COMPUTER ORGANIZATION AND ARCHITECTURE

Chapter 2

Instruction Structure and Addressing Modes, Number Representation

---

2.1 Instruction Format, 0/1/2/3-Address Instructions, and Execution of an Instruction

Instruction Format

A machine instruction is the basic unit of work given to the CPU. Every instruction is divided into fields, and the arrangement of these fields is called the instruction format. In general, an instruction format has two main parts:

- Opcode (Operation Code) — specifies the operation to be performed, such as ADD, SUB, MOV, or LOAD.
- Operand / Address field(s) — specify the data or the location (register, memory address) on which the operation is to be performed.

The number of address fields present in an instruction is called its 'address count', and this decides how many operands can be referred to directly by that instruction. Based on this, instructions are classified as 0, 1, 2, or 3-address instructions.

0-Address Instructions (Stack-Based)

A 0-address instruction does not specify any operand address explicitly. It operates on data present at the top of a stack. Such instructions are used in stack-organized computers, where PUSH and POP operations move data between memory and the stack.

Example: To evaluate X = (A + B) * C using a stack machine:

- PUSH A
- PUSH B
- ADD (pops A and B, pushes A+B)
- PUSH C
- MUL (pops top two values, pushes result)
- POP X (stores result into X)

1-Address Instructions (Accumulator-Based)

A 1-address instruction specifies only one operand address explicitly. The other operand is implied to be in a special register called the accumulator (AC), and the result is also stored back in the accumulator.

Example: To compute X = A + B:

- LOAD A (AC ← A)
- ADD B (AC ← AC + B)
- STORE X (X ← AC)

2-Address Instructions

A 2-address instruction specifies two operand addresses. Usually, one of the two addresses also serves as the destination where the result is stored, so the original value at that address is overwritten.

Example: To compute X = A + B (result stored in A itself):

- MOV A, R1 (copy A into register R1)
- ADD R1, B (R1 ← R1 + B)
- MOV X, R1 (store result into X)

3-Address Instructions

A 3-address instruction specifies three operand addresses — two for the source operands and one for the destination. This format is common in RISC and general register-based machines.

Example: To compute X = A + B:

- ADD X, A, B (X ← A + B)

Larger address instructions generally give shorter and more readable programs but require longer instruction words (more memory per instruction), whereas fewer address fields save memory per instruction but need more instructions to do the same job.

Execution of a Typical Instruction Through the CPU and Memory

Every instruction goes through a repeating cycle known as the Fetch–Decode–Execute cycle. The following registers and units participate in this cycle:

- Program Counter (PC) — holds the address of the next instruction to be fetched.
- Memory Address Register (MAR) — holds the address of the memory location currently being accessed.
- Memory Data/Buffer Register (MDR/MBR) — holds the data read from, or to be written to, memory.
- Instruction Register (IR) — holds the instruction currently being decoded and executed.
- Control Unit (CU) — decodes the opcode in IR and generates control signals for all other units.
- Arithmetic Logic Unit (ALU) — performs the actual arithmetic/logical operation.
- General Purpose Registers — hold operands and intermediate/final results.

The typical steps are as follows:

1. Fetch: The address in PC is copied into MAR. The instruction at that memory address is read into MDR and then copied into IR. PC is incremented to point to the next instruction.
2. Decode: The Control Unit examines (decodes) the opcode part of IR to determine which operation is to be performed and which addressing mode is used.
3. Operand Fetch: If operands are in memory, their addresses are calculated and placed in MAR; the operands are then fetched into MDR or directly into CPU registers.
4. Execute: The ALU performs the required operation (e.g., addition, comparison) using the fetched operands.

5. Store Result: The result produced by the ALU is written back — either into a general-purpose register or, via MDR and MAR, into a memory location.

This cycle then repeats for the next instruction pointed to by PC. Figure 2.1 below shows how data flows between memory and the CPU's internal units during this cycle.

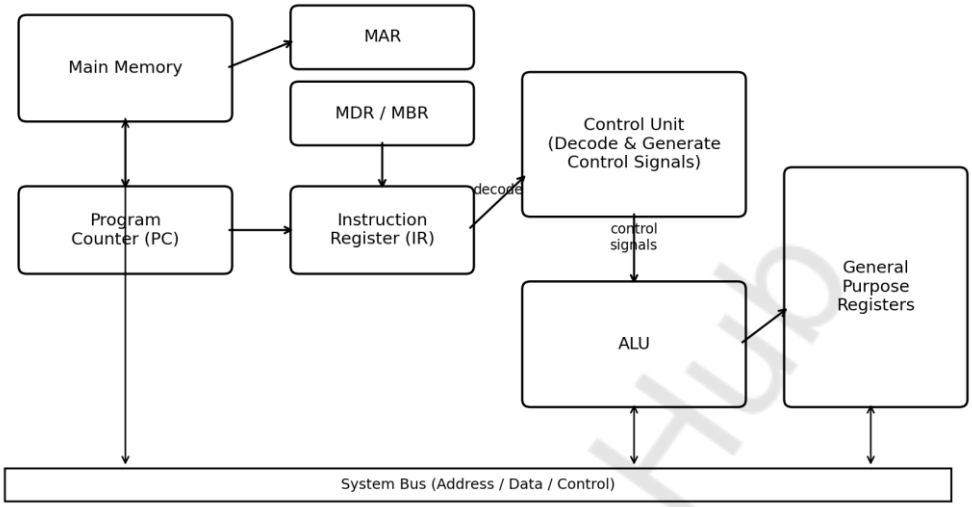

Fig 2.1: Flow of an Instruction Through the CPU and Memory

2.2 Addressing Modes

An addressing mode is the method by which the operand of an instruction is specified — that is, how the CPU determines the actual (effective) address of the data to be used. Addressing modes give programs flexibility, allow compact code, and support data structures such as arrays and pointers. The important addressing modes are described below.

1. Immediate Addressing

The operand value itself is given directly in the instruction; no memory reference is needed to fetch it.

Example: MOV R1, #5 ; R1 ← 5 (the value 5 is used directly)

2. Direct (Absolute) Addressing

The address field of the instruction directly gives the memory address where the operand is stored.

Example: MOV R1, 2000 ; R1 ← Memory[2000]

3. Indirect Addressing

The address field gives the address of a memory location which itself contains the address of the actual operand (a pointer).

Example: MOV R1, @2000 ; R1 ← Memory[Memory[2000]]

4. Register Addressing

The operand is held in a CPU register, and the instruction names that register directly. This is fast since no memory access is required.

Example: ADD R1, R2 ; R1 ← R1 + R2

5. Register Indirect Addressing

The named register holds the address of the memory location containing the operand, rather than the operand itself.

Example: MOV R1, (R2) ; R1 ← Memory[content of R2]

6. Indexed Addressing

The effective address is obtained by adding the content of an index register to a constant (base address) given in the instruction. It is widely used for accessing elements of an array.

Example: MOV R1, 2000(R2) ; Effective Address = 2000 + [R2]

7. Base Register Addressing

Similar to indexed addressing, but the register used is designated as a base register that holds the starting (base) address of a program or data block; a displacement is added to it to get the effective address.

Example: MOV R1, 50(RB) ; Effective Address = [RB] + 50

8. Relative Addressing

The effective address is obtained by adding the displacement given in the instruction to the current content of the Program Counter (PC). It is commonly used in branch/jump instructions.

Example: JMP 10 ; Effective Address = [PC] + 10

9. Auto-Increment / Auto-Decrement Addressing

A special case of register indirect addressing in which the register content is automatically incremented or decremented after (or before) it is used to access memory. This is very useful for stepping through arrays or stacks.

Example: MOV R1, (R2)+ ; R1 ← Memory[R2] ; then R2 ← R2 + 1

Addressing Mode	Effective Address / Operand	Typical Use
Immediate	Operand is part of instruction	Constants
Direct	EA = Address field	Simple variables
Indirect	EA = Memory[Address field]	Pointers
Register	Operand in register	Fast access
Register Indirect	EA = content of register	Pointers in registers
Indexed	EA = Base + Index register	Array access
Base Register	EA = Base register + Displacement	Relocation of programs
Relative	EA = PC + Displacement	Branch instructions
Auto Increment/Decrement	EA = register, then updated	Stack/array traversal

2.3 Representation of Integers in Computer System

Computers store all data, including numbers, as binary digits (0s and 1s). Integers can be represented in several ways, depending on whether they are signed or unsigned.

Unsigned Integer Representation

In this form, all bits represent the magnitude of the number; there is no sign bit. An n-bit unsigned number can represent values from 0 to (2^n − 1). For example, an 8-bit unsigned number ranges from 0 to 255.

Signed Integer Representation

To represent both positive and negative numbers, one bit (usually the most significant bit, MSB) is reserved to indicate the sign, while the remaining bits represent the magnitude. There are three common schemes:

(a) Sign-Magnitude Representation

The MSB represents the sign (0 for positive, 1 for negative), and the remaining (n−1) bits represent the magnitude of the number.

Example (8-bit): +9 = 00001001, −9 = 10001001

Drawback: There are two representations of zero (+0 = 00000000 and −0 = 10000000), and arithmetic circuits are more complex.

(b) 1's Complement Representation

A negative number is obtained by inverting (complementing) every bit of the corresponding positive number.

Example (8-bit): +9 = 00001001, −9 = 11110110 (each bit of 00001001 inverted)

Drawback: Like sign-magnitude, it also has two representations of zero (00000000 and 11111111), and requires an 'end-around carry' correction during addition.

(c) 2's Complement Representation

A negative number is obtained by taking the 1's complement of the positive number and adding 1 to it. This is the representation used by almost all modern computers because it has a single representation for zero and allows addition/subtraction to be performed with simple binary addition circuits.

Example (8-bit): +9 = 00001001. 1's complement = 11110110. Adding 1 gives −9 = 11110111.

Range for an n-bit 2's complement number: −2^(n−1) to +2^(n−1) − 1. For example, an 8-bit signed number ranges from −128 to +127.

Decimal	Sign-Magnitude	1's Complement	2's Complement
+9	00001001	00001001	00001001
−9	10001001	11110110	11110111

2.4 Representation of Floating Point Numbers in Computer System

Integer representations cannot conveniently express very large numbers, very small numbers, or fractional (real) numbers. Floating point representation solves this by expressing a number in a form similar to scientific notation, allowing a very wide range of magnitudes to be represented using a fixed number of bits.

General Floating Point Format

A floating point number is generally represented using three fields:

- Sign (S) — indicates whether the number is positive or negative.
- Exponent (E) — represents the power to which the base (usually 2) is raised; it determines the range/magnitude of the number.
- Mantissa / Significand (M) — represents the precision or significant digits of the number.

The value represented is generally of the form: $N = (-1)^S \times M \times 2^E$

Normalization

A floating point number is said to be normalized when the mantissa is adjusted so that there is exactly one non-zero digit before the radix (binary) point, i.e. the mantissa is of the form 1.xxxx in binary. Normalization ensures a unique representation for each number and makes the maximum use of the available mantissa bits for precision.

Example: The binary number 1011.101 is normalized as 1.011101×2^3.

Example Conversion

To represent the decimal number 10.75 in binary floating point:

- Convert 10.75 to binary: 10.75 = 1010.11
- Normalize: $1010.11 = 1.01011 \times 2^3$
- Here, Sign = 0 (positive), true Exponent = 3, Mantissa (fractional part after the leading 1) = 01011

2.5 Biased Exponent and IEEE 754 Format

Need for a Biased Exponent

The exponent of a floating point number can itself be positive or negative. Storing it in signed (2's complement) form would make comparison of two floating point numbers difficult, because comparing exponents would require separate sign-handling logic.

To avoid this, a fixed constant called the bias is added to the actual exponent before storing it, so that the stored exponent is always a non-negative number. This is called the biased exponent, given by:

Biased Exponent = Actual Exponent + Bias

Using a biased exponent allows two floating point numbers to be compared using simple unsigned integer comparison of their stored bit patterns, which greatly simplifies hardware for comparison and ordering.

IEEE 754 Standard

IEEE 754 is the most widely adopted standard for representing floating point numbers in computers. It defines two common formats — single precision and double precision — each consisting of a sign bit, a biased exponent field, and a mantissa (fraction) field.

IEEE 754 Single Precision (32-bit) Format

- Sign (S): 1 bit — 0 for positive, 1 for negative.
- Exponent (E): 8 bits — stored as a biased exponent with a bias of 127.
- Mantissa (M): 23 bits — fractional part of the normalized mantissa (the leading 1 is implicit and not stored).

Value = (−1)^S × 1.M × 2^(E − 127)

IEEE 754 Double Precision (64-bit) Format

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

Value = (−1)^S × 1.M × 2^(E − 1023)

Figure 2.2 below shows the bit-field layout of both formats.

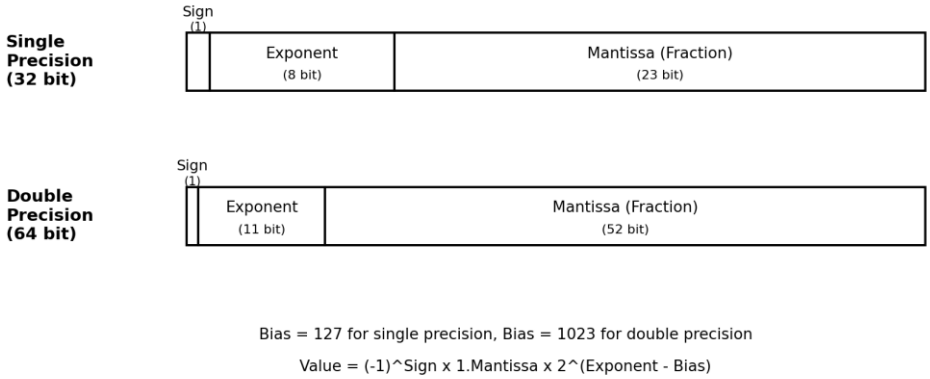

Fig 2.2: IEEE 754 Floating Point Format

Worked Example (Single Precision)

Represent +10.75 in IEEE 754 single precision format:

6. Normalize: 10.75 = 1010.11 = 1.01011 × 2^3
7. Sign bit S = 0 (positive number).
8. Biased Exponent = Actual Exponent + Bias = 3 + 127 = 130 = 10000010 (8 bits).
9. Mantissa (23 bits, pad with zeros) = 01011000000000000000000
10. Final IEEE 754 representation: 0 10000010 01011000000000000000000

Special Values in IEEE 754

- Exponent all 0s and Mantissa 0: represents zero (+0 or −0 depending on sign bit).
- Exponent all 1s and Mantissa 0: represents Infinity (+∞ or −∞).
- Exponent all 1s and Mantissa non-zero: represents NaN (Not a Number), used for undefined results such as 0/0.

Multiple Choice Questions (MCQs)

Answer the following questions by choosing the most appropriate option.

1. In a 0-address instruction format, operands are accessed using:

(a) Accumulator

(b) Stack

(c) General purpose registers

(d) Index register

2. Which instruction format requires the fewest bits per instruction on average but the largest number of instructions for a given computation?

(a) 3-address

(b) 2-address

(c) 1-address

(d) 0-address

3. The register that holds the address of the next instruction to be executed is:

(a) MAR

(b) MDR

(c) Program Counter (PC)

(d) Instruction Register (IR)

4. The Instruction Register (IR) is used to:

(a) Hold the address of data in memory

(b) Hold the instruction currently being executed

(c) Hold the result of ALU operation

(d) Point to the top of the stack

5. In MOV R1, #5, the addressing mode used is:

(a) Direct

(b) Immediate

(c) Indirect

(d) Register Indirect

6. If R2 contains the address of the operand, then MOV R1, (R2) uses:

(a) Register Addressing

(b) Register Indirect Addressing

(c) Indexed Addressing

(d) Immediate Addressing

7. Which addressing mode is most suitable for accessing elements of an array?

(a) Immediate

(b) Indexed

(c) Direct

(d) Relative

8. Branch/jump instructions commonly use which addressing mode?

(a) Base Register

(b) Relative Addressing

(c) Immediate

(d) Register

9. In auto-increment addressing mode, the register content is:

(a) Decremented before use

(b) Left unchanged

(c) Incremented after being used to access memory

(d) Used only for branching

10. Which of the following is NOT a valid signed integer representation?

(a) Sign-Magnitude

(b) 1's Complement

(c) 2's Complement

(d) Grey Code

11. The range of an 8-bit signed number in 2's complement form is:

(a) −128 to +127

(b) −127 to +127

(c) 0 to 255

(d) −128 to +128

12. Which representation of signed integers has only ONE representation for zero?

(a) Sign-Magnitude

(b) 1's Complement

(c) 2's Complement

(d) None of these

13. A floating point number is said to be normalized when:

(a) The exponent is zero

(b) There is exactly one non-zero digit before the radix point

(c) The mantissa is zero

(d) The sign bit is zero

14. In IEEE 754 single precision format, the bias value used for the exponent is:

(a) 63

(b) 127

(c) 255

(d) 1023

15. The number of bits used for the mantissa in IEEE 754 double precision format is:

(a) 23

(b) 32

(c) 52

(d) 64

Answer Key

Q. No.	Answer	Q. No.	Answer
1	(b) Stack	2	(d) 0-address
3	(c) Program Counter (PC)	4	(b) Hold the instruction currently being executed
5	(b) Immediate	6	(b) Register Indirect Addressing
7	(b) Indexed	8	(b) Relative Addressing
9	(c) Incremented after being used to access memory	10	(d) Grey Code
11	(a) −128 to +127	12	(c) 2's Complement
13	(b) There is exactly one non-zero digit before the radix point	14	(b) 127
15	(c) 52		

Broad / 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.

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

3. What is an addressing mode? Explain any five addressing modes with suitable examples.

4. Differentiate between direct addressing and indirect addressing. Explain with examples how the effective address is calculated in each case.

5. Explain indexed addressing and base register addressing. How are they useful in accessing arrays and in program relocation respectively?

6. Explain the sign-magnitude, 1's complement, and 2's complement representations of signed integers with an example for each. Why is 2's complement preferred in modern computers?

7. What is normalization in floating point representation? Explain with an example how a given decimal number is converted into normalized binary floating point form.

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

9. 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.

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