CMPSCI 201 – Fall 2005

Midterm #2 Solution

Professor William T. Verts

<1> 10 Points – Convert the decimal number -47.375 into (a) binary scientific notation (i.e., $\pm 1.xxxx \times 2^Y$), and (b) the equivalent binary single-precision floating-point representation.

In binary, -47.375 is -101111.011, which is -1.01111011×2^5. The sign bit is 1 because the number is negative, the leading 1 (to the left of the decimal point) is dropped from the significant digits to form the mantissa, and the exponent 5 is added to the bias 127 to get the biased exponent 132 (10000100_2).

Sign	Exponent								Mantissa																						
1	1	0	0	0	0	1	0	0	0	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

<2> 10 Points – The following code shows a data table defined for the ARM containing all 13 possible factorial values that fit into a 32-bit integer. Write an ARM code fragment (not a complete routine) that replaces the integer value in `R0` with its factorial from the table (i.e., `R0` ← `R0!`). Use any auxiliary registers you need, and don't worry about register transparency. Your code must return zero for any value outside the range from 0 to 12.

```
TABLE DCD 1         ; 0!
      DCD 1         ; 1!
      DCD 2         ; 2!
      DCD 6         ; 3!
      DCD 24        ; 4!
      DCD 120       ; 5!
      DCD 720       ; 6!
      DCD 5040      ; 7!
      DCD 40320     ; 8!
      DCD 362880    ; 9!
      DCD 3628800   ; 10!
      DCD 39916800  ; 11!
      DCD 479001600 ; 12!
```

The pseudo-code for this problem is as follows:

```
If (R0 >= 0) And (R0 <= 12) Then
     R0 ← Table[R0]
Else
     R0 ← 0
```

The core code is an array reference using R0 as the index. Thus, half the credit is assigned to the following two lines of code:

```
          ADR   R2,TABLE
          LDR   R0,[R2,R0,LSL #2]
```

The ADR pseudo-instruction puts the base address of TABLE into R2. Remember that the LSL #2 is to convert the array index into a byte offset, since each storage cell is four bytes long.

The rest of this problem is to filter out illegal values. One approach is to explicitly test for the low and high values:

```
          CMP   R0,#0
          BLT   Illegal
          CMP   R0,#12
          BGT   Illegal
```

Another is to perform what is essentially an *unsigned* comparison on a *signed* value:

```
          CMP   R0,#13
          BHS   Illegal    ; (Branch on High or Same)
```

In this case, all negative values are treated as very large unsigned values, and can be eliminated at the same time as too-large positives.

The final expected code is then either of the following approaches:

```
          CMP   R0,#0
          BLT   Illegal
          CMP   R0,#12                CMP   R0,#13
          BGT   Illegal               BHS   Illegal
---------------------------------------------------
          ADR   R2,TABLE
          LDR   R0,[R2,R0,LSL #2]
---------------------------------------------------
          B     Done
Illegal   MOV   R0,#0
Done      …
```

<3> 10 Points – In each of the following problems you are to multiply the contents of integer register R0 by a constant value, in **<u>one</u>** instruction, without using any other registers, and without using any explicit multiplication instruction such as MUL, MLA, or UMULL. For all questions you may assume that the initial value in R0 is positive.

```
1.   R0 := R0 × 15           RSB   R0,R0,R0,LSL #4

2.   R0 := R0 × 16           MOV   R0,R0,LSL #4

3.   R0 := R0 × 17           ADD   R0,R0,R0,LSL #4

4.   R0 := R0 × 1¼           ADD   R0,R0,R0,LSR #2

5.   R0 := R0 × ¾            SUB   R0,R0,R0,LSR #2
```

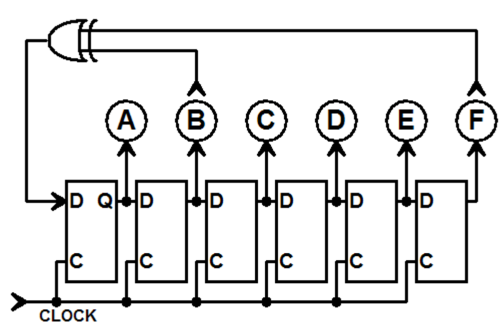
<4> 10 Points – For this problem you will need to use the `MUL` R_D,R_M,R_S instruction. The `MUL` instruction cannot multiply by constants (only registers), and R_D cannot be the same register as R_M (i.e., `MUL R0,R0,R1` is illegal, but `MUL R0,R1,R0` is OK). Create a complete ARM subroutine to evaluate the integer polynomial $y = 9x^2 + 4x + 5$, where the value of x is passed in through `R0` and the result y is passed back through `R1`. You must maintain full transparency on all used registers, except for `R1`, `LR`, and `IP`.

As it turns out, *no* registers need to be explicitly saved in order to perform this subroutine. The destination register `R1` can be used in all calculations, along with the barrel-shifter, and no other temporaries are necessary. Since this subroutine doesn't call other subroutines, the `LR` doesn't need to be saved and restored. Saving registers is OK, but if you do then they must be restored properly. Here is the expected code:

```
Subroutine
        MUL   R1,R0,R0              ; R1 ← R0²
        ADD   R1,R1,R1,LSL #3       ; R1 ← 9R0²
        ADD   R1,R1,R0,LSL #2       ; R1 ← 9R0² + 4R0
        ADD   R1,R1,#5              ; R1 ← 9R0² + 4R0 + 5
        MOV   PC,LR                 ; Return
```

<5> 5 Points – In the shift register random number generator shown here, the current value (from left to right) is 111001. Show the contents of the register after each of the next five clock cycles.

1. 0 1 1 1 0 0
2. 1 0 1 1 1 0
3. 0 1 0 1 1 1
4. 0 0 1 0 1 1
5. 1 0 0 1 0 1



<6> 5 Points – Short Answer – Explain the difference in action between the following two ARM instructions.

```
STR R0,[SP,#-4]

STR R0,[SP,#-4]!
```

The first does not update the stack pointer (`SP`), but simply stores into the free area below the stack top (remember that stacks grow downwards). The second updates the stack pointer after storage, creating a true stack push.

<7> 10 Points – Rewrite the following code fragment to perform the same task in fewer instructions. The purpose is to replace the value in R0 with +1 for positive numbers, -1 for negative numbers, and 0 for 0 (this is called the signum or SGN function).

```
      CMP R0,#0
      BLT Small
      BGT Big
      MOV R0,#0
      B   Done
Small MOV R0,#-1
      B   Done
Big   MOV R0,#1
Done  …
```

Conditional execution is the key to reducing the code. The CMP is still necessary, since we do not know what went on before this code snippet that may or may not have modified the flag bits. Afterwards, MOV instructions are used to put the correct value into R0 based on *which* of those bits have been set or cleared:

```
CMP   R0,#0
MOVLT R0,#-1
MOVGT R0,#1
```

A third MOV to set the value of R0 to zero (MOVEQ R0,#0) is *not* necessary since the only condition where neither the MOVLT nor the MOVGT get triggered is when R0 is already zero.

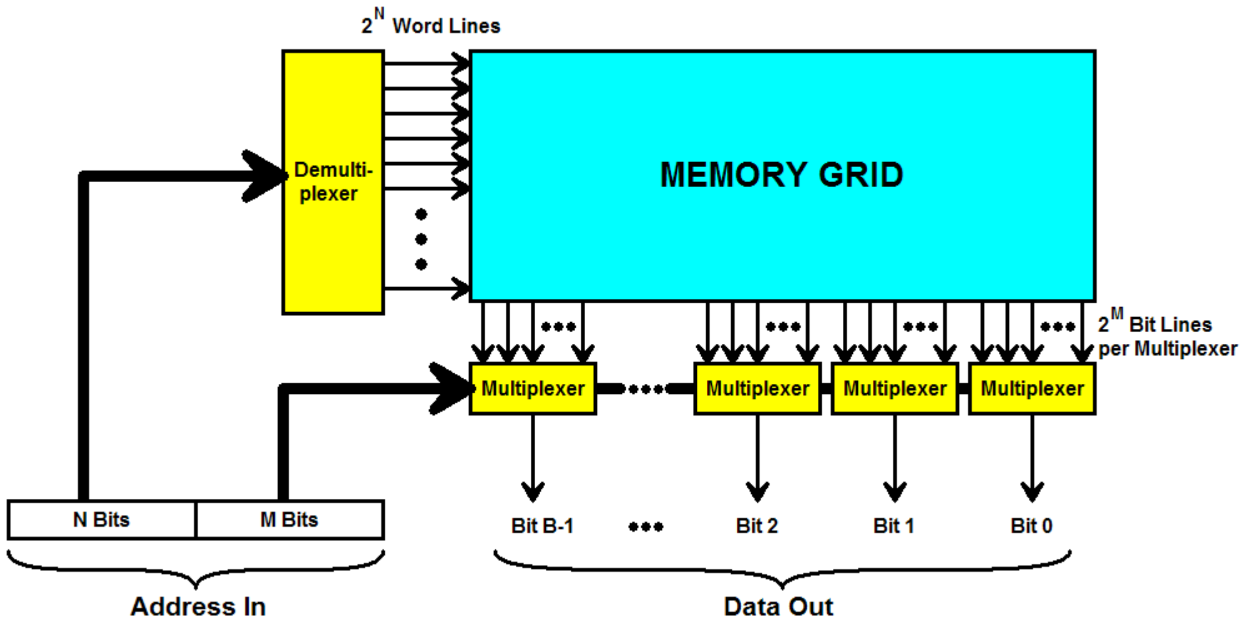
The problem with this code as it stands is whether or not the -1 constant can be created directly. It can't, not by selecting a value in the range 0…255 and then right-rotating it by any arbitrary number of bits. It can be created, however, by using the MVN (move negative) command to move the 1's complement of the specified value into the register. The value -1 is represented by all 1 bits, so moving the 1's complement of 0 into the register will do the same thing. Thus, the "proper" code is:

```
CMP   R0,#0
MVNLT R0,#0    ; -1 by 1's complement of 0
MOVGT R0,#1
```

However, it doesn't take too much work to create a "smart" assembler that replaces an assembly language instruction MOV of -1 with the binary equivalent to a MVN of 0.

Thus, either form is considered correct for this problem.

<8> 10 Points – Examine the memory grid below. With N=6, M=10, and B=8, then…



1. …how many **address lines** are there? (2 points)

   $N+M = 16$

2. …how many **word lines** are there? (2 points)

   $2^N = 2^6 = 64$

3. …how many **bit lines** are there? (2 points)

   $B \times 2^M = 8 \times 2^{10} = 8 \times 1024 = 2^3 \times 2^{10} = 2^{13} = 8192$

4. …how many **memory bits** are there? (2 points)

   2^6 word lines $\times$ 2^{13} bit lines = 2^{19} bits = 512K = 524288

5. …how many **bytes of memory** are there (2 points)

   $2^{N+M} = 2^{16} = 64K = 65536$

<9> 20 Points – In this question we are inventing a new floating-point format that follows many of the basic ideas behind the IEEE-754 rules. In this format, numbers are 128 bits in length, and occupy four successive 32-bit integer words of storage (little endian). The most significant word contains the sign bit and 31 bits of biased exponent; the other three words contain the 96-bit mantissa. This layout is shown below:

BASE+0		**Rightmost Mantissa Word**
BASE+4		**Middle Mantissa Word**
BASE+8		**Leftmost Mantissa Word**
BASE+12	**Sign**	**Biased Exponent**

The range of numbers is thus between $\pm 1.0 \times 2^{-e}$ and $\pm 1.\mathrm{FFFF...FFF} \times 2^{+e}$ (ignoring infinity, NaNs, and denormals of the form $\pm 0.\mathrm{xxxxx...xxx} \times 2^{-e}$), for some value of e.

A. (2 points) What is the decimal value of the ***bias***?

2^{N-1}-1, where N=31 (the number of bits in the exponent), which is 2^{30}-1 = **1,073,741,823**.

B. (2 points) What is the decimal range of valid ***exponent values***?

I expected that people would figure out the power of ten available, which is $\mathrm{Log}_{10}(2^{1,073,741,823})$, or $10^{\pm 323228496}$. This was interpreted by some students as the decimal value of the binary unbiased exponent, or between -1,073,741,823…+1,073,741,824, which was acceptable.

C. (2 points) Approximately how many decimal ***digits of precision*** can be represented with this floating-point format?

Since there are 96 bits of mantissa available, but the leading 1-bit of the binary number is dropped, the precision is 97 total bits. Thus the number of decimal digits is $\log_{10}(2^{97})$, which is 29.1999…, or **29 digits**.

D. (2 points) What is the ***hexadecimal*** value of the floating-point number -12.125 in this format? Show breaks between the four words of your answer as follows:

```
xxxxxxxx  xxxxxxxx  xxxxxxxx  xxxxxxxx
Base+12   Base+8    Base+4    Base+0
```

The number in binary scientific notation is -1.100001×2^3. The sign bit is 1 since the number is negative, the stored mantissa is 100001000…000, and the exponent without the bias is 3. In hex, the number is then:

```
C0000002  84000000  00000000  00000000
```

E. (12 points) Write a chunk of ARM code using the model shown (where symbol BASE is defined as the address of the first of the four successive words of memory) to compute an estimate of the ***square root*** of the stored number. Use only integer registers `R0` through `R11` in your answer; no floating point registers. You may assume that the sign bit is 0 to avoid issues with imaginary numbers. To compute your estimate you must divide the *signed exponent* by 2 as we have done in previous examples (discarding any remainder from the division), but this time you must also divide the *mantissa* by 2 instead of simply setting it to zero. Thus, for non-zero binary floating-point values of the form $1.abcd…xyz \times 2^{e}$, the approximation we want is $1.0abcd…wxy \times 2^{e/2}$. (Hint: you may have to do some thinking and/or research about shifting and rotating values in ways we haven't discussed in class, as well as for the generation of some of the constants used by your routine.)

```
ADR    R5,BASE        ; Get address of array start
LDR    R1,BIAS        ; Get bias (however defined)
LDR    R0,[R5,#12]    ; Load sign/exponent word
SUB    R0,R0,R1       ; Subtract the bias
MOVS   R0,R0,LSR #1   ; Divide by 2…
ADCMI  R0,R0,#0       ; …and correct for odd negatives
ADD    R0,R0,R1       ; Add the bias
STR    R0,[R5,#12]    ; Store back into sign/exponent

LDR    R0,[R5,#8]     ; Load most significant mantissa word
MOVS   R0,R0,LSR #1   ; Divide it by 2, remainder goes to C
STR    R0,[R5,#8]     ; Store most significant mantissa word

LDR    R0,[R5,#4]     ; Load next mantissa word
MOVS   R0,R0,RRX      ; Rotate C into left, rightmost into C
STR    R0,[R5,#4]     ; Store next mantissa word

LDR    R0,[R5,#0]     ; Load least significant mantissa word
MOVS   R0,R0,RRX      ; Rotate C into left, rightmost into C
STR    R0,[R5,#0]     ; Store least significant mantissa word

…

BIAS DCD   1073741823
```

The `MOVS R0,R0,RRX` is the piece we hadn't discussed in class, but was in the ARM instruction reference. The `RRX` stands for "Rotate Right eXtended" which shifts the current value of the carry flag `C` into the left end of the word, and the rightmost bit of that word is shifted out to become the new value of the carry flag. Thus, the carry is used to shift a bit out of one word into the next, implementing a multiple precision shift.

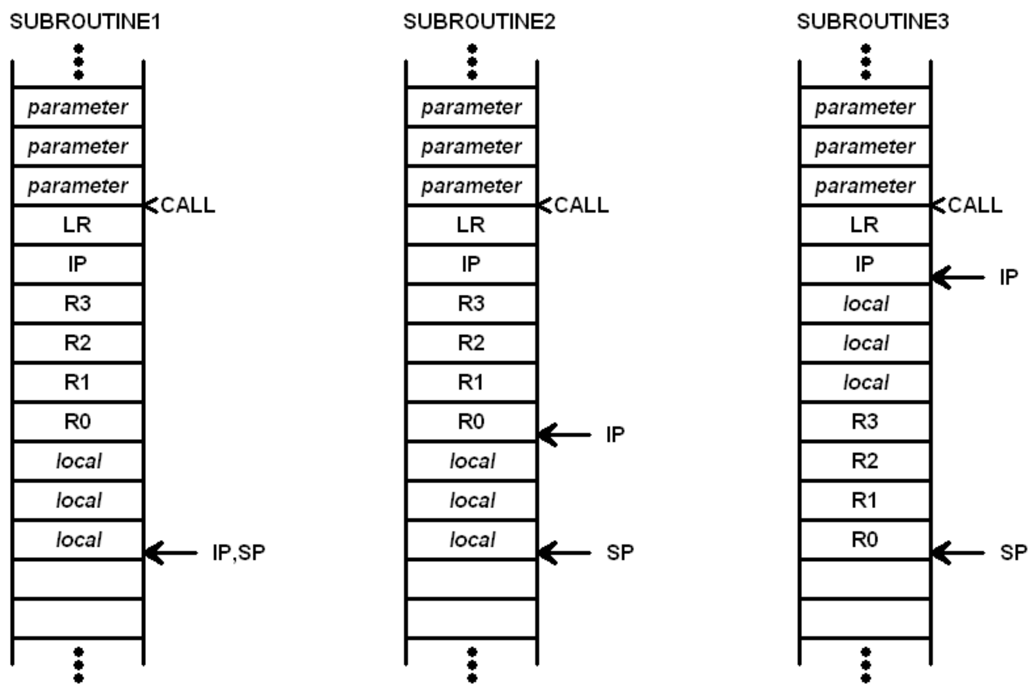
<10> 10 Points – Essay answer – Here are three ways of handling stack frames for subroutines. The three subroutines have identical purposes and internal functions, except for the *offsets* used in referencing their parameters and local variables. In all three cases there are parameters pushed onto the stack before the call, and inside each subroutine the registers `R0`, `R1`, `R2`, `R3`, `IP`, and `LR` are pushed on the stack, as well as three words of storage reserved for local variables (represented by the `SUB SP,SP,#12` instruction). Your answer to this problem doesn't require knowing exactly how many parameters were pushed onto the stack before each subroutine call, but the number is greater than zero and it is the same number of parameters in each of the three cases. Discuss the ***advantages and disadvantages*** of each of the three strategies shown below.

```
SUBROUTINE1
     STMDB SP!,{R0-R3,IP,LR}
     SUB   SP,SP,#12
     MOV   IP,SP
     …
     …    ; do useful work
     …
     ADD   SP,SP,#12
     LDMIA SP!,{R0-R3,IP,PC}


SUBROUTINE2
     STMDB SP!,{R0-R3,IP,LR}
     MOV   IP,SP
     SUB   SP,SP,#12
     …
     …    ; do useful work
     …
     ADD   SP,SP,#12
     LDMIA SP!,{R0-R3,IP,PC}


SUBROUTINE3
     STMDB SP!,{IP,LR}
     MOV   IP,SP
     SUB   SP,SP,#12
     STMDB SP!,{R0-R3}
     …
     …    ; do useful work
     …
     LDMIA SP!,{R0-R3}
     ADD   SP,SP,#12
     LDMIA SP!,{IP,PC}
```

In all three cases the stack is loaded up with the same information (perhaps in different orders) by the time the "do useful work" point is encountered. The trick to understanding the differences is to plot out the stack frames in all three cases, shown as follows:



In `SUBROUTINE1` all offsets in the current stack frame are positive with respect to the `IP` register. Any stack action (such as local pushes and pops, setting up a stack frame for a new subroutine call, etc.) will be at negative offsets relative to `IP`. Thus, there is a clean and distinct separation between *this* stack frame and the *next* stack frame. The distances to the parameters on the stack are very large, however, and those offsets will change depending on changes to the numbers of registers saved and to the amount of local storage reserved.

In `SUBROUTINE2` all parameters (including the registers) are at positive offsets relative to `IP`, but local variables are at negative offsets. The distances to the parameters are shorter than for `SUBROUTINE1`, and are still subject to the numbers of registers saved, but these offsets are not affected by the amount of local storage reserved (which is below `IP`). There is not a clean separation between *this* stack frame and the *next* stack frame.

In `SUBROUTINE3` there are very short positive distances to the parameters on the stack and very short negative distances to the local variables. Those distances are not affected by either the number of registers saved or by the amount of local storage reserved. Offsets into the local storage are not affected by the numbers of registers saved. As with `SUBROUTINE2` there is not a clean separation between *this* stack frame and the *next* stack frame, and the code is slightly longer than either of the previous versions.