11.1 Introduction

Achieving high performance from a computer is one of the main goals of a computer architect. Implementing concurrency in the organization of a computer enhances its performance as multiple operations are simultaneously performed. The design of the control unit is complex due to multiple datapaths operating simultaneously. As discussed in Chapter 1, different types and several techniques of concurrency are available. Providing an overlap or parallelism within a computer is a standard technique that enhances the performance of the computer. Another technique that is followed is multiple processors within a computer system. This chapter discusses concurrency in uniprocessor except superscalar architecture which is dealt with in Chapter 12 along with a discussion on multiprocessors. Also, this chapter provides a detailed study of the pipelining concept and the associated design issues. Besides these, a brief coverage on the vector processing and array processing is also presented. As we would see, in most of the techniques, the operating system and application program (machine language) have to take necessary steps to exploit the architectural provision of the system.

11.2 Performance Enhancement Strategies

A computer's performance is measured by the time taken for executing a program. The program execution involves performing instruction cycles, which includes two types of activities:

1. *Internal microoperations* performed inside the hardware functional units such as the processor, memory, I/O controllers etc.
2. *Transfer of information* between different hardware functional units for instruction fetch, operand fetch, input/output operation etc.

The optimal performance of a computer depends on the type of the hardware used. To exceed the system constraints, various performance improvement strategies are used. The objective here is to increase the number of operations executed in a given time. Two basic strategies are followed:

1. *Overlap:* Splitting a task into multiple subtasks that can be performed in an overlapped manner.
2. *Parallelism:* Executing more than one task in parallel.

The *Overlap strategy* uses concurrent operations by heterogeneous functional units whereas the *Parallelism* uses concurrent operations by homogeneous functional units.

Figure 11.1 depicts the commonly used overlap techniques. These have already been discussed in the earlier chapters except pipelining and multiple functional units, that are covered in this chapter. Figure 11.2 illustrates various techniques of parallelism. Broadly speaking, there are two levels of parallelism:

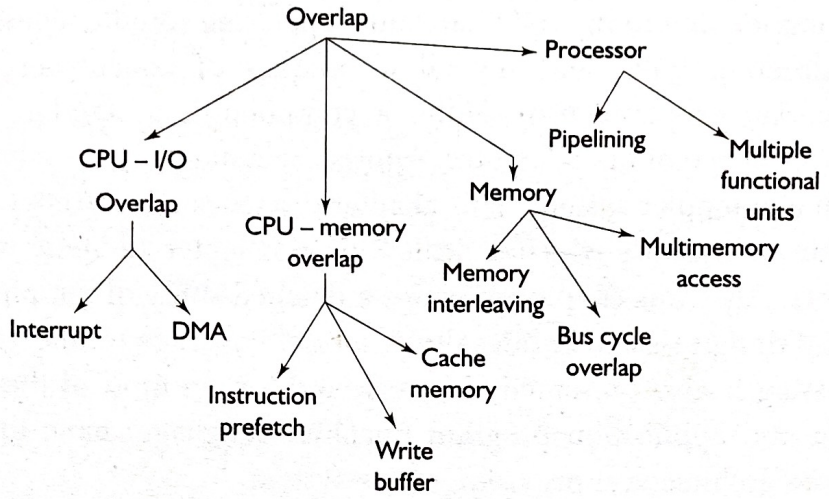


Fig. 11.1 Overlap techniques

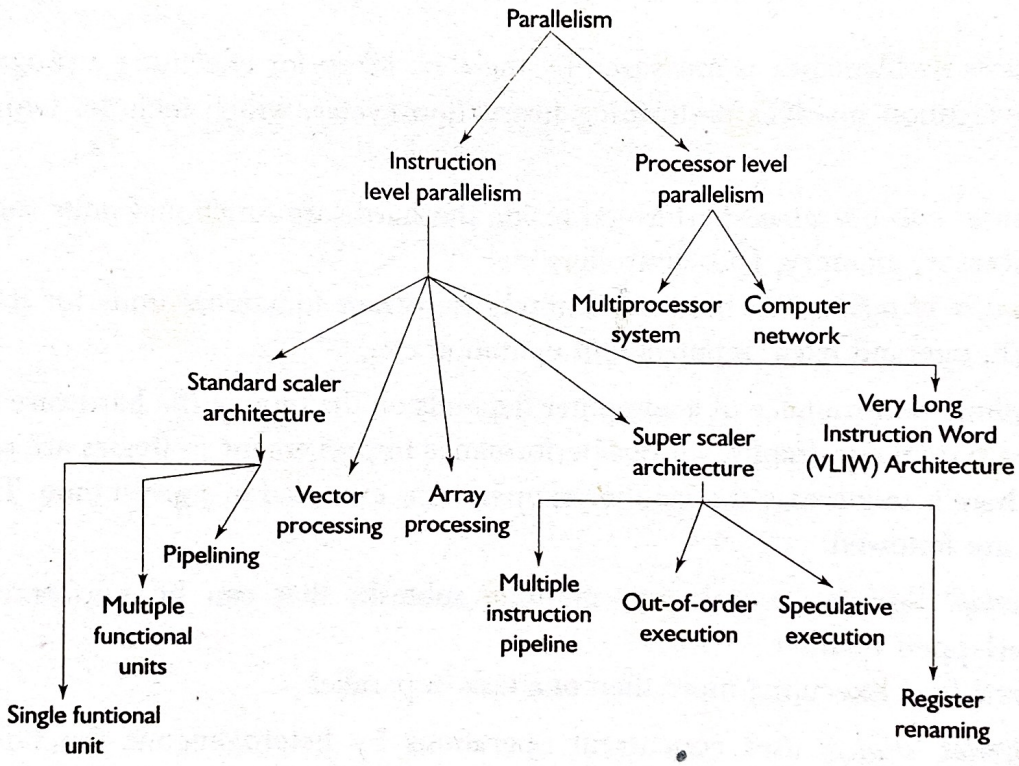


Fig. 11.2 Techniques of parallelism

1. Instruction level parallelism
2. Processor level parallelism

In the *Instruction level parallelism*, parallelism is applied inside a single processor. Traditional architectures use various techniques to extract instruction-level parallelism from the instruction stream of an application program and enhance the performance. The vector and array architectures take a different approach and exploit the advantage of data parallelism. In *Processor level parallelism*, there are multiple processors that share the workload (tasks). This can be accomplished either by using more than one processor inside a single computer system (multiprocessor system) or distributing the tasks amongst multiple computer systems linked either as a cluster or a network.

11.3 Classification of Parallelism

Parallelism is classified by using Flynn's classification of computer organization into four different types. Figure 11.3 illustrates the four types which are as follows:

- Single Instruction stream, Single Data stream (SISD)
- Single Instruction stream, Multiple Data stream (SIMD)
- Multiple Instruction stream, Single Data stream (MISD)
- Multiple Instruction stream, Multiple Data stream (MIMD)

The capability of the processor varies in these different types. The execution unit is indicated as a processor (PR) since it is often more complex than a traditional execution unit.

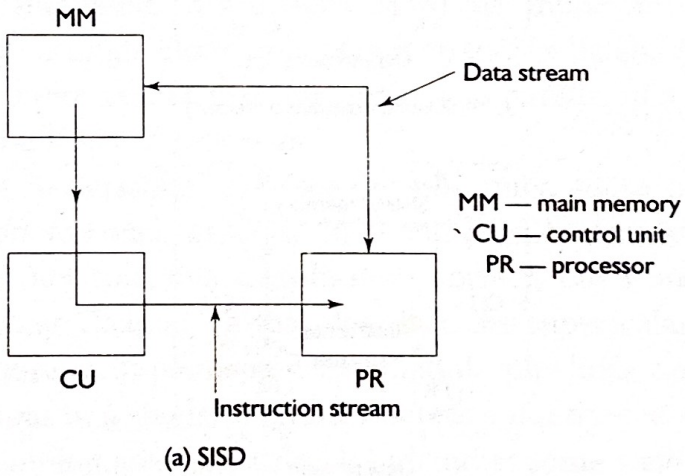
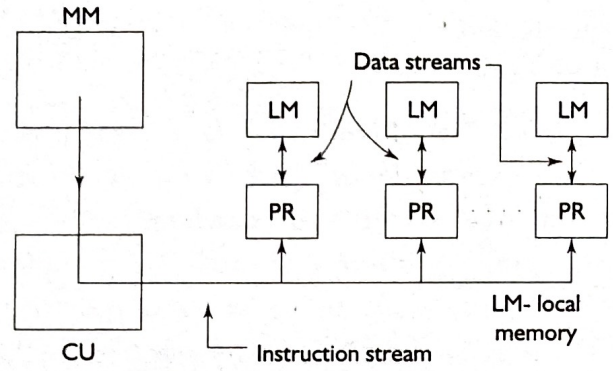
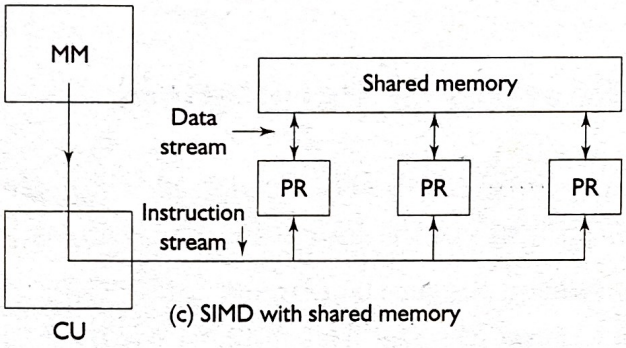


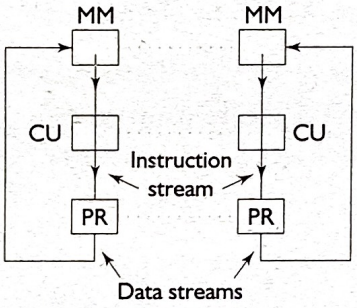
Fig. 11.3 Flynn's classification of computers



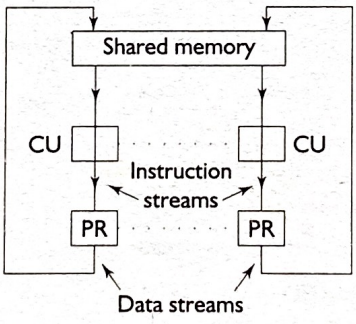
(b) SIMD with distributed memory



(c) SIMD with shared memory



(d) MIMD with distributed memory



(e) MIMD with shared memory

Fig. 11.3 Flynn's classification of computers

The SISD is a traditional uniprocessor. There is a *single control unit* and a *single execution unit.* Hence, it has one instruction and one data stream.

The SIMD has *one control unit* that handles *multiple execution units.* Each execution unit has a separate data stream. The array processor and Multi-Media Extension (MMX) in Pentium are examples of this type. Basically, while executing a single instruction, simultaneously multiple datapaths operate on multiple data elements.

The MISD involves *multiple control units* but *a single execution unit.* Such a system is not practically feasible. The MIMD refers to *multiple control units* and *multiple execution units.* Multiprocessors and parallel processors are examples of this type.

11.3.1 Scalar, Vector, Superscalar and Pipelined Processor

A simple *Scalar processor* performs only one arithmetic operation at a time—it executes one instruction at a given time and each instruction has only one set of operands. In a *Vector processor,* each instruction has multiple sets of operands such as elements of two arrays and the same operation is carried out simultaneously on different sets of operands. Hence a *vector processor* performs multiple arithmetic operations simultaneously, on different operands. In a *Non-pipelined (sequential) scalar processor,* there is no overlap of successive instruction cycles. In a *Pipelined processor,* at any given point of time multiple instructions are at various stages of instruction cycle simultaneously in different sections of the processor. Hence, the processing of multiple instructions are overlapped in a pipelined scalar processor. An *Array processor* has multiple execution units that operate simultaneously on multiple elements of a vector.

A *Superscalar processor* is a scalar processor that performs multiple instruction cycles simultaneously for successive instructions. Two or more instructions are initiated simultaneously and in a single clock cycle, same operation (fetch, decode, execute etc.) is executed on two or more consecutive instructions in parallel. To achieve this, multiple functional units are built in the processor.

Though the term 'superscalar' appears recently only, there have been some early superscalar processors too such as CDC 6600 and IBM System/360 Model 91. Perhaps some readers would not find this classification correct, but I am sure they would be convinced after reading Chapter 12 that describes the superscalar architecture. Usually superscalar architecture is implemented by multiple pipelines operating in parallel on consecutive instructions in a single program whereas a multiprocessor system operates on multiple programs simultaneously. Table 11.1 identifies some well-known processors and their classifications.

Another new classification of processor architecture is Very Long Instruction Word (VLIW) that is a refinement over standard superscalar architecture. The VLIW processor is discussed in Chapter 12.

TABLE 11.1 A sample of processors

S. no.	Processor/system	Types	Additional remarks	Flynn's type
1	IBM 1401	Scalar, sequential	Mainframe; 2nd generation computer	SISD
2	PDP-8	Scalar, sequential	Minicomputer using core memory	SISD
3	Intel 8080	Scalar, sequential	Microprocessor	SISD
4	IBM System/360 Model 30	Scalar, sequential	Mainframe; 3rd generation computer	SISD
5	IBM System/360 Model 91	Superscalar, pipelined	Mainframe; 3rd generation computer	SISD
6	Intel 80386	Scalar, pipelined	Microprocessor	SISD
7	Intel Pentium	Superscalar, two way; pipelined	Microprocessor with three execution units	SISD
8	Intel Pentium-PRO	Superscalar, three way; pipelined	Microprocessor: five execution units	SISD
9	Power PC	Superscalar	Microprocessor; RISC	SISD
10	CRAY-1	Vector processor	Supercomputer	SISD
11	TI ASC	Vector processor	Supercomputer	SISD
12	ILLIAC IV	Array processor	64 processors in a single system	SIMD
13	STARAN	Array processor	256 (bit-serial) processors	SIMD
14	FSP 164/MAX	Array processor	Attached array processor	SIMD
15	IBM 2938	Array processor	Attached array processor for IBM System/360 Models 44, 65 or 75	SIMD
16	Pluribus	Multiprocessor	Interface Message Processor (IMP)	MIMD
17	IBM System/370 Model 158 MP	Multiprocessor, CISC	Mainframe	MIMD
18	IBM System/360 Model 67	Multiprocessor, CISC	Designed to support efficient time sharing	MIMD

Contd...

→

S. no.	Processor/system	Types	Additional remarks	Flynn's type
19	Intel Pentium II	Superscalar with Multi-Media Extension (MMX), CISC	MMX operates on eight data elements simultaneously	SIMD MMX
20	CDC 6600	Superscalar, CISC	One central processor with 10 functional units	SIMD
21	UltraSPARC III	Superscalar, RISC	Supports multiprocessing	SISD
22	Intel 860	RISC microprocessor	Has on-chip graphics unit	SISD

11.4 Multiple Functional Units

Figure 11.4 presents a system with multiple hardware functional units operating in parallel inside a single processor. The instruction dispatch unit assigns the current instruction to the relevant unit based on the decoding result. The assigned unit continues with the following steps such as operand address calculation, operand fetch, execution etc. As soon

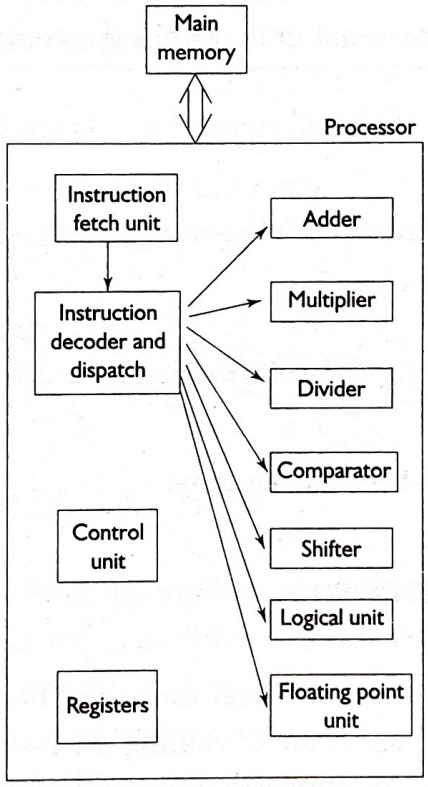


Fig. 11.4 Multiple functional units in a single processor

as the current instruction is assigned to the functional unit, the instruction dispatch unit assigns next instruction to another unit when it is free. It can have more than one functional unit of the same type such as multiple adders, multiple shifters etc. The control unit becomes more complex than the one in a simple processor. However, the throughput achieved is several times higher. The control unit should be alert in conflict handling such as dependency cases. For example, an instruction A whose operand depends on the execution of the instruction B, should not be executed until B is completed. This technique of employing multiple functional units in a single processor has a modern name, 'superscalar' architecture, meaning multiple instructions of a single program are simultaneously started and processed, in parallel, in different functional units. The Chapter 12 provides the details of the superscalar architecture present in the modern microprocessors.

11.4.1 CASE STUDY 1— CDC 6600

The CDC 6600 is an example of a high performance computer with multiple functional units. The CDC 6600 is known as a network computer since it has a special organization with one central processor which consist of 10 functional units and 10 peripheral and control processors. The 10 functional units within the central processor are as follows:

- One fixed point adder
- Two multipliers
- One divider
- Two incrementers
- One floating-point adder
- One shifter
- One logical unit
- One branch unit

The central processor has 24 useful registers:

- eight index registers
- eight operand address registers
- eight floating-point registers

The program does not see the functional units and hence does not need any special modification. The control unit takes care of routing the instructions to functional units and taking care of 'clash' situations. As long as there are no conflicts present in the instructions, up to 10 instructions can be issued in the central processor.

Figure 11.5 shows the functional units and registers of CDC 6600. The central processor consist of an instruction stack that can store up to eight words of 60 bits. The instructions are of two types: 16 bit and 30 bit instructions.

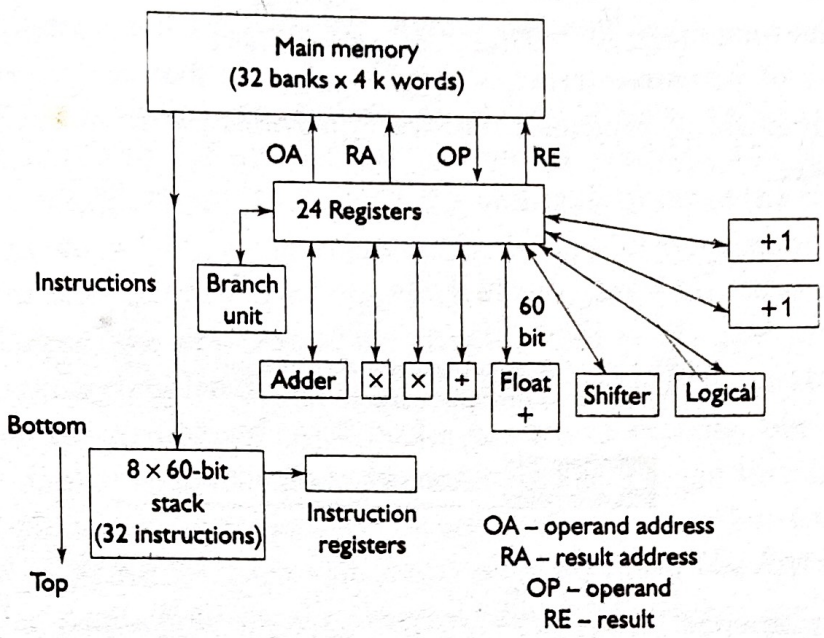


Fig. 11.5 CDC 6600 central processor—registers and functional units

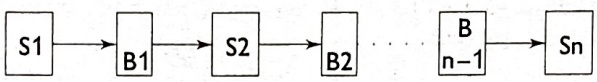
The following are the different items of concurrency in CDC 6600:

1. Instruction prefetch: eight words of 60 bits each.
2. 10 functional units along with a 'scoreboard' that provides a queue and reservation scheme.
3. 32-way memory interleaving.
4. 10 peripheral processors with own memory for I/O programs.

11.5 Pipelining

The term 'pipelining' is borrowed from the traditional production floor where a product is assembled as a pipeline in multiple stages at different stations. Pipelining is a technique of splitting one task into multiple subtasks, and executing the subtasks of successive tasks parallelly, in multiple hardware units or sections. At each station, one step of the assembly operation is executed and the partially completed product is transferred to next station that carries out the next step. Figure 11.6 shows the pipelining concept. The S1, S2, ... , Sn are n sections that form the pipeline. Each section performs a subtask on the input re-

ceived from the interstage buffer that stores the output of the previous section. The interstage buffers isolate the adjacent sections. All sections can perform their operations simultaneously. The objective is to achieve the increased throughput due to the overlap between the execution of the successive tasks. The time taken for the execution of each task is same as the time taken for a sequential (non-pipelined) execution. But due to concurrent execution of successive tasks, a number of tasks that can be executed within a given time is higher i.e., a pipelined hardware provides better throughput than a non-pipelined hardware.



$S1, S2, \ldots S_n$ — sections/segments of a processor
$B1, B2, B_{n-1}$ — interstage buffers

Fig. 11.6 Concept of pipelining

There are two types of pipelines:

1. Instruction pipeline
2. Arithmetic pipeline

An *Instruction pipeline* splits an instruction cycle actions into multiple steps that are executed one-by-one in different sections of the processor. An *Arithmetic pipeline* divides an arithmetic operation, such as a multiply, into multiple arithmetic steps each of which are executed one-by-one in different arithmetic sections in the ALU (arithmetic logic unit). A processor consist of either one or both types of pipelines.

The concept of an arithmetic pipeline is discussed for floating-point operations in Section 11.6. The following discussion deals with instruction pipeline.

11.5.1 Instruction Pipeline

An instruction pipeline consists of multiple sections each of which perform one of the functions of the instruction cycle for all the instructions. The processor hardware is structured into different independent sections for this. The actual number of sections (and hence the number of steps in the instruction cycle) in the pipeline is designed by the computer architect. The simplest case is a two stage pipeline as shown in Fig. 11.7. The instruction cycle is split into two steps—Instruction Fetch (FI) and Instruction Execute (EI). The Fetch Unit (FU) performs instruction fetch step, whereas the Execution Unit (EU) carries out the instruction execution step that includes the remaining actions of an

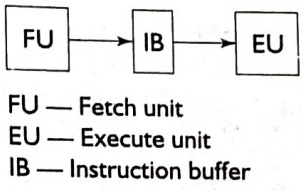


Fig. 11.7 Two-stage pipeline

instruction cycle. The Instruction Buffer (IB) is used to temporarily store the instruction fetched by the FU. The FU transfers the instruction into IB, and the EU receives the instruction from the IB. Figure 11.8 shows the timing diagram of the two stage instruction pipeline while executing four instructions i.e. from I1 to I4. We assume that the time taken by FU or EU for its operation is just one clock cycle. Each instruction is completed in two clock cycles. Hence, the first instruction is completed at the end of second clock. The second instruction is completed at the end of third clock cycle. From the previous figure, it is observed that the number of clock cycles needed to execute four instructions is five. Similarly, the number of clock cycles needed to execute seven instructions is eight. In a non-pipelined processor, eight clock cycles are needed to complete four instructions. The Intel 8088 and 8086 are designed with a two stage pipeline. The FU of 8088 fetches four bytes from the instruction stream whereas the 8086 fetches six bytes. Hence, the instruction buffer is a FIFO queue in which the FU stores the pre-fetched instructions.

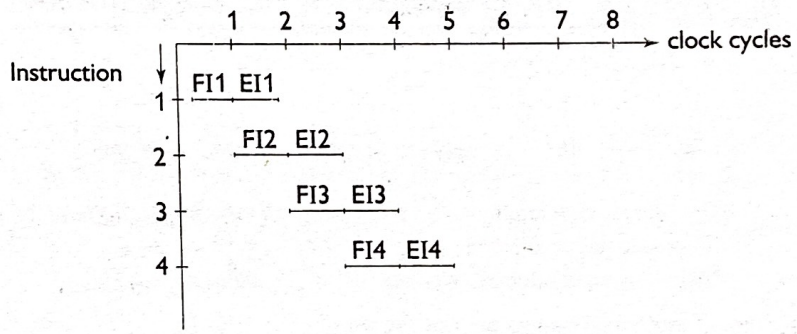


Fig. 11.8 Timing diagram for two-stage pipeline

Figure 11.9 exhibits a four stage instruction pipeline and Figure 11.10 shows the timing diagram for executing six instructions. Figure 11.11 presents a six stage instruction pipeline.

The following analysis establishes that time taken to complete m instructions in a n stage pipeline is approximately n.

Time taken for first instruction = nt_c where t_c is the duration of one clock cycle.

Time taken for remaining $(m - 1)$ instructions = $(m - 1)t_c$.

Total time taken for m instructions = $(nt_c) + (m - 1)t_c = (n + m - 1)t_c$.

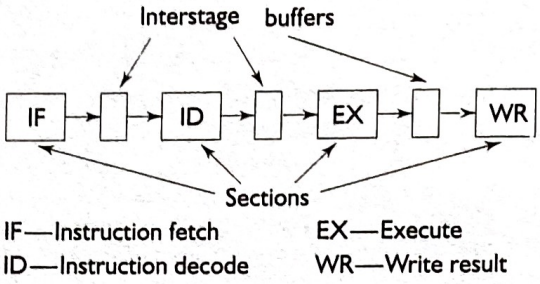


Fig. 11.9 A four-stage pipeline

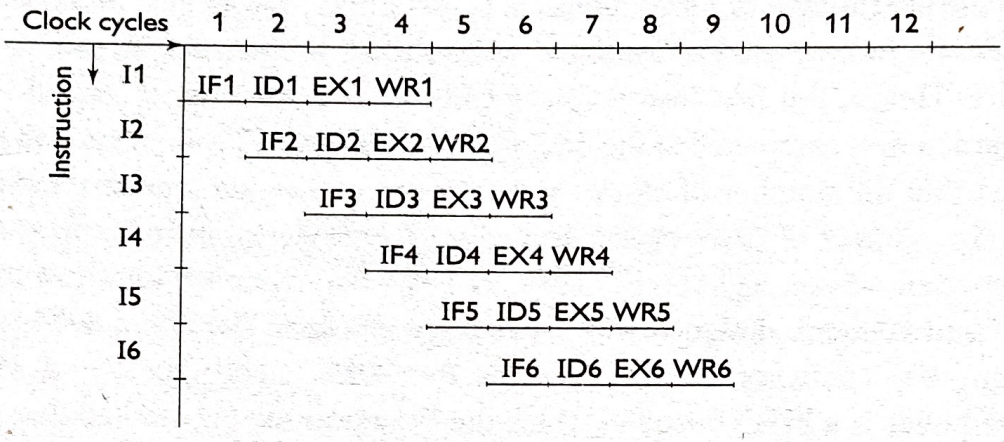


Fig. 11.10 Timing diagram for four-stage pipeline

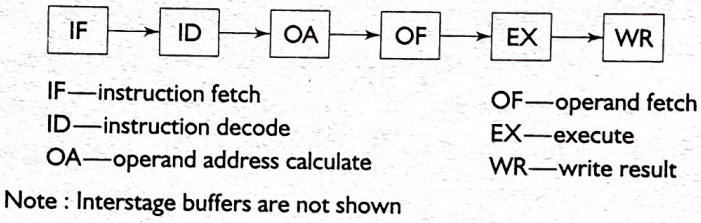


Fig. 11.11 A six-stage pipelined processor

If the processor is non-pipelined, the time taken for m instructions is nmt_c assuming the instruction cycle time is equal to nt_c.

Performance gain due to pipeline = time taken in non-pipelined mode/time taken in pipelined mode $= \dfrac{nmt_c}{(n+m-1)t_c} = \dfrac{nm}{n+m-1}$

For a large value of m, it is much larger than $n-1$. Hence $(n+m-1)$ approaches m. Hence, performance gain or speed-up $= nm/m = n$.

Thus, the theoretical maximum speed-up is equal to the number of stages in the pipeline. Number of clock cycles needed for executing m instructions is m since one instruction is completed per clock cycle.

In practice, the presence of branch instructions in the program reduce the speed-up gained due to pipelining and increase the number of clock cycles needed for m instructions.

Pipeline efficiency: Qualitatively speaking, the pipeline is not used to its maximum efficiency during initial filling time and end flushing time. The efficiency or utilization factor (E) of the pipeline is always less than 1. Hence, speed-up = nE.

Example 11.1 A six stage instruction pipeline has an utilization factor of 0.7. What is the speed-up due to pipelining?

$$\text{Speed-up} = nE$$
$$= 6 \times 0.7 = 4.2$$

Example 11.2 A program takes 500 ηs for execution on a non-pipelined processor. Suppose we need to run 100 programs of same type on a five stage pipelined processor with a clock period of 20 ηs. What is the speed-up ratio of the pipeline? What is the maximum achievable speed-up?

Time taken by one program on non-pipelined processor = $mnt_c = 500$ ηs.

Assuming the instruction cycle takes five clocks of each 20 ηs, $m = mnt_c/nt_c = 500/100 = 5$.

Hence, one program has five instructions. Number of instructions in 100 programs = 500 instructions.

Let us calculate the speed-up while running 100 programs in the pipelined processor.

Performance gain due to pipeline is = time taken in non-pipelined mode/time taken in pipelined mode = $\dfrac{nmt_c}{(n+m-1)t_c}$

Here, $n = 5$, $m = 500$, $t_c = 20$ ηs

$$\text{Speed-up} = (5 \times 500 \times 20)/(5 + 500 - 1) \times 20$$
$$= 50000/(504 \times 20)$$
$$= 4.96$$

Maximum speed-up = $n = 5$

11.5.2 Clock Frequency

The time taken for the completion of instruction steps at various stages is different. For example, if the instruction fetch needs more time than instruction decode, then, the clock

period of the pipeline should be not less than the time taken by the longest stage in the pipeline. As a result, those pipeline sections which complete their operations faster are forced to wait. To minimize this waiting time and achieve better efficiency from the pipelining, longer steps are split into more than one small stage so that time required for each stage is reduced. This permits a higher clock frequency. In addition, alternate solutions reduce the time taken for long activities. For instance, the use of cache memory reduces time needed for instruction fetch step. Another method includes multiple units operating parallelly for a single stage. In an ideal pipeline, time taken by each stage is same. This time is the duration of the clock period.

11.5.3 Pipeline Bubbles

To carry out certain instructions, some steps in the pipeline are irrelevant. This is known as 'Pipeline Bubble'. For some instructions, some of the steps in the pipeline are not to be done. For example, for a NOOP instruction, only two steps are needed: Instruction fetch and Instruction decode. Hence, this instruction causes 'bubble' in the pipeline after first two steps. Similarly, a LOAD instruction does not perform in WR stage. Nevertheless, such instructions have to go through all the stages without skipping the unwanted steps.

11.5.4 Pipeline Hazards

The pipeline operates with full efficiency as long as the program is an ideal one that does not interfere with the objective of completing one instruction for every clock. But a practical program has different types of inter-instruction relationships that result in stalling the pipeline for sometime. In other words, dependencies between successive instructions cause hazards in pipeline and affect the smooth operation of the instruction pipeline. The three major causes for the hazards are as follows:

1. Structural Hazard or Resource Conflict
2. Data Hazard or Data Dependency
3. Control Hazard or Branch Difficulty

11.5.4.1 Resource Conflict

When two different sections in the pipeline need the same hardware resource simultaneously, it results in the resource conflict. Some typical examples are as follows:

1. Suppose the final stage WR needs memory access for storing the result of the completed instruction, and at the same time, the OF also requires memory access for fetching the operand for the subsequent instruction. Obviously, one of them will be

delayed, which will affect the pipeline performance. If we delay the WR memory access, it will stall the pipeline operation for subsequent instructions till WR completes memory access. On the other hand, if we delay the OF memory access, partial operation of the instructions between OF and WR will proceed whereas the sections preceding OF will be frozen. One solution to this problem is the use of a 'write output buffer'. The WR stage simply stores the 'result' and 'other control information' (i.e. memory address) in the write buffer instead of accessing the main memory. In other words, the operation of storing in the main memory is removed from the pipeline. Additional control circuit performs a memory write operation when the memory is available by taking the information from the write buffer. Intel 80486 follows this technique. It has four output buffers that are common for both memory write and I/O write operations. Hence, up to four instructions are allowed to go out of pipeline when the bus is not available. If still, the bus is continuously not available, the pipeline is halted till the write buffer becomes at least partially empty.

2. Another resource conflict occurs when the IF stage requires memory access for the instruction fetch and OF stage needs memory access for the operand fetch at the same time. This can be resolved by having separate cache memory units for the instructions (code cache) and operands (data cache) as in the case of Pentium microprocessor. Simultaneously, both code cache and data cache can be accessed.

11.5.4.2 *Data Dependency*

If the operand for an instruction is the result of the previous instruction, which is not yet completed in the pipeline, it is known as data dependency. Figure 11.12 shows pipeline

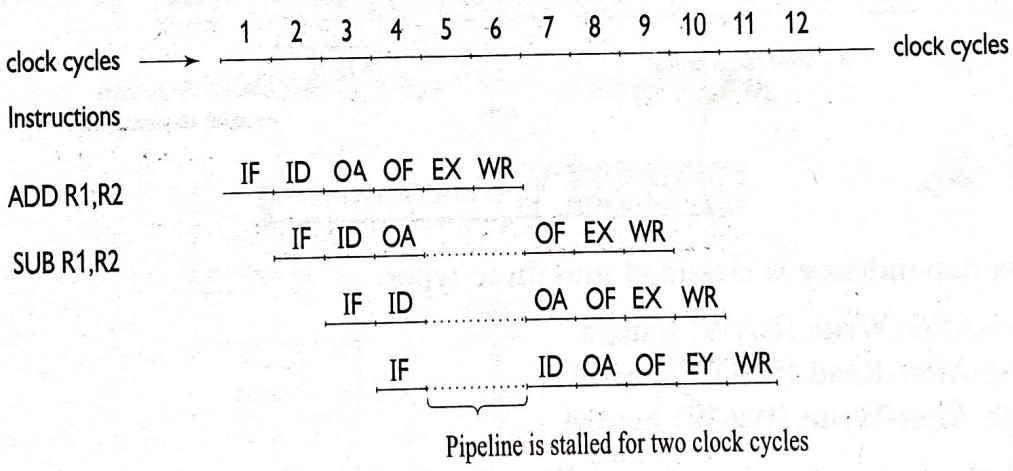


Fig. 11.12 Pipeline stalling due to data dependency—six stage

stalling due to the non-availability of the operand for the OF stage of the second instruction since the first instruction is not yet completed. Nothing is done in clock cycles 5 and 6 for second and subsequent instructions. Hence, penalty has to be paid due to data dependency on the two clock cycles. Figure 11.13 shows similar effect in the four stage pipeline. A standard solution to the data dependency is known as 'operand forwarding'. Figure 11.14 illustrates this concept. The result of Execute (EX) section is the feedback to the input of the same section so that it is easily available (before the required time) for use in the subsequent steps for the next instruction. The control unit enables this path at appropriate time.

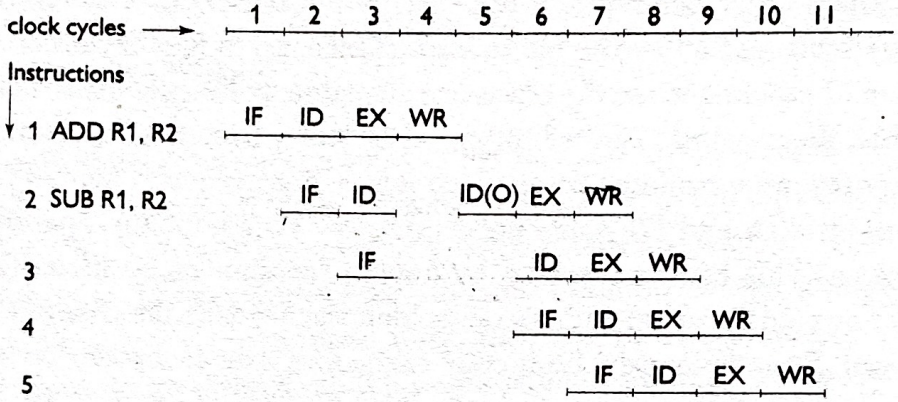


Fig. 11.13 Pipeline stalling due to data dependency—four stage case

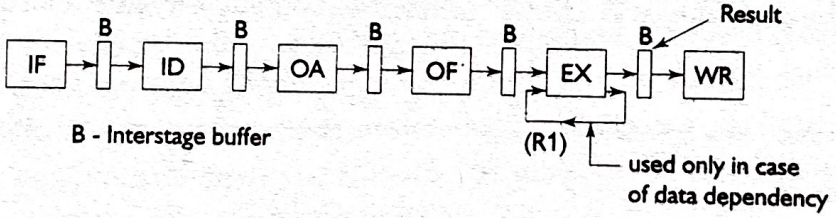


Fig. 11.14 Operand forwarding

The data dependency is classified into three types:

1. Read-After-Write (RAW) hazard
2. Write-After-Read (WAR) hazard
3. Write-After-Write (WAW) hazard

Between the two instructions A and B occurs the 'RAW' hazard if B reads some data item that is modified by A. A 'WAR' hazard occurs when B modifies some data item that

is read by A. A 'WAW' hazard occurs if both A and B modify some data item. The advanced detection of the hazardous situations and commencement of the correctives to prevent the hazards is usually carried out by the appropriate hardware logic.

11.5.4.2.1 Software Solution to Data Hazard

Here, the compiler senses the hazard by scanning the instructions in the program. In case of data hazard, it inserts a NOOP instruction in between the two instructions, which are data dependent. As a result, a delay of two or more clock cycles occurs and resolves the data dependency. The merit of this method is the usage of simpler hardware and the freedom to insert some useful instructions in the NOOP slots wherever possible by the compiler. However the program size increases with the introduction of the NOOP instructions.

11.5.4.3 Branch Difficulty

When a branch instruction enters the pipeline, it is recognized during the instruction decode step. Special action is performed by the processor for unconditional and conditional branch instruction, see Fig. 11.15. Figure 11.16 illustrates a four stage

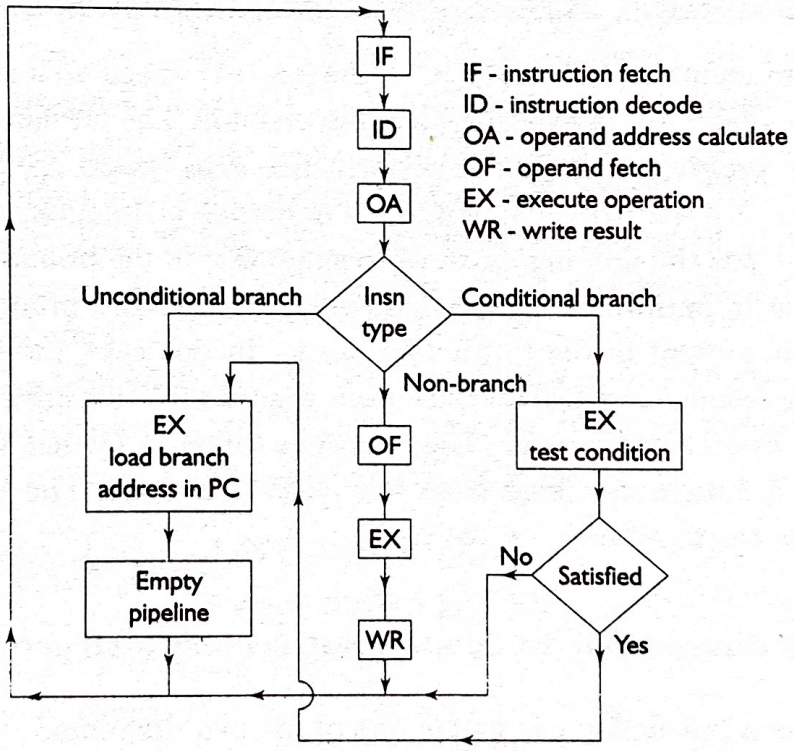


Fig. 11.15 Effect of branch instruction

pipeline an unconditional branch instruction. We have two clocks penalty due to this instruction. In clock 4, branch address is loaded in PC. Hence, the steps carried out in clocks 3 and 4 for third and fourth instructions are flushed. In clock 5, new instruction is fetched from the branch address. If there are no branch instruction, then the clock 5 performs IF5, ID4, and EX3. The wastage due to unconditional branch instruction results in the actions IF3, ID3 and IF4 which are completed but not utilized. In other words, the branch instruction causes $(n - 1)$ additional clocks delay.

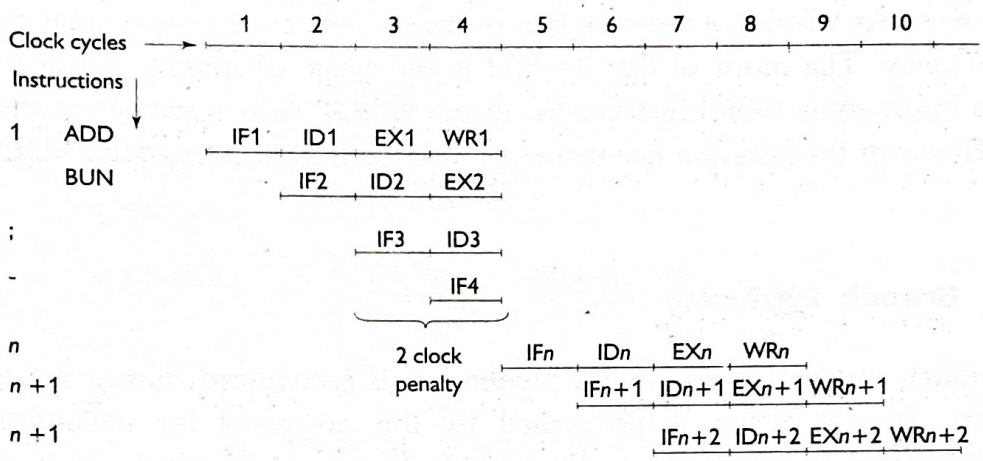


Fig. 11.16 Effect of branch penalty—unconditional branch

The branch instruction results in aborting of the partially executed instructions by flushing them out of the pipeline and a fresh refill of the pipeline. The presence of many branch instructions in the program reduces the performance gain (speed-up) achievable by instruction pipelining. If p is the fraction of number of branch instructions to total number of instructions, then $1/p$ is the limiting factor of speedup due to the branch instructions.

A special action is performed by the hardware to sense the branch instructions in advance when it is present in the instruction queue. In this case, the instruction in the branch address are fetched earlier than usual time. Figure 11.17 illustrates the effect of this on unconditional branch instruction. The branch address is loaded in PC in clock 2 whereas in clock 3, instruction fetch from this address is done. The branch penalty is reduced to just one clock cycle.

11.5.4.4 Delay Caused Due to Conditional Branch Instruction

A conditional branch instruction can create one of the two situations:

1. The condition tested is successful; here, the contents of the pipeline (instructions following the branch instruction that have been partly executed) are flushed out and the instruction at the branch address (and following instructions) enter the pipeline.
2. The condition tested is unsuccessful and hence there is sequential execution; in this case, the pipeline contents are not affected.

The delay caused by conditional branch instruction is well-explained in the following example.

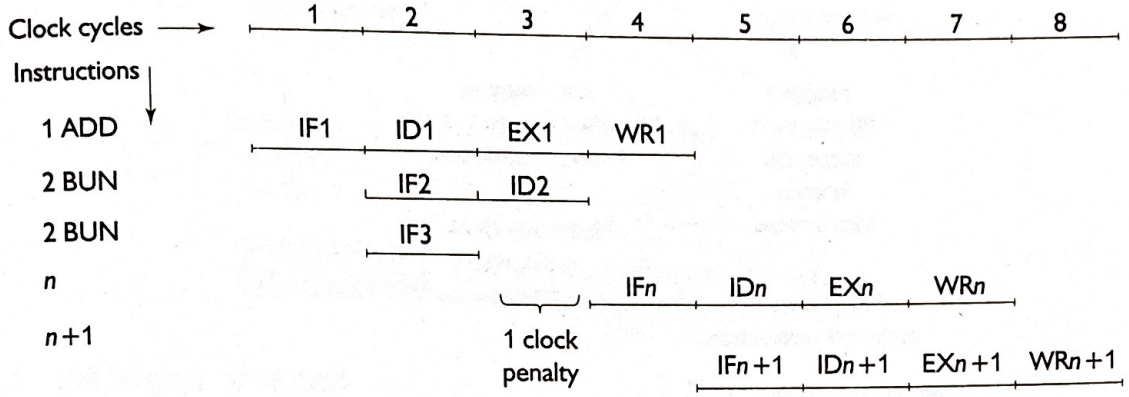


Fig. 11.17 Effect of branch penalty—use of additional logic

Suppose there are m instructions to be executed. Let p be the probability of the conditional branch instruction and q be the probability for the success of the branch. Hence, number of instructions causing successful branch is mpq (Figure.11.18).

In a pipelined processor, the instructions are completed at the rate of one instruction per clock cycle. If there are no branch instruction in the stream, the program routine is completed in m clocks. Each branch instruction results in flushing the pipeline and refilling from the branch address. The instruction from the branch address (target address) needs n clock cycles for execution.

Total clock cycles needed to execute the m instructions = (time delay in clock cycles introduced by branch instructions) + (clock cycles needed for non-branch instructions)

$$= mpqn + <(m - mp) + mp - mpq)$$

$$= mpqn + <(m - mpq)>$$

$$= mpqn + m(1 - pq)$$

Thus, m instructins are executed in $mpqn + m(1 - pq)$ clock cycles.

Average number of Clocks Per Instruction (CPI) $= 1 + pq\,(n - 1)$

If the program causes no branching, $q = 0$; CPI $= 1$.

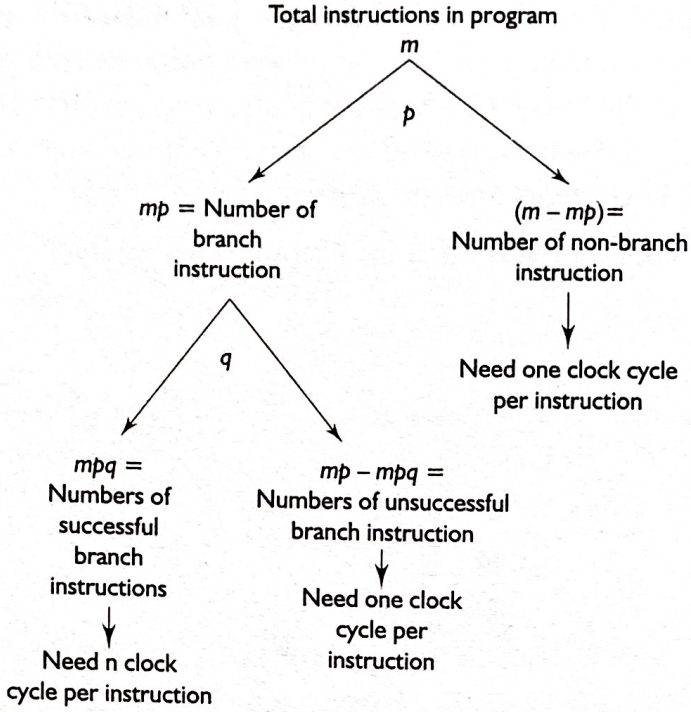


Fig. 11.18 Addition time involved due to branch

Example 11.3 A program has 20% branch instructions. Assuming all the branches are successful, calculate the speed-up, in a pipelined processor with six stages.

If there are no branch instructions in the program, the speed-up = n = no. of stages = 6

$$p = 0.2,\ q = 1.0,\ n = 6$$

$$\text{CPI} = 1 + pq(n-1) = 1 + (0.2 \times 1.0 \times 5) = 2.0$$

A non-pipelined processor needs 6 clocks per instruction.

$$\text{Speed-up} = 6/2 = 3.0$$

Example 11.4 A program has 20% branch instructions. When it is run on a five stage pipelined processor, it was observed that 60% of the branches were successful. Calculate performance improvement in the pipelined processor. What is the maximum speed-up possible if there are no branches?

$$p = 0.2,\ q = 0.6,\ n = 5$$

$$\text{CPI} = 1 + pq(n-1) = 1 + (0.2 \times 0.6 \times 4) = 1.48$$

A non-pipelined processor needs 5 clocks per instruction.

$$\text{Speed-up} = 5/1.48 = 3.38$$

Maximum speed-up in no-branch case = $n = 5$.

11.5.5 Minimizing Impact of Conditional Branch Instructions

Several techniques are followed in different processors for minimizing the delay due to conditional branch instructions see Fig. 11.19. The delayed branch is a software technique which is discussed in the following paragraphs whereas others are hardware techniques explained in Chapter 12 under section 'superscalar processors'.

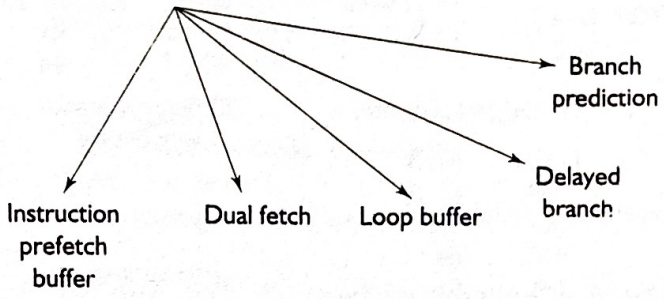


Fig. 11.19 Branch handling techniques

11.5.5.1 Delayed Branch

The instruction following the branch instruction acquires a delay slot. Accordingly, there are two delay slots in Fig. 11.16 and one delay slot in Fig. 11.17. The compiler tries to rearrange the instructions wherever possible (without affecting the program logic) so that some unrelated instructions are moved after the conditional branch instruction that can be continued in the pipeline. If the situation is such that no other instruction can be brought after the branch instruction, then the compiler inserts a NOOP instruction after the branch instruction. Figure 11.20 illustrates both these cases.

11.5.5.2 CASE STUDY 2: Conditional Branch Handling in Intel 80486

The pipelined CPU takes special care in handling a conditional branch instruction. For the decoding of a conditional branch instruction, there are two possible options of design implementation for the pipeline control:

1. Continue fetch and decoding of subsequent instructions (assuming no branch) until the result of condition testing is known.
2. Fetching (in advance anticipation) the instruction from the branch destination address (expecting a branch).

Depending on the results of the condition testing, dropping of the already fetched and restart of a new fetch may be necessary in both cases. This leads to rework for the DO loop

like situations in the program. To minimize this, Intel 80486 and Pentium follow various different strategies.

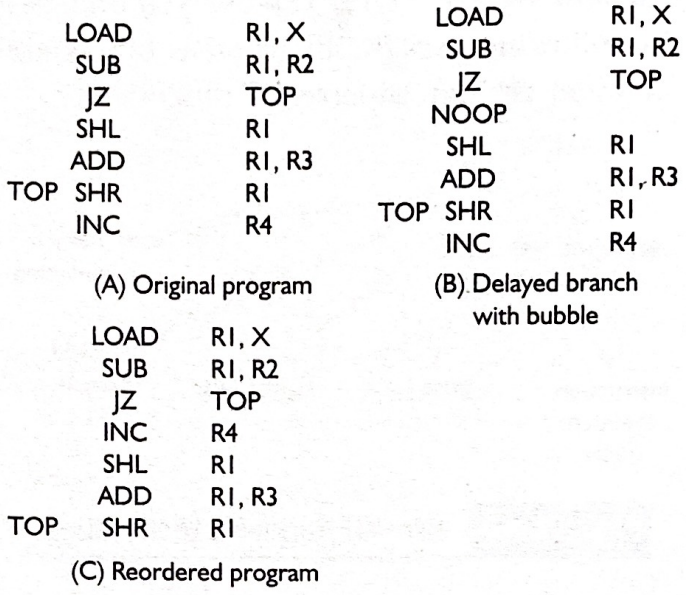


Fig. 11.20 Delayed branch by compiler optimization

11.5.5.3 *80486 and Conditional Branch*

The pipeline stages of 80486 are displayed in Fig. 11.21. The 80486 follows the *cat-on-the-wall* strategy for conditional branch instruction. When a branch instruction is decoded, the pipeline continues as if the non-branch instruction is decoded. But when the branch instruction reaches the EX stage, the pipeline fetches the instruction from the branch address. The fate (to branch or not to) is known only at the end of EX stage when the condition testing is carried out. However, in both the cases, 486 is in a comfortable situation:

1. If branch does not occur, the prefetched instruction from the branch address is thrown out. The pipeline proceeds without any penalty (delay).
2. If branch occurs, the pipeline starts decoding the prefetched instruction from the branch address and throws out the sequential instructions following the branch instruction. This situation has two clocks delay.

11.6 Vector Computing

A vector processor is used for high-performance scientific computing, where matrix and vector arithmetic are common. Some examples are:

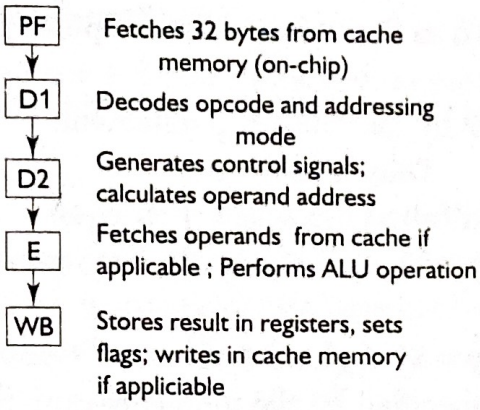


Fig. 11.21 Instruction pipeline in 80486

- Weather forecasting
- Space flight simulations
- Image processing
- Remote sensing

These are some of the applications that perform some basic set of operations repeatedly on a large amount of data. The traditional processors used for such computations took several days. However, this type of applications is easily vectorizable and executable on a vector processor. The Cray Y–MP and the Convex C3880 are two examples of vector processors used nowadays.

A vector processor efficiently handles arithmetic operations on elements of arrays, known as the 'vectors'. For a single vector instruction, the vector processor simultaneously performs multiple homogeneous operations on an array of elements. The vector processor uses data parallelism by compact parallel datapath structures with multiple functional units operating for the same vector instruction. Since a single instruction specifies a whole set of operations, time spent on instruction fetch and decode operations is reduced compared to the traditional processor which needs a series of instructions.

11.6.1 Vector Arithmetic

A vector, v, is a list of elements

$v = (v1, v2, v3, ..., vn)$,

where n is the number of elements in the vector. In a computer program, the vector is an array of one dimension. The terms vector, array and list are used interchangeably. For example, in Fortran, we declare v by the following statement:

$$\text{Dimension} = v(N)$$

where N is an integer variable indicating the *length of the vector.*

Arithmetic operations can be performed on vectors. Two vectors are added by adding the corresponding elements:

$$Z = x + y = x1 + y1, x2 + y2, ..., xn + yn .$$

In Fortran, vector addition is specified by the following routine:

```
DO 300 I = 1, N
300 Z(I) = x(I) + y(I)
```

where Z is the vector for the sum and Z, x, and y are known as the arrays of dimension N. This operation is called *elementwise* addition. For traditional processor (scalar), the compiler translates the previous program into the following sequence of instructions (furnished as assembly language statements):

```
      INITIALISE I = 0
300   READ x( I )
      READ y( I )
      READ N
      ADD Z(I) = x( I ) + y( I )
      INCREMENT I = I + 1
      IF I ≤ N GO TO 300
      CONTINUE
```

While executing the program, the scalar processor performs the addition instruction, n times, in order to add n elements of array x with corresponding elements of array y. The loop is encountered n times. If n is more, time spent on instruction fetch and decoding is significant. In the case of the vector processor, a single vector instruction covers all the n items as indicated previously.

11.6.2 Vector Computing Concepts

A vector processor contains a set of arithmetic *pipelines.* These pipelines overlap the execution of the different parts of an arithmetic operation on the elements of the vector. This helps in efficient execution of the arithmetic operation. Let us examine how a vector pipeline operates.

11.6.2.1 Floating-point operation

Consider the following steps for a floating-point addition on a traditional processor: $Z = x + y$.

1. *Exponent Comparison*: The exponents of the two floating-point numbers are compared to find the number with the smaller magnitude.
2. *Mantissa Alignment*: The mantissa of the number with the smaller magnitude is shifted so that the exponents of the two numbers match.
3. *Mantissa Addition*: The mantissas are added.
4. *Normalization*: The result of the addition is normalized.
5. *Exception Testing*: Checking is done to detect if any floating-point exceptions such as overflow occurred during the addition.
6. *Result Rounding*: Rounding off the result is done.

Figure 11.22 presents the step-by-step progress of such a scalar addition. The numbers considered are $x = 1328.00$ and $y = -742.6$. For simplicity sake, these are represented in decimal notation with a mantissa of four digits.

Step→	1	2	3	4	5	6
x	0.1328E4	0.13280E4				
y	−0.7426E3	−0.07426E4				
z (x+y)			0.058540E4	0.58540E3	0.58540E3	0.58540E3

Fig. 11.22 Six stages of floating-point addition

Suppose this addition has to be performed on all the elements of a pair of vectors (arrays) of length n. All the six stages are executed for every pair of elements. If each stage of the execution takes δ units of time, then each addition takes 6δ units of time (excluding the time taken to fetch and decode the instruction or fetching the two operands). Hence, the time taken to add the elements of the two vectors in a serial fashion is $T_s = 6n\delta$. Figure 11.23 displays the execution stages with respect to time.

11.6.2.2 Arithmetic Pipeline

Suppose the floating-point addition operation is pipelined by six different arithmetic sections, then each stage of the addition is performed at each section in the pipeline. Each section of the pipeline consist of a separate arithmetic unit designed for the operation to be

performed at that stage. Once stage 1 is completed for the first pair of elements, the output is transferred to the next section 2 while the second pair of elements enters into the first section 1. Assume each section takes δ units of time. The data flow through the pipeline stages with respect to time is shown in Fig. 11.24.

Time	δ	2δ	3δ	4δ	5δ	6δ	7δ	8δ
Step								
1	x1+y1						x2+y2	
2		x1+y1						x2+y2
3			x1+y1					
4				x1+y1				
5					x1+y1			
6						x1+y1		

Fig. 11.23 Scalar floating-point addition of vectors

It is obvious that it takes same amount of 6δ units of time to obtain the sum of the first pair of elements. But the sum of every successive pair is available at intervals of δ units of time. Hence the time, T_p, to do the pipelined addition of two vectors of length n is

$$T_p = 6\delta + (n-1)\delta = (n+5)\delta$$

Time	δ	2δ	3δ	4δ	5δ	6δ	7δ	8δ	9δ	10δ	11δ
Step											
1	x1+y1	x2+y2	x3+y3	x4+y4	x5+y5	x6+y6	x7+y7	x8+y8	x9+y9	x10+y10	x11+y11
2		x1+y1	x2+y2	x3+y3	x4+y4	x5+y5	x6+y6	x7+y7	x8+y8	x9+y9	x10+y10
3			x1+y1	x2+y2	x3+y3	x4+y4	x5+y5	x6+y6	x7+y7	x8+y8	x9+y9
4				x1+y1	x2+y2	x3+y3	x4+y4	x5+y5	x6+y6	x7+y7	x8+y8
5					x1+y1	x2+y2	x3+y3	x4+y4	x5+y5	x6+y6	x7+y7
6						x1+y1	x2+y2	x3+y3	x4+y4	x5+y5	x6+y6

Fig. 11.24 Pipelined floating-point addition of vectors

The first 6δ units of time includes the time to fill the pipeline and the time to obtain the first result. After receiving the last result, the pipeline is flushed.

The pipeline mode of the addition is faster than the serial mode by nearly the number of stages in the pipeline. If n is large, the pipeline mode addition is about six times faster than scalar addition.

The number of stages in a floating-point addition differs in different processors—more or less than six. The operations for floating-point multiplication are slightly different from operations for addition; also the number of sections in a multiplication pipeline is different from an addition pipeline.

11.6.2.3 Vector Registers

Most vector processors have *Vector registers.* In comparison to a general purpose or a floating-point register that stores a single value, a vector register stores several elements of a vector, simultaneously. The contents of a vector register are transferred to the vector pipeline one element at a time.

11.6.2.4 Scalar Registers

A Scalar register, like general purpose or floating-point registers, stores a single value. But it is configured to be used by a vector pipeline. The value in the register is read once every δ units of time and released into the pipeline, which is similar to receiving a vector element from the vector pipeline. This facilitates the operation on the elements of a vector by a scalar. For example, consider the computation:

$$j = 3.4 \times i$$

The constant 3.4 is stored in a scalar register and transmitted into the vector multiplication pipeline every δ units of time in order to participate in the multiplication of each element of i.

11.6.2.5 Chaining

Generally vector processors have multiple pipelines which are of different types. In some vector processors, the output of one pipeline is directly released into another pipeline. This technique is called chaining. It eliminates the intermediates storage for the result of the first pipeline before transmitting it into the second pipeline. Figure 11.25 displays the use of chaining in the computation of following vector operation:

$$K(x + y)$$

where x and y are vectors and K is a scalar constant. Chaining doubles the number of floating-point operations that are performed in δ units of time. Once both the multiplication and addition pipelines are filled, two floating-point operations (one multiplication and one addition) are completed every δ time units. It is possible to chain more than two functional units but it involves complex timing considerations.

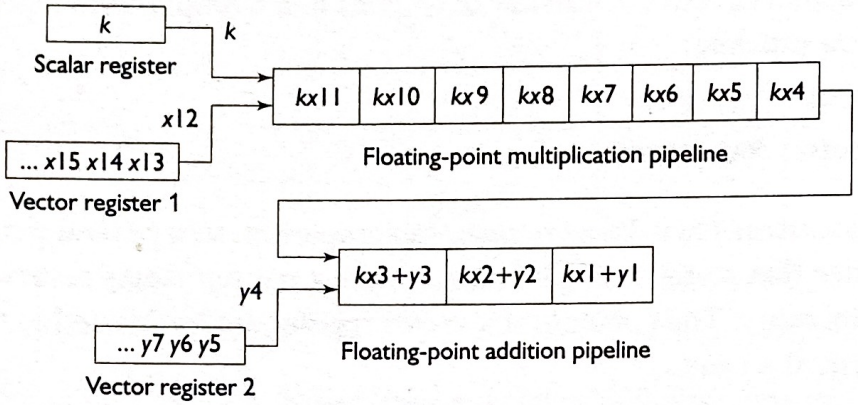


Fig. 11.25 Chaining to compute $kx + y$

11.6.2.6 Scatter and Gather Operations

Suppose only certain elements of a vector are required for computation. The vector processor picks up the appropriate elements (a *Gather* operation) and joins into a vector or a vector register. If the elements used are in a regularly-spaced pattern, then the spacing between the elements to be gathered is called the *stride.* For example, if the elements:

$$x1, x6, x11, x16, ...,$$

are to be extracted from the vector:

$$(x1, x2, x3, x4, x5, x6, ..., xn)$$

for some vector operation, the stride is 5. A *Scatter* operation reformats the output vector so that the elements are spaced correctly.

11.6.2.7 Vector–Register Vector Processors

A vector–register processor fills the vector pipelines from the vector element values currently in the vector registers. This reduces the time to fill the pipelines (the startup time) for vector arithmetic operations; the vector registers are be filled while the pipelines are performing some other operation. The vector results are put back into a vector register after the completion of the operation, or they may be piped directly into another pipeline for an additional vector operation (chaining).

In these processors, arithmetic or logical vector operations are performed only on vectors that are already in the vector registers. A *load vector* operation reads the elements of the vector from the memory into the vector register. The vector result of a vector operation is stored in a vector register. It is stored in the memory by a *store vector* operation. This operation can be overlapped with other operations. For subsequent requirements of the result vector, it is read from the vector register.

11.6.2.7.1 Memory–Memory Vector Processors

The memory–memory vector processor has no vector registers. It fetches vectors directly from memory to fill the pipelines and stores the pipeline results directly to memory. The startup time for the vector pipelines is more because it takes more time to start a vector operation due to increased memory access time. One example of a memory–memory vector processor is the CDC Cyber 205.

Due to reduced vector access time and the overlap between store vector and other opeartions, vector–register vector processors are usually more efficient than memory–memory vector processors. However, if the vectors are long, the difference in efficiency is insignificant. On the other hand, the memory–memory vector processors are attractive if the vectors are extremely long.

11.6.2.8 Interleaved Memory Banks

To reduce the access time for vector elements stored in memory, the memory of a vector processor is usually interleaved into multiple *memory banks. Successive* memory banks possess successive memory addresses cyclically. In a k way interleaved memory, there are k number of memory banks. The word $\overline{0}$ is stored in bank $\overline{0}$, word $\overline{1}$ is in bank $\overline{1}$, ..., word $k-\overline{1}$ is in bank $k-\overline{1}$, word k is in bank $\overline{0}$, word $k+\overline{1}$ is in bank $\overline{1}$, ..., etc. When the elements of a vector are read from the interleaved memory, the reads are staggered across the memory banks so that one vector element is read from a bank per clock cycle. If one memory access takes n clock cycles, then n elements of a vector may be fetched in one memory access; this is n times faster than the same number of memory accesses to a single bank.

11.6.3 Vector Computing Performance

For most vector processors, the time to complete one pipeline stage (δ) is equivalent to one clock cycle of the processor. (In some processors, it is equal to two or more clock cycles.)

Once a pipeline is filled, it generates one result for every δ units of time, that is, for each clock cycle. In other words, the hardware performs one floating-point operation per clock cycle.

Let k be the number of δ time units needed by the same sequential operation (or the number of stages in the pipeline). Then the time to execute that sequential operation on a vector of length n is:

$$T_s = kn\delta$$

and the time to perform the pipeline mode is

$$T_p = k\delta + (n-1)\delta = (n+k-1)\delta$$

As before, for $n > 1$, $T_s > T_p$.

Startup time: The startup time is the time needed to initiate the operation. In the traditional sequential machine, there are overheads to set up a loop to repeat the same floating-point operation for an entire vector. Also, the elements of the vector are fetched from memory. If S_s is the number of δ time units for the sequential startup time, then T_s is presented as:

$$T_s = (S_s + kn)\delta$$

In a pipelined machine, the flow from the vector registers or the memory to the pipeline has to initiate; this time is S_p. The overhead cost for $k\delta$ time units is the time required to initially fill the pipeline. Hence, T_p must include the startup time for the pipelined operation; thus,

$$T_p = (S_p + k)\delta + (n-1)\delta$$

or

$$T_p = (S_p + k + n - 1)\delta$$

Since, the length of the vector is very large, (as n goes to infinity), the startup time becomes negligible in both cases. Hence,

$$Ts \sim kn\delta$$

while

$$Tp \sim n\delta$$

Thus, for large n, T_s is k times larger than T_p.

Table 11.2 lists some typical vector computers. Most of the supercomputers are vector computers which offer very high performance for special applications. However, they are not suitable for standard general purpose applications.

TABLE 11.2 Comparison of vector computers

Vector computers	Types	Additional remarks
CDC Star 100	A supercomputer with built-in vector processor	Based on APL programming language
TI ASC	Suitable for long vectors	—
CDC Cyber 205	A memory-memory vector processor	Has four general-purpose pipelines; it also provides both gather and scatter operations
Alliant FX/8	A shared-memory multiprocessor with eight CPU's, each with an attached vector processor	—
Cray-1	Supercomputer with pipelined vector arithmetic units; A vector-register processor	Provides scatter and gather operations; uses chaining, has 12 different pipelines or *functional units*; each vector register contains 64 single elements; each element or word contains 64 bits
Cray X-MP	Shared-memory multiprocessor with each CPU controlling its own set of vector processors; more support for overlapped operations along with multiple memory pipelines	Chaining of all three floating-point pipelines is allowed
Cray-2	A multiprocessor with up to four processors	Does not support chaining

11.6.4 CASE STUDY 3: Cray-1

Cray-1 is a supercomputer with pipelined vector arithmetic units. It is a vector–register machine. Each vector register stores 64 elements. Each element or word has 64 bits. The Cray-1 has 12 functional pipelined units. All the 12 units can operate concurrently. The Cray-1 is the first vector processor to apply chaining technique. These are four groups of functional units:

- *Vector pipelines* that perform integer or logical operations on vectors.
- *Floating-point pipelines* that execute floating-point operations using scalars or vectors.
- *Scalar pipelines* that carry out integer or logical operations on scalars.
- *Address pipelines* that perform address calculations.

Figure 11.26 illustrates the functional units and registers. There is no floating-point divide unit. A floating-point reciprocal approximation pipeline is used for floating-point division, that is, x/y is computed as $x(1/y)$.

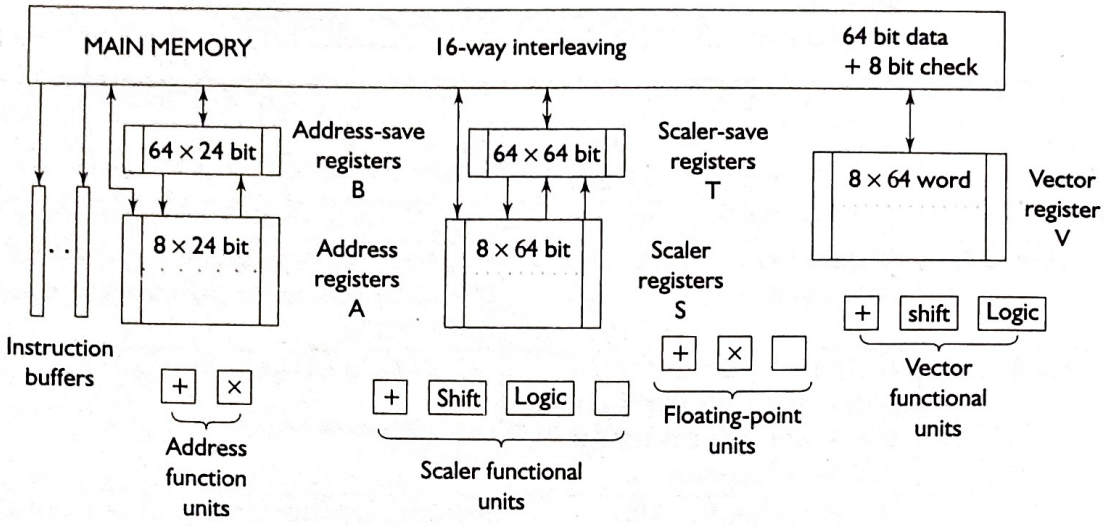


Fig. 11.26 Cray-1 Functional units and registers

There are mainly five types of programmable registers: A, B, S, T and V. In addition, there some supporting registers: Vector Length register (VL), Vector Mask register (VM), Program counter (P), Base Address register (BA), Limit Address register (LA), Exchange Address register (XA), Flag register (F) and Mode register (M). The 'A' registers are known address registers. They work as address registers for memory reference and as index registers. Also, these are used for shift counts and loop counts. The 'B' registers are called as address-save registers and are used as auxiliary storage (buffer) for the 'A' registers. The 'S' registers are known as scalar registers and are used as source and destination registers for scalar arithmetic and logical instructions. The 'T' registers are known as scalar-save registers which are used as auxiliary storage for the 'S' registers. The 'V' registers are known as vector registers and are used as source and destination registers for the vector functional units (pipelines).

The Cray-1 allows one memory read and write per clock cycle. Hence, it can read only one vector and write one vector result at the same time. When more than one read (or write) is required for the same operation, one of the operations is postponed.

Consider the multiplication of two vectors of length greater than 64:

$$z = x \times y$$

Initially two vector registers are loaded from memory: one the first 64 elements of x; and the other, the first 64 elements of y. The result for z enters a third vector register. After the first 64 elements are covered, the source vector registers are reloaded from memory and the result vector register contents are stored in memory. Since only one read and one write are executed per cycle, elements of both input vectors cannot be read at the same time; so the pipeline delays one of the read operations.

The main memory is a 16-way interleaved memory. It is a SEmiConDuctor memory with Error Detection and correction (SECDED) logic. There are 12 I/O channels.

The Cray-1's Fortran compiler is an optimizing compiler. The programmers need not modify their old source programs since the compiler takes care of 'vectorization' of DO loops and generates vector instructions.

The Cray-1 is the first of a series of Cray supercomputers.

11.7 Array Processor

An array processor performs simultaneous computations on the elements of a vector (an array or a table of data in multi-dimensions). It executes a sequence of necessary actions for an operation on the entire array. The multiple units in the array processor perform the same operation on different data. The common usage of the array processors include analyses of fluid dynamics and rotation of 3D objects, weather data processing as well as data retrieval, in which the elements of a database are scanned simultaneously, signal processing and applications in which differential equations, matrix manipulations or linear algebra are involved. Often people fail to differentiate between array processor and vector processor. A vector processor does not operate on all the elements of a vector at the same time, it only operates on the multiple elements concurrently. The array processors are classified into two types:

1. Dedicated Array Processor
2. Attached Array Processor

The *Dedicated array processor* is a stand alone computer system whereas the *Attached array processor* is attached to another general purpose computer system as an extension. ILLIAC IV and STARAN are two popular array processors. Figure 11.27 illustrates the interfacing of an attached array processor to a general computer. The FSP–164/MAX is an attached processor for VAX 11 system. Its attachment (interfacing) to another general purpose computer adds vector architecture to the general purpose computer. The dedicated array processor falls under SIMD architecture. Hence, it is generally referred as SIMD array processor.

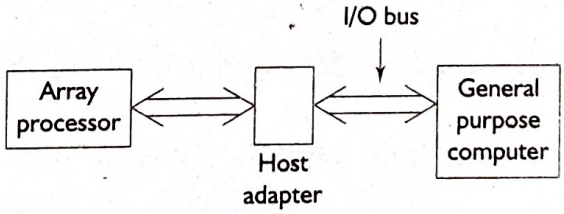


Fig. 11.27 Attaching an array processor

11.7.1 CASE STUDY 4: ILLIAC IV—a SIMD Array Processor

The ILLIAC IV is a special purpose array processor. Its initial design had 256 Processing Elements (PE) organized as four quadrants of 64 PEs. The 64 PE quadrant is an 8 by 8 array of PEs. Each PE is linked to its neighbors in the four directions. Each PE has 2K words of memory. The central Control Unit (CU) broadcasts instructions that are executed by the PEs. The CU is a highly complex and sophisticated unit with its own resources for executing certain scalar instructions. In other words, the CU is equivalent to a processor. Figure 11.28 illustrates the structure of ILLIAC IV. All the PEs perform the instruction operation (say ADD) simultaneously on different components of vectors. The ILLIAC IV is linked to B6500 control computer that acts as the link between the user and ILLIAC IV. Logically, the ILLIAC IV is linked as a peripheral subsystem to B6500.

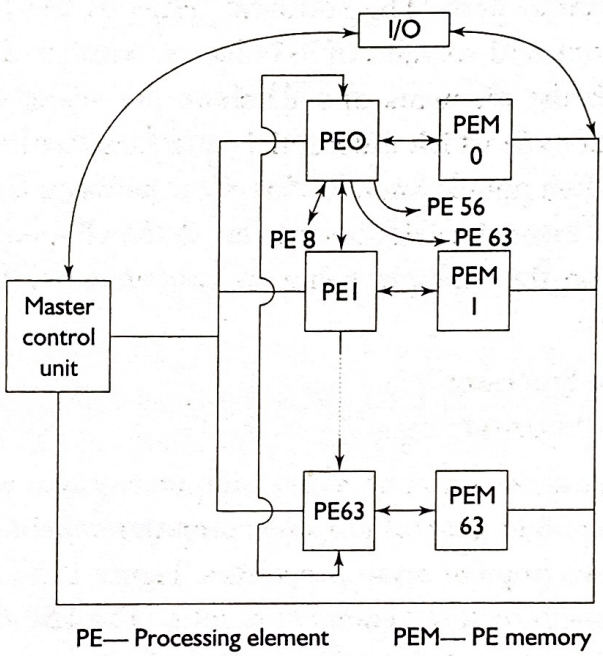


Fig. 11.28 ILLIAC IV array processor

SUMMARY

There are two basic strategies which are followed to enhance the performance of a computer:

1. *Overlap:* Splitting a task into multiple subtasks that can be performed in an overlapped manner.
2. *Parallelism:* Executing more than one task in parallel.

In *instruction level parallelism*, parallelism is applied in a single processor whereas in *processor level parallelism*, multiple processors share the workload (tasks). This can be achieved either by using more than one processor in a single computer system or by distributing the tasks among the multiple computer systems linked either as a cluster or a network.

Flynn's classification of computer organization into four different types is as follows:

Single Instruction stream, Single Data stream (SISD)
Single Instruction stream, Multiple Data stream (SIMD)
Multiple Instruction stream, Single Data stream (MISD)
Multiple Instruction stream, Multiple Data stream (MIMD)

A *simple scalar processor* performs only one arithmetic operation at a time. It executes one instruction at a given time and each instruction has one set of operands. In a *vector processor*, each instruction has multiple sets of operands such as elements of two arrays and the same operation is performed simultaneously on different sets of operands. In a *non-pipelined (sequential) scalar processor*, there is no overlap of successive instruction cycles. In a *pipelined processor*, multiple instructions operate at various stages of the instruction cycle simultaneously, in different sections of the processor. An *array processor* has multiple execution units that operate simultaneously on multiple elements of a vector. A *superscalar processor* is a scalar processor that performs multiple instruction cycles simultaneously for successive instructions.

The pipelining is a technique of splitting one task into multiple subtasks and executing the subtasks of the successive tasks in parallel. An instruction pipeline splits an instruction cycle actions into multiple steps that are executed one-by-one in different sections of the processor. An arithmetic pipieline splits an arithmetic operation into multiple arithmetic steps each of which are executed one-by-one in different arithmetic sections of the ALU. The branch instructions in the program reduce the speedup gained due to pipelining and increase the number of clock cycles needed for m instructions. The dependencies between the successive instructions cause hazards in the pipeline and affect smooth operation of the instruction pipeline. The following are three major causes of hazard:

1. Structural Hazard or Resource conflicts
2. Data Hazard of Data dependency
3. Control Hazard or Branch difficulty

A vector processor efficiently handles arithmetic operations on elements of arrays called vectors. For a single instruction, the vector processor simultaneously performs multiple homogeneous operations on an array of elements. It uses the data parallelism by parallel datapath structures with multiple functional units operating fo the same vector instruction.

An array processor performs simultaneous computations on the elements of a vector (an array or table of data in multi-dimensions). It executes a sequence of actions necessary for an operation on an entire array. The multiple units in the array processor perform the same operation on different data. The dedicated array processor is a stand-alone computer system whereas the attached array processor is attached to another general purpose computer system as an extension.