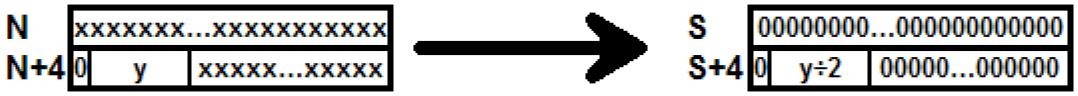
CMPSCI 201 – Fall 2004

Midterm #2 Answers

Professor William T. Verts

<1> 15 Points – You should be quite familiar by now with the single-precision floating point numeric format (one 32-bit word containing the sign bit, 8 bits for the biased exponent, and 23 bits of mantissa, with bias=127). Now we need to consider the double-precision format (two 32-bit words stored in little-endian order, containing the sign bit, 11 bits for the biased exponent, and 52 bits of mantissa, with bias=1023). A positive double precision number is located in main memory at addresses N and N+4. Write the ARM code using ***only integer instructions*** to generate an approximate square root for the number into memory locations S and S+4, similar to the method we developed in class for single precision. In essence, we are converting a floating-point number of the form N=+1.xxxx×2^{Y} into S=+1.0000×2$^{Y÷2}$, as shown below (remember that exponent value Y is biased):



```
MOV    R0,#0          Store 0 into first word
                           of two-word double.
LDR    R0,N+4         Load word containing exponent.
MOV    R0,R0,LSR #20  Shift exponent down to low end.
LDR    R1,=1023       Load bias from constant pool.
SUB    R0,R0,R1       Subtract bias.
MOVS   R0,R0,LSR #1   Divide bias by 2.
ADCMI  R0,R0,#0       Correct divide for odd negatives.
ADD    R0,R0,R1       Add bias.
MOV    R0,R0,LSL #20  Shift exponent to correct place.
STR    R0,S+4         Store into second word of double.
```

Analysis: This is very similar to the single-precision technique that we developed in class: shift the biased exponent 23 bits to the right to clear out mantissa bits and align the biased exponent to the low end of the word, subtract the bias, divide the exponent by two, re-add the bias, and shift the exponent back into its proper position.

The difficulties here are that the mantissa is 52 bits distributed across two words instead of 23 bits, and the bias is 1023 instead of 127. Clearing out the first word of the mantissa is trivial. Since the mantissa is 52 bits in length, clearing one word clears 32 of those 52 bits, leaving 20 bits in the low end of the second word. Shifting that second word to the right by 20 bits clears those bits and aligns the biased exponent to the low end of the word.

Unfortunately, you cannot simply subtract off 1023, since that constant will not fit in the space allowed for constants in and instruction (it is wider than eight bits). Therefore, you must either build the constant in a register (`MOV R1,#1024`, which *will* fit in an instruction, followed by `SUB R1,R1,#1`) or you must load the constant from memory. The instruction form `LDR R1,=1023` causes the assembler to reserve a word of memory in what is called the constant pool (an area right after the end of the program), initialize it to 1023, then assemble a `LDR` instruction to reference that location. Either way is fine.

Grading: In general, accept any solution that works, but no solution should contain loops. Remove 5 points for any solution, even one that works, which is very long and/or very complicated (containing loops or longer than 20 instructions). Remove 10 points for any solution that gets a few things right but is otherwise totally off-the-wall. The expected solution is similar to the solution developed in class (and in the notes). In any more-or-less correct solution, remove 2 points for attempting to incorrectly use 1023 as an embedded constant. Remove 1 point for each case where the wrong constant was used. It is OK to use `ADR` pseudoinstructions to get the base address of `N` or `S` and then treat them as arrays.

<2> 5 Points – In each of the following problems you are to multiply the contents of integer register `R0` by a constant value, in one instruction, without using any other registers, and without using any explicit multiplication instructions such as `MUL`, `MLA`, or `UMULL`. If the task cannot be accomplished in a single instruction, answer “Can’t be Done”.

(1)	`R0 := R0 × 1025`	`ADD R0,R0,R0,LSL #10`
(2)	`R0 := R0 × 1024`	`MOV R0,R0,LSL #10`
(3)	`R0 := R0 × 257`	`ADD R0,R0,R0,LSL #8`
(4)	`R0 := R0 × 256`	`MOV R0,R0,LSL #8`
(5)	`R0 := R0 × 255`	`RSB R0,R0,R0,LSL #8`

Grading: 1 point each, all or nothing.

<3> 10 Points – In the subroutine call below, which of the parameters are ***call-by-value***, ***call-by-return***, ***call-by-value-return***, and ***call-by-reference***?

```
LDR R0,X               X is call-by-value
STR R0,[SP,#-4]!
ADR R0,Y               Y is call-by-reference
STR R0,[SP,#-4]!
LDR R0,Z               Z is call-by value
STR R0,[SP,#-4]!
BL  SUBROUTINE
LDR R0,[SP],#12
STR R0,Thingy          Thingy is call-by-return
                       (Shares parameter space with Z)
```

Analysis: Three items are pushed onto the stack (the value of `X`, the address of `Y`, and the value of `Z`), but only one item is popped off (the value to store into `Thingy`). At the same time that `Thingy` is loaded off of the stack, its storage and the two following words are also discarded at the same time.

Grading: Remove 3 points for each wrong answer, but do not go below 10 points. Remove only 1 point for stating that `Thingy` is call-by-value-return without mentioning `Z`.

<4> 10 Points – Examine the following recursive subroutine. Describe in words what it does and how it works. (What registers are changed, how and when are they changed, what happens to the stack, etc.?)

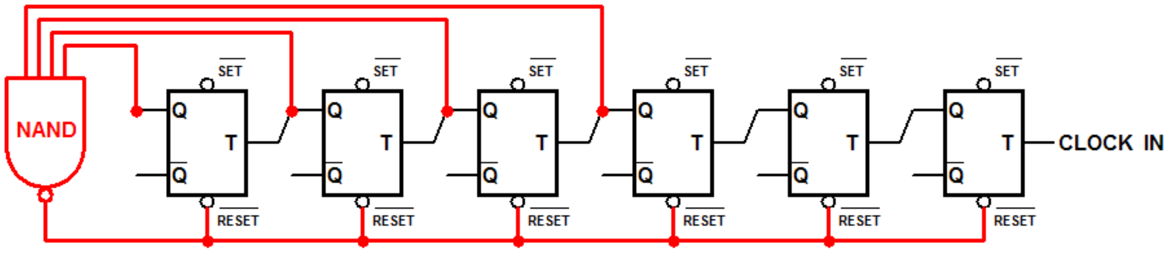
```
SUB   STR   LR,[SP,#-4]!
      SUBS  R0,R0,#1
      BLNE  SUB
      ADD   R1,R1,#1
      LDR   PC,[SP],#4
```

The subroutine recursively calls itself, decrementing `R0` until it becomes zero. At this point the number of return addresses on the stack equals the original value of `R0`. The subroutine then starts unwinding the recursion, adding 1 to `R1` each time. The net effect, without considering stack operations, is the following code sequence:

```
ADD   R1,R1,R0
MOV   R0,#0
```

Grading: Give full credit for any reasonable answer that describes the number of addresses pushed onto the stack and the final effects in registers `R0` and `R1`. Remove 3 points if no description of stack operations is present. Remove 2 points for omitting the effects on `R0`. Remove 2 points for omitting the effects on `R1`.

<5> 10 Points – The following diagram shows a counter chain of six T (toggle) flip-flops. Normally, this circuit would take 2^6=64 clock pulses to count up from zero around to zero again. Add gates to this circuit so that it counts from 0 to 59 and then back to 0 on the 60TH pulse (forming the seconds or minutes counters in a binary time-of-day clock).



Analysis: The counter adds 1 to the binary value of the register after every complete clock pulse. The rightmost flip-flop is the 1's place, the next flip-flop is the 2's bit, etc. The idea here is to detect the value 60 and use that value to clear the register. In binary the value 60 is 111100 (32+16+8+4). As the counter counts up, the transition between 59 (111011) and 60 (111100) guarantees that as the leftmost four bits all become one, the rightmost two bits must be zero. This case is detected with an AND gate, but since the reset lines are active on 0 the output of the AND gate must be inverted; hence we use a 4-input NAND. Using an explicit AND gate followed by a NOT gate is OK. Since the rightmost two bits are known to be zero, they do not need to be cleared; their reset lines may be left unconnected.

Grading: Accept any solution that works. Remove 1 point for clearing on 59 instead of 60. Remove 1 point for forgetting the NOT on the resets. Remove 2 points for using OR+NOT/NOR instead of AND+NOT/NAND. Remove 1 point for each case where lines are left dangling or are shorted out, or where any line is connected to a "set" input, but do not go below 0.

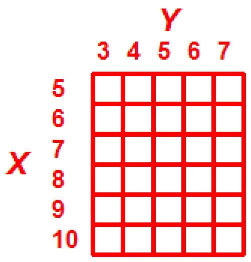
<6> 15 Points – In a high-level language such as Pascal, I declare a row-major two-dimensional array with six rows and five columns of 32-bit integers by writing the statement:

```
Var A : Array [5..10,3..7] Of Integer ;
```

The upper left element of the array is at `A[5,3]` and the lower right element of the array is at `A[10,7]`. In translating this array declaration into ARM assembly language, I use the directive **`A % 120`** to allocate and initialize to zero all bytes of the array (6 rows × 5 columns = 30 elements; 30 elements × 4 bytes per element = 120 bytes of memory).

(1) (5 points) In pure algebra, write a mathematical expression that shows the mapping function from array indices `X` and `Y` to the ***byte offset*** in memory of the required item, relative to the base address of the array, and using row-major form. Your answer should be a polynomial on `X` and `Y` of the form: Offset ← *f*(`X,Y`). Tell me what *f* is.

```
Offset = [(X - 5) × 5 + (Y - 3)] × 4
       = [5X - 25 + Y - 3] × 4
       = [5X + Y - 28] × 4
       = 20X + 4Y - 112
```



Grading: Accept any equivalent equation. Remove 2 points for an equation equivalent to `[5X+Y-28]` but forgetting to multiply by 4 to get the byte offset. Remove 2 points for implementing a column-major solution instead of a row-major solution.

(2) (5 points) Write the correct ARM assembly language statements to load into register `R0` the contents of `A[X,Y]` where `X` and `Y` are integer variables stored in memory. You do not need to perform range checking on `X` or `Y`.

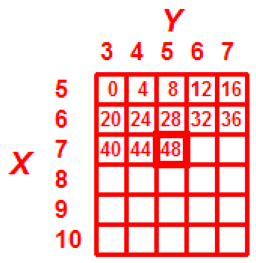
```
LDR  R1,X
MOV  R2,#20
MUL  R1,R2,R1
LDR  R2,Y
ADD  R1,R1,R2,LSL #2
SUB  R1,R1,#112
ADR  R5,A
LDR  R0,[R5,R1]
```

```
LDR  R1,X
LDR  R2,Y
ADD  R1,R1,R1,LSL #2
ADD  R1,R1,R2
SUB  R1,R1,#28
ADR  R5,A
LDR  R0,[R5,R1,LSL #2]
```

Grading: Accept any working solution. Remove 1 point for using multiplying by a constant inside a `MUL` instruction. Remove 1 point for each occurrence of omitting an important section (such as the `ADR` pseudoinstruction, the `LSL #2` in the right-hand solution, etc.), but do not go below zero.

(3) (5 points) Write the correct ARM assembly language statements to load into register `R0` the contents of `A[7,5]`. By using constant subscripts (the 7 and the 5) you are free to optimize your code in any manner you see fit.

Offset $= [(X - 5) \times 5 + (Y - 3)] \times 4$, where $X=7$ and $Y=5$
$= [(7 - 5) \times 5 + (5 - 3)] \times 4$
$= [2 \times 5 + 2] \times 4$
$= [12] \times 4$
$= 48$

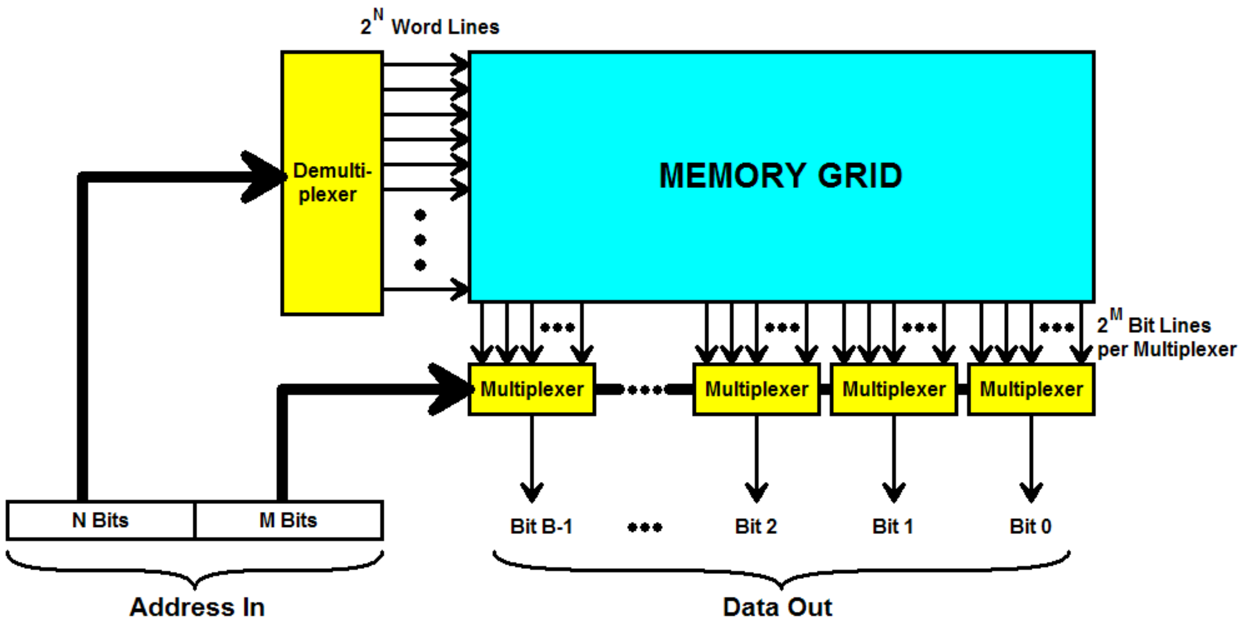


```
ADR   R5,A
LDR   R0,[R5,#48]

ADR   R5,A+48
LDR   R0,[R5]
```

Grading: Accept any working solution. Remove 1 point for a column-major solution instead of a row-major solution. Remove 1 point for any solution longer than 5 or 6 lines of code.

<7> 15 Points – Examine the memory grid below. With N=3, M=7, and B=12, then…



1. …how many **address lines** are there? (2 points)
   N+M = 3+7 = **10**

2. …how many **word lines** are there? (2 points)
   $2^N = 2^3 = \mathbf{8}$

3. …how many **bit lines** are there? (2 points)
   $2^M \times B = 2^7 \times 12 = 128 \times 12 = \mathbf{1536}$

4. …how many **memory bits** are there? (2 points)
   8 word lines × 1536 bit lines = **12288 total bits**

5. …how many **AND-gates** are there in the entire circuit? (2 points)
   $2^N + 2^M \times B = 2^3 + 2^7 \times 12 = 8 + 128 \times 12 = 8 + 1536 = \mathbf{1544}$

6. …is this the most efficient arrangement in terms of overall hardware? I.e., is there a better choice for N and M which results in the same amount of memory but uses fewer gates? If so, what are the best values for N and M? (5 points)
   No, this is not the best arrangement. The "Sweet Spot" is **N=7 and M=3**.

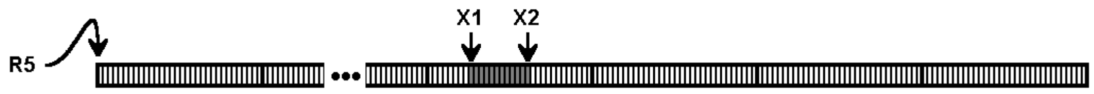
N	M	ANDs = $2^N + 2^M \times B$	
3	7	$2^3 + 2^7 \times 12 = 8 + 128 \times 12 = 8 + 1536 = 1544.$	-4 points
4	6	$2^4 + 2^6 \times 12 = 16 + 64 \times 12 = 16 + 768 = 784.$	-3 points
5	5	$2^5 + 2^5 \times 12 = 32 + 32 \times 12 = 32 + 384 = 416.$	-2 points
6	4	$2^6 + 2^4 \times 12 = 64 + 16 \times 12 = 64 + 192 = 256.$	-1 point
7	**3**	$\mathbf{2^7 + 2^3 \times 12 = 128 + 8 \times 12 = 128 + 96 = 224.}$ ←	**Correct**
8	2	$2^8 + 2^2 \times 12 = 256 + 4 \times 12 = 256 + 48 = 304.$	-1 point

<8> 20 Points – One extension to the graphics lab assignment is to create a procedure to draw a horizontal line across one raster of the screen. To do this you need three parameters: the desired raster (`Y`) and the two endpoints of the line (`X1` and `X2`). Assuming that `X1` is less than or equal to `X2` and that `X1`, `X2`, and `Y` all contain legal values and are "visible" on screen, one approach to drawing a horizontal line is the following:

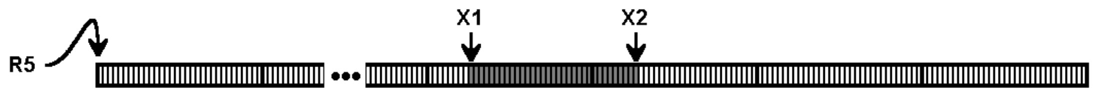
```
For X := X1 To X2 Do Set_Pixel(X,Y) ;
```

This approach, however, is very inefficient, since in our model pixels are stored 32 per memory word (one bit per pixel). What we would like to do instead is to write a procedure that makes as ***few memory accesses*** as possible, and uses masking techniques (AND, OR, NOT) to fill in large blocks of pixels simultaneously. Once the offset into memory of the beginning of `Y`'s raster line is known (assume that the value is in ARM register `R5`), there are three cases to consider, as follows.

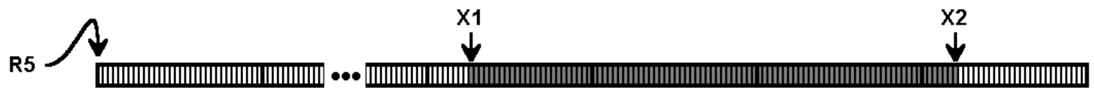
1. `X1` and `X2` are inside the same memory word:



2. `X1` and `X2` are in adjacent memory words:



3. `X1` and `X2` are in widely separated memory words:



Write a code fragment to set to 1 all bits/pixels between `X1` and `X2` using ***at most one*** `LDR` and ***at most one*** `STR` to each word of memory. Do not change any other pixels.

Assume that `X1` is in register `R1` and `X2` is in register `R2`, that the pointer to the address of `Y`'s raster line in memory is in `R5`, and that all values are legal (i.e., no clipping is required). All other registers are free for use by your code. This is not a subroutine, so no register transparency is necessary. Write your code to be as compact and efficient as possible.

Type your name and your answer into Notepad, print it out using 12 point Courier New, and then staple your answer to the back of this exam section. Please no handwritten answers!

```
     AND  R3,R1,#31             R3 := R1 Mod 32  (Bit offset X1)
     MVN  R0,#0                 R0 := all 1 bits
     MOV  R3,R0,LSR R3          R3 := Left Mask  (000…0111111…111)
------------------------------------------------------------------
     AND  R4,R2,#31             R4 := R2 Mod 32  (Bit offset X2)
     EOR  R4,R4,#31             R4 := 31 - R4
     MVN  R0,#0                 R0 := all 1 bits
     MOV  R4,R0,LSL R4          R4 := Right Mask (111…111100…0000)
------------------------------------------------------------------
     MOV  R1,R1,LSR #5          R1 := R1 Div 32 (Word of X1)
     MOV  R2,R2,LSR #5          R2 := R2 Div 32 (Word of X2)
------------------------------------------------------------------
     CMP  R1,R2                 See if X1 is in same word as X2
     BNE  Many
------------------------------------------------------------------
     AND  R3,R3,R4              Are in same word; combine masks
     B    Done
------------------------------------------------------------------
Many LDR  R0,[R5,R2,LSL #2]     Fill right end of line
     ORR  R0,R0,R4
     STR  R0,[R5,R2,LSL #2]
------------------------------------------------------------------
     MVN  R0,#0                 R0 := all 1 bits
Loop SUB  R2,R2,#1              Fill middle words from Word(X2)-1
     CMP  R1,R2                               down to Word(X1)+1
     BEQ  Done
     STR  R0,[R5,R2,LSL #2]
     B    Loop
------------------------------------------------------------------
Done LDR  R0,[R5,R1,LSL #2]     Fill left end of line (or entire
     ORR  R0,R0,R3              line if X1 in same word as X2)
     STR  R0,[R5,R1,LSL #2]
```

Analysis: The hardest task is to generate the masks for the left end of the line and for the right end of the line. If it is then determined that the left end and the right end both fall in the same word of memory the two masks are ANDed together to form a composite mask. This mask is ORed into the proper word of memory. Otherwise, the left mask is ORed into the left word of the line, the right mask is ORed into the right word of the line, and all words in between are filled with 1 bits. The simplest way to fill a register with 1 bits is `MVN R0,#0` which moves the 1's complement of the constant into the register. This can also be done by using `MOV R0,#-1`, which the assembler will convert automatically into the equivalent `MVN` instruction.

Finding the correct word in the raster is X Div 32, performed by shifting the register to the right by 5 bits. Once the word offset is known, this offset must be multiplied by 4 (LSL #2) to get the actual byte address. Finding the correct pixel offset within the word is X Mod 32, performed by ANDing the register with 31 (0000…000011111).

Grading: Accept any working solution. Solutions shorter than the one here may work, but they may not handle all cases properly. Remove 5 points for not handling the case where the endpoints are in the same word. Remove 2 points for each improperly constructed mask. Remove 5 points for hard-coding a three-word raster line instead of a general purpose routine. Remove 2 points for each minor syntax or semantic error.