Computer Organization and Architecture

Study Notes

Chapter 7

Vector Processing and Array Processor

Topics Covered

7.1 Concept of Vector Processing. Techniques Used in Vector Processing
7.2 Speed Advantage of Vector Processing. Vector Processing Instruction Format
7.3 Concept of Array Processor
7.4 Different Types of Array Processors

For Engineering Students

7.1 Concept of Vector Processing. Techniques Used in Vector Processing

7.1.1 What is Vector Processing?

Many scientific and engineering problems — such as weather modelling, structural analysis, weapon research, medical diagnosis, aerodynamics and seismic data processing — require arithmetic operations to be repeated over large sets of numbers that are organised as vectors (one-dimensional arrays) or matrices (two-dimensional arrays). A conventional scalar computer processes one pair of data elements at a time inside a single instruction (for example, C = A + B is a scalar operation on single values). This means that if the same operation has to be applied to, say, 100 pairs of numbers, the scalar machine must execute the instruction 100 times inside a program loop.

A vector processor is a computer that is specially organised to process entire vectors (or matrices) of numbers as a single unit of work, using one instruction. Instead of writing a loop that adds each pair of elements separately, a single vector instruction such as C(1:100) = A(1:100) + B(1:100) directs the hardware to add all 100 corresponding element pairs. The vector processor therefore has hardware — pipelines, multiple functional units, or arrays of processing elements — that can operate on many data elements concurrently or in a highly overlapped fashion.

Vector processing is thus defined as a mode of computation in which a single instruction operates simultaneously (or in a heavily pipelined, overlapped manner) on an ordered set of data elements — a vector — rather than on a single scalar element. It forms the basis of high-performance and supercomputer architecture and falls under the broader category of data-level parallelism.

7.1.2 Why Vectors Arise in Computation

Vectors and matrices are natural mathematical structures in scientific computation:

- A vector is a one-dimensional array of n elements: V = (v1, v2, v3, ..., vn).
- A matrix is a two-dimensional array of m x n elements, which can itself be treated as a set of row or column vectors.
- Operations such as dot product, matrix multiplication, solving linear equations, Fourier transforms, and numerical integration all consist of the same arithmetic operation repeated over many elements — exactly the pattern that vector hardware is built to exploit.

7.1.3 Techniques Used in Vector Processing

Vector processing achieves high throughput mainly through the following hardware and organisational techniques:

(a) Pipeline Processing

The arithmetic unit (adder, multiplier) is divided into a number of independent sub-units called segments or stages, each performing one small part of the overall operation (for example, floating-point addition may be split into: compare exponents, align mantissas, add mantissas, normalise result). Successive pairs of vector elements are fed into the pipeline on every clock cycle. While one pair is being normalised in the last stage, the next pair is being added in an earlier stage, and a further pair is being aligned even earlier — many element pairs are therefore in different stages of computation at the same time. Once the pipeline is full, one result

emerges every clock cycle instead of one result every several clock cycles, giving a large speed-up for long vectors.

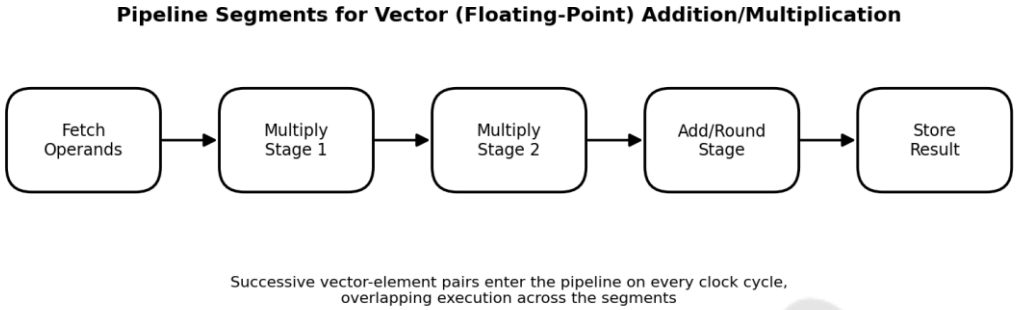

Fig. 7.1 — Pipeline segments used for vector addition/multiplication

(b) Use of Multiple/Parallel Functional Units

Instead of, or in addition to, pipelining, a vector processor may provide several identical functional (arithmetic) units working in parallel. Independent pairs of vector elements are distributed among these units so that several element operations complete during the same time interval.

(c) Vector Registers

High-speed vector registers are provided to hold an entire vector (or a segment/section of it) so that the pipelined functional units can be fed operands at very high speed without repeatedly accessing main memory for every element. Machines such as the Cray-1 use this register-to-register approach.

(d) Memory Interleaving / Access Techniques

Because a pipeline needs a new operand every clock cycle, main memory itself is organised into multiple interleaved memory modules so that successive vector elements (which usually lie in consecutive memory locations) can be fetched from different modules in an overlapped manner, matching the rate demanded by the pipeline.

(e) Chaining

When the result vector produced by one pipelined functional unit is itself an operand needed by another vector instruction, the output of the first pipeline can be fed directly ("chained") into the input of the second pipeline as soon as each element becomes available, rather than waiting for the entire first vector operation to finish and be written back. This overlaps successive dependent vector instructions and further increases throughput.

7.1.4 Summary Points

- Vector processing applies one instruction to an entire ordered set (vector) of data elements.
- It exploits data-level parallelism found in scientific/engineering computation.
- Main techniques: pipelining of arithmetic units, parallel functional units, vector registers, interleaved memory, and chaining.
- It underlies the design of supercomputers and vector/array-oriented architectures.

7.2 Speed Advantage of Vector Processing. Vector Processing Instruction Format

7.2.1 Speed Advantage of Vector Processing

The performance gain of vector processing over conventional scalar processing comes chiefly from removing the overhead that a program loop imposes when the same operation is repeated for every element.

Consider adding two vectors of n elements, C(I) = A(I) + B(I) for I = 1 to n, on a scalar (non-pipelined) machine using a program loop. Each pass of the loop must, in addition to the actual addition, perform loop-control work: incrementing the index, comparing the index against the limit n, computing the effective addresses of A(I), B(I) and C(I), branching back to the top of the loop, and fetching/decoding the same instructions again and again. This control overhead is repeated n times even though the arithmetic operation being carried out is identical every time.

A vector instruction, by contrast, specifies the entire operation — the two source vectors, the destination vector and the vector length n — in a single instruction. The instruction is fetched and decoded only once; the vector-length count and address generation are handled automatically by dedicated hardware (often called a vector-length and address-generation unit), and the pipelined arithmetic unit then streams the n element operations through, producing one result every clock cycle once the pipeline is filled. The loop-control overhead, and the repeated instruction fetch/decode overhead, are essentially eliminated.

Speed-up can be summarised as arising from two sources:

1. Reduction of instruction-processing overhead: one instruction fetch/decode replaces n fetch/decode cycles of a scalar loop.
2. Pipeline overlap of arithmetic: after an initial pipeline fill time, one result is produced every clock cycle, so the effective time per element approaches a single clock cycle instead of the many clock cycles a non-pipelined scalar operation would need.

For long vectors, the fixed pipeline start-up (fill) time is amortised over many elements, so the average execution time per element approaches the cycle time of the pipeline, giving a substantial overall speed-up — this is why vector processors show their greatest advantage for large vector lengths, and comparatively little advantage for very short vectors (where the pipeline fill/drain overhead dominates).

7.2.2 Vector Processing Instruction Format

A vector instruction differs from a scalar instruction in that it must identify, in addition to the operation to be performed, the location of entire vectors (not single operands) and how many elements they contain. A typical vector instruction format used in memory-to-memory vector organisations includes the following fields:

- Operation Code (Opcode): specifies the operation to be performed, e.g. vector add, vector multiply, vector load, vector compare.
- Base Address of Source Vector 1 (A1): the starting memory address of the first source vector.
- Base Address of Source Vector 2 (A2): the starting memory address of the second source vector (omitted for a single-operand operation).

- Base Address of Destination Vector (A3): the starting memory address where the result vector is to be stored.
- Vector Length (N): the number of elements in the vector(s) being processed; this tells the control hardware how many times to repeat the pipelined operation and when to stop.
- Address Increment (optional): the address increment between successive elements, used when elements are not stored in strictly consecutive locations (useful for accessing a row or column of a matrix).

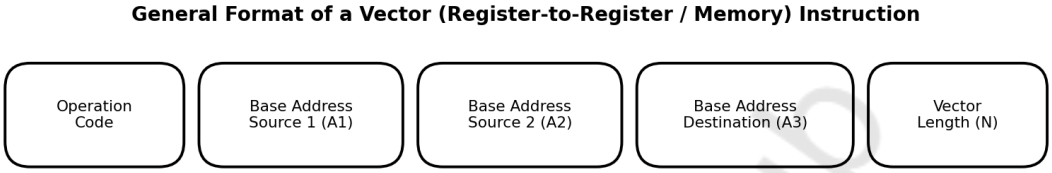

Fig. 7.2 — General fields of a vector instruction

In register-based vector machines (such as the Cray-1 design), the instruction instead names vector registers (which already hold the base address, length and increment information for the currently loaded vector) rather than raw memory addresses; separate vector load/store instructions are used to move data between memory and the vector registers.

Regardless of the exact format, the essential point is that a single vector instruction encodes an entire repetitive operation — operation, operand locations, and count — that a scalar architecture would otherwise require an explicit multi-instruction loop to express.

7.2.3 Summary Points

- Vector instructions remove per-element loop-control and instruction fetch/decode overhead.
- Pipelining allows one result per clock cycle after the pipeline fills, giving large speed-up for long vectors.
- A vector instruction format typically carries: opcode, source address(es), destination address, vector length, and (optionally) address increment.

7.3 Concept of Array Processor

7.3.1 What is an Array Processor?

An array processor is a computer that achieves high-speed computation by employing a large number of identical processing elements (PEs), all working under the supervision of a single control unit, so that the same instruction is applied simultaneously (in true parallel, not merely pipelined overlap) to many different data elements. Where a pipelined vector processor overlaps successive element operations in time within one arithmetic pipeline, an array processor duplicates the actual arithmetic hardware itself, spreading the elements of a vector or matrix across many processing elements that compute in the same clock cycle.

This organisation is classified, under Flynn's classification of computer architectures, as SIMD — Single Instruction stream, Multiple Data stream. A single control unit fetches and decodes one instruction and broadcasts it to all processing elements; each processing element executes that same instruction, but on the different piece of data held in its own local memory or register.

7.3.2 Basic Organisation

The general organisation of an SIMD array processor consists of:

- Control Unit (CU): fetches and decodes instructions from the program, generates control signals, and broadcasts the decoded instruction (and any scalar/common operands) to every processing element over a control/broadcast bus.
- Array of Processing Elements (PE0, PE1, ..., PE(n-1)): each PE contains an ALU, registers and (usually) its own local memory. All PEs execute the same broadcast instruction in lock-step, but each operates on the data present in its own local memory.
- Local Memory: each PE typically has its own memory module holding the particular data elements (e.g., one element of each vector) assigned to it, allowing simultaneous, conflict-free memory access by all PEs.
- Interconnection Network: allows data to be exchanged between processing elements (for example, to shift, broadcast, or permute data among PEs) when an algorithm requires a PE to use a value computed by a neighbouring PE.
- A Masking Mechanism (in most designs): allows individual PEs to be selectively enabled or disabled for a given instruction, so that conditional (data-dependent) operations can still be expressed even though all active PEs execute the same instruction stream.

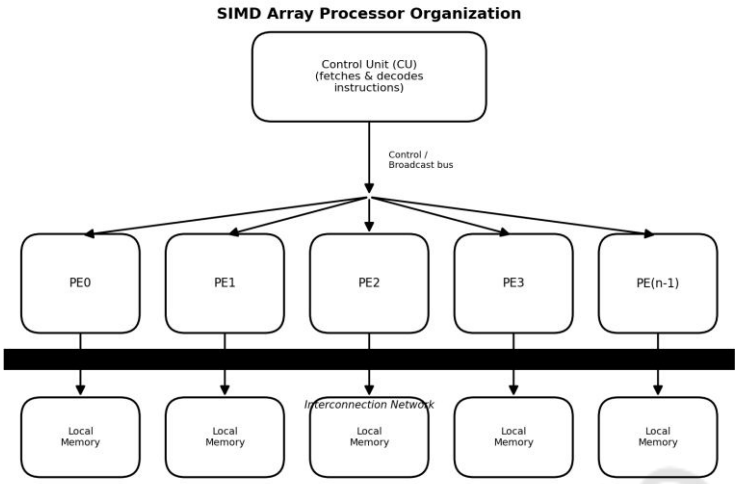

Fig. 7.3 — General organisation of an SIMD array processor

Because the same instruction is applied to N elements at the same instant across N processing elements, an array of N PEs can, ideally, complete the same vector operation in roughly 1/N of the time a single processor would take, giving very high throughput for problems that are naturally organised as large, uniform arrays of data — for example, image processing, matrix operations and finite-difference simulations.

7.3.3 Array Processor versus Vector (Pipeline) Processor

Both are aimed at exploiting the same underlying data parallelism present in vector/matrix computation, but they achieve it differently:

- A pipelined vector processor uses one (or a few) pipelined arithmetic units and overlaps successive element operations in time — elements pass through the pipeline one after another, staggered.
- An array processor uses many identical arithmetic units (processing elements) and performs the same operation on many elements at the same instant in space, all synchronised by one control unit.
- In practice, real high-performance systems often combine both ideas — arrays of pipelined processing elements — to obtain both forms of parallelism together.

7.3.4 Summary Points

- An array processor is an SIMD machine: one control unit, many processing elements, one broadcast instruction executed simultaneously on different data.
- Its key components are the control unit, the PE array, local memories, and the interconnection network.
- It differs from pipelined vector processing by achieving parallelism in space (many ALUs at once) rather than in time (one pipelined ALU, overlapped).

7.4 Different Types of Array Processors

Array processors are broadly classified according to where the parallel processing hardware is placed relative to the main control/data path of the computer system. The two principal categories are attached array processors and SIMD array processors (also described as true array processors); SIMD array processors are further distinguished from associative/content-addressable processors and MIMD-style multiprocessor arrays.

7.4.1 Attached Array Processor

An attached array processor is an auxiliary, high-speed processing unit that is connected to a general-purpose host computer for the specific purpose of accelerating numerically-intensive, vector-oriented computation. The host computer continues to run the overall program, operating system, and I/O; whenever a vector/array-intensive computation is encountered, the required data is transferred to the attached processor, which performs the high-speed vector arithmetic and returns the results to the host.

- Purpose: to improve the performance of a host computer inexpensively, by off-loading vector arithmetic to specialised, pipelined hardware, without redesigning the host itself.
- Structure: typically built from pipelined arithmetic units and organised as an independent add-on unit, connected to the host through an I/O channel or a dedicated high-speed interface/bus.
- Usage pattern: the host is responsible for general-purpose processing, program control and input/output, while the attached processor is invoked specifically for vector/matrix computations.

7.4.2 SIMD Array Processor (True Array Processor)

An SIMD array processor is a stand-alone computer, built from the outset around the array-processing organisation described in Section 7.3: a single control unit driving a synchronous array of processing elements, each with its own ALU and local memory, interconnected through a network. Unlike an attached processor, it is not a mere accelerator bolted onto a separate host — the array organisation is itself the main processing structure of the machine. Examples of historically important SIMD array-processor designs include the ILLIAC IV and the Connection Machine (CM-1/CM-2) type architectures, which used thousands of simple processing elements operating in lock-step.

- Single control unit, common instruction stream, broadcast to all PEs.
- Large number of simple, identical processing elements, each with local memory.
- Best suited to problems with a regular, uniform data-parallel structure, such as image and signal processing, matrix computations, and fluid-dynamics simulations on structured grids.

7.4.3 Associative / Content-Addressable Array Processor

A variant of the SIMD array organisation replaces conventional address-based local memories with associative (content-addressable) memories in each processing element. Here, data is retrieved not by specifying a memory address but by matching data content against a search pattern broadcast by the control unit; every PE compares the pattern with its own stored data simultaneously. This is particularly useful for search, sorting, and pattern-matching type operations across a large, uniform data set.

7.4.4 Multiple-ALU / MIMD-oriented Processor Arrays

While the classical array processor is SIMD (one instruction stream, many data streams), some multiprocessor systems arrange many independent processors, each with its own control logic, into an array-like interconnection structure (mesh, hypercube, etc.) so that each processor can execute a different instruction stream on its own data — this is Multiple Instruction stream, Multiple Data stream (MIMD) organisation. Such systems are mentioned alongside array processors because they share a similar physical arrangement of many processing nodes and an interconnection network, but they differ fundamentally from SIMD array processors in that each node has independent control rather than a single shared control unit.

7.4.5 Comparison of Array Processor Types

Type	Control	Typical Use
Attached array processor	Separate host supplies overall control; attached unit only accelerates vector arithmetic	Adding vector-processing speed to an existing general-purpose host system
SIMD array processor	Single control unit; instruction broadcast to all PEs	Large, regular data-parallel problems: image processing, matrix/grid computation
Associative (content-addressable) array processor	Single control unit; PEs use content-addressable local memory	Searching, sorting, and pattern-matching over large uniform data sets
MIMD-oriented processor array	Each processing node has its own independent control unit	General-purpose parallel/distributed computation with independent tasks per node

7.4.6 Summary Points

- Attached array processors accelerate a host computer by off-loading vector computation to a dedicated pipelined unit.
- SIMD array processors (e.g., ILLIAC IV type designs) are stand-alone machines built entirely around a control-unit-plus-PE-array organisation.
- Associative array processors use content-addressable memory in each PE instead of address-based memory.
- MIMD-style processor arrays resemble array processors physically but give each node independent control, unlike true SIMD array processors.

Multiple Choice Questions (MCQs)

15 MCQs covering Sections 7.1 to 7.4. Answer key is provided at the end of this section.

1. A vector processor is best described as a machine that:

- A. Executes one instruction on a single data element
- B. Executes a single instruction on an ordered set of data elements
- C. Executes many instructions on a single data element
- D. Has no arithmetic pipelining at all

2. Which of the following is NOT a technique used in vector processing?

- A. Pipelining of arithmetic units
- B. Chaining of pipelines
- C. Use of interleaved memory
- D. Use of a single non-pipelined scalar ALU only

3. In a pipelined floating-point adder used for vector addition, successive element pairs are:

- A. Processed one at a time with no overlap
- B. Fed into the pipeline every clock cycle, overlapping in different stages
- C. Stored until the whole vector is loaded before any processing begins
- D. Processed only after the previous pair fully exits the pipeline

4. Chaining in vector processing refers to:

- A. Connecting the output of one pipeline directly as input to another pipeline as elements become available
- B. Linking two scalar registers permanently
- C. Combining two vector instructions into one opcode only
- D. Physically joining two separate computers

5. Memory interleaving is used in vector processors mainly to:

- A. Reduce the number of vector registers needed
- B. Supply operands to the pipeline at the rate the pipeline demands
- C. Increase the opcode length
- D. Eliminate the need for a control unit

6. The main source of speed advantage of vector instructions over an equivalent scalar loop is:

- A. Larger opcode size
- B. Elimination of per-iteration loop-control and repeated instruction fetch/decode overhead
- C. Use of slower clock rate
- D. Avoiding all forms of parallelism

7. For very short vectors, the speed advantage of a pipelined vector unit is:

- A. Always maximum, regardless of length
- B. Reduced, because pipeline fill/drain time is not well amortised
- C. Completely unrelated to vector length
- D. Only visible in scalar processing

8. In a typical vector instruction format, the field 'N' generally represents:

- A. The opcode
- B. The base address of the destination vector
- C. The vector length (number of elements)
- D. The interconnection network size

9. In register-based vector machines (such as Cray-1 style design), vector instructions typically reference:

- A. Raw physical disk addresses
- B. Vector registers, with separate load/store instructions moving data to/from memory
- C. No registers at all
- D. Only the control unit's internal counter

10. An array processor, as classified by Flynn's taxonomy, is an example of:

- A. SISD
- B. MISD
- C. SIMD
- D. MIMD

11. In an SIMD array processor, the control unit's main role is to:

- A. Execute all arithmetic operations itself
- B. Fetch and decode instructions and broadcast them to all processing elements
- C. Store the entire program in each processing element's memory
- D. Act as one of the processing elements

12. The key structural difference between an array processor and a pipelined vector processor is that the array processor achieves parallelism:

- A. In time, via a single pipelined unit
- B. In space, via many processing elements operating simultaneously
- C. By using only one ALU
- D. By avoiding a control unit altogether

13. An attached array processor is best described as:

- A. A stand-alone SIMD machine with no host computer

- B. An auxiliary high-speed unit connected to a host computer to accelerate vector computation
- C. A single scalar ALU with no pipelining
- D. A memory-only expansion device

14. A processor array in which each processing element retrieves data by content rather than by address is called:

- A. An attached array processor
- B. A MIMD processor array
- C. An associative (content-addressable) array processor
- D. A scalar processor

15. Which of the following organisations gives each processing node its own independent control unit, unlike a true SIMD array processor?

- A. Attached array processor
- B. SIMD array processor
- C. MIMD-oriented processor array
- D. Associative array processor

Answer Key

1-B 2-D 3-B 4-A 5-B 6-B 7-B 8-C 9-B 10-C 11-B 12-B 13-B 14-C 15-C

Broad / Long Answer Questions

10 descriptive questions for practice and examination preparation.

1. Define vector processing. Explain, with a suitable example, how a vector instruction differs from an equivalent scalar program loop.
2. Describe the various techniques used in vector processing to achieve high computational speed. Explain the role of pipelining in this context.
3. What is chaining in the context of vector pipelines? Explain with a diagram how chaining improves performance when successive vector instructions are data-dependent.
4. Explain, with proper justification, why vector processing gives a greater speed advantage for long vectors than for short vectors.
5. Describe the general format of a vector processing instruction. Explain the purpose of each field, including operation code, source/destination addresses and vector length.
6. Compare instruction-processing overhead in a scalar loop-based implementation versus a single vector instruction implementation of the same vector-addition problem.
7. Explain the concept of an array processor. Describe its organisation with the help of a block diagram, mentioning the roles of the control unit, processing elements, and interconnection network.
8. Distinguish between a pipelined vector processor and an SIMD array processor in terms of how each exploits data-level parallelism.
9. Describe the different types of array processors. Explain the working principle and application area of an attached array processor.
10. Explain the concept of associative (content-addressable) array processors and how they differ from conventional address-based SIMD array processors, as well as from MIMD-oriented processor arrays.