



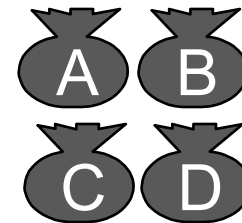
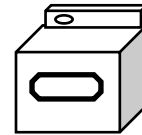
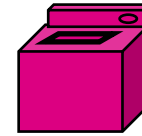
Pipelined Processor Design

Outlines

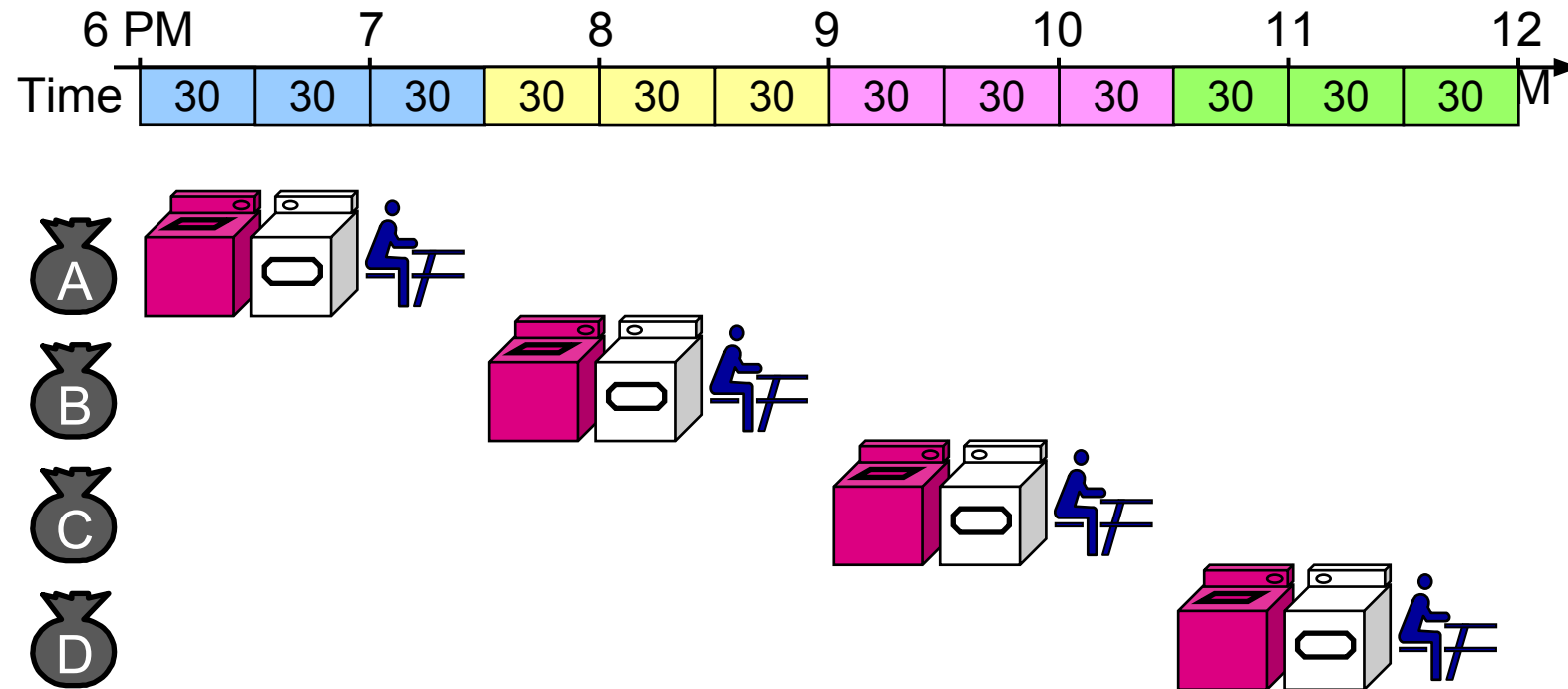
- ▶ **Pipelining versus Serial Execution**
- ▶ Pipelined Data path
- ▶ Pipeline Hazards
- ▶ Data Hazards and Forwarding
- ▶ Pipelined Control
- ▶ Load Delay, Hazard Detection, and Stall Unit
- ▶ Control Hazards
- ▶ Delayed Branch and Dynamic Branch Prediction

Pipelining Example

- ▶ Laundry Example: Three Stages
 1. Wash dirty load of clothes
 2. Dry wet clothes
 3. Fold and put clothes into drawers
- ▶ Each stage takes 30 minutes to complete
- ▶ Four loads of clothes to wash, dry, and fold

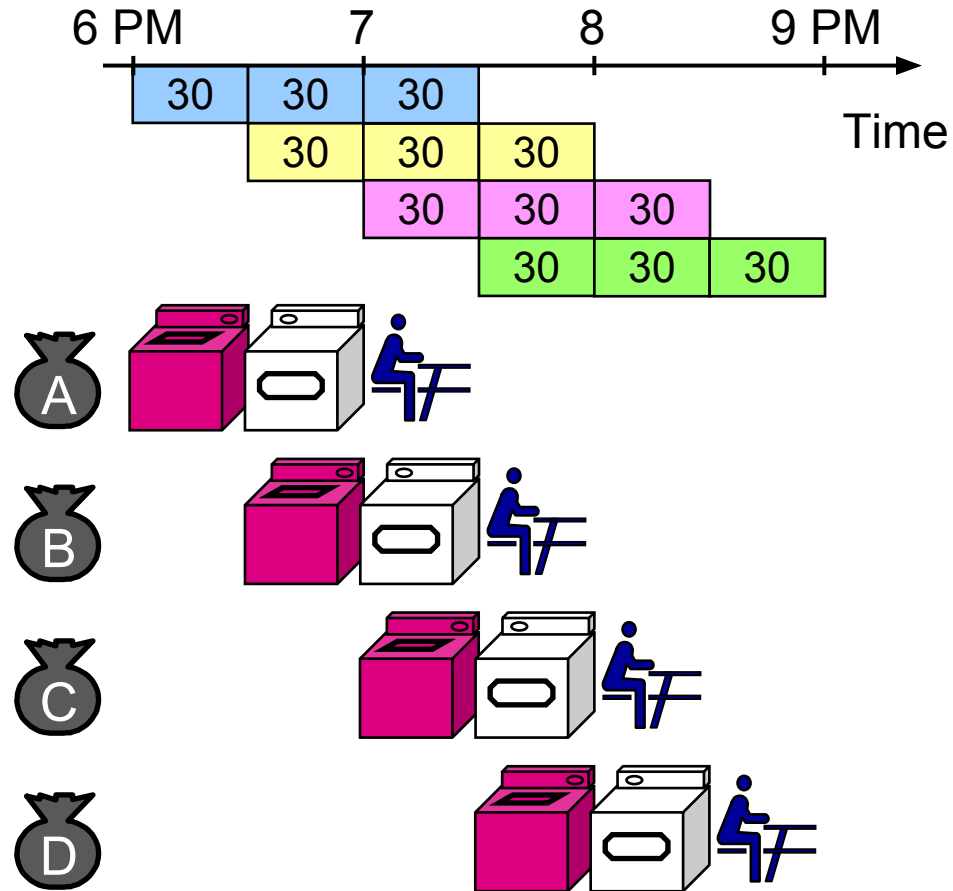


Sequential Laundry



- Sequential laundry takes **6 hours** for **4 loads**
- Intuitively, we can use **pipelining** to speed up laundry

Pipelined Laundry: Start Load ASAP

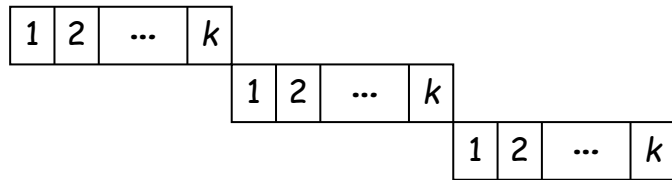


- ▶ Pipelined laundry takes **3 hours** for **4 loads**
- ▶ Speedup factor is **2** for **4 loads**
- ▶ Time to wash, dry, and fold one load is still the same (90 minutes)

Serial Execution versus Pipelining

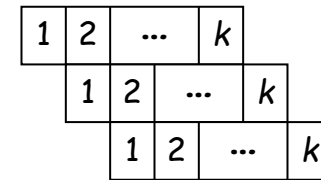
- ▶ Consider a task that can be divided into k subtasks
 - ▶ The k subtasks are executed on k different stages
 - ▶ Each subtask requires one time unit
 - ▶ The total execution time of the task is k time units

- ▶ **Pipelining** is to start a new task before finishing previous
 - ▶ The k stages work in parallel on k different tasks
 - ▶ **Tasks enter/leave pipeline at the rate of one task per time unit.**



Without Pipelining

One completion every k time units

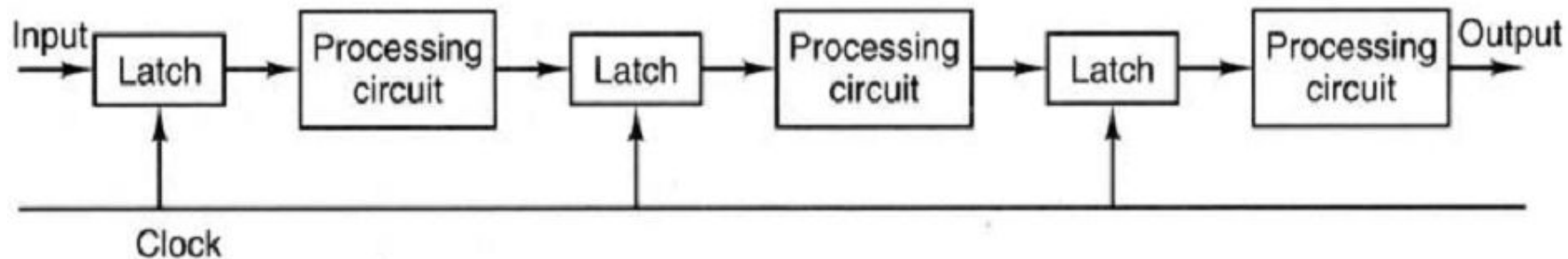


With Pipelining

One completion every 1 time unit

- ▶ Pipelining is one way of improving the overall processing performance of a processor.
 - ▶ This architectural approach allows the simultaneous execution of several instructions
- ▶ The pipeline design technique decomposes a sequential process into several subprocesses, called **stages** or **segments**.
- ▶ A *stage* performs a particular function and produces an intermediate result.
- ▶ It consists of an **input latch**, also called a register or buffer, followed by a **processing circuit**.

- ▶ The processing circuit of a given stage is connected to the input latch of the next stage.
- ▶ A **clock signal** is connected to each input latch.
 - ▶ At each clock pulse, every stage transfers its intermediate result to the input latch of the next stage.



- ▶ The period of the clock pulse should be large enough to provide sufficient time for a signal to traverse through the slowest stage, which is called the *bottleneck*.
- ▶ In addition, there should be enough time for a latch to store its input signals.

Pipeline Performance

- The ability to overlap stages of a sequential process for different input tasks (data or operations) results in an overall theoretical completion time of:

$$T_{pipe} = k \times P + (n - 1) \times P$$

where ***n*** is the number of input tasks, ***k*** is the number of stages in the pipeline, and ***P*** is the clock period.

- ▶ Ideal speedup of a *k*-stage pipeline over serial execution

$$S_k = \frac{\text{Serial Execution in cycles}}{\text{Pipelined Execution in Cycles}} = \frac{n \times k}{k + (n - 1)}$$

- ▶ To be more specific, when n is large, a pipelined processor can produce output approximately k times faster than a nonpipelined processor.

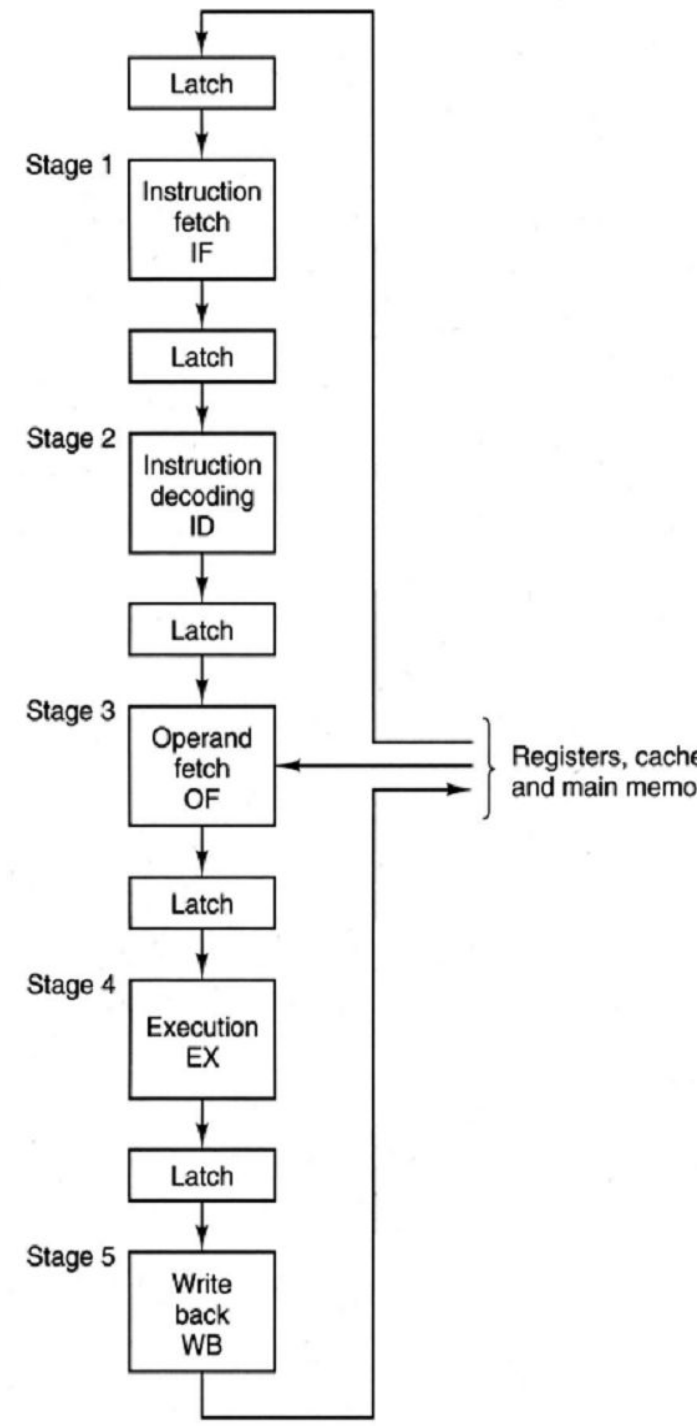
Next . . .

- ▶ Pipelining versus Serial Execution
- ▶ **Pipelined Datapath**
- ▶ Pipeline Hazards
- ▶ Pipelined Control
- ▶ Data Hazards and Forwarding
- ▶ Load Delay, Hazard Detection, and Stall Unit
- ▶ Control Hazards
- ▶ Delayed Branch and Dynamic Branch Prediction

INSTRUCTION PIPELINE

- In a von Neumann architecture, the process of executing an instruction involves several steps.
- 1. **Instruction fetch (IF)**. Retrieval of instructions from cache (or main memory).
- 2. **Instruction decoding (ID)**. Identification of the operation to be performed.
- 3. **Operand fetch (OF)**. Decoding and retrieval of any required operands.
- 4. **Execution (EX)**. Performing the operation on the operands.
- 5. **Write-back (WB)**. Updating the destination operands.

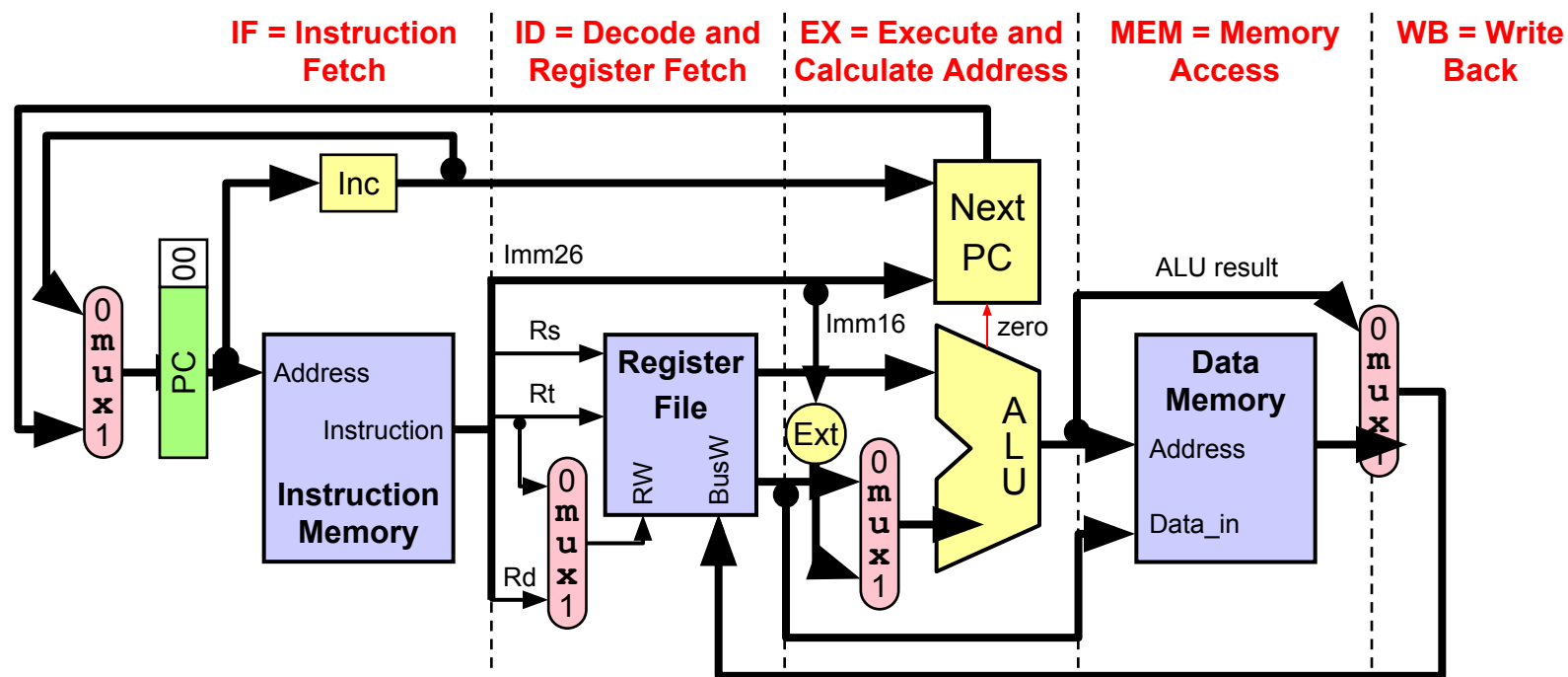
- ✓ An instruction pipeline increases the performance of a processor by overlapping the processing of several different instructions.
- ✓ Instruction pipeline overlaps the process of the preceding stages for different instructions to achieve a much lower total completion time, on average, for a series of instructions.



Single-Cycle Datapath

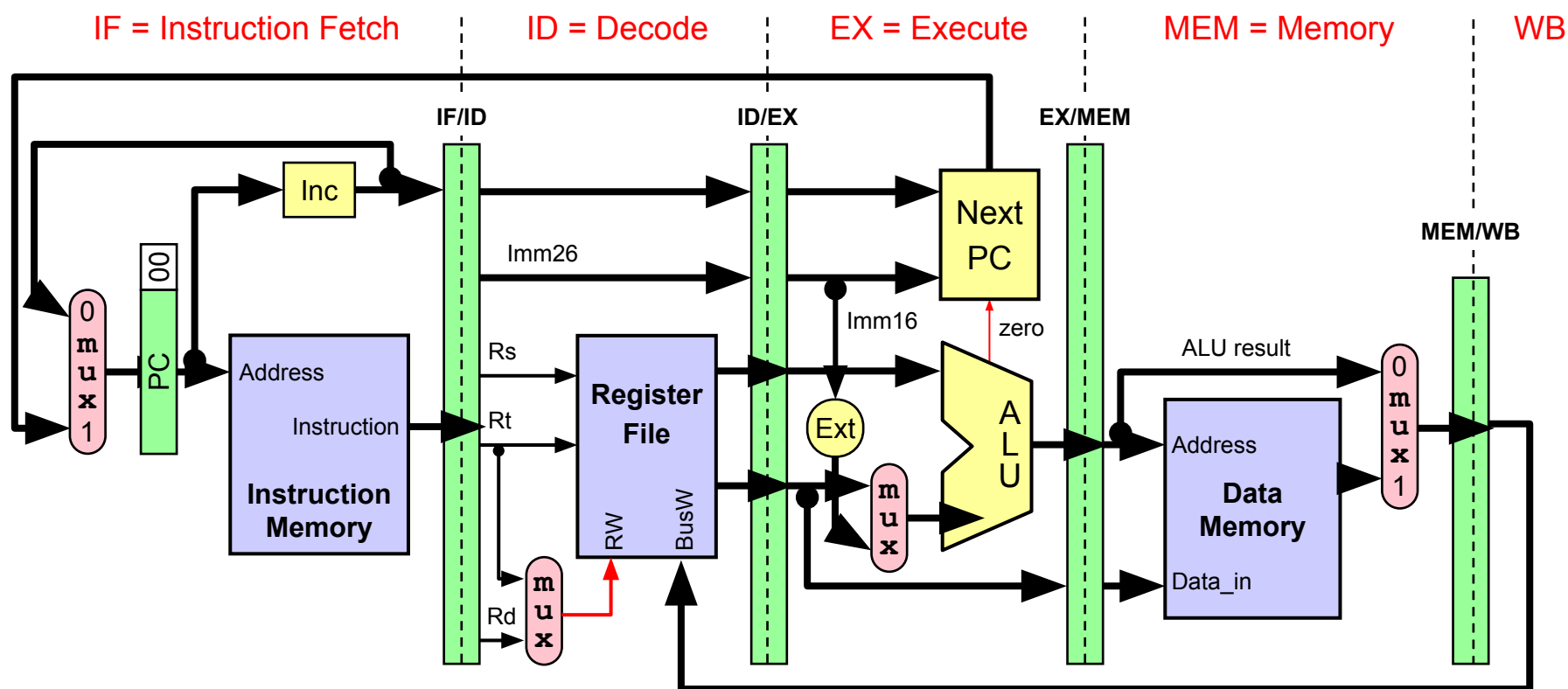
- Shown below is the single-cycle datapath
- How to pipeline this single-cycle datapath?

Answer: Introduce registers at the end of each stage



Pipelined Datapath

- ▶ Pipeline registers, in green, separate each pipeline stage
- ▶ Pipeline registers are labeled by the stages they separate



	Cycles							
	1	2	3	4	5	6	7	8
Instruction i_1	IF	ID	OF	EX	WB			
i_2		IF	ID	OF	EX	WB		
i_3			IF	ID	OF	EX	WB	
i_4				IF	ID	OF	EX	WB

Figure 3.4 Execution cycles of four consecutive instructions in an instruction pipeline.

assuming the clock period to be **10 ns**

$$T_{pipe} = k \times P + (n - 1) \times P$$

$$= 5 \times 10 + (4 - 1) \times 10 = \mathbf{80\ ns}.$$

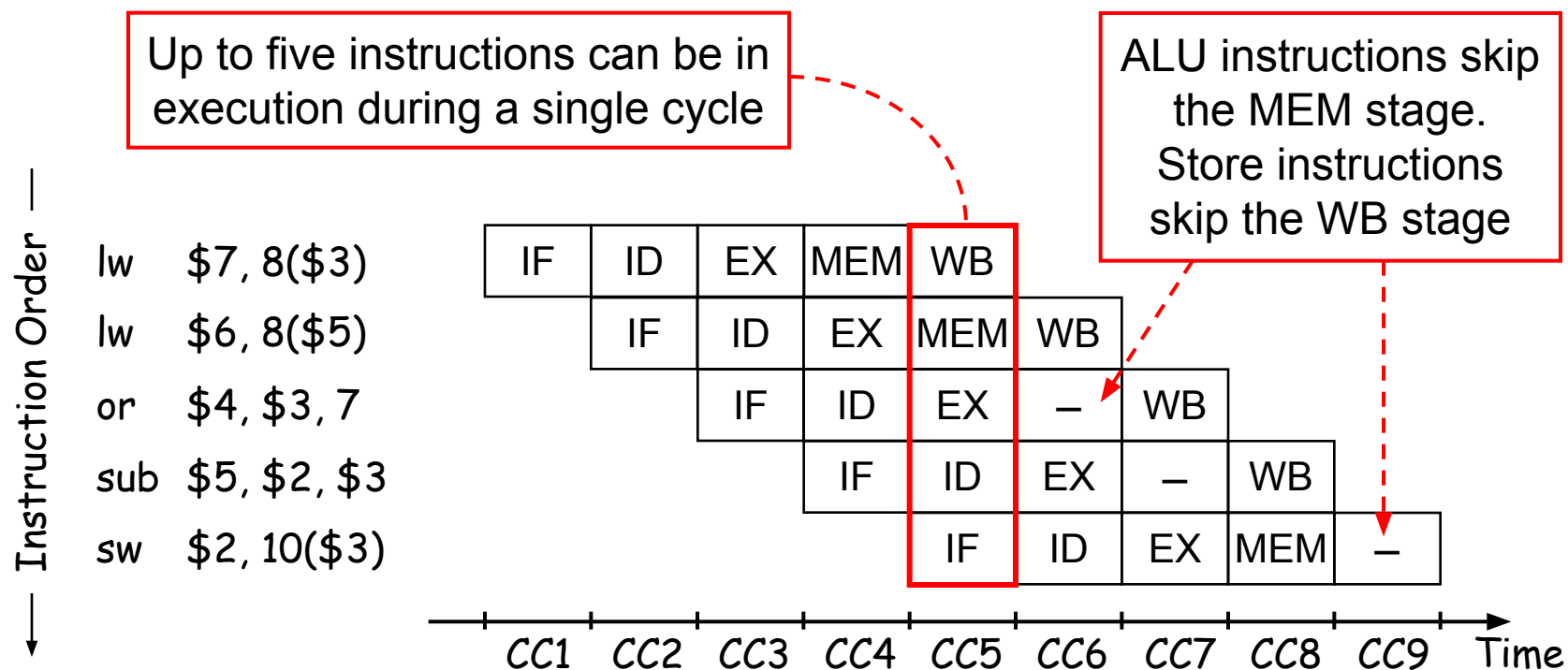
- ➔ Note that in a nonpipelined design the completion time will be much higher:

$$T_{seq} = n \times k \times P$$

$$= 4 \times 5 \times 10 = 200 \text{ ns}$$

Instruction–Time Diagram

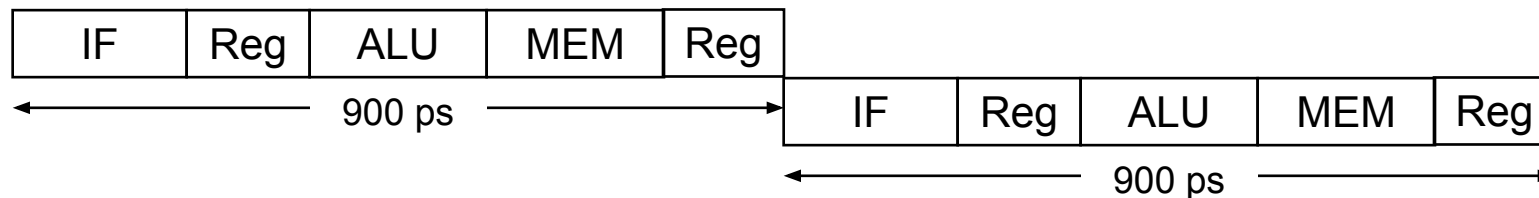
- ▶ Diagram shows:
 - ▶ Which instruction occupies what stage at each clock cycle
- ▶ Instruction execution is pipelined over the 5 stages



Single-Cycle vs Pipelined Performance

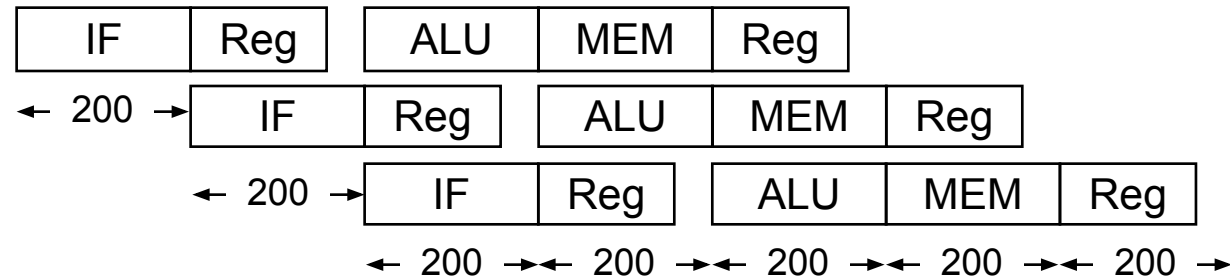
- ▶ Consider a 5-stage instruction execution in which ...
 - ▶ Instruction fetch = ALU operation = Data memory access = 200 ps
 - ▶ Register read = register write = 150 ps
- ▶ What is the single-cycle non-pipelined time?
- ▶ What is the pipelined cycle time?
- ▶ What is the speedup factor for pipelined execution?
- ▶ **Solution**

Non-pipelined cycle = $200+150+200+200+150 = 900 \text{ ps}$



Single-Cycle versus Pipelined – cont'd

- Pipelined cycle time = $\max(200, 150) = 200 \text{ ps}$



- CPI for pipelined execution = 1
 - One instruction completes each cycle (ignoring pipeline fill)
- Speedup of pipelined execution = $900 \text{ ps} / 200 \text{ ps} = 4.5$
 - Instruction count and CPI are equal in both cases
- Speedup factor is **less than 5 (number of pipeline stage)**
 - Because the pipeline stages are **not balanced**

- ▶ The goal of pipelining is to allow multiple instructions execute at the same time.
- ▶ We may need to perform several operations in a cycle
 - ▶ Increment the **PC** and add registers at the same time.
 - ▶ Fetch one instruction while another one reads or writes data.
- ▶ The big benefit from pipelining comes from the fact that once you fed the first element into the pipe (an instruction, for example) the next cycle it's free to accept the next one, long before the previous elements finished the complete datapath.

Next . . .

- ▶ Pipelining versus Serial Execution
- ▶ Pipelined Datapath
- ▶ **Pipeline Hazards**
- ▶ Data Hazards and Forwarding
- ▶ Load Delay, Hazard Detection, and Stall Unit
- ▶ Pipelined Control
- ▶ Control Hazards
- ▶ Delayed Branch and Dynamic Branch Prediction

Pipeline Hazards

- ▶ **Hazards:** situations that would cause incorrect execution
 - ▶ If next instruction were launched during its designated clock cycle
- 1. **Structural hazards**
 - ▶ When a resource is not available, Caused by resource contention
 - ▶ Using same resource by two instructions during the same cycle
- 2. **Data hazards**
 - ▶ An instruction may compute a result needed by next instruction
 - ▶ Hardware can detect dependencies between instructions
- 3. **Control hazards**
 - ▶ A control hazard refers to a situation in which an instruction, such as branch, causes a change in the program flow. (branches/jumps)
 - ▶ Delays in changing the flow of control.
- ▶ Hazards are problems with the instruction pipeline in CPU microarchitectures when the next instruction cannot execute in the following clock cycle

Structural Hazards

► Problem

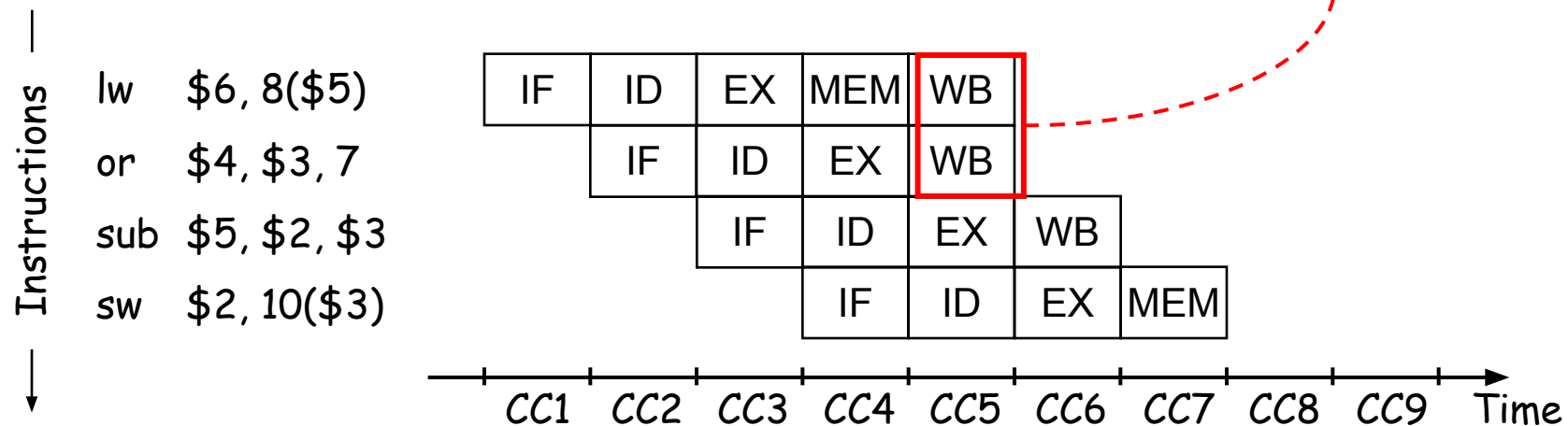
- Attempt to use the same hardware resource by two different instructions during the same cycle

► Example

- Writing back ALU result in stage 4
- Conflict with writing load data in stage 5

Structural Hazard

Two instructions are attempting to write the register file during same cycle



Resolving Structural Hazards

- ▶ **Serious Hazard:**
 - ▶ Hazard cannot be ignored
- ▶ **Solution 1: Delay Access to Resource**
 - ▶ Must have mechanism to delay instruction access to resource
 - ▶ Delay all write backs to the register file to stage 5
 - ▶ ALU instructions bypass stage 4 (memory) without doing anything
- ▶ **Solution 2: Add more hardware resources (more costly)**
 - ▶ Add more hardware to eliminate the structural hazard
 - ▶ Redesign the register file to have two write ports
 - ▶ First write port can be used to write back ALU results in stage 4
 - ▶ Second write port can be used to write back load data in stage 5

Next . . .

- ▶ Pipelining versus Serial Execution
- ▶ Pipelined Datapath
- ▶ Pipeline Hazards
- ▶ **Data Hazards and Forwarding**
- ▶ Load Delay, Hazard Detection, and Stall Unit
- ▶ Control Hazards
- ▶ Delayed Branch and Dynamic Branch Prediction
- ▶ Pipelined Control

Data Hazards

- ▶ In a pipelined processor, instruction executions are overlapped. An instruction may be started and completed before the previous instruction is completed.
- ▶ The **data hazard**, or the **data dependency problem**, comes about as a result of overlapping (or changing the order of) the execution.

i_1	Add	$R_2,$	$R_3,$	R_4	-- $R_2 = R_3 + R_4$
i_2	Add	$R_5,$	$R_2,$	R_1	-- $R_5 = R_2 + R_1$

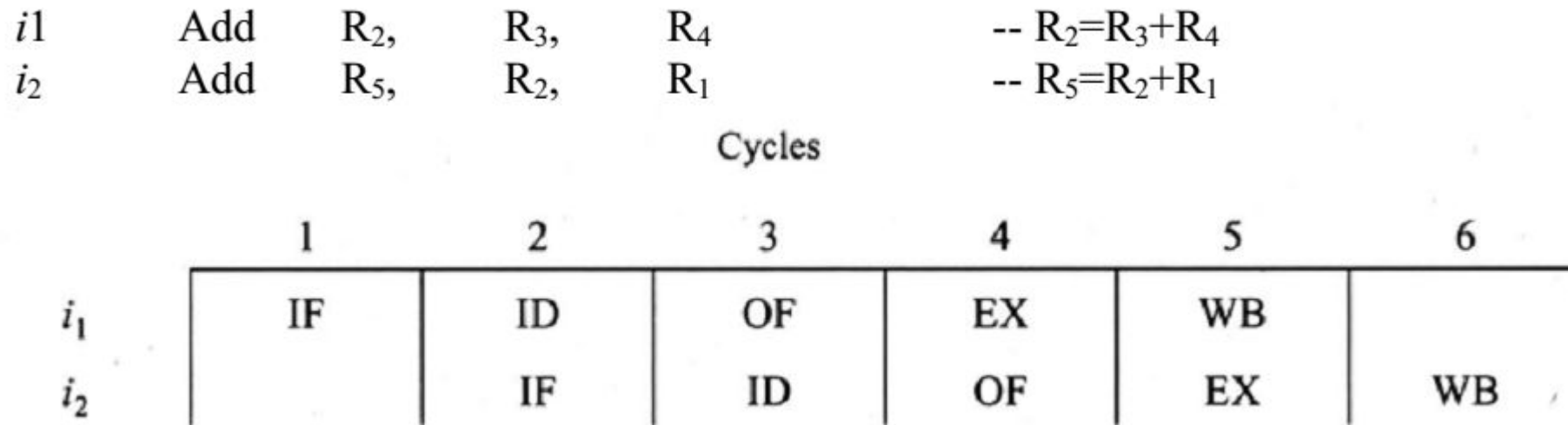


Figure 3.5 Instruction i_2 has data dependency on i_1 .

This would result in using the old contents of R_2 for computing a new value for R_5 , leading to an invalid result.

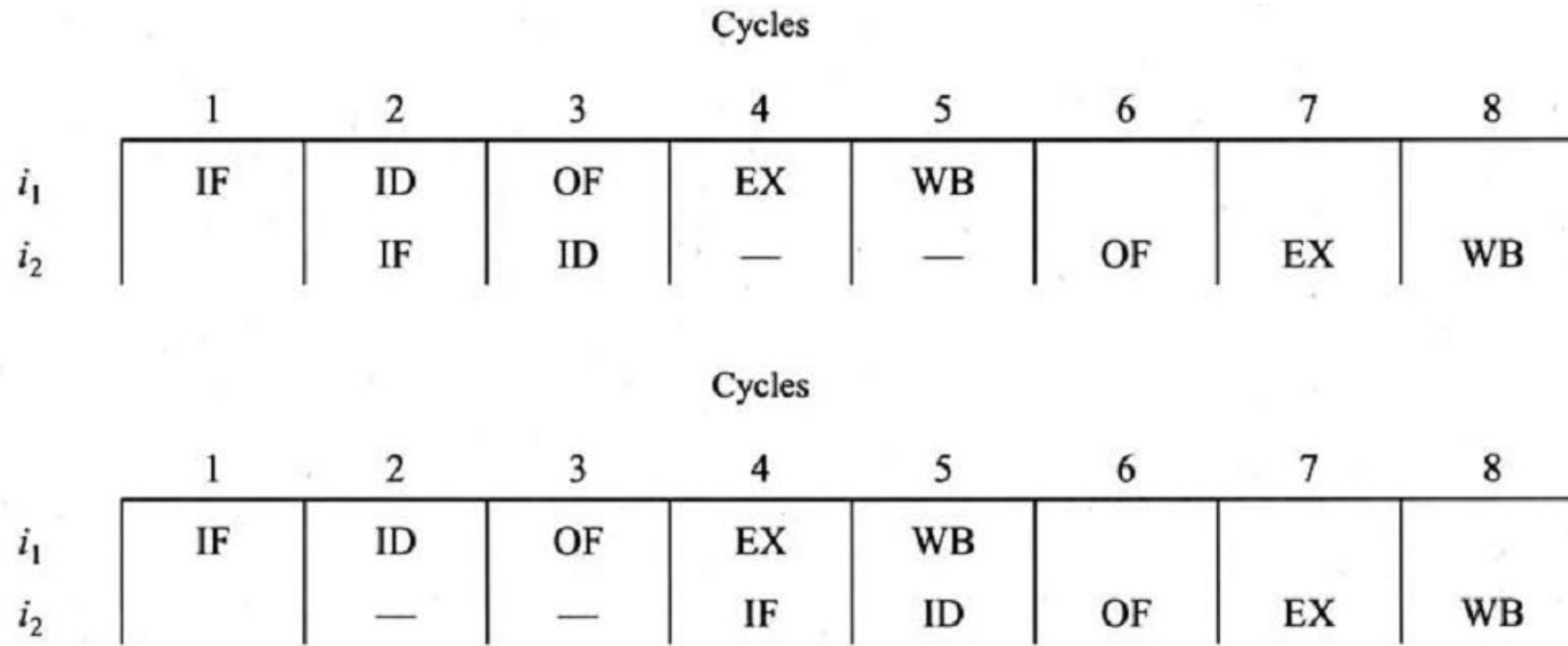


Figure 3.6 Two ways of executing data dependent instructions.

Solution1: Stalling

Instruction i_2 is said to be *stalled* for two clock cycles. No Op(**Bubbles** are inserted)

Often, when an instruction is stalled, the instructions that are positioned after the stalled instruction will also be stalled. However, the instructions before the stalled instruction can continue execution.

To insert a delay, an extra hardware component called a **pipeline interlock** can be added to the pipeline. A *pipeline interlock* detects the dependency and delays the dependent instructions until the conflict is resolved

- ▶ Another way is to let the compiler solve the dependency problem.
- ▶ During compilation, the compiler detects the dependency between data and instructions. It then rearranges these instructions so that the dependency is not hazardous to the system.

Compiler Scheduling

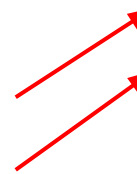
- Consider the following statements:

$a = b + c; d = e - f;$

- Slow code:

```
lw    $10, ($1) # $1 = addr b
lw    $11, ($2) # $2      = addr c
add   $12, $10, $11 # stall
sw    $12, ($3) # $3      = addr a
lw    $13, ($4) # $4      = addr e
lw    $14, ($5) # $5      = addr f
sub   $15, $13, $14 # stall
sw    $15, ($6) # $6      = addr d
```

✓ Fast code: No Stalls



```
lw    $10,    0($1)
lw    $11,    0($2)
lw    $13,    0($4)
lw    $14,    0($5)
add   $12,    $10, $11
sw    $12,    0($3)
sub   $15,    $13,
      $14
sw    $14,    0($6)
```


- ▶ There are three primary types of data hazards: **RAW (read after write)**, **WAR (write after read)**, and **WAW (write after write)**.
- 1. **RAW**: This type of data hazard was discussed previously; it refers to the situation in which *i2* reads a data source before *i1* writes to it.

i_1 :	Add	R_2 ,	R_3 ,	R_4	$--R_2 = R_3 + R_4$
i_2 :	Add	R_5 ,	R_2 ,	R_1	$--R_5 = R_2 + R_1$

2. **WAR:** This refers to the situation in which *i2* writes to a location before *i1* reads it.

<i>i</i> ₁ :	Add	R ₂ ,	R ₃ ,	R ₄	--R ₂ =R ₃ +R ₄
<i>i</i> ₂ :	Add	R ₄ ,	R ₅ ,	R ₆	--R ₄ =R ₅ +R ₆

- ▶ an invalid result may be produced if *i2* writes to **R4** before *i1* reads it; that is, the instruction *i1* might use the wrong value of **R4**.
- ▶ WAR cannot occur in our basic 5-stage pipeline because:
 - ▶ Reads are always in stage 2, and
 - ▶ Writes are always in stage 5
 - ▶ Instructions are processed in order

3. **WAW:** This refers to the situation in which *i2* writes to a location before *i1* writes to it.

<i>i</i> ₁ :	Add	R ₂ ,	R ₃ ,	R ₄	--R ₂ =R ₃ +R ₄
<i>i</i> ₂ :	Add	R ₂ ,	R ₅ ,	R ₆	--R ₂ =R ₅ +R ₆

the value of **R2** is recomputed by *i2*.

- ▶ WAW can't happen in our basic 5-stage pipeline because:
 - ▶ All writes are ordered and always take place in stage 5
- ▶ WAR and WAW hazards can occur in complex pipelines
- ▶ Notice that Read After Read – RAR is NOT a hazard.

- ▶ One way to enhance the architecture of an instruction pipeline is to increase concurrent execution of the instructions by dispatching several independent instructions to different functional units
- ▶ That is, the instructions can be executed out of order, and so their execution may be completed out of order too.

Instruction 0: Register 1 = 6

Instruction 1: Register 1 = 3

Instruction 2: Register 2 = Register 1 + 7 = 10

- Following execution, register 2 should contain the value **10**. However, if Instruction 1 (write **3** to register 1) does not fully exit the pipeline before Instruction 2 starts executing, it means that Register 1 does not contain the value **3** when Instruction 2 performs its addition
- In such an event, Instruction 2 adds 7 to the old value of register 1 (6), and so register 2 contains 13 instead, i.e.:

Instruction 0: Register 1 = 6

Instruction 2: Register 2 = Register 1 + 7 = 13

Instruction 1: Register 1 = 3

Forwarding helps correct such errors by depending on the fact that the output of Instruction 1 (which is **3**) can be used by subsequent instructions *it* is stored in Register 1

Forwarding

- ▶ Forwarding applied to the example means that *there is **no wait to commit/store the output** of Instruction 1 in Register 1 (in this example, the output is 3) before making that output available to the subsequent instruction (in this case, Instruction 2).*
- ▶ The effect is that Instruction 2 uses the correct (the more recent) value of Register 1: the commit/store was made immediately and not pipelined.
- ▶ Forwarding is another solution to data hazard (dependency) with no bubbles/stalls.

- ▶ In today's architectures, the dependencies between instructions are checked statically by the compiler and/or dynamically by the hardware at run time.
- ▶ This preserves the execution order for dependent instructions, which ensures valid results.

Next . . .

- ▶ Pipelining versus Serial Execution
- ▶ Pipelined Datapath
- ▶ Pipeline Hazards
- ▶ Data Hazards and Forwarding
- ▶ Control Hazards
- ▶ Delayed Branch and Dynamic Branch Prediction
- ▶ Load Delay, Hazard Detection, and Stall Unit
- ▶ Pipelined Control

Control Hazards

- ▶ In any set of instructions, there is normally a need for some kind of *branches*.
- ▶ In general, about 30% of all instructions in a program are branches.
 - ▶ Thus branch instructions in the pipeline can reduce the throughput tremendously if not handled properly.
- ▶ Each such branch requires a new address to be loaded into the program counter, which may invalidate all the instructions that are either already in the pipeline or prefetched in the buffer.
- ▶ This **draining** and **refilling** of the pipeline for each branch degrades the throughput of the pipeline to that of a sequential processor
 - ▶ A branch not taken allows the continued sequential flow of uninterrupted instructions to the pipeline.

- ▶ Branch instruction may be one of:
(1) **unconditional branch**, (2) **conditional branch**, and (3) **loop branch**.
- ▶ An **unconditional branch** always alters the sequential program flow.
- ▶ A **conditional branch** sets a new target address in the program counter only when a certain condition is satisfied.
 - ▶ When the condition is satisfied, the path starts from the target address and is called a *target path*. If it is not, the path starts from the next sequential instruction and is called a *sequential path*.
- ▶ A **loop branch** in a loop statement usually jumps back to the beginning of the loop and executes it either a fixed or a variable (data-dependent) number of times.
- ▶ Among all of above, conditional branches are the hardest to handle

e.g.

i_1
 i_2 (conditional branch to i_k)
 i_3
 \cdot
 \cdot
 \cdot
 i_k (target)
 i_{k+1}

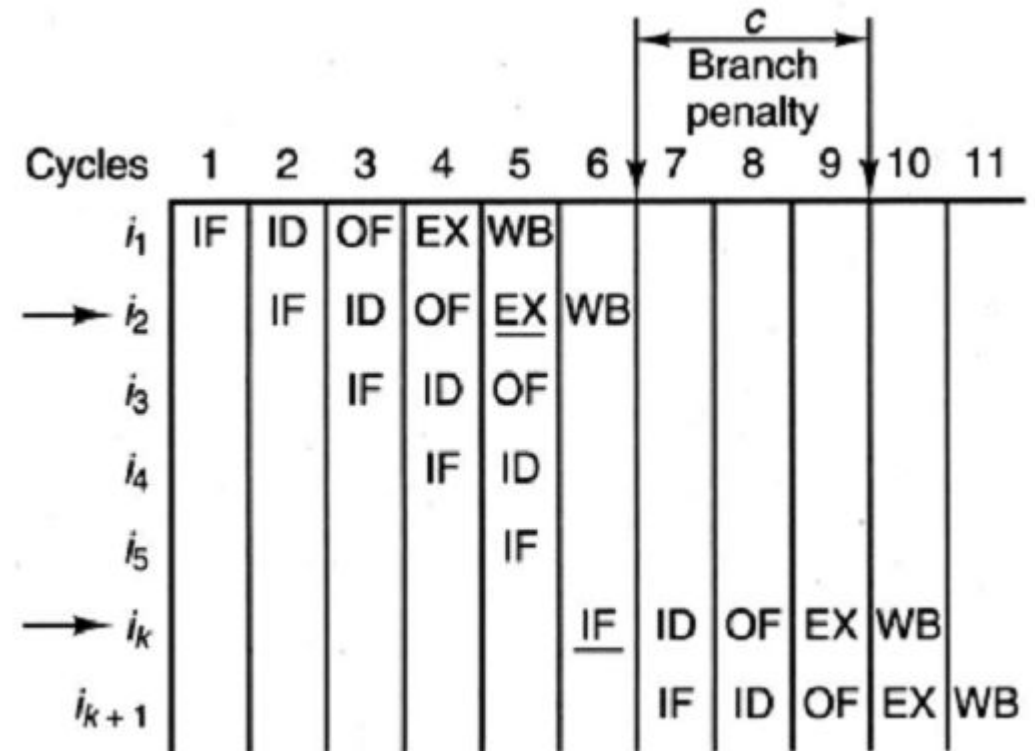


Figure 3.11 Branch Penalty.

- ▶ To reduce the effect of branching some of the better known techniques are **branch prediction**, **delayed branching**, and **multiple prefetching**.

Branch Prediction

- ▶ In this type of design, the outcome of a branch decision is predicted before the branch is actually executed.
- ▶ Based on a particular prediction, the **sequential path** or the **target path** is chosen for execution.
- ▶ Chosen path may often reduces the branch penalty, it may increase the penalty in case of *incorrect prediction*.

Static vs Dynamic Prediction

- ▶ In **static prediction**, a **fixed decision** for prefetching one of the two paths is made before the program runs.
 - ▶ For example, a simple technique would be to *always assume that the branch is taken*.
- ▶ This technique simply loads the program counter with the target address when a branch is encountered
- ▶ Another such technique is to automatically choose one path (sequential or target) for some branch types and another for the rest of the branch types.
- ▶ If the chosen path is wrong, the pipeline is drained and instructions corresponding to the correct path are fetched; the penalty is paid

Static vs Dynamic Prediction

- ▶ In **dynamic prediction**, during the execution of the program the processor makes a decision based on the **past information** of the previously executed branches.
- ▶ For example, recording the history of the last two paths taken by each branch instruction. If the last two executions of a branch instruction have chosen the same path, that path will be chosen for the current execution of the branch instruction. If the two paths do not match, one of the paths will be chosen randomly.

Counter Based Branch Prediction

- A better approach is to associate an ***n*-bit** counter with each branch instruction. This is known as the **counter-based branch prediction** approach.
- ▶ In this method, after executing a branch instruction for the first time, its counter, C , is set to a threshold, T , if the target path was taken, or to $T-1$ if the sequential path was taken.
- ▶ From then on, whenever the branch instruction is about to be executed, if $C \geq T$, then the target path is taken; otherwise, the sequential path is taken.
 - ▶ If the correct path is the target path, the counter is incremented by 1; if not, C is decremented by 1.

Delayed Branching

- ▶ In this type of design, a certain number of instructions after the branch instruction is fetched and executed **regardless** of which path will be chosen for the branch.
- ▶ For example, a processor with a branch delay of k executes a path containing the next k sequential instructions and then either continues on the same path or starts a new path from a new target address.
- ▶ The compiler tries to fill the next k instruction slots after the branch with instructions that are **independent** from the branch instruction. NOP (no operation) instructions/bubbles are placed in any remaining empty slots

e.g. $k=2$

50

i_1 :	Load	R_1 ,	A	
i_2 :	Load	R_2 ,	B	
i_3 :	BrZr	R_2 ,	i_7	-- branch to i_7 if $R_2=0$;
i_4 :	Load	R_3 ,	C	
i_5 :	Add	R_4 ,	R_2 ,	R_3
i_6 :	Mul	R_5 ,	R_1 ,	R_2
i_7 :	Add	R_4 ,	R_1 ,	R_2

-- $R_4 = R_2 + R_3$
-- $R_5 = R_1 * R_2$
-- $R_4 = R_1 + R_2$.



i_2 :	Load	R_2 ,	B	
i_3 :	BrZr	R_2 ,	i_7	
i_1 :	Load	R_1 ,	A	
	NOP			
i_4 :	Load	R_3 ,	C	
i_5 :	Add	R_4 ,	R_2 ,	R_3
i_6 :	Mul	R_5 ,	R_1 ,	R_2
i_7 :	Add	R_4 ,	R_1 ,	R_2 .

Multiple Prefetching

- ▶ In this type of design, the processor fetches **both possible paths**.
- ▶ Once the branch decision is made, the unwanted path is thrown away.
- ▶ By prefetching both possible paths, the fetch penalty is avoided in the case of an incorrect prediction.
- ▶ To fetch both paths, **two buffers** are employed to service the pipeline.

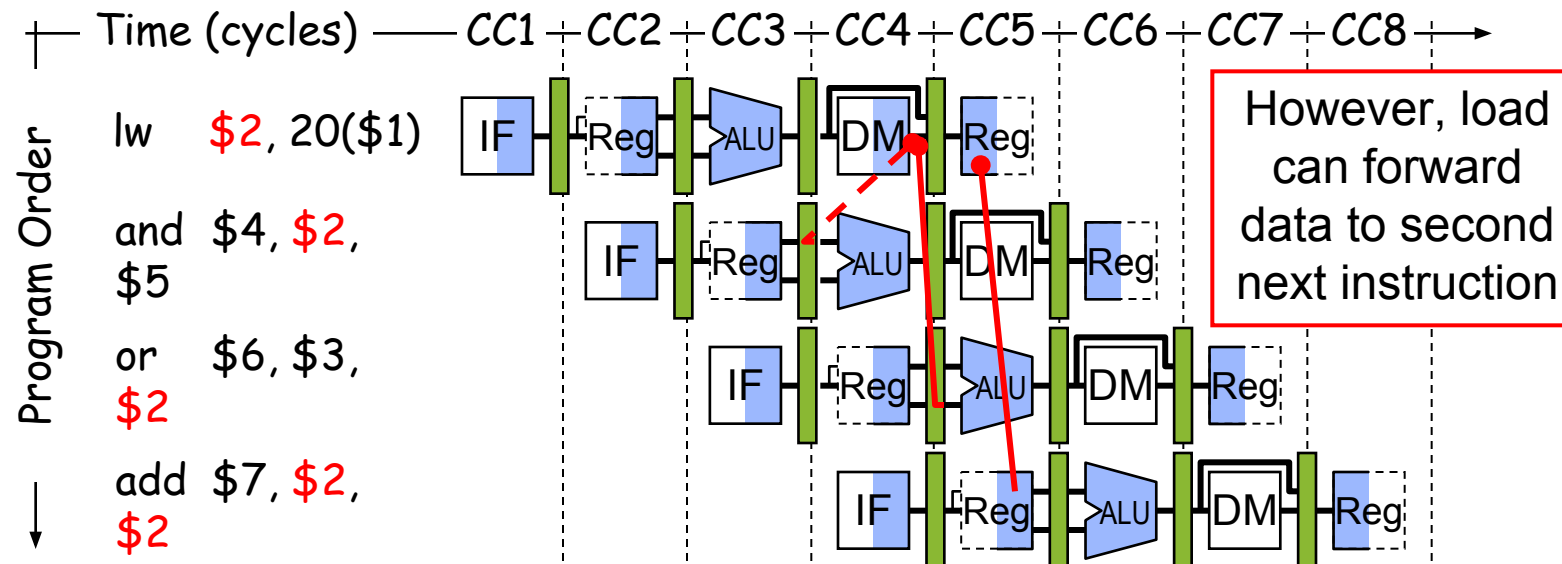
- ▶ In normal execution, the first buffer is loaded with instructions from the next sequential address of the branch instruction.
- ▶ If **a branch occurs**, the contents of the first buffer are invalidated, and the secondary buffer, which has been loaded with instructions from the target address of the branch instruction, is used as the **primary buffer**.
- ▶ This double buffering scheme ensures a constant flow of instructions and data to the pipeline and reduces the time delays caused by the draining and refilling of the pipeline

Next . . .

- ▶ Pipelining versus Serial Execution
- ▶ Pipelined Datapath
- ▶ Pipeline Hazards
- ▶ Data Hazards and Forwarding
- ▶ Control Hazards
- ▶ Delayed Branch and Dynamic Branch Prediction
- ▶ Load Delay, Hazard Detection, and Stall Unit

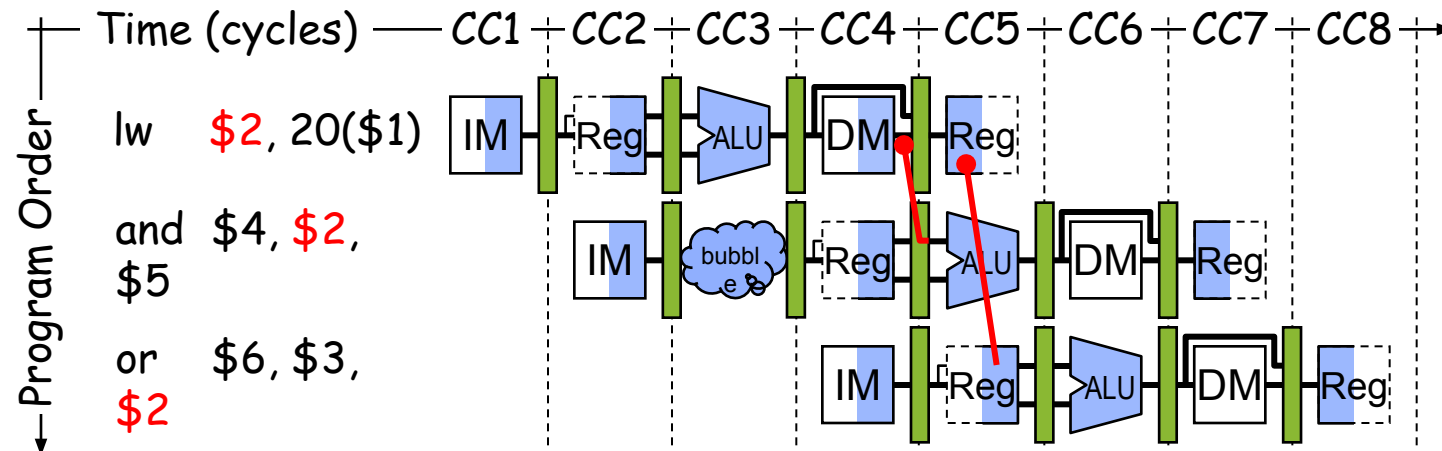
Load Delay

- ▶ Unfortunately, not all data hazards can be forwarded
 - ▶ **Load** has a delay that cannot be eliminated by forwarding
- ▶ In the example shown below ...
 - ▶ The **LW** instruction does not have data until end of CC4
 - ▶ **AND** instruction wants data at beginning of CC4 - **NOT possible**



Stall the Pipeline for one Cycle

- Freeze the **PC** and the **IF/ID** registers
 - No new instruction is fetched and instruction after load is stalled
- Allow the **Load** instruction in **ID/EX** register to proceed
- Introduce a **bubble** into the **ID/EX** register
- Load** can forward data to next instruction after delaying it



Pipeline Control

- ▶ Controlling the sequence of tasks presented to a pipeline for execution is extremely important for maximizing its utilization.
- ▶ If two tasks are initiated requiring the same stage of the pipeline at the same time, a **collision** occurs, which temporarily disrupts execution.

Pipeline Control: Scheduling

- ▶ **Reservation Table:** A pipeline reservation table shows when stages of a pipeline are in use for a particular function.
- ▶ Each **stage** of the pipeline is represented by a **row** in the reservation table.
- ▶ Each row of the reservation table is in turn broken into columns, one per clock cycle.

Pipelining Summary

- ▶ Pipelining doesn't improve **latency** of a single instruction
- ▶ However, it improves **throughput** of entire workload
 - ▶ Instructions are initiated and completed at a higher rate
- ▶ In a **k -stage** pipeline, **k** instructions operate **in parallel**
 - ▶ Overlapped execution using multiple hardware resources
 - ▶ Potential speedup = **number of pipeline stages k** .
 - ▶ Unbalanced lengths of pipeline stages reduces speedup
- ▶ Pipeline rate is limited by **slowest (bottleneck)** pipeline stage.
- ▶ Unbalanced lengths of pipeline stages reduces speedup
- ▶ Also, time to **fill** and **drain** pipeline reduces speedup

Pipelining Summary

- ▶ Three types of pipeline hazards
 - ▶ **Structural hazards**: conflicts using a resource during same cycle
 - ▶ **Data hazards**: due to data dependencies between instructions
 - ▶ **Control hazards**: due to branch and jump instructions
- ▶ Hazards limit the performance and complicate the design
 - ▶ Structural hazards: eliminated by careful design or more hardware
 - ▶ Data hazards are eliminated by forwarding
 - ▶ However, load delay cannot be eliminated and stalls the pipeline
 - ▶ Delayed branching can be a solution when branch delay = 1 cycle
 - ▶ Branch prediction can reduce branch delay to zero
 - ▶ Branch misprediction should drain the wrongly fetched instructions