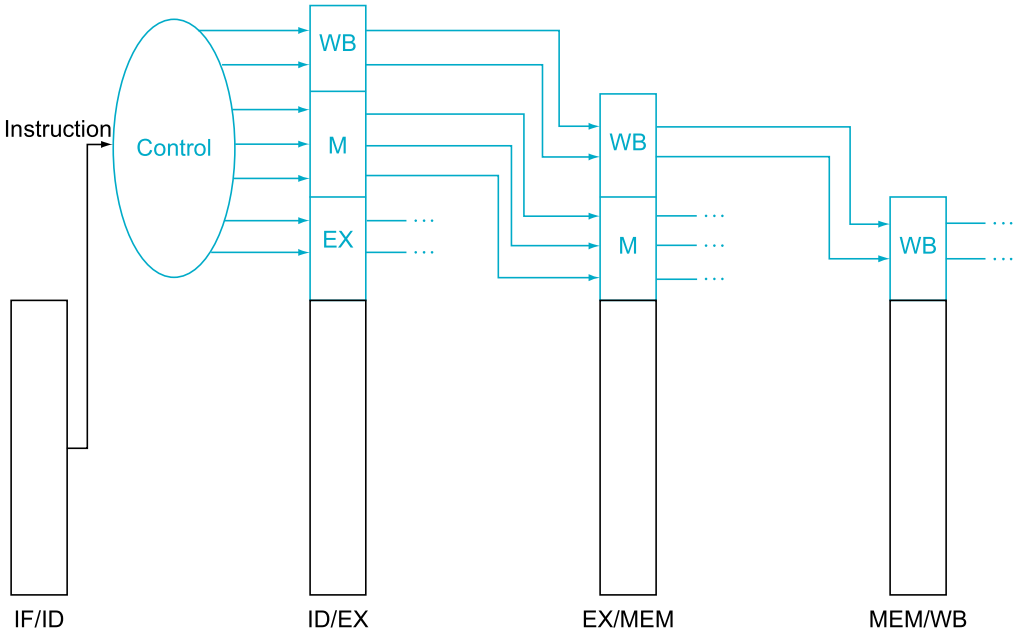



**FIGURE 4.52 The seven control lines for the final three stages.** Note that two of the seven control lines are used in the EX phase, with the remaining five control lines passed on to the EX/MEM pipeline register extended to hold the control lines; three are used during the MEM stage, and the last two are passed to MEM/WB for use in the WB stage.

## 4.8 Data Hazards: Forwarding versus Stalling

*What do you mean, why's it got to be built? It's a bypass. You've got to build bypasses.*

Douglas Adams, *The Hitchhiker's Guide to the Galaxy*, 1979

The examples in the previous section show the power of pipelined execution and how the hardware performs the task. It's now time to take off the rose-colored glasses and look at what happens with real programs. The RISC-V instructions in Figures 4.45 through 4.47 were independent; none of them used the results calculated by any of the others. Yet, in Section 4.6, we saw that data hazards are obstacles to pipelined execution.

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

```
sub   x2, x1, x3    // Register z2 written by sub
and   x12, x2, x5   // 1st operand(x2) depends on sub
or    x13, x6, x2   // 2nd operand(x2) depends on sub
add   x14, x2, x2   // 1st(x2) & 2nd(x2) depend on sub
sw    x15, 100(x2)  // Base (x2) depends on sub
```

The last four instructions are all dependent on the result in register `x2` of the first instruction. If register `x2` 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 `x2`.

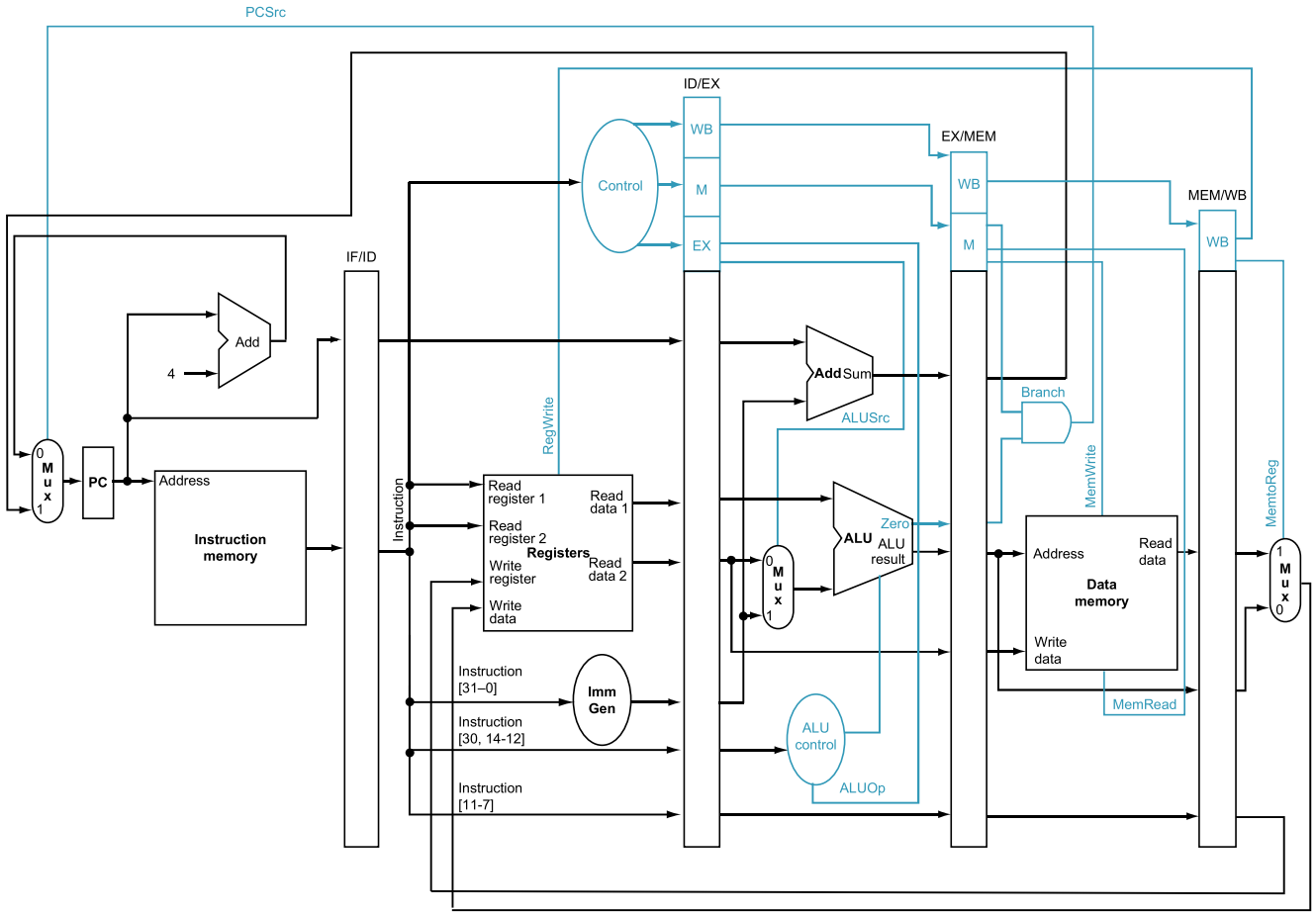



**FIGURE 4.53 The pipelined datapath of Figure 4.48, with the control signals connected to the control portions of the pipeline registers.** The control values for the last three stages are created during the instruction decode stage and then placed in the ID/EX pipeline register. The control lines for each pipe stage are used, and remaining control lines are then passed to the next pipeline stage.

How would this sequence perform with our pipeline? Figure 4.54 illustrates the execution of these instructions using a multiple-clock-cycle pipeline representation. To demonstrate the execution of this instruction sequence in our current pipeline, the top of Figure 4.54 shows the value of register `x2`, which changes during the middle of clock cycle 5, when the `sub` instruction writes its result.

The potential of `add` hazard can be resolved by the design of the register file hardware: What happens when a register is read and written in the same clock cycle? We assume that the write is in the first half of the clock cycle and the read is in the second half, so the read delivers what is written. As is the case for many implementations of register files, we have no data hazard in this case.

Figure 4.67 shows that the values read for register `x2` would *not* be the result of the `sub` instruction unless the read occurred during clock cycle 5 or later. Thus, the instructions that would get the correct value of −20 are `add` and `sw`; the `and` and `or` instructions would get the incorrect value 10! Using this style of drawing, such problems become apparent when a dependence line goes backward in time.

As mentioned in Section 4.6, the desired result is available at the end of the EX stage of the `sub` instruction or clock cycle 3. When are the data actually needed by the `and` and `or` instructions? The answer is at the beginning of the EX stage of the `and` and `or` instructions, or clock cycles 4 and 5, respectively. Thus, we can execute this segment without stalls if we simply *forward* the data as soon as it is available to any units that need it before it is ready to read from the register file.

How does forwarding work? For simplicity in the rest of this section, we consider only the challenge of forwarding to an operation in the EX stage, which may be either an ALU operation or an effective address calculation. This means that when an instruction tries to use a register in its EX stage that an earlier instruction intends to write in its WB stage, we actually need the values as inputs to the ALU.

A notation that names the fields of the pipeline registers allows for a more precise notation of dependences. For example, "ID/EX.RegisterRs1" refers to the number of one register whose value is found in the pipeline register ID/EX; that is, the one from the first read port of the register file. The first part of the name, to the left of the period, is the name of the pipeline register; the second part is the name of the field in that register. Using this notation, the two pairs of hazard conditions are

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1

1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1

2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

The first hazard in the sequence on page 313 is on register `x2`, between the result of `sub x2, x1, x3` and the first read operand of `and x12, x2, x5`. This hazard can be detected when the `and` instruction is in the EX stage and the prior instruction is in the MEM stage, so this is hazard 1a:

EX/MEM.RegisterRd = ID/EX.RegisterRs1 = `x2`

**EXAMPLE**

**Dependence Detection**

Classify the dependences in this sequence from page 313:

```
sub x2, x1, x3      // Register x2 set by sub
and x12, x2, x5     // 1st operand(z2) set by sub
or  x13, x6, x2     // 2nd operand(x2) set by sub
add x14, x2, x2     // 1st(x2) & 2nd(x2) set by sub
sw  x15, 100(x2)    // Index(x2) set by sub
```

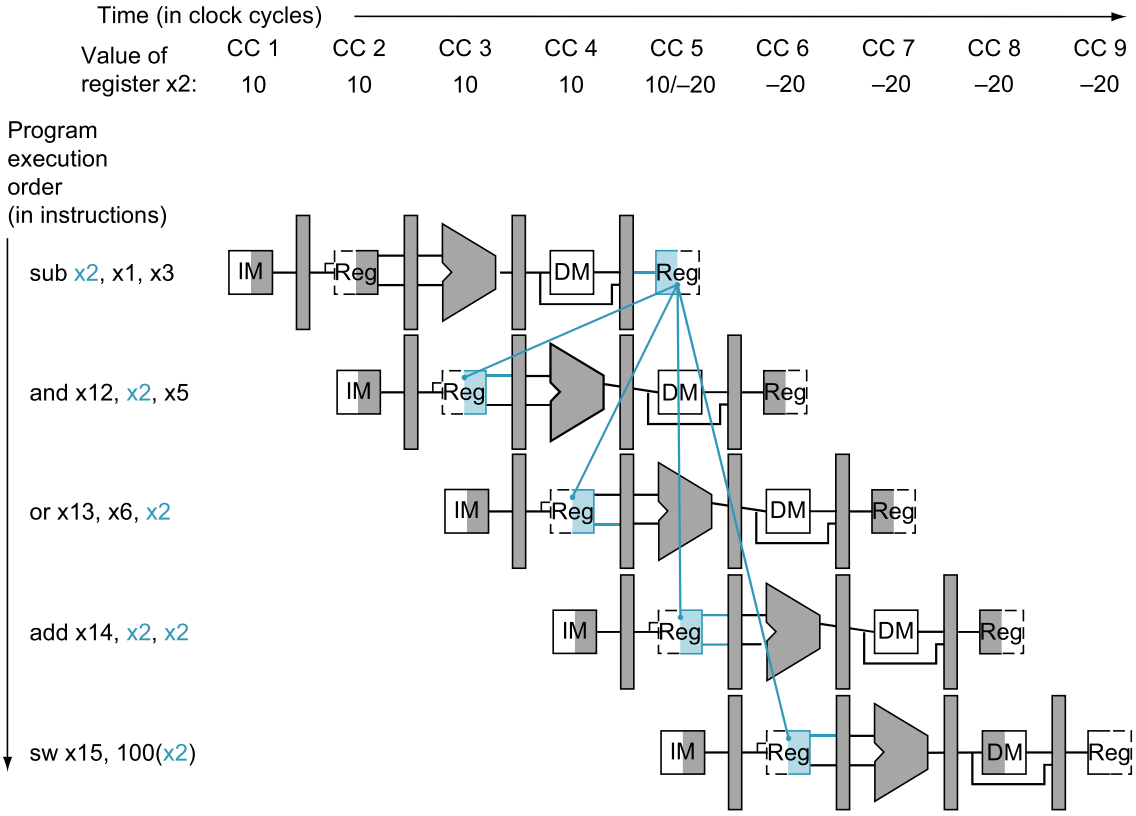



**FIGURE 4.54 Pipelined dependences in a five-instruction sequence using simplified datapaths to show the dependences.** All the dependent actions are shown in color, and "CC 1" at the top of the figure means clock cycle 1. The first instruction writes into `x2`, and all the following instructions read `x2`. 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*.

**ANSWER**

As mentioned above, the `sub-and` is a type 1a hazard. The remaining hazards are as follows:

- The `sub-or` is a type 2b hazard:

  MEM/WB.RegisterRd = ID/EX.RegisterRs2 = `x2`

- The two dependences on `sub-add` are not hazards because the register file supplies the proper data during the ID stage of `add`.
- There is no data hazard between `sub` and `sw` because `sw` reads `x2` the clock cycle *after* `sub` writes `x2`.

Because some instructions do not write registers, this policy is inaccurate; sometimes it would forward when it shouldn't. One solution is simply to check to see if the RegWrite signal will be active: examining the WB control field of the pipeline register during the EX and MEM stages determines whether RegWrite is asserted. Recall that RISC-V requires that every use of `x0` as an operand must yield an operand value of 0. If an instruction in the pipeline has `x0` as its destination (for

example, `addi x0, x1, 2`), we want to avoid forwarding its possibly nonzero result value. Not forwarding results destined for `x0` frees the assembly programmer and the compiler of any requirement to avoid using `x0` as a destination. The conditions above thus work properly as long as we add EX/MEM.RegisterRd ≠ 0 to the first hazard condition and MEM/WB.RegisterRd ≠ 0 to the second.

Now that we can detect hazards, half of the problem is resolved—but we must still forward the proper data.

Figure 4.55 shows the dependences between the pipeline registers and the inputs to the ALU for the same code sequence as in Figure 4.54. The change is that the dependence begins from a *pipeline* register, rather than waiting for the WB stage to write the register file. Thus, the required data exist in time for later instructions, with the pipeline registers holding the data to be forwarded.

If we can take the inputs to the ALU from *any* pipeline register rather than just ID/EX, then we can forward the correct data. By adding multiplexors to the input

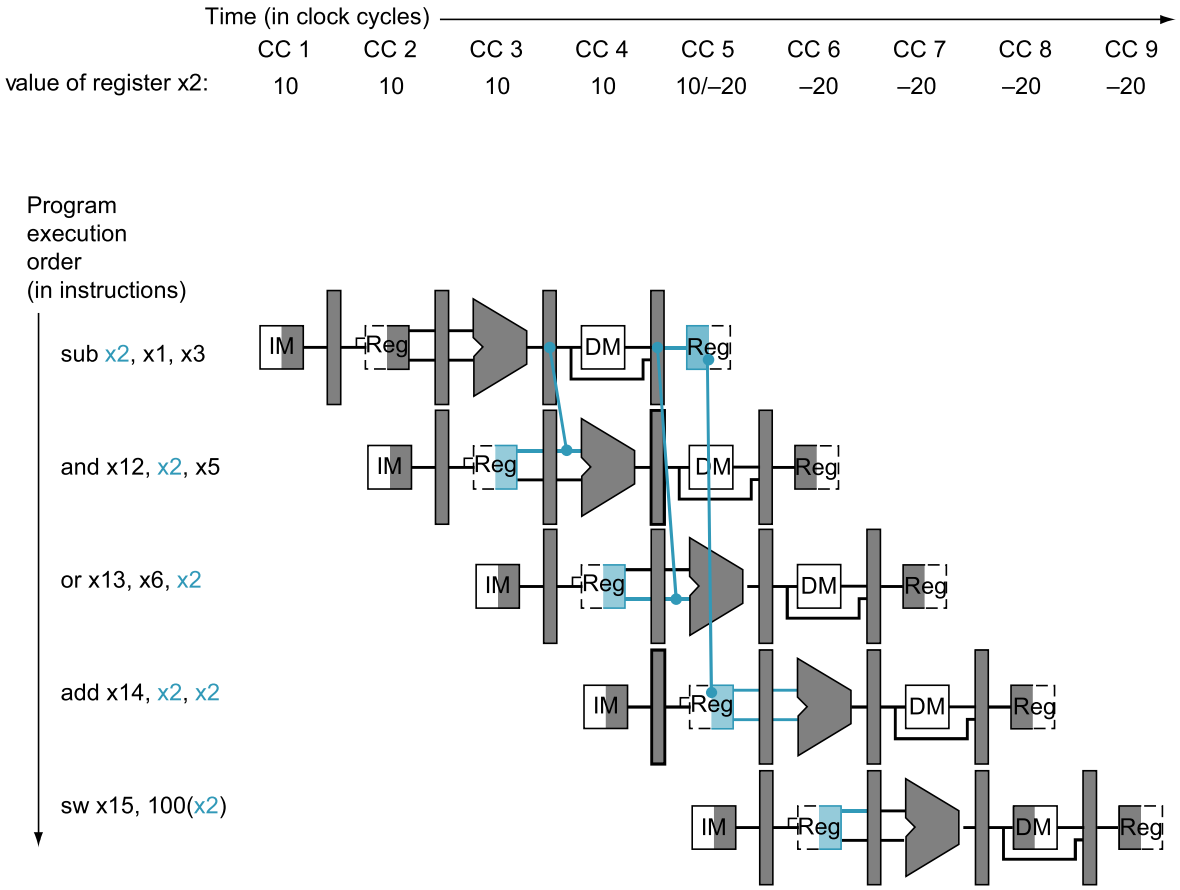
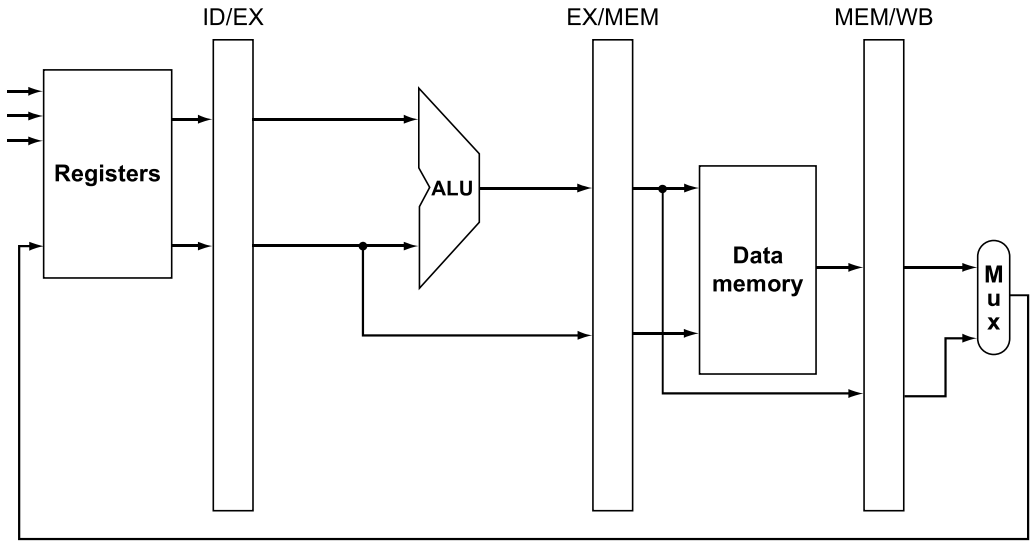
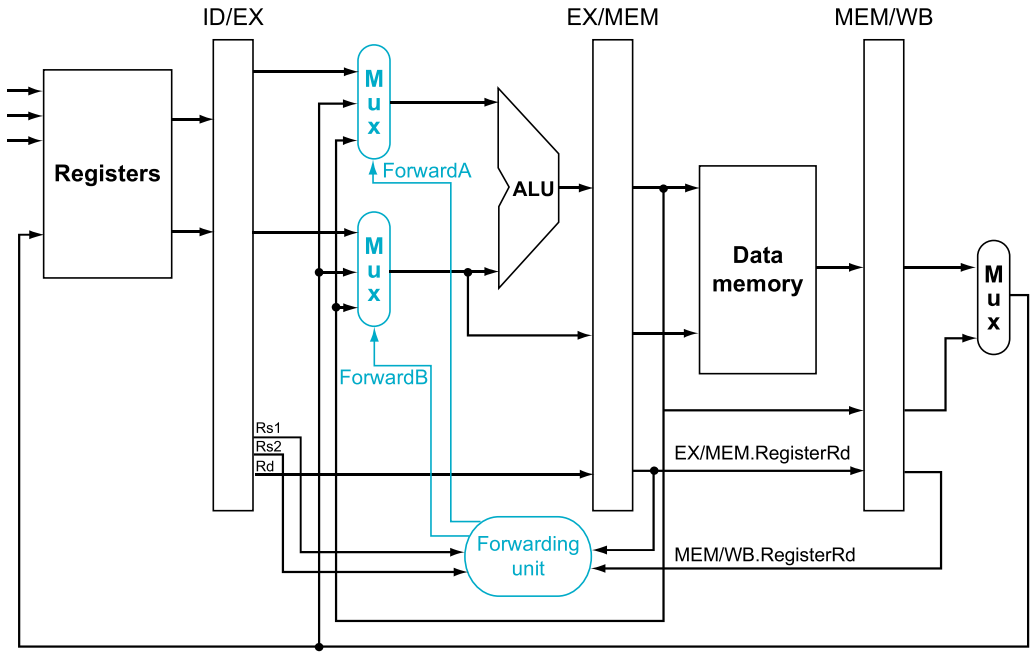



**FIGURE 4.55 The dependences between the pipeline registers move forward in time, so it is possible to supply the inputs to the ALU needed by the `and` instruction and `or` instruction by forwarding the results found in the pipeline registers.** The values in the pipeline registers show that the desired value is available before it is written into the register file. We assume that the register file forwards values that are read and written during the same clock cycle, so the `add` does not stall, but the values come from the register file instead of a pipeline register. Register file "forwarding"—that is, the read gets the value of the write in that clock cycle—is why clock cycle 5 shows register `x2` having the value 10 at the beginning and –20 at the end of the clock cycle.




a. No forwarding




b. With forwarding

**FIGURE 4.56 On the top are the ALU and pipeline registers before adding forwarding.** On the bottom, the multiplexors have been expanded to add the forwarding paths, and we show the forwarding unit. The new hardware is shown in color. This figure is a stylized drawing, however, leaving out details from the full datapath such as the sign extension hardware.

| Mux control | Source | Explanation |
|---|---|---|
| ForwardA = 00 | ID/EX | The first ALU operand comes from the register file. |
| ForwardA = 10 | EX/MEM | The first ALU operand is forwarded from the prior ALU result. |
| ForwardA = 01 | MEM/WB | The first ALU operand is forwarded from data memory or an earlier ALU result. |
| ForwardB = 00 | ID/EX | The second ALU operand comes from the register file. |
| ForwardB = 10 | EX/MEM | The second ALU operand is forwarded from the prior ALU result. |
| ForwardB = 01 | MEM/WB | The second ALU operand is forwarded from data memory or an earlier ALU result. |

**FIGURE 4.57 The control values for the forwarding multiplexors in Figure 4.56.** The signed immediate that is another input to the ALU is described in the *Elaboration* at the end of this section.

of the ALU, and with the proper controls, we can run the pipeline at full speed in the presence of these data hazards.

For now, we will assume the only instructions we need to forward are the four R-format instructions: `add`, `sub`, `and`, and `or`. Figure 4.56 shows a close-up of the ALU and pipeline register before and after adding forwarding. Figure 4.57 shows the values of the control lines for the ALU multiplexors that select either the register file values or one of the forwarded values.

This forwarding control will be in the EX stage, because the ALU forwarding multiplexors are found in that stage. Thus, we must pass the operand register numbers from the ID stage via the ID/EX pipeline register to determine whether to forward values. Before forwarding, the ID/EX register had no need to include space to hold the rs1 and rs2 fields. Hence, they were added to ID/EX.

Let's now write both the conditions for detecting hazards, and the control signals to resolve them:

1. *EX hazard:*

```
if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 10

if  (EX/MEM.RegWrite
and (EX/MEM.RegisterRd ≠ 0)
and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 10
```

   This case forwards the result from the previous instruction to either input of the ALU. If the previous instruction is going to write to the register file, and the write register number matches the read register number of ALU inputs A or B, provided it is not register 0, then steer the multiplexor to pick the value instead from the pipeline register EX/MEM.

2. *MEM hazard:*

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
```

```
if  (MEM/WB.RegWrite
and (MEM/WB.RegisterRd ≠ 0)
and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

As mentioned above, there is no hazard in the WB stage, because we assume that the register file supplies the correct result if the instruction in the ID stage reads the same register written by the instruction in the WB stage. Such a register file performs another form of forwarding, but it occurs within the register file.

One complication is potential data hazards between the result of the instruction in the WB stage, the result of the instruction in the MEM stage, and the source operand of the instruction in the ALU stage. For example, when summing a vector of numbers in a single register, a sequence of instructions will all read and write to the same register:

```
add x1, x1, x2
add x1, x1, x3
add x1, x1, x4
. . .
```

In this case, the result should be forwarded from the MEM stage because the result in the MEM stage is the more recent result. Thus, the control for the MEM hazard would be (with the additions highlighted):

```
if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 0)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01

if   (MEM/WB.RegWrite
and  (MEM/WB.RegisterRd ≠ 0)
and  not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
        and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
and  (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01
```

Figure 4.58 shows the hardware necessary to support forwarding for operations that use results during the EX stage. Note that the EX/MEM.RegisterRd field is the register destination for either an ALU instruction or a load.

If you would like to see more illustrated examples using single-cycle pipeline drawings, **Section 4.14** has figures that show two pieces of RISC-V code with hazards that cause forwarding.

**Elaboration:** Forwarding can also help with hazards when store instructions are dependent on other instructions. Since they use just one data value during the MEM stage, forwarding is easy. However, consider loads immediately followed by stores, useful when performing memory-to-memory copies in the RISC-V architecture. Since copies are frequent, we need to add more forwarding hardware to make them run faster.

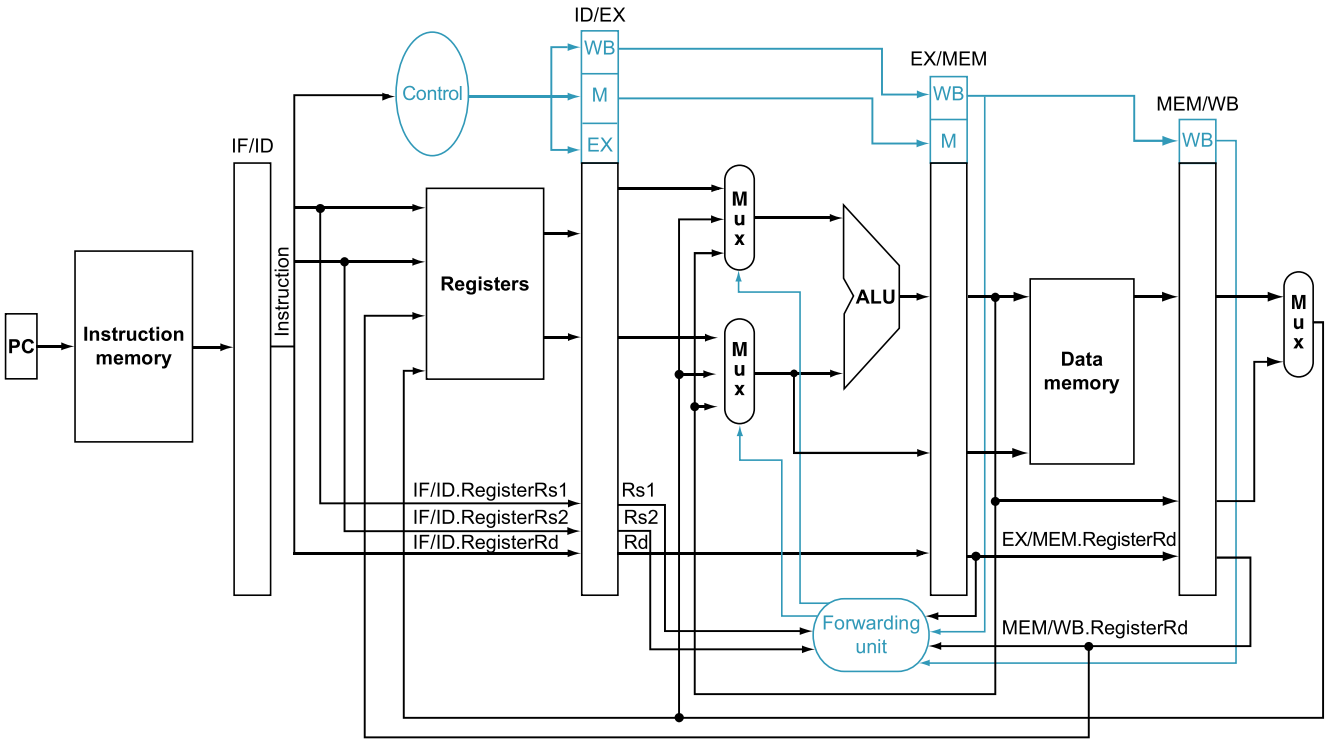



**FIGURE 4.58 The datapath modified to resolve hazards via forwarding.** Compared with the datapath in Figure 4.53, the additions are the multiplexors to the inputs to the ALU. This figure is a more stylized drawing, however, leaving out details from the full datapath, such as the branch hardware and the sign extension hardware.

If we were to redraw Figure 4.55, replacing the `sub` and `and` instructions with `lw` and `sw`, we would see that it is possible to avoid a stall, since the data exist in the MEM/WB register of a load instruction in time for its use in the MEM stage of a store instruction. We would need to add forwarding into the memory access stage for this option. We leave this modification as an exercise to the reader.

In addition, the signed-immediate input to the ALU, needed by loads and stores, is missing from the datapath in Figure 4.58. Since central control decides between register and immediate, and since the forwarding unit chooses the pipeline register for a register input to the ALU, the easiest solution is to add a 2:1 multiplexor that chooses between the ForwardB multiplexor output and the signed immediate. Figure 4.59 shows this addition.

## Data Hazards and Stalls

*If at first you don't succeed, redefine success.*

Anonymous

As we said in Section 4.6, one case where forwarding cannot save the day is when an instruction tries to read a register following a load instruction that writes the same register. Figure 4.60 illustrates the problem. The data is still being read from memory in clock cycle 4 while the ALU is performing the operation for the following instruction. Something must stall the pipeline for the combination of load followed by an instruction that reads its result.

Hence, in addition to a forwarding unit, we need a *hazard detection unit*. It operates during the ID stage so that it can insert the stall between the load and

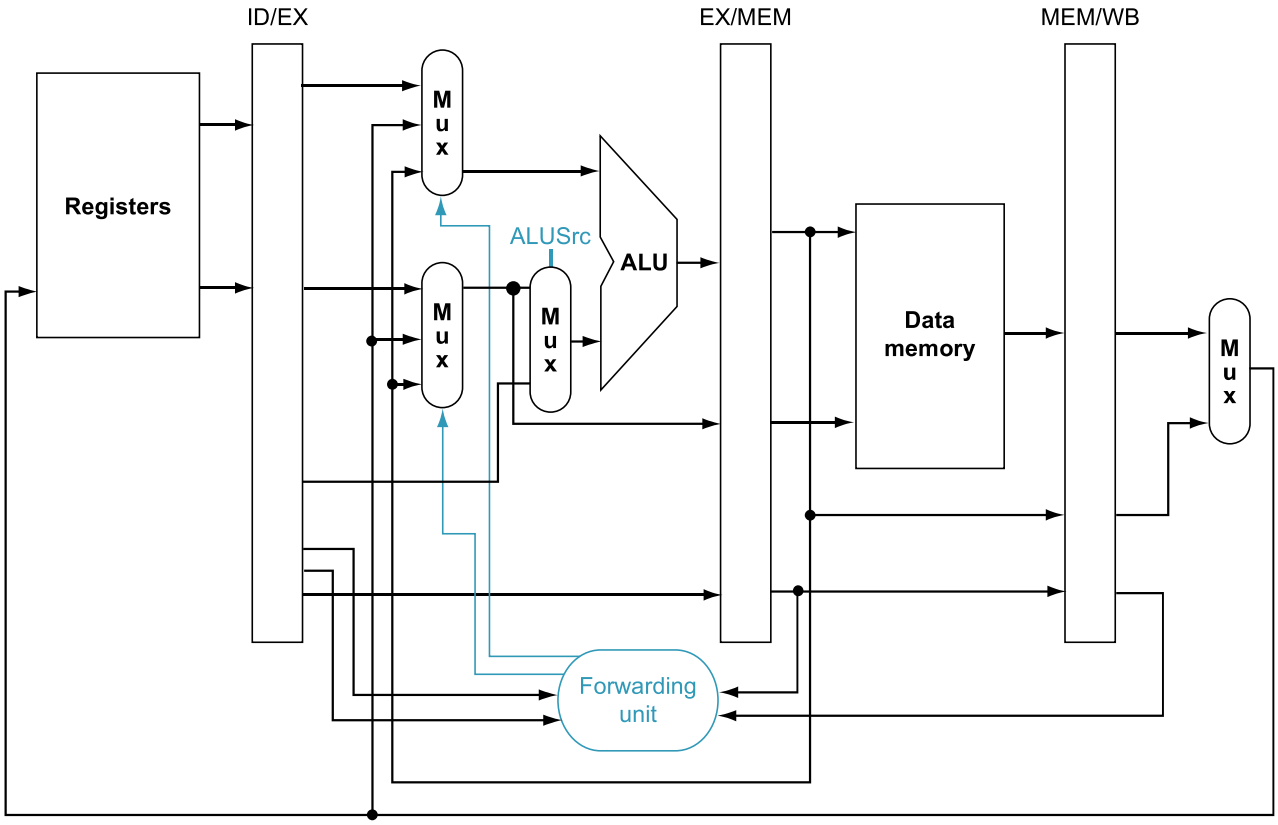



**FIGURE 4.59 A close-up of the datapath in Figure 4.56 shows a 2:1 multiplexor, which has been added to select the signed immediate as an ALU input.**

the instruction dependent on it. Checking for load instructions, the control for the hazard detection unit is this single condition:

```
if  (ID/EX.MemRead and
    ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or
      (ID/EX.RegisterRd = IF/ID.RegisterRs2)))
      stall the pipeline
```

Recall that we are using the RegisterRd to refer the register specified in instruction bits 11:7 for both load and R-type instructions. The first line tests to see if the instruction is a load: the only instruction that reads data memory is a load. The next two lines check to see if the destination register field of the load in the EX stage matches either source register of the instruction in the ID stage. If the condition holds, the instruction stalls one clock cycle. After this one-cycle stall, the forwarding logic can handle the dependence and execution proceeds. (If there were no forwarding, then the instructions in Figure 4.60 would need another stall cycle.)

If the instruction in the ID stage is stalled, then the instruction in the IF stage must also be stalled; otherwise, we would lose the fetched instruction. Preventing these two instructions from making progress is accomplished simply by preventing

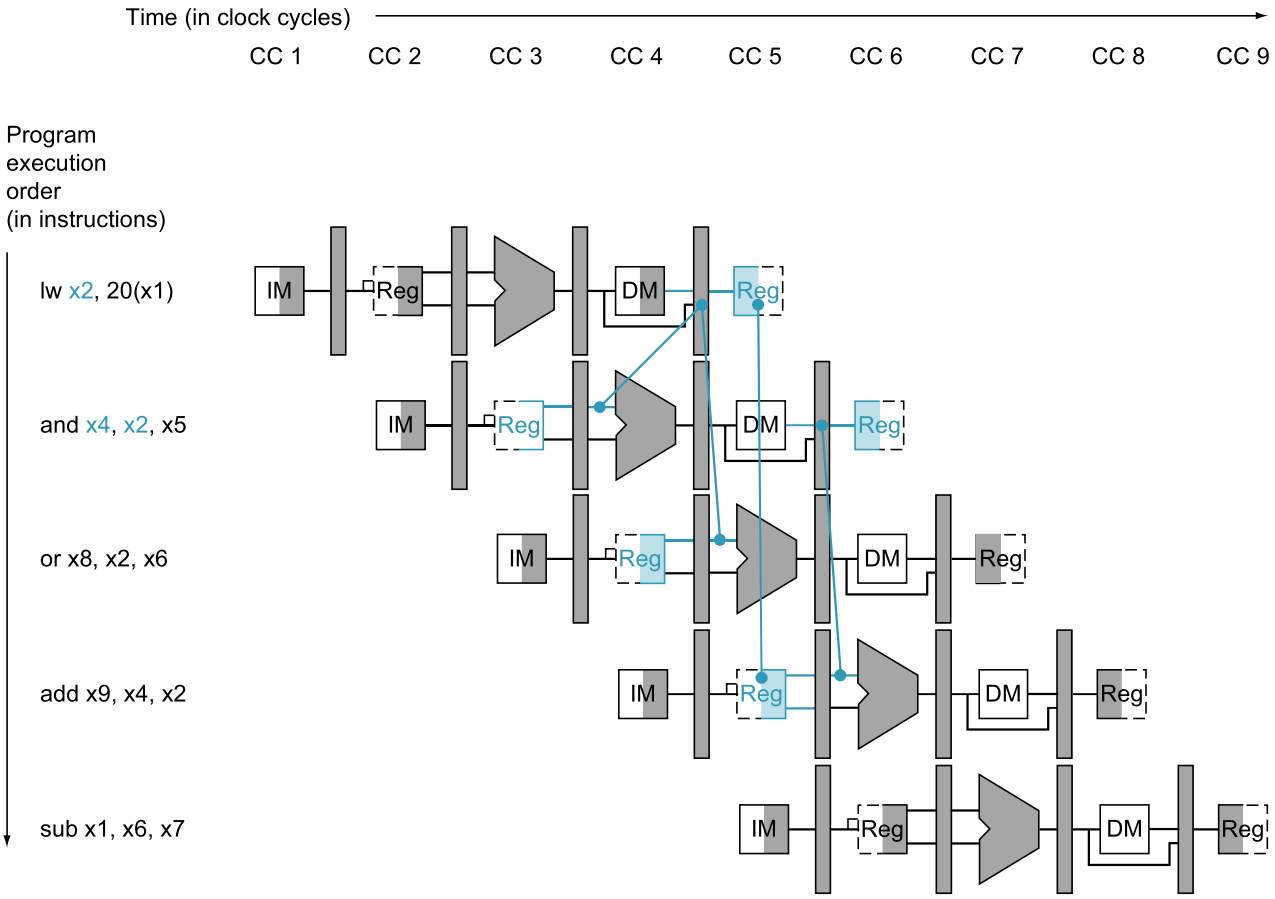



**FIGURE 4.60 A pipelined sequence of instructions.** Since the dependence between the load and the following instruction (`and`) goes backward in time, this hazard cannot be solved by forwarding. Hence, this combination must result in a stall by the hazard detection unit.

the PC register and the IF/ID pipeline register from changing. Provided these registers are preserved, the instruction in the IF stage will continue to be read using the same PC, and the registers in the ID stage will continue to be read using the same instruction fields in the IF/ID pipeline register. Returning to our favorite analogy, it's as if you restart the washer with the same clothes and let the dryer continue tumbling empty. Of course, like the dryer, the back half of the pipeline starting with the EX stage must be doing something; what it is doing is executing instructions that have no effect: **nops**.

**nops** An instruction that does no operation to change state.

How can we insert these nops, which act like bubbles, into the pipeline? In Figure 4.51, we see that deasserting all seven control signals (setting them to 0) in the EX, MEM, and WB stages will create a "do nothing" or nop instruction. By identifying the hazard in the ID stage, we can insert a bubble into the pipeline by changing the EX, MEM, and WB control fields of the ID/EX pipeline register to 0. These benign control values are percolated forward at each clock cycle with the proper effect: no registers or memories are written if the control values are all 0.

Figure 4.44 shows what really happens in the hardware: the pipeline execution slot associated with the `and` instruction is turned into a nop and all instructions beginning

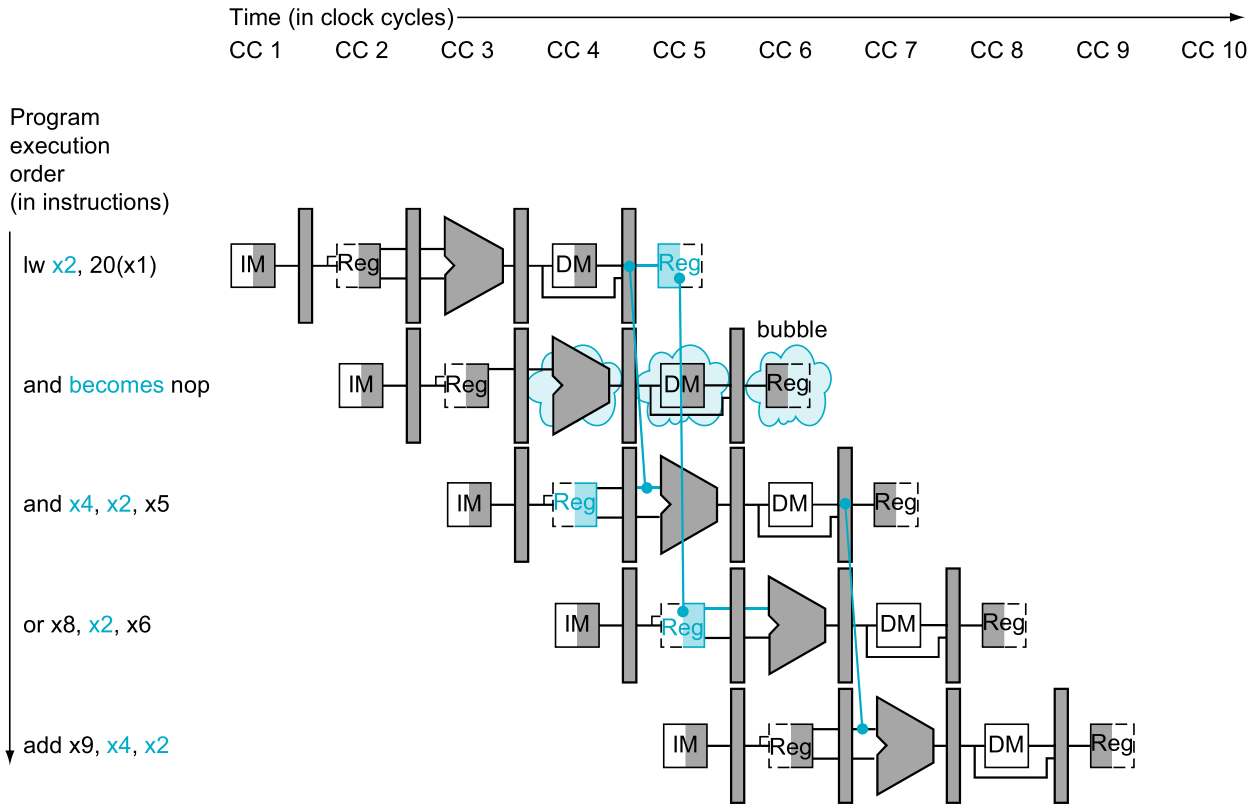



**FIGURE 4.61 The way stalls are really inserted into the pipeline.** A bubble is inserted beginning in clock cycle 4, by changing the `and` instruction to a nop. Note that the `and` instruction is really fetched and decoded in clock cycles 2 and 3, but its EX stage is delayed until clock cycle 5 (versus the unstalled position in clock cycle 4). Likewise, the `or` instruction is fetched in clock cycle 3, but its ID stage is delayed until clock cycle 5 (versus the unstalled clock cycle 4 position). After insertion of the bubble, all the dependences go forward in time and no further hazards occur.

with the `and` instruction are delayed one cycle. Like an air bubble in a water pipe, a stall bubble delays everything behind it and proceeds down the instruction pipe one stage each clock cycle until it exits at the end. In this example, the hazard forces the `and` and `or` instructions to repeat in clock cycle 4 what they did in clock cycle 3: `and` reads registers and decodes, and `or` is refetched from instruction memory. Such repeated work is what a stall looks like, but its effect is to stretch the time of the `and` and `or` instructions and delay the fetch of the `add` instruction.

Figure 4.62 highlights the pipeline connections for both the hazard detection unit and the forwarding unit. As before, the forwarding unit controls the ALU multiplexors to replace the value from a general-purpose register with the value from the proper pipeline register. The hazard detection unit controls the writing of the PC and IF/ID registers plus the multiplexor that chooses between the real control values and all 0s. The hazard detection unit stalls and deasserts the control fields if the load-use hazard test above is true. If you would like to see more details, **Section 4.14** gives an example illustrated using single-clock pipeline diagrams of RISC-V code with hazards that cause stalling.

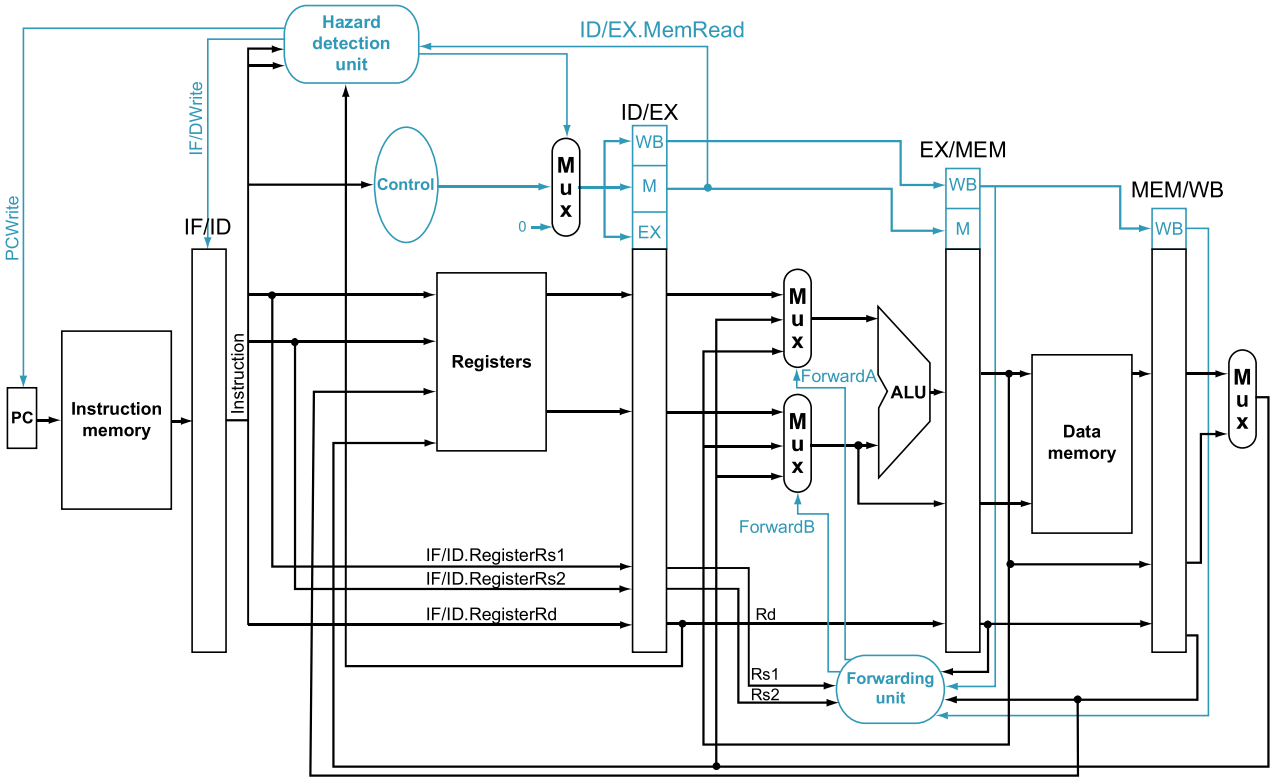



**FIGURE 4.62 Pipelined control overview, showing the two multiplexors for forwarding, the hazard detection unit, and the forwarding unit.** Although the ID and EX stages have been simplified—the sign-extended immediate and branch logic are missing—this drawing gives the essence of the forwarding hardware requirements.

Although the compiler generally relies upon the hardware to resolve hazards and thereby ensure correct execution, the compiler must understand the pipeline to achieve the best performance. Otherwise, unexpected stalls will reduce the performance of the compiled code.

**The BIG Picture**

**Elaboration:** Regarding the remark earlier about setting control lines to 0 to avoid writing registers or memory: only the signals RegWrite and MemWrite need be 0, while the other control signals can be don't cares.

## 4.9 Control Hazards

*There are a thousand hacking at the branches of evil to one who is striking at the root.*

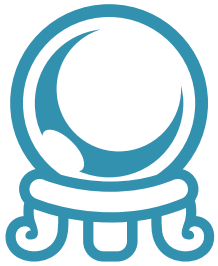
Henry David Thoreau, *Walden*, 1854

Thus far, we have limited our concern to hazards involving arithmetic operations and data transfers. However, as we saw in Section 4.6, there are also pipeline hazards involving conditional branches. Figure 4.63 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. As mentioned in Section 4.6, this delay in determining the proper instruction to fetch is called a *control hazard* or *branch hazard*, in contrast to the *data hazards* we have just examined.

This section on control hazards is shorter than the previous sections on data hazards. The reasons are that 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. We look at two schemes for resolving control hazards and one optimization to improve these schemes.

## Assume Branch Not Taken



PREDICTION

**flush** To discard instructions in a pipeline, usually due to an unexpected event.

As we saw in Section 4.6, stalling until the branch is complete is too slow. One improvement over branch stalling is to **predict** that the conditional branch will not be taken and thus continue execution down the sequential instruction stream. If the conditional branch is taken, the instructions that are being fetched and decoded must be discarded. Execution continues at the branch target. If conditional branches are untaken half the time, and if it costs little to discard the instructions, this optimization halves the cost of control hazards.

To discard instructions, we merely change the original control values to 0s, much as we did to stall for a load-use data hazard. The difference is that we must also change the three instructions in the IF, ID, and EX stages when the branch reaches the MEM stage; for load-use stalls, we just change control to 0 in the ID stage and let them percolate through the pipeline. Discarding instructions, then, means we must be able to **flush** instructions in the IF, ID, and EX stages of the pipeline.

## Reducing the Delay of Branches

One way to improve conditional branch performance is to reduce the cost of the taken branch. Thus far, we have assumed the next PC for a branch is selected in the MEM stage, but if we move the conditional branch execution earlier in the pipeline, then fewer instructions need be flushed. Moving the branch decision up requires two actions to occur earlier: computing the branch target address and evaluating the branch decision. The easy part of this change is to move up the branch address calculation. We already have the PC value and the immediate field in the IF/ID pipeline register, so we just move the branch adder from the EX stage to the ID stage; of course, the address calculation for branch targets will be performed for all instructions, but only used when needed.

The harder part is the branch decision itself. For branch if equal, we would compare two register reads during the ID stage to see if they are equal. Equality can be tested by XORing individual bit positions of two registers and ORing the XORed result. (A zero output of the OR gate means the two registers are equal.) Moving

the branch test to the ID stage implies additional forwarding and hazard detection hardware, since a branch dependent on a result still in the pipeline must still work properly with this optimization. For example, to implement branch if equal (and its inverse), we will need to forward results to the equality test logic that operates during ID. There are two complicating factors:

1. During ID, we must decode the instruction, decide whether a bypass to the equality test unit is needed, and complete the equality test so that if the instruction is a branch, we can set the PC to the branch target address. Forwarding for the operand of branches was formerly handled by the ALU forwarding logic, but the introduction of the equality test unit in ID will require new forwarding logic. Note that the bypassed source operands of a branch can come from either the EX/MEM or MEM/WB pipeline registers.

2. Because the value in a branch comparison is needed during ID but may be produced later in time, it is possible that a data hazard can occur and a stall will be needed. For example, if an ALU instruction immediately preceding a branch produces the operand for the test in the conditional branch, a stall will be required, since the EX stage for the ALU instruction will occur after the ID cycle of the branch. By extension, if a load is immediately followed by a conditional branch that depends on the load result, two stall cycles will be needed, as the result from the load appears at the end of the MEM cycle but is needed at the beginning of ID for the branch.

Despite these difficulties, moving the conditional branch execution to the ID stage is an improvement, because it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched. The exercises explore the details of implementing the forwarding path and detecting the hazard.

To flush instructions in the IF stage, we add a control line, called *IF.Flush*, that zeros the instruction field of the IF/ID pipeline register. Clearing the register transforms the fetched instruction into a `nop`, an instruction that has no action and changes no state.

**EXAMPLE**

**Pipelined Branch**

Show what happens when the branch is taken in this instruction sequence, assuming the pipeline is optimized for branches that are not taken, and that we moved the branch execution to the ID stage:

```
36  sub  x10, x4, x8
40  beq  x1,  x3, 16  // PC-relative branch to 40+16*2=72
44  and  x12, x2, x5
48  or   x13, x2, x6
```

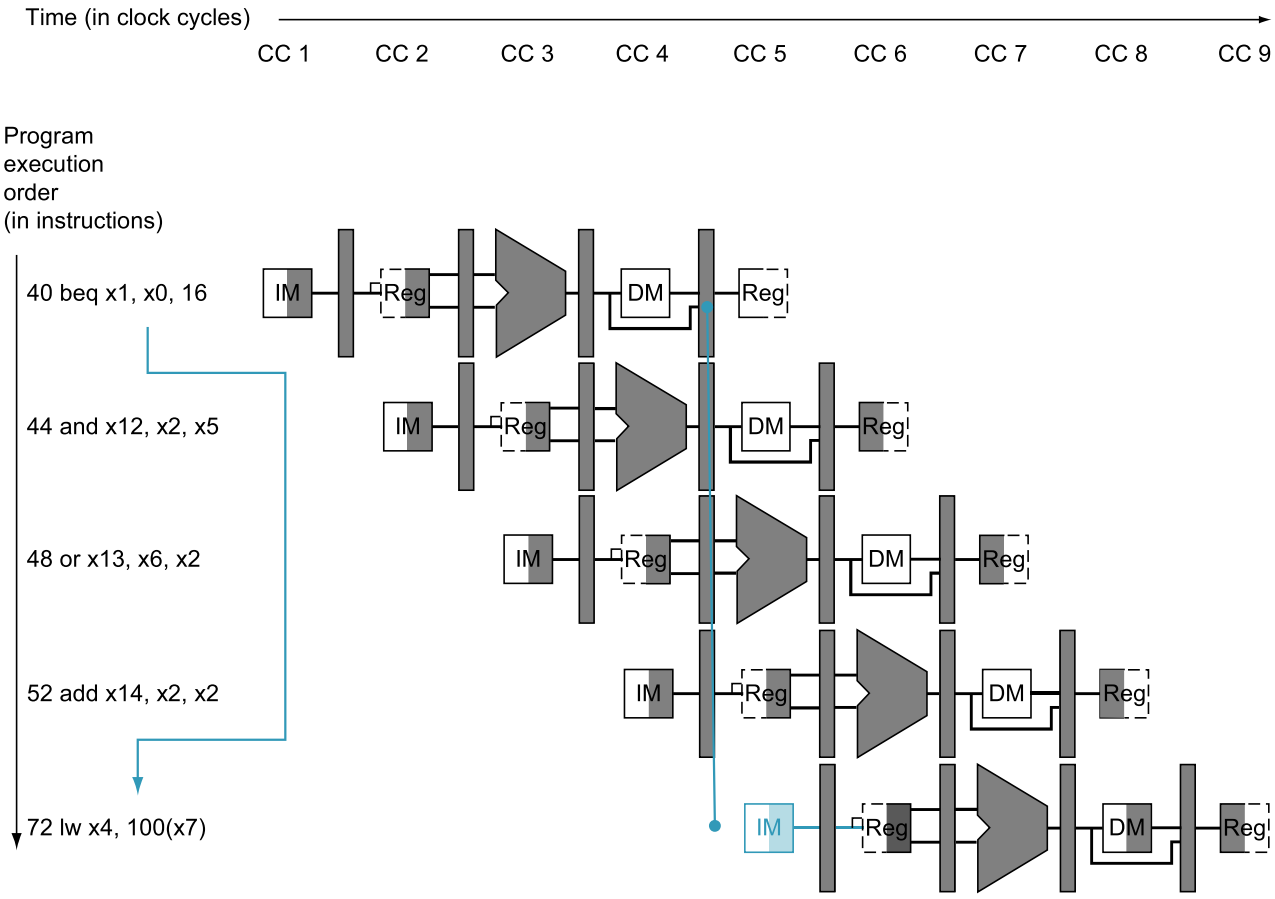
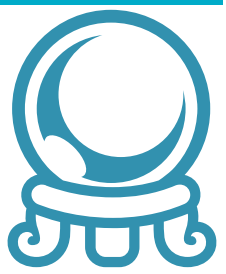



**FIGURE 4.63 The impact of the pipeline on the branch instruction.** The numbers to the left of the instruction (40, 44, …) are the addresses of the instructions. Since the branch instruction decides whether to branch in the MEM stage—clock cycle 4 for the `beq` instruction above—the three sequential 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 4.33 assumed extra hardware to reduce the control hazard to one clock cycle; this figure uses the nonoptimized datapath.)

```
52  add  x14, x4, x2
56  sub  x15, x6, x7
. . .
72  lw   x4, 50(x7)
```

**ANSWER**

Figure 4.64 shows what happens when a conditional branch is taken. Unlike Figure 4.63, there is only one pipeline bubble on a taken branch.




## Dynamic Branch Prediction

Assuming a conditional branch is not taken is one simple form of *branch prediction*. In that case, we predict that conditional branches are untaken, flushing the pipeline when we are wrong. For the simple five-stage pipeline, such an approach,

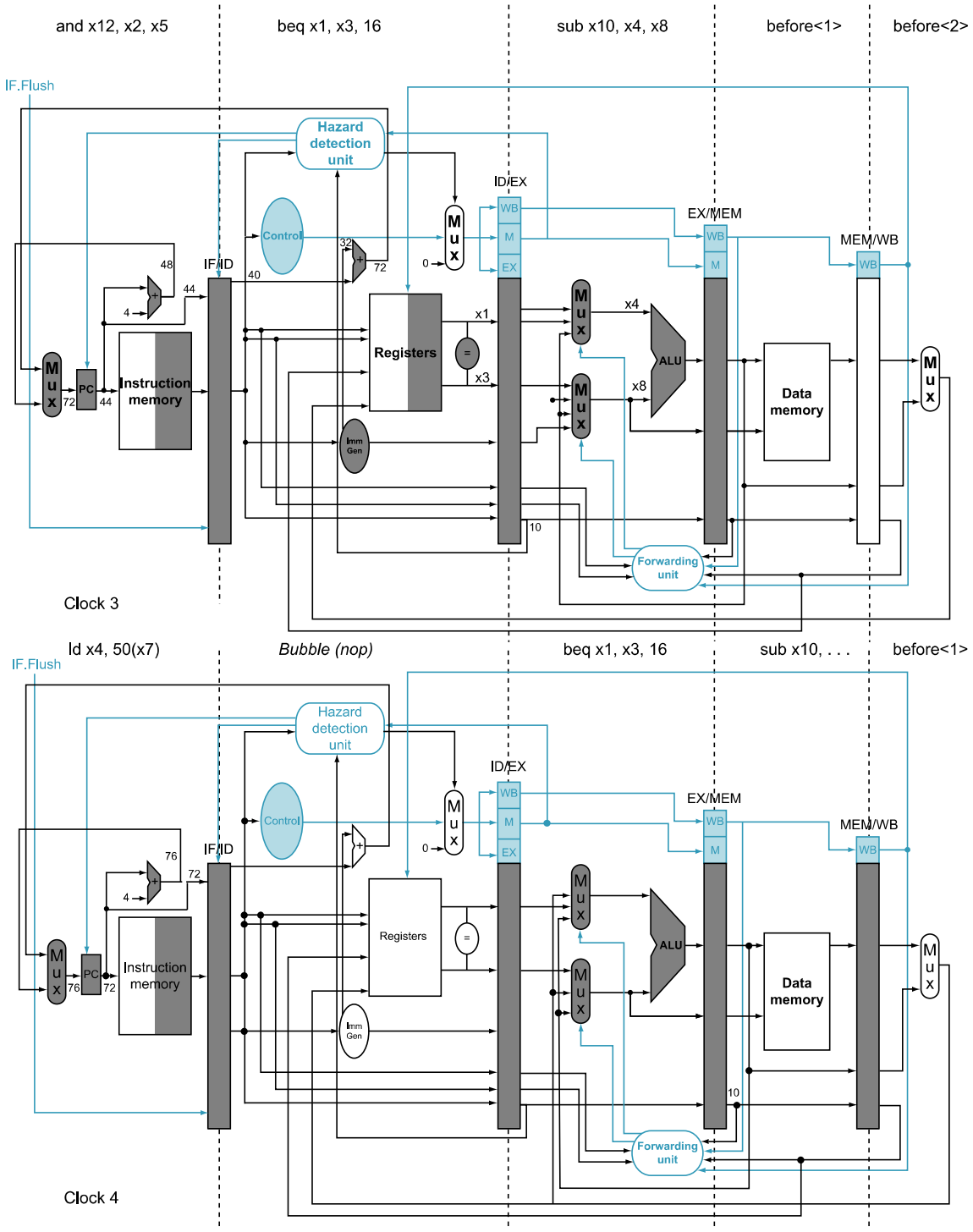



**FIGURE 4.64 The ID stage of clock cycle 3 determines that a branch must be taken, so it selects 72 as the next PC address and zeros the instruction fetched for the next clock cycle.** Clock cycle 4 shows the instruction at location 72 being fetched and the single bubble or `nop` instruction in the pipeline because of the taken branch.

possibly coupled with compiler-based prediction, is probably adequate. With deeper pipelines, the branch penalty increases when measured in clock cycles. Similarly, with multiple issue (see Section 4.11), the branch penalty increases in terms of instructions lost. This combination means that in an aggressive pipeline, a simple static prediction scheme will probably waste too much performance. As we mentioned in Section 4.6, with more hardware it is possible to try to **predict** branch behavior during program execution.

**dynamic branch prediction** Prediction of branches at runtime using runtime information.

One approach is to look up the address of the instruction to see if the conditional 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**.

**branch prediction buffer** Also called **branch history** table. A small memory that is indexed by the lower portion of the address of the branch instruction and that contains one or more bits indicating whether the branch was recently taken or not.

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.

This prediction uses the simplest sort of buffer; we don't know, in fact, if the prediction is the right one—it may have been put there by another conditional branch that has the same low-order address bits. However, this doesn't affect correctness. Prediction is just a hint that we hope is accurate so fetching begins in the predicted direction. If the hint turns out to be wrong, the incorrectly predicted instructions are deleted, the prediction bit is inverted and stored back, and the proper sequence is fetched and executed.

This simple 1-bit prediction scheme has a performance shortcoming: even if a conditional branch is almost always taken, we can predict incorrectly twice, rather than once, when it is not taken. The following example shows this dilemma.

**EXAMPLE**

### Loops and Prediction

Consider a loop branch that branches nine times in a row, and then is not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?

**ANSWER**

The steady-state prediction behavior will mispredict on the first and last loop iterations. Mispredicting the last iteration is inevitable since the prediction bit will indicate taken, as the branch has been taken nine times in a row at that point. The misprediction on the first iteration happens because the bit is flipped on prior execution of the last iteration of the loop, since the branch was not taken on that exiting iteration. Thus, the prediction accuracy for this branch that is taken 90% of the time is only 80% (two incorrect predictions and eight correct ones).

Ideally, the accuracy of the predictor would match the taken branch frequency for these highly regular branches. To remedy this weakness, we can use more prediction bits. In a 2-bit scheme, a prediction must be wrong twice before it is changed. Figure 4.65 shows the finite-state machine for a 2-bit prediction scheme.

A branch prediction buffer can be implemented as a small, special buffer accessed with the instruction address during the IF pipe stage. If the instruction is predicted as taken, fetching begins from the target as soon as the PC is known; as mentioned on page 308, it can be as early as the ID stage. Otherwise, sequential fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed shows Figure 4.65.

**branch target buffer** A structure that caches the destination PC or destination instruction for a branch. It is usually organized as a cache with tags, making it more costly than a simple prediction buffer.

**Elaboration:** A branch predictor tells us whether a conditional branch is taken, but still requires the calculation of the branch target. In the five-stage pipeline, this calculation takes one cycle, meaning that taken branches will have a one-cycle penalty. One approach is to use a cache to hold the destination program counter or destination instruction using a **branch target buffer**.

The 2-bit dynamic prediction scheme uses only information about a particular branch. Researchers noticed that using information about both a local branch and the global behavior of recently executed branches together yields greater prediction accuracy for the same number of prediction bits. Such predictors are called **correlating predictors**. A typical correlating predictor might have two 2-bit predictors for each branch, with the choice between predictors made based on whether the last executed branch was taken or not taken. Thus, the global branch behavior can be thought of as adding additional index bits for the prediction lookup.

**correlating predictor** A branch predictor that combines local behavior of a particular branch and global information about the behavior of some recent number of executed branches.

Another approach to branch prediction is the use of tournament predictors. A **tournament branch predictor** uses multiple predictors, tracking, for each branch, which predictor yields the best results. A typical tournament predictor might contain two predictions for each branch index: one based on local information and one based on global branch behavior. A selector would choose which predictor to use for any given prediction. The selector can operate similarly to a 1- or 2-bit predictor, favoring whichever of the two predictors has been more accurate. Some recent microprocessors use such ensemble predictors.

**tournament branch predictor** A branch predictor with multiple predictions for each branch and a selection mechanism that chooses which predictor to enable for a given branch.

**Elaboration:** One way to reduce the number of conditional branches is to add *conditional move* instructions. Instead of changing the PC with a conditional branch, the instruction conditionally changes the destination register of the move. For example, the ARMv8 instruction set architecture has a conditional select instruction called `CSEL`. It specifies a destination register, two source registers, and a condition. The destination register gets a value of the first operand if the condition is true and the second operand otherwise. Thus, `CSEL X8, X11, X4, NE` copies the contents of register 11 into register 8 if the condition codes say the result of the operation was not equal zero or a copy of register 4 into register 11 if it was zero. Hence, programs using the ARMv8 instruction set could have fewer conditional branches than programs written in RISC-V.

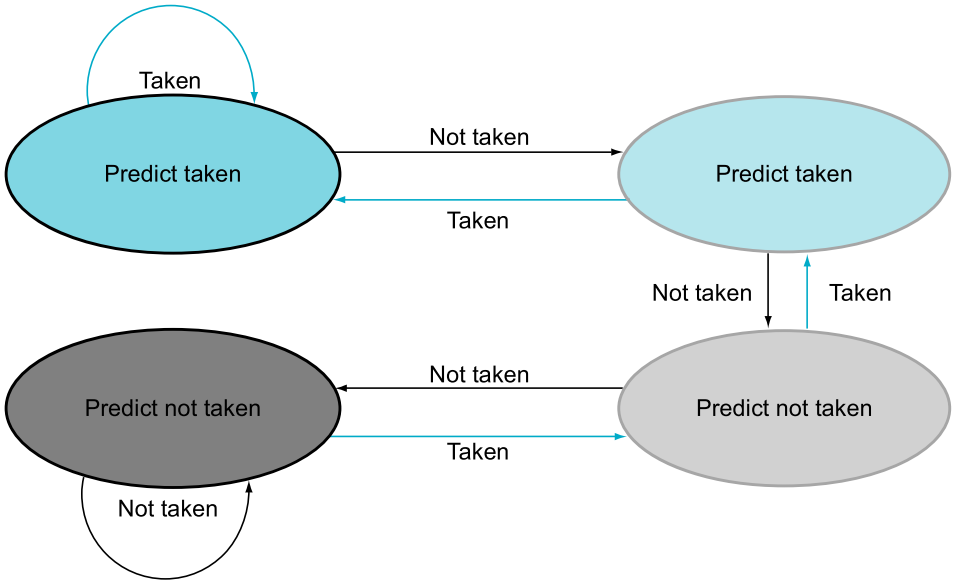



**FIGURE 4.65 The states in a 2-bit prediction scheme.** By using 2 bits rather than 1, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted only once. The 2 bits are used to encode the four states in the system. The 2-bit scheme is a general instance of a counter-based predictor, which is incremented when the prediction is accurate and decremented otherwise, and uses the mid-point of its range as the division between taken and not taken.

## Pipeline Summary

We started in the laundry room, showing principles of pipelining in an everyday setting. Using that analogy as a guide, we explained instruction pipelining step-by-step, starting with the single-cycle datapath and then adding pipeline registers, forwarding paths, data hazard detection, branch prediction, and flushing instructions on mispredicted branches or load-use data hazards. Figure 4.66 shows the final evolved datapath and control. We now are ready for yet another control hazard: the sticky issue of exceptions.

**Check Yourself** Consider three branch prediction schemes: predict not taken, predict taken, and dynamic prediction. Assume that they all have zero penalty when they predict correctly and two cycles when they are wrong. Assume that the average predict accuracy of the dynamic predictor is 90%. Which predictor is the best choice for the following branches?

1. A conditional branch that is taken with 5% frequency
2. A conditional branch that is taken with 95% frequency
3. A conditional branch that is taken with 70% frequency