An Assembly Language Programming Style Guide

by Ken Lambert

The purpose of these style guidelines is to help the assembly language programmer to write readable, well-documented, maintainable programs.

In a high-level programming language like Python, many short, complete programs can be written almost as quickly as the programmer can type. That is obviously not the case with assembly language. The assembly language programmer often must painstakingly translate a high-level program design by hand to low-level assembly language code. During this process, any signs of a high-level design in the resulting code may easily disappear, assuming they were even present to begin with, leaving a mangled disarray of code.

In Python programming, the use of consistent formatting, conventional naming schemes, and appropriate documentation are essential elements of programming style. These elements are even more critical in an assembly language program. A significant program might be read months, if not years, after it is written, and the reader likely will not be its original author. Good program style can dramatically cut the amount of time it takes for a reader to understand code, and can significantly enhance the reading experience.

While you cannot ever make an assembly language program look just like a well-written Python program, your aim should be to make your readers, including yourself, feel almost as much at home in an assembly language program as they would in a Python program.

Naming Conventions

Unlike Python code, assembly language code is not case sensitive. But you should spell all opcodes, register symbols, labels, and assembler directives using UPPERCASE letters. This usage will help the reader pick out the actual program code from the surrounding comments, which should use sentence case. The only exception to the uppercase rule for symbols is the conditional branch (**`BR`**), which uses lowercase to indicate positive, negative, zero, or any combination (**`BRzp`**).

The names of data labels should reflect their purpose and role in the program. This is also the case for the names of instruction labels. For example, **`SIZE`** is a good name for the number of data values currently stored in an array, whereas **`GCD`** is a good name for a subroutine that computes the greatest common divisor. **`WHILE`** and **`ENDWHILE`** are good names for labels for the beginning and end of a **`while`** loop (but only if there is at most one such loop in your program!).

Use of Whitespace

Each instruction should appear on a single line. The component parts of each instruction should be in columns aligned with the same components of the instructions above and below it. In general, labels should appear in the first column, opcodes or assembler

directives in the second column, operands in the third column, and end-of-line comments in the fourth column. At least one tab should separate two columns. Multiple operands should include a single space after each comma.

Not every line of code requires an end-of-line comment, but those lines whose meanings are less than obvious certainly do. We'll see some examples shortly.

Well-placed blank lines between different functional parts of a program can really help the reader. For example, you should put at least one blank line between the instructions and the declarations of the data labels on which those instructions operate.

The Structure of Simple Assembly Language Programs

A simple assembly language program, like a simple Python script, consists of a set of statements and data declarations, but no subroutine definitions. The code itself might amount to 6-20 lines of text; but you should include other text, in the form of program comments, to describe its purpose and clarify any maneuvers or tactics that might seem obscure. To aid in establishing such a program format, here is a template to follow for structuring short, simple programs:

```
<prefatory comment>

<pseudocode design>

<comment on register usage>

<program instructions>

<program data declarations>
```

Now here is a short program that is structured according to this template, followed by a discussion of each of the structural pieces:

```
;; Author: Ken Lambert

;; This program resets the value of the variable NUMBER
;; to its absolute value

           .ORIG x3000

;; Pseudocode design:

; if number < 0
;    number = -number

;; Main program register usage:
;  R1 = number

; Main program code
           LD    R1, NUMBER
           ADD   R1, R1, #0       ; if number < 0
           BRzp ENDIF
           NOT   R1, R1           ;     number = -number
           ADD   R1, R1, #1
           ST    R1, NUMBER
ENDIF      HALT

; Data for the main program
NUMBER     .BLKW 1

           .END
```

Prefatory Comment

The prefatory comment should always include the author's name. For a homework project, this comment should include the exercise number as well. The comment should also include a brief statement of what the program does.

Pseudocode Design

Each major component of a program should begin with a comment containing its pseudocode design. Pseudocode is a language for describing algorithms that looks a lot like Python. Note that even this simple example program has a pseudocode design.

Register Usage

Following the pseudocode design, each major component of a program should have a comment that lists the usage of the registers in that component. This comment answers such questions as what values will the registers hold, and what roles in that program component will they serve? Remember that registers are like temporary variables in a Python function, but the reader needs a "key" to interpret them properly.

Using Whitespace Wisely

Note the use of blank lines, columns of whitespace, and comments to mark the major components of the example program. You would not need this in a Python program (except for the syntactically significant indentations), but it's imperative here.

The Structure of Complex Assembly Language Programs

A complex assembly language program, like a complex Python script, consists of a set of main program statements and data declarations and one or more subroutine definitions used by the main program or other subroutines.

The program format of a complex assembly language program is just an extension of the format of a simple program discussed earlier. The format of the main program instructions is exactly as it was before, but now each subroutine is listed below the data for the main program and appears in a somewhat similar format.

Here is the template for the structure of a program with subroutines:

```
<prefatory comment>

<pseudocode design of main program code>

<comment on register usage by main program code>

<main program instructions>

<main program data declaration>

<comments, instructions, and data declarations for subroutine-1>
.
.
.
<comments, instructions, and data declarations for subroutine-n>
```

The next program example revises the earlier example by defining and calling a subroutine to compute a number's absolute value.

```
;; Author: Ken Lambert

;; This program resets the value of the variable NUMBER
;; to its absolute value, using the ABS subroutine

;; Pseudocode design:

; number = abs(number)

      .ORIG x3000

;; Main program register usage:
;  R0 = number

; Main program code
      LD    R0, NUMBER       ; Set argument for abs
      JSR   ABS
      ST    R0, NUMBER       ; Use returned value
      HALT

; Data for the main program
NUMBER     .BLKW 1

;; Subroutine ABS
;  Converts the number in R0 to its absolute value
;  Parameter R0 = number to test
;  Return value R0
ABS   ADD   R0, R0, #0       ; if number < 0
      BRzp  ENDABS
      NOT   R0, R0           ;     number = - number
      ADD   R0, R0, #1
ENDABS     RET

      .END
```

Note that the subroutine's comments and code follow the main program's code and data.

Subroutine Comments

The prefatory comment of a subroutine should include

- its name
- a brief statement of what it does
- a list of its parameters and their roles (these will be registers, never data labels)
- a list of its return values (these too will be registers, never data labels)

Subroutines and Data Labels

A subroutine may declare its own data labels, although there are none in the current example. These data labels are like temporary variables in Python, and are intended for temporary working storage for that subroutine only. We will see their legitimate use in later examples.

With very rare exceptions, a subroutine should NEVER use or modify the contents of a data label declared in the main program or in another subroutine. This is a recipe for bug infections, maintenance headaches, and subroutines that cannot be ported to other programs. Like Python functions, assembly language subroutines should operate only on their own data labels and on registers designated as parameters or as temporary storage.

Using Registers for Parameters and Return Values

As in Python, data should be passed to a subroutine via parameters only, and in assembly language these should always be registers (unless we're supporting recursive routines with a system stack). Likewise, data to be returned from a subroutine should also be in registers.

Preserving Input-Only Parameters to Subroutines

A simple subroutine of one parameter, like **`abs`** shown earlier, uses the same register for the parameter and the return value. While this is economical and makes some sense for this subroutine, it does not really mimic the safety and security of the parameter-passing mechanism of Python functions.

Python functions pass their parameters ***by value***. This means that temporary storage is allocated for the parameter's value, so that the caller's original storage cannot overwritten by assignment within the function's code. For example, the function call **`sqrt(n)`** in Python cannot not modify the caller's variable **`n`**, even though the code within the function can do whatever it pleases with the corresponding parameter name.

It often makes sense to return a value in a different register than the one used for the parameter. Functions with two or more parameters are a case in point, as is a function of one parameter in which the returned value is of a different type than the argument.

Generally, it is best to treat the registers/parameters through which a subroutine receives data from its caller as ***input-only***. This means that when the subroutine returns, the contents of such registers will appear to the caller to be unchanged. That does not mean that the subroutine cannot modify these registers; but if it does so, it must first save their contents in temporary variables. These data must then be restored from the variables to the registers before the subroutine returns. This discipline is known as ***callee saves***.

Any other registers not designated as parameters or return values must also be backed up within a subroutine before they are used, and then restored before the routine returns. Such registers might serve as temporary working storage within the routine, but might also be holding data used by the routine's caller for its own purposes.

The next example shows a subroutine named **SUM**, which returns the sum of the values between a lower bound and an upper bound, inclusive. Note that the input parameters for the lower and upper bounds, **R1** and **R2**, are input only, and the output only parameter **R3** holds the sum. **R2** and **R4** serve as working storage within the routine, so their original contents must be saved and restored. Here is the code for the subroutine:

```
      .ORIG x3000

;; Pseudocode design:

; theSum = sum(lower, upper)

;; Main program register usage:
;  R1 = lower bound
;  R2 = upperbound
;  R3 = sum of numbers in the sequence

; Main program code
           LD    R1, LOWER
           LD    R2, UPPER
           JSR   SUM
           ST    R3, THESUM
           HALT

; Data for the main program
LOWER      .FILL 1
UPPER      .FILL 5
THESUM     .BLKW 1

;; Subroutine SUM
;  Returns the sum of the numbers between lower and upper bounds
;  Parameters R1 = lower bound
;             R2 = upper bound
;  Return value R3 = the sum of the sequence
;  Working storage R4 used for comparisons
SUM        ST    R2, R2SUM       ; Back up the registers
           ST    R4, R4SUM
           AND   R3, R3, #0      ; sum = 0
WHILESUM   NOT   R4, R1          ; while upper - lower >= 0
           ADD   R4, R4, #1
           ADD   R4, R2, R4
           BRn   ENDSUM
           ADD   R3, R3, R2      ;     sum += upper
           ADD   R2, R2, #-1     ;     upper -= 1
           BRnzp WHILESUM
ENDSUM     LD    R4, R4SUM       ; Restore the registers
           LD    R2, R2SUM
           RET

;; Data for subroutine sum
R2SUM      .BLKW 1
R4SUM      .BLKW 1
```

Note that the end-of-line comments take the form of pseudocode, reflecting the correspondence between the high-level algorithm and the assembly language code that implements it.

Finally, note the naming scheme for the data and instruction labels in the subroutine. Each label uses the suffix "sum," to avoid name conflicts with similar labels in other subroutines.

Passing Arrays as Parameters

Arrays are usually passed to subroutines by using two parameters. One is a register containing the array's base address. The other is a register containing the logical size, or number of data values to be processed.

A loop pattern might move through the array by incrementing the base register. Before that ever happens, this register must be backed up. Later, before the routine returns, the base register must be restored.

To illustrate these points, the next example shows a subroutine named **MIN**, which returns the minimum value in an array:

```
      .ORIG x3000

;; Pseudocode design:

; theMin = min(array, logicalSize)

;; Main program register usage:
;  R1 = base address of array
;  R2 = logical size of array
;  R3 = sum of numbers in array

; Main program code
           LEA   R1, ARRAY
           LD    R2, SIZE
           JSR   MIN
           ST    R3, THEMIN
           HALT

; Data for main program
ARRAY      .BLKW 10
SIZE       .FILL 5
THEMIN     .BLKW 1

;; Subroutine MIN
;  Returns the minimum value in an array
;  Assumes that the logical size >= 1
;  Parameters R1 = the base address of the array
;             R2 = the logical size of the array
;  Return value R3 = the minimum value in the array
;  Working storage R4 and R5 used for comparisons
```

```
MIN         ST    R1, R1MIN        ; Back up the registers
            ST    R2, R2MIN
            ST    R4, R4MIN
            ST    R5, R5MIN
            LDR   R3, R1, #0       ; min = array[0]
WHILEMIN    ADD   R2, R2, #-1      ; while logical size > 0
            BRz   ENDMIN
            ADD   R1, R1, #1       ;     index += 1
            LDR   R4, R1, #0       ;     temp = array[index]
            NOT   R5, R4           ;     if min > temp
            ADD   R5, R5, #1
            ADD   R5, R3, R5
            BRnz  WHILEMIN
            ADD   R3, R4, #0       ;         min = temp
            BRnzp WHILEMIN
ENDMIN      LD    R5, R5MIN        ; Restore the registers
            LD    R4, R4MIN
            LD    R2, R2MIN
            LD    R1, R1MIN
            RET

;; Data for subroutine MIN
R1MIN       .BLKW 1
R2MIN       .BLKW 1
R4MIN       .BLKW 1
R5MIN       .BLKW 1
```