ECE/CS 252 Programming Style Guide

Using good programming style and properly documenting your code is a **critical** element of the programming process. This not only makes it easier for others to read and understand your code, but it also helps **you** while you are writing, debugging, or re-using your code.

Too often, novice programmers wait until after their code is complete to go back and add comments. If you do this, you gain absolutely nothing from the documentation process and have **missed the point**.

DO NOT WAIT UNTIL YOU ARE "DONE" WRITING CODE TO COMMENT IT!

Documenting your code ***as you go*** and ***in your own words*** forces you to organize your thoughts and continuously examine your program's logic and methodology. This helps you to see errors and inconsistencies in your program before the debugging phase. During debugging, comments help you remember what you were trying to do so that you can verify if your program actually does that.

When writing comments, remember that the purpose is to explain **why** a line of code is there, not to reiterate exactly what that instruction does! Comments should make it easy for someone who already knows the language to understand your thought process and design decisions. Descriptive label names help document programs, and is a must for this course (and good programming).

Finally, **neatness counts**! Neatly formatted and organized code is much easier to understand, debug, and reuse.

The next page lists code requirements for all programs, followed by additional requirements for subroutines in particular. This is followed by examples of bad vs. good style. Be sure to look at these so that you can see the difference, and think about which **YOU** would rather look at!

Code Requirements

- Make your code neat and easy to look at.
    - Instructions should be indented (to differentiate from labels) and should "line up". Comments should also "line up". See the examples of good practices.
- Use useful label names. Labels are an important form of documentation!
- Comments should explain the **purpose** of instructions (what they do in terms of the overall program), not simply restate the actual operation of the instruction.
    - Comments like "add 1 to R3" are not useful
- Block-level comments and blank lines should be used where helpful to "group" related instructions.
- Programs should include a header (example given below) with the file name, author name, and a description of the program.

```
; Filename:     hw09.asm
; Author:       <student name>
; Description:  Finds the maximum of eight numbers stored in memory
```

Additional Requirements for Subroutines

- Subroutine names (labels) must be meaningful and provide information about their purpose.
- All labels in a subroutine must be prefixed by the name of the subroutine (see the subroutine "good style" example), including those used for branch targets, local variables, and register save/restore space.
- Each subroutine must contain a single RETurn instruction, and it must be the last instruction of the subroutine (not in the middle…), but before any subroutine data and/or register save/restore space.
    - Branch to the single RETurn if the subroutine needs to return from multiple points of its code. The "good style" subroutine example shows how to do this.
- All register save instructions should be at the very beginning of the subroutine, and all register restore instructions should be right before the RETurn instruction.
    - Do not mix these in with the rest of the code – mixing makes it easy to make a mistake.
    - If you need to save/restore more registers, there's only one place you need to add a save instruction, and one place to add a restore instruction!
- If a subroutine needs to have local variables (those accessed only by the subroutine) and/or register save/restore space, those memory locations must be located after the subroutine's RETurn instruction (but before any following subroutine).
    - Do not "mix" subroutine data or register save/restore space for multiple subroutines, and do not separate subroutine code and data by putting the data by the main program!
    - Register save/restore labels must include both the subroutine name and register number (to help avoid confusion and mistakes!).
    - Do not interleave subroutine variables and register save/restore space.
- Subroutines must have comment headers as shown in the example below. The header indicates what the subroutine does, and what information is passed/returned in each register.

```
; Subroutine:   RotateStepper
; Description:  Rotates stepper motor the specified number
;               of revolutions in the specified direction
; Assumes:      R0 – direction (CW if R0 > 0, CCW otherwise)
;               R1 – revolution count
; Returns:      None
```

Program Example: Bad vs. Good Style Comparison

This program reads a value from memory, negates it, then writes it back to the original location (overwriting its previous value).

BAD STYLE

```
.ORIG x0200
START LD R1, LABEL1 ; perform memory load
NOT R1, R1 ; complement R1
ADD R1, R1, #1 ; R1 <- R1 + 1
ST R1, LABEL1 ;memory write
BR START ; branch
LABEL1 .FILL #42 ; forty-two
.END
```

The above version of the code has been poorly formatted and poorly documented, so is difficult to understand or read. **Do not comment your code this way!**

Some key deficiencies are:

- Program lacks "header" comments that explain what the program does.
- The generic label "LABEL1" is not helpful, and is a missed opportunity for documentation. This approach is even worse if a program contains multiple labels.
- Comments on each instruction only restate what each instruction does without explaining **why**.
- Poor formatting makes the program extremely difficult to look at.
- Lack of separation (blank line) between program code and program data makes it harder to notice the data unless you're specifically looking for it.

GOOD STYLE

```
; File name:   negate.asm
; Author:      <author's name>
; Description: Negates a value in memory
.ORIG x0200

START
      LD  R1, VALUE     ; get value to negate
      NOT R1, R1        ; negate value
      ADD R1, R1, #1
      ST  R1, VALUE     ; write negated value back to memory
      BR  START         ; repeat forever

      ; program data
VALUE .FILL #42

.END
```

The above is the same program, but with better style and documentation. This version is far more understandable. Comments explain the **purpose** of the instructions, and formatting makes the code easier to read (labels and comments are separated from instructions, and everything lines up to make it neat and tidy). The program's data is also separated from the program code so that it is more noticeable.

Note: this is a very simple program – a more complex program with multiple data values would need even more descriptive labels to explain the purpose of each data value (as shown in the subroutine example), and possibly comments to explain the purpose of each variable if the purpose was not completely clear from the labels.

Subroutine Example: Bad vs. Good Style Comparison

This subroutine determines if an ASCII character passed to it is lowercase.

<u>BAD</u> STYLE

```
subroutine1  ; this is a subroutine
ST R0,SAVE1 ; save to memory
LD R0, VALUE ; get value from memory
  ADD R0, R1, R0 ; r0 gets r1 plus r0
BRn LABEL1 ; if negative, go to LABEL1
LD R0, VALUE2 ; get value from memory
 ADD R0, R1, R0 ; R0 <- R1 + R0 again
BRnz LABEL2; go to LABEL2 if negative/zero
LABEL1  AND R1, R1, #0 ; R1 <- 0
 LABEL2 LD R0, SAVE1  ;load from memory
RET ; return
SAVE1 .BLKW  1 ; allocate memory
VALUE .FILL  xFF9F ; a value
VALUE2 .FILL  xFF86 ; another value
```

Extremely poor style makes this subroutine nearly unreadable. Even though every line has been commented, those comments are useless because they do not provide information beyond the instructions themselves.

Some key deficiencies are:

- The subroutine name (*subroutine1*) says absolutely nothing about the subroutine's purpose.
- Code lacks subroutine "header" comments to explain what the subroutine does and how to use it.
- The labels are too generic. "SAVE1" does not say **<u>what</u>** will be saved, and the two "VALUE" labels do not differentiate their purposes. These are lost opportunities for labels that help document code!
- Comments do not convey any information that isn't already obvious from the code.
- Code is poorly formatted, making it hard to look at, understand, modify, debug, etc.

<u>GOOD</u> STYLE

```
; Subroutine:  islower
; Description: Determines if a character is lowercase
; Assumes:     R1 - character to check
; Returns:     R1 – orig. character if lowercase, 0 otherwise
islower
   ST   R0, islower_R0       ; context save

   ; check if character is less than ‘a’
   LD   R0, islower_nega
   ADD  R0, R1, R0
   BRn  islower_false        ; if so, not lowercase (return 0)

   ; check if character is less than or equal to ‘z’
   LD   R0, islower_negz
   ADD  R0, R1, R0
   BRnz islower_exit         ; if so, is lowercase (return char)

islower_false
   AND  R1, R1, #0           ; makes subroutine return 0 (false)

islower_exit
   LD   R0, islower_R0       ; context restore
   RET

   ; subroutine data
islower_nega  .FILL  xFF9F   ; negative ASCII ‘a’
islower_negz  .FILL  xFF86   ; negative ASCII ‘z’
   ; context save/restore space
islower_R0    .BLKW  1
```

This version uses much better style. Label names and comments are far more useful. The formatting is also far better, making it much easier to read. **As part of making code modular and easy to read, note that each subroutine's data and register save space should go right after the RET for that subroutine!**