LECTURE 9

Pipeline Hazards

PIPELINED DATAPATH AND CONTROL

In the previous lecture, we finalized the pipelined datapath for instruction sequences which do not include hazards of any kind.

Remember that we have two kinds of hazards to worry about:

- ***Data hazards***: an instruction is unable to execute in the planned cycle because it is dependent on some data that has not yet been committed.

```
      lw    $s0, $s1, $s2   # $s0 written in cycle 5
      add   $s3, $s0, $s4   # $s0 read in cycle 3
```

- ***Control hazards***: we do not know which instruction needs to be executed next.

```
      beq   $t0, $t1, L1    # target in cycle 3
      add   $t0, $t0, 1     # loaded in cycle 2
L1:   sub   $t0, $t0, 1
```

DATA HAZARDS

Let us now turn our attention to data hazards. As we've already seen, we have two solutions for data hazards:

- ***Forwarding*** (or *bypassing*): the needed data is forwarded as soon as possible to the instruction which depends on it.
- ***Stalling***: the dependent instruction is "pushed back" for one or more clock cycles. Alternatively, you can think of stalling as the execution of a noop for one or more cycles.

DATA HAZARDS

Let's take a look at the following instruction sequence.

```
sub   $2,   $1, $3
and   $12, $2, $5
or    $13, $6, $2
add   $14, $2, $2
sw    $15, 100($2)
```

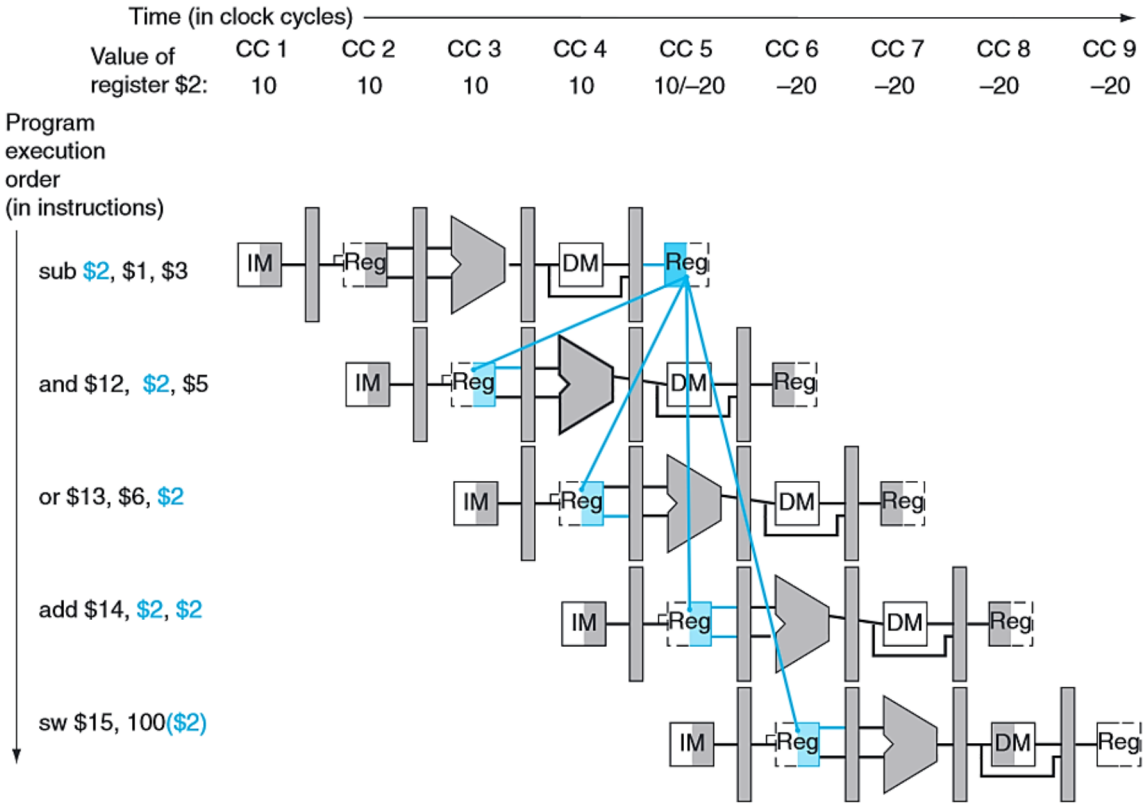
We have a number of dependencies here. The last four instructions, which read register $2, are all dependent on the first instruction, which writes a value to register $2. Let's naively pipeline these instructions and see what happens.

DATA HAZARDS

The blue lines indicate the data dependencies.

We can notice immediately that we have a problem.

The last four instructions require the register value to be -20 (the value during the WB stage of the first instruction). However, the middle three instructions will read the value to be 10 if we do not intervene.

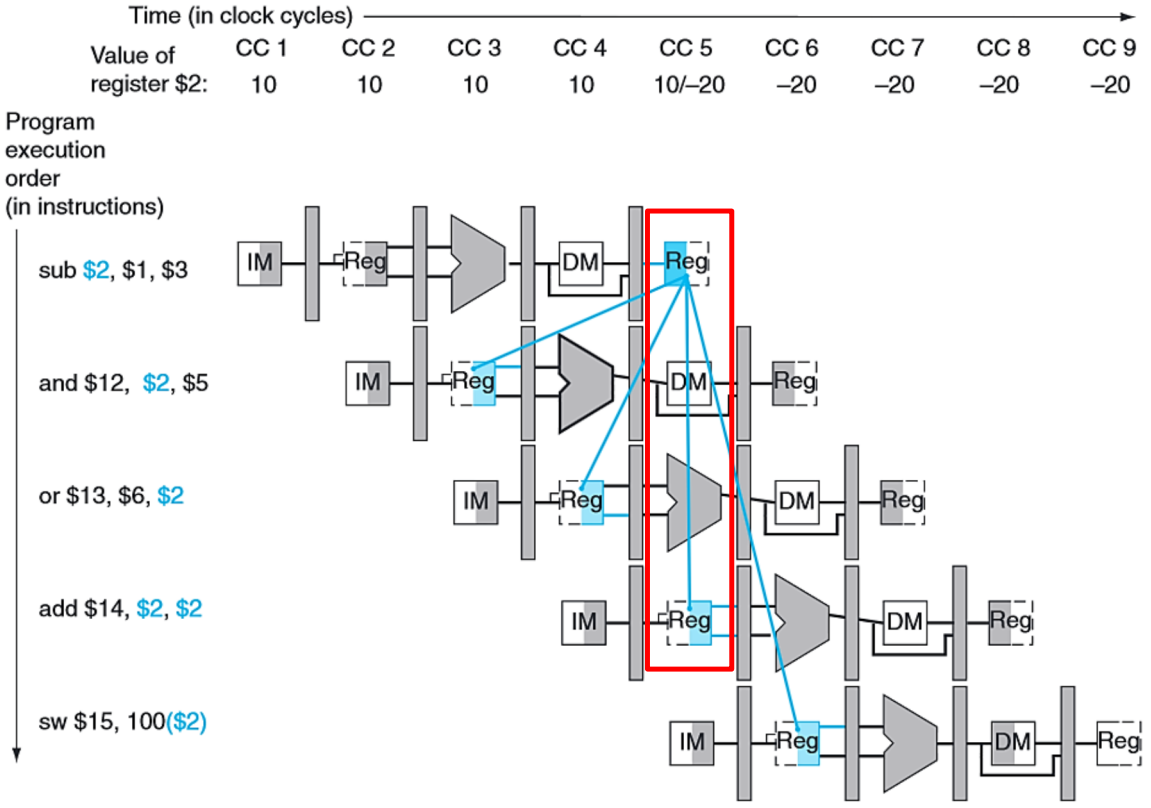


DATA HAZARDS

We can resolve the last potential hazard in our design of the register file unit.

Instruction 1 is writing the value of $2 to the register file in the same cycle that instruction 4 is reading the value of $2.

We assume that the **write operation takes place in the first half** of the clock cycle, while the **read operation takes place in the second half.** Therefore, the updated $2 value is available.

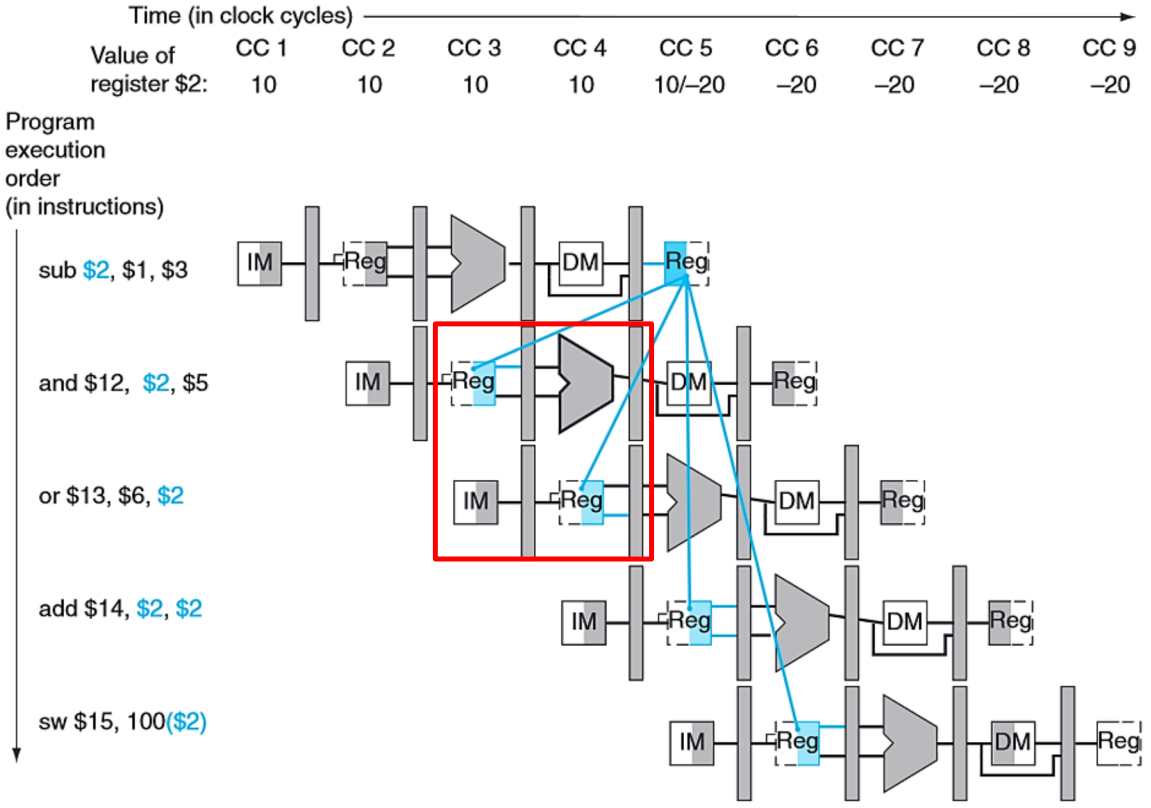


DATA HAZARDS

So the only data hazards occur for instructions 2 and 3.

In this style of representation, we can easily identify **true data hazards** as they are the only ones whose **dependency lines go back in time.**

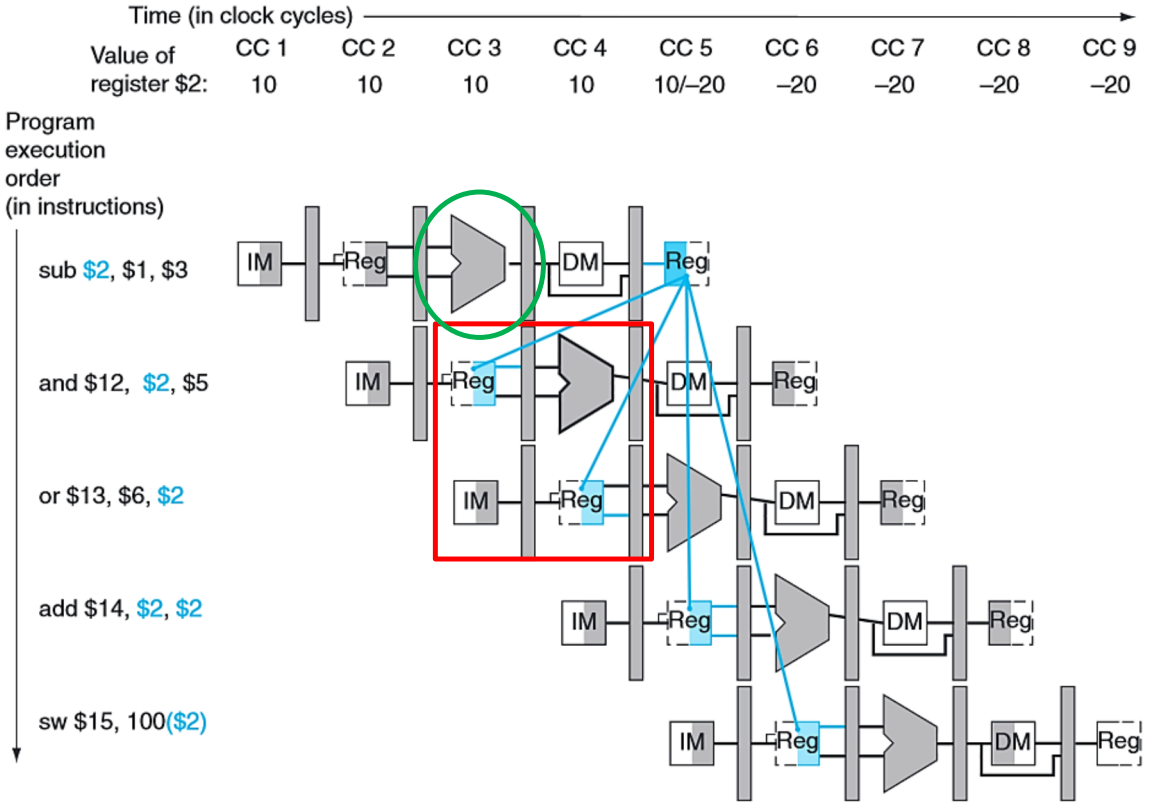
Note that instruction 2 reads $2 in cycle 3 and instruction 3 reads $2 in cycle 4.



DATA HAZARDS

Luckily **instruction 1 calculates the new values in cycle 3.**

If we simply *forward* the data as soon as it is calculated, then we will have it in time for the subsequent instructions to execute.

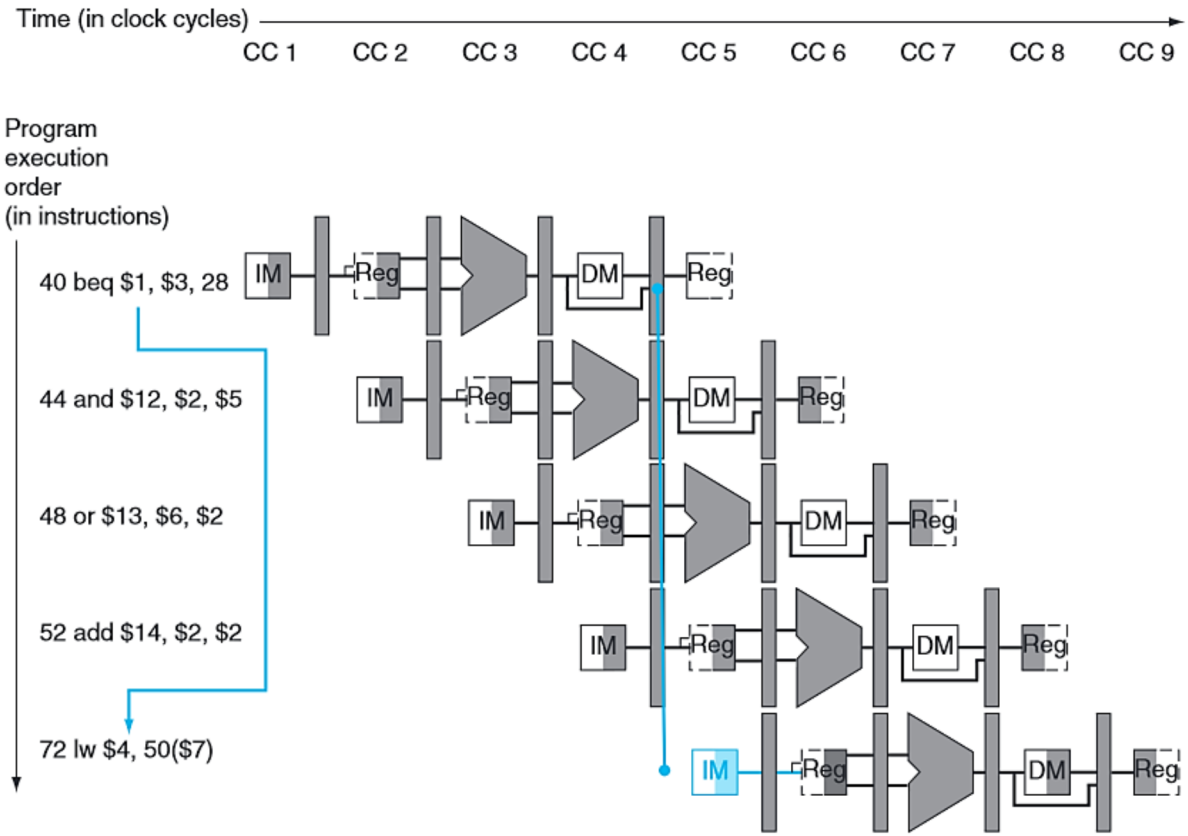


CONTROL HAZARDS

The simplest approach is to always **assume the branch isn't taken.** Then the correct instructions will already be executing anyway.

However, if we're proven wrong in the 4th stage of the branch instruction, then **the other three instructions need to be flushed from the datapath.**

This is done by **setting their control lines to 0 when the reach the EX stage.** The next significant instruction will be the branch target .



CONTROL HAZARDS

So, we've looked at two possible solutions:

- Assuming branch not taken.
  - Easy to implement.
  - High cost – three stalls if wrong.
- Performing branching in the ID stage.
  - Harder to implement – must add forwarding and hazard control earlier.
  - Lower cost – one stall if branch is taken.

We have another solution we can try: branch prediction.

CONTROL HAZARDS

In *branch prediction*, we attempt to predict the branching decisions and act accordingly.

When we assumed the branch wasn't taken, we were making a simple static prediction. Luckily, the performance cost on a 5-stage pipeline is low but on a *deeper* pipeline with many more stages, that could be a huge performance cost!

In *dynamic branch prediction*, we look up the address of the instruction to see if the branch was taken last time. If so, we will predict that the branch will be taken again and optimistically fetch the instructions from the branch target rather than the subsequent instructions.

CONTROL HAZARDS

A branch prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory simply contains one bit indicating whether the branch was taken last time or not.

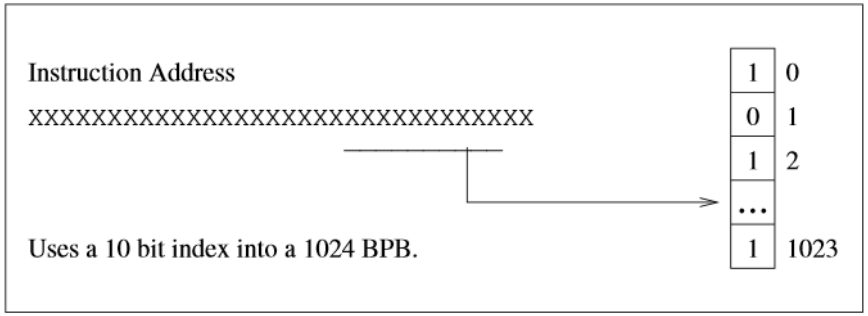
This isn't a perfect scheme by any means. It's just a simple mechanism that might give us a hint as to what the right decision might be. Note:

- The buffer is shared between branches with the same lower addresses. A buffer value may reflect another branch instruction.
- The branch instruction may simply make a different decision than it did before.

CONTROL HAZARDS

Here's how the 1-bit branch prediction buffer works:

- Each element in the buffer contains a single bit indicating whether the branch prediction was taken last time.
- We make our prediction based on the bit found in the buffer.
- If the prediction turns out to be incorrect, then we flip the bit in the buffer and correct the pipeline.



CONTROL HAZARDS

Consider a loop that branches nine times in a row, 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?

```
int i = 0;
do{
    /* loop body */
    i = i + 1
}while(i < 10);
```

```
        add    $t0, $0, $0
L1:     /* loop body*/
        addi   $t0, $t0, 1
        slti   $t1, $t0, 10
        bne    $t1, $0, L1
```

CONTROL HAZARDS

Consider a loop that branches nine times in a row, 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?

```
int i = 0;
do{
    /* loop body */
    i = i + 1
}while(i < 10);
```

```
        add    $t0, $0, $0
L1:     /* loop body*/
        addi   $t0, $t0, 1
        slti   $t1, $t0, 10
        bne    $t1, $0, L1
```

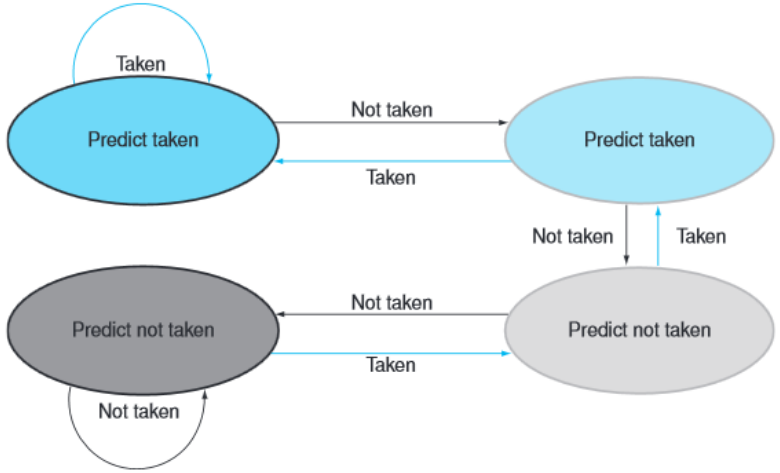
The prediction behavior will mispredict on both the first and last loop iterations. The last loop iteration prediction happens because we've already taken the branch nine-times so far. The first loop iteration happens because the bit was flipped on the last iteration of the previous execution. So, the prediction accuracy is 80%.

CONTROL HAZARDS

To increase this prediction accuracy, we can use a 2-bit prediction buffer. In the 2-bit scheme, a prediction must be wrong twice before the bit is flipped.

This way, a branch that strongly favors a particular decision (such as in the previous slide) will be wrong only once.

We can access the buffer during the IF stage to determine whether the next instruction needs to be calculated or we can continue with sequential execution.



CONTROL HAZARDS

As you've probably realized by now, even if we can predict a branch will be taken, we cannot write the branch target to PC until the ID stage. Therefore, we have a 1-cycle stall on predict-taken branches.

To avoid this, we use a *branch target buffer*. A branch target buffer contains a tag (the higher bits of the instruction) and the target address of the branch.

If the BPB predicts the branch is taken, and the higher order bits of the instruction match the BTB tag, then we write the target to PC.