Now let's figure out how exceptions impact pipelined execution.

When an exception occurs because of an illegal instruction or an external interrupt, we need to store the current PC+4 value in the XP register and load the program counter with the address of the appropriate exception handler.

Exceptions cause control flow hazards since they are effectively implicit branches.

In an unpipelined implementation, exceptions affect the execution of the current instruction.

We want to achieve exactly the same effect in our pipelined implementation.

So first we have to identify which one of the instructions in our pipeline is affected, then ensure that instructions that came earlier in the code complete correctly and that we annul the affected instruction and any following instructions that are in the pipeline.

Since there are multiple instructions in the pipeline, we have a bit of sorting out to do.

When, during pipelined execution, do we determine that an instruction will cause an exception?

An obvious example is detecting an illegal opcode when we decode the instruction in the RF stage.

But we can also generate exceptions in other pipeline stages.

For example, the ALU stage can generate an exception if the second operand of a DIV instruction is 0.

Or the MEM stage may detect that the instruction is attempting to access memory with an illegal address.

Similarly the IF stage can generate a memory exception when fetching the next instruction.

In each case, instructions that follow the one that caused the exception may already be in the pipeline and will need to be annulled.

The good news is that since register values are only updated in the WB stage, annulling an instruction only requires replacing it with a NOP.

We won't have to restore any changed values in the register file or main memory.

Here's our plan.

If an instruction causes an exception in stage i, replace that instruction with this BNE instruction, whose only side effect is writing the PC+4 value into the XP register.

Then flush the pipeline by annulling instructions in earlier pipeline stages.

And, finally, load the program counter with the address of the exception handler.

In this example, assume that LD will generate a memory exception in the MEM stage, which occurs in cycle 4.

The arrows show how the instructions in the pipeline are rewritten for cycle 5, at which point the IF stage is working on fetching the first instruction in the exception handler.

Here are the changes required to the execution pipeline.

We modify the muxes in the instruction path so that they can replace an actual instruction with either NOP if the instruction is to be annulled, or BNE if the instruction caused the exception.

Since the pipeline is executing multiple instructions at the same time, we have to worry about what happens if multiple exceptions are detected during execution.

In this example assume that LD will cause a memory exception in the MEM stage and note that it is followed by an instruction with an illegal opcode.

Looking at the pipeline diagram, the invalid opcode is detected in the RF stage during cycle 3, causing the illegal instruction exception process to begin in cycle 4.

But during that cycle, the MEM stage detects the illegal memory access from the LD instruction and so causes the memory exception process to begin in cycle 5.

Note that the exception caused by the earlier instruction (LD) overrides the exception caused by the later illegal opcode even though the illegal opcode exception was detected first.

.348 That's the correct behavior since once the execution of LD is abandoned, the pipeline should behave as if none of the instructions that come after the LD were executed.

If multiple exceptions are detected in the *same* cycle, the exception from the instruction furthest down the pipeline should be given precedence.

External interrupts also behave as implicit branches, but it turns out they are a bit easier to handle in our pipeline.

We'll treat external interrupts as if they were an exception that affected the IF stage.

Let's assume the external interrupt occurs in cycle 2.

This means that the SUB instruction will be replaced by our magic BNE to capture the PC+4 value and we'll force the next PC to be the address of the interrupt handler.

After the interrupt handler completes, we'll want to resume execution of the interrupted program at the SUB instruction, so we'll code the handler to correct the value saved in the XP register so that it points to the SUB instruction.

This is all shown in the pipeline diagram.

Note that the ADD, LD, and other instructions that came before SUB in the program are unaffected by the interrupt.

We can use the existing instruction-path muxes to deal with interrupts, since we're treating them as IF-stage exceptions.

We simply have to adjust the logic for IRSrc_IF to also make it 1 when an interrupt is requested.