For More Practice

Understanding Pipelines by Drawing Them

6.7 [5] <§6.2> On page 396, we gave the example code sequence:

```
lw   $10, 20($1)
sub  $11, $2, $3
```

Figures 6.19 on page 397 and 6.20 on page 397 showed the multiple-clock-cycle pipeline diagrams for this two-instruction sequence executing across 6 clock cycles. Figures 6.14.1 through 6.14.3 show the corresponding single-clock-cycle pipeline diagrams for these two instructions. Note that the order of the instructions differs between these two types of diagrams: the newest instruction is at the *bottom and to the right* of the multiple-clock-cycle pipeline diagram, and it is on the *left* in the single-clock-cycle pipeline diagram.

In the following three exercises, use the following code sequence:

```
add   $4, $2, $3
sw    $5, 4($2)
```

For the above code sequence, draw the multiple-clock-cycle pipeline diagram using the format shown in Figure 6.19 on page 397.

6.8 [5] <§6.2> For the code sequence in Exercise 6.7, draw the multiple-clock-cycle pipeline diagram using the format shown in Figure 6.20 on page 397.

6.9 [15] <§6.2> For the above code sequence show the pipeline over 6 clock cycles using the single-clock-cycle diagrams, as in Figures 6.14.1 through 6.14.3. Figure 6.14.4 has a blank single-clock-cycle pipeline diagram that may be reproduced to ease your task!

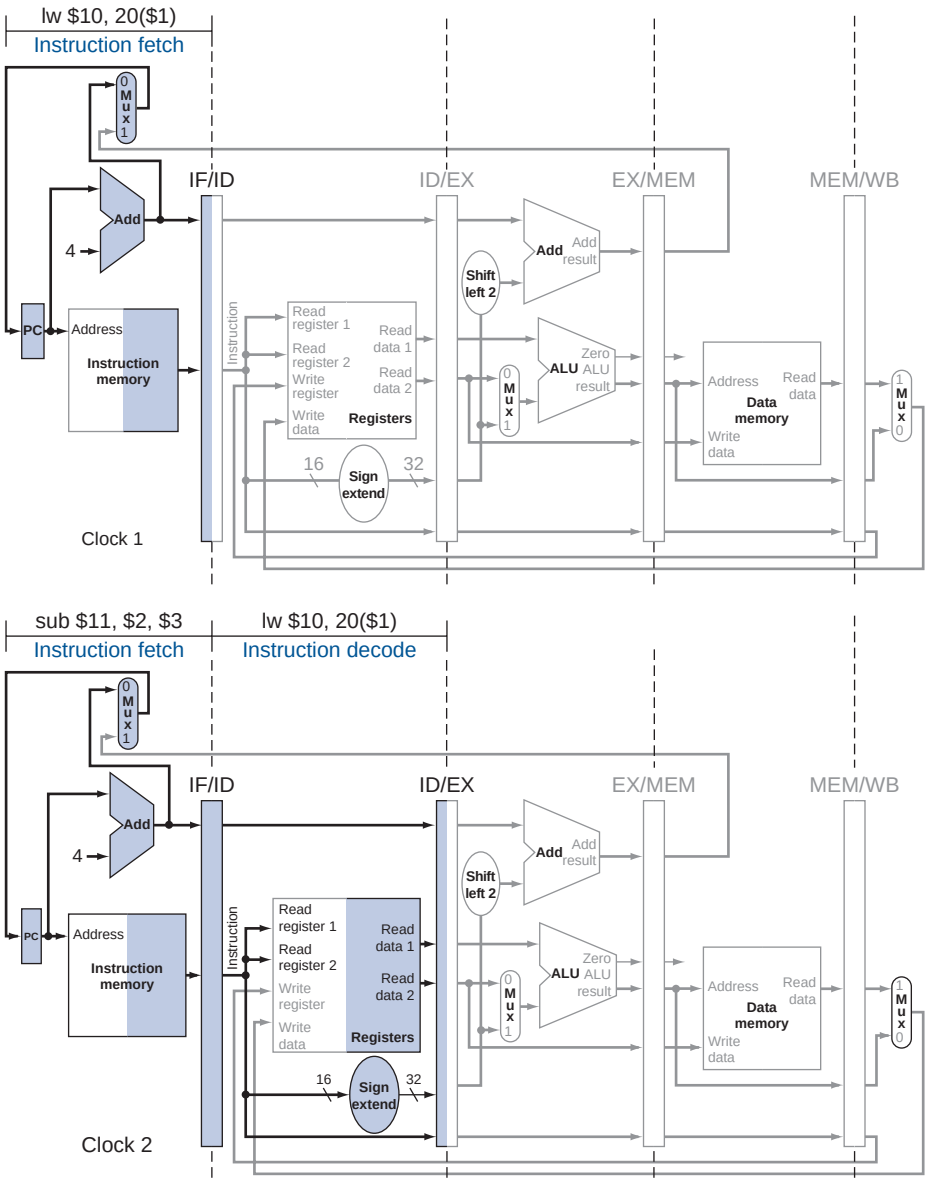

FIGURE 6.14.1 Single-cycle pipeline diagrams for clock cycles 1 (top diagram) and 2 (bottom diagram). This style of pipeline representation is a snapshot of every instruction executing during 1 clock cycle. Our example has but two instructions, so at most two stages are identified in each clock cycle; normally, all five stages are occupied. The highlighted portions of the datapath are active in that clock cycle. The load is fetched in clock cycle 1 and decoded in clock cycle 2, with the subtract fetched in the second clock cycle. To make the figures easier to understand, the other pipeline stages are empty, but normally there is an instruction in every pipeline stage.

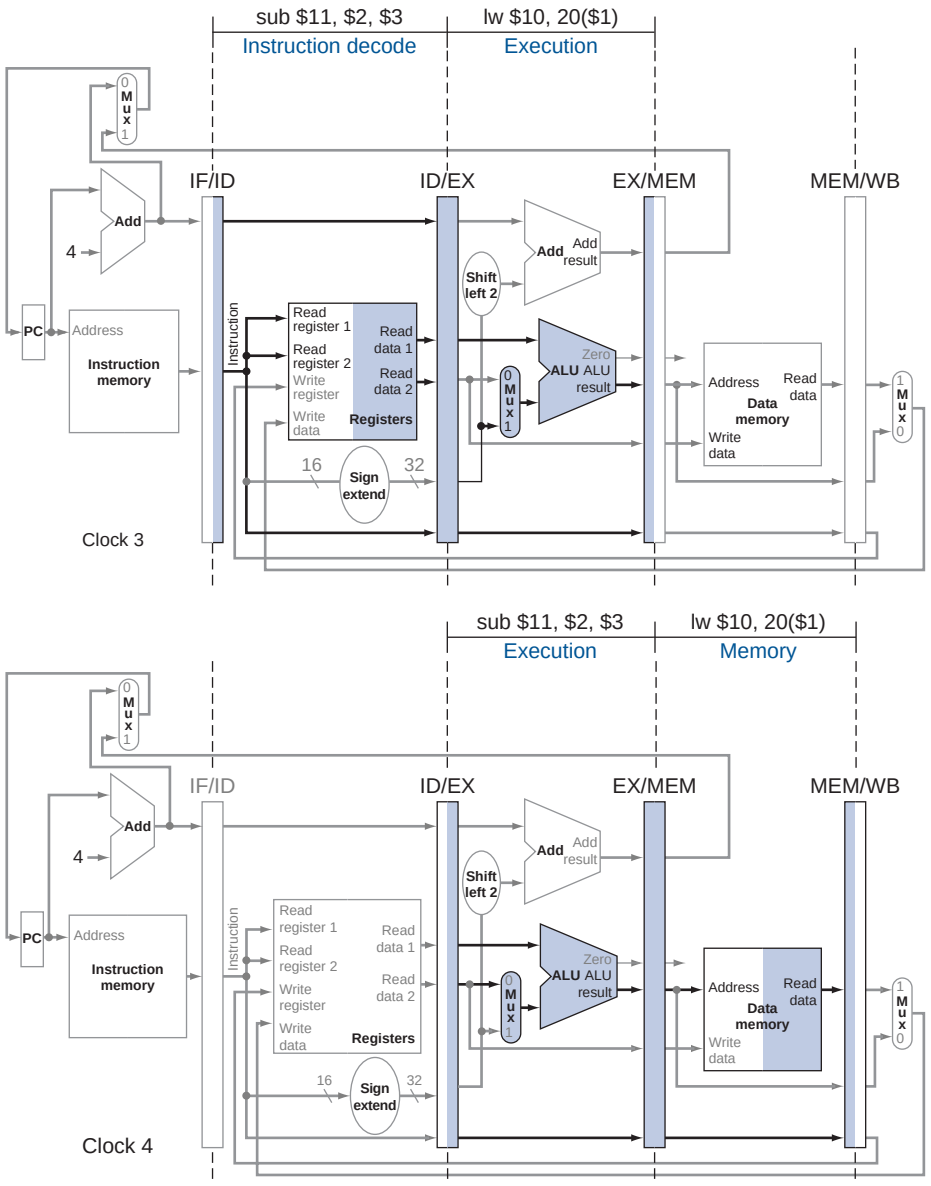

FIGURE 6.14.2 Single-cycle pipeline diagrams for clock cycles 3 (top diagram) and 4 (bottom diagram). In the third clock cycle in the top diagram, `lw` enters the EX stage. At the same time, `sub` enters ID. In the fourth clock cycle (bottom datapath), `lw` moves into MEM stage, reading memory using the address found in EX/MEM at the beginning of clock cycle 4. At the same time, the ALU subtracts and then places the difference into EX/MEM at the end of the clock cycle.

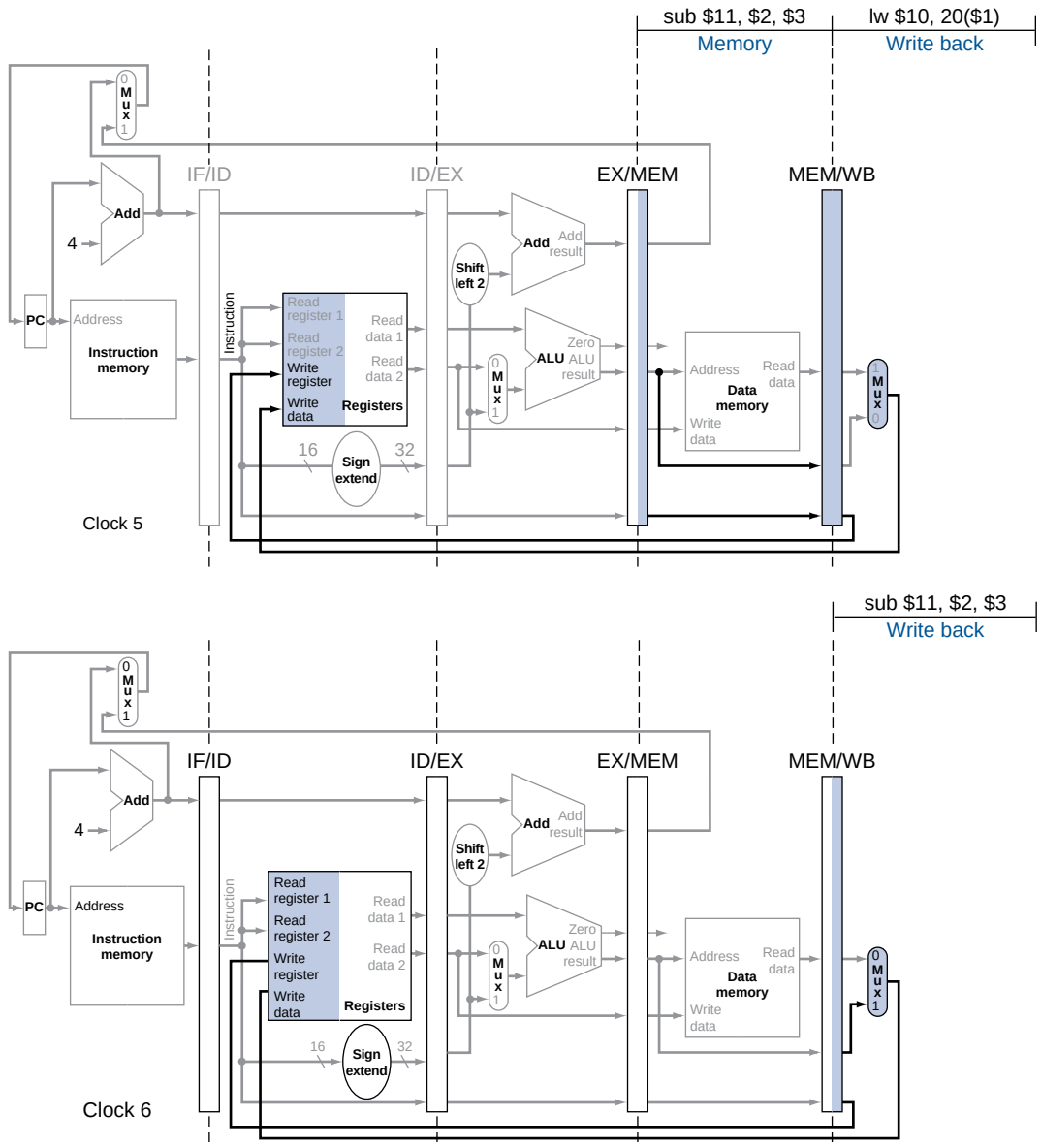

FIGURE 6.14.3 Single-cycle pipeline diagrams for clock cycles 5 (top diagram) and 6 (bottom diagram). In clock cycle 5, `lw` completes by writing the data in MEM/WB into register 10, and `sub` sends the difference in EX/MEM to MEM/WB. In the next clock cycle, `sub` writes the value in MEM/WB to register 11.

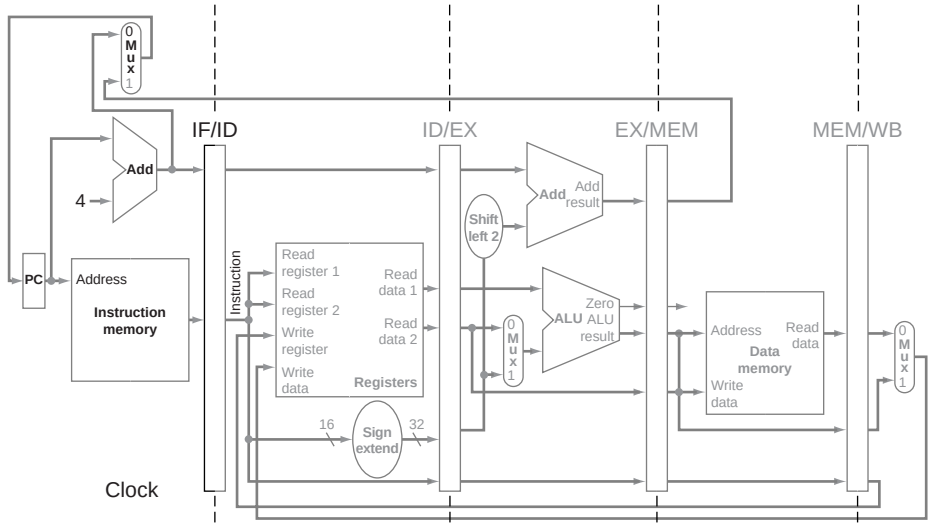

FIGURE 6.14.4 A blank single-clock-cycle pipeline diagram for use in illustrating pipeline execution.

Labeling Pipeline Diagrams with Control

6.15 [20] <§6.3> To understand how pipeline control works, let's consider these five instructions going through the pipeline:

```
lw     $10, 20($1)
sub    $11, $2, $3
and    $12, $4, $5
or     $13, $6, $7
add    $14, $8, $9
```

Show the instructions in the pipeline that precede the `lw` as before <1>, before <2>, . . . , and the instructions after the `add` as after <1>, after <2>, . . . Figures 6.14.5 through 6.14.9 show these instructions proceeding through the nine clock cycles it takes them to complete execution, highlighting what is active in a stage and identifying the instruction associated with each stage during a clock cycle. Reviewing these figures carefully will give you insight into how pipelines work. A few items you may notice:

- In Figure 6.14.7 you can see the sequence of the destination register numbers from left to right at the bottom of the pipeline registers. The numbers advance to the right during each clock cycle, with the MEM/WB pipeline register supplying the number of the register written during the WB stage.
- When a stage is inactive, the values of control lines that are deasserted are shown as 0 or X (for don't care).
- In contrast to Chapter 5, where sequencing of control required special hardware, sequencing of control is embedded in the pipeline structure itself. First, all instructions take the same number of clock cycles, so there is no special control for instruction duration. Second, all control information is computed during instruction decode, and then passed along by the pipeline registers.

Using the same format as Figure 6.14.5, and starting with the blank pipelining diagram in Figure 6.14.10, draw the pipeline diagrams for the above sequence for a total of 4 clock cycles.

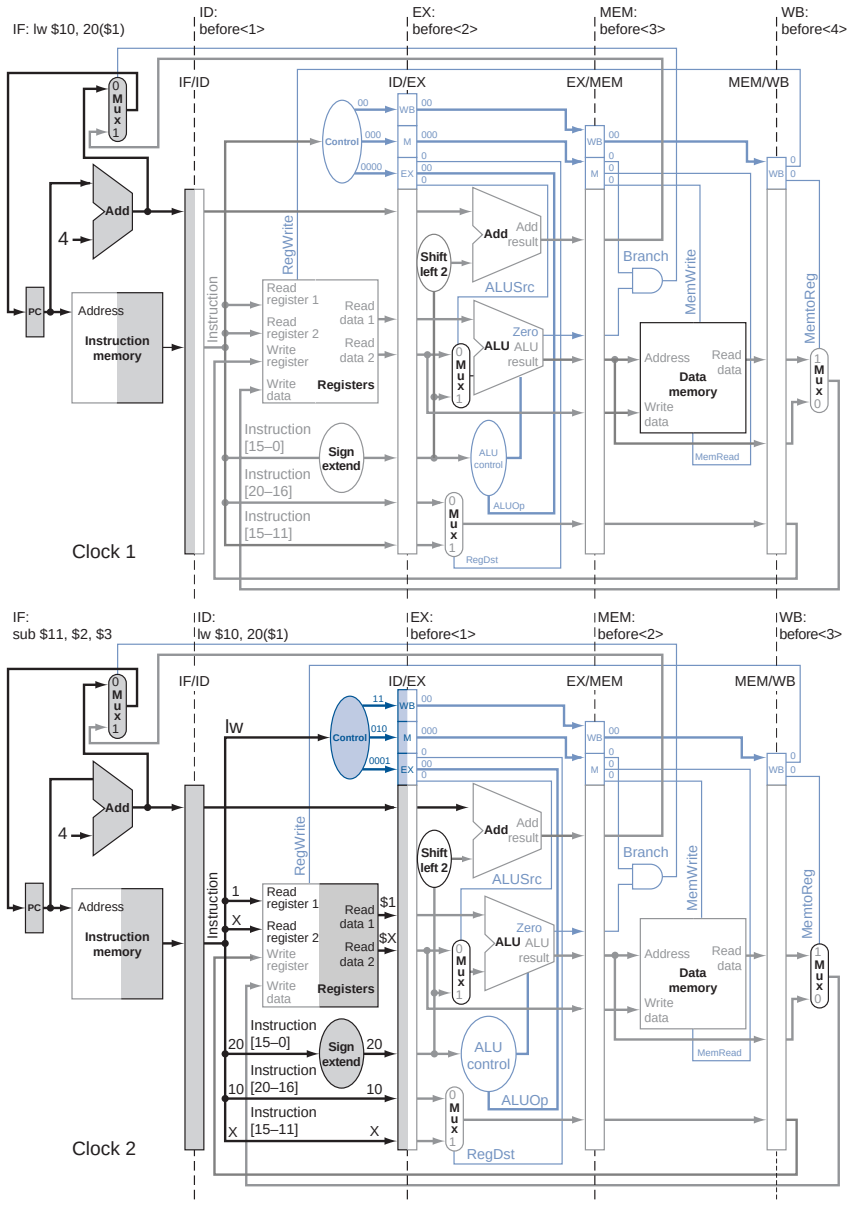

FIGURE 6.14.5 Clock cycles 1 and 2. The phrase "before<*i*>" means the ith instruction before `lw`. The `lw` instruction in the top datapath is in the IF stage. At the end of the clock cycle, the `lw` instruction is in the IF/ID pipeline registers. In the second clock cycle, seen in the bottom datapath, the `lw` moves to the ID stage, and `sub` enters in the IF stage. Note that the values of the instruction fields and the selected source registers are shown in the ID stage. Hence register `$1` and the constant 20, the operands of `lw`, are written into the ID/EX pipeline register. The number 10, representing the destination register number of `lw`, is also placed in ID/EX. Bits 15–11 are 0, but we use *X* to show that a field plays no role in a given instruction. The top of the ID/EX pipeline register shows the control values for `lw` to be used in the remaining stages. These control values can be read from the `lw` row of the table in Figure 6.25 on page 401.

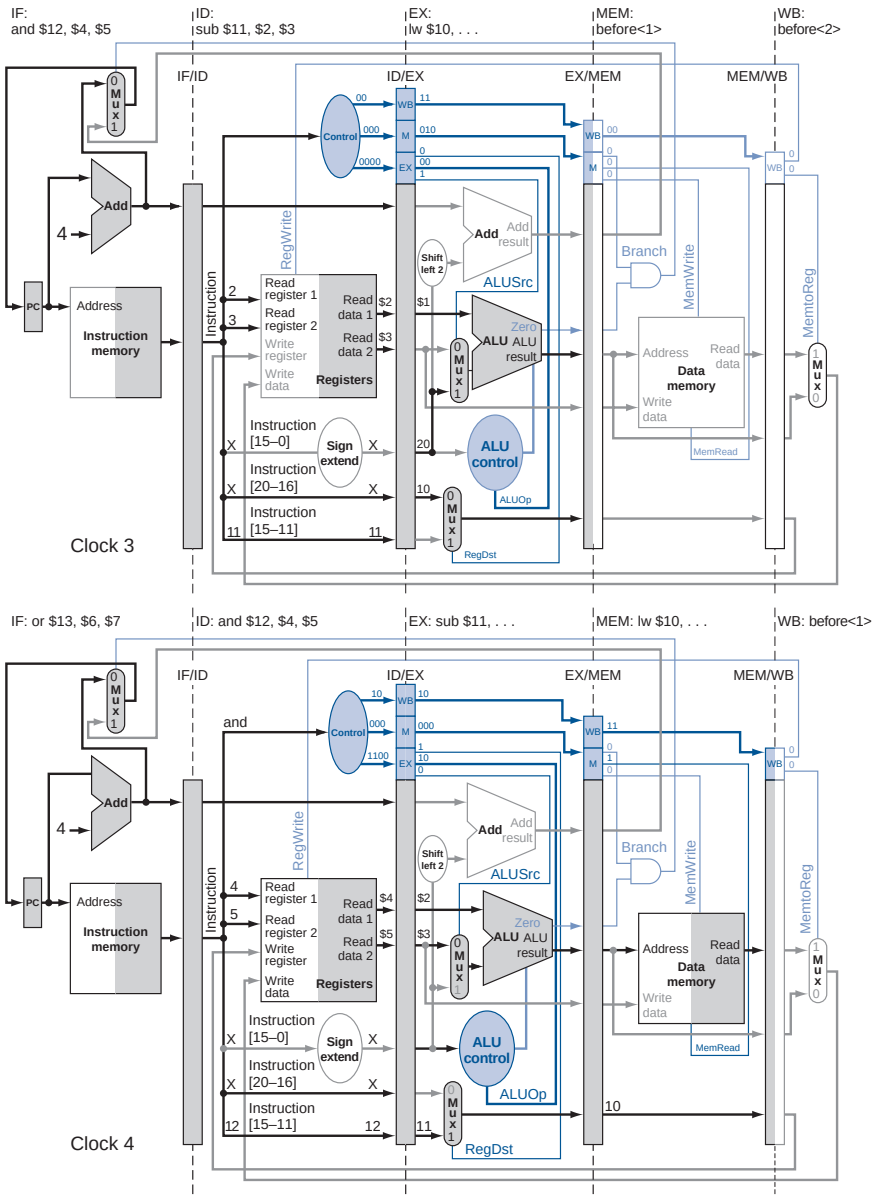

FIGURE 6.14.6 Clock cycles 3 and 4. In the top diagram, `lw` enters the EX stage in the third clock cycle, adding `$1` and 20 to form the address in the EX/MEM pipeline register. (The `lw` instruction is written `lw $10, . . .` upon reaching EX because the identity of instruction operands is not needed by EX or the subsequent stages. In this version of the pipeline, the actions of EX, MEM, and WB depend only on the instruction and its destination register or its target address.) At the same time, `sub` enters ID, reading registers `$2` and `$3`, and the `and` instruction starts IF. In the fourth clock cycle (bottom datapath), `lw` moves into MEM stage, reading memory using the value in EX/MEM as the address. In the same clock cycle, the ALU subtracts `$3` from `$2` and places the difference into EX/MEM, `and` reads registers `$4` and `$5` during ID, and the `or` instruction enters IF. The two diagrams show the control signals being created in the ID stage and peeled off as they are used in subsequent pipe stages.

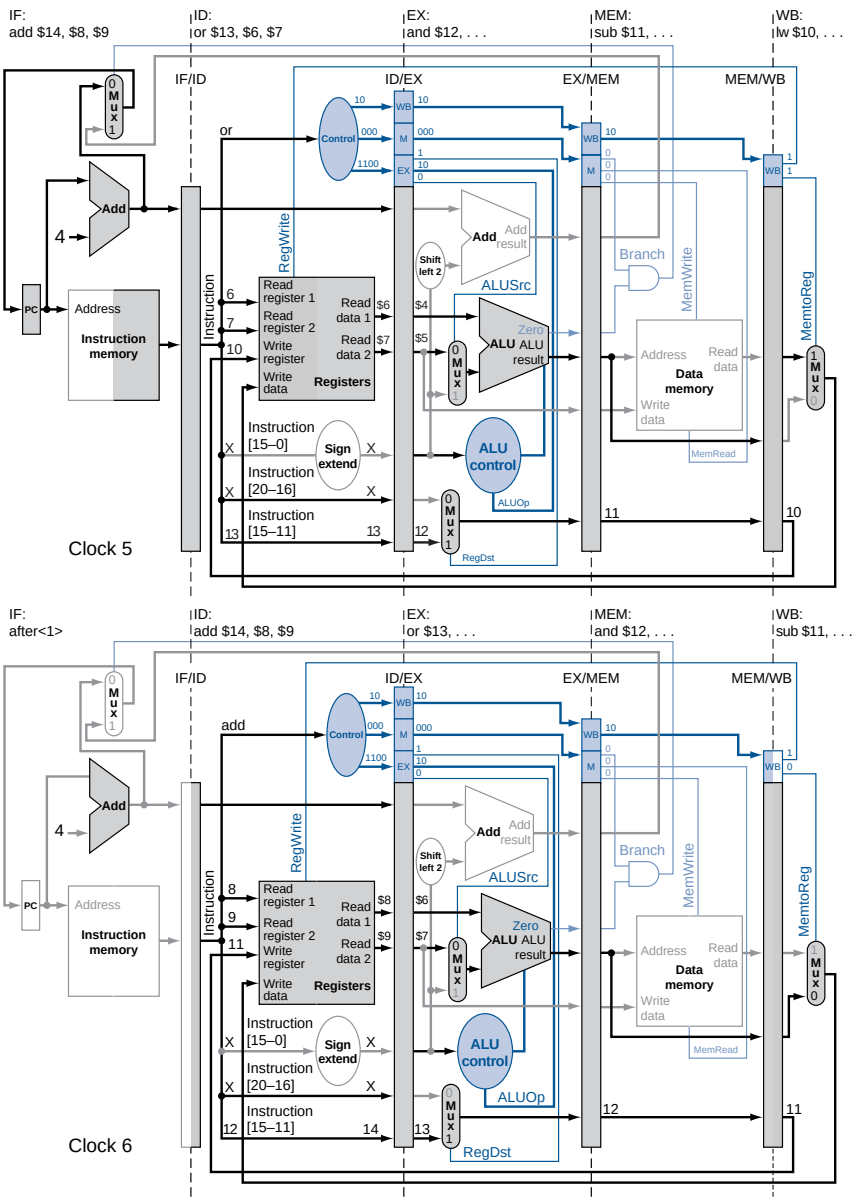

FIGURE 6.14.7 Clock cycles 5 and 6. With `add`, the final instruction in this example, entering IF in the top datapath, all instructions are engaged. By writing the data in MEM/WB into register 10, `lw` completes; both the data and the register number are in MEM/WB. In the same clock cycle, `sub` sends the difference in EX/MEM to MEM/WB, and the rest of the instructions move forward. In the next clock cycle, `sub` selects the value in MEM/WB to write to register number 11, again found in MEM/WB. The remaining instructions play follow-the-leader: the ALU calculates the OR of `$6` and `$7` for the `or` instruction in the EX stage, and registers `$8` and `$9` are read in the ID stage for the `add` instruction. The instructions after `add` are shown as inactive just to emphasize what occurs for the five instructions in the example. The phrase"after<*i*>" means the ith instruction after `add`.

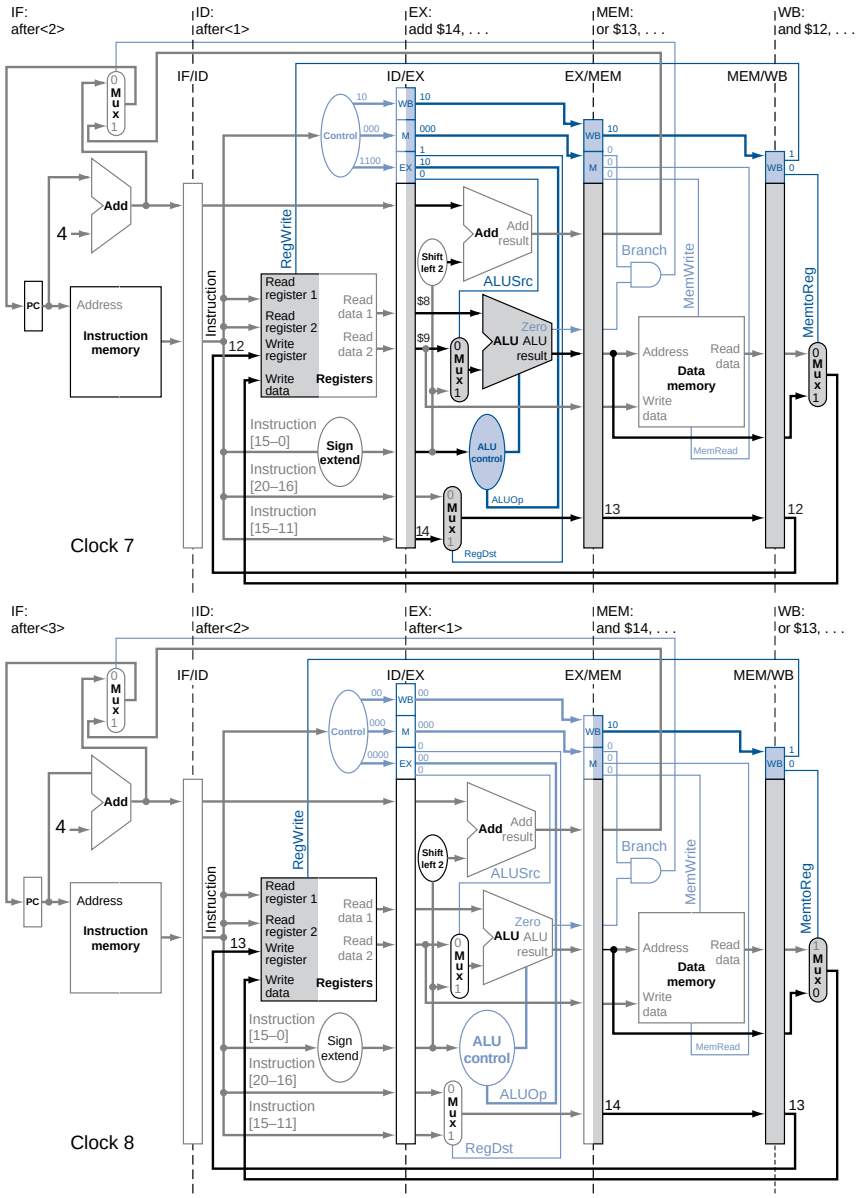

FIGURE 6.14.8 Clock cycles 7 and 8. In the top datapath, the `add` instruction brings up the rear, adding the values corresponding to registers `$8` and `$9` during the EX stage. The result of the `or` instruction is passed from EX/MEM to MEM/WB in the MEM stage, and the WB stage writes the result of the `and` instruction in MEM/WB to register `$12`. Note that the control signals are deasserted (set to 0) in the ID stage, since no instruction is being executed. In the following clock cycle (lower drawing), the WB stage writes the result to register `$13`, thereby completing `or`, and the MEM stage passes the sum from the `add` in EX/MEM to MEM/WB. The instructions after `add` are shown as inactive for pedagogical reasons.

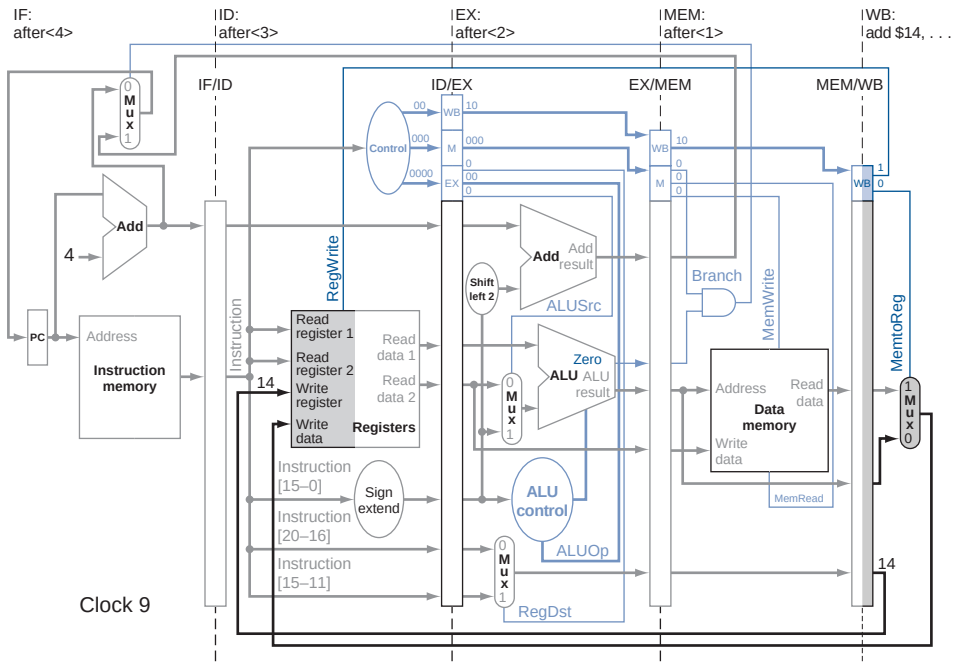

FIGURE 6.14.9 Clock cycle 9. The WB stage writes the sum in MEM/WB into register `$14`, completing `add` and the five-instruction sequence. The instructions after `add` are shown as inactive for pedagogical reasons.

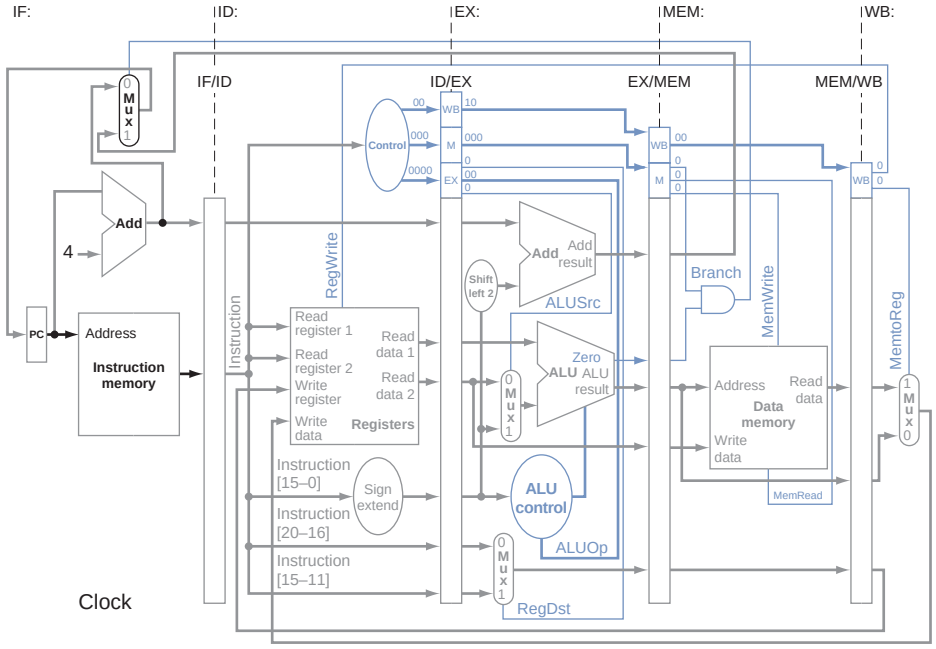

FIGURE 6.14.10 A blank single-clock-cycle pipeline diagram with control.

Illustrating Pipelines with Forwarding

6.16 [20] <§6.4> We can use the single-clock-cycle pipeline diagrams to show how forwarding operates, as well as how the control activates the forwarding paths. Consider the following code sequence in which the dependencies have been highlighted:

```
sub    $2, $1, $3
and    $4, $2, $5
or     $4, $4, $2
add    $9, $4, $2
```

Figures 6.14.11 and 6.14.12 show the events in clock cycles 3–6 in the execution of these instructions. In clock cycle 4, the forwarding unit sees the writing by the `sub` instruction of register `$2` in the MEM stage, while the `and` instruction in the EX stage is reading register `$2`. The forwarding unit selects the EX/MEM pipeline register instead of the ID/EX pipeline register as the upper input to the ALU to get the proper value for register `$2`. The following `or` instruction reads register `$4`, which is written by the `and` instruction, and register `$2`, which is written by the `sub` instruction. Thus in clock cycle 5 the forwarding unit selects the EX/MEM pipeline register for the upper input to the ALU and the MEM/WB pipeline register for the lower input to the ALU. The following `add` instruction reads both register `$4`, the target of the `and` instruction, and register `$2`, which the `sub` instruction has already written. Notice that the prior two instructions both write register `$4`, so the forwarding unit must pick the immediately preceding one (MEM stage). In clock cycle 6, the forwarding unit thus selects the EX/MEM pipeline register, containing the result of the `or` instruction, for the upper ALU input but uses the nonforwarding register value for the lower input to the ALU. Using the blank diagram in Figure 6.14.13 and the format of Figure 6.14.11, draw the single-clock-cycle pipeline diagrams for clock cycles 3 through 5 when executing this sequence:

```
add    $1, $1, $3
add    $4, $2, $1
and    $5, $4, $1
```

Illustrating Pipelines with Stalls and Fowarding

6.24 [20] <§6.5> We can use the single-clock-cycle pipeline diagrams to show how the control for stalls works. Figures 6.14.14 to 6.14.16 show the single-cycle

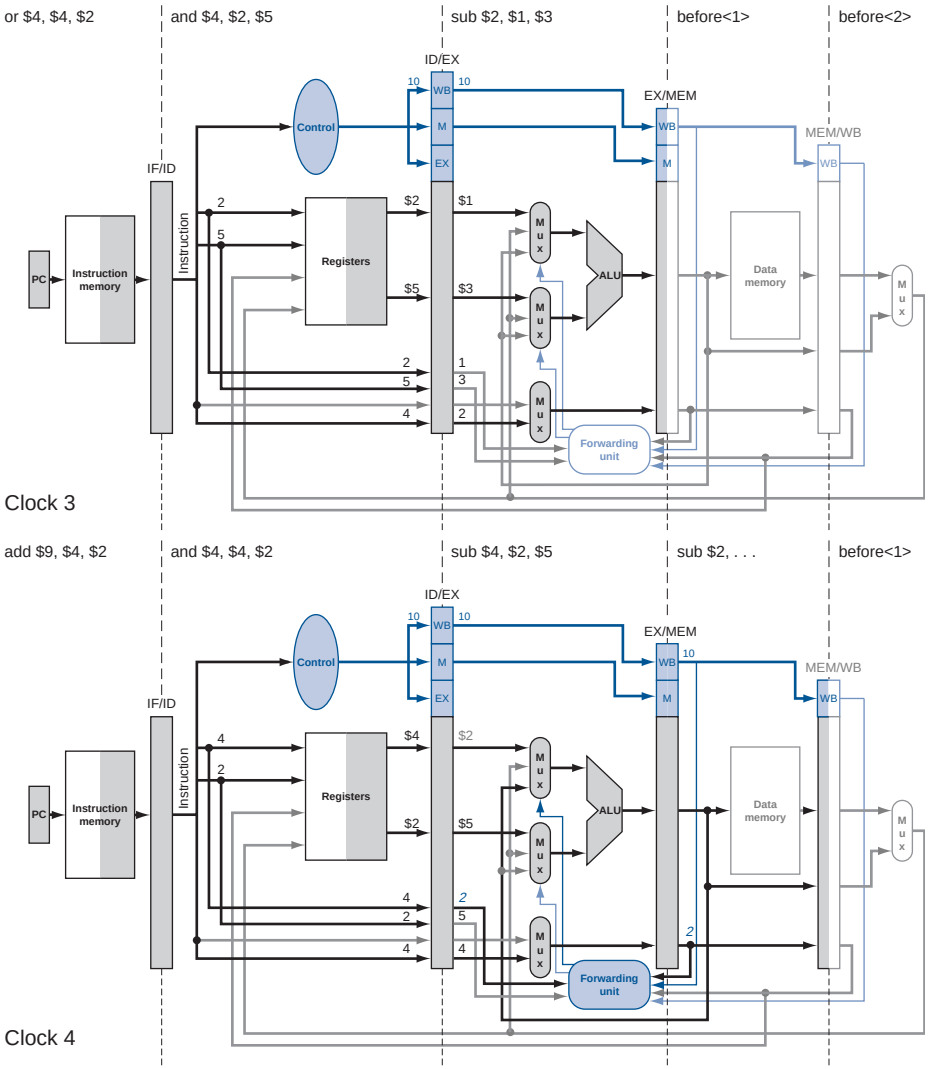

FIGURE 6.14.11 Clock cycles 3 and 4 of the instruction sequence in Exercise 6.16. The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard. The forwarding unit is highlighted by shading it when it is forwarding data to the ALU. The instructions before `sub` are shown as inactive just to emphasize what occurs for the four instructions in the example. Operand names are used in EX for control of forwarding; thus they are included in the instruction label for EX. Operand names are not needed in MEM or WB, so . . . is used. Compare this with Figures 6.14.6 through 6.14.9 showing the datapath without forwarding where ID is the last stage to need operand information.

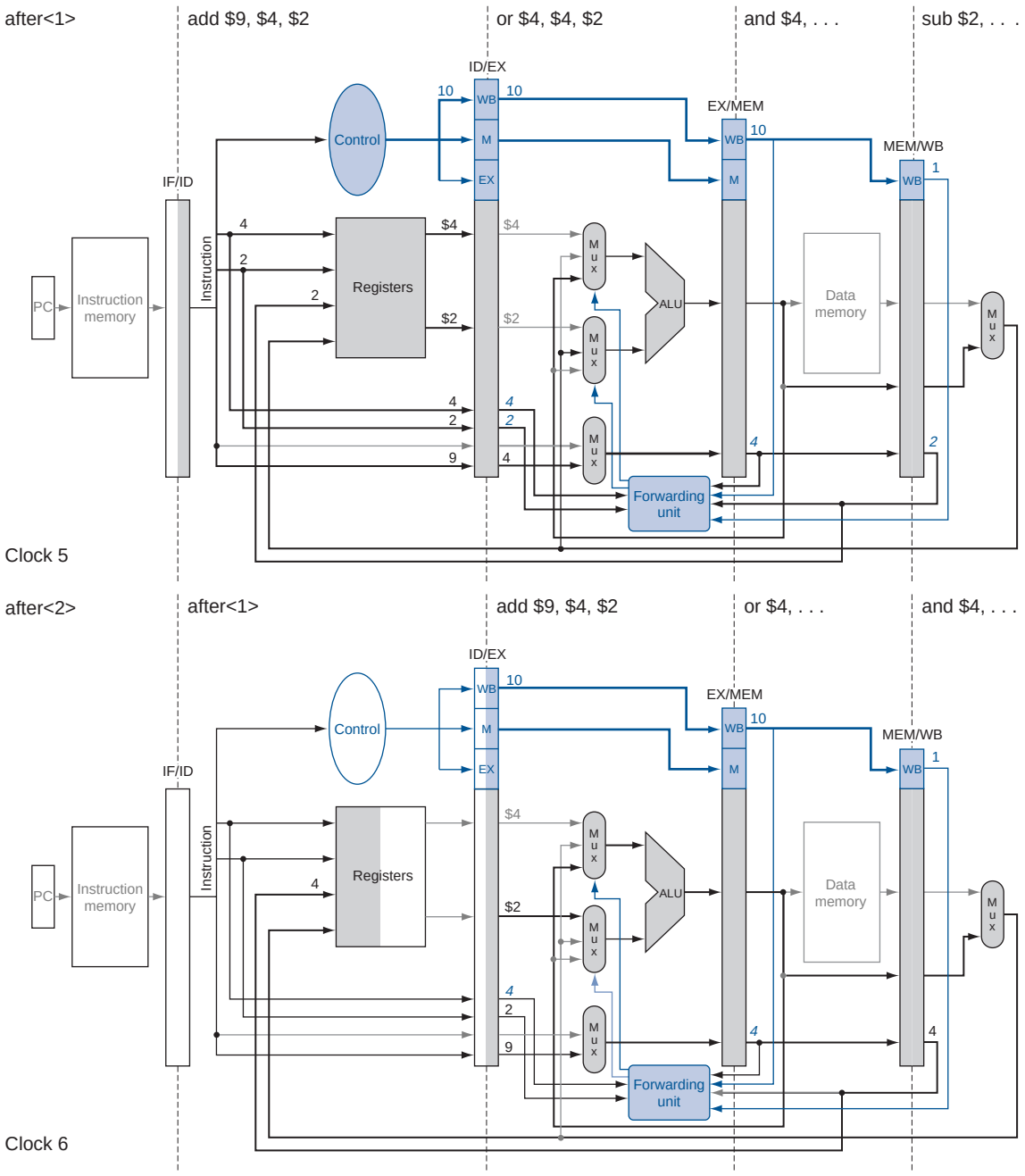

FIGURE 6.14.12 Clock cycles 5 and 6 of the instruction sequence in Exercise 6.16. The forwarding unit is highlighted when it is forwarding data to the ALU. The two instructions after `add` are shown as inactive just to emphasize what occurs for the four instructions in the example. The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard.

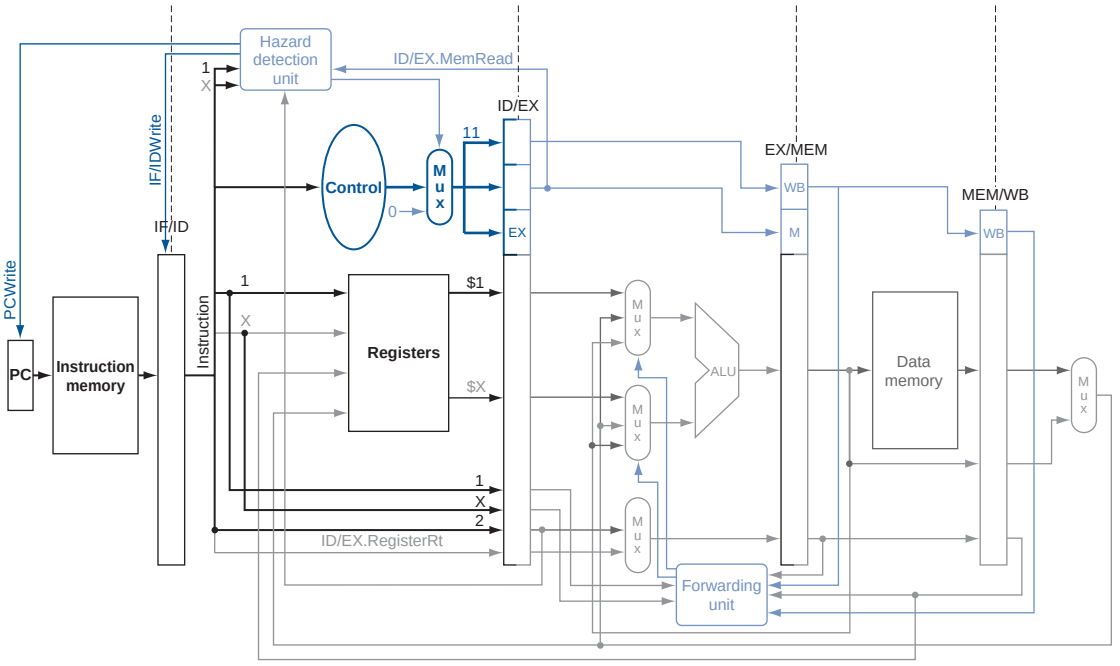

FIGURE 6.14.13 Empty pipeline diagram with forwarding datapaths.

diagram for clocks 2 to 7 for the following code sequence (dependences highlighted):

```
lw      $2, 20($1)
and     $4, $2,$5
or      $4, $4,$2
add     $9, $4,$2
```

Using the same format as Figure 6.14.15, illustrate the execution of the following sequence for clock cycles 3 through 6. Use Figure 6.14.17 in doing the exercise.

```
lw      $2, 20($1)
add     $4, $5, $2
sub     $4, $4, $2
```

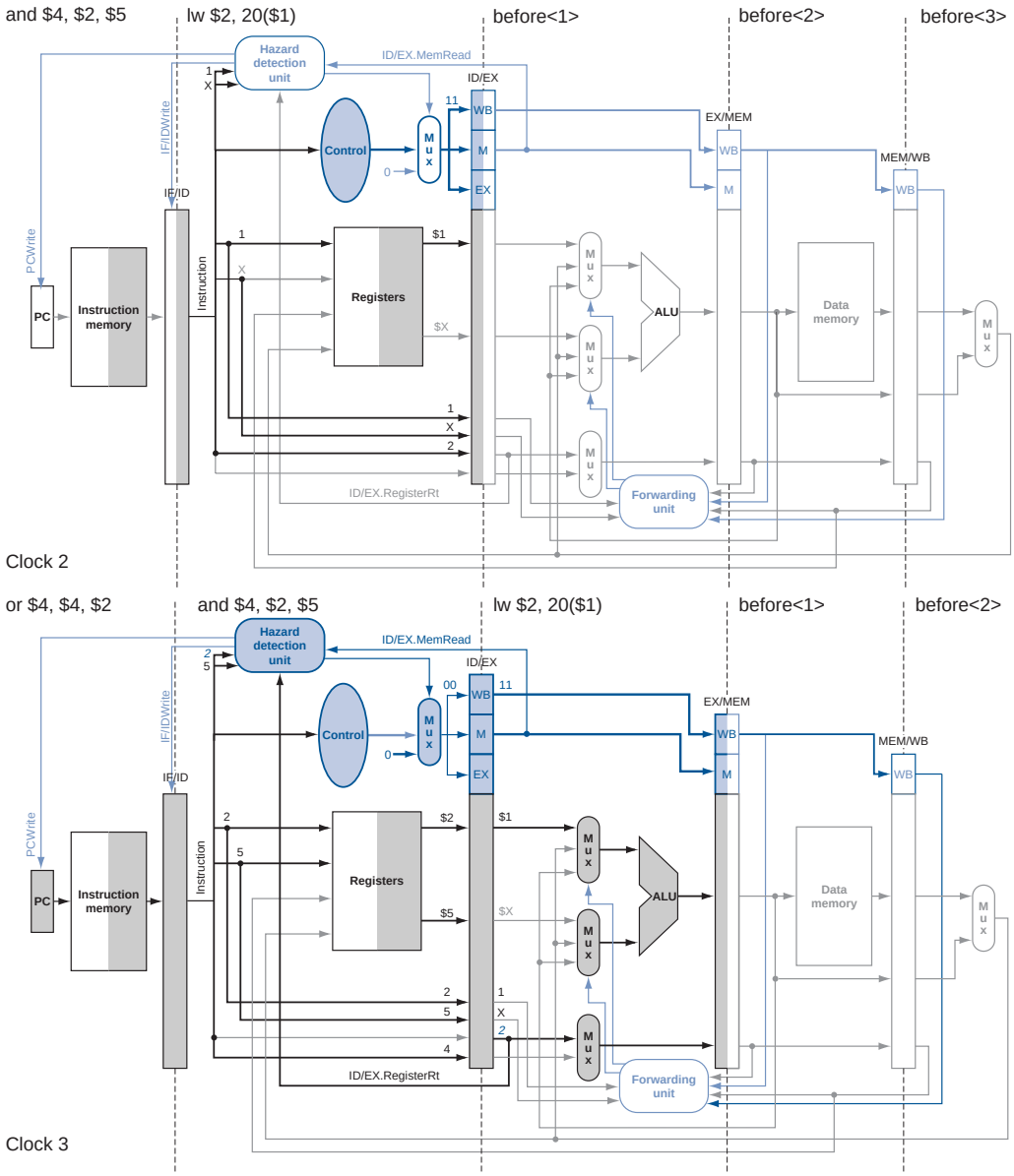

FIGURE 6.14.14 Clock cycles 2 and 3 of the instruction sequence in Exercise 6.24 with a load replacing `sub`. The bold lines are those active in a clock cycle, the italicized register numbers in color indicate a hazard, and the. . . in the place of operands means that their identity is information not needed by that stage. The values of the significant control lines, registers, and register numbers are labeled in the figures. The `and` instruction wants to read the value created by the `lw` instruction in clock cycle 3, so the hazard detection unit stalls the `and` and `or` instructions. Hence, the hazard detection unit is highlighted.

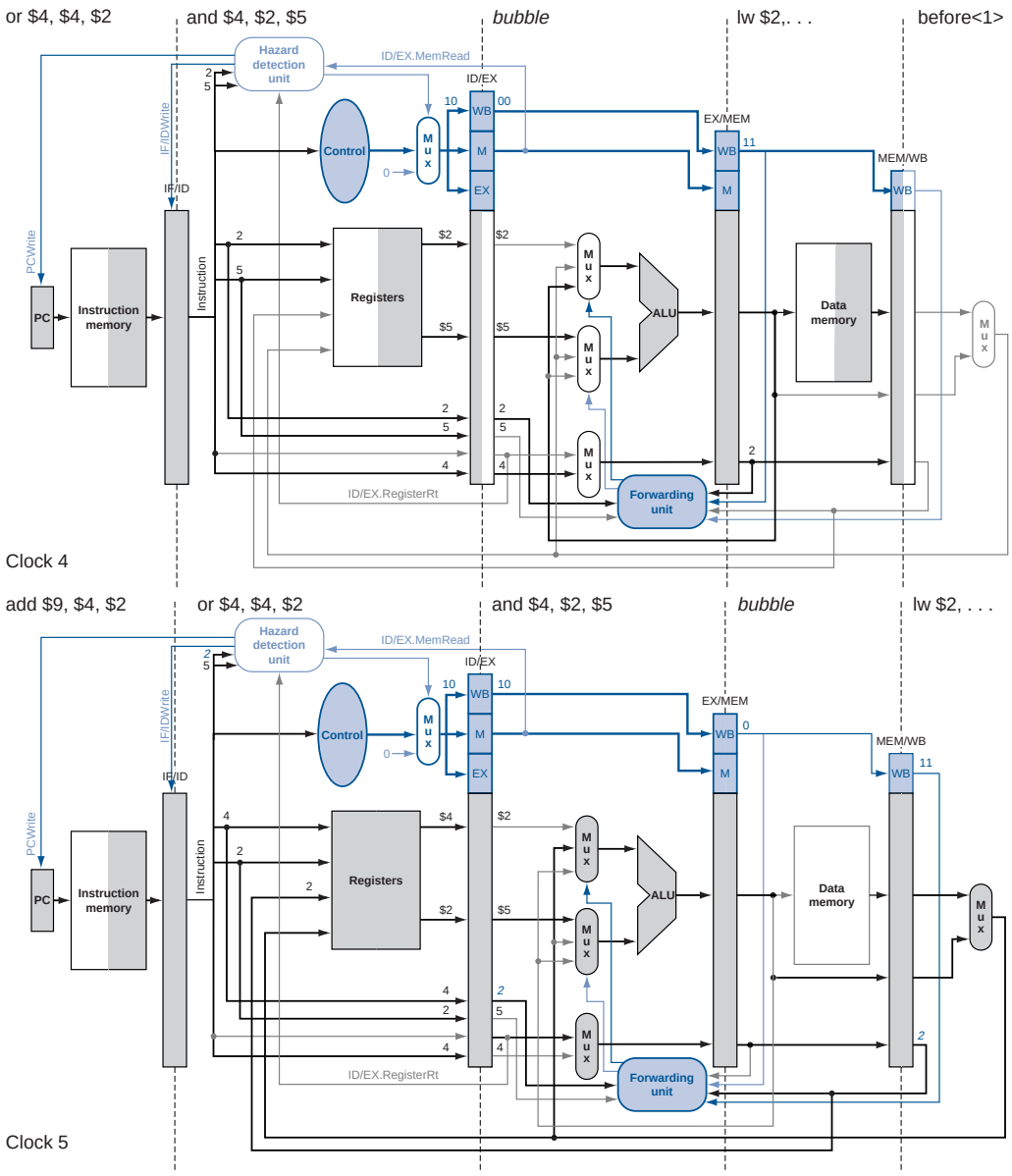

FIGURE 6.14.15 Clock cycles 4 and 5 of the instruction sequence in Exercise 6.24 with a load replacing `sub`. The bubble is inserted in the pipeline in clock cycle 4, and then the `and` instruction is allowed to proceed in clock cycle 5. The forwarding unit is highlighted in clock cycle 5 because it is forwarding data from `lw` to the ALU. Note that in clock cycle 4 the forwarding unit forwards the address of the `lw` as if it were the contents of register `$2`; this is rendered harmless by the insertion of the bubble. The bold lines are those active in a clock cycle, and the italicized register numbers in color indicate a hazard.

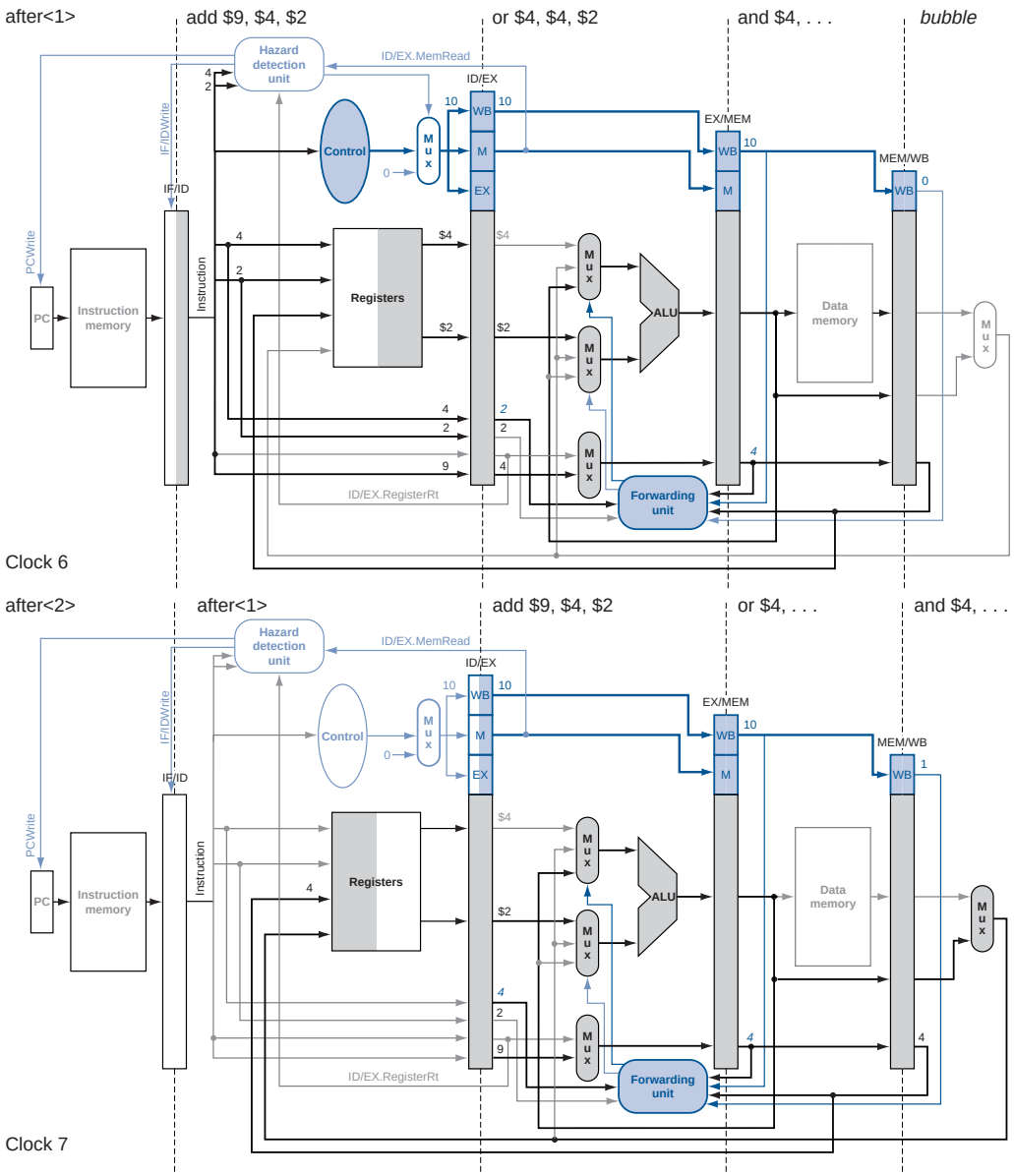

FIGURE 6.14.16 Clock cycles 6 and 7 of the instruction sequence in Exercise 6.24 with a load replacing `sub`. Note that unlike in Figure 6.14.12, the stall allows the `lw` to complete, and so there is no forwarding from MEM/WB in clock cycle 6. Register `$4` for the `add` in the EX stage still depends on the result from `or` in EX/MEM so the forwarding unit passes the result to the ALU. The bold lines show ALU input lines active in a clock cycle, and the italicized register numbers indicate a hazard. The instructions after `add` are shown as inactive for pedagogical reasons.

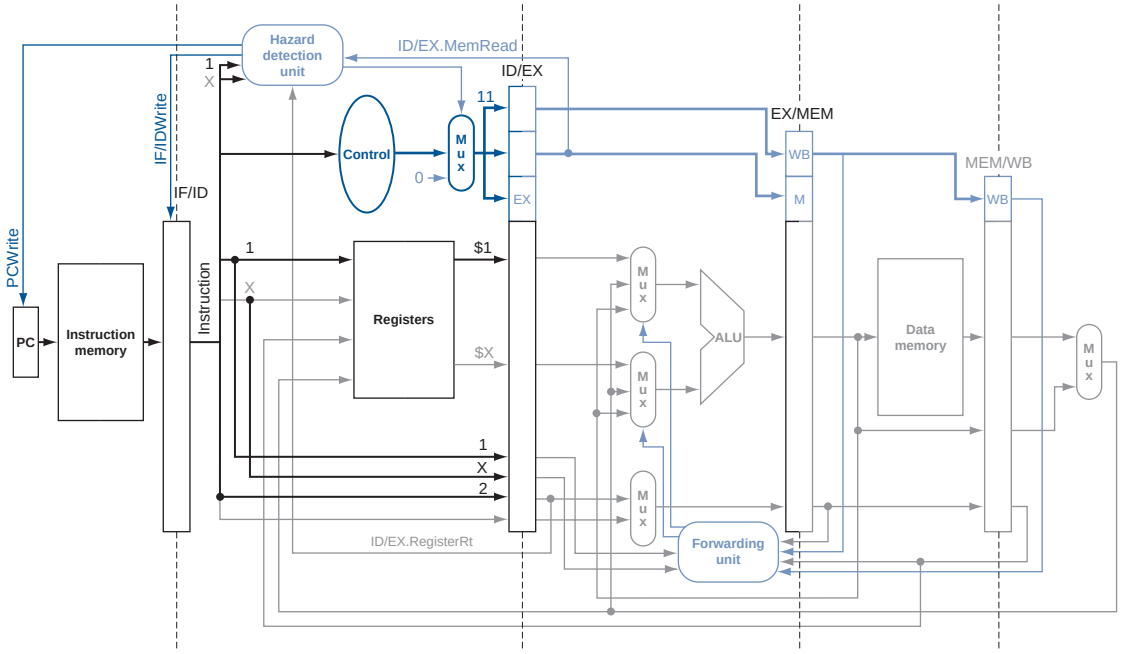

FIGURE 6.14.17 Blank pipeline diagram with stall hardware.

Miscellaneous Exercises

6.5 [5] <§6.1> How could we modify the following code to make use of a delayed branch slot?

```
Loop:       lw   $2, 100($3)
            addi $3, $3, 4
            beq  $3, $4, Loop
```

6.10 [5] <§6.2> For each pipeline register in Figure 6.22 on page 400, label each portion of the pipeline register with the name of the value that is loaded into the register. Determine the length of each field in bits. For example, the IF/ID pipeline register contains two fields, one of which is an instruction field that is 32 bits wide.

6.11 [15] <§§3.6, 6.2> Using Figure 3.13 on page 187 as your foundation, figure out a reasonable pipelined datapath structure for floating-point addition, and integrate the floating-point registers and the data memory into your picture to produce a figure similar to Figure 6.11 on page 388, except that your diagram should also contain an alternative pipeline for floating-point add instructions. Don't worry too much about the delays associated with each of the functions in Figure 3.13. You should simply assume that the sum of the delays is fairly large, and thus re-

quires you to use 2 or 3 cycles to perform floating-point addition. The addition should be pipelined so that a new floating-point add instruction can be started every cycle, assuming that there are no dependencies (you can ignore dependencies for this problem).

6.20 [20] <§§6.4, 6.5> Consider an instruction sequence used for a memory-to-memory copy:

```
lw    $2, 100($5)
sw    $2, 200($6)
```

The elaboration starting on page 412 of the text discusses this situation and states that additional forwarding hardware can improve its performance. Show the necessary additions to the datapath of Figure 6.33 to allow code like this to run without stalling. Include forwarding equations (such as the ones appearing on pages 407–411) for all of the control signals for any new or modified multiplexors in your datapath. Finally, rewrite the stall formula on page 413 so that this code sequence won't stall.

6.25 [20] <§6.5> The forwarding unit could be moved to the ID stage and forwarding decisions could be made earlier. The results of these decisions would need to be passed along with the instruction and used in the EX stage when actual forwarding would take place. This modification would speed up the EX stage and might allow for possible cycle-time improvement. Perform the modification. Provide a revised datapath and a description of the necessary changes. How has the ID/EX register changed? Provide new forwarding equations to replace those appearing on pages 407–411.

6.26 [15] <§§6.2–6.5> In the exercises for Chapter 5, we explored the implications of having load and store instructions with no offset. How would this modification to the instruction set architecture affect a pipelined implementation? Describe changes to the datapath and how performance would be impacted. Be sure to include a discussion of forwarding in your answer.

6.27 [10] <§§6.2–6.5> The 80x86 ISA has arithmetic instructions that can directly access memory. Write a paragraph or two explaining why it would be hard to add this instruction to the MIPS pipeline described in this chapter. (Hint: You may have to add one or more additional stages to the pipeline.)

6.28 [30] <§6.5, Appendix C> Using Appendix C and the answer to Exercise 6.23, design the hardware to implement the forwarding unit. (Hint: To decide if register numbers are equal, try using an exclusive OR gate. See the elaboration on page B-46 in Appendix B).

6.31 [10] <§6.6> One extension of the MIPS instruction set architecture has two new instructions called `movn` (move if not zero) and `movz` (move if zero). For example, the instruction

```
movn $8, $11, $4
```

copies the contents of register 11 into register 8, provided that the value in register 4 is nonzero (otherwise it does nothing). The `movz` instruction is similar but copying takes place only if the register's value is zero. Show how to use the new instructions to put whichever is larger, register 8's value or register 9's value, into register 10. If the values are equal, copy either into register 10. You may use register 1 as an extra register for temporary use. Do not use any conditional branches.

6.32 [10] <§6.6> {Ex. 6.31} The solution to Exercise 6.31 should involve the execution of fewer instructions than would be required using conditional branches. Sometimes, however, rewriting code to use `movn` and `movz` rather than conditional branches doesn't reduce the number of instructions executed. Nonetheless, even if the use of `movn` and `movz` doesn't reduce the number of instructions executed, it can still make the program faster if it is being executed on a pipelined datapath. Explain why.