Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C

Chapter 4
ARM Arithmetic and Logic Instructions
Exercises ANS

Z. Gu

Fall 2025

Acknowledgement: Lecture slides based on Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C, University of Maine https://web.eece.maine.edu/~zhu/book/

Bit Manipulations

- Compute register values after each instruction
- MOV R0, #0xABC
- MOV R1, #0xDEF
- AND R2, R0, R1
- ORR R3, R0, R1
- EOR R4, R0, R1
- ORN R5, R0, R1
- BIC r6, R0, R1

Bit Manipulations ANS

- MOV R0, #0xABC
- MOV R1, #0xDEF
- AND R2, R0, R1
- ORR R3, R0, R1
- EOR R4, R0, R1
- ORN R5, R0, R1
- BIC r6, R0, R1

- R0 = 0x0ABC
- R1 = 0x0DEF
- R2 = R0 AND R1 = 0x08AC
- R3 = R0 ORR R1 = 0x0FFF
- R4 = R0 EOR R1 = 0x0753
- R5 = R0 ORN R1 = R0 | (~R1) = 0xF210 (assuming 32-bit registers)
- r6 = r0 BIC R1 = r0 & (~R1) = 0x0533

AND Rn Operand2 performs bitwise conjunction on Rn and Operand2.

ORR performs bitwise OR on Rn and Operand2.

EOR performs bitwise XOR on Rn and Operand2.

ORN computes Rn OR NOT Operand2, i.e., Rn | (~Rm).

BIC computes Rn AND NOT Operand2, i.e., Rn & (~Rm).

(Hint: convert hex to binary, perform operation, and convert back to hex.)

Bit Manipulations

- Find the Register Value to Complement, CLEAR & SET 5th, 7th, 12th bit of the given value and also find the result: 0xDECB.

Bit Manipulations ANS

- Find the Register Value to Complement, CLEAR & SET 5th, 7th, 12th bit of the given value and also find the result: 0xDECB.
- ANS:
- Set/Clear/Complement
  - Set a bit: x |= mask, which forces those bit positions to 1 without changing others. (| is OR)
  - Clear a bit: x &= ~mask, which forces those bit positions to 0. (& is AND)
  - Complement (toggle) a bit: x ^= mask, which flips those bit positions. (^ is XOR)
- Masks for 5th, 7th, 12th
  - Bit 5 → 1<<5 = 0x20.
  - Bit 7 → 1<<7 = 0x80.
  - Bit 12 → 1<<12 = 0x1000.
- Combined mask = 0x1000 | 0x80 | 0x20 = 0x10A0 = 0b0001 0000 1010 0000
- Apply to 0xDECB
  - Set: 0xDECB | 0x10A0 = 0xEEEB.
  - Clear: 0xDECB & ~0x10A0 = 0xD84B.
  - Complement: 0xDECB ^ 0x10A0 = 0xC46B

Clearing a Register

- What are the different ways by which all bits in register r12 can be cleared? No other register is to be used.

Clearing a Register ANS

- What are the different ways by which all bits in register r12 can be cleared? No other register is to be used.
- Method 1: XOR with itself
  - EOR r12, r12, r12
- Method 2: AND with zero
  - AND r12, r12, #0
- Method 3: Move zero
  - MOV r12, #0
- Method 4: Subtract from itself
  - SUB r12, r12, r12

Set bits

- Write an instruction that sets bits 0, 4, and 12 in register r6 and leave the remaining bits unchanged
- Write an instruction that clears bits 0, 4, and 12 in register r6 and leave the remaining bits unchanged

Set bits ANS

- Write an instruction that sets bits 0, 4, and 12 in register r6 and leave the remaining bits unchanged
- ANS:
  - ORR r6, r6, #(1<<0) | (1<<4) | (1<<12)
- Write an instruction that clears bits 0, 4, and 12 in register r6 and leave the remaining bits unchanged
- ANS:
  - BIC r6, r6, #(1<<0) | (1<<4) | (1<<12)
  - Or: AND r6, r6, #~((1<<0) | (1<<4) | (1<<12))

Add two 128-bit numbers

- Add two 128-bit numbers, assuming one number is stored in r4, r5, r6, r7 registers and the other stored in r8, r9, r10, r11. Store the result in r0, r1, r2, r3.

Add two 128-bit numbers ANS

- Add two 128-bit numbers, assuming one number is stored in r4, r5, r6, r7 registers and the other stored in r8, r9, r10, r11. Store the result in r0, r1, r2, r3.
- ANS:
  - @ Add 128-bit numbers (r4,r5,r6,r7) + (r8,r9,r10,r11) = (r0,r1,r2,r3)
  - ADDS r0, r4, r8 @ Add lowest 32 bits with carry flag update
  - ADCS r1, r5, r9 @ Add with carry
  - ADCS r2, r6, r10 @ Add with carry
  - ADC r3, r7, r11 @ Add with carry (final)

Absolute value

- Write a program to calculate the absolute value of a number by using only two instructions (HINT: Check CMP and RSB)

Absolute value ANS

- Write a program to calculate the absolute value of a number by using only two instructions (HINT: Check CMP and RSB)
- ANS
  - CMP r0, #0   @ Compare with zero
  - RSBLT r0, r0, #0 @ If less than zero, r0 = 0 - r0

Review

Barrel Shifter: Explanations

- LSL (logical shift left): shifts left, fills zeros on the right; C gets the last bit shifted out of bit 31. This is multiply by 2^n for non-overflowing values.
- LSR (logical shift right): shifts right, fills zeros on the left; C gets the last bit shifted out of bit 0. This is unsigned division by 2^n.
- ASR (arithmetic shift right): shifts right, fills the sign bit on the left to preserving the sign; C gets the last bit shifted out of bit 0. This is signed division by 2^n with sign extension
- ROR (rotate right): rotates bits right with wraparound; bits leaving bit 0 re-enter at bit 31, and C receives the bit that wrapped. This is a pure rotation without data loss.
- RRX (rotate right extended): rotates right by one through the carry flag, treating C as a 33rd bit; new bit 31 comes from old C, and C receives old bit 0.

Arithmetic with Shifts

- Assuimg 32-bit registers:
- Q1:
  - LDR r0, =0x00000007
  - MOV r0, r0, LSL 7
- Q2:
  - LDR r0, =0x00000400
  - MOV r0, r0, LSR 2
- Q3:
  - LDR r0, =0xFFFFC000
  - MOV r0, r0, LSR 2
- Q4:
  - LDR r0, =0xFFFFC000
  - MOV r0, r0, ASR 2
- Q5:
  - LDR r0, =0x00000007
  - MOV r0, r0, ROR 2

Q1 ANS

- Q1:
  - LDR r0, =0x00000007
  - MOV r0, r0, LSL 7
- ANS:
  - Original r0 = 0000 0000 0000 0000 0000 0000 0000 0111
  - After LSL 7, r0 = 0000 0000 0000 0000 0000 0011 1000 0000 = 0x0380 (896 in decimal)
  - In decimal: 7 × 2^7 = 7 × 128 = 896 (Not required for exam)

Q2 ANS

- Q2:
  - LDR r0, =0x00000400
  - MOV r0, r0, LSR 2
- ANS:
  - Original r0 = 0000 0000 0000 0000 0000 0100 0000 0000
  - After LSR 2, r0 = 0000 0000 0000 0000 0000 0001 0000 0000 = 0x00000100 (256 in decimal)
  - In decimal: 1024 ÷ 2^2 = 256 (Not required for exam)

Q3 ANS

- Q3:
  - LDR r0, =0xFFFFC000
  - MOV r0, r0, LSR 2
- ANS:
  - Original r0 = 1111 1111 1111 1111 1111 1100 0000 0000
  - After LSR 2, r0 = 0011 1111 1111 1111 1111 1111 0000 0000 = 0x3FFFF000
  - In decimal: 4,294,951,424 ÷ 4 = 1,073,737,728 (Not required for exam)

Q4 ANS

- Q3:
  - LDR r0, =0xFFFFC000
  - MOV r0, r0, ASR 2
- ANS:
  - Original r0 = 1111 1111 1111 1111 1111 1100 0000 0000
  - After ASR 2, r0 = 1111 1111 1111 1111 1111 1111 0000 0000 = 0xFFFFFF00 (-256 in decimal)
  - In decimal: -16384 ÷ 2^2 = -4096 (Not required for exam)

Q5 ANS

- Q4:
  - LDR r0, =0x00000007
  - MOV r0, r0, ROR 2
- ANS:
  - Original r0 = 0000 0000 0000 0000 0000 0000 0000 0111
  - After ROR r0 = 1100 0000 0000 0000 0000 0000 0000 0001 = 0xC0000001

Assembly Programming

- Write ARMv7 assembly for pseudocode
  - r1 = (r0 >> 4) & 15

Assembly Programming ANS

- Write ARMv7 assembly for pseudocode
  - r1 = (r0 >> 4) & 15
- ANS:
  - MOV r1, r0, LSR #4
  - AND r1, r1, #15

Shift LSL

- Compute register values:
  - LDR R1, =0X11223344
  - MOV R2, R1, LSL #4
  - MOV R3, R1, LSL #8
  - MOV R4, R1, LSL #16
  - MOV R5, R1, LSL #6

Shift LSL ANS

- Compute register values:
  - LDR R1, =0X11223344
  - MOV R2, R1, LSL #4
  - MOV R3, R1, LSL #8
  - MOV R4, R1, LSL #16
- ANS:
  - Pseudo-instruction LDR R1, =0x11223344 loads 32-bit constant 0x11223344 into R1.
  - LSL performs a logical left shift of Rm by the immediate count and writes the result to Rd, zero-filling low bits and discarding overflow in 32-bit registers
  - MOV R2, R1, LSL #4 performs a logical left shift of R1 by 4 bits, inserting zeros; 0x11223344 << 4 = 0x12233440.
  - MOV R3, R1, LSL #8 shifts by 8 bits; 0x11223344 << 8 = 0x22334400.
  - MOV R4, R1, LSL #16 shifts by 16 bits; 0x11223344 << 16 = 0x33440000.
  - MOV R5, R1, LSL #6 shifts by 6 bits; 0x11223344 << 6 = 0x488CD100. (convert to binary and back)

Shift LSR ASR ANS

- Compute register values:
  - LDR R1, =0X11223344
  - MOV R2, R1, LSR #4
  - MOV R3, R1, LSR #8
  - MOV R4, R1, LSR #16
  - MOV R5, R1, ASR #4
  - MOV R6, R1, ASR #8
  - MOV R7, R1, ASR #16
- ANS:
  - Pseudo-instruction LDR R1, =0x11223344 loads 32-bit constant 0x11223344 into R1.
  - LSR #4: R2 = 0x11223344 >> 4 = 0x01122334.
  - LSR #8: R3 = 0x11223344 >> 8 = 0x00112233.
  - LSR #16: R4 = 0x11223344 >> 16 = 0x00001122.
  - ASR #4: R5 = 0x11223344 >> 4 = 0x01122334.
  - ASR #8: R6 = 0x11223344 >> 8 = 0x00112233.
  - ASR #16: R7 = 0x11223344 >> 16 = 0x00001122.
  - ASR (arithmetic shift right) replicates the sign bit; for a positive value like 0x11223344 (bit 31 is 0), ASR behaves exactly like LSR and shifts in zeros.

Shift ASR

- Compute register values:
- LDR R1, =0x81223344
- MOV R2, R1, ASR #4
- MOV R3, R1, ASR #8
- MOV R4, R1, ASR #16

Shift ASR ANS

- Compute register values:
- LDR R1, =0x81223344
- MOV R2, R1, ASR #4
- MOV R3, R1, ASR #8
- MOV R4, R1, ASR #16
- ANS:
  - R2 = R1 ASR #4 = 0xF8122334 (right shift 4; top nibble becomes F due to sign extension).
  - R3 = R1 ASR #8 = 0xFF812233 (right shift 8; top byte becomes FF).
  - R4 = R1 ASR #16 = 0xFFFF8122 (right shift 16; top halfword becomes FFFF)

Multiply without MUL

- Without using MUL instruction, give instructions that multiply a register, r3 by
  - 135
  - 153
  - 255
  - 18
  - 16384

Multiply without MUL ANS

- Without using MUL instruction, give instructions that multiply a register, r3 by:
  - 135
  - 153
  - 255
  - 18
  - 1025
- ANS: Decompose 135 = 128 + 4 + 2 + 1
  - `MOV   r0, r3, LSL #7     @ 128*r3`
  - `ADD   r0, r0, r3, LSL #2 @ +4*r3  -> 132*r3`
  - `ADD   r0, r0, r3, LSL #1 @ +2*r3  -> 134*r3`
  - `ADD   r0, r0, r3         @ +1*r3  -> 135*r3`
- Decompose 153 = (8 + 1) * (16 + 1)

  -
    ```
    ADD r0, r3, r3, LSL#3   @ r0 = r3 + 8*r3 = 9*r3
    ADD r0, r0, r0, LSL#4   @ r0 = 9*r3 + 16*9*r3 = 135*r3
    ```
- Decompose 255 = 256 - 1
  - `RSB r0, r3, r3, LSL#8   @ r0 = 256*r3 - r3 = 255*r3`
- Decompose 1025=2^10+1:
  - `ADD r0, r3, r3, LSL#10   @ r0 = r3 + 1024*r3 = 1025*r3`
- Decompose 18 = (16 + 1) + 1

  -
    ```
    ADD r0, r3, r3, LSL#4   @ r0 = r3 + 16*r3 = 17*r3
    ADD r0, r0, r3          @ r0 = 17*r3 + r3 = 18*r3
    ```

Count number of ones

- Write a program to count the number of ones in a 32-bit register r0.

Count the number of ones ANS

- Write a program to count the number of ones in a 32-bit register r0.
- ANS:

```
@ Count ones in r0, result in r3
MOV r3, #0      @ Initialize counter
count_loop:
    CMP r0, #0
    BEQ count_end
    AND r1, r0, #1   @ Check LSB
    ADD r3, r3, r1   @ Add to counter
    LSR r0, r0, #1   @ Shift right
    B count_loop
count_end:
```

Count the number of zeros

- Based on the program that counts 1's, modify it to count the number of zeros a 32-bit register r0.

Count the number of zeros ANS

- Based on the program that counts 1's, modify it to count the number of zeros a 32-bit register r0.
- ANS: count ones, then subtract from 32

```
@ Count zeros in r0, result in r3
MOV r3, #0      @ Initialize counter
count_loop:
    CMP r0, #0
    BEQ count_end
    AND r1, r0, #1   @ Check LSB
    ADD r3, r3, r1   @ Add to counter
    LSR r0, r0, #1   @ Shift right
    B count_loop
count_end:
RSB r3, r3, #32 @ zeros = 32 – ones
```

- Or Invert all bits of r0, then count 1's

```
@ Count zeros in r0, result in r3
MVN r0, r0
MOV r3, #0      @ Initialize counter
count_loop:
    CMP r0, #0
    BEQ count_end
    AND r1, r0, #1   @ Check LSB
    ADD r3, r3, r1   @ Add to counter
    LSR r0, r0, #1   @ Shift right
    B count_loop
count_end:
```

Note that SUB r3, #32, r3 is not valid ARM syntax,
Since the middle operand cannot be an immediate.
You can load 32 into a register and subtract:

```
MOV r2, #32
SUB r3, r2, r3
```

Compute Polynomial

- Write a program that computes 6x^2 – 9x + 2 and stores the result in register r2. Assume x is stored in register r3.

Compute Polynomial

Option A: Direct Translation

```
// r3 = x, result -> r2
MUL r0, r3, r3      @ r0 = x²
MOV r1, #6
MUL r0, r0, r1      @ r0 = 6x²
MOV r1, #9
MUL r1, r1, r3      @ r1 = 9x
SUB r2, r0, r1      @ r2 = 6x² - 9x
ADD r2, r2, #2      @ r2 = 6x² - 9x + 2
```

Note that MUL r0, r0, #6 is not a valid instruction.

Compute Polynomial

Option B: Minimal multiplies with a single MUL and shift-adds

```
// r3 = x, result -> r2
MUL     r0, r3, r3          // r0 = x*x
ADD     r0, r0, r0, LSL #1 // r0 = 3*x*x
ADD     r0, r0, r0          // r0 = 6*x*x
ADD     r1, r3, r3, LSL #3 // r1 = x + 8x = 9x
SUB     r2, r0, r1          // r2 = 6x^2 - 9x
ADD     r2, r2, #2          // r2 = 6x^2 - 9x + 2
```

Option C: Horner form 6x^2 – 9x + 2 = x(6x – 9) + 2

```
// r3 = x, result -> r2
ADD     r0, r3, r3, LSL #1 // r0 = 3x
ADD     r0, r0, r0          // r0 = 6x
SUB     r0, r0, #9              // r0 = 6x - 9
MUL     r2, r3, r0          // r2 = x*(6x - 9)
ADD     r2, r2, #2          // r2 = 6x^2 - 9x + 2
```

Option D: Use MLA (multiply-accumulate) and Horner form x(6x – 9) + 2

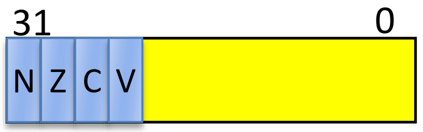
```
// r3 = x, result -> r2
ADD     r0, r3, r3, LSL #1 // r0 = 3x
ADD     r0, r0, r0          // r0 = 6x
SUB     r0, r0, #9              // r0 = 6x - 9
MLA     r2, r3, r0, #2          // r2 = r3*r0 + 2
```

Review

Summary of Carry and Overflow Flags

Bit	Name	Meaning after add or sub
N	negative	result is negative
Z	zero	result is zero
V	overflow	signed overflow
C	carry	unsigned overflow

Carry flag C = 1 upon an **unsigned** addition if the answer is wrong (true result > 2^n-1)
Carry flag C = 0 (Borrow flag = 1) upon an unsigned subtraction if the answer is wrong (true result < 0)
Overflow flag V =1 upon a **signed** addition if the answer is wrong (true result > 2^{n-1}-1 or true result < -2^{n-1})



CPSR (Current Program Status Register)

Review

References

- Lecture 2: Carry flag for unsigned addition and subtraction
  - https://www.youtube.com/watch?v=MxGW2WurKuM&list=PLRJhV4hUhlymmp5CCelFPyxbknsdcXCc8&index=2
- Lecture 3: Overflow flag for signed addition and subtraction
  - https://www.youtube.com/watch?v=BIn6iyYIGio&list=PLRJhV4hUhlymmp5CCelFPyxbknsdcXCc8&index=3

Flags ANDS

- What are value of r2, and NZCV flags after execution, assuming all flags are initially 0.

```
LDR r0, =0xFFFFFF00
LDR r1, =0x00000001
ANDS r2, r1, r0, LSL #1
```

Flags ANDS ANS

- What are value of r2, and NZCV flags after execution, assuming all flags are initially 0.

```
LDR r0, =0xFFFFFF00
LDR r1, =0x00000001
ANDS r2, r1, r0, LSL #1
```

- ANS: r2 = 0x00000000
- NZCV = 0110
- Left shift (LSL #1) moves the bits in r0 left by 1 bit. The bitwise AND is then computed: r2 = r1 & (r0 << 1).
- N (Negative flag): Set to 0 because result r2 = 0x00000000, which is not negative.
- Z (Zero flag): Set to 1 because the result is zero (r2 = 0).
- C (Carry flag): Set to 1. The carry flag is updated based on the last bit shifted out during the left shift operation on r0. Since the left shift of 0xFFFFFF00 by 1 bit causes a '1' to be shifted out, the carry flag is set to 1.
- V (Overflow flag): Unchanged by AND operation.

Flags ADDS

- What are value of r2, and NZCV flags after execution, assuming all flags are initially 0.

```
LDR r0, =0xFFFFFF00
LDR r1, =0x00000001
ADDS r2, r1, r0, LSL #1
```

Flags ADDS ANS

- What are value of r2, and NZCV flags after execution, assuming all flags are initially 0.

```
LDR r0, =0xFFFFFF00
LDR r1, =0x00000001
ADDS r2, r1, r0, LSL #1
```

- ANS: r2 = 0xFFFFFE01
- NZCV = 1000
  - Left shift (LSL #1) moves the bits in r0 left by 1 bit, so r0 = 0xFFFFFE00. The ADDS is then computed: r2 = r1 + (r0 << 1) = 0xFFFFFE01
  - N (Negative): Since result r2 = 0xFFFFFE01 when interpreted as signed 32-bit (two's complement) is negative (most significant bit is 1), N = 1.
  - Z (Zero): Result is not zero, Z = 0.
  - C (Carry): Carry is set if there is an unsigned overflow. Here carry flag C = 0.
  - V (Overflow): Overflow is set if there is a signed overflow. Here:
    - $r1$ is positive; $r0 << 1$ is negative since high bit is set; 2 operands have different signs, no overflow, so V = 0.
    - Recall "Overflow cannot occur when adding 2 operands with different signs or when subtracting 2 operands with the same sign."

Flags

r0	0xffffffff
r1	0x00000001
r2	0x00000003
r3	0xfffffff0

- Suppose registers have the following values:
- What are value of r4, and NZCV flags after execution, assuming all flags are initially 0. (Each instruction runs individually.)
- (a) ADD r4, r0, r2, ASR #3
- (b) ADDS r4, r0, r1
- (c) LSRS r4, r0, #1
- (d) ANDS r4, r0, r3
- (e) CMP r2, #3

r0	0xffffffff
r1	0x00000001
r2	0x00000003
r3	0xfffffff0

Flags ANS

- (a) ADD r4, r0, r2, ASR #3
  - First, r2 ASR #3: 0x00000003 >> 3 = 0x00000000 (arithmetic shift preserves sign)
  - Then: r4 = r0 + 0 = 0xffffffff + 0 = 0xffffffff
  - Result: r4 = 0xffffffff, NZCV = 0000 (ADD doesn't affect flags)
- (b) ADDS r4, r0, r1
  - r4 = 0xffffffff + 0x00000001 = 0x00000000 (truncated to 32-bit)
  - N = 0 (result bit 31 = 0, not negative)
  - Z = 1 (result is zero)
  - C = 1 (carry out from bit 31: 0xffffffff + 1 produces carry)
  - V = 0 (no signed overflow: -1 + 1 = 0, valid in 32-bit signed range)
  - Result: r4 = 0x00000000, NZCV = 0110
- (c) LSRS r4, r0, #1
  - Logical shift right with flag update: 0xffffffff >> 1 = 0x7fffffff
  - N = 0 (result bit 31 = 0)
  - Z = 0 (result is not zero)
  - C = 1 (last bit shifted out was 1)
  - V = 0 (logical shifts don't affect overflow flag)
  - Result: r4 = 0x7fffffff, NZCV = 0010

Flags ANS

r0	0xffffffff
r1	0x00000001
r2	0x00000003
r3	0xfffffff0

- (d) ANDS r4, r0, r3
  - Bitwise AND with flag update: 0xffffffff & 0xfffffff0 = 0xfffffff0
  - N = 1 (result bit 31 = 1, negative)
  - Z = 0 (result is not zero)
  - C = 0 (logical operations clear carry flag)
  - V = 0 (logical operations don't affect overflow)
  - Result: r4 = 0xfffffff0, NZCV = 1000
- (e) CMP r2, #3
  - Compare operation: r2 - 3 = 3 - 3 = 0 (result not stored, only flags updated)
  - N = 0 (subtraction result is 0, bit 31 = 0)
  - Z = 1 (subtraction result is 0)
  - C = 1 (no borrow: 3 ≥ 3, so C = 1 = not borrow)
  - V = 0 (no signed overflow in 3 - 3)
  - Result: NZCV = 0110