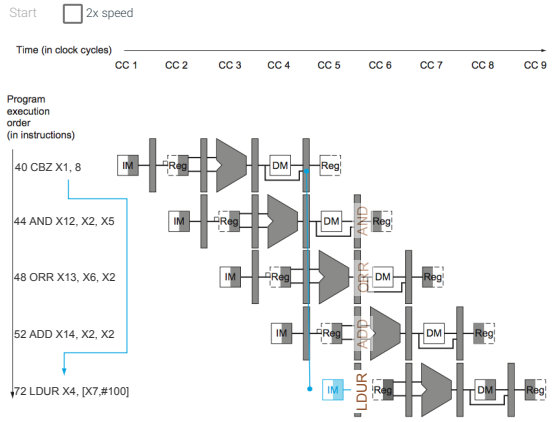
4.8 Control hazards

> "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 COD Section 4.5 (An overview of pipelining), there are also pipeline hazards involving conditional branches. The figure below 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 COD Section 4.5 (An overview of pipelining), 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.

In the animation below, 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 `CBZ` instruction below—the three sequential instructions that follow the branch will be fetched and begin execution. Without intervention, those three following instructions will begin execution before `CBZ` branches to `LDUR` at location 72. (COD Figure 4.30 (Pipeline showing stalling ...) assumed extra hardware to reduce the control hazard to one clock cycle; this animation uses the nonoptimized datapath.)

PARTICIPATION ACTIVITY 4.8.1: The impact of the pipeline on the branch instruction (COD Figure 4.60).



In CC4, the datapath determines the branch should be taken to address 72. The middle instructions (AND, ORR, ADD) must somehow be discarded.

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.

PARTICIPATION ACTIVITY 4.8.2: Control hazards.

Refer to the above animation.

1) Instruction `CBZ` is at address 40. If the value in register X1 is not equal to 0, the address of the next instruction that should execute is ____ten.

Check Show answer

2) Instruction `CBZ` is at address 40. If the value in register X1 is equal to 0, the address of the next instruction that should execute is ____ten.

Check Show answer

3) In the pipelined implementation above, whether the `CBZ` instruction's branch is taken is determined in which clock cycle? (Type CC1, or CC2, etc.).

Check Show answer

4) In the pipeline depiction above, even though whether the branch will be taken is unknown until CC4, the instruction after `CBZ`, namely `AND`, is fetched

anyways into the pipeline in which clock cycle? (Type CC1, or CC2, etc.).

Check Show answer

Assume branch not taken

As we saw in COD Section 4.5 (An overview of pipelining), 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.

Flush: To discard instructions in a pipeline, usually due to an unexpected event.

PARTICIPATION ACTIVITY 4.8.3: Flushing instructions.

Refer to the above animation.

1) If CBZ's branch is determined to be taken, the instructions that must be discarded from the pipeline are: AND, ORR, _____.
   - ○ ADD
   - ○ LDUR

2) To discard the AND instruction, the control values in the _____ pipeline register must be set to 0's.
   - ○ third
   - ○ fourth

3) If CBZ's branch is determined to be taken, the control values in how many pipeline registers shown above must be set to 0's?
   - ○ 3
   - ○ 4

4) Discarding instructions from a pipeline is known as _____ the pipeline.
   - ○ flushing
   - ○ purging

5) If CBZ's branch is not taken, only 1 instruction needs to be flushed.
   - ○ True
   - ○ False

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 compare and branch zero, we would compare a register read during the ID stage to see if it is zero. Zero can be tested by ORing all 64 bits. 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 compare and branch on zero (and its inverse), we will need to forward results to the zero 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 zero test unit is needed, and complete the zero 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 zero test unit in ID will require new forwarding logic. Note that the bypassed source operands of a branch can come from either the ALU/MEM or MEM/WB pipeline latches.
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 4.8.1: 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:

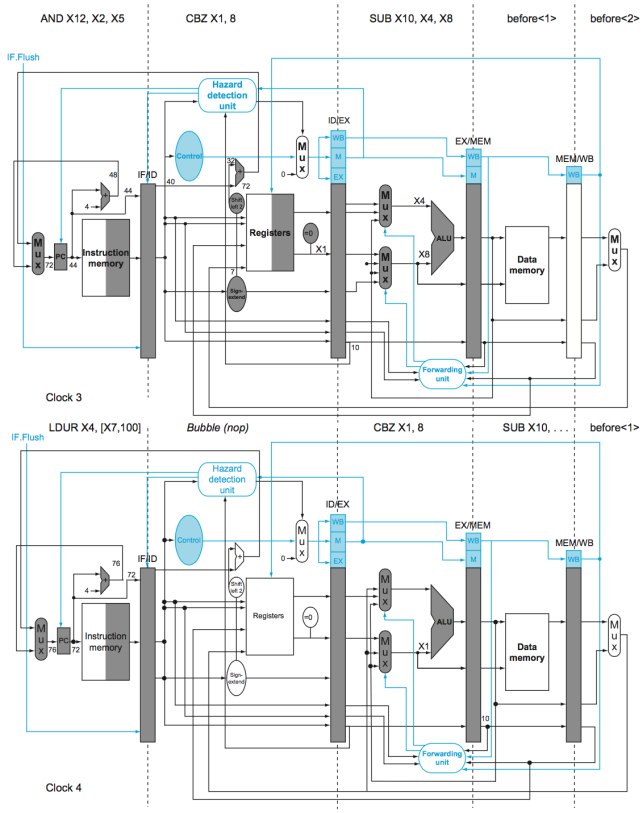
```
36 SUB X10, X4, X8
40 CBZ X1,  X3, 8   // PC-relative branch to 40+8*4=72
44 AND X12, X2, X5
48 ORR X13, X2, X6
52 ADD X14, X4, X2
56 SUB X15, X6, X7
. . .
72 LDUR  X4,  [X7,#50]
```

Answer

The figure below shows what happens when a conditional branch is taken. Unlike the figure above, there is only one pipeline bubble on a taken branch.

Figure 4.8.1: 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 (COD Figure 4.61).

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.



PARTICIPATION ACTIVITY 4.8.4: Reducing a branch's delay.

1) Two actions must be completed before a CBZ's branch can be taken, actions that take time. Obviously, one is to determine *whether* CBZ's register value is equal to 0. The other is to compute ____.

   - the CBZ's target address
   - the CBZ instruction's register address
   - the CBZ instruction's address

2) The action of computing the CBZ's target address can be done earlier, in the ID stage rather than the EX stage. That action means the target address

will be computed for *all* instructions, not just beq instructions. A problem that may occur with such computing for all instructions is ____.

- ○ branching to a wrong target address
- ○ longer flushing.
- ○ (no problem exists)

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, 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 COD Section 4.10 (Parallelism via instructions)), 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 COD Section 4.5 (An overview of pipelining), with more hardware it is possible to try to **predict** branch behavior during program execution.

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*.

> ***Dynamic branch prediction***: Prediction of branches at runtime using runtime information.

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.

> ***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.

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 correct, 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 4.8.2: 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?

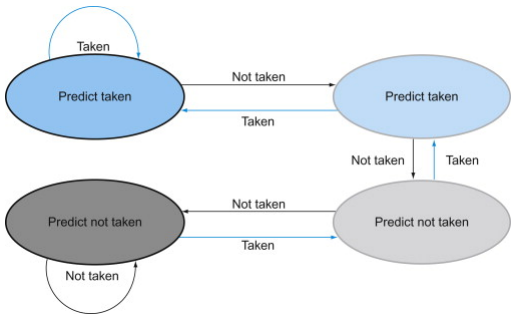
Answer

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, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must be wrong twice before it is changed. The figure below shows the finite-state machine for a 2-bit prediction scheme.

Figure 4.8.2: The states in a 2-bit prediction scheme (COD Figure 4.62).

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.



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; 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 as shown in the figure above.

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.

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.

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

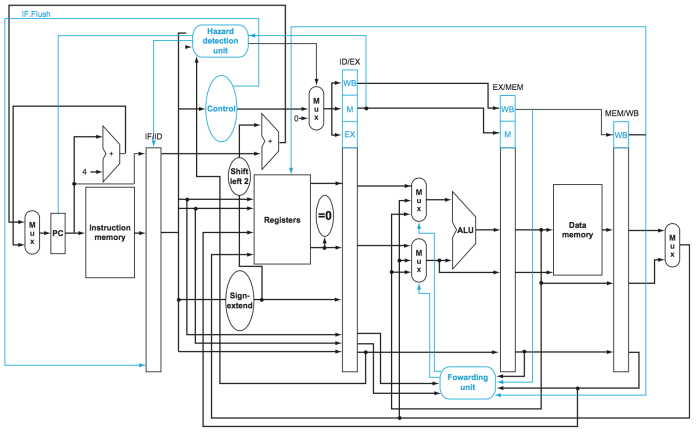
One way to reduce the number of conditional branches is to add conditional move instructions. Instead of changing the PC with a conditional branch, the instruction conditionally changes the destination register of the move. For example, the full ARMv8 instruction set architecture has a conditional select 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 full ARMv8 instruction set could have fewer conditional branches than in programs using just the LEGv8 core.*

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. The figure below shows the final evolved datapath and control. We now are ready for yet another control hazard: the sticky issue of exceptions.

Figure 4.8.3: The final datapath and control for this chapter (COD Figure 4.63).

Note that this is a stylized figure rather than a detailed datapath, so it's missing the ALUsrc Mux from COD Figure 4.56 (A close-up of the datapath in COD Figure 4.53 ...) and the multiplexor controls from COD Figure 4.50 (The corrected pipelined datapath ...).



PARTICIPATION ACTIVITY 4.8.5: Check yourself: Branch prediction schemes.

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 branch that is taken with 5% frequency.
   - ○ Predict not taken
   - ○ Predict taken
   - ○ Dynamic prediction

2) A branch that is taken with 95% frequency.
   - ○ Predict not taken
   - ○ Predict taken
   - ○ Dynamic prediction

3) A branch that is taken with 70% frequency.
   - ○ Predict not taken
   - ○ Predict taken
   - ○ Dynamic prediction

Provide feedback on this section