Computer Organization and Architecture

Chapter 3: Arithmetic

Exercise and Exam-Oriented Questions & Answers

This booklet contains practice questions and complete, exam-ready answers based on Chapter 3 (Arithmetic): the addition/subtraction unit, multiplication of positive and negative numbers (including Booth's algorithm), restoring and non-restoring division, and floating-point addition/subtraction. It is organised into three parts — Short Answer Questions, Long Answer / Descriptive Questions, and Solved Numerical Problems — to help with both quick revision and full exam preparation.

Section A: Short Answer Questions

Q1. Why is subtraction converted into addition inside digital arithmetic circuits?

Because building two separate circuits — one for addition and one for subtraction — would waste hardware. By taking the 2's complement of the number being subtracted and then adding it, subtraction (A − B) becomes A + (2's complement of B), so a single adder circuit, controlled by a mode signal, can perform both operations.

Q2. What is the function of the overflow detector in an adder circuit?

It checks whether the result of an addition/subtraction has exceeded the range that can be represented in the given number of bits. This is done by XOR-ing the carry into the most significant bit (MSB) with the carry out of the MSB; if the two differ, an overflow has occurred.

Q3. What is the purpose of the Sequence Counter (SC) in the multiplication and division hardware algorithms?

SC keeps track of how many add/subtract-and-shift cycles have been completed. It is initialised to n (the number of bits), decremented by 1 after every cycle, and the process stops as soon as SC becomes 0, ensuring the algorithm runs for exactly n cycles.

Q4. Define Booth's algorithm in brief.

Booth's algorithm is a technique for multiplying two signed binary numbers represented in 2's complement form. It examines two adjacent bits of the multiplier (Qn and Qn+1) at each step and decides whether to add the multiplicand, subtract the multiplicand, or simply shift, allowing correct signed multiplication without first converting operands to positive numbers.

Q5. What do the bits Qn and Qn+1 represent in Booth's algorithm?

Qn is the current least significant bit of the multiplier register (QR), and Qn+1 is an extra flip-flop that stores the value of Qn from the previous cycle (initially set to 0). Comparing Qn with Qn+1 tells the algorithm whether it is at the start, middle, or end of a run of 1s in the multiplier.

Q6. What is an Arithmetic Shift Right (ASR), and why does Booth's algorithm use it instead of a normal (logical) shift?

An arithmetic shift right shifts all bits one place to the right while keeping (replicating) the sign bit (MSB) unchanged, so that the sign of the number is preserved. Booth's algorithm uses ASR instead of a logical shift because the partial results in AC can be negative (in 2's complement form), and a logical shift would incorrectly destroy the sign.

Q7. What is the key difference between restoring and non-restoring division?

In restoring division, whenever the trial subtraction makes the remainder negative, the divisor is immediately added back (restored) before continuing. In non-restoring division, no immediate restoration is done — instead, the next operation (add or subtract) is chosen based on the sign of the previous remainder, and only a single correction step is applied at the very end, if needed.

Q8. Why must the exponents of two floating-point numbers be made equal before their mantissas can be added?

Floating-point numbers represent a value as mantissa × 2^exponent. Mantissas can only be added or subtracted directly when they carry the same 'weight', i.e., the same power of two. If the exponents differ, the mantissa of the number with the smaller exponent must first be shifted right (and its exponent increased) until both exponents match.

Q9. What is meant by normalizing a floating-point mantissa?

Normalizing means adjusting the mantissa (by shifting it left or right and correspondingly changing the exponent) so that its most significant bit is non-zero (typically 1), i.e., there is no leading zero. Normalized numbers use the available mantissa bits most efficiently and give maximum precision.

Q10. In non-restoring division, what correction is applied if the final remainder turns out to be negative?

If, after all the division cycles are complete, the value in the remainder register A is still negative, the divisor M is added to it exactly once more (A = A + M) to obtain the correct, true remainder.

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 block diagram of a combined addition/subtraction unit and describe how a single adder circuit is used to perform both addition and subtraction.

A digital computer uses one common circuit for both addition and subtraction instead of two separate circuits, by exploiting the fact that A − B can be computed as A + (2's complement of B). A mode signal M controls the operation: each bit of operand B is passed through an XOR gate together with M before reaching the adder.

- When M = 0: B XOR 0 = B and Cin = 0, so the adder simply computes A + B (addition).
- When M = 1: B XOR 1 = B′ (1's complement of B) and Cin = 1, so the adder computes A + B′ + 1, which equals A + (2's complement of B) = A − B (subtraction).

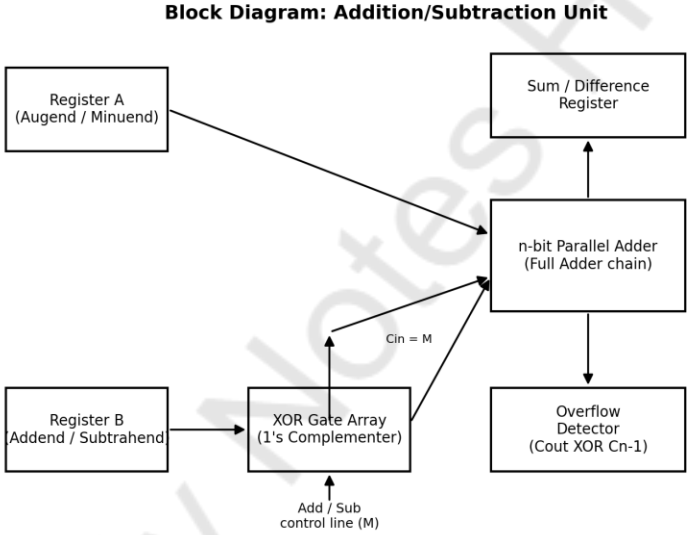

Fig B1: Block diagram of a combined Addition/Subtraction unit

The main components are: Register A and Register B (hold the operands), the XOR gate array (acts as a controlled complementer), the n-bit parallel adder (a chain of full adders), the Sum/Difference register (stores the result), and the overflow detector, which XORs the carry into the MSB with the carry out of the MSB to flag an overflow. This design saves hardware because only one adder circuit and one extra XOR gate array are needed to support both arithmetic operations.

Q2. Describe, with a neat block diagram, the hardware algorithm used for multiplying two unsigned (positive) binary numbers.

Binary multiplication of positive numbers is performed in hardware using the shift-and-add method. The multiplicand is held in register BR, the multiplier in register QR, and the accumulator AC (initially 0) builds up the partial sums. An extra flip-flop E stores any carry-out, and a sequence counter SC keeps track of the number of cycles completed.

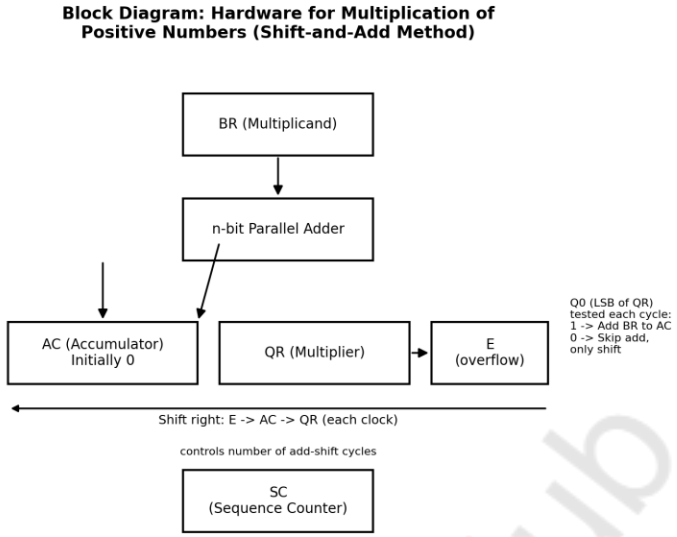

Fig B2: Hardware block diagram for binary multiplication (positive numbers)

The algorithm proceeds as follows:

1. Initialise AC = 0, E = 0; load BR = multiplicand and QR = multiplier; set SC = n (number of bits).
2. Examine Q0, the least significant bit of QR.
3. If Q0 = 1, add BR to AC and store any carry-out in E; if Q0 = 0, do nothing.
4. Shift E, AC and QR together one bit to the right, then reset E to 0.
5. Decrement SC by 1, and repeat from step 2 until SC = 0.

At the end of n cycles, the 2n-bit product is available jointly in AC (upper half) and QR (lower half). Section C of this booklet contains a fully worked numerical example of this algorithm.

Q3. Using the shift-and-add method, multiply two positive binary numbers, showing the accumulator, multiplier register and sequence counter at every step.

Let the multiplicand BR = 1011 (11 in decimal) and the multiplier QR = 1101 (13 in decimal), using 4-bit registers, so SC starts at 4. AC and E are initially 0.

Cycle	Q0	Operation	AC	QR	SC
Initial	1	—	0000	1101	4
1	1	AC = AC + BR, then shift right	0101	1110	3
2	0	No add, only shift	0010	1111	2
3	1	AC = AC + BR, then shift right	0110	1111	1
4	1	AC = AC + BR (carry to E), then shift right	1000	1111	0

Table B1: Step-by-step trace of shift-and-add multiplication (11 × 13)

At the end, AC:QR = 1000 1111, i.e., 143 in decimal, which correctly equals 11 × 13.

Q4. What is Booth's algorithm? Explain why it is needed for multiplying signed numbers represented in 2's complement form.

The plain shift-and-add method of Section B2 only works correctly for unsigned (positive) numbers; it cannot handle negative operands represented in 2's complement form, because blindly adding a 2's complement negative

number using unsigned rules gives a wrong result. Booth's algorithm solves this by working directly with signed 2's complement numbers, without first converting them to a positive form and fixing the sign afterwards.

Booth's algorithm examines a pair of bits of the multiplier at every step: the current bit Qn and the previous bit Qn+1 (initially 0). Depending on whether this pair is 00, 01, 10, or 11, the algorithm decides whether to add the multiplicand, subtract the multiplicand, or simply shift with no arithmetic operation, followed always by an arithmetic shift right that preserves the sign bit. This bit-pair technique correctly accounts for the sign, so both operands may be positive or negative in 2's complement form. As a bonus, Booth's algorithm can also be faster than plain shift-and-add when the multiplier has long runs of consecutive 1s or 0s, since such runs need only a single add/subtract at their boundary instead of one operation per bit.

Q5. Draw and explain, step by step, the flowchart of Booth's multiplication algorithm.

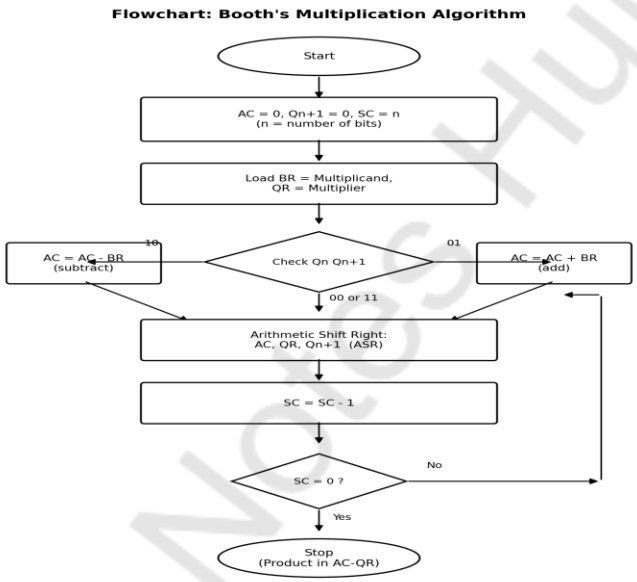

Fig B3: Flowchart of Booth's Multiplication Algorithm

The flowchart works as follows:

- Initialise AC = 0, Qn+1 = 0, SC = n, and load BR = multiplicand, QR = multiplier.
- Check the bit pair Qn Qn+1: if it is 10, subtract BR from AC; if it is 01, add BR to AC; if it is 00 or 11, perform no arithmetic operation.
- Regardless of which case applied, perform an arithmetic shift right (ASR) on the combined AC, QR and Qn+1, preserving the sign bit of AC.
- Decrement SC by 1.
- If SC $\neq$ 0, repeat from the bit-pair check; otherwise stop. The final signed product is held in AC (upper half) and QR (lower half).

Q6. Illustrate Booth's algorithm with a suitable numerical example involving the multiplication of a positive number by a negative number.

Let multiplicand BR = 1001 (representing −7 in 4-bit 2's complement) and multiplier QR = 0011 (representing 3). Then −BR (2's complement of BR) = 0111. Initialise AC = 0000, Qn+1 = 0, SC = 4.

Cycle	Qn Qn+1	Operation	AC	QR	Qn+1
Initial	1 0	—	0000	0011	0
1	1 0	AC = AC − BR, then ASR	0011	1001	1
2	1 1	No operation, only ASR	0001	1100	1
3	0 1	AC = AC + BR, then ASR	1101	0110	0
4	0 0	No operation, only ASR	1110	1011	0

Table B2: Step-by-step trace of Booth's algorithm for (−7) × (3)

The final product AC:QR = 1110 1011 is the 8-bit 2's complement representation of −21, which is correct since (−7) × 3 = −21.

Q7. Explain the restoring division algorithm in detail, along with its flowchart and a suitable numerical example.

In restoring division, the divisor is repeatedly subtracted from a running partial remainder after each left shift. If a subtraction makes the remainder negative, the divisor is added back (restored) immediately, and the quotient bit is set to 0; if the remainder stays non-negative, no restoration is needed and the quotient bit is set to 1. This way, register A always holds the correct restored remainder before the next shift.

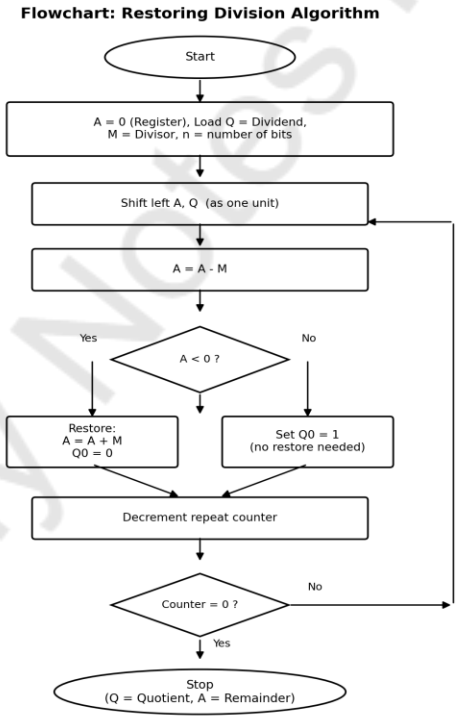

Fig B4: Flowchart of the Restoring Division Algorithm

Worked example: divide 11 by 3, with dividend Q = 1011, divisor M = 0011, A = 0000, and 4-bit registers (4 cycles).

Cycle	A after shift	A = A − M	Decision	Final A	Q
1	0001	1 − 3 = −2	Negative → restore, Q0 = 0	0001	0110
2	0010	2 − 3 = −1	Negative → restore, Q0 = 0	0010	1100
3	0101	5 − 3 = 2	Non-negative → Q0 = 1	0010	1001

Cycle	A after shift	A = A − M	Decision	Final A	Q
4	0101	5 − 3 = 2	Non-negative → Q0 = 1	0010	0011

Table B3: Step-by-step trace of restoring division (11 ÷ 3)

At the end, Q = 0011 (quotient = 3) and A = 0010 (remainder = 2), matching 11 ÷ 3 = 3 remainder 2.

Q8. Explain the non-restoring division algorithm and compare it with the restoring division algorithm in terms of the number of operations required.

Non-restoring division avoids the wasted restoring step of the restoring method. At each cycle, the operation performed depends on the sign of the previous remainder: if it was non-negative, the divisor is subtracted; if it was negative, the divisor is added instead. The quotient bit is then set based on the sign of this new remainder. Because there is no immediate restoring addition inside the loop, at most one single correction step (adding the divisor once) is needed at the very end, only if the final remainder is negative.

Aspect	Restoring Division	Non-Restoring Division
Correction step	Adds back divisor immediately whenever remainder is negative	No immediate correction; alternates add/subtract; at most one correction at the very end
Operations per cycle	Up to two (subtract, then possibly restore)	Exactly one (either add or subtract)
Speed	Slower, due to possible extra restoring addition	Faster, since only one arithmetic operation per cycle
Control complexity	Simpler control logic	Slightly more complex (must remember previous sign)

In short, non-restoring division performs the same number of shift operations as restoring division but roughly halves the number of arithmetic (add/subtract) operations in the worst case, making it the faster of the two in hardware implementations.

Q9. Explain the steps involved in floating-point addition/subtraction with the help of a neat flowchart.

Floating-point numbers are represented as sign, exponent and mantissa. Because two numbers may have different exponents, they must first be brought to a common exponent before their mantissas can be combined. The overall procedure is:

1. Read the two operands A (Ea, Ma) and B (Eb, Mb).
2. If Ea ≠ Eb, align the exponents by shifting the mantissa of the number with the smaller exponent to the right and increasing that exponent to match the other.
3. Add or subtract the mantissas (Mr = Ma ± Mb) using the now-common exponent E.
4. If the result mantissa overflows, shift it right by one bit and increment E by one.
5. If the result mantissa needs normalizing (has a leading zero), shift it left and decrement E, repeating until normalized.
6. If the result mantissa is zero, force the exponent to zero as well (true zero).
7. Check for exponent overflow or underflow and report an exception if either occurs.
8. Otherwise, round the mantissa to the allowed width and output the result as (E, Mr).

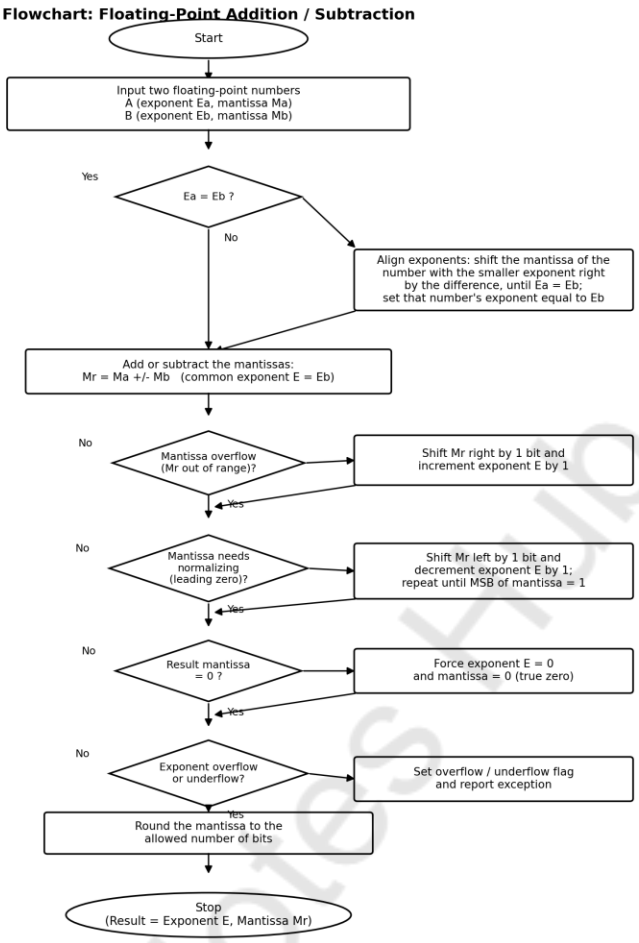

Fig B5: Flowchart of Floating-Point Addition / Subtraction

A fully worked numerical example of this procedure is given in Section C of this booklet.

Q10. Compare restoring and non-restoring division algorithms on the basis of speed, hardware/control complexity, and correction steps.

Basis of Comparison	Restoring Division	Non-Restoring Division
Correction/restoring step	Performed inside the loop whenever remainder < 0	Not performed inside the loop; at most one correction at the end
Number of arithmetic operations	Can be up to two per cycle (subtract + restore)	Exactly one per cycle (add or subtract)
Overall speed	Comparatively slower	Comparatively faster
Control logic	Simple — same subtract-then-check logic every cycle	Slightly more complex — must remember the sign of the previous remainder
Hardware requirement	Standard adder/subtractor and comparator	Same adder/subtractor; additional logic to choose add or subtract

Overall, non-restoring division is generally preferred in practical hardware implementations because it achieves the same result with fewer arithmetic operations, at the cost of a marginally more complex control unit.

Section C: Solved Numerical Problems

Additional practice problems (with different numbers from Section B) are solved below for extra exam preparation.

Q1. Perform 12 − 9 and 9 − 12 using 2's complement addition (use 5-bit registers).

Let A = 12 = 01100 and B = 9 = 01001 (5-bit representation, range −16 to +15).

(a) 12 − 9:

- 2's complement of B (9): invert 01001 → 10110, add 1 → 10111.
- A + (2's complement of B) = 01100 + 10111 = 1 00011.
- Discard the carry-out (bit beyond 5 bits): result = 00011 = 3, which is correct since 12 − 9 = 3.

(b) 9 − 12:

- 2's complement of B (12): invert 01100 → 10011, add 1 → 10100.
- A + (2's complement of B) = 01001 + 10100 = 11101, with no carry-out.
- 11101 is negative (MSB = 1); its magnitude is found by taking its 2's complement: invert 11101 → 00010, add 1 → 00011 = 3. So the result represents −3, which is correct since 9 − 12 = −3.

Q2. Multiply 9 × 5 (positive numbers) using the shift-and-add method (4-bit registers).

Let BR = 1001 (9), QR = 0101 (5), AC = 0000, E = 0, SC = 4.

Cycle	Q0	Operation	AC	QR	SC
Initial	1	—	0000	0101	4
1	1	AC = AC + BR, then shift right	0100	1010	3
2	0	No add, only shift	0010	0101	2
3	1	AC = AC + BR, then shift right	0101	1010	1
4	0	No add, only shift	0010	1101	0

Table C1: Step-by-step trace of shift-and-add multiplication (9 × 5)

Final product = AC:QR = 0010 1101 = 45 in decimal, which correctly equals 9 × 5.

Q3. Multiply 6 × (−5) using Booth's algorithm (4-bit registers).

Let BR = 0110 (6, multiplicand) and QR = 1011 (−5, multiplier, in 4-bit 2's complement). Then −BR (2's complement of BR) = 1010. AC = 0000, Qn+1 = 0, SC = 4.

Cycle	Qn Qn+1	Operation	AC	QR	Qn+1
Initial	1 0	—	0000	1011	0
1	1 0	AC = AC − BR, then ASR	1101	0101	1
2	1 1	No operation, only ASR	1110	1010	1
3	0 1	AC = AC + BR, then ASR	0010	0101	0
4	1 0	AC = AC − BR, then ASR	1110	0010	1

Table C2: Step-by-step trace of Booth's algorithm for 6 × (−5)

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

Q4. Divide 13 by 4 using the Restoring Division algorithm (4-bit registers).

Let dividend Q = 1101 (13), divisor M = 0100 (4), A = 0000, SC runs for 4 cycles.

Cycle	A after shift	A = A − M	Decision	Final A	Q
1	0001	1 − 4 = −3	Negative → restore, Q0 = 0	0001	1010
2	0011	3 − 4 = −1	Negative → restore, Q0 = 0	0011	0100
3	0110	6 − 4 = 2	Non-negative → Q0 = 1	0010	1001
4	0101	5 − 4 = 1	Non-negative → Q0 = 1	0001	0011

Table C3: Step-by-step trace of restoring division (13 ÷ 4)

Final result: Q = 0011 (quotient = 3), A = 0001 (remainder = 1), matching 13 ÷ 4 = 3 remainder 1.

Q5. Divide 13 by 4 using the Non-Restoring Division algorithm (4-bit registers) and verify the result matches Q4.

Using the same dividend Q = 1101 (13), divisor M = 0100 (4), A = 0000:

Cycle	Prev. sign of A	A after shift	Operation	New A	Q0	Q
1	A ≥ 0	0001	A = A − M = 1 − 4	1101 (−3)	0	1010
2	A < 0	1011 (−5)	A = A + M = −5 + 4	1111 (−1)	0	0100
3	A < 0	1110 (−2)	A = A + M = −2 + 4	0010 (2)	1	1001
4	A ≥ 0	0101 (5)	A = A − M = 5 − 4	0001 (1)	1	0011

Table C4: Step-by-step trace of non-restoring division (13 ÷ 4)

The final remainder A = 0001 (1) is already non-negative, so no end correction is needed. Q = 0011 (quotient = 3), which matches the restoring-division result exactly: quotient 3, remainder 1.

Q6. Add the two floating-point numbers X = 0.1100 × 2^3 and Y = 0.1000 × 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 0.1000 shifted right by 2 bits → 0.0010, with exponent now 3.

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

- Mx = 0.1100, My (aligned) = 0.0010
- Mr = 0.1100 + 0.0010 = 0.1110

Step 3 — Check normalization: the result mantissa 0.1110 already has a 1 as its most significant bit, so it is already normalized; no overflow has occurred (the mantissa did not exceed 1 bit-pattern width), so no further shifting is required.

Final result: Sum = 0.1110 × 2^3. Verification in decimal: X = 0.75 × 8 = 6, Y = 0.5 × 2 = 1, so X + Y should equal 7. Checking the result: 0.1110 binary = 0.875, and 0.875 × 8 = 7. The result is correct.