5 Points – There is a value in register \( R0 \) which needs to have some bits set to 1, some bits set to 0, some bits complemented, and the rest left unchanged. The desired pattern is shown below, where “\( X \)” indicates a bit to be left unchanged and “\( - \)” indicates a bit to be complemented. Write the appropriate ARM assembly code to perform this task, using no more than three individual instructions. Extra credit: +2 points if you solve this problem in exactly two assembly language statements. Every instruction must be of the form \text{OPCODE} \ R0, R0, \#___,___ \text{ where constants are numbers in the range } [0...255] \text{ right-rotated by even numbers between 0 and 30.}

\[
\begin{array}{c}
\times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times \times 1 1 1 1 0 0 0 0 0 0 - - - \times \times \times \times \times \times \times \\
\end{array}
\]

Discussion:

The general solution is as follows:

- \text{ORR \ R0 with: } 00000000000011111111000000000000
- \text{AND \ R0 with: } 11111111111111111000111111111111
- \text{EOR \ R0 with: } 00000000000000000000111100000000

As it turns out, there isn’t a three-instruction solution, sorry, but there \textit{is} a two-instruction solution. The problem here is that the given AND-mask can’t be created in one ARM instruction (but it can be created in two instructions at the expense of another register). We need an \text{AND-NOT} instruction! Here is an acceptable four-instruction solution:

- \text{ORR \ R0,R0,#15,16} \quad \text{1111 right-rotated 16 bits}
- \text{MVN \ R1,#15,20} \quad \text{0000 right-rotated 20 bits}
- \text{AND \ R0,R0,R1} \quad \text{with 1’s everywhere else}
- \text{EOR \ R0,R0,#15,24} \quad \text{1111 right-rotated 24 bits}

The three sequences (\text{ORR, MVN+AND, EOR}) can be executed in any order and the result will be the same.

To reduce this to two instructions, you first execute an \text{ORR} of a mask suitable for both the \text{ORR} and the \text{AND} instructions, setting to 1 all the bits for the \text{AND} as well as for the \text{ORR}. Next, you \text{EOR} (exclusive-OR) with a mask suitable for both the \text{AND} and the \text{EOR} instructions to invert to 0 the bits set to 1 by the \text{ORR} as well as those that should be complemented in the first place. Here is the solution (which must be in this order):

- \text{ORR \ R0,R0,#255,20} \quad \text{11111111 right-rotated 20 bits}
- \text{EOR \ R0,R0,#255,24} \quad \text{11111111 right-rotated 24 bits}

This sequence is equivalent to the following mask patterns:

- \text{OR \ R0 with: } 0000000000001111111111110000000000
- \text{EOR \ R0 with: } 000000000000000000001111111111000000

Grading: Accept for full credit any solution, of any length, that accomplishes the task. Accept for +2 extra credit the two-instruction solution above (or anything equivalent). Remove 1 point for each coding error, but do not go below zero.
<2> 5 Points – A word-size memory location contains the hexadecimal value \(0x408C0000\), interpreted as *single precision floating point*. What is the equivalent binary fraction, expressed as \(\pm1.xxxxx\times2^{\pm Y}\)? What is its decimal (base 10) value?

**Binary fraction:**

The binary representation of the given value is \(010000001000110000000\ldots\), so the sign is 0, the biased exponent is 1000001, and the mantissa is 00011 (omitting trailing zeroes). In decimal the biased exponent is 129; subtracting the bias of 127 reveals that the true binary exponent is +2. The mantissa is missing the leading 1 to the left of the decimal point, so 1.00011 is its true value. Thus, the binary fraction is \(1.00011\times2^2\).

**Decimal value:**

The value \(1.00011\times2^2\) means 100.011, or \(4+\frac{1}{4}+\frac{1}{8} = 4+0.25+0.125 = 4.375 = 4\frac{3}{8}\).

Grading: Assign 3 points to the binary fraction and 2 points to the decimal value. Subtract 1 point for getting the sign wrong, 1 point for getting the exponent wrong, and 1 point for getting the mantissa wrong. Subtract 2 points for getting the decimal value wrong, but do not penalize students for correctly converting to decimal an incorrect binary fraction.

<3> 5 Points – If you interpret the same word as in question 2 as a *signed integer*, what is its correct decimal (base 10) value?

From \(408C0000\), we get: \(4\times16^7 + 0\times16^6 + 8\times16^5 + C\times16^4\)

From the equivalent binary, we get: \(2^{30} + 2^{23} + 2^{19} + 2^{18}\)

This computes to \(1,073,741,824 + 8,388,608 + 524,288 + 262,144 = 1,082,916,864\)

Grading: This was difficult to get correct without a calculator. Remove 1 point for using either power-form without computing the final number. Otherwise, remove 1 point for each visible computational error that leads to an incorrect final result, but do not go below zero. Remove all credit if there is a wrong final answer with no supporting work.

<4> 5 Points – Short Answer – Describe the function of the following recursive subroutine. What registers are modified? How are they changed? What is the maximum possible depth of the stack (in words) at the point where the basis case is detected?

```
SUB    STR    LR, [SP, #-4]!
STR    R0, [SP, #-4]!
MOV     R0, R0, LSR #1
ADDCS   R1, R1, #1
BLNE    SUB
LDR    R0, [SP], #4
LDR    PC, [SP], #4
```
This subroutine counts into $R1$ the number of 1-bits in $R0$. The subroutine works by right-shifting $R0$ into the carry bit, which is an indication whether or not $R1$ should be incremented (the ADDCS instruction means “ADD if Carry is Set”). Recursion happens with the successively smaller shifted $R0$ value until it becomes zero. Unwinding the recursive calls successively restores each earlier version of $R0$, back to its original value at the time of the initial call.

The maximum stack depth is proportional to the position of the left-most 1-bit in $R0$. If the left-most 1-bit is initially at the left end of the word (where the sign bit would be), then the stack will be 32 calls deep. Each call pushes two words (the return address and the saved $R0$ value), so the maximum stack depth is 64 words (256 bytes).

Grading: The minimum acceptable answer is “counts into $R1$ the number of 1-bits in $R0” and “maximum depth 64 words”. Accept for 3 points any reasonable written description of the process. Remove 1 point for minor errors, 2 points for major errors. Accept for 2 points the correct stack depth (remove 1 point for off-by-one or off-by-two errors such as 62, 63, 65, or 66 words, and remove 1 point for answering in bytes instead of in words).

<5> 10 Points – Show, in binary, the values of $R0$ and $R1$ after each of the instructions in the code fragment below (show only the rightmost six bits of each register, and write “???” in any slot where the value is unknown).

<table>
<thead>
<tr>
<th>Code</th>
<th>R0</th>
<th>R1</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV R0,#11</td>
<td>001011</td>
<td>??</td>
</tr>
<tr>
<td>MOV R1,#13</td>
<td>001011</td>
<td>001101</td>
</tr>
<tr>
<td>EOR R0,R0,R1</td>
<td>000110</td>
<td>001101</td>
</tr>
<tr>
<td>EOR R1,R0,R1</td>
<td>000110</td>
<td>001011</td>
</tr>
<tr>
<td>EOR R0,R0,R1</td>
<td>001101</td>
<td>001011</td>
</tr>
</tbody>
</table>

Grading: 1 point for each answer.

<6> 5 Points – Describe in words what happens to the values in $R0$ and $R1$ in the previous question as a result of the three EOR instructions.

Swaps the values in $R0$ and $R1$.

Grading: Accept for full credit any answer that indicates that the values in $R0$ and $R1$ are exchanged from their original values. Remove 2 points for minor errors (perhaps caused by errors in generating the values in problem #5), no credit for wildly wrong answers.
10 Points – The subroutine below sorts the numbers in R0 and R1 in ascending order (using pass-by-value-return). Rewrite the subroutine to be more efficient with respect to register usage and number of instructions, without changing any of the semantics of the subroutine. Don’t change the parameter passing mechanism, keep the routine fully transparent, don’t add any features, and don’t remove any functionality.

```
Sort     STR    LR, [SP, #-4]!
          STR    R2, [SP, #-4]!
          STR    R3, [SP, #-4]!
          STR    R4, [SP, #-4]!
          CMP    R0, R1
          BLE    Done
          MOV    R2, R1
          MOV    R1, R0
          MOV    R0, R2
Done     LDR    R4, [SP], #4
          LDR    R3, [SP], #4
          LDR    R2, [SP], #4
          LDR    LR, [SP], #4
          MOV    PC, LR
```

The first thing to notice is that the R3 and R4 registers are never used, so their push-pops can be eliminated. Similarly, since the routine never calls another subroutine the LR register is never modified, and so its corresponding push-pop can be eliminated as well. A minimal overhaul is thus:

```
Sort     STR    R2, [SP, #-4]!
          CMP    R0, R1
          BLE    Done
          MOV    R2, R1
          MOV    R1, R0
          MOV    R0, R2
Done     LDR    R2, [SP], #4
          MOV    PC, LR
```

Next, the inner section corresponding to an IF-statement can be tightened by exploiting conditional execution:

```
Sort     STR    R2, [SP, #-4]!
          CMP    R0, R1
          MOVGT  R2, R1
          MOVGT  R1, R0
          MOVGT  R0, R2
          LDR    R2, [SP], #4
          MOV    PC, LR
```
Finally, problems 5 and 6 demonstrated a method of exchanging two values by using exclusive-OR instructions. Doing so eliminates the need for the temporary register $R2$, and thus its push-pop can be eliminated. The tightest answer I can think of is thus:

```
Sort   CMP   R0,R1
       EORGT R0,R0,R1
       EORGT R1,R0,R1
       EORGT R0,R0,R1
       MOV   PC,LR
```

Grading: Give full credit for either of the first two answers. Remove 2 points each for forgetting to remove the push-pop for $R3$, for $R4$, and for $LR$. Remove 1 point for each syntax error or for each case where an illegal change was made (such as removing the push-pop for $R2$ without eliminating the need for $R2$, for example). I don’t expect the third approach to be a frequent answer, but it is sufficiently sophisticated that if anyone turns it in it should be worth +3 points extra credit.

<8> 8 Points – Register $R0$ contains the value $0xFFFFFFFF$. Show the contents of the stack, as well as any change in the position of the stack pointer, after each of the following instructions is executed (all four problems are independent from one another):

Grading: 2 points each. Remove 1 point each for minor errors.

1. STR R0, [SP, #−8]
2. STR R0, [SP, #−8]!
3. ADD SP, SP, #4
4. LDR R0, [SP], #4

---

The value $0xFFFFFFFF$ gets put into slot $0x008008$, but $SP$ doesn’t change.

The value $0xFFFFFFFF$ gets put into slot $0x008008$ and $SP$ changes to $0x008008$.

Register $R0$ gets the value $0x001A2203$ and $SP$ is changed to $0x008014$.

---

SP is increased to point at $0x008014$, but nothing else changes.
10 Points – In a hypothetical ARM system, an input device is memory-mapped onto a 8-bit memory location which has the symbolic label InPort defined as pointing to that address (i.e., the instruction LDRB Rx,InPort loads the selected register from the input device). A push-button is connected to the rightmost bit of that device, and is wired so that the input bit sees a value of 0 when the push-button is not pressed and sees a value of 1 when the push-button is pressed. The other 7 bits are connected to input devices we are not currently interested in, but which are still active and have values that will be read in at the same time. Write a subroutine called Wait_Button that polls the input device until the push-button is pressed. There are no parameters to this routine. As usual, your routine must be completely transparent to all modified registers.

```
Wait_Button   STR R0,[SP,#-4]!
Loop   LDRB R0,InPort
       ANDS R0,R0,#1
       BEQ Loop

       LDR R0,[SP],#4
       MOV PC,LR
```

The LDRB instruction means “load byte”, so LDRB R0,InPort loads the low order byte of R0 from the input port, but as a side effect of this instruction the upper three bytes of R0 are set to zero. The trick here is to mask off just the input bit and test the whole word against zero, then repeat until someone pushes the button. We need to have the input value in a register, so that register must be saved and restored. This is a terrible waste of computational resources, since the entire attention of the processor is focused on the central three instructions. No other processing can happen until a button press allows the processor out of this subroutine.

Grading: Accept any reasonable coded answer. Remove 1 point per syntax error, “bug” in the code, or for forgetting the code to keep the routine transparent, but do not go below zero.
5 Points – A byte-addressed computer system with 24-bit addresses (assume that the entire 16-megabyte address space is filled with real memory) has a direct-mapped cache as shown below:

1. How many bytes are in each cache block?
   \[ 2^6 = 64. \] There are 6 bits to specify the offset within the cache block, so that tells you the size of each block.

2. How many cache blocks are present?
   \[ 2^9 = 512. \] There are 9 bits to specify the cache block.

3. How many blocks of memory map onto each block in the cache?
   \[ 2^9 = 512. \] There are 24 total address bits, but the 15 low-order bits are already used to indicate a cache block number and an offset within a block. With 9 bits left over, there are \( 2^9 \) blocks that map onto each location in the cache.

4. If only a single program is allowed to run at any one time, how big can that program be before it cannot fit into the cache in its entirety?
   \[ 2^{15} = 32768. \] For any larger program there will be two bytes from different blocks that map onto the same byte of the cache.

5. What is the minimum distance between two bytes in the address space that will guarantee a cache collision?
   \[ 2^{15} + 1 = 32769. \] See part 4.

Grading: 1 point each. All or nothing.
5 Points – Short Answer – In caching, what are the advantages and disadvantages of the write-through strategy versus the write-back strategy?

In write-through every write to the cache is also written back to primary memory. This always keeps the cache consistent with primary memory, but it slows the system down. In write-back writes go only to the cache until a cache block is replaced; at that time the cache block is written back to primary memory. This tends to improve overall system performance, but the cache and primary memory are out of synchronization with each other (some other memory-mapped I/O device referencing primary memory may get stale data).

Grading: Accept any reasonable answer. Look for discussion of consistency versus system performance. Remove 2 points for an answer that misses something obvious.

5 Points – Trace the circuit to the right and show the result of every gate or block for the case where Operand A = 10 and Operand B = 11. What is the overall function of the circuit? What is the purpose of the four AND gates?

The circuit multiplies two 2-bit numbers to create a 4-bit product. The AND-gates generate all possible binary partial-products, which are then added together to form the true product. For A=10 and B=11, the Output=0110 (or in decimal, 2×3=6).

Grading: Assign 3 points for the correct answer and outputs from all gates (remove 1 point if the outputs of the AND-gates are not labeled). Assign 1 point for the explanation of the AND-gates. Assign 1 point for the explanation of the circuit in general.
12 Points – In a memory system with N=8, M=8, and 8 Data Out bits, then…

1. …how many address lines are there?
   \[ N + M = 8 + 8 = 16. \]

2. …how many word lines are there?
   \[ 2^N = 2^8 = 256. \]

3. …how many bit lines are there?
   \[ 8 \times 2^M = 8 \times 2^8 = 2^3 \times 2^8 = 2^{11} = 2048. \]

4. …how many memory bits are there?
   \[ 2^8 \times 2^{11} = 2^{19} = 524,288. \]

5. …how many AND-gates are there? (in the entire circuit)
   \[ 2^8 + 2^{11} = 256 + 2048 = 2304. \]

6. …how many bytes of memory are there?
   \[ 2^{N+M} = 2^{16} = 65,536. \]

Grading: 2 points each. Accept power-terms such as \(2^8\) instead of 256. Remove 1 point for “off-by-one” errors such as \(2^7\) or \(2^9\) in a case where the expected answer is \(2^8\).
5 Points – Is the situation in the previous problem minimal? Are there different values for M and N that result in the same number of memory bytes but fewer AND-gates?

No, it isn’t minimal. As an example, N=9 and M=7 gives $2^9 + 8 \times 2^7 = 512 + 1024 = 1536$ AND-gates.

Grading: It is not necessary to find the optimal solution, but it is necessary to show at least one such example to justify the “no” answer. Assign 3 points for answering “no” and assign 2 points for an example to justify the answer.

5 Points – Short Answer – What are the differences between and relative advantages and disadvantages of static RAM cells versus dynamic RAM cells? In what hardware situations would you use each one instead of the other?

Static RAM is fast and retains its contents indefinitely so long as the power is applied, but each cell requires six transistors. Dynamic RAM cells are smaller and cheaper than static cells since they only require a single transistor (and a capacitor) per cell, but since the capacitors “leak” their contents must be continuously refreshed to avoid data loss. Static RAM is used in cases where a small amount of very fast RAM is required (such as in cache). Dynamic RAM used in cases where large amounts of cheap RAM are required (such as in primary memory).

Grading: Assign 3 points for any reasonable answer that compares size, cost, or number of transistors to speed and/or data retention. Assign 2 points for any reasonable explanation of their uses.