ADVANCED SYSTEM ARCHITECTURES

Trần Ngọc Thịnh
http://www.cse.hcmut.edu.vn/~tnthinh

Pipelining

What is pipelining?

- Implementation technique in which multiple instructions are overlapped in execution
- Real-life pipelining examples?
  - Laundry
  - Factory production lines
  - Traffic??

Instruction Pipelining (1/2)

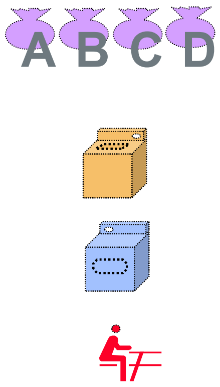
- Instruction pipelining is CPU implementation technique where multiple operations on a number of instructions are overlapped.
- An instruction execution pipeline involves a number of steps, where each step completes a part of an instruction. Each step is called *a pipeline stage* or *a pipeline segment.*
- The stages or steps are connected in a linear fashion: one stage to the next to form the pipeline -- instructions enter at one end and progress through the stages and exit at the other end.
- The time to move an instruction one step down the pipeline is is equal to *the machine cycle* and is determined by the stage with the longest processing delay.

Instruction Pipeling (2/2)

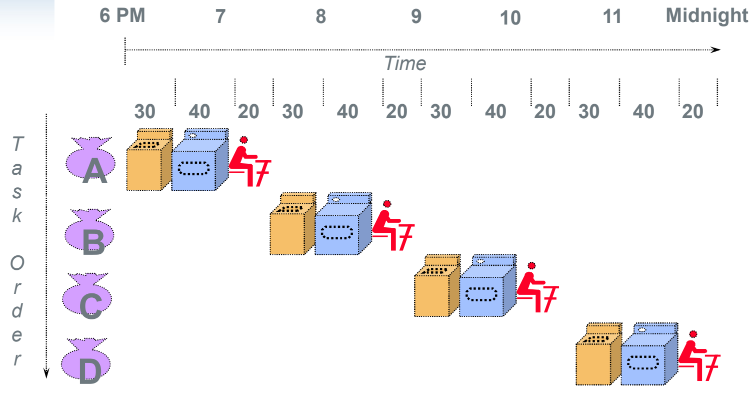
- Pipelining increases the CPU instruction throughput: The number of instructions completed per cycle.
  - Under ideal conditions (no stall cycles), instruction throughput is one instruction per machine cycle, or ideal CPI = 1
- Pipelining does not reduce the execution time of an individual instruction: The time needed to complete all processing steps of an instruction (also called instruction completion latency).
  - Minimum instruction latency = n cycles, where n is the number of pipeline stages

Pipelining Example: Laundry

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- “Folder” takes 20 minutes

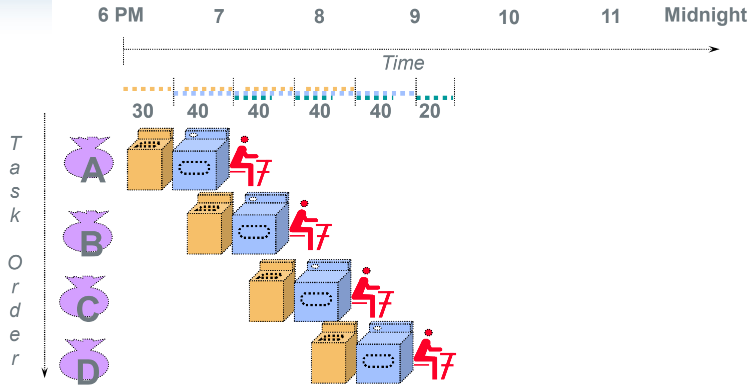


Sequential Laundry

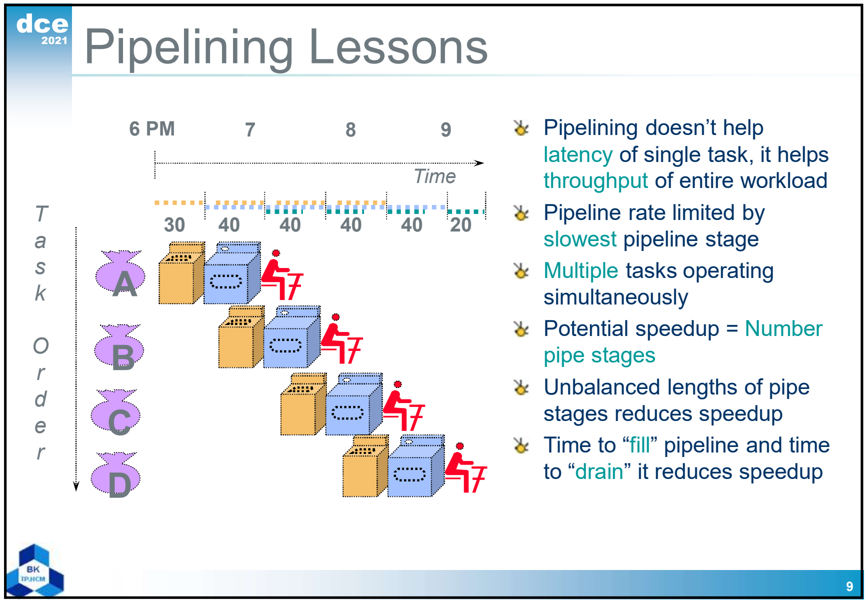


- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

Pipelined Laundry Start work ASAP

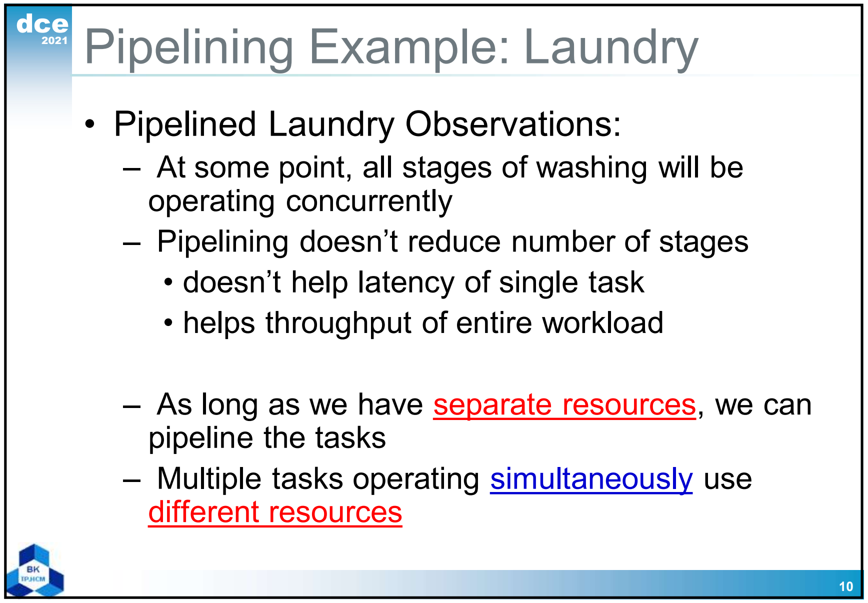


- Pipelined laundry takes 3.5 hours for 4 loads
- Speedup = 6/3.5 = 1.7



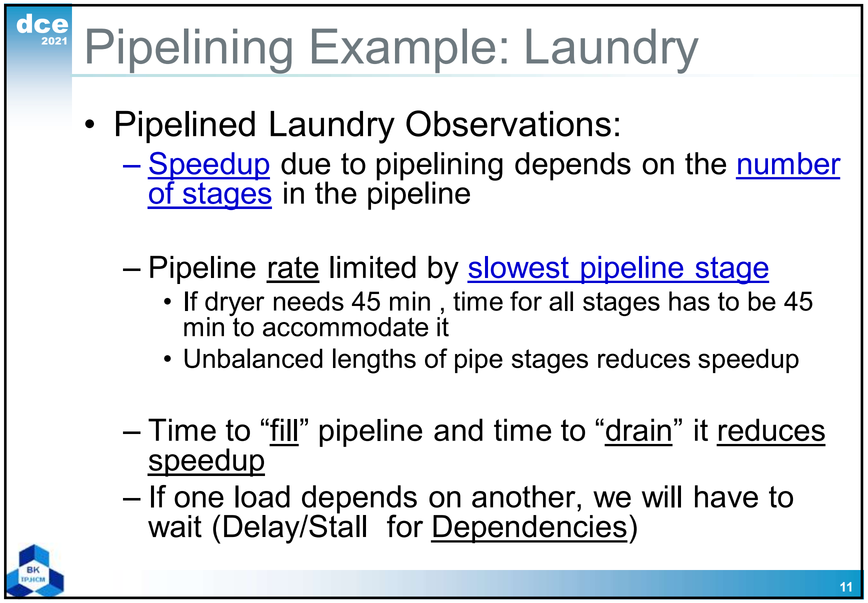
Pipelining Lessons

- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup



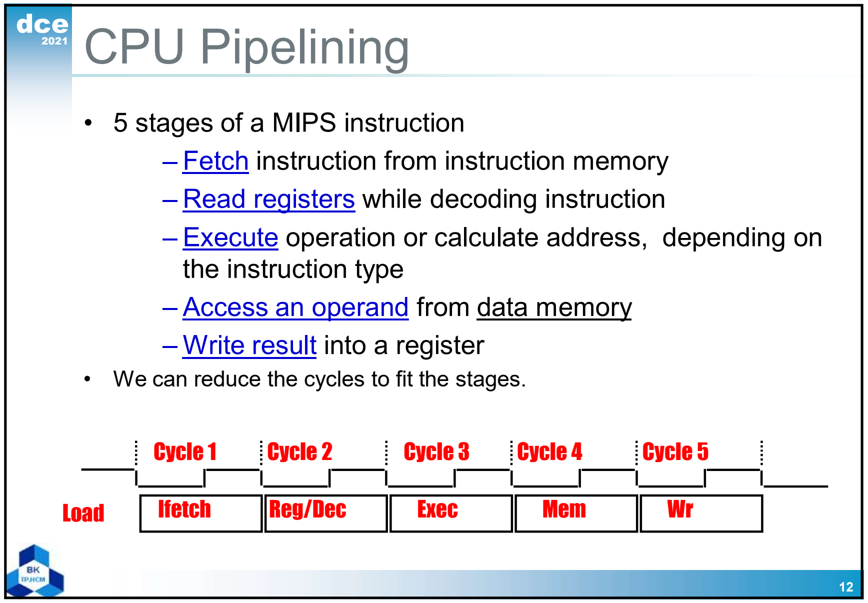
Pipelining Example: Laundry

- Pipelined Laundry Observations:
  - At some point, all stages of washing will be operating concurrently
  - Pipelining doesn't reduce number of stages
    - doesn't help latency of single task
    - helps throughput of entire workload
  - As long as we have separate resources, we can pipeline the tasks
  - Multiple tasks operating simultaneously use different resources



Pipelining Example: Laundry

- Pipelined Laundry Observations:
  - Speedup due to pipelining depends on the number of stages in the pipeline
  - Pipeline rate limited by slowest pipeline stage
    - If dryer needs 45 min , time for all stages has to be 45 min to accommodate it
    - Unbalanced lengths of pipe stages reduces speedup
  - Time to "fill" pipeline and time to "drain" it reduces speedup
  - If one load depends on another, we will have to wait (Delay/Stall for Dependencies)



CPU Pipelining

- 5 stages of a MIPS instruction
  - Fetch instruction from instruction memory
  - Read registers while decoding instruction
  - Execute operation or calculate address, depending on the instruction type
  - Access an operand from data memory
  - Write result into a register
- We can reduce the cycles to fit the stages.

dce 2021

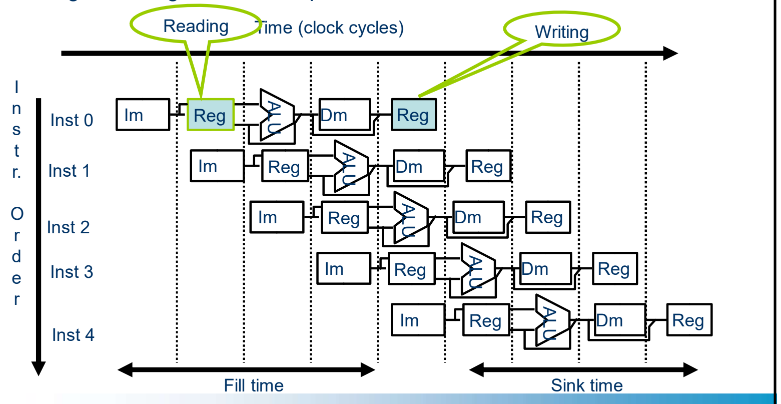
CPU Pipelining

- Example: Resources for Load Instruction
  - Fetch instruction from instruction memory (Ifetch)
    - Instruction memory (IM)
  - Read registers while decoding instruction (Reg/Dec)
    - Register file & decoder (Reg)
  - Execute operation or calculate address, depending on the instruction type (Exec)
    - ALU
  - Access an operand from data memory (Mem)
    - Data memory (DM)
  - Write result into a register (Wr)
    - Register file (Reg)

dce 2021

CPU Pipelining

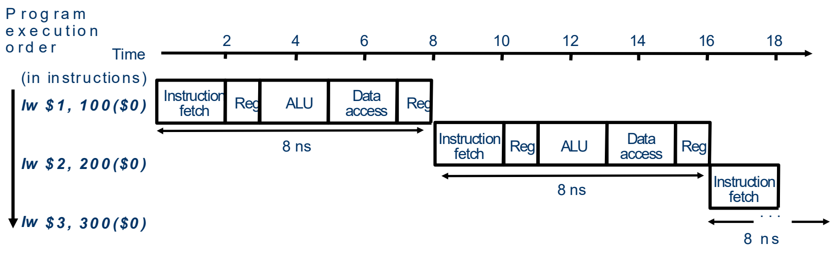
- Note that accessing source & destination registers is performed in two different parts of the cycle
- We need to decide upon which part of the cycle should reading and writing to the register file take place.



dce 2021

CPU Pipelining: Example

- Single-Cycle, non-pipelined execution
  - Total time for 3 instructions: 24 ns



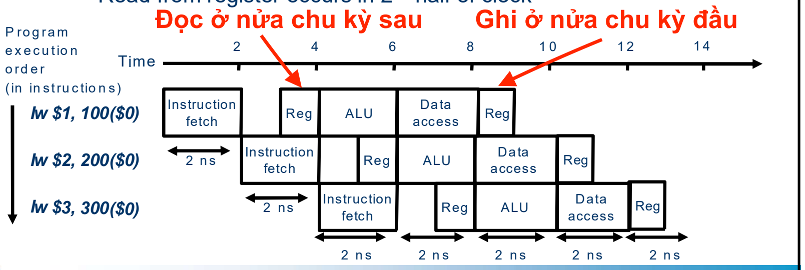
dce 2021

CPU Pipelining: Example

- Single-cycle, pipelined execution
  - Improve performance by increasing instruction throughput
  - Total time for 3 instructions = 14 ns
  - Each instruction adds 2 ns to total execution time
  - Stage time limited by slowest resource (2 ns)
  - Assumptions:
    - Write to register occurs in 1st half of clock
    - Read from register occurs in 2nd half of clock

Đọc ở nửa chu kỳ sau

Ghi ở nửa chu kỳ đầu



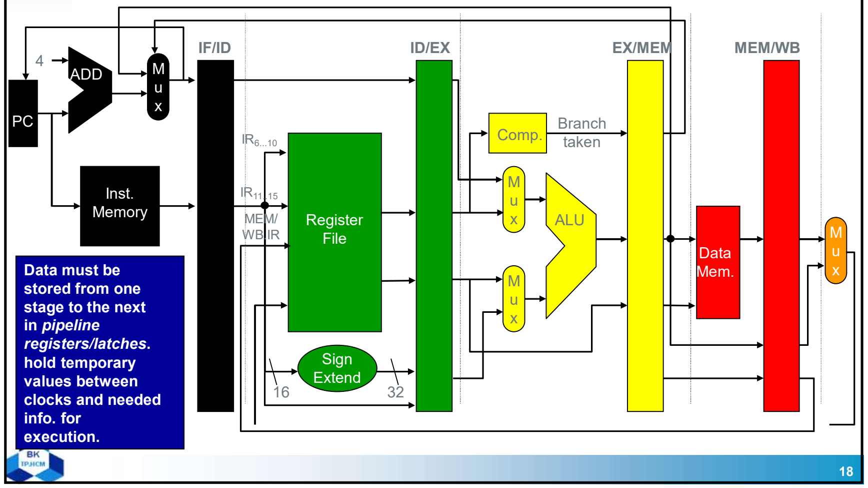
CPU Pipelining: Example

- Assumptions:
  - Only consider the following instructions:

    lw, sw, add, sub, and, or, slt, beq
  - Operation times for instruction classes are:
    - Memory access 2 ns
    - ALU operation 2 ns
    - Register file read or write 1 ns
  - Use a single- cycle (not multi-cycle) model
  - Clock cycle must accommodate the slowest instruction (2 ns)
  - Both pipelined & non-pipelined approaches use the same HW components

InstrClass	IstrFetch	RegRead	ALUOp	DataAccess	RegWrite	TotTime
lw	2 ns	1 ns	2 ns	2 ns	1 ns	8 ns
sw	2 ns	1 ns	2 ns	2 ns		7 ns
add, sub, and, or, slt	2 ns	1 ns	2 ns		1 ns	6 ns
beq	2 ns	1 ns	2 ns			5 ns

MIP dataflow



Data must be stored from one stage to the next in *pipeline registers/latches*. hold temporary values between clocks and needed info. for execution.

CPU Pipelining Example: (1/2)

- Theoretically:
  - Speedup should be equal to number of stages (*n tasks, k stages, p latency*)
  - $$\text{Speedup} = \frac{n*p}{p/k*(n-1) + p} \approx k \text{ (for large n)}$$
- Practically:
  - Stages are imperfectly balanced
  - Pipelining needs overhead
  - Speedup less than number of stages

CPU Pipelining Example: (2/2)

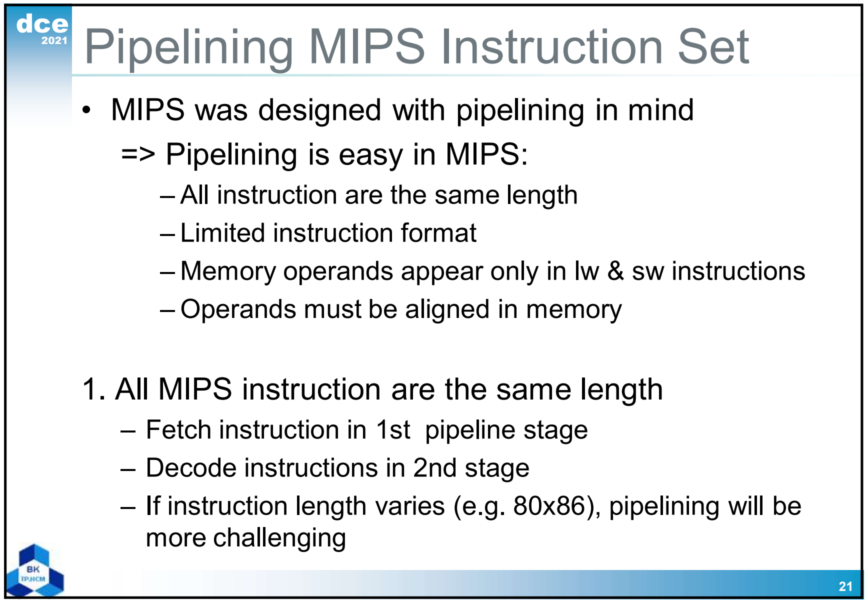
- If we have 3 consecutive instructions
  - Non-pipelined needs 8 x 3 = 24 ns
  - Pipelined needs 14 ns

    => Speedup = 24 / 14 = 1.7
- If we have 1003 consecutive instructions
  - Add more time for 1000 instruction (i.e. 1003 instruction) on the previous example
    - Non-pipelined total time= 1000 x 8 + 24 = 8024 ns
    - Pipelined total time = 1000 x 2 + 14 = 2014 ns

  => Speedup ~ 3.98~ (8 ns / 2 ns]

  ~ near perfect speedup

  => Performance increases for larger number of instructions (throughput)



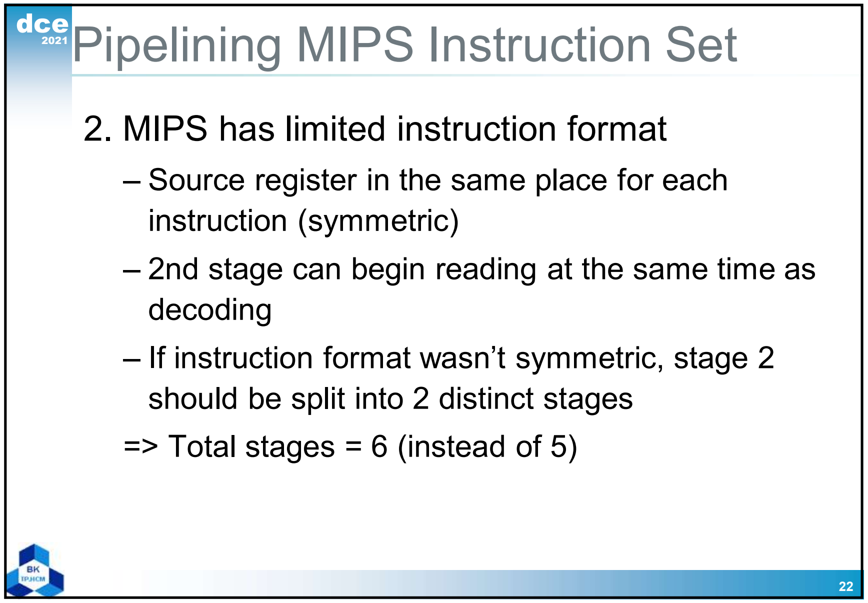
Pipelining MIPS Instruction Set

- MIPS was designed with pipelining in mind

  => Pipelining is easy in MIPS:
  - All instruction are the same length
  - Limited instruction format
  - Memory operands appear only in lw & sw instructions
  - Operands must be aligned in memory

1. All MIPS instruction are the same length
   - Fetch instruction in 1st pipeline stage
   - Decode instructions in 2nd stage
   - If instruction length varies (e.g. 80x86), pipelining will be more challenging

21

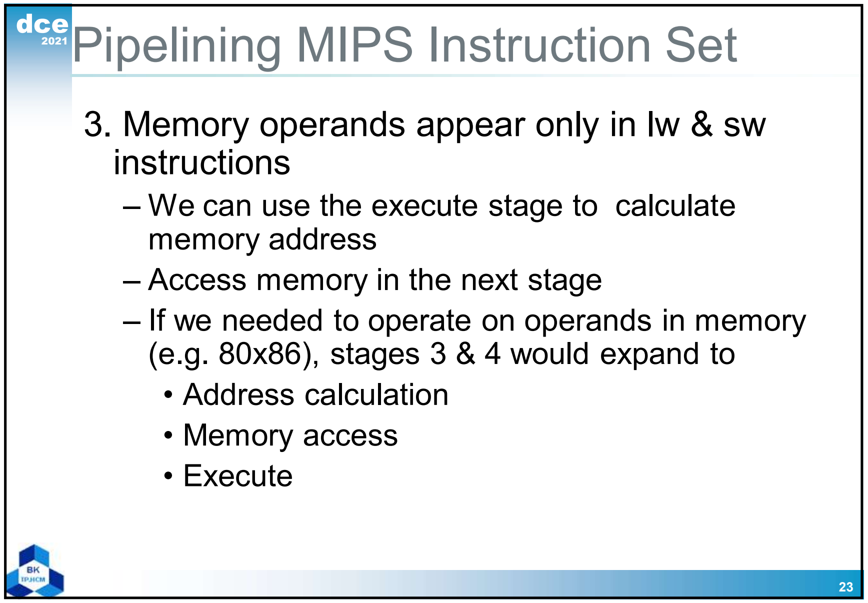


Pipelining MIPS Instruction Set

2. MIPS has limited instruction format
   - Source register in the same place for each instruction (symmetric)
   - 2nd stage can begin reading at the same time as decoding
   - If instruction format wasn't symmetric, stage 2 should be split into 2 distinct stages

   => Total stages = 6 (instead of 5)

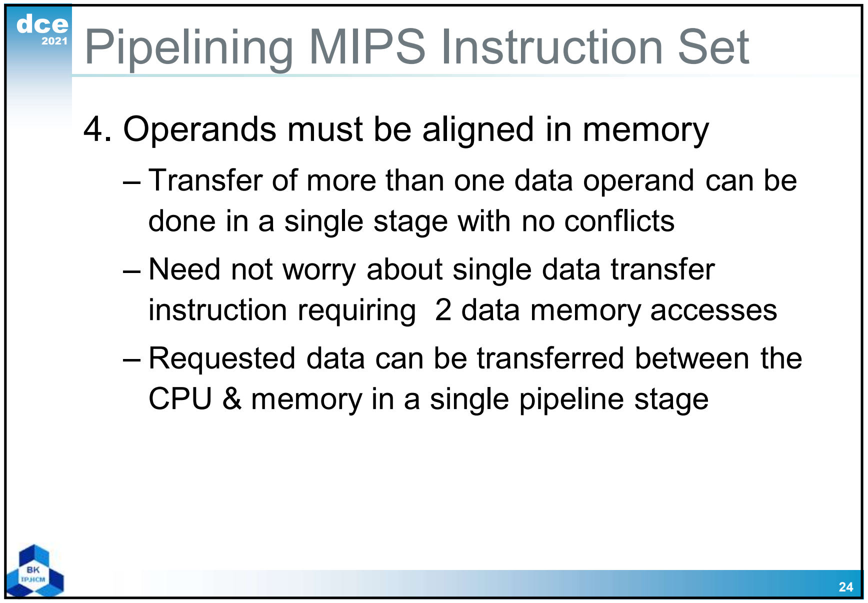
22



Pipelining MIPS Instruction Set

3. Memory operands appear only in lw & sw instructions
   - We can use the execute stage to calculate memory address
   - Access memory in the next stage
   - If we needed to operate on operands in memory (e.g. 80x86), stages 3 & 4 would expand to
     - Address calculation
     - Memory access
     - Execute

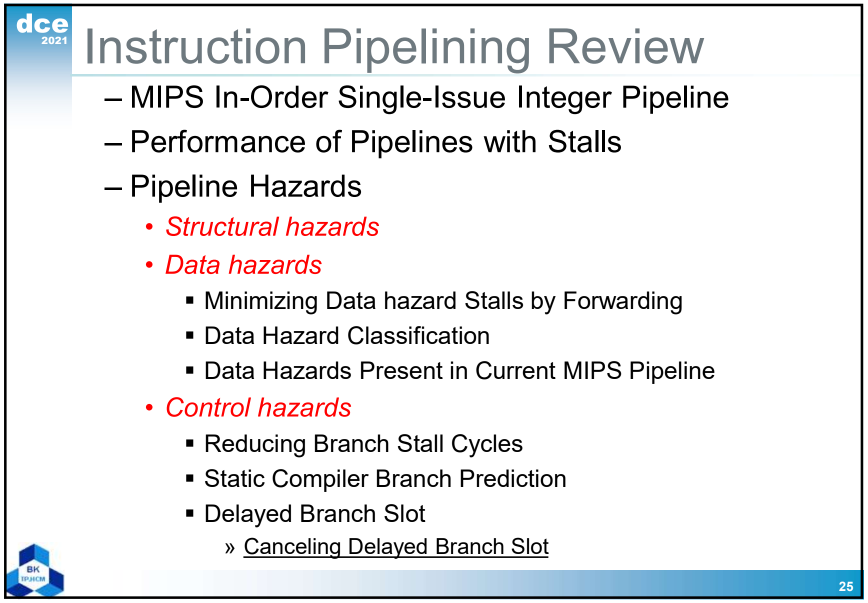
23



Pipelining MIPS Instruction Set

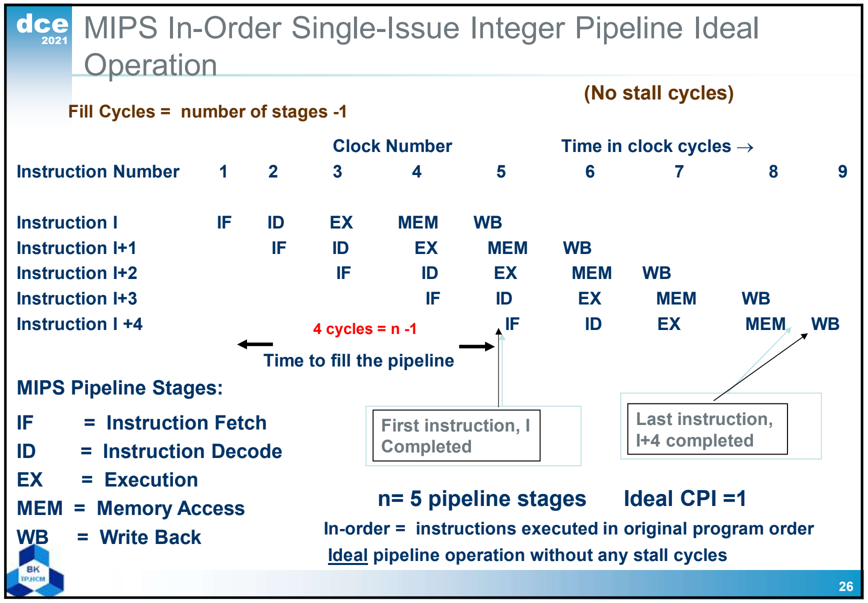
4. Operands must be aligned in memory
   - Transfer of more than one data operand can be done in a single stage with no conflicts
   - Need not worry about single data transfer instruction requiring 2 data memory accesses
   - Requested data can be transferred between the CPU & memory in a single pipeline stage

24



Instruction Pipelining Review

- MIPS In-Order Single-Issue Integer Pipeline
- Performance of Pipelines with Stalls
- Pipeline Hazards
  - *Structural hazards*
  - *Data hazards*
    - Minimizing Data hazard Stalls by Forwarding
    - Data Hazard Classification
    - Data Hazards Present in Current MIPS Pipeline
  - *Control hazards*
    - Reducing Branch Stall Cycles
    - Static Compiler Branch Prediction
    - Delayed Branch Slot
      - » Canceling Delayed Branch Slot



MIPS In-Order Single-Issue Integer Pipeline Ideal Operation

(No stall cycles)

Fill Cycles = number of stages -1

Clock Number Time in clock cycles →

Instruction Number	1	2	3	4	5	6	7	8	9
Instruction I	IF	ID	EX	MEM	WB				
Instruction I+1		IF	ID	EX	MEM	WB			
Instruction I+2			IF	ID	EX	MEM	WB		
Instruction I+3				IF	ID	EX	MEM	WB	
Instruction I +4					IF	ID	EX	MEM	WB

4 cycles = n -1

Time to fill the pipeline

First instruction, I Completed

Last instruction, I+4 completed

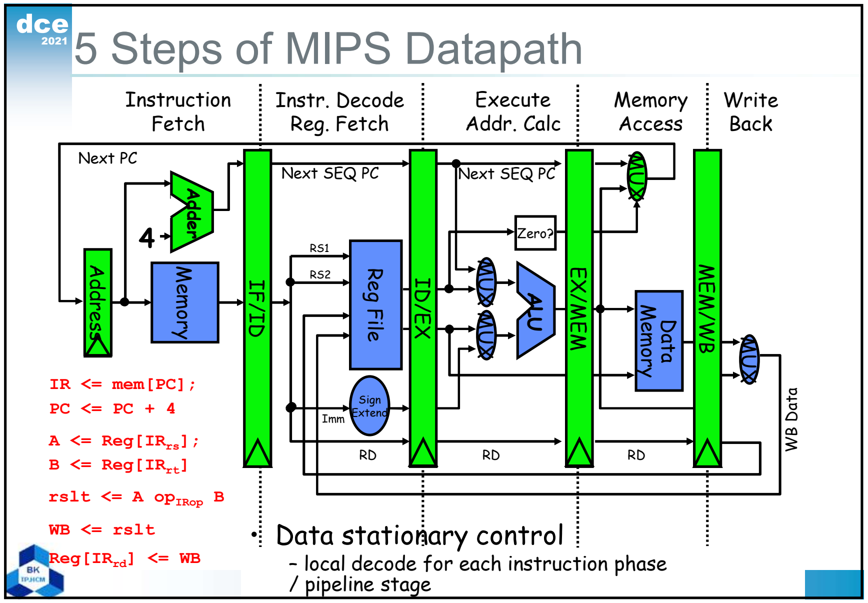
MIPS Pipeline Stages:

IF = Instruction Fetch
ID = Instruction Decode
EX = Execution
MEM = Memory Access
WB = Write Back

n= 5 pipeline stages Ideal CPI =1

In-order = instructions executed in original program order

Ideal pipeline operation without any stall cycles



5 Steps of MIPS Datapath

IR <= mem[PC];
PC <= PC + 4

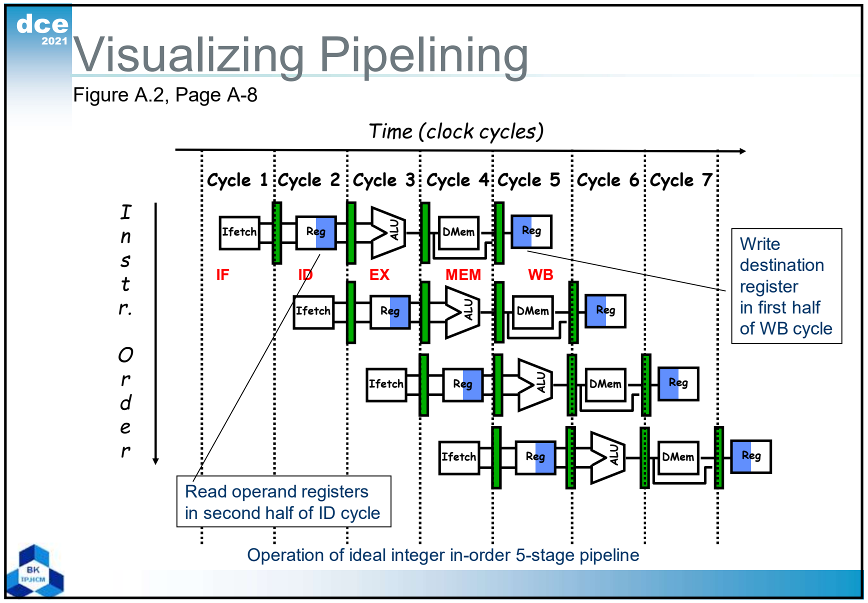
A <= Reg[IR_{rs}];
B <= Reg[IR_{rt}]

rslt <= A op_{IRop} B

WB <= rslt

Reg[IR_{rd}] <= WB

- Data stationary control
  - local decode for each instruction phase / pipeline stage



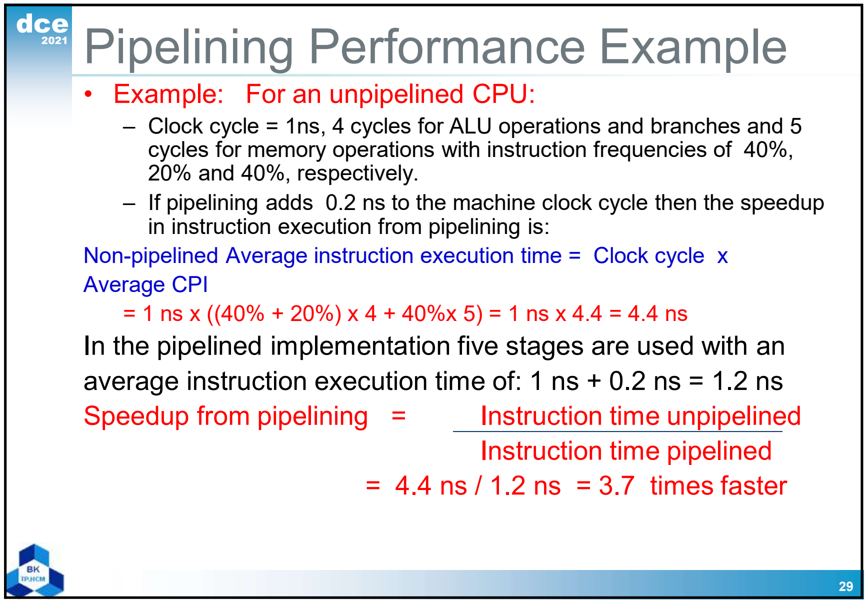
Visualizing Pipelining

Figure A.2, Page A-8

Write destination register in first half of WB cycle

Read operand registers in second half of ID cycle

Operation of ideal integer in-order 5-stage pipeline



Pipelining Performance Example

- Example: For an unpipelined CPU:
  - Clock cycle = 1ns, 4 cycles for ALU operations and branches and 5 cycles for memory operations with instruction frequencies of 40%, 20% and 40%, respectively.
  - If pipelining adds 0.2 ns to the machine clock cycle then the speedup in instruction execution from pipelining is:

Non-pipelined Average instruction execution time = Clock cycle x Average CPI

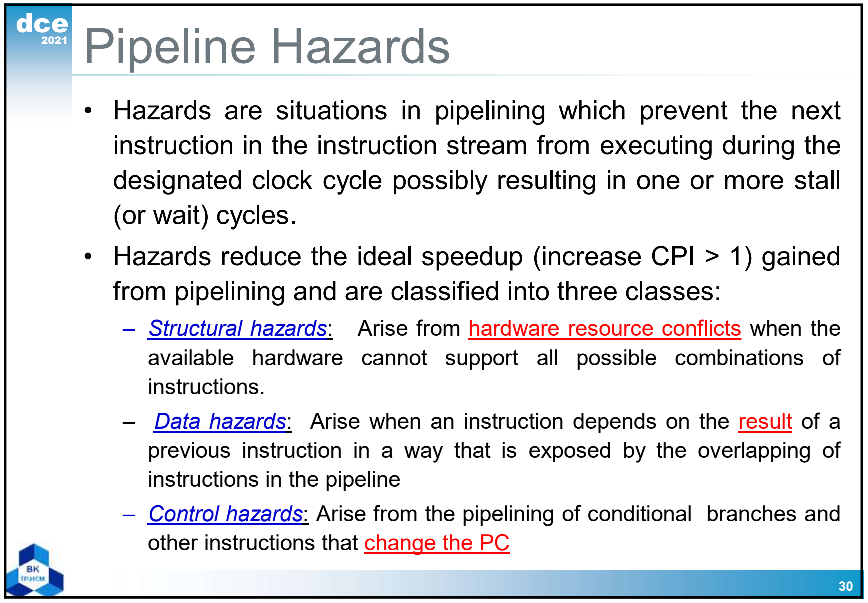
= 1 ns x ((40% + 20%) x 4 + 40%x 5) = 1 ns x 4.4 = 4.4 ns

In the pipelined implementation five stages are used with an average instruction execution time of: 1 ns + 0.2 ns = 1.2 ns

$$\text{Speedup from pipelining} = \frac{\text{Instruction time unpipelined}}{\text{Instruction time pipelined}}$$

= 4.4 ns / 1.2 ns = 3.7 times faster

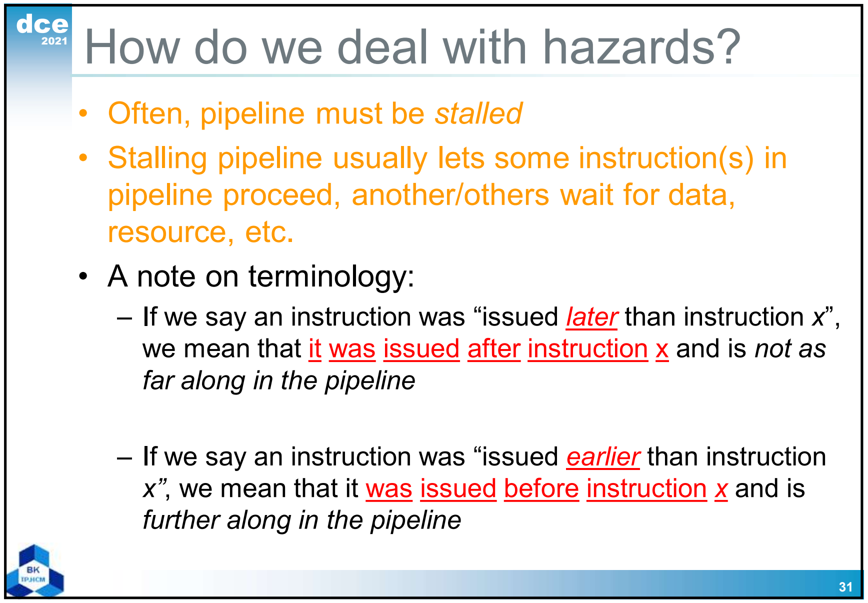
29



Pipeline Hazards

- Hazards are situations in pipelining which prevent the next instruction in the instruction stream from executing during the designated clock cycle possibly resulting in one or more stall (or wait) cycles.
- Hazards reduce the ideal speedup (increase CPI > 1) gained from pipelining and are classified into three classes:
  - *Structural hazards*: Arise from hardware resource conflicts when the available hardware cannot support all possible combinations of instructions.
  - *Data hazards*: Arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline
  - *Control hazards*: Arise from the pipelining of conditional branches and other instructions that change the PC

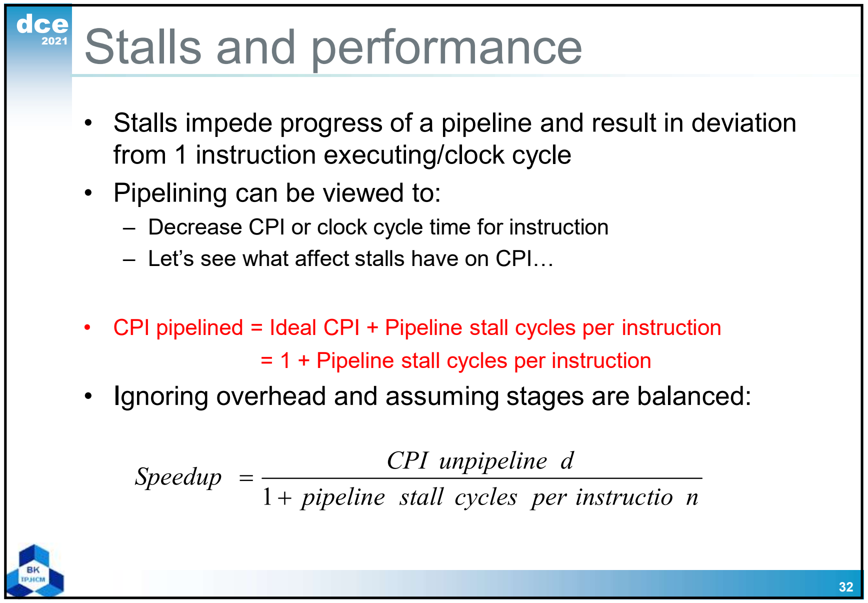
30



How do we deal with hazards?

- Often, pipeline must be *stalled*
- Stalling pipeline usually lets some instruction(s) in pipeline proceed, another/others wait for data, resource, etc.
- A note on terminology:
  - If we say an instruction was "issued *later* than instruction *x*", we mean that it was issued after instruction x and is *not as far along in the pipeline*
  - If we say an instruction was "issued *earlier* than instruction *x*", we mean that it was issued before instruction *x* and is *further along in the pipeline*

31



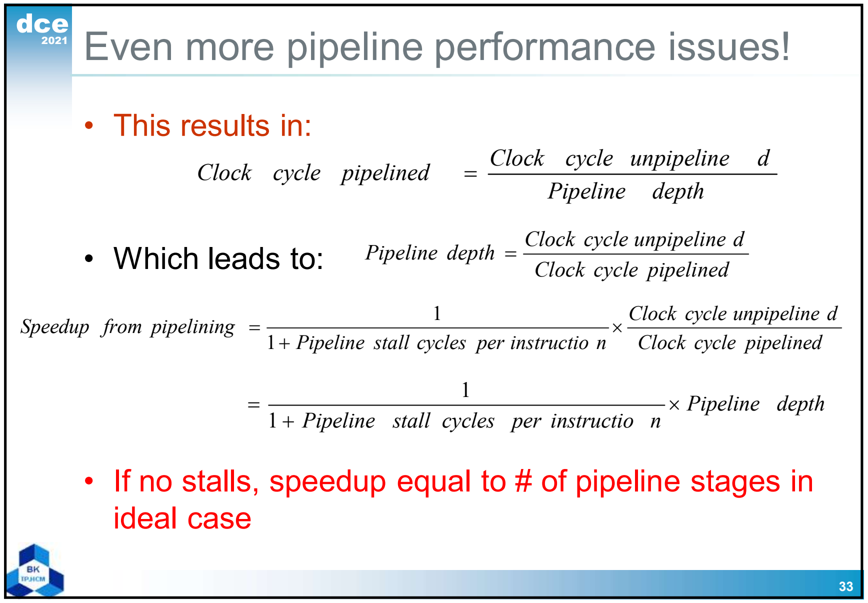
Stalls and performance

- Stalls impede progress of a pipeline and result in deviation from 1 instruction executing/clock cycle
- Pipelining can be viewed to:
  - Decrease CPI or clock cycle time for instruction
  - Let's see what affect stalls have on CPI…
- CPI pipelined = Ideal CPI + Pipeline stall cycles per instruction

  = 1 + Pipeline stall cycles per instruction
- Ignoring overhead and assuming stages are balanced:

$$Speedup = \frac{CPI\ unpipelined}{1 + pipeline\ \ stall\ \ cycles\ \ per\ instruction}$$

32



Even more pipeline performance issues!

- This results in:

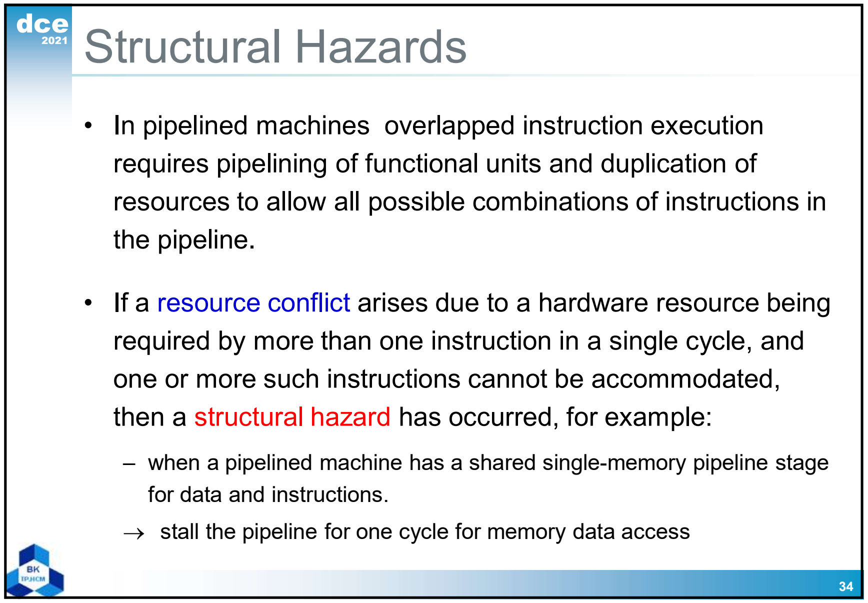
$$Clock\ \ cycle\ \ pipelined = \frac{Clock\ \ cycle\ \ unpipeline\ \ d}{Pipeline\ \ depth}$$

- Which leads to: $Pipeline\ depth = \frac{Clock\ cycle\ unpipeline\ d}{Clock\ cycle\ pipelined}$

$$Speedup\ \ from\ \ pipelining = \frac{1}{1 + Pipeline\ \ stall\ cycles\ \ per\ instructio\ n} \times \frac{Clock\ cycle\ unpipeline\ d}{Clock\ cycle\ pipelined}$$

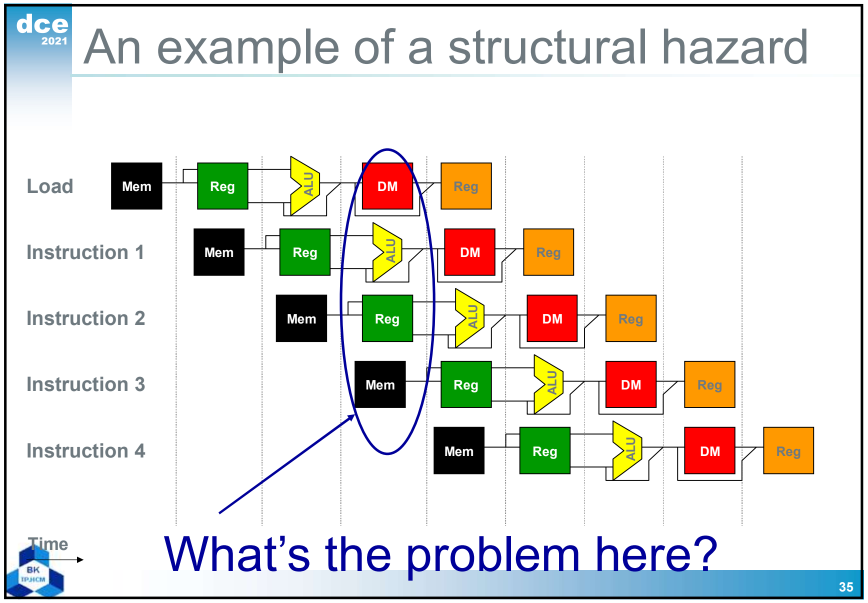
$$= \frac{1}{1 + Pipeline\ \ \ stall\ \ cycles\ \ \ per\ \ instructio\ \ \ n} \times Pipeline\ \ \ depth$$

- If no stalls, speedup equal to # of pipeline stages in ideal case



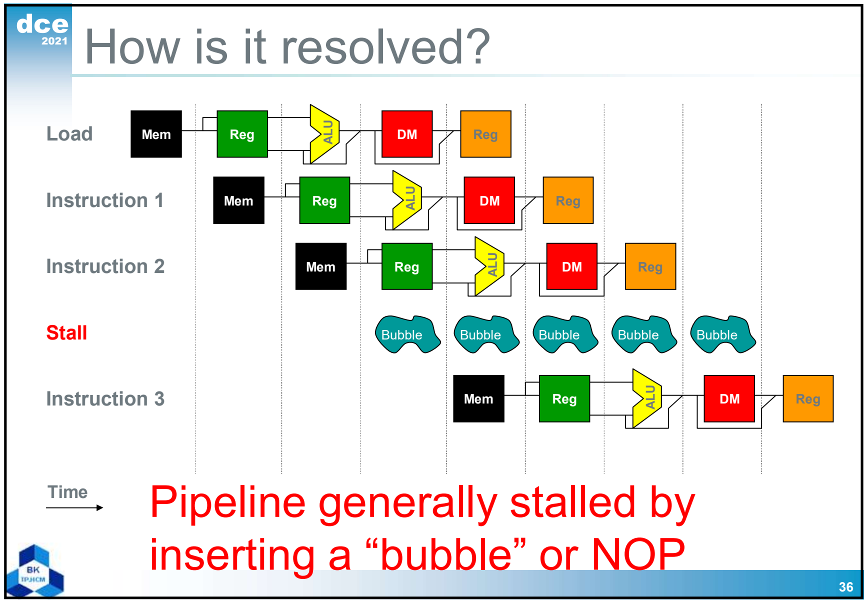
Structural Hazards

- In pipelined machines overlapped instruction execution requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline.
- If a resource conflict arises due to a hardware resource being required by more than one instruction in a single cycle, and one or more such instructions cannot be accommodated, then a structural hazard has occurred, for example:
  - when a pipelined machine has a shared single-memory pipeline stage for data and instructions.
  - → stall the pipeline for one cycle for memory data access



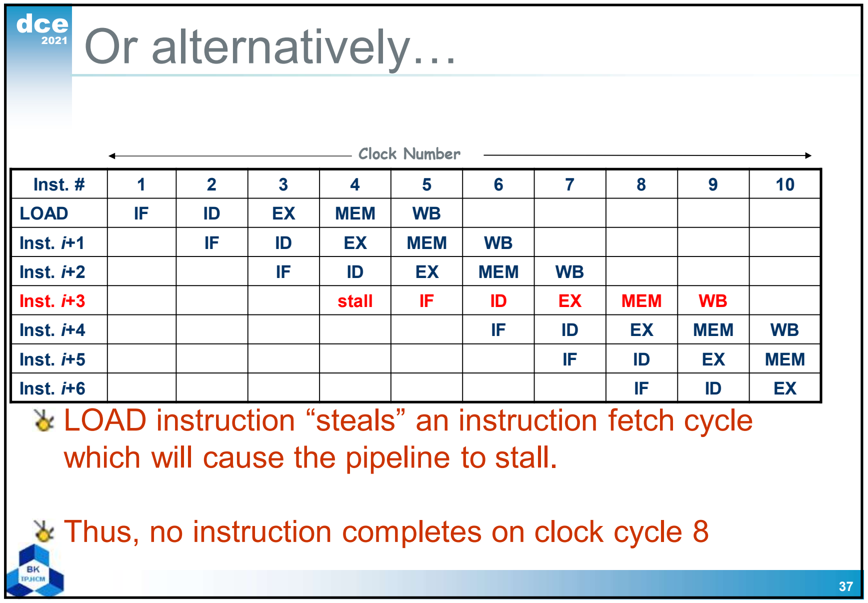
An example of a structural hazard

What’s the problem here?



How is it resolved?

Pipeline generally stalled by inserting a “bubble” or NOP



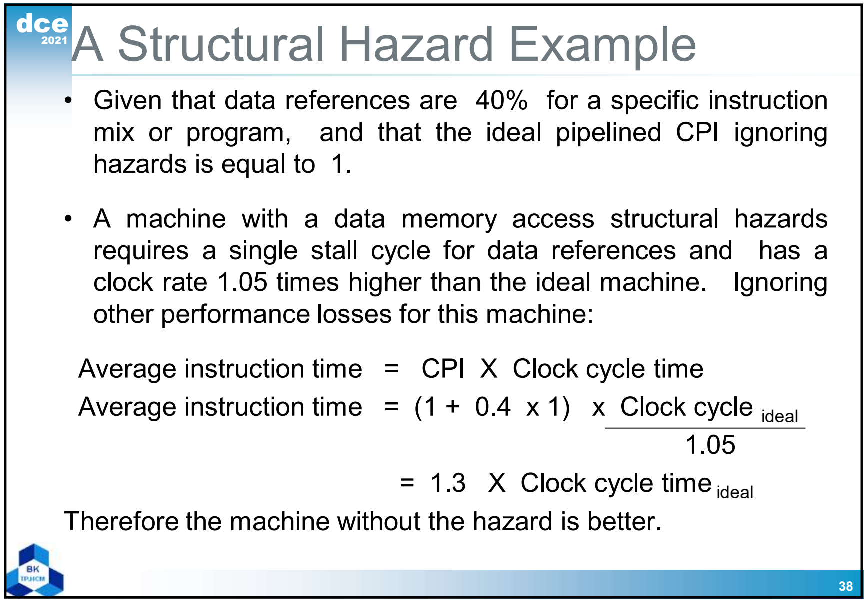
Or alternatively…

Clock Number

Inst. #	1	2	3	4	5	6	7	8	9	10
LOAD	IF	ID	EX	MEM	WB					
Inst. *i*+1		IF	ID	EX	MEM	WB				
Inst. *i*+2			IF	ID	EX	MEM	WB			
Inst. *i*+3				stall	IF	ID	EX	MEM	WB	
Inst. *i*+4						IF	ID	EX	MEM	WB
Inst. *i*+5							IF	ID	EX	MEM
Inst. *i*+6								IF	ID	EX

- LOAD instruction “steals” an instruction fetch cycle which will cause the pipeline to stall.

- Thus, no instruction completes on clock cycle 8



A Structural Hazard Example

- Given that data references are 40% for a specific instruction mix or program, and that the ideal pipelined CPI ignoring hazards is equal to 1.

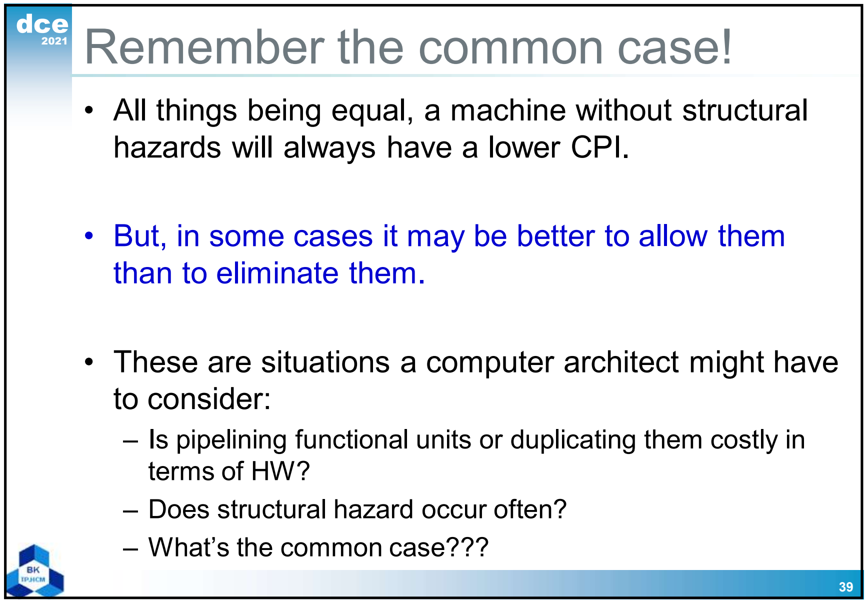
- A machine with a data memory access structural hazards requires a single stall cycle for data references and has a clock rate 1.05 times higher than the ideal machine. Ignoring other performance losses for this machine:

Average instruction time = CPI X Clock cycle time

$$\text{Average instruction time} = (1 + 0.4 \times 1) \times \frac{\text{Clock cycle}_{\text{ideal}}}{1.05}$$

$$= 1.3 \text{ X Clock cycle time}_{\text{ideal}}$$

Therefore the machine without the hazard is better.

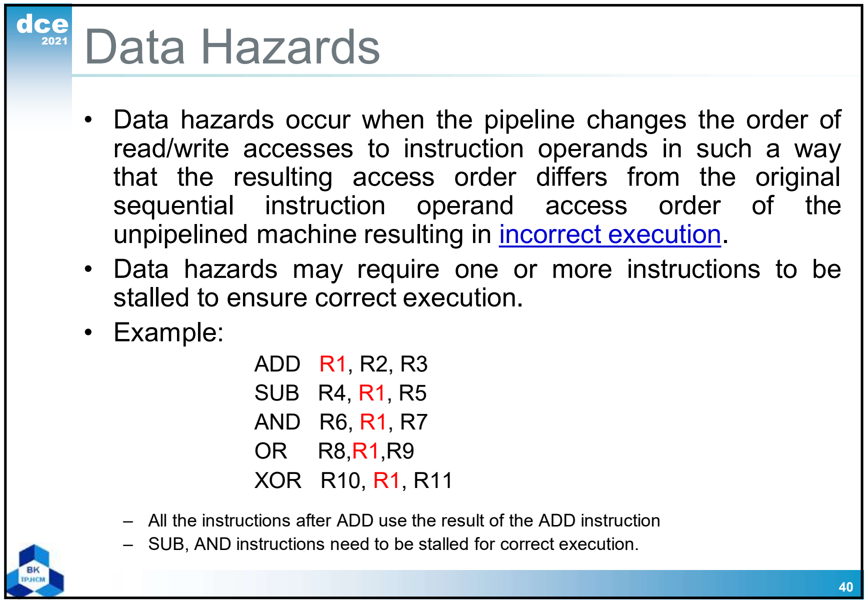


Remember the common case!

- All things being equal, a machine without structural hazards will always have a lower CPI.

- But, in some cases it may be better to allow them than to eliminate them.

- These are situations a computer architect might have to consider:
  - Is pipelining functional units or duplicating them costly in terms of HW?
  - Does structural hazard occur often?
  - What’s the common case???

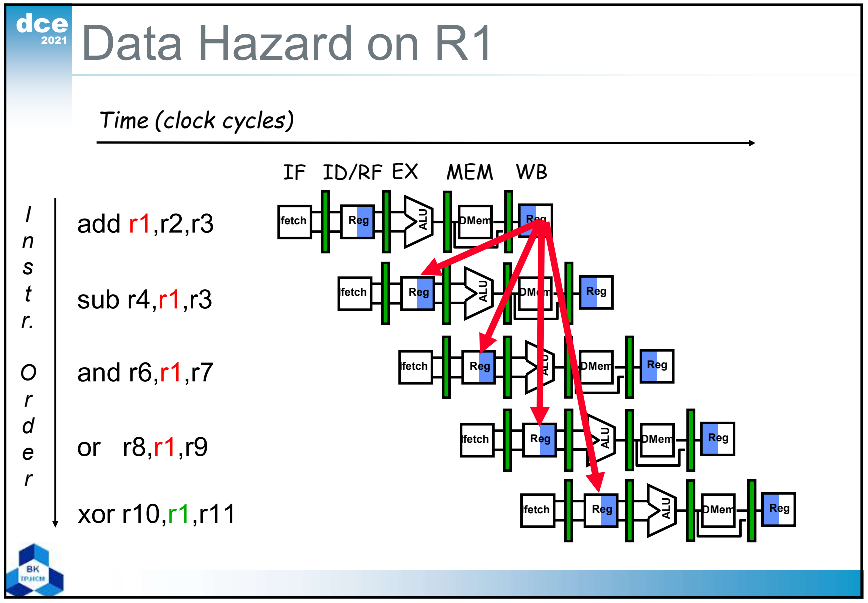


Data Hazards

- Data hazards occur when the pipeline changes the order of read/write accesses to instruction operands in such a way that the resulting access order differs from the original sequential instruction operand access order of the unpipelined machine resulting in incorrect execution.
- Data hazards may require one or more instructions to be stalled to ensure correct execution.
- Example:

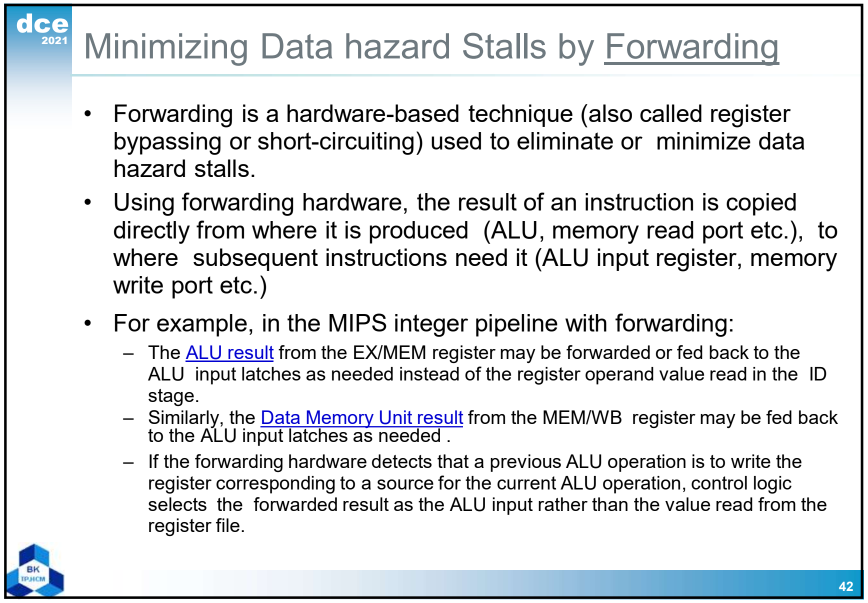
```
ADD   R1, R2, R3
SUB   R4, R1, R5
AND   R6, R1, R7
OR    R8,R1,R9
XOR   R10, R1, R11
```

- All the instructions after ADD use the result of the ADD instruction
- SUB, AND instructions need to be stalled for correct execution.



Data Hazard on R1

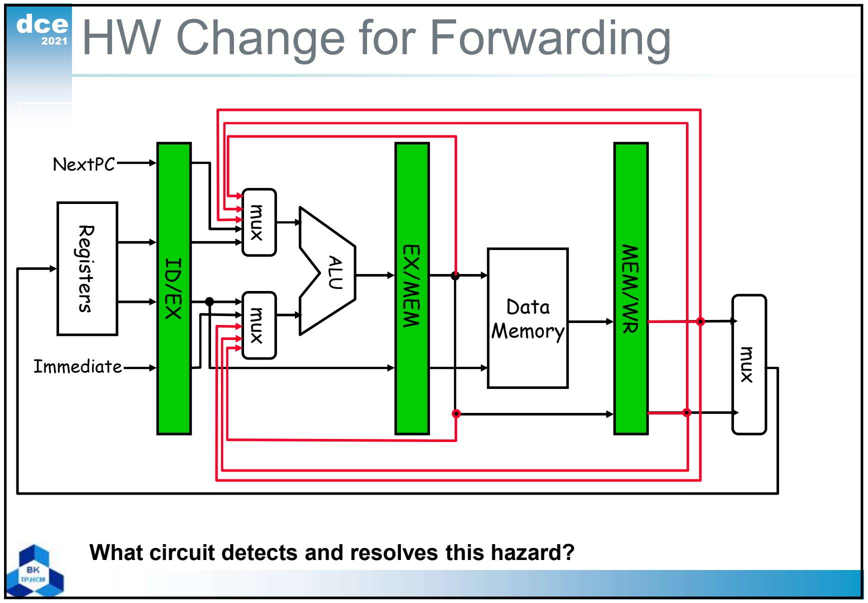
41



Minimizing Data hazard Stalls by Forwarding

- Forwarding is a hardware-based technique (also called register bypassing or short-circuiting) used to eliminate or minimize data hazard stalls.
- Using forwarding hardware, the result of an instruction is copied directly from where it is produced (ALU, memory read port etc.), to where subsequent instructions need it (ALU input register, memory write port etc.)
- For example, in the MIPS integer pipeline with forwarding:
  - The ALU result from the EX/MEM register may be forwarded or fed back to the ALU input latches as needed instead of the register operand value read in the ID stage.
  - Similarly, the Data Memory Unit result from the MEM/WB register may be fed back to the ALU input latches as needed .
  - If the forwarding hardware detects that a previous ALU operation is to write the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

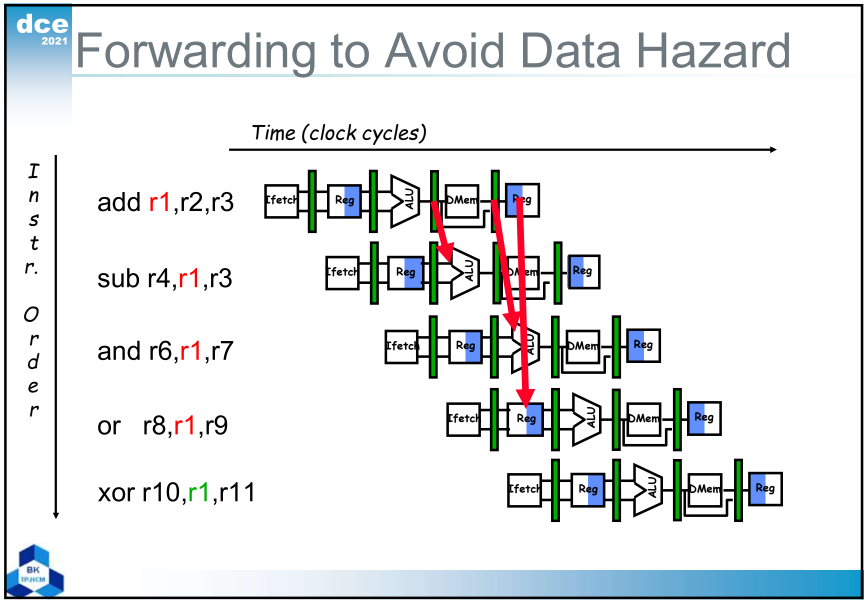
42



HW Change for Forwarding

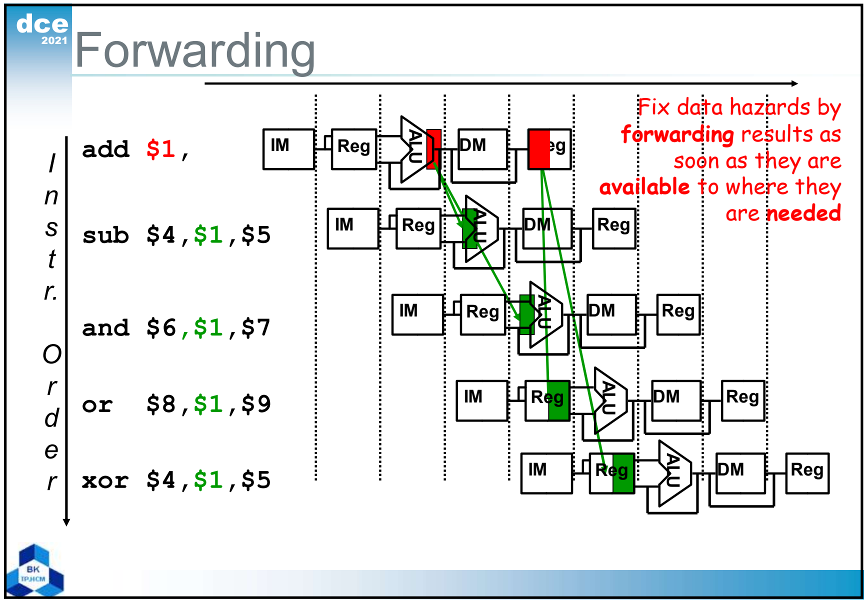
What circuit detects and resolves this hazard?

43



Forwarding to Avoid Data Hazard

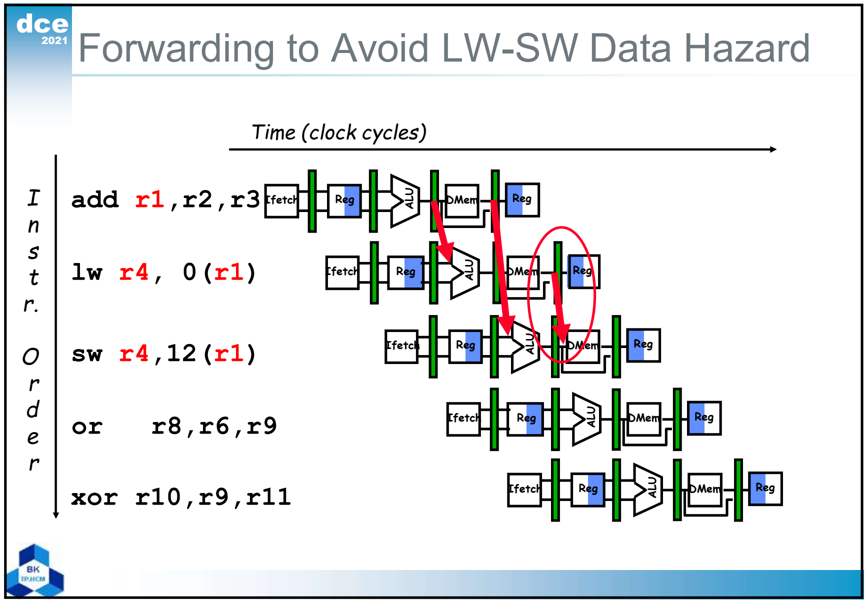
44



Forwarding

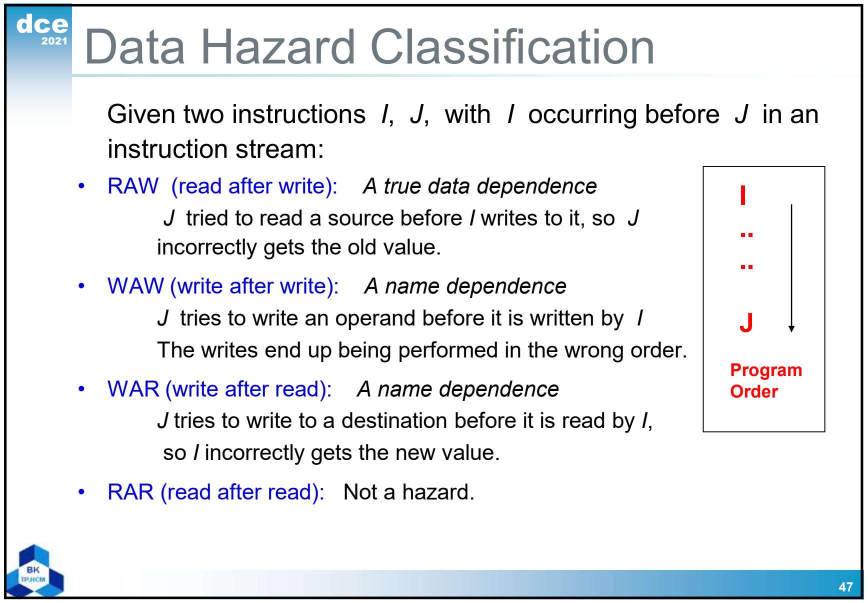
Fix data hazards by **forwarding** results as soon as they are **available** to where they are **needed**

45



Forwarding to Avoid LW-SW Data Hazard

46



Data Hazard Classification

Given two instructions *I*, *J*, with *I* occurring before *J* in an instruction stream:

- RAW (read after write): *A true data dependence*

  J tried to read a source before *I* writes to it, so *J* incorrectly gets the old value.

- WAW (write after write): *A name dependence*

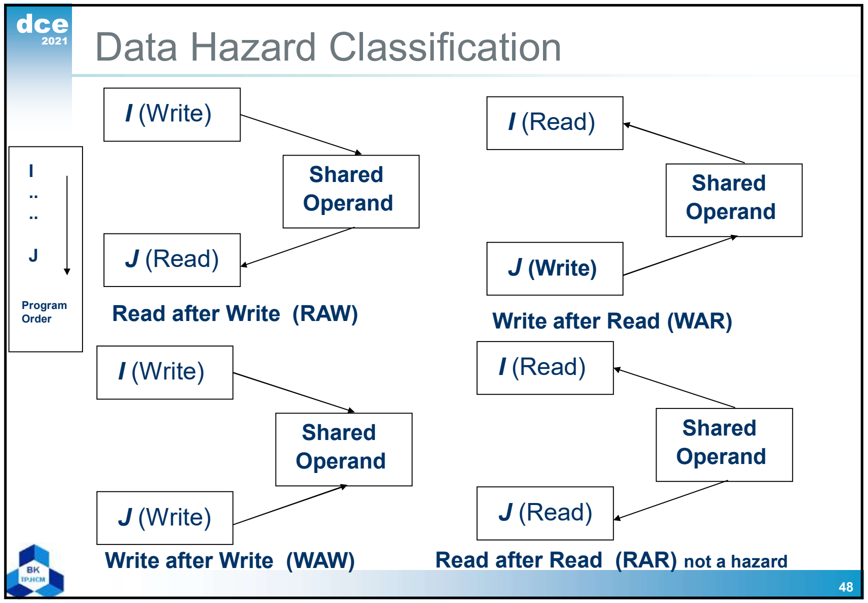
  J tries to write an operand before it is written by *I*. The writes end up being performed in the wrong order.

- WAR (write after read): *A name dependence*

  J tries to write to a destination before it is read by *I*, so *I* incorrectly gets the new value.

- RAR (read after read): Not a hazard.

47



Data Hazard Classification

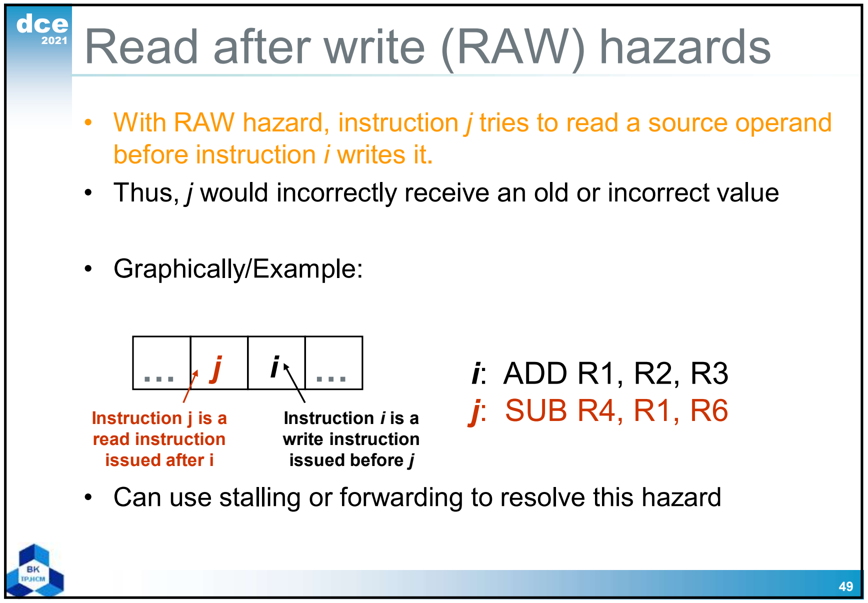
Read after Write (RAW)

Write after Read (WAR)

Write after Write (WAW)

Read after Read (RAR) not a hazard

48



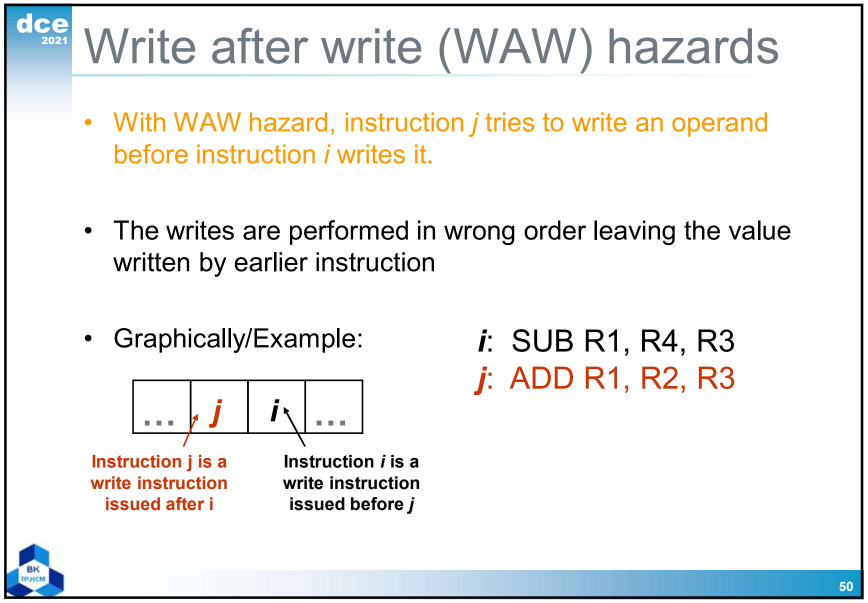
Read after write (RAW) hazards

- With RAW hazard, instruction *j* tries to read a source operand before instruction *i* writes it.
- Thus, *j* would incorrectly receive an old or incorrect value
- Graphically/Example:

… | *j* | *i* | …
Instruction j is a read instruction issued after i
Instruction *i* is a write instruction issued before *j*

i: ADD R1, R2, R3
j: SUB R4, R1, R6

- Can use stalling or forwarding to resolve this hazard

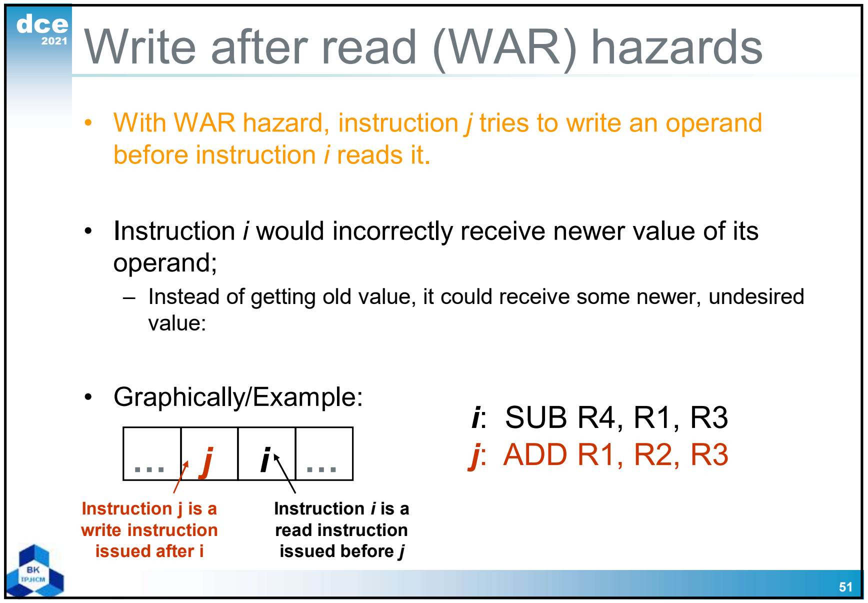


Write after write (WAW) hazards

- With WAW hazard, instruction *j* tries to write an operand before instruction *i* writes it.
- The writes are performed in wrong order leaving the value written by earlier instruction
- Graphically/Example:

i: SUB R1, R4, R3
j: ADD R1, R2, R3

… | *j* | *i* | …
Instruction j is a write instruction issued after i
Instruction *i* is a write instruction issued before *j*

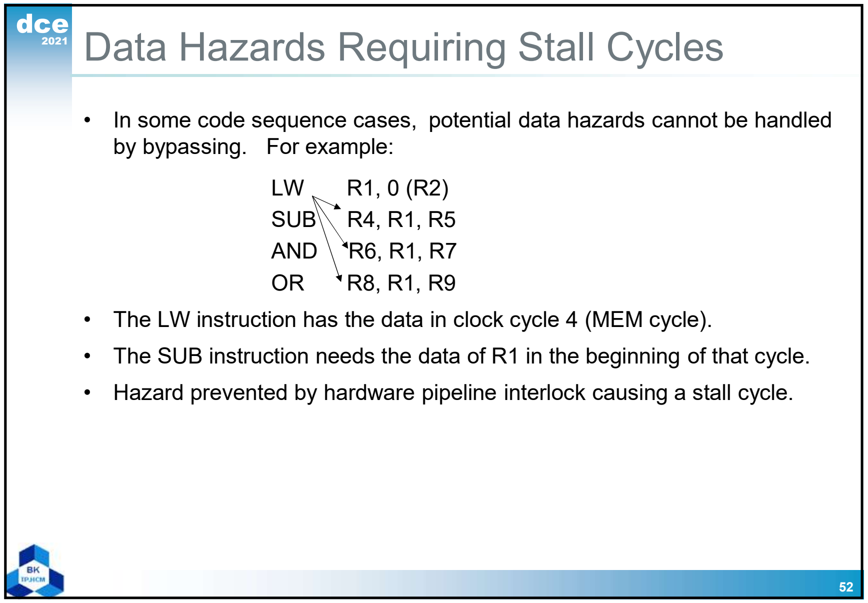


Write after read (WAR) hazards

- With WAR hazard, instruction *j* tries to write an operand before instruction *i* reads it.
- Instruction *i* would incorrectly receive newer value of its operand;
  - Instead of getting old value, it could receive some newer, undesired value:
- Graphically/Example:

i: SUB R4, R1, R3
j: ADD R1, R2, R3

… | *j* | *i* | …
Instruction j is a write instruction issued after i
Instruction *i* is a read instruction issued before *j*



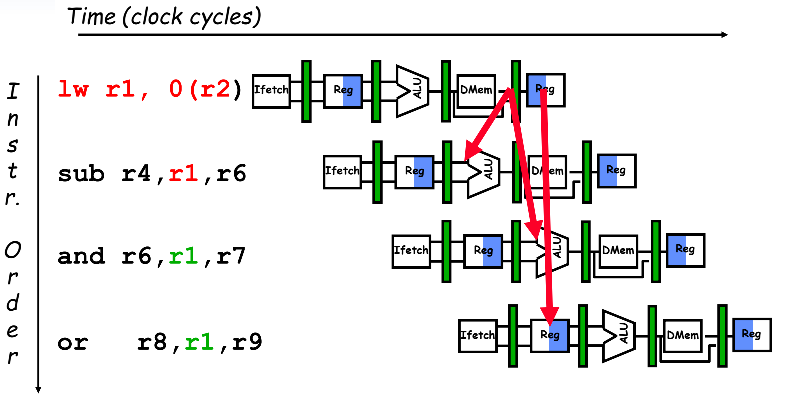
Data Hazards Requiring Stall Cycles

- In some code sequence cases, potential data hazards cannot be handled by bypassing. For example:

```
LW     R1, 0 (R2)
SUB    R4, R1, R5
AND    R6, R1, R7
OR     R8, R1, R9
```

- The LW instruction has the data in clock cycle 4 (MEM cycle).
- The SUB instruction needs the data of R1 in the beginning of that cycle.
- Hazard prevented by hardware pipeline interlock causing a stall cycle.

Data Hazard Even with Forwarding



Time (clock cycles)

Instr. Order

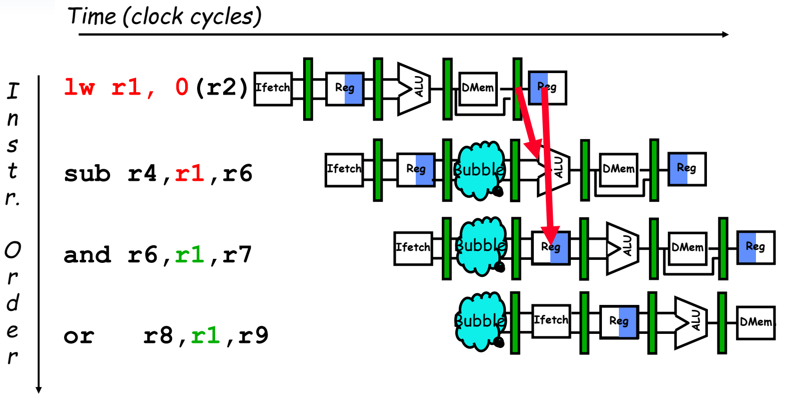
lw r1, 0(r2)

sub r4,r1,r6

and r6,r1,r7

or r8,r1,r9

Data Hazard Even with Forwarding



Time (clock cycles)

Instr. Order

lw r1, 0(r2)

sub r4,r1,r6

and r6,r1,r7

or r8,r1,r9

Hardware Pipeline Interlocks

- A hardware pipeline interlock detects a data hazard and stalls the pipeline until the hazard is cleared.
- The CPI for the stalled instruction increases by the length of the stall.
- For the Previous example, (no stall cycle):

LW R1, 0(R1)	IF	ID	EX	MEM	WB			
SUB R4,R1,R5		IF	ID	EX	MEM	WB		
AND R6,R1,R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

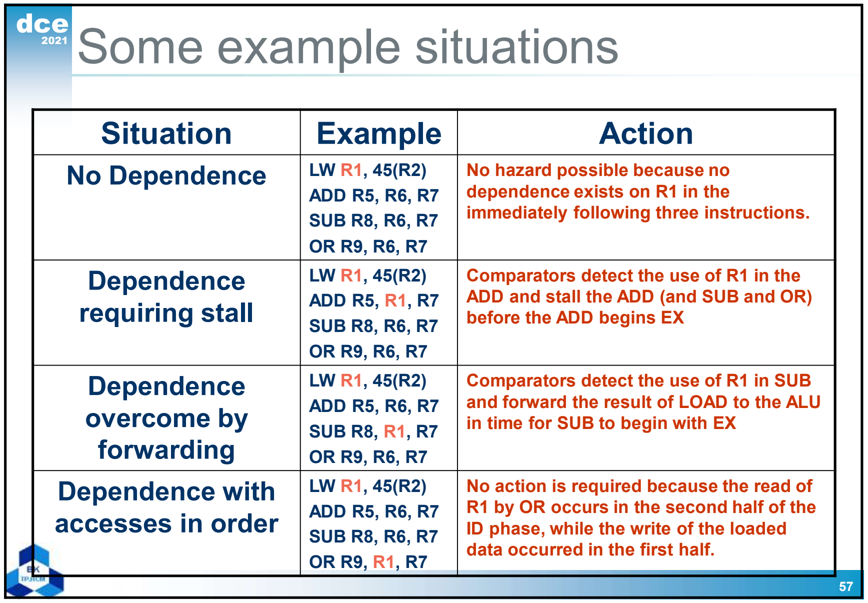
With Stall Cycle:

Stall + Forward

LW R1, 0(R1)	IF	ID	EX	MEM	WB				
SUB R4,R1,R5		IF	ID	STALL	EX	MEM	WB		
AND R6,R1,R7			IF	STALL	ID	EX	MEM	WB	
OR R8, R1, R9				STALL	IF	ID	EX	MEM	WB

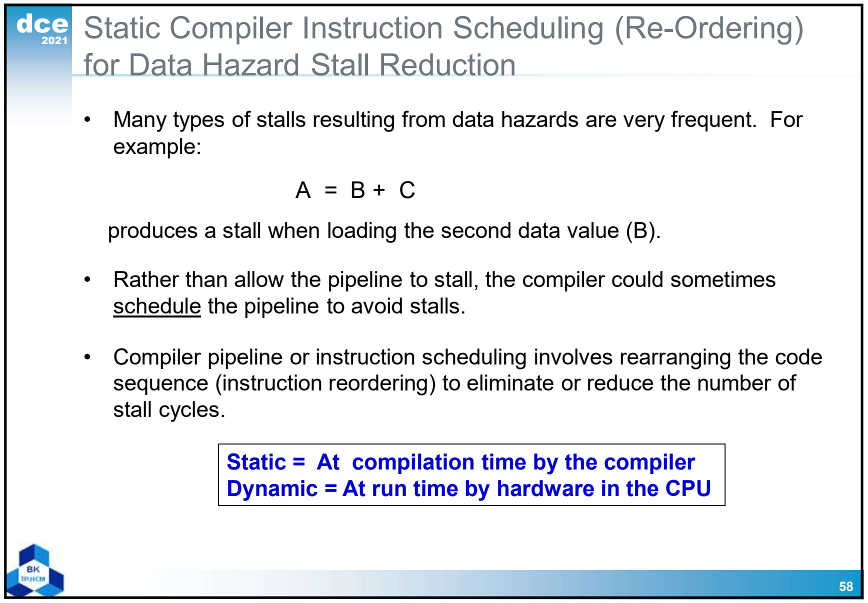
Data hazards and the compiler

- Compiler should be able to help eliminate some stalls caused by data hazards
- i.e. compiler could not generate a LOAD instruction that is immediately followed by instruction that uses result of LOAD's destination register.
- Technique is called "pipeline/instruction scheduling"



Some example situations

Situation	Example	Action
No Dependence	LW R1, 45(R2) ADD R5, R6, R7 SUB R8, R6, R7 OR R9, R6, R7	No hazard possible because no dependence exists on R1 in the immediately following three instructions.
Dependence requiring stall	LW R1, 45(R2) ADD R5, R1, R7 SUB R8, R6, R7 OR R9, R6, R7	Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX
Dependence overcome by forwarding	LW R1, 45(R2) ADD R5, R6, R7 SUB R8, R1, R7 OR R9, R6, R7	Comparators detect the use of R1 in SUB and forward the result of LOAD to the ALU in time for SUB to begin with EX
Dependence with accesses in order	LW R1, 45(R2) ADD R5, R6, R7 SUB R8, R6, R7 OR R9, R1, R7	No action is required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.



Static Compiler Instruction Scheduling (Re-Ordering) for Data Hazard Stall Reduction

- Many types of stalls resulting from data hazards are very frequent. For example:

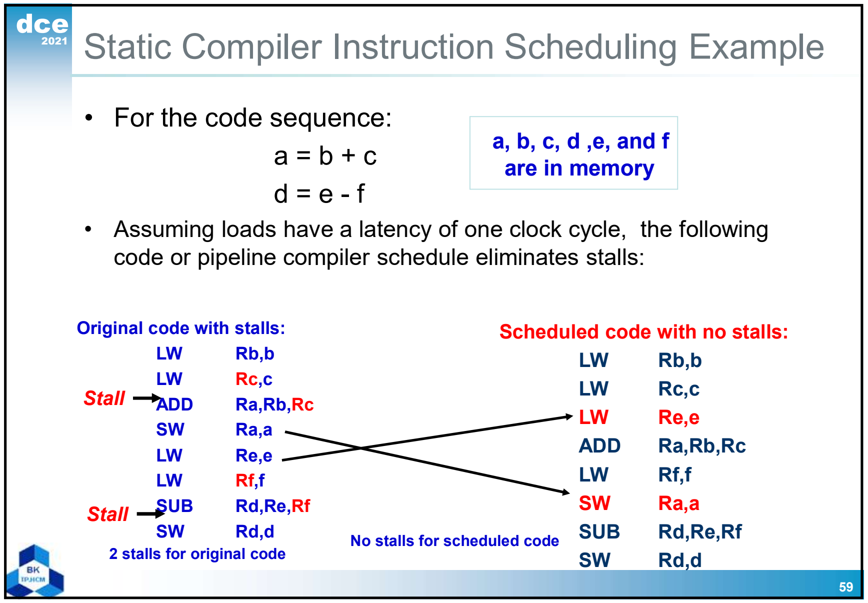
  A = B + C

  produces a stall when loading the second data value (B).

- Rather than allow the pipeline to stall, the compiler could sometimes schedule the pipeline to avoid stalls.

- Compiler pipeline or instruction scheduling involves rearranging the code sequence (instruction reordering) to eliminate or reduce the number of stall cycles.

Static = At compilation time by the compiler
Dynamic = At run time by hardware in the CPU



Static Compiler Instruction Scheduling Example

- For the code sequence:

  a = b + c

  d = e - f

a, b, c, d ,e, and f are in memory

- Assuming loads have a latency of one clock cycle, the following code or pipeline compiler schedule eliminates stalls:

Original code with stalls:

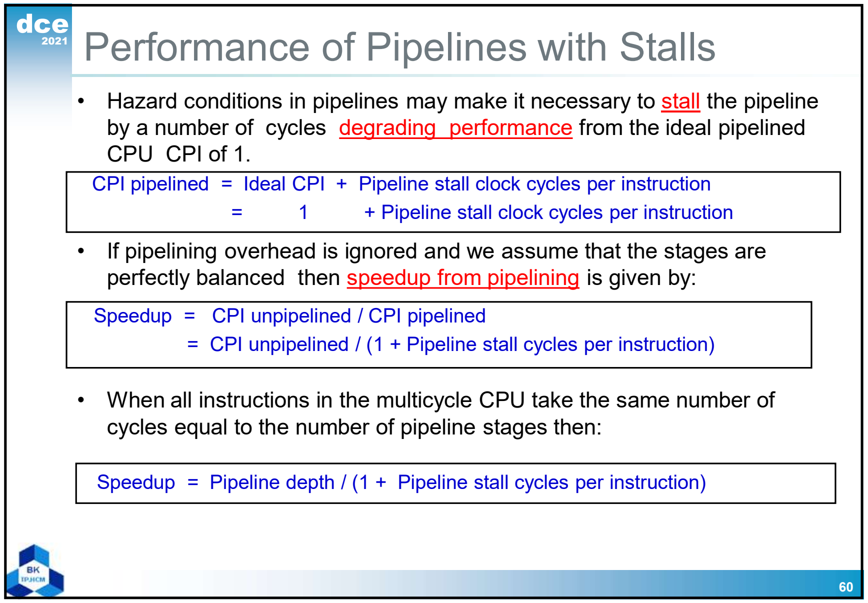
```
      LW   Rb,b
      LW   Rc,c
Stall ADD  Ra,Rb,Rc
      SW   Ra,a
      LW   Re,e
      LW   Rf,f
Stall SUB  Rd,Re,Rf
      SW   Rd,d
```

2 stalls for original code

Scheduled code with no stalls:

```
LW   Rb,b
LW   Rc,c
LW   Re,e
ADD  Ra,Rb,Rc
LW   Rf,f
SW   Ra,a
SUB  Rd,Re,Rf
SW   Rd,d
```

No stalls for scheduled code



Performance of Pipelines with Stalls

- Hazard conditions in pipelines may make it necessary to stall the pipeline by a number of cycles degrading performance from the ideal pipelined CPU CPI of 1.

CPI pipelined = Ideal CPI + Pipeline stall clock cycles per instruction
= 1 + Pipeline stall clock cycles per instruction

- If pipelining overhead is ignored and we assume that the stages are perfectly balanced then speedup from pipelining is given by:

Speedup = CPI unpipelined / CPI pipelined
= CPI unpipelined / (1 + Pipeline stall cycles per instruction)

- When all instructions in the multicycle CPU take the same number of cycles equal to the number of pipeline stages then:

Speedup = Pipeline depth / (1 + Pipeline stall cycles per instruction)

Control Hazards

- When a conditional branch is executed it may change the PC and, without any special measures, leads to stalling the pipeline for a number of cycles until the branch condition is known (branch is resolved).
  - Otherwise the PC may not be correct when needed in IF
- In current MIPS pipeline, the conditional branch is resolved in stage 4 (MEM stage) resulting in three stall cycles as shown below:

Branch instruction	IF	ID	EX	MEM	WB					
Branch successor		stall	stall	stall	IF	ID	EX	MEM	WB	
Branch successor + 1						IF	ID	EX	MEM	WB
Branch successor + 2							IF	ID	EX	MEM
Branch successor + 3								IF	ID	EX
Branch successor + 4									IF	ID
Branch successor + 5										IF

3 stall cycles

Assuming we stall or flush the pipeline on a branch instruction:
Three clock cycles are wasted for every branch for current MIPS pipeline

Branch Penalty = stage number where branch is resolved - 1
here Branch Penalty = 4 - 1 = 3 Cycles

Control Hazard on Branches
Three Stage Stall

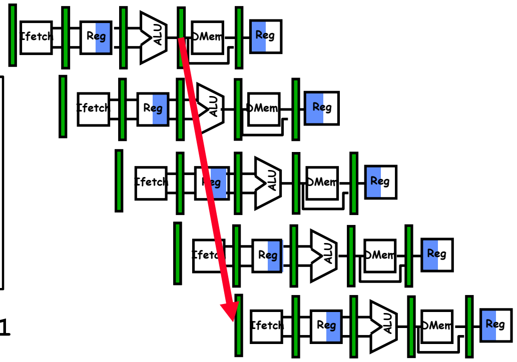
```
10: beq r1,r3,36

14: and r2,r3,r5

18: or  r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11
```



Reducing Branch Stall Cycles

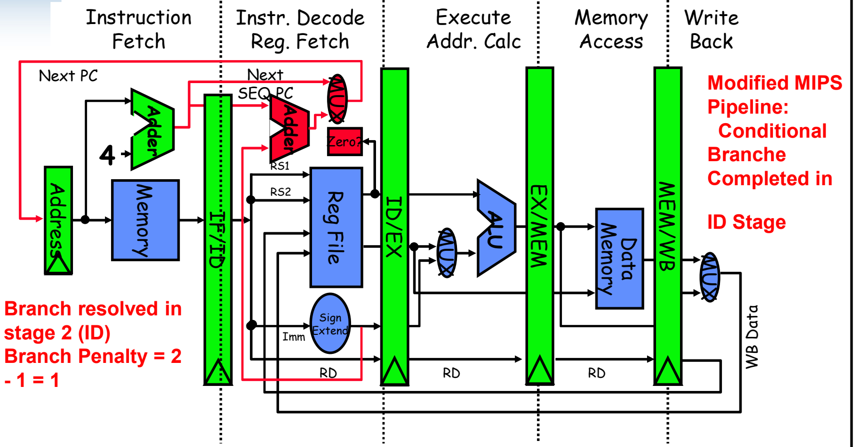
Pipeline hardware measures to reduce branch stall cycles:

1- Find out whether a branch is taken earlier in the pipeline.
2- Compute the taken PC earlier in the pipeline.

In MIPS:

- In MIPS branch instructions BEQZ, BNE, test a register for equality to zero.
- This can be completed in the ID cycle by moving the zero test into that cycle.
- Both PCs (taken and not taken) must be computed early.
- Requires an additional adder because the current ALU is not useable until EX cycle.
- This results in just a single cycle stall on branches.

Pipelined MIPS Datapath



Modified MIPS Pipeline: Conditional Branche Completed in ID Stage

Branch resolved in stage 2 (ID) Branch Penalty = 2 - 1 = 1

- Interplay of instruction set design and cycle time.

Branch Stall Impact

- If CPI = 1, 30% branch,
  Stall 3 cycles => new CPI = 1.9!
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ≠ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken

- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

Most Common

#3: Predict Branch Taken

- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
  - MIPS still incurs 1 cycle branch penalty
  - Other machines: branch target known before outcome
- What happens when hit not-taken branch?

Predict Branch Not-Taken Scheme

(most common scheme)

Not Taken Branch (no stall)

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Taken Branch (stall)

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction *i* + 1	Stall	IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

Assuming the MIPS pipeline with reduced branch penalty = 1

The predict-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom).

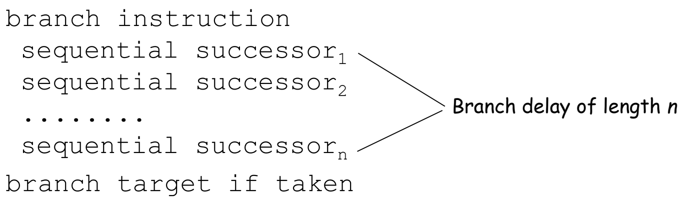
Stall when the branch is taken

Pipeline stall cycles from branches = frequency of taken branches X branch penalty

Four Branch Hazard Alternatives

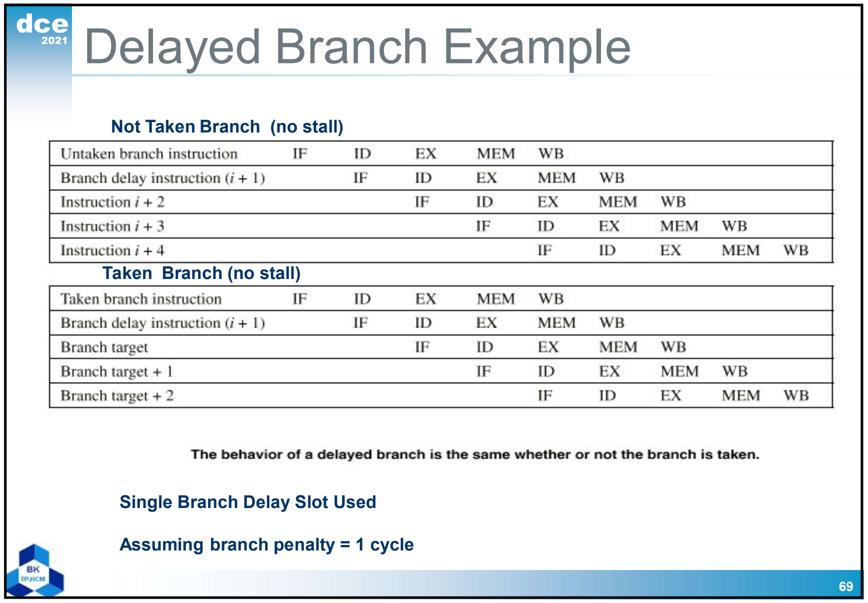
#4: Delayed Branch

- Define branch to take place AFTER a following instruction



```
branch instruction
 sequential successor₁
 sequential successor₂
 ........
 sequential successorₙ
branch target if taken
```

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this



Delayed Branch Example

Not Taken Branch (no stall)

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (*i* + 1)		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Taken Branch (no stall)

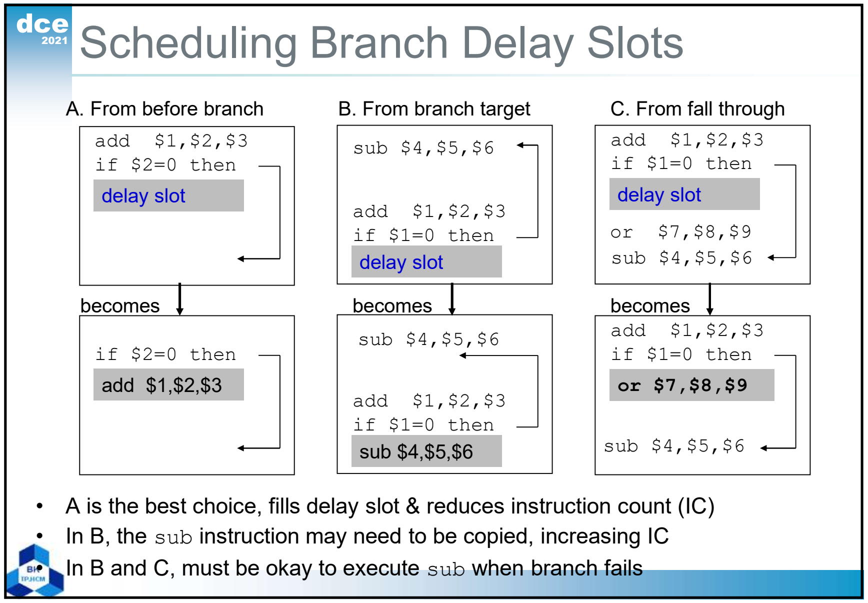
Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction (*i* + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

The behavior of a delayed branch is the same whether or not the branch is taken.

Single Branch Delay Slot Used

Assuming branch penalty = 1 cycle

69



Scheduling Branch Delay Slots

A. From before branch

```
add  $1,$2,$3
if $2=0 then
delay slot
```

becomes

```
if $2=0 then
add $1,$2,$3
```

B. From branch target

```
sub $4,$5,$6

add  $1,$2,$3
if $1=0 then
delay slot
```

becomes

```
sub $4,$5,$6

add  $1,$2,$3
if $1=0 then
sub $4,$5,$6
```

C. From fall through

```
add  $1,$2,$3
if $1=0 then
delay slot

or  $7,$8,$9
sub $4,$5,$6
```

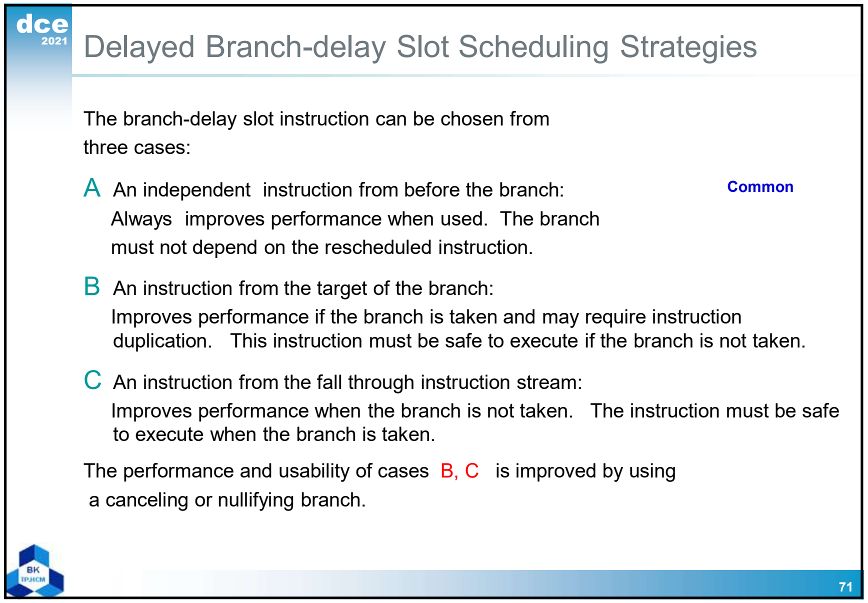
becomes

```
add  $1,$2,$3
if $1=0 then
or  $7,$8,$9

sub $4,$5,$6
```

- A is the best choice, fills delay slot & reduces instruction count (IC)
- In B, the `sub` instruction may need to be copied, increasing IC
- In B and C, must be okay to execute `sub` when branch fails

70



Delayed Branch-delay Slot Scheduling Strategies

The branch-delay slot instruction can be chosen from three cases:

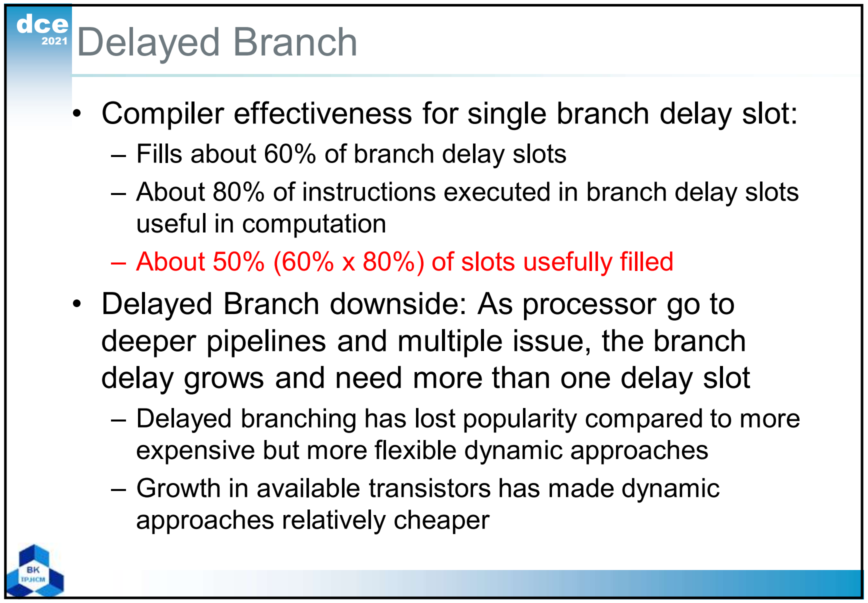
A An independent instruction from before the branch: **Common**
Always improves performance when used. The branch must not depend on the rescheduled instruction.

B An instruction from the target of the branch:
Improves performance if the branch is taken and may require instruction duplication. This instruction must be safe to execute if the branch is not taken.

C An instruction from the fall through instruction stream:
Improves performance when the branch is not taken. The instruction must be safe to execute when the branch is taken.

The performance and usability of cases B, C is improved by using a canceling or nullifying branch.

71



Delayed Branch

- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: As processor go to deeper pipelines and multiple issue, the branch delay grows and need more than one delay slot
  - Delayed branching has lost popularity compared to more expensive but more flexible dynamic approaches
  - Growth in available transistors has made dynamic approaches relatively cheaper

72

Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Assume: 4% unconditional branch,
6% conditional branch- untaken,
10% conditional branch-taken

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. unpipelined*	*speedup v. stall*
Stall pipeline	3	1.60	3.1	1.0
Predict not taken	1x0.04+3x0.10	1.34	3.7	1.19
Predict taken	1x0.14+2x0.06	1.26	4.0	1.29
Delayed branch	0.5	1.10	4.5	1.45

Pipeline Performance Example

- Assume the following MIPS instruction mix:

Type	Frequency	
Arith/Logic	40%	
Load	30%	of which 25% are followed immediately by an instruction using the loaded value
Store	10%	
branch	20%	of which 45% are taken

- What is the resulting CPI for the pipelined MIPS with forwarding and branch address calculation in ID stage when using a branch not-taken scheme?

Branch Penalty = 1 cycle

- CPI = Ideal CPI + Pipeline stall clock cycles per instruction

```
  =    1  +     stalls by loads   +   stalls by branches
  =    1  +     .3 x .25 x 1      +     .2 x .45 x 1
  =    1  +     .075              +     .09
  =    1.165
```

Pipelining Summary

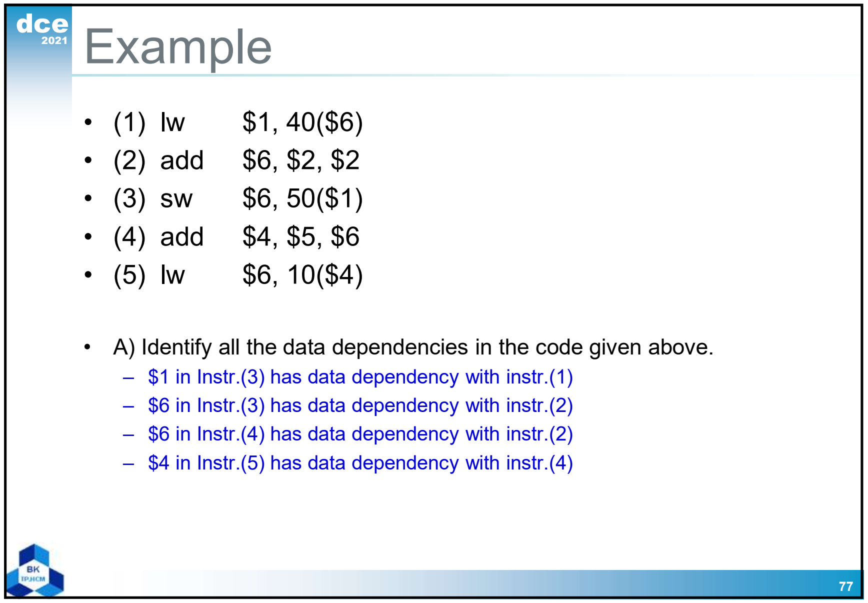
- Pipelining overlaps the execution of multiple instructions.
- With an idea pipeline, the CPI is one, and the speedup is equal to the number of stages in the pipeline.
- However, several factors prevent us from achieving the ideal speedup, including
  - Not being able to divide the pipeline evenly
  - The time needed to empty and flush the pipeline
  - Overhead needed for pipelining
  - Structural, data, and control hazards
- Just overlap tasks, and easy if tasks are independent

Pipelining Summary

- Speed Up VS. Pipeline Depth; if ideal CPI is 1, then:

$$\textbf{Speedup} = \frac{\textbf{Pipeline Depth}}{\textbf{1 + Pipeline stall CPI}} \textbf{ X } \frac{\textbf{Clock Cycle Unpipelined}}{\textbf{Clock Cycle Pipelined}}$$

- Hazards limit performance
  - Structural: need more HW resources
  - Data: need forwarding, compiler scheduling
  - Control: early evaluation & PC, delayed branch, prediction
- Increasing length of pipe increases impact of hazards; pipelining helps instruction bandwidth, not latency
- Compilers reduce cost of data and control hazards
  - Load delay slots
  - Branch delay slots
  - Branch prediction

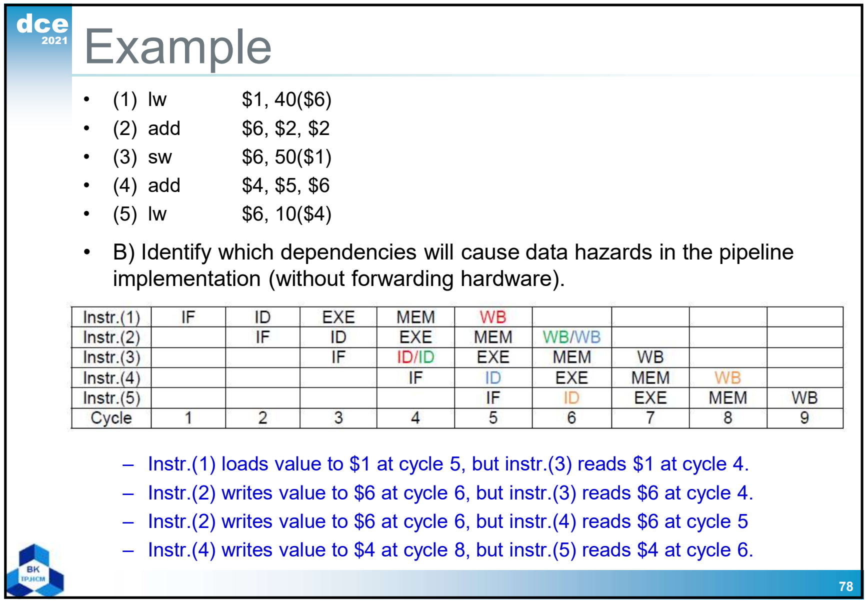


dce 2021

Example

- (1) lw $1, 40($6)
- (2) add $6, $2, $2
- (3) sw $6, 50($1)
- (4) add $4, $5, $6
- (5) lw $6, 10($4)

- A) Identify all the data dependencies in the code given above.
  - $1 in Instr.(3) has data dependency with instr.(1)
  - $6 in Instr.(3) has data dependency with instr.(2)
  - $6 in Instr.(4) has data dependency with instr.(2)
  - $4 in Instr.(5) has data dependency with instr.(4)



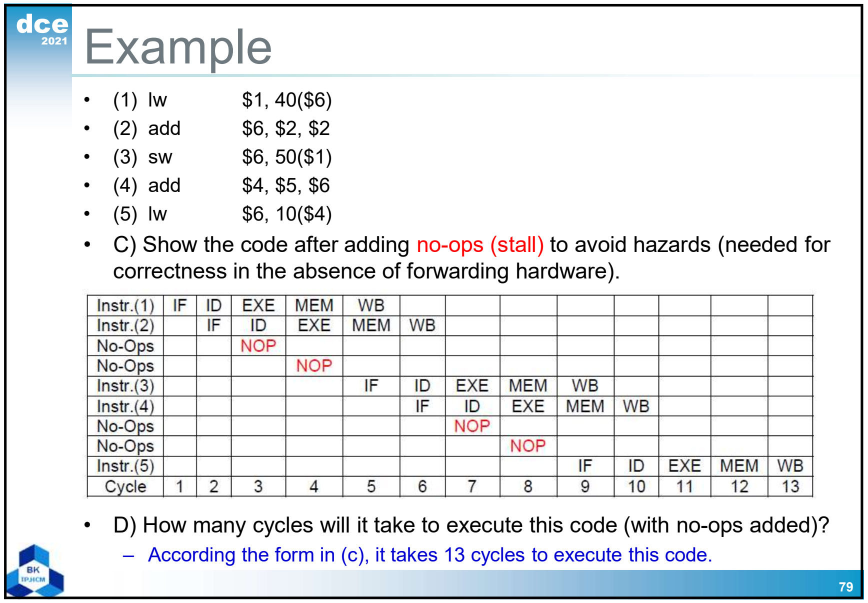
dce 2021

Example

- (1) lw $1, 40($6)
- (2) add $6, $2, $2
- (3) sw $6, 50($1)
- (4) add $4, $5, $6
- (5) lw $6, 10($4)
- B) Identify which dependencies will cause data hazards in the pipeline implementation (without forwarding hardware).

Instr.(1)	IF	ID	EXE	MEM	WB				
Instr.(2)		IF	ID	EXE	MEM	WB/WB			
Instr.(3)			IF	ID/ID	EXE	MEM	WB		
Instr.(4)				IF	ID	EXE	MEM	WB	
Instr.(5)					IF	ID	EXE	MEM	WB
Cycle	1	2	3	4	5	6	7	8	9

- Instr.(1) loads value to $1 at cycle 5, but instr.(3) reads $1 at cycle 4.
- Instr.(2) writes value to $6 at cycle 6, but instr.(3) reads $6 at cycle 4.
- Instr.(2) writes value to $6 at cycle 6, but instr.(4) reads $6 at cycle 5
- Instr.(4) writes value to $4 at cycle 8, but instr.(5) reads $4 at cycle 6.



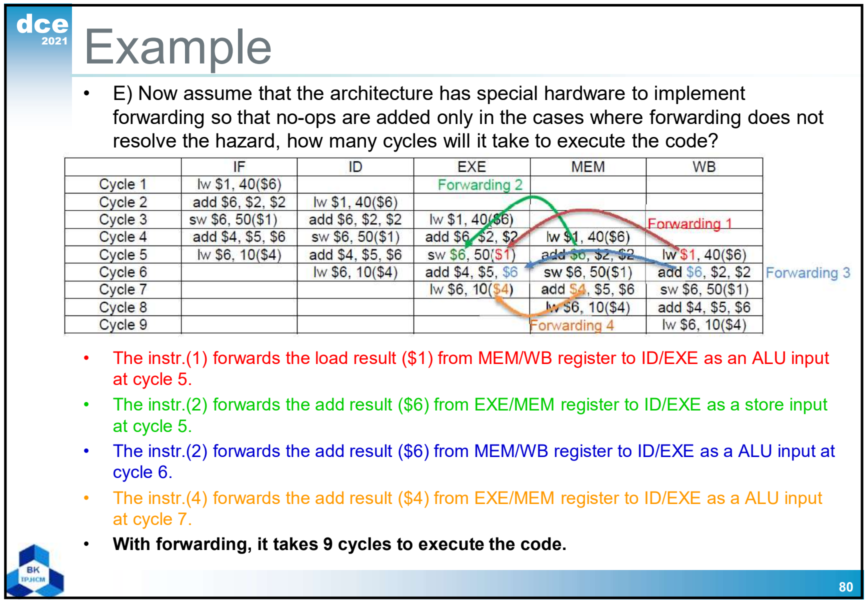
dce 2021

Example

- (1) lw $1, 40($6)
- (2) add $6, $2, $2
- (3) sw $6, 50($1)
- (4) add $4, $5, $6
- (5) lw $6, 10($4)
- C) Show the code after adding no-ops (stall) to avoid hazards (needed for correctness in the absence of forwarding hardware).

Instr.(1)	IF	ID	EXE	MEM	WB								
Instr.(2)		IF	ID	EXE	MEM	WB							
No-Ops			NOP										
No-Ops				NOP									
Instr.(3)					IF	ID	EXE	MEM	WB				
Instr.(4)						IF	ID	EXE	MEM	WB			
No-Ops							NOP						
No-Ops								NOP					
Instr.(5)									IF	ID	EXE	MEM	WB
Cycle	1	2	3	4	5	6	7	8	9	10	11	12	13

- D) How many cycles will it take to execute this code (with no-ops added)?
  - According the form in (c), it takes 13 cycles to execute this code.



dce 2021

Example

- E) Now assume that the architecture has special hardware to implement forwarding so that no-ops are added only in the cases where forwarding does not resolve the hazard, how many cycles will it take to execute the code?

	IF	ID	EXE	MEM	WB
Cycle 1	lw $1, 40($6)				
Cycle 2	add $6, $2, $2	lw $1, 40($6)			
Cycle 3	sw $6, 50($1)	add $6, $2, $2	lw $1, 40($6)		
Cycle 4	add $4, $5, $6	sw $6, 50($1)	add $6, $2, $2	lw $1, 40($6)	
Cycle 5	lw $6, 10($4)	add $4, $5, $6	sw $6, 50($1)	add $6, $2, $2	lw $1, 40($6)
Cycle 6		lw $6, 10($4)	add $4, $5, $6	sw $6, 50($1)	add $6, $2, $2
Cycle 7			lw $6, 10($4)	add $4, $5, $6	sw $6, 50($1)
Cycle 8				lw $6, 10($4)	add $4, $5, $6
Cycle 9					lw $6, 10($4)

- The instr.(1) forwards the load result ($1) from MEM/WB register to ID/EXE as an ALU input at cycle 5.
- The instr.(2) forwards the add result ($6) from EXE/MEM register to ID/EXE as a store input at cycle 5.
- The instr.(2) forwards the add result ($6) from MEM/WB register to ID/EXE as a ALU input at cycle 6.
- The instr.(4) forwards the add result ($4) from EXE/MEM register to ID/EXE as a ALU input at cycle 7.
- **With forwarding, it takes 9 cycles to execute the code.**