Guidelines for ARM Assembly Code

1. Comments
   (a) The comment block at the top of each file should indicate the file name (helpful if it is printed), authors, and a general description of the function or purpose of the code. In the real world you would also document revisions here.
   (b) Comments for variables and constants may be in a common comment block that precedes all of the declarations, or each identifier may be defined with a single line comment that is just above the declaration or an in-line comment after the declaration.
   (c) Never use trivial comments like this:
       
       ```
       add r8, 1 ; Add 1 to R8
       mov r0, r2 ; Copy R2 to R0
       ```
       
   (d) Comments for variable and constant names should do more than restate the obvious. Describe the meaning of the data. In-line comments are shown here but the same idea applies to block comments.
       - Worst:
         ```
         foobar: .word
         .equ barfoo, 127
         ```
       - Bad:
         ```
         foobar: .word ; foobar is stored here
         .equ barfoo, 127 ; barfoo is 127
         ```
       - Better:
         ```
         InputData: .word ; Audio input, right channel
         .equ DOOR_SWITCH, 0x1234 ; Access door tamper switch address
         ```
       - Best:
         ```
         RtAudioInput: .word ; 16-bit right chan audio in, signed
         .equ DOOR_SWITCH, 0x1234 ; Access door tamper switch addr, 1=open
         ```
   (e) For executable code, a separate comment block can properly document no more than about half a page of code. If in-line comments are used then you should expect to add a comment for every 3 to 6 lines of code.

2. Identifiers
   (a) Above all, identifiers should be descriptive.
   (b) Identifiers for constants should be all uppercase with underscores separating words.
   (c) All other identifiers should be mixed case with capitalization used to indicate words, or lower case with underscores between words. However, you should use one style consistently throughout your code.
   (d) Names of variables and constants should be noun phrases that indicate the meaning of the data (AudioIsMuted, audio_is_muted, bottlesOnTheWall, MEANING_OF_LIFE).
   (e) Branch-target labels, as well as names of code modules, subroutines, and functions should be verb phrases that indicate the function performed (WaitForPayment, ejectPassenger).
   (f) Some identifiers violate these guidelines for historical reasons (e.g. _start, _main). Some identifiers violate these guidelines because they had preexisting definitions (e.g. GPIO3DATA, HardFault_Handler). So it goes.
   (g) Use symbols for non-trivial constants in your code:

       ```
       movw r8, #15000 ; This is bad!
       movw r8, #DELAY_COUNT ; This is better
       ```
movw r8, 1  ; Trivial constants are OK

(h) Delete loop labels, .equ directives, and .word directives if they are not used.
(i) Don’t specify memory addresses for variables or constants directly in your code. Use a label and the proper directive to allocate memory and provide a symbol with the memory address.

3. Indentation should be consistent. Labels start in the first column. Directives (e.g. .equ) and instructions (e.g. LDR) should have the same level of indentation. Use spaces or tabs but try not to mix them for the same kinds of indentation. If you have one section of code that defines all of your variables, and another that defines all of your constants, and another that is executable code, then the indentation in these sections may be different as long as there is a clear visual barrier between the sections. A clear visual barrier is something like a few lines of whitespace or a long line of comment characters (;;; ... ;;;).

4. Avoid using registers unnecessarily when an immediate operand will work just as well and be more self-documenting. Replace this:

```assembly
mov r8, #BITMASK
orr r0, r8
```

with this:

```assembly
orr r0, #BITMASK
```

5. Avoid convoluted branches like this:

```assembly
beq someplace
b someplaceElse
someplace:
```

and replace the two branches with one:

```assembly
bne someplaceElse
someplace:
```

6. Don’t use an arithmetic instruction (e.g. ADD or SUB) when you really want a bit-wise logical instruction (e.g. ORR or BIC).

7. When doing bit-banging, be careful that you only modify the bits that you need to modify and no others.

8. Use the left-shift operator to make your bit masks self-documenting and less error-prone:

```assembly
.equ BUTTON1, (1 << 9); Button is on bit 9 of port 2
```

9. It’s usually best to let the assembler decide whether to use a 16-bit or 32-bit encoding for an instruction. If you use a .N or .W width specifier, clearly document your reasons.