Computer Organization and Architecture

Chapter 3: Arithmetic

Exercise and Exam-Oriented Questions & Answers — Set 2

This is the second set of practice questions and complete, exam-ready answers based on Chapter 3 (Arithmetic), covering fresh questions and new worked examples not repeated from Set 1: the addition/subtraction unit and overflow detection, multiplication of positive and negative numbers (including Booth's algorithm), restoring and non-restoring division, and floating-point arithmetic. It is organised into three parts — Short Answer Questions, Long Answer / Descriptive Questions, and Solved Numerical Problems.

Section A: Short Answer Questions

Q1. Differentiate between fixed-point and floating-point number representation.

Fixed-point representation stores numbers with the binary point at an implied, constant position, giving a fixed range and fixed precision, and is simple and fast to compute with. Floating-point representation instead stores a number as a mantissa and an exponent, allowing a much wider range of magnitudes (very large and very small numbers) with variable precision, at the cost of more complex arithmetic hardware.

Q2. What is exponent bias in floating-point representation, and why is it used?

Exponent bias is a fixed constant added to the true (signed) exponent before it is stored, so that the stored exponent value is always a non-negative number. This removes the need for a separate sign bit on the exponent and allows two floating-point exponents to be compared using ordinary unsigned integer comparison.

Q3. What are guard bits, and why are they used in floating-point arithmetic?

Guard bits are extra bits kept temporarily beyond the normal mantissa width during intermediate steps such as exponent alignment. They preserve bits that would otherwise be lost during shifting, which improves the accuracy of the final rounded result.

Q4. Why does the 2's complement method allow subtraction to be performed using an adder?

The 2's complement of an n-bit number B equals (2^n − B). Adding A to the 2's complement of B therefore computes A − B + 2^n. The extra 2^n term appears only as a carry-out beyond the register width; once this carry is discarded, the remaining bits give exactly A − B, so a plain adder circuit can also perform subtraction.

Q5. What is the maximum number of bits required to store the product of two n-bit unsigned numbers?

Up to 2n bits. The largest possible product of two n-bit unsigned numbers, (2^n − 1) × (2^n − 1), is just less than 2^(2n), so 2n bits are needed to represent any possible product without overflow.

Q6. What is the significance of the carry-out flip-flop E in the multiplication hardware algorithm?

E temporarily stores any carry-out produced when the multiplicand BR is added to the accumulator AC. Without E this carry bit would be lost; instead, E takes part in the following shift-right operation, correctly moving that extra bit into the most significant position of AC.

Q7. How does arithmetic overflow in fixed-point addition differ from exponent overflow in floating-point arithmetic?

In fixed-point addition, overflow occurs when the sum of two numbers exceeds the range representable in the available number of bits, and is detected by comparing the carry into and the carry out of the sign bit. In floating-point arithmetic, exponent overflow instead occurs when the computed exponent of the result is too large to fit in the exponent field, meaning the magnitude of the result cannot be represented in that format at all, regardless of the mantissa.

Q8. Why can Booth's algorithm need more operations for an isolated alternating bit pattern (such as 0101) than for one long run of identical bits?

Booth's algorithm performs an add or subtract only at a boundary between a run of 0s and a run of 1s. A long run of the same bit has only one such boundary at each end, needing very few arithmetic operations, whereas an alternating pattern like 0101 creates a boundary at almost every bit position, forcing an add or subtract at nearly every cycle.

Q9. What is the purpose of the repeat/sequence counter in the division hardware algorithm, and to what value is it initialised?

The repeat counter tracks how many shift-and-subtract (or shift-and-add) cycles have been completed. It is initialised to n, the number of bits in the dividend, and the division process stops as soon as the counter reaches zero, ensuring exactly n cycles are executed to produce an n-bit quotient.

Q10. Why is division generally considered more complex to implement in hardware than multiplication?

Multiplication always performs the same fixed sequence of steps (test a bit, optionally add, then shift) regardless of the operand values. Division requires a data-dependent decision at every step — the sign of the partial remainder must be checked to decide whether to add or subtract and what quotient bit to set — which adds control complexity, along with the need to handle special cases such as division by zero.

Section B: Long Answer / Descriptive Questions

These questions typically carry higher marks in examinations and expect a block diagram or flowchart along with the explanation.

Q1. Explain the design of an n-bit binary adder/subtractor circuit and explain how overflow is detected for signed number addition, with a suitable diagram.

A combined adder/subtractor circuit uses one n-bit parallel adder together with an XOR gate array that conditionally complements the second operand B, controlled by a mode signal M (M = 0 for addition, M = 1 for subtraction, with M also applied as the carry-in). This lets one adder circuit perform both addition and subtraction.

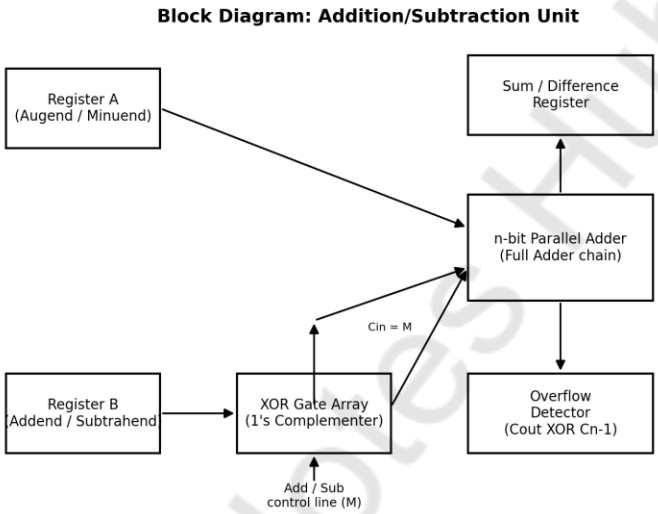

Fig B1: Block diagram of a combined Addition/Subtraction unit

Overflow in signed (2's complement) addition is detected by comparing the carry generated into the most significant bit (Cn−1, the carry into the sign bit) with the carry generated out of the most significant bit (Cn, the final carry-out). If these two carries are different, an overflow has occurred; if they are the same, the result is valid. This is implemented simply by XOR-ing Cn−1 and Cn.

Illustrative Example (4-bit registers, range −8 to +7):

Add 5 (0101) and 4 (0100):

- Bit 0: 1 + 0 = 1, carry 0
- Bit 1: 0 + 0 = 0, carry 0
- Bit 2: 1 + 1 = 0, carry 1 (this carry, C2, is the carry INTO the sign bit)
- Bit 3 (sign bit): 0 + 0 + carry-in(1) = 1, carry-out C3 = 0

Result = 1001, which as a 4-bit 2's complement number represents −7 — clearly wrong, since 5 + 4 should equal 9. Checking the carries: C2 (carry into sign bit) = 1, C3 (carry out of sign bit) = 0. Since C2 ≠ C3, the overflow detector correctly flags an overflow, because +9 cannot be represented in the 4-bit signed range (−8 to +7).

Q2. Show, step-by-step, how binary multiplication of two n-bit unsigned numbers is performed using the shift-and-add algorithm, and derive the number of bits required to hold the product.

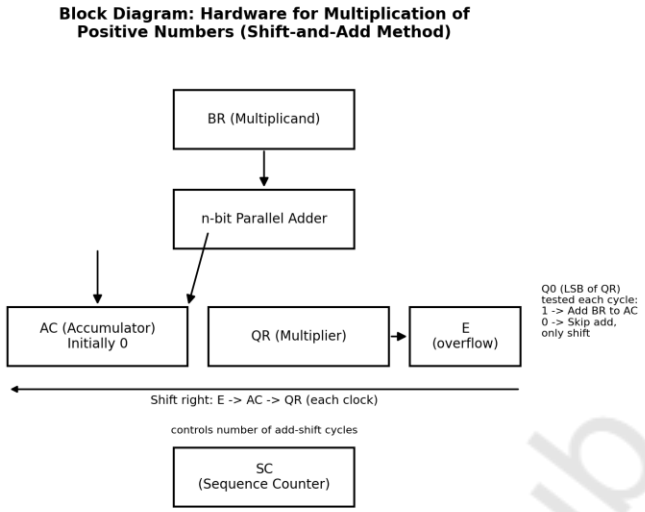

Fig B2: Hardware block diagram for binary multiplication (positive numbers)

1. Initialise AC = 0, E = 0; load BR = multiplicand, QR = multiplier; set SC = n.
2. Test Q0, the least significant bit of QR.
3. If Q0 = 1, add BR to AC, storing any carry-out in E; if Q0 = 0, take no action.
4. Shift E, AC and QR together one bit to the right, then reset E to 0.
5. Decrement SC; repeat from step 2 until SC = 0.

Bit-count derivation: an n-bit unsigned number can represent values from 0 to (2^n − 1). The largest possible product of two such numbers is therefore (2^n − 1) × (2^n − 1) = 2^(2n) − 2^(n+1) + 1, which is strictly less than 2^(2n) but can be as large as (2^(2n) − 2·2^n + 1). Since this maximum value requires 2n bits to represent (it is less than 2^(2n) but can exceed 2^(2n−1)), the hardware reserves a full 2n-bit result — n bits in AC and n bits in QR — to guarantee that no product ever overflows the result registers.

Q3. Multiply 10 × 6 using the shift-and-add algorithm, showing all register values (4-bit registers).

Let BR = 1010 (10), QR = 0110 (6), AC = 0000, SC = 4.

Cycle	Q0	Operation	AC	QR	SC
Initial	0	—	0000	0110	4
1	0	No add, only shift	0000	0011	3
2	1	AC = AC + BR, then shift right	0101	0001	2
3	1	AC = AC + BR, then shift right	0111	1000	1
4	0	No add, only shift	0011	1100	0

Table B1: Step-by-step trace of shift-and-add multiplication (10 × 6)

Final product = AC:QR = 0011 1100 = 60 in decimal, which correctly equals 10 × 6.

Q4. Explain Booth's algorithm and discuss its best-case and worst-case performance in terms of the number of add/subtract operations, with suitable illustrative bit patterns.

Booth's algorithm examines the multiplier two bits at a time (Qn and Qn+1) and performs an add or subtract of the multiplicand only at a transition point — that is, at the boundary between a run of 0s and a run of 1s in the

multiplier. Within an unbroken run of the same bit value, no arithmetic operation is performed; only a shift takes place.

Best case — one long run of 1s (or 0s):

Consider a multiplier such as 0 0111110 (a single run of six 1s surrounded by 0s). Booth's algorithm needs only one subtract at the start of the run (10 pattern) and one add at the end of the run (01 pattern) — just 2 arithmetic operations for the entire multiplier, regardless of how long the run is. All the remaining cycles perform only a shift.

Worst case — alternating bit pattern:

Consider a multiplier such as 0101 0101. Every single bit position forms a transition (a boundary between 0 and 1, or 1 and 0), so the bit-pair check produces 10 or 01 at almost every cycle, forcing an add or subtract operation at nearly every one of the n cycles. In this case Booth's algorithm performs about as many arithmetic operations as the plain shift-and-add method, losing its speed advantage.

In general, the number of arithmetic operations Booth's algorithm performs equals the number of 0-to-1 and 1-to-0 transitions in the multiplier (plus the initial boundary with the implied Qn+1 = 0), so multipliers with long uniform runs are multiplied fastest, while multipliers with rapidly alternating bits gain little or no benefit.

Q5. Multiply (+5) × (−6) using Booth's algorithm, showing all steps (4-bit registers).

Let BR = 0101 (+5, multiplicand) and QR = 1010 (−6, multiplier, in 4-bit 2's complement). Then −BR (2's complement of BR) = 1011. AC = 0000, Qn+1 = 0, SC = 4.

Cycle	Qn Qn+1	Operation	AC	QR	Qn+1
Initial	0 0	—	0000	1010	0
1	0 0	No operation, only ASR	0000	0101	0
2	1 0	AC = AC − BR, then ASR	1101	1010	1
3	0 1	AC = AC + BR, then ASR	0001	0101	0
4	1 0	AC = AC − BR, then ASR	1110	0010	1

Table B2: Step-by-step trace of Booth's algorithm for (+5) × (−6)

Final product = AC:QR = 1110 0010, the 8-bit 2's complement representation of −30, which correctly equals (+5) × (−6) = −30.

Q6. Explain, with the help of a flowchart, how restoring division determines the sign of the partial remainder and decides whether to restore.

At every cycle of restoring division, the combined register (A, Q) is first shifted left by one bit, and then the divisor M is subtracted from A: A = A − M. The most significant bit of A after this subtraction indicates its sign — if it is 1, A is negative; if it is 0, A is non-negative (since A is treated as a signed register during this comparison).

- If A is negative after the subtraction, the trial subtraction is undone by adding the divisor back — A = A + M (restoring A to what it was before the subtraction) — and the newly generated quotient bit Q0 is set to 0.
- If A is non-negative, no restoration is required — the subtracted value is kept as the new A, and Q0 is set to 1.

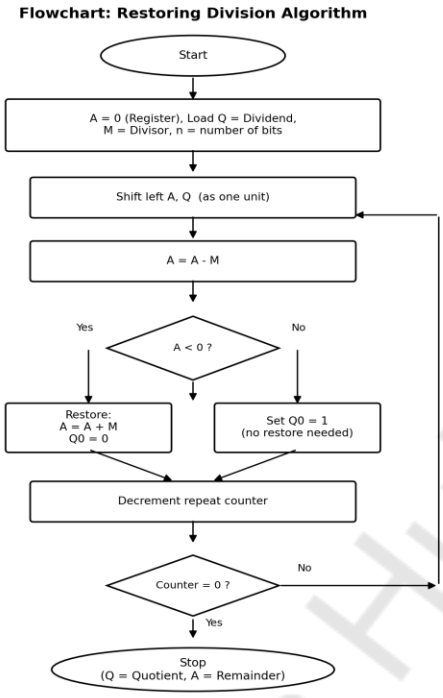

Fig B3: Flowchart of the Restoring Division Algorithm

This decision (restore or keep) is exactly the branch shown in the flowchart at the 'A < 0 ?' decision box, and it is repeated once for every bit of the dividend, guaranteeing that A always holds the correct, valid partial remainder before the next shift.

Q7. Divide 14 by 3 using restoring division, showing the complete step-by-step trace (4-bit registers).

Let dividend Q = 1110 (14), divisor M = 0011 (3), A = 0000, SC runs for 4 cycles.

Cycle	A after shift	A = A − M	Decision	Final A	Q
1	0001	1 − 3 = −2	Negative → restore, Q0 = 0	0001	1100
2	0011	3 − 3 = 0	Non-negative → Q0 = 1	0000	1001
3	0001	1 − 3 = −2	Negative → restore, Q0 = 0	0001	0010
4	0010	2 − 3 = −1	Negative → restore, Q0 = 0	0010	0100

Table B3: Step-by-step trace of restoring division (14 ÷ 3)

Final result: Q = 0100 (quotient = 4), A = 0010 (remainder = 2), matching 14 ÷ 3 = 4 remainder 2.

Q8. Divide 14 by 3 using non-restoring division and verify the result matches Q7.

Using the same dividend Q = 1110 (14), divisor M = 0011 (3), A = 0000:

Cycle	Prev. sign of A	A after shift	Operation	New A	Q0	Q
1	A ≥ 0	0001	A = A − M = 1 − 3	1110 (−2)	0	1100
2	A < 0	1101 (−3)	A = A + M = −3 + 3	0000 (0)	1	1001
3	A ≥ 0	0001 (1)	A = A − M = 1 − 3	1110 (−2)	0	0010
4	A < 0	1100 (−4)	A = A + M = −4 + 3	1111 (−1)	0	0100

Table B4: Step-by-step trace of non-restoring division (14 ÷ 3)

The final remainder A = 1111 (−1) is negative, so a correction is applied at the end: A = A + M = −1 + 3 = 2 (0010). Final result: Q = 0100 (quotient = 4), A = 0010 (remainder = 2) — exactly matching the restoring-division result in Q7.

Q9. Explain the concept of normalization and rounding in floating-point arithmetic, and why both are necessary.

Normalization is the process of adjusting a floating-point mantissa (by shifting it left or right and correspondingly changing the exponent) so that its most significant bit is non-zero — typically exactly 1 for binary normalized numbers. A normalized mantissa always uses the full width of the mantissa field to represent the value, giving the maximum possible precision for a given number of bits; an un-normalized mantissa (with leading zeros) wastes some of its bits and needlessly loses precision.

Rounding is necessary because, during operations such as exponent alignment (shifting a mantissa right to match a larger exponent) or after multiplication, extra bits are generated beyond the width of the mantissa register. These extra bits cannot all be stored, so the result must be rounded off to fit back into the available mantissa width — typically by truncating (simply discarding the extra bits) or by round-to-nearest (adjusting the last retained bit based on the discarded bits), the latter giving smaller average error. Together, normalization ensures the stored representation is as precise as possible, while rounding controls how the unavoidable loss of extra bits is handled, keeping the accumulated numerical error as small as possible over repeated calculations.

Q10. State and explain, with an example, the special cases (zero result, exponent overflow, exponent underflow) that can occur during floating-point addition/subtraction.

(a) Zero result:

This occurs when the mantissas of the two operands are equal in magnitude but opposite in sign (as in subtraction of two equal numbers), so that Mr becomes exactly 0. Since a mantissa of 0 has no meaningful 'leading bit' to normalize, the exponent is forcibly set to 0 as well, giving a clean representation of true zero rather than an arbitrary or undefined exponent value.

(b) Exponent overflow:

This occurs when the computed exponent of the result is larger than the maximum value the exponent field can hold. For example, if a floating-point format's exponent field can represent values only up to +127 (as in IEEE-754 single precision), and normalizing/adding two very large numbers requires an exponent of +130, the result cannot be represented in this format at all — an overflow exception must be reported, since the magnitude of the true result is simply too large.

(c) Exponent underflow:

This occurs when the computed exponent is smaller (more negative) than the minimum value the exponent field can represent. This typically happens when subtracting two nearly equal numbers, producing a very small result mantissa that must be shifted left many times to normalize, driving the exponent below the representable minimum (for example, below −126 in IEEE-754 single precision). Since the result is too small in magnitude to represent in normalized form, an underflow exception is reported (or the result may be flushed to zero, depending on the system).

Section C: Solved Numerical Problems

A further set of fresh practice problems (with numbers different from both Section B and the previous Set 1 booklet) is solved below.

Q1. Perform 15 − 7 and 7 − 15 using 2's complement addition (use 5-bit registers).

Let A = 15 = 01111 and B = 7 = 00111 (5-bit representation, range −16 to +15).

(a) 15 − 7:

- 2's complement of B (7): invert 00111 → 11000, add 1 → 11001.
- A + (2's complement of B) = 01111 + 11001 = 1 01000.
- Discard the carry-out: result = 01000 = 8, which is correct since 15 − 7 = 8.

(b) 7 − 15:

- 2's complement of B (15): invert 01111 → 10000, add 1 → 10001.
- A + (2's complement of B) = 00111 + 10001 = 11000, with no carry-out.
- 11000 is negative (MSB = 1); taking its 2's complement: invert 11000 → 00111, add 1 → 01000 = 8. So the result represents −8, which is correct since 7 − 15 = −8.

Q2. Multiply 7 × 9 (positive numbers) using the shift-and-add method (4-bit registers).

Let BR = 0111 (7), QR = 1001 (9), AC = 0000, SC = 4.

Cycle	Q0	Operation	AC	QR	SC
Initial	1	—	0000	1001	4
1	1	AC = AC + BR, then shift right	0011	1100	3
2	0	No add, only shift	0001	1110	2
3	0	No add, only shift	0000	1111	1
4	1	AC = AC + BR, then shift right	0011	1111	0

Table C1: Step-by-step trace of shift-and-add multiplication (7 × 9)

Final product = AC:QR = 0011 1111 = 63 in decimal, which correctly equals 7 × 9.

Q3. Multiply (−9) × (−4) using Booth's algorithm (use 5-bit registers, since −9 needs 5 bits in 2's complement).

Let BR = 10111 (−9, multiplicand) and QR = 11100 (−4, multiplier), both in 5-bit 2's complement. Then −BR (2's complement of BR) = 01001. AC = 00000, Qn+1 = 0, SC = 5.

Cycle	Qn Qn+1	Operation	AC	QR	Qn+1
Initial	0 0	—	00000	11100	0
1	0 0	No operation, only ASR	00000	01110	0
2	0 0	No operation, only ASR	00000	00111	0
3	1 0	AC = AC − BR, then ASR	00100	10011	1
4	1 1	No operation, only ASR	00010	01001	1
5	1 1	No operation, only ASR	00001	00100	1

Table C2: Step-by-step trace of Booth's algorithm for (−9) × (−4)

Final product = AC:QR = 00001 00100, i.e. 0000100100 in 10 bits, which equals 36 in decimal — correctly matching (−9) × (−4) = 36.

Q4. Divide 9 by 2 using the Restoring Division algorithm (4-bit registers).

Let dividend Q = 1001 (9), divisor M = 0010 (2), A = 0000, SC runs for 4 cycles.

Cycle	A after shift	A = A − M	Decision	Final A	Q
1	0001	1 − 2 = −1	Negative → restore, Q0 = 0	0001	0010
2	0010	2 − 2 = 0	Non-negative → Q0 = 1	0000	0101
3	0000	0 − 2 = −2	Negative → restore, Q0 = 0	0000	1010
4	0001	1 − 2 = −1	Negative → restore, Q0 = 0	0001	0100

Table C3: Step-by-step trace of restoring division (9 ÷ 2)

Final result: Q = 0100 (quotient = 4), A = 0001 (remainder = 1), matching 9 ÷ 2 = 4 remainder 1.

Q5. Divide 9 by 2 using the Non-Restoring Division algorithm (4-bit registers) and verify the result matches Q4.

Using the same dividend Q = 1001 (9), divisor M = 0010 (2), A = 0000:

Cycle	Prev. sign of A	A after shift	Operation	New A	Q0	Q
1	A ≥ 0	0001	A = A − M = 1 − 2	1111 (−1)	0	0010
2	A < 0	1110 (−2)	A = A + M = −2 + 2	0000 (0)	1	0101
3	A ≥ 0	0000 (0)	A = A − M = 0 − 2	1110 (−2)	0	1010
4	A < 0	1101 (−3)	A = A + M = −3 + 2	1111 (−1)	0	0100

Table C4: Step-by-step trace of non-restoring division (9 ÷ 2)

The final remainder A = 1111 (−1) is negative, so a correction is applied: A = A + M = −1 + 2 = 1 (0001). Final result: Q = 0100 (quotient = 4), A = 0001 (remainder = 1) — exactly matching the restoring-division result in Q4.

Q6. Subtract the two floating-point numbers: X = 0.1101 × 2^3 minus Y = 0.1100 × 2^1 (binary mantissas).

Step 1 — Compare exponents: Ex = 3, Ey = 1. Since they are unequal, align them by shifting the mantissa of Y (the smaller exponent) right by (3 − 1) = 2 bits, and raising Ey to 3.

- Y's mantissa 0.1100 shifted right by 2 bits → 0.0011, with exponent now 3.

Step 2 — Subtract the mantissas, now that both share exponent 3:

- Mx = 0.1101, My (aligned) = 0.0011
- Mr = 0.1101 − 0.0011 = 0.1010

Step 3 — Check normalization: the result mantissa 0.1010 already has a 1 as its most significant bit, so it is already normalized, and no overflow has occurred, so no further shifting is required.

Final result: Difference = 0.1010 × 2^3. Verification in decimal: X = 0.8125 × 8 = 6.5, Y = 0.75 × 2 = 1.5, so X − Y should equal 5.0. Checking the result: 0.1010 binary = 0.625, and 0.625 × 8 = 5.0. The result is correct.