COMPUTER ORGANIZATION AND ARCHITECTURE

Chapter 2

Instruction Structure and Addressing Modes, Number Representation

Exercise and Exam-Oriented Questions with Answers — Set 2

This is a second, independent set of practice and exam-style questions on the same chapter — Instruction Formats, Addressing Modes, and Number Representation — with new questions and complete model answers, useful as further revision alongside Set 1.

Section A — Very Short Answer Questions (1–2 Marks)

1. Define an operand.

Ans: An operand is the data or the address of the data on which an instruction's operation is to be performed.

2. What is a 3-address instruction? Give one example.

Ans: An instruction that specifies three operand addresses — two source operands and one destination. Example: ADD X, A, B means X ← A + B.

3. Which register temporarily holds the instruction currently being executed?

Ans: The Instruction Register (IR).

4. What is the function of the Control Unit (CU)?

Ans: The Control Unit decodes the opcode of the instruction in IR and generates the control signals needed to carry out the operation.

5. What is register addressing mode?

Ans: An addressing mode in which the operand itself is held in a CPU register, named directly in the instruction, e.g., ADD R1, R2.

6. What is base register addressing used for?

Ans: It is used mainly for program relocation; the effective address is obtained by adding a displacement to the content of a base register that holds the starting address of a program or data block.

7. How many representations of zero exist in 1's complement form?

Ans: Two — a +0 (00000000) and a −0 (11111111) in an 8-bit example.

8. What is the range of an 8-bit unsigned binary number?

Ans: 0 to 255.

9. What does the mantissa of a floating point number represent?

Ans: It represents the precision or significant digits of the number.

10. What is the bias value used in IEEE 754 double precision format?

Ans: 1023.

11. In IEEE 754 format, when is a number's value considered NaN?

Ans: When the exponent field is all 1s and the mantissa field is non-zero.

12. What is the purpose of the system bus in a CPU–memory system?

Ans: It carries address, data, and control signals between the CPU and memory (and other devices) during every fetch and execute operation.

Section B — Short Answer Questions (3–5 Marks)

1. Differentiate between a 0-address instruction and a 3-address instruction with an example.

A 0-address instruction has no explicit operand field; it works on data placed at the top of a stack using PUSH/POP. Example, for X = A + B: PUSH A ; PUSH B ; ADD ; POP X.

A 3-address instruction explicitly names all three operands — two sources and one destination — in a single instruction. Example: ADD X, A, B directly computes X ← A + B. A 3-address format therefore needs far fewer instructions for the same computation, but each instruction word is longer since it must encode three addresses instead of none.

2. Explain base register addressing with a suitable example, and state why it is useful.

In base register addressing, a register (the base register) holds the starting address of a program or a block of data in memory. The effective address is calculated by adding a displacement, given in the instruction, to the content of this base register: EA = [Base Register] + Displacement.

Example: MOV R1, 50(RB). If RB contains 4000, then EA = 4000 + 50 = 4050, so R1 ← Memory[4050]. This mode is useful because a program can be loaded (relocated) to any area of memory simply by changing the value in the base register, without needing to change any of the displacement values inside the program's instructions.

3. What is meant by 'end-around carry' in 1's complement addition? Explain briefly.

When two numbers are added in 1's complement form and the addition produces a carry out of the most significant bit, that carry cannot simply be discarded (unlike in 2's complement addition). Instead, it must be brought back around and added to the least significant bit of the result — this correction step is called the end-around carry.

This extra step makes 1's complement arithmetic circuits more complex than 2's complement circuits, which never require such a correction. This is one of the main reasons 2's complement is preferred in practical computer hardware.

4. Explain relative addressing mode and state why it is commonly used for branch instructions.

In relative addressing, the effective address is obtained by adding a displacement given in the instruction to the current content of the Program Counter (PC): EA = [PC] + Displacement.

It is widely used for branch and jump instructions because it makes the branch target position-independent — the same program can be loaded at different starting locations in memory and the branch will still correctly jump to the intended instruction relative to the current PC value, without needing to know the program's absolute load address.

5. What is the significance of the implicit leading 1 in IEEE 754 normalized mantissa storage?

Every normalized binary floating point number has the form 1.xxxx × 2^E, meaning the digit immediately before the binary point is always 1 (except for the special cases of zero, infinity, and NaN). Since this leading

1 is always present, IEEE 754 does not store it explicitly — only the fractional part (the bits after the point) is stored as the mantissa field.

This 'hidden bit' technique effectively gives one extra bit of precision for free: a 23-bit stored mantissa field in single precision actually represents 24 bits of precision (1 implicit + 23 stored), and similarly a 52-bit field in double precision represents 53 bits of precision.

6. Differentiate between register addressing and register indirect addressing.

In register addressing, the named register itself contains the operand value directly; the instruction operates on that value with no memory access at all. Example: ADD R1, R2 means R1 ← R1 + R2 (R2's own content is used as the operand).

In register indirect addressing, the named register does not hold the operand — it holds the memory address where the operand is stored, so one memory access is required after reading the register. Example: MOV R1, (R2) means R1 ← Memory[content of R2]. Register addressing is therefore faster, since it avoids the extra memory access that register indirect addressing needs.

7. What is meant by overflow in signed binary arithmetic? Under what condition does it occur in 2's complement addition?

Overflow occurs when the result of an arithmetic operation on signed numbers is too large (or too small) to be represented correctly within the fixed number of bits available, causing the sign of the result to be wrong.

In 2's complement addition, overflow can occur only when both operands have the same sign but the result has the opposite sign. For example, adding two large positive numbers may incorrectly produce a negative result if the sum exceeds the maximum representable positive value. Overflow can never occur when adding two numbers of different signs.

8. List the fields of an IEEE 754 single precision number and state the total number of bits used by each.

- Sign (S): 1 bit
- Exponent (E): 8 bits, stored as a biased value with bias = 127
- Mantissa (M): 23 bits, the fractional part of the normalized mantissa

Together these three fields use 1 + 8 + 23 = 32 bits, which is why this format is called single precision (32-bit) floating point representation.

9. Explain, with an example, how indirect addressing differs from register indirect addressing.

In indirect addressing, the address field of the instruction points to a memory location, and that memory location in turn holds the address of the actual operand: EA = Memory[Address field]. Example: MOV R1, @2000 means R1 ← Memory[Memory[2000]].

In register indirect addressing, a register (not a memory location) holds the address of the operand directly: EA = [Register]. Example: MOV R1, (R2) means R1 ← Memory[R2]. Register indirect addressing therefore needs only one memory access to fetch the operand, whereas plain indirect addressing (through a memory pointer) needs two.

10. Why does an increase in the number of address fields in an instruction generally reduce the number of instructions needed for a program, but increase the length of each instruction?

Each additional address field lets a single instruction refer to one more operand or destination directly, so more work can be packed into a single instruction — this is why 3-address machines usually need fewer instructions than 1-address or 0-address machines for the same computation.

However, every address field occupies extra bits within the instruction word to encode a register number or memory address, so instructions with more address fields are inherently longer. There is thus a trade-off between the number of instructions (program length) and the size of each individual instruction (memory used per instruction).

Section C — Numerical / Problem-Solving Questions

1. Represent −67 in 8-bit Sign-Magnitude, 1's Complement, and 2's Complement forms.

Step 1: Convert 67 to 8-bit binary: 67 = 01000011

Sign-Magnitude: Set the MSB to 1, keep remaining 7 bits as magnitude → 11000011

1's Complement: Invert every bit of 01000011 → 10111100

2's Complement: Take the 1's complement (10111100) and add 1 → 10111101

Answer: Sign-Magnitude = 11000011, 1's Complement = 10111100, 2's Complement = 10111101

2. Find the decimal equivalent of the 8-bit 2's complement number 10011100.

Step 1: The MSB is 1, so the number is negative.

Step 2: Take the 2's complement of 10011100: Invert the bits → 01100011; add 1 → 01100100

Step 3: Convert 01100100 to decimal: 01100100 = 64 + 32 + 4 = 100

Answer: 10011100 in 2's complement represents −100 in decimal.

3. Add (−35) and (−20) using 8-bit 2's complement arithmetic and check whether overflow occurs.

Step 1: −35 in 8-bit 2's complement: 35 = 00100011; 1's complement = 11011100; add 1 → 11011101

Step 2: −20 in 8-bit 2's complement: 20 = 00010100; 1's complement = 11101011; add 1 → 11101100

Step 3: Add the two 8-bit numbers:

```
  11011101
+ 11101100
----------
 1 11001001   (the carry out of the 8th bit is discarded)
```

Step 4: Result = 11001001. Both inputs were negative but the result's MSB is also 1 (negative), so no overflow — convert to check: 2's complement of 11001001 → invert = 00110110, add 1 = 00110111 = 55, so the result represents −55.

Verification: (−35) + (−20) = −55, which matches. Since both operands had the same sign and the result also has that sign, there is no overflow (overflow would only occur if the result's sign were positive here).

4. Represent +37.25 in IEEE 754 single precision format.

Step 1: Convert 37 to binary: 37 = 100101

Step 2: Convert 0.25 to binary: 0.25 × 2 = 0.5 → 0; 0.5 × 2 = 1.0 → 1. So 0.25 = .01

Step 3: Combine: 37.25 = 100101.01

Step 4: Normalize: 100101.01 = 1.0010101 × 2^5

Step 5: Sign bit S = 0 (positive number).

Step 6: Biased Exponent = 5 + 127 = 132 = 10000100 (8 bits)

Step 7: Mantissa (23 bits) = 00101010000000000000000

Answer: 0 10000100 00101010000000000000000

5. Represent −6.125 in IEEE 754 single precision format.

Step 1: Convert 6 to binary: 6 = 110

Step 2: Convert 0.125 to binary: 0.125 × 2 = 0.25 → 0; 0.25 × 2 = 0.5 → 0; 0.5 × 2 = 1.0 → 1. So 0.125 = .001

Step 3: Combine: 6.125 = 110.001

Step 4: Normalize: 110.001 = 1.10001 × 2^2

Step 5: Sign bit S = 1 (negative number).

Step 6: Biased Exponent = 2 + 127 = 129 = 10000001 (8 bits)

Step 7: Mantissa (23 bits) = 10001000000000000000000

Answer: 1 10000001 10001000000000000000000

6. An IEEE 754 single precision number is given as: 1 01111110 10000000000000000000000. Find its decimal value.

Step 1: Sign bit S = 1, so the number is negative.

Step 2: Biased Exponent (binary) = 01111110 = 126 (in decimal).

Step 3: Actual Exponent = 126 − 127 = −1.

Step 4: Mantissa bits = 10000000000000000000000, so the normalized number is 1.1 (binary) = 1 + 0.5 = 1.5 (decimal).

Step 5: Value = −(1.5 × 2^−1) = −(1.5 × 0.5) = −0.75

Answer: The decimal value represented is −0.75

7. For the instruction MOV R1, (R2)+, if R2 initially contains 6000 and Memory[6000] = 18, find the value loaded into R1 and the final content of R2.

Addressing mode used: Auto-Increment (Register Indirect with auto-increment).

Step 1: The CPU first accesses the operand at the address currently in R2: EA = [R2] = 6000, so R1 ← Memory[6000] = 18.

Step 2: After the access, R2 is automatically incremented by 1 (assuming byte/word-addressed increment of 1 unit): R2 ← 6000 + 1 = 6001.

Answer: R1 = 18, and the final content of R2 = 6001.

8. A conditional branch instruction BEQ 12 uses relative addressing. If the Program Counter (PC) contains 3080 immediately after fetching this instruction, find the effective (target) address.

In relative addressing, EA = [PC] + Displacement

Here, [PC] = 3080 and Displacement = 12

EA = 3080 + 12 = 3092

Answer: If the branch condition is true, the CPU will jump to memory location 3092.

Section D — Long Answer / Descriptive Questions

1. Compare 0-address, 1-address, 2-address, and 3-address instruction formats on the basis of instruction length, number of instructions required, and typical use, using X = A + B as a running example.

- 0-address (stack-based): PUSH A ; PUSH B ; ADD ; POP X — 4 instructions, each very short (no address field), used in stack-organized machines.
- 1-address (accumulator-based): LOAD A ; ADD B ; STORE X — 3 instructions, each with one address field, used in simple/older CPUs built around a single accumulator.
- 2-address: MOV A, R1 ; ADD R1, B ; MOV X, R1 — 3 instructions, each with two address fields; common in CISC-style machines such as x86.
- 3-address: ADD X, A, B — just 1 instruction, but with three address fields; common in RISC and general register machines.

As the address count increases from 0 to 3, the number of instructions needed for the same computation generally decreases, while the length (and memory footprint) of each individual instruction increases, since more address fields must be encoded within it. The right choice is therefore a trade-off between program size (number of instructions) and instruction word width, influenced by the target CPU's register architecture and memory bus width.

2. With reference to the Fetch–Decode–Execute cycle, explain the role of each of the following: PC, MAR, MDR, IR, and the Control Unit.

- Program Counter (PC): Always holds the memory address of the next instruction to be fetched; it is automatically incremented after each fetch so the CPU knows where to find the following instruction.
- Memory Address Register (MAR): Holds the address of whichever memory location the CPU is currently reading from or writing to, whether that is an instruction fetch or an operand access.
- Memory Data Register (MDR): Holds the actual data being transferred to or from memory at the address in MAR — either the instruction word just fetched, or an operand being read or written.
- Instruction Register (IR): Holds the instruction currently being decoded and executed, once it has been copied in from MDR during the fetch step.
- Control Unit (CU): Decodes the opcode held in IR and, based on it, generates the sequence of control signals needed to drive the ALU, registers, and memory interface through the remaining steps of execution.

Together, these units allow the CPU to fetch an instruction from memory, understand what it must do, and carry it out, before repeating the cycle for the next instruction.

3. Explain the remaining addressing modes not covered in Set 1 — namely register, base register, relative, and auto-increment/decrement — describing when each is preferred over the others.

- Register Addressing: Operand lies directly in a CPU register (e.g., ADD R1, R2). Preferred whenever a value is already in a register, since it avoids memory access entirely and executes fastest.
- Base Register Addressing: EA = [Base register] + Displacement (e.g., MOV R1, 50(RB)). Preferred when a program or data block must be relocatable in memory, since only the base register's value needs to change to move the whole block.
- Relative Addressing: EA = [PC] + Displacement (e.g., JMP 10). Preferred for branch/jump instructions because it makes the target address independent of where the program is actually loaded in memory.
- Auto-Increment/Decrement Addressing: The register used as a pointer is automatically updated after (or before) each access (e.g., MOV R1, (R2)+). Preferred when stepping sequentially through an array or a stack, since it combines data access and pointer update in a single instruction, reducing instruction count.

The right addressing mode is chosen based on where the data physically resides (register vs. memory), whether the location needs to be computed at run time, and whether the program must be position-independent or must traverse a data structure efficiently.

4. Explain overflow in signed 2's complement arithmetic. State the rule for detecting overflow during addition, and illustrate it with one example that causes overflow and one that does not.

Overflow occurs in signed arithmetic when the true mathematical result of an operation cannot be represented correctly within the fixed number of bits used, so the computed result's sign comes out wrong.

Rule: In 2's complement addition, overflow can occur only when the two operands being added have the same sign, and the resulting sum has the opposite sign. If the operands have different signs, overflow can never occur, regardless of the result.

Example with overflow (8-bit, range −128 to +127): +100 (01100100) + +50 (00110010) = 10010110, whose MSB is 1 — this is misread as a negative number, even though two positive numbers were added, because the true sum (150) exceeds the maximum representable value of +127. This is overflow.

Example without overflow: +100 (01100100) + (−50) (11001110). Adding these 8-bit values gives 1 00110010 in 9 bits; discarding the carry out of the MSB leaves 00110010 = 50 in decimal, which correctly matches 100 + (−50) = 50. Here the operands have different signs, so overflow cannot occur, and indeed none does.

5. Discuss the trade-off between range and precision in floating point representation, referring to the roles of the exponent field and the mantissa field.

In a floating point number, the exponent field determines the range of magnitudes that can be represented — a wider exponent field (more bits) allows both very large and very small numbers to be expressed, since it allows the binary point to be shifted further in either direction.

The mantissa field determines the precision of the number — a wider mantissa field (more bits) allows more significant digits to be stored, so numbers can be represented with less rounding error.

Since the total number of bits in a given format (such as 32 or 64) is fixed, there is a direct trade-off: allocating more bits to the exponent increases range but leaves fewer bits for the mantissa, reducing precision, and vice versa. IEEE 754 double precision improves on single precision by widening both fields (11-bit exponent and 52-bit mantissa versus 8-bit and 23-bit), thereby improving range and precision simultaneously, but at the cost of doubling the storage required per number.

6. Explain why IEEE 754 does not store the leading 1 of a normalized mantissa, and how this affects the actual precision obtained from the stored mantissa bits.

Every normalized binary floating point number (other than the special cases of zero, infinity, and NaN) has exactly one non-zero digit before the binary point, and in binary that digit must always be 1. Since this leading 1 is guaranteed to be present in every normalized number, there is no need to spend a bit storing it — it is simply assumed ('implicit' or 'hidden') and reconstructed by hardware whenever the number is used.

This means the stored mantissa field only needs to record the bits after the binary point. As a result, a 23-bit stored field in single precision actually represents 24 bits of precision (1 implicit + 23 stored), and a 52-bit stored field in double precision represents 53 bits of precision. This hidden-bit technique effectively gives one extra bit of precision 'for free' compared to explicitly storing the leading 1.

7. Explain how comparison of two IEEE 754 floating point numbers can be performed using ordinary unsigned integer comparison of their bit patterns, and explain the role of the biased exponent in making this possible.

For two positive normalized floating point numbers, comparing which one is larger can be done by comparing their full 32-bit (or 64-bit) patterns as if they were plain unsigned integers, without decoding the sign, exponent, and mantissa separately. This works because the fields are arranged in the specific order Sign, Exponent, Mantissa (most significant to least significant), so a number with a larger exponent field always has a larger overall bit pattern, and if the exponents are equal, a larger mantissa field always gives a larger overall bit pattern.

This property depends entirely on the exponent field being stored as a biased (always non-negative) value. If the exponent were instead stored using 2's complement or sign-magnitude, a negative actual exponent could produce a bit pattern that looks numerically larger than a positive actual exponent when compared as an unsigned integer, breaking this simple comparison technique. By adding a fixed bias so that all stored exponents are non-negative and increase monotonically with the actual exponent, IEEE 754 allows hardware to compare floating point magnitudes using the same simple comparator circuit used for unsigned integers.

8. Convert the decimal number +0.1875 into IEEE 754 single precision floating point representation, showing all the steps clearly.

Step 1: Convert 0.1875 to binary by repeated multiplication by 2:

0.1875 × 2 = 0.375 → 0; 0.375 × 2 = 0.75 → 0; 0.75 × 2 = 1.5 → 1; 0.5 × 2 = 1.0 → 1

So 0.1875 = .0011 in binary.

Step 2: Normalize: 0.0011 = 1.1 × 2^−3 (the binary point is shifted 3 places to the right, so the exponent is −3).

Step 3: Sign bit S = 0, since the number is positive.

Step 4: Biased Exponent = Actual Exponent + Bias = −3 + 127 = 124 = 01111100 (8 bits)

Step 5: Mantissa (23 bits, pad remaining positions with 0) = 10000000000000000000000

Step 6: Final IEEE 754 single precision representation:

0 01111100 10000000000000000000000