# CMPSCI 201 – Fall 2006

# Final Exam Solution

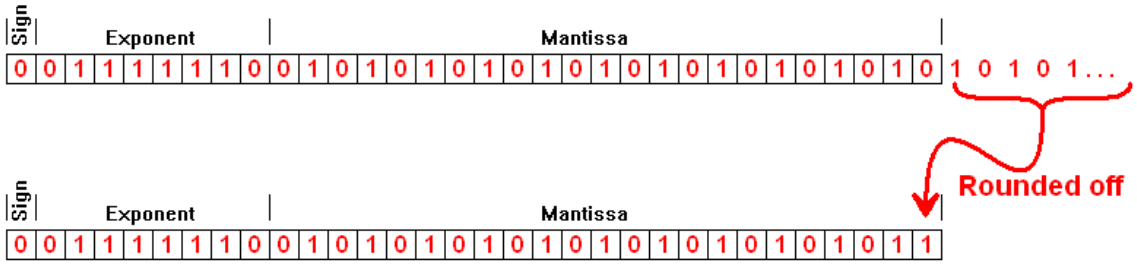
<1> 10 Points – Convert the decimal number ⅔ into single-precision floating-point representation. Since ⅔ is a repeating fraction in both decimal and in binary, put into the mantissa the ***closest possible approximation*** (i.e., perform rounding as needed).

Start by converting ⅔ into true binary. If you multiply a fraction by 2, it is the same as left-shifting the number and what was the left-most bit of the fraction shifts to the left of the decimal point. Repeating the process successively shifts bits to the left of the decimal. You can do this by using either vulgar fractions or explicit decimal fractions:

| Step | Vulgar | Decimal | Bit |
|---|---|---|---|
| 0 | ⅔ | .666… | |
| 1 | ⅔×2=1⅓ | .666…×2=1.333… | 1 |
| 2 | ⅓×2=0⅔ | .333…×2=0.666… | 0 |
| 3 | ⅔×2=1⅓ | .666…×2=1.333… | 1 |
| 4 | ⅓×2=0⅔ | .333…×2=0.666… | 0 |
| … | … | … | … |

The sequence repeats after two steps, so the fraction ⅔ is 0.1010101010… (as a side note the fraction ⅓ is by the same technique 0.0101010101… in binary, and adding ⅓+⅔= 0.11111111111… which is exactly one in the same sense that the decimal value 0.99999999999… is also exactly equal to one).

Converting the binary fractional representation of ⅔ into binary scientific notation gives us $1.01010101010\ldots\times2^{-1}$, with the exponent equal to -1. The biased exponent in single precision is 127-1=126, or 01111110 in binary. The remaining fraction (minus the leading 1 bit) is .01010101010…, and when you place that pattern into the mantissa you will find that the last bit kept is 0 and the first bit discarded is 1. Since the part being discarded is greater than or equal to half the value of the last bit kept, that last (right-most) bit is rounded up from 0 to 1. Since the original number was positive, the sign bit is 0.



<2> 5 Points – In this problem we are converting hexadecimal nybbles (values between 0 and 15) into the equivalent ASCII characters according to the given table. The "difference" column shows the offset to add to the nybble value to convert it into the corresponding ASCII character. In all cases 48 is added; if the nybble value is greater than 9 then 7 more is added as well.

| Nybble Value | ASCII | | |
|---|---|---|---|
| | Character | Value | Difference |
| 0 | '0' | 48 = 0110000 | 48 |
| 1 | '1' | 49 = 0110001 | 48 |
| 2 | '2' | 50 = 0110010 | 48 |
| 3 | '3' | 51 = 0110011 | 48 |
| 4 | '4' | 52 = 0110100 | 48 |
| 5 | '5' | 53 = 0110101 | 48 |
| 6 | '6' | 54 = 0110110 | 48 |
| 7 | '7' | 55 = 0110111 | 48 |
| 8 | '8' | 56 = 0111000 | 48 |
| 9 | '9' | 57 = 0111001 | 48 |
| 10 | 'A' | 65 = 1000001 | 55 = 48 + 7 |
| 11 | 'B' | 66 = 1000010 | 55 = 48 + 7 |
| 12 | 'C' | 67 = 1000011 | 55 = 48 + 7 |
| 13 | 'D' | 68 = 1000100 | 55 = 48 + 7 |
| 14 | 'E' | 69 = 1000101 | 55 = 48 + 7 |
| 15 | 'F' | 70 = 1000110 | 55 = 48 + 7 |

Write a correct ARM code fragment to perform the conversion, given that the nybble value is in register **R0**. The ASCII character must be returned in **R0**, and no other registers may be used. Your solution may not use more than three ARM instructions (one point will be removed for every extraneous instruction).

There are a number of approaches that can be taken, all of which compare the value in **R0** against 9 or 10 and add the appropriate offset depending on the result of the comparison. Here are a few legal solutions. In some cases we could use unsigned condition codes instead of signed (i.e., **HI** and **LS** instead of **GT** and **LE**), since all values under consideration fall between 0 and 15:

```
CMP   R0,#9
ADDLE R0,R0,#'0'
ADDGT R0,R0,#'A'-10

CMP   R0,#10
ADDLT R0,R0,#'0'
ADDGE R0,R0,#'A'-10
```

```
CMP   R0,#9
ADDLE R0,R0,#48
ADDGT R0,R0,#55

CMP   R0,#10
ADDLT R0,R0,#48
ADDGE R0,R0,#55
```

The value 48 is the same as the ASCII code for **'0'** and the value 55 is the value for **'A'** (65) minus 10 because the first of the letters starts at input value 10 in the sequence.

The problem setup, however, was attempting to guide the thought process towards the following solutions, which are appropriate for the subsequent exam problem:

```
CMP   R0,#9                    CMP   R0,#9
ADD   R0,R0,#'0'               ADD   R0,R0,#48
ADDGT R0,R0,#'A'-10-'0'        ADDGT R0,R0,#7
```

In this case, we are *always* adding 48 (the ASCII code for **'0'**). If the input number happens to be in the range 10…15, *only then* do we add the difference between **'A'-10** and the **'0'** that we already added; that difference is 7. Note that in this solution the flag bits are set in the first instruction, ignored in the second instruction, and finally acted upon in the third instruction. Now look at the 8-bit binary version of what is going on in this solution:

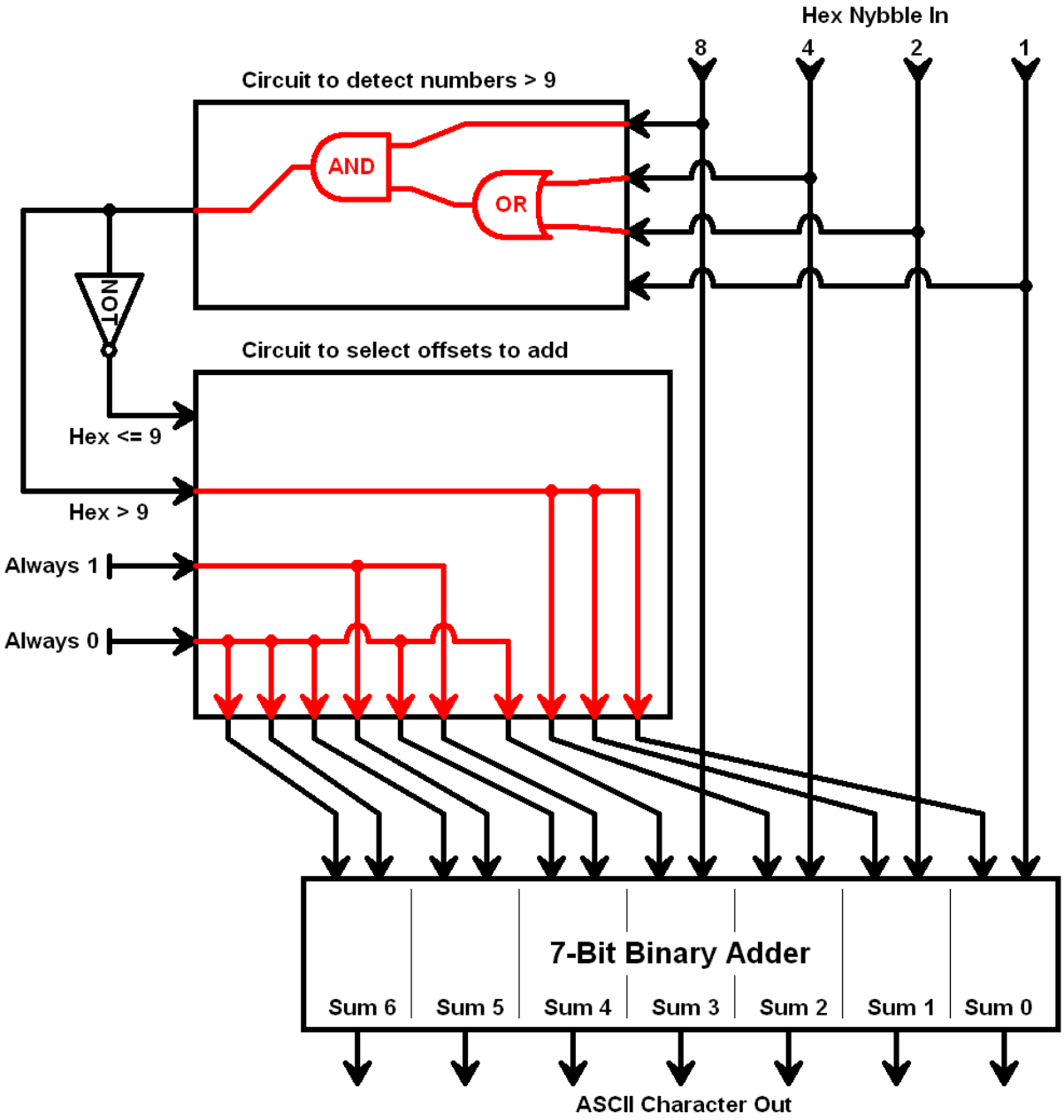
```
0000XXXX     The Binary number in R0 between 0 and 15
01100000     The 48 we are always adding
00000111     The 7 we might add if XXXX > 9
```

The 1-bits of the two correction factors do not overlap, so if we have a binary adder where one operand is **0000XXXX**, we can pump in the 1-bits for the two correction factors into the other operand without any additional logic gates (as long as we know whether or not **XXXX**>9). This observation helps in the following problem.

<3> 15 Points – Now we wish to perform the previous problem in hardware. I have laid out the overall circuit framework, but you need to fill in the boxes with the appropriate gate logic. In the top box draw a circuit (using simple AND, OR, or NOT gates) to detect when the Hex Nybble In contains a value that is greater than 9. In the bottom box draw a circuit to send the correct values to the adder for when the input is greater than 9 and when the input is less than or equal to 9 (the solution to this task is surprisingly simple, and not all of the information coming in to this box will be required in the solution). You will need to consult the table on the previous page.

Before we look at the circuit solution, we first need to think about how to detect whether or not a 4-bit number is greater than 9. All such numbers, in binary, have their #8 bit (the leftmost bit) set to 1. Binary numbers between 10 and 15 (inclusive) will also have either the #4 bit set or the #2 bit set, or both. Numbers less than or equal to 9 might have the #8 bit set, and either the #4 bit set or the #2 bit set, but *not all at the same time*. Whether or not the #1 (rightmost) bit is set is immaterial:

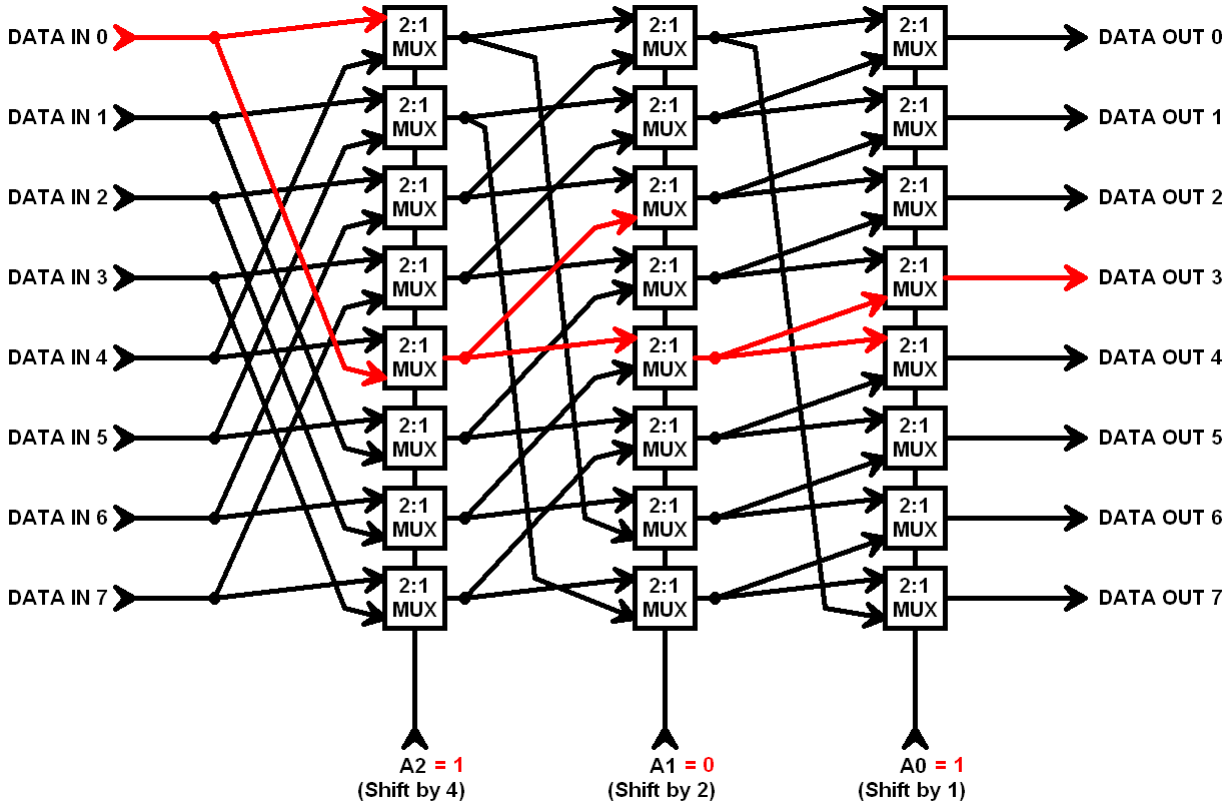
| Decimal | Binary | Bits Set |
|---|---|---|
| 10 | 1010 | #8, #2 |
| 11 | 1011 | #8, #2, #1 |
| 12 | 1100 | #8, #4 |
| 13 | 1101 | #8, #4, #1 |
| 14 | 1110 | #8, #4, #2 |
| 15 | 1111 | #8, #4, #2, #1 |




The solution to the "Hex>9" circuit is simply to AND the #8 input bit with the OR of the #4 and #2 input bits. The only way that the output will be 1 is if the input is 8+2=10, 8+4=12, or 8+4+2=14, and since the #1 bit is ignored it also detects 8+2+1=11, 8+4+1=13, and 8+4+2+1=15.

The adder gets most of its inputs from the "Always 1" and "Always 0" lines, setting up the pattern to always add 48 to the input nybble. The "Hex>9" line is connected to the lowest three bits of the adder, so if "Hex>9" is 1 then the sum is increased by the additional 7 needed to correct for the conversion between the ASCII digits and the ASCII letters. The adder handles carries internally. In reality this circuit can be simplified even more by replacing the upper four adder cells with half-adders instead of full-adders. Why have extra circuitry to only add zero?

<4> 5 Points – In the following barrel shifter circuit, ***trace the path*** for the DATA IN 0 bit when the shift pattern is 101 (i.e., A2=1, A1=0, A0=1), and show where that bit emerges.




Each 2:1 MUX (multiplexer) passes its top input to the output when its control line is 0, and passes its bottom input to the output when its control line is 1. In the first (leftmost) stage the control line is 1, so all multiplexers in that column are listening to their bottom inputs and are ignoring their top inputs. Even though the Data In 0 bit is going to multiplexers #0 and #4 in the first stage, only one of them (#4) passes that value on, to multiplexers #2 and #4 in the second stage. All multiplexers in the second stage are listening to their top inputs since the control line is 0, so multiplexer #4 in the second stage passes its top input on to multiplexers #3 and #4 in the third stage. All multiplexers in the third stage are listening to their bottom inputs since the control line is 1, so #3 passes its bottom input to the output, Data Out 3.

<5> 5 Points – Does the above circuit perform a left shift, a right shift, a left rotate, or a right rotate? (Consider bit position 0 to be rightmost and bit position 7 to be leftmost.)

The value input into Data In 0 appears in the output at Data Out 3, and because *all* data bits appear at the output offset from the input by the same amount this must be a rotate rather than a shift. Only if a 0 or 1 was always pumped in at one end or the other (thus losing data bits) would this be a shift. The immediate inference to draw is that this is a left-rotate-by-3, but since the control lines are 1-0-1, which has the binary value 5, this is actually a ***right-rotate-by-5***.

<6> 5 Points – I want to compute the polynomial $2x^2-3x+10$ using the floating point registers. The value for *x* is in `F0`, and the result must appear in `F1`. EXTRA CREDIT: +5 points if you modify no register other than `F1` in your solution.

The constants in this problem were chosen so that they are all legal values as the immediate portion of an ARM floating point instruction (the only legal immediate floating point constants are 0.5, 0.0 through 5.0, and 10.0). Remember that shifting is not allowed on the floating point side of the processor.

The easiest brute-force solution is to burn another register, such as **`F2`**, in order to hold temporary values.

```
MUFS F1,F0,F0       ; F1 ← X²
MUFS F1,F1,#2.0     ; F1 ← 2X²
MUFS F2,F0,#3.0     ; F2 ← 3X
SUFS F1,F1,F2       ; F1 ← 2X² – 3X
ADFS F1,F1,#10.0    ; F1 ← 2X² – 3X + 10
```

For extra credit, no extra registers may be used. Here is one approach, which requires multiple subtracts to perform the **`-3X`** portion.

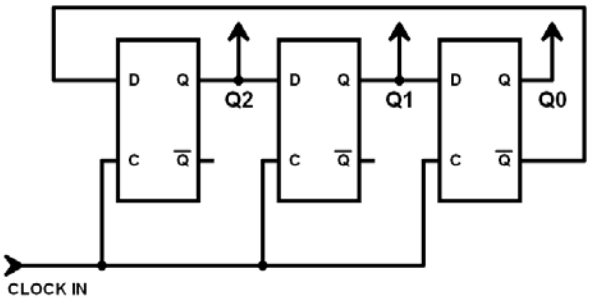
```
MUFS F1,F0,F0       ; F1 ← X²
ADFS F1,F1,F1       ; F1 ← 2X²
SUFS F1,F1,F0       ; F1 ← 2X² - X
SUFS F1,F1,F0       ; F1 ← 2X² - 2X
SUFS F1,F1,F0       ; F1 ← 2X² - 3X
ADFS F1,F1,#10.0    ; F1 ← 2X² - 3X + 10
```

Here is another extra credit approach, which requires that Horner's rule be applied to convert the polynomial from $2x^2 - 3x + 10$ into $(2x - 3)x + 10$, thus greatly simplifying the solution (two variants shown).

```
MUFS F1,F0,#2.0     ; F1 ← 2X
SUFS F1,F1,#3.0     ; F1 ← 2X - 3
MUFS F1,F1,F0       ; F1 ← 2X² - 3X
ADFS F1,F1,#10.0    ; F1 ← 2X² – 3X + 10
```

```
ADFS F1,F0,F0       ; F1 ← 2X
SUFS F1,F1,#3.0     ; F1 ← 2X - 3
MUFS F1,F1,F0       ; F1 ← 2X² - 3X
ADFS F1,F1,#10.0    ; F1 ← 2X² – 3X + 10
```

<7> 10 Points – Trace the following "walking ring" counter through several clock cycles, starting with all flip-flops set to zero. Will all possible states occur during the count? If not, which states will not occur?

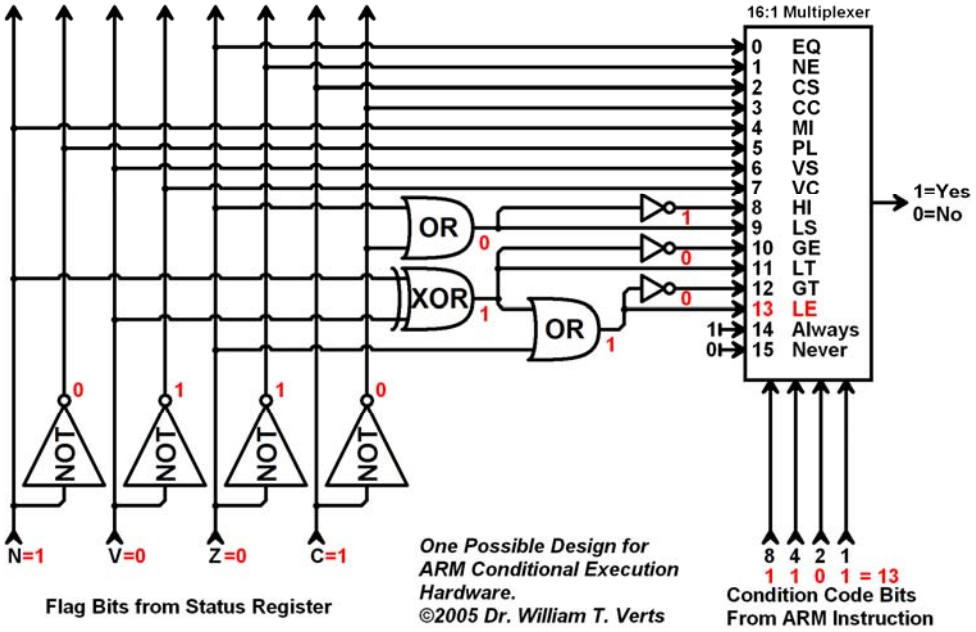



| Clock | Q2 | Q1 | Q0 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 |
| 2 | 1 | 1 | 0 |
| 3 | 1 | 1 | 1 |
| 4 | 0 | 1 | 1 |
| 5 | 0 | 0 | 1 |
| 6 | 0 | 0 | 0 |
| 7 | 1 | 0 | 0 |
| 8 | 1 | 1 | 0 |

Even though there are three flip-flops, ***not all*** of the $2^3$=8 possible states appear in the table. The sequence repeats after six clocks. The only patterns which do not appear are **`010`** and **`101`**. If either pattern is forced into the walking-ring counter through external means (i.e., through the set or clear inputs on the flip-flops), the two patterns will alternate on every clock cycle (`010`, to `101`, to `010`, forever).

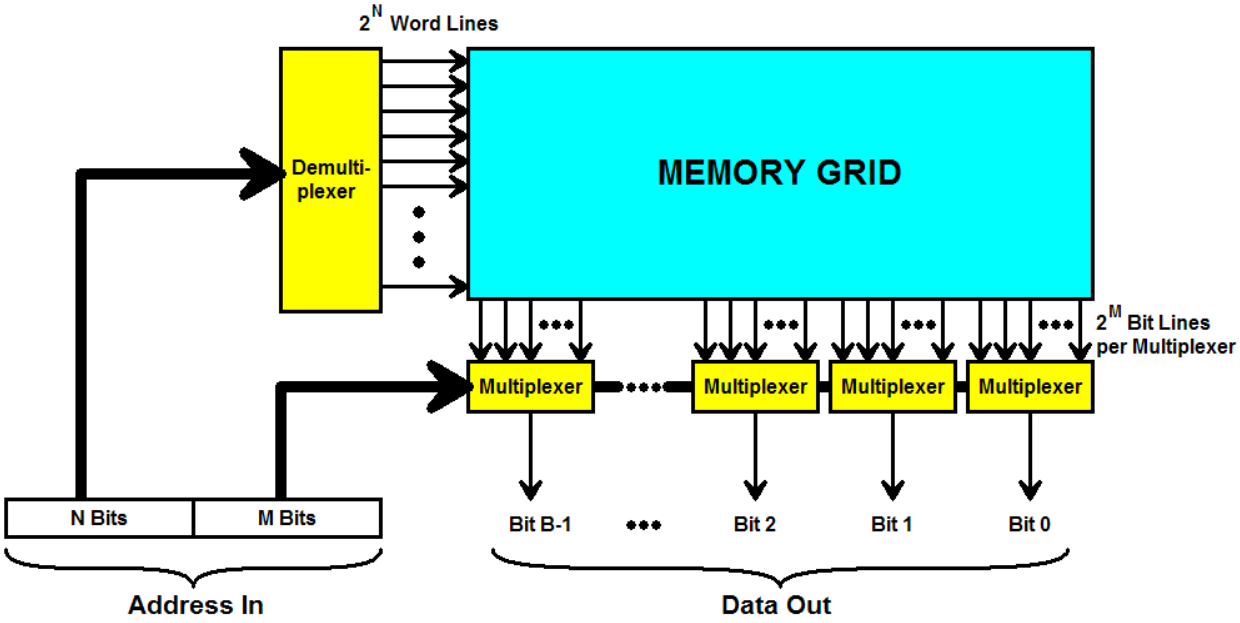
Note that a two-flip-flop walking-ring counter *will* clock through all four of its legal states, but any longer walking-ring counter *will not* clock though all possible states. The degenerate case is that a one-flip-flop walking-ring counter is the same as a toggle flip-flop, which alternates its output on every clock cycle.

<8> 10 Points – An ARM instruction sets the status bits as follows: N=1, V=0, C=1 and Z=0. The next instruction has the condition code 1101. Trace the circuit below and determine, ***yes-or-no***, if this new instruction will execute or not. Show the output of ***every*** gate.




The condition code 1101 means that input #13 on the multiplexer is the one that will be passed on to the output. Regardless of their values, the other multiplexer inputs are ignored. By tracing the circuit from the known values of the status bits, we determine that multiplexer input #13 is 1 (from the OR-gate), so ***yes***, this instruction will execute.

<9> 10 Points – Examine the memory grid below. With N=15, M=7, and B=24, then…




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

   $N + M = 15 + 7 =$ **22**

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

   $2^N = 2^{15} =$ **32,768**

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

   $B \times 2^M = 24 \times 2^7 = 3 \times 2^3 \times 2^7 = 3 \times 2^{10} = 3 \times 1024 =$ **3072**

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

   $2^{15}$ word lines × 24 × $2^7$ bit lines = $24 \times 2^{22} = 3 \times 2^{25} =$ **100,663,296**

   32768 × 3072 = **100,663,296**

5. …how many **<u>bytes</u> of memory** are there? (2 points)

   $2^{N+M}$ addresses × 3 bytes per address = $2^{22} \times 3 = 3 \times 4{,}194{,}304 =$ **12,582,912 bytes**

<10> 25 Points – In this final set of problems we have invented a new floating-point format that fits entirely into an 8-bit byte. In this representation, which we will call FLOAT8, there is one bit reserved for the sign (as usual), three bits for the biased exponent, and four bits for the mantissa. Plus-infinity is `01110000`, positive denormalized numbers are of the form `0000xxxx` (for `xxxx` not all zero), and zero is `00000000`.

1. (1 point) What is the ***bias*** value for the FLOAT8 format?

   If the number of bits in the exponent is *e*, then the bias is $2^{e-1}-1$. In this problem *e*=3, so the bias is computed as follows:

   $2^{3-1}-1 = 2^2 - 1 = \mathbf{3}$

2. (3 points) What is the ***binary scientific notation*** value ($\pm 1.xxxx \times 2^{\pm Y}$), the ***true binary value***, and the ***decimal*** value of the FLOAT8 number `01011101`?

   Biased exponent = $101_2 = 5$, therefore the exponent = 5 - 3 = 2

   Binary Scientific Notation: $+1.1101 \times 2^2$

   True Binary: +111.01

   Decimal: +7.25

3. (4 points) Show the ***largest possible normalized*** (non-infinite) positive FLOAT8 number, and then show its value in binary scientific notation, in true binary and in decimal.

   Since the number *must* be normalized, and cannot be infinite or NaN, the biased exponent cannot have all three bits set to 1. The next smallest number is 110, or 6. Stripping off the bias gives a true exponent of 6 - 3 = 3. To maximize the mantissa simply fill it with 1 bits.

   FLOAT8:

   Binary Scientific Notation: $+1.1111 \times 2^3$

   True Binary: +1111.1

   Decimal: +15.5

4. (4 points) Show the ***smallest possible normalized*** (non-denormal) positive FLOAT8 number, and then show its value in binary scientific notation, in true binary and in decimal.

Since the number *must* be normalized, and cannot be denormal or zero, the biased exponent cannot have all three bits set to 0. The next largest number is 001, or 1. Stripping off the bias gives a true exponent of 1 - 3 = -2. To minimize the mantissa simply fill it with 0 bits. The corresponding fraction is thus 1.0000, but we drop the leading 1 bit in building the mantissa in order to obtain one more bit of precision in the stored number.

FLOAT8:


Binary Scientific Notation: $+1.0000 \times 2^{-2}$

True Binary: +0.01

Decimal: +0.25

5. (5 points) I loaded a FLOAT8 number from memory location `Temp` into `R0`, using the `LDRB` instruction as shown below (the upper three bytes of `R0` are automatically set to zero). I want to convert the FLOAT8 number into ***fixed point*** and put the result into integer register `R1`, where the implied decimal point is in the middle of the word (16 bits to the left of the decimal and 16 to the right). Assuming that the FLOAT8 number is legal, normalized, and positive (not infinite, NaN, or denormal), write an ARM code fragment to create the fixed point value in `R1`. Use as many integer registers as you need, and do not worry about register transparency.

```
LDRB R0,Temp          ; R0 contains 000000XX (FLOAT8=XX)
MOV  R2,R0,LSR #4     ; R2 contains biased exponent
SUB  R2,R2,#3         ; R2 contains true exponent
AND  R1,R0,#15        ; R1 contains mantissa   (0.xxxx)
ORR  R1,R1,#16        ; R1 contains restored 1 (1.xxxx)
ADD  R2,R2,#12        ; R2 contains 16 - 4 + exponent
MOV  R1,R1,LSL R2     ; R1 contains final fixed point
```

This is probably the hardest single problem of the exam. It requires that you strip out both the biased exponent and the mantissa, un-bias the exponent, restore the missing 1 to the mantissa, and then shift the result into the proper place. The biased exponent goes into the rightmost bits of **`R2`**, where the bias is removed by subtracting 3 (determined in part 1 of this question). Everything but the mantissa goes into **`R1`**, because AND-ing the FLOAT8 number with 15 sets to 0 all bits except the rightmost four ($15_{10} = 1111_2$). OR-ing that result with 16 restores the

missing 1-bit ($16_{10} = 10000_2$). The decimal point on this value is now between the four bits of the mantissa and the restored 1, but it needs to be in the middle of the 32-bit word (with 16 bits on each side) if the exponent is zero – a difference of 16 - 4 = 12 bit positions. If the exponent is *not* zero then the value of the exponent needs to be *added to the 12* to get the correct shift factor. This value goes into **R2**, and **R2** is finally used as the amount to shift the fraction in **R1** to normalize it to the correct position.

6. (3 points) Instead of doing the computations in part 5, we now wish to convert positive FLOAT8 numbers into fixed point through a ***table-lookup*** algorithm. How many ***bytes*** of memory would the table require?

Since the FLOAT8 format is eight bits, but we are ignoring sign information, there are at most $2^7$=128 possible FLOAT8 values to consider (between zero = `00000000` and the largest NaN = `01111111`). At four bytes per word, the table (which contains 32-bit fixed-point values) occupies only 4 × 128 = **512 bytes** of memory.

In that group, there are 16 denormals and 16 values which are either infinite or NaN. If those 32 values are discarded the table will only contain 128 - 32 = 96 entries, or 4 × 96 = 384 bytes.

This problem does not really specify whether or not to include or discard non-normalized values, so either answer is acceptable. Unfortunately, the next problem depends on the full version of the table – restricting the size of the table has the side effect of requiring more than two instructions to effect the table lookup algorithm (checking for non-normalized values and adjusting the offsets into the table accordingly).

7. (5 points) Write an ARM code fragment to convert the FLOAT8 number in `R0` into a fixed point number in `R1` as you did in part 5, but this time using ***table lookup***. Call the table `TABLE8`. Your solution must contain exactly *two* new lines of ARM code (one point will be removed for every extraneous instruction).

```
LDRB R0,Temp              ; R0 contains 000000XX
ADR  Rx,TABLE8            ; Rx contains base address
LDR  R1,[Rx,R0,LSL #2]    ; R1 contains fixed point
```

In this solution **Rx** can be any otherwise unused register, including **R1** itself. Since the FLOAT8 number is being used as an index into an array of 4-byte numbers (the table of fixed point values), the index must be multiplied by 4 to convert it from a word index into a byte index. This is a standard array-reference approach on the ARM that uses the barrel shifter to shift the word index in **R0** by 2 bits in order to convert it into a byte offset without permanently changing **R0**.