# Computer Organization and Architecture

*Exercise and Exam-Oriented Questions with Answers*

**Version 2 — Second Practice Set**

## Chapter 7

## Vector Processing and Array Processor

**Covers**

Very Short Answer Type Questions (VSAQ) — New Set
Short Answer Type Questions (SAQ) — New Set
Long Answer Type Questions (LAQ) — New Set with Full Model Answers

*Note: This set contains a fresh group of questions, distinct from Version 1, for additional practice and revision on the same chapter.*

## Section A — Very Short Answer Type Questions

*(1–2 marks each) — Version 2 set, suitable for quick revision and viva-voce preparation.*

**Q1.** What type of parallelism does vector processing primarily exploit?

**Ans.** Vector processing primarily exploits data-level parallelism — the same operation applied repeatedly to many independent data elements.

**Q2.** What is a vector register?

**Ans.** A vector register is a high-speed register capable of holding an entire vector, or a section of one, so that a pipelined functional unit can be supplied with operands at high speed without repeated main-memory accesses.

**Q3.** What is pipeline 'fill time' in a vector processor?

**Ans.** Pipeline fill time is the initial number of clock cycles a pipelined functional unit takes before its first result emerges, after which one new result is produced every subsequent clock cycle.

**Q4.** Why do very short vectors show a smaller speed advantage on a pipelined vector unit?

**Ans.** Because the fixed pipeline fill and drain time is not well amortised over only a few elements, so this overhead forms a large fraction of the total execution time for short vectors.

**Q5.** What does the vector-length (N) field in a vector instruction specify?

**Ans.** It specifies the number of elements in the vector(s) being processed, telling the control hardware how many times to repeat the pipelined operation and when to stop.

**Q6.** What is the purpose of an address-increment field in some vector instructions?

**Ans.** It specifies the spacing between successive elements in memory, which is needed when elements are not stored in strictly consecutive locations, such as when accessing a row or column of a matrix.

**Q7.** What is the function of the interconnection network in an array processor?

**Ans.** It allows data to be exchanged, shifted, broadcast, or permuted between processing elements whenever an algorithm requires one processing element to use a value produced by another.

**Q8.** What is the purpose of a masking mechanism in an SIMD array processor?

**Ans.** It allows individual processing elements to be selectively enabled or disabled for a given instruction, so that data-dependent (conditional) operations can be expressed even though all active PEs execute the same broadcast instruction.

**Q9.** How does an SIMD array processor differ from an SISD (single processor) machine?

**Ans.** An SISD machine has one control unit and one processing unit executing one instruction on one data item at a time, whereas an SIMD array processor has one control unit broadcasting a single instruction simultaneously to many processing elements, each operating on its own data.

**Q10.** Name one historical example of a true SIMD array processor other than ILLIAC IV.

**Ans.** The Connection Machine (CM-1/CM-2) is another historical example of a true SIMD array processor.

**Q11.** What kind of computing problems are best suited to array processors?

**Ans.** Problems with a large, regular, uniform data-parallel structure, such as image processing, matrix/grid computations, and finite-difference simulations, are best suited to array processors.

**Q12.** What is the fundamental difference between an SIMD array processor and an MIMD-oriented processor array?

**Ans.** In an SIMD array processor all processing elements share one control unit and execute the same instruction, whereas in an MIMD-oriented processor array each processing node has its own independent control unit and can execute a different instruction stream.

## Section B — Short Answer Type Questions

*(5 marks each) — Version 2 set, expected answer length: 6–10 lines.*

**Q1.** Explain, with a diagram, how interleaved memory supports vector processing.

***Answer:***

A pipelined vector functional unit requires a new operand on every clock cycle once it is full, but a single memory module cannot usually supply data that fast on its own because of memory access/cycle time. To solve this, main memory is divided into several independent modules, and successive elements of a vector — which are normally stored in consecutive addresses — are distributed cyclically across these modules (element 0 in module 0, element 1 in module 1, and so on). Because each module can be accessed independently, the memory system as a whole can supply one element per clock cycle to the pipeline by accessing a different module on each cycle, even though any individual module is slower than that.

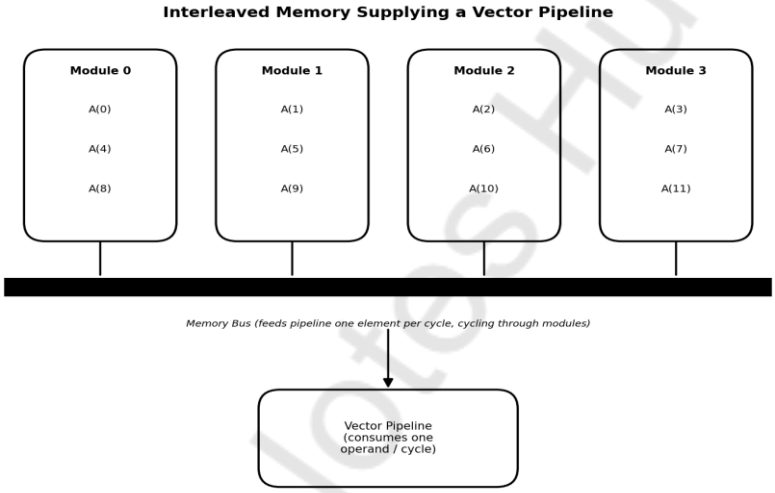


*Fig. — Interleaved memory modules feeding a vector pipeline*

**Q2.** With the help of a diagram, explain how chaining overlaps two dependent vector operations.

***Answer:***

Suppose a program needs to compute D = (A + B) times C as vector operations. Without chaining, the vector-add instruction would have to complete entirely — producing and storing the full result vector (A + B) — before the vector-multiply instruction could begin. With chaining, as soon as the ADD pipeline produces the first element of (A + B), that element is forwarded directly into the MULTIPLY pipeline, without waiting for the remaining elements of the sum to be computed. The two pipelines therefore operate concurrently on the vector, one stage behind the other, and the total time for the combined operation approaches the time for a single pipelined operation plus a small additional startup delay, rather than the sum of two full separate operation times.

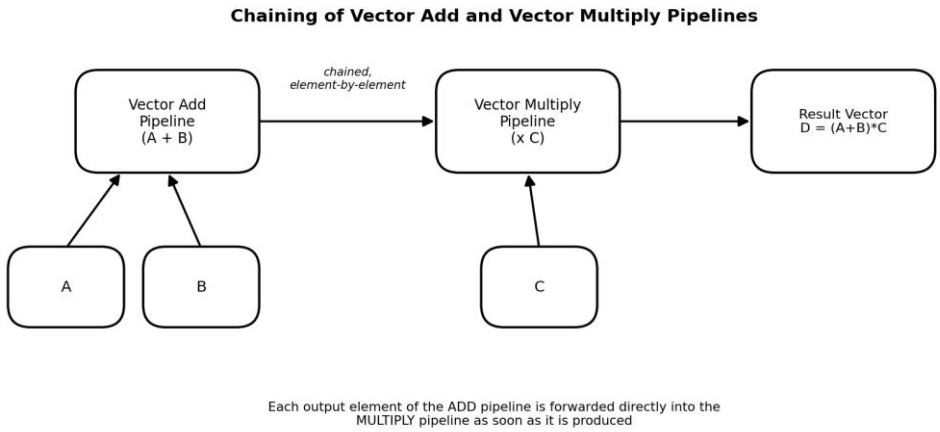


*Fig. — Chaining of the vector-add and vector-multiply pipelines*

**Q3.** What are vector registers? Explain their advantage over a purely memory-to-memory vector organisation.

***Answer:***

Vector registers are high-speed registers, provided within the processor, each capable of holding an entire vector or a fixed-size section of one. In a register-to-register vector organisation (as used in machines such as the Cray-1), a vector instruction reads its operands from vector registers and writes its result to a vector register, rather than going directly to and from main memory for every element. Because register access is much faster than memory access, and because a loaded vector can be reused by several subsequent instructions without being fetched from memory again, this organisation reduces memory traffic and allows the pipelined arithmetic units to be supplied with operands at the full rate they can consume them, improving overall performance compared with a purely memory-to-memory approach.

**Q4.** List and briefly explain the fields typically found in a vector processing instruction.

***Answer:***

A typical vector instruction contains: an operation code (opcode) specifying the action to be performed; the base address of the first source vector; the base address of the second source vector, where the operation needs two operand vectors; the base address of the destination vector, where the result is to be written; the vector length, indicating how many elements are to be processed; and, optionally, an address increment, used when successive elements are not stored in strictly consecutive memory locations. Together, these fields let one instruction describe an entire repetitive vector operation that a scalar machine would otherwise need an explicit loop to express.

**Q5.** Explain why an array processor is classified as SIMD, and briefly describe how a single instruction can still allow conditional (data-dependent) behaviour.

***Answer:***

An array processor is classified as SIMD (Single Instruction stream, Multiple Data stream) because one control unit fetches and decodes a single instruction stream and broadcasts each instruction to all processing elements, which execute it simultaneously but on different data held in their own local memories. Since every active processing element must execute the same instruction, conditional behaviour that differs from one data element to another is handled through a masking mechanism: a mask register in each processing element records whether that PE should be active for a given instruction, based on the outcome of an earlier

comparison; PEs whose mask bit is off simply do not modify their state for that instruction, allowing the array as a whole to emulate data-dependent branching even though it follows a single instruction stream.

**Q6.** Explain the role of local memory in the processing elements of an SIMD array processor.

***Answer:***

Each processing element in an SIMD array processor is typically provided with its own local memory module, which holds the particular data element(s) that PE is responsible for processing. Because every PE has independent access to its own memory, all processing elements can fetch and store their operands in the same clock cycle without competing for a shared memory port, which is essential for achieving true simultaneous execution across the array. The interconnection network is then used only when data must move between PEs, rather than for every ordinary memory access.

**Q7.** What is an associative (content-addressable) array processor, and in what type of application does it have an advantage?

***Answer:***

An associative array processor is a variant of the SIMD array organisation in which each processing element's local memory is content-addressable: instead of retrieving data by specifying a memory address, the control unit broadcasts a search pattern, and every processing element compares this pattern against the data it holds, simultaneously, to determine a match. This gives such processors a natural advantage in applications involving searching, sorting, and pattern-matching across large, uniform data sets, since a single broadcast comparison can be evaluated by every element of the data set in parallel, rather than requiring an explicit address-based search over the data one item at a time.

# Section C — Long Answer Type Questions

*(10–15 marks each) — Version 2 set, full model answers suitable for descriptive/university examinations.*

**Q1.** Explain chaining and interleaved memory as two supporting techniques of vector processing. Illustrate each with a diagram.

***Answer:***

Vector processors rely not only on pipelined arithmetic units but also on supporting techniques that keep those pipelines supplied with data and allow dependent operations to overlap. Two important such techniques are interleaved memory and chaining.

## Interleaved Memory

A pipelined vector functional unit, once full, requires a new pair of operands on every clock cycle. A single memory module, however, typically has an access or cycle time longer than one clock cycle, and so cannot by itself supply data fast enough. To overcome this, main memory is organised into several independent modules, and successive elements of a vector — which usually occupy consecutive addresses — are distributed cyclically across the modules: element 0 goes to module 0, element 1 to module 1, and so on, wrapping back to module 0 after all modules have been used once. Because each module operates independently and can be accessed in an overlapped manner, the memory system as a whole can deliver one vector element per clock cycle to the pipeline, even though each individual module is slower than that.

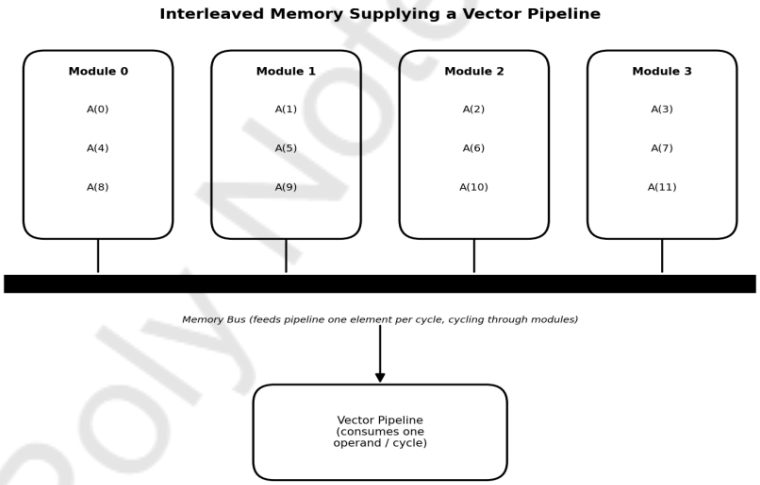


*Fig. — Interleaved memory modules feeding a vector pipeline*

## Chaining

Chaining addresses the case where the result of one vector operation is itself needed as an input to a second vector operation — for example, computing D = (A + B) times C. Without chaining, the entire vector-add operation (A + B) would need to complete and be written back before the vector-multiply operation could begin, wasting time. With chaining, each element of the sum (A + B) is forwarded directly from the output of the add pipeline into the input of the multiply pipeline as soon as that individual element is produced, rather than waiting for the whole vector to be finished. The add and multiply pipelines thus work concurrently, one

slightly behind the other, and the combined operation completes in close to the time of a single pipelined pass rather than the sum of two separate full passes.

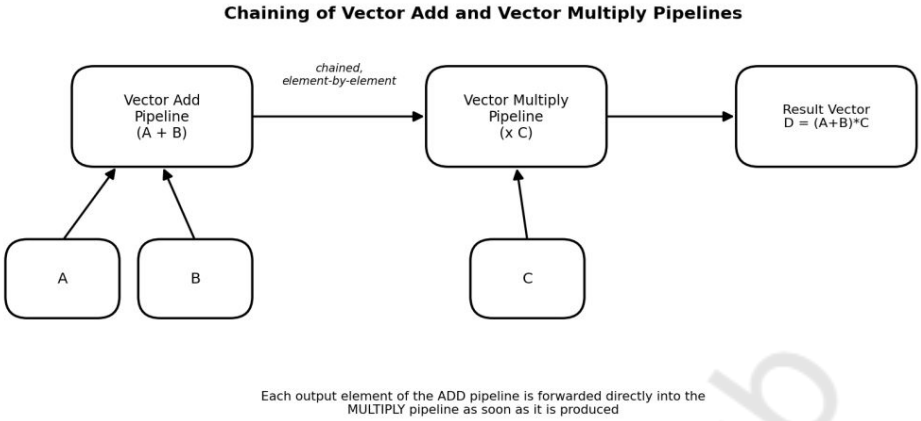


*Fig. — Chaining of the vector-add and vector-multiply pipelines*

Both techniques address different bottlenecks in vector processing — interleaved memory ensures the pipeline is fed with operands fast enough, while chaining ensures that a sequence of data-dependent vector instructions does not have to be executed in strict, non-overlapped sequence — and together they allow a vector processor to sustain close to one useful result per clock cycle across a realistic program consisting of several dependent vector operations.

**Q2.** Explain vector registers and the vector instruction format used in register-based vector machines such as the Cray-1. Compare this with a memory-to-memory vector organisation.

***Answer:***

A register-based vector machine provides a set of high-speed vector registers, each capable of holding an entire vector or a fixed-length section of one, in addition to the usual scalar registers. Rather than reading and writing every operand directly to and from main memory, a vector arithmetic instruction in such a machine reads its source operands from vector registers and writes its result into another vector register. Separate vector load and vector store instructions are used to transfer data between main memory and the vector registers, so that a vector, once loaded, can be used by several arithmetic instructions without being re-fetched from memory each time.

## Instruction Format in Register-Based Machines

A vector arithmetic instruction in a register-based design typically specifies an operation code together with the identifiers of the vector registers holding the source operand(s) and the vector register that will receive the result; the length and addressing details of the vector are already associated with the register (having been established when the vector was loaded), so they need not be repeated in every arithmetic instruction. The separate vector load/store instructions, by contrast, do specify a memory base address, an address increment (if elements are not contiguous), and the vector length, since it is at load/store time that the mapping between memory and the vector register is established.

## Comparison with Memory-to-Memory Organisation

In a memory-to-memory vector organisation, each vector arithmetic instruction directly specifies the base addresses of the source vectors in memory, the base address of the destination vector, and the vector length, and the pipelined unit reads and writes memory directly for every element of every instruction.

| **Aspect** | **Register-Based (e.g., Cray-1)** | **Memory-to-Memory** |
|---|---|---|
| Operand source/destination for arithmetic instructions | Vector registers | Main memory directly |
| Memory traffic | Lower — a loaded vector can be reused across several instructions | Higher — every instruction accesses memory for every element |
| Instruction fields for arithmetic ops | Opcode + register identifiers (length/address already tied to register) | Opcode + source/destination addresses + vector length |
| Extra instructions required | Separate vector load/store instructions needed | Not required — direct memory access built into each instruction |

In general, the register-based approach trades a small amount of extra instruction overhead (explicit load/store instructions) for a significant reduction in memory bandwidth demand, which is usually the more valuable trade-off in high-performance vector machines, since memory bandwidth is typically the harder resource to scale.

**Q3.** Discuss, in detail, the role of the control unit, processing elements, local memory, interconnection network, and masking mechanism in an SIMD array processor, explaining how they work together to execute a program.

***Answer:***

An SIMD array processor executes a program by having a single control unit direct a synchronous array of processing elements, all executing the same instruction at the same time on different data. Each of the major components plays a distinct role in making this possible.

## Control Unit

The control unit fetches instructions from the program memory, decodes each one, and broadcasts the decoded instruction, together with any common scalar operands, to every processing element over a control/broadcast bus. It also holds and updates the program counter and manages overall program flow, including conditional branches based on aggregate conditions across the array.

## Processing Elements

Each processing element contains its own arithmetic-logic unit and register set. On receiving a broadcast instruction from the control unit, every active processing element executes that same instruction, in lock-step with all the others, but applies it to the data held in its own registers or local memory, so that N processing elements can complete N independent instances of the same operation in the same amount of time a single processor would need for just one instance.

## Local Memory

Each processing element is typically given its own local memory module, holding the specific data element(s) it is responsible for. Because these memories are independent of one another, every processing element can read and write its own operands in the same clock cycle without contending for a shared memory port, which is essential for genuinely simultaneous execution across the whole array.

## Interconnection Network

Many algorithms require a processing element to use a value produced by, or belonging to, a different processing element — for example, exchanging boundary values in a grid computation. The interconnection network provides the paths needed to shift, broadcast, or otherwise route data between processing elements when such exchanges are required, since local memory alone cannot supply data that logically belongs to another PE.

## Masking Mechanism

Because every active processing element must execute the same broadcast instruction, data-dependent (conditional) behaviour is handled through masking: each PE maintains a mask (enable) bit, typically set according to the result of an earlier comparison specific to that PE's own data. When an instruction is broadcast, only PEs whose mask bit is set actually update their state; the others effectively execute the instruction as a no-operation. This allows the array, as a whole, to emulate per-element conditional branching despite following a single, shared instruction stream.

Working together, these components allow an SIMD array processor to apply a program's operations, instruction by instruction, across an entire array of data simultaneously: the control unit supplies the common instruction stream, the processing elements and their local memories provide independent, parallel execution and storage, the interconnection network handles the comparatively rare need for inter-element communication, and the masking mechanism reconciles the single instruction stream with the need for element-specific, data-dependent behaviour.

**Q4.** Explain, with justification, why attached array processors were historically an attractive design choice, and discuss their limitations compared with a true SIMD array processor.

***Answer:***

An attached array processor is an auxiliary, high-speed processing unit connected to an existing general-purpose host computer, used specifically to accelerate vector- or array-intensive numerical computation, while the host continues to manage overall program execution, the operating system, and input/output. Historically, this design was attractive for several practical reasons.

## Advantages

- Cost-effectiveness: rather than designing and building an entirely new stand-alone high-performance computer, an existing general-purpose host could be upgraded relatively inexpensively by adding a specialised attached unit only for the numerically intensive part of the workload.
- Reuse of existing infrastructure: the host's operating system, compilers, I/O subsystem, and general-purpose software environment could continue to be used unchanged, with only the performance-critical vector computations being off-loaded.
- Incremental adoption: organisations could add vector-processing capability to their computing environment gradually, attaching a processor to accelerate specific applications without committing to replacing their entire computing infrastructure.

## Limitations Compared with a True SIMD Array Processor

- Data transfer overhead: data must be transferred between the host's memory and the attached processor before and after each accelerated computation, which adds overhead that a true, stand-alone array processor (where the array organisation is the main processing structure) does not incur to the same extent.
- Limited integration: because the attached processor is a separate unit connected through an I/O channel or interface, it cannot participate as tightly in the host's overall control flow and scheduling as the processing elements of a true SIMD array processor, which are built into the machine's primary execution path from the outset.
- Performance ceiling set by the interface: the speed at which data can be moved across the host-to-attached-processor connection can become a bottleneck, limiting the achievable speed-up regardless of how fast the attached unit's own arithmetic pipelines are.

In summary, an attached array processor offers a practical, lower-cost path to accelerating vector computation on an existing host, but a true, stand-alone SIMD array processor — where the array organisation is the primary architecture of the machine rather than an add-on — avoids the data-transfer and integration overheads inherent in the attached approach, at the cost of requiring a dedicated, purpose-built machine.