Chapter 7
Pipeline and Vector Processing

7.1 Parallel Processing, Flynn's Classification of Computers

Parallel Processing

→ Parallel processing is a technique that is used to provide simultaneously data processing task for the purpose of increasing the conceptual speed of computer system.

→ The system may have two or more ALUs and be able to execute two or more instructions at the same time.

→ The purpose of parallel processing is to speed up the computer processing capability and increase its throughput (the amount of task those are completed during given interval of time). It is called parallel computing.

Flynn's Classification of Computers

→ There are four groups of computers according to the Flynn's classification based on the number of concurrent instruction and data streams manipulated by the computer system.

→ The normal operation of a computer is to fetch instructions from memory and execute them in processor.

→ The sequence of instructions read from memory constitutes an *instruction stream*.

→ The operations performed on the data in the processor constitute a *data stream*.

→ Parallel processing may occur in the instruction stream, in the data stream, or in both.

→ The Flynn's classification of computer are:

i. single instruction stream, single data stream (**SISD**)
ii. single instruction stream, multiple data stream (**SIMD**)
iii. multiple instruction stream, single data stream (**MISD**)
iv. multiple instruction stream, multiple data stream (**MIMD**)

→ SISD has a processor, a control unit, and a memory unit. There is a single processor which executes a single instruction stream to operate on the data stored in a single memory in this system. The parallel processing in this case may be achieved by means of multiple functional units or pipeline processing.

→ SIMD executes a single machine instruction on different set of data by different processors.

- Each processing element has associated data memory.
- All the processor receives the same instruction from control unit but operate on different items of data.
- Application of SIMD is vector and array processing.

→ MISD has many functional units which perform different operations on the same data. It is a theoretical model of computer.

→ MIMD consists of a set of processors which simultaneously execute different instruction sequences on the different set of data.

7.2 Pipelining

→ Pipelining is a technique of decomposing a sequential process into sub operations, with each sub process being executed in a special dedicated segment that operates concurrently with all other segments.

→ Each segment performs partial processing dictated by the way the task partitioned.

→ The result obtained from the computation in each segment is transferred to the next segment in the pipeline. The final result is obtained after the data have passed through all segments.

→ It is characteristic of pipelines that several computations can be in progress in distinct segments at the same time.

→ The registers provide isolation between each segment so that each can operate on distinct data simultaneously.

Suppose we want to perform the combined multiply and add operations with a stream of numbers.

E.g. $A_i * B_i + C_i$ for i = 1 , 2, 3,……,7

The sub operations performed in each segment of the pipeline are as follows:

$R1 \leftarrow A_i$, $R2 \leftarrow B_i$	Input A_i and B_i
$R3 \leftarrow R1*R2$, $R4 \leftarrow C_i$	Multiply and input C_i
$R5 \leftarrow R3+R4$	Add C_i to product

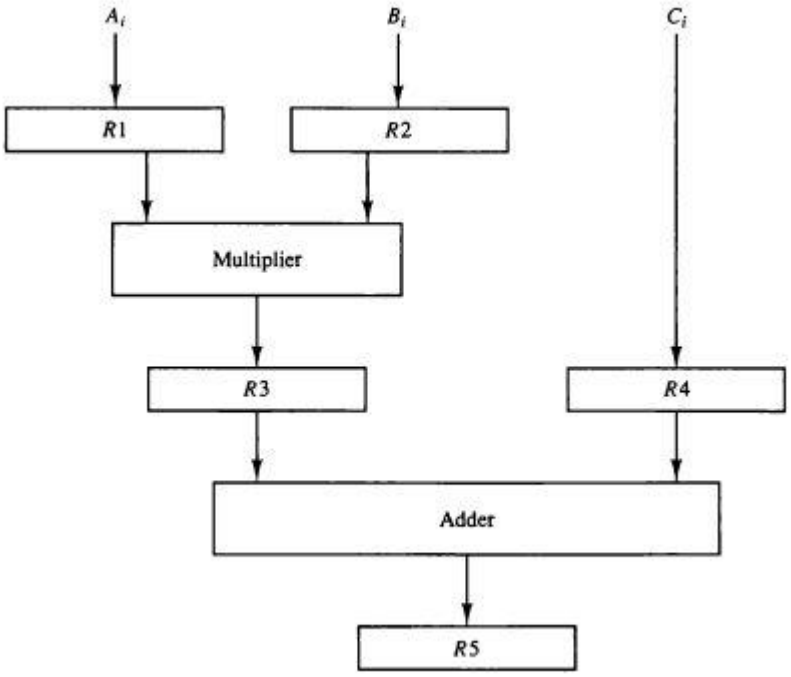

Fig: Example of pipeline processing

Clock Pulse Number	Segment 1		Segment 2		Segment 3
	R1	R2	R3	R4	R5
1	A_1	B_1	-	-	-
2	A_2	B_2	A_1*B_1	C_1	-
3	A_3	B_3	A*B	C	$A_1*B_1+C_1$
4	A_4	B_4	A*B	C	A*B+C
5	A_5	B_5	A*B	C	A*B+C
6	A_6	B_6	A*B	C	A*B+C
7	A_7	B_7	A*B	C	A*B+C
8	-	-	A*B	C	A*B+C
9	-	-	-	-	A*B+C

Speedup Equation

→ Consider a K-segment pipeline with a clock cycle time t_p to execute n tasks. The first task T_1 require time Kt_p to complete. The remaining (n-1) tasks finish at the rate of one task per click cycle and will be completed after time (n-1) t_p. The total time to complete the n task is $[Kt_p + (n - 1) t_p] = (K + n - 1) t_p$.

→ Consider a non–pipeline unit that performs the same operation and takes t_n time to complete each task. The total time required for n tasks would be nt_n.

→ The speedup of pipeline processing over an equivalent non–pipeline processing is defined by the ratio

$$\text{Speedup (S)} = \frac{\text{Total time taken by non–pipeline structure to complete n tasks}}{\text{Total time taken by pipeline structure to complete n tasks}}$$

$$S = \frac{nt_n}{(K+n-1)t_p}$$

→ As number of tasks increases, n becomes much larger than K - 1, and (K + n – 1) approaches to n then

$$S = \frac{t_n}{t_p}$$

→ If we assume that the time to process a task is same in both circuits then $t_n = Kt_p$

$$S = \frac{Kt_p}{t_p} = K$$

Q. Calculate pipeline speedup if time taken to complete a task in conventional machine is 25 ns. In the pipeline machine, one task is divided into 5 segments and each sub operation tasks take 4 ns. Number of tasks to be completed is 100.

Solution: Here,

Time taken to complete a task (t_n) = 25 ns
Number of segment (K) = 5
Clock cycle time (t_p) = 4 ns
No. of tasks (n) = 100

$$S = \frac{nt_n}{(K+n-1)t_p}$$

Q. Calculate the speed up rate of 5-segment pipeline with a clock cycle time 25ns to execute 100 tasks.

Solution: Here,
n = 100
Number of segment (K) = 5
Time to complete a task (t_p) = 25 ns
$t_n = Kt_p$

$$S = \frac{nt_n}{(K+n-1)t_p}$$

Q. A non–pipeline system takes 100 ns to process a task. The same task can be processed in a six-segment pipeline with time delay of each segment in the pipeline is as follows; 20 ns, 25 ns, 30 ns. Determine the speed of ratio of pipeline for 100 tasks.

t_n = 100 ns
K = 6
t_p = 30 ns
n = 100

$$S = \frac{nt_n}{(K+n-1)t_n}$$

Q. Suppose that time delays of four segments are t_1 = 60ns, t_2 = 70 ns, t_3 = 100 ns, t_4 = 80 ns and interface register have a delay of 10 ns. Determine the speed up ratio.

Solution: Here,

t_p = 100 + 10 = 110 ns
$t_n = t_1 + t_2 + t_3 + t_4$
=60+70+100+80 = 320 ns

$$S = \frac{t_n}{t_p}$$

7.3 Arithmetic Pipeline

→ Pipeline arithmetic units are usually found in very high speed computers. They are used to implement floating point operations, multiplication of fixed point numbers, and similar computations encountered in scientific problems.

→ We will now show an example of a pipeline unit for floating point addition and subtraction. The inputs to the floating point adder pipeline are two normalized floating point binary numbers.

$$X = A \times 2^a$$
$$Y = B \times 2^b$$

→ The floating point addition and subtraction can be performed in four segments, as shown in figure. The registers labeled R are placed between the segments to store intermediate results.

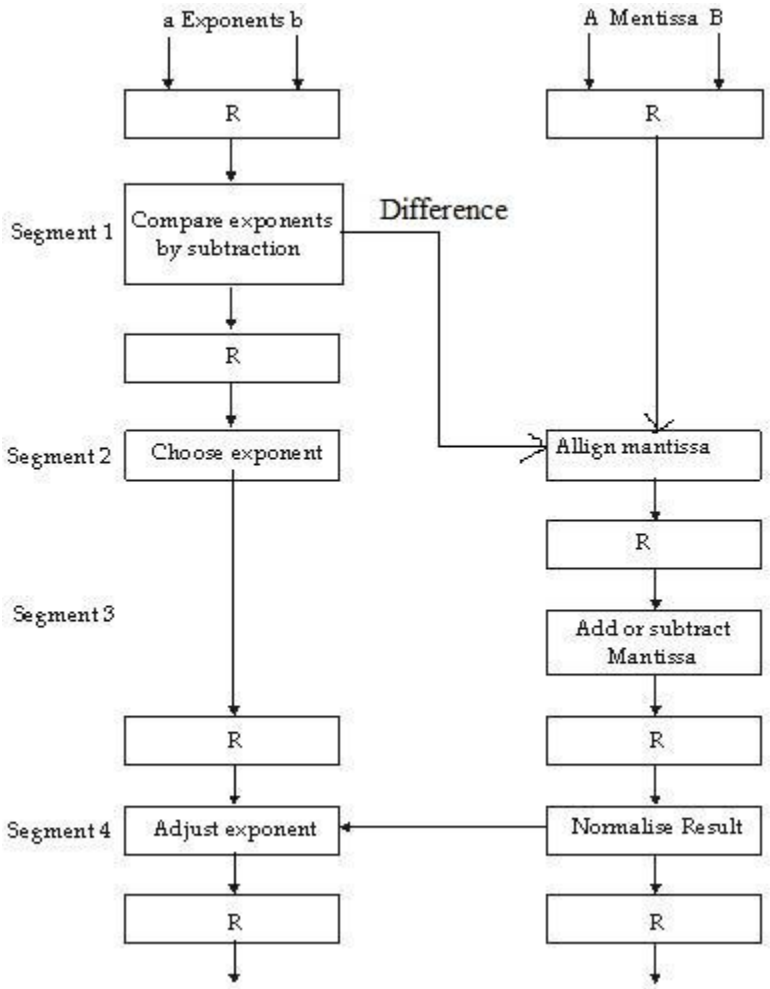

Fig: Pipeline for floating-point addition and subtraction

→ The suboperations that are performed in the four segments are:

i) Compare the exponents.
ii) Align the mantissas.
iii) Add or subtract the mantissas.
iv) Normalize the result.

→ Procedure: The exponents are compared by subtracting them to determine their difference. The larger exponent is chosen as the exponent of the result. The exponent difference determines how many times the mantissa associated with the smaller exponent must be shifted to the right. This produces an alignment of the two mantissas. It should be noted that the shift must be designed as a combinational circuit to reduce the shift time. The two mantissas are added or subtracted in segment 3. The result is normalized in segment 4. When an overflow occurs, the mantissa of the sum or difference is shifted right and the exponent

incremented by one. If an underflow occurs, the number of leading zeros in the mantissa determines the number of left shifts in the mantissa and the number that must be subtracted from the exponent.

→ For simplicity, we use decimal numbers, the two normalized floating-point numbers:

$$X = 0.9504 \times 10^3$$
$$Y = 0.8200 \times 10^2$$

The two exponents are subtracted in the first segment to obtain 3-2 = 1. The larger exponent 3 is chosen as the exponent of the result. The next segment shifts the mantissa of Y to the right to obtain

$$X = 0.9504 \times 10^3$$
$$Y = 0.0820 \times 10^3$$

This aligns the two mantissas under the same exponent. The addition of the two mantissas in segment 3 produces the sum

$$Z = 1.0324 \times 10^3$$

7.4 Instruction Pipeline

→ Pipeline processing can not only occur in the *data stream* but in the *instruction stream* as well.

→ An instruction pipeline reads consecutive instructions from memory while previous instructions are being executed in other segments. This causes the instruction fetch and execute phases to overlap and perform simultaneous operations. This technique is called instruction pipelining.

→ Consider sub dividing instruction processing into two ways:

— fetch instruction
— execute instruction

→ During execution, there is time when main memory is not being accessed. During this time, the next instruction could be fetched and buffered (called instruction pre-fetch or fetch overlap).

→ In the most general case, the computer needs to process each instruction with the following sequence of steps:

i. Fetch the instruction from memory.
ii. Decode the instruction.
iii. Calculate the effective address.
iv. Fetch the operands from memory.
v. Execute the instruction.
vi. Store the result in the proper place.

→ To again further speed up, the pipeline must have more stages consider the following decomposition of instruction processing.

a) Fetch instruction (FI)
b) Decode instruction (DI)
c) Fetch operands (FO)
d) Execute instruction (EI)

	1	2	3	4	5	6
Insruction1	FI	DI	FO	EI		
Instruction2		FI	DI	FO	EI	
Instruction3			FI	DI	FO	EI

Fig: Timing diagram for 4-segment instruction pipeline

7.5 Pipeline Hazards and their Solution

Hazards

1. **resource conflicts**

→ If one common memory is used for both data and instruction, and there is need to read/write data and fetch the instruction at the same time, the resource conflicts occur.

2. **data dependency conflict**

→ Data dependency conflicts arise when an instruction depends on the result of a previous instruction but result of that instruction is not yet available.

3. **branch difficulties**

→ Branch difficulties arise from program control instructions that may change the value of PC.

→ A program is not a straight flow of sequential instructions. There may be branch instructions after the normal flow of program which delay the pipelining executions and affects the programs.

Solution of pipeline hazards

1. Resource conflicts can be solved by using separate instruction and data memory.
2. To solve the data dependency conflict, we have following methods:
   - i) **hardware interlock**
     Hardware interlock is a circuit that is responsible to detect the data dependency. After detecting the particular instruction needs data from instruction which is being executed, hardware interface delays the instruction till the data is not available.
   - ii) **operand forwarding**
     Operand forwarding uses special hardware to detect a conflict and then avoid by routing the data through special paths between pipeline segments.
   - iii) **delayed load**
     Delayed load is a procedure that gives the responsibilities for solving data conflicts to the compiler. The compiler is designed to detect conflict & reorder the instructions as necessary to delay the loading of the conflicting data by inserting no operation instructions.
3. Solution to branch difficulties
   - i) **pre fetch target instruction**
     Both the branch target instruction & the instruction following the branch are pre fetched and are saved until the branch instruction is executed. If branching occurs then branch target instruction is continuous.
   - ii) **branch target buffer (BTB)**
     A branch target buffer is an associative memory included in fetch segment of branch instruction that stores the target situation for the previously executed branch. It also stores the next few instructions after the branch target instruction. This way, the branch instructions that have occurred previously are readily available in the pipeline without interruption.
   - iii) **loop buffer**
     The loop buffer is similar to BTB but its speed is high. Program loops are stored in the loop buffer. The program loop then can be executed directly without having access to memory.
   - iv) **branch prediction**
     Special hardware is used to detect the branch in the conditional branch instruction. On the basis of prediction, the instructions are pre fetched.
   - v) **delayed branch**
     Compiler detects the branch instructions, so it re-arranges the instruction to make delay by inserting no operation instruction.

7.5 Array and Vector Processing

Vector Processing

There is a class of computational problems that are beyond the capabilities of a conventional computer. These problems characterized by the fact that they require a vast number of computations that will take a conventional computer days or even weeks to complete. In many science and engineering applications, the problems can be formulated in terms of vectors and matrices that lend themselves to vector processing.

Computers with vector processing capabilities are in demand in specialized applications. The following are representative application areas where vector processing is of the utmost importance.

Long-range weather forecasting
Petroleum explorations
Seismic data analysis
Medical diagnosis
Aerodynamics and space flight simulations
Artificial intelligence and expert systems
Mapping the human genome
Image processing

To achieve the required level of high performance, it is necessary to utilize the fastest and most reliable hardware and apply innovative procedures from vector and parallel processing techniques.

Array Processor

→ An array processor is a processor that performs computations on large arrays of data. The term is used to refer to two different types of processors:

i) **Attached Array Processor**
   — It is the auxiliary processor attached to a general purpose computer which is intended to improve the performance of the host computer in specific numerical computation tasks.

ii) **SIMD Array Processor**
   — It is a processor that has a single instruction multiple data organization. It manipulates vector instruction by means of multiple functional units responding to a common instruction.