Pipeline Hazards

- There are situations in pipelining when the next instruction cannot execute in the following clock cycle.
- These events are called hazards, and there are three different types; Structural Hazards, Data Hazards and Control Hazards.

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.

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.

Pipeline Hazards

Structural Hazard

- The first hazard is called a structural hazard.
- It means that the hardware cannot support the combination of instructions that we want to execute in the same clock cycle.
- A structural hazard in the laundry room would occur if we used a washerdryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away.
- Carefully scheduled pipeline plans would be foiled.

Pipeline Hazards

Structural Hazard

- The MIPS instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline.
- Suppose we had a single memory instead of two memories. If the pipeline in Figure 27 had a fourth instruction, we would see that in the same clock cycle the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory.
- Without two memories, our pipeline could have a structural hazard.

Pipeline Hazards

Data Hazards

- Data hazards occur when the pipeline must be stalled because one step must wait for another to complete.
- Suppose you found a sock at the folding station for which no match existed.
- One possible strategy is to run down to your room and search through your clothes bureau to see if you can find the match.
- Obviously, while you are doing the search, loads must wait that have completed drying and are ready to fold as well as those that have finished washing and are ready to dry.

Pipeline Hazards

Data Hazards

- In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline (a relationship that does not really exist when doing laundry).
- For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum ($s0):

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

Pipeline Hazards

Data Hazards

- Without intervention, a data hazard could severely stall the pipeline.
- The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.
- Although we could try to rely on compilers to remove all such hazards, the results would not be satisfactory.
- These dependences happen just too often and the delay is just too long to expect the compiler to rescue us from this dilemma.

Figure 27

Pipeline Hazards

Data Hazards

- The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard.
- For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract.
- Adding extra hardware to retrieve the missing item early from the internal resources is called forwarding or bypassing.

Pipeline Hazards

EXAMPLE

Data Hazards

- Forwarding with Two Instructions

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

For the two instructions above, show what pipeline stages would be connected by forwarding.

Figure 28 represents the datapath during the five stages of the pipeline. Align a copy of the datapath for each instruction, similar to the laundry pipeline.

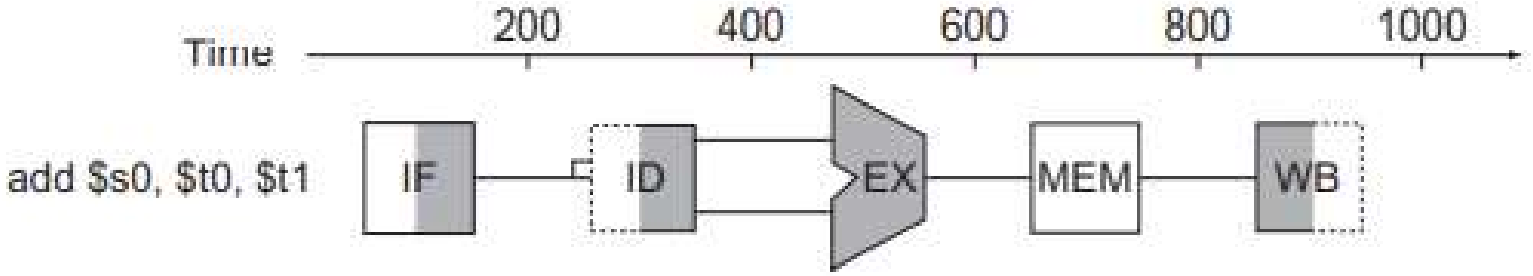


FIGURE 28 Graphical representation of the instruction pipeline, similar in spirit to the laundry pipeline

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipeline Hazards

ANSWER

Data Hazards

- Forwarding with Two Instructions

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

Figure 29 shows the connection to forward the value in $s0 after the execution stage of the add instruction as input to the execution stage of the sub instruction.

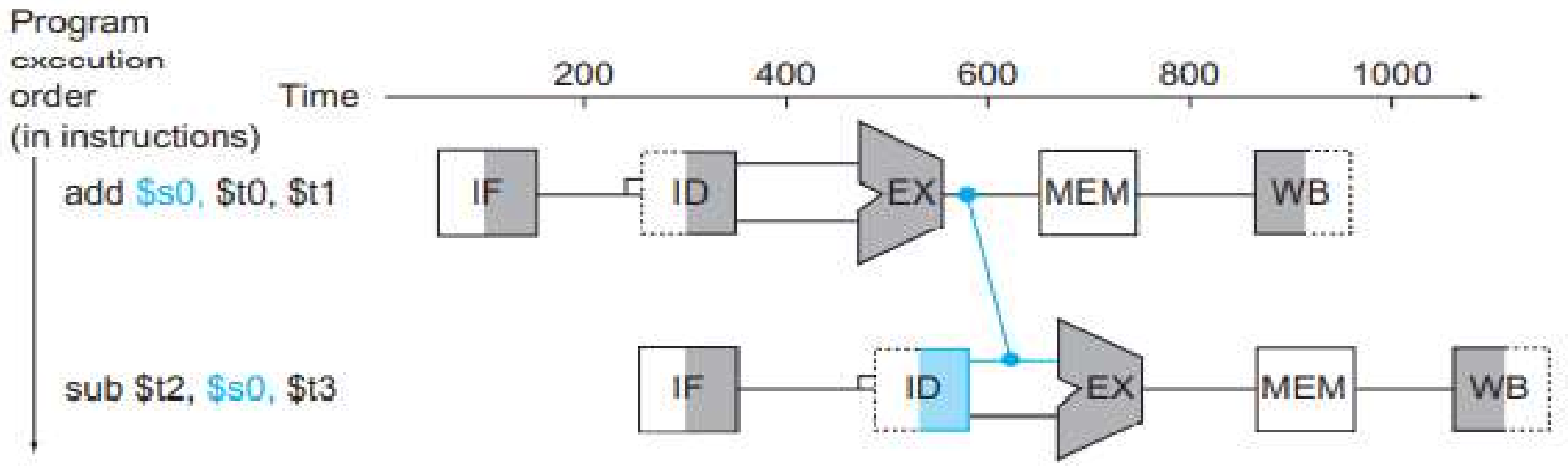


FIGURE 29 Graphical representation of forwarding.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipeline Hazards

ANSWER

Data Hazards

- Forwarding with Two Instructions

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

- In this graphical representation of events, forwarding paths are valid only if the destination stage is later in time than the source stage.
- For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following, since that would mean going backward in time.

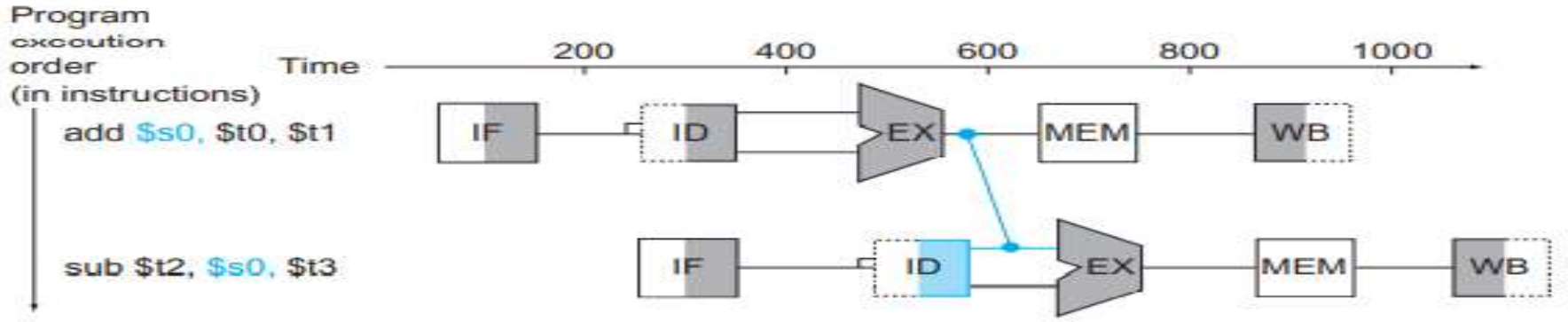


FIGURE 29 Graphical representation of forwarding.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipeline Hazards

Data Hazards

- Forwarding cannot prevent all pipeline stalls.
- For example, suppose the first instruction was a load of $s0 instead of an add. As we can imagine from looking at Figure 30, the desired data would be available only after the fourth stage of the first instruction in the dependence, which is too late for the input of the third stage of sub.

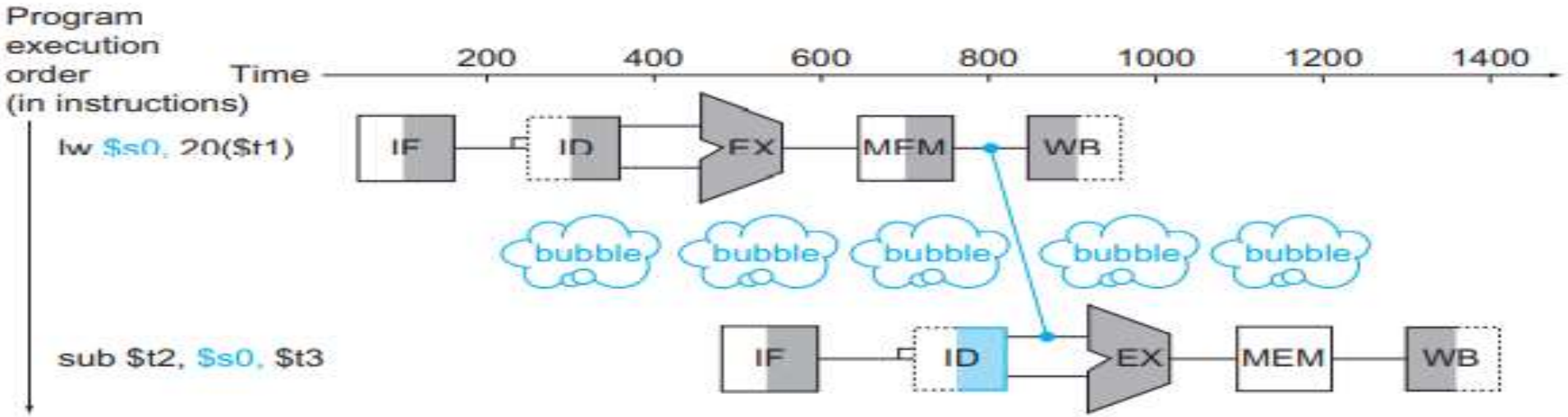


FIGURE 30 We need a stall even with forwarding when an R-format instruction following a load tries to use the data.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipeline Hazards

Data Hazards

- Hence, even with forwarding, we would have to stall one stage for a load-use data hazard, as Figure 30 shows.
- This figure shows an important pipeline concept, officially called a pipeline stall, but often given the nickname bubble.

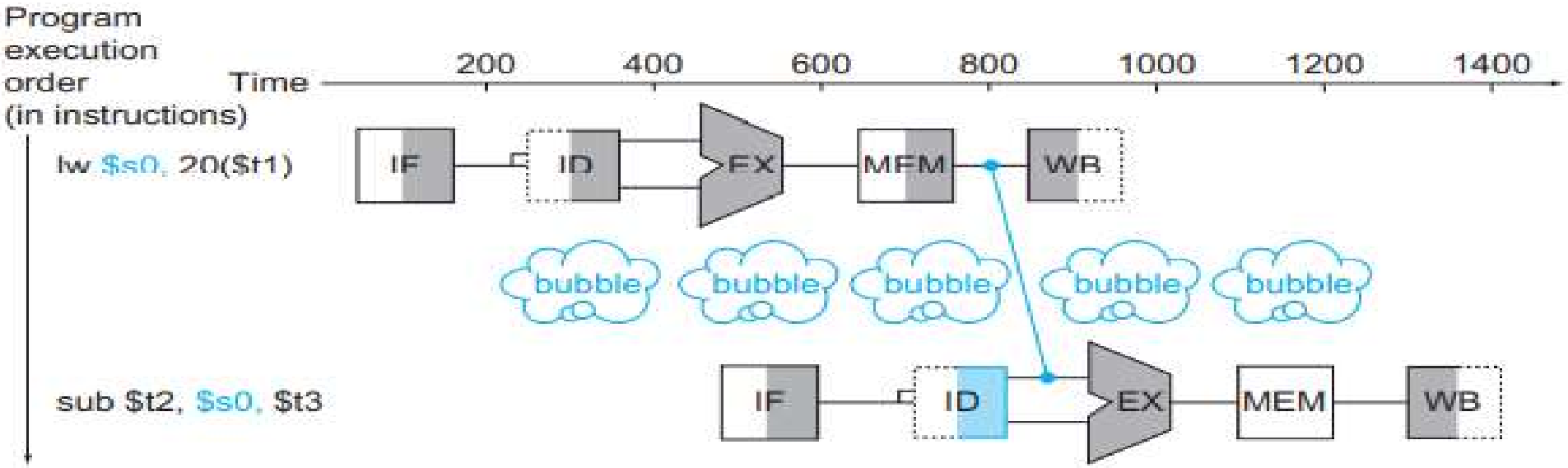


FIGURE 30 We need a stall even with forwarding when an R-format instruction following a load tries to use the data.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipeline Hazards

Data Hazards

Load-use data hazard: A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction.

Pipeline stall: Also called bubble. A stall initiated in order to resolve a hazard.

Pipeline Hazards

EXAMPLE

Data Hazards

- Reordering Code to Avoid Pipeline Stall

  Consider the following code segment in C:

  ```
  a = b + e;
  c = b + f;
  ```

  Here is the generated MIPS code for this segment, assuming all variables are in memory and are addressable as offsets from $t0:

  ```
  lw $t1, 0($t0)
  lw $t2, 4($t0)
  add $t3, $t1,$t2
  sw $t3, 12($t0)
  lw $t4, 8($t0)
  add $t5, $t1,$t4
  sw $t5, 16($t0)
  ```

Find the hazards in the preceding code segment and reorder the instructions to avoid any pipeline stalls?

Pipeline Hazards

ANSWER

Data Hazards

- Both add instructions have a hazard because of their respective dependence on the immediately preceding lw instruction.
- Notice that bypassing eliminates several other potential hazards, including the dependence of the first add on the first lw and any hazards for store instructions.
- Moving up the third lw instruction to become the third instruction eliminates both hazards:

```
lw $t1, 0($t0)
lw $t2, 4($t0)
lw $t4, 8($t0)
add $t3, $t1,$t2
sw $t3, 12($t0)
add $t5, $t1,$t4
sw $t5, 16($t0)
```

On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

Pipeline Hazards

Data Hazards

- Each MIPS instruction writes at most one result and does this in the last stage of the pipeline.
- Forwarding is harder if there are multiple results to forward per instruction or if there is a need to write a result early on in instruction execution.

- Elaboration: The name "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

Pipeline Hazards

Control Hazards

- Third type of hazard is called a control hazard, arising from the need to make a decision based on the results of one instruction while others are executing.
- Suppose our laundry crew was given the happy task of cleaning the uniforms of a football team.
- Given how filthy the laundry is, we need to determine whether the detergent and water temperature setting we select is strong enough to get the uniforms clean but not so strong that the uniforms wear out sooner. In our laundry pipeline, we have to wait until after the second stage to examine the dry uniform to see if we need to change the washer setup or not. What to do?

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

Pipeline Hazards

Control Hazards

- Here is the first of two solutions(Stall and Predict) to control hazards in the laundry room and its computer equivalent.
- Stall: Just operate sequentially until the first batch is dry and then repeat until you have the right formula.
- This conservative option certainly works, but it is slow.
- The equivalent decision task in a computer is the branch instruction.
- Notice that we must begin fetching the instruction following the branch on the very next clock cycle. Pipeline cannot possibly know what the next instruction should be, since it only just received the branch instruction from memory!
- Just as with laundry, one possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from.

Pipeline Hazards

Control Hazards

- Let's assume that we put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage of the pipeline.
- Even with this extra hardware, the pipeline involving conditional branches would look like Figure 31.

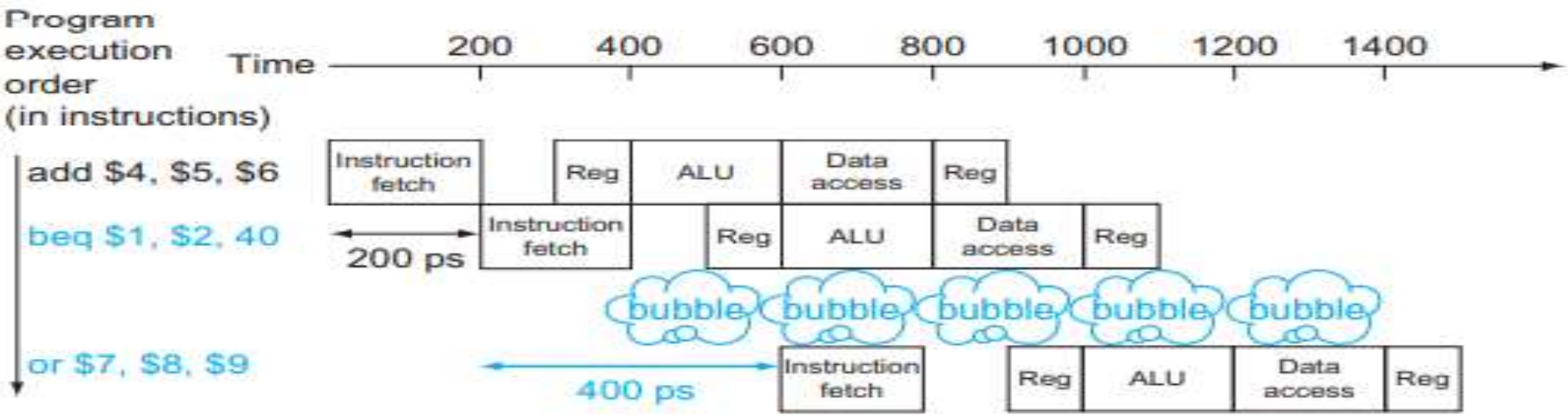


FIGURE 31 Pipeline showing stalling on every conditional branch as solution to control hazards.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipeline Hazards

Control Hazards

- If we cannot resolve the branch in the second stage, as is often the case for longer pipelines, then we'd see an even larger slowdown if we stall on branches.
- The cost of this option is too high for most computers to use and motivates a second solution to the control hazard using one of great ideas: Predict
- Predict: If you're pretty sure you have the right formula to wash uniforms, then just predict that it will work and wash the second load while waiting for the first load to dry.
- This option does not slow down the pipeline when you are correct. When you are wrong, however, you need to redo the load that was washed while guessing the decision.

Pipeline Hazards

Control Hazards

- Computers do indeed use prediction to handle branches.
- One simple approach is to predict always that branches will be untaken.
- When you're right, the pipeline proceeds at full speed.
- Only when branches are taken does the pipeline stall. Figure 32 shows such an example.

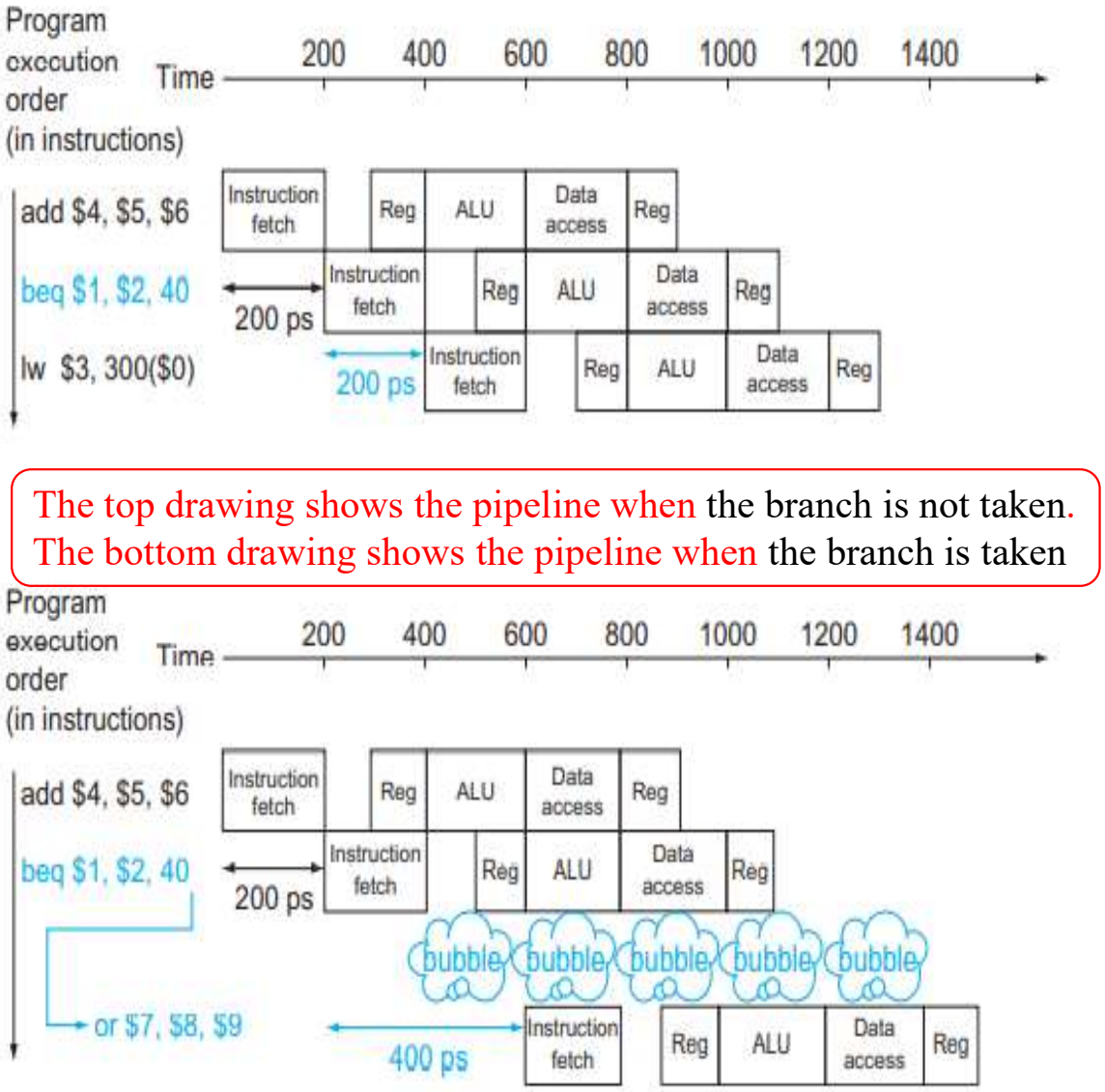


FIGURE 32 Predicting that branches are not taken as a solution to control hazard.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipeline Hazards

Control Hazards

- A more sophisticated version of branch prediction would have some branches predicted as taken and some as untaken.
- In our analogy, the dark or home uniforms might take one formula while the light or road uniforms might take another.
- In the case of programming, at the bottom of loops are branches that jump back to the top of the loop.
- Since they are likely to be taken and they branch backward, we could always predict taken for branches that jump to an earlier address.

Branch prediction: A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome

Pipeline Hazards

Control Hazards

- Such rigid approaches to branch prediction rely on stereotypical behavior and don't account for the individuality of a specific branch instruction.
- Dynamic hardware predictors, in stark contrast, make their guesses depending on the behavior of each branch and may change predictions for a branch over the life of a program.
- Following our analogy, in dynamic prediction a person would look at how dirty the uniform was and guess at the formula, adjusting the next prediction depending on the success of recent guesses.

Pipeline Hazards

Control Hazards

- One popular approach to dynamic prediction of branches is keeping a history for each branch as taken or untaken, and then using the recent past behavior to predict the future.
- When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed branch have no effect and must restart the pipeline from the proper branch address.
- In our laundry analogy, we must stop taking new loads so that we can restart the load that we incorrectly predicted.

Pipeline Overview Summary

- Pipelining is a technique that exploits parallelism among the instructions in a sequential instruction stream.
- It has the substantial advantage that, unlike programming a multiprocessor, it is fundamentally invisible to the programmer.
- Pipelining increases the number of simultaneously executing instructions and the rate at which instructions are started and completed.
- Pipelining does not reduce the time it takes to complete an individual instruction, also called the latency.
- For example, the five-stage pipeline still takes 5 clock cycles for the instruction to complete.
- Pipelining improves instruction throughput rather than individual instruction execution time or latency.

Pipelined Datapath and Control

- We must separate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution.
    1. IF: Instruction fetch
    2. ID: Instruction decode and register file read
    3. EX: Execution or address calculation
    4. MEM: Data memory access
    5. WB: Write back
- In Figure 33, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through five stages as they complete execution.

Pipelined Datapath and Control

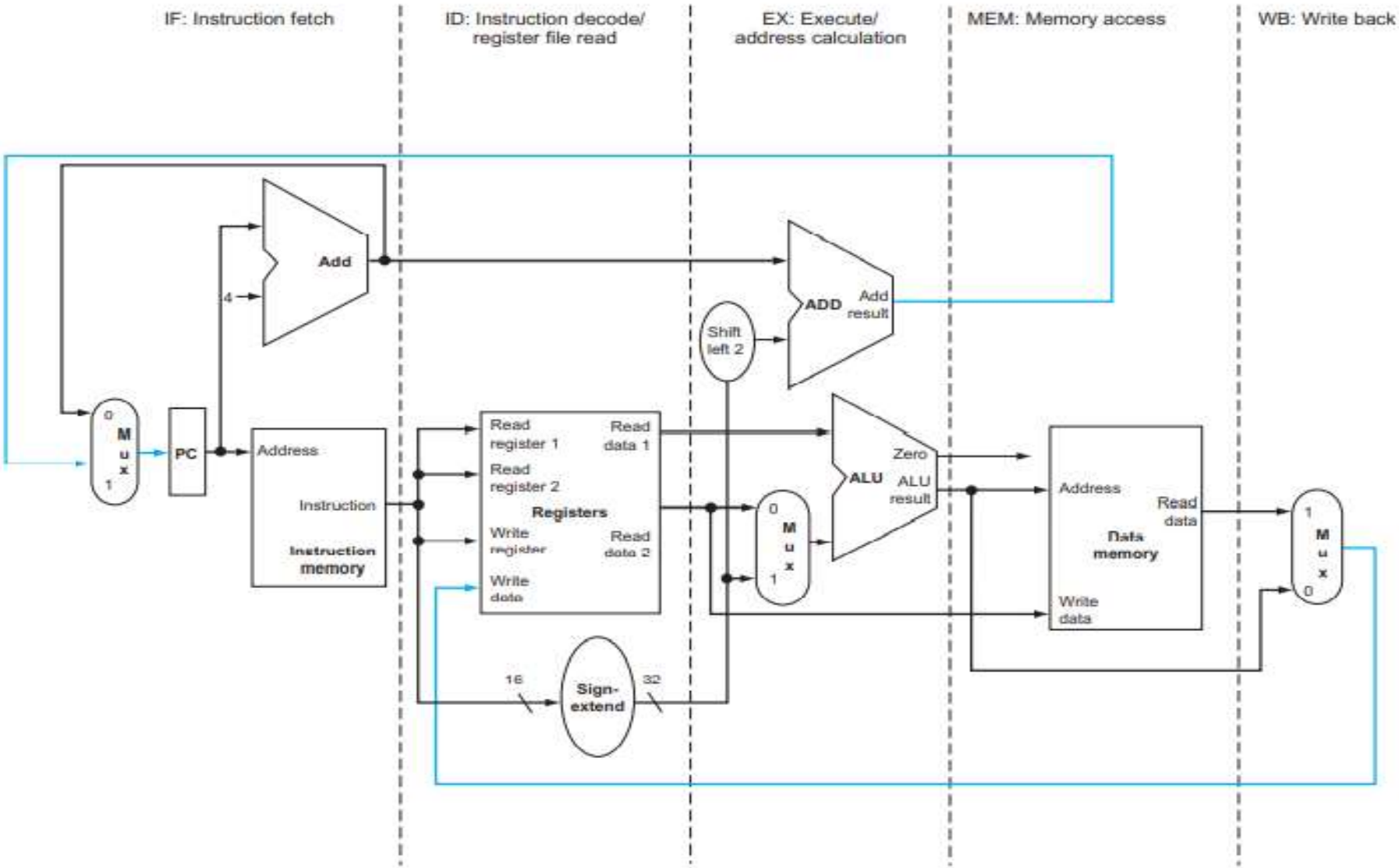


FIGURE .33 The single-cycle datapath

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipelined Datapath and Control

- There are two exceptions to this left-to-right flow of instructions:
  - The write-back stage, which places the result back into the register file in the middle of the datapath. [may lead to data hazards]
  - The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage. . [may lead to control hazards]

Pipelined Datapath and Control

- Data flowing from right to left does not affect the current instruction; these reverse data movements influence only later instructions in the pipeline.
- Note that the first right-to-left flow of data can lead to data hazards and the second leads to control hazards.

Pipelined Datapath and Control

- One way to show what happens in pipelined execution is to pretend that each instruction has its own datapath, and then to place these datapaths on a timeline to show their relationship.
- Figure 34 shows the execution of the instructions by displaying their private datapaths on a common timeline.

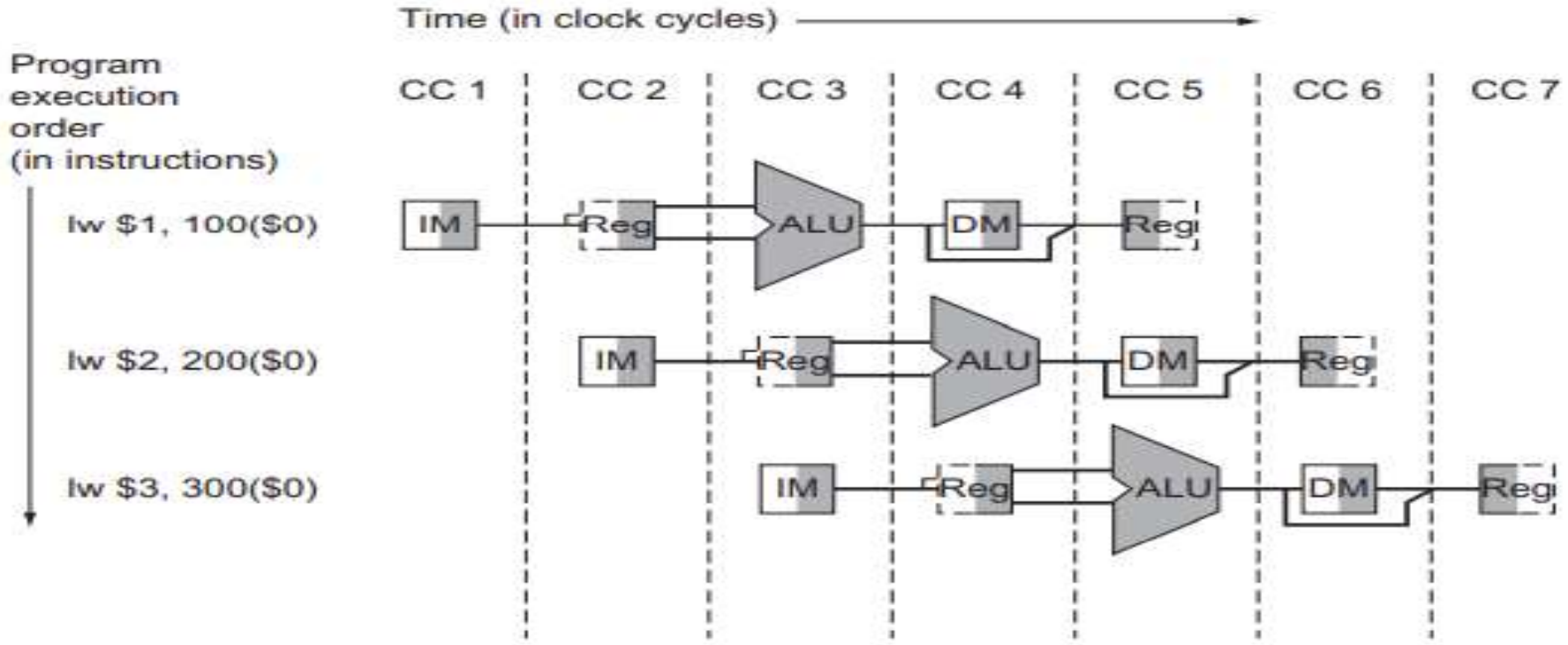


FIGURE 34 Instructions being executed using the single-cycle datapath assuming pipelined execution.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipelined Datapath and Control

- Figure 34 seems to suggest that three instructions need three datapaths.
- Instead, we add registers to hold data so that portions of a single datapath can be shared during instruction execution.

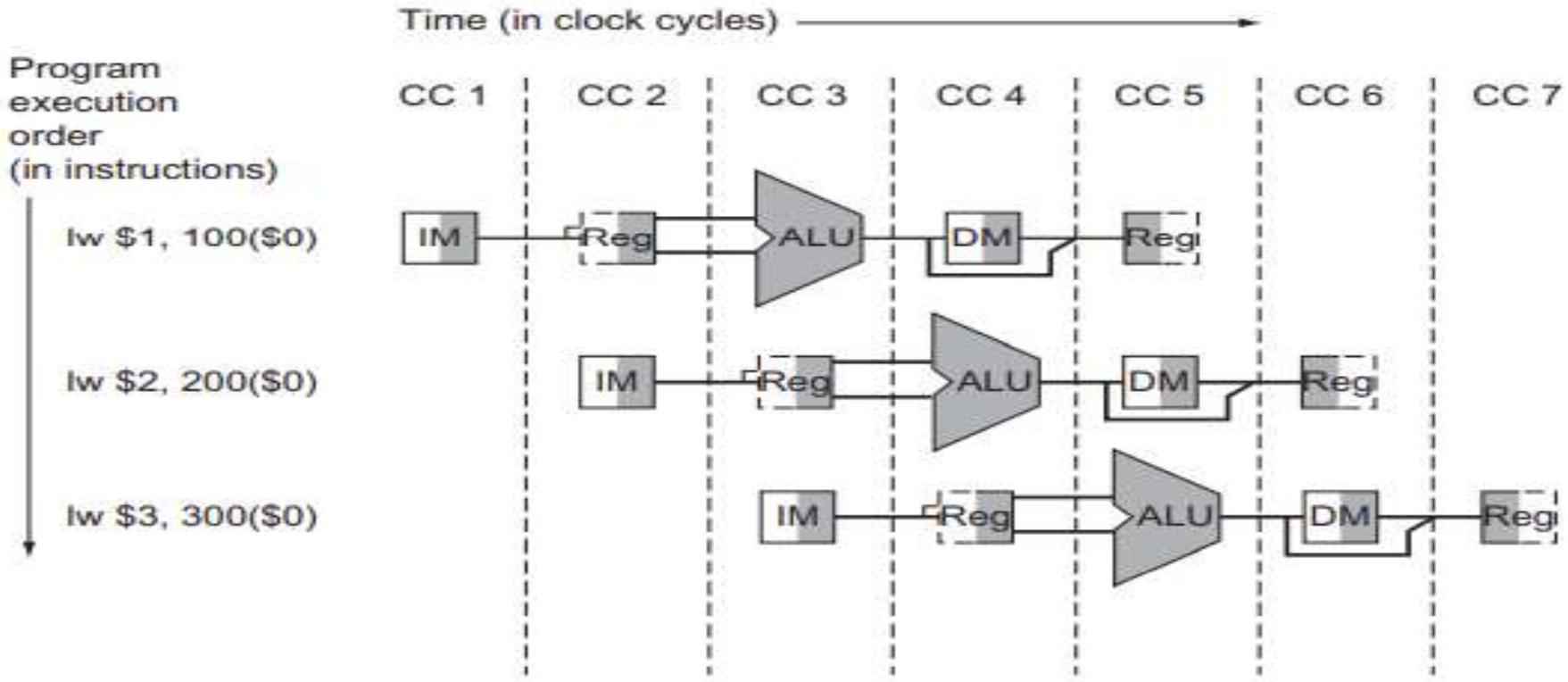


FIGURE 34 Instructions being executed using the single-cycle datapath assuming pipelined execution.

BACK

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipelined Datapath and Control

- For example, as Figure 34 shows, the instruction memory is used during only one of the five stages of an instruction, allowing it to be shared by following instructions during the other four stages.
- To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register.
- Similar arguments apply to every pipeline stage, so we must place registers wherever there are dividing lines between stages in Figure 33.
- Returning to our laundry analogy, we might have a basket between each pair of stages to hold the clothes for the next step.

Pipelined Datapath and Control

- Figure 35 shows the pipelined datapath with the pipeline registers highlighted.
- All instructions advance during each clock cycle from one pipeline register to the next.

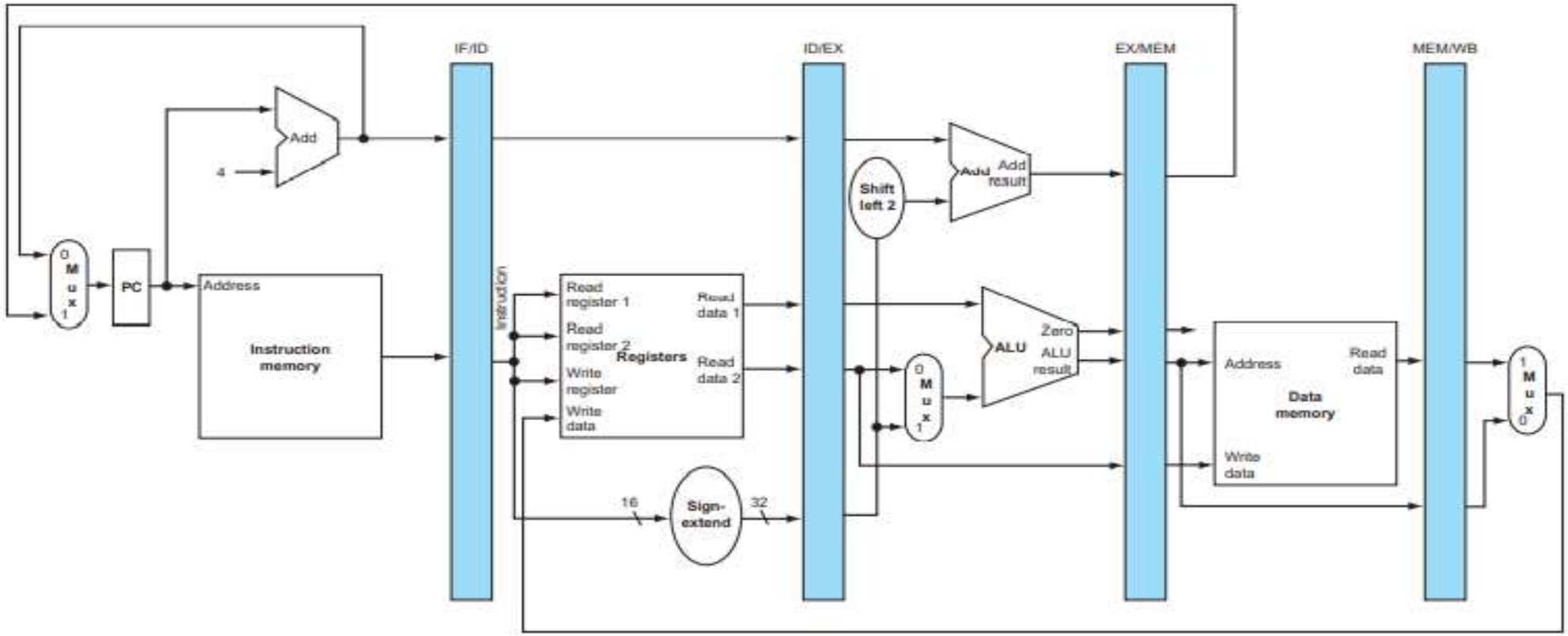


FIGURE 35 The pipelined version of the datapath in Figure 33. The pipeline registers, in color, separate each pipeline stage.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipelined Datapath and Control

- The registers are named for the two stages separated by that register.
- For example, the pipeline register between the IF and ID stages is called IF/ID.

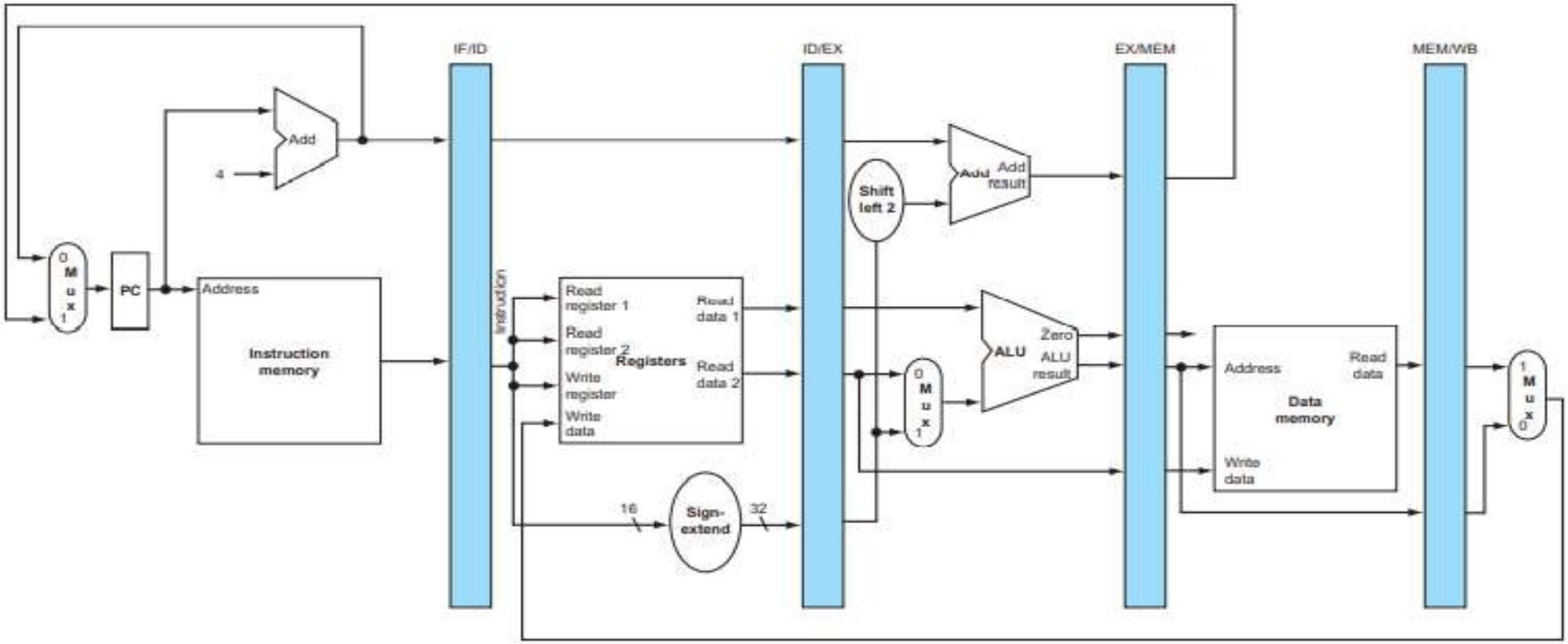


FIGURE 35 The pipelined version of the datapath in Figure 33. The pipeline registers, in color, separate each pipeline stage.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipelined Datapath and Control

- All instructions must update some state in the processor—the register file, memory, or the PC—so a separate pipeline register is redundant to the state that is updated.
- For example, a load instruction will place its result in 1 of the 32 registers, and any later instruction that needs that data will simply read the appropriate register.
- Of course, every instruction updates the PC, whether by incrementing it or by setting it to a branch destination address.
- The PC can be thought of as a pipeline register: one that feeds the IF stage of the pipeline.
- Unlike the shaded pipeline registers in Figure 35, however, the PC is part of the visible architectural state; its contents must be saved when an exception occurs, while the contents of the pipeline registers can be discarded.

Pipelined Datapath and Control

- Figures 36 through 38, our first sequence, show the active portions of the datapath highlighted as a load instruction goes through the five stages of pipelined execution.
- We show a load first because it is active in all five stages.
- As in Figures 28 through 30, we highlight the right half of registers or memory when they are being read and highlight the left half when they are being written.

Pipelined Datapath and Control - LOAD

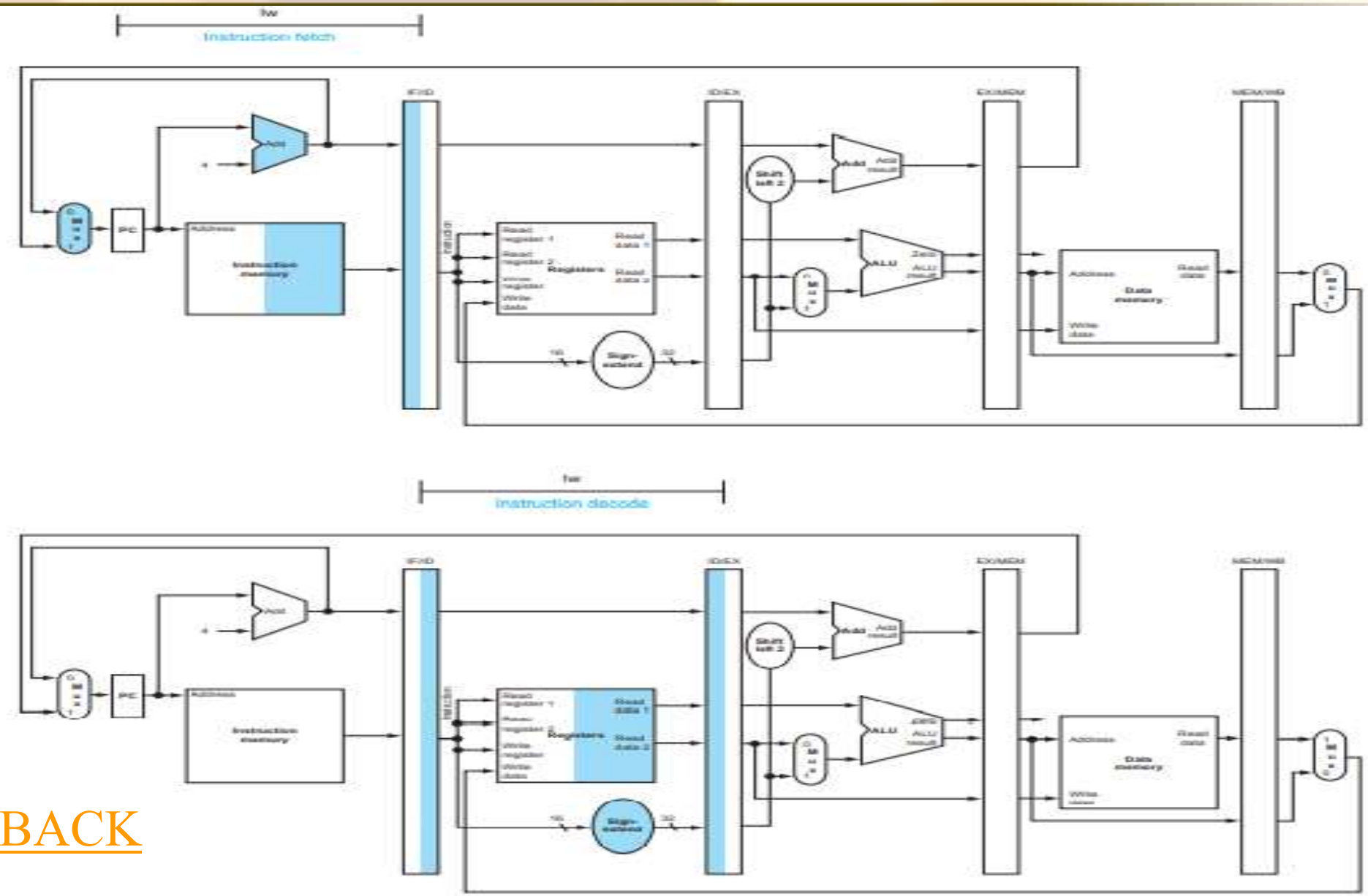
Instruction abbreviation, lw

The five stages are the following:

1. Instruction fetch:
   - The top portion of Figure 36 shows 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.
   - The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

Pipelined Datapath and Control - LOAD

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]



BACK

FIGURE 36 IF and ID: First and second pipe stages of an instruction, with the active portions of the datapath in Figure 35 highlighted.

Pipelined Datapath and Control - LOAD

The five stages are the following:

2. Instruction decode and register file read:

- The bottom portion of Figure 36 shows the instruction portion of the IF/ID pipeline register supplying the 16-bit immediate field, which is sign-extended to 32 bits, and the register numbers to read the two registers.
- All three values are stored in the ID/EX pipeline register, along with the incremented PC address. We again transfer everything that might be needed by any instruction during a later clock cycle

Pipelined Datapath and Control - LOAD

The five stages are the following:

3. Execute or address calculation:

- Figure 37 shows that the load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register.

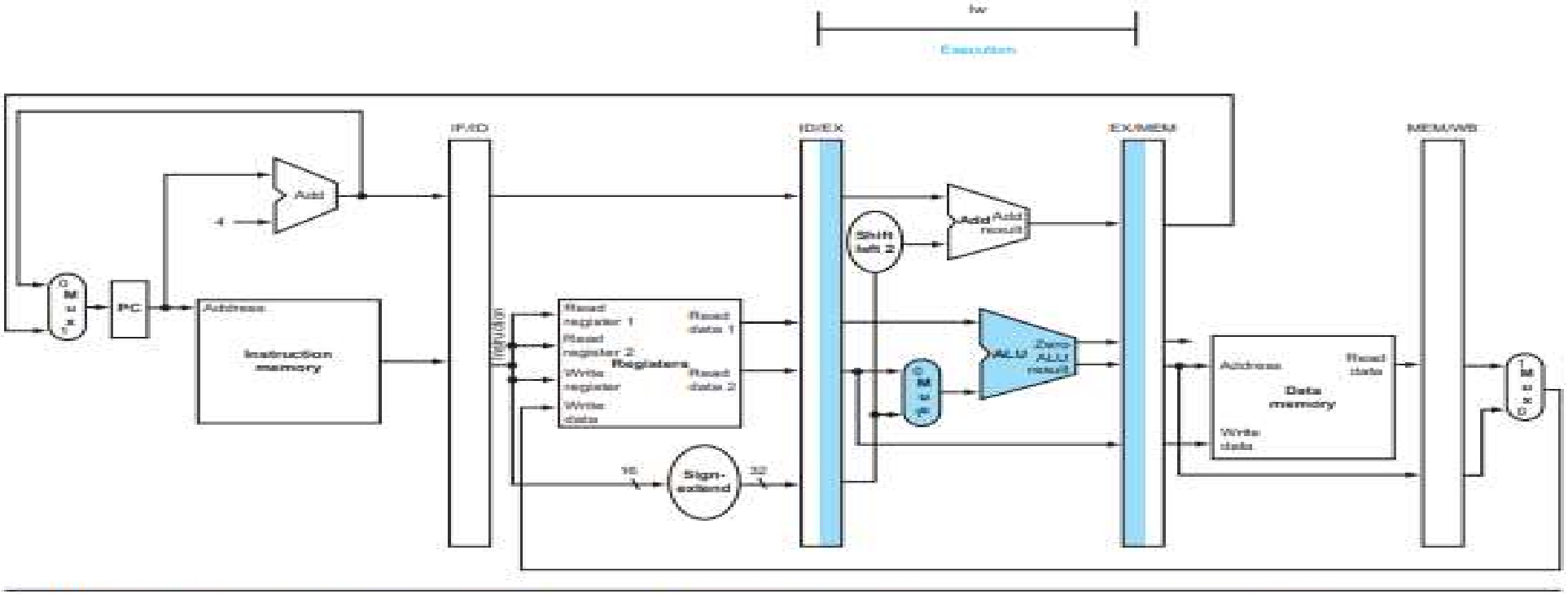


FIGURE 37 EX: The third pipe stage of a load instruction, highlighting the portions of the datapath

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipelined Datapath and Control - LOAD

The five stages are the following:

4. Memory access:

The top portion of Figure 38 shows the load instruction reading the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.

5. Write-back:

The bottom portion of Figure 38 shows the final step: reading the data from the MEM/WB pipeline register and writing it into the register file in the middle of the figure.

Pipelined Datapath and Control - LOAD

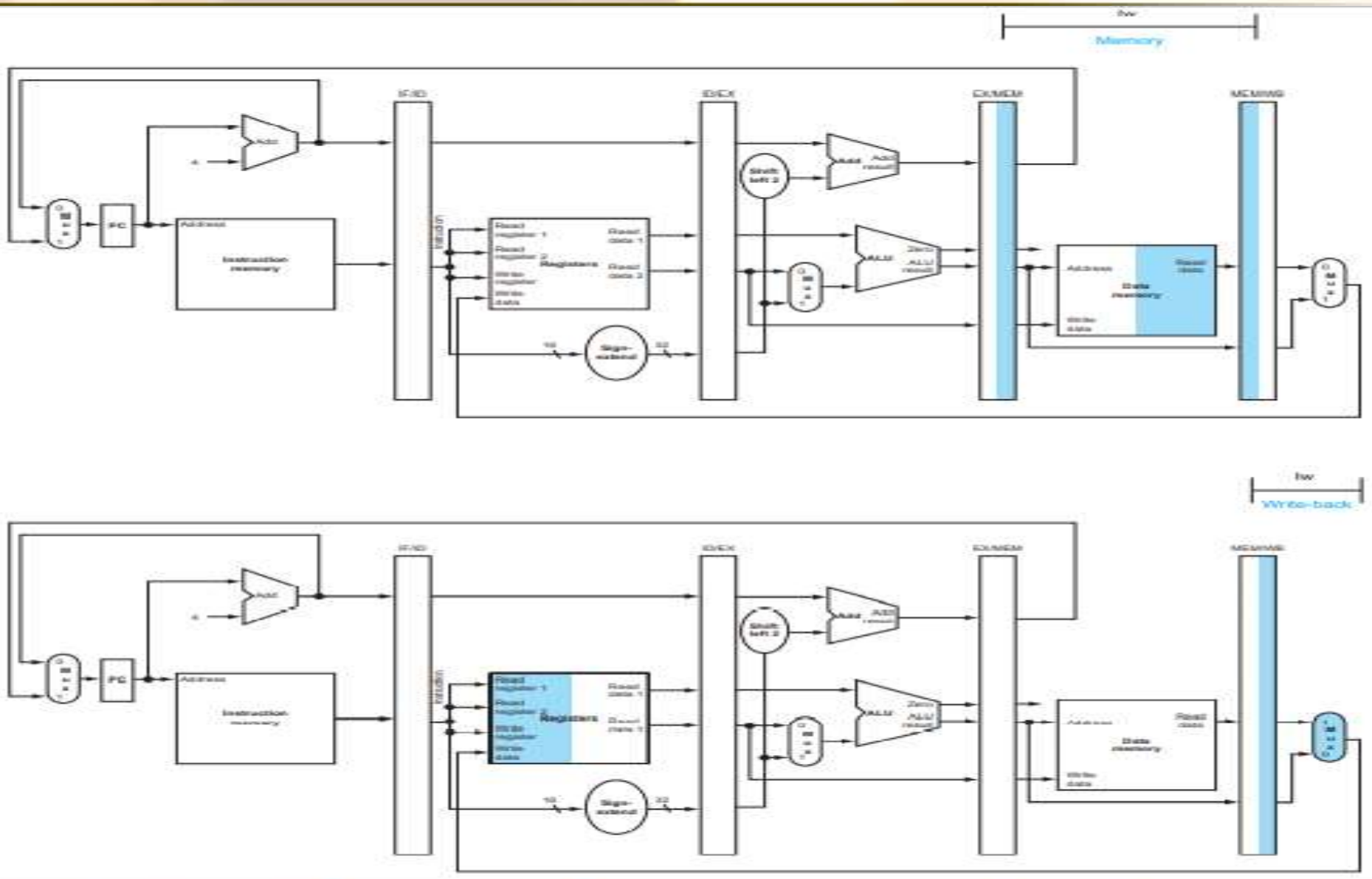


FIGURE 38 MEM and WB: The fourth and fifth pipe stages of a load instruction, highlighting the portions of the datapath

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipelined Datapath and Control -STORE

Five pipeline stages of the store instruction:

1. Instruction fetch: The instruction is read from memory using the address in the PC and then is placed in the IF/ID pipeline register. This stage occurs before the instruction is identified, so the top portion of Figure 36 works for store as well as load.
2. Instruction decode and register file read: The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the 16-bit immediate. These three 32-bit values are all stored in the ID/EX pipeline register. The bottom portion of Figure 36 for load instructions also shows the operations of the second stage for stores.

These first two stages are executed by all instructions, since it is too early to know the type of the instruction.

Pipelined Datapath and Control -STORE

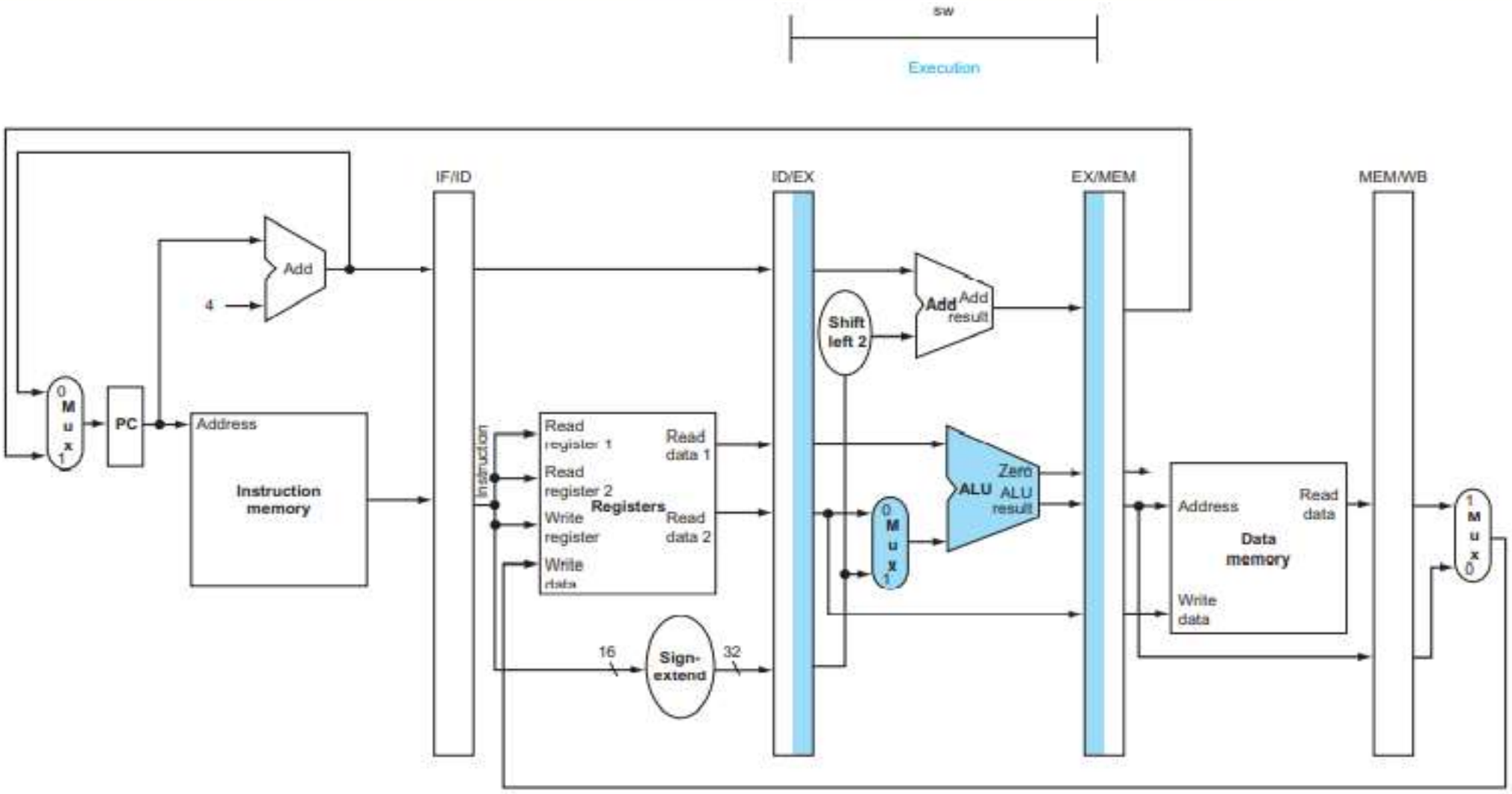


FIGURE 39 EX: The third pipe stage of a store instruction.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

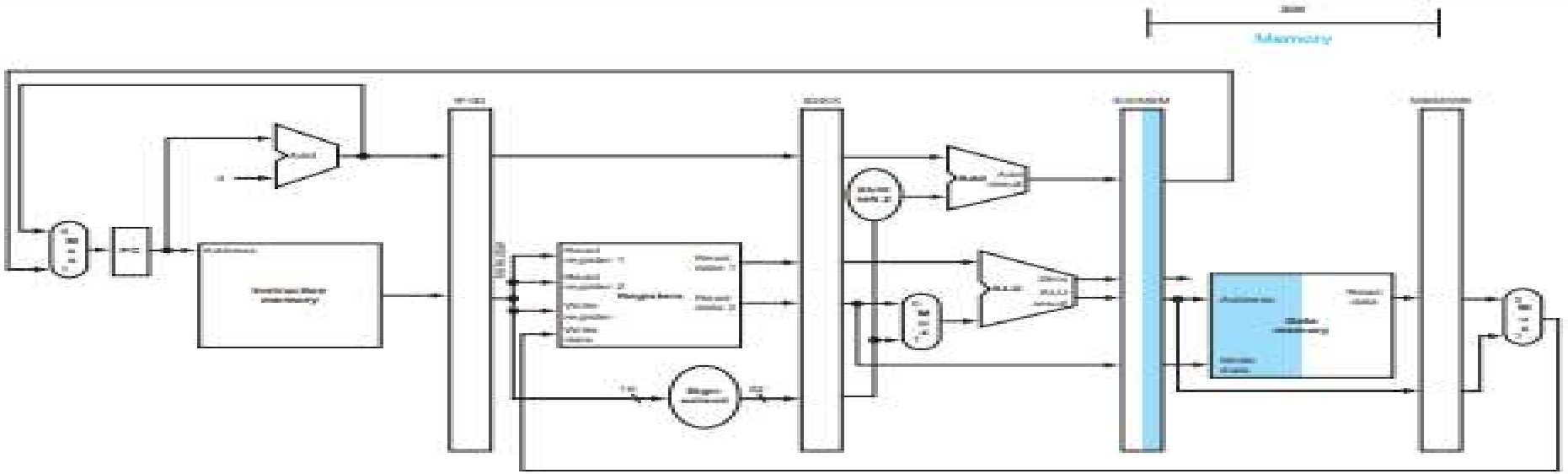
Pipelined Datapath and Control -STORE

Five pipe stages of the store instruction:

3. Execute and address calculation: Figure 39 shows the third step; the effective address is placed in the EX/MEM pipeline register.

4. Memory access: The top portion of Figure 40 shows the data being written to memory. Note that the register containing the data to be stored was read in an earlier stage and stored in ID/EX. The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address into EX/MEM.

Pipelined Datapath and Control -STORE



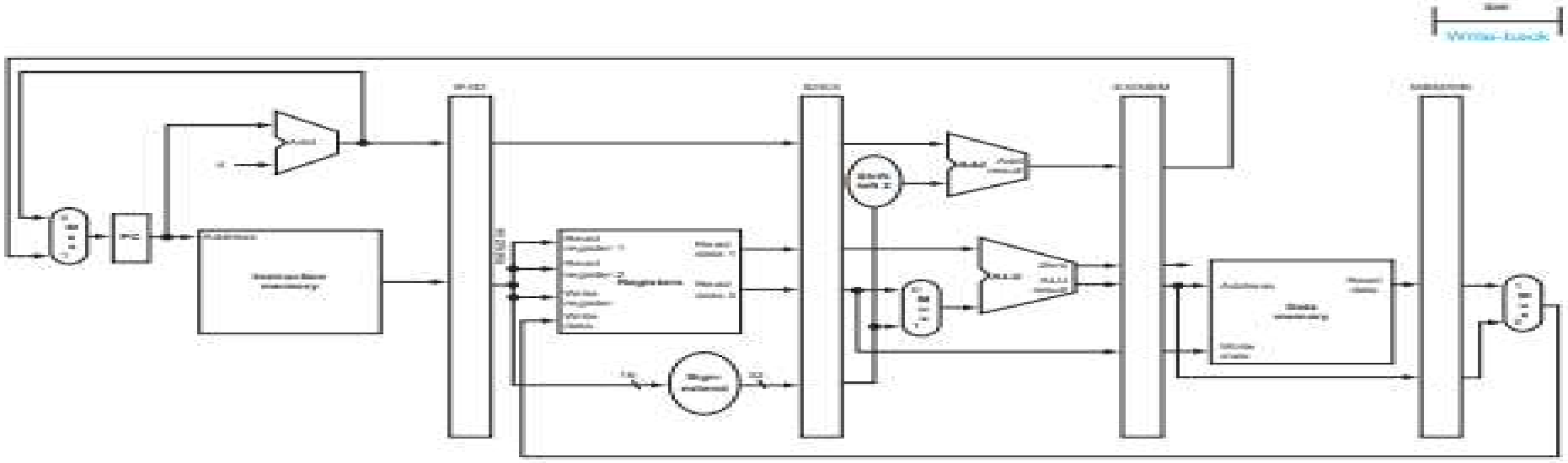


FIGURE 4.40 MEM and WB: The fourth and fifth pipe stages of a store instruction.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipelined Datapath and Control -STORE

Five pipe stages of the store instruction:

5. Write-back: The bottom portion of Figure 40 shows the final step of the store. For this instruction, nothing happens in the write-back stage. Since every instruction behind the store is already in progress, we have no way to accelerate those instructions. Hence, an instruction passes through a stage even if there is nothing to do, because later instructions are already progressing at the maximum rate.

Pipelined Datapath and Control

- The store instruction again illustrates that to pass something from an early pipe stage to a later pipe stage, the information must be placed in a pipeline register; otherwise, the information is lost when the next instruction enters that pipeline stage.
- The data was first placed in the ID/EX pipeline register and then passed to the EX/MEM pipeline register.
- Load and store illustrate a second key point: each logical component of the datapath—such as instruction memory, register read ports, ALU, data memory, and register write port—can be used only within a single pipeline stage. Otherwise, we would have a structural hazard. Hence these components, and their control, can be associated with a single pipeline stage.

Pipelined Datapath and Control

- There are two basic styles of pipeline figures: multiple-clock-cycle pipeline diagrams(Figure 43) and single-clock-cycle pipeline diagrams(Figure 36-40).
- The multiple-clock-cycle diagrams are simpler but do not contain all the details.
- For example, consider the following five-instruction sequence:

```
lw $10, 20($1)
sub $11, $2, $3
add $12, $3, $4
lw $13, 24($1)
add $14, $5, $6
```

- Figure 43 shows the multiple-clock-cycle pipeline diagram for these instructions.

Pipelined Datapath and Control

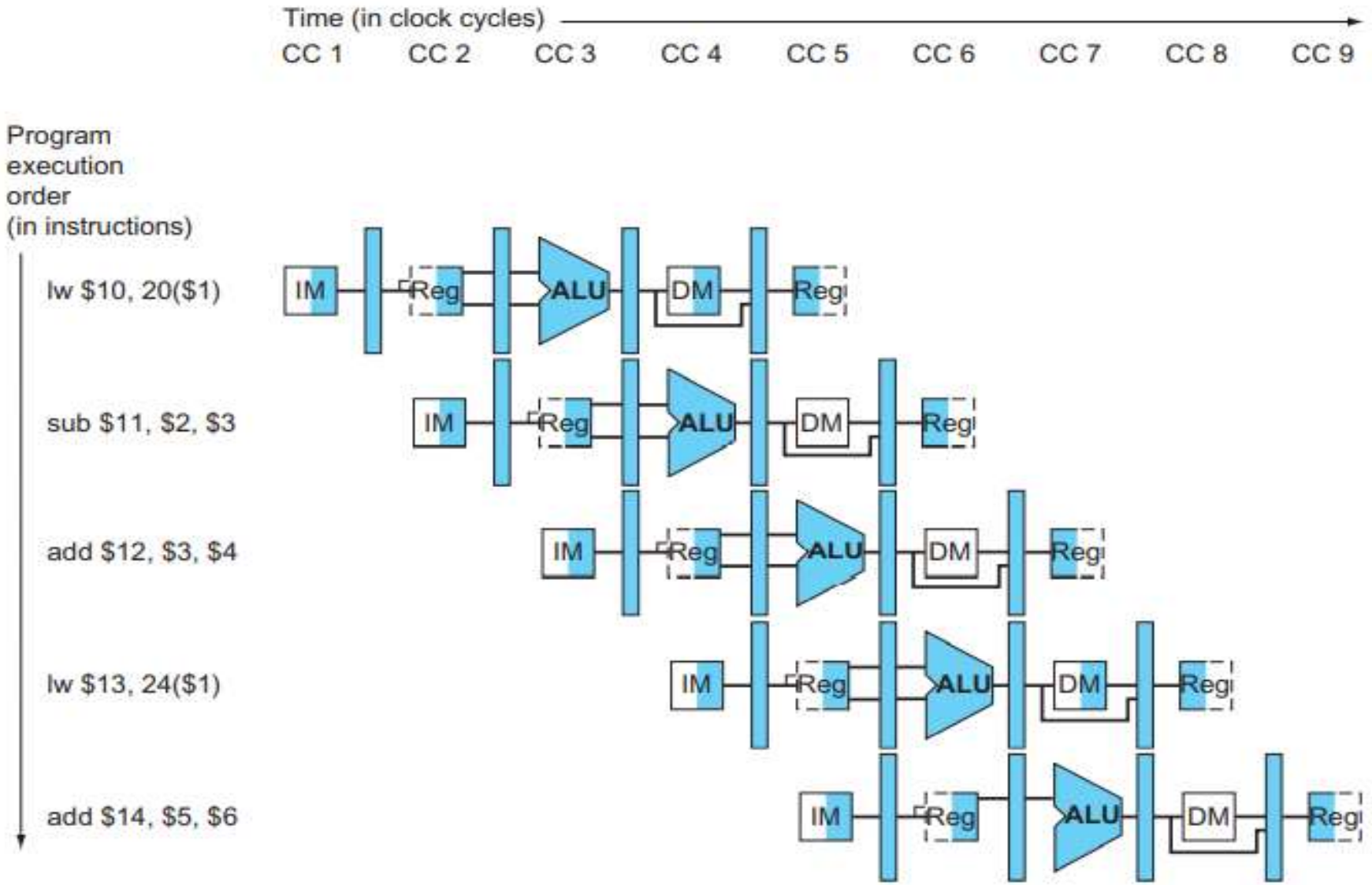


FIGURE 43 Multiple-clock-cycle pipeline diagram of five instructions. This style of pipeline representation shows the complete execution of instructions in a single figure.

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Pipelined Datapath and Control

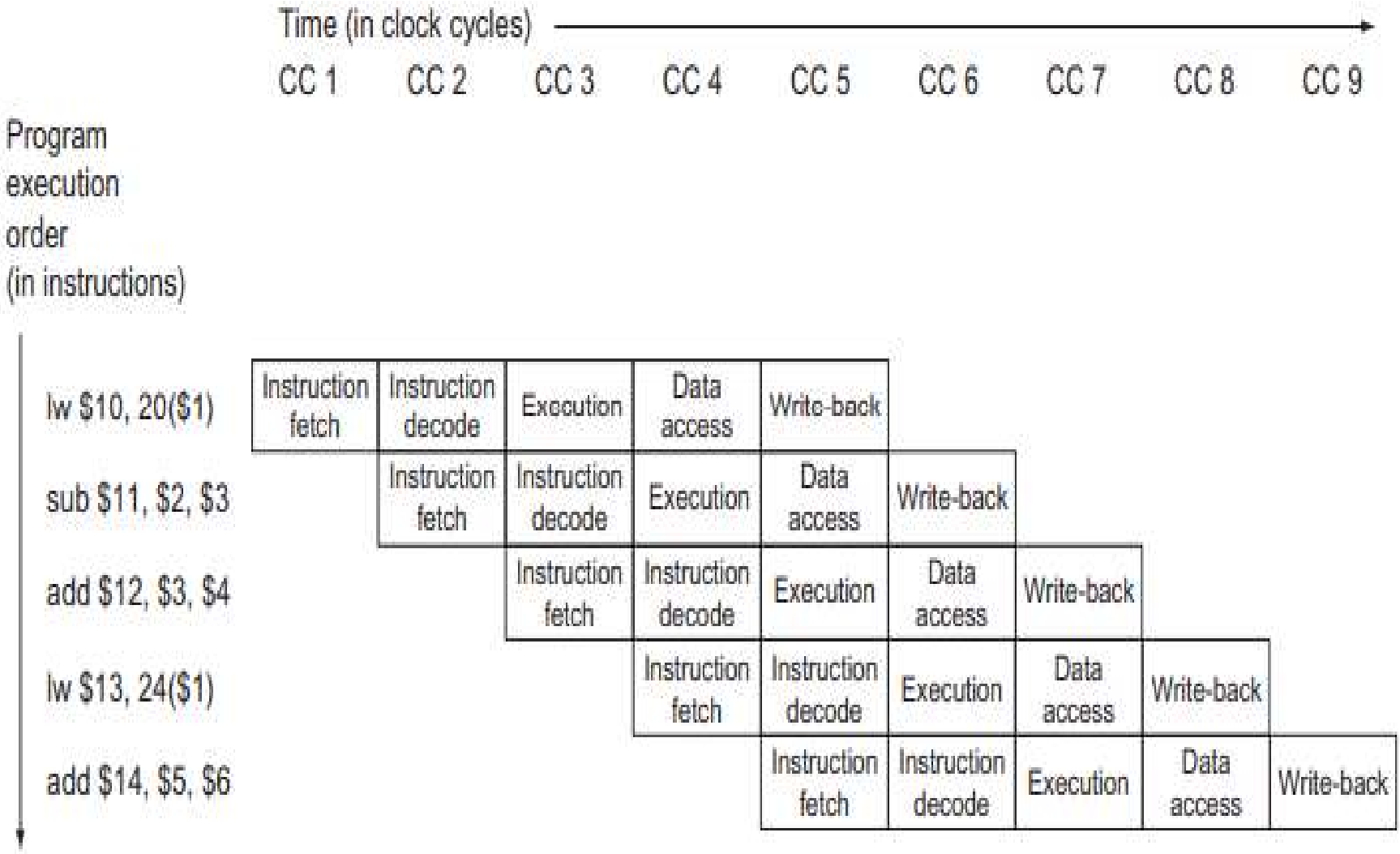


FIGURE 44 Traditional multiple-clock-cycle pipeline diagram of five instructions

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Data Hazards: Forwarding versus Stalling

- Let's look at a sequence with many dependences, shown in color:

```
sub $2, $1,$3 # Register $2 written by sub
and $12,$2,$5 # 1st operand($2) depends on sub
or $13,$6,$2 # 2nd operand($2) depends on sub
add $14,$2,$2 # 1st($2) & 2nd($2) depend on sub
sw $15,100($2) # Base ($2) depends on sub
```

- The last four instructions are all dependent on the result in register $2 of the first instruction.
- If register $2 had the value 10 before the subtract instruction and −20 afterwards, the programmer intends that −20 will be used in the following instructions that refer to register $2.
- How would this sequence perform with our pipeline?

Data Hazards: Forwarding versus Stalling

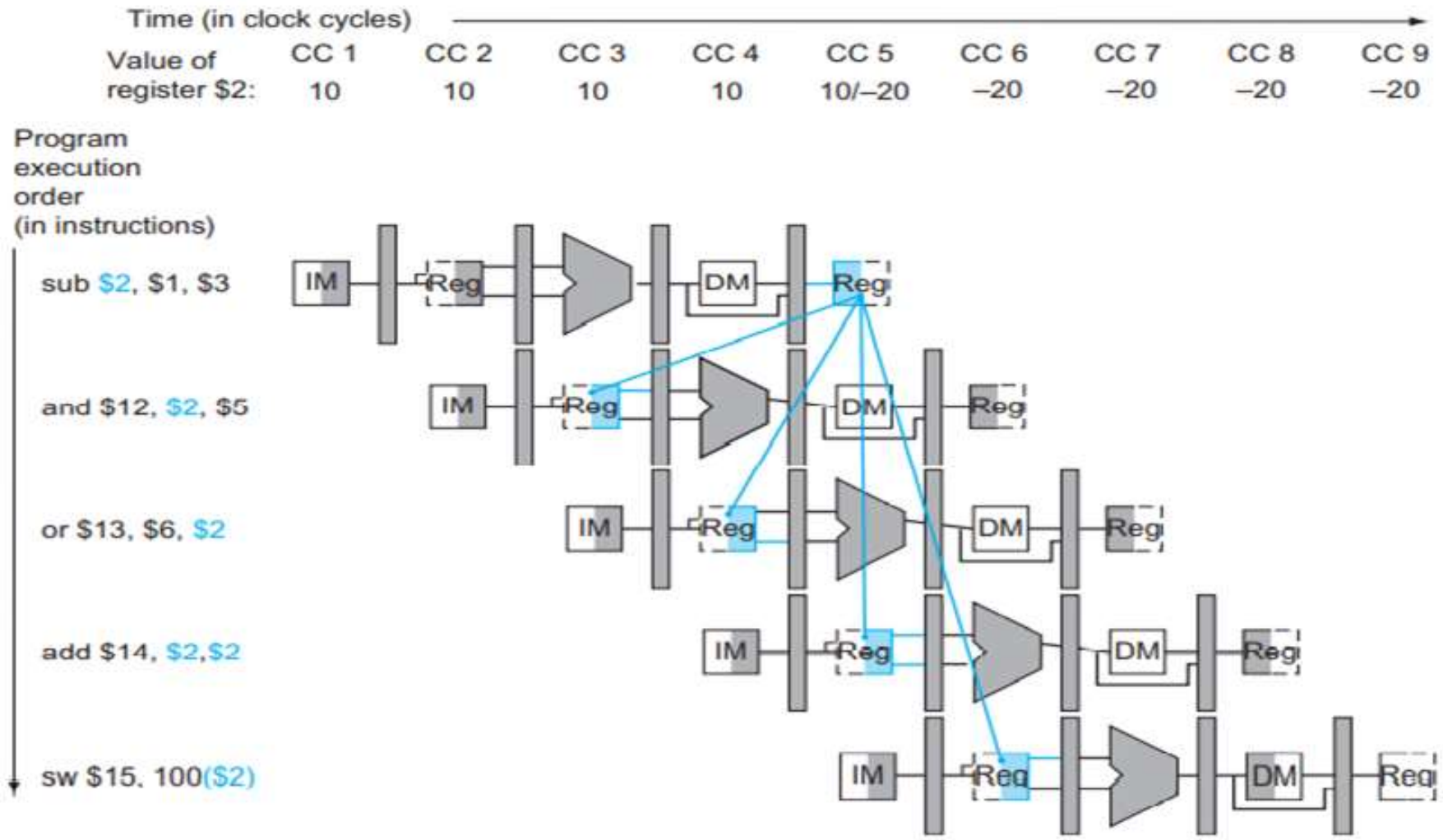


FIGURE 4.5 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences. All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into $2, and all the following instructions read $2. This register is written in clock cycle 5, so the proper value is unavailable before clock cycle 5. (A read of a register during a clock cycle returns the value written at the end of the first half of the cycle, when such a write occurs.) The colored lines from the top datapath to the lower ones show the dependences. Those that must go backward in time are *pipeline data hazards.*

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Control Hazards

- There are also pipeline hazards involving branches.
- Figure 46 shows a sequence of instructions and indicates when the branch would occur in this pipeline.
- An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to branch doesn't occur until the MEM pipeline stage.
- This delay in determining the proper instruction to fetch is called a control hazard or branch hazard.
- Control hazards are relatively simple to understand, they occur less frequently than data hazards, and there is nothing as effective against control hazards as forwarding is against data hazards. Hence, we use simpler schemes.
- There are two schemes for resolving control hazards.

Control Hazards

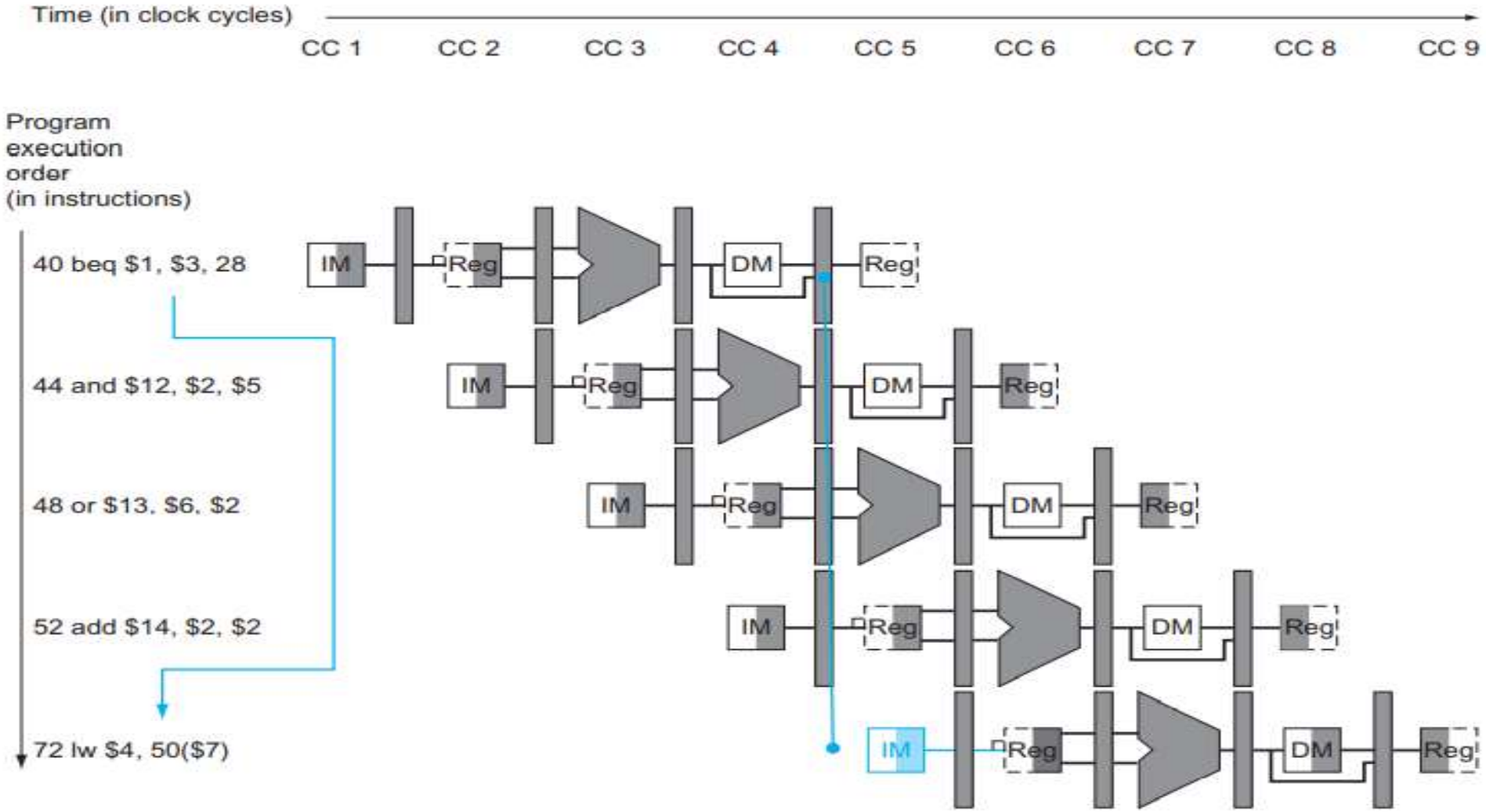


FIGURE 4.6 The impact of the pipeline on the branch instruction. The numbers to the left of the instruction (40, 44, ...) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the `beq` instruction above—the three sequential instructions 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. (Figure 31 assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

[J. Hennessy and D. Patterson, "Computer Organization and Design: The Hardware/Software Interface", 5th Edition.]

Control Hazards

Assume Branch Not Take

- One improvement over branch stalling is to predict that the branch will not be taken and thus continue execution down the sequential instruction stream.
- If the branch is taken, the instructions that are being fetched and decoded must be discarded.
- Execution continues at the branch target.
- If branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.
- Discarding instructions means we must be able to flush instructions in the IF, ID, and EX stages of the pipeline.

Control Hazards

Reducing the Delay of Branches

- One way to improve branch performance is to reduce the cost of the taken branch.
- We have assumed the next PC for a branch is selected in the MEM stage, but if we move the branch execution earlier in the pipeline, then fewer instructions need be flushed.
- The MIPS architecture was designed to support fast single-cycle branches that could be pipelined with a small branch penalty.
- The designers observed that many branches rely only on simple tests (equality or sign, for example) and that such tests do not require a full ALU operation but can be done with at most a few gates.

Control Hazards

Dynamic Branch Prediction

- Assuming a branch is not taken is one simple form of branch prediction.
- In that case, we predict that branches are untaken, flushing the pipeline when we are wrong.
- For the simple five-stage pipeline, such an approach, possibly coupled with compiler based prediction, is probably adequate.
- With deeper pipelines, the branch penalty increases when measured in clock cycles.
- One approach is to look up the address of the instruction to see if a branch was taken the last time this instruction was executed, and, if so, to begin fetching new instructions from the same place as the last time. This technique is called dynamic branch prediction

Control Hazards

Dynamic Branch Prediction

- One implementation of that approach is a branch prediction buffer or branch history table.
- A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction.
- The memory contains a bit that says whether the branch was recently taken or not.