Data Hazards.

A Data hazard is a situation in which the Pipeline is stalled because the data to be operated on are delayed.

eg 1 A ← 3+A   A=5

B ← 4 × A

Data dependancy → the data used in second instruction depend on the result of first instruction. ie) the destination of one Instruction is used as source in next instruction.

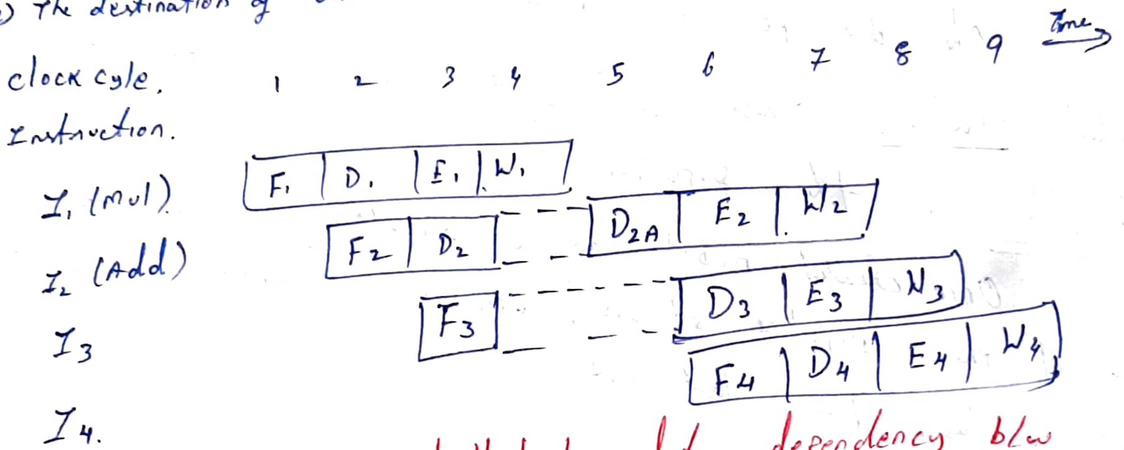


Fig 8.6 Pipeline stalled by data dependency b/w D_2 & W_1

When 2 operations depend on each other, they must be performed sequentially in the correct order.

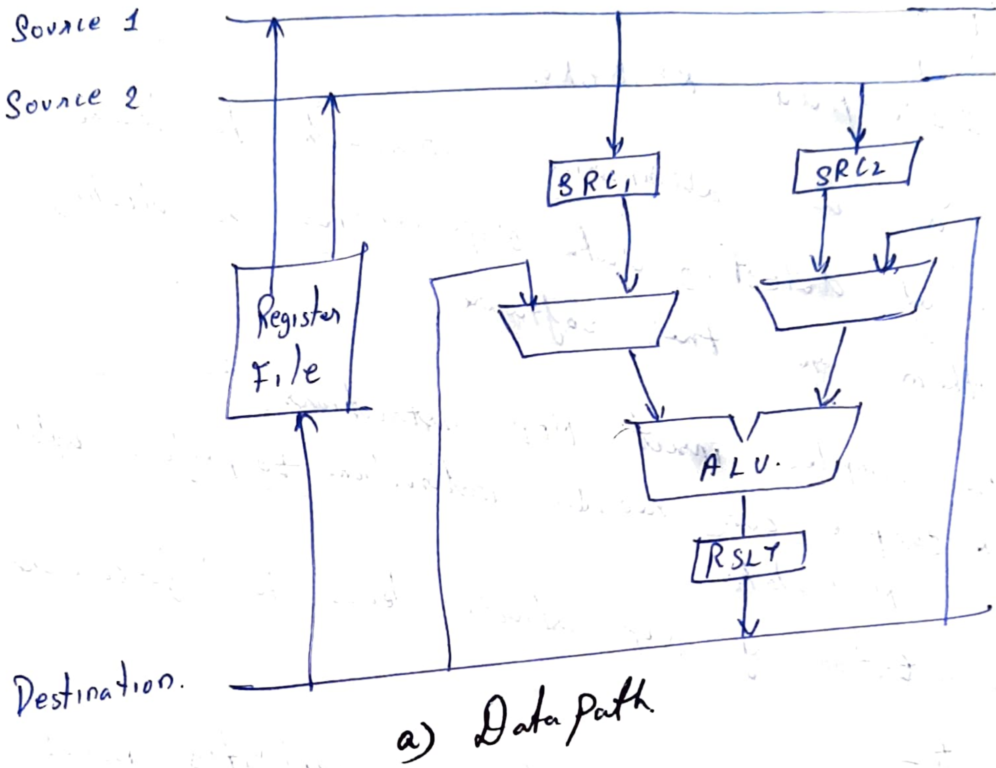
eg 2. Mul R2, R3, R4

Add R5, R4, R6

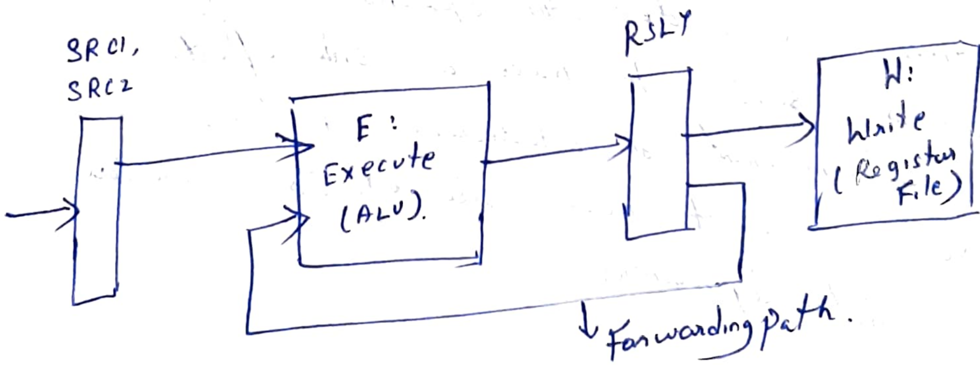
The D step of Instruction 2 cant be compleated until the step of Instruction 1 is completed. Thus D_{2A} delay is shown in above example. Hence pipeline execution is stalled for two cycles.

Operand Forwarding -

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) Data path



b) Position of Source & Result registers in processor pipeline.

Fig 8.7 Operand forwarding in a pipelined Processor.

These registers (RSLT) constitutes of interstage buffers. Thus execution of instructions are proceeded without interruption.

Handling Data hazards in Software -

1) Operand forwarding.
2) Inserting NOP(No-operation) Instruction,

eg I1 : Mul R2, R3, R4
NOP
NOP
I2 : Add R5, R4, R6.

This is an alternative approach to leave the task of detecting data dependencies & dealing with them to the software.

The compiler insert NOP instructions.
The compiler can reorder instructions to perform useful tasks in NOP slots.
The Insertion of NOP instructions leads to larger code size.

Side Effects -

When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a side effect.
eg due to auto increment or autodecrement operation.
Instruction that have side effects give raise to multiple data dependencies.

For better pipelined Organization -

* The give Instruction should affect, only the contents of destination location, either a register or memory location.
* Side effects such as setting condition code flags or address pointers should be kept minimum.

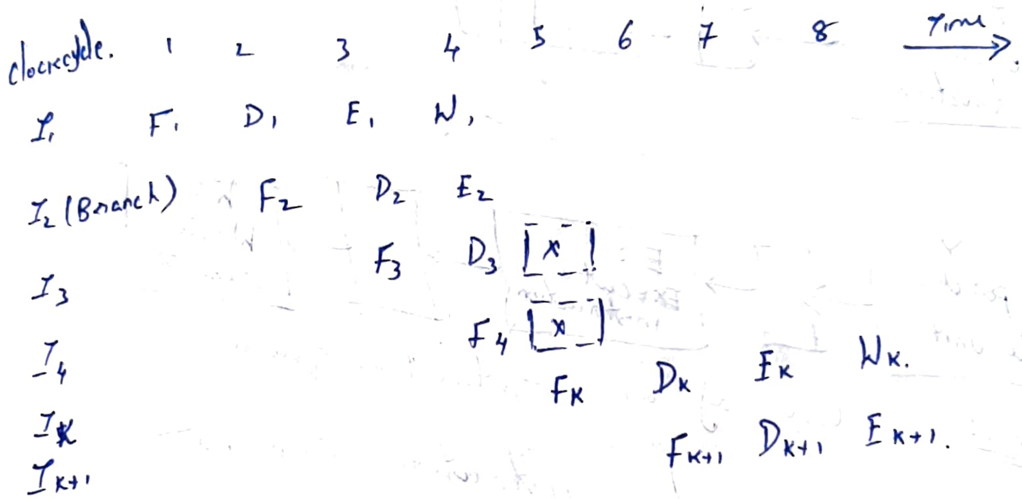
Instruction Hazards -

The instruction stream is interrupted & pipeline stalls due to cache miss.

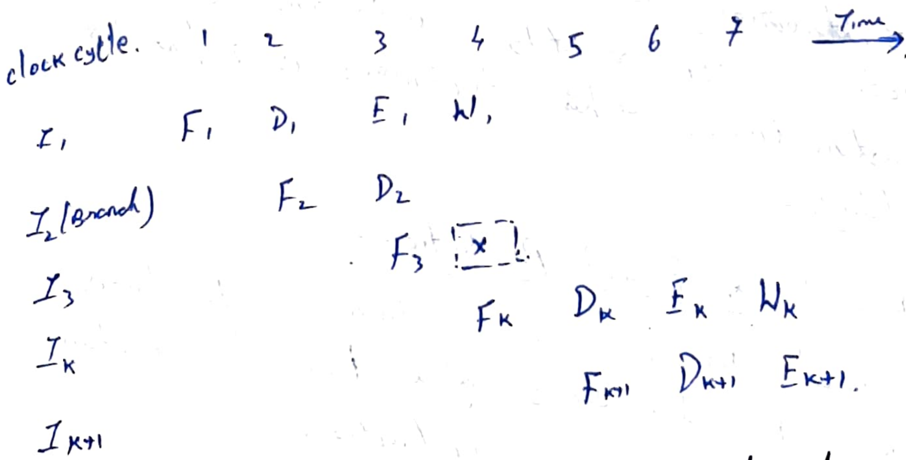
The Branch instruction may also cause the pipeline to stall.

Unconditional Branches -

The time lost as a result of branch instruction is referred to as branch penalty.



a) Branch address at execute stage.



b) Branch address at Decode stage.

Fig 8.9 Branch Timing

Instruction Queue and Prefetching -

Procs employ sophisticated fetch units, that can fetch instructions before they are needed & put them in a queue. They can store several instructions.

The Dispatch unit, takes instructions from the front of the queue and sends to execution unit.

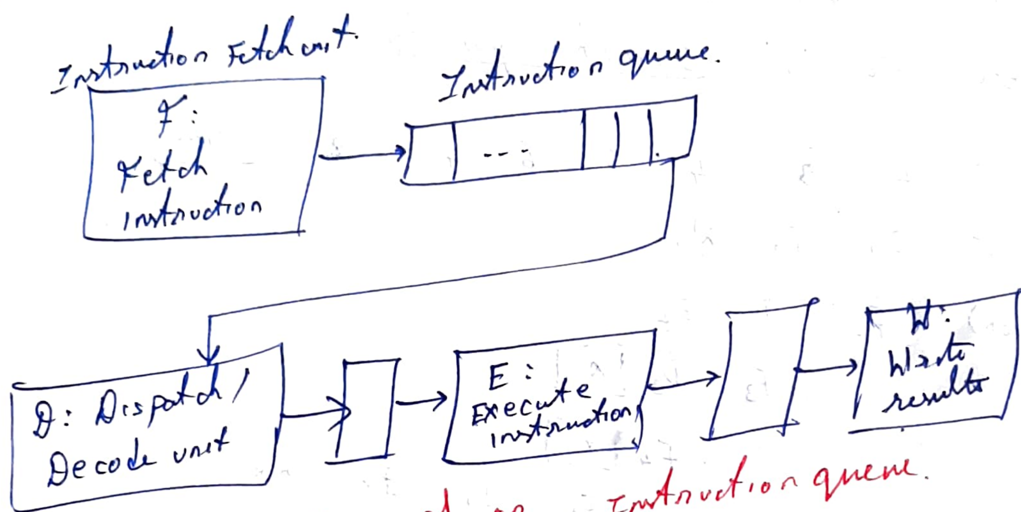


Fig 8.10 Use of an Instruction queue.

Hence, the Branch instruction does not increase the over all execution time.

The Instruction fetch unit has executed the branch instruction concurrently with the execution of other instructions. This technique is referred to as **Branch folding** -

Conditional Branch & Branch Prediction -

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

Delayed Branch-

The location following a branch instruction is called a branch delay slot.

Delayed Branching technique minimize the penalty occured as a result of conditional branch instructions.

In this, the instruction in the delay slots are always fetched. And useful instructions are placed in these slots. If no useful instructions can be placed in delay slots, it will be filled with NOP instructions.

Loop	Shift-left	R_1
	Decrement	R_2
	Branch=0	Loop
Next	Add	R_1, R_3

a) Original program loop.

Loop	Decrement	R_2
	Branch=0	Loop
	Shift-left	R_1
Next	Add	R_1, R_3

b) Reordered instructions.

Fig 8.12 Reordering of instructions for a Delayed Branch.

Branch prediction

It is attempt to predict weather or not a particular branch will be taken.

Simplest form of branch prediction is to assume branch will not take place and to continue to fetch instructions in sequential address order.

↳ Static Branch Prediction -

The Branch prediction decision is always same every time a given instruction is executed.

↳ Dynamic Branch Prediction -

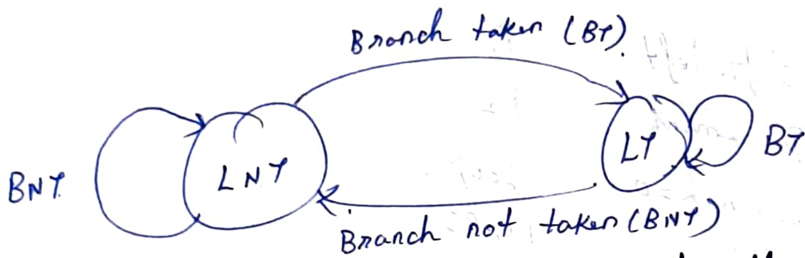
The prediction decision may change depending on execution history.

The objective is to reduce probability of making wrong decision.

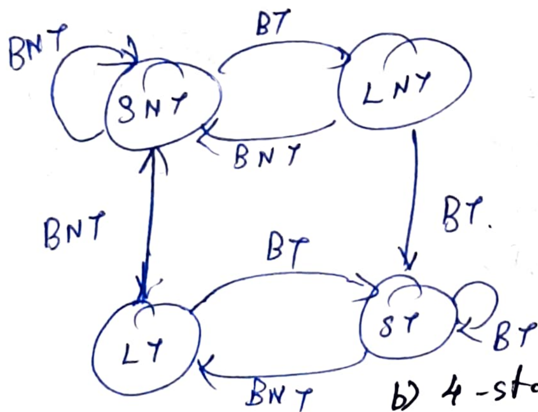
Two states:

LT : Branch is likely to be taken.

LNT : Branch is likely not to be taken.



a) 2-state algorithm.



b) 4-state algorithm.

Fig 8.15 Representation of Branch prediction algorithm.

Four stages:

ST : strongly likely to be taken.

LT : Likely to be taken.

LNT : Likely not to be taken.

SNT : Strongly likely not to be taken.