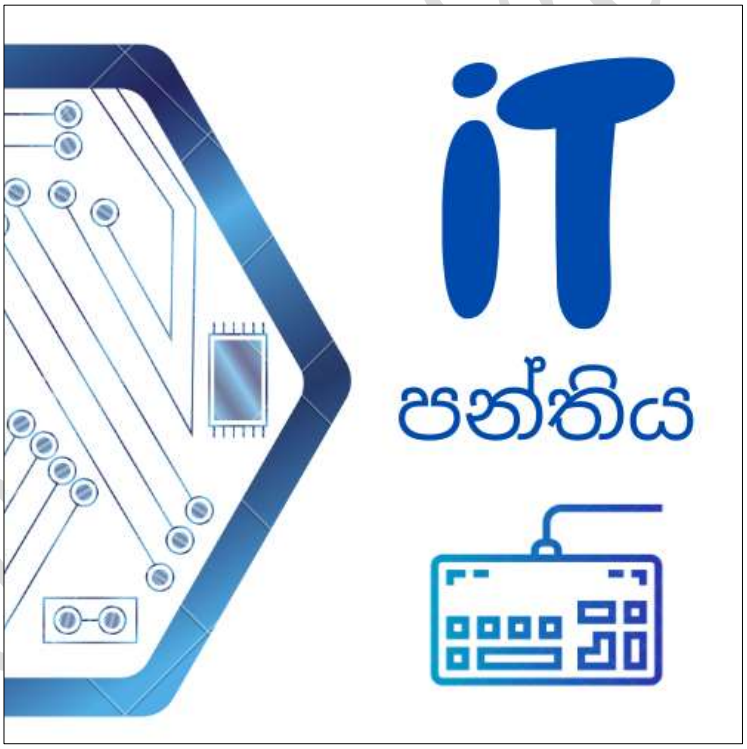
2025 MODEL PAPER 04

MCQ - Analyzation



Model Paper 04 – MCQ Marking Scheme

1) 4	11) 3	21) 3	31) 2	41) 5
2) 2	12) 2	22) 3	32) 4	42) 3
3) 3	13) 4	23) 1	33) 3	43) 4
4) 5	14) 4	24) 4	34) 2	44) 2
5) 2	15) 2	25) 1	35) 2	45) 2
6) 5	16) 3	26) 3	36) 3	46) 2
7) 1	17) 4	27) 1	37) 2	47) 5
8) 4	18) 2	28) 1	38) 2	48) 4
9) 1	19) 1	29) 3	39) 2	49) 3
10) 1	20) 4	30) 1	40) 3	50) 4

1. **Altair 8800 → (4)**
   The 1975 MITS Altair 8800 kicked off the microcomputer era and sold thousands as a kit—commonly cited as the first **commercially successful** personal computer. IBM System/360 was a mainframe; Apple I had tiny sales; PET and IBM PC came later.

---

2. **Vint Cerf → (2)**
   Vint Cerf (with Bob Kahn) designed TCP/IP and is widely called the "Father of the Internet."

---

3. **C → (3)**
   C was created in the early 1970s and Unix was rewritten in C; it became the foundation for modern OS kernels and system software.

---

4. **Addition of −20 and +26 (8-bit two's complement) → (5) 00000110**

   - +26 = `00011010`
   - +20 = `00010100` ⇒ invert → `11101011`, +1 → **−20** = `11101100`
   - Add:
   - `11101100`

- 00011010
  = 100000110 (9 bits; discard the leftmost carry)
  → 00000110

- `00000110` = **6**, which matches −20 + 26.

---

5. **Addition of +25 and −15 (8-bit two's complement) → (2) 00001010**

- +25 = `00011001`
- +15 = `00001111` ⇒ invert → `11110000`, +1 → **−15** = `11110001`
- Add:
- `00011001`

- 11110001
  = 1 00001010 (discard the carry)
  → 00001010

- `00001010` = **10**, which is 25 − 15.

---

6. **BCD of decimal 430 → (5) 0100 0011 0000**

- BCD encodes each digit separately:
  - 4 → `0100`
  - 3 → `0011`
  - 0 → `0000`
- Concatenate: `0100 0011 0000`

---

7. **has no carry-in → (1)**
   A half-adder adds two single bits (A, B) producing Sum and Carry-out, but it **doesn't** accept a carry-in input—that's a full-adder.

---

8. **−127 in 8-bit two's complement → (4) 10000001**

- +127 = `01111111`
- Two's complement (invert + add 1):
  - invert → `10000000`
  - +1 → `10000001`
- So −127 = `10000001`.

---

9. **(1) A + AB = A**
   This is the Boolean **absorption law**. Adding a term that already contains A (AB) to A doesn't change A.

10. **(4) A**

$$A \cdot (A + B) = A \cdot A + A \cdot B = A + AB = A \text{ (absorption).}$$

---

11. **(3) Round Robin**
Time-sharing OSes aim for fairness and quick response among many interactive users, so the CPU is preempted at fixed quanta and given to the next ready task; that policy is exactly Round Robin, whereas FCFS/SJN/Priority can starve jobs or are non-preemptive by default, and Multilevel Queue is about class separation, not strict time-slice fairness.

---

12. **(2) To create a new process**
In Unix-like systems, `fork()` duplicates the calling process to produce a child with a new PID and a copy-on-write address space; it does not terminate a process, allocate memory, or handle I/O (other calls do those), and context switching is the scheduler's job, not `fork()`'s purpose.

---

13. **(4) Archived**
Canonical process states are typically New, Ready, Running, Waiting/Blocked, and Terminated; "Archived" is not part of standard OS state models, so among the options it is the one that is **not** a valid process state.

---

14. **(4) To translate logical (virtual) addresses to physical addresses**
In virtual memory, the page table maps each virtual (logical) page number to a physical frame number (plus status bits), enabling the MMU to translate CPU-generated addresses; it neither tracks free memory nor manages disk files, and mapping physical→logical is the reverse of what is needed.

---

15. **(2) Non-preemptive scheduling does not allow process interruption.**
The core distinction is that preemptive schedulers can forcibly interrupt a running process (e.g., by timer) to give the CPU to another, while non-preemptive ones run a process until it blocks or finishes; statements about priorities or "runs indefinitely" don't capture this defining difference.

---

16. **(3) Session establishment**
The Transport layer (TCP/UDP) handles segmentation/reassembly, flow control, end-to-end error control, and multiplexing/demultiplexing ports; session establishment/synchronization is the role of the Session layer in the OSI model, so it is **not** a Transport-layer function.

---

17. **(4) CSMA/CD**
Of the choices, only CSMA/CD is a Data Link–layer MAC protocol used by classic Ethernet; it governs medium access and collision handling, while Ethernet frames themselves include preamble and FCS for synchronization/error detection—TCP (Transport), IP (Network), ARP (auxiliary Network/LL), and DNS (Application) are not Data Link MAC mechanisms.

18. **(2) media access control sublayer**
The MAC sublayer of the Data Link layer implements medium-dependent access rules (e.g., CSMA/CD for Ethernet, CSMA/CA for Wi-Fi) and framing tied to the physical medium; the LLC sublayer offers medium-independent services above MAC.

---

19. **(1) logical link control sublayer**
Automatic repeat request (ARQ) and related link-level error/flow control are LLC responsibilities that sit above the MAC's medium-access duties; MAC governs who transmits when, while LLC handles reliability services such as acknowledgments and retransmissions.

---

20. **(4) CSMA/CD & CSMA/CA**
Multiple access (MAC) protocols are the rules that let many nodes share the same medium; classic wired Ethernet uses **CSMA/CD** (carrier sense multiple access with collision detection) and Wi-Fi uses **CSMA/CA** (collision avoidance), so the option listing those MAC schemes is the correct one, whereas HDLC is a link protocol but not a multiple-access method and ADSL is a physical-layer access technology, not a MAC.

---

21. **(3) Network Layer**
In the OSI model the **Network layer** (Layer 3) is responsible for routing, i.e., selecting the best logical path for packets to traverse interconnected networks; Transport handles end-to-end delivery on a chosen path, and Session/Presentation/Application are above routing.

---

22. **(3) Both IPv6 and IPv4**
DHCP exists as DHCPv4 and DHCPv6 to automatically provide IP configuration (address, gateway, DNS, etc.) for both IPv4 and IPv6 hosts; it is not specific to only one version of IP and IPv5 is not a deployed Internet protocol.

---

23. **(1) Session Layer**
The **Session layer** manages dialogues between systems, including establishing, maintaining, and synchronizing sessions (checkpoints and turn-taking) to control how data is exchanged; these are not routing (Network) or end-to-end transport functions.

---

24. **(4) 192.168.10.3**
- /30 ⇒ block size = 4 → subnet range: `192.168.10.0 – 192.168.10.3`
- Host usable: .1, .2
- **Broadcast:** `192.168.10.3`

25. **(1) 192.168.50.1**

- /27 ⇒ block size = 32 → subnet range: `192.168.50.0 - 192.168.50.31`
- First usable host = `192.168.50.1`
- **Default gateway:** `192.168.50.1`

---

26. **(3) 255.255.255.192**

- Need ≥60 usable addresses.
- /26 ⇒ 64 addresses, 62 usable → satisfies requirement.
- **Subnet mask:** `255.255.255.192`

---

27. **(1) 30 feet**
Bluetooth (esp. common Class-2 devices) typically supports a short-range personal-area link of about 10 meters ≈ **30 ft**; 30 yards and beyond are outside the normal operating range, and "miles" is orders of magnitude too far.

---

28. **(1) Bits, bytes, fields, records, files and databases**
The storage/data hierarchy goes from the smallest information unit to the largest container: **bit → byte → field → record → file → database**; other options scramble the order.

---

29. **(3) Encryption**
"Encoding or scrambling data" so it's unintelligible to eavesdroppers during transmission is precisely **encryption**; protection/detection are broader security goals, and decryption is the inverse process.

---

30. **(1) Internet explorer**
Internet Explorer is proprietary Microsoft software, whereas **Fedora Linux, OpenOffice, and the Apache HTTP Server** are well-known open-source projects; hence IE is the item that is **not** open source.

---

31. **(2) This violation involves unauthorized modification of data**
Security "integrity" means data remains correct and unaltered; a breach of integrity occurs when an attacker **modifies** data without authorization (not merely reading or destroying it).

---

32. **(4)** "Theft of service" refers to consuming computing/network resources (CPU time, bandwidth, storage, etc.) without permission or payment, i.e., **unauthorized use of resources** rather than reading, modifying, or deleting data.

33. **(3) It is a rogue program which tricks users**
A **Trojan horse** masquerades as a legitimate program to deceive users into running it so it can perform malicious actions; it's not an encryption method, a brute-force algorithm, or a "user."

---

34. **(2) It is a hole in software left by designer**
A **trap door/backdoor** is a hidden entry point intentionally (or negligently) left in software that bypasses normal authentication or checks; it's neither a Trojan nor a virus, and the movie reference isn't a definition.

---

35. **(2) Virus is a standalone program**
A classic computer virus is **not** standalone; it attaches itself to a host file/program and propagates when that host runs. Viruses can indeed destroy/modify data, are code embedded in legitimate programs, and while they can sometimes evade tools, "cannot be detected" is not a defining characteristic—therefore "standalone program" is the incorrect (not-a-characteristic) option.

---

36. **(3) Program Counter (PC)**
In the von Neumann model, the **Program Counter** holds the memory **address of the next instruction** to fetch; the MAR holds the address being accessed this cycle, the MDR holds the data being transferred, the IR holds the current instruction itself, and the accumulator stores arithmetic/logic results.

---

37. **(2) Instruction Decode**
The fetch–decode–execute cycle works as named: first the CPU **fetches** an instruction from memory, then it **decodes** the fetched bits to determine the operation and operands, and only after that does it **execute** the operation; thus the stage involving decoding the fetched instruction is Instruction Decode.

---

38. **(2) Memory Data Register (MDR)**
The MDR holds the **actual data** being transferred to or from the location in memory specified by the Memory Address Register. The MDR (also called the Memory Buffer Register) holds the **actual data** being read from or written to the memory location whose address is in the MAR; the PC is for next-instruction addresses, the IR stores the current instruction, the AC holds ALU results, and GPRs are general purpose but not specifically for memory transfers.

---

39. **(2) Accumulator (AC)**
Arithmetic and logic operations performed by the ALU typically place their **result** in the **Accumulator** (or a designated result register); the MAR/IR/PC have control and addressing roles, while general-purpose registers are broader storage but the accumulator is the canonical destination for ALU results in the von Neumann architecture.

---

40. **(3) The CPU executes the decoded instruction.**
In the classic fetch–decode–execute cycle, *fetch* gets the instruction from memory, *decode* interpret

opcode/operands, and the *execute* stage actually performs the specified operation using the ALU, registers, and/or memory; storing results or updating the PC are separate effects that occur as part of or immediately after execution, but the essence of this stage is to **execute** the decoded instruction.

---

41. **(5) Memory Address Register (MAR).**
The MAR always holds the **address** of the memory location that the CPU intends to read from or write to; the complementary MDR holds the data being transferred, the PC holds the address of the next instruction, the IR holds the current instruction, and the accumulator holds arithmetic/logic results—therefore the register that contains the currently referenced memory address is the **MAR**.

---

42. **(3)** `SELECT Name FROM Employee INNER JOIN Department ON Employee.DepartmentID = Department.DepartmentID WHERE DepartmentName = 'IT';`
Employee rows only have a `DepartmentID`, while the human-readable `"IT"` label lives in `Department.DepartmentName`; to filter employees who work in IT you must join `Employee` to `Department` on the matching `DepartmentID` and then apply the `WHERE DepartmentName = 'IT'` predicate—queries that compare `DepartmentID` to `'IT'` or reference `DepartmentName` directly from `Employee` are invalid.

---

43. **(4) NOT NULL.**
Preventing a column from containing missing values is exactly what the **NOT NULL** constraint does; `UNIQUE` enforces distinctness, `PRIMARY KEY` implies `UNIQUE` + `NOT NULL` but applies to key columns, `FOREIGN KEY` enforces referential integrity, and `DEFAULT` only supplies a value when one isn't provided—it doesn't forbid nulls by itself.

---

44. **(2)** `SELECT DISTINCT Manager FROM Department INNER JOIN Employee ON Department.DepartmentID = Employee.DepartmentID;`
We want manager names **only for departments that have at least one employee**; an INNER JOIN between `Department` and `Employee` naturally removes departments with zero employees, and `DISTINCT` collapses any duplicates created when a department has multiple employees; (EXISTS/IN variants could also work, but this join with `DISTINCT` is the straightforward, correct formulation).

---

45. **(2) FOREIGN KEY constraint.**
To ensure every `Employee.DepartmentID` points to a valid department row, we use a **FOREIGN KEY** on `Employee(DepartmentID)` that references `Department(DepartmentID)`; primary/unique check value properties on the parent don't enforce cross-table validity from the child, a `CHECK` can't look into another table, and `NOT NULL` only prevents nulls—not invalid IDs.

---

46. **(2)** `SELECT DepartmentName, SUM(Salary) FROM Employee INNER JOIN Department ON Employee.DepartmentID = Department.DepartmentID GROUP BY DepartmentName;`

The total salary **per department** requires grouping by a department identifier and summing salaries. Since `DepartmentName` is in the `Department` table while salaries are in `Employee`, we must **JOIN** the two tables on `DepartmentID` and then `GROUP BY DepartmentName` to get one total per named department; options that refer to `DepartmentName` without a join are invalid, and those that don't group per department return the wrong aggregation.

---

47. (5) None of the mentioned

Calc (page translation):

- Page size = 8 KB ⇒ **offset = 13 bits** (= 8192).
- Virtual address **34816** ⇒ page number = `34816 ÷ 8192 = 4`, offset = `34816 − 4×8192 = 2048`.
- From the page table: **Page 4 is Absent (0) ⇒ no frame** is assigned.
- Therefore, **no physical address exists** for this VA → **None of the mentioned**.

---

48. (4) It will result in a page fault

Calc (timing):

- The question explicitly says "accesses **Page 4**"; the page table marks Page 4 as **Absent (0)**.
- Accessing an absent page triggers a **page fault** (no 100/120/150/200 ns access applies).
- Hence the result is a **page fault**.

---

49. (3) 4

Calc (unused frames):

- Physical memory = 64 KB → frames = `64KB / 8KB = 8` (frame IDs 0–7).
- Present pages and their frames: Page0→`011`(3), Page1→`101`(5), Page3→`110`(6), Page5→`010`(2).
- **Used frames = {2,3,5,6}** → count = 4.
- **Unused frames = 8 − 4 = 4.**

---

50. (4) A page fault occurs

Calc (address 65536):

- VA **65536** is within the 0–131071 range.
- Page number = `65536 ÷ 8192 = 8`, offset = `0`.
- The table has entries only for pages 0–5; Page **8** is not present ⇒ **no frame**.
- Therefore, accessing VA 65536 **causes a page fault**.