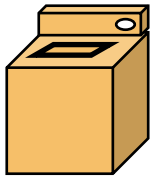
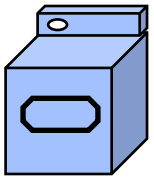
Chapter -6

Enhancing Performance with Pipelining

Pipelining is Natural: Assembly Line!

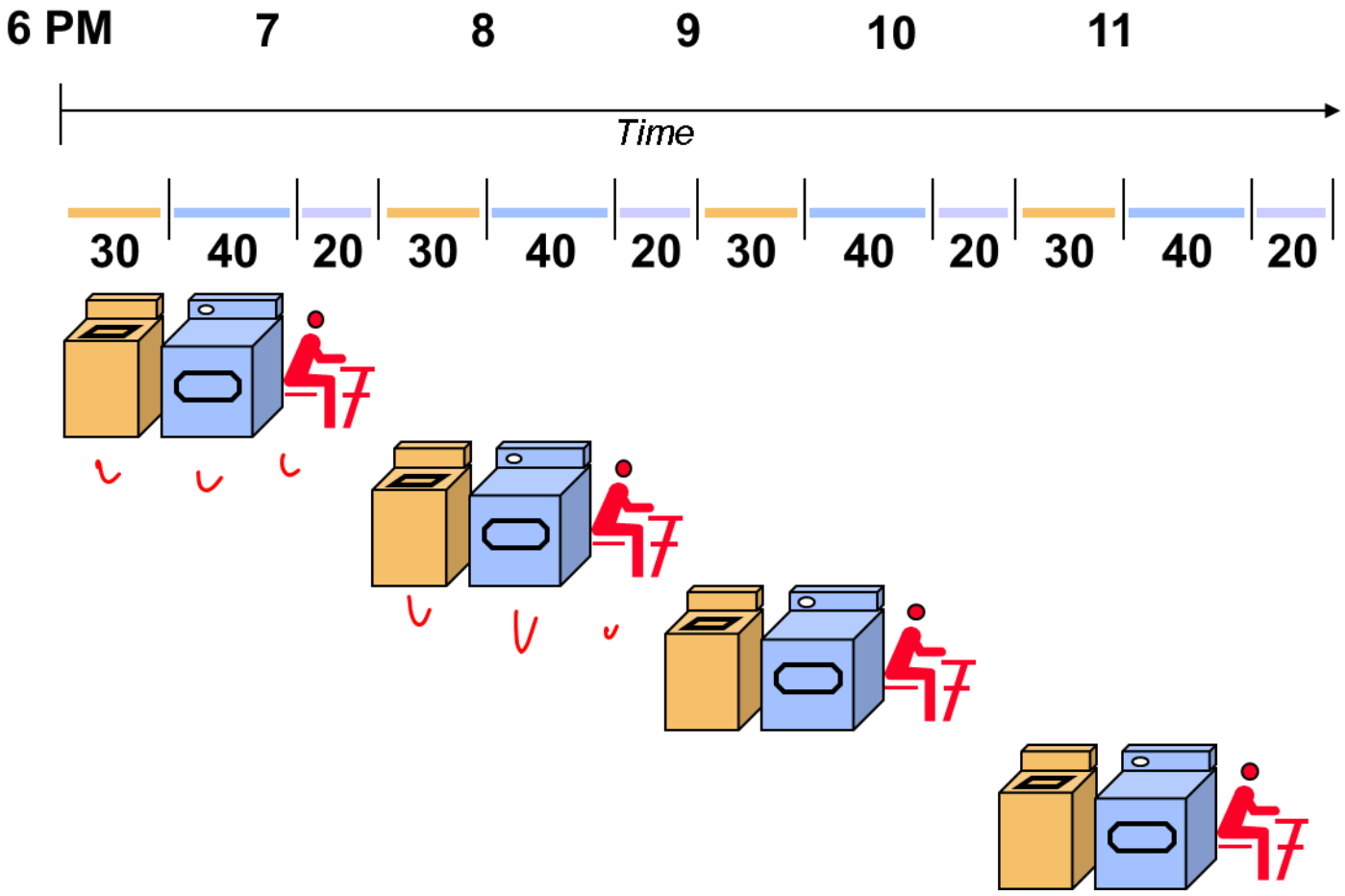
- Assuming you've got:
  - One washer (takes 30 minutes)
  - One drier (takes 40 minutes)
  - One "folder" (takes 20 minutes)





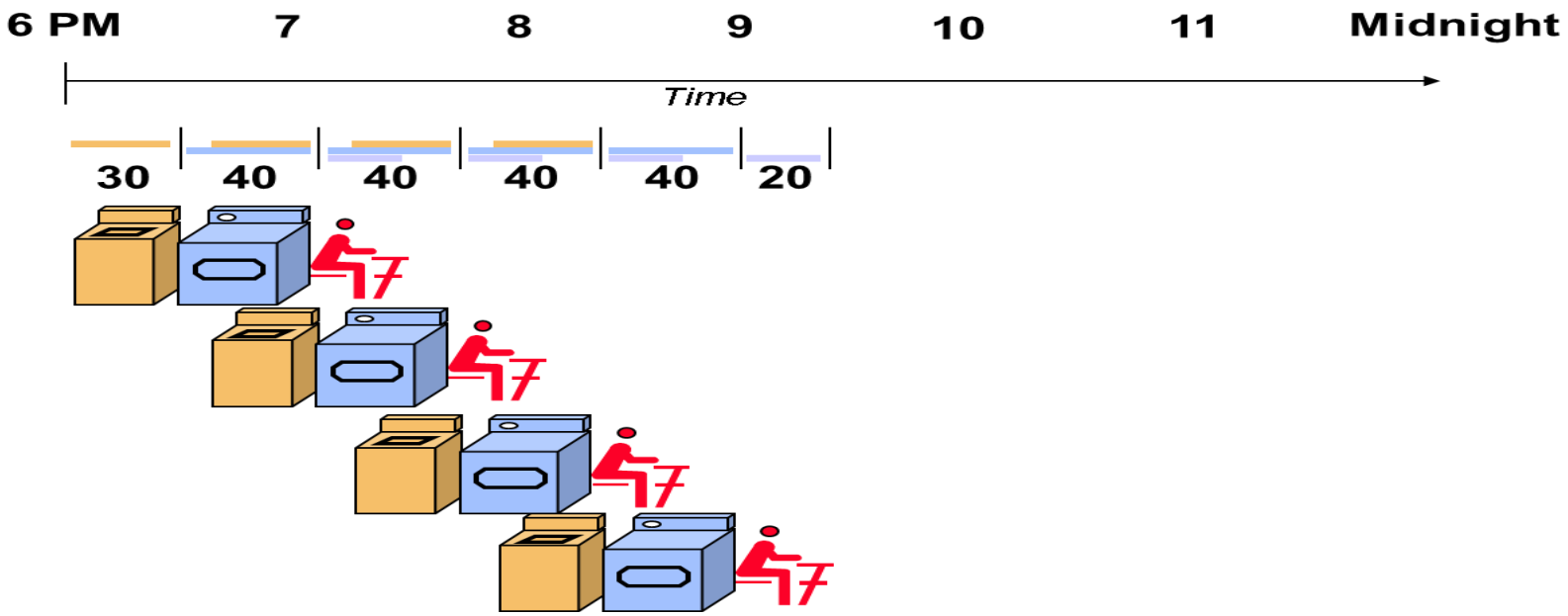
- It takes 90 minutes to wash, dry, and fold 1 load of laundry.
  - How long does 4 loads take?

The slow way(Non-pipelined)



Laundry Pipelining

- Start each load as soon as possible
  - Overlap loads



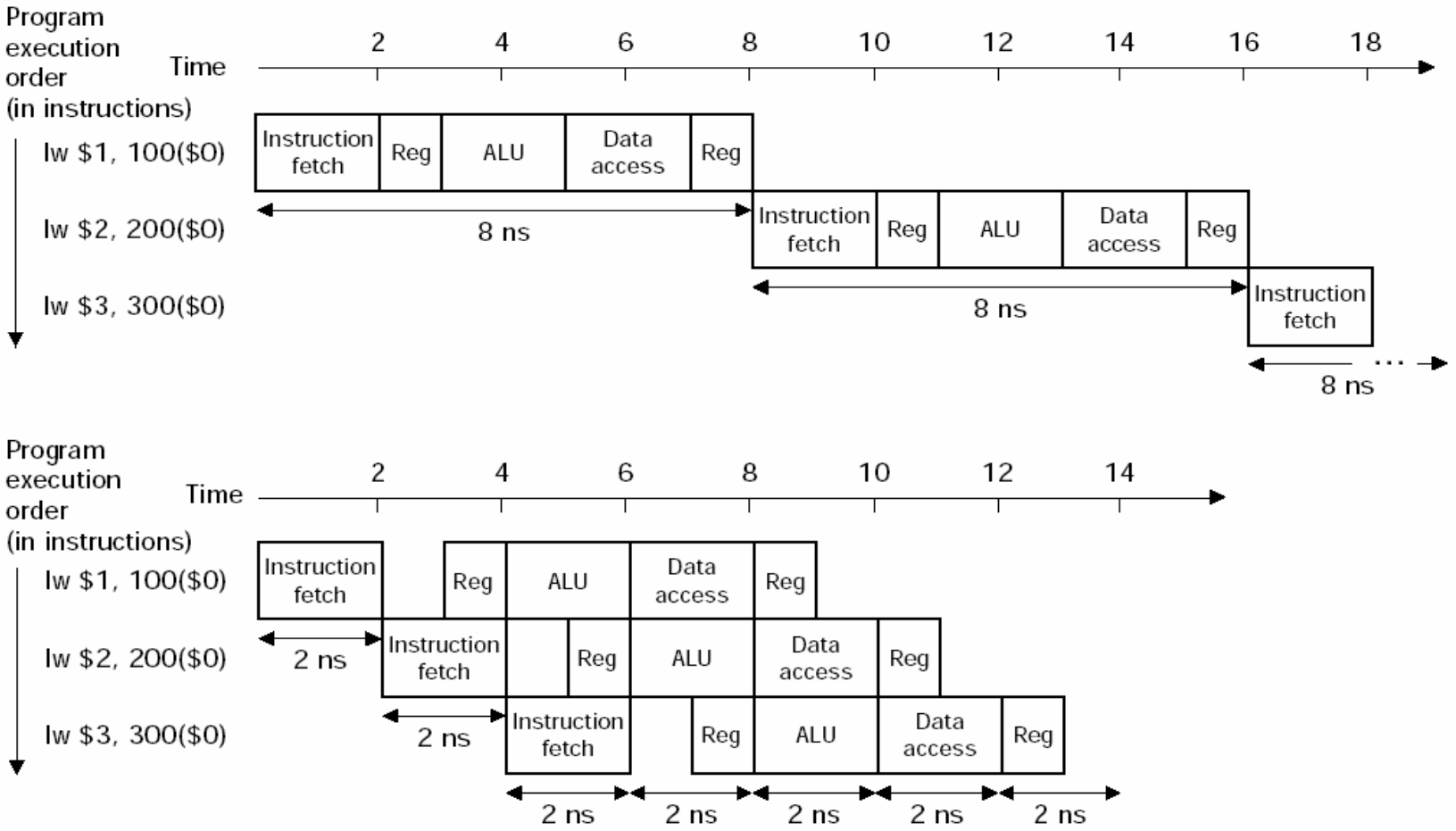
Pipelining

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

MIPS

- Pipe Stages == The Five Execution Steps
- 1. Instruction Fetch
- 2. Instruction Decode and Register Fetch
- 3. Execution, Memory Address Computation, or Branch Completion
- 4. Memory Access or R-type instruction completion
- 5. Write-Back Step

Pipelining in MIPS



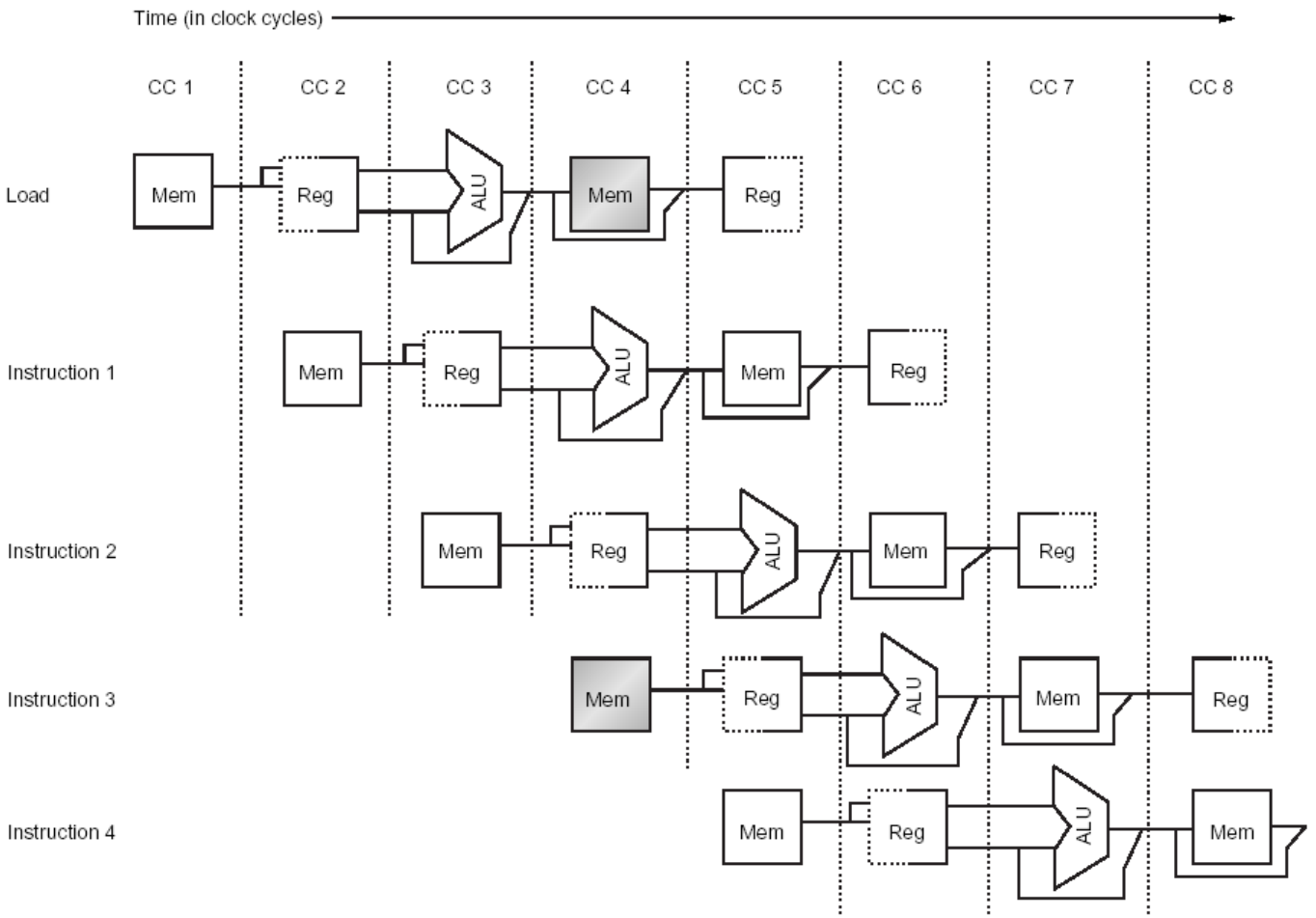
Pipeline Hazards

- There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards.
- Three types of hazards
  - ***Structural hazards***
    - Arise from resource conflicts when the hardware can't support all possible combinations of overlapping instructions
  - ***Data hazards***
    - Arise when an instruction depends on the results of a previous instruction in a way that is exposed by overlapping of instruction in pipeline
  - ***Control hazards***
    - Arise from the pipelining of branches and other instructions that change the PC (Program Counter)

Structural Hazarads

- If certain combination of instructions can't be accommodated because of resource conflicts, the machine is said to have a *structural hazard*
- It can be generated by:
  - Some functional unit is not fully pipelined
  - Some resources has not been duplicated enough to allow all the combinations in the pipeline to execute
  - For example: a machine may have only one register file write port, but under certain conditions, the pipeline might want to perform two writes in one clock cycle – this will generate structural hazard

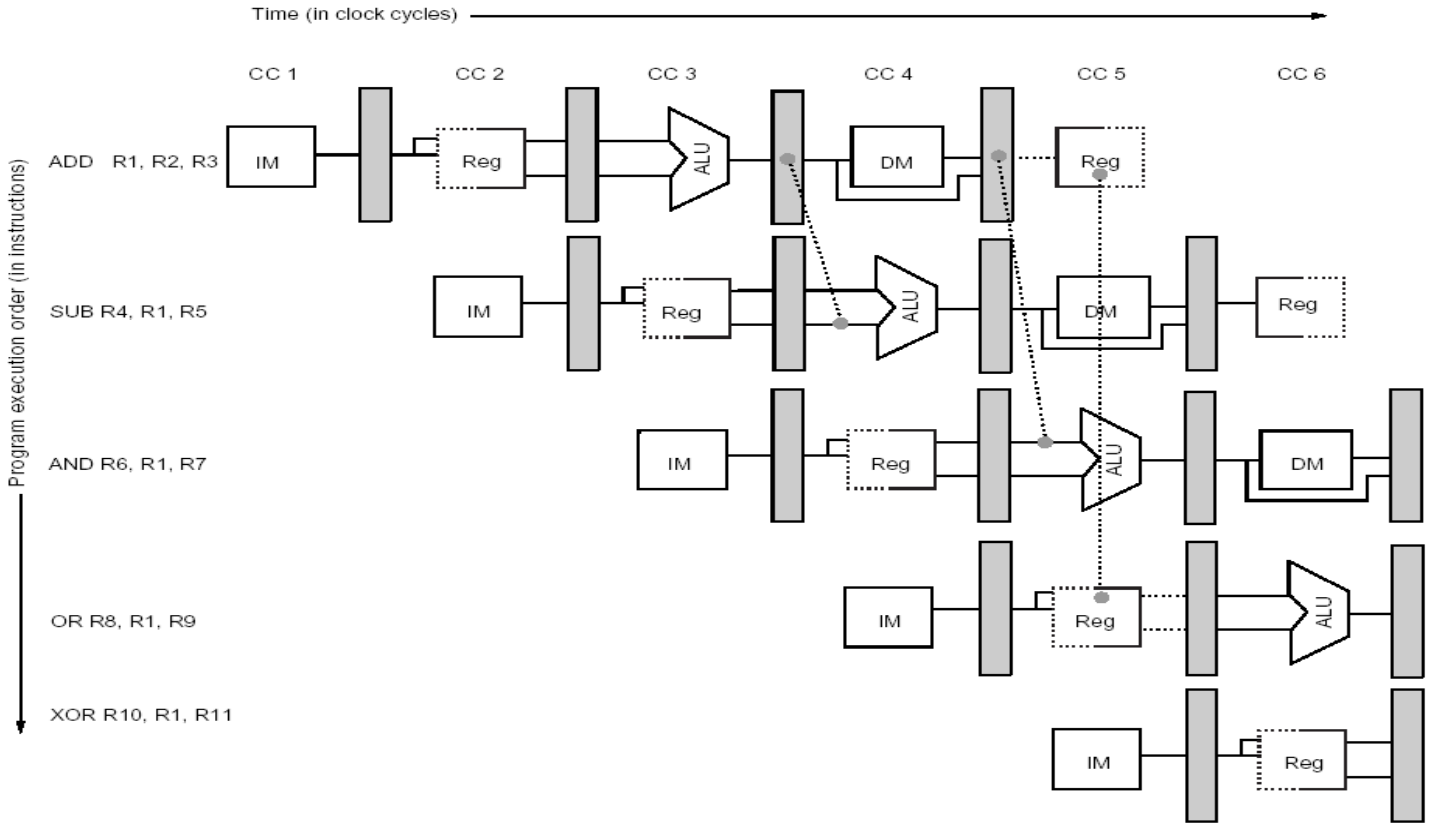
Structural Hazards(Cont)



Data Hazards

- An instruction depends on the results of a previous instruction still in the pipeline.
- Consider the execution of following instructions, on our pipelined example processor:
  - ADD R1, R2, R3
  - SUB R4, R1, R5
  - AND R6, R1, R7
  - OR R8, R1, R9
  - XOR R10, R1, R11

Data Hazards(Cont..)



Eliminate the stalls for the hazard involving SUB and AND instructions using a technique called ***forwarding***

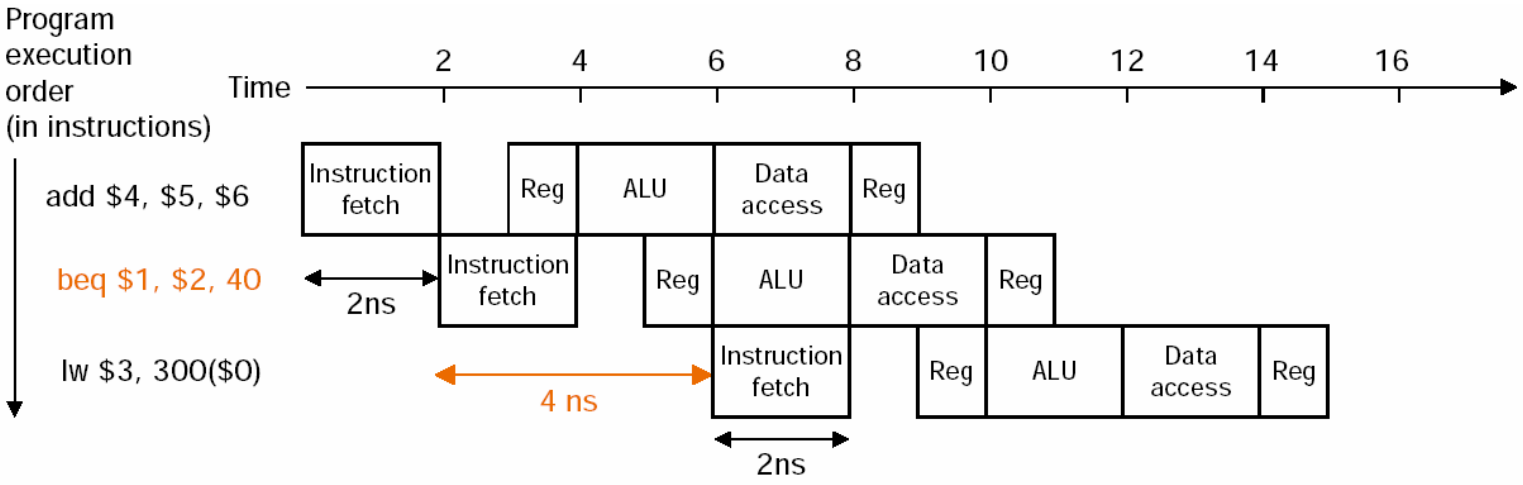
Control Hazards

- Can cause a greater performance loss than the data hazards
- When a branch is executed it may or it may not change the PC (to other value than its value + 4)
  - If a branch is changing the PC to its target address, than it is a ***taken*** branch
  - If a branch doesn't change the PC to its target address, than it is a ***not taken*** branch
- If instruction i is a taken branch, than the value of PC will not change until the end MEM stage of the instruction execution in the pipeline
  - A simple method to deal with branches is to stall the pipe as soon as we detect a branch until we know the result of the branch

Control Hazards

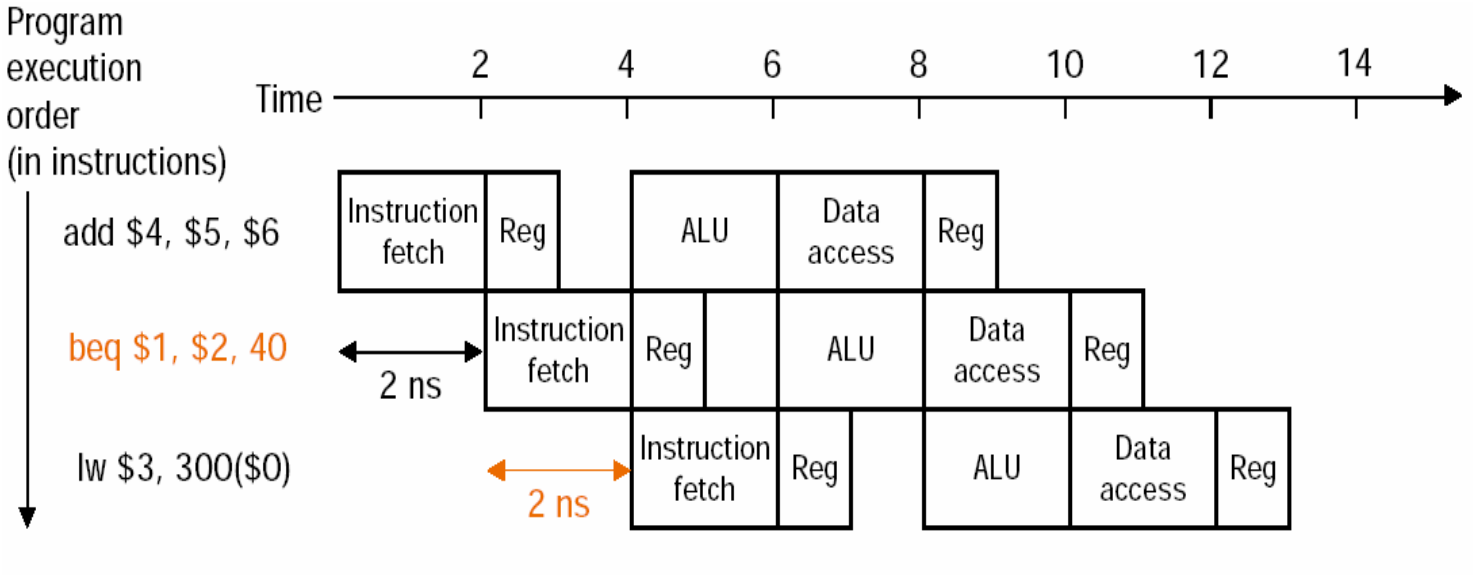
- What is the next instruction?
- Branch instructions take time to compute this.

Solution 1: Stall



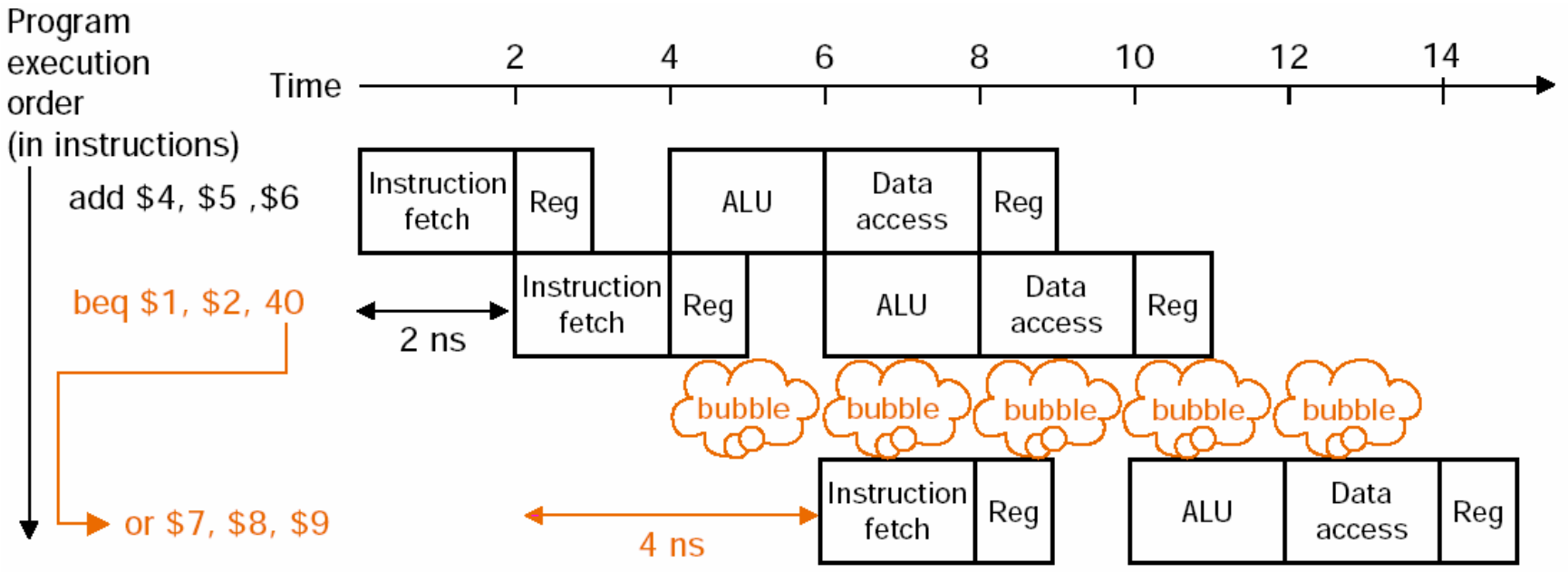
Control Hazards

- Solution 2: Predict the Branch Target



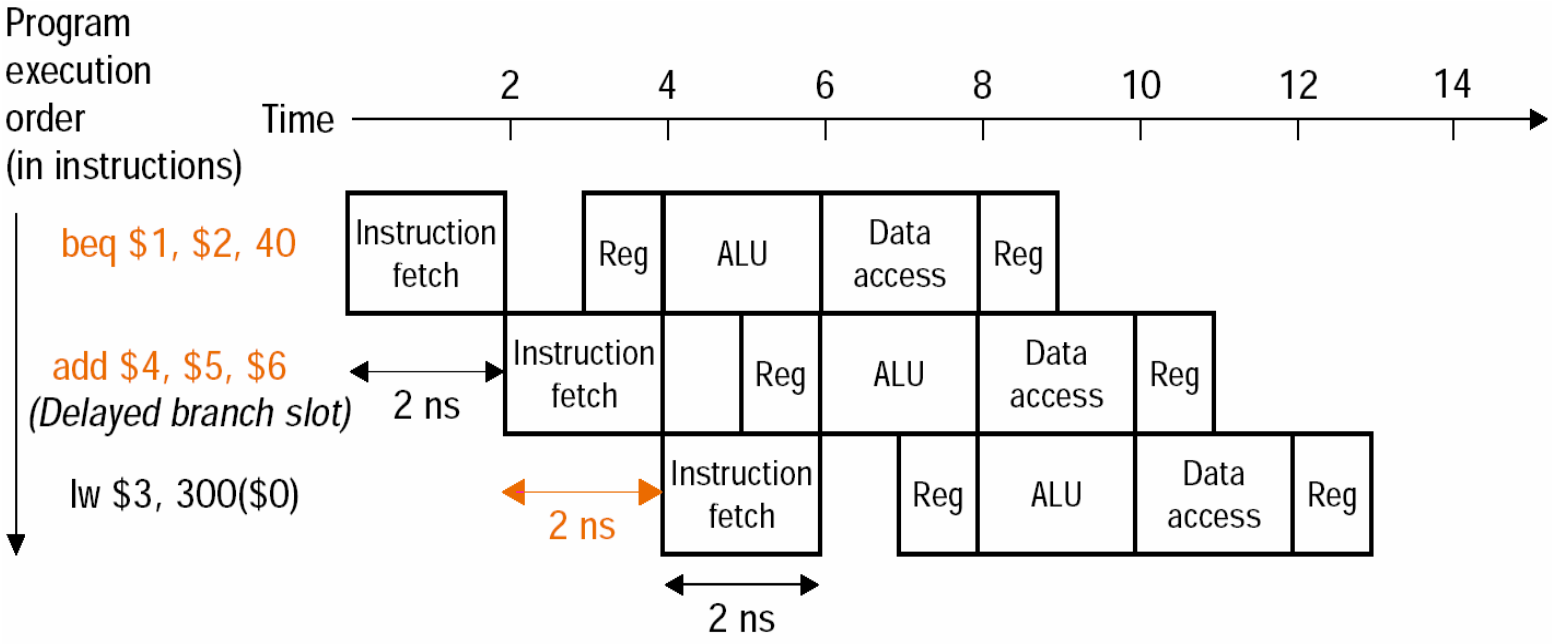
Control Hazards

Solution 2: (Mis)Predict the Branch Target



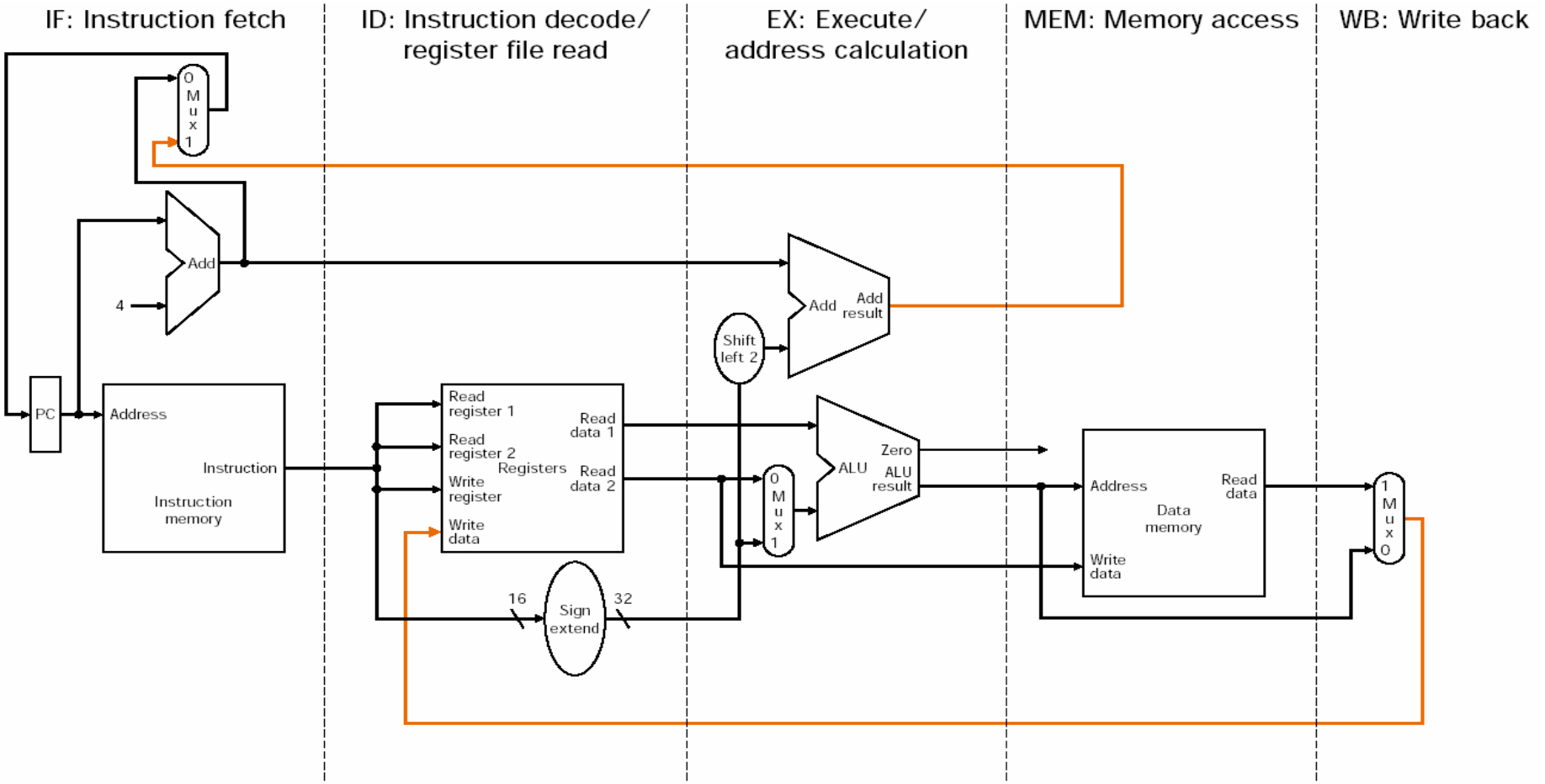
Solution 3: Delayed Decision (Used in MIPS)

- Solution 3: Delayed Decision (Used in MIPS)

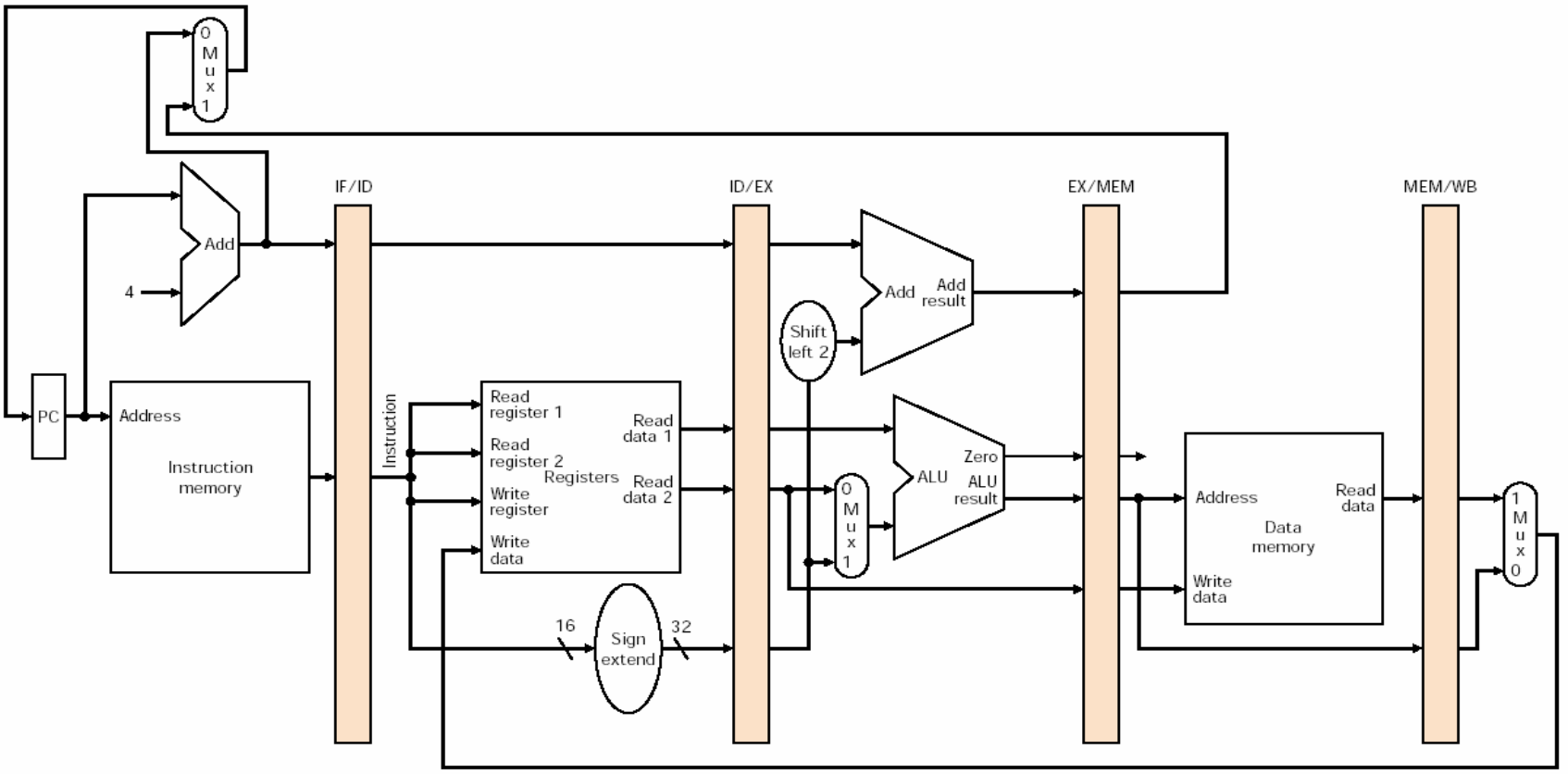


A Pipelined Datapath()

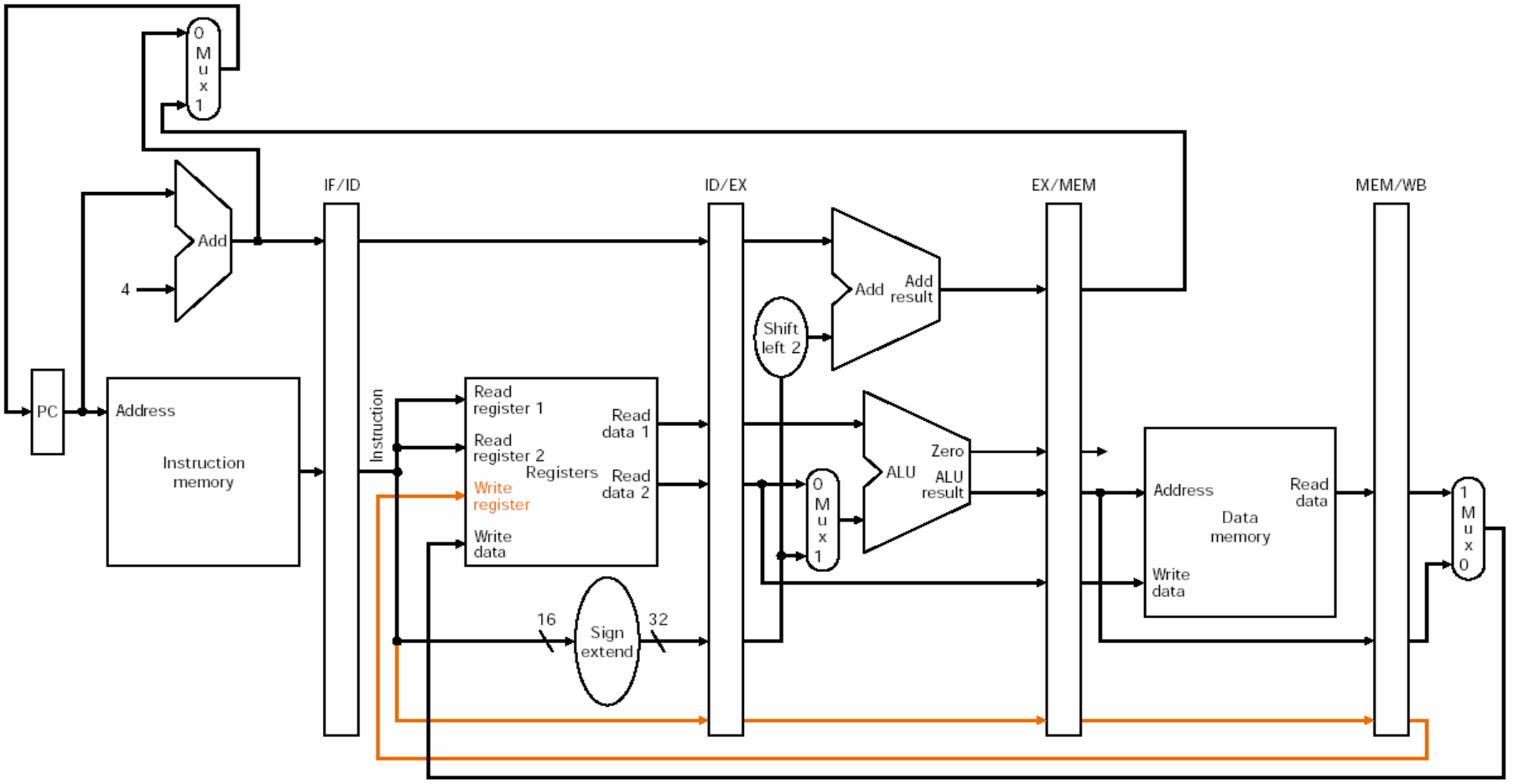
- Basic Idea



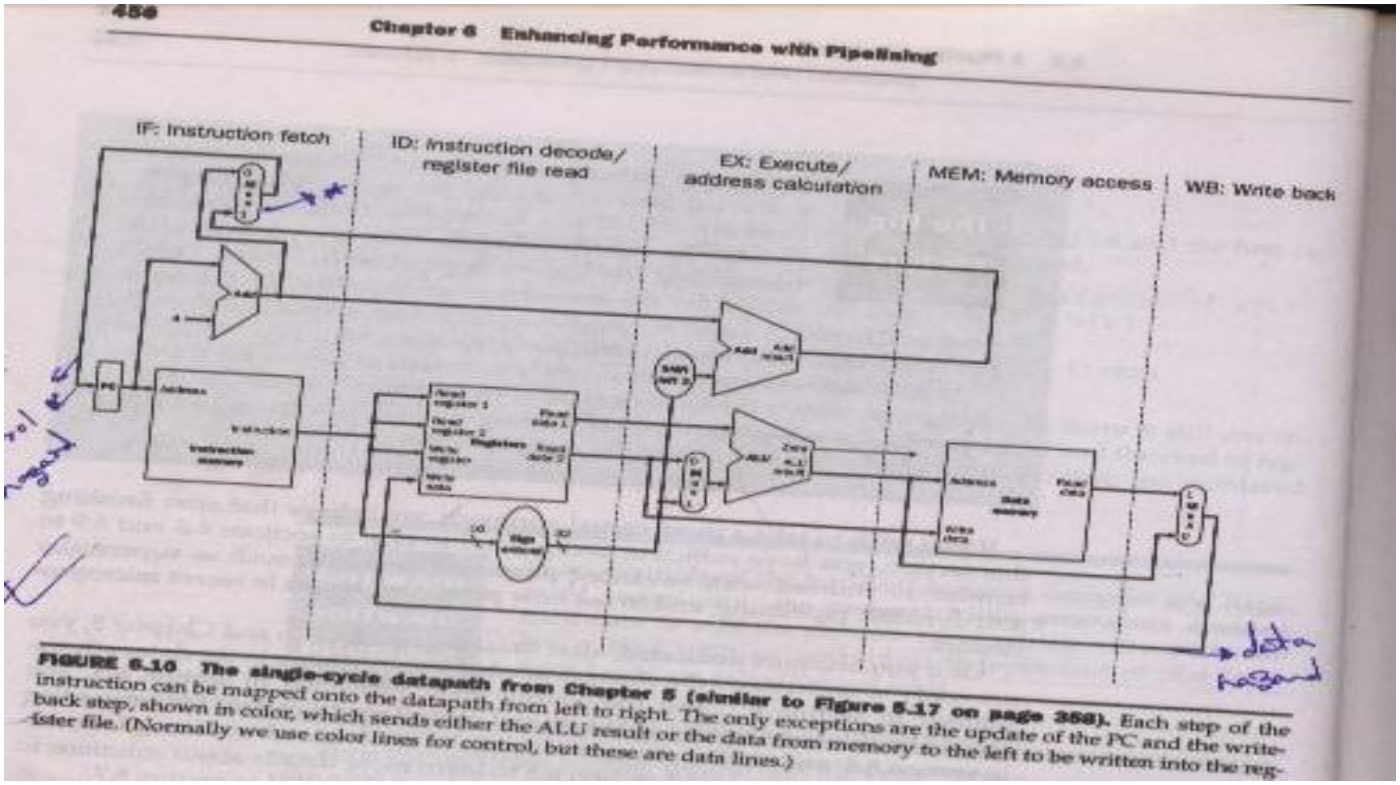
- Rectangles are pipeline registers
- Anything wrong in this picture?



Corrected Datapath



The Single Cycle Datapath



Pipeline Execution for Single Cycle Datapath

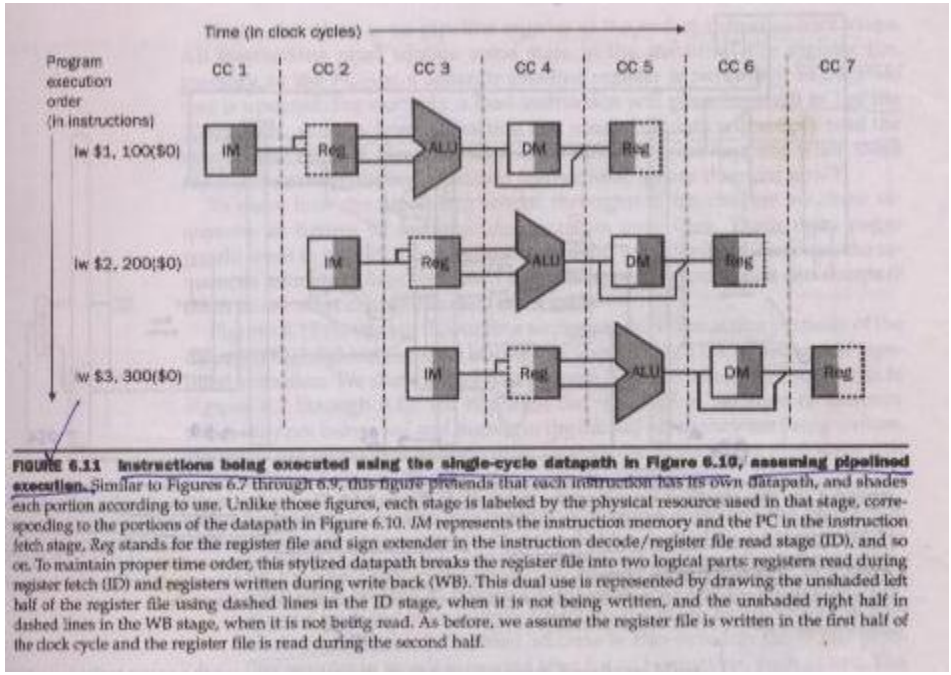


FIGURE 6.11 Instructions being executed using the single-cycle datapath in Figure 6.10, assuming pipelined execution. Similar to Figures 6.7 through 6.9, this figure presents that each instruction has its own datapath, and shades each portion according to use. Unlike those figures, each stage is labeled by the physical resource used in that stage, corresponding to the portions of the datapath in Figure 6.10. *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 the register file is read during the second half.

The Pipeline Version of the Datapath

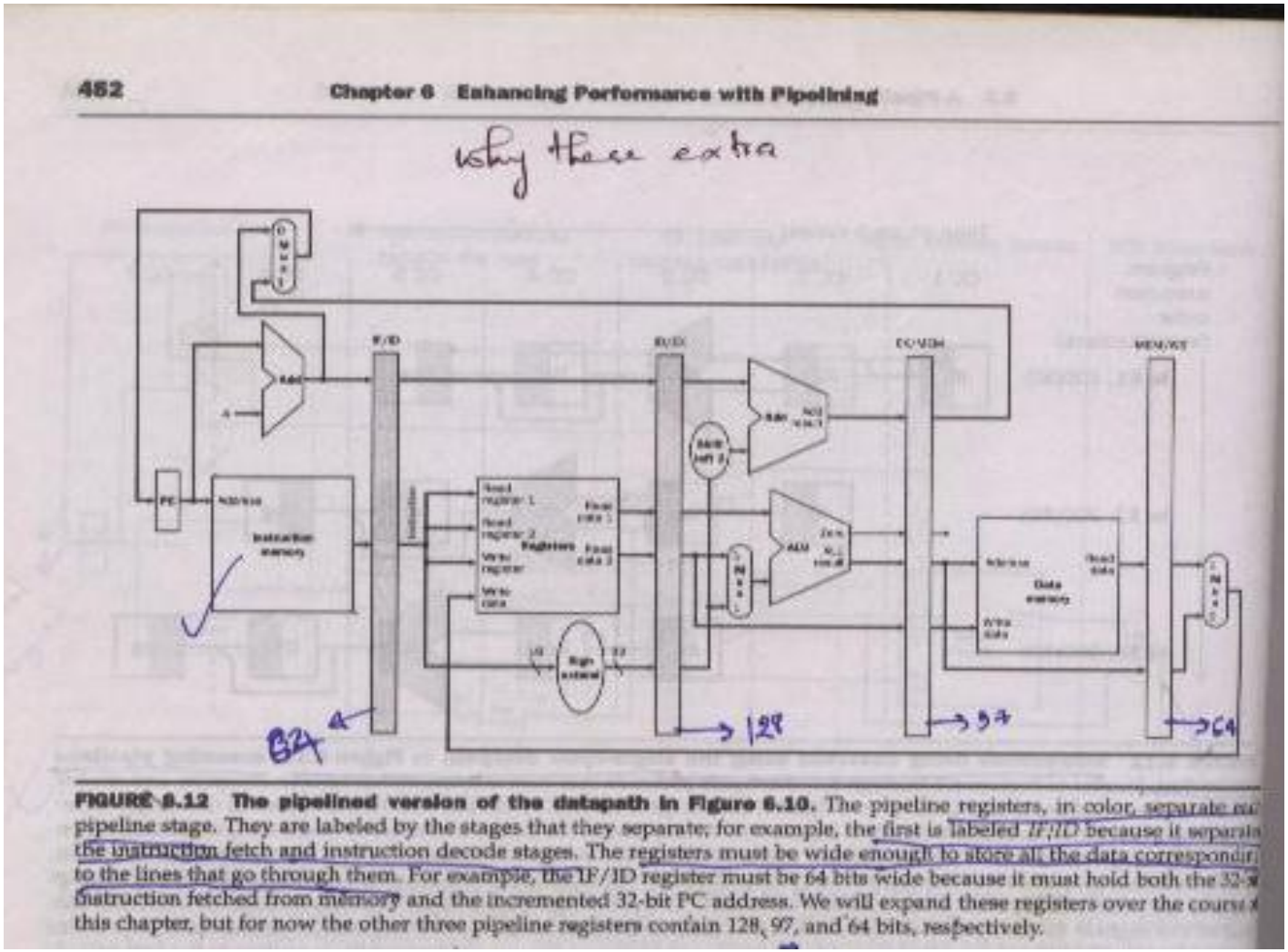


FIGURE 6.12 The pipelined version of the datapath in Figure 6.10. The pipeline registers, in color, separate each pipeline stage. They are labeled by the stages that they separate; for example, the first is labeled *IF/ID* because it separates the instruction fetch and instruction decode stages. The 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.

The Pipeline Version of the Datapath(Cont.…)

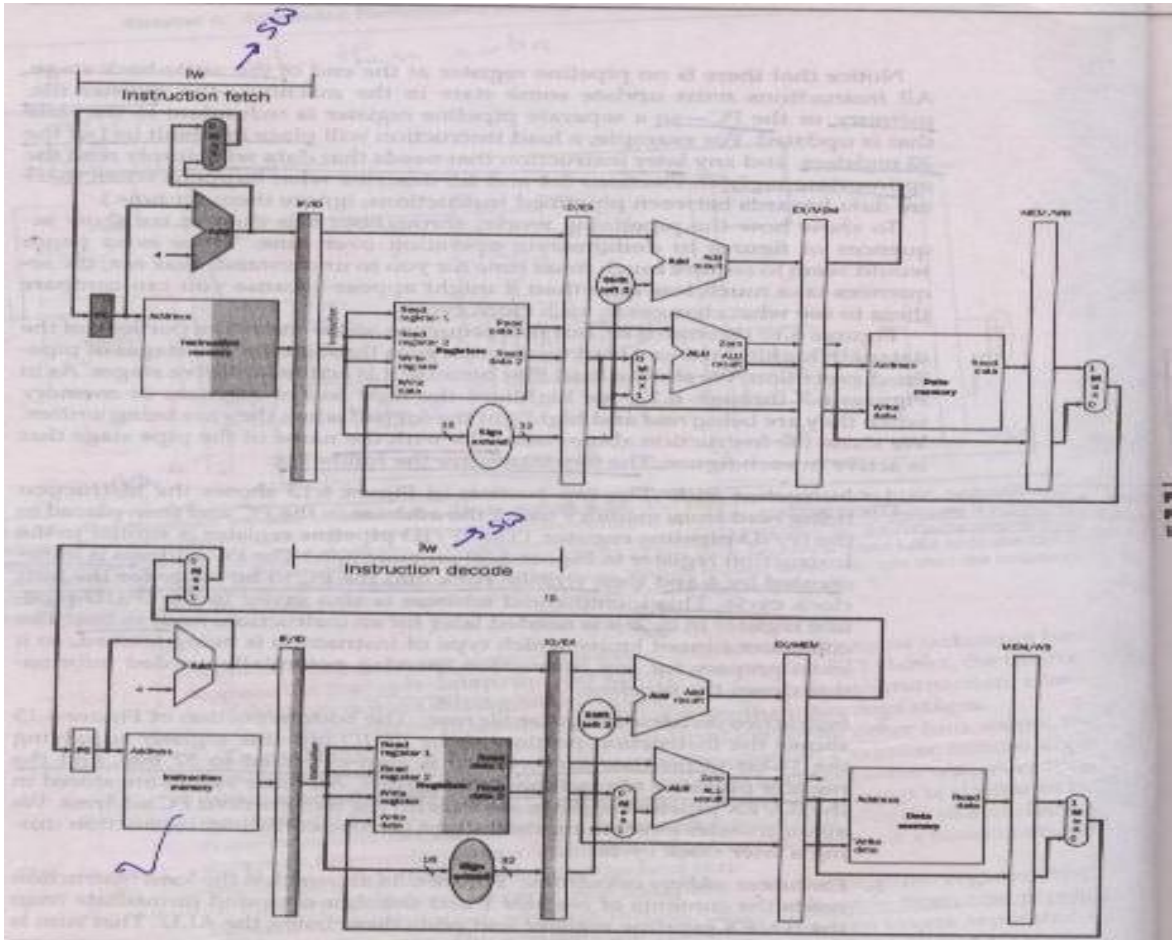
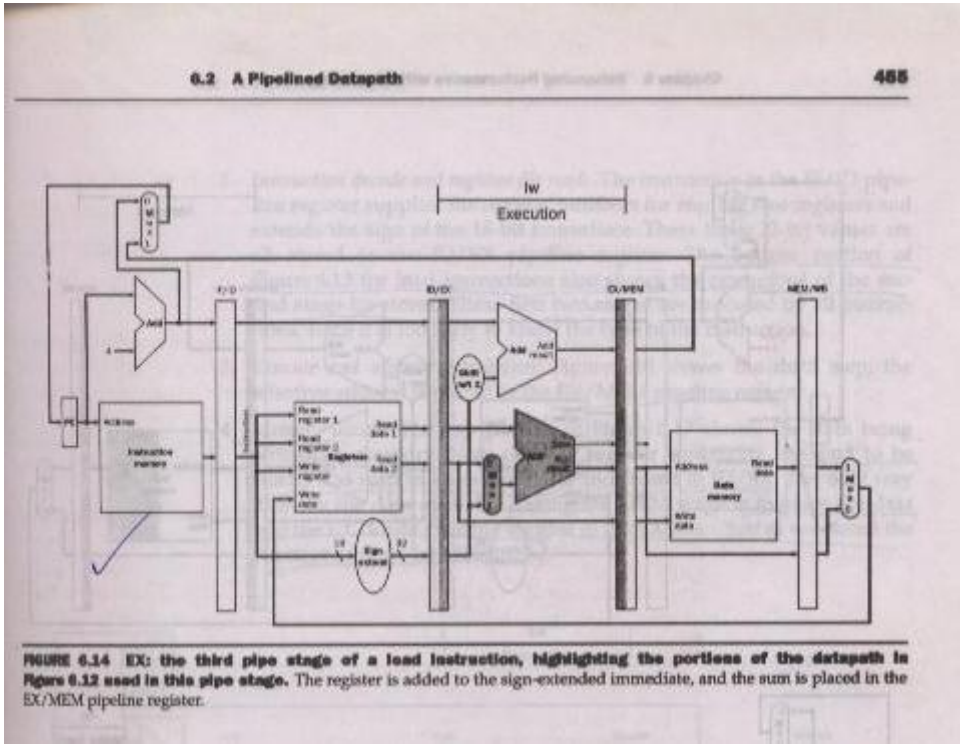


FIGURE 6.13 IF and ID: first and second pipe stages of an instruction, with the active portions of the datapath in Figure 6.12 highlighted. The highlighting convention is the same as that used in Figure 6.7. As in Chapter 5, there is no confusion when reading and writing registers because the contents change only on the clock edge. 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.

The Pipeline Version of the Datapath(Cont.…)



The Pipeline Version of the Datapath(Cont….)

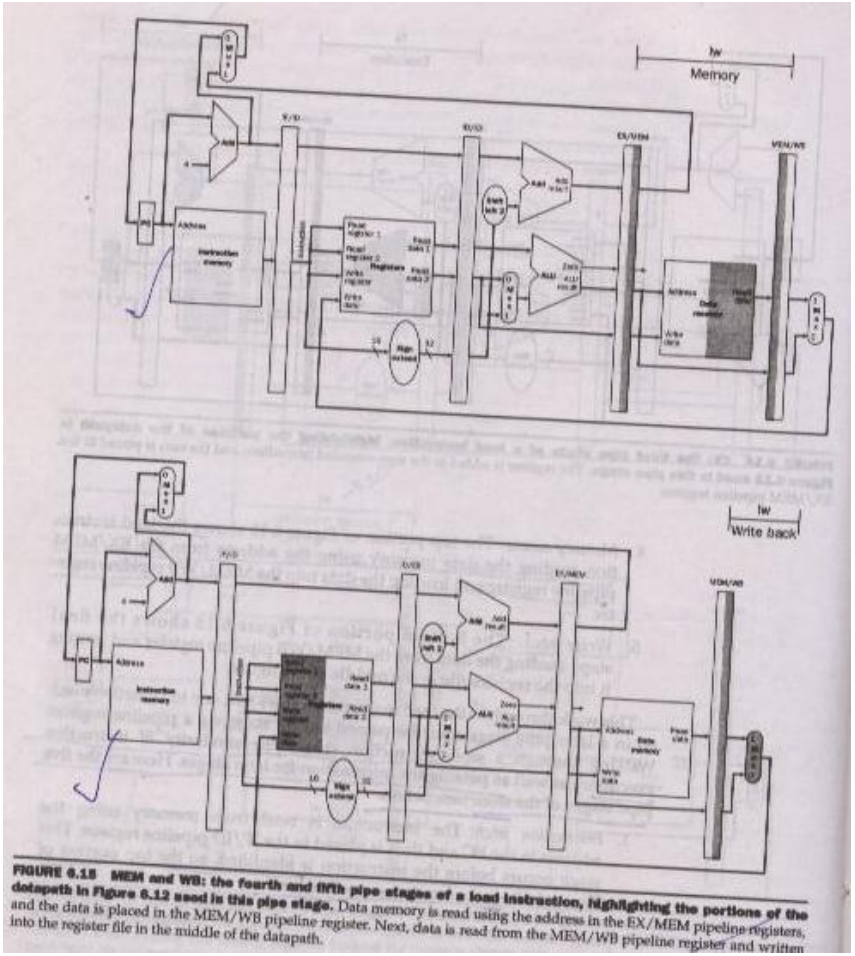


FIGURE 6.18 MEM and WB: the fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath in Figure 6.12 used in this pipe stage. Data memory is read using the address in the EX/MEM pipeline registers, and the data is placed in the MEM/WB pipeline register. Next, data is read from the MEM/WB pipeline register and written into the register file in the middle of the datapath.

The Pipeline Version of the Datapath(Cont.…)

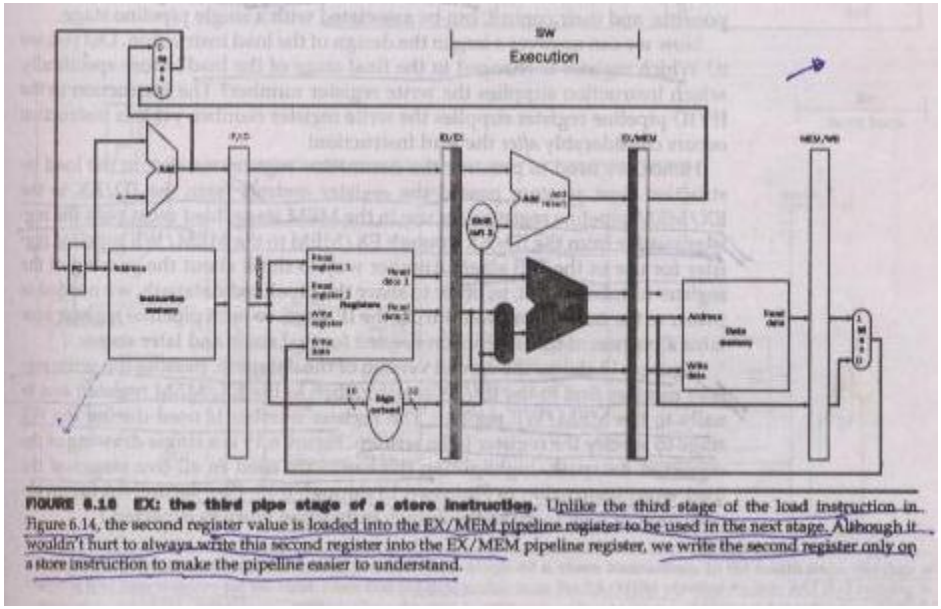


FIGURE 6.18 EX: the third pipe stage of a store instruction. Unlike the third stage of the load instruction in Figure 6.14, the second register value is loaded into the EX/MEM pipeline register to be used in the next stage. Although it wouldn't hurt to always write this second register into the EX/MEM pipeline register, we write the second register only on a store instruction to make the pipeline easier to understand.

The Pipeline Version of the Datapath(Cont.…)

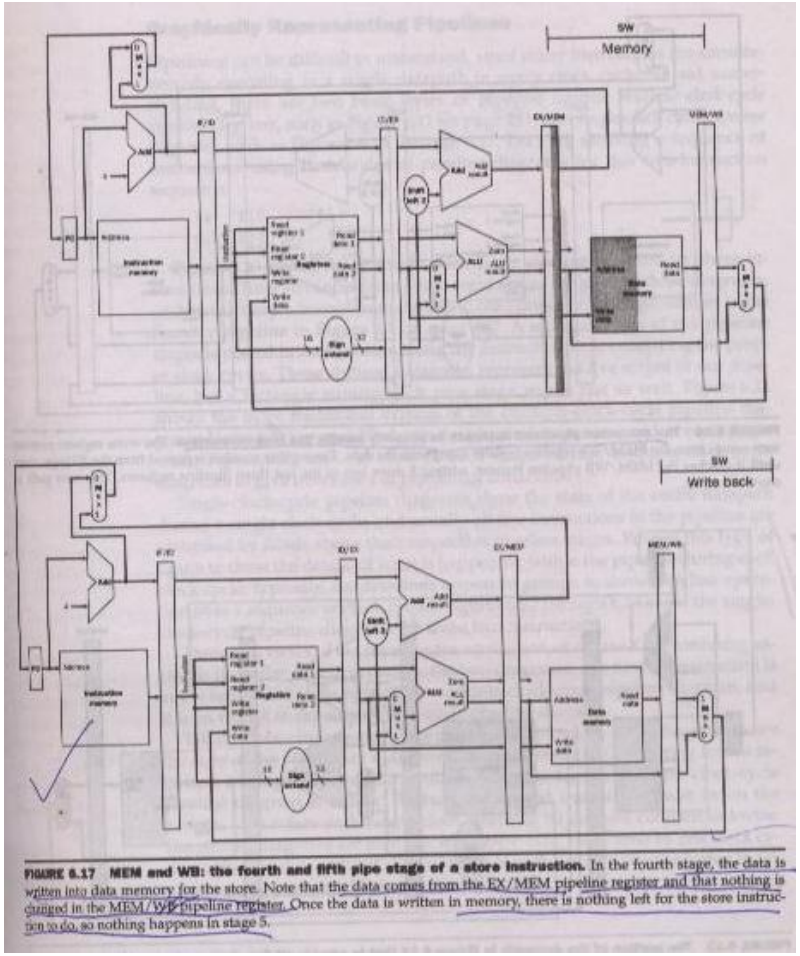


FIGURE 6.17 MEM and WB: the fourth and fifth pipe stage of a store instruction. In the fourth stage, the data is written into data memory for the store. Note that the data comes from the EX/MEM pipeline register and that nothing is changed in the MEM/WB pipeline register. Once the data is written in memory, there is nothing left for the store instruction to do, so nothing happens in stage 5.

Corrected pipeline version

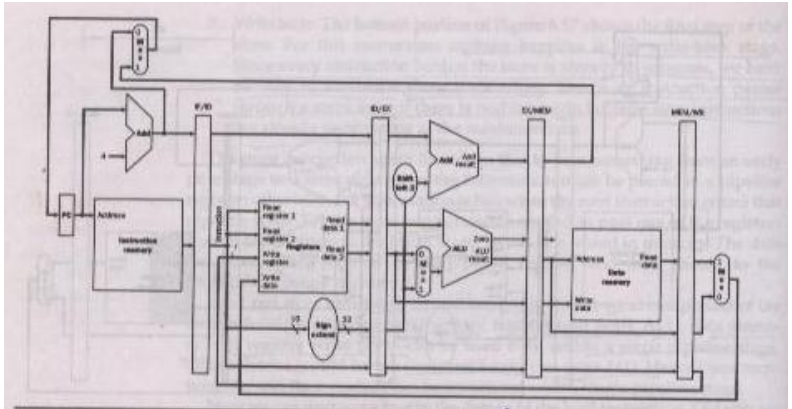


FIGURE 6.18 The corrected pipelined datapath to properly handle the load instruction. The write register number now comes from the MEM/WB pipeline register along with the data. The register number is passed from the ID pipe stage until it reaches the MEM/WB pipeline register, adding 5 more bits to the last three pipeline registers. This new path is shown in color.

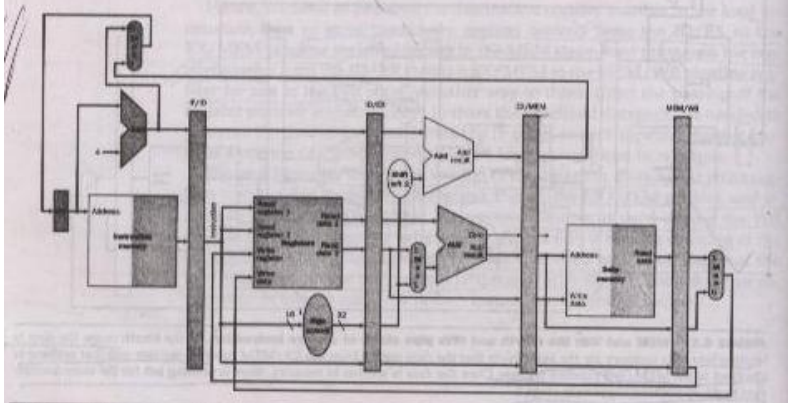


FIGURE 6.19 The portion of the datapath in Figure 6.18 that is used in all five stages of a load instruction.

Pipelined Control

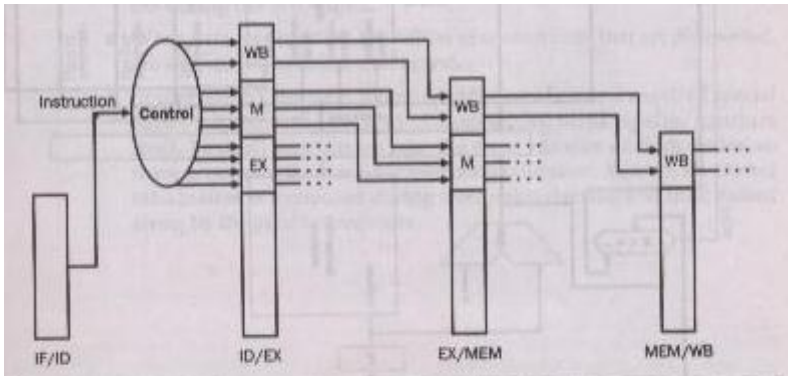


FIGURE 6.29 The control lines for the final three stages. Note that four of the nine control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

Superscalar and Dynamic pipelining

- Superpipelining: Simply means longer pipeline.

- Superscaler:
- Dynamic Pipelining:

Dynamic Pipeline Scheduling

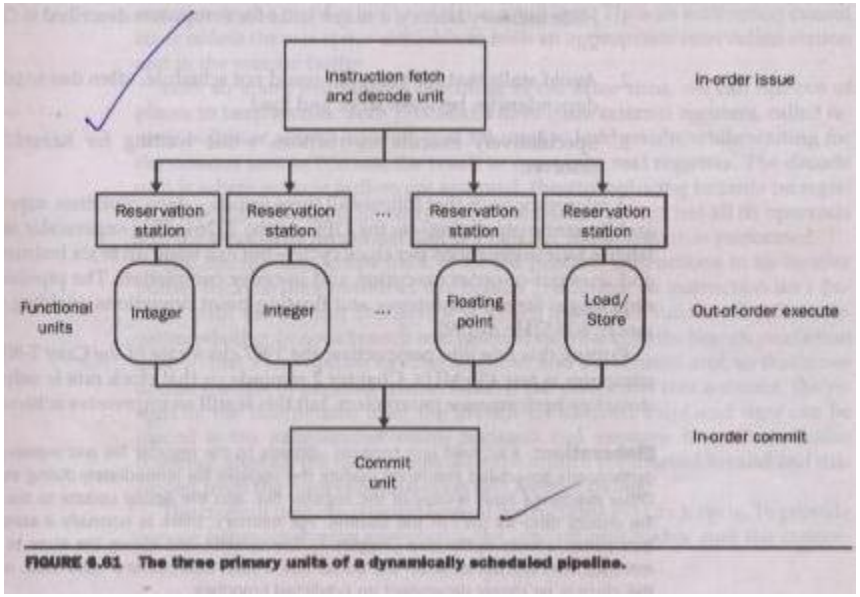


FIGURE 6.61 The three primary units of a dynamically scheduled pipeline.