COMPUTER ORGANIZATION AND ARCHITECTURE

Chapter 2

Instruction Structure and Addressing Modes, Number Representation

Exercise and Exam-Oriented Questions with Answers

This booklet contains previous-exam-style and practice questions covering Instruction Formats, Addressing Modes, and Number Representation, each with a complete model answer or worked solution — useful for revision before examinations.

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

1. What is an instruction format?

Ans: The way an instruction is divided into fields such as opcode and operand address(es) is called its instruction format.

2. Define opcode.

Ans: Opcode (operation code) is the part of an instruction that specifies the operation to be performed, such as ADD, SUB, or MOV.

3. What is a 0-address instruction? Give one example.

Ans: An instruction that does not specify any operand address explicitly and operates on data at the top of a stack. Example: ADD (adds the top two values of the stack).

4. Which register holds the address of the next instruction to be executed?

Ans: The Program Counter (PC).

5. What is an addressing mode?

Ans: An addressing mode is the method used to specify the operand of an instruction, i.e., how the effective address of the data is determined.

6. What is immediate addressing?

Ans: An addressing mode in which the operand value is given directly within the instruction itself, e.g., MOV R1, #5.

7. What is meant by 'effective address' (EA)?

Ans: The actual memory address (or register) where the required operand is finally located, after applying the rule of the addressing mode used.

8. What is the range of an n-bit signed number in 2's complement form?

Ans: The range is from −2^(n−1) to +2^(n−1) − 1. For n = 8, the range is −128 to +127.

9. What is normalization in floating point representation?

Ans: Adjusting the mantissa so that there is exactly one non-zero digit before the radix point (i.e., mantissa of the form 1.xxxx in binary).

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

Ans: 127.

11. How many bits are used for the exponent and mantissa in IEEE 754 double precision format?

Ans: 11 bits for the exponent and 52 bits for the mantissa (with 1 bit for sign, totalling 64 bits).

12. What does MAR stand for and what is its function?

Ans: Memory Address Register — it holds the address of the memory location currently being accessed.

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

1. Differentiate between a 1-address instruction and a 2-address instruction with an example.

A 1-address instruction specifies only one operand address; the second operand and the result both use an implied register called the accumulator (AC). Example, for X = A + B: LOAD A ; ADD B ; STORE X.

A 2-address instruction specifies two operand addresses, and one of them also acts as the destination, so the value stored there is overwritten. Example, for X = A + B: MOV A, R1 ; ADD R1, B ; MOV X, R1. Thus a 2-address instruction generally needs fewer instructions than a 1-address one, but each instruction is longer since it carries two address fields instead of one.

2. Explain register indirect addressing with a suitable example.

In register indirect addressing, a CPU register does not hold the operand itself but holds the memory address where the operand is stored. The CPU first reads the register to get this address, and then accesses memory at that address to obtain the operand.

Example: MOV R1, (R2) means R1 ← Memory[content of R2]. If R2 contains 3000, and memory location 3000 contains the value 25, then after execution R1 = 25. This mode is useful for working with pointers and linked structures.

3. Why is 2's complement representation preferred over sign-magnitude and 1's complement in modern computers?

2's complement has only a single representation of zero, unlike sign-magnitude and 1's complement, which both have a +0 and a −0. This removes ambiguity and simplifies the design of comparison circuits.

Addition and subtraction can be performed using the same simple binary adder circuit for both positive and negative numbers in 2's complement form, without needing an end-around carry correction (which 1's complement requires). This reduces hardware complexity and increases speed, which is why virtually all modern processors use 2's complement for integer arithmetic.

4. What are the special values represented in IEEE 754 format? Explain briefly.

- Zero: represented when the exponent field is all 0s and the mantissa is all 0s (sign bit decides +0 or −0).
- Infinity: represented when the exponent field is all 1s and the mantissa is all 0s (sign bit decides +∞ or −∞).
- NaN (Not a Number): represented when the exponent field is all 1s and the mantissa is non-zero; it is used to represent undefined or invalid results such as 0/0 or √(−1).

5. Briefly explain the Fetch–Decode–Execute cycle.

Fetch: The address held in the Program Counter (PC) is placed in the Memory Address Register (MAR); the instruction at that address is read from memory into the Memory Data Register (MDR) and then copied into the Instruction Register (IR). The PC is then incremented.

Decode: The Control Unit examines the opcode part of the instruction in IR to determine the operation to be performed and the addressing mode used.

Execute: The required operands are fetched (if in memory or registers), the ALU performs the operation, and the result is stored back in a register or in memory. This three-step cycle repeats continuously for every instruction of a program.

6. What are the functions of MAR and MDR in a computer system?

The Memory Address Register (MAR) holds the address of the memory location that the CPU wants to read from or write to. The Memory Data Register (MDR), also called the Memory Buffer Register (MBR), holds the actual data being transferred — either the data just read from that memory location, or the data about to be written into it. Together, MAR and MDR act as the interface between the CPU and main memory during every memory access.

7. Explain indexed addressing mode with a suitable example.

In indexed addressing, the effective address is calculated by adding the content of an index register to a constant (base/displacement) given in the instruction: EA = Base address + [Index register]. It is particularly useful for accessing successive elements of an array, since the index register can simply be incremented in a loop to move from one element to the next.

Example: MOV R1, 2000(R2). If R2 contains 5, then EA = 2000 + 5 = 2005, so R1 ← Memory[2005].

8. What is a biased exponent? Why is it used in floating point representation instead of a signed exponent?

A biased exponent is obtained by adding a fixed constant (the bias) to the actual exponent before storing it, so that the stored value is always non-negative: Biased Exponent = Actual Exponent + Bias.

It is used because storing the exponent directly in signed (2's complement) form would make it difficult to compare two floating point numbers using ordinary unsigned comparison of bit patterns. With a biased exponent, larger actual exponents always produce larger stored values, so two floating point numbers can be compared correctly using simple unsigned integer comparison of their bit patterns — this greatly simplifies comparison hardware.

9. Distinguish between direct addressing and indirect addressing.

In direct addressing, the address field of the instruction directly gives the memory address of the operand: EA = Address field. Example: MOV R1, 2000 means R1 ← Memory[2000].

In indirect addressing, the address field gives the address of a memory location which itself holds the address of the actual operand (i.e., a pointer to the operand): EA = Memory[Address field]. Example: MOV R1, @2000 means R1 ← Memory[Memory[2000]]. Indirect addressing therefore requires one extra memory access compared to direct addressing, but allows the actual operand address to be changed at run time.

10. What is auto-increment addressing mode? Give an example.

Auto-increment addressing is a special case of register indirect addressing in which the content of the register (which holds a memory address) is automatically incremented after it is used to access the operand. This is very useful for stepping sequentially through the elements of an array or a stack, since it combines the memory access and pointer update into a single instruction.

Example: MOV R1, (R2)+ means R1 ← Memory[R2], and then R2 ← R2 + 1. On the next execution of the same instruction, the next element in memory is automatically accessed.

Section C — Numerical / Problem-Solving Questions

1. Represent −45 in 8-bit Sign-Magnitude, 1's Complement, and 2's Complement forms.

Step 1: Convert 45 to 8-bit binary: 45 = 00101101

Sign-Magnitude: Set the MSB to 1 to indicate a negative sign, keep the remaining 7 bits as magnitude → 10101101

1's Complement: Invert every bit of 00101101 → 11010010

2's Complement: Take the 1's complement (11010010) and add 1 → 11010011

Answer: Sign-Magnitude = 10101101, 1's Complement = 11010010, 2's Complement = 11010011

2. Find the decimal equivalent of the 8-bit 2's complement number 11010110.

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

Step 2: To find the magnitude, take the 2's complement of 11010110: Invert the bits → 00101001; add 1 → 00101010

Step 3: Convert 00101010 to decimal: 00101010 = 32 + 8 + 2 = 42

Answer: 11010110 in 2's complement represents −42 in decimal.

3. Add (+25) and (−18) using 8-bit 2's complement arithmetic and verify the result.

Step 1: +25 in 8-bit binary = 00011001

Step 2: −18 in 8-bit 2's complement: 18 = 00010010; 1's complement = 11101101; add 1 → 11101110

Step 3: Add the two 8-bit numbers:

```
  00011001
+ 11101110
----------
 1 00000111   (the carry out of the 8th bit is discarded)
```

Step 4: Result = 00000111 = 7 in decimal.

Verification: 25 + (−18) = 7, which matches the computed result. Since both operands had different signs, no overflow can occur, and the discarded carry is normal in 2's complement addition.

4. Represent +25.625 in IEEE 754 single precision format.

Step 1: Convert 25 to binary: 25 = 11001

Step 2: Convert 0.625 to binary: 0.625 × 2 = 1.25 → 1; 0.25 × 2 = 0.5 → 0; 0.5 × 2 = 1.0 → 1. So 0.625 = .101

Step 3: Combine: 25.625 = 11001.101

Step 4: Normalize: 11001.101 = 1.1001101 × 2^4

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

Step 6: Biased Exponent = Actual Exponent + Bias = 4 + 127 = 131 = 10000011 (8 bits)

Step 7: Mantissa (23 bits, pad with zeros after the fractional part) = 10011010000000000000000

Answer: 0 10000011 10011010000000000000000

5. Represent –12.5 in IEEE 754 single precision format.

Step 1: Convert 12 to binary: 12 = 1100

Step 2: Convert 0.5 to binary: 0.5 = .1

Step 3: Combine: 12.5 = 1100.1

Step 4: Normalize: 1100.1 = 1.1001 × 2^3

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

Step 6: Biased Exponent = 3 + 127 = 130 = 10000010 (8 bits)

Step 7: Mantissa (23 bits) = 10010000000000000000000

Answer: 1 10000010 10010000000000000000000

6. An IEEE 754 single precision number is given as: 0 10000001 01000000000000000000000. Find its decimal value.

Step 1: Sign bit S = 0, so the number is positive.

Step 2: Biased Exponent (in binary) = 10000001 = 129 (in decimal).

Step 3: Actual Exponent = Biased Exponent − Bias = 129 − 127 = 2.

Step 4: Mantissa bits = 01000000000000000000000, so the normalized number is 1.01 (binary) = 1 + 0.25 = 1.25 (decimal).

Step 5: Value = 1.25 × 2^2 = 1.25 × 4 = 5.0

Answer: The decimal value represented is +5.0

7. For the instruction MOV R1, 500(R2), if the content of register R2 is 300, find the effective address. Which addressing mode is used?

Addressing mode used: Indexed Addressing (a constant displacement of 500 is added to the content of index register R2).

Effective Address (EA) = Displacement + [R2] = 500 + 300 = 800

Answer: EA = 800; the CPU will access Memory[800] and load its content into R1.

8. A branch instruction JMP −8 uses relative addressing. If the Program Counter (PC) contains 2050 immediately after fetching this instruction, find the effective (target) address.

In relative addressing, the effective address is calculated as: EA = [PC] + Displacement

Here, [PC] = 2050 and Displacement = −8

EA = 2050 + (−8) = 2042

Answer: The CPU will branch to (jump to) memory location 2042.

Section D — Long Answer / Descriptive Questions

1. Explain the different instruction formats (0-address, 1-address, 2-address, and 3-address) with suitable examples. Discuss their relative advantages and disadvantages.

The number of address fields present in an instruction decides how it is classified:

- 0-address instruction: No operand address is given; operations act on the top of a stack. Example, for X = (A+B)*C: PUSH A; PUSH B; ADD; PUSH C; MUL; POP X.
- 1-address instruction: One operand address is given; the other operand and the result use an implied accumulator (AC). Example, for X = A+B: LOAD A; ADD B; STORE X.
- 2-address instruction: Two operand addresses are given; one of them is overwritten with the result. Example: MOV A, R1; ADD R1, B; MOV X, R1.
- 3-address instruction: Three operand addresses are given — two sources and one destination. Example: ADD X, A, B.

As the number of address fields increases, fewer instructions are needed to perform the same computation (shorter, more readable programs), but each instruction word becomes longer, requiring more memory and wider buses. Conversely, 0-address and 1-address formats need shorter instruction words but require more instructions overall, along with additional load/store or push/pop operations to move data in and out of the stack or accumulator.

2. Describe the sequence of steps involved in the execution of a typical instruction through various parts of the CPU and memory.

Every instruction passes through a repeating Fetch–Decode–Execute cycle, using the following units: Program Counter (PC), Memory Address Register (MAR), Memory Data/Buffer Register (MDR/MBR), Instruction Register (IR), Control Unit (CU), Arithmetic Logic Unit (ALU), and the General Purpose Registers.

1. Fetch: [PC] is copied into MAR; the instruction is read from that memory address into MDR and then into IR; PC is incremented to point to the next instruction.
2. Decode: The Control Unit decodes the opcode in IR to identify the operation and the addressing mode(s) used.
3. Operand Fetch: If required, operand addresses are computed and placed in MAR, and the operands are fetched into MDR or directly into CPU registers.
4. Execute: The ALU carries out the required arithmetic or logic operation on the fetched operands.
5. Store Result: The result is written back into a register, or via MDR and MAR into a memory location.

This cycle then repeats for the instruction now pointed to by PC, continuing until the program ends.

3. Explain any six addressing modes with suitable examples, and state one typical use for each.

- Immediate: MOV R1, #5 → operand (5) is part of the instruction itself. Use: loading constants.
- Direct: MOV R1, 2000 → EA = 2000. Use: accessing simple variables.
- Indirect: MOV R1, @2000 → EA = Memory[2000]. Use: working with pointers.
- Register: ADD R1, R2 → operand is in register R2 itself. Use: fast access without memory reference.
- Register Indirect: MOV R1, (R2) → EA = content of R2. Use: pointers held in registers.
- Indexed: MOV R1, 2000(R2) → EA = 2000 + [R2]. Use: accessing array elements.

Other important modes include Base Register addressing (EA = base register + displacement, used for program relocation), Relative addressing (EA = PC + displacement, used in branch instructions), and Auto-Increment/Decrement addressing (register is automatically updated after use, useful for stack and array traversal).

4. Explain the different methods of representing signed integers in a computer system, with an example for each.

An n-bit signed integer can be represented in three common ways:

- Sign-Magnitude: MSB is the sign bit (0 = positive, 1 = negative); remaining bits give the magnitude. Example (8-bit): +9 = 00001001, −9 = 10001001. Drawback: two representations of zero (+0 and −0).
- 1's Complement: A negative number is formed by inverting every bit of the positive number. Example: −9 = 11110110. Drawback: also has two representations of zero, and needs end-around carry correction during addition.
- 2's Complement: A negative number is formed by taking the 1's complement and adding 1. Example: −9 = 11110111. This has only one representation of zero and allows simple binary addition circuits to handle both positive and negative numbers, which is why it is used in virtually all modern computers.

For an n-bit 2's complement number, the representable range is −2^(n−1) to +2^(n−1) − 1; for n = 8, this is −128 to +127.

5. Explain floating point representation and the concept of normalization, with an example of converting a decimal number into normalized binary floating point form.

A floating point number is represented using three fields: Sign (S), Exponent (E), and Mantissa/Significand (M), giving a value of the form N = (−1)^S × M × 2^E. This scientific-notation-like form allows a very wide range of magnitudes — from very large to very small numbers — to be represented using a fixed number of bits.

Normalization means adjusting the mantissa so that there is exactly one non-zero digit before the binary point, i.e., the mantissa takes the form 1.xxxx. This ensures a unique representation for every number and makes the best use of the available mantissa bits for precision.

Example: Convert 10.75 to normalized binary floating point form. 10.75 in binary is 1010.11. Normalizing gives 1.01011 × 2^3, so Sign = 0, true Exponent = 3, and Mantissa (fractional part) = 01011.

6. What is a biased exponent? Explain, with reasoning, why it is used in floating point representation instead of storing the exponent in signed form.

The actual exponent of a floating point number can be positive or negative. If this signed exponent were stored directly (e.g., in 2's complement form), comparing the magnitudes of two floating point numbers using simple unsigned bit-pattern comparison would not work correctly, since a negative exponent would appear numerically 'larger' when compared as an unsigned pattern than certain positive exponents.

To solve this, a fixed positive constant called the bias is added to the actual exponent before it is stored: Biased Exponent = Actual Exponent + Bias. This shifts the whole range of exponents so that the stored value is always non-negative, and larger actual exponents always correspond to larger stored (biased) values.

As a result, two floating point numbers can be compared for magnitude by directly comparing their stored bit patterns as if they were unsigned integers, without any special sign-handling logic — this considerably simplifies comparator hardware. IEEE 754 uses a bias of 127 for single precision and 1023 for double precision.

7. Explain the IEEE 754 single precision and double precision floating point formats in detail, giving the bit allocation for sign, exponent, and mantissa in each case.

IEEE 754 is the standard used almost universally for floating point representation. It defines two common formats:

Single Precision (32 bits):

- Sign (S): 1 bit (0 = positive, 1 = negative)
- Exponent (E): 8 bits, stored as a biased exponent with bias = 127
- Mantissa (M): 23 bits, the fractional part of the normalized mantissa (leading 1 is implicit)

Value = (−1)^S × 1.M × 2^(E − 127)

Double Precision (64 bits):

- Sign (S): 1 bit
- Exponent (E): 11 bits, stored as a biased exponent with bias = 1023
- Mantissa (M): 52 bits

Value = (−1)^S × 1.M × 2^(E − 1023)

Double precision uses more bits for both the exponent and the mantissa than single precision, giving it a wider range of representable magnitudes and greater numerical precision, at the cost of using twice the storage (64 bits instead of 32 bits) per number.

8. Convert the decimal number −29.5 into IEEE 754 single precision floating point representation, showing all the steps clearly.

Step 1: Convert the magnitude 29 to binary: 29 = 11101

Step 2: Convert 0.5 to binary: 0.5 = .1

Step 3: Combine: 29.5 = 11101.1

Step 4: Normalize: 11101.1 = 1.11011 × 2^4

Step 5: Sign bit S = 1, since the number is negative.

Step 6: Biased Exponent = Actual Exponent + Bias = 4 + 127 = 131 = 10000011 (8 bits)

Step 7: Mantissa (23 bits, pad remaining positions with 0) = 11011000000000000000000

Step 8: Final IEEE 754 single precision representation:

1 10000011 11011000000000000000000