant that specifies the offset to the method's entry in the vtbl.

You should use these constants instead of hard-coding their values in the code.

vtbls

Vtbls should be declared in the `.data` section as follows:

`__`*`classname`*`_vtbl:  .word `*`method1, method2, method3, ...`*

`method1`, `method2`, `method3`, ... are the names of the class's method procedures. The MIPS assembler will convert the names to addresses and create an array containing pointers to each procedure.

Construction

You must write the following procedures for every class:

- `make_`*`classname`* - The first argument is a pointer to the object, and the remaining arguments are arguments to the constructor. This procedure should set the object's vtbl pointer and then call `construct_`*`classname`*.
- `construct_`*`classname`* - This is the class's constructor and should initialize the class's fields. The first argument is a pointer to the object. If it's a subclass, remember to call the superclass's constructor.

To instantiate an object:

1. Allocate memory for the object (use the *`classname`*`_size` constant). This could be done on the stack or with a call to `malloc`.
2. Call `make_`*`classname`*. The first argument should be the address of the memory you just allocated. The remaining arguments should be the arguments to the constructor.

Why so many steps?

1. Memory allocation is distinct from initialization: you should be able to initialize an object the same way regardless of whether it was allocated on the stack or on the heap.
2. Constructors don't know the exact type of the object being constructed (in the case of superclass constructors). Therefore, the vtbl needs to be set *before* calling the constructors, necessitating both a *make* procedure and a *construct* procedure.

In Java, steps similar to this occur when you use `new`.

8 Strings

All strings should be null terminated. This is most easily accomplished by declaring them using `.asciiz`. This automatically allocates one extra byte at the end of the string and puts a null (0 byte) into that position. If you declare a string using `.ascii` instead it will not include an extra byte and you will see unexpected behavior. (For example, the print_string syscall will not stop at the end of the string and will print garbage data until it reaches a null byte.)

9 Efficiency

As with any code, efficiency is important and we want you to write efficient assembly. *However*, we also want you to write correct code and understand how high-level concepts translate to assembly. When these goals conflict, correctness and convention take priority.

So, it's not OK to replace a nicely-structured if statements with some gotos, nor is it OK to skip register saving because you're pretty sure your caller never uses those registers anyways.

However, we should think about memory accesses and avoid unnecessary loads and stores. For example, if you're loading a memory address at the beginning of a loop body, and then storing it at the end, you could use a register instead and only load once before the loop and store once at after the loop.

If you have any doubts, talk to a TA.

10 Sample Code

This program, which calculates prime numbers using the Sieve of Eratosthenes, is an example of perfect commenting and adherence to convention. It also demonstrates proper saving and restoring of registers.

The code can be found in /course/cs031/pub/primes.s.

```
# primes.s
# Description: Calculate prime numbers using the Sieve of Eratosthenes,
#      Author: andrew

        .data
        NBR_PRIMES = 4096
primes:     .word   0:NBR_PRIMES
newline:    .asciiz "\n"        # Newline character, for convenience

        .text
main:
# Register usage:
#  $s0 - primes_left (number of primes left to find - starts
#                     at NBR_PRIMES and decrements until zero)
#  $s1 - n (next number to try - starts at 2 and increments by 1)
#  $s2 - i (loop counter)
#  $s3 - prime (pointer into primes array)
#####################################
    sub     $sp,    $sp,    20      #
    sw      $ra,    20($sp)         #
    sw      $s3,    16($sp)         #
    sw      $s2,    12($sp)         #
    sw      $s1,    8($sp)          #
    sw      $s0,    4($sp)          #
                                    #
    li      $s0,    NBR_PRIMES      # int primes_left = NBR_PRIMES;
    li      $s1,    2               # int n = 2;
                                    #
outer_begin:                        #
    blez    $s0,    outer_end       # while (primes_left > 0) {
    li      $s2,    NBR_PRIMES      #   int i = NBR_PRIMES;
    la      $s3,    primes          #   int* prime = primes;
inner_begin:                        #
    blez    $s2,    inner_end       #   while (i > 0) {
    lw      $t0,    ($s3)           #       int p = *prime;
    bnez    $t0,    ifelse          #       if (p == 0) {   // New prime found
    sw      $s1,    ($s3)           #           *prime = n;
    sub     $s0,    $s0,    1       #           --primes_left;
                                    #
    move    $a0,    $s1             #
```

```
    li      $v0,    1               #
    syscall                         #           print_int(n);
                                    #
    la      $a0,    newline         #
    li      $v0,    4               #
    syscall                         #           print_string("\n");
    j       inner_end               #           break;
ifelse:                             #       }
    rem     $t1,    $s1,    $t0     #
    bnez    $t1,    if_end          #       else if (n % p == 0) {
                                    #           // n is divisible by this prime
                                    #           // so n can't be prime
    j       inner_end               #           break;
if_end:                             #       }
    add     $s3,    $s3,    4       #       ++prime;
    sub     $s2,    $s2,    1       #       --i;
    j       inner_begin             #
inner_end:                          #   }
    add     $s1,    $s1,    1       #   ++n;
    j       outer_begin             #
outer_end:                          # }
    lw      $s0,    4($sp)          #
    lw      $s1,    8($sp)          #
    lw      $s2,    12($sp)         #
    lw      $s3,    16($sp)         #
    lw      $ra,    20($sp)         #
    add     $sp,    $sp,    20      #
    jr      $ra                     # return;
#####################################
```