CSC111 Assembly Language Programming
Spring 2026 Midterm Exam (B Version) ANS

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	B	B	C	C	A	B	B	B	C

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, adding unsigned ints 28 and 6 sets which condition?
   A. No flags set
   B. Carry flag set
   C. Overflow flag set
   D. Zero flag set
   ANS: B (true sum 34 exceeds $2^5-1=31$)
2. In a 5-bit system, subtracting unsigned ints 3 - 5 results in which carry/borrow status?
   A. Carry=1 (Borrow=0)
   B. Carry=0 (Borrow=1)
   C. Carry=1 (Borrow=1)
   D. Carry=0 (Borrow=0)
   ANS: B (3 < 5, so a borrow occurs. ARM defines C = NOT Borrow, therefore C = 0.)
3. On ARM Cortex-M3, the borrow and carry flags relation is:
   A. Carry = Borrow
   B. Carry = NOT Borrow
   C. Borrow always 0
   D. Carry always 0
   ANS: B
4. In two's complement, TC(x) can be obtained by:
   A. Invert bits
   B. Invert bits and subtract one
   C. Invert bits and add one
   D. Add one then invert bits
   ANS: C

5. What is the bit width of each register in ARM Cortex-M processors?
   A) 16 bits
   B) 24 bits
   C) 32 bits
   D) 64 bits
   ANS: C

6. Which registers are considered "Low Registers" in ARM Cortex-M and can be accessed by any instruction?
   A) R0-R7
   B) R8-R12
   C) R13-R15
   D) R0-R12
   ANS: A

7. In ARM subtraction, what does the carry flag C indicate when a borrow occurs in SUBS?
   A. C = 1 when there is a borrow
   B. C = 0 when there is a borrow
   C. C toggles regardless of borrow
   D. C is always preserved from the previous instruction
   ANS: B (Carry equals not Borrow)

8. Which instruction performs reverse subtraction in ARM?
   A. SBC
   B. RSB
   C. SUB
   D. ADC
   ANS: B (RSB (Reverse Subtract) instruction subtracts the first operand (Rn) from the second operand (Operand2))

9. 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)

10. In LDR r1, [r0, r2, LSL #2], what is the effective address used?
    A) r0 + r2
    B) r0 + (r2 × 2)
    C) r0 + (r2 × 4)
    D) r0 + (r2 × 8)
    ANS: C

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
1001		
0110		
1100		
0001		
1111		

ANS:

Binary Value	Signed Decimal Value	Unsigned Decimal Value
1001	-7	9
0110	6	6
1100	-4	12
0001	1	1
1111	-1	15

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
0011 - 1101	0110	**3 - 13 = 6 (GT=-10)**	**3 - -3 = 6**	**0000**
0011 + 0111	1010			
1011 + 1101	1000			
1011 + 1001	0100			
0011 - 0101	1110			
1011 - 0101	0110			

ANS:

Review:

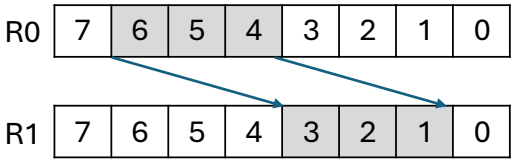
1) N (Negative): 1 if the highest bit of the result is 1.
2) Z (Zero): 1 if the result is 0000.
3) C (Carry): In addition, 1 if unsigned result > 15. In subtraction, 1 if there is NO borrow (i.e. unsigned A ⩾ unsigned B).
4) V (Overflow): 1 if the signed result falls outside the -8 to 7 range.
5) Unsigned 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
0011 - 1101	0110	**3 - 13 = 6 (GT=-10)**	**3 - -3 = 6**	**0000**
0011 + 0111	1010	3 + 7 = 10	3 + 7 = -6 (GT=10)	1001
1011 + 1101	1000	11 + 13 = 8 (GT=24)	-5 + -3 = -8	1010
1011 + 1001	0100	11 + 9 = 4 (GT=20)	-5 + -7 = 4 (GT=-12)	0011
0011 - 0101	1110	3 - 5 = 14 (GT=-2)	3 - 5 = -2	1000
1011 - 0101	0110	11 - 5 = 6	-5 - 5 = 6 (GT=-10)	0011

Note: for the row 1011 + 1101 = 1000, the signed description -5 + -3 = -8 is exactly at the boundary of the signed range [−8, 7]. V=0 here because −8 is representable with signed 4-bit int.

Q4. (20 points) Bit manipulations

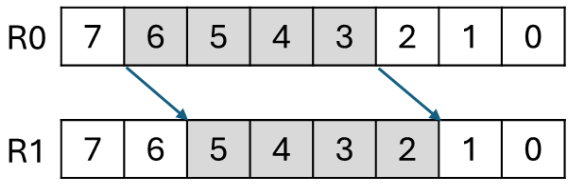
(a) (5 points) Assume an 8-bit system. Copy the top three bits 6..4 in R0 to bits 3..1 in R1, and keep all other bits in R1 unchanged.



ANS: Arithmetic:
R1 = (R1 & 0b11110001) | ((R0 >> 3) & 0b00001110)
Or R1 = (R1 & 0b11110001) | ((R0 & 0b01110000) >> 3)

ARM Assembly:
AND R1, R1, #0xF1 ; clear destination bits 3..1
AND R2, R0, #0x70 ; mask source bits 6..4
ORR R1, R1, R2, LSR #3 ; shift and OR in one instruction

(b) (5 points) Assume an 8-bit system. Copy bits 6..3 in R0 to bits 5..2 in R1, and keep R1[7:6] and R1[1:0] unchanged.



ANS: Arithmetic:
R1 = (R1 & 0b11000011) | ((R0 & 0b01111000) >> 1)

ARM Assembly:
AND R1, R1, #0xC3 ; R1 = R1 & 0b11000011 (clear bits 5..2)
AND R2, R0, #0x78 ; mask source bits 6..3 of R0 (0b01111000)
ORR R1, R1, R2, LSR #1 ; shift right 1 and OR into R1

(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	0xCAFEBABE
R1	0x0000FFFF

Produce:

R2	0xCAFE4541
R3	0x0000CAFE
R4	0xBABEFFFF

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

Explanation (optional):

1. Instruction: **EOR R2, R0, R1**
   - Performs a bitwise Exclusive-OR (XOR) between R0 and R1.
   - R2 = 0xCAFEBABE ^ 0x0000FFFF = 0xCAFE4541
   - (Note: In hex, *B ^ F = 4, A ^ F = 5, B ^ F = 4, E ^ F = 1, hence BABE ^ FFFF = 4541)*
2. Instruction: **LSR R3, R0, #16**
   - Logical Shift Right by 16 bits.
   - 0xCAFEBABE >> 16 = 0x0000CAFE (unlike ASR, LSR fills the vacated most significant bits with 0s regardless of the sign bit).

- R3 = 0x0000CAFE

3. Instruction: **LSL R4, R0, #16** followed by **ORR R4, R4, R1**
   - **Step 1:** Logical Shift Left by 16 bits. Only the lower 16 bits (BABE) remain visible, shifted into the Most Significant Halfword position:
     R4 = 0xBABE0000
   - **Step 2:** Bitwise OR merges with R1:
     R4 = 0xBABE0000 | 0x0000FFFF = 0xBABEFFFF
   - Hence, **R4 = 0xBABEFFFF**.

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 (0xBABE) into bits [31:16] of R4. This yields R4 = 0xBABEFFFF.

Q5. (20 points) Program execution tracing

Assume a 32-bit system with **big-endian** memory addressing. 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, #12
MOVW  R5, #7
LSL   R6, R5, #8
BIC   R7, R2, R6
STR   R7, [R1, #8]
```

Initial memory contents:

Addr	Content															
0x20000000	AB	CD	EF	01	23	45	67	89	FE	DC	BA	98	76	54	32	10

Final register values:

R1	R2	R3	R4
R5	R6	R7	

Final memory contents:

Addr	Content															
0x20000000																

ANS:

Final register values:

R1	R2	R3	R4
0x20000004	0xABCDEF01	0x23456789	0x000ABCDE
R5	R6	R7	
0x00000007	0x00000700	0xABCDE801	

Final memory contents:

Addr	Content															
0x20000000	AB	CD	EF	01	23	45	67	89	FE	DC	BA	98	AB	CD	E8	01

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 AB CD EF 01 translate directly to 0xABCDEF01.
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 (23 45 67 89) into R3.
4. **LSR R4, R2, #12** → Logical shift right 0xABCDEF01 by 12 bits (shifting out the F01). Fills the top with zeros. R4 becomes 0x000ABCDE.
5. **MOVW R5, #7** → Loads the decimal value 7 (hex 0x00000007) into R5. MOVW R5, #7 places 0x0007 in the lower 16 bits and forces 0x0000 into the upper 16 bits, resulting in 0x00000007 regardless of the initial value of R5. (MOV R5, #7 produces the same result as MOVW R5, #7 for the small immediate value 7 that fits within 8 bits.)
6. **LSL R6, R5, #8** → Logical shift left 0x00000007 by 8 bits. R6 becomes 0x00000700.
7. **BIC R7, R2, R6** →computes R7 = R2 & ~R6. The mask R6 = 0x00000700 has bits 10, 9, and 8 set (0111 0000 0000 in binary). These three bits fall within the F nibble of 0xABCDEF01 — specifically, 1111 at bits 11..8 becomes 1000 after clearing bits 10..8, converting 0xEF → 0xE8. All other bits are unaffected. R7 = 0xABCDE801.
8. **STR R7, [R1, #8]** → Stores R7 (0xABCDE801) to memory. The address is R1 + 8 = 0x20000004 + 8 = 0x2000000C. In big-endian, it writes AB CD E8 01 sequentially into the memory bytes 0x2000000C through 0x2000000F replacing 76 54 32 10.

Q6. (20 points) Assume a 32-bit system. Consider an array[] of 20 integers (each integer is 4 bytes). Assume register R0 holds the base memory address of array, i.e., array[0]. A compiler associates 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: Assume z is an unsigned integer. You may choose to use either pre-index, or pre-index with update, or post-index addressing.)

C program:

```
uint32_t z = array[3] - y;
array[^4] = z * 16;
array[^5] = z / 4;
array[^6] = z + 20;
array[^7] = z | 0x1F;
```

Assembly program:

_____; load array[3] into temporary register R3

_____; z = array[3] - y

_____; R3 = z * 16

_____; store into array[4]

_____; R3 = z / 4

_____; store into array[5]

_____; R3 = z + 20

_____; store into array[6]

_____; R3 = z | 0x1F

_____; store into array[7]

ANS: (For the division by 4, because z is an unsigned integer, LSR R3, R1, #2 is the mathematically correct shift. However, if the student uses UDIV R3, R1, #4, this is invalid syntax as UDIV requires a register for the denominator, not an immediate, so you need to assign #4 to a register before using DIV).

Version 1: Using pre-index addressing

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

```
ORR  R3, R1, #0x1F    ; R3 = z | 0x1F
STR  R3, [R0, #28]    ; store into array[7] (offset 7*4 = 28)
```

Version 2: Using pre-index with update addressing.

```
LDR  R3, [R0, #12]!   ; load array[3] and update R0 to point to array[3]
SUB  R1, R3, R2       ; z = array[3] - y
LSL  R3, R1, #4       ; R3 = z * 16
STR  R3, [R0, #4]!    ; update R0 to array[4], store z * 16
LSR  R3, R1, #2       ; R3 = z / 4
STR  R3, [R0, #4]!    ; update R0 to array[5], store z / 4
ADD  R3, R1, #20      ; R3 = z + 20
STR  R3, [R0, #4]!    ; update R0 to array[6], store z + 20
ORR  R3, R1, #0x1F    ; R3 = z | 0x1F
STR  R3, [R0, #4]!    ; update R0 to array[7], store z | 0x1F
```

Version 3: Using post-index addressing.

```
LDR  R3, [R0, #12]    ; load array[3] into R3
SUB  R1, R3, R2       ; z = array[3] - y
ADD  R0, R0, #16      ; increment R0 to point to array[4]
LSL  R3, R1, #4       ; R3 = z * 16
STR  R3, [R0], #4     ; store into array[4], then update R0 to array[5]
LSR  R3, R1, #2       ; R3 = z / 4
STR  R3, [R0], #4     ; store into array[5], then update R0 to array[6]
ADD  R3, R1, #20      ; R3 = z + 20
STR  R3, [R0], #4     ; store into array[6], then update R0 to array[7]
ORR  R3, R1, #0x1F    ; R3 = z | 0x1F
STR  R3, [R0]         ; store into array[7]
```