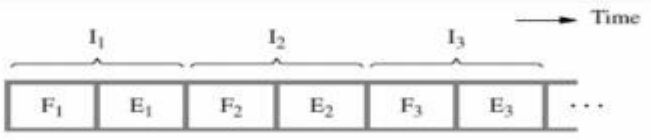
V UNIT

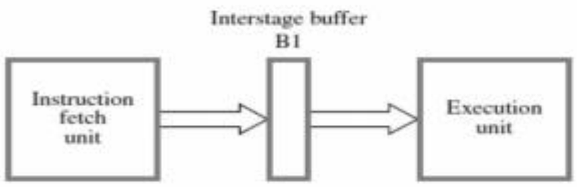
PIPELINING

BASIC CONCEPTS:

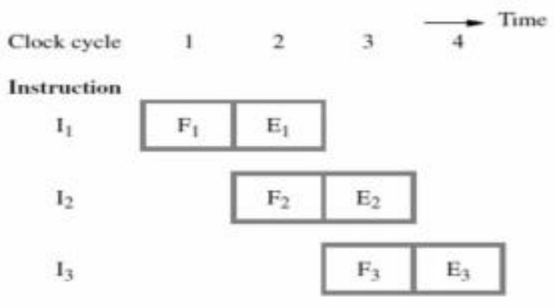
Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. The processor executes a program by fetching and executing instructions, one after the other. Let Fi and Ei refer to the fetch and execute steps for instruction Ii . Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure a. Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure b. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction. We assume that both the source and the destination of the data operated on by the instructions are inside the block labelled "Execution unit."



(a) Sequential execution



(b) Hardware organization



(c) Pipelined execution

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure c. In the first clock cycle, the fetch unit fetches an instruction I1 (step F1) and stores it in buffer B1 at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I2 (step F2).

Meanwhile, the execution unit performs the operation specified by instruction I1, which is available to it in buffer B1 (step E1). By the end of the second clock cycle, the execution of instruction I1 is completed and instruction I2 is available. Instruction I2 is stored in B1, replacing I1, which is no longer needed. Step E2 is performed by the execution unit during the third clock cycle, while instruction I3 is being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time.

The fetch and execute units in Figure b constitute a **two-stage pipeline** in which each stage performs one step in processing an instruction. An inter-stage storage buffer, B1, is needed to hold the information being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle.

4 STAGE PIPIELINING

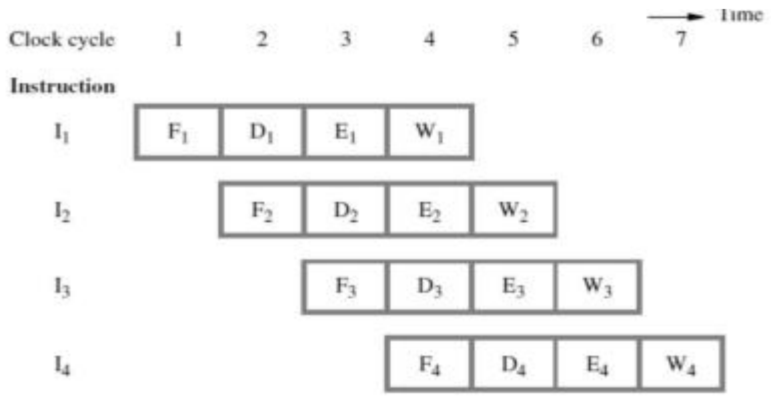
The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

F- Fetch: read the instruction from the memory.

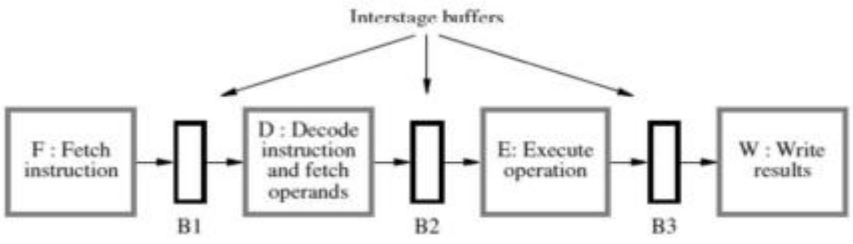
D -Decode: decode the instruction and fetch the source operand(s).

E -Execute: perform the operation specified by the instruction.

W -Write: store the result in the destination location.



(a) Instruction execution divided into four steps



(b) Hardware organization

The sequence of events for this case is shown in Figure a. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure b. These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

• Buffer B1 holds instruction I3, which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.

• Buffer B2 holds both the source operands for instruction I2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I2 (stepW2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.

• Buffer B3 holds the results produced by the execution unit and the destination information for instruction I1.

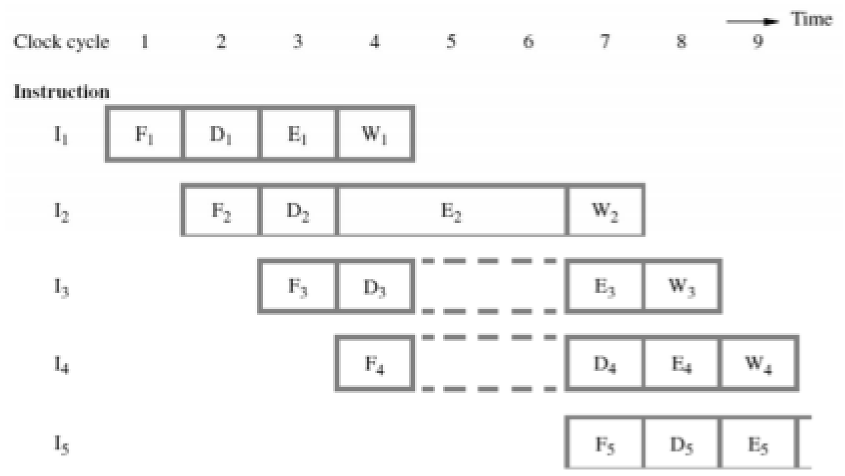
Role of Cache Memory:

Each stage in a pipeline is expected to complete its operation in one clock cycle. Hence, the clock period should be sufficiently long to complete the task being performed in any stage. If different units require different amounts of time, the clock period must allow the longest task to be completed. A unit that completes its task early is idle for the remainder of the clock period. Hence, pipelining is most effective in improving performance if the tasks being performed in different stages require about the same amount of time. This consideration is particularly important for the instruction fetch step, which is assigned one clock period. The clock cycle has to be equal to or greater than the time needed to complete a fetch operation. However, the access time of the main memory may be as much as ten times greater than the time needed to perform basic pipeline stage operations inside the processor, such as adding two numbers. Thus, if each instruction fetches required access to the main memory, pipelining would be of little value.

The use of cache memories solves the memory access problem. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor. This makes it possible to divide instruction fetching and processing into steps that are more or less equal in duration. Each of these steps is performed by a different pipeline stage, and the clock period is chosen to correspond to the longest one.

Pipeline Performance:

For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted. For example, stage E in the four stage pipeline of Figure b is responsible for arithmetic and logic operations, and one clock cycle is assigned for this task. Although this may be sufficient for most operations, some operations, such as divide, may require more time to complete. Figure shows an example in which the operation specified in instruction I2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage 1 is blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps D4 and F5 must be postponed as shown.

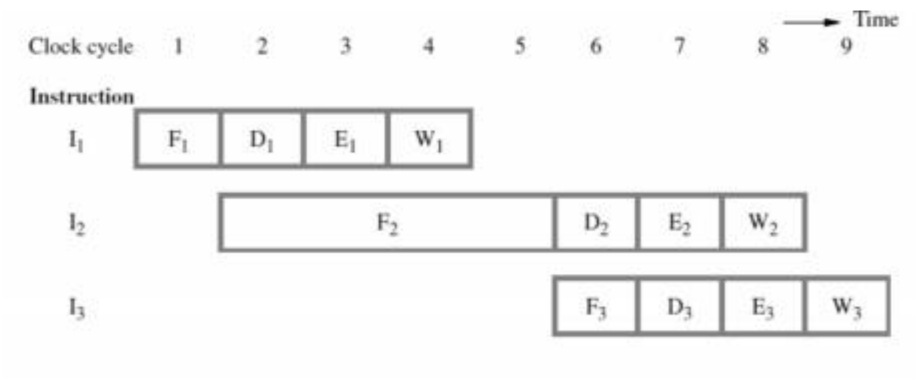


Time
Clock cycle 1 2 3 4 5 6 7 8 9
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 F_2 D_2 E_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4 W_4
I_5 F_5 D_5 E_5

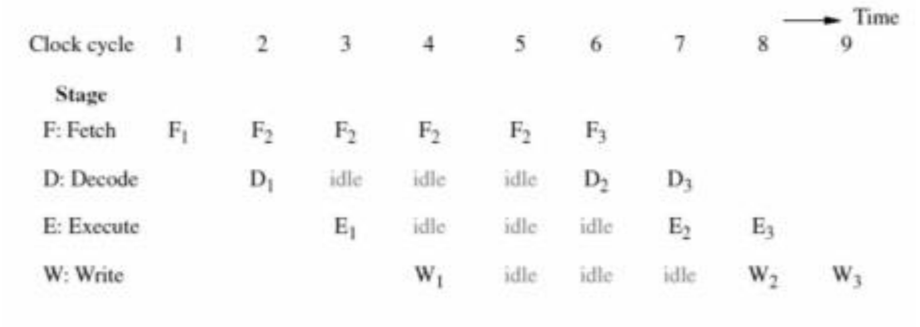
Effect of an execution operation taking more than one clock cycle

Pipelined operation in Figure is said to have been stalled for two clock cycles. Normal pipelined operation resumes in cycle 7. Any condition that causes the pipeline to stall is called a **hazard**. We have just seen an example of a data hazard. "**A data hazard** is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline." As a result some operation has to be delayed, and the pipeline stalls.

"The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called **control hazards or instruction hazards."** The effect of a cache miss on pipelined operation is illustrated in Figure. Instruction I1 is fetched from the cache in cycle 1, and its execution proceeds normally. However, the fetch operation for instruction I2, which is started in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for I2 to arrive. We assume that instruction I2 is received and loaded into buffer B1 at the end of cycle 5. The pipeline resumes its normal operation at that point.


Time
Clock cycle 1 2 3 4 5 6 7 8 9
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 F_2 D_2 E_2 W_2
I_3 F_3 D_3 E_3 W_3

(a) Instruction execution steps in successive clock cycles



Clock cycle	1	2	3	4	5	6	7	8	9
Stage									
F: Fetch	F_1	F_2	F_2	F_2	F_2	F_3			
D: Decode		D_1	idle	idle	idle	D_2	D_3		
E: Execute			E_1	idle	idle	idle	E_2	E_3	
W: Write				W_1	idle	idle	idle	W_2	W_3

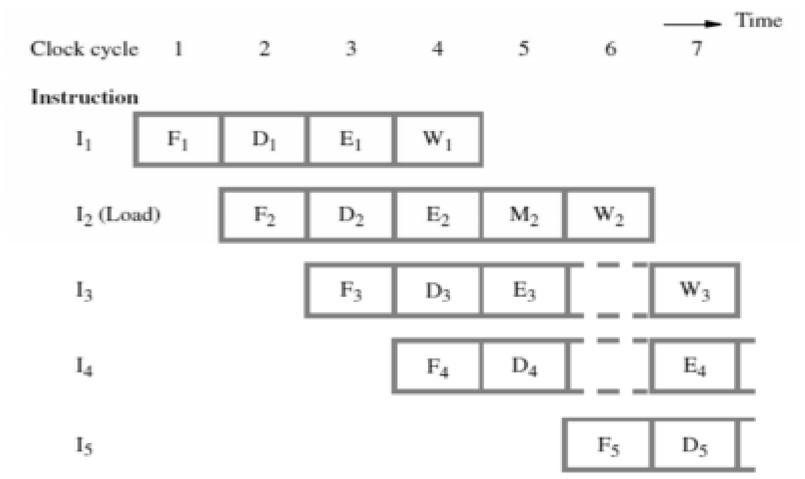
(b) Function performed by each processor stage in successive clock cycles

An alternative representation of the operation of a pipeline in the case of a cache miss is shown in Figure b. This figure gives the function performed by each pipeline stage in each clock cycle. The Decode unit is idle in cycles 3 through 5, the Execute unit is idle in cycles 4 through 6, and the Write unit is idle in cycles 5 through 7. Such idle periods are called stalls. They are also often referred to as bubbles in the pipeline.

A third type of hazard that may be encountered in pipelined operation is known as a "**structural hazard**. This is the situation when two instructions require the use of a given hardware resource at the same time. The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory as part of the Execute or Write stage while another instruction is being fetched." If instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. Many processors use separate instruction and data caches to avoid this delay. An example of a structural hazard is shown in Figure. This figure shows how the load instruction

Load X(R1),R2

The memory address, X+[R1], is computed in stepE2 in cycle 4, then memory access takes place in cycle 5. The operand read from memory is written into register R2 in cycle 6. This means that the execution step of this instruction takes two clock cycles (cycles 4 and 5). It causes the pipeline to stall for one cycle, because both instructions I2 and I3 require access to the register file in cycle 6. In general, structural hazards are avoided by providing sufficient hardware resources on the processor chip.



Time
Clock cycle 1 2 3 4 5 6 7
Instruction
I_1 F_1 D_1 E_1 W_1
I_2 (Load) F_2 D_2 E_2 M_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4
I_5 F_5 D_5

Effect of a Load instruction on pipeline timing

DATA HAZARDS:

A data hazard is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason, as illustrated in Figure. Consider a program that contains two instructions, I1 followed by I2. When this program is executed in a pipeline, the execution of I2 can begin before the execution of I1 is completed. This means that the results generated by I1 may not be available for use by I2. We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially. The potential for obtaining incorrect results when operations are performed concurrently can be demonstrated by a simple example. Assume that A=5, and consider the following two operations:

$$A \leftarrow 3 + A$$

$$B \leftarrow 4 \times A$$

When these operations are performed in the order given, the result is B = 32. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result. If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction.

On the other hand, the two operations

$$A \leftarrow 5 \times C$$

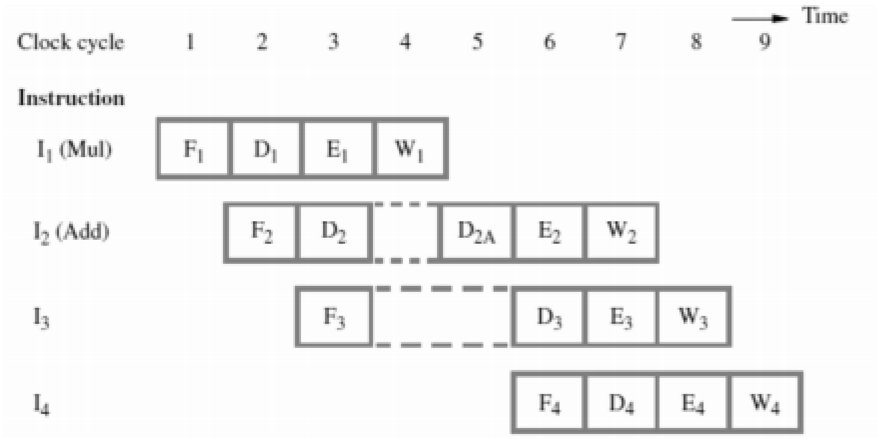
$$B \leftarrow 20 + C$$

Can be performed concurrently, because these operations are independent. When two instructions depend on each other, they must be performed sequentially in the correct order. The data dependency arises when the destination of one instruction is used as a source in the next instruction. For example, the two instructions

Mul R2,R3,R4

Add R5,R4,R6

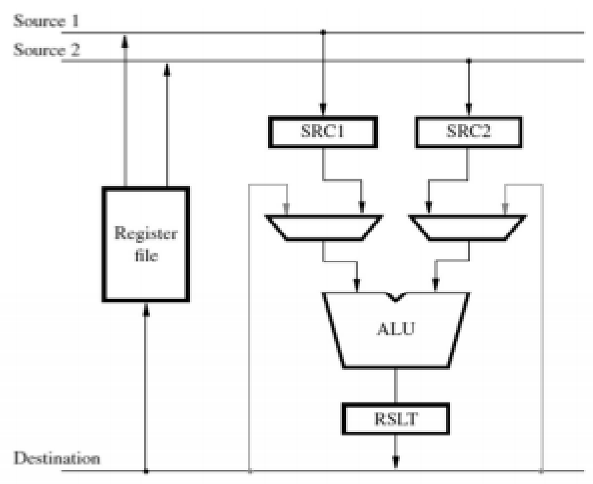
give rise to a data dependency. The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction. Assuming that the multiply operation takes one clock cycle to complete; execution would proceed as shown in Figure. As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand. Hence, the D step of that instruction cannot be completed until the W step of the multiply instruction has been completed. Completion of step D2 must be delayed to clock cycle 5, and is shown as step D2A in the figure. Instruction I3 is fetched in cycle 3, but its decoding must be delayed because step D3 cannot precede D2. Hence, pipelined execution is stalled for two cycles.



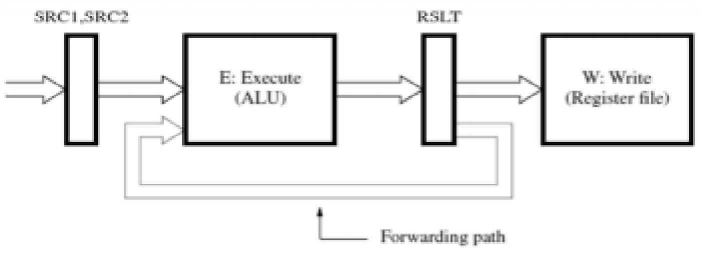
Operand Forwarding:

The data hazard arises because one instruction, instruction I2 in Figure, is waiting for data to be written in the register file. However, these data are available at the output of the ALU once the Execute stage completes step E1. Hence, the delay can be reduced, or possibly eliminated, if we arrange for the result of instruction I1 to be forwarded directly for use in step E2.

This arrangement is similar to the three-bus structure in Figure, except that registers SRC1, SRC2, and RSLT have been added. These registers constitute the inter stage buffers needed for pipelined operation, as illustrated in Figure b. With reference to Figure b, registers SRC1 and SRC2 are part of buffer B2 and RSLT is part of B3. The data forwarding mechanism is provided by the blue connection lines. The two multiplexers connected at the inputs to the ALU allow the data on the destination bus to be selected instead of the contents of either the SRC1 or SRC2 register. When the instructions in Figure are executed in the data path of Figure, the operations performed in each clock cycle are as follows. After decoding instruction I2 and detecting the data dependency, a decision is made to use data forwarding. The operand not involved in the dependency, register R2, is read and loaded in register SRC1 in clock cycle 3. In the next clock cycle, the product produced by instruction I1 is available in register RSLT, and because of the forwarding connection, it can be used in step E2. Hence, execution of I2 proceeds without interruption.



(a) Datapath



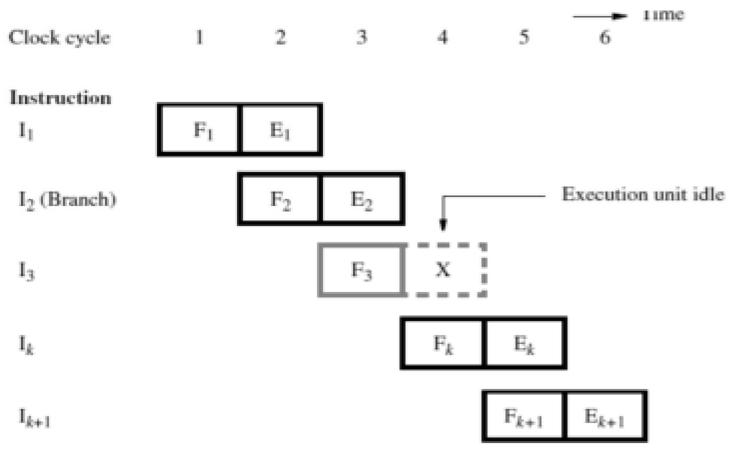
(b) Position of the source and result registers in the processor pipeline

<u>INSTRUCTION HAZARDS:</u>

The purpose of the instruction fetch unit is to supply the execution units with a steady stream of instructions. Whenever this stream is interrupted, the pipeline stalls. A branch instruction may also cause the pipeline to stall.

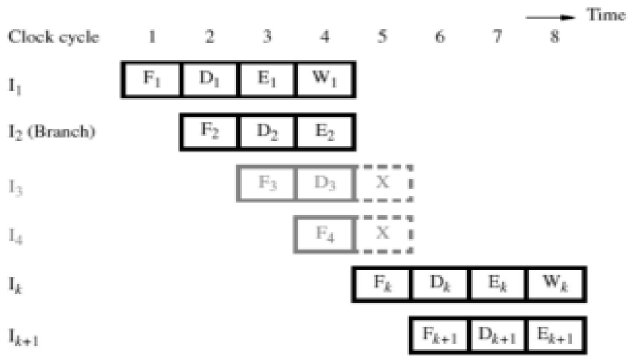
<u>Unconditional Branches</u>

Figure shows a sequence of instructions being executed in a two-stage pipeline. Instructions I1 to I3 are stored at successive memory addresses, and I2 is a branch instruction. Let the branch target be instruction Ik. In clock cycle 3, the fetch operation for instruction I3is in progress at the same time that the branch instruction is being decoded and the target address computed. In clock cycle 4, the processor must discard I3, which has been incorrectly fetched, and fetch instruction Ik. In the meantime, the hardware unit responsible for the Execute (E) step must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.

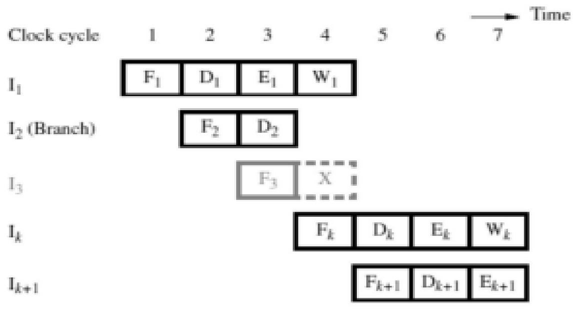


The time lost as a result of a branch instruction is often referred to as the **branch penalty**. In Figure, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher.

For example, Figure a shows the effect of a branch instruction on a four-stage pipeline. We have assumed that the branch address is computed in step E2. Instructions I3 and I4 must be discarded, and the target instruction, Ik , is fetched in clock cycle 5. Thus, the branch penalty is two clock cycles. Reducing the branch penalty requires the branch address to be computed earlier in the pipeline. Typically, the instruction fetch unit has dedicated hardware to identify a branch instruction and compute the branch target address as quickly as possible after an instruction is fetched. With this additional hardware, both of these tasks can be performed in step D2, leading to the sequence of events shown in Figure b. In this case, the branch penalty is only one clock cycle.



(a) Branch address computed in Execute stage



(b) Branch address computed in Decode stage

Instruction Queue and Pre fetching:

Either a cache miss or a branch instruction stalls the pipeline for one or more clock cycles. To reduce the effect of these interruptions, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue. Typically, the instruction queue can store several instructions. A separate unit, which we call the dispatch unit, takes instructions from the front of the queue and sends them to the execution unit. This leads to the organization shown in Figure. The dispatch unit also performs the decoding function.

To be effective, the fetch unit must have sufficient decoding and processing capability to recognize and execute branch instructions. It attempts to keep the instruction queue filled at all times to reduce the impact of occasional delays when fetching instructions. When the pipeline stalls because of a data hazard, for example, the dispatch unit is not able to issue instructions from the instruction queue. However, the fetch unit continues to fetch instructions and add them to the queue. Conversely, if there is a delay in fetching instructions because of a branch or a cache miss, the dispatch unit continues to issue instructions from the instruction queue.

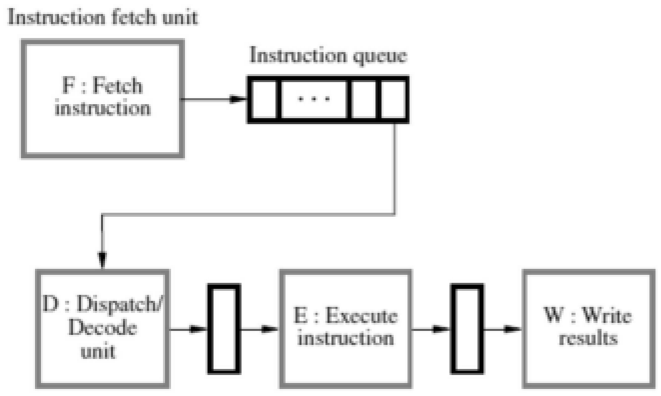
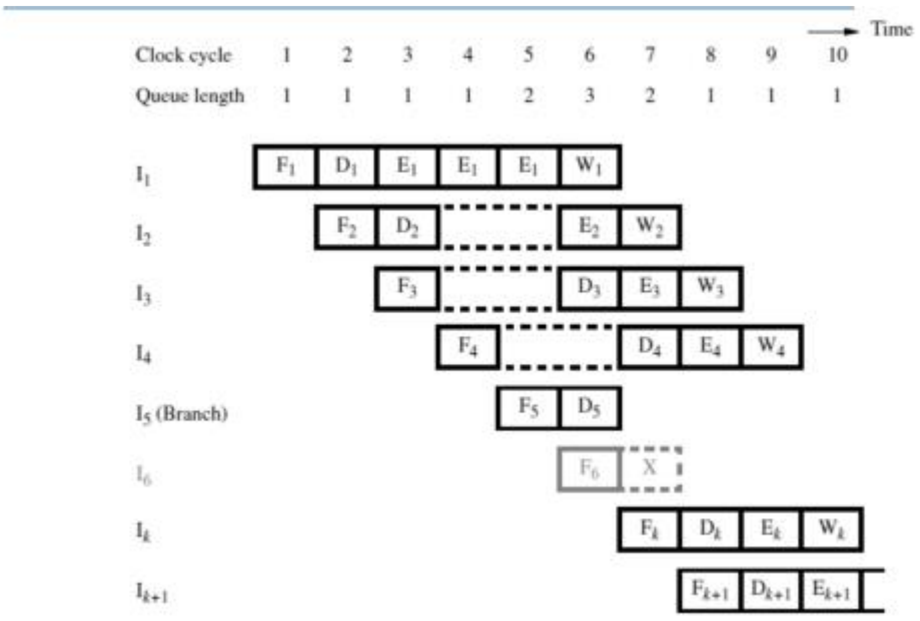


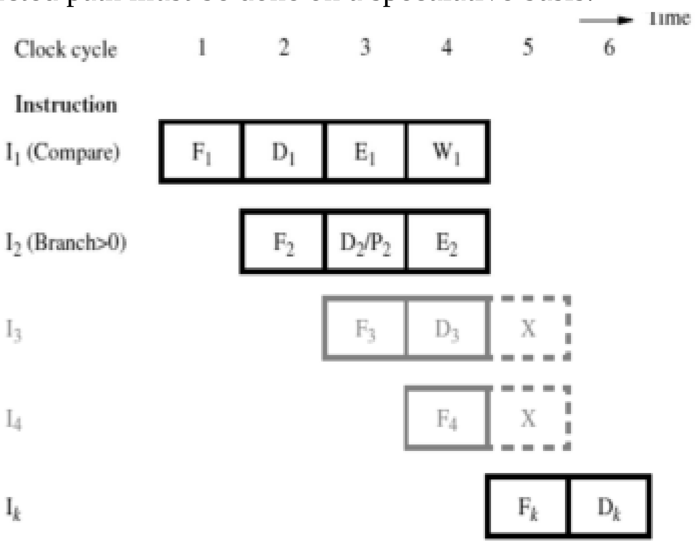
Figure illustrates how the queue length changes and how it affects the relationship between different pipeline stages. We have assumed that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one. Hence, the queue length remains the same for the first four clock cycles. (There is both an F and a D step in each of these cycles.) Suppose that instruction I1 introduces a 2-cycle stall. Since space is available in the queue, the fetch unit continues to fetch instructions and the queue length rises to 3 in clock cycle 6. Instruction I5 is a branch instruction. Its target instruction, Ik , is fetched in cycle 7, and instruction I6 is discarded. The branch instruction would normally cause a stall in cycle 7 as a result of discarding instruction I6. Instead, instruction I4 is dispatched from the queue to the decoding stage. After discarding I6, the queue length drops to 1 in cycle 8. The queue length will be at this value until another stall is encountered.

This is because the instruction fetch unit has executed the branch instruction (by computing the branch address) concurrently with the execution of other instructions. This technique is referred to as branch folding.



Branch Prediction:

Another technique for reducing the branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken. The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order. Until the branch condition is evaluated, instruction execution along the predicted path must be done on a speculative basis.



An incorrectly predicted branch is illustrated in Figure for a four-stage pipeline. The figure shows a Compare instruction followed by a Branch>0 instruction. Branch prediction takes place in cycle 3, while instruction I3 is being fetched. The fetch unit predicts that the branch will not be taken, and it continues to fetch instruction I4 as I3 enters the Decode stage. The results of the compare operation are available at the end of cycle 3. Assuming that they are forwarded immediately to the instruction fetch unit, the branch condition is evaluated in cycle 4. At this point, the instruction fetch unit realizes that the prediction was incorrect, and the two instructions in the execution pipe are purged. A new instruction, Ik , is fetched from the branch target address in clock cycle 5.

If branch outcomes were random, then half the branches would be taken. Then the simple approach of assuming that branches will not be taken would save the time lost to conditional branches 50 percent of the time. However, better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken.

A more flexible approach is to have the compiler decide whether a given branch instruction should be predicted taken or not taken. The branch instructions of some processors, such as SPARC, include a branch prediction bit, which is set to 0 or 1 by the compiler to indicate the desired behaviour. The instruction fetch unit checks this bit to predict whether the branch will be taken or not taken.

With either of these schemes, the branch prediction decision is always the same every time a given instruction is executed. Any approach that has this characteristic is called **static branch prediction.**

Dynamic branch prediction:

Another approach in which the prediction decision may change depending on execution history is called dynamic branch prediction. The objective of branch prediction algorithms is to reduce the probability of making a wrong decision, to avoid fetching instructions that eventually have to be discarded. In dynamic branch prediction schemes, the processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that instruction is executed.

INFLUENCE ON INSTRUCTION SETS:

Instruction side effects can lead to undesirable data dependencies. Two key aspects of machine instructions lead to instruction hazards—addressing modes and condition code flags.

Addressing Modes:

Addressing modes should provide the means for accessing a variety of data structures simply and efficiently. Useful addressing modes include index, indirect, auto increment, and auto decrement. Many processors provide various combinations of these modes to increase the flexibility of their instruction sets. Complex addressing modes, such as those involving double indexing, are often encountered.

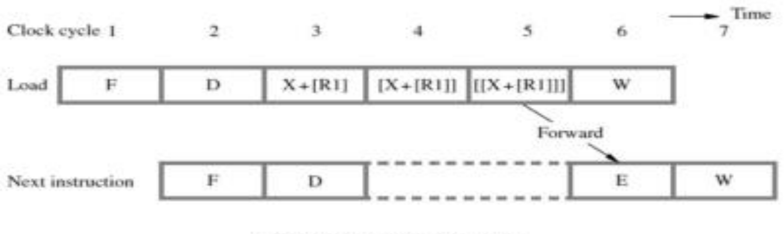
Two important considerations in this regard are the side effects of modes such as auto increment and auto decrement and the extent to which complex addressing modes cause the pipeline to stall. Another important factor is whether a given mode is likely to be used by compilers. To compare various approaches, we assume a simple model for accessing operands in the memory. The load instruction

```
Load X(R1),R2
```

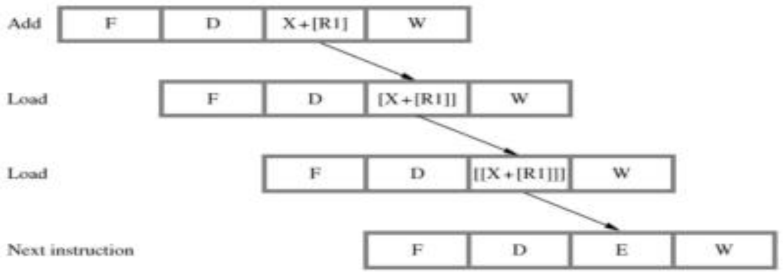
takes five cycles to complete execution, as indicated in Figure. However, the instruction Load (R1),R2 can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E. A more complex addressing mode may require several accesses to the memory to reach the named operand. For example, the instruction

Load (X(R1)),R2

Assuming that the index offset, X, is given in the instruction word. After computing the address in cycle 3, the processor needs to access memory twice — first to read location X+[R1] in clock cycle 4 and then to read location [X+[R1]] in cycle 5. If R2 is a source operand in the next instruction, that instruction would be stalled for three cycles, which can be reduced to two cycles with operand forwarding, as shown.



(a) Complex addressing mode



(b) Simple addressing mode

To implement the same Load operation using only simple addressing modes requires several instructions. For example, on a computer that allows three operand addresses, we can use

Add #X,R1,R2

Load (R2),R2

Load (R2),R2

The Add instruction performs the operation R2←X+[R1]. The two Load instructions fetch the address and then the operand from the memory. This sequence of instructions takes exactly the same number of clock cycles as the original, single Load instruction, as shown in Figure b. This example indicates that, in a pipelined processor, complex addressing modes that involve several accesses to the memory do not necessarily lead to faster execution. The main advantage of such modes is that they reduce the number of instructions needed to perform a given task and thereby reduce the program space needed in the main memory.

The addressing modes used in modern processors often have the following features:

• Access to an operand does not require more than one access to the memory.

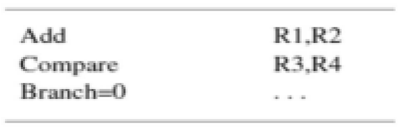
• Only load and store instructions access memory operands.

• The addressing modes used do not have side effects.

Three basic addressing modes that have these features are register, register indirect, and index. The first two require no address computation. In the index mode, the address can be computed in one cycle, whether the index value is given in the instruction or in a register.

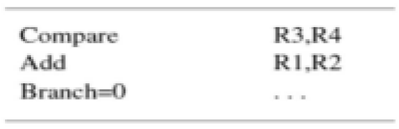
<u>Condition Codes</u>:

The condition code flags either set or cleared by many instructions, so that they can be tested by subsequent conditional branch instructions to change the flow of program execution. An optimizing compiler for a pipelined processor attempts to reorder instructions to avoid stalling the pipeline when branches or data dependencies between successive instructions occur. In doing so, the compiler must ensure that reordering does not cause a change in the outcome of a computation. The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.



```
Add            R1,R2
Compare        R3,R4
Branch=0       . . .
```

(a) A program fragment



```
Compare        R3,R4
Add            R1,R2
Branch=0       . . .
```

(b) Instructions reordered

Consider the sequence of instructions in Figure a, and assume that the execution of the Compare and Branch=0 instructions proceeds as in Figure. The execution time of the Branch instruction can be reduced by interchanging the Add and Compare instructions, as shown in Figure b. This will delay the branch instruction by one cycle relative to the Compare instruction. As a result, at the time the Branch instruction is being decoded the result of the Compare instruction will be available and a correct branch decision will be made. First, to provide flexibility in reordering instructions, the condition-code flags should be affected by as few instructions as possible. Second, the compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not. An instruction set designed with pipelining in mind usually provides the desired flexibility. Figure b shows the instructions reordered assuming that the condition code flags are affected only when this is explicitly stated as part of the instruction OP code.

<u>FORMS OF PARALLEL PROCESSING (FLYNN'S CLASSIFICATION):</u>

Parallel Processing Systems are designed to speed up the execution of programs by dividing the program into multiple fragments and processing these fragments simultaneously. Such systems are multiprocessor systems also known as tightly coupled systems. Parallel systems deal with the simultaneous use of multiple computer resources that can include a single computer with multiple processors, a number of computers connected by a network to form a parallel processing cluster or a combination of both.

Flynn has classified the computer systems based on parallelism in the instructions and in the data streams. These are:

1. Single instruction stream, single data stream (SISD).
2. Single instruction stream, multiple data stream (SIMD).
3. Multiple instruction streams, single data stream (MISD).
4. Multiple instruction stream, multiple data stream (MIMD).

The above classification of parallel computing system is focused in terms of two independent factors: the number of data streams that can be simultaneously processed, and the number of instruction streams that can be simultaneously processed. Here 'instruction stream' we mean an algorithm that instructs the computer what to do whereas 'data stream' (i.e. input to an algorithm) we mean the data that are being operated upon.

Even though Flynn has classified the computer 'systems into four types based on parallelism but only two of them are relevant to parallel computers. These are SIMD and MIMD computers.

SISD Computer Organization

SISD represents a computer organization with a control unit, a processing unit, and a memory unit. SISD is like the serial computer in use. SISD executes instructions sequentially and they may or may not have parallel processing capabilities.

Instructions executed sequentially may get overlapped in their execution stages. A SISD computer can have greater than one functional unit in it. But all the functional units are below the administration of one control unit. Parallel processing in such systems can be attained by pipeline processing or by using multiple functional units.

SIMD Computer Organization

SIMD organization includes multiple processing elements. All these elements are below the administration of a common control unit. All processors get identical instruction from the control unit but work on multiple data items.

The shared subsystem contains multiple modules which help in communicating with all the processors simultaneously. This is further divided into word slice and bit-slice mode organizations.

MISD Computer Organization

MISD organization includes multiple processing units, each receiving separate instructions operating over a similar data flow. The result of one processor becomes the input of the next processor. The introduction of this organization received less attention and was not practically implemented in architecture.

MIMD Computer Organization

A MIMD computer organization contains interactions among the multiprocessors since all memory flows are changed from the common data area transmitted by all processors. If the multi-data streams were derived from different shared memories then it is a multiple SISD operation that is equal to a set of ‘n’ independent SISD systems.