CSC111 Assembly Language Programming

Spring 2026 Midterm Exam

Student Name: ID: _s

Total Points	

Note: This exam paper should be kept confidential and any dissemination violates copyright.

Q1	Q2	Q3	Q4	Q5	Q6
/10	/10	/20	/20	/20	/20

Q1 (10 points) Multiple-choice questions: enter your answer keys here:

1	2	3	4	5	6	7	8	9	10
B	C	A	D	B	B	B	B	C	A

For the following multiple-choice questions, each question has exactly one correct answer key. If multiple choices are correct, choose the option "All of the above". **Fill in the answer keys in the table above. (Answer keys written in the question area will not be counted.)**

1. In a 5-bit system, which statement is true about -16 (10000_2)?
   A. Its two's complement is 00000_2
   B. Its two's complement is itself
   C. It cannot be represented
   D. It equals +16
   ANS: B (most negative number maps to itself)

2. Signed overflow can occur when:
   A. Adding operands with different signs
   B. Subtracting operands with the same sign
   C. Adding two negatives
   D. Subtracting a negative from a negative
   ANS: C (same-sign add or different-sign subtract can overflow)

3. For char str = "ARM Assembly", what must the final byte be and what is the string's size in Bytes?
   A. 0x00; 13
   B. 0x20; 12
   C. 0x41; 13
   D. 0x79; 12 (ASCII hex code for lowercase 'y' is 0x79)
   ANS: A (includes space for NULL terminator; 12 chars + 1 NUL = 13)

4. Which of the following is not a barrel shifter operation in ARM?
   A. LSL
   B. ROR
   C. RRX
   D. ROL

ANS: D

5. How can a rotate-left by n bits be implemented on a 32-bit value using available rotate instructions?
   A. As RRX by n
   B. As ROR by 32 − n
   C. As LSL by 32 − n
   D. As ASR by n
   ANS: B

6. Which shift corresponds to signed division by a power of two with sign extension?
   A. LSR
   B. ASR
   C. LSL
   D. ROR
   ANS: B

7. Assume Little Endian and r0 = 0x20008000 with memory contents: [0x20008000]=0xEF, [0x20008001]=0xCD, [0x20008002]=0xAB, [0x20008003]=0x89; what does LDRH r1, [r0] load into r1?
   A) 0x0000EFCD
   B) 0x0000CDEF
   C) 0x89ABCDEF
   D) 0x000089AB
   ANS: B (Load halfword and zero-extension)

8. What happens when you execute LDRSB r1, [r0] and the byte at the memory location has value 0xEF?
   A) r1 = 0x000000EF
   B) r1 = 0xFFFFFFEF
   C) r1 = 0xEF000000
   D) r1 = 0x0000FFEF
   ANS: B (LDRSB sign-extends, 0xEF is negative so extends with 1s)

9. Which pattern correctly toggles bit 5 of r0 without affecting other bits?
   A. ORRS r0, r0, r4, LSL #5 with r4 = 1
   B. ANDS r0, r0, r4, LSL #5 with r4 = 1
   C. EORS r0, r0, r4, LSL #5 with r4 = 1
   D. BICS r0, r0, r4, LSL #5 with r4 = 1
   ANS: C (EORS instruction is the bitwise Exclusive OR (EOR) instruction with the S suffix, which means it performs the EOR operation and updates the condition flags (N, Z, C, V) based on the result.)

10. Which single instruction multiplies a register by 17 using the barrel shifter on the second operand?
    A. ADD r4, r4, r4, LSL #4
    B. RSB r5, r5, r5, LSL #5
    C. ADD r1, r0, r0, ASR #3
    D. MUL r1, r0, #17
    ANS: A

Q2. (10 points) Binary numbers

Assuming a 4-bit system. Show the equivalent decimal values when the data is interpreted as unsigned binary or signed binary.

Binary Value	Signed Decimal Value	Unsigned Decimal Value
1000		
0111		
1101		
0000		
1110		

ANS:

Binary Value	Signed Decimal Value	Unsigned Decimal Value
1000	-8	8
0111	7	7
1101	-3	13
0000	0	0
1110	-2	14

Q3. (20 points) Unsigned and signed arithmetic

Assume a 4-bit system. For each of the following operations, give the equivalent arithmetic calculation in decimal, and the NZCV flags, based on the first row that has been given to you. GT denotes Ground Truth value when there is a carry or overflow.

Operation	Result in binary	Equivalent unsigned arithmetic in decimal	Equivalent signed arithmetic in decimal	NZCV
0100 - 1110	0110	**4 - 14 = 6 (GT=-10)**	**4 - -2 = 6**	**0000**
0100 + 0110	1010			
1100 + 1110	1010			
1100 + 1010	0110			
0100 - 0110	1110			
1100 - 0110	0110			

ANS:

Review:

1) N (Negative): 1 if the highest bit of the result is 1.
2) Z (Zero): 1 if the result is 0000.
3) C (Carry): In addition, 1 if unsigned result > 15. In subtraction, 1 if there is NO borrow (i.e. unsigned A $\geqslant$ unsigned B).
4) V (Overflow): 1 if the signed result falls outside the -8 to 7 range.
5) Unsigned 4-bit range: 0 to 15, Signed 4-bit range: −8 to +7.

Operation	Result in binary	Equivalent unsigned arithmetic in decimal	Equivalent signed arithmetic in decimal	NZC V
0100 - 1110	0110	**4 - 14 = 6 (GT=-10)**	**4 - -2 = 6**	**0000**
0100 + 0110	1010	4 + 6 = 10	4 + 6 = -6 (GT=10)	1001
1100 + 1110	1010	12 + 14 = 10 (GT=26)	-4 + -2 = -6	1010
1100 + 1010	0110	12 + 10 = 6 (GT=22)	-4 + -6 = 6 (GT=-10)	0011
0100 - 0110	1110	4 - 6 = 14 (GT=-2)	4 - 6 = -2	1000
1100 - 0110	0110	12 - 6 = 6	-4 - 6 = 6 (GT=-10)	0011

Q4. (20 points) Bit manipulations

(a) (5 points) Assume an 8-bit system. Copy the top three bits 7..5 in R0 to bits 4..2 in R1, and keep all other bits in R1 unchanged.

ANS: R1 = (R1 & 0b11100011) | ((R0 >> 3) & 0b00011100)
Or R1 = (R1 & 0xE3) | ((R0 >> 3) & 0x1C)

Explanation: Clear bits 4..2 in `R1`, shift the desired bits from `R0` into positions 4..2, then combine them with OR so all other bits in `R1` stay unchanged. In `R1 = (R1 & 0b11100011) | ((R0 >> 3) & 0b00011100)`, the first part clears the destination bits, and the second part moves and inserts the source bits. (Explanations are optional.)

(b) (5 points) Assume an 8-bit system. Copy bits 5..2 in R0 to bits 3..0 in R1, and keep the upper four bits in R1 unchanged.

ANS: R1 = (R1 & 0b11110000) | ((R0 >> 2) & 0b00001111)
Or R1 = (R1 & 0xF0) | ((R0 >> 2) & 0x0F)

Explanation: Mask `R1` with `0b11110000` to keep the upper four bits and clear bits 3..0, then shift `R0` right by 2 so bits 5..2 line up with bits 3..0. Finally, OR the two parts together to copy those four bits into `R1` while leaving the upper four bits unchanged.

(c) (10 points) Assume a 32-bit system. R0 and R1 are initialized with the values below. Write a sequence of assembly instructions that would produce the values in registers R2–R5, from R0 and R1. Do not use MOVW + MOVT). Do not use pseudo-instructions LDR to load large constants into a register.

Given:

R0	0xDEADBEEF
R1	0x0000FFFF

Produce:

R2	0xDEAD4110
R3	0x0000DEAD
R4	0xBEEFFFFF

ANS:

```
EOR R2, R0, R1
LSR R3, R0, #16
LSL R4, R0, #16
ORR R4, R4, R1
```

Alternative 1-instruction approach for R4:

```
ORR R4, R1, R0, LSL #16
```

Alternative equivalent sequence for R4:

```
MOV R4, R1
BFI R4, R0, #16, #16
```

Explanations (optional):

1. Instruction: **EOR R2, R0, R1**
   - Performs a bitwise Exclusive-OR (XOR) between R0 and R1.
   - R2 = 0xDEADBEEF ^ 0x0000FFFF = 0xDEAD4110
   - (Note: In hex, *E ^ F = 1, F ^ F = 0, hence BEEF ^ FFFF = 4110)*
2. Instruction: **LSR R3, R0, #16**
   - Logical Shift Right by 16 bits.
   - 0xDEADBEEF >> 16 = 0x0000DEAD (unlike ASR, LSR fills the vacated most significant bits with 0s regardless of the sign bit).
   - R3 = 0x0000DEAD

3. Instruction: **LSL R4, R0, #16** followed by **ORR R4, R1**
    - **Step 1:** Logical Shift Left by 16 bits. Only the lower 16 bits (BEEF) remain visible, shifted into the Most Significant Halfword position:
      R4 = 0xBEEF0000
    - **Step 2:** Bitwise OR merges with R1:
      R4 = R4 | R1 = 0xBEEF0000 | 0x0000FFFF = 0xBEEFFFFF
    - Hence, **R4 = 0xBEEFFFFF**.

Explanation of the **BFI Alternative Sequence:**

- MOV R4, R1 ; Copies R1 to R4 → R4 = 0x0000FFFF
- BFI R4, R0, #16, #16 ; Bit Field Insert: inserts the lowest 16 bits ([15:0]) of R0 (0xBEEF) into bits [31:16] of R4. This yields R4 = 0xBEEFFFFF.

Q5. (20 points) Program execution tracing

Assume a 32-bit system with **big-endian** byte order. The following assembly program operates on memory and registers. Fill in the values of registers R1 through R7 and the final memory contents after the entire program has finished executing. Write your answers in hexadecimal format. Assume initially NZCV = 0000.

Assembly program:

```
LDR   R1, =0x20000000
LDR   R2, [R1]
LDR   R3, [R1, #4]!
LSR   R4, R2, #8
MOVW  R5, #15
LSL   R6, R5, #4
BIC   R7, R2, R6
STR   R7, [R1, #8]
```

Initial memory contents:

Addr	Content															
0x20000000	12	34	56	78	9A	BC	DE	F0	11	22	33	44	55	66	77	88

Final register values:

R1	R2	R3	R4
R5	R6	R7	

Final memory contents:

Addr	Content															
0x20000000																

ANS:

Final register values (14 pts)

R1	R2	R3	R4
0x20000004	0x12345678	0x9ABCDEF0	0x00123456
R5	R6	R7	
0x0000000F	0x000000F0	0x12345608	

Final memory contents (6 pts):

Addr	Content															
0x20000000	12	34	56	78	9A	BC	DE	F0	11	22	33	44	12	34	56	08

Explanations (not required in exam):

1. **LDR R1, =0x20000000** → Initializes R1 with the base memory address 0x20000000.
2. **LDR R2, [R1]** → Loads 4 bytes from 0x20000000. In big-endian format, the bytes 12 34 56 78 translate directly to 0x12345678.
3. **LDR R3, [R1, #4]!** → **Pre-indexed load**. It first increments R1 to 0x20000004 (updating the register due to the ! suffix), then loads 4 bytes from that new address (9A BC DE F0) into R3.
4. **LSR R4, R2, #8** → Logical shift right 0x12345678 by 8 bits (shifting out the 78). Fills the top with zeros. R4 becomes 0x00123456.
5. **MOVW R5, #15** → Loads the decimal value 15 (hex 0x0000000F) into R5. MOVW R5, #15 places 0x000F in the lower 16 bits and forces 0x0000 into the upper 16 bits, resulting in 0x0000000F regardless of the initial value of R5. (MOV R5, #15 produces the same result as MOVW R5, #15 for the small immediate value 15 that fits within 8 bits.)
6. **LSL R6, R5, #4** → Logical shift left 0x0000000F by 4 bits. R6 becomes 0x000000F0.
7. **BIC R7, R2, R6** → Bit Clear (AND NOT). It clears the bits in R2 (0x12345678) where there are 1s in R6 (0x000000F0). The 7 in 78 is 0111 in binary, and F is 1111, so clearing it yields 0000. R7 becomes 0x12345608.
8. **STR R7, [R1, #8]** → Stores R7 (0x12345608) to memory. The address is R1 + 8 = 0x20000004 + 8 = 0x2000000C. In big-endian, it writes 12 34 56 08 sequentially into the memory bytes 0x2000000C through 0x2000000F replacing 55 66 77 88.

Q6. (20 points) Assume a 32-bit system. Consider an array[] of 20 **unsigned integers** (each integer is 4 bytes). Assume register R0 holds the base memory address of array, i.e., array[0]. A compiler associates **unsigned integer** variables z and y with registers R1 and R2, respectively. Translate this C program into ARM assembly language based on the provided comments. Use R3 as a temporary register. (Note: You may choose to use either pre-index, or pre-index with update, or post-index addressing.)

C program:

```
uint32_t z = array[2] - y;
array[3] = z * 8;
array[4] = z / 2;
array[5] = z + 12;
array[6] = z | 0x0F;
```

Assembly program:

_____; load array[2] into temporary register R3

_____; z = array[2] - y

_____; R3 = z * 8

_____; store into array[3]

_____; R3 = z / 2

_____; store into array[4]

_____; R3 = z + 12

_____; store into array[5]

_____; R3 = z | 0x0F

_____; store into array[6]

ANS:

Version 1: Using pre-index addressing

LDR R3, [R0, #8] ; load array[2] (offset 2*4 = 8) into a temp register
SUB R1, R3, R2 ; z = array[2] - y
LSL R3, R1, #3 ; R3 = z * 8
STR R3, [R0, #12] ; store into array[3] (offset 3*4 = 12)
LSR R3, R1, #1 ; R3 = z / 2
STR R3, [R0, #16] ; store into array[4] (offset 4*4 = 16)
ADD R3, R1, #12 ; R3 = z + 12
STR R3, [R0, #20] ; store into array[5] (offset 5*4 = 20)
ORR R3, R1, #0x0F ; R3 = z | 0x0F
STR R3, [R0, #24] ; store into array[6] (offset 6*4 = 24)

Version 2: Using pre-index with update addressing.

LDR R3, [R0, #8]! ; load array[2] and update R0 to point to array[2]
SUB R1, R3, R2 ; z = array[2] - y
LSL R3, R1, #3 ; R3 = z * 8
STR R3, [R0, #4]! ; update R0 to array[3], store z * 8
LSR R3, R1, #1 ; R3 = z / 2

```
STR  R3, [R0, #4]!    ; update R0 to array[4], store z / 2
ADD  R3, R1, #12      ; R3 = z + 12
STR  R3, [R0, #4]!    ; update R0 to array[5], store z + 12
ORR  R3, R1, #0x0F    ; R3 = z | 0x0F
STR  R3, [R0, #4]!    ; update R0 to array[6], store z | 0x0F
```

Version 3: Using post-index addressing.

```
LDR  R3, [R0, #8]     ; load array[2] into R3
SUB  R1, R3, R2       ; z = array[2] - y
ADD  R0, R0, #12      ; increment R0 to point to array[3]
LSL  R3, R1, #3       ; R3 = z * 8
STR  R3, [R0], #4     ; store into array[3], then update R0 to array[4]
LSR  R3, R1, #1       ; R3 = z / 2
STR  R3, [R0], #4     ; store into array[4], then update R0 to array[5]
ADD  R3, R1, #12      ; R3 = z + 12
STR  R3, [R0], #4     ; store into array[5], then update R0 to array[6]
ORR  R3, R1, #0x0F    ; R3 = z | 0x0F
STR  R3, [R0]         ; store into array[6]
```

Grading note: Since z is an unsigned integer, LSL R3, R1, #3 is the correct instruction for R3 = z * 8, and LSR R3, R1, #1 is the correct instruction for R3 = z / 2. If a student writes MUL R3, R1, #8 or UDIV R3, R1, #2, this is invalid syntax because ARM MUL and UDIV instructions require all operands to be registers (immediate values are not allowed). To use multiplication or division, the constant must first be loaded into a register, e.g.:

```
MOV R4, #8
MUL R3, R1, R4
```

or

```
MOV R4, #2
UDIV R3, R1, R4
```