Now let's turn our attention to control hazards, illustrated by the code fragment shown here.

Which instruction should be executed after the BNE?

If the value in R3 is non-zero, ADDC should be executed.

If the value in R3 is zero, the next instruction should be SUB.

If the current instruction is an explicit transfer of control (i.e., JMPs or branches), the choice of the next instruction depends on the execution of the current instruction.

What are the implications of this dependency on our execution pipeline?

How does the unpipelined implementation determine the next instruction?

For branches (BEQ or BNE), the value to be loaded into the program counter depends on

(1) the opcode, i.e., whether the instruction is a BEQ or a BNE,

(2) the current value of the program counter since that's used in the offset calculation, and

(3) the value stored in the register specified by the RA field of the instruction since that's the value tested by the branch.

For JMP instructions, the next value of the program counter depends once again on the opcode field and the value of the RA register.

For all other instructions, the next PC value depends only the opcode of the instruction and the value PC+4.

Exceptions also change the program counter.

We'll deal with them later in the lecture.

The control hazard is triggered by JMP and branches since their execution depends on the value in the RA register, i.e., they need to read from the register file, which happens in the RF pipeline stage.

Our bypass mechanisms ensure that we'll use the correct value for the RA register even if it's not yet written into the register file.

What we're concerned about here is that the address of the instruction following the JMP or branch will be loaded into program counter at the end of the cycle when the JMP or branch is in the RF stage.

But what should the IF stage be doing while all this is going on in RF stage?

The answer is that in the case of JMPs and taken branches, we don't know what the IF stage should be doing until those instructions are able to access the value of the RA register in the RF stage.

One solution is to stall the IF stage until the RF stage can compute the necessary result.

This was the first of our general strategies for dealing with hazards.

How would this work?

If the opcode in the RF stage is JMP, BEQ, or BNE, stall the IF stage for one cycle.

In the example code shown here, assume that the value in R3 is non-zero when the BNE is executed, i.e., that the instruction following BNE should be the ADDC at the top of the loop.

The pipeline diagram shows the effect we're trying to achieve: a NOP is inserted into the pipeline in cycles 4 and 8.

Then execution resumes in the next cycle after the RF stage determines what instruction comes next.

Note, by the way, that we're relying on our bypass logic to deliver the correct value for R3 from the MEM stage since the ADDC instruction that wrote into R3 is still in the pipeline, i.e., we have a data hazard to deal with too!

Looking at, say, the WB stage in the pipeline diagram, we see it takes 4 cycles to execute one iteration of our 3-instruction loop.

So the effective CPI is 4/3, an increase of 33%.

Using stall to deal with control hazards has had an impact on the instruction throughput of our execution pipeline.

We've already seen the logic needed to introduce NOPs into the pipeline.

In this case, we add a mux to the instruction path in the IF stage, controlled by the IRSrc_IF signal.

We use the superscript on the control signals to indicate which pipeline stage holds the logic they control.

If the opcode in the RF stage is JMP, BEQ, or BNE we set IRSrc_IF to 1, which causes a NOP to replace the instruction that was being read from main memory.

And, of course, we'll be setting the PCSEL control signals to select the correct next PC value, so the IF stage will

fetch the desired follow-on instruction in the next cycle.

If we replace an instruction with NOP, we say we “annulled” the instruction.

The branch instructions in the Beta ISA make their branch decision in the RF stage since they only need the value in register RA.

But suppose the ISA had a branch where the branch decision was made in ALU stage.

When the branch decision is made in the ALU stage, we need to introduce two NOPs into the pipeline, replacing the now unwanted instructions in the RF and IF stages.

This would increase the effective CPI even further.

But the tradeoff is that the more complex branches may reduce the number of instructions in the program.

If we annul instructions in all the earlier pipeline stages, this is called “flushing the pipeline”.

Since flushing the pipeline has a big impact on the effective CPI, we do it when it’s the only way to ensure the correct behavior of the execution pipeline.

We can be smarter about when we choose to flush the pipeline when executing branches.

If the branch is not taken, it turns out that the pipeline has been doing the right thing by fetching the instruction following the branch.

Starting execution of an instruction even when we’re unsure whether we really want it executed is called “speculation”.

Speculative execution is okay if we’re able to annul the instruction before it has an effect on the CPU state, e.g., by writing into the register file or main memory.

Since these state changes (called “side effects”) happen in the later pipeline stages, an instruction can progress through the IF, RF, and ALU stages before we have to make a final decision about whether it should be annulled.

How does speculation help with control hazards?

Guessing that the next value of the program counter is PC+4 is correct for all but JMPs and taken branches.

Here’s our example again, but this time let’s assume that the BNE is not taken, i.e., that the value in R3 is zero.

The SUB instruction enters the pipeline at the start of cycle 4.

At the end of cycle 4, we know whether or not to annul the SUB.

If the branch is not taken, we want to execute the SUB instruction, so we just let it continue down the pipeline.

In other words, instead of always annulling the instruction following branch, we only annul it if the branch was taken.

If the branch is not taken, the pipeline has speculated correctly and no instructions need to be annulled.

However if the BNE is taken, the SUB is annulled at the end of cycle 4 and a NOP is executed in cycle 5.

So we only introduce a bubble in the pipeline when there's a taken branch.

Fewer bubbles will decrease the impact of annulment on the effective CPI.

We'll be using the same data path circuitry as before, we'll just be a bit more clever about when the value of the IRSrc_IF control signal is set to 1.

Instead of setting it to 1 for all branches, we only set it to 1 when the branch is taken.

Our naive strategy of always speculating that the next instruction comes from PC+4 is wrong for JMPs and taken branches.

Looking at simulated execution traces, we'll see that this error in speculation leads to about 10% higher effective CPI.

Can we do better?

This is an important question for CPUs with deep pipelines.

For example, Intel's Nehalem processor from 2009 resolves the more complex x86 branch instructions quite late in the pipeline.

Since Nehalem is capable of executing multiple instructions each cycle, flushing the pipeline in Nehalem actually annuls the execution of many instructions, resulting in a considerable hit on the CPI.

Like many modern processor implementations, Nehalem has a much more sophisticated speculation mechanism.

Rather than always guessing the next instruction is at PC+4, it only does that for non-branch instructions.

For branches, it predicts the behavior of each individual branch based on what the branch did last time it was executed and some knowledge of how the branch is being used.

For example, backward branches at the end of loops, which are taken for all but the final iteration of the loop, can be identified by their negative branch offset values.

Nehalem can even determine if there's correlation between branch instructions, using the results of an another, earlier branch to speculate on the branch decision of the current branch.

With these sophisticated strategies, Nehalem's speculation is correct 95% to 99% of the time, greatly reducing the impact of branches on the effective CPI.

There's also the lazy option of changing the ISA to deal with control hazards.

For example, we could change the ISA to specify that the instruction following a jump or branch is always executed.

In other words the transfer of control happens *after* the next instruction.

This change ensures that the guess of PC+4 as the address of the next instruction is always correct!

In the example shown here, assuming we changed the ISA, we can reorganize the execution order of the loop to place the MUL instruction after the BNE instruction, in the so-called "branch delay slot".

Since the instruction in the branch delay slot is always executed, the MUL instruction will be executed during each iteration of the loop.

The resulting execution is shown in this pipeline diagram.

Assuming we can find an appropriate instruction to place in the delay slot, the branch will have zero impact on the effective CPI.

Are branch delay slots a good idea?

Seems like they reduce the negative impact that branches might have on instruction throughput.

The downside is that only half the time can we find instructions to move to the branch delay slot.

The other half of the time we have to fill it with an explicit NOP instruction, increasing the size of the code.

And if we make the branch decision later in the pipeline, there are more branch delay slots, which would be even

harder to fill.

In practice, it turns out that branch prediction works better than delay slots in reducing the impact of branches.

So, once again we see that it’s problematic to alter the ISA to improve the throughput of pipelined execution.

ISAs outlive implementations, so it’s best not to change the execution semantics to deal with performance issues created by a particular implementation.