

UNIT 6

Pipeline, Vector Processing and Multiprocessors

(6 Hours) — Computer Organization and Architecture

Modern computers rarely execute one instruction at a time in complete isolation — performance comes from doing multiple things at once. This unit studies three levels at which a CPU exploits parallelism: pipelining, which overlaps the different phases of several instructions in one CPU; vector processing, which applies a single operation to many data elements at once; and multiprocessing, which uses several complete processors working together on the same problem.

6.1 Parallel Processing

Parallel processing is a technique of dividing a computational task into smaller parts that can be executed simultaneously, rather than one after another. The goal is always the same — increase the amount of useful work done per unit time — but this can be achieved at several different levels inside a computer system.

Levels of Parallel Processing

Level	What Runs in Parallel	Typical Mechanism
Job / Program level	Independent programs or jobs	Multiprogramming, multiple CPUs / cores
Task / Procedure level	Procedures or tasks within one program	Multithreading, multiprocessors
Instruction level	Individual machine instructions	Instruction pipelining, superscalar execution
Arithmetic / Bit level	Bits within a single arithmetic operation	Carry-lookahead adders, parallel multipliers

Flynn's classification is a widely used way to categorize parallel computer architectures according to how many instruction streams and data streams they can handle at once.

Category	Meaning	Example
SISD — Single Instruction, Single Data	One instruction stream operates on one data stream at a time	A conventional, non-pipelined uniprocessor
SIMD — Single Instruction, Multiple Data	One instruction stream operates on many data elements simultaneously	Vector / array processors, GPU cores
MISD — Multiple Instruction, Single Data	Several instruction streams operate on the same data stream	Rare in practice; mainly of theoretical interest

Category	Meaning	Example
MIMD — Multiple Instruction, Multiple Data	Several independent processors execute different instructions on different data	Multiprocessor and multicomputer systems

Example 6.1 — Classifying Systems Under Flynn's Taxonomy

Classify the following under Flynn's taxonomy: (a) a single old-style calculator CPU executing one instruction on one operand pair at a time; (b) a GPU core applying the same shading instruction to every pixel of an image at once; (c) a data-center cluster where each node runs a different program on its own data.

Solution

- (a) SISD — one instruction stream, one data stream, executed strictly sequentially.
- (b) SIMD — a single instruction is broadcast and applied to many data elements (pixels) in parallel.
- (c) MIMD — multiple independent processors, each running its own instruction stream on its own data.

Note: Pipelining, covered next, is usually described as a form of temporal parallelism (different stages of different instructions overlap in time on the same hardware), while SIMD/vector processing is a form of spatial parallelism (the same operation is replicated across many data-processing units at once).

Key Takeaways — Section 6.1

- ✓ Parallel processing exploits simultaneity at several levels: job, task, instruction, and bit/arithmetic level.
- ✓ Flynn's taxonomy classifies architectures as SISD, SIMD, MISD, or MIMD based on the number of instruction and data streams handled concurrently.
- ✓ Pipelining is temporal parallelism (overlap in time); vector/SIMD processing is spatial parallelism (replication in space).

6.2 Pipeline Examples

Pipelining is a technique that divides the execution of an instruction into a series of smaller sub-operations, called segments or stages, each carried out by dedicated hardware. Because each segment is a separate piece of hardware, several instructions can be in different segments at the same time — while one instruction is being decoded, another can be fetched, and yet another can be executing. This overlap is the source of a pipeline's speed-up, much like an assembly line in a factory.

Four-Segment Instruction Pipeline

A classic example divides instruction execution into four segments:

Segment	Name	Task Performed
FI	Fetch Instruction	Reads the next instruction from memory into an instruction register
DA	Decode & Calculate Address	Decodes the opcode and computes the effective address of any operand
FO	Fetch Operand	Reads the operand from memory or a register, using the effective address
EX	Execute Instruction	Performs the operation in the ALU and stores the result

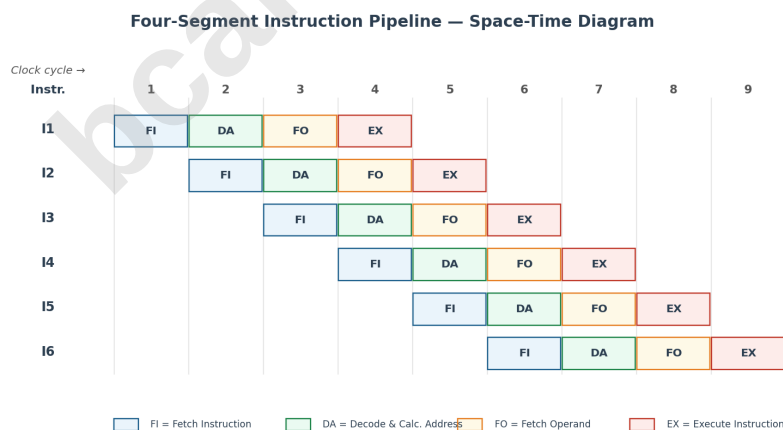


Fig. 6.1 — Space-time diagram of a four-segment pipeline: once the pipeline is full, one instruction completes every clock cycle even though each instruction itself takes four cycles to pass through the whole pipeline.

Once the pipeline has filled (after $k = 4$ cycles for a 4-segment pipeline), it produces one completed instruction per clock cycle. For n instructions through a k -segment pipeline, the total time is $(k + n - 1)$ clock cycles, compared with $k \times n$ cycles for fully sequential (non-pipelined) execution.

Example 6.2 — Speed-up of a Four-Segment Pipeline

A four-segment pipeline ($k = 4$) executes $n = 100$ instructions, and each segment takes one clock cycle (clock period = 20 ns). Compare the pipelined execution time with the non-pipelined execution time, and find the speed-up.

Solution

Pipelined time = $(k + n - 1) \times \text{clock period} = (4 + 100 - 1) \times 20 \text{ ns} = 103 \times 20 \text{ ns} = 2060 \text{ ns}$.

Non-pipelined time = $k \times n \times \text{clock period} = 4 \times 100 \times 20 \text{ ns} = 8000 \text{ ns}$.

Speed-up = Non-pipelined time / Pipelined time = $8000 / 2060 \approx 3.88$.

As n grows very large, the speed-up approaches $k = 4$ — the number of pipeline segments — which is the theoretical maximum for this design.

Note: The maximum possible speed-up of a k -segment pipeline is k , and this limit is approached only when the pipeline stays continuously full, i.e. when n is large and there are no stalls from data dependencies or branches.

Data Dependency

A pipeline assumes that instructions are independent enough to be processed concurrently. This assumption breaks down when one instruction needs a result that a previous instruction, still in the pipeline, has not yet produced — this is called a data dependency (or a Read-After-Write hazard). If the pipeline is allowed to proceed anyway, the dependent instruction would read a stale or incorrect value.

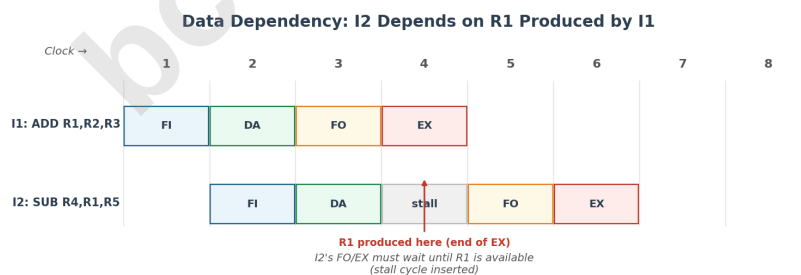


Fig. 6.2 — I2 needs R1, which I1 does not produce until the end of its EX stage. The pipeline must stall I2's later stages until R1 becomes available.

Three broad ways of handling a data dependency are commonly used:

- Hardware interlock: the control circuitry detects the dependency and automatically inserts one or more stall cycles (also called pipeline bubbles) until the required value is available.
- Operand forwarding (bypassing): instead of waiting for the result to be written back to a register and then read again, the result is routed directly from the producing stage to the

consuming stage as soon as it is computed.

- Compiler / software scheduling: the compiler detects the dependency at compile time and reorders independent instructions (or inserts NOPs) so that, by the time the dependent instruction reaches the stage that needs the value, it is already available.

Example 6.3 — Counting Stall Cycles

I1: ADD R1, R2, R3 and I2: SUB R4, R1, R5 execute back-to-back on the four-segment pipeline of Fig. 6.1, with R1 available only at the end of I1's EX stage. How many stall cycles must be inserted before I2 can proceed to its FO stage?

Solution

I1 occupies FI, DA, FO, EX in cycles 1, 2, 3, 4 respectively, so R1 is ready at the end of cycle 4.

I2 normally would occupy FI, DA, FO, EX in cycles 2, 3, 4, 5 — but I2's FO stage (cycle 4) needs R1, which is not ready until the end of cycle 4.

So I2's FO stage must be delayed by one stall cycle: I2 now occupies FI(2), DA(3), stall(4), FO(5), EX(6), exactly as shown in Fig. 6.2.

Handling of Branch Instructions

A branch instruction is especially disruptive to a pipeline, because the pipeline must decide, immediately after fetching the branch, what instruction to fetch next — but the branch's target address and whether it is even taken are usually not known until the branch reaches a later stage (e.g. EX). Meanwhile, the pipeline has already fetched several instructions that sequentially follow the branch in memory; if the branch is taken, these must be discarded.

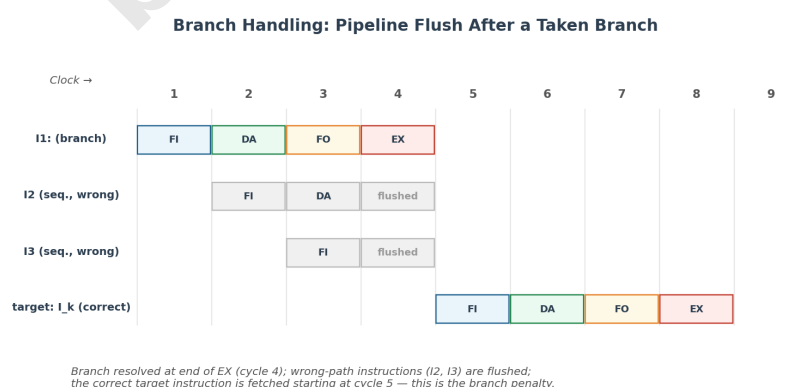


Fig. 6.3 — When the branch I1 is resolved as taken at the end of its EX stage, the sequential instructions I2 and I3 that were already fetched must be flushed, and the correct target instruction is fetched afresh — costing a branch penalty of lost cycles.

Common techniques for reducing the impact of branch instructions on a pipeline:

Technique	Idea
Pipeline flush (branch not taken assumed)	Continue fetching sequential instructions; if the branch turns out taken, discard (flush) the wrongly fetched instructions and restart fetch at the target
Delayed branch	Define one or more instruction slots after the branch that are always executed regardless of outcome, filled by the compiler with useful, independent instructions
Branch prediction	Guess whether the branch will be taken (statically, or dynamically using past behavior) and start fetching down the predicted path; roll back only on a misprediction
Branch target buffer / prefetch	Cache the target address of recently taken branches so the target instruction can be fetched immediately, without waiting for the branch to be decoded again
Multiple prefetch (both paths)	Fetch instructions from both the branch-taken and branch-not-taken paths simultaneously, and simply discard the path not chosen once the branch resolves

Example 6.4 — Branch Penalty

In the four-segment pipeline of Fig. 6.3, a branch is resolved (taken) at the end of its EX stage, which is 3 cycles after it was fetched. How many cycles of useful work are lost (the branch penalty) each time a branch is taken, assuming no prediction is used?

Solution

Instructions fetched in the 3 cycles immediately following the branch's fetch cycle (here, I2 and I3, occupying 2 of those cycles before being flushed, but more generally up to $k - 1 = 3$ cycles' worth of fetch slots) must be discarded.

For this 4-segment pipeline, the branch penalty is 3 clock cycles per taken branch — exactly the number of stages between fetch and the stage where the branch outcome becomes known.

Note: A good branch-prediction scheme does not eliminate the branch penalty — it only pays the penalty when the prediction turns out wrong, so a high prediction accuracy can make the average penalty very small even though the worst-case penalty per misprediction is unchanged.

Key Takeaways — Section 6.2

- ✓ A pipeline overlaps the segments (FI, DA, FO, EX) of successive instructions; once full, it completes one instruction per clock cycle, giving a theoretical speed-up approaching k (the number of segments).
- ✓ A data dependency (RAW hazard) forces the pipeline to stall, use operand forwarding, or rely on compiler reordering until the needed value is available.
- ✓ Branch instructions disrupt sequential fetching; techniques such as pipeline flush, delayed branch, and branch prediction are used to reduce the resulting branch penalty.

bcalink.com

6.3 Vector Processing

Many scientific and engineering problems — weather simulation, structural analysis, image processing, machine learning — are naturally expressed as operations on large arrays of numbers rather than on single scalar values. A vector processor is a CPU with instructions that operate on an entire vector (a one-dimensional array of data) as a single operand, instead of requiring a separate instruction, and a separate pass through the fetch–decode–execute cycle, for every individual element.

Why Vector Processing?

Consider adding two arrays of n numbers element-by-element using an ordinary scalar instruction set: a loop must fetch each pair of elements, add them, store the result, increment the index, and test for the loop's end — repeated n times, with all the instruction-fetch and loop-control overhead that entails. A vector processor instead issues one instruction, such as `VADD C, A, B`, that internally streams all n element pairs through a pipelined adder, eliminating almost all of the per-element instruction overhead.

Vector Operations

A vector instruction set typically provides operations such as:

Operation	Notation	Description
Vector–Vector	$C \leftarrow A + B$	An element-wise operation between two vectors of the same length, producing a vector result
Vector–Scalar	$B \leftarrow k \times A$	Every element of a vector is combined with a single scalar value
Vector Reduction	$s \leftarrow \sum A_i$	Combines all elements of a vector into a single scalar, e.g. summing all elements
Gather / Scatter	—	Loads (gathers) or stores (scatters) vector elements from/to non-contiguous memory locations, guided by an index vector
Vector Compare / Mask	$M \leftarrow (A > B)$	Produces a Boolean mask vector used to conditionally select or suppress operations on individual elements

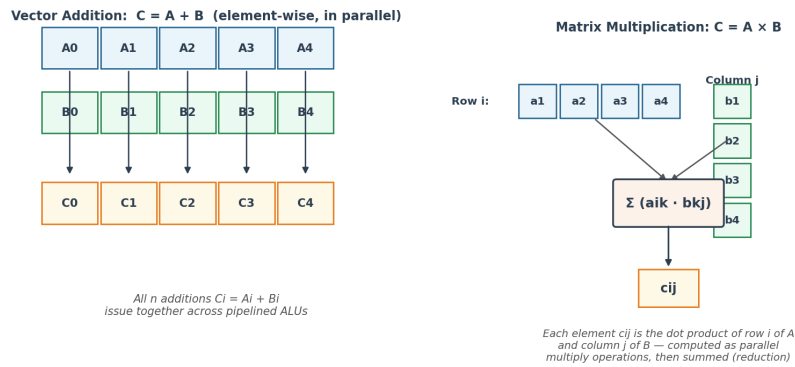


Fig. 6.4 — Left: vector addition $C = A + B$ performed element-wise, with all additions issued together across pipelined ALUs. Right: matrix multiplication, where each output element is the dot product of a row of A and a column of B .

Internally, a vector processor most often implements these operations using a pipelined functional unit: successive pairs of vector elements are fed into the pipeline on successive clock cycles, so — just as with the instruction pipeline of Section 6.2 — the pipeline fills up and then produces one result per cycle, even though each individual addition takes several cycles to complete.

Example 6.5 — Time to Add Two Vectors on a Pipelined Adder

A pipelined adder has 4 stages, one clock cycle per stage. Two vectors of length $n = 64$ are added element-by-element using this pipeline. How many clock cycles are needed, and what is the effective throughput once the pipeline is full?

Solution

Using the pipeline timing formula from Section 6.2: total cycles = $(k + n - 1) = (4 + 64 - 1) = 67$ clock cycles.

Without pipelining, 64 separate additions taking 4 cycles each would need $4 \times 64 = 256$ cycles, so the speed-up here is $256 / 67 \approx 3.82$.

Once filled, the pipeline delivers one completed sum every clock cycle — a throughput of one result per cycle, approaching the ideal of $k = 4\times$ speed-up for very large n .

Matrix Multiplication

Matrix multiplication is one of the most common and most computationally demanding operations that vector processors are designed to accelerate. For an $m \times n$ matrix A multiplied by an $n \times p$ matrix B , giving an $m \times p$ result matrix C , every output element is defined as:

$$C_{ij} = \sum_{k=1}^n A_{ik} \cdot B_{kj} \text{ (the dot product of row } i \text{ of } A \text{ and column } j \text{ of } B)$$

A scalar processor computes this with three nested loops (over i , j , and k), performing n multiply-add operations for every one of the $m \times p$ output elements — a total of $m \times n \times p$ multiply-add operations, each requiring its own instruction fetch/decode cycle in a purely scalar machine.

A vector processor instead treats the innermost loop — computing one dot product — as a single vector operation: it multiplies row i of A element-by-element with column j of B using one vector-multiply instruction (producing a vector of n partial products all at once through the pipeline), and then applies a vector reduction (sum) instruction to add up those n partial products into the single scalar result C_{ij} . This is repeated for each of the $m \times p$ output elements, but each repetition now costs one vector-multiply plus one reduction instead of n separate scalar multiply-add instructions.

Example 6.6 — Multiply-Add Count for Matrix Multiplication

Matrix A is 4×3 and matrix B is 3×5 . Find the size of the result matrix C , and find the total number of scalar multiply operations needed to compute C using the ordinary row-by-column definition.

Solution

$C = A \times B$ has dimensions (rows of A) $\times$ (columns of B) = 4×5 .

Each of the $4 \times 5 = 20$ elements of C is a dot product of length $n = 3$ (the shared inner dimension), needing 3 multiplications (and 2 additions) each.

Total multiplications = $20 \times 3 = 60$ scalar multiply operations (and $20 \times 2 = 40$ additions) — on a vector processor, this becomes 20 vector-multiply-and-reduce operations, each internally streaming 3 elements through the pipeline.

Note: Because matrix multiplication reuses each row of A across every column of B (and vice versa), real vector/array processors also apply techniques such as blocking (tiling) and data reuse in cache or local vector registers, so that operands already loaded into fast storage are used for as many dot products as possible before being replaced.

Key Takeaways — Section 6.3

- ✓ A vector processor applies one instruction to an entire array of data, drastically reducing per-element instruction-fetch overhead compared with scalar loops.
- ✓ Vector operations include vector-vector, vector-scalar, reduction, gather/scatter, and compare/mask operations, usually implemented on pipelined functional units.
- ✓ Matrix multiplication maps naturally onto vector hardware: each output element is a dot product, computed as a vector multiply followed by a vector-reduction (sum).

— *End of Unit 6* —

bcalink.com