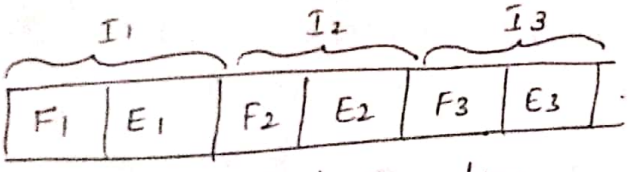
PIPELINING - BASIC CONCEPT:

* The speed of execution of programs is influenced by many factors One way is to arrange the hardware so that more than one Operation can be performed at the same time.

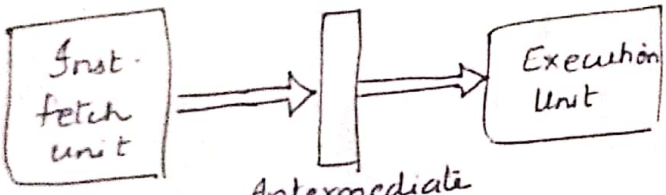
* Pipelining is an effective way of organizing concurrent activity in a computer system

* The processor executes a program by fetching and executing the instructions, one after the other Let E_i and E_i refer to the fetch and execute steps for instruction I_i.

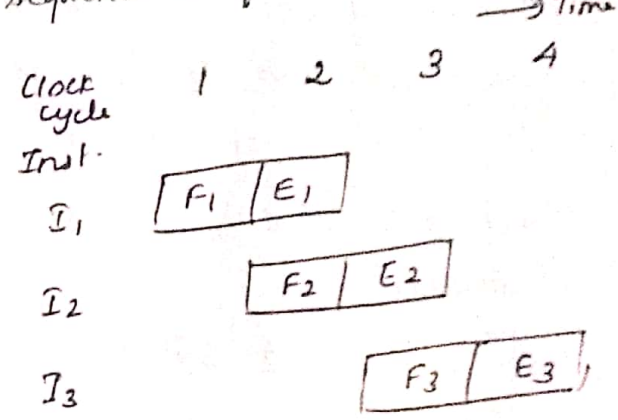
* Execution of program consists of a sequence of fetch & execute steps.



Sequential Execution.



Hardware Organization



Pipelined Execution.

* Assume, computer has 2 separate hardware units, one for fetching instructions and another for executing them.

* The instruction fetched by fetch unit is deposited in an intermediate storage buffer B1. This buffer enables execution unit to execute the instruction while the fetch unit is fetching the next instruction.

* 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

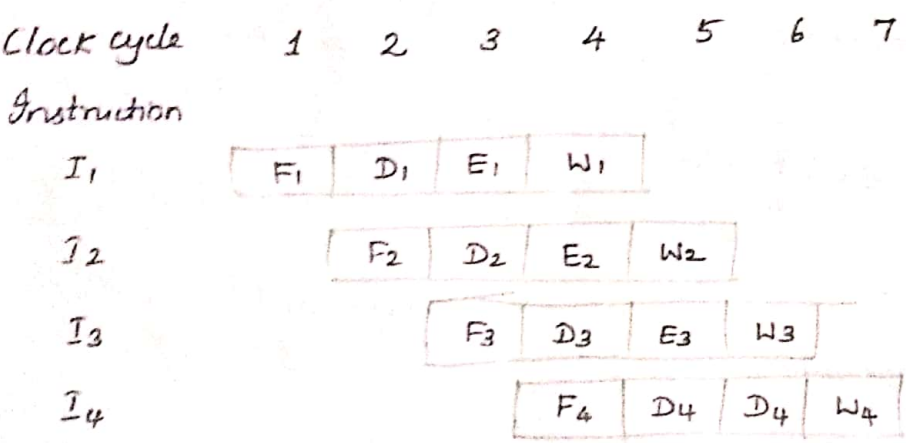
* In 1st clock cycle, fetch unit fetches I_1 and stores it in buffer B1 at the end of the clock cycle.

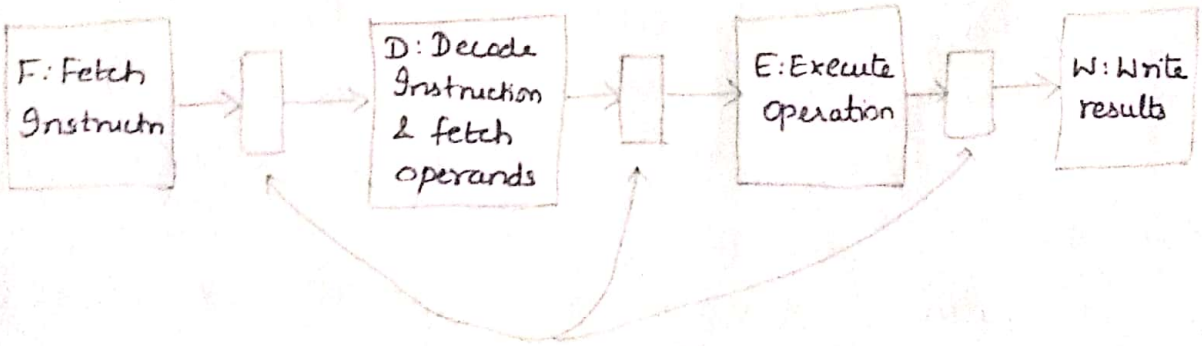
* In 2nd clock cycle, the fetch unit proceeds with the fetch of I_2. Meanwhile, execution unit executes I_1 and is completed I_2 is stored in B_1 replacing I_1.

* The fetch and execute unit constitutes a 2-stage pipeline in which each stage performs one step in processing an instruction.

* The processing of an instruction need not be divided into only 2 steps. For eg. a pipelined processor may process each instruction in 4 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.



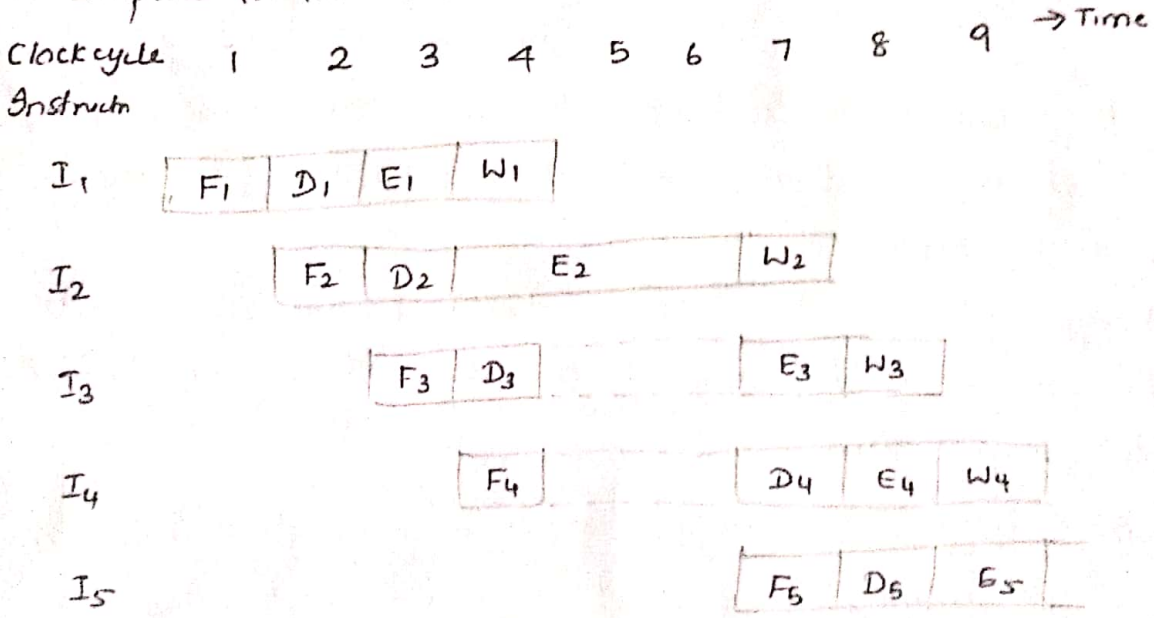


* Four instructions are in progress at any given time. Hence, four distinct hardware units are needed, capable of performing their tasks simultaneously and without interfering with one another.

* Information is passed from one unit to next through a storage buffer.
* 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.
* During a fetch operation, the access time to main memory is greater than the time needed to perform basic pipeline stage operation.
* The use of cache memory solves the memory access problem. When a cache is included on the same chip of processor, access time to cache is less when compared to memory access.

PIPELINE PERFORMANCE:

* The pipelined processor completes the processing of one instruction in each clock cycle, which means the rate of instruction processing is four times that of sequential operation.
* For a variety of reasons, one of the pipeline stages may not be able to complete its task in the time alloted.

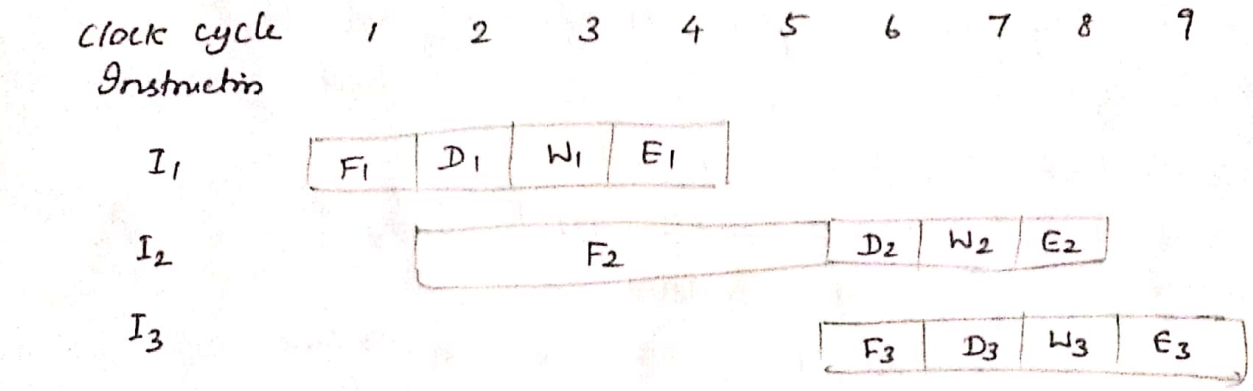


* For eg, the stage E_2 of instruction I_2 requires 3 cycles to complete, from cycle 4 to cycle 6. Thus in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with.
* Meanwhile, the information in buffer B_2 must remain intact until Execute stage has completed its operation.
* Stage 1 and Stage 2 are blocked from accepting new instructions and hence steps D_4 and F_5 are postponed.

* Pipelined operation is said to have been stalled for 2 clock cycles. Normal pipelined operation resumes at cycle 7. Any condition that causes the pipeline to stall is called a hazard.

* The above example is 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.

* The pipeline may also be stalled because of a delay in the availability of an instruction. For eg. this may be a result of miss in the cache, requiring the instruction to be fetched from main memory. Such hazards are called control hazards or instruction hazards.



* The fetch operation for instruction I_2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for I_2 to arrive.

* A third type of hazard is a structural hazard. This is the situation when 2 instructions require the use of a given hardware resource at the same time.

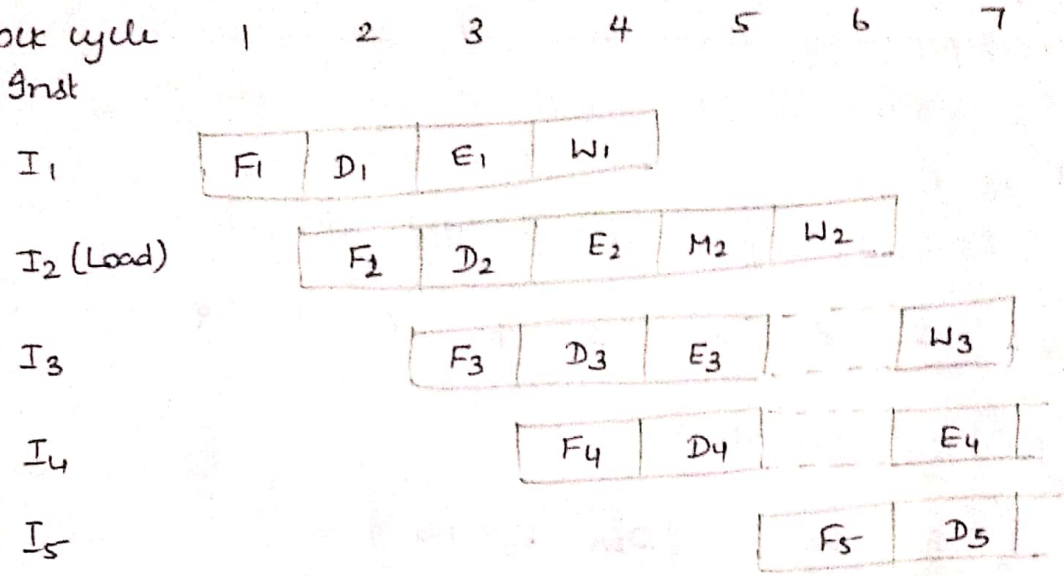
* This hazard may arise in access to memory. One instruction may need to access memory as part of Execute or Write stage while another instruction is being fetched.

* If instruction and data reside in same cache unit, only one instruction can proceed and the other instruction is delayed. Many processors use seperate instruction and data caches to avoid this delay.

* For eg, the load instruction Load X(R_1), R_2. The memory addres

is computed in step E_2 in cycle 4, memory access in cycle 5. The operand read from memory is written into register R_2 in cycle 6.

* It causes the pipeline to stall for one clock cycle, because both instructions I_2 and I_3 require access to register file in cycle 6.



* Even though the instruction and their data are all available, the pipeline is stalled because one hardware resource, the register file, cannot handle 2 operations at once.

DATA HAZARD

* A data hazard is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason.

* When 2 instructions are executed concurrently using pipeline, the result obtained must be same as that of sequential execution.

* The potential for obtaining incorrect results when operations are performed concurrently can be demonstrated with an example.

* Assume $A = 5$, and consider the following 2 operations

$$A \leftarrow 3 + A$$
$$B \leftarrow 4 \times A$$

When these operations are performed in the order, 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.

* On the other hand, the 2 operations

$A \leftarrow 5 \times C$ and $B \leftarrow 20 + C$

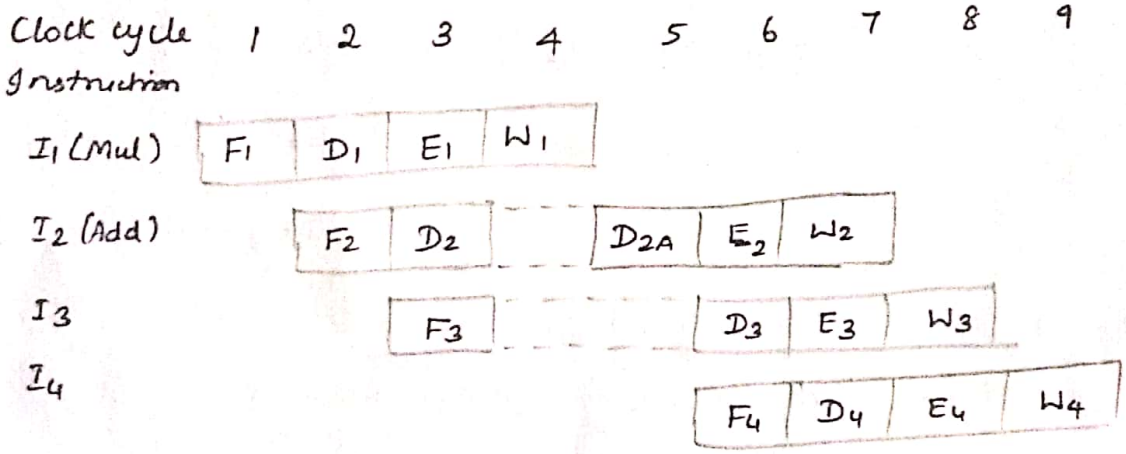
can be performed concurrently, because these operations are independent

* Hence, when 2 operations depend on each other, they must be performed sequentially in correct order to get correct result.

* The data dependency arises when the destination of 1 instruction is used as a source in the next instruction. For eg,

Mul R_2, R_3, R_4

Add R_5, R_4, R_6, gives rise to data dependency.

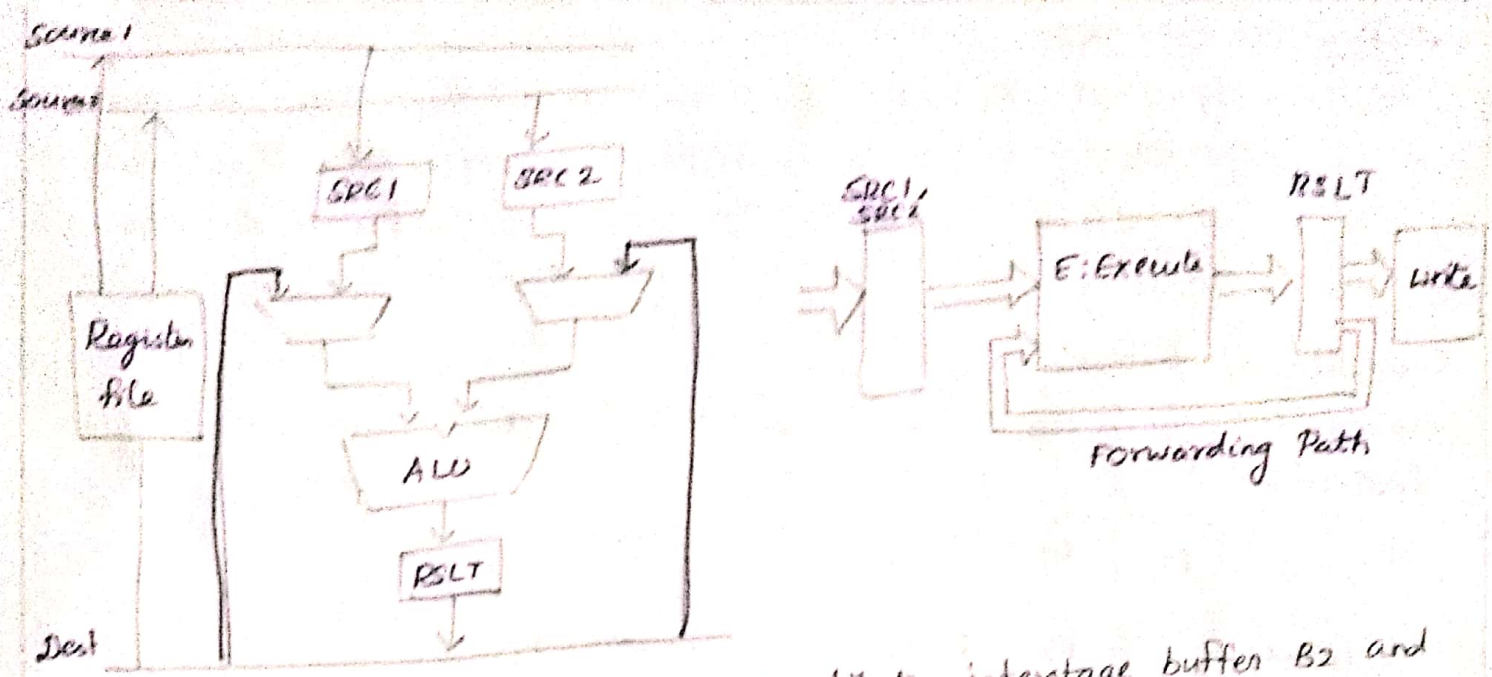


* As the decode unit, decodes the Add instruction in cycle 3, it realizes that R_4 is used as a source operand. Hence, D step of that instruction cannot be completed until the W step of multiply has been completed.

* Completion of step D_2 must be delayed to clock cycle 5. Inst I_3 is fetched in cycle 3, but its decoding must be delayed because step D_3 cannot precede D_2. Hence pipelined execution is stalled for 2 cycles

OPERAND FORWARDING:

* The data hazard arises because one instruction is waiting for data to be written in register file. However, these data are available in the output of ALU once the execute stage completes step E_1. Hence, the delay can be reduced or eliminated, if the result of I_1 can be forwarded directly for use in step E_2



+ The registers SRC1 and SRC2 constitute interstage buffer B2 and RSLT is part of B3. The 2 multiplexers connected at the inputs to ALU allow the data on the destination bus to be selected instead of the contents of either SRC1 or SRC2 register.

HANDLING DATA HAZARDS IN SOFTWARE

+ The data hazard and data dependency is discovered by the hardware while decoding the instruction.

+ An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software

+ In such case, the compiler can introduce delay needed by inserting NOP (No-Operation) instructions.

eg. I_1 : Mul R_2, R_3, R_4
NOP
NOP
I_2 : Add R_4, R_5, R_6

+ Two-cycle delay is needed, hence compiler inserts 2 NOPs.

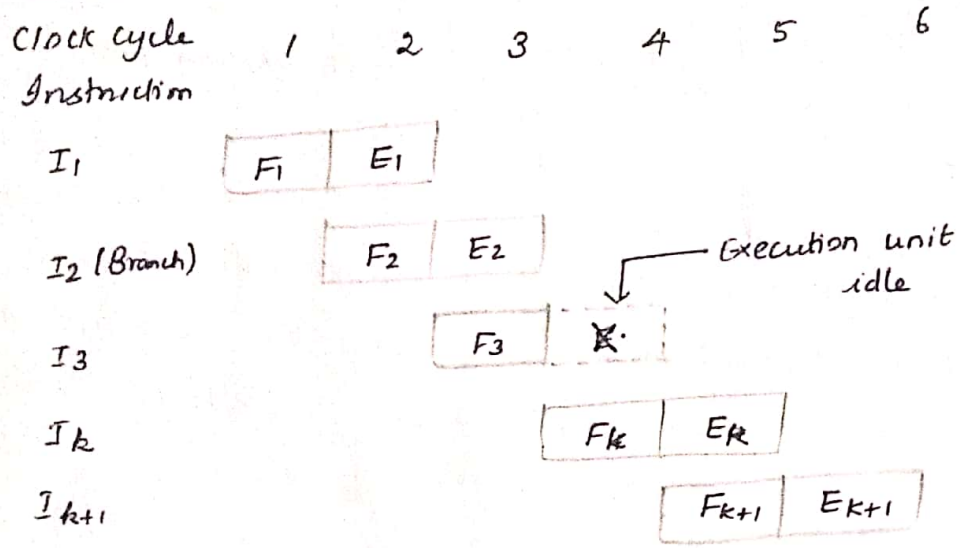
+ Leaving tasks such as inserting NOP instructions to the compiler leads to simpler hardware.

+ Being aware of the need for a delay, the compiler can attempt to reorder instructions to perform useful tasks in the NOP slots and thus achieve better performance.

INSTRUCTION HAZARD:

* 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. A cache miss or a branch instruction may cause stalling

Unconditional Branches:

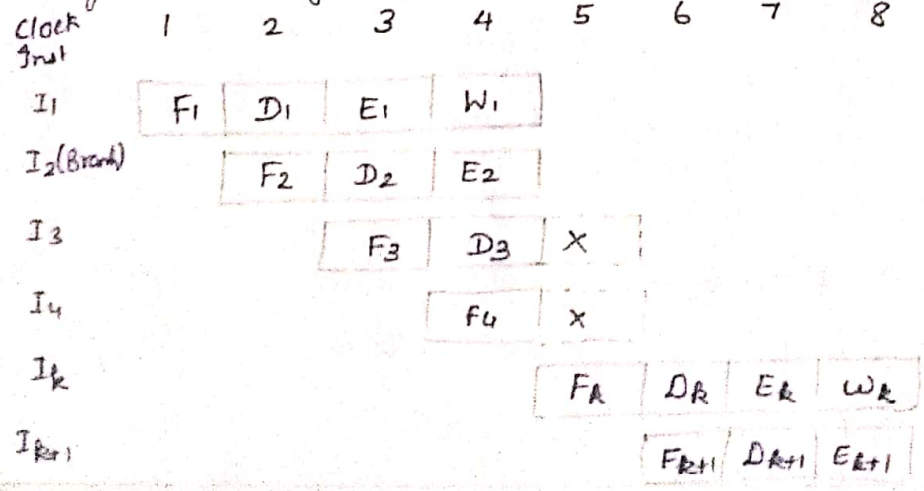


* Instructions I_1 to I_3 are stored in successive memory addresses, and I_2 is a branch instruction. Let the branch target be instruction I_k.

* In clock cycle 3, the fetch operation for I_3 is in progress. At the same time, the branch instruction is being decoded and the target address computed.

* In clock cycle 4, the processor must discard I_3 and fetch I_k. The hardware unit for Execute (E) must be told to do nothing during that clock period. Thus pipeline is stalled for 1 clock cycle.

* The time lost as a result of a branch instruction is referred to as branch penalty. For a longer pipeline, the branch penalty may be higher



* The figure shows the effect of branch instruction on a 4-stage pipeline. In this, the branch address is computed in step E_2. Inst I_3 and I_4 are discarded and the target instruction, I_k, is fetched in clock cycle 5. Thus, the branch penalty is 2 clock cycles.

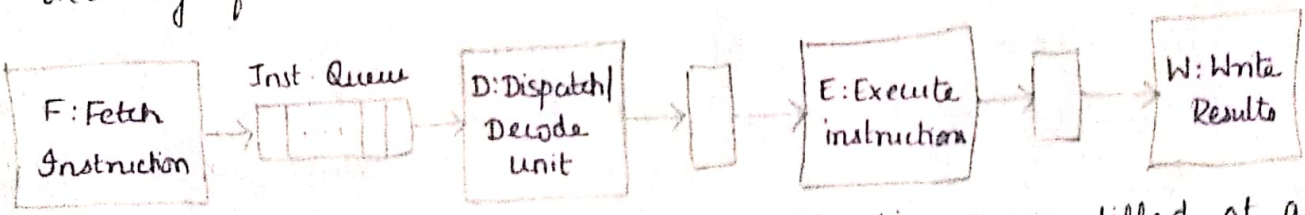
* Reducing the branch penalty, requires the branch address to be computed earlier in pipeline.

* The instruction fetch unit has dedicated hardware to identify a branch instruction and compute branch target address as quickly as possible. With this, branch penalty is reduced to 1 clock cycle.

Instruction Queue and Prefetching:

* To reduce pipeline stalls, fetch units are designed such that they can fetch instructions before they are needed and put them in a queue. The instruction queue can store several instructions.

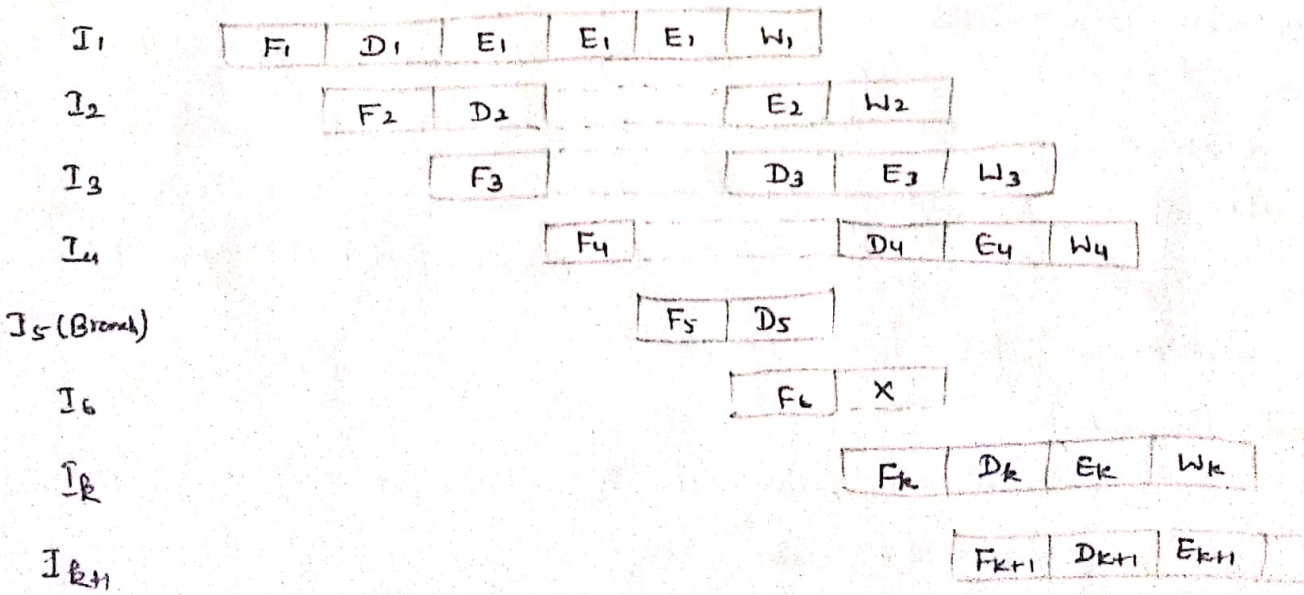
* A seperate unit, called dispatch unit, takes instructions from the front of the queue and sends them to the execution unit. The dispatch unit performs the decoding function.



* The fetch unit attempts to keep the instruction queue filled at all the times to reduce the impact of occasional delays.

* The figure illustrate the effect of queue length.

Clock cycle	1	2	3	4	5	6	7	8	9	10
Queue length	1	1	1	1	2	3	2	1	1	1



* Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one.

* Assume I_1 introduces a 2-cycle stall. Since space is available in queue, the fetch unit continues to fetch instructions and the queue length rises to 3 in clock cycle 6.

* Instruction I_5 is a branch instruction & its target instruction, I_k, is fetched in cycle 7 and instruction I_6 is discarded.

* The branch instruction would normally cause a stall in cycle 7 as a result of discarding I_6. Instead, I_4 is dispatched from the queue to the decoding stage. After discarding I_6, the queue length drops to 1 in cycle 8.

* On observing the sequence of instruction completions, I_1, I_2, I_3, I_4 & I_k complete execution in successive clock cycles. Hence, branch instruction does not increase the overall execution time. This is because the fetch unit has executed branch instruction concurrently with the execution of other instructions. This is called branch folding.

* Branch folding occurs only if at the time a branch instruction is encountered, atleast one instruction is available in the queue other than the branch instruction.

* The instruction queue is also beneficial in dealing with cache misses. When a cache miss occurs, the dispatch unit continues to send instructions for execution as long as queue is not empty.

CONDITIONAL BRANCHES:

* A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction.

* The decision to branch cannot be made until the execution of that instruction has been completed.

Delayed Branch:

* The location following a branch instruction is called branch delay slot. There may be more than 1 delay slot, depending on the execution time of the branch instruction.

* A technique called delayed branching can minimize the penalty incurred as a result of a conditional branch instructions.

* The instruction in the delay slots are always fetched. Hence, arrange for them to be fully executed whether or not the branch is taken.

* The objective is to place useful instructions in these slots. If no useful instructions can be placed in the delay slots, these slots must be filled with NOP instructions.

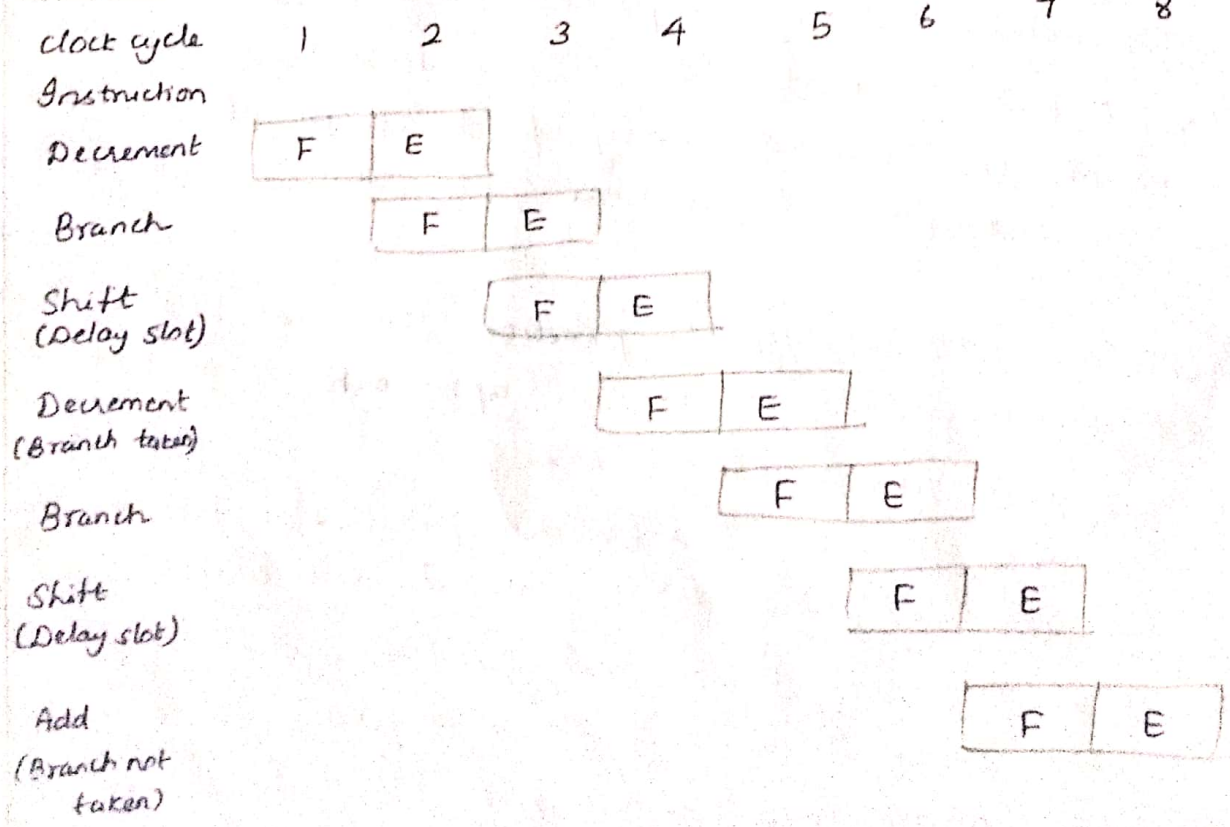
```
LOOP   Shift_left   R1
       Decrement    R2
       Branch=0     LOOP
NEXT   Add          R1, R3
```

a) Orginal Pgm Loop

```
LOOP   Decrement    R2
       Branch=0     LOOP
       Shift_left   R1
NEXT   Add          R1, R3
```

b) Reordered Instructions

* For a processor with one delay slot, the instructions can be reordered as shown in b. The shift instruction is fetched while the branch inst 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 execution of the shift instruction.



* Pipelined operation is not interrupted at any time, and there are no idle cycles.

* Logically, the program is executed as if the branch instruction were placed after the shift instruction.

* Branching takes place one instruction later than where the branch instruction appears in the instruction sequence in memory, hence called "delayed branch".

Branch Prediction:

* Another technique to reduce branch penalty is to predict whether or not a particular branch will be taken. The simplest prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order.

* If the branch decision indicates otherwise, the instructions and all their associated data in the execution units must be purged, and the correct instructions to be fetched and executed.

* If branch outcomes were random, then half the branches would be taken. Hence, the simple approach of assuming that branches will not take place would save 50 percent of time.

* To achieve better performance, branch prediction can be taken depending on the expected program behavior.

* For eg, A branch instruction in a loop cause a branch to the start of the loop for every pass except the last one. Hence it is advantageous to assume that this branch will be taken.

* A decision on which way to predict the result of branch may be made by hardware. A flexible approach is to have the compiler decide about branch prediction.

* Eg. In SPARC processors, a branch prediction bit, is set to 0 or 1 by the compiler to indicate desired behavior.

* With these schemes, the branch prediction decision is always the same every time a given instruction is executed. Such prediction characteristic is called static branch prediction.

Dynamic Branch Prediction

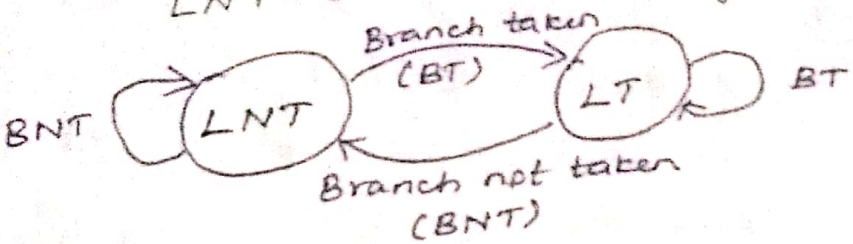
* In dynamic branch prediction, the execution history of the instruction (the most recent execution) is used for branch prediction.

* The processor assumes that the next time instruction is executed, the result is likely to be the same.

* First algorithm, described by 2-state machine. The 2 states are:

LT : Branch is likely to be taken

LNT : Branch is likely not to be taken



* Assume algorithm starts in state LNT. When the branch instruction is executed and if the branch is taken, machine moves to state LT. Otherwise, it remains in state LNT.

* If state is LT, and if branch prediction is taken, then it remains in same state LT, otherwise moves to LNT.

* This simple scheme, which requires one bit of history information for each branch instruction, works well inside program loops.

* Second algorithm, is described by 4-states. The second algorithm yields better performance than first algorithm by maintaining more information about execution history.

* The 4 states are:

ST : Strongly likely to be taken

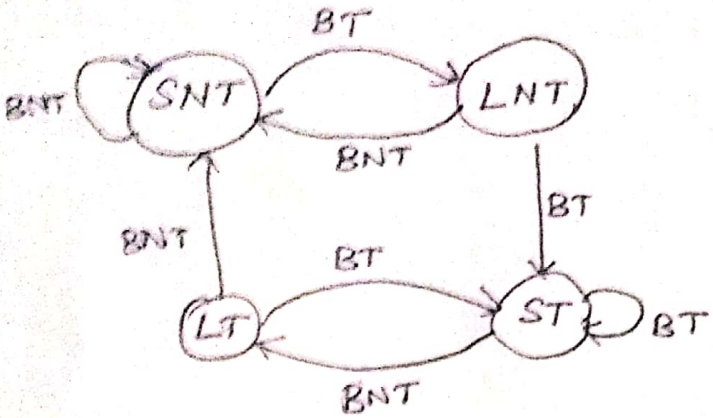
LT : Likely to be taken

LNT : Likely not to be taken

SNT : Strongly likely not to be taken.

* Assume algorithm starts in state LNT. After the branch instruction has been executed, and if the branch is actually

taken, the state is changed to ST; otherwise, it is changed to SNT.



* When a branch instruction is encountered, the instruction fetch unit predicts that 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 instructions in sequential address order.

CONTROL HAZARD:

INFLUENCE OF HAZARDS ON INSTRUCTION SET:

Addressing Modes:

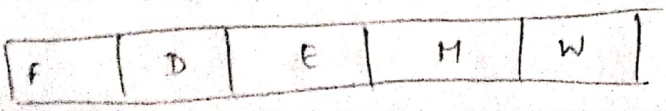
* Addressing modes provides various means for accessing data simply and efficiently. Useful addressing modes include index, indirect, autoincrement and autodecrement. Some processors provide various combination of addressing modes.

* In choosing the addressing modes to be implemented in pipelined processor, the effect of each addressing mode on instruction flow in the pipeline is to be considered.

* TWO considerations:
  - Side effects of modes such as autoincrement / autodecrement.
  - the extend to which complex addressing modes cause the pipeline to stall.

Example:

1) The load instruction Load X(R1), R2 takes 5 clock cycles to complete.

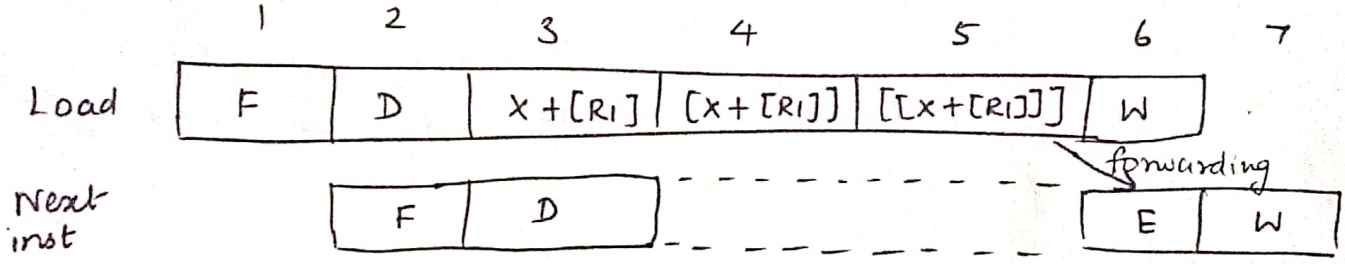


2) The instruction Load (R1), R2 can fit a 4-stage pipeline because no address computation is required.

3) More complex addressing modes may require several access to the memory to reach the named operand.

eg. Load (X(R1)), R2

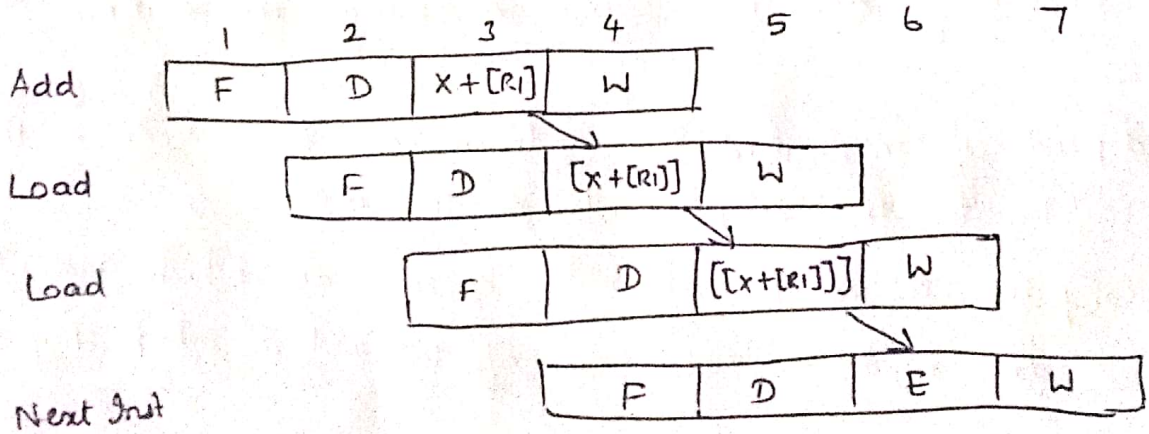
Assuming index offset (X) is given in the instruction word. It computes the address in cycle 3, the processor needs to access the memory twice - first to read location X+[R1] in clock cycle 4 and then to read location [X+[R1]] in cycle 5.



* The next instruction following the load will be stalled for 3 cycles, which can be reduced to 2 cycles with operand forwarding.

* To implement the same Load operation using only simple addressing modes requires several instructions.

eg. Add #X, R1, R2 ⇒ performs R2 ← X + [R1]
Load (R2), R2 ⇒ fetches the address
Load (R2), R2 ⇒ fetches the operand.



* Both takes same number of clock cycles to complete its execution.

* The advantage of complex addressing modes is that it reduce the number of instructions needed to peform the task. The disadvantage is that their longer execution times cause the pipeline to stall, thus reducing its effectiveness.

* The instruction sets of modern processors are designed to take maximum advantage of pipelined hardware. The addressing modes used in modern processors 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, indirect and index.

Condition Codes:

* The condition code flags are stored in processor status register. They are set or cleared by many instructions, so that they can be tested by subsequent conditional branch inst. 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, it must ensure that reordering does not cause a change in outcome of a computation.

* The dependency introduced by condition code flags reduces the flexibility available for the compiler to reorder instructions.

* The compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not.

* To provide flexibility in reordering instructions, the condition code flags should be affected by as few instruction as possible