MODULE 5

PIPELINE DESIGN

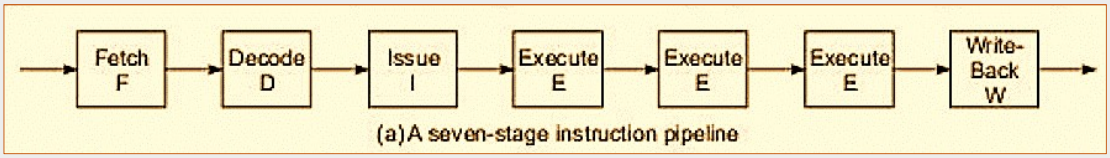
INSTRUCTION PIPELINE DESIGN

A program consist of multiple instructions, and incase of pipeline architecture each and every instructions will executed in multiple different phases or stages or segments.

A typical instruction execution consist of a sequence of operations or phases. They are

- Instruction fetch
- Decode (Identify the meaning)
- Operand fetch (Data)
- Execute
- Write Back (Store result)

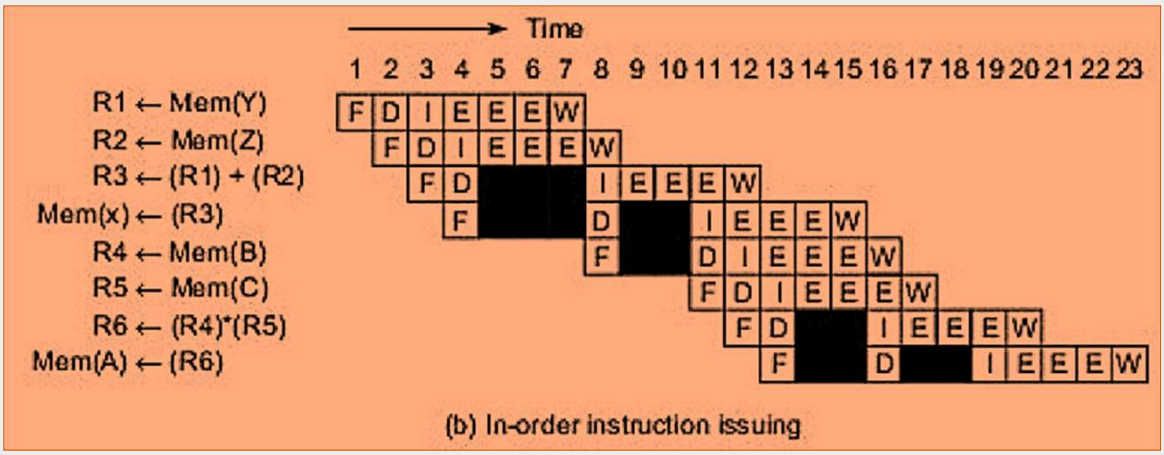
Consider the following instruction pipeline:



(a)A seven-stage instruction pipeline

- Fetch stage fetches instructions from a cache memory (one per cycle)
- Decode stage identifies the function to be performed and identifies the resources needed. (Resources- GPRs, Functional unit)
- Issue stage reserves the resources, and read operands from registers
- Execution – Instructions executed in several execute stages (3 execute stage are shown)
- Write back stage used to write results into the registers.

- The following figure shows the flow of machine instructions through a typical pipeline. It shows the 8 instructions of the high-level language statements X=Y+Z and A=B×C



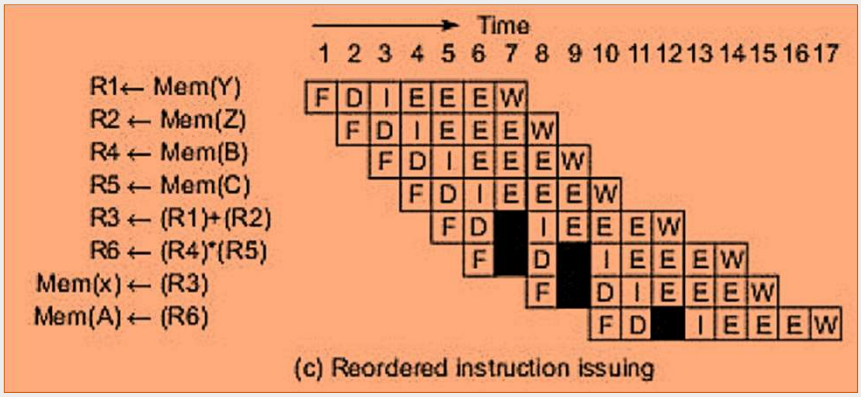
(b) In-order instruction issuing

➢With in-order instruction issuing, if an instruction is blocked from issuing due to a data or resource dependence, all instructions following it are blocked.

- The shaded boxes correspond to idle cycle when instruction issues are blocked due to resource latency, conflicts or due to data dependences.
- First two load instructions issue on consecutive cycles. The add is dependent on both loads and must wait 3 cycles before the data (X and Y) are loaded in
- The store of the sum to memory location X must wait 3 cycles for the add to finish due to a flow dependence.

➢The following Fig: shows an improved timing after the instruction issuing order is changed. The idea is to issue all 4 load operations in the beginning.

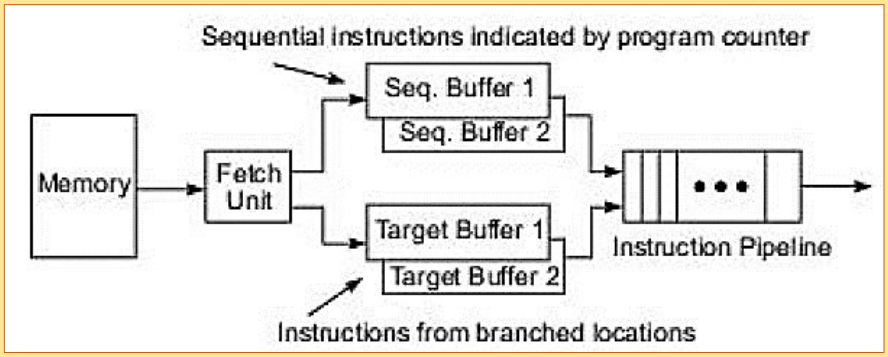
➢Both the add and multiply instructions are blocked fewer cycles due to this data prefetching. The reordering should not change the end result.



(c) Reordered instruction issuing

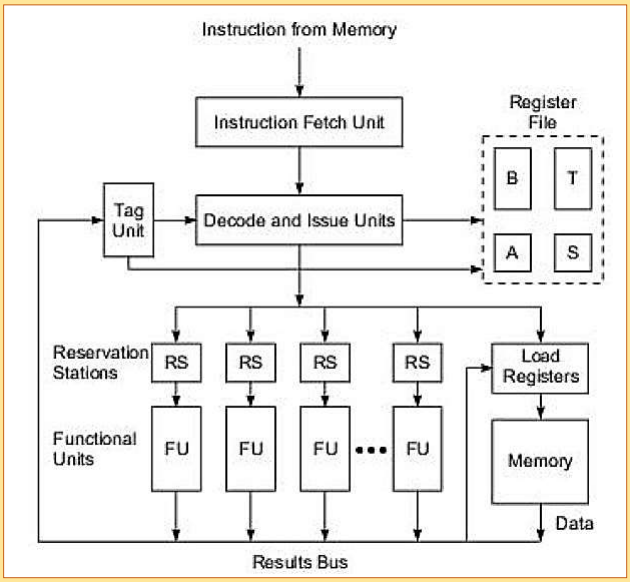
MECHANISMS FOR INSTRUCTION PIPELINING

❖Prefetch buffers



- In one memory access time, a block of consecutive instructions are fetched in to prefetch buffer.
- Block access can be achieved using interleaved memory modules or using a cache.
- Sequential instructions are loaded in to sequential buffers
- Instructions from branched locations are loaded into target buffers
- Both buffers operate in FIFO fashion
- Another prefetch buffer is loop buffer. It holds sequential instructions contained in a small loop.
- The CDC 6600 and cray1 use the loop buffer.

❖Multiple Functional Units

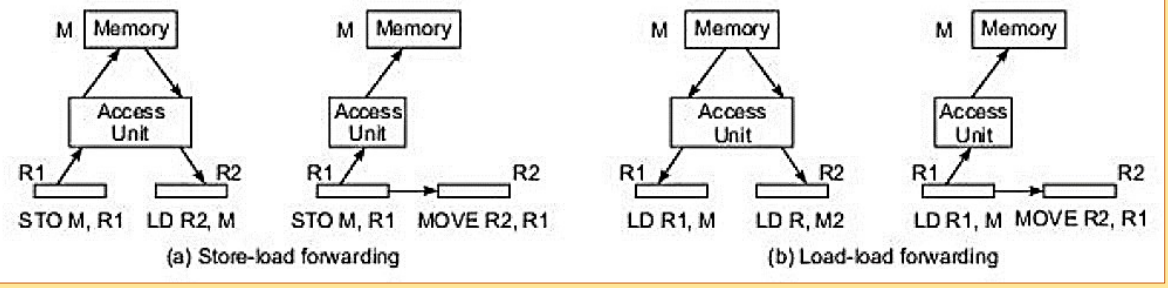


- Some times a certain pipeline stage becomes a bottleneck. This problem can be removed by using multiple copies of the same stage simultaneously.
- Consider the above figure: - to resolve data or resource dependences among the successive instructions entering the pipeline, the reservation stations(RS) are used with each functional units.
- Operations wait in the RS until their data dependences have been resolved.
- Each RS is uniquely identified by “tag”, which is monitored by tagunit.
- The tagunit check the tags from all currently used registers or RSs.

- Register tagging technique allows the hardware to resolve conflicts between source and destination registers assigned for multiple instructions.
- Once the dependences are resolved, the multiple functional units can operate in parallel.

❖Internal Data Forwarding

- Throughput of pipelined processor can be improved with internal data forwarding among multiple functional units
- Memory access operations can be replaced by register transfer operations
- Since register transfer is faster than memory access, this data forwarding reduce memory traffic.

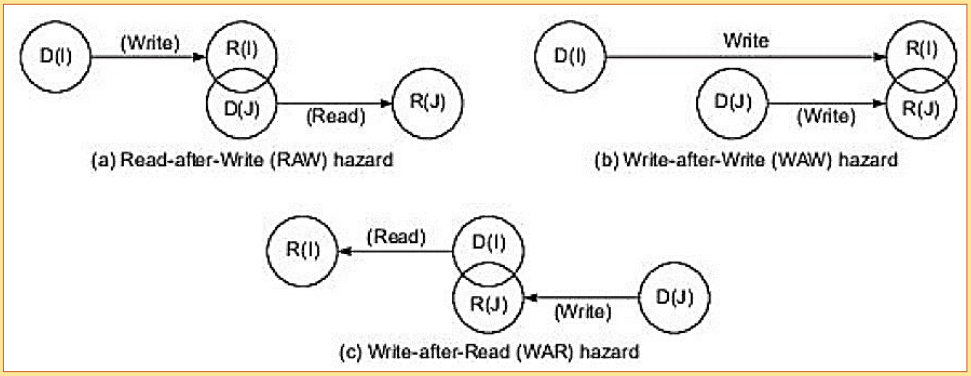


(a) Store-load forwarding (b) Load-load forwarding

❖Hazard Avoidance

- Consider 2 instructions I and J. Instruction J is logically follow instruction I. If the actual execution order of these two instructions violates the program order, incorrect result may be read or written, and produce hazards.

- Let D(I) is the domain of an instruction I. Domain contains the input set (operands) used by instruction I.
- R(I) is the range of an instruction I. The range contains output set of instruction I.
- The possible hazards are given below.



- The RAW hazard corresponds to the flow dependence
- WAR to the antidependence
- WAW to the output dependence

$$R(I) \cap D(J) \neq \phi \text{ for RAW hazard}$$
$$R(I) \cap R(J) \neq \phi \text{ for WAW hazard}$$
$$D(I) \cap R(J) \neq \phi \text{ for WAR hazard}$$

INSTRUCTION SCHEDULING

The methods for scheduling instructions through an instruction pipeline are:

- Static scheduling using an optimizing compiler
- Dynamic scheduling
  - Tomasulo's Algorithm
  - Score boarding scheme

❖Static Scheduling

- Data dependences of instructions create an interlock relationship between instructions. It can be resolved through compiler base static scheduling approach.
- Using a compiler or a post processor we can increase the separation between interlocked instructions.

Instruction:		
Add	R0, R1	/R0 ← (R0) + (R1)/
Move	R1, R5	/R1 ← (R5)/
Load	R2, M(α)	/R2 ← (Memory (α))/
Load	R3, M(β)	/R3 ← (Memory (β))/
Multiply	R2, R3	/R2 ← (R2) × (R3)/

- Consider the above code. Here, the multiply instruction cannot be initiated until the preceding load is complete. This data dependence will stall the pipeline, for 3 clock cycles.
- The two Load instructions are independent of the add and move instructions. So we can move these instructions to increase the spacing between them and multiply instruction.
- After modification we get,

Load	R2, M(α)
Load	R3, M (β)
Add	R0, R1
Move	R1, R5
Multiply	R2, R3

❖Dynamic Scheduling

- Here, the hardware rearranges the instruction execution to reduce the stall.
- Dynamic scheduling simplifies the compiler.
- Dynamic scheduled processor cannot change the data flow, it tries to avoid stalling when dependences are present.
- In contrast, static pipeline scheduling by the compiler, tries to minimize stalls by separating dependent instructions so that they will not lead to hazards.

Tomasulo's Approach

- This scheme resolved resource conflict and data dependence using register tagging to allocate and deallocate the source and destination registers.
- It eliminates WAR and WAW hazards
- An issued instructions whose operands are not available is forwarded to an RS (Reservation Station) associated with the functional units it will use.
- It waits until its data dependence have been resolved and its operands become available. The dependence is resolved by monitoring the CDB (Common Data Bus)

- The dependence is resolved by monitoring the result bus. When an instruction has completed the execution, the result along with its tag appears on the result bus.
- The registers as well as the reservation stations monitor the result bus and update their contents when a matching tag is found.
- It is used in high performance computers.
- It needs special hardware units including reservation stations & multiple functional units.

Score boarding

- Unlike out of order execution, this technique issues instructions inorder (in-order-issue).
- Score boarding is a hardware mechanism that maintains an execution rate of one instructions per cycle, by executing an instruction as soon as its operands are made available, and no hazard conditions prevent it.
- Every instructions go through a score board where a record of data dependences is constructed corresponding to instruction issue.
- A system with a scoreboard is assumed to have several functional units with their status information reported to the score board.

- If the scoreboard determines that an instruction cannot execute immediately, it executes another waiting instruction and keeps monitoring hardware unit status and decides when the instruction can proceed to execute.
- All hazard detection and resolution are centralized in scoreboarding

TOMASULO'S APPROACH	SCOREBOARDING
Register renaming is used to eliminate WAR and WAW hazards	It must wait for WAR and WAW hazards to clear
Hazard detection and execution control is distributed to each functional unit	Hazard detection and execution control is centralized
Forwards results directly to the functional units	Result is forwarded to the register file

ARITHMETIC PIPELINE DESIGN

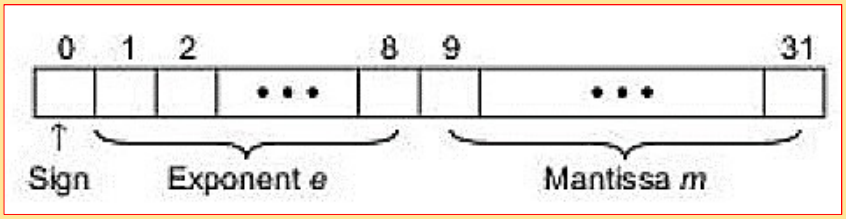
- In modem processors, fixed-point (integer) and floating-point arithmetic operations are very often performed by separate hardware on the same processor chip
- IEEE has developed standard formats for 32- and 64-bit floating numbers known as the IEEE 754 Standard.
- This standard has been adopted for most of today's computers.

❖Fixed-Point Operations

- Most computers use the two's complement notation because of its unique representation of all numbers (including zero).
- Add, subtract, multiply and divide are the four primitive arithmetic operations. The add or subtract of two n-bit integers produces an n-bit result with at most one carry-out.
- The multiplication of two n-bit numbers produces a 2n-bit result which requires the use of two memory words or two registers to hold the full-precision result.
- The division of an n-bit number by another may create an arbitrarily long quotient and a remainder.

❖Floating-Point Operations

- A floating-point number X is represented by a pair **(m,e)** , when: m is the mantissa and e is the exponent.
- A 32-bit floating—point number is specified in the IEEE 754 Standard as follows



❖Static Arithmetic Pipelines

- The ALU perform fixed-point and floating-point operations separately.
- The fixed-point unit is also called the integer unit. The floating-point unit can be built either as part of the central processor or on a separate coprocessor.
- These arithmetic units perform scalar operations. The pipelining in scalar arithmetic pipelines is controlled by software loops. Vector arithmetic units can be designed with pipeline hardware directly under firmware or hardwired control

Arithmetic Pipeline Stages

- Since all arithmetic operations can be implemented with the basic add and Shifting operations, the core arithmetic stages require some form of hardware to add and to shift.

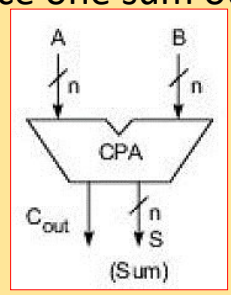
For example, a typical three stage floating-point adder

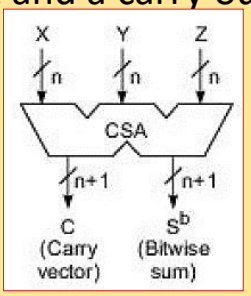
- first stage for exponent comparison and equalization which is implemented with an integer adder and some shifting logic
- second stage for fraction addition using a high-speed carry lookahead adder
- third stage for fraction normalization and exponent readjustment using a shifter and another addition logic.

➢Arithmetic or logical shifts can be easily implemented with shift registers.

➢High-speed addition requires

- either the use of a carry-propagation adder (CPA) which adds two numbers and produces an arithmetic sum
- Or the use of a carry-save adder (CSA) to three input numbers and produce one sum output and a carry output



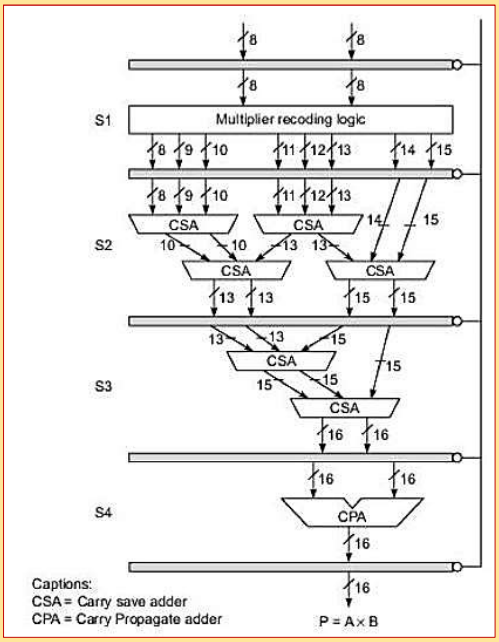


Multiply Pipeline Design

Consider as an example the multiplication of two 8-bit integers A×B=P where P is the 16 –bit product. This fixed-point multiplication can be written as the summation of eight partial products as shown below

							1	0	1	1	0	1	0	1	=	A
						×)	1	0	0	1	0	0	1	1	=	B
							1	0	1	1	0	1	0	1	=	P_0
						1	0	1	1	0	1	0	1	0	=	P_1
					0	0	0	0	0	0	0	0	0	0	=	P_2
				0	0	0	0	0	0	0	0	0	0	0	=	P_3
			1	0	1	1	0	1	0	1	0	0	0	0	=	P_4
		0	0	0	0	0	0	0	0	0	0	0	0	0	=	P_5
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	=	P_6
+)	1	0	1	1	0	1	0	1	0	0	0	0	0	0	=	P_7
0	1	1	0	0	1	1	1	1	1	1	0	1	1	1	=	P

Multiply Pipeline Design



- The first stage (S1) generates all eight partial products, ranging from 8 bits to 15 bits, simultaneously.
- The second stage (S2) is made up of two levels of four CSAs, and it essentially merges eight numbers into four numbers ranging from I3 to I5 bits.
- The third stage (S3) consists of two CSAs, and it merges four numbers from S1 into two l6-bit numbers. ‘
- The final stage (S4) is a CPA, which adds up the last two numbers to produce the final product P.

SUPERSCALAR PIPELINE DESIGN

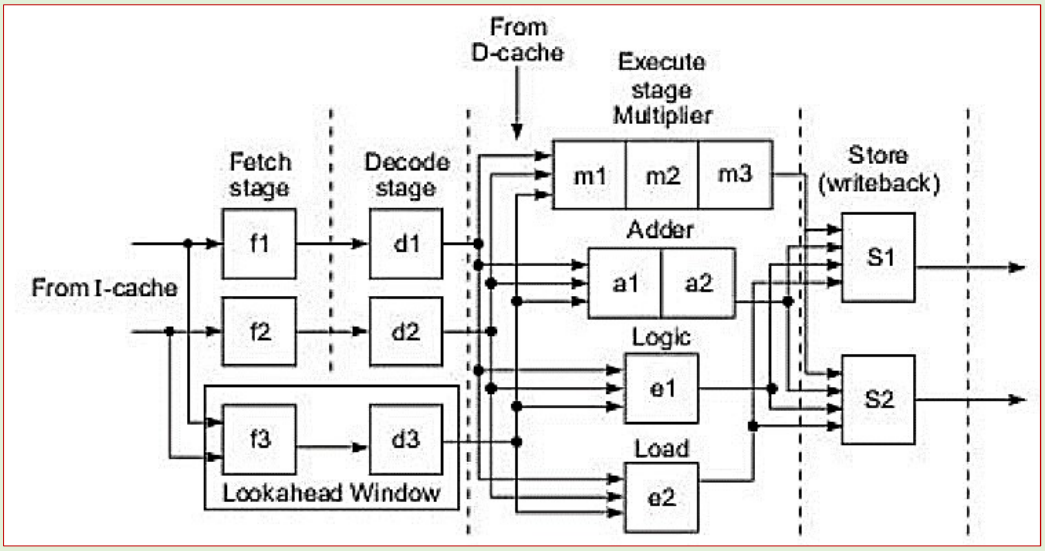
Pipeline Design Parameters

A comparison between scalar & superscalar processors

Machine type	*Scalar base machine of k pipeline stages*	*Superscalar machine of degree m*
Machine pipeline cycle	1 (base cycle)	1
Instruction issue rate	1	*m*
Instruction issue latency	1	1
Simple operation latency	1	1
ILP to fully utilize the pipeline	1	*m*

- The Instruction Level Parallelism (ILP) is the maximum number of instructions that can be simultaneously executed in the pipeline.

Super Scalar Pipeline Structure



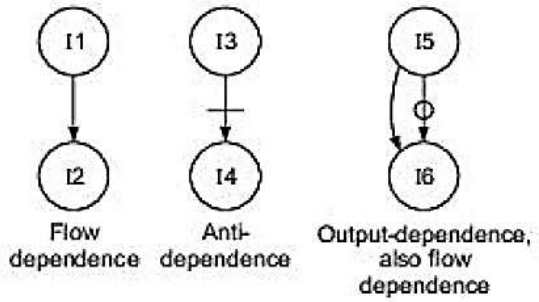
- In this design, the processor can issue two instructions per cycle if there is no resource conflict and no data dependence problem.
- There are essentially two pipelines in the design.
- Both pipelines have four processing stages labeled fetch, decode, execute, and store, respectively.
- Each pipeline essentially has its own fetch unit. decode unit. and store unit.
- The two instruction streams flowing through the two pipelines are retrieved from a single source stream (the I-cache).
- The fan-out from a single instruction stream is subject to resource constraints and a data dependence relationship among the successive instructions.

- Four functional units, multiplier, adder, logic unit, and load unit, are available for use in the execute stage. These functional units are shared by the two pipelines on a dynamic basis.
- The multiplier itself has three pipeline stages, the adder has two stages, and the others each have only one stage.
- There is a lookahead window with its own fetch and decoding logic. This window is used for instruction lookahead in case out-of-order instruction issue is desired to achieve better pipeline throughput.
- It requires complex logic to schedule multiple pipelines simultaneously. The aim is to avoid pipeline stalling and minimize pipeline idle time

➢A sample program and its dependence graph shown below, where I_2 and I_3 share the adder and I_4 and I_6 share the multiplier

Program order

I1.	Load	R1,	A	/R1 ← Memory (A)/
I2.	Add	R2,	R1	/R2 ← (R2) + (R1)/
I3.	Add	R3,	R4	/R3 ← (R3) + (R4)/
I4.	Mul	R3,	R5	/R4 ← (R4) * (R5)/
I5.	Comp	R6		/R6 ← ($\overline{R6}$)/
I6.	Mul	R6,	R7	/R6 ← (R6) * (R7)/



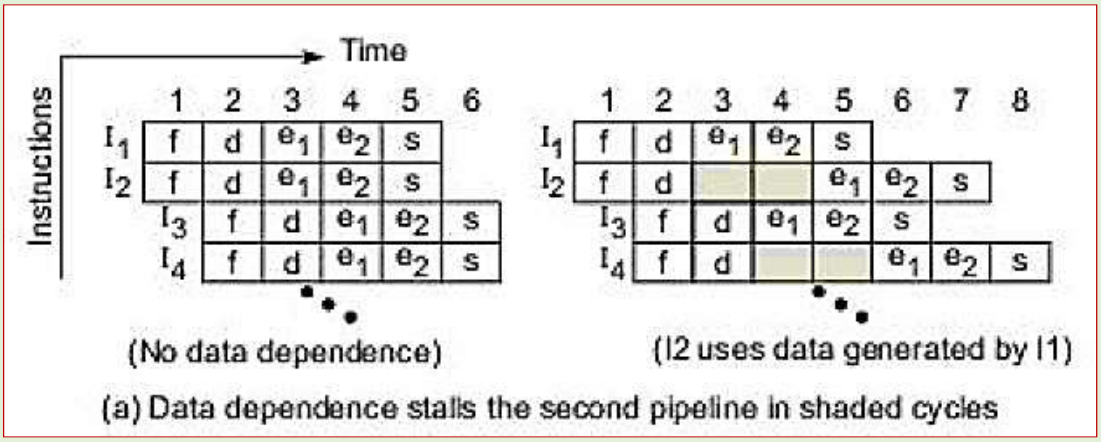
- Because the register content in R1 is loaded by I_1 and then used by I_2, we have flow dependence: **I_1—> I_2.**

- Because the result in register R_4 after executing I_4 may affect the operand register R_4 used by I_3, we have anti-dependence.
- Since both I_5 and I_6 modify the register R_6, and R_6 supplies an operand for I_6, we have both flow and output dependence
- To schedule instructions through one or more pipelines, these data dependences must not be violated. Otherwise, erroneous results may be produced.

❖Pipeline Stalling

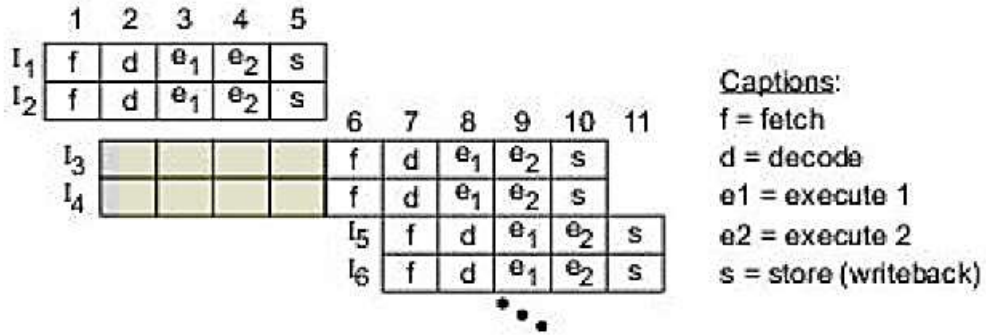
- This is a problem which may seriously lower pipeline utilization.
- Proper scheduling avoids pipeline stalling. The problem exists in both scalar and superscalar processors.
- However, it is more serious in a superscalar pipeline. Stalling can be caused by data dependences or by resource conflicts among instructions already in the pipeline or about to enter the pipeline
- Following figure shows the case of no data dependence on the left and flow dependence ($I_1 \rightarrow I_2$) on the right. Without data dependence all pipeline stages are utilized without idling.

- With dependence, instruction I_2 entering the second pipeline must wait for two cycles (shaded time slots) before entering the execution stages. This delay may also pass to the next instruction I_4 entering the pipeline.



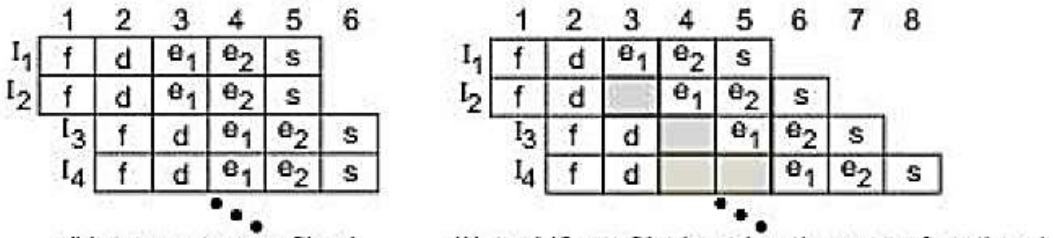
(a) Data dependence stalls the second pipeline in shaded cycles

- The following fig: shows the effect of branching (instruction I_2). A delay slot of four cycles results from a branch taken by I_2 at cycle 5. Therefore, both pipelines must be flushed before the target. instructions I_3 and I_4 can enter the pipelines from cycle 6.



(b) Branch instruction I2 causes a delay slot of length 4 in both pipelines

- The following fig: shows a combined problem involving both resource conflict and data dependence. Instructions I_1 and I_2 need to use the same functional unit, and $I_2 \rightarrow I_4$ exists.



(No resource conflicts)

(I1 and I2 conflict in using the same functional unit, and I4 uses data generated by I2)

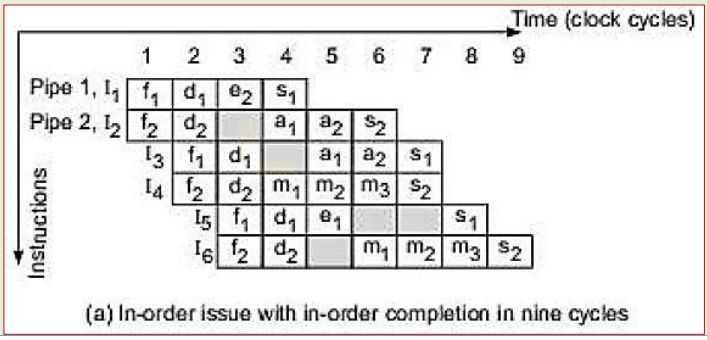
(c) Resource conflicts and data dependences cause the stalling of pipeline operations for some cycles

❖Superscalar Pipeline Scheduling

- The scheduling policies are introduced below.
- When instructions are issued in program order, we call it in-order issue.
- When program order is violated, out-of-order issue is being practiced.
- In-order issue is easier to implement but may not yield the optimal performance. In-order issue may result in either in-order or out-of-order completion.
- The purpose of out-of-order issue and completion is to improve performance

❖In-Order issue with In-Order Completion

- Figure shows a schedule for the six instructions being issued in program order $I_1, I_2, \ldots I_6$.
- Pipeline 1 receives I_1, I_3, and I_5, and pipeline 2 receives I_2, I_4, and I_6 in three consecutive cycles. Due to I_1 —> I_2, I_2 has to wait one cycle to use the data loaded in by I_1.

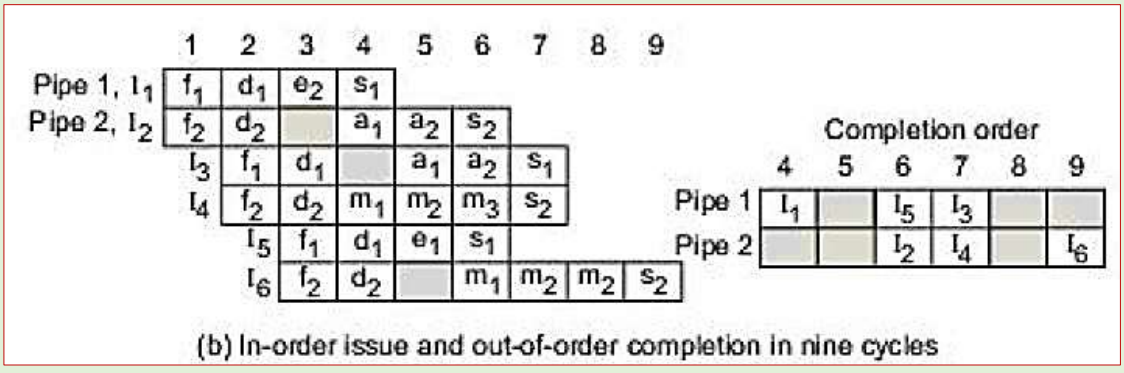


(a) In-order issue with in-order completion in nine cycles

- I_3 is delayed one cycle for the same adder used by I_2.
- I_6 has to wait for the result of 1_5 before it can enter the multiplier stages.
- In order to maintain in-order completion, I_5 is forced to wait for two cycles to come out of pipeline 1.
- In total, nine cycles are needed and five idle cycles (shaded boxes) are observed

❖In-Order issue with Out-of-Order Completion

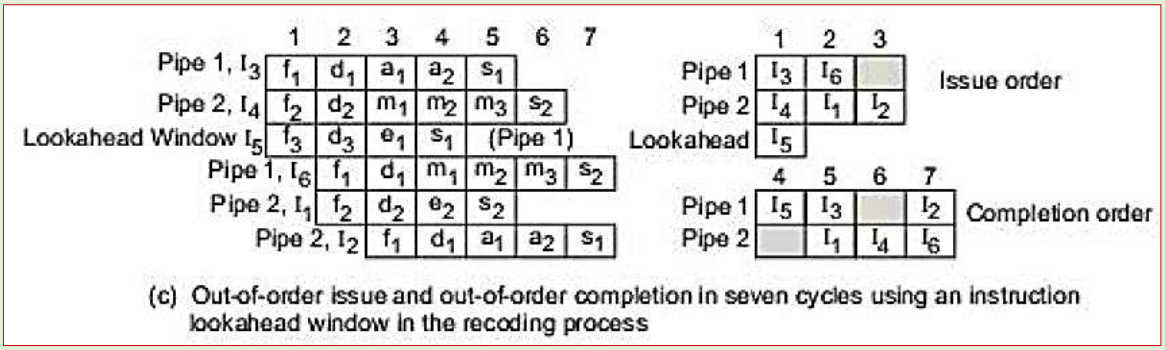
- The only difference between this out-of-order schedule and the in-order schedule is that I_5 is allowed to complete ahead of I_3 and I_4, which are totally independent of I_5. The total execution time does not improve. However, the pipeline utilization rate does.



(b) In-order issue and out-of-order completion in nine cycles

❖Out-of-Order issue with Out-of-Order Completion

- In order to shorten the total execution time, the window can be used to reorder the instruction issues.



(c) Out-of-order issue and out-of-order completion in seven cycles using an instruction lookahead window in the recoding process

- By using the lookahead window, instruction I_5 can be decoded in advance because it is independent of all the other instructions.
- The six instructions are issued in three cycles as shown: I_5 is fetched and decoded by the window, while I_3 and I_4 are decoded concurrently.
- Because the issue is out of order, the completion is also out of order
- The total execution time has been reduced to seven cycles with no idle stages during the execution of these six instructions.
- Out-of-order issue gives the processor more freedom to exploit parallelism, and thus pipeline efficiency is enhanced.