Computer Architecture and Organization
Pipelining "Instruction Level Parallelism": Part I"

Topics

- Review of "Pipelining"
- Pipelining
  - Data path Modification
  - Control path Modification
- Hazards
  - Data Hazards
  - Structural Hazards
  - Control Hazards

With thanks to W. Stallings, Hamacher, J. Hennessy, M. J. Irwin for lecture slide contents
Many slides adapted from the PPT slides accompanying the textbook and CSE331 Course
Prof. Anand Motwani

2

Pipelining

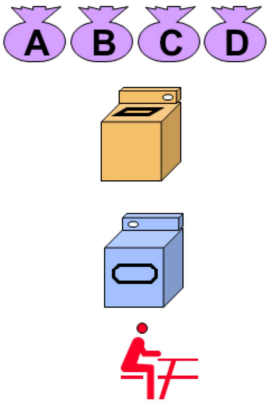
Prof. Anand Motwani 3

Entry Ticket

- What are Instruction Cycle stages?
- Four registers are essential to instruction execution. Names?
- How a CPU operation can be improved? Tell two ways to improve performance?

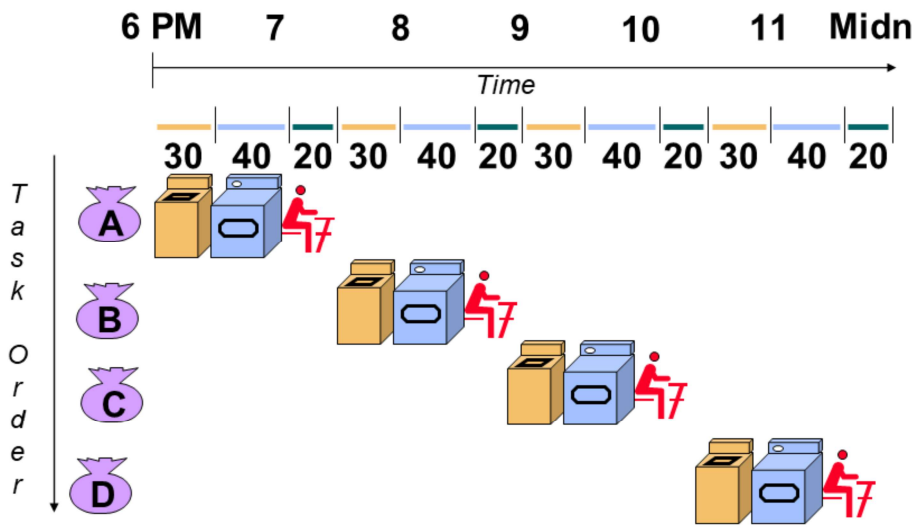
Pipelining: Example

- Laundry
- Alkesh, Bhavna, Chetanya, Dave
  each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- "Folder" takes 20 minutes



Prof. Anand Motwani 5

Sequential Laundry



- Sequential laundry takes (90 x 4 = 360 minutes) 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

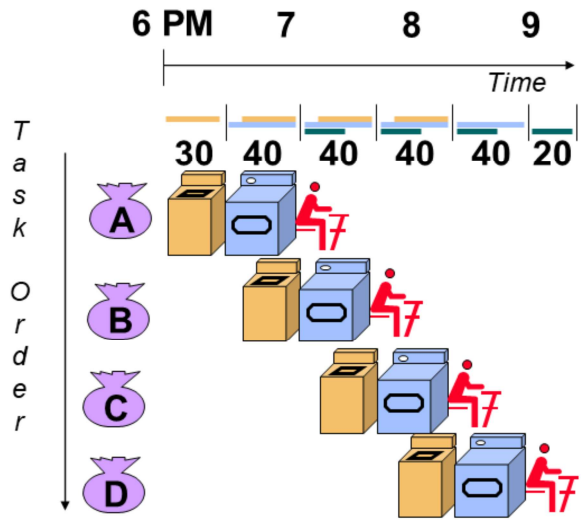
Pipelining

Pipelining ***exploits parallelism*** at the instruction level.

- Pipelining is an implementation technique in which ***multiple instructions are overlapped in execution***.

- Today pipelining is key to making processors fast.

Pipelining Lessons



- Tot Time: 210 minutes!! versus 360 with no pipelining
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Pipelining doesn't help latency of single task, it helps throughput of entire workload

Prof. Anand Motwani 8

Assembly Line Analogy to Data Path Pipeline

- A custom product being built may pass the assembly line many times before it is completed.
- A conveyor belt moves components from stage to stage
- This technique increases throughput

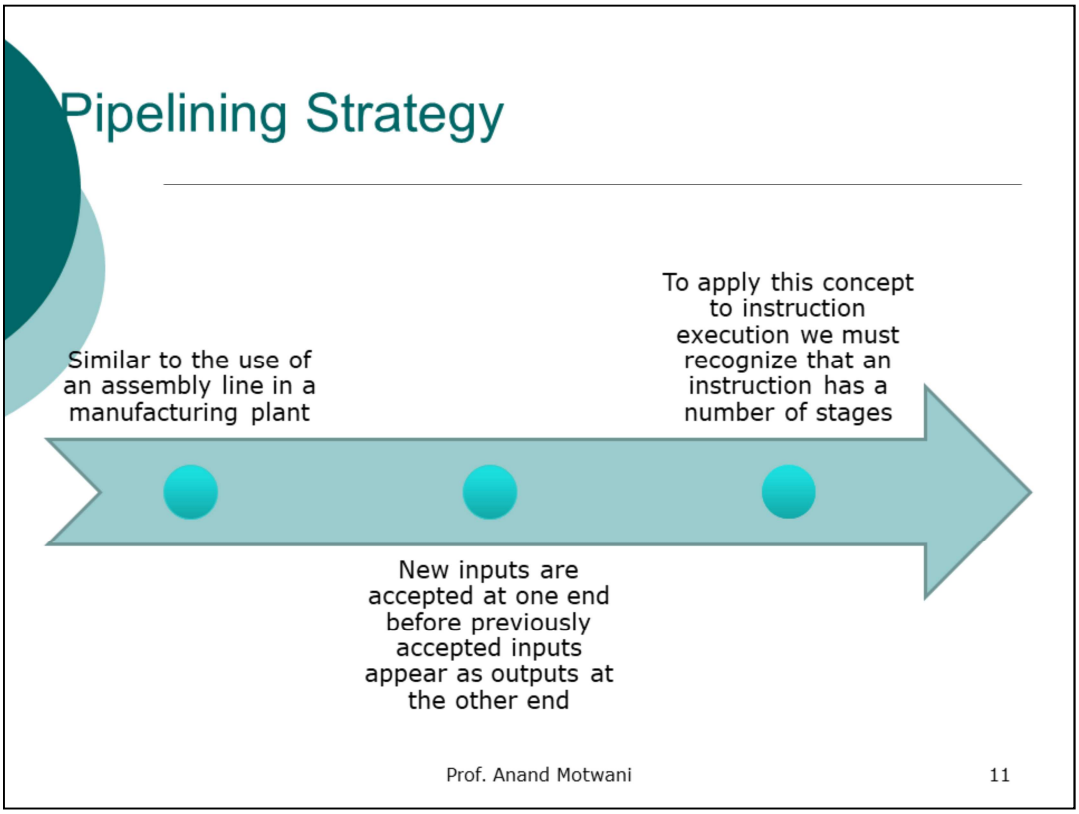
Instruction Pipeline

- Pipelining in instruction stream is termed as Instruction pipelining.
- Instruction pipeline reads consecutive instructions from memory while previous instruction are executed in other segments.
- Fetch and execute phases of instruction are overlapped for simultaneous operations.
- Disadvantage : if a branch instruction comes then the pipeline must be emptied and all instruction following the branch instruction are discarded. This degrade the performance.

Instruction cycle : To execute an instruction a computer needs the following sequence of steps called instruction cycle.

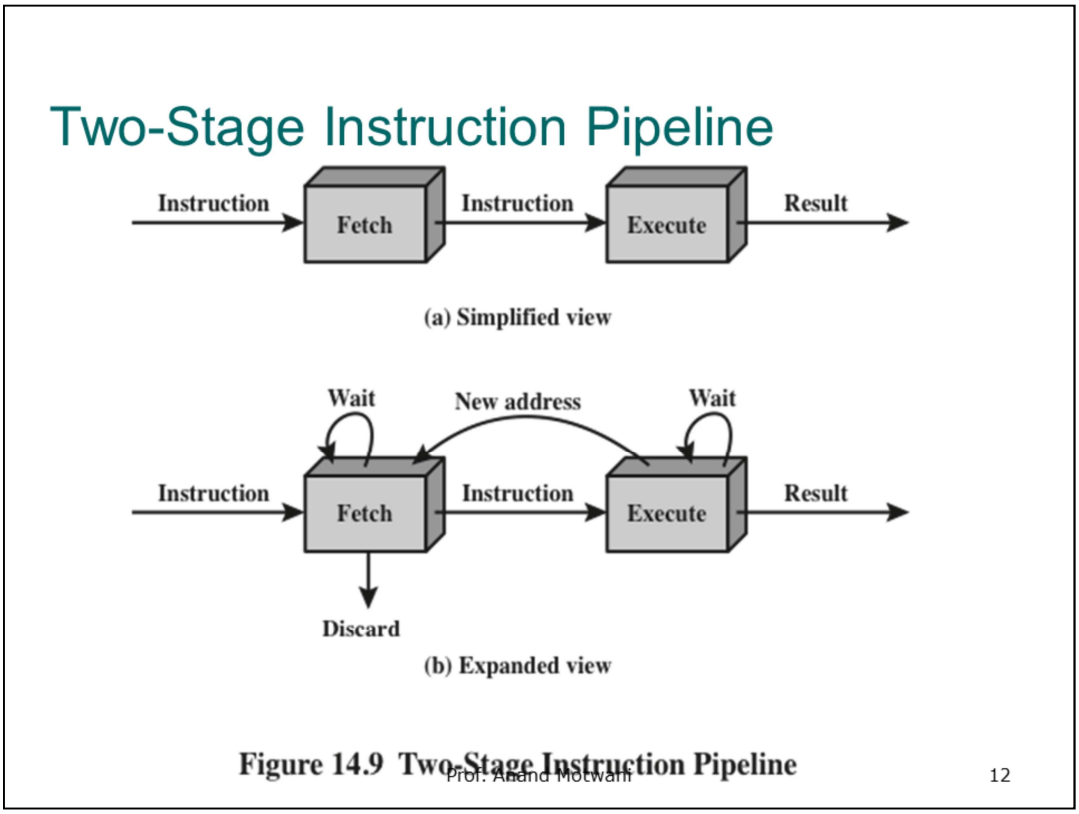
1. Fetch instruction from memory.
2. Decode the instruction.
3. Calculate effective address.
4. Fetch the operands from memory.
5. Execute the instruction.
6. Store the result in memory.

Prof. Anand Motwani 10



Instruction pipelining is similar to the use of an assembly line in a manufacturing plant. An assembly line takes advantage of the fact that a product goes through various stages of production. By laying the production process out in an assembly line, products at various stages can be worked on simultaneously. This process is also referred to as *pipelining,* because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

To apply this concept to instruction execution, we must recognize that, in fact, an instruction has a number of stages. Figures 14.5, for example, breaks the instruction cycle up into 10 tasks, which occur in sequence. Clearly, there should be some opportunity for pipelining.



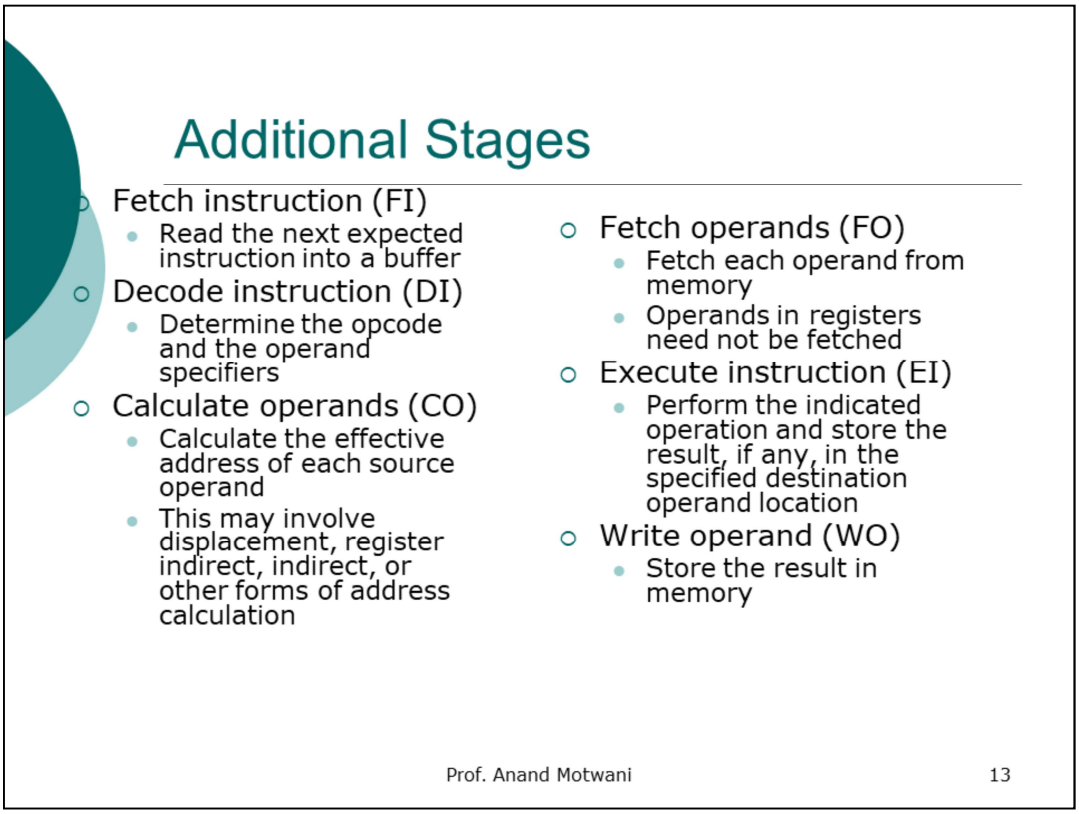
As a simple approach, consider subdividing instruction processing into two stages: fetch instruction and execute instruction. There are times during the execution of an instruction when main memory is not being accessed. This time could be used to fetch the next instruction in parallel with the execution of the current one. Figure depicts this approach. The pipeline has two independent stages. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called instruction prefetch or *fetch overlap.* Note that this approach, which involves instruction buffering, requires more registers. In general, pipelining requires registers to store data between stages.

It should be clear that this process will speed up instruction execution. If the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Figure 14.9b), we will see that this doubling of execution rate is unlikely for two reasons:

The execution time will generally be longer than the fetch time. Execution will involve reading and storing operands and the performance of some operation. Thus, the fetch stage may have to wait for some time before it can empty its buffer.

A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

Guessing can reduce the time loss from the second reason. A simple rule is the following: When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost. If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.



While these factors reduce the potential effectiveness of the two-stage pipe- line, some speedup occurs. To gain further speedup, the pipeline must have more stages. Let us consider the following decomposition of the instruction processing.

Fetch instruction (FI): Read the next expected instruction into a buffer.

Decode instruction (DI): Determine the opcode and the operand specifiers.

Calculate operands (CO): Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.

Fetch operands (FO): Fetch each operand from memory. Operands in registers need not be fetched.

Execute instruction (EI): Perform the indicated operation and store the result, if any, in the specified destination operand location.

Write operand (WO): Store the result in memory.

With this decomposition, the various stages will be of more nearly equal duration.

Timing Diagram for Instruction Pipeline Operation

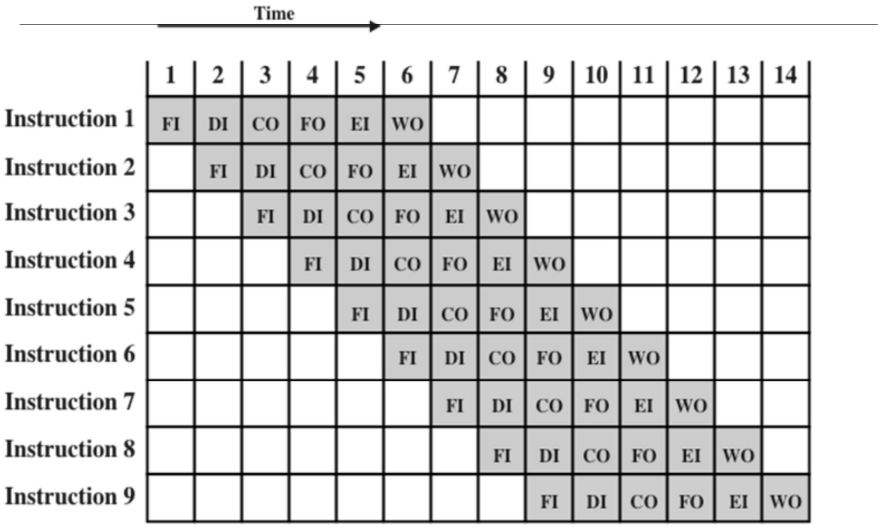


Figure 14.10 Timing Diagram for Instruction Pipeline Operation

For the sake of illustration, let us assume equal duration. Using this assumption, Figure 14.10 shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

Several comments are in order: The diagram assumes that each instruction goes through all six stages of the pipeline. This will not always be the case. For example, a load instruction does not need the WO stage. However, to simplify the pipeline hardware, the timing is set up assuming that each instruction requires all six stages. Also, the diagram assumes that all of the stages can be performed in parallel. In particular, it is assumed that there are no memory conflicts. For example, the FI, FO, and WO stages involve a memory access. The diagram implies that all these accesses can occur simultaneously. Most memory systems will not permit that. However, the desired value may be in cache, or the FO or WO stage may be null. Thus, much of the time, memory conflicts will not slow down the pipeline.

The Effect of a Conditional Branch on Instruction Pipeline Operation

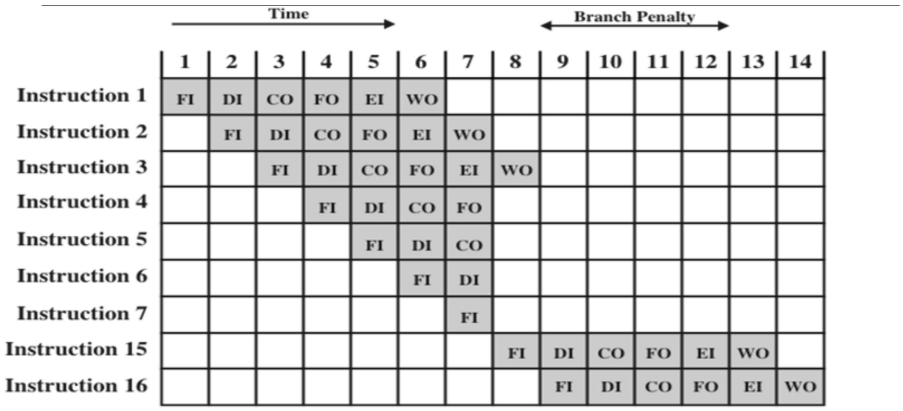


Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

Not all instructions go thru all six stages; FI, FO, & WO stages involve a memory access, i.e., potential memory conflicts; however desired values may be in cache or one or more stages may be null; CO stage may be dependent upon register contents to be modified by previous instruction still in the pipeline;

Prof. Anand Motwani 15

Several other factors serve to limit the performance enhancement. If the six stages are not of equal duration, there will be some waiting involved at various pipe- line stages, as discussed before for the two-stage pipeline. Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches. A similar unpredictable event is an interrupt. Figure 14.11 illustrates the effects of the conditional branch, using the same program as Figure 14.10. Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is executed, there is no way of knowing which instruction will come next. The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds. In Figure 14.10, the branch is not taken, and we get the full performance benefit of the enhancement. In Figure 14.11, the branch is taken. This is not determined until the end of time unit 7. At this point, the pipeline must be cleared of instructions that are not useful. During time unit 8, instruction 15 enters the pipeline. No instructions complete during time units 9 through 12; this is the performance penalty incurred because we could not anticipate the branch.

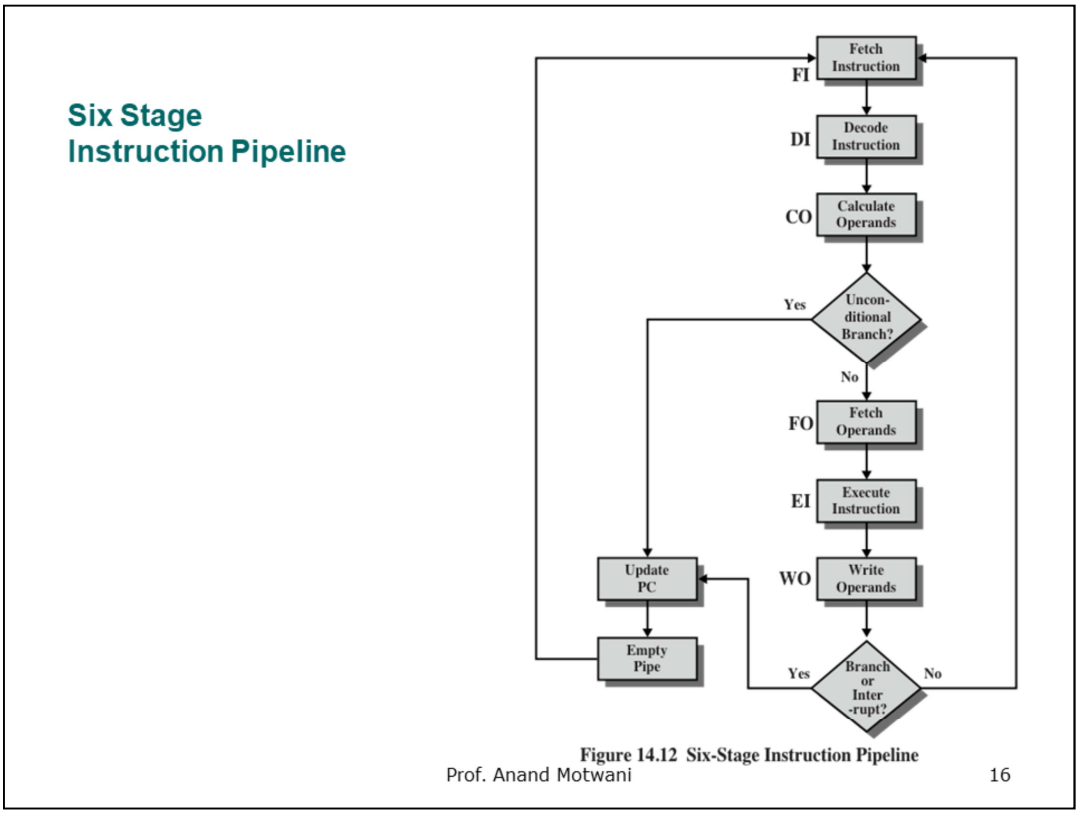
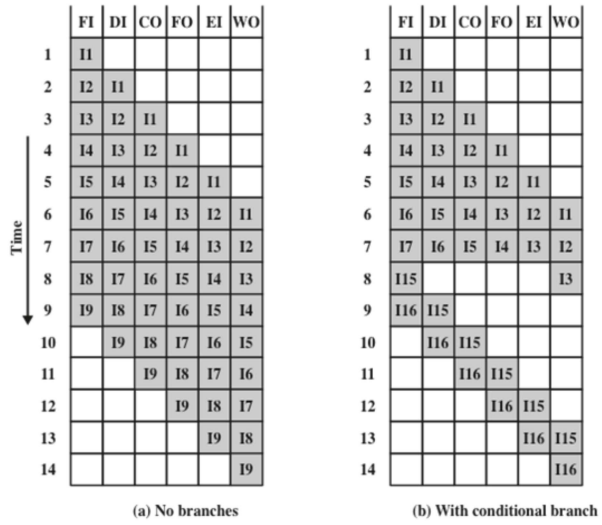


Figure 14.12 indicates the logic needed for pipelining to account for branches and interrupts.

Other problems arise that did not appear in our simple two-stage organization. The CO stage may depend on the contents of a register that could be altered by a previous instruction that is still in the pipeline. Other such register and memory conflicts could occur. The system must contain logic to account for this type of conflict.

Alternative Pipeline Depiction

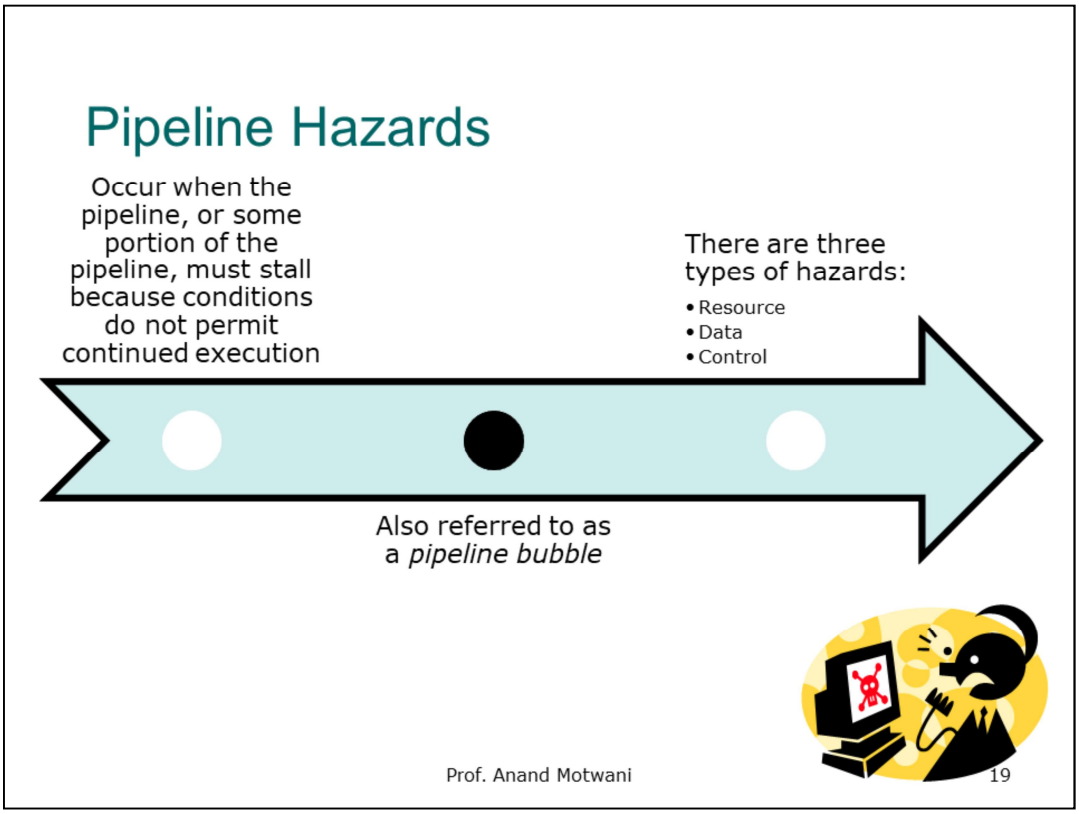


In Figure 14.13b, (which corresponds to Figure 14.11), the pipeline is full at times 6 and 7. At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15. At this point, instructions I4 through I7 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.

Figure 14.13 An Alternative Pipeline Depiction

To clarify pipeline operation, it might be useful to look at an alternative depiction. Figures 14.10 and 14.11 show the progression of time horizontally across the figures, with each row showing the progress of an individual instruction. Figure 14.13 shows same sequence of events, with time progressing vertically down the figure, and each row showing the state of the pipeline at a given point in time. In Figure 14.13a (which corresponds to Figure 14.10), the pipeline is full at time 6, with 6 different instructions in various stages of execution, and remains full through time 9; we assume that instruction I9 is the last instruction to be executed. In Figure 14.13b, (which corresponds to Figure 14.11), the pipeline is full at times 6 and 7. At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15. At this point, instructions I4 through I7 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.


Hazards
Prof. Anand Motwani
18



In the previous subsection, we mentioned some of the situations that can result in less than optimal pipeline performance. In this subsection, we examine this issue in a more systematic way. Chapter 16 revisits this issue, in more detail, after we have introduced the complexities found in superscalar pipeline organizations.

A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipe- line stall is also referred to as a *pipeline bubble.* There are three types of hazards: resource, data, and control.

1. Structural Hazards

- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches

Prof. Anand Motwani 20

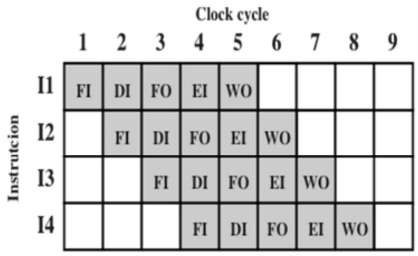
Resource Hazards

A resource hazard occurs when two or more instructions that are already in the pipeline need the same resource

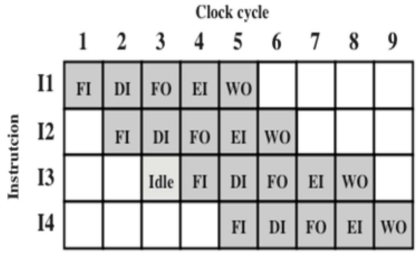
The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline

A resource hazard is sometimes referred to as a *structural hazard*

Assume a simplified five-stage pipeline, in which each stage takes one clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. Further, ignore the cache. In this case, an operand read to or write from memory cannot be performed in parallel with an instruction fetch. Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3. The figure assumes that all other operands are in registers.



(a) Five-stage pipeline, ideal case



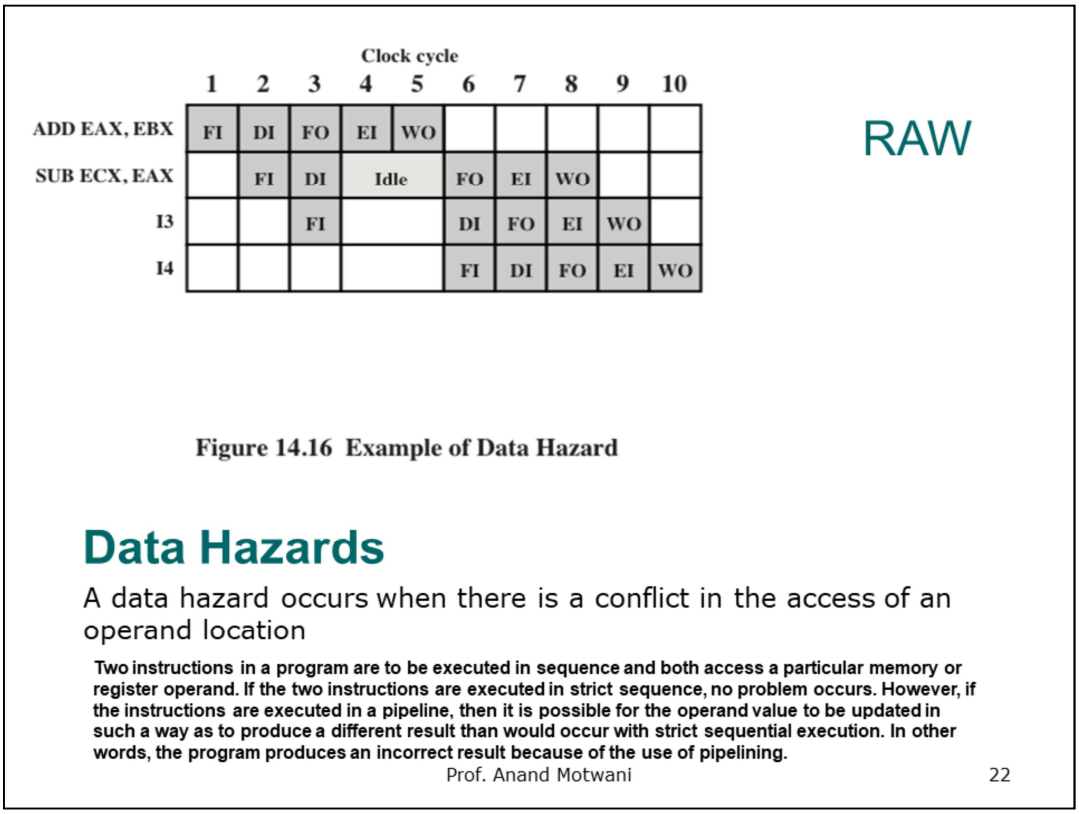
(b) I1 source operand in memory

 Figure 14.15 Example of Resource Hazard

A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline. A resource hazard is sometime referred to as a *structural hazard.*

Let us consider a simple example of a resource hazard. Assume a simplified five-stage pipeline, in which each stage takes one clock cycle. Figure 14.15a shows the ideal case, in which a new instruction enters the pipeline each clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. Further, ignore the cache. In this case, an operand read to or write from memory cannot be performed in parallel with an instruction fetch. This is illustrated in Figure 14.15b, which assumes that the source operand for instruction I1 is in memory, rather than a register. Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3. The figure assumes that all other operands are in registers.

Another example of a resource conflict is a situation in which multiple instructions are ready to enter the execute instruction phase and there is a single ALU. One solutions to such resource hazards is to increase available resources, such as having multiple ports into main memory and multiple ALU units.



A data hazard occurs when there is a conflict in the access of an operand location. In general terms, we can state the hazard in this form: Two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs. However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution. In other words, the program produces an incorrect result because of the use of pipelining.

Examples

RAW

- R1 = 5; R2 = 7; R3 = 9
- R1 <- R1+R2 (W)
- R3 <- R1+R3 (R)

- Final output R3 = 21
- In case of RAW R3 = 14

WAR

- R1 = 5; R2 = 7; R3 = 9
- R1 <- R1+R2 (R)
- R2 <- R2+R3 (W)

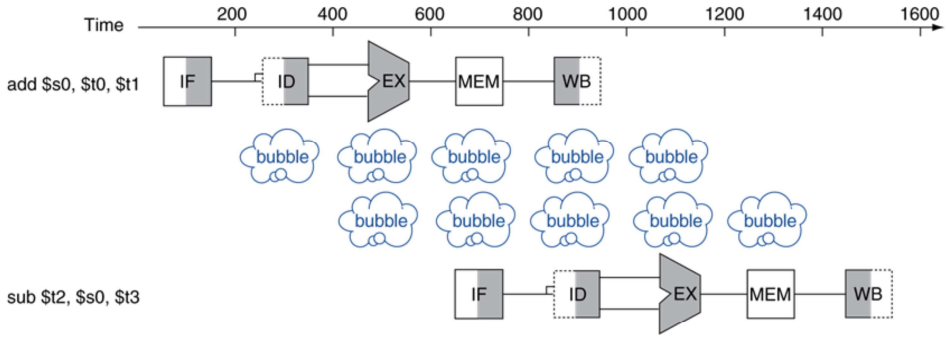
- Final output
  - R1 = 12
  - R2 = 16

- In case of WAR
  - R1 = 21
  - R2 = 16

2. Data Hazards

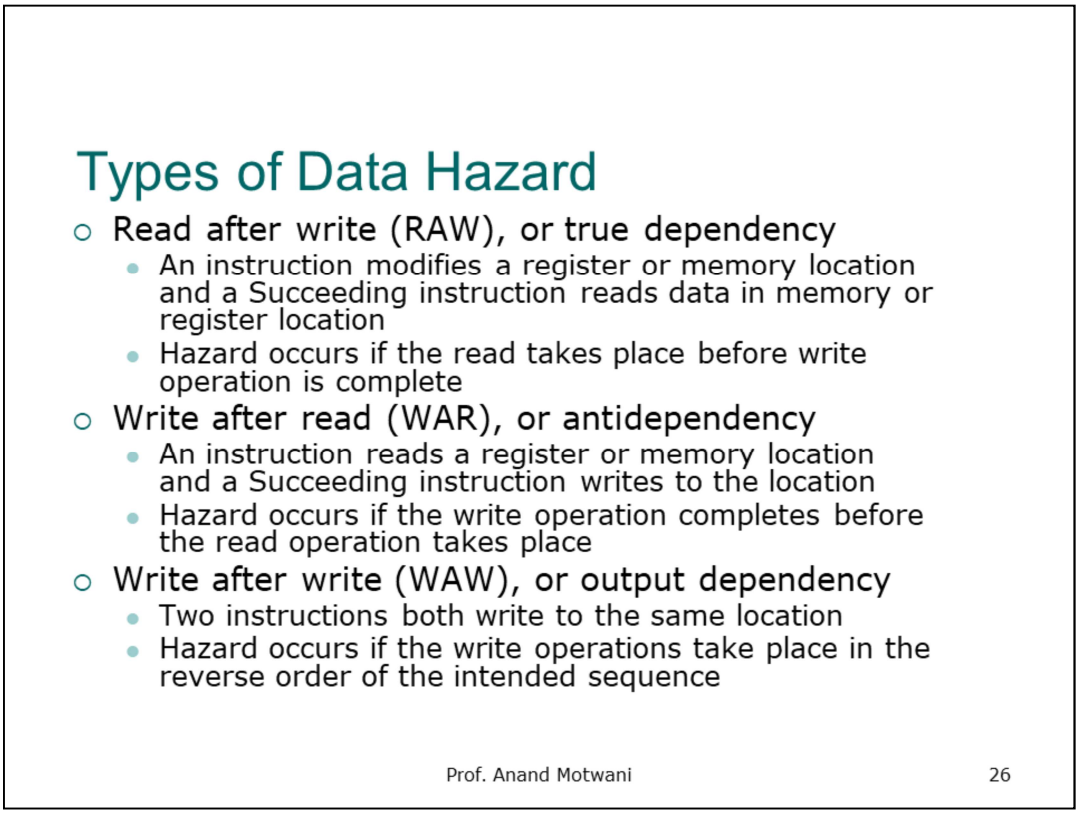
An instruction depends on completion of data access by a previous instruction

```
add   $s0, $t0, $t1
sub   $t2, $s0, $t3
```



Prof. Anand Motwani 24


Prof. Anand Motwani
25



There are three types of data hazards;

• **Read after write (RAW), or true dependency:** An instruction modifies a register or memory location and a succeeding instruction reads the data in that memory or register location. A hazard occurs if the read takes place before the write operation is complete.

• **Write after read (WAR), or antidependency:** An instruction reads a register or memory location and a succeeding instruction writes to the location. A hazard occurs if the write operation completes before the read operation takes place.

• **Write after write (WAW), or output dependency:** Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence.

The example of Figure 14.16 is a RAW hazard. The other two hazards are best discussed in the context of superscalar organization, discussed in Chapter 16.

RAW

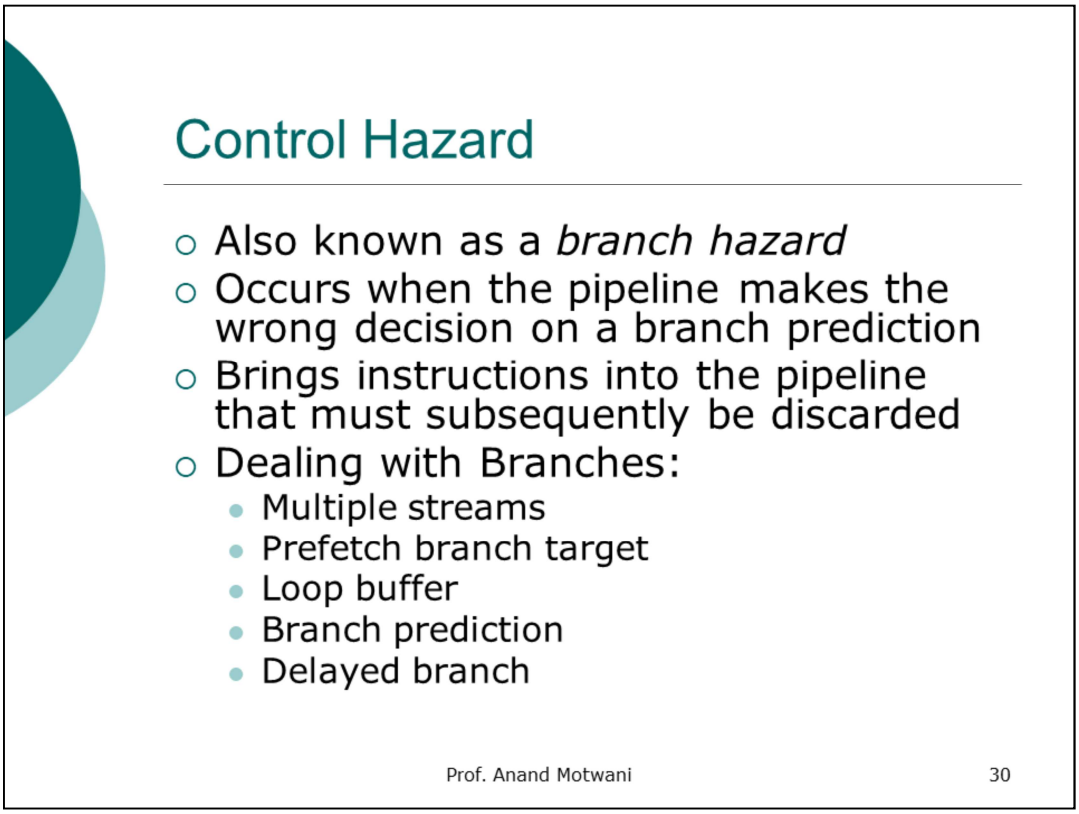
- Example: Let there be two instructions I1 and I2 such that:
  - I1 : ADD R1, R2, R3
    I2 : SUB R4, R1, R2
- RAW hazard occurs when instruction I2 tries to read data before instruction I1 writes it.

WAR

- Example: Let there be two instructions I1 and I2 such that:
  - I1 : ADD R2, R1, R3
    I2 : ADD R3, R4, R5
- WAR hazard occurs when instruction I2 tries to write data before instruction I1 reads it.

WAW

- Example: Let there be two instructions I1 and I2 such that:
  - I1 : ADD R2, R1, R3
    I2 : ADD R2, R4, R5
- WAW hazard occurs when instruction I2 tries to write output before instruction I1 writes it.



A control hazard, also known as a *branch hazard,* occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded. We discuss approaches to dealing with control hazards next.

One of the major problems in designing an instruction pipeline is assuring a steady flow of instructions to the initial stages of the pipeline. The primary impediment, as we have seen, is the conditional branch instruction. Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.

A variety of approaches have been taken for dealing with conditional branches:

Multiple streams

Prefetch branch target

Loop buffer

Branch prediction

Delayed branch

MIPS Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

Prof. Anand Motwani


End Slides
Prof. Anand Motwani
32

Numerical

1. Consider a pipeline having 4 phases with duration 60, 50, 90 and 80 ns. Given latch delay is 10 ns. Calculate-

- Pipeline cycle time
- Non-pipeline execution time
- Speed up ratio
- Pipeline time for 1000 tasks
- Sequential time for 1000 tasks
- Throughput

Solution 1:

- **Part-01: Pipeline Cycle Time-**
- Cycle time
- = Maximum delay due to any stage + Delay due to its register
- = Max { 60, 50, 90, 80 } + 10 ns = 100ns
- **Part-02: Non-Pipeline Execution Time-**
- = 60 ns + 50 ns + 90 ns + 80 ns
- = 280 ns

- **Solution 1:**
- **Speed up**
- = Non-pipeline execution time / Pipeline execution time
- = 280 ns / Cycle time
- = 280 ns / 100 ns
- = 2.8
- **Part-04: Pipeline Time For 1000 Tasks-**
- = Time taken for 1st task + Time taken for remaining 999 tasks
- = 1 x 4 clock cycles + 999 x 1 clock cycle
- = 4 x cycle time + 999 x cycle time
- = 4 x 100 ns + 999 x 100 ns = 400 ns + 99900 ns = 100300 ns
- **Part-05: Sequential Time For 1000 Tasks-**
- = 1000 x Time taken for one task = 1000 x 280 ns = 280000 ns
- **Part-06: Throughput for pipelined execution-**
- = Number of instructions executed per unit time
- = 1000 tasks / 100300 ns

2. A non-pipelined single cycle processor operating at 100 MHz is converted into a synchronous pipelined processor with five stages requiring 2.5 ns, 1.5 ns, 2 ns, 1.5 ns and 2.5 ns respectively. The delay of the latches is 0.5 sec. The speed up of the pipeline processor for a large number of instructions is-

- 4.5
- 4.0
- 3.33
- 3.0

Solution 2

- **Solution 2:**
- **Cycle Time in Non-Pipelined Processor-**
- Frequency of the clock = 100 MHz
- Cycle time = 1 / frequency = 1 / (100 MHz) = 1 / (100 x 10^6 hertz) = 0.01 μs

- **Non-Pipeline Execution Time-**
- Non-pipeline execution time to process 1 instruction
- = Number of clock cycles taken to execute one instruction = 1 clock cycle = 0.01 μs = 10 ns

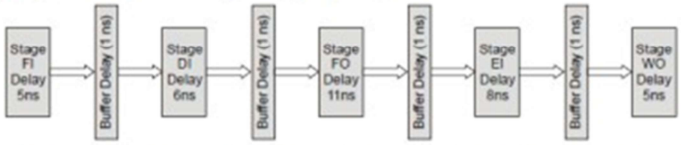
- **Cycle Time in Pipelined Processor-**
- Cycle time = Maximum delay due to any stage + Delay due to its register = Max { 2.5, 1.5, 2, 1.5, 2.5 } + 0.5 ns
- = 2.5 ns + 0.5 ns = 3 ns

- **Pipeline Execution Time-**
- = 1 clock cycle = 3 ns

- **Speed Up-**
- = Non-pipeline execution time / Pipeline execution time
- = 10 ns / 3 ns = 3.33
- Thus, Option (C) is correct

Numerical - 3

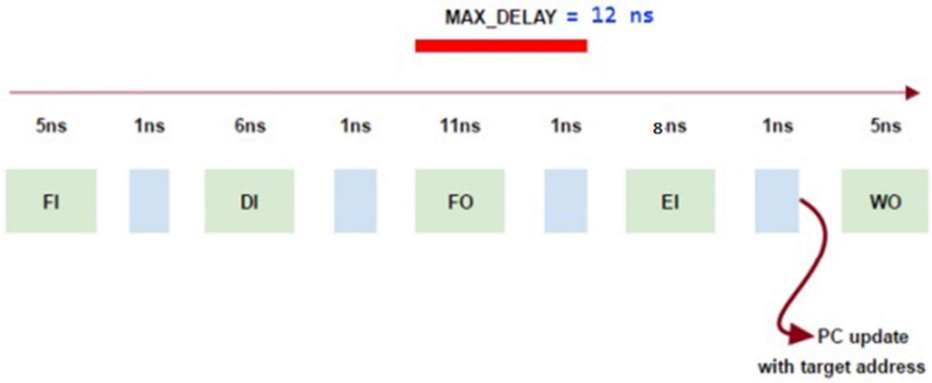
Consider an instruction pipeline with 5 stages [Fetch Instruction (FI), Decode Instruction (DI), Fetch Operand (FO), Execute Instruction (EI) and write operand (WO)]. Delays for the stages and for the pipeline registers are as given below :



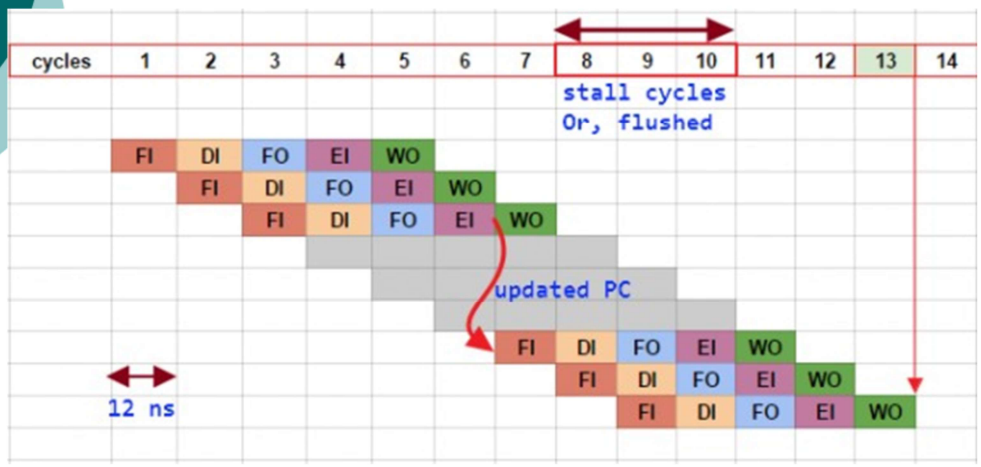
A program consisting of 9 instruction $I_1, I_2, I_3, \ldots I_9$ is executed in this pipelined processor. Instruction I_3 is the only conditional branch instruction and its branch target is I_7. If, the branch is taken taken after EI state, the time (in ns) needed to complete the program is _____.

a. 143ns
b. 150ns
c. 156ns
d. 168ns

Solution - 3


MAX_DELAY = 12 ns
5ns
1ns
6ns
1ns
11ns
1ns
8ns
1ns
5ns
FI
DI
FO
EI
WO
PC update
with target address

Solution -3


cycles
1
2
3
4
5
6
7
8
9
10
11
12
13
14
stall cycles
Or, flushed
FI
DI
FO
EI
WO
updated PC
12 ns

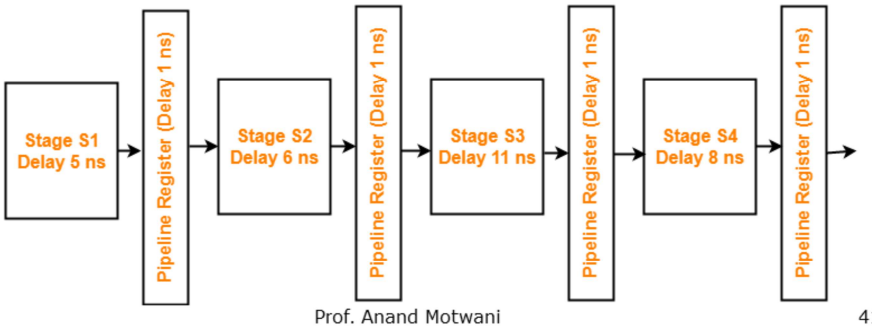
Solution 3:

- we have I1->I2-> I3->I7-> I8->I9 total instructions to execute.
- n= no of instruction=6
- no of stages =k=5
- delay=max cycle time=tp=11+1=12 ns
- branch penalty =4-1=3 cycle.
- total time to execute = time w/o stall +branch penalty overhead
  =(k+n-1)*tp+ 3*tp
  = (5+6-1)*tp+3*tp
  =13 * tp =13*12=**156 ns**

Prof. Anand Motwani 40

Numerical 4

- Consider an instruction pipeline with four stages (S1, S2, S3 and S4) each with combinational circuit only. The pipeline registers are required between each stage and at the end of the last stage. Delays for the stages and for the pipeline registers are as given in the figure-
- What is the approximate speed up of the pipeline in steady state under ideal conditions when compared to the corresponding non-pipeline implementation?
- Ans: 2.5



Solution 4

- Pipeline registers overhead is not counted in normal time execution, So the total count will be
  - 5+6+11+8= 30 [without pipeline]
- Now, for pipeline, each stage will be of 11 n-sec (+ 1 n-sec for overhead).
- In steady state output is produced after every pipeline cycle.
- Here, in this case 11 n-sec. After adding 1n-sec overhead, we will get 12 n-sec of constant output producing cycle.
- dividing 30/12 we get 2.5

Two "Types" of Stalls

- `Noop` instruction (or bubble) inserted between two instructions in the pipeline (e.g., load-use hazards)
  - Keep the instructions *earlier* in the pipeline (later in the code) from progressing down the pipeline for a cycle ("bounce" them in place with write control signals)
  - Insert `noop` instruction by zeroing control bits in the pipeline register at the appropriate stage
  - Let the instructions later in the pipeline (earlier in the code) progress normally down the pipeline
- Flushes (or instruction squashing) where an instruction in the pipeline is replaced with a `noop` instruction (as done for instructions located sequentially after `j` and `beq` instructions)
  - Zero the control bits for the instruction to be flushed