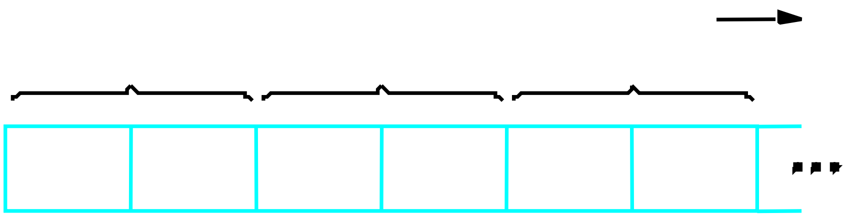
UNIT III PIPELINING

Basic concepts – Data hazards – Instruction hazards – Influence on instruction sets – Data path and control considerations – Performance considerations – Exception handling.

1.Basic Concepts:

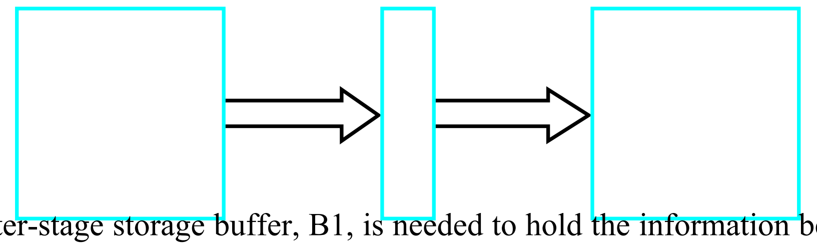
- Pipelining is a particularly effective way of organizing concurrent activity in a computer system.
- In this way number of operation per second is increased.
- It is commonly known as assembly line operation.
- Pipelining used in modern computer to achieve high performance pipelined organization.
- it requires sophisticated compilation techniques and optimizing compiler have been developed.

Sequencial execution



- The processor executes a program by fetching and executing instructions one after the other.
- Let F_i and E_i refer to the fetch and execute steps for instruction I_i.
- Execution of a program consists of a sequence of fetch and executes steps, as shown in fig.

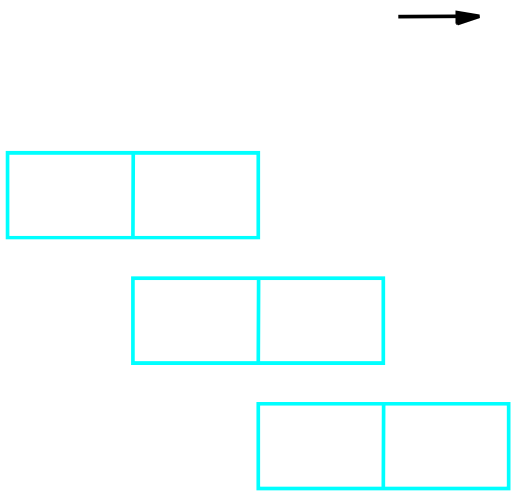
Hardware organization:



- 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.

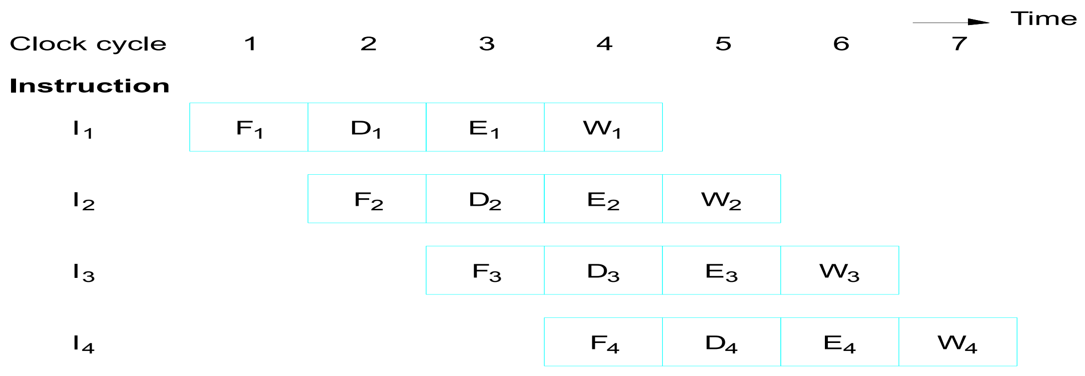
<u>Pipelined execution</u>



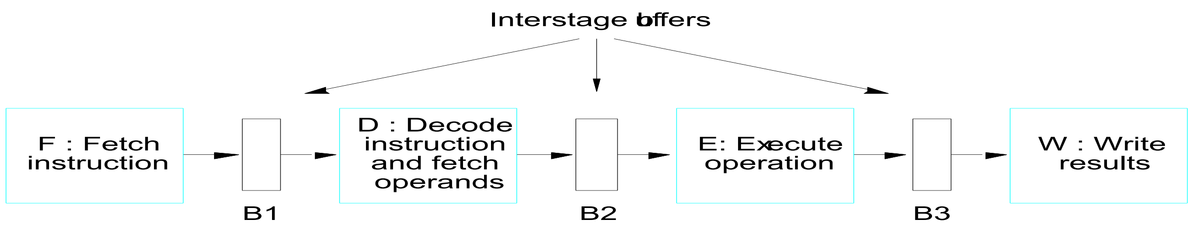
- In the first clock cycle, the fetch unit fetches an instruction I1 (step F1) and stores it in buffer B1 at the end of the clock cycle.
- In the second clock cycle the instruction fetch unit proceeds with the fetch operation for instruction I2 (step F2).
- Meanwhile, the execution unit performs the 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.

<u>4-stage Pipeline</u>

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

Figure 8.2. A 4-stage pipeline.

- Four instructions are in progress at any given time.
- So it needs four distinct hardware units.

➢ 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.

➢ 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 specifications 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(step W2).
- 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.

Pipeline performance:

- Pipelining is proportional to the number of pipeline stages.
- For 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.

Stalls:

- Idle Periods are called stalls. They are also often referred to as bubbles in the pipeline.

Hazard:

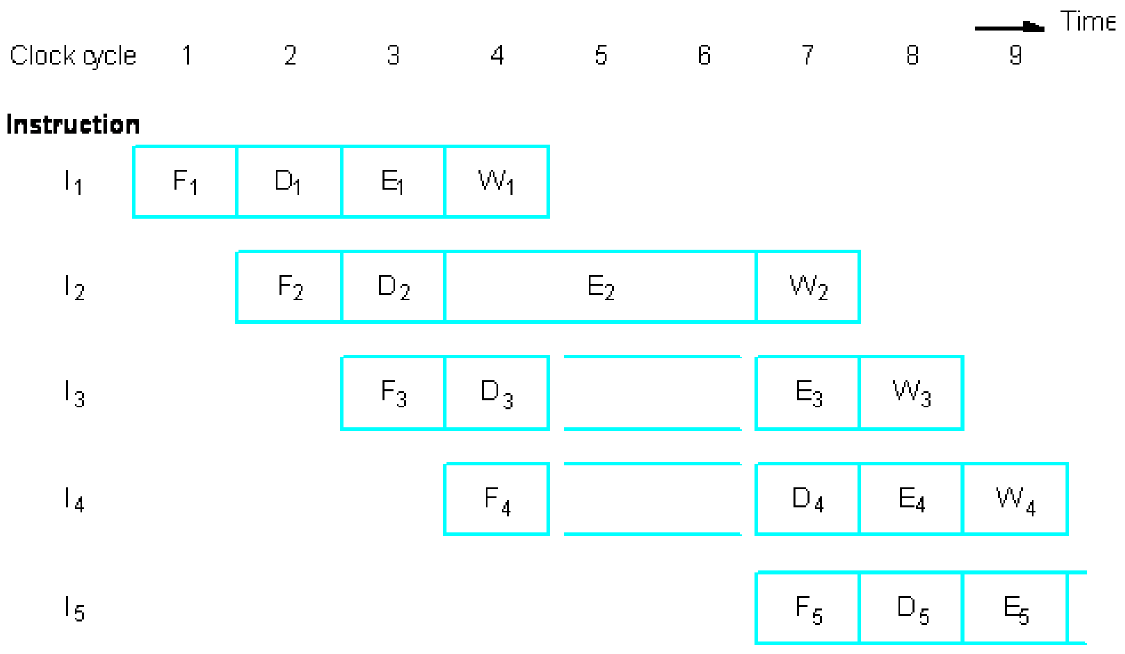
- Any condition that causes the pipeline to stall is called a hazard.

Hazard Types:

- Data hazard
- Control hazard or instruction hazard
- Structural hazard

Data hazard – any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. So some operation has to be delayed, and the pipeline stalls.

Example:

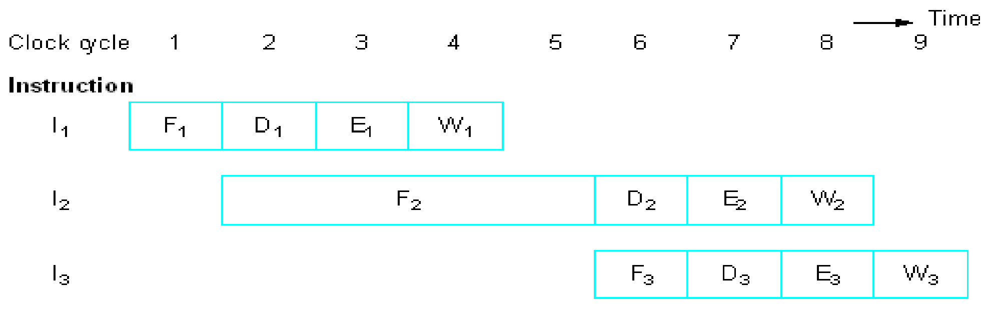


Effect of an execution operation taking more than one clock cycle.

- Stage E in the four-stage pipeline 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.
- Instruction I2 requires 3 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 stage1 are blocked from accepting new instructions because the information in B1 cannot be overwritten.
- Thus steps D4 and F5 must be postponed.

<u>Instruction (control) hazard</u> – a delay in the availability of an instruction causes the pipeline to stall.



(a) Instruction execution steps in successive clock cycles

Clock cycle	1	2	3	4	5	6	7	8	9 → Time
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

Pipeline stall caused by a cache miss in F2.

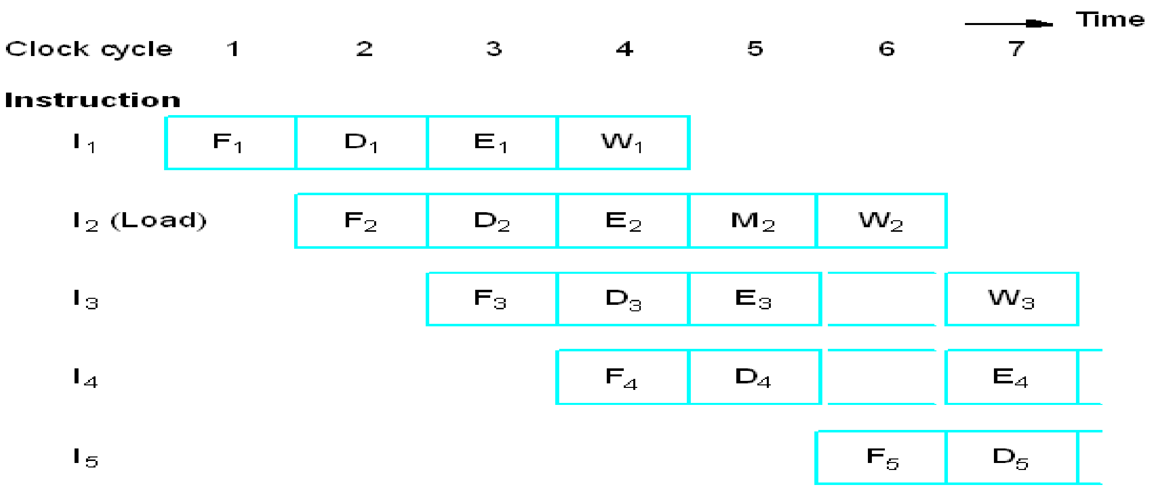
- Instruction I_1 is fetched from the cache in cycle1, and its execution proceeds normally.
- Thus in cycles 5 and 6, the write stage must be told to do nothing, because it has no data to work with.
- Meanwhile, the information in buffer B2 must remain intact until the execute stage has completed its operation.
- This means that stage 2 and in turn stage1 are blocked from accepting now instructions because the information in B1 cannot be overwritten.
- Thus steps D4 and F4 must be postponed.
- Pipelined operation is said to have been stacked for two clock cycles.
- Normal pipelined operation resumes in cycle 7.

Structural hazard:

- This is the situation when two instructions require the use of a given hardware resource it the same time. This hazard arises in case of accessing memory.
- Memory is required by different stages simultaneously.
- One instruction may need to access memory as part of the execute/write stage while another is being fetched. If instruction and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. many processor use separate instruction and data cache to avoid this delay.

Example:

Load X (R_1), R2



Effect of a Load instruction on pipeline timing.

Example:

- The memory address **X+ [R_1],** is computed in step E_2 in cycle 4, then memory access takes place in cycle 5.
- The operand read from memory is written into register R_2 in cycle 6.
- This means that the execution step of this instruction takes two clock cycles (cycles 4&5)
- It causes the pipeline to stall for one cycle, because both instructions I_2 and I_3 require access to the register file in cycle 6.
- Even though the instructions and their data are all available, the pipeline is stalled because one hardware resource, the register file, cannot handle two operations at once.
- If the register file had two input ports, that is, if it allowed two simultaneous write operations, the pipeline would not be stalled.

- In general, **structural hazards are avoided by providing sufficient hardware resources on the processor chip.**
- It is important to understand that pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases, where throughput is measured by the rate at which instruction execution is completed.
- Any time one of the stages in the pipeline cannot complete its operation in one clock cycle, the pipeline stalls, and some degradation in performance occurs.
- An important goal in designing processors is to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact.

2.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 8.3.
- Consider a program that contains two instructions I_1 followed by I_2 .
- 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 I_1 may not be available for use by I_2.
- 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,consider the following two operations:**

  A ← 3 + A
  B ← 4 × 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 Can be performed concurrently, because these operations are independent.

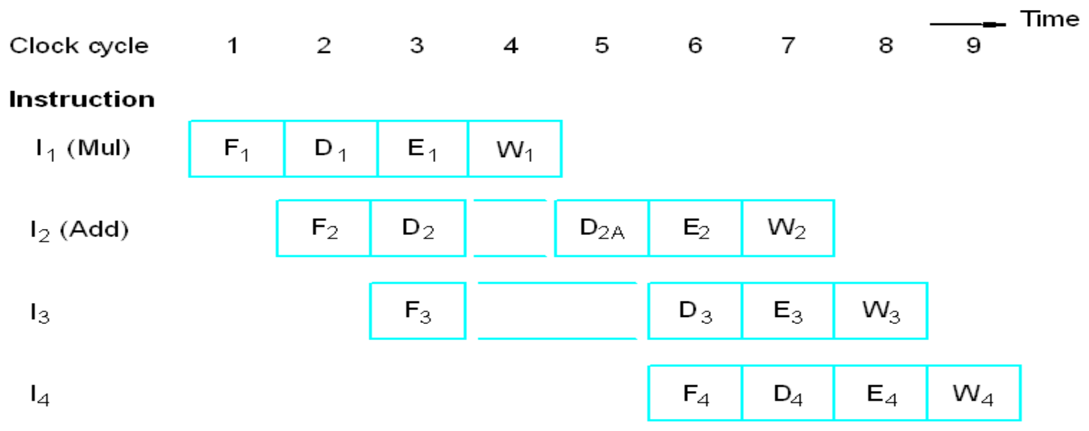
  A ← 5 × C

  B ← 20 + C

Another example which cause pipeline to stall, the two instructions

Mul R2,R3,R4

Add R5,R4,R6

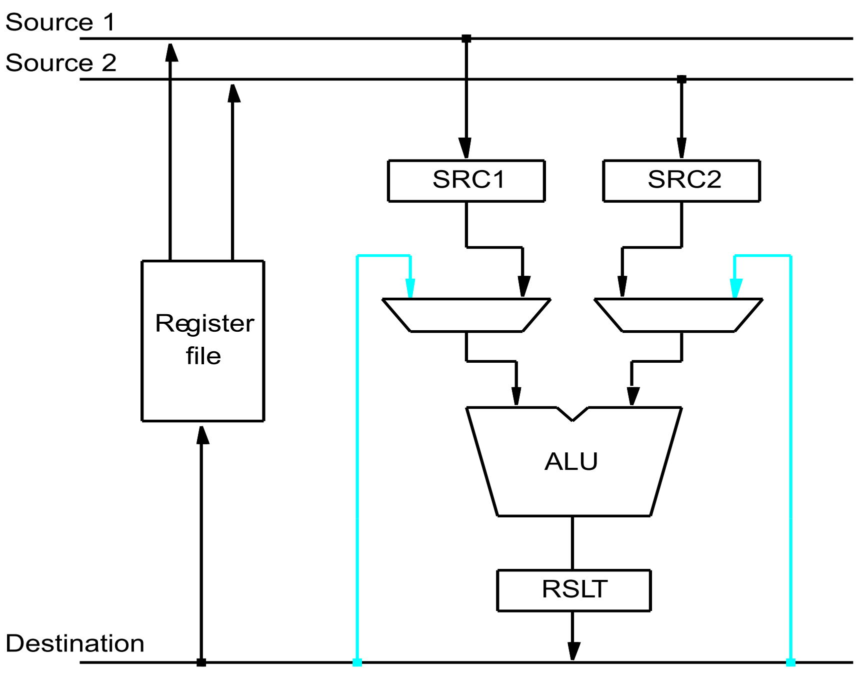


Pipeline stalled by data dependency between D_2 and W_1.

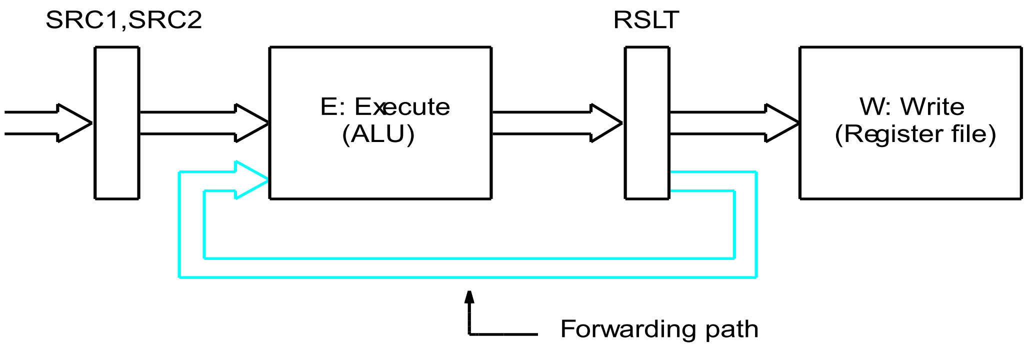
- This example illustrates a basic constraint that must be enforced to guarantee correct results.
- When two operations depend on each other, they must be performed sequentially in the correct order.
- This rather obvious condition has far-reaching con- sequences.
- Understanding its implications is the key to understanding the variety of design alternatives and trade-offs encountered in pipelined computers.
- Consider the pipeline in **Figure (A 4- stage pipeline)** The data dependency just described arises when the destination of one instruction is used as a source in the next instruction.

Operand Forwarding:

- One technique to avoid the data dependency is operand forwarding.
- The data hazard arises because one instruction, instruction I_2 in fig, 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 stepE_1.
- Hence the delay can be reduced, or possibly eliminated, if we arrange for the result of instruction I_1 to be forwarded directly for use in step E_2.



(a) Datapath



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

Figure 8.7. Operand forwarding in a pipelined processor

Figure shows a part of the processor data path involving the ALU and the register file

- The registers constitute the inter stage buffers needed for pipelined operation, as illustrated in Figure 8.7b. With reference to **Figure (4-stage pipeline (hardware organization)),** registers SRC1 and SRC2 are part of buffer B2 and RSLT is part of B3.
- The data forwarding mechanism is provided by forwarding path 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.
- The operations performed in each clock cycle are as follows.

- ✓ After decoding instruction I_2; 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 registerSRC1 in clock cycle 3.
- ✓ In the next clock cycle, the product produced by instruction I1is available in register RSLT, and because of the forwarding connection, it can be used in step E2.
- ✓ Hence, execution of I; proceeds without interruption.

2.2.Handling Data Hazards in Software:

- ➢ In Figure 8.6, we assumed the data dependency is discovered by the hardware while the instruction is being decoded.
- ➢ The control hardware delays reading register R4 until cycle 5, thus introducing a 2-cycle stall unless operand forwarding is used.
- ➢ An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software.
- ➢ In this case, the compiler can introduce the two-cycle delay needed between instructions I1 and I2 by inserting NOP (No-operation) instructions, as follows:

I_1: Mul R2, R3, R4
NOP
NOP
I_2: Add R5, R4, R6

- The compiler must insert the NOP Instruction to obtain a correct result. This possibility illustrate the close link between the compiler and the hardware.
- Leaving task such as inserting NOP instruction to the compiler leads to simpler hardware.
- The compiler can attempt to reorder instruction to perform useful task in the NOP slots and thus achieving better performance. but insertion of NOP instruction leads to larger code size.

3.INSTRUCTION HAZARDS

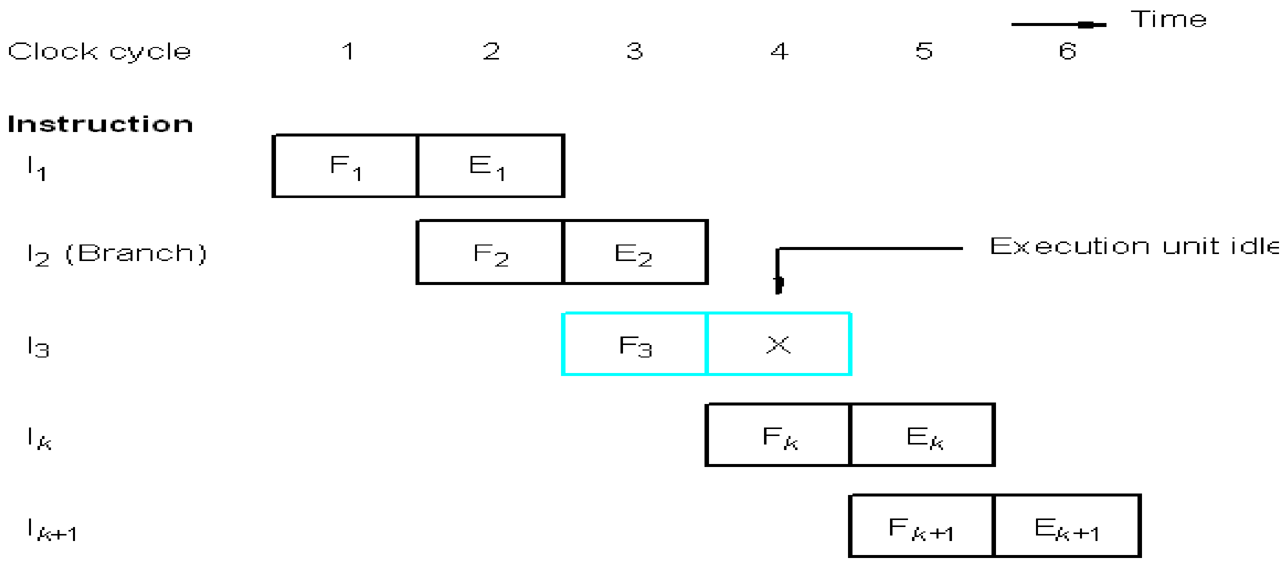
- The pipeline may also be stalled because of a delay in the availability of an instruction.
- This may be result of a miss in the cache, requiring the instruction to be fetched from the main memory, such hazards are often called control/Instruction hazards.

Two Reasons For Instruction Hazard

- Cache Miss
- Branch Instruction
  - ✓ Unconditional branch
  - ✓ conditional branch

- ➢ The purpose of instruction fetch unit is to supply the execution units with a steady stream of instructions.
- ➢ Whenever this stream is interrupted, the pipeline stalls, as **fig (pipeline stall caused by a cache miss in F2)** illustrates for the case of a cache miss.
- ➢ A branch instruction may also cause the pipeline to stall.

3.1Unconditional Branches

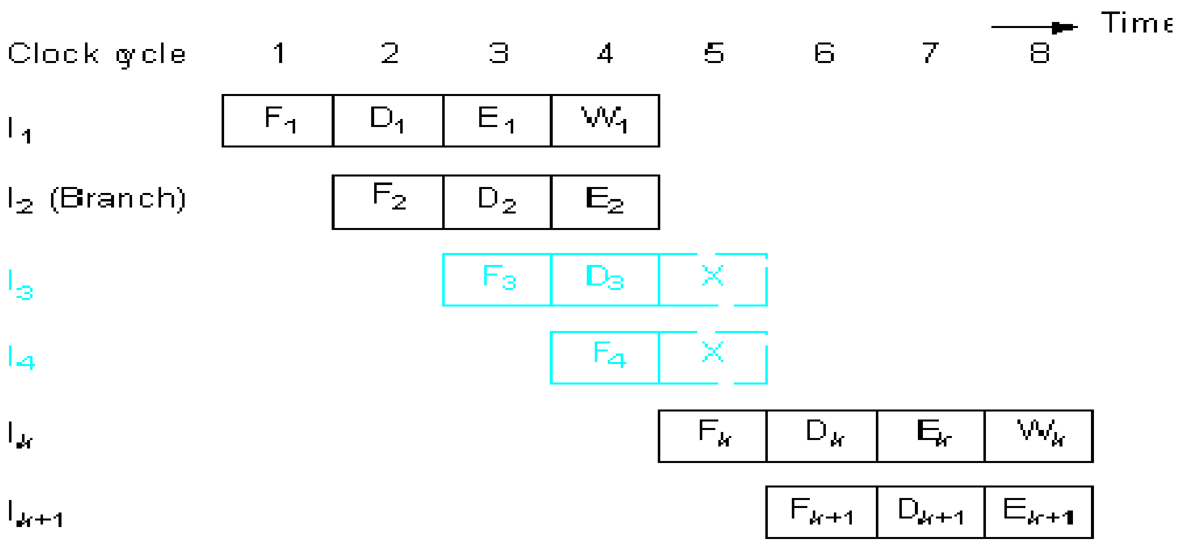


An idle cycle caused by a branch instruction.

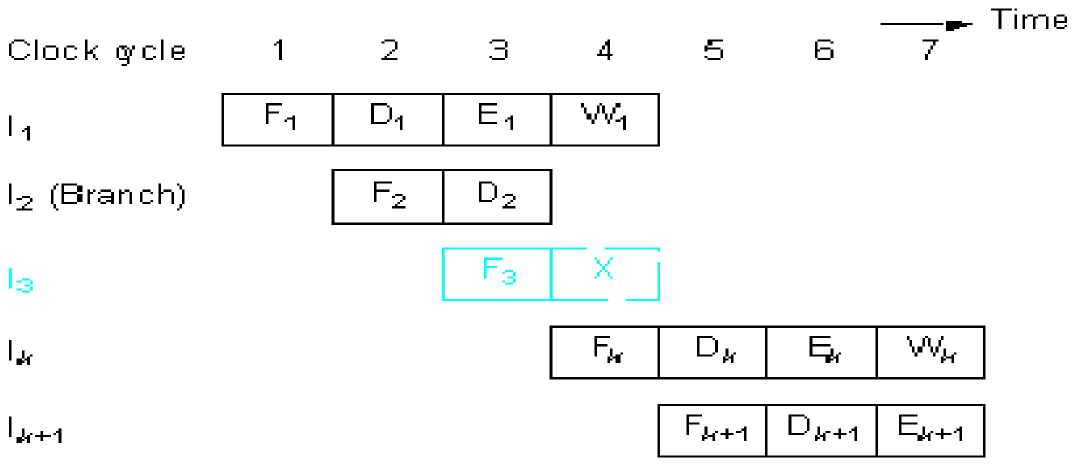
- ➢ Figure shows a sequence of instructions being executed in a two-stage pipeline.
- ➢ Instructions I_1 to I_3 are stored at successive memory addresses, and I_2; is a branch instruction.

- Let the branch target be instruction I_k. In clock cycle 3, the fetch operation for instruction I_3 is 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 discardI3, which has been incorrectly fetched, and fetch instruction I_k .
- 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.

Branch Timing



(a) Branch address computed in Execute stage



(b) Branch address computed in Decode stage

Branch penalty.

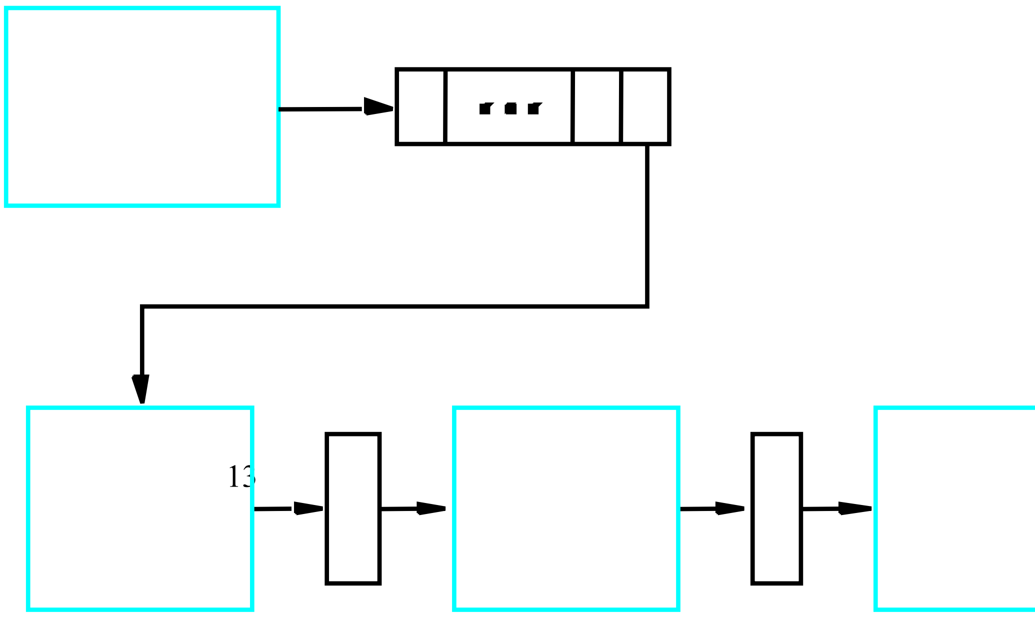
- The time lost as a result of a branch instruction is often referred to as the **branch penalty.**
- **In fig (an idle cycle caused by a branch instruction),** the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher.
- **For example,** Figure a shows the effect of a branch instruction on a four-stage pipeline.
- We have assumed that the branch address is computed in step E2.
- Instructions I_3 and I_4 must be discarded ,and the target instruction,$I_{k,}$ fetched in clock cycle

Instruction Queue and Prefetching(To avoid cache miss)

- The **Fetch unit** may contain instruction queue to store the instruction before they are needed to avoid interruption.
- Another unit called **dispatch unit** takes instruction from the front of the queue and sends them to the execution unit. The dispatch unit alslo performs the decoding function.

The fetch unit must have sufficient decoding and processing capability to recognize and execute branch instruction

- The fetch unit always keeps the instruction queue filled at all times.
- Fetch unit continues to fetch instructions and add them to the queue.
- Similarly if there is a delay in fetching instructions, the dispatch unit continues to issue instruction from the instruction queue.



Branch timing in the presence of an Instruction queue, branch target address is computed in the D stage

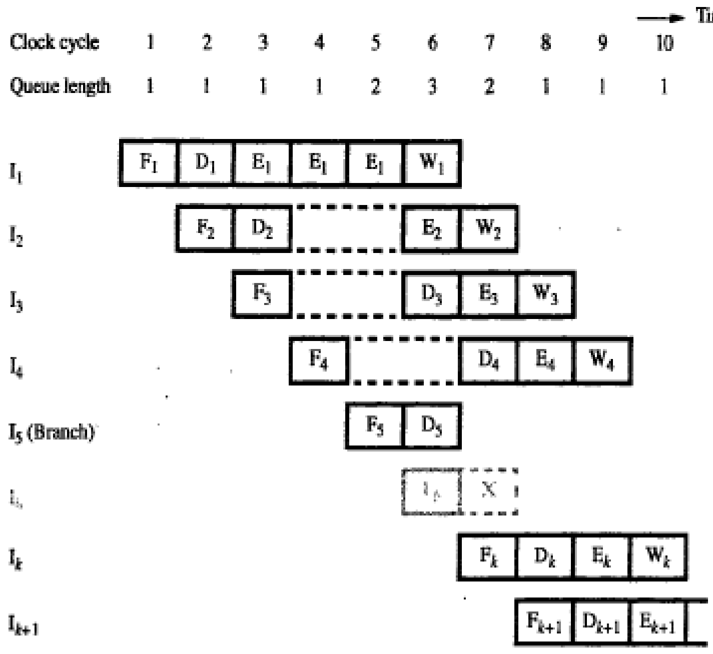
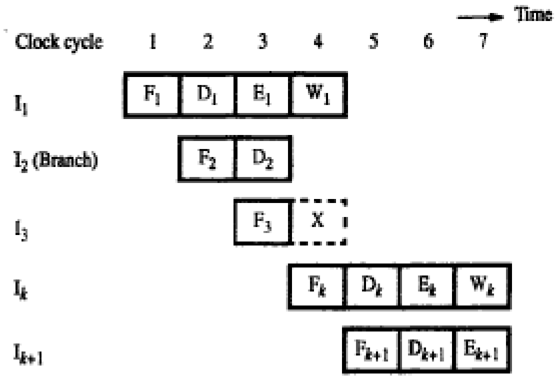

Time
Clock cycle 1 2 3 4 5 6 7 8 9 10
Queue length 1 1 1 1 2 3 2 1 1 1
I_1 F_1 D_1 E_1 E_1 E_1 W_1
I_2 F_2 D_2 E_2 W_2
I_3 F_3 D_3 E_3 W_3
I_4 F_4 D_4 E_4 W_4
I_5 (Branch) F_5 D_5
I_6 F_6 X
I_k F_k D_k E_k W_k
I_{k+1} F_{k+1} D_{k+1} E_{k+1}

Figure 8.11 Branch timing in the presence of an instruction queue. Branch target address is computed in the D stage.

Every fetch operation adds one instruction to the queue and decode operation reduces the queue length by one.

Branch Folding:

In this diagram:
The branch instruction does not increase overall execution time. This is because the instruction fetch unit has executed the branch instruction (by computing the branch target address) currently with the execution of other instruction. This technique is referred to as branch folding.


Time
Clock cycle 1 2 3 4 5 6 7
I_1 F_1 D_1 E_1 W_1
I_2 (Branch) F_2 D_2
I_3 F_3 X
I_k F_k D_k E_k W_k
I_{k+1} F_{k+1} D_{k+1} E_{k+1}

(b) Branch address computed in Decode stage

Figure 8.9 Branch timing.

Branch delay slot:

- The location following a branch instruction is called branch delay slot. There may be more than one branch delay slo**t.**
- There may be more than one branch delay slot, depending on the time it takes to execute a branch Instruction
- A technique called delayed branching can minimize the penalty incurred as a result of conditional Branch Instruction.

3. Conditional Branches and Branch Prediction

The condition branching is a major factor that affects the performance of instruction pipelining. When a conditional branch is executed if may or may not change the PC. If a branch changes the PC to its target address, it is a taken branch, if it falls through, it is not taken. The decision to branch cannot be taken until the execution of that instruction has been completed.

Delayed Branch

- The location following the branch instruction is called branch delay slot. There may be more than one branch delay slot depending on the time it takes to execute the branch instruction.
- The instructions in the delay slot are always fetched at least partially executed before the branch decision is made and the branch target address is completed.
- A technique called **delayed branching** can minimize the penalty caused by conditional branch instruction.
- The instructions in the delay slot are always fetched. Therefore, arrange the instructions which are fully executed, whether or not the branch is taken.
- Place the useful instructions in the delay slot.
- If no useful instructions available, fill the slot with NOP instructions.

- Register R2 is used as counter to determine the number of times contents of R1 are shifted left. For a processor with one delay slot, the instructions can be recorded a above. For a processor with one delay slot, the instructions can be recorded a above. For a processor with one delay slot, the instructions can be reordered as shown in figure (b).
- The shift instruction is fetched while branch instruction is being executed.
- After evaluating the branch condition, the processor fetches the instruction at LOOP or at NEXT, depending on whether the branch condition is true or false respectively. In either case, it completes the execution of the shift instructions.

The sequence of events during the last two passes in the loop is illustrated in figure.

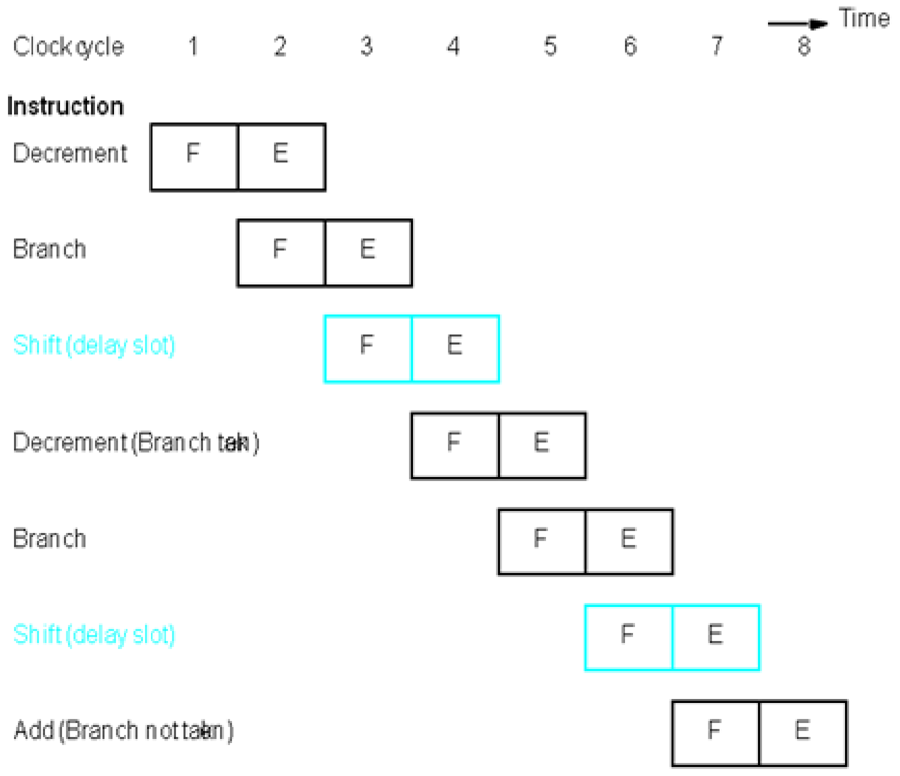


Figure 8.13. Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 8.12.

✓ Pipelined operation is not interrupted at any time, and there are no idle cycles. Branching takes place one instruction later than where branch instruction appears in the sequence, hence named "delayed branch".

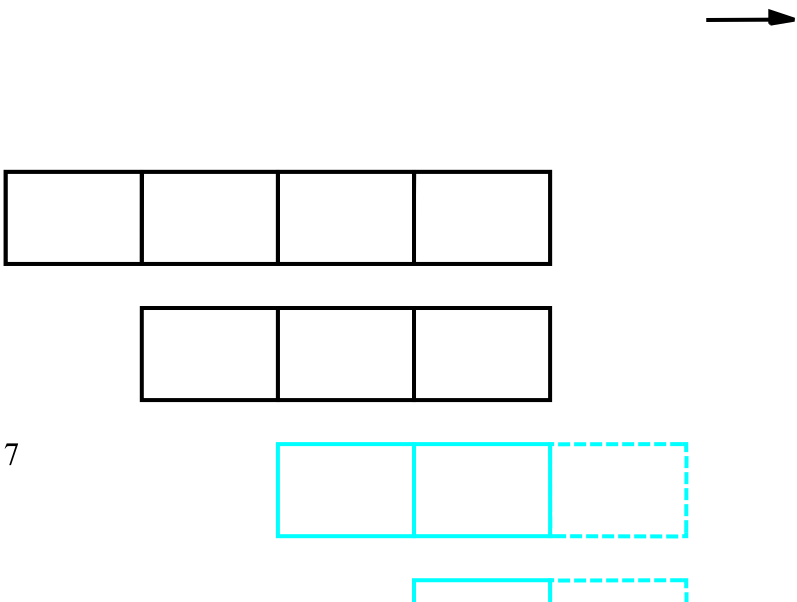
<u>Branch Prediction:</u>

- To predict whether or not a particular branch will be taken.
- Simplest form: assume branch will not take place and continue to fetch instructions in sequential address order.
- Until the branch is evaluated, instruction execution along the predicted path must be done on a speculative basis.
- Speculative execution: instructions are executed before the processor is certain that they are in the correct execution sequence.
- Need to be careful so that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed.

  - Prediction techniques can be used to check whether a branch will be valid or not valid. 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.
  - **Speculative execution** means that instructions are executed before the processor is certain that they are in the correct execution sequence.

➢ Figure 8.14 illustrates the incorrectly predicted branch.

➢ Figure shows a compare instruction followed by a Branch>0 instruction. In cycle 3 the branch prediction takes place; the fetch unit predicts that branch will not be taken and it continues to fetch instruction I4 as I3 enters the Decode Stage. The results of compare operation are available at the ends of cycle 3. 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 the clock cycle 5. We will examine prediction schemes static and dynamic prediction.



Branch prediction:

- Static Branch Prediction
- Dynamic Branch Prediction

Static Prediction

- ➢ The branch prediction is always the same every time a given instruction is executed.
- ➢ Static prediction is usually carried out by the compiler and it is static because the prediction is already known even before the program is executed.

Dynamic Branch Prediction

- ➢ Dynamic prediction in which the prediction decision may change depending on execution history.

Dynamic Prediction Algorithm

- The objective of branch prediction algorithm is to reduce the probability of making the wrong decision.
- It has two different state algorithm

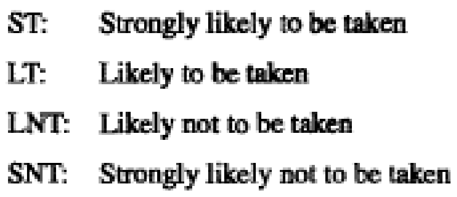
  - ✓ **2 state algorithm-**LT,LNT
  - ✓ **4 state algorithm-**LT,LNT,ST,SNT

- ➢ If the branch taken recently, the next time if the same branch is executed, it is likely that the branch is taken.
- ○ State 1: LT: Branch is likely to be taken
- ○ State 2: LNT: Branch is likely not to be taken

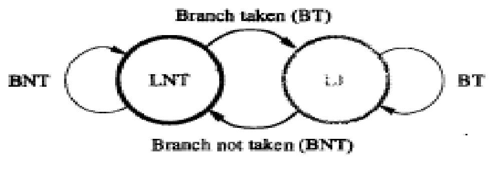
The algorithm is stated in state LNT when the branch is executed.

1. If the branch is taken, the machine moves to LT. Otherwise it remains in state LNT.
2. The branch is predicted as taken if the corresponding state machine is in state LT, otherwise it is predicted as not taken.

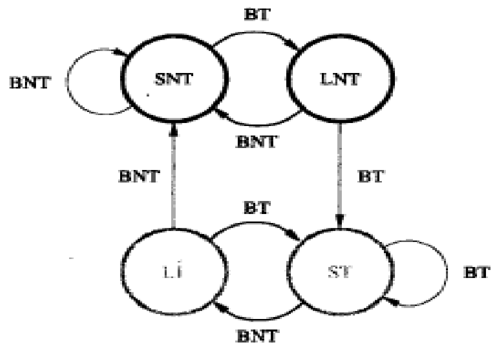
In 4 state algorithm,



ST:	Strongly likely to be taken
LT:	Likely to be taken
LNT:	Likely not to be taken
SNT:	Strongly likely not to be taken



(a) A 2-state algorithm



(b) A 4-state algorithm

Figure 8.15 State-machine representation of branch-prediction algorithms.

4 State Algorithm

- An algorithm that uses 4 states, thus requiring two bits of history information for each branch instruction is shown in figure. The four states are:
  - ✓ ST: Strongly likely to be taken
  - ✓ LT: Likely to be taken
  - ✓ LNT: Likely not to be taken
  - ✓ SNT: Strongly likely not to be taken.

Step 1: Assume that the state of algorithm is initially set to LNT.
Step 2: If the branch is actually taken change to ST, otherwise it is changed to SNT.
Step 3: When a branch instruction is encountered, the branch will be taken if the state is either LT or ST and it begins to fetch instructions at the branch target address. Otherwise, it continues to fetch instruction in sequential address order.

- ✓ When in state SNT, the instruction fetch unit predicts that the branch will not be taken.
- ✓ If the branch is actually taken, that is if the prediction is incorrect, the state changes to LNT.

The state information used in dynamic branch prediction algorithm requires 2 bits for 4 states and may be kept by the processes in a variety of ways:

1. Use look-up table, which is accessed using low-order part of the branch of instruction address.
2. Store as tag bits associated with branch instruction in the instruction cache.