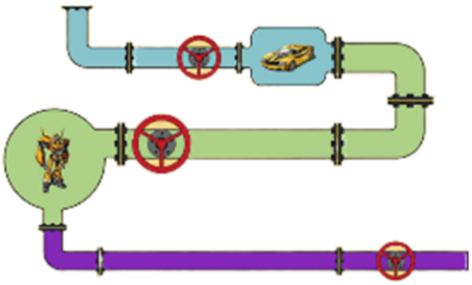
# معماری کامپیوتر

## فصل هفت

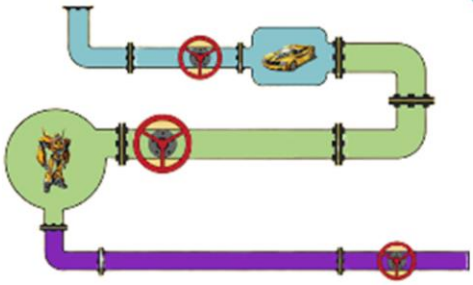
## پردازش خط لوله



# Computer Architecture

Chapter Seven

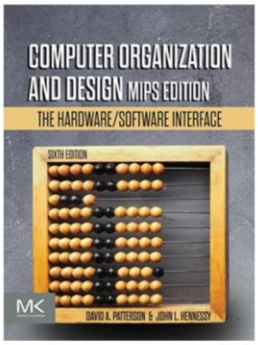
## Pipelining



# Copyright Notice

The slides of this lecture are adopted from:

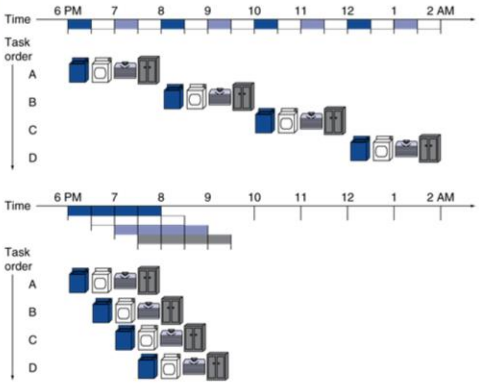
- D. Patterson & J. Hennessey, “Computer Organization & Design, The Hardware/Software Interface”, 6th Ed., MK publishing, 2020



# Pipelining Analogy



- Pipelined laundry: overlapping execution
  - Parallelism improves performance



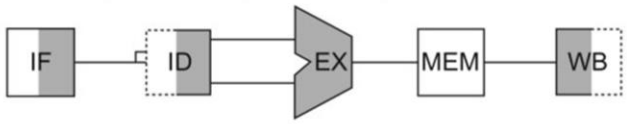




- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/(0.5n+1.5) ≈ 4 = number of stages

# 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

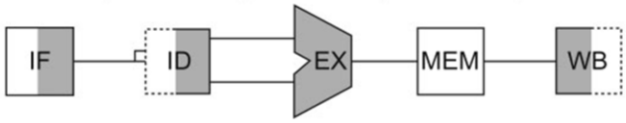


# Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

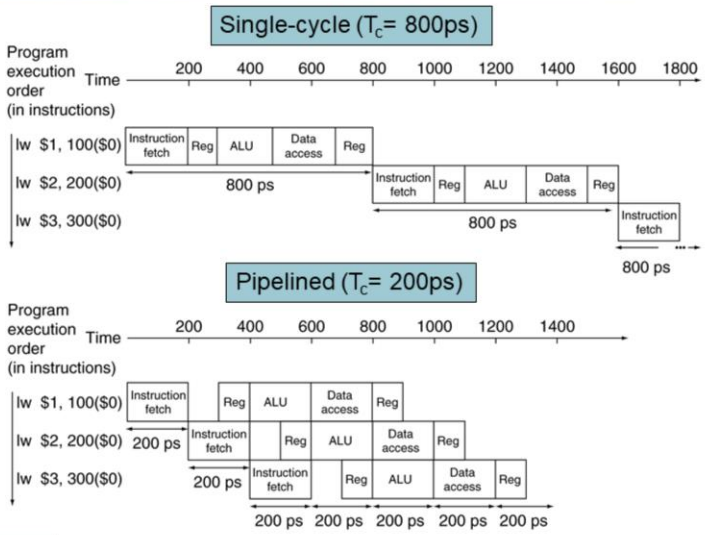
| Instruction | Fetch | Register read | ALU op | Memory access | Register write | Total time |
|---|---|---|---|---|---|---|
| lw | | | | | | |
| sw | | | | | | |
| R-format | | | | | | |
| beq | | | | | | |

# Pipeline Performance




| Instruction | Fetch | Register read | ALU op | Memory access | Register write | Total time |
|---|---|---|---|---|---|---|
| lw | 200 ps | 100 ps | 200 ps | 200 ps | 100 ps | 800 ps |
| sw | 200 ps | 100 ps | 200 ps | 200 ps | | 700 ps |
| R-format | 200 ps | 100 ps | 200 ps | | 100 ps | 600 ps |
| beq | 200 ps | 100 ps | 200 ps | | | 500 ps |

# Pipeline Performance



# Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time

$$\text{Time between instructions}_{pipelined} = \frac{\text{Time between instructions}_{nonpipelined}}{\text{Number of stages}}$$

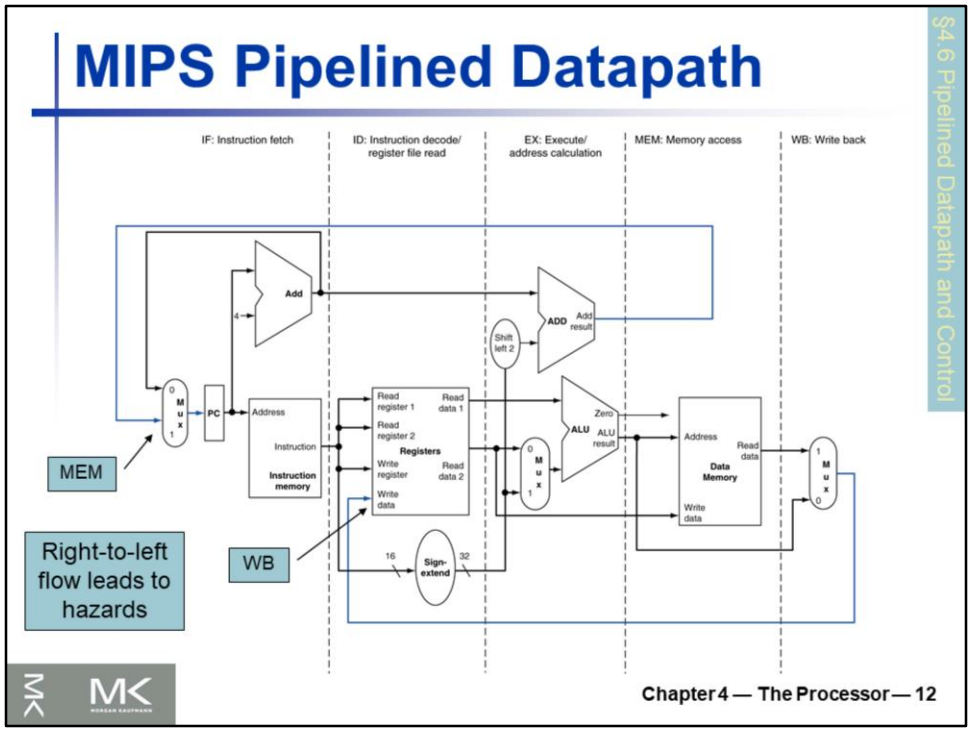
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) does not decrease

# Pipeline Speedup (cont.)

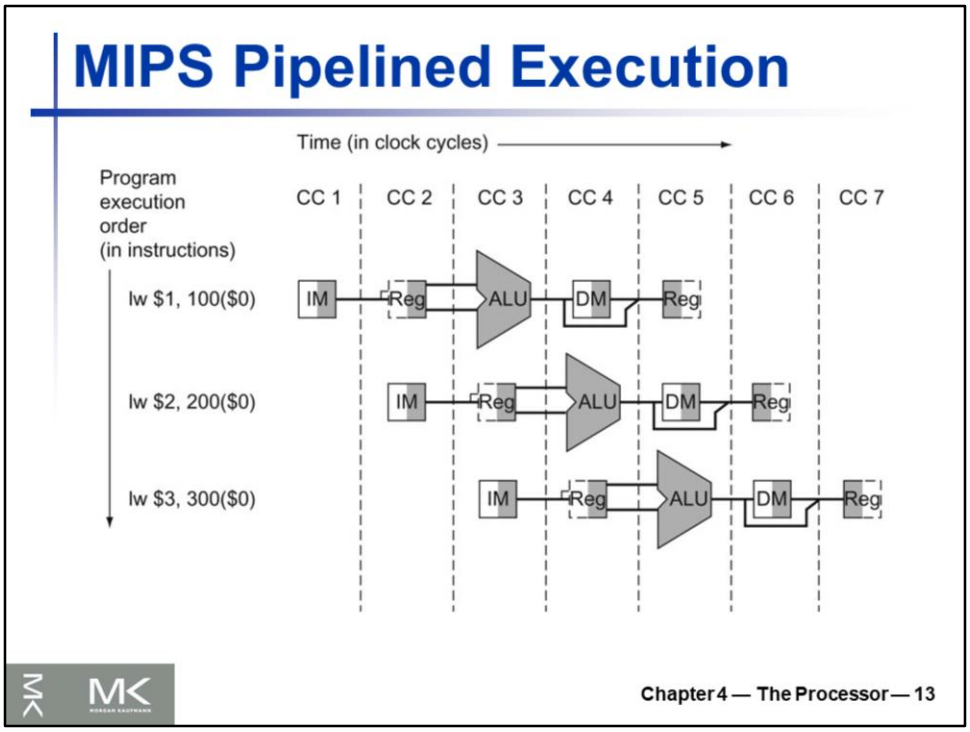
- Under ideal conditions and with a large number of instructions, the speed-up from pipelining is approximately equal to the number of pipe stages
- But due to some overhead speed-up will be less than the number of pipeline stages

# Pipelining and ISA Design

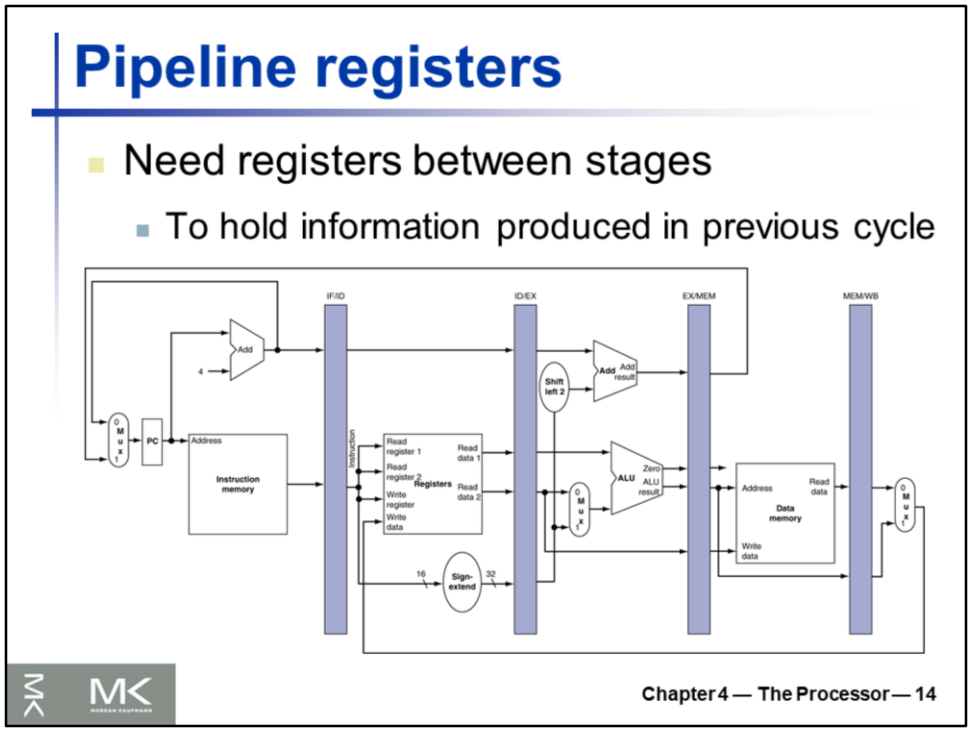
- MIPS ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch in one cycle and decode in next cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Register fields located in the same place in all instructions
    - Can decode and read registers in one step
  - Load/store addressing
    - Calculate address in 3rd stage, access memory in 4th stage
    - No need to read operands from mem before ALU stage
  - Alignment of memory operands
    - Memory access takes only one cycle




Each step of the instruction can be mapped onto the datapath from left to right. The only exceptions are the **update of the PC** and the **write-back** step, shown in color, which sends either the ALU result or the data from memory to the left to be written into the register file.
Note that the first right-to-left flow of data can lead to **data hazards** and the second leads to **control hazards**.



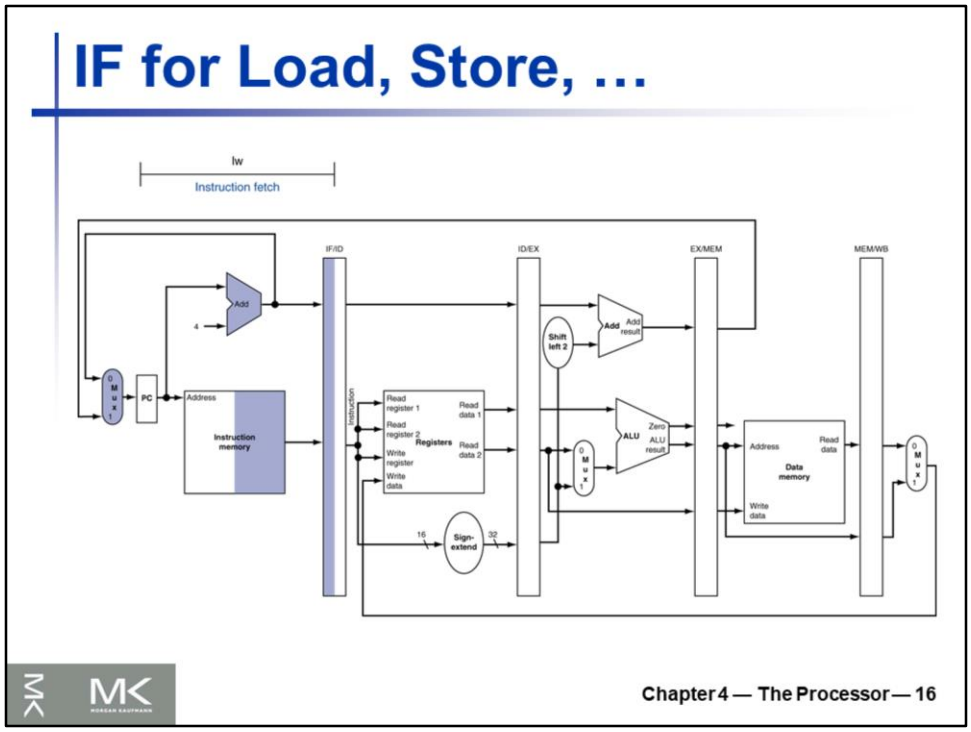

This figure pretends that each instruction has its own datapath, and shades each portion according to use. Each stage is
labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 4.33.
***IM*** represents the instruction memory and the PC in the instruction fetch stage, ***Reg*** stands for the register file and sign extender in the instruction decode/register file read stage (ID), and so on.
To maintain proper time order, this stylized datapath breaks the register file into two logical parts:
registers read during register fetch (ID) and registers written during write back (WB).
This dual use is represented by drawing the unshaded left half of the register file using dashed lines in the ID stage, when it is not being written, and the unshaded right half in dashed lines in the WB stage, when it is not being read.
As before, we assume the register file is **written in the first half** of the clock cycle and is **read during the second half**.




The registers must be wide enough to store all the data corresponding to the lines that go through them.
For example, the IF/ID register must be 64 bits wide, because it must hold both the 32-bit instruction fetched from memory and the incremented 32-bit PC address.
We will expand these registers over the course of this chapter, but for now the other three pipeline registers contain 128, 97, and 64 bits, respectively.

# Pipeline Operation

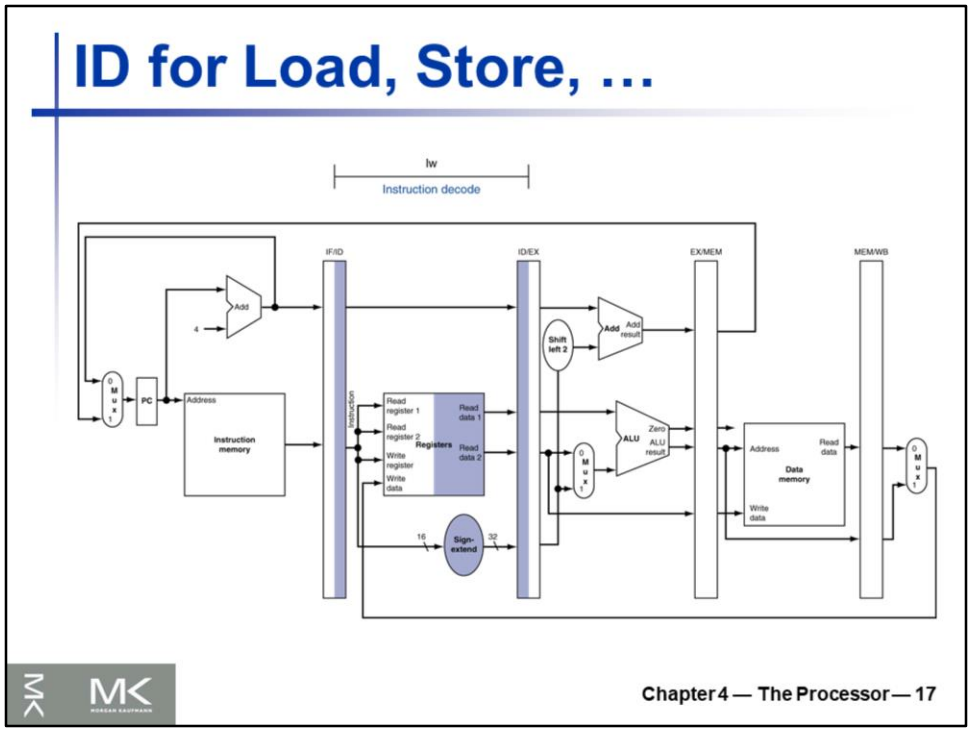
- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store




we highlight the *right half* of registers or memory when they are being *read* and highlight the *left half* when they are being *written*

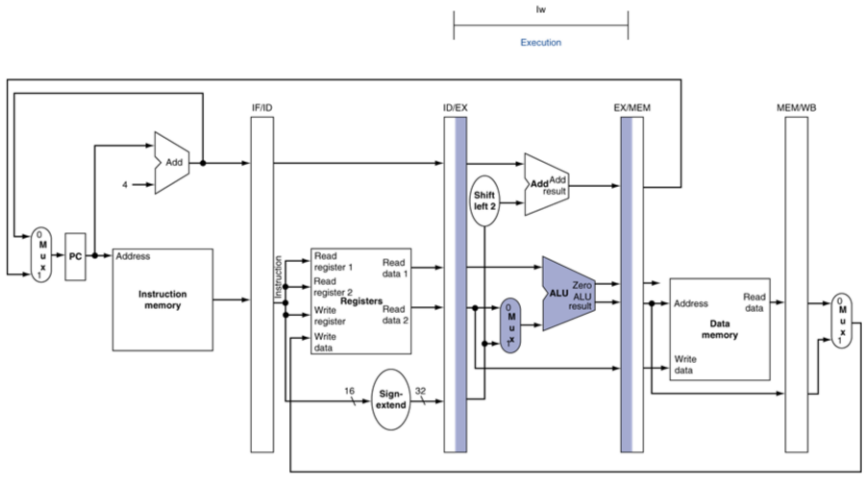
The instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register.

The PC address is incremented by 4 and then written back into the PC to be ready for the **next clock cycle**. This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as beq.



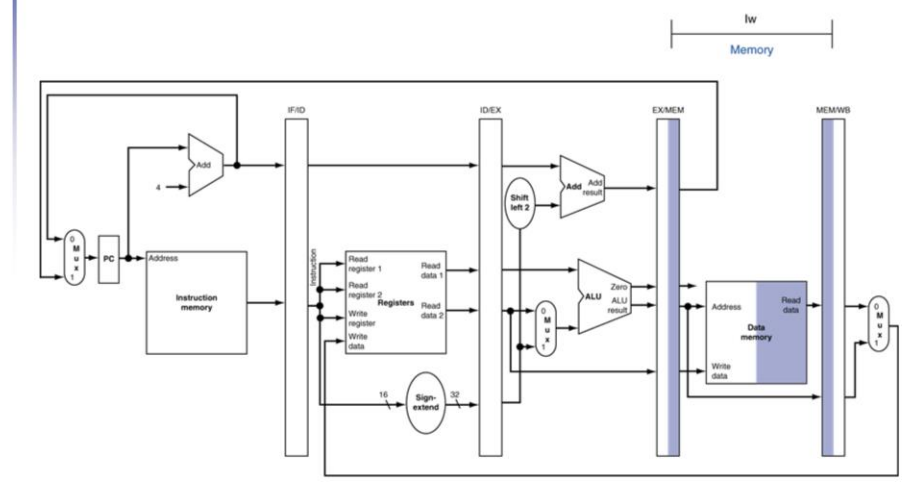

Although the load needs only the top register in stage 2, the processor doesn't know what instruction is being decoded, so it sign-extends the 16-bit constant and reads both registers into the ID/EX pipeline register.
We don't need all three operands, but it simplifies control to keep all three.

# EX for Load

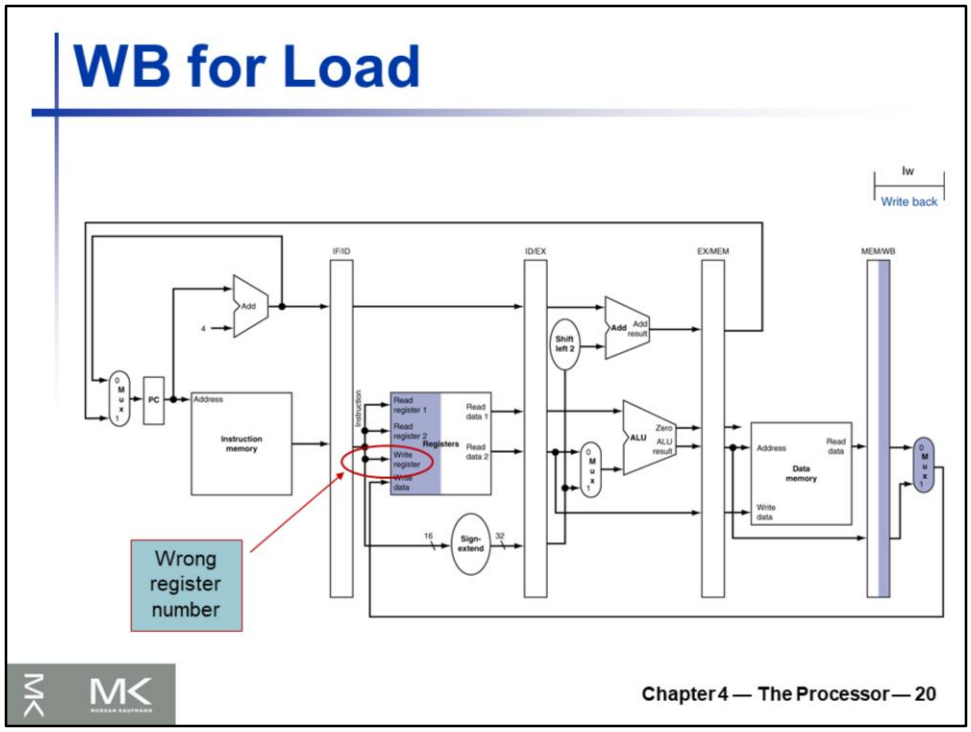



Chapter 4 — The Processor — 18

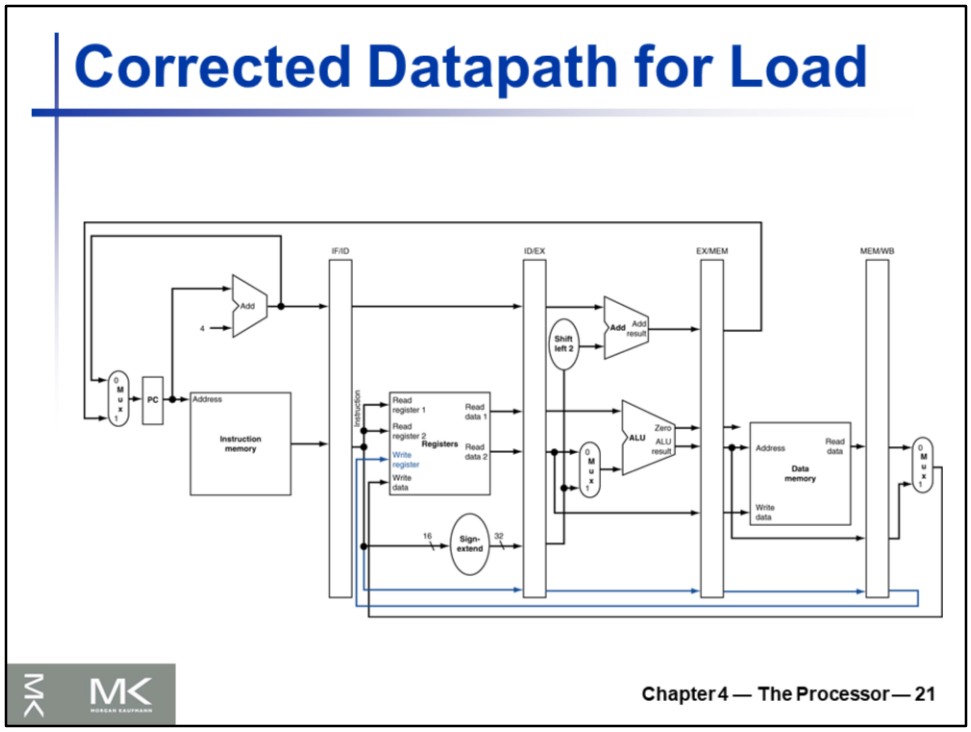
# MEM for Load




Chapter 4 — The Processor — 19

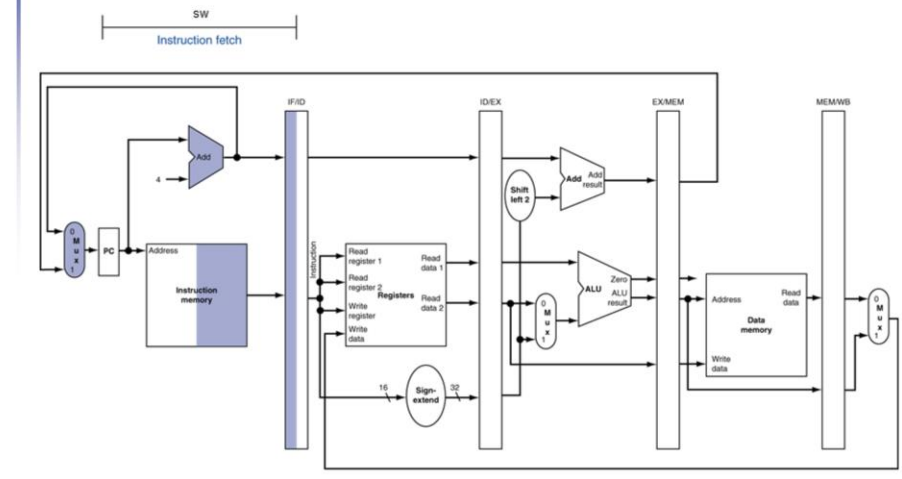



Note: there is a bug in this design that is repaired in Figure 4.41.



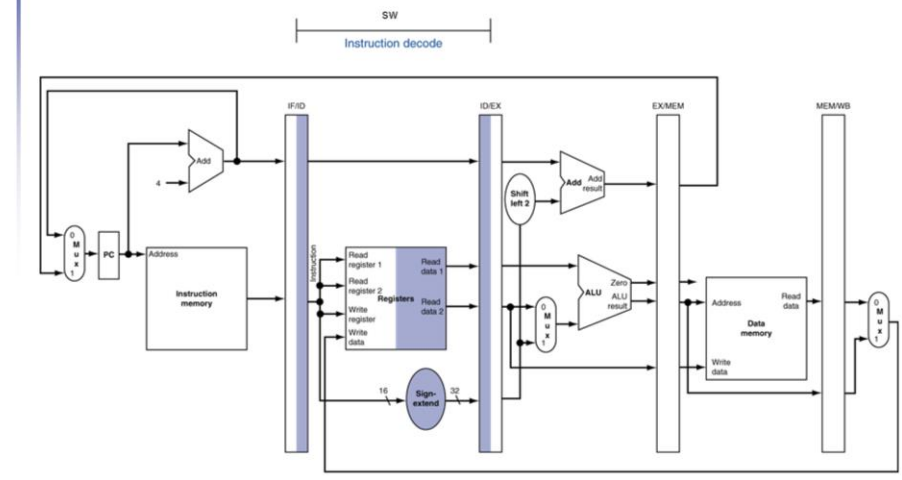

The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding five more bits to the last three pipeline registers. This new path is shown in color.

# IF for Load, Store, …



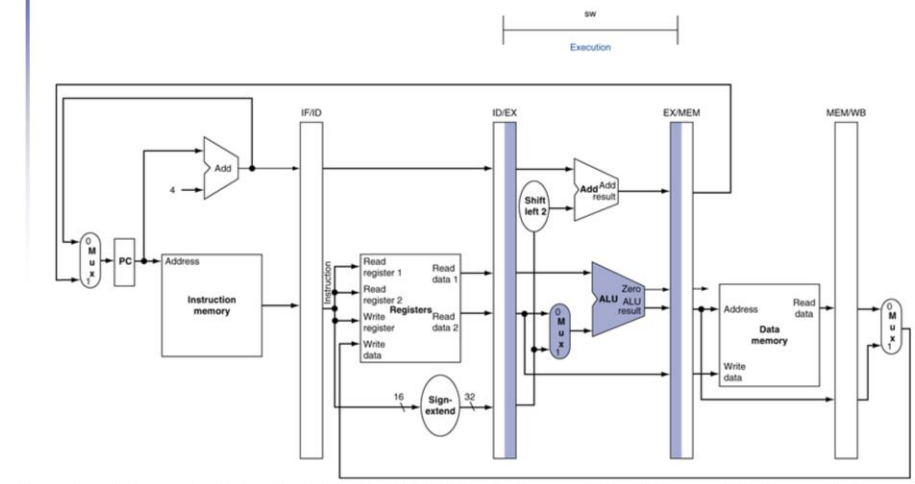

Chapter 4 — The Processor — 22

# ID for Load, Store, …



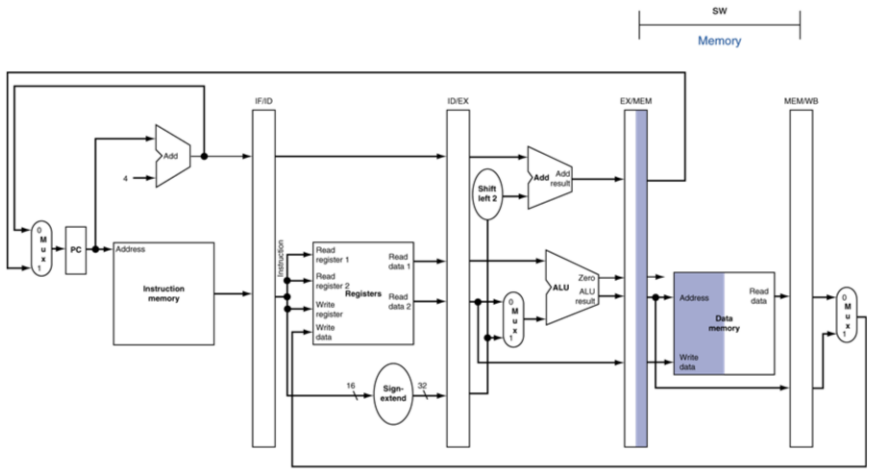

Chapter 4 — The Processor — 23

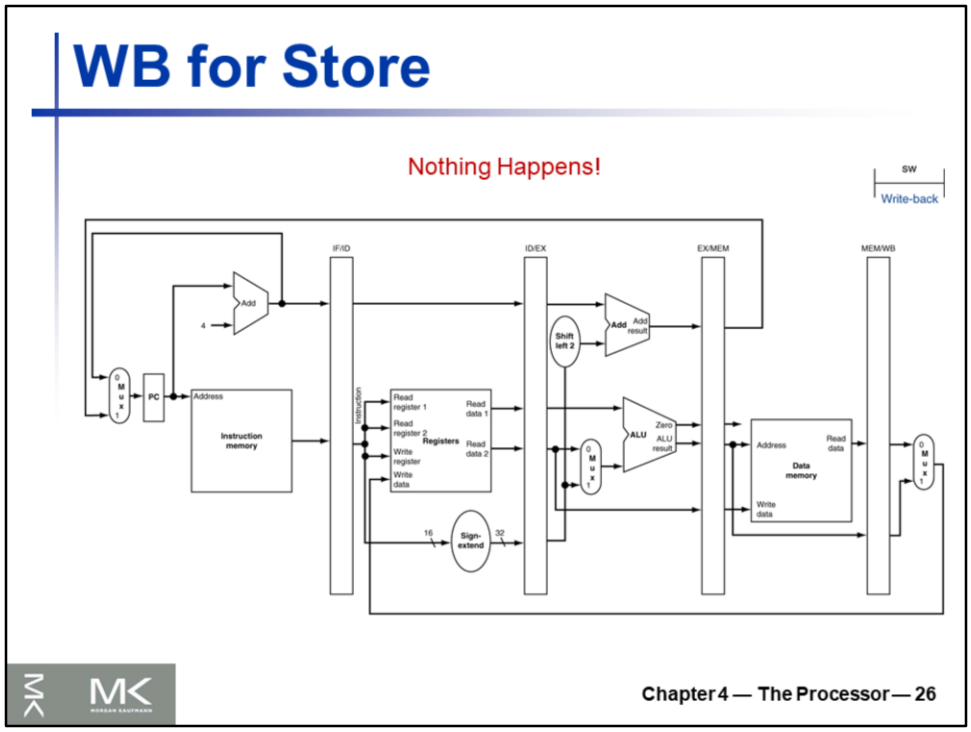
# EX for Store




The 2nd register value is loaded into the EX/MEM pipeline register to be used in the next stage

# MEM for Store

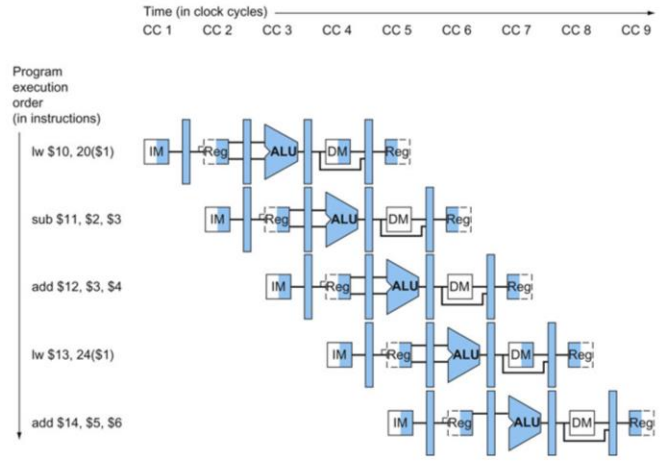





An instruction passes through a stage even if there is nothing to do, because later instructions are already progressing at the maximum rate.

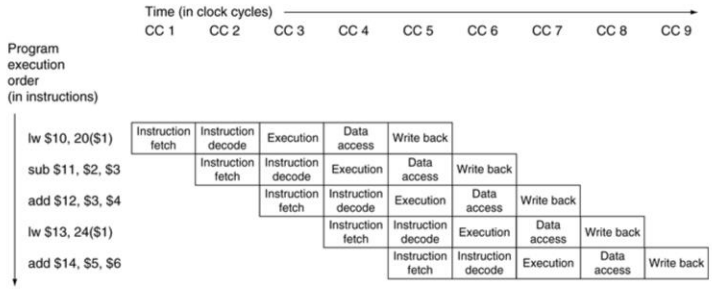
# Multi-Cycle Pipeline Diagram

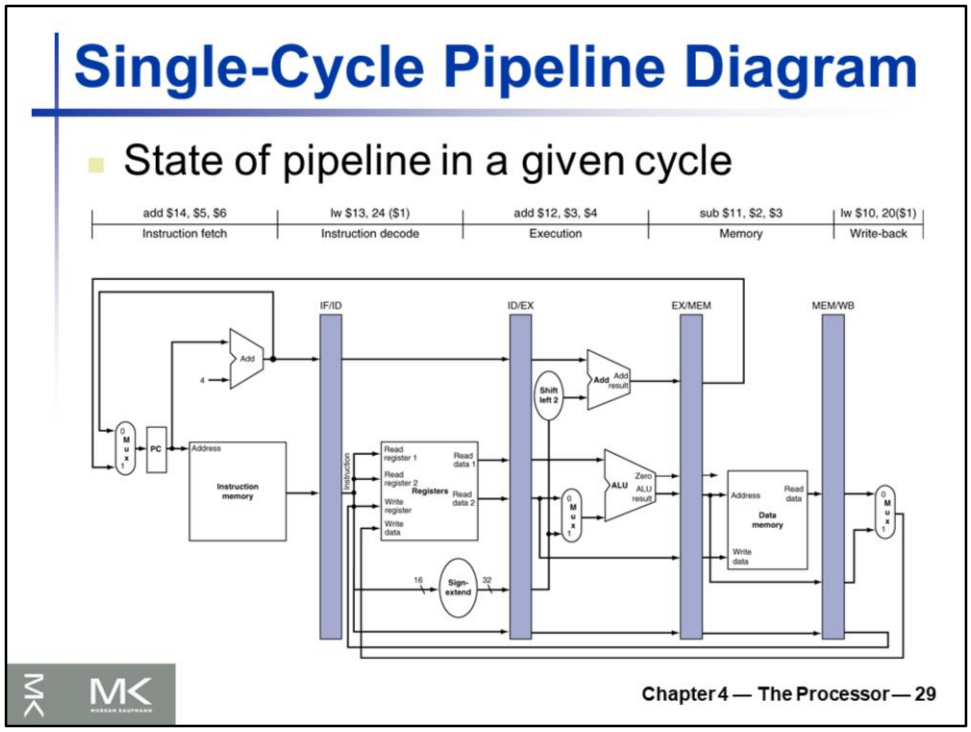
- Form showing resource usage



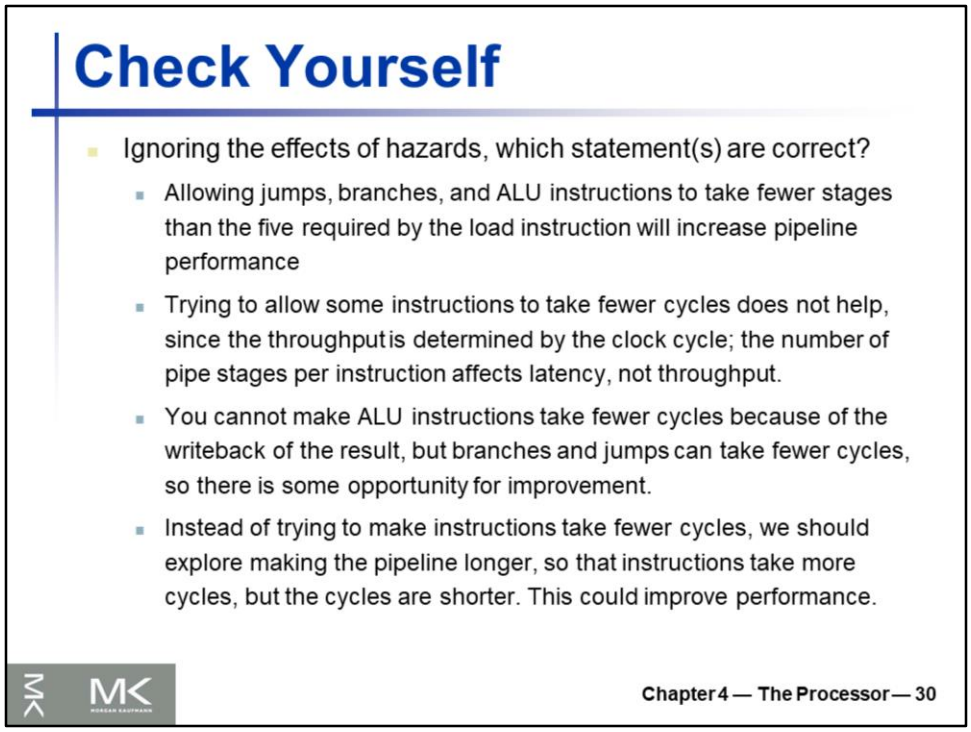
# Multi-Cycle Pipeline Diagram

- Traditional form

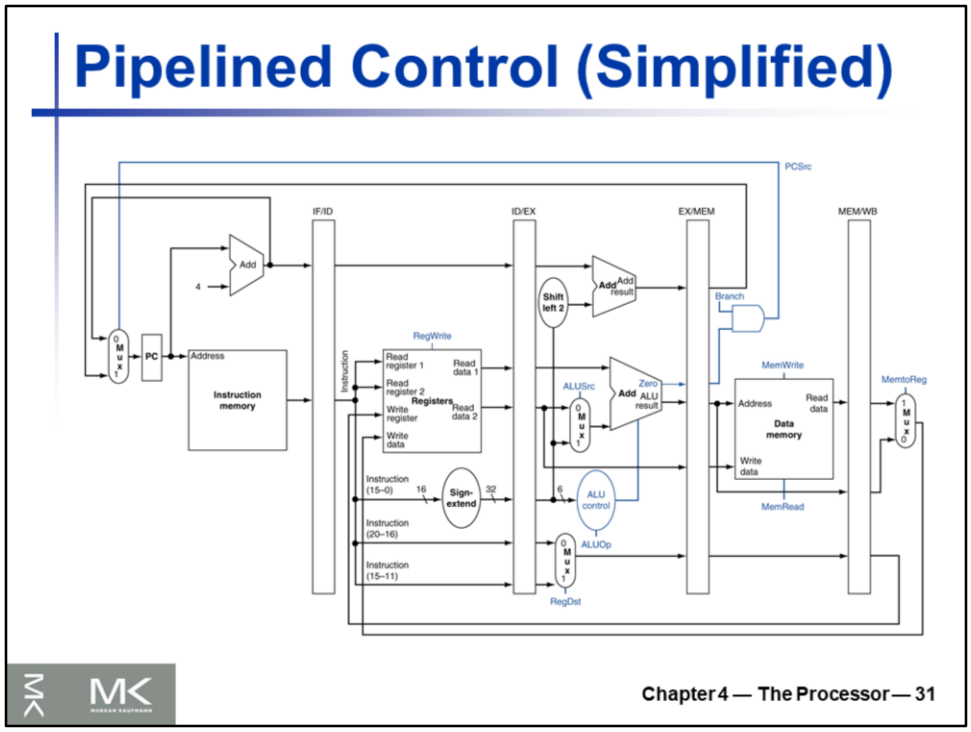




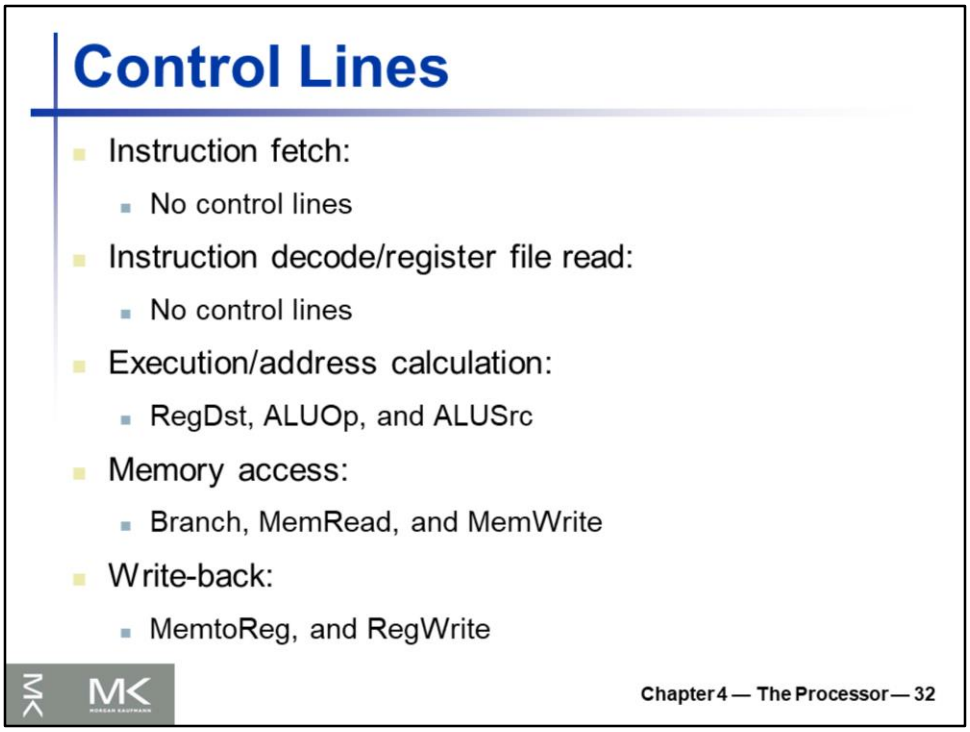

The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in the previous slide.




Statements 2 and 4 are correct; the rest are incorrect.



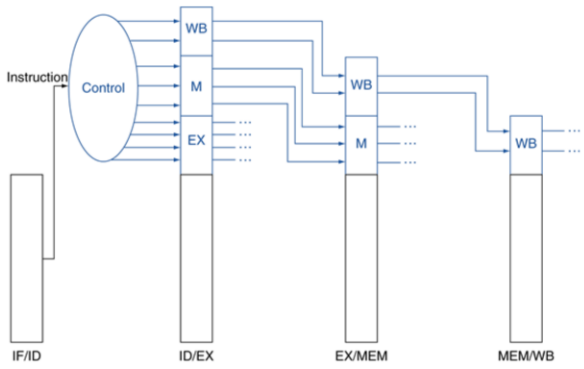

This datapath borrows the control logic for PC source, register destination number, and ALU control from Section 4.4. Note that we now need the 6-bit funct field (function code) of the instruction in the EX stage as input to ALU control, so **these bits must also be included** in the ID/EX pipeline register. Recall that these 6 bits are also the 6 least significant bits of the immediate field in the instruction, so the ID/EX pipeline register can supply them from the immediate field since sign extension leaves these bits unchanged.

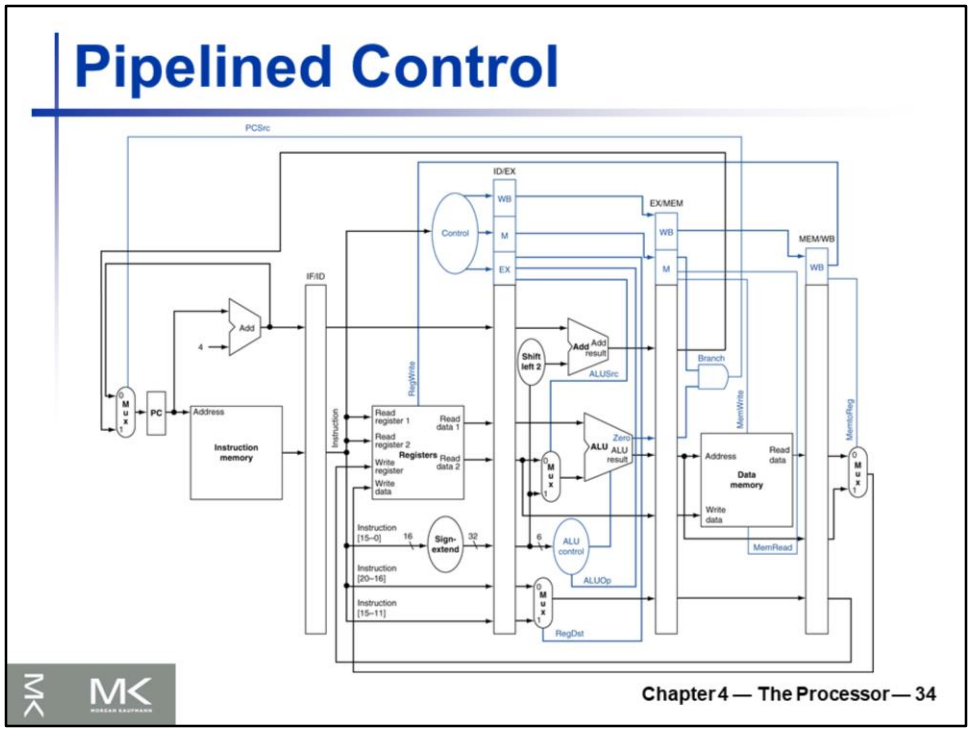



To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with a component active in only a single pipeline stage, we can divide the control lines into five groups according to the pipeline stage.
As was the case for the single-cycle implementation, we assume that the PC is written on each clock cycle, so there is no separate write signal for the PC.
By the same argument, there are no separate write signals for the pipeline registers (IF/ID, ID/EX, EX/MEM, and MEM/WB), since the pipeline registers are also written during each clock cycle.

# Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation

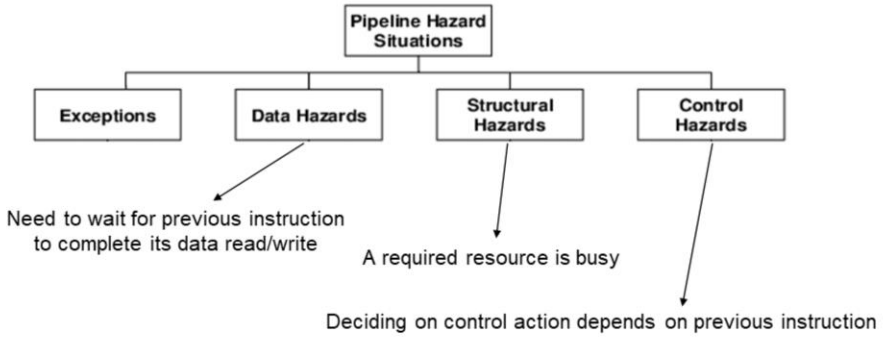


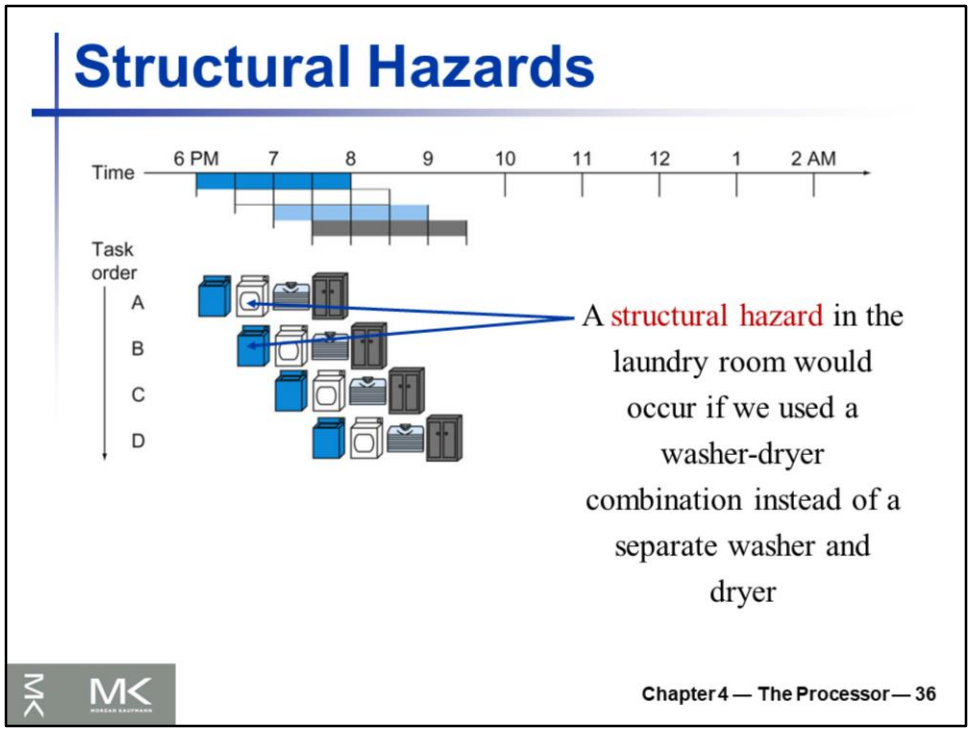



Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information.
Since the control lines start with the EX stage, we can **create** the control information **during instruction decode**.

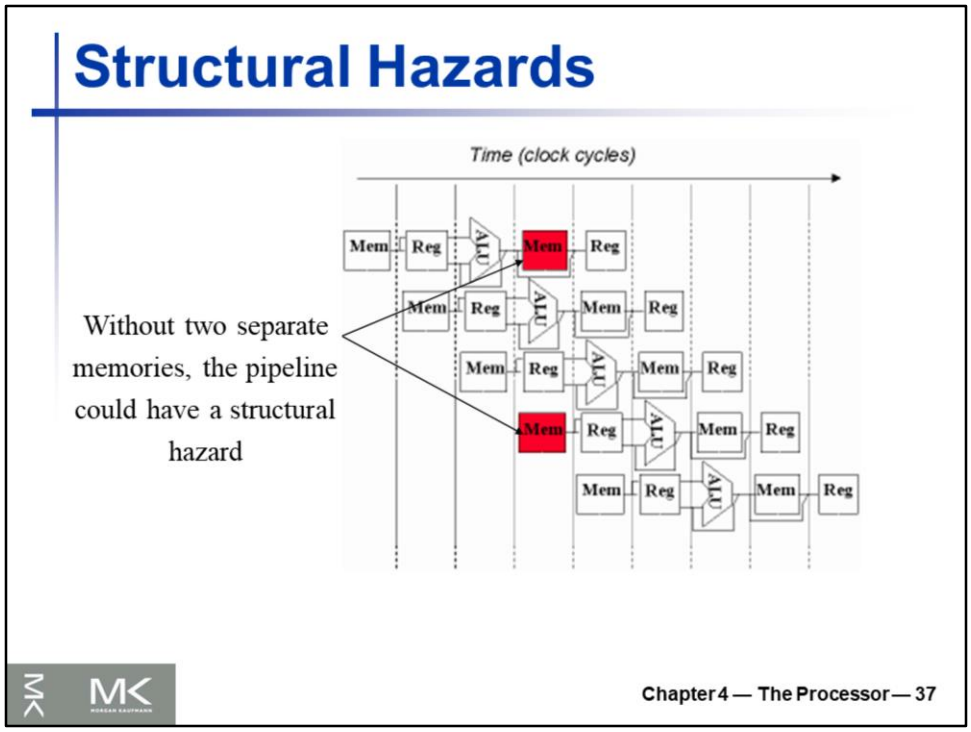
# Hazards

Situations that prevent starting the next instruction in the next cycle

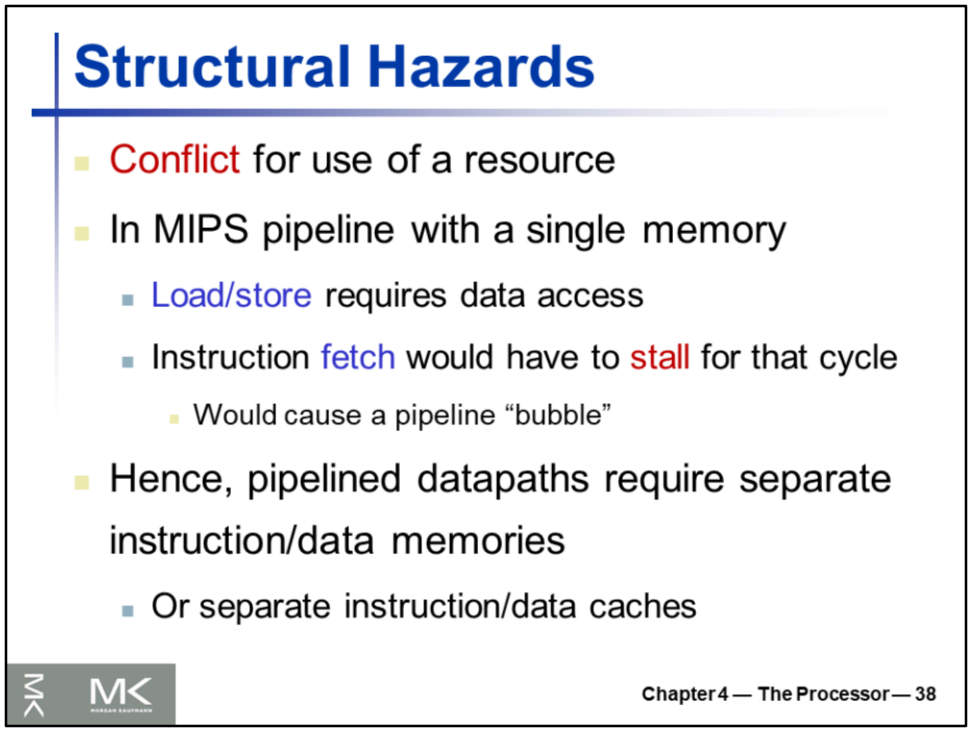




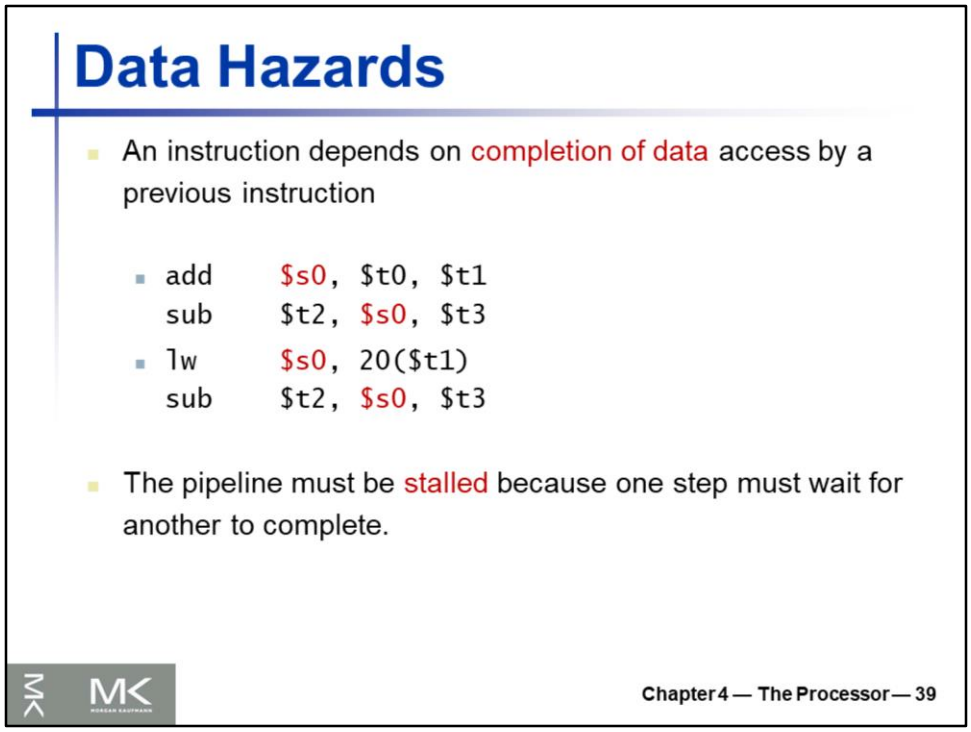

**structural hazard**
When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.




**structural hazard**
When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

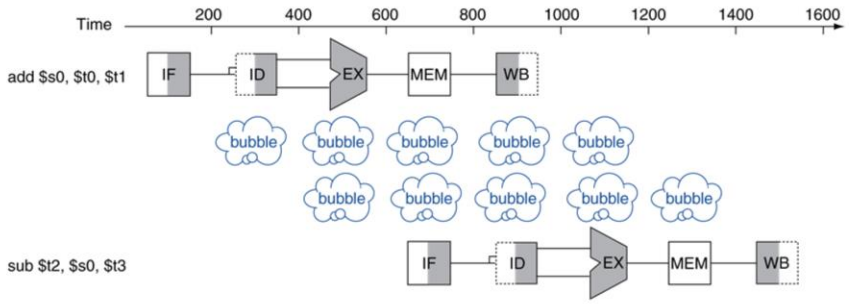



**structural hazard**
When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

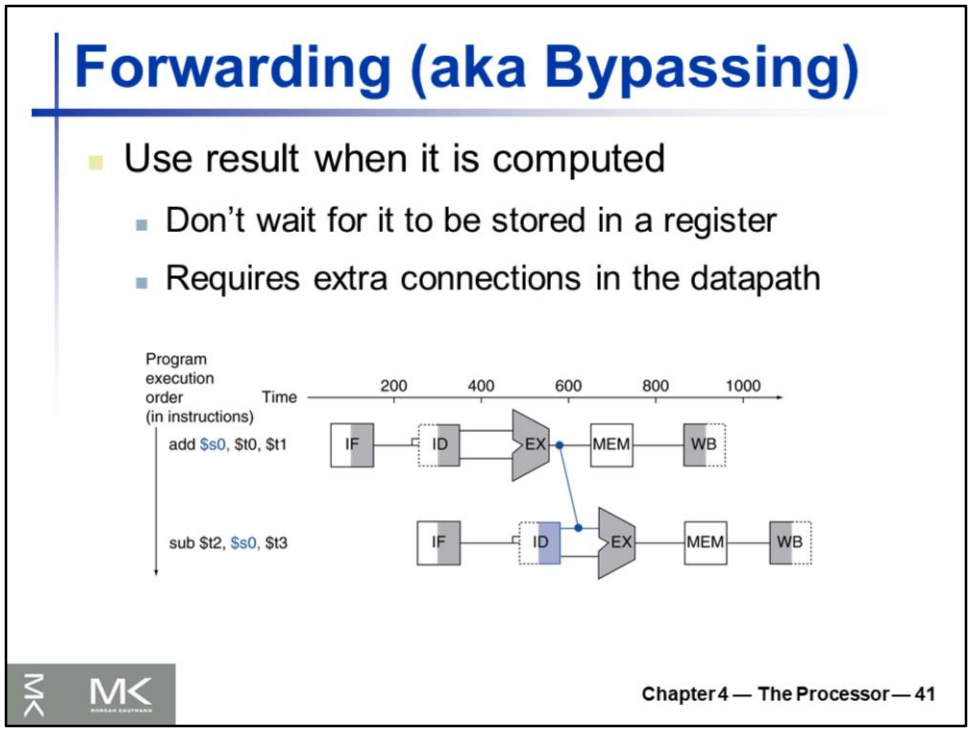



**data hazard**
Also called a **pipeline data hazard**. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

# Data Hazards



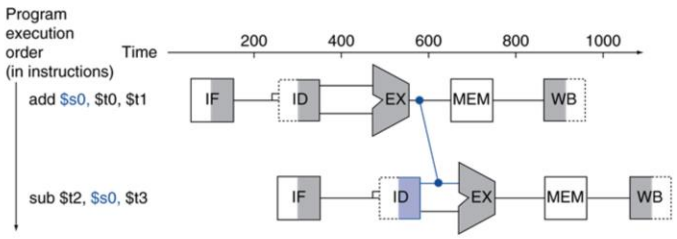

Chapter 4 — The Processor — 40




**forwarding**
Also called **bypassing**. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

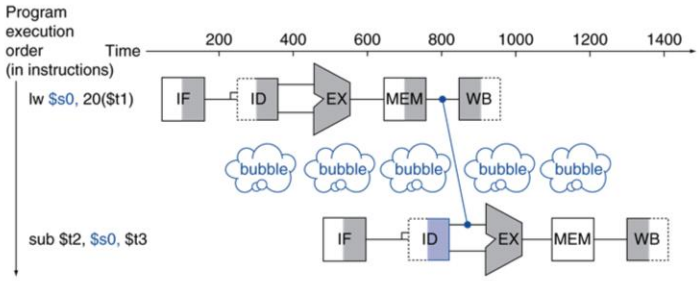
# Forwarding (aka Bypassing)

- "Forwarding" comes from the idea that the result is passed forward from an earlier instruction to a later instruction.
- "Bypassing" comes from passing the result around the register file to the desired unit.



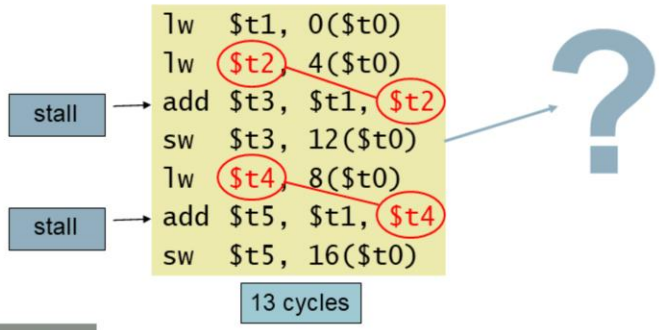
# Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



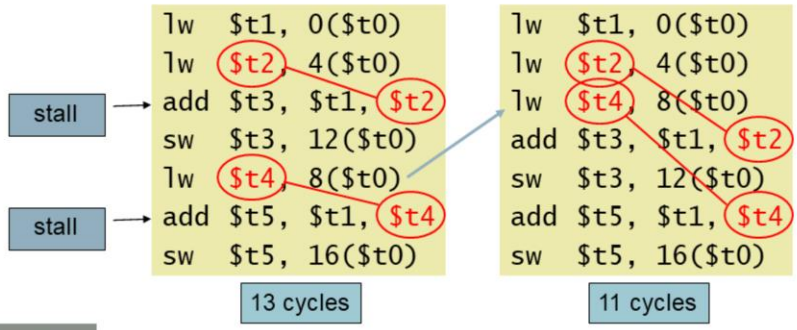
# Code Scheduling to Avoid Stalls

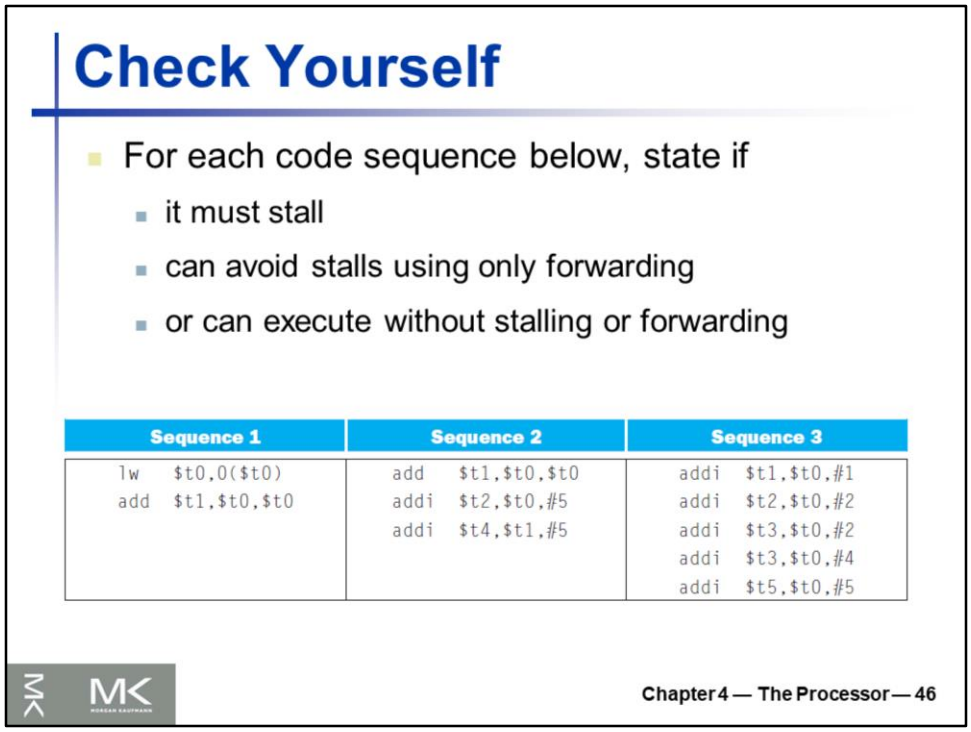
- Reorder code to avoid use of load result in the next instruction
- C code for A=B+E; C=B+F;



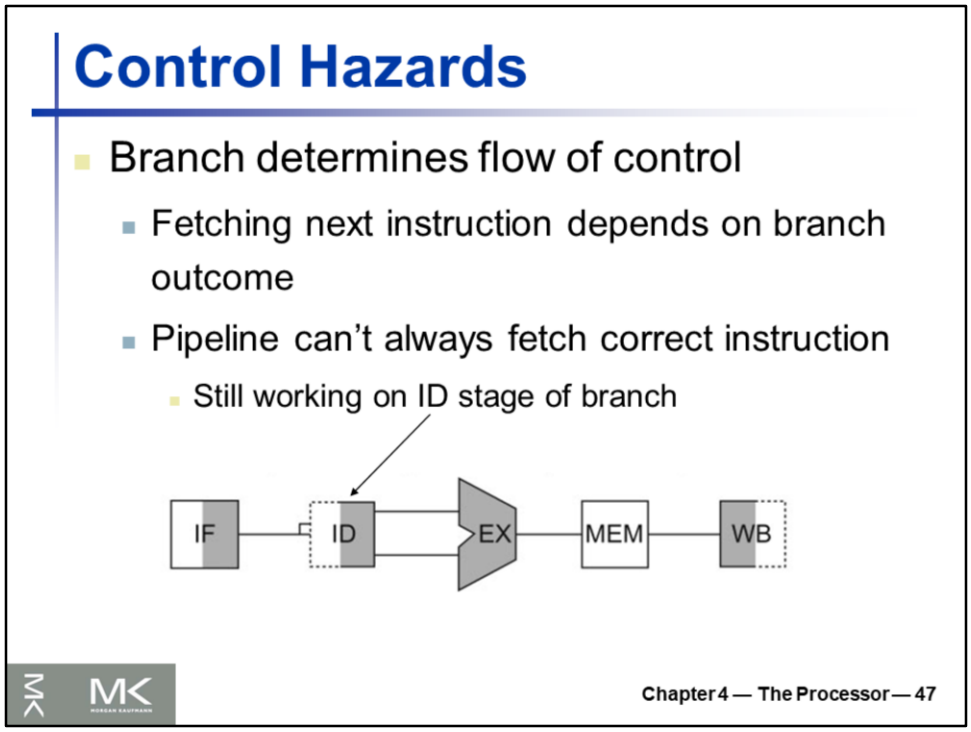
# Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for A=B+E; C=B+F;

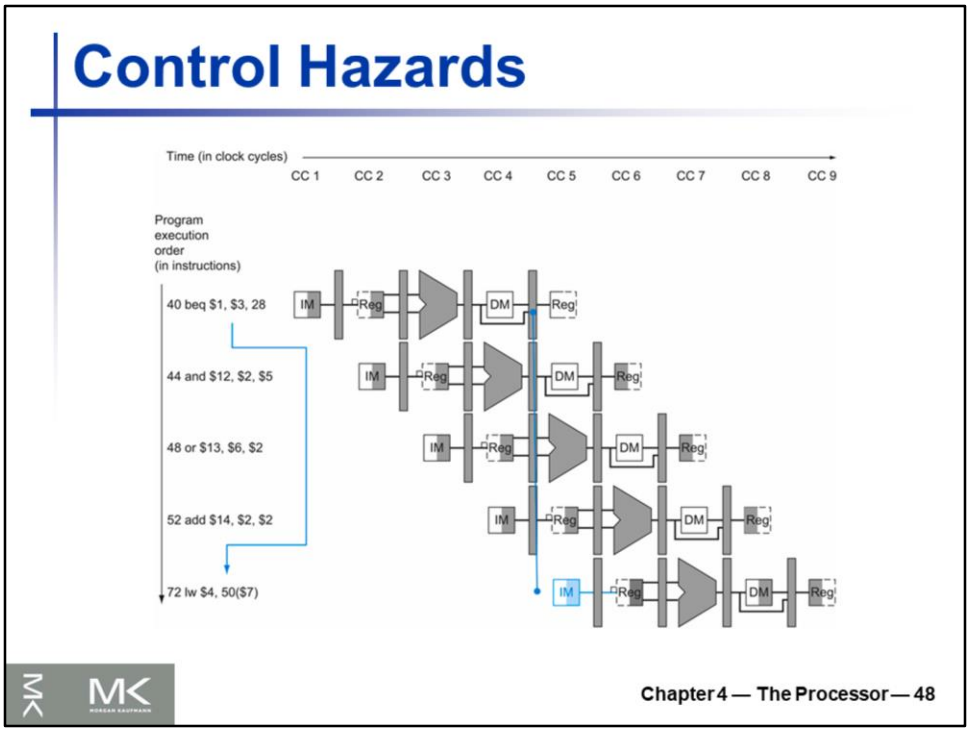





1. Stall on the lw result
2. Bypass the first add result written into $t1
3. No stall or bypass required




**control hazard**
Also called **branch hazard**. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.



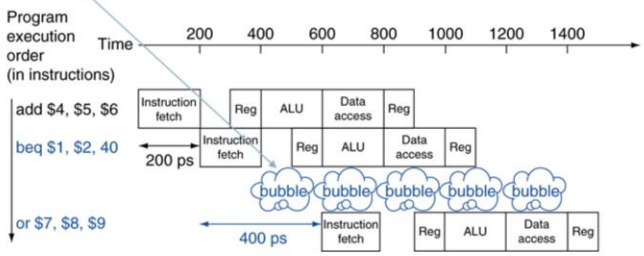

The numbers to the left of the instruction (40, 44, …) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the beq instruction above—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before beq branches to lw at location 72.

# Control Hazards

- In MIPS pipeline
  - one possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows the next instruction
  - Instead we can compare registers and compute target early in the pipeline
    - add hardware to do it in ID stage

# Stall on Branch

- Compare registers and compute target early in the pipeline
- Add hardware to do it in ID stage
- Wait until branch outcome determined before fetching next instruction



# Performance Penalty

- What's the impact on the clock cycles per instruction (CPI) of stalling on branches
  - Assume all other instructions have a CPI of 1.
  - Assume 17% of instructions are branch

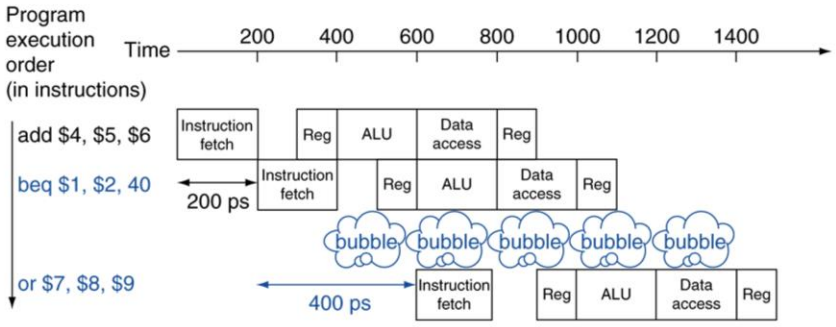
# Performance Penalty

- What's the impact on the clock cycles per instruction (CPI) of stalling on branches
  - Assume all other instructions have a CPI of 1.
  - Assume 17% of instructions are branch
- We would see a CPI of 1.17 and hence a slowdown of 1.17 versus the ideal case

# Control Hazards

- In MIPS pipeline
  - Add hardware to compare registers and compute target in ID stage
  - Still need one stage pipeline stall
- Solutions:
  - Stall on branch
  - Predict
  - Delayed branch

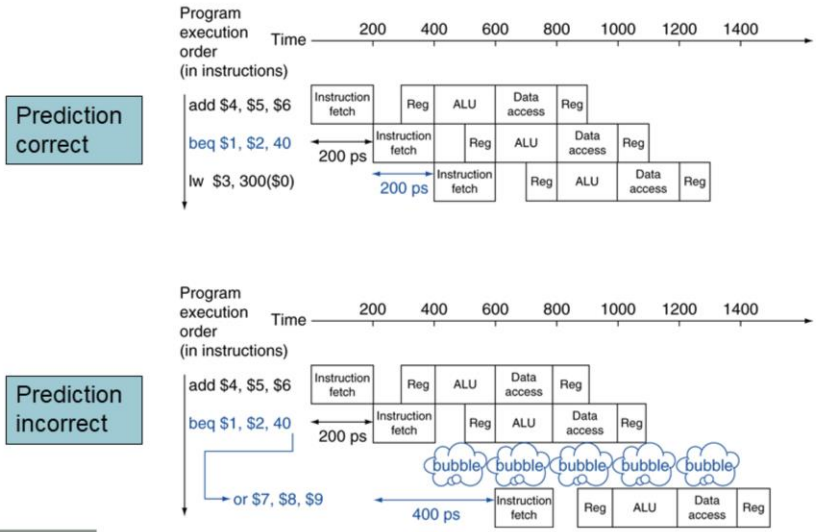
# Stall on Branch



# Branch Prediction

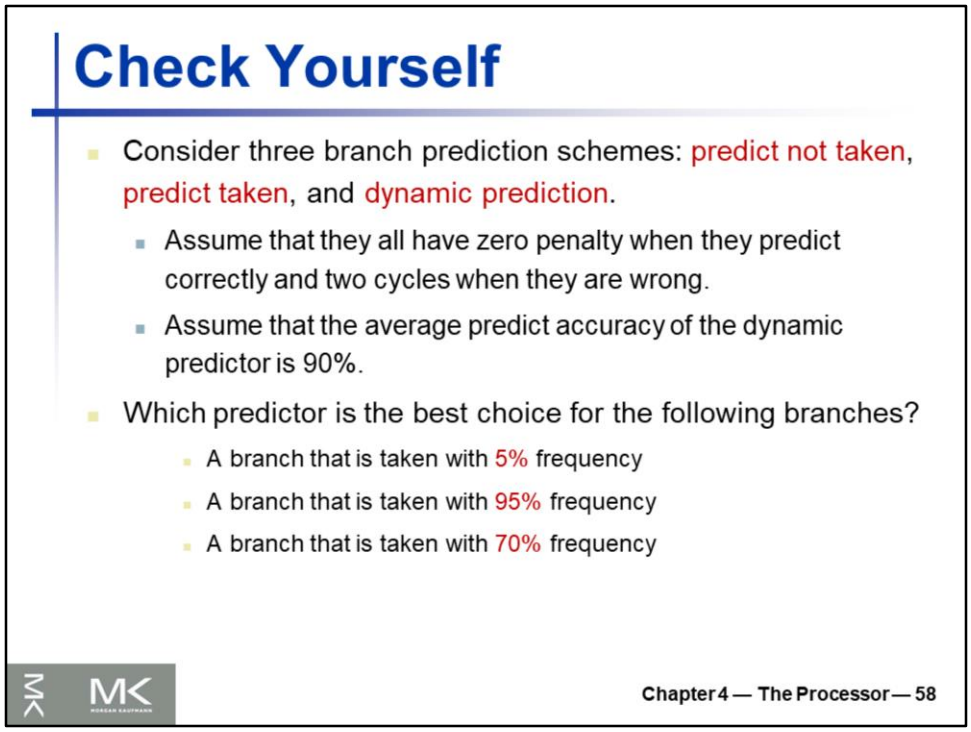
- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

# MIPS with Predict Not Taken



# More-Realistic Branch Prediction

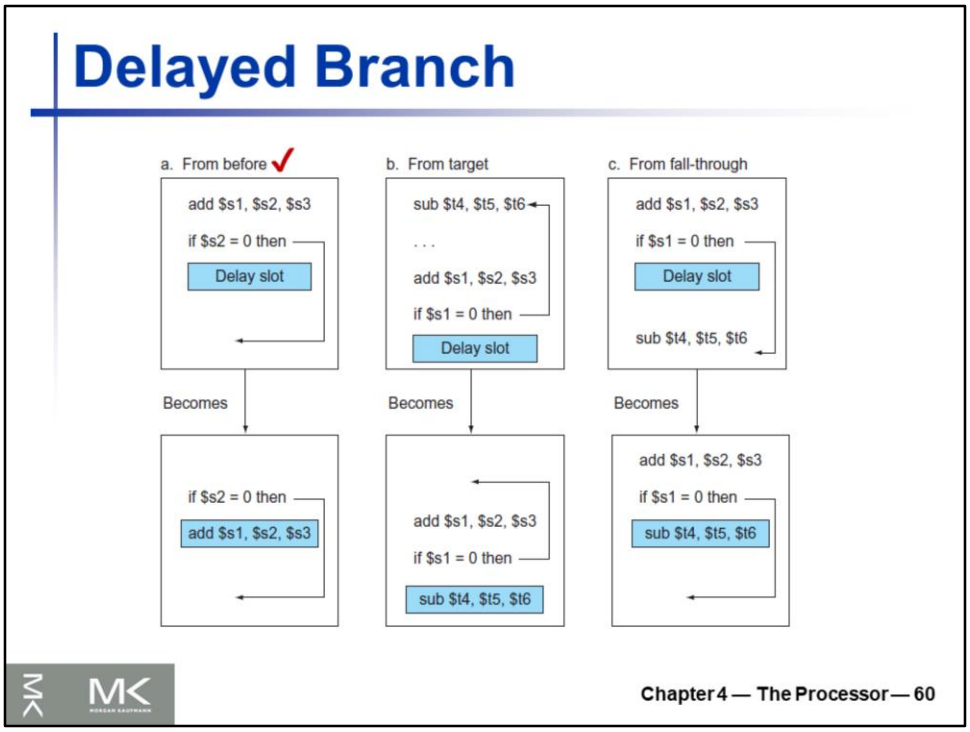
- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history




1. Predict not taken
2. Predict taken
3. Dynamic prediction

# Delayed Branch

- Always execute the next sequential instruction, with the branch taking place after that one instruction delay
  - Place an instruction that is not affected by the branch immediately after the delayed branch
  - A taken branch changes the address of the instruction that follows this safe instruction
- Solution used by the MIPS architecture
- Hidden from the assembly language programmer
  - Performed automatically by the MIPS assembler
- No processor uses a delayed branch of more than one cycle
  - HW-based branch prediction is used for longer branch delays

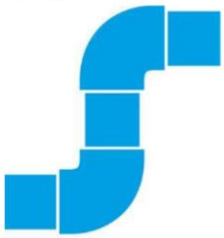



**Scheduling the branch delay slot.**
The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a), the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of $s1 in the branch condition prevents the add instruction (whose destination is $s1) from being moved into the branch delay slot. In (b) the branch delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall-through as in (c).
To make this optimization legal for (b) or (c), it must be OK to execute the sub instruction when the branch goes in the unexpected direction. By “OK” we mean that the work is wasted, but the program will still execute correctly. This is the case, for example, if $t4 were an unused temporary register when the branch goes in the unexpected direction.

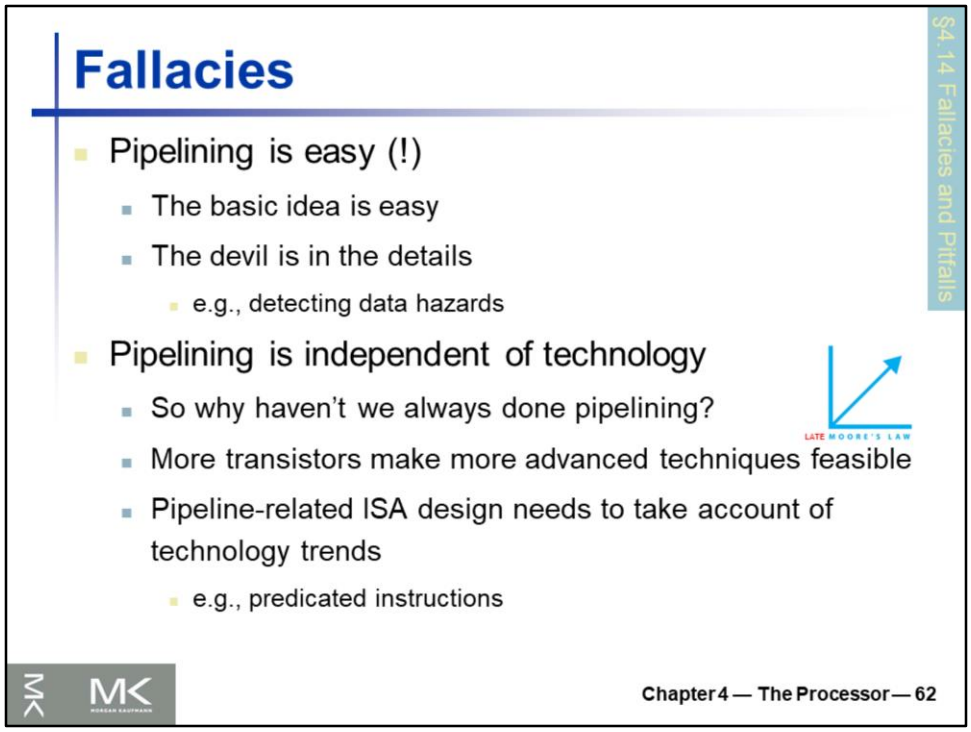
# Concluding Remarks



- Pipelining improves instruction throughput using parallelism
  - More instructions completed per second
  - Latency for each instruction not reduced
- Hazards:
  - structural
  - data
  - control



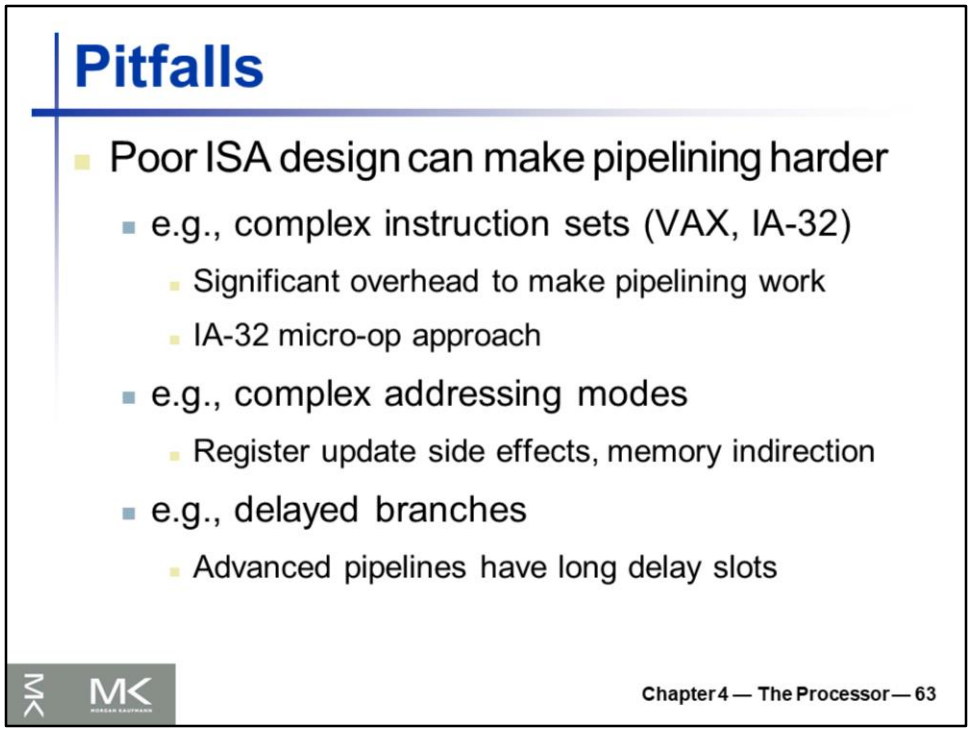
PIPELINING




When the number of transistors on-chip and the speed of transistors made a five-stage pipeline the best solution, then the **delayed branch** was a simple solution to control hazards. With longer pipelines, superscalar execution, and dynamic branch prediction, it is now redundant.

In the early 1990s, **dynamic pipeline scheduling** took too many resources and was not required for high performance, but as transistor budgets continued to double due to Moore's Law logic became much faster than memory, then multiple functional units and dynamic pipelining made more sense.

Today, concerns about **power** are leading to less aggressive designs.




Sophisticated addressing modes can lead to different sorts of problems:
Addressing modes that update registers (e.g. auto-inc/dec) complicate hazard detection.
Other addressing modes that require multiple memory accesses substantially complicate pipeline control and make it difficult to keep the pipeline flowing smoothly.

# Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation