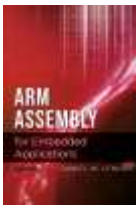


ARM Assembly for Embedded Applications, 4th edition
ISBN 978-1-54393-624-7

CHAPTER 4: COPYING DATA

Section 4.7 (Addressing Modes), page 65, paragraph titled "PC-Relative addressing":

The use of PC-Relative addressing with the STR instruction and its variants (STRB, STRH, STRD) has been deprecated, similar to the use of PC-Relative addressing with the VSTR instruction as described in Section 9.4.4. Although there are several examples such as `STR R0,x` found in chapter 4, they would actually be rejected by the assembler. To store into a memory location using its label requires a two-instruction sequence such as `ADR R1,x` or LDR R1,=x followed by `STR R0,[R1]`. The ADR instruction is only able to reference locations that are within 4095 bytes of the ADR, which would usually mean that the referenced location resides in flash memory (and therefore cannot be modified during execution). To use a label to reference data in read/write memory thus requires the LDR R1,=x instruction.

Note that since the text uses assembly to write small functions called from a C main program, there is rarely (if ever) a need for PC-Relative addressing because the operands of such functions are typically only the function parameters that are passed to the function in registers and the result is left in a register.

Page 54, section 4.1, end of 1st paragraph: ASCII constants are written with a single leading apostrophe, as in 'A. Note that this is different from how they are written in C (e.g., 'A'). C-style character constants also work ('a'), but not all escape sequences ('\0 or '\0') do. It's safer to use hex (0x00) instead.

CHAPTER 6: MAKING DECISIONS AND WRITING LOOPS

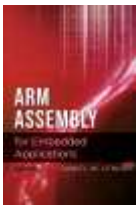
Page 105-106: Replace the last paragraph and subsequent code by the following:

Since a compare is simply a subtraction that discards the difference but records the characteristics of that result in the flags, one might assume that all that is needed is to simply perform a 64-bit subtraction and check the resulting flags. However, although the `N`, `V` and `C` flags will be correct, the `Z` flag may not since it will only indicate if the most-significant half of the difference is zero.

When the condition to be tested does *not* depend on the `Z` flag (`GE,LT,HS,LO,MI,PL,VS,VC,AL`), then no correction is necessary and the condition test may immediately follow the 64-bit subtraction:

```
...                     // Load operands as before
SUBS      R0,R0,R2      // subtract LS halves, capture borrow
SBCS      R1,R1,R3      // subtract MS halves w/brw; set flags
...                     // OK to test GE,LT,HS,LO,MI,PL,VS,VC,AL
```

However, all other conditions (`EQ,NE,GT,LE,HI,LS`) require a different approach. The solution for `EQ` and `NE` is quite simple:



ISBN 978-1-54393-624-7

```
...                     // Load operands as before
SUBS      R0,R0,R2      // compute LS half of difference
SBC       R1,R1,R3      // compute MS half of difference
ORRS      R1,R0,R1      // Z=1 iff both halves are zero
...                     // Now OK to test for EQ or NE
```

For LE, GT, LS and HI we can avoid testing the Z flag by reversing the operands. Since x <= y is equivalent to y >= x, this allows us to replace LE by GE or LS by HS, which don't require testing the value of Z:

```
...                     // Load operands as before
SUBS      R2,R2,R0      // subtract LS halves (operands reversed)
SBCS      R3,R3,R1      // subtract MS halves (operands reversed)
...                     // Replace LE/GT by GE/LT, or
...                     // Replace LS/HI by HS/LO.
```

Finally, note that in all cases except EQ and NE, we can avoid modifying one register by replacing the SUBS instruction by a CMP instruction.

CHAPTER 8: MUTIPLICATION AND DIVISION REVISITED

Page 156, Table 8-3, 3rd row (11x), 2nd column (Composition), 2nd line: Replace $R0 = R1 + 2xR0$ by $R0 = R0 + 2xR1$.

Section 8.3 (Division by an Arbitrary Constant), page 162, right-hand column (code for "Divides by -7"):

Replace the instruction, "ADD R0,R1,**R0**,LSR 31" by "ADD R0,R1,R1,LSR 31"

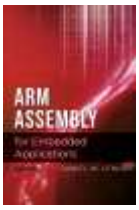
Section 8.3 (Division by an Arbitrary Constant), page 163, paragraph that starts at the bottom of the page:

Replace the entire paragraph by the following:

> The SMMUL instruction computes the most-significant half of a 64-bit signed product and eliminates the need to use register R2 in the previous code. The SMMLA instruction does the same, but also adds the value of another 32-bit operand to the result. This instruction can thus replace the SMULL/ADDS.N sequence in the earlier code that divides by +7. The SMMLS instruction computes the difference of a 32-bit operand less the most-significant half of a 64-bit operand. Unfortunately, this doesn't help simplify the code that divides by -7. There are no unsigned versions of these instructions.

Section 8.3 (Division by an Arbitrary Constant), page 172, Table 8-6:

In the third column of the table, second row, replace "$Rd \leftarrow (Rn \times Rm)$ <63..32> + R_a" by "$Rd \leftarrow R_a + (Rn \times Rm)$ <63..32>".



ISBN 978-1-54393-624-7

In the third column of the table, third row, replace "$Rd \leftarrow (Rn \times Rm)$ <63..32> $- R_a$" by "$Rd \leftarrow R_a - (Rn \times Rm)$ <63..32>".

CHAPTER 10: WORKING WITH FIXED-POINT REAL NUMBERS

Section 10.5.6 (Multiplying a Q32 Value by a Fractional Value), page 217, Figure 10-7:

The wrong partial products are indicated as being equal to zero ("= 0"). Those that are zero are only those that are cross-hatched and have a large "X" on top of them.

CHAPTER 13: INLINE CODING

Replace the identifier "`asm`" by "`__asm`" in every place that inline assembly is defined. Although not required when using the deprecated EmBitz IDE, this change is necessary when using the ARM Embedded Toolchain introduced in Appendix B of the 4th edition. Note: The file library.h found in the archive workspace.zip on the textbook support website has been edited to include a macro definition so that "`__asm`" will be automatically substituted in place of "`asm`" in C source code files.