7.5 PIPELINED PROCESSOR

Pipelining, introduced in Section 3.6, is a powerful way to improve the throughput of a digital system. We design a pipelined processor by subdividing the single-cycle processor into five pipeline stages. Thus, five instructions can execute simultaneously, one in each stage. Because each stage has only one-fifth of the entire logic, the clock frequency is approximately five times faster. So, ideally, the latency of each instruction is unchanged, but the throughput is five times better. Microprocessors execute millions or billions of instructions per second, so throughput is more important than latency. Pipelining introduces some overhead, so the throughput will not be as high as we might ideally desire, but pipelining nevertheless gives such great advantage for so little cost that all modern high-performance microprocessors are pipelined.

Recall that *throughput* is the number of tasks (in this case, instructions) that complete per second. *Latency* is the time it takes for a given instruction to complete, from start to finish. (See Section 3.6)

Reading and writing the memory and register file and using the ALU typically constitute the biggest delays in the processor. We choose five pipeline stages so that each stage involves exactly one of these slow steps. Specifically, we call the five stages *Fetch*, *Decode*, *Execute*, *Memory*, and *Writeback*. They are similar to the five steps that the multicycle processor used to perform `lw`. In the *Fetch* stage, the processor reads the instruction from instruction memory. In the *Decode* stage, the processor reads the source operands from the register file and decodes the instruction to produce the control signals. In the *Execute* stage, the processor performs a computation with the ALU. In the *Memory* stage, the processor reads or writes data memory, if applicable. Finally, in the *Writeback* stage, the processor writes the result to the register file, if applicable.

Figure 7.47 shows a timing diagram comparing the single-cycle and pipelined processors. Time is on the horizontal axis and instructions are on the vertical axis. The diagram assumes component delays from Table 7.7 (see page 415) but ignores multiplexers and registers for simplicity. In the single-cycle processor in Figure 7.47(a), the first instruction is read from memory at time 0. Next, the operands are read from the register file. Then, the ALU executes the necessary computation. Finally, the data memory may be accessed, and the result is written back to the register file at 680 ps. The second instruction begins when the first completes. Hence, in this diagram, the single-cycle processor has an instruction latency of 200 + 100 + 120 + 200 + 60 = 680 ps (see Table 7.7 on page 415) and a throughput of 1 instruction per 680 ps (1.47 billion instructions per second).

Remember that for this abstract comparison of single-cycle and pipelined processor performance, we are ignoring the overhead of decoder, multiplexer, and register delays.

In the pipelined processor in Figure 7.47(b), the length of a pipeline stage is set at 200 ps by the slowest stage, the memory access in the Fetch or Memory stage. Each pipeline stage is indicated by solid or dashed vertical blue lines. At time 0, the first instruction is fetched from memory. At 200 ps, the first instruction enters the Decode stage, and a second instruction is fetched. At 400 ps, the first instruction executes, the second instruction enters the Decode stage, and a third instruction is fetched.

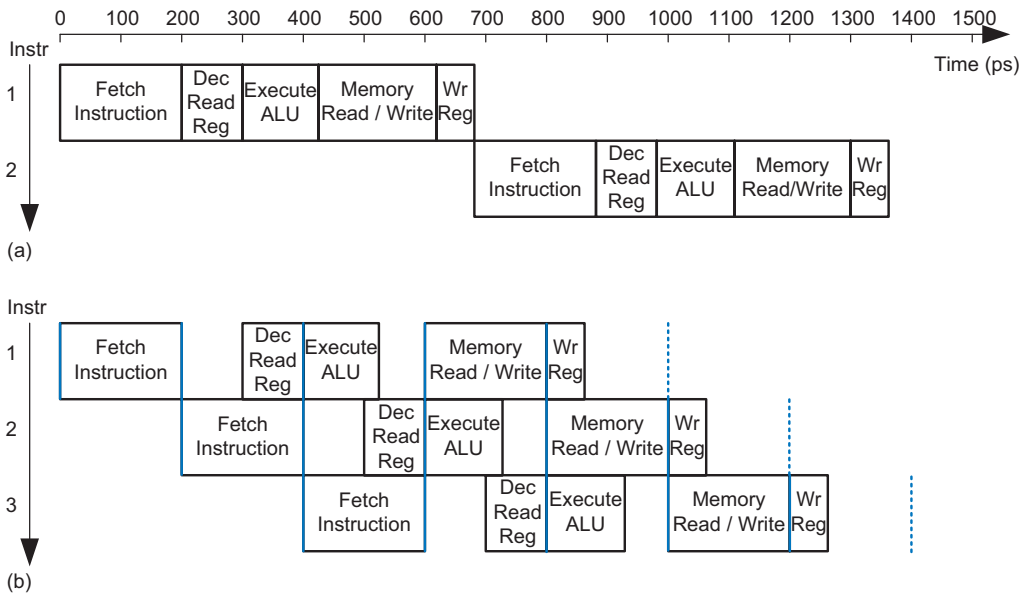

Figure 7.47 **Timing diagrams: (a) single-cycle processor and (b) pipelined processor**

And so forth, until all the instructions complete. The instruction latency is $5 \times 200 = 1000$ ps. Because the stages are not perfectly balanced with equal amounts of logic, the latency is longer for the pipelined processor than for the single-cycle processor. The throughput is 1 instruction per 200 ps (5 billion instructions per second)—that is, one instruction completes every clock cycle. This throughput is 3.4 times as much as the single-cycle processor—not quite 5 times but, nonetheless, a substantial speedup.

Figure 7.48 shows an abstracted view of the pipeline in operation in which each stage is represented pictorially. Each pipeline stage is represented with its major component—instruction memory (IM), register file (RF) read, ALU execution, data memory (DM), and register file writeback—to illustrate the flow of instructions through the pipeline. Reading across a row shows the clock cycle in which a particular instruction is in each stage. For example, the `sub` instruction is fetched in cycle 3 and executed in cycle 5. Reading down a column shows what the various pipeline stages are doing on a particular cycle. For example, in cycle 6, the register file is writing a sum to `s3`, the data memory is idle, the ALU is computing (`s11 & t0`), `t4` is being read from the register file, and the `or` instruction is being fetched from instruction memory. Stages are shaded to indicate when they are used. For example, the data memory is used by `lw` in cycle 4 and by `sw` in cycle 8. The instruction memory and ALU are used in every cycle. The register file is written by every instruction except `sw`. In the pipelined processor, the register file is

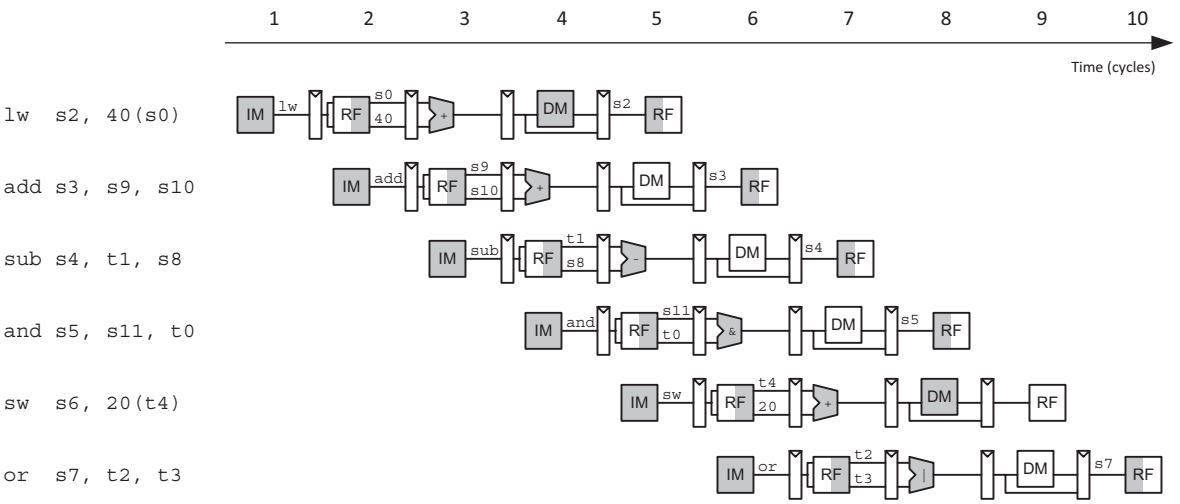

Figure 7.48 Abstract view of pipeline in operation

used twice in every cycle: it is written in the first part of a cycle and read in the second part, as suggested by the shading. This way, data can be written by one instruction and read by another within a single cycle.

A central challenge in pipelined systems is handling hazards that occur when one instruction's result is needed by a subsequent instruction before the former instruction has completed. For example, if the `add` in Figure 7.48 used `s2` as a source instead of `s10`, a hazard would occur because the `s2` register has not yet been written by the `lw` instruction when it is read by `add` in cycle 3. After designing the pipelined datapath and control, this section explores *forwarding*, *stalls*, and *flushes* as methods to resolve hazards. Finally, this section revisits performance analysis considering sequencing overhead and the impact of hazards.

7.5.1 Pipelined Datapath

The pipelined datapath is formed by chopping the single-cycle datapath into five stages separated by pipeline registers. Figure 7.49(a) shows the single-cycle datapath stretched out to leave room for the pipeline registers. Figure 7.49(b) shows the pipelined datapath formed by inserting four pipeline registers to separate the datapath into five stages. The stages and their boundaries are indicated in blue. Signals are given a suffix (F, D, E, M, or W) to indicate the stage in which they reside.

The register file is peculiar because it is read in the Decode stage and written in the Writeback stage. So, although the register file is drawn in the Decode stage, its write address and write data come from the Writeback stage. This feedback will lead to pipeline hazards, which are discussed in

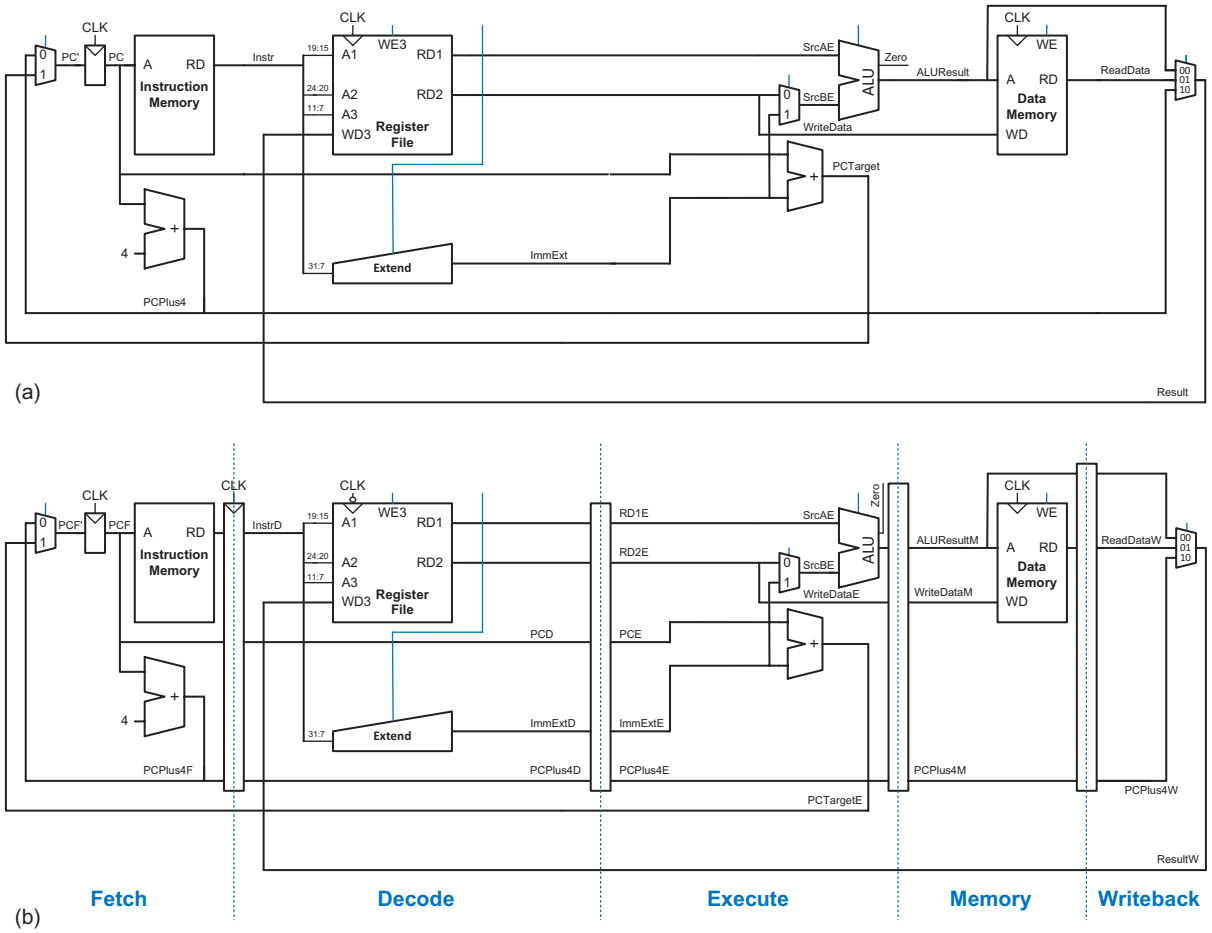

Figure 7.49 Datapaths: (a) single-cycle and (b) pipelined

Section 7.5.3. The register file in the pipelined processor writes on the falling edge of *CLK* so that it can write a result in the first half of a cycle and read that result in the second half of the cycle for use in a subsequent instruction.

One of the subtle but critical issues in pipelining is that all signals associated with a particular instruction must advance through the pipeline in unison. Figure 7.49(b) has an error related to this issue. Can you find it?

The error is in the register file write logic, which should operate in the Writeback stage. The data value comes from *ResultW*, a Writeback stage signal. But the destination register comes from *RdD* ($InstrD_{11:7}$), which is a Decode stage signal. In the pipeline diagram of Figure 7.48, during cycle 5, the result of the `lw` instruction would be incorrectly written to `s5` rather than `s2`.

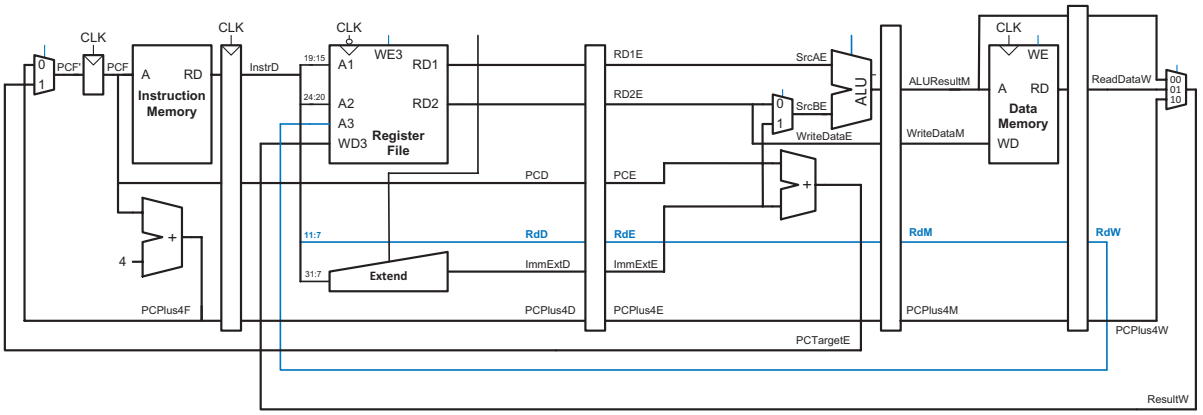

Figure 7.50 Corrected pipelined datapath

Figure 7.50 shows a corrected datapath, with the modification in blue. The *Rd* signal is now pipelined along through the Execution, Memory, and Writeback stages, so it remains in sync with the rest of the instruction. *RdW* and *ResultW* are fed back together to the register file in the Writeback stage.

The astute reader may note that the logic to produce *PCF'* (the next PC) is also problematic because it could be updated with either a Fetch or an Execute stage signal (*PCPlus4F* or *PCTargetE*). This control hazard will be fixed in Section 7.5.3.

7.5.2 Pipelined Control

The pipelined processor uses the same control signals as the single-cycle processor and, therefore, has the same control unit. The control unit examines the **op**, **funct3**, and **funct7**$_5$ fields of the instruction in the Decode stage to produce the control signals, as was described in Section 7.3.3 for the single-cycle processor. These control signals must be pipelined along with the data so that they remain synchronized with the instruction.

The entire pipelined processor with control is shown in Figure 7.51. *RegWrite* must be pipelined into the Writeback stage before it feeds back to the register file, just as *Rd* was pipelined in Figure 7.50. In addition to R-type ALU instructions, `lw`, `sw`, and `beq`, this pipelined processor also supports `jal` and I-type ALU instructions.

7.5.3 Hazards

In a pipelined system, multiple instructions are handled concurrently. When one instruction is *dependent* on the results of another that has

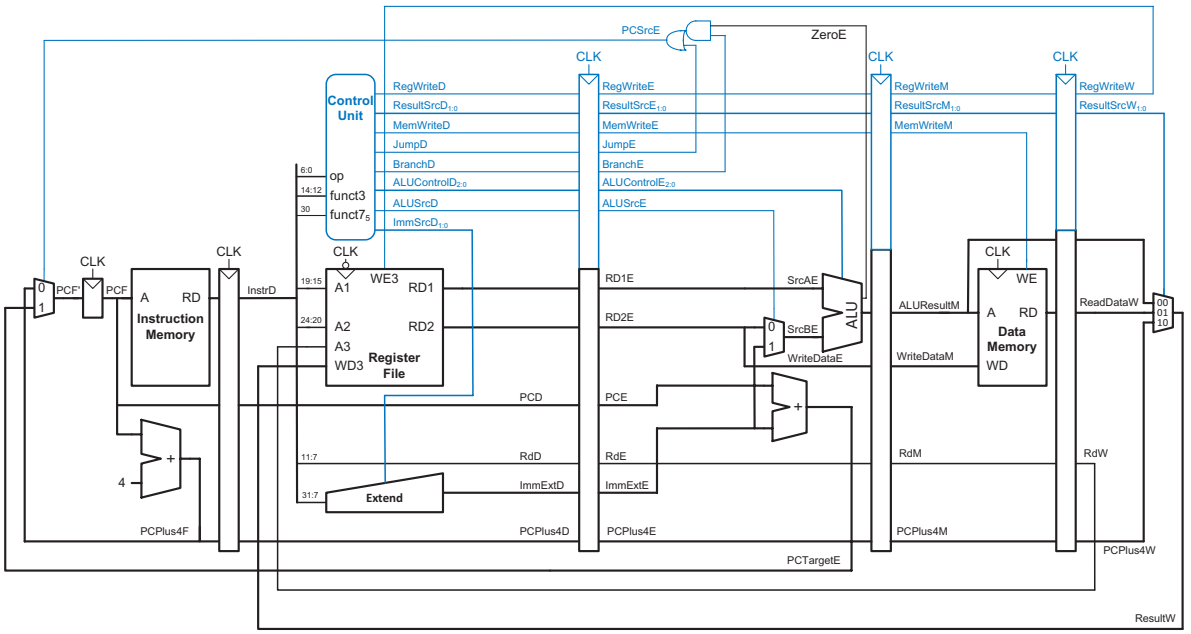

Figure 7.51 **Pipelined processor with control**

not yet completed, a *hazard* occurs. The register file is written during the first half of the cycle and read during the second half of the cycle, so a register can be written and read back in the same cycle without introducing a hazard.

Figure 7.52 illustrates hazards that occur when one instruction writes a register (`s8`) and subsequent instructions read this register. The blue arrows highlight when `s8` is written to the register file (in cycle 5) as compared to when it is needed by subsequent instructions. This is called a *read after write* (*RAW*) *hazard*. The `add` instruction writes a result into `s8` in the first half of cycle 5. However, the `sub` instruction reads `s8` on cycle 3, obtaining the wrong value. The `or` instruction reads `s8` on cycle 4, again obtaining the wrong value. The `and` instruction reads `s8` in the second half of cycle 5, obtaining the correct value, which was written in the first half of cycle 5. Subsequent instructions also read the correct value of `s8`. The diagram shows that hazards may occur in this pipeline when an instruction writes a register and either of the two subsequent instructions reads that register. Without special treatment, the pipeline will compute the wrong result.

A software solution would be to require the programmer or compiler to insert nop instructions between the `add` and `sub` instructions so that the dependent instruction does not read the result (`s8`) until it is available

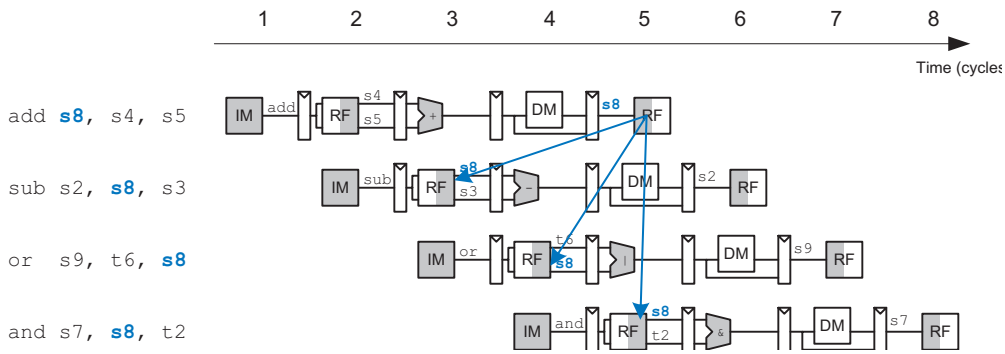

Figure 7.52 Abstract pipeline diagram illustrating hazards

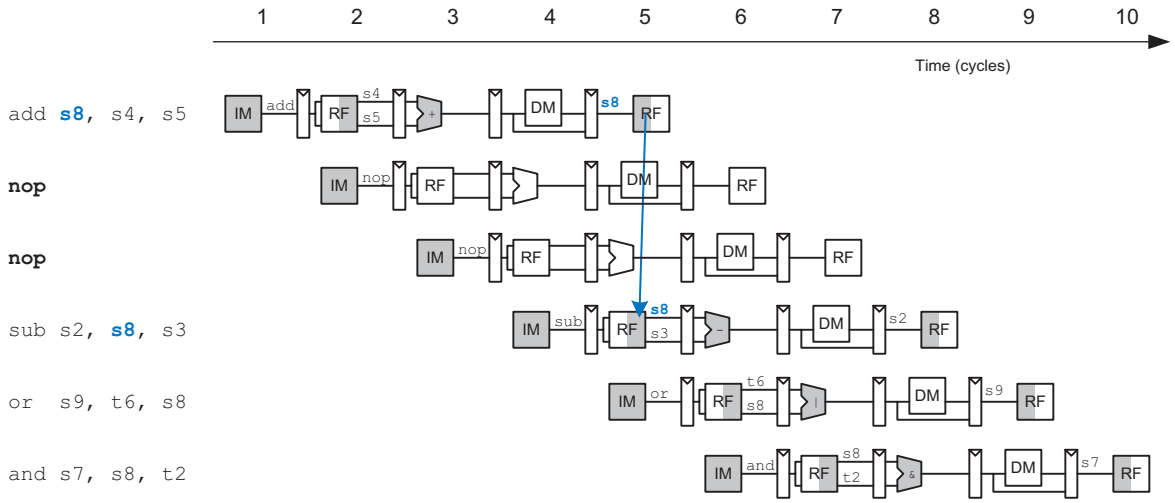

Figure 7.53 Solving data hazard with nops

in the register file, as shown in Figure 7.53. Such a *software interlock* complicates programming and degrades performance, so it is not ideal.

On closer inspection, observe from Figure 7.52 that the sum from the `add` instruction is computed by the ALU in cycle 3 and is not strictly needed by the `and` instruction until the ALU uses it in cycle 4. In principle, we should be able to forward the result from one instruction to the next to resolve the RAW hazard without waiting for the result to appear in the register file and without slowing down the pipeline. In other situations explored later in this section, we may have to stall the pipeline to give time for a result to be produced before the subsequent instruction uses the result. In any event, something must be done to solve hazards so that the program executes correctly despite the pipelining.

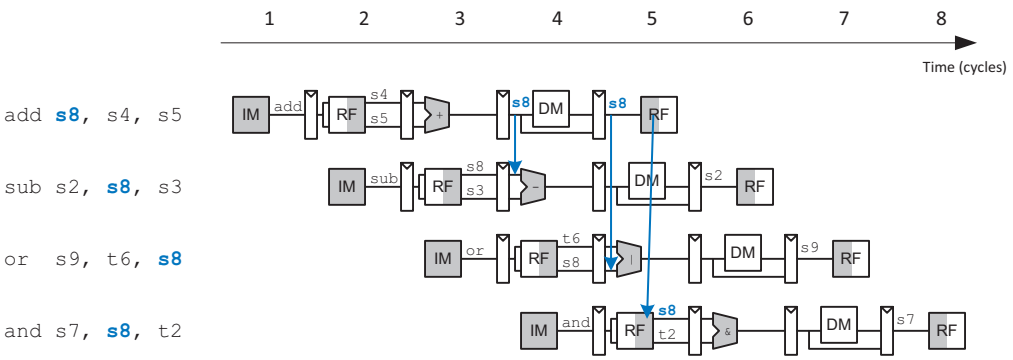

Figure 7.54 Abstract pipeline diagram illustrating forwarding

Hazards are classified as data hazards or control hazards. A *data hazard* occurs when an instruction tries to read a register that has not yet been written back by a previous instruction. A *control hazard* occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place. In the remainder of this section, we enhance the pipelined processor with a Hazard Unit that detects hazards and handles them appropriately so that the processor executes the program correctly.

Solving Data Hazards with Forwarding

Some data hazards can be solved by *forwarding* (also called *bypassing*) a result from the Memory or Writeback stage to a dependent instruction in the Execute stage. This requires adding multiplexers in front of the ALU to select its operands from the register file or the Memory or Writeback stage. Figure 7.54 illustrates this principle. This program computes `s8` with the `add` instruction and then uses `s8` in the three subsequent instructions. In cycle 4, `s8` is forwarded from the Memory stage of the `add` instruction to the Execute stage of the dependent `sub` instruction. In cycle 5, `s8` is forwarded from the Writeback stage of the `add` instruction to the Execute stage of the dependent `or` instruction. Again, no forwarding is needed for the `and` instruction because `s8` is written to the register file in the first half of cycle 5 and read in the second half.

Forwarding is necessary when an instruction in the Execute stage has a source register matching the destination register of an instruction in the Memory or Writeback stage. Figure 7.55 modifies the pipelined processor to support forwarding. It adds a *Hazard Unit* and two *forwarding multiplexers*. The hazard detection unit receives the two source registers from the instruction in the Execute stage, *Rs1E* and *Rs2E*, and the destination registers from the instructions in the Memory and Writeback

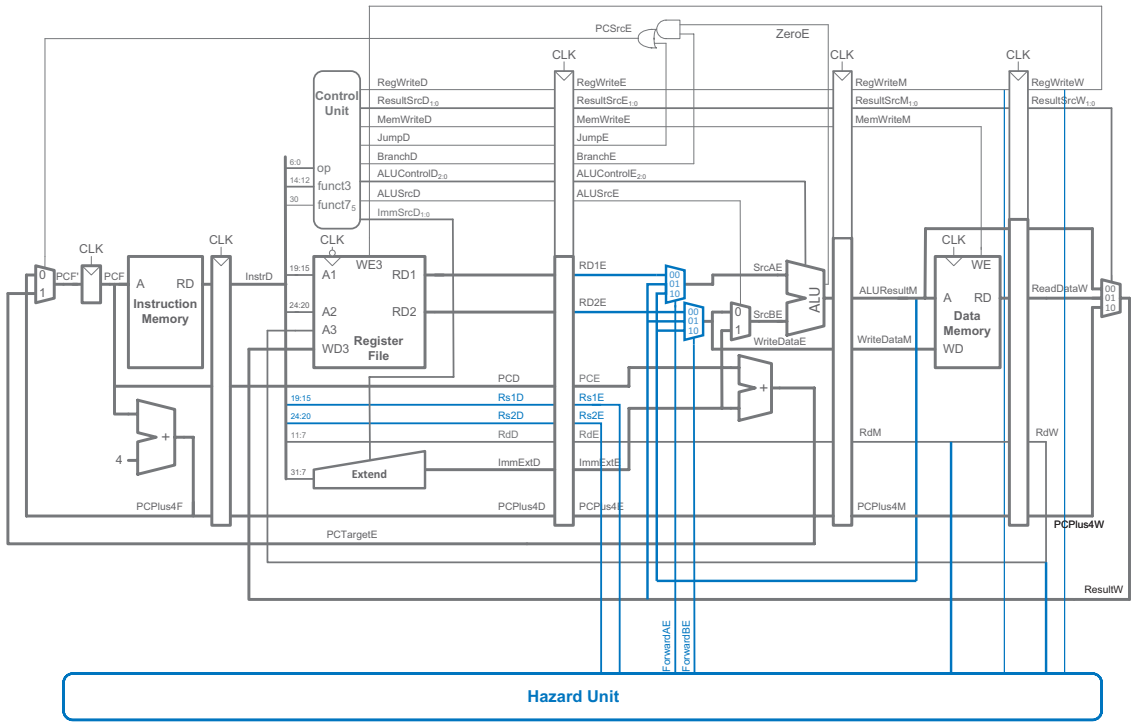

Figure 7.55 Pipelined processor with forwarding to solve some data hazards

stages, *RdM* and *RdW*. It also receives the *RegWrite* signals from the Memory and Writeback stages (*RegWriteM* and *RegWriteW*) to know whether the destination register will actually be written (e.g., the `sw` and `beq` instructions do not write results to the register file and, hence, do not have their results forwarded).

The Hazard Unit computes control signals for the forwarding multiplexers to choose operands from the register file or from the results in the Memory or Writeback stage (*ALUResultM* or *ResultW*). The Hazard Unit should forward from a stage if that stage will write a destination register *and* the destination register matches the source register. However, `x0` is hardwired to 0 and should never be forwarded. If both the Memory and Writeback stages contain matching destination registers, then the Memory stage should have priority because it contains the more recently executed instruction. In summary, the function of the forwarding logic for *SrcAE* (*ForwardAE*) is given on the next page. The forwarding logic for *SrcBE* (*ForwardBE*) is identical except that it checks *Rs2E* instead of *Rs1E*.

```
if      ((Rs1E == RdM) & RegWriteM) & (Rs1E != 0) then // Forward from Memory stage
        ForwardAE = 10
else if ((Rs1E == RdW) & RegWriteW) & (Rs1E != 0) then // Forward from Writeback stage
        ForwardAE = 01
else    ForwardAE = 00                                 // No forwarding (use RF output)
```

Solving Data Hazards with Stalls

Forwarding is sufficient to solve RAW data hazards when the result is computed in the Execute stage of an instruction because its result can then be forwarded to the Execute stage of the next instruction. Unfortunately, the `lw` instruction does not finish reading data until the end of the Memory stage, so its result cannot be forwarded to the Execute stage of the next instruction. We say that the `lw` instruction has a *two-cycle latency* because a dependent instruction cannot use its result until two cycles later. Figure 7.56 shows this problem. The `lw` instruction receives data from memory at the end of cycle 4, but the `and` instruction needs that data (the value in `s7`) as a source operand at the beginning of cycle 4. There is no way to solve this hazard with forwarding.

A solution is to *stall* the pipeline, holding up operation until the data is available. Figure 7.57 shows stalling the dependent instruction (`and`) in the Decode stage. `and` enters the Decode stage in cycle 3 and stalls there through cycle 4. The subsequent instruction (`or`) must remain in the Fetch stage during both cycles as well because the Decode stage is full.

In cycle 5, the result can be forwarded from the Writeback stage of `lw` to the Execute stage of `and`. Also, in cycle 5, source `s7` of the `or` instruction is read directly from the register file, with no need for forwarding.

Note that the Execute stage is unused in cycle 4. Likewise, Memory is unused in cycle 5 and Writeback is unused in cycle 6. This unused stage propagating through the pipeline is called a *bubble*, which behaves like a nop instruction. The bubble is introduced by zeroing out the Execute stage control signals during a Decode stage stall so that the bubble performs no action and changes no architectural state.

In summary, stalling a stage is performed by disabling its pipeline register (i.e., the register to the left of a stage) so that the stage's inputs do not change. When a stage is stalled, all previous stages must also be stalled so that no subsequent instructions are lost. The pipeline register directly after the stalled stage must be cleared (flushed) to prevent bogus information from propagating forward. Stalls degrade performance, so they should be used only when necessary.

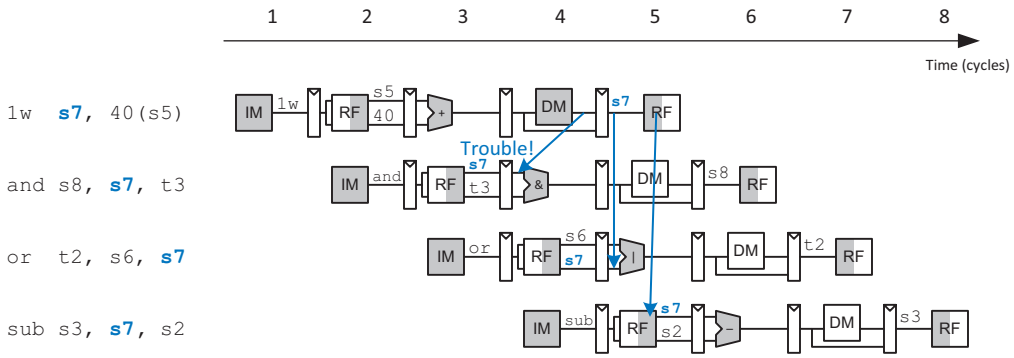

Figure 7.56 Abstract pipeline diagram illustrating trouble forwarding from `lw`

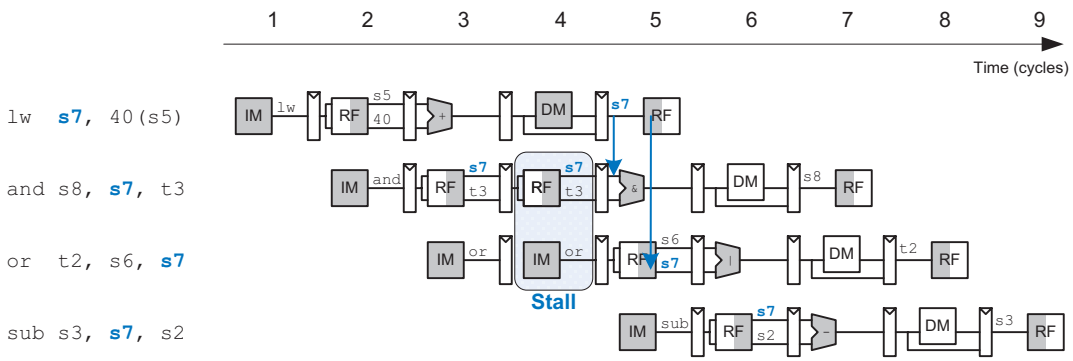

Figure 7.57 Abstract pipeline diagram illustrating stall to solve hazards

Figure 7.58 modifies the pipelined processor to add stalls for `lw` data dependencies. In order for the Hazard Unit to stall the pipeline, the following conditions must be met:

1. A load word is in the Execute stage (indicated by $ResultSrcE_0 = 1$) and
2. The load's destination register (*RdE*) matches *Rs1D* or *Rs2D*, the source operands of the instruction in the Decode stage

Stalls are supported by adding enable inputs (*EN*) to the Fetch and Decode pipeline registers and a synchronous reset/clear (*CLR*) input to the Execute pipeline register. When a load word (`lw`) stall occurs, *StallD* and *StallF* are asserted to force the Decode and Fetch stage pipeline registers to retain their existing values. *FlushE* is also asserted to clear the contents of the Execute stage pipeline register, introducing a bubble. The Hazard Unit *lwStall* (load word stall) signal indicates when the pipeline

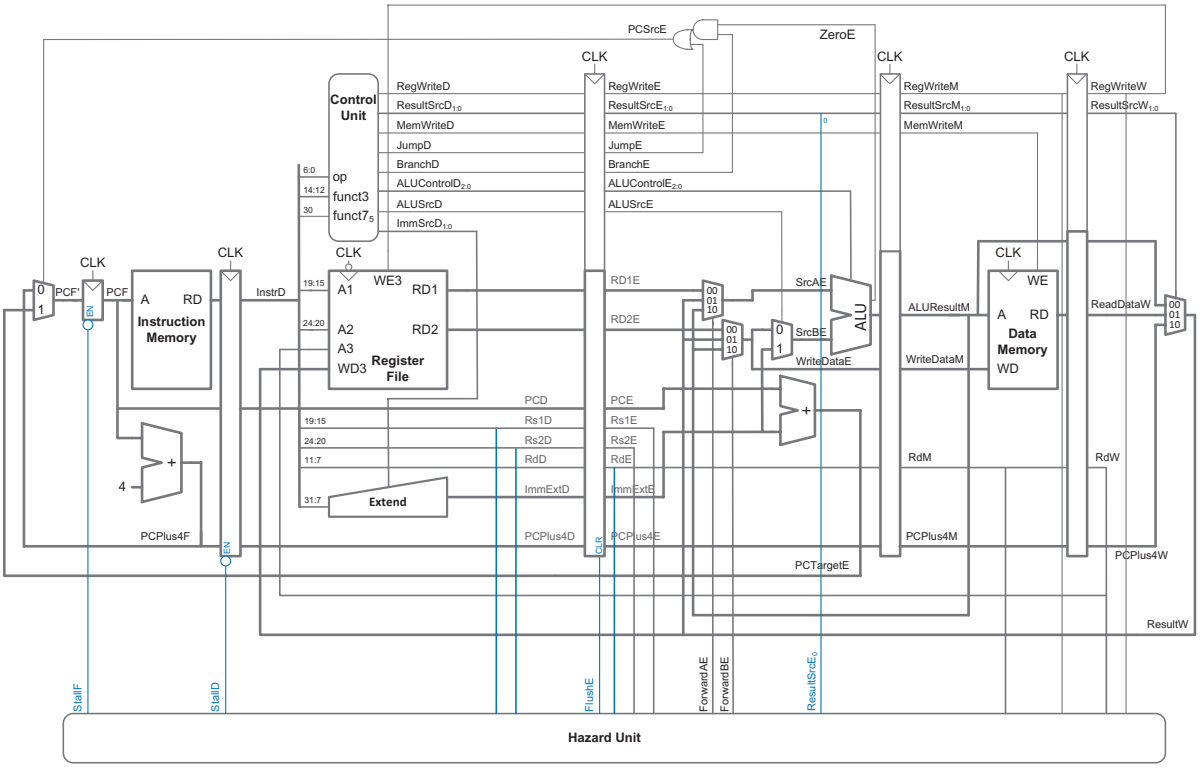

Figure 7.58 Pipelined processor with stalls to solve `lw` data hazard

The *lwStall* logic described here could cause the processor to stall unnecessarily when the destination of the load is `x0` or when a false dependency exists—that is, when the instruction in the Decode stage is a J- or I-type instruction that randomly causes a false match between bits in their immediate fields and *RdE*. However, these cases are rare (and poor coding practice, in the case of `x0` being the load destination) and they cause only a small performance loss.

should be stalled due to a load word dependency. Whenever *lwStall* is TRUE, all of the stall and flush signals are asserted. Hence, the logic to compute the stalls and flushes is:

$$lwStall = ResultSrcE_0 \;\&\; ((Rs1D == RdE) \mid (Rs2D == RdE))$$
$$StallF = StallD = FlushE = lwStall$$

Solving Control Hazards

The `beq` instruction presents a control hazard: the pipelined processor does not know what instruction to fetch next because the branch decision has not been made by the time the next instruction is fetched.

One mechanism for dealing with this control hazard is to stall the pipeline until the branch decision is made (i.e., *PCSrcE* is computed). Because the decision is made in the Execute stage, the pipeline would have to be stalled for two cycles at every branch. This would severely degrade the system performance if branches occur often, which is typically the case.

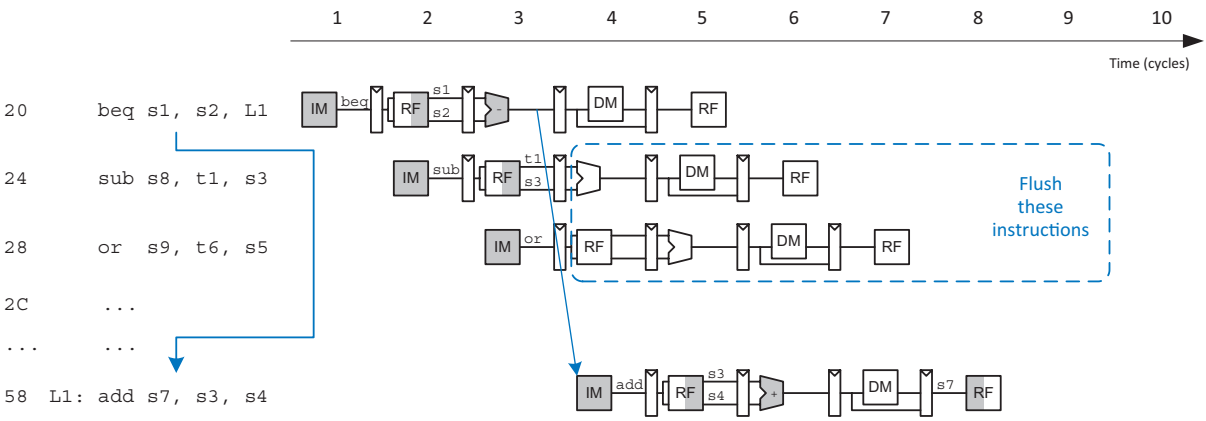

Figure 7.59 Abstract pipeline diagram illustrating flushing when a branch is taken

An alternative to stalling the pipeline is to predict whether the branch will be taken and begin executing instructions based on the prediction. Once the branch decision is available, the processor can throw out the instructions if the prediction was wrong. In the pipeline presented so far (Figure 7.58), the processor predicts that branches are not taken and simply continues executing the program in order until *PCSrcE* is asserted to select the next PC from *PCTargetE* instead. If the branch should have been taken, then the two instructions following the branch must be *flushed* (discarded) by clearing the pipeline registers for those instructions. These wasted instruction cycles are called the *branch misprediction penalty*.

Figure 7.59 shows such a scheme in which a branch from address 0x20 to address 0x58 is taken. The PC is not written until cycle 3, by which point the `sub` and `or` instructions at addresses 0x24 and 0x28 have already been fetched. These instructions must be flushed, and the `add` instruction is fetched from address 0x58 in cycle 4.

Finally, we must work out the stall and flush signals to handle branches and PC writes. When a branch is taken, the subsequent two instructions must be flushed from the pipeline registers of the Decode and Execute stages. Thus, we add a synchronous clear input (CLR) to the Decode pipeline register and add the *FlushD* output to the Hazard Unit. (When CLR = 1, the register contents are cleared, that is, become 0.) When a branch is taken (indicated by *PCSrcE* being 1), *FlushD* and *FlushE* must be asserted to flush the Decode and Execute pipeline registers. Figure 7.60 shows the enhanced pipelined processor for handling control hazards. The flushes are now calculated as:

$$FlushD = PCSrcE$$
$$FlushE = lwStall \mid PCSrcE$$

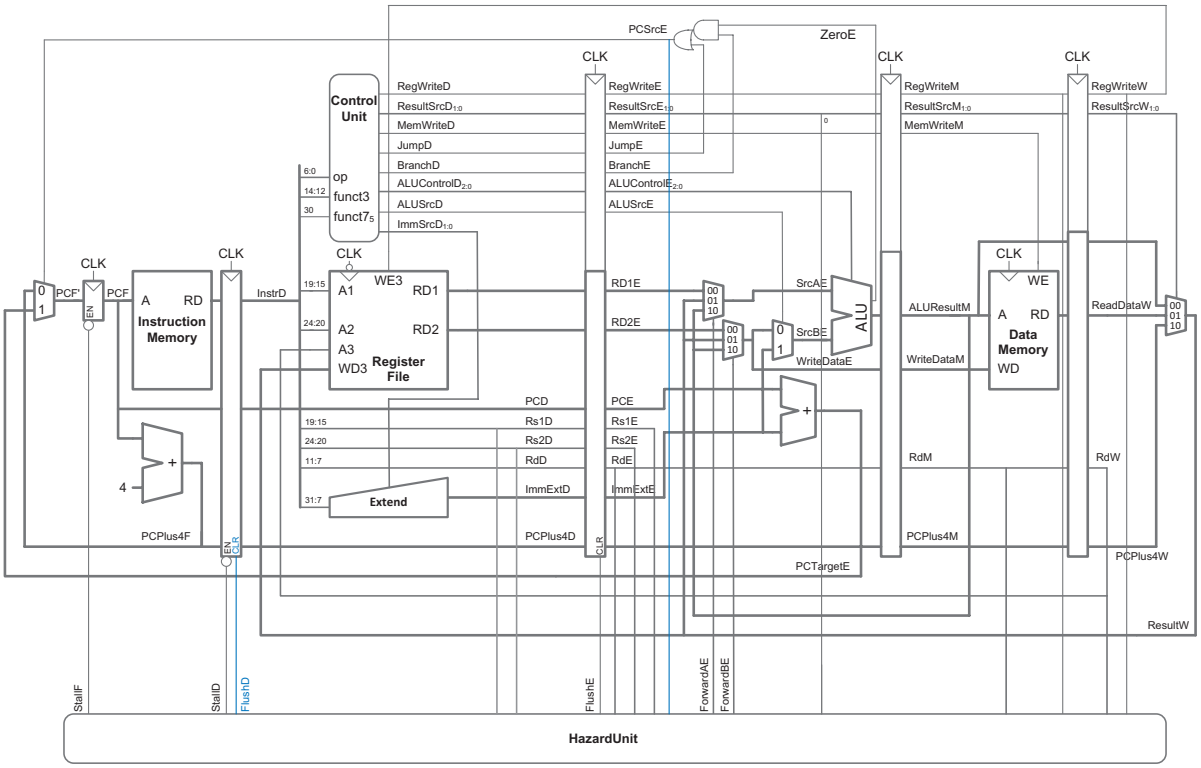

Figure 7.60 Expanded Hazard Unit for handling branch control hazard

Hazard Summary

In summary, RAW data hazards occur when an instruction depends on a result (from another instruction) that has not yet been written into the register file. Data hazards can be resolved by forwarding if the result is computed soon enough; otherwise, they require stalling the pipeline until the result is available. Control hazards occur when the decision of what instruction to fetch has not been made by the time the next instruction must be fetched. Control hazards are solved by stalling the pipeline until the decision is made or by predicting which instruction should be fetched and flushing the pipeline if the prediction is later determined to be wrong. Moving the decision as early as possible minimizes the number of instructions that are flushed on a misprediction. You may have observed by now that one of the challenges of designing a pipelined processor is to understand all possible interactions between instructions and to discover all of the hazards that may exist. Figure 7.61 shows the complete pipelined processor handling all of the hazards. The hazard logic is summarized on the next page.

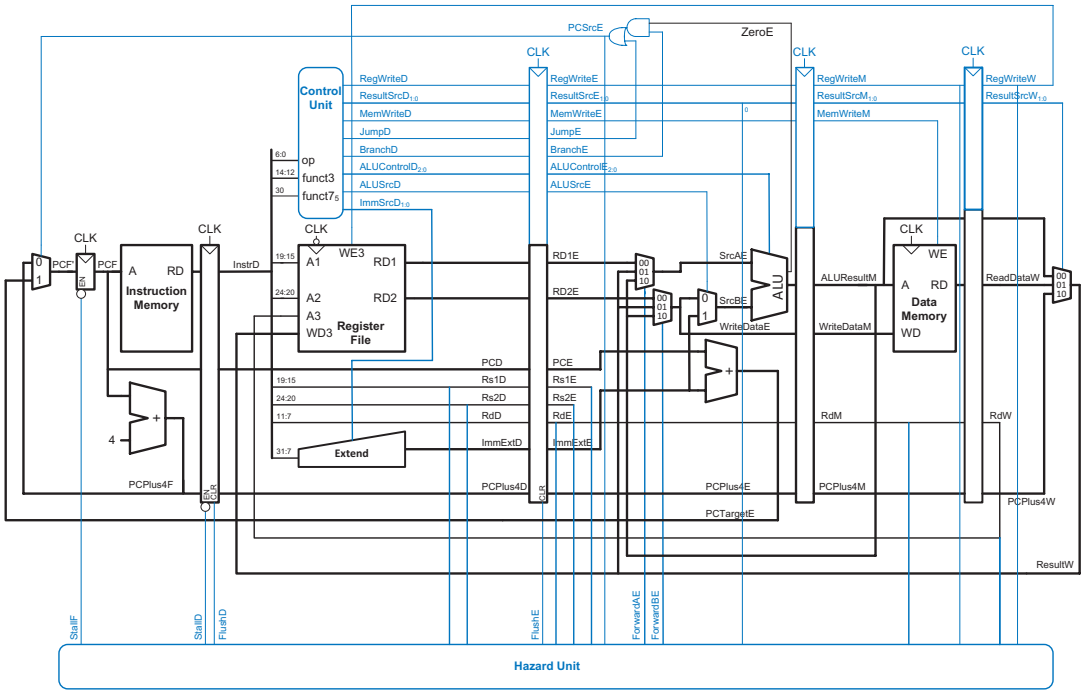

Figure 7.61 **Pipelined processor with full hazard handling**

Forward to solve data hazards when possible[3]**:**

if ((*Rs1E* == *RdM*) & *RegWriteM*) & (*Rs1E* != 0) then
ForwardAE = 10
else if ((*Rs1E* == *RdW*) & *RegWriteW*) & (*Rs1E* != 0) then
ForwardAE = 01
else *ForwardAE* = 00

Stall when a load hazard occurs:

$lwStall = ResultSrcE_0$ & ((*Rs1D* == *RdE*) | (*Rs2D* == *RdE*))
StallF = *lwStall*
StallD = *lwStall*

Flush when a branch is taken or a load introduces a bubble:

FlushD = *PCSrcE*
FlushE = *lwStall* | *PCSrcE*

[3] Recall that the forwarding logic for *SrcBE* (*ForwardBE*) is identical except that it checks *Rs2E* instead of *Rs1E*.

7.5.4 Performance Analysis

The pipelined processor ideally would have a CPI of 1 because a new instruction is *issued*—that is, fetched—every cycle. However, a stall or a flush wastes 1 to 2 cycles, so the CPI is slightly higher and depends on the specific program being executed.

Example 7.9 PIPELINED PROCESSOR CPI

The SPECINT2000 benchmark considered in Example 7.4 consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R- or I-type ALU instructions. Assume that 40% of the loads are immediately followed by an instruction that uses the result, requiring a stall, and that 50% of the branches are taken (mispredicted), requiring two instructions to be flushed. Ignore other hazards. Compute the average CPI of the pipelined processor.

Solution The average CPI is the weighted sum over each instruction of the CPI for that instruction multiplied by the fraction of time that instruction is used. Loads take one clock cycle when there is no dependency and two cycles when the processor must stall for a dependency, so they have a CPI of (0.6)(1) + (0.4)(2) = 1.4. Branches take one clock cycle when they are predicted properly and three when they are not, so they have a CPI of (0.5)(1) + (0.5)(3) = 2. Jumps take three clock cycles (CPI = 3). All other instructions have a CPI of 1. Hence, for this benchmark, the average CPI = (0.25)(1.4) + (0.1)(1) + (0.11)(2) + (0.02)(3) + (0.52)(1) = 1.25.

We can determine the cycle time by considering the critical path in each of the five pipeline stages shown in Figure 7.61. Recall that the register file is used twice in a single cycle: it is written in the first half of the Writeback cycle and read in the second half of the Decode cycle; so these stages can use only half of the cycle time for their critical path. Another way of saying it is this: twice the critical path for each of those stages must fit in a cycle. Figure 7.62 shows the critical path for the Execute stage. It occurs when a branch is in the Execute stage that requires forwarding from the Writeback stage: the path goes from the Writeback pipeline register, through the Result, ForwardBE, and SrcB multiplexers, through the ALU and AND-OR logic to the PC multiplexer and, finally, to the PC register.

The critical path analysis for the Execute stage assumes that the Hazard Unit delay for calculating *ForwardAE* and *ForwardBE* is less than or equal to the delay of the Result multiplexer. If the Hazard Unit delay is longer, it must be included in the critical path instead of the Result multiplexer delay.

$$T_{c\_pipelined} = max \begin{bmatrix} t_{pcq} + t_{mem} + t_{setup} & Fetch \\ 2(t_{RFread} + t_{setup}) & Decode \\ t_{pcq} + 4t_{mux} + t_{ALU} + t_{AND-OR} + t_{setup} & Execute \\ t_{pcq} + t_{mem} + t_{setup} & Memory \\ 2(t_{pcq} + t_{mux} + t_{RFsetup}) & Writeback \end{bmatrix} \quad (7.5)$$

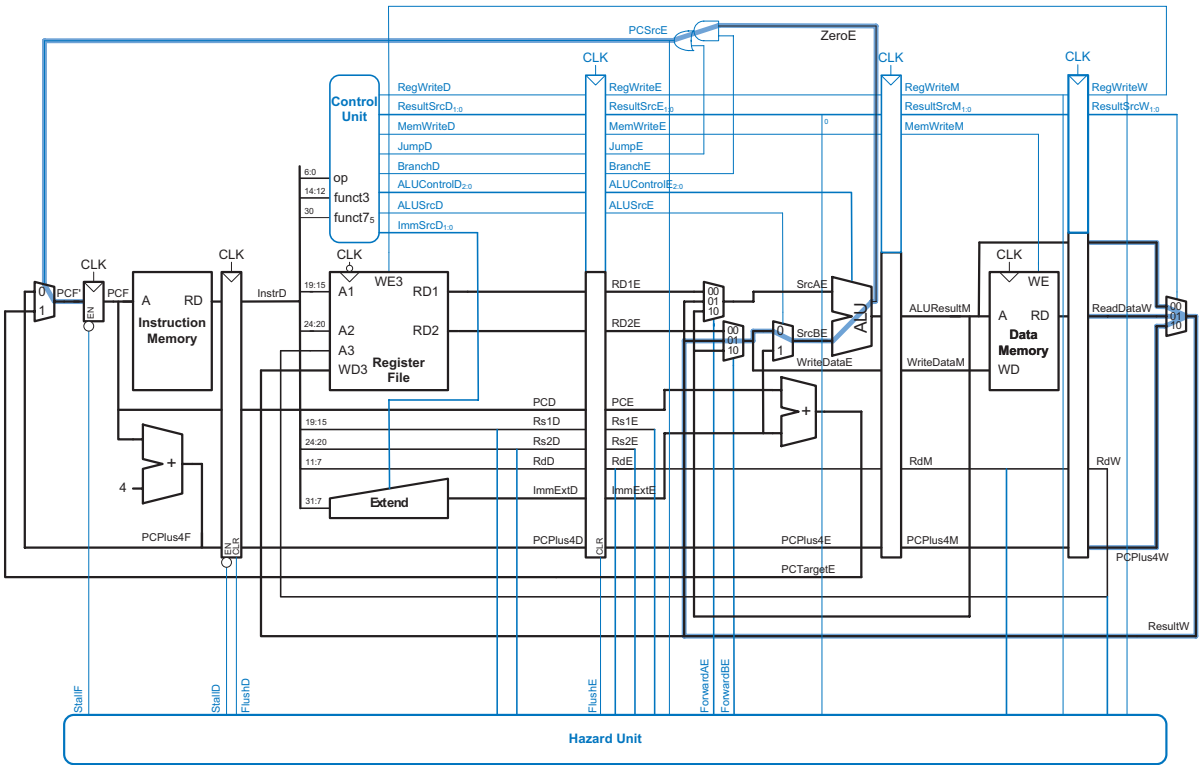

Figure 7.62 Pipelined processor critical path

Example 7.10 PIPELINED PROCESSOR PERFORMANCE COMPARISON

Ben Bitdiddle needs to compare the pipelined processor performance with that of the single-cycle and multicycle processors considered in Examples 7.4 and 7.8. The logic delays were given in Table 7.7 (on page 415). Help Ben compare the execution time of 100 billion instructions from the SPECINT2000 benchmark for each processor.

Solution According to Equation 7.5, the cycle time of the pipelined processor is $T_{c\_pipelined} = \max[40 + 200 + 50, 2(100 + 50), 40 + 4(30) + 120 + 20 + 50, 40 + 200 + 50, 2(40 + 30 + 60)] = 350$ ps. The Execute stage takes the longest. According to Equation 7.1, the total execution time is $T_{pipelined} = (100 \times 10^9$ instructions$)(1.25$ cycles/instruction$)(350 \times 10^{-12}$ s/cycle$) = 44$ seconds. This compares with 75 seconds for the single-cycle processor and 155 seconds for the multicycle processor.

Our pipelined processor is unbalanced, with branch resolution in the Execute stage taking much longer than any other stage. The pipeline could be balanced better by pushing the Result multiplexer back into the Memory stage, reducing the cycle time to 320 ps.

The pipelined processor is substantially faster than the others. However, its advantage over the single-cycle processor is nowhere near the fivefold speedup one might hope to get from a five-stage pipeline.