The data path diagram isn't all that useful in diagramming the pipelined execution of an instruction sequence since we need a new copy of the diagram for each clock cycle.

A more compact and easier-to-read diagram of pipelined execution is provided by the pipeline diagrams we met back in Part 1 of the course.

There's one row in the diagram for each pipeline stage and one column for each cycle of execution.

Entries in the table show which instruction is in each pipeline stage at each cycle.

In normal operation, a particular instruction moves diagonally through the diagram as it proceeds through the five pipeline stages.

To understand data hazards, let's first remind ourselves of when the register file is read and written for a particular instruction.

Register reads happen when the instruction is in the RF stage, i.e., when we're reading the instruction's register operands.

Register writes happen at the end of the cycle when the instruction is in the WB stage.

For example, for the first LD instruction, we read R1 during cycle 2 and write R2 at the end of cycle 5.

Or consider the register file operations in cycle 6: we're reading R12 and R13 for the MUL instruction in the RF stage, and writing R4 at the end of the cycle for the LD instruction in the WB stage.

Okay, now let's see what happens when there are data hazards.

In this instruction sequence, the ADDC instruction writes its result in R2, which is immediately read by the following SUBC instruction.

Correct execution of the SUBC instruction clearly depends on the results of the ADDC instruction.

This what we'd call a read-after-write dependency.

This pipeline diagram shows the cycle-by-cycle execution where we've circled the cycles during which ADDC writes R2 and SUBC reads R2.

Oops!

ADDC doesn't write R2 until the end of cycle 5, but SUBC is trying to read the R2 value in cycle 3.

The value in R2 in the register file in cycle 3 doesn't yet reflect the execution of the ADDC instruction.

So as things stand the pipeline would *not* correctly execute this instruction sequence.

This instruction sequence has triggered a data hazard.

We want the pipelined CPU to generate the same program results as the unpipelined CPU, so we'll need to figure out a fix.

There are three general strategies we can pursue to fix pipeline hazards.

Any of the techniques will work, but as we'll see they have different tradeoffs for instruction throughput and circuit complexity.

The first strategy is to stall instructions in the RF stage until the result they need has been written to the register file.

"Stall" means that we don't reload the instruction register at the end of the cycle, so we'll try to execute the same instruction in the next cycle.

If we stall one pipeline stage, all earlier stages must also be stalled since they are blocked by the stalled instruction.

If an instruction is stalled in the RF stage, then the IF stage is also stalled.

Stalling will always work, but has a negative impact on instruction throughput.

Stall for too many cycles and you'll loose the performance advantages of pipelined execution!

The second strategy is to route the needed value to earlier pipeline stages as soon as its computed.

This called bypassing or forwarding.

As it turns out, the value we need often exists somewhere in the pipelined data path, it just hasn't been written yet to the register file.

If the value exists and can be forwarded to where it's needed, we won't need to stall.

We'll be able to use this strategy to avoid stalling on most types of data hazards.

The third strategy is called speculation.

We'll make an intelligent guess for the needed value and continue execution.

Once the actual value is determined, if we guessed correctly, we're all set.

If we guessed incorrectly, we have to back up execution and restart with the correct value.

Obviously speculation only makes sense if it's possible to make accurate guesses.

We'll be able to use this strategy to avoid stalling on control hazards.

Let's see how the first two strategies work when dealing with our data hazard.

Applying the stall strategy to our data hazard, we need to stall the SUBC instruction in the RF stage until the ADDC instruction writes its result in R2.

So in the pipeline diagram, SUBC is stalled three times in the RF stage until it can finally access the R2 value from the register file in cycle 6.

Whenever the RF stage is stalled, the IF stage is also stalled.

You can see that in the diagram too.

But when RF is stalled, what should the ALU stage do in the next cycle?

The RF stage hasn't finished its job and so can't pass along its instruction!

The solution is for the RF stage to make-up an innocuous instruction for the ALU stage, what's called a NOP instruction, short for "no operation".

A NOP instruction has no effect on the CPU state, i.e., it doesn't change the contents of the register file or main memory.

For example any OP-class or OPC-class instruction that has R31 as its destination register is a NOP.

The NOPs introduced into the pipeline by the stalled RF stage are shown in red.

Since the SUBC is stalled in the RF stage for three cycles, three NOPs are introduced into the pipeline.

We sometimes refer to these NOPs as "bubbles" in the pipeline.

How does the pipeline know when to stall?

It can compare the register numbers in the RA and RB fields of the instruction in the RF stage with the register numbers in the RC field of instructions in the ALU, MEM, and WB stage.

If there's a match, there's a data hazard and the RF stage should be stalled.

The stall will continue until there's no hazard detected.

There are a few details to take care of: some instructions don't read both registers, the ST instruction doesn't use its RC field, and we don't want R31 to match since it's always okay to read R31 from the register file.

Stalling will ensure correct pipelined execution, but it does increase the effective CPI.

This will lead to longer execution times if the increase in CPI is larger than the decrease in cycle time afforded by pipelining.

To implement stalling, we only need to make two simple changes to our pipelined data path.

We generate a new control signal, STALL, which, when asserted, disables the loading of the three pipeline registers at the input of the IF and RF stages, which means they'll have the same value next cycle as they do this cycle.

We also introduce a mux to choose the instruction to be sent along to the ALU stage.

If STALL is 1, we choose a NOP instruction, e.g., an ADD with R31 as its destination.

If STALL is 0, the RF stage is not stalled, so we pass its current instruction to the ALU.

And here we see how to compute STALL as described in the previous slide.

The additional logic needed to implement stalling is pretty modest, so the real design tradeoff is about increased CPI due to stalling vs. decreased cycle time due to pipelining.

So we have a solution, although it carries some potential performance costs.

Now let's consider our second strategy: bypassing, which is applicable if the data we need in the RF stage is somewhere in the pipelined data path.

In our example, even though ADDC doesn't write R2 until the end of cycle 5, the value that will be written is computed during cycle 3 when the ADDC is in the ALU stage.

In cycle 3, the output of the ALU is the value needed by the SUBC that's in the RF stage in the same cycle.

So, if we detect that the RA field of the instruction in the RF stage is the same as the RC field of the instruction in the ALU stage, we can use the output of the ALU in place of the (stale) RA value being read from the register file.

No stalling necessary!

In our example, in cycle 3 we want to route the output of the ALU to the RF stage to be used as the value for R2.

We show this with a red "bypass arrow" showing data being routed from the ALU stage to the RF stage.

To implement bypassing, we'll add a many-input multiplexer to the read ports of the register file so we can select the appropriate value from other pipeline stages.

Here we show the combinational bypass paths from the ALU, MEM, and WB stages.

For the bypassing example of the previous slides, we use the blue bypass path during cycle 3 to get the correct value for R2.

The bypass muxes are controlled by logic that's matching the number of the source register to the number of the destination registers in the ALU, MEM, and WB stages, with the usual complications of dealing with R31.

What if there are multiple matches, i.e., if the RF stage is trying to read a register that's the destination for, say, the instructions in both the ALU and MEM stages?

No problem!

We want to select the result from the most recent instruction, so we'd chose the ALU match if there is one, then the MEM match, then the WB match, then, finally, the output of the register file.

Here's diagram showing all the bypass paths we'll need.

Note that branches and jumps write their PC+4 value into the register file, so we'll need to bypass from the PC+4 values in the various stages as well as the ALU values.

Note that the bypassing is happening at the end of the cycle, e.g., after the ALU has computed its answer.

To accommodate the extra t_PD of the bypass mux, we'll have to extend the clock period by a small amount.

So once again there's a design tradeoff - the increased CPI of stalling vs the slightly increased cycle time of bypassing.

And, of course, in the case of bypassing there's the extra area needed for the necessary wiring and muxes.

We can cut back on the costs by reducing the amount of bypassing, say, to only bypassing ALU results from the ALU stage and use stalling to deal with all the other data hazards.

If we implement full bypassing, do we still need the STALL logic?

As it turns out, we do!

There's one data hazard that bypassing doesn't completely address.

Consider trying to immediately the use the result of a LD instruction.

In the example shown here, the SUBC is trying to use the value the immediately preceding LD is writing to R2.

This is called a load-to-use hazard.

Recalling that LD data isn't available in the data path until the cycle when LD reaches the WB stage, even with full bypassing we'll need to stall SUBC in the RF stage until cycle 5, introducing two NOPs into the pipeline.

Without bypassing from the WB stage, we need to stall until cycle 6.

In summary, we have two strategies for dealing with data hazards.

We can stall the IF and RF stages until the register values needed by the instruction in the RF stage are available in the register file.

The required hardware is simple, but the NOPs introduced into the pipeline waste CPU cycles and result in an higher effective CPI.

Or we can use bypass paths to route the required values to the RF stage assuming they exist somewhere in the pipelined data path.

This approach requires more hardware than stalling, but doesn't reduce the effective CPI.

Even if we implement bypassing, we'll still need stalls to deal with load-to-use hazards.

Can we keep adding pipeline stages in the hopes of further reducing the clock period?

More pipeline stages mean more instructions in the pipeline at the same time, which in turn increases the chance of a data hazard and the necessity of stalling, thus increasing CPI.

Compilers can help reduce dependencies by reorganizing the assembly language code they produce.

Here's the load-to-use hazard example we saw earlier.

Even with full bypassing, we'd need to stall for 2 cycles.

But if the compiler (or assembly language programmer!) notices that the MUL and XOR instructions are independent of the SUBC instruction and hence can be moved before the SUBC, the dependency is now such that the LD is naturally in the WB stage when the SUBC is in the RF stage, so no stalls are needed.

This optimization only works when the compiler can find independent instructions to move around.

Unfortunately there are plenty of programs where such instructions are hard to find.

Then there's one final approach we could take - change the ISA so that data hazards are part of the ISA, i.e., just explain that writes to the destination register happen with a 3-instruction delay!

If NOPs are needed, make the programmer add them to the program.

Simplify the hardware at the "small" cost of making the compilers work harder.

You can imagine exactly how much the compiler writers will like this suggestion.

Not to mention assembly language programmers!

And you can change the ISA again when you add more pipeline stages!

This is how a compiler writer views CPU architects who unilaterally change the ISA to save a few logic gates :) The bottom line is that successful ISAs have very long lifetimes and so shouldn't include tradeoffs driven by short-term implementation considerations.

Best not to go there.